Dynamic Programming

Giới thiệu

- *Dynamic programming*
  - giải bài toán bằng cách kết hợp các lời giải của các bài toán con.
  - (ở đây "programming" không có nghĩa là lập trình).
- So sánh *dynamic programming* và "chia-và-trị" (divide-and-conquer)
  - Giải thuật chia-và-trị
    - chia bài toán thành các bài toán con độc lập ,
    - giải chúng bằng đệ quy,
    - kết hợp chúng để có lời giải cho bài toán ban đầu.
  - Giải thuật dynamic programming
    - các bài toán con không độc lập với nhau: chúng có chung các bài toán con nhỏ hơn.
    - giải mỗi bài toán con chỉ một lần, và ghi nhớ lời giải đó trong một bảng để truy cập khi cần đến.

Bài toán tối ưu

- Bài toán tối ưu
  - có thể có nhiều lời giải
  - mỗi lời giải có một trị
- Tìm lời giải có trị tối ưu (cực tiểu hay cực đại).

Nguyên tắc của dynamic programming

- Một giải thuật dynamic programming được xây dựng qua bốn bước:
  1. Xác định cấu trúc của một lời giải tối ưu.
  2. Định nghĩa đệ quy cho giá trị của một lời giải tối ưu.
  3. Tính giá trị của một lời giải tối ưu từ dưới lên (“bottom-up”).
  4. Xây dựng lời giải tối ưu từ các thông tin đã tính.

Nhân một chuỗi ma trận

- Cho một chuỗi ma trận $\langle A_1, A_2, ..., A_n \rangle$.
- Xác định tích $A_1 A_2 \cdots A_n$ dựa trên giải thuật xác định tích của hai ma trận.
- Biểu diễn cách tính tích của một chuỗi ma trận bằng cách "*đặt giữa ngoặc*" (*pa'renthesize*) các cặp ma trận sẽ được nhân với nhau.
- Một tích của một chuỗi ma trận là *fully parenthesized* nếu nó là
  - một ma trận hoặc là
  - tích của hai tích của chuỗi ma trận *fully parenthesized* khác, và được đặt giữa ngoặc.

  Ví dụ: một vài tích của chuỗi ma trận được fully parenthesized
  - A
  - (AB)
  - $((AB)C)$.

Chuỗi ma trận fully parenthesized

- Ví dụ: Cho một chuỗi ma trận $\langle A_1, A_2, A_3, A_4 \rangle$. Tích $A_1A_2A_3A_4$ có thể được *fully parenthesized* theo đúng 5 cách khác nhau:

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

Nhân hai ma trận

- Tích của hai ma trận A và B với
  - A có chiều là $p \times q$
  - B có chiều là $q \times r$

  là một ma trận C có chiều là $p \times r$.

```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2     then error "các chiều không tương thích"
3     else for i ← 1 to rows[A]
4               do for j ← 1 to columns[B]
5                     do C[i, j] ← 0
6                        for k ← 1 to columns[A]
7                            do C[i, j] ← C[i, j] + A[i, k]·B[k, j]
8           return C
```

- Thời gian để tính C tỷ lệ với số phép nhân vô hướng thực thi trong dòng 7, tức là $p \times q \times r$.

Phí tổn để nhân một chuỗi ma trận

- Nhận xét: Phí tổn nhân một chuỗi ma trận tùy thuộc vào cách đặt giữa ngoặc (parenthesization).
- Ví dụ: Cho chuỗi ma trận $\langle A_1, A_2, A_3 \rangle$ trong đó các chiều (dimension) của các ma trận là 10×100, 100×5, và 5×50

  Có đúng 2 cách để đóng ngoặc hoàn toàn tích $A_1A_2A_3$:
  - Cách 1: $((A_1A_2)A_3)$
    - Tính A_1A_2 cần $10 \cdot 100 \cdot 5 = 5000$ phép nhân vô hướng
    - Kế đó nhân A_1A_2 với A_3 cần $10 \cdot 5 \cdot 50 = 2500$ phép nhân vô hướng
    - Tổng cộng: 7500 phép nhân vô hướng
  - Cách 2: $(A_1(A_2A_3))$
    - Tính A_2A_3 cần $100 \cdot 5 \cdot 50 = 25000$ phép nhân vô hướng
    - Kế đó nhân A_1 với A_2A_3 cần $10 \cdot 100 \cdot 50 = 50000$ phép nhân vô hướng
    - Tổng cộng: 75000 phép nhân vô hướng.

Bài toán nhân chuỗi ma trận

- Cho chuỗi ma trận $\langle A_1, A_2, ..., A_n \rangle$ gồm n ma trận, trong đó chiều của A_i là $p_{i-1} \times p_i$, với $i = 1, 2, \ldots, n$.
- Bài toán: Xác định một đóng ngoặc hoàn toàn cho tích $A_1 A_2 \cdots A_n$ sao cho số phép nhân vô hướng là tối thiểu.
- Giải bài toán trên bằng cách vét cạn?

Đếm số cách đóng ngoặc

- Cho một chuỗi gồm n ma trận $\langle A_1, A_2, A_3, ..., A_n \rangle$.
- Nhận xét: tạo ra một cách đóng ngoặc bằng cách *tách* (split) giữa A_k và A_{k+1}, với $k = 1, 2, ..., n-1$, tạo ra hai chuỗi con $A_1 A_2 \cdots A_k$ và $A_{k+1} \cdots A_n$, sau đó đóng ngoặc mỗi chuỗi con.
- Gọi $P(n)$ là số các cách đóng ngoặc cho một chuỗi n ma trận
  - nếu $n = 1$ thì chỉ có một cách đóng ngoặc (không cần dấu ngoặc tường minh). Vậy $P(1) = 1$.
  - nếu $n \geq 2$ thì từ nhận xét trên ta có

$$P(n) = \sum_{k=1}^{n-1} P(k) P(n-k)$$

Từ đó chứng minh được:

$$P(n) = \Omega(4^n / n^{3/2})$$

- Vậy dùng phương pháp vét cạn duyệt qua tất cả các cách đóng ngoặc để tìm một đóng ngoặc tối ưu cần thời gian chạy lũy thừa.

Bước 1: Cấu trúc của một đóng ngoặc tối ưu

- Bước 1 của phương pháp dynamic programming là
  - xác định tính chất *cấu trúc con tối ưu*
  - dựa vào đó xây dựng lời giải tối ưu cho bài toán từ các lời giải tối ưu cho các bài toán con.

Ở đây:

- Gọi $A_{i..j}$ là ma trận có được từ tích $A_i A_{i+1} \cdots A_j$.
- Nhận xét: Một đóng ngoặc tối ưu bất kỳ của tích $A_i A_{i+1} \cdots A_j$ tách nó giữa A_k và A_{k+1}, với k nào đó thõa $i \le k < j$:

  $$(A_i A_{i+1} \cdots A_k)(A_{k+1} \cdots A_j)$$

  Nghĩa là đầu tiên ta tính các ma trận $A_{i..k}$ và $A_{k+1..j}$, sau đó ta nhân chúng với nhau để có tích cuối cùng $A_{i..j}$. Do đó phí tổn để tính tích từ đóng ngoặc tối ưu là phí tổn để tính $A_{i..k}$, cộng phí tổn để tính $A_{k+1..j}$, cộng phí tổn để nhân chúng với nhau.

Bước 1: Cấu trúc của một đóng ngoặc tối ưu (tiếp)

- Cấu trúc con tối ưu
  - Đóng ngoặc của chuỗi con "tiền tố" $A_i A_{i+1} \cdots A_k$ có được từ đóng ngoặc tối ưu của $A_i A_{i+1} \cdots A_j$ phải là một đóng ngoặc tối ưu của $A_i A_{i+1} \cdots A_k$. (Chứng minh bằng phản chứng).
  - Tương tự, đóng ngoặc của chuỗi con còn lại $A_{k+1} A_{k+2} \cdots A_j$ có được từ đóng ngoặc tối ưu của $A_i A_{i+1} \cdots A_j$ phải là một đóng ngoặc tối ưu của $A_{k+1} A_{k+2} \cdots A_j$.
- Để cho gọn, sẽ nói "phí tổn của một đóng ngoặc" thay vì nói "phí tổn để tính tích từ một đóng ngoặc".
- Xây dựng lời giải tối ưu
  - Chia bài toán thành hai bài toán con
  - Tìm lời giải tối ưu cho mỗi bài toán con
  - Kết hợp các lời giải tìm được ở trên.

  Cần tìm vị trí thích hợp (trị của k) để tách chuỗi ma trận $A_i A_{i+1} \cdots A_j$!

Bước 2: Giải đệ quy

- Bước 2 của phương pháp dynamic programming là
  - định nghĩa đệ quy phí tổn (trị) của một lời giải tối ưu tùy theo các lời giải tối ưu của các bài toán con.
- Bài toán con ở đây: Xác định phí tổn tối thiểu cho một đóng ngoặc của chuỗi ma trận $A_i A_{i+1} \cdots A_j$ với $1 \le i \le j \le n$.
- Định nghĩa $m[i, j]$ là số phép nhân vô hướng tối thiểu để tính ma trận $A_{i..j}$. Phân biệt hai trường hợp:
  - nếu $i = j$ thì $A_i A_{i+1} \cdots A_j = A_i$. Vậy, với $i = 1,..., n$,

    $m[i, i] = 0$.
  - nếu $i < j$ thì từ bước 1 ta có

    $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

    Nhưng trị của k?

Bước 2: Giải đệ quy (tiếp)

Trả lời:

Bằng cách duyệt qua tất cả các trị của k, $i \le k \le j-1$, ta tìm được

$$m[i,j] = \min_{i \le k \le j-1} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\}.$$

- Để ghi lại cách xây dựng lời giải tối ưu ta định nghĩa $s[i,j]$ là trị của k xác định nơi tách chuỗi $A_i A_{i+1} \cdots A_j$ để có một đóng ngoặc tối ưu. Nghĩa là $s[i,j]$ là một trị k sao cho

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j .$$

Bước 3: Tính các chi phí tối ưu

- Bước 3 của phương pháp dynamic programming là tính chi phí tối ưu bằng một phương pháp từ dưới lên (bottom-up) và dùng bảng.
- Nhận xét:
  - Có thể viết được ngay một giải thuật đệ quy (dựa trên hàm đệ quy đã tìm được) để tính phí tổn tối ưu $m[1, n]$ cho tính tích $A_1 A_2 \cdots A_n$. Nhưng sau này chúng ta sẽ thấy là giải thuật này chạy trong thời gian lũy thừa.

Bước 3: Tính các chi phí tối ưu (tiếp)

- Ma trận A_i có chiều là $p_{i-1} \times p_i$, với $i = 1, 2, ..., n$.
- Input là một chuỗi $p = < p_0, p_1, ..., p_n >$
- Giải thuật trả về hai bảng $m[1..n, 1..n]$ và $s[1..n, 1..n]$.

```
MATRIX-CHAIN-ORDER(p)
 1   n ← length[p] − 1
 2   for i ← 1 to n
 3       do m[i, i] ← 0
 4   for l ← 2 to n
 5       do for i ← 1 to n − l + 1
 6              do j ← i + l − 1
 7                 m[i, j] ← ∞
 8                 for k ← i to j − 1
 9                     do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                        if q < m[i, j]
11                           then m[i, j] ← q
12                                s[i, j] ← k
13   return m and s
```

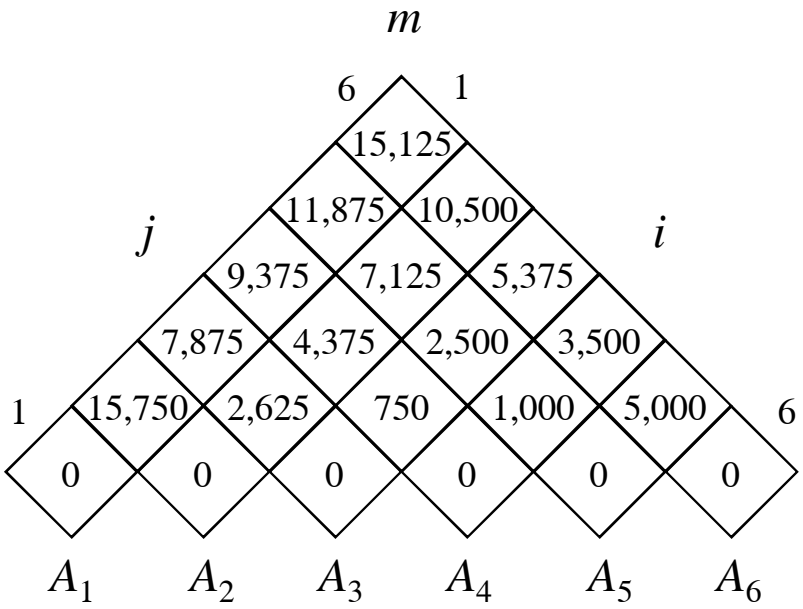
Phân tích MATRIX-CHAIN-ORDER

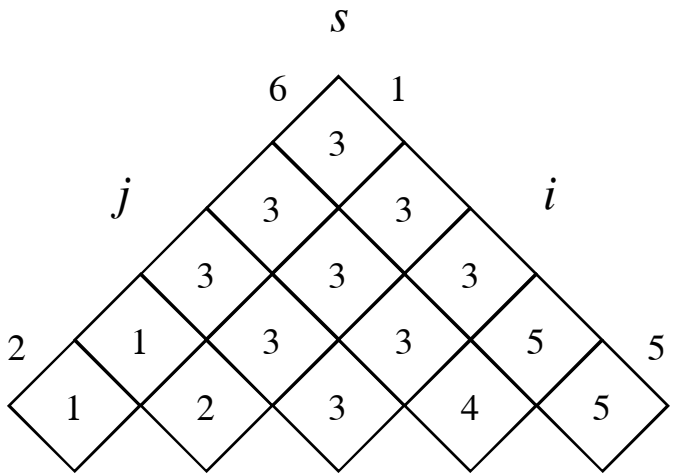
- Thời gian chạy của MATRIX-CHAIN-ORDER là $O(n^3)$.
- Giải thuật cần bộ nhớ $\Theta(n^2)$ cho các bảng m và s.

Chạy MATRIX-CHAIN-ORDER lên một ví dụ

ma trận	chiều
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

- Các bảng m và s tính được:





Bước 4: Xây dựng một lời giải tối ưu

- Bảng $s[1..n, 1..n]$ trữ một cách đóng ngoặc tối ưu do MATRIX-CHAIN-ORDER tìm ra.
- Thủ tục sau, MATRIX-CHAIN-MULTIPLY, trả về tích của chuỗi ma trận $A_{i..j}$ khi cho $A = \langle A_1, A_2, A_3, ..., A_n \rangle$, bảng s, và các chỉ số i và j.

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
1       if j > i
2          then X ← MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
3               Y ← MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
4               return MATRIX-MULTIPLY(X, Y)
5          else return A_i
```

- Gọi MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$ để tính tích của chuỗi ma trận A.

Các yếu tố để áp dụng dynamic programming

- Hai yếu tố để áp dụng được phương pháp dynamic programming vào một bài toán tối ưu
  - "Cấu trúc con tối ưu"
  - "Các bài toán con trùng nhau".

Một lời giải không tối ưu

- Giải thuật không ghi nhớ lời giải của các bài toán con.

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1       if i = j
2           then return 0
3       m[i, j] ← ∞
4       for k ← i to j – 1
5       do q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
                   + RECURSIVE-MATRIX-CHAIN(p, k + 1, j) + p_{i–1} p_k p_j
6           if q < m[i, j]
7              then m[i, j] ← q
8       return m[i, j]
```

Phân tích RECURSIVE-MATRIX-CHAIN

- Gọi $T(n)$ là thời gian chạy của RECURSIVE-MATRIX-CHAIN(p, 1, n), thì $T(n)$ phải thỏa (xem code)

$$T(1) \geq 1$$

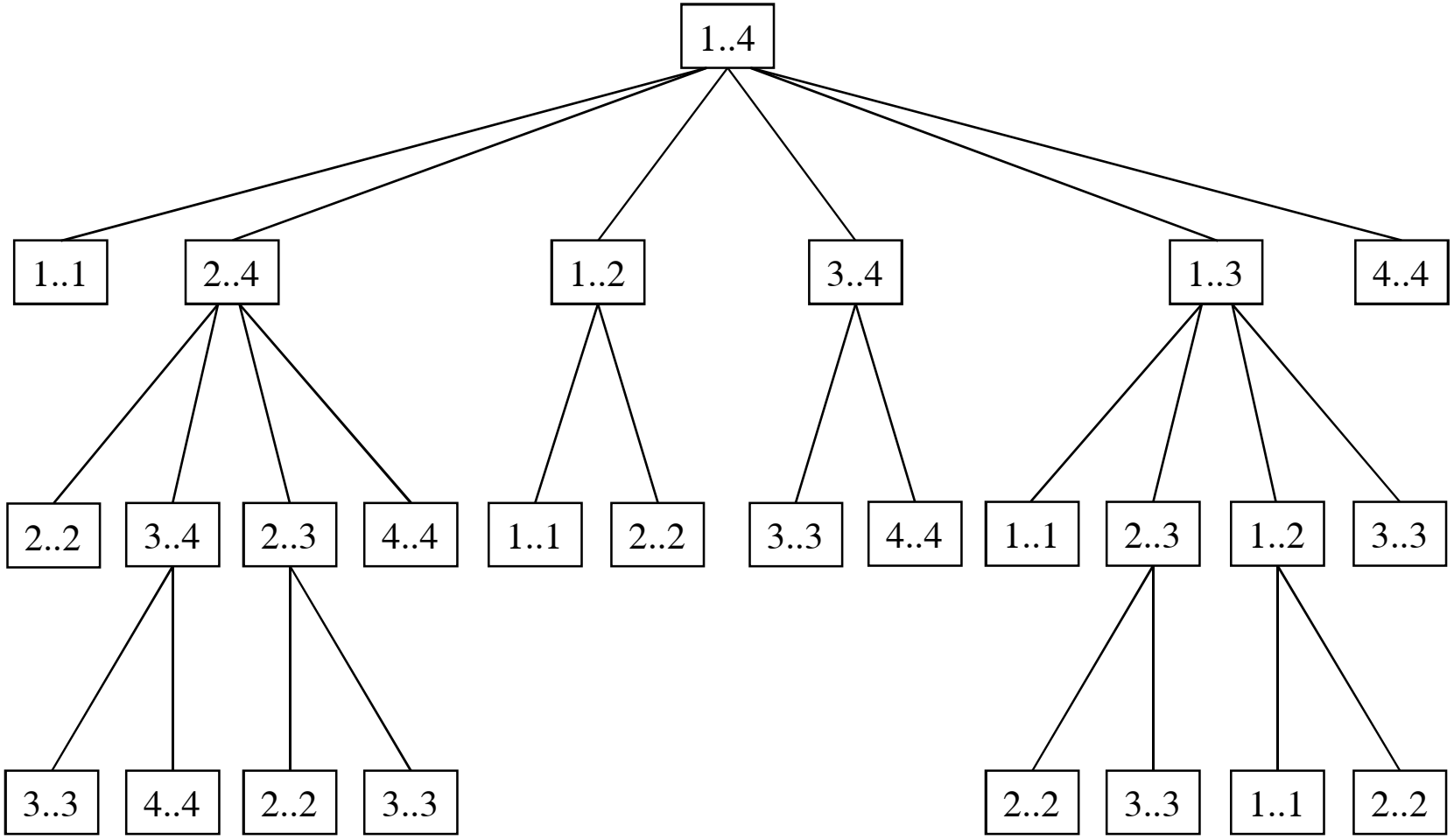
$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{for } n > 1.$$

  Từ đó chứng minh được: $T(n) = \Omega(2^n)$.

- Tại sao RECURSIVE-MATRIX-CHAIN chạy trong thời gian $\Omega(2^n)$ còn MATRIX-CHAIN-ORDER chỉ cần thời gian đa thức? Đó là vì
  - RECURSIVE-MATRIX-CHAIN là giải thuật đệ quy từ trên xuống (top-down) và không tận dụng được tính chất "các bài toán con trùng nhau" (overlapping subproblems).
  - MATRIX-CHAIN-ORDER là giải thuật dynamic-programming từ dưới lên (bottom-up), tận dụng được tính chất "các bài toán con trùng nhau".

Cây đệ quy

- Cây đệ quy cho RECURSIVE-MATRIX-CHAIN(p, 1, 4)



Một biến dạng của dynamic programming: *memoization*

- *Memoization* là phương pháp tận dụng tính chất "các bài toán con trùng nhau" để cải tiến giải thuật đệ quy từ trên xuống bằng cách
  - sử dụng một bảng chung mà mỗi triệu gọi của giải thuật đệ quy có thể truy cập để
    - ghi kết quả sau khi giải một bài toán con mới
    - đọc kết quả của một bài toán con đã được giải rồi.

Memoize giải thuật RECURSIVE-MATRIX-CHAIN

- Memoize giải thuật RECURSIVE-MATRIX-CHAIN bằng cách sử dụng bảng $m[1..n, 1..n]$.
- MEMOIZED-MATRIX-CHAIN có input là một chuỗi $p = < p_0 , p_1,..., p_n >$

```
MEMOIZED-MATRIX-CHAIN(p)
1       n ← length[p] – 1
2       for i ← 1 to n
3           do for j ← i to n
4                   do m[i, j] ← ∞
5       return LOOKUP-CHAIN(p, 1, n)
```

Memoization

- LOOKUP-CHAIN bao giờ cũng trả về $m[i, j]$. Nhưng nó chỉ tính $m[i, j]$ khi nào đó là lần gọi đầu tiên với các tham số i và j.

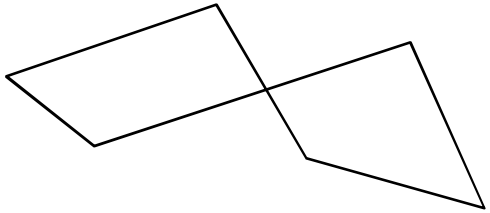
```
LOOKUP-CHAIN(p, i, j)
1       if m[i, j] < ∞
2          then return m[i, j]
3       if i = j
4          then m[i, j] ← 0
5       else for k ← i to j – 1
6                do q ← LOOKUP-CHAIN(p, i, k)
                        + LOOKUP-CHAIN(p, k + 1, j) + p_{i–1} p_k p_j
7                   if q < m[i, j]
8                      then m[i, j] ← q
9       return m[i, j]
```

Phân tích MEMOIZED-MATRIX-CHAIN

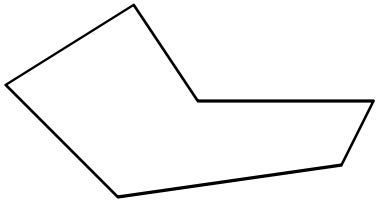
- MEMOIZED-MATRIX-CHAIN chạy trong thời gian $O(n^3)$.
- Nhận xét:
  - MEMOIZED-MATRIX-CHAIN tận dụng được tính chất "các bài toán con trùng nhau",
  - còn RECURSIVE-MATRIX-CHAIN chạy trong thời gian $\Omega(2^n)$ vì nó luôn luôn giải các bài toán con mà không để ý xem bài toán con đã được giải rồi hay chưa.

Phân tam giác

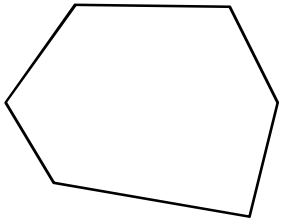
- *Đa giác*



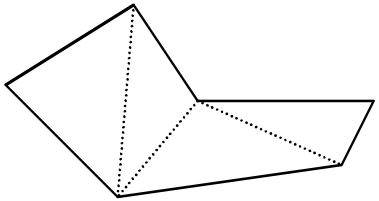
- Đa giác *đơn* (“simple”)



- Đa giác *lồi*

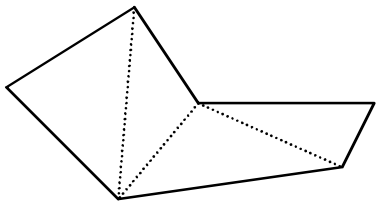


- *Phân tam giác* (triangulation)



Các khái niệm cơ bản

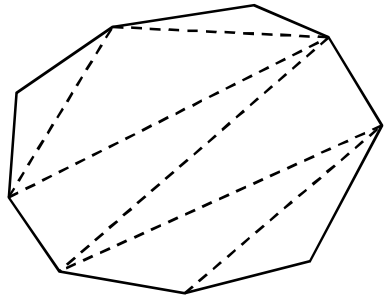
- *Cạnh*, *đỉnh*, *biên* của một đa giác
- Ta biểu diễn một đa giác lồi P bằng danh sách các đỉnh theo thứ tự ngược chiều kim đồng hồ: $P = \langle v_0 , v_1,..., v_{n-1} \rangle$
- *Cung* (“chord”) của một đa giác
- Một *phân tam giác* của một đa giác là một tập hợp các cung của đa giác chia đa giác thành các tam giác rời nhau.



Bài toán phân tam giác tối ưu

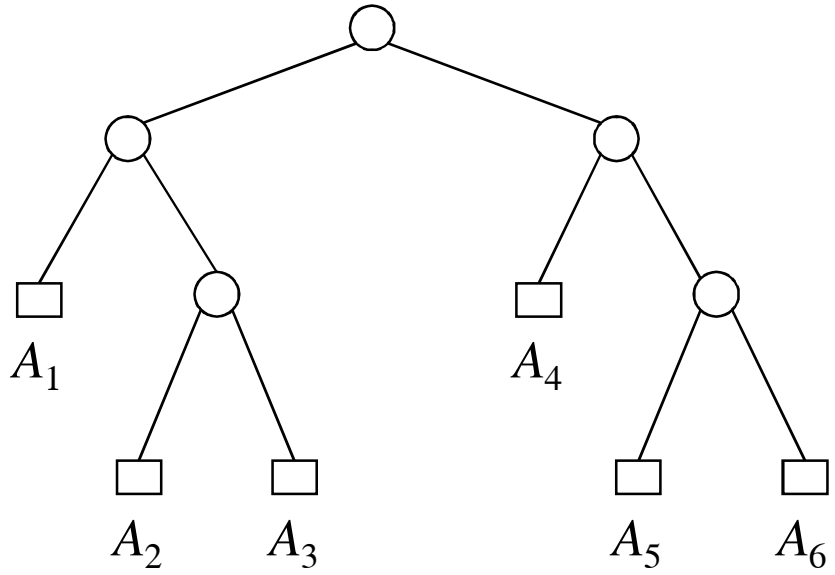
- Cho:
  - Một đa giác lồi $P = \langle v_0, v_1, ..., v_{n-1} \rangle$
  - Một h*àm trọng số* w ("weight function") được định nghĩa trên các tam giác tạo bởi cạnh và cung của P.
- Bài toán: Tìm một phân tam giác cho P sao cho tổng các trọng số của các tam giác trong phân tam giác này là nhỏ nhất.
- Ví dụ một hàm trọng số:

  w(một tam giác) = tổng các chiều dài của các cạnh của tam giác.



Parse tree của một biểu thức

- *Biểu thức* (expression)
  - Ví dụ một biểu thức: tích của một chuỗi ma trận đã được đóng ngoặc hoàn toàn $((A_1(A_2A_3))(A_4(A_5A_6)))$
- *Parse tree*.
  - Ví dụ: parse tree của biểu thức $((A_1(A_2A_3))(A_4(A_5A_6)))$ là

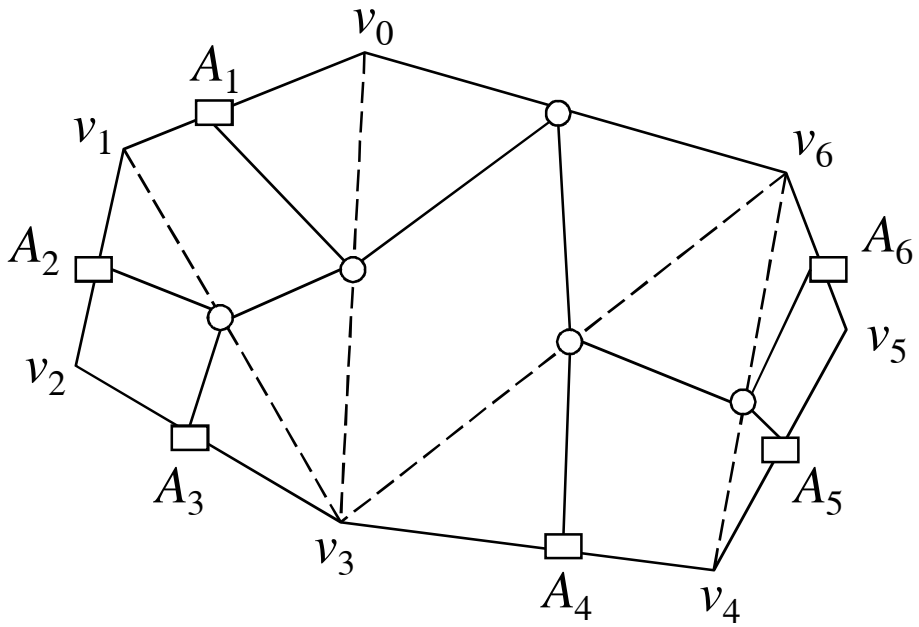


Parse tree của một biểu thức

- Định nghĩa: *parse tree của một biểu thức* là một cây mà
  - *Lá*: có nhản là một trong các nguyên tử ("atomic element", ví dụ: A_1) tạo nên biểu thức.
  - Nếu gốc của một cây con của parse tree có cây con bên trái tượng trưng biểu thức E_l và có cây con bên phải tượng trưng biểu thức E_r , thì cây con này tượng trưng biểu thức $(E_l E_r)$.

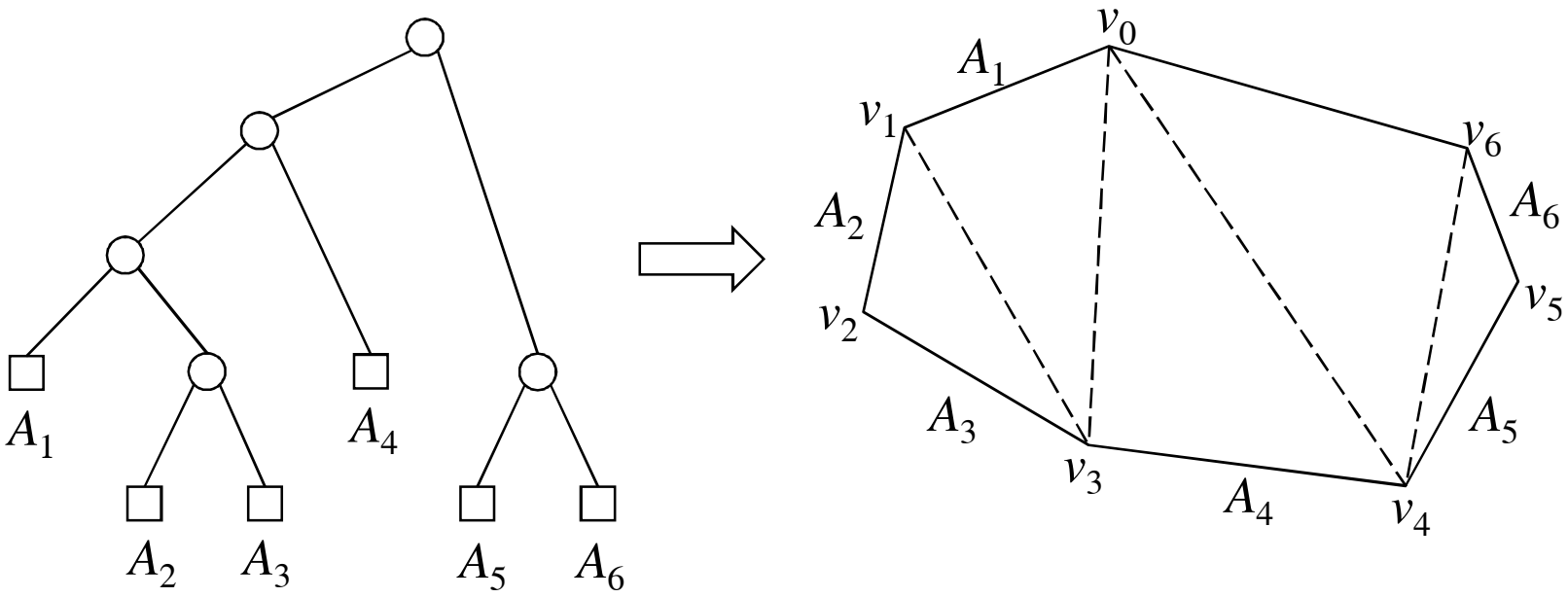
Từ phân tam giác sinh ra parse tree

- Ví dụ: Parse tree cho đa giác $P = \langle v_0, v_1, \ldots, v_6 \rangle$ sau.



Từ parse tree sinh ra phân tam giác

- Cho parse tree biểu diễn bởi $(((A_1(A_2A_3))A_4)(A_5A_6))$. Phân tam giác tương ứng:



Tương ứng giữa parse tree và phân chia tam giác

- Tương ứng giữa parse trees và các phân chia tam giác là tương ứng *một-đối-một*:
  - Mỗi phân tam giác của một đa giác lồi có $n +$ lcạnh tương ứng với parse tree cho một biểu thức gồm n nguyên tử.
  - Mỗi parse tree có n lá tương ứng với phân tam giác của một đa giác lồi có $n +$ lcạnh.

Tương ứng giữa đóng ngoặc hoàn toàn của tích của n ma trận và phân chia tam giác

- Đóng ngoặc hoàn toàn của tích của n ma trận tương ứng với phân tam giác của một đa giác lồi có $n +$ ldỉnh.
  - Mỗi ma trận A_i trong tích $A_1 A_2 \cdots A_n$ tương ứng với cạnh $v_{i-1} v_i$ của đa giác lồi.
  - Cung $v_i v_j$, với $i < j$, tương ứng với ma trận $A_{i+1..j}$.

Nhân chuỗi ma trận và phân tam giác tối ưu

- Bài toán nhân chuỗi ma trận là một trường hợp đặc biệt của bài toán phân tam giác tối ưu.
  - Tính tích $A_1 A_2 \cdots A_n$ thông qua một bài toán phân tam giác tối ưu:
    - Định nghĩa một đa giác lồi có $n + 1$ đỉnh $P = \langle v_0 , v_1, \ldots, v_n \rangle$
    - Nếu ma trận A_i có dimension $p_{i-1} \times p_i$, với $i = 1, 2, \ldots, n$, định nghĩa *hàm trọng số* w cho phân tam giác là

      $$w(\Delta v_i v_j v_k) = p_i p_j p_k$$

    - Một phân tam giác tối ưu của P cho ta parse tree của một đóng ngoặc tối ưu của $A_1 A_2 \cdots A_n$.

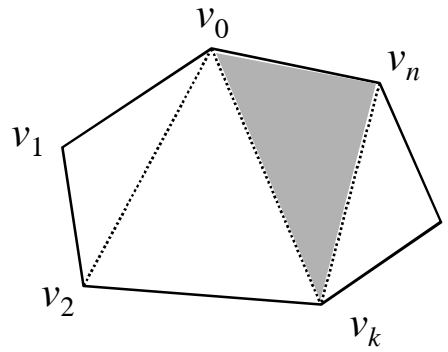
Nhân chuỗi ma trận và phân tam giác tối ưu (tiếp)

- Ngược lại là không đúng: bài toán phân tam giác tối ưu không là trường hợp đặc biệt của bài toán nhân chuỗi ma trận.
  - Mặc dù vậy, có thể chỉnh lại MATRIX-CHAIN-ORDER để giải bài toán phân tam giác tối ưu cho một đa giác có $n + 1$ đỉnh như sau
    - Thay chuỗi $p = < p_0 , p_1 ,..., p_n >$ của các chiều của ma trận bằng chuỗi $\langle v_0 , v_1 ,..., v_n \rangle$ của các đỉnh.
    - Thay các truy cập đến p bằng các truy cập đến v và thay dòng 9 bởi

      9 **do** $q \leftarrow m[i, k] + m[k + 1, j] + w(\Delta v_{i-1} v_k v_j)$
    - Khi giải thuật thực thi xong, $m[1, n]$ chứa trọng số của một phân tam giác tối ưu.
  - Phần sau cho thấy tại sao làm được như vậy.

Bước 1: Cấu trúc con của một phân tam giác tối ưu

- Cho T là một phân tam giác tối ưu của một đa giác $P = \langle v_0 , v_1,\ldots, v_n\rangle$, T chứa tam giác $\Delta v_0 v_k v_n$ với k nào đó, $1 \le k \le n - 1$.



- Trọng số của T là tổng của các trọng số của tam giác $\Delta v_0 v_k v_n$ và các tam giác chứa trong phân tam giác của hai đa giác con $\langle v_0, v_1,\ldots, v_k\rangle$ và $\langle v_k , v_{k+1},\ldots, v_n\rangle$. Một đa giác con suy thoái có trọng số là 0.
- Do đó các phân tam giác (xác định bởi T) của các đa giác con trên cũng phải là tối ưu. (Chứng minh bằng phản chứng.)

Bước 2: Lời giải đệ quy

- Định nghĩa $t[i, j]$ là trọng số của một phân tam giác tối ưu của đa giác $\langle v_{i-1}, v_i, \ldots, v_j \rangle$. Như vậy trọng số của một phân tam giác tối ưu của đa giác P là $t[1, n]$.
- Xác định $t[\cdot,\cdot]$
  - nếu đa giác chỉ có 2 đỉnh (đa giác suy thoái)

    $t[i, i] = 0$ cho $i = 1,..., n$
  - nếu đa giác có ít nhất 3 đỉnh, $i < j$

    $t[i, j] = t[i, k] + t[k + 1, j] + w(\Delta v_{i-1} v_k v_j)$.

    Nhưng trị của k?

Bước 2: Lời giải đệ quy (tiếp)

Bằng cách duyệt qua tất cả các trị của k, $i \le k \le j-1$, ta nhận được

$$t[i, i] = 0, \qquad i = 1,..., n$$

$$t[i, j] = \min_{i \le k \le j-1} \{t[i, k] + t[k+1, j] + w(\Delta v_{i-1} v_k v_j)\} \quad \text{nếu } i < j .$$

- Hàm đệ quy này tương tự hàm đệ quy $m[\cdot,\cdot]$ cho số phép nhân vô hướng tối thiểu để tính $A_i A_{i+1} \cdots A_j$. Do đó có thể chỉnh lại thủ tục MATRIX-CHAIN-ORDER (như đã nói) để tính trọng số của một phân tam giác tối ưu.
- Vậy thủ tục phân tam giác tối ưu chạy trong thời gian $O(n^3)$ và cần bộ nhớ $\Theta(n^2)$.

Bài toán chọn hoạt động (activity-selection problem)

- Cho
  - Một tập các *hoạt động* $S = \{1, 2, \ldots, n\}$
  - Một *tài nguyên chung* mà tại mọi thời điểm nó được dùng bởi nhiều lắm là một hoạt động
  - Hoạt động i có thời điểm bắt đầu là s_i và thời điểm chấm dứt là f_i
  - Nếu hoạt động i được chọn thì i tiến hành trong thời gian $[s_i, f_i)$
  - Hai hoạt động i và j là *tương thích nhau* ("*compatible*") nếu $[s_i, f_i)$ và $[s_j, f_j)$ không chạm nhau.
- *Bài toán chọn hoạt động* là tìm một tập các hoạt động tương thích nhau mà có số phần tử lớn nhất.

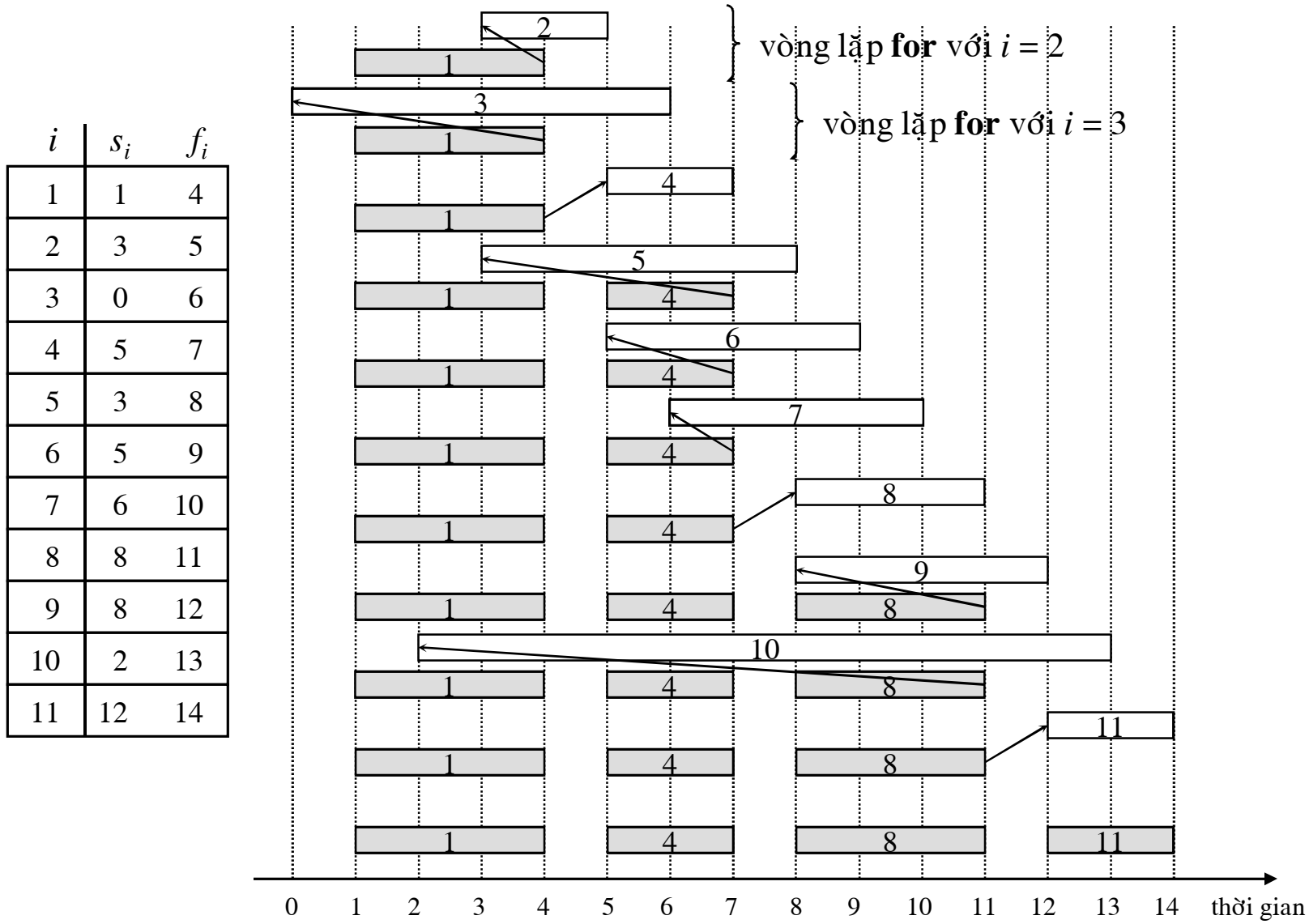
Giải thuật greedy cho bài toán chọn hoạt động

- Giải thuật
  - Giả sử: $f_1 \le f_2 \le \ldots \le f_n$.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1    n ← length[s]
2    A ← {1}
3    j ← 1
4    for i ← 2 to n
5         do if s_i ≥ f_j
6              then A ← A ∪ {i}
7                   j ← i
8    return A
```

Chạy GREEDY-ACTIVITY-SELECTOR lên một ví dụ

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



Correctness của giải thuật greedy cho bài toán chọn hoạt động

- **Định lý** Giải thuật GREEDY-ACTIVITY-SELECTOR tìm được lời giải tối ưu cho bài toán chọn hoạt động.
- **Chứng minh**
  - Gọi $S = \{1, 2, \ldots, n\}$ là tập hợp các hoạt động. Các hoạt động được xếp thứ tự theo thời điểm chấm dứt. Do đó hoạt động 1 có thời điểm chấm dứt sớm nhất.
  - Ta chứng minh có lời giải tối ưu bắt đầu bằng hoạt động do chọn lựa greedy, tức là bắt đầu bằng hoạt động 1:
    - Giả sữ $A \subseteq S$ là lời giải tối ưu, ta xếp thứ tự các hoạt động trong A theo thời điểm chấm dứt. Giả sữ k là hoạt động đầu tiên trong A.
    - Nếu $k = 1$ thì ta xong. Nếu $k \neq 1$, ta xét $B = A - \{k\} \cup \{1\}$. Vì $f_1 \leq f_k$, nên B cũng là lời giải tối ưu.
  - Nếu A là lời giải tối ưu cho bài toán nguyên thủy S, thì $A' = A - \{1\}$ là lời giải tối ưu cho bài toán $S' = \{i \in S : s_i \geq f_1\}$.

Phân Tích Khấu Hao

Phân tích khấu hao

- Gọi $T(n)$ là thời gian cần thiết để thực thi một chuỗi n thao tác lên một cấu trúc dữ liệu.
  - Ví dụ: thực thi một chuỗi PUSH, POP, MULTIPOP lên một stack.
- *Phân tích khấu hao* (amortized analysis):
  - Thời gian thực thi một chuỗi các thao tác được lấy trung bình trên số các thao tác đã thực thi,

    $T(n)/n$,

    được gọi là *phí tổn khấu hao*.

    (Đây chỉ là một trong các định nghĩa của phí tổn khấu hao, các định nghĩa khác sẽ được trình bày trong các phần sau.)

Sơ lược

- Ba phương pháp để xác định phí tổn khấu hao:
  - *gộp chung* (the aggregate method)
  - *kế toán* (the accounting method)
  - *thế năng* (the potential method)
- Sẽ minh họa các phương pháp trên thông qua việc phân tích các cấu trúc dữ liệu:
  - stack
  - bộ đếm nhị phân dài k bits
  - bảng động.

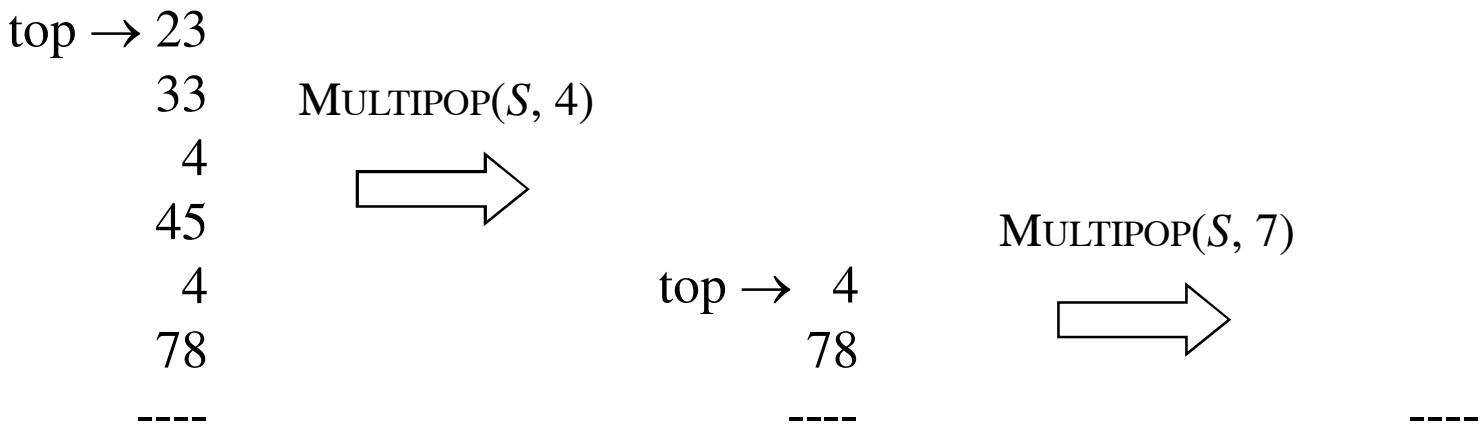
Cấu trúc dữ liệu stack

- Các thao tác lên một stack S
  - PUSH(S, x)
  - POP(S)
  - MULTIPOP(S, k)

```
MULTIPOP(S, k)
1   while not STACK-EMPTY(S) and k ≠ 0
2         do POP(S)
3            k ← k – 1
```

Cấu trúc dữ liệu stack (tiếp)

- Minh họa thao tác MULTIPOP



Bộ đếm nhị phân dài k bit

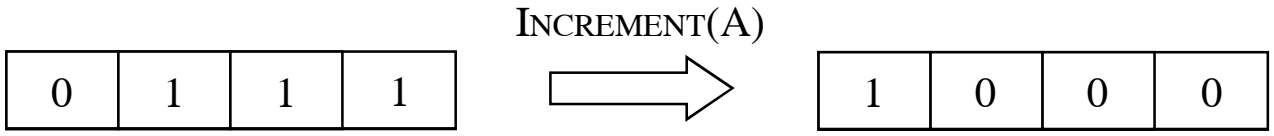
- Bộ đếm nhị phân dài k-bit (k-bit binary counter)
  - là một mảng $A[0..k-1]$ của các bit
  - có độ dài, $length[A]$, là k.

Bộ đếm nhị phân dài k bit (tiếp)

- Dùng bộ đếm để trữ một số nhị phân x:

  $x = A[k-1]\cdot 2^{k-1} + \ldots + A[0]\cdot 2^0$

  – INCREMENT: cộng 1 vào trị đang có trong bộ đếm (modulo 2^k)



  – Thủ tục INCREMENT

```
INCREMENT(A)
1  i ← 0
2  while i < length[A] and A[i] = 1
3        do A[i] ← 0
4           i ← i + 1
5  if i < length[A]
6     then A[i] ← 1
```

Phân tích một stack

- Bài toán: xác định thời gian chạy của một chuỗi n thao tác lên một stack (ban đầu stack là trống).

  Giải bằng phân tích "thô sơ":

  – Phí tổn trong trường hợp xấu nhất của MULTIPOP là $O(n)$. Vậy phí tổn trong trường hợp xấu nhất của một thao tác bất kỳ lên stack là $O(n)$.

  – Do đó phí tổn của một chuỗi n thao tác là $O(n^2)$.

- Nhận xét: Chận $O(n^2)$ tìm được là quá thô.
- Tìm chận trên tốt hơn!
  – Dùng phân tích khấu hao.

Phương pháp gộp chung

- Định nghĩa: Trong phương pháp *gộp chung* (*aggregate*) của phân tích khấu hao, chúng ta gộp chung thời gian mà một chuỗi n thao tác cần trong trường hợp xấu nhất (worst case) thành $T(n)$.
  - *Chi phí khấu hao* (*amortized cost*) của mỗi thao tác được định nghĩa là chi phí trung bình cho mỗi thao tác, tức là $T(n)/n$.

Phương pháp gộp chung: phân tích stack

- Phân tích một chuỗi n thao tác lên một stack S (mà khi bắt đầu thì stack là trống), chuỗi gồm các loại thao tác
  - PUSH(S, x)
  - POP(S)
  - MULTIPOP(S, k)
- Dùng phương pháp gộp chung để xác định chi phí khấu hao của mỗi thao tác
  - Mỗi đối tượng có thể được popped tối đa là một lần sau khi nó được pushed. Số PUSH tối đa là n, vậy số lần POP được gọi, kể cả từ MULTIPOP, là n.
  - Vậy phí tổn của chuỗi n thao tác PUSH, POP, và MULTIPOP là $O(n)$.
  - Do đó phí tổn khấu hao của mỗi thao tác là $O(n)/n = O(1)$.

Phương pháp gộp chung: phân tích bộ đếm nhị phân

- Phân tích một chuỗi gồm n thao tác INCREMENT lên một bộ đếm nhị phân có chiều dài k bit
- Dùng phương pháp gộp chung để xác định chi phí khấu hao của mỗi thao tác.
  - Nhận xét (giả sử trị ban đầu của bộ đếm là 0)

    bit $A[0]$ flips n lần

    bit $A[1]$ $\lfloor n/2 \rfloor$

    bit $A[2]$ $\lfloor n/4 \rfloor$

    ...

    Tổng quát:

    bit $A[i]$ flips $\lfloor n/2^i \rfloor$ lần, $i = 0,\ldots,\lfloor \lg n \rfloor$

    bit $A[i]$ không flips nếu $i > \lfloor \lg n \rfloor$.
  - Tính được tổng của $\lfloor n/2^i \rfloor$ từ $i = 0$ đến $\lfloor \lg n \rfloor$ là $\leq 2n$.

Phương pháp gộp chung: phân tích bộ đếm nhị phân

- Dùng phương pháp gộp chung để xác định chi phí khấu hao của mỗi thao tác (tiếp)
  - Vậy thời gian xấu nhất cho một chuỗi n thao tác INCREMENT lên một bộ đếm (mà trị ban đầu là 0) là $O(n)$.

    $\Rightarrow$ Thời gian khấu hao cho mỗi thao tác là $O(n)/n = O(1)$.
- Nhận xét: Phân tích thô sơ
  - Khi mọi bit của bộ đếm là 1, thao tác INCREMENT có thể cần đến thời gian xấu nhất là $\Theta(k)$.
  - Do đó một chuỗi n thao tác INCREMENT cần thời gian xấu nhất là $nO(k) = O(nk)$.

Phương pháp kế toán

- Trong phương pháp *kế toán* của phân tích khấu hao, chúng ta định giá (charge) khác nhau cho các thao tác khác nhau.

  Ta có thể định giá cho một thao tác cao hơn hay thấp hơn phí tổn thực sự của nó.

  – Định nghĩa: Số lượng mà ta định giá cho một thao tác được gọi là *phí tổn khấu hao* của nó.

  – Định nghĩa *chi phí trả trước* hay *credit* là chi phí khấu hao trừ chi phí thực sự.

    - Để cho chi phí khấu hao tổng cộng là chận trên lên chi phí thực sự tổng cộng thì tại mọi thời điểm credit tổng cộng phải ≥ 0.
    - Nhưng credit cho một thao tác cá biệt có thể < 0 .

Phương pháp kế toán: phân tích stack

- Các thao tác lên một stack
  - PUSH(S, x)
  - POP(S)
  - MULTIPOP(S, k) (POP k phần tử từ stack S)
- Dùng phương pháp kế toán để xác định chi phí khấu hao của mỗi thao tác lên một stack (ban đầu stack là trống).
  - Nhắc lại: phí tổn thực sự của các thao tác là
    - PUSH 1
    - POP 1
    - MULTIPOP min(k, s), (s là số phầntử trong S khi được gọi)
  - Ta quy cho các thao tác các phí tổn khấu hao như sau
    - PUSH 2
    - POP 0
    - MULTIPOP 0 là một hằng số.

Phương pháp kế toán: phân tích stack (tiếp)

– Ta chứng tỏ rằng có thể trả cho một chuỗi thao tác bất kỳ khi tính giá theo các phí tổn khấu hao. (Dùng 1 đồng để tượng trưng 1 đơn vị phí tổn.)

  Mô hình stack bằng một chồng đĩa.

  - Giả sử thao tác là PUSH: trả 2 đồng, trong đó
    – dùng 1 đồng để trả cho phí tổn thực sự
    – dùng 1 đồng còn lại để trả trước phí tổn cho POP sau này (nếu có). Để đồng này trong đĩa được pushed vào chồng đĩa.
  - Giả sử thao tác là POP: không cần trả, mà
    – dùng 1 đồng đã được trả trước khi trả cho PUSH. Đồng này nằm trong đĩa được popped khỏi chồng đĩa.
  - Giả sử thao tác là MULTIPOP: không cần trả, mà
    – dùng 1 đồng đã được trả trước khi trả cho PUSH.

Phương pháp kế toán: phân tích stack (tiếp)

- Kết luận: Cho một chuỗi bất kỳ gồm n thao tác PUSH, POP, và MULTIPOP, phí tổn khấu hao tổng cộng là một cận trên cho phí tổn thực sự tổng cộng.
  - Vì mỗi thao tác có phí tổn khấu hao là $O(1)$ nên một cận trên cho phí tổn thực sự tổng cộng là $O(n)$.

Phương pháp kế toán: phân tích một bộ đếm nhị phân

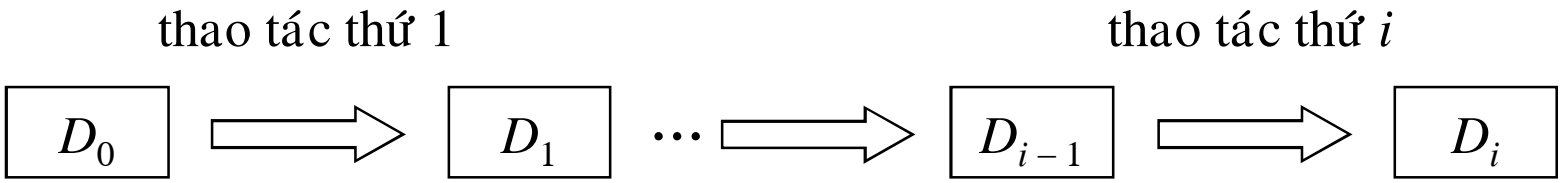
- Phân tích một chuỗi các thao tác INCREMENT lên một bộ đếm nhị phân dài *k*-bit mà trị ban đầu là 0.
- Dùng phương pháp kế toán để xác định chi phí khấu hao của INCREMENT
  - Quy một phí tổn khấu hao là 2 đồng để set một bit thành 1.
    - dùng 1 đồng để trả phí tổn thực sự
    - dùng 1 đồng còn lại để trả trước cho phí tổn để reset bit này thành 0 sau này (nếu có).

Phương pháp kế toán: phân tích một bộ đếm nhị phân

- Xác định phí tổn khấu hao của INCREMENT (tiếp)
  - Phí tổn khấu hao của INCREMENT:
    - Phí tổn cho resetting các bits trong vòng lặp while được trả bằng các đồng đã được trả trước khi bit được set.
    - Nhiều nhất là 1 bit được set.
  - Vậy phí tổn khấu hao của INCREMENT tối đa là 2 đồng.
- Vậy chi phí khấu hao cho n thao tác INCREMENT là $O(n)$.
  - Vì tổng số tiền trả trước không bao giờ âm (= số các bit có trị là 1 trong bộ đếm) nên chi phí khấu hao tổng cộng là cận trên cho chi phí thực sự tổng cộng.

Phương pháp thế năng

- Phân tích một chuỗi các thao tác lên một cấu trúc dữ liệu.
  - Cho một cấu trúc dữ liệu mà n thao tác thực thi lên đó. Ban đầu cấu trúc dữ liệu là D_0
  - Gọi c_i là chi phí thực sự của thao tác thứ i, với $i = 1,..., n$.
  - Gọi D_i là cấu trúc dữ liệu có được sau khi áp dụng thao tác thứ i lên cấu trúc dữ liệu D_{i-1}, với $i = 1,..., n$.



Phương pháp thế năng (tiếp)

- Định nghĩa một hàm số

  $\Phi : \{D_0, ..., D_n\} \to \Re$, với $\Re$ là tập hợp các số thực.

  Hàm Φ được gọi là (*hàm*) *thế năng* (*potential function*).

  Trị $\Phi(D_i)$ được gọi là *thế năng* của cấu trúc dữ liệu D_i, $i = 0,...,n$.

- Định nghĩa: *phí tổn khấu hao* (*amortized cost*) của thao tác thứ i là

  $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ (*)

Phương pháp thế năng (tiếp)

- Điều kiện cho thế năng để phí tổn khấu hao tổng cộng là cận trên lên phí tổn thực sự tổng cộng:
  - Ta có từ (*)

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

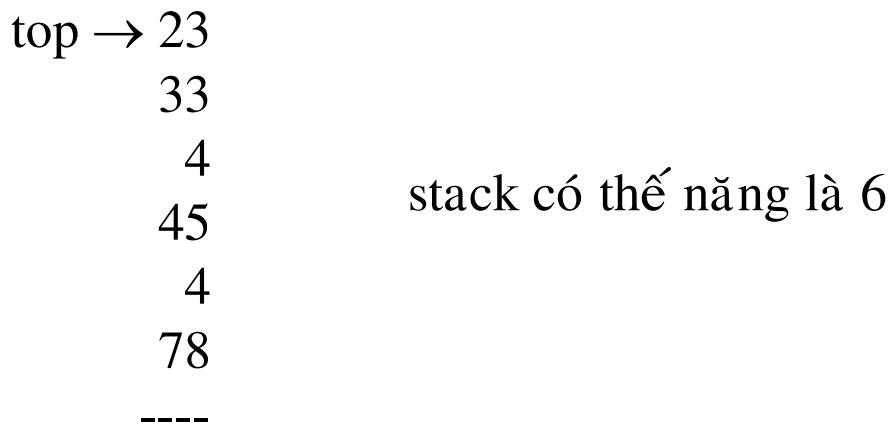
$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

  - Vậy phí tổn khấu hao tổng cộng là cận trên của phí tổn thực sự tổng cộng nếu $\Phi(D_n) - \Phi(D_0) \geq 0$. Vì không biết trước n nên ta có điều kiện sau

$$\Phi(D_i) - \Phi(D_0) \geq 0 \qquad \text{cho mọi } i.$$

Phương pháp thế năng: phân tích một stack

- Phân tích một chuỗi gồm n thao tác lên một stack
  - PUSH(S, x)
  - POP(S)
  - MULTIPOP(S, k).
- Áp dụng phương pháp thế năng để xác định chi phí khấu hao của mỗi thao tác
  - Định nghĩa: thế năng Φ của một stack là số đối tượng trong stack.



Phương pháp thế năng: phân tích một stack (tiếp)

- Nhận xét:
  - Khi bắt đầu thì stack trống nên $\Phi(D_0) = 0$.
  - $\Phi(D_i) \geq 0$, vậy $\Phi(D_i) \geq \Phi(D_0)$ cho mọi i.

  Vậy phí tổn khấu hao tổng cộng là cận trên của phí tổn thực sự tổng cộng.

- Áp dụng phương pháp thế năng để xác định chi phí khấu hao của mỗi thao tác
  - Giả sử thao tác thứ i lên stack là PUSH

    (stack có kích thước là s)
    - Hiệu thế là $\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s$

      $= 1$
    - Vậy phí tổn khấu hao của PUSH

      $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

      $= 1 + 1 = 2$.

Phương pháp thế năng: phân tích một stack

- Áp dụng phương pháp thế năng để xác định chi phí bù trừ của mỗi thao tác (tiếp)
  - Giả sử thao tác thứ i lên stack là MULTIPOP(S, k)

    (stack có kích thước là s)
    - Phí tổn thực sự là $k' = \min(k, s)$
    - Hiệu thế là $\Phi(D_i) - \Phi(D_{i-1}) = -k'$
    - Vậy phí tổn khấu hao của MULTIPOP là

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 . \end{aligned}$$

Phương pháp thế năng: phân tích một stack

- Áp dụng phương pháp thế năng để xác định chi phí bù trừ của mỗi thao tác (tiếp)
  - Phí tổn khấu hao của POP là 0.
  - Phí tổn khấu hao của mỗi thao tác lên stack là $O(1)$.
- Vậy phí tổn khấu hao tổng cộng của một chuỗi n thao tác lên stack là $O(n)$.

  Đã thấy là $\Phi(D_i) \geq \Phi(D_0)$ cho mọi i, vậy phí tổn trong trường hợp xấu nhất của n thao tác là $O(n)$.

Phương pháp thế năng: phân tích bộ đếm nhị phân dài *k* bits

- Phân tích một chuỗi các thao tác lên một bộ đếm nhị phân dài *k*-bit.
- Dùng phương pháp thế năng để xác định chi phí khấu hao của mỗi thao tác
  - Định nghĩa *thế năng* Φ *của bộ đếm* sau thao tác INCREMENT thứ *i* là b_i, số các bits bằng 1 trong bộ đếm.

Thế năng là 3

Phương pháp thế năng: phân tích bộ đếm nhị phân dài k bits

- Dùng phương pháp thế năng để xác định chi phí khấu hao của mỗi thao tác (tiếp)
  - Tính phí tổn khấu hao của một thao tác INCREMENT
    - INCREMENT thứ i resets t_i bits và sets nhiều lắm là 1 bit.

      Vậy phí tổn thực sự của INCREMENT thứ i nhiều lắm là $t_i + 1$.
    - Hiệu thế là

      $\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1}$, mà $b_i \le b_{i-1} - t_i + 1$. Vậy

      $\Phi(D_i) - \Phi(D_{i-1}) \le 1 - t_i$.
    - Phí tổn khấu hao là

      $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le t_i + 1 + 1 - t_i = 2$.

Phương pháp thế năng: phân tích bộ đếm nhị phân dài k bits

- Trị của bộ đếm bắt đầu bằng 0, nên $\Phi(D_0) = 0$, do đó $\Phi(D_i) \geq \Phi(D_0)$. Vậy chi phí khấu hao tổng cộng là chận trên cho chi phí thực sự tổng cộng.

  $\Rightarrow$ Phí tổn trong trường hợp xấu nhất của n thao tác là $O(n)$.

Bảng động

- Trong một số ứng dụng, dùng một "bảng" để trữ các đối tượng mà không biết trước bao nhiêu đối tượng sẽ được trữ. Do đó
  - khi bảng hiện thời không có đủ chổ cho các đối tượng mới, cần một bảng mới với kích thước lớn hơn.
  - khi bảng hiện thời dư nhiều chổ trống do xoá nhiều đối tượng, cần một bảng mới với kích thước nhỏ hơn.
- Các thao tác lên một bảng
  - TABLE-INSERT: chèn một item vào bảng
  - TABLE-DELETE: xóa một item khỏi bảng.

Hệ số sử dụng của một bảng

- Định nghĩa *hệ số sử dụng* (load factor) của một bảng T (không trống) là $\alpha(T)$:

  $\alpha(T)$ = số khe (slots) có chứa đối tượng trong bảng chia cho số khe của bảng.

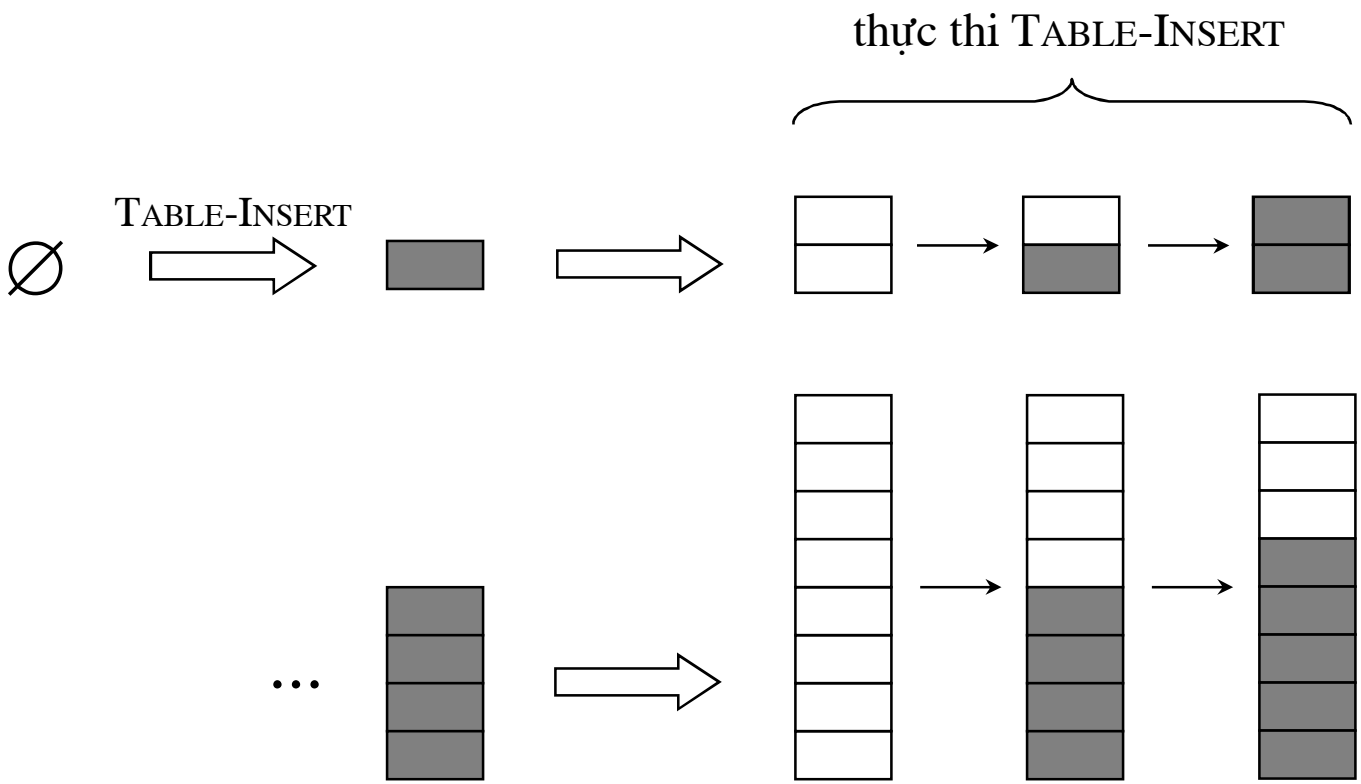
- Bài toán: Xác định chiến lược *nới rộng* và *thu nhỏ* bảng sao cho
  - hệ số sử dụng của bảng *cao*
    - được chận dưới bởi một hằng số
  - chi phí khấu hao của TABLE-INSERT và TABLE-DELETE là $O(1)$.

Chiến lược mở rộng một bảng

- Chiến lược khi nào thì nới rộng một bảng được diễn tả trong giải thuật sau.

