Chương 1: **Giới thiệu cấu trúc dữ liệu và giải thuật**

1. Mối liên hệ giữa Cấu trúc dữ liệu và Giải thuật

1.1. Cấu trúc dữ liệu.

Có thể nói rằng không có 1 chương trình máy tính nào mà không có dữ liệu để xử lý, dữ liệu có thể là dữ liệu vào, dữ liệu trung gian hoặc dữ liệu ra. 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 và cài đặt chương trình.

1.2. Giải thuật.

Khái niệm *giải thuật* hay *thuật giải* nhiều khi còn được gọi là thuật toán dùng để chỉ phương pháp hay cách thức để giải quyết vấn 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 dựa trên 1 hay 1 số ngôn ngữ lập trình nào đó (thường là ngôn ngữ mà người lập trình chọn để cài đặt thuật toán) như Pascal, C,...

Khi đã xác định được CTDL 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 CTDL đã được chọn. Để giải quyết 1 vấn đề có thể có nhiều phương pháp, do vậy sự lựa chọn phương pháp phù hợp là việc mà người lập trình cần phải cân nhắc và tính toán lựa chọn. Sự lựa chọn này cũng góp phần đáng kể trong việc giảm bớt công việc của người lập trình trong phần cài đặt trên 1 ngôn ngữ cụ thể.

1.3. Mối liên hệ giữa CTDL và GT

Thực hiện một đề án tin học là chuyển bài toán thực tế thành bài toán có thể giải quyết trên máy tính. Một bài toán thực tế bất kỳ đều bao gồm các đối tượng dữ liệu và các yêu cầu xử lý trên những đối tượng đó. Vì thế, để xây dựng một mô hình tin học phản ánh được bài toán thực tế cần chú trọng đến hai vấn đề:

- Tổ chức biểu diễn các đối tượng thực tế :

Các thành phần dữ liệu thực tế đa dạng, phong phú và thường chứa đựng những quan hệ nào đó với nhau, do đó trong mô hình tin học của bài toán, cần phải tổ chức , xây dựng các cấu trúc thích hợp nhất sao cho vừa có thể phản

ánh chính xác các dữ liệu thực tế này, vừa có thể dễ dàng dùng máy tính để xử lý. Công việc này được gọi là xây dựng *cấu trúc dữ liệu* cho bài toán.

Xây dựng các thao tác xử lý dữ liệu:

Từ những yêu cầu xử lý thực tế, cần tìm ra các giải thuật tương ứng để xác định trình tự các thao tác máy tính phải thi hành để cho ra kết quả mong muốn, đây là bước xây dựng *giải thuật* cho bài toán.

*Tuy nhiên khi giải quyết một bài toán trên máy tính, chúng ta thường có khuynh hướng chỉ chú trọng đến việc xây dựng giải thuật mà quên đi tầm quan trọng của việc tổ chức dữ liệu trong bài toán. Giải thuật phản ánh các phép xử lý , còn đối tượng xử lý của giải thuật lại là dữ liệu, chính dữ liệu chứa đựng các thông tin cần thiết để thực hiện giải thuật. Để xác định được giải thuật phù hợp cần phải biết nó tác động đến loại dữ liệu nào (ví dụ để làm nhuyễn các hạt đậu , người ta dùng cách xay chứ không băm bằng dao, vì đậu sẽ văng ra ngoài) và khi chọn lựa cấu trúc dữ liệu cũng cần phải hiểu rõ những thao tác nào sẽ tác động đến nó (ví dụ để biểu diễn các điểm số của sinh viên người ta dùng số thực thay vì chuỗi ký tự vì còn phải thực hiện thao tác tính trung bình từ những điểm số đó). Như vậy trong một đề án tin học, giải thuật và cấu trúc dữ liệu có mối quan hệ chặt chẽ với nhau, được thể hiện qua công thức :

Cấu trúc dữ liệu + Giải thuật = Chương trình

*Với một cấu trúc dữ liệu đã chọn, sẽ có những giải thuật tương ứng, phù hợp. Khi cấu trúc dữ liệu thay đổi thường giải thuật cũng phải thay đổi theo để tránh việc xử lý gượng ép, thiếu tự nhiên trên một cấu trúc không phù hợp. Hơn nữa, một cấu trúc dữ liệu tốt sẽ giúp giải thuật xử lý trên đó có thể phát huy tác dụng tốt hơn, vừa đáp ứng nhanh vừa tiết kiệm vật tư, giải thuật cũng dễ hiểu và đơn giản hơn.

2. Kiểu dữ liệu, mô hình dữ liệu, kiểu dữ liệu trừu tượng.

2.1. Kiểu dữ liệu và cấu trúc dữ liệu.

Trong các ngôn ngữ lập trình bậc cao, các dữ liệu được phân lớp thành các lớp dữ liệu dựa vào bản chất của dữ liệu. Mỗi một lớp dữ liệu được gọi là một *kiểu dữ liệu.* Như vậy, một kiểu T là một tập hợp nào đó, các phần tử của tập được gọi là các *giá trị* của kiểu. Chẳng hạn, kiểu *integer* là tập hợp các số nguyên, kiểu *char* là một tập hữu hạn các ký hiệu. Các ngôn ngữ lập trình khác nhau có thể có các kiểu dữ liệu khác nhau. Fortran có các kiểu dữ liệu là *integer*, *real*, *logical*, *complex* và *double complex*. Các kiểu dữ liệu trong ngôn ngữ C là *int, float, char, con trỏ, struct*..., Kiểu dữ liệu trong ngôn ngữ Lisp lại là các S-biểu thức. Một cách tổng quát, mỗi ngôn ngữ lập trình có một *hệ kiểu* của riêng mình. Hệ kiểu của một ngôn ngữ bao gồm các kiểu

dữ liệu cơ sở và các phương pháp cho phép ta từ các kiểu dữ liệu đã có xây dựng nên các kiểu dữ liệu mới.

Khi nói đến một kiểu dữ liệu, chúng ta cần phải đề cập đến hai đặc trưng sau đây:

1. Tập hợp các giá trị thuộc kiểu. Chẳng hạn, kiểu integer trong ngôn ngữ Pascal gồm tất cả các số nguyên được biểu diễn bởi hai byte, tức là gồm các số nguyên từ -32768 đến + 32767. Trong các ngôn ngữ lập trình bậc cao mỗi hằng, biến, biểu thức hoặc hàm cần phải được gắn với một kiểu dữ liệu xác định. Khi đó, mỗi biến (biểu thức, hàm) chỉ có thể nhận các giá trị thuộc kiểu của biến (biểu thức, hàm) đó.

Ví dụ , nếu X là biến có kiểu *boolean* trong Pascal (*var* X : *boolean*) thì X chỉ có thể nhận một trong hai giá trị *true, false*.

2. Với mỗi kiểu dữ liệu, cần phải xác định một tập hợp nào đó các phép toán có thể thực hiện được trên các dữ liệu của kiểu. Chẳng hạn, với kiểu real, các phép toán có thể thực hiện được là các phép toán số học thông thường +, -, *, / , và các phép toán so sánh = , < >, < , < =, >, > =.

Thông thường trong một hệ kiểu của một ngôn ngữ lập trình sẽ có một số kiểu dữ liệu được gọi là *kiểu dữ liệu đơn* hay *kiểu dữ liệu phân tử* (atomic).

Chẳng hạn, trong ngôn ngữ Pascal, các kiểu dữ liệu *integer, real*, *boolean* , *char* và các kiểu liệt kê được gọi là các kiểu dữ liệu đơn. Sở dĩ gọi là đơn, vì các giá trị của các kiểu này được xem là các đơn thể đơn giản nhất không thể phân tích thành các thành phần đơn giản hơn được nữa.

Như đã nói, khi giải quyết các bài toán phức tạp, chỉ sử dụng các dữ liệu đơn là không đủ, ta phải cần đến các *cấu trúc dữ liệu*. Một cấu trúc dữ liệu bao gồm một tập hợp nào đó các *dữ liệu thành phần*, các dữ liệu thành phần này được liên kết với nhau bởi một phương pháp nào đó. Các dữ liệu thành phần có thể là dữ liệu đơn, hoặc cũng có thể là một cấu trúc dữ liệu đã được xây dựng. Có thể hình dung một cấu trúc dữ liệu được tạo nên từ các *tế bào* (khối xây dựng), mỗi tế bào có thể xem như một cái hộp chứa dữ liệu thành phần.

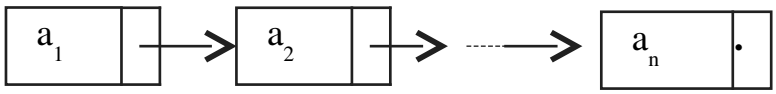
Trong Pascal và trong nhiều ngôn ngữ thông dụng khác có một cách đơn giản nhất để liên kết các tế bào, đó là sắp xếp các tế bào chứa các dữ liệu cùng một kiểu thành một dãy. Khi đó ta có một cấu trúc dữ liệu được gọi là *mảng* (array). Như vậy, có thể nói, một mảng là một cấu trúc dữ liệu gồm một dãy xác định các dữ liệu thành phần cùng một kiểu. Ta vẫn thường nói đến mảng các số nguyên, mảng các số thực, mảng các bản ghi, ... Mỗi một dữ liệu thành phần của mảng được gắn với một chỉ số từ một tập chỉ số nào đó. Ta có thể truy cập đến một thành phần nào đó của mảng bằng cách chỉ ra tên mảng và chỉ số của thành phần đó.

Một phương pháp khác để tạo nên các cấu trúc dữ liệu mới, là kết hợp một số tế bào (có thể chứa các dữ liệu có kiểu khác nhau) thành một *bản ghi* (record). Các tế bào thành phần của bản ghi được gọi là các *trường* của bản ghi. Các bản ghi đến lượt lại được sử dụng làm các tế bào để tạo nên các cấu trúc dữ liệu khác. Chẳng hạn, một trong các cấu trúc dữ liệu hay được sử dụng nhất là mảng các bản ghi.

Còn một phương pháp quan trọng nữa để kiến tạo các cấu trúc dữ liệu, đó là sử dụng con trỏ. Trong phương pháp này, mỗi tế bào là một bản ghi gồm hai phần INFO và LINK, phần INFO có thể có một hay nhiều trường dữ liệu, còn phần LINK có thể chứa một hay nhiều con trỏ trỏ đến các tế bào khác có quan hệ với tế bào đó. Chẳng hạn, ta có thể cài đặt một danh sách bởi cấu trúc dữ liệu danh sách liên kết, trong đó mỗi thành phần của danh sách liên kết là bản ghi gồm hai trường

```
type    Cell = record
            element : Item ;
            next : ^Cell ;
        end ;
```

Ở đây, trường *element* có kiểu dữ liệu *Item*, một kiểu dữ liệu nào đó của các phần tử của danh sách. Trường *nex*t là con trỏ trỏ tới phần tử tiếp theo trong danh sách. Cấu trúc dữ liệu danh sách liên kết biểu diễn danh sách $(a_1, a_2, ..., a_n)$ có thể được biểu diễn như trong hình 1.1



Hình 1.1: Cấu trúc dữ liệu danh sách liên kết.

Sử dụng con trỏ để liên kết các tế bào là một trong các phương pháp kiến tạo các cấu trúc dữ liệu được áp dụng nhiều nhất. Ngoài danh sách liên kết, người ta còn dùng các con trỏ để tạo ra các cấu trúc dữ liệu biểu diễn cây, một mô hình dữ liệu quan trọng bậc nhất.

Trên đây chúng ta đã nêu ba phương pháp chính để kiến tạo các cấu trúc dữ liệu. (Ở đây chúng ta chỉ nói đến các cấu trúc dữ liệu trong bộ nhớ trong, các cấu trúc dữ liệu ở bộ nhớ ngoài như file chỉ số, B-cây sẽ được đề cập riêng.)

Một kiểu dữ liệu mà các giá trị thuộc kiểu không phải là các dữ liệu đơn mà là các cấu trúc dữ liệu được gọi là *kiểu dữ liệu có cấu trúc*. Trong ngôn ngữ Pascal, các kiểu dữ liệu mảng, bản ghi, tập hợp, file đều là các kiểu dữ liệu có cấu trúc.

**** Các phép toán kết hợp dữ liệu.***

1. Các phép toán số học. Đó là các phép toán + , -, * , / trên các số thực ; các phép toán +, - *, /, div, mod trên các số nguyên.

2. Các phép toán so sánh. Trên các đối tượng thuộc các kiểu có thứ tự (đó là các kiểu cơ sở và kiểu tập), ta có thể thực hiện các phép toán so sánh =, < >, <, <=, >, >=. Cần lưu ý rằng, kết quả của các phép toán này là một giá trị kiểu *boolaen* (tức là *true* hoặc *false*).

3. Các phép toán logic. Đó là các phép toán *and, or, not* được thực hiện trên hai giá trị *false* và *truc* của kiểu *boolean*.

4. Các phép toán tập hợp. Các phép toán hợp, giao, hiệu của các tập hợp trong pascal được biểu diễn bởi +, *, - tương ứng. Việc kiểm tra một đối tượng x có là phần tử của tập A hay không được thực hiện bởi phép toán x *in* A. Kết quả của phép toán này là một giá *boolean.*

2.2. Mô hình dữ liệu và kiểu dữ liệu trừu tượng

2.2.1. Mô hình dữ liệu.

Để giải quyết một vấn đề trên MTĐT thông thường chúng ta cần phải qua một số giai đoạn chính sau đây :

1. Đặt bài toán

2. Xây dựng mô hình

3. Thiết kế thuật toán và phân tích thuật toán

4. Viết chương trình

5. Thử nghiệm.

Chúng ta sẽ không đi sâu phân tích từng giai đoạn. Chúng ta chỉ muốn làm sáng tỏ vai trò của mô hình dữ liệu trong việc thiết kế chương trình. Xét ví dụ sau.

Ví dụ. Một người giao hàng, hàng ngày anh ta phải chuyển hàng từ một thành phố đến một số thành phố khác rồi lại quay về thành phố xuất phát. Vấn đề của anh ta là làm thế nào có được một hành trình chỉ qua mỗi thành phố một lần với đường đi ngắn nhất có thể được.

Chúng ta thử giúp người giao hàng mô tả chính xác bài toán. Trước hết, ta cần trả lời câu hỏi, những thông tin đã biết trong bài toán người giao hàng là gì ? Đó là tên của các thành phố anh ta phải ghé qua và độ dài các con đường có thể có giữa hai thành phố. Chúng ta cần tìm cái gì ? Một hành trình mà người giao hàng mong muốn là một danh sách các thành phố $A_1,A_2...A_{n+1}$ (giả sử có n thành phố), trong đó các A_1 (i=1,2,...,n+1) đều khác nhau, trừ $A_{n+1} = A_1$.

Với một vấn đề đặt ra từ thực tiễn, ta có thể mô tả chính xác vấn đề đó hoặc các bộ phận của nó (vấn đề con) bởi một mô hình toán học nào đó. Chẳng hạn, mô hình toán học thích hợp nhất để mô tả bài toán người giao

hàng là đồ thị. Các đỉnh của đồ thị là các thành phố, các cạnh của đồ thị là các đường nối hai thành phố. Trọng số của các cạnh là độ dài các đường nối hai thành phố. Trong thuật ngữ của lý thuyết đồ thị, danh sách các thành phố biểu diễn hành trình của người giao hàng, là một chu trình qua tất cả các đỉnh của đồ thị. Như vậy, bài toán người giao hàng được qui về bài toán trong lý thuyết đồ thị. Tìm một chu trình xuất phát từ một đỉnh qua tất cả các đỉnh còn lại với độ dài ngắn nhất.

Bài toán người giao hàng là một trong các bài toán đã trở thành kinh điển. Nó dễ mô hình hoá, song cũng rất khó giải. Chúng ta sẽ quay lại bài toán này.

Cần lưu ý rằng, để tìm ra cấu trúc toán học thích hợp với một bài toán đã cho, chúng ta phải phân tích kỹ bài toán để tìm ra câu trả lời cho các câu hỏi sau.

Các thông tin quan trọng của bài toán có thể biểu diễn bởi các đối tượng toán học nào?

Có các mối quan hệ nào giữa các đối tượng?

Các kết quả phải tìm của bài toán có thể biểu diễn bởi các khái niệm toán học nào.

Sau khi đã có mô hình toán học mô tả bài toán, một câu hỏi đặt ra là, ta phải làm việc với mô hình như thế nào để tìm ra lời giải của bài toán ? Chúng ta sẽ thiết kế thuật toán thông qua các hành động, các phép toán thực hiện trên các đối tượng của mô hình.

Một mô hình toán học cùng với các phép toán có thể thực hiện trên các đối tượng của mô hình được gọi là *mô hình dữ liệu*. Chẳng hạn, trong mô hình dữ liệu đồ thị, trong số rất nhiều các phép toán, ta có thể kể ra một số phép toán sau : tìm các đỉnh kề của mỗi đỉnh, xác định đường đi ngắn nhất nối hai đỉnh bất kỳ, tìm các thành phần liên thông, tìm các đỉnh treo,.. Về mặt toán học, *danh sách* là một dãy hữu hạn n phần tử (a_1, a_2, ..., a_n). Trong mô hình dữ liệu danh sách, chúng ta cũng có thể thực hiện một tập hợp rất đa dạng các phép toán, chẳng hạn như, xác định độ dài của danh sách, xen một phần tử mới vào danh sách, loại một phần từ nào đó khỏi danh sách, sắp xếp lại danh sách theo một trật tự nào đó, gộp hai danh sách thành một danh sách.

Trở lại bài toán người giao hàng. Có nhiều thuật toán giải bài toán này. Chẳng hạn, ta có thể giải bằng phương pháp vét cạn : giả sử có n thành phố, khi đó mỗi hành trình là một hoán vị của n-1 thành phố (trừ thành phố xuất phát), thành lập (n-1)! hoán vị, tính độ dài của hành trình ứng với mỗi hoán vị và so sánh, ta sẽ tìm được hành trình ngắn nhất. Ta cũng có thể giải bài toán bằng phương pháp qui hoạch động (Phương pháp này sẽ được trình bày ở tập 2 của sách này). Sau đây ta đưa ra một thuật toán đơn giản. Thuật toán này tìm ra rất nhanh nghiệm "gần đúng", trong trường hợp có đường đi nối hai thành phố bất kỳ. Giả sử G là một đồ thị (Graph), V là tập hợp các đỉnh

(Node), E là tập hợp các cạnh của nó. Giả sử c(u,v) là độ dài (nguyên dương) của cạnh (u,v). Hành trình (Tour) của người giao hàng có thể xem như một tập hợp nào đó các cạnh. Cost là độ dài của hành trình. Thuật toán được biểu diễn bởi thủ tục Salesperson.

```
procedure    Salespersen (G : Graph ; var Tour : set of E ;
                          var cost : integer) ;
    var      v, w : Node
             U : set of V ;
begin
    Tour : = [ ] ;
    Cost  : = 0 ;
       v   : = v_o ; {v_o - đỉnh xuất phát}
           U : = V - [v_o] ;
    while      U < > [   ] do
    begin
    Chọn (v, w) là cạnh ngắn nhất với w thuộc U ;
    Tour : = Tour + [(v, w)] ;
    Cost  : = Cost + c (v,w) ;
        v  : = w ;
        U : = U - [w] ;
    end ;
    Tour : = Tour + [(v,v_o)] ;
    Cost  : = Cost + c(v,v_o) ;
end;
```

Thuật toán Salesperson được xây dựng trên cơ sở mô hình dữ liệu đồ thị và mô hình dữ liệu tập hợp. Nó chứa các thao tác trên đồ thị, các phép toán tập hợp. Tư tưởng của thuật toán như sau. Xuất phát từ Tour là tập rỗng. Giả sử ta xây dựng được đường đi từ đỉnh xuất phát v_0 tới đỉnh v. Bước tiếp theo, ta sẽ thêm vào Tour cạnh (v,w), đó là cạnh ngắn nhất từ v tới các đỉnh w không nằm trên đường đi từ v_0 tới v. Để có được chương trình, chúng ta phải biểu diễn đồ thị, tập hợp bởi các cấu trúc dữ liệu. Sau đó viết các thủ tục (hoặc hàm) thực hiện các thao tác, các phép toán trên đồ thị, tập hợp có trong thuật toán.

Tóm lại, quá trình giải một bài toán có thể quy về hai giai đoạn kế tiếp như sau

1. Xây dựng các mô hình dữ liệu mô tả bài toán. Thiết kế thuật toán bằng cách sử dụng các thao tác, các phép toán trên các mô hình dữ liệu.

2. Biểu diễn các mô hình dữ liệu bởi các cấu trúc dữ liệu. Với các cấu trúc dữ liệu đã lựa chọn, các phép toán trên các mô hình dữ liệu được thể hiện bởi các thủ tục (hàm) trong ngôn ngữ lập trình nào đó.

Toán học đã cung cấp cho Tin học rất nhiều cấu trúc toán học có thể dùng làm mô hình dữ liệu. Chẳng hạn, các khái niệm toán học như dãy, tập hợp, ánh xạ, cây, đồ thị, quan hệ, nửa nhóm, nhóm, otomat,...Trong các chương sau chúng ta sẽ lần lượt nghiên cứu một số mô hình dữ liệu quan trọng nhất, được sử dụng thường xuyên trong các thuật toán. Đó là các mô hình dữ liệu danh sách, cây, tập hợp. Với mỗi mô hình dữ liệu chúng ta nghiên cứu các cách cài đặt nó bởi các cấu trúc dữ liệu khác nhau. Trong mỗi cách cài đặt, một số phép toán trên mô hình có thể được thực hiện dễ dàng, nhưng các phép toán khác có thể lại không thuận tiện. Việc lựa chọn cấu trúc dữ liệu nào để biểu diễn mô hình phụ thuộc vào từng áp dụng.

