Giáo trình
Cấu trúc dữ liệu và giải thuật

MỤC LỤC

Mục **Trang**

CHƯƠNG 1: TỔNG QUAN VỀ CẤU TRÚC DỮ LIỆU & GT 3

1.1. Tầm quan trọng của CTDL & GT trong một đề án tin học 3

1.1.1. Xây dựng cấu trúc dữ liệu *3*

1.1.2. Xây dựng giải thuật *3*

1.1.3. Mối quan hệ giữa cấu trúc dữ liệu và giải thuật *3*

1.2. Đánh giá Cấu trúc dữ liệu & Giải thuật 3

1.2.1. Các tiêu chuẩn đánh giá cấu trúc dữ liệu *3*

1.2.2. Đánh giá độ phức tạp của thuật toán *4*

1.3. Kiểu dữ liệu 4

1.3.1. Khái niệm về kiểu dữ liệu *4*

1.3.2. Các kiểu dữ liệu cơ sở *4*

1.3.3. Các kiểu dữ liệu có cấu trúc *5*

1.3.4. Kiểu dữ liệu con trỏ *5*

1.3.5. Kiểu dữ liệu tập tin *5*

Câu hỏi và bài tập 6

CHƯƠNG 2: KỸ THUẬT TÌM KIẾM (Searching) 8

2.1. Khái quát về tìm kiếm 8

2.2. Các giải thuật tìm kiếm nội 8

2.2.1. Đặt vấn đề *8*

2.2.2. Tìm tuyến tính *8*

2.2.3. Tìm nhị phân *10*

2.3. Các giải thuật tìm kiếm ngoại 14

2.3.1. Đặt vấn đề *14*

2.3.2. Tìm tuyến tính *14*

2.3.3. Tìm kiếm theo chỉ mục *16*

Câu hỏi và bài tập 17

CHƯƠNG 3: KỸ THUẬT SẮP XẾP (SORTING) 19

3.1. Khái quát về sắp xếp 19

3.2. Các giải thuật sắp xếp nội 19

3.2.1 Sắp xếp bằng phương pháp đổi chỗ *20*

3.2.2. Sắp xếp bằng phương pháp chọn *28*

3.2.3. Sắp xếp bằng phương pháp chèn *33*

3.2.4. Sắp xếp bằng phương pháp trộn *40*

3.3. Các giải thuật sắp xếp ngoại 60

3.3.1. Sắp xếp bằng phương pháp trộn *60*

3.3.2. Sắp xếp theo chỉ mục *79*

Câu hỏi và bài tập 82

CHƯƠNG 4: DANH SÁCH (LIST) 84

4.1. Khái niệm về danh sách 84
4.2. Các phép toán trên danh sách 84
4.3. Danh sách đặc 85
4.3.1. Định nghĩa *85*
4.3.2. Biểu diễn danh sách đặc *85*
4.3.3. Các thao tác trên danh sách đặc *85*
4.3.4. Ưu nhược điểm và Ứng dụng *91*
4.4. Danh sách liên kết 92
4.4.1. Định nghĩa *92*
4.4.2. Danh sách liên kết đơn *92*
4.4.3. Danh sách liên kết kép *111*
4.4.4. Ưu nhược điểm của danh sách liên kết *135*
4.5. Danh sách hạn chế 135
4.5.1. Hàng đợi *135*
4.5.2. Ngăn xếp *142*
4.5.3. Ứng dụng của danh sách hạn chế *147*
Câu hỏi và bài tập 147

CHƯƠNG 5: CÂY (TREE) 149

5.1. Khái niệm – Biểu diễn cây 149
5.1.1. Định nghĩa cây *149*
5.1.2. Một số khái niệm liên quan *149*
5.1.3. Biểu diễn cây *151*
5.2. Cây nhị phân 152
5.2.1. Định nghĩa *152*
5.2.2. Biểu diễn và Các thao tác *152*
5.2.3. Cây nhị phân tìm kiếm *163*
5.3. Cây cân bằng 188
5.3.1. Định nghĩa – Cấu trúc dữ liệu *188*
5.3.2. Các thao tác *189*
Câu hỏi và bài tập 227

ÔN TẬP (REVIEW) 224

Hệ thống lại các Cấu trúc dữ liệu và các Giải thuật đã học 224
Câu hỏi và Bài tập ôn tập tổng hợp 227

TÀI LIỆU THAM KHẢO 229

Chương 1: TỔNG QUAN VỀ CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

1.1. Tầm quan trọng của cấu trúc dữ liệu và giải thuật trong một đề án tin học

1.1.1. Xây dựng cấu trúc dữ liệu

Có thể nói rằng không có một chương trình máy tính nào mà không có dữ liệu để xử lý. Dữ liệu có thể là dữ liệu đưa vào (input data), dữ liệu trung gian hoặc dữ liệu đưa ra (output data). Do vậy, việc tổ chức để lưu trữ dữ liệu phục vụ cho chương trình có ý nghĩa rất quan trọng trong toàn bộ hệ thống chương trình. Việc xây dựng cấu trúc dữ liệu quyết định rất lớn đến chất lượng cũng như công sức của người lập trình trong việc thiết kế, cài đặt chương trình.

1.1.2. Xây dựng giải thuật

Khái niệm giải thuật hay thuật giải mà nhiều khi còn được gọi là thuật toán dùng để chỉ phương pháp hay cách thức (method) để giải quyết vần đề. Giải thuật có thể được minh họa bằng ngôn ngữ tự nhiên (natural language), bằng sơ đồ (flow chart) hoặc bằng mã giả (pseudo code). Trong thực tế, giải thuật thường được minh họa hay thể hiện bằng mã giả tựa trên một hay một số ngôn ngữ lập trình nào đó (thường là ngôn ngữ mà người lập trình chọn để cài đặt thuật toán), chẳng hạn như C, Pascal, ...

Khi đã xác định được cấu trúc dữ liệu thích hợp, người lập trình sẽ bắt đầu tiến hành xây dựng thuật giải tương ứng theo yêu cầu của bài toán đặt ra trên cơ sở của cấu trúc dữ liệu đã được chọn. Để giải quyết một vấn đề có thể có nhiều phương pháp, do vậy sự lựa chọn phương pháp phù hợp là một việc mà người lập trình phải cân nhắc và tính toán. Sự lựa chọn này cũng có thể góp phần đáng kể trong việc giảm bớt công việc của người lập trình trong phần cài đặt thuật toán trên một ngôn ngữ cụ thể.

1.1.3. Mối quan hệ giữa cấu trúc dữ liệu và giải thuật

Mối quan hệ giữa cấu trúc dữ liệu và Giải thuật có thể minh họa bằng đẳng thức:

Cấu trúc dữ liệu + Giải thuật = Chương trình

Như vậy, khi đã có cấu trúc dữ liệu tốt, nắm vững giải thuật thực hiện thì việc thể hiện chương trình bằng một ngôn ngữ cụ thể chỉ là vấn đề thời gian. Khi có cấu trúc dữ liệu mà chưa tìm ra thuật giải thì không thể có chương trình và ngược lại không thể có Thuật giải khi chưa có cấu trúc dữ liệu. Một chương trình máy tính chỉ có thể được hoàn thiện khi có đầy đủ cả Cấu trúc dữ liệu để lưu trữ dữ liệu và Giải thuật xử lý dữ liệu theo yêu cầu của bài toán đặt ra.

1.2. Đánh giá cấu trúc dữ liệu và giải thuật

1.2.1. Các tiêu chuẩn đánh giá cấu trúc dữ liệu

Để đánh giá một cấu trúc dữ liệu chúng ta thường dựa vào một số tiêu chí sau:

- Cấu trúc dữ liệu phải tiết kiệm tài nguyên (bộ nhớ trong),

- Cấu trúc dữ liệu phải phản ảnh đúng thực tế của bài toán,
- Cấu trúc dữ liệu phải dễ dàng trong việc thao tác dữ liệu.

1.2.2. Đánh giá độ phức tạp của thuật toán

Việc đánh giá độ phức tạp của một thuật toán quả không dễ dàng chút nào. Ở dây, chúng ta chỉ muốn ước lượng thời gian thực hiện thuận toán T(n) để có thể có sự so sánh tương đối giữa các thuật toán với nhau. Trong thực tế, thời gian thực hiện một thuật toán còn phụ thuộc rất nhiều vào các điều kiện khác như cấu tạo của máy tính, dữ liệu đưa vào, ..., ở đây chúng ta chỉ xem xét trên mức độ của lượng dữ liệu đưa vào ban đầu cho thuật toán thực hiện.

Để ước lượng thời gian thực hiện thuật toán chúng ta có thể xem xét thời gian thực hiện thuật toán trong hai trường hợp:

- Trong trường hợp tốt nhất: Tmin
- Trong trường hợp xấu nhất: Tmax

Từ đó chúng ta có thể ước lượng thời gian thực hiện trung bình của thuật toán: Tavg

1.3. Kiểu dữ liệu

1.3.1. Khái niệm về kiểu dữ liệu

Kiểu dữ liệu T có thể xem như là sự kết hợp của 2 thành phần:

- Miền giá trị mà kiểu dữ liệu T có thể lưu trữ: V,
- Tập hợp các phép toán để thao tác dữ liệu: O.

T = <V, O>

Mỗi kiểu dữ liệu thường được đại diện bởi một tên (định danh). Mỗi phần tử dữ liệu có kiểu T sẽ có giá trị trong miền V và có thể được thực hiện các phép toán thuộc tập hợp các phép toán trong O.

Để lưu trữ các phần tử dữ liệu này thường phải tốn một số byte(s) trong bộ nhớ, số byte(s) này gọi là kích thước của kiểu dữ liệu.

1.3.2. Các kiểu dữ liệu cơ sở

Hầu hết các ngôn ngữ lập trình đều có cung cấp các kiểu dữ liệu cơ sở. Tùy vào mỗi ngôn ngữ mà các kiểu dữ liệu cơ sở có thể có các tên gọi khác nhau song chung quy lại có những loại kiểu dữ liệu cơ sở như sau:

- Kiểu số nguyên: Có thể có dấu hoặc không có dấu và thường có các kích thước sau:
  + Kiểu số nguyên 1 byte
  + Kiểu số nguyên 2 bytes
  + Kiểu số nguyên 4 bytes

Kiểu số nguyên thường được thực hiện với các phép toán: O = {+, -, *, /, DIV, MOD, <, >, <=, >=, =, ...}

- Kiểu số thực: Thường có các kích thước sau:

+ Kiểu số thực 4 bytes

+ Kiểu số thực 6 bytes

+ Kiểu số thực 8 bytes

+ Kiểu số thực 10 bytes

Kiểu số thực thường được thực hiện với các phép toán: O = {+, -, *, /, <, >, <=, >=, =, ...}

- Kiểu ký tự: Có thể có các kích thước sau:

+ Kiểu ký tự byte

+ Kiểu ký tự 2 bytes

Kiểu ký tự thường được thực hiện với các phép toán: O = {+, -, <, >, <=, >=, =, ORD, CHR, ...}

- Kiểu chuỗi ký tự: Có kích thước tùy thuộc vào từng ngôn ngữ lập trình

Kiểu chuỗi ký tự thường được thực hiện với các phép toán: O = {+, &, <, >, <=, >=, =, Length, Trunc, ...}

- Kiểu luận lý: Thường có kích thước 1 byte

Kiểu luận lý thường được thực hiện với các phép toán: O = {NOT, AND, OR, XOR, <, >, <=, >=, =, ...}

1.3.3. Các kiểu dữ liệu có cấu trúc

Kiểu dữ liệu có cấu trúc là các kiểu dữ liệu được xây dựng trên cơ sở các kiểu dữ liệu đã có (có thể lại là một kiểu dữ liệu có cấu trúc khác). Tùy vào từng ngôn ngữ lập trình song thường có các loại sau:

- Kiểu mảng hay còn gọi là dãy: kích thước bằng tổng kích thước của các phần tử

- Kiểu bản ghi hay cấu trúc: kích thước bằng tổng kích thước các thành phần (Field)

1.3.4. Kiểu dữ liệu con trỏ

Các ngôn ngữ lập trình thường cung cấp cho chúng ta một kiểu dữ liệu đặc biệt để lưu trữ các địa chỉ của bộ nhớ, đó là con trỏ (Pointer). Tùy vào loại con trỏ gần (near pointer) hay con trỏ xa (far pointer) mà kiểu dữ liệu con trỏ có các kích thước khác nhau:

+ Con trỏ gần: 2 bytes

+ Con trỏ xa: 4 bytes

1.3.5. Kiểu dữ liệu tập tin

Tập tin (File) có thể xem là một kiểu dữ liệu đặc biệt, kích thước tối đa của tập tin tùy thuộc vào không gian đĩa nơi lưu trữ tập tin. Việc đọc, ghi dữ liệu trực tiếp trên tập tin rất mất thời gian và không bảo đảm an toàn cho dữ liệu trên tập tin đó. Do vậy, trong thực tế, chúng ta không thao tác trực tiếp dữ liệu trên tập tin mà chúng ta cần chuyển từng phần hoặc toàn bộ nội dung của tập tin vào trong bộ nhớ trong để xử lý.

Câu hỏi và Bài tập

1. Trình bày tầm quan trọng của Cấu trúc dữ liệu và Giải thuật đối với người lập trình?

2. Các tiêu chuẩn để đánh giá cấu trúc dữ liệu và giải thuật?

3. Khi xây dựng giải thuật có cần thiết phải quan tâm tới cấu trúc dữ liệu hay không? Tại sao?

4. Liệt kê các kiểu dữ liệu cơ sở, các kiểu dữ liệu có cấu trúc trong C, Pascal?

5. Sử dụng các kiểu dữ liệu cơ bản trong C, hãy xây dựng cấu trúc dữ liệu để lưu trữ trong bộ nhớ trong *(RAM)* của máy tính đa thức có bậc tự nhiên n ($0 \le n \le 100$) trên trường số thực *(a_i, $x \in R$)*:

$$fn(x) = \sum_{i=0}^{n} a_i x^i$$

Với cấu trúc dữ liệu được xây dựng, hãy trình bày thuật toán và cài đặt chương trình để thực hiện các công việc sau:

- Nhập, xuất các đa thức.
- Tính giá trị của đa thức tại giá trị x_0 nào đó.
- Tính tổng, tích của hai đa thức.

6. Tương tự như bài tập 5. nhưng đa thức trong trường số hữu tỷ Q (các hệ số a_i và x là các phân số có tử số và mẫu số là các số nguyên).

7. Cho bảng giờ tàu đi từ ga Saigon đến các ga như sau (ga cuối là ga Hà nội):

TÀU ĐI	S2	S4	S6	S8	S10	S12	S14	S16	S18	LH2	SN2
HÀNH TRÌNH	32 giờ	41 giờ	41 giờ	41 giờ	41 giờ	41 giờ	41 giờ	41 giờ	41 giờ	27giờ	10g30
SAIGON ĐI	***21g00***	***21g50***	***11g10***	***15g40***	***10g00***	***12g30***	***17g00***	***20g00***	***22g20***	***13g20***	***18g40***
MƯƠNG MÁN		2g10	15g21	19g53	14g07	16g41	21g04	1g15	3g16	17g35	22g58
THÁP CHÀM		5g01	18g06	22g47	16g43	19g19	0g08	4g05	6g03	20g19	2g15
NHA TRANG	***4g10***	***6g47***	***20g00***	***0g47***	***18g50***	***21g10***	***1g57***	***5g42***	***8g06***	***22g46***	***5g15***
TUY HÒA		9g43	23g09	3g39	21g53	0g19	5g11	8g36	10g50	2g10	
DIÊU TRÌ	***8g12***	***11g49***	***1g20***	***5g46***	***0g00***	***2g30***	***7g09***	***10g42***	***13g00***	***4g15***	
QUẢNG NGÃI		15g41	4g55	9g24	3g24	5g55	11g21	14g35	17g04	7g34	
TAM KỲ			6g11	10g39	4g38	7g10	12g40	16g08	18g21	9g03	
ĐÀ NẴNG	***13g27***	***19g04***	***8g29***	***12g20***	***6g19***	***9g26***	***14g41***	***17g43***	***20g17***	***10g53***	
HUẾ	***16g21***	***22g42***	***12g29***	***15g47***	***11g12***	***14g32***	***18g13***	***21g14***	***23g50***	***15g10***	
ĐÔNG HÀ		0g14	13g52	17g12	12g42	16g05	19g38	22g39	1g25		
ĐỒNG HỚI	19g15	2g27	15g52	19g46	14g41	17g59	21g38	0g52	3g28		
VINH	***23g21***	***7g45***	***21g00***	***1g08***	***20g12***	***23g50***	***2g59***	***7g07***	***9g20***		
THANH HÓA		10g44	0g01	4g33	23g09	3g33	6g39	9g59	12g20		
NINH BÌNH		12g04	1g28	5g54	0g31	4g50	7g57	11g12	13g51		
NAM ĐỊNH		12g37	2g01	6g26	1g24	5g22	8g29	11g44	14g25		
PHỦ LÝ		13g23	2g42	7g08	2g02	6g00	9g09	12g23	15g06		
ĐẾN HÀ NỘI	***5g00***	***14g40***	***4g00***	***8g30***	***3g15***	***7g10***	***10g25***	***13g45***	***16g20***		

Sử dụng các kiểu dữ liệu cơ bản, hãy xây dựng cấu trúc dữ liệu thích hợp để lưu trữ bảng giờ tàu trên vào bộ nhớ trong và bộ nhớ ngoài (disk) của máy tính.

Với cấu trúc dữ liệu đã được xây dựng ở trên, hãy trình bày thuật toán và cài đặt chương trình để thực hiện các công việc sau:

- Xuất ra giờ đến của một tàu T_0 nào đó tại một ga G_0 nào đó.

- Xuất ra giờ đến các ga của một tàu T_0 nào đó.
- Xuất ra giờ các tàu đến một ga G_0 nào đó.
- Xuất ra bảng giờ tàu theo mẫu ở trên.

Lưu ý:

- Các ô trống ghi nhận tại các ga đó, tàu này không đi đến hoặc chỉ đi qua mà không dừng lại.
- Dòng "HÀNH TRÌNH" ghi nhận tổng số giờ tàu chạy từ ga Saigon đến ga Hà nội.

8. Tương tự như bài tập 7. nhưng chúng ta cần ghi nhận thêm thông tin về đoàn tàu khi dừng tại các ga chỉ để tránh tàu hay để cho khách lên/xuống *(các dòng in nghiêng tương ứng với các ga có khách lên/xuống, các dòng khác chỉ dừng để tránh tàu)*.

9. Sử dụng kiểu dữ liệu cấu trúc trong C, hãy xây dựng cấu trúc dữ liệu để lưu trữ trong bộ nhớ trong (RAM) của máy tính trạng thái của các cột đèn giao thông (có 3 đèn: Xanh, Đỏ, Vàng). Với cấu trúc dữ liệu đã được xây dựng, hãy trình bày thuật toán và cài đặt chương trình để mô phỏng (minh họa) cho hoạt động của 2 cột đèn trên hai tuyến đường giao nhau tại một ngã tư.

10. Sử dụng các kiểu dữ liệu cơ bản trong C, hãy xây dựng cấu trúc dữ liệu để lưu trữ trong bộ nhớ trong *(RAM)* của máy tính trạng thái của một bàn cờ CARO có kích thước $M \times N$ ($0 \le M, N \le 20$). Với cấu trúc dữ liệu được xây dựng, hãy trình bày thuật toán và cài đặt chương trình để thực hiện các công việc sau:

- In ra màn hình bàn cờ CARO trong trạng thái hiện hành.
- Kiểm tra xem có ai thắng hay không? Nếu có thì thông báo "Kết thúc", nếu không có thì thông báo "Tiếp tục".

Chương 2: KỸ THUẬT TÌM KIẾM (SEARCHING)

2.1. Khái quát về tìm kiếm

Trong thực tế, khi thao tác, khai thác dữ liệu chúng ta hầu như lúc nào cũng phải thực hiện thao tác tìm kiếm. Việc tìm kiếm nhanh hay chậm tùy thuộc vào trạng thái và trật tự của dữ liệu trên đó. Kết quả của việc tìm kiếm có thể là không có (không tìm thấy) hoặc có (tìm thấy). Nếu kết quả tìm kiếm là có tìm thấy thì nhiều khi chúng ta còn phải xác định xem vị trí của phần tử dữ liệu tìm thấy là ở đâu? Trong phạm vi của chương này chúng ta tìm cách giải quyết các câu hỏi này.

Trước khi đi vào nghiên cứu chi tiết, chúng ta giả sử rằng mỗi phần tử dữ liệu được xem xét có một thành phần khóa (Key) để nhận diện, có kiểu dữ liệu là T nào đó, các thành phần còn lại là thông tin (Info) liên quan đến phần tử dữ liệu đó. Như vậy mỗi phần tử dữ liệu có cấu trúc dữ liệu như sau:

```
typedef     struct  DataElement
   {  T       Key;
      InfoType      Info;
   } DataType;
```

Trong tài liệu này, khi nói tới giá trị của một phần tử dữ liệu chúng ta muốn nói tới giá trị khóa (Key) của phần tử dữ liệu đó. Để đơn giản, chúng ta giả sử rằng mỗi phần tử dữ liệu chỉ là thành phần khóa nhận diện.

Việc tìm kiếm một phần tử có thể diễn ra trên một dãy/mảng (tìm kiếm nội) hoặc diễn ra trên một tập tin/ file (tìm kiếm ngoại). Phần tử cần tìm là phần tử cần thỏa mãn điều kiện tìm kiếm (thường có giá trị bằng giá trị tìm kiếm). Tùy thuộc vào từng bài toán cụ thể mà điều kiện tìm kiếm có thể khác nhau song chung quy việc tìm kiếm dữ liệu thường được vận dụng theo các thuật toán trình bày sau đây.

2.2. Các giải thuật tìm kiếm nội (Tìm kiếm trên dãy/mảng)

2.2.1. Đặt vấn đề

Giả sử chúng ta có một mảng M gồm N phần tử. Vấn đề đặt ra là có hay không phần tử có giá trị bằng X trong mảng M? Nếu có thì phần tử có giá trị bằng X là phần tử thứ mấy trong mảng M?

2.2.2. Tìm tuyến tính (Linear Search)

Thuật toán tìm tuyến tính còn được gọi là Thuật toán tìm kiếm tuần tự (Sequential Search).

a. Tư tưởng:

Lần lượt so sánh các phần tử của mảng M với giá trị X bắt đầu từ phần tử đầu tiên cho đến khi tìm đến được phần tử có giá trị X hoặc đã duyệt qua hết tất cả các phần tử của mảng M thì kết thúc.

b. Thuật toán:

B1: k = 1 //Duyệt từ đầu mảng
B2: IF M[k] ≠ X AND k ≤ N //Nếu chưa tìm thấy và cũng chưa duyệt hết mảng
    B2.1: k++
    B2.2: Lặp lại B2
B3: IF k ≤ N
    Tìm thấy tại vị trí k
B4: ELSE
    Không tìm thấy phần tử có giá trị X
B5: Kết thúc

c. Cài đặt thuật toán:

Hàm LinearSearch có prototype:

```
int  LinearSearch (T  M[], int  N, T  X);
```

Hàm thực hiện việc tìm kiếm phần tử có giá trị X trên mảng M có N phần tử. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ 0 đến N-1 là vị trí tương ứng của phần tử tìm thấy. Trong trường hợp ngược lại, hàm trả về giá trị –1 (không tìm thấy). Nội dung của hàm như sau:

```
int  LinearSearch (T  M[], int  N, T  X)
  {  int k = 0;
     while (M[k] != X && k < N)
           k++;
     if (k < N)
           return (k);
     return (-1);
  }
```

d. Phân tích thuật toán:

- Trường hợp tốt nhất khi phần tử đầu tiên của mảng có giá trị bằng X:

  Số phép gán: Gmin = 1
  Số phép so sánh: Smin = 2 + 1 = 3

- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:

  Số phép gán: Gmax = 1
  Số phép so sánh: Smax = 2N+1

- Trung bình:

  Số phép gán: Gavg = 1
  Số phép so sánh: Savg = (3 + 2N + 1) : 2 = N + 2

e. Cải tiến thuật toán:

Trong thuật toán trên, ở mỗi bước lặp chúng ta cần phải thực hiện 2 phép so sánh để kiểm tra sự tìm thấy và kiểm soát sự hết mảng trong quá trình duyệt mảng. Chúng ta có thể giảm bớt 1 phép so sánh nếu chúng ta thêm vào cuối mảng một phần tử cầm canh (sentinel/stand by) có giá trị bằng X để nhận diện ra sự hết mảng khi duyệt mảng, khi đó thuật toán này được cải tiến lại như sau:

B1: k = 1
B2: M[N+1] = X //Phần tử cầm canh
B3: IF M[k] ≠ X
B3.1: k++
B3.2: Lặp lại B3
B4: IF k < N
Tìm thấy tại vị trí k
B5: ELSE //k = N song đó chỉ là phần tử cầm canh
Không tìm thấy phần tử có giá trị X
B6: Kết thúc

Hàm LinearSearch được viết lại thành hàm LinearSearch1 như sau:

```
int  LinearSearch1 (T  M[], int  N, T  X)
  {  int k = 0;
     M[N] = X;
     while (M[k] != X)
           k++;
     if (k < N)
           return (k);
     return (-1);
  }
```

f. Phân tích thuật toán cải tiến:

- Trường hợp tốt nhất khi phần tử đầu tiên của mảng có giá trị bằng X:

  Số phép gán: Gmin = 2
  Số phép so sánh: Smin = 1 + 1 = 2

- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:

  Số phép gán: Gmax = 2
  Số phép so sánh: Smax = (N+1) + 1 = N + 2

- Trung bình:

  Số phép gán: Gavg = 2
  Số phép so sánh: Savg = (2 + N + 2) : 2 = N/2 + 2

- Như vậy, nếu thời gian thực hiện phép gán không đáng kể thì thuật toán cải tiến sẽ chạy nhanh hơn thuật toán nguyên thủy.

2.2.3. Tìm nhị phân (Binary Search)

Thuật toán tìm tuyến tính tỏ ra đơn giản và thuận tiện trong trường hợp số phần tử của dãy không lớn lắm. Tuy nhiên, khi số phần tử của dãy khá lớn, chẳng hạn chúng ta tìm kiếm tên một khách hàng trong một danh bạ điện thoại của một thành phố lớn theo thuật toán tìm tuần tự thì quả thực mất rất nhiều thời gian. Trong thực tế, thông thường các phần tử của dãy đã có một thứ tự, do vậy thuật toán tìm nhị phân sau đây sẽ rút ngắn đáng kể thời gian tìm kiếm trên dãy đã có thứ tự. Trong thuật toán này chúng ta giả sử các phần tử trong dãy đã có thứ tự tăng (không giảm dần), tức là các phần tử đứng trước luôn có giá trị nhỏ hơn hoặc bằng (không lớn hơn) phần tử đứng sau nó. Khi đó, nếu X nhỏ hơn giá trị phần tử đứng ở giữa dãy (M[Mid]) thì X chỉ có thể tìm

thấy ở nửa đầu của dãy và ngược lại, nếu X lớn hơn phần tử M[Mid] thì X chỉ có thể tìm thấy ở nửa sau của dãy.

a. Tư tưởng:

Phạm vi tìm kiếm ban đầu của chúng ta là từ phần tử đầu tiên của dãy (First = 1) cho đến phần tử cuối cùng của dãy (Last = N).

So sánh giá trị X với giá trị phần tử đứng ở giữa của dãy M là M[Mid].

Nếu X = M[Mid]: Tìm thấy

Nếu X < M[Mid]: Rút ngắn phạm vi tìm kiếm về nửa đầu của dãy M (Last = Mid–1)

Nếu X > M[Mid]: Rút ngắn phạm vi tìm kiếm về nửa sau của dãy M (First = Mid+1)

Lặp lại quá trình này cho đến khi tìm thấy phần tử có giá trị X hoặc phạm vi tìm kiếm của chúng ta không còn nữa (First > Last).

b. Thuật toán đệ quy (Recursion Algorithm):

B1: First = 1
B2: Last = N
B3: IF (First > Last) //Hết phạm vi tìm kiếm
    B3.1: Không tìm thấy
    B3.2: Thực hiện Bkt
B4: Mid = (First + Last)/ 2
B5: IF (X = M[Mid])
    B5.1: Tìm thấy tại vị trí Mid
    B5.2: Thực hiện Bkt
B6: IF (X < M[Mid])
    Tìm đệ quy từ First đến Last = Mid – 1
B7: IF (X > M[Mid])
    Tìm đệ quy từ First = Mid + 1 đến Last
Bkt: Kết thúc

c. Cài đặt thuật toán đệ quy:

Hàm BinarySearch có prototype:

int BinarySearch (T M[], int N, T X);

Hàm thực hiện việc tìm kiếm phần tử có giá trị X trong mảng M có N phần tử đã có thứ tự tăng. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ 0 đến N-1 là vị trí tương ứng của phần tử tìm thấy. Trong trường hợp ngược lại, hàm trả về giá trị –1 (không tìm thấy). Hàm BinarySearch sử dụng hàm đệ quy RecBinarySearch có prototype:

int RecBinarySearch(T M[], int First, int Last, T X);

Hàm RecBinarySearch thực hiện việc tìm kiếm phần tử có giá trị X trên mảng M trong phạm vi từ phần tử thứ First đến phần tử thứ Last. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ First đến Last là vị trí tương ứng của phần tử tìm thấy. Trong trường hợp ngược lại, hàm trả về giá trị –1 (không tìm thấy). Nội dung của các hàm như sau:

```
int  RecBinarySearch (T  M[], int  First, int  Last, T  X)
   {  if (First > Last)
            return (-1);
      int  Mid = (First + Last)/2;
      if (X == M[Mid])
            return (Mid);
      if (X < M[Mid])
            return(RecBinarySearch(M, First, Mid – 1, X));
      else
            return(RecBinarySearch(M, Mid + 1, Last, X));
   }

//=======================================================

int  BinarySearch (T  M[], int  N, T  X)
   {  return (RecBinarySearch(M, 0, N – 1, X));
   }
```

d. Phân tích thuật toán đệ quy:

- Trường hợp tốt nhất khi phần tử ở giữa của mảng có giá trị bằng X:

  Số phép gán: Gmin = 1
  Số phép so sánh: Smin = 2

- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:

  Số phép gán: Gmax = $\log_2 N + 1$
  Số phép so sánh: Smax = $3\log_2 N + 1$

- Trung bình:

  Số phép gán: Gavg = $\frac{1}{2}\log_2 N + 1$
  Số phép so sánh: Savg = $\frac{1}{2}(3\log_2 N + 3)$

e. Thuật toán không đệ quy (Non-Recursion Algorithm):

B1: First = 1
B2: Last = N
B3: IF (First > Last)
    B3.1: Không tìm thấy
    B3.2: Thực hiện Bkt
B4: Mid = (First + Last)/ 2
B5: IF (X = M[Mid])
    B5.1: Tìm thấy tại vị trí Mid
    B5.2: Thực hiện Bkt
B6: IF (X < M[Mid])
    B6.1: Last = Mid – 1
    B6.2: Lặp lại B3
B7: IF (X > M[Mid])
    B7.1: First = Mid + 1
    B7.2: Lặp lại B3
Bkt: Kết thúc

f. Cài đặt thuật toán không đệ quy:

Hàm NRecBinarySearch có prototype: int NRecBinarySearch (T M[], int N, T X);

Hàm thực hiện việc tìm kiếm phần tử có giá trị X trong mảng M có N phần tử đã có thứ tự tăng. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ 0 đến N-1 là vị trí tương ứng của phần tử tìm thấy. Trong trường hợp ngược lại, hàm trả về giá trị –1 (không tìm thấy). Nội dung của hàm NRecBinarySearch như sau:

```
int NRecBinarySearch (T M[], int N, T X)
  { int First = 0;
    int Last = N – 1;
    while (First <= Last)
      { int Mid = (First + Last)/2;
        if (X == M[Mid])
            return(Mid);
        if (X < M[Mid])
            Last = Mid – 1;
        else
            First = Mid + 1;
      }
    return(-1);
  }
```

g. Phân tích thuật toán không đệ quy:

- Trường hợp tốt nhất khi phần tử ở giữa của mảng có giá trị bằng X:

  Số phép gán: Gmin = 3
  Số phép so sánh: Smin = 2

- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:

  Số phép gán: Gmax = $2\log_2 N + 4$
  Số phép so sánh: Smax = $3\log_2 N + 1$

- Trung bình:

  Số phép gán: Gavg = $\log_2 N + 3.5$
  Số phép so sánh: Savg = $\frac{1}{2}(3\log_2 N + 3)$

h. Ví dụ:

Giả sử ta có dãy M gồm 10 phần tử có khóa như sau (N = 10):

1 3 4 5 8 15 17 22 25 30

- Trước tiên ta thực hiện tìm kiếm phần tử có giá trị X = 5 (tìm thấy):

Lần lặp	First	Last	First > Last	Mid	M[Mid]	X = M[Mid]	X < M[Mid]	X > M[Mid]
Ban đầu	0	9	False	4	8	False	True	False
1	0	3	False	1	3	False	False	True
2	2	3	False	2	4	False	False	True
3	3	3	False	3	5	**True**		

Kết quả sau 3 lần lặp (đệ quy) thuật toán kết thúc.

- Bây giờ ta thực hiện tìm kiếm phần tử có giá trị X = 7 (không tìm thấy):

Lần lặp	First	Last	First > Last	Mid	M[Mid]	X = M[Mid]	X < M[Mid]	X > M[Mid]
Ban đầu	0	9	False	4	8	False	True	False
1	0	3	False	1	3	False	False	True
2	2	3	False	2	4	False	False	True
3	3	3	False	3	5	False	False	True
4	4	3	**True**					

Kết quả sau 4 lần lặp (đệ quy) thuật toán kết thúc.

☞ **Lưu ý**:

- Thuật toán tìm nhị phân chỉ có thể vận dụng trong trường hợp dãy/mảng đã có thứ tự. Trong trường hợp tổng quát chúng ta chỉ có thể áp dụng thuật toán tìm kiếm tuần tự.
- Các thuật toán đệ quy có thể ngắn gọn song tốn kém bộ nhớ để ghi nhận mã lệnh chương trình (mỗi lần gọi đệ quy) khi chạy chương trình, do vậy có thể làm cho chương trình chạy chậm lại. Trong thực tế, khi viết chương trình nếu có thể chúng ta nên sử dụng thuật toán không đệ quy.

2.3. Các giải thuật tìm kiếm ngoại (Tìm kiếm trên tập tin)

2.3.1. Đặt vấn đề

Giả sử chúng ta có một tập tin F lưu trữ N phần tử. Vấn đề đặt ra là có hay không phần tử có giá trị bằng X được lưu trữ trong tập tin F? Nếu có thì phần tử có giá trị bằng X là phần tử nằm ở vị trí nào trên tập tin F?

2.3.2. Tìm tuyến tính

a. Tư tưởng:

Lần lượt đọc các phần tử từ đầu tập tin F và so sánh với giá trị X cho đến khi đọc được phần tử có giá trị X hoặc đã đọc hết tập tin F thì kết thúc.

b. Thuật toán:

```
B1: k = 0
B2: rewind(F)                  //Về đầu tập tin F
B3: read(F, a)                 //Đọc một phần tử từ tập tin F
B4: k = k + sizeof(T)          //Vị trí phần tử hiện hành (sau phần tử mới đọc)
B5: IF a ≠ X AND !(eof(F))
        Lặp lại B3
B6: IF (a = X)
        Tìm thấy tại vị trí k byte(s) tính từ đầu tập tin
B7: ELSE
        Không tìm thấy phần tử có giá trị X
```

B8: Kết thúc

c. Cài đặt thuật toán:

Hàm FLinearSearch có prototype:

long FLinearSearch (char * FileName, T X);

Hàm thực hiện tìm kiếm phần tử có giá trị X trong tập tin có tên FileName. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ 0 đến filelength(FileName) là vị trí tương ứng của phần tử tìm thấy so với đầu tập tin (tính bằng byte). Trong trường hợp ngược lại, hoặc có lỗi khi thao tác trên tập tin hàm trả về giá trị –1 (không tìm thấy hoặc lỗi thao tác trên tập tin). Nội dung của hàm như sau:

```
long FLinearSearch (char * FileName, T X)
{ FILE * Fp;
  Fp = fopen(FileName, "rb");
  if (Fp == NULL)
        return (-1);
  long k = 0;
  T a;
  int SOT = sizeof(T);
  while (!feof(Fp))
    { if (fread(&a, SOT, 1, Fp) == 0)
            break;
      k = k + SOT;
      if (a == X)
            break;
    }
  fclose(Fp);
  if (a == X)
        return (k - SOT);
  return (-1);
}
```

d. Phân tích thuật toán:

- Trường hợp tốt nhất khi phần tử đầu tiên của tập tin có giá trị bằng X:

  Số phép gán: Gmin = 1 + 2 = 3
  Số phép so sánh: Smin = 2 + 1 = 3
  Số lần đọc tập tin: Dmin = 1

- Trường hợp xấu nhất khi không tìm thấy phần tử nào có giá trị bằng X:

  Số phép gán: Gmax = N + 2
  Số phép so sánh: Smax = 2N + 1
  Số lần đọc tập tin: Dmax = N

- Trung bình:

  Số phép gán: Gavg = ½(N + 5)
  Số phép so sánh: Savg = (3 + 2N + 1) : 2 = N + 2
  Số lần đọc tập tin: Davg = ½(N + 1)

2.3.3. Tìm kiếm theo chỉ mục (Index Search)

Như chúng ta đã biết, mỗi phần tử dữ liệu được lưu trữ trong tập tin dữ liệu F thường có kích thước lớn, điều này cũng làm cho kích thước của tập tin F cũng khá lớn. Vì vậy việc thao tác dữ liệu trực tiếp lên tập tin F sẽ trở nên lâu, chưa kể sự mất an toàn cho dữ liệu trên tập tin. Để giải quyết vấn đề này, đi kèm theo một tập tin dữ liệu thường có thêm các tập tin chỉ mục (Index File) để làm nhiệm vụ điều khiển thứ tự truy xuất dữ liệu trên tập tin theo một khóa chỉ mục (Index key) nào đó. Mỗi phần tử dữ liệu trong tập tin chỉ mục IDX gồm có 2 thành phần: Khóa chỉ mục và Vị trí vật lý của phần tử dữ liệu có khóa chỉ mục tương ứng trên tập tin dữ liệu. Cấu trúc dữ liệu của các phần tử trong tập tin chỉ mục như sau:

```
typedef     struct  IdxElement
   {  T       IdxKey;
      long  Pos;
   } IdxType;
```

Tập tin chỉ mục luôn luôn được sắp xếp theo thứ tự tăng của khóa chỉ mục. Việc tạo tập tin chỉ mục IDX sẽ được nghiên cứu trong Chương 3, trong phần này chúng ta xem như đã có tập tin chỉ mục IDX để thao tác.

a. Tư tưởng:

Lần lượt đọc các phần tử từ đầu tập tin IDX và so sánh thành phần khóa chỉ mục với giá trị X cho đến khi đọc được phần tử có giá trị khóa chỉ mục lớn hơn hoặc bằng X hoặc đã đọc hết tập tin IDX thì kết thúc. Nếu tìm thấy thì ta đã có vị trí vật lý của phần tử dữ liệu trên tập tin dữ liệu F, khi đó chúng ta có thể truy cập trực tiếp đến vị trí này để đọc dữ liệu của phần tử tìm thấy.

b. Thuật toán:

```
B1: rewind(IDX)
B2: read(IDX, ai)
B3: IF ai.IdxKey < X AND !(eof(IDX))
       Lặp lại B2
B4: IF ai.IdxKey = X
       Tìm thấy tại vị trí ai.Pos byte(s) tính từ đầu tập tin
B5: ELSE
       Không tìm thấy phần tử có giá trị X
B6: Kết thúc
```

c. Cài đặt thuật toán:

Hàm IndexSearch có prototype:

```
long  IndexSearch (char  *  IdxFileName, T  X);
```

Hàm thực hiện tìm kiếm phần tử có giá trị X dựa trên tập tin chỉ mục có tên IdxFileName. Nếu tìm thấy, hàm trả về một số nguyên có giá trị từ 0 đến filelength(FileName)-1 là vị trí tương ứng của phần tử tìm thấy so với đầu tập tin dữ liệu (tính bằng byte). Trong trường hợp ngược lại, hoặc có lỗi khi thao tác trên tập tin chỉ mục hàm trả về giá trị –1 (không tìm thấy). Nội dung của hàm như sau:

```
long  IndexSearch (char  *  IdxFileName, T  X)
   {  FILE * IDXFp;
      IDXFp = fopen(IdxFileName, "rb");
      if (IDXFp == NULL)
            return (-1);
      IdxType  ai;
      int  SOIE = sizeof(IdxType);
      while (!feof(IDXFp))
         {  if (fread(&ai, SOIE, 1, IDXFp) == 0)
                  break;
            if (ai.IdxKey >= X)
                  break;
         }
      fclose(IDXFp);
      if (ai.IdxKey == X)
            return (ai.Pos);
      return (-1);
   }
```

d. Phân tích thuật toán:

- Trường hợp tốt nhất khi phần tử đầu tiên của tập tin chỉ mục có giá trị khóa chỉ mục lớn hơn hoặc bằng X:

  Số phép gán: Gmin = 1
  Số phép so sánh: Smin = 2 + 1 = 3
  Số lần đọc tập tin: Dmin = 1

- Trường hợp xấu nhất khi mọi phần tử trong tập tin chỉ mục đều có khóa chỉ mục nhỏ hơn giá trị X:

  Số phép gán: Gmax = 1
  Số phép so sánh: Smax = 2N + 1
  Số lần đọc tập tin: Dmax = N

- Trung bình:

  Số phép gán: Gavg = 1
  Số phép so sánh: Savg = (3 + 2N + 1) : 2 = N + 2
  Số lần đọc tập tin: Davg = ½(N + 1)

Câu hỏi và Bài tập

1. Trình bày tư tưởng của các thuật toán tìm kiếm: Tuyến tính, Nhị phân, Chỉ mục? Các thuật toán này có thể được vận dụng trong các trường hợp nào? Cho ví dụ?

2. Cài đặt lại thuật toán tìm tuyến tính bằng các cách:
- Sử dụng vòng lặp for,
- Sử dụng vòng lặp do ... while?

Có nhận xét gì cho mỗi trường hợp?

3. Trong trường hợp các phần tử của dãy đã có thứ tự tăng, hãy cải tiến lại thuật toán tìm tuyến tính? Cài đặt các thuật toán cải tiến? Đánh giá và so sánh giữa thuật toán nguyên thủy với các thuật toán cải tiến.

4. Trong trường hợp các phần tử của dãy đã có thứ tự giảm, hãy trình bày và cài đặt lại thuật toán tìm nhị phân trong hai trường hợp: Đệ quy và Không đệ quy?

5. Vận dụng thuật toán tìm nhị phân, hãy cải tiến và cài đặt lại thuật toán tìm kiếm dựa theo tập tin chỉ mục? Đánh giá và so sánh giữa thuật toán nguyên thủy với các thuật toán cải tiến?

6. Sử dụng hàm random trong C để tạo ra một dãy (mảng) M có tối thiểu 1.000 số nguyên, sau đó chọn ngẫu nhiên (cũng bằng hàm random) một giá trị nguyên K. Vận dụng các thuật toán tìm tuyến tính, tìm nhị phân để tìm kiếm phần tử có giá trị K trong mảng M.

Với cùng một dữ liệu như nhau, cho biết thời gian thực hiện các thuật toán.

7. Trình bày và cài đặt thuật toán tìm tuyến tính đối với các phần tử trên mảng hai chiều trong hai trường hợp:

- Không sử dụng phần tử "Cầm canh".
- Có sử dụng phần tử "Cầm canh".

Cho biết thời gian thực hiện của hai thuật toán trong hai trường hợp trên.

8. Sử dụng hàm random trong C để tạo ra tối thiểu 1.000 số nguyên và lưu trữ vào một tập tin có tên SONGUYEN.DAT, sau đó chọn ngẫu nhiên (cũng bằng hàm random) một giá trị nguyên K. Vận dụng thuật toán tìm tuyến tính để tìm kiếm phần tử có giá trị K trong tập tin SONGUYEN.DAT.

9. Thông tin về mỗi nhân viên bao gồm: Mã số – là một số nguyên dương, Họ và Đệm – là một chỗi có tối đa 20 ký tự, Tên nhân viên – là một chuỗi có tối đa 10 ký tự, Ngày, Tháng, Năm sinh – là các số nguyên dương, Phái – Là "Nam" hoặc "Nữ", Hệ số lương, Lương căn bản, Phụ cấp – là các số thực. Viết chương trình nhập vào danh sách nhân viên (ít nhất là 10 người, không nhập trùng mã giữa các nhân viên với nhau) và lưu trữ danh sách nhân viên này vào một tập tin có tên NHANSU.DAT, sau đó vận dụng thuật toán tìm tuyến tính để tìm kiếm trên tập tin NHANSU.DAT xem có hay không nhân viên có mã là K (giá trị của K có thể nhập vào từ bàn phím hoặc phát sinh bằng hàm random). Nếu tìm thấy nhân viên có mã là K thì in ra màn hình toàn bộ thông tin về nhân viên này.

10. Với tập tin dữ liệu có tên NHANSU.DAT trong bài tập **9,** thực hiện các yêu cầu sau:

- Tạo một bảng chỉ mục theo Tên nhân viên.
- Tìm kiếm trên bảng chỉ mục xem trong tập tin NHANSU.DAT có hay không nhân viên có tên là X, nếu có thì in ra toàn bộ thông tin về nhân viên này.
- Lưu trữ bảng chỉ mục này vào trong tập tin có tên NSTEN.IDX.
- Vận dụng thuật toán tìm kiếm dựa trên tập tin chỉ mục NSTEN.IDX để tìm xem có hay không nhân viên có tên là X trong tập tin NHANSU.DAT, nếu có thì in ra toàn bộ thông tin về nhân viên này.
- Có nhận xét gì khi thực hiện tìm kiếm dữ liệu trên tập tin bằng các phương pháp: Tìm tuyến tính và Tìm kiếm dựa trên tập tin chỉ mục.

Chương 3: KỸ THUẬT SẮP XẾP (SORTING)

3.1. Khái quát về sắp xếp

Để thuận tiện và giảm thiểu thời gian thao tác mà đặc biệt là để tìm kiếm dữ liệu dễ dàng và nhanh chóng, thông thường trước khi thao tác thì dữ liệu trên mảng, trên tập tin đã có thứ tự. Do vậy, thao tác sắp xếp dữ liệu là một trong những thao tác cần thiết và thường gặp trong quá trình lưu trữ, quản lý dữ liệu.

Thứ tự xuất hiện dữ liệu có thể là thứ tự tăng (không giảm dần) hoặc thứ tự giảm (không tăng dần). Trong phạm vi chương này chúng ta sẽ thực hiện việc sắp xếp dữ liệu theo thứ tự tăng. Việc sắp xếp dữ liệu theo thứ tự giảm hoàn toàn tương tự.

Có rất nhiều thuật toán sắp xếp song chúng ta có thể phân chia các thuật toán sắp xếp thành hai nhóm chính căn cứ vào vị trí lưu trữ của dữ liệu trong máy tính, đó là:

- Các giải thuật sắp xếp thứ tự nội (sắp xếp thứ tự trên dãy/mảng),
- Các giải thuật sắp xếp thứ tự ngoại (sắp xếp thứ tự trên tập tin/file).

Cũng như trong chương trước, chúng ta giả sử rằng mỗi phần tử dữ liệu được xem xét có một thành phần khóa (Key) để nhận diện, có kiểu dữ liệu là T nào đó, các thành phần còn lại là thông tin (Info) liên quan đến phần tử dữ liệu đó. Như vậy mỗi phần tử dữ liệu có cấu trúc dữ liệu như sau:

```
typedef     struct  DataElement
   {  T       Key;
      InfoType      Info;
   } DataType;
```

Trong chương này nói riêng và tài liệu này nói chung, các thuật toán sắp xếp của chúng ta là sắp xếp sao cho các phần tử dữ liệu có thứ tự tăng theo thành phần khóa (Key) nhận diện. Để đơn giản, chúng ta giả sử rằng mỗi phần tử dữ liệu chỉ là thành phần khóa nhận diện.

3.2. Các giải thuật sắp xếp nội (Sắp xếp trên dãy/mảng)

Ở đây, toàn bộ dữ liệu cần sắp xếp được đưa vào trong bộ nhớ trong (RAM). Do vậy, số phần tử dữ liệu không lớn lắm do giới hạn của bộ nhớ trong, tuy nhiên tốc độ sắp xếp tương đối nhanh. Các giải thuật sắp xếp nội bao gồm các nhóm sau:

- Sắp xếp bằng phương pháp đếm (counting sort),
- Sắp xếp bằng phương pháp đổi chỗ (exchange sort),
- Sắp xếp bằng phương pháp chọn lựa (selection sort),
- Sắp xếp bằng phương pháp chèn (insertion sort),
- Sắp xếp bằng phương pháp trộn (merge sort).

Trong phạm vi của giáo trình này chúng ta chỉ trình bày một số thuật toán sắp xếp tiêu biểu trong các thuật toán sắp xếp ở các nhóm trên và giả sử thứ tự sắp xếp N phần tử có kiểu dữ liệu T trong mảng M là thứ tự tăng.

3.2.1. Sắp xếp bằng phương pháp đổi chỗ (Exchange Sort)

Các thuật toán trong phần này sẽ tìm cách đổi chỗ các phần tử đứng sai vị trí (so với mảng đã sắp xếp) trong mảng M cho nhau để cuối cùng tất cả các phần tử trong mảng M đều về đúng vị trí như mảng đã sắp xếp.

Các thuật toán sắp xếp bằng phương pháp đổi chỗ bao gồm:

- Thuật toán sắp xếp nổi bọt (bubble sort),
- Thuật toán sắp xếp lắc (shaker sort),
- Thuật toán sắp xếp giảm độ tăng hay độ dài bước giảm dần (shell sort),
- Thuật toán sắp xếp dựa trên sự phân hoạch (quick sort).

Ở đây chúng ta trình bày hai thuật toán phổ biến là thuật toán sắp xếp nổi bọt và sắp xếp dựa trên sự phân hoạch.

a. Thuật toán sắp xếp nổi bọt (Bubble Sort):

- Tư tưởng:

+ Đi từ cuối mảng về đầu mảng, trong quá trình đi nếu phần tử ở dưới (đứng phía sau) nhỏ hơn phần tử đứng ngay trên (trước) nó thì theo nguyên tắc của bọt khí phần tử nhẹ sẽ bị "trồi" lên phía trên phần tử nặng (hai phần tử này sẽ được đổi chỗ cho nhau). Kết quả là phần tử nhỏ nhất (nhẹ nhất) sẽ được đưa lên (trồi lên) trên bề mặt (đầu mảng) rất nhanh.

+ Sau mỗi lần đi chúng ta đưa được một phần tử trồi lên đúng chỗ. Do vậy, sau N–1 lần đi thì tất cả các phần tử trong mảng M sẽ có thứ tự tăng.

- Thuật toán:

```
B1: First = 1
B2: IF (First = N)
        Thực hiện Bkt
B3: ELSE
        B3.1: Under = N
        B3.2: If (Under = First)
                Thực hiện B4
        B3.3: Else
                B3.3.1: if (M[Under] < M[Under - 1])
                        Swap(M[Under], M[Under – 1])        //Đổi chỗ 2 phần tử cho nhau
                B3.3.2: Under--
                B3.3.3: Lặp lại B3.2
B4: First++
B5: Lặp lại B2
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm BubbleSort có prototype như sau:

```
void  BubbleSort(T  M[], int  N);
```

Hàm thực hiện việc sắp xếp N phần tử có kiểu dữ liệu T trên mảng M theo thứ tự tăng dựa trên thuật toán sắp xếp nổi bọt. Nội dung của hàm như sau:

```
void  BubbleSort(T  M[], int  N)
   {  for (int I = 0; I < N-1; I++)
         for (int J = N-1; J > I; J--)
            if (M[J] < M[J-1])
               Swap(M[J], M[J-1]);
      return;
   }
```

Hàm Swap có prototype như sau:

```
void  Swap(T  &X, T  &Y);
```

Hàm thực hiện việc hoán vị giá trị của hai phần tử X và Y cho nhau. Nội dung của hàm như sau:

```
void  Swap(T  &X, T  &Y)
   {  T  Temp = X;
      X = Y;
      Y = Temp;
      return;
   }
```

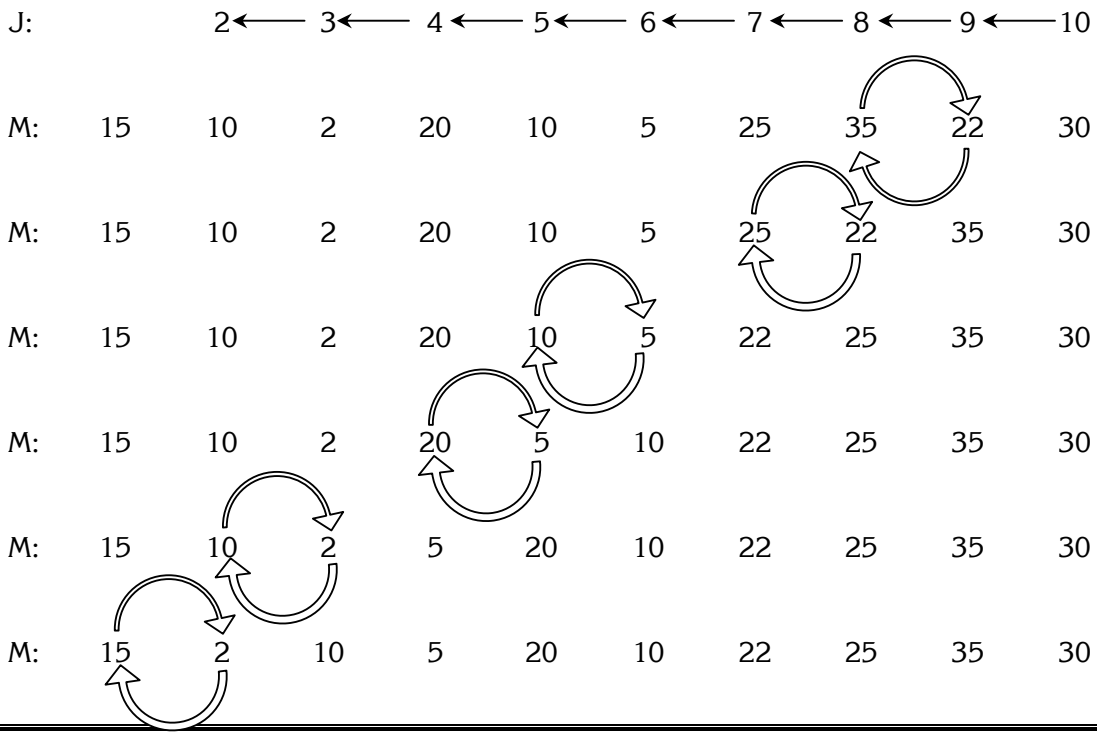
- Ví dụ minh họa thuật toán:

Giả sử ta cần sắp xếp mảng M có 10 phần tử sau (N = 10):

M: 15 10 2 20 10 5 25 35 22 30

Ta sẽ thực hiện 9 lần đi (N - 1 = 10 - 1 = 9) để sắp xếp mảng M:

Lần 1: First = 1

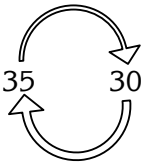


M: ***2*** 15 10 5 20 10 22 25 35 30

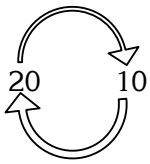
Lần 2: First = 2

J: 3 ← 4 ← 5 ← 6 ← 7 ← 8 ← 9 ← 10

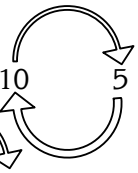
M: ***2*** 15 10 5 20 10 22 25 35 30



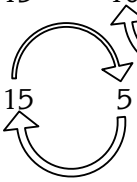
M: ***2*** 15 10 5 20 10 22 25 30 35



M: ***2*** 15 10 5 10 20 22 25 30 35



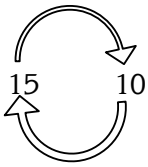
M: ***2*** 15 5 10 10 20 22 25 30 35



M: ***2*** ***5*** 15 10 10 20 22 25 30 35

Lần 3: First = 3

J: 4 ← 5 ← 6 ← 7 ← 8 ← 9 ← 10

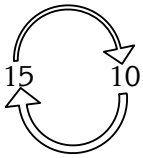


M: ***2*** ***5*** 15 10 10 20 22 25 30 35

M: ***2*** ***5*** ***10*** 15 10 20 22 25 30 35

Lần 4: First = 4

J: 5 ← 6 ← 7 ← 8 ← 9 ← 10



M: ***2*** ***5*** ***10*** 15 10 20 22 25 30 35

M: ***2*** ***5*** ***10*** ***10*** 15 20 22 25 30 35

Lần 5: First = 5

J: 6 ← 7 ← 8 ← 9 ← 10

M: ***2*** ***5*** ***10*** ***10*** ***15*** 20 22 25 30 35

Lần 6: First = 6

J: 7 ← 8 ← 9 ← 10

M: ***2*** ***5*** ***10*** ***10*** ***15*** ***20*** 22 25 30 35

Lần 7: First = 7

J: 8 ← 9 ← 10

M: ***2*** ***5*** ***10*** ***10*** ***15*** ***20*** ***22*** 25 30 35

Lần 8: First = 8

J: 9 ← 10

M: ***2*** ***5*** ***10*** ***10*** ***15*** ***20*** ***22*** ***25*** 30 35

Lần 9: First = 9

J: 10

M: ***2*** ***5*** ***10*** ***10*** ***15*** ***20*** ***22*** ***25*** ***30*** 35

Sau 9 lần đi mảng M trở thành:

M: 2 5 10 10 15 20 22 25 30 35

- Phân tích thuật toán:

+ Trong mọi trường hợp:

Số phép gán: G = 0

Số phép so sánh: S = (N-1) + (N-2) + ... + 1 = ½N(N-1)

+ Trong trường hợp tốt nhất: khi mảng ban đầu đã có thứ tự tăng

Số phép hoán vị: Hmin = 0

+ Trong trường hợp xấu nhất: khi mảng ban đầu đã có thứ tự giảm

Số phép hoán vị: Hmin = (N-1) + (N-2) + ... + 1 = ½N(N-1)

+ Số phép hoán vị trung bình: Havg = ¼N(N-1)

- Nhận xét về thuật toán nổi bọt:

+ Thuật toán sắp xếp nổi bọt khá đơn giản, dễ hiểu và dễ cài đặt.

+ Trong thuật toán sắp xếp nổi bọt, mỗi lần đi từ cuối mảng về đầu mảng thì phần tử nhẹ được trồi lên rất nhanh trong khi đó phần tử nặng lại "chìm" xuống khá chậm chạp do không tận dụng được chiều đi xuống (chiều từ đầu mảng về cuối mảng).

+ Thuật toán nổi bọt không phát hiện ra được các đoạn phần tử nằm hai đầu của mảng đã nằm đúng vị trí để có thể giảm bớt quãng đường đi trong mỗi lần đi.

b. Thuật toán sắp xếp dựa trên sự phân hoạch (Partitioning Sort):

Thuật toán sắp xếp dựa trên sự phân hoạch còn được gọi là thuật toán sắp xếp nhanh (Quick Sort).

- Tư tưởng:

+ Phân hoạch dãy M thành 03 dãy con có thứ tự tương đối thỏa mãn điều kiện:

Dãy con thứ nhất (đầu dãy M) gồm các phần tử có giá trị nhỏ hơn giá trị trung bình của dãy M,

Dãy con thứ hai (giữa dãy M) gồm các phần tử có giá trị bằng giá trị trung bình của dãy M,

Dãy con thứ ba (cuối dãy M) gồm các phần tử có giá trị lớn hơn giá trị trung bình của dãy M,

+ Nếu dãy con thứ nhất và dãy con thứ ba có nhiều hơn 01 phần tử thì chúng ta lại tiếp tục phân hoạch đệ quy các dãy con này.

+ Việc tìm giá trị trung bình của dãy M hoặc tìm kiếm phần tử trong M có giá trị bằng giá trị trung bình của dãy M rất khó khăn và mất thời gian. Trong thực tế, chúng ta chọn một phần tử bất kỳ (thường là phần tử đứng ở vị trí giữa) trong dãy các phần tử cần phân hoạch để làm giá trị cho các phần tử của dãy con thứ hai (dãy giữa) sau khi phân hoạch. Phần tử này còn được gọi là phần tử biên (boundary element). Các phần tử trong dãy con thứ nhất sẽ có giá trị nhỏ hơn giá trị phần tử biên và các phần tử trong dãy con thứ ba sẽ có giá trị lớn hơn giá trị phần tử biên.

+ Việc phân hoạch một dãy được thực hiện bằng cách tìm các cặp phần tử đứng ở hai dãy con hai bên phần tử giữa (dãy 1 và dãy 3) nhưng bị sai thứ tự (phần tử đứng ở dãy 1 có giá trị lớn hơn giá trị phần tử giữa và phần tử đứng ở dãy 3 có giá trị nhỏ hơn giá trị phần tử giữa) để đổi chỗ (hoán vị) cho nhau.

- Thuật toán:

B1: First = 1
B2: Last = N
B3: IF (First ≥ Last) //Dãy con chỉ còn không quá 01 phần tử
    Thực hiện Bkt
B4: X = M[(First+Last)/2] //Lấy giá trị phần tử giữa
B5: I = First //Xuất phát từ đầu dãy 1 để tìm phần tử có giá trị > X
B6: IF (M[I] > X)
    Thực hiện B8
B7: ELSE
    B7.1: I++
    B7.2: Lặp lại B6
B8: J = Last //Xuất phát từ cuối dãy 3 để tìm phần tử có giá trị < X
B9: IF (M[J] < X)
    Thực hiện B11
B10: ELSE
    B10.1: J--
    B10.2: Lặp lại B9
B11: IF (I ≤ J)
    B11.1: Hoán_Vị(M[I], M[J])
    B11.2: I++
    B11.3: J--
    B11.4: Lặp lại B6
B12: ELSE
    B12.1: Phân hoạch đệ quy dãy con từ phần tử thứ First đến phần tử thứ J
    B12.2: Phân hoạch đệ quy dãy con từ phần tử thứ I đến phần tử thứ Last
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm QuickSort có prototype như sau:

```
void  QuickSort(T  M[], int  N);
```

Hàm thực hiện việc sắp xếp N phần tử có kiểu dữ liệu T trên mảng M theo thứ tự tăng dựa trên thuật toán sắp xếp nhanh. Hàm QuickSort sử dụng hàm phân hoạch đệ quy PartitionSort để thực hiện việc sắp xếp theo thứ tự tăng các phần tử của một dãy con giới hạn từ phần tử thứ First đến phần tử thứ Last trên mảng M. Hàm PartitionSort có prototype như sau:

```
void  PartitionSort(T  M[], int  First, int  Last);
```

Nội dung của các hàm như sau:

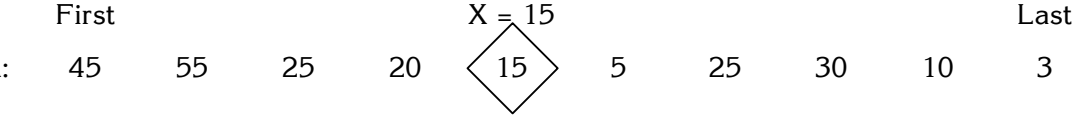
```
void  PartitionSort(T  M[], int  First, int  Last)
   {  if (First >= Last)
         return;
      T  X = M[(First+Last)/2];
      int  I = First;
      int  J = Last;
      do {  while (M[I] < X)
               I++;
            while (M[J] > X)
               J--;
            if (I <= J)
               {  Swap(M[I], M[J]);
                  I++;
                  J--;
               }
         }
      while (I <= J);
      PartitionSort(M, First, J);
      PartitionSort(M, I, Last);
      return;
   }

//==========================================

void  QuickSort(T  M[], int  N)
   {  PartitionSort(M, 0, N-1);
      return;
   }
```

- Ví dụ minh họa thuật toán:

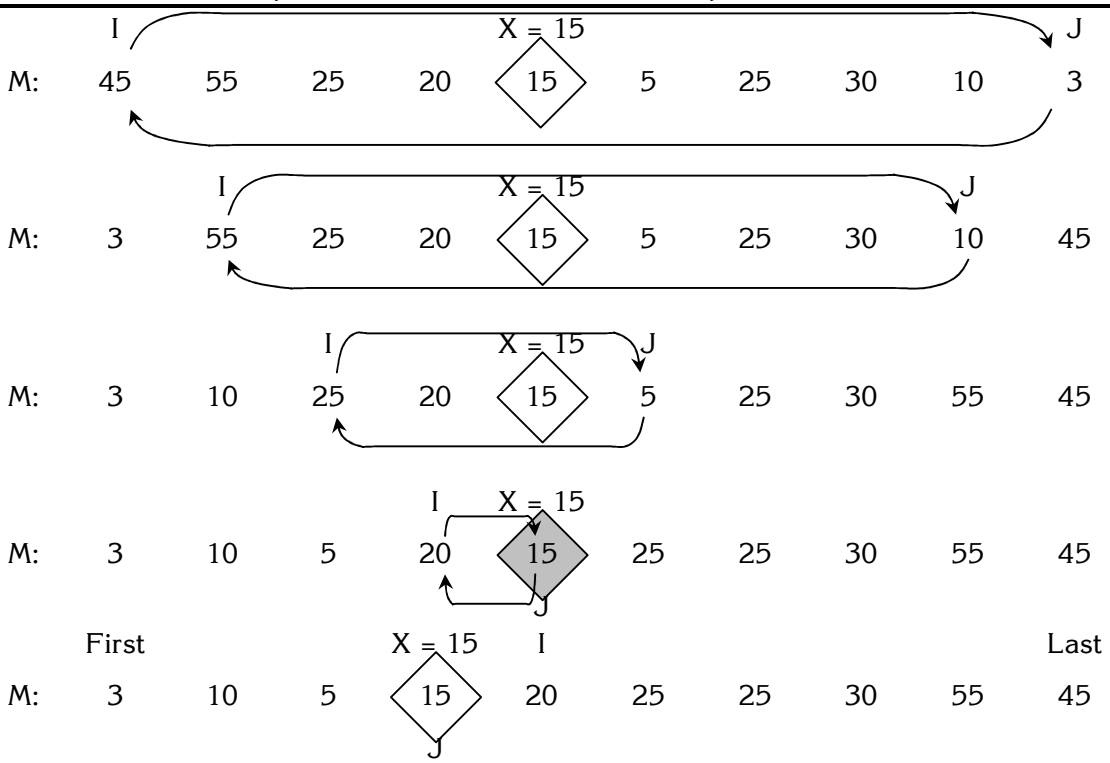
Giả sử ta cần sắp xếp mảng M có 10 phần tử sau (N = 10):

M: 45 55 25 20 15 5 25 30 10 3

Ban đầu: First = 1 Last = 10 X = M[(1+10)/2] =M[5] = 15



Phân hoạch:



Phân hoạch các phần tử trong dãy con từ First -> J:

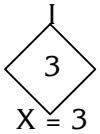
First = 1 Last = J = 4 X = M[(1+4)/2] = M[2] = 10

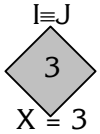
Phân hoạch:

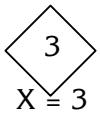
Phân hoạch các phần tử trong dãy con từ First -> J:

First = 1 Last = J = 2 X = M[(1+2)/2] = M[1] = 3

Phân hoạch:

		I	J							
M:			5	10	15	20	25	25	30	55	45
		X = 3									

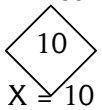
		I≡J								
M:			5	10	15	20	25	25	30	55	45
		X = 3									

J			I							
M:			5	10	15	20	25	25	30	55	45
		X = 3									

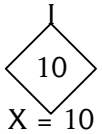
	First	J	I	Last						
M:	3	5	10	15	20	25	25	30	55	45

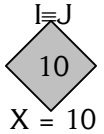
Phân hoạch các phần tử trong dãy con từ I -> Last:

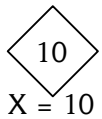
First = I = 3 Last = 4 X = M[(3+4)/2] = M[3] = 10

			First	Last						
M:	3	5		15	20	25	25	30	55	45
			X = 10							

Phân hoạch:

			I	J						
M:	3	5		15	20	25	25	30	55	45
			X = 10							

			I≡J							
M:	3	5		15	20	25	25	30	55	45
			X = 10							

		J		I						
M:	3	5		15	20	25	25	30	55	45
			X = 10							

	First			J	I					Last
M:	3	5	10	15	20	25	25	30	55	45

Phân hoạch các phần tử trong dãy con từ I -> Last:

First = I = 5 Last = 10 X = M[(5+10)/2] = M[7] = 25

					First		X = 25			Last
M:	3	5	10	15	20	25		30	55	45

Phân hoạch:

					I		X = 25			J
M:	3	5	10	15	20	25	25	30	55	45

						I	X = 25			
M:	3	5	10	15	20	25	25	30	55	45
							J			

					First	X = 25	I			Last
M:	3	5	10	15	20	25	25	30	55	45
						J				

Phân hoạch các phần tử trong dãy con từ First -> J:

First = 5 Last = J = 6 X = M[(5+6)/2] = M[5] = 20

					First		Last			
M:	3	5	10	15	20	25	25	30	55	45
					X = 20					

Phân hoạch:

					I		J			
M:	3	5	10	15	20	25	25	30	55	45
					X = 20					

					I≡J					
M:	3	5	10	15	20	25	25	30	55	45
					X = 20					

				J		I				
M:	3	5	10	15	20	25	25	30	55	45
					X = 20					

					First	J	I			Last
M:	3	5	10	15	20	25	25	30	55	45

Phân hoạch các phần tử trong dãy con từ I -> Last:

First = I = 7 Last = 10 X = M[(7+10)/2] = M[8] = 30

							First	X = 30		Last
M:	3	5	10	15	20	25	25	30	55	45

Phân hoạch:

							I	X = 30		J
M:	3	5	10	15	20	25	25	30	55	45

								I≡J		
M:	3	5	10	15	20	25	25	30	55	45

X = 30

J I

M: 3 5 10 15 20 25 25 30 55 45

X = 30

First≡J I Last

M: 3 5 10 15 20 25 25 30 55 45

X = 30

Phân hoạch các phần tử trong dãy con từ I -> Last:

First = I = 9 Last = 10 X = M[(9+10)/2] = M[9] = 55

First Last

M: 3 5 10 15 20 25 25 30 55 45

X = 55

Phân hoạch:

I J

M: 3 5 10 15 20 25 25 30 55 45

X = 55

J I

M: 3 5 10 15 20 25 25 30 45 55

X = 55

M: 3 5 10 15 20 25 25 30 45 55

Toàn bộ quá trình phân hoạch kết thúc, dãy M trở thành:

M: 3 5 10 15 20 25 25 30 45 55

- Phân tích thuật toán:

+ Trường hợp tốt nhất, khi mảng M ban đầu đã có thứ tự tăng:

Số phép gán: Gmin = 1 + 2 + 4 + ... + 2^[Log_2(N) – 1] = N-1
Số phép so sánh: Smin = N×Log_2(N)/2
Số phép hoán vị: Hmin = 0

+ Trường hợp xấu nhất, khi phần tử X được chọn ở giữa dãy con là giá trị lớn nhất của dãy con. Trường hợp này thuật toán QuickSort trở nên chậm chạp nhất:

Số phép gán: Gmax = 1 + 2 + ... + (N-1) = N×(N-1)/2
Số phép so sánh: Smax = (N-1)×(N-1)
Số phép hoán vị: Hmax = (N-1) + (N-2) + ... + 1 = N×(N-1)/2

+ Trung bình:

Số phép gán: Gavg = [(N-1)+N(N-1)/2]/2 = (N-1)×(N+2)/4
Số phép so sánh: Savg = [N×Log_2(N)/2 + N×(N-1)]/2 = N×[Log_2(N)+2N–2]/4
Số phép hoán vị: Havg = N×(N-1)/4

3.2.2. Sắp xếp bằng phương pháp chọn (Selection Sort)

Các thuật toán trong phần này sẽ tìm cách lựa chọn các phần tử thỏa mãn điều kiện chọn lựa để đưa về đúng vị trí của phần tử đó, cuối cùng tất cả các phần tử trong mảng M đều về đúng vị trí.

Các thuật toán sắp xếp bằng phương pháp chọn bao gồm:

- Thuật toán sắp xếp chọn trực tiếp (straight selection sort),
- Thuật toán sắp xếp dựa trên khối/heap hay sắp xếp trên cây (heap sort).

Ở đây chúng ta chỉ trình bày thuật toán sắp xếp chọn trực tiếp

Thuật toán sắp xếp chọn trực tiếp (Straight Selection Sort):

- Tư tưởng:

+ Ban đầu dãy có N phần tử chưa có thứ tự. Ta chọn phần tử có giá trị nhỏ nhất trong N phần tử chưa có thứ tự này để đưa lên đầu nhóm N phần tử.
+ Sau lần thứ nhất chọn lựa phần tử nhỏ nhất và đưa lên đầu nhóm chúng ta còn lại N-1 phần tử đứng ở phía sau dãy M chưa có thứ tự. Chúng ta tiếp tục chọn phần tử có giá trị nhỏ nhất trong N-1 phần tử chưa có thứ tự này để đưa lên đầu nhóm N-1 phần tử. …. Do vậy, sau N–1 lần chọn lựa phần tử nhỏ nhất để đưa lên đầu nhóm thì tất cả các phần tử trong dãy M sẽ có thứ tự tăng.
+ Như vậy, thuật toán này chủ yếu chúng ta đi tìm giá trị nhỏ nhất trong nhóm N-K phần tử chưa có thứ tự đứng ở phía sau dãy M. Việc này đơn giản chúng ta vận dụng thuật toán tìm kiếm tuần tự.

- Thuật toán:

```
B1: K = 0
B2: IF (K = N-1)
        Thực hiện Bkt
B3: Min = M[K+1]
B4: PosMin = K+1
B5: Pos = K+2
B6: IF (Pos > N)
        Thực hiện B8
B7: ELSE
        B7.1: If (Min > M[Pos])
                B7.1.1: Min = M[Pos]
                B7.1.2: PosMin = Pos
        B7.2: Pos++
        B7.3: Lặp lại B6
B8: HoánVị(M[K+1], M[PosMin])
B9: K++
B10: Lặp lại B2
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm SelectionSort có prototype như sau:

```
void  SelectionSort(T  M[], int  N);
```

Hàm thực hiện việc sắp xếp N phần tử có kiểu dữ liệu T trên mảng M theo thứ tự tăng dựa trên thuật toán sắp xếp chọn trực tiếp. Nội dung của hàm như sau:

```
void  SelectionSort(T  M[], int  N)
  {  int  K = 0, PosMin;
     while (K < N-1)
        {  T  Min = M[K];
           PosMin = K;
           for (int  Pos = K+1; Pos < N; Pos++)
              if (Min > M[Pos])
                 {  Min = M[Pos];
                    PosMin = Pos
                 }
           Swap(M[K], M[PosMin]);
           K++;
        }
     return;
  }
```

- Ví dụ minh họa thuật toán:

Giả sử ta cần sắp xếp mảng M có 10 phần tử sau (N = 10):

M: 1 60 2 25 15 45 5 30 33 20

Ta sẽ thực hiện 9 lần chọn lựa (N - 1 = 10 - 1 = 9) phần tử nhỏ nhất để sắp xếp mảng M:

Lần 1: Min = 1 PosMin = 1 K = 0

K+1

M: 1 60 2 25 15 45 5 30 33 20

Lần 2: Min = 2 PosMin = 3 K = 1

K+1

M: ***1*** 60 2 25 15 45 5 30 33 20

K+1

M: ***1*** ***2*** 60 25 15 45 5 30 33 20

Lần 3: Min = 5 PosMin = 7 K = 2

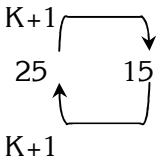
K+1

M: ***1*** ***2*** 60 25 15 45 5 30 33 20

K+1

M: ***1*** ***2*** ***5*** 25 15 45 60 30 33 20

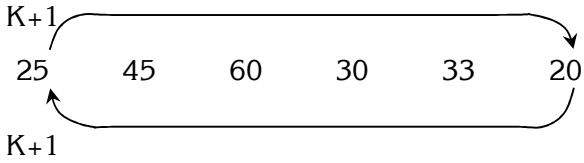
Lần 4: Min = 15 PosMin = 5 K = 3



M: ***1*** ***2*** ***5*** 25 15 45 60 30 33 20

M: ***1*** ***2*** ***5*** ***15*** 25 45 60 30 33 20

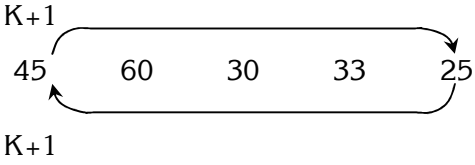
Lần 5: Min = 20 PosMin = 10 K = 4



M: ***1*** ***2*** ***5*** ***15*** 25 45 60 30 33 20

M: ***1*** ***2*** ***5*** ***15*** ***20*** 45 60 30 33 25

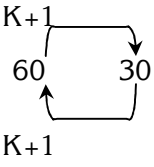
Lần 6: Min = 25 PosMin = 10 K = 5



M: ***1*** ***2*** ***5*** ***15*** ***20*** 45 60 30 33 25

M: ***1*** ***2*** ***5*** ***15*** ***20*** ***25*** 60 30 33 45

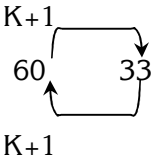
Lần 7: Min = 30 PosMin = 8 K = 6



M: ***1*** ***2*** ***5*** ***15*** ***20*** ***25*** 60 30 33 45

M: ***1*** ***2*** ***5*** ***15*** ***20*** ***25*** ***30*** 60 33 45

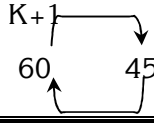
Lần 8: Min = 33 PosMin = 9 K = 7



M: ***1*** ***2*** ***5*** ***15*** ***20*** ***25*** ***30*** 60 33 45

M: ***1*** ***2*** ***5*** ***15*** ***20*** ***25*** ***30*** ***33*** 60 45

Lần 9: Min = 45 PosMin = 10 K = 8



M: ***1*** ***2*** ***5*** ***15*** ***20*** ***25*** ***30*** ***33*** 60 45

									K+1	
M:	*1*	*2*	*5*	*15*	*20*	*25*	*30*	*33*	*45*	60

Sau lần 9: K = 9 và mảng M trở thành:

M:	*1*	*2*	*5*	*15*	*20*	*25*	*30*	*33*	*45*	*60*

- Phân tích thuật toán:

+ Trong mọi trường hợp:
  Số phép so sánh: S = (N-1)+(N-2)+…+1 = N×(N-1)/2
  Số phép hoán vị: H = N-1

+ Trường hợp tốt nhất, khi mảng M ban đầu đã có thứ tự tăng:

  Số phép gán: Gmin = 2×(N-1)

+ Trường hợp xấu nhất, khi mảng M ban đầu đã có thứ tự giảm dần:

  Số phép gán: Gmax = 2×[N+(N-1)+ … +1] = N×(N+1)

+ Trung bình:

  Số phép gán: Gavg = [2×(N-1)+N×(N+1)]/2 = (N-1) + N×(N+1)/2

3.2.3. Sắp xếp bằng phương pháp chèn (Insertion Sort)

Các thuật toán trong phần này sẽ tìm cách tận dụng K phần tử đầu dãy M đã có thứ tự tăng, chúng ta đem phần tử thứ K+1 chèn vào K phần tử đầu dãy sao cho sau khi chèn chúng ta có K+1 phần tử đầu dãy M đã có thứ tự tăng.