```
TABLE-INSERT(T, x)
 1   if size[T] = 0
 2      then allocate table[T] with 1 slot
 3           size[T] ← 1
 4   if num[T] = size[T]
 5      then allocate new-table with 2 · size[T] slots
 6           insert all items in table[T] into new-table
 7           free table[T]
 8           table[T] ← new-table
 9           size[T] ← 2 · size[T]
10   insert x into table[T]
11   num[T] ← num[T] + 1
```

Chiến lược mở rộng một bảng (tiếp)



Phân tích một chuỗi TABLE-INSERT

- Giả sử:
  - Thời gian thực thi của TABLE-INSERT tỉ lệ với thời gian chèn từng item ("chèn sơ đẳng") vào bảng ở dòng 6 và 10.
  - Chi phí của một chèn sơ đẳng là 1.
- Sẽ phân tích chi phí của một chuỗi gồm n INSERT lên một bảng động dùng lần lượt các phương pháp
  - gộp chung
  - kế toán
  - thế năng.

Phân tích chuỗi TABLE-INSERT bằng phương pháp gộp chung

- Dùng phương pháp gộp chung để xác định chi phí khấu hao của INSERT
  - Chi phí c_i của thao tác thứ i
    - là i nếu $i - 1 = 2^j$
    - là 1 trong các trường hợp còn lại.
  - Chi phí của n thao tác TABLE-INSERT
    - là n + tổng các 2^j từ $j = 0$ đến $\lfloor \lg n \rfloor$

      $\leq n + 2n = 3n$.
- Vậy chi phí khấu hao của INSERT là $3n / n = 3$.

Phân tích chuỗi TABLE-INSERT bằng phương pháp kế toán

- Dùng phương pháp kế toán để xác định chi phí khấu hao của TABLE-INSERT
  - Chi phí khấu hao của TABLE-INSERT là 3.
  - Trả như sau:
    - 1 đồng để trả cho chi phí thực sự cho riêng nó
    - 1 đồng để trả trước cho chính nó một khi nó được di chuyển lúc bảng nới rộng
    - 1 đồng để trả trước cho một item khác trong bảng mà không còn tiền trả trước (vì đã di chuyển).

Phân tích chuỗi TABLE-INSERT bằng phương pháp thế năng

- Dùng phương pháp thế năng để phân tích một chuỗi gồm n thao tác INSERT lên một bảng
  - Định nghĩa thế năng Φ là

    $\Phi(T) = 2num[T] - size[T]$.
  - Nhận xét
    - Ngay *sau* khi nới rộng (dòng 9 của TABLE-INSERT thực thi xong)
      - $num[T] = size[T] / 2$
      - $\Phi(T) = 0$.
    - Ngay *trước* khi nới rộng $num[T] = size[T]$
      - $\Phi(T) = num[T]$.
    - $\Phi(0) = 0$, $\Phi(T) \geq 0$. Vì vậy tổng của các chi phí khấu hao của n thao tác TABLE-INSERT là một chận trên lên tổng của các chi phí thực sự.

Phân tích chuỗi TABLE-INSERT bằng phương pháp thế năng

(tiếp)

- Xác định chi phí khấu hao của mỗi thao tác
  - Giả sử thao tác thứ i không gây nới rộng. Ta có $size_i = size_{i-1}$.
  - Chi phí khấu hao của thao tác là

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\ &= 1 + (2num_i - size_i) - (2(num_i - 1) - size_i) \\ &= 3 . \end{aligned}$$

Phân tích chuỗi TABLE-INSERT bằng phương pháp thế năng

– Xác định chi phí khấu hao của mỗi thao tác (tiếp)

- Giả sử thao tác thứ i gây nới rộng. Ta có

$$size_i / 2 = size_{i-1}$$
$$= num_i - 1 .$$

Chi phí khấu hao của thao tác là

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$
$$= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1})$$
$$= num_i + (2num_i - (2num_i - 2)) - (2(num_i - 1) - (num_i - 1))$$
$$= num_i + 2 - (num_i - 1)$$
$$= 3 .$$

Xóa một item khỏi bảng

- Thêm thao tác "*Xóa một item khỏi bảng*": TABLE-DELETE.
  - Hệ số sử dụng của bảng có thể trở nên quá nhỏ.
- Nhắc lại định nghĩa của *hệ số sử dụng* là

  $\alpha(T) = num[T] / size[T]$.
- Ta muốn
  - giử hệ số sử dụng cao, tức là nó được chận dưới bằng một hằng số.
  - chi phí bù trừ của một thao tác lên bảng được chận trên bằng một hằng số.
- Giả sử chi phí được đo bằng số lần chèn hay xoá item sơ đẳng.

Chiến lược nới rộng và thu nhỏ bảng

- Một chiến lược tự nhiên cho nới rộng và thu nhỏ bảng là
  - Gấp đôi bảng khi chèn một item vào một bảng đã đầy.
  - Giảm nửa bảng khi xóa một item khỏi một bảng chỉ đầy nửa bảng.
- Phân tích
  - Chiến lược trên bảo đảm $\alpha(T) \geq 1/2$.

Chiến lược nới rộng và thu nhỏ bảng

- Phân tích (tiếp)
  - Tuy nhiên phí tổn khấu hao của mỗi thao tác có thể rất lớn:
    - Lấy n có dạng 2^m
    - Xét một chuỗi n thao tác (I là một insert và D là một delete)

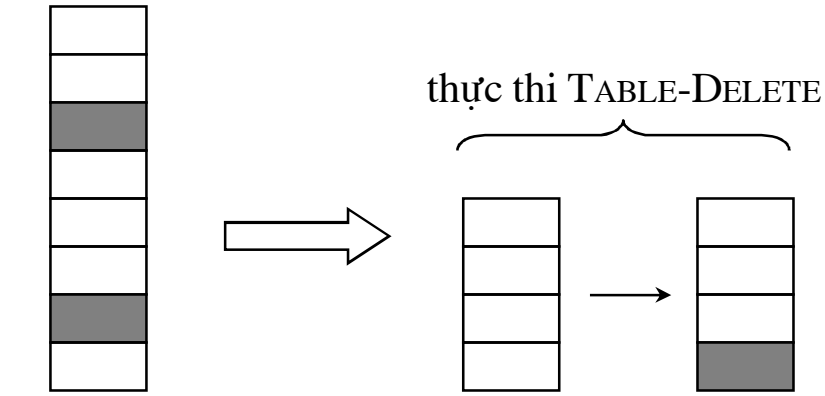
      I … I ($n/2$ lần), kế đó là

      I D D I I D D I I … ($n/2$ lần).
    - Chuỗi n thao tác này có phí tổn là $\Theta(n^2)$, do đó phí tổn khấu hao của mỗi thao tác là $\Theta(n)$.

Chiến lược nới rộng và thu nhỏ bảng (tiếp)

- Cải tiến chiến lược trên bằng cách cho phép hệ số sử dụng có thể trở nên nhỏ hơn 1/2:
  - Nếu $\alpha(T) = 1$, thì thao tác TABLE-INSERT sẽ gấp đôi bảng.
  - Nếu $\alpha(T) = 1/4$, thì thao tác TABLE-DELETE sẽ giảm nửa bảng.

Chiến lược thu nhỏ một bảng (tiếp)



$\alpha(T) = 1/4$

bảng mới =
1/2 bảng cũ

Phương pháp thế năng

- Dùng phương pháp thế năng để phân tích một chuỗi gồm n thao tác TABLE-INSERT và TABLE-DELETE lên một bảng.
  - Định nghĩa *thế năng* Φ *trên một bảng* là

$$\Phi(T) = 2\,num[T] - size[T] \qquad \text{nếu } \alpha(T) \geq 1/2$$
$$= size[T] / 2 - num[T] \qquad \text{nếu } \alpha(T) < 1/2$$

Phương pháp thế năng (tiếp)

- Nhận xét:
  - Φ(bảng trống) = 0, và $\Phi(T) \geq 0$
  - Nếu hệ số sử dụng là 1/2 thì $\Phi(T) = 0$.
  - Nếu hệ số sử dụng là 1 thì $\Phi(T) = num[T]$.
    - Đủ để trả phí tổn một khi có nới rộng bảng do chèn một item.
  - Hệ số sử dụng là 1/4 thì $\Phi(T) = num[T]$.
    - Đủ để trả phí tổn một khi có thu nhỏ bảng do xoá một item.

Phân tích một chuỗi các TABLE-INSERT và TABLE-DELETE

- Xác định chi phí khấu hao của mỗi thao tác
  - Nếu thao tác thứ i là TABLE-INSERT, ta phân biệt các trường hợp:
    - $\alpha_{i-1} \geq 1/2$
      - theo Section 18.4.1, thì $\hat{c}_i$ nhiều lắm là 3.
    - $\alpha_{i-1} < 1/2$, ta phân biệt 2 trường hợp:
      - trường hợp $\alpha_i < 1/2$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 0 .\end{aligned}$$

      - trường hợp $\alpha_i \geq 1/2$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 3 .\end{aligned}$$

  - Vậy chi phí khấu hao của thao tác TABLE-INSERT nhiều lắm là 3.

Phân tích một chuỗi các TABLE-INSERT và TABLE-DELETE

- Xác định chi phí khấu hao của mỗi thao tác (tiếp)
  - Nếu thao tác thứ i là TABLE-DELETE, thì $num_i = num_{i-1} - 1$, ta phân biệt các trường hợp:
    - $\alpha_{i-1} < 1/2$. Có hai trường hợp con
      - không gây thu nhỏ: $size_i = size_{i-1}$

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 2 . \end{aligned}$$

      - gây thu nhỏ: $c_i = num_i + 1$, $size_i / 2 = size_{i-1} / 4 = num_i + 1$

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 . \end{aligned}$$

    - $\alpha_{i-1} \geq 1/2$. Bài tập 18.4-3.
  - Vậy chi phí khấu hao của TABLE-DELETE được chận trên bởi một hằng số.

Phân tích một chuỗi các TABLE-INSERT và TABLE-DELETE (tiếp)

- Kết luận: Vì chi phí khấu hao của mỗi thao tác TABLE-INSERT và TABLE-DELETE được chận trên bởi một hằng số, nên thời gian chạy cho một chuỗi bất kỳ gồm n thao tác lên một bảng động là $O(n)$.

B-Cây

Cấu trúc dữ liệu trong bộ nhớ ngoài

- B-cây tổng quát hoá cây tìm kiếm nhị phân.
  - "Hệ số phân nhánh" (branching factor)
- B-cây là cây tìm kiếm cân bằng được thiết kế để làm việc hữu hiệu trong bộ nhớ ngoài (đĩa cứng).
  - Bộ nhớ chính (main memory)
  - Bộ nhớ ngoài (secondary storage)
    - Disk
      - Track
      - Page
- Thời gian chạy gồm
  - số các truy cập vào đĩa
  - thời gian CPU

Truy cập đĩa

- Một nút của B-cây thường chiếm nguyên cả một disk page.
- Hệ số phân nhánh tùy thuộc vào tỉ lệ giữa kích thước của khóa và kích thước của disk page.

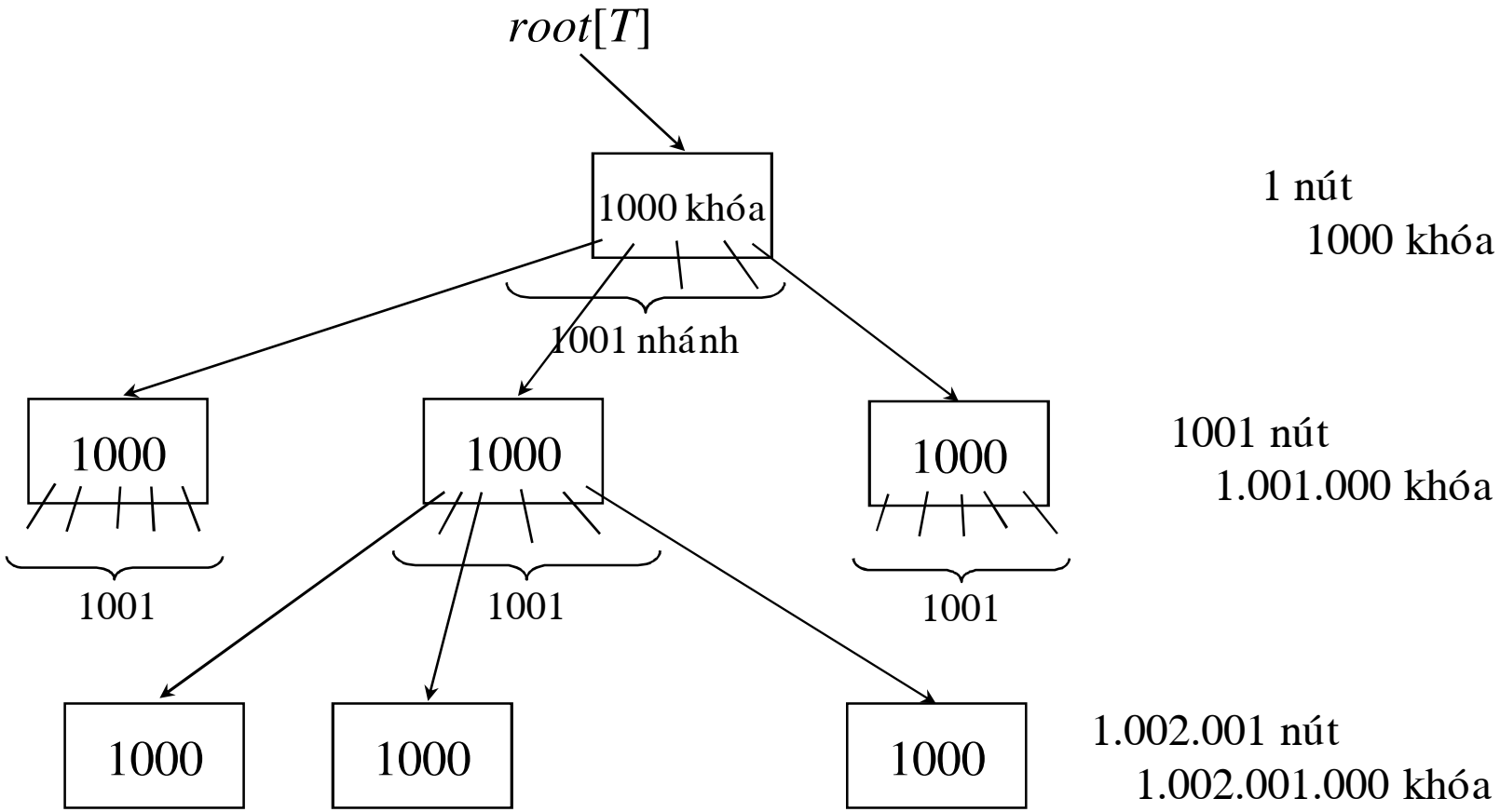
Các thao tác lên một đĩa

- Cho x là một con trỏ đến một đối tượng (ví dụ: một nút của một B-cây). Đối tượng x có thể có nhiều trường
  - Nếu x nằm trong bộ nhớ chính, truy cập các trường của x như thường lệ, ví dụ như *key*[x], *leaf* [x],...
  - Nếu x còn nằm trên đĩa thì dùng DISK-READ(x) để đọc nó vào bộ nhớ chính.
  - Nếu x đã thay đổi thì dùng DISK-WRITE(x) để trữ nó vào đĩa.
- Cách làm việc tiêu biểu với một đối tượng x

```
...
x ← một con trỏ đến một đối tượng nào đó
DISK-READ(x)
các thao tác truy cập/thay đổi các trường của x
DISK-WRITE(x)
các thao tác không thay đổi một trường của x
...
```

Hệ số phân nhánh

- Ví dụ một B-cây mà:
  - mỗi nút có 1000 khóa, tức là B-cây có hệ số phân nhánh là 1001



Định nghĩa của B-cây

- Một B-cây T là một cây có gốc, mà gốc là $root[T]$, có các tính chất sau
  - Mỗi nút x có các trường sau
    - $n[x]$, số lượng khóa đang được chứa trong nút x
    - các khóa: có $n[x]$ khóa, được xếp theo thứ tự không giảm, tức là

      $$key_1[x] \le key_2[x] \le \cdots \le key_{n[x]}[x]$$

    - $leaf\,[x]$, có trị bool là

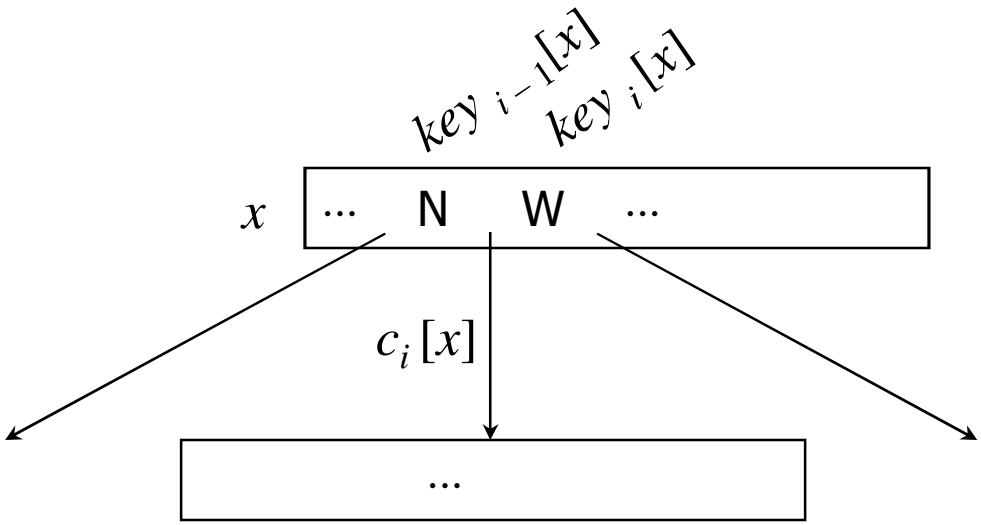
      TRUE nếu x là một lá

      FALSE nếu x là một nút trong
  - Mỗi nút trong x chứa $n[x] + 1$ con trỏ $c_1\,[x]$, $c_2\,[x]$,…, $c_{n[x]+1}[x]$ đến các nút con của nó.

Định nghĩa của B-cây

(tiếp)

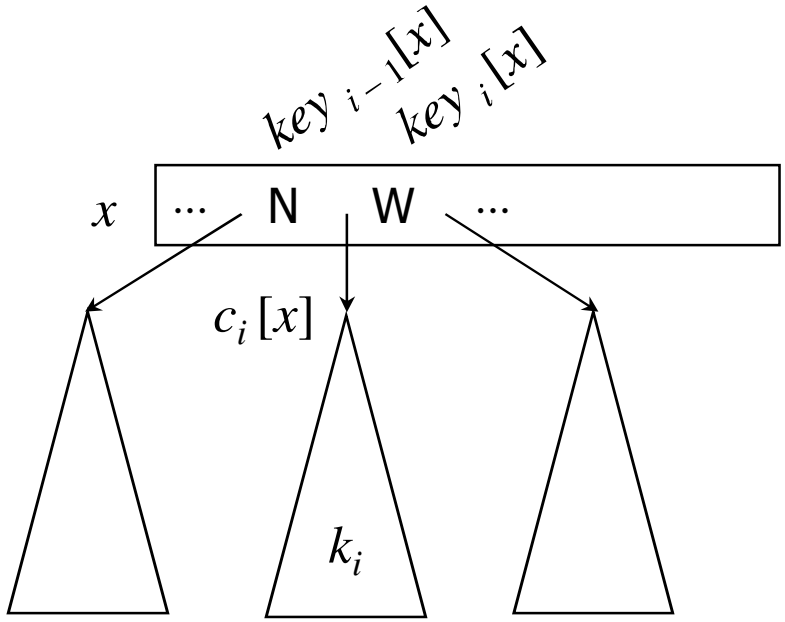
Mô hình một nút của B-cây



Định nghĩa của B-cây

(tiếp)

- Nếu k_i là khóa trữ trong cây con có gốc là $c_i[x]$ thì
  - $k_1 \le key_1[x] \le k_2 \le key_2[x] \le \cdots \le k_{n[x]} \le key_{n[x]}[x] \le k_{n[x]+1}$



Định nghĩa của B-cây

(tiếp)

- Tất cả các lá có cùng một *độ sâu*, đó là chiều cao h của cây
- Có một số nguyên $t \geq 2$ gọi là *bậc tối thiểu* của cây sao cho
  - Mọi nút không phải là nút gốc phải có ít nhất $t - 1$ khóa. Nếu cây $\neq \emptyset$ thì nút gốc phải có ít nhất một khóa.
  - Mỗi nút có thể chứa tối đa $2t - 1$ khóa. Một nút là *đầy* nếu nó chứa đúng $2t - 1$ khóa.

Chiều cao của một B-cây

Định lý

Nếu $n \geq 1$ thì mọi B-cây T với n khóa, chiều cao h, và bậc tối thiểu $t \geq 2$ có

$$h \leq \log_t \frac{n+1}{2}$$

Chứng minh

Có tối thiểu 2 nút ở độ sâu 1, $2t$ nút ở độ sâu 2,..., và $2t^{h-1}$ nút ở độ sâu h. Vậy số khóa tối thiểu là

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

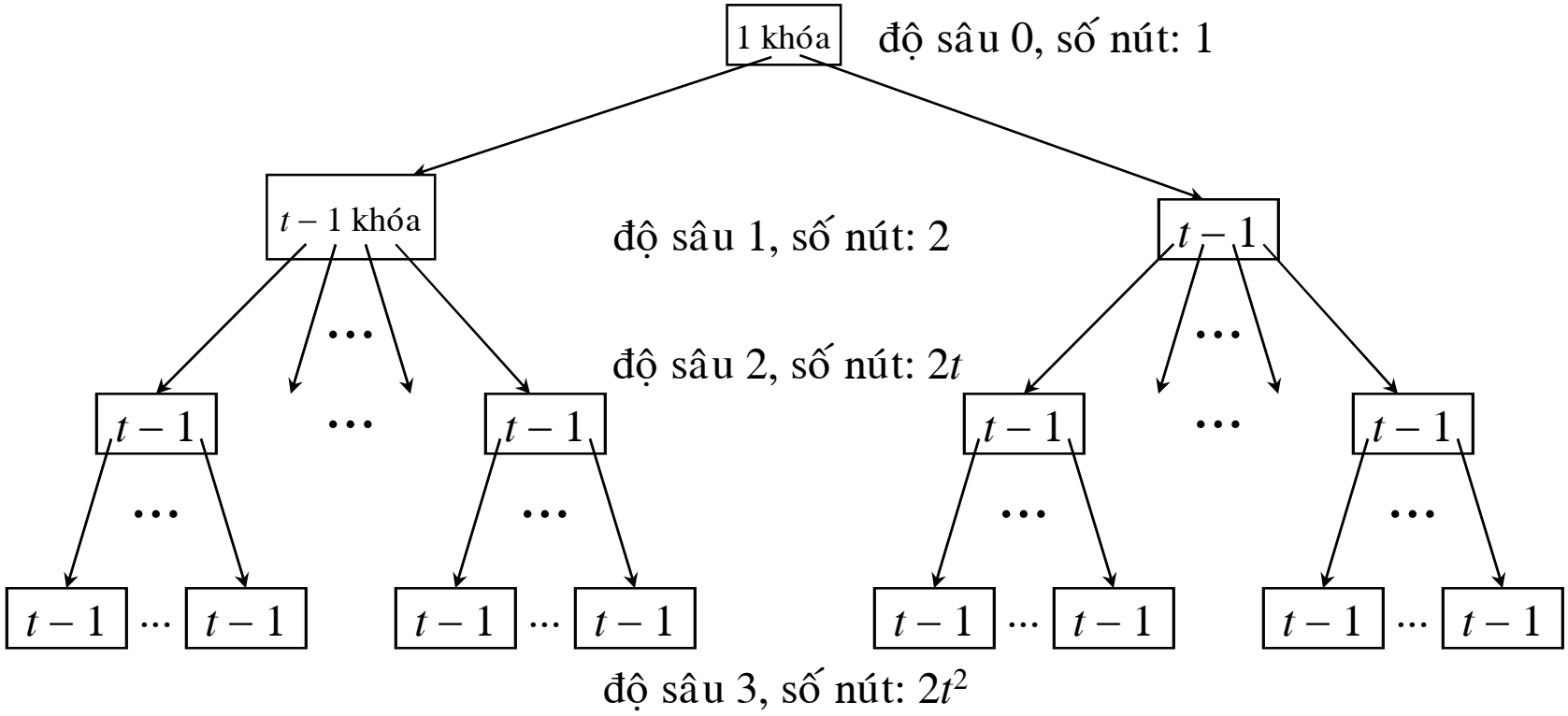
$$= 1 + 2(t-1)\frac{t^h - 1}{t-1}$$

$$= 2t^h - 1$$

Do đó $t^h \leq \frac{n+1}{2}$, từ đây suy ra định lý.

Số khóa tối thiểu trong một B-cây

- B-cây sao cho mọi nút đều có $t - 1$ khóa, ngoại trừ nút gốc chỉ có 1 khóa.
  - Vậy số khóa trong cây là tối thiểu cho mọi cây có bậc tối thiểu là t và chiều cao là h.



Các thao tác lên một B-cây

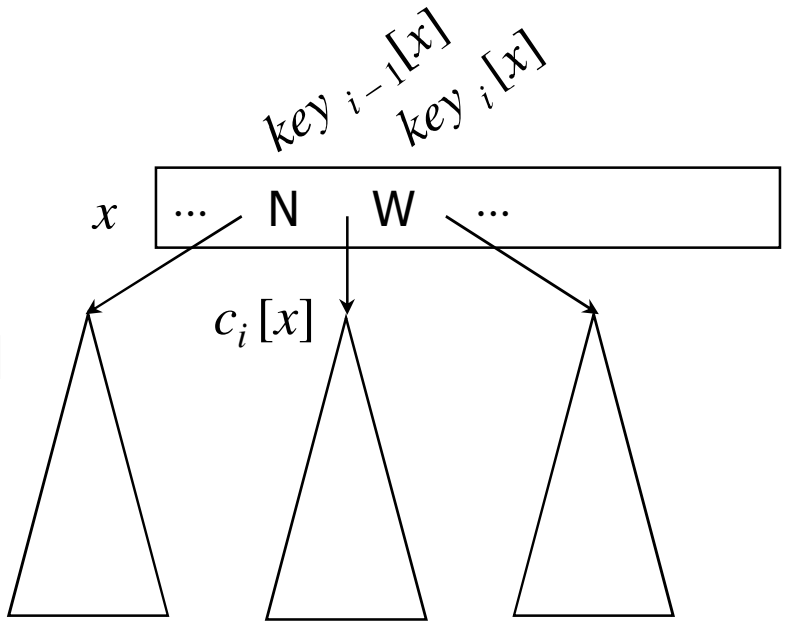
- Các thao tác lên một B-cây:
  - B-TREE-SEARCH
  - B-TREE-CREATE
  - B-TREE-INSERT
  - B-TREE-DELETE
- Trong các thủ tục trên ta quy ước:
  - Gốc của B-cây luôn luôn nằm trong bộ nhớ chính.
  - Bất kỳ một nút mà là một tham số được truyền đi trong một thủ tục thì đều đã thực thi thao tác DISK-READ lên nó.

Tìm trong một B-cây

- Thủ tục để tìm một khóa trong một B-cây
  - Input:
    - một con trỏ chỉ đến nút gốc x của một cây con, và
    - một khóa k cần tìm trong cây con.
  - Output:
    - nếu k có trong cây thì trả về một cặp (y, i) gồm một nút y và một chỉ số i mà $key_i[y] = k$
    - nếu k không có trong cây thì trả về NIL.

Tìm trong một B-cây

(tiếp)




```
B-TREE-SEARCH(x, k)
1       i ← 1
2       while i ≤ n[x] and k > key_i [x]
3             do i ← i + 1
4       if i ≤ n[x] and k = key_i [x]
5          then return (x, i)
6       if leaf [x]
7          then return NIL
8          else DISK-READ(c_i [x])
9               return B-TREE-SEARCH(c_i [x], k)
```

Tìm trong một B-cây

(tiếp)

- Các nút mà giải thuật truy cập tạo nên một đường đi từ gốc xuống đến nút có chứa khóa (nếu có).

  Thời gian CPU để xử lý mỗi nút là $O(t)$.

- Do đó
  - số disk pages mà B-TREE-SEARCH truy cập là $\Theta(h) = \Theta(\log_t n)$, với h là chiều cao của cây, n là số khoá của cây.
  - B-TREE-SEARCH cần thời gian CPU $O(t\,h) = O(t \log_t n)$.

Tạo một B-cây trống

- Thủ tục để tạo một nút gốc trống
  - Gọi thủ tục ALLOCATE-NODE để chiếm một disk page làm một nút mới.

```
B-TREE-CREATE(T)
1    x ← ALLOCATE-NODE()
2    leaf[x] ← TRUE
3    n[x] ← 0
4    DISK-WRITE(x)
5    root[T] ← x
```

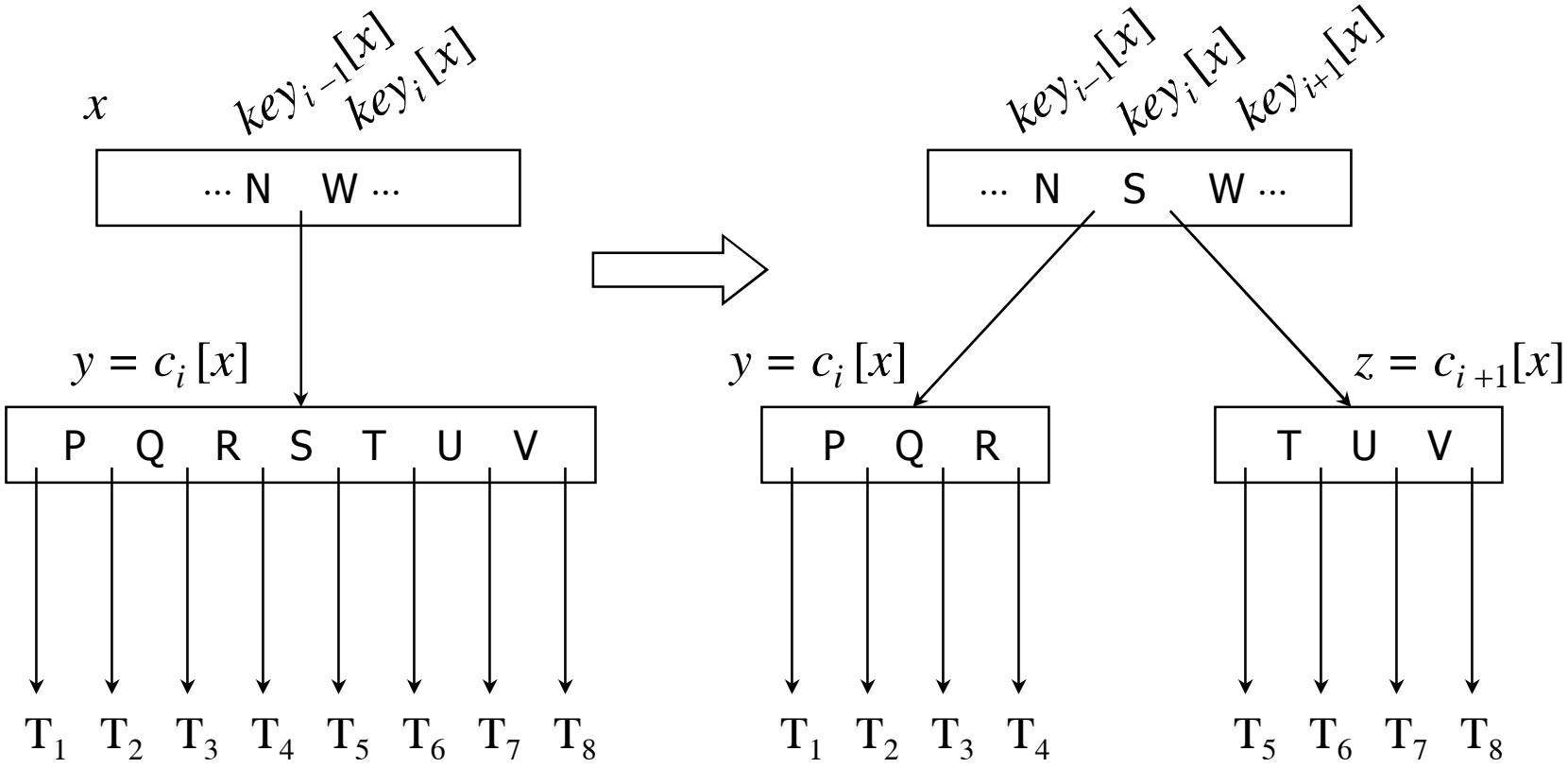
  - B-TREE-CREATE cần $O(1)$ thời gian CPU và $O(1)$ disk operations.

Chèn một khóa vào một B-cây

- Khi một nút y là đầy ($n[y] = 2t - 1$), định nghĩa *khóa giữa* (median key) của y là khóa $key_t[y]$.
- Ta sẽ chèn khóa vào một lá của cây. Để tránh trường hợp chèn khóa vào một lá đã đầy, ta cần một thao tác *tách* (split) một nút đầy y. Thao tác này
  - tách nút đầy y quanh nút giữa của nó thành hai nút, mỗi nút có $t - 1$ khóa
  - di chuyển nút giữa lên nút cha của y (phải là nút không đầy) vào một vị trí thích hợp.
- Để chèn khóa mà chỉ cần một lượt đi từ nút gốc đến một lá, ta sẽ tách mọi nút đầy mà ta gặp trên đường đi từ gốc đến nút lá.
  - Phải đảm bảo được rằng khi tách một nút đầy y thì nút cha của nó phải là không đầy.

Ví dụ tách một nút đầy

- Bậc tối thiểu $t = 4$. Vậy số khóa tối đa của một nút là 7.
- Tách nút đầy y là con của nút không đầy x.

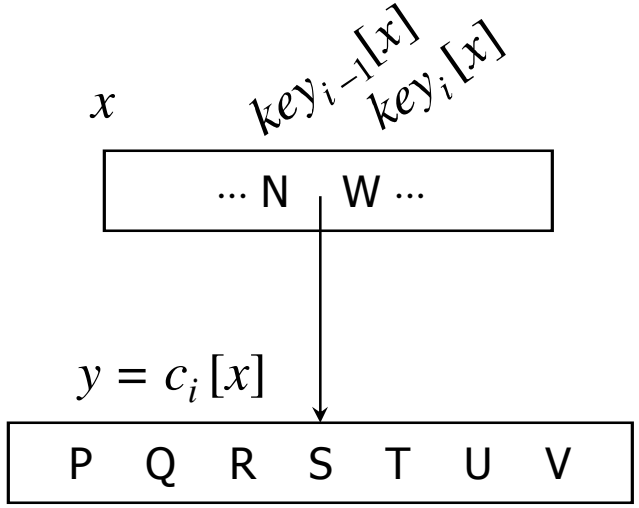


Tách một nút của một B-cây

- Thủ tục B-TREE-SPLIT-CHILD
  - Input: một nút trong <u>không đầy</u> x, một chỉ số i mà nút $y = c_i[x]$ là một nút đầy
  - Thủ tục tách y thành hai nút và chỉnh x để cho x có thêm một nút con.

B-TREE-SPLIT-CHILD(x, i, y)

```
1   z ← ALLOCATE-NODE()
2   leaf[z] ← leaf[y]
3   n[z] ← t − 1
4   for j ← 1 to t − 1
5       do key_j[z] ← key_{j+t}[y]
6   if not leaf[y]
7      then for j ← 1 to t
8               do c_j[z] ← c_{j+t}[y]
9   n[y] ← t − 1
```



Tách một nút của một B-cây

(tiếp)

```
10      for j ← n[x] + 1 downto i + 1
11          do c_{j+1}[x] ← c_j[x]
12      c_{i+1}[x] ← z
13      for j ← n[x] downto i
14          do key_{j+1}[x] ← key_j[x]
15      key_i[x] ← key_t[y]
16      n[x] ← n[x] + 1
17      DISK-WRITE(y)
18      DISK-WRITE(z)
19      DISK-WRITE(x)
```

Tách một nút của một B-cây

(tiếp)

- B-TREE-SPLIT-CHILD cần
  - $\Theta(t)$ thời gian CPU (các dòng 4-5 và 7-8)
  - $O(1)$ disk operations (các dòng 17-19).

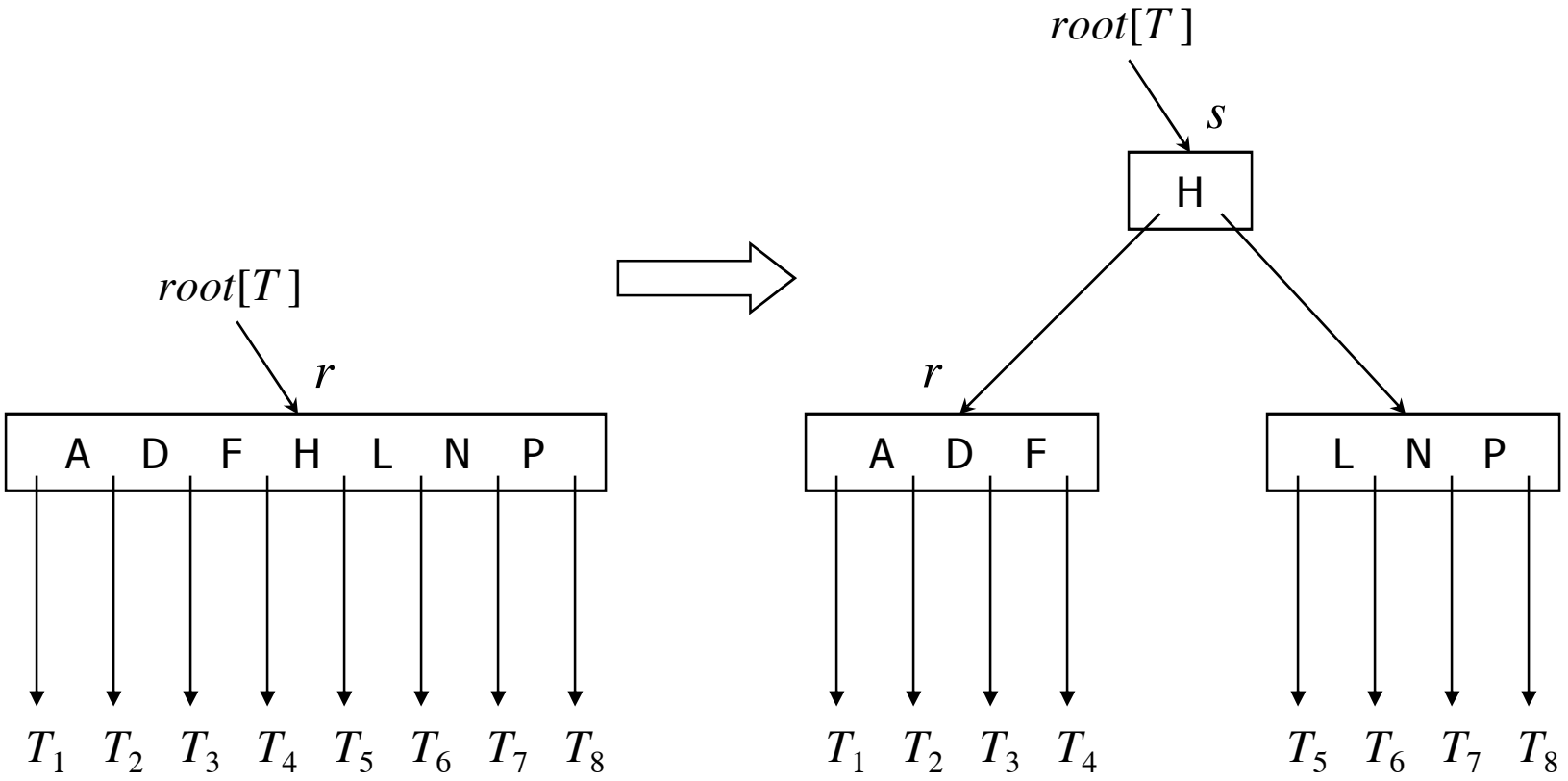
Chèn một khóa vào trong một B-cây

- Thủ tục B-TREE-INSERT để chèn một khóa k vào một B-cây T.
  - Thủ tục gọi B-TREE-SPLIT-CHILD để đảm bảo khi gọi đệ quy thì sẽ không bao giờ xuống một nút đã đầy.

```
B-TREE-INSERT(T, k)
 1   r ← root[T]
 2   if n[r] = 2t − 1
 3      then s ← ALLOCATE-NODE()
 4           root[T] ← s
 5           leaf[s] ← FALSE
 6           n[s] ← 0
 7           c1[s] ← r
 8           B-TREE-SPLIT-CHILD(s, 1, r)
 9           B-TREE-INSERT-NONFULL(s, k)
10      else B-TREE-INSERT-NONFULL(r, k)
```

Tách một nút gốc đầy

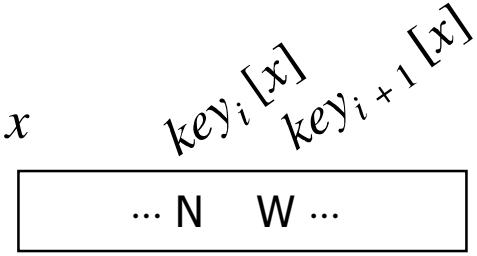
- Ví dụ: tách một nút gốc đầy của một B-cây mà bậc tối thiểu là $t = 4$.
- Nút gốc mới là s. Nút gốc cũ r được tách thành hai nút con của s.
- Chiều cao của một B-cây tăng thêm 1 mỗi khi nút gốc được tách.



Chèn một khóa vào một nút không đầy

- Thủ tục để chèn một khóa vào một nút không đầy

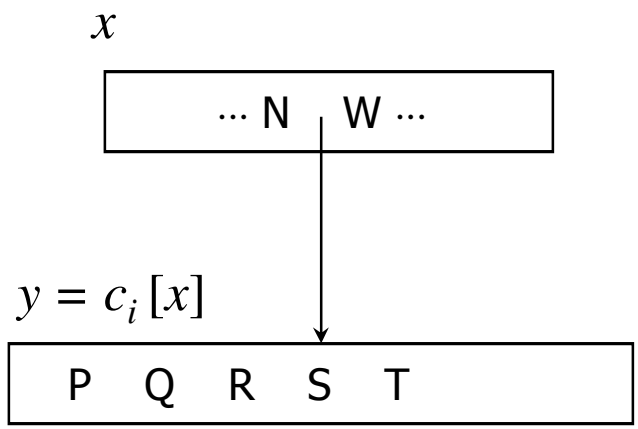
```
B-TREE-INSERT-NONFULL(x, k)
1       i ← n[x]
2       if leaf [x]
3          then while i ≥ 1 and k < key_i [x]
4                    do key_{i+1}[x] ← key_i [x]
5                       i ← i – 1
6                  key_{i+1}[x] ← k
7                  n[x] ← n[x] + 1
8                  DISK-WRITE(x)
```



nút lá

Chèn một khóa vào một nút không đầy

(tiếp)



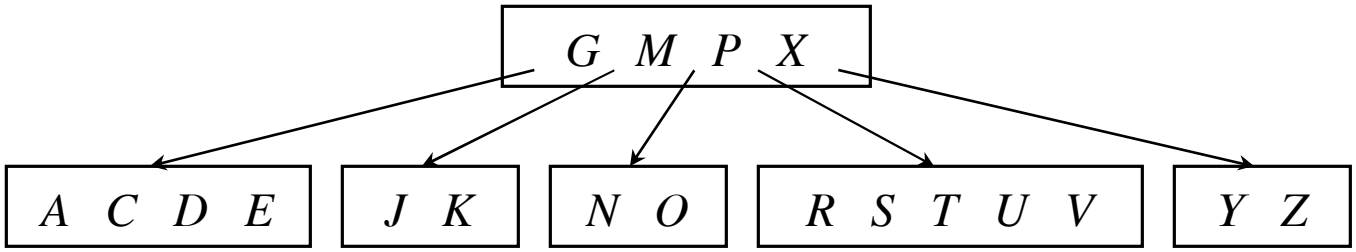