Như đã nói, với mỗi mô hình dữ liệu, chúng ta có thể thực hiện một tập hợp các phép toán rất đa dạng, phong phú. Song trong nhiều áp dụng, chúng ta chỉ sử dụng mô hình với một số xác định các phép toán nào đó. Khi đó chúng ta sẽ có một kiểu dữ liệu trừu tượng.

Như vậy, một *kiểu dữ liệu trừu tượng* (abstract data type) là một mô hình dữ liệu được xét cùng với một số xác định các phép toán nào đó. Chẳng hạn, các tập hợp chỉ xét với các phép toán : thêm một phần tử vào một tập đã cho, loại một phần tử khỏi một tập hợp đã cho, tìm xem một phần tử đã cho có nằm trong một tập hợp hay không, lập thành kiểu dữ liệu trừu tượng (KDLTT) *từ điển* (dictionnaire).

Còn KDLTT *hàng* (hàng đợi) là mô hình dữ liệu danh sách cùng với hai phép toán chính là : thêm một phần tử mới vào một đầu danh sách, và loại một phần tử ở một đầu khác của danh sách. Chúng ta sẽ nghiên cứu kỹ một số kiểu dữ liệu trừu tượng quan trọng nhất : hàng, ngăn xếp (stack), từ điển, hàng ưu tiên. Với mỗi KDLTT, các cấu trúc dữ liệu để biểu diễn nó sẽ được nghiên cứu. Chúng ta cũng sẽ đánh giá hiệu quả của các phép toán trong từng cách cài đặt.

2.2.2. Trừu tượng hóa dữ liệu

Máy tính thực sự chỉ có thể lưu trữ dữ liệu ở dạng nhị phân thô sơ. Nếu muốn phản ánh được dữ liệu thực tế đa dạng và phong phú,cần phải xây dựng những phép ánh xạ, những qui tắc tổ chức phức tạp che lên tầng dữ liệu thô, nhằm đưa ra những khái niệm logic về hình thức lưu trữ khác nhau thường được gọi là *kiểu dữ liệu* . Như đã phân tích ở phần 1.1, giữa hình

thức lưu trữ dữ liệu và các thao tác xử lý trên đó có quan hệ mật thiết với nhau. Từ đó có thể đưa ra một định nghĩa cho kiểu dữ liệu như sau :

2.2.2.1 Định nghĩa kiểu dữ liệu

Kiểu dữ liệu **T** được xác định bởi một bộ **<V,O>** , với :

V : tập các giá trị hợp lệø mà một đối tượng kiểu T có thể lưu trữ

O : tập các thao tác xử lý có thể thi hành trên đối tượng kiểu T.

Ví du: Giả sử có kiểu dữ liệu **mẫu tự** = <Vc ,Oc> với

Vc = { a-z,A-Z}

Oc = { lấy mã ASCII của ký tự, biến đổi ký tự thường thành ký tự hoa…}

Giả sử có kiểu dữ liệu **số nguyên** = <Vi ,Oi> với

Vi = { -32768..32767}

Oi = { +, -, *, /, %}

Như vậy, muốn sử dụng một kiểu dữ liệu cần nắm vững cả nội dung dữ liệu được phép lưu trữ và các xử lý tác động trên đó.

Các thuộc tính của 1 KDL bao gồm:

- Tên KDL
- Miền giá trị
- Kích thước lưu trữ
- Tập các toán tử tác động lên KDL

2.2.2.2. Các kiểu dữ liệu cơ bản

Các loại dữ liệu cơ bản thường là các loại dữ liệu đơn giản, không có cấu trúc. Chúng thường là các giá trị vô hướng như các số nguyên, số thực, các ký tự, các giá trị logic ... Các loại dữ liệu này, do tính thông dụng và đơn giản của mình, thường được các ngôn ngữ lập trình (NNLT) cấp cao xây dựng sẵn như một thành phần của ngôn ngữ để giảm nhẹ công việc cho người lập trình. Chính vì vậy đôi khi người ta còn gọi chúng là các kiểu dữ liệu định sẵn.

Thông thường, các kiểu dữ liệu cơ bản bao gồm :

Kiểu có thứ tự rời rạc: số nguyên, ký tự, logic , liệt kê, miền con …

Kiểu không rời rạc: số thực

Tùy ngôn ngữ lập trình, các kiểu dữ liệu định nghĩa sẵn có thể khác nhau đôi chút. Với ngôn ngữ C, các kiểu dữ liệu này chỉ gồm số nguyên, số thực, ký tự. Và theo quan điểm của C, kiểu ký tự thực chất cũng là kiểu số nguyên về mặt lưu trữ, chỉ khác về cách sử dụng. Ngoài ra, giá trị logic ĐÚNG (TRUE) và giá trị logic SAI (FALSE) được biểu diễn trong C như là các giá trị nguyên khác zero và zero. Trong khi đó PASCAL định nghĩa tất cả các kiểu dữ

liệu cơ sở đã liệt kê ở trên và phân biệt chúng một cách chặt chẽ. Trong giới hạn giáo trình này ngôn ngữ chính dùng để minh họa sẽ là C.

Các kiểu dữ liệu định sẵn trong C gồm các kiểu sau:

Tên kiểu	**Kthước**	**Miền giá trị**	***Ghi chú***
Char	1 byte	-128 đến 127	Có thể dùng như số nguyên 1 byte có dấu hoặc kiểu ký tự
unsign char	1 byte	0 đến 255	Số nguyên 1 byte không dấu
Int	2 byte	-32738 đến 32767	
unsign int	2 byte	0 đến 65335	Có thể gọi tắt là unsign
Long	4 byte	-2^{32} đến 2^{31} -1	
unsign long	4 byte	0 đến 2^{32}-1	
Float	4 byte	3.4E-38 3.4E38	Giới hạn chỉ trị tuyệt đối.Các giá trị <3.4E-38 được coi = 0. Tuy nhiên kiểu float chỉ có 7 chữ số có nghĩa.
Double	8 byte	1.7E-308 1.7E308	
long double	10 byte	3.4E-4932 1.1E4932	

Một số điều đáng lưu ý đối với các kiểu dữ liệu cơ bản trong C là kiểu ký tự (char) có thể dùng theo hai cách (số nguyên 1 byte hoặc ký tự). Ngoài ra C không định nghĩa kiểu logic (boolean) mà nó đơn giản đồng nhất một giá trị nguyên khác 0 với giá trị TRUE và giá trị 0 với giá trị FALSE khi có nhu cầu xét các giá trị logic. Như vậy, trong C xét cho cùng chỉ có 2 loại dữ liệu cơ bản là số nguyên và số thực. Tức là chỉ có dữ liệu số. Hơn nữa các số nguyên trong C có thể được thể hiện trong 3 hệ cơ số là hệ thập phân, hệ thập lục phân và hệ bát phân. Nhờ những quan điểm trên, C rất được những người lập trình chuyên nghiệp thích dùng.

Các kiểu cơ sở rất đơn giản và không thể hiện rõ sự tổ chức dữ liệu trong một cấu trúc, thường chỉ được sử dụng làm nền để xây dựng các kiểu dữ liệu phức tạp khác.

2.2.2.3 Các kiểu dữ liệu có cấu trúc

Trong nhiều trường hợp, chỉ với các kiểu dữ liệu cơ sở không đủ để phản ánh tự nhiên và đầy đủ bản chất của sự vật thực tế, dẫn đến nhu cầu phải xây dựng các kiểu dữ liệu mới dựa trên việc tổ chức, liên kết các thành phần dữ liệu có kiểu dữ liệu đã được định nghĩa. Những kiểu dữ liệu được xây dựng như thế gọi là kiểu dữ liệu có cấu trúc. Đa số các ngôn ngữ lập trình đều cài đặt sẵn một số kiểu có cấu trúc cơ bản như mảng, chuỗi, tập tin, bản ghi...và cung cấp cơ chế cho lập trình viên tự định nghĩa kiểu dữ liệu mới.

Ví dụ : Để mô tả một đối tượng sinh viên, cần quan tâm đến các thông tin sau:

- Mã sinh viên: chuỗi ký tự
- Tên sinh viên: chuỗi ký tự
- Ngày sinh: kiểu ngày tháng
- Nơi sinh: chuỗi ký tự
- Điểm thi: số nguyên

Các kiểu dữ liệu cơ sở cho phép mô tả một số thông tin như :

```
int Diemthi;
```

Các thông tin khác đòi hỏi phải sử dụng các kiểu có cấu trúc như :

```
char masv[15];
char tensv[15];
char noisinh[15];
```

Để thể hiện thông tin về ngày tháng năm sinh cần phải xây dựng một kiểu bản ghi,

```
typedef struct tagDate
{
    char ngay;
    char thang;
    char thang;
}Date;
```

Cuối cùng, ta có thể xây dựng kiểu dữ liệu thể hiện thông tin về một sinh viên :

```
typedef struct tagSinhVien
{
    char masv[15];
    char tensv[15];
    char noisinh[15];
```

```
    int Diem thi;
}SinhVien;
```

Giả sử đã có cấu trúc phù hợp để lưu trữ một sinh viên, nhưng thực tế lại cần quản lý nhiều sinh viên, lúc đó nảy sinh nhu cầu xây dựng kiểu dữ liệu mới...Mục tiêu của việc nghiên cứu cấu trúc dữ liệu chính là tìm những phương cách thích hợp để tổ chức, liên kết dữ liệu, hình thành các kiểu dữ liệu có cấu trúc từ những kiểu dữ liệu đã được định nghĩa.

3. Thiết kế và phân tích giải thuật.

3.1. Tính hiệu quả của thuật toán.

Khi giải một vấn đề, chúng ta cần chọn trong số các thuật toán, một thuật toán mà chúng ta cho là "tốt" nhất. Vậy ta cần lựa chọn thuật toán dựa trên cơ sở nào? Một cấu trúc dữ liệu tốt phải thỏa mãn các tiêu chuẩn sau:

*__Phản ánh đúng thực tế__: Đây là tiêu chuẩn quan trọng nhất, quyết định tính đúng đắn của toàn bộ bài toán. Cần xem xét kỹ lưỡng cũng như dự trù các trạng thái biến đổi của dữ liệu trong chu trình sống để có thể chọn cấu trúc dữ liệu lưu trữ thể hiện chính xác đối tượng thực tế.

Ví dụ : Một số tình huống chọn cấu trúc lưu trữ sai :

- Chọn một biến số nguyên **int** để lưu trữ tiền thưởng bán hàng (được tính theo công thức tiền thưởng bán hàng = trị giá hàng * 5%), do vậy sẽ làm tròn mọi giá trị tiền thưởng gây thiệt hại cho nhân viên bán hàng. Trường hợp này phải sử dụng biến số thực để phản ánh đúng kết quả của công thức tính thực tế.

- Trong trường trung học, mỗi lớp có thể nhận tối đa 28 học sinh. Lớp hiện có 20 học sinh, mỗi tháng mỗi học sinh đóng học phí $10. Chọn một biến số nguyên **unsigned char** (khả năng lưu trữ 0 - 255) để lưu trữ tổng học phí của lớp học trong tháng, nếu xảy ra trường hợp có thêm 6 học sinh được nhận vào lớp thì giá trị tổng học phí thu được là $260, vượt khỏi khả năng lưu trữ của biến đã chọn, gây ra tình trạng tràn, sai lệch.

*__Phù hợp với các thao tác trên đó__**:** Tiêu chuẩn này giúp tăng tính hiệu quả của đề án: việc phát triển các thuật toán đơn giản, tự nhiên hơn; chương trình đạt hiệu quả cao hơn về tốc độ xử lý.

Ví dụ : Một tình huống chọn cấu trúc lưu trữ không phù hợp:

Cần xây dựng một chương trình soạn thảo văn bản, các thao tác xử lý thường xảy ra là chèn, xoá sửa các ký tự trên văn bản. Trong thời gian xử lý văn bản, nếu chọn cấu trúc lưu trữ văn bản trực tiếp lên tập tin thì sẽ gây khó khăn khi xây dựng các giải thuật cập nhật văn bản và làm chậm tốc độ xử lý của chương trình vì phải làm việc trên bộ nhớ ngoài. Trường hợp này nên tìm một cấu trúc dữ liệu có thể tổ chức ở bộ nhớ trong để lưu trữ văn bản suốt thời gian soạn thảo.

LƯU Ý :

Đối với mỗi ứng dụng , cần chú ý đến thao tác nào được sử dụng nhiều nhất để lựa chọn cấu trúc dữ liệu cho thích hợp.

***Tiết kiệm tài nguyên hệ thống:** Cấu trúc dữ liệu chỉ nên sử dụng tài nguyên hệ thống vừa đủ để đảm nhiệm được chức năng của nó.Thông thường có 2 loại tài nguyên cần lưu tâm nhất : CPU và bộ nhớ. Tiêu chuẩn này nên cân nhắc tùy vào tình huống cụ thể khi thực hiện đề án . Nếu tổ chức sử dụng đề án cần có những xử lý nhanh thì khi chọn cấu trúc dữ liệu yếu tố tiết kiệm thời gian xử lý phải đặt nặng hơn tiêu chuẩn sử dụng tối ưu bộ nhớ, và ngược lại.

Ví dụ : Một số tình huống chọn cấu trúc lưu trữ lãng phí:

- Sử dụng biến **int** (2 bytes) để lưu trữ một giá trị cho biết tháng hiện hành. Biết rằng tháng chỉ có thể nhận các giá trị từ 1-12, nên chỉ cần sử dụng kiểu **char** (1 byte) là đủ.

- Để lưu trữ danh sách học viên trong một lớp, sử dụng mảng 50 phần tử (giới hạn số học viên trong lớp tối đa là 50). Nếu số lượng học viên thật sự ít hơn 50, thì gây lãng phí. Trường hợp này cần có một cấu trúc dữ liệu linh động hơn mảng- ví dụ xâu liên kết

Khi ta viết một chương trình chỉ để sử dụng một số ít lần, và cái giá của thời gian viết chương trình vượt xa cái giá của chạy chương trình thì tiêu chuẩn (1) là quan trọng nhất. Nhưng có trường hợp ta cần viết các chương trình (hoặc thủ tục, hàm) để sử dụng nhiều lần, cho nhiều người sử dụng, khi đó giá của thời gian chạy chương trình sẽ vượt xa giá viết nó. Chẳng hạn, các thủ tục sắp xếp, tìm kiếm được sử dụng rất nhiều lần, bởi rất nhiều người trong các bài toán khác nhau. Trong trường hợp này ta cần dựa trên tiêu chuẩn (2). Ta sẽ cài đặt thuật toán có thể rất phức tạp, miễn là chương trình nhận được chạy nhanh hơn các chương trình khác.

Tiêu chuẩn (2) được xem là *tính hiệu quả* của thuật toán. Tính hiệu quả của thuật toán bao gồm hai nhân tố cơ bản.

1. Dung lượng không gian nhớ cần thiết để lưu giữ các dữ liệu vào, các kết quả tính toán trung gian và các kết quả của thuật toán.

2. Thời gian cần thiết để thực hiện thuật toán (ta gọi là thời gian chạy).

Chúng ta sẽ chỉ quan tâm đến thời gian thực hiện thuật toán. Vì vậy, khi nói đến đánh giá độ phức tạp của thuật toán, có nghĩa là ta nói đến đánh gia thời gian thực hiện. Một thuật toán có hiệu quả được xem là thuật toán có thời gian chạy ít hơn các thuật toán khác.

3.2. Tại sao lại cần thuật toán có hiệu quả.

Kỹ thuật máy tính tiến bộ rất nhanh, ngày nay các máy tính lớn có thể đạt tốc độ tính toán hàng trăm triệu phép tính một giây. Vậy thì có bõ công phải

tiêu tốn thời gian để thiết kế các thuật toán có hiệu quả không ? Một số ví dụ sau đây sẽ trả lời cho câu hỏi này.

Ví dụ 1 : Tính định thức.

Giả sử M là một ma trận vuông cấp n :

$$M = \begin{matrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & . & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{matrix}$$

Định thức của ma trận M ký hiệu là det(M) được xác định đệ qui như sau:

Nếu n = 1, det(M) = a_{11}. Nếu n >1, ta gọi M_{ij} là ma trận con cấp n -1, nhận được từ ma trận M bằng cách loại bỏ dòng thứ i và cột thứ j, và

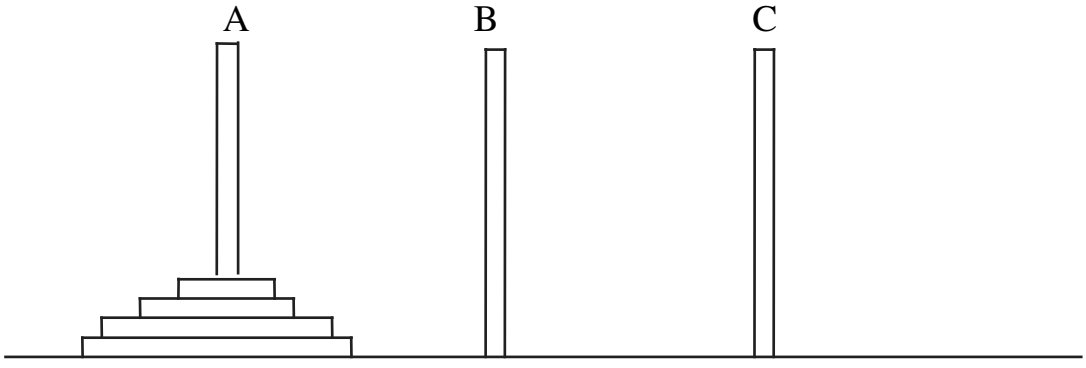
$$\det(M) \quad \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(M_{1j})$$

Dễ dàng thấy rằng, nếu ta tính định thức trực tiếp dựa vào công thức đệ qui này, cần thực hiện n! phép nhân. Một con số khổng lồ với n không lấy gì làm lớn. Ngay cả với tốc độ của máy tính lớn hiện đại, để tính định thức của ma trận cấp n = 25, cũng cần hàng triệu năm !

Một thuật toán cổ điển khác, đó là thuật toán Gauss - Jordan thuật toán này tính định thức cấp n trong thời gian n^3.

Để tính định thức cấp n = 100 bằng thuật toán này trên máy tính lớn ta chỉ cần đến 1 giây.

Ví dụ 2 : Bài toán tháp Hà Nội.



Hình 1.2. Bài toán tháp Hà Nội

Bài toán tháp Hà Nội là bài toán cần đưa ra một thuật toán để chuyển n đĩa từ cọc A sang cọc B. Ta thử tính xem, cần thực hiện bao nhiêu lần chuyển đĩa từ cọc này sang cọc khác (không đặt đĩa to lên trên đĩa nhỏ) để chuyển được n đĩa từ cọc A sang cọc B. Gọi số đó là F(n). Từ thuật toán, ta có :

F(1) = 1,

$F(n) = 2F(n-1) + 1$ với $n > 1$.

với n = 1, 2, 3 ta có F(1) = 1, F(2) = 3, F(3) = 7.

Bằng cách qui nạp, ta chứng minh được $F(n) = 2^n - 1$.

Với n = 64, ta có $F(64) = 2^{64} - 1$ lần chuyển. Giả sử mỗi lần chuyển một đĩa từ cọc này sang cọc khác, cần 1 giây. Khi đó để thực hiện $2^{64} - 1$ lần chuyển, ta cần 5×10^{11} năm. Nếu tuổi của vũ trụ là 10 tỉ năm, ta cần 50 lần tuổi của vũ trụ để chuyển 64 đĩa !.

Đối với một vấn đề có thể có nhiều thuật toán, trong số đó có thể thuật toán này hiệu quả hơn (chạy nhanh hơn) thuật toán kia. Tuy nhiên, cũng có những vấn đề không tồn tại thuật toán hiệu quả, tức là có thuật toán, song thời gian thực hiện nó là quá lớn, trong thực tế không thể thực hiện được, dù là trên các máy tính lớn hiện đại nhất.

3.3. Đánh giá thời gian thực hiện thuật toán như thế nào ?

Có hai cách tiếp cận để đánh giá thời gian thực hiện của một thuật toán.Trong phương pháp thử nghiệm, chúng ta viết chương trình và cho chạy chương trình với các dữ liệu vào khác nhau trên một máy tính nào đó. Thời gian chạy chương trình phụ thuộc vào các nhân tố chính sau đây :

1. Các dữ liệu vào

2. Chương trình dịch để chuyển chương trình nguồn thành mã máy.

3. Tốc độ thực hiện các phép toán của máy tính được sử dụng để chạy chương trình.

Vì thời gian chạy chương trình phụ thuộc vào nhiều nhân tố, nên ta không thể biểu diễn chính xác thời gian chạy là bao nhiêu đơn vị thời gian chuẩn, chẳng hạn nó là bao nhiêu giây.

Trong phương pháp lý thuyết (đó là phương pháp được sử dụng trong sách này), ta sẽ coi thời gian thực hiện thuật toán như là hàm số của *cỡ dữ liệu vào*. Cỡ của dữ liệu vào là một tham số đặc trưng cho dữ liệu vào, nó có ảnh hưởng quyết định đến thời gian thực hiện chương trình. Cái mà chúng ta chọn làm cỡ của dữ liệu vào phụ thuộc vào các thuật toán cụ thể. Đối với các thuật toán sắp xếp mảng, cỡ của dữ liệu là số thành phần của mảng. Đối với thuật toán giải hệ n phương trình tuyến tính với n ẩn, ta chọn n là cỡ. Thông thường cỡ của dữ liệu vào là một số nguyên dương n. Ta sẽ sử dụng hàm số T(n), trong đó n là cỡ dữ liệu vào, để biểu diễn thời gian thực hiện của một thuật toán.

