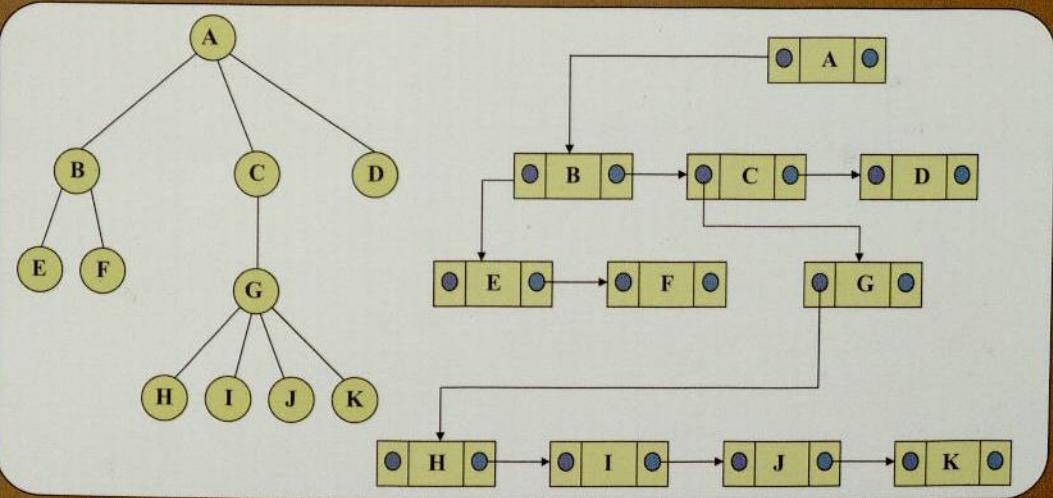


NGUYỄN ĐỨC NGHĨA

CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN



NHÀ XUẤT BẢN BÁCH KHOA - HÀ NỘI

NGUYỄN ĐỨC NGHĨA

**CẤU TRÚC DỮ LIỆU
VÀ
THUẬT TOÁN**

NHÀ XUẤT BẢN BÁCH KHOA – HÀ NỘI

LỜI NÓI ĐẦU

Cuốn sách **Cấu trúc dữ liệu và thuật toán** được biên soạn dựa trên nội dung các bài giảng mà tác giả sử dụng để giảng dạy cho sinh viên ngành Công nghệ Thông tin, Đại học Bách Khoa Hà Nội.

Với thời lượng để giảng dạy trong 60 tiết, cuốn sách chỉ đề cập được một số vấn đề cơ bản trong lĩnh vực “Cấu trúc dữ liệu và Thuật toán” – một môn học có ý nghĩa quan trọng trong hành trang kiến thức của sinh viên ngành Công nghệ Thông tin.

Nội dung cuốn sách bao gồm bảy chương:

Chương 1. Các khái niệm cơ bản.

Chương 2. Thuật toán đệ quy.

Chương 3. Các cấu trúc dữ liệu cơ bản.

Chương 4. Cây.

Chương 5. Các thuật toán sắp xếp.

Chương 6. Tìm kiếm.

Chương 7. Đồ thị và các thuật toán đồ thị.

Cuốn sách được biên soạn lần đầu nên chắc chắn sẽ còn rất nhiều thiếu sót. Tác giả rất mong nhận được ý kiến đóng góp của độc giả để có thể sửa chữa và bổ sung.

Mọi góp ý xin gửi về địa chỉ email: nghiand@soict.hut.edu.vn.

Xin chân thành cảm ơn!

Tác giả

MỤC LỤC

LỜI NÓI ĐẦU	3
CHƯƠNG 1. CÁC KHÁI NIỆM CƠ BẢN	9
1.1. Ví dụ mở đầu	9
1.1.1. Thuật toán trực tiếp.....	9
1.1.2. Thuật toán nhanh hơn	10
1.1.3. Thuật toán đệ quy	11
1.1.4. Thuật toán Quy hoạch động.....	13
1.2. Thuật toán và độ phức tạp.....	15
1.2.1. Khái niệm bài toán và thuật toán	15
1.2.2. Độ phức tạp của thuật toán	16
1.2.3. Các loại thời gian tính.....	16
1.3. Ký hiệu tiệm cận	16
1.3.1. Ký hiệu Θ	16
1.3.2. Ký hiệu O (đọc là ô lớn – big O)	17
1.3.3. Ký hiệu Ω	18
1.3.4. Sử dụng ký hiệu tiệm cận Θ , Ω , O	18
1.3.5. Một số lớp thuật toán	21
1.4. Giả ngôn ngữ.....	22
1.4.1. Khai báo biến.....	22
1.4.2. Câu lệnh gán	22
1.4.3. Các cấu trúc điều khiển.....	23
1.4.4. Câu lệnh lặp	23
1.4.5. Câu lệnh Vào–Ra	23
1.4.6. Hàm và thủ tục (Function and procedure)	24
1.5. Một số kỹ thuật phân tích thuật toán	25
1.5.1. Cấu trúc tuần tự	25
1.5.2. Phân tích vòng lặp for	26
1.5.3. Phân tích vòng lặp While và Repeat.....	26
1.5.4. Câu lệnh đặc trưng.....	28
Bài tập chương 1	30

CHƯƠNG 2. THUẬT TOÁN ĐỆ QUY	35
2.1. Khái niệm đệ quy.....	35
2.1.1. Hàm đệ quy (Recursive Functions)	35
2.1.2. Tập hợp được xác định đệ quy.....	36
2.2. Thuật toán đệ quy.....	38
2.3. Một số ví dụ minh họa.....	42
2.4. Phân tích thuật toán đệ quy.....	47
2.5. Đệ quy có nhớ	50
2.6. Chứng minh tính đúng đắn của thuật toán đệ quy.....	51
2.7. Thuật toán quay lui	55
Bài tập chương 2.....	60
CHƯƠNG 3. CÁC CẤU TRÚC DỮ LIỆU CƠ BẢN	65
3.1. Các khái niệm	65
3.1.1. Kiểu dữ liệu	65
3.1.2. Kiểu dữ liệu trùu tượng	66
3.1.3. Cấu trúc dữ liệu	67
3.2. Mảng.....	69
3.2.1. Kiểu dữ liệu trùu tượng mảng.....	69
3.2.2. Phân bổ bộ nhớ cho mảng.....	70
3.2.3. Các thao tác với mảng	73
3.3. Danh sách	74
3.3.1. Danh sách tuyến tính	74
3.3.2. Các cách cài đặt danh sách tuyến tính	75
3.3.3. Các ví dụ ứng dụng	88
3.3.4. Phân tích sử dụng danh sách móc nối.....	93
3.3.5. Một số biến thể của danh sách móc nối	95
3.4. Ngăn xếp.....	99
3.4.1. Kiểu dữ liệu trùu tượng ngăn xếp	99
3.4.2. Ngăn xếp dùng mảng	99
3.4.3. Cài đặt ngăn xếp với danh sách móc nối	100
3.4.3. Một số ứng dụng của ngăn xếp	108
3.5. Hàng đợi.....	125

3.5.1. ADT hàng đợi	125
3.5.2. Cài đặt hàng đợi bằng mảng	126
3.5.3. Cài đặt hàng đợi bởi danh sách mốc nối	127
3.5.4. Một số ví dụ ứng dụng hàng đợi	133
Bài tập chương 3	135
CHƯƠNG 4. CÂY	143
4.1. Định nghĩa và các khái niệm.....	143
4.1.1. Định nghĩa cây	143
4.1.2. Các thuật ngữ chính	147
4.1.3. Cây có thứ tự (Ordered Tree).....	150
4.1.4. Cây có nhãn (Labeled Tree)	153
4.1.5. ADT Cây.....	155
4.2. Cây nhị phân	158
4.2.1. Định nghĩa và tính chất.....	158
4.2.2. Biểu diễn cây nhị phân	161
4.3. Các ví dụ ứng dụng.....	172
4.3.1. Cây nhị phân biểu thức (Binary Expression Trees)	172
4.3.2. Cây quyết định (Decision Trees)	173
4.3.3. Mã Huffman (Huffman Code)	176
Bài tập chương 4	181
CHƯƠNG 5. CÁC THUẬT TOÁN SẮP XÉP	190
5.1. Bài toán sắp xếp	190
5.1.1. Bài toán sắp xếp.....	190
5.1.2. Giới thiệu sơ lược về các thuật toán sắp xếp	192
5.2. Ba thuật toán sắp xếp cơ bản.....	193
5.2.1. Sắp xếp chèn (Insertion Sort).....	193
5.2.2. Sắp xếp lựa chọn (Selection Sort).....	196
5.2.3. Sắp xếp nổi bọt (Bubble Sort).....	197
5.3. Sắp xếp trộn (Merge Sort)	199
5.4. Sắp xếp nhanh (Quick Sort)	202
5.4.1. Sơ đồ tổng quát	202
5.4.2. Phép phân đoạn.....	204
5.4.3. Độ phức tạp của sắp xếp nhanh	207

5.5. Sắp xếp vun đồng (Heap Sort).....	210
5.5.1. Cấu trúc dữ liệu đồng (Heap)	210
5.5.2. Sắp xếp vun đồng.....	215
5.5.3. Hàng đợi có ưu tiên (Priority queue)	216
5.6. Cận dưới cho độ phức tạp tính toán của bài toán sắp xếp.....	220
5.7. Các phương pháp sắp xếp đặc biệt	222
5.7.1. Mở đầu.....	222
5.7.2. Sắp xếp đếm (Counting Sort).....	222
5.7.3. Sắp xếp theo cơ số (Radix Sort)	224
5.7.4. Sắp xếp phân cụm (Bucket Sort)	228
Bài tập chương 5.....	230
CHƯƠNG 6. TÌM KIẾM.....	234
6.1. Tìm kiếm tuần tự và tìm kiếm nhị phân.....	234
6.1.1. Tìm kiếm tuần tự (Linear Search or Sequential Search).....	234
6.1.2. Tìm kiếm nhị phân (Binary Search)	235
6.2. Cây nhị phân tìm kiếm (Binary Search Tree).....	238
6.2.1. Định nghĩa	238
6.2.2. Biểu diễn cây nhị phân tìm kiếm	240
6.2.3. Các phép toán	242
6.3. Cây nhị phân tìm kiếm cân bằng – cây AVL	254
6.3.1. Định nghĩa	254
6.3.2. Các thao tác với cây AVL.....	262
6.4. Tìm kiếm xâu mẫu (String searching).....	264
6.4.1. Phát biểu bài toán.....	264
6.4.2. Thuật toán trực tiếp – Naïve algorithm.....	265
6.4.3. Thuật toán Boyer-Moore.....	266
6.4.4. Thuật toán Rabin-Karp.....	268
6.4.5. Thuật toán Knuth-Morris-Pratt (KMP)	271
6.5. Bảng băm (Mapping and Hashing).....	273
6.5.1. Đặt vấn đề	273
6.5.2. Địa chỉ trực tiếp – Direct Addressing	274
6.5.3. Hàm băm (Hash Functions)	275
Bài tập chương 6.....	279

CHƯƠNG 7. ĐỒ THỊ VÀ CÁC THUẬT TOÁN ĐỒ THỊ	286
7.1. Đồ thị	286
7.1.1. Các loại đồ thị	286
7.1.2. Đường đi, chu trình và tính liên thông của đồ thị	292
7.2. Biểu diễn đồ thị	294
7.2.1. Biểu diễn đồ thị bởi ma trận	295
7.2.2. Biểu diễn đồ thị bởi danh sách kè	297
7.3. Các thuật toán duyệt đồ thị	302
7.3.1. Thuật toán tìm kiếm theo chiều rộng (BFS)	302
7.3.2. Thuật toán tìm kiếm theo chiều sâu (DFS)	307
7.4. Một số ứng dụng của tìm kiếm trên đồ thị	314
7.4.1. Bài toán đường đi	314
7.4.2. Bài toán liên thông	315
7.4.3. Đồ thị không chứa chu trình và bài toán sắp xếp tôpô	316
7.4.4. Bài toán tô màu đỉnh đồ thị	324
7.4.5. Bài toán xây dựng bao đóng truyền ứng của đồ thị	326
7.5. Bài toán cây khung nhỏ nhất	329
7.5.1. Thuật toán Kruskal	329
7.5.2. Cấu trúc dữ liệu biểu diễn phân hoạch	334
7.6. Bài toán đường đi ngắn nhất	338
7.6.1. Thuật toán Dijkstra	340
7.6.2. Cài đặt thuật toán với các cấu trúc dữ liệu	341
Bài tập chương 7	347
TÀI LIỆU THAM KHẢO	361
PHỤ LỤC	362

Chương 1

CÁC KHÁI NIỆM CƠ BẢN

1.1. VÍ DỤ MỞ ĐẦU

Bài toán tìm dãy con lớn nhất. Cho dãy số:

$$a_1, a_2, \dots, a_n.$$

Dãy số a_i, a_{i+1}, \dots, a_j với $1 \leq i \leq j \leq n$ được gọi là **dãy con** của dãy đã cho và $\sum_{k=i}^j a_k$ được gọi là **trọng lượng** của dãy con này.

Vấn đề đặt ra là: Hãy tìm trọng lượng lớn nhất của dãy con, tức là tìm cực đại giá trị $\sum_{k=i}^j a_k$. Để đơn giản ta gọi dãy con có trọng lượng lớn nhất là **dãy con lớn nhất**.

Ví dụ: Nếu dãy đã cho là $-2, 11, -4, 13, -5, 2$ thì cần đưa ra câu trả lời là 20 (là trọng lượng của dãy con $11, -4, 13$).

1.1.1. Thuật toán trực tiếp

Thuật toán đơn giản đầu tiên có thể nghĩ để giải bài toán đặt ra là: duyệt tất cả các dãy con có thể:

$$a_i, a_{i+1}, \dots, a_j \text{ với } 1 \leq i \leq j \leq n$$

và tính tổng của mỗi dãy con để tìm ra trọng lượng lớn nhất.

Trước hết nhận thấy rằng, tổng số các dãy con có thể của dãy đã cho là:

$$C(n,2) + n = n^2/2 + n/2.$$

Thuật toán này có thể cài đặt trong đoạn chương trình sau:

```
int maxSum = 0;
for (int i=0; i<n; i++) {
    for (int j=i; j<n; j++) {
        int sum = 0;
        for (int k=i; k<=j; k++)
            sum += a[k];
        if sum > maxSum
```

```

    maxSum = sum;
}
}

```

Phân tích thuật toán: Ta sẽ tính số lượng phép cộng mà thuật toán phải thực hiện, tức là đếm xem dòng lệnh:

Sum += a[k]

phải thực hiện bao nhiêu lần. Số lượng phép cộng sẽ là:

$$\begin{aligned}
 \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j-i+1) &= \sum_{i=0}^{n-1} (1+2+\dots+(n-i)) = \sum_{i=0}^{n-1} \frac{(n-i)(n-i+1)}{2} \\
 &= \frac{1}{2} \sum_{k=1}^n k(k+1) = \frac{1}{2} \left[\sum_{k=1}^n k^2 + \sum_{k=1}^n k \right] = \frac{1}{2} \left[\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} \right] \\
 &= \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}
 \end{aligned}$$

1.1.2. Thuật toán nhanh hơn

Để ý rằng tổng các số hạng từ i đến j có thể thu được từ tổng của các số hạng từ i đến $j-1$ bởi một phép cộng, cụ thể ta có:

$$\sum_{k=i}^j a[k] = a[j] + \sum_{k=i}^{j-1} a[k].$$

Nhận xét này cho phép rút bỏ vòng lặp for trong cùng.

Ta có thể cài đặt như sau:

```

int maxSum = a[0];
for (int i=0; i<n; i++) {
    int sum = 0;
    for (int j=i; j<n; j++) {
        sum += a[j];
        if sum > maxSum
            maxSum = sum;
    }
}

```

Phân tích thuật toán: Ta lại tính số lần thực hiện phép cộng và thu được kết quả sau:

$$\sum_{i=0}^{n-1} (n-i) = n + (n-1) + \dots + 1 = \frac{n^2}{2} + \frac{n}{2}.$$

Để ý rằng số này đúng bằng số lượng dãy con. Đường như thuật toán thu được là rất tốt, vì ta phải xét mỗi dãy con đúng một lần.

1.1.3. Thuật toán đệ quy

Ta còn có thể xây dựng thuật toán tốt hơn nữa bằng cách sử dụng kỹ thuật chia để trị. Kỹ thuật này bao gồm các bước sau:

- Chia bài toán cần giải ra thành các bài toán con cùng dạng;
- Giải mỗi bài toán con một cách đệ quy;
- Tổ hợp lời giải của các bài toán con để thu được lời giải của bài toán xuất phát.

Áp dụng kỹ thuật này đối với bài toán tìm trọng lượng lớn nhất của các dãy con. Ta chia dãy đã cho ra thành hai dãy sử dụng phần tử ở chính giữa và thu được hai dãy số (gọi tắt là dãy bên trái và dãy bên phải) với độ dài giảm đi một nửa.

Để tổ hợp lời giải, nhận thấy rằng chỉ có thể xảy ra một trong ba trường hợp:

- Dãy con lớn nhất nằm ở dãy con bên trái (nửa trái);
- Dãy con lớn nhất nằm ở dãy con bên phải (nửa phải);
- Dãy con lớn nhất bắt đầu ở nửa trái và kết thúc ở nửa phải (giữa).

Do đó, nếu ký hiệu trọng lượng của dãy con lớn nhất ở nửa trái là w_L , ở nửa phải là w_R và ở giữa là w_M thì trọng lượng cần tìm sẽ là:

$$\text{Max}(w_L, w_R, w_M).$$

Việc tìm trọng lượng của dãy con lớn nhất ở nửa trái (w_L) và nửa phải (w_R) có thể thực hiện một cách đệ quy.

Để tìm trọng lượng w_M của dãy con lớn nhất bắt đầu ở nửa trái và kết thúc ở nửa phải, ta thực hiện như sau:

- Tính trọng lượng của dãy con lớn nhất trong nửa trái kết thúc ở điểm chia (w_{ML});
- Tính trọng lượng của dãy con lớn nhất trong nửa phải bắt đầu ở điểm chia (w_{MR});
- Khi đó $w_M = w_{ML} + w_{MR}$.

Để tính trọng lượng của dãy con lớn nhất ở nửa trái (từ $a[i]$ đến $a[j]$) kết thúc ở $a[j]$, ta dùng thuật toán sau:

```
MaxLeft(a, i, j);
{
    maxSum := -∞; sum := 0;
    for k := j downto i {
        sum := sum + a[k];
        maxSum := max(sum, maxSum);
    }
}
```

```

    }
    return maxSum;
}

```

Để tính trọng lượng của dãy con lớn nhất ở nửa phải (từ $a[i]$ đến $a[j]$) bắt đầu từ $a[i]$, ta dùng thuật toán sau:

```

MaxRight(a, i, j);
{
    maxSum := -∞; sum := 0;
    for k := i to j {
        sum := sum + a[k];
        maxSum := max(sum, maxSum);
    }
    return maxSum;
}

```

Sơ đồ của thuật toán đệ quy có thể mô tả như sau:

```

MaxSub(a, i, j);
{
    if (i = j) return a[i]
    else
    {
        m := (i+j)/2;
        wL := MaxSub(a, i, m);
        wR := MaxSub(a, m+1, j);
        wM := MaxLeft(a, i, m) +
              MaxRight(a, m+1, j);
        return max(wL, wR, wM);
    }
}

```

Phân tích thuật toán: Ta cần tính xem lệnh gọi $\text{MaxSub}(a, l, n)$ để thực hiện thuật toán đòi hỏi bao nhiêu phép cộng?

Trước hết, ta nhận thấy MaxLeft và MaxRight đòi hỏi:

$$n/2 + n/2 = n \text{ phép cộng.}$$

Vì vậy, nếu gọi $T(n)$ là số phép cộng cần tìm, ta có công thức đệ quy sau:

$$T(n) = \begin{cases} 0 & n = 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n = 2T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Ta khẳng định rằng $T(2^k) = k \cdot 2^k$ và chứng minh bằng quy nạp.

- Cơ sở quy nạp: Nếu $k = 0$ thì $T(2^0) = T(1) = 0 = 0.2^0$.
- Chuyển quy nạp: Nếu $k > 0$, giả sử rằng $T(2^{k-1}) = (k-1)2^{k-1}$ là đúng. Khi đó:
$$T(2^k) = 2T(2^{k-1}) + 2^k = 2(k-1).2^{k-1} + 2^k = k.2^k.$$
- Quay lại với ký hiệu n , ta có:

$$T(n) = n \log n$$

Kết quả cho thấy thuật toán thu được nhanh hơn thuật toán thứ hai.

1.1.4. Thuật toán Quy hoạch động

Ta còn có thể phát triển thuật toán nhanh hơn nữa nhờ sử dụng kỹ thuật quy hoạch động. Việc phát triển thuật toán dựa trên quy hoạch động bao gồm ba giai đoạn:

1. **Phân rã:** chia bài toán cần giải thành những bài toán con nhỏ hơn có cùng dạng với bài toán ban đầu.
2. **Ghi nhận lời giải:** lưu trữ lời giải của các bài toán con vào một bảng.
3. **Tổng hợp lời giải:** lần lượt từ lời giải của các bài toán con kích thước nhỏ hơn tìm cách xây dựng lời giải của bài toán kích thước lớn hơn, cho đến khi thu được lời giải của bài toán xuất phát (là bài toán con có kích thước lớn nhất).

Phân rã: Gọi s_i là trọng lượng của dãy con lớn nhất trong dãy a_1, a_2, \dots, a_i , $i = 1, 2, \dots, n$. Rõ ràng s_n là giá trị cần tìm.

Tổng hợp lời giải: Trước hết, ta có:

$$s_1 = a_1.$$

Giả sử $i > 1$ và s_k là đã biết với $k = 1, 2, \dots, i-1$. Ta cần tính s_i là trọng lượng dãy con lớn nhất của dãy:

$$a_1, a_2, \dots, a_{i-1}, a_i.$$

Do dãy con lớn nhất của dãy này hoặc là có chứa phần tử a_i hoặc không chứa phần tử a_i , nên nó chỉ có thể là một trong hai dãy:

- Dãy con lớn nhất của dãy a_1, a_2, \dots, a_{i-1} ;
- Dãy con lớn nhất của dãy a_1, a_2, \dots, a_{i-1} kết thúc tại a_i .

Từ đó suy ra:

$$s_i = \max \{s_{i-1}, e_i\}, i = 2, \dots, n.$$

Trong đó: e_i là trọng lượng dãy con lớn nhất của dãy a_1, a_2, \dots, a_{i-1} kết thúc tại a_i .

Để tính e_i , ta cũng có thể sử dụng công thức đệ quy sau:

$$e_1 = a_1;$$

$$e_i = \max \{a_i, e_{i-1} + a_i\}, i = 2, \dots, n.$$

Tổng hợp các kết quả ta đi đến thuật toán sau:

```
MaxSub(a) ;  
{  
    smax = a[1]; (* smax - trọng lượng của dãy con lớn  
nhất *)  
    maxendhere = a[1];  
    imax = 1;      (* imax - vị trí kết thúc của dãy con  
lớn nhất *)  
    for i = 2 to n {  
        u = maxendhere + a[i];  
        v = a[i];  
        if (u > v) maxendhere = u;  
        else maxendhere = v;  
        if (maxendhere > smax) then {  
            smax := maxendhere;  
            imax := i;  
        }  
    }  
}
```

Phân tích thuật toán: Để thấy số phép toán cộng phải thực hiện trong thuật toán (số lần thực hiện câu lệnh $u = \text{maxendhere} + a[i]$) là n .