```
 9        else while i ≥ 1 and k < key_i [x]
10                do i ← i − 1
11             i ← i + 1
12             DISK-READ(c_i [x])
13             if n[c_i [x]] = 2t − 1
14                then B-TREE-SPLIT-CHILD(x, i, c_i [x])
15                     if k > key_i [x]
16                        then i ← i + 1
17             B-TREE-INSERT-NONFULL(c_i [x], k)
```

Phân tích chèn một khóa vào trong một B-cây

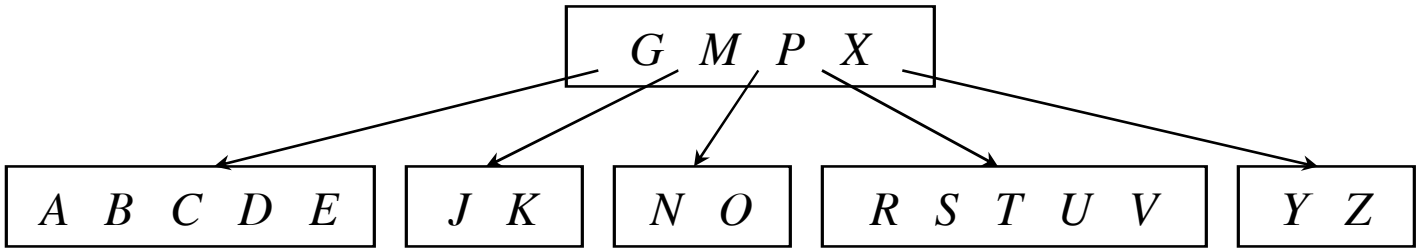
- Thủ tục B-TREE-INSERT cần
  - số truy cập đĩa là $O(h)$ vì số lần gọi DISK-READ và DISK-WRITE giữa các gọi B-TREE-INSERT-NONFULL là $O(1)$.
  - thời gian CPU là $O(t\,h) = \mathrm{O}(t \log_t n)$

Ví dụ cho các trường hợp khi chèn một khóa vào một B-cây

- Cho một B-cây với bậc tối thiểu $t = 3$
- Cây lúc đầu



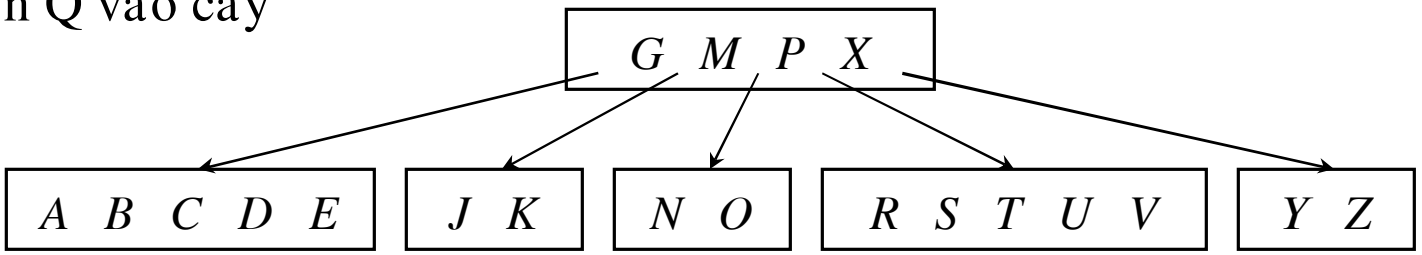
- Đã chèn *B* vào

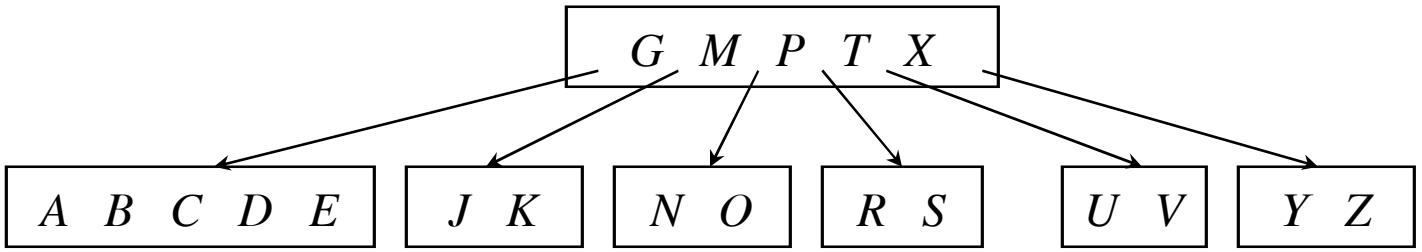


Ví dụ cho các trường hợp khi chèn một khóa vào một B-cây

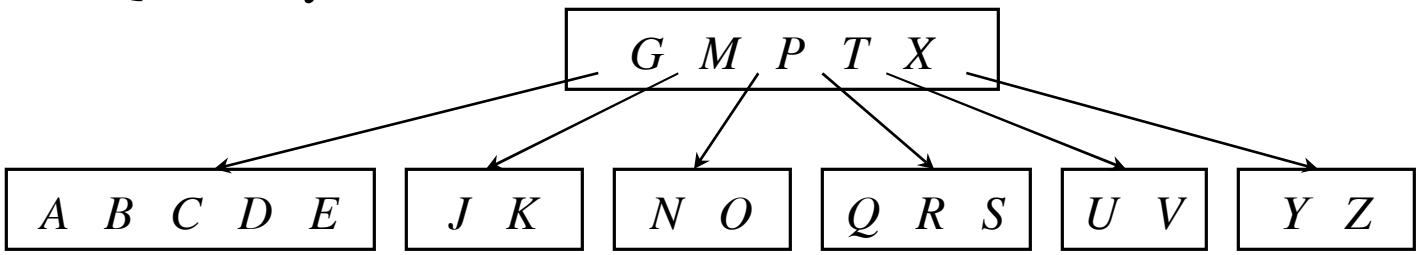
(tiếp)

Chèn Q vào cây





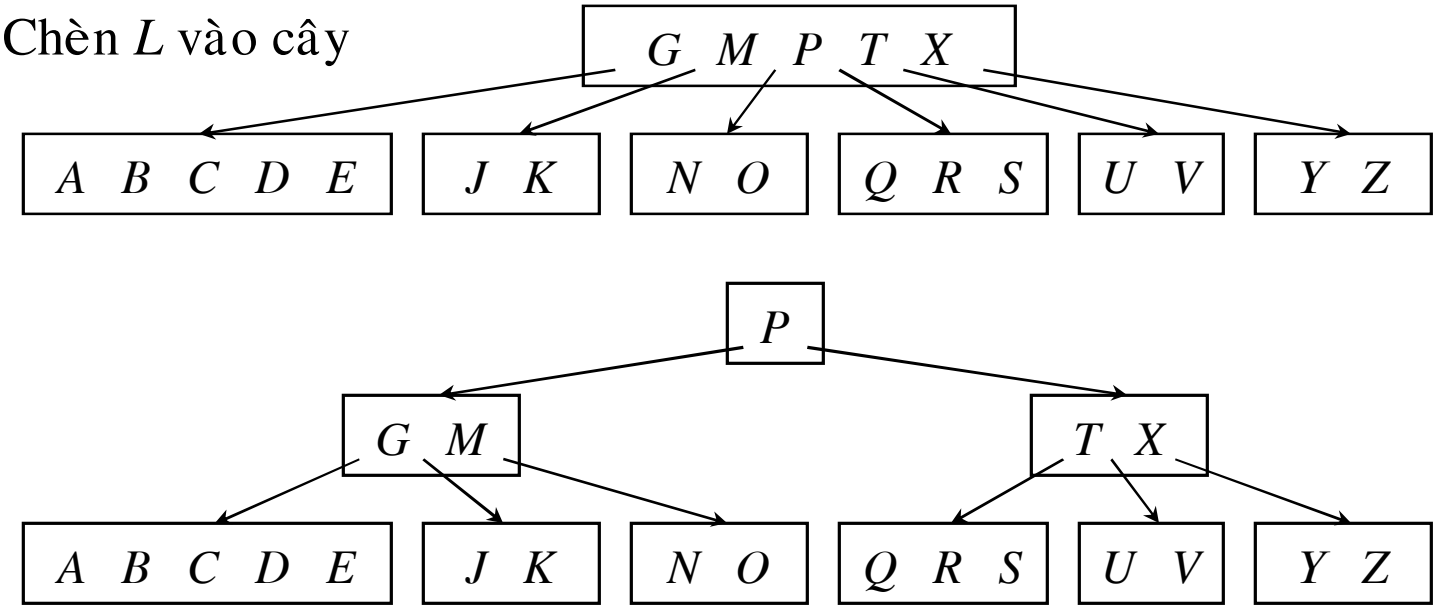
Đã chèn Q vào cây:



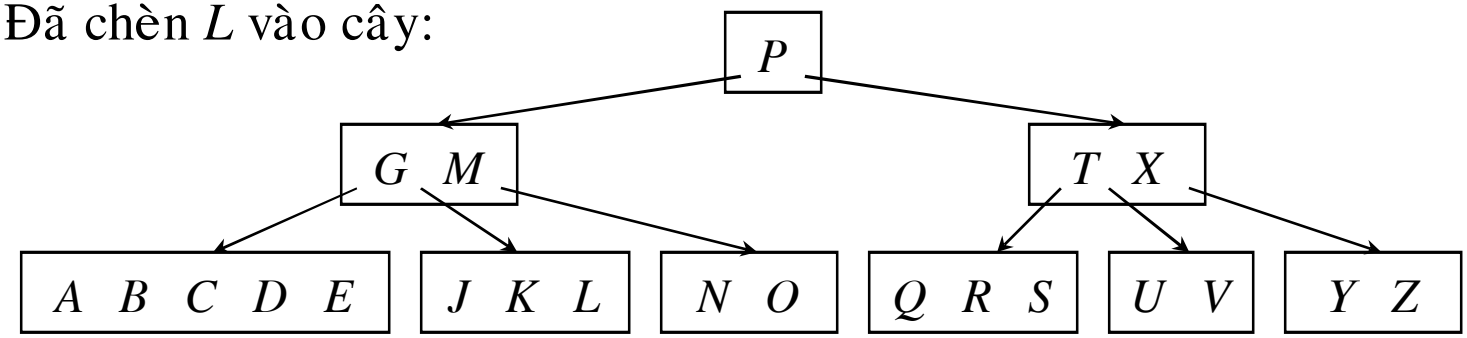
Ví dụ cho các trường hợp khi chèn một khóa vào một B-cây

(tiếp)

- Chèn *L* vào cây



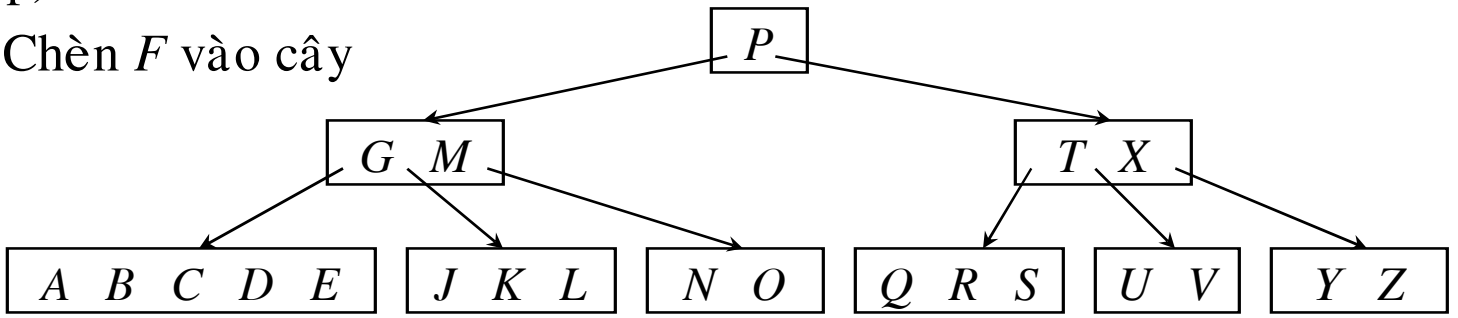
- Đã chèn *L* vào cây:

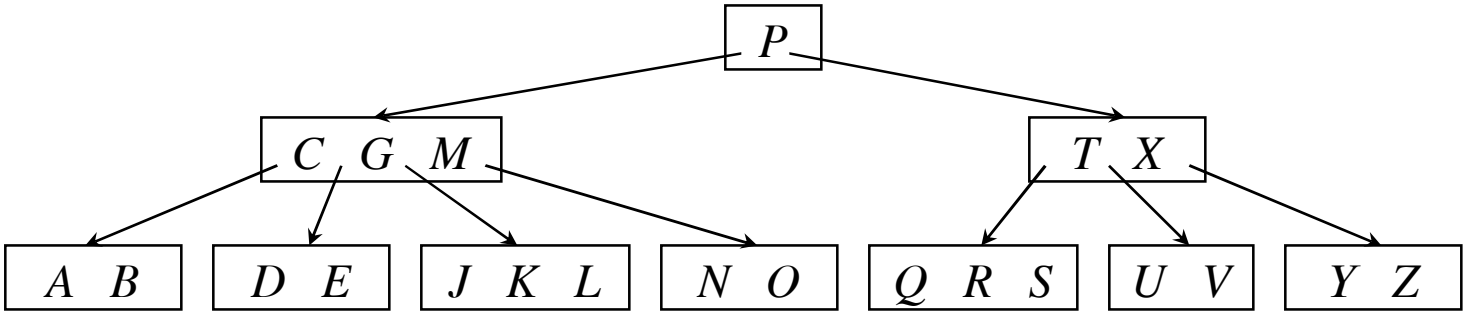


Ví dụ cho các trường hợp khi chèn một khóa vào một B-cây

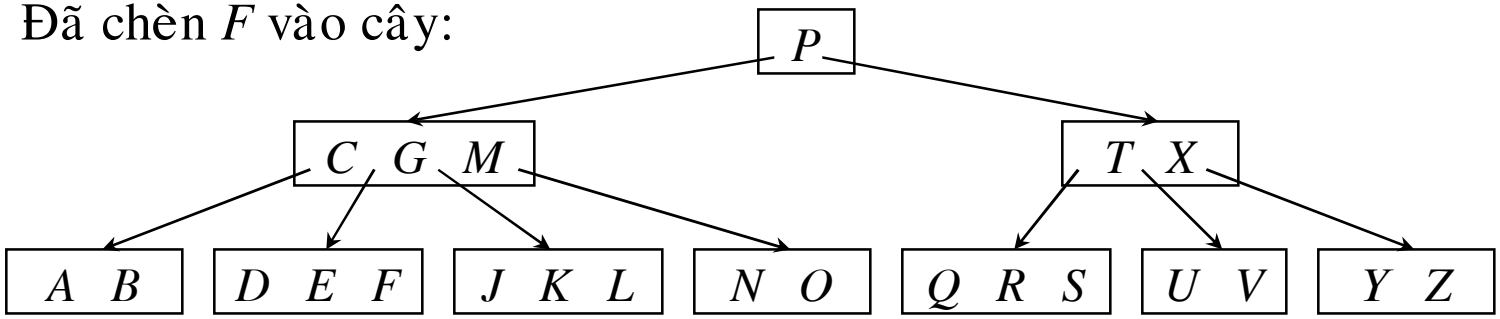
(tiếp)

- Chèn *F* vào cây





- Đã chèn *F* vào cây:



Xóa một khóa khỏi một B-cây

Thủ tục B-TREE-DELETE(x, k) để xóa khóa k khỏi cây con có gốc tại x bảo đảm rằng khi B-TREE-DELETE được gọi đệ quy lên x thì

- số khóa trong x phải $\geq t$ (bậc tối thiểu của cây).

Do đó đôi khi một khóa được di chuyển (từ một nút thích hợp khác) vào một nút trước khi đệ quy xuống nút đó.

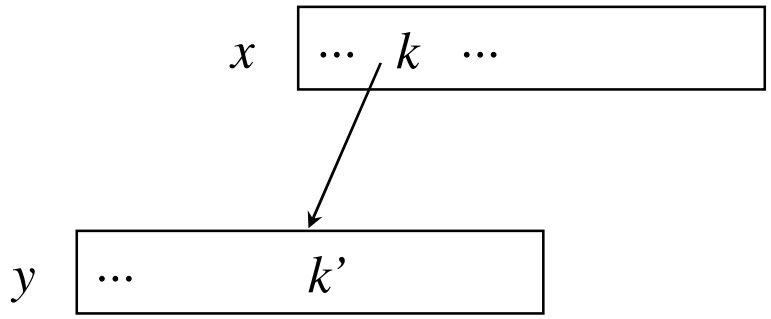
Xóa một khóa khỏi một B-cây

B-TREE-DELETE(x, k)

1. Nếu khóa k có trong nút x và x là một nút lá thì xóa k khỏi x.

2. Nếu khóa k có trong nút x và x là một nút trong thì

a. Nếu nút con y ở trước k có ít nhất t khóa thì tìm khóa trước (predecessor) k' của k trong cây con có gốc tại y. Xóa k' bằng cách gọi đệ quy B-TREE-DELETE(y, k'), thay k bằng k' trong x.



Xóa một khóa khỏi một B-cây

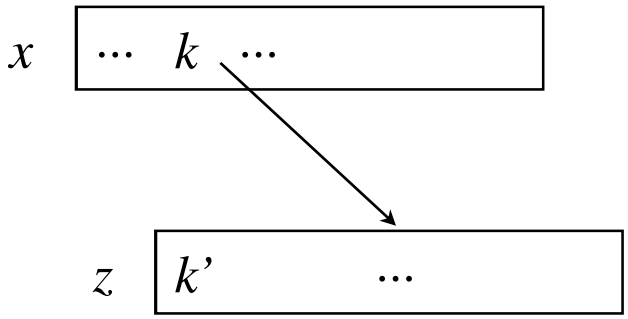
B-TREE-DELETE(x, k)

1. ...

2. Nếu khóa k có trong nút x và x là một nút trong thì

a. ...

b. Tương tự, nếu nút con z ở sau k có ít nhất t khóa thì tìm khóa sau (successor) k' của k trong cây con có gốc tại z. Xóa k' bằng cách gọi đệ quy B-TREE-DELETE(z , k'), thay k bằng k' trong x.



Xóa một khóa khỏi một B-cây

B-TREE-DELETE(x, k)

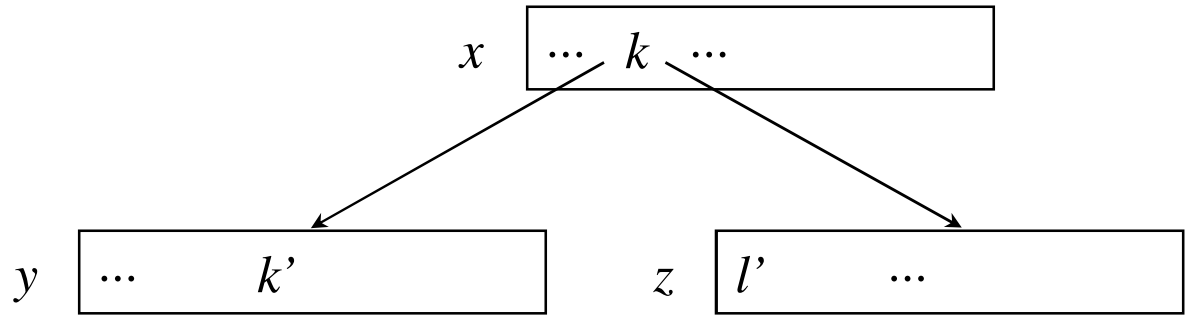
1. ...

2. Nếu khóa k có trong nút x và x là một nút trong thì

a. ...

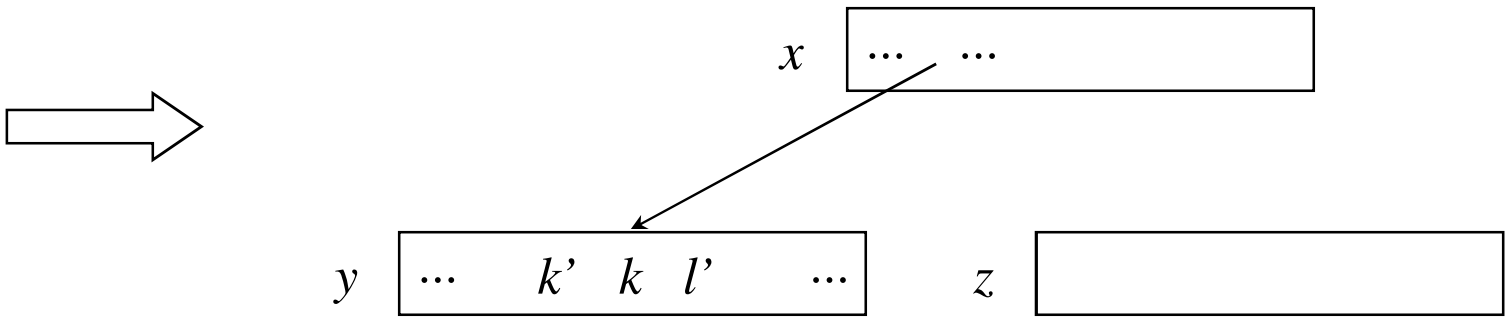
b. ...

c. Nếu không, cả y lẩn z đều chỉ có $t - 1$ khóa, hợp nhất k và nguyên cả z vào y, thành ra x mất k và con trỏ đến z , và bây giờ y chứa $2t - 1$ khóa. Giải phóng (free) z và gọi đệ quy B-TREE-DELETE(y, k) để xóa k khỏi cây có gốc y.



Xóa một khóa khỏi một B-cây

(tiếp)



Xóa một khóa khỏi một B-cây

B-TREE-DELETE(x, k)

1. ...

2. ...

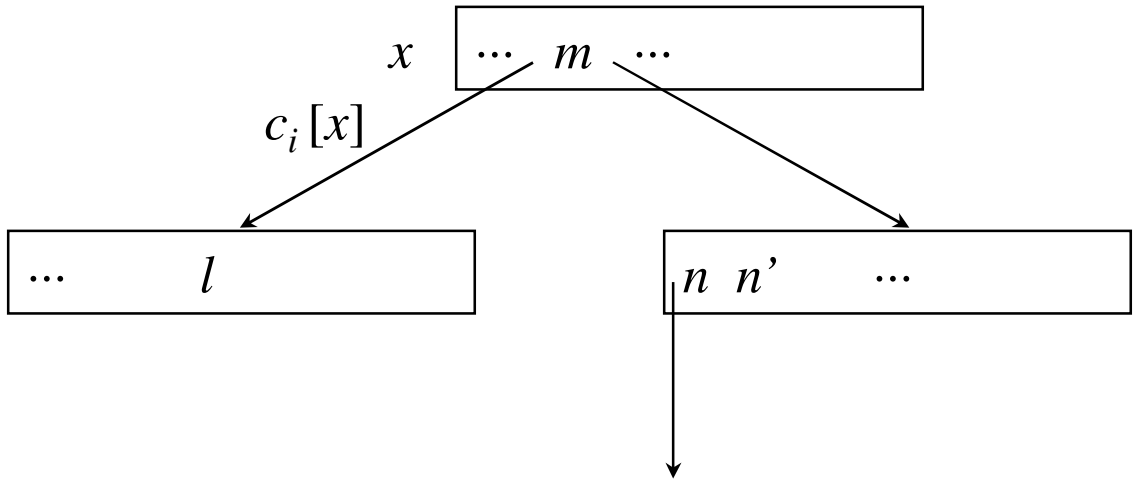
3. Nếu khóa k không có trong nút trong x thì xác định gốc $c_i[x]$ của cây con chứa k, nếu k có trong cây. Nếu $c_i[x]$ chỉ có $t - 1$ khóa, thực thi bước 3a hay 3b nếu cần để đảm bảo rằng ta sẽ xuống đến một nút chứa ít nhất t khóa. Xong rồi gọi B-TREE-DELETE lên nút con thích hợp của x.

Xóa một khóa khỏi một B-cây

(tiếp)

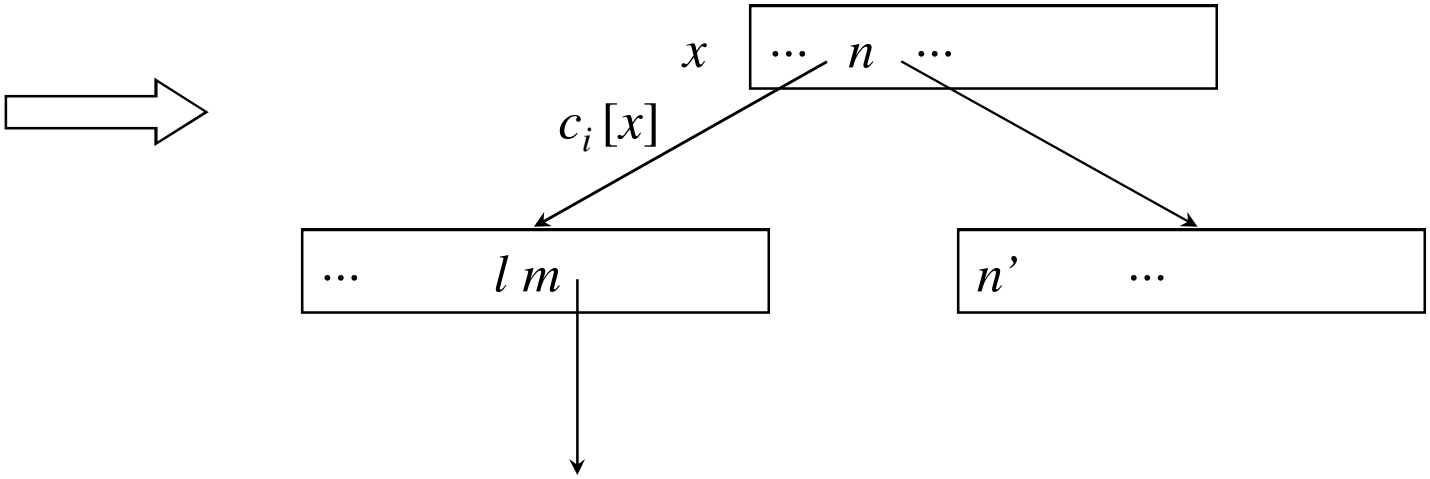
3. ...

a. Nếu $c_i[x]$ chỉ có $t - 1$ khóa, nhưng lại có một nút anh em với ít nhất t khóa, thì cho $c_i[x]$ thêm một khóa bằng cách đem một khóa từ x xuống $c_i[x]$, đem một khóa từ nút anh em ngay bên trái hay ngay bên phải của $c_i[x]$ lên x, và đem con trỏ tương ứng từ nút anh em vào $c_i[x]$.



Xóa một khóa khỏi một B-cây

(tiếp)



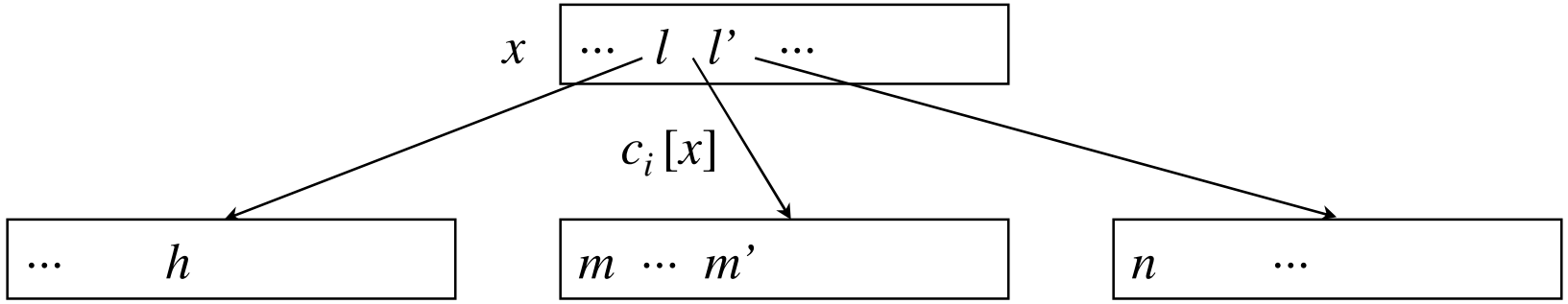
Xóa một khóa khỏi một B-cây

(tiếp)

3. ...

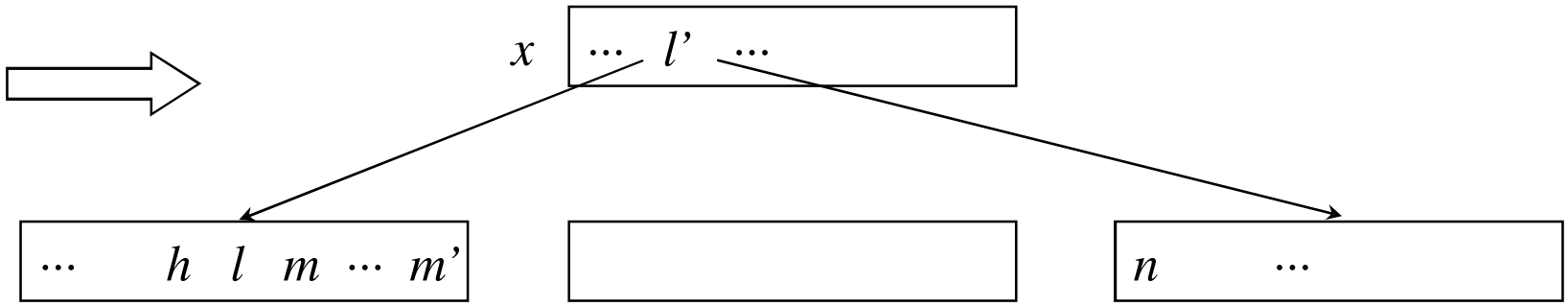
a. ...

b. Nếu $c_i[x]$ và mọi nút anh em của nó chỉ có $t - 1$ khóa, thì hợp nhất $c_i[x]$ và một nút anh em bằng cách đem một khóa từ x xuống nút mới tạo, khóa này sẽ là khóa giữa của nút.



Xóa một khóa khỏi một B-cây

(tiếp)



Xóa một khóa khỏi một B-cây

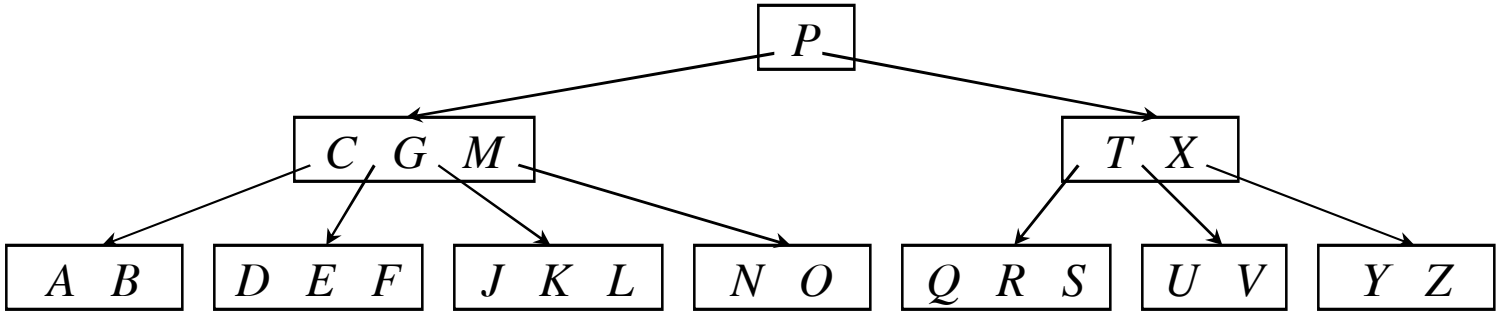
(tiếp)

Thủ tục B-TREE-DELETE cần

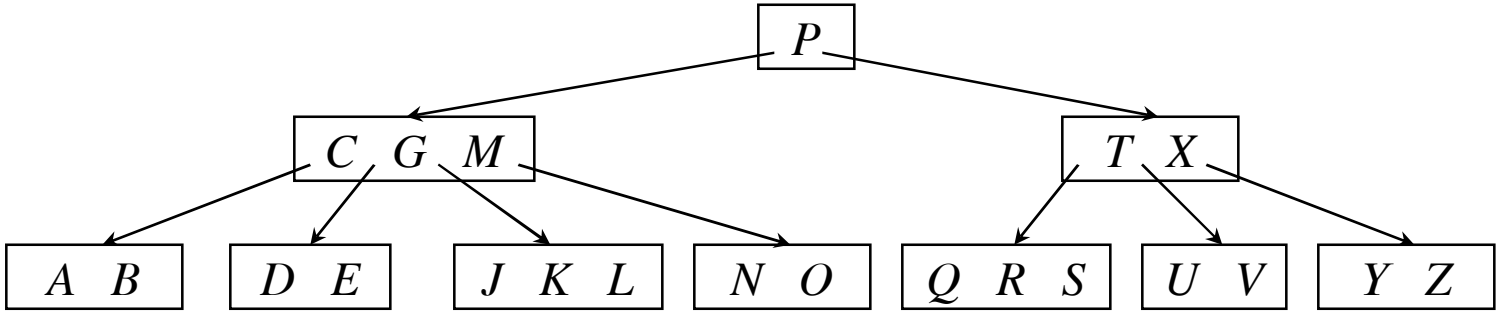
- số truy cập lên đĩa là $O(h)$ vì có $O(1)$ lần gọi DISK-READ và DISK-WRITE giữa các gọi đệ quy của thủ tục.
- thời gian CPU của thủ tục là $O(t\,h) = O(t \log_t n)$.

Ví dụ cho các trường hợp khi xóa một khóa khỏi một B-cây

- Cho một B-cây có bậc tối thiểu $t = 3$
- Cây lúc đầu, xóa F khỏi cây

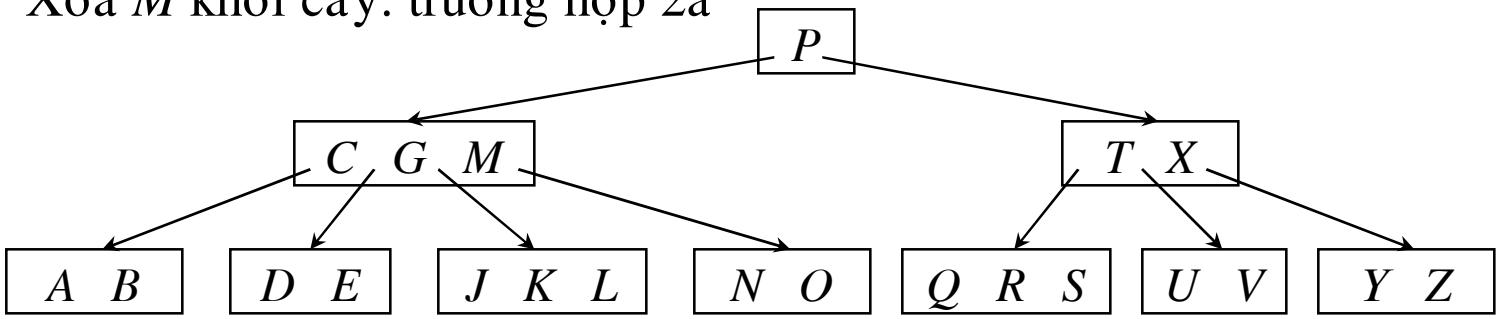


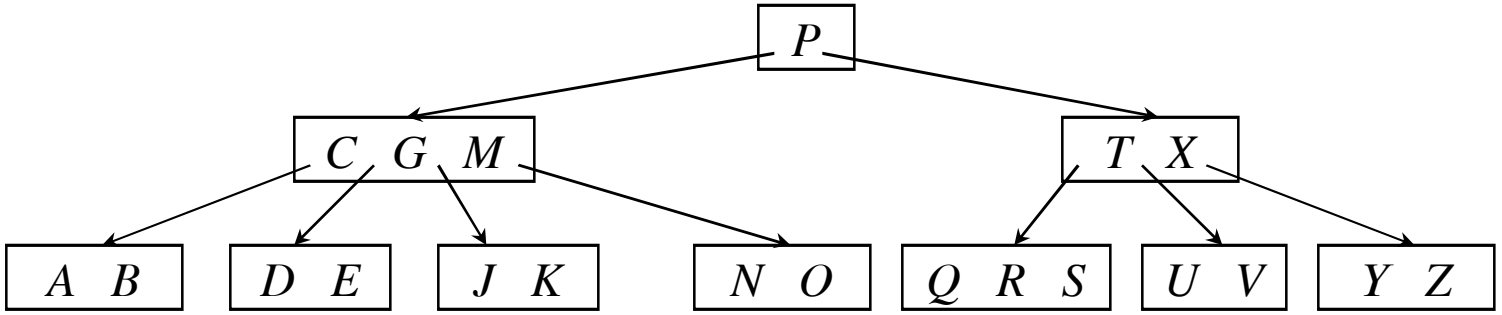
- F đã được xóa: trường hợp 1



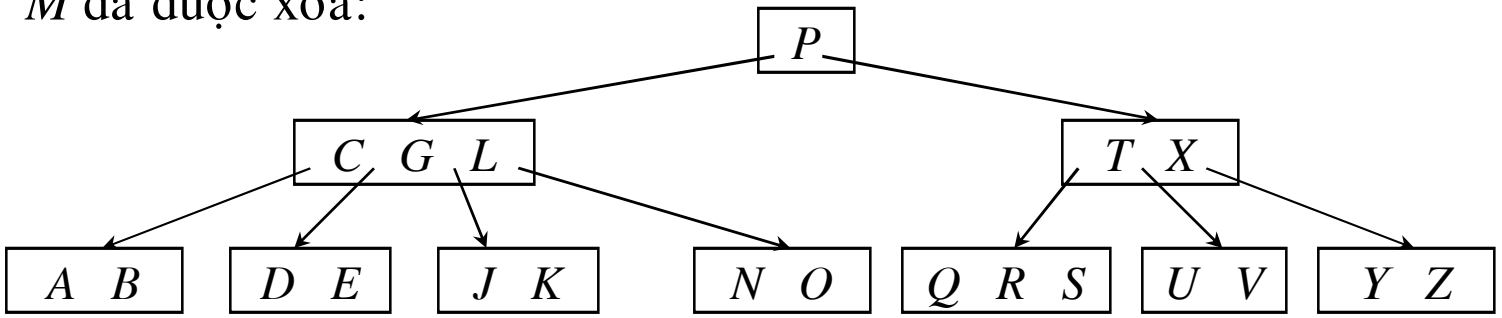
Ví dụ cho các trường hợp khi xóa một khóa khỏi một B-cây

- Xóa *M* khỏi cây: trường hợp 2a



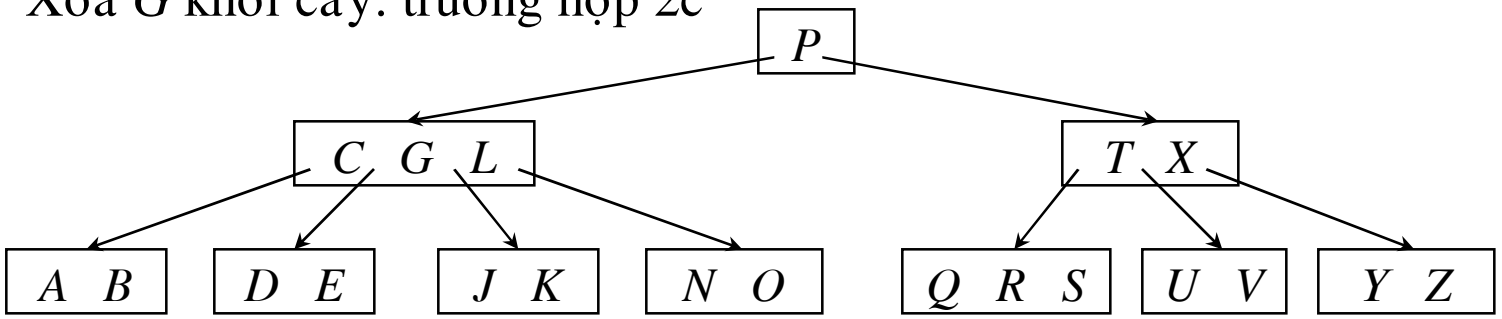


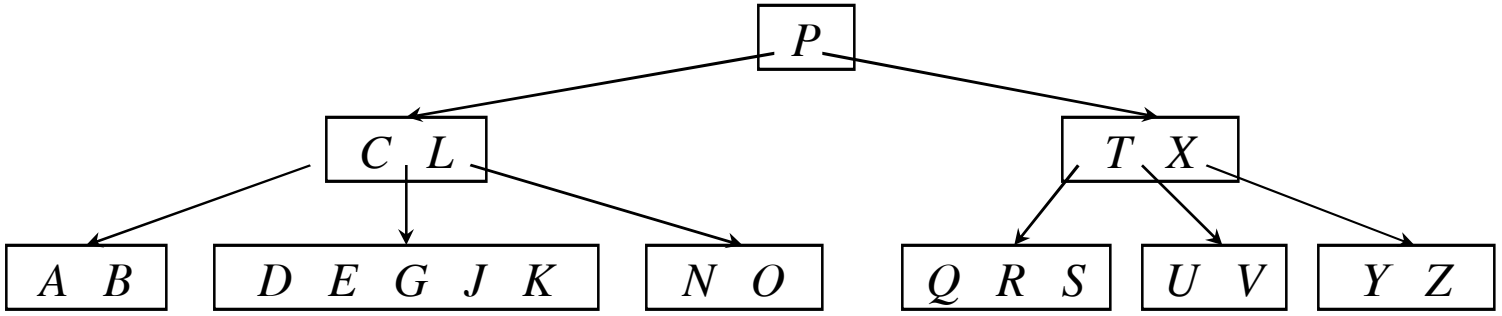
- *M* đã được xóa:



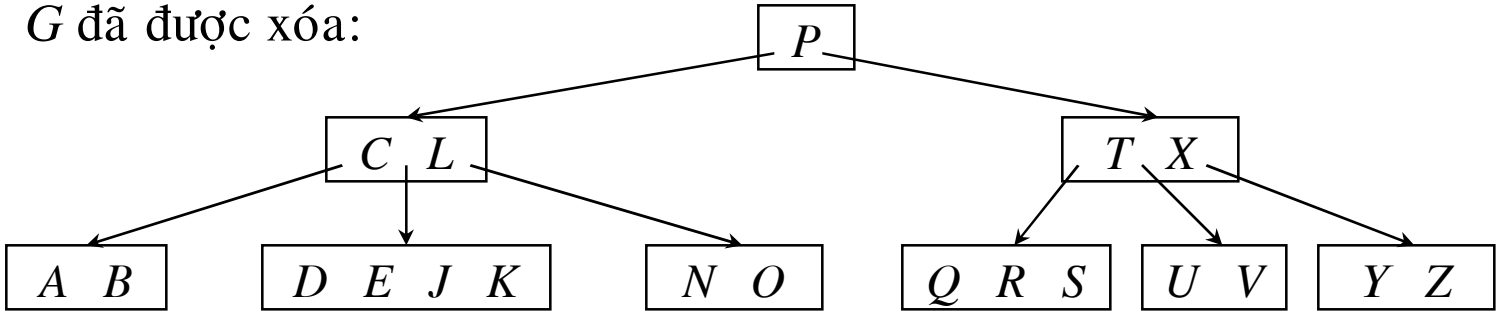
Ví dụ cho các trường hợp khi xóa một khóa khỏi một B-cây

- Xóa *G* khỏi cây: trường hợp 2c



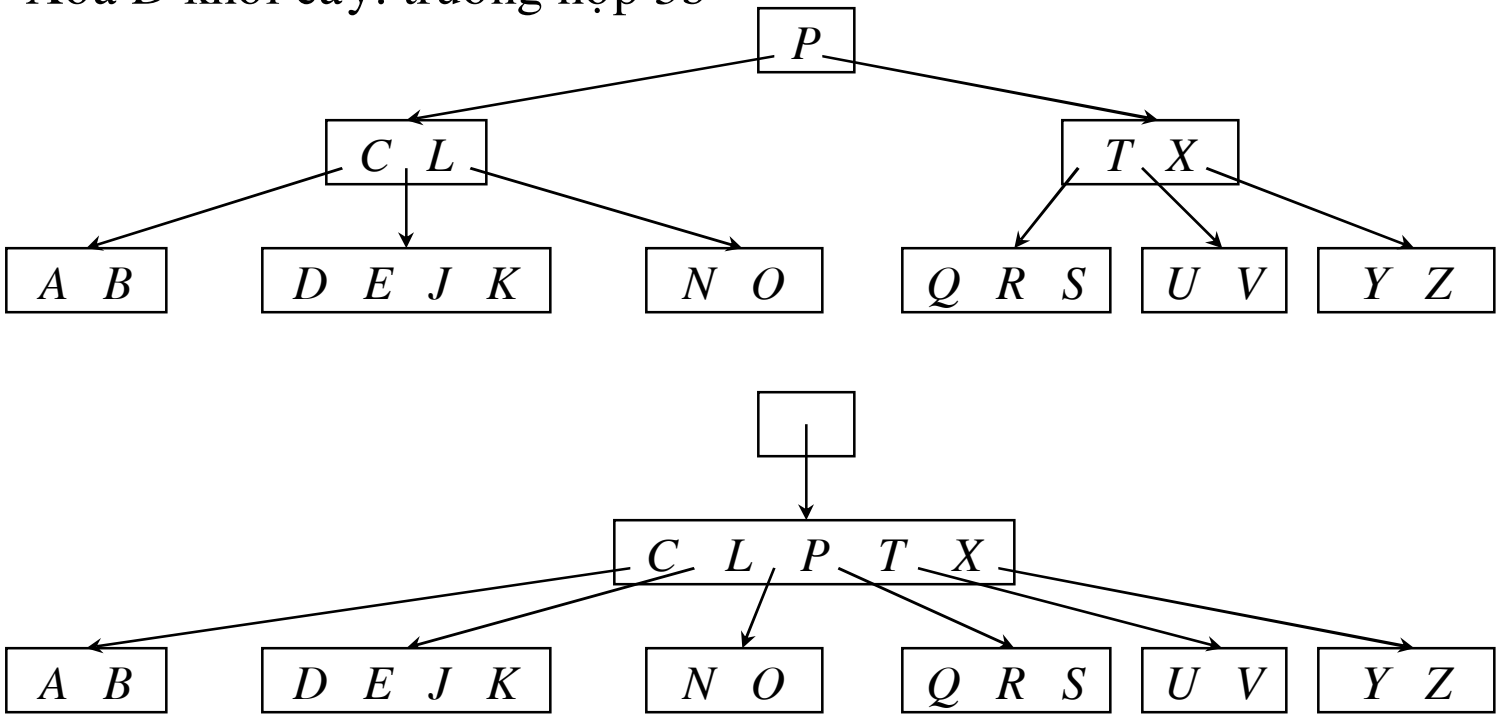


- *G* đã được xóa:

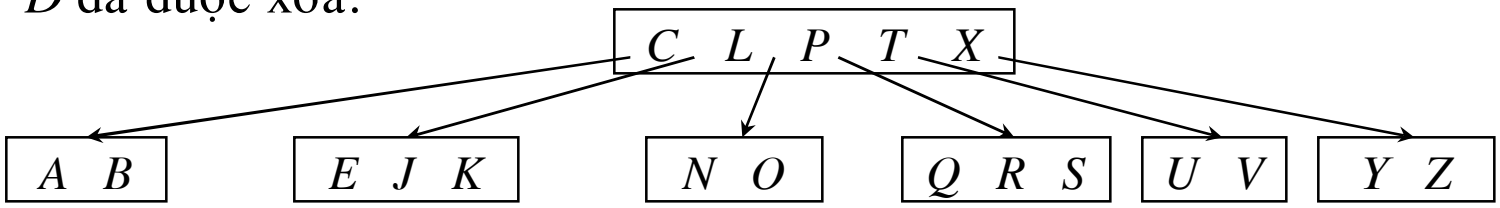


Ví dụ cho các trường hợp khi xóa một khóa khỏi một B-cây

- Xóa *D* khỏi cây: trường hợp 3b

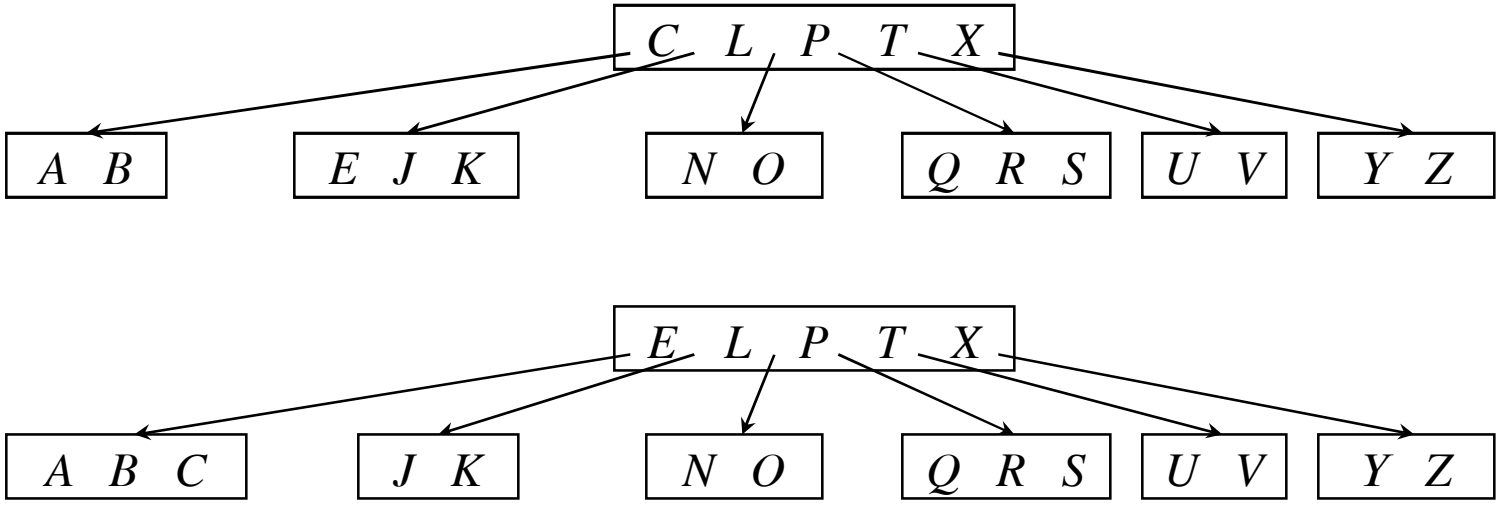


- *D* đã được xóa:

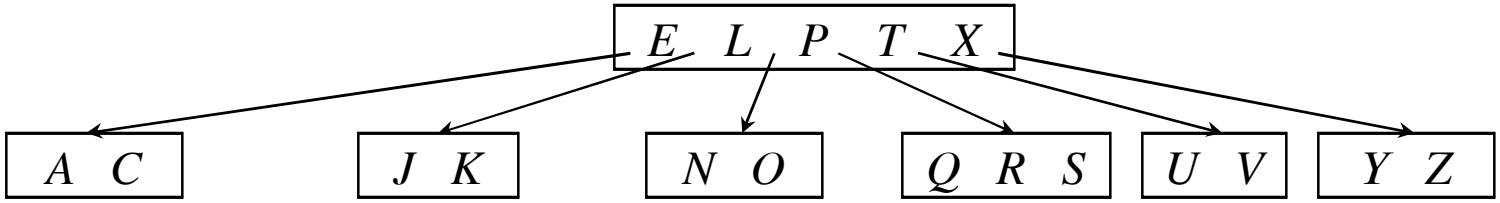


Ví dụ cho các trường hợp khi xóa một khóa khỏi một B-cây

- Xóa *B* khỏi cây: trường hợp 3a



- *B* đã được xóa khỏi cây:



Các heap hợp nhất được

- Heap nhị phân
- Một *heap hợp nhất được* (mergeable heap) là một cấu trúc dữ liệu hỗ trợ năm thao tác sau (gọi là các *thao tác heap hợp nhất được*).
  - MAKE-HEAP() tạo và trả về một heap trống.
  - INSERT(H, x) chèn nút x, mà trường *key* của nó đã được điền, vào heap H .
  - MINIMUM(H) trả về một con trỏ chỉ đến nút trong heap H mà khóa của nó là nhỏ nhất.
  - EXTRACT-MIN(H) tách ra nút có khóa nhỏ nhất khỏi H, và trả về một con trỏ chỉ đến nút đó.
  - UNION(H_1, H_2) tạo và trả về một heap mới chứa tất cả các nút của các heaps H_1 và H_2 . Các heaps H_1 và H_2 sẽ bị hủy bởi thao tác này.

Thời gian chạy của các thao tác lên heaps hợp nhất được

- n là số nút của heap

Thủ tục	heap nhị phân (worst-case)	heap nhị thức (worst-case)	heap Fibonacci (khấu hao)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Heap nhị thức

- Heap nhị thức

  Hỗ trợ thêm các thao tác

  – DECREASE-KEY(H, x, k) gán vào nút x trong heap H trị mới k của khóa, k nhỏ hơn hay bằng trị hiện thời của khóa.

  – DELETE(H, x) xóa nút x khỏi heap H.

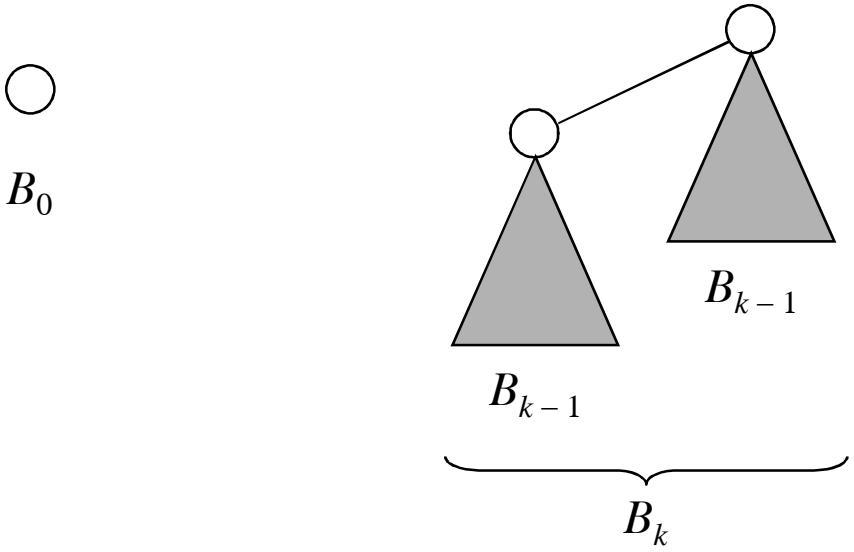
- Nhận xét:

  – Heap nhị thức không hổ trợ thao tác SEARCH hữu hiệu.

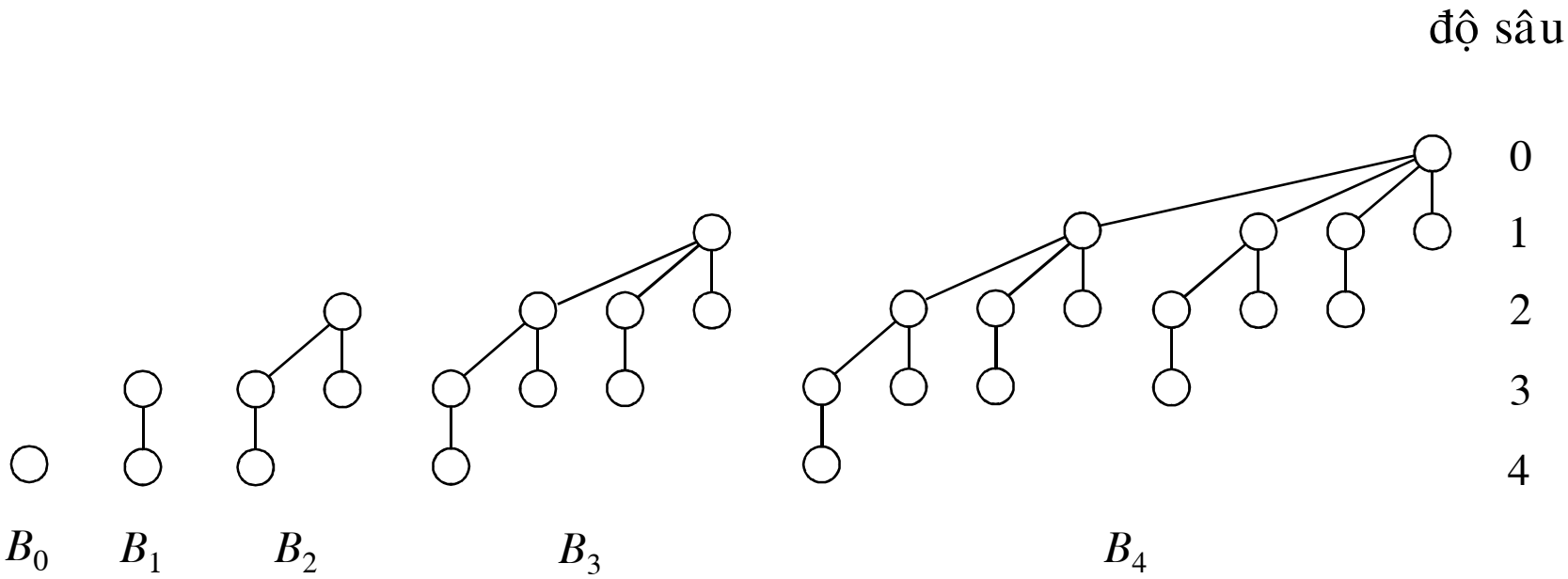
  – Do đó, các thao tác DECREASE-KEY và DELETE cần một con trỏ đến nút cần được xử lý.

Định nghĩa cây nhị thức

- *Cây nhị thức* B_k với $k = 0, 1, 2,\ldots$ là một cây có thứ tự được định nghĩa đệ quy:
  - Cây nhị thức B_0 gồm một nút duy nhất.
  - Cây nhị thức B_k gồm hai cây nhị thức B_{k-1} được *liên kết* với nhau theo một cách nhất định:
    - Nút gốc của cây này là con bên trái nhất của nút gốc của cây kia.



Định nghĩa cây nhị thức



Đặc tính của cây nhị thức

- **Lemma (Đặc tính của một cây nhị thức)**

  Cây nhị thức B_k có các tính chất sau:

  1. có 2^k nút,

  2. chiều cao của cây là k,

  3. có đúng $\binom{k}{i}$ nút tại độ sâu i với $i = 0, 1, \ldots, k$

  4. bậc của nút gốc của cây là k, nó lớn hơn bậc của mọi nút khác; ngoài ra nếu các con của nút gốc được đánh số từ trái sang phải bằng $k-1, k-2, \ldots, 0$, thì nút con i là gốc của cây con B_i .

$$\binom{k}{i} = \frac{k!}{i!(k-i)!}$$

Đặc tính của cây nhị thức

Chứng minh

Dùng quy nạp theo k.

Bước cơ bản: dễ dàng thấy các tính chất là đúng cho B_0

Bước quy nạp: giả sử lemma là đúng cho B_{k-1} .

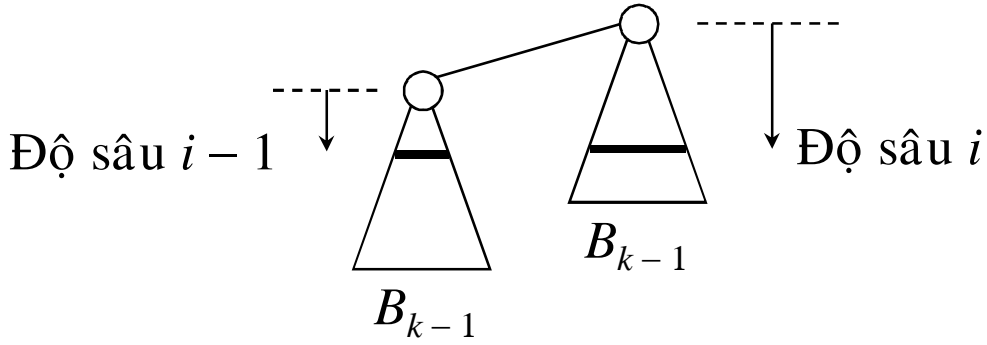
1. Cây nhị thức B_k gồm hai B_{k-1} nên B_k có $2^{k-1} + 2^{k-1} = 2^k$ nút.

2. Do cách liên kết hai cây nhị thức B_{k-1} với nhau để tạo nên B_k nên độ sâu tối đa của nút trong B_k bằng độ sâu tối đa của nút trong B_{k-1} cộng thêm 1, tức là: $(k-1)+1=k$.

Đặc tính của cây nhị thức

Chứng minh (tiếp)

3. Gọi $D(k, i)$ là số các nút tại độ sâu i của cây nhị thức B_k .

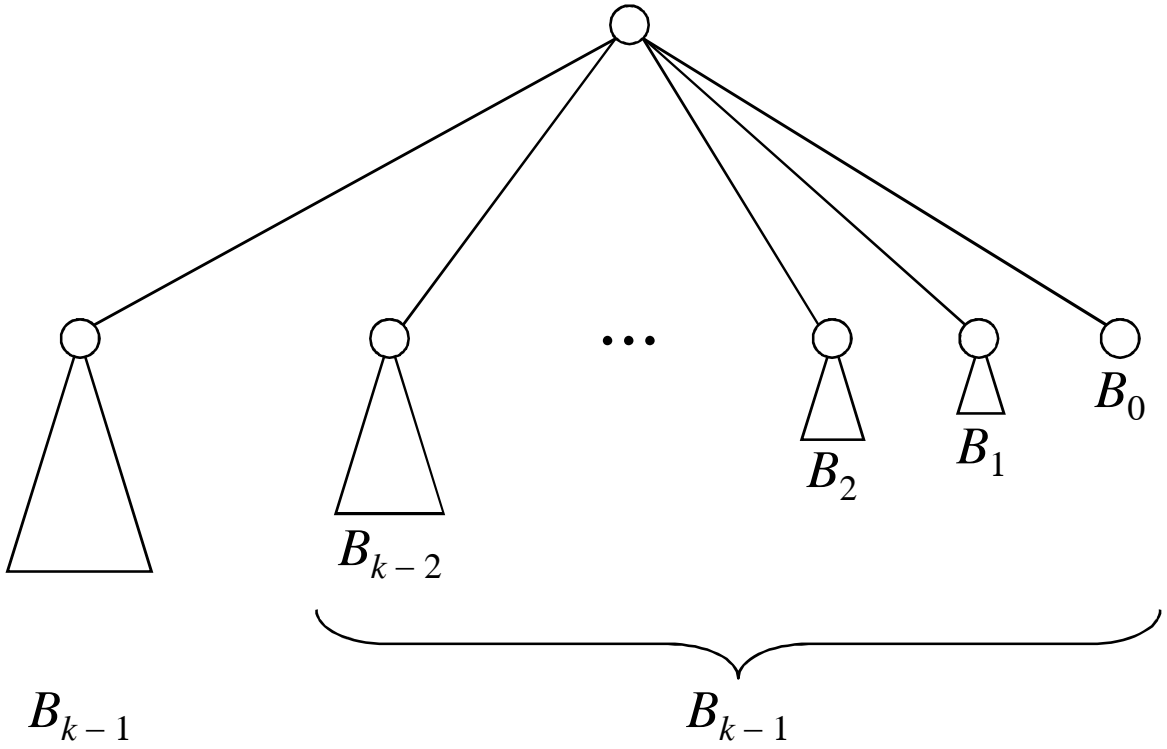


$$D(k,i) = D(k-1,i) + D(k-1,i-1)$$
$$= \binom{k-1}{i} + \binom{k-1}{i-1}$$
$$= \binom{k}{i}$$

Đặc tính của cây nhị thức

Chứng minh (tiếp)

4. Sử dụng hình sau.



Đặc tính của cây nhị thức

- **Hệ luận**

  Bậc tối đa của một nút bất kỳ trong một cây nhị thức có n nút là $\lg n$.

Định nghĩa heap nhị thức

- **Định nghĩa**

  Một *heap nhị thức H* là một tập các cây nhị thức thỏa các *tính chất heap nhị thức* sau

  1. Mọi cây nhị thức trong H là *heap-ordered*: mọi nút đều có khóa lớn hơn hay bằng khóa của nút cha của nó.
  2. Với mọi số nguyên $k \geq 0$ cho trước thì có nhiều nhất một cây nhị thức trong H mà gốc của nó có bậc là k.

Tính chất của heap nhị thức

- **Tính chất**

  1. Gốc của một cây trong một heap nhị thức chứa khóa nhỏ nhất trong cây.
  2. Một heap nhị thức H với n nút gồm nhiều lắm là $\lfloor \lg n \rfloor + 1$ cây nhị thức.

Chứng minh

1. Hiển nhiên.
2. n có biểu diễn nhị phân duy nhất, biểu diễn này cần $\lfloor \lg n \rfloor + 1$ bits, có dạng $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, ..., b_0 \rangle$ sao cho

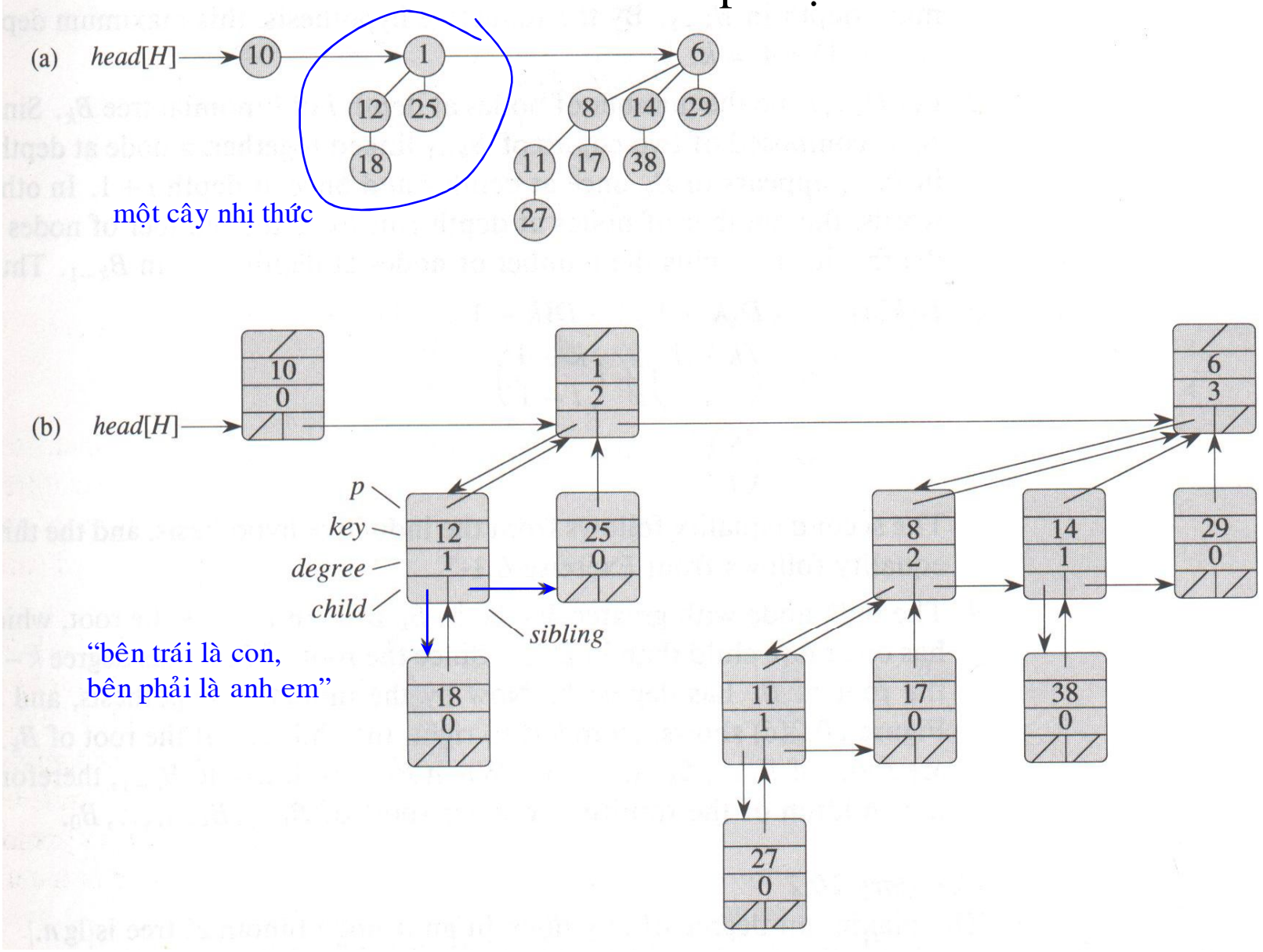
$$n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$$

$10 =$

3	2	1	0
1	0	1	0

Cùng với định nghĩa 2, ta thấy cây nhị thức B_i xuất hiện trong H nếu và chỉ nếu $b_i = 1$.

Biểu diễn heap nhị thức



Biểu diễn heap nhị thức

Qui tắc trữ cho mỗi cây nhị thức trong một heap nhị thức:

- biểu diễn theo kiểu "*Bên trái là con, bên phải là anh em*" (left-child, right-sibling representation)

- Mỗi nút x có một trường sau:
  - $key[x]$: trữ khóa của nút.
- Mỗi nút x có các con trỏ sau:
  - $p[x]$: trữ con trỏ đến nút cha của x.
  - $child[x]$: con trỏ đến con bên trái nhất của x.
    - Nếu x không có con thì $child[x]$ = NIL
  - $sibling[x]$: con trỏ đến anh em của x ở ngay bên phải x.
    - Nếu x là con bên phải nhất của cha của nó thì $sibling[x]$ = NIL.

Biểu diễn heap nhị thức (tiếp)

- Ngoài ra mỗi nút x còn có một trường sau
  - *degree*[x]: bậc của x (= số các con của x)
- Các gốc của các cây nhị thức trong một heap nhị thức được tổ chức thành một danh sách liên kết, gọi là *danh sách các gốc* của heap nhị thức.
  - Khi duyệt danh sách các gốc của một heap nhị thức thì các bậc của các gốc theo thứ tự tăng dần.
  - Nếu x là một gốc thì *sibling*[x] chỉ đến gốc kế đến trong danh sách các gốc.
- Để truy cập một heap nhị thức H
  - *head*[H]: con trỏ chỉ đến gốc đầu tiên trong danh sách các gốc của H.
    - *head*[H] = NIL nếu H không có phần tử nào.

Tạo một heap nhị thức

- Thủ tục để tạo một heap nhị thức mới:

  MAKE-BINOMIAL-HEAP

  – chiếm chổ cho và trả về một đối tượng H với $head[H] = \text{NIL}$.

  – có thời gian chạy là $\Theta(1)$.

Tìm khóa nhỏ nhất

- Thủ tục để tìm khóa nhỏ nhất trong một heap nhị thức H có n nút: BINOMIAL-HEAP-MINIMUM
  - trả về một con trỏ đến nút có khóa nhỏ nhất.

```
BINOMIAL-HEAP-MINIMUM(H)
1  y ← NIL
2  x ← head[H]
3  min ← ∞
4  while x ≠ NIL
5        do if key[x] < min
6              then min ← key[x]
7                   y ← x
8           x ← sibling[x]
9  return y
```

  - Thời gian chạy của thủ tục là $O(\lg n)$ vì cần kiểm tra nhiều lắm là $\lfloor \lg n \rfloor + 1$ nút gốc.

Liên kết hai cây nhị thức

- Thủ tục để liên kết hai cây nhị thức:

  BINOMIAL-LINK

  – liên kết cây nhị thức B_{k-1} có gốc tại nút y vào cây nhị thức B_{k-1} có gốc tại nút z để tạo ra cây nhị thức B_k . Nút z trở nên gốc của một cây B_k .

```
BINOMIAL-LINK(y, z)
1  p[y] ← z
2  sibling[y] ← child[z]
3  child[z] ← y
4  degree[z] ← degree[z] + 1
```

  – Thời gian chạy của thủ tục là $O(1)$.

Hòa nhập hai heap nhị thức

- Thủ tục để hòa nhập (merge) danh sách các gốc của heap nhị thức H_1 và danh sách các gốc của heap nhị thức H_2 :

  BINOMIAL-HEAP-MERGE(H_1 , H_2)

  – hòa nhập các danh sách các gốc của H_1 và H_2 thành một danh sách các gốc duy nhất mà các bậc có thứ tự tăng dần.

  – nếu các danh sách các gốc của H_1 và H_2 có tổng cộng là m gốc, thì thời gian chạy của thủ tục là $O(m)$.

Hợp hai heap nhị thức

- Thủ tục để hợp hai heap nhị thức:

  BINOMIAL-HEAP-UNION

  – hợp nhất hai heap nhị thức H_1 và H_2 và trả về heap kết quả.

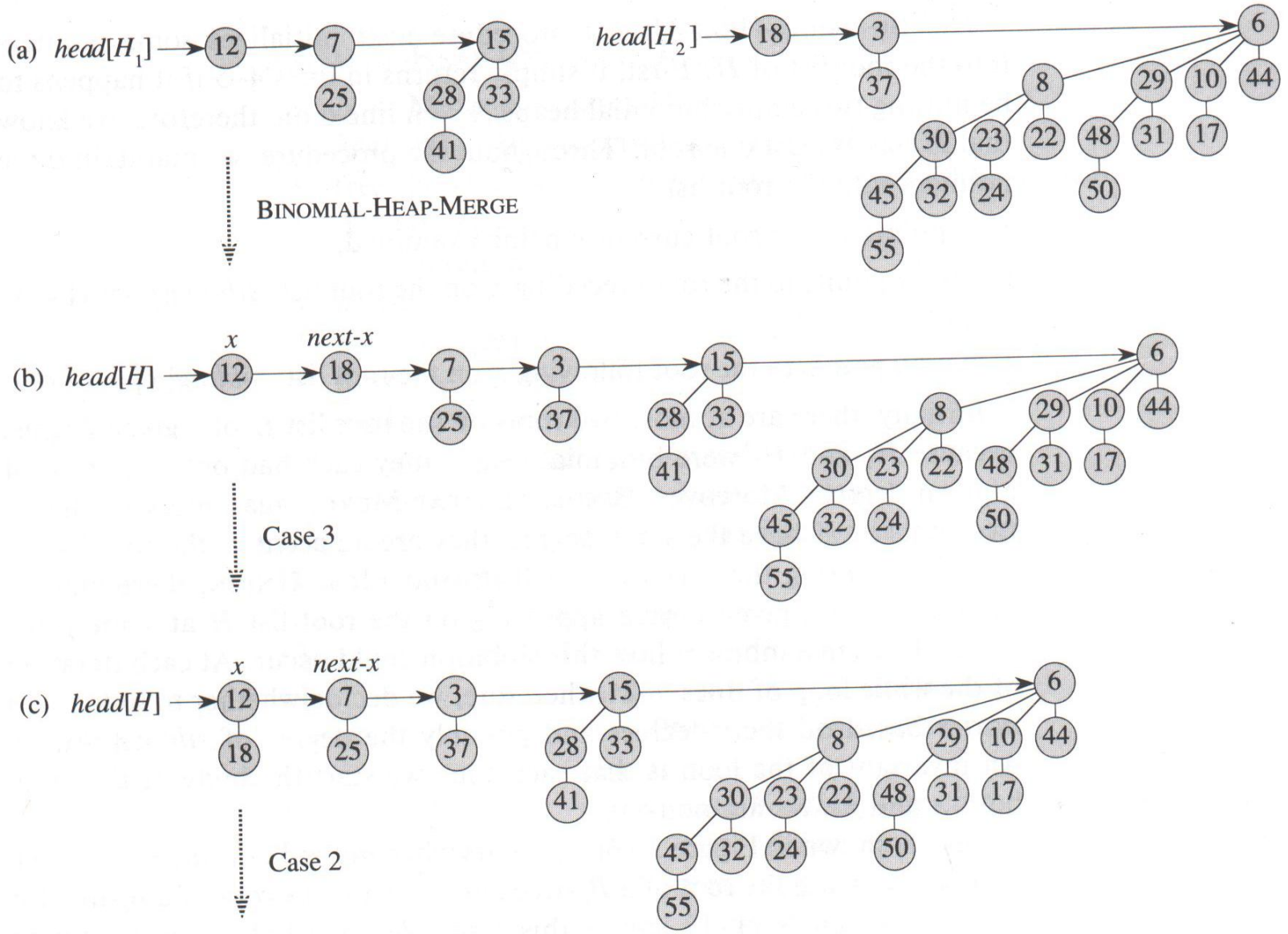
```
BINOMIAL-HEAP-UNION(H1 , H2)
1    H ← MAKE-BINOMIAL-HEAP()
2    head[H] ← BINOMIAL-HEAP-MERGE(H1 , H2)
3    trả lại các đối tượng H1 và H2 nhưng giử lại các danh sách mà
     chúng chỉ vào
4    if head[H] = NIL
5       then return H
6    prev-x ← NIL
7    x ← head[H]
8    next-x ← sibling[x]
```

Hợp hai heap nhị thức

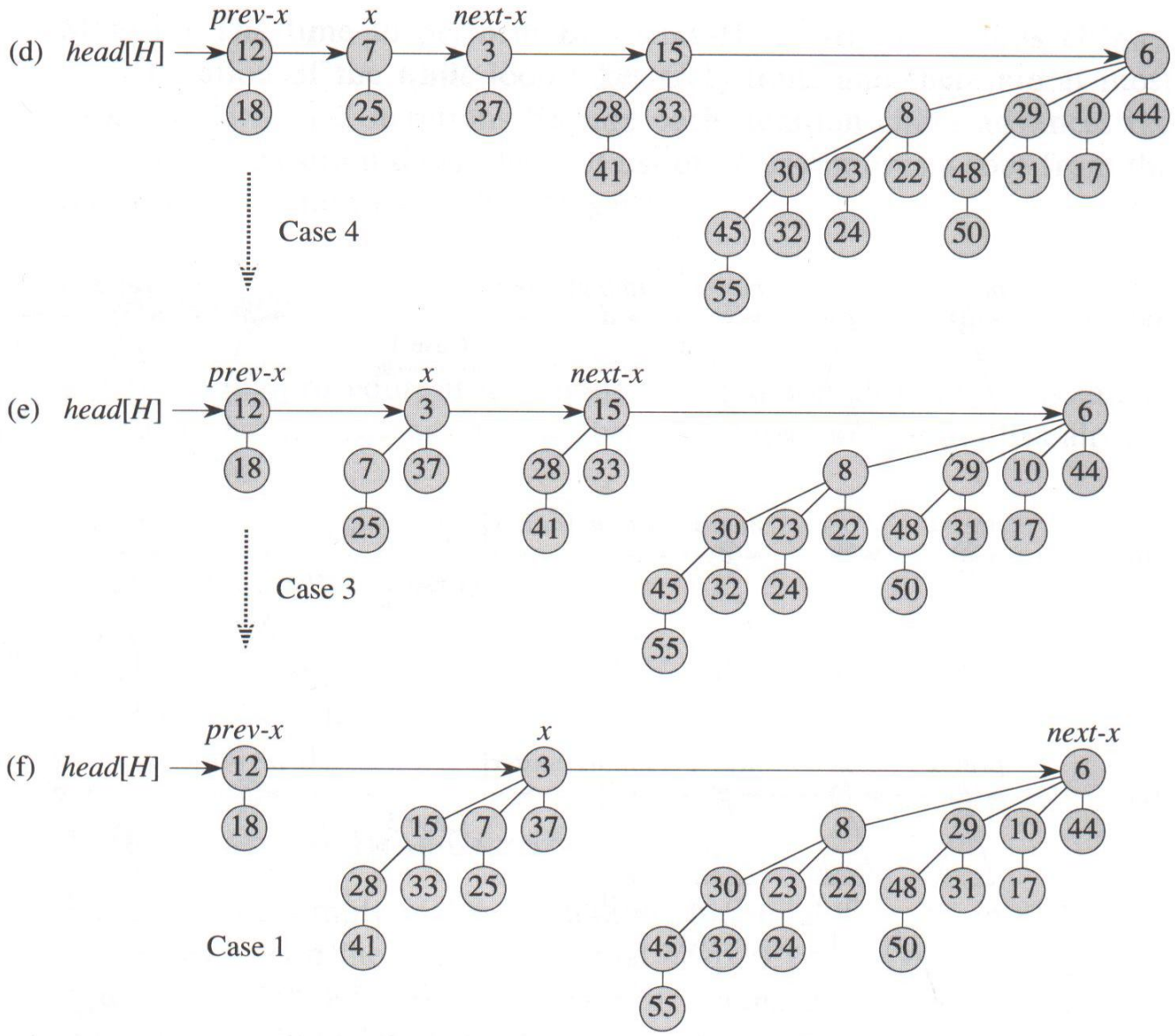
(tiếp)

```
 9  while next-x ≠ NIL
10      do if (degree[x] ≠ degree[next-x] hay
                  (sibling[next-x] ≠ NIL
                   và degree[sibling[next-x]] = degree[x])
11           then prev-x ← x                          ∇ Trường hợp 1 và 2
12                x ← next-x                          ∇ Trường hợp 1 và 2
13           else if key[x] ≤ key [next-x]
14                then sibling[x] ← sibling[next-x]   ∇ Trường hợp 3
15                     BINOMIAL-LINK(next-x, x)       ∇ Trường hợp 3
16                else if prev-x = NIL                ∇ Trường hợp 4
17                     then head[H] ← next-x          ∇ Trường hợp 4
18                     else sibling[prev-x] ← next-x  ∇ Trường hợp 4
19                     BINOMIAL-LINK(x, next-x)       ∇ Trường hợp 4
20                     x ← next-x                     ∇ Trường hợp 4
21           next-x ← sibling[x]
22  return H
```

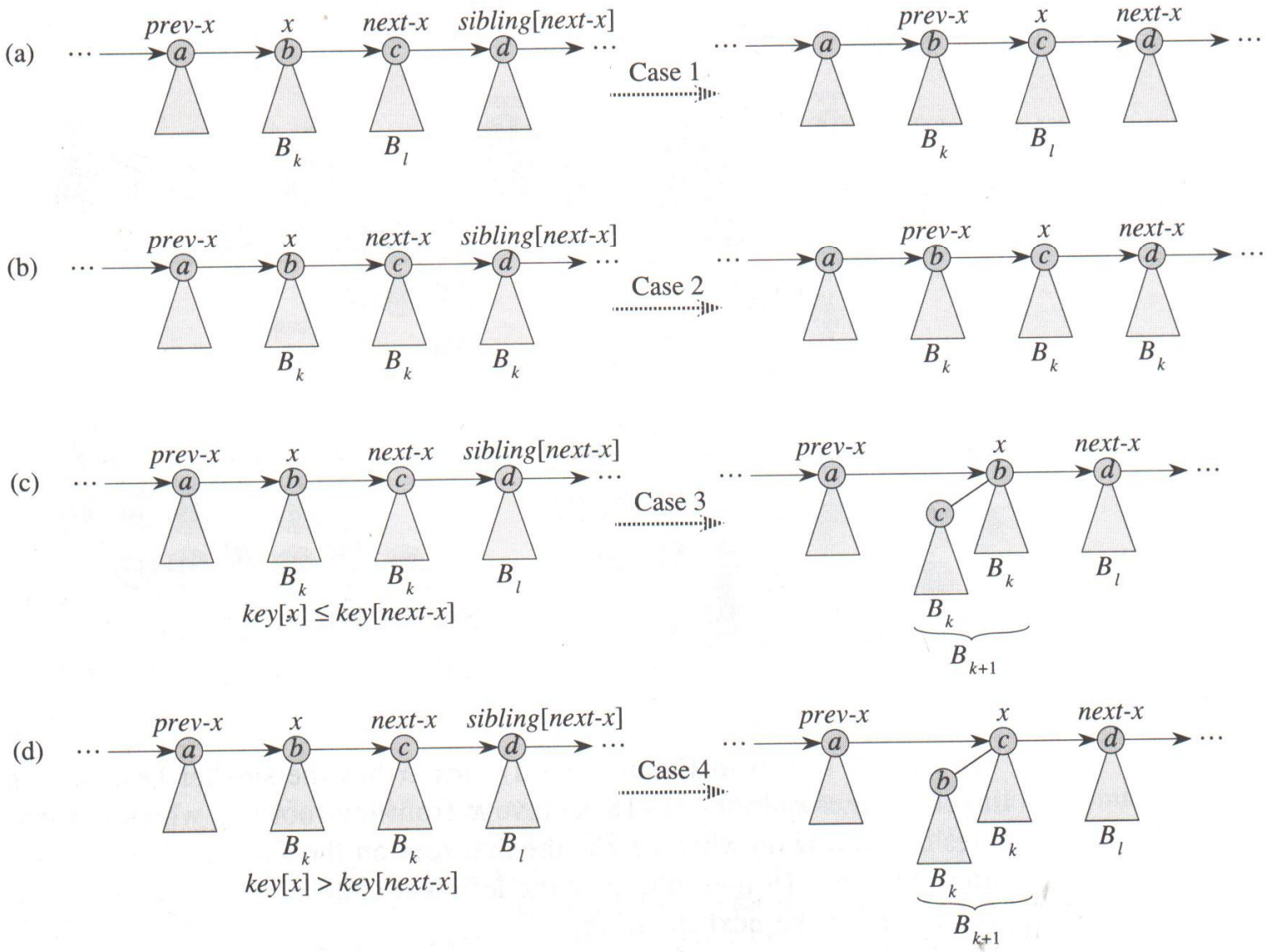
Ví dụ thực thi BINOMIAL-HEAP-UNION



Ví dụ thực thi BINOMIAL-HEAP-UNION (tiếp)



Các trường hợp xảy ra trong BINOMIAL-HEAP-UNION



Phân tích BINOMIAL-HEAP-UNION

- Thời gian chạy của BINOMIAL-HEAP-UNION là $O(\lg n)$, với n là số nút tổng cộng trong các heaps H_1 và H_2 . Đó là vì
  - Gọi n_1 là số nút của H_1, và n_2 là số nút của H_2 , ta có $n = n_1 + n_2$.
  - Do đó H_1 chứa tối đa $\lfloor \lg n_1 \rfloor + 1$ nút gốc, và H_2 chứa tối đa $\lfloor \lg n_2 \rfloor + 1$ nút gốc. Vậy BINOMIAL-HEAP-MERGE chạy trong thời gian $O(\lg n)$.
  - H chứa tối đa $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 = O(\lg n)$ nút ngay sau khi thực thi xong BINOMIAL-HEAP-MERGE. Do đó vòng lặp **while** lặp tối đa $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ lần, mỗi lần lặp tốn $O(1)$ thời gian.

Chèn một nút

- Thủ tục để chèn một nút vào một heap nhị thức:

  BINOMIAL-HEAP-INSERT

  – chèn một nút x vào một heap nhị thức H, giả sử đã dành chổ cho x và khóa của x, $key[x]$, đã được điền vào.