Thời gian thực hiện thuật toán T(n) nói chung không chỉ phụ thuộc vào cỡ của dữ liệu vào, mà còn phụ thuộc vào dữ liệu vào cá biệt. Chẳng hạn, ta xét bài toán xác định một đối tượng a có mặt trong danh sách n phần tử (a_1, a_2, ... a_n) hay không. Thuật toán ở đây là, so sánh a với từng phần tử của danh sách đi từ đầu đến cuối danh sách, khi gặp phần tử a_i đầu tiên $a_i = a$ thì dừng lại,

hoặc đi đến hết danh sách mà không gặp a_i nào bằng a, trong trường hợp này a không có trong danh sách. Các dữ liệu vào là a và danh sách (a_1, a_2, ..., a_n) (có thể biểu diễn danh sách bằng mảng, chẳng hạn). Cỡ của dữ liệu vào là n. Nếu a_1 = a chỉ cần một phép so sánh. Nếu a_1 a, a_2 = a, cần 2 phép so sánh. Còn nếu a_i a, i = 1, ..., n-1 và a_n = a, hoặc a không có trong danh sách, ta cần n phép so sánh. Nếu xem thời gian thực hiện T(n) là số phép toán so sánh, ta có T(n) <= n, trong trường hợp xấu nhất T(n) = n. Trong các trường hợp như thế, ta nói đến thời gian thực hiện thuật toán trong trường hợp xấu nhất.

Ngoài ra, ta còn sử dụng khái niệm thời gian thực hiện trung bình. Đó là thời gian trung bình $T_{tb}(n)$ trên tất cả các dữ liệu vào có cỡ n. Nói chung thời gian thực hiện trung bình khó xác định hơn thời gian thực hiện trong trường hợp xấu nhất.

Chúng ta có thể xác định thời gian thực hiện T(n) là số *phép toán sơ cấp* cần phải tiến hành khi thực hiện thuật toán. Các phép toán sơ cấp là các phép toán mà thời gian thực hiện bị chặn trên bởi một hằng số chỉ phụ thuộc vào cách cài đặt được sử dụng (ngôn ngữ lập trình, máy tính ...). Chẳng hạn các phép toán số học +, - , *, /, các phép toán so sánh = , < >, <, <= , > , > = là các phép toán sơ cấp. Phép toán so sánh hai xâu ký tự không thể xem là phép toán sơ cấp, vì thời gian thực hiện nó phụ thuộc vào độ dài của xâu.

3.4 Ký hiệu ô lớn và đánh giá thời gian thực hiện thuật toán bằng ký hiệu ô lớn.

Khi đánh giá thời gian thực hiện bằng phương pháp toán học, chúng ta sẽ bỏ qua nhân tố phụ thuộc vào cách cài đặt chỉ tập trung vào xác định độ lớn của thời gian thực hiện T(n).

Giả sử n là số nguyên không âm, T(n) và f(n) là các hàm thực không âm. Ta viết T(n) = 0(f(n)), nếu và chỉ nếu tồn tại các hằng số dương c và n_o sao cho T(n) c f(n), với mọi n n_o.

Nếu một thuật toán có thời gian thực hiện T(n) = 0(f(n)), chúng ta sẽ nói rằng thuật toán có thời gian thực hiện cấp f(n).

Ví dụ : Giả sử $T(n) = 3n^2 + 5n + 4$. Ta có

$3n^2 + 5n + 4 <= 3n^2 + 5n^2 + 4n^2 = 12n^2$, với mọi n 1.

Vậy $T(n) = 0(n^2)$. Trong trường hợp này ta nói thuật toán có thời gian thực hiện cấp n^2, hoặc gọn hơn, thuật toán có thời gian thực hiện bình phương.

Dễ dàng thấy rằng, nếu T(n) = 0(f(n)) và f(n) = $0(f_1(n))$, thì $T(n)=0(f_1(n))$. Thật vậy, vì T(n) là ô lớn của f(n) và f(n) là ô lớn của $f_1(n)$, do đó tồn tại các hằng số c_o, n_o, c_1, n_1 sao cho T(n) c_o f(n) với mọi n n_o và f(n) $c_1f_1(n)$ với mọi n n_1. Từ đó ta có T(n) $c_oc_1f_1(n)$ với mọi n $\max(n_o,n_1)$.

Khi biểu diễn cấp của thời gian thực hiện thuật toán bởi hàm f(n), chúng ta sẽ chọn f(n) là hàm số nhỏ nhất, đơn giản nhất có thể được sao cho T(n) =

0(f(n)). Thông thường f(n) là các hàm số sau đây : f(n) = 1 ; f(n)=logn; f(n)=n ; f(n) = nlogn ; f(n) = n^2, n^3, ... và f(n) = 2^n.

Nếu T(n) = 0(1) điều này có nghĩa là thời gian thực hiện bị chặn trên bởi một hằng số nào đó, trong trường hợp này ta nói thuật toán có thời gian thực hiện hằng.

Nếu T(n) =0(n), tức là bắt đầu từ một n_o nào đó trở đi ta có T(n)<=cn với một hằng số c nào đó, thì ta nói thuật toán có thời gian thực hiện tuyến tính.

Để thấy rõ sự khác nhau của các cấp thời gian thực hiện thuật toán, ta xét ví dụ sau. Giả sử đối với một vấn đề nào đó, ta có hai thuật toán giải A và B. Thuật toán A có thời gian thực hiện $T_A(n) = 0(n^2)$, còn thuật toán B có thời gian thực hiện $T_B(n) = 0(nlogn)$. Với n = 1024, thuật toán A đòi hỏi khoảng 1048.576 phép toán sơ cấp, còn thuật toán B đòi hỏi khoảng 10.240 phép toán sơ cấp. Nếu cần một micrô-giây cho một phép toán sơ cấp thì thuật toán A cần khoảng 1,05 giây, trong khi thuật toán B chỉ cần khoảng 0,01 giây. Nếu n = 1024 x 2, thì thuật toán A đòi hỏi khoảng 4,2 giây, trong khi thuật toán B chỉ đòi hỏi khoảng 0,02 giây. Với n càng lớn thì thời gian thực hiện thuật toán B càng ít hơn so với thời gian thực hiện thuật toán A. Vì vậy, nếu một vấn đề nào đó đã có một thuật toán giải với thời gian thực hiện cấp n^2, bạn tìm ra thuật toán mới với thời gian thực hiện cấp nlogn, thì đó là một kết quả rất có ý nghĩa.

Những thuật toán có thời gian thực hiện cấp n^k, với k là số nguyên nào đó 1, được gọi là các thuật toán có thời gian thực hiện đa thức.

3.5 Các qui tắc để đánh giá thời gian thực hiện thuật toán.

Sau đây chúng ta đưa ra một qui tắc cần thiết về ô lớn để đánh giá thời gian thực hiện một thuật toán.

Qui tắc tổng : Nếu $T_1(n) = 0(f_1(n)$ và $T_2(n) = 0(f_2(n)$ thì

$$T_1(n) + T_2(n) = 0(max(f_1(n), f_2(n))).$$

Thật vậy, vì $T_1(n)$, $T_2(n)$ là ô lớn của $f_1(n)$, $f_2(n)$ tương ứng do đó tồn tại hằng số c_1, c_2, n_1, n_2 sao cho $T_1(n)$ $c_1f_1(n)$ với mọi n n_1 và $T_2(n)$ $c_2f_2(n)$ với mọi n n_2. Đặt n_o = max(n1, n2). Khi đó với mọi n n_o, ta có $T_1(n)$ + $T_2(n)$ $(c_1 + c_2)$ max $(f_1(n), f_2(n))$.

Qui tắc này thường được áp dụng như sau. Giả sử thuật toán của ta được phân thành ba phần tuần tự. Phần một có thời gian thực hiện $T_1(n)$ được đánh giá là 0(1), phần hai có thời gian $T_2(n)$ là $0(n^2)$, phần ba có thời gian $T_3(n)$ là 0(n). Khi đó thời gian thực hiện thuật toán $T(n) = T_1(n) + T_2(n) + T_3(n)$ sẽ là $0(n^2)$, vì n^2 = max $(1,n^2,n)$.

Trong sách báo quốc tế các thuật toán thường được trình bầy dưới dạng các thủ tục hoặc hàm trong ngôn ngữ tựa Pascal. Để đánh giá thời gian thực hiện thuật toán, ta cần biết cách đánh giá thời gian thực hiện các câu lệnh của

Pascal. Trước hết, chúng ta hãy xác định các câu lệnh trong Pascal. Các câu lệnh trong Pascal được định nghĩa đệ qui như sau :

1. Các phép gán, đọc, viết, **goto** là câu lệnh. Các lệnh này được gọi là các lệnh đơn.

2. Nếu S_1, S_2,..., S_n là câu lệnh thì

begin S_1, S_2, ..., Sn **end**

là câu lệnh. Lệnh này được gọi là lênh hợp thành (hoặc khối).

3. Nếu S_1 và S_2 là các câu lệnh và E là biểu thức logic thì

if E **then** S_1 **else** S_2

là câu lệnh, và

if E **then** S_1

là câu lệnh. Các lệnh này được gọi là lệnh **if**.

4. Nếu S_1, S_2, ..., S_{n+1} là các câu lệnh, E là biểu thức có kiểu thứ tự đếm được, và v_1, v_2,...v_n là các giá trị cùng kiểu với E thì

case E **of**

v_1 : S_1 ;

v_2 : S_2 ;

. . . . . .

v_n : S_n ;

[**else** Sn+1]

end

là câu lệnh. Lệnh này được gọi là lệnh **case**

5. Nếu S là câu lệnh và E là biểu thức logie thì

while E **do** S

là câu lệnh. Lệnh này được gọi là lệnh **while**.

6. Nếu S_1, S_2, ..., S_n là các câu lệnh, và E là biểu thức logic thì

repeat S_1, S_2, ..., S_n **until** E

là câu lệnh. Lệnh này được gọi là lệnh **repeat**.

7. Với S là câu lệnh, E_1 và E_2 là các biểu cùng một kiểu thứ tự đếm được, thì

for i : = E_1 **to** E_2 **do** S

là câu lệnh, và

for i : = E_2 **downto** E_1 do S

là câu lệnh. Các câu lệnh này được gọi là lệnh **for**.

Nhờ định nghĩa đệ qui của các lệnh, chúng ta có thể phân tích một chương trình xuất phát từ các lệnh đơn, rồi từng bước đánh giá các lệnh phức tạp hơn, cuối cùng đánh giá được thời gian thực hiện chương trình.

Giả sử rằng, các lệnh gán không chứa các lời gọi hàm. Khi đó để đánh giá thời gian thực hiện một chương trình, ta có thể áp dụng phương pháp đệ qui sau đây :

1. Thời gian thực hiện các lệnh đơn : gán, đọc, viết, **goto** là 0(1).

2. Lệnh hợp thành. Thời gian thực hiện lệnh hợp thành được xác định bởi luật tổng.

3. Lệnh **if**. Giả sử thời gian thực hiện các lệnh S_1, S_2 là $0(f_1(n))$ và $0(f_2(n))$ tương ứng. Khi đó thời gian thực hiện lệnh if là $0(\max(f_1(n), f_2(n)))$.

4. Lệnh **case**. Được đánh giá như lệnh **if**.

5. Lệnh **while**. Giả sử thời gian thực hiện lệnh S (thân của lệnh **while**) là 0(f(n)). Giả sử g(n) là số tối đa các lần thực hiện lệnh S, khi thực hiện lệnh **while**. Khi đó thời gian thực hiện lệnh **while** là 0(f(n)g(n).

6. Lệnh repeat. Giả sử thời gian thực hiện khối begin S_1, S_2, ... S_n end là 0(f(n)). Giả sử g(n) là số tối đa các lần lặp. Khi đó thời gian thực hiện lệnh repeat là 0(f(n)g(n)).

7. Lệnh for. Được đánh giá tương tự lệnh **while** và **repeat**.

Nếu lệnh gán có chứa các lời gọi hàm, thì thời gian thực hiện nó không thể xem là 0(1) được, vì khi đó thời gian thực hiện lệnh gán còn phụ thuộc vào thời gian thực hiện các hàm có trong lệnh gán. Việc đánh giá thời gian thực hiện các thủ tục (hoặc hàm) không đệ qui được tiến hành bằng cách áp dụng các qui tắc trên. Việc đánh giá thời gian thực hiện các thủ tục (hoặc hàm) đệ quy sẽ khó khăn hơn nhiều.

Đánh giá thủ tục (hoặc hàm) đệ qui.

Trước hết chúng ta xét một ví dụ cụ thể. Ta sẽ đánh giá thời gian thực hiện của hàm đệ qui sau (hàm này tính n!).

```
function   fact(n : integer) : integer ;
begin
    if n <=1 then fact : =1
            else fact  : = n * fac(n-1)
end ;
```

Trong hàm này cỡ của dữ liệu vào là n. Giả sử thời gian thực hiện hàm là T(n). Với n=1, chỉ cần thực hiện lệnh gán fact : = 1, do đó T(1) = 0(1). Với n > 1, cần thực hiện lệnh gán fact : = n*fact(n-1). Do đó, thời gian T(n) là 0(1) (để

thực hiện phép nhân và phép gán) cộng với T(n-1) (để thực hiện lời gọi đệ qui fact(n-1). Tóm lại, ta có quan hệ đệ qui sau :

T(1) = 0(1) ;

T(n) = 0(1) + T(n-1).

Thay các 0(1) bởi các hằng nào đó, ta nhận được quan hệ đệ qui sau

$T(1) = C_1$

$T(n) = C_2 + T(n-1)$

Để giải phương trình đệ qui, tìm T(n), chúng ta áp dụng phương pháp thế lặp. Ta có phương trình đệ qui

$T(m) = C_2 + T(m-1)$, với $m > 1$

Thay m lần lượt bởi 2,3,..., n - 1, n, ta nhận được các quan hệ sau.

$T(2) = C_2 + T(1)$

$T(3) = C_2 + T(2)$

.....

$T(n-1) = C_2 + T(n-2)$

$T(n) = C_2 + T(n-1)$

Bằng các phép thế liên tiếp, ta nhận được

$T(n) = (n-1)C_2 + T(1)$

hay $T(n) = (n-1)\ C_2 + C_1$, trong đó C_1 và C_2 là các hằng nào đó. Do đó, T(n)=0(n).

Từ ví dụ trên, ta suy ra phương pháp tổng quát sau đây để đánh giá thời gian thực hiện thủ tục (hàm) đệ qui. Để đơn giản, ta giả thiết rằng các thủ tục (hàm) là đệ qui trực tiếp. Điều đó có nghĩa là các thủ tục (hàm) chỉ chứa các lời gọi đệ qui đến chính nó (không qua một thủ tục (hàm) khác nào cả). Giả sử thời gian thực hiện thủ tục là T(n), với n là cỡ dữ liệu vào. Khi đó thời gian thực hiện các lời gọi đệ qui thủ tục sẽ là T(m), với m < n. Đánh giá thời gian $T(n_0)$, với n_0 là cỡ dữ liệu vào nhỏ nhất có thể được (trong ví dụ trên, đó là T(1). Sau đó đánh giá thân của thủ tục theo các qui tắc 1-7, ta sẽ nhận được quan hệ đệ qui sau đây.

$$T(n) = F(T(m_1), T(m_2), ..., T(m_k))$$

trong đó $m_1, m_2, ..., m_k < n$. Giải phương trình đệ qui này, ta sẽ nhận được sự đánh giá của T(n). Tuy nhiên, cần biết rằng, việc giải phương trình đệ qui, trong nhiều trường hợp, là rất khó khăn, không đơn giản như trong ví dụ đã trình bày.

3.6. Phân tích một số thuật toán.

Sau đây chúng ta sẽ áp dụng các phương pháp đã trình bày để phân tích độ phức tạp của một số thuật toán.

Ví dụ 1 : Phân tích thuật toán Euclid. Chúng ta biểu diễn thuật toán Euclid bởi hạm sau.

```
function  Euclid (m, n : integer) : integer ;
    var     r : integer ;
begin
(1)     r : = m  mod  n ;
(2)     while r < > 0 do
        begin
(3)         m : = n ;
(4)         n : = r ;
(5)         r : = m mod n ;
        end ;
(6)     Euclid : = n ;
end ;
```

Thời gian thực hiện thuật toán phụ thuộc vào số nhỏ nhất trong hai số m và n. Giả sử n n > 0, do đó cỡ của dữ liệu vào là n. Các lệnh (1) và (6) có thời gian thực hiện là 0(1). Vì vậy thời gian thực hiện thuật toán là thời gian thực hiện lệnh while ta đánh giá thời gian thực hiện lệnh while (2). Thân của lệnh này, là khối gồm ba lệnh (3), (4) và (5). Mỗi lệnh có thời gian thực hiện là 0(1), do đó khối có thời gian thực hiện là 0(1). Còn phải đánh giá số lớn nhất các lần thực hiện lặp khối.

Ta có :

$m = n.q_1 + r_1 , 0 \quad r_1 < n$

$n = r_1q_2 + r_2 , 0 \quad r_2 < r_1$

Nếu r_1 n/2 thì $r_2 < r_1$ n/2, do đó $r_2 < n/2$. Nếu $r_1 > n/2$ thì $q_2 = 1$, tức là $n = r_1 + r_2$, do đó $r_2 < n/2$. Tóm lại, ta luôn có $r_2 < n/2$

Như vậy, cứ hai lần thực hiện khối thì phần dư r giảm đi một nửa của n. Gọi k là số nguyên lớn nhất sao cho 2^k n. Số lần lặp khối tối đa là 2k+1 $2\log_2 n+1$. Do đó thời gian thực hiện lệnh **while** là $0(\log_2 n)$. Đó cũng là thời gian thực hiện thuật toán.

Ví dụ 2. Dãy số Fibonacci được xác định một cách đệ qui như sau :

$f_o = 0$;

$f_1 = 1$;

$f_n = f_{n-1} + f_{n-2}$ với n 2

Các thành phần đầu tiên của dãy là 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Dãy này có nhiều áp dụng trong toán học, tin học và lý thuyết trò chơi.

Thuật toán đệ qui :

```
function  Fibo1 (n : integer) : integer ;
begin
    if n <= 2 then Fibo1 : = n
            else  Fibo1 : = Fibo1(n-1) + Fibo1(n-2)
end ;
```

Có thể đánh giá được hàm này có thời gian thực hiện là 0(^{n}), với =(1+ $\sqrt{5}$)/2. Tức là, thuật toán Fibo1 có thời gian thực hiện mũ, nó không có ý nghĩa thực tiễn với n lớn. Sau đây là một thuật toán khác, có thời gian thực hiện chỉ là 0(n).

```
funciton Fibo2 (n: integer) : integer ;
    var i, j, k : integer
    begin
(1)     i : = 1 ;
(2)     j : = 0 ;
(3)     for k : = 1 to n do
        begin
            j : = i + j ;
            i : = j - i ;
        end ;
(4)     Fibo2 : = j ;
    end ;
```

Ta phân tích hàm Fibo2. Các lệnh gán (1) , (2) và (4) có thời gian thực hiện 0(1). Thân của lệnh for(3) có thời gian thực hiện là 0(1), số lần lặp là n. Do đó lệnh for(3) có thời gian thực hiện là 0(n). Kết hợp lại, ta có thời gian thực hiện hàm Fibo2 là 0(n).

Với n = 50, thuật toán Fibo1 cần khoảng 20 ngày trên máy tính lớn, trong khi đó thuật toán Fibo2 chỉ cần khoảng 1 micrô giây. Với n = 100, thời gian chạy của thuật toán Fibo1 là 10^9 năm ! còn thuật toán Fibo2 chỉ cần khoảng 1,5 micro giây. Thuật toán Fibo2 chưa phải là thuật toán hiệu quả nhất. Bạn thử tìm một thuật toán hiệu quả hơn.

4. Một số ví dụ về thiết kế và phân tích giải thuật.

Ví dụ 1: Một chương trình quản lý điểm thi của sinh viên cần lưu trữ các điểm số của 3 sinh viên. Do mỗi sinh viên có 4 điểm số ứng với 4 môn học khác nhau nên dữ liệu có dạng bảng như sau:

Sinh viên	Môn 1	Môn 2	Môn3	Môn4
SV 1	7	9	5	2
SV 2	5	0	9	4
SV 3	6	3	7	4

Chỉ xét thao tác xử lý là xuất điểm số các môn của từng sinh viên.

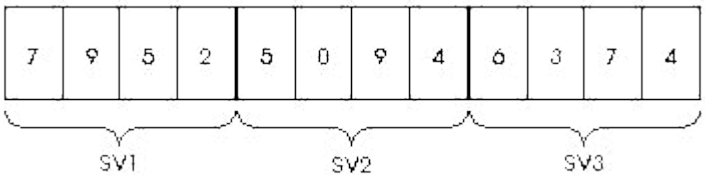
Giả sử có các phương án tổ chức lưu trữ sau:

Phương án 1 : Sử dụng mảng một chiều

Có tất cả 3(SV)*4(Môn) = 12 điểm số cần lưu trữ, do đó khai báo mảng *result* như sau :

```
int result [ 12 ] = {7, 9, 5, 2,
5, 0, 9, 4,
6, 3, 7, 4};
```

khi đó trong mảng *result* các phần tử sẽ được lưu trữ như sau:



Và truy xuất điểm số môn j của sinh viên i - là phần tử tại (dòng i, cột j) trong bảng - phải sử dụng một công thức xác định chỉ số tương ứng trong mảng result:

bảngđiểm(dòng i, cột j) result[((i-1)*số cột) + j]

Ngược lại, với một phần tử bất kỳ trong mảng, muốn biết đó là điểm số của sinh viên nào, môn gì, phải dùng công thức xác định sau

result[i] bảngđiểm (dòng((i / số cột) +1), cột (i % số cột))

Với phương án này, thao tác xử lý được cài đặt như sau :

```
void XuatDiem() //Xuất điểm số của tất cả sinh viên
{
    const int so_mon = 4;
    int sv,mon;
    for (int i=0; i<12; i+)
```

```
        {
            sv = i/so_mon;
            mon = i % so_mon;
            printf("Điểm môn %d của sv %d là: %d", mon, sv,result[i]);
        }
    }
```

Phương án 2 : Sử dụng mảng 2 chiều

Khai báo mảng 2 chiều *result* có kích thước 3 dòng* 4 cột như sau :

```
int result[3][4] ={{ 7, 9, 5, 2},
{ 5, 0, 9, 4},
{ 6, 3, 7, 4 }};
```

khi đó trong mảng *result* các phần tử sẽ được lưu trữ như sau :

	Cột 0	Cột 1	Cột 2	Cột 3
Dòng 0	result[0][0] =7	result[0][1] =9	result[0][2] =5	result[0][3] =2
Dòng 1	result[1][0] =5	result[1][1] =0	result[1][2] =9	result[1][3] =4
Dòng 2	result[2][0] =6	result[2][1] =3	result[2][2] =7	result[2][3] =4

Và truy xuất điểm số môn j của sinh viên i - là phần tử tại (dòng i, cột j) trong bảng - cũng chính là phần tử nằm ở vị trí (dòng i, cột j) trong mảng

bảngđiểm(dòng i,cột j) result[i] [j]

Với phương án này, thao tác xử lý được cài đặt như sau :

```
void XuatDiem() //Xuất điểm số của tất cả sinh viên
{
int so_mon = 4, so_sv =3;
for ( int i=0; i<so_sv; i+)
    for ( int j=0; i<so_mon; j+)
    printf("Điểm môn %d của sv %d là: %d", j, i,result[i][j]);
}
```

NHẬN XÉT

Có thể thấy rõ phương án 2 cung cấp một cấu trúc lưu trữ phù hợp với dữ liệu thực tế hơn phương án 1, và do vậy giải thuật xử lý trên cấu trúc dữ liệu của phương án 2 cũng đơn giản, tự nhiên hơn.