So sánh các thuật toán

- Cùng một bài toán ta đã đề xuất bốn thuật toán đòi hỏi số lượng phép toán khác nhau và vì thế sẽ đòi hỏi thời gian tính khác nhau.
- Các bảng trinh bày dưới đây cho thấy thời gian tính với giả thiết máy tính có thể thực hiện 10^8 phép cộng trong 1 giây.

Số phép toán	$n = 10$	Thời gian	$n = 100$	Thời gian
$\log(n)$	3.32	3.3×10^{-8} giây	6.64	6×10^{-8} giây
$n \log(n)$	33.2	3.3×10^{-7} giây	664	6.6×10^{-6} giây
n^2	100	10^{-6} giây	10000	10^{-4} giây
n^3	1×10^3	10^{-5} giây	1×10^6	10^{-2} giây
e^n	2.2×10^4	2×10^{-4} giây	2.69×10^{10}	$> 10^{26}$ thế kỷ

Số phép toán	$n = 10000$	Thời gian	$n = 10^6$	Thời gian
$\log(n)$	13.3	10^{-6} giây	19.9	$< 10^{-5}$ giây
$n \log(n)$	1.33×10^3	10^{-3} giây	1.99×10^7	2×10^{-1} giây
n^2	1×10^8	1 giây	1×10^{12}	2.77 giờ
n^3	1×10^{12}	2.7 giờ	1×10^{15}	115 ngày
e^n	8.81×10^{342}	$> 10^{4327}$ thế kỷ		

1.2. THUẬT TOÁN VÀ ĐỘ PHỨC TẠP

1.2.1. Khái niệm bài toán và thuật toán

Định nghĩa: *Bài toán tính toán* F là ánh xạ từ tập các xâu nhị phân độ dài hữu hạn vào tập các xâu nhị phân độ dài hữu hạn:

$$F : \{0, 1\}^* \rightarrow \{0, 1\}^*.$$

Ví dụ:

– Mỗi số nguyên x đều có thể biểu diễn dưới dạng xâu nhị phân là cách viết trong hệ đếm nhị phân của nó.

– Hệ phương trình tuyến tính $Ax = b$ có thể biểu diễn dưới dạng xâu là ghép nối của các xâu biểu diễn nhị phân của các thành phần của ma trận A và vecto b .

– Đa thức một biến $P(x) = a_0 + a_1 x + \dots + a_n x^n$ hoàn toàn xác định bởi dãy số n , a_0, a_1, \dots, a_n , mà để biểu diễn dãy số này chúng ta có thể sử dụng xâu nhị phân.

Định nghĩa: *Ta hiểu thuật toán giải bài toán đặt ra là một thủ tục xác định bao gồm một dãy hữu hạn các bước cần thực hiện để thu được đầu ra cho một đầu vào cho trước của bài toán.*

Thuật toán có các đặc trưng sau đây:

– **Đầu vào (Input):** Thuật toán nhận dữ liệu vào từ một tập nào đó.

– **Đầu ra (Output):** Với mỗi tập các dữ liệu đầu vào, thuật toán đưa ra các dữ liệu tương ứng với lời giải của bài toán.

– **Chính xác (Precision):** Các bước của thuật toán được mô tả chính xác.

– **Hữu hạn (Finiteness):** Thuật toán cần phải đưa được đầu ra sau một số hữu hạn (có thể rất lớn) bước với mọi đầu vào.

– **Đơn trị (Uniqueness):** Các kết quả trung gian của từng bước thực hiện thuật toán được xác định một cách đơn trị và chỉ phụ thuộc vào đầu vào cũng như các kết quả của các bước trước.

– **Tổng quát (Generality):** Thuật toán có thể áp dụng để giải mọi bài toán có dạng đã cho.

1.2.2. Độ phức tạp của thuật toán

Dánh giá độ phức tạp tính toán của thuật toán là đánh giá lượng tài nguyên các loại mà thuật toán đòi hỏi sử dụng. Có hai loại tài nguyên quan trọng là thời gian và bộ nhớ. Trong giáo trình này ta đặc biệt quan tâm đến việc đánh giá thời gian cần thiết để thực hiện thuật toán mà ta sẽ gọi là *thời gian tính* của thuật toán.

Rõ ràng thời gian tính phụ thuộc vào dữ liệu vào.

Định nghĩa: *Ta gọi kích thước dữ liệu đầu vào (hay độ dài dữ liệu vào) là số bit cần thiết để biểu diễn nó.*

Ta sẽ tìm cách đánh giá thời gian tính của thuật toán bởi một *hàm của độ dài dữ liệu* vào.

Phép toán cơ bản: Đo thời gian tính bằng đơn vị đo nào?

Định nghĩa: *Ta gọi phép toán cơ bản là phép toán có thể thực hiện với thời gian bị chặn bởi một hằng số không phụ thuộc vào kích thước dữ liệu.*

Để tính toán thời gian tính của thuật toán ta sẽ đếm số phép toán cơ bản mà nó phải thực hiện.

1.2.3. Các loại thời gian tính

Chúng ta sẽ quan tâm đến:

- Thời gian tối thiểu cần thiết để thực hiện thuật toán với mọi bộ dữ liệu đầu vào kích thước n . Thời gian như vậy sẽ được gọi là *thời gian tính tốt nhất* của thuật toán với đầu vào kích thước n .
- Thời gian nhiều nhất cần thiết để thực hiện thuật toán với mọi bộ dữ liệu đầu vào kích thước n . Thời gian như vậy sẽ được gọi là *thời gian tính tồi nhất* của thuật toán với đầu vào kích thước n .
- Thời gian trung bình cần thiết để thực hiện thuật toán trên tập hữu hạn các đầu vào kích thước n . Thời gian như vậy sẽ được gọi là *thời gian tính trung bình* của thuật toán.

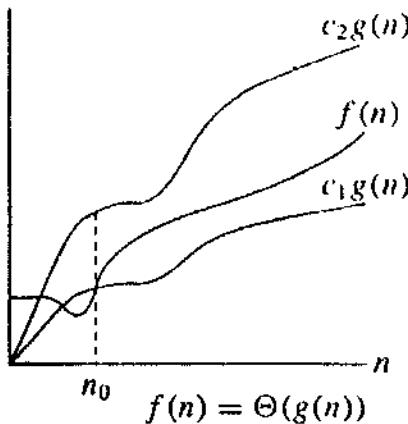
1.3. KÝ HIỆU TIỆM CĂN

1.3.1. Ký hiệu Θ

Đối với hàm $g(n)$ cho trước, ta ký hiệu $\Theta(g(n))$ là tập các hàm.

$\Theta(g(n)) = \{f(n): \exists$ các hằng số c_1, c_2 và n_0 sao cho $0 \leq c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0\}$.

Hình vẽ dưới đây minh họa ý nghĩa hình học của định nghĩa.



Ta nói rằng $g(n)$ là đánh giá tiệm cận đúng cho $f(n)$ và viết $f(n) \in \Theta(g(n))$. Chú ý rằng thay vì viết $f(n) \in \Theta(g(n))$, người ta cũng rất hay viết: $f(n) = \Theta(g(n))$.

Ví dụ: Cần chỉ ra rằng:

$$10n^2 - 3n = \Theta(n^2).$$

Ta cần xác định với giá trị nào của các hằng số n_0 , c_1 và c_2 thì bất đẳng thức trong định nghĩa là đúng. Nếu lấy c_1 bé hơn hệ số của số hạng với số mũ cao nhất, còn c_2 lấy lớn hơn, ta có:

$$n^2 \leq 10n^2 - 3n \leq 11n^2, \text{ với mọi } n \geq 1.$$

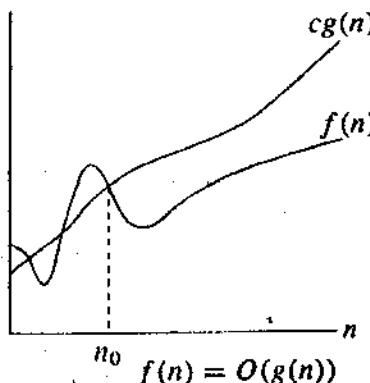
Kết quả này dễ dàng tổng quát như sau: đối với hàm đa thức, để so sánh tốc độ tăng cần nhìn vào số hạng với số mũ cao nhất.

1.3.2. Ký hiệu O (đọc là ô lớn – big O)

Đối với hàm $g(n)$ cho trước, ta ký hiệu $O(g(n))$ là tập các hàm.

$$O(g(n)) = \{f(n) : \exists \text{ các hằng số dương } c \text{ và } n_0 \text{ sao cho: } f(n) \leq cg(n) \forall n \geq n_0\}.$$

Hình vẽ dưới đây minh họa ý nghĩa hình học của định nghĩa.



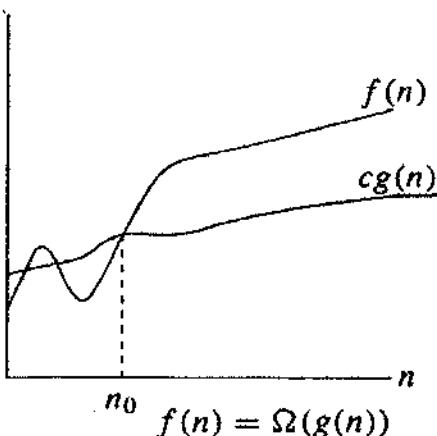
Ta nói $g(n)$ là *cận trên tiệm cận* của $f(n)$ và viết $f(n) \in O(g(n))$ hoặc $f(n) = O(g(n))$.

1.3.3. Ký hiệu Ω

Đối với hàm $g(n)$ cho trước, ta ký hiệu $\Omega(g(n))$ là tập các hàm.

$$\Omega(g(n)) = \{f(n): \exists \text{ các hằng số dương } c \text{ và } n_0 \text{ sao cho: } cg(n) \leq f(n) \forall n \geq n_0\}.$$

Hình vẽ dưới đây minh họa ý nghĩa hình học của định nghĩa.



Ta nói $g(n)$ là *cận dưới tiệm cận* cho $f(n)$ và viết $f(n) \in \Omega(g(n))$ hoặc $f(n) = \Omega(g(n))$.

1.3.4. Sử dụng ký hiệu tiệm cận Θ , Ω , O

Liên hệ giữa Θ , Ω , O

Đối với hai hàm bất kỳ $g(n)$ và $f(n)$,

$$f(n) = \Theta(g(n))$$

khi và chỉ khi:

$$f(n) = O(g(n)) \text{ và } f(n) = \Omega(g(n)),$$

tức là:

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)).$$

Cách nói về thời gian tính

– Câu nói “Thời gian tính là $O(f(n))$ ” được hiểu là: đánh giá trong tình huống tồi nhất (worst case) là $O(f(n))$. Thường nói: “Đánh giá thời gian tính trong tình huống tồi nhất là $O(f(n))$ ”, nghĩa là thời gian tính trong tình huống tồi nhất được xác định bởi một hàm nào đó $g(n) \in O(f(n))$.

– Câu nói “Thời gian tính là $\Omega(f(n))$ ” được hiểu là: đánh giá trong tình huống tốt nhất (best case) là $\Omega(f(n))$. Thường nói: “Đánh giá thời gian tính trong tình huống tốt

nhất là $\Omega(f(n))$ ", nghĩa là thời gian tính trong tình huống tốt nhất được xác định bởi một hàm nào đó $g(n) \in \Omega(f(n))$.

Ví dụ:

- *Sắp xếp chèn* đòi hỏi thời gian $\Theta(n^2)$ trong tình huống tồi nhất, vì thế bài toán sắp xếp có thời gian là $O(n^2)$.
- Mọi thuật toán sắp xếp đều đòi hỏi duyệt qua tất cả các phần tử, vì thế bài toán sắp xếp có thời gian $\Omega(n)$ trong tình huống tốt nhất.
- Trên thực tế, sử dụng (chẳng hạn) sắp xếp trộn, bài toán sắp xếp có thời gian $\Theta(n \log n)$ trong tình huống tồi nhất.

Ký hiệu tiệm cận trong các đẳng thức

Trong các đẳng thức, ký hiệu tiệm cận được sử dụng để thay thế các biểu thức chứa các toán hạng với tốc độ tăng chậm.

Ví dụ:

$$\begin{aligned}4n^3 + 3n^2 + 2n + 1 &= 4n^3 + 3n^2 + \Theta(n) \\&= 4n^3 + \Theta(n^2) = \Theta(n^3)\end{aligned}$$

Trong các đẳng thức, $\Theta(f(n))$ thay thế cho một hàm nào đó $g(n) \in \Theta(f(n))$. Trong ví dụ trên, $\Theta(n^2)$ thay thế cho $3n^2 + 2n + 1$.

Sự tương tự giữa so sánh các hàm số và so sánh các số

$$f \leftrightarrow g \approx a \leftrightarrow b$$

$$f(n) = O(g(n)) \approx a \leq b$$

$$f(n) = \Omega(g(n)) \approx a \geq b$$

$$f(n) = \Theta(g(n)) \approx a = b$$

Liên hệ với khái niệm giới hạn

- Nếu $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$ thì $f(n) \in O(g(n))$.
- Nếu $0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$ thì $f(n) \in \Theta(g(n))$.
- Nếu $0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ thì $f(n) \in \Omega(g(n))$.
- Nếu $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ không xác định, thì không thể đưa ra kết luận gì.

Các tính chất

– Truyền ứng:

$$f(n) = \Theta(g(n)) \text{ & } g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n));$$

$$f(n) = O(g(n)) \text{ & } g(n) = O(h(n)) \Rightarrow f(n) = O(h(n));$$

$$f(n) = \Omega(g(n)) \text{ & } g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n)).$$

– Đổi xứng:

$$f(n) = \Theta(g(n)) \text{ khi và chỉ khi } g(n) = \Theta(f(n)).$$

– Đổi xứng chuyển vị:

$$f(n) = O(g(n)) \text{ khi và chỉ khi } g(n) = \Omega(f(n)).$$

Ví dụ: Xác định các hằng số trong các định nghĩa để chứng minh các đánh giá tiệm cận sau đây:

$$- 2n^2 = O(n^3). \text{ Ta cần có } 2n^2 \leq cn^3 \Rightarrow 2 \leq cn \Rightarrow c = 1 \text{ và } n_0 = 2.$$

$$- n^2 = O(n^2). \text{ Ta cần có } n^2 \leq cn^2 \Rightarrow 1 \leq c \Rightarrow c = 1 \text{ và } n_0 = 1.$$

$$- 1000n^2 + 1000n = O(n^2). \text{ Ta cần có } 1000n^2 + 1000n \leq cn^2 \Rightarrow c = 1001 \text{ và } n_0 = 1.$$

$$- n = O(n^2). \text{ Ta cần có } n \leq cn^2 \Rightarrow 1 \leq cn \Rightarrow c = 1 \text{ và } n_0 = 1.$$

$$\begin{aligned} - 5n^2 = \Omega(n). \text{ Ta cần tìm } c, n_0 \text{ sao cho: } 0 \leq cn \leq 5n^2 \forall n \geq n_0 \text{ hay } cn \leq 5n^2 \\ \Rightarrow c = 1 \text{ và } n_0 = 1. \end{aligned}$$

$$- 100n + 5 \neq \Omega(n). \text{ Giả sử tồn tại } c, n_0 \text{ sao cho: } 0 \leq cn^2 \leq 100n + 5 \forall n \geq n_0.$$

Ta có: $100n \leq 100n + 5n$, suy ra $cn^2 \leq 105n$ hay $n(cn - 105) \leq 0$. Do n và c là các số dương nên từ đó suy ra $n \leq 105/c$. Đó là điều không thể xảy ra, do n không thể nhỏ hơn hằng số.

Chú ý: Giá trị của n_0 và c *không phải là duy nhất* trong chứng minh công thức tiệm cận.

Ví dụ, ta cần chứng minh $100n + 5 = O(n^2)$. Ta có:

$$100n + 5 \leq 100n + n = 101n \leq 101n^2 \text{ với mọi } n \geq 1.$$

Như vậy, $n_0 = 5$ và $c = 101$ là các hằng số cần tìm. Ta cũng có:

$$100n + 5 \leq 100n + 5n = 105n \leq 105n^2 \text{ với mọi } n \geq 1,$$

do đó $n_0 = 1$ và $c = 105$ cũng là các hằng số cần tìm.

Khi chứng minh các công thức tiệm cận ta chỉ cần tìm các hằng c và n_0 nào đó thỏa mãn bất đẳng thức trong định nghĩa công thức tiệm cận.

Ví dụ: Với mỗi cặp hàm sau đây, hoặc $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, hoặc $f(n) = \Theta(g(n))$. Hãy xác định quan hệ nào là đúng.

$f(n)$	$g(n)$	<i>So sánh</i>
$f(n) = \log n^2$	$g(n) = \log n + 5$	$f(n) = \Theta(g(n))$
$f(n) = n$	$g(n) = \log n^2$	$f(n) = \Omega(g(n))$
$f(n) = \log \log n$	$g(n) = \log n$	$f(n) = O(g(n))$
$f(n) = n$	$g(n) = \log_2 n$	$f(n) = \Omega(g(n))$
$f(n) = n \log n + n$	$g(n) = \log n$	$f(n) = \Omega(g(n))$
$f(n) = 10$	$g(n) = \log 10$	$f(n) = \Theta(g(n))$
$f(n) = 2n$	$g(n) = 10n^2$	$f(n) = \Omega(g(n))$
$f(n) = 2n; g(n) = 3n$	$g(n) = 3n$	$f(n) = O(g(n))$

1.3.5. Một số lớp thuật toán

Tên gọi của một số đánh giá trong ký hiệu O :

- $O(1)$: hằng số (constant).
- $O(\log n)$: logarithmic.
- $O(n)$: tuyến tính (linear).
- $O(n \log n)$: trên tuyến tính (superlinear).
- $O(n^2)$: bình phương (quadratic).
- $O(n^3)$: bậc ba (cubic).
- $O(a^n)$: hàm mũ (exponential) ($a > 1$).
- $O(n^k)$: đa thức (polynomial) ($k \geq 1$).

Thuật toán có đánh giá thời gian tính là $O(n^k)$ được gọi là thuật toán thời gian tính đa thức (hay vẫn tắt: thuật toán đa thức). Các thuật toán đa thức được coi là thuật toán hiệu quả. Các thuật toán với thời gian tính hàm mũ là không hiệu quả.

Liên quan đến khái niệm độ phức tạp tính toán của bài toán chúng ta có các định nghĩa sau đây:

Định nghĩa: Cho bài toán P , ta nói cận trên cho thời gian tính của P là $O(g(n))$ nếu để giải P tồn tại thuật toán giải với thời gian tính là $O(g(n))$.

Định nghĩa: Cho bài toán P , ta nói cận dưới cho thời gian tính của P là $\Omega(g(n))$ nếu mọi thuật toán giải P đều có thời gian tính là $\Omega(g(n))$.

Định nghĩa: Cho bài toán P , ta nói thời gian tính của P là $\Theta(g(n))$ nếu P có cận trên là $O(g(n))$ và cận dưới là $\Omega(g(n))$.

Một bài toán được gọi là *dễ giải* nếu như nó có thể giải được nhờ thuật toán đa thức. Rõ ràng, các bài toán có cận trên cho thời gian tính là đa thức là các bài toán dễ giải. Ví dụ: bài toán dãy con lớn nhất, bài toán sắp xếp dãy n số,...

Một bài toán được gọi là *khó giải* nếu như nó không thể giải được bằng thuật toán đa thức. Rõ ràng, các bài toán có cận dưới cho thời gian tính là hàm mũ thuộc vào lớp bài toán này. Ví dụ: Bài toán liệt kê các hoán vị của n số, Bài toán liệt kê các xâu nhị phân độ dài n ,...

Một dạng bài toán nữa cũng được *coi là khó giải*, đó là những bài toán cho đến thời điểm hiện tại vẫn chưa tìm được thuật toán đa thức để giải nó. Ví dụ: Bài toán cái túi, Bài toán người du lịch,...

Một bài toán được gọi là *không giải được* nếu như không tồn tại thuật toán để giải nó. Ví dụ: Bài toán về tính dừng, Bài toán về nghiệm nguyên của đa thức,...

1.4. GIÁ NGÔN NGỮ

Để mô tả thuật toán có thể sử dụng một ngôn ngữ lập trình nào đó. Tuy nhiên điều đó có thể làm cho việc mô tả thuật toán trở nên phức tạp đồng thời rất khó nắm bắt. Vì thế, để mô tả thuật toán, người ta thường sử dụng *giá ngôn ngữ*, trong đó cho phép vừa mô tả thuật toán bằng ngôn ngữ đời thường vừa sử dụng những cấu trúc lệnh tương tự như của ngôn ngữ lập trình. Dưới đây ta liệt kê một số câu lệnh chính được sử dụng trong giáo trình để mô tả thuật toán.

1.4.1. Khai báo biến

```
integer x,y;  
real u, v;  
boolean a, b;  
char c, d;  
datatype x.
```

1.4.2. Câu lệnh gán

Câu lệnh gán có dạng:

$x = \text{expression};$

hoặc:

$x \leftarrow \text{expression};$

Ví dụ: $x \leftarrow 1 + 4; y = a * y + 2;$

1.4.3. Các cấu trúc điều khiển

Câu lệnh if:

```
if condition then  
    dãy câu lệnh  
else  
    dãy câu lệnh  
endif;
```

1.4.4. Câu lệnh lặp

Vòng lặp while:

```
while condition do  
    dãy câu lệnh  
endwhile;
```

Câu lệnh repeat:

```
repeat  
    dãy câu lệnh  
until condition;
```

Vòng lặp for:

```
for i=n1 to n2 [step d]  
    dãy câu lệnh  
endfor;
```

Câu lệnh case:

```
Case  
    cond1: stat1;  
    cond2: stat2;  
    ...  
    condn: stat n;  
endcase;
```

1.4.5. Câu lệnh Vào–Ra

```
read(X); /* X là biến đơn hoặc mảng */  
print(data) hoặc print(thông báo)
```

1.4.6. Hàm và thủ tục (Function and procedure)

Function name(các tham số)

begin

mô tả biến;

các câu lệnh trong thân của hàm;

return (giá trị)

end;

Procedure name(các tham số)

begin

mô tả biến;

các câu lệnh trong thân của hàm;

end;

Ví dụ: Thuật toán tìm phần tử lớn nhất trong mảng A(1:n).

Function max(A(1:n))

begin

datatype x; /* để giữ giá trị lớn nhất tìm được */

integer i;

x=A[1];

for i=2 **to** n **do**

if x < A[i] **then**

x=A[i];

endif

endfor ;

return (x);

end max;

Ví dụ: Thuật toán hoán đổi nội dung hai biến.

Procedure swap(x, y)

begin

temp=x;

x = y;

y = temp;

end swap;

Ví dụ: Tìm số nguyên tố lớn hơn số nguyên dương n.

- Trước hết ta xây dựng hàm kiểm tra xem một số nguyên dương m có phải là số nguyên tố hay không (hàm Is_prime).
- Sử dụng hàm này ta xây dựng thuật toán giải bài toán đặt ra.
- Do ước số nguyên tố của số nguyên dương m bao giờ cũng không vượt quá \sqrt{m} , nên m sẽ là số nguyên tố nếu như nó không có ước số nào trong các số nguyên dương từ 2 đến $\lfloor \sqrt{m} \rfloor$.

Thuật toán kiểm tra một số nguyên dương có phải là nguyên tố hay không

Đầu vào: số nguyên dương m .