Ban đầu dãy M có ít nhất 1 phần tử đầu dãy đã có thứ tự tăng (K=1). Như vậy sau tối đa N-1 bước chèn là chúng ta sẽ sắp xếp xong dãy M có N phần tử theo thứ tự tăng.

Các thuật toán sắp xếp bằng phương pháp chèn bao gồm:

- Thuật toán sắp xếp chèn trực tiếp (straight insertion sort),
- Thuật toán sắp xếp chèn nhị phân (binary insertion sort).

Trong tài liệu này chúng ta chỉ trình bày thuật toán sắp xếp chèn trực tiếp.

Thuật toán sắp xếp chèn trực tiếp (Straight Insertion Sort):

- Tư tưởng:

Để chèn phần tử thứ K+1 vào K phần tử đầu dãy đã có thứ tự chúng ta sẽ tiến hành tìm vị trí đúng của phần tử K+1 trong K phần tử đầu bằng cách vận dụng thuật giải tìm kiếm tuần tự (Sequential Search). Sau khi tìm được vị trí chèn (chắc chắn có vị trí chèn) thì chúng ta sẽ tiến hành chèn phần tử K+1 vào đúng vị trí chèn bằng cách dời các phần tử từ vị trí chèn đến phần tử thứ K sang phải (ra phía sau) 01 vị trí và chèn phần tử K+1 vào vị trí của nó.

- Thuật toán:

B1: K = 1
B2: IF (K = N)

Thực hiện Bkt
B3: X = M[K+1]
B4: Pos = 1
B5: IF (Pos > K)
Thực hiện B7
B6: ELSE //Tìm vị trí chèn
B6.1: If (X <= M[Pos])
Thực hiện B7
B6.2: Pos++
B6.3: Lặp lại B6.1
B7: I = K+1
B8: IF (I > Pos) //Nếu còn phải dời các phần tử từ Pos->K về phía sau 1 vị trí
B8.1: M[I] = M[I-1]
B8.2: I--
B8.3: Lặp lại B8
B9: ELSE //Đã dời xong các phần tử từ Pos->K về phía sau 1 vị trí
B9.1: M[Pos] = X //Chèn X vào vị trí Pos
B9.2: K++
B9.3: Lặp lại B2
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm InsertionSort có prototype như sau:

```
void InsertionSort(T M[], int N);
```

Hàm thực hiện việc sắp xếp N phần tử có kiểu dữ liệu T trên mảng M theo thứ tự tăng dựa trên thuật toán sắp xếp chèn trực tiếp. Nội dung của hàm như sau:

```
void InsertionSort(T M[], int N)
{ int K = 1, Pos;
  while (K < N)
    { T X = M[K];
      Pos = 0;
      while (X > M[Pos])
        Pos++;
      for (int I = K; I > Pos; I--)
        M[I] = M[I-1];
      M[Pos] = X;
      K++;
    }
  return;
}
```

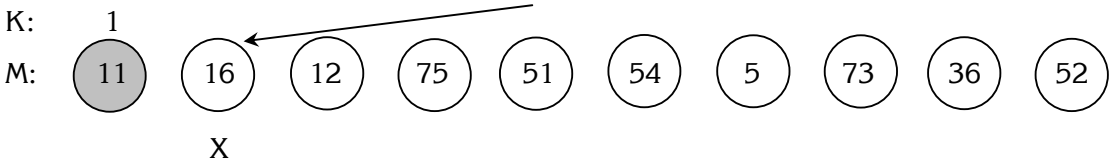
- Ví dụ minh họa thuật toán:

Giả sử ta cần sắp xếp mảng M có 10 phần tử sau (N = 10):

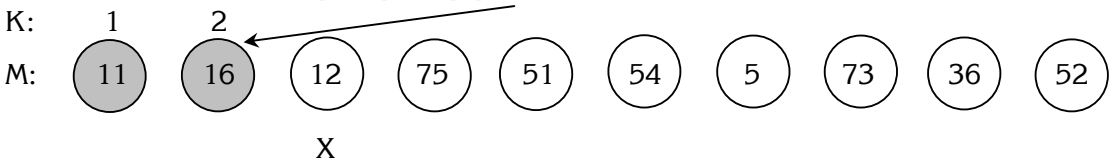
M: 11 16 12 75 51 54 5 73 36 52

Ta sẽ thực hiện 9 lần chèn (N - 1 = 10 - 1 = 9) các phần tử vào dãy con đã có thứ tự tăng đứng đầu dãy M:

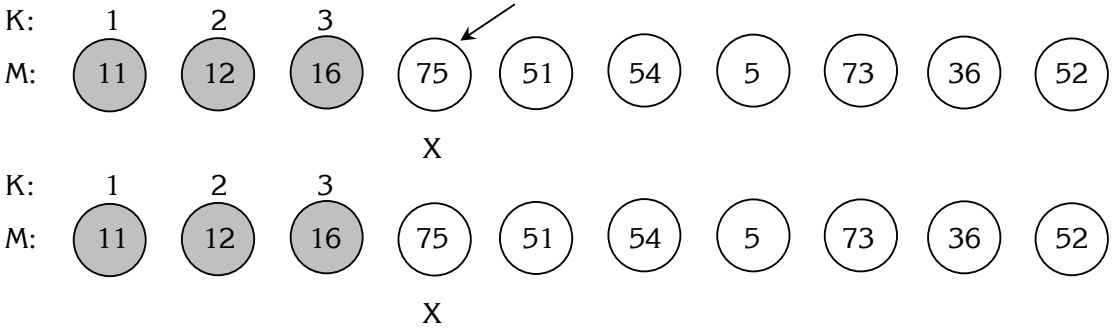
Lần 1: K = 1 X = M[K+1] = M[2] = 16 Pos = 2



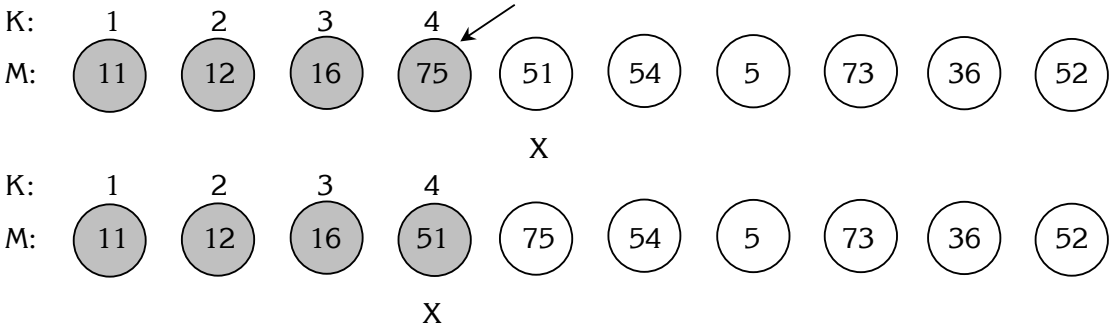
Lần 2: K = 2 X = M[K+1] = M[3] = 12 Pos = 2



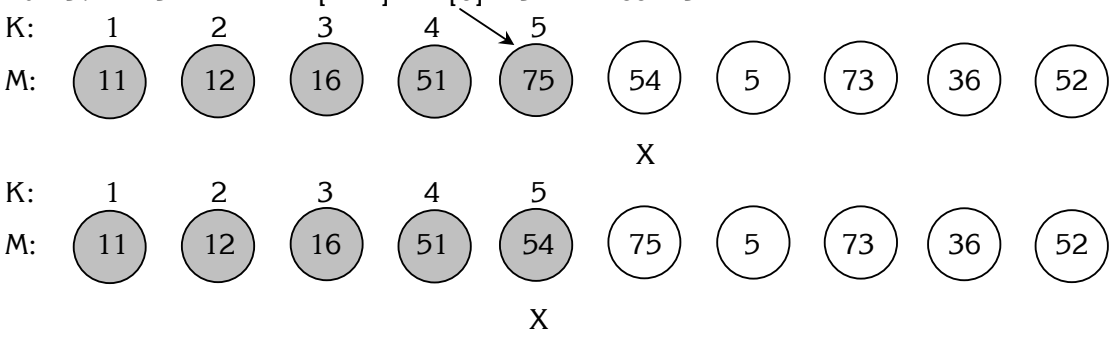
Lần 3: K = 3 X = M[K+1] = M[4] = 75 Pos = 4



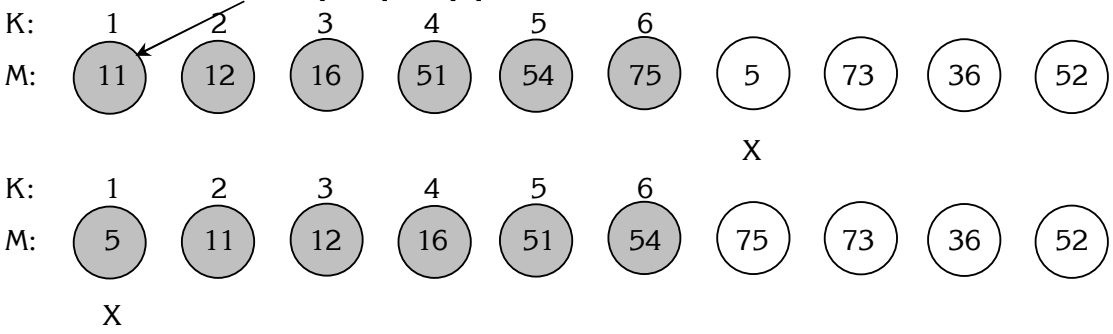
Lần 4: K = 4 X = M[K+1] = M[5] = 51 Pos = 4



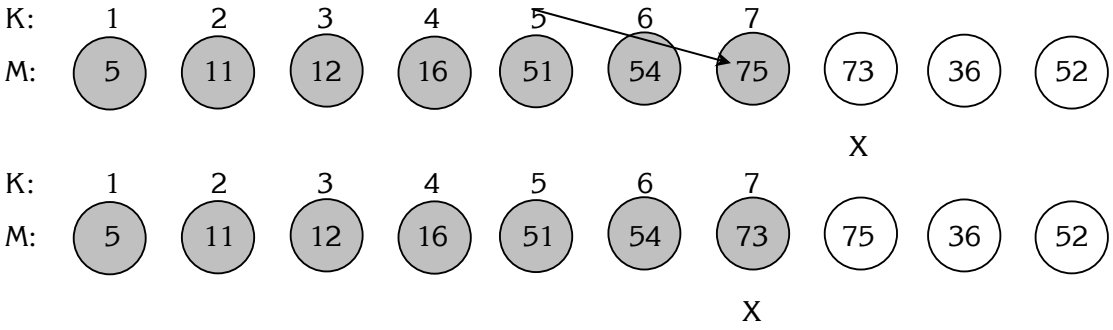
Lần 5: K = 5 X = M[K+1] = M[6] = 54 Pos = 5



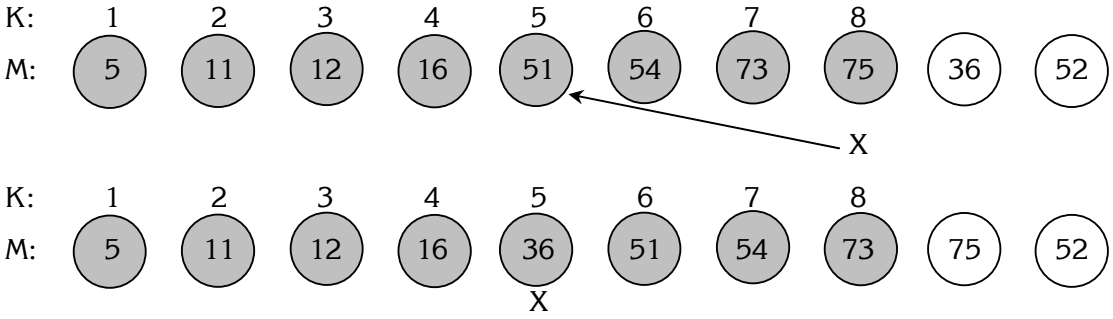
Lần 6: K = 6 X = M[K+1] = M[7] = 5 Pos = 1



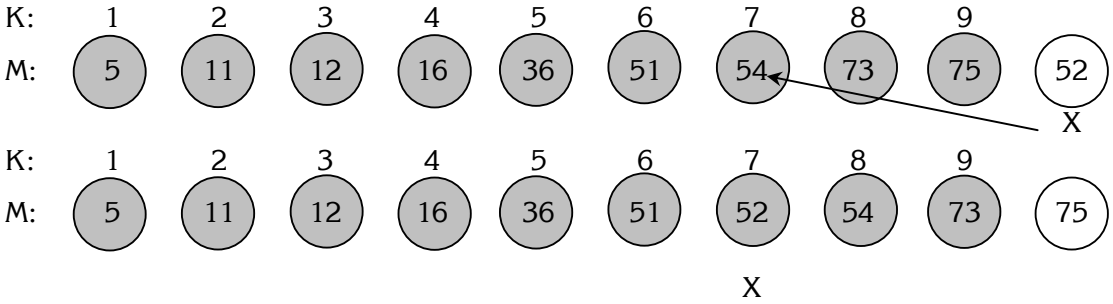
Lần 7: K = 7 X = M[K+1] = M[8] = 73 Pos = 7



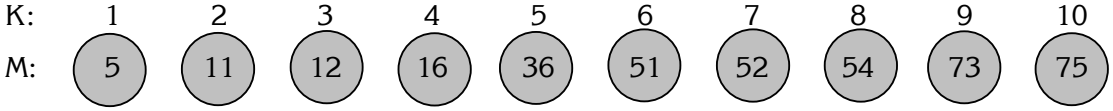
Lần 8: K = 8 X = M[K+1] = M[9] = 36 Pos = 5



Lần 9: K = 9 X = M[K+1] = M[10] = 52 Pos = 7



Thuật toán kết thúc: K = 10, mảng M đã được sắp xếp theo thứ tự tăng



- Phân tích thuật toán:

+ Trường hợp tốt nhất, khi mảng M ban đầu đã có thứ tự tăng:

Số phép gán: Gmin = 2×(N-1)
Số phép so sánh: Smin = 1+2+…+(N-1) = N×(N-1)/2
Số phép hoán vị: Hmin = 0

+ Trường hợp xấu nhất, khi mảng M ban đầu luôn có phần tử nhỏ nhất trong N-K phần tử còn lại đứng ở vị trí sau cùng sau mỗi lần hoán vị:

Số phép gán: Gmax = [2×(N-1)]+[1+2+…+(N-1)] = [2×(N-1)] + [N×(N-1)/2]
Số phép so sánh: Smax = (N-1)
Số phép hoán vị: Hmax = 0

+ Trung bình:

Số phép gán: Gavg = 2×(N-1) + [N×(N-1)/4]
Số phép so sánh: Savg = [N×(N-1)/2 + (N-1)]/2 = (N+2)×(N-1)/4
Số phép hoán vị: Havg = 0

+ Chúng ta nhận thấy rằng quá trình tìm kiếm vị trí chèn của phần tử K+1 và quá trình dời các phần tử từ vị trí chèn đến K ra phía sau 01 vị trí có thể kết hợp lại với nhau. Như vậy, quá trình di dời các phần tử ra sau này sẽ bắt đầu từ phần tử thứ K trở về đầu dãy M cho đến khi gặp phần tử có giá trị nhỏ hơn phần tử K+1 thì chúng ta đồng thời vừa di dời xong và đồng thời cũng bắt gặp vị trí chèn. Ngoài ra, chúng ta cũng có thể tính toán giá trị ban đầu cho K tùy thuộc vào số phần tử đứng đầu dãy M ban đầu có thứ tự tăng là bao nhiêu phần tử chứ không nhất thiết phải là 1. Khi đó, thuật toán sắp xếp chèn trực tiếp của chúng ta có thể được hiệu chỉnh lại như sau:

- Thuật toán hiệu chỉnh:

B1: K = 1
B2: IF (M[K] <= M[K+1] And K < N)
    B2.1: K++
    B2.2: Lặp lại B2
B3: IF (K = N)
    Thực hiện Bkt
B4: X = M[K+1]
B5: Pos = K
B6: IF (Pos > 0 And X < M[Pos])
    B6.1: M[Pos+1] = M[Pos]
    B6.2: Pos--
    B6.3: Lặp lại B6
B7: ELSE //Chèn X vào vị trí Pos+1
    B7.1: M[Pos+1] = X
    B7.2: K++
    B7.3: Lặp lại B3
Bkt: Kết thúc

- Cài đặt thuật toán hiệu chỉnh:

Hàm InsertionSort1 có prototype như sau:

void InsertionSort1(T M[], int N);

Hàm thực hiện việc sắp xếp N phần tử có kiểu dữ liệu T trên mảng M theo thứ tự tăng dựa trên thuật toán sắp xếp chèn trực tiếp đã hiệu chỉnh. Nội dung của hàm như sau:

```
void InsertionSort1(T M[], int N)
  { int K = 1, Pos;
    while(M[K-1] <= M[K] && K<N)
      K++;
    while (K < N)
      { T X = M[K];
        Pos = K-1;
        while (X < M[Pos] && Pos >= 0)
          { M[Pos+1] = M[Pos]; Pos--; }
        M[Pos+1] = X;
        K++;
      }
    return;
  }
```

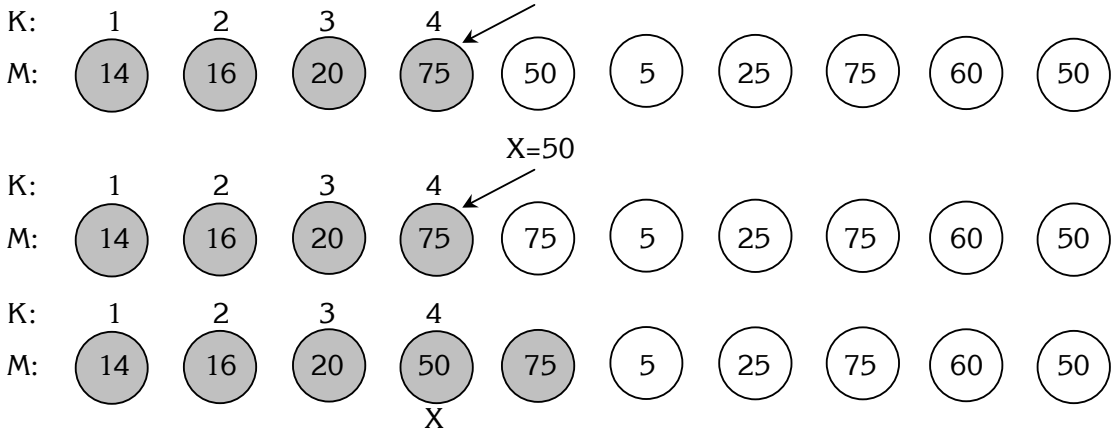
- Ví dụ minh họa thuật toán hiệu chỉnh:

Giả sử ta cần sắp xếp mảng M có 10 phần tử sau (N = 10):

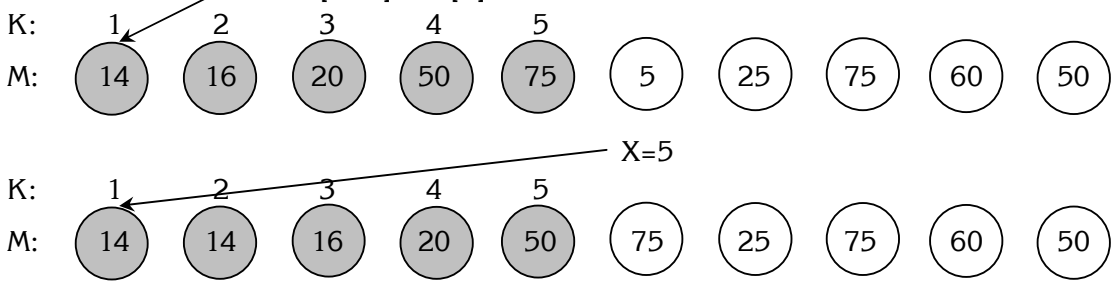
M: 14 16 20 75 50 5 25 75 60 50

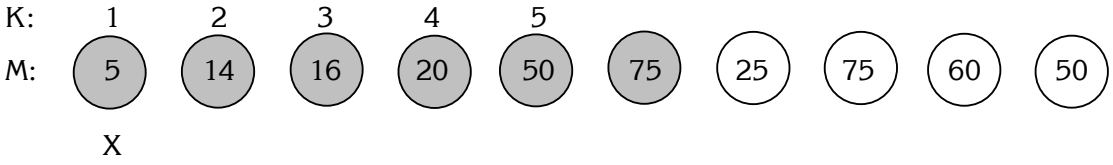
Ban đầu K = 4 nên ta sẽ thực hiện 6 lần chèn (N - 4 = 10 - 4 = 6) các phần tử vào dãy con đã có thứ tự tăng đứng đầu dãy M:

Lần 1: K = 4 X = M[K+1] = M[5] = 50 Pos = 3 => Pos + 1 = 4

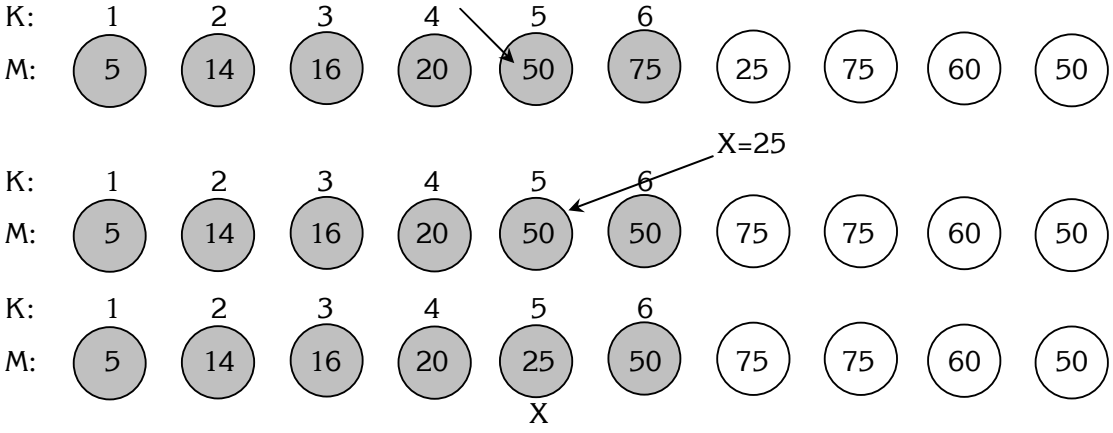


Lần 2: K = 5 X = M[K+1] = M[6] = 5 Pos = 0 => Pos + 1 = 1

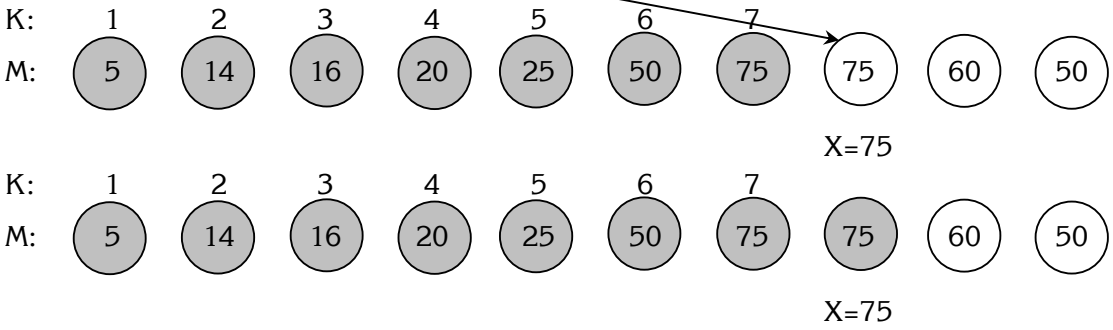




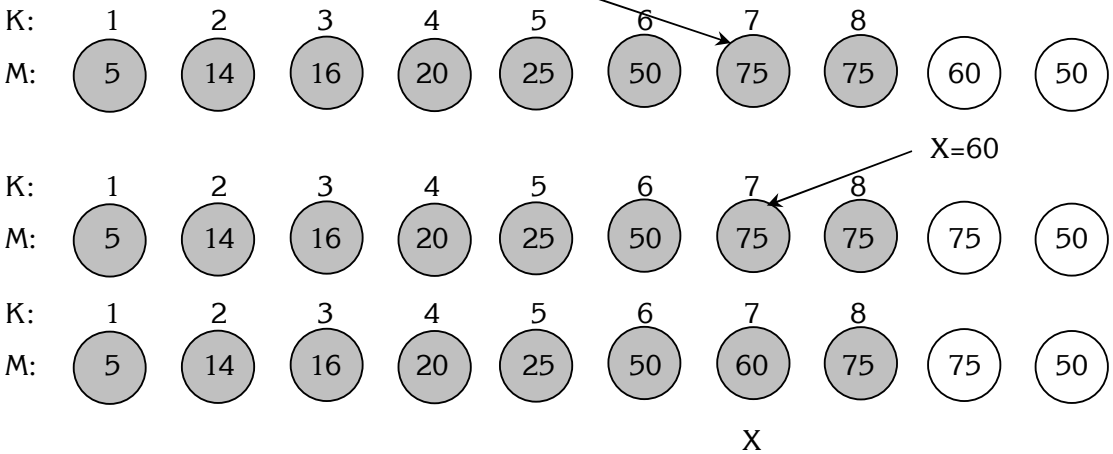
Lần 3: K = 6 X = M[K+1] = M[7] = 25 Pos = 4 => Pos + 1 = 5



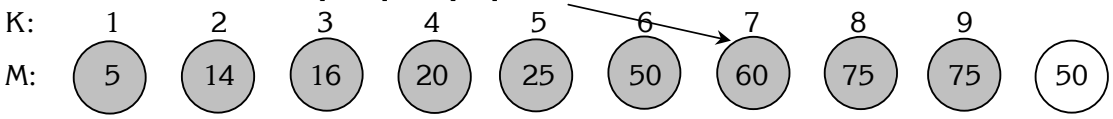
Lần 4: K = 7 X = M[K+1] = M[8] = 75 Pos = 7 => Pos + 1 = 8

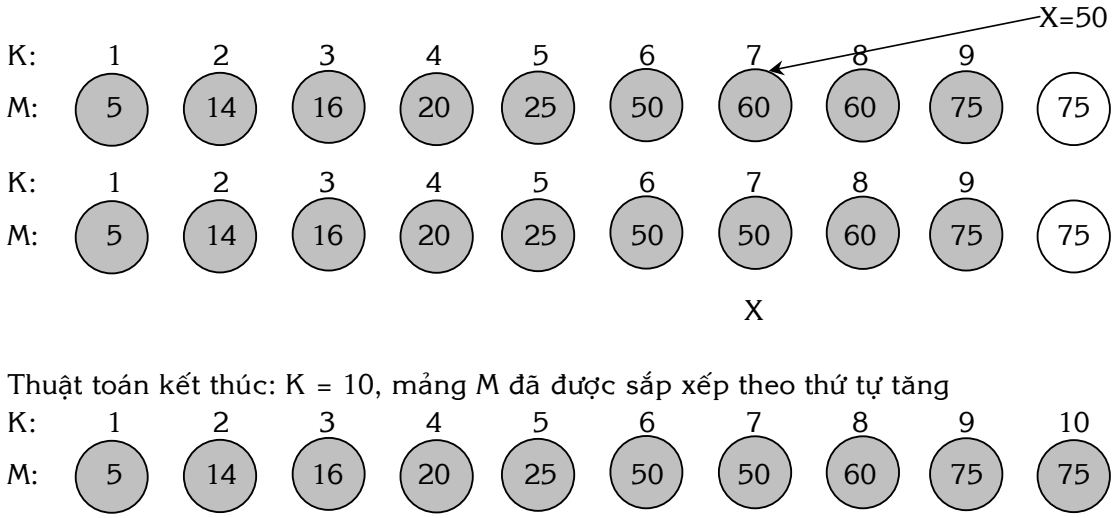


Lần 5: K = 8 X = M[K+1] = M[9] = 60 Pos = 6 => Pos + 1 = 7



Lần 6: K = 9 X = M[K+1] = M[10] = 50 Pos = 6 => Pos + 1 = 7





Thuật toán kết thúc: K = 10, mảng M đã được sắp xếp theo thứ tự tăng

- Phân tích thuật toán hiệu chỉnh:

+ Trường hợp tốt nhất, khi mảng M ban đầu đã có thứ tự tăng:

Số phép gán: Gmin = 1
Số phép so sánh: Smin = 2×(N-1) + 1
Số phép hoán vị: Hmin = 0

+ Trường hợp xấu nhất, khi mảng M ban đầu đã có thứ tự giảm dần:

Số phép gán: Gmax = 1+[1+2+…+(N-1)]+[N-1] = N×(N+1)/2
Số phép so sánh: Smax = 1+2×[1+2+…+(N-1)]+[N-1] = N^2
Số phép hoán vị: Hmax = 0

+ Trung bình:

Số phép gán: Gavg = [1+ N×(N-1)/2]/2
Số phép so sánh: Savg = [2×(N-1) + 1+N^2]/2
Số phép hoán vị: Havg = 0

3.2.4. Sắp xếp bằng phương pháp trộn (Merge Sort)

Các thuật toán trong phần này sẽ tìm cách tách mảng M thành các mảng con theo các đường chạy (run) rồi sau đó tiến hành nhập các mảng này lại theo từng cặp đường chạy để tạo thành các đường chạy mới có chiều dài lớn hơn đường chạy cũ. Sau một số lần tách/nhập thì cuối cùng mảng M chỉ còn lại 1 đường chạy, lúc đó thì các phần tử trên mảng M sẽ trở nên có thứ tự.

Các thuật toán sắp xếp bằng phương pháp trộn bao gồm:

- Thuật toán sắp xếp trộn thẳng hay trộn trực tiếp (straight merge sort),
- Thuật toán sắp xếp trộn tự nhiên (natural merge sort).

Trước khi đi vào chi tiết từng thuật toán chúng ta hãy tìm hiểu khái niệm và các vấn đề liên quan đến đường chạy (run)

- Đường chạy (Run):

Dãy M[I], M[I+1], …, M[J] (I ≤ J: 1 ≤ I, J ≤ N) là một đường chạy nếu nó có thứ tự.

- Chiều dài của đường chạy (Run's Length):

Số phần tử của một đường chạy còn được gọi là chiều dài của đường chạy.

Như vậy:

+ Mỗi phần tử của dãy là một đường chạy có chiều dài bằng 1.
+ Một dãy có thể bao gồm nhiều đường chạy.

- Trộn các đường chạy:

Khi ta trộn các đường chạy lại với nhau sẽ cho ra một đường chạy mới có chiều dài bằng tổng chiều dài các đường chạy ban đầu.

a. Thuật toán sắp xếp trộn trực tiếp hay trộn thẳng (Straight Merge Sort):

- Tư tưởng:

Ban đầu dãy M có N run(s) với chiều dài mỗi run: L = 1, ta tiến hành phân phối luân phiên N run(s) của dãy M về hai dãy phụ Temp1, Temp2 (Mỗi dãy phụ có N/2 run(s)). Sau đó trộn tương ứng từng cặp run ở hai dãy phụ Temp1, Temp2 thành một run mới có chiều dài L = 2 để đưa về M và dãy M trở thành dãy có N/2 run(s) với chiều dài mỗi run: L = 2.

Như vậy, sau mỗi lần phân phối và trộn các run trên dãy M thì số run trên dãy M sẽ giảm đi một nửa, đồng thời chiều dài mỗi run sẽ tăng gấp đôi. Do đó, sau $Log_2(N)$ lần phân phối và trộn thì dãy M chỉ còn lại 01 run với chiều dài là N và khi đó dãy M trở thành dãy có thứ tự.

Trong thuật giải sau, để dễ theo dõi chúng ta trình bày riêng 02 thuật giải:

+ Thuật giải phân phối luân phiên (tách) các đường chạy với chiều dài L trên dãy M về các dãy phụ Temp1, Temp2.
+ Thuật giải trộn (nhập) các cặp đường chạy trên Temp1, Temp2 có chiều dài L về M thành các đường chạy với chiều dài 2*L.

- Thuật toán phân phối:

B1: I = 1 //Chỉ số trên M
B2: J1 = 1 //Chỉ số trên Temp1
B3: J2 = 1 //Chỉ số trên Temp2
B4: IF (I > N) //Đã phân phối hết
    Thực hiện Bkt
//Chép 1 run từ M sang Temp1
B5: K = 1 //Chỉ số để duyệt các run
B6: IF (K > L) //Duyệt hết 1 run
    Thực hiện B13
B7: Temp1[J1] = M[I] //Chép các phần tử của run trên M sang Temp1
B8: I++
B9: J1++
B10: K++
B11: IF (I > N) //Đã phân phối hết
    Thực hiện Bkt
B12: Lặp lại B6
//Chép 1 run từ M sang Temp2

B13: K = 1
B14: IF (K > L)
Thực hiện B21
B15: Temp2[J2] = M[I] //Chép các phần tử của run trên M sang Temp2
B16: I++
B17: J2++
B18: K++
B19: IF (I > N) //Đã phân phối hết
Thực hiện Bkt
B20: Lặp lại B14
B21: Lặp lại B4
B22: N1 = J1-1 //Số phần tử trên Temp1
B23: N2 = J2-1 //Số phần tử trên Temp2
Bkt: Kết thúc

- Thuật toán trộn:

B1: I = 1 // Chỉ số trên M
B2: J1 = 1 //Chỉ số trên Temp1
B3: J2 = 1 //Chỉ số trên Temp2
B4: K1 = 1//Chỉ số để duyệt các run trên Temp1
B5: K2 = 1//Chỉ số để duyệt các run trên Temp2
B6: IF (J1 > N1) //Đã chép hết các phần tử trong Temp1
Thực hiện B25
B7: IF (J2 > N2) //Đã chép hết các phần tử trong Temp2
Thực hiện B30
B8: IF (Temp1[J1] ≤ Temp2[J2]) //Temp1[J1] đứng trước Temp2[J2] trên M
B8.1: M[I] = Temp1[J1]
B8.2: I++
B8.3: J1++
B8.4: K1++
B8.5: If (K1 > L) //Đã duyệt hết 1 run trong Temp1
Thực hiện B11
B8.6: Lặp lại B6
B9: ELSE //Temp2[J2] đứng trước Temp1[J1] trên M
B9.1: M[I] = Temp2[J2]
B9.2: I++
B9.3: J2++
B9.4: K2++
B9.5: If (K2 > L) //Đã duyệt hết 1 run trong Temp2
Thực hiện B18
B9.6: Lặp lại B6
B10: Lặp lại B4
//Chép phần run còn lại trong Temp2 về M
B11: IF (K2 > L) //Đã chép hết phần run còn lại trong Temp2 về M
Lặp lại B4
B12: M[I] = Temp2[J2]
B13: I++
B14: J2++

B15: K2++
B16: IF (J2 > N2) //Đã chép hết các phần tử trong Temp2
Thực hiện B30
B17: Lặp lại B11
//Chép phần run còn lại trong Temp1 về M
B18: IF (K1 > L) //Đã chép hết phần run còn lại trong Temp1 về M
Lặp lại B4
B19: M[I] = Temp1[J1]
B20: I++
B21: J1++
B22: K1++
B23: IF (J1 > N1)//Đã chép hết các phần tử trong Temp1
Thực hiện B25
B24: Lặp lại B18
//Chép các phần tử còn lại trong Temp2 về M
B25: IF (J2>N2)
Thực hiện Bkt
B26: M[I] = Temp2[J2]
B27: I++
B28: J2++
B29: Lặp lại B25
//Chép các phần tử còn lại trong Temp1 về M
B30: IF (J1>N1)
Thực hiện Bkt
B31: M[I] = Temp1[J1]
B32: I++
B33: J1++
B34: Lặp lại B30
Bkt: Kết thúc

- Thuật toán sắp xếp trộn thẳng:

B1: L = 1 //Chiều dài ban đầu của các run
B2: IF (L ≥ N) //Dãy chỉ còn 01 run
Thực hiện Bkt
B3: Phân_Phối(M, N, Temp1, N1, Temp2, N2, L)
B4: Trộn(Temp1, N1, Temp2, N2, M, N, L)
B5: L = 2*L
B6: Lặp lại B2
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm StraightMergeSort có prototype như sau:

```
void StraightMergeSort(T M[], int N);
```

Hàm thực hiện việc sắp xếp N phần tử có kiểu dữ liệu T trên mảng M theo thứ tự tăng dựa trên thuật toán sắp trộn trực tiếp. Hàm sử dụng các hàm Distribute, Merge có prototype và ý nghĩa như sau:

```
void Distribute(T M[], int N, T Temp1[], int &N1, T Temp2[], int &N2, int L);
```

Hàm thực hiện việc phân phối luân phiên các đường chạy có chiều dài L trên dãy M có N phần tử về thành các dãy Temp1 và Temp2 có tương ứng N1 và N2 phần tử.

void Merge(T Temp1[], int N1, T Temp2[], int N2, T M[], int &N, int L);

Hàm thực hiện việc trộn từng cặp tương ứng các đường chạy với độ dài L trên Temp1, Temp2 về dãy M thành các đường chạy có chiều dài 2*L.

Nội dung của các hàm như sau:

```
void Distribute(T M[], int N, T Temp1[], int &N1, T Temp2[], int &N2, int L)
  { int I = 0, J1 = 0, J2 = 0;
    while (I < N)
      { for(int K = 0; K<L && I<N; K++, I++, J1++)
          Temp1[J1] = M[I];
        for(K = 0; K<L && I<N; K++, I++, J2++)
          Temp2[J2] = M[I];
      }
    N1 = J1;
    N2 = J2;
    return;
  }
//=========================================================
void Merge(T Temp1[], int N1, T Temp2[], int N2, T M[], int &N, int L)
  { int I = 0, J1 = 0, J2 = 0, K1 = 0, K2 = 0;
    while (J1 < N1 && J2 < N2)
      { while (Temp1[J1] <= Temp2[J2] && J1 < N1 && J2 < N2)
          { M[I] = Temp1[J1];
            I++;
            J1++;
            if (J1 == N1)
              { for (; J2 < N2; J2++, I++)
                  M[I] = Temp2[J2];
                return;
              }
            K1++;
            if (K1 == L)
              { for (; K2 < L && J2 < N2; K2++, I++, J2++)
                  M[I] = Temp2[J2];
                K1 = K2 = 0;
                break;
              }
          }
        while (Temp2[J2] < Temp1[J1] && J1 < N1 && J2 < N2)
          { M[I] = Temp2[J2];
            I++;
            J2++;
            if (J2 == N2)
              { for (; J1 < N1; J1++, I++)
                  M[I] = Temp1[J1];
```

```
                        return;
                     }
                  K2++;
                  if (K2 == L)
                     {  for (; K1 < L && J1 < N1; K1++, I++, J1++)
                           M[I] = Temp1[J1];
                        K1 = K2 = 0;
                        break;
                     }
               }
         }
      while (J1 < N1)
         {  M[I] = Temp1[J1];
            I++;
            J1++;
         }
      while (J2 < N2)
         {  M[I] = Temp2[J2];
            I++;
            J2++;
         }
      N = N1 + N2;
      return;
   }

//=========================================================

void  StraightMergeSort(T  M[], int  N)
   {  int  L = 1, N1 = 0, N2 = 0;
      T * Temp1 = new  T[N];
      T * Temp2 = new  T[N];
      if (Temp1 == NULL || Temp2 == NULL)
         return;
      while (L < N)
         {  Distribute(M, N, Temp1, N1, Temp2, N2, L);
            Merge(Temp1, N1, Temp2, N2, M, N, L);
            L = 2*L;
         }
      delete  Temp1;
      delete  Temp2;
      return;
   }
```

- Ví dụ minh họa thuật toán:

Giả sử ta cần sắp xếp mảng M có 10 phần tử sau (N = 10):

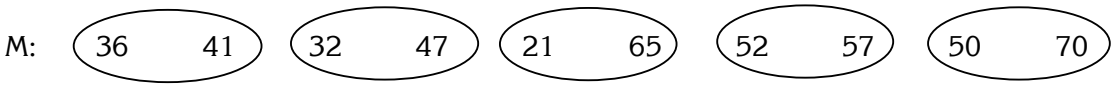
M: 41 36 32 47 65 21 52 57 70 50

Ta thực hiện các lần phân phối và trộn các phần tử của M như sau:

Lần 1: L = 1

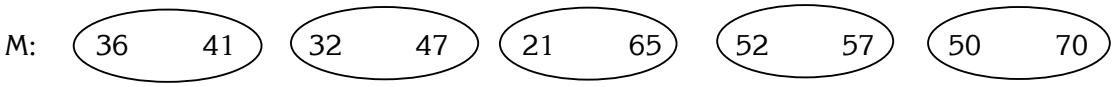
Phân phối M thành Temp1, Temp2:

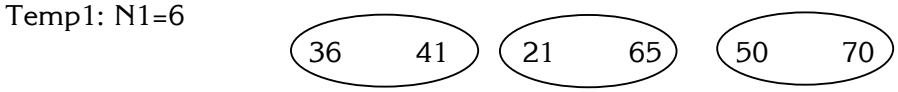
Trộn Temp1, Temp2 thành M:

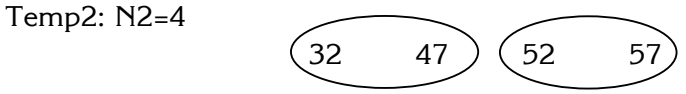


Lần 2: L = 2

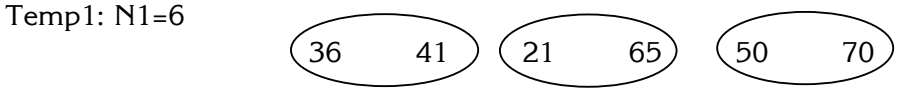
Phân phối M thành Temp1, Temp2:

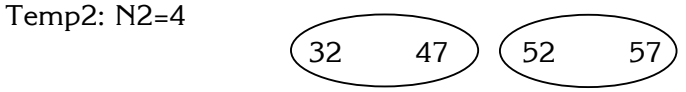


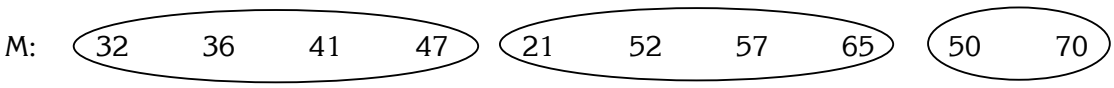




Trộn Temp1, Temp2 thành M:







Lần 3: L = 4

Phân phối M thành Temp1, Temp2:

M: 32 36 41 47 21 52 57 65 50 70

Temp1:N1=6 32 36 41 47 50 70

Temp2: N2=4 21 52 57 65

Trộn Temp1, Temp2 thành M:

Temp1:N1=6 32 36 41 47 50 70

Temp2: N2=4 21 52 57 65

M: 21 32 36 41 47 52 57 65 50 70

Lần 4: L = 8

Phân phối M thành Temp1, Temp2:

M: 21 32 36 41 47 52 57 65 50 70

Temp1: N1=8 21 32 36 41 47 52 57 65

Temp2: N2=2 50 70

Trộn Temp1, Temp2 thành M:

Temp1: N1=8 21 32 36 41 47 52 57 65

Temp2: N2=2 50 70

M: 21 32 36 41 47 50 52 57 65 70

L = 16 > 10: Kết thúc thuật toán

- Phân tích thuật toán:

+ Trong thuật giải này chúng ta luôn thực hiện $\log_2(N)$ lần phân phối và trộn các run.

+ Ở mỗi lần phân phối run chúng ta phải thực hiện: N phép gán và 2N phép so sánh (N phép so sánh hết đường chạy và N phép so sánh hết dãy).

+ Ở mỗi lần trộn run chúng ta cũng phải thực hiện: N phép gán và 2N+N/2 phép so sánh (N phép so sánh hết đường chạy, N phép so sánh hết dãy và N/2 phép so sánh giá trị các cặp tương ứng trên 2 dãy phụ).

+ Trong mọi trường hợp:

Số phép gán: $G = 2N \times Log_2(N)$
Số phép so sánh: $S = (4N+N/2) \times Log_2(N)$
Số phép hoán vị: Hmin = 0

+ Trong thuật giải này chúng ta sử dụng 02 dãy phụ, tuy nhiên tổng số phần tử ở 02 dãy phụ này cũng chỉ bằng N, do vậy đã tạo ra sự lãng phí bộ nhớ không cần thiết. Để giải quyết vấn đề này chúng ta chỉ cần sử dụng 01 dãy phụ song chúng ta kết hợp quá trình trộn các cặp run có chiều dài L tương ứng ở hai đầu dãy thành các run có chiều dài 2L và phân phối luân phiên về hai đầu của một dãy phụ. Sau đó chúng ta đổi vai trò của 02 dãy này cho nhau.

+ Trước khi hiệu chỉnh lại thuật giải chúng ta xét dãy M gồm 10 phần tử sau để minh họa cho quá trình này:

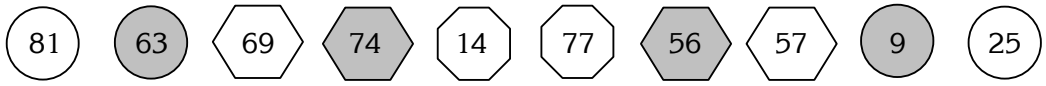
Giả sử ta cần sắp xếp mảng M có 10 phần tử sau (N = 10):

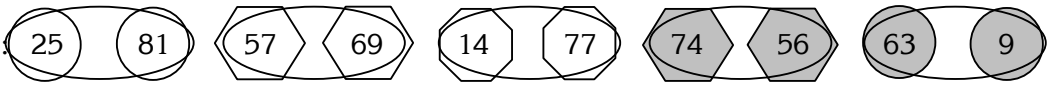
M: 81 63 69 74 14 77 56 57 9 25

Ta thực hiện các lần trộn các cặp run ở hai đầu dãy này và kết hợp phân phối các run mới trộn về hai đầu dãy kia như sau:

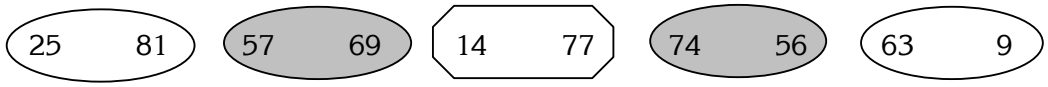
Lần 1: L = 1

Trộn các cặp run có chiều dài L = 1 trên M thành các run có chiều dài L = 2 và kết hợp phân phối luân phiên các run này về hai đầu dãy Tmp:



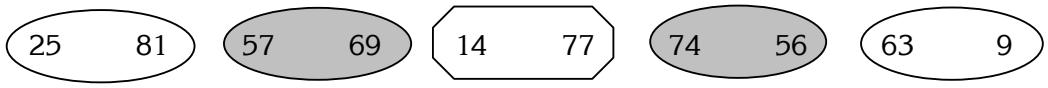


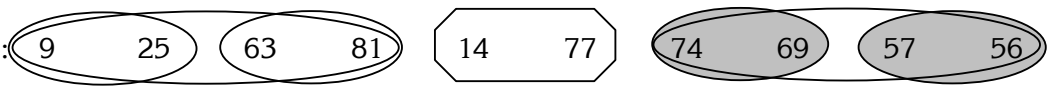
Đổi vai trò của M và Tmp cho nhau



Lần 2: L = 2

Trộn các cặp run có chiều dài L = 2 trên M thành các run có chiều dài L = 4 và kết hợp phân phối luân phiên các run này về hai đầu dãy Tmp:





Đổi vai trò của M và Tmp cho nhau

M: 9 25 63 81 14 77 74 69 57 56

Lần 3: L = 4

Trộn các cặp run có chiều dài L = 4 trên M thành các run có chiều dài L = 8 và kết hợp phân phối luân phiên các run này về hai đầu dãy Tmp:

M: 9 25 63 81 14 77 74 69 57 56

Tmp: 9 25 56 57 63 69 74 81 77 14

Đổi vai trò của M và Tmp cho nhau

M: 9 25 56 57 63 69 74 81 77 14

Lần 4: L = 8

Trộn các cặp run có chiều dài L = 4 trên M thành các run có chiều dài L = 8 và kết hợp phân phối luân phiên các run này về hai đầu dãy Tmp:

M: 9 25 56 57 63 69 74 81 77 14

Tmp: 9 14 25 56 57 63 69 74 77 81

Đổi vai trò của M và Tmp cho nhau

M: 9 14 25 56 57 63 69 74 77 81

L = 16 > 10: Kết thúc thuật toán

+ Như vậy, trong thuật giải này chúng ta chỉ còn thao tác trộn các cặp run có chiều dài L tương ứng ở hai đầu dãy thành một run mới có chiều dài 2L để đưa về dãy phụ. Vấn đề là chúng ta sẽ luân phiên đặt run mới vào đầu dãy phụ (theo thứ tự tăng) và cuối dãy phụ (theo thứ tự giảm). Tức là hai bước trộn và phân phối đã được kết hợp lại với nhau. Thuật giải hiệu chỉnh như sau:

- Thuật toán Trộn – Phân phối các cặp đường chạy:

B1: I1 = 1 // Chỉ số từ đầu dãy M
B2: I2 = N // Chỉ số từ cuối dãy M
B3: J1 = 1 // Chỉ số từ đầu dãy Temp
B4: J2 = N// Chỉ số từ cuối dãy Temp
B5: Head = True // Cờ báo phía đặt run mới trong quá trình trộn - phân phối
B6: K1 = 1// Chỉ số để duyệt các run đầu dãy M
B7: K2 = 1// Chỉ số để duyệt các run cuối dãy M
B8: IF (I1 > I2) // Đã trộn và phân phối hết các run

Thực hiện Bkt
B9: IF (M[I1] ≤ M[I2]) // M[I1] đứng trước M[I2] trên Temp
B9.1: If (Head = True)
B9.1.1: Temp[J1] = M[I1]
B9.1.2: J1++
B9.2: Else
B9.2.1: Temp[J2] = M[I1]
B9.2.2: J2--
B9.3: I1++
B9.4: If (I1 > I2)
Thực hiện Bkt
B9.5: K1++
B9.6: If (K1 > L) //Đã duyệt hết 1 run phía đầu trong M
Thực hiện B11
B9.7: Lặp lại B9
B10: ELSE // M[I2] đứng trước M[I1] trên Temp
B10.1: If (Head = True)
B10.1.1: Temp[J1] = M[I2]
B10.1.2: J1++
B10.2: Else
B10.2.1: Temp[J2] = M[I2]
B10.2.2: J2--
B10.3: I2--
B10.4: If (I1 > I2)
Thực hiện Bkt
B10.5: K2++
B10.6: If (K2 > L) //Đã duyệt hết 1 run phía sau trong M
Thực hiện B18
B10.7: Lặp lại B9

//Chép phần run còn lại ở phía sau trong M về Temp

B11: IF (K2 > L) //Đã chép hết phần run còn lại ở phía sau trong M về Temp
B11.1: Head = Not(Head)
B11.2: Lặp lại B6
B12: IF (Head = True)
B12.1: Temp[J1] = M[I2]
B12.2: J1++
B13: ELSE
B13.1: Temp[J2] = M[I2]
B13.2: J2--
B14: I2--
B15: If (I1 > I2)
Thực hiện Bkt
B16: K2++
B17: Lặp lại B11

//Chép phần run còn lại ở phía trước trong M về Temp

B18: IF (K1 > L) //Đã chép hết phần run còn lại ở phía trước trong M về Temp

B18.1: Head = Not(Head)
B18.2: Lặp lại B6
B19: IF (Head = True)
B19.1: Temp[J1] = M[I1]
B19.2: J1++
B20: ELSE
B20.1: Temp[J2] = M[I1]
B20.2: J2--
B21: I1++
B22: If (I1 > I2)
Thực hiện Bkt
B23: K1++
B24: Lặp lại B18
Bkt: Kết thúc

- Thuật toán sắp xếp trộn thẳng hiệu chỉnh:

B1: L = 1 //Chiều dài ban đầu của các run
B2: IF (L ≥ N) //Dãy chỉ còn 01 run
Thực hiện Bkt
B3: Trộn_Phân_Phối(M, N, Temp, L)
B4: L = 2*L
B5: IF (L > N)
// Chép các phần tử từ Temp về M
B5.1: I = 1
B5.2: If (I > N)
Thực hiện Bkt
B5.3: M[I] = Temp[I]
B5.4: I++
B5.5: Lặp lại B5.2

B6: Trộn_Phân_Phối(Temp, N, M, L)
B7: L = 2*L
B8: Lặp lại B2
Bkt: Kết thúc

- Cài đặt thuật toán hiệu chỉnh:

Hàm StraightMergeSortModify có prototype như sau:

```
void StraightMergeSortModify(T M[], int N);
```

Hàm thực hiện việc sắp xếp N phần tử có kiểu dữ liệu T trên mảng M theo thứ tự tăng dựa trên thuật toán sắp trộn trực tiếp đã hiệu chỉnh. Hàm sử dụng hàm MergeDistribute có prototype và ý nghĩa như sau:

```
void MergeDistribute(T M[], int N, T Temp[], int L);
```

Hàm thực hiện việc trộn các cặp run có chiều dài L ở hai đầu dãy M thành một run có chiều dài 2L và phân phối luân phiên các run có chiều dài 2L này về hai đầu dãy Temp. Nội dung của hàm như sau:

```
void MergeDistribute(T M[], int N, T Temp[], int L)
```

```
{  int  I1 = 0, I2 = N-1, J1 = 0, J2 = N-1, K1 = 0, K2 = 0, Head = 1;
   while (I1 <= I2)
      {  while (M[I1] <= M[I2] && I1 <= I2)
            {  if (Head == 1)
                  {  Temp[J1] = M[I1];
                     J1++;
                  }
               else
                  {  Temp[J2] = M[I1];
                     J2--;
                  }
               I1++;
               if (I1 > I2)
                  break;
               K1++;
               if (K1 == L)
                  {  for (; K2 < L && I1 <= I2; K2++, I2--)
                        if (Head == 1)
                           {  Temp[J1] = M[I2];
                              J1++;
                           }
                        else
                           {  Temp[J2] = M[I2];
                              J2--;
                           }
                     Head = 0-Head;
                     K1 = K2 = 0;
                     break;
                  }
            }
         while (M[I2] <= M[I1] && I1 <= I2)
            {  if (Head == 1)
                  {  Temp[J1] = M[I2];
                     J1++;
                  }
               else
                  {  Temp[J2] = M[I2];
                     J2--;
                  }
               I2--;
               if (I1 > I2)
                  break;
               K2++;
               if (K2 == L)
                  {  for (; K1 < L && I1<= I2; K1++, I1++)
                        if (Head == 1)
                           {  Temp[J1] = M[I1];
                              J1++;
```

```
                        }
                    else
                        {  Temp[J2] = M[I1]
                           J2--;
                        }
                    Head = 0-Head;
                    K1 = K2 = 0;
                    break;
                }
            }
        }
    return;
}

//=========================================================

void  StraightMergeSortModify(T  M[], int  N)
  {  int  L = 1 ;
     T * Temp = new  T[N];
     if (Temp == NULL)
        return;
     while (L < N)
        {  MergeDistribute(M, N, Temp, L);
           L = 2*L;
           if (L >= N)
              {  for (int I = 0; I < N; I++)
                     M[I] = Temp[I];
                 break;
              }
           MergeDistribute(Temp, N, M, L);
           L = 2*L;
        }
     delete  Temp;
     return;
  }
```

- Phân tích thuật toán hiệu chỉnh:

+ Trong thuật giải này chúng ta luôn thực hiện $\log_2(N)$ lần trộn - phân phối các run.

+ Mỗi lần trộn-phân phối chúng ta phải thực hiện: N phép gán và N+N/2+N/2=2N phép so sánh.

+ Trong mọi trường hợp:

Số phép gán: $G = N \times \log_2(N)$
Số phép so sánh: $S = 2N \times \log_2(N)$
Số phép hoán vị: Hmin = 0

+ Như vậy thuật giải trộn thẳng hiệu chỉnh vừa tiết kiệm bộ nhớ, vừa thực hiện nhanh hơn thuật giải trộn thẳng ban đầu.

+ Tuy nhiên, trong thuật giải trộn thẳng chúng ta đã thực hiện việc phân phối và trộn các cặp đường chạy có chiều dài cố định mà trong thực tế trên dãy các đường

chạy có thể có chiều dài lớn hơn. Điều này sẽ giảm bớt số lần phân phối và trộn các cặp đường chạy cho chúng ta. Thuật giải trộn tự nhiên được trình bày sau đây sẽ loại bỏ được nhược điểm này của thuật giải trộn thẳng.

b. Thuật toán sắp xếp trộn tự nhiên (Natural Merge Sort):

- Tư tưởng:

Tận dụng các đường chạy tự nhiên có sẵn trên dãy, tiến hành trộn tương ứng các cặp đường chạy tự nhiên nằm hai đầu dãy M thành một đường chạy mới và phân phối luân phiên các đường chạy mới này về hai đầu dãy phụ Temp. Sau đó lại tiếp tục trộn tương ứng từng cặp run ở hai đầu dãy phụ Temp thành một run mới và phân phối luân phiên run mới này về hai đầu dãy M. Cứ tiếp tục như vậy cho đến khi trên M hoặc trên Temp chỉ còn lại 01 run thì kết thúc.

- Thuật toán Trộn – Phân phối các cặp đường chạy tự nhiên:

B1: I1 = 1 // Chỉ số từ đầu dãy M
B2: I2 = N // Chỉ số từ cuối dãy M
B3: J1 = 1 // Chỉ số từ đầu dãy Temp
B4: J2 = N// Chỉ số từ cuối dãy Temp

B5: Head = True // Cờ báo phía đặt run mới trong quá trình trộn - phân phối

B6: IF (I1 > I2) // Đã trộn và phân phối hết các run
    Thực hiện Bkt

B7: IF (M[I1] ≤ M[I2]) // M[I1] đứng trước M[I2] trên Temp
    B7.1: If (Head = True)
        B7.1.1: Temp[J1] = M[I1]

        B7.1.2: J1++
    B7.2: Else
        B7.2.1: Temp[J2] = M[I1]
        B7.2.2: J2--
    B7.3: I1++
    B7.4: If (I1 > I2)
        Thực hiện Bkt
    B7.5: If (M[I1] < M[I1-1]) //Đã duyệt hết 1 run phía đầu trong M
        Thực hiện B9
    B7.6: Lặp lại B7
B8: ELSE // M[I2] đứng trước M[I1] trên Temp
    B8.1: If (Head = True)
        B8.1.1: Temp[J1] = M[I2]
        B8.1.2: J1++
    B8.2: Else
        B8.2.1: Temp[J2] = M[I2]
        B8.2.2: J2--
    B8.3: I2--
    B8.4: If (I1 > I2)
        Thực hiện Bkt

B8.5: If (M[I2] < M[I2+1]) //Đã duyệt hết 1 run phía sau trong M
Thực hiện B15
B8.6: Lặp lại B7

//Chép phần run còn lại ở phía sau trong M về Temp

B9: IF (M[I2] < M[I2+1]) //Đã chép hết phần run còn lại ở phía sau trong M về Temp
B9.1: Head = Not(Head)
B9.2: Lặp lại B6
B10: IF (Head = True)
B10.1: Temp[J1] = M[I2]
B10.2: J1++
B11: ELSE
B11.1: Temp[J2] = M[I2]
B11.2: J2--
B12: I2--
B13: IF (I1> I2)
Thực hiện Bkt
B14: Lặp lại B9

//Chép phần run còn lại ở phía trước trong M về Temp

B15: IF (M[I1]< M[I1-1]) //Đã chép hết phần run còn lại phía trước trong M về Temp
B15.1: Head = Not(Head)
B15.2: Lặp lại B6
B16: IF (Head = True)
B16.1: Temp[J1] = M[I1]
B16.2: J1++
B17: ELSE
B17.1: Temp[J2] = M[I1]
B17.2: J2--
B18: I1++
B19: IF (I1> I2)
Thực hiện Bkt
B20: Lặp lại B15
Bkt: Kết thúc

- Thuật toán sắp xếp trộn tự nhiên:

B1: L = 1 //Khởi tạo chiều dài ban đầu của run đầu tiên
//Tìm chiều dài ban đầu của run đầu tiên
B2: IF (N < 2)
B2.1: L=N
B2.2: Thực hiện Bkt
B3: IF (M[L] ≤ M[L+1] And L < N)
B3.1: L++
B3.2: Lặp lại B3
B4: IF (L = N) //Dãy chỉ còn 01 run
Thực hiện Bkt
B5: Trộn_Phân_Phối(M, N, Temp, L)
B6: IF (L = N)

// Chép các phần tử từ Temp về M
B6.1: I = 1
B6.2: If (I > N)
Thực hiện Bkt
B6.3: M[I] = Temp[I]
B6.4: I++
B6.5: Lặp lại B6.2
B7: Trộn_Phân_Phối(Temp, N, M, L)
B8: Lặp lại B4
Bkt: Kết thúc

- Cài đặt thuật toán trộn tự nhiên:

Hàm NaturalMergeSort có prototype như sau:

void NaturalMergeSort(T M[], int N);

Hàm thực hiện việc sắp xếp N phần tử có kiểu dữ liệu T trên mảng M theo thứ tự tăng dựa trên thuật toán sắp trộn trực tự nhiên. Hàm sử dụng hàm NaturalMergeDistribute có prototype và ý nghĩa như sau:

void NaturalMergeDistribute(T M[], int N, T Temp[], int &L);

Hàm thực hiện việc trộn các cặp run ở hai đầu dãy M mà run đầu tiên có chiều dài L thành một run mới chiều dài lớn hơn hoặc bằng L và phân phối luân phiên run mới này về hai đầu dãy Temp. Nội dung của hàm như sau:

```
void NaturalMergeDistribute(T M[], int N, T Temp[], int &L)
  { int I1 = 0, I2 = N-1, J1 = 0, J2 = N-1, Head = 1, FirstPair = 1;
    while (I1 < I2)
      { while (M[I1] <= M[I2] && I1 < I2)
          { if (Head == 1)
              { Temp[J1] = M[I1];
                J1++;
              }
            else
              { Temp[J2] = M[I1];
                J2--;
              }
            I1++;
            if (M[I1] < M[I1-1])
              { while (M[I2] <= M[I2-1] && I2 > I1)
                  { if (Head == 1)
                      { Temp[J1] = M[I2];
                        J1++;
                        if (FirstPair == 1)
                          L++;
                      }
                    else
                      { Temp[J2] = M[I2];
                        J2--;
                      }
```

```
                I2--;
            }
        if (Head == 1)
            { Temp[J1] = M[I2];
              J1++;
              If (FirstPair == 1)
                 L++;
            }
        else
            { Temp[J2] = M[I2];
              J2--;
            }
        I2--;
        FirstPair = 0;
        if (I1 > I2)
           return;
        Head = 0 – Head;
        break;
      }
  }
if (I1 == I2)
  { Temp[J1] = M[I1];
    if (I1 == N-1)
       L = N;
    return;
  }
while (M[I2] <= M[I1] && I1 < I2)
  { if (Head == 1)
      { Temp[J1] = M[I2];
        J1++;
        if (FirstPair == 1)
           L++;
      }
    else
      { Temp[J2] = M[I2];
        J2--;
      }
    I2--;
    if (M[I2] < M[I2+1])
      { while (M[I1] <= M[I1+1] && I1 < I2)
          { if (Head == 1)
              { Temp[J1] = M[I1];
                J1++;
              }
            else
              { Temp[J2] = M[I1];
                J2--;
              }
```

```
                        I1++;
                    }
                if (Head == 1)
                    { Temp[J1] = M[I1];
                      J1++;
                    }
                else
                    { Temp[J2] = M[I1];
                      J2--;
                    }
                I1++;
                FirstPair = 0;
                if (I1 > I2)
                   return;
                Head = 0 – Head;
                break;
              }
          }
        if (I1 == I2)
          { Temp[J1] = M[I1];
            if (I1 == N-1)
               L = N;
            return;
          }
      }
    return;
  }
//=========================================================
void NaruralMergeSort1(T M[], int N)
  { int L = 1 ;
    if (N < 2)
      return;
    while (M[L-1] < M[L] && L<N)
      L++;
    T * Temp = new T[N];
    if (Temp == NULL)
      return;
    while (L < N)
      { NaturalMergeDistribute(M, N, Temp, L);
        if (L == N)
          { for (int I = 0; I < N; I++)
              M[I] = Temp[I];
            break;
          }
        NaturalMergeDistribute(Temp, N, M, L);
      }
    delete Temp;
    return;
```

}

- Ví dụ minh họa thuật toán:

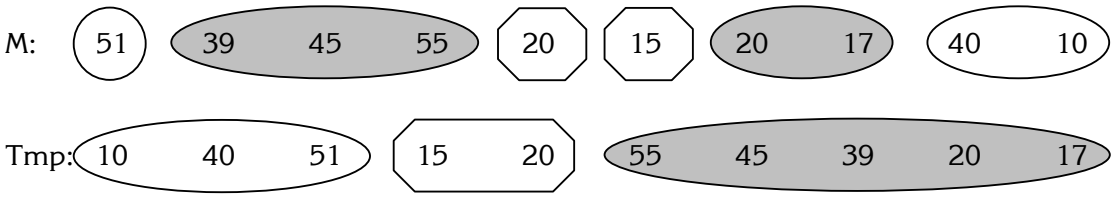
Giả sử ta cần sắp xếp mảng M có 10 phần tử sau (N = 10):

M: 51 39 45 55 20 15 20 17 40 10

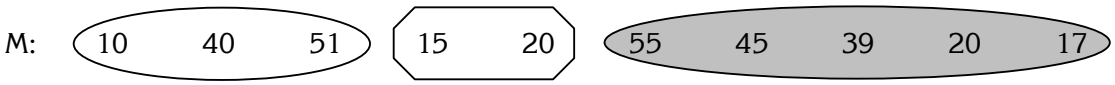
Ta thực hiện các lần trộn các cặp run tự nhiên ở hai đầu dãy này và kết hợp phân phối các run mới trộn về hai đầu dãy kia như sau:

Lần 1: L = 1

Trộn các cặp run tự nhiên có chiều dài L1 = 1 và L2 = 2 trên M thành các run có chiều dài L = 3 và kết hợp phân phối luân phiên các run này về hai đầu dãy Tmp:

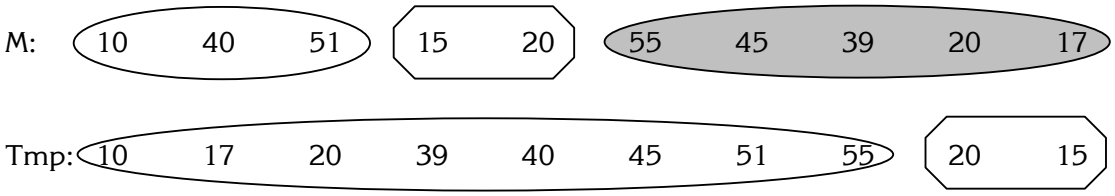


Đổi vai trò của M và Tmp cho nhau

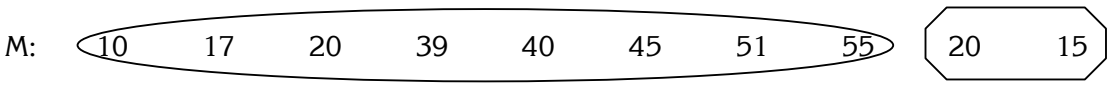


Lần 2: L = 3

Trộn các cặp run tự nhiên có chiều dài L1 = 3 và L2 = 5 trên M thành các run có chiều dài L = 8 và kết hợp phân phối luân phiên các run này về hai đầu dãy Tmp:

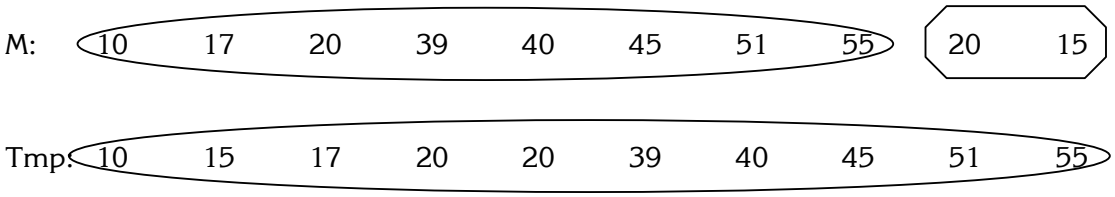


Đổi vai trò của M và Tmp cho nhau

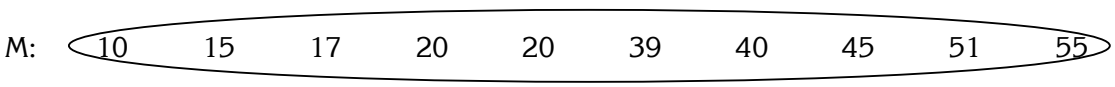


Lần 3: L = 8

Trộn các cặp run tự nhiên có chiều dài L1 = 8 và L2 = 2 trên M thành các run có chiều dài L = 10 và kết hợp phân phối luân phiên các run này về hai đầu dãy Tmp:



Đổi vai trò của M và Tmp cho nhau



L = 10: Kết thúc thuật toán

- Phân tích thuật toán trộn tự nhiên:

+ Trong trường hợp tốt nhất, khi dãy có thứ tự tăng thì chúng ta không phải qua bước phân phối và trộn nào hết:

  Số phép gán: Gmin = 1
  Số phép so sánh: Smin = 2(N-1) + 2 = 2N
  Số phép hoán vị: Hmin = 0

+ Trong trường hợp xấu nhất, khi dãy có thứ tự giảm ở nửa đầu và có thứ tự tăng ở nửa cuối và ở mỗi bước trộn phân phối thì độ dài đường chạy mới cũng chỉ tăng gấp đôi. Trong trường hợp này sẽ giống như thuật toán trộn thẳng đã hiệu chỉnh:

  Số phép gán: Gmax = $N \times Log_2(N)+1$
  Số phép so sánh: Smax = $2N \times Log_2(N)+2$
  Số phép hoán vị: Hmin = 0

+ Trung bình:

  Số phép gán: Gavg = $N \times Log_2(N)/2+1$
  Số phép so sánh: Savg = $N \times Log_2(N) + N + 1$
  Số phép hoán vị: Havg = 0

☛ **Lưu ý**:

+ Trong thuật toán này chúng ta cũng có thể sử dụng 2 dãy phụ Temp1, Temp2 như trong thuật toán trộn trực tiếp, khi đó chúng ta luôn luôn duyệt từ đầu đến cuối các dãy chứ không cần thiết phải duyệt từ hai đầu dãy vào giữa. Tuy nhiên, trong trường này thì bộ nhớ trung gian sẽ tốn nhiều hơn.

+ Trong các thuật toán sắp xếp theo phương pháp trộn, việc sử dụng nhiều dãy phụ có thể làm giảm bớt số lần phân phối và trộn các run. Tuy nhiên, việc quản lý các dãy phụ sẽ phức tạp hơn.

3.3. Các giải thuật sắp xếp ngoại (Sắp xếp trên tập tin)

Ở đây, do số phần tử dữ liệu thường lớn nên một phần dữ liệu cần sắp xếp được đưa vào trong bộ nhớ trong (RAM), phần còn lại được lưu trữ ở bộ nhớ ngoài (DISK). Do vậy, tốc độ sắp xếp dữ liệu trên tập tin tương đối chậm. Các giải thuật sắp xếp ngoại bao gồm các nhóm sau:

- Sắp xếp bằng phương pháp trộn (merge sort),
- Sắp xếp theo chỉ mục (index sort).

Như vậy trong phần này chúng ta tìm cách sắp xếp tập tin F có N phần tử dữ liệu có kiểu T (khóa nhận diện các phần tử dữ liệu có kiểu T) theo thứ tự tăng.

3.3.1. Sắp xếp bằng phương pháp trộn (Merge Sort)

Tương tự như đối với sắp xếp theo phương pháp trộn trên mảng, trong các thuật giải ở đây chúng ta sẽ tìm cách phân phối các đường chạy trong tập tin dữ liệu về các tập tin trung gian và sau đó lại trộn tương ứng các cặp đường chạy trên các tập tin trung gian thành một đường chạy mới có chiều dài lớn hơn.

Các thuật toán sắp xếp bằng phương pháp trộn trên tập tin bao gồm:

- Thuật toán sắp xếp trộn thẳng hay trộn trực tiếp (straight merge sort),
- Thuật toán sắp xếp trộn tự nhiên (natural merge sort),
- Thuật toán trộn đa lối cân bằng (multiways merge sort),
- Thuật toán trộn đa pha (multiphases merge sort).

Ở đây chúng ta chỉ nghiên cứu hai thuật toán trộn đầu tiên.

a. Thuật toán sắp xếp trộn trực tiếp (Straight Merge Sort):

- Tư tưởng:

Tương tự như thuật toán trộn trực tiếp trên mảng, ban đầu tập tin Fd có N run(s) với chiều dài mỗi run: L = 1, ta tiến hành phân phối luân phiên N run(s) của tập tin Fd về K tập tin phụ Ft1, Ft2, ..., FtK (Mỗi tập tin phụ có N/K run(s)). Sau đó trộn tương ứng từng bộ K run(s) ở K tập tin phụ Ft1, Ft2, ..., FtK thành một run mới có chiều dài L = K để đưa về tập tin Fd và tập tin Fd trở thành tập tin có N/K run(s) với chiều dài mỗi run: L = K.

Như vậy, sau mỗi lần phân phối và trộn các run trên tập tin Fd thì số run trên tập tin Fd sẽ giảm đi K lần, đồng thời chiều dài mỗi run trên Fd sẽ tăng lên K lần. Do đó, sau $Log_K(N)$ lần phân phối và trộn thì tập tin Fd chỉ còn lại 01 run với chiều dài là N và khi đó tập tin Fd trở thành tập tin có thứ tự.

Trong thuật giải này, để dễ theo dõi chúng ta sử dụng 2 tập tin phụ (K = 2) và quá trình phân phối, trộn các run được trình bày riêng thành 2 thuật giải:

+ Thuật giải phân phối luân phiên (tách) các đường chạy với chiều dài L trên tập tin Fd về hai tập tin phụ Ft1, Ft2;
+ Thuật giải trộn (nhập) các cặp đường chạy trên hai tập tin Ft1, Ft2 có chiều dài L về tập tin Fd thành các đường chạy với chiều dài 2*L;

Giả sử rằng các lỗi thao tác trên tập tin sẽ bị bỏ qua trong quá trình thực hiện.