Chương 2: **Các kiểu dữ liệu nâng cao**

1. Dữ liệu kiểu mảng

1.1. Khái niệm dữ liệu kiểu mảng

Một mảng dữ liệu bao gồm một số hữu hạn phần tử có cùng kiểu gọi là kiểu cơ bản. Số phần tử của mảng được xác định ngay từ định nghĩa ra mảng. Mỗi phần tử của mảng được truy nhập trực tiếp thông qua tên mảng cùng với chỉ dẫn truy nhập được để giữa hai ngoặc vuông [ ].

Định nghĩa kiểu mảng T có kiểu các phần tử (KPT), có kiểu chỉ dẫn (KCD) để hướng dẫn cách tổ chức mảng cũng như cách truy nhập vào các thành phần của mảng được viết trong Pascal với các từ khóa như sau:

```
Type
    T = Array [KCD] of KPT;
```

Khi đó, việc khai báo một biến A có kiểu dữ liệu là kiểu mảng có thể được viết như sau:

```
Var
    A: T;
```

Hoặc khai báo trực tiếp biến A cùng với kiểu của mảng trong phần khai báo biến không có định nghĩa trong phần TYPE:

```
Var
    A : Array [KCD] of KPT;
```

Ví dụ kiểu khai báo cho dữ liệu kiểu mảng:

```
Type
    AI = Array [1..10] of integer;
    AC = Array [1..10] of char;
    Color = (Red, Blue, Green, White, Black);
Var
    A, B, C: AI;
    X, Y: AC;
    M1, M2: Array [-3..5] of real;
    MC: Array ['A'..'Z'] of integer;
    MM: Array [Color] of Boolean;
```

AI, AC là hai kiểu mảng gồm 10 phần tử được đánh số thứ tự từ 1 đến 10 thông qua kiểu chỉ dẫn là một đoạn con các số nguyên 1..10. Các phần tử của

kiểu AI có kiểu là số nguyên, còn các phần tử của AC có kiểu là các kí tự A, B, C,... là các biến có kiểu là AI.

M1, M2 là 2 biến được định nghĩa luôn khi khai báo. Đây là hai biến mảng gồm 9 phần tử là các số thực, được đánh số từ -3 đến 5.

MC là một biến mảng gồm 26 số nguyên được đánh số qua các chỉ dẫn là các chữ cái từ 'A' đến 'Z'.

MM là một mảng gồm 5 phần tử kiểu Boolean, các phần tử được đánh dấu qua các chỉ dẫn là tên của 5 mầu sắc.

Một điều lưu ý là khi khai báo mảng, kiểu chỉ dẫn chỉ có thể là các kiểu đơn giản sau: kí tự (như biến MC), đoạn con (đoạn con integer như các biến AI, AC), kiểu liệt kê do người sử dụng định nghĩa ra như biến MM và kiểu Boolean. Kiểu chỉ dẫn không được là kiểu Real hoặc Integer, nghĩa là không được viết:

```
X: Array [Integer] of Integer;
Y: Array [Real] of Integer;
```

Việc truy nhập vào 1 phần tử nào đó của mảng được thực hiện qua tên biến mảng, theo sau là giá trị chỉ dẫn để trong ngoặc vuông như:

```
MM [Red] := True;
MC ['B'] := 5;
```

Do thời gian truy nhập vào một phần tử của mảng không phụ thuộc vào giá trị của chỉ dẫn nên cấu trúc mảng thuộc kiểu cấu trúc truy nhập trực tiếp.

Ví dụ: Sắp xếp dãy số bất kì theo thứ tự:

(Giả sử dãy số gồm 5 chứ số cần được sắp xếp theo thứ tự tăng dần)

```
Program SAP_XEP;
Const
   N = 5;
Var
   MI: Array[1..N] of Integer;
   T: Integer;            (* T: biến trung gian *)
   i, j: Integer;
Begin
   (* Đọc các số cần sắp xếp vào mảng MI *)
   For i := 1 to N do
```

```
        Begin
            Write (' M[ ',i,'] = ');
            Readln (MI[i]);
        End;
    (* Sắp xếp *)
    For i := 1 to N-1 do
        For j := i+1 to N do
            Begin
                If MI[i] > MI[j] then
                (* Đổi chỗ 2 phần tử cho nhau *)
                    Begin
                        T := MI[i];
                        MI[i] := MI[j];
                        MI[j] :=T;
                    End;
    (* In ra kết quả *)
    Writeln;
    Writeln ('Sau khi sap xep: ');
    For i := 1 to N do Writeln (MI[i] :6);
End.
```

1.2. Mảng nhiều chiều .

Kiểu phần tử của mảng không bị hạn chế chiều như kiểu chỉ dẫn. Nó còn có thể là các kiểu có cấu trúc. Ví dụ sau cho thấy việc khai báo một mảng có các phần tử cũng là mảng.

Ví dụ:

```
Type
    PT: Array [1..5] of real;              (*PT: Kiểu phần tử*)
    Color = (Red, Blue, Green, White, Black);
Var
    MPT: Array [1..3] of PT;               (*MPT: Mảng phần tử*)
    Z: Array [1..3, 'A'..'C'] of Color;
```

Hoặc viết một lần như sau:

```
Var
    MPT: Array [1..3] of Array [1..5] of Real;
```

Hoặc thường được viết gọn lại:

```
Var
    MPT: Array [1..3, 1..5] of Real;
```

MPT được định nghĩa như trên chính là ma trận 3 chiều với 3 hàng và 5 cột, ta có thể minh họa ma trận này như sau:

		CỘT				
		1	2	3	4	5
HÀNG	1	[1,1]	[1,2]	[1,3]	[1,4]	[1,5]
	2	[2,1]	[2,2]	[2,3]	[2,4]	[2,5]
	3	[3,1]	[3,2]	[3,3]	[3,4]	[3,5]

Việc truy nhập đối với mảng có định nghĩa phức tạp như MPT được tiến hành qua 2 cách:

- Quan 2 lần đóng mở ngoặc vuông. Thí dụ: MPT [3] [5] biểu diễn phần tử cuối cùng của mảng MPT.

- Hoặc thường được viết đơn giản trong một ngoặc vuông với các chỉ dẫn cách nhau bằng dấu phẩy: MPT [3,5]

Cách viết MPT [i] [j] và MPT [i, j] là tương đương nhau. Mảng được định nghĩa như trên có thể hiểu là ma trận nhiều chiều. Phần tử MPT [i, j] sẽ là phần tử hàng thứ i, cột thứ j của MPT.

Ví dụ: Chương trình nhân 2 ma trận vuông cấp N:

$$C = A * B$$

Phần tử của ma trận tích được tính theo công thức:

$$C_{ij} \sum_{k=1}^{N} A_{ik} * B_{kj}$$

```
Program Nhan_ma_tran;
Const
    N = 3;
    Dau_phay = ‘,’; (*Hằng ký tự: Dấu phẩy *)
Var
    A, B, C: Array [1..N, 1..N] of integer;
    i, j, k: integer;
```

```
Begin
    (* Nhập vào giá trị của ma trận A *)
    For i := 1 to N do              (* Chỉ số hàng *)
        For j := 1 to N do          (* Chỉ số cột *)
            Begin
                Write (‘ A[ ‘,i, Dau_phay , j,’] = ‘);
                Readln (A[i,j]);
            End;
    (* Nhập vào giá trị của ma trận B *)
    For i := 1 to N do              (* Chỉ số hàng *)
        For j := 1 to N do          (* Chỉ số cột *)
            Begin
                Write (‘ B[ ‘,i, Dau_phay , j,’] = ‘);
                Readln (B[i,j]);
            End;
    (* Nhân 2 ma trận vuông cấp N: C = A*B *)
    For i := 1 to N do
        For j := 1 to N do
            Begin
                C[i, j] = 0;
                For k := 1 to N do
                    C[i, j] := C[i, j] + A[i, k] * B[k, j];
            End;
    (* In kết quả ra màn hình *)
    Writeln (‘Tich cua 2 ma tran = ‘);
    For i := 1 to N do
        Begin
            For j := 1 to N do write (C[i, j] :5);
            Writeln;
        End;
    End.
```

2. Dữ liệu kiểu bản ghi.

2.1. Khái niệm dữ liệu kiểu bản ghi.

Cấu trúc dữ liệu kiểu mảng được tạo ra bằng một tập hợp các phần tử có cùng kiểu, để tạo ra một kiểu cấu trúc dữ liệu mới với các phần tử dữ liệu có kiểu khác nhau nhưng có liên kết với nhau, người ta định nghĩa ra **Bản ghi** (hay còn gọi là **thẻ ghi, phiếu ghi**; tiếng Anh là **Record**). Nói 1 cách khác để mô tả cùng một đối tượng bằng các phần tử dữ liệu có mô tả kiểu khác nhau, chúng ta phải dùng cấu trúc kiểu Record, đó là phương tiện hiệu quả nhất để xây dựng kiểu dữ liệu mới.

Mô tả kiểu RECORD được viết bằng chữ RECORD, theo sau là danh sách mô tả các phần tử dữ liệu của RECORD mà ta gọi là các trường. Mỗi trường có một tên trường và được dùng để chọn các phần tử dữ liệu của RECORD, tiếp theo là mô tả kiểu của trường, mô tả RECORD bao giờ cũng được kết thúc bằng END;

Để mô tả 1 kiểu T có cấu trúc RECORD với danh sách các trường có tên là s1, s2, ..., sn và có các mô tả kiểu tương ứng là T1, T2, ..., Tn, ta có cách viết tổng quát sau:

```
Type
   T = Record
      s1: T1;
      s2: T2;
      .....
      sn: Tn;
   End;
```

Ví dụ: Một địa chỉ bao gồm các dữ liệu như số nhà, tên phố, thành phố. Ta có thể mô tả Record Dia_chi như sau:

```
Type
   Dia_chi = Record
      So_nha: integer;
      Pho: String[20];
      Thanh_pho: String[15];
   End;
```

Như vậy ta có 3 trường là So_nha, Pho, Thanh_pho với kiểu khác nhau và được liên kết với nhau để mô tả địa chỉ.

Ví dụ: Mô tả thời gian DATE ta có 3 trường là ngày, tháng và năm:

```
Type
    Date = Record
        Ngay: 1..31;
        Thang: 1..12;
        Nam: integer;
    End;
```

Hoặc theo tiếng Anh, người ta hay dùng tháng với tên gọi của tháng:

```
Type
    Date = Record
        Day: 1..31;
        Month: (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Otc, Nov, Dec);
        Year: integer;
    End;
```

Ví dụ: Để mô tả nhân sự hay phiếu cán bộ, ta phải dùng các trường Họ tên, Ngày sinh, Chỗ ở, Lương, ...

```
Type
    Nhan_su = Record
        Ho_ten: String[30];
        Ngay_sinh: Record
            Ngay: 1..31;
            Thang: 1..12;
            Nam: integer;
        End;
    Gioi_tinh: (Nam, Nu);
    Cho_o: Dia_chi;
    Luong: Real;
End;
```

2.2. Sử dụng RECORD.

Để thâm nhập vào 1 trường của Record, ta phải dùng tên biến kiểu Record, sau đó là dấu (.) và tên trường của Record.

Ví dụ: Với mô tả kiểu Nhan_su ở trên, ta tiếp tục khai báo biến và thực hiện một đoạn chương trình để đọc dữ liệu vào các biến như sau:

```
Var
    Nguoi1, Nguoi2: Nhan_su;
Begin
    Write  ('Ho va ten nguoi 1: ');
    Readln    (Nguoi1.Ho_ten);
    Write  ('Ngay sinh: ')
    Readln    (Nguoi1.Ngay_sinh.Ngay);
    Write  ('Thang sinh: ')
    Readln    (Nguoi1.Ngay_sinh.Thang);
    Write  ('Nam sinh: ')
    Readln    (Nguoi1.Ngay_sinh.Nam);
    Write  ('Cho o: ')
    Readln    (Nguoi1.Cho_o.So_nha);
    Write  ('Pho: ')
    Readln    (Nguoi1.Cho_o.Pho);
    Write  ('Thanh pho: ')
    Readln    (Nguoi1.Ngay_sinh.Thanh_pho);
    ...
End.
```

Tuy nhiên cần lưu ý là không thể dùng các thao tác sau:

- Viết ra mà hình hoặc đọc từ bàn phím cả 1 biến kiểu Record như:

Readln (Nguoi1); hoặc Writeln (Nguoi1);

- So sánh các Record bằng các phép toán quan hệ <, >, <=, >=. Riêng các phép toán so sánh <> (khác nhau) và = (bằng nhau) thì có thể được dùng với 2 biến có cùng 1 kiểu Record.

3. Dữ liệu kiểu tập tin.

3.1. Khái niệm dữ liệu kiểu 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 thực tế 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.

Tập tin là một tập hợp các dữ liệu liên quan với nhau và có cùng kiểu được nhóm lại với nhau tạo thành 1 dãy. Tập tin theo nghĩa rộng có thể là chương trình, số liệu, có thể là các dữ khác như kí tự, văn bản,...

Tập tin là 1 kiểu dữ liệu có cấu trúc, định nghĩa của tệp có phần nào giống mảngở chỗ chúng đều là tập hợp của các phần tử dữ liệu có cùng kiểu, song mảng được định nghĩa và khai báo trong chương trình với số phần tử xác định, còn số phần tử của tập tin không được xác định khi định nghĩa.

Định nghĩa 1 kiểu tập tin T với các phần tử có kiểu là KPT được viết trong phần mô tả kiểu với từ khóa FILE OR như sau:

```
Type
    T = File of KPT
```

Sau đó khai báo một biến tập tin (FileVar) trong phần khai báo biến:

```
Var
    Bien_tep: T;
```

Hoặc khai báo trực tiếp một biến tệp với mô tả kiểu:

```
Var
    Bien_tep: File of KPT;
```

Ví dụ: Một số định nghĩa kiểu tệp:

```
Type
    FileInteger = File of integer;
    FileReal = File of real;
    Nhan_su = Record
            Ten: String[30];
            Tuoi: byte;
            Luong: Real;
        End;
    FNhan_su = File of Nhan_su;
```

```
Var
    F1, F2: FileInteger;     (* F1, F2 là 2 biến tệp có các phần tử là số nguyên *)
    F3: FileReal;            (* F3 là biến tệp kiểu số thực *)
    FNS: FNhan_su;           (* FNS: Tệp các bản ghi nhân sự *)
    F5: File of Char;
    F6: File of Array[1..5] of integer;
```

Trong ví dụ:

- F6 là biến tệp được khai báo trực tiếp trong phần VAR với các phần tử là mảng 1 chiều, độ dài bằng 5.

- FileInteger là kiểu tệp có các phần tử là các số nguyên.

- FileReal là kiểu tệp có các phần tử là các số thực.

Kiểu của phần tử của tệp có thể là bất kì kiểu dữ liệu nào (kiểu vô hướng, kiểu có cấu trúc như mảng, bản ghi,...) trừ kiểu tệp (nghĩa là không có kiểu tệp của tệp).

3.2. Mở tệp mới để cất dữ liệu.

Chương trình chỉ có thể cất dữ liệu vào tệp sau khi ta làm thủ tục mở tệp.

3.2.1. Mở tệp để ghi.

Để mở tệp và ghi dữ liệu ta dùng 2 cặp thủ tục đi liền nhau theo thứ tự:

```
Assign (Biến_tệp, Tên_tệp);
Rewrite (Biến_tệp);
```

Hay bằng tiếng Anh:

```
Assign (FileVar, FileName);
Rewrite (FileVar);
```

Ví dụ: Gán tên tệp NGUYENTO.DAT cho biến F1:

```
Assign (F1, 'NGUYENTO.DAT');
Rewrite (F1);
```

Tên tệp (FileName) là tên của tệp đặt trong thiết bị nhớ ngoài được đưa vào dưới dạng là một xâu kí tự (String). Ta thường đặt tên sao cho tên tệp phản ánh được ý nghĩa, bản chất hoặc nội dung của tệp.

Tên tệp theo quy định chung gồm có 2 phần cách nhau bằng dấu (.), phần thứ nhất là tên riêng thể hiện ý nghĩa, nội dung của tệp, phần thứ 2 là phần mở rộng gồm 3 chữ cái thường nói lên loại tệp.

3.2.2. Ghi các giá trị vào tệp với thủ tục WRITE.

Thủ tục Write sẽ đặt các giá trị mới vào tệp, cách sử dụng:

```
Write (FileVar, Item1,..., ItemN);
```

Trong đó các Item có thể là hằng, các biến, các biểu thức, tất nhiên các Item phải có giá trị cùng kiểu với phần tử của tệp.

Ví dụ:

```
Write (F1, 3, i + 2*j, k, 5);
```

Bước cuối cùng của việc đặt dữ liệu vào tệp là đóng tệp lại bằng thủ tục:

```
Close (FileVar);
```

Ví dụ: Tạo một tệp chứa các số nguyên từ 1 đến 100 với tên tệp trên bộ nhớ ngoài là NGUYEN.DAT.

```
Program Tep_nguyen;
    Var
        i: integer;
        F: File of integer;
    Begin
        Assign (F, 'NGUYEN.DAT');
        Rewrite (F);
        For i := 1 to 100 do write(F, i);
        Close (F);
    End.
```

3.3. Đọc dữ liệu từ tệp đã có.

Tệp gồm 2 loại, tệp truy cập tuần tự (sequential access) và tệp truy cập trực tiếp (direct access). Với tệp truy cập trực tiếp ta có thể truy cập đến vị trí bất kỳ của tệp tại mọi thời điểm, nhưng với tệp truy cập tuần tự ta không thể vừa đọc vừa ghi dữ liệu cùng lúc, bạn chỉ có thể đọc được dữ liệu sau khi ghi dữ liệu đã được ghi vào tệp rồi đóng lại.

3.3.1. Mở tệp để đọc.

Khi mở tệp ra để đọc ta dùng 2 thủ túc sau đi liền với nhau theo thứ tự:

```
Assign (FileVar, FileName);
Reset (FileVar);
```

Khi chạy chương trình sẽ đọc phần tử dữ liệu tại vị trí cửa sổ đang trỏ, sau lệnh RESET, nếu tệp không rỗng thì cửa sổ bao giờ cũng trỏ vào phần tử đầu tiên của tệp và chương trình sẽ copy phần tử của tệp được trỏ sang biến đệm cửa sổ.

3.3.2. Đọc dữ liệu từ tệp:

Việc đọc các phần tử từ tệp ra sau khi mở tệp được thực hiện bằng thủ tục READ:

Read (FileVar, Var1, Var2, ..., VarN); Read (Biến_tệp, các biến);

Thực hiện: Đọc các giá trị tại vị trí cửa sổ đang trỏ (nếu có) gán sang biến tương ứng cùng kiểu, sau đó cửa sổ dịch chuyển sang vị trí tiếp theo và đọc giá trị cho biến khác, cứ thế đọc đến biến cuối cùng.

Việc đọc một phần tử của tệp còn cần có điều kiện là phải xem tệp có còn phần tử không, tức là cửa sổ chưa trỏ đến EOF **(End Of File)**, con trỏ ở cuối tệp thì EOF(FileVar) = True, nếu không thì EOF(FileVar) = False, do vậy trước khi thực hiện thao tác gì để đọc tệp gán cho biến X cần phải xem tệp đó đã kết thúc chưa bằng câu lệnh:

```
If not EOF(FileVar) then read(FileVar, X);
```

hoặc nếu muốn đọc tất cả các phần tử của tệp thì dùng:

```
While not EOF(FileVar) do
    Begin
        Read (FileVar, X);
        ....
    End;
```

Ví dụ: Đọc tất cả các phần tử của 1 tệp các số nguyên nào đó và ghi ra màn hình giá trị các số nguyên đó và cuối cùng là ghi ra các phần tử của tệp. Tên tệp sẽ được xác định lúc chạy chương trình chứ không có ngay từ lúc lập trình:

```
Program Doc_tep;
Var
    i: integer;
    Sophantu: integer;
    FI: File of integer;
```

```
    FileName: String[20];
Begin
    Write ('Tentep chua cac so nguyen: ');
    Readln (FileName);
    Assign (Fl, FileName);
    Reset (Fl);
    Sophantu := 0;
    While not EOF (Fl) do
        Begin
            Read (Fl, i);       (* Đọc 1 phần tử tệp ra biến i *)
            Writeln (i);
            Sophantu := Sophantu + 1;(* Đếm số phần tử *)
        End;
    Close (Fl);
Writeln('So phan tu cua tep ',FileName,' la  ',Sophantu);
End.
```

4. Dữ liệu kiểu con trỏ

4.1. Khái niệm dữ liệu kiểu con trỏ.

Các kiểu dữ liệu như Array, Record là kiểu dữ liệu tĩnh bởi chúng đã được xác định trước 1 cách rõ ràng lúc mô tả và khai báo biến, sau đó chúng được dùng thông qua tên của chúng (nên địa chỉ các biến tĩnh cũng được xác định ngay khi gọi chương trình dịch). Thời gian tồn tại của các biến tĩnh cũng là thời gian tồn tại của khối chương trình có chứa khai báo biến này.