```
BINOMIAL-HEAP-INSERT(H, x)
1  H' ← MAKE-BINOMIAL-HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
4  sibling[x] ← NIL
5  degree[x] ← 0
6  head[H'] ← x
7  H ← BINOMIAL-HEAP-UNION(H, H')
```

  – Thời gian chạy của thủ tục là $O(\lg n)$.

Tách ra nút có khóa nhỏ nhất

- Thủ tục để tách ra nút có khóa nhỏ nhất khỏi heap nhị thức:

  BINOMIAL-HEAP-EXTRACT-MIN

  – đem nút có khóa nhỏ nhất khỏi heap nhị thức H và trả về một con trỏ chỉ đến nút được tách ra.

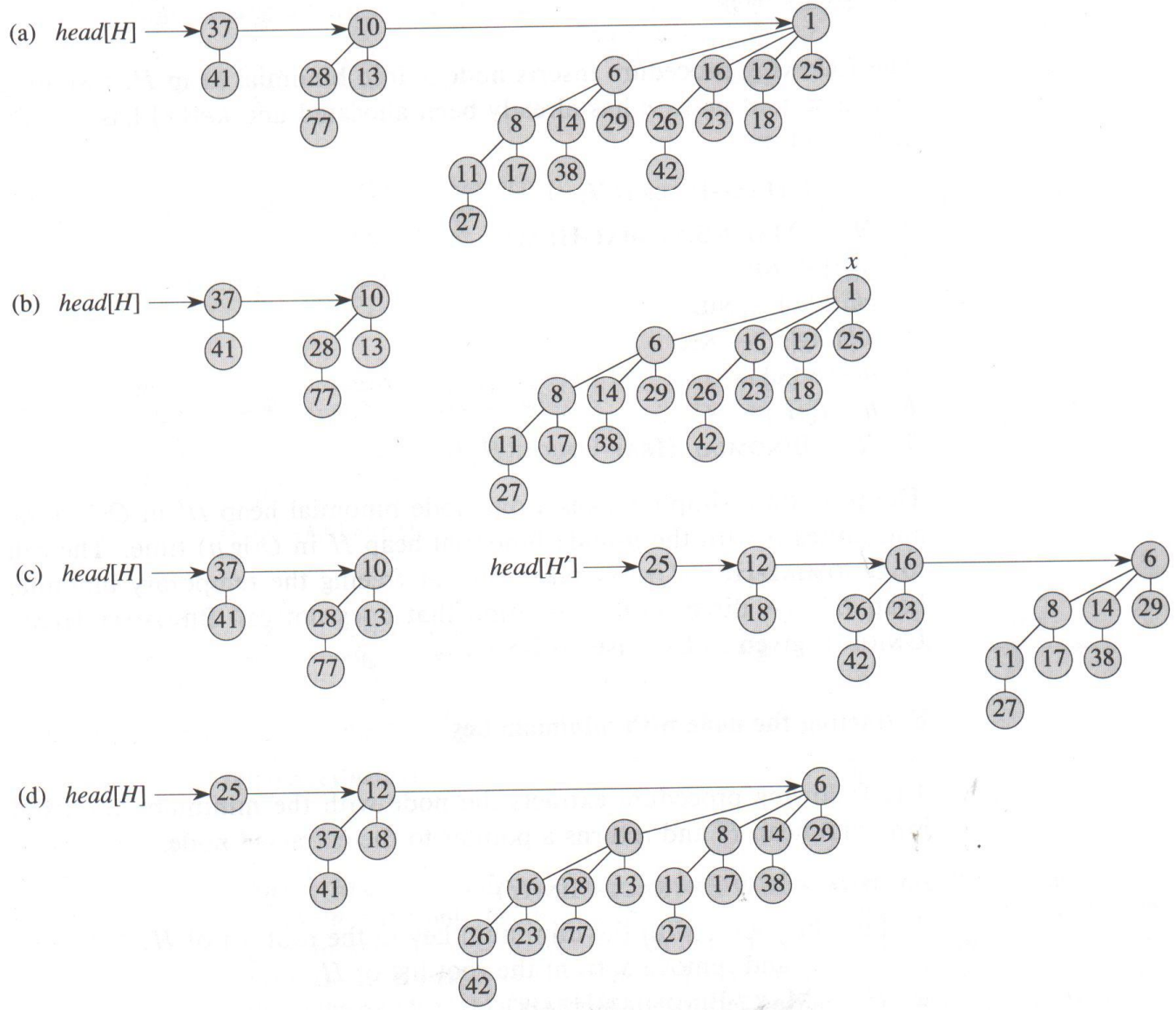
```
BINOMIAL-HEAP-EXTRACT-MIN(H)
1   tìm trong danh sách các gốc của H gốc x có khóa nhỏ nhất,
        và đem x ra khỏi danh sách các gốc của H
2   H' ← MAKE-BINOMIAL-HEAP()
3   đảo ngược thứ tự của các con của x trong danh sách liên
        kết của chúng, và gán vào head[H'] con trỏ chỉ đến đầu
        của danh sách có được
4   H ← BINOMIAL-HEAP-UNION(H, H')
5   return x
```

Tách ra nút có khóa nhỏ nhất

(tiếp)

– Thời gian chạy của thủ tục là $O(\lg n)$ vì nếu H có n nút thì mỗi dòng từ 1 đến 4 thực thi trong thời gian $O(\lg n)$.

Ví dụ thực thi BINOMIAL-HEAP-EXTRACT-MIN



Giảm khóa

- Thủ tục để giảm khóa của một nút trong một heap nhị thức thành một trị mới:

  BINOMIAL-HEAP-DECREASE-KEY

  – giảm khóa của một nút x trong một heap nhị thức H thành một trị mới k.

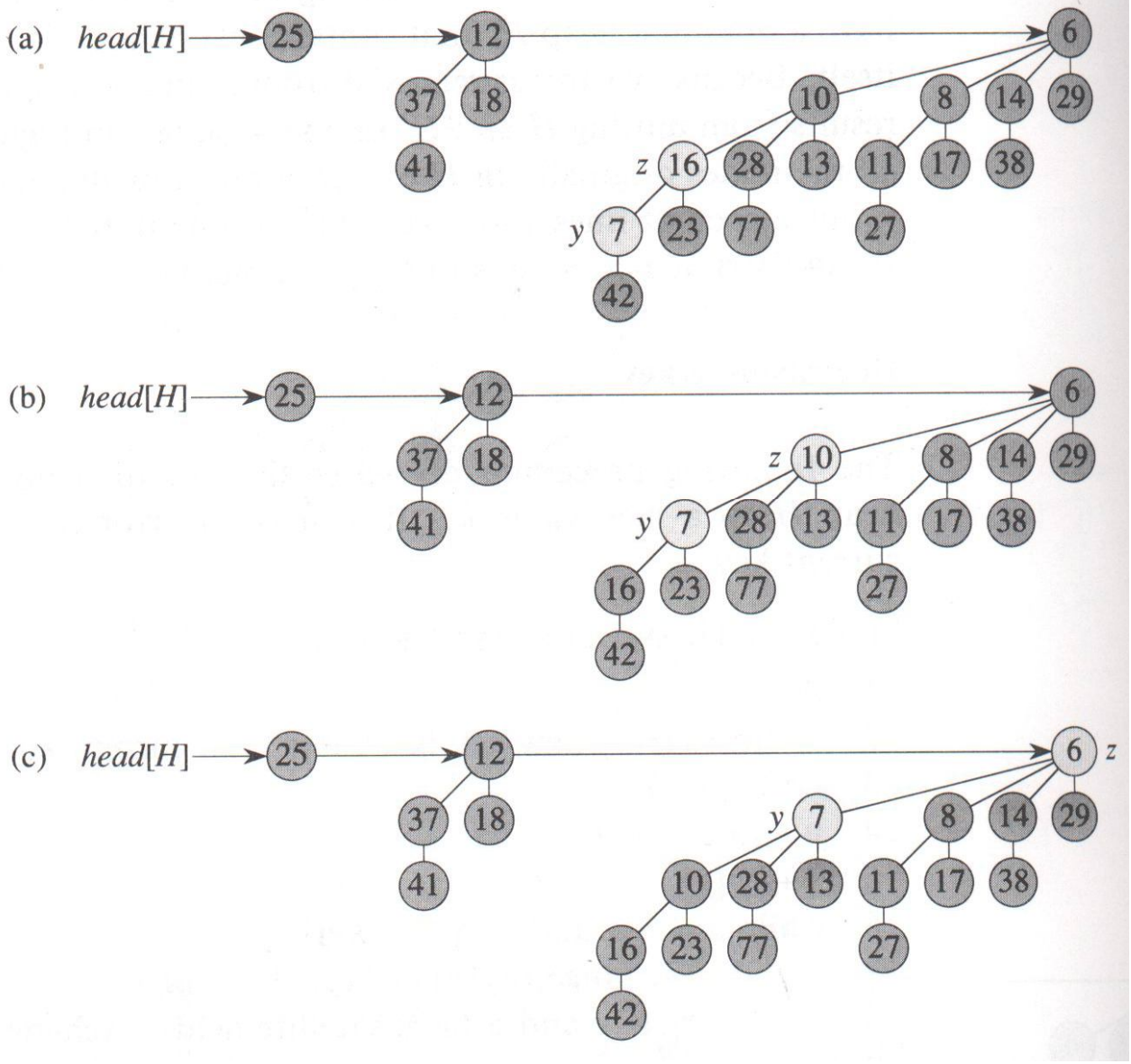
Giảm khóa

– Tính chất *heap-ordered* của cây chứa x phải được duy trì!

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
 1  if k > key[x]
 2      then error "khóa mới lớn hơn khóa hiện thời"
 3  key[x] ← k
 4  y ← x
 5  z ← p[y]
 6  while z ≠ NIL và key[y] < key[z]
 7        do tráo key[y] ↔ key[z]
 8           ∇ Nếu y và z có thông tin phụ thì cũng tráo chúng
 9           y ← z
10           z ← p[y]
```

– Thời gian chạy của thủ tục là $O(\lg n)$: vì x có độ sâu tối đa là $\lfloor \lg n \rfloor$ nên vòng lặp **while** (dòng 6-10) lặp tối đa $\lfloor \lg n \rfloor$ lần.

Ví dụ thực thi BINOMIAL-HEAP-DECREASE-KEY



Xóa một khóa

- Thủ tục để xóa khóa của một nút x:

  BINOMIAL-HEAP-DELETE

  – xóa khóa của một nút x khỏi heap nhị thức H.

```
BINOMIAL-HEAP-DELETE(H, x)
1  BINOMIAL-HEAP-DECREASE-KEY(H, x, −∞)
2  BINOMIAL-HEAP-EXTRACT-MIN(H)
```

  – Thời gian chạy của thủ tục là $O(\lg n)$.

Fibonacci heap

- Ứng dụng của Fibonacci heap
  - Giải thuật Prim để xác định một cây khung nhỏ nhất trong một đồ thị có trọng số.
  - Giải thuật Dijkstra để tìm một đường đi ngắn nhất trong đồ thị có hướng và có trọng số dương.

Cấu trúc của Fibonacci heap

- **Định nghĩa**
- Một *Fibonacci heap* là một tập các cây mà mỗi cây đều là heap-ordered.
  - Cây trong Fibonacci heap không cần thiết phải là cây nhị thức.
  - Cây trong Fibonacci heap là có gốc nhưng không có thứ tự (unordered).

Cấu trúc của Fibonacci heap (tiếp)

- Hiện thực Fibonacci heap trong bộ nhớ:

  Mỗi nút x có các trường

  – $p[x]$: con trỏ đến nút cha của nó.

  – *child*$[x]$: con trỏ đến một con nào đó trong các con của nó.

  - Các con của x được liên kết với nhau trong một danh sách vòng liên kết kép (circular, doubly linked list), gọi là *danh sách các con* của x.
  - Mỗi con y trong danh sách các con của x có các con trỏ

    – *left*$[y]$, *right*$[y]$ chỉ đến các anh em bên trái và bên phải của y.

    Nếu y là con duy nhất của x thì *left*$[y]$ = *right*$[y]$ = y.

Cấu trúc của Fibonacci heap

(tiếp)

Các trường khác trong nút x

- *degree*[x]: số các con chứa trong danh sách các con của nút x
- *mark*[x]: có trị bool là TRUE hay FALSE,

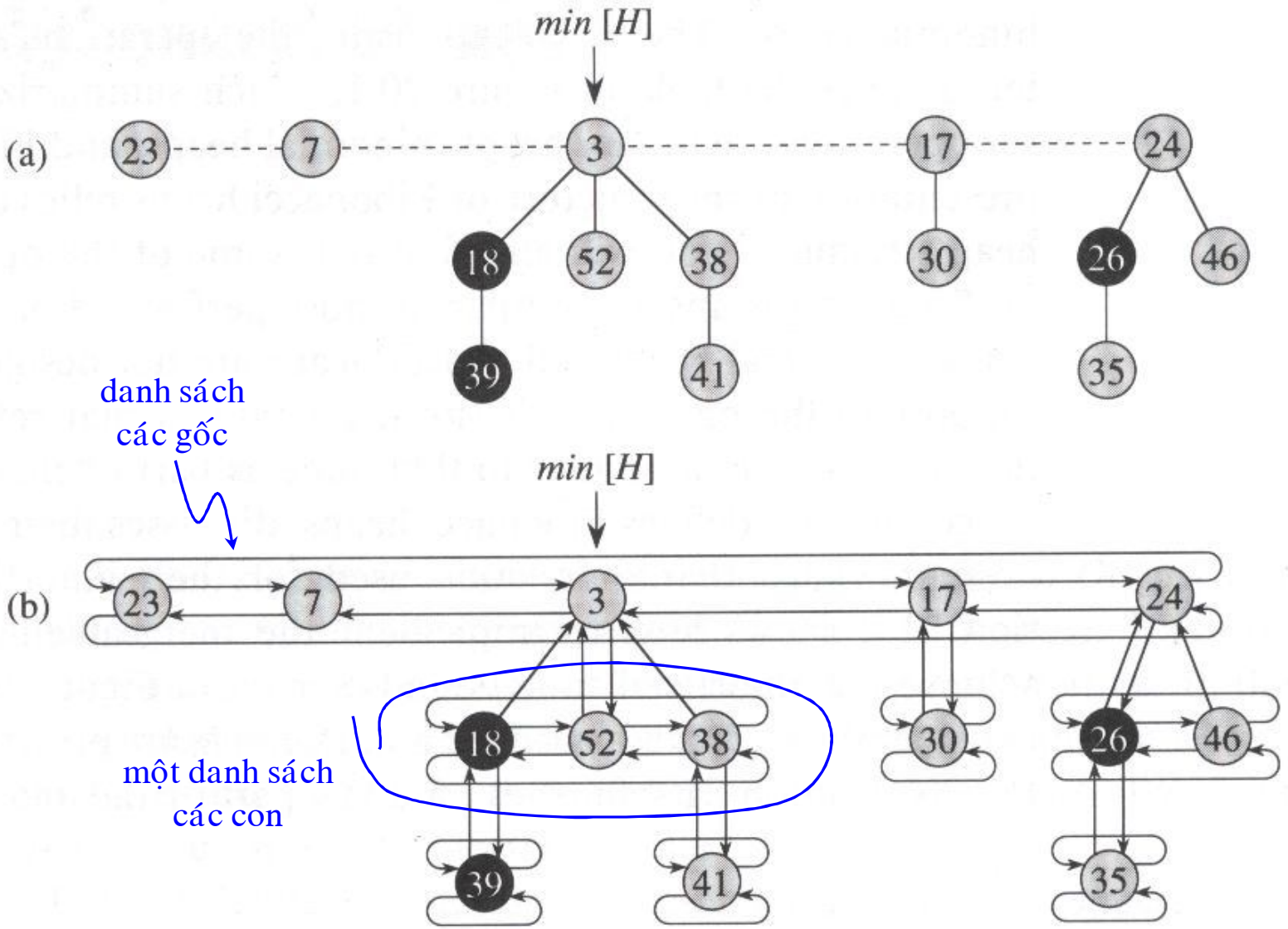
  chỉ rằng x có mất một con hay không kể từ lần cuối mà x được làm thành con của một nút khác.

Cấu trúc của Fibonacci heap

(tiếp)

- Nếu H là Fibonacci heap
  - Truy cập H bằng con trỏ *min*[H] đến nút gốc của cây chứa khoá nhỏ nhất gọi là *nút nhỏ nhất* của H.
    - Nếu H là trống thì *min*[H] = NIL.
  - Tất cả các nút gốc của các cây trong H được liên kết với nhau bởi các con trỏ *left* và *right* của chúng thành một sách liên kết kép vòng gọi là *danh sách các gốc* của H.
  - n[H]: số các nút hiện có trong H.

Cấu trúc của Fibonacci heap: ví dụ



Hàm thế năng

- Dùng phương pháp thế năng để phân tích hiệu suất của các thao tác lên các Fibonacci heap.
- Cho một Fibonacci heap H
  - gọi số các cây của Fibonacci heap H là $t(H)$
  - gọi số các nút x được đánh dấu ($mark[x]$ = TRUE) là $m(H)$.
- *Hàm thế năng* của H được định nghĩa như sau
  - $\Phi(H) = t(H) + 2\, m(H)$
  - thế năng của một tập các Fibonacci heap là tổng của các thế năng của các Fibonacci heap thành phần.

Hàm thế năng (tiếp)

- Khi bắt đầu hàm thế năng có trị là 0, sau đó hàm thế năng có trị ≥ 0. Do đó chi phí khấu hao tổng cộng là một cận trên của chi phí thực sự tổng cộng cho dảy các thao tác.

Bậc tối đa

- Gọi $D(n)$ là cận trên cho bậc lớn nhất của một nút bất kỳ trong một Fibonacci heap có n nút.
- Sẽ chứng minh: $D(n) = O(\lg n)$.

Các thao tác lên heap hợp nhất được

- Nếu chỉ dùng các thao tác lên heap hợp nhất được:
  - MAKE-HEAP
  - INSERT
  - MINIMUM
  - EXTRACT-MIN
  - UNION
- thì mỗi Fibonacci heap là một tập các cây nhị thức "*không thứ tự*".

Cây nhị thức không thứ tự

- Một *cây nhị thức không thứ tự* (unordered binomial tree) được định nghĩa đệ quy như sau
  - Cây nhị thức không thứ tự U_0 gồm một nút duy nhất.
  - Một cây nhị thức không thứ tự U_k được tạo bởi hai cây nhị thức không thứ tự U_{k-1} bằng cách lấy gốc của cây này làm con (vị trí trong danh sách các con là tùy ý) của gốc của cây kia.
- Lemma 19.1 đúng cho các cây nhị thức cũng đúng cho các cây nhị thức không thứ tự, nhưng với thay đổi sau cho tính chất 4:

  4'. Đối với cây nhị thức không thứ tự U_k , nút gốc có bậc là k, trị k lớn hơn bậc của mọi nút bất kỳ khác. Các con của gốc là gốc của các cây con U_0 , U_1 ,..., U_{k-1} trong một thứ tự nào đó.

Tạo một Fibonacci heap mới

- Thủ tục để tạo một Fibonacci heap trống:
- MAKE-FIB-HEAP
  - cấp phát và trả về đối tượng Fibonacci heap H, với

    $n[H] = 0$,

    $min[H] = \text{NIL}$
- Phân tích thủ tục MAKE-FIB-HEAP
  - Chi phí thực sự là $O(1)$
  - Thế năng của Fibonacci heap rỗng là

    $\Phi(H) = t(H) + 2\, m(H)$

    $= 0$.
  - Vậy chi phí khấu hao là $O(1)$.

Chèn một nút vào Fibonacci heap

- Thủ tục để chèn một nút vào một Fibonacci heap:

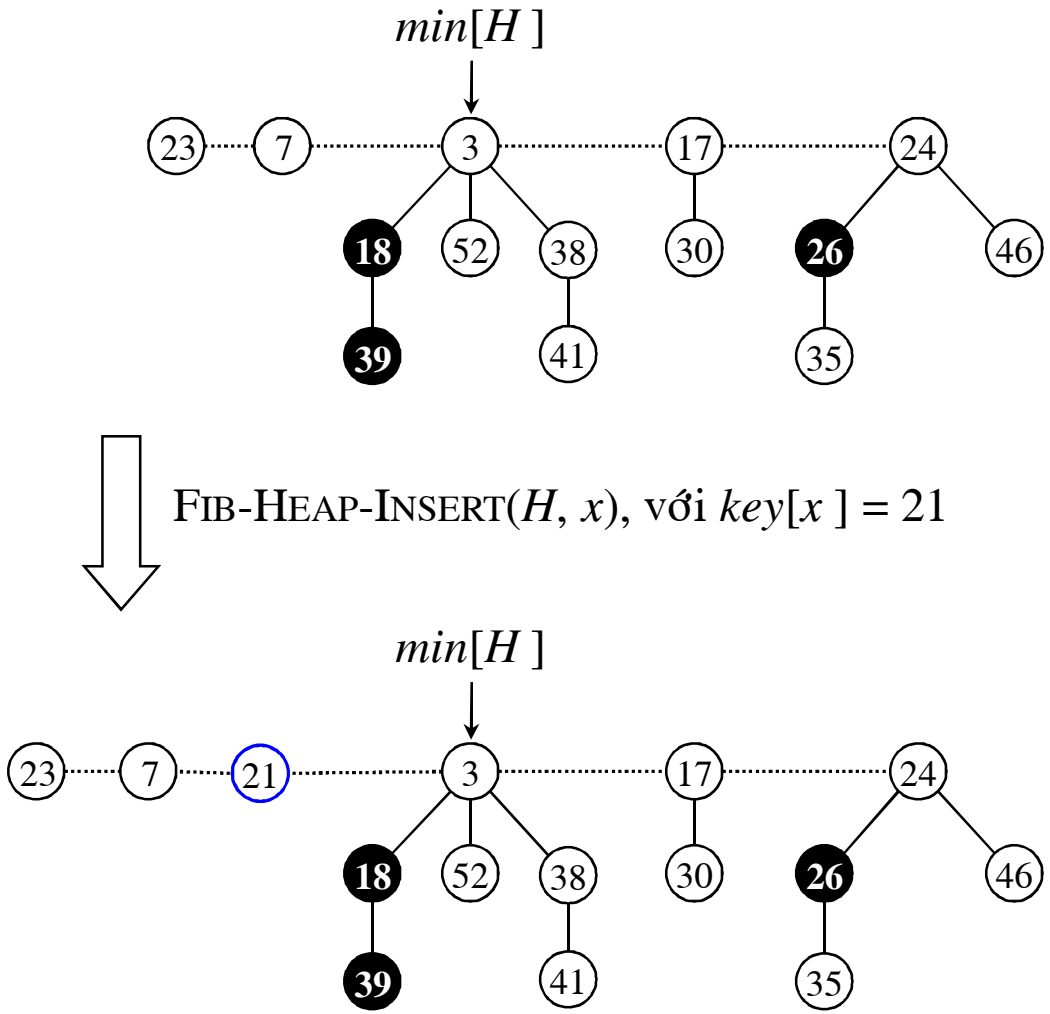
  FIB-HEAP-INSERT

  – chèn nút x (mà $key[x]$ đã được gán trị) vào Fibonacci heap H

```
FIB-HEAP-INSERT(H, x)
 1  degree[x] ← 0
 2  p[x] ← NIL
 3  child[x] ← NIL
 4  left [x] ← x
 5  right [x] ← x
 6  mark [x] ← FALSE
 7  nối danh sách các gốc chứa x vào danh sách các gốc của H
 8  if min[H] = NIL or key[x] < key [min[H]]
 9     then min[H] ← x
10  n[H] ← n[H] + 1
```

Ví dụ chèn một nút vào Fibonacci heap

- (tiếp)



Chèn một nút vào Fibonacci heap (tiếp)

- Phân tích thủ tục FIB-HEAP-INSERT:

  Phí tổn khấu hao là $O(1)$ vì

  – Gọi H là Fibonacci heap đầu vào, và H' là Fibonacci heap kết quả.

  – Ta có: $t(H') = t(H) + 1$,

  $$m(H') = m(H).$$

  Vậy hiệu thế $\Phi(H') - \Phi(H)$ bằng

  $$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1.$$

  – Phí tổn khấu hao bằng

  $$\text{phí tổn thực sự} + \text{hiệu thế} = O(1) + 1$$
  $$= O(1).$$

Tìm nút nhỏ nhất

- Con trỏ $min[H]$ chỉ đến nút nhỏ nhất của Fibonacci heap H.
- Phân tích:
  - Phí tổn thực sự là $O(1)$
  - Hiệu thế là 0 vì thế năng của H không thay đổi
  - Vậy phí tổn khấu hao là $O(1)$ (= phí tổn thực sự).

Hợp nhất hai Fibonacci heap

- Thủ tục để hợp nhất hai Fibonacci heap:

  FIB-HEAP-UNION

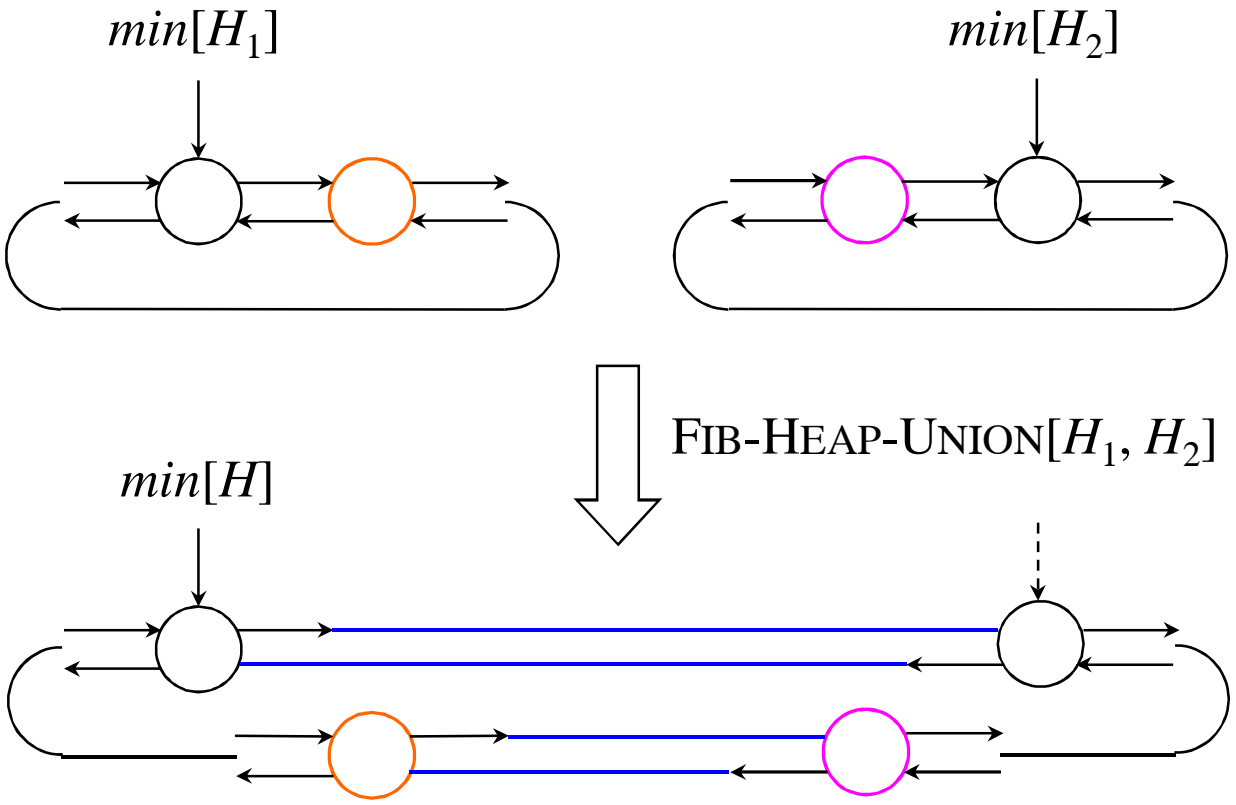
  – hợp nhất các Fibonacci heap H_1 và H_2

  – trả về H, Fibonacci heap kết quả

```
FIB-HEAP-UNION(H1, H2)
1  H ← MAKE-FIB-HEAP()
2  min[H] ← min[H1]
3  nối danh sách các gốc của H2 với danh sách các gốc của H
4  if (min[H1] = NIL) or (min[H2] ≠ NIL and min[H2] < min[H1])
5     then min[H] ← min[H2]
6  n[H] ← n[H1] + n[H2]
7  giải phóng (free) các đối tượng H1 và H2
8  return H
```

Hợp nhất hai Fibonacci heap

- (tiếp)
  - Ví dụ: giả sử $min[H_1] < min[H_2]$



Hợp nhất hai Fibonacci heap (tiếp)

- Phân tích thủ tục FIB-HEAP-UNION:

  Phí tổn khấu hao được tính từ

  – phí tổn thực sự là $O(1)$

  – hiệu thế là

  $$\Phi(H) - (\Phi(H_1) + \Phi(H_2))$$
  $$= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2)))$$
  $$= 0, \quad \text{vì} \quad t(H) = t(H_1) + t(H_2) \text{ và}$$
  $$m(H) = m(H_1) + m(H_2)$$

  – Vậy phí tổn khấu hao = phí tổn thực sự + hiệu thế
  $$= O(1)$$

Tách ra nút nhỏ nhất

- Thủ tục để tách ra nút nhỏ nhất:

  FIB-HEAP-EXTRACT-MIN

  – tách nút nhỏ nhất khỏi Fibonacci heap H

```
FIB-HEAP-EXTRACT-MIN(H)
 1   z ← min[H]
 2   if z ≠ NIL
 3      then for mỗi con x của z
 4                do thêm x vào danh sách các gốc của H
 5                   p[x] ← NIL
 6           đem z ra khỏi danh sách các gốc của H
 7           if z = right[z]
 8              then min[H] ← NIL
 9              else min[H] ← right[z]
10                   CONSOLIDATE(H)
11           n[H] ← n[H] − 1
12   return z
```

Củng cố (consolidate)

- Thủ tục phụ: *củng cố* danh sách các gốc của một Fibonacci heap H
  - liên kết mọi cặp gốc có cùng bậc thành một gốc mới cho đến khi mọi gốc có bậc khác nhau.

```
CONSOLIDATE(H )
 1  for i ← 0 to D(n[H ])
 2        do A[i ] ← NIL
 3  for mỗi nút w trong danh sách các gốc của H
 4        do x ← w
 5           d ← degree[x ]
 6           while A[d ] ≠ NIL
 7                 do y ← A[d ]
 8                    if key[x ] > key[y ]
 9                       then tráo x ↔ y
10                    FIB-HEAP-LINK(H, y, x)
11                    A[d ] ← NIL
12                    d ← d + 1
13           A[d ] ← x
```

Củng cố (consolidate)

- (tiếp)

```
14  min[H ] ← NIL
15  for i ← 0 to D(n[H ])
16        do if A[i ] ≠ NIL
17              then thêm A[i ] vào danh sách các gốc của H
18                    if min[H ] = NIL or key[A[i ]] < key[min[H ]]
19                      then min[H ] ← A[i ]
```

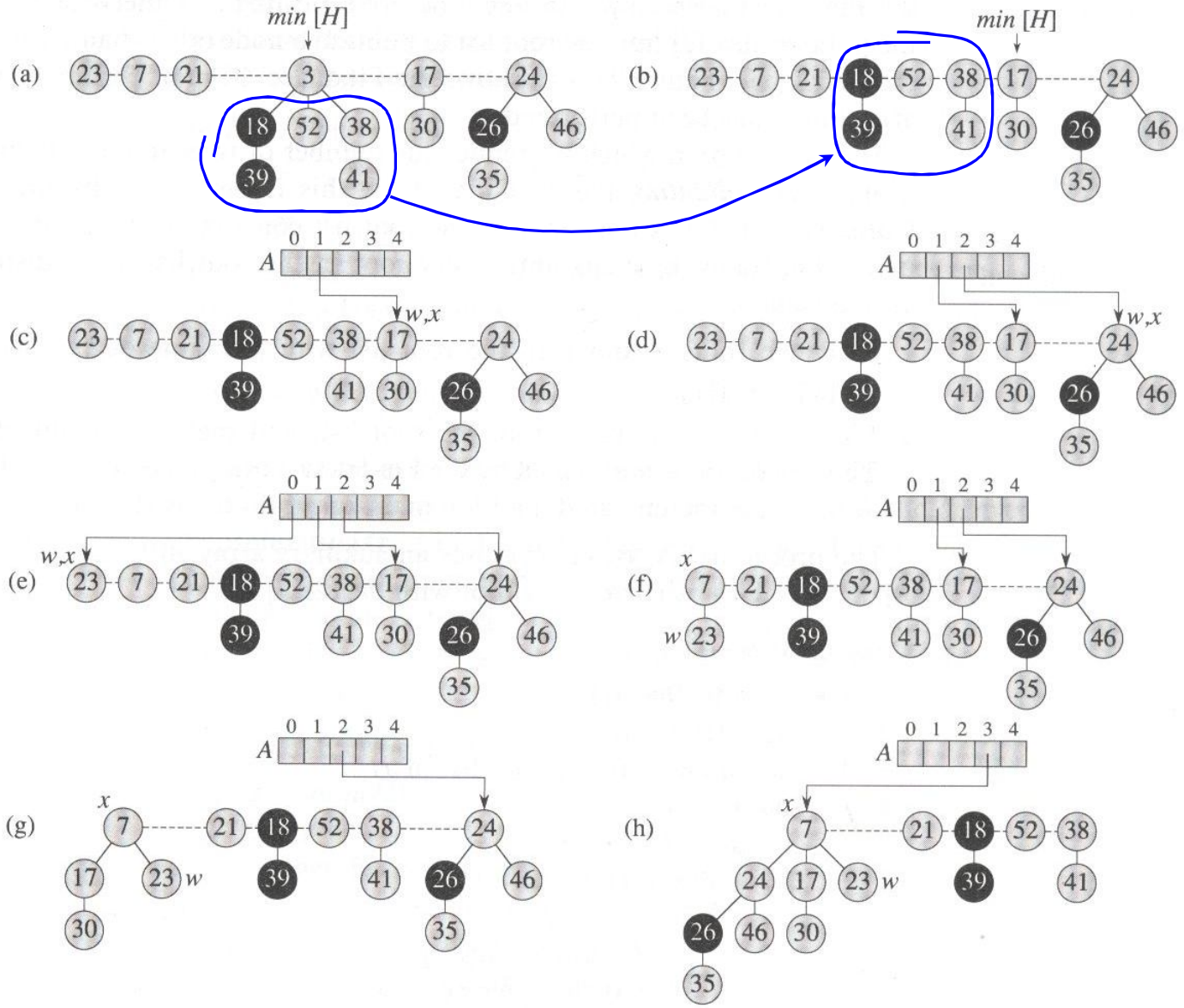
Liên kết hai gốc có cùng bậc

– Thủ tục CONSOLIDATE liên kết các gốc có cùng bậc mãi cho đến khi mọi gốc có được sau đó đều có bậc khác nhau.

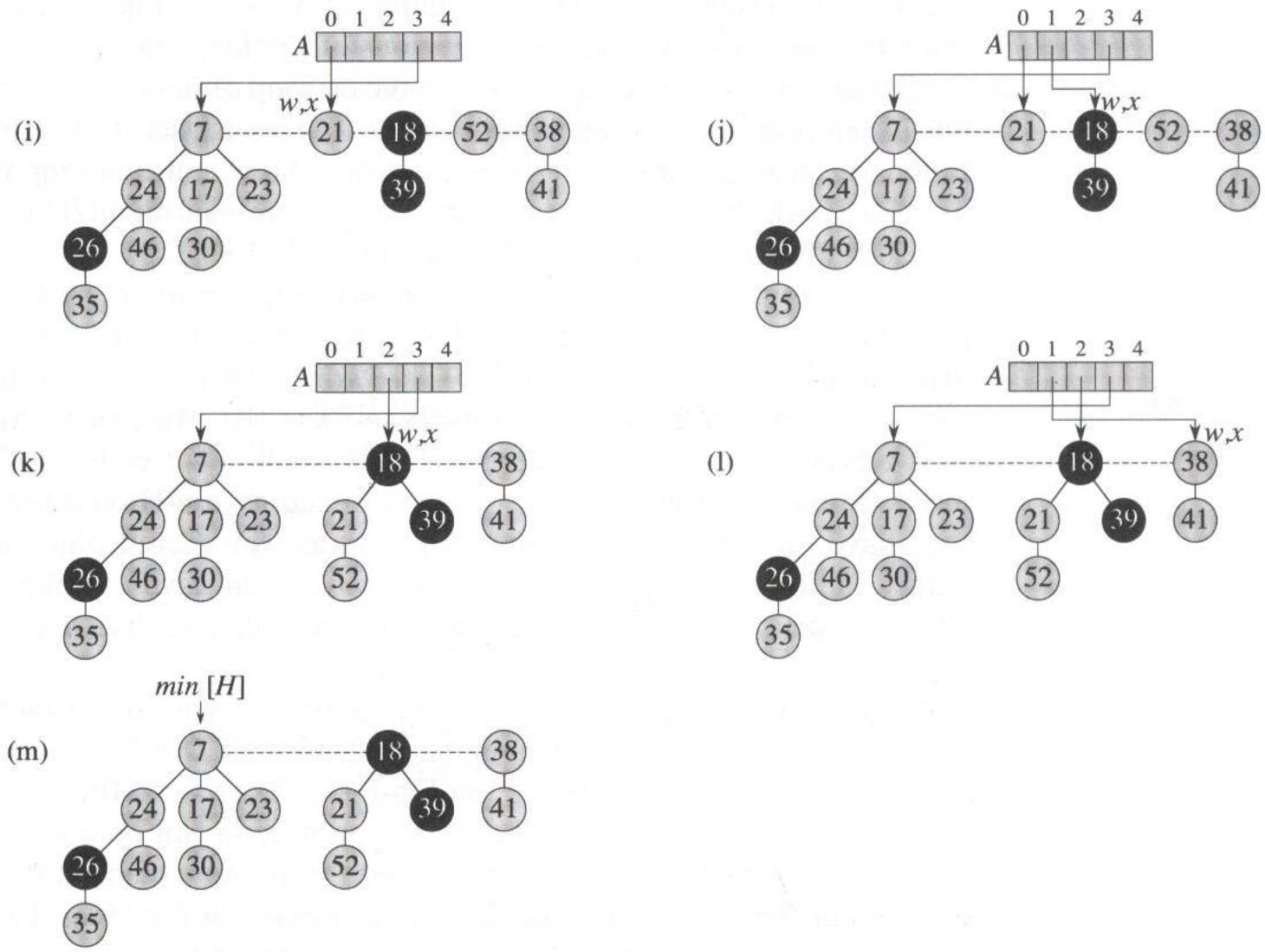
- Dùng thủ tục FIB-HEAP-LINK(H, y, x) để tách gốc y khỏi danh sách gốc của H, sau đó liên kết gốc y vào gốc x, gốc x và gốc y có cùng bậc.

```
FIB-HEAP-LINK(H, y, x)
1   đem y ra khỏi danh sách các gốc của H
2   làm y thành con của x, tăng degree[x]
3   mark[y] ← FALSE
```

Thực thi FIB-HEAP-EXTRACT-MIN: ví dụ



Thực thi FIB-HEAP-EXTRACT-MIN: ví dụ (tiếp)



Chi phí thực sự của FIB-HEAP-EXTRACT-MIN

- Gọi H là Fibonacci heap ngay trước khi gọi FIB-HEAP-EXTRACT-MIN, số nút của H là n.
  - Chi phí thực sự bao gồm:
    - $O(D(n))$: vì có nhiều lắm là $D(n)$ con của nút nhỏ nhất cần được xử lý bởi:
      - FIB-HEAP-EXTRACT-MIN
      - các dòng 1-2 và 14-19 của CONSOLIDATE
    - $O(D(n) + t(H))$: vì khi gọi CONSOLIDATE chiều dài của danh sách gốc nhiều lắm là $D(n) + t(H) - 1$, mà thời gian chạy vòng lặp **for** dòng 3-13 nhiều lắm là tỉ lệ với chiều dài của danh sách gốc này.
  - Vậy chi phí thực sự của FIB-HEAP-EXTRACT-MIN là $O(D(n) + t(H))$.

Chi phí khấu hao của FIB-HEAP-EXTRACT-MIN

- Gọi H' là Fibonacci heap sau khi gọi FIB-HEAP-EXTRACT-MIN lên H.
  - Nhắc lại: hàm thế năng của H được định nghĩa là
    - $\Phi(H) = t(H) + 2\, m(H)$
  - Biết:

    chi phí khấu hao = chi phí thực sự + $\Phi(H') - \Phi(H)$
    - Đã tính: phí tổn thực sự của FIB-HEAP-EXTRACT-MIN là $O(D(n) + t(H))$.
    - Sau khi gọi FIB-HEAP-EXTRACT-MIN lên H, số gốc (hay số cây) của H' nhiều lắm là $D(n) + 1$, và không có nút nào được đánh dấu. Vậy
      - $\Phi(H') = (D(n) + 1) + 2\, m(H)$.

Chi phí khấu hao của FIB-HEAP-EXTRACT-MIN

- (tiếp)
  - Do đó chi phí khấu hao của FIB-HEAP-EXTRACT-MIN là
  - $O(D(n) + t(H)) + ((D(n) + 1) + 2\,m(H)) - (t(H) + 2\,m(H))$
    - $= O(D(n)) + O(t(H)) - \underbrace{t(H)}$

      đến từ thế năng

    - $= O(D(n))$,
  - vì có thể scale up đơn vị của thế năng để khống chế hằng số ẩn trong $O(t(H))$.
  - (Ví dụ: với $O(t(H)) = 99t(H) + 77$ thì có thể chọn 1 đơn vị thế năng = 78)

Giảm khóa của một nút

- Thủ tục để giảm khóa của một nút:

  FIB-HEAP-DECREASE-KEY

  – giảm khóa của nút x trong Fibonacci heap H thành trị mới k nhỏ hơn trị cũ của khóa.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > key[x]
2     then error "khóa mới lớn hơn khóa hiện thời"
3  key[x] ← k
4  y ← p[x]
5  if y ≠ NIL and key[x] < key[y]
6     then CUT(H, x, y)
7          CASCADING-CUT(H, y)
8  if key[x] < key[min[H]]
9     then min[H] ← x
```

Giảm khóa của một nút (tiếp)

- Thủ tục phụ để cắt liên kết giữa x và y, cha của nó, sau đó làm x thành một gốc.

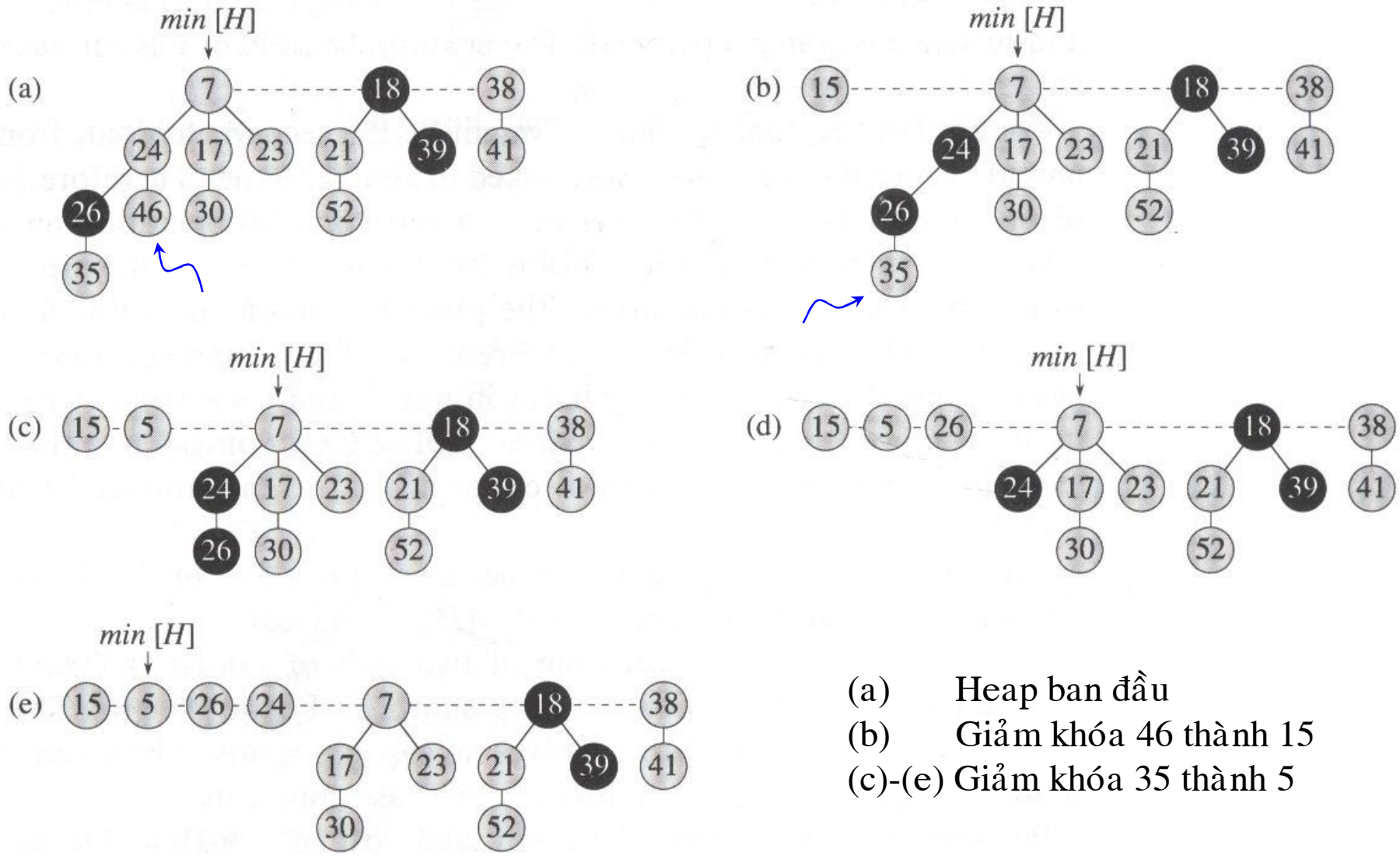
```
CUT(H, x, y)
1  đem x ra khỏi danh sách các con của y, giảm degree[y]
2  thêm x vào danh sách các gốc của H
3  p[x] ← NIL
4  mark[x] ← FALSE
```

Giảm khóa của một nút (tiếp)

- Thủ tục phụ để xử lý cha của nút bị cắt dựa trên trường *mark*[*x*].

```
CASCADING-CUT(H, y)
1   z ← p[y]
2   if z ≠ NIL
3      then if mark[y] = FALSE
4                then mark[y] ← TRUE
5                else CUT(H, y, z)
6                     CASCADING-CUT(H, z)
```

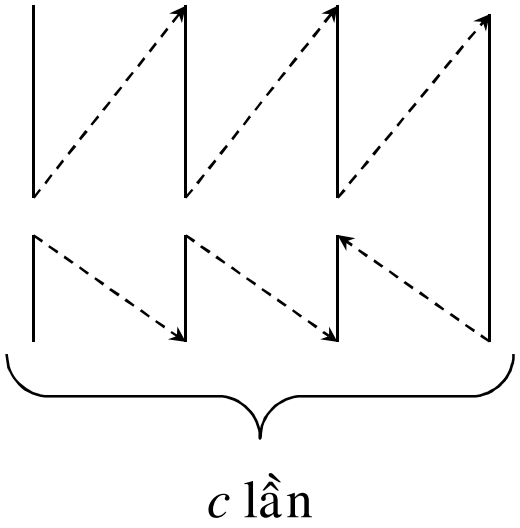
Giảm khoá của một nút: ví dụ



(a) Heap ban đầu
(b) Giảm khóa 46 thành 15
(c)-(e) Giảm khóa 35 thành 5

Chi phí thực sự của FIB-HEAP-DECREASE-KEY

- Gọi H là Fibonacci heap ngay trước khi gọi FIB-HEAP-DECREASE-KEY, số nút của H là n.
  - Chi phí thực sự của FIB-HEAP-DECREASE-KEY bao gồm:
    - $O(1)$: dòng 1-5 và 8-9,
    - thời gian thực thi các cascading cuts. Giả sử CASCADING-CUT được gọi đệ quy c lần. Thời gian thực thi CASCADING-CUT là $O(1)$ không kể các gọi đệ quy.



Chi phí thực sự của FIB-HEAP-DECREASE-KEY

- (tiếp)
  - Vậy phí tổn thực sự của FIB-HEAP-DECREASE-KEY là $O(c)$.

Chi phí khấu hao của FIB-HEAP-DECREASE-KEY

- Gọi H' là Fibonacci heap sau khi gọi FIB-HEAP-DECREASE-KEY lên H.
  - Nhắc lại: hàm thế năng của H được định nghĩa là
    - $\Phi(H) = t(H) + 2\,m(H)$
  - chi phí khấu hao = chi phí thực sự + $\Phi(H') - \Phi(H)$
    - Đã tính: chi phí thực sự của FIB-HEAP-DECREASE-KEY là $O(c)$.
    - Sau khi gọi FIB-HEAP-DECREASE-KEY lên H, thì H' có $t(H) + c$ cây.
    - $\Phi(H') - \Phi(H) \le (t(H) + c) + 2(m(H) - c + 2) - (t(H) + 2\,m(H))$
    - $\le 4 - c.$

số lần gọi CUT bằng số lần gọi CASCADING-CUT = c, mà
– mỗi lần thực thi CUT thì 1 nút trở thành cây
– mỗi lần thực thi CASCADING-CUT ngoại trừ lần cuối của gọi đệ quy thì 1 nút được unmarked và lần cuối của gọi đệ quy CASCADING-CUT có thể marks 1 nút.

Chi phí khấu hao của FIB-HEAP-DECREASE-KEY

- (tiếp)
  - Do đó chi phí khấu hao của FIB-HEAP- DECREASE-KEY là
    - $O(c) + \underbrace{4 - c}_{\text{đến từ thế năng}} = O(1)$,
  - vì có thể scale up đơn vị của thế năng để khống chế hằng số ẩn trong $O(c)$.

Xóa một nút

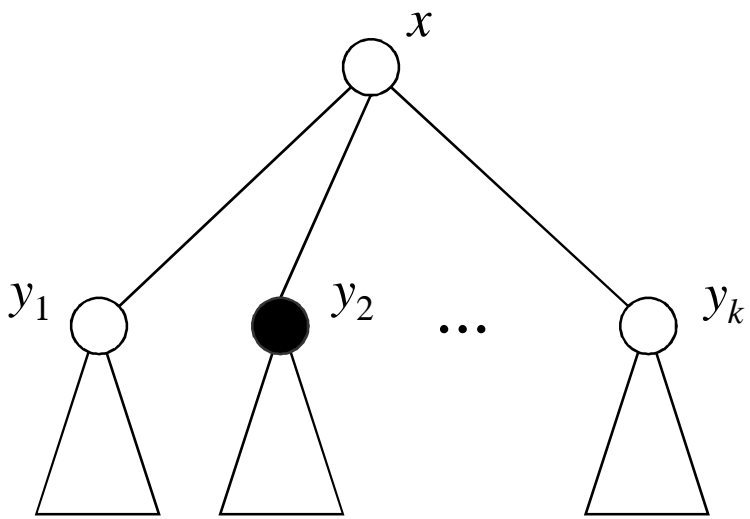
- Thủ tục để xóa một nút:

  FIB-HEAP-DELETE

  – Xóa một nút x khỏi một Fibonacci heap H.

```
FIB-HEAP-DELETE(H, x)
1   FIB-HEAP-DECREASE-KEY(H, x, −∞)
2   FIB-HEAP -EXTRACT-MIN(H)
```

Chận trên lên bậc lớn nhất



- **Lemma** (sách: Lemma 21.1)

  Cho x là một nút bất kỳ trong một Fibonacci heap, và giả sử $degree[x] = k$. Gọi $y_1, y_2, ..., y_k$ là các con của x được xếp theo thứ tự lúc chúng được liên kết vào x, từ lúc sớm nhất đến lúc trễ nhất. Thì $degree[y_1] \geq 0$ và $degree[y_i] \geq i - 2$ với $i = 2, 3, ..., k$.

Chận trên lên bậc lớn nhất

(tiếp)

Chứng minh

– Rõ ràng là $degree[y_1] \geq 0$.

$i \geq 2$:

– Khi y_i được liên kết vào x thì $y_1, y_2, ..., y_{i-1}$ là trong tập các con của x nên khi đó $degree[x] \geq i - 1$.

- Nút y_i được liên kết vào x chỉ khi nào $degree[x] = degree[y_i]$, vậy khi đó $degree[y_i]$ cũng $\geq i - 1$.

– Kể từ khi đó đến nay, nút y_i mất nhiều lắm là một con, vì nếu nó mất hai con thì nó đã bị cắt khỏi x. Vậy

$$degree[y_i] \geq (i - 1) - 1$$
$$\geq i - 2 .$$

Chận trên lên bậc lớn nhất (tiếp)

- **Định nghĩa**
- Với $k = 0, 1, 2,...$ định nghĩa F_k là *số Fibonacci thứ k*:

$$F_k = \begin{cases} 0 & \text{nếu } k = 0, \\ 1 & \text{nếu } k = 1, \\ F_{k-1} + F_{k-2} & \text{nếu } k \geq 2. \end{cases}$$

- **Lemma** (sách: Lemma 21.2, bài tập)
  Với mọi số nguyên $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i \ .$$

Lemma (Bài tập 2.2-8)
Với mọi số nguyên $k \geq 0$, ta có $F_{k+2} \geq \phi^k$, trong đó $\phi = (1 + \sqrt{5}) / 2$, *số vàng*.

Chận trên lên bậc lớn nhất (tiếp)

- **Lemma** (sách: Lemma 21.3)

  Cho x là một nút bất kỳ trong một Fibonacci heap, và cho $k = degree[x]$. Thì size$(x) \geq F_{k+2} \geq \phi^k$, trong đó $\phi = (1 + \sqrt{5}) / 2$.

  Chứng minh

  – Gọi s_k là trị nhỏ nhất có thể được của size(z) trên mọi nút z mà $degree[z] = k$.

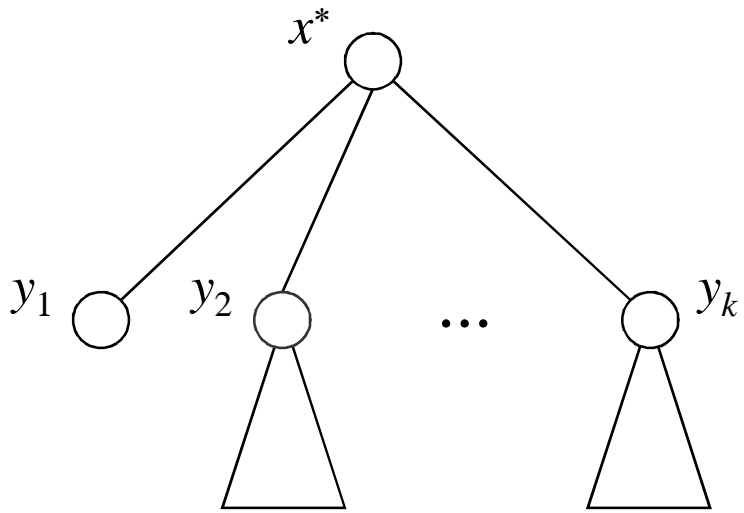
  – Rõ ràng là $s_0 = 1$, $s_1 = 2$, $s_2 = 3$.

  – Ta có $s_k \leq$ size(x)

Chận trên lên bậc lớn nhất

Chứng minh (tiếp)

$$\begin{aligned} \text{size}(x) &\geq s_k \\ &= \text{size}(x^*) \\ &= 2 + \sum_{i=2}^{k} s_{degree[y_i]} \end{aligned}$$



– vì s_k là tăng đơn điệu theo k, nên từ $degree[y_i] \geq i - 2$ (Lemma 21.1) ta có $s_{degree[y_i]} \geq s_{i-2}$. Vậy

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

Chận trên lên bậc lớn nhất

Chứng minh (tiếp)

– dùng quy nạp theo k để chứng minh rằng $s_k \geq F_{k+2}$, với $k \geq 0$:

- Bước cơ bản: với $k = 0$ và $k = 1$ là rõ ràng.
- Bước quy nạp:
  – Giả thiết quy nạp: $k \geq 2$ và $s_i \geq F_{i+2}$ với $i = 0, 1, \ldots, k-1$. Từ trên ta có

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

$$\geq 2 + \sum_{i=2}^{k} F_i$$

$$= 1 + \sum_{i=0}^{k} F_i$$

$$= F_{k+2} \qquad \text{(Lemma 21.2)}$$

– vậy: $\text{size}(x) \geq s_k \geq F_{k+2} \geq \phi^k$.

Chận trên lên bậc lớn nhất (tiếp)

- **Hệ luận**

  Bậc lớn nhất $D(n)$ của nút bất kỳ trong một Fibonacci heap có n nút là $O(\lg n)$.

  Chứng minh

  Dùng Lemma 21.3.

Các Cấu Trúc Dữ Liệu cho các Tập Rời Nhau

Các thao tác lên cấu trúc dữ liệu các tập rời nhau

- Cấu trúc dữ liệu *các tập rời nhau* được định nghĩa bởi
  - Một tập S của các tập động rời nhau, $S = \{S_1, S_2, ..., S_k\}$
    - Mỗi tập S_i được tượng trưng bởi một phần tử *đại diện* là một phần tử nào đó của nó.
  - Các thao tác
    - MAKE-SET(x): tạo một tập mới chỉ gồm x. Vì các tập là rời nhau nên x không được đang nằm trong một tập khác.
    - UNION(x, y): tạo tập hội của các tập động S_x và S_y lần lượt chứa x và y, với điều kiện là S_x và S_y là rời nhau.
    - FIND-SET(x): trả về một con trỏ chỉ đến phần tử đại diện của tập chứa x.
- Để cho gọn, sẽ dùng "các tập rời nhau" để gọi "cấu trúc dữ liệu các tập rời nhau".

Các thao tác lên các tập rời nhau (tiếp)

- Phân tích thời gian chạy của các thao tác sẽ dựa trên hai tham số sau
  - n, số các thao tác MAKE-SET
  - m, số tổng cộng các thao tác MAKE-SET, UNION, và FIND-SET.
- Nhận xét:
  - Sau $n - 1$ lần gọi UNION lên các tập rời nhau thì còn lại đúng một tập.
  - $m \geq n$.

Một ứng dụng của các tập rời nhau

- Xác định các thành phần liên thông của một đồ thị vô hướng
  - Thủ tục CONNECTED-COMPONENTS xác định các thành phần liên thông của một đồ thị vô hướng.

$V[G]$ là tập các đỉnh của đồ thị G, $E[G]$ là tập các cạnh của G.

```
CONNECTED-COMPONENTS(G)
1  for mỗi đỉnh v ∈ V[G]
2       do MAKE-SET(v)
3  for mỗi cạnh (u, v) ∈ E[G]
4       do if FIND-SET(u) ≠ FIND-SET(v)
5             then UNION(u, v)
```

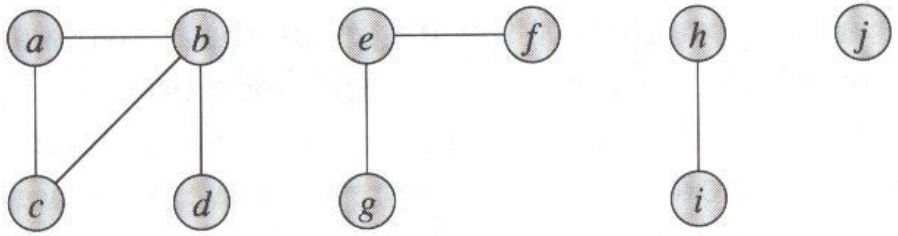
Một ứng dụng của các tập rời nhau (tiếp)

– Thủ tục SAME-COMPONENT xác định hai đỉnh có cùng một thành phần liên thông hay không.