- Thuật toán phân phối:

```
B1: Fd = fopen(DataFile, "r")    //Mở tập tin dữ liệu cần sắp xếp để đọc dữ liệu
B2: Ft1 = fopen(DataTemp1, "w")      //Mở tập tin trung gian thứ nhất để ghi dữ liệu
B3: Ft2 = fopen(DataTemp2, "w")      //Mở tập tin trung gian thứ hai để ghi dữ liệu
B4: IF (feof(Fd)) //Đã phân phối hết
      Thực hiện Bkt
//Chép 1 run từ Fd sang Ft1
B5: K = 1  //Chỉ số đếm để duyệt các run
B6: IF (K > L)     //Duyệt hết 1 run
      Thực hiện B12
B7: fread(&a, sizeof(T), 1, Fd)  //Đọc 1 phần tử của run trên Fd ra biến tạm a
B8: fwrite(&a, sizeof(T), 1, Ft1) //Ghi giá trị biến tạm a vào tập tin Ft1
B9: K++
B10: IF (feof(Fd))          //Đã phân phối hết
      Thực hiện Bkt
B11: Lặp lại B6
//Chép 1 run từ Fd sang Ft2
B12: K = 1
```

B13: IF (K > L)
Thực hiện B19
B14: fread(&a, sizeof(T), 1, Fd) //Đọc 1 phần tử của run trên Fd ra biến tạm a
B15: fwrite(&a, sizeof(T), 1, Ft2) //Ghi giá trị biến tạm a vào tập tin Ft2
B16: K++
B17: IF (feof(Fd)) //Đã phân phối hết
Thực hiện Bkt
B18: Lặp lại B13
B19: Lặp lại B4
Bkt: Kết thúc

- Thuật toán trộn:

B1: Ft1 = fopen(DataTemp1, "r") //Mở tập tin trung gian thứ nhất để đọc dữ liệu
B2: Ft2 = fopen(DataTemp2, "r") //Mở tập tin trung gian thứ hai để đọc dữ liệu
B3: Fd = fopen(DataFile, "w") //Mở tập tin dữ liệu để ghi dữ liệu
B4: fread(&a1, sizeof(T), 1, Ft1) //Đọc 1 phần tử của run trên Ft1 ra biến tạm a1
B5: fread(&a2, sizeof(T), 1, Ft2) //Đọc 1 phần tử của run trên Ft2 ra biến tạm a2
B6: K1 = 1 //Chỉ số để duyệt các run trên Ft1
B7: K2 = 1 //Chỉ số để duyệt các run trên Ft2
B8: IF (a1 ≤ a2) // a1 đứng trước a2 trên Fd
B8.1: fwrite(&a1, sizeof(T), 1, Fd)
B8.2: K1++
B8.3: If (feof(Ft1)) //Đã chép hết các phần tử trong Ft1
Thực hiện B23
B8.4: fread(&a1, sizeof(T), 1, Ft1)
B8.5: If (K1 > L) //Đã duyệt hết 1 run trong Ft1
Thực hiện B11
B8.6: Lặp lại B8
B9: ELSE // a2 đứng trước a1 trên Fd
B9.1: fwrite(&a2, sizeof(T), 1, Fd)
B9.2: K2++
B9.3: If (feof(Ft2)) //Đã chép hết các phần tử trong Ft2
Thực hiện B27
B9.4: fread(&a2, sizeof(T), 1, Ft2)
B9.5: If (K2 > L) //Đã duyệt hết 1 run trong Ft2
Thực hiện B17
B9.6: Lặp lại B8
B10: Lặp lại B6

//Chép phần run còn lại trong Ft2 về Fd

B11: IF (K2 > L) //Đã chép hết phần run còn lại trong Ft2 về Fd
Lặp lại B6
B12: fwrite(&a2, sizeof(T), 1, Fd)
B13: K2++
B14: IF (feof(Ft2)) //Đã chép hết các phần tử trong Ft2
Thực hiện B27
B15: fread(&a2, sizeof(T), 1, Ft2)
B16: Lặp lại B11

//Chép phần run còn lại trong Ft1 về Fd

B17: IF (K1 > L) //Đã chép hết phần run còn lại trong Ft1 về Fd
Lặp lại B6
B18: fwrite(&a1, sizeof(T), 1, Fd)
B19: K1++
B20: IF (feof(Ft1)) //Đã chép hết các phần tử trong Ft1
Thực hiện B23
B21: fread(&a1, sizeof(T), 1, Ft1)
B22: Lặp lại B17

//Chép các phần tử còn lại trong Ft2 về Fd

B23: fwrite(&a2, sizeof(T), 1, Fd)
B24: IF (feof(Ft2))
Thực hiện Bkt
B25: fread(&a2, sizeof(T), 1, Ft2)
B26: Lặp lại B23

//Chép các phần tử còn lại trong Ft1 về Fd

B27: fwrite(&a1, sizeof(T), 1, Fd)
B28: IF (feof(Ft1))
Thực hiện Bkt
B29: fread(&a1, sizeof(T), 1, Ft1)
B30: Lặp lại B27
Bkt: Kết thúc

- Thuật toán sắp xếp trộn thẳng:

B1: L = 1 //Chiều dài ban đầu của các run
B2: IF (L ≥ N) //Tập tin Fd chỉ còn 01 run
Thực hiện Bkt
B3: Phân_Phối(DataFile, DataTemp1, DataTemp2, L)
B4: Trộn(DataTemp1, DataTemp2, DataFile, L)
B5: L = 2*L
B6: Lặp lại B2
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm FileStraightMergeSort có prototype như sau:

```
int  FileStraightMergeSort(char * DataFile);
```

Hàm thực hiện việc sắp xếp các phần tử có kiểu dữ liệu T trên tập tin có tên DataFile theo thứ tự tăng dựa trên thuật toán sắp trộn trực tiếp. Nếu việc sắp xếp thành công hàm trả về giá trị 1, trong trường hợp ngược lại (do có lỗi khi thực hiện các thao tác trên tập tin) hàm trả về giá trị –1. Hàm sử dụng các hàm FileDistribute, FileMerge có prototype và ý nghĩa như sau:

```
int  FileDistribute(char * DataFile, char * DataTemp1, char * DataTemp2, int  L);
```

Hàm thực hiện việc phân phối luân phiên các đường chạy có chiều dài L trên tập tin dữ liệu có tên DataFile về cho các tập tin tạm thời có tên tương ứng là DataTemp1

và DataTemp2. Hàm trả về giá trị 1 nếu việc phân phối hoàn tất, trong trường hợp ngược lại hàm trả về giá trị –1.

int FileMerge(char * DataTemp1, char * DataTemp2, char * DataFile, int L);

Hàm thực hiện việc trộn từng cặp tương ứng các đường chạy với độ dài L trên hai tập tin tạm thời có tên DataTemp1, DataTemp2 về tập tin dữ liệu ban đầu có tên DataFile thành các đường chạy có chiều dài 2*L. Hàm trả về giá trị 1 nếu việc trộn hoàn tất, trong trường hợp ngược lại hàm trả về giá trị –1.

Cả hai hàm này đều sử dụng các hàm Finished để làm nhiệm vụ "dọn dẹp" (đóng các tập tin đã mở, hủy vùng nhớ đã cấp phát, ...) và trả về một giá trị nguyên để kết thúc. Các hàm Finished có prototype như sau:

```
int  Finished (FILE * F1, int  ReturnValue);
int  Finished (FILE * F1, FILE * F2, int  ReturnValue);
int  Finished (FILE * F1, FILE * F2, FILE * F3, int  ReturnValue);
```

Nội dung của các hàm như sau:

```
int  Finished (FILE * F1, int  ReturnValue)
   {  fclose (F1);
      return (ReturnValue);
   }

//=========================================================

int  Finished (FILE * F1, FILE * F2, int  ReturnValue)
   {  fclose (F1);
      fclose (F2);
      return (ReturnValue);
   }

//=========================================================

int  Finished (FILE * F1, FILE * F2, FILE * F3, int  ReturnValue);
   {  fclose (F1);
      fclose (F2);
      fclose (F3);
      return (ReturnValue);
   }

//=========================================================

int  FileDistribute(char * DataFile, char * DataTemp1, char * DataTemp2, int  L)
   {  FILE * Fd = fopen(DataFile, "rb");
      if (Fd == NULL)
         return (-1);
      FILE * Ft1 = fopen(DataTemp1, "wb");
      if (Ft1 == NULL)
         return(Finished(Fd, -1));
      FILE * Ft2 = fopen(DataTemp2, "wb");
      if (Ft2 == NULL)
         return(Finished(Fd, Ft1, -1));
      T  a;
```

```
        int  SOT = sizeof(T);
        while (!feof(Fd))
           {  for(int K = 0; K<L && !feof(Fd); K++)
                 {  int  t = fread(&a, SOT, 1, Fd);
                    if (t < 1)
                       {  if (feof(Fd))
                             break;
                          return (Finished(Fd, Ft1, Ft2, -1));
                       }
                       t = fwrite(&a, SOT, 1, Ft1);
                    if (t < 1)
                       return(Finished(Fd, Ft1, Ft2, -1));
                 }
              for(K = 0; K<L && !feof(Fd); K++)
                 {  int  t = fread(&a, SOT, 1, Fd);
                    if (t < 1)
                       {  if (feof(Fd))
                             break;
                          return (Finished(Fd, Ft1, Ft2, -1));
                       }
                       t = fwrite(&a, SOT, 1, Ft2);
                    if (t < 1)
                       return(Finished(Fd, Ft1, Ft2, -1));
                 }
           }
        return (Finished(Fd, Ft1, Ft2, 1));
     }
  //=========================================================
  int  FileMerge(char * DataTemp1, char * DataTemp2, char * DataFile, int  L)
     {  FILE * Ft1 = fopen(DataTemp1, "rb");
        if (Ft1 == NULL)
           return (-1);
        FILE * Ft2 = fopen(DataTemp2, "rb");
        if (Ft2 == NULL)
           return (Finished(Ft1, -1);
        FILE * Fd = fopen(DataFile, "wb");
        if (Fd == NULL)
           return (Finished(Ft1, Ft2, -1);
        int  K1 = 0, K2 = 0;
        T  a1, a2;
        int  SOT = sizeof(T);
        if (fread(&a1, SOT, 1, Ft1) < 1)
           return (Finished(Fd, Ft1, Ft2, -1));
        if (fread(&a2, SOT, 1, Ft2) < 1)
           return (Finished(Fd, Ft1, Ft2, -1));
        while (!feof(Ft1) && !feof(Ft2))
           {  if (a1 <= a2)
```

```
    {  int t = fwrite(&a1, SOT, 1, Fd);
       if (t < 1)
          return (Finished(Fd, Ft1, Ft2, -1));
       K1++;
       t = fread(&a1, SOT, 1, Ft1);
       if (t < 1)
          {  if (feof(Ft1))
                break;
             return (Finished (Fd, Ft1, Ft2, -1));
          }
       if (K1 == L)
          {  for (; K2 < L && !feof(Ft2); K2++)
                {  t = fwrite(&a2, SOT, 1, Fd);
                   if (t < 1)
                      return (Finished(Fd, Ft1, Ft2, -1));
                   t = fread(&a2, SOT, 1, Ft2);
                   if (t < 1)
                      {  if (feof(Ft2))
                            break;
                         return (Finished(Fd, Ft1, Ft2, -1));
                      }
                }
             if (feof(Ft2))
                break;
          }
       if (K1 == L && K2 == L)
          K1 = K2 = 0;
    }
 else
    {  int t = fwrite(&a2, SOT, 1, Fd);
       if (t < 1)
          return (Finished(Fd, Ft1, Ft2, -1));
       K2++;
       t = fread(&a2, SOT, 1, Ft2);
       if (t < 1)
          {  if (feof(Ft1))
                break;
             return (Finished (Fd, Ft1, Ft2, -1));
          }
       if (K2 == L)
          {  for (; K1 < L && !feof(Ft1); K1++)
                {  t = fwrite(&a1, SOT, 1, Fd);
                   if (t < 1)
                      return (Finished(Fd, Ft1, Ft2, -1));
                   t = fread(&a1, SOT, 1, Ft1);
                   if (t < 1)
                      {  if (feof(Ft1))
                            break;
```

```
                                    return (Finished(Fd, Ft1, Ft2, -1));
                                }
                            }
                        if (feof(Ft1))
                            break;
                    }
                if (K1 == L && K2 == L)
                    K1 = K2 = 0;
            }
        }
    while (!feof(Ft1))
        {  int t = fwrite(&a1, SOT, 1, Fd);
            if (t < 1)
                return (Finished(Fd, Ft1, Ft2, -1));
            t = fread(&a1, SOT, 1, Ft1);
            if (t < 1)
                {  if (feof(Ft1))
                        break;
                    return (Finished (Fd, Ft1, Ft2, -1));
                }
        }
    while (!feof(Ft2))
        {  int t = fwrite(&a2, SOT, 1, Fd);
            if (t < 1)
                return (Finished(Fd, Ft1, Ft2, -1));
            t = fread(&a2, SOT, 1, Ft2);
            if (t < 1)
                {  if (feof(Ft2))
                        break;
                    return (Finished (Fd, Ft1, Ft2, -1));
                }
        }
    return (Finished(Fd, Ft1, Ft2, 1));
    }
//=========================================================
int  FileStraightMergeSort(char * DataFile)
    {  int  Fhd = open(DataFile, O_RDONLY);
        if (Fhd < 0)
            return (-1);
        int N = filelength(Fhd)/sizeof(T);
        close(Fhd);
        if (N < 2)
            return (1);
        int L = 1;
        char * Temp1 = "Data1.Tmp";
        char * Temp2 = "Data2.Tmp";
        while (L < N)
            {  if (FileDistribute(DataFile, Temp1, Temp2, L) == -1)
```

```
            {  remove(Temp1);
               remove(Temp2);
               return (-1);
            }
         if (FileMerge(Temp1, Temp2, DataFile, L) == -1)
            {  remove(Temp1);
               remove(Temp2);
               return (-1);
            }
         L = 2*L;
       }
    remove (Temp1);
    remove (Temp2);
    return (1);
}
```

- Ví dụ minh họa thuật toán sắp xếp trộn thẳng:

Giả sử dữ liệu ban đầu trên tập tin Fd như sau:

10 4 15 2 1 20 22 15 14 30 5 8 40 31 36

Ta tiến hành phân phối và trộn các đường chạy có chiều dài cố định L:

Lần 1: L = 1

Phân phối luân phiên các đường chạy chiều dài L = 1 trên Fd về Ft1 và Ft2:

Fd: **10** *4* **15** *2* **1** *20* **22** *15* **14** *30* **5** *8* **40** *31* **36**

Ft1: **10** **15** **1** **22** **14** **5** **40** **36**

Ft2: *4* *2* *20* *15* *30* *8* *31*

Trộn các cặp đường chạy tương ứng chiều dài L = 1 trên Ft1 và Ft2 thành các đường chạy chiều dài L = 2 (thực tế L có thể nhỏ hơn 2) và đưa về Fd:

Ft1: **10** **15** **1** **22** **14** **5** **40** **36**

Ft2: *4* *2* *20* *15* *30* *8* *31*

Fd: **4 10** *2 15* **1 20** *15 22* **14 30** *5 8* **31 40** *36*

Lần 2: L = 2

Phân phối luân phiên các đường chạy chiều dài L ≤ 2 trên Fd về Ft1 và Ft2:

Fd: **4 10** *2 15* **1 20** *15 22* **14 30** *5 8* **31 40** *36*

Ft1: **4 10** **1 20** **14 30** **31 40**

Ft2: *2 15* *15 22* *5 8* *36*

Trộn các cặp đường chạy tương ứng chiều dài L ≤ 2 trên Ft1 và Ft2 thành các đường chạy chiều dài L ≤ 4 và đưa về Fd:

Ft1: **4 10** **1 20** **14 30** **31 40**

Ft2: *2 15* *15 22* *5 8* *36*

Fd: **2 4 10 15** *1 15 20 22* **5 8 14 30** *31 36 40*

Lần 3: L = 4

Phân phối luân phiên các đường chạy chiều dài L ≤ 4 trên Fd về Ft1 và Ft2:

Fd: **2 4 10 15** *1 15 20 22* **5 8 14 30** *31 36 40*

Ft1: **2 4 10 15 5 8 14 30**

Ft2: *1 15 20 22 31 36 40*

Trộn các cặp đường chạy tương ứng chiều dài L ≤ 4 trên Ft1 và Ft2 thành các đường chạy chiều dài L ≤ 8 và đưa về Fd:

Ft1: **2 4 10 15 5 8 14 30**

Ft2: *1 15 20 22 31 36 40*

Fd: **1 2 4 10 15 15 20 22** *5 8 14 30 31 36 40*

Lần 4: L = 8

Phân phối luân phiên các đường chạy chiều dài L ≤ 8 trên Fd về Ft1 và Ft2:

Fd: **1 2 4 10 15 15 20 22** *5 8 14 30 31 36 40*

Ft1: **1 2 4 10 15 15 20 22**

Ft2: *5 8 14 30 31 36 40*

Trộn các cặp đường chạy tương ứng chiều dài L ≤ 8 trên Ft1 và Ft2 thành các đường chạy chiều dài L ≤ 16 và đưa về Fd. Thuật toán kết thúc:

Ft1: **1 2 4 10 15 15 20 22**

Ft2: *5 8 14 30 31 36 40*

Ft1: **1 2 4 5 8 10 14 15 15 20 22 30 31 36 40**

- Phân tích thuật toán:

+ Trong thuật giải này chúng ta luôn thực hiện $\log_2(N)$ lần phân phối và trộn các run.

+ Ở mỗi lần phân phối run chúng ta phải thực hiện: N lần đọc và ghi đĩa, 2N phép so sánh (N lần so sánh hết run và N lần so sánh hết tập tin).

+ Ở mỗi lần trộn run chúng ta cũng phải thực hiện: N lần đọc và ghi đĩa, 2N+N/2 phép so sánh (N lần so sánh hết run, N lần so sánh hết tập tin và N/2 lần so sánh các cặp giá trị tương ứng trên 2 tập tin phụ).

+ Trong mọi trường hợp:

Số lần đọc và ghi đĩa: $D = 2N \times Log_2(N)$
Số phép so sánh: $S = (4N + N/2) \times Log_2(N)$

+ Trong thuật toán này chúng ta sử dụng 2 tập tin phụ để thực hiện việc phân phối và trộn các đường chạy. Khi số tập tin phụ từ 3 tập tin trở lên (K>2) thì các thuật toán trộn được gọi là trộn đa lối (multiways) và sẽ làm giảm số lần phân phối – trộn các đường chạy, tức là làm giảm số lần đọc và ghi đĩa.

+ Cần lưu ý là thời gian thực hiện các thuật giải sắp xếp/tìm kiếm trên tập tin phụ thuộc rất nhiều vào các thao tác đọc và ghi đĩa.

b. Thuật toán sắp xếp trộn tự nhiên (Natural Merge Sort):

- Tư tưởng:

Tương tự như thuật toán trộn tự nhiên trên mảng, chúng ta tận dụng các đường chạy tự nhiên ban đầu trên tập tin Fd có chiều dài không cố định. Tiến hành phân phối luân phiên các đường chạy tự nhiên này của tập tin Fd về 2 tập tin phụ Ft1, Ft2. Sau đó trộn tương ứng từng cặp đường chạy tự nhiên ở 2 tập tin phụ Ft1, Ft2 thành một đường chạy mới có chiều dài bằng tổng chiều dài của cặp hai đường chạy đem trộn và đưa về tập tin Fd.

Như vậy, sau mỗi lần phân phối và trộn các đường chạy tự nhiên trên tập tin Fd thì số đường chạy tự nhiên trên tập tin Fd sẽ giảm đi một nửa, đồng thời chiều dài các đường chạy tự nhiên cũng được tăng lên. Do đó, sau tối đa $Log_2(N)$ lần phân phối và trộn thì tập tin Fd chỉ còn lại 01 đường chạy với chiều dài là N và khi đó tập tin Fd trở thành tập tin có thứ tự.

Trong thuật giải này chúng ta sử dụng 2 tập tin phụ (có thể sử dụng nhiều hơn) và quá trình phân phối, trộn các đường chạy tự nhiên được trình bày riêng biệt thành 2 thuật giải:

+ Thuật giải phân phối luân phiên (tách) các đường chạy tự nhiên trên tập tin Fd về hai tập tin phụ Ft1, Ft2;
+ Thuật giải trộn (nhập) các cặp đường chạy tự nhiên trên hai tập tin Ft1, Ft2 về tập tin Fd thành các đường chạy tự nhiên với chiều dài lớn hơn;

và chúng ta cũng giả sử rằng các lỗi thao tác trên tập tin sẽ bị bỏ qua.

- Thuật toán phân phối:

B1: Fd = fopen(DataFile, “r”) //Mở tập tin dữ liệu cần sắp xếp để đọc dữ liệu
B2: Ft1 = fopen(DataTemp1, “w”) //Mở tập tin trung gian thứ nhất để ghi dữ liệu
B3: Ft2 = fopen(DataTemp2, “w”) //Mở tập tin trung gian thứ hai để ghi dữ liệu
B4: IF (feof(Fd)) //Đã phân phối hết
    Thực hiện Bkt
B5: fread(&a, sizeof(T), 1, Fd) //Đọc 1 phần tử của run trên Fd ra biến tạm a

//Chép 1 đường chạy tự nhiên từ Fd sang Ft1

B6: fwrite(&a, sizeof(T), 1, Ft1) //Ghi giá trị biến tạm a vào tập tin Ft1
B7: IF (feof(Fd)) //Đã phân phối hết
    Thực hiện Bkt
B8: fread(&b, sizeof(T), 1, Fd) //Đọc tiếp 1 phần tử của run trên Fd ra biến tạm b
B9: IF (a > b) // Đã duyệt hết 1 đường chạy tự nhiên
    B9.1: a = b // Chuyển vai trò của b cho a
    B9.2: Thực hiện B12
B10: a = b
B11: Lặp lại B6

//Chép 1 đường chạy tự nhiên từ Fd sang Ft2

B12: fwrite(&a, sizeof(T), 1, Ft2) //Ghi giá trị biến tạm a vào tập tin Ft2
B13: IF (feof(Fd)) //Đã phân phối hết
    Thực hiện Bkt

B14: fread(&b, sizeof(T), 1, Fd)//Đọc 1 phần tử của run trên Fd ra biến tạm b
B15: IF (a > b) // Đã duyệt hết 1 đường chạy tự nhiên
B15.1: a = b// Chuyển vai trò của b cho a
B15.2: Thực hiện B18
B16: a = b
B17: Lặp lại B12
B18: Lặp lại B6
Bkt: Kết thúc

- Thuật toán trộn:

B1: Ft1 = fopen(DataTemp1, "r") //Mở tập tin trung gian thứ nhất để đọc dữ liệu
B2: Ft2 = fopen(DataTemp2, "r") //Mở tập tin trung gian thứ hai để đọc dữ liệu
B3: Fd = fopen(DataFile, "w") //Mở tập tin dữ liệu để ghi dữ liệu
B4: fread(&a1, sizeof(T), 1, Ft1) //Đọc 1 phần tử của run trên Ft1 ra biến tạm a1
B5: fread(&a2, sizeof(T), 1, Ft2) //Đọc 1 phần tử của run trên Ft2 ra biến tạm a2
B6: IF (a1 ≤ a2) // a1 đứng trước a2 trên Fd
B6.1: fwrite(&a1, sizeof(T), 1, Fd)
B6.2: If (feof(Ft1)) //Đã chép hết các phần tử trong Ft1
Thực hiện B21 //Chép các phần tử còn lại trong Ft2 về Fd
B6.3: fread(&b1, sizeof(T), 1, Ft1) //Đọc tiếp 1 phần tử trên Ft1 ra biến tạm b1
B6.4: If (a1 > b1) //Đã duyệt hết đường chạy tự nhiên trong Ft1
B6.4.1: a1 = b1 // Chuyển vai trò của b1 cho a1
B6.4.2: Thực hiện B9
B6.5: a1 = b1
B6.6: Lặp lại B6
B7: ELSE // a2 đứng trước a1 trên Fd
B7.1: fwrite(&a2, sizeof(T), 1, Fd)
B7.2: If (feof(Ft2)) // Đã chép hết các phần tử trong Ft2
Thực hiện B25 // Chép các phần tử còn lại trong Ft1 về Fd
B7.3: fread(&b2, sizeof(T), 1, Ft2) //Đọc tiếp 1 phần tử trên Ft2 ra biến tạm b2
B7.4: If (a2 > b2) // Đã duyệt hết đường chạy tự nhiên trong Ft2
B7.4.1: a2 = b2 // Chuyển vai trò của b2 cho a2
B7.4.2: Thực hiện B15
B7.5: a2 = b2
B7.6: Lặp lại B7
B8: Lặp lại B6

//Chép phần đường chạy tự nhiên còn lại trong Ft2 về Fd

B9: fwrite(&a2, sizeof(T), 1, Fd)
B10: IF (feof(Ft2)) // Đã chép hết các phần tử trong Ft2
Thực hiện B25 //Chép các phần tử còn lại trong Ft1 về Fd
B11: fread(&b2, sizeof(T), 1, Ft2)
B12: IF (a2 > b2) // Đã chép hết 1 đường chạy tự nhiên trong Ft2
B12.1: a2 = b2
B12.2: Lặp lại B6
B13: a2 = b2
B14: Lặp lại B9

//Chép phần đường chạy tự nhiên còn lại trong Ft1 về Fd

B15: fwrite(&a1, sizeof(T), 1, Fd)
B16: IF (feof(Ft1)) // Đã chép hết các phần tử trong Ft1
Thực hiện B21 //Chép các phần tử còn lại trong Ft2 về Fd
B17: fread(&b1, sizeof(T), 1, Ft1)
B18: IF (a1 > b1) // Đã chép hết 1 đường chạy tự nhiên trong Ft1
B18.1: a1 = b1
B18.2: Lặp lại B6
B19: a1 = b1
B20: Lặp lại B15

//Chép các phần tử còn lại trong Ft2 về Fd

B21: fwrite(&a2, sizeof(T), 1, Fd)
B22: IF (feof(Ft2))
Thực hiện Bkt
B23: fread(&a2, sizeof(T), 1, Ft2)
B24: Lặp lại B21

//Chép các phần tử còn lại trong Ft1 về Fd
B25: fwrite(&a1, sizeof(T), 1, Fd)
B26: IF (feof(Ft1))
Thực hiện Bkt
B27: fread(&a1, sizeof(T), 1, Ft1)
B28: Lặp lại B25
Bkt: Kết thúc

- Thuật toán sắp xếp trộn tự nhiên:

B1: L = Phân_Phối(DataFile, DataTemp1, DataTemp2)
B2: IF (L ≥ N) //Tập tin Fd chỉ còn 01 run
Thực hiện Bkt
B3: L = Trộn(DataTemp1, DataTemp2, DataFile)
B4: IF (L ≥ N) //Tập tin Fd chỉ còn 01 run
Thực hiện Bkt
B5: Lặp lại B1
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm FileNaturalMergeSort có prototype như sau:

int FileNaturalMergeSort(char * DataFile);

Hàm thực hiện việc sắp xếp các phần tử có kiểu dữ liệu T trên tập tin có tên DataFile theo thứ tự tăng dựa trên thuật toán sắp trộn tự nhiên. Nếu việc sắp xếp thành công hàm trả về giá trị 1, trong trường hợp ngược lại (do có lỗi khi thực hiện các thao tác trên tập tin) hàm trả về giá trị –1. Hàm sử dụng các hàm FileNaturalDistribute, FileNaturalMerge có prototype và ý nghĩa như sau:

int FileNaturalDistribute(char * DataFile, char * DataTemp1, char * DataTemp2);

Hàm thực hiện việc phân phối luân phiên các đường chạy tự nhiên trên tập tin dữ liệu có tên DataFile về cho các tập tin tạm thời có tên tương ứng là DataTemp1 và

DataTemp2. Hàm trả về giá trị là chiều dài của đường chạy tự nhiên đầu tiên trong tập tin dữ liệu DataFile nếu việc phân phối hoàn tất, trong trường hợp ngược lại hàm trả về giá trị –1.

int FileNaturalMerge(char * DataTemp1, char * DataTemp2, char * DataFile);

Hàm thực hiện việc trộn từng cặp tương ứng các đường chạy tự nhiên trên hai tập tin tạm thời có tên DataTemp1, DataTemp2 về tập tin dữ liệu ban đầu có tên DataFile thành các đường chạy có chiều bằng tổng chiều dài 2 đường chạy đem trộn. Hàm trả về chiều dài của đường chạy tự nhiên đầu tiên sau khi trộn trên tập tin DataFile nếu việc trộn hoàn tất, trong trường hợp ngược lại hàm trả về giá trị –1.

Nội dung của các hàm như sau:

```
int FileNaturalDistribute(char * DataFile, char * DataTemp1, char * DataTemp2)
  { FILE * Fd = fopen(DataFile, "rb");
    if (Fd == NULL)
       return (-1);
    FILE * Ft1 = fopen(DataTemp1, "wb");
    if (Ft1 == NULL)
       return (Finished (Fd, -1));
    FILE * Ft2 = fopen(DataTemp2, "wb");
    if (Ft2 == NULL)
       return (Finished (Fd, Ft1, -1));
    T  a, b;
    int  SOT = sizeof(T);
    int L = 0, FirstRun1 = 1;
    if (fread(&a, SOT, 1, Fd) < 1)
       { if (feof(Fd))
             return (Finished(Fd, Ft1, Ft2, 0));
          return (Finished (Fd, Ft1, Ft2, -1));
       }
    while (!feof(Fd))
       { do { int  t = fwrite(&a, SOT, 1, Ft1);
              if (t < 1)
                 return (Finished (Fd, Ft1, Ft2, -1));
              if (FirstRun1 == 1)
                 L++;
              t = fread(&b, SOT, 1, Fd);
              if (t < 1)
                 { if (feof(Fd))
                       break;
                    return (Finished (Fd, Ft1, Ft2, -1));
                 }
              if (a > b)
                 { a = b;
                    break;
                 }
              a = b;
            }
```

```
            while (1);
            if (feof(Fd))
               break;
            do {  int  t = fwrite(&a, SOT, 1, Ft2);
                  if (t < 1)
                     return (Finished (Fd, Ft1, Ft2, -1));
                  t = fread(&b, SOT, 1, Fd);
                  if (t < 1)
                     {  if (feof(Fd))
                           break;
                        return (Finished (Fd, Ft1, Ft2, -1));
                     }
                  if (a > b)
                     {  a = b;
                        FirstRun1 = 0;
                        break;
                     }
                  a = b;
               }
            while (1);
         }
      return (Finished (Fd, Ft1, Ft2, L);
   }

//=========================================================

int  FileNaturalMerge(char * DataTemp1, char * DataTemp2, char * DataFile)
   {  FILE * Fd = fopen(DataFile, "wb");
      if(Fd == NULL)
         return(-1);
      FILE * Ft1 = fopen(DataTemp1, "rb");
      if(Ft1 == NULL)
         return(Finished(Fd, -1));
      FILE * Ft2 = fopen(DataTemp2, "rb");
      if(Ft2 == NULL)
         return(Finished(Fd, Ft1, -1));
      int a1, a2, b1, b2;
      if (fread(&a1, SOT, 1, Ft1) < 1)
         return(Finished(Fd, Ft1, Ft2, -1));
      if (fread(&a2, SOT, 1, Ft2) < 1)
         return(Finished(Fd, Ft1, Ft2, -1));
      int L = 0;
      int FirstRun1 = 1, FirstRun2 = 1;
      while(!feof(Ft1) && !feof(Ft2))
         {  if (a1 <= a2)
               {  int t = fwrite(&a1, SOT, 1, Fd);
                  if (t < 1)
                     return(Finished(Fd, Ft1, Ft2, -1));
                  if (FirsRun1 == 1)
```

```
            L++;
        t = fread(&b1, SOT, 1, Ft1);
        if (t < 1)
            {  if (feof(Ft1))
                   break;
               return(Finished(Fd, Ft1, Ft2, -1));
            }
        if (a1 > b1)
            {  do { t = fwrite(&a2, SOT, 1, Fd);
                     if (t < 1)
                        return(Finished(Fd, Ft1, Ft2, -1));
                     if (FirstRun2 == 1)
                        L++;
                     t = fread(&b2, SOT, 1, Ft2);
                     if (t < 1)
                        {  if (feof(Ft2))
                              {  FirstRun2 = 0;
                                 break;
                              }
                           return(Finished(Fd, Ft1, Ft2, -1));
                        }
                     if (a2 > b2)
                        {  FirstRun2 = 0;
                           a2 = b2;
                           break;
                        }
                  }
               while(1);
               a1 = b1;
               FirstRun1 = 0;
               if (feof(Ft2))
                  break;
            }
        a1 = b1;
    }
else
    {  int t = fwrite(&a2, SOT, 1, Fd);
        if (t < 1)
           return(Finished(Fd, Ft1, Ft2, -1));
        if (FirstRun2 == 1)
           L++;
        t = fread(&b2, SOT, 1, Ft2);
        if (t < 1)
            {  if (feof(Ft2))
                   break;
               return(Finished(Fd, Ft1, Ft2, -1));
            }
        if (a2 > b2)
```

```
                { do { t = fwrite(&a1, SOT, 1, Fd);
                       if (t < 1)
                          return(Finished(Fd, Ft1, Ft2, -1));
                       if (Fr1 == 1)
                          L++;
                       t = fread(&b1, SOT, 1, Ft1);
                       if (t < 1)
                          { if (feof(Ft1))
                               { FirstRun1 = 0;
                                 break;
                               }
                            return(Finished(Fd, Ft1, Ft2, -1));
                          }
                       if (a1 > b1)
                          { FirstRun1 = 0;
                            a1 = b1;
                            break;
                          }
                     }
                  while(1);
                  a2 = b2;
                  FirstRun2 = 0;
                  if (feof(Ft1))
                     break;
                }
             a2 = b2;
          }
   }
while(!feof(Ft1))
   { int t = fwrite(&a1, SOT, 1, Fd);
     if (t < 1)
        return(Finished(Fd, Ft1, Ft2, -1));
     if (FirstRun1 == 1)
        L++;
     t = fread(&a1, SOT, 1, Ft1);
     if (t < 1)
        { if (feof(Ft1))
             break;
          return(Finished(Fd, Ft1, Ft2, -1));
        }
   }
while(!feof(Ft2))
   { int t = fwrite(&a2, SOT, 1, Fd);
     if (t < 1)
        return(Finished(Fd, Ft1, Ft2, -1));
     if (FirstRun2 == 1)
        L++;
     t = fread(&a2, SOT, 1, Ft2);
```

```
            if (t < 1)
               {  if (feof(Ft2))
                     break;
                  return(Finished(Fd, Ft1, Ft2, -1));
               }
         }
      return(Finished(Fd, Ft1, Ft2, L));
   }
//=========================================================
int  FileNaturalMergeSort(char * DataFile)
   {  int  Fhd = open(DataFile, O_RDONLY);
      if (Fhd < 0)
         return (-1);
      int N = filelength(Fhd)/sizeof(T);
      close (Fhd);
      if (N < 2)
         return (1);
      char * Temp1 = "Data1.Tmp";
      char * Temp2 = "Data2.Tmp";
      int L = 0;
      do{  L = FileNaturalDistribute(DataFile, Temp1, Temp2);
            if (L == -1)
               {  remove(Temp1);
                  remove(Temp2);
                  return (-1);
               }
            if (L == N)
               break;
            L = FileNaturalMerge(Temp1, Temp2, DataFile);
            if (L == -1)
               {  remove(Temp1);
                  remove(Temp2);
                  return (-1);
               }
            if (L == N)
               break;
         }
      while (L < N);
      remove(Temp1);
      remove(Temp2);
      return (1);
   }
```

- Ví dụ minh họa thuật toán sắp xếp trộn tự nhiên:

Giả sử dữ liệu ban đầu trên tập tin Fd như sau:

80 24 5 12 11 2 2 15 10 35 35 18 4 1 6

Ta tiến hành phân phối và trộn các đường chạy tự nhiên:

Lần 1: L = 1

Phân phối luân phiên các đường chạy tự nhiên trên Fd về Ft1 và Ft2:

Fd: **80** *24* **5 12** *11* **2 2 15** *10 35 35* **18** *4* **1 6**

Ft1: **80 5 12 2 2 15 18 1 6**

Ft2: *24 11 10 35 35 4*

Trộn các cặp đường chạy tự nhiên tương ứng trên Ft1 và Ft2 thành các đường chạy tự nhiên trong đó đường chạy tự nhiên đầu tiên có chiều dài L = 2 và đưa về Fd:

Ft1: **80 5 12 2 2 15 18 1 6**

Ft2: *24 11 10 35 35 4*

Fd: **24 80** *5 11 12* **2 2 10 15 18 35 35** *1 4 6*

Lần 2: L = 2

Phân phối luân phiên các đường chạy tự nhiên trên Fd về Ft1 và Ft2:

Fd: **24 80** *5 11 12* **2 2 10 15 18 35 35** *1 4 6*

Ft1: **24 80 2 2 10 15 18 35 35**

Ft2: *5 11 12 1 4 6*

Trộn các cặp đường chạy tự nhiên tương ứng trên Ft1 và Ft2 thành các đường chạy tự nhiên trong đó đường chạy tự nhiên đầu tiên có chiều dài L = 5 và đưa về Fd:

Ft1: **24 80 2 2 10 15 18 35 35**

Ft2: *5 11 12 1 4 6*

Fd: **5 11 12 24 80** *1 2 2 4 6 10 15 18 35 35*

Lần 3: L = 5

Phân phối luân phiên các đường chạy tự nhiên trên Fd về Ft1 và Ft2:

Fd: **5 11 12 24 80** *1 2 2 4 6 10 15 18 35 35*

Ft1: **5 11 12 24 80**

Ft2: *1 2 2 4 6 10 15 18 35 35*

Trộn các cặp đường chạy tự nhiên tương ứng trên Ft1 và Ft2 thành các đường chạy tự nhiên trong đó đường chạy tự nhiên đầu tiên có chiều dài L = 15 và đưa về Fd. Thuật toán kết thúc:

Ft1: **5 11 12 24 80**

Ft2: *1 2 2 4 6 10 15 18 35 35*

Fd: **1 2 2 4 5 6 10 11 12 15 18 24 35 35 80**

- Phân tích thuật toán:

+ Trong trường hợp tốt nhất, khi dãy có thứ tự tăng thì sau khi phân phối lần thứ nhất thuật toán kết thúc, do đó:

  Số lần đọc – ghi đĩa: Dmin = N

Số phép so sánh: Smin = 2N

+ Trong trường hợp xấu nhất, khi dãy có thứ tự giảm và ở mỗi bước trộn phân phối thì độ dài đường chạy mới cũng chỉ tăng gấp đôi. Trong trường hợp này sẽ giống như thuật toán trộn trực tiếp:

Số lần đọc và ghi đĩa: Dmax = $2N \times Log_2(N)$
Số phép so sánh: Smax = $(4N + N/2) \times Log_2(N)$

+ Trung bình:

Số lần đọc và ghi đĩa: Davg = $N \times Log_2(N) + N/2$
Số phép so sánh: Savg = $(2N + N/4) \times Log_2(N) + N$

3.3.2. Sắp xếp theo chỉ mục (Index Sort)

Thông thường kích thước của các phần tử dữ liệu trên tập tin dữ liệu khá lớn và kích thước của tập tin dữ liệu cũng lớn. Vả lại biến động dữ liệu trên tập tin dữ liệu ít liên tục mà chủ yếu là chúng ta truy xuất dữ liệu thường xuyên. Do vậy, việc đọc – ghi nhiều lên tập tin dữ liệu sẽ làm cho thời gian truy xuất tập tin dữ liệu rất mất nhiều thời gian và không bảo đảm an toàn cho dữ liệu. Để giải quyết vấn đề này chúng ta tiến hành thao tác tập tin dữ liệu thông qua một tập tin tuần tự chỉ mục theo khóa nhận diện của các phần tử dữ liệu.

a. Tư tưởng:

Từ tập tin dữ liệu ban đầu, chúng ta tiến hành tạo tập tin chỉ mục theo khóa nhận diện của các phần tử dữ liệu (Tập tin chỉ mục được sắp xếp tăng theo khóa nhận diện của các phần tử dữ liệu). Trên cơ sở truy xuất lần lượt các phần tử trong tập tin chỉ mục chúng ta sẽ điều khiển trật tự xuất hiện của các phần tử dữ liệu trong tập tin dữ liệu theo đúng trật tự trên tập tin chỉ mục. Như vậy trong thực tiễn, tập tin dữ liệu không bị thay đổi thứ tự vật lý ban đầu trên đĩa mà chỉ bị thay đổi trật tự xuất hiện các phần tử dữ liệu khi được liệt kê ra màn hình, máy in, ….

Về cấu trúc các phần tử trong tập tin chỉ mục thì như đã trình bày trong phần tìm kiếm theo chỉ mục (Chương 2). Ở đây chúng ta chỉ trình bày cách tạo tập tin chỉ mục theo khóa nhận diện từ tập tin dữ liệu ban đầu và cách thức mà tập tin chỉ mục sẽ điều khiển thứ tự xuất hiện của các phần tử dữ liệu trên tập tin dữ liệu. Hai thao tác này sẽ được trình bày riêng thành hai thuật toán:

- Thuật toán tạo tập tin chỉ mục
- Thuật toán điều khiển thứ tự xuất hiện các phần tử dữ liệu dựa trên tập tin chỉ mục.

b. Thuật toán:

- Thuật toán tạo tập tin chỉ mục

B1: Fd = open(DataFile, “r”) //Mở tập tin dữ liệu để đọc dữ liệu
B2: Fidx = open(IdxFile, “w”) // Mở để tạo mới tập tin chỉ mục
B3: CurPos = 0
B4: read (Fd, a)
B5: IF (EOF(Fd))
    Thực hiện B11
B6: ai.Key = a.Key

B7: ai.Pos = CurPos
B8: write (Fidx, ai)
B9: CurPos += SOT
B10: Lặp lại B4
B11: close (Fd)
B12: close (Fidx)
B13: FileNaturalMergeSort(IdxFile)
Bkt: Kết thúc

- Thuật toán điều khiển thứ tự xuất hiện các phần tử dữ liệu dựa trên tập tin chỉ mục

B1: Fd = open(DataFile, "r") //Mở tập tin dữ liệu để đọc dữ liệu
B2: Fidx = open(IdxFile, "r") // Mở tập tin chỉ mục để đọc
B3: read (Fidx, ai)
B4: IF (EOF(Fidx))
    Thực hiện B9
B5: seek(Fd, ai.Pos)
B6: read (Fd, a)
B7: Output (a) //Xử lý phần tử dữ liệu mới đọc được
B8: Lặp lại B3
B9: close (Fd)
B10: close (Fidx)
Bkt: Kết thúc

c. Cài đặt thuật toán:

Hàm CreateIndex thực hiện việc tạo tập tin chỉ mục từ tập tin dữ liệu và sắp xếp các phần tử trong tập tin chỉ mục theo thứ tự tăng theo khóa nhận diện. Nếu việc tạo tập tin chỉ mục thành công, hàm trả về giá trị 1, ngược lại hàm trả về giá trị –1. Hàm CreateIndex có prototype như sau:
int CreateIndex (char * DataFile, char * IdxFile);

Nội dung của hàm CreateIndex:

```
int  CreateIndex (char * DataFile, char * IdxFile)
  {  FILE * Fd = fopen (DataFile, "rb");
     if (Fd == NULL)
        return (-1);
     FILE * Fidx = fopen (IdxFile, "wb");
     if (Fidx == NULL)
        return (Finished (Fd, -1));
     DataType  a;
     IdxType  ai;
     int  SOT = sizeof(DataType);
     int  SOI = sizeof(IdxType);
     long  CurPos = 0;
     while (!feof(Fd))
        {  if (fread (&a, SOT, 1, Fd) < 1)
              {  if (feof(Fd))
                    break;
                 return (Finished (Fd, Fidx, -1));
```

```
            }
            ai.Key = a.Key;
            ai.Pos = CurPos;
            if (fwrite (&ai, SOI, 1, Fidx) < 1)
               return (Finished (Fd, Fidx, -1));
            CurPos += SOT;
         }
      fclose (Fd);
      fclose (Fidx);
      if (FileNaturalMergeSort(IdxFile) == -1)
         {  remove (IdxFile);
            return (-1);
         }
      return (1);
   }
```

Hàm DisplayData thực hiện điều khiển thứ tự xuất hiện các phần tử dữ liệu trên tập tin dữ liệu dựa trên tập tin chỉ mục đã được tạo. Nếu việc liệt kê thành công, hàm trả về giá trị 1, ngược lại hàm trả về giá trị –1. Hàm DisplayData có prototype như sau:

```
int  DisplayData (char * DataFile, char * IdxFile);
```

Nội dung của hàm DisplayData:

```
int  DisplayData (char * DataFile, char * IdxFile)
   {  FILE * Fd = fopen (DataFile, "rb");
      if (Fd == NULL)
         return (-1);
      FILE * Fidx = fopen (IdxFile, "rb");
      if (Fidx == NULL)
         return (Finished (Fd, -1));
      DataType  a;
      IdxType  ai;
      int  SOT = sizeof(DataType);
      int  SOI = sizeof(IdxType);
      while (!feof(Fidx))
         {  if (fread (&ai, SOI, 1, Fidx) < 1)
               {  if (feof(Fidx))
                     return (Finished (Fd, Fidx, 1));
                  return (Finished (Fd, Fidx, -1));
               }
            fseek(Fd, ai.Pos, SEEK_SET);
            if (fread (&a, SOT, 1, Fd) < 1)
               return (Finished (Fd, Fidx, -1));
            Output(a);
         }
      return (Finished (Fd, Fidx, 1));
   }
```

☛ **Lưu ý**:

Hàm Output thực hiện việc xuất thông tin của một phần tử dữ liệu ra thiết bị xuất thông tin. Ngoài ra, nếu chúng ta muốn xử lý dữ liệu trong phần tử dữ liệu này theo thứ tự điều khiển bởi tập tin chỉ mục thì chúng ta cũng có thể viết một hàm thực hiện thao tác xử lý thay cho hàm Output này.

d. Phân tích thuật toán:

Trong thuật toán này chúng ta phải thực hiện ít nhất 01 lần tạo tập tin chỉ mục. Để tạo tập tin chỉ mục chúng ta phải thực hiện N lần đọc – ghi đĩa. Khi thực hiện việc liệt kê các phần tử dữ liệu chúng ta cũng phải thực hiện 2N lần đọc đĩa.

Nhược điểm lớn nhất trong thuật toán này là chúng ta phải cập nhật lại tập tin chỉ mục khi có sự thay đổi dữ liệu trên tập tin dữ liệu.

Câu hỏi và Bài tập

1. Trình bày tư tưởng của các thuật toán sắp xếp?

2. Trong các thuật toán sắp xếp bạn thích nhất là thuật toán nào? Thuật toán nào bạn không thích nhất? Tại sao?

3. Trình bày và cài đặt tất cả các thuật toán sắp xếp nội, ngoại theo thứ tự giảm? Cho nhận xét về các thuật toán này?

4. Hãy trình bày những ưu khuyết điểm của mỗi thuật toán sắp xếp? Theo bạn cách khắc phục những nhược điểm này là như thế nào?

5. Sử dụng hàm random trong C để tạo ra một dãy M có 1.000 số nguyên. Vận dụng các thuật toán sắp xếp để sắp xếp các phần tử của mảng M theo thứ tự tăng dần về mặt giá trị. Với cùng một dữ liệu như nhau, cho biết thời gian thực hiện các thuật toán? Có nhận xét gì đối với các thuật toán sắp xếp này? Bạn hãy đề xuất và cài đặt thuật toán Quick-Sort trong trường hợp không dùng đệ quy?

6. Thông tin về mỗi số hạng của một đa thức bậc n bao gồm: Hệ số – là một số thực, Bậc – là một số nguyên có giá trị từ 0 đến 100. Hãy định nghĩa cấu trúc dữ liệu để lưu trữ các đa thức trong bộ nhớ trong của máy tính. Với cấu trúc dữ liệu đã được định nghĩa, hãy vận dụng một thuật toán sắp xếp và cài đặt chương trình thực hiện việc sắp xếp các số hạng trong đa thức theo thứ tự tăng dần của các bậc.

7. Thông tin về các phòng thi tại một hội đồng thi bao gồm: Số phòng – là một số nguyên có giá trị từ 1 đến 200, Nhà – là một chữ cái in hoa từ A → Z, Khả năng chứa – là một số nguyên có giá trị từ 10 → 250. Hãy định nghĩa cấu trúc dữ liệu để lưu trữ các phòng thi này trong bộ nhớ trong của máy tính. Với cấu trúc dữ liệu đã được định nghĩa, vận dụng các thuật toán sắp xếp và cài đặt chương trình thực hiện việc các công việc sau:

- Sắp xếp và in ra màn hình danh sách các phòng thi theo thứ tự giảm dần về Khả năng chứa.
- Sắp xếp và in ra màn hình danh sách các phòng thi theo thứ tự tăng dần theo Nhà (Từ A → Z), các phòng cùng một nhà thì sắp xếp theo thứ tự tăng dần theo Số phòng.

- Sắp xếp và in ra màn hình danh sách các phòng thi theo thứ tự tăng dần theo Nhà (Từ A → Z), các phòng cùng một nhà thì sắp xếp theo thứ tự giảm dần theo Khả năng chứa.

8. Tạo tập tin dữ liệu SONGUYEN.DAT gồm 10000 số nguyên. Vận dụng các thuật toán sắp xếp trên file, hãy cài đặt chương trình để sắp xếp dữ liệu trên tập tin này theo thứ tự tăng dần về giá trị của các số nguyên trong đó. Cho biết thời gian thực hiện mỗi thuật toán? Có nhận xét gì đối với các thuật toán này?

9. Thông tin về một sinh viên bao gồm: Mã số – là một số nguyên dương, Họ và đệm – là một chuỗi có tối đa 20 ký tự, Tên sinh viên – là một chuỗi có tối đa 10 ký tự, Ngày, tháng, năm sinh – là các số nguyên dương, Phái – Là "Nam" hoặc "Nữ", Điểm trung bình – là các số thực có giá trị từ 0.00 → 10.00. Viết chương trình nhập vào danh sách sinh viên (ít nhất là 10 sinh viên, không nhập trùng mã giữa các sinh viên với nhau) và lưu trữ danh sách này vào tập tin có tên SINHVIEN.DAT, sau đó vận dụng các thuật toán sắp xếp trên file để sắp xếp danh sách sinh viên theo thứ tự tăng dần theo Mã sinh viên. In danh sách sinh viên trong file SINHVIEN.DAT sau khi sắp xếp ra màn hình.

10. Với tập tin dữ liệu có tên SINHVIEN.DAT trong bài tập 9, thực hiện các yêu cầu sau:

- Tạo các tập tin chỉ mục theo các khóa trong các trường hợp sau:
  + Chỉ mục sắp xếp theo Mã sinh viên tăng dần;
  + Chỉ mục sắp xếp theo Tên sinh viên từ A → Z, nếu cùng tên thì sắp sếp Họ và đệm theo thứ tự từ A → Z;
  + Chỉ mục sắp xếp theo Điểm trung bình giảm dần.
- Lưu các tập tin chỉ mục theo các khóa như trong ba trường hợp nêu trên vào trong dĩa với các tên tương ứng là SVMASO.IDX, SVTH.IDX, SVDTB.IDX.
- Dựa vào các tập tin chỉ mục, in ra toàn bộ danh sách sinh viên trong tập tin SINHVIEN.DAT theo đúng thứ sự sắp xếp quy định trong các tập tin chỉ mục.
- Có nhận xét gì khi thực hiện việc sắp xếp dữ liệu trên tập tin theo chỉ mục.

11. Trình bày và cài đặt các thuật toán để cập nhật lại tập tin chỉ mục khi tập tin dữ liệu bị thay đổi trong các trường hợp sau:

- Khi thêm 01 phần tử dữ liệu vào tập tin dữ liệu.
- Khi hủy 01 phần tử dữ liệu trong tập tin dữ liệu.
- Khi hiệu chỉnh thành khóa chỉ mục của 01 phần tử dữ liệu trong tập tin dữ liệu.

12. Trình bày và cài đặt các thuật toán để minh họa (mô phỏng) các bước trong quá trình sắp xếp dữ liệu cho các thuật toán sắp xếp nội (Sử dụng các giao diện đồ họa để cài đặt),

Chương 4: DANH SÁCH (LIST)

4.1. Khái niệm về danh sách

Danh sách là tập hợp các phần tử có kiểu dữ liệu xác định và giữa chúng có một mối liên hệ nào đó.

Số phần tử của danh sách gọi là chiều dài của danh sách. Một danh sách có chiều dài bằng 0 là một danh sách rỗng.

4.2. Các phép toán trên danh sách

Tùy thuộc vào đặc điểm, tính chất của từng loại danh sách mà mỗi loại danh sách có thể có hoặc chỉ cần thiết có một số phép toán (thao tác) nhất định nào đó. Nói chung, trên danh sách thường có các phép toán như sau:

- Tạo mới một danh sách:

Trong thao tác này, chúng ta sẽ đưa vào danh sách nội dung của các phần tử, do vậy chiều dài của danh sách sẽ được xác định. Trong một số trường hợp, chúng ta chỉ cần khởi tạo giá trị và trạng thái ban đầu cho danh sách.

- Thêm một phần tử vào danh sách:

Thao tác này nhằm thêm một phần tử vào trong danh sách, nếu việc thêm thành công thì chiều dài của danh sách sẽ tăng lên 1. Cũng tùy thuộc vào từng loại danh sách và từng trường hợp cụ thể mà việc thêm phần tử sẽ được tiến hành đầu, cuối hay giữa danh sách.

- Tìm kiếm một phần tử trong danh sách:

Thao tác này sẽ vận dụng các thuật toán tìm kiếm để tìm kiếm một phần tử trên danh sách thỏa mãn một tiêu chuẩn nào đó (thường là tiêu chuẩn về giá trị).

- Loại bỏ bớt một phần tử ra khỏi danh sách:

Ngược với thao tác thêm, thao tác này sẽ loại bỏ bớt một phần tử ra khỏi danh sách do vậy, nếu việc loại bỏ thành công thì chiều dài của danh sách cũng bị giảm xuống 1. Thông thường, trước khi thực hiện thao tác này chúng ta thường phải thực hiện thao tác tìm kiếm phần tử cần loại bỏ.

- Cập nhật (sửa đổi) giá trị cho một phần tử trong danh sách:

Thao tác này nhằm sửa đổi nội dung của một phần tử trong danh sách. Tương tự như thao tác loại bỏ, trước khi thay đổi thường chúng ta cũng phải thực hiện thao tác tìm kiếm phần tử cần thay đổi.

- Sắp xếp thứ tự các phần tử trong danh sách:

Trong thao tác này chúng ta sẽ vận dụng các thuật toán sắp xếp để sắp xếp các phần tử trên danh sách theo một trật tự xác định.

- Tách một danh sách thành nhiều danh sách:

Thao tác này thực hiện việc chia một danh sách thành nhiều danh sách con theo một tiêu thức chia nào đó. Kết quả sau khi chia là tổng chiều dài trong các danh sách con phải bằng chiều dài của danh sách ban đầu.

- Nhập nhiều danh sách thành một danh sách:

Ngược với thao tác chia, thao tác này tiến hành nhập nhiều danh sách con thành một danh sách có chiều dài bằng tổng chiều dài các danh sách con. Tùy vào từng trường hợp mà việc nhập có thể là:

+ Ghép nối đuôi các danh sách lại với nhau,

+ Trộn xen lẫn các phần tử trong danh sách con vào danh sách lớn theo một trật tự nhất định.

- Sao chép một danh sách:

Thao tác này thực hiện việc sao chép toàn bộ nội dung của danh sách này sang một danh sách khác sao cho sau khi sao chép, hai danh sách có nội dung giống hệt nhau.

- Hủy danh sách:

Thao tác này sẽ tiến hành hủy bỏ (xóa bỏ) toàn bộ các phần tử trong danh sách. Việc xóa bỏ này tùy vào từng loại danh sách mà có thể là xóa bỏ toàn bộ nội dung hay cả nội dung lẫn không gian bộ nhớ lưu trữ danh sách.

4.3. Danh sách đặc (Condensed List)

4.3.1. Định nghĩa

Danh sách đặc là danh sách mà không gian bộ nhớ lưu trữ các phần tử được đặt liên tiếp nhau trong bộ nhớ.

4.3.2. Biểu diễn danh sách đặc

Để biểu diễn danh sách đặc chúng ta sử dụng một dãy (mảng) các phần tử có kiểu dữ liệu là kiểu dữ liệu của các phần tử trong danh sách. Do vậy, chúng ta cần biết trước số phần tử tối đa của mảng cũng chính là chiều dài tối đa của danh sách thông qua một hằng số nguyên. Ngoài ra, do chiều dài của danh sách luôn luôn biến động cho nên chúng ta cũng cần quản lý chiều dài thực của danh sách thông qua một biến nguyên.

Giả sử chúng ta quy ước chiều dài tối đa của danh sách đặc là 10000, khi đó cấu trúc dữ liệu để biểu diễn danh sách đặc như sau:

```
const int MaxLen = 10000;          // hoặc:   #define MaxLen 10000
int Length;
T CD_LIST[MaxLen];                 // hoặc:   T * CD_LIST = new T[MaxLen];
```

Nếu chúng ta sử dụng cơ chế cấp phát động để cấp phát bộ nhớ cho danh sách đặc thì cần kiểm tra sự thành công của việc cấp phát động.

4.3.3. Các thao tác trên danh sách đặc

Ở đây có nhiều thao tác đã được trình bày ở các chương trước, do vậy chúng ta không trình bày lại mà chỉ liệt kê cho có hệ thống hoặc trình bày tóm tắt những nội dung chính của các thao tác này.

Các thao tác cơ bản trên danh sách đặc như sau:

a. Khởi tạo danh sách (Initialize):

Trong thao tác này chỉ đơn giản là chúng ta cho chiều dài của danh sách về 0. Hàm khởi tạo danh sách đặc như sau:

```
void  CD_Initialize(int  &Len)
   {  Len = 0;
      return;
   }
```

b. Tạo mới danh sách/ Nhập danh sách:

Hàm CD_Create_List có prototype:

```
int  CD_Create_List(T  M[], int  &Len);
```

Hàm tạo danh sách đặc có chiều dài tối đa MaxLen. Hàm trả về chiều dài thực của danh sách sau khi tạo.

Nội dung của hàm như sau:

```
int  CD_Create_List(T  M[], int  &Len)
   {  if (Len > MaxLen)
         Len = MaxLen;
      for (int i = 0; i < Len; i++)
         M[i] = Input_One_Element();
      return (Len);
   }
```

Lưu ý: Hàm Input_One_Element thực hiện nhập vào nội dung của một phần tử có kiểu dữ liệu T và trả về giá trị của phần tử mới nhập vào. Tùy vào từng trường hợp cụ thể mà chúng ta viết hàm Input_One_Element cho phù hợp.

c. Thêm một phần tử vào trong danh sách:

Giả sử chúng ta cần thêm một phần tử có giá trị NewValue vào trong danh sách M có chiều dài Length tại vị trí InsPos.

- Thuật toán:

```
B1: IF (Length = MaxLen)
          Thực hiện Bkt
//Dời các phần tử từ vị trí InsPos->Length ra sau một vị trí
B2: Pos = Length+1
B3: IF (Pos = InsPos)
          Thực hiện B7
B4: M[Pos] = M[Pos-1]
B5: Pos--
B6: Lặp lại B3
B7: M[InsPos] = NewValue    //Đưa phần tử có giá trị NewValue vào vị trí InsPos
B8: Length++              //Tăng chiều dài của danh sách lên 1
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm CD_Insert_Element có prototype:

int CD_Insert_Element(T M[], int &Len, T NewValue, int InsPos);

Hàm thực hiện việc chèn phần tử có giá trị NewValue vào trong danh sách M có chiều dài Len tại vị trí InsPos. Hàm trả về chiều dài thực của danh sách sau khi chèn nếu việc chèn thành công và ngược lại, hàm trả về giá trị -1. Nội dung của hàm như sau:

```
int CD_Insert_Element(T M[], int &Len, T NewValue, int InsPos)
{  if (Len == MaxLen)
       return (-1);
   for (int i = Len; i > InsPos; i--)
       M[i] = M[i-1];
   M[InsPos] = NewValue;
   Len++;
   return (Len);
}
```

d. Tìm kiếm một phần tử trong danh sách:

Thao tác này chúng ta sử dụng các thuật toán tìm kiếm nội (Tìm tuyến tính hoặc tìm nhị phân) đã được trình bày trong Chương 2.

e. Loại bỏ bớt một phần tử ra khỏi danh sách:

Giả sử chúng ta cần loại bỏ phần tử tại vị trí DelPos trong danh sách M có chiều dài Length (Trong một số trường hợp có thể chúng ta phải thực hiện thao tác tìm kiếm để xác định vị trí của phần tử cần xóa).

- Thuật toán:

```
B1: IF (Length = 0 OR DelPos > Len)
        Thực hiện Bkt

//Dời các phần tử từ vị trí DelPos+1->Length ra trước một vị trí
B2: Pos = DelPos
B3: IF (Pos = Length)
        Thực hiện B7
B4: M[Pos] = M[Pos+1]
B5: Pos++
B6: Lặp lại B3
B7: Length--            //Giảm chiều dài của danh sách đi 1
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm CD_Delete_Element có prototype:

int CD_Delete_Element(T M[], int &Len, int DelPos);

Hàm thực hiện việc xóa phần tử tại vị trí DelPos trong danh sách M có chiều dài Len. Hàm trả về chiều dài thực của danh sách sau khi xóa nếu việc xóa thành công và ngược lại, hàm trả về giá trị -1. Nội dung của hàm như sau:

```
int  CD_Delete_Element(T  M[], int  &Len, int  DelPos)
   {  if (Len == 0 || DelPos >= Len)
         return (-1);
      for (int i = DelPos; i < Len-1; i++)
         M[i] = M[i+1];
      Len--;
      return (Len);
   }
```

f. Cập nhật (sửa đổi) giá trị cho một phần tử trong danh sách:

Giả sử chúng ta cần sửa đổi phần tử tại vị trí ChgPos trong danh sách M có chiều dài Length thành giá trị mới NewValue. Thao tác này chỉ đơn giả là việc gán lại giá trị mới cho phần tử cần thay đổi: M[ChgPos] = NewValue;

Trong một số trường hợp, trước tiên chúng ta phải thực hiện thao tác tìm kiếm phần tử cần thay đổi giá trị để xác định vị trí của nó sau đó mới thực hiện phép gán như trên.

g. Sắp xếp thứ tự các phần tử trong danh sách:

Thao tác này chúng ta sử dụng các thuật toán sắp xếp nội (trên mảng) đã trình bày trong Chương 3.

h. Tách một danh sách thành nhiều danh sách:

Tùy thuộc vào từng yêu cầu cụ thể mà việc tách một danh sách thành nhiều danh sách có thể thực hiện theo những tiêu thức khác nhau:

+ Có thể phân phối luân phiên theo các đường chạy như đã trình bày trong các thuật toán sắp xếp theo phương pháp trộn ở Chương 3;

+ Có thể phân phối luân phiên từng phần của danh sách cần tách cho các danh sách con. Ở dây chúng ta sẽ trình bày theo cách phân phối này;

+ Tách các phần tử trong danh sách thỏa mãn một điều kiện cho trước.

Giả sử chúng ta cần tách danh sách M có chiều dài Length thành các danh sách con SM1, SM2 có chiều dài tương ứng là SLen1, SLen2.

- Thuật toán:

```
// Kiểm tra tính hợp lệ của SLen1 và SLen2: SLen1 + SLen2 = Length
B1: IF (SLen1 ≥ Length)
      B1.1: SLen1 = Length
      B1.2: SLen2 = 0
B2: IF (SLen2 ≥ Length)
      B2.1: SLen2 = Length
      B2.2: SLen1 = 0
B3: IF (SLen1 + Slen2 ≠ Length)
      SLen2 = Length – SLen1
B4: IF (SLen1 < 0)
      SLen1 = 0
B5: IF (SLen2 < 0)
      SLen2 = 0
```

// Chép SLen1 phần tử đầu trong M vào SM1
B6: i = 1, si = 1
B7: IF (i > SLen1)
    Thực hiện B11
B8: SM1[si] = M[i]
B9: i++, si++
B10: Lặp lại B7
// Chép SLen2 phần tử cuối trong M vào SM2
B11: si = 1
B12: IF (i > Length)
    Thực hiện Bkt
B13: SM2[si] = M[i]
B14: i++, si++
B15: Lặp lại B12
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm CD_Split có prototype:

```
void CD_Split(T M[], int Len, T SM1[], int &SLen1, T SM2[], int &SLen2);
```

Hàm thực hiện việc sao chép nội dung SLen1 phần tử đầu tiên trong danh sách M vào trong danh con SM1 và sao chép SLen2 phần tử cuối cùng trong danh sách M vào trong danh sách con SM2. Hàm hiệu chỉnh lại SLen1, SLen2 nếu cần thiết.

Nội dung của hàm như sau:

```
void CD_Split(T M[], int Len, T SM1[], int &SLen1, T SM2[], int &SLen2)
  { if (SLen1 >= Len)
      { SLen1 = Len;
        SLen2 = 0;
      }
    if (SLen2 >= Len)
      { SLen2 = Len;
        SLen1 = 0;
      }
    if (SLen1 < 0)   SLen1 = 0;
    if (SLen2 < 0)   SLen2 = 0;
    if (SLen1 + SLen2 != Len)
      SLen2 = Len – SLen1;
    for (int i = 0; i < SLen1; i++)
      SM1[i] = M[i];
    for (int j = 0; i < Len; i++, j++)
      SM2[j] = M[i];
    return;
  }
```

i. Nhập nhiều danh sách thành một danh sách:

Tùy thuộc vào từng yêu cầu cụ thể mà việc nhập nhiều danh sách thành một danh sách có thể thực hiện theo các phương pháp khác nhau, có thể là:

+ Ghép nối đuôi các danh sách lại với nhau;

+ Trộn xen lẫn các phần tử trong danh sách con vào danh sách lớn theo một trật tự nhất định như chúng ta đã trình bày trong các thuật toán trộn ở Chương 3.

Ở đây chúng ta trình bày cách ghép các danh sách thành một danh sách.

Giả sử chúng ta cần ghép các danh sách SM1, SM2 có chiều dài SLen1, SLen2 vào thành một danh sách M có chiều dài Length = SLen1 + SLen2 theo thứ tự từ SM1 rồi đến SM2.

- Thuật toán:

// Kiểm tra khả năng chứa của M: SLen1 + SLen2 ≤ MaxLen
B1: IF (SLen1 + SLen2 > MaxLen)
    Thực hiện Bkt

// Chép SLen1 phần tử đầu trong SM1 vào đầu M
B2: i = 1
B3: IF (i > SLen1)
    Thực hiện B7
B4: M[i] = SM1[i]
B5: i++
B6: Lặp lại B3

// Chép SLen2 phần tử đầu trong SM2 vào sau M
B7: si = 1
B8: IF (si > SLen2)
    Thực hiện Bkt
B9: M[i] = M2[si]
B10: i++, si++
B11: Lặp lại B8
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm CD_Concat có prototype:

```
int CD_Concat (T SM1[], int SLen1, T SM2[], int SLen2, T M[], int &Len);
```

Hàm thực hiện việc sao ghép nội dung hai danh sách SM1, SM2 có chiều dài tương ứng SLen1, SLen2 về danh sách M có chiều dài Len = SLen1 + SLen2 theo thứ tự SM1 đến SM2. Hàm trả về chiều dài của danh sách M sau khi ghép nếu việc ghép thành công, trong trường hợp ngược lại hàm trả về giá trị -1.

Nội dung của hàm như sau:

```
int CD_Concat (T SM1[], int SLen1, T SM2[], int SLen2, T M[], int &Len)
  { if (SLen1 + SLen2 > MaxLen)
      return (-1);
    for (int i = 0; i < SLen1; i++)
      M[i] = SM1[i];
    for (int j = 0; j < SLen2; i++, j++)
      M[i] = SM2[j];
    Len = SLen1 + SLen2;
```

```
    return (Len);
}
```

j. Sao chép một danh sách:

Giả sử chúng ta cần sao chép nội dung dach sách M có chiều dài Length vào thành danh sách CM có cùng chiều dài.

- Thuật toán:

B1: i = 1
B2: IF (i > Length)
    Thực hiện Bkt
B3: CM[i] = M[i]
B4: i++
B5: Lặp lại B2
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm CD_Copy có prototype:

```
int  CD_Copy (T  M[], int  Len, T  CM[]);
```

Hàm thực hiện việc sao chép nội dung danh sách M có chiều dài Len về danh sách CM có cùng chiều dài. Hàm trả về chiều dài của danh sách CM sau khi sao chép.

Nội dung của hàm như sau:

```
int  CD_Copy (T  M[], int  Len, T  CM[])
   {  for (int i = 0; i < Len; i++)
         CM[i] = M[i];
      return (Len);
   }
```

k. Hủy danh sách:

Trong thao tác này, nếu danh sách được cấp phát động thì chúng ta tiến hành hủy bỏ (xóa bỏ) toàn bộ các phần tử trong danh sách bằng toán tử hủy bỏ (trong C/C++ là free/delete). Nếu danh sách được cấp phát tĩnh thì việc hủy bỏ chỉ là tạm thời cho chiều dài của danh sách về 0 còn việc thu hồi bộ nhớ sẽ do ngôn ngữ tự thực hiện.

4.3.4. Ưu nhược điểm và Ứng dụng

a. Ưu nhược điểm:

Do các phần tử được lưu trữ liên tiếp nhau trong bộ nhớ, do vậy danh sách đặc có các ưu nhược điểm sau đây:

- Mật độ sử dụng bộ nhớ của danh sách đặc là tối ưu tuyệt đối (100%);
- Việc truy xuất và tìm kiếm các phần tử của danh sách đặc là dễ dàng vì các phần tử đứng liền nhau nên chúng ta chỉ cần sử dụng chỉ số để định vị vị trí các phần tử trong danh sách (định vị địa chỉ các phần tử);
- Việc thêm, bớt các phần tử trong danh sách đặc có nhiều khó khăn do chúng ta phải di dời các phần tử khác đi qua chỗ khác.

b. Ứng dụng của danh sách đặc:

Danh sách đặc được ứng dụng nhiều trong các cấu trúc dữ liệu mảng: mảng 1 chiều, mảng nhiều chiều; Mảng cấp phát tĩnh, mảng cấp phát động; ... mà chúng ta đã nghiên cứu và thao tác khá nhiều trong quá trình lập trình trên nhiều ngôn ngữ lập trình khác nhau.

4.4. Danh sách liên kết (Linked List)

4.4.1. Định nghĩa

Danh sách liên kết là tập hợp các phần tử mà giữa chúng có một sự nối kết với nhau thông qua vùng liên kết của chúng.

Sự nối kết giữa các phần tử trong danh sách liên kết đó là sự quản lý, ràng buộc lẫn nhau về nội dung của phần tử này và địa chỉ định vị phần tử kia. Tùy thuộc vào mức độ và cách thức nối kết mà danh sách liên kết có thể chia ra nhiều loại khác nhau:

- Danh sách liên kết đơn;
- Danh sách liên kết đôi/kép;
- Danh sách đa liên kết;
- Danh sách liên kết vòng (vòng đơn, vòng đôi).

Mỗi loại danh sách sẽ có cách biểu diễn các phần tử (cấu trúc dữ liệu) riêng và các thao tác trên đó. Trong tài liệu này chúng ta chỉ trình bày 02 loại danh sách liên kết cơ bản là danh sách liên kết đơn và danh sách liên kết đôi.

4.4.2. Danh sách liên kết đơn (Singly Linked List)

A. Cấu trúc dữ liệu:

Nội dung của mỗi phần tử trong danh sách liên kết (còn gọi là một nút) gồm hai vùng: Vùng dữ liệu và Vùng liên kết và có cấu trúc dữ liệu như sau:

```
typedef  struct  SLL_Node
  {  T  Key;
     InfoType  Info;
     SLL_Node * NextNode; // Vùng liên kết quản lý địa chỉ phần tử kế tiếp
  }    SLL_OneNode;
```

Tương tự như trong các chương trước, ở đây để đơn giản chúng ta cũng giả thiết rằng vùng dữ liệu của mỗi phần tử trong danh sách liên kết đơn chỉ bao gồm một thành phần khóa nhận diện (Key) cho phần tử đó. Khi đó, cấu trúc dữ liệu trên có thể viết lại đơn giản như sau:

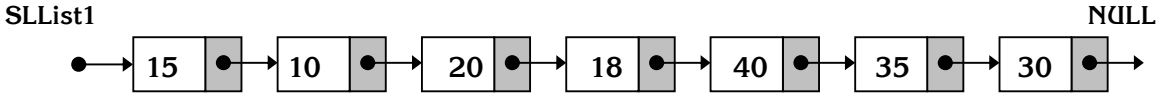
```
typedef  struct  SLL_Node
  {  T  Key;
     SLL_Node * NextNode; // Vùng liên kết quản lý địa chỉ phần tử kế tiếp
  }    SLL_OneNode;

typedef  SLL_OneNode  *  SLL_Type;
```

Để quản lý một danh sách liên kết chúng ta có thể sử dụng nhiều phương pháp khác nhau và tương ứng với các phương pháp này chúng ta sẽ có các cấu trúc dữ liệu khác nhau, cụ thể:

- Quản lý địa chỉ phần tử đầu danh sách:

```
SLL_Type  SLList1;
```

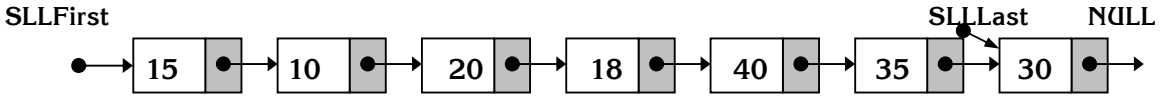
Hình ảnh minh họa:



- Quản lý địa chỉ phần tử đầu và cuối danh sách:

```
typedef  struct  SLL_PairNode
   {  SLL_Type  SLLFirst;
      SLL_Type  SLLLast;
   }      SLLP_Type;

SLLP_Type  SLList2;
```

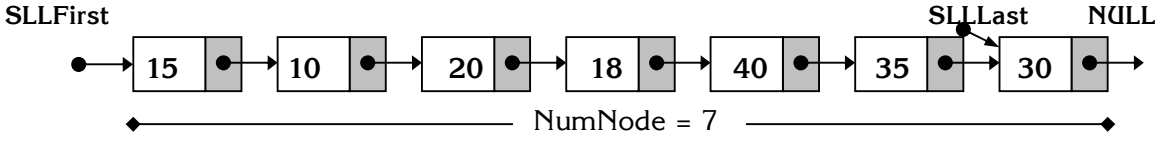
Hình ảnh minh họa:



- Quản lý địa chỉ phần tử đầu, địa chỉ phần tử cuối và số phần tử trong danh sách:

```
typedef  struct  SLL_PairNNode
   {  SLL_Type  SLLFirst;
      SLL_Type  SLLLast;
      unsigned  NumNode;
   }      SLLPN_Type;

SLLPN_Type  SLList3;
```

Hình ảnh minh họa:



B. Các thao tác trên danh sách liên kết đơn:

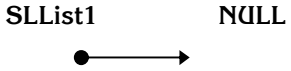
Với mỗi cách quản lý khác nhau của danh sách liên kết đơn , các thao tác cũng sẽ có sự khác nhau về mặt chi tiết song nội dung cơ bản ít có sự khác nhau. Do vậy, ở đây chúng ta chỉ trình bày các thao tác theo cách quản lý thứ nhất (quản lý địa chỉ của phần tử đầu danh sách liên kết đơn), các cách quản lý khác sinh viên tự vận dụng để điều chỉnh cho thích hợp.

a. Khởi tạo danh sách (Initialize):

Trong thao tác này chỉ đơn giản là chúng ta cho giá trị con trỏ quản lý địa chỉ phần tử đầu danh sách về con trỏ NULL. Hàm khởi tạo danh sách liên kết đơn như sau:

```
void  SLL_Initialize(SLL_Type  &First)
   {  First = NULL;
      return;
   }
```

Hình ảnh minh họa:



b. Tạo mới một phần tử / nút:

Giả sử chúng ta cần tạo mới một phần tử có thành phần dữ liệu là NewData.