Tùy theo yêu cầu, các biến động có thể được tạo ra lúc chạy chương trình, các biến động này không có tên (vì việc đặt tên thực chất là gán cho nó một địa chỉ xác định). Việc tạo ra biến động và xóa nó đi được thực hiện nhờ các thủ tục NEW và DISPOSE đã có sẵn. Việc truy nhập các biến động được tiến hành nhờ các **biến con trỏ**. Các biến con trỏ được định nghĩa như biến tĩnh (nghĩa là có tên và được khai báo ngay từ đầu) và được dùng để chứa địa chỉ các biến động.

Kiểu con trỏ Tp và biến con trỏ Ptr tương ứng được dùng để chứa địa chỉ các biến động có kiểu là T được định nghĩa như sau:

```
Type
    Tp = ^T;          (* Mô tả kiểu con trỏ Tp *)
```

```
Var
    Ptr: Tp;        (* Khai báo biến Ptr *)
```

Để thâm nhập vào biến động có địa chỉ nằm trong biến con trỏ Ptr ta dùng ký hiệu Ptr^.

Ví dụ:

```
Type
    IntPointer = ^Integer;
Var
    Ptr: IntPointer;
```

hoặc ta có thể khai báo trực tiếp:

```
Var
    Ptr: ^integer;
```

Ví dụ:

```
Type
    Point = ^Nhan_su;
    Nhan_su = Record
            Ten: string [30];
            Tuoi: integer;
Var
    P: ^Nhan_su
```

Trong ví dụ trên **P** là biến con trỏ, **^Nhan_su** là biến động hay biến được trỏ:

4.2. Các thao tác đối với kiểu dữ liệu con trỏ.

4.2.1. Các thao tác đối với con trỏ.

Chỉ có một số thao tác sau giữa các biến con trỏ:

- Phép gán (:=),

- Phép so sánh bằng nhau (=) và so sánh khác nhau (<>).

Các giá trị của biến con trỏ không thể đọc vào từ bàn phím hay in ra trực tiếp trên màn hình, máy in, tức là không thể dùng read(P) hoặc write(P).

4.2.2. Hằng con trỏ NIL.

NIL là một giá trị hằng đặc biệt dành cho các biến con trỏ và nó được dùng để báo rằng con trỏ không trỏ vào đâu cả, NIL có thể được đem gán cho bất cứ biến con trỏ nào. Đương nhiên khi đó việc thâm nhập vào biến động thông qua con trỏ có giá trị NIL là vô nghĩa, thực chất NIL là con trỏ đặc biệt chứa giá trị 0.

4.2.3. Cách tạo ra và giải phóng ô nhớ của biến động.

Để tạo ra biến động P^ do con trỏ P trỏ tới ta viết:

```
New (P);
```

Sau câu lệnh này, bộ nhớ dành cho một biến động sẽ được sắp xếp. Nếu trong 1 chương trình ta dùng n lần NEW(P) liên tục thì cuối cùng ta có n biến động song cỏntỏ P sẽ chỉ trỏ vào biến động được tạo ra lần cuối cùng.

Muốn thu hồi ô nhớ dành cho biến động đã được tạo ra sau lệnh NEW người ta dùng chương trình con DISPOSE:

```
Dispose (P);
```

Ví dụ: Cách sử dụng New và Dispose:

```
Type
    Tp = ^Mang;
    Mang = Array [1 .. 100] of integer;
Var
    P, Q: Tp;
Begin
    .........
    New (P);
    New (Q);
    .........
    Dispose (P);
    Dispose (Q);
    .........
    New (P);
    .........
    Dispose (P);
End;
```

Chương 3: **Danh sách**

1. Các khái niệm cơ bản về cấu trúc dữ liệu.

1.1. Cấp phát bộ nhớ.

Cấp phát bộ nhớ là 1 trong các nhiệm vụ của chương trình dịch, căn cứ vào mục khai báo biến và khai báo kiểu dữ liệu mà chương trình dịch sẽ thực hiện cấp phát bộ nhớ tương ứng.

1.1.1. Các loại cấp phát bộ nhớ.

Có 2 phương pháp cấp phát bộ nhớ:

- Cấp phát tĩnh bộ nhớ: Là việc cấp phát bộ nhớ chỉ thực hiện 1 lần trước khi thực hiện chương trình, ví dụ như việc cấp phát bộ nhớ cho các biến được khai báo trong chương trình chính.

- Cấp phát động bộ nhớ: Thực hiện cấp phát bộ nhớ nhiều lần xen kẽ khi thực hiện chương trình theo nguyên tắc cần đến đâu cấp phát đến đó và sử dụng song phải thu hồi ngay, ví dụ như cấp phát bộ nhớ cho các biến cục bộ của chương trình con.

1.1.2. Các phương pháp lưu trữ thông tin trong bộ nhớ.

- Lưu trữ kế tiếp: Có những kiểu dữ liệu mà giá trị thuộc kiểu dữ liệu này được lưu trữ trong nhiều hơn 1 đơn vị bộ nhớ cơ sở, nếu giá trị của 1 biên được lưu trữ như vậy thì chương trình dịch phải cấp phát cho biến này 1 vùng bộ nhớ có địa chỉ liên tiếp, sau đó lấy địa chỉ đầu của vùng bộ nhớ đã cấp phát gắn cho tên biến. Vì dùng 1 vùng bộ nhớ có địa chỉ liên tiếp nên phương pháp này gọi là lưu trữ kế tiếp.

- Lưu trữ móc nối:

Ví dụ: Khi ta ghi nội dung của 1 file lên đĩa, hệ điều hành sẽ kiểm tra toàn bộ đĩa để tìm ra những vùng bộ nhớ còn trống để ghi nội dung của file lên vùng bộ nhớ trống đó. Vì vùng bộ nhớ nằm rải rác trên đĩa lên khi ghi nội dung sẽ phải liên kết các vùng rải rác trên đĩa để chỉ ra đó là nội dung của cùng 1 file. Tại các vùng lưu trữ, ngoài phần bộ nhớ dành để lưu trữ thông tin của file ta cần thêm 1 phần nhớ để chứa địa chỉ của vùng lưu trữ thông tin tiếp theo, vùng địa chỉ đó gọi là mối nối hay địa chỉ liên kết (link, Ptr).

Với cách thức lưu trữ này ta nói việc lưu trữ nội dung của 1 file lên đĩa là theo phương pháp lưu trữ móc nối.

Cặp thuật ngữ "cấp phát tĩnh – lưu trữ kế tiếp" và "cấp phát động – lưu trữ móc nối" thường đi liền với nhau trong cấu trúc dữ liệu và giải thuật.

1.2. Cấp phát tĩnh bộ nhớ và lưu trữ kế tiếp.

1.2.1. Yêu cầu đối với người lập trình:

- Phải dự kiến tối đa dữ liệu (không gian nhớ) được sử dụng.

- Mỗi khi có yêu cầu truy nhập thì người lập trình phải cung cấp các tham số để chương trình dịch xác định địa chỉ tại đó lưu trữ giá trị của các biến, các tham số này hoàn toàn tùy thuộc vào kiểu dữ liệu của các biến.

1.2.2. Yêu cầu đối với chương trình dịch.

- Căn cứ vào khai báo biến và kiểu, chương trình dịch phải dành cho mỗi biến 1 vùng bộ nhớ có địa chỉ liên tiếp theo kích thước tối đa mà người lập trình đã khai báo, sau đó mang địa chỉ đầu của vùng bộ nhớ này gắn cho tên biến.

- Căn cứ vào các tham số mà người lập trình đã cung cấp khi có yêu cầu truy nhập dựa theo quy luật (f) nào đó để xác định địa chỉ của vùng bộ nhớ chứa giá trị cần lấy ra. Quy luật này có tên là hàm địa chỉ, nó có dạng tổng quát sau:

f(thành phần thứ i) = A_0 + tổng bộ nhớ dành để lưu trữ mọi thành phần được lưu trữ trước thành phần thứ i

Ví dụ: Mảng 1 chiều x được khai báo tối đa Nmax phần tử:

```
x: Array [1..NMax] of KPT;
```

Căn cứ kiểu phần tử của mảng ta sẽ xác định được số byte lưu trữ của 1 phần tử là C. Hàm f(x[i]) dùng để xác định địa chỉ phần tử thứ i trong mảng x:

$$f(x[i]) = A_0 + (i-1)C$$

1.2.3. Ưu nhược điểm của phương pháp.

- Ưu điểm: Cho phép truy nhập nhanh và bình đẳng đối với mọi thành phần trong cấu trúc dữ liệu.

- Nhược điểm: Tốn bộ nhớ vì yêu cầu cấp phát bộ nhớ với kích thước tối đa sẽ gây ra thiếu bộ nhớ giả tạo (bộ nhớ được cấp phát nhưng không dùng hết, hoặc do yêu cầu 1 vùng nhớ có địa chỉ liên tiếp với kích thước tối đa nên nếu có vùng bộ nhớ còn trống nhỏ hơn yêu cầu thì cùng không cấp phát được)

1.2.4. Sử dụng cấp phát tĩnh bộ nhớ và lưu trữ kế tiếp.

Phương pháp này được sử dụng khi số lượng phần tử trong 1 cấu trúc dữ liệu ít biến đổi và khi có yêu cầu truy nhập là yêu cầu trực tiếp hay ngẫu nhiên chứ không phải là toàn bộ các phần tử trong cấu trúc dữ liệu.

1.3. Cấp phát động bộ nhớ và lưu trữ móc nối.

1.3.1. Yêu cầu đối với ngôn ngữ lập trình và chương trình dịch.

- Phải có kiểu dữ liệu mà giá trị của nó là địa chỉ (kiểu dữ liệu con trỏ).

- Chương trình dịch phải cho các thủ tục mẫu để thực hiện cấp phát và giải phóng bộ nhớ.

- Phải có tên hằng để mô tả địa chỉ liên kết rỗng hay địa chỉ liên kết không xác định để dùng cho khối thông tin cuối cùng trong 1 cấu trúc dữ liệu.

Thủ tục mẫu để cấp phát bộ nhớ được ký hiệu là **Avail**, trong đó:

- Vùng bộ nhớ do biến con trỏ P trỏ đến được cấp phát động do sử dụng thủ tục xin cấp phát bộ nhớ cho biến con trỏ P: Avail P

- Để giải phóng bộ nhớ dùng ký hiệu: Avail P

1.3.2. Yêu cầu đối với người lập trình.

- Toàn quyền quản lý bộ nhớ đối với 1 cấu trúc dữ liệu: Cần phải biết được địa chỉ của phần tử đầu tiên trong cấu trúc dữ liệu hoặc trong 1 biến trỏ hoặc trong 1 phần tử giả gọi là phần tử đầu và địa chỉ phần tử đầu tiên được lưu trữ trong phần địa chỉ liên kết của phần tử giả này. Mỗi khi truy nhập 1 phần tử trong 1 cấu trúc dữ liệu chỉ có cách là truy nhập tuần tự do biết địa chỉ của phần tử đầu tiên, sao đó căn cứ vào địa chỉ của phần tử đầu tiên ta biết được địa chỉ của phần tử thứ 2 và cứ như vậy cho đến khi kết thúc.

1.3.3. Ưu, nhược điểm của phương pháp.

- Ưu điểm: Tiết kiệm bộ nhớ và không gây hiện tượng thiếu bộ nhớ giả tạo.

- Nhược điểm: Truy nhập chậm và không bình đẳng đối với các thành phần trong 1 cấu trúc dữ liệu do phương pháp truy nhập là tuần tự.

1.3.4. Sử dụng.

Phương pháp được sử dụng khi số lượng các phần tử thay đổi liên tục và mỗi khi có yêu cầu truy nhập thì phải truy nhập tuần tự.

2. Danh sách tuyến tính.

2.1. Định nghĩa và các khái niệm.

2.1.1. Định nghĩa.

Danh sách tuyến tính là 1 tập hợp các phần tử và được đánh số từ 1..n. Với n = 0 ta nói danh sách tuyến tính rỗng (cạn), với n > 0 thì với 1 phần tử

thứ i ta luôn xác định được duy nhất 1 phần tử đứng liền trước và 1 phần tử đứng liền sau (i – 1 và i + 1)

2.1.2. Các phép xử lý thường gặp với danh sách tuyến tính.

- Thống kê;
- Sắp xếp;
- Tìm kiếm;

} Chỉ liên quan đến 1 danh sách tuyến tính

- Bổ sung;

} Phương pháp này làm thay đổi số lượng phần tử hiện có trong danh sách nên phải quan tâm đến vị trí phần tử được loại bỏ hay bổ sung.

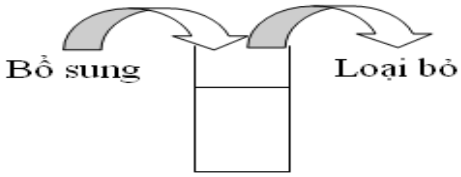
- Loại bỏ;
- Tách;

} Liên quan đến từ 2 danh sách tuyến tính trở lên

- Ghép;

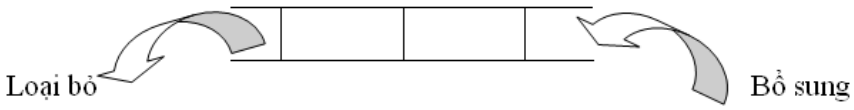
2.1.3. Các loại danh sách tuyến tính đặc biệt:

- Stack: Là danh sách tuyến tính đặc biệt mà phép bổ sung và loại bỏ được thực hiện cùng 1 phía của danh sách (phía đỉnh danh sách). Danh sách này còn có tên LIFO (Last In First Out)



Hình 3.1. Ngăn xếp Stack

- Queue: Là danh sách tuyến tính đặc biệt mà phép bổ sung và loại bỏ được thực hiện ở 1 đầu gọi là lối sau và phép loại bỏ được thực hiện ở đầu còn lại gọi là lối trước. Danh sách này còn có tên FIFO (First In First Out)



Hình 3.2. Hàng đợi Queue

2.2. Lưu trữ danh sách tuyến tính bằng phân bố kế tiếp.

Nguyên tắc chung: Để lưu trữ danh sách tuyến tính người ta dùng mảng 1 chiều, số lượng phần tử trong danh sách và kích thước tối đa được khai báo bằng khai báo của mảng 1 chiều.

2.2.1. Stack.

Stack được lưu trữ bằng 1 mảng S nào đó dạng:

```
S: Array [1..NMax] of KPT;
```

- Thuật toán bổ sung phần tử cho Stack:

Với S là mảng 1 chiều vừa là tham số vào vừa là tham số ra; n là đỉnh của Stack; X là phần tử cần bổ sung vào Stack, ta có thuật toán bổ sung 1 phần tử vào Stack như sau:

```
Procedure Push (S, n, X);          (* X: Tham số vào *)
    Begin
        If n = NMax then
            Write ('Stack tràn!');
        Else
            Begin
                n := n +1;
                S [n] := X;
            End;
    End;
```

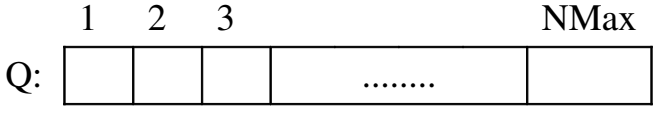
- Thuật toán loại bỏ phần tử khỏi Stack:

```
Procedure Pop (S, n, X);       (* X: Tham số ra *)
    Begin
        If n = 0 then
            Write ('Stack cạn!');
        Else
            Begin
                X := S[n];
                n := n - 1;
            End;
    End;
```

2.2.2. Queue.

Queue cũng được lưu trữ dưới dạng 1 mảng Q nào đó:

```
Q: Array [1..NMax] of KPT;
```



Gọi: F là chỉ số phần tử nằm ở lối trước của Queue;

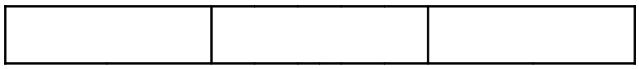
R là chỉ số phần tử nằm ở lối sau của Queue;

- Khi Queue rỗng ta quy ước F = R = 0

- Khi bổ sung lần đầu tiên thì F = R = 1, đây là trường hợp duy nhất làm thay đổi chỉ số F khi thực hiện phép bổ sung.

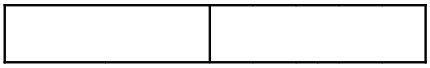
- Những lần bổ sung tiếp theo thì R := R +1.

Sau khi thực hiện 1 số phép bổ sung và loại bỏ thì hình ảnh bộ nhớ cho Q sẽ có dạng:



F R

Hình ảnh bộ nhớ này của Q sẽ luôn có dạng như vậy cho đến khi ta sử dụng đến phần tử cuối cùng của mảng Q (R = Nmax)



F R = Nmax

Nếu chỉ số R < Nmax thì khi bổ sung ta có R := R +1, ngược lại thì R := 1. Tức là:

```
If R < Nmax then R := R + 1
    Else R := 1;
```

hoặc tương đương với: R := R Mod Nmax + 1

Quy luật thay đổi chỉ số F cũng tương tự như chỉ số R, chú ý khi R Mod Nmax + 1 = F thì dẫn đến hiện tượng tràn bộ nhớ của Q, khi loại bỏ phần tử cuối cùng của Q ta có F = 0; R = 0.

* Thuật toán bổ sung phần tử vào Queue:

```
Procedure QInsert (Q, F, R, x);
Begin
    If R Mod Nmax + 1 = F then
        Writeln ('Queue tràn')
    Else
        Begin
            R := R Mod Nmax + 1;
            Q[R] := x;
            If F = 0 then F := 1;
        End;
End;
```

* Thuật toán loại bỏ phần tử khỏi Queue:

```
Procedure QDelete (Q, F, R, x);
Begin
    If F = 0 then
        Writeln ('Queue cạn')
    Else
        Begin
            x := Q[F];
        If F = R then
                Begin
                    F := 0;
                    R := 0;
                End;
        Else
            R := R Mod Nmax + 1;
        End;
End;
```

2.2.3. Các ứng dụng trên danh sách tuyến tính.

a. Dùng Stack để đổi 1 số nguyên dương từ hệ 10 sang hệ 2.

* Phân tích giải thuật:

- Dùng Stack S để lưu trữ phần dư của phép chia n cho 2

- While n <> 0: đẩy phần dư của phép chia n cho 2 vào stack S đỉnh k sau đó lấy phần nguyên của phép chia chia tiếp cho 2. Quá trình này cứ lặp đi lặp lại.

- k <> 0 thì lấy i từ stack S đỉnh k để đưa i ra.

* Giải thuật:

```
Procedure Doi n (n)
    Begin
        R := 0;
        While n <> 0 do
            Begin
                Push (S, R, n Mod 2);
                n = n div 2;
```

```
        End;
    Write (‘ Kết quả’);
    While k <> 0 do
        Begin
            Pop (S, k, i);
            Write (i);
        End;
End;
```

b. Dùng Queue để đổi 1 số 0 < x <1 từ hệ 10 sang hệ 2.

* Phân tích giải thuật: Mang giá trị x nhân liên tiếp với 2, giữ lại phần nguyên rồi lại lấy phần lẻ nhân với 2 cho đến khi x = 0 hoặc nhân đến lúc số lần nhân bằng số lượng số lẻ mà máy tính có thể biểu diễn được, số lượng số lẻ đó quyết định Nmax trong Queue, nếu Nmax càng lớn thì sai số càng hạn chế.

Kết quả chuyển đổi là phần nguyên của các tích mà ta nhận được viết theo đúng thứ tự.

```
Procedure Doi_co_so (x);
    Begin
        F := 0; R := 0;
        While (x <> 0) and (R < Nmax) do
        Begin
            R := Trunc (x * 2);
            CQInsert (Q, F, R, k);
            x := x*2 – trunc(x*2)
        End;
        Write (‘0’);
        While F <> 0 do
            Begin
                CQDelete (Q, F, R, i);
                Write (i);
            End;
End;
```

c. Dùng Stack để tính giá trị biểu thức dạng hậu tố .

*** Biểu thức dạng hậu tố:** Các thành phần tham gia vào biểu thức có thể bao gồm các toán hạng (TH) và dấu phép toán ( ). Khi tính toán phép toán ta cần quan tâm đến thứ tự thực hiện phép toán.

Chú ý: Khi tính giá trị của biểu thức ta luôn xác định phép toán thực hiện cuối cùng, căn cứ vào các quy ước về thứ tự thực hiện phép toán. Trong cách viết thông thường ta phải dùng các dấu ngoặc để thực hiện thứ tự phép toán nếu nó ngược với thứ tự quy ước.

Ví dụ: x + y * (z – t) – u/v

Biểu thức viết dạng như trên gọi là biểu thức dạng trung tố. Với cách sử dụng dấu ngoặc như vậy thì khi viết chương trình tính toán giá trị của biểu thức cần thêm dung lượng bộ nhớ để lưu dấu ngoặc, để giảm sự cồng kềnh này, 1 nhà toán học người Balan đã đưa ra phương pháp biểu diễn biểu thức theo ký pháp hậu tố (Poitfix Notation) hoặc tiền tố (Postfix Notation) mà gọi chung là ký pháp Balan (Polish Notation)

Ví dụ:

Biểu thức dạng trung tố	Biểu thức dạng hậu tố	Biểu thức dạng tiền tố
(A +B) * C	A B + C *	+ A B * C
A + B * C	A B C * +	+ A * B C

Trong đa số các ngôn ngữ lập trình, các biểu thức vẫn được viết dưới dạng trung tố thông thường, nhưng khi việc tính toán trong máy thì lại được chương trình dịch tự động chuyển sang dạng hậu tố hoặc tiền tố để tính giá trị của biểu thức.