Đầu ra: true nếu m là số nguyên tố, false nếu ngược lại.

```
function Is_prime(m);
```

```
begin
```

```
    i = 2; m2 = sqrt(m); /* m2 là căn bậc 2 của m */
```

```
    while (i <= m2) and (m mod i ≠ 0) do i=i+1;
```

```
    Is_prime = i > m2;
```

```
end Is_Prime;
```

Thuật toán tìm số nguyên tố lớn hơn số nguyên dương n

Thuật toán sẽ sử dụng Is_prime như chương trình con.

Đầu vào: số nguyên dương n .

Đầu ra: m – số nguyên tố lớn hơn n .

```
procedure Lagre_Prime(n);
```

```
begin
```

```
    m = n+1;
```

```
    while not Is_prime(m) do m=m+1;
```

```
end;
```

Do tập các số nguyên tố là vô hạn, nên thuật toán Lagre_Prime là hữu hạn.

1.5. MỘT SỐ KỸ THUẬT PHÂN TÍCH THUẬT TOÁN

1.5.1. Cấu trúc tuần tự

Giả sử P và Q là hai đoạn của thuật toán, có thể là một câu lệnh nhưng cũng có thể là một thuật toán con. Gọi $Time(P)$, $Time(Q)$ là thời gian tính của P và Q tương ứng. Khi đó ta có:

Quy tắc tuần tự: Thời gian tính đòi hỏi bởi “ $P; Q$ ”, nghĩa là P thực hiện trước, tiếp đến là Q , sẽ là:

$$Time(P; Q) = Time(P) + Time(Q),$$

hoặc trong ký hiệu Theta:

$$Time(P; Q) = \Theta(\max(Time(P), Time(Q))).$$

1.5.2. Phân tích vòng lặp for

for i = 1 **to** m **do** P(i);

Giả sử thời gian thực hiện $P(i)$ là $t(i)$. Khi đó thời gian thực hiện vòng lặp for sẽ là $\sum_{i=1}^m t(i)$.

Ví dụ: Tính số Fibonaci.

```
function Fibiter(n)
begin
    i = 0; j = 1;
    for k = 2 to n do
        begin
            j = j + i;
            i = j - i;
        end;
    Fibiter = j;
end;
```

Nếu coi các phép toán số học đòi hỏi thời gian là hằng số c và không tính đến chi phí tổ chức vòng lặp for thì thời gian tính của hàm trên là $\Theta(n)$.

Theo công thức Muavre ta có:

$$f_k = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right),$$

suy ra số bit biểu diễn f_k là $\Theta(k)$. Do đó thời gian tính của một lần lặp là $\Theta(k)$. Vậy thời gian tính của Fibiter là:

$$\sum_{k=1}^n c.k = c \sum_{k=1}^n k = c \frac{n(n+1)}{2} = \Theta(n^2).$$

1.5.3. Phân tích vòng lặp While và Repeat

Để phân tích vòng lặp while ta cần xác định một hàm của các biến trong vòng lặp sao cho hàm này có giá trị giảm dần trong quá trình lặp. Khi đó:

– Để chứng minh tính kết thúc của vòng lặp ta chỉ cần chỉ ra giá trị của hàm là số nguyên dương.

– Còn để xác định số lần lặp ta cần phải khảo sát xem giá trị của hàm giảm như thế nào.

Việc phân tích vòng lặp Repeat được tiến hành tương tự như phân tích vòng lặp While.

Ví dụ: Tìm kiếm nhị phân (Binary Search).

Input: Mảng $T[1..n]$ và số x

Output: Giá trị $i: 1 \leq i \leq n$ sao cho $T[i] = x$.

Giả thiết x có mặt trong mảng T .

function Binary-Search($T[1..n]$, x);

begin

$i:=1$; $j:=n$;

while $i < j$ **do**

$k:=(i+j) \text{ div } 2$;

case

$x < T[k]$: $j:=k-1$;

$x = T[k]$: $i:=k$; $j:=k$; Binary-Search= k ; exit; {Found!}

$x > T[k]$: $i:=k+1$;

end case

endwhile

end;

Gọi $d = j - i + 1$. d – số phần tử của mảng cần tiếp tục khảo sát. Ký hiệu i, j, i^*, j^* theo thứ tự là giá trị của i, j trước và sau khi thực hiện vòng lặp. Khi đó:

– Nếu $x < T[k]$, thì $i^* = i$, $j^* = (i + j) \text{ div } 2 - 1$, vì thế $d^* = j^* - i^* + 1 = (i + j) \text{ div } 2 - 1 - i + 1 \leq (i + j)/2 - i < (j - i + 1)/2 = d/2$.

– Nếu $x > T[k]$, thì $j^* = j$, $i^* = (i + j) \text{ div } 2 + 1$, vì thế $d^* = j^* - i^* + 1 = j - (i + j) \text{ div } 2 \leq j - (i + j - 1)/2 - i = (j - i + 1)/2 = d/2$.

– Nếu $x = T[k]$, thì $d^* = 1$ còn $d \geq 2$.

Vậy luôn có: $d^* \leq d/2$.

Do vòng lặp kết thúc khi $d \leq 1$, nên từ đó suy ra thuật toán phải kết thúc. Gọi d_p là giá trị của $j - i + 1$ ở lần lặp thứ $p \geq 1$ và $d_0 = n$. Khi đó:

$$d_p \leq d_{p-1}/2, p \geq 1.$$

Thuật toán kết thúc tại bước p nếu $d_p \leq 1$, điều đó xảy ra khi $p = \lceil \log n \rceil$.

Vậy thời gian tính của thuật toán là $O(\log n)$.

Hàm tìm kiếm nhị phân có thể viết trên C như sau:

```
boolean binary_search_1(int* a, int n, int x) {
    /* Test xem x có mặt trong mảng a[] kích thước n. */
    int i;
    while (n > 0) {
        i = n / 2;
        if (a[i] == x)
            return true;
        if (a[i] < x) /* Tiếp tục tìm ở nửa trái */
            a = &a[i + 1];
        n = n - i - 1;
    }
    else /* Tiếp tục tìm ở nửa phải */
        n = i;
}
return false;
}
```

1.5.4. Câu lệnh đặc trưng

Định nghĩa: Câu lệnh đặc trưng là câu lệnh được thực hiện thường xuyên ít nhất là cũng như bất kỳ câu lệnh nào trong thuật toán.

Nếu giả thiết thời gian thực hiện mỗi câu lệnh bị chặn bởi hằng số thì thời gian tính của thuật toán sẽ cùng cỡ với số lần thực hiện câu lệnh đặc trưng. Do đó để đánh giá thời gian tính có thể đếm số lần thực hiện câu lệnh đặc trưng.

Ví dụ: FibIter.

- Câu lệnh đặc trưng là $j = j + i$.
- Số lần thực hiện câu lệnh đặc trưng là n .
- Vậy thời gian tính của thuật toán là $O(n)$.

Ví dụ: Thuật toán một ví dụ mở đầu.

```
int maxSum = 0;
for (int i=0; i<n; i++) {
    for (int j=i; j<n; j++) {
        int sum = 0;
        for (int k=i; k<=j; k++)
            sum += a[k];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```

```

        sum += a[k];
    if sum > maxSum
        maxSum = sum;
    }
}

```

Chọn câu lệnh đặc trưng là $\text{sum} + \text{a}[k]$. Suy ra đánh giá thời gian tính của thuật toán là $O(n^3)$.

Nhận xét: Khi thuật toán đòi hỏi thực hiện nhiều vòng lặp lồng nhau, thì có thể lấy câu lệnh nằm trong vòng lặp trong cùng làm câu lệnh đặc trưng. Tuy vậy, cũng cần hết sức thận trọng khi sử dụng cách lựa chọn này.

Ví dụ: Sắp xếp kiểu nhốt chim vào lồng.

Bài toán đặt ra là: Sắp xếp n số nguyên dương có giá trị nằm trong khoảng $1..s$.

Đầu vào: $T[1..n]$: mảng chứa dãy cần sắp xếp.

Đầu ra: $T[1..n]$: mảng chứa dãy được sắp xếp không giảm.

Thuật toán bao gồm hai thao tác:

- **Đếm chim:** Xây dựng mảng $U[1..s]$, trong đó phần tử $U[k]$ đếm số lượng số có giá trị là k trong mảng T .

- **Nhốt chim:** Lần lượt nhốt $U[1]$ con chim loại 1 vào $U[1]$ lồng đầu tiên, $U[2]$ con chim loại 2 vào $U[2]$ lồng tiếp theo, ...

procedure Pigeonhole-Sorting;

begin

for $i:=1$ **to** n **do inc**($U[T[i]]$); (* đếm chim *)

$i:=0$;

for $k:=1$ **to** s **do** (* nhốt chim *)

while $U[k]>0$ **do**

begin

$i:=i+1$;

$T[i]:=k$;

$U[k]:=U[k]-1$;

end;

end;

Nếu chọn câu lệnh bất kỳ trong vòng lặp while làm câu lệnh đặc trưng, thì rõ ràng với mỗi k , câu lệnh này phải thực hiện $U[k]$ lần. Do đó số lần thực hiện câu lệnh đặc trưng là:

$$\sum_{k=1}^s U(k) = n$$

(do có tất cả n số cần sắp xếp). Từ đó ta có thời gian tính là $\Theta(n)$.

Ví dụ sau đây cho thấy đánh giá đó chưa chính xác.

Giả sử $T[i] = i^2$, $i = 1, \dots, n$. Ta có:

$$U[k] = \begin{cases} 1, & \text{nếu } k \text{ là số chính phương,} \\ 0, & \text{nếu trái lại.} \end{cases}$$

Khi đó $s = n^2$. Rõ ràng thời gian tính của thuật toán không phải là $O(n)$, mà là $O(n^2)$.

Lỗi ở đây là do ta xác định câu lệnh đặc trưng chưa đúng, các câu lệnh trong thân vòng lặp *while* không thể dùng làm câu lệnh đặc trưng trong trường hợp này, do có rất nhiều vòng lặp rỗng (khi $U[k] = 0$).

Nếu ta chọn câu lệnh kiểm tra $U[k] < 0$ làm câu lệnh đặc trưng thì rõ ràng số lần thực hiện nó sẽ là $n + s$. Vậy thuật toán có thời gian tính $O(n+s) = O(n^2)$.

Chú ý: Nếu $s < n$, thì sắp xếp kiểu nhốt chim vào lồng là thuật toán sắp xếp đặc biệt có thời gian tính tuyến tính.

BÀI TẬP CHƯƠNG 1

1. So sánh hai hàm sau đây trong ký hiệu O :
 - a) $N \log N$ và N^2 : $N \log N = O(N^2)$ hay $N^2 = O(N \log N)$?
 - b) $N \log N$ và N : $N \log N = O(N)$ hay $N = O(N \log N)$?
2. Đưa ra đánh giá trong ký hiệu O cho các hàm sau đây:
 - a) $N^2 + N \log N = O(\dots)$.
 - b) $N + N \log N = O(\dots)$.
 - c) $N^2 + \log N = O(\dots)$.
3. Cho hàm:

$$T(n) = 100n^{3/2} + 500n + 1000$$
 xác định với đối số nguyên dương n .
 - a) Chứng minh rằng $T(n) \in O(n^2)$.
 - b) Chứng minh rằng $T(n) \notin \Omega(n^2)$.

4. Điền vào chỗ trống trong ngoặc hàm $f(N)$ đánh giá sát nhất cho hàm vế trái:
- $N + \log N = \Theta(\)$.
 - $N + N^2 \log N = \Theta(\)$.
 - $N^2 + (\log N)^3 = \Theta(\)$.
 - $N^2 + \log N = \Theta(\)$.
 - $N \log N = \Theta(\)$.
 - $\log(N^2) = \Theta(\)$.
 - $(\log(N^2))^2 = \Theta(\)$.
5. Cho biết đánh giá nào trong các đánh giá sau đây là đúng/sai và giải thích câu trả lời:
- $N^2 + (\log N)^6 = O((\log N)^6)$.
 - $N^2 + \log N = O(N \log N)$.
 - $N^2 + \log N = O(N^2)$.
 - $\log(N^2) = O(N^2 \log N)$.
 - $(\log(N^2))^2 = O(N^2 \log N)$.
 - $(\log(N^2))^2 = O(N^2)$.
 - $(\log(N^2))^2 = O(N)$.
 - $(\log(N^2))^2 = O(\log(N))$.
 - $N^2 + \log N = O((\log N)^3)$.
 - $N^2 + N^2 \log N = O(N^2)$.
 - $N^2 \log N = O(N^2)$.
 - $N^2 = O(N \log N)$.
 - $N^2 + N \log N = O(N \log N)$.
 - $N^2 + N \log N = O(N^2 \log N)$.
 - $2^{n+1} = O(2^n)$.
 - $2^{2^n} = O(2^n)$.
6. Ta nói $f(n)$ có tốc độ tăng chậm hơn $g(n)$ nếu $f(n) = O(g(n))$. Sắp xếp các hàm sau đây theo thứ tự không giảm của tốc độ tăng (hai hàm có tốc độ tăng như nhau có thể xếp theo thứ tự tùy ý):
- $\log_2(n), n^{1/2}, n \log_2(n), n^3, \log_{10}(n), 10^9, n^{3/2}, n^2 \log_{10}(n), e^n, n$.
 - $n^2, 2^n, n \log n, n!, n^6 + n^2, n^4, n^6 - n^3, 4^n, n, n^n, 2^{2^n}$.

7. Đánh giá thời gian tính của các đoạn chương trình sau:

a)

```
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < n * n; j++)
        sum++;
```

b)

```
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < i; j++)
        sum++;
```

c)

```
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < i*i; j++)
        for( k = 0; k < j; k++)
            sum++;
```

d)

```
sum = 0;
for( i = 0; i < n; i++)
    sum++;
val = 1;
for( j = 0; j < n*n; j++)
    val = val * j;
```

e)

```
sum = 0;
for( i = 0; i < n; i++)
    sum++;
for( j = 0; j < n*n; j++)
    compute_val(sum, j);
```

f)

```
S=0;
for (i=1; i<=n-2; i++)
    for (j=i+1; j<=n-1; j++)
        for (k=j+1; k<=n; k++)
            S=S+1;
```

Biết rằng thời gian tính của hàm `compute_val(x, y)` là $O(n \log n)$.

```

g)
S=0;
for (i=1; i<=n-2; i++)
    for(j=n; j>0; j=j/2)
        S=S+1;

```

8. Xét thuật toán đệ quy sau đây:

```

ALGORITHM Q(n)
// input: số nguyên dương n
if n = 1 return 1
else return Q(n - 1) + 2*n - 1

```

- a) Xác định xem thuật toán tính giá trị gì: Xây dựng công thức đệ quy cho $Q(n)$ và giải công thức đệ quy.
- b) Xây dựng công thức đệ quy đếm số phép nhân mà thuật toán phải thực hiện và giải công thức thu được.
- c) Xây dựng công thức đệ quy đếm số phép toán cộng/trừ mà thuật toán phải thực hiện và giải công thức thu được.
- d) Xây dựng công thức đệ quy cho $T(n)$ là thời gian tính của thuật toán và giải công thức.

9. Xét thuật toán sau đây để tìm khoảng cách giữa hai số gần nhau nhất trong mảng số:

```

ALGORITHM MinDistance (A[0..n-1])
// Input: Mảng A[0..n-1] gồm các số nguyên
// Output: Khoảng cách nhỏ nhất giữa hai số trong
mảng