- Thuật toán:

B1: First = new SLL_OneNode
B2: IF (First = NULL)
    Thực hiện Bkt
B3: First->NextNode = NULL
B4: First->Key = NewData
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm SLL_Create_Node có prototype:

```
SLL_Type  SLL_Create_Node(T  NewData);
```

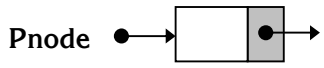
Hàm tạo mới một nút có thành phần dữ liệu là NewData, hàm trả về con trỏ trỏ tới địa chỉ của nút mới tạo. Nếu không đủ bộ nhớ để tạo, hàm trả về con trỏ NULL.

```
SLL_Type  SLL_Create_Node(T  NewData)
   {  SLL_Type  Pnode = new  SLL_OneNode;
      if (Pnode != NULL)
         {  Pnode->NextNode = NULL;
            Pnode->Key = NewData;
         }
      return (Pnode);
   }
```

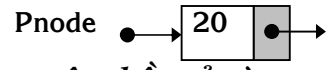
- Minh họa thuật toán:

Giả sử chúng ta cần tạo nút có thành phần dữ liệu là 20: NewData = 20

Pnode = new SLL_OneNode



Pnode->NextNode = NULL
Pnode->Key = NewData



c. Thêm một phần tử vào trong danh sách:

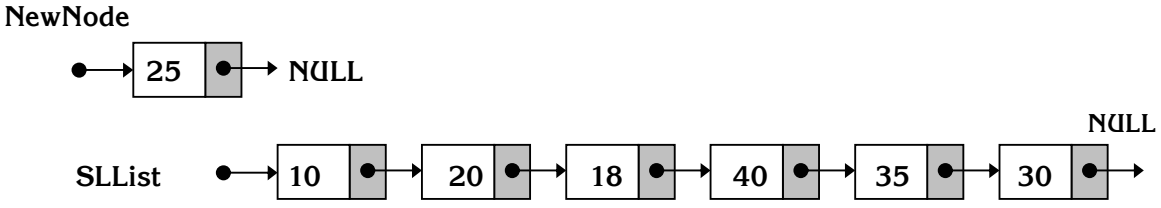
Giả sử chúng ta cần thêm một phần tử có giá trị thành phần dữ liệu là NewData vào trong danh sách. Việc thêm có thể diễn ra ở đầu, cuối hay ở giữa danh sách liên kết. Do vậy, ở đây chúng ta trình bày 3 thao tác thêm riêng biệt nhau:

- Thuật toán thêm phần tử vào đầu danh sách liên kết đơn:

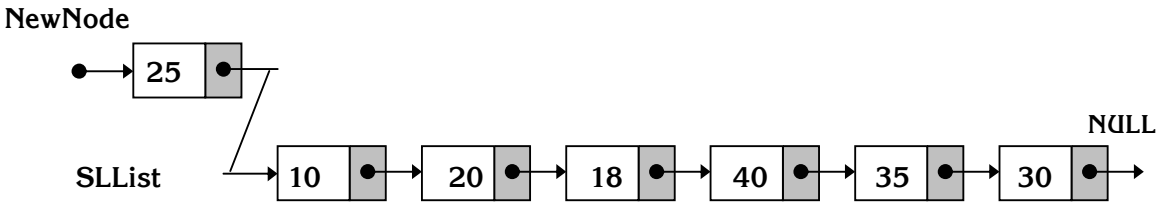
B1: NewNode = SLL_Create_Node (NewData)
B2: IF (NewNode = NULL)
Thực hiện Bkt
B3: NewNode->NextNode = SLList // Nối SLList vào sau NewNode
B4: SLList = NewNode // Chuyển vai trò đứng đầu của NewNode cho SLList
Bkt: Kết thúc

- Minh họa thuật toán:

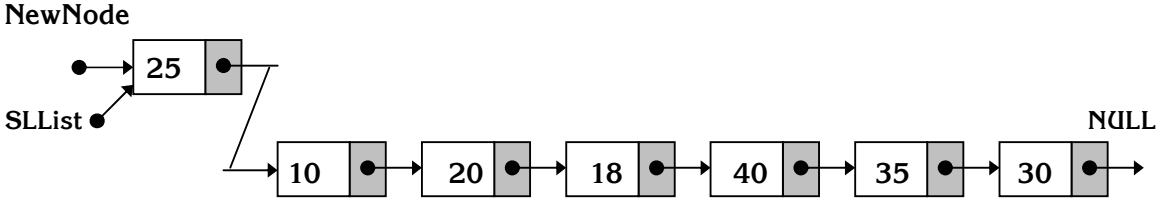
Giả sử chúng ta cần thêm nút có thành phần dữ liệu là 25: NewData = 25



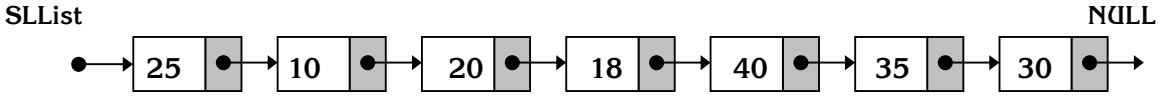
NewNode->NextNode = SLList:



SLList = NewNode:



Kết quả sau khi chèn:



- Thuật toán thêm phần tử vào cuối danh sách liên kết đơn:

B1: NewNode = SLL_Create_Node (NewData)
B2: IF (NewNode = NULL)

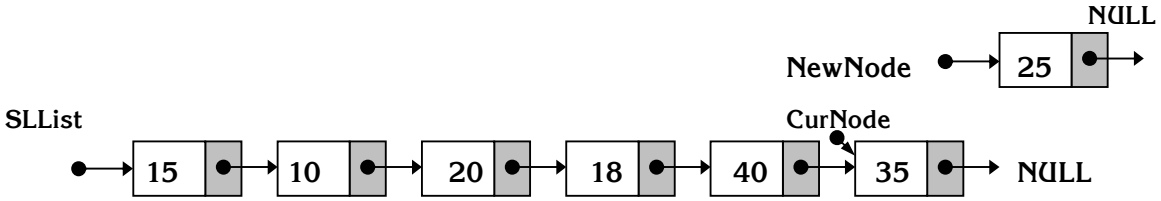
Thực hiện Bkt
B3: IF (SLList = NULL)
B3.1: SLList = NewNode
B3.2: Thực hiện Bkt

// Tìm đến địa chỉ của phần tử cuối cùng trong danh sách liên kết đơn

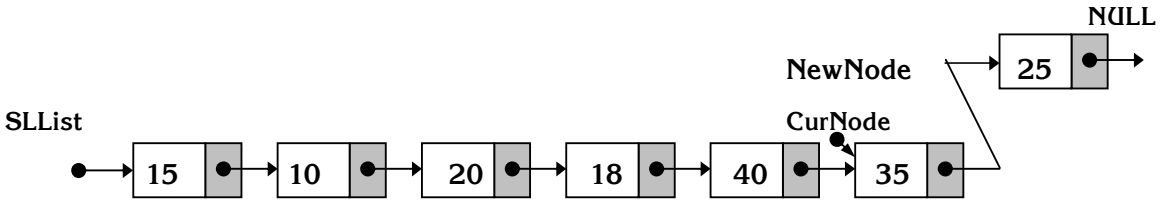
B4: CurNode = SLList
B5: IF (CurNode->NextNode = NULL)
Thực hiện B8
B6: CurNode = CurNode->NextNode // Chuyển qua nút kế tiếp
B7: Lặp lại B5
B8: CurNode->NextNode = NewNode // Nối NewNode vào sau CurNode
Bkt: Kết thúc

- Minh họa thuật toán:

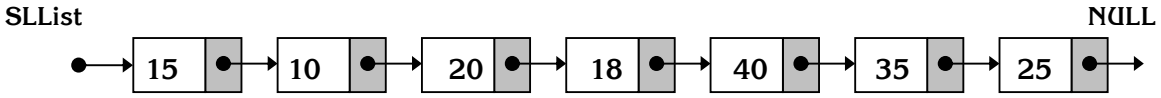
Giả sử chúng ta cần thêm nút có thành phần dữ liệu là 25: NewData = 25



CurNode->NextNode = NewNode:



Kết quả sau khi chèn:



- Thuật toán thêm phần tử vào giữa danh sách liên kết đơn:

Giả sử chúng ta cần thêm một phần tử có giá trị thành phần dữ liệu là NewData vào trong danh sách SLList vào ngay sau nút có địa chỉ InsNode. Trong thực tế nhiều khi chúng ta phải thực hiện thao tác tìm kiếm để xác định địa chỉ InsNode, ở đây giả sử chúng ta đã xác định được địa chỉ này.

B1: NewNode = SLL_Create_Node (NewData)
B2: IF (NewNode = NULL)
Thực hiện Bkt
B3: IF (InsNode->NextNode = NULL)
B3.1: InsNode->NextNode = NewNode
B3.2: Thực hiện Bkt

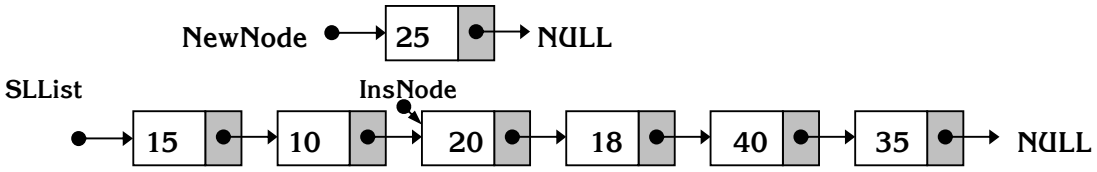
// Nối các nút kế sau InsNode vào sau NewNode

B4: NewNode->NextNode = InsNode->NextNode

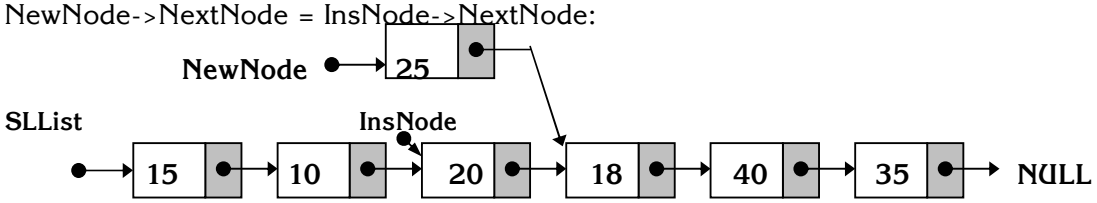
// Chuyển mối liên kết giữa InsNode với nút kế của nó về NewNode
B5: InsNode->NextNode = NewNode
Bkt: Kết thúc

- Minh họa thuật toán:

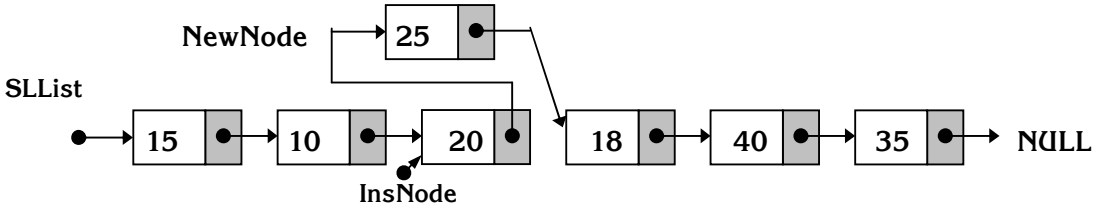
Giả sử chúng ta cần thêm nút có thành phần dữ liệu là 25 vào sau nút có địa chỉ InsNode như sau: NewData = 25



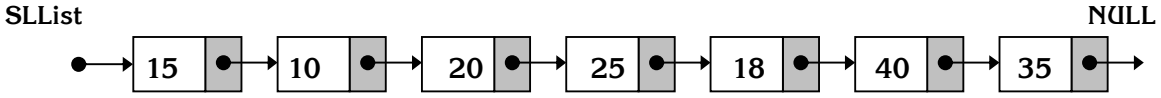
NewNode->NextNode = InsNode->NextNode:



InsNode->NextNode = NewNode:



Kết quả sau khi chèn:



- Cài đặt thuật toán:

Các hàm thêm phần tử tương ứng với các trường hợp có prototype như sau:

SLL_Type SLL_Add_First(SLL_Type &SList, T NewData);

SLL_Type SLL_Add_Last(SLL_Type &SList, T NewData);

SLL_Type SLL_Add_Mid(SLL_Type &SList, T NewData, SLL_Type &InsNode);

Hàm thực hiện việc chèn phần tử có giá trị thành phần dữ liệu NewData vào trong danh sách liên kết đơn quản lý bởi con trỏ đầu danh sách SList tương ứng với 3 trường hợp: Thêm đầu, thêm cuối, thêm giữa. Các hàm trả về giá trị là địa chỉ của nút đầu tiên nếu việc thêm thành công. Trong trường hợp ngược lại, các hàm trả về con trỏ NULL.

Riêng đối với trường hợp thêm giữa, hàm SLL_Add_Mid thực hiện việc thêm vào ngay sau nút có địa chỉ InsNode. Nội dung của các hàm như sau:

```
SLL_Type  SLL_Add_First(SLL_Type  &SList, T  NewData)
   {  SLL_Type  NewNode = SLL_Create_Node(NewData);
      if (NewNode == NULL)
         return (NULL);
      NewNode->NextNode = SList;
      SList = NewNode;
      return (SList);
   }
//=================================================================
   SLL_Type  SLL_Add_Last(SLL_Type  &SList, T  NewData)
   {  SLL_Type  NewNode = SLL_Create_Node(NewData);
      if (NewNode == NULL)
         return (NULL);
      if (SList == NULL)
         {  SList = NewNode;
            return (SList);
         }
      SLL_Type  CurNode = SList;
      while (CurNode->NextNode != NULL)
         CurNode = CurNode->NextNode;
      CurNode->NextNode = NewNode;
      return (SList);
   }
//=================================================================
   SLL_Type  SLL_Add_Mid(SLL_Type  &SList, T  NewData, SLL_Type  &InsNode)
   {  SLL_Type  NewNode = SLL_Create_Node(NewData);
      if (NewNode == NULL)
         return (NULL);
      if (InsNode->NextNode == NULL)
         {  InsNode->NextNode = NewNode;
            return (SList);
         }
      NewNode->NextNode = InsNode->NextNode;
      InsNode->NextNode = NewNode;
      return (SList);
   }
```

d. Duyệt qua các nút trong danh sách:

Đây là một thao tác thường xuyên xảy ra trên danh sách liên kết đơn nói chung và các danh sách khác nói riêng để thực hiện thao tác xử lý các nút hoặc xử lý dữ liệu tại các nút. Có nhiều thao tác xử lý tùy từng trường hợp và yêu cầu song ở đây đơn giản chúng ta chỉ duyệt để xem nội dung thành phần dữ liệu trong danh sách.

- Thuật toán:

B1: CurNode = SLList
B2: IF (CurNode = NULL)
    Thực hiện Bkt

B3: OutputData(CurNode->Key) // Xuất giá trị thành phần dữ liệu trong 1 nút
B4: CurNode = CurNode->NextNode
B5: Lặp lại B2
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm SLL_Travelling có prototype:

```
void   SLL_Travelling(SLL_Type  SList);
```

Hàm duyệt qua các nút trong danh sách liên kết đơn quản lý bởi địa chỉ nút đầu tiên thông qua SList để xem nội dung thành phần dữ liệu của mỗi nút.

Nội dung của hàm như sau:

```
void   SLL_Travelling (SLL_Type  SList)
   {  SLL_Type  CurNode = SList;
      while (CurNode != NULL)
         {  OutputData(CurNode->Key);
            CurNode = CurNode->NextNode;
         }
      return;
   }
```

☛ **Lưu ý:**

Hàm OutputData thực hiện việc xuất nội dung của một biến có kiểu dữ liệu T. Tùy vào từng trường hợp cụ thể mà chúng ta viết hàm OutputData cho phù hợp.

e. Tìm kiếm một phần tử trong danh sách:

Giả sử chúng ta cần tìm kiếm xem trong danh sách liên kết đơn có tồn tại nút có thành phần dữ liệu là SearchData hay không. Thao tác này chúng ta vận dụng thuật toán tìm tuyến tính để tìm kiếm.

- Thuật toán:

B1: CurNode = SLList
B2: IF (CurNode = NULL OR CurNode->Key = SearchData)
      Thực hiện Bkt
B3: CurNode = CurNode->NextNode
B4: Lặp lại B2
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm SLL_Searching có prototype:

```
SLL_Type   SLL_Searching(SLL_Type  SList, T  SearchData);
```

Hàm thực hiện việc tìm kiếm nút có thành phần dữ liệu là SearchData trên danh sách liên kết đơn quản lý bởi địa chỉ nút đầu tiên thông qua SList. Hàm trả về địa chỉ của nút đầu tiên trong danh sách khi tìm thấy, ngược lại hàm trả về con trỏ NULL.

Nội dung của hàm như sau:

```
SLL_Type  SLL_Searching(SLL_Type  SList, T  SearchData)
  {  SLL_Type  CurNode = SList;
     while (CurNode != NULL)
        {  if (CurNode->Key == SearchData)
              break;
           CurNode = CurNode->NextNode;
        }
     return (CurNode);
  }
```

f. Loại bỏ bớt một phần tử ra khỏi danh sách:

Giả sử chúng ta cần loại bỏ phần tử có giá trị thành phần dữ liệu là DelData trong danh sách liên kết đơn. Để thực hiện điều này trước tiên chúng ta phải thực hiện thao tác tìm kiếm địa chỉ của nút có thành phần dữ liệu là DelData, sau đó mới thực hiện thao tác loại bỏ nếu tìm thấy. Tuy nhiên trong quá trình tìm kiếm, nếu tìm thấy chúng ta phải ghi nhận địa chỉ của nút đứng ngay trước nút tìm thấy là PreDelNode.

- Thuật toán:

// Tìm kiếm nút có Key là DelData trong danh sách

B1: DelNode = SLList
B2: PreDelNode = NULL
B3: IF (DelNode = NULL)
        Thực hiện Bkt
B4: IF (DelNode->Key=DelData)
        Thực hiện B8
B5: PreDelNode = DelNode
B6: DelNode = DelNode->NextNode
B7: Lặp lại B3

// Loại bỏ nút tại địa chỉ DelNode ra khỏi danh sách

B8: IF (PreDelNode = NULL) // Loại bỏ nút đầu tiên trong danh sách
    B8.1: SLList = SLList->NextNode
    B8.2: Thực hiện B10

// Liên kết các nốt sau DelNode về nút PreDelNode

B9: PreDelNode->NextNode = DelNode->NextNode

// Cắt mối liên kết giữa DelNode với các nút còn lại trong danh sách
// và hủy DelNode
B10: DelNode->NextNode = NULL
B11: delete DelNode
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm SLL_Delete_Node có prototype:

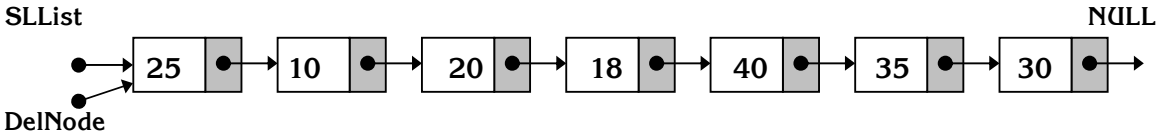
```
int  SLL_Delete_Node (SLL_Type  &SList, T  DelData);
```

Hàm thực hiện việc xóa phần tử có thành phần dữ liệu là DelData trong danh sách liên kết quản lý bởi con trỏ đầu SList. Hàm trả về giá trị 1 nếu việc xóa thành công và ngược lại, hàm trả về giá trị -1. Nội dung của hàm như sau:

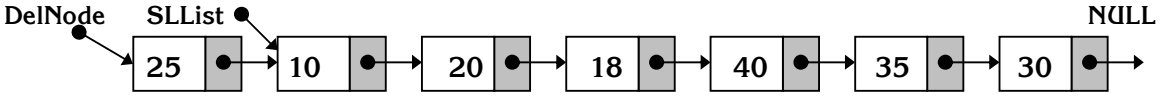
```
int  SLL_Delete_Node (SLL_Type  &SList, T  DelData)
   {  SLL_Type  DelNode = SList;
      SLL_Type  PreDelNode = NULL;
      while (DelNode != NULL)
         {  if (DelNode->Key == DelData)
               break;
            PreDelNode = DelNode;
            DelNode = DelNode->NextNode;
         }
      if (DelNode == NULL)
         return (-1);
      if (PreDelNode == NULL)
         SList = SList->NextNode;
      else
         PreDelNode->NextNode = DelNode->NextNode;
      DelNode->NextNode = NULL;
      delete DelNode;
      return (1);
   }
```

- Minh họa thuật toán:

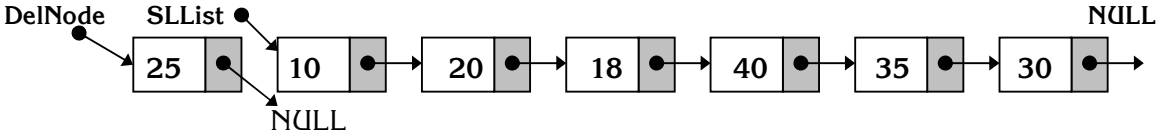
+ Giả sử chúng ta cần hủy nút có thành phần dữ liệu là 25: DelData = 25



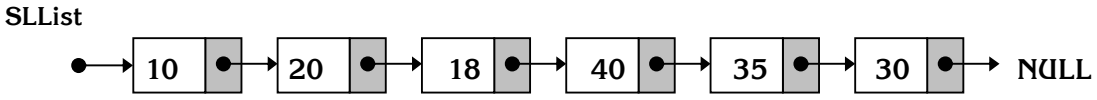
SLList = SLList->NextNode



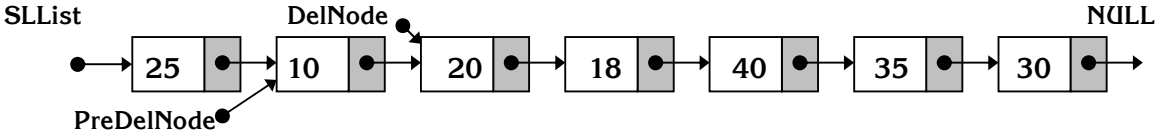
DelNode->NextNode = NULL



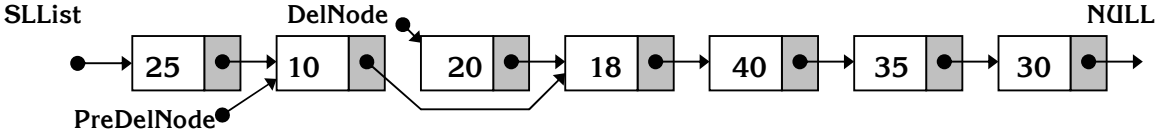
Kết quả sau khi hủy:



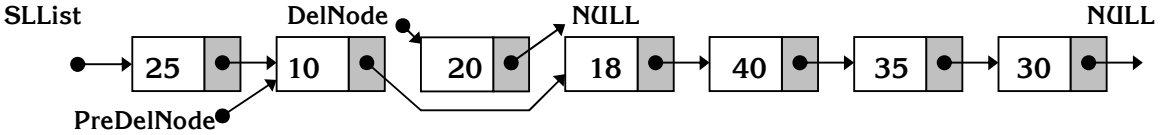
+ Bây giờ giả sử chúng ta cần hủy nút có thành phần dữ liệu là 20: DelData = 20



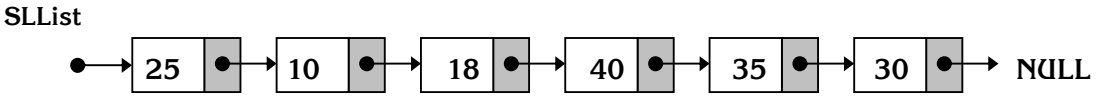
PreDelNode->NextNode = DelNode->Next



DelNode->Next = NULL



Kết quả sau khi hủy:



g. Hủy danh sách:

Thao tác này thực chất là thực hiện nhiều lần thao tác hủy một nút.

- Thuật toán:

B1: IF (SLList = NULL)
  Thực hiện Bkt
B2: TempNode = SLList
B3: SLList = SLList->NextNode
B4: TempNode->NextNode = NULL
B5: delete TempNode
B6: Lặp lại B1
Bkt: Kết thúc

- Cài đặt:

Hàm SLL_Delete có prototype:

```
void  SLL_Delete (SLL_Type  &SList);
```

Hàm thực hiện việc hủy toàn bộ danh sách SList.

Nội dung của hàm như sau:

```
void  SLL_Delete (SLL_Type  &SList)
   {  SLL_Type  TempNode = SList;
      while (SList != NULL)
         {  SList = SList->NextNode;
            TempNode->NextNode = NULL;
```

```
            delete  TempNode;
            TempNode = SList;
        }
    return ;
}
```

h. Tạo mới danh sách/ Nhập danh sách:

Việc tạo mới một danh sách liên kết đơn thực chất là chúng ta liên tục thực hiện thao tác thêm một phần tử vào danh sách mà ban đầu danh sách này là một danh sách rỗng. Có thể sử dụng một trong ba hàm thêm phần tử để thêm phần tử, ở đây chúng ta sử dụng hàm SLL_Add_First.

Giả sử chúng ta cần tạo danh sách liên kết đơn có N phần tử.

- Thuật toán:

B1: SLL_Initialize(SLList)
B2: i = 1
B3: IF (i > N)
    Thực hiện Bkt
B4: NewData = InputNewData() // Nhập giá trị cho biến NewData
B5: SLL_Add_First(SLList, NewData)
B6: i++
B7: Lặp lại B3
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm SLL_Create có prototype:

```
SLL_Type   SLL_Create(SLL_Type  &SList, int N);
```

Hàm tạo danh sách liên kết đơn có N nút quản lý bởi địa chỉ nút đầu tiên thông qua SList. Hàm trả về địa chỉ của nút đầu tiên trong danh sách nếu việc tạo thành công, ngược lại hàm trả về con trỏ NULL.

Nội dung của hàm như sau:

```
SLL_Type   SLL_Create(SLL_Type   &SList, int N)
  {  SLL_Initialize(SList);
     T  NewData;
     for (int i = 0; i < N; i++)
        {  NewData = InputNewData();
           if (SLL_Add_First(SList, NewData) == NULL)
              {  SLL_Delete (SList);
                 break;
              }
        }
     return (SList);
  }
```

☛ **Lưu ý:**

Hàm InputNewData thực hiện việc nhập vào nội dung của một biến có kiểu dữ liệu T và trả về giá trị mới nhập vào. Tùy vào từng trường hợp cụ thể mà chúng ta viết hàm InputNewData cho phù hợp.

i. Tách một danh sách thành nhiều danh sách:

Tương tự như danh sách đặc, việc tách một danh sách liên kết đơn thành nhiều danh sách liên kết đơn khác nhau cũng có nhiều tiêu thức khác nhau mà chúng ta sẽ thực hiện theo các cách khác nhau. Ngoài ra việc tách cũng sẽ khác nhau trong trường hợp có hay không giữ lại danh sách ban đầu. Ở đây chúng ta thực hiện việc tách các nút trong danh sách liên kết đơn SLList thành hai danh sách liên kết đơn con SLList và SLList1 luân phiên theo các đường chạy tự nhiên và không giữ lại danh sách liên kết ban đầu. Các trường hợp khác sinh viên tự vận dụng để thao tác.

- Thuật toán:

B1: CurNode = SLList
B2: SLList1 = SLList
B3: LastNode1 = NULL, LastNode2 = NULL

// Cắt các nút từ sau đường chạy tự nhiên thứ nhất về SLList1

B4: IF (CurNode = NULL OR CurNode->NextNode = NULL)
    Thực hiện Bkt
B5: IF (CurNode->Key > CurNode->NextNode->Key)
    B5.1: LastNode1 = CurNode
    B5.2: SLList1 = SLList1->NextNode
    B5.3: CurNode = CurNode->NextNode
    B5.4: LastNode1->NextNode = NULL
    B5.5: Thực hiện B8
B6: CurNode = CurNode->NextNode, SLList1 = SLList1->NextNode
B7: Lặp lại B4

// Cắt các nút từ sau đường chạy tự nhiên thứ hai về SLList

B8: IF (CurNode = NULL OR CurNode->NextNode = NULL)
    Thực hiện Bkt
B9: IF (CurNode->Key > CurNode->NextNode->Key)
    B9.1: LastNode2 = CurNode
    B9.2: CurNode = CurNode->NextNode
    B9.3: LastNode2->NextNode = NULL
    B9.4: Thực hiện B12
B10: CurNode = CurNode->NextNode
B11: Lặp lại B8

// Phân phối (giữ lại) đường chạy kế tiếp trong SLList

B12: LastNode1->NextNode = CurNode
B13: IF (CurNode = NULL OR CurNode->NextNode = NULL)
    Thực hiện Bkt
B14: IF (CurNode->Key > CurNode->NextNode->Key)
    B14.1: LastNode1 = CurNode
    B14.2: CurNode = CurNode->NextNode

B14.3: LastNode1->NextNode = NULL
B14.4: Thực hiện B17
B15: CurNode = CurNode->NextNode
B16: Lặp lại B13

// Phân phối (giữ lại) đường chạy kế tiếp trong SLList1

B17: LastNode2->NextNode = CurNode
B18: IF (CurNode = NULL OR CurNode->NextNode = NULL)
Thực hiện Bkt
B19: IF (CurNode->Key > CurNode->NextNode->Key)
B19.1: LastNode2 = CurNode
B19.2: CurNode = CurNode->NextNode
B19.3: LastNode2->NextNode = NULL
B19.4: Lặp lại B12
B20: CurNode = CurNode->NextNode
B21: Lặp lại B18
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm SLL_Split có prototype:

```
SLL_Type  SLL_Split(SLL_Type  &SList, SLL_Type &SList1);
```

Hàm thực hiện việc phân phối bớt các đường chạy tự nhiên trong SList sang SList1. Hàm trả về con trỏ trỏ tới địa chỉ phần tử đầu tiên trong SList1.

Nội dung của hàm như sau:

```
SLL_Type  SLL_Split(SLL_Type  &SList, SLL_Type &SList1)
  {  SList1 = SList;
     if (SList1 == NULL)
        return (NULL);
     SLL_Type  Last1;
     SLL_Type  Last2;
     while (SList1->NextNode != NULL)
        {  if (SList1->Key > SList1->NextNode->Key)
              break;
           SList1 = SList1->NextNode;
        }
     if (SList1->NextNode != NULL)
        Last1 = SList1;
     SList1 = SList1->NextNode;
     Last1->NextNode = NULL;
     SLL_Type  CurNode = SList1;
     if (CurNode == NULL)
        return (NULL);
     while (CurNode->NextNode != NULL)
        {  if (CurNode->Key > CurNode->NextNode->Key)
              break;
           CurNode = CurNode->NextNode;
```

```
        }
    if (CurNode->NextNode == NULL)
        return (SList1);
    Last2 = CurNode;
    CurNode = CurNode->NextNode;
    Last2->NextNode = NULL;
    while (CurNode != NULL)
        {  Last1->NextNode = CurNode;
            if (CurNode->NextNode == NULL)
                break;
            while (CurNode->NextNode != NULL)
                { if (CurNode->Key > CurNode->NextNode->Key)
                        break;
                    Cur Node = CurNode->NextNode;
                }
            if (CurNode->NextNode == NULL)
                break;
            Last1 = CurNode;
            CurNode = CurNode->NextNode;
            Last1->NextNode = NULL;
            Last2->NextNode = CurNode;
            if (CurNode->NextNode == NULL)
                break;
            while (CurNode->NextNode != NULL)
                { if (CurNode->Key > CurNode->NextNode->Key)
                        break;
                    Cur Node = CurNode->NextNode;
                }
            if (CurNode->NextNode == NULL)
                break;
            Last2 = CurNode;
            CurNode = CurNode->NextNode;
            Last2->NextNode = NULL;
        }
    return (SList1);
}
```

j. Nhập nhiều danh sách thành một danh sách:

Tương tự, việc nhập nhiều danh sách thành một danh sách chúng ta thực hiện theo hai trường hợp khác nhau:

+ Ghép nối đuôi các danh sách lại với nhau;
+ Trộn xen lẫn các phần tử trong danh sách con vào thành một danh sách lớn theo một trật tự nhất định.

Ngoài ra việc nhập có thể giữ lại các danh sách con ban đầu hoặc không giữ lại các danh sách con ban đầu. Ở đây chúng ta trình bày theo cách không giữ lại các danh sách con ban đầu và trình bày theo hai trường hợp:

+ Ghép nối đuôi hai danh sách lại với nhau;

+ Trộn hai danh sách lại với nhau theo các đường chạy tự nhiên thành một danh sách có chiều dài lớn hơn.

Giả sử chúng ta cần nhập hai danh sách SLList1, SLList2 lại với nhau.

- Thuật toán ghép danh sách SLList2 vào sau SLList1:

B1: IF (SLList1 = NULL)
    B1.1: SLList1 = SLList2
    B1.2: Thực hiện Bkt
B2: IF (SLList2 = NULL)
    Thực hiện Bkt

// Lấy địa chỉ nút cuối cùng trong SLList1

B3: LastNode = SLList1
B4: IF (LastNode->NextNode = NULL)
    Thực hiện B7
B5: LastNode = LastNode->NextNode
B6: Lặp lại B4

// Ghép SLList2 vào sau LastNode

B7: LastNode->NextNode = SLList2
Bkt: Kết thúc

- Thuật toán trộn danh sách SLList2 và SLList1 thành SLList theo các đường chạy tự nhiên:

B1: IF (SLList1 = NULL)
    B1.1: SLList = SLList2
    B1.2: Thực hiện Bkt
B2: IF (SLList2 = NULL)
    B2.1: SLList = SLList1
    B2.2: Thực hiện Bkt

// Lấy nút có dữ liệu nhỏ hơn trong 2 nút đầu của 2 danh sách đưa về SLList

B3: IF (SLList1->Key ≤ SLList2->Key)
    B3.1: TempNode = SLList1
    B3.2: SLList1 = SLList1->NextNode
B4: ELSE
    B4.1: TempNode = SLList2
    B4.2: SLList2 = SLList2->NextNode
B5: TempNode->NextNode = NULL
B6: IF (SLList1 = NULL)
    B6.1: TempNode->NextNode = SLList2
    B6.2: Thực hiện Bkt
B7: IF (SLList2 = NULL)
    B7.1: TempNode->NextNode = SLList1
    B7.2: Thực hiện Bkt
B8: IF (SLList1->Key ≤ SLList2->Key) AND (TempNode->Key ≤ SLList1->Key)
    B8.1: MinNode = SLList1
    B8.2: SLList1 = SLList1->NextNode

B9: ELSE
B9.1: MinNode = SLList2
B9.2: SLList2 = SLList2->NextNode
B10: TempNode->NextNode = MinNode
B11: MinNode->NextNode = NULL
B12: TempNode = MinNode
B13: Lặp lại B6
Bkt: Kết thúc

- Cài đặt:

Các hàm nhập danh sách có prototype:

```
SLL_Type  SLL_Concat (SLL_Type  &SList1, SLL_Type  &SList2);
```

```
SLL_Type SLL_Merge(SLL_Type  &SList1, SLL_Type  &SList2, SLL_Type &SList);
```

Hàm thực hiện việc nhập các nút trong hai danh sách SList1, SList2 thành một danh sách theo thứ tự như hai thuật toán vừa trình bày. Hàm trả về địa chỉ của nút đầu của danh sách sau khi ghép.

Nội dung của các hàm như sau:

```
SLL_Type  SLL_Concat (SLL_Type  &SList1, SLL_Type  &SList2)
   {  if (SList1 == NULL)
         {  SList1 = SList2;
            return (SList1);
         }
      if (SList2 == NULL)
         return (SList1);
      SLL_Type  LastNode = SList1;
      while (LastNode->NextNode != NULL)
         LastNode = LastNode->NextNode;
      LastNode->NextNode = SList2;
      return (SList1);
   }
//================================================================
   SLL_Type  SLL_Merge (SLL_Type  &SList1, SLL_Type  &SList2, SLL_Type &SList)
   {  if (SList1 == NULL)
         {  SList = SList2;
            return (SList);
         }
      if (SList2 == NULL)
         {  SList = SList1;
            return (SList);
         }
      SLL_Type  LastNode = NULL;
      SLL_Type  TempNode;
      while (SList1 != NULL && SList2 != NULL)
         {  if (SList1->Key <= SList2->Key)
               {  TempNode = SList1;
                  SList1 = SList1->NextNode;
```

```
                TempNode->NextNode = NULL;
                if (LastNode == NULL)
                   SList = LastNode = TempNode;
                else
                   { LastNode->NextNode = TempNode;
                     LastNode = TempNode;
                   }
                if (SList1 == NULL)
                   break;
                if (SList1->Key < LastNode->Key)
                   while (SList2 != NULL)
                      { LastNode->Next = SList2;
                        LastNode = LastNode->NextNode;
                        SList2 = SList2->NextNode;
                        LastNode->NextNode = NULL;
                        if (SList2 == NULL || SList2->Key < LastNode->Key)
                           break;
                      }
             }
          else
             { TempNode = SList2;
               SList2 = SList2->NextNode;
               TempNode->NextNode = NULL;
               if (LastNode == NULL)
                  SList = LastNode = TempNode;
               else
                  { LastNode->NextNode = TempNode;
                    LastNode = TempNode;
                  }
               if (SList2 == NULL)
                  break;
               if (SList2->Key < LastNode->Key)
                  while (SList1 != NULL)
                     { LastNode->Next = SList1;
                       LastNode = LastNode->NextNode;
                       SList1 = SList1->NextNode;
                       LastNode->NextNode = NULL;
                       if (SList1 == NULL || SList1->Key < LastNode->Key)
                          break;
                     }
             }
       }
    if (SList1 == NULL)
       LastNode->NextNode = SList2;
    else
       LastNode->NextNode = SList1;
    return (SList);
}
```

k. Sắp xếp thứ tự các phần tử trong danh sách:

Thao tác này chúng ta có thể vận dụng các thuật toán sắp xếp đã trình bày trong Chương 3 để sắp xếp dữ liệu trong danh sách liên kết đơn. Ở đây chúng ta chỉ trình bày sự vận dụng thuật toán trộn tự nhiên để sắp xếp.

Cũng cần lưu ý rằng đối với thao tác hoán vị hai phần tử thì chúng ta có thể hoán vị hoàn toàn hai nút hoặc chỉ hoán vị phần dữ liệu. Tuy nhiên việc hoán vị hoàn toàn hai nút sẽ phức tạp hơn.

- Thuật toán sắp xếp trộn tự nhiên:

B1: IF (SLL_Split(SLList, TempList) = NULL)
    Thực hiện Bkt
B2: SLL_Merge(SLList, TempList, SLList)
B3: Lặp lại B1
Bkt: Kết thúc

- Cài đặt:

Hàm SLL_Natural_Merge_Sort có prototype:

```
void  SLL_Natural_Merge_Sort (SLL_Type  &SList);
```

Hàm thực hiện việc sắp xếp thành phần dữ liệu của các nút trong danh sách SList theo thứ tự tăng dựa trên thuật toán trộn tự nhiên vừa trình bày.

Nội dung của hàm như sau:

```
void  SLL_Natural_Merge_Sort (SLL_Type  &SList)
   {  SLL_Type  TempList = NULL, List = NULL;
      while (SLL_Split(SList, TempList) != NULL)
         {  SLL_Merge(SList, TempList, List);
            SList = List;
         }
      return ;
   }
```

h. Sao chép một danh sách:

Thực chất thao tác này là chúng ta tạo mới danh sách NewList bằng cách duyệt qua các nút của SLList để lấy thành phần dữ liệu rồi tạo thành một nút mới và bổ sung nút mới này vào cuối danh sách NewList.

- Thuật toán:

B1: NewList = NULL
B2: CurNode = SLList
B3: IF (CurNode = NULL)
    Thực hiện Bkt
B4: SLL_Add_Last(NewList, CurNode->Key)
B5: CurNode = CurNode->NextNode
B6: Lặp lại B3
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm SLL_Copy có prototype:
SLL_Type SLL_Copy (SLL_Type SList, SLL_Type &NewList);

Hàm thực hiện việc sao chép nội dung danh sách SList thành danh sách NewList có cùng nội dung thành phần dữ liệu theo thứ tự của các nút trong SList. Hàm trả về địa chỉ nút đầu trong danh sách mới nếu việc sao chép thành công, ngược lại hàm trả về con trỏ NULL.

Nội dung của hàm như sau:

```
SLL_Type SLL_Copy (SLL_Type SList, SLL_Type &NewList)
  { NewList = NULL;
    SLL_Type CurNode = SList;
    while (CurNode != NULL)
      { SLL_Type NewNode = SLL_Add_Last(NewList, CurNode->Key);
        if (NewNode == NULL)
          { SLL_Delelte(NewList);
            break;
          }
        CurNode = CurNode->NextNode;
      }
    return (NewList);
  }
```

4.4.3. Danh sách liên kết kép (Doubly Linked List)

A. Cấu trúc dữ liệu:

Nếu như vùng liên kết của danh sách liên kết đơn có 01 mối liên kết với 01 phần tử khác trong danh sách thì vùng liên kết trong danh sách liên đôi có 02 mối liên kết với 02 phần tử khác trong danh sách, cấu trúc dữ liệu của mỗi nút trong danh sách liên kết đôi như sau:

```
typedef struct DLL_Node
  { T Key;
    InfoType Info;
    DLL_Node * NextNode; // Vùng liên kết quản lý địa chỉ phần tử kế tiếp nó
    DLL_Node * PreNode; // Vùng liên kết quản lý địa chỉ phần tử trước nó
  }  DLL_OneNode;
```

Ở đây chúng ta cũng giả thiết rằng vùng dữ liệu của mỗi phần tử trong danh sách liên kết đôi chỉ bao gồm một thành phần khóa nhận diện (Key) cho phần tử đó. Do vậy, cấu trúc dữ liệu trên có thể viết lại đơn giản như sau:

```
typedef struct DLL_Node
  { T Key;
    DLL_Node * NextNode; // Vùng liên kết quản lý địa chỉ phần tử kế tiếp nó
    DLL_Node * PreNode; // Vùng liên kết quản lý địa chỉ phần tử trước nó
  }  DLL_OneNode;

typedef DLL_OneNode * DLL_Type;
```

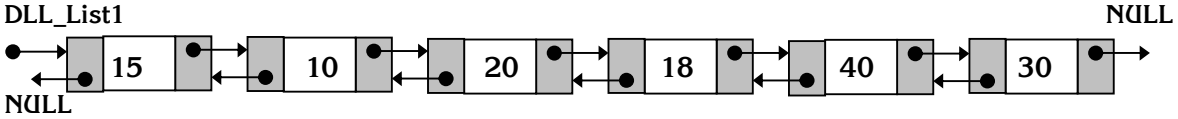
Có nhiều phương pháp khác nhau để quản lý các danh sách liên kết đôi và tương ứng với các phương pháp này sẽ có các cấu trúc dữ liệu khác nhau, cụ thể:

- Quản lý địa chỉ phần tử đầu danh sách:

Cách này hoàn toàn tương tự như đối với danh sách liên kết đơn.

```
DLL_Type  DLL_List1;
```

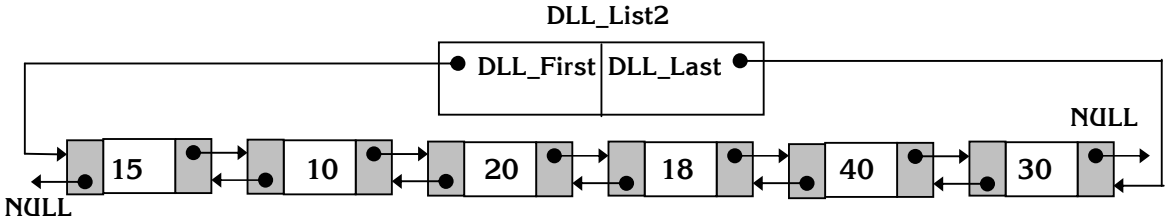
Hình ảnh minh họa:



- Quản lý địa chỉ phần tử đầu và cuối danh sách:

```
typedef  struct  DLL_PairNode
   {  DLL_Type  DLL_First;
      DLL_Type  DLL_Last;
   }      DLLP_Type;

DLLP_Type  DLL_List2;
```

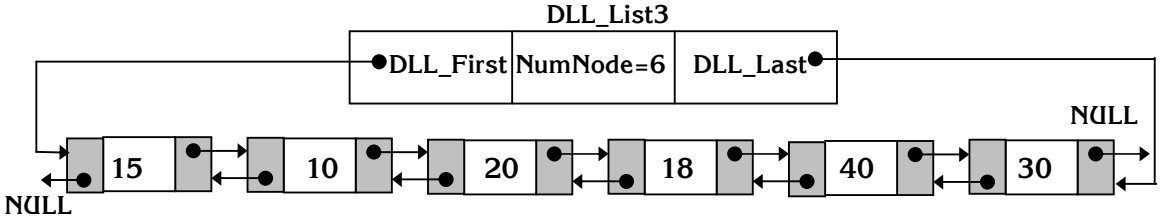
Hình ảnh minh họa:



- Quản lý địa chỉ phần tử đầu, địa chỉ phần tử cuối và số phần tử trong danh sách:

```
typedef  struct  DLL_PairNNode
   {  DLL_Type  DLL_First;
      DLL_Type  DLL_Last;
      unsigned  NumNode;
   }      DLLPN_Type;

DLLPN_Type  DLL_List3;
```

Hình ảnh minh họa:



B. Các thao tác trên danh sách liên kết đôi:

Cũng như trong phần danh sách liên kết đơn, các thao tác tương ứng với mỗi cách quản lý khác nhau của danh sách liên kết đôi có sự khác nhau về mặt chi tiết song nội dung cơ bản ít có sự khác nhau. Do vậy, ở đây chúng ta chỉ trình bày các thao tác theo cách quản lý thứ hai (quản lý các địa chỉ của hai nút đầu và cuối danh sách liên kết đôi), các thao tác này trên các cách quản lý khác sinh viên tự vận dụng để điều chỉnh cho thích hợp.

a. Khởi tạo danh sách (Initialize):

Trong thao tác này chỉ đơn giản là chúng ta cho giá trị các con trỏ quản lý địa chỉ hai nút đầu và cuối danh sách liên kết đôi về con trỏ NULL. Hàm khởi tạo danh sách liên kết đôi như sau:

```
DLLP_Type  DLL_Initialize(DLLP_Type  &DList)
   {  DList.DLL_First = NULL;
      DList.DLL_Last = NULL;
      return (DList);
   }
```

Hình ảnh minh họa:

b. Tạo mới một phần tử / nút:

Giả sử chúng ta cần tạo mới một phần tử có thành phần dữ liệu là NewData.

- Thuật toán:

B1: DNode = new DLL_OneNode
B2: IF (DNode = NULL)
    Thực hiện Bkt
B3: DNode->NextNode = NULL
B4: DNode->PreNode = NULL
B5: DNode->Key = NewData
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm DLL_Create_Node có prototype: DLL_Type DLL_Create_Node(T NewData);

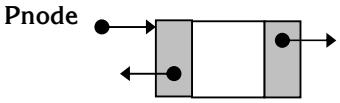
Hàm tạo mới một nút có thành phần dữ liệu là NewData, hàm trả về con trỏ trỏ tới địa chỉ của nút mới tạo. Nếu không đủ bộ nhớ để tạo, hàm trả về con trỏ NULL.

```
DLL_Type  DLL_Create_Node(T  NewData)
   {  DLL_Type  Pnode = new  DLL_OneNode;
      if (Pnode != NULL)
         {  Pnode->NextNode = NULL;
            Pnode->PreNode = NULL;
            Pnode->Key = NewData;
         }
      return (Pnode);
   }
```

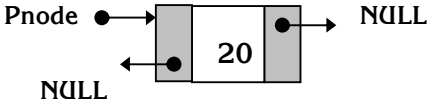
- Minh họa thuật toán:

Giả sử chúng ta cần tạo nút có thành phần dữ liệu là 20: NewData = 20

Pnode = new DLL_OneNode



Pnode->NextNode = NULL
Pnode->PreNode = NULL
Pnode->Key = NewData



c. Thêm một phần tử vào trong danh sách:

Giả sử chúng ta cần thêm một phần tử có giá trị thành phần dữ liệu là NewData vào trong danh sách. Việc thêm có thể diễn ra ở đầu, cuối hay ở giữa danh sách liên kết. Do vậy, ở đây chúng ta trình bày 3 thao tác thêm riêng biệt nhau:

- Thuật toán thêm phần tử vào đầu danh sách liên kết đôi:

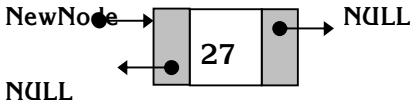
B1: NewNode = DLL_Create_Node (NewData)
B2: IF (NewNode = NULL)
    Thực hiện Bkt
B3: IF (DLL_List.DLL_First = NULL) // Danh sách rỗng
    B3.1: DLL_List.DLL_First = NewNode
    B3.2: DLL_List.DLL_Last = NewNode
    B3.3: Thực hiện Bkt
B4: NewNode->NextNode = DLL_List.DLL_First // Nối DLL_First vào
B5: DLL_List.DLL_First->PreNode = NewNode // sau NewNode

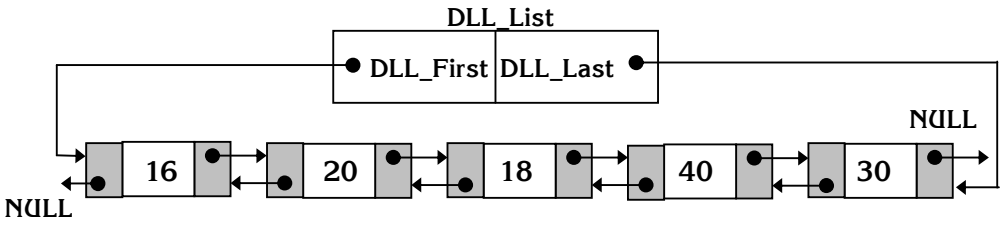
// Chuyển vai trò đứng đầu của NewNode cho DLL_First

B6: DLL_List.DLL_First = NewNode
Bkt: Kết thúc

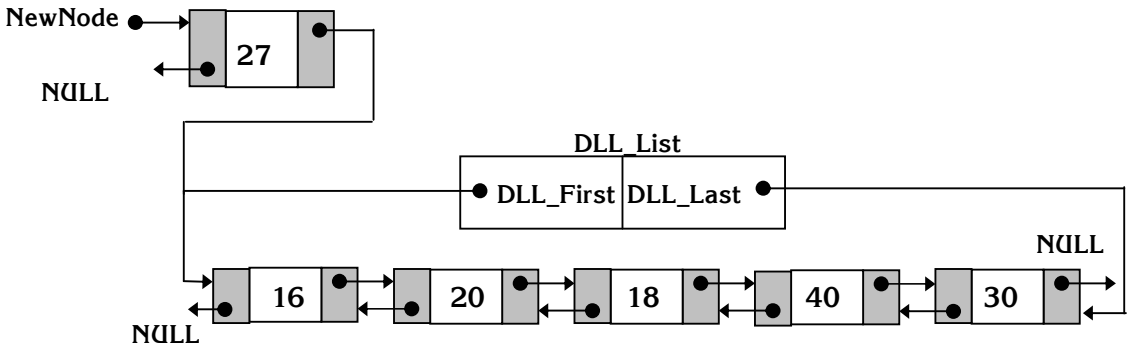
- Minh họa thuật toán:

Giả sử chúng ta cần thêm nút có thành phần dữ liệu là 27: NewData = 27

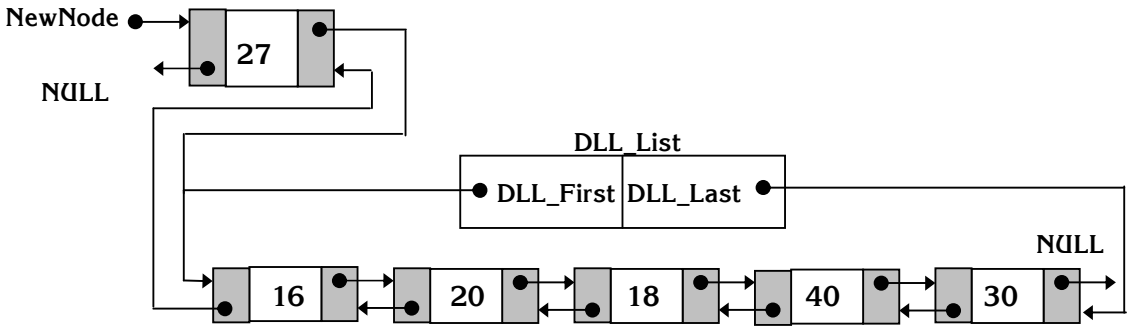




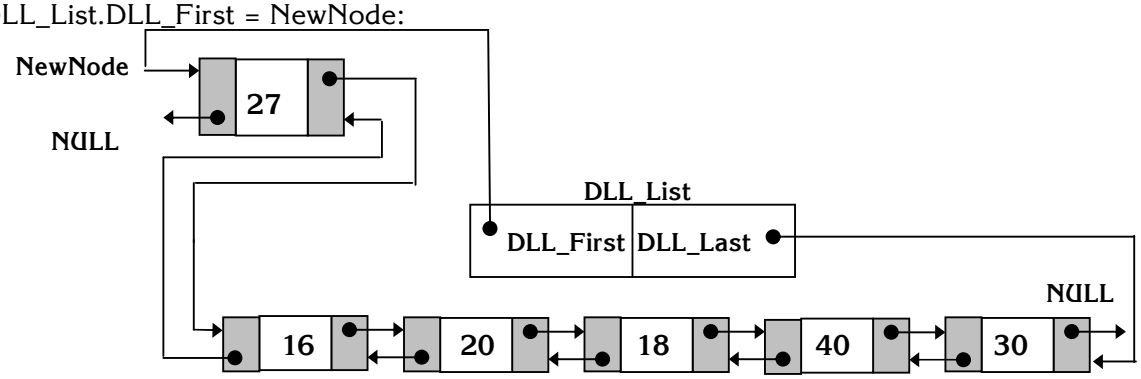
NewNode->NextNode = DLL_List.DLL_First:



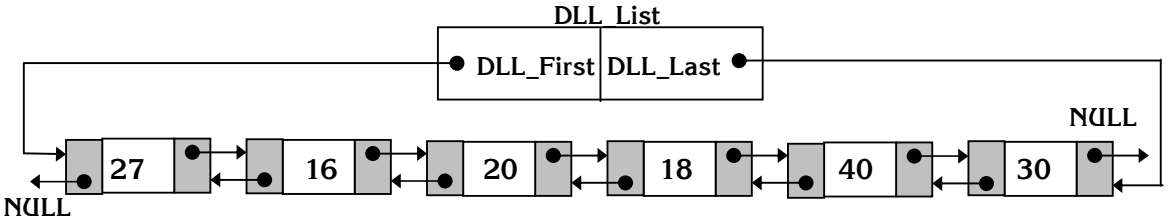
DLL_List.DLL_First->PreNode = NewNode:



DLL_List.DLL_First = NewNode:



Kết quả sau khi chèn:



- Thuật toán thêm phần tử vào cuối danh sách liên kết đôi:

B1: NewNode = DLL_Create_Node (NewData)
B2: IF (NewNode = NULL)
    Thực hiện Bkt
B3: IF (DLL_List.DLL_First = NULL) // Danh sách rỗng

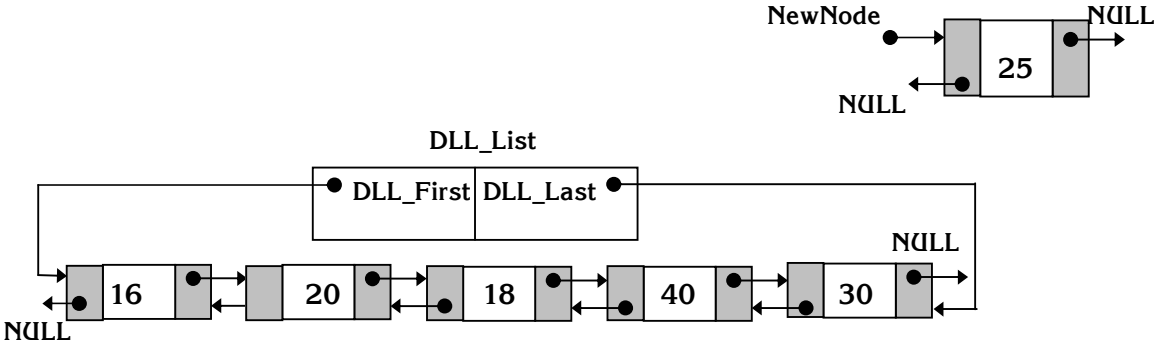
B3.1: DLL_List.DLL_First = NewNode
B3.2: DLL_List.DLL_Last = NewNode
B3.3: Thực hiện Bkt
B4: DLL_List.DLL_Last->NextNode = NewNode // Nối NewNode vào
B5: NewNode->PreNode = DLL_List.DLL_Last // sau DLL_Last

// Chuyển vai trò đứng cuối của NewNode cho DLL_Last

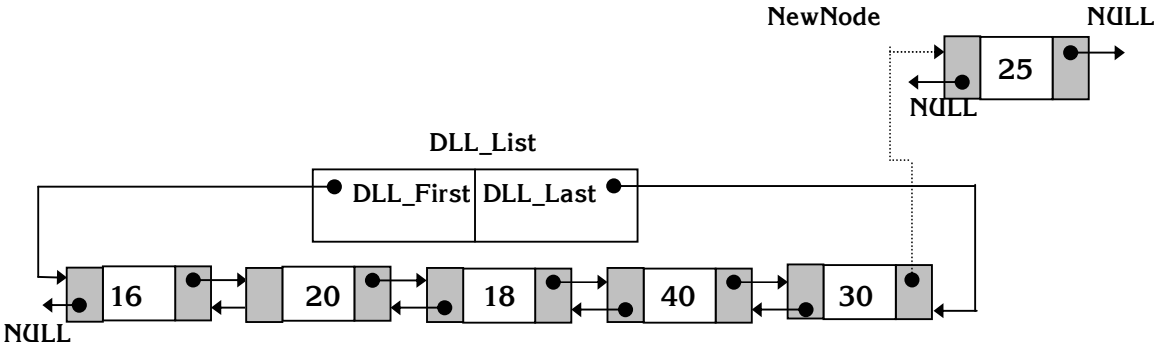
B6: DLL_List.DLL_Last = NewNode
Bkt: Kết thúc

- Minh họa thuật toán:

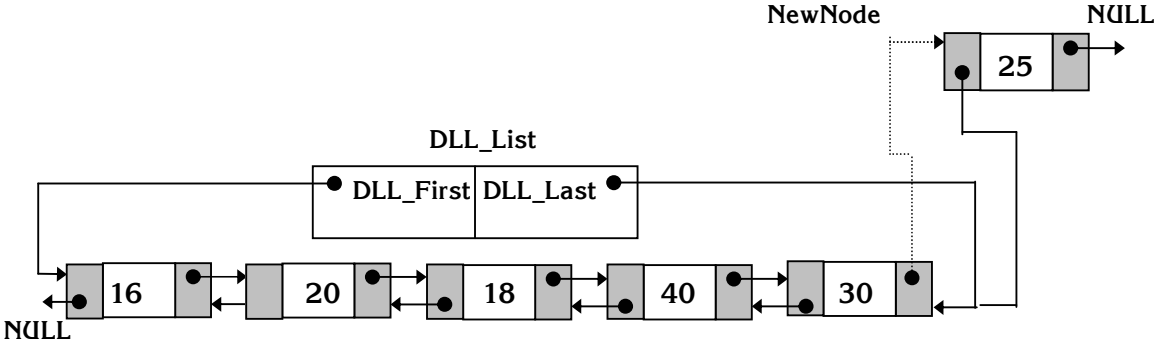
Giả sử chúng ta cần thêm nút có thành phần dữ liệu là 25: NewData = 25



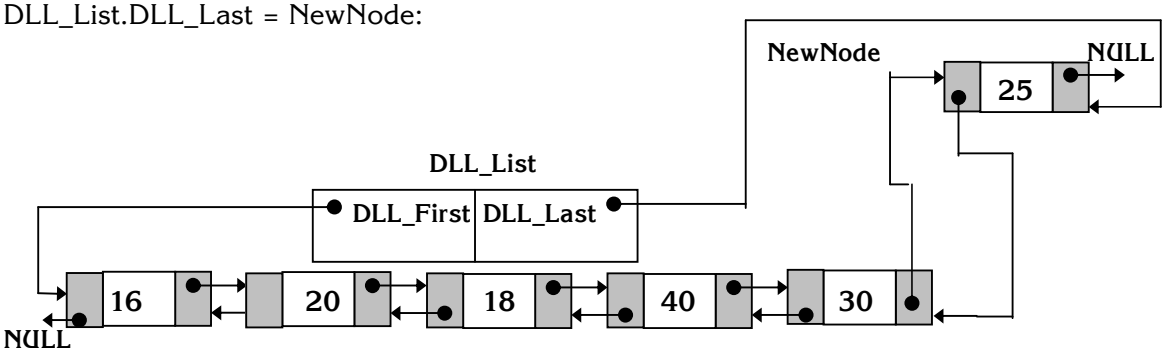
DLL_List.DLL_Last->NextNode = NewNode:



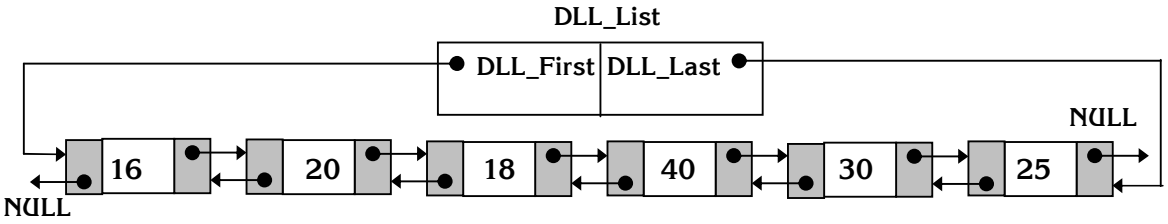
NewNode->PreNode = DLL_List.DLL_Last



DLL_List.DLL_Last = NewNode:



Kết quả sau khi chèn:



- Thuật toán thêm phần tử vào giữa danh sách liên kết đôi:

Giả sử chúng ta cần thêm một phần tử có giá trị thành phần dữ liệu là NewData vào trong danh sách DLL_List vào ngay sau nút có địa chỉ InsNode. Trong thực tế nhiều khi chúng ta phải thực hiện thao tác tìm kiếm để xác định địa chỉ InsNode, ở đây giả sử chúng ta đã xác định được địa chỉ này.

B1: IF (InsNode->NextNode = NULL) // Thêm vào cuối DSLK
    B1.1: DLL_Add_Last (DLL_List, NewData)
    B1.2: Thực hiện Bkt

B2: NewNode = DLL_Create_Node (NewData)
B3: IF (NewNode = NULL)
    Thực hiện Bkt

// Nối các nút kế sau InsNode vào sau NewNode

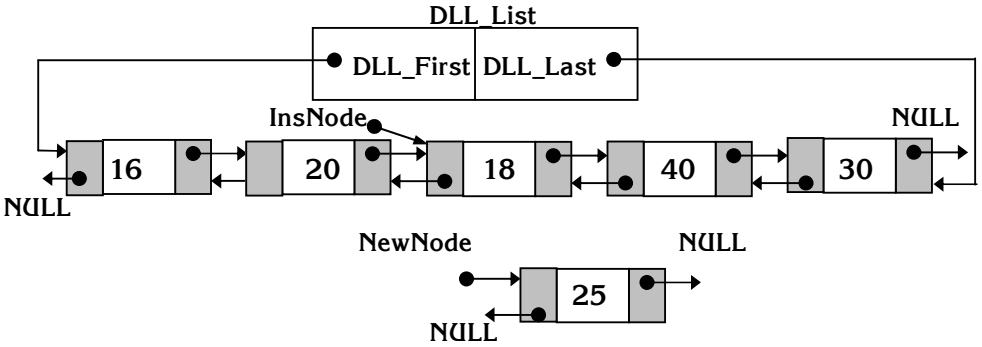
B4: NewNode->NextNode = InsNode->NextNode
B5: InsNode->NextNode->PreNode = NewNode

// Chuyển mối liên kết giữa InsNode với nút kế của nó về NewNode

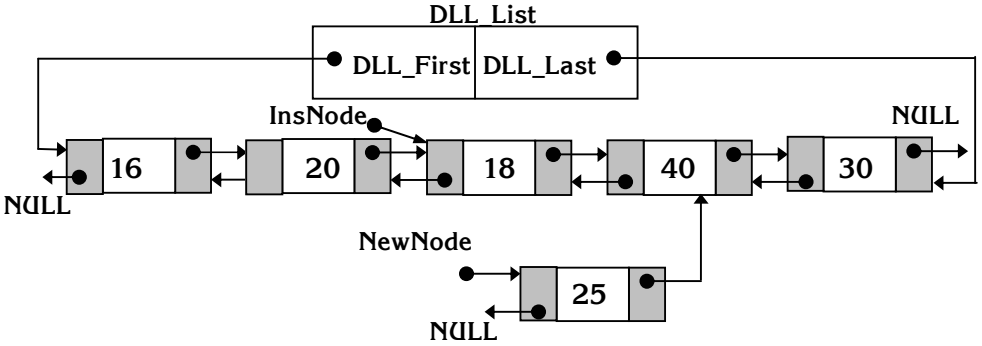
B6: InsNode->NextNode = NewNode
B7: NewNode->PreNode = InsNode
Bkt: Kết thúc

- Minh họa thuật toán:

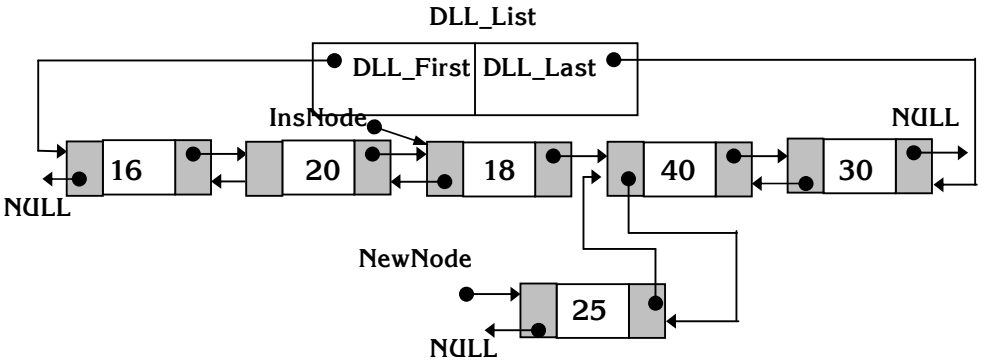
Giả sử chúng ta cần thêm nút có thành phần dữ liệu là 25 vào sau nút có địa chỉ InsNode như sau: NewData = 25



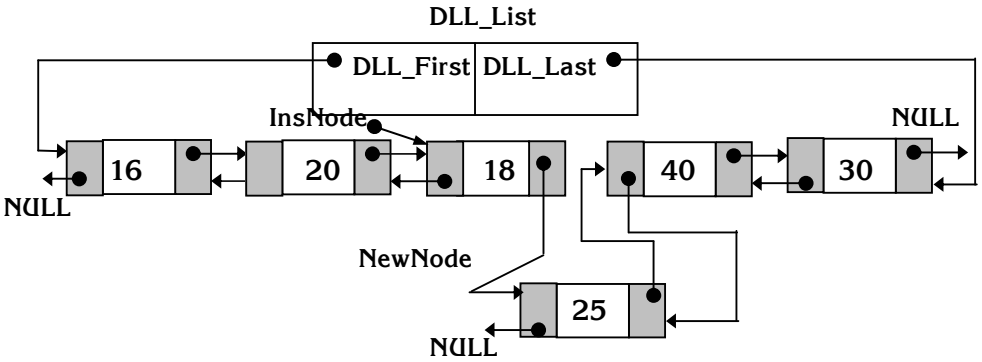
NewNode->NextNode = InsNode->NextNode:



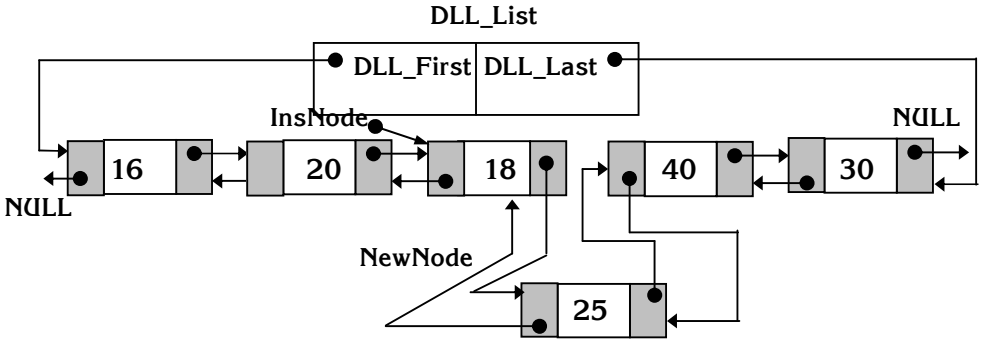
InsNode->NextNode->PreNode = NewNode:



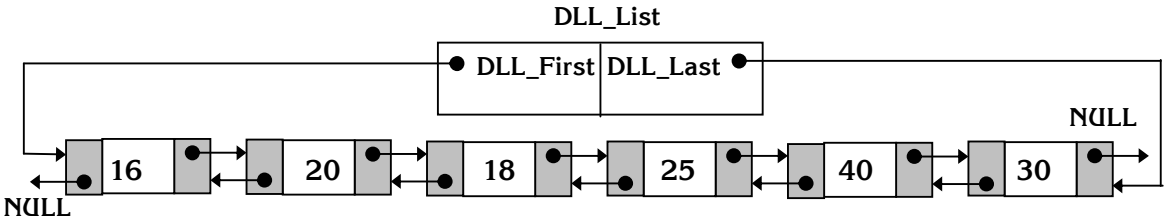
InsNode->NextNode = NewNode:



NewNode->PreNode = InsNode



Kết quả sau khi chèn:



- Cài đặt thuật toán:

Các hàm thêm phần tử tương ứng với các trường hợp có prototype như sau:

DLL_Type DLL_Add_First(DLLP_Type &DList, T NewData);

DLL_Type DLL_Add_Last(DLLP_Type &DList, T NewData);

DLL_Type DLL_Add_Mid(DLLP_Type &DList, T NewData, DLL_Type &InsNode);

Hàm thực hiện việc chèn phần tử có giá trị thành phần dữ liệu NewData vào trong danh sách liên kết đôi quản lý bởi hai con trỏ đầu và cuối danh sách trong DList tương ứng với 3 trường hợp: Thêm đầu, thêm cuối, thêm giữa. Các hàm trả về giá trị là một địa chỉ của nút vừa mới thêm nếu việc thêm thành công. Trong trường hợp ngược lại, các hàm trả về con trỏ NULL.

Riêng đối với trường hợp thêm giữa, hàm DLL_Add_Mid thực hiện việc thêm vào ngay sau nút có địa chỉ InsNode. Nội dung của các hàm như sau:

```
DLL_Type  DLL_Add_First(DLLP_Type &DList, T NewData)
   {  DLL_Type NewNode = DLL_Create_Node(NewData);
      if (NewNode == NULL)
         return (NULL);
      if (DList.DLL_First == NULL)
         DList.DLL_First = DList.DLL_Last = NewNode;
      else
         {  NewNode->NextNode = DList.DLL_First;
            DList.DLL_First->PreNode = NewNode;
            DList.DLL_First = NewNode;
         }
      return (NewNode);
   }
//=====================================================================
   DLL_Type  DLL_Add_Last(DLLP_Type &DList, T NewData)
```

```
    {  DLL_Type  NewNode = DLL_Create_Node(NewData);
       if (NewNode == NULL)
          return (NULL);
       if (DList.DLL_Last == NULL)
          DList.DLL_First = DList.DLL_Last = NewNode;
       else
          {  DList.DLL_Last->NextNode = NewNode;
             NewNode->PreNode = DList.DLL_Last;
             DList.DLL_Last = NewNode;
          }
       return (NewNode);
    }
//================================================================
   DLL_Type  DLL_Add_Mid(DLLP_Type &DList, T  NewData, DLL_Type  &InsNode)
      {  DLL_Type  NewNode = DLL_Create_Node(NewData);
         if (NewNode == NULL)
            return (NULL);
         if (InsNode->NextNode == NULL)
            {  InsNode->NextNode = NewNode;
               NewNode->PreNode = InsNode;
               DList.DLL_Last = NewNode;
            }
         else
            { NewNode->NextNode = InsNode->NextNode;
               InsNode->NextNode->PreNode = NewNode;
               InsNode->NextNode = NewNode;
               NewNode->PreNode = InsNode;
            }
         return (NewNode);
      }
```

d. Duyệt qua các nút trong danh sách:

Thao tác này nhằm nhiều mục đích, ở đây đơn giản chúng ta chỉ duyệt để xem nội dung thành phần dữ liệu trong danh sách. Thuật toán này hoàn toàn tương tự như trong danh sách liên kết đơn.

- Thuật toán:

B1: CurNode = DLL_List.First
B2: IF (CurNode = NULL)
    Thực hiện Bkt
B3: OutputData(CurNode->Key) // Xuất giá trị thành phần dữ liệu trong 1 nút
B4: CurNode = CurNode->NextNode
B5: Lặp lại B2
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm DLL_Travelling có prototype:

```
void   DLL_Travelling(DLLP_Type  DList);
```

Hàm duyệt qua các nút trong danh sách liên kết đôi quản lý bởi hai địa chỉ nút đầu tiên và nút cuối cùng thông qua DList để xem nội dung thành phần dữ liệu của mỗi nút.

Nội dung của hàm như sau:

```
void   DLL_Travelling (DLLP_Type  DList)
   {  DLL_Type  CurNode = DList.DLL_First;
      while (CurNode != NULL)
         {  OutputData(CurNode->Key);
            CurNode = CurNode->NextNode;
         }
      return;
   }
```

☞ **Lưu ý:**

Hàm OutputData thực hiện việc xuất nội dung của một biến có kiểu dữ liệu T. Tùy vào từng trường hợp cụ thể mà chúng ta viết hàm OutputData cho phù hợp.

e. Tìm kiếm một phần tử trong danh sách:

Giả sử chúng ta cần tìm kiếm xem trong danh sách liên kết đôi có tồn tại nút có thành phần dữ liệu là SearchData hay không. Thao tác này chúng ta vận dụng thuật toán tìm tuyến tính để tìm kiếm.