* Quy ước về biểu diễn biểu thức dưới dạng hậu tố:

- Dấu phép toán đặt sau các toán hạng tham gia phép toán này:

TH TH

TH1 TH2 TH1 TH 2

- Dấu phép toán thực hiện cuối cùng khi tính giá trị của biểu thức luôn nằm bên phải nhất.

* Các bước biến đổi biểu thức dạng trung tố thành biểu thức dạng hậu tố:

- Phân tích biểu thức để xác định dấu phép toán thực hiện cuối cùng khi tính giá trị, Nếu các toán hạng tham gia vào phép toán này là biểu thức con của biểu thức ban đầu thì tiếp tục phân tích các biểu thức con đó để xác định dấu

phép toán được thực hiện cuối cùng để tính giá trị biểu thức con đó. Phép toán cứ thế được thực hiện cho đến khi tất cả mọi toán hạng đều là nguyên tố (là hằng hoặc là biến)

- Thực hiện quy ước 1 để chuyển dấu phép toán ra sau các toán hạng tương ứng của nó. Biểu thức thu được ta gọi là biểu thức dạng hậu tố.

Ví dụ: Chuyển biểu thức x + y * (z – t) – u/v sang biểu thức dạng hậu tố:

x	+	y	*	(	z	-	t	)	-	u	/	v
x	+	y	*	(	z	-	t	)	u	/	v	-
x	y	*	(	z	-	t	)	+	u	v	/	-
x	y	(	z	-	t	)	*	+	u	v	/	-
x	y	z	t	-	*	+	u	v	/	-		

* Giải thuật chuyển biểu thức từ dạng trung tố sang biểu thức dạng hậu tố:

Procedure Polish (Q, P);

{Q là biểu thức viết dưới dạng trung tố, giải thuật này biến đổi Q sang dạng hậu tố P, thoạt đầu P rỗng}

1. Thêm dấu ")" vào cuối Q; {để làm dấu kết thúc}

Call Push(S,T, "("); {Dấu ngoặc mở này tương ứng với dấu ngoặc đóng mới thêm vào cuối Q}

2. **Repeat** {Đọc ký tự X trong Q khi duyệt từ trái qua phải}

3. **Case**

X là toán hạng: Bổ sung thêm X vào P;

X là dấu ngoặc mở: **Call** Push (S, T, "(";

X là toán tử: **While** thứ tự ưu tiên của S[T] thứ tự ưu tiên của X **do Begin**

Call POP (S, T, W); {Bổ sugn thêm W vào P}

End;

Call Push(S, T, X);

X là dấu ngoặc đóng: **reapeat**

Call POP(S, T, W); {Bổ sung thêm W vào P};

Until gặp dấu "(" thì loại dấu "(" ra khỏi Stack S

End Case

Until Stack rỗng

4. **Return**;

* Giải thuật tính giá trị của biểu thức dạng hậu tố:

Procedure Eval (P, Val);

1. Ghi thêm dấu ")" vào cuối biểu thức P để làm kết thúc;
2. **Repeat** {Đọc ký tự X trong P khi duyệt từ trái sang phải}
3. **If** X là toán hạng **then**

   Call Push (S, T, X)
4. **Else Begin**

   Call POP (S, T, Y);

   Call POP (S, T, Z);

   W := Z (X) Y; {(X) chỉ toán tử X}

   Call PUSH (S, T, W)

   End;

Until gặp dấu kết thúc ")";

5. **Call** POP (S, T, Val);
6. **Return**;

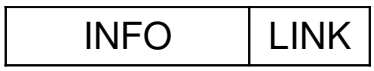
2.3. Lưu trữ móc nối với danh sách tuyến tính.

Lưu trữ kế tiếp đối với danh sách tuyến tính đã bộc lộ rõ nhược điểm trong trường hợp thực hiện thường xuyên các phép bổ sung hoặc loại bỏ phần tử, trường hợp xử lý đồng thời nhiều danh sách,... bởi việc luôn khai báo Nmax sẽ gây hiện tượng tốn bộ nhớ bởi nhiều khi không dùng hết, hoặc nếu muốn bổ sung hay loại bỏ phần tử không phải ở đầu hoặc cuối danh sách sẽ đòi hỏi phải dồn hoặc dãn danh sách trước khi thực hiện bổ sung hoặc loại bỏ làm chậm lại quá trình xử lý thông tin.

Việc sử dụng con trỏ hoặc mối nối để tổ chức danh sách tuyến tính, mà ta gọi là danh sách móc nối chính là 1 giải pháp để khắc phục các nhược điểm của lưu trữ kế tiếp.

2.3.1. Nguyên tắc lưu trữ.

Mỗi phần tử trong danh sách được lưu trữ trong 1 phần tử nhớ, gọi là nút (node) và có thể nằm ở vị trí bất kỳ trong bộ nhớ, trong mỗi nút, ngoài phần thông tin ứng với 1 phần tử còn chứa địa chỉ của phần tử đứng sau nó trong danh sách, cấu trúc của 1 nút:



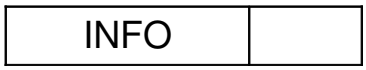
Trong đó:

Info: Chứa thông tin của phần tử.

Link: Chứa địa chỉ của nút tiếp theo

- Để chỉ nút đầu tiên của danh sách có thể sử dụng biến trỏ L hoặc dùng 1 nút giả để ký hiệu rằng đó là nút đầu tiên của danh sách:

- Riêng nút cuối cùng thì không có nút đứng sau nó lên mối nối ở nút này là 1 "địa chỉ đặc biệt" chỉ dùng để đánh dấu kết thúc danh sách chứ không như các địa chỉ ở các nút khác, ta gọi là "mối nối không" và ký hiện là null



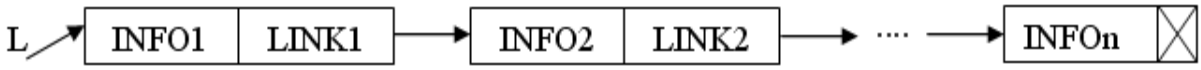
- Để có thể truy nhập được vào mọi nút trong danh sách ta cũng bắt đầu truy nhập vào nút đầu tiên, nghĩa là cần có 1 con trỏ L trỏ vào nút đầu tiên này.

- Để chỉ mối nối từ danh nút này sang nút kia của danh sách ta dùng mũi tên xuất phát từ phần LINK của nút trước đi tới phần INFO của nút tiếp theo

2.3.2. Các loại danh sách tuyến tính sử dụng lưu trữ móc nối.

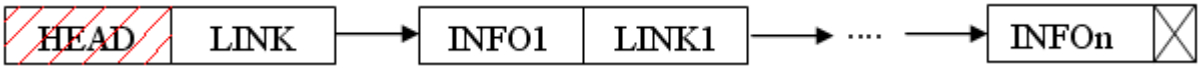
2.3.2.1. Danh sách nối đơn.

- Dạng 1: Ghi nhớ địa chỉ nút đầu tiên bằng con trỏ L:



Danh sách sẽ rỗng khi L = Null.

- Dạng 2: Sử dụng nút giả để đánh dấu nút đầu tiên của danh sách:



Danh sách sẽ trống khi Link (Head) = Null.

Với cấu trúc của danh sách dạng 1 và dạng 2 này đều có chung nhược điểm là muốn duyệt tất cả các phần tử trong danh sách phải xuất phát từ phần tử đầu tiên, để khắc phục nhược điểm này ta sử dụng cấu trúc lưu trữ là danh sách nối vòng.

2.3.2.2 Danh sách nối vòng.

- **Dạng 3:** Với phương pháp lưu trữ của danh sách nối vòng, để duyệt danh sách ta có thể bắt đầu từ bất kỳ phần tử nào. Người ta ghi nhớ địa chỉ

của một trong các nút trong danh sách trong 1 con trỏ Tp, nút có địa chỉ chứa trong con trỏ này được gọi là nút cực phải của danh sách nối vòng.

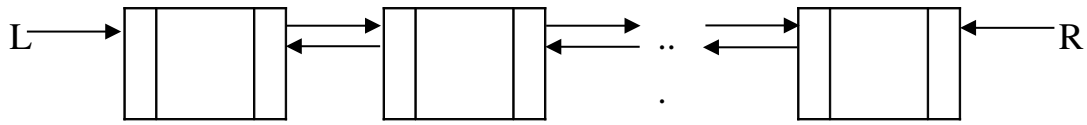
Danh sách này rỗng khi con trỏ Tp = Null.

- Dạng 4:

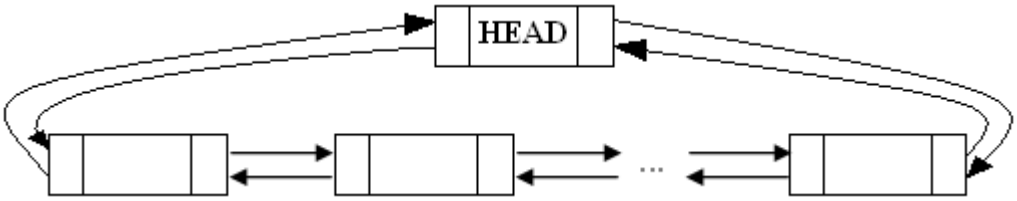
2.3.2.3. Danh sách nối kép.

Với 4 loại cấu trúc lưu trữ trên đều có chung 1 nhược điểm là chỉ duyệt danh sach theo 1 chiều, để khắc phục nhược điểm này người ta thay cấu trúc nút bằng cách thêm vào 1 địa chỉ liên kết nữa:

- **Dạng 5:** Để có thể duyệt danh sách chiều từ trái sang phải dùng con trỏ L để ghi nhận địa chỉ của nút nằm bên trái nhất, để duyệt danh sách theo chiều từ phải sang trái dùng con trỏ R để ghi nhận địa chỉ của nút nằm bên phải nhất, danh sách rỗng khi L = R = Null



- Dạng 6:



Với dạng danh sách này thì danh sách rỗng khi LLink (Head) = RLink (Head) = Head

2.3.3. Các phép toán trên danh sách tuyến tính sử dụng lưu trữ móc nối.

2.3.3.1. Các phép toán trên danh sách liên kết đơn.

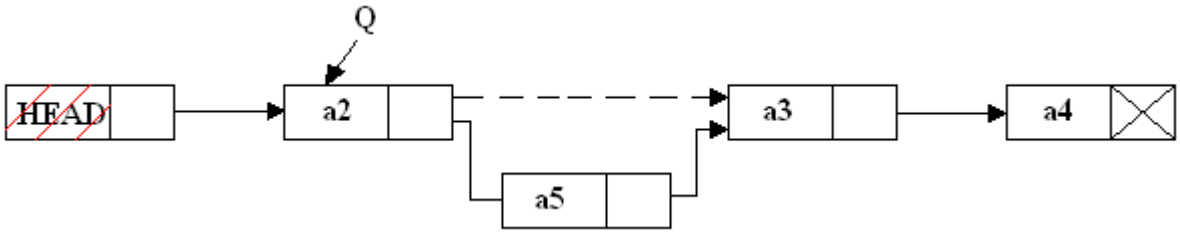
* Bổ sung 1 nút mới vào danh sách liên kết đơn.

Danh sách liên kết đơn chỉ đầy khi không còn không gian nhớ để cấp phát, giả sử luôn còn không gian nhớ nghĩa là phép bổ sung 1 phần tử vào danh sách luôn thực hiện được.

Giả sử Q là 1 con trỏ trỏ vào 1 nút nào đó của danh sách, ta cần bổ sung 1 nút mới với Info là X vào sau nút được trỏ bởi Q, ta có giải thuật:

```
Procedure InsertAfter (Var L: Pointer, Q: Pointer, X: Item);
    Var
        P: Pointer;
    Begin
        New(P);                          {Tạo một nút
        P^.Info := X;
        If L = Null then
            Begin
                P^.Link := Null;         {Thực hiện bổ sung, nếu danh sách rỗng thì bổ sung nút mới vào thành nút đầu tiên, nếu không thì bổ sung nút mới vào sau nút được trỏ bởi Q}
                Head := P;
            End
            Else Begin
                P^.Link := Q^.Link;
                Q^.Link := P;
                End;
    End;
```

Thao tác bổ sung nút mới có thể mô tả bằng hình vẽ:



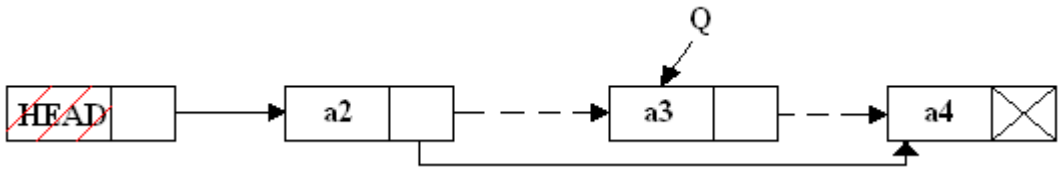
* Loại bỏ 1 nút ra khỏi danh sách liên kết đơn:

Giả sử ta cần loại bỏ nút được trỏ bởi Q, nếu Q không trỏ vào nút đầu tiên thì ta cần xác định được nút đứng trước Q để đưa con trỏ R sang đó rồi mới loại bỏ nút Q:

```
Procedure Delete (Var L: Pointer, Q: Pointer, X: Item);
    Var
```

```
        R: Pointer;
    Begin
        {Trường hợp danh sách rỗng}
        If L = Null then
        Begin
            Writeln ('Danh sach rong');
            Exit;
        End;
        {Trường hợp nút trỏ bởi Q là nút đầu tiên}
        If Q = Head then
            Begin
                Head := Q^.Link;
                Dispose (Q);
                Exit;
            End;
        {Tìm đến nút đứng trước nút trỏ bởi Q}
        R := Head;
        While R^.Link <> Q do R := R^Link;
        {Loại bỏ nút trỏ bởi Q}
        R^.Link := Q^.Link;
        Dispose (Q);
    End;
```

Phép toán loại bỏ trên có thể mô tả bằng hình vẽ:



* Ghép 2 danh sách liên kết đơn.

Giả sử có 2 danh sách liên kết đơn lần lượt được trỏ bởi P và Q, 2 danh sách này được ghép lại để trở thành 1 danh sách được trỏ bởi P:

```
    Procedure Combine (Var P: Pointer, Q: Pointer);
        Var
            R: Pointer;
```

```
Begin
    {Trường hợp danh sách trỏ bởi Q rỗng}
    If Q = Null then Exit;
    {Trường hợp danh sách trỏ bởi P rỗng}
    If P = Null then
        Begin
            P := Q;
            Exit;
        End;
    {Tìm đến nút cuối danh sách trỏ bởi P}
    R := P;
    While R^.Link <> Null do
        R := R^Link;
    {Ghép 2 danh sách}
    R^.Link := Q;
End;
```

2.3.3.2. Các phép toán trên danh sách liên kết đôi.

Để truy nhập vào danh sách liên kết đôi ta dùng 2 con trỏ, con trỏ L trỏ vào nút đầu tiên của danh sách, con trỏ R trỏ vào nút cuối cùng của danh sách. Việc cài đặt danh sách liên kết đôi sẽ tốn nhiều bộ nhớ hơn so với danh sách liên kết đơn nhưng lại có nhiều ưu điểm hơn như có thể duyệt danh sách theo chiều tiến hoặc lùi lại tùy ý, các phép toán trên danh sách liên kết đôi cũng thực hiện dễ dàng hơn.

Ta có thể khai báo cấu trúc dữ liệu danh sách liên kết đôi như sau:

```
Type
    TRO = ^Nut;
    Nut = Record
        Info: Item;
        L, R: TRO;
    End;
```

* Bổ sung 1 nút mới vào danh sách: Cho 2 con trỏ L, R lần lượt trỏ tới nút đầu và cuối của danh sách liên kết đôi, M là con trỏ trỏ tới 1 nút trong danh sách này, thực hiện bổ sung 1 nút mới vào nút trước nút được trỏ bởi M:

```
Procedure Bo_sung;
    Var
        L, P, P, M: TRO;
        X: Item;
    Begin
    {Tạo nút mới}
    New (P);
    P^.Info := X;
    {Xét trường hợp danh sách rỗng}
    If (R = Null) then
        Begin
            P^.LLink := Null;
            P^.RLink := Null;
            L := P;
            R := P;
        End
        Else
        {Trường hợp M trỏ tới nút đầu tiên}
            If (M = L) then
                Begin
                    P^.LLink := Null;
                    P^.RLink := M;
                    M^.LLink := P;
                End
            Else
                Begin
                    P^.LLink := M^.LLink;
                    P^.RLink := M;
                    M^.LLink := P;
                    P^.LLink^.RLink := P;
                End;
    End;
```

* Loại bỏ 1 nút trên danh sách:

Cho 2 con trỏ L, R lần lượt trỏ tới nút đầu và cuối của danh sách liên kết đôi, M là con trỏ trỏ tới 1 nút trong danh sách này, thực hiện loại bỏ nút được trỏ bởi M:

```
Procedure Loai_bo;
    Var
        L, R, M: TRO;
    Begin
        If R = Null then write ('Danh sach rong')
        Else
            If L = R then
                Begin
                    L := Null;
                    R := Null;
                End
            Else
                If M = L then
                    Begin
                        L := L^.RLink;
                        L^.LLink := Null;
                    End
                Else If M = R then
                    Begin
                        R := R^.LLink;
                        R^.RLink := Null;
                    End
                Else
                    Begin
                        M^.LLink^.RLink := M^.RLink;
                        M^.RLink^.LLink := M^.LLink;
                    End;
    End;
```

2.4. Lưu trữ móc nối với Stack và Queue.

Đối với Stack, việc truy cập chỉ thực hiện ở 1 đầu nên việc cài đặt Stack bằng danh sách liên kết là khá tự nhiên. Chẳng hạn, với danh sách liên kết đơn được trỏ bởi Head thì có thể coi như đây là đỉnh của Stack, bổ sung 1 nút mới chính là bổ sung 1 nút thành nút đầu tiên của danh sách, loại bỏ nút đầu tiên chính là loại bỏ nút đầu tiên của danh sách.

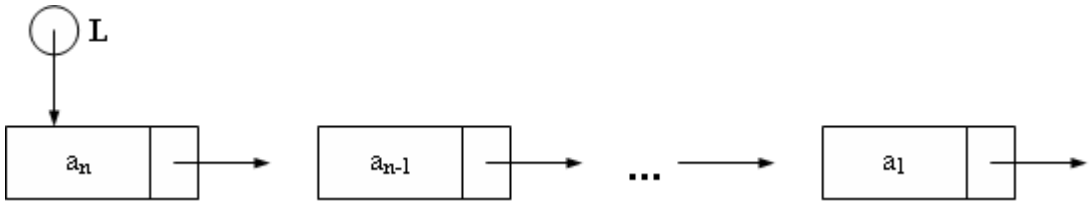
Đối với Queue thì việc loại bỏ được tiến hành ở 1 đầu và có thể coi là nút đầu tiên của danh sách, bổ sung thì tiến hành ở đầu còn lại, tức là ta gắn thêm nút vào phía sau nút cuối cùng, trong trường hợp này người ta dùng 2 con trỏ để trỏ luôn vào vị trí đầu và vị trí cuối của danh sách để thuận tiện cho bổ sung và loại bỏ.

2.4.1. Các phép toán đối với Stack.

Giả sử con trỏ L trỏ vào đỉnh của Stack, cấu trúc dữ liệu của Stack được khai báo như sau:

```
Type
    Tro = ^Node;
    Node = Record
        Info: Item;
        Link: Tro;
    End;
Var L: Tro;
```

Trong trường hợp này Stack rỗng khi L = Null. Giả sử việc cấp phát bộ nhớ là động, tức là Stack không bao giờ đầy và ta luôn thực hiện được phép toán bổ sung.