dmin ← ∞
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        if i ≠ j and |A[i] - A[j]| < dmin
            dmin ← |A[i] - A[j]|
return dmin

```

Hãy chỉ ra cách tăng hiệu quả của thuật toán trên. Trình bày thuật toán để xuất và đưa ra đánh giá hiệu quả của thuật toán.

10. Có n người được gán số hiệu từ 1 đến n . Phần tử $K[i][j]$ của mảng $K[1..n][1..n]$ là 1 nếu người i biết người j và là 0 nếu ngược lại. (Ta giả thiết $K[i][i] = 1$ với mọi i và nếu người i biết người j thì không nhất thiết người j cũng phải biết người i). Một người được gọi là người đặc biệt nếu như ngoại trừ chính mình, anh ta không biết ai cả nhưng tất cả những người còn lại đều biết anh ta. Ví dụ, trong hai bảng dưới đây, bảng bên trái có người 3 là người đặc biệt, còn bảng bảng bên phải không có người đặc biệt.

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

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

Bài toán đặt ra là: Cho bảng K, hãy tìm người đặc biệt hoặc trả lời là không có người như vậy. Thuật toán trực tiếp giải bài toán đặt ra có thời gian tính $O(n^2)$. Hãy phát triển thuật toán thời gian tính $O(n)$ để giải bài toán.

Chương 2

THUẬT TOÁN ĐỆ QUY

2.1. KHÁI NIỆM ĐỆ QUY

Trong thực tế ta thường gặp những đối tượng bao gồm chính nó hoặc được định nghĩa dưới dạng của chính nó. Ta nói các đối tượng đó được xác định một cách đệ quy.

Ví dụ:

- Điểm quân số.
- Fractal.
- Các hàm được định nghĩa đệ quy.
- Tập hợp được định nghĩa đệ quy.
- Định nghĩa đệ quy của cây.
- ...

2.1.1. Hàm đệ quy (Recursive Functions)

Các hàm đệ quy được xác định phụ thuộc vào biến nguyên không âm n theo sơ đồ sau:

Bước cơ sở (Basic Step): Xác định giá trị của hàm tại $n = 0$: $f(0)$.

Bước đệ quy (Recursive Step): Cho giá trị của $f(k)$, $k \leq n$, đưa ra quy tắc tính giá trị của $f(n + 1)$.

Ví dụ 1: $f(0) = 3$, $n = 0$

$$f(n + 1) = 2f(n) + 3, \quad n > 0$$

Khi đó ta có: $f(1) = 2 \times 3 + 3 = 9$, $f(2) = 2 \times 9 + 3 = 21$, ...

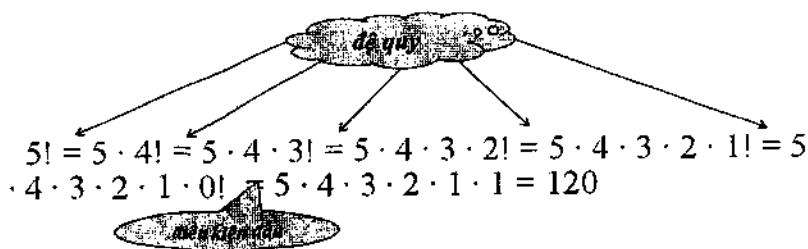
Ví dụ 2: Định nghĩa đệ quy của $n!$:

$$f(0) = 1$$

$$f(n + 1) = f(n) \times (n + 1)$$

Để tính giá trị của hàm đệ quy ta thay thế dần theo định nghĩa đệ quy để thu được biểu thức với đối số càng ngày càng nhỏ cho đến tận điều kiện đầu.

Chẳng hạn:



Ví dụ 3: Định nghĩa đệ quy của tổng $s_n = \sum_{i=1}^n a_i$.

$$s_1 = a_1,$$

$$s_n = s_{n-1} + a_n.$$

Ví dụ 4: Dãy số Fibonacci.

$$f(0) = 0, f(1) = 1,$$

$$f(n) = f(n-1) + f(n-2) \text{ với } n > 1.$$

2.1.2. Tập hợp được xác định đệ quy

Tập hợp có thể xác định một cách đệ quy theo sơ đồ tương tự như hàm đệ quy:

Bước cơ sở: định nghĩa tập cơ sở (chẳng hạn tập rỗng).

Bước đệ quy: Xác định quy tắc để sản sinh tập mới từ các tập đã có.

Ví dụ 1: Xét tập S được định nghĩa đệ quy như sau:

Bước cơ sở: 3 là phần tử của S .

Bước đệ quy: nếu x thuộc S và y thuộc S thì $x + y$ thuộc S .

Ta có một số phần tử đầu tiên của S là:

3

$$3 + 3 = 6$$

$$3 + 6 = 9 \text{ & } 6 + 6 = 12$$

...

Ví dụ 2. Định nghĩa đệ quy của xâu.

Giả sử Σ = bảng chữ cái (alphabet).

Tập Σ^* các xâu (strings) trên bảng chữ cái A được định nghĩa đệ quy như sau:

Bước cơ sở: xâu rỗng là phần tử của Σ^* .

Bước đệ quy: nếu w thuộc Σ^* và x thuộc A $\rightarrow wx$ thuộc Σ^* .

Chẳng hạn, tập các xâu nhị phân **B** thu được khi $\Sigma = \{0, 1\}$: Bắt đầu từ xâu rỗng, theo quy tắc đệ quy ta lần lượt sinh ra được các phần tử trong **B** như sau:

- 0) xâu rỗng
- 1) 0 & 1
- 2) 00 & 01 & 10 & 11
- 3) ...

Ví dụ 3: Công thức toán học.

Một *công thức hợp lệ* của các biến, các số và các phép toán từ tập $\{+, -, *, /, ^\}$ có thể định nghĩa như sau:

Bước cơ sở: x là công thức hợp lệ nếu x là biến hoặc là số.

Bước đệ quy: Nếu f, g là các công thức hợp lệ thì:

$$(f + g), (f - g), (f * g), (f / g), (f ^ g)$$

là công thức hợp lệ.

Theo định nghĩa ta có thể xây dựng các công thức hợp lệ như:

$$\begin{array}{ll} (x - y); & ((z / 3) - y); \\ ((z / 3) - (6 + 5)); & ((z / (2 * 4)) - (6 + 5)). \end{array}$$

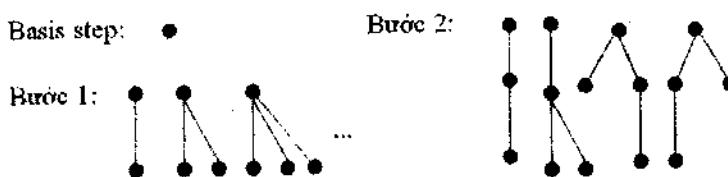
Ví dụ 4: Cây có gốc.

Cây có gốc bao gồm các nút, có một nút đặc biệt được gọi là gốc (root) và các cạnh nối các nút.

Bước cơ sở: Một nút là cây có gốc.

Bước đệ quy: Giả sử T_1, \dots, T_n, \dots là các cây với gốc là r_1, \dots, r_n, \dots

Nếu ta tạo một gốc mới r và nối gốc này với mỗi một trong số các gốc r_1, \dots, r_n, \dots bằng một cạnh tương ứng, ta thu được một cây có gốc mới.



Định nghĩa đệ quy và quy nạp

Định nghĩa đệ quy và quy nạp toán học có những nét tương đồng và bổ sung cho nhau. Định nghĩa đệ quy thường giúp cho chứng minh bằng quy nạp các tính chất của các đối tượng được định nghĩa đệ quy. Ngược lại, các chứng minh bằng quy nạp toán học thường là cơ sở để xây dựng các thuật toán đệ quy nhằm giải quyết nhiều bài toán.

Chứng minh bằng quy nạp toán học thường bao gồm hai phần:

1. **Bước cơ sở quy nạp** – giống như bước cơ sở trong định nghĩa đệ quy.
2. **Bước chuyển quy nạp** – giống như bước đệ quy.

2.2. THUẬT TOÁN ĐỆ QUY

Thuật toán đệ quy là thuật toán tự gọi đến chính mình với đầu vào kích thước nhỏ hơn.

Việc phát triển thuật toán đệ quy là thuận tiện khi cần xử lý với các đối tượng được định nghĩa đệ quy (chẳng hạn: tập hợp, hàm, cây,... trong các ví dụ nêu trong mục trước).

Các thuật toán được phát triển dựa trên phương pháp chia để trị thông thường được mô tả dưới dạng các thuật toán đệ quy (xem ví dụ mở đầu).

Các ngôn ngữ lập trình cấp cao thường cho phép xây dựng các hàm (thủ tục) đệ quy, nghĩa là trong thân của hàm (thủ tục) có chứa những lệnh gọi đến chính nó. Vì thế, khi cài đặt các thuật toán đệ quy, người ta thường xây dựng các hàm (thủ tục) đệ quy.

Cấu trúc của thuật toán đệ quy

Thuật toán đệ quy thường có cấu trúc sau đây:

Thuật toán RecAlg(input)

begin

if (kích thước của input là nhỏ nhất) **then**

 Thực hiện Bước cơ sở /* giải bài toán kích thước đầu vào nhỏ nhất */

else

 RecAlg(input với kích thước nhỏ hơn); /* bước đệ quy */

 /* có thể có thêm những lệnh gọi đệ quy */

 Tổ hợp lời giải của các bài toán con để thu được lời_giải;

return (lời_giải)

endif

end;

Thuật toán chia để trị

Thuật toán chia để trị là một trong những phương pháp thiết kế thuật toán cơ bản, bao gồm ba thao tác:

- Chia (Divide) bài toán cần giải ra thành một số bài toán con.
- + Bài toán con có kích thước nhỏ hơn và có cùng dạng với bài toán cần giải.
- Trị (Conquer) các bài toán con.
- + Giải các bài toán con một cách đệ quy.
- + Bài toán con có kích thước đủ nhỏ sẽ được giải trực tiếp.
- Tổng hợp (Combine) lời giải của các bài toán con.
- + Thu được lời giải của bài toán xuất phát.

Sơ đồ của phương pháp có thể trình bày trong thủ tục đệ quy sau đây:

procedure D-and-C (n);

begin

if n ≤ n₀ **then**

Giải bài toán một cách trực tiếp

else

begin

Chia bài toán thành α bài toán con kích thước n/b ;

for (mỗi bài toán trong α bài toán con) **do** D-and-C(n/b);

Tổng hợp lời giải của α bài toán con để thu được lời giải của bài toán gốc;

end;

end;

Các thông số quan trọng của thuật toán:

– n_0 : kích thước nhỏ nhất của bài toán con (còn gọi là neo đệ quy). Bài toán con với kích thước n_0 sẽ được giải trực tiếp.

– α : số lượng bài toán con cần giải.

– b : liên quan đến kích thước của bài toán con được chia.

Ví dụ: Sắp xếp trộn.

Bài toán: Cần sắp xếp mảng A[1 .. n]:

Chia

– Chia dãy gồm n phần tử cần sắp xếp ra thành hai dãy, mỗi dãy có $n/2$ phần tử.

Trị

– Sắp xếp mỗi dãy con một cách đệ quy sử dụng *sắp xếp trộn*.

– Khi dãy chỉ còn một phần tử thì trả lại phần tử này.

Tổng hợp

– Trộn hai dãy con được sắp xếp để thu được dãy được sắp xếp gồm tất cả các phần tử của cả hai dãy con.

Thuật toán được mô tả như sau:

MERGE-SORT(A, p, r)

if p < r // Kiểm tra điều kiện neo

then q ← ⌊(p + r)/2⌋ // Chia

 MERGE-SORT(A, p, q) // Trị

 MERGE-SORT(A, q + 1, r) // Trị

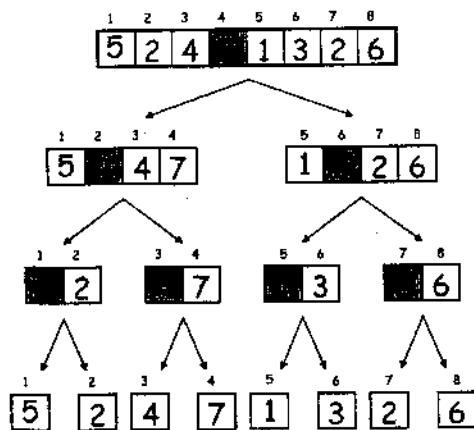
 MERGE(A, p, q, r) // Tổng hợp

endif

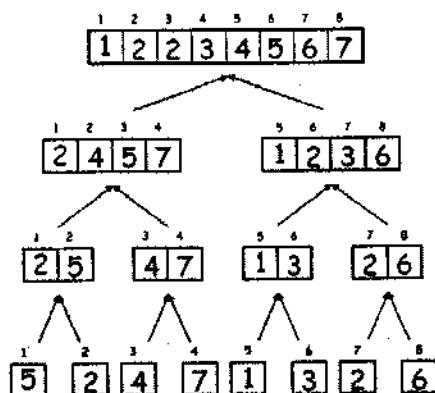
Lệnh gọi thực hiện thuật toán: MERGE-SORT(A, 1, n).

Ví dụ: Khi n là luỹ thừa của 2.

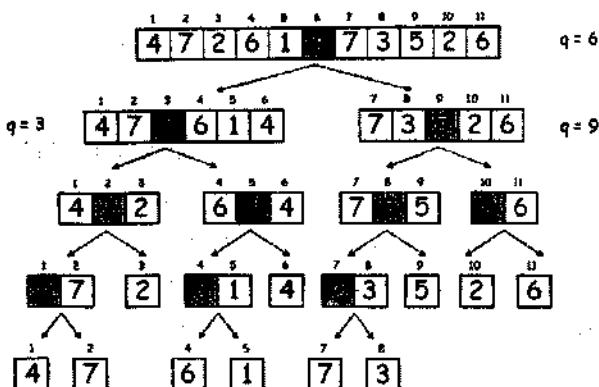
Thao tác Chia

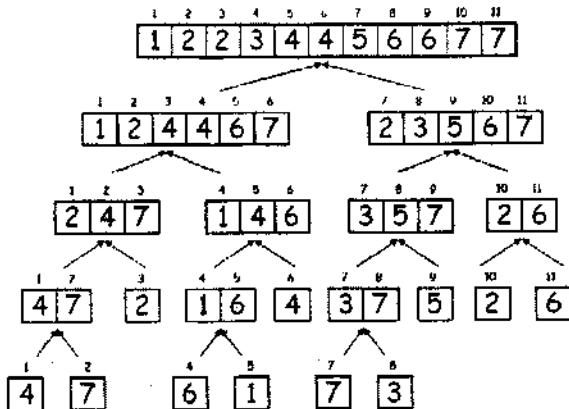


Thao tác Trị



Khi n không là luỹ thừa của 2.





Thuật toán trộn

Đầu vào: Mảng A và các chỉ số p, q, r sao cho $p \leq q < r$.

Các mảng con A[p .. q] và A[q + 1 .. r] đã được sắp xếp.

Đầu ra: Mảng con được sắp xếp A[p .. r].

Ý tưởng của thuật toán trộn:

- Có hai dãy con đã được sắp xếp:
 - + Chọn phần tử nhỏ hơn ở đầu hai dãy;
 - + Loại nó khỏi dãy con tương ứng và đưa vào dãy kết quả.
- Lặp lại điều đó cho đến khi một trong hai dãy trở thành dãy rỗng.
- Các phần tử còn lại của dãy con kia sẽ được đưa nốt vào đuôi của dãy kết quả.

MERGE(A, p, q, r)

1. Tính n_1 và n_2 .
2. Sao n_1 phần tử đầu tiên vào L[1 .. n_1] và n_2 phần tử tiếp theo vào R[1 .. n_2].
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$.
4. $i \leftarrow 1$; $j \leftarrow 1$.
5. **for** $k \leftarrow p$ **to** r **do**.
6. **if** $L[i] \leq R[j]$.
7. **then** $A[k] \leftarrow L[i]$.
8. *i* $\leftarrow i + 1$.
9. **else** $A[k] \leftarrow R[j]$.
10. *j* $\leftarrow j + 1$.

Thời gian tính của trộn

- Khởi tạo (tạo hai mảng con tạm thời L và R):

$$\Theta(n_1 + n_2) = \Theta(n).$$

- Đưa các phần tử vào mảng kết quả (vòng lặp for cuối cùng):

n lần lặp, mỗi lần đòi hỏi thời gian hằng số $\Rightarrow \Theta(n)$.

- Tổng cộng thời gian của trộn là:

$$\Theta(n).$$

2.3. MỘT SỐ VÍ DỤ MINH HỌA

Cài đặt các thuật toán đệ quy: Để cài đặt các thuật toán đệ quy trên các ngôn ngữ lập trình cấp cao quen thuộc như Pascal, C, C++, ... ta thường xây dựng các hàm (thủ tục) đệ quy.

Trong mục này ta xét một số ví dụ minh họa cài đặt thuật toán đệ quy:

- Hàm đệ quy tính $n!$.
- Hàm đệ quy tính số Fibonacci.
- Hàm đệ quy tính hệ số nhị thức.
- Tìm kiếm nhị phân.
- Bài toán tháp Hà Nội.

Ví dụ 1: Tính $n!$

Hàm $f(n) = n!$ được định nghĩa đệ quy như sau:

$$f(0) = 0! = 1, \quad n = 0,$$

$$f(n) = n f(n - 1), \quad n > 0.$$

Hàm đệ quy trên C:

```
int Fact(int n) {
    if (n==0) return 1;
    else return n*Fact(n-1);
}
```

Ví dụ 2: Tính số Fibonacci.

Dãy số Fibonacci được định nghĩa đệ quy như sau:

$$F(0) = 0, \quad F(1) = 1;$$

$$F(n) = F(n - 1) + F(n - 2), \quad n \geq 2.$$

Hàm đệ quy trên C:

```
int FibRec(int n){  
    if (n<=1) return n;  
    else return FibRec(n-1)+FibRec(n-2);  
}
```

Ví dụ 3: Tính hệ số nhị thức.

Hệ số nhị thức $C(n,k)$ được định nghĩa đệ quy như sau:

$$C(n,0) = 1, \quad C(n,n) = 1; \quad \text{với mọi } n \geq 0,
C(n,k) = C(n-1,k-1) + C(n-1,k), \quad 0 < k < n.$$

Hàm đệ quy trên C:

```
int C(int n, int k){  
    if ((k==0) || (k==n)) return 1;  
    else return C(n-1,k-1)+C(n-1,k);  
}
```

Ví dụ 4: Tìm kiếm nhị phân.

Bài toán: Cho mảng số $x[1..n]$ được sắp xếp theo thứ tự không giảm và số y .

Cần tìm chỉ số i ($1 \leq i \leq n$) sao cho $x[i] = y$.

Để đơn giản ta giả thiết rằng chỉ số như vậy là tồn tại. Thuật toán để giải bài toán được xây dựng dựa trên lập luận sau:

- Số y cho trước;
- Hoặc bằng phần tử nằm ở vị trí ở giữa mảng x ;
- Hoặc nằm ở nửa bên trái (L) của mảng x ;
- Hoặc nằm ở nửa bên phải (R) của mảng x .

Tình huống L (R) xảy ra chỉ khi y nhỏ hơn (lớn hơn) phần tử ở giữa của mảng x .

Phân tích trên dẫn đến thuật toán sau đây:

```
function Bsearch(x[1..n], start, finish) {  
    middle := (start+finish)/2;  
    if (y == x[middle]) return middle;  
    else {  
        if (y < x[middle]) return Bsearch(x, start,  
middle-1);  
        else /* y > x[middle] */  
            return Bsearch(x, middle+1, finish)  
    }  
}
```

Tổng quát, nếu xét cả tình huống y không có mặt trong mảng:

```

function Bsearch(x[1..n], start, finish) {
    if (start==finish) { /* Base step */
        if (x[middle]==y) return middle;
        else return notFound;
    }
    middle:= (start+finish)/2;
    if (y == x[middle]) return middle;
    else {
        if (y < x[middle]) return Bsearch(x, start,
middle-1);
        else /* y > x[middle] */
            return Bsearch(x, middle+1, finish)
    }
}

```

Cài đặt trên C

```

boolean binary_search(int* a, int n, int x) {
    /* Test xem x có mặt trong mảng a[] kích thước n. */
    int i;
    if (n > 0) {
        i = n / 2;
        if (a[i] == x)
            return true;
        if (a[i] < x)
            return binary_search(&a[i + 1], n - i - 1, x);
        return binary_search(a, i, x);
    }
    return false;
}

```

Ví dụ 5: Bài toán tháp Hà Nội.

Trò chơi tháp Hà Nội được trình bày như sau: “Có ba cọc a, b, c. Trên cọc a có một chồng gồm n cái đĩa đường kính giảm dần từ dưới lên trên. Cần phải chuyển chồng đĩa từ cọc a sang cọc c tuân thủ quy tắc: mỗi lần chỉ chuyển một đĩa và chỉ được xếp đĩa có đường kính nhỏ hơn lên trên đĩa có đường kính lớn hơn. Trong quá trình chuyển được phép dùng cọc b làm cọc trung gian”.

Bài toán đặt ra là: Tìm cách chơi đòi hỏi số lần di chuyển đĩa ít nhất.

Lập luận sau đây được sử dụng để xây dựng thuật toán giải quyết bài toán đặt ra.

– Nếu $n = 1$ thì ta chỉ việc chuyển đĩa từ cọc a sang cọc c.

– Giả sử $n \geq 2$. Việc di chuyển đĩa gồm các bước:

(i) Chuyển $n - 1$ đĩa từ cọc a đến cọc b sử dụng cọc c làm trung gian. Bước này phải được thực hiện với số lần di chuyển đĩa nhỏ nhất, nghĩa là ta phải giải bài toán tháp Hà Nội với $n - 1$ đĩa.

(ii) Chuyển 1 đĩa (đĩa với đường kính lớn nhất) từ cọc a đến cọc c.

(iii) Chuyển $n - 1$ đĩa từ cọc b đến cọc c (sử dụng cọc a làm trung gian). Bước này cũng phải được thực hiện với số lần di chuyển đĩa nhỏ nhất, nghĩa là ta phải giải bài toán tháp Hà Nội với $n - 1$ đĩa.

Bước (i) và (iii) đòi hỏi giải bài toán tháp Hà Nội với $n - 1$ đĩa, vì vậy số lần di chuyển đĩa ít nhất cần thực hiện trong hai bước này là $2h_{n-1}$. Do đó, nếu gọi h_n là số lần di chuyển đĩa ít nhất, ta có công thức đệ quy sau:

$$h_1 = 1,$$

$$h_n = 2h_{n-1} + 1, n \geq 2.$$

Ta có thể tìm được công thức trực tiếp cho h_n bằng phương pháp thế:

$$\begin{aligned} h_n &= 2h_{n-1} + 1 \\ &= 2(2h_{n-2} + 1) + 1 &= 2^2 h_{n-2} + 2 + 1 \\ &= 2^2(2h_{n-3} + 1) + 2 + 1 &= 2^3 h_{n-3} + 2^2 + 2 + 1 \\ &\dots \\ &= 2^{n-1} h_1 + 2^{n-2} + \dots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + \dots + 2 + 1 \quad (\text{do } h_1 = 1) \\ &= 2^n - 1 \end{aligned}$$

Thuật toán có thể mô tả trong thủ tục đệ quy sau đây:

procedure HanoiTower(n , a, b, c);

begin

if $n=1$ **then** **<**chuyển đĩa từ cọc a sang cọc c**>**

else

```

HanoiTower(n-1,a,c,b);
HanoiTower(1,a,b,c);
HanoiTower(n-1,b,a,c);

end;
Cài đặt trên C
#include <stdio.h>
#include <conio.h>
void move (int, char, char, char);
int i = 0;
int main()
{
    int disk;
    printf ("Please input number of disk: ");
    scanf ("%d", &disk);
    move (disk, '1', '3', '2');
    printf ("Total number of moves = %d", i);
    getch();
    return 0;
}
void move (int n, char start, char finish, char spare)
{
    if (n == 1){
        printf ("Move disk from %c to %c\n", start,
finish);
        i++;
        return;
    } else {
        move (n-1, start, spare, finish);
        move (1, start, finish, spare);
        move (n-1, spare, finish, start);
    }
}

```

2.4. PHÂN TÍCH THUẬT TOÁN ĐỆ QUY

Để phân tích thuật toán đệ quy ta thường tiến hành như sau:

- Gọi $T(n)$ là thời gian tính của thuật toán.
- Xây dựng công thức đệ quy cho $T(n)$.
- Giải công thức đệ quy thu được để đưa ra đánh giá cho $T(n)$.

Nói chung ta chỉ cần một đánh giá sát cho tốc độ tăng của $T(n)$ nên việc giải công thức đệ quy đối với $T(n)$ được hiểu là việc đưa ra đánh giá tốc độ tăng của $T(n)$ trong ký hiệu tiệm cận.

Ví dụ 1: Thuật toán FibRec.

```
int FibRec(int n){  
    if (n<=1) return n;  
    else  
        return FibRec(n-1)+ FibRec(n-2);  
}
```

Gọi $T(n)$ là số phép toán cộng phải thực hiện trong lệnh gọi FibRec(n). Ta có:

$$T(0) = 0, T(1) = 0;$$

$$T(n) = T(n - 1) + T(n - 2) + 1, n > 1.$$

Từ đó có thể chứng minh bằng quy nạp: $T(n) = \Theta(F_n)$.

Phân tích trên cho thấy FibRec có thời gian tính tăng với tốc độ cỡ $(1.6)^n$. Vì thế việc sử dụng FibRec là không hiệu quả.

Để tính số Fibonacci thứ n , ta nên dùng thủ tục lặp FibIter sau đây:

```
int FibIter(int n){  
    int f1, f2, f3, k;  
    if (n<=1) return n;  
    else { f1=0; f2=1;  
        for (k=2;k<=n;k++) {  
            f3=f1+f2; f1=f2; f2=f3;}  
        return f3;}  
}
```

Chú ý: Việc thay thế hàm đệ quy bởi hàm không đệ quy thường được gọi là việc *khử đệ quy*. Khử đệ quy không phải bao giờ cũng dễ thực hiện như trong tình huống bài toán tính số Fibonacci.

Phân tích thời gian của thuật toán chia để trị

Xét thuật toán chia để trị được mô tả trong sơ đồ sau đây:

procedure D-and-C (n);

begin

if $n \leq n_0$ **then**

Giải bài toán một cách trực tiếp

else

begin

Chia bài toán thành a bài toán con kích thước n/b ;

for (mỗi bài toán trong b bài toán con) **do** D-and-C(n/b);

Tổng hợp lời giải của b bài toán con để thu được lời giải của bài toán gốc;

end;

end;

Gọi $T(n)$ – thời gian giải bài toán kích thước n . Thời gian của chia để trị được đánh giá dựa trên đánh giá thời gian thực hiện ba thao tác của thuật toán:

– **Chia** bài toán ra thành a bài toán con, mỗi bài toán kích thước n/b : đòi hỏi thời gian: $D(n)$.

– **Trị** (giải) các bài toán con: $aT(n/b)$.

– **Tổ hợp** các lời giải: $C(n)$.

Vậy ta có công thức đệ quy sau đây để tính $T(n)$:

$$T(n) = \begin{cases} \Theta(1), & n \leq n_0 \\ aT(n/b) + D(n) + C(n), & n > n_0 \end{cases}$$

Giải công thức đệ quy: Định lý rút gọn

Ta cần đưa ra đánh giá dưới dạng hiện cho $T(n)$.

Định lý rút gọn cung cấp công cụ để đánh giá số hạng tổng quát của dãy số thỏa mãn công thức đệ quy dạng:

$$T(n) = aT(n/b) + cn^k$$

trong đó: $a \geq 1$ và $b \geq 1$, $c > 0$ là các hằng số;

$T(n)$ được xác định với đối số nguyên không âm;

Ta dùng n/b thay cho cả $\lfloor n/b \rfloor$ lẫn $\lceil n/b \rceil$.

Định lý rút gọn: Giả sử $a \geq 1$ và $b > 1$, $c > 0$ là các hằng số. Xét $T(n)$ là công thức đệ quy:

$$T(n) = aT(n/b) + cn^k$$

xác định với $n \geq 0$.

- Nếu $a > b^k$, thì $T(n) = \Theta(n^{\log_b a})$.
- Nếu $a = b^k$, thì $T(n) = \Theta(n^k \lg n)$.
- Nếu $a < b^k$, thì $T(n) = \Theta(n^k)$.

Ví dụ 1: $T(n) = 3T(n/4) + cn^2$

Trong ví dụ này: $a = 3$, $b = 4$, $k = 2$.

Do $3 < 4^2$, ta có tính huống 3, nên $T(n) = \Theta(n^2)$.

Ví dụ 2: $T(n) = 2T(n/2) + n^{0.5}$

Trong ví dụ này: $a = 2$, $b = 2$, $k = 1/2$.

Do $2 > 2^{1/2}$ nên ta có tính huống 1. Vậy $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$.

Ví dụ 3: $T(n) = 16T(n/4) + n$

$a = 16$, $b = 4$, $k = 1$.

Ta có $16 > 4$, vì thế có tính huống 1. Vậy $T(n) = \Theta(n^2)$.

Ví dụ 4: $T(n) = T(3n/7) + 1$

$a = 1$, $b = 7/3$, $k = 0$.

Ta có $a = b^k$, suy ra có tính huống 2. Vậy $T(n) = \Theta(n^k \log n) = \Theta(\log n)$.

Thời gian tính của sắp xếp trộn

Chia:

– Tính q như là giá trị trung bình của p và r : $D(n) = \Theta(1)$.

Trộn:

– Giải đệ quy hai bài toán con, mỗi bài toán kích thước $n/2$: $2T(n/2)$.

Tổ hợp:

– Trộn hai mảng con với $n/2$ phần tử đòi hỏi thời gian $\Theta(n)$: $C(n) = \Theta(n)$.

Do đó ta có:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T(n/2) + \Theta(n), & n > 1 \end{cases}$$

Suy ra theo định lý thố: $T(n) = \Theta(n \log n)$.

Phân tích Bsearch

Gọi $T(n)$ là thời gian tính của việc thực hiện Bsearch($x[1..n], 1, n$), ta có:

$$T(1) = c$$

$$T(n) = T(n/2) + d$$

trong đó: c và d là các hằng số.

Từ đó theo định lý thố, $T(n) = \Theta(\log n)$.

Phân tích thuật toán tính C(n,k)

Xét thuật toán đệ quy để tính hệ số nhị thức $C(n,k)$:

```

int C(int n, int k){
    if ((k==0) || (k==n)) return 1;
    else return C(n-1,k-1)+C(n-1,k);
}

```

Gọi $C^*(n,k)$ là thời gian thực hiện lệnh gọi hàm $C(n,k)$. Khi đó, dễ thấy $C^*(n,k)$ thỏa mãn công thức đệ quy sau đây:

$$C^*(n,0) \geq 1, \quad C^*(n,n) \geq 1; \quad \text{với mọi } n \geq 0,$$

$$C^*(n,k) \geq C^*(n-1,k-1) + C^*(n-1,k) + 1, \quad 0 < k < n.$$

Từ đó, có thể chứng minh $C^*(n,k) \geq C_n^k$. Do đó thuật toán đệ quy nói trên để tính hệ số nhị thức là không hiệu quả.

2.5. ĐỆ QUY CÓ NHỚ

Trong phần trước ta đã thấy các thuật toán đệ quy để tính số Fibonacci và hệ số nhị thức là kém hiệu quả.

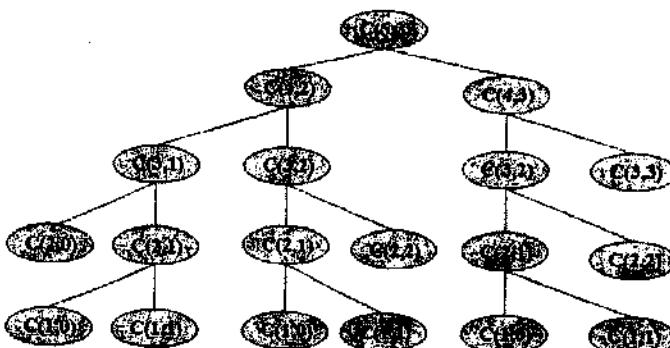
Để tăng hiệu quả của các thuật toán đệ quy mà không cần tiến hành xây dựng các thủ tục lặp hay khử đệ quy, ta có thể sử dụng kỹ thuật đệ quy có nhớ.

Sử dụng kỹ thuật này, trong nhiều trường hợp, ta giữ nguyên được cấu trúc đệ quy của thuật toán và đồng thời đảm bảo được hiệu quả của nó. Nhược điểm lớn nhất của cách làm này là đòi hỏi về bộ nhớ.

Bài toán con trùng lặp

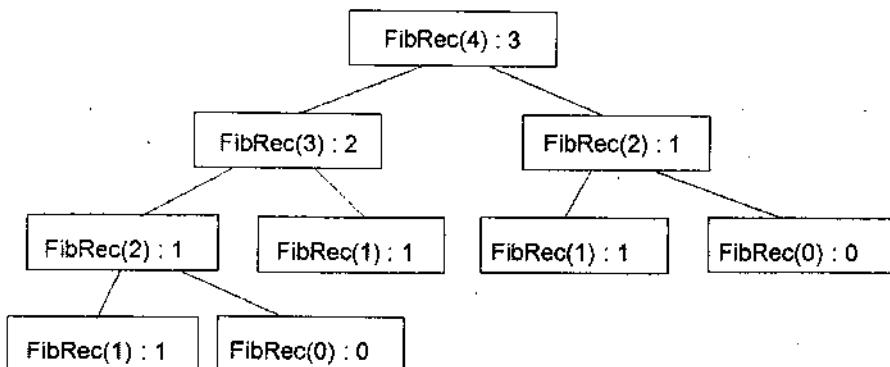
Ta nhận thấy trong các thuật toán đệ quy, mỗi khi cần đến lời giải của một bài toán con lại phải trị nó một cách đệ quy. Do đó, có những bài toán con bị giải đi giải lại nhiều lần, điều đó dẫn đến tính kém hiệu quả của thuật toán. Hiện tượng này gọi là hiện tượng bài toán con trùng lặp.

Ta xét ví dụ thuật toán tính hệ số nhị thức $C(5,3)$. Cây đệ quy thực hiện lệnh gọi hàm $C(5,3)$ được chỉ ra trong hình sau đây:



Hình 1. Bài toán con trùng lặp khi tính $C(5,3)$

Cây đệ quy thực hiện lệnh gọi hàm FibRec(4) được chỉ ra trong hình sau đây:



Hình 2. Bài toán con trùng lặp khi tính FibRec(4)

Để khắc phục hiện tượng này, ý tưởng của đệ quy có nhớ là: dùng biến ghi nhớ lại thông tin về lời giải của các bài toán con ngay sau lần đầu tiên nó được giải. Điều đó cho phép rút ngắn thời gian tính của thuật toán, bởi vì mỗi khi cần đến có thể tra cứu mà không phải giải lại những bài toán con đã được giải trước đó.

Xét ví dụ thuật toán đệ quy tính hệ số nhị thức. Ta đưa vào biến:

- $D[n,k]$ để ghi nhận những giá trị đã tính.
- Đầu tiên $D[n,k] = 0$, mỗi khi tính được $C(n,k)$ giá trị này sẽ được ghi nhận vào $D[n,k]$. Như vậy, nếu $D[n,k] > 0$ thì điều đó có nghĩa là không cần gọi đệ quy hàm $C(n,k)$.

Hàm tính $C(n,k)$ có nhớ:

Function $C(n,k)$ {

```
if  $D[n,k] > 0$  return  $D[n,k]$ 
else{
     $D[n,k] = C(n-1,k-1)+C(n-1,k);$ 
    return  $D[n,k];$ 
}
```

Trước khi gọi hàm $C(n,k)$ cần khởi tạo mảng $D[,]$ như sau:

$D[i,0] = 1$, $D[i,i] = 1$, với $i = 0, 1, \dots, n$.

2.6. CHỨNG MINH TÍNH ĐÚNG ĐÁN CỦA THUẬT TOÁN ĐỆ QUY

Để chứng minh tính đúng đắn của thuật toán đệ quy, thông thường ta sử dụng quy nạp toán học.

Ngược lại chứng minh bằng quy nạp cũng thường là cơ sở để xây dựng nhiều thuật toán đệ quy.

Ta xét một số ví dụ minh họa.

Ví dụ 1: Chứng minh hàm Fact(n) cho ta giá trị của n!

Chứng minh bằng quy nạp toán học.

Cơ sở quy nạp: Ta có $\text{Fact}(0) = 1 = 0!$

Chuyển quy nạp: Giả sử $\text{Fact}(n - 1)$ cho giá trị của $(n - 1)!$, ta chứng minh $\text{Fact}(n)$ cho giá trị của $n!$. Thật vậy, lệnh $\text{Fact}(n)$ sẽ trả lại giá trị:

$$n * \text{Fact}(n - 1) = (\text{theo giả thiết quy nạp}) = n * (n - 1)! = n!$$

Ví dụ 2:

Xét hàm trên Pascal:

```
function Count(x: integer): integer;
begin
    if x=0 then
        begin
            Count:=0; exit
        end
    else Count:= x mod 2 + Count(x div 2)
end;
```

hoặc trên C:

```
int Count(int x) {
    if x==0 return 0;
    else Count = x % 2 + Count(x/2);
```

Hỏi hàm nói trên cho ta đặc trưng gì của số nguyên x? Chứng minh tính đúng đắn của khẳng định đề xuất.

Có thể chứng minh bằng quy nạp khẳng định sau đây: "Hàm Count đếm số bit 1 trong biểu diễn nhị phân của số nguyên đầu vào x".

Ta xét một số ví dụ cho thấy chứng minh bằng quy nạp nhiều khi lại là cơ sở để xây dựng thuật toán đệ quy giải quyết vấn đề đặt ra.

Ví dụ 3:

Trên mặt phẳng vẽ n đường thẳng ở vị trí tổng quát. Hỏi ít nhất phải sử dụng bao nhiêu màu để tô các phần bị chia bởi các đường thẳng này sao cho không có hai phần có chung cạnh nào bị tô bởi cùng một màu?

Ta chứng minh bằng quy nạp mệnh đề sau đây:

$P(n)$: Luôn có thể tô các phần được chia bởi n đường thẳng vẽ ở vị trí tông quát bởi hai màu xanh và đỏ sao cho không có hai phần có chung cạnh nào bị tô bởi cùng một màu.

Cơ sở quy nạp: Khi $n = 1$, mặt phẳng được chia làm hai phần, một phần sẽ tô màu xanh, phần còn lại tô màu đỏ.

Chuyển quy nạp: Giả sử khẳng định đúng với $n - 1$, ta chứng minh khẳng định đúng với n .

Thực vậy, trước hết ta vẽ $n - 1$ đường thẳng. Theo giả thiết, quy nạp có thể tô màu các phần sinh ra bởi hai màu thỏa mãn điều kiện đặt ra. Bây giờ ta vẽ n đường thẳng thứ n . Đường thẳng này chia mặt phẳng ra làm hai phần, gọi là phần A và B. Ta tiến hành tô màu như sau: các phần của mặt phẳng được chia bởi n đường thẳng ở bên nửa mặt phẳng B sẽ giữ nguyên màu đã tô trước đó. Trái lại, các phần trong nửa mặt phẳng A mỗi phần sẽ được tô màu đảo ngược xanh thành đỏ và đỏ thành xanh. Rõ ràng:

– Hai phần có chung cạnh ở cùng một nửa mặt phẳng A hoặc B là không có chung màu.

– Hai phần có chung cạnh trên đường thẳng thứ n rõ ràng cũng không bị tô cùng màu (do màu bên nửa A bị đảo ngược).

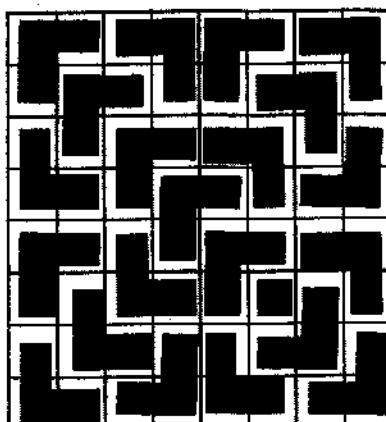
Vậy $P(n)$ đúng. Theo quy nạp khẳng định đúng với mọi n .

Từ chứng minh trên dễ dàng phát triển thuật toán để quy đẽ tô bởi hai màu các phần bị chia bởi n đường thẳng này sao cho không có hai phần có chung cạnh nào bị tô bởi cùng một màu.

Ví dụ 4: Cho lưới ô vuông kích thước $2^n \times 2^n$ bị đục mất một ô tùy ý. Hỏi có thể phủ kín lưới bởi viên gạch chữ L?

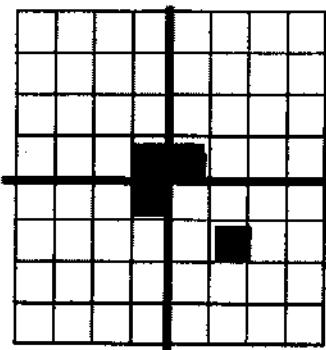
Hình vẽ bên dưới minh họa lưới ô vuông kích thước 8×8 bị đục đi một ô được phủ kín bởi các viên gạch hình chữ L.

Ta chứng minh mệnh đề: “Lưới ô vuông kích thước $2^n \times 2^n$ bị đục mất một ô tùy ý luôn có thể phủ kín bởi các viên gạch hình chữ L”.



Cơ sở quy nạp: Rõ ràng lưới 2×2 có thể phủ được.

Chuyển quy nạp: Giả sử có thể phủ kín lưới $2^n \times 2^n$. Để chỉ ra có thể phủ kín lưới $2^{n+1} \times 2^{n+1}$, ta chia lưới thành bốn lưới con:



Đặt viên gạch L vào vị trí giữa (xem hình vẽ). Bốn lưới con mỗi lưới đều có kích thước $2^n \times 2^n$ và bị khuyết một ô, có thể phủ kín theo giả thiết quy nạp.

Theo quy nạp, khẳng định của mệnh đề được chứng minh.

Dựa trên chứng minh bằng quy nạp vừa nêu, ta có thể phát triển thuật toán để quy đẽ phủ kín lưới bởi các viên gạch hình chữ L.

Ví dụ 5: Kết thúc một giải vô địch bóng chuyền gồm n đội tham gia, trong đó các đội thi đấu vòng tròn một lượt, người ta mời đội trưởng của các đội ra đứng thành một hàng ngang để chụp ảnh.

Ta sẽ chứng minh bằng quy nạp khẳng định sau:

$P(n)$: Luôn có thể xếp n đội trưởng ra thành một hàng ngang sao cho ngoại trừ hai người đứng ở hai mép, mỗi người trong hàng luôn đứng cạnh một đội trưởng của đội thắng đội mình và một đội trưởng của đội thua đội mình trong giải.

Cơ sở quy nạp: Rõ ràng $P(1)$ là đúng.

Chuyển quy nạp: Giả sử $P(n - 1)$ là đúng, ta chứng minh $P(n)$ là đúng.

Trước hết, ta xếp $n - 1$ đội trưởng của các đội $1, 2, \dots, n - 1$. Theo giả thiết quy nạp, luôn có thể xếp họ ra thành hàng ngang thỏa mãn điều kiện đầu bài. Không giảm tông quát ta có thể giả thiết hàng đó là:

$$1 \rightarrow 2 \rightarrow \dots \rightarrow n - 1.$$

Bây giờ ta sẽ tìm chỗ cho đội trưởng của đội n . Có ba tình huống:

– Nếu đội n thắng đội 1 , thì hàng cần tìm là:

$$n \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow n - 1.$$

– Nếu đội n thua đội $n - 1$, thì hàng cần tìm là:

$$1 \rightarrow 2 \rightarrow \dots \rightarrow n - 1 \rightarrow n.$$

– Nếu đội n thua đội 1 và thắng đội $n - 1$, gọi k là chỉ số nhỏ nhất sao cho đội n thắng đội k , rõ ràng tồn tại k như vậy. Khi đó hàng cần tìm sẽ là:

$$1 \rightarrow \dots \rightarrow k-1 \rightarrow n \rightarrow k \rightarrow k+1 \rightarrow \dots \rightarrow n-1.$$

Khẳng định được chứng minh.

2.7. THUẬT TOÁN QUAY LUI

Thuật toán quay lui là một thuật toán cơ bản được áp dụng để giải quyết nhiều vấn đề khác nhau. Ở đây ta mô tả thuật toán quay lui để giải bài toán sau đây:

Bài toán liệt kê (Q): Cho A_1, A_2, \dots, A_n là các tập hữu hạn. Ký hiệu:

$$X = A_1 \times A_2 \times \dots \times A_n = \{ (x_1, x_2, \dots, x_n) : x_i \in A_i, i = 1, 2, \dots, n \}.$$

Giả sử P là tính chất cho trên X . Vấn đề đặt ra là liệt kê tất cả các phần tử của X thỏa mãn tính chất P :

$$D = \{ x = (x_1, x_2, \dots, x_n) \in X : x \text{ thỏa mãn tính chất } P \}.$$

Các phần tử của tập D được γ là các lời giải chấp nhận được.

Ví dụ:

– Bài toán liệt kê xâu nhị phân độ dài n dẫn về việc liệt kê các phần tử của tập:

$$B^n = \{ (x_1, \dots, x_n) : x_i \in \{0, 1\}, i = 1, 2, \dots, n \}.$$

– Bài toán liệt kê các tập con m phần tử của tập $N = \{1, 2, \dots, n\}$ đòi hỏi liệt kê các phần tử của tập:

$$S(m, n) = \{ (x_1, \dots, x_m) \in N^m : 1 \leq x_1 < \dots < x_m \leq n \}.$$

– Tập các hoán vị của các số tự nhiên $1, 2, \dots, n$ là tập:

$$\Pi_n = \{ (x_1, \dots, x_n) \in N^n : x_i \neq x_j ; i \neq j \}.$$

Định nghĩa: Ta gọi lời giải bộ phận cấp k ($0 \leq k \leq n$) là bộ có thứ tự gồm k thành phần (a_1, a_2, \dots, a_k) , trong đó $a_i \in A_i, i = 1, 2, \dots, k$.

Khi $k = 0$, lời giải bộ phận cấp 0 được ký hiệu là \emptyset và còn được gọi là lời giải rỗng.

Nếu $k = n$, ta có lời giải đầy đủ hay đơn giản là một lời giải của bài toán.

Ý tưởng chung

Thuật toán quay lui được xây dựng dựa trên việc xây dựng dần từng thành phần của lời giải.

Thuật toán bắt đầu từ lời giải rỗng \emptyset . Trên cơ sở tính chất P , ta xác định được những phần tử nào của tập A_1 có thể chọn vào vị trí thứ nhất của lời giải. Những phần tử như vậy ta sẽ gọi là những ứng cử viên (viết tắt là UCV) vào vị trí thứ nhất của lời giải. Ký hiệu tập các UCV vào vị trí thứ nhất của lời giải là S_1 . Lấy $a_1 \in S_1$, bổ sung nó vào lời giải rỗng đang có, ta thu được lời giải bộ phận cấp 1: (a_1) .

Tại bước tổng quát, giả sử ta đang có lời giải bộ phận cấp $k-1$: $(a_1, a_2, \dots, a_{k-1})$.

Trên cơ sở tính chất P ta xác định được những phần tử nào của tập A_k có thể chọn vào vị trí thứ k của lời giải. Những phần tử như vậy ta sẽ gọi là những ứng cử viên (viết tắt là UCV) vào vị trí thứ k của lời giải khi $k - 1$ thành phần đầu của nó đã được chọn là $(a_1, a_2, \dots, a_{k-1})$. Ký hiệu tập các ứng cử viên này là S_k .

Xét hai tình huống:

- $S_k \neq \emptyset$: Khi đó lấy $a_k \in S_k$, bổ sung nó vào lời giải bộ phận cấp $k - 1$ đang có $(a_1, a_2, \dots, a_{k-1})$,

ta thu được lời giải bộ phận cấp k :

$$(a_1, a_2, \dots, a_{k-1}, a_k).$$

+ Nếu $k = n$ thì ta thu được một lời giải.

+ Nếu $k < n$, ta tiếp tục đi xây dựng thành phần thứ $k + 1$ của lời giải.

- $S_k = \emptyset$ (tình huống ngõ cụt): Điều đó có nghĩa là lời giải bộ phận $(a_1, a_2, \dots, a_{k-1})$ không thể tiếp tục phát triển thành lời giải đầy đủ. Trong tình huống này ta *quay trở lại* tìm ứng cử viên mới vào vị trí thứ $k - 1$ của lời giải.

+ Nếu tìm thấy UCV như vậy thì bổ sung nó vào vị trí thứ $k - 1$ rồi lại tiếp tục đi xây dựng thành phần thứ k .

+ Nếu không tìm được thì ta lại *quay trở lại* thêm một bước nữa để tìm UCV mới vào vị trí thứ $k - 2$, ... Nếu quay lại tận lời giải rỗng mà vẫn không tìm được UCV mới vào vị trí thứ 1, thì thuật toán kết thúc.

Thuật toán quay lui có thể mô tả trong thủ tục đệ quy sau đây

procedure Backtrack(k: integer);

begin

Xây dựng S_k ;

for $y \in S_k$ do (* Với mỗi UCV y từ S_k *)

begin

$a_k := y$;

if $k = n$ then <Ghi nhận lời giải (a_1, a_2, \dots, a_k) >

else Backtrack(k+1);

end;

end;

Lệnh gọi để thực hiện thuật toán quay lui là: **Backtrack(1)**.

Hai vấn đề mẫu chốt

Để cài đặt thuật toán quay lui giải các bài toán cụ thể, ta cần giải quyết hai vấn đề cơ bản sau:

- Tìm thuật toán xây dựng các tập UCV S_k .

– Tìm cách mô tả các tập này để có thể cài đặt thao tác liệt kê các phần tử của chúng (cài đặt vòng lặp quy ước $\text{for } y \in S_k \text{ do}$).

Hiệu quả của thuật toán liệt kê phụ thuộc vào việc ta có xác định được chính xác các tập UCV này hay không.

Chú ý:

– Nếu chỉ cần tìm một lời giải thì cần tìm cách chấm dứt các thủ tục gọi đệ quy lồng nhau sinh bởi lệnh gọi **Backtrack(1)** sau khi ghi nhận được lời giải đầu tiên.

– Nếu kết thúc thuật toán mà ta không thu được một lời giải nào thì điều đó có nghĩa là bài toán không có lời giải.

– Thuật toán dễ dàng mở rộng cho bài toán liệt kê, trong đó lời giải có thể mô tả như là bộ $(a_1, a_2, \dots, a_n, \dots)$ độ dài hữu hạn, tuy nhiên giá trị của độ dài là không biết trước và các lời giải cũng không nhất thiết phải có cùng độ dài.

Khi đó chỉ cần sửa lại câu lệnh:

```
if k = n then <Ghi nhận lời giải (a1, a2, ..., ak)>
else Backtrack(k+1);
```

thành:

```
if <(a1, a2, ..., ak) là lời giải> then <Ghi nhận (a1, a2, ..., ak)>
else Backtrack(k+1);
```

Ta cần xây dựng hàm nhận biết (a_1, a_2, \dots, a_k) đã là lời giải hay chưa.

Như một ví dụ minh họa thuật toán, ta xét bài toán xếp Hậu sau đây.

Bài toán xếp Hậu

– Liệt kê tất cả các cách xếp n quân Hậu trên bàn cờ $n \times n$ sao cho chúng không ăn được lẫn nhau, nghĩa là sao cho không có hai con nào trong số chúng nằm trên cùng một dòng hay một cột hay một đường chéo của bàn cờ.

Biểu diễn lời giải:

– Đánh số các cột và dòng của bàn cờ từ 1 đến n . Một cách xếp Hậu có thể biểu diễn bởi bộ có n thành phần (a_1, a_2, \dots, a_n) , trong đó a_i là tọa độ cột của con Hậu ở dòng i .

– Các điều kiện đặt ra đối với bộ (a_1, a_2, \dots, a_n) :

+ $a_i \neq a_j$, với mọi $i \neq j$ (nghĩa là hai con Hậu ở hai dòng i và j không được nằm trên cùng một cột);

+ $|a_i - a_j| \neq |i - j|$, với mọi $i \neq j$ (nghĩa là hai con Hậu ở hai ô (a_i, i) và (a_j, j) không được nằm trên cùng một đường chéo).

Như vậy bài toán xếp Hậu dẫn về bài toán liệt kê các phần tử của tập:

$$D = \{(a_1, a_2, \dots, a_n) \in N^n : a_i \neq a_j \text{ và } |a_i - a_j| \neq |i - j|, i \neq j\}.$$

Dễ thấy là ta có:

$$D = \{(a_1, a_2, \dots, a_n) \in \Pi_n : |a_i - a_j| \neq |i - j|, i \neq j\}.$$

Do đó bài toán xếp Hậu dẫn về bài toán liệt kê các hoán vị thỏa mãn tính chất bổ sung:

$$|a_i - a_j| \neq |i - j|, i \neq j.$$

Hàm nhận biết ứng cử viên

```
int UCVh(int j, int k) {
    // UCVh nhận giá trị 1
    // khi và chỉ khi j ∈ Sk
    int i;
    for (i=1; i<k; i++)
        if ((j == a[i]) || (fabs(j-a[i]) == k-i))
            return 0;
    return 1;
}
```

Chương trình trên C:

```
#include <stdio.h>
#include <math.h>

int n, count;
int a[20];
int Ghinhан() {
    int i;
    count++; printf("%i. ",count);
    for (i=1; i<=n; i++) printf("%i ",a[i]);
    printf("\n");
}

int UCVh(int j, int k) {
    /* UCVh nhận giá trị 1 khi và chỉ khi j ∈ Sk */
    int i;
    for (i=1; i<k; i++)
        if ((j == a[i]) || (fabs(j-a[i]) == k-i)) return 0;
```

```

    return 1;
}

int Hau(int i){
    int j;
    for (j=1; j<=n; j++)
        if (UCVh(j, i)){
            a[i] = j;
            if (i == n) Ghinhant();
            else Hau(i+1);
        }
}

int main() {
    printf("Input n = "); scanf("%i", &n);
    printf("\n==== RESULT ====\n");
    count = 0; Hau(1);
    if (count == 0) printf("No solution!\n");
    getchar();
    printf("\n Press Enter to finish... ");
    getchar();
}

```

Chú ý:

– Rõ ràng bài toán xếp Hậu không phải luôn luôn có lời giải, chẳng hạn bài toán không có lời giải khi $n = 2, 3$. Do đó điều này cần được thông báo khi kết thúc thuật toán.

– Thuật toán trình bày ở trên chưa hiệu quả. Nguyên nhân là do không xác định được chính xác các tập UCV vào các vị trí của lời giải.

BÀI TẬP CHƯƠNG 2

1. Xét thuật toán sau đây:

A(n)