- Thuật toán:

```
B1: CurNode = DLL_List.DLL_First
B2: IF (CurNode = NULL OR CurNode->Key = SearchData)
       Thực hiện Bkt
B3: CurNode = CurNode->NextNode
B4: Lặp lại B2
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm DLL_Searching có prototype:

```
DLL_Type   DLL_Searching(DLLP_Type  DList, T  SearchData);
```

Hàm thực hiện việc tìm kiếm nút có thành phần dữ liệu là SearchData trên danh sách liên kết đôi quản lý bởi hai địa chỉ nút đầu tiên và nút cuối cùng thông qua DList. Hàm trả về địa chỉ của nút đầu tiên trong danh sách được tìm thấy, ngược lại hàm trả về con trỏ NULL.

Nội dung của hàm như sau:

```
DLL_Type   DLL_Searching(DLLP_Type  DList, T  SearchData)
   {  DLL_Type  CurNode = DList.DLL_First;
      while (CurNode != NULL)
         {  if (CurNode->Key == SearchData)
               break;
            CurNode = CurNode->NextNode;
         }
```

```
    return (CurNode);
}
```

f. Loại bỏ bớt một phần tử ra khỏi danh sách:

Giả sử chúng ta cần loại bỏ phần tử có giá trị thành phần dữ liệu là DelData trong danh sách liên kết đôi, Để thực hiện điều này trước tiên chúng ta phải thực hiện thao tác tìm kiếm địa chỉ của nút có thành phần dữ liệu là DelData, sau đó mới thực hiện thao tác loại bỏ nếu tìm thấy.

- Thuật toán:

```
// Tìm kiếm nút có Key là DelData trong danh sách
B1: DelNode = DLL_Searching(DLL_List, DelData)
B2: IF (DelNode = NULL)
        Thực hiện Bkt

// Loại bỏ nút tại địa chỉ DelNode ra khỏi danh sách
B3: IF (DelNode->PreNode = NULL AND DelNode->NextNode = NULL)
      B3.1: DLL_List.DLL_First = DLL_List.DLL_Last = NULL
      B3.2: Thực hiện B8
B4: IF (DelNode->PreNode = NULL)        // Loại bỏ nút đầu tiên trong danh sách
      B4.1: DLL_List.DLL_First = DLL_List.DLL_First->NextNode
      B4.2: DLL_List.DLL_First->PreNode = NULL
      B4.3: Thực hiện B8
B5: IF (DelNode->NextNode = NULL) // Loại bỏ nút cuối cùng trong danh sách
      B5.1: DLL_List.DLL_Last = DLL_List.DLL_Last->PreNode
      B5.2: DLL_List.DLL_Last->NextNode = NULL
      B5.3: Thực hiện B8

// Liên kết các nốt trước và sau DelNode với nhau
B6: DelNode->PreNode->NextNode = DelNode->NextNode
B7: DelNode->NextNode->PreNode = DelNode->PreNode

//Bỏ mối liên kết giữa DelNode với hai nút trước và sau nó, và hủy DelNode
B8: DelNode->NextNode = DelNode->PreNode = NULL
B9: delete DelNode
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm DLL_Delete_Node có prototype:

```
int  DLL_Delete_Node (DLLP_Type  &DList, T  DelData);
```

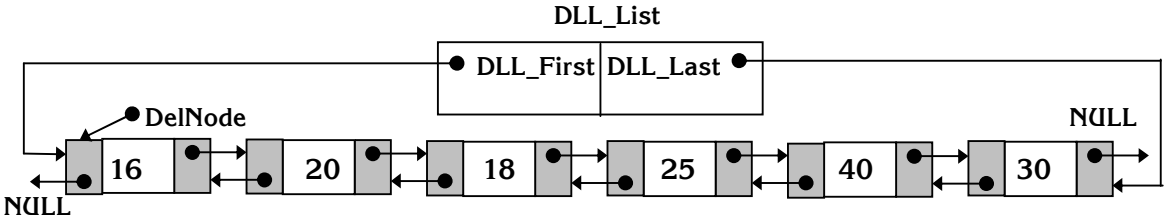
Hàm thực hiện việc xóa phần tử có thành phần dữ liệu là DelData trong danh sách liên kết đôi quản lý bởi hai con trỏ đầu và cuối ghi nhận trong DList. Hàm trả về giá trị 1 nếu việc xóa thành công và ngược lại, hàm trả về giá trị -1. Nội dung của hàm như sau:

```
int  DLL_Delete_Node (DLLP_Type  &DList, T  DelData)
  {  DLL_Type  DelNode = DLL_Searching(DList, DelData);
     if (DelNode == NULL)
        return (-1);
```

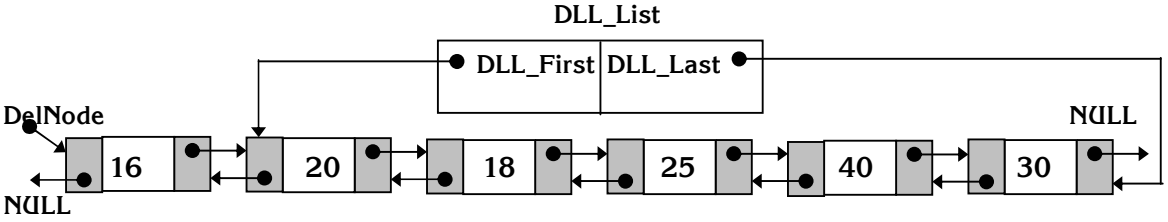
```
        if (DelNode->NextNode == NULL && DelNode->PreNode == NULL)
            DList.DLL_First = DList.DLL_Last = NULL;
        else
            if (DelNode->PreNode == NULL)
                { DList.DLL_First = DList.DLL_First->NextNode;
                  DList.DLL_First->PreNode = NULL;
                }
            else
                if (DelNode->NextNode == NULL)
                    { DList.DLL_Last = DList.DLL_Last->PreNode;
                      DList.DLL_Last->NextNode = NULL;
                    }
                else
                    { DelNode->PreNode->NextNode = DelNode->NextNode;
                      DelNode->NextNode->PreNode = DelNode->PreNode;
                    }
        DelNode->NextNode = DelNode->PreNode = NULL;
        delete DelNode;
        return (1);
    }
```

- Minh họa thuật toán:

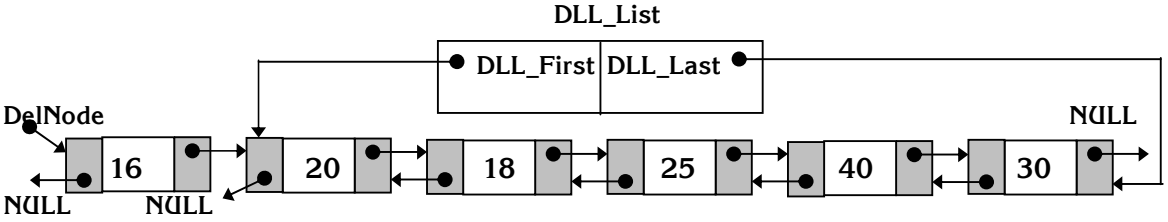
+ Hủy nút đầu: DelData = 16



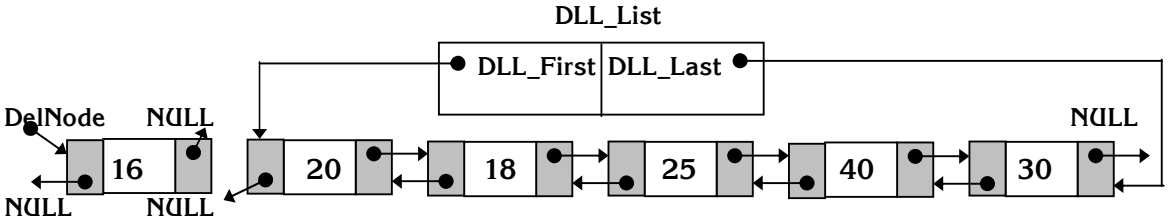
DLL_List.DLL_First = DLL_List.DLL_First->NextNode



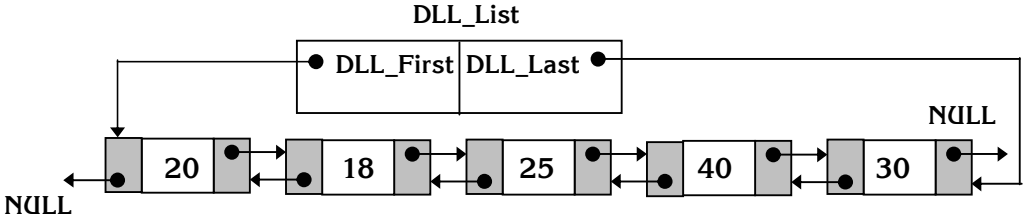
DLL_List.DLL_First->PreNode = NULL



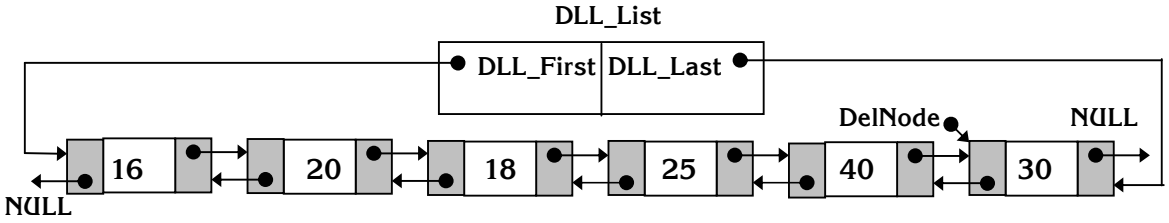
DelNode->NextNode = DelNode->PreNode = NULL;



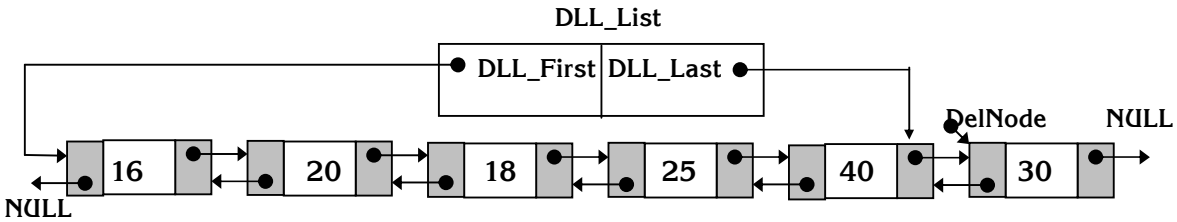
Kết quả sau khi hủy:



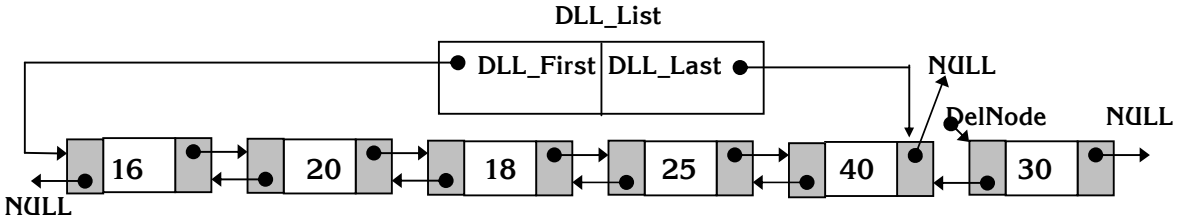
+ Hủy nút cuối: DelData = 30



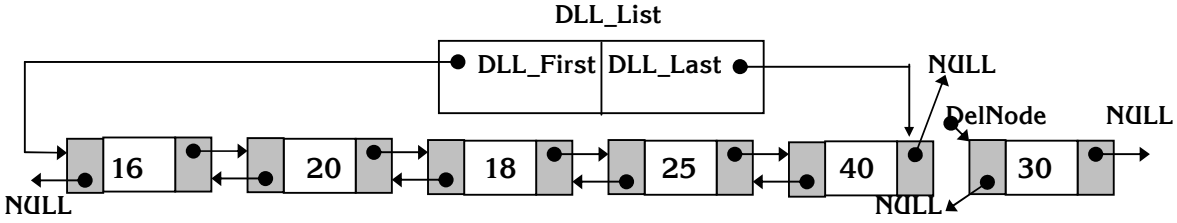
DLL_List.DLL_Last = DLL_List.DLL_Last->PreNode



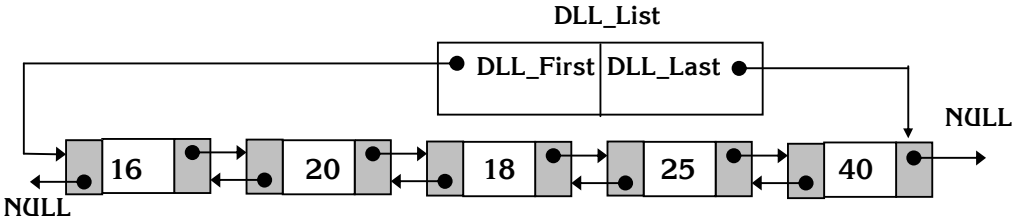
DLL_List.DLL_Last->NextNode = NULL



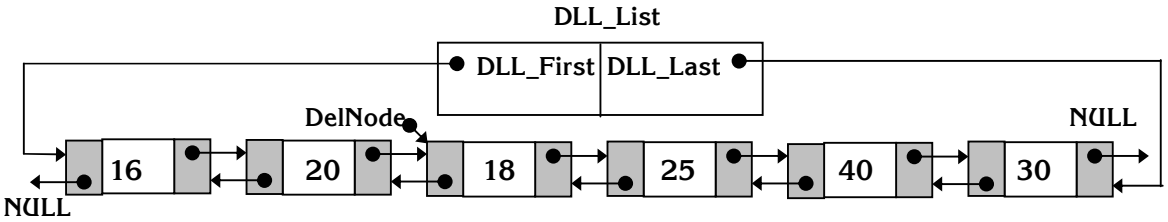
DelNode->NextNode = DelNode->PreNode = NULL



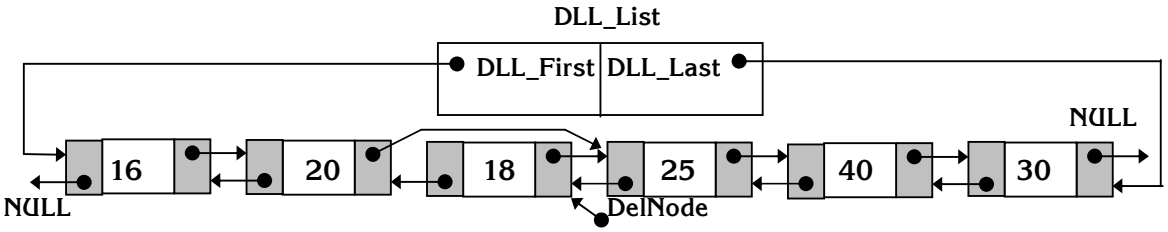
Kết quả sau khi hủy:



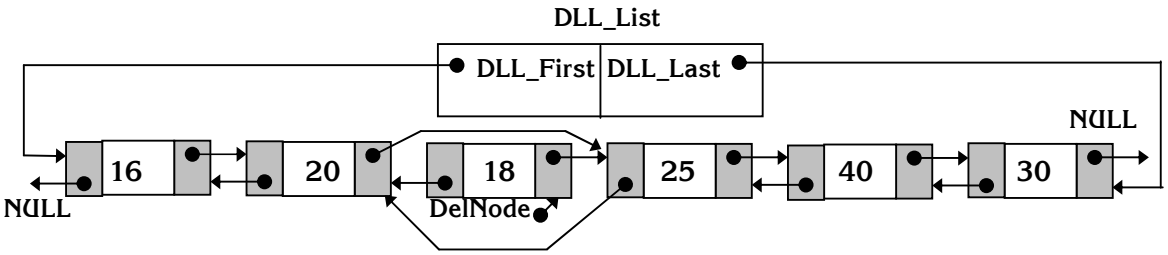
+ Hủy nút giữa: Giả sử chúng ta cần hủy nút có thành phần dữ liệu là 18 (DelData = 18)



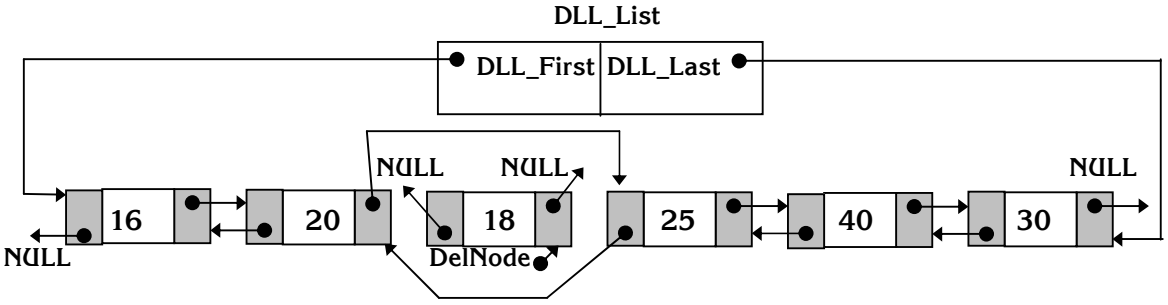
DelNode->PreNode->NextNode = DelNode->NextNode



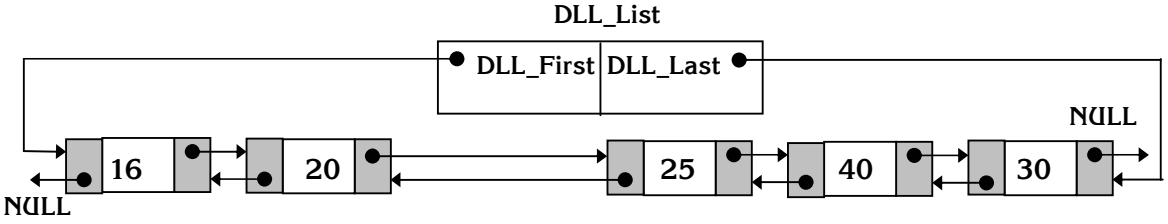
DelNode->NextNode->PreNode = DelNode->PreNode



DelNode->NextNode = DelNode->PreNode = NULL



Kết quả sau khi hủy:



g. Hủy toàn bộ danh sách:

Ở đây, chúng ta thực hiện nhiều lần thao tác hủy một nút.

- Thuật toán:

B1: IF (DLL_List.DLL_First = NULL)
    Thực hiện Bkt
B2: TempNode = DLL_List.DLL_First
B3: DLL_List.DLL_First = DLL_List.DLL_First->NextNode
B4: IF (DLL_List.DLL_First = NULL)
    B4.1: DLL_List.DLL_Last = NULL
    B4.2: Thực hiện B7
B5: DLL_List.DLL_First->PreNode = NULL
B6: TempNode->NextNode = NULL
B7: delete TempNode
B8: Lặp lại B1
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm DLL_Delete có prototype:

```
void  DLL_Delete (DLLP_Type  &DList);
```

Hàm thực hiện việc hủy toàn bộ danh sách liên kết đôi DList.

Nội dung của hàm như sau:

```
void  DLL_Delete (DLLP_Type  &DList)
   {  DLL_Type  TempNode = DList.DLL_First;
      while (TempNode != NULL)
         {  DList.DLL_First = DList.DLL_First->NextNode;
            TempNode->NextNode = NULL;
            if (DList.DLL_First != NULL)
               DList.DLL_First->PreNode = NULL;
            delete  TempNode;
            TempNode = DList.DLL_First;
         }
      return ;
   }
```

☛ **Lưu ý:**

Chúng ta cũng có thể vận dụng hàm DLL_Delete_Node để thực hiện thao tác này, lúc đó hàm DLL_Delete có thể viết lại như sau:

```
void  DLL_Delete (DLLP_Type  &DList)
   {  DLL_Type  TempNode = DList.DLL_First;
      while (TempNode != NULL)
         {  DLL_Delete_Node(DList, TempNode->Key);
            TempNode = DList.DLL_First;
         }
      return ;
   }
```

h. Tạo mới danh sách/ Nhập danh sách:

Cũng tương tự như trong danh sách liên kết đơn trong thao tác này, chúng ta liên tục thực hiện thao tác thêm một phần tử vào danh sách mà ban đầu danh sách này là một danh sách rỗng (Gồm hai con trỏ NULL). Chúng ta cũng có thể sử dụng một trong ba hàm thêm phần tử để thêm phần tử, ở đây sử dụng hàm SLL_Add_Last.

Giả sử chúng ta cần tạo danh sách liên kết đôi có N phần tử.

- Thuật toán:

B1: DLL_Initialize(DLL_List)
B2: i = 1
B3: IF (i > N)
    Thực hiện Bkt
B4: NewData = InputNewData() // Nhập giá trị cho biến NewData
B5: DLL_Add_Last(DLL_List, NewData)
B6: i++
B7: Lặp lại B3
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm DLL_Create có prototype:

```
DLLP_Type   DLL_Create (DLLP_Type  &DList, int N);
```

Hàm tạo danh sách liên kết đôi có N nút quản lý bởi hai địa chỉ nút đầu tiên và nút cuối cùng thông qua DList. Hàm trả về giá trị ghi nhận hai địa chỉ của nút đầu tiên và nút cuối cùng trong danh sách nếu việc tạo thành công, ngược lại hàm trả về danh sách rỗng (cả hai địa chỉ đều là giá trị NULL).

Nội dung của hàm như sau:

```
DLLP_Type   DLL_Create(DLLP_Type   &DList, int N)
  {  DLL_Initialize(DList);
     T  NewData;
     for (int i = 0; i < N; i++)
        {  NewData = InputNewData();
           if (DLL_Add_Last(DList, NewData) == NULL)
              {  DLL_Delete(DList);
                 break;
              }
        }
     return (DList);
  }
```

☞ **Lưu ý:**

Hàm InputNewData thực hiện nhập vào nội dung của một biến có kiểu dữ liệu T và trả về giá trị mới nhập vào. Tùy vào từng trường hợp cụ thể mà chúng ta viết hàm InputNewData cho phù hợp.

i. Tách một danh sách thành nhiều danh sách:

Giả sử chúng ta cần thực hiện việc tách các nút trong danh sách liên kết đôi DLL_List thành hai danh sách liên kết đôi con DLL_List1 và DLL_List2 luân phiên theo các đường chạy tự nhiên và cần *giữ lại danh sách liên kết ban đầu.*

- Thuật toán:

B1: DLL_Initialize(DLL_List1)
B2: DLL_Initialize(DLL_List2)
B3: CurNode = DLL_List.DLL_First

// Cắt các nút từ 1 đường chạy tự nhiên về DLL_List1
B4: IF (CurNode = NULL)
    Thực hiện Bkt
B5: DLL_Add_Last(DLL_List1, CurNode->Key)
B6: CurNode = CurNode->NextNode
B7: IF (CurNode = NULL)
    Thực hiện Bkt
B8: IF (CurNode->PreNode->Key > CurNode->Key)
    Thực hiện B10
B9: Lặp lại B4

// Cắt các nút từ 1 đường chạy tự nhiên về DLL_List2
B10: IF (CurNode = NULL)
    Thực hiện Bkt
B11: DLL_Add_Last(DLL_List2, CurNode->Key)
B12: CurNode = CurNode->NextNode
B13: IF (CurNode = NULL)
    Thực hiện Bkt
B14: IF (CurNode->PreNode->Key > CurNode->Key)
    Thực hiện B4
B15: Lặp lại B10
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm DLL_Split có prototype:

void DLL_Split(DLLP_Type &DList, DLLP_Type &DList1, DLLP_Type &DList2);

Hàm thực hiện việc phân phối các đường chạy tự nhiên trong DList thành về hai danh sách mới DList1 và DList2 (Danh sách cũ DList vẫn được giữ nguyên).

Nội dung của hàm như sau:

```
void  DLL_Split(DLLP_Type  &DList, DLLP_Type &DList1, DLLP_Type &DList2)
   {  DLL_Initialize(DList1);
      DLL_Initialize(DList2);
      DLL_Type  CurNode = DList.DLL_First;
      while (CurNode != NULL)
         {  do  { if (DLL_Add_Last(DList1, CurNode->Key) == NULL)
                     {  DLL_Delete (DList1);
                        DLL_Delete (DList2);
```

```
                        break;
                    }
                CurNode = CurNode->NextNode;
                if (CurNode == NULL)
                    break;
                if (CurNode->Key < CurNode->PreNode->Key)
                    break;
            }
        while (1);
        if (CurNode == NULL)
            break;
        do  { if (DLL_Add_Last(DList2, CurNode->Key) == NULL)
                { DLL_Delete (DList1);
                  DLL_Delete (DList2);
                  break;
                }
            CurNode = CurNode->NextNode;
            if (CurNode == NULL)
                break;
            if (CurNode->Key < CurNode->PreNode->Key)
                break;
            }
        while (1);
    }
  return ;
}
```

j. Nhập nhiều danh sách thành một danh sách:

Chúng ta thực hiện thao tác này trong hai trường hợp:

+ Ghép nối đuôi các danh sách lại với nhau;

+ Trộn xen lẫn các phần tử trong các danh sách vào thành một danh sách theo một trật tự nhất định

và sau khi nhập xong vẫn *giữ lại các danh sách ban đầu*.

Giả sử chúng ta cần nhập hai danh sách DLL_List1 và DLL_List2 lại với nhau thành một danh sách DLL_List.

- Thuật toán ghép nối hai danh sách thành một danh sách mới:

B1: DLL_Initialize (DLL_List)

// Đưa DLL_List1 vào đầu DLL_List

B2: CurNode = DLL_List1.DLL_First
B3: IF (CurNode = NULL)
    Thực hiện B7
B4: IF (DLL_Add_Last(DLL_List, CurNode->Key) = NULL)
    B4.1: DLL_Delete (DLL_List)
    B4.2: Thực hiện Bkt
B5: CurNode = CurNode->NextNode

B6: Lặp lại B3

// Đưa DLL_List2 vào sau DLL_List

B7: CurNode = DLL_List2.DLL_First
B8: IF (CurNode = NULL)
    Thực hiện Bkt
B9: IF (DLL_Add_Last(DLL_List, CurNode->Key) = NULL)
    B4.1: DLL_Delete (DLL_List)
    B4.2: Thực hiện Bkt
B10: CurNode = CurNode->NextNode
B11: Lặp lại B8
Bkt: Kết thúc

- Thuật toán trộn 2 danh sách thành 1 danh sách mới theo các đường chạy tự nhiên:

B1: CurNode1 = DLL_List1.DLL_First
B2: CurNode2 = DLL_List2.DLL_First
B3: IF (CurNode1 = NULL OR CurNode2 = NULL)
    Thực hiện B6

B4: IF (CurNode1->Key ≤ CurNode2->Key)
    B4.1: If (DLL_Add_Last (DLL_List, CurNode1->Key) = NULL)
        B4.1.1: DLL_Delete(DLL_List)
        B4.1.2: Thực hiện Bkt
    B4.2: CurNode1 = CurNode1->NextNode
    B4.3: If (CurNode1 = NULL)
        Thực hiện B10
    B4.4: If (CurNode1->PreNode->Key > CurNode1->Key)
        B4.4.1: if (DLL_Add_Last (DLL_List, CurNode2->Key) = NULL)
            B4.4.1.1: DLL_Delete(DLL_List)
            B4.4.1.2: Thực hiện Bkt
        B4.4.2: CurNode2 = CurNode2->NextNode
        B4.4.3: if (CurNode2 = NULL)
            Thực hiện B6
        B4.4.4: if (CurNode2->PreNode->Key > CurNode2->Key)
            Thực hiện B3
        B4.4.5: Lặp lại B4.4.1
    B4.5: Lặp lại B4

B5: ELSE
    B5.1: If (DLL_Add_Last (DLL_List, CurNode2->Key) = NULL)
        B5.1.1: DLL_Delete(DLL_List)
        B5.1.2: Thực hiện Bkt
    B5.2: CurNode2 = CurNode2->NextNode
    B5.3: If (CurNode2 = NULL)
        Thực hiện B6
    B5.4: If (CurNode2->PreNode->Key > CurNode2->Key)
        B5.4.1: if (DLL_Add_Last (DLL_List, CurNode1->Key) = NULL)
            B5.4.1.1: DLL_Delete(DLL_List)
            B5.4.1.2: Thực hiện Bkt

B5.4.2: CurNode1 = CurNode1->NextNode
B5.4.3: if (CurNode1 = NULL)
Thực hiện B10
B5.4.4: if (CurNode1->PreNode->Key > CurNode1->Key)
Thực hiện B3
B5.4.5: Lặp lại B5.4.1
B5.5: Lặp lại B4

// Đưa phần còn lại trong DLL_List1 về DLL_List

B6: IF (CurNode1 = NULL)
Thực hiện Bkt
B7: IF (DLL_Add_Last(DLL_List, CurNode1->Key) = NULL)
B7.1: DLL_Delete (DLL_List)
B7.2: Thực hiện Bkt
B8: CurNode1 = CurNode1->NextNode
B9: Lặp lại B6

// Đưa phần còn lại trong DLL_List2 về DLL_List

B10: IF (CurNode2 = NULL)
Thực hiện Bkt
B11: IF (DLL_Add_Last(DLL_List, CurNode2->Key) = NULL)
B11.1: DLL_Delete (DLL_List)
B11.2: Thực hiện Bkt
B12: CurNode2 = CurNode2->NextNode
B13: Lặp lại B10
Bkt: Kết thúc

- Cài đặt:

Các hàm nhập danh sách có prototype:

```
DLLP_Type  DLL_Concat (DLLP_Type  &DList1, DLLP_Type  &DList2,
                                                    DLLP_Type &DList);
DLLP_Type  DLL_Merge (DLLP_Type  &DList1, DLLP_Type  &DList2,
                                                    DLLP_Type &DList);
```

Hàm thực hiện việc nhập các nút trong hai danh sách DList1, DList2 thành một danh sách theo hai trường hợp đã trình bày trong hai thuật toán trên đây. Hàm trả về giá trị của danh sách sau khi ghép.

Nội dung của các hàm như sau:

```
DLLP_Type  DLL_Concat (DLLP_Type  &DList1, DLLP_Type  &DList2,
                                                    DLLP_Type &DList)
  {  DLL_Initialize (DList);
     DLL_Type  CurNode = DList1.DLL_First;
     while (CurNode != NULL)
        {  if (DLL_Add_Last (DList, CurNode->Key) == NULL)
              {  DLL_Delete(DList);
                 return (DList);
              }
```

```
            CurNode = CurNode->NextNode;
         }
      CurNode = DList2.DLL_First;
      while (CurNode != NULL)
         {  if (DLL_Add_Last (DList, CurNode->Key) == NULL)
               {  DLL_Delete(DList);
                  return (DList);
               }
            CurNode = CurNode->NextNode;
         }
      return (DList);
   }
//================================================================

DLLP_Type  DLL_Merge (DLLP_Type  &DList1, DLLP_Type  &DList2,
                                                      DLLP_Type &DList)
   {  DLL_Type  CurNode1 = DList1.DLL_First;
      DLL_Type  CurNode2 = DList2.DLL_First;
      while (CurNode1 != NULL && CurNode2 != NULL)
         {  if (CurNode1->Key <= CurNode2->Key)
               {  if (DLL_Add_Last (DList, CurNode1->Key) == NULL)
                     {  DLL_Delete (DList);
                        return (DList);
                     }
                  CurNode1 = CurNode1->NextNode;
                  if (CurNode1 == NULL)
                     break;
                  if (CurNode1->PreNode->Key > CurNode1->Key)
                     do { if (DLL_Add_Last (DList, CurNode2->Key) == NULL)
                              {  DLL_Delete (DList);
                                 return (DList);
                              }
                          CurNode2 = CurNode2->NextNode;
                        }
                     while (CurNode2 != NULL &&
                             CurNode2->PreNode->Key <= CurNode2->Key);
               }
            else
               {  if (DLL_Add_Last (DList, CurNode2->Key) == NULL)
                     {  DLL_Delete (DList);
                        return (DList);
                     }
                  CurNode2 = CurNode2->NextNode;
                  if (CurNode2 == NULL)
                     break;
                  if (CurNode2->PreNode->Key > CurNode2->Key)
                     do { if (DLL_Add_Last (DList, CurNode1->Key) == NULL)
                              {  DLL_Delete (DList);
```

```
                                return (DList);
                            }
                        CurNode1 = CurNode1->NextNode;
                    }
                while (CurNode1 != NULL &&
                        CurNode1->PreNode->Key <= CurNode1->Key);
            }
        }
    while (CurNode1 != NULL)
        { if (DLL_Add_Last (DList, CurNode1->Key) == NULL)
            { DLL_Delete (DList);
              break;
            }
          CurNode1 = CurNode1->NextNode;
        }
    while (CurNode2 != NULL)
        { if (DLL_Add_Last (DList, CurNode2->Key) == NULL)
            { DLL_Delete (DList);
              break;
            }
          CurNode2 = CurNode2->NextNode;
        }
    return (DList);
}
```

k. Sắp xếp thứ tự thành phần dữ liệu các nút trong danh sách:

Thao tác này rất thuận tiện trong việc áp dụng thuật toán sắp xếp trộn để sắp xếp, sinh viên có thể tự thực hiện. Ở đây, chúng ta vận dụng thuật toán sắp xếp nổi bọt để sắp xếp dữ liệu.

- Thuật toán sắp xếp vận dụng thuật toán nổi bọt:

B1: Inode = DLL_List.DLL_First
B2: IF (Inode = NULL)
    Thực hiện Bkt
B3: IF (Inode = DLL_List.DLL_Last)
    Thực hiện Bkt
B4: Jnode = DLL_List.DLL_Last
B5: IF (Jnode = Inode)
    Thực hiện B7
B6: ELSE
    B6.1: If (Jnode->Key < Jnode->PreNode->Key)
        Swap (Jnode->Key, Jnode->PreNode->Key)
    B6.2: Jnode = Jnode->PreNode
    B6.3: Lặp lại B5
B7: Inode = Inode->NextNode
B8: Lặp lại B3
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm DLL_Bubble_Sort có prototype:

void DLL_Bubble_Sort (DLLP_Type &DList);

Hàm thực hiện việc sắp xếp thành phần dữ liệu của các nút trong danh sách liên kết đôi DList theo thứ tự tăng dựa trên thuật toán sắp xếp nổi bọt.

Nội dung của hàm như sau:

```
void  DLL_Bubble_Sort (DLLP_Type  &DList)
   {  DLL_Type  Inode = DList.DLL_First;
      if (Inode == NULL)
         return;
      while (Inode != DList.DLL_Last)
         {  DLL_Type  Jnode = DList.DLL_Last;
            while (Jnode != Inode)
               {  if (Jnode->Key < Jnode->PreNode->Key)
                        Swap (Jnode->Key, Jnode->PreNode->Key);
                  Jnode = Jnode->PreNode;
               }
            Inode = Inode->NextNode;
         }
      return ;
   }
```

l. Sao chép một danh sách thành một danh sách mới:

Thao tác này hoàn toàn tương tự như trong danh sách liên kết đơn.

- Thuật toán:

B1: DLL_Initialize(NewList)
B2: CurNode = DLL_List.DLL_First
B3: IF (CurNode = NULL)
    Thực hiện Bkt
B4: DLL_Add_Last(NewList, CurNode->Key)
B5: CurNode = CurNode->NextNode
B6: Lặp lại B3
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm DLL_Copy có prototype:

DLLP_Type DLL_Copy (DLLP_Type &DList, DLLP_Type &NewList);

Hàm thực hiện việc sao chép nội dung danh sách DList thành danh sách NewList có cùng nội dung thành phần dữ liệu theo thứ tự của các nút trên DList. Hàm trả về giá trị của danh sách mới nếu việc sao chép thành công, ngược lại hàm trả về giá trị khởi tạo của danh sách.

Nội dung của hàm như sau:

```
DLLP_Type  DLL_Copy (DLLP_Type  &DList, DLLP_Type  &NewList)
   {  DLL_Initialize(NewList);
      DLL_Type  CurNode = DList.DLL_First;
```

```
        while (CurNode != NULL)
           {  if (DLL_Add_Last (NewList, CurNode->Key) == NULL)
                 {  DLL_Detete (NewList);
                    break;
                 }
              CurNode = CurNode->NextNode;
           }
        return (NewList);
     }
```

4.4.4. Ưu nhược điểm của danh sách liên kết

Do các phần tử (nút) được lưu trữ không liên tiếp nhau trong bộ nhớ, do vậy danh sách liên kết có các ưu nhược điểm sau đây:

- Mật độ sử dụng bộ nhớ của danh sách liên kết không tối ưu tuyệt đối (<100%);
- Việc truy xuất và tìm kiếm các phần tử của danh sách liên kết mất nhiều thời gian bởi luôn luôn phải duyệt tuần tự qua các phần tử trong danh sách;
- Tận dụng được những không gian bộ nhớ nhỏ để lưu trữ từng nút, tuy nhiên bộ nhớ lưu trữ thông tin mỗi nút lại tốn nhiều hơn do còn phải lưu thêm thông tin về vùng liên kết. Như vậy nếu vùng dữ liệu của mỗi nút là lớn hơn thì tỷ lệ mức tiêu tốn bộ nhớ này là không đáng kể, ngược lại thì nó lại gây lãng phí bộ nhớ.
- Việc thêm, bớt các phần tử trong danh sách, tách nhập các danh sách khá dễ dàng do chúng ta chỉ cần thay đổi mối liên kết giữa các phần tử với nhau.

4.5. Danh sách hạn chế

Trong các thao tác trên danh sách không phải lúc nào cũng có thể thực hiện được tất cả mà nhiều khi các thao tác này bị hạn chế trong một số loại danh sách, đó là danh sách hạn chế.

Như vậy, danh sách hạn chế là danh sách mà các thao tác trên đó bị hạn chế trong một chừng mực nào đó tùy thuộc vào danh sách. Trong phần này chúng ta xem xét hai loại danh sách hạn chế chủ yếu đó là:

- Hàng đợi (Queue);
- Ngăn xếp (Stack).

4.5.1. Hàng đợi (Queue)

A. Khái niệm - Cấu trúc dữ liệu:

Hàng đợi là một danh sách mà trong đó thao tác thêm một phần tử vào trong danh sách được thực hiện ở một đầu này và thao tác lấy ra một phần tử từ trong danh sách lại được thực hiện ở đầu kia.

Như vậy, các phần tử được đưa vào trong hàng đợi trước sẽ được lấy ra trước, phần tử đưa vào trong hàng đợi sau sẽ được lấy ra sau. Do đó mà hàng đợi còn được gọi là danh sách vào trước ra trước (FIFO List) và cấu trúc dữ liệu này còn được gọi là cấu trúc FIFO (First In – First Out).

Có nhiều cách để biểu diễn và tổ chức các hàng đợi:

- Sử dụng danh sách đặc,

- Sử dụng danh sách liên kết,

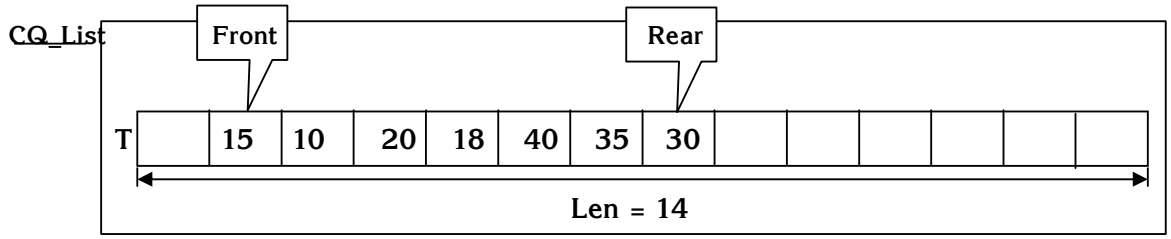
Tuy nhiên, điều quan trọng và cần thiết là chúng ta phải quản lý vị trí hai đầu của hàng đợi thông qua hai biến: Biến trước (Front) và Biến sau (Rear). Hai biến này có thể cùng chiều hoặc ngược chiều với thứ tự các phần tử trong mảng và trong danh sách liên kết. Điều này có nghĩa là đầu hàng đợi có thể là đầu mảng, đầu danh sách liên kết mà cũng có thể là cuối mảng, cuối danh sách liên kết. Để thuận tiện, ở đây chúng ta giả sử đầu hàng đợi cũng là đầu mảng, đầu danh sách liên kết. Trường hợp ngược lại, sinh viên tự áp dụng tương tự.

Ở đây chúng ta sẽ biểu diễn và tổ chức hàng đợi bằng danh sách đặc và bằng danh sách liên kết đơn được quản lý bởi hai con trỏ đầu và cuối danh sách. Do vậy cấu trúc dữ liệu của hàng đợi cũng như các thao tác trên hàng đợi sẽ được trình bày thành hai trường hợp khác nhau.

- Biểu diễn và tổ chức bằng danh sách đặc:

```
typedef  struct  Q_C
   {  int  Len;   // Chiều dài hàng đợi
      int  Front, Rear;
      T  *  List;// Nội dung hàng đợi
   }      C_QUEUE;

C_QUEUE  CQ_List;
```

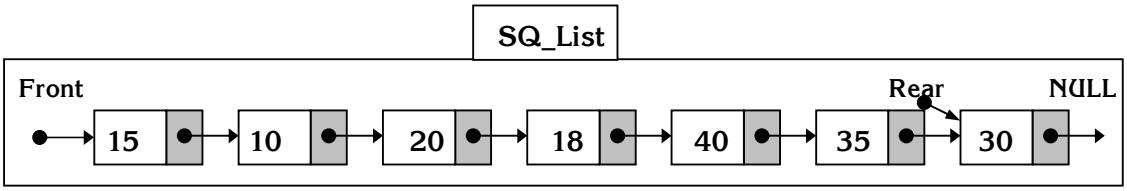
Hình ảnh minh họa:



- Biểu diễn và tổ chức bằng danh sách liên kết đơn;

```
typedef  struct  Q_Element
   {  T  Key;
      Q_Element * Next; // Vùng liên kết quản lý địa chỉ phần tử kế tiếp
   }      Q_OneElement;

typedef  Q_OneElement  *  Q_Type;

typedef  struct  QP_Element
   {  Q_Type  Front;
      Q_Type  Rear;
   }      S_QUEUE;

S_QUEUE  SQ_List;
```

Hình ảnh minh họa:



B. Các thao tác trên hàng đợi tổ chức bằng danh sách đặc:

Do hạn chế của danh sách đặc cho nên mỗi hàng đợi đều có một chiều dài cố định. Do vậy, trong quá trình thao tác trên hàng đợi có thể xảy ra hiện tượng hàng đợi bị đầy hoặc hàng đợi bị tràn.

- Khi hàng đợi bị đầy: số phần tử của hàng đợi bằng chiều dài cho phép của hàng đợi. Lúc này chúng ta không thể thêm bất kỳ một phần tử nào vào hàng đợi.
- Khi hàng đợi bị tràn: số phần tử của hàng đợi nhỏ hơn chiều dài cho phép của hàng đợi nhưng Rear = Len. Lúc này chúng ta phải khắc phục tình trạng tràn hàng đợi bằng cách dịch tất cả các phần tử của hàng đợi ra phía trước Front-1 vị trí hoặc xoay vòng để Rear chuyển lên vị trí đầu danh sách đặc. Trong phần này chúng ta sử dụng phương pháp xoay vòng. Như vậy theo phương pháp này, hàng đợi bị đầy trong các trường hợp sau:
  + Front = 1 và Rear = Len, khi: Front < Rear
  + Rear + 1 = Front, khi: Rear < Front

✍ **Ghi chú:**

Nếu chúng ta khắc phục hàng đợi bị tràn bằng phương pháp dịch tất cả các phần tử của hàng đợi ra phía trước Front-1 vị trí thì hàng đợi bị đầy khi thỏa mãn điều kiện: Front = 1 và Rear = Len (Ở đây ta luôn luôn có: Front ≤ Rear).

a. Khởi tạo hàng đợi (Initialize):

Trong thao tác này chúng ta thực hiện việc xác định kích thước hàng đợi, cấp phát bộ nhớ để lưu trữ phần dữ liệu cho hàng đợi, đồng thời cho giá trị các thành phần Front, Rear về giá trị 0 (trong C chúng ta khởi tạo về giá trị –1).

- Thuật toán:

B1: CQ_List.Len = Length
B2: CQ_List.List = new T[Length]
B3: IF (CQ_List.List = NULL)
    Thực hiện Bkt
B4: CQ_List.Front = CQ_List.Rear = 0
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm CQ_Initialize có prototype:

```
T * CQ_Initialize (C_QUEUE &QList, int Length);
```

Hàm thực hiện việc khởi tạo giá trị ban đầu cho hàng đợi quản lý bởi QList có kích thước Length. Hàm trả về con trỏ trỏ tới địa chỉ đầu khối dữ liệu của hàng đợi nếu việc khởi tạo thành công, ngược lại hàm trả về con trỏ NULL.

Nội dung của hàm như sau:

```
T  *  CQ_Initialize (C_QUEUE  &QList, int  Length)
   {  QList.Len = Length;
      QList.List = new  T[Length];
      if (QList.List == NULL)
         return (NULL);
      QList.Front = QList.Rear = -1;
      return (QList.List);
   }
```

b. Thêm (Đưa) một phần tử vào hàng đợi (Add):

Trong hàng đợi chúng ta luôn luôn đưa phần tử mới vào cuối hàng đợi, ngay sau vị trí Rear (nếu hàng đợi chưa bị đầy). Giả sử chúng ta cần đưa phần tử có giá trị NewData vào trong hàng đợi:

- Thuật toán:

```
// B1+B2: Nếu hàng đợi bị đầy
B1: IF (CQ_List.Front = 1 AND CQ_List.Rear = CQ_List.Len)
      Thực hiện Bkt
B2: IF (CQ_List.Rear+1 = CQ_List.Front)
      Thực hiện Bkt
B3: IF (CQ_List.Front = 0)    // Nếu hàng đợi rỗng
      CQ_List.Front = 1
B4: IF (CQ_List.Rear = CQ_List.Len)        //Nếu hàng bị tràn
      CQ_List.Rear = 1
B5: ELSE
      CQ_List.Rear++
B6: CQ_List.List[CQ_List.Rear] = NewData
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm CQ_Add có prototype:

```
int  CQ_Add (C_QUEUE  &QList, T  NewData);
```

Hàm thực hiện việc thêm phần tử có nội dung NewData vào trong hàng đợi quản lý bởi QList. Hàm trả về vị trí của phần tử vừa mới thêm nếu việc thêm thành công, ngược lại khi hàng đợi bị đầy hàm trả về giá trị -1.

Nội dung của hàm như sau:

```
int  CQ_Add (C_QUEUE  &QList, T  NewData)
   {  if (QList.Front == 0 && QList.Rear == QList.Len-1)
         return (-1);
      if (QList.Rear+1 == QList.Front)
         return (-1);
```

```
    if (QList.Front == -1)
       QList.Front = 0;
    if (QList.Rear == QList.Len)
       QList.Rear = 0;
    else
       QList.Rear += 1;
    QList.List[QList.Rear] = NewData;
    return (QList.Rear);
}
```

c. Lấy nội dung một phần tử trong hàng đợi ra để xử lý (Get):

Trong hàng đợi chúng ta luôn luôn lấy nội dung phần tử ở ngay đầu hàng đợi, tại vị trí Front (nếu hàng đợi không rỗng). Giả sử ta cần lấy dữ liệu ra biến Data:

- Thuật toán:

```
// Nếu hàng đợi bị rỗng
B1: IF (CQ_List.Front = 0)
      Thực hiện Bkt
B2: Data = CQ_List.List[CQ_List.Front]
B3: IF (CQ_List.Rear = CQ_List.Front)      // Hàng đợi chỉ có 1 phần tử
      B3.1: CQ_List.Rear = CQ_List.Front = 0
      B3.2: Thực hiện Bkt
B4: IF (CQ_List.Front = CQ_List.Len)
      CQ_List.Front = 1
B5: ELSE
      CQ_List.Front++
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm CQ_Get có prototype:

```
int  CQ_Get (C_QUEUE  &QList, T  &Data);
```

Hàm thực hiện việc lấy nội dung phần tử đầu hàng đợi quản lý bởi QList và ghi nhận vào Data nếu lấy được. Hàm trả về giá trị 1 nếu việc lấy thành công, ngược lại khi hàng đợi bị rỗng hàm trả về giá trị -1.

Nội dung của hàm như sau:

```
int  CQ_Get (C_QUEUE  &QList, T  &Data)
  {  if (QList.Front == -1)
        return (-1);
     Data = QList.List[QList.Front];
     if (QList.Front == QList.Rear)
        {  QList.Front = QList.Rear = -1;
           return (1);
        }
     if (QList.Front == QList.Len-1)
        QList.Front = 0;
     else
```

```
        QList.Front += 1;
      return (1);
   }
```

d. Hủy hàng đợi:

Trong thao tác này chúng ta thực hiện việc hủy bộ nhớ đã cấp phát cho hàng đợi. Hàm CQ_Delete có nội dung như sau:

```
void  CQ_Delete (C_QUEUE  &QList)
   {  delete QList.List;
      return;
   }
```

C. Các thao tác trên hàng đợi tổ chức bằng danh liên kết đơn:

Khác với hàng đợi biểu diễn bằng danh sách đặc, ở đây hàng đợi chỉ bị đầy khi hết bộ nhớ và không bao giờ bị tràn.

a. Khởi tạo hàng đợi (Initialize):

Tương tự như trong danh sách liên kết đơn, trong thao tác này chúng ta chỉ đơn giản thực hiện việc gán các con trỏ Front và Rear về con trỏ NULL. Hàm SQ_Initialize có nội dung như sau:

```
S_QUEUE  SQ_Initialize (S_QUEUE  &QList)
   {  QList.Front = QList.Rear = NULL;
      return (QList);
   }
```

b. Thêm (Đưa) một phần tử vào hàng đợi (Add):

Ở đây chúng ta thêm một phần tử vào sau Rear (Thêm vào cuối danh sách liên kết). Giả sử chúng ta cần đưa phần tử có giá trị dữ liệu là NewData vào trong hàng đợi:

- Thuật toán:

```
B1: NewElement = SLL_Create_Node(NewData)
B2: IF (NewElement = NULL)
       Thực hiện Bkt
B3: IF (SQ_List.Front = NULL)       // Nếu hàng đợi bị rỗng
       B3.1: SQ_List.Front = SQ_List.Rear = NewElement
       B3.2: Thực hiện Bkt
B4: SQ_List.Rear->Next = NewElement
B5: SQ_List.Rear = NewElement
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm SQ_Add có prototype:

```
Q_Type  SQ_Add (S_QUEUE  &QList, T  NewData);
```

Hàm thực hiện việc thêm phần tử có nội dung NewData vào trong hàng đợi quản lý bởi QList. Hàm trả về địa chỉ của phần tử vừa mới thêm nếu việc thêm thành công, ngược lại hàm trả về con trỏ NULL.

Nội dung của hàm như sau:

```
Q_Type  SQ_Add (S_QUEUE  &QList, T  NewData)
   {  Q_Type  NewElement = SLL_Create_Node(NewData);
      if (NewElement == NULL)
         return (NULL);
      if (QList.Front == NULL)
         QList.Front = QList.Rear = NewElement;
      else
         {  QList.Rear->Next = NewElement;
            QList.Rear = NewElement;
         }
      return (NewElement);
   }
```

c. Lấy nội dung một phần tử trong hàng đợi ra để xử lý (Get):

Ở đây chúng ta lấy nội dung thành phần dữ liệu của phần tử ở địa chỉ Front ra biến Data và tiến hành hủy luôn phần tử này.

- Thuật toán:

```
// Nếu hàng đợi bị rỗng
B1: IF (SQ_List.Front = NULL)
       Thực hiện Bkt
B2: TempElement = SQ_List.Front
B3: SQ_List.Front = SQ_List.Front->Next
B4: TempElement->Next = NULL
B5: Data = TempElement->Key
B6: IF (SQ_List.Front = NULL)       // Hàng đợi chỉ có 1 phần tử
       SQ_List.Rear = NULL
B7: delete  TempElement
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm SQ_Get có prototype:

```
int  SQ_Get (S_QUEUE  &QList, T  &Data);
```

Hàm thực hiện việc lấy nội dung thành phần dữ liệu của phần tử đầu hàng đợi quản lý bởi QList và ghi nhận vào Data nếu lấy được. Hàm trả về giá trị 1 nếu việc lấy thành công, ngược lại khi hàng đợi bị rỗng hàm trả về giá trị -1.

Nội dung của hàm như sau:

```
int  SQ_Get (S_QUEUE  &QList, T  &Data)
   {  if (QList.Front == NULL)
         return (-1);
      Q_Type  TempElement = QList.Front;
      QList.Front = QList.Front->Next;
      TempElement->Next = NULL;
      Data = TempElement->Key;
      if (QList.Front == NULL)
```

```
        QList.Rear = NULL;
      delete  TempElement;
      return (1);
   }
```

d. Hủy hàng đợi:

Trong thao tác này chúng ta thực hiện việc hủy toàn bộ các phần tử trong hàng đợi. Hàm SQ_Delete có nội dung như sau:

```
void  SQ_Delete (S_QUEUE  &QList)
   {  QList.Rear = NULL;
      while (QList.Front != NULL)
         {  Q_Type  TempElement = QList.Front;
            QList.Front = QList.Front->Next;
            TempElement->Next = NULL;
            delete  TempElement;
         }
      return;
   }
```

4.5.2. Ngăn xếp (Stack)

A. Khái niệm - Cấu trúc dữ liệu:

Ngăn xếp là một danh sách mà trong đó thao tác thêm một phần tử vào trong danh và thao tác lấy ra một phần tử từ trong danh sách được thực hiện ở cùng một đầu.

Như vậy, các phần tử được đưa vào trong ngăn xếp sau cùng sẽ được lấy ra trước tiên, phần tử đưa vào trong hàng đợi trước tiên sẽ được lấy ra sau cùng. Do đó mà ngăn xếp còn được gọi là danh sách vào sau ra trước (LIFO List) và cấu trúc dữ liệu này còn được gọi là cấu trúc LIFO (Last In – First Out).

Tương tự như hàng đợi, có nhiều cách để biểu diễn và tổ chức các ngăn xếp:

- Sử dụng danh sách đặc,
- Sử dụng danh sách liên kết,

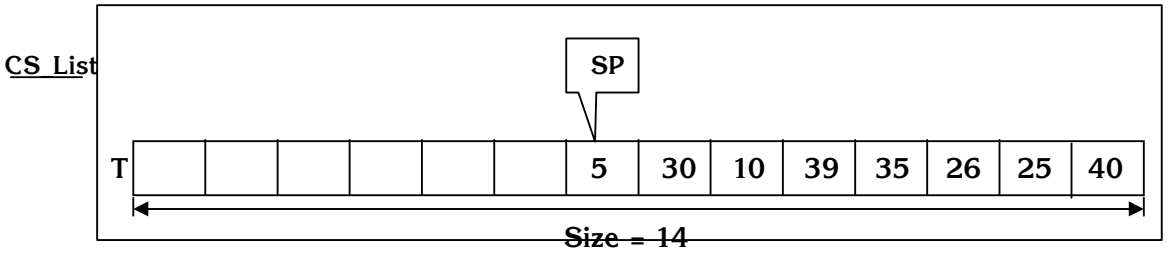
Do ở đây cả hai thao tác thêm vào và lấy ra đều được thực hiện ở một đầu nên chúng ta chỉ cần quản lý vị trí đầu của danh sách dùng làm mặt cho ngăn xếp thông qua biến chỉ số bề mặt SP (Stack Pointer). Chỉ số này có thể là cùng chiều (đầu) hoặc ngược chiều (cuối) với thứ tự các phần tử trong mảng và trong danh sách liên kết. Điều này có nghĩa là bề mặt ngăn xếp có thể là đầu mảng, đầu danh sách liên kết mà cũng có thể là cuối mảng, cuối danh sách liên kết. Để thuận tiện, ở đây chúng ta giả sử bề mặt của ngăn xếp là đầu mảng, đầu danh sách liên kết. Trường hợp ngược lại, sinh viên tự áp dụng tương tự.

Ở đây chúng ta cũng sẽ biểu diễn và tổ chức hàng đợi bằng danh sách đặc và bằng danh sách liên kết đơn được quản lý bởi con trỏ đầu danh sách. Do vậy cấu trúc dữ liệu của ngăn xếp và các thao tác trên đó sẽ được trình bày thành hai trường hợp khác nhau.

- Biểu diễn và tổ chức bằng danh sách đặc:

```
typedef  struct  S_C
   {  int  Size;  // Kích thước ngăn xếp
      int  SP;
      T  *  List;// Nội dung ngăn xếp
   }      C_STACK;

C_STACK  CS_List;
```

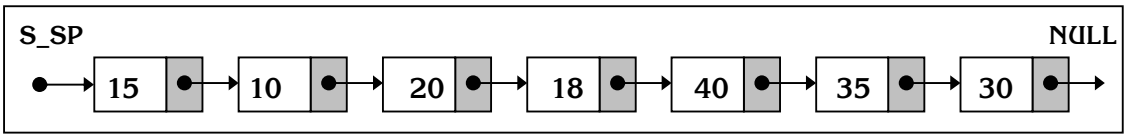
Hình ảnh minh họa:



- Biểu diễn và tổ chức bằng danh sách liên kết đơn;

```
typedef  struct  S_Element
   {  T  Key;
      S_Element * Next; // Vùng liên kết quản lý địa chỉ phần tử kế tiếp
   }      S_OneElement;

typedef  S_OneElement  *  S_STACK;

S_STACK  S_SP;
```

Hình ảnh minh họa:



B. Các thao tác trên ngăn xếp tổ chức bằng danh sách đặc:

Do hạn chế của danh sách đặc cho nên mỗi ngăn xếp sẽ có một kích thước cố định. Do vậy, trong quá trình thao tác trên ngăn xếp có thể xảy ra hiện tượng ngăn xếp bị đầy. Ngăn xếp bị đầy khi số phần tử của ngăn xếp bằng kích thước cho phép của ngăn xếp (SP = 1). Lúc này chúng ta không thể thêm bất kỳ một phần tử nào vào trong ngăn xếp.

a. Khởi tạo ngăn xếp (Initialize):

Trong thao tác này chúng ta thực hiện việc xác định kích thước ngăn xếp, cấp phát bộ nhớ để lưu trữ phần dữ liệu cho ngăn xếp và cho giá trị thành phần SP về giá trị Size+1.

- Thuật toán:

B1: CS_List.Size = MaxSize
B2: CS_List.List = new T[MaxSize]

B3: IF (CS_List.List = NULL)
    Thực hiện Bkt
B4: CS_List.SP = CS_List.Size + 1
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm CS_Initialize có prototype:

```
T  *  CS_Initialize (C_STACK  &SList, int  MaxSize);
```

Hàm thực hiện việc khởi tạo giá trị ban đầu cho ngăn xếp quản lý bởi SList có kích thước MaxSize. Hàm trả về con trỏ trỏ tới địa chỉ đầu khối dữ liệu của ngăn xếp nếu việc khởi tạo thành công, ngược lại hàm trả về con trỏ NULL.

Nội dung của hàm như sau:

```
T  *  CS_Initialize (C_STACK  &SList, int  MaxSize)
   {  SList.Size = MaxSize;
      SList.List = new  T[MaxSize];
      if (SList.List == NULL)
         return (NULL);
      SList.SP = SList.Size;
      return (SList.List);
   }
```

b. Thêm (Đẩy) một phần tử vào ngăn xếp (Push):

Trong ngăn xếp chúng ta luôn luôn đưa phần tử mới vào trên cùng của ngăn xếp, ngay trước vị trí SP (nếu ngăn xếp chưa bị đầy). Giả sử chúng ta cần đưa phần tử có giá trị NewData vào trong ngăn xếp:

- Thuật toán:

B1: IF (CS_List.SP = 1) // Nếu ngăn xếp bị đầy
    Thực hiện Bkt
B2: CS_List.SP--
B3: CS_List.List[CS_List.SP] = NewData
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm CS_Push có prototype:

```
int  CS_Push (C_STACK  &SList, T  NewData);
```

Hàm thực hiện việc đẩy thêm phần tử có nội dung NewData vào trong ngăn xếp quản lý bởi SList. Hàm trả về vị trí của phần tử vừa mới thêm nếu việc thêm thành công, ngược lại khi ngăn xếp bị đầy hàm trả về giá trị -1.

Nội dung của hàm như sau:

```
int  CS_Push (C_STACK  &SList, T  NewData)
   {  if (SList.SP == 0)
         return (-1);
      SList.SP -= 1;
      SList.List[SList.SP] = NewData;
```

```
    return (SList.SP);
}
```

c. Lấy nội dung một phần tử trong ngăn xếp ra để xử lý (Pop):

Ở đây chúng ta cũng luôn luôn lấy nội dung phần tử ở ngay bề mặt ngăn xếp, tại vị trí SP (nếu ngăn xếp không rỗng). Giả sử ta cần lấy dữ liệu ra biến Data:

- Thuật toán:

```
// Nếu ngăn xếp bị rỗng
B1: IF (CS_List.SP = CS_List.Size+1)
      Thực hiện Bkt
B2: Data = CS_List.List[CS_List.SP]
B3: CS_List.SP++
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm CS_Pop có prototype:

```
int  CS_Pop (C_STACK  &SList, T  &Data);
```

Hàm thực hiện việc lấy nội dung phần tử ở trên bề mặt ngăn xếp quản lý bởi SList và ghi nhận vào Data nếu lấy được. Hàm trả về giá trị 1 nếu việc lấy thành công, ngược lại khi ngăn xếp bị rỗng hàm trả về giá trị -1.

Nội dung của hàm như sau:

```
int  CS_Pop (C_STACK  &SList, T  &Data)
   {  if (SList.SP == SList.Size)
         return (-1);
      Data = SList.List[SList.SP];
      SList.SP += 1;
      return (1);
   }
```

d. Hủy ngăn xếp:

Trong thao tác này chúng ta thực hiện việc hủy bộ nhớ đã cấp phát cho ngăn xếp. Hàm CS_Delete có nội dung như sau:

```
void  CS_Delete (C_STACK  &SList)
   {  delete SList.List;
      return;
   }
```

C. Các thao tác trên ngăn xếp tổ chức bằng danh liên kết đơn:

a. Khởi tạo ngăn xếp:

Hàm SS_Initialize có nội dung như sau:

```
S_STACK  SS_Initialize (S_STACK  &SList)
   {  SList = NULL;
      return (SList);
   }
```

b. Thêm (Đẩy) một phần tử vào ngăn xếp (Push):

Ở đây chúng ta thêm một phần tử vào trước S_SP (Thêm vào đầu danh sách liên kết). Giả sử chúng ta cần đưa phần tử có giá trị dữ liệu là NewData vào trong ngăn xếp:

- Thuật toán:

```
B1: NewElement = SLL_Create_Node(NewData)
B2: IF (NewElement = NULL)
        Thực hiện Bkt
B3: IF (S_SP = NULL)          // Nếu ngăn xếp bị rỗng
        B3.1: S_SP = NewElement
        B3.2: Thực hiện Bkt
B4: NewElement->Next = S_SP
B5: S_SP = NewElement
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm SS_Push có prototype:

```
S_STACK  SS_Push (S_STACK  &SList, T  NewData);
```

Hàm thực hiện việc thêm phần tử có nội dung NewData vào trong ngăn xếp quản lý bởi SList. Hàm trả về địa chỉ của phần tử vừa mới thêm nếu việc thêm thành công, ngược lại hàm trả về con trỏ NULL.

Nội dung của hàm như sau:

```
S_STACK  SS_Push (S_STACK  &SList, T  NewData)
  {  S_STACK  NewElement = SLL_Create_Node(NewData);
     if (NewElement == NULL)
        return (NULL);
     NewElement->Next = SList;
     SList = NewElement;
     return (NewElement);
  }
```

c. Lấy nội dung một phần tử trong ngăn xếp ra để xử lý (Pop):

Ở đây chúng ta lấy nội dung thành phần dữ liệu của phần tử ở địa chỉ S_SP ra biến Data và tiến hành hủy luôn phần tử này.

- Thuật toán:

```
// Nếu ngăn xếp bị rỗng
B1: IF (S_SP = NULL)
        Thực hiện Bkt
B2: TempElement = S_SP
B3: S_SP = S_SP->Next
B4: TempElement->Next = NULL
B5: Data = TempElement->Key
B6: delete  TempElement
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm SS_Pop có prototype:

int SS_Pop (S_STACK &SList, T &Data);

Hàm thực hiện việc lấy nội dung thành phần dữ liệu của phần tử ở bề mặt ngăn xếp quản lý bởi SList và ghi nhận vào Data nếu lấy được. Hàm trả về giá trị 1 nếu việc lấy thành công, ngược lại khi ngăn xếp bị rỗng hàm trả về giá trị -1.

Nội dung của hàm như sau:

```
int SS_Pop (S_STACK &SList, T &Data)
  { if (SList == NULL)
       return (-1);
    S_STACK TempElement = SList;
    SList = SList->Next;
    TempElement->Next = NULL;
    Data = TempElement->Key;
    delete TempElement;
    return (1);
  }
```

d. Hủy ngăn xếp:

Trong thao tác này chúng ta thực hiện việc hủy toàn bộ các phần tử trong ngăn xếp. Hàm SS_Delete có nội dung như sau:

```
void SS_Delete (S_STACK &SList)
  { while (SList != NULL)
      { S_STACK TempElement = SList;
        SList = SList->Next;
        TempElement->Next = NULL;
        delete TempElement;
      }
    return;
  }
```

4.5.3. Ứng dụng của danh sách hạn chế

Danh sách hạn chế được sử dụng trong nhiều trường hợp, ví dụ:

- Hàng đợi thường được sử dụng để lưu trữ các luồng dữ liệu cần xử lý tuần tự;
- Ngăn xếp thường được xử lý trong các luồng dữ liệu truy hồi, đặc biệt là trong việc khử đệ quy cho các thuật toán.

Câu hỏi và Bài tập

1. Trình bày khái niệm của các loại danh sách? Ưu, nhược điểm và ứng dụng của mỗi loại danh sách?
2. Hãy đưa ra các cấu trúc dữ liệu để quản lý các loại danh sách vừa kể trên? Mỗi loại bạn hãy chọn ra một cấu trúc dữ liệu mà theo bạn là hay nhất? Giải thích sự lựa chọn đó?

3. Trình bày thuật toán và cài đặt tất cả các thao tác trên danh sách liên kết đơn trong trường hợp quản lý bằng con trỏ đầu và cuối trong danh sách?

4. Trình bày thuật toán và cài đặt tất cả các thao tác trên danh sách liên kết đôi trong trường hợp chỉ quản lý bằng con trỏ đầu trong danh sách?

5. Trình bày thuật toán và cài đặt tất cả các thao tác trên hàng đợi, ngăn xếp biểu diễn bởi danh sách liên kết đôi trong hai trường hợp: Danh sách liên kết cùng chiều và ngược chiều với hàng đợi, ngăn xếp?

6. Vận dụng các thuật toán sắp xếp đã học, hãy cài đặt các hàm sắp xếp trên danh sách liên kết đơn, liên kết đôi theo hai cách quản lý:

- Quản lý địa chỉ nút đầu danh sách;
- Quản lý địa chỉ nút đầu và cuối danh sách.

Theo bạn thuật toán sắp xếp nào dễ vận dụng hơn trên danh sách liên kết đơn, liên kết đôi trong hai trường hợp này?

7. Hãy trình bày thuật toán và cài đặt thao tác tách một danh sách liên kết (đơn/đôi) có thành phần dữ liệu là các số nguyên thành hai danh sách liên kết có thành phần dữ liệu tương ứng là các số chẵn và các số lẻ, sao cho tối ưu bộ nhớ máy tính nếu như danh sách ban đầu sau khi tách không còn cần thiết?

8. Hãy trình bày thuật toán và cài đặt thao tác trộn các danh sách liên kết (đơn/đôi) có thứ tự thành một danh sách liên kết có thứ tự sao cho tối ưu bộ nhớ máy tính nếu như các danh sách sau khi trộn không còn cần thiết?

9. Vận dụng danh sách liên kết đôi, trình bày thuật toán và cài đặt các thao tác tạo mới, thêm, bớt các mục trong một menu thanh ngang, menu dọc?

10. Sử dụng Stack, viết chương trình nhập vào một số nguyên, không âm bất kỳ, sau đó xuất ra màn hình số đảo ngược thứ tự các chữ số của số nhập vào.

Ví dụ: - Nhập vào một số nguyên: 10245

- Số nguyên ở dạng đảo ngược: 54201

11. Sử dụng Stack, viết chương trình chuyển đổi một số nguyên N trong hệ thập phân *(hệ 10)* sang biểu diễn ở:

a. Hệ nhị phân *(hệ 2)*

b. Hệ thập lục phân *(hệ 16)*

12. Viết chương trình mô phỏng cho bài toán "Tháp Hà nội" và "Tháp Saigon" với các cấu trúc dữ liệu như sau:

a. Sử dụng danh sách liên kết để lưu trữ các cột tháp;

b. Sử dụng Stack để lưu trữ các cột của tháp

Có nhận xét gì cho từng trường hợp?

13. Vận dụng Stack để gỡ đệ quy cho thuật toán QuickSort?

14. Vận dụng danh sách liên kết vòng để giải bài toán Josephus.

Chương 5: CÂY (TREE)

5.1. Khái niệm – Biểu diễn cây

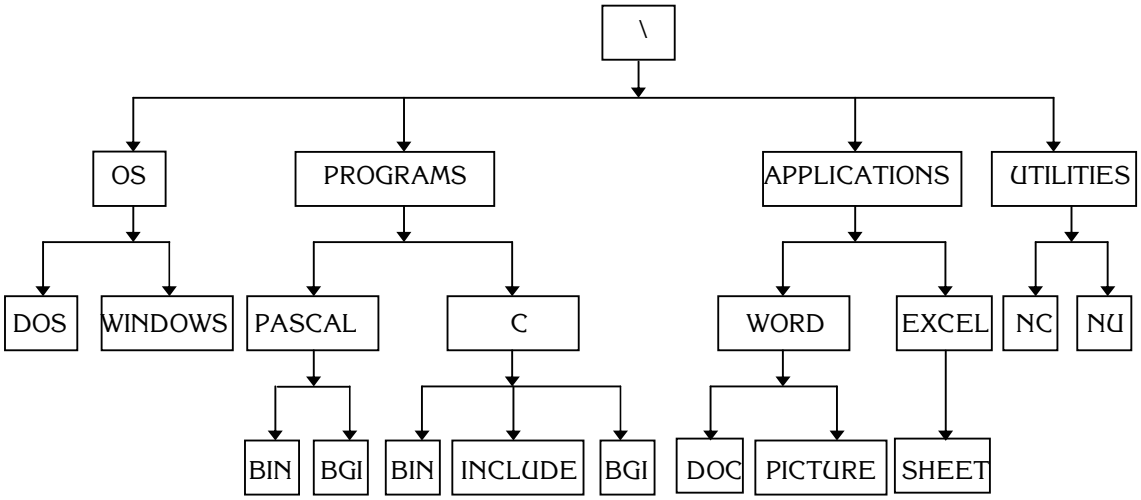
5.1.1. Định nghĩa cây

Cây là một tập hợp các phần tử (các nút) được tổ chức và có các đặc điểm sau:

- Hoặc là một tập hợp rỗng (cây rỗng)
- Hoặc là một tập hợp khác rỗng trong đó có một nút duy nhất được làm nút gốc (Root's Node), các nút còn lại được phân thành các nhóm trong đó mỗi nhóm lại là một cây gọi là cây con (Sub-Tree).

Như vậy, một cây con có thể là một tập rỗng các nút và cũng có thể là một tập hợp khác rỗng trong đó có một nút làm nút gốc cây con.

Ví dụ: Cây thư mục trên một đĩa cứng



5.1.2. Một số khái niệm liên quan

a. Bậc của một nút:

Bậc của một nút (node's degree) là số cây con của nút đó

Ví dụ: Bậc của nút OS trong cây trên bằng 2

b. Bậc của một cây:

Bậc của một cây (tree's degree) là bậc lớn nhất của các nút trong cây.

Cây có bậc N gọi là cây N-phân (N-Tree)

Ví dụ: Bậc của cây trên bằng 4 (bằng bậc của nút gốc) và cây trên gọi là cây tứ phân (Quartz-Tree)

c. Nút gốc:

Nút gốc (root's node) là nút không phải là nút gốc cây con của bất kỳ một cây con nào khác trong cây (nút không làm nút gốc cây con).

Ví dụ: Nút \ của cây trên là các nút gốc.

d. Nút kết thúc:

Nút kết thúc hay còn gọi là nút lá (leaf's node) là nút có bậc bằng 0 (nút không có nút cây con).

Ví dụ: Các nút DOS, WINDOWS, BIN, INCLUDE, BGI, DOC, PICTURE, SHEET, NC, NU của cây trên là các nút lá.

e. Nút trung gian:

Nút trung gian hay còn gọi là nút giữa (interior's node) là nút không phải là nút gốc và cũng không phải là nút kết thúc (nút có bậc khác không và là nút gốc cây con của một cây con nào đó trong cây).

Ví dụ: Các nút OS, PROGRAMS, APPLICATIONS, UTILITIES, PASCAL, C, WORD, EXCEL của cây trên là các nút trung gian.

f. Mức của một nút:

Mức của một nút (node's level) bằng mức của nút gốc cây con chứa nó cộng thêm 1, trong đó mức của nút gốc bằng 1.

Ví dụ: Mức của các nút DOS, WINDOWS, PASCAL, C, WORD, EXCEL, NC, NU của cây trên bằng 3; mức của các nút BIN, INCLUDE, BGI, DOC, PICTURE, SHEET, của cây trên bằng 4.

g. Chiều cao hay chiều sâu của một cây:

Chiều cao của một cây (tree's height) hay chiều sâu của một cây (tree's depth) là mức cao nhất của các nút trong cây.

Ví dụ: Chiều cao của cây trên bằng 4.

h. Nút trước và nút sau của một nút:

Nút T được gọi là nút trước (ancestor's node) của nút S nếu cây con có gốc là T chứa cây con có gốc là S. Khi đó, nút S được gọi là nút sau (descendant's node) của nút T.

Ví dụ: Nút PROGRAMS là nút trước của các nút BIN, BGI, INCLUDE, PASCAL, C và ngược lại các nút BIN, BGI, INCLUDE, PASCAL, C là nút sau của nút PROGRAMS trong cây trên.

i. Nút cha và nút con của một nút:

Nút B được gọi là nút cha (parent's node) của nút C nếu nút B là nút trước của nút C và mức của nút C lớn hơn mức của nút B là 1 mức. Khi đó, nút C được gọi là nút con (child's node) của nút B.

Ví dụ: Nút PROGRAMS là nút cha của các nút PASCAL, C và ngược lại các nút PASCAL, C là nút con của nút PROGRAMS trong cây trên.

j. Chiều dài đường đi của một nút:

Chiều dài đường đi của một nút là số đỉnh (số nút) tính từ nút gốc để đi đến nút đó.

Như vậy, chiều dài đường đi của nút gốc luôn luôn bằng 1, chiều dài đường đi tới một nút bằng chiều dài đường đi tới nút cha nó cộng thêm 1.

Ví dụ: Chiều dài đường đi tới nút PROGRAMS trong cây trên là 2.

k. Chiều dài đường đi của một cây:

Chiều dài đường đi của một cây (path's length of the tree) là tổng tất cả các chiều dài đường đi của tất cả các nút trên cây.

Ví dụ: Chiều dài đường của cây trên là 65.

Ghi chú: Đây là chiều dài đường đi trong (internal path's length) của cây. Để có được chiều dài đường đi ngoài (external path's length) của cây người ta mở rộng tất cả các nút của cây sao cho tất cả các nút của cây có cùng bậc bằng cách thêm vào các nút giả sao cho tất cả các nút có bậc bằng bậc của cây. Chiều dài đường đi ngoài của cây bằng tổng chiều dài của tất cả các nút mở rộng.

l. Rừng:

Rừng (forest) là tập hợp các cây.

Như vậy, một cây khi mất nút gốc sẽ trở thành một rừng.

5.1.3. Biểu diễn cây

Có nhiều cách để biểu diễn cây:

- Sử dụng đồ thị: Như ví dụ về cây thư mục ở trên.
- Sử dụng giản đồ tập hợp
- Sử dụng dạng phân cấp chỉ số: Như bảng mục lục trong các tài liệu, giáo trình, ...
- ...

Biểu diễn cây trong bộ nhớ máy tính:

Để biểu diễn cây trong bộ nhớ máy tính chúng ta có thể sử dụng danh sách liên kết. Như vậy, để biểu diễn cây N-phân chúng ta sử dụng danh sách có N mối liên kết để quản lý địa chỉ N nút gốc cây con. Như vậy cấu trúc dữ liệu của cây N-phân tương tự như cấu trúc dữ liệu của danh sách đa liên kết:

```
const  int  N = 100;

typedef  struct  NT_Node
   {  T  Key;
      NT_Node * SubNode[N]; // Vùng liên kết quản lý địa chỉ N nút gốc cây con
   } NT_OneNode;

typedef  NT_OneNode  *  NT_Type;
```

Để quản lý các cây chúng ta chỉ cần quản lý địa chỉ nút gốc của cây:

```
NT_Type  NTree;
```

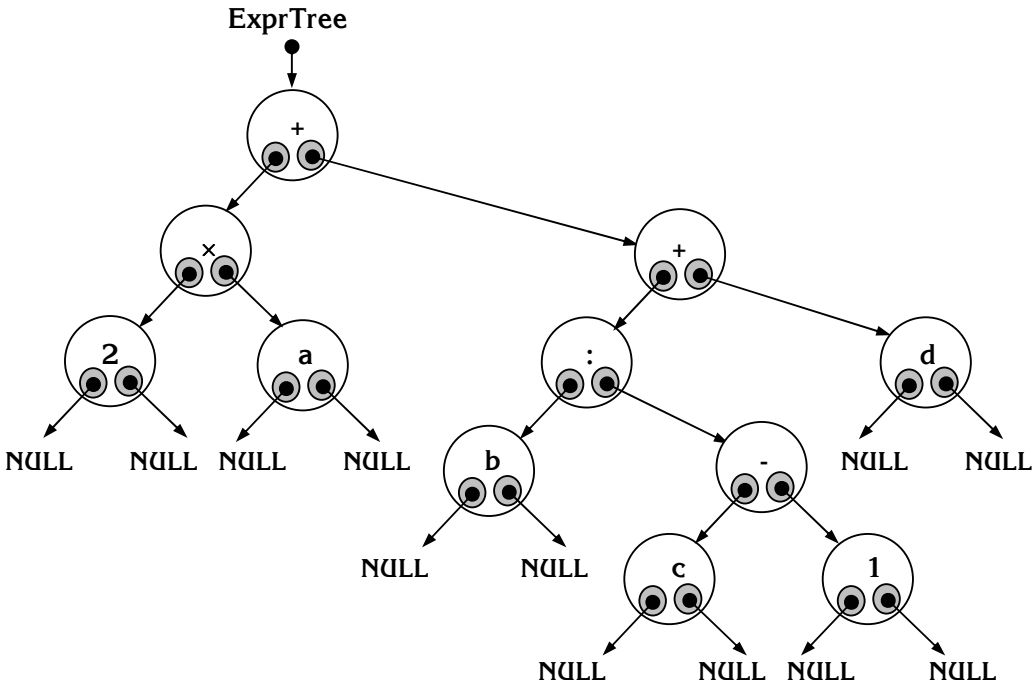
Trong phạm vi phần này chúng ta sẽ trình bày các thao tác trên cây nhị phân (Binary Tree) là cây phổ biến và thông dụng nhất.

5.2. Cây nhị phân (Binary Tree)

5.2.1. Định nghĩa

Cây nhị phân là cây có bậc bằng 2 (bậc của mỗi nút tối đa bằng 2).

Ví dụ: Cây nhị phân biểu diễn biểu thức (2 × a) + [b : (c – 1) + d] như sau:



5.2.2. Biểu diễn và Các thao tác

A. Biểu diễn cây nhị phân:

Để biểu diễn cây nhị phân trong bộ nhớ máy tính chúng ta có thể sử dụng danh sách có 2 mối liên kết để quản lý địa chỉ của 2 nút gốc cây con (cây con trái và cây con phải). Như vậy cấu trúc dữ liệu của cây nhị phân tương tự như cấu trúc dữ liệu của danh sách liên kết đôi nhưng về cách thức liên kết thì khác nhau:

```
typedef  struct  BinT_Node
  {  T  Key;
     BinT_Node * BinT_Left; // Vùng liên kết quản lý địa chỉ nút gốc cây con trái
     BinT_Node * BinT_Right; // Vùng liên kết quản lý địa chỉ nút gốc cây con phải
  }    BinT_OneNode;

typedef  BinT_OneNode  *  BinT_Type;
```

Để quản lý các cây nhị phân chúng ta cần quản lý địa chỉ nút gốc của cây:

```
BinT_Type  BinTree;
```

B. Các thao tác trên cây nhị phân:

a. Khởi tạo cây nhị phân:

Việc khởi tạo cây nhị phân chỉ đơn giản chúng ta cho con trỏ quản lý địa chỉ nút gốc về con trỏ NULL. Hàm khởi tạo cây nhị phân như sau:

```
BinT_Type  BinT_Initialize (BinT_Type  &BTree)
   {  BTree = NULL;
      return (BTree);
   }
```

b. Tạo mới một nút:

Thao tác này hoàn toàn tương tự như đối với thao tác tạo mới một nút trong danh sách liên kết đôi. Giả sử chúng ta cần tạo mới một nút có thành phần dữ liệu là NewData.

- Thuật toán:

```
B1: BTNode = new BinT_OneNode
B2: IF (BTNode = NULL)
       Thực hiện Bkt
B3: BTNode->BinT_Left = NULL
B4: BTNode->BinT_Right = NULL
B5: BTNode->Key = NewData
Bkt: Kết thúc
```

- Cài đặt thuật toán:

Hàm BinT_Create_Node có prototype:

```
BinT_Type  BinT_Create_Node(T  NewData);
```

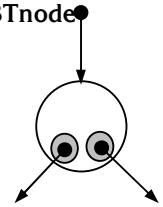
Hàm tạo mới một nút có thành phần dữ liệu là NewData, hàm trả về con trỏ trỏ tới địa chỉ của nút mới tạo. Nếu không đủ bộ nhớ để tạo, hàm trả về con trỏ NULL.