```
SAME-COMPONENT(u, v)
1   if FIND-SET(u) = FIND-SET(v)
2       then return TRUE
3       else return FALSE
```

Thao tác lên các tập rời nhau

- Ví dụ: một đồ thị với 4 thành phần liên thông



(a)

Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

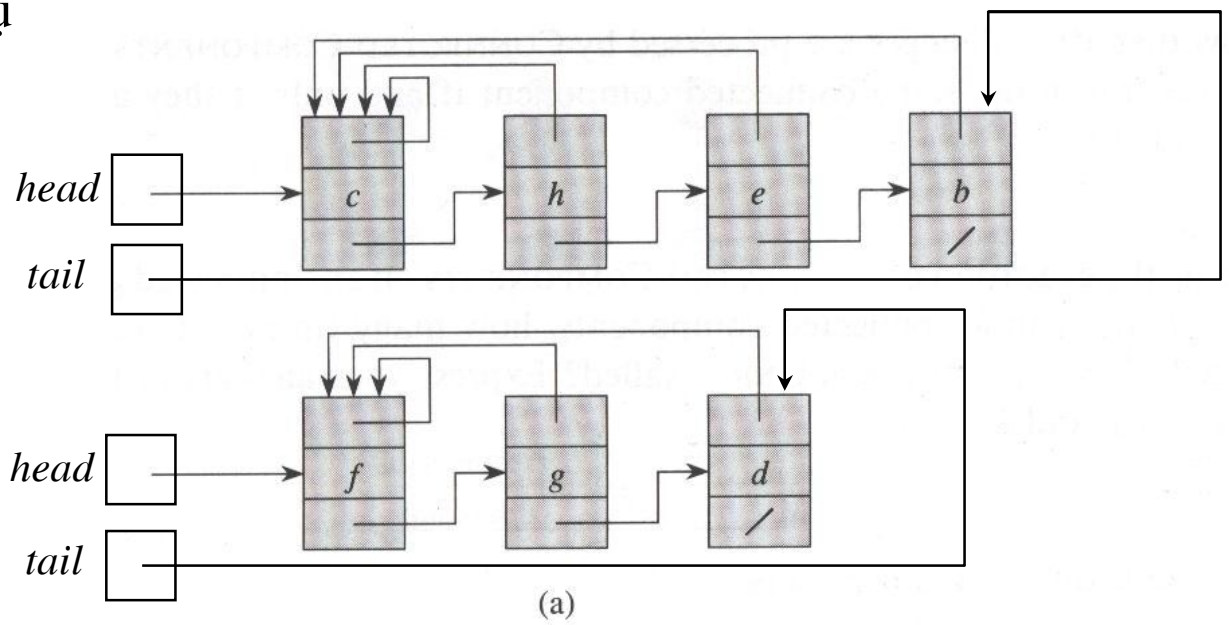
(b)

Biểu diễn các tập rời nhau dùng danh sách liên kết

- Biểu diễn các tập rời nhau dùng danh sách liên kết (linked-list representation of disjoint sets):
  - Biểu diễn mỗi tập bằng một danh sách liên kết. Trong mỗi danh sách liên kết
    - Đối tượng đứng đầu được dùng làm phần tử đại diện của tập.
    - Mỗi đối tượng trong danh sách liên kết chứa
      - phần tử của tập
      - con trỏ chỉ đến đối tượng chứa phần tử kế tiếp
      - con trỏ chỉ đến phần tử đại diện của tập.
    - Con trỏ *head* chỉ đến đại diện của tập. Con trỏ *tail* chỉ đến phần tử cuối trong danh sách.

Biểu diễn tập bằng danh sách liên kết

- Ví dụ



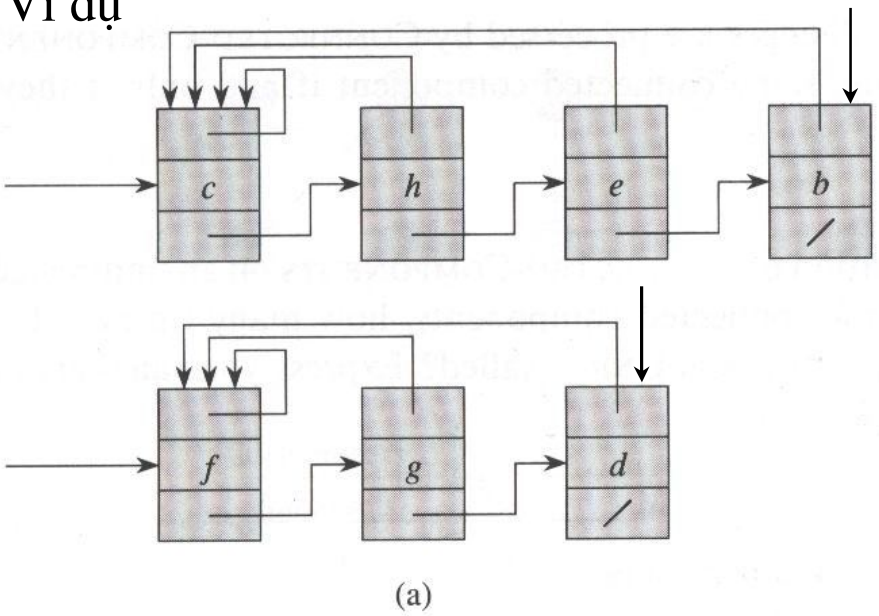
(a)

Biểu diễn tập bằng danh sách liên kết (tiếp)

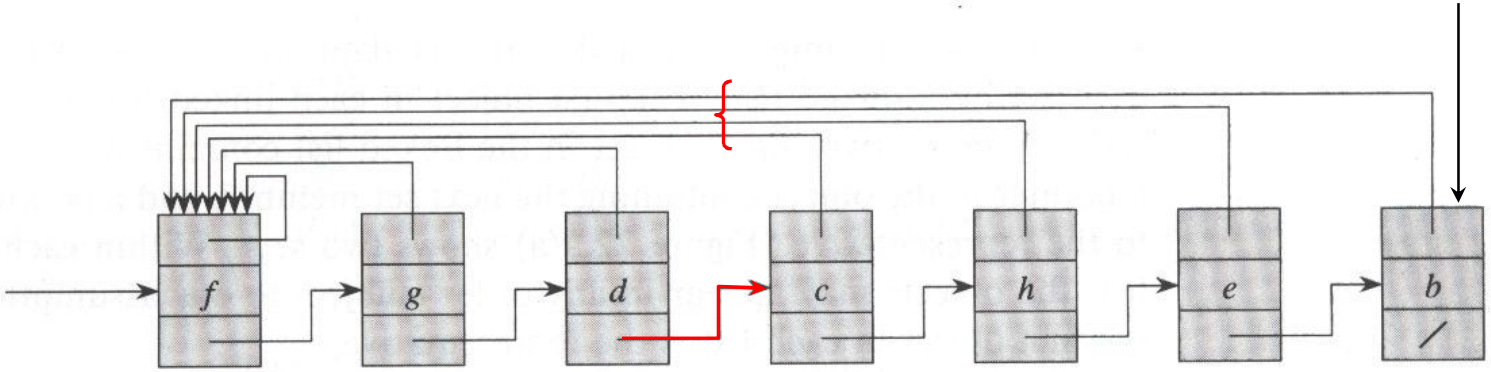
- Hiện thực các thao tác
  - Hiện thực MAKE-SET(x): tạo một danh sách liên kết chỉ gồm đối tượng x.
  - Hiện thực FIND-SET(x): trả về con trỏ đến đại diện của tập chứa x.
  - Hiện thực UNION(x, y):
    - gắn danh sách của x vào đuôi của danh sách của y
    - cập nhật các con trỏ của các đối tượng trong danh sách cũ của x để chúng chỉ đến đại diện của tập, tức là đầu của danh sách cũ của y.

Biểu diễn tập bằng danh sách liên kết (tiếp)

- Ví dụ



(a)



(b)

Thao tác UNION không dùng heuristic

- Ví dụ một chuỗi gồm $2n - 1$ thao tác lên n đối tượng mà cần $\Theta(n^2)$ thời gian.

Thao tác	Số các đối tượng được cập nhật		
MAKE-SET(x_1)	1		n
MAKE-SET(x_2)	1		
.			
.			
.			
MAKE-SET(x_n)	1		
UNION(x_1 , x_2)		1	$\Theta(n^2)$
UNION(x_2 , x_3)		2	
UNION(x_3 , x_4)		3	
.			
.			
.			
UNION(x_{n-1} , x_n)		$n - 1$	

Heuristic để tăng tốc của UNION

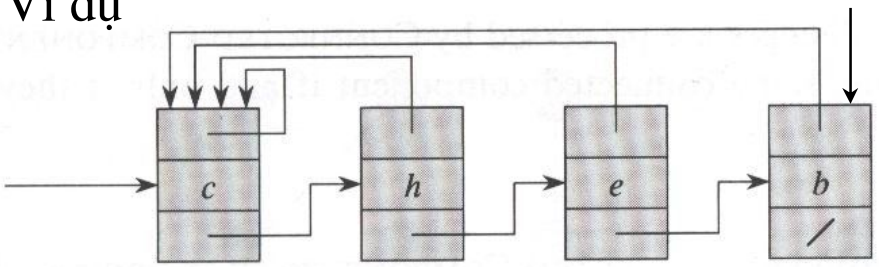
- Nhận xét: Khi hợp hai danh sách trong UNION, mọi con trỏ (chỉ đến đại diện mới) của các phần tử trong danh sách được gắn vào đuôi của danh sách kia phải được cập nhật.

  Giả sử mỗi danh sách có chứa thêm chiều dài của nó.

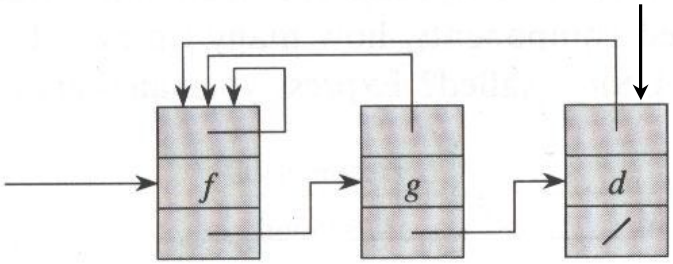
- Heuristic *hợp theo trọng số* (weighted-union heuristic): khi hợp hai danh sách
  - gắn danh sách ngắn hơn vào đuôi của danh sách dài hơn (nếu các danh sách dài như nhau thì có thể gắn tùy ý).

Heuristic hợp theo trọng số

- Ví dụ

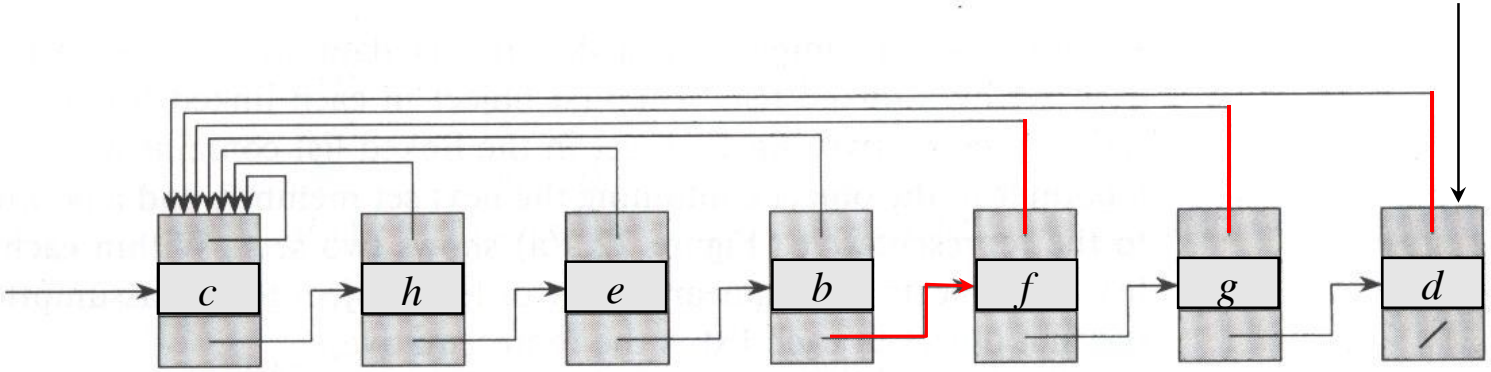


chiều dài = 4



chiều dài = 3

(a)



(b)

Biểu diễn tập bằng danh sách liên kết: thời gian chạy

- **Định lý** (Theorem 22.1)

  Bằng cách dùng biểu diễn danh sách liên kết cho các tập rời nhau và heuristic hợp theo trọng số (weighted-union heuristic), một dãy gồm có m thao tác MAKE-SET, UNION, và FIND-SET, trong đó có n thao tác là MAKE-SET, tốn $O(m + n \lg n)$ thời gian.

  Chứng minh

  - Mỗi MAKE-SET chạy trong thời gian $O(1)$
  - Mỗi FIND-SET chạy trong thời gian $O(1)$
  - Xác định thời gian chạy của các thao tác UNION:
    - Thời gian chạy của các thao tác UNION là thời gian tổng cộng lấy trên mọi phần tử của mọi lần cập nhật con trỏ chỉ đến phần tử đại diện của tập chứa phần tử đó.

Biểu diễn tập bằng danh sách liên kết: thời gian chạy

Chứng minh (tiếp theo)

- Xét đối tượng x bất kỳ trong một tập bất kỳ của các tập rời nhau. Mỗi lần con trỏ chỉ đến phần tử đại diện của tập chứa x được cập nhật, thì x phải đã nằm trong tập nhỏ hơn
- $\Rightarrow$
  - Lần 1 cập nhật con trỏ của x: tập kết quả phải có ít nhất 2 phần tử
  - Lần 2 cập nhật con trỏ của x: tập kết quả phải có ít nhất 4 phần tử
  - …
  - Lần k cập nhật con trỏ của x: tập kết quả phải có ít nhất 2^k phần tử.

  Vì tập có nhiều lắm là n phần tử nên $2^k \le n$. Vậy số lần cập nhật con trỏ của x nhiều lắm là $k \le \lg n$.
- Vì x là phần tử bất kỳ nên thời gian tổng cộng để cập nhật các con trỏ của mọi phần tử là $O(n \lg n)$.

– Thời gian chạy tổng cộng của dãy m thao tác là: $O(m) + O(n \lg n) = O(m + n \lg n)$.

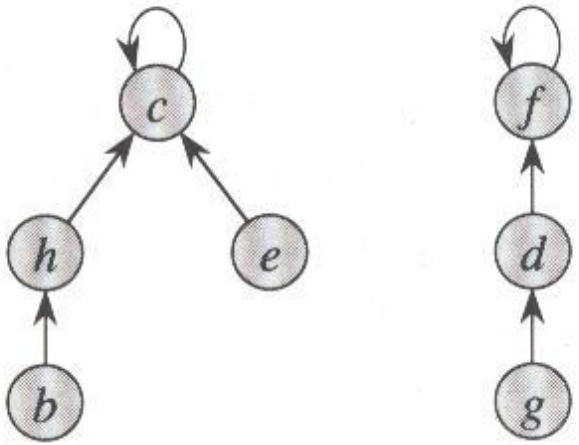
Biểu diễn các tập rời nhau bằng rừng

- Biểu diễn các tập rời nhau bằng rừng (*disjoint-set forest*)
  - Biểu diễn mỗi tập bằng một cây có gốc:
    - Mỗi nút của cây chứa một phần tử của tập

    ngoài ra
    - Mỗi nút chứa một con trỏ chỉ đến cha của nó
    - Gốc của mỗi cây chứa đại diện của tập và là cha của chính nó.

Biểu diễn các tập rời nhau bằng rừng (tiếp)

- Ví dụ
  - Hai cây sau biểu diễn các tập $\{b, c, e, h\}$ và $\{d, f, g\}$.
  - c và f lần lượt là phần tử đại diện của các tập $\{b, c, e, h\}$ và $\{d, f, g\}$.



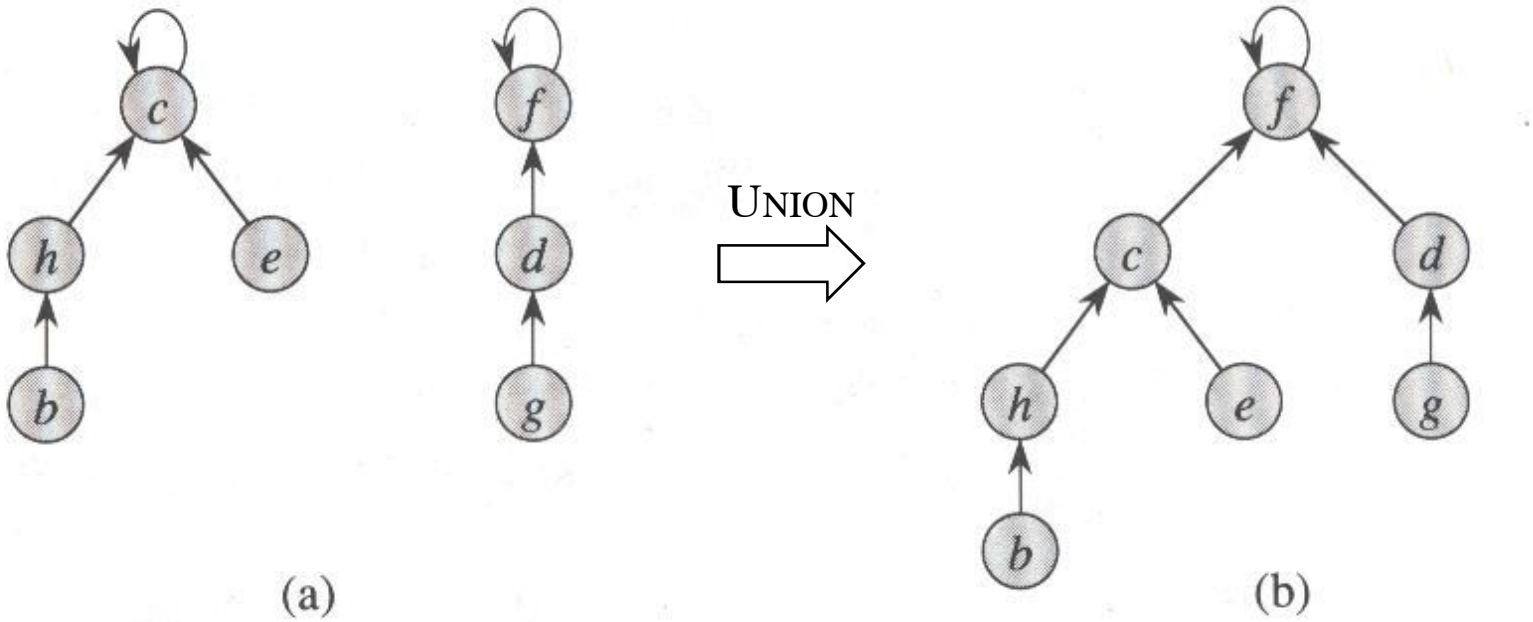
(a)

Biểu diễn các tập rời nhau bằng rừng: các thao tác

- Các thao tác lên các tập rời nhau khi biểu diễn bằng rừng
  - Hiện thực MAKE-SET: tạo một cây chỉ có một nút.
  - Hiện thực FIND-SET bằng cách đuổi theo các con trỏ chỉ đến nút cha cho đến khi tìm được nút gốc của cây.
    - Các nút được ghé qua khi gọi FIND-SET tạo thành *đường dẫn* (*find path*).
  - Hiện thực UNION: làm cho con trỏ của gốc cây này chỉ đến gốc của cây kia.

Biểu diễn các tập rời nhau bằng rừng

- Ví dụ
  - Hình (b) là kết quả của UNION(e, g).



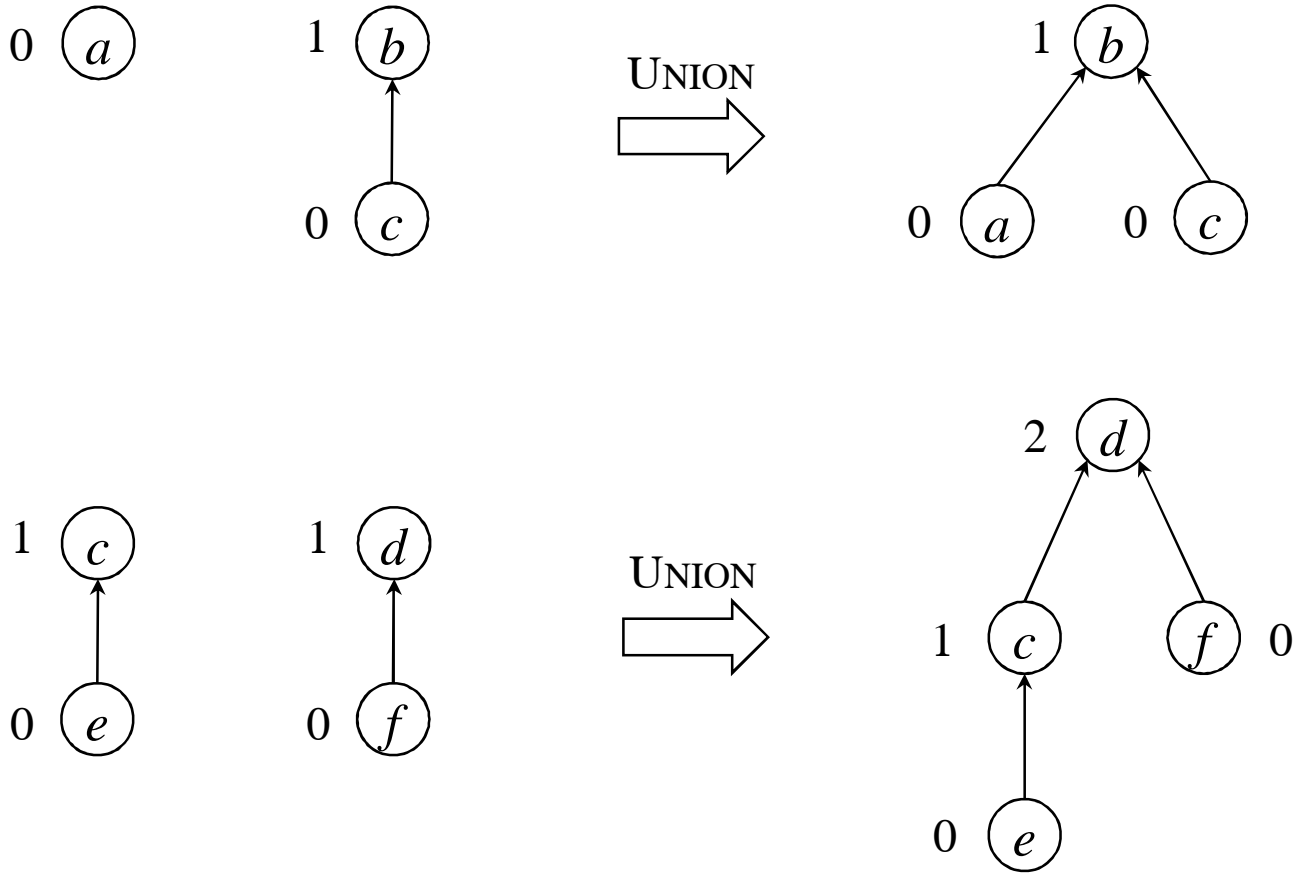
Biểu diễn tập bằng cây

- Dùng hai heuristics để giảm thời gian chạy của các dãy các thao tác lên các tập rời nhau khi hiện thực bằng rừng:
  - Heuristic *hợp theo thứ hạng* (*union by rank*) khi thực thi UNION:
    - duy trì cho mỗi nút một *rank*. *Rank* là cận trên cho độ cao (*) của nút. Mọi nút được khởi tạo với *rank* = 0.
    - khi hợp theo thứ hạng hai cây, nút gốc có *rank* nhỏ hơn được làm thành con của nút có *rank* lớn hơn.
  - Heuristic *nén đường dẫn* (*path compression*).

(*) *Độ cao* của một nút trong một cây là số các cạnh nằm trên đường đi đơn dài nhất từ nút đến một nút lá.

Heuristic hợp theo thứ hạng

- Ví dụ: (số bên cạnh mỗi đối tượng là *rank* của nó.)



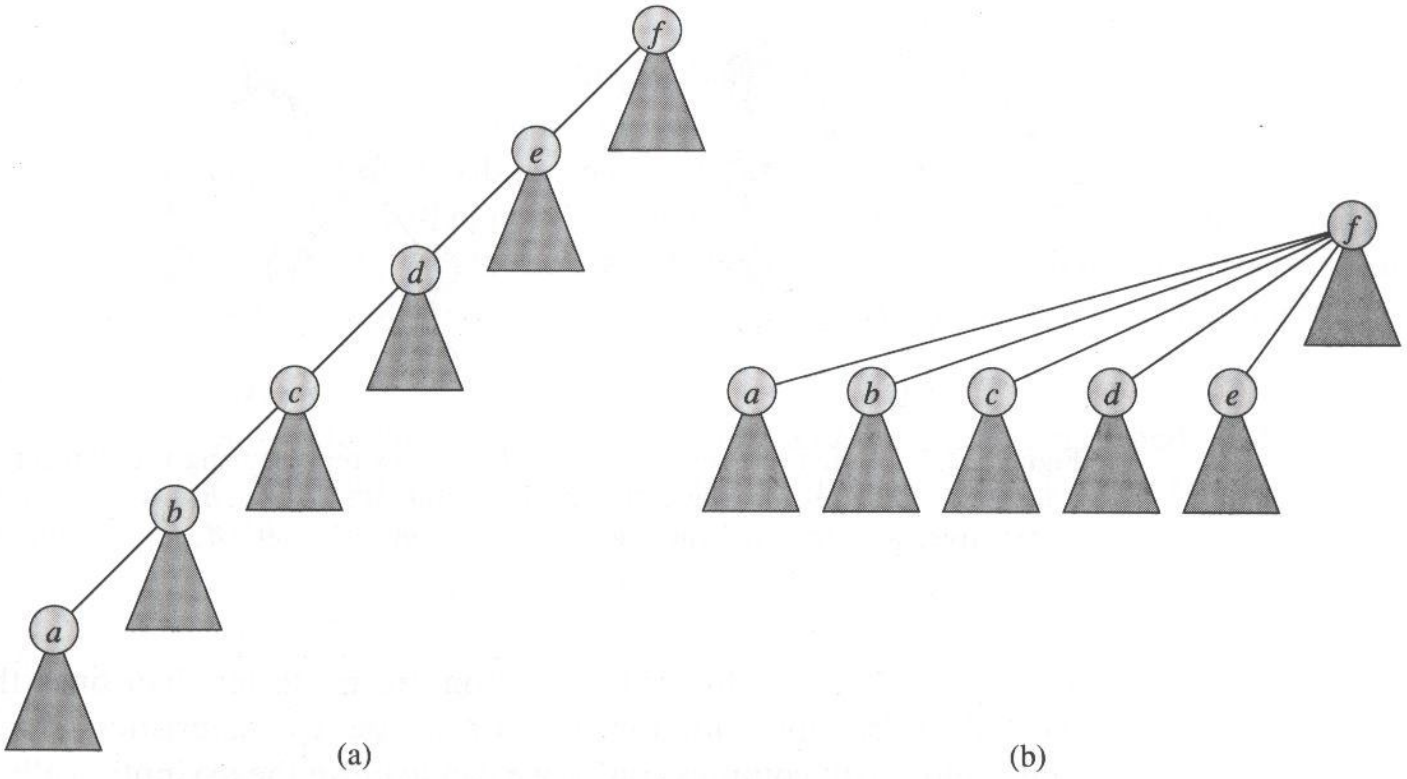
Heuristic nén đường dẫn

– Heuristic *nén đường dẫn* (*path compression*). Chạy qua hai giai đoạn khi thực thi FIND-SET:

- giai đoạn chạy lên để tìm gốc của cây,
- giai đoạn chạy xuống để cập nhật các nút trên đường dẫn để chúng chỉ trực tiếp đến gốc.

Heuristic nén đường dẫn (tiếp)

- Minh họa heuristic nén đường dẫn do thao tác FIND-SET
  - Các hình tam giác tượng trưng các cây con có gốc tại các nút trong hình (a). Mỗi nút có con trỏ chỉ đến nút cha của nó.
  - Hình (b): sau khi thực thi FIND-SET(a)



Các heuristic hợp theo thứ hạng và nén đường dẫn

- Các thủ tục hiện thực các heuristics hợp theo thứ hạng và nén đường dẫn: MAKE-SET, UNION, và FIND-SET
  - Cha của nút x là $p[x]$.

```
MAKE-SET(x)
1   p[x] ← x
2   rank[x] ← 0

UNION(x, y)
1   LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1   if rank[x] > rank[y]
2       then p[y] ← x
3       else  p[x] ← y
4             if rank[x] = rank[y]
5                then rank[y] ← rank[y] + 1
```

Các heuristics hợp theo thứ hạng và nén đường dẫn (tiếp)

```
FIND-SET(x)
1  if x ≠ p[x]
2     then p[x] ← FIND-SET(p[x])
3  return p[x]
```

Ảnh hưởng của các heuristics lên thời gian chạy

- Thời gian chạy của một dãy các thao tác gồm m MAKE-SET, UNION, và FIND-SET, trong đó có n thao tác MAKE-SET:
  - Nếu chỉ dùng heuristic hợp theo thứ hạng
    - $O(m \lg n)$
  - Nếu chỉ dùng heuristic nén đường dẩn, với f là số thao tác FIND-SET
    - $\Theta(f \log_{(1+f/n)} n)$ nếu $f \geq n$
    - $\Theta(n + f \lg n)$ nếu $f < n$
  - Nếu dùng cả hai heuristics hợp theo thứ hạng và nén đường dẩn
    - $O(m\, \alpha(m, n))$
      - $\alpha(m, n)$ là hàm đảo của hàm Ackermann
      - trong mọi ứng dụng thực tế, $\alpha(m, n) \leq 4$.

- (Không chứng minh các chận đã liệt kê ở trên.)

Giải thuật tìm kiếm trong đồ thị

Biểu diễn các đồ thị

- Hai cách biểu diễn một đồ thị $G = (V, E)$:
  - Biểu diễn *danh sách kề* (adjacency list)
    - mảng *Adj* gồm $|V|$ danh sách, 1 danh sách cho mỗi đỉnh trong V.
    - $\forall u \in V$, $Adj[u]$ chứa tất cả các đỉnh v (hoặc các con trỏ đến chúng) sao cho $(u, v) \in E$.
  - Nhận xét
    - Biểu diễn danh sách kề cần $\Theta(V + E)$ memory. (Để đơn giản, ký hiệu V và E thay vì $|V|$ và $|E|$.)

Biểu diễn các đồ thị

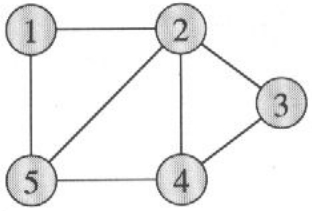
(tiếp)

- Biểu diễn *ma trận kề* (adjacency matrix)
  - Đánh số đỉnh 1, 2,..., $|V|$
  - $A = (a_{ij})$,ma trận $|V| \times |V|$

$$\begin{cases} a_{ij} = 1 & \text{nếu } (i, j) \in E \\ 0 & \text{trong các trường hợp còn lại.} \end{cases}$$

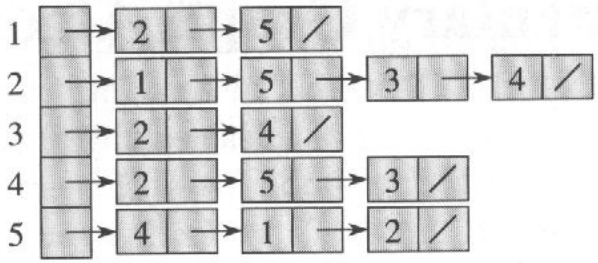
- <u>Nhận xét</u>
  - Biểu diễn ma trận kề cần $\Theta(V^2)$ memory.
  - Đồ thị thưa (sparse), $|E| < |V|^2$: nên dùng *danh sách kề* .
  - đồ thị đặc (dense), $|E| \approx |V|^2$: nên dùng *ma trận kề* .

Biểu diễn một đồ thị vô hướng



(a)

Một đồ thị vô hướng



(b)

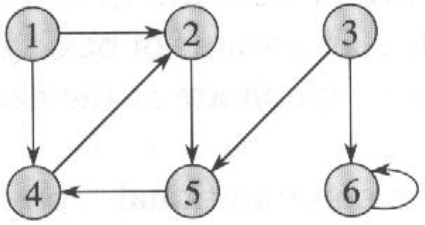
Biểu diễn
bằng một danh sách kề

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

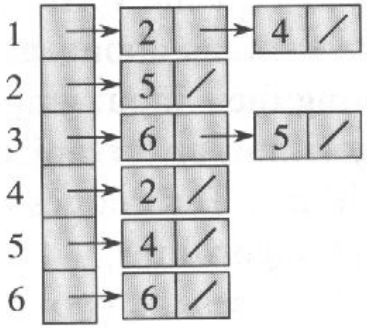
Biểu diễn
bằng một ma trận kề

Biểu diễn một đồ thị có hướng



(a)

Một đồ thị có hướng



(b)

Biểu diễn bằng một danh sách kề

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Biểu diễn bằng một ma trận kề

Tìm kiếm theo chiều rộng

Tìm kiếm theo chiều rộng (breadth-first-search, BFS)

- Một đồ thị $G = (V, E)$
  - một *đỉnh nguồn* s
  - biểu diễn bằng danh sách kề
- Mỗi đỉnh $u \in V$
  - $color[u]$: WHITE, GREY, BLACK
  - $\pi[u]$: con trỏ chỉ đến đỉnh *cha mẹ* (predecessor hay parent) của u nếu có.
  - $d[u]$: khoảng cách từ s đến u mà giải thuật tính được.
- first-in first-out queue Q
  - $head[Q]$
  - thao tác ENQUEUE(Q, v)
  - thao tác DEQUEUE(Q).

Tìm kiếm theo chiều rộng

(tiếp)

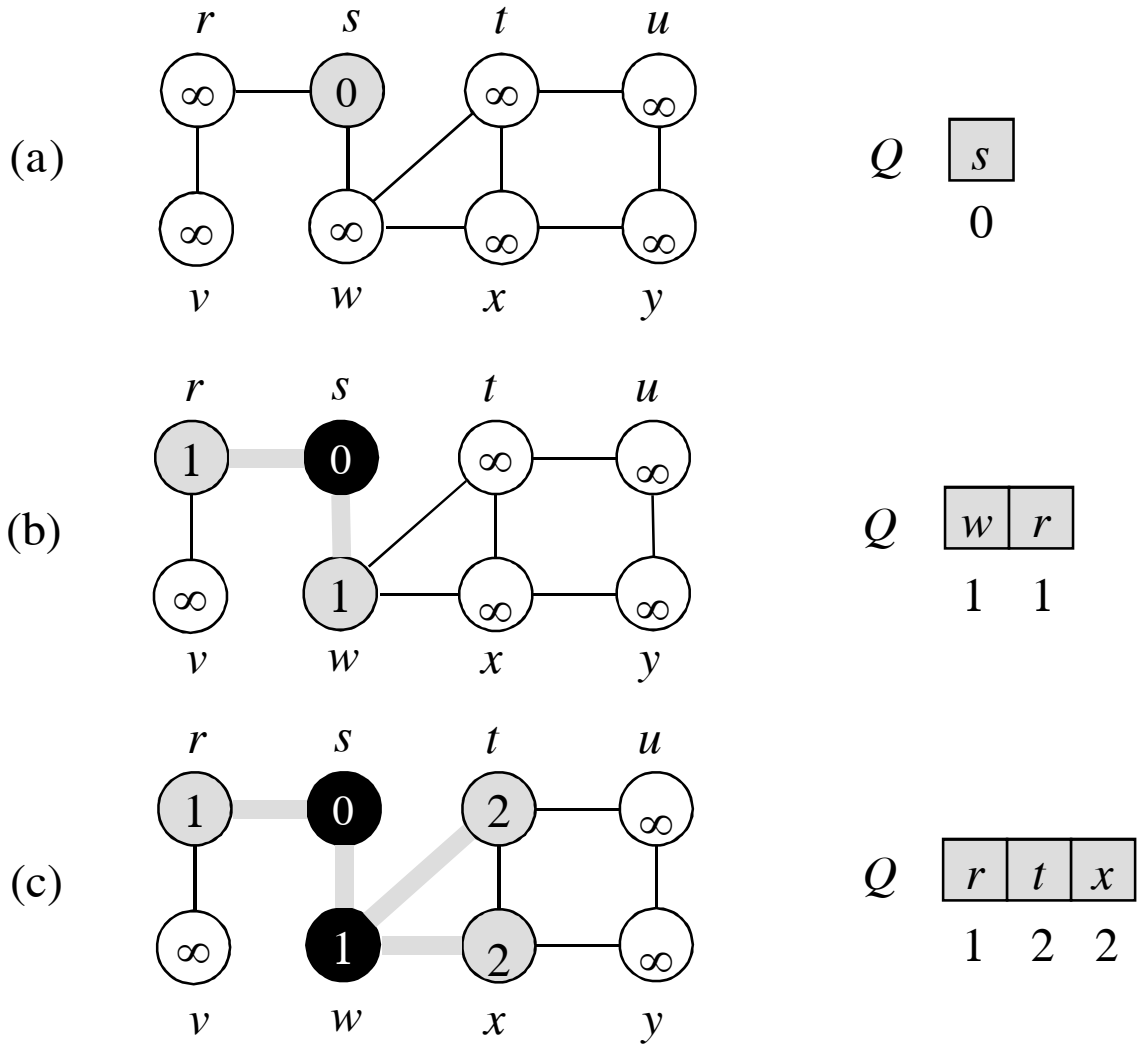
```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
```

Tìm kiếm theo chiều rộng

(tiếp)

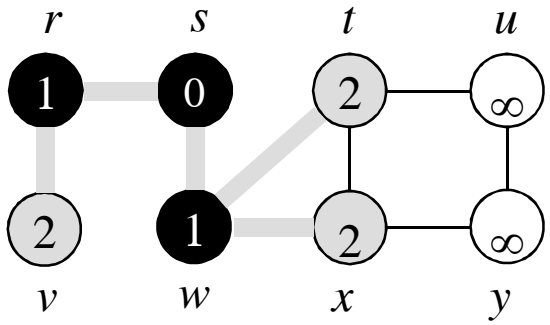
```
 8  Q ← {s}
 9  while Q ≠ ∅
10       do u ← head[Q]
11          for each v ∈ Adj[u]
12               do if color[v] = WHITE
13                    then color[v] ←  GRAY
14                         d[v] ← d[u] + 1
15                         π[v] ← u
16                         ENQUEUE(Q, v)
17          DEQUEUE(Q)
18          color[u] ← BLACK
```

Thao tác của BFS lên một đồ thị vô hướng -- Ví dụ



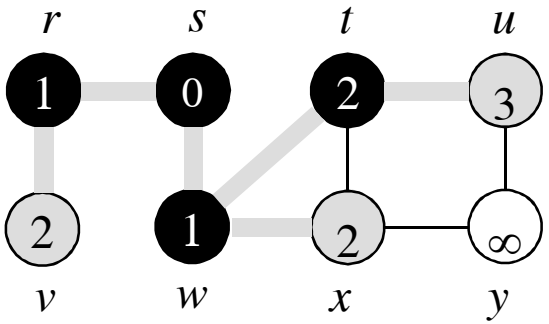
Thao tác của BFS lên một đồ thị vô hướng -- Ví dụ (tiếp)

(d)



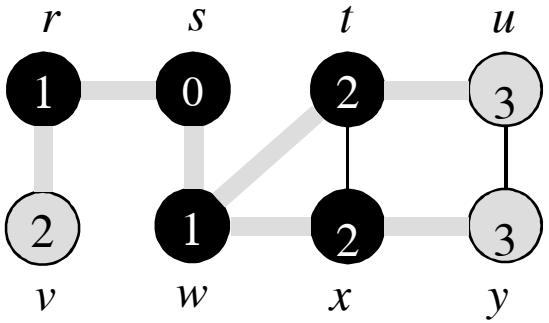
Q | t | x | v |
2 2 2

(e)



Q | x | v | u |
2 2 3

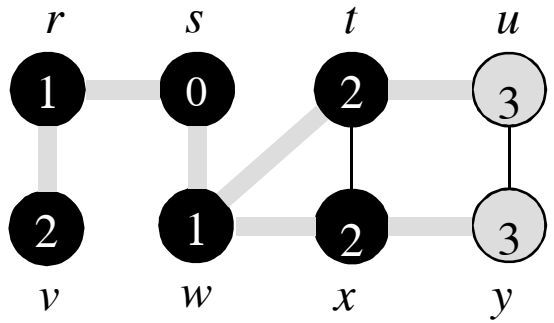
(f)



Q | v | u | y |
2 3 3

Thao tác của BFS lên một đồ thị vô hướng -- Ví dụ (tiếp)

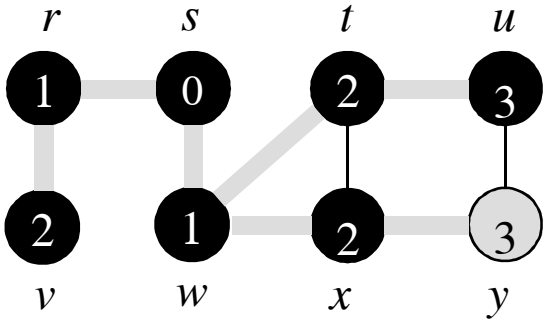
(g)



Q | u | y

3 3

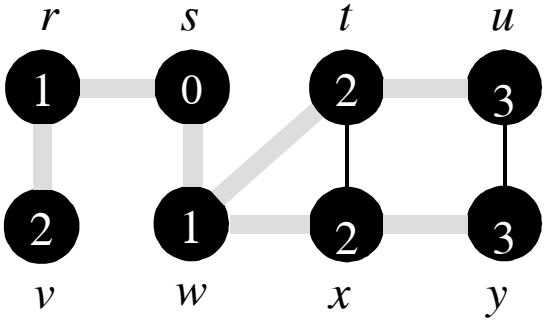
(h)



Q | y

3

(i)



Q $\varnothing$

Phân tích BFS

- Thời gian chạy của BFS là $O(V + E)$ vì
  - mỗi đỉnh của đồ thị được sơn trắng đúng một lần, do đó
    - mỗi đỉnh được enqueued nhiều lắm là một lần (màu đỉnh $\rightarrow$ xám) và dequeued nhiều lắm là một lần (màu đỉnh $\rightarrow$ đen). Mỗi thao tác enqueue hay dequeue tốn $O(1)$ thời gian nên các thao tác lên queue tốn $O(V)$ thời gian.
    - Danh sách kề của mỗi đỉnh được duyệt chỉ khi đỉnh được dequeued nên nó được duyệt nhiều lắm là một lần. Vì chiều dài tổng cộng của các danh sách kề là $\Theta(E)$ nên thời gian để duyệt các danh sách kề là $O(E)$.

Đường đi ngắn nhất

Định nghĩa

- *Khoảng cách đường đi ngắn nhất* $\delta(s, v)$ (*shortest path distance*) từ s đến v
  - là số cạnh tối thiểu lấy trong mọi đường đi từ s đến v, nếu có đường đi từ s đến v.
  - là ∞ nếu không có đường đi từ s đến v.
- Một *đường đi ngắn nhất* (*shortest path*) từ s đến v là một đường đi từ s đến v có chiều dài là $\delta(s, v)$.

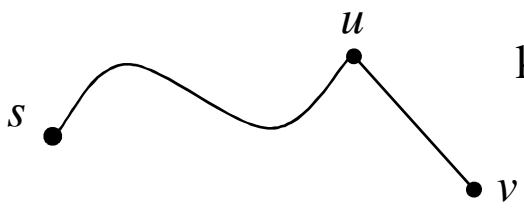
Đường đi ngắn nhất

Lemma 23.1

- $G = (V, E)$ là một đồ thị hữu hướng hay vô hướng,
- một đỉnh $s \in V$ bất kỳ

$\Rightarrow$ đối với một cạnh bất kỳ $(u, v) \in E$, ta có $\delta(s, v) \leq \delta(s, u) + 1$.

Chứng minh



khi u đến được từ s

– Nếu u đến được từ s thì v cũng đến được từ s. Đường đi từ s đến v gồm một đường đi ngắn nhất từ s đến u và cạnh (u,v) có chiều dài là $\delta(s, u) + 1$. Dĩ nhiên là $\delta(s, v) \leq \delta(s, u) + 1$.

– Nếu u không đến được từ s thì $\delta(s, u) = \infty$, vậy bất đẳng thức cũng đúng trong trường hợp này.

Đường đi ngắn nhất

Lemma 23.2

- $G = (V, E)$ là một đồ thị hữu hướng hay vô hướng.
- Chạy BFS lên G từ một đỉnh nguồn s.

$\Rightarrow$ khi BFS xong, có $d[v] \geq \delta(s, v)$ tại mọi đỉnh v.

Chứng minh

♣ "Inductive hypothesis": với mọi $v \in V$, sau mỗi lần một đỉnh nào đó được đưa vào queue thì $d[v] \geq \delta(s, v)$.

– "Basis": sau khi s được đưa vào Q. Kiểm tra ♣ là đúng.

– "Inductive step": xét một đỉnh trắng v được tìm thấy khi thăm dò từ một đỉnh u. Từ ♣ có $d[u] \geq \delta(s, u)$.

- $d[v] = d[u] + 1$, dòng 14
- $\geq \delta(s, u) + 1$
- $\geq \delta(s, v)$, Lemma 23.1

- Sau đó v được đưa vào Q và $d[v]$ không thay đổi nữa. Vậy ♣ được duy trì.

Đường đi ngắn nhất

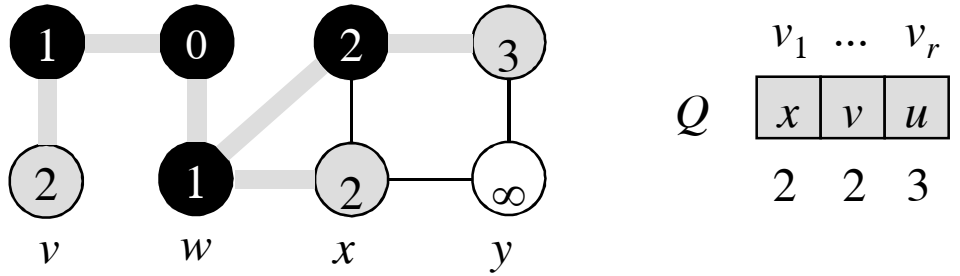
Lemma 23.3

- chạy BFS lên một đồ thị $G = (V, E)$
- trong khi thực thi BFS, queue Q chứa các đỉnh $\langle v_1, v_2, \ldots, v_r \rangle$, với v_1 là đầu và v_r là đuôi của Q.

$\Rightarrow$

- $d[v_r] \le d[v_1] + 1$,
- $d[v_i] \le d[v_{i+1}]$, với $i = 1, 2, \ldots, r - 1$.

- Ví dụ



Đường đi ngắn nhất

Chứng minh

- "Induction lên số các thao tác lên queue"

  ♣ "Inductive hypothesis": (xem Lemma) sau mỗi thao tác lên queue.

  – "Basis": queue chỉ chứa *s*. Kiểm tra Lemma là đúng.

Đường đi ngắn nhất

Chứng minh (tiếp theo)

- "Inductive step"
  - Sau khi dequeue một đỉnh bất kỳ. Kiểm tra Lemma là đúng dùng inductive hypothesis:
    - dequeue v_1, đầu mới của queue là v_2
    - $d[v_r] \le d[v_1] + 1 \le d[v_2] + 1$
    - các bất đẳng thức còn lại không bị ảnh hưởng tới.
  - Sau khi enqueue một đỉnh bất kỳ. Kiểm tra Lemma là đúng dùng inductive hypothesis
    - enqueue v, nó thành v_{r+1} trong queue
    - xem code thấy: cạnh (u, v) đang được thăm dò với $u = v_1$, v_1 là đầu của queue, từ đó
      - $d[v_{r+1}] = d[v] = d[u] + 1 = d[v_1] + 1$,
      - $d[v_r] \le d[v_1] + 1 = d[u] + 1 = d[v] = d[v_{r+1}]$
      - các bất đẳng thức còn lại không bị ảnh hưởng tới.

Đường đi ngắn nhất

Định lý 23.4 Tính đúng đắn (correctness) của BFS

- $G = (V, E)$ là một đồ thị hữu hướng hay vô hướng
- chạy BFS lên G từ một đỉnh nguồn s

$\Rightarrow$ trong khi BFS chạy

- BFS tìm ra mọi đỉnh của G tới được từ s
- khi BFS xong, $d[v] = \delta(s, v)$ cho mọi $v \in V$
- đối với mọi đỉnh $v \neq s$ tới được từ s, một trong các đường đi ngắn nhất từ s đến v là đường đi ngắn nhất từ s đến $\pi[v]$ theo sau là cạnh $(\pi[v], v)$.

Chứng minh

- Trường hợp đỉnh v không đến được từ s:

  (a) $d[v] \geq \delta(s, v) = \infty$ (Lemma 23.2)

  (b) Dòng 14 chỉ được thực thi khi v có $d[v] < \infty$, do đó nếu không đến được v từ s thì $d[v] = \infty$ và v sẽ không được tìm thấy.

Đường đi ngắn nhất

Chứng minh (tiếp)

- Trường hợp đỉnh v đến được từ s: định nghĩa $V_k = \{v \in V : \delta(s, v) = k\}$
  - "Induction on k".
    - "Inductive hypothesis": với mọi $v \in V_k$, trong khi thực thi BFS thì có đúng một thời điểm mà
      - v được sơn xám
      - $d[v]$ được gán trị k
      - Nếu $v \neq s$ thì $\pi[v]$ được gán trị u với $u \in V_{k-1}$
      - v được đưa vào queue Q.
    - "Basis": $k = 0$, kiểm tra là đúng
    - "Inductive step"
      - Xét $v \in V_k$ bất kỳ, $k \geq 1$.
      - Có $u \in V_{k-1}$ sao cho: u là head của queue và (u, v) được thăm dò.
- Phần còn lại: ...

Cây theo chiều rộng

- Cho một đồ thị $G = (V, E)$ và một đỉnh nguồn s.
- Sau khi thực thi BFS lên G , dùng trường π trong mỗi đỉnh để định nghĩa một "cây theo chiều rộng":
  - *Đồ thị các đỉnh cha mẹ* (*predecessor subgraph*) của G là đồ thị $G_\pi = (V_\pi , E_\pi)$ với
    - $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$
    - $E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$

  Định nghĩa: G_π là một *cây theo chiều rộng* nếu
  - V_π gồm các đỉnh trong V đến được từ s , và
  - có một đường đi đơn duy nhất từ s đến v cho mọi $v \in V_\pi$, đây cũng là đường đi ngắn nhất từ s đến v trong G.
- Nhận xét:
  - Một cây theo chiều rộng là một cây.
  - Các cạnh trong E_π được gọi là các *cạnh cây* (*tree edges*).

Cây tìm kiếm theo chiều rộng

Lemma 23.5

Khi BFS chạy trên đồ thị vô hướng hay hữu hướng $G = (V, E)$ thì nó sẽ xây dựng π sao cho G_π là cây theo chiều rộng.

Chứng minh

- V_π gồm các đỉnh trong V đến được từ s: đó là vì trong dòng 15 của BFS, gán $\pi[v] = u$ nếu $(u, v) \in E$ và $\delta(s, v) < \infty$, tức là v đến được từ s.
- Có đường đơn duy nhất từ s đến v cho mọi $v \in V_\pi$, đây cũng là đường đi ngắn nhất từ s đến v trong G: đó là vì G_π là một cây nên tồn tại đường đi đơn duy nhất trong G_π từ s đến mỗi đỉnh v trong V_π. Theo định lý 23.4 đường đi đơn duy nhất này là đường đi ngắn nhất từ s đến v.

Tìm kiếm theo chiều sâu

Tìm kiếm theo chiều sâu (depth-first-search, DFS)

- Cho một đồ thị $G = (V, E)$
- Sau khi thực thi DFS lên G , dùng trường π trong mỗi đỉnh để định nghĩa một "cây theo chiều sâu":
  - *Đồ thị các đỉnh cha mẹ* (*predecessor subgraph*) do tìm kiếm theo chiều sâu là $G_\pi = (V, E_\pi)$ với
    - $E_\pi = \{(\pi[v], v) : v \in V \text{ và } \pi[v] \neq \text{NIL}\}$
  - Predecessor subgraph do tìm kiếm theo chiều sâu tạo nên một *rừng theo chiều sâu*, gồm nhiều *cây theo chiều sâu*.
  - Các cạnh trong E_π được gọi là các *cạnh cây*.

Tìm kiếm theo chiều sâu

- Trong khi tìm kiếm, các đỉnh được tô màu để chỉ ra trạng thái của nó
  - khởi đầu: màu trắng
  - *được tìm ra* (*discovered*): màu xám
  - *hoàn tất*, *xong* (*finished*): màu đen
  - Mỗi đỉnh v được ghi giờ (*timestamp)*, có hai timestamps
    - $d[v]$: (*d*iscovered) đỉnh v được tìm ra, sơn v xám
    - $f[v]$: (*f*inished) hoàn tất việc thăm dò từ đỉnh v, sơn v đen.

Tìm kiếm theo chiều sâu

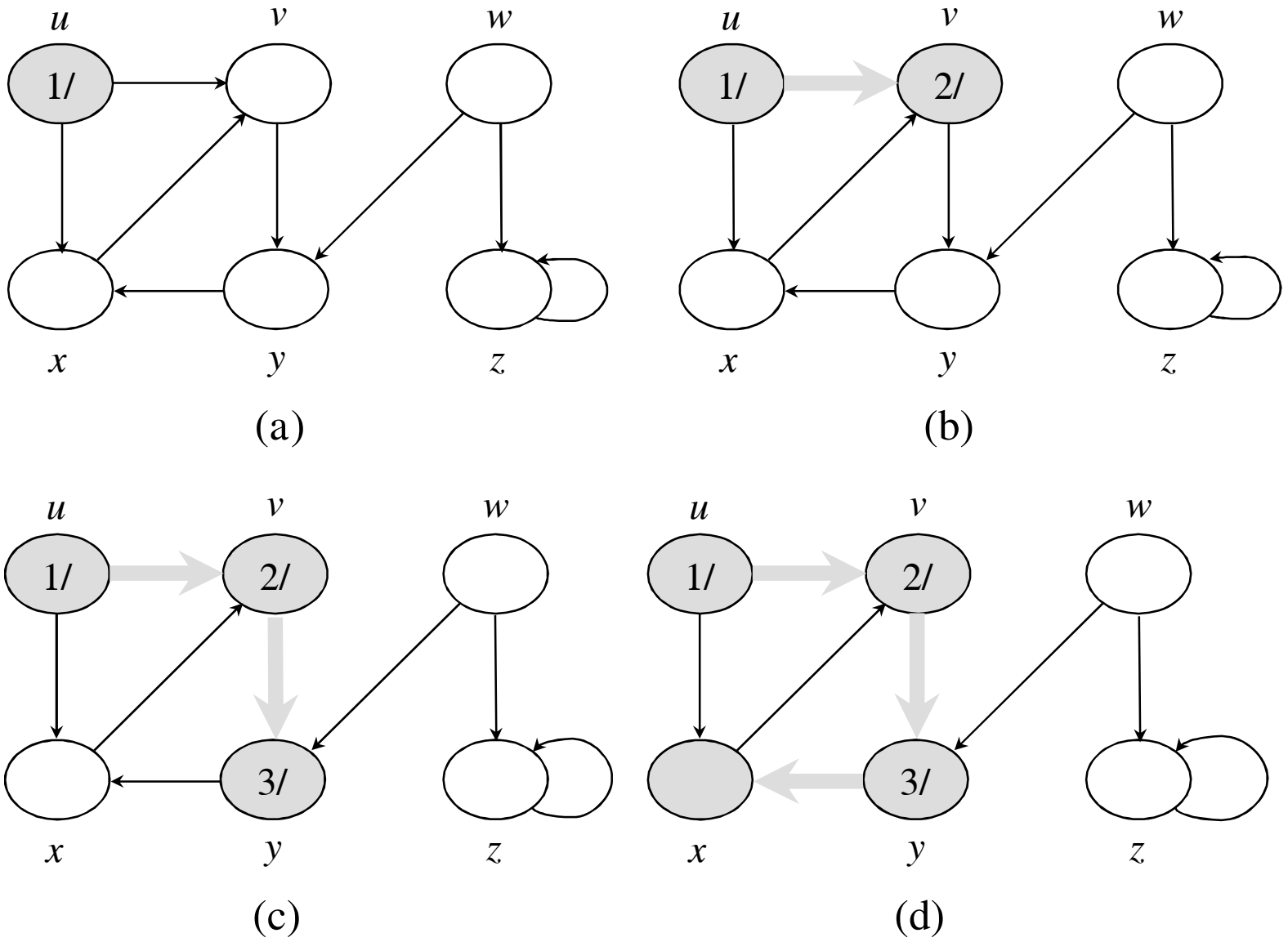
- Một đồ thị $G = (V, E)$ vô hướng hay có hướng
  - biểu diễn dùng danh sách kề
  - biến toàn cục *time*: dùng cho timestamp
- Mỗi $u \in V$
  - *color*[u]: WHITE, GREY, BLACK
  - $d[u]$: thời điểm đỉnh u được tìm ra
  - $f[u]$: thời điểm hoàn tất thăm dò từ đỉnh u
  - $\pi[u]$: con trỏ chỉ đến cha mẹ của u.

Tìm kiếm theo chiều sâu

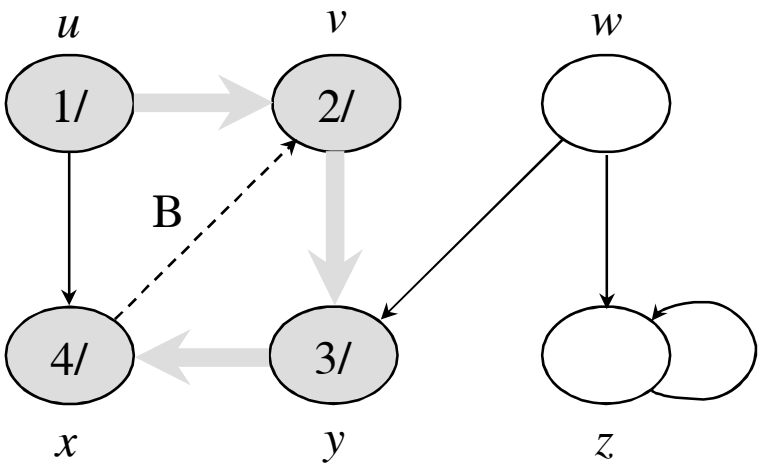
```
DFS(G)
1   for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4   time ← 0
5   for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7          then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1   color[u] ← GRAY
2   d[u] ← time ← time + 1
3   for each v ∈ Adj[u]
4       do if color[v] = WHITE
5          then π[v] ← u
6               DFS-VISIT(v)
7   color[u] ← BLACK
8   f[u] ← time ← time + 1
```

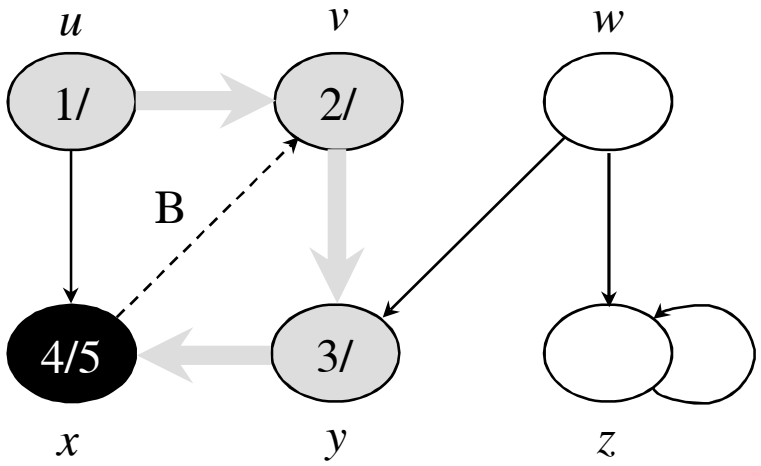
Thao tác của DFS lên đồ thị -- Ví dụ



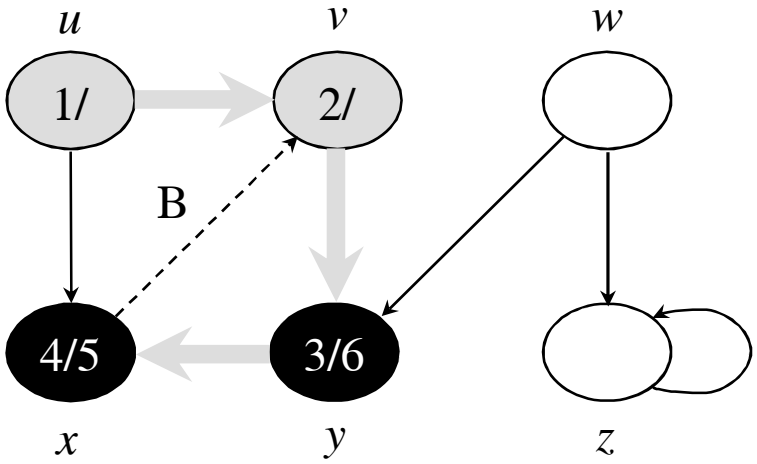
Thao tác của DFS lên đồ thị -- Ví dụ (tiếp theo)



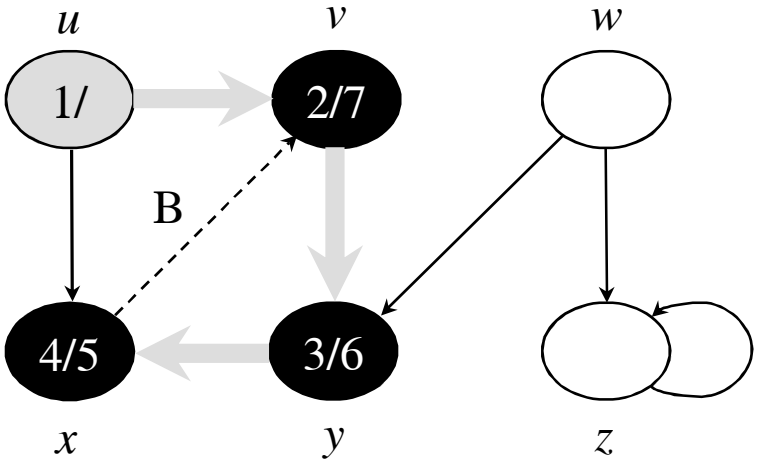
(e)



(f)

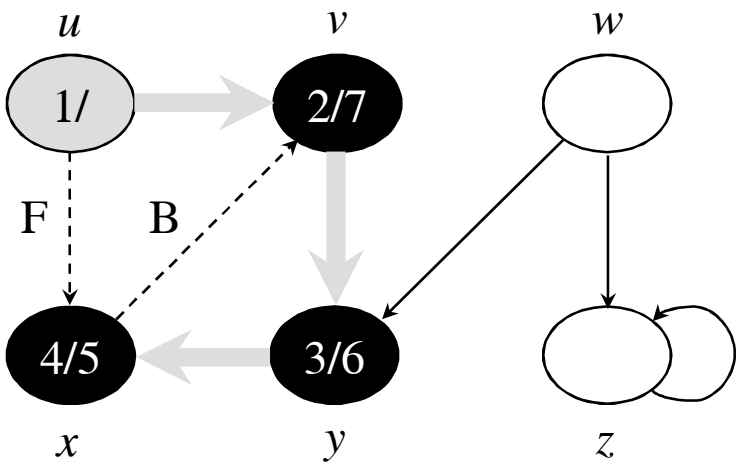


(g)

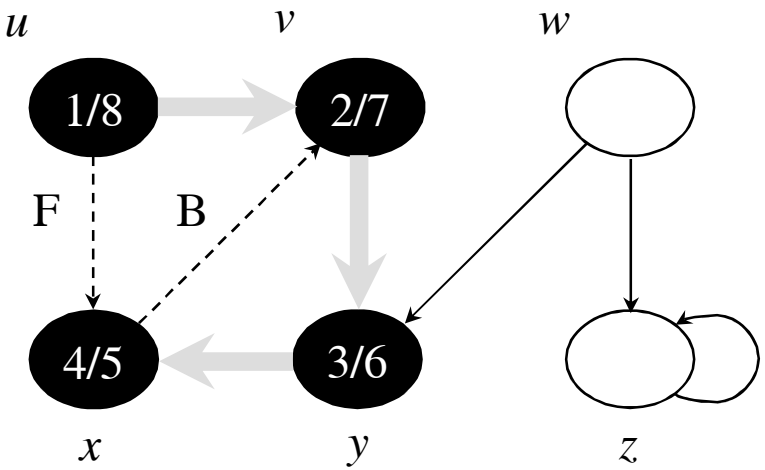


(h)

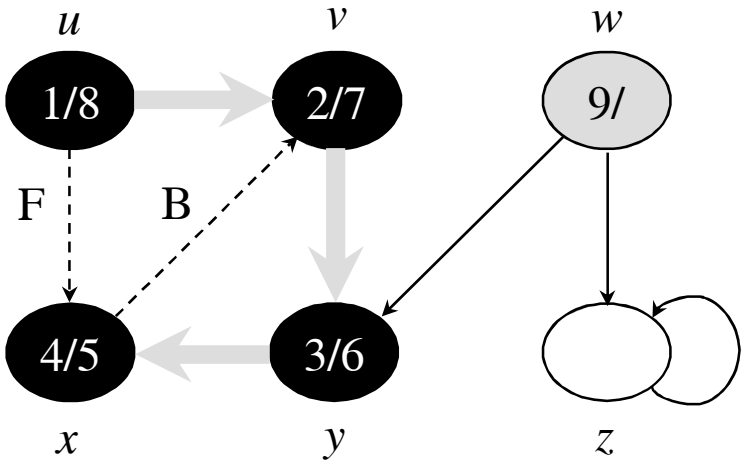
Thao tác của DFS lên đồ thị -- Ví dụ (tiếp theo)



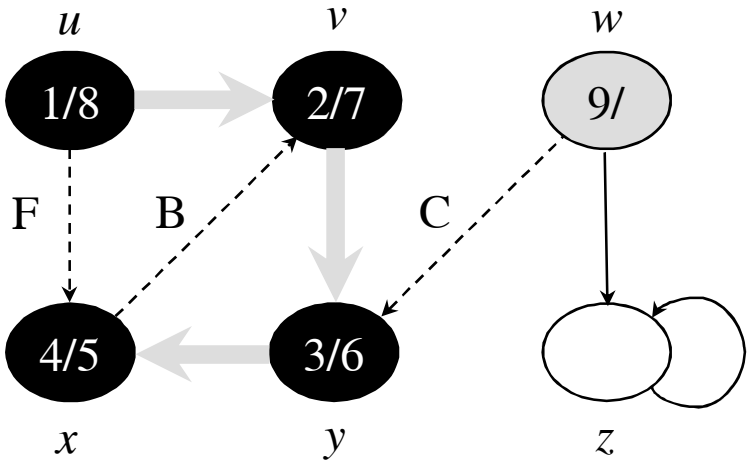
(i)



(j)

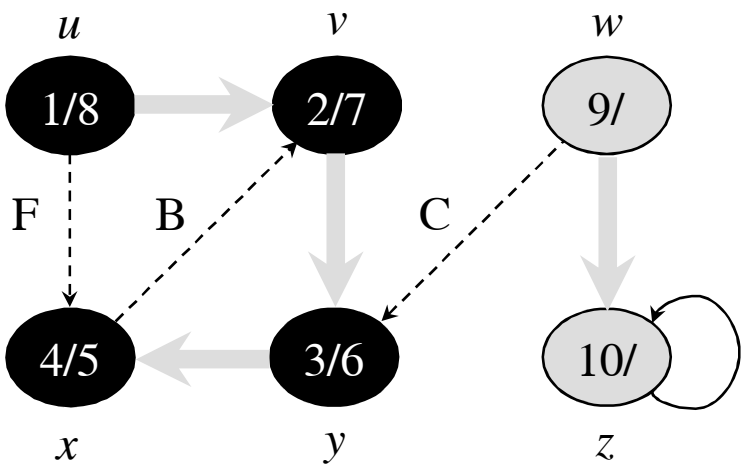


(k)

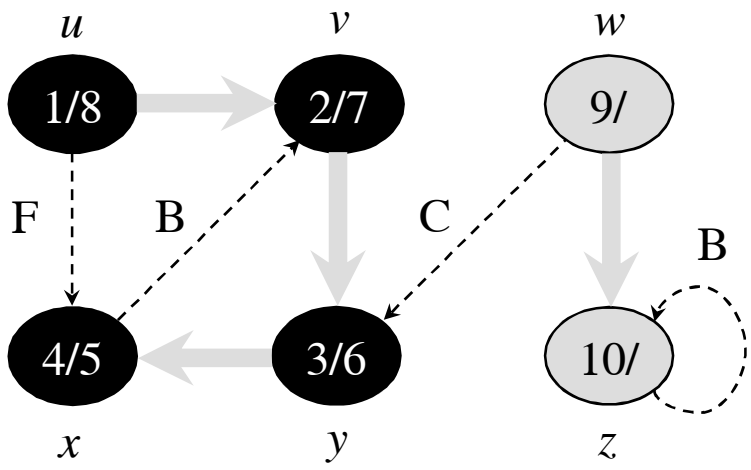


(l)

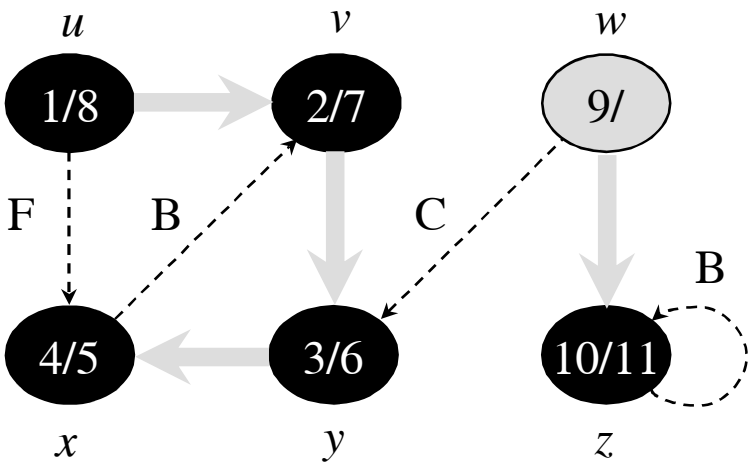
Thao tác của DFS lên đồ thị -- Ví dụ (tiếp theo)



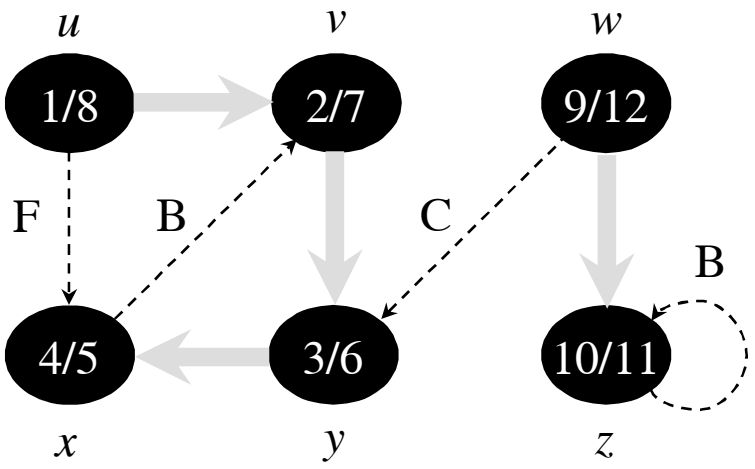
(m)



(n)



(o)



(p)

Phân tích DFS

- Thời gian chạy của DFS là $\Theta(V + E)$ vì
  - Các vòng lặp trong DFS cần $\Theta(V)$ thời gian, chưa kể thời gian thực thi các lần gọi DFS-VISIT.
  - DFS-VISIT được gọi đúng một lần cho mỗi đỉnh v (vì ngay khi đó màu đỉnh $v \rightarrow$ xám).
    - Thực thi DFS-VISIT(v): danh sách kề của v được duyệt.

    Vậy thời gian để duyệt tất cả các danh sách kề là $\Theta(E)$.

Đặc tính của tìm kiếm theo chiều sâu

Định lý 23.6 Định lý dấu ngoặc, Parenthesis theorem

Trong mọi tìm kiếm theo chiều sâu của một đồ thị hữu hướng hay vô hướng $G = (V, E)$, đối với mọi cặp đỉnh u và v, chỉ một trong ba điều sau là đúng

- các khoảng $[d[u], f[u]]$ và $[d[v], f[v]]$ là rời nhau,
- khoảng $[d[u], f[u]]$ hoàn toàn nằm trong khoảng $[d[v], f[v]]$, và u là một hậu duệ của v trong cây theo chiều sâu,
- khoảng $[d[v], f[v]]$ hoàn toàn nằm trong khoảng $[d[u], f[u]]$, và v là một hậu duệ của u trong cây theo chiều sâu.

Chứng minh

- Trường hợp $d[u] < d[v]$: xét hai trường hợp con
  - $d[v] < f[u]$. Đỉnh v được tìm ra trong khi u còn là xám, vậy v là một hậu duệ của u. Hơn nữa vì v được tìm ra sau u, nên một khi mọi cạnh từ v được thăm dò xong thì v hoàn tất, trước khi việc tìm

Đặc tính của tìm kiếm theo chiều sâu

Chứng minh (tiếp)

kiếm quay về u và hoàn tất u, do đó $f[v] < f[u]$. Tổng kết:

$d[u] < d[v] < f[v] < f[u]$, tức là khoảng $[d[v], f[v]]$ hoàn toàn nằm trong khoảng $[d[u], f[u]]$.

- $f[u] < d[v]$. Hơn nữa, vì $d[u] < f[u]$ và $d[v] < f[v]$ nên $d[u] < f[u] < d[v] < f[v]$, tức là các khoảng $[d[u], f[u]]$ và $[d[v], f[v]]$ là rời nhau.

Trường hợp $d[v] < d[u]$. Tương tự.

Đặc tính của tìm kiếm theo chiều sâu

Định lý 23.8 Định lý white-path

Cho một đồ thị vô hướng hay có hướng $G = (V, E)$.

- Trong rừng theo chiều sâu của G, đỉnh v là một hậu duệ của đỉnh u
- $\Leftrightarrow$ vào thời điểm $d[u]$ khi DFS tìm ra u, đỉnh v có thể đến được từ u theo một đường đi chỉ gồm các đỉnh màu trắng.

- ***Chứng minh***
- $\Rightarrow$: Giả sử v là một hậu duệ của đỉnh u. Gọi w là một đỉnh bất kỳ nằm trên đường đi từ u đến v trong cây theo chiều sâu, thì w là một hậu duệ của u. Vậy $d[u] < d[w]$, do đó w là trắng vào lúc $d[u]$.

- $\Leftarrow$: (sketch) $d[u] < d[v] < f[v] < f[u]$.

Đặc tính của tìm kiếm theo chiều sâu

- Phân loại các cạnh của $G = (V, E)$
  - Các cạnh *cây* (*tree* edge): là các cạnh trong G_π .
  - Các cạnh *lùi* (*back* edge): là các cạnh (u, v) nối u đến một nút tổ tiên (ancestor) v trong một depth-first tree.
  - Các cạnh *tiến* (*forward* edge): là các cạnh, không phải là các cạnh cây, (u, v) nối một đỉnh u đến một hậu duệ (descendant) v trong một depth-first tree.
  - Các cạnh *xuyên* (*cross* edge): là tất cả các cạnh còn lại.

Đặc tính của tìm kiếm theo chiều sâu

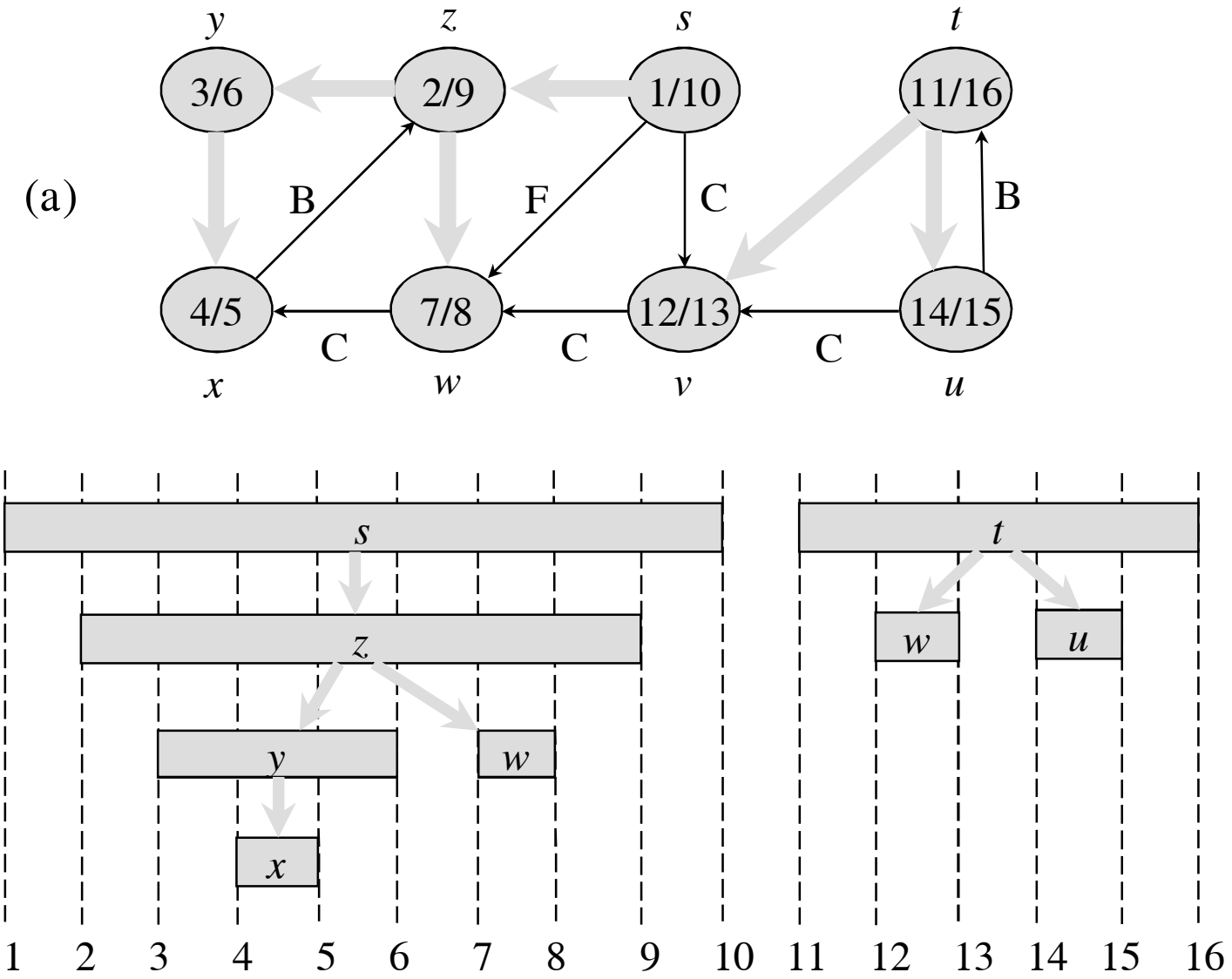
Định lý 23.9

Trong tìm kiếm theo chiều sâu của một đồ thị vô hướng G, mỗi cạnh của G hoặc là một cạnh cây hoặc là một back edge.

Chứng minh

- Xét một cạnh bất kỳ (u,v) của G. Giả sử $d[u] < d[v]$.
- v phải được hoàn tất trước u vì v nằm trong danh sách các đỉnh kề của u.
- Hai trường hợp:
  - Cạnh (u,v) được thăm dò lần đầu theo hướng từ u đến v: (u,v) là cạnh cây.
  - Cạnh (u,v) được thăm dò lần đầu theo hướng từ v đến u: (u,v) là back edge vì đỉnh u còn là xám (u hoàn tất sau v).

Các tính chất của tìm kiếm theo chiều sâu



Ứng dụng của DFS: sắp thứ tự tô pô

- Cho một đồ thị có hướng *không có chu trình* (directed acyclic graph, hay *dag*) $G = (V, E)$. Một *sắp thứ tự tôpô* của dag G là một sắp xếp tuyến tính của tất cả các đỉnh của G sao cho
  - nếu G chứa một cạnh (u, v) thì u xuất hiện trước v trong sắp xếp.
- Nhận xét

  Nếu một đồ thị có hướng có chu trình thì không sắp thứ tự tô pô cho nó được.

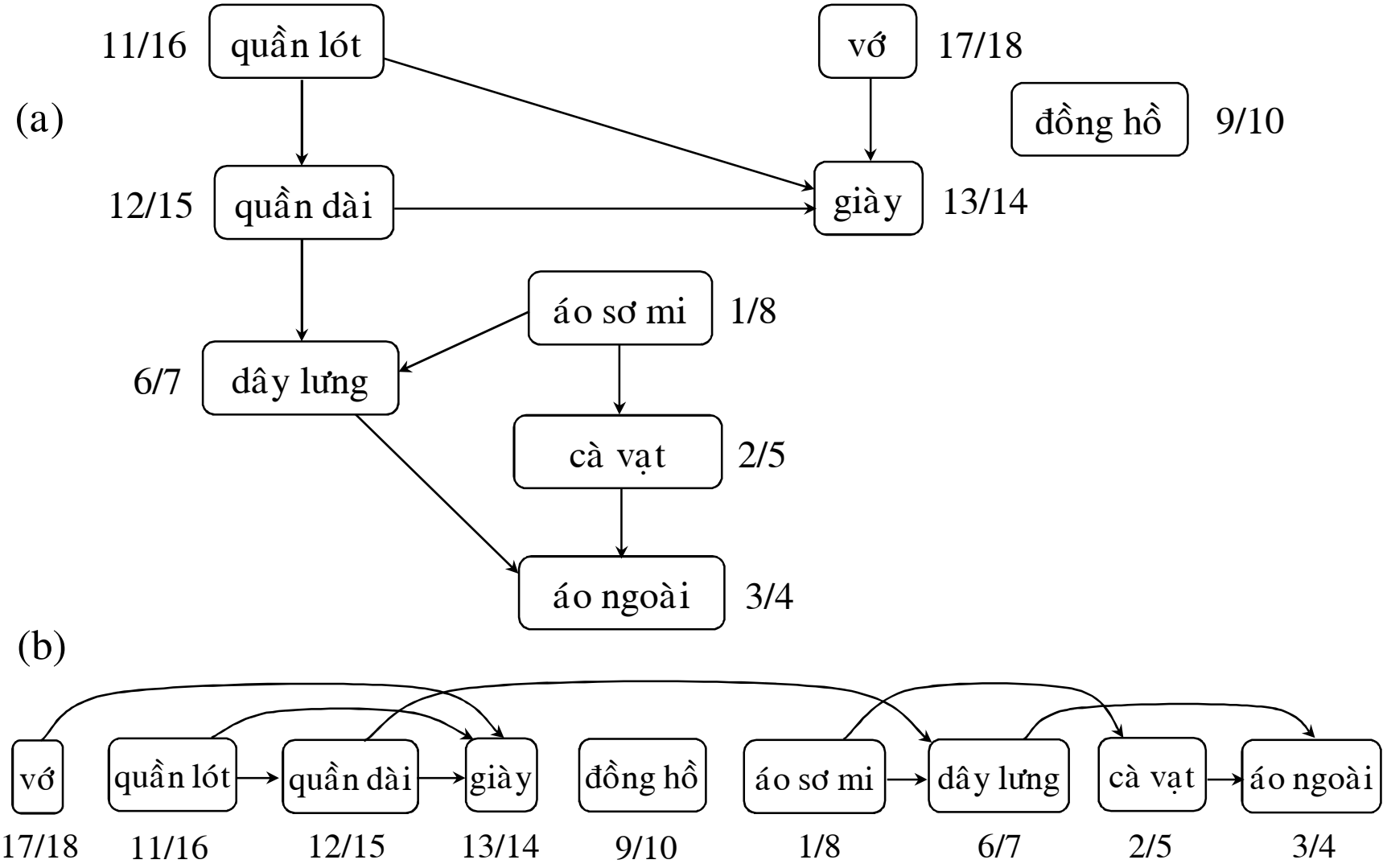
Sắp thứ tự tô pô

- Cho một dag $G = (V, E)$.

TOPOLOGICAL-SORT(G)
1 gọi DFS(G) để tính thời điểm hoàn tất $f[v]$ cho mọi đỉnh v
2 mỗi khi một đỉnh hoàn tất, chèn nó vào phía trước một danh sách liên kết
3 **return** danh sách liên kết các đỉnh

Thời gian chạy của TOPOLOGICAL-SORT là $\Theta(V + E)$.

Sắp thứ tự tô pô -- Ví dụ



Đặc tính của sắp thứ tự tô pô

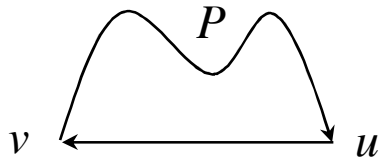
Lemma 23.10

Một đồ thị có hướng G là không có chu trình (acyclic)

$\Leftrightarrow$ một tìm kiếm theo chiều sâu của G không cho ra back edge.

Chứng minh

$\Rightarrow$:



Giả sử tìm kiếm theo chiều sâu của G cho ra back edge (u, v), với v là một tổ tiên của u. Có đường đi P trong rừng theo chiều sâu từ v đến u. Như vậy P và back edge (u, v) tạo ra một chu trình.

$\Leftarrow$: Bài tập!

Đặc tính của sắp thứ tự tô pô

Định lý 23.11

TOPOLOGICAL-SORT(G) tìm được một sắp thứ tự tôpô của một đồ thị có hướng không chứa chu trình G.

Chứng minh

- Chạy DFS lên dag $G = (V, E)$ để xác định thời điểm hoàn tất của các đỉnh.
- Cần chứng tỏ: với mọi cặp $u, v \in V$ khác nhau, nếu có một cạnh trong G từ u đến v thì $f[v] < f[u]$.

  Xét một cạnh bất kỳ (u, v) được thăm dò bởi DFS(G). Khi đó v không là xám (vì nếu như vậy thì v là tổ tiên của u, và do đó (u, v) là back edge, mâu thuẩn! dùng Lemma 23.10). Vậy v là trắng hoặc đen:
  - nếu trắng: v trở thành con cháu của u, do đó $f[v] < f[u]$
  - nếu đen: dỉ nhiên là $f[v] < f[u]$.

Cây Khung Nhỏ Nhất

Cây khung nhỏ nhất

- Cho
  - một đồ thị liên thông, vô hướng $G = (V, E)$
  - một *hàm trọng số*

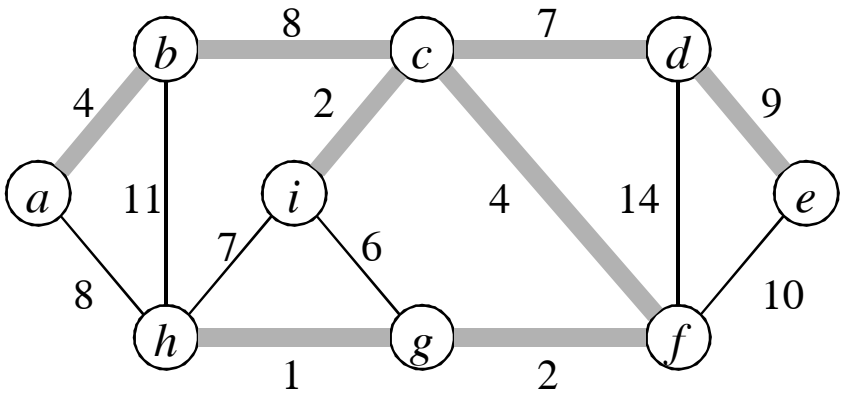
    $w : E \to \mathbf{R}$
- Tìm một tập con <u>không chứa chu trình</u> $T \subseteq E$ nối tất cả các đỉnh sao cho tổng các trọng số

  $w(T) = \sum_{(u, v) \in T} w(u, v)$
- là nhỏ nhất.
  - Tập T là một cây, và được gọi là một *cây khung nhỏ nhất*.
- *Bài toán tìm cây khung nhỏ nhất*: bài toán tìm T.

Cây khung nhỏ nhất (tiếp)

- Giải bài toán tìm cây khung nhỏ nhất
  - Giải thuật của Kruskal
  - Giải thuật của Prim.

Cây khung nhỏ nhất: ví dụ



- Tập các cạnh xám là một cây khung nhỏ nhất
- Trọng số tổng cộng của cây là 37.
- Cây là không duy nhất: nếu thay cạnh (b, c) bằng cạnh (a, h) sẽ được một cây khung khác cũng có trọng số là 37.

Cạnh an toàn

- Cho một đồ thị liên thông, vô hướng $G = (V, E)$ và một hàm trọng số $w : E \to \mathbf{R}$. Tìm một cây khung nhỏ nhất cho G!
- Giải bài toán bằng một chiến lược greedy: nuôi một cây khung lớn dần bằng cách thêm vào cây từng cạnh một.

- Định nghĩa cạnh an toàn

  Nếu A là một tập con của một cây khung nhỏ nhất nào đó, nếu (u, v) là một cạnh của G sao cho tập $A \cup \{(u, v)\}$ vẫn còn là một tập con của một cây khung nhỏ nhất nào đó, thì (u, v) là một *cạnh an toàn* cho A.

Một giải thuật tổng quát (generic)

- Một giải thuật tổng quát (generic) để tìm một cây khung nhỏ nhất
  - Input: một đồ thị liên thông, vô hướng G
    một hàm trọng số w trên các cạnh của G
  - Output: Một cây khung nhỏ nhất cho G.