Sau đây là các hàm và thủ tục thực hiện các phép toán trên ngăn xếp Stack:

* Khởi tạo Stack rỗng:

```
Procedure Khoi_tao;
    Var
        L: Tro;
    Begin
```

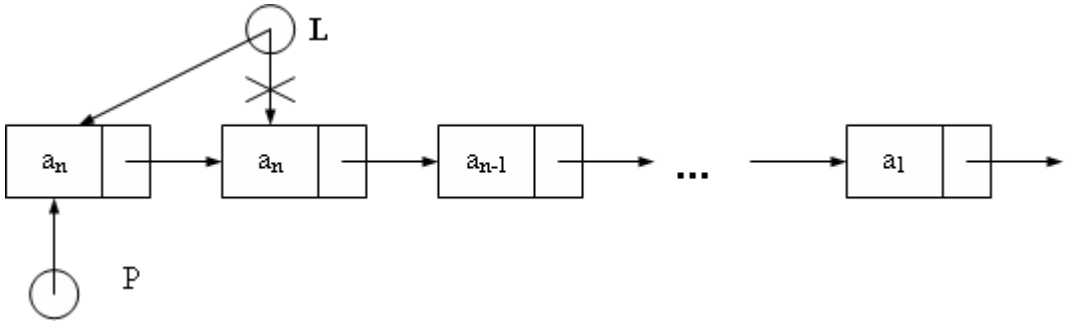
```
        L := Null;
    End;
```

* Kiểm tra Stack rỗng:

```
Function Empty (L: Tro): Boolean;
    Begin
        Empty := (L = Null);
    End;
```

* Bổ sung 1 phần tử vào đỉnh của Stack:

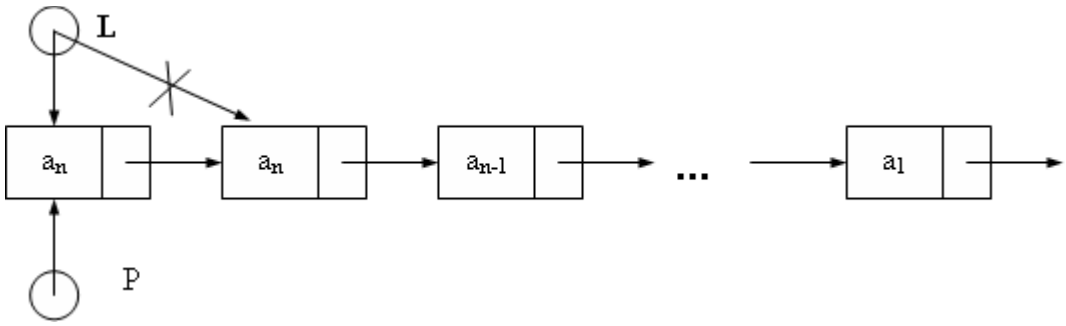
```
Procedure Push (L, X);
    Var
        P: Tro;
    Begin
        New(P);
        P^.Info := X;
        P^.Link := Null;
        If L = Null then
            L := P
        Else
            Begin
                P^.Link := L;
                L := P;
            End;
    End;
```



* Lấy 1 phần tử ở đỉnh ra khỏi Stack:

```
Procedure Pop(L, X);
    Var
```

```
        OK: Boolean;
        P: Tro;
    Begin
        If Empty(L) then OK := False
        Else
            Begin
                P := L;
                X := L^.Info;
                L := L^Link;
                OK := True;
                Dispose(P);
            End;
    End;
```



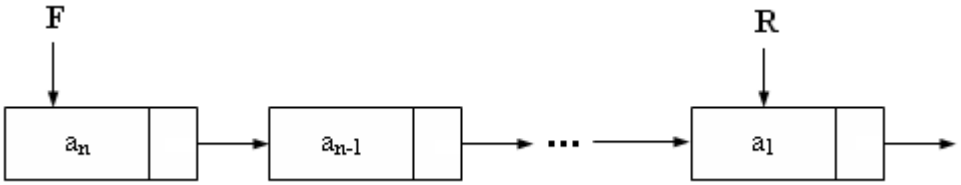
2.4.2. Các phép toán đối với Queue.

Cấu trúc dữ liệu biểu diễn Queue:

```
Type
    Tro = ^Node;
    Node = Record
        Info: Item;
        Next: Tro;
    End;
    Queue = Record
        F: Tro;          {F: Front}
        R: Tro;          {R: Rear}
    End;
Var
```

```
Q: Queue;
```

Trong cách biểu diễn trên, F là con trỏ trỏ vào nút đầu của Queue, R là con trỏ trỏ vào nút sau của Queue:



Với cách cài đặt như vậy ta có các phép toán trên Queue như sau:

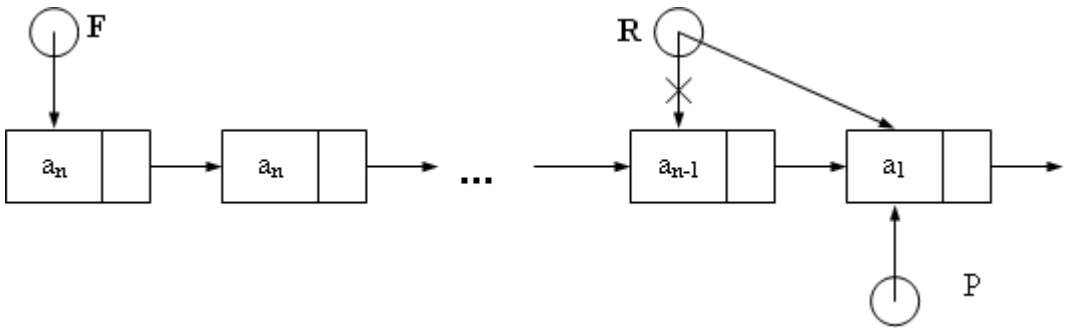
* Khởi tạo Queue rỗng:

```
Procedure Creat (Q);
    Begin
        Q.F := Null;
        Q.R := Null;
    End;
```

* Kiểm tra xem Queue có rỗng hay không:

```
Function Empty (Q: Queue): Boolean;
    Begin
        Empty := (Q.F = Null);
    End;
```

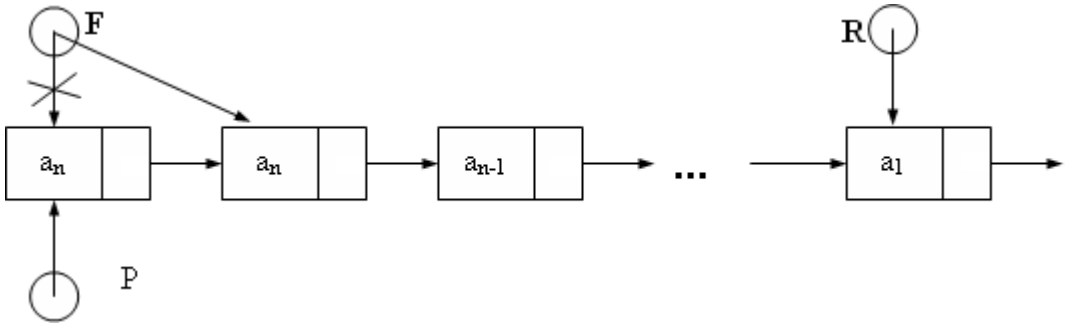
* Bổ sung phần tử vào cuối hàng đợi:




```
Procedure Add(Q, X);
    Var
        P: Tro;
    Begin
        New(P);
        P^.Info := X;
        P^.Link := Null;
```

```
        If Empty (Q) then
            Begin
                Q.F := P;
                Q.R := P;
            End
        Else
            Begin
                Q.R^.Link := P;
                Q.R := P;
            End;
    End;
```

* Lấy ra 1 phần tử ở đầu Queue:




```
    Procedure Del (Q, X);
        Var
            OK: Boolean;
            P: Tro;
        Begin
            If Empty(Q) then OK := False
            Else   Begin
                        P := Q.F;
                        X := Q.F^.Info;
                        Q.F := Q.F^.Link;
                        OK := True;
                        Dispose(P);
                   End;
        End;
```

Chương 4: **Sắp xếp và tìm kiếm**

1. Sắp xếp.

1.1. Đặt vấn đề.

Sắp xếp (sorting) là quá trình bố trí lại các phần tử của 1 tập đối tượng nào đó, theo 1 thứ tự nhất định nhằm thực hiện tìm kiếm thuận lợi, xử lý các kết quả nhanh chóng,...

Dữ liệu trong máy tính có thể xuất hiện dưới nhiều dạng khác nhau, nhưng ở đây ta quy ước tập đối tượng được sắp xếp là tập các bản ghi (record), mỗi bản ghi bao gồm các trường dữ liệu, tương ứng là các thuộc tính dữ liệu khác nhau.

Giả sử rằng mỗi phần tử dữ liệu được xem xét có 1 thành phần khóa (key) để nhận diện, việc sắp xếp dữ liệu coi như sắp xếp đối với khóa, để minh họa cho các phương pháp sắp xếp ta coi khóa K_1, K_2,..., K_n là các số và thứ tự sắp xếp là tăng dần, giả sử dãy số được coi là các khóa dùng minh họa cho các giải thuật là:

42 23 74 11 65 58 94 36 99 87

1.2. Các phương pháp sắp xếp cơ bản.

1.2.1. Sắp xếp kiểu lựa chọn (Selection Sort)

* Tư tưởng của phương pháp:

- Ban đầu dãy có n phần tử có thứ tự ngẫu nhiên, 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 của dãy K chưa có thứ tự. Chúng ta lại tiếp tục lựa chọn phần tử nhỏ nhất trong n – 1 phần tử này để đưa lên đầu nhóm n – 1 phần tử,... Do vậy, sau n – 1 lần lựa chọn phần tử nhỏ nhất để đưa lên đầu nhóm thì tất cả các phần tử trong dãy M sẽ có thứ tự tăng dần.

- Như vậy, thuật toán này chủ yếu chúng ta đi tìm giá trị nhỉ nhất trong nhóm n – i phần tử chưa có thứ tự đứng ở phía sau dãy K, việc này đơn giản chúng ta vận dụng thuật toán tìm kiếm tuần tự:

* Thuật toán:

B1: i = 1;

B2: Tìm phần tử nhỏ nhất K[min] trong dãy từ K[i] đến K[n];

B3: Hoán vị K[min] và K[i];

B4: Nếu i <= n – 1 thì i := i + 1; Lặp lại bước 2

Ngược lại: Dừng

Dãy K đã được sắp xếp đúng thứ tự.

Nguyên tắc cơ bản của thuật toán là ở lượt thứ i (i = 1, 2, ..., n) ta sẽ chọn trong dãy khóa K_i, K_{i+1}, ..., K_n khóa nhỏ nhất và đổi chỗ của nó với K_i. Sau j lượt ta có j khóa đã sắp xếp đúng theo vị trí mong muốn từ thứ 1 đến thứ j.

* Ví dụ minh họa:

Sắp xếp dãy số sau theo phương pháp sắp xếp lựa chọn:

K: 42 23 74 11 65 58 94 36 99 87

- Lần 1: i = 1, Pos = 1, K[i] = 42, K[min] = 11 < K[i], Pos[min] = 4:

K: **11** 23 74 **42** 65 58 94 36 99 87

- Lần 2: i = 2, Pos = 2, K[i] = 23, K[min] = 36 > K[i], Pos[min] = 8:

K: 11 **23** 74 42 65 58 94 36 99 87

- Lần 3: i = 3, Pos = 3, K[i] = 74, K[min] = 36 < K[i], Pos[min] = 8:

K: 11 23 **36** 42 65 58 94 **74** 99 87

- Lần 4: i = 4, Pos = 4, K[i] = 42, K[min] = 58 > K[i], Pos[min] = 6:

K: 11 23 36 **42** 65 58 94 74 99 87

- Lần 5: i = 5, Pos = 5, K[i] = 65, K[min] = 58 < K[i], Pos[min] = 6:

K: 11 23 36 42 **58** **65** 94 74 99 87

- Lần 6: i = 6, Pos = 6, K[i] = 65, K[min] = 74 > K[i], Pos[min] = 8:

K: 11 23 36 42 58 **65** 94 74 99 87

- Lần 7: i = 7, Pos = 7, K[i] = 94, K[min] = 74 < K[i], Pos[min] = 8:

K: 11 23 36 42 58 65 **74** **94** 99 87

- Lần 8: i = 8, Pos = 8, K[i] = 94, K[min] = 87 < K[i], Pos[min] = 10:

K: 11 23 36 42 58 65 74 **87** 99 **94**

- Lần 9: i = 9, Pos = 9, K[i] = 99, K[min] = 94 < K[i], Pos[min] = 10:

K: 11 23 36 42 58 65 74 87 **94** **99**

- Lần 10: i = 10 = n, dãy khóa đã được sắp xếp theo thứ tự tăng dần

* Giải thuật: (trang 210 – CTDL và GT)

```
Procedure Select_Soft (K, n);
    Begin
        For i:= 1 to n – 1 do
        Begin
            Pos := i;
            For j := i + 1 to n do
                If K[Pos] < K[j] then Pos := i
                Else
                    Begin
                        X := K[i];
                        K[i] := K[Pos];
                        K[Pos] := X;
                    End;
        End;
    End;
```

* Phân tích thuật toán:

- Trong mọi trường hợp ta có:

  + 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 dãy khóa K ban đầu đã có thứ tự tăng ta có:

  + Số phép gán: $G_{min} = 2(n-1)$

- Trường hợp xấu nhất, khi dãy khóa K ban đầu có thứ tự giảm ta có:

  + Số phép gán: $G_{max} = n(n-1)$.

- Trung bình, số phép gán: $G_{tb} = [2(n – 1) + n(n + 1)]/2$

1.2.2. Sắp xếp kiểu thêm dần (Insertion Soft)

* Tư tưởng của phương pháp: Sắp xếp chèn (Insertion Soft) là một thuật toán sắp xếp dựa theo các sắp xếp quân bài của những người chơi bài, muốn sắp xếp một bộ bài theo trật tự người chơi bài rút lần lượt từ quân thứ 2, so sánh với các quân đứng trước nó để chèn vào vị trí thích hợp.

Cơ sở lập luận của sắp xếp chèn có thể mô tả như sau: Xét danh sách con gồm i phần tử đầu K_1, K_2,..., K_i. Với i = 1, danh sách gồm 1 phần tử đã được sắp xếp. Giả sử trong danh sách i – 1 phần tử đầu K_1,..., K_{i-1} đã được sắp xếp, để sắp tiếp phần tử $K_i = x$ ta tìm vị trí thích hợp của nó trong dãy K_1,..., K_{i-1}, đó là vị trí đứng trước phần tử lớn hơn nó và đứng sau phần tử nhỏ hơn hoặc bằng nó.

Các phần tử x			Vị trí thích hợp	Các phần tử > x			Các phần tử chưa sắp xếp		
K_1	...	K_{j-1}	x	K_{j+1}	...	K_{i-1}	K_{i+1}	...	K_n

* Ví dụ minh họa:

Sắp xếp dãy số sau theo phương pháp sắp xếp thêm dần:

K: 42 23 74 11 65 58 94 36 99 87

Theo cơ sở lập luận của phương pháp, ta xem xét dãy K, lần lượt đưa các khóa vào sắp xếp ta có bảng minh họa:

Lượt		1	2	3	4	5	6	7	8	9	10
Khóa đưa vào		42	23	74	11	65	58	94	36	99	87
Vị trí sắp xếp hợp lý		1	1	3	1	3	4	7	3	9	8
Tình trạng của dãy khóa K đang sắp xếp	1	**42**	**23**	23	**11**	11	11	11	11	11	11
	2	-	42	42	23	23	23	23	23	23	23
	3	-	-	**74**	42	42	42	42	**36**	36	36
	4	-	-	-	74	**65**	**58**	58	42	42	42
	5	-	-	-	-	74	65	65	58	58	58
	6	-	-	-	-	-	74	74	65	65	65
	7	-	-	-	-	-	-	**94**	74	74	74
	8	-	-	-	-	-	-	-	94	94	**87**
	9	-	-	-	-	-	-	-	-	**99**	94
	10	-	-	-	-	-	-	-	-	-	99

* Giải thuật: Để đảm bảo trong mọi trường hợp, ngay cả khi khóa được sắp xếp đương nhiên có vị trí đầu tiên, đều được chèn vào giữa khóa nhỏ hơn và khóa lớn hơn nó ta đưa thêm vào 1 khóa giả K_0 và quy ước giá trị của khóa này nhỏ vô hạn: $K_0 = -$.

```
Procedure Insert_Soft (K, n);
    Begin
        K[0] := -  ;
        For i := 1 to n do
            Begin
                X := K[i];
                j := i – 1;
                While X < K[j] do
                    Begin
                        K[j + 1] := K[j];
                        j := j – 1;
                    End;
                K[j + 1] := X;
            End;
    End;
```

* Phân tích thuật toán:

- Trường hợp tốt nhất, khi dãy K ban đầu đã có thứ tự tăng:

  + Số phép gán: $G_{min} = 2(n - 1)$

  + Số phép so sánh: $S_{min} = 1 + 2 +...+ (n – 1) = n(n – 1)/2$

  + Số phép hoán vị: $H_{min} = 0$;

- Trường hợp xấu nhất, khi dãy K ban đầu luôn có thứ tự ngược với thứ tự cần sắp xếp:

  + Số phép gán: $G_{max} = 2(n - 1) + [1 + 2 +...+(n - 1)]$

  + Số phép so sánh: $S_{max} = (n – 1)$

  + Số phép hoán vị: $H_{max} = 0$;

- Trung bình:

  + Số phép gán: $G_{tb} = 2(n - 1) + n(n – 1)/4$

  + Số phép so sánh: $S_{tb} = n(n – 1)/2 + (n – 1)/2 = (n + 2)(n – 1)/4$

+ Số phép hoán vị: $H_{tb} = 0$;

1.2.3. Sắp xếp kiểu nổi bọt (Bubble Soft).

* Tư tưởng của phương pháp: Giả sử sắp xếp dãy khóa K gồm n phần tử, đi từ cuối dãy, ta so sánh 2 phần tử cuối, nếu phần tử đứng trước lớn hơn phần tử đứng sau thì đổi chỗ chúng cho nhau. Tiếp tục làm như vậy với cặp phần tử tiếp theo và tiếp tục cho đến hết tập hợp dữ liệu, nghĩa là so sánh (và đổi chỗ nếu cần). Sau bước này, phần tử đầu tiên của dãy là phần tử nhỏ nhất của dãy, tiếp tục là như vậy cho đến khi không phải đổi chỗ bất cứ cặp phần tử nào thì dãy đã được sắp xếp xong.

* Giải thuật:

```
Procedure Bubble_Soft (K,n);
1.      For i := 1 to n – 1 do
            Begin
2.              For j := n downto i + 1 do
3.                  If K[j] < K[j - 1] then
4.                          Begin
                                X := K[j];
                                K[j] := K[j - 1];
                                K[j - 1] := X;
                            End;
            End;
5.  Return;
```

* Ví dụ minh họa giải thuật:

Sắp xếp dãy số sau theo phương pháp sắp xếp nổi bọt:

K: 42 23 74 11 65 58 94 36 99 87

Với mỗi lượt sắp xếp trong vòng lặp nhỏ ta sắp xếp được 1 cặp kế cận nhau đúng thứ tự và sau 1 vòng lặp lớn ta sắp được 1 số trong dãy khóa vào đúng vị trí, khi đó ta có:

- Lượt 1: i = 1.

Ptử ss j	1	2	3	4	5	6	7	8	9	10
K[9], K[10]	42	23	74	11	65	58	94	36	**87** ↔	**99**

K[7], K[8]	42	23	74	11	65	58	**36** ↔	**94**	87	99
K[6], K[7]	42	23	74	11	65	**36** ↔	**58**	94	87	99
K[5], K[6]	42	23	74	11	**36** ↔	**65**	58	94	87	99
K[3], K[4]	42	23	**11** ↔	**74**	36	65	58	94	87	99
K[2], K[3]	42	**11** ↔	**23**	74	36	65	58	94	87	99
K[1], K[2]	**11** ↔	**42**	23	74	36	65	58	94	87	99

- Lượt 2: i = 2.

Ptử ss j		2	3	4	5	6	7	8	9	10
K[8], K[9]	11	42	23	74	36	65	58	**87** ↔	**94**	99
K[6], K[7]	11	42	23	74	36	**58** ↔	**65**	87	94	99
K[4], K[5]	11	42	23	**36** ↔	**74**	58	65	87	94	99
K[2], K[3]	**11**	**23** ↔	**42**	36	74	58	65	87	94	99

- Lượt 3: i = 3.

Ptử ss j			3	4	5	6	7	8	9	10
K[5], K[6]	**11**	**23**	42	36	**58** ↔	**74**	65	87	94	99
K[3], K[4]	**11**	**23**	**36** ↔	**42**	58	74	65	87	94	99

- Lượt 4: i = 4.

Ptử ss j				4	5	6	7	8	9	10
K[3], K[4]	**11**	**23**	**36**	**42**	58	**65** ↔	**74**	87	94	99

- Đến lượt sắp xếp này, ta thấy các khóa đã nằm đúng vị trí như mong muốn, các vòng lặp tiếp theo vẫn tiếp tục sắp xếp, tuy nhiên trong trường hợp này không có giá trị thực sự.

* Phân tích thuật toán.

- Trong mọi trường hợp:

+ Số phép gán: G = 0;

+ Số phép so sánh: S = (n – 1) + (n – 2) +...+ 1 = n(n – 1)/2.

- Trong trường hợp tốt nhất khi dãy khóa ban đầu đã có thứ tự tăng:

+ Số phép hoán vị: $H_{min} = 0$;

- Trong trường hợp xấu nhất khi dãy khóa ban đầu có thứ tự giảm:

+ Số phép hoán vị: $H_{max} = (n - 1) + (n - 2) +...+ 1 = n(n - 1)/2$

- Số phép hoán vị trung bình: $H_{tb} = n(n - 1)/4$.

* Nhận xét:

- Thuật toán sắp xếp nổi bọt khóa đơn giản, phần tử nhẹ được nổi lên khá nhanh sau mỗi lần duyệt dãy từ cuối nhưng phần tử nặng lại chìm xuống rất chậm do không có chiều duyệt dãy theo chiều đi xuống.

- Thuật toán nổi bọt không phát hiện được các đoạn phần tử đã nằm ở đúng vị trí để có thể giảm bớt số lần duyệt.

1.3. Phương pháp sắp xếp phân đoạn – Sắp xếp nhanh.

1.3.1. Giới thiệu phương pháp.

Sắp xếp nhanh *(Quicksort)*, còn được gọi là sắp xếp kiểu phân chia (part sort) là một thuật toán sắp xếp phát triển bởi C.A.R. Hoare, dựa trên phép phân chia danh sách được sắp thành hai danh sách con bằng cách so sánh từng phần tử của danh sách với một phần tử được chọn được gọi là **phần tử chốt** (pivot). Những phần tử nhỏ hơn hoặc bằng phần tử chốt được đưa về phía trước và nằm trong danh sách con thứ nhất, các phần tử lớn hơn chốt được đưa về phía sau và thuộc danh sách đứng sau. Cứ tiếp tục chia như vậy tới khi các danh sách con đều có độ dài bằng 1.

1.3.2. Chọn phần tử chốt.

Kỹ thuật chọn phần tử chốt ảnh hưởng khá nhiều đến khả năng rơi vào các vòng lặp vô hạn đối với các trường hợp đặc biệt. Tốt nhất là chọn phần tử chốt là trung vị của danh sách, khi đó thời gian thực hiện thuật toán sẽ là nhanh nhất, tuy nhiên rất khó để chọn được phần tử trung vị này. Có các cách chọn phần tử chốt như sau:

- Chọn phần tử đứng đầu hoặc đứng cuối làm phần tử chốt.