```
BinT_Type  BinT_Create_Node(T  NewData)
   {  BinT_Type  BTnode = new  BinT_OneNode;
      if (BTnode != NULL)
         {  BTnode->BinT_Left = NULL;
            BTnode->BinT_Right = NULL;
            BTnode->Key = NewData;
         }
      return (BTnode);
   }
```

- Minh họa thuật toán:

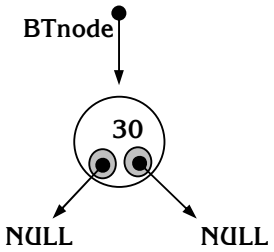
Giả sử chúng ta cần tạo nút có thành phần dữ liệu là 30: NewData = 30

BTnode = new BinT_OneNode




```
BTnode->BinT_Left = NULL
BTnode->BinT_Right = NULL
```

BTnode->Key = NewData



c. Thêm một nút vào trong cây nhị phân:

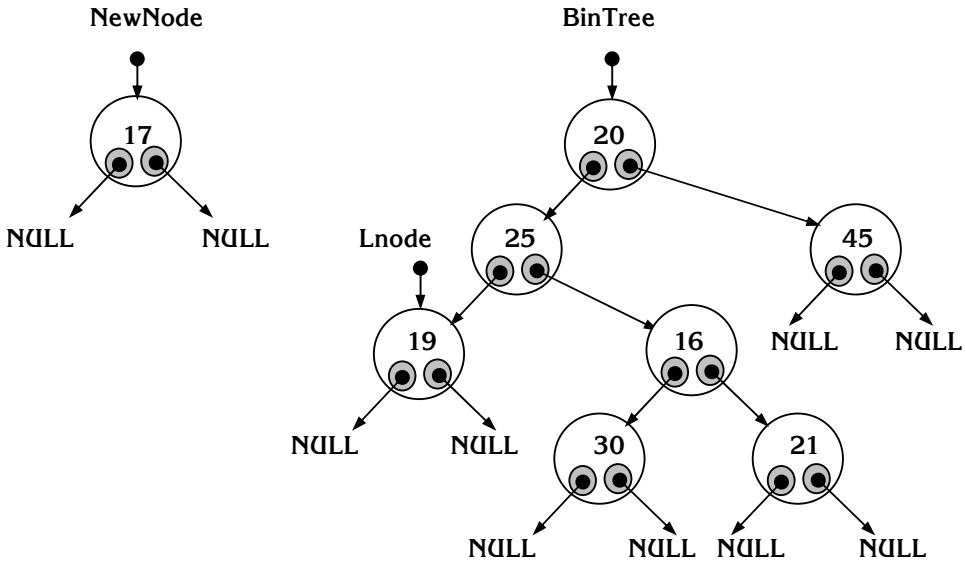
Giả sử chúng ta cần thêm một nút có giá trị thành phần dữ liệu là NewData vào trong cây nhị phân. Việc thêm có thể diễn ra ở cây con trái hoặc cây con phải của cây nhị phân. Do vậy, ở đây chúng ta trình bày 2 thao tác thêm riêng biệt nhau:

- Thuật toán thêm 1 nút vào bên trái nhất của cây:

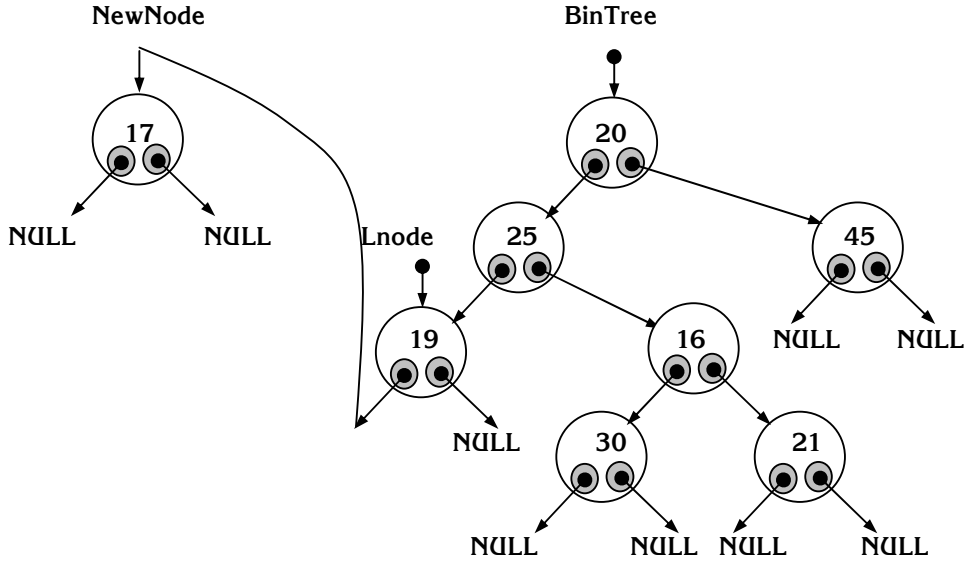
B1: NewNode = BinT_Create_Node (NewData)
B2: IF (NewNode = NULL)
    Thực hiện Bkt
B3: IF (BinTree = NULL) // Cây rỗng
    B3.1: BinTree = NewNode
    B3.2: Thực hiện Bkt
B4: Lnode = BinTree
B5: IF (Lnode->BinT_Left = NULL) // Cây con trái rỗng
    B5.1: Lnode->BinT_Left = NewNode
    B5.2: Thực hiện Bkt
B6: Lnode = Lnode->BinT_Left // Đi theo nhánh cây con trái
B7: Lặp lại B5
Bkt: Kết thúc

- Minh họa thuật toán:

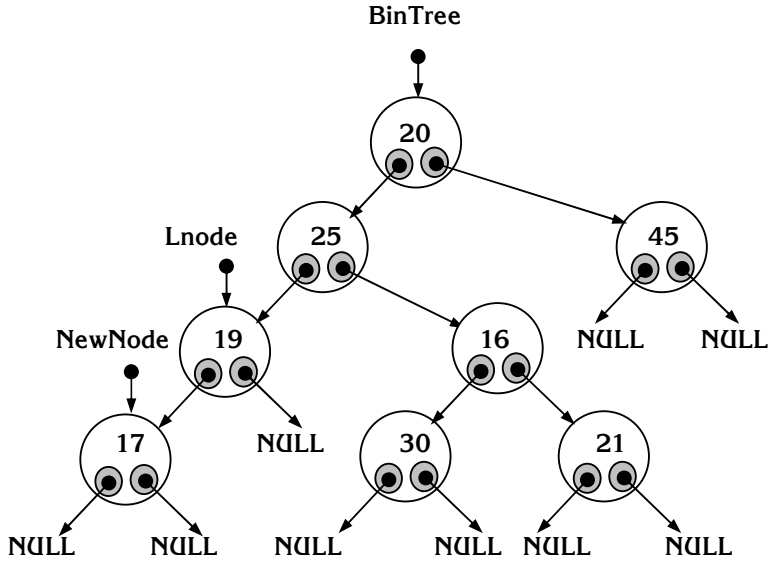
Giả sử chúng ta cần thêm nút có thành phần dữ liệu là 17 vào bên trái nhất của cây nhị phân: NewData = 17



B5.1: Lnode->BinT_Left = NewNode



Kết quả sau khi thêm:



- Cài đặt thuật toán:

Hàm BinT_Add_Left có prototype:

BinT_Type BinT_Add_Left(BinT_Type &BT_Tree, T NewData);

Hàm thực hiện việc thêm vào bên trái nhất trong cây nhị phân BT_Tree một nút có thành phần dữ liệu là NewData, hàm trả về con trỏ trỏ tới địa chỉ của nút mới thêm nếu việc thêm thành công, ngược lại nếu không đủ bộ nhớ, hàm trả về con trỏ NULL.

```
BinT_Type BinT_Add_Left(BinT_Type &BT_Tree, T NewData)
  { BinT_Type NewNode = BinT_Create_Node(NewData);
    if (NewNode == NULL)
       return (NewNode);
    if (BT_Tree == NULL)
```

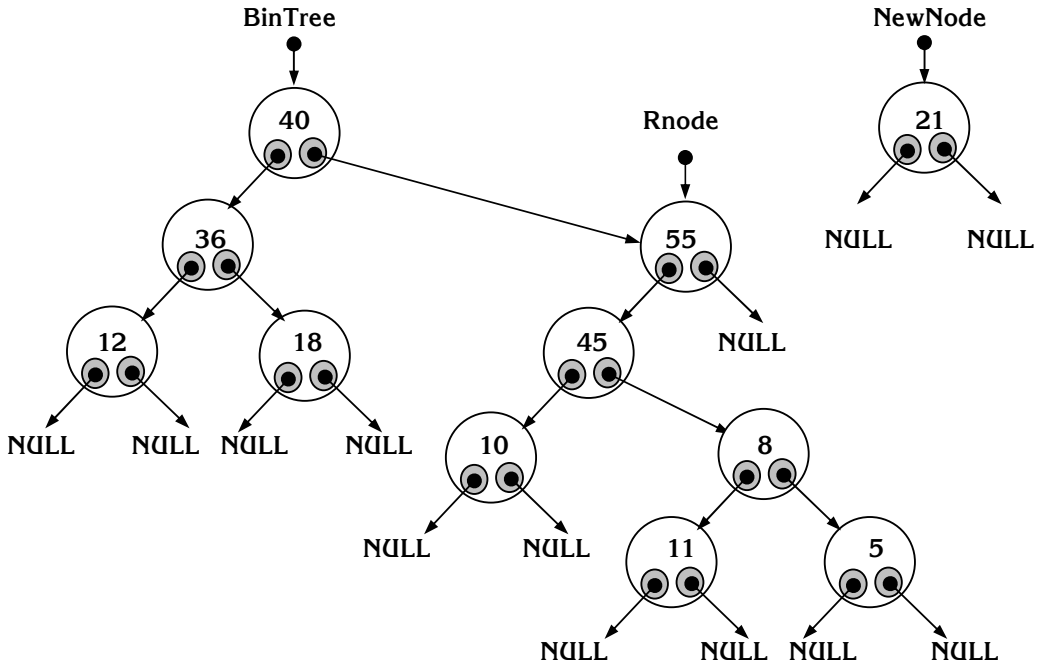
```
            BT_Tree = NewNode;
        else
            {  BinT_Type  Lnode = BT_Tree;
               while (Lnode->BinT_Left != NULL)
                  Lnode = Lnode->BinT_Left;
               Lnode->BinT_Left = NewNode;
            }
        return (NewNode);
    }
```

- Thuật toán thêm 1 nút vào bên phải nhất của cây nhị phân:

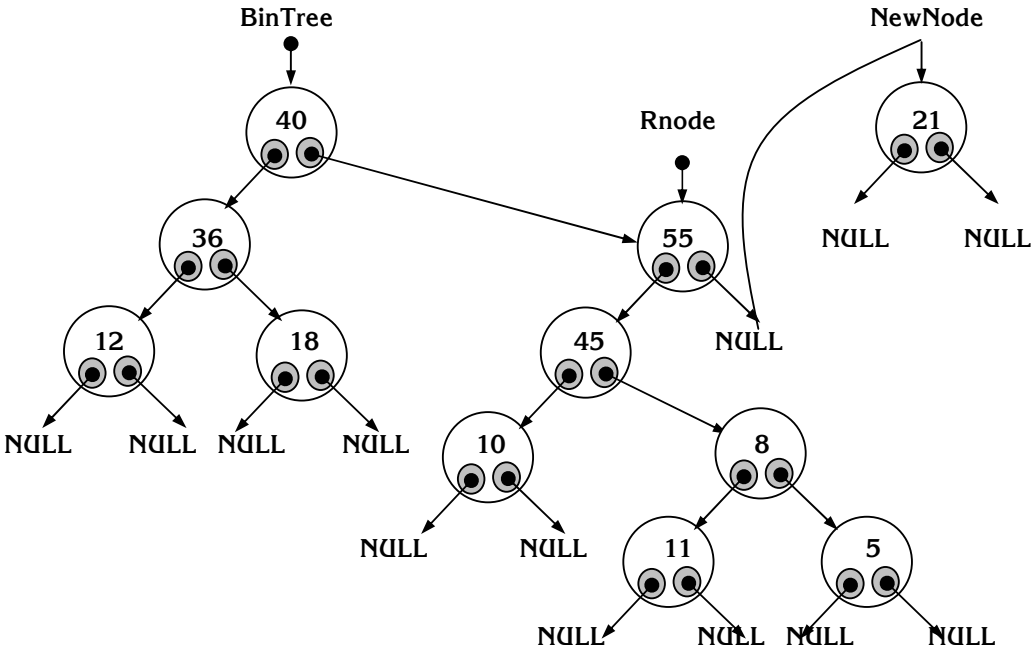
B1: NewNode = BinT_Create_Node (NewData)
B2: IF (NewNode = NULL)
    Thực hiện Bkt
B3: IF (BinTree = NULL) // Cây rỗng
    B3.1: BinTree = NewNode
    B3.2: Thực hiện Bkt
B4: Rnode = BinTree
B5: IF (Rnode->BinT_Right = NULL) // Cây con phải rỗng
    B5.1: Rnode->BinT_Right = NewNode
    B5.2: Thực hiện Bkt
B6: Rnode = Rnode->BinT_Right // Đi theo nhánh cây con phải
B7: Lặp lại B5
Bkt: Kết thúc

- Minh họa thuật toán:

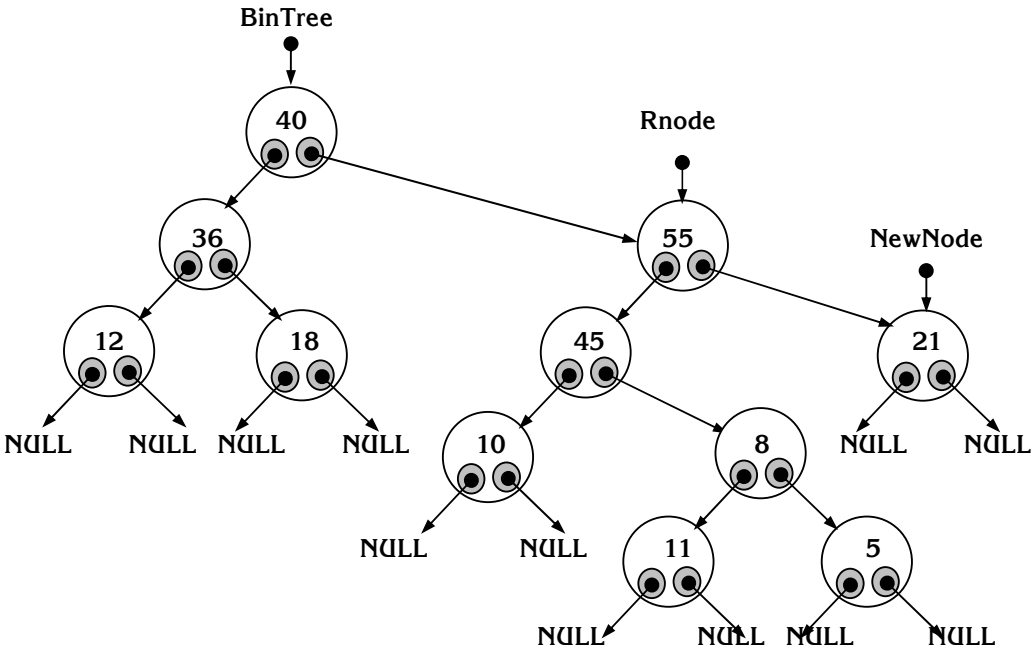
Giả sử chúng ta cần thêm nút có thành phần dữ liệu là 21 vào bên phải nhất của cây nhị phân: NewData = 21



B5.1: Rnode->BinT_Right = NewNode



Kết quả sau khi thêm:



- Cài đặt thuật toán:

Hàm BinT_Add_Right có prototype:

BinT_Type BinT_Add_Right(BinT_Type &BT_Tree, T NewData);

Hàm thực hiện việc thêm vào bên phải nhất trong cây nhị phân BT_Tree một nút có thành phần dữ liệu là NewData, hàm trả về con trỏ trỏ tới địa chỉ của nút mới thêm nếu việc thêm thành công, ngược lại nếu không đủ bộ nhớ, hàm trả về con trỏ NULL.

BinT_Type BinT_Add_Right(BinT_Type &BT_Tree, T NewData)

```
{  BinT_Type  NewNode = BinT_Create_Node(NewData);
   if (NewNode == NULL)
      return (NewNode);
   if (BT_Tree == NULL)
      BT_Tree = NewNode;
   else
      {  BinT_Type  Rnode = BT_Tree;
         while (Rnode->BinT_Right != NULL)
            Rnode = Rnode->BinT_Right;
         Rnode->BinT_Right = NewNode;
      }
   return (NewNode);
}
```

d. Duyệt qua các nút trên cây nhị phân:

Trong thao tác này chúng ta tìm cách duyệt qua (ghé thăm) tất cả các nút trong cây nhị phân để thực hiện một thao tác xử lý nào đó đối với nút này (Xem nội dung thành phần dữ liệu chẳng hạn). Căn cứ vào thứ tự duyệt nút gốc so với 2 nút gốc cây con, thao tác duyệt có thể thực hiện theo một trong ba thứ tự:

- Duyệt theo thứ tự nút gốc trước (Preorder):

Theo cách duyệt này thì nút gốc sẽ được duyệt trước sau đó mới duyệt đến hai cây con. Căn cứ vào thứ tự duyệt hai cây con mà chúng ta có hai cách duyệt theo thứ tự nút gốc trước:

+ Duyệt nút gốc, duyệt cây con trái, duyệt cây con phải (Root – Left – Right)
+ Duyệt nút gốc, duyệt cây con phải, duyệt cây con trái (Root – Right - Left)

- Duyệt theo thứ tự nút gốc giữa (Inorder):

Theo cách duyệt này thì chúng ta duyệt một trong hai cây con trước rồi đến duyệt nút gốc và sau đó mới duyệt cây con còn lại. Căn cứ vào thứ tự duyệt hai cây con chúng ta cũng sẽ có hai cách duyệt theo thứ tự nút gốc giữa:

+ Duyệt cây con trái, duyệt nút gốc, duyệt cây con phải (Left – Root - Right)
+ Duyệt cây con phải, duyệt nút gốc, duyệt cây con trái (Right – Root - Left)

- Duyệt theo thứ tự nút gốc sau (Postorder):

Tương tự như duyệt theo nút gốc trước, trong cách duyệt này thì nút gốc sẽ được duyệt sau cùng so với duyệt hai nút gốc cây con. Do vậy, căn cứ vào thứ tự duyệt hai cây con mà chúng ta cũng có hai cách duyệt theo thứ tự nút gốc sau:

+ Duyệt cây con trái, duyệt cây con phải, duyệt nút gốc (Left – Right - Root)
+ Duyệt cây con phải, duyệt cây con trái, duyệt nút gốc (Right – Left - Root)

Trong phần này chúng ta chỉ trình bày một cách duyệt theo một thứ tự cụ thể đó là: Duyệt cây con trái, duyệt nút gốc và duyệt cây con phải (Left – Root – Right) và sử dụng thuật toán đệ quy. Các cách duyệt khác bằng thuật toán đệ quy hay không đệ quy sinh viên tự vận dụng tương tự.

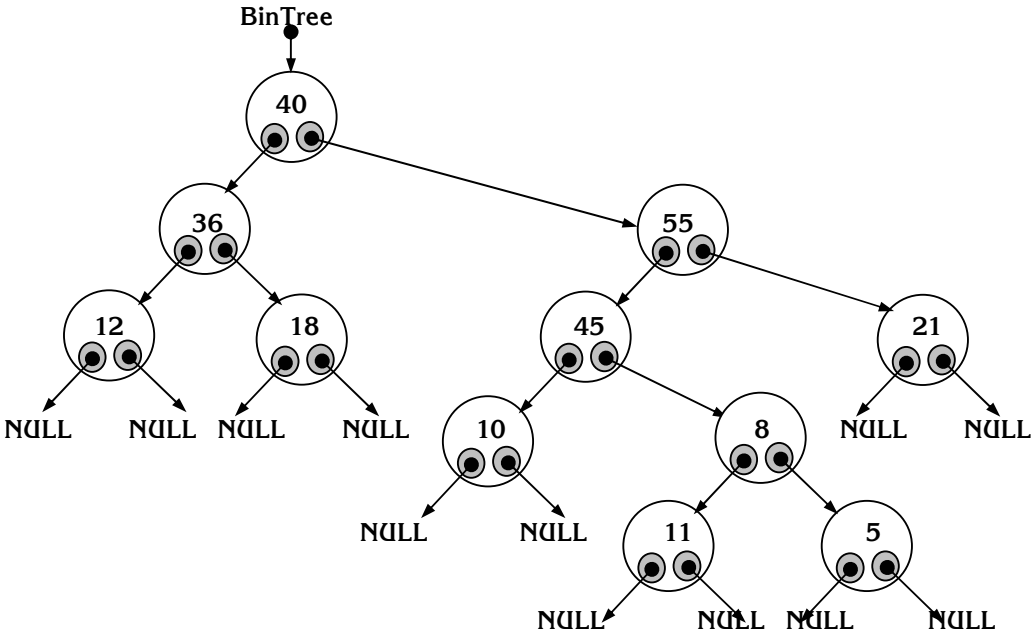
- Thuật toán đệ quy để duyệt cây nhị phân theo thứ tự Left – Root – Right (LRootR):

B1: CurNode = BinTree

B2: IF (CurNode = NULL)
    Thực hiện Bkt
B3: LRootR (BinTree->BinT_Left) // Duyệt cây con trái
B4: Process (CurNode->Key) // Xử lý thông tin nút gốc
B5: LRootR (BinTree->BinT_Right) // Duyệt cây con phải
Bkt: Kết thúc

- Minh họa thuật toán:

Giả sử chúng ta cần duyệt qua các nút trong cây nhị phân dưới đây theo thứ tự Left – Root – Right:



LRootR(BinTree->BinT_Left)

LRootR(BinTree->BinT_Left->BinT_Left)

LRootR(NULL)
Process(12)
LRootR(NULL)

Process(36)

LRootR(BinTree->BinT_Left->BinT_Right)

LRootR(NULL)
Process(18)
LRootR(NULL)

Process(40)

LRootR(BinTree->BinT_Right)

LRootR(BinTree->BinT_Right->BinT_Left)

LRootR(BinTree->BinT_Right->BinT_Left->BinT_Left)
LRootR(NULL)
Process(10)
LRootR(NULL)

Process(45)

LRootR(BinTree->BinT_Right->BinT_Left->BinT_Right)

LRootR(BinTree->BinT_Right->BinT_Left->BinT_Right->BinT_Left)

LRootR(NULL)
Process(11)
LRootR(NULL)

Process(8)

LRootR(BinTree->BinT_Right->BinT_Left->BinT_Right->BinT_Right)

LRootR(NULL)
Process(5)
LRootR(NULL)

Process(55)

LRootR(BinTree->BinT_Right->BinT_Right)

LRootR(NULL)
Process(21)
LRootR(NULL)

Như vậy thứ tự các thông tin của các nút được xử lý như sau:

12 -> 36 -> 18 -> 40 -> 10 -> 45 -> 11 -> 8 -> 5 -> 55 -> 21

- Cài đặt thuật toán:

Hàm BinT_LRootR_Travelling có prototype:

```
void  BinT_LRootR_Travelling(BinT_Type  BT_Tree);
```

Hàm thực hiện thao tác duyệt qua tất cả các nút trong cây nhị phân BT_Tree theo thứ tự duyệt Left – Root – Right để xử lý thông tin ở mỗi nút.

```
void  BinT_LRootR_Travelling(BinT_Type  BT_Tree)
  {  if (BT_Tree == NULL)
        return;
     BinT_LRootR_Travelling (BT_Tree->BinT_Left);
     Process (BT_Tree->Key)
     BinT_LRootR_Travelling (BT_Tree->BinT_Right);
     return;
  }
```

☛ **Lưu ý**:

Hàm Process thực hiện việc xử lý thông tin (Key) của mỗi nút. Do vậy tùy từng trường hợp cụ thể mà chúng ta viết hàm cho phù hợp. Chẳng hạn để xuất thông tin thì chỉ cần các lệnh xuất dữ liệu để xuất thành phần Key.

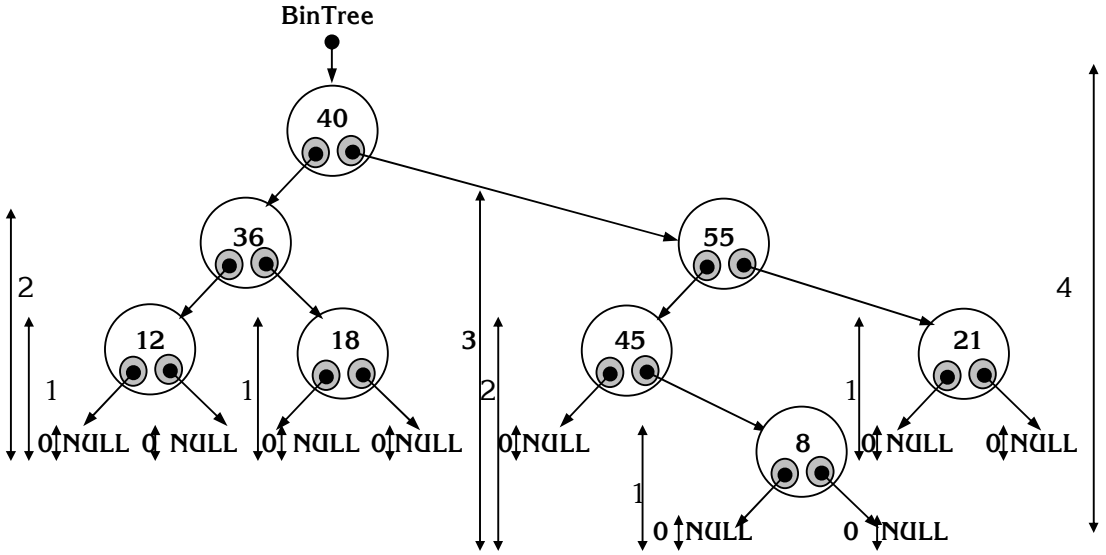
e. Tính chiều cao của cây:

Để tính chiều cao của cây (TH) chúng ta phải tính chiều cao của các cây con, khi đó chiều cao của cây chính là chiều cao lớn nhất của các cây con cộng thêm 1 (chiều cao nút gốc). Như vậy thao tác tính chiều cao của cây là thao tác tính đệ quy chiều cao của các cây con (chiều cao của cây con có gốc là nút lá bằng 1).

- Thuật toán:

B1: IF (BinTree = NULL)
    B1.1: TH = 0
    B1.2: Thực hiện Bkt
B2: THL = TH(BinTree->BinT_Left)
B3: THR = TH(BinTree->BinT_Right)
B4: IF (THL > THR)
    TH = THL + 1
B5: ELSE
    TH = THR + 1
Bkt: Kết thúc

Ví dụ: Chiều cao của cây nhị phân sau bằng 4.



- Cài đặt thuật toán:

Hàm BinT_Height có prototype:

int BinT_Height(BinT_Type BTree);

Hàm tính chiều cao của cây BTree theo thuật toán đệ quy. Hàm trả về chiều cao của cây cần tính.

```
int BinT_Height(BinT_Type BTree)
  { if (BTree == NULL)
        return (0);
    int HTL = BinT_Height(BTree->BinT_Left);
    int HTR = BinT_Height(BTree->BinT_Right);
    if (HTL > HTR)
       return (HTL+1);
    return (HTR+1);
  }
```

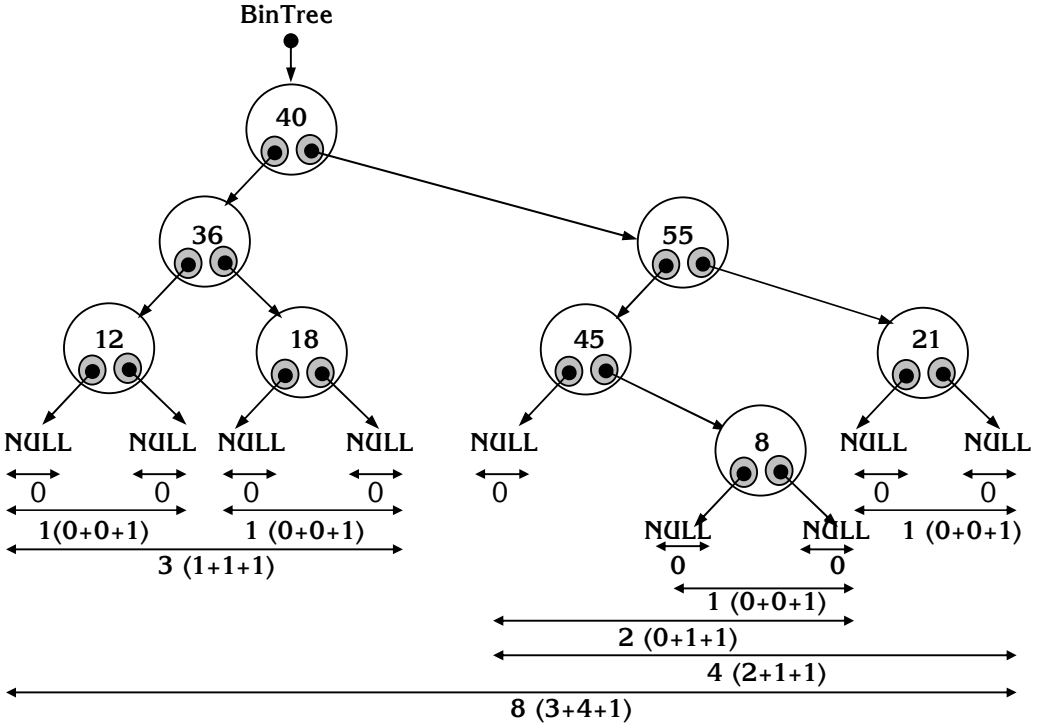
f. Tính số nút của cây:

Tương tự như tính chiều cao của cây, số nút của cây (NN) bằng tổng số nút của hai cây con cộng thêm 1. Do vậy thao tác này chúng ta cũng sẽ tính đệ quy số nút của các cây con (số nút của cây con có gốc là nút lá bằng 1).

- Thuật toán:

B1: IF (BinTree = NULL)
    B1.1: NN = 0
    B1.2: Thực hiện Bkt
B2: NNL = NN(BinTree->BinT_Left)
B3: NNR = NN(BinTree->BinT_Right)
B4: NN = NNL + NNR + 1
Bkt: Kết thúc

Ví dụ: Số nút của cây nhị phân sau bằng 8.



- Cài đặt thuật toán:

Hàm BinT_Num_Node có prototype:

int BinT_Num_Node(BinT_Type BTree);

Hàm tính số nút của cây BTree theo thuật toán đệ quy. Hàm trả về số nút của cây cần tính.

```
int  BinT_Num_Node(BinT_Type  BTree)
   {  if (BTree == NULL)
         return (0);
      int  NNL = BinT_Num_Node(BTree->BinT_Left);
      int  NNR = BinT_Num_Node(BTree->BinT_Right);
      return (NNL + NNR + 1);
   }
```

g. Hủy một nút trên cây nhị phân:

Việc hủy một nút trong cây có thể làm cho cây trở thành rừng. Do vậy trong thao tác này nếu chúng ta tiến hành hủy một nút lá thì không có điều gì xảy ra, song nếu hủy

nút không phải là nút lá thì chúng ta phải tìm cách chuyển các nút gốc cây con là các nút con của nút cần hủy thành các nút gốc cây con của các nút khác rồi mới tiến hành hủy nút này.

- Trường hợp nếu nút cần hủy chỉ có 01 nút gốc cây con thì chúng ta có thể chuyển nút gốc cây con này thành nút gốc cây con của nút cha của nút cần hủy.
- Trường hợp nếu nút cần hủy có 2 nút gốc cây con thì chúng ta phải chuyển 02 nút gốc cây con này thành nút gốc cây con của các nút khác với nút cần hủy. Việc chọn các nút để làm nhiệm vụ nút cha của các nút gốc cây con này tùy vào từng trường hợp cụ thể của cây nhị phân mà chúng ta sẽ lựa chọn cho phù hợp.

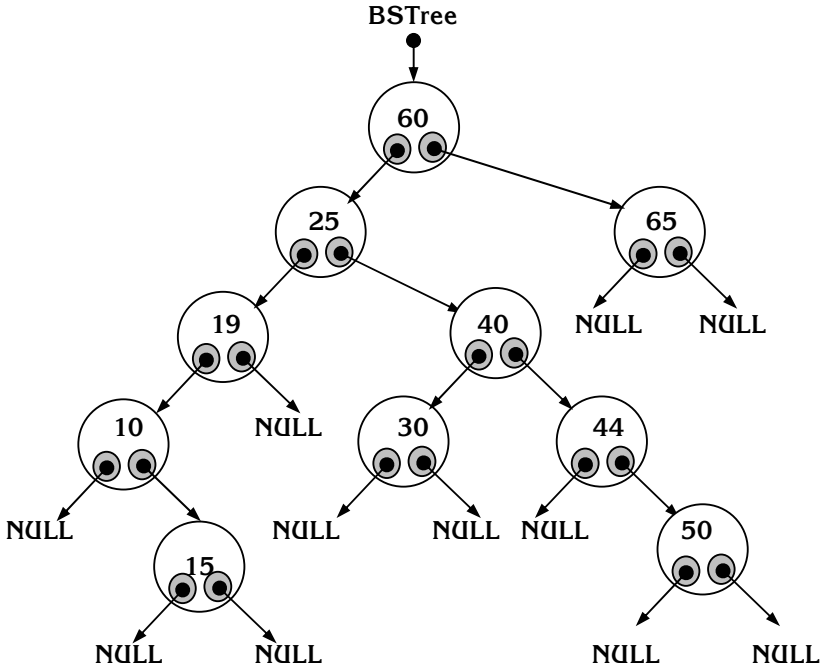
Do vậy, thao tác hủy một nút sẽ được trình bày cụ thể trong các loại cây cụ thể được trình bày ở các phần sau.

5.2.3. Cây nhị phân tìm kiếm (Binary Searching Tree)

A. Khái niệm – Cấu trúc dữ liệu:

Cây nhị phân tìm kiếm là cây nhị phân có thành phần khóa của mọi nút lớn hơn thành phần khóa của tất cả các nút trong cây con trái của nó và nhỏ hơn thành phần khóa của tất cả các nút trong cây con phải của nó.

Ví dụ: Hình ảnh sau là hình ảnh của một cây nhị phân tìm kiếm



Từ khái niệm này chúng ta có một số nhận xét:

- Cấu trúc dữ liệu của cây nhị phân tìm kiếm là cấu trúc dữ liệu để biểu diễn các cây nhị phân nói chung.

```
typedef  struct  BST_Node
   {  T  Key;
      BST_Node * BST_Left; // Vùng liên kết quản lý địa chỉ nút gốc cây con trái
      BST_Node * BST_Right; // Vùng liên kết quản lý địa chỉ nút gốc cây con phải
   }    BST_OneNode;
```

```
typedef BST_OneNode * BST_Type;
```

Để quản lý các cây nhị phân tìm kiếm chúng ta cần quản lý địa chỉ nút gốc của cây:

```
BST_Type BSTree;
```

- Khóa nhận diện (Key) của các nút trong cây nhị phân tìm kiếm đôi một khác nhau (không có hiện tượng trùng khóa).

Tuy nhiên trong trường hợp cần quản lý các nút có khóa trùng nhau trong cây nhị phân tìm kiếm thì chúng ta có thể mở rộng cấu trúc dữ liệu của mỗi nút bằng cách thêm thành phần Count để ghi nhận số lượng các nút trùng khóa. Khi đó, cấu trúc dữ liệu để quản lý các cây nhị phân tìm kiếm được mở rộng như sau:

```
typedef struct BSE_Node
{ T Key;
  int Count;
  BSE_Node * BSE_Left; // Vùng liên kết quản lý địa chỉ nút gốc cây con trái
  BSE_Node * BSE_Right; // Vùng liên kết quản lý địa chỉ nút gốc cây con phải
}  BSE_OneNode;

typedef BSE_OneNode * BSE_Type;
```

và chúng ta quản lý cây nhị phân tìm kiếm này bằng cách quản lý địa chỉ nút gốc:

```
BSE_Type BSETree;
```

- Nút ở bên trái nhất là nút có giá trị khóa nhận diện nhỏ nhất và nút ở bên phải nhất là nút có giá trị khóa nhận diện lớn nhất trong cây nhị phân tìm kiếm.
- Trong một cây nhị phân tìm kiếm thứ tự duyệt cây Left – Root – Right là thứ tự duyệt theo sự tăng dần các giá trị của Key trong các nút và thứ tự duyệt cây Right – Root – Left là thứ tự duyệt theo sự giảm dần các giá trị của Key trong các nút.

B. Các thao tác trên cây nhị phân tìm kiếm:

a. Tìm kiếm trên cây:

Giả sử chúng ta cần tìm trên cây nhị phân tìm kiếm xem có tồn tại nút có khóa Key là SearchData hay không.

Để thực hiện thao tác này chúng ta sẽ vận dụng thuật toán tìm kiếm nhị phân: Do đặc điểm của cây nhị phân tìm kiếm thì tại một nút, nếu Key của nút này khác với SearchData thì SearchData chỉ có thể tìm thấy hoặc trên cây con trái của nút này nếu SearchData nhỏ hơn Key của nút này hoặc trên cây con phải của nút này nếu SearchData lớn hơn Key của nút này.

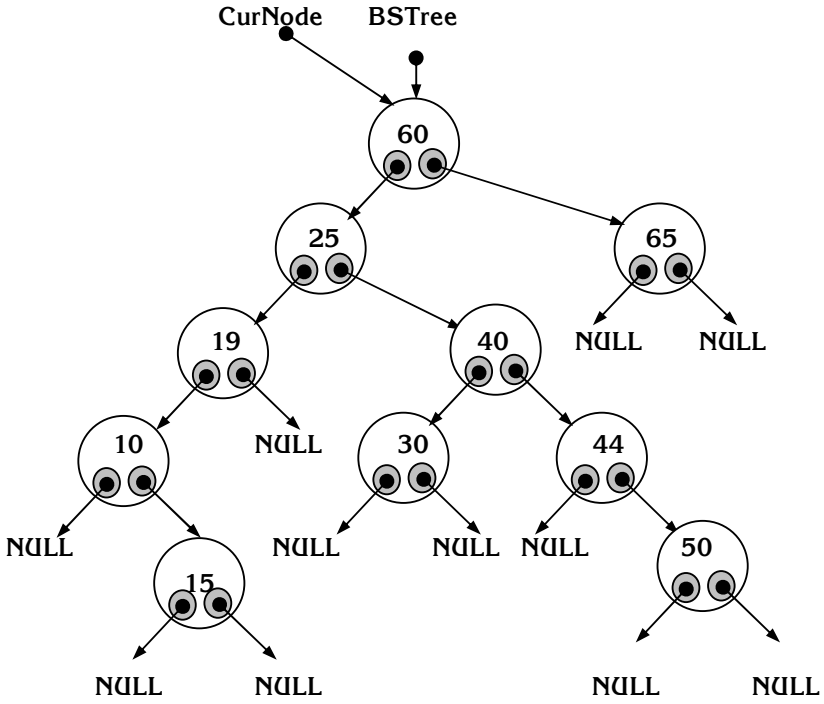
- Thuật toán tìm kiếm 1 nút trên cây nhị phân tìm kiếm:

```
B1: CurNode = BSTree
B2: IF (CurNode = NULL) or (CurNode->Key = SearchData)
        Thực hiện Bkt
B3: IF (CurNode->Key > SearchData)       // Tìm kiếm trên cây con trái
        CurNode = CurNode->BST_Left
B4: ELSE        // Tìm kiếm trên cây con phải
        CurNode = CurNode->BST_Right
B5: Lặp lại B2
```

Bkt: Kết thúc

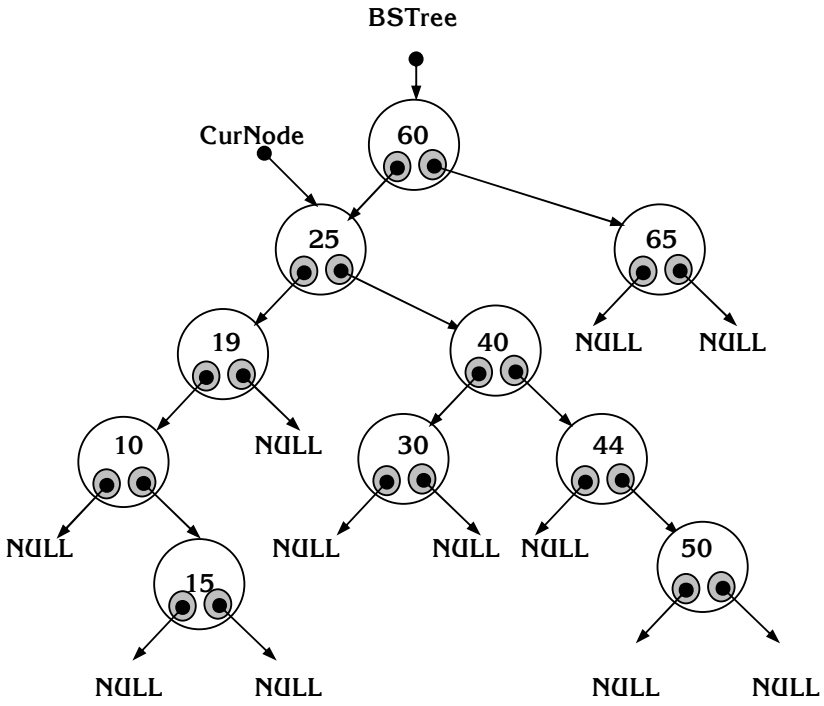
- Minh họa thuật toán:

Giả sử chúng ta cần tìm kiếm nút có thành phần dữ liệu là 30 trên cây nhị phân tìm kiếm sau: SearchData = 30

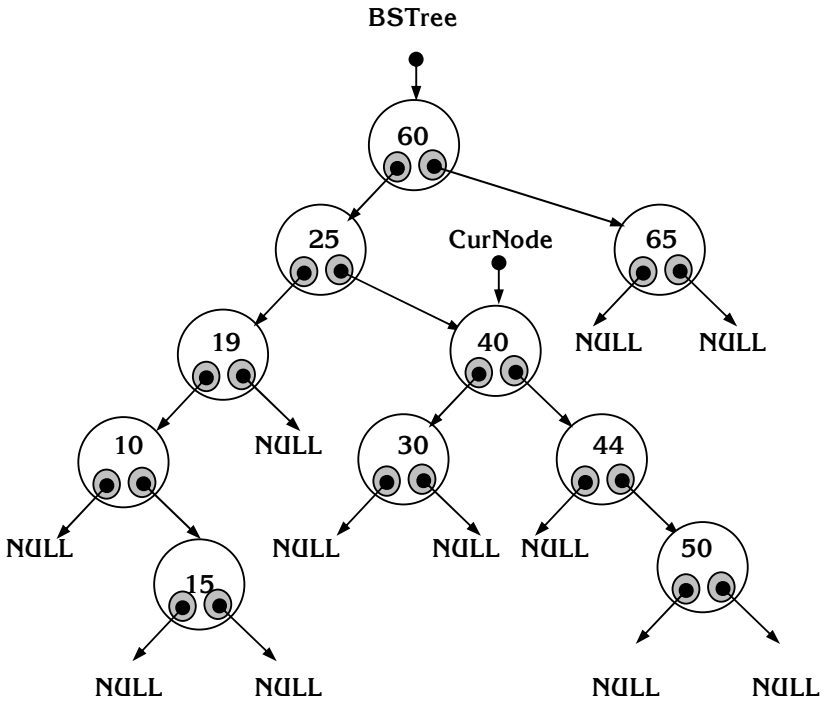


CurNode->Key > SearchData // Tìm kiếm trên cây con trái

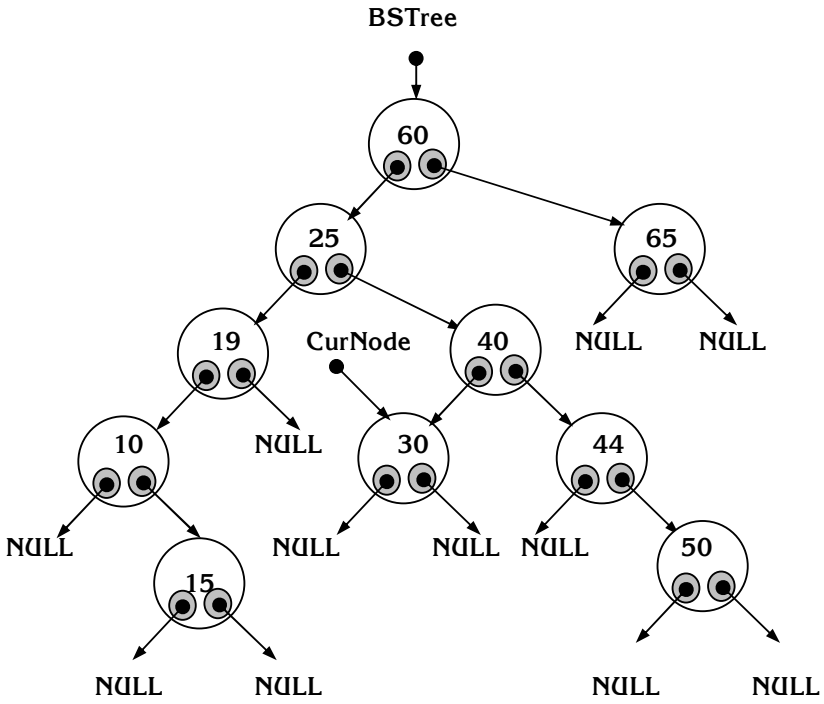
⇒ CurNode = CurNode->BST_Left



CurNode->Key < SearchData // Tìm kiếm trên cây con phải
⇒ CurNode = CurNode->BST_Right

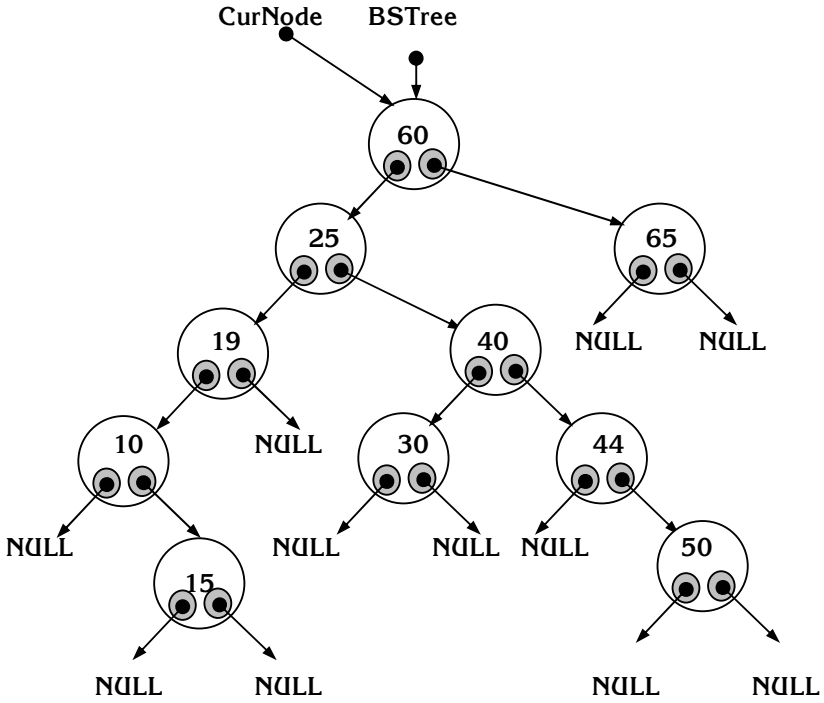


CurNode->Key > SearchData // Tìm kiếm trên cây con trái
⇒ CurNode = CurNode->BST_Left



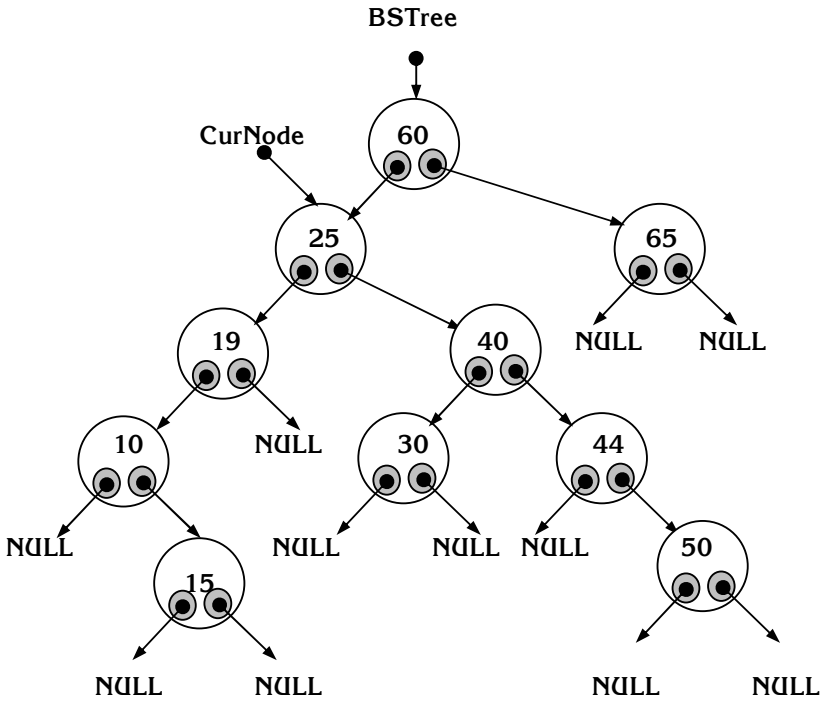
CurNode->Key = SearchData ⇒ Thuật toán kết thúc (Tìm thấy)

Bây giờ giả sử chúng ta cần tìm kiếm nút có thành phần dữ liệu là 35 trên cây nhị phân tìm kiếm trên: SearchData = 35

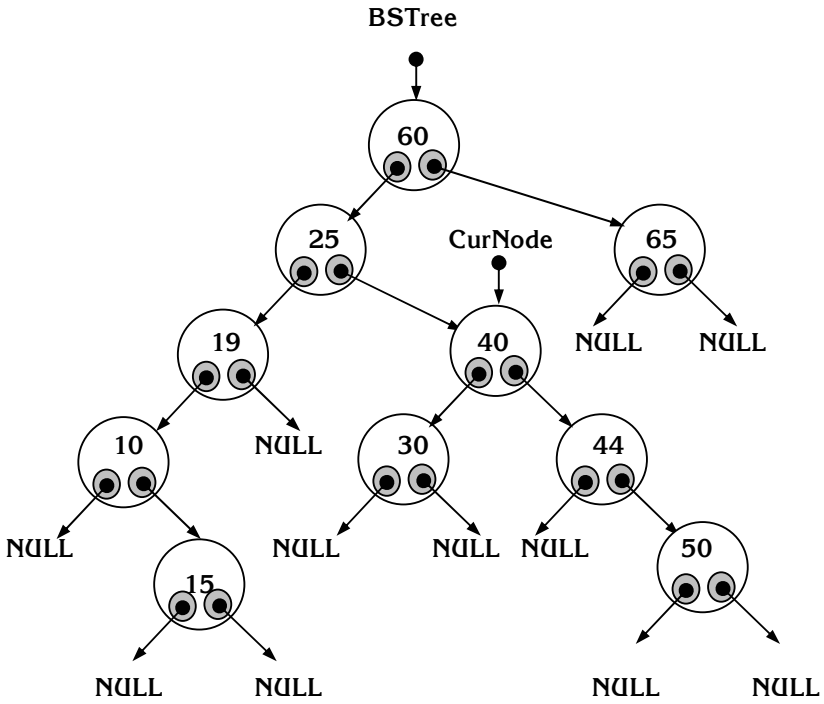


CurNode->Key > SearchData // Tìm kiếm trên cây con trái

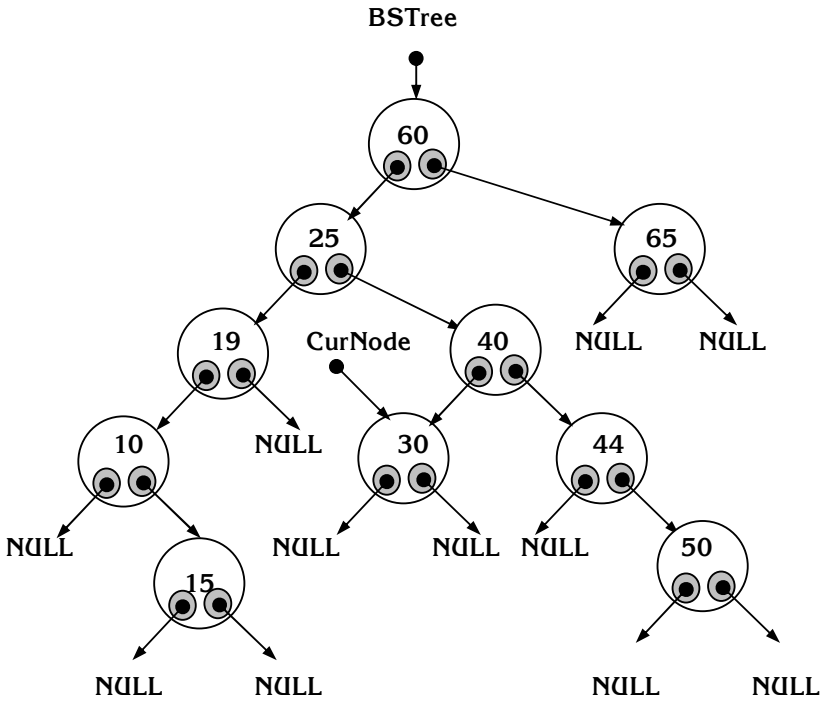
⇒ CurNode = CurNode->BST_Left



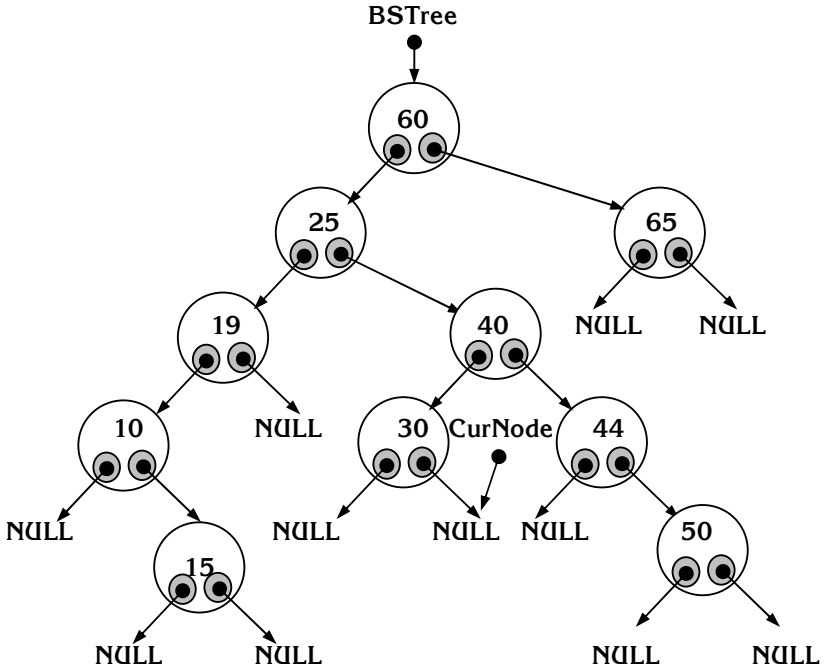
CurNode->Key < SearchData // Tìm kiếm trên cây con phải
⇒ CurNode = CurNode->BST_Right



CurNode->Key > SearchData // Tìm kiếm trên cây con trái
⇒ CurNode = CurNode->BST_Left



CurNode->Key < SearchData // Tìm kiếm trên cây con phải
⇒ CurNode = CurNode->BST_Right



CurNode = NULL ⇒ Thuật toán kết thúc (Không tìm thấy)

- Cài đặt thuật toán:

Hàm BST_Searching có prototype:

BST_Type BST_Searching(BST_Type BS_Tree, T SearchData);

Hàm thực hiện thao tác tìm kiếm trên cây nhị phân tìm kiếm BS_Tree nút có thành phần Key là SearchData. Hàm trả về con trỏ trỏ tới địa chỉ của nút có Key là SearchData nếu tìm thấy, trong trường hợp ngược lại hàm trả về con trỏ NULL.

```
BST_Type BST_Searching(BST_Type BS_Tree, T SearchData)
  { BST_Type CurNode = BS_Tree;
    while (CurNode != NULL && CurNode->Key != SearchData)
      { if (CurNode->Key > SearchData)
          CurNode = CurNode->BST_Left;
        else
          CurNode = CurNode->BST_Right;
      }
    return (CurNode);
  }
```

b. Thêm một nút vào trong cây:

Giả sử chúng ta cần thêm một nút có thành phần dữ liệu (Key) là NewData vào trong cây nhị phân tìm kiếm sao cho sau khi thêm cây vẫn là một cây nhị phân tìm kiếm. Trong thao tác này trước hết chúng ta phải tìm kiếm vị trí thêm, sau đó mới tiến hành thêm nút mới vào cây (Do vậy thuật toán còn được gọi là thuật toán tìm kiếm và thêm vào cây). Quá trình tìm kiếm tuân thủ các bước trong thuật toán tìm kiếm đã trình bày ở trên.

Trong thuật toán này chúng ta sẽ trình bày thao tác thêm vào cây nhị phân tìm kiếm trong trường hợp không có hiện tượng trùng lắp khóa. Do vậy, nếu NewData bị trùng

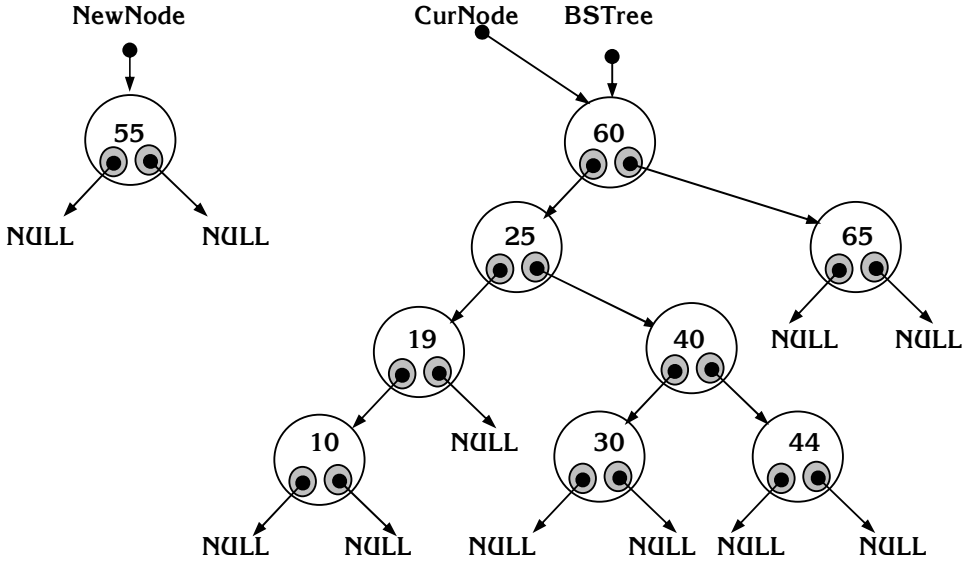
với Key của một trong các nút ở trong cây nhị phân tìm kiếm thì chúng ta sẽ không thực hiện thao tác thêm này. Tuy nhiên, nếu chúng ta sử dụng cấu trúc dữ liệu mở rộng thì việc trùng khóa sẽ giải quyết đơn giản vì không làm tăng số nút của cây nhị phân tìm kiếm mà chỉ làm tăng thành phần Count của nút bị trùng khóa thêm 1.

- Thuật toán thêm 1 nút vào cây nhị phân tìm kiếm:

```
B1: NewNode = BinT_Create_Node(NewData)
B2: IF (NewNode = NULL)
        Thực hiện Bkt
B3: IF (BSTree = NULL)      // Cây rỗng
        B3.1: BSTree = NewNode
        B3.2: Thực hiện Bkt
B4: CurNode = BSTree
B5: IF (CurNode->Key = NewData) // Trùng khóa
        Thực hiện Bkt
B6: IF (CurNode->Key > NewData)
        B6.1: AddLeft = True    // Thêm vào cây con trái của CurNode
        B6.2: If (CurNode->BST_Left != NULL)
                CurNode = CurNode->BST_Left
B7: IF (CurNode->Key < NewData)
        B7.1: AddLeft = False   // Thêm vào cây con phải của CurNode
        B7.2: If (CurNode->BST_Right != NULL)
                CurNode = CurNode->BST_Right
B8: Lặp lại B5
B9: IF (AddLeft = True)
        CurNode->BST_Left = NewNode
B10: ELSE
        CurNode->BST_Right = NewNode
Bkt: Kết thúc
```

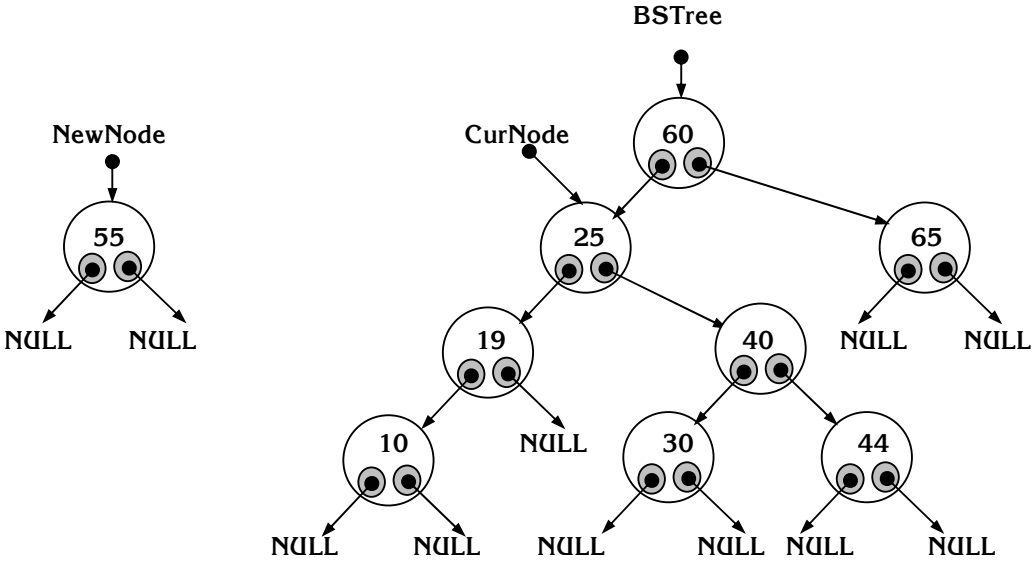
- Minh họa thuật toán:

Giả sử chúng ta cần thêm vào trong cây nhị phân tìm kiếm 1 nút có thành phần dữ liệu là 55: NewData = 55



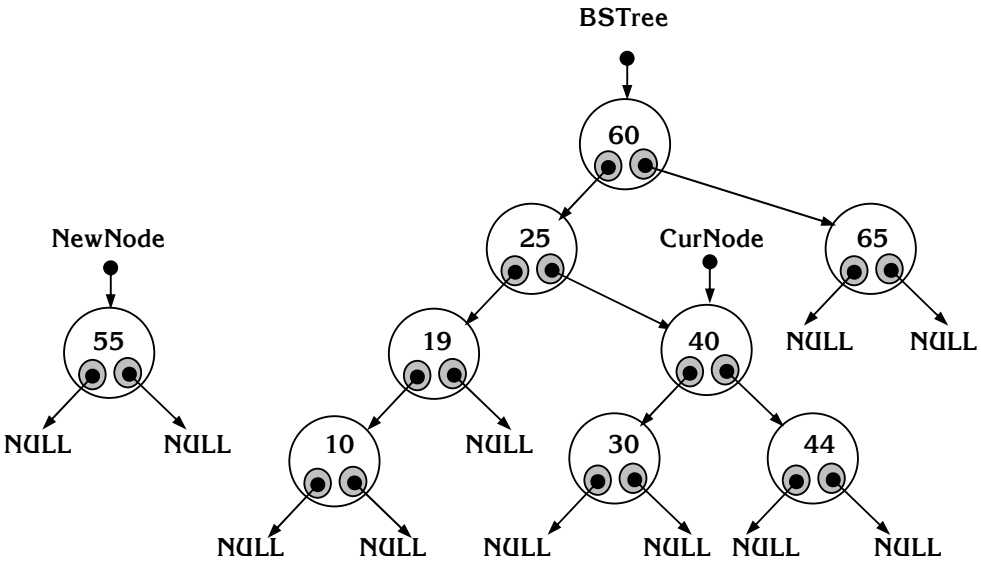
CurNode->Key > NewData // Thêm vào cây con trái
⇒ AddLeft = True

CurNode->BST_Left != NULL // Chuyển sang cây con trái
⇒ CurNode = CurNode->BST_Left



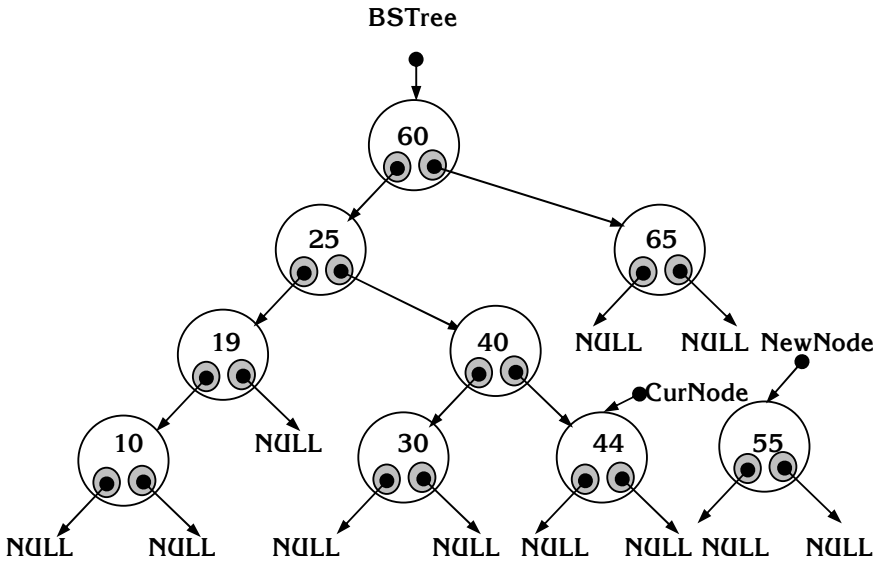
CurNode->Key < NewData // Thêm vào cây con phải
⇒ AddLeft = False

CurNode->BST_Right != NULL // Chuyển sang cây con phải
⇒ CurNode = CurNode->BST_Right



CurNode->Key < NewData // Thêm vào cây con bên phải
⇒ AddLeft = False

CurNode->BST_Right != NULL // Chuyển sang cây con bên phải
⇒ CurNode = CurNode->BST_Right



CurNode->Key < NewData // Thêm vào cây con phải

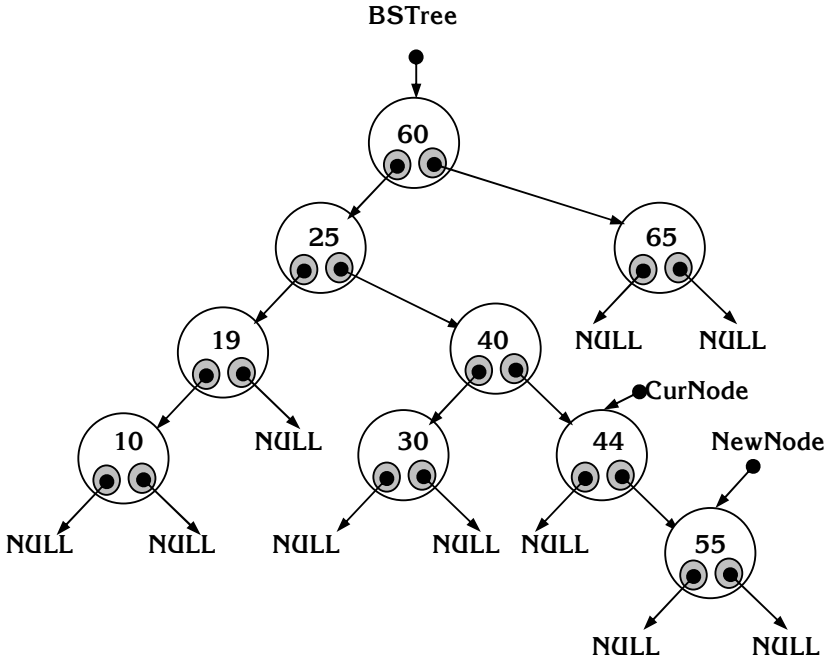
⇒ AddLeft = False

CurNode->BST_Right == NULL

// Thêm NewNode vào thành nút gốc cây con phải của CurNode
// (AddLeft = False), thuật toán kết thúc.

⇒ CurNode->BST_Right = NewNode

Kết quả sau khi thêm:



- Cài đặt thuật toán:

Hàm BST_Add_Node có prototype:

BST_Type BST_Add_Node(BST_Type &BS_Tree, T NewData);

Hàm thực hiện việc thêm vào cây nhị phân tìm kiếm BS_Tree một nút có thành phần Key là NewData. Hàm trả về con trỏ trỏ tới địa chỉ của nút mới thêm nếu việc thêm thành công, trong trường hợp ngược lại hàm trả về con trỏ NULL.