```
GENERIC-MST(G, w)
1       A ← ∅
2       while A không là một cây khung nhỏ nhất
3           do tìm cạnh (u, v) an toàn cho A
4              A ← A ∪ {(u, v)}
5       return A
```

Phép cắt

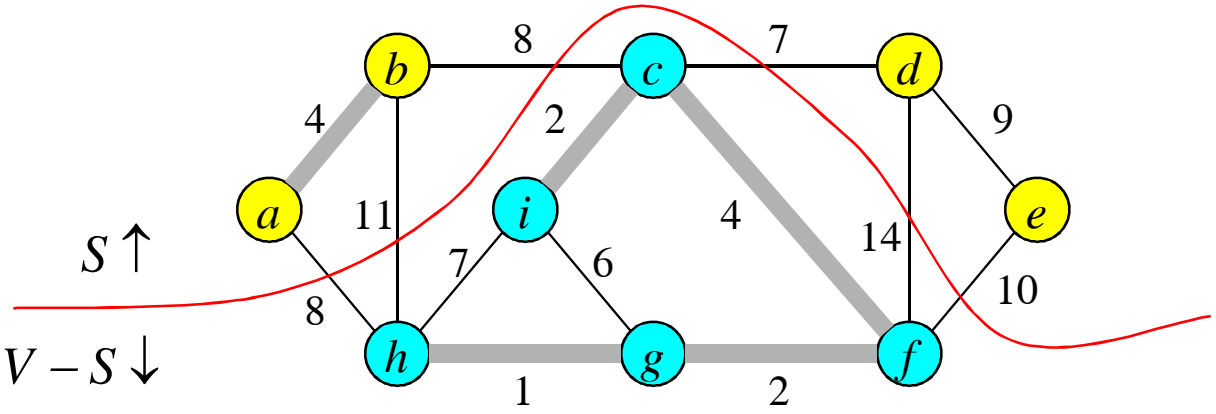
Các khái niệm quan trọng

- Một *phép cắt* $(S, V - S)$ của $G = (V, E)$ là một phân chia (partition) của V.

  Ví dụ: $S = \{a, b, d, e\}$ trong đồ thị sau.

- Một cạnh $(u, v) \in E$ *xuyên qua* (*cross*) một phép cắt $(S, V - S)$ nếu một đỉnh của nó nằm trong S và đỉnh kia nằm trong $V - S$.

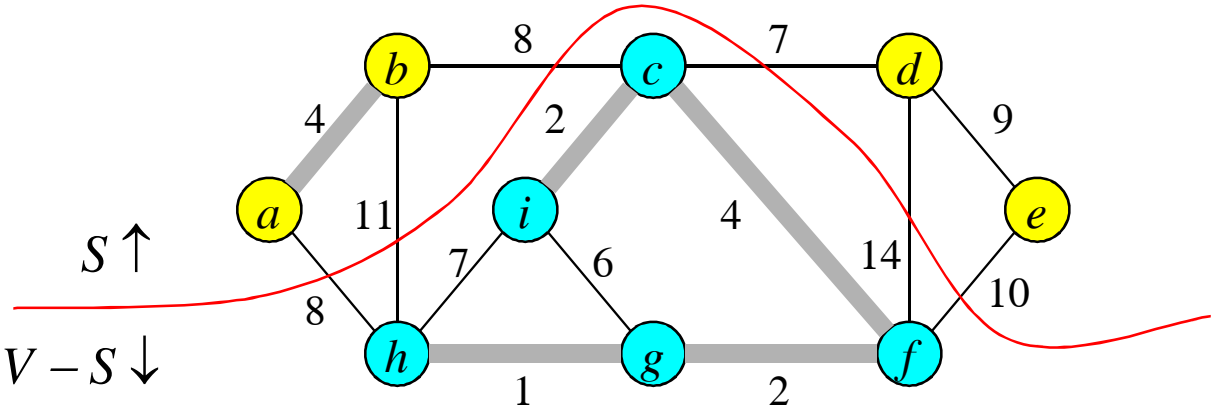
  Ví dụ: cạnh (b, c).



Cạnh nhẹ (light edge)

Các khái niệm quan trọng (tiếp)

- Một phép cắt *bảo toàn* tập các cạnh A (*respects A*) nếu không có cạnh nào của A xuyên qua phép cắt.
- Một cạnh là một *cạnh nhẹ* vượt qua phép cắt nếu trọng số của nó là nhỏ nhất trong mọi trọng số của các cạnh xuyên qua phép cắt. Ví dụ: cạnh (c, d).



Nhận ra một cạnh an toàn

Định lý 24.1

Cho

- $G = (V, E)$ là một đồ thị liên thông, vô hướng
- w là một hàm trọng số trên E
- A là một tập con của một cây khung nhỏ nhất cho G
- $(S, V - S)$ là một phép cắt bất kỳ của G bảo toàn A
- (u, v) là một cạnh nhẹ vượt qua $(S, V - S)$

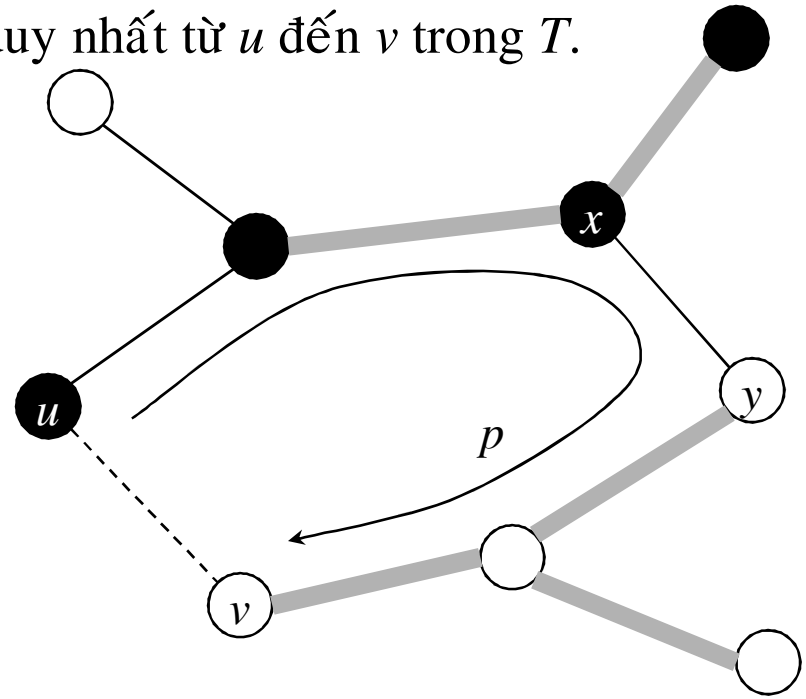
$\Rightarrow$ cạnh (u, v) là an toàn cho A.

Chứng minh

Nhận ra một cạnh an toàn

(tiếp)

- S: tập các đỉnh đen, $V - S$: tập các đỉnh trắng
- Các cạnh của một cây khung nhỏ nhất T được vẽ ra trong hình, còn các cạnh của G thì không
- A: tập các cạnh xám
- Cạnh (u, v) là cạnh nhẹ xuyên qua phép cắt $(S, V - S)$.
- p là đường đi duy nhất từ u đến v trong T.



Nhận ra một cạnh an toàn

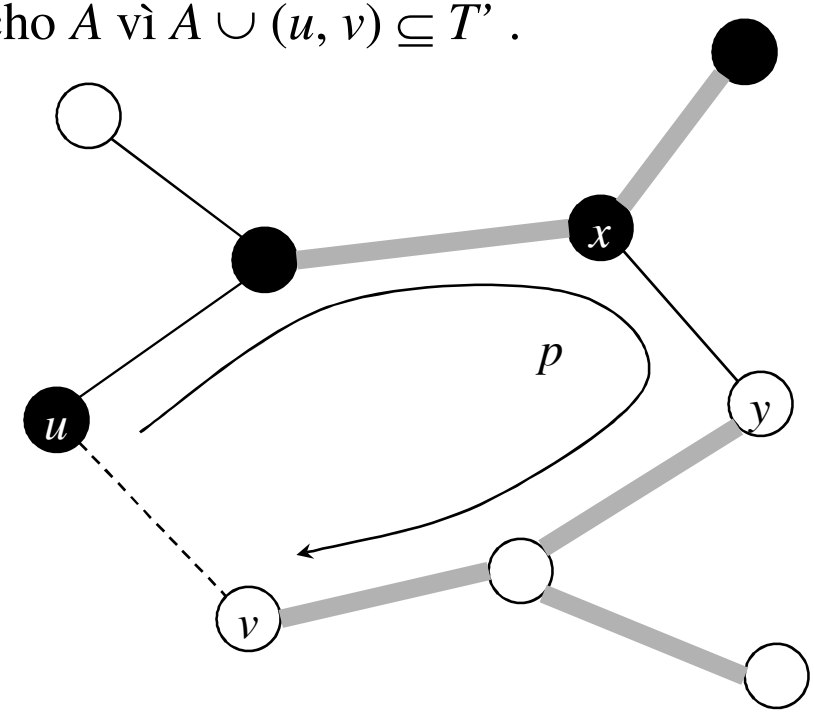
(tiếp)

- Định nghĩa cây khung $T' = T - (x, y) \cup (u, v)$

  T' là cây khung nhỏ nhất vì

  $$w(T') = w(T) - w(x, y) + w(u, v)$$
  $$\leq w(T), \qquad \text{vì } w(u, v) \leq w(x, y)$$

- (u, v) là an toàn cho A vì $A \cup (u, v) \subseteq T'$.



Nhận ra một cạnh an toàn (tiếp)

Hệ luận 24.2

Cho

- $G = (V, E)$ là một đồ thị liên thông, vô hướng với một hàm trọng số w trên E
- A là một tập con của E sao cho A nằm trong một cây khung nhỏ nhất cho G
- $C = (V_C, E_C)$ là một thành phần liên thông (cây) trong rừng $G_A = (V, A)$.

Thì,

nếu (u, v) là một cạnh nhẹ nối C với một thành phần khác trong G_A

$\Rightarrow (u, v)$ là an toàn cho A.

Chứng minh

Phép cắt $(V_C, V - V_C)$ bảo toàn A, do đó (u, v) là một cạnh nhẹ đối với phép cắt này.

Giải thuật của Kruskal

- Giải thuật của Kruskal
  - dựa trên giải thuật GENERIC-MST, mà A ban đầu là một rừng mà mỗi cây chỉ chứa một đỉnh của G.

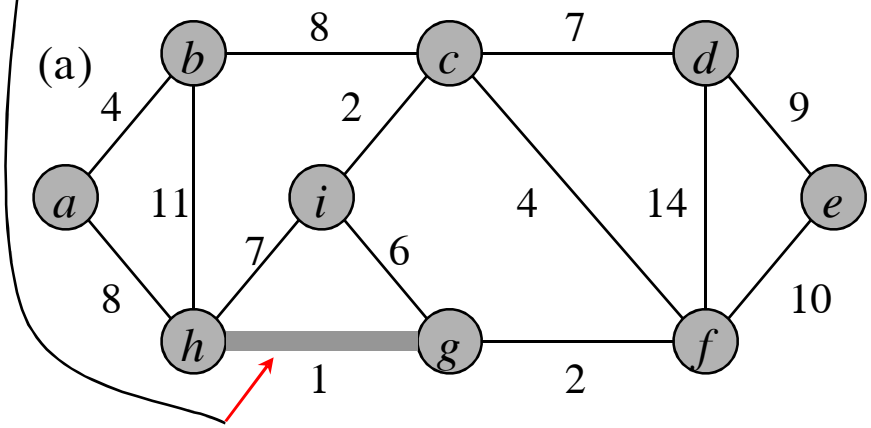
```
MST-KRUSKAL(G, w)
1       A ← ∅
2       for mỗi đỉnh v ∈ V[G]
3           do MAKE-SET(v)
4       xếp các cạnh ∈ E theo thứ tự trọng số w không giảm
5       for mỗi cạnh (u, v) ∈ E, theo thứ tự trọng số không giảm
6           do if FIND-SET(u) ≠ FIND-SET(v)
7              then A ← A ∪ {(u, v)}
8                   UNION(u, v)
9       return A
```

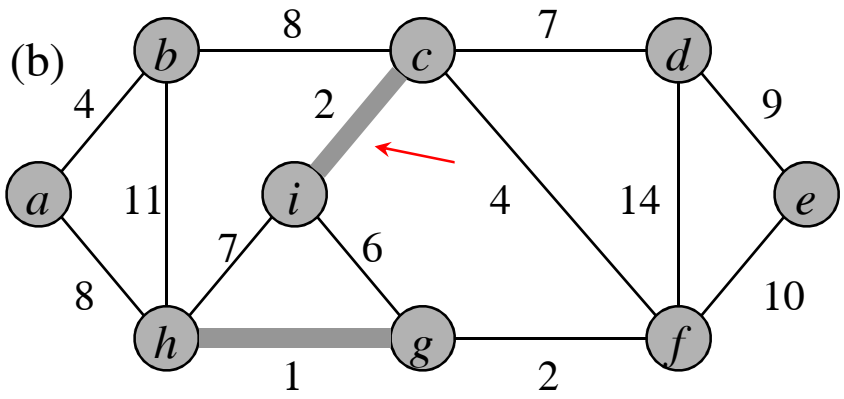
- mỗi tập rời nhau chứa các đỉnh của một cây trong rừng hiện thời.

Thực thi giải thuật của Kruskal

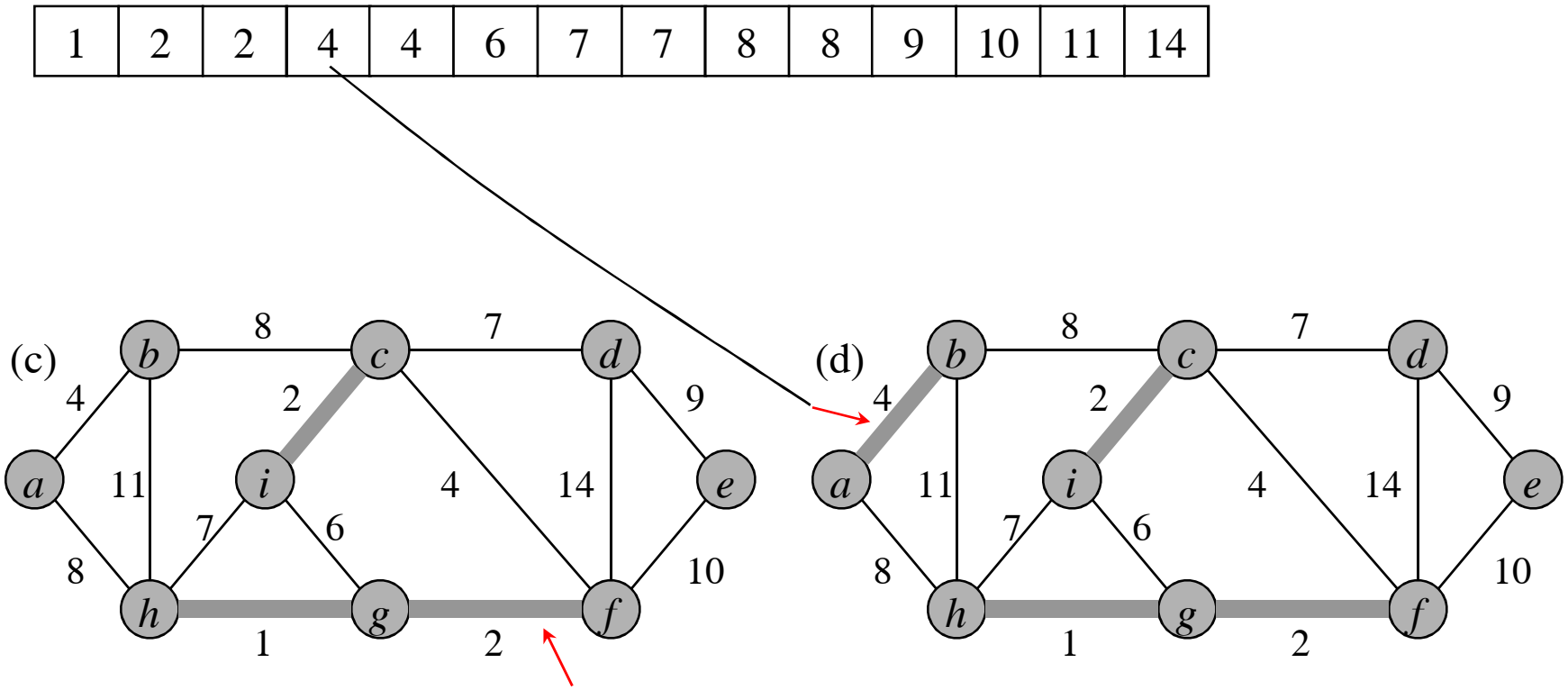
Các cạnh được xếp theo thứ tự trọng số không giảm:

1	2	2	4	4	6	7	7	8	8	9	10	11	14

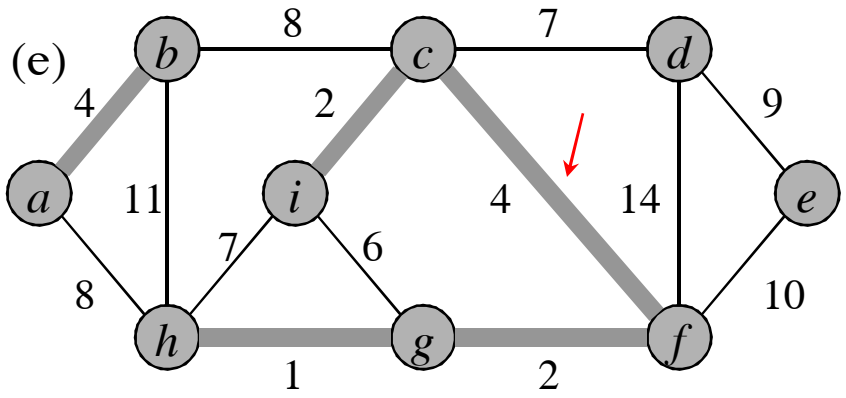


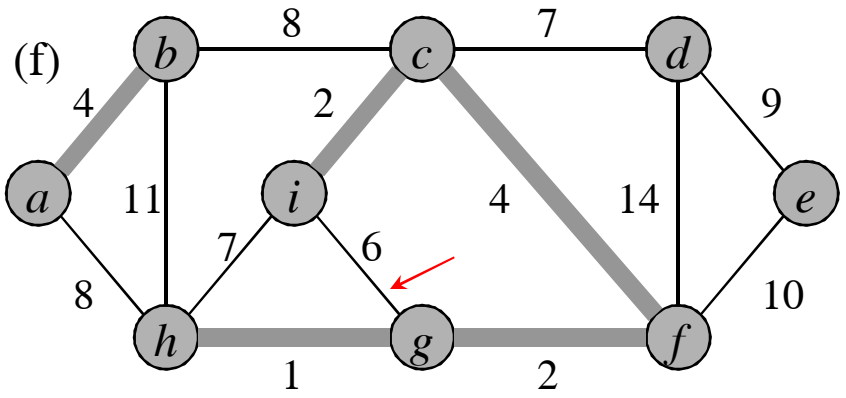


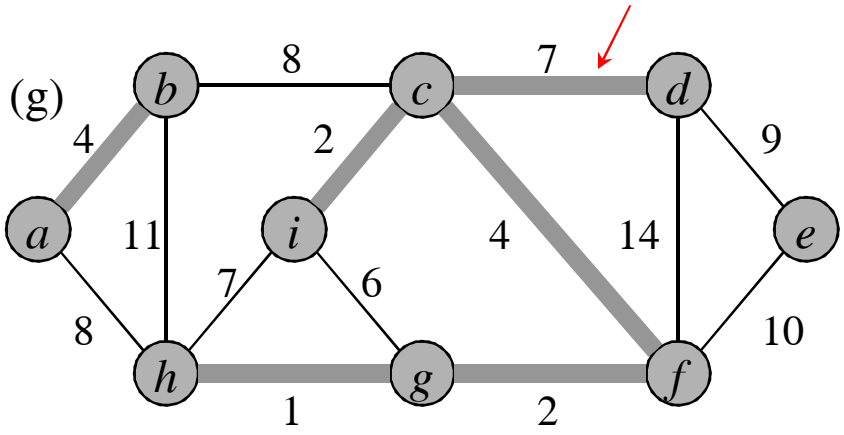
Thực thi giải thuật của Kruskal (tiếp)

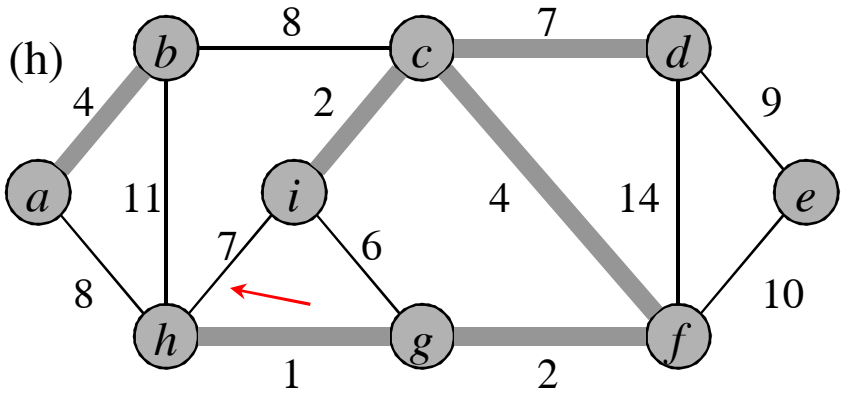


Thực thi giải thuật của Kruskal (tiếp)

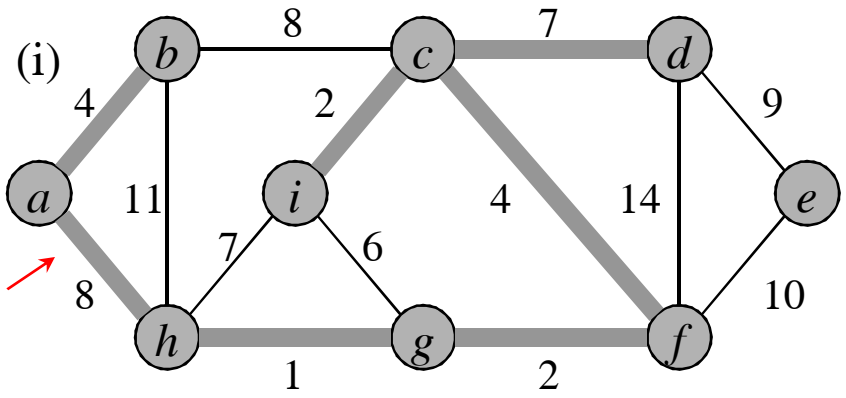


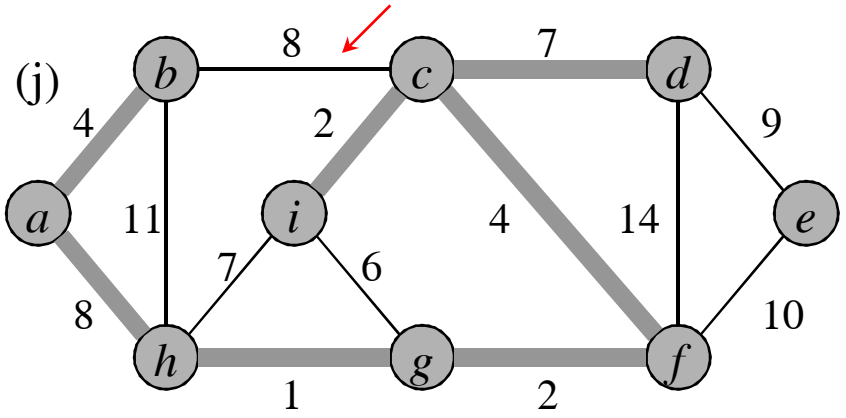


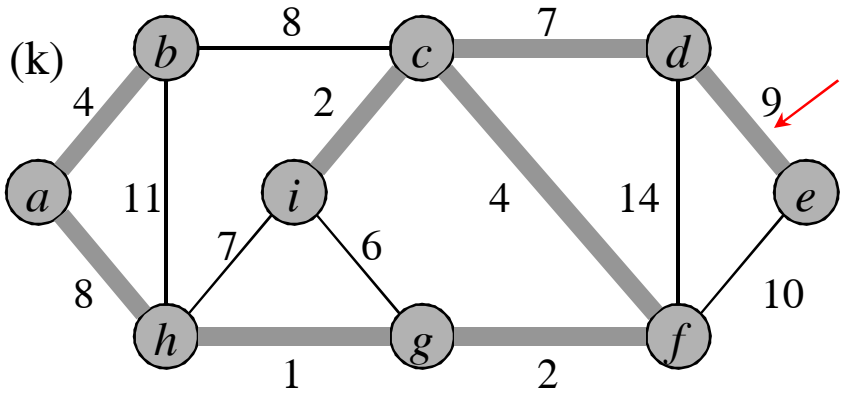


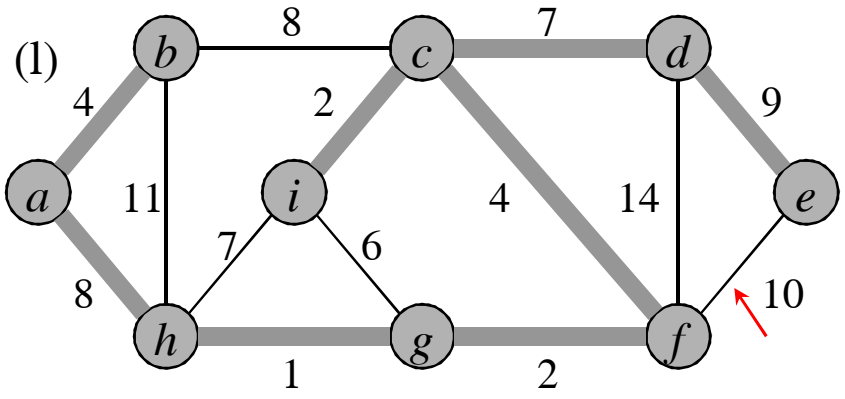


Thực thi giải thuật của Kruskal (tiếp)



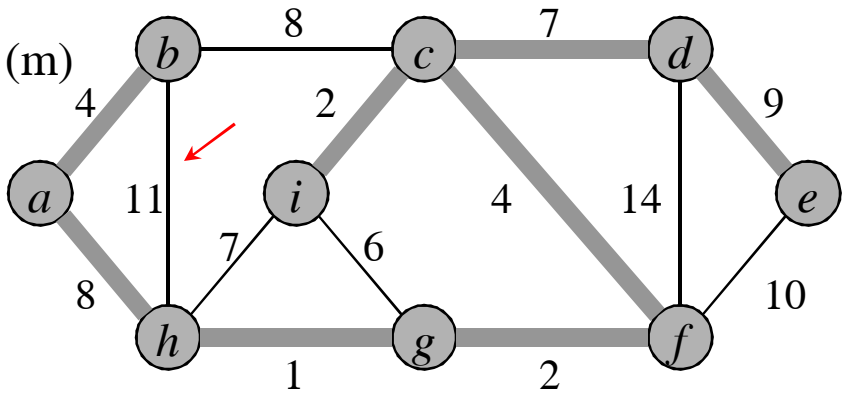


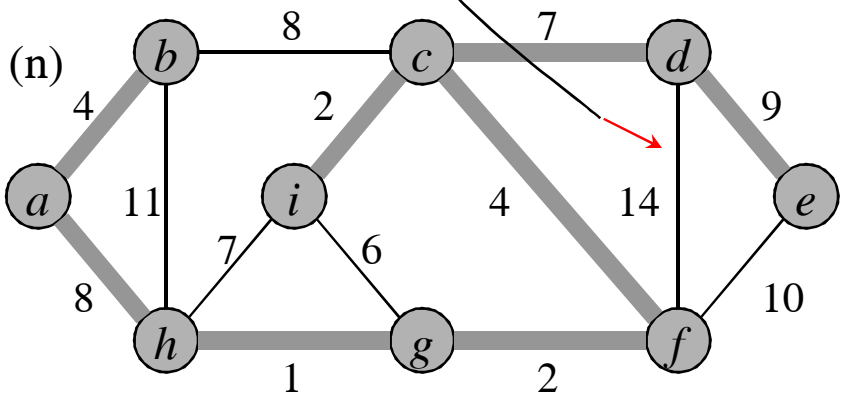




Thực thi giải thuật của Kruskal (tiếp)

1	2	2	4	4	6	7	7	8	8	9	10	11	14





Phân tích giải thuật của Kruskal

- Dùng cấu trúc dữ liệu các tập rời nhau (disjoint sets), chương 22, với các heuristics
  - Hợp theo thứ hạng (union-by-rank)
  - Nén đường dẫn (path-compression).
- Nhận xét (cần đến khi đánh giá thời gian chạy)
  - Giải thuật gọi $|V|$ lần MAKE-SET và gọi tổng cộng $O(E)$ lần các thao tác MAKE-SET, UNION, FIND-SET.
  - Vì G liên thông nên $|E| \geq |V| - 1$.

Phân tích giải thuật của Kruskal (tiếp)

- Thời gian chạy của MST-KRUSKAL gồm
  - Khởi động: $O(V)$
  - Sắp xếp ở dòng 4: $O(E \lg E)$
  - Dòng 5-8: $O(E\ \alpha(E, V))$ (xem nhận xét),
    $= O(E \lg E)$ vì $\alpha(E, V) = O(\lg E)$.
- Vậy thời gian chạy của MST-KRUSKAL là $O(E \lg E)$.

Giải thuật của Prim

- Giải thuật của Prim
  - dựa trên giải thuật GENERIC-MST, ở đây A là một cây duy nhất
    - trong khi thực thi giải thuật
      $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$
    - khi giải thuật xong, $Q = \varnothing$, nên
      $A = \{(v, \pi[v]) : v \in V - \{r\}\}$

Giải thuật của Prim (tiếp)

```
MST-PRIM(G, w, r)
 1   Q ← V[G]
 2   for mỗi đỉnh u ∈ Q
 3       do key[u] ← ∞
 4   key[r] ← 0
 5   π[r] ← NIL
 6   while Q ≠ ∅
 7       do u ← EXTRACT-MIN(Q)
 8          for mỗi đỉnh v ∈ Adj[u]
 9              do if v ∈ Q và w(u, v) < key[v]
10                    then π[v] ← u
11                         key[v] ← w(u, v)
```

r : gốc của cây khung nhỏ nhất sẽ trả về

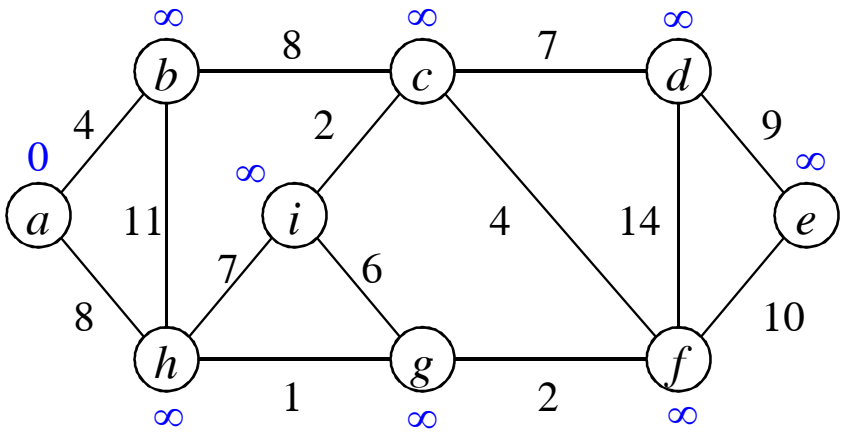
Q : priority queue mà khóa là trường *key*

$\pi[v]$: đỉnh cha mẹ của v.

- Tập $V - Q$ chứa các đỉnh của cây đang được nuôi lớn.

Thực thi giải thuật của Prim

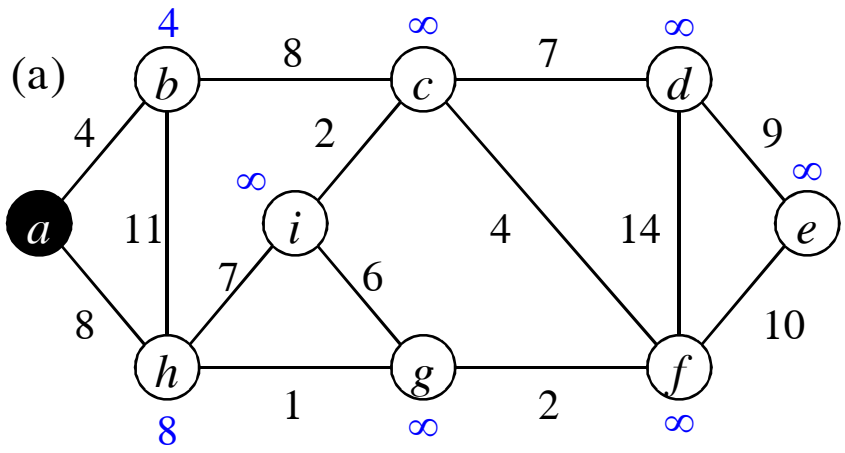
Sau khi khởi động:
(các số bên mỗi đỉnh là trị của *key* của đỉnh)



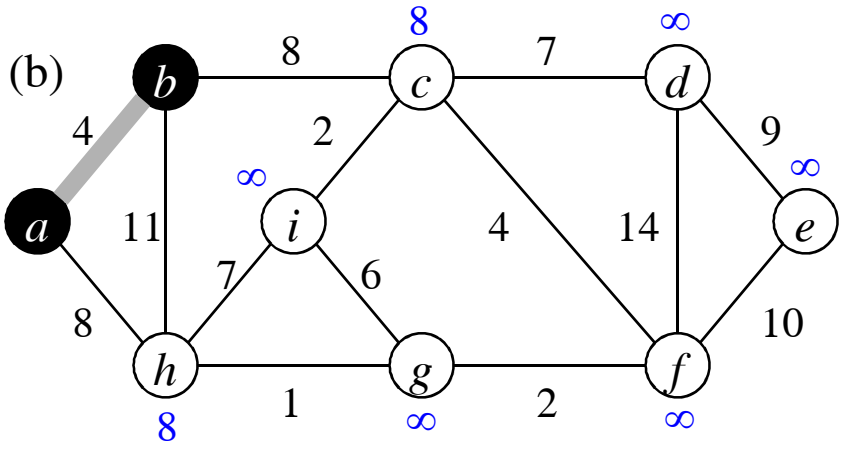
Thực thi giải thuật của Prim (tiếp)

Các đỉnh còn trong Q màu trắng, các đỉnh đã được đưa ra khỏi Q màu đen

Sau lần lặp 1:

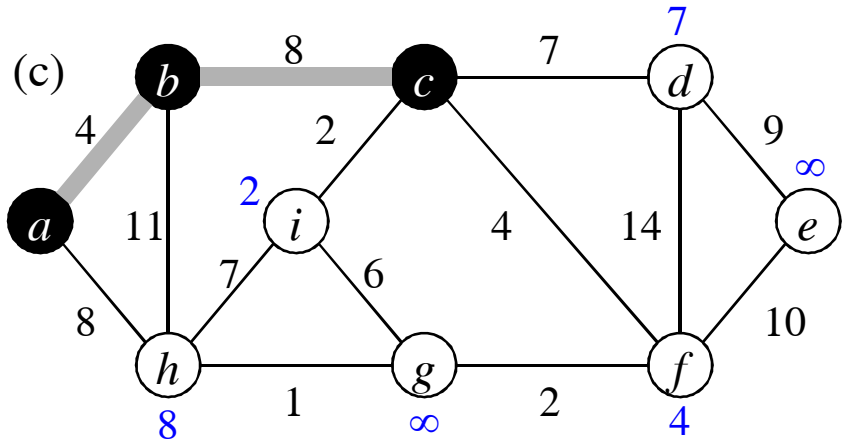


Sau lần lặp 2:

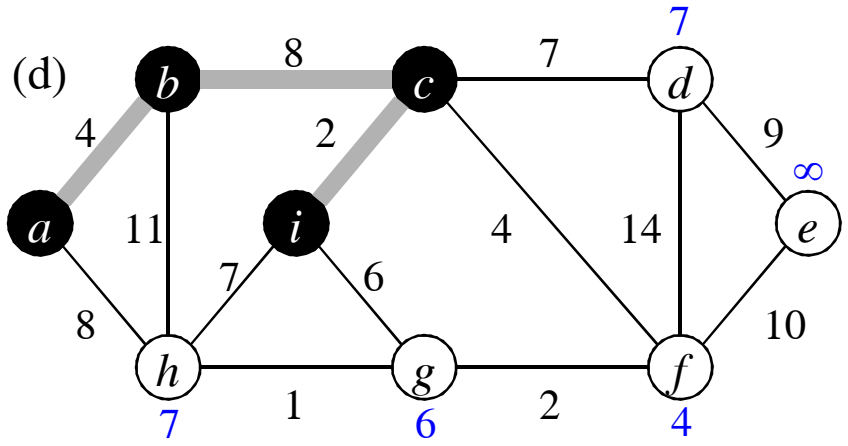


Thực thi giải thuật của Prim (tiếp)

Sau lần lặp 3:

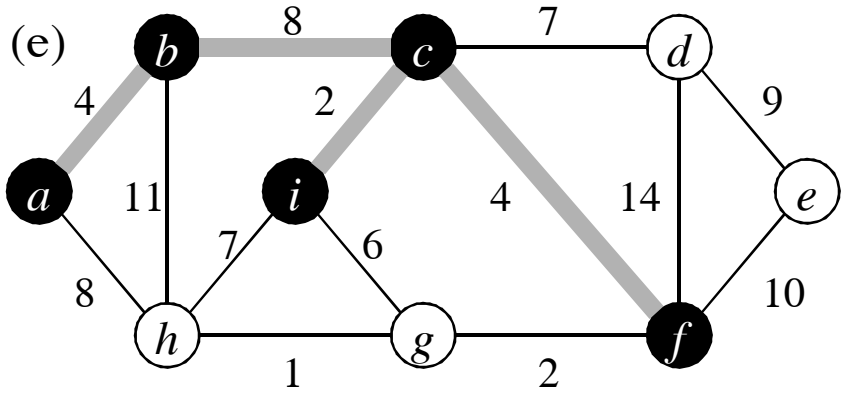


Sau lần lặp 4:

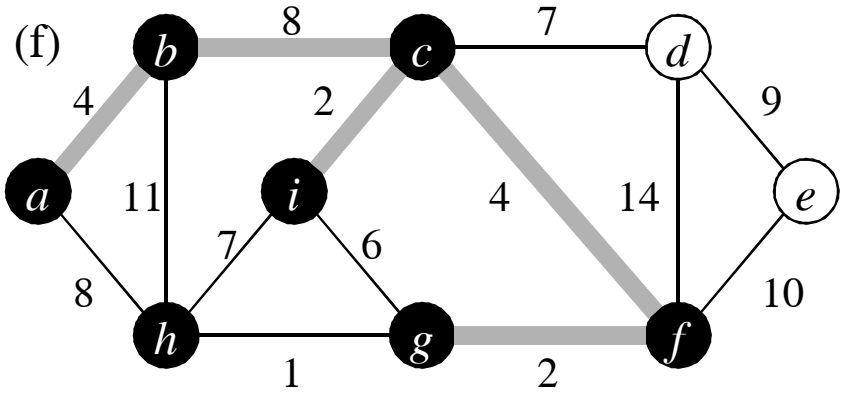


Thực thi giải thuật của Prim (tiếp)

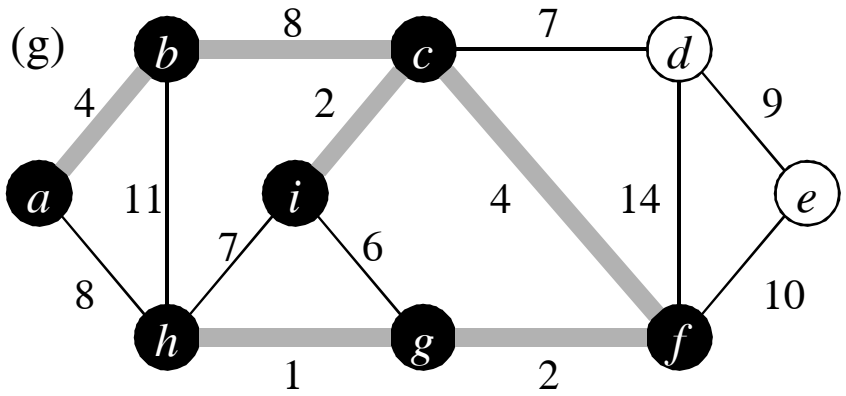
Sau lần lặp 5:



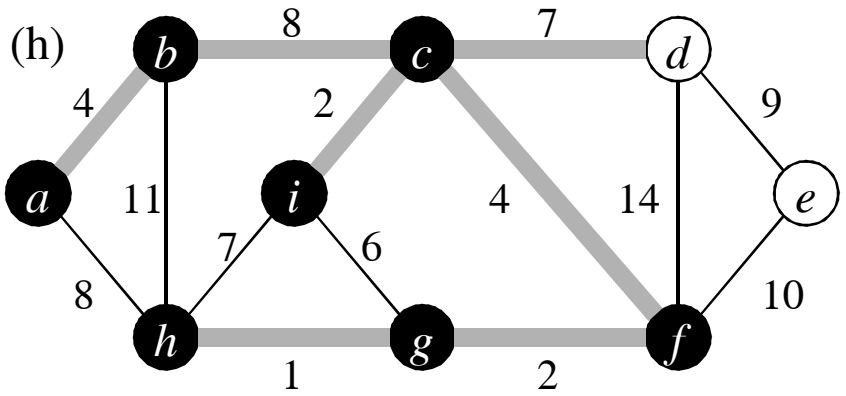
Sau lần lặp 6:



Sau lần lặp 7:

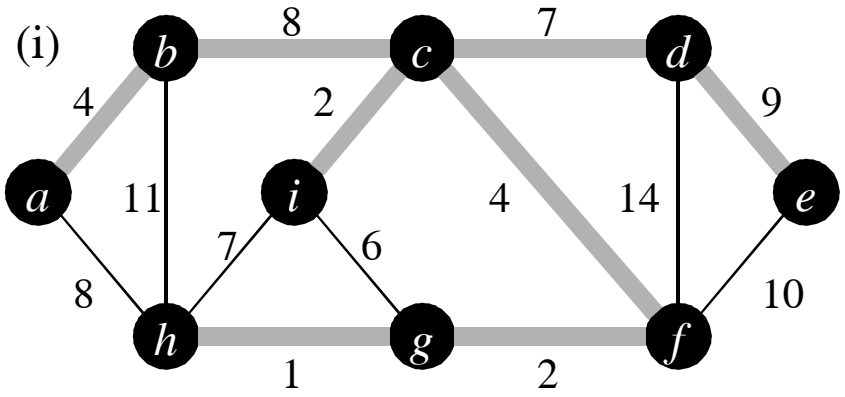


Sau lần lặp 8:



Thực thi giải thuật của Prim (tiếp)

Sau lần lặp 9: (i)



Phân tích giải thuật của Prim

- Thời gian chạy của MST-PRIM tùy thuộc vào cách hiện thực priority queue Q
  - Trường hợp hiện thực Q là binary heap
    - Khởi tạo trong dòng 1-4 dùng BUILD-HEAP tốn $O(V)$ thời gian
    - Vòng **while** được lặp $|V|$ lần, mỗi EXTRACT-MIN tốn $O(\lg V)$ thời gian. Như vậy các lần gọi EXTRACT-MIN tốn tất cả $O(V \lg V)$ thời gian.
      - Vòng **for** được lặp $O(E)$ lần, trong vòng lặp này dòng 11 (dùng HEAPIFY) tốn $O(\lg V)$ thời gian.
    - Vậy thời gian chạy tổng cộng của MST-PRIM là $O(V \lg V + E \lg V) = O(E \lg V)$.

Phân tích giải thuật của Prim (tiếp)

– Trường hợp hiện thực Q là Fibonacci heap

- Khởi tạo trong dòng 1- 4 dùng MAKE-FIB-HEAP và FIB-HEAP-INSERT tốn $O(V)$ amortized time
- Mỗi FIB-HEAP-EXTRACT-MIN tốn $O(\lg V)$ amortized time
- Mỗi thao tác FIB-HEAP-DECREASE-KEY cần để hiện thực dòng 11 tốn $O(1)$ amortized time
- Vậy thời gian chạy tổng cộng của MST-PRIM là $O(E + V \lg V)$.

Single-Source Shortest Paths

Các đường đi ngắn nhất từ một đỉnh nguồn

- *Bài toán các đường đi ngắn nhất*: một số thuật ngữ.

  Cho một đồ thị có trọng số, có hướng $G = (V, E)$, với một hàm trọng số $w : E \to \Re$

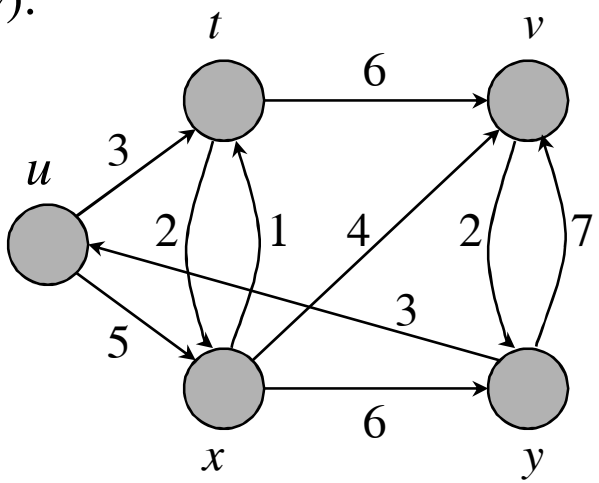
  – *Trọng số* của một đường đi $p = \langle v_0, v_1, \ldots, v_k \rangle$

    • $w(p) = \sum_{i = 1 \ldots k} w(v_{i-1}, v_i)$

  – *Trọng số của đường đi ngắn nhất* (shortest path weight) từ u đến v

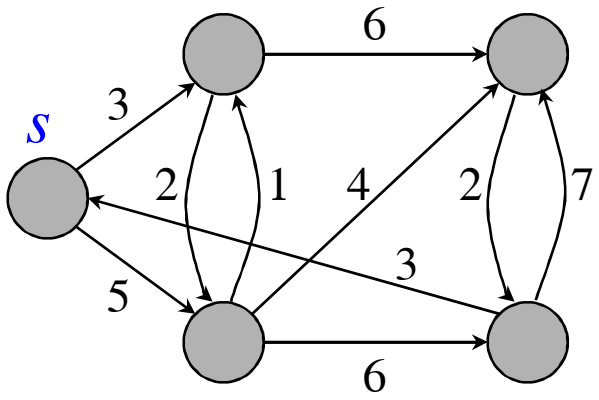
$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{nếu có đường đi từ } u \text{ đến } v \\ \infty & \text{các trường hợp khác} \end{cases}$$

  – *Đường đi ngắn nhất* từ u đến v là bất kỳ đường đi p nào từ u đến v sao cho $w(p) = \delta(u, v)$.



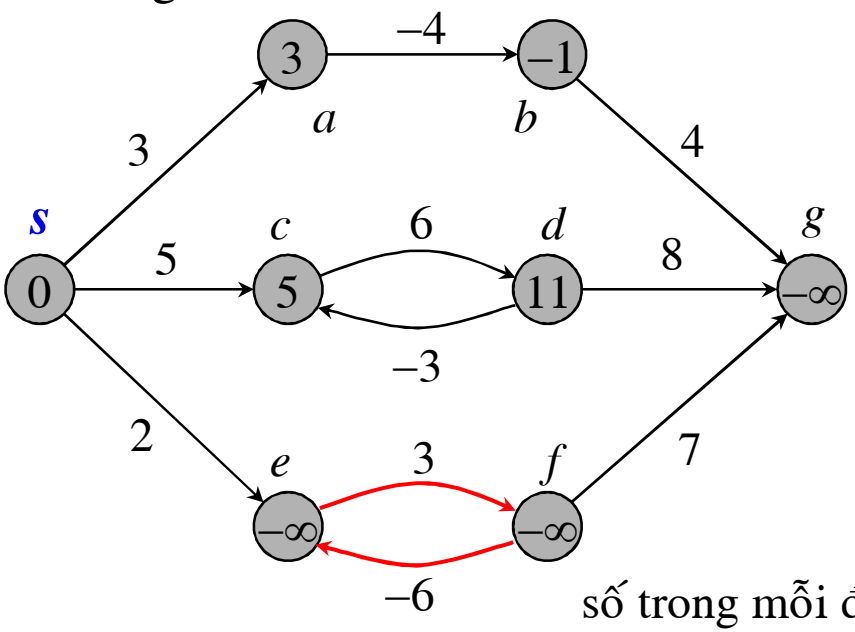
Các đường đi ngắn nhất từ một đỉnh nguồn (tiếp)

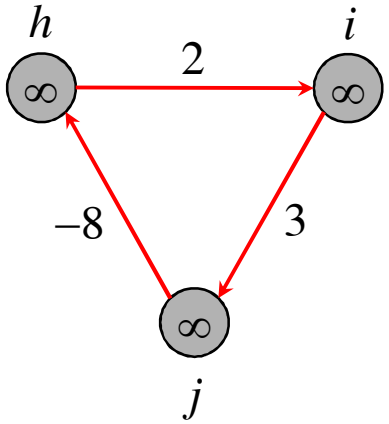
- *Bài toán các đường đi ngắn nhất từ một nguồn duy nhất* (Single-source shortest-paths problem):
  - Cho đồ thị $G = (V, E)$ và một *đỉnh nguồn* $s \in V$.
  - Tìm đường đi ngắn nhất từ s đến <u>mọi</u> đỉnh $v \in V$.



Cạnh có trọng số âm

- Giả thiết: Trọng số của cạnh có thể <u>âm</u>
  - <u>Chu trình</u> có thể có trọng số <u>âm</u>
  - Nếu tồn tại một chu trình có trọng số âm đến được (reachable) từ *s* thì trọng số của đường đi ngắn nhất không được định nghĩa: không đường đi nào từ *s* đến một đỉnh nằm trên chu trình có thể là đường đi ngắn nhất.

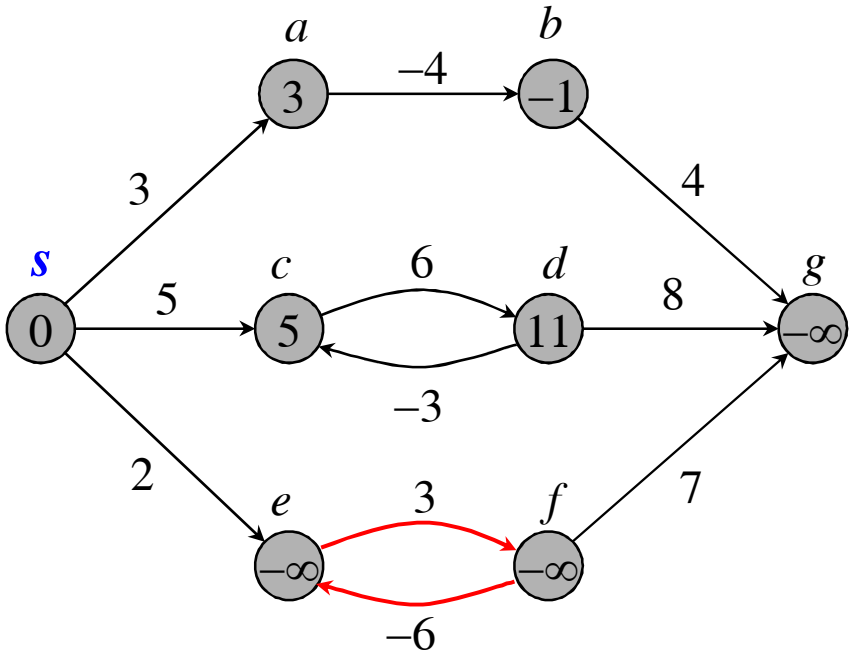


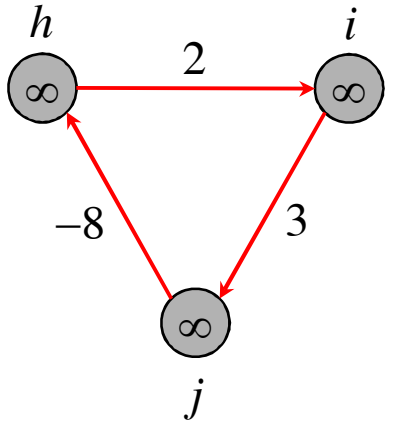


số trong mỗi đỉnh là trọng số đường đi ngắn nhất từ đỉnh nguồn *s*.

Cạnh có trọng số âm (tiếp)

- Nếu tồn tại một chu trình có trọng số âm trên một đường đi từ s đến v, ta định nghĩa $\delta(s, v) = -\infty$.
- Trong ví dụ sau, các đỉnh h, i, j không đến được từ s nên có trọng số đường đi ngắn nhất là ∞ (chứ không là $-\infty$ mặc dù chúng nằm trên một chu trình có trọng số âm).

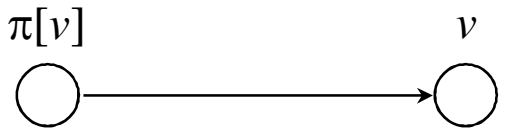




Biểu diễn các đường đi ngắn nhất

- Đồ thị $G = (V, E)$
  - Với mọi đỉnh v, *đỉnh cha* (predecessor) của v là một đỉnh khác hay là NIL.

    Duy trì $\pi[v]$, con trỏ đến đỉnh cha. Dùng π để suy ra đường đi ngắn nhất từ s đến v.
  - *Đồ thị con* $G_\pi = (V_\pi, E_\pi)$ (*predecessor subgraph*)
  - $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$
  - $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$



Biểu diễn các đường đi ngắn nhất (tiếp)

- Cho $G = (V, E)$ là một đồ thị có hướng, có trọng số;

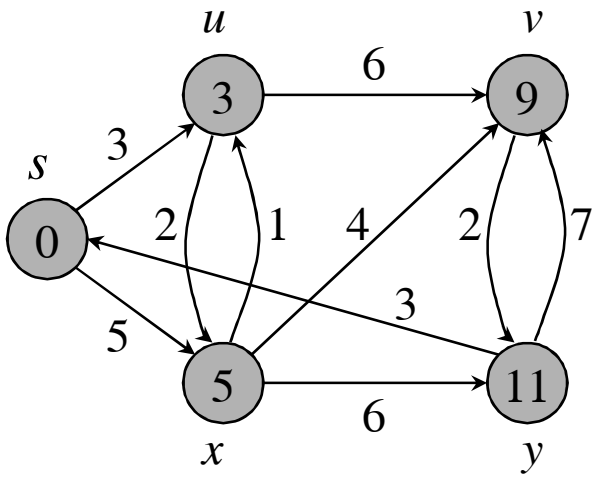
  G không chứa chu trình trọng số âm đến được từ đỉnh nguồn $s \in V$.

  Cây các đường đi ngắn nhất với gốc tại s là đồ thị có hướng $G' = (V', E')$, với $V' \subseteq V$ và $E' \subseteq E$ sao cho

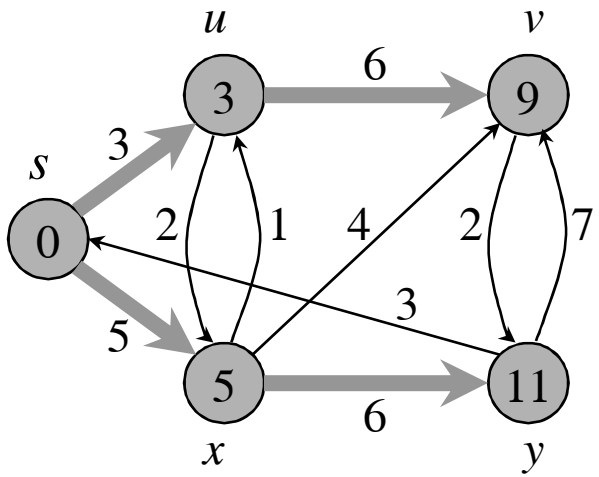
  1. V' là tập các đỉnh đến được (reachable) từ s trong G
  2. G' là cây có gốc với gốc là s
  3. Với mọi $v \in V'$, đường đi đơn duy nhất từ s đến v là đường đi ngắn nhất từ s đến v trong G .

Cây các đường đi ngắn nhất có gốc tại đỉnh nguồn *s*

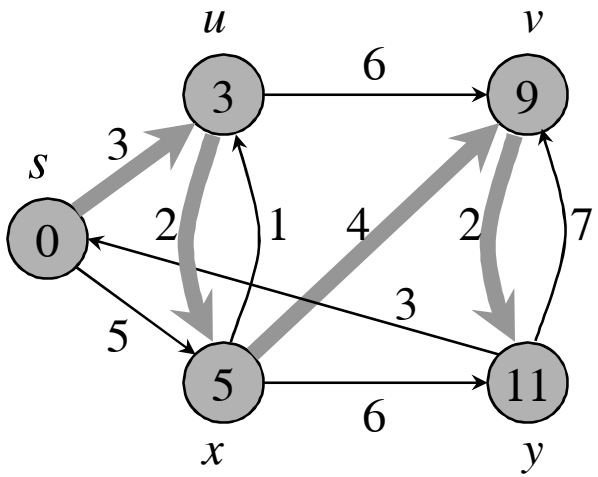
Ví dụ: trong (b) và (c) là hai cây các đường đi ngắn nhất có gốc tại đỉnh nguồn *s* của đồ thị trong (a)



(a)



(b)



(c)

Cấu trúc của đường đi ngắn nhất

- ***Lemma 25.1 (Đường đi con của đường đi ngắn nhất cũng là đường đi ngắn nhất)***

  Cho

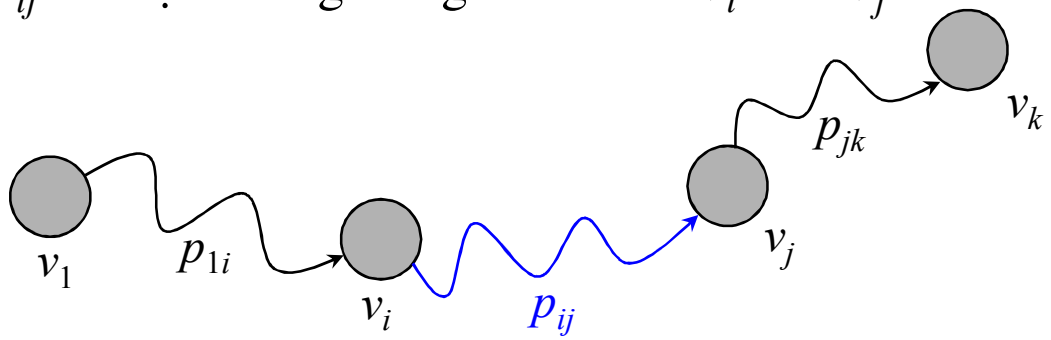
  – Đồ thị có trọng số, có hướng $G = (V, E)$ với hàm trọng số $w : E \to \Re$

  – $p = \langle v_1 , v_2 , \ldots , v_k \rangle$ đường đi ngắn nhất từ v_1 đến v_k

  – Với mọi i, j mà $1 \le i \le j \le k$, gọi $p_{ij} = \langle v_i , v_{i+1} , \ldots , v_j \rangle$ là đường đi con (subpath) của p từ v_i đến v_j .

  $\Rightarrow$

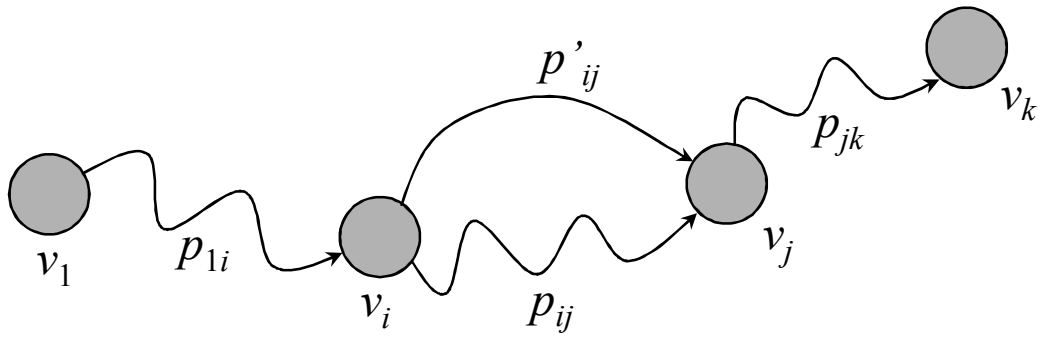
  p_{ij} là một đường đi ngắn nhất từ v_i đến v_j .



Cấu trúc của đường đi ngắn nhất (tiếp)

Chứng minh

Phản chứng.



Cấu trúc của đường đi ngắn nhất (tiếp)

- ***Hệ luận 25.2***

  Cho

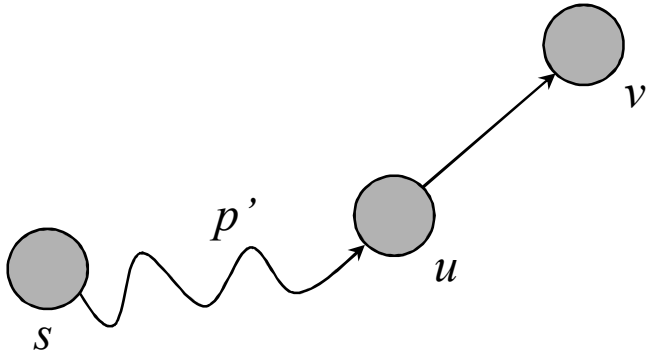
  – Đồ thị có trọng số, có hướng $G = (V, E)$ với hàm trọng số $w : E \to \Re$

  – p là đường đi ngắn nhất từ s đến v, và có thể được phân thành $s \overset{p'}{\rightsquigarrow} u \to v$.

  $\Rightarrow$

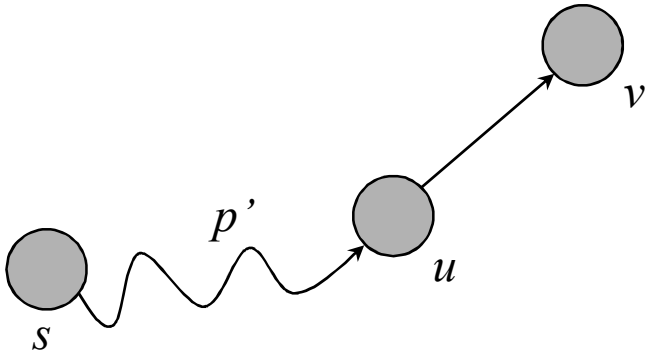
  Trọng số của đường đi ngắn nhất từ s đến v là

  $\delta(s, v) = \delta(s, u) + w(u, v)$.



Cấu trúc của đường đi ngắn nhất (tiếp)

Chứng minh



Cấu trúc của đường đi ngắn nhất (tiếp)

- ***Lemma 25.3***

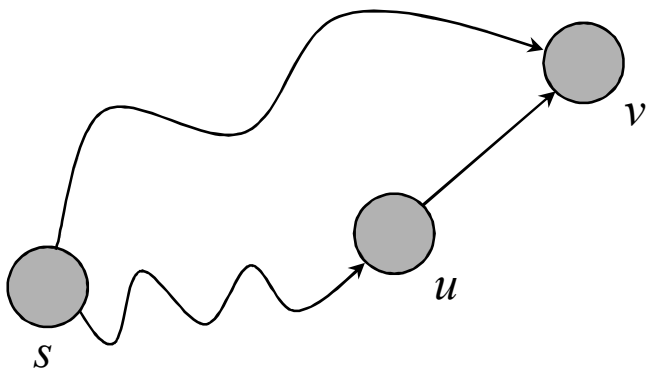
  Cho

  – Đồ thị có trọng số, có hướng $G = (V, E)$ với hàm trọng số $w : E \to \Re$

  – Đỉnh nguồn s

  $\Rightarrow$

  Với mọi cạnh $(u, v) \in E$, ta có $\delta(s, v) \leq \delta(s, u) + w(u, v)$.



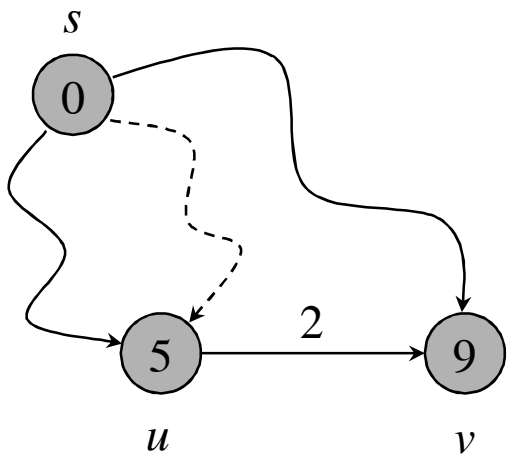
Kỹ thuật nới lỏng

- Kỹ thuật nới lỏng (relaxation)
  - Duy trì cho mỗi đỉnh v một thuộc tính $d[v]$ dùng làm chận trên cho trọng số của một đường đi ngắn nhất từ s đến v.
  - biến $d[v]$ được gọi là *ước lượng đường đi ngắn nhất* (*shortest path estimate*)
  - Khởi động các ước lượng đường đi ngắn nhất và các predecessors bằng thủ tục sau