- Chọn phần tử đứng giữa danh sách làm phần tử chốt.

- Chọn phần tử trung vị trong 3 phần tử đứng đầu, đứng giữa và đứng cuối làm phần tử chốt.

- Chọn phần tử ngẫu nhiên làm phần tử chốt. (Cách này có thể dẫn đến khả năng rơi vào các trường hợp đặc biệt)

1.3.3. Phân đoạn

Sau khi phần tử chốt được chọn giải thuật phân chia nên tiến hành như thế nào?

- Cách 1: Duyệt dãy khóa từ đầu đến cuối lần lượt so sánh các phần tử của dãy khóa với phần tử chốt. Theo cách này, ta phải tiến hành n phép so sánh, ngoài ra còn phải dành n đơn vị bộ nhớ để lưu giữ các giá trị trung gian.

- Cách 2: Duyệt dãy khóa theo hai đường. Một đường từ đầu dãy, một đường từ cuối dãy. Theo cách này, ta tìm phần tử đầu tiên tính từ trái lớn hơn phần tử chốt và phần tử đầu tiên phía phải nhỏ hơn hoặc bằng phần tử chốt rồi đổi chỗ cho nhau. Tiếp tục như vậy cho đến khi hai đường gặp nhau.

1.3.4. Giải thuật

Sau khi chọn chốt, ta sẽ tiến hành phân đoạn cho dãy khóa nhờ vào chốt. Quy ước L là chỉ số phần tử đầu của dãy khóa đang xét (biên dưới), R là chỉ số phần tử cuối của dãy khóa đang xét (biên trên), K biểu diễn dãy khóa đang xét, j là chỉ số ứng với khóa chốt sau khi đã tách dãy khóa thành 2 phân đoạn.

* Giải thuật phân đoạn:

```
Procedure Part (K, L, R, j);
    1. i := L +1; j := R;
    2. While i     j do begin
    3.       While K[i] < K[L] do i := i +1;
    4.       While K[j] > K[L] do j := j - 1;
    5.       If i < j then begin
                  K[i]     K[j];
                  i := i +1;
                  j := j - 1;
    6. K[L]     K[j];
    7. Return;
```

1.3.5. Ví dụ minh họa.

Lượt	Chốt	42	23	74	11	65	58	94	36	99	87
1	**42**	(11	23	36)	**42**	(65	58	94	74	99	87)
2	**11**	**11**	(23	36)	42	65	58	94	74	99	87
3	**23**	11	**23**	(36)	42	65	58	94	74	99	87
4	**65**	11	23	36	42	(58)	**65**	(94	74	99	87)
5	**94**	11	23	36	42	58	65	(87	74)	**94**	(99)
6	**87**	11	23	36	42	58	65	(74)	**87**	94	99

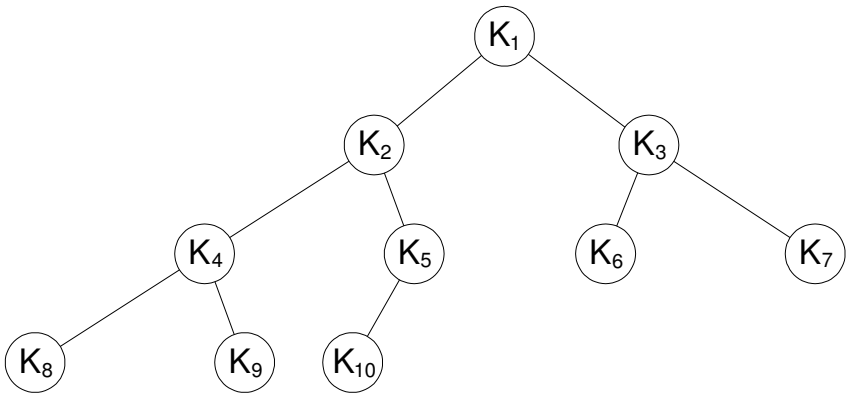
1.4. Sắp xếp vun đống (Head Soft)

1.4.1. Định nghĩa đống.

Dãy khóa $K_1, K_2,..., K_n$ được gọi là 1 đống nếu với mọi chỉ số i ta luôn có

$$\begin{cases} K_i > K_{2i} \\ K_i > K_{2i+1} \end{cases}$$

Như vậy dãy khóa sẽ là đống nếu với mọi i = 1.. (n div 2) thỏa mãn điều kiện trên. Dựa theo nguyên tắc lưu trữ cây nhị phân hoàn chỉnh bằng mảng 1 chiều, ta có thể coi dãy khóa là lưu trữ kế tiếp của 1 cây nhị phân hoàn chỉnh (là cây nhị phân có các nút ứng với các mức trừ mức cuối cùng đều đạt tốt đa, ở mức cuối cùng thì các nút đều dạt về bên phải), mỗi nút trên cây có 1 giá trị là K_i, và cây nhị phân sẽ có dạng:



Kết hợp định nghĩa về đống và cây nhị phân ta có định nghĩa khác về đống: Đống là cây nhị phân hoàn chỉnh mà mỗi nút trên cây được gắn 1 giá trị khóa sao cho giá trị khóa của nút cha luôn lớn hơn giá trị khóa của các nút con. Với cây nhị phân hoàn chỉnh là 1 đống ta luôn có:

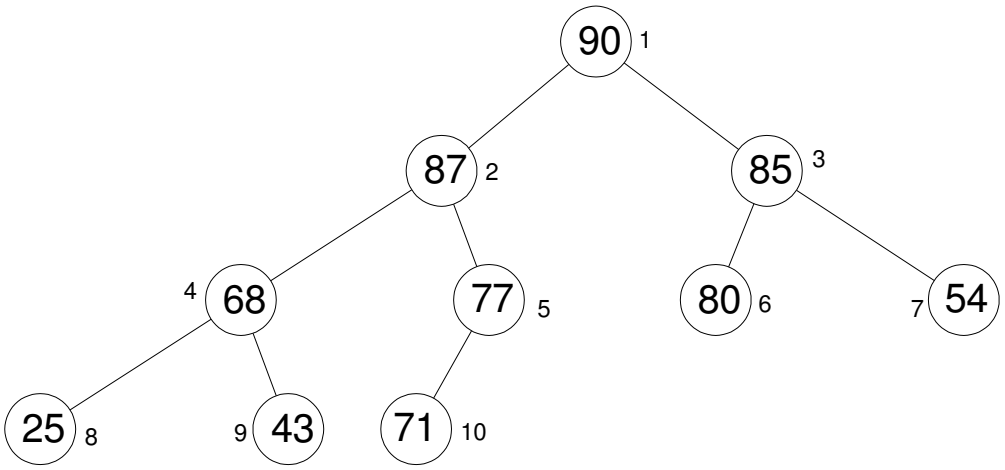
- Các nút lá trên cây nhị phân hoàn chỉnh đương nhiên thỏa mãn định nghĩa đống.

- Do tính chất phân cấp của các nút trên cây nên nút gốc luôn có giá trị lớn nhất.

Ví dụ: Với dãy khóa K:

K	90	87	85	68	77	80	54	25	43	71
i	1	2	3	4	5	6	7	8	9	10

Theo nguyên tắc lưu trữ tuyến tính, ta sẽ có cây nhị phân dạng:



Xét các điều kiện theo định nghĩa ta thấy cây nhị phân này là 1 đống bởi các khóa cha luôn lớn hơn các khóa con, các mức trên đều đạt tối đa còn ở mức cuối cùng các nút đều dạt về bên trái.

Tuy nhiên, với 1 dãy khóa K bất kì nào đó ta có thể không tạo ngay được 1 đống, khi đó cần phải tiến hành xây dựng đống (vun đống) từ cây nhị phân hoàn chỉnh.

1.4.2. Phép tạo đống.

Việc sắp xếp lại các giá trị tại các nút của 1 cây nhị phân hoàn chỉnh ban đầu để trở thành 1 đống được gọi là phép tạo đống.

Theo định nghĩa về đống ta thấy:

- Một cây nhị phân hoàn chỉnh đã là đống thì các cây con của các nút (nếu có) cũng là cây nhị phân hoàn chỉnh và đồng thời cũng là 1 đống.

- Một cây nhị phân hoàn chỉnh có n nút thì chỉ có (n div 2) nút được làm nút cha.

- Một nút lá bao giờ cũng có thể coi là 1 đống.

Từ đó ta thấy có thể thực hiện phép tạo đống bằng cách thực hiện vun đống từ đáy, nghĩa là bắt đầu từ các cây con có gốc là các nút có số thứ tự là (n div 2); (n div 2) - 1;...; 1. Bắt đầu từ các cây con này tức là ta lựa chọn các cây con có lá (đã là đống). Ta có giải thuật vun đống như sau:

```
Procedure Adjust (i, n);
1.      j := 2 * i;
        KEY := K[i];
2.      While j <= n do
            Begin
3.              If (j < n) and (K[j] < K[j + 1]) then j := j +1;
```

```
4.          If K[j] < KEY then
                Begin
                    K[j div 2] := K[j];
                    return
                End;
5.          K[j div 2] := K[j];
            j := 2 * J;
        End;
6.    K[j div 2] := KEY;
7.  Return
```

Với giải thuật này, việc tạo thành đống cho 1 cây nhị phân hoàn chỉnh có n nút sẽ được thực hiện bởi:

```
For i := (n div 2) down to 1 do call Adjust (i, n);
```

Ví dụ minh họa phép tạo đống.

1.4.3. Sắp xếp vun đống.

Sắp xếp kiểu vun đống là phương pháp sắp xếp dựa vào cấu trúc đống, phần tử lớn nhất nằm trên đỉnh của đống khi sắp xếp theo thứ tự tăng dần sẽ được đổi chỗ cho phần tử cuối cùng nằm ở đáy của đống, sau khi đổi chỗ ta được 1 phần tử nằm đúng vị trí trong dãy cần sắp xếp. Nếu không kể đến phần tử này thì các phần tử còn lại này cũng là 1 cây nhị phân hoàn chỉnh, ta lại tiến hành vun đống cho cây nhị phân này để tìm được phần tử lớn nhất. Cứ tiếp tục quá trình này cho đến khi cây chỉ còn 1 nút thì tức là tất cả các khóa đã được sắp xếp đúng vị trí theo thứ tự tăng dần.

Như vậy sắp xếp dãy khóa K theo kiểu vun đống gồm các giai đoạn:

1. Xây dựng cây nhị phân hoàn chỉnh.
2. Vun đống ban đầu.
3. Sắp xếp.

Thủ tục sắp xếp vun đống được thể hiện bởi giải thuật sau:

```
Procedure Head_sort (K, n);
    1. For i := (n div 2) down to 1 do
        Call Adjust (i, n);
    2. For i := (n - 1) down to 1 do
```

```
        Begin
            K[1]     K[i +1];
            Call Adjust (1, i);
        End;
    3. Return.
```

Ví dụ minh họa giải thuật vun đống:

2. Tìm kiếm.

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. Ví dụ tra cứu từ điển, tìm kiếm sách trong thư viện.

Do các hệ thông thông tin thường phải lưu trữ một khối lượng dữ liệu đáng kể nên việc xây dựng các giải thuật cho phép tìm kiếm nhanh sẽ có ý nghĩa rất lớn. Nếu dữ liệu trong hệ thống đã được tổ chức theo 1 trật tự nào đó thì việc tìm kiếm sẽ tiến hành nhanh chóng và hiệu quả hơn. Ví dụ: Các từ điểm được sắp xếp theo vần, trong mỗi vần lại được sắp xếp theo trình tự alphabet sẽ giúp chúng ta tìm kiếm nhanh hơn.

Vì thế, khi xây dựng một hệ quản lí thông tin, bên cạnh thuật toán tìm kiếm, các thuật toán sắp xếp dự liệu cũng là chủ đề được quan tâm hàng đầu.

Việc tìm kiếm nhanh hay chậm tùy thuộc vào trạng thái và trật tự của dữ liệu trên đó. Kết quả 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 ở đâu.

2.2. Tìm kiếm tuyến tính (Sequential Searching).

2.2.1. Cơ sở lí thuyết.

Tìm kiếm tuyến tính là một kỹ thuật tìm kiếm rất đơn giản và cổ điển. Thuật toán tiến hành so sánh phần tử cần tìm kiếm X lần lượt với phần tử thứ 1, thứ 2,... của dãy K cho đến khi gặp được phần tử có khóa cần tìm hoặc đã tìm hết mảng mà không thấy X. Các bước tiến hành như sau:

- Bước 1: i := 1; //Bắt đầu tìm từ phần tử đầu tiên của dãy.

- Bước 2: So sánh K[i] với X sẽ có 2 khả năng xảy ra:

+ K[i] = X. Tìm thấy, dừng lại quá trình tìm kiếm.

+ K[i] <> X. Chưa tìm thấy, chuyển sang bước 3.

- Bước 3: i := i +1 //Xét phần tử kế tiếp trong mảng.

+ Nếu i > n. Hết dãy khóa mà không tìm thấy. Dừng quá trình tìm kiếm và kết luận không có phần tử cần tìm trong dãy.

+ Nếu i <= n thì tiếp tục lặp lại bước 2.

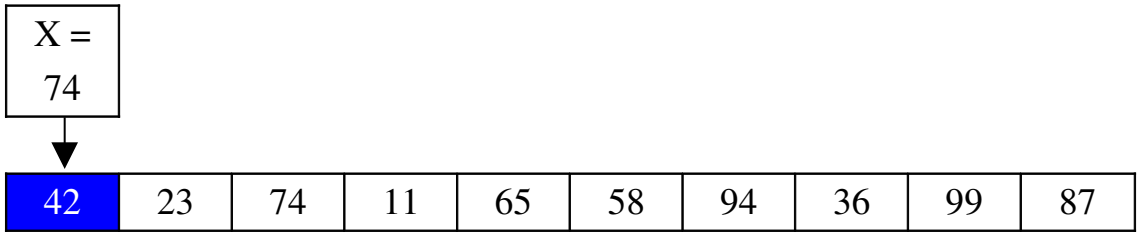
2.2.2. Ví dụ minh họa.

Cho dãy khóa:

K:	42	23	74	11	65	58	94	36	99	87

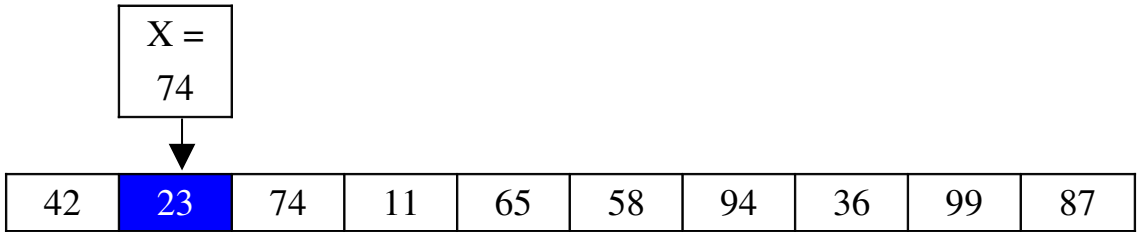
Nếu giá trị cần tìm là 74 thì giải thuật được tiến hành như sau:

i = 1:



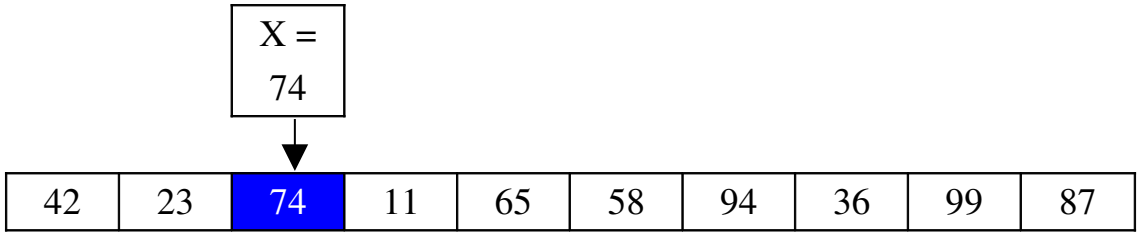
So sánh X = 74 với phần tử thứ 1 (i = 1), thấy rằng X <> K_1 nên chuyển sang phần tử tiếp theo.

i = 2:



So sánh X = 74 với phần tử thứ 2 (i = 2), thấy rằng X <> K_2 nên chuyển sang phần tử tiếp theo.

i = 3:



So sánh X = 74 với phần tử thứ 3 (i = 3), thấy rằng X = K_3 nên dừng lại quá trình tìm kiếm, kết luận đã tìm thấy phần tử cần tìm là phần tử thứ 3 trong dãy.

2.2.3. Giải thuật tìm kiếm tuyến tính.

* Giải thuật: Giả sử dãy khóa K gồm n phần tử, giải thuật tìm kiếm phần tử X trong dãy khóa. Nếu có sẽ đưa ra chỉ số của khóa đó, nếu không sẽ đưa ra giá trị 0. Trong giải thuật sử dụng khóa phụ K_{n+1} mà giá trị chính là X.

```
Function Sequen_Search (K, n, X);
1. {Khởi đầu}
        i := 1; K[n +1] := X;
2. {Tìm khóa trong dãy}
        While K[i] <> X do i := i +1;
3. {Tìm thấy hoặc không thấy}
        If i = n +1 then return(0)
        Else return(i);
```

* Đánh giá giải thuật: Thông qua ước lượng các phép so sánh được tiến hành để tìm ra X, ta có thể đánh giá độ phức tạp của giải thuật tìm kiếm tuyến tính:

Trường hợp	**Số lần so sánh**	**Giải thích**
Tốt nhất	1	Phần tử đầu tiên có giá trị X
Xấu nhất	n + 1	Phần tử cuối cùng có giá trị X
Trung bình	(n + 1)/2	Giả sử xác suất các phần tử trong mảng nhận giá trị X là như nhau

* Nhận xét:

- Giải thuật tìm kiếm tuyến tính không phụ thuộc vào thứ tự của các phần tử trong dãy khóa, do đó đây là phương pháp tổng quát nhất để tìm kiếm trên 1 dãy khóa bất kỳ.

- Một thuật toán có thể được cài đặt theo nhiều cách khác nhau, kỹ thuật cài đặt ảnh hưởng đến tốc độ thực hiện của thuật toán.

2.3. Tìm kiếm nhị phân (Binary Searching)

2.3.1. Cơ sở lí thuyết.

Tìm kiếm nhị phân là phương pháp tìm kiếm thông dụng và thường áp dụng cho các trường hợp tìm kiếm trong các dãy khóa có thứ tự. Phương pháp này tương tự phương pháp tìm kiếm mà chúng ta làm khi tra từ điển hoặc tìm số điện thoại.

Khi tìm kiếm phần tử X trong dãy đã có thứ tự (giả sử thứ tự tăng), các phần tử trong dãy có quan hệ $K_{i-1} < K_i < K_{i+1}$ nên nếu $X > K_i$ thì X chỉ có thể xuất hiện trong đoạn $[K_{i+1}, K_n]$, ngược lại nếu $X < K_i$ thì X chỉ có thể xuất hiện trong đoạn $[K_1, K_{i-1}]$. Giải thuật tìm kiếm nhị phân áp dụng nhận xét trên để giới hạn phạm vi tìm kiếm sau mỗi lần so sánh X với 1 phần tử trong dãy. Ý tưởng của giải thuật là tại mỗi bước tiến hành so sánh phần tử X với phần tử nằm ở **vị trí giữa** của dãy tìm kiếm hiện hành, dựa vào kết quả so sánh này để quyết định giới hạn dãy tìm kiếm ở bước kế tiếp là nửa trên hay nửa dưới của dãy tìm kiếm hiện hành. Quá trình tìm kiếm được tiếp tục cho đến khi tìm thấy khóa mong muốn hoặc dãy khóa đang xét đó trở nên rỗng (không tìm thấy)

Giả sử dãy tìm kiếm hiện hành nhận K_{left} là phần tử đầu phía trái, K_{right} là phần tử đầu phía phải, khi đó các bước tìm kiếm như sau:

- Bước 1: left = 1; right = n {tìm kiếm trên tất cả các phần tử của dãy}
- Bước 2:
  + mid = ((left + right) div 2) {lấy mốc so sánh}
  + So sánh K_{mid} với X sẽ có 3 khả năng xảy ra:

    K_{mid} = X. Đã tìm thấy, quá trình tìm kiếm dừng lại.

    K_{mid} > X. Tìm kiếm tiếp tục trong dãy con K_{left} .. K_{mid-1}, tức là right := mid - 1.

    K_{mid} < X. Tìm kiếm tiếp tục trong dãy con K_{mid+1} .. K_{right}, tức là left := mid + 1.

- Bước 3: Nếu left <= right tức là vẫn còn phần tử chưa xét, quá trình tìm kiếm vẫn tiếp tục lặp lại bước 2. Ngược lại, đã tìm hết các phần tử, quá trình tìm kiếm dừng lại.

2.3.2. Ví dụ minh họa.

Cho dãy khóa K đã được sắp xếp theo thứ tự tăng dần:

K:	[11	23	36	42	58	65	74	87	94	99]
	left = 1				mid					right = 10

Nếu phần tử cần tìm là X = 74 thì quá trình tìm kiếm được minh họa như sau:

- left = 1, right = 10 thì mid = 5. (K_{mid} = 58) < (X = 74) nên left = mid + 1 = 6, dãy tìm kiếm mới là:

K:	**[65**	**74**	**87**	**94**	**99]**
	left = 6		mid		right = 10

- left = 6, right = 10 thì mid = 8. (K_{mid} = 87) > (X = 74) nên right = mid - 1 = 7 dãy tìm kiếm mới là:

K:	**[65**	**74]**
	left = 6	right = 7

- left = 6, right = 7 thì mid = 6. (K_{mid} = 65) < (X = 74) nên left = mid + 1 = 7, dãy tìm kiếm mới là:

K:	**[74]**
	left = right = 7

- left = right = 7 thì mid = 7. (K_{mid} = 74) = (X = 74). Đã tìm thấy X = 74 là phần tử thứ 7 trong dãy.