```
1. if n < 2
2.     return
3. else
4.     count ← 0
5.     for i ← 1 to 8
6.         A(└n/2┘)
7.     for i ← 1 to n3
8.         count ← count + 1
```

Ký hiệu $T(n)$ là thời gian tính của thuật toán phải thực hiện với đầu vào là số nguyên dương n .

a) Xây dựng công thức đệ quy cho $T(n)$.

b) Đưa ra đánh giá tiệm cận cho $T(n)$ bằng bất cứ phương pháp nào.

2. Đánh giá thời gian tính của các hàm sau đây:

a)

```
int f4(int n) {
    if (n > 1)
        return 1 + 2*f4(n-1);
    return 0;
}
```

b)

```
int f5(int n) {
    if (n > 1)
        return 3*f5(n/2) + 1
    return 0;
}
```

3. Xét thuật toán tính giá trị của $f(x,n) = x^n$ thể hiện trong hàm F(x,n) sau đây:

```
int F(int x, int n)
{
    if (n== 0)
        return 1;
    else if (n%2 == 0)
        return F(x, n/2)* F(x, n/2) ;
    else
        return F(x, n/2)* F(x, n/2)*x ;
}
```

Gọi $T(n)$ là thời gian tính của thuật toán nói trên. Giả thiết các phép toán số học được thực hiện với thời gian bị chặn bởi hằng số.

- a) Xây dựng công thức đệ quy cho $T(n)$.
 - b) Giải công thức đệ quy để đưa ra đánh giá của $T(n)$ trong ký hiệu O.
4. Xét hàm đệ quy sau đây:

```
int myst(int x,int n){
    int y;
    if (n==0) return 1;
    if (n%2 == 1)
    {   y= myst(x,(n-1)/2);
        return x*y*y;
    }
    else
    {   y= myst(x,n/2);
        return y*y;
    }
}
```

- a) Với giả thiết x là số nguyên, n là số nguyên không âm, hàm trên thực hiện công việc gì? Chứng minh khẳng định đưa ra.
- b) Với giả thiết các phép toán cần thực hiện đều có thể thực hiện trong thời gian $O(1)$ và $n = 2^k$, hãy đưa ra đánh giá thời gian tính của thuật toán trong ký hiệu O .

5. Hai thuật toán sau đây có thể sử dụng để tính 2^n , với n là số nguyên dương:

Thuật toán pow2A

```
int pow2A(int n){  
    if (n==0)  
        return 1;  
    else  
        return 2*pow2A(n-1);  
}
```

Thuật toán pow2B

```
int pow2B(int n){  
    if (n==0)  
        return 1;  
    else  
        return pow2B(n-1)+pow2B(n-1);  
}
```

a) Đánh giá thời gian tính của mỗi thuật toán (giả thiết tất cả các phép toán đều được thực hiện trong thời gian $O(1)$).

b) Thuật toán nào hiệu quả hơn?

6. Xét hàm đệ quy sau đây:

```
int myst(int x,int n){  
    int y;  
    if (n==0) return 1;  
    if (n%2 == 1)  
    {  
        y= myst(x, (n-1)/2);  
        return x*y*y;  
    }  
    else  
    {  
        y= myst(x, n/2);  
        return y*y;  
    }  
}
```

- a) Với giả thiết x là số nguyên, n là số nguyên không âm, hàm trên thực hiện công việc gì? Chứng minh khẳng định đưa ra.
- b) Đánh giá thời gian tính của hàm đã cho.
7. Giải công thức đệ quy:
- a) $t_0 = 0, t_1 = 1,$

$$t_n = 5 t_{n-1} - 6 t_{n-2}, n \geq 2.$$
- b) $T(0) = a, T(1) = a,$

$$T(n) = T(n-1) + T(n-2) + c, n \geq 2.$$

 trong đó: a, c là các hằng số dương.
- c) $T(0) = a, T(1) = a,$

$$T(n) = T(n-1) + T(n-2) + cn, n \geq 2.$$

 trong đó: a, c là các hằng số dương.
8. Xét hàm đệ quy trên C để tính giá trị của $f(x, n) = x^n$ thể hiện trong hàm F(x,n) sau đây:
- ```
int F(int x, int n) {
 int tmp;
 if (n == 0) return 1;
 else if (n%2 == 0) {
 tmp = F(x, n/2);
 return tmp * tmp;
 } else {
 tmp = F(x, n/2);
 return tmp * tmp * x;
 }
}
```
- Sử dụng cấu trúc ngăn xếp, hãy viết lại hàm trên thành hàm không đệ quy Fiter(x,n).
9. Bài toán liệt kê các mảng thứ tự.
- Một hoán vị  $(p_1, p_2, \dots, p_n)$  của  $n$  số tự nhiên  $1, 2, \dots, n$  được gọi là một *mảng thứ tự* nếu  $p_i \neq i$  với mọi  $i = 1, 2, \dots, n$ . Hãy viết chương trình trên C liệt kê tất cả các mảng thứ tự của các số  $1, 2, \dots, n$ .
10. Xét bài toán sau đây: "Cho hai tập  $X$  và  $Y$ , mỗi tập gồm  $n$  số nguyên và số nguyên  $k$ . Hỏi có thể tìm được một số  $x$  thuộc tập  $X$  và một số  $y$  thuộc tập  $Y$  sao cho  $x + y = k$  hay không?"

- a) Phát triển thuật toán hiệu quả để giải bài toán đặt ra (mô tả thuật toán trong giả ngôn ngữ).
- b) Chứng minh tính đúng đắn và đánh giá thời gian tính của thuật toán.
11. Cho dãy số nguyên  $a_1, a_2, \dots, a_n$ . Cần tìm hai chỉ số  $i, j$  thỏa mãn  $1 \leq i < j \leq n$  sao cho  $a_j - a_i$  là lớn nhất trong tất cả các cách chọn chỉ số như vậy. Nói cách khác, ta cần giải bài toán tối ưu sau đây:
- Tìm giá trị lớn nhất của hiệu  $a_j - a_i$ , với điều kiện  $1 \leq i < j \leq n$ .
- Dữ liệu vào của bài toán được cho trong file văn bản với tên SEQUENCE.INP có cấu trúc như sau:
- Dòng đầu tiên chứa số nguyên  $n$  ( $1 \leq n \leq 200000$ ) là số phần tử trong dãy số đã cho.
  - Dòng thứ  $i$  trong số  $n$  dòng tiếp theo chứa số nguyên  $a_i$ ,  $i = 1, 2, \dots, n$ .
- Hãy viết chương trình nhập dữ liệu vào từ file văn bản SEQUENCE.INP và đưa ra file văn bản SEQUENCE.OUT giá trị lớn nhất tìm được.

# Chương 3

## CÁC CẤU TRÚC DỮ LIỆU CƠ BẢN

### 3.1. CÁC KHÁI NIỆM

#### 3.1.1. Kiểu dữ liệu

Kiểu dữ liệu (data type) được đặc trưng bởi:

- Tập các giá trị;
- Cách biểu diễn dữ liệu được sử dụng chung cho tất cả các giá trị này;
- Tập các phép toán có thể thực hiện trên tất cả các giá trị.

#### Các kiểu dữ liệu dựng sẵn

Trong các ngôn ngữ lập trình thường có một số kiểu dữ liệu nguyên thủy đã được xây dựng sẵn. Ví dụ:

- Kiểu số nguyên: chẵng hạn, byte, char, short, int, long.
- Kiểu số thực dấu phẩy động: chẵng hạn, float, double.
- Các kiểu nguyên thủy khác: chẵng hạn, boolean.
- Kiểu mảng: chẵng hạn, mảng các phần tử cùng kiểu.

Bảng sau đây liệt kê một số kiểu dữ liệu số trong ngôn ngữ lập trình C:

| Type          | Bits | Minimum value          | Maximum value         |
|---------------|------|------------------------|-----------------------|
| <i>byte</i>   | 8    | -128                   | 127                   |
| <i>short</i>  | 16   | -32768                 | 32767                 |
| <i>char</i>   | 16   | 0                      | 65535                 |
| <i>int</i>    | 32   | $-2^{31}$              | $2^{31}-1$            |
| <i>long</i>   | 64   | $-9223372036854775808$ | $9223372036854775807$ |
| <i>float</i>  | 32   |                        |                       |
| <i>double</i> | 64   |                        |                       |

#### Phép toán đối với kiểu dữ liệu nguyên thủy

- Đối với kiểu: byte, char, short, int, long

+, -, \*, /, %, đổi thành xâu,...

- Đổi với kiểu: float, double  
+,-, \*, /, round, ceil, floor, ...
- Đổi với kiểu: boolean  
Kiểm giá trị true, hay kiểm giá trị false.

Nhận thấy rằng, các ngôn ngữ lập trình khác nhau có thể sử dụng để mô tả kiểu dữ liệu khác nhau. Chẳng hạn, PASCAL và C có những mô tả các dữ liệu số khác nhau.

### 3.4.2. Kiểu dữ liệu trừu tượng

Kiểu dữ liệu trừu tượng (sẽ viết tắt là ADT) bao gồm:

- Tập các giá trị;
- Tập các phép toán có thể thực hiện với tất cả các giá trị này.

**Dễ dàng nhận thấy rằng:** một thành phần trong định nghĩa của kiểu dữ liệu đã bị bỏ qua để thu được định nghĩa ADT, đó là cách biểu diễn dữ liệu (*data representation*) được sử dụng chung cho tất cả các giá trị này. Việc làm này có ý nghĩa làm trừu tượng hóa khái niệm kiểu dữ liệu. ADT không còn phụ thuộc vào cài đặt, không phụ thuộc ngôn ngữ lập trình.

**Ví dụ:**

| ADT              | Đổi tượng                                  | Phép toán               |
|------------------|--------------------------------------------|-------------------------|
| Danh sách (List) | Các nút                                    | Chèn, xóa, tìm,...      |
| Đồ thị (Graphs)  | Đỉnh, cạnh                                 | Duyệt, đường đi, ...    |
| Integer          | $-\infty, \dots, -1, 0, 1, \dots, +\infty$ | $+, -, *, \dots$        |
| Real             | $-\infty, \dots, +\infty$                  | $+, -, *, \dots$        |
| Ngăn xếp         | Các phần tử                                | Pop, push, is Empty,... |
| Hàng đợi         | Các phần tử                                | Enqueue, dequeue,...    |
| Cây nhị phân     | Các nút                                    | Traversal, find,...     |

Điều dễ hiểu là các kiểu dữ liệu nguyên thủy mà các ngôn ngữ lập trình cài đặt sẵn cũng được coi là thuộc vào kiểu dữ liệu trừu tượng. Trên thực tế chúng là cài đặt của kiểu dữ liệu trừu tượng trên ngôn ngữ lập trình cụ thể.

**Định nghĩa:** Ta gọi việc *cài đặt* một ADT là việc diễn tả bởi các câu lệnh của một ngôn ngữ lập trình để mô tả các biến trong ADT và các thủ tục trong ngôn ngữ lập trình để thực hiện các phép toán của ADT, hoặc trong các ngôn ngữ hướng đối tượng, là các lớp bao gồm cả dữ liệu và các phương thức xử lý.

### 3.1.3. Cấu trúc dữ liệu

Có thể nói những thuật ngữ: kiểu dữ liệu, kiểu dữ liệu trừu tượng và cấu trúc dữ liệu (trong tiếng Anh tương ứng là Data Types, Abstract Data Types, Data Structures) nghe rất giống nhau, nhưng thực ra chúng có ý nghĩa khác nhau.

Trong ngôn ngữ lập trình, *kiểu dữ liệu* của biến là tập các giá trị mà biến này có thể nhận. Ví dụ, biến kiểu boolean chỉ có thể nhận giá trị đúng hoặc sai. Các kiểu dữ liệu cơ bản có thể thay đổi từ ngôn ngữ lập trình này sang ngôn ngữ lập trình khác. Ta có thể tạo những kiểu dữ liệu phức hợp từ những kiểu dữ liệu cơ bản. Cách tạo cũng phụ thuộc vào ngôn ngữ lập trình.

*Kiểu dữ liệu trừu tượng* là mô hình toán học cùng với những phép toán xác định trên mô hình này. Nó không phụ thuộc vào ngôn ngữ lập trình.

Để biểu diễn mô hình toán học trong ADT, ta sử dụng *cấu trúc dữ liệu*.

Cấu trúc dữ liệu (Data Structures) là một họ các biến, có thể có kiểu dữ liệu khác nhau, được liên kết lại theo một cách thức nào đó.

Việc cài đặt ADT đòi hỏi lựa chọn cấu trúc dữ liệu để biểu diễn ADT. Ta sẽ xét xem việc làm đó được tiến hành như thế nào?

Ở là đơn vị cơ sở cấu thành cấu trúc dữ liệu. Có thể hình dung ô như một hộp đựng giá trị phát sinh từ một kiểu dữ liệu cơ bản hay phức hợp.

Cấu trúc dữ liệu được tạo nhờ đặt tên cho một *nhóm* các ô và đặt giá trị cho một số ô để mô tả sự liên kết giữa các ô. Ta xét một số cách tạo nhóm.

Một trong những cách tạo nhóm đơn giản nhất trong các ngôn ngữ lập trình là *mảng*. Mảng là một dãy các ô có cùng kiểu xác định nào đó.

**Ví dụ:** Khai báo trong C sau đây:

```
int name[10];
```

khai báo biến *name* gồm 10 phần tử kiểu cơ sở nguyên (integer).

Có thể truy xuất đến phần tử của mảng nhờ chỉ ra tên mảng cùng với chỉ số của nó. Ta sẽ xét kỹ hơn kiểu mảng trong mục tiếp theo.

Một phương pháp chung nữa hay dùng để nhóm các ô là *cấu trúc bản ghi* (*record structure*).

*Bản ghi* là ô được tạo bởi một họ các ô (gọi là các trường) có thể có kiểu rất khác nhau. Các bản ghi lại thường được nhóm lại thành mảng; kiểu được xác định bởi việc nhóm các trường của bản ghi trở thành kiểu của phần tử của mảng.

**Ví dụ:** Trong C mô tả:

```
struct record {
 float data;
 int next; } reclist[100];
```

khai báo reclist là mảng 10 phần tử, mỗi ô là một bàn ghi gồm hai trường: *data* và *next*.

Phương pháp thứ ba để nhóm các ô là *file*. File, cũng giống như mảng một chiều, là một dãy các giá trị cùng kiểu nào đó. Tuy nhiên, các phần tử trong file chỉ có thể truy xuất được một cách tuần tự, theo thứ tự mà chúng xuất hiện trong file. Trái lại, mảng và bàn ghi là các cấu trúc trực truy ("random-access"), nghĩa là thời gian để truy xuất đến các thành phần của mảng (hay bàn ghi) không phụ thuộc vào chỉ số mảng (hay trường được lựa chọn). Bên cạnh đó cần nhấn mạnh một ưu điểm của kiểu file là số phần tử của nó không bị giới hạn!

Khi lựa chọn cấu trúc dữ liệu cài đặt ADT, một vấn đề cần được quan tâm là thời gian thực hiện các phép toán đối với ADT sẽ như thế nào. Bởi vì, các cách cài đặt khác nhau có thể dẫn đến thời gian thực hiện phép toán khác nhau.

**Ví dụ:** Xét cài đặt ADT từ điển.

Kiểu dữ liệu trùu tượng từ điển bao gồm:

- Cần lưu trữ tập các cặp  $\langle \text{key}, \text{value} \rangle$  để tìm kiếm value theo key. Mỗi key có không quá một value;
- Các phép toán;
- $\text{insert}(k, v)$ : chèn cặp  $(k, v)$  vào từ điển;
- $\text{find}(k)$ : nếu  $(k, v)$  có trong từ điển thì trả lại  $v$ , ngược lại trả về 0;
- $\text{remove}(k)$ : xóa bỏ cặp  $(k, v)$  trong từ điển.

Xét ba phương pháp cài đặt từ điển:

- Danh sách móc nối (Linked list);
- Mảng sắp xếp (Sorted array);
- Cây tìm kiếm (Search tree).

Bảng dưới đây cho biết đánh giá thời gian của việc thực hiện các phép toán:

| Cài đặt      | Bổ sung (Insert) | Tìm (Find)  | Loại bỏ (Remove) |
|--------------|------------------|-------------|------------------|
| Linked List  | $O(n)$           | $O(n)$      | $O(n)$           |
| Sorted Array | $O(n)$           | $O(\log n)$ | $O(n)$           |
| Search Tree  | $O(\log n)$      | $O(\log n)$ | $O(\log n)$      |

### Con trỏ

Một trong những ưu thế của phương pháp nhóm các ô trong các ngôn ngữ lập trình là ta có thể biểu diễn mối quan hệ giữa các ô nhờ sử dụng con trỏ.

**Định nghĩa:** *Con trỏ* là ô mà giá trị của nó chỉ ra một ô khác.

Khi vẽ các cấu trúc dữ liệu, để thể hiện ô A là con trỏ đến ô B, ta sẽ sử dụng mũi tên hướng từ A đến B.



**Ví dụ:** Để tạo biến con trỏ *ptr* để trỏ đến ô có kiểu cho trước, chẳng hạn *celltype*, ta có thể khai báo:

**celltype \*ptr**

#### Phân loại các cấu trúc dữ liệu

Trong nhiều tài liệu về cấu trúc dữ liệu thường sử dụng phân loại cấu trúc dữ liệu sau đây:

– Cấu trúc dữ liệu cơ sở. Ví dụ, trong Pascal: integer, char, real, boolean, ...; trong C: int, char, float, double,...

– Cấu trúc dữ liệu tuyến tính. Ví dụ, Mảng (Array), Danh sách liên kết (Linked list), Ngăn xếp (Stack), Hàng đợi (Queue),...

– Cấu trúc dữ liệu phi tuyến. Ví dụ, Cây (Trees), Đồ thị (Graphs), Bảng băm (Hash tables),...

## 3.2. MẢNG

### 3.2.1. Kiểu dữ liệu trừu tượng mảng

Mảng được định nghĩa như kiểu dữ liệu trừu tượng như sau:

**Đối tượng:** Tập các cặp  $\langle \text{index}, \text{value} \rangle$  trong đó với mỗi giá trị của *index* có một giá trị từ tập *item*. *Index* là tập có thứ tự một chiều hay nhiều chiều, chẳng hạn,  $\{0, \dots, n - 1\}$  đối với một chiều,  $\{(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)\}$  đối với hai chiều,...

#### Các thao tác cơ bản

– Khởi tạo mảng:

Array Create(*j*, *list*) ::= trả lại mảng *j* chiều, trong đó *list* là bộ *j* thành phần với thành phần thứ *i* là kích thước của chiều thứ *i*. Item không được định nghĩa.

– Truy xuất phần tử: *Item Retrieve(A, i)*

**if** (*i* in *index*) **return** item tương ứng với giá trị chỉ số *i* trong mảng *A*.

**else return error**

– Cắt giữ phần tử: *Array Store(A, i, x)*

**if** (*i* in *index*) **return** mảng giống hệt mảng *A* ngoại trừ cặp mới  $\langle i, x \rangle$  được chèn vào.

**else return error**

### 3.2.2. Phân bổ bộ nhớ cho mảng

Ta xét việc cài đặt mảng trong các ngôn ngữ lập trình, ta hạn chế việc xét mảng một chiều và hai chiều. **Mảng** là dãy các thành phần được đánh chỉ số. Thông thường mảng chiếm giữ một dãy từ máy liên tiếp trong bộ nhớ (cách lưu trữ này được gọi là *lưu trữ kế tiếp*). Độ dài của mảng được xác định khi khởi tạo và không thể thay đổi. Mỗi thành phần của mảng có một chỉ số cố định duy nhất. Chỉ số nhận giá trị trong khoảng từ một cận dưới đến một cận trên nào đó. Mỗi thành phần của mảng được truy xuất nhờ sử dụng chỉ số của nó. Phép toán này được thực hiện một cách hiệu quả với thời gian  $\Theta(1)$ .

#### Mảng trong các ngôn ngữ lập trình

- Các chỉ số có thể là số nguyên (C, Java) hoặc là các giá trị kiểu rời rạc (Pascal, Ada).
- Cận dưới là 0 (C, Java), 1 (Fortran), hoặc tùy chọn bởi người lập trình (Pascal, Ada).

- Trong hầu hết các ngôn ngữ, mảng là thuần nhất (nghĩa là tất cả các phần tử của mảng có cùng một kiểu); trong một số ngôn ngữ (như Lisp, Prolog), các thành phần có thể là không thuần nhất (có các kiểu khác nhau).

- Trong các ngôn ngữ lập trình hướng đối tượng (object-oriented language), mảng có thể là đối tượng (objects trong Java) hoặc không phải là đối tượng (C++).

- Mảng có thể mang thêm những thông tin bổ sung về chính nó, như kiểu và độ dài (Java), hoặc có thể bao gồm chỉ các thành phần của nó (C, C++).

**Chú ý:** Có ngôn ngữ (như PASCAL) thực hiện kiểm tra lỗi vượt mảng (truy xuất đến thành phần với chỉ số không trong phạm vi từ cận dưới đến cận trên) và chấm dứt thực hiện chương trình nếu xảy ra lỗi này. Tuy nhiên, nhiều ngôn ngữ khác (C, C++, JAVA) lại không thực hiện điều này. Trong trường hợp xảy ra lỗi vượt mảng, chương trình có thể tiếp tục chạy, cũng có thể bị dừng, tùy thuộc vào thao tác truy xuất trong tình huống cụ thể. Nếu chương trình không dừng thì sẽ rất khó phát hiện lỗi!

#### Khai báo mảng một chiều trong C

Khai báo mảng một chiều trong C có dạng:

**<kiểu thành phần> <tên biến>[size]**

**Ví dụ:** Xét khai báo:

**int list[5];**

Khai báo trên sẽ khai báo biến mảng tên list với 5 phần tử có kiểu là số nguyên (2 byte).

Địa chỉ của các phần tử trong mảng một chiều:

list[0]                  địa chỉ gốc =  $\alpha$

list[1]                   $\alpha + \text{sizeof}(\text{int})$

list[2]                   $\alpha + 2 * \text{sizeof}(\text{int})$

list[3]             $\alpha + 3 * \text{sizeof(int)}$

list[4]             $\alpha + 4 * \text{size(int)}$

Chương trình trên C sau đây đưa ra địa chỉ của các phần tử của mảng một chiều trên C:

```
#include <stdio.h>
#include <conio.h>
int main()
{
 int one[] = {0, 1, 2, 3, 4};
 int *ptr; int rows=5;
/* in địa chỉ của mảng một chiều nhờ dùng con trỏ */
 int i; ptr= one;
 printf("Address Contents\n");
 for (i=0; i < rows; i++)
 printf("%8u%5d\n", ptr+i, *(ptr+i));
 printf("\n");
 getch();
}
```

Kết quả chạy trong DEVC  
(`sizeof(int)=4`)

The screenshot shows a memory dump titled "Address Contents". The data is as follows:

| Address | Contents |
|---------|----------|
| 2293584 | 0        |
| 2293588 | 1        |
| 2293592 | 2        |
| 2293596 | 3        |
| 2293600 | 4        |

Kết quả chạy trong TC:  
(`sizeof(int)=2`)

The screenshot shows a memory dump titled "Turbo C++ IDE". The data is as follows:

| Address | Contents |
|---------|----------|
| 000516  | 3        |
| 000518  | 1        |
| 000520  | 2        |
| 000522  | 3        |
| 000524  | 1        |

### Mảng hai chiều

Khai báo mảng hai chiều trong C:

`<element-type> <arrayName> [size 1][size2];`

Ví dụ:

```
double table[5] [4];
```

Truy xuất đến phần tử của mảng: `table[2] [4];`

Xét khai báo:

```
int a [4] [3];
```

Thông thường có thể coi nó như một bảng.

|        |        |        |        |
|--------|--------|--------|--------|
| dòng 0 | a[0,0] | a[0,1] | a[0,2] |
| dòng 1 | a[1,0] | a[1,1] | a[1,2] |
| dòng 2 | a[2,0] | a[2,1] | a[2,2] |
| dòng 3 | a[3,0] | a[3,1] | a[3,2] |

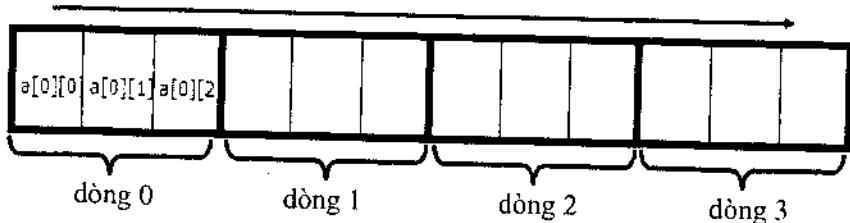
cột 0      cột 1      cột 2

### Phân bổ bộ nhớ cho mảng hai chiều

Trong bộ nhớ (chỉ có một chiều), các *hàng* của mảng hai chiều được sắp xếp kế tiếp nhau theo một trong hai cách sau:

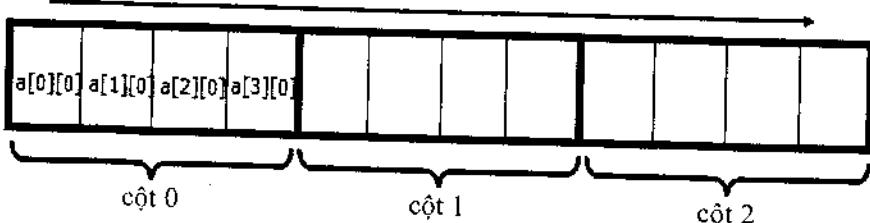
– *Hết dòng này đến dòng khác*: Thứ tự sắp xếp này được gọi là *thứ tự ưu tiên dòng* (*row major order*). Ví dụ, PASCAL và C sử dụng cách sắp xếp này.

Theo chiều tăng dần của địa chỉ bộ nhớ



– *Hết cột này đến cột khác*: Thứ tự sắp xếp này gọi là *thứ tự ưu tiên cột* (*column major order*). Ví dụ, FORTRAN, MATLAB sử dụng cách sắp xếp này.

Theo chiều tăng dần của địa chỉ bộ nhớ



Nếu biết địa chỉ của phần tử đầu tiên của mảng, ta dễ dàng tính được địa chỉ của phần tử tùy ý trong mảng. Thực vậy, xét khai báo:

`int a[m][n]`

Giả sử địa chỉ của phần tử đầu tiên của mảng ( $a[0][0]$ ) là  $\alpha$ . Khi đó địa chỉ của  $a[i][j]$  sẽ là  $\alpha + (i * n + j) * \text{sizeof(int)}$ .

**Ví dụ:** Địa chỉ của các phần tử trong mảng hai chiều được khai báo bởi:

`int a[4][3]`

sẽ là:

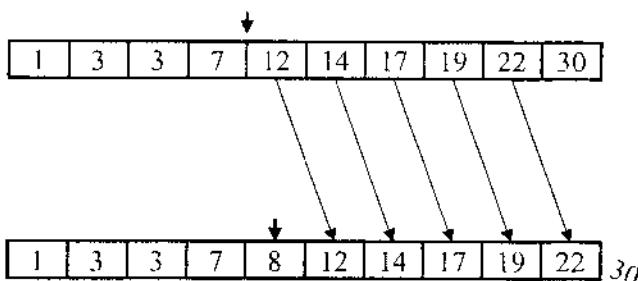
a[0][0] có địa chỉ là  $\alpha$   
a[0][1]  $\alpha + \text{sizeof(int)}$   
a[0][2]  $\alpha + 2 * \text{sizeof(int)}$   
a[1][0]  $\alpha + 3 * \text{sizeof(int)}$   
a[1][1]  $\alpha + 4 * \text{sizeof(int)}$   
a[1][2]  $\alpha + 5 * \text{sizeof(int)}$   
a[2][0]  $\alpha + 6 * \text{sizeof(int)}$

...

### 3.2.3. Các thao tác với mảng

#### Chèn phần tử vào mảng

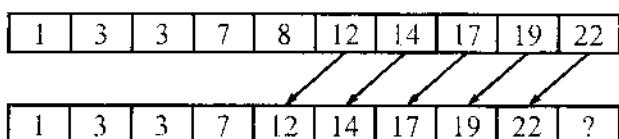
Giả sử ta muốn chèn 8 vào một mảng được sắp xếp (và đảm bảo dãy vẫn là được sắp xếp). Ta có thể thực hiện điều này nhờ việc dịch chuyển sang phải một ô tất cả các phần tử đứng sau vị trí đánh dấu. Tất nhiên, ta phải loại bỏ 30 khi thực hiện điều này.



Việc dịch chuyển tất cả các phần tử là một thao tác chậm (đòi hỏi thời gian tuyến tính đối với kích thước mảng).

#### Xóa bỏ một phần tử

Việc xóa bỏ một phần tử được thực hiện tương tự, ta phải dịch chuyển sang trái tất cả các phần tử sau nó. Phép toán xóa là phép toán chậm; việc thực hiện thường xuyên thao tác này là điều không mong muốn. Phép xóa sẽ làm xuất hiện một vị trí tự do ở cuối mảng. Để đánh dấu nó là tự do, ta cần giữ số lượng phần tử được cắt giữ trong mảng.



### 3.3. DANH SÁCH

#### 3.3.1. Danh sách tuyến tính

**Định nghĩa:** Danh sách tuyến tính là dãy gồm 0 hoặc nhiều hơn các phần tử cùng kiểu cho trước:  $(a_1, a_2, \dots, a_n)$ ,  $n \geq 0$ .

- $a_i$  là phần tử của danh sách.
- $a_1$  là phần tử đầu tiên và  $a_n$  là phần tử cuối cùng.
- $n$  là độ dài của danh sách.

Khi  $n = 0$ , ta có danh sách rỗng (empty list). Các phần tử được sắp xếp thứ tự tuyến tính theo vị trí của chúng trong danh sách. Ta nói  $a_i$  đi trước  $a_{i+1}$ ,  $a_{i+1}$  đi sau  $a_i$  và  $a_i$  ở vị trí  $i$ .

**Ví dụ:**

- Danh sách các sinh viên được sắp thứ tự theo tên.
- Danh sách điểm thi sắp xếp theo thứ tự giảm dần.

Đưa vào ký hiệu:

- $L$  : Danh sách các đối tượng có kiểu *element\_type*.
- $x$  : Một đối tượng kiểu này.
- $p$  : Kiểu vị trí.

$\text{END}(L)$  : Hàm trả lại vị trí đi sau vị trí cuối cùng trong danh sách  $L$ .

**Các phép toán cơ bản**

Dưới đây ta kể ra một số phép toán đối với danh sách tuyến tính:

##### 0. **Creat()**

Khởi tạo danh sách rỗng.

##### 1. **Insert ( $x, p, L$ )**

- Chèn  $x$  vào vị trí  $p$  trong danh sách  $L$ .
- Nếu  $p = \text{END}(L)$ , chèn  $x$  vào cuối danh sách.
- Nếu  $L$  không có vị trí  $p$ , kết quả là không xác định.

##### 2. **Locate ( $x, L$ )**

- Trả lại vị trí của  $x$  trong  $L$ .
- Trả lại  $\text{END}(L)$  nếu  $x$  không xuất hiện.

##### 3. **Retrieve ( $p, L$ )**

- Trả lại phần tử ở vị trí  $p$  trong  $L$ .
- Không xác định nếu  $p$  không tồn tại hoặc  $p = \text{END}(L)$ .

##### 4. **Delete ( $p, L$ )**

- Xóa phần tử ở vị trí  $p$  trong  $L$ . Nếu  $L$  là  $a_1, a_2, \dots, a_n$ , thì  $L$  sẽ trở thành  $a_1, a_2, \dots, a_{p-1}, a_{p+1}, \dots, a_n$ .
- Kết quả không xác định nếu  $L$  không có vị trí  $p$  hoặc  $p = \text{END}(L)$ .

### **5. Next ( $p, L$ )**

- Trả lại vị trí đi ngay sau vị trí  $p$ .
- Nếu  $p$  là vị trí cuối cùng trong  $L$ , thì  $\text{NEXT}(p, L) = \text{END}(L)$ .  $\text{NEXT}$  không xác định nếu  $p$  là  $\text{END}(L)$  hoặc  $p$  không tồn tại.

### **6. Prev ( $p, L$ )**

- Trả lại vị trí trước  $p$ .
- $\text{Prev}$  không xác định nếu  $p$  là 1 hoặc nếu  $L$  không có vị trí  $p$ .

### **7. Makenull ( $L$ )**

- Hàm này biến  $L$  trở thành danh sách rỗng và trả lại vị trí  $\text{END}(L)$

### **8. First ( $L$ )**

- Trả lại vị trí đầu tiên trong  $L$ . Nếu  $L$  là rỗng thì hàm này trả lại  $\text{END}(L)$ .

### **9. Printlist ( $L$ )**

- In ra danh sách các phần tử của  $L$  theo thứ tự xuất hiện.

## **3.3.2. Các cách cài đặt danh sách tuyến tính**

Ta xét các cách cài đặt danh sách tuyến tính cơ bản sau đây:

– Dùng mảng:

+ Cắt giữ các phần tử của danh sách vào các ô liên tiếp của mảng.

– Danh sách móc nối:

+ Các phần tử của danh sách có thể cắt giữ ở các chỗ tùy ý trong bộ nhớ.

+ Mỗi phần tử có con trỏ (hoặc móc nối) đến phần tử tiếp theo.

– Địa chỉ không trực tiếp:

+ Các phần tử của danh sách có thể cắt giữ ở các chỗ tùy ý trong bộ nhớ.

+ Tạo bảng trong đó phần tử thứ  $i$  của bảng cho biết nơi lưu trữ phần tử thứ  $i$  của danh sách.

### **3.3.2.1. Biểu diễn danh sách tuyến tính dưới dạng mảng**

Ta cắt giữ các phần tử của danh sách tuyến tính vào các ô (liên tiếp) của mảng.

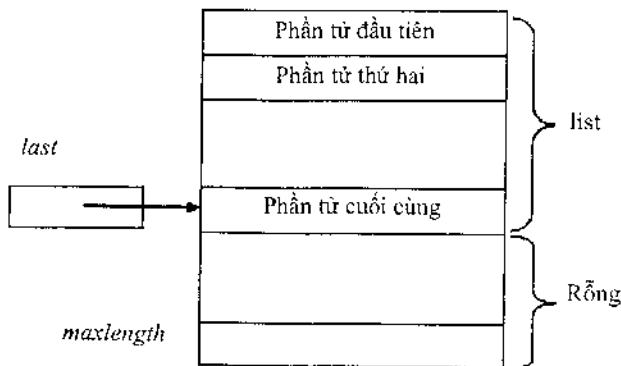
Danh sách sẽ là cấu trúc gồm hai thành phần:

– Thành phần 1: là mảng các phần tử.

– Thành phần 2: **last** – cho biết vị trí của phần tử cuối cùng trong danh sách.

Vị trí có kiểu nguyên (integer) và chạy trong khoảng từ 0 đến  $\text{maxlength}-1$ .

Hàm  $\text{END}(L)$  trả lại giá trị  $\text{last}+1$ .



### Cấu trúc dữ liệu mô tả danh sách dưới dạng mảng

```
const maxlen = 1000 { giá trị thích hợp };
type
```

```
 elementtype = integer; {kiểu của phần tử là nguyên}
 LIST = record
 elements: array[1..maxlen] of elementtype;
 last: integer
 end;
 position = integer;
```

#### Hàm END(L)

```
function END (var L: LIST): position;†
```

```
begin
```

```
 return (L.last + 1)
```

```
end;
```

```
define maxlen 1000
```

```
typedef int element_type;
/* elements are integers */
```

```
typedef struct LIST {
 element_type elements [maxlen];
 int last;
}
```

```
} list_type;
```

#### Cài đặt Insert

```
procedure INSERT (x: elementtype; p: position; var L: LIST);
```

```
{ INSERT đặt x vào vị trí p trong danh sách L }
```

```
var q: position;
```

```

begin
 if $L.last \geq maxlength$ then error('list is full')
 else if ($p > L.last + 1$) or ($p < 1$)
 then error('vị trí là không tồn tại')
 else begin { đây các phần tử tại $p, p + 1, \dots$ xuống dưới một vị trí }
 for $q := L.last$ downto p do $L.elements[q + 1] := L.elements[q];$
 $L.last := L.last + 1;$
 $L.elements[p] := x$
 end
end; { INSERT }

```

### Cài đặt DELETE

```

procedure DELETE (p : position; var L : LIST);
{ DELETE loại phần tử ở vị trí p trong danh sách L }
var q : position;
begin
 if ($p > L.last$) or ($p < 1$) then error('vị trí là không tồn tại')
 else begin
 $L.last := L.last - 1;$
 { đẩy các phần tử ở $p + 1, p + 2, \dots$ lên trên một vị trí }
 for $q := p$ to $L.last$ do $L.elements[q] := L.elements[q + 1]$
 end
end; { DELETE }

```

### Cài đặt LOCATE

```

function LOCATE (x : elementtype; L : LIST): position;
{ LOCATE trả lại vị trí của x trong danh sách L }
var q : position;
begin
 for $q := 1$ to $L.last$ do
 if $L.elements[q] = x$ then
 return (q);
 return ($L.last + 1$) { nếu không tìm thấy }
end; { LOCATE }

```

Có thể nhận thấy một số ưu – khuyết điểm sau đây của cách tổ chức lưu trữ này:

- Cách biểu diễn này rất tiện cho việc truy xuất đến các phần tử của danh sách.
- Do danh sách biến động, số phần tử trong danh sách là không biết trước. Vì vậy ta thường phải khai báo kích thước tối đa cho mảng để dự phòng (*maxlength*). Điều này dẫn đến lãng phí bộ nhớ.
- Các thao tác chèn một phần tử vào danh sách và xóa bỏ một phần tử khỏi danh sách được thực hiện chậm (với thời gian tuyến tính đối với kích thước danh sách).

### 3.3.2.2. Danh sách móc nối

Lưu trữ kế tiếp có những nhược điểm cơ bản đã được phân tích ở trên: đó là việc bô sung và loại trừ phần tử rất tốn kém thời gian, ngoài ra phải kể đến việc sử dụng một không gian liên tục trong bộ nhớ. Việc tổ chức con trỏ (hoặc móc nối) để tổ chức danh sách tuyến tính – mà ta gọi là danh sách móc nối – là giải pháp khắc phục nhược điểm này, tuy nhiên cái giá mà ta phải trả là bộ nhớ dành cho con trỏ.

Ta sẽ xét các cách tổ chức danh sách móc nối sau đây:

- Danh sách móc nối đơn;
- Danh sách móc nối vòng;
- Danh sách móc nối kép.

#### Khi nào dùng danh sách móc nối

– Khi không biết kích thước của dữ liệu – hãy dùng con trỏ và bộ nhớ động (*Unknown data size – use pointers & dynamic storage*).

– Khi không biết kiểu dữ liệu – hãy dùng con trỏ void (*Unknown data type – use void pointers*).

– Khi không biết số lượng dữ liệu – hãy dùng danh sách móc nối (*Unknown number of data – linked structure*).

#### Danh sách móc nối đơn

Trong cách biểu diễn này, danh sách bao gồm các ô (còn gọi là các nút), mỗi ô chứa một phần tử của danh sách và con trỏ đến ô tiếp theo của danh sách.

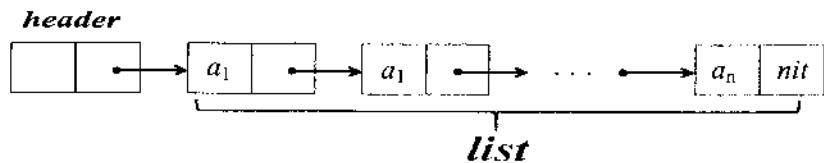
Nếu danh sách là  $a_1, a_2, \dots, a_n$ , thì ô lưu trữ  $a_i$  có con trỏ (mối nối) đến ô lưu trữ  $a_{i+1}$  với  $i = 1, 2, \dots, n - 1$ . Ô lưu trữ  $a_n$  có con trỏ rỗng, mà ta sẽ ký hiệu là **nil**. Như vậy mỗi ô có cấu trúc:

| Element | Link/Pointer |
|---------|--------------|
|---------|--------------|

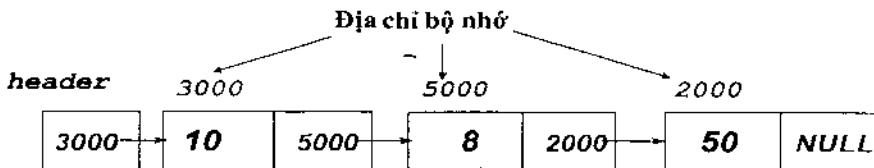
Có một ô đặc biệt gọi là ô **header** để trỏ ra ô chứa phần tử đầu tiên trong danh sách ( $a_1$ ). Ô header không lưu trữ phần tử nào cả. Trong trường hợp danh sách rỗng, con trỏ của header là **nil** (hoặc **NULL**), và không có ô nào khác.

Các ô có thể nằm ở vị trí bất kỳ trong bộ nhớ.

Danh sách mòc nối được tổ chức như trong hình vẽ sau:



Mỗi nối chỉ ra địa chỉ bộ nhớ của nút tiếp theo trong danh sách.



### Mô tả danh sách nối đơn trong C

```
typedef <Kiểu dữ liệu của phần tử> ElementType;
struct PointerType{
 ElementType Inf;
 struct PointerType *Next;);
typedef struct PointerType LIST;
```

Khai báo trên định nghĩa kiểu Node, trong đó Node là kiểu báy ghi mô tả một nút gồm hai trường:

– Inf – lưu dữ liệu có kiểu là ElementType (đã được định nghĩa, có thể gồm nhiều thành phần).

– Next thuộc kiểu con trỏ PointerType, lưu địa chỉ của nút kế tiếp.

Cần có biến con trỏ First lưu địa chỉ của nút đầu tiên trong danh sách:

```
PointerType *First;
```

### Hàm INSERT(x,p)

Giả sử cần chèn một phần tử có nội dung dữ liệu là  $x$  (có kiểu DataType) vào danh sách. Vị trí cần chèn được xác định là sau nút được trỏ bởi con trỏ **Pred**. Thao tác được tiến hành theo các bước:

- (1) Xin cấp phát một nút mới cho con trỏ **TempNode** để lưu  $x$ ;
- Nối nút này vào danh sách tại vị trí cần chèn:
  - + (2) **TempNode**→**Next** bằng **Pred**→**Next**
  - + (3) Ghi nhận lại **Pred**→**Next** bằng **TempNode**.

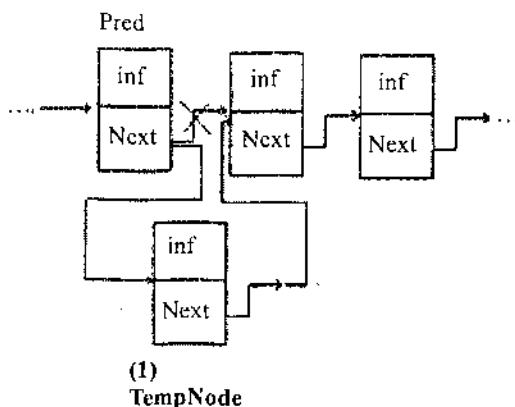
```
PointerType *Insert_Middle(PointerType *Pred, ElementType X)
{ PointerType *TempNode;
```

```

TempNode = (PointerType *)malloc(sizeof(PointerType)); // (1)
TempNode->Inf=X; // (1)
TempNode->Next=Pred->Next; // (2)
Pred->Next=TempNode; // (3)
return TempNode;
}

```

Sơ đồ của thao tác cần làm với danh sách được minh họa như sau:



**Chú ý:** Việc chèn một nút không ảnh hưởng đến mối liên kết của các nút đứng sau vị trí chèn, do đó không phải thực hiện dồn phần tử như trong cài đặt mảng.

#### Hàm DELETE(p)

Hàm này thực hiện loại bỏ nút đứng sau nút đang được trỏ bởi **Pred** và trả lại giá trị **X** của phần tử trong nút bị xóa. Việc xóa chỉ tiến hành khi danh sách là khác rỗng (cần phải kiểm tra điều kiện này trước khi gọi hàm DELETE để thực hiện thao tác xóa).

Sử dụng con trỏ **TempNode** để ghi nhận nút cần xóa, thao tác xóa được tiến hành theo các bước sau:

- (1) Gán **TempNode** bằng **Pred->Next**.
- (2) Đặt lại **Pred->Next** bằng **TempNode->Next**.
- (3) Thu hồi vùng nhớ được trỏ bởi **TempNode**.