```
INITIALIZE-SINGLE-SOURCE(G, s)
1   for each vertex v ∈ V[G]
2       do d[v] ← ∞
3          π[v] ← NIL
4   d[s] ← 0
```

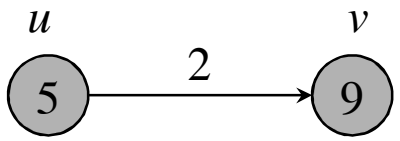
Kỹ thuật nới lỏng (tiếp)

- Thực thi nới lỏng lên một cạnh (u, v): kiểm tra xem một đường đi đến v thông qua cạnh (u, v) có ngắn hơn một đường đi đến v đã tìm được hiện thời hay không. Nếu ngắn hơn thì cập nhật $d[v]$ và $\pi[v]$.

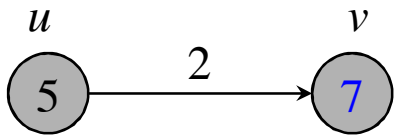


RELAX(u, v, w)
1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Thực thi nới lỏng lên một cạnh

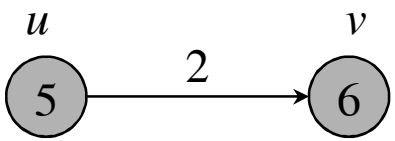


RELAX(u, v, w)

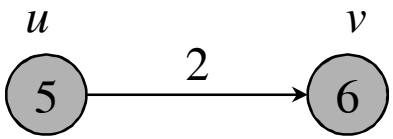


(a)

Trị của $d[v]$ giảm sau khi gọi RELAX(u, v, w)



RELAX(u, v, w)



(b)

Trị của $d[v]$ không thay đổi sau khi gọi RELAX(u, v, w)

Kỹ thuật nới lỏng (tiếp)

- Các giải thuật trong chương này gọi INITIALIZE-SINGLE-SOURCE và sau đó gọi RELAX một số lần để nới lỏng các cạnh.
  - Nới lỏng là cách duy nhất được dùng để thay đổi các ước lượng đường đi ngắn nhất và các predecessors.
  - Các giải thuật khác nhau ở thứ tự và số lần gọi RELAX lên các cạnh.

Các tính chất của kỷ thuật nới lỏng

- ***Lemma 25.4***

  Cho

  – Đồ thị có trọng số, có hướng $G = (V, E)$, với hàm trọng số $w : E \to \Re$

  – Cạnh $(u, v) \in E$.

  $\Rightarrow$

  Ngay sau khi gọi RELAX(u, v, w) ta có

  $d[v] \leq d[u] + w(u, v)$.

Các tính chất của kỷ thuật nới lỏng (tiếp)

- ***Lemma 25.5***

  Cho

  – Đồ thị có trọng số, có hướng $G = (V, E)$, với hàm trọng số $w : E \to \Re$

  – Đỉnh nguồn s.

  – G được khởi động bởi INITIALIZE-SINGLE-SOURCE(G, s).

  $\Rightarrow$

  – Với mọi $v \in V$, $d[v] \geq \delta(s, v)$, bất biến này được duy trì đối với mọi dãy các bước nới lỏng lên các cạnh của G

  – Một khi $d[v]$ đạt đến cận dưới $\delta(s, v)$ của nó thì nó sẽ không bao giờ thay đỗi.

Các tính chất của kỷ thuật nới lỏng (tiếp)

- ***Hệ luận 25.6***

  Cho

  – Đồ thị có trọng số, có hướng $G = (V, E)$, với hàm trọng số $w : E \to \Re$

  – Đỉnh nguồn s

  – Không có đường đi từ s đến một đỉnh $v \in V$.

  $\Rightarrow$

  Sau khi G được khởi động bởi INITIALIZE-SINGLE-SOURCE(G, s), ta có

  – đẳng thức $d[v] = \delta(s, v)$

  – đẳng thức này được duy trì thành bất biến đối với mọi dãy các bước nới lỏng lên các cạnh của G.

Các tính chất của kỹ thuật nới lỏng (tiếp)

- Để chứng minh tính đúng đắn của các giải thuật tìm đường đi ngắn nhất (giải thuật Dijkstra và giải thuật Bellman-Ford) ta cần Lemma sau.
- ***Lemma 25.7***

  Cho
  - Đồ thị có trọng số và có hướng $G = (V, E)$, với hàm trọng số $w : E \to \Re$
  - Một đỉnh nguồn s, và $s \rightsquigarrow u \to v$ là một đường đi ngắn nhất trong G với các đỉnh nào đó $u, v \in V$.
  - G được khởi động bởi INITIALIZE-SINGLE-SOURCE(G, s) và sau đó một chuỗi các bước nới lỏng trong đó có gọi RELAX(u, v, w) được thực thi lên các cạnh của G.

  $\Rightarrow$

  Nếu $d[u] = \delta(s, u)$ trước khi gọi RELAX(u, v, w), thì sau khi gọi luôn luôn có $d[v] = \delta(s, v)$.

Cây các đường đi ngắn nhất

- ***Lemma 25.8***

  Cho

  – Đồ thị có trọng số và có hướng $G = (V, E)$, với hàm trọng số $w : E \to \Re$

  – Đỉnh nguồn $s \in V$

  – G không chứa chu trình có trọng số âm đến được từ s

  $\Rightarrow$

  Sau khi khởi động G bởi INITIALIZE-SINGLE-SOURCE(G, s), ta có

  – Đồ thị G_π là cây có gốc s

  – Mọi chuỗi các bước nới lỏng lên các cạnh của G duy trì tính chất trên thành một bất biến.

Cây các đường đi ngắn nhất (tiếp)

- ***Lemma 25.9***

  Cho

  – Đồ thị có trọng số và có hướng $G = (V, E)$, với hàm trọng số $w : E \to \Re$

  – Đỉnh nguồn $s \in V$

  – G không chứa chu trình có trọng số âm đến được từ s.

  Khởi động G bằng INITIALIZE-SINGLE-SOURCE(G, s) và thực thi chuỗi bất kỳ các bước nới lỏng lên các cạnh của G sao cho $d[v] = \delta(s, v)$ với mọi đỉnh $v \in V$.

  $\Rightarrow$

  Đồ thị G_π là một cây các đường đi ngắn nhất có gốc tại s.

Giải thuật của Dijkstra

- Đồ thị $G = (V, E)$ có hướng, có trọng số với
  - Hàm trọng số $w : E \to \Re$ mà $w(u, v) \geq 0$ cho mọi cạnh $(u, v) \in E$
  - Đỉnh nguồn s.
- Giải thuật của Dijkstra:
  - Dùng một priority queue Q với khóa là các trị $d[\ ]$

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5       do u ← EXTRACT-MIN(Q)
6          S ← S ∪ {u}
7          for each vertex v ∈ Adj[u]
8               do RELAX(u, v, w)
```

Phân tích giải thuật của Dijkstra

- Thời gian chạy phụ thuộc vào hiện thực của priority queue Q:
  - Linear array
    - Mỗi EXTRACT-MIN tốn $O(V)$ thời gian, vậy tất cả các EXTRACT-MIN tốn $O(V^2)$
    - Tất cả các lần gọi RELAX tốn $O(E)$ thời gian

    Thời gian chạy tổng cộng: $O(V^2 + E) = O(V^2)$.

Phân tích giải thuật của Dijkstra (tiếp)

- Thời gian chạy phụ thuộc vào hiện thực của priority queue Q:
  - Binary heap
    - Tạo heap tốn $O(V)$ thời gian.
    - Mỗi EXTRACT-MIN tốn $O(\lg V)$ thời gian, vậy tất cả các EXTRACT-MIN tốn $O(V \lg V)$
    - Tất cả các lần gọi RELAX tốn $O(E \lg V)$ thời gian, vì mỗi DECREASE-KEY để hiện thực RELAX tốn $O(\lg V)$ thời gian

  Thời gian chạy tổng cộng: $O((V + E) \lg V)$.

Phân tích giải thuật của Dijkstra

(tiếp)

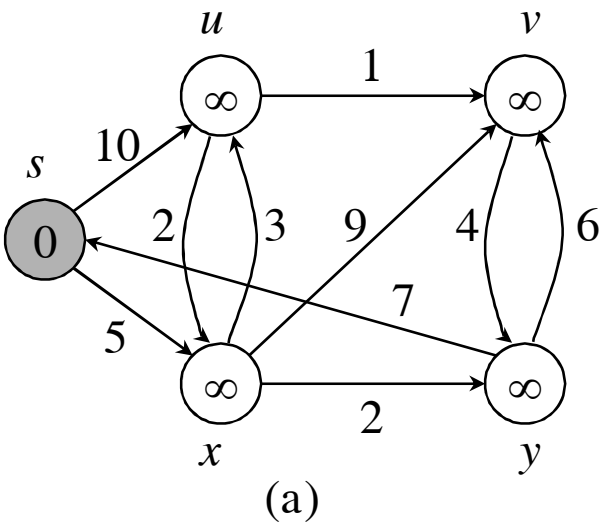
- Fibonacci heap
  - Tạo heap với $|V|$ phần tử tốn $O(V)$ thời gian.
  - Mỗi EXTRACT-MIN tốn $O(\lg V)$ phí tổn bù trừ, vậy tất cả các EXTRACT-MIN tốn $O(V \lg V)$ thời gian
  - Tất cả các lần gọi RELAX tốn $O(E)$ thời gian, vì mỗi DECREASE-KEY để hiện thực RELAX tốn $O(1)$ phí tổn bù trừ

Thời gian chạy tổng cộng: $O(V \lg V + E)$.

Thực thi giải thuật của Dijkstra

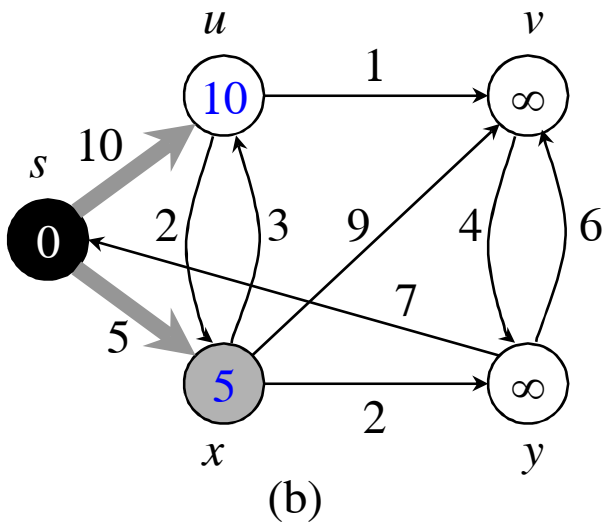
Đỉnh màu đen là đỉnh trong S
Đỉnh màu xám là đỉnh được đem vào S trong lần lặp tới

Ngay trước khi vào vòng lặp **while:**



(a)

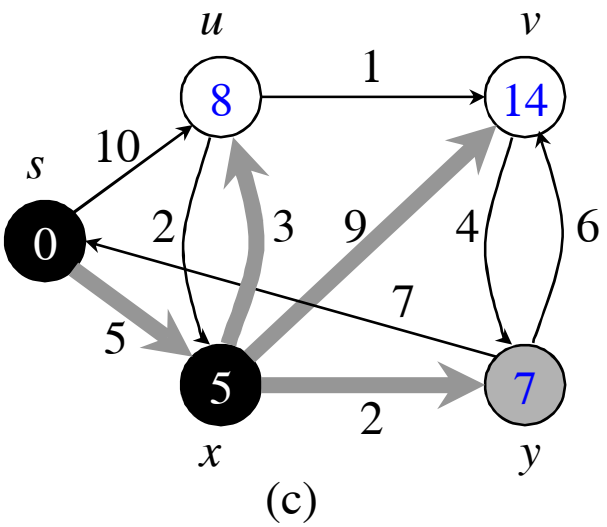
Vòng **while:** ngay sau lần lặp 1



(b)

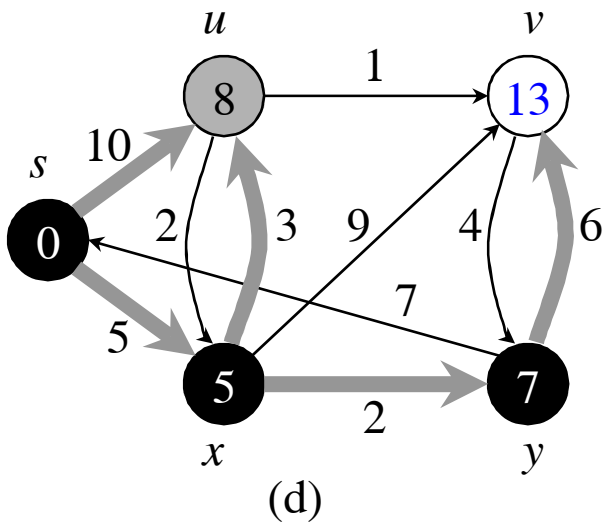
Thực thi giải thuật của Dijkstra (tiếp)

Vòng **while:** ngay sau lần lặp 2



(c)

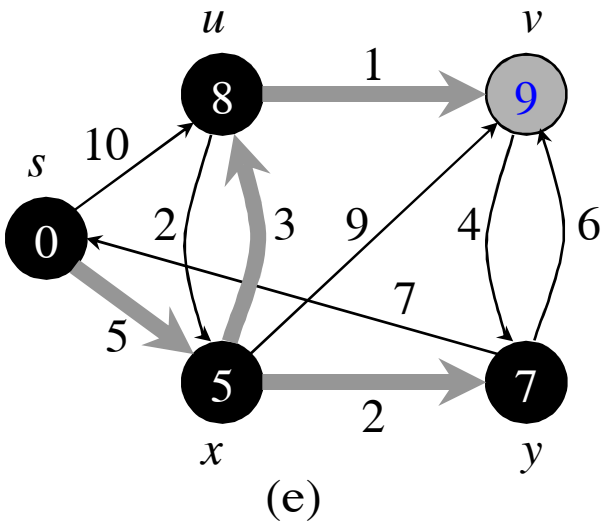
Vòng **while:** ngay sau lần lặp 3



(d)

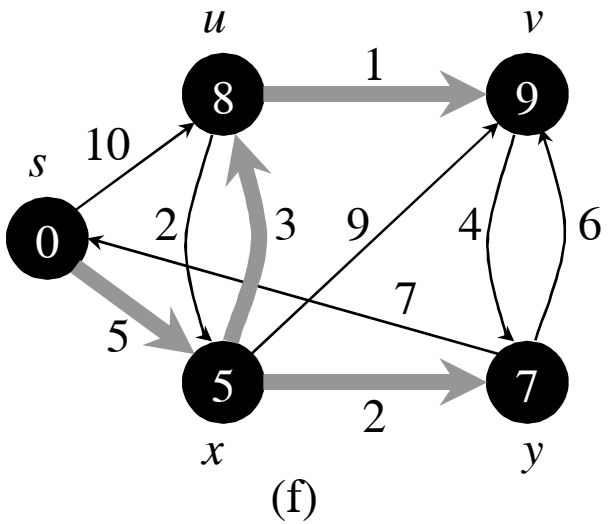
Thực thi giải thuật của Dijkstra (tiếp)

Vòng **while:** ngay sau lần lặp 4



(e)

Vòng **while:** ngay sau lần lặp 5



(f)

Tính đúng đắn của giải thuật của Dijkstra

- ***Định lý 25.10 (Tính đúng đắn của giải thuật của Dijkstra)***

  Thực thi giải thuật của Dijkstra lên đồ thị $G = (V, E)$ có trọng số và có hướng với

  – hàm trọng số $w : E \to \Re$ <u>không âm</u>

  – đỉnh nguồn s.

  $\Rightarrow$

  Khi giải thuật thực thi xong,

  $d[u] = \delta(s, u)$ cho mọi đỉnh $u \in V$.

  Chứng minh

  Sẽ chứng minh: $\forall u \in V, d[u] = \delta(s, u)$ khi u được đưa vào tập S và sau đó đẳng thức luôn được duy trì.

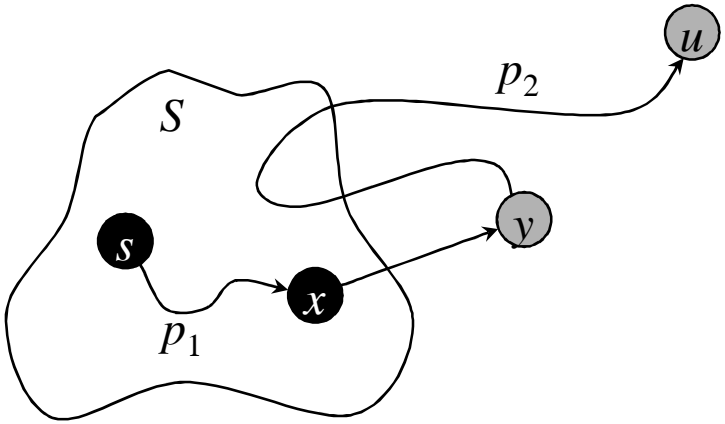
Tính đúng đắn của giải thuật của Dijkstra

Chứng minh (tiếp)

Chứng minh bằng phản chứng.

(*) Gọi u là đỉnh đầu tiên sao cho $d[u] \neq \delta(s, u)$ khi u được đưa vào tập S.

- Phải có một đường đi từ s đến u. Vì nếu không thì $\delta(s, u) = \infty$, do đó $d[u] = \infty$, do đó $d[u] = \delta(s, u)$ dùng Hệ luận 25.6, mâu thuẫn!
- Do đó có đường đi ngắn nhất p từ s đến u, với $s \in S$ và $u \in V - S$. Gọi y là đỉnh đầu tiên trên p sao cho $y \in V - S$. Đặt $x = \pi[y]$.

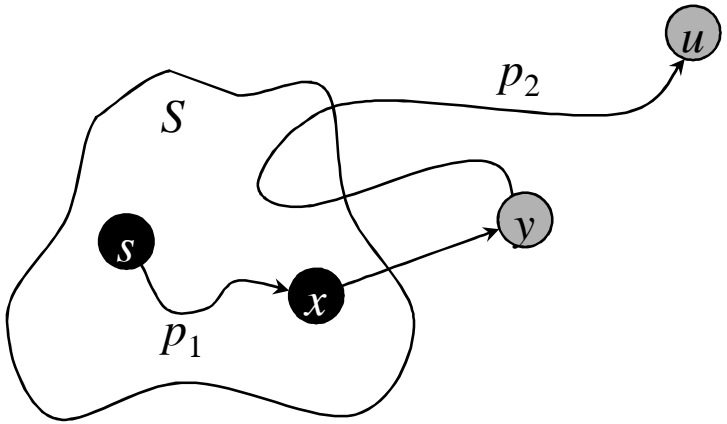


Tính đúng đắn của giải thuật của Dijkstra

Chứng minh (tiếp)

- Chứng tỏ $d[y] = \delta(s, y)$ khi u được đưa vào tập S: theo (*) ta phải có $d[x] = \delta(s, x)$ khi x được đưa vào S. Khi đó cạnh (x, y) được nới lỏng nên $d[y] = \delta(s, y)$ dùng Lemma 25.7.
- Vì y trước u trên đường đi ngắn nhất từ s đến u và mọi trọng số đều dương nên $\delta(s, y) \le \delta(s, u)$.

$$
\begin{aligned}
d[y] &= \delta(s, y) \\
&\le \delta(s, u) \\
&\le d[u] \qquad \text{dùng Lemma 25.5.}
\end{aligned}
$$



Tính đúng đắn của giải thuật của Dijkstra

Chứng minh (tiếp)

– Khi u được chọn bởi EXTRACT-MIN thì y cũng còn trong Q nên $d[u] \leq d[y]$, do đó bất đẳng thức :

$d[y] = \delta(s, y) = \delta(s, u) = d[u]$, từ đó $d[u] = \delta(s, u)$, mâu thuẫn với (*)!

– Dùng Lemma 25.5 để chứng minh phần còn lại.

Tính đúng đắn của giải thuật của Dijkstra (tiếp)

- ***Hệ luận 25.11***

  Thực thi giải thuật của Dijkstra lên đồ thị $G = (V, E)$ có trọng số và có hướng với

  – hàm trọng số $w : E \to \Re$ không âm

  – đỉnh nguồn s.

  $\Rightarrow$

  Khi giải thuật thực thi xong thì đồ thị G_π là cây các đường đi ngắn nhất có gốc tại s.

Giải thuật của Bellman-Ford

- Cho $G = (V, E)$ là đồ thị có trọng số, có hướng
  - Hàm trọng số $w : E \to \Re$
  - Đỉnh nguồn s.

BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 **for** $i \leftarrow 1$ **to** $|V[G]| - 1$
3 **do for** each edge $(u, v) \in E[G]$
4 **do** RELAX(u, v, w)
5 **for** each edge $(u, v) \in E[G]$
6 **do if** $d[v] > d[u] + w(u, v)$
7 **then return** FALSE
8 **return** TRUE

Phân tích giải thuật của Bellman-Ford

- Thời gian chạy:
  - Khởi tạo: $\Theta(V)$ thời gian
  - $|V| - 1$ lượt, mỗi lượt $O(E)$ thời gian
  - vòng lặp **for** dòng 5-7: $O(E)$ thời gian

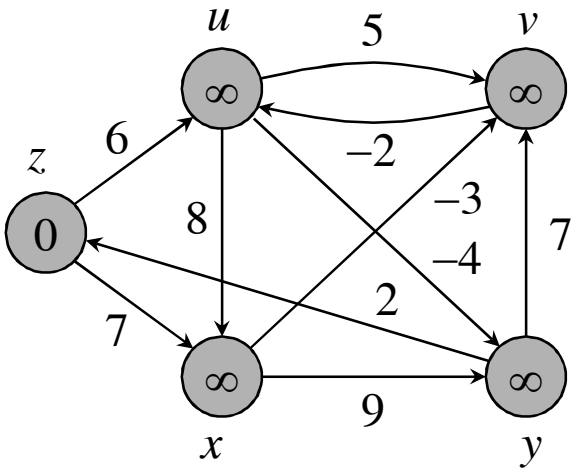
  Thời gian chạy tổng cộng: $O(V\,E)$

Thực thi giải thuật Bellman-Ford

Trong mỗi lượt, thứ tự relax các cạnh là:
(u, v) (u, x) (u, y) (v, u) (x, v) (x, y) (y, v) (y, z) (z, u) (z, x)

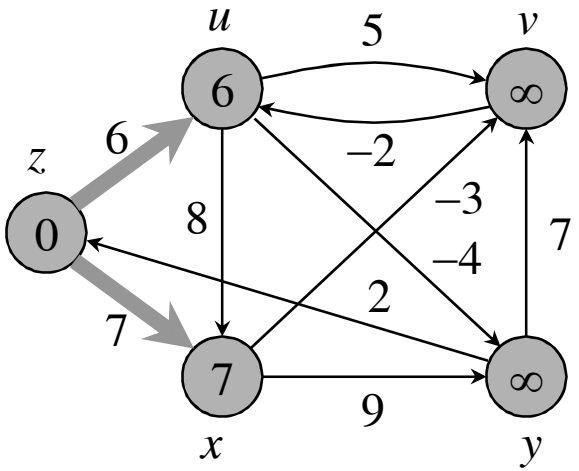
Cạnh (u, v) được sơn xám nếu $\pi[v] = u$

Ngay trước lượt đầu:



(a)

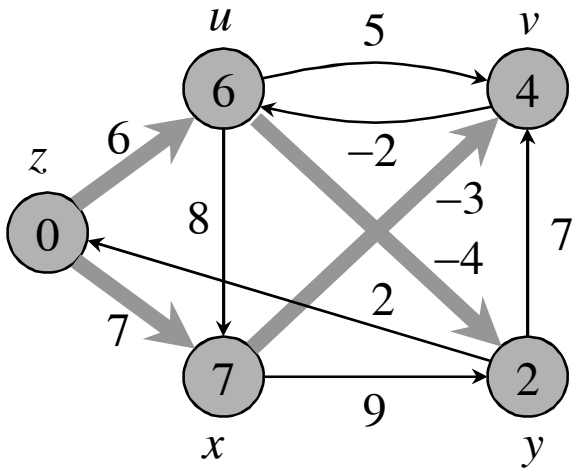
Ngay sau lượt đầu:



(b)

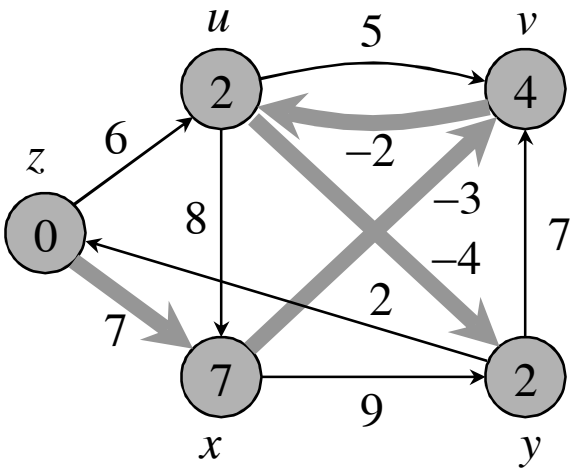
Thực thi giải thuật Bellman-Ford (tiếp)

Ngay sau lượt 2:



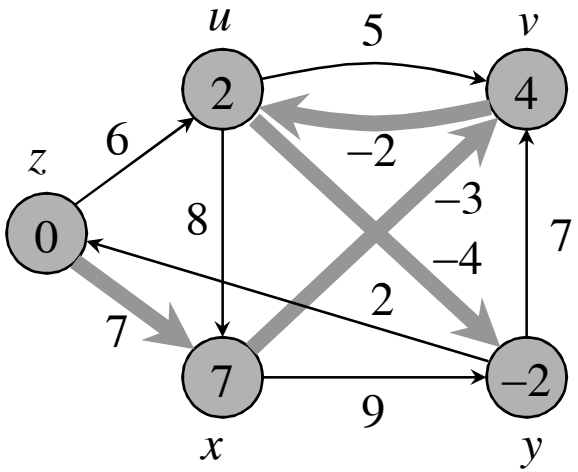
(c)

Ngay sau lượt 3:



(d)

Ngay sau lượt 4:



(e)

Tính đúng đắn của giải thuật Bellman-Ford

- ***Lemma 25.12***

  Cho

  – Đồ thị có trọng số và có hướng $G = (V, E)$, với hàm trọng số $w : E \to \Re$

  – Đỉnh nguồn s

  – G không chứa các chu trình có trọng số âm có thể đến được từ s.

  $\Rightarrow$

  Khi giải thuật BELLMAN-FORD thực thi xong thì $d[v] = \delta(s,v)$ cho mọi đỉnh v đến được từ s.

Tính đúng đắn của giải thuật Bellman-Ford (tiếp)

Chứng minh

Gọi v là một đỉnh đến được từ s. Gọi $p = <v_0, ..., v_k>$ là một đường đi ngắn nhất từ s đến v. Vì p là đường đi đơn nên $k \leq |V| - 1$.

Sẽ chứng minh: $d[v_i] = \delta(s, v_i)$ sau lượt nới lỏng thứ i, với $i = 0,...,k$, và đẳng thức được duy trì sau đó.

Dùng quy nạp:

- Cơ bản: $d[v_0] = \delta(s, v_0) = 0$ (vì $v_0 = s$)
- Giả thiết quy nạp: $d[v_{i-1}] = \delta(s, v_{i-1})$ sau lượt nới lỏng thứ $i - 1$.
- Bước quy nạp: Cạnh (v_{i-1}, v_i) được nới lỏng trong lượt thứ i nên $d[v_i] = \delta(s, v_i)$ sau lượt i và tại mọi thời điểm sau đó, theo Lemma 25.7.

Để ý là $k \leq |V| - 1$ và có $|V| - 1$ lượt nới lỏng.

Tính đúng đắn của giải thuật Bellman-Ford (tiếp)

- ***Hệ luận 25.13***

  Cho

  – Đồ thị có trọng số và có hướng $G = (V, E)$, với hàm trọng số $w : E \to \Re$

  – Đỉnh nguồn s

  $\Rightarrow$

  Với mọi đỉnh $v \in V$, tồn tại đường đi từ s đến v nếu và chỉ nếu BELLMAN-FORD hoàn tất với $d[v] < \infty$ khi nó thực thi trên G.

Tính đúng đắn của giải thuật của Bellman-Ford (tiếp)

- ***Định lý 25.14 (Tính đúng đắn của giải thuật của Bellman-Ford)***

  Thực thi BELLMAN-FORD lên đồ thị $G = (V, E)$ có trọng số và có hướng với
  - hàm trọng số $w : E \to \Re$
  - đỉnh nguồn s

  $\Rightarrow$

  (i) Nếu G không chứa chu trình có trọng số âm đến được từ s, thì
  (1) giải thuật trả về TRUE.
  (2) $d[v] = \delta(s,v)$ cho mọi đỉnh $v \in V$
  (3) đồ thị G_π là cây các đường đi ngắn nhất có gốc tại s.

  (ii) Nếu G chứa một chu trình có trọng số âm đến được từ s, thì giải thuật trả về FALSE.

Tính đúng đắn của giải thuật của Bellman-Ford (tiếp)

Chứng minh

(i) Giả sử G không chứa chu trình có trọng số âm đến được từ s.

– Nếu v đến được từ s thì Lemma 25.12 chứng minh (2).
  Nếu v không đến được từ s thì có (2) từ Hệ luận 25.6

– Lemma 25.9 cùng với (2) chứng minh (3).

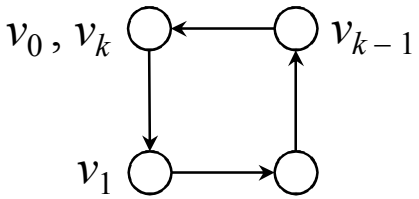
– Khi giải thuật hoàn tất, ta có với mọi cạnh (u, v):
  $d[v] = \delta(s,v) \le \delta(s,u) + w(u, v)$(từ Lemma 25.3)
  $= d[u] + w(u, v)$,
  vậy các test dòng 6 khiến giải thuật trả về TRUE, chứng minh (1).

Tính đúng đắn của giải thuật của Bellman-Ford

Chứng minh (tiếp)

(ii) Giả sử G chứa một chu trình có trọng số âm đến được từ s là

$c = \langle v_0, \ldots, v_k \rangle$ với $v_0 = v_k$



Vậy $\sum_{i=1\ldots k} w(v_{i-1}, v_i) < 0$ (*)

Chứng minh (ii) bằng phản chứng:

- Giả sử Bellman-Ford trả về TRUE, thì (dòng 5-8)

  $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$, với $i = 1,\ldots,k$

  Từ trên, lấy tổng,

  $\sum_{i=1\ldots k} d[v_i] \le \sum_{i=1\ldots k} d[v_{i-1}] + \sum_{i=1\ldots k} w(v_{i-1}, v_i)$ #

- Vì

  $\sum_{i=1\ldots k} d[v_i] = \sum_{i=1\ldots k} d[v_{i-1}]$, và

  $d[v_i] < \infty$ (Hệ luận 25.13), nên cùng với # ta có

  $0 \le \sum_{i=1\ldots k} w(v_{i-1}, v_i)$, mâu thuẫn với (*)!

Hình Học Tính Toán

Tính chất của đoạn thẳng

- ***Định nghĩa***
  - *Một tổ hợp lồi của hai điểm* khác nhau $p_1 = (x_1, y_1)$ và $p_2 = (x_2, y_2)$ là một điểm $p_3 = (x_3, y_3)$ sao cho

    $$x_3 = \alpha\, x_1 + (1 - \alpha)\, x_2$$

    $$y_3 = \alpha\, y_1 + (1 - \alpha)\, y_2$$

    $$0 \le \alpha \le 1 \quad .$$

  - *Đoạn thẳng* p_1p_2 là tập mọi tổ hợp lồi của p_1 và p_2, ký hiệu đt p_1p_2
  - Các *điểm đầu mút* của đoạn thẳng p_1p_2 là p_1 và p_2
  - *Đoạn thẳng có hướng* p_1p_2 là đoạn thẳng p_1p_2 được định hướng từ p_1 đến p_2, ký hiệu $p_1 \rightarrow p_2$.

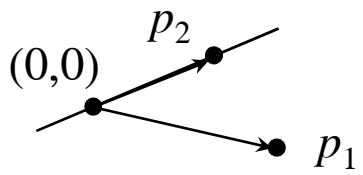
Tích chéo

- ***Định nghĩa*** *Tích chéo* của hai vectors $p_1 = (x_1, y_1)$ và $p_2 = (x_2, y_2)$ là

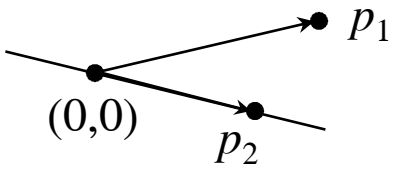
$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$$

$$= x_1 y_2 - x_2 y_1$$

- ***Nhận xét***
  - Nếu $p_1 \times p_2 > 0$ thì vectơ p_1 nằm theo chiều kim đồng hồ từ vectơ p_2 đối với (0, 0)

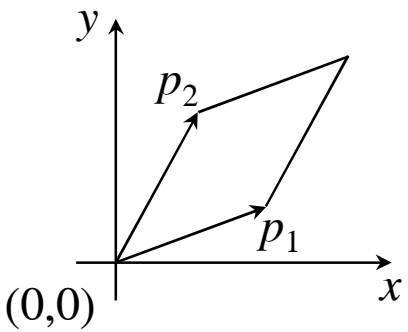


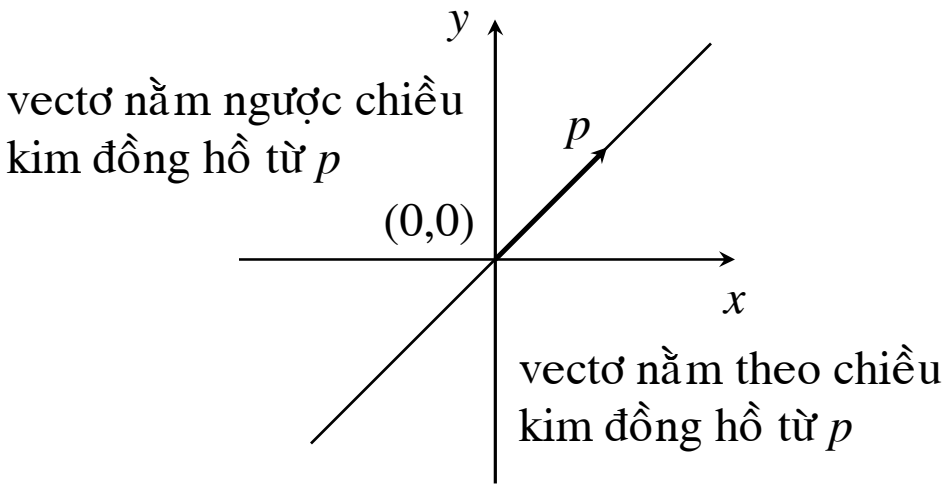
  - Nếu $p_1 \times p_2 < 0$ thì vectơ p_1 nằm ngược chiều kim đồng hồ từ vectơ p_2 đối với (0, 0)



  - Nếu $p_1 \times p_2 = 0$ thì O, p_1 và p_2 thẳng hàng.

Tích chéo (tiếp)





$|p_1 \times p_2|$ là diện tích của hình bình hành

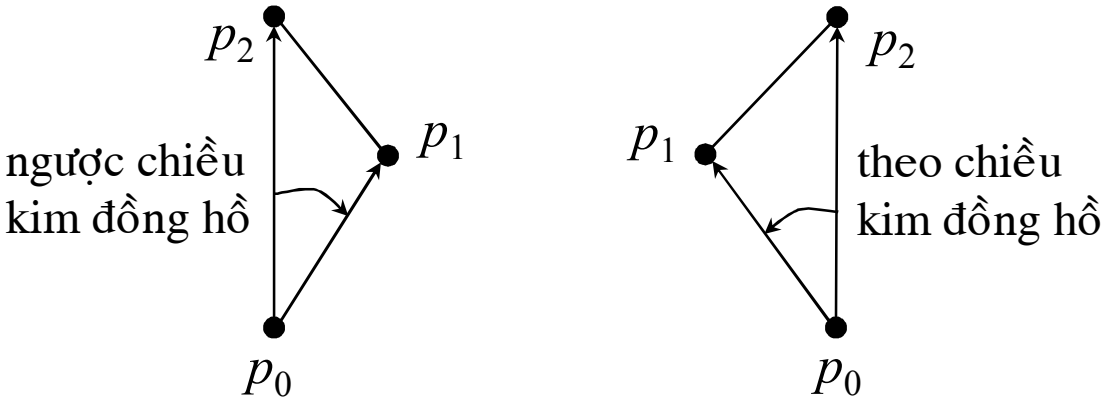
Tích chéo (tiếp)

- ***Nhận xét***

  Cho hai đoạn thẳng có hướng $p_0 \rightarrow p_1$ và $p_0 \rightarrow p_2$. Dùng phép tịnh tiến mà vectơ tịnh tiến là $-p_0$, ta thấy

  – Nếu $(p_1 - p_0) \times (p_2 - p_0) > 0$ thì $p_0 \rightarrow p_1$ nằm theo chiều kim đồng hồ từ $p_0 \rightarrow p_2$

  – Nếu $(p_1 - p_0) \times (p_2 - p_0) < 0$ thì $p_0 \rightarrow p_1$ nằm ngược chiều kim đồng hồ từ $p_0 \rightarrow p_2$.



Xác định hai đoạn thẳng có cắt nhau không

- ***Bài toán***

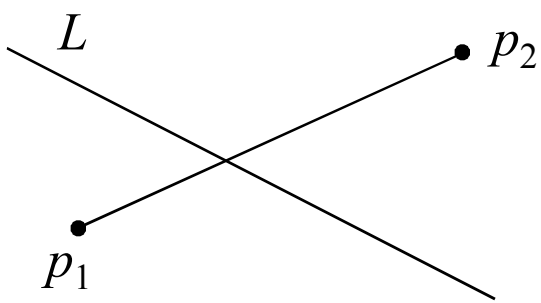
  Cho hai đoạn thẳng p_1p_2 và p_3p_4. Hỏi: Hai đoạn thẳng có cắt nhau không?

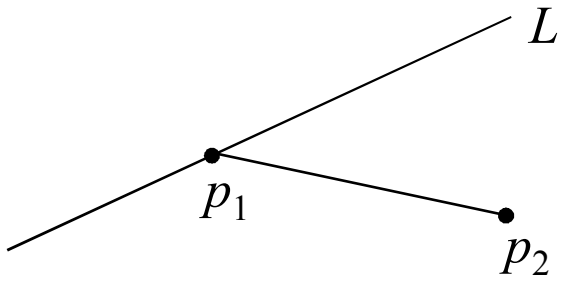
Hai cách giải quyết

- Cách giải 1: giải hệ thống phương trình bậc nhất để tìm tọa độ của điểm cắt (nếu có). Cách giải này cần dùng phép chia nên không chính xác khi tử số gần bằng 0.
- Cách giải 2: không cần dùng phép chia (xem slide tới).

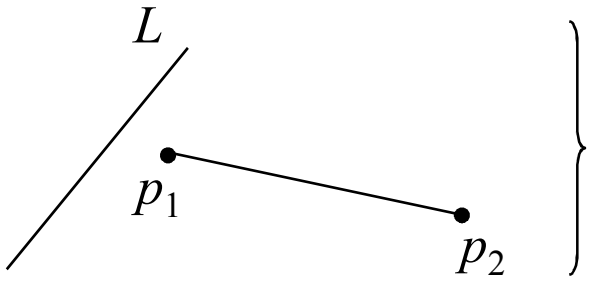
Xác định hai đoạn thẳng có cắt nhau không (tiếp)

– Định nghĩa: Một đoạn thẳng p_1p_2 *nằm hai bên* ("straddle") một đường thẳng nếu p_1 và p_2 nằm ở hai bên khác nhau của đường thẳng. (Trường hợp biên: p_1 hay p_2 nằm trên đường thẳng.)





đt p_1p_2 nằm hai bên đường thẳng L

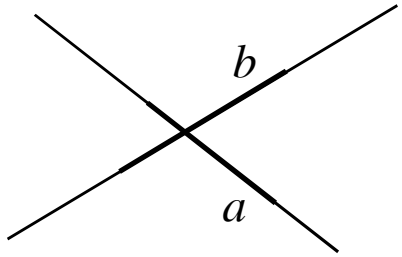


đt p_1p_2 không nằm hai bên đường thẳng L

Xác định hai đoạn thẳng có cắt nhau không (tiếp)

– **Định lý**: Hai đoạn thẳng cắt nhau nếu và chỉ nếu một trong các điều kiện sau (hoặc cả hai) là đúng.

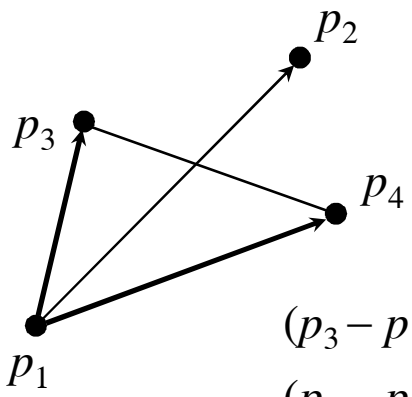
- 1. Mỗi đoạn thẳng nằm hai bên đường thẳng chứa đoạn thẳng kia.
- 2. Một điểm đầu mút (điểm cuối) của đoạn thẳng này nằm trên đoạn thẳng kia.



Đoạn thẳng *a* nằm hai bên đường thẳng chứa *b*, và đoạn thẳng *b* nằm hai bên đường thẳng chứa *a*

Xác định hai đoạn thẳng có cắt nhau không (tiếp)

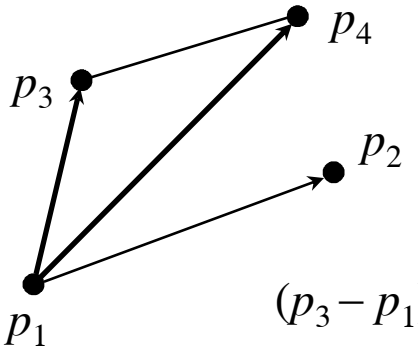
Dùng tích chéo để xác định một đoạn thẳng có nằm hai bên một đường thẳng hay không.



$(p_3 - p_1) \times (p_2 - p_1) < 0$

$(p_4 - p_1) \times (p_2 - p_1) > 0$

Các tích chéo $(p_3 - p_1) \times (p_2 - p_1)$ và $(p_4 - p_1) \times (p_2 - p_1)$ có dấu khác nhau, do đó đt $p_3 p_4$ nằm hai bên đường thẳng chứa đt $p_1 p_2$ (và ngược lại)

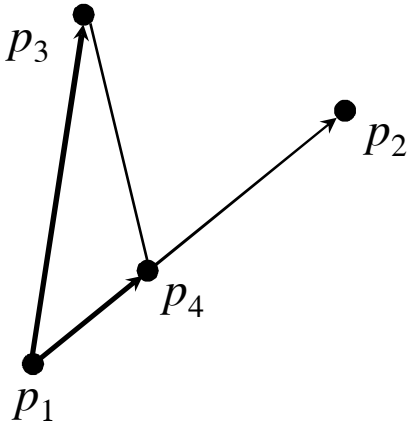


$(p_3 - p_1) \times (p_2 - p_1) < 0$

$(p_4 - p_1) \times (p_2 - p_1) < 0$

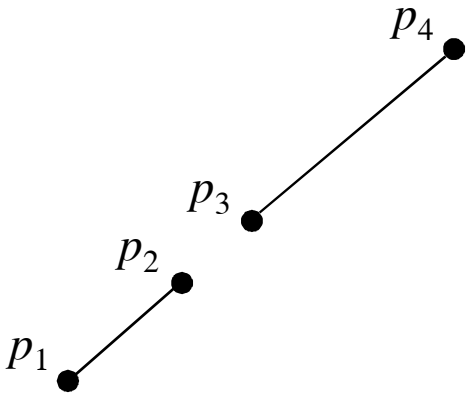
Các tích chéo $(p_3 - p_1) \times (p_2 - p_1)$ và $(p_4 - p_1) \times (p_2 - p_1)$ có cùng dấu, do đó đt $p_3 p_4$ không nằm hai bên đường thẳng chứa đt $p_1 p_2$ (và ngược lại)

Xác định hai đoạn thẳng có cắt nhau không (tiếp)



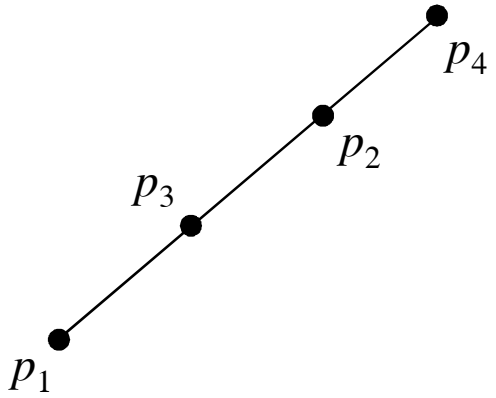
$$(p_3 - p_1) \times (p_2 - p_1) < 0$$

$$(p_4 - p_1) \times (p_2 - p_1) = 0$$



$$(p_4 - p_1) \times (p_2 - p_1) = 0$$

$$(p_3 - p_1) \times (p_2 - p_1) = 0$$



Xác định hai đoạn thẳng có cắt nhau không (tiếp)

– Thủ tục để kiểm tra hai đoạn thẳng p_1p_2 và p_3p_4 có cắt nhau không (mã giả). Thủ tục trả về TRUE nếu hai đoạn thẳng cắt nhau và trả về FALSE nếu chúng không cắt nhau.

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
1   d1 ← DIRECTION(p3, p4, p1)
2   d2 ← DIRECTION(p3, p4, p2)
3   d3 ← DIRECTION(p1, p2, p3)
4   d4 ← DIRECTION(p1, p2, p4)
5   if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
       ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
6      then return TRUE
```

(xem tiếp slide tới)

Xác định hai đoạn thẳng có cắt nhau không (tiếp)

(tiếp)

```
 7    elseif d1 = 0 and ON-SEGMENT(p3, p4, p1)
 8        then return TRUE
 9    elseif d2 = 0 and ON-SEGMENT(p3, p4, p2)
10        then return TRUE
11    elseif d3 = 0 and ON-SEGMENT(p1, p2, p3)
12        then return TRUE
13    elseif d4 = 0 and ON-SEGMENT(p1, p2, p4)
14        then return TRUE
15    else return FALSE
```

Xác định hai đoạn thẳng có cắt nhau không (tiếp)

Thủ tục ON-SEGMENT

Input: p_i , p_j , p_k , mà p_k thẳng hàng với đoạn $p_i p_j$

Output: TRUE nếu p_k nằm trên đoạn $p_i p_j$

FALSE nếu p_k nằm ngoài đoạn $p_i p_j$

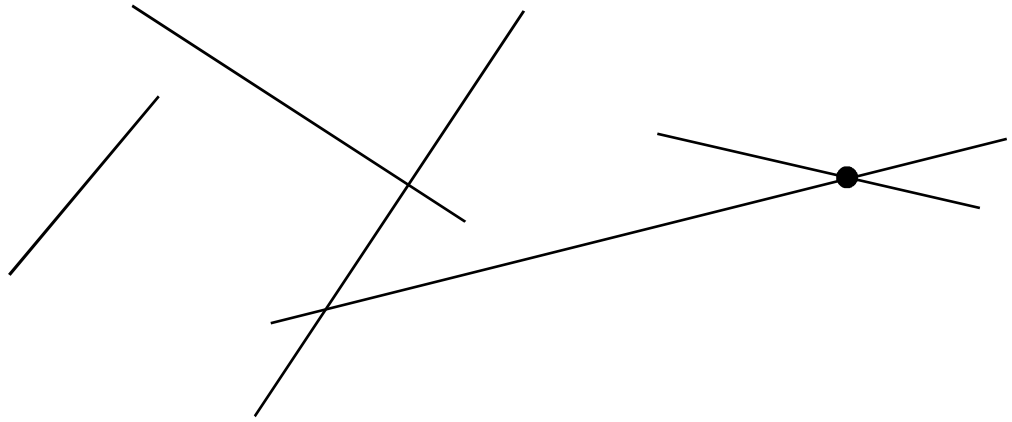
```
DIRECTION(p_i , p_j , p_k )
1   return (p_k − p_i ) × (p_j − p_i )