```
BST_Type  BST_Add_Node(BST_Type  &BS_Tree, T  NewData)
  {  BST_Type  NewNode = BinT_Create_Node(NewData);
     if (NewNode == NULL)
        return (NewNode);
     if (BS_Tree == NULL)
        BS_Tree = NewNode;
     else
        {  BST_Type  CurNode = BS_Tree;
           int  AddLeft = 1;
           while (CurNode->Key != NewData)
              {  if (CurNode->Key > NewData)
                    {  AddLeft = 1;
                       if (CurNode->BST_Left != NULL)
                          CurNode = CurNode->BST_Left;
                       else
                          break;
                    }
                 else   // CurNode->Key < NewData
                    {  AddLeft = 0;
                       if (CurNode->BST_Right != NULL)
                          CurNode = CurNode->BST_Right;
                       else
                          break;
                    }
              }
           if (AddLeft == 1)
              CurNode->BST_Left = NewNode;
           else
              CurNode->BST_Right = NewNode;
        }
     return (NewNode);
  }
```

c. Loại bỏ (hủy) một nút trên cây:

Cũng như thao tác thêm một nút vào trong cây nhị phân tìm kiếm, thao tác hủy một nút trên cây nhị phân tìm kiếm cũng phải bảo đảm cho cây sau khi hủy nút đó thì cây vẫn là một cây nhị phân tìm kiếm. Đây là một thao tác không đơn giản bởi nếu không cẩn thận chúng ta sẽ biến cây thành một rừng.

Giả sử chúng ta cần hủy nút có thành phần dữ liệu (Key) là DelData ra khỏi cây nhị phân tìm kiếm. Điều đầu tiên trong thao tác này là chúng ta phải tìm kiếm địa chỉ của nút cần hủy là DelNode, sau đó mới tiến hành hủy nút có địa chỉ là DelNode này nếu tìm thấy (Do vậy thuật toán này còn được gọi là thuật toán tìm kiếm và loại bỏ trên cây). Quá trình tìm kiếm đã trình bày ở trên, ở đây chúng ta chỉ trình bày thao

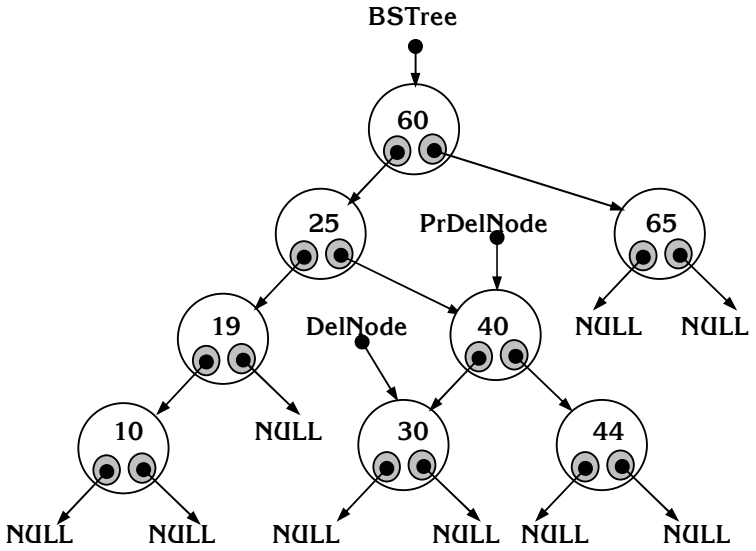
tác hủy khi tìm thấy nút có địa chỉ DelNode (DelNode->Key = DelData) và trong quá trình tìm kiếm chúng ta giữ địa chỉ nút cha của nút cần hủy là PrDelNode.

Việc hủy nút có địa chỉ DelNode có thể xảy ra một trong ba trường hợp sau:

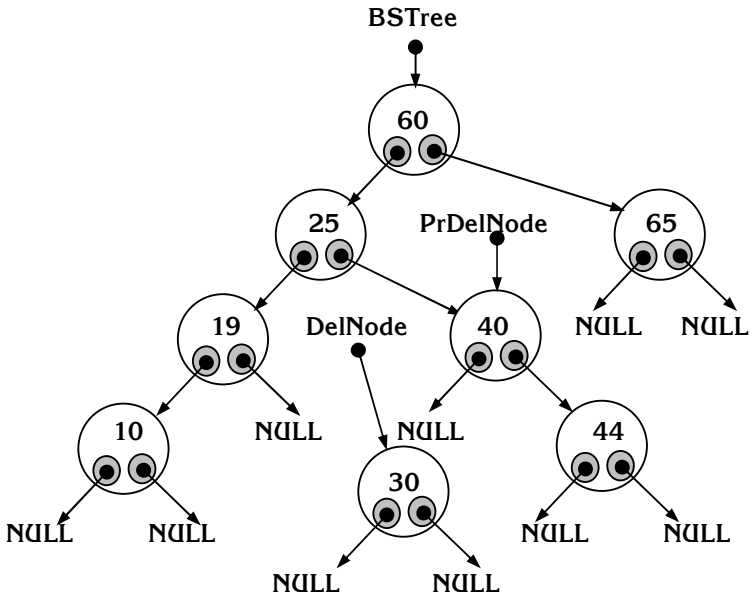
c_1) DelNode là nút lá:

Trong trường hợp này đơn giản chúng ta chỉ cần cắt bỏ mối quan hệ cha-con giữa PrDelNode và DelNode bằng cách cho con trỏ PrDelNode->BST_Left (nếu DelNode là nút con bên trái của PrDelNode) hoặc cho con trỏ PrDelNode->BST_Right (nếu DelNode là nút con bên phải của PrDelNode) về con trỏ NULL và tiến hành hủy (delete) nút có địa chỉ DelNode này.

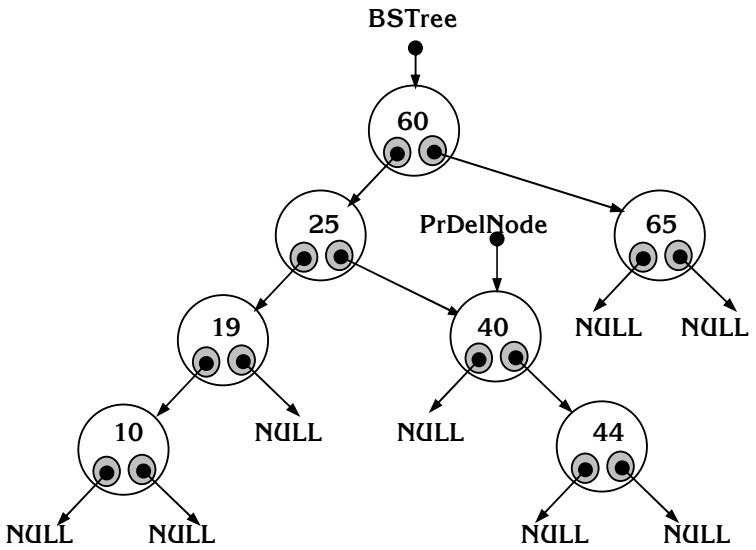
Ví dụ: Giả sử cần hủy nút có Key = 30 (DelData = 30)



Trong trường hợp này chúng ta cho PrDelNode->BST_Left = NULL:



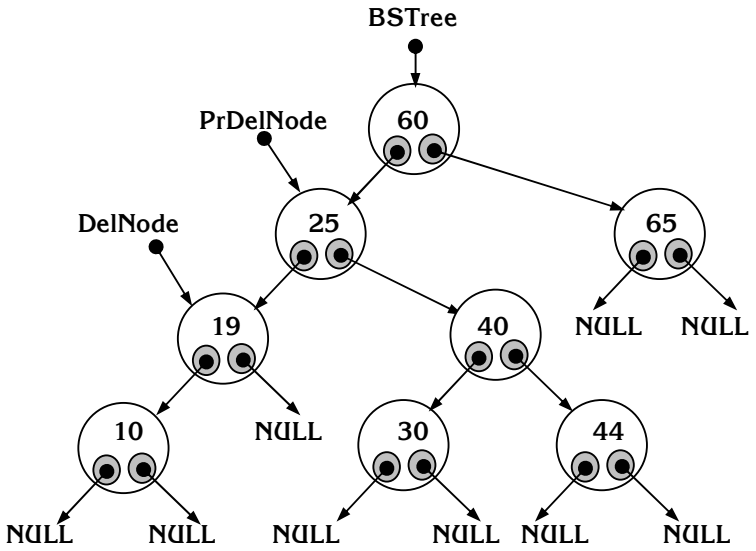
Kết quả sau khi hủy:



c_2) DelNode là nút chỉ có 01 nút gốc cây con:

Trong trường hợp này cũng khá đơn giản chúng ta chỉ cần chuyển mối quan hệ cha-con giữa PrDelNode và DelNode thành mối quan hệ cha-con giữa PrDelNode và nút gốc cây con của DelNode rồi tiến hành cắt bỏ mối quan hệ cha-con giữa DelNode và 01 nút gốc cây con của nó và tiến hành hủy nút có địa chỉ DelNode này.

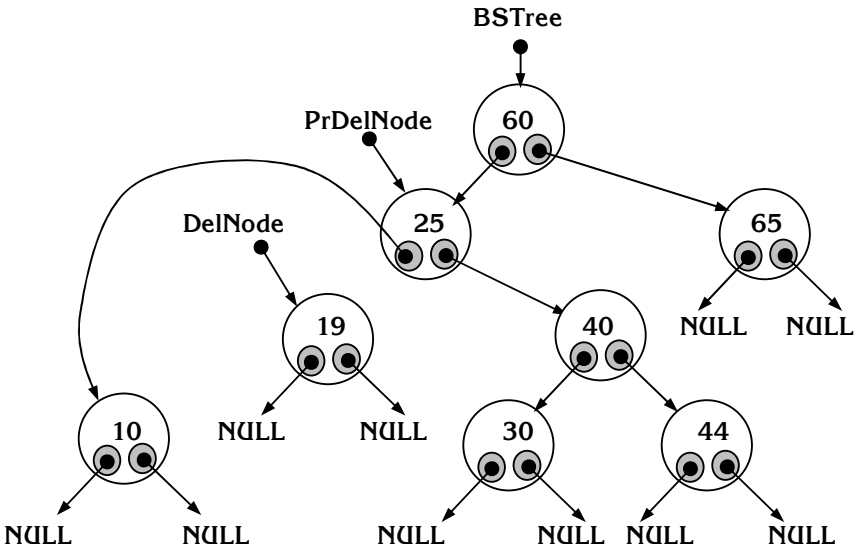
Ví dụ: Giả sử cần hủy nút có Key = 19 (DelData = 19)



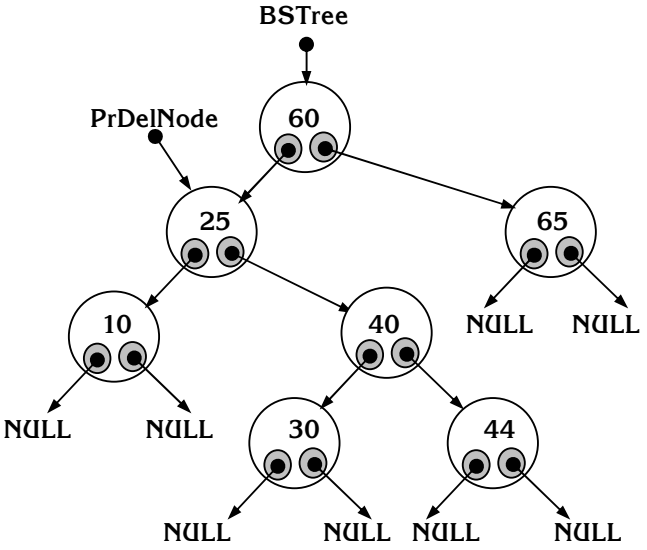
Trong trường hợp này chúng ta thực hiện các bước:

B1: PrDelNode->BST_Left = DelNode->BST_Left

B2: DelNode->BST_Left = NULL



Kết quả sau khi hủy:



c_3) DelNode là nút có đủ 02 nút gốc cây con:

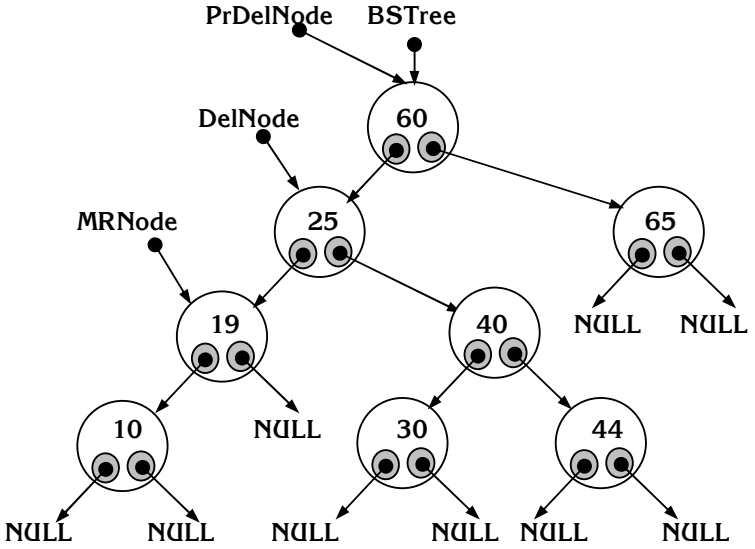
Trường hợp này khá phức tạp, việc hủy có thể tiến hành theo một trong hai cách sau đây (có thể có nhiều cách khác nữa song ở đây chúng ta chỉ trình bày hai cách):

- Chuyển 02 cây con của DelNode về thành một cây con:

Theo phương pháp này chúng ta sẽ chuyển cây con phải của DelNode (DelNode→BST_Right) về thành cây con phải của cây con có nút gốc là nút phải nhất trong cây con trái của DelNode (phải nhất trong DelNode->BST_Left), hoặc chuyển cây con trái của DelNode (DelNode->BST_Left) về thành cây con trái của cây con có nút gốc là nút trái nhất trong cây con phải của DelNode (trái nhất trong

DelNode->BST_Right). Sau khi chuyển thì DelNode sẽ trở thành nút lá hoặc nút chỉ có 01 cây con và chúng ta hủy DelNode như đối với trường hợp c_1) và c_2) ở trên.

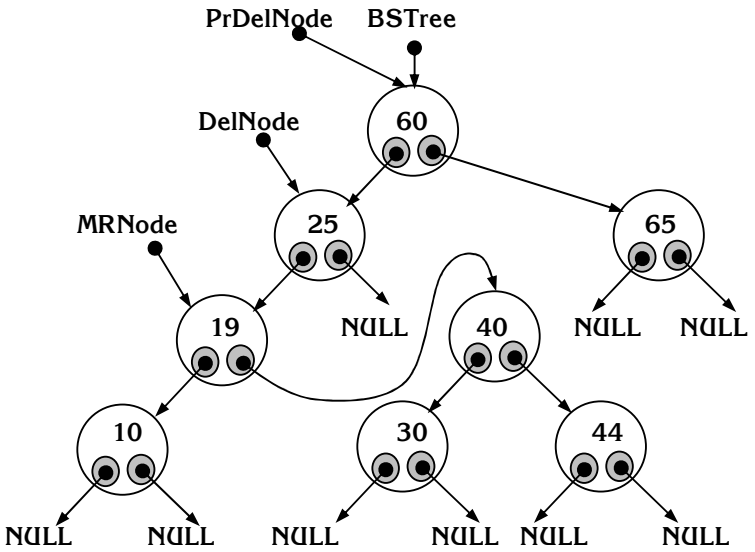
Ví dụ: Giả sử cần hủy nút có Key = 25 (DelData = 25). Chúng ta sẽ chuyển cây con phải của DelNode (DelNode->BST_Right) về thành cây con phải của cây con có nút gốc là nút phải nhất trong cây con trái của DelNode (nút MRNode).



Trong trường hợp này chúng ta thực hiện các bước:

B1: MRNode->BST_Right = DelNode->BST_Right

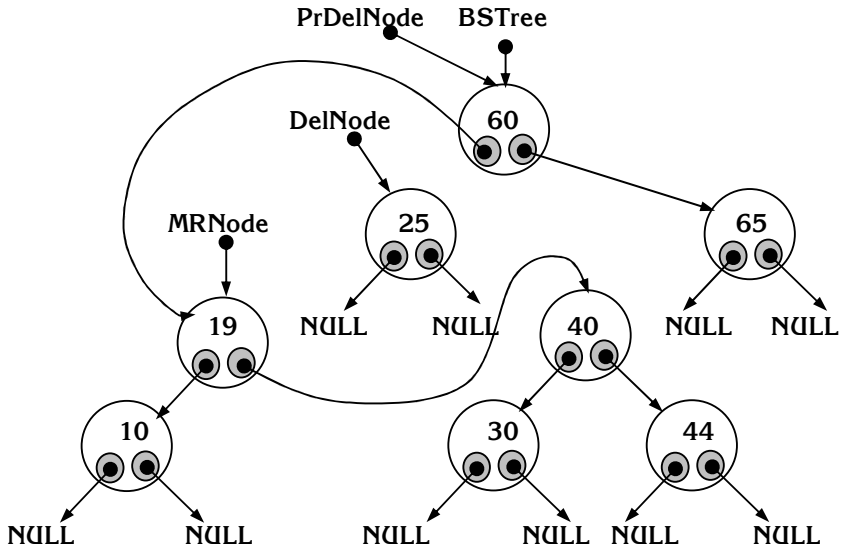
B2: DelNode->BST_Right = NULL



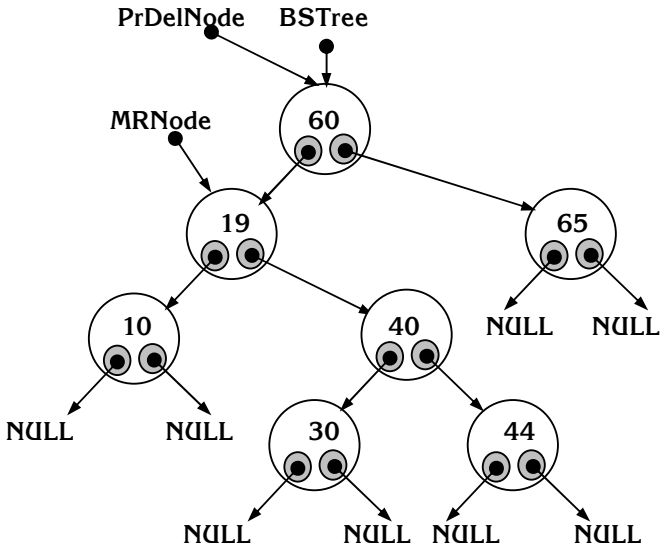
Tiến hành các bước để hủy DelNode:

B3: PrDelNode->BST_Left = DelNode->BST_Left

B4: DelNode->BST_Left = NULL



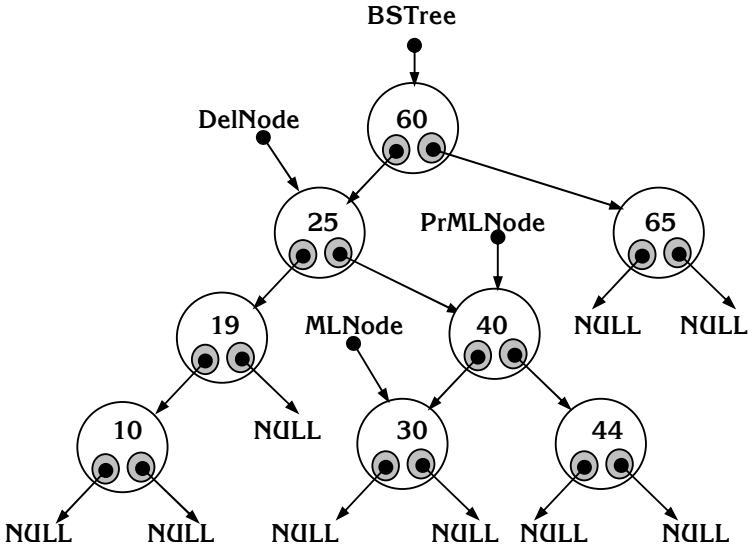
Kết quả sau khi hủy:



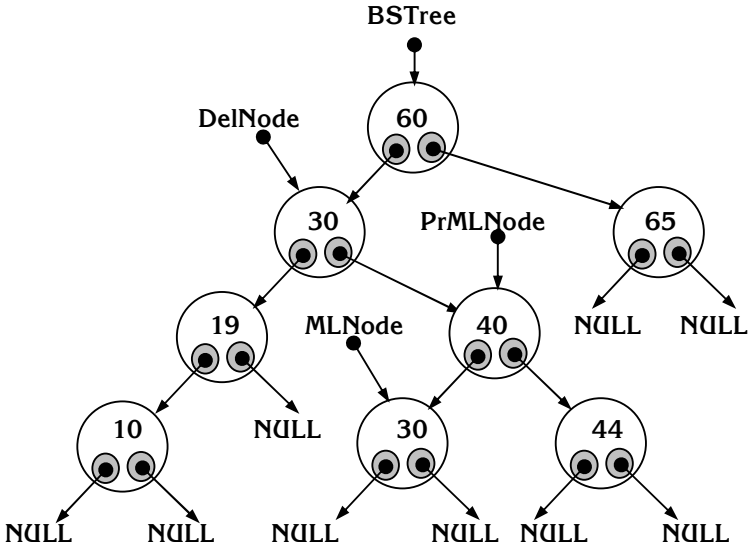
- Sử dụng phần tử thế mạng (standby):

Theo phương pháp này chúng ta sẽ không hủy nút có địa chỉ DelNode mà chúng ta sẽ hủy nút có địa chỉ của phần tử thế mạng là nút phải nhất trong cây con trái của DelNode (MRNode), hoặc là nút trái nhất trong cây con phải của DelNode (MLNode). Sau khi chuyển toàn bộ nội dung dữ liệu của nút thế mạng cho DelNode (DelNode→Key = MRNode->Key hoặc DelNode->Key = MLNode->Key) thì chúng ta sẽ hủy nút thế mạng như đối với trường hợp c_1) và c_2) ở trên.

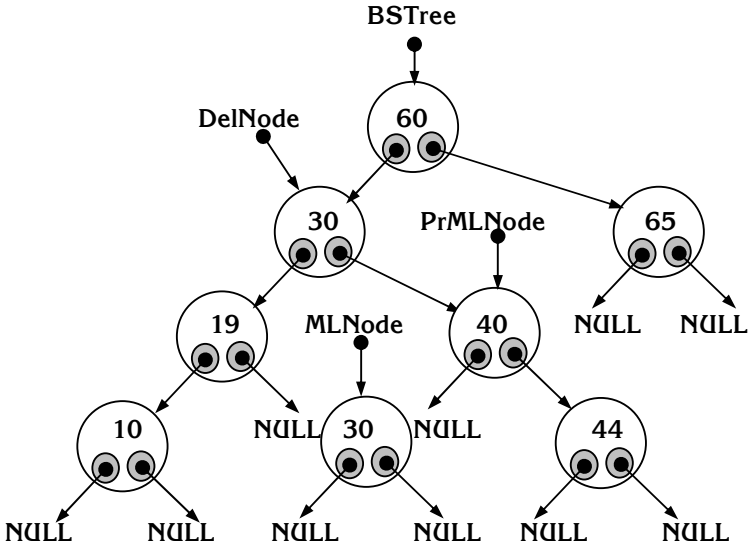
Ví dụ: Giả sử cần hủy nút có Key = 25 (DelData = 25). Chúng ta sẽ chọn phần tử thế mạng MLNode là nút trái nhất trong cây con phải của DelNode (trái nhất trong DelNode->BST_Right) để hủy,



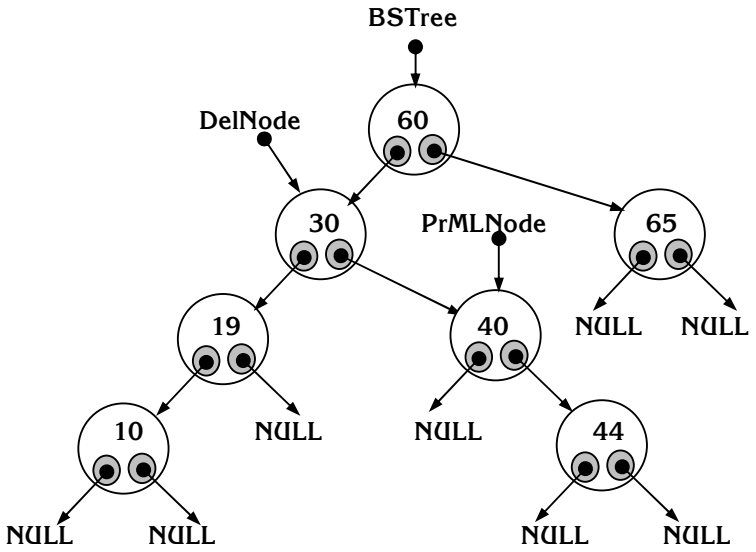
Chuyển dữ liệu trong MLNode về cho DelNode: DelNode->Key = MLNode->Key



Tiến hành hủy MLNode (hủy nút lá): PrMLNode->BST_Left = NULL



Kết quả sau khi hủy:



- Thuật toán hủy 1 nút trong cây nhị phân tìm kiếm bằng phương pháp chuyển cây con phải của nút cần hủy về thành cây con phải của cây con có nút gốc là nút phải nhất trong cây con trái của nút cần hủy (nếu nút cần hủy có đủ 02 cây con):

// Tìm nút cần hủy và nút cha của nút cần hủy
B1: DelNode = BSTree
B2: PrDelNode = NULL
B3: IF (DelNode = NULL)
    Thực hiện Bkt
B4: IF (DelNode->Key = DelData)
    Thực hiện B8
B5: IF (DelNode->Key > DelData) // Chuyển sang cây con trái
    B5.1: PrDelNode = DelNode
    B5.2: DelNode = DelNode->BST_Left
    B5.3: OnTheLeft = True
    B5.4: Thực hiện B7
B6: IF (DelNode->Key < DelData) // Chuyển sang cây con phải
    B6.1: PrDelNode = DelNode
    B6.2: DelNode = DelNode->BST_Right
    B6.3: OnTheLeft = False
    B6.4: Thực hiện B7
B7: Lặp lại B3

// Chuyển các mối quan hệ của DelNode cho các nút khác
B8: IF (PrDelNode = NULL) // DelNode là nút gốc

    // Nếu DelNode là nút lá
    B8.1: If (DelNode->BST_Left = NULL) and (DelNode->BST_Right = NULL)
        B8.1.1: BSTree = NULL
        B8.1.2: Thực hiện B10

    // Nếu DelNode có một cây con phải
    B8.2: If (DelNode->BST_Left = NULL) and (DelNode->BST_Right != NULL)
        B8.2.1: BSTree = BSTree->BST_Right

B8.2.2: DelNode->BST_Right = NULL
B8.2.3: Thực hiện B10

// Nếu DelNode có một cây con trái
B8.3: If (DelNode->BST_Left != NULL) and (DelNode->BST_Right = NULL)
B8.3.1: BSTree = BSTree->BST_Left
B8.3.2: DelNode->BST_Left = NULL
B8.3.3: Thực hiện B10

// Nếu DelNode có hai cây con
B8.4: If (DelNode->BST_Left != NULL) and (DelNode->BST_Right != NULL)

// Tìm nút phải nhất trong cây con trái của DelNode
B8.4.1: MRNode = DelNode->BST_Left
B8.4.2: if (MRNode->BST_Right = NULL)
Thực hiện B8.4.5
B8.4.3: MRNode = MRNode->BST_Right
B8.4.4: Lặp lại B8.4.2

// Chuyển cây con phải của DelNode về cây con phải của MRNode
B8.4.5: MRNode->BST_Right = DelNode->BST_Right
B8.4.6: DelNode->BST_Right = NULL

// Chuyển cây con trái còn lại của DelNode về cho BSTree
B8.4.7: BSTree = BSTree->BST_Left
B8.4.8: DelNode->BST_Left = NULL
B8.4.9: Thực hiện B10

B9: ELSE // DelNode không phải là nút gốc

// Nếu DelNode là nút lá
B9.1: If (DelNode->BST_Left = NULL) and (DelNode->BST_Right = NULL)

// DelNode là cây con trái của PrDelNode
B9.1.1: if (OnTheLeft = True)
PrDelNode->BST_Left = NULL
B9.1.2: else // DelNode là cây con phải của PrDelNode
PrDelNode->BST_Right = NULL
B9.1.3: Thực hiện B10

// Nếu DelNode có một cây con phải
B9.2: If (DelNode->BST_Left = NULL) and (DelNode->BST_Right != NULL)
B9.2.1: if (OnTheLeft = True)
PrDelNode->BST_Left = DelNode->BST_Right
B9.2.2: else
PrDelNode->BST_Right = DelNode->BST_Right
B9.2.3: DelNode->BST_Right = NULL
B9.2.4: Thực hiện B10

// Nếu DelNode có một cây con trái
B9.3: If (DelNode->BST_Left != NULL) and (DelNode->BST_Right = NULL)
B9.3.1: if (OnTheLeft = True)
PrDelNode->BST_Left = DelNode->BST_Left

B9.3.2: else
PrDelNode->BST_Right = DelNode->BST_Left
B9.3.3: DelNode->BST_Left = NULL
B9.3.4: Thực hiện B10

// Nếu DelNode có hai cây con
B9.4: If (DelNode->BST_Left != NULL) and (DelNode->BST_Right != NULL)

// Tìm nút phải nhất trong cây con trái của DelNode
B9.4.1: MRNode = DelNode->BST_Left
B9.4.2: if (MRNode->BST_Right = NULL)
Thực hiện B9.4.5
B9.4.3: MRNode = MRNode->BST_Right
B9.4.4: Lặp lại B9.4.2

// Chuyển cây con phải DelNode về thành cây con phải MRNode
B9.4.5: MRNode->BST_Right = DelNode->BST_Right
B9.4.6: DelNode->BST_Right = NULL

// Chuyển cây con trái còn lại của DelNode về cho PrDelNode
B9.4.7: if (OnTheLeft = True)
PrDelNode->BST_Left = DelNode->BST_Left
B9.4.8: else
PrDelNode->BST_Right = DelNode->BST_Left
B9.4.9: DelNode->BST_Left = NULL
B9.4.10: Thực hiện B10

// Hủy DelNode
B10: delete DelNode
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm BST_Delete_Node_TRS có prototype:

int BST_Delete_Node_TRS(BST_Type &BS_Tree, T DelData);

Hàm thực hiện việc hủy nút có thành phần Key là DelData trên cây nhị phân tìm kiếm BS_Tree bằng phương pháp chuyển cây con phải của nút cần hủy về thành cây con phải của cây có nút gốc là nút phải nhất trong cây con trái của nút cần hủy (nếu nút cần hủy có hai cây con). Hàm trả về giá trị 1 nếu việc hủy thành công (có nút để hủy), trong trường hợp ngược lại hàm trả về giá trị 0 (không tồn tại nút có Key là DelData hoặc cây rỗng).

```
int BST_Delete_Node_TRS(BST_Type &BS_Tree, T DelData)
  { BST_Type DelNode = BS_Tree;
    BST_Type PrDelNode = NULL;
    int OnTheLeft = 0;
    while (DelNode != NULL)
      { if (DelNode->Key == DelData)
          break;
        PrDelNode = DelNode;
        if (DelNode->Key > DelData)
```

```
            {  DelNode = DelNode->BST_Left;
               OnTheLeft = 1;
            }
         else // (DelNode->Key < DelData)
            {  DelNode = DelNode->BST_Right;
               OnTheLeft = 0;
            }
      }
   if (DelNode == NULL)  // Không có nút để hủy
      return (0);
   if (PrDelNode == NULL)         // DelNode là nút gốc
      {  if (DelNode->BST_Left == NULL && DelNode->BST_Right == NULL)
            BS_Tree = NULL;
         else
            if (DelNode->BST_Left == NULL)    // DelNode có 1 cây con phải
               {  BS_Tree = BS_Tree->BST_Right;
                  DelNode->BST_Right = NULL;
               }
            else
               if (DelNode->BST_Right == NULL) // DelNode có 1 cây con trái
                  {  BS_Tree = BS_Tree->BST_Left;
                     DelNode->BST_Left = NULL;
                  }
               else   // DelNode có hai cây con
                  {  BST_Type  MRNode = DelNode->BST_Left;
                     while (MRNode->BST_Right != NULL)
                        MRNode = MRNode->BST_Right;
                     MRNode->BST_Right = DelNode->BST_Right;
                     DelNode->BST_Right = NULL;
                     BS_Tree = BS_Tree->BST_Left;
                     DelNode->BST_Left = NULL;
                  }
      }
   else        // DelNode là nút trung gian
      {  if (DelNode->BST_Left == NULL && DelNode->BST_Right == NULL)
            if (OnTheLeft == 1)
               PrDelNode->BST_Left = NULL;
            else
               PrDelNode->BST_Right = NULL;
         else
            if (DelNode->BST_Left == NULL)    // DelNode có 1 cây con phải
               {  if (OnTheLeft == 1)
                     PrDelNode->BST_Left = DelNode->BST_Right;
                  else
                     PrDelNode->BST_Right = DelNode->BST_Right;
                  DelNode->BST_Right = NULL;
               }
            else
```

```
                if (DelNode->BST_Right == NULL) // DelNode có 1 cây con trái
                   {  if (OnTheLeft == 1)
                         PrDelNode->BST_Left = DelNode->BST_Left;
                      else
                         PrDelNode->BST_Right = DelNode->BST_Left;
                      DelNode->BST_Left = NULL;
                   }
                else  // DelNode có hai cây con
                   {  BST_Type  MRNode = DelNode->BST_Left;
                      while (MRNode->BST_Right != NULL)
                         MRNode = MRNode->BST_Right;
                      MRNode->BST_Right = DelNode->BST_Right;
                      DelNode->BST_Right = NULL;
                      if (OnTheLeft == 1)
                         PrDelNode->BST_Left = DelNode->BST_Left;
                      else
                         PrDelNode->BST_Right = DelNode->BST_Left;
                      DelNode->BST_Left = NULL;
                   }
             }
          delete  DelNode;
          return (1);
       }
```

- **Thuật toán hủy 1 nút trong cây nhị phân tìm kiếm bằng phương pháp hủy phần tử thế mạng là phần tử trái nhất trong cây con phải của nút cần hủy (nếu nút cần hủy có đủ 02 cây con):**

// Tìm nút cần hủy và nút cha của nút cần hủy
B1: DelNode = BSTree
B2: PrDelNode = NULL
B3: IF (DelNode = NULL)
    Thực hiện Bkt
B4: IF (DelNode->Key = DelData)
    Thực hiện B8
B5: IF (DelNode->Key > DelData) // Chuyển sang cây con trái
    B5.1: PrDelNode = DelNode
    B5.2: DelNode = DelNode->BST_Left
    B5.3: OnTheLeft = True
    B5.4: Thực hiện B7
B6: IF (DelNode->Key < DelData) // Chuyển sang cây con phải
    B6.1: PrDelNode = DelNode
    B6.2: DelNode = DelNode->BST_Right
    B6.3: OnTheLeft = False
    B6.4: Thực hiện B7
B7: Lặp lại B3

// Chuyển các mối quan hệ của DelNode cho các nút khác
B8: IF (PrDelNode = NULL) // DelNode là nút gốc

```
        // Nếu DelNode là nút lá
        B8.1: If (DelNode->BST_Left = NULL) and (DelNode->BST_Right = NULL)
              B8.1.1: BSTree = NULL
              B8.1.2: Thực hiện B11

        // Nếu DelNode có một cây con phải
        B8.2: If (DelNode->BST_Left = NULL) and (DelNode->BST_Right != NULL)
              B8.2.1: BSTree = BSTree->BST_Right
              B8.2.2: DelNode->BST_Right = NULL
              B8.2.3: Thực hiện B11

        // Nếu DelNode có một cây con trái
        B8.3: If (DelNode->BST_Left != NULL) and (DelNode->BST_Right = NULL)
              B8.3.1: BSTree = BSTree->BST_Left
              B8.3.2: DelNode->BST_Left = NULL
              B8.3.3: Thực hiện B11

    B9: ELSE      // DelNode không phải là nút gốc

        // Nếu DelNode là nút lá
        B9.1: If (DelNode->BST_Left = NULL) and (DelNode->BST_Right = NULL)

              // DelNode là cây con trái của PrDelNode
              B9.1.1: if (OnTheLeft = True)
                    PrDelNode->BST_Left = NULL
              B9.1.2: else      // DelNode là cây con phải của PrDelNode
                    PrDelNode->BST_Right = NULL
              B9.1.3: Thực hiện B11

        // Nếu DelNode có một cây con phải
        B9.2: If (DelNode->BST_Left = NULL) and (DelNode->BST_Right != NULL)
              B9.2.1: if (OnTheLeft = True)
                    PrDelNode->BST_Left = DelNode->BST_Right
              B9.2.2: else
                    PrDelNode->BST_Right = DelNode->BST_Right
              B9.2.3: DelNode->BST_Right = NULL
              B9.2.4: Thực hiện B11

        // Nếu DelNode có một cây con trái
        B9.3: If (DelNode->BST_Left != NULL) and (DelNode->BST_Right = NULL)
              B9.3.1: if (OnTheLeft = True)
                    PrDelNode->BST_Left = DelNode->BST_Left
              B9.3.2: else
                    PrDelNode->BST_Right = DelNode->BST_Left
              B9.3.3: DelNode->BST_Left = NULL
              B9.3.4: Thực hiện B11

    // Nếu DelNode có hai cây con
    B10: If (DelNode->BST_Left != NULL) and (DelNode->BST_Right != NULL)

        // Tìm nút trái nhất trong cây con phải của DelNode và nút cha của nó
        B10.1: MLNode = DelNode->BST_Right
        B10.2: PrMLNode = DelNode
```

B10.3: if (MLNode->BST_Left = NULL)
Thực hiện B10.7
B10.4: PrMLNode = MLNode
B10.5: MLNode = MLNode->BST_Left
B10.6: Lặp lại B10.3

// Chép dữ liệu từ MLNode về DelNode
B10.7: DelNode->Key = MLNode->Key

// Chuyển cây con phải của MLNode về cây con trái của PrMLNode
B10.8: if (PrMLNode = DelNode) // MLNode là nút phải của PrMLNode
PrMLNode->BST_Right = MLNode->BST_Right
B10.9: else // MLNode là nút trái của PrMLNode
PrMLNode->BST_Left = MLNode->BST_Right
B10.10: MLNode->BST_Right = NULL

// Chuyển vai trò của MLNode cho DelNode
B10.11: DelNode = MLNode
B10.12: Thực hiện B11

// Hủy DelNode
B11: delete DelNode
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm BST_Delete_Node_SB có prototype:

```
int BST_Delete_Node_SB(BST_Type &BS_Tree, T DelData);
```

Hàm thực hiện việc hủy nút có thành phần Key là DelData trên cây nhị phân tìm kiếm BS_Tree bằng phương pháp hủy phần tử thế mạng là phần tử trái nhất trong cây con phải của nút cần hủy (nếu nút cần hủy có hai cây con). Hàm trả về giá trị 1 nếu việc hủy thành công (có nút để hủy), trong trường hợp ngược lại hàm trả về giá trị 0 (không tồn tại nút có Key là DelData hoặc cây rỗng).

```
int BST_Delete_Node_SB(BST_Type &BS_Tree, T DelData)
{ BST_Type DelNode = BS_Tree;
  BST_Type PrDelNode = NULL;
  int OnTheLeft = 0;
  while (DelNode != NULL)
    { if (DelNode->Key == DelData)
        break;
      PrDelNode = DelNode;
      if (DelNode->Key > DelData)
        { DelNode = DelNode->BST_Left;
          OnTheLeft = 1;
        }
      else // (DelNode->Key < DelData)
        { DelNode = DelNode->BST_Right;
          OnTheLeft = 0;
        }
    }
```

```
if (DelNode == NULL)  // Không có nút để hủy
   return (0);
if (PrDelNode == NULL)        // DelNode là nút gốc
   {  if (DelNode->BST_Left == NULL && DelNode->BST_Right == NULL)
         BS_Tree = NULL;
      else
         if (DelNode->BST_Left == NULL)    // DelNode có 1 cây con phải
            {  BS_Tree = BS_Tree->BST_Right;
               DelNode->BST_Right = NULL;
            }
         else
            if (DelNode->BST_Right == NULL) // DelNode có 1 cây con trái
               {  BS_Tree = BS_Tree->BST_Left;
                  DelNode->BST_Left = NULL;
               }
   }
else       // DelNode là nút trung gian
   {  if (DelNode->BST_Left == NULL && DelNode->BST_Right == NULL)
         if (OnTheLeft == 1)
            PrDelNode->BST_Left = NULL;
         else
            PrDelNode->BST_Right = NULL;
      else
         if (DelNode->BST_Left == NULL)    // DelNode có 1 cây con phải
            {  if (OnTheLeft == 1)
                  PrDelNode->BST_Left = DelNode->BST_Right;
               else
                  PrDelNode->BST_Right = DelNode->BST_Right;
               DelNode->BST_Right = NULL;
            }
         else
            if (DelNode->BST_Right == NULL) // DelNode có 1 cây con trái
               {  if (OnTheLeft == 1)
                     PrDelNode->BST_Left = DelNode->BST_Left;
                  else
                     PrDelNode->BST_Right = DelNode->BST_Left;
                  DelNode->BST_Left = NULL;
               }
   }
// DelNode có hai cây con
if (DelNode->BST_Left != NULL && DelNode->BST_Right != NULL)
   {  BST_Type  MLNode = DelNode->BST_Right;
      BST_Type  PrMLNode = DelNode;
      while (MLNode->BST_Left != NULL)
         {  PrMLNode = MLNode;
            MLNode = MLNode->BST_Left;
         }
      DelNode->Key = MLNode->Key;
```

```
            if (PrMLNode == DelNode)
               PrMLNode->BST_Right = MLNode->BST_Right;
            else
               PrMLNode->BST_Left = MLNode->BST_Right;
            MLNode->BST_Right = NULL;
            DelNode = MLNode;
         }
      delete  DelNode;
      return (1);
   }
```

d. Hủy toàn bộ cây:

Thao tác chỉ đơn giản là việc thực hiện nhiều lần thao tác hủy một nút trên cây nhị phân tìm kiếm cho đến khi cây trở thành rỗng.

Hàm BST_Delete có prototype:

```
void  BST_Delete(BST_Type  &BS_Tree);
```

Hàm thực hiện việc hủy tất cả các nút trong cây nhị phân tìm kiếm BS_Tree.

```
void  BST_Delete(BST_Type  &BS_Tree)
   {  BST_Type  DelNode = BS_Tree;
      while (BST_Delete_Node_TRS(BS_Tree, DelNode->Key) == 1)
         DelNode = BS_Tree;
      return;
   }
```

5.3. Cây cân bằng (Balanced Tree)

5.3.1. Định nghĩa – Cấu trúc dữ liệu

a. Định nghĩa:

- Cây cân bằng tương đối:

Theo Adelson-Velskii và Landis đưa ra định nghĩa về cây cân bằng tương đối như sau:

Cây cân bằng tương đối là một cây nhị phân thỏa mãn điều kiện là đối với mọi nút của cây thì chiều cao của cây con trái và chiều cao của cây con phải của nút đó hơn kém nhau không quá 1.

Cây cân bằng tương đối còn được gọi là cây AVL (AVL tree).

- Cây cân bằng hoàn toàn:

Cây cân bằng hoàn toàn là một cây nhị phân thỏa mãn điều kiện là đối với mọi nút của cây thì số nút ở cây con trái và số nút ở cây con phải của nút đó hơn kém nhau không quá 1.

Như vậy, một cây cân bằng hoàn toàn chắc chắn là một cây cân bằng tương đối.

b. Cấu trúc dữ liệu của cây cân bằng:

Để ghi nhận mức độ cân bằng tại mỗi nút gốc cây con chúng ta sử dụng thêm một thành phần Bal trong cấu trúc dữ liệu của mỗi nút. Do vậy, cấu trúc dữ liệu của cây nhị phân tìm kiếm cân bằng tương đối và cây nhị phân tìm kiếm cân bằng hoàn toàn nói riêng và của cây cân bằng nói chung tương tự như cấu trúc dữ liệu của cây nhị phân ngoại trừ trong đó chúng ta đưa thêm thành phần Bal làm chỉ số cân bằng tại mỗi nút như sau:

```
typedef  struct  BAL_Node
  {  T  Key;
     int  Bal; // Chỉ số cân bằng tại nút gốc cây con
     BAL_Node * BAL_Left; // Vùng liên kết quản lý địa chỉ nút gốc cây con trái
     BAL_Node * BAL_Right; // Vùng liên kết quản lý địa chỉ nút gốc cây con phải
  }    BAL_OneNode;

typedef  BAL_OneNode  *  BAL_Type;
```

Để quản lý các cây nhị phân tìm kiếm cân bằng chúng ta chỉ cần quản lý địa chỉ nút gốc của cây:

```
BAL_Type  BALTree;
```

Giá trị chỉ số cân bằng Bal tại một nút gốc cây con trong cây cân bằng tương đối bằng hiệu số giữa chiều cao cây con trái và chiều cao cây con phải của nút đó.

Giá trị chỉ số cân bằng Bal tại một nút gốc cây con trong cây cân bằng hoàn toàn bằng hiệu số giữa số nút ở cây con trái và số nút ở cây con phải của nút đó.

Như vậy, nếu tại mọi nút trong cây nhị phân mà thỏa mãn điều kiện $-1 \le Bal \le 1$ thì cây là cây cân bằng và phạm vi từ –1 đến +1 là phạm vi cho phép của chỉ số cân bằng Bal:

+ Nếu Bal = 0: cây con trái và cây con phải đều nhau

+ Nếu Bal = -1: cây con trái nhỏ hơn (thấp hơn) cây con phải (lệch phải)

+ Nếu Bal = +1: cây con trái lớn hơn (cao hơn) cây con phải (lệch trái)

5.3.2. Các thao tác

Trong phạm vi của phần này chúng ta xem xét các thao tác trên cây nhị phân tìm kiếm cân bằng tương đối. Các thao tác trên cây cân bằng hoàn toàn sinh viên tự vận dụng tương tự. Do vậy, khi trình bày các thao tác mà nói tới cây cân bằng nghĩa là cây nhị phân tìm kiếm cân bằng và chúng ta cũng chỉ xét cây nhị phân tìm kiếm trong trường hợp không trùng khóa nhận diện.

Trong các thao tác trên cây nhị phân tìm kiếm cân bằng tương đối thì có hai thao tác Thêm một nút vào cây và Hủy một nút khỏi cây là hai thao tác khá phức tạp vì có nguy cơ phá vỡ sự cân bằng của cây, khi đó chúng ta phải thực hiện việc cân bằng lại cây. Các thao tác khác hoàn toàn tương tự như trong cây nhị phân nói chung và cây nhị phân tìm kiếm nói riêng. Do vậy, trong phần này chúng ta chỉ trình bày hai thao tác này mà thôi.

a. Thêm một nút vào cây cân bằng:

Giả sử chúng ta cần thêm một nút NewNode có thành phần dữ liệu là NewData vào trong cây cân bằng BALTree sao cho sau khi thêm BALTree vẫn là một cây cân bằng. Để thực hiện điều này trước hết chúng ta tìm kiếm vị trí của nút cần thêm là nút con trái hoặc nút con phải của một nút PrNewNode tương tự như trong cây nhị phân tìm kiếm. Sau khi thêm NewNode vào cây con trái hoặc cây con phải của PrNewNode thì chỉ số cân bằng của các nút từ PrNewNode trở về các nút trước sẽ bị thay đổi dây chuyền và chúng ta phải lần ngược từ PrNewNode về theo các nút trước để theo dõi sự thay đổi này. Nếu phát hiện tại một nút AncestorNode có sự thay đổi vượt quá phạm vi cho phép (bằng –2 hoặc +2) thì chúng ta tiến hành cân bằng lại cây ngay tại nút AncestorNode này.

Việc cân bằng lại cây tại nút AncestorNode được tiến hành cụ thể theo các trường hợp như sau:

Trường hợp 1: Nếu AncestorNode->Bal = -2:

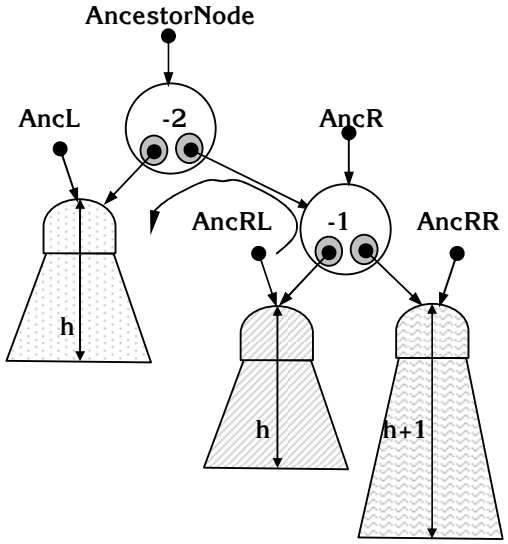
Gọi: AncL = AncestorNode->BAL_Left
AncR = AncestorNode->BAL_Right

⇒ AncL có chiều cao là h và AncR có chiều cao là h+2 (h ≥ 0)
⇒ Có ít nhất 1 cây con của AncR có chiều cao là h+1

Gọi: AncRL = AncR->BAL_Left
AncRR = AncR->BAL_Right

⇒ Cây con có nút gốc AncestorNode có thể ở vào một trong ba dạng sau:

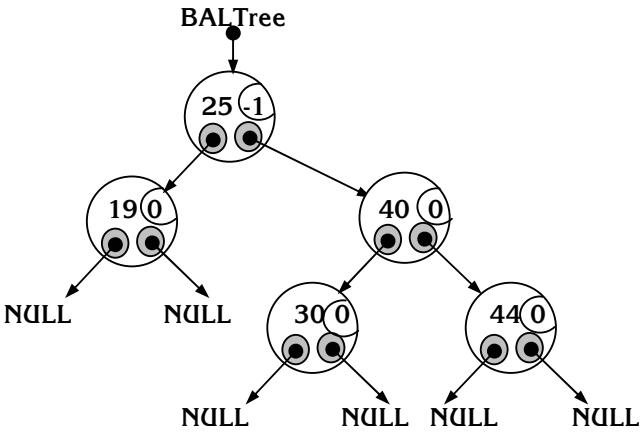
a_1) AncRL có chiều cao là h và AncRR có chiều cao là h+1 (AncR->Bal = -1)



Để cân bằng lại AncestorNode chúng ta thực hiện việc quay đơn cây con phải AncR của nút này lên thành nút gốc; chuyển AncestorNode thành nút con trái của nút gốc và AncestorNode có hai cây con là AncL và AncRL (BAL_Right Rotation).

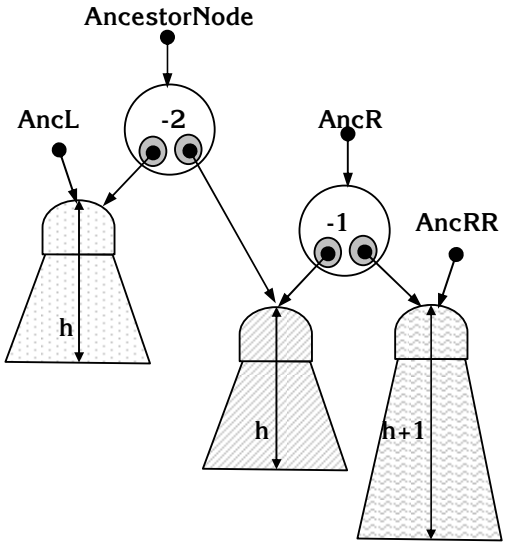
Cây con AncestorNode sau khi quay cây con phải AncR sẽ là một cây cân bằng.

Ví dụ: Việc thêm nút có Key = 50 vào cây nhị phân tìm kiếm cân bằng sau đây sẽ làm cho cây mất cân bằng và chúng ta phải cân bằng lại theo trường hợp này:

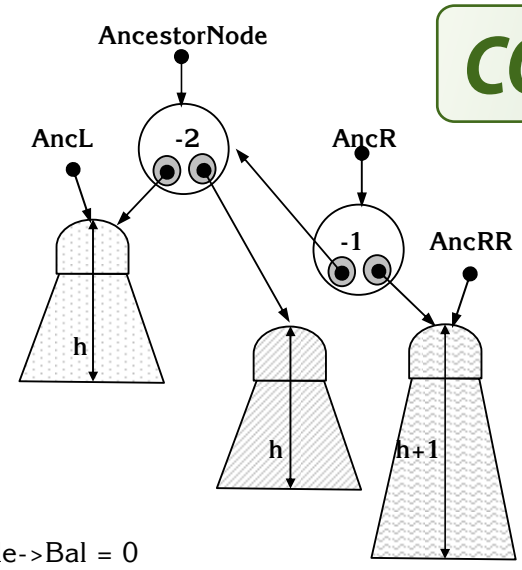


Để thực hiện cân bằng lại bằng phép quay đơn này chúng ta thực hiện các bước sau:

B1: AncestorNode->BAL_Right = AncR->BAL_Left



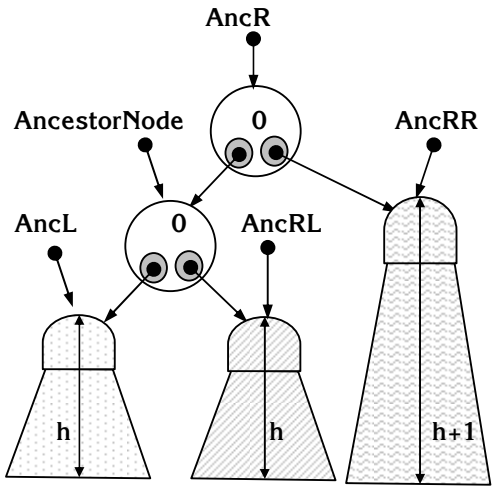
B2: AncR->BAL_Left = AncestorNode



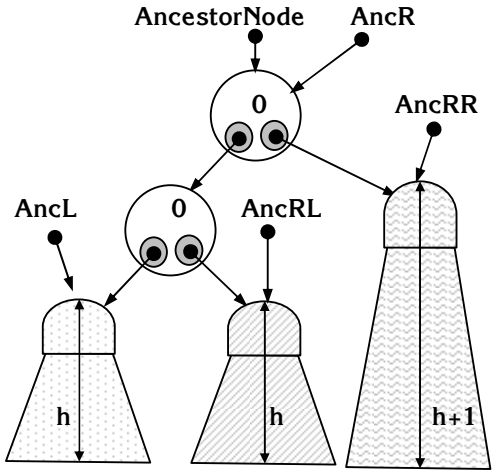
COMPLETED

B3: AncR->Bal = AncestorNode->Bal = 0

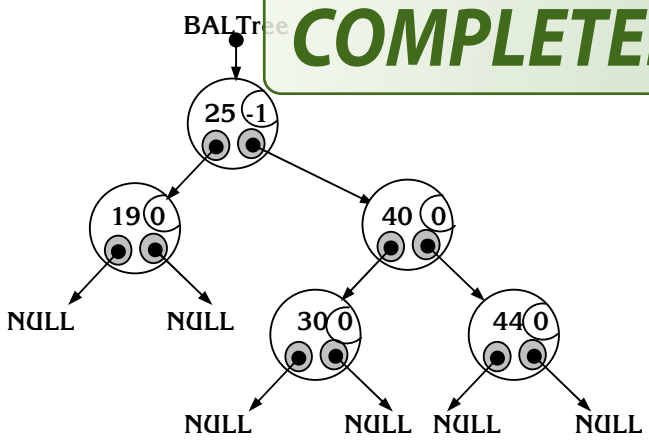
Việc quay kết thúc, cây trở thành cây cân bằng.



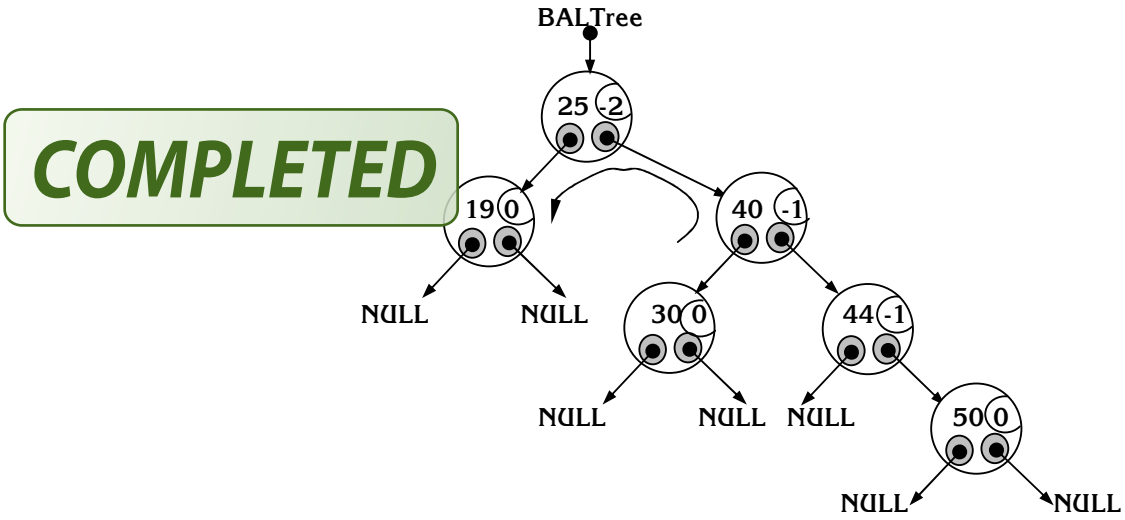
Chuyển vai trò của AncR cho AncestorNode: AncestorNode = AncR
Kết quả sau phép quay:



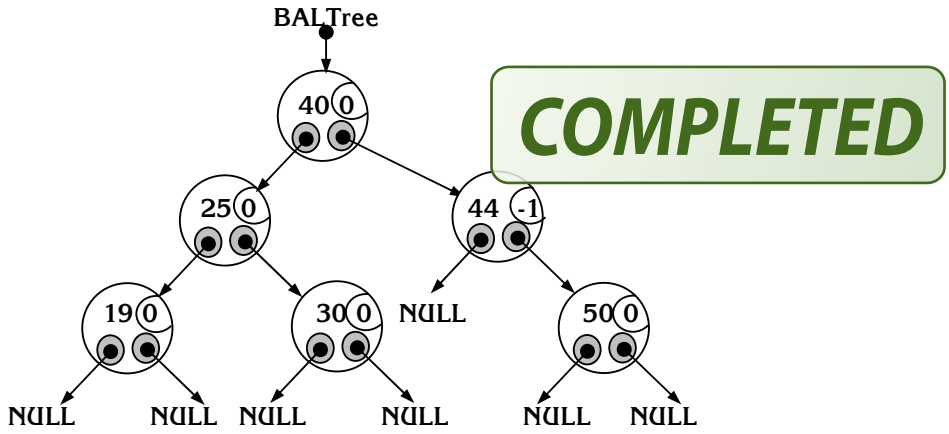
Ví dụ: Thêm nút có Key = 50 vào cây nhị phân tìm kiếm cân bằng sau đây:



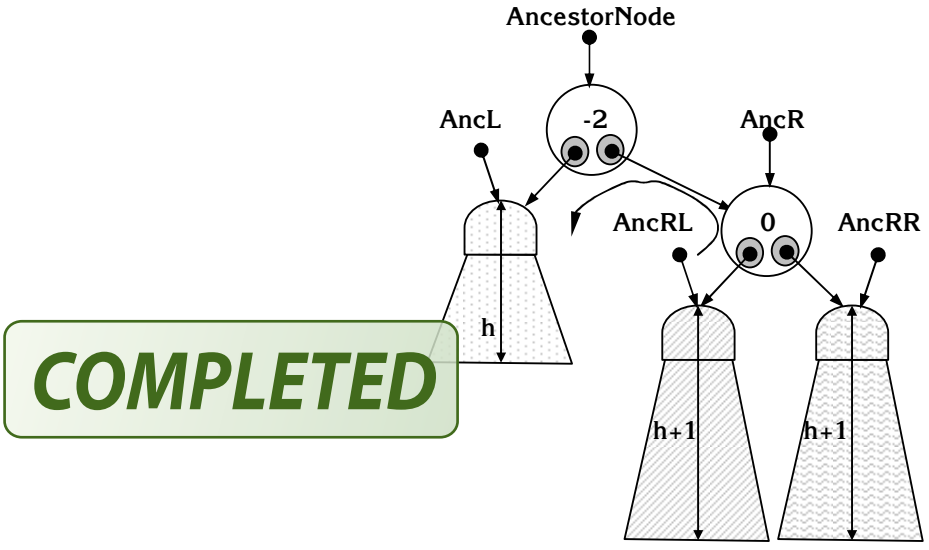
Cây nhị phân tìm kiếm cân bằng sau khi thêm nút có Key = 50 như sau:



Thực hiện quay cây con phải của BALTree, cây nhị phân tìm kiếm sau khi quay trở thành cây nhị phân tìm kiếm cân bằng như sau:

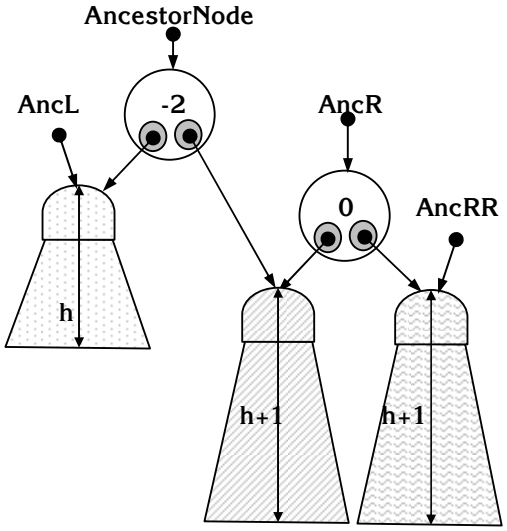


b_1) <u>AncRL và AncRR đều có chiều cao là h+1 (AncR->Bal = 0)</u>



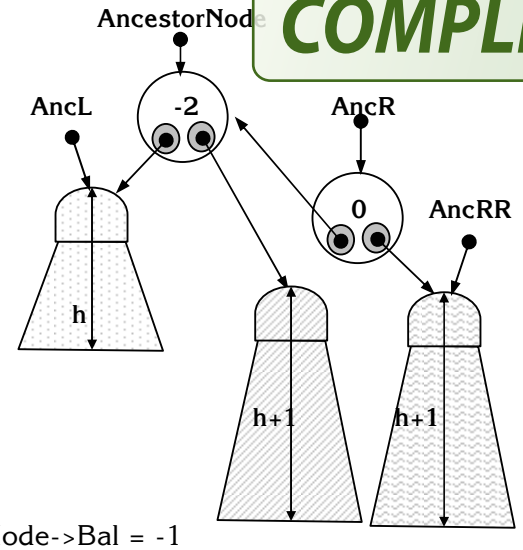
Việc bằng lại được thực hiện tương tự như trường hợp a_1) ở trên:

B1: AncestorNode->BAL_Right = AncR->BAL_Left



B2: AncR->BAL_Left = AncestorNode

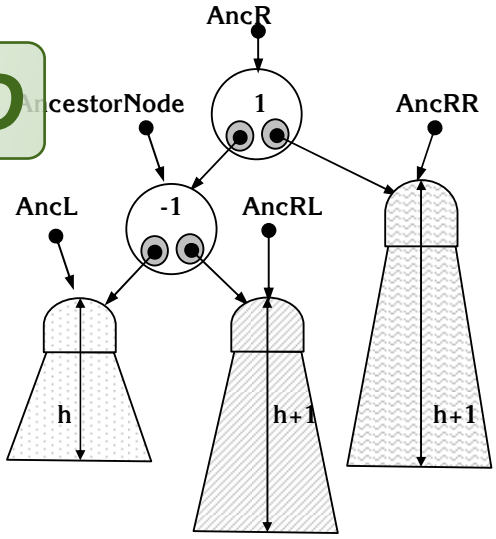
COMPLETED



B3: AncR->Bal = 1, AncestorNode->Bal = -1

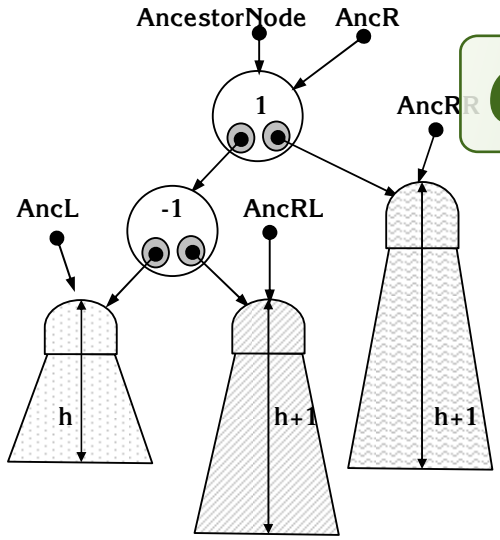
Việc quay kết thúc, cây trở thành cây cân bằng.

COMPLETED

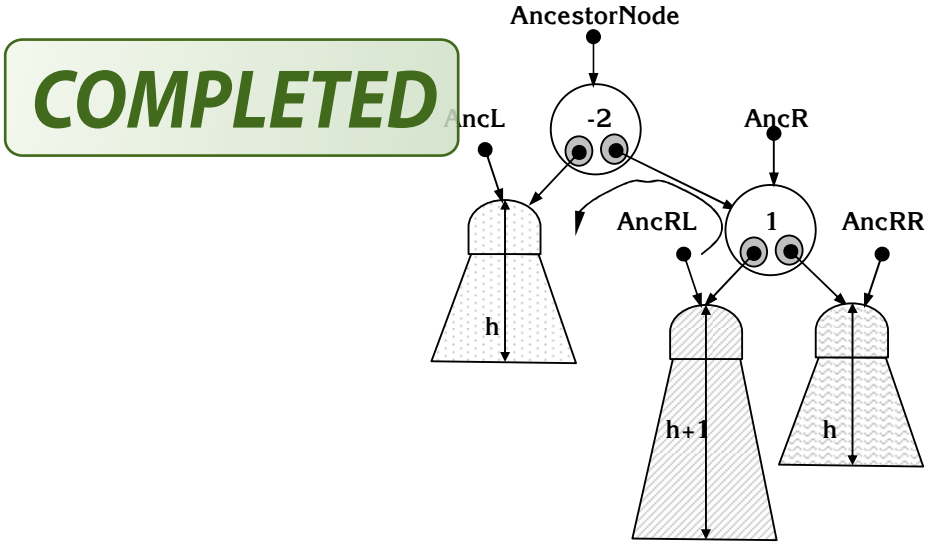


Chuyển vai trò của AncR cho AncestorNode: AncestorNode = AncR

Kết quả sau phép quay:

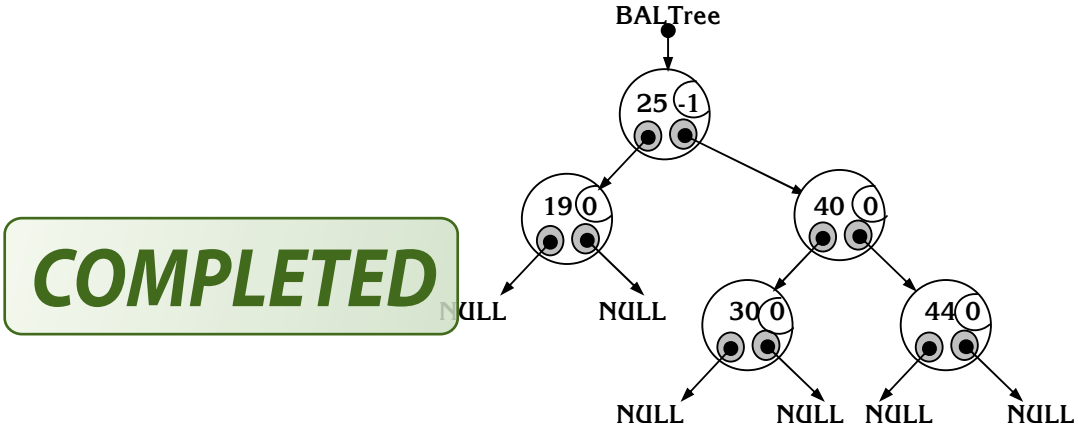


c_1) <u>AncRL có chiều cao là h+1 và AncRR có chiều cao là h (AncR->Bal = 1)</u>



Để cân bằng lại AncestorNode chúng ta thực hiện việc quay kép: quay cây con trái AncRL và quay cây con phải AncR (Double Rotation).

<u>Ví dụ</u>: Việc thêm nút có Key = 27 vào cây nhị phân tìm kiếm cân bằng sau đây sẽ làm cho cây mất cân bằng và chúng ta phải cân bằng lại theo trường hợp này:



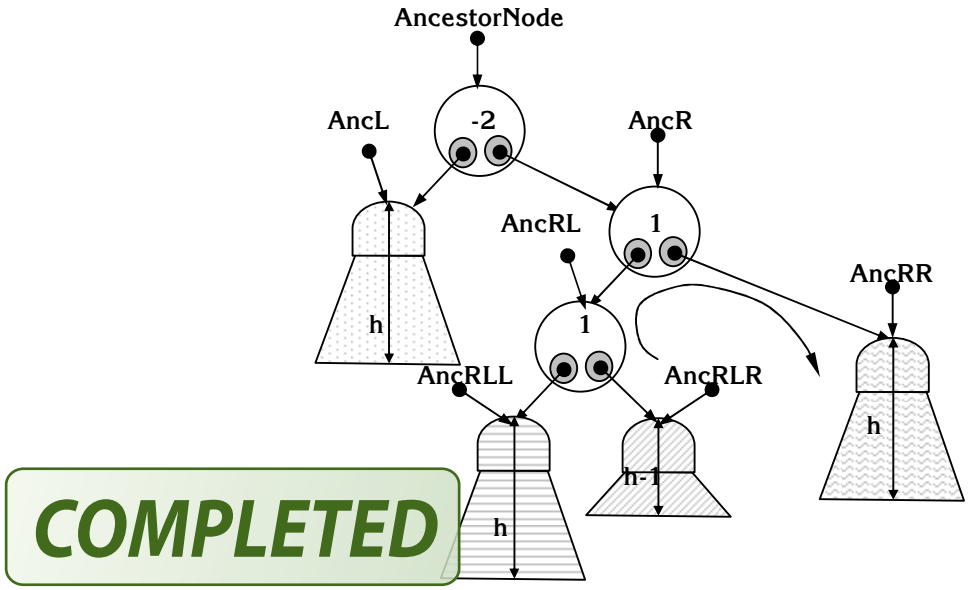
Việc quay được tiến hành cụ thể như sau:

Gọi: AncRLL = AncRL->BAL_Left
AncRLR = AncRL->BAL_Right

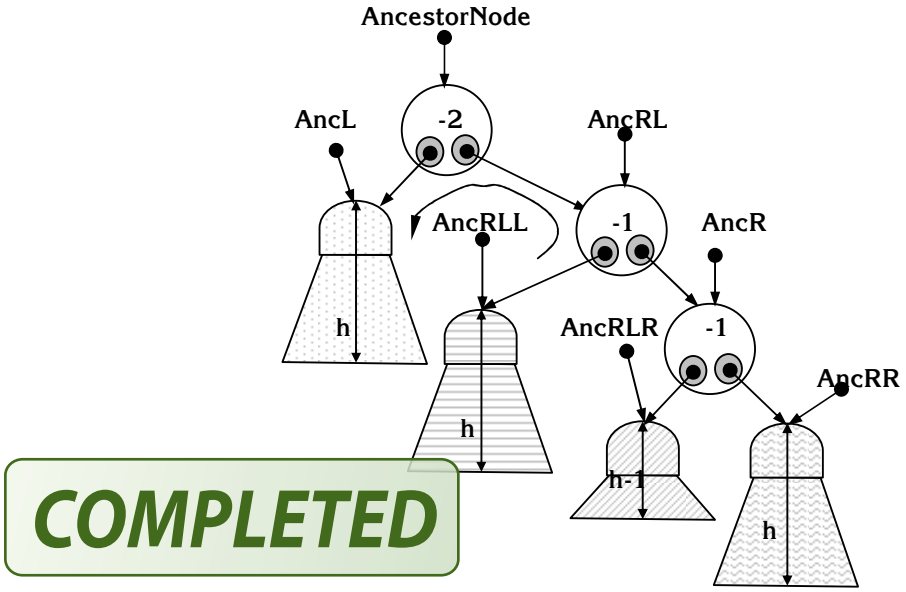
⇒ AncRLL và AncRLR có chiều cao tối đa là h

⇒ Cây con có nút gốc AncestorNode có thể ở vào một trong ba dạng sau:

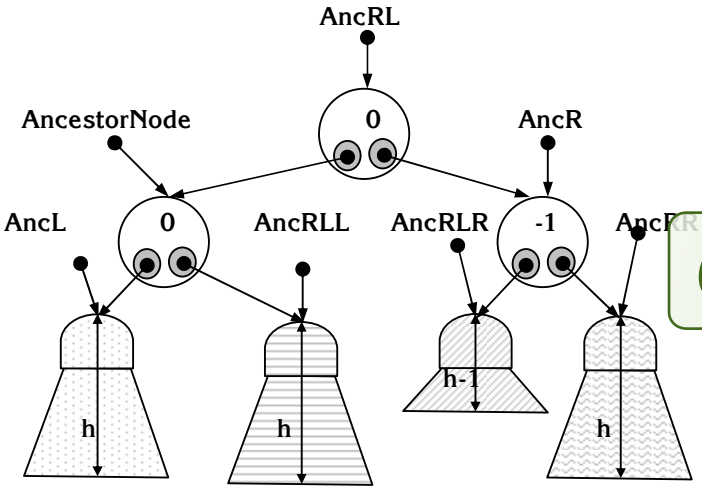
- AncRLL có chiều cao là h và AncRLR có chiều cao là h-1 (AncRL->Bal = 1; h ≥ 1)



Để cân bằng lại AncestorNode đầu tiên chúng ta thực hiện việc quay đơn cây con trái AncRL của AncR lên thành nút gốc cây con phải của AncestorNode, chuyển AncR nút thành nút gốc cây con phải của AncRL và chuyển AncRLR thành nút gốc cây con trái của AncR. Sau khi quay cây sẽ trở thành:

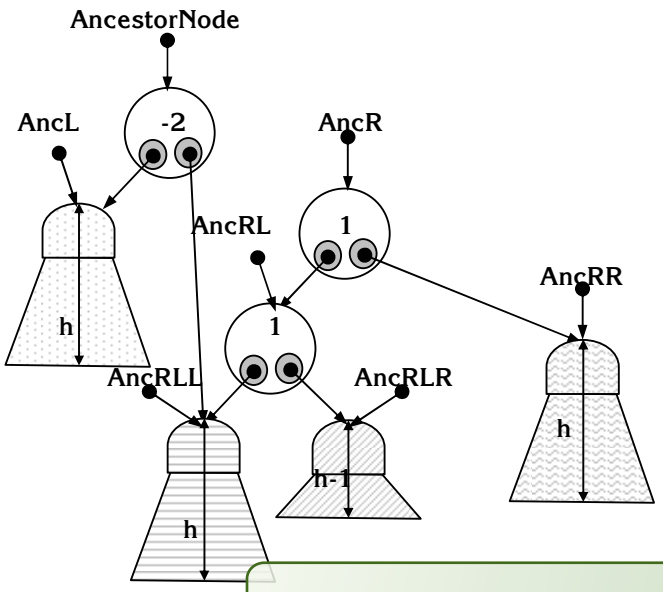


Bây giờ chúng ta tiếp tục thực hiện việc quay đơn cây con phải AncRL của AncestorNode lên thành nút gốc và chuyển AncRLL nút thành nút gốc cây con phải của AncestorNode. Sau khi quay cây sẽ trở nên cân bằng:

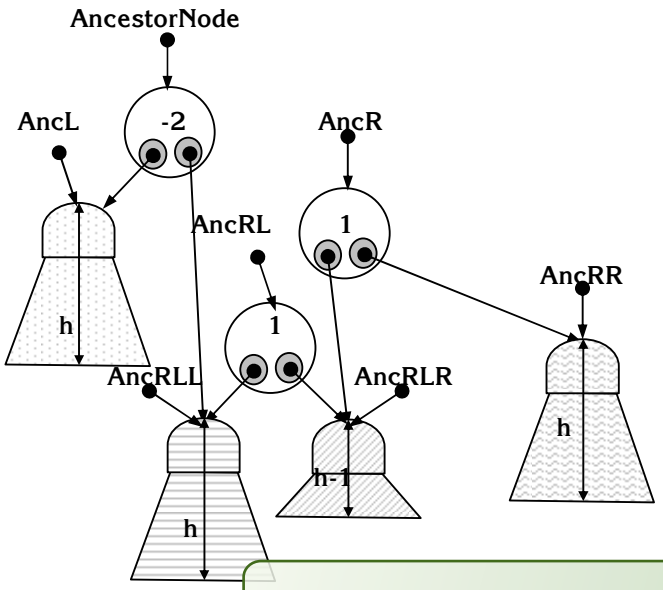


Như vậy để thực hiện quá trình quay kép này chúng ta thực hiện các bước sau:

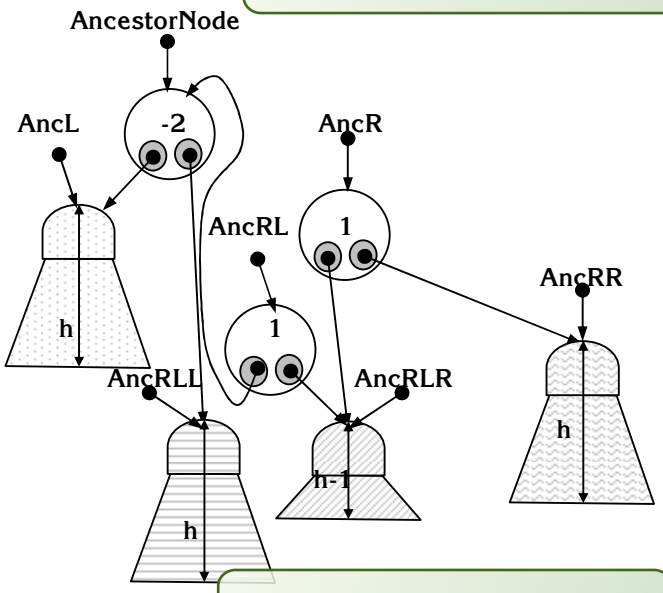
B1: AncestorNode->BAL_Right = AncRL->BAL_Left



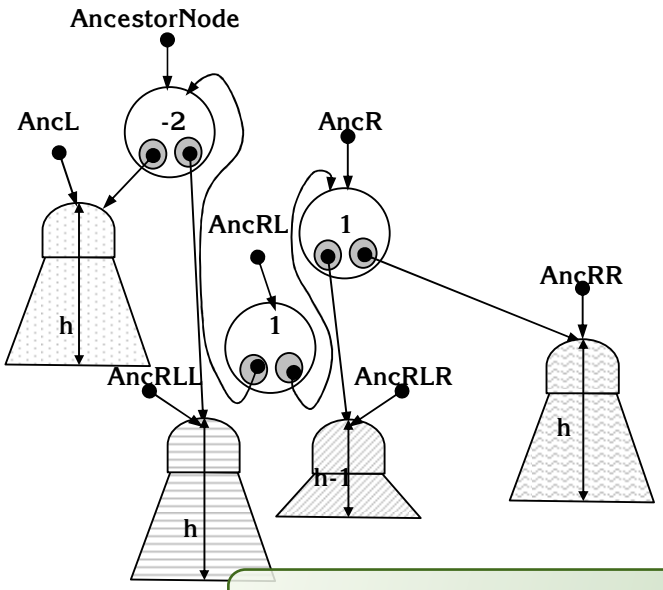
B2: AncR->BAL_Left = AncRL->BAL_Right



B3: AncRL->BAL_Left = AncestorNode



B4: AncRL->BAL_Right = AncR



Hiệu chỉnh lại các chỉ số cân bằng:

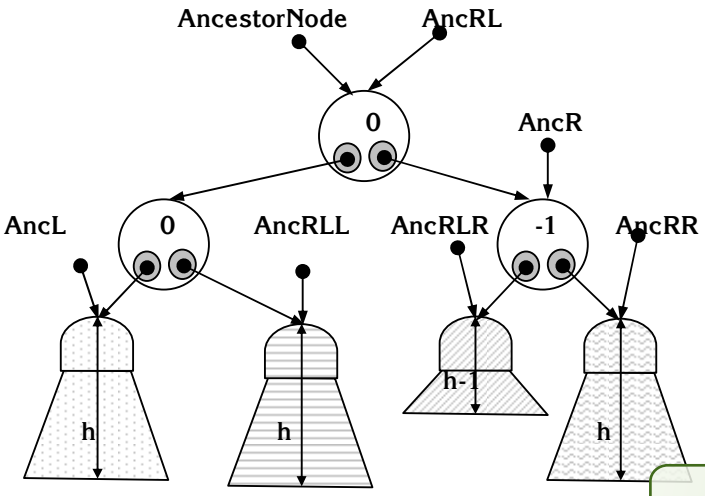
B5: AncestorNode->Bal = 0

B6: AncRL->Bal = 0

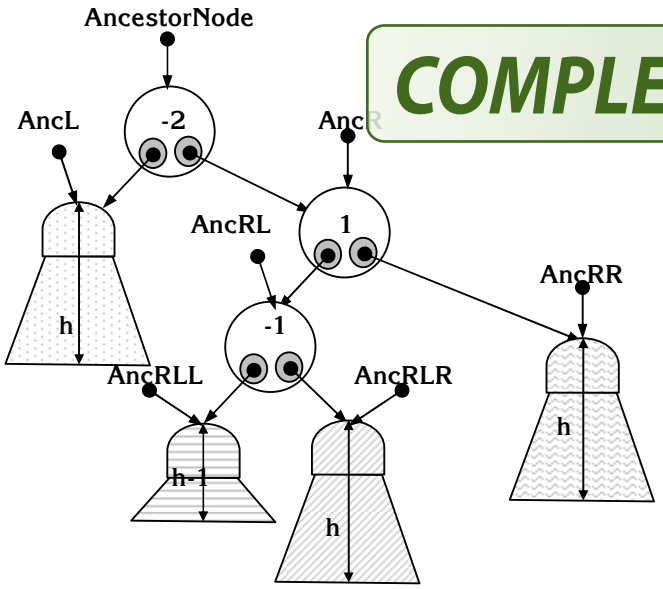
B7: AncR->Bal = -1

Chuyển vai trò của AncRL cho AncestorNode và chúng ta có cây cân bằng mới:

B8: AncestorNode = AncRL



- AncRLL có chiều cao là h-1 và AncRLR có chiều cao là h (AncRL->Bal =-1; h ≥ 1)



Để cân bằng lại AncestorNode hoàn toàn giống với trường hợp trên, duy chỉ khác nhau về giá trị chỉ số cân bằng sau khi quay kép. Chúng ta cũng thực hiện các bước sau:

B1: AncestorNode->BAL_Right = AncRL->BAL_Left

B2: AncR->BAL_Left = AncRL->BAL_Right

B3: AncRL->BAL_Left = AncestorNode

B4: AncRL->BAL_Right = AncR

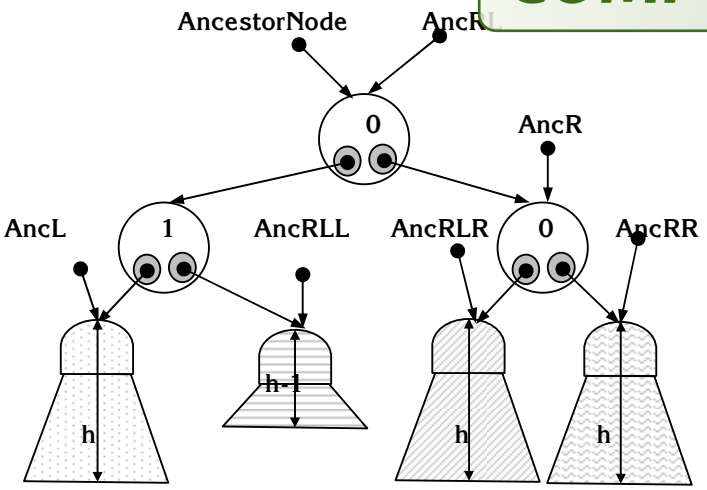
B5: AncestorNode->Bal = 1

B6: AncR->Bal = 0

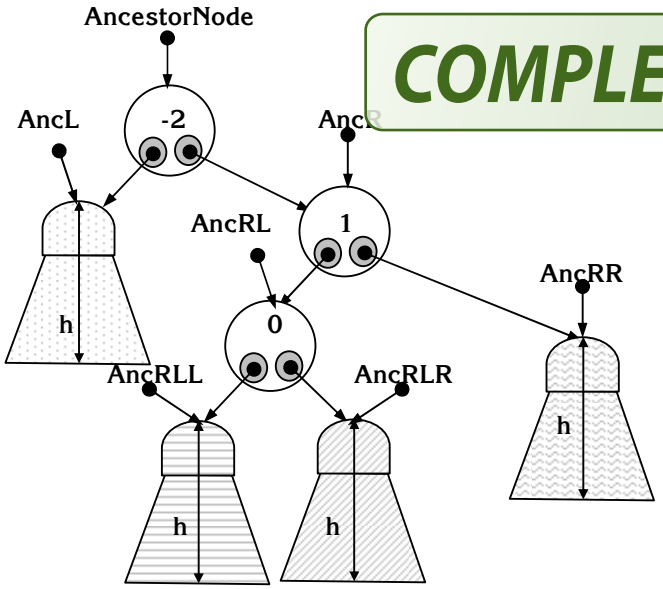
B7: AncRL->Bal = 0

B8: AncestorNode = AncRL

Sau khi quay kép cây sẽ trở thành:



- Cả AncRLL và AncRLR đều có chiều cao là h (AncRL->Bal =0; h ≥ 0)



Cũng tương tự, chúng ta cân bằng lại AncestorNode bằng cách quay kép giống như trường hợp trên nhưng về giá trị chỉ số cân bằng sau khi quay thì khác nhau. Các bước thực hiện như sau:

B1: AncestorNode->BAL_Right = AncRL->BAL_Left

B2: AncR->BAL_Left = AncRL->BAL_Right

B3: AncRL->BAL_Left = AncestorNode

B4: AncRL->BAL_Right = AncR

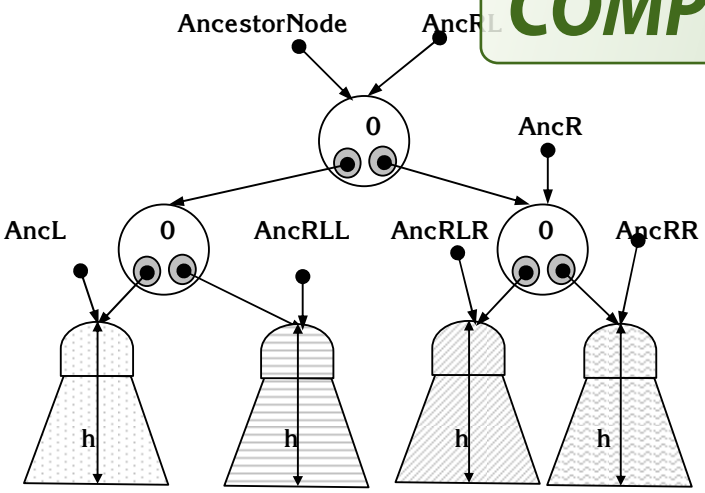
B5: AncestorNode->Bal = 0

B6: AncR->Bal = 0

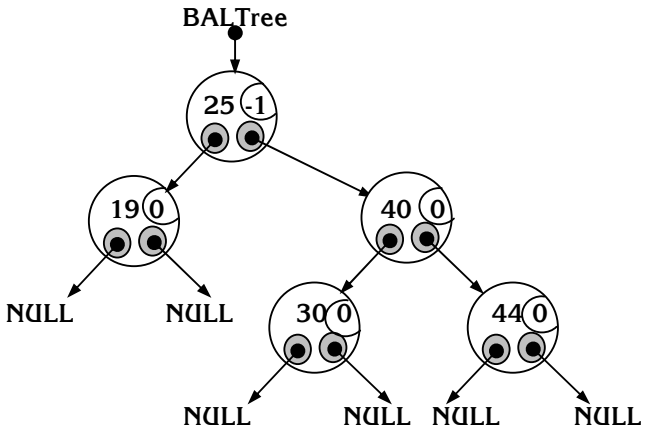
B7: AncRL->Bal = 0

B8: AncestorNode = AncRL

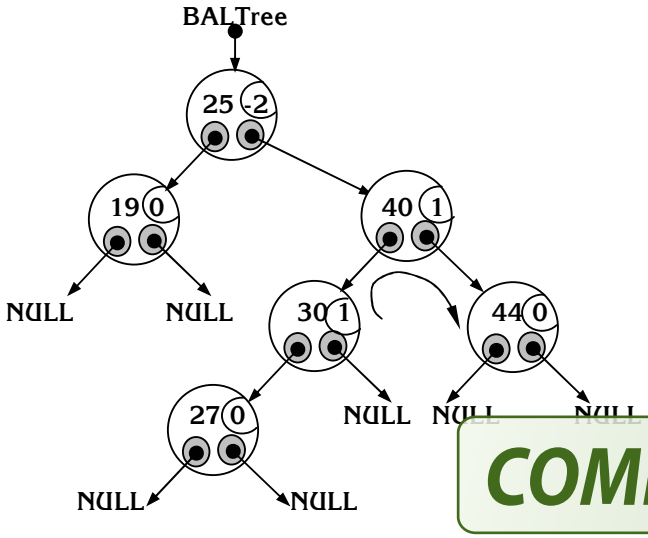
Sau khi quay kép cây sẽ trở thành:



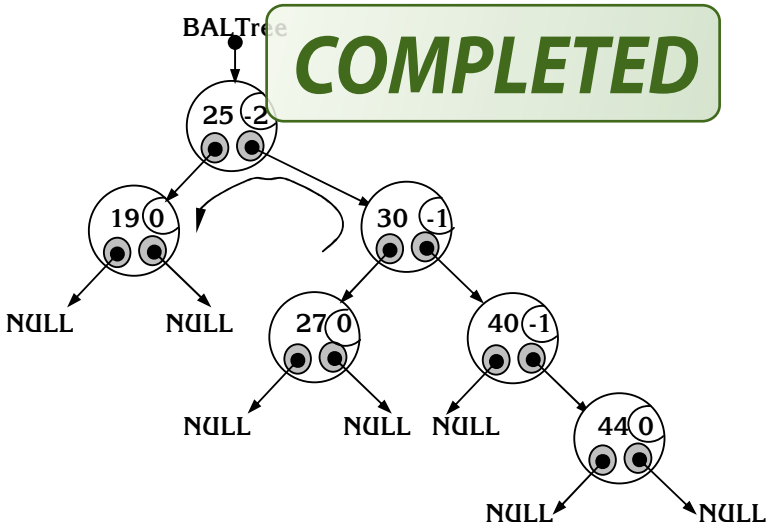
Ví dụ: Thêm nút có Key = 27 vào cây nhị phân tìm kiếm cân bằng sau đây:



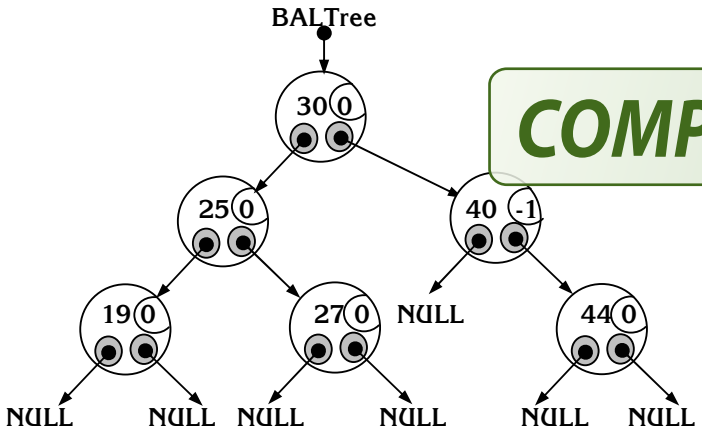
Cây nhị phân tìm kiếm cân bằng sau khi thêm nút có Key = 27 như sau:



Thực hiện quay đơn cây con trái của BALTree->BAL_Right cây nhị phân tìm kiếm sau khi quay trở thành cây nhị phân tìm kiếm như sau:



Thực hiện quay đơn cây con phải của BALTree cây nhị phân tìm kiếm sau khi quay trở thành cây nhị phân tìm kiếm cân bằng như sau:



Trường hợp 2: Nếu AncestorNode->Bal = 2:

Cũng tương tự như trường hợp 1 song ở đây chúng ta sẽ thực hiện quay đơn hoặc quay kép các nhánh phía ngược lại

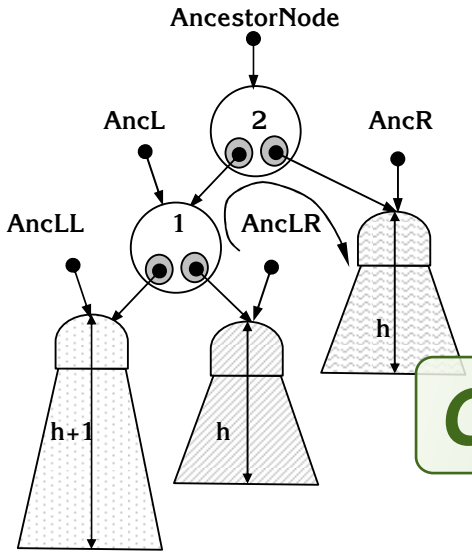
Gọi: AncL = AncestorNode->BAL_Left
AncR = AncestorNode->BAL_Right

⇒ AncL có chiều cao là h+2 và AncR có chiều cao là h (h ≥ 0)
⇒ Có ít nhất 1 cây con của AncL có chiều cao là h+1

Gọi: AncLL = AncL->BAL_Left
AncLR = AncL->BAL_Right

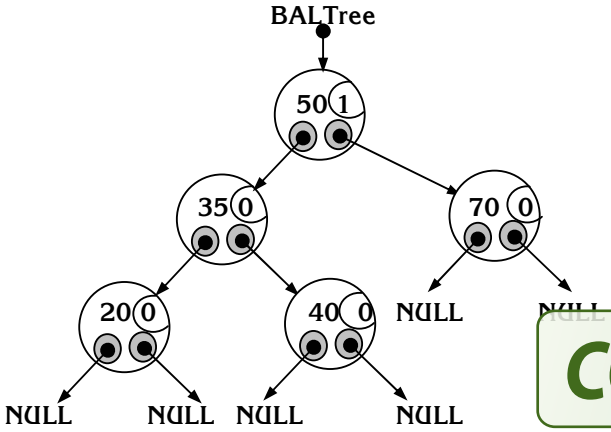
⇒ Cây con có nút gốc AncestorNode có thể ở vào một trong ba dạng sau:

a_2) AncLL có chiều cao là h+1 và AncLR có chiều cao là h (AncL->Bal = 1)



Để cân bằng lại AncestorNode chúng ta thực hiện việc quay đơn cây con trái AncL của nút này lên thành nút gốc; chuyển AncestorNode thành nút con phải của nút gốc và AncestorNode có hai cây con là AncLR và AncR (BAL_Left Rotation).

Ví dụ: Việc thêm nút có Key = 10 vào cây nhị phân tìm kiếm cân bằng sau đây sẽ làm cho cây mất cân bằng và chúng ta phải cân bằng lại theo trường hợp này:

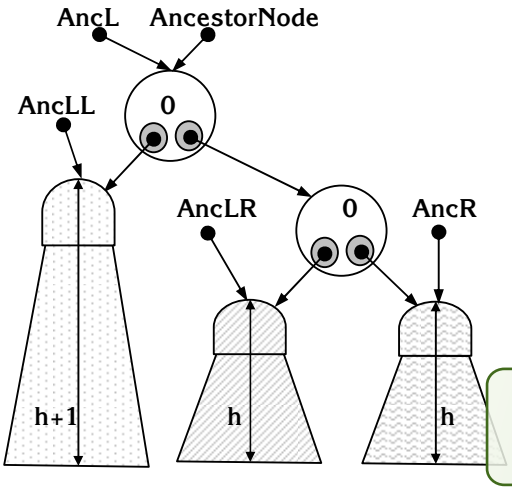


Các bước thực hiện việc cân bằng lại bằng phép quay này như sau:

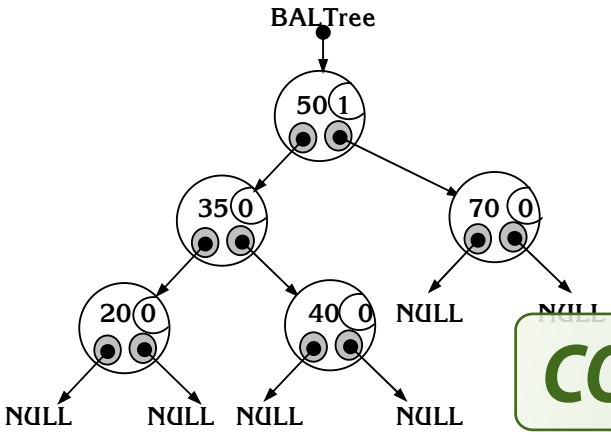
B1: AncestorNode->BAL_Left = AncL->BAL_Right
B2: AncL->BAL_Right = AncestorNode
B3: AncL->Bal = AncestorNode->Bal = 0

Chuyển vai trò của AncL cho AncestorNode:
B4: AncestorNode = AncL

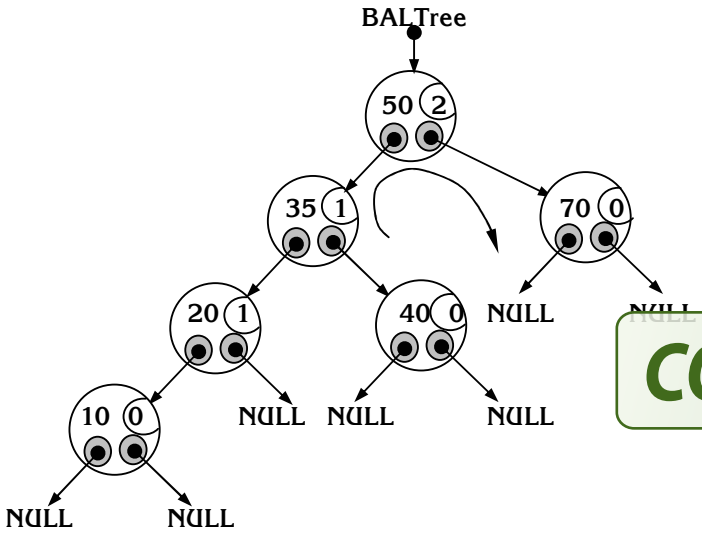
Kết quả sau phép quay đơn cây con trái:



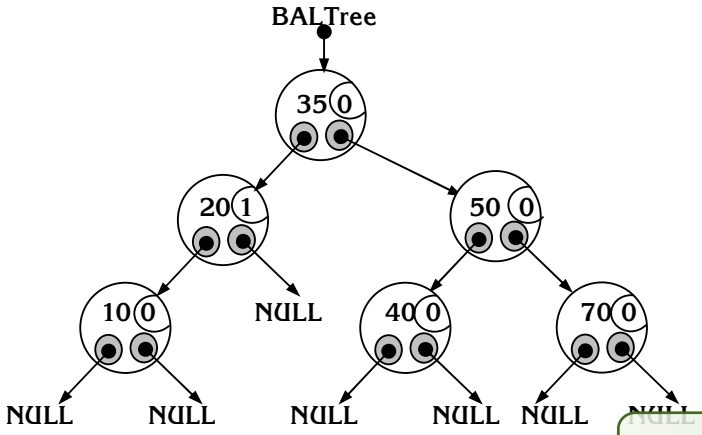
Ví dụ: Thêm nút có Key = 10 vào cây nhị phân tìm kiếm cân bằng sau đây:



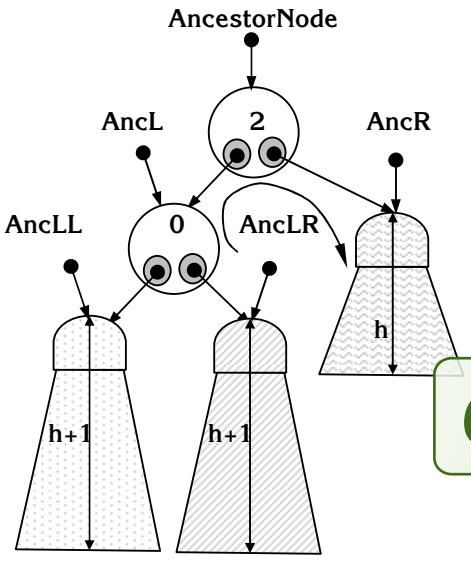
Cây nhị phân tìm kiếm cân bằng sau khi thêm nút có Key = 10 như sau:



Thực hiện quay cây con trái của BALTree, cây nhị phân tìm kiếm sau khi quay trở thành cây nhị phân tìm kiếm cân bằng như sau:



b_2) <u>AncLL và AncLR đều có chiều cao là h+1 (AncL->Bal = 0)</u>

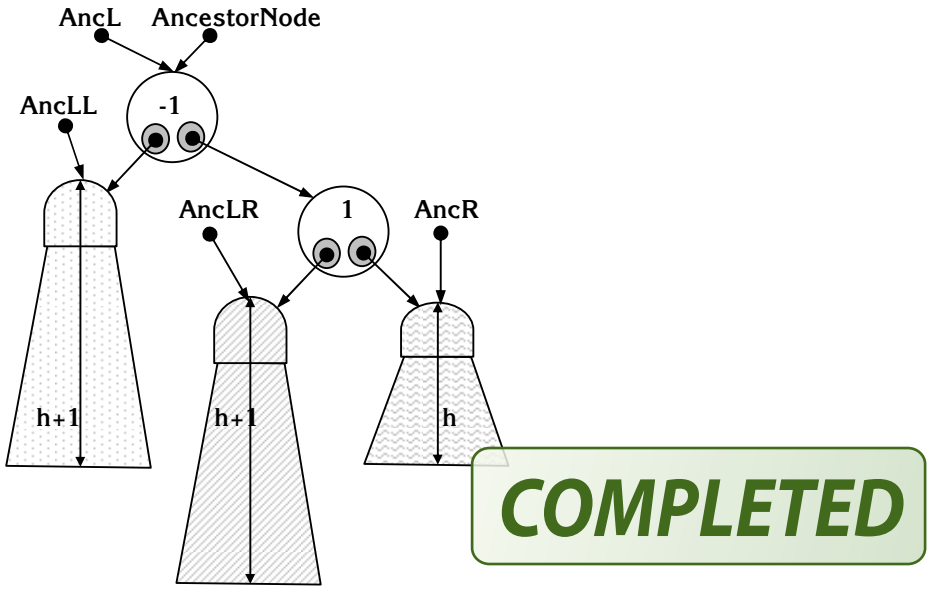


Việc cân bằng lại AncestorNode cũng thực hiện thao tác quay đơn như trên song chỉ số cân bằng sẽ khác. Do vậy, các bước thực hiện việc quay như sau:

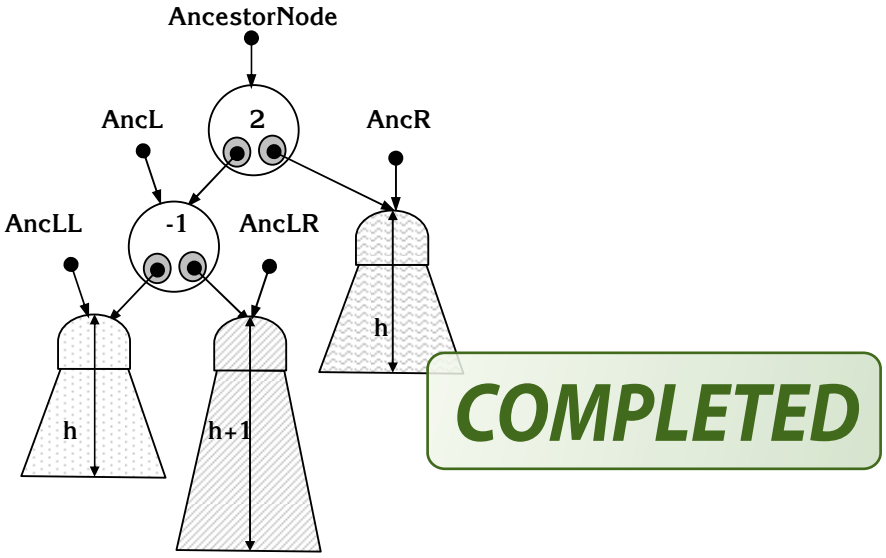
B1: AncestorNode->BAL_Left = AncL->BAL_Right
B2: AncL->BAL_Right = AncestorNode
B3: AncL->Bal = -1
B4: AncestorNode->Bal = 1

Chuyển vai trò của AncL cho AncestorNode:
B5: AncestorNode = AncL

Kết quả sau phép quay đơn cây con trái:

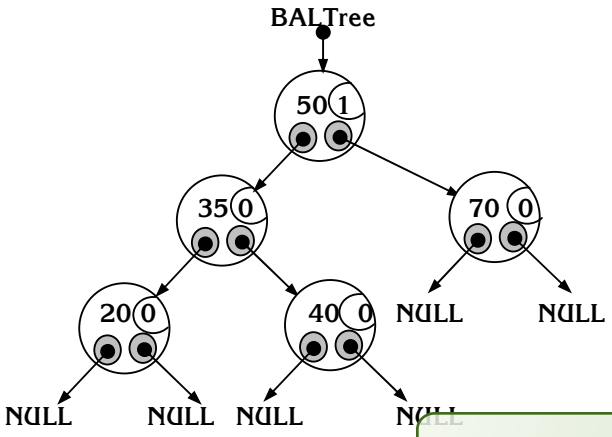


c_2) <u>AncLL có chiều cao là h và AncLR có chiều cao là h+1 (AncL->Bal = -1)</u>



Cũng tương tự như trường hợp c_1) Việc cân bằng lại AncestorNode được thực hiện thông qua phép quay kép: quay cây con phải AncLR và quay cây con trái AncL (Double Rotation).

Ví dụ: Việc thêm nút có Key = 44 vào cây nhị phân tìm kiếm cân bằng sau đây sẽ làm cho cây mất cân bằng và chúng ta phải cân bằng lại theo trường hợp này:

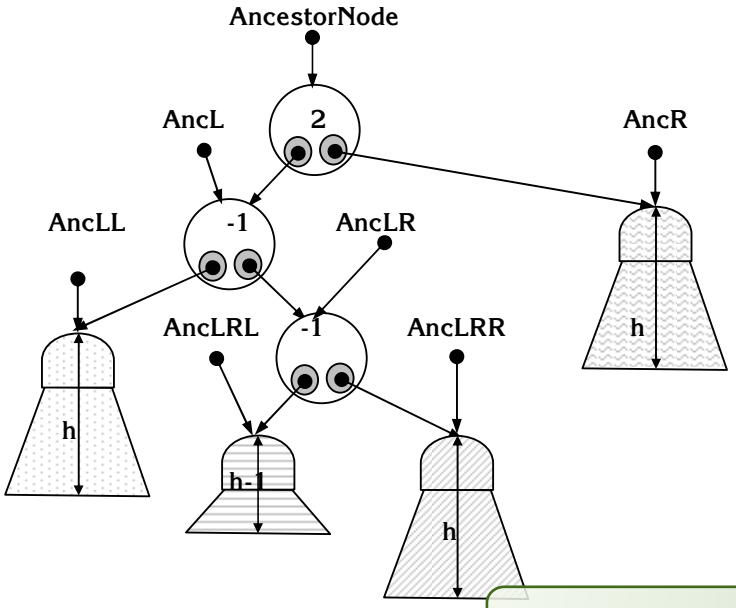


Việc quay được tiến hành cụ thể như sau:

Gọi: AncLRL = AncLR->BAL_Left
AncLRR = AncLR->BAL_Right

⇒ AncLRL và AncLRR có chiều cao tối đa là h
⇒ Cây con có nút gốc AncestorNode có thể ở vào một trong ba dạng sau:

- AncLRL có chiều cao là h-1 và AncLRR có chiều cao là h (AncRL->Bal =-1; h ≥ 1)



Quá trình quay kép được thực hiện thông các bước sau:

B1: AncestorNode->BAL_Left = AncLR->BAL_Right
B2: AncL->BAL_Right = AncLR->BAL_Left
B3: AncLR->BAL_Right = AncestorNode
B4: AncLR->BAL_Left = AncL

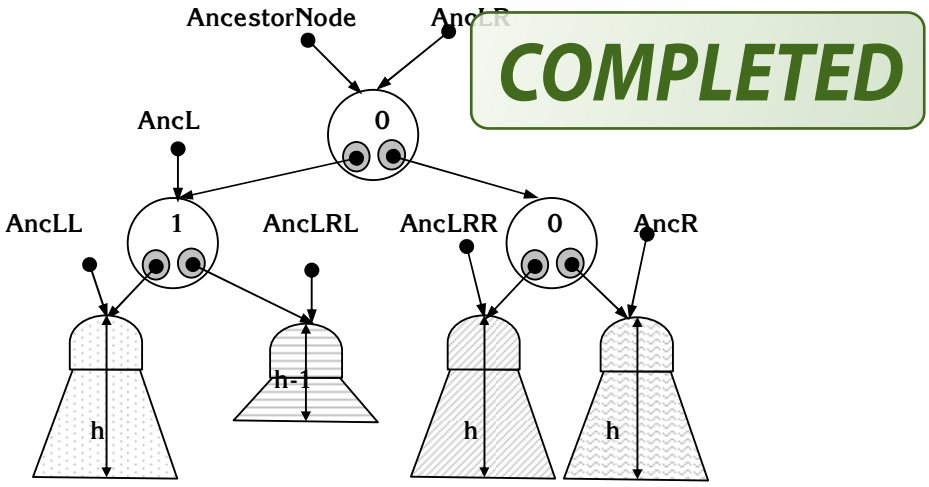
Hiệu chỉnh lại các chỉ số cân bằng:

B5: AncestorNode->Bal = 0
B6: AncLR->Bal = 0

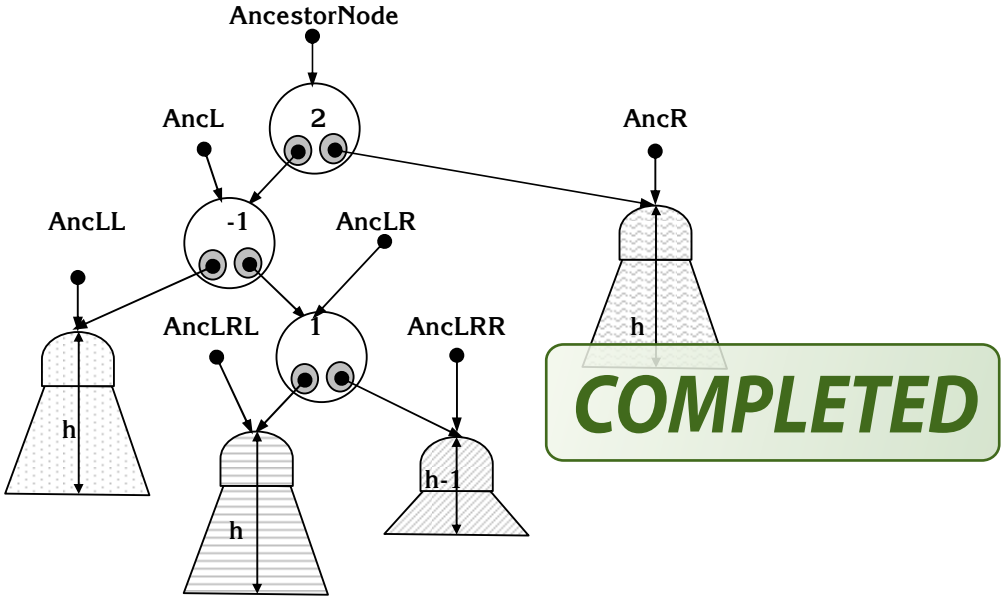
B7: AncL->Bal = 1

Chuyển vai trò của AncLR cho AncestorNode và chúng ta có cây cân bằng mới:

B8: AncestorNode = AncLR



- AncLRL có chiều cao là h và AncLRR có chiều cao là h-1 (AncRL->Bal =1; h ≥ 1)



Quá trình quay kép được thực hiện thông các bước sau:

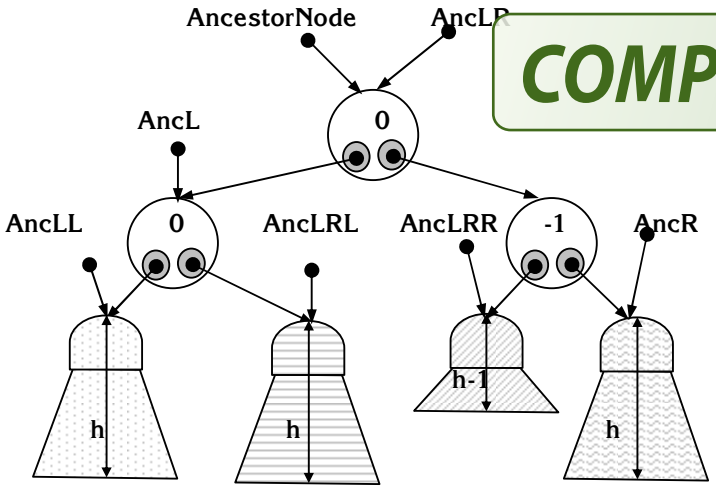
B1: AncestorNode->BAL_Left = AncLR->BAL_Right
B2: AncL->BAL_Right = AncLR->BAL_Left
B3: AncLR->BAL_Right = AncestorNode
B4: AncLR->BAL_Left = AncL

Hiệu chỉnh lại các chỉ số cân bằng:

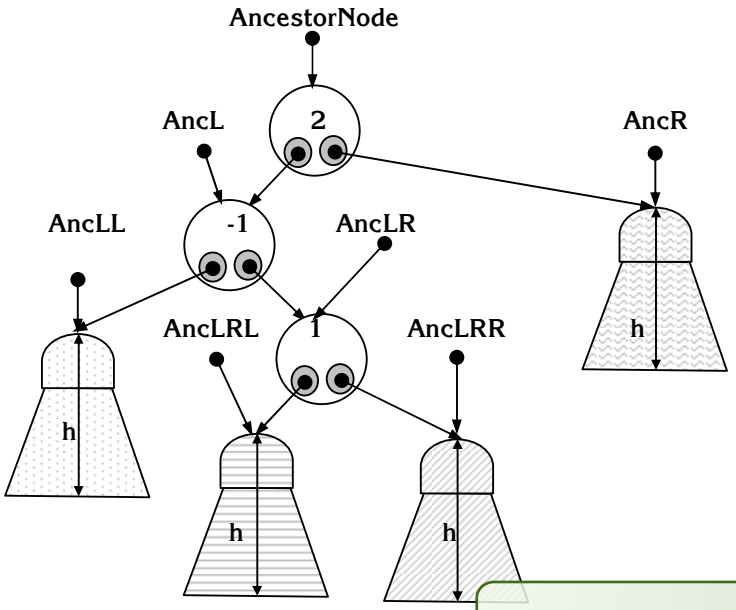
B5: AncestorNode->Bal = -1
B6: AncLR->Bal = 0
B7: AncL->Bal = 0

Chuyển vai trò của AncLR cho AncestorNode và chúng ta có cây cân bằng mới:

B8: AncestorNode = AncLR



- Cả AncLRL và AncLRR đều có chiều cao là h (AncRL->Bal =0; h ≥ 0)



Quá trình quay kép được thực hiện thông các bước sau:

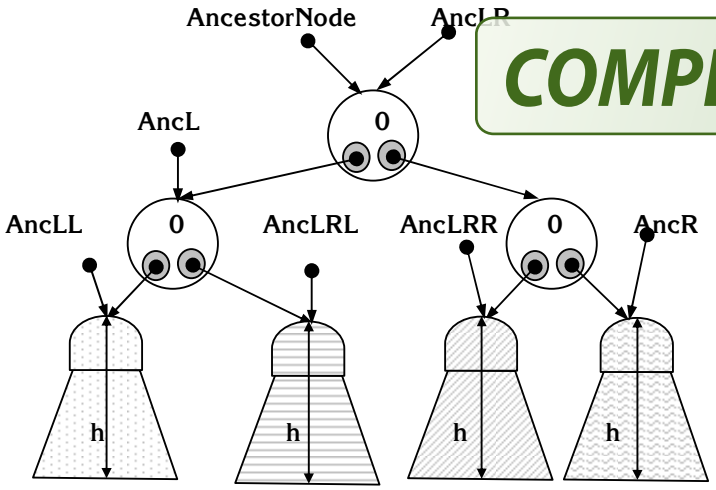
B1: AncestorNode->BAL_Left = AncLR->BAL_Right
B2: AncL->BAL_Right = AncLR->BAL_Left
B3: AncLR->BAL_Right = AncestorNode
B4: AncLR->BAL_Left = AncL

Hiệu chỉnh lại các chỉ số cân bằng:

B5: AncestorNode->Bal = 0
B6: AncLR->Bal = 0
B7: AncL->Bal = 0

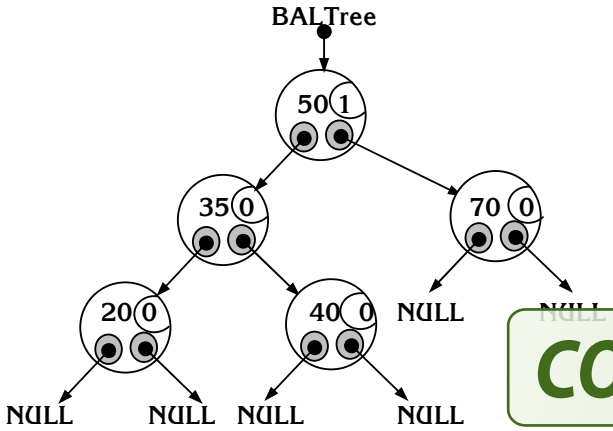
Chuyển vai trò của AncLR cho AncestorNode và chúng ta có cây cân bằng mới:

B8: AncestorNode = AncLR



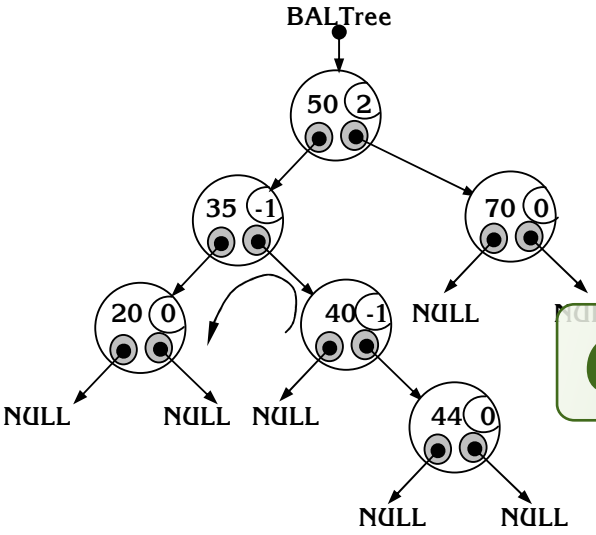
COMPLETED

Ví dụ: Thêm nút có Key = 44 vào cây nhị phân tìm kiếm cân bằng sau đây:



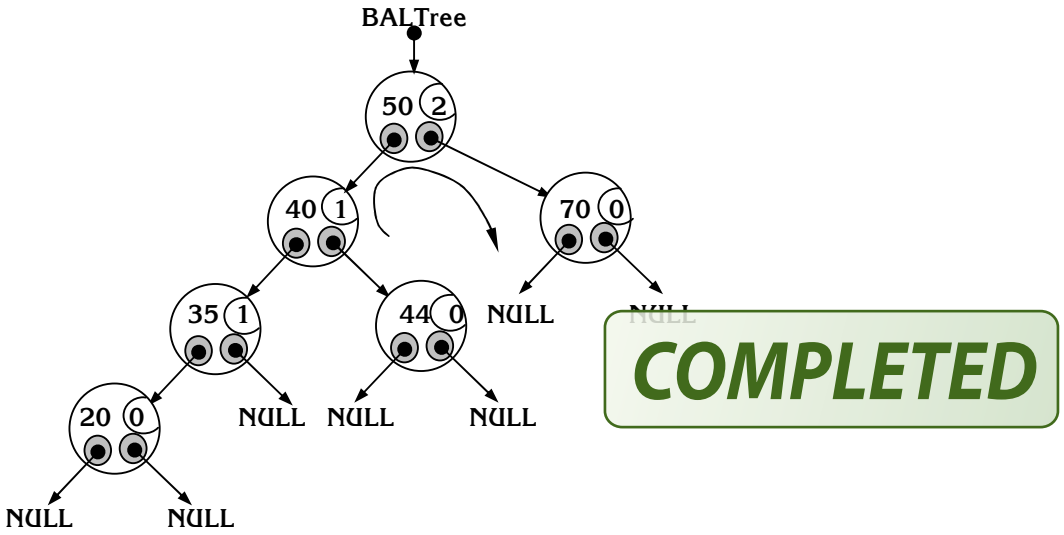
COMPLETED

Cây nhị phân tìm kiếm cân bằng sau khi thêm nút có Key = 44 như sau:

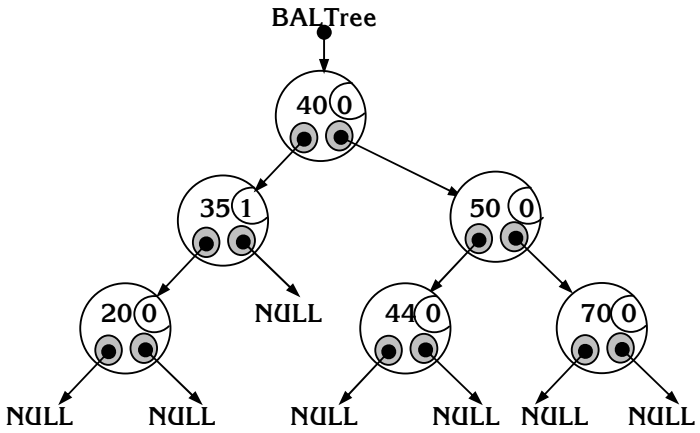


COMPLETED

Thực hiện quay cây con phải của BALTree->BAL_Left, cây nhị phân tìm kiếm sau khi quay trở thành cây nhị phân tìm kiếm như sau:



Thực hiện quay cây con phải của BALTree->BAL_Left, cây nhị phân tìm kiếm sau khi quay trở thành cây nhị phân tìm kiếm như sau:



- Thuật toán đệ quy để thêm 1 nút vào cây nhị phân tìm kiếm cân bằng tương đối (AddNew):

// Tạo nút mới có Key là NewData để thêm vào cây NPTKCBTĐ
B1: NewNode = new BAL_OneNode
B2: IF (NewNode = NULL)
    Thực hiện Bkt
B3: NewNode->BAL_Left = NewNode->BAL_Right = NULL
B4: NewNode->Key = NewData
B5: NewNode->Bal = 0
B6: IF (BALTree = NULL) // Cây rỗng
    B6.1: BALTree = NewNode
    B6.2: Taller = True // Cây NPTKCBTĐ bị cao lên hơn trước khi thêm
    B6.3: Thực hiện Bkt
B7: IF (BALTree->Key = NewData) // Trùng khóa
    Thực hiện Bkt
B8: IF (BALTree->Key < NewData)
    // Thêm đệ quy vào cây con phải của BALTree

B8.1: AddNew(NewData, BALTree->BAL_Right, Taller)
B8.2: If (Taller = True) // Việc thêm vào làm cho cây con phải cao thêm
B8.2.1: if (BALTree->Bal = 1) // Cây sẽ cân bằng tốt hơn
B8.2.1.1: BALTree->Bal = 0
B8.2.1.2: Taller = False
B8.2.1.3: Thực hiện Bkt
B8.2.2: if (BALTree->Bal = 0) // Cây vẫn còn cân bằng
B8.2.2.1: BALTree->Bal = -1
B8.2.2.2: Thực hiện Bkt
B8.2.3: if (BALTree->Bal = -1)
// Cây mất cân bằng theo trường hợp 1, phải cân bằng lại
B8.2.3.1: AncR = BALTree->BAL_Right
B8.2.3.2: if (AncR->Bal ≠ 1) // Thực hiện quay đơn theo a_1), b_1)
B8.2.3.2.1: BALTree->BAL_Right = AncR->BAL_Left
B8.2.3.2.2: AncR->BAL_Left = BALTree
B8.2.3.2.3: if (AncR->Bal = -1)
BALTree->Bal = AncR->Bal = 0
B8.2.3.2.4: else
AncR->Bal = 1
B8.2.3.2.5: BALTree = AncR
B8.2.3.3: else // Thực hiện quay kép theo c_1)
B8.2.3.3.1: AncRL = AncR->BAL_Left
B8.2.3.3.2: BALTree->BAL_Right = AncRL->BAL_Left
B8.2.3.3.3: AncR->BAL_Left = AncRL->BAL_Right
B8.2.3.3.4: AncRL->BAL_Left = BALTree
B8.2.3.3.5: AncRL->BAL_Right = AncR
B8.2.3.3.6: if (AncRL->Bal = 1)
B8.2.3.3.6.1: BALTree->Bal = AncRL->Bal = 0
B8.2.3.3.6.2: AncR->Bal = -1
B8.2.3.3.7: if (AncRL->Bal = -1)
AncR->Bal = AncRL->Bal = 0
B8.2.3.3.8: if (AncRL->Bal = 0)
AncR->Bal = BALTree->Bal = 0
B8.2.3.3.9: BALTree = AncRL
B8.2.3.4: Taller = False
B9: IF (BALTree->Key > NewData)
// Thêm đệ quy vào cây con trái của BALTree
B9.1: AddNew(NewData, BALTree->BAL_Left, Taller)
B9.2: If (Taller = True) // Việc thêm vào làm cho cây con trái cao thêm
B9.2.1: if (BALTree->Bal = -1) // Cây sẽ cân bằng tốt hơn
B9.2.1.1: BALTree->Bal = 0
B9.2.1.2: Taller = False
B9.2.1.3: Thực hiện Bkt
B9.2.2: if (BALTree->Bal = 0) // Cây vẫn còn cân bằng
B9.2.2.1: BALTree->Bal = 1
B9.2.2.2: Thực hiện Bkt
B9.2.3: if (BALTree->Bal = 1)

COMPLETED

COMPLETED

// Cây mất cân bằng theo trường hợp 2, phải cân bằng lại
B9.2.3.1: AncL = BALTree->BAL_Left
B9.2.3.2: if (AncL->Bal ≠ -1) // Thực hiện quay đơn theo a_2), b_2)
B9.2.3.2.1: BALTree->BAL_Left = AncL->BAL_Right
B9.2.3.2.2: AncL->BAL_Right = BALTree
B9.2.3.2.3: if (AncL->Bal = 1)
BALTree->Bal = AncL->Bal = 0
B9.2.3.2.4: else
AncL->Bal = -1
B9.2.3.2.5: BALTree = AncR

COMPLETED

B9.2.3.3: else // Thực hiện quay kép theo c_2)
B9.2.3.3.1: AncLR = AncL->BAL_Right
B9.2.3.3.2: BALTree->BAL_Left = AncLR->BAL_Right
B9.2.3.3.3: AncL->BAL_Right = AncLR->BAL_Left
B9.2.3.3.4: AncLR->BAL_Right = BALTree
B9.2.3.3.5: AncLR->BAL_Left = AncL
B9.2.3.3.6: if (AncLR->Bal = -1)
B9.2.3.3.6.1: BALTree->Bal = AncLR->Bal = 0
B9.2.3.3.6.2: AncL->Bal = 1
B9.2.3.3.7: if (AncLR->Bal = 1)
AncL->Bal = AncLR->Bal = 0
B9.2.3.3.8: if (AncLR->Bal = 0)
AncL->Bal = BALTree->Bal = 0
B9.2.3.3.9: BALTree = AncLR
B9.2.3.4: Taller = False

COMPLETED

Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm BAL_Add_Node có prototype:

BAL_Type BAL_Add_Node (BAL_Type &BTree, T NewData, int &Taller);

Hàm thực hiện việc thêm vào cây nhị phân tìm kiếm cân bằng BTree một nút có thành phần Key là NewData. Hàm trả về con trỏ trỏ tới địa chỉ của nút mới thêm nếu việc thêm thành công, trong trường hợp ngược lại hàm trả về con trỏ NULL. Trong trường hợp việc thêm làm cho cây phát triển chiều cao thì Taller có giá trị là 1, ngược lại Taller có giá trị là 0.