ON-SEGMENT(p_i , p_j , p_k )
1   if min(x_i , x_j ) ≤ x_k ≤ max(x_i , x_j ) and min(y_i , y_j ) ≤ y_k ≤ max(y_i , y_j )
2      then return TRUE
3      else return FALSE
```

Hình Học Tính Toán

35.2 Xác định có cặp đoạn thẳng nào cắt nhau không

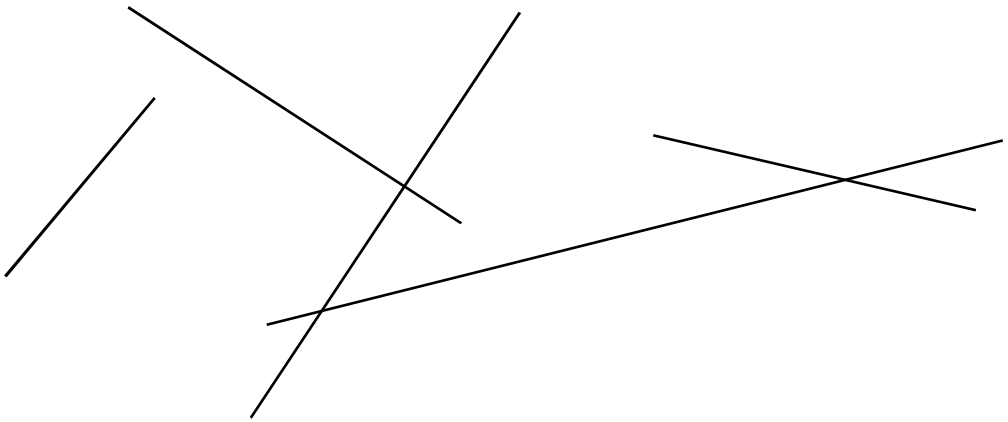
Bài toán: Cho tập các đoạn thẳng trong mặt phẳng. Xác định có cặp đoạn thẳng nào cắt nhau hay không.



- Để đơn giản, giả sử:
  - Không có đoạn thẳng nào là thẳng đứng
  - Không có ba đoạn thẳng nào cắt nhau tại một điểm chung.

Giải thuật thô sơ

- Giải thuật thô sơ: Kiểm tra xem mỗi cặp đoạn thẳng có cắt nhau hay không. Thời gian chạy là $\Theta(n^2)$, với n là số các đoạn thẳng.

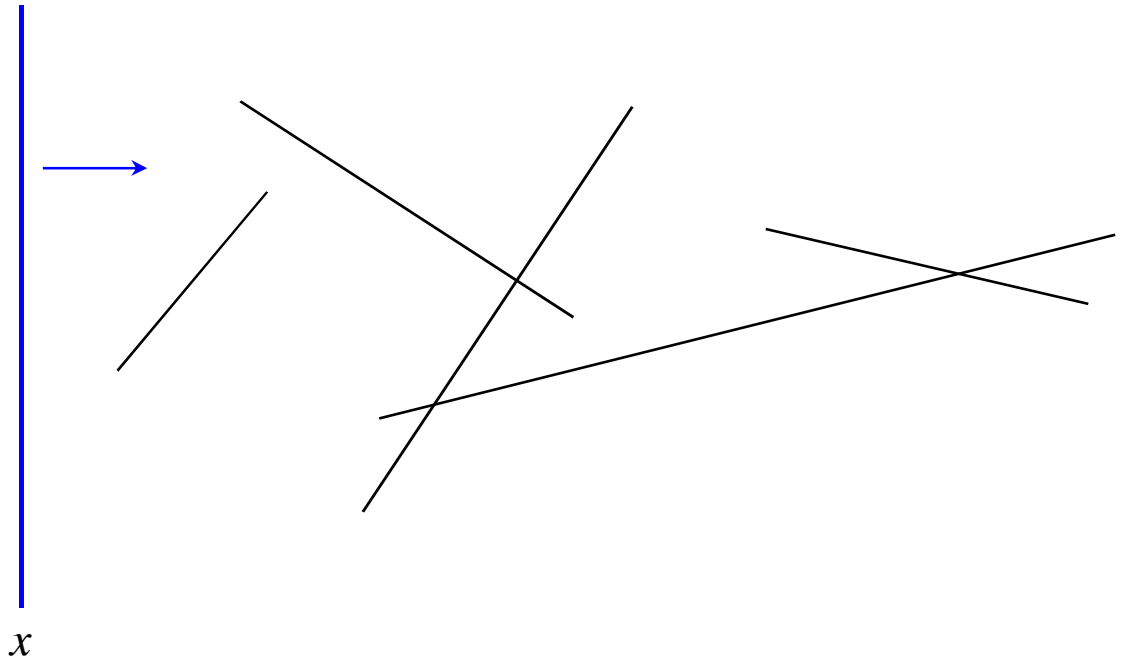


Kỹ thuật quét

- Giải thuật hữu hiệu dùng *kỷ thuật quét* (sweeping):

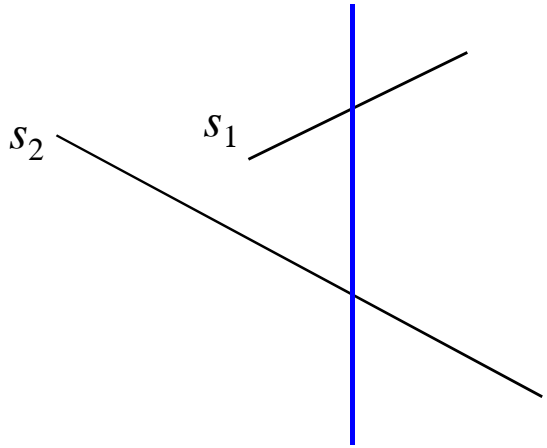
  Dùng một đường thẳng thẳng đứng quét từ trái sang phải và xem xét các thay đổi của phần giao của *đường thẳng quét* với các đoạn thẳng.

  - *Đường thẳng quét* (sweep line)
    - Đường thẳng quét thẳng đứng, vị trí hiện thời là toạ độ x



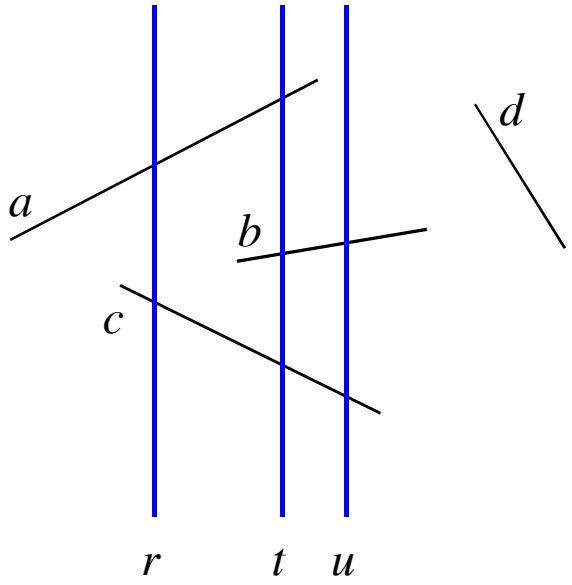
Thứ tự các đoạn thẳng

- Định nghĩa một thứ tự hoàn toàn trên các đoạn thẳng cắt bởi đường thẳng quét.
  - Hai đoạn thẳng s_1 và s_2 không cắt nhau là *có thể so sánh được* tại x nếu đường thẳng quét tại vị trí x cắt cả hai đoạn thẳng đó.



  - Nếu s_1 và s_2 là có thể so sánh được tại x và giao điểm của s_1 với đường thẳng quét ở cao hơn giao điểm của s_2 với cùng đường thẳng quét đó, thì ta nói s_1 *ở trên* s_2 , ký hiệu $s_1 >_x s_2$.

Thứ tự các đoạn thẳng (tiếp)



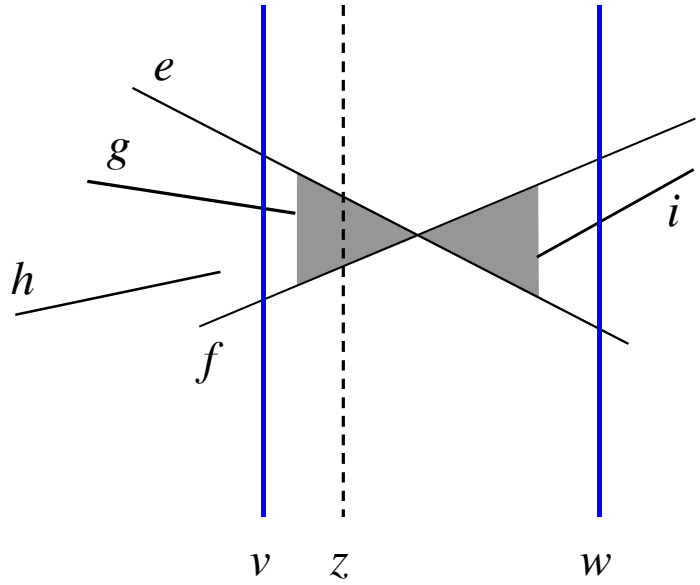
(a)

$a >_r c$ $a >_t b$ $b >_u c$

$b >_t c$

$a >_t c$

Đoạn thẳng *d* <u>không so sánh được</u> với các đoạn thẳng khác trong hình (a).



(b)

$e >_v f$ nhưng $f >_w e$

Mọi đường thẳng quét mà đi qua vùng xám đều có các đoạn thẳng *e* và *f* ở liên tiếp nhau trong quan hệ thứ tự của nó

Các cấu trúc dữ liệu trong kỹ thuật quét

- Đường thẳng quét
  - Khi di chuyển đường thẳng quét, giải thuật trữ và duy trì các thông tin sau
    - *Tình trạng của đường thẳng quét* (sweep-line status): cho biết thứ tự giữa các đối tượng (đoạn thẳng) bị cắt bởi đường thẳng quét với nhau
    - *Lịch của các biến cố* (event-point schedule): dãy các tọa độ x, sắp từ trái sang phải, xác định các vị trí dừng của đường thẳng quét.

Các thao tác lên sweep-line status

- Chi tiết giải thuật hữu hiệu dùng kỷ thuật quét
  - Đường thẳng quét
    - Khi di chuyển đường thẳng quét, giải thuật trữ và duy trì các thông tin sau
      - *Tình trạng của đường thẳng quét* (sweep-line status): Các thao tác lên T:
        - INSERT(T, s): chèn đoạn thẳng s vào T
        - DELETE(T, s): xoá đoạn thẳng s khỏi T
        - ABOVE(T, s): trả về đoạn thẳng ở ngay trên s trong T
        - BELOW(T, s): trả về đoạn thẳng ở ngay dưới s trong T.

Event-point schedule

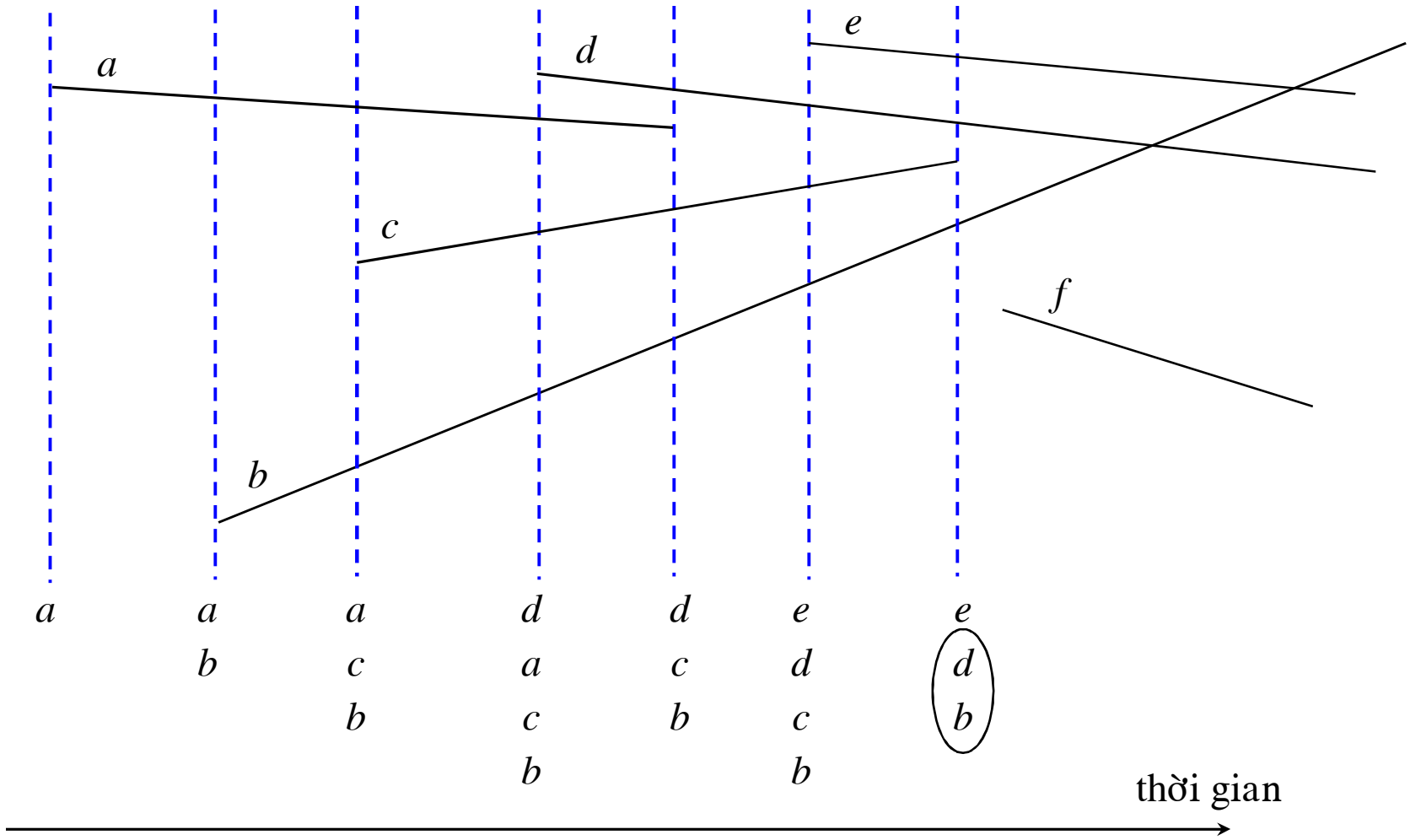
– *Lịch của các biến cố* (event-point schedule): dãy các tọa độ x, sắp từ trái sang phải, xác định các vị trí dừng của đường thẳng quét.

- Mỗi điểm đầu mút của các đoạn thẳng (của tập input S) là một *điểm biến cố* (event point), là điểm mà thứ tự T thay đổi.
- Lịch của các biến cố là tĩnh và được xây dựng bằng cách sắp xếp các điểm đầu mút của các đoạn thẳng theo thứ tự từ trái qua phải.

Xác định có cặp đoạn thẳng nào cắt nhau không

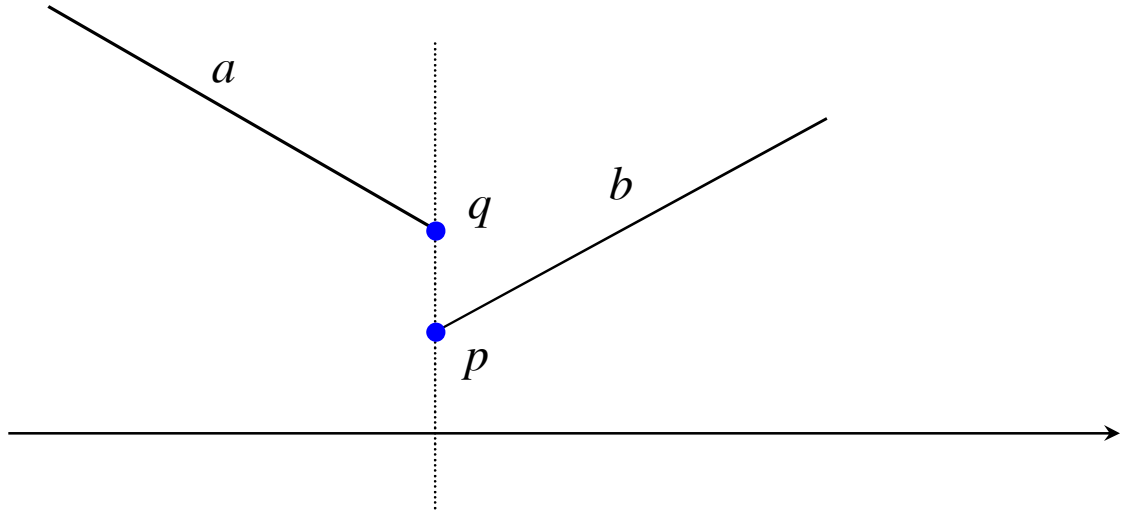
```
ANY-SEGMENTS-INTERSECT(S)
1  T ← ∅
2  Sắp các điểm đầu mút của các đoạn thẳng trong S theo
         thứ tự từ trái sang phải, breaking ties...
3  for mỗi điểm p trong danh sách sắp xếp của các điểm đầu mút
4      do if p là điểm đầu mút bên trái của đoạn thẳng s
5           then INSERT(T, s)
6                if (ABOVE(T, s) tồn tại và cắt s)
                      hay (BELOW(T, s) tồn tại và cắt s)
7                  then return TRUE
8         if p là điểm đầu mút bên phải của đoạn thẳng s
9           then if cả hai ABOVE(T, s) và BELOW(T, s) đều tồn tại
                      và ABOVE(T, s) cắt BELOW(T, s)
10               then return TRUE
11               DELETE(T, s)
12 return FALSE
```

Thực thi ANY-SEGMENTS-INTERSECT



Breaking ties

- Nếu khi sắp xếp các điểm đầu mút của các đoạn thẳng trong S từ trái sang phải mà có nhiều điểm có cùng tọa độ x thì breaking ties như sau
  - Các điểm đầu mút bên trái được xếp trước các điểm đầu mút bên phải.



p được xếp trước q khi sắp xếp các điểm đầu mút ở dòng 2 của ANY-SEGMENTS-INTERSECT

Tính đúng đắn

- ***Theorem 35.1 (Tính đúng đắn)***

  Giải thuật ANY-SEGMENTS-INTERSECT chạy trên tập S trả về TRUE nếu và chỉ nếu có cắt nhau giữa các đoạn thẳng.

- ***Chứng minh***

  "$\Rightarrow$": xem mã ta thấy ANY-SEGMENTS-INTERSECT trả về TRUE chỉ khi nào nó tìm thấy hai đoạn thẳng cắt nhau.

  "$\Leftarrow$": Sẽ chứng minh rằng nếu tồn tại hai đoạn thẳng cắt nhau thì ANY-SEGMENTS-INTERSECT trả về TRUE.

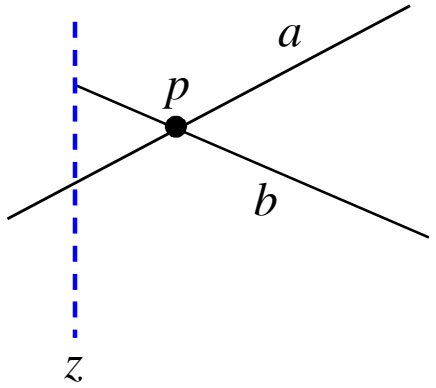
Tính đúng đắn (tiếp)

Giả sử tồn tại một giao điểm.

Gọi p là giao điểm bên trái nhất, gọi a và b là các đoạn thẳng cắt nhau tại p.

Tồn tại đường quét z mà tại đó a và b trở nên liên tiếp nhau trong thứ tự toàn phần.

Tồn tại điểm đầu mút q mà là event point để cho a và b trở nên liên tiếp nhau trong thứ tự toàn phần.

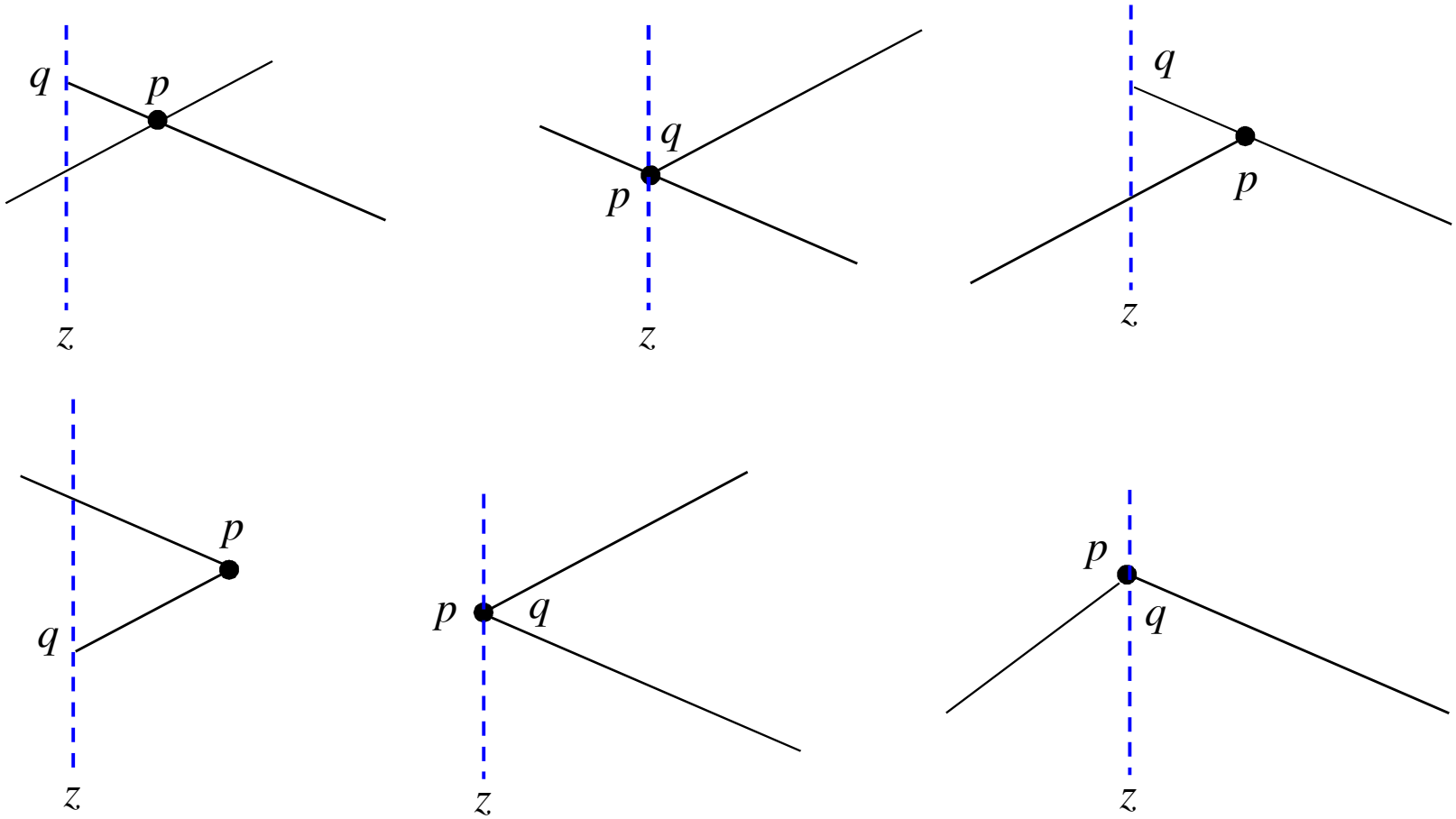


Có 2 trường hợp: A) giải thuật xử lý q và B) giải thuật không xử lý q.

Tính đúng đắn (tiếp)

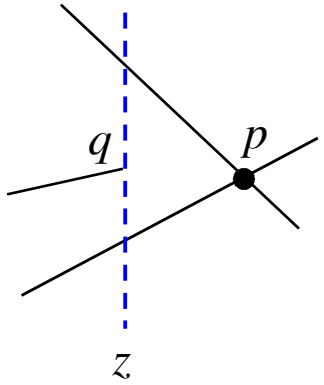
A)

Trường hợp 1: đoạn thẳng *a* hay *b* được chèn vào *T*, và đoạn thẳng kia ở trên hay dưới nó. Các dòng 4-7 tìm thấy trường hợp này.



Tính đúng đắn (tiếp)

Trường hợp 2: các đoạn thẳng *a* và *b* đang trong *T*, và một đoạn thẳng ở giữa chúng được xóa. Các dòng 8-11 tìm thấy trường hợp này.



Trong cả hai trường hợp, giải thuật tìm thấy *p* và trả về TRUE.

B)

Nếu *q* không được giải thuật xử lý, thì có nghĩa là giải thuật đã quay về trước khi xử lý xong tất cả các event points. Vậy giải thuật đã tìm thấy một giao điểm và trả về TRUE.

Phân tích ANY-SEGMENTS-INTERSECT

- ***Thời gian chạy***
  - Giả sử tập đoạn thẳng S gồm có n đoạn thẳng. Dùng cấu trúc dữ liệu thích hợp (ví dụ: dựa trên cây nhị phân cân bằng) để hiện thực T sao cho các thao tác lên T đều tốn $O(\lg n)$ thời gian.
  - Thời gian chạy của giải thuật ANY-SEGMENTS-INTERSECT gồm
    - Dòng 1: $O(1)$ thời gian
    - Dòng 2: $O(n \lg n)$ thời gian
    - Vòng lặp **for**: $O(n \lg n)$ thời gian

  Vậy thời gian chạy tổng cộng của giải thuật là $O(n \lg n)$.

35.4 Tìm bao lồi

- Tự đọc.

35.4 Tìm cặp điểm gần nhau nhất

- Tự đọc.

NP-Đầy Đủ

Vài khái niệm cơ bản

- Bài toán
  - các tham số
  - các tính chất mà lời giải cần phải thỏa mãn
- Một thực thể (instance) của bài toán là bài toán mà các tham số có trị cụ thể.

Hình thức hóa khái niệm bài toán

- Ví dụ: bài toán SHORTEST-PATH là
  - "không hình thức": bài toán tìm đường đi ngắn nhất giữa hai đỉnh cho trước trong một đồ thị vô hướng, không có trọng số $G = (V, E)$.
  - "hình thức":
    - Một thực thể của bài toán là một cặp ba gồm một đồ thị cụ thể và hai đỉnh cụ thể.
    - Một lời giải là một dãy các đỉnh của đồ thị .
    - Bài toán SHORTEST-PATH là quan hệ kết hợp mỗi thực thể gồm một đồ thị và hai đỉnh với một đường đi ngắn nhất (nếu có) trong đồ thị nối hai đỉnh:

      SHORTEST-PATH $\subseteq I \times S$

Bài toán trừu tượng

- Định nghĩa: một *bài toán trừu tượng* Q là một quan hệ nhị phân trên một tập I, được gọi là tập các *thực thể* (instances) của bài toán, và một tập S, được gọi là tập các *lời giải* của bài toán:

  $$Q \subseteq I \times S$$

Bài toán quyết định

- *Một bài toán quyết định* Q là một bài toán trừu tượng mà quan hệ nhị phân Q là một hàm từ I đến $S = \{0, 1\}$, 0 tương ứng với "no", 1 tương ứng với "yes".

- Ví dụ: bài toán quyết định PATH là

  Cho một đồ thị $G = (V, E)$, hai đỉnh $u, v \in V$, và một số nguyên dương k.

  Đặt $i = \langle G, u, v, k \rangle$, một thực thể của bài toán quyết định PATH,

  - PATH(i) = 1 (yes) nếu tồn tại một đường đi giữa u và v có chiều dài $\leq k$
  - PATH(i) = 0 (no) trong các trường hợp khác.

Bài toán tối ưu

- *Một bài toán tối ưu* là một bài toán trong đó ta cần xác định trị lớn nhất hay trị nhỏ nhất của một đại lượng.
- Đối tượng của lý thuyết NP-đầy đủ là các bài toán quyết định, nên ta phải ép (recast) các bài toán tối ưu thành các bài toán quyết định.

  Ví dụ: ta đã ép bài toán tối ưu đường đi ngắn nhất thành bài toán quyết định PATH bằng cách làm chận k thành một tham số của bài toán.

Mã hoá (encodings)

- Để một chương trình máy tính giải một bài toán trừu tượng thì các thực thể của bài toán cần được biểu diễn sao cho chương trình máy tính có thể đọc và "hiểu" chúng được.
- Ta mã hóa (encode) các thực thể của một bài toán trừu tượng để một chương trình máy tính có thể đọc chúng được.
  - Ví dụ: Mã hoá tập $\mathbf{N} = \{0, 1, 2, 3, 4, \ldots\}$ thành tập các chuỗi $\{0, 1, 10, 11, 100, \ldots\}$. Trong mã hoá này, $e(17) = 10001$.
  - Mã hóa một đối tượng đa hợp (chuỗi, tập, đồ thị,...) bằng cách kết hợp các mã hóa của các thành phần của nó.

Mã hoá (tiếp)

- Một *bài toán cụ thể* là một bài toán mà tập các thực thể của nó là tập các chuỗi nhị phân.
- Một giải thuật *giải* một bài toán cụ thể trong thời gian $O(T(n))$ nếu, khi đưa nó một thực thể i có độ dài $n = |i|$, thì nó sẽ cho ra lời giải trong thời gian $O(T(n))$
- Một bài toán cụ thể là *có thể giải được trong thời gian đa thức* nếu tồn tại một giải thuật giải nó trong thời gian $O(n^k)$ với một hằng số k nào đó.

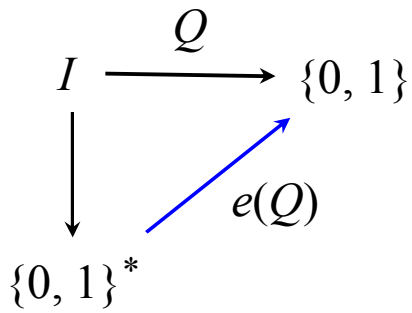
Lớp P

- Định nghĩa: ***Lớp P*** (complexity class P) là tập các bài toán quyết định cụ thể có thể giải được trong thời gian đa thức.

Bài toán trừu tượng và bài toán cụ thể

- Ta dùng mã hoá để ánh xạ các bài toán trừu tượng đến các bài toán cụ thể.
  - Cho một bài toán quyết định trừu tượng Q, Q ánh xạ một tập các thực thể I đến $\{0, 1\}$, ta có thể dùng một mã hóa $e : I \rightarrow \{0, 1\}^*$ để sinh ra một bài toán quyết định cụ thể tương ứng, ký hiệu $e(Q)$.

    Mã hóa e phải thõa điều kiện
    - Nếu $Q(i) \in \{0, 1\}$ là lời giải cho $i \in I$, thì lời giải cho thực thể $e(i) \in \{0, 1\}^*$ của bài toán quyết định cụ thể $e(Q)$ cũng là $Q(i)$.



Các mã hoá

- Một hàm $f : \{0, 1\}^* \to \{0, 1\}^*$ là *có thể tính được trong thời gian đa thức* nếu tồn tại một giải thuật thời gian đa thức A sao cho, với mọi input $x \in \{0, 1\}^*$, A cho ra output là $f(x)$.
- Cho I là một tập các thực thể của một bài toán, ta nói rằng hai mã hoá e_1 và e_2 là *có liên quan đa thức* nếu tồn tại hai hàm có thể tính được trong thời gian đa thức f_{12} và f_{21} sao cho với mọi $i \in I$ ta có $f_{12}(e_1(i)) = e_2(i)$ và $f_{21}(e_2(i)) = e_1(i)$.

Liên quan giữa các mã hóa

- ***Lemma 36.1***
- Cho Q là một bài toán quyết định trừu tượng trên một tập các thực thể I, và cho e_1 và e_2 là các mã hoá trên I có liên quan đa thức
- $\Rightarrow$
- $e_1(Q) \in \mathrm{P} \Leftrightarrow e_2(Q) \in \mathrm{P}$.

- Theo Lemma trên, "độ phức tạp" của một bài toán trừu tượng mà các thực thể của nó được mã hóa trong cơ số 2 hay 3 thì như nhau.
- Yêu cầu: sẽ chỉ dùng các mã hóa mà liên quan đa thức với "mã hóa chuẩn".

Mã hóa chuẩn (standard encoding)

- *Mã hóa chuẩn*

  ánh xạ các thực thể vào các "chuỗi có cấu trúc" trên tập các ký tự $\Psi = \{0, 1, -, [,], (,), ,\}$.

  Các *chuỗi có cấu trúc* (structured string) được định nghĩa đệ quy. Ở đây chỉ trình bày vài ví dụ

  – Số nguyên 13 được biểu diễn bởi chuỗi có cấu trúc 1101.
  – Số nguyên –13 được biểu diễn bởi chuỗi có cấu trúc –1101.
  – Chuỗi [1101] là một chuỗi có cấu trúc có thể dùng làm "tên" (ví dụ, cho một phần tử của một tập, một đỉnh trong một đồ thị,...)

Mã hóa chuẩn (tiếp)

- Tập {a, b, c, d} có thể được biểu diễn bởi chuỗi có cấu trúc ([0], [1], [10], [11])
- Đồ thị

có thể được biểu diễn bởi chuỗi có cấu trúc

(([0], [1], [10]), (([0], [1]), ([1], [10])))

tập các đỉnh — tập các cạnh

- Mã hóa chuẩn của một đối tượng *D* được ký hiệu là <*D*>.

Một khung ngôn ngữ hình thức

- Một *bảng chữ cái* Σ là một tập hữu hạn các ký hiệu.
- Một *ngôn ngữ* L trên Σ là một tập các chuỗi tạo bởi các ký hiệu từ Σ.
  - Ví dụ: nếu $\Sigma = \{0, 1\}$, thì $L = \{10, 11, 101, 111, 1011,...\}$ là ngôn ngữ của các biểu diễn nhị phân của các số nguyên tố.
  - *Chuỗi rỗng* được ký hiệu là ε, *ngôn ngữ rỗng* được ký hiệu là $\varnothing$.
- Ngôn ngữ của tất cả các chuỗi trên Σ được ký hiệu là Σ^* .
  - Ví dụ: nếu $\Sigma = \{0, 1\}$, thì $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000,\ldots\}$ là tập tất cả các chuỗi nhị phân.
  - Mỗi ngôn ngữ L trên Σ đều là một tập con của Σ^* .
  - *Hợp* và *giao* của các ngôn ngữ được định nghĩa giống như trong lý thuyết tập hợp
  - *Phần bù* của L là $\overline{L} = \Sigma^* - L$.

Bài toán quyết định và ngôn ngữ tương ứng

- Đồng nhất một bài toán quyết định với một ngôn ngữ:
  - Tập các thực thể cho bất kỳ bài toán quyết định Q nào là tập Σ^*. Vì Q là hoàn toàn được đặc trưng bởi tập của tất cả các thực thể nào của nó mà lời giải là 1 (yes), nên có thể xem Q như là một ngôn ngữ L trên $\Sigma = \{0, 1\}$, với

$$L = \{x \in \Sigma^* : Q(x) = 1\}$$

Bài toán quyết định và ngôn ngữ tương ứng (tiếp)

– Ví dụ: bài toán quyết định PATH là ngôn ngữ

$\{\langle G, u, v, k\rangle : G = (V, E)$ là một đồ thị vô hướng,
$u, v \in V$,
$k \geq 0$ là một số nguyên, và tồn tại một
đường đi giữa u và v trong G mà chiều dài $\leq k\}$

Ta viết:

PATH $= \{\langle G, u, v, k\rangle : G = (V, E)$ là một đồ thị vô hướng,
$u, v \in V$,
$k \geq 0$ là một số nguyên, và tồn tại một
đường đi giữa u và v trong G mà chiều dài
$\leq k\}$

Ngôn ngữ và giải thuật

- Một giải thuật A *chấp nhận* (accept) một chuỗi $x \in \{0, 1\}^*$ nếu, với input là x, A outputs $A(x) = 1$.
- Một giải thuật A *từ chối* (reject) một chuỗi $x \in \{0, 1\}^*$ nếu $A(x) = 0$.
- Ngôn ngữ *được chấp nhận bởi* một giải thuật A là tập các chuỗi $L = \{x \in \{0, 1\}^* : A(x) = 1\}$.
- Một ngôn ngữ L *được quyết định bởi* một giải thuật A nếu
  - mọi chuỗi nhị phân trong L được chấp nhận bởi A và
  - mọi chuỗi nhị phân không trong L được từ chối bởi A.

Chấp nhận và quyết định ngôn ngữ trong thời gian đa thức

- Một ngôn ngữ L *được chấp nhận trong thời gian đa thức bởi* một giải thuật A nếu
- 1. nó được chấp nhận bởi A và nếu
- 2. có một hằng số k sao cho với mọi chuỗi $x \in L$ có độ dài n thì A chấp nhận x trong thời gian $O(n^k)$.
- Một ngôn ngữ L *được quyết định trong thời gian đa thức bởi* một giải thuật A nếu có một hằng số k sao cho với mọi chuỗi $x \in \{0, 1\}^*$ có chiều dài n thì A quyết định chính xác x có trong L hay không trong thời gian $O(n^k)$.

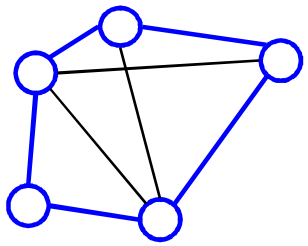
Lớp P

- Một định nghĩa khác của lớp P:
- P = $\{L \subseteq \{0, 1\}^*$: tồn tại một giải thuật A quyết định L trong thời gian đa thức$\}$
- ***Định lý 36.2***
- P = $\{L$: L được chấp nhận bởi một giải thuật chạy trong thời gian đa thức$\}$

Chứng thực trong thời gian đa thức

Bài toán chu trình Hamilton

- Một *chu trình hamilton* của một đồ thị vô hướng $G = (V, E)$ là một chu trình đơn chứa mỗi đỉnh trong V đúng một lần.



- Một đồ thị được gọi là *hamilton* nếu nó chứa một chu trình hamilton, và được gọi là *không hamilton* trong các trường hợp khác.
- *Bài toán chu trình Hamilton* là "Đồ thị G có một chu trình hamilton không?" Bài toán này dưới dạng một ngôn ngữ hình thức:
- HAM-CYCLE = $\{\langle G\rangle : G$ là một đồ thị hamilton$\}$.

Chứng thực trong thời gian đa thức (tiếp)

- Làm thế nào để một giải thuật quyết định được ngôn ngữ HAM-CYCLE?
  - Cho một thực thể $<G>$ của bài toán, a possible decision algorithm liệt kê tất cả các giao hoán của các đỉnh của G và kiểm tra mỗi giao hoán có là một chu trình hamilton hay không.
  - Thời gian chạy của giải thuật trên?
    - Giả sử mã hóa một đồ thị bằng ma trận kề của nó, thì số các đỉnh của nó là $m = \Omega(\sqrt{n})$, với $n = |<G>|$ là chiều dài của mã hóa của G.
    - Có $m!$ giao hoán của các đỉnh nên thời gian chạy là

      $$\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$$

      Giải thuật không chạy trong thời gian đa thức.

Kiểm tra trong thời gian đa thức

Bài toán chu trình Hamilton (tiếp)

- Xét một bài toán đơn giản hơn: cho một đường đi (một danh sách các đỉnh) trong một đồ thị $G = (V, E)$, kiểm tra xem nó có phải là một chu trình hamilton hay không.
  - Giải thuật:
    - kiểm tra các đỉnh trên đường đi đã cho có phải là một giao hoán của các đỉnh của V hay không.
    - kiểm tra các cạnh trên đường đi có thực sự là các cạnh của E và tạo nên một chu trình hay không.
  - Thời gian chạy: $O(n^2)$.

Giải thuật chứng thực

- Ta định nghĩa một *giải thuật chứng thực* (verification algorithm) là một giải thuật A có hai đối số (two-argument algorithm), trong đó một đối số là một chuỗi input thông thường x và đối số kia là một chuỗi nhị phân y, y được gọi là một *chứng thư* (certificate).
- *Ngôn ngữ được chứng thực* bởi một giải thuật chứng thực A là
- $L = \{x \in \{0, 1\}^* : \text{tồn tại } y \in \{0, 1\}^* \text{ sao cho } A(x, y) = 1\}$.
  - Ví dụ: Trong bài toán chu trình hamilton, chứng thư là danh sách của các đỉnh trong chu trình hamilton.

Lớp NP

- ***Lớp NP*** (NP: "nondeterministic polynomial time") là lớp các ngôn ngữ có thể được chứng thực bởi một giải thuật thời gian đa thức. Chính xác hơn:

  Cho một ngôn ngữ L.

- Ngôn ngữ L thuộc về NP
- $\Leftrightarrow$
- Tồn tại một giải thuật thời gian đa thức hai đối số A cùng với một hằng số c sao cho
- $L = \{x \in \{0, 1\}^* :$ tồn tại một chứng thư y với độ dài $|y| = O(|x|^c)$ sao cho $A(x, y) = 1\}$
- Ta nói rằng giải thuật A *chứng thực* ngôn ngữ L *trong thời gian đa thức*.

Lớp NP

– Ví dụ: HAM-CYCLE $\in$ NP.

Tính có thể rút gọn được (reducibility)

- Làm thế nào để so sánh "độ khó" của các bài toán?
- Ví dụ
  - Bài toán Q: Giải phương trình bậc nhất $ax + b = 0$
  - Bài toán Q': Giải phương trình bậc hai $px^2 + qx + r = 0$
- Giải phương trình bậc nhất $ax + b = 0$ bằng cách giải phương trình bậc hai: $0x^2 + ax + b = 0$.

  Ta nói:

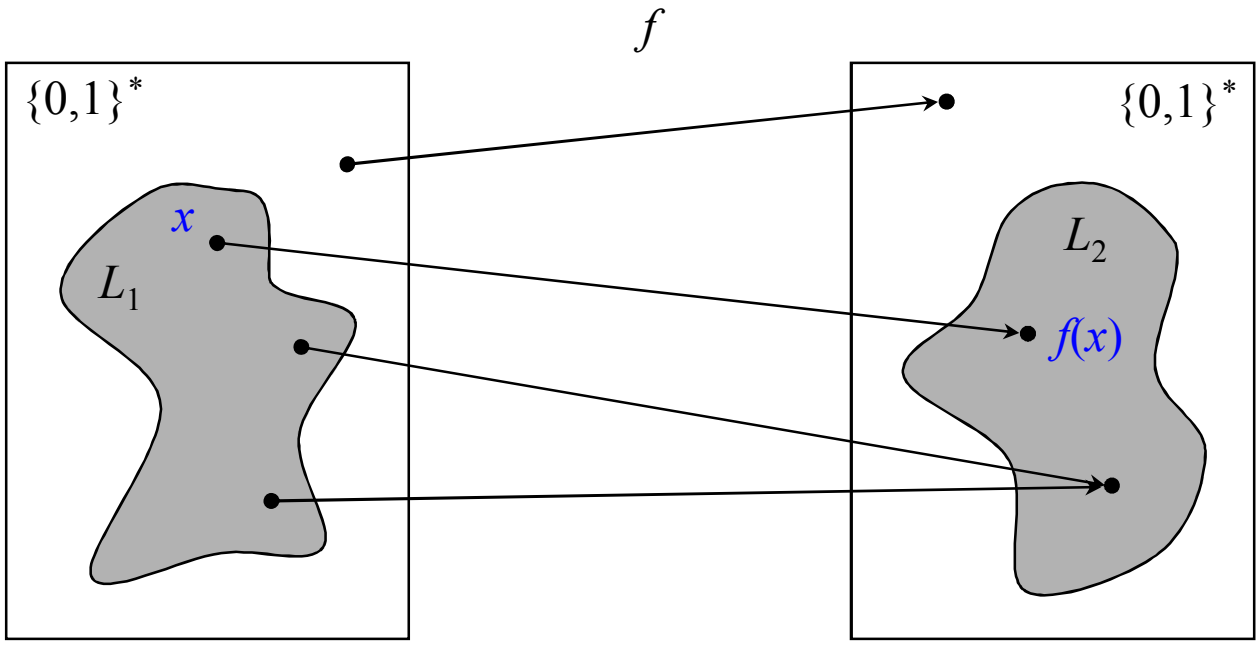
  Bài toán Q "có thể rút gọn được" về bài toán Q' bằng cách biểu diễn phương trình bậc nhất dưới dạng: $0x^2 + qx + r = 0$.
- Q là "không khó hơn" Q'.
- Điều kiện: thời gian để rút gọn bài toán "không được lâu hơn" thời gian để giải chính bài toán đó.

Tính có thể rút gọn được (tiếp)

- Một ngôn ngữ L_1 là *có thể rút gọn được trong thời gian đa thức về* một ngôn ngữ L_2, ký hiệu $L_1 \leq_P L_2$, nếu tồn tại một hàm có thể tính được trong thời gian đa thức $f : \{0, 1\}^* \to \{0, 1\}^*$ sao cho với mọi $x \in \{0, 1\}^*$,
  - $x \in L_1 \Leftrightarrow f(x) \in L_2$.
  - Ta gọi hàm f là *hàm rút gọn* (reduction function).

Tính có thể rút gọn được (tiếp)

- Nhận xét: $\forall x \in \{0,1\}^*$, trả lời "$x \in L_1$?" bằng cách trả lời "$f(x) \in L_2$?"
  - khi $f(x) \in L_2$ thì $x \in L_1$
  - khi $f(x) \notin L_2$ thì $x \notin L_1$



Tính có thể rút gọn được (tiếp)

– Một giải thuật thời gian đa thức F tính f được gọi là một *giải thuật rút gọn* (reduction algorithm).

Rút gọn trong thời gian đa thức

- ***Lemma 36.3***
- $L_1, L_2 \subseteq \{0, 1\}^*$ là các ngôn ngữ sao cho $L_1 \leq_P L_2$
- $\Rightarrow$
- Nếu $L_2 \in P$ thì $L_1 \in P$.

NP-đầy đủ

- Một ngôn ngữ $L \subseteq \{0, 1\}^*$ là *NP-đầy đủ* (NP-complete) nếu
- 1. $L \in$ NP
- 2. $L' \leq_P L$ với mọi $L' \in$ NP.
- Một ngôn ngữ $L \subseteq \{0, 1\}^*$ là *NP-khó* (NP-hard) nếu tính chất 2 ở trên được thỏa (trong khi tính chất 1 không nhất thiết phải được thỏa.)

  "Mọi bài toán trong NP đều không khó hơn bài toán L"
- Ta định nghĩa NPC là lớp các ngôn ngữ NP-đầy đủ.

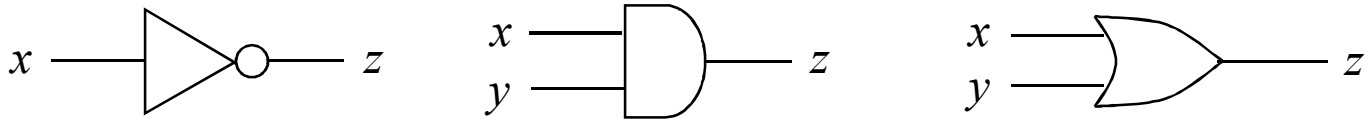
  "NPC là lớp các bài toán khó nhất trong NP"

NP-đầy đủ (tiếp)

- ***Định lý 36.4***
  - Nếu có bất kỳ một bài toán NP-đầy đủ nào có thể giải được trong thời gian đa thức, thì P = NP.
- Tương đương như thế:
  - Nếu có bất kỳ một bài toán nào trong NP là không thể giải được trong thời gian đa thức, thì không có bài toán NP-đầy đủ nào là giải được trong thời gian đa thức.

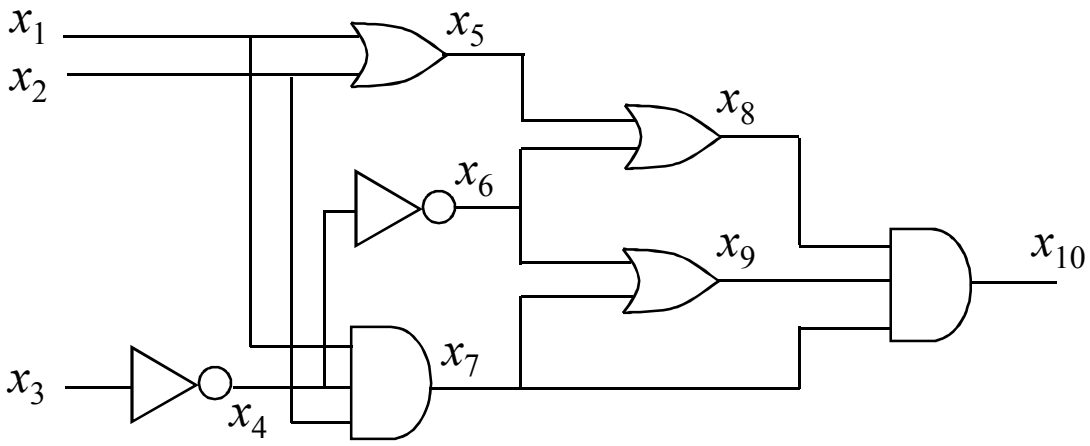
Bài toán thỏa mãn mạch

- Cổng logic



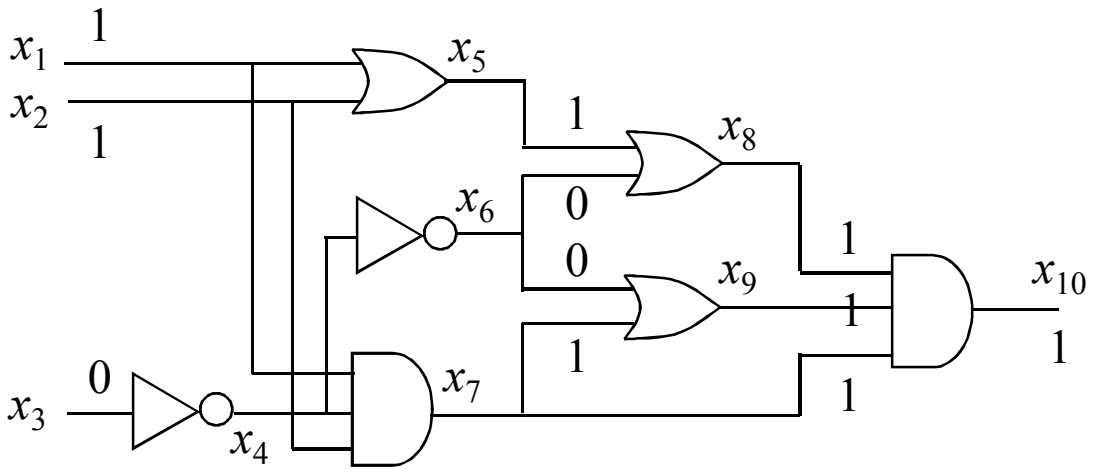
Cổng NOT Cổng AND Cổng OR

- Mạch tổ hợp bool



Bài toán thỏa mãn mạch (tiếp)

- Tính chất thỏa mãn mạch
  - Một *cách gán trị bool* (truth assignment) cho một mạch tổ hợp bool là một tập các trị input bool
  - Một mạch tổ hợp bool với chỉ một output là *có thể thoả mãn được* (satisfiable) nếu nó có một *cách gán thỏa mãn* (satisfying assignment), tức là một cách gán trị bool khiến cho output của mạch là 1.



Bài toán thỏa mãn mạch (tiếp)

- *Bài toán thỏa mãn mạch* là "Cho một mạch tổ hợp bool tạo bởi các cổng AND, OR, và NOT, nó có thể thỏa mãn được không?"
- CIRCUIT-SAT = { $\langle C \rangle$: C là một mạch tổ hợp bool có thể thỏa mãn được}.
- ***Lemma 36.5***
- Bài toán thỏa mãn mạch thuộc về lớp NP.
- ***Lemma 36.6***
- Bài toán thỏa mãn mạch là NP-khó.
- ***Theorem 36.7***
- Bài toán thỏa mãn mạch là NP-đầy đủ.

Cách chứng minh NP-đầy đủ

- ***Lemma 36.8***
- Nếu L là một ngôn ngữ sao cho $L' \leq_P L$ với một $L' \in$ NPC, thì L là NP-khó. Thêm vào đó, nếu $L \in$ NP, thì $L \in$ NPC.

Bài toán thỏa mãn biểu thức bool

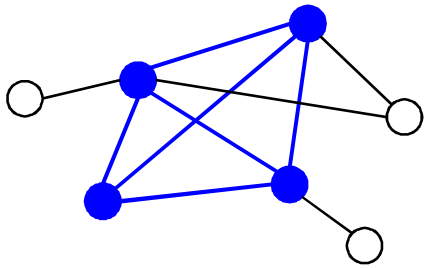
- *Biểu thức bool*
- $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
  - Một *cách gán trị bool* (truth assignment) cho một biểu thức bool ϕ là một tập các trị cho các biến của ϕ.
  - Một *cách gán thoả mãn* (satisfying assignment) là một cách gán trị bool khiến cho biểu thức bool có trị là 1.
  - Một biểu thức bool có một cách gán thỏa mãn gọi là một biểu thức *có thể thỏa mãn được*.
- *Bài toán thỏa mãn biểu thức bool*
- SAT = $\{\langle \phi \rangle : \phi$ là biểu thức bool có thể thỏa mãn được$\}$.
- ***Theorem 36.9***
- Bài toán thỏa mãn biểu thức bool là NP-đầy đủ.

Bài toán thỏa mãn biểu thức bool dạng 3-CNF

- *Biểu thức bool dạng 3-CNF* (3-conjunctive normal form)
- $\phi = (x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$
- *Bài toán thỏa mãn biểu thức bool dạng 3-CNF*
- 3-CNF-SAT = $\{\langle \phi \rangle : \phi$ là biểu thức bool dạng 3-CNF có thể thỏa mãn được$\}$.
- ***Theorem 36.9***
- Bài toán thỏa mãn biểu thức bool dạng 3-CNF là NP-đầy đủ.

Bài toán clique

- Các định nghĩa
  - Một *clique* của một đồ thị vô hướng $G = (V, E)$ là một đồ thị con đầy đủ của G.



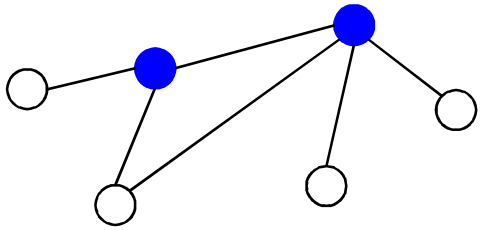
  - *Kích thước* của một clique là số đỉnh mà nó chứa.

Bài toán clique (tiếp)

- *Bài toán clique* là bài toán tối ưu tìm clique có kích thước lớn nhất của một đồ thị.
- Bài toán quyết định tương ứng với bài toán clique là
- CLIQUE = $\{ \langle G, k \rangle : G$ là một đồ thị có một clique có kích thước $k \}$.
- ***Theorem 36.11***
- Bài toán clique là NP-đầy đủ.

Bài toán che phủ đỉnh

- Các khái niệm cơ bản
  - Một *che phủ đỉnh* (vertex cover) của một đồ thị vô hướng $G = (V, E)$ là một tập con $V' \subseteq V$ sao cho nếu $(u, v) \in E$ thì $u \in V'$ hoặc $v \in V'$ (hoặc cả hai).



  - *Kích thước* của một che phủ đỉnh là số đỉnh trong đó.

Bài toán che phủ đỉnh (tiếp)

- *Bài toán che phủ đỉnh* là tìm một che phủ đỉnh có kích thước nhỏ nhất trong một đồ thị cho trước.
- Bài toán quyết định tương ứng dưới dạng một ngôn ngữ là:
- VERTEX-COVER = { $\langle G, k \rangle$: đồ thị G có một che phủ đỉnh có kích thước k}.
- ***Theorem 36.12***
- Bài toán che phủ đỉnh là NP-đầy đủ.

Bài toán tổng của tập con

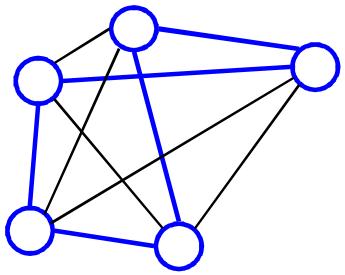
- Cho một tập hữu hạn $S \subset \mathbf{N}$ và một *trị đích* $t \in \mathbf{N}$.
- *Bài toán tổng của tập con* là hỏi có tồn tại một tập con $S' \subseteq S$ sao cho tổng các phần tử của nó bằng t hay không.
  - Ví dụ: với $S = \{1, 3, 5, 7, 11, 13\}$, và $t = 12$ thì tập con $S' = \{1, 11\}$ là một lời giải.
- Bài toán tổng của tập con dưới dạng một ngôn ngữ:
- SUBSET-SUM = $\{ \langle S, t \rangle$: tồn tại một tập con $S' \subseteq S$ sao cho
  - $t = \sum_{s \in S'} s \}$.
- ***Theorem 36.13***
- Bài toán tổng của tập con là NP-đầy đủ.

Bài toán chu trình Hamilton

- Bài toán chu trình Hamilton
- HAM-CYCLE = $\{\langle G\rangle : G$ là một đồ thị hamilton$\}$
- ***Theorem 36.14***
- Bài toán chu trình hamilton là NP-đầy đủ.

Bài toán người bán hàng rong

- Các khái niệm cơ bản
  - Cho một đồ thị đầy đủ G. Mỗi cạnh (i, j) nối hai đỉnh i và j của G có một chi phí là một số nguyên $c(i, j)$
  - Ta định nghĩa một *tua* (tour) là một chu trình hamilton của G, chi phí của tua là tổng của các chi phí của mỗi cạnh của tua.



Bài toán người bán hàng rong (tiếp)

- *Bài toán người bán hàng rong* (TSP, travelling-salesperson problem) là tìm một tua có chi phí nhỏ nhất.
- Ngôn ngữ hình thức cho bài toán quyết định tương ứng là
- TSP = { $\langle G, c, k \rangle$: $G = (V, E)$ là một đồ thị đầy đủ,
  - c là một hàm số $V \times V \to \mathbf{Z}$,
  - $k \in \mathbf{Z}$, và
  - G có một tua với chi phí $\leq k$}.
- ***Theorem 36.15*** Bài toán người bán hàng rong là NP-đầy đủ.

P = NP?

Bài toán mở quan trọng nhất trong khoa học máy tính lý thuyết.

Giải Thuật Xấp Xỉ

Chapter 37
Approximation Algorithms

Tiếp cận một bài toán NP-đầy đủ

- Nếu một bài toán là NP-đầy đủ thì không chắc rằng ta sẽ tìm được một giải thuật thời gian đa thức để giải nó một cách chính xác.
- Tiếp cận một bài toán NP-đầy đủ

  1) Nếu các input có kích thước nhỏ thì một giải thuật chạy trong thời gian số mũ vẫn có thể thoả mãn yêu cầu

  2) Thay vì tìm các lời giải tối ưu, có thể tìm các lời giải gần tối ưu trong thời gian đa thức.

Giải thuật xấp xỉ

- Một *giải thuật xấp xỉ* là một giải thuật trả về lời giải gần tối ưu.
- Giả sử: chi phí của lời giải > 0. Gọi C^* là chi phí của lời giải tối ưu.

  Một giải thuật xấp xỉ cho một bài toán tối ưu được gọi là có *tỉ số xấp xỉ* $\rho(n)$ (approximation ratio, ratio bound) nếu với mọi input có kích thước n thì chi phí của lời giải do giải thuật xấp xỉ tìm được sẽ thoả
  - $\max(C/C^*, C^*/C) \le \rho(n)$.

Giải thuật xấp xỉ

- Chi phí của lời giải do giải thuật xấp xỉ tìm được thỏa, với tỉ số xấp xỉ $\rho(n)$,
  - $\max(C/C^*, C^*/C) \le \rho(n)$
  - Bài toán tối đa: $0 < C \le C^*$, vậy

    $\max(C/C^*, C^*/C) = C^*/C \le \rho(n)$.

    Chi phí của lời giải tối ưu $\le \rho(n)$ lần chi phí của lời giải gần đúng.
  - Bài toán tối thiểu: $0 < C^* \le C$, vậy

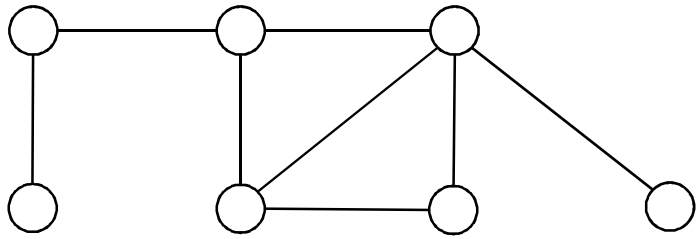
    $\max(C/C^*, C^*/C) = C/C^* \le \rho(n)$.

    Chi phí của lời giải gần đúng $\le \rho(n)$ lần chi phí của lời giải tối ưu.
- Một giải thuật xấp xỉ có tỉ số xấp xỉ $\rho(n)$ được gọi là một giải thuật $\rho(n)$-*xấp xỉ*.

Bài toán che phủ đỉnh

Nhắc lại

- Một *che phủ đỉnh* (vertex cover) của một đồ thị vô hướng $G = (V, E)$ là một tập con $V' \subseteq V$ sao cho nếu $(u, v) \in E$ thì $u \in V'$ hay $v \in V'$ (hoặc cả hai $\in V'$).



Kích thước của một che phủ đỉnh là số phần tử của nó.

- *Bài toán che phủ đỉnh* là tìm một che phủ đỉnh có kích thước nhỏ nhất trong một đồ thị vô hướng đã cho.

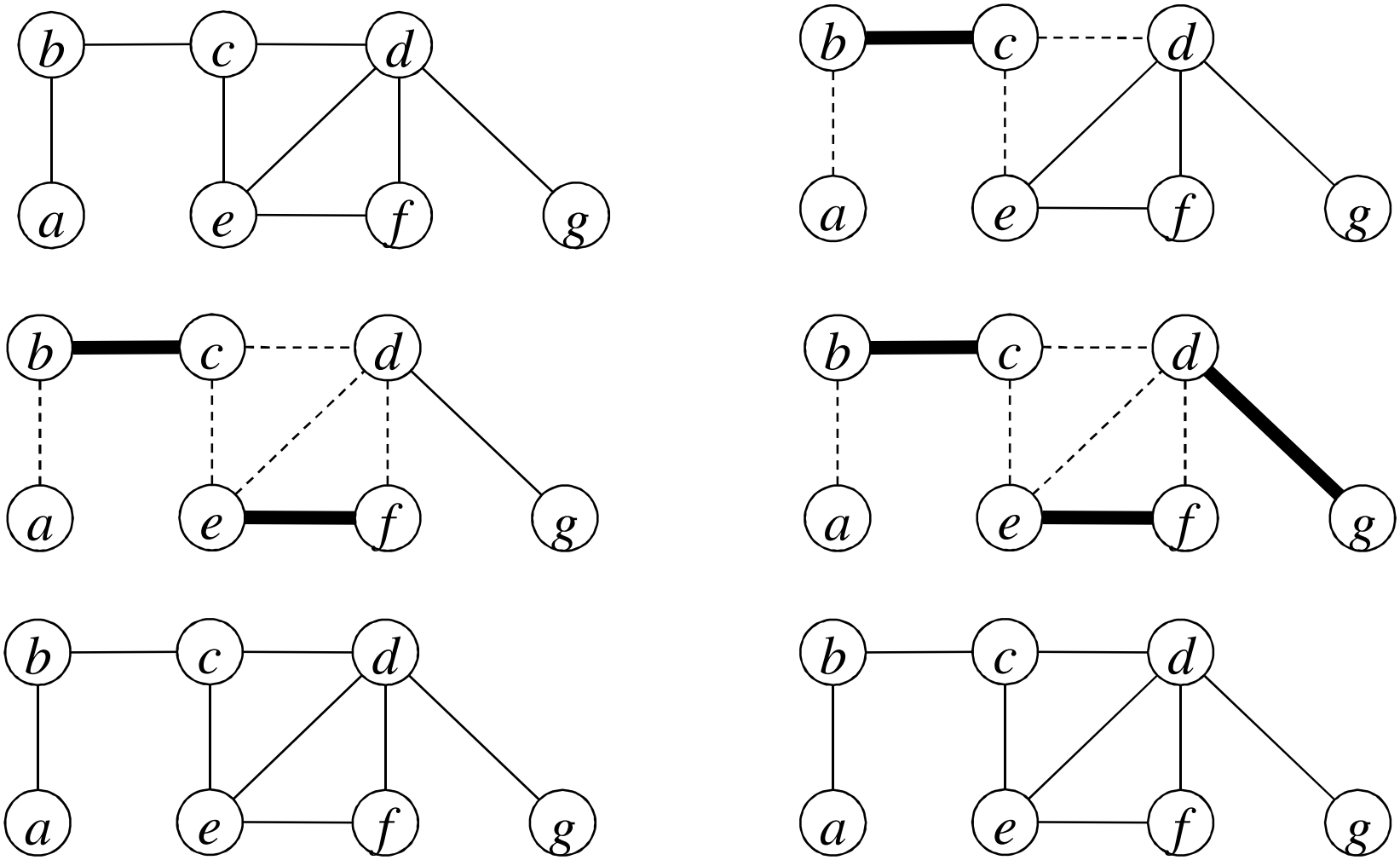
Bài toán này là dạng bài toán tối ưu của ngôn ngữ NP-đầy đủ

VERTEX-COVER = $\{\langle G, k\rangle$: đồ thị G có một che phủ đỉnh có kích thước $k\}$.

Một giải thuật xấp xỉ cho bài toán che phủ đỉnh

```
APPROX-VERTEX-COVER(G)
1   C ← ∅
2   E' ← E[G]
3   while E' ≠ ∅
4       do xét (u, v) là một cạnh bất kỳ của E'
5          C ← C ∪ {u, v}
6          tách khỏi E' tất cả các cạnh liên thuộc tại u hay v
7   return C
```

Thực thi APPROX-VERTEX-COVER



Phân tích APPROX-VERTEX-COVER

Nhận xét: Thời gian chạy của APPROX-VERTEX-COVER là $O(E)$.

Định lý 37.1

APPROX-VERTEX-COVER là một giải thuật 2-xấp xỉ trong thời gian đa thức.

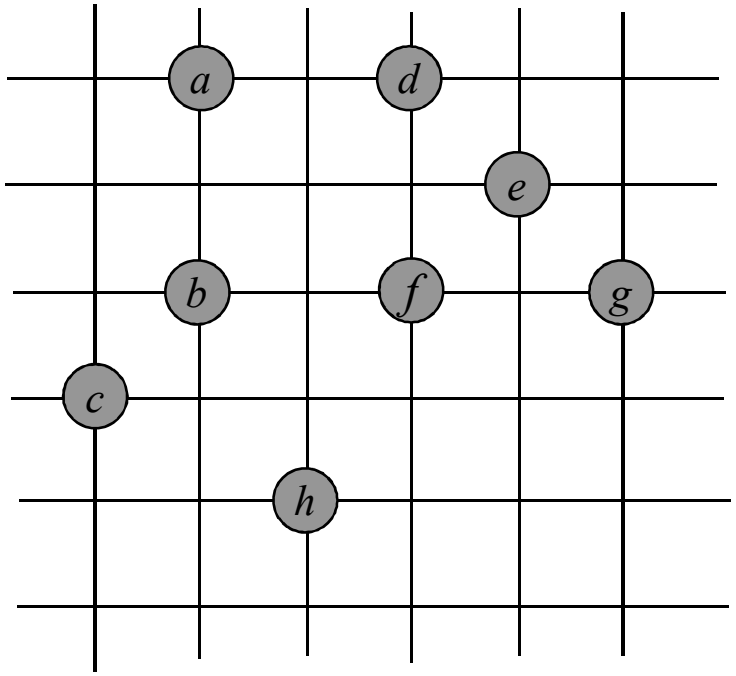
Bài toán người bán hàng rong với bất đẳng thức tam giác

- Cho một đồ thị đầy đủ vô hướng $G = (V, E)$ cùng với một hàm chi phí $c : E \to \mathbf{Z}^+$. Tìm một chu trình hamilton (một tour) của G với phí tổn nhỏ nhất.
- Điều kiện: Hàm chi phí c: $E \to \mathbf{Z}^+$ thỏa mãn *bất đẳng thức tam giác*

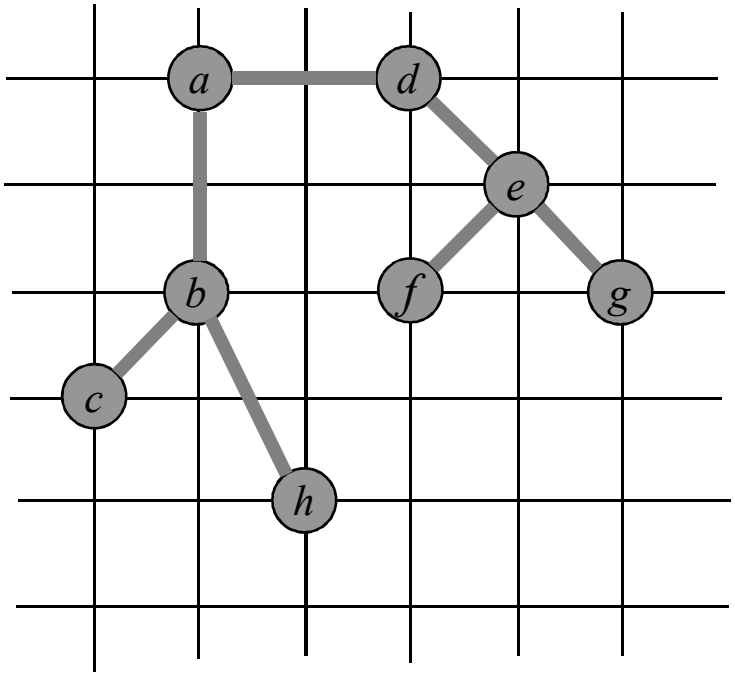
$$c(u, w) \le c(u, v) + c(v, w), \forall u, v, w \in V .$$

```
APPROX-TSP-TOUR(G, c)
1  chọn một đỉnh r ∈ V[G] làm một đỉnh "gốc"
2  nuôi lớn một cây khung nhỏ nhất T cho G từ
   gốc r dùng giải thuật MST-PRIM(G, c, r)
3  gọi L là danh sách các đỉnh được thăm viếng
   bởi phép duyệt cây theo kiểu tiền thứ tự
4  return chu trình hamilton H viếng các đỉnh
   theo thứ tự L
```

Thực thi APPROX-TSP-TOUR lên một ví dụ



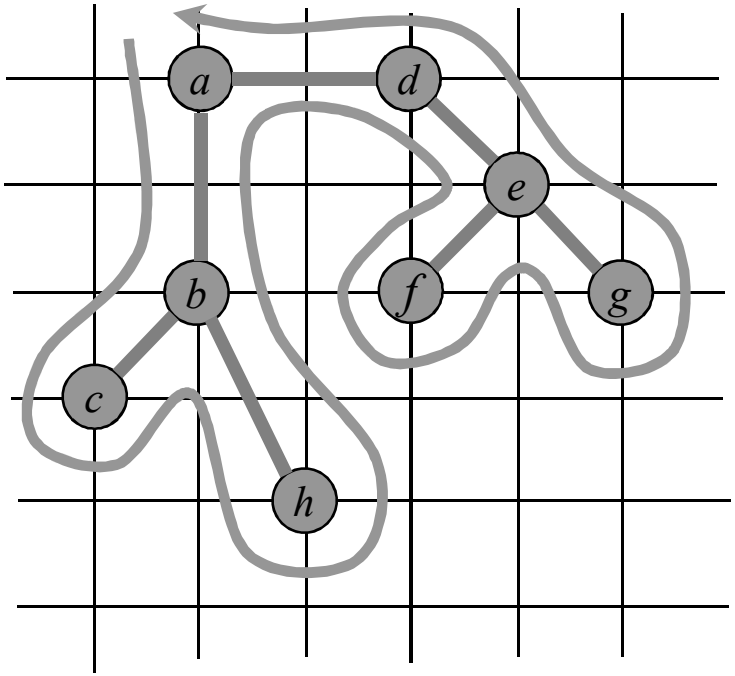
(a)



(b)

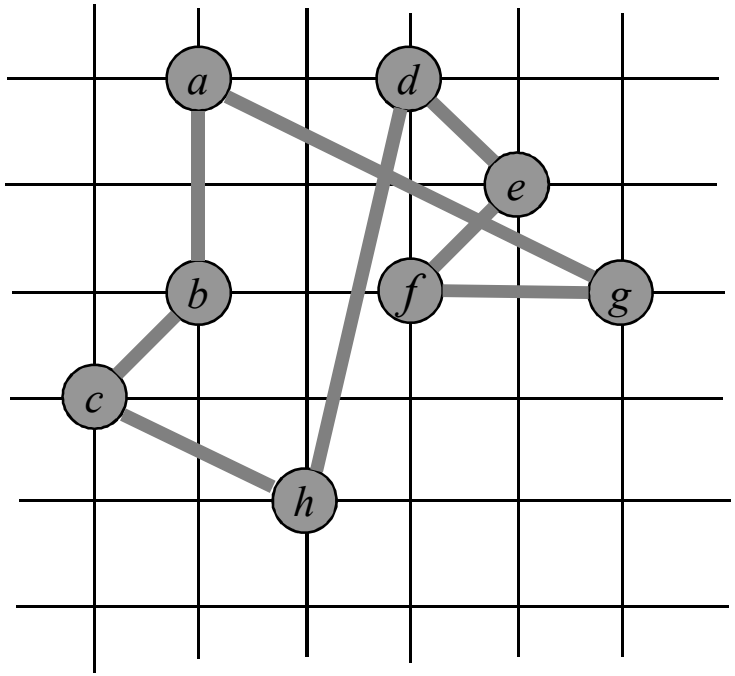
Cây khung nhỏ nhất T tính bởi MST-PRIM, đỉnh a là đỉnh gốc.

Thực thi APPROX-TSP-TOUR lên một ví dụ (tiếp)



(c)

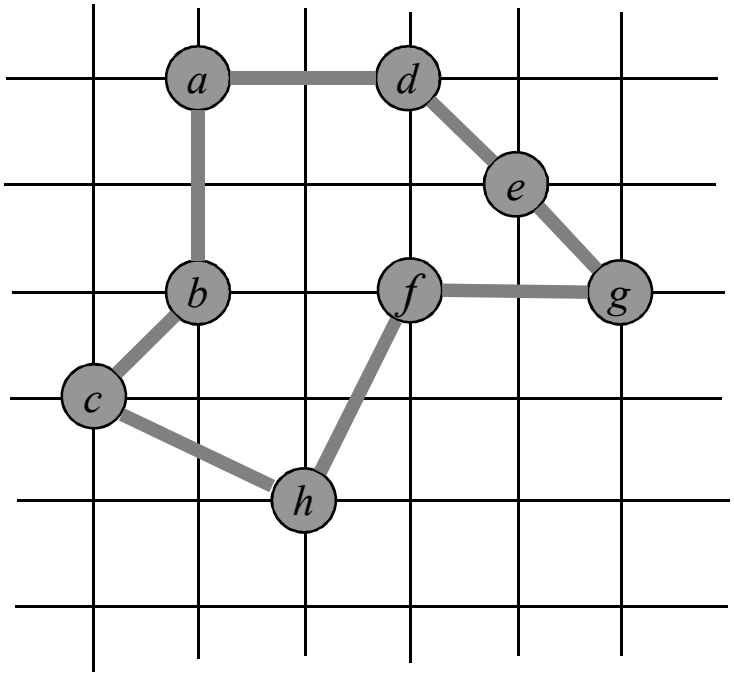
Duyệt cây T bắt đầu từ a. Thứ tự các đỉnh khi duyệt kiểu hoàn toàn là: $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. Thứ tự các đỉnh khi duyệt kiểu tiền thứ tự là: a, b, c, h, d, e, f, g.



(d)

Tua H có được từ kết quả duyệt cây theo kiểu tiền thứ tự mà APPROX-TSP-TOUR tìm được. Chi phí của tua H là khoảng chừng 19,074.

Thực thi APPROX-TSP-TOUR lên một ví dụ (tiếp)



(e)

Tua tối ưu H^*, có chi phí là 14,715.

Bài toán người bán hàng rong với bất đẳng thức tam giác

- ***Định lý 37.2***
- APPROX-TSP-TOUR là một giải thuật 2-xấp xỉ thời gian đa thức cho bài toán người bán hàng rong với bất đẳng thức tam giác.
- ***Chứng minh***

  Cho $A \subseteq E$, định nghĩa $c(A) = \sum_{(u,v)\in A} c(u,v)$

- Gọi H^* là một tua tối ưu, gọi H là tua mà APPROX-TSP-TOUR tìm được
- Cần chứng minh: $c(H) \le 2c(H^*)$.
- (*) Ta có $c(T) \le c(H^* - e) \le c(H^*)$ vì nếu xoá đi bất cứ cạnh e nào của H^* thì được một cây khung, mà T lại là cây khung nhỏ nhất.

Bài toán người bán hàng rong với bất đẳng thức tam giác

Chứng minh (tiếp)

- $c(W) = 2c(T)$,với W là kết quả một duyệt hoàn toàn cây T từ đỉnh r, vì mỗi cạnh của T được đi qua hai lần.
- $c(W) \le 2c(H^*)$,từ trên và vì (*).
- Nhưng W không phải là tua vì mỗi đỉnh được thăm hai lần, do đó "tránh thăm mọi đỉnh lần thứ hai" (= duyệt cây theo kiểu tiền thứ tự) để có được tua H, chi phí không tăng vì bất đẳng thức tam giác, do đó
- $c(H) \le c(W) \le 2c(H^*)$.

Bài toán người bán hàng rong tổng quát

- ***Định lý 37.3***
- Nếu $P \neq NP$ và $\rho \geq 1$, thì không tồn tại giải thuật xấp xỉ thời gian đa thức với tỉ số xấp xỉ ρ cho bài toán người bán hàng rong tổng quát.

- ***Chứng minh***

  Chứng minh bằng phản chứng.
- Giả sử có một số nguyên $\rho \geq 1$ và một giải thuật ρ-xấp xỉ thời gian đa thức A cho bài toán người bán hàng rong tổng quát.

  Hướng chứng minh: Sẽ dùng A để giải bài toán chu trình Hamilton HAM-CYCLE trong thời gian đa thức. Vì HAM-CYCLE là NP-đầy đủ và theo giả thiết $P \neq NP$ nên A không chạy trong thời gian đa thức, mâu thuẩn!

Bài toán người bán hàng rong tổng quát

- ***Chứng minh*** (tiếp)
- Gọi $G = (V, E)$ là một thực thể (instance) của bài toán chu trình hamilton.

  Từ G định nghĩa đồ thị $G' = (V, E')$ là đồ thị đầy đủ trên V, với hàm chi phí
- $c(u, v) = 1$ nếu $(u, v) \in E$
- $= \rho|V| + 1$ trong các trường hợp khác.
- Các biểu diễn của G' và c có thể tính được từ một biểu diễn của G trong thời gian đa thức theo $|V|$ và $|E|$.

Bài toán người bán hàng rong tổng quát

Chứng minh (tiếp)

- Gọi H^* là tua tối ưu của G', gọi H là tua mà A tìm được, ta có
- $c(H) \le \rho \cdot c(H^*)$. Phân biệt hai trường hợp:
  - Trường hợp $c(H) > \rho|V|$

    $\rho|V| < c(H) \le \rho \cdot c(H^*) \Rightarrow |V| < c(H^*)$

    Vậy H^* phải chứa ít nhất một cạnh $\notin E$. Suy ra G không có chu trình hamilton.
  - Trường hợp $c(H) \le \rho|V|$
    - $c(H) < \rho|V| + 1$ = chi phí của một cạnh bất kỳ $\notin E$. Do đó H chỉ chứa cạnh của G, từ đó suy ra H là một chu trình hamilton của G.
- Vậy ta có thể dùng giải thuật A để giải bài toán chu trình hamilton trong thời gian đa thức. Mâu thuẫn với giả thiết P ≠ NP!

Bài toán che phủ tập

- Một thực thể $(X, \mathcal{F})$ của *bài toán che phủ tập* gồm một tập hữu hạn X và một họ $\mathcal{F}$ các tập con của X sao cho

$$X = \bigcup_{S \in \mathcal{F}} S .$$

  Một tập con $\mathcal{C} \subseteq \mathcal{F}$ được gọi là *che phủ* X nếu $X = \bigcup_{S \in \mathcal{C}} S$.

- Bài toán che phủ tập là tìm một tập con $\mathcal{C} \subseteq \mathcal{F}$, với $|\mathcal{C}|$ là nhỏ nhất, sao cho $\mathcal{C}$ che phủ X.

Dạng quyết định của bài toán che phủ tập

- Dạng bài toán quyết định cho bài toán che phủ tập là tìm một che phủ sao cho kích thước của nó $\leq k$, với k là một tham số của một thực thể của bài toán quyết định.
- Bài toán quyết định cho bài toán che phủ tập là NP-đầy đủ.

Một giải thuật xấp xỉ cho bài toán che phủ tập

- Một giải thuật xấp xỉ cho bài toán che phủ tập
  - dùng phương pháp greedy.

```
GREEDY-SET-COVER(X, F)
1   U ← X
2   C ← ∅
3   while U ≠ ∅
4       do chọn một S ∈ F sao cho | S ∩ U | là lớn nhất
5          U ← U − S
6          C ← C ∪ {S}
7   return C
```

Phân tích GREEDY-SET-COVER

Gọi *số điều hòa* thứ d là H_d :

$$H_d = \sum_{i=1}^{d} \frac{1}{i}$$

Tính chất: $H_d \le \ln d + 1$.

Định lý 37.4

GREEDY-SET-COVER là một giải thuật $\rho(n)$-xấp xỉ thời gian đa thức, với $\rho(n) = H(\max\{|S| : S \in \mathcal{F}\})$.

Nhận xét: $\max\{|S| : S \in \mathcal{F}\} \le |X|$

Hệ luận 37.5

GREEDY-SET-COVER là một giải thuật $(\ln|X| + 1)$-xấp xỉ thời gian đa thức.