```
BAL_Type BAL_Add_Node (BAL_Type &BTree, T NewData, int &Taller)
  { if (BS_Tree == NULL)
      { BTree = new BAL_OneNode;
        if (BTree != NULL)
          { BTree->Key = NewData;
            BTree->Bal = 0;
            BTree->BAL_Left = BTree->BAL_Right = NULL;
            Taller = 1;
          }
        return (BTree);
      }
```

COMPLETED

```
if (BTree->Key == NewData)
   {  Taller = 0;
      return (NULL);
   }
if (BTree->Key < NewData)
   {  BAL_Add_Node (BTree->BAL_Right, NewData, Taller);
      if (Taller == 1)
         {  switch (BTree->Bal)
               {  case 1: BTree->Bal = 0;
                          Taller = 0;
                          break;
                  case 0: BTree->Bal = -1;
                          break;
                  case -1: BAL_Type  AncR = BTree->BAL_Right;
                          if (AncR->Bal != 1)
                             {  BTree->BAL_Right = AncR->BAL_Left
                                AncR->BAL_Left = BTree;
                                if (AncR->Bal == -1)
                                   BTree->Bal = AncR->Bal = 0;
                                else
                                   AncR->Bal = 1;
                                BTree = AncR;
                             }
                          else
                             {  BAL_Type  AncRL = AncR->BAL_Left;
                                BTree->BAL_Right = AncRL->BAL_Left;
                                AncR->BAL_Left = AncRL->BAL_Right;
                                AncRL->BAL_Left = BTree;
                                AncRL->BAL_Right = AncR;
                                if (AncRL->Bal == 1)
                                   {  BTree->Bal = AncRL->Bal = 0;
                                      AncR->Bal = -1;
                                   }
                                else
                                   if (AncRL->Bal == -1)
                                      AncR->Bal = AncRL->Bal = 0;
                                   else
                                      AncR->Bal = BTree->Bal = 0;
                                BTree = AncRL;
                             }
                          Taller = 0;
                          break;
               } // switch
         } // if (Taller == 1)
   } // if (BTree->Key < NewData)
else // (BTree->Key > NewData)
   {  BAL_Add_Node (BTree->BAL_Left, NewData, Taller);
      if (Taller == 1)
```

COMPLETED

COMPLETED

```
{  switch (BTree->Bal)
      {  case -1: BTree->Bal = 0;
                  Taller = 0;
                  break;
         case 0: BTree->Bal = 1;
                  break;

         case 1: BAL_Type  AncL = BTree->BAL_Left;
                  if (AncL->Bal != -1)
                     {  BTree->BAL_Left = AncL->BAL_Right
                        AncL->BAL_Right = BTree;
                        if (AncL->Bal == 1)
                           BTree->Bal = AncL->Bal = 0;
                        else
                           AncL->Bal = -1;
                        BTree = AncL;
                     }
                  else
                     {  BAL_Type  AncLR = AncL->BAL_Right;
                        BTree->BAL_Left = AncLR->BAL_Right;
                        AncL->BAL_Right = AncLR->BAL_Left;
                        AncLR->BAL_Right = BTree;
                        AncLR->BAL_Left = AncL;
                        if (AncLR->Bal == -1)
                           {  BTree->Bal = AncLR->Bal = 0;
                              AncL->Bal = 1;
                           }
                        else
                           if (AncLR->Bal == 1)
                              AncL->Bal = AncLR->Bal = 0;
                           else
                              AncL->Bal = BTree->Bal = 0;
                        BTree = AncLR;
                     }
                  Taller = 0;
                  break;
      } // switch
   } // if (Taller == 1)
} // else: (BTree->Key > NewData)
return (BTree);
}
```

b. Hủy một nút ra khỏi cây cân bằng:

Tương tự như trong tháo tác thêm, giả sử chúng ta cần hủy một nút DelNode có thành phần dữ liệu là DelData ra khỏi cây cân bằng BALTree sao cho sau khi hủy BALTree vẫn là một cây cân bằng. Để thực hiện điều này trước hết chúng ta phải thực hiện việc tìm kiếm vị trí của nút cần hủy là nút con trái hoặc nút con phải của

một nút PrDelNode tương tự như trong cây nhị phân tìm kiếm. Việc hủy cũng chia làm ba trường hợp như đối với trong cây nhị phân tìm kiếm:

- DelNode là nút lá,
- DelNode là nút trung gian có 01 cây con,
- DelNode là nút có đủ 02 cây con.

Trong trường hợp DelNode có đủ 02 cây con chúng ta sử dụng phương pháp hủy phần tử thế mạng vì theo phương pháp này sẽ làm cho chiều cao của cây ít biến động hơn phương pháp kia.

Sau khi hủy DewNode ra khỏi cây con trái hoặc cây con phải của PrNewNode thì chỉ số cân bằng của các nút từ PrDelNode trở về các nút trước cũng sẽ bị thay đổi dây chuyền và chúng ta phải lần ngược từ PrDelNode về theo các nút trước để theo dõi sự thay đổi này. Nếu phát hiện tại một nút AncNode có sự thay đổi vượt quá phạm vi cho phép (bằng –2 hoặc +2) thì chúng ta tiến hành cân bằng lại cây ngay tại nút AncNode này.

Việc cân bằng lại cây tại nút AncNode được tiến hành cụ thể theo các trường hợp tương tự như trong thao tác thêm:

- Thuật toán đệ quy để hủy 1 nút trong cây nhị phân tìm kiếm cân bằng tương đối (BAL_Delete_Node):

```
// Tìm nút cần hủy và nút cha của nút cần hủy
B1: PrDelNode = NULL
B2: IF (BALTree = NULL)
        B2.1: Shorter = False
        B2.2: Thực hiện Bkt
B3: PrDelNode = BALTree
B4: IF (BALTree->Key > DelData)  // Chuyển sang cây con trái
        B4.1: OnTheLeft = True
        B4.2: BAL_Delete_Node (BALTree->BAL_Left, DelData, Shorter)
B5: IF (BALTree->Key < DelData)  // Chuyển sang cây con phải
        B5.1: OnTheLeft = False
        B5.2: BAL_Delete_Node (BALTree->BAL_Right, DelData, Shorter)
B6: If (Shorter = True)
        B6.1: if (OnTheLeft = True)
                B6.1.1: if (BALTree->Bal = 1)   // Cây cân bằng tốt hơn
                        B6.1.1.1: BALTree->Bal = 0
                        B6.1.1.2: Shorter = False
                // Cây vẫn bị thấp nhưng vẫn còn cân bằng
                B6.1.2: if (BALTree->Bal = 0)
                        BALTree->Bal = -1
                B6.1.3: if (BALTree->Bal = -1)  // Cây mất cân bằng
                        B6.1.3.1: AncR = BALTree->BAL_Right
                        B6.1.3.2: if (AncR->Bal ≠ 1) // Thực hiện quay đơn
                                B6.1.3.2.1: BALTree->BAL_Right = AncR->BAL_Left
                                B6.1.3.2.2: AncR->BAL_Left = BALTree
                                B6.1.3.2.3: if (AncR->Bal = -1)
```

BALTree->Bal = AncR->Bal = 0
B6.1.3.2.4: else
AncR->Bal = 1
B6.1.3.2.5: BALTree = AncR
B6.1.3.3: else // Thực hiện quay kép
B6.1.3.3.1: AncRL = AncR->BAL_Left
B6.1.3.3.2: BALTree->BAL_Right = AncRL->BAL_Left
B6.1.3.3.3: AncR->BAL_Left = AncRL->BAL_Right
B6.1.3.3.4: AncRL->BAL_Left = BALTree
B6.1.3.3.5: AncRL->BAL_Right = AncR
B6.1.3.3.6: if (AncRL->Bal = 1)
B6.1.3.3.6.1: BALTree->Bal = AncRL->Bal = 0
B6.1.3.3.6.2: AncR->Bal = -1
B6.1.3.3.7: if (AncRL->Bal = -1)
AncR->Bal = AncRL->Bal = 0
B6.1.3.3.8: if (AncRL->Bal = 0)
AncR->Bal = BALTree->Bal = 0
B6.1.3.3.9: BALTree = AncRL
B6.1.3.4: Shorter = False
B6.2: else // (OnTheLeft = False)
B6.2.1: if (BALTree->Bal = -1) // Cây cân bằng tốt hơn
B6.2.1.1: BALTree->Bal = 0
B6.2.1.2: Shorter = False

// Cây vẫn bị thấp nhưng vẫn còn cân bằng
B6.2.2: if (BALTree->Bal = 0)
BALTree->Bal = 1
B6.2.3: if (BALTree->Bal = 1) // Cây mất cân bằng
B6.2.3.1: AncL = BALTree->BAL_Left
B6.2.3.2: if (AncL->Bal ≠ -1) // Thực hiện quay đơn
B6.2.3.2.1: BALTree->BAL_Left = AncL->BAL_Right
B6.2.3.2.2: AncL->BAL_Right = BALTree
B6.2.3.2.3: if (AncL->Bal = 1)
BALTree->Bal = AncL->Bal = 0
B6.2.3.2.4: else
AncL->Bal = 1
B6.2.3.2.5: BALTree = AncL
B6.2.3.3: else // Thực hiện quay kép
B6.2.3.3.1: AncLR = AncL->BAL_Right
B6.2.3.3.2: BALTree->BAL_Left = AncLR->BAL_Right
B6.2.3.3.3: AncL->BAL_Right = AncLR->BAL_Left
B6.2.3.3.4: AncLR->BAL_Right = BALTree
B6.2.3.3.5: AncLR->BAL_Left = AncL
B6.2.3.3.6: if (AncLR->Bal = -1)
B6.2.3.3.6.1: BALTree->Bal = AncLR->Bal = 0
B6.2.3.3.6.2: AncL->Bal = 1
B6.2.3.3.7: if (AncLR->Bal = 1)
AncL->Bal = AncLR->Bal = 0

COMPLETED

B6.2.3.3.8: if (AncLR->Bal = 0)
AncL->Bal = BALTree->Bal = 0
B6.2.3.3.9: BALTree = AncLR
B6.2.3.4: Shorter = False

// Chuyển các mối quan hệ của DelNode cho các nút khác
B7: IF (PrDelNode = NULL) // Hủy là nút gốc

// Nếu nút cần hủy là nút lá
B7.1: If (BALTree->BAL_Left = NULL) and (BALTree->BAL_Right = NULL)
B7.1.1: BALTree = NULL
B7.1.2: delete BALTree
B7.1.3: Thực hiện Bkt

COMPLETED

// Nếu nút cần hủy có một cây con phải
B7.2: If (BALTree->BAL_Left = NULL) and (BALTree->BAL_Right != NULL)
B7.2.1: BALTree = BALTree->BAL_Right
B7.2.2: BALTree->BAL_Right = NULL
B7.2.3: delete BALTree
B7.2.4: Thực hiện Bkt

// Nếu nút cần hủy có một cây con trái
B7.3: If (BALTree->BAL_Left != NULL) and (BALTree->BAL_Right = NULL)
B7.3.1: BALTree = BALTree->BAL_Left
B7.3.2: BALTree->BAL_Left = NULL
B7.3.3: delete BALTree
B7.3.4: Thực hiện Bkt

B8: ELSE // nút cần hủy không phải là nút gốc

// Nếu nút cần hủy là nút lá
B8.1: If (BALTree->BAL_Left = NULL) and (BALTree->BAL_Right = NULL)

// Nút cần hủy là cây con trái của PrDelNode
B8.1.1: if (OnTheLeft = True)
PrDelNode->BAL_Left = NULL
B8.1.2: else // Nút cần hủy là cây con phải của PrDelNode
PrDelNode->BAL_Right = NULL
B8.1.3: delete BALTree
B8.1.4: Thực hiện Bkt

// Nếu nút cần hủy có một cây con phải
B8.2: If (BALTree->BAL_Left = NULL) and (BALTree->BAL_Right != NULL)
B8.2.1: if (OnTheLeft = True)
PrDelNode->BAL_Left = BALTree->BAL_Right
B8.2.2: else
PrDelNode->BAL_Right = BALTree->BAL_Right
B8.2.3: BALTree->BAL_Right = NULL
B8.2.4: delete BALTree
B8.2.5: Thực hiện Bkt

// Nếu nút cần hủy có một cây con trái
B8.3: If (BALTree->BAL_Left != NULL) and (BALTree->BAL_Right = NULL)

B8.3.1: if (OnTheLeft = True)
PrDelNode->BAL_Left = BALTree->BAL_Left
B8.3.2: else
PrDelNode->BAL_Right = BALTree->BAL_Left
B8.3.3: BALTree->BAL_Left = NULL
B8.3.4: delete BALTree
B8.3.5: Thực hiện Bkt

// Nếu DelNode có hai cây con
B9: If (BALTree->BAL_Left != NULL) and (BALTree->BAL_Right != NULL)

// Tìm nút trái nhất trong cây con phải của nút cần hủy và nút cha của nó
B9.1: MLNode = BALTree->BAL_Right
B9.2: PrMLNode = BALTree
B9.3: if (MLNode->BAL_Left = NULL)
Thực hiện B9.7
B9.4: PrMLNode = MLNode
B9.5: MLNode = MLNode->BAL_Left
B9.6: Lặp lại B9.3

// Chép dữ liệu từ MLNode về DelNode
B9.7: BALTree->Key = MLNode->Key

// Chuyển cây con phải của MLNode về cây con trái của PrMLNode
B9.8: if (PrMLNode = BALTree) // MLNode là nút phải của PrMLNode
PrMLNode->BAL_Right = MLNode->BAL_Right
B9.9: else // MLNode là nút trái của PrMLNode
PrMLNode->BAL_Left = MLNode->BAL_Right
B9.10: MLNode->BAL_Right = NULL

// Chuyển vai trò của MLNode cho nút cần hủy
B9.11: BALTree = MLNode
Bkt: Kết thúc

- Cài đặt thuật toán:

Hàm BAL_Del_Node có prototype:

```
int  BAL_Del_Node(BAL_Type  &BALTree, T  Data,
                              int &Shorter, BAL_Type  &PrDNode, int  &OnTheLeft);
```

Hàm thực hiện việc hủy nút có thành phần Key là Data trên cây nhị phân tìm kiếm cân bằng BALTree bằng phương pháp hủy phần tử thế mạng là phần tử phải nhất trong cây con trái của nút cần hủy (nếu nút cần hủy có hai cây con). Hàm trả về giá trị 1 nếu việc hủy thành công (có nút để hủy), trong trường hợp ngược lại hàm trả về giá trị 0 (không tồn tại nút có Key là Data hoặc cây rỗng).

```
int  BAL_Del_Node(BAL_Type  &BALTree, T  Data,
                              int &Shorter, BAL_Type  &PrDNode, int  &OnTheLeft)
{  if (BALTree != NULL)
      {  Shorter = 0;
         PrDNode = NULL;
         return (0)
      }
```

COMPLETED

```
PrDNode = BALTree;
if (BALTree->Key > Data)       // Hủy nút ở cây con trái
   {  OnTheLeft = 1;
      return(BAL_Del_Node (BALTree->BAL_Left, Data, Shorter, PrDNode));
   }
if (BALTree->Key < Data)       // Hủy nút ở cây con phải
   {  OnTheLeft = 0;
      return(BAL_Del_Node (BALTree->BAT_Right, Data, Shorter, PrDNode));
   }
if (Shorter == True)
   {  if (OnTheLeft == 1)
         {  if (BALTree->Bal == 1)     // Cây cân bằng tốt hơn
               {  BALTree->Bal = 0;
                  Shorter = 0;
               }
            if (BALTree->Bal==0) //Cây vẫn bị thấp nhưng vẫn còn cân bằng
               BALTree->Bal = -1;
            if (BALTree->Bal == -1)    // Cây mất cân bằng
               {  BAL_Type  AncR = BALTree->BAL_Right;
                  if (AncR->Bal != 1) // Thực hiện quay đơn
                     {  BALTree->BAL_Right = AncR->BAL_Left;
                        AncR->BAL_Left = BALTree;
                        if (AncR->Bal == -1)
                           BALTree->Bal = AncR->Bal = 0;
                        else
                           AncR->Bal = 1;
                        BALTree = AncR;
                     }
                  else     // Thực hiện quay kép
                     {  BAL_Type  AncRL = AncR->BAL_Left;
                        BALTree->BAL_Right = AncRL->BAL_Left;
                        AncR->BAL_Left = AncRL->BAL_Right;
                        AncRL->BAL_Left = BALTree;
                        AncRL->BAL_Right = AncR;
                        if (AncRL->Bal == 1)
                           {  BALTree->Bal = AncRL->Bal = 0;
                              AncR->Bal = -1;
                           }
                        if (AncRL->Bal == -1)
                           AncR->Bal = AncRL->Bal = 0;
                        if (AncRL->Bal == 0)
                           AncR->Bal = BALTree->Bal = 0;
                        BALTree = AncRL;
                     }
                  Shorter = 0;
               }
         }
      else // (OnTheLeft = 0)
```

```
            {  if (BALTree->Bal == -1)    // Cây cân bằng tốt hơn
                  {  BALTree->Bal = 0;
                     Shorter = 0;
                  }
            // Cây vẫn bị thấp nhưng vẫn còn cân bằng
               if (BALTree->Bal == 0)
                  BALTree->Bal = 1;
               if (BALTree->Bal == 1)     // Cây mất cân bằng
                  {  BAL_Type  AncL = BALTree->BAL_Left;
                     if (AncL->Bal != -1) // Thực hiện quay đơn
                        {  BALTree->BAL_Left = AncL->BAL_Right;
                           AncL->BAL_Right = BALTree;
                           if (AncL->Bal == 1)
                              BALTree->Bal = AncL->Bal = 0;
                           else
                              AncL->Bal = 1;
                           BALTree = AncL;
                        }
                     else     // Thực hiện quay kép
                        {  BAL_Type  AncLR = AncL->BAL_Right;
                           BALTree->BAL_Left = AncLR->BAL_Right;
                           AncL->BAL_Right = AncLR->BAL_Left;
                           AncLR->BAL_Right = BALTree;
                           AncLR->BAL_Left = AncL;
                           if (AncLR->Bal == -1)
                              {  BALTree->Bal = AncLR->Bal = 0;
                                 AncL->Bal = 1;
                              }
                           if (AncLR->Bal == 1)
                              AncL->Bal = AncLR->Bal = 0;
                           if (AncLR->Bal == 0)
                              AncL->Bal = BALTree->Bal = 0;
                           BALTree = AncLR
                        }
                     Shorter = 0;
                  }
            }
      }
   if (PrDNode == NULL)  // hủy nút gốc
      {  if (BALTree->BAL_Left == NULL && BALTree->BAL_Right == NULL)
            BALTree = NULL;
         else
            if (BALTree->BST_Left == NULL)    // nút cần hủy có 1 cây con phải
               { BALTree = BALTree->BAL_Right;
                  BALTree->BAL_Right = NULL;
               }
            else
               if (BALTree->BAL_Right == NULL) //nút cần hủy có 1 cây con trái
```

```
                { BALTree = BALTree->BAL_Left;
                  BALTree->BAL_Left = NULL;
                }
        }
    else        // nút cần hủy là nút trung gian
        {  if (BALTree->BAL_Left == NULL && BALTree->BAL_Right == NULL)
              if (OnTheLeft == 1)
                 PrDNode->BAL_Left = NULL;
              else
                 PrDNode->BAL_Right = NULL;
           else
              if (BALTree->BAL_Left == NULL)
                 {  if (OnTheLeft == 1)
                       PrDNode->BAL_Left = BALTree->BAL_Right;
                    else
                       PrDNode->BAL_Right = BALTree->BAL_Right;
                    BALTree->BAL_Right = NULL;
                 }
              else
                 if (BALTree->BAL_Right == NULL)
                    {  if (OnTheLeft == 1)
                          PrDNode->BAL_Left = BALTree->BAL_Left;
                       else
                          PrDNode->BAL_Right = BALTree->BAL_Left;
                       BALTree->BAL_Left = NULL;
                    }
        }
    if (BALTree->BAL_Left != NULL && BALTree->BAL_Right != NULL)
        {  BAL_Type  MLNode = BALTree->BAL_Right;
           BAL_Type  PrMLNode = BALTree;
           while (MLNode->BAL_Left != NULL)
              {  PrMLNode = MLNode;
                 MLNode = MLNode->BAL_Left;
              }
           BALTree->Key = MLNode->Key;
           if (PrMLNode == BALTree)
              PrMLNode->BAL_Right = MLNode->BAL_Right;
           else
              PrMLNode->BAL_Left = MLNode->BAL_Right;
           MLNode->BAL_Right = NULL;
           BALTree = MLNode;
        }
    delete  BALTree;
    return (1);
}
```

Câu hỏi và Bài tập

1. Trình bày khái niệm, đặc điểm và cấu trúc dữ liệu của các loại cây? So sánh với danh sách liên kết?
2. Hãy đưa ra phương pháp để chuyển từ cấu trúc dữ liệu của một cây N-phân nói chung thành một cây nhị phân?
3. Trình bày thuật toán và cài đặt tất cả các thao tác trên cây nhị phân tìm kiếm, cây nhị phân tìm kiếm cân bằng?
4. Trình bày thuật toán và cài đặt tất cả các thao tác trên cây nhị phân tìm kiếm, cây nhị phân tìm kiếm cân bằng trong trường hợp chấp nhận sự trùng khóa nhận diện của các nút trong cây?
5. Trình bày tất cả các thuật toán và cài đặt tất cả các thuật toán để thực hiện việc hủy một nút trên cây nhị phân tìm kiếm nếu cây có 02 cây con? Theo bạn, thuật toán nào là đơn giản? Cho nhận xét về mỗi thuật toán?
6. Trình bày và cài đặt tất cả các thuật toán để thực hiện các thao tác trên cây nhị phân tìm kiếm, cây nhị phân tìm kiếm cân bằng trong hai trường hợp: Chấp nhận và Không chấp nhận sự trùng lắp về khóa của các nút bằng cách không sử dụng thuật toán đệ quy (Trừ các thao tác đã trình bày trong tài liệu)?
7. Trình bày thuật toán và cài đặt chương trình thực hiện các công việc sau trên cây nhị phân:
   a) Tính số nút lá của cây.
   b) Tính số nút trung gian của cây.
   c) Tính chiều dài đường đi tới một nút có khóa là K trên cây.
   d) Cho biết cấp của một nút có khóa là K trên cây.
8. Trình bày thuật toán và cài đặt chương trình thực hiện công việc tạo cây nhị phân tìm kiếm mà khóa của các nút là khóa của các nút trong một danh sách liên kết đôi sao cho tối ưu hóa bộ nhớ. Biết rằng, danh sách liên kết đôi ban đầu không cần thiết sau khi tạo xong cây nhị phân tìm kiếm và giả sử không cho phép sự trùng khóa giữa các nút trong cây.
9. Với yêu cầu trong bài tập 8 ở trên, trong trường hợp nếu danh sách liên kết có nhiều nút có thành phần dữ liệu giống nhau, bạn hãy đề xuất phương án giải quyết để không bị mất dữ liệu sau khi tạo xong cây nhị phân tìm kiếm.
10. Trình bày thuật toán và cài đặt chương trình thực hiện công việc chuyển cây nhị phân tìm kiếm thành danh sách liên kết đôi sao cho tối ưu hóa bộ nhớ. Biết rằng, cây nhị phân tìm kiếm ban đầu không cần thiết sau khi tạo xong danh sách liên kết *(ngược với yêu cầu trong bài tập 8)*.
11. Trình bày thuật toán và cài đặt chương trình thực hiện công việc nhập hai cây nhị phân tìm kiếm thành một cây nhị phân tìm kiếm duy nhất sao cho tối ưu bộ nhớ. Biết rằng, hai cây nhị phân tìm kiếm ban đầu không cần thiết sau khi tạo xong cây mới.

ÔN TẬP (REVIEW)

Hệ thống lại các Cấu trúc dữ liệu và các Giải thuật đã học

COMPLETED

Chương 1: **Tổng quan về Cấu Trúc Dữ Liệu và Giải Thuật**

1. Tầm quan trọng của Cấu trúc dữ liệu và Giải thuật trong một đề án tin học

1.1. Xây dựng Cấu trúc dữ liệu

1.2. Xây dựng Giải thuật

1.3. Mối quan hệ giữa Cấu trúc dữ liệu và Giải thuật

2. Đánh giá Cấu trúc dữ liệu và Giải thuật

2.1. Các tiêu chuẩn đánh giá Cấu trúc dữ liệu

- Thời gian thực hiện
- Mức độ tiêu tốn bộ nhớ
- Tính thực tế

2.2. Đánh giá độ phức tạp của thuật toán

3. Kiểu dữ liệu

3.1. Khái niệm về Kiểu dữ liệu

T = {V, O}

3.2. Các kiểu dữ liệu cơ sở

- Nguyên
- Thực
- Ký tự

3.3. Các kiểu dữ liệu có cấu trúc

- Mảng
- Cấu trúc (struct)

3.4. Kiểu dữ liệu con trỏ

T * Pt;

3.5. Kiểu dữ liệu tập tin

FILE * Fp;

int Fh;

COMPLETED

Chương 2: **Kỹ thuật tìm kiếm *(Searching)***

1. Khái quát về tìm kiếm

2. Các giải thuật tìm kiếm nội (tìm kiếm trên dãy)

2.1. Tìm tuyến tính (Linear Search)

Duyệt từ đầu đến cuối mảng để tìm

2.2. Tìm nhị phân (Binary Search)

Duyệt từng nửa các phần tử, chỉ áp dụng cho mảng đã có thứ tự.

3. Các giải thuật tìm kiếm ngoại (tìm kiếm trên tập tin)

3.1. Tìm tuyến tính (Linear Search)

Duyệt từ đầu đến cuối file để tìm

3.2. Tìm kiếm theo chỉ mục (Index Search)

Duyệt từ đầu đến tập tin chỉ mục để lấy dữ liệu trong tập tin dữ liệu.

Chương 3: **Kỹ thuật sắp xếp *(Sorting)***

1. Khái quát về sắp xếp

2. Các phương pháp sắp xếp nội (sắp xếp dãy)

2.1. Sắp xếp bằng phương pháp đổi chỗ (Exchange)
- Nổi bọt (Bubble Sort)
- Phân hoạch (Quick Sort)

2.3. Sắp xếp bằng phương pháp chọn (Selection)
Chọn trực tiếp (Straight Selection Sort)

2.4. Sắp xếp bằng phương pháp chèn (Insertion)
- Chèn trực tiếp (Straight Insertion Sort)

2.5. Sắp xếp bằng phương pháp trộn (Merge)
- Trộn trực tiếp (Straight Merge Sort)
- Trộn tự nhiên (Natural Merge Sort)

3. Các phương pháp sắp xếp ngoại (sắp xếp tập tin)

3.1. Sắp xếp bằng phương pháp trộn
- Trộn trực tiếp (Straight Merge Sort)
- Trộn tự nhiên (Natural Merge Sort)

3.2. Sắp xếp theo chỉ mục

COMPLETED

Chương 4: Danh sách *(List)*

1. Khái niệm về danh sách

2. Các phép toán trên danh sách

3. Danh sách đặc (Condensed List)

3.1. Định nghĩa

3.2. Biểu diễn và Các thao tác

```
const int MaxLen = 10000;      // hoặc:      #define MaxLen 10000
int Length;
T CD_LIST[MaxLen];// hoặc:    T * CD_LIST = new T[MaxLen];
```

3.3. Ưu nhược điểm và Ứng dụng

4. Danh sách liên kết (Linked List)

4.1. Định nghĩa

4.2. Danh sách liên kết đơn (Singly Linked List)

```
typedef struct SLL_Node
  { T Key;
    SLL_Node * NextNode;
  } SLL_OneNode;

typedef SLL_OneNode * SLL_Type;
```

4.3. Danh sách liên kết kép (Doubly Linked List)

```
typedef struct DLL_Node
  { T Key;
    DLL_Node * NextNode;
    DLL_Node * PreNode;
  }     DLL_OneNode;

typedef DLL_OneNode * DLL_Type;
```

COMPLETED

4.4. Ưu nhược điểm của danh sách liên kết

5. Danh sách hạn chế

5.1. Hàng đợi (Queue)

```
typedef struct Q_C
```

```
{  int  Len;   // Chiều dài hàng đợi
   int  Front, Rear;
   T  *  List; // Nội dung hàng đợi
}     C_QUEUE;

C_QUEUE  CQ_List;
```

5.2. Ngăn xếp (Stack)

```
typedef  struct  S_C
   {  int  Size;   // Kích thước ngăn xếp
      int  SP;
      T  *  List; // Nội dung ngăn xếp
   }     C_STACK;

C_STACK  CS_List;
```

Chương 5: Cây *(Tree)*

1. Các khái niệm

2. Cây nhị phân (Binary tree)

2.1. Định nghĩa

2.2. Biểu diễn và Các thao tác

```
typedef  struct  BinT_Node
   {  T  Key;
      BinT_Node * BinT_Left;
      BinT_Node * BinT_Right;
   } BinT_OneNode;

typedef  BinT_OneNode  *  BinT_Type;
```

2.3. Cây nhị phân tìm kiếm (Binary Searching Tree)

```
typedef  struct  BST_Node
   {  T  Key;
      BST_Node * BST_Left;
      BST_Node * BST_Right;
   } BST_OneNode;

typedef  BST_OneNode  *  BST_Type;
```

3. Cây cân bằng (Balanced tree)

3.1. Định nghĩa

```
typedef  struct  BAL_Node
   {  T  Key;
      int  Bal;
      BAL_Node * BAL_Left;
      BAL_Node * BAL_Right;
   } BAL_OneNode;

typedef  BAL_OneNode  *  BAL_Type;
```

COMPLETED

3.2. Các thao tác

Câu hỏi và Bài tập ôn tập tổng hợp

1. Phân biệt về cấu trúc dữ liệu, ý nghĩa và tác dụng giữa: danh sách liên kết đôi, danh sách đa liên kết có hai mối liên kết và cây nhị phân?

2. Hãy sử dụng cấu trúc dữ liệu thích hợp để lưu trữ các số nguyên có dấu có giá trị tuyệt đối quá lớn trong bộ nhớ trong của máy tính. Với cấu trúc dữ liệu này, hãy trình bày thuật toán và cài đặt chương trình thực hiện việc cộng, trừ, nhân, chia nguyên, lấy dư, so sánh các số nguyên có giá trị lớn này.

3. Hãy sử dụng cấu trúc dữ liệu thích hợp để lưu trữ độ dài đường đi giữa các Thành phố với nhau trong một quốc gia vào trong bộ nhớ trong của máy tính. Với cấu trúc dữ liệu này, hãy trình bày thuật toán và cài đặt chương trình thực hiện việc liệt kê tất cả các đường đi từ Thành phố A đến Thành phố B? Đường đi nào là đường đi ngắn nhất?

4. Các văn bản được lưu trữ thành từng dòng trên các file văn bản, mỗi dòng có chiều dài không quá 127 ký tự. Hãy đề xuất cấu trúc dữ liệu thích hợp để lưu trữ trong bộ nhớ trong của máy tính tần suất xuất hiện của các từ trong tập tin văn bản này. Với cấu trúc dữ liệu này, hãy trình bày thuật toán và cài đặt chương trình thực hiện việc thống kê xem các từ trong file văn bản xuất hiện với tần suất như thế nào? Cho biết văn bản có bao nhiêu từ, bao nhiêu tên từ?

5. Các văn bản được lưu trữ thành từng dòng trên các file văn bản, mỗi dòng có chiều dài không quá 127 ký tự. Hãy đề xuất cấu trúc dữ liệu thích hợp để lưu trữ trong bộ nhớ trong của máy tính các dòng văn bản trong tập tin văn bản này (có thể bộ nhớ không đủ để lưu toàn bộ nội dung tập tin văn bản này vào trong bộ nhớ trong của máy tính). Với cấu trúc dữ liệu này, hãy trình bày thuật toán và cài đặt chương trình thực hiện việc hiện nội tập tin văn bản này theo từng trang màn hình sao cho chúng ta có thể sử dụng các phím PgUp/PgDn để lật lên/xuống theo từng trang màn hình và sử dụng các phím Up-arrow/Down-arrow để cho trôi lên/xuống từng dòng văn bản trên màn hình? Cho biết văn bản có bao nhiêu dòng?

6. Hãy sử dụng cấu trúc dữ liệu thích hợp để lưu trữ các ma trận thưa *(ma trận mà chủ yếu giá trị các phần tử bằng 0)* trong bộ nhớ trong của máy tính. Với cấu trúc dữ liệu này, hãy trình bày thuật toán và cài đặt chương trình thực hiện việc cộng, trừ, nhân hai ma trận thưa với nhau, tạo ma trận thưa chuyển vị từ một ma trận thưa khác.

7. Hãy sử dụng cấu trúc dữ liệu thích hợp để lưu trữ Gia phả của một dòng họ nào đó trong bộ nhớ trong của máy tính. Với cấu trúc dữ liệu này, hãy trình bày thuật toán và cài đặt chương trình thực hiện việc kiểm tra xem 02 người có tên là X và Y có phải là hai anh em ruột hay không? Nếu không phải thì ai có "vai vế" cao hơn? Giả sử rằng mỗi cặp vợ chồng có không quá 05 người con.

8. Hãy sử dụng cấu trúc dữ liệu thích hợp để lưu trữ một hệ thống Menu có nhiều mục chọn, nhiều cấp trong bộ nhớ trong của máy tính. Với cấu trúc dữ liệu này, hãy trình bày thuật toán và cài đặt chương trình thực hiện việc cho menu xuất hiện trên màn hình và cho phép người sử dụng chọn một chức năng nào đó của menu.

9. Kết hợp cấu trúc dữ liệu ở trong bài tập và 4, 5 và 8. Hãy trình bày thuật toán và cài đặt chương trình thực hiện các chức năng của một phần mềm soạn thảo văn bản đơn giản?

10. Hãy sử dụng cấu trúc dữ liệu thích hợp để lưu trữ các từ của một từ điển vào trong tập tin có tên DICT.DAT. Thông tin giải nghĩa về một từ bao gồm: Tên từ, Loại từ *(Danh từ, động từ, tính từ, ...)*, nghĩa tiếng Việt.

a) Sử dụng tập tin chỉ mục để liệt kê các từ theo thứ tự Alphabet (A -> Z).

b) Hãy đề xuất cấu trúc dữ liệu thích hợp để lưu trữ trong bộ nhớ trong của máy tính thông tin giải nghĩa của các từ trong tập tin DICT.DAT này (có thể bộ nhớ không đủ để lưu toàn bộ nội dung tập tin DICT.DAT này vào trong bộ nhớ trong của máy tính). Với cấu trúc dữ liệu này, hãy trình bày thuật toán và cài đặt chương trình thực hiện việc tra nghĩa cho một từ. Ngoài ra, ta có thể sử dụng các phím PgUp/PgDn để lật lên/xuống theo từng trang (do mình quy định) màn hình và sử dụng các phím Up-arrow/Down-arrow để cho trôi lên/xuống từng từ trên màn hình? Sử dụng cấu trúc dữ liệu thích hợp để lưu trữ trong bộ nhớ trong các từ đã được tra nghĩa.

11. Người ta lưu trữ các hệ số của một đa thức bậc n thành các dòng văn bản trong file DATHUC.DAT theo nguyên tắc: Mỗi dòng là hệ số và số mũ của 1 đa thức và các hệ số và số mũ này cách nhau ít nhất là một khoảng trắng.

Hãy sử dụng cấu trúc dữ liệu thích hợp để lưu trữ một đa thức vào trong bộ nhớ trong của máy tính. Với cấu trúc dữ liệu này, hãy trình bày thuật toán và cài đặt chương trình thực hiện các công việc sau:

- Xuất các đa thức trong file DATHUC.DAT ra màn hình;
- Tính đa thức tổng, đa thức hiệu của các đa thức này;
- Tính đa thức tích của các đa thức này.

12. Một hình vuông có độ dài cạnh là *a* được tô 02 màu: trắng và đen. Người ta tiến hành chia hình vuông này thành 04 hình vuông con đều nhau và ghi nhận vị trí của chúng trong hình vuông lớn. Nếu trong mỗi hình vuông con chỉ gồm toàn màu trắng hoặc màu đen thì giữ nguyên, còn nếu trong hình vuông con còn có 02 màu thì tiếp tục chia hình vuông con này thành 04 hình vuông con nhỏ hơn và ghi nhận vị trí, ..., cứ như vậy sau một số hữu hạn phép chia sẽ kết thúc việc chia. Hãy sử dụng cấu trúc dữ liệu thích hợp để lưu trữ các hình vuông này vào trong bộ nhớ trong của máy tính. Với cấu trúc dữ liệu lựa chọn, hãy trình bày thuật toán và cài đặt chương trình thực hiện các công việc sau:

- Tính tổng số hình vuông tạo thành qua các lần chia.
- Tính tổng số hình vuông màu trắng, màu đen và tổng diện tích tương ứng của chúng.
- Tái tạo lại hình vuông ban đầu.

13. Định nghĩa cấu trúc dữ liệu thích hợp để lưu trữ các giá trị trong tam giác Pascal vào trong bộ nhớ trong của máy tính. Với cấu trúc dữ liệu này hãy trình bày thuật toán và viết chương trình thực hiện các công việc sau:

- In tam giác Pascal có N dòng ra màn hình.
- In và tính giá trị biểu thức $(a+b)^N$ ra màn hình.

14. Trình bày thuật toán và viết chương trình thực hiện công việc minh họa (Demo) quá trình thực hiện tất cả các thuật toán đã học.

IV. HƯỚNG DẪN SỬ DỤNG TÀI LIỆU THAM KHẢO

1. Cấu trúc dữ liệu
Tác giả: Nguyễn Trung Trực
Khoa CNTT, trường ĐHBK TP.HCM

2. Giáo trình Cấu trúc dữ liệu 1
Tác giả: Trần Hạnh Nhi – Dương Anh Đức
Khoa CNTT, trường ĐHKHTN – ĐHQG TP.HCM

3. Algorithms + Data Structures = Programs
Tác giả: N.Wirth
NXB: Prentice Hall, 1976

4. Data Structures and Algorithms
Tác giả: Alfred V.Aho - John E.Hopcroft – Jeffrey D.Ullman
NXB: Addison-Wesley Publishing Company

5. Algorithms (Second Edition)
Tác giả: Robert Sedgewick
NXB: Addison-Wesley Publishing Company

6. Data Structures and Program Design (Third Edition)
Tác giả: Robert L.Kruse
NXB: Prentice Hall