```

ElementType Delete(PointerType *Pred)
{
 ElementType X; PointerType *TempNode;
 TempNode=Pred->Next; // (1)
 Pred->Next=Pred->Next->Next; // (2)
 X=TempNode->Inf;
}

```

```

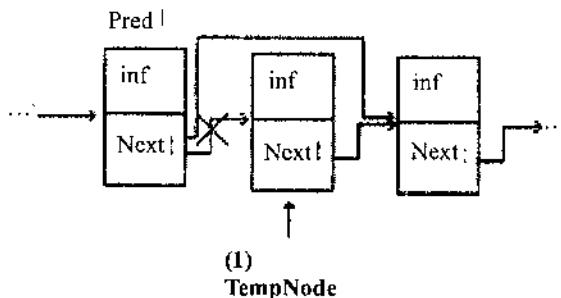
 free(TempNode); // (3)

 return X;

}

```

Sơ đồ của thao tác cần làm với danh sách được minh họa trong hình dưới đây:



**Chèn một nút vào đầu danh sách**

```

PointerType *Insert_ToHead(PointerType *First,
 ElementType X)

{ PointerType *TempNode;

 TempNode = (PointerType *) malloc(sizeof(PointerType));
 TempNode->Inf=X; TempNode->Next=First; First=TempNode;
 return First;
}

```

**Chèn một nút vào cuối danh sách được trả bởi First**

```

PointerType *Insert_ToLast(PointerType *First, ElementType X)

{ PointerType *NewNode; PointerType *TempNode;

 NewNode = (PointerType *) malloc(sizeof(PointerType));
 NewNode->Inf=X; TempNode=First;
 while (TempNode->next != NULL) // Lần đến cuối danh sách
 TempNode= TempNode->next;
 TempNode->next = NewNode;
 return First;
}

```

**Chú ý:** Nếu thao tác này phải thực hiện thường xuyên thì cần đưa vào con trả Last để trả đến phần tử cuối cùng của danh sách.

Xóa nút ở đầu danh sách

```
PointerType *Delete_Head(PointerType *First)
{ PointerType *TempNode;
 TempNode=First->Next;
 free(First);
 return TempNode;
}
```

Xóa nút ở cuối danh sách

```
PointerType *Delete_Last(PointerType *First)
{ PointerType *TempNode1, *TempNode2;
 TempNode1=First; TempNode2=First;
 while (TempNode1->next != NULL) // Lần đến cuối danh sách
 { TempNode2 = TempNode1;
 TempNode1= TempNode1->next; }
 TempNode2->next = NULL;
 free(TempNode1);
 return First;
}
```

**Chú ý:** Nếu thao tác này phải thực hiện thường xuyên thì cần đưa vào con trỏ Last để trả đến phần tử cuối cùng của danh sách.

**IsEmpty** và **PrintLIST**

-- Kiểm tra danh sách rỗng

```
int IsEmpty(PointerType *First)
{
 return !First;
}
```

- Xóa danh sách

```
PointerType *MakeNull(PointerType *First)
{
 while (!IsEmpty(First))
 First=Delete_Head(First);
 return First;
}
```

- Đưa ra tất cả các phần tử của danh sách

```

void Print(PointerType *First)
{
 PointerType *TempNode;
 TempNode=First; int count = 0;
 while (TempNode) {
 printf("%6d", TempNode->Inf); count++;
 TempNode=TempNode->Next;
 if (count % 12 == 0) printf("\n");
 }
 printf("\n");
}

```

### Ví dụ 1: Chương trình minh họa trên C

Xây dựng chương trình thực hiện công việc sau đây:

- Tạo ngẫu nhiên một danh sách với các phần tử là các số nguyên. Sau đó:
- Từ danh sách tạo được xây dựng hai danh sách: một danh sách chứa tất cả các số dương còn danh sách kia chứa tất cả các số âm của danh sách ban đầu.

Chương trình dưới đây sẽ sử dụng một số phép toán cơ bản đối với danh sách mốc nối để thực hiện các công việc đặt ra.

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef long ElementType;
struct PointerType{
 ElementType Inf;
 PointerType *Next; };

PointerType *Insert_ToHead(PointerType *First, ElementType X);
PointerType *Insert_Middle(PointerType *Pred, ElementType X);
PointerType *Delete_Head(PointerType *First);
ElementType Delete(PointerType *Pred);
void Print(PointerType *First);
int IsEmpty(PointerType *First);
PointerType *MakeNull(PointerType *First);

```

```

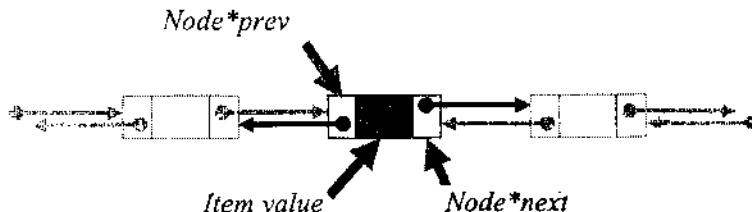
int main() {
 PointerType *S1, *S2, *S3, *V1, *V2, *V3;
 ElementType a; int i, n;
 clrscr(); randomize(); S1=NULL;
 // Tao phan tu dau tien
 a=-100+random(201);
 S1=Insert_ToHead(S1, a);
 printf("Nhap vao so luong phan tu n = ");
 scanf("%i", &n); printf("\n");

 // Tao ngau nhien danh sach va dua ra man hinh
 V1=S1;
 for (i=2; i<=n; i++) {
 a=-100+random(201);
 V1=Insert_Middle(V1, a);
 }
 printf("==> Danh sach ban dau: \n"); Print(S1);
printf("\n");
 V1 = S1; S2 = NULL; S3 = NULL;
 while (V1) {
 if (V1->Inf > 0)
 if (!S2) { S2=Insert_ToHead(S2, V1->Inf); V2 = S2; }
 else { Insert_Middle(V2, V1->Inf); V2 = V2->Next; }
 if (V1->Inf < 0)
 if (!S3) { S3=Insert_ToHead(S3, V1->Inf); V3 = S3; }
 else { Insert_Middle(V3, V1->Inf); V3 = V3->Next; }
 V1= V1->Next;
 }
 printf("==> Danh sach so duong: \n"); Print(S2);
printf("\n");
 printf("==> Danh sach so am: \n"); Print(S3);
printf("\n");
 S1=MakeNull(S1); S2=MakeNull(S2); S3=MakeNull(S3);
 getch(); getch();
}

```

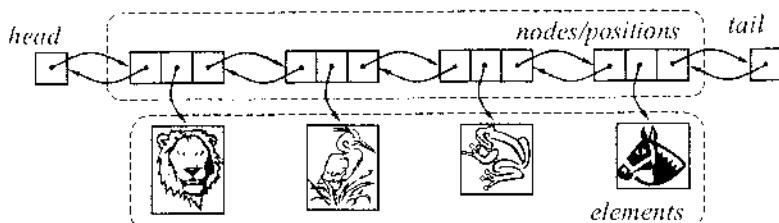
## Danh sách mốc nối đôi

Trong nhiều ứng dụng, ta muốn duyệt danh sách theo cả hai chiều một cách hiệu quả. Hoặc cho một phần tử, ta cần xác định cả phần tử đi trước lẫn phần tử đi sau nó trong danh sách một cách nhanh chóng. Trong tình huống như vậy, ta có thể gán cho mỗi ô trong danh sách con trỏ đến cả phần tử đi trước lẫn phần tử đi sau nó trong danh sách.



Cách tổ chức này được gọi là *danh sách mốc nối đôi*.

Cách tổ chức danh sách mốc nối đôi được minh họa trong hình vẽ sau:



Có hai mứt đặc biệt: tail (đuôi) và head (đầu).

- head có con trỏ trái prev là null.
- tail có con trỏ phải next là null.

Các phép toán cơ bản được xét tương tự như danh sách nối đơn.

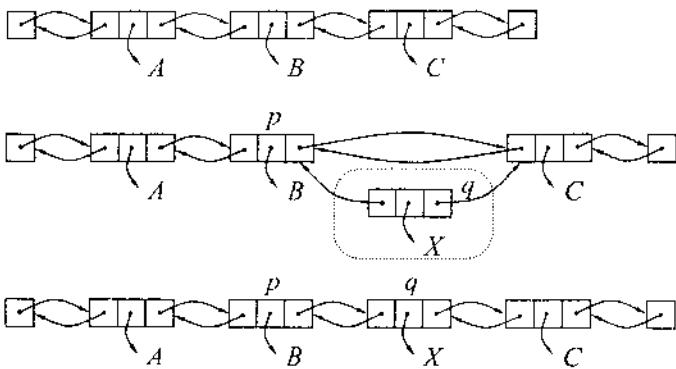
### Cách mô tả danh sách mốc nối đôi trên C

```
struct dllist {
 int number;
 struct dllist *next;
 struct dllist *prev;
};

struct dllist *head, *tail;
```

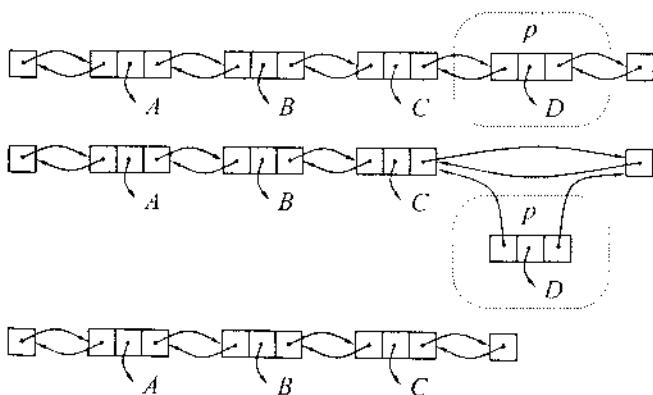
### Phép toán chèn

Dãy các thao tác cần làm để thực hiện phép toán chèn được mô tả trong hình vẽ sau:



### Phép toán Xóa

Ta mô tả phép toán  $\text{remove}(p)$ , trong đó  $p = \text{last}()$ .



### Chương trình trên C minh họa một số phép toán

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct dllist {
 int number;
 struct dllist *next;
 struct dllist *prev;
};

struct dllist *head, *tail;

/* Nối đuôi một phần tử mới */
void append_node(struct dllist *lnode);
```

```

/* Chèn một phần tử mới vào ô trống bởi after */
void insert_node(struct dllist *lnode, struct dllist *after);
/* Xóa ô trống bởi lnode */
void remove_node(struct dllist *lnode);

int main(void) {
 struct dllist *lnode; int i = 0;
 /* Bổ sung một vài số vào danh sách mốc nối dài */
 for(i = 0; i <= 5; i++) {
 lnode = (struct dllist *)malloc(sizeof(struct dllist));
 lnode->number = i;
 append_node(lnode);
 }

 /* Đưa ra các phần tử của danh sách mốc nối dài từ đầu đến đuôi */
 printf(" Traverse the dll list forward \n");
 for(lnode = head; lnode != NULL; lnode = lnode->next)
 { printf("%d\n", lnode->number); }

 /* Đưa ra các phần tử của danh sách mốc nối dài từ đuôi về đầu */
 printf(" Traverse the dll list backward \n");
 for(lnode = tail; lnode != NULL; lnode = lnode->prev)
 { printf("%d\n", lnode->number); }

 /* Hủy danh sách mốc nối dài */
 while(head != NULL)
 remove_node(head);
 getch(); /* Wait for ...*/
 return 0;
}

void append_node(struct dllist *lnode) {
 if(head == NULL) {
 head = lnode;
 lnode->prev = NULL;
 }
 else {

```

```

 tail->next = lnode;
 lnode->prev = tail;
 }
 tail = lnode;
 lnode->next = NULL;
}

void insert_node(struct dllist *lnode, struct dllist *after) {
 lnode->next = after->next;
 lnode->prev = after;
 if(after->next != NULL)
 after->next->prev = lnode;
 else
 tail = lnode;
 after->next = lnode;
}

void remove_node(struct dllist *lnode) {
 if(lnode->prev == NULL)
 head = lnode->next;
 else
 lnode->prev->next = lnode->next;
 if(lnode->next == NULL)
 tail = lnode->prev;
 else
 lnode->next->prev = lnode->prev;
}

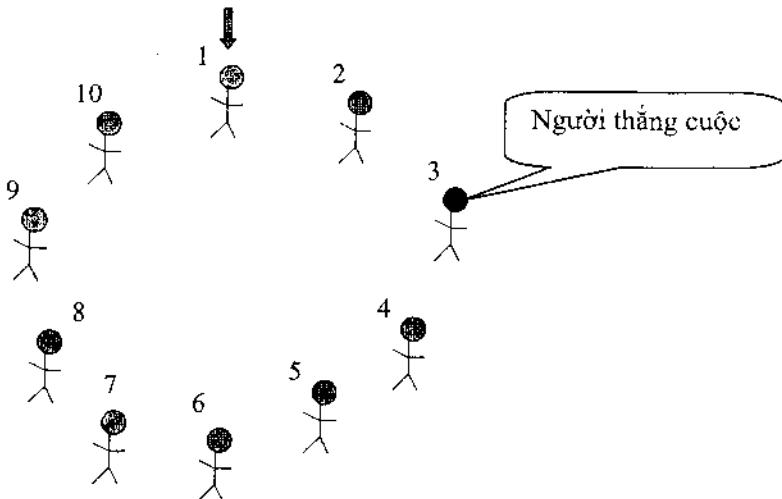
```

### 3.3.3. Các ví dụ ứng dụng

#### 3.3.3.1. Bài toán Josephus

n khách hàng tham gia vào vòng quay trúng thưởng của Công ty X. Các khách hàng được xếp thành một vòng tròn. Giám đốc công ty lựa chọn ngẫu nhiên một số m ( $m \leq n$ ). Bắt đầu từ một người được chọn ngẫu nhiên trong số các khách hàng, Giám đốc đếm theo chiều kim đồng hồ và dừng lại mỗi khi đếm đến m. Khách hàng ở vị trí này sẽ rời khỏi cuộc chơi. Quá trình được lặp lại cho đến khi chỉ còn một người. Người cuối cùng còn lại là người trúng thưởng!

Hình vẽ dưới đây mô tả trò chơi với n = 10, m = 5.



Có thể giải bài toán này nhờ danh sách nối đôi.

### Thuật toán giải Josephus Problem

```

void josephus(int n, int m){
 // Khai báo danh sách nối kép
 struct dllist { int number;
 struct dllist *next; struct dllist *prev;};
 struct dllist *dList, *curr;
 int i, j;
 // khởi tạo danh sách gồm các khách hàng 1 2 3 ... n
 for (i = 1; i <= n; i++)
 insert(dList, i); // Phải xây dựng hàm này
 // khởi động vòng đếm bắt đầu từ người 1
 curr = dList->next;
 // thực hiện vòng đếm để loại tất cả
 // chỉ để lại 1 người trong danh sách
 for (i=1; i < n; i++) {
 // đếm bắt đầu từ người hiện tại curr, đi qua m người.
 // ta phải thực hiện điều này m-1 lần.
 for (j=1; j <= m-1; j++)
 { // con trỏ kế tiếp
 curr = curr->next;
 // nếu curr dừng tại header, cần thực hiện
di chuyển tiếp
 if (curr == dList)

```

```

 curr = curr->next;
 }

 printf("Xoa khach hang %i \n", curr->nodeValue);
 // trien khai tiep curr va xoa nut tai diem dung
 curr = curr->next;
 erase(curr->prev); // Phai xay dung ham nay
 // co the loại bỏ nút ở cuối danh sách,
 // vi the curr phai tro ve head va tiep tuc
 if (curr == dList) curr = curr->next;
}

printf("\n Khach hang %i la nguoi thang
cuoc\n", curr->nodeValue);
// xoa bo nút cuối cùng và đầu danh sách
delete curr; delete dList;
}

void main()
{
 // n - số lượng người chơi
 // m - số cần đếm
 int n, m;

 printf("Nhập vào n = ");
 scanf("%i", &n); printf("\n");

 // tạo ngẫu nhiên số m: 1<=m<=n
 m = 1 + random(n);

 printf(" m = %i", m);
 // giải bài toán và đưa ra người thắng cuộc
 josephus(n, m);
}

```

### **3.3.3.2. Biểu diễn đa thức**

Xét đa thức:

$$P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n.$$

Để biểu diễn đa thức, đơn giản nhất là dùng mảng  $a[i]$  cất giữ hệ số của  $x^i$ . Các phép toán với các đa thức như: cộng hai đa thức, nhân hai đa thức,..., khi các đa thức được biểu diễn bởi mảng, có thể cài đặt một cách đơn giản.

Tuy nhiên, khi đa thức có nhiều hệ số bằng 0, cách biểu diễn đa thức dưới dạng mảng rất tốn kém bộ nhớ, chẳng hạn, việc biểu diễn đa thức  $x^{1000} - 1$  đòi hỏi mảng gồm 1001 phần tử.

Trong trường hợp đa thức thưa (có nhiều hệ số bằng 0), có thể sử dụng biểu diễn đa thức bởi danh sách mốc nối: ta sẽ xây dựng danh sách chỉ chứa các hệ số khác 0 cùng số mũ tương ứng. Tuy nhiên, khi đó việc cài đặt các phép toán lại phức tạp hơn.

**Ví dụ:** Có thể sử dụng khai báo sau đây để khai báo danh sách mốc nối của hai đa thức Poly1 và Poly2.

```
struct Polynom {
 int coeff;
 int pow;
 struct Polynom *link;
} *Poly1, *Poly2
```

Việc cài đặt phép cộng hai đa thức Poly1 và Poly2 đòi hỏi phải duyệt qua hai danh sách mốc nối của chúng để tính hệ số của đa thức tổng PolySum (cũng được biểu diễn bởi danh sách mốc nối).

Dưới đây là bài tập tốt để luyện tập cài đặt các thao tác với danh sách mốc nối.

**Ví dụ:** Thuật toán tính tổng hai đa thức.

### Thuật toán SumTwoPol

```
node=(TPol *)malloc (sizeof(TPol));
PolySum=node;
ptr1=Poly1;
ptr2=Poly2;
while(ptr1!=NULL && ptr2!=NULL)
{
 ptr=node;
 if (ptr1->pow > ptr2->pow)
 {
 node->coeff=ptr2->coeff;
 node->pow=ptr2->pow;
 ptr2=ptr2->link; //update ptr list 2
 }
 else if (ptr1->pow < ptr2->pow)
```

```

 {
 node->coeff=ptr1->coeff;
 node->pow=ptr1->pow;
 ptr1=ptr1->link; //update ptr list 1
 }
 else
 {
 node->coeff=ptr2->coeff+ptr1->coeff;
 node->pow=ptr2->pow;
 ptr1=ptr1->link; //update ptr list 1
 ptr2=ptr2->link; //update ptr list 2
 }

 node=(TPol *)malloc (sizeof(TPol));
 ptr->link=node; //update ptr list 3
} //end of while

if (ptr1==NULL) //end of list 1
{
 while(ptr2!=NULL)
 {
 node->coeff=ptr2->coeff;
 node->pow=ptr2->pow;
 ptr2=ptr2->link; //update ptr list 2
 ptr=node; node=(TPol *)malloc (sizeof(TPol));
 ptr->link=node;
 } //update ptr list 3
}
else if (ptr2==NULL) //end of list 2
{
 while(ptr1!=NULL)
 {
 node->coeff=ptr1->coeff;
 node->pow=ptr1->pow;
 ptr1=ptr1->link; //update ptr list 2
 ptr=node;
 node=(TPol *)malloc (sizeof(TPol));
 }
}

```

```

 ptr->link=node;
} //update ptr list 3
}
node=NULL;
ptr->link=node;
}

```

### 3.3.4. Phân tích sử dụng danh sách mốc nối

Những ưu điểm của việc dùng danh sách mốc nối:

- Không xảy ra vượt mảng, ngoại trừ hết bộ nhớ.
- Chèn và Xóa được thực hiện dễ dàng hơn cài đặt mảng.
- Với những bản ghi lớn, thực hiện di chuyển con trỏ nhanh hơn nhiều so với thực hiện di chuyển các phần tử của danh sách.

Những bất lợi khi sử dụng danh sách mốc nối:

- Dùng con trỏ đòi hỏi bộ nhớ phụ.
- Danh sách mốc nối không cho phép trực truy.
- Tốn thời gian cho việc duyệt và biến đổi con trỏ.
- Lập trình với con trỏ *khá rắc rối*.

#### So sánh các phương pháp

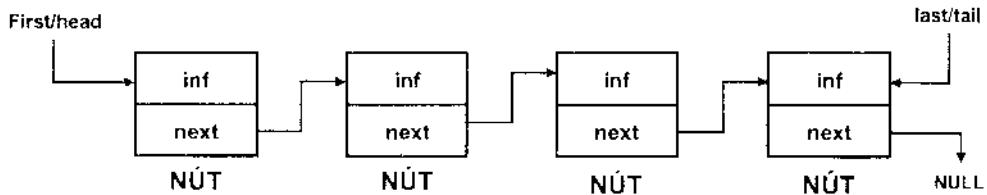
Việc lựa chọn cách cài đặt mảng hay cài đặt con trỏ để biểu diễn danh sách tùy thuộc vào việc thao tác nào là thao tác thường phải dùng nhất. Dưới đây là một số nhận xét về hai cách cài đặt:

- Cách cài đặt mảng phải khai báo kích thước tối đa. Nếu ta không lường trước được giá trị này thì nên dùng cài đặt con trỏ.
- Có một số thao tác có thể thực hiện nhanh trong cách cài đặt này nhưng lại chậm trong cách cài đặt kia: INSERT và DELETE đòi hỏi thời gian hằng số trong cài đặt con trỏ nhưng trong cài đặt mảng lại đòi hỏi thời gian  $O(N)$  với  $N$  là số phần tử của danh sách. PREVIOUS và END đòi hỏi thời gian hằng số trong cài đặt mảng, nhưng thời gian đó trong cài đặt con trỏ lại là  $O(N)$ .
- Cách cài đặt mảng đòi hỏi dành không gian nhớ định trước không phụ thuộc vào số phần tử thực tế của danh sách. Trong khi đó cài đặt con trỏ chỉ đòi hỏi bộ nhớ cho các phần tử đang có trong danh sách. Tuy nhiên cách cài đặt con trỏ lại đòi hỏi thêm bộ nhớ cho con trỏ.

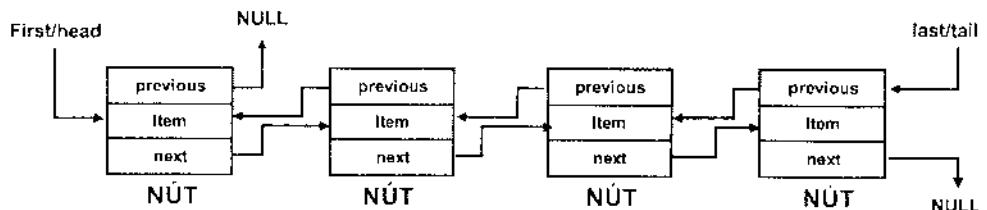
#### Tóm tắt hai cách biểu diễn bởi con trỏ

- Trong cách biểu diễn danh sách mốc nối đơn hoặc danh sách mốc nối đôi, cho dù bằng cách nào, **header** và **tail** cũng chỉ là các con trỏ đến nút đầu tiên và nút cuối cùng, chúng không phải là nút.

– Danh sách mốc nối đơn:



– Danh sách mốc nối đôi:



#### Kiểu dữ liệu trừu tượng danh sách: So sánh các cách cài đặt

|                 | Mảng                                           | Móc nối đơn                        | Móc nối đôi                   |
|-----------------|------------------------------------------------|------------------------------------|-------------------------------|
| Creation        | O(1) nếu dùng kiểu có sẵn và O(n) nếu trãi lại | O(1)                               | O(1)                          |
| Destruction     | giống khởi tạo                                 | O(n)                               | O(n)                          |
| isEmpty()       | O(1)                                           | O(1)                               | O(1)                          |
| getLength()     | O(1)                                           | O(1) có biến size<br>O(n) không có | O(1) có size<br>O(n) không có |
| insertFirst()   | O(n)                                           | O(1)                               | O(1)                          |
| insertLast()    | O(1)                                           | O(1) có tail<br>O(n) không có      | O(1) có tail<br>O(n) không có |
| insertAtIndex() | O(n)                                           | O(n)                               | O(n)                          |
| deleteFirst()   | O(n)                                           | O(1)                               | O(1)                          |
| deleteLast()    | O(1)                                           | O(n) ngay cả có tail               | O(1) có tail<br>O(n) không có |

|                              |        |                                          |                                          |
|------------------------------|--------|------------------------------------------|------------------------------------------|
| <code>deleteAtIndex()</code> | $O(n)$ | $O(n)$                                   | $O(n)$                                   |
| <code>getFirst()</code>      | $O(1)$ | $O(1)$                                   | $O(1)$                                   |
| <code>getLast()</code>       | $O(1)$ | $O(1)$ có <i>tail</i><br>$O(n)$ không có | $O(1)$ có <i>tail</i><br>$O(n)$ không có |
| <code>getAtIndex()</code>    | $O(1)$ | $O(n)$                                   | $O(n)$                                   |

Nhiều khi cần xác định thêm một số phép toán khác, chúng có ích khi các phép toán insertions/deletions phải thực hiện nhiều trong danh sách. Trong các tình huống như vậy nên sử dụng danh sách mốc nối đôi.

|                             | <i>Array</i> | <i>Singly-L</i> | <i>Doubly-L</i> |
|-----------------------------|--------------|-----------------|-----------------|
| <code>insertBefore()</code> | $O(n)$       | $O(n)$          | $O(1)$          |
| <code>insertAfter()</code>  | $O(n)$       | $O(1)$          | $O(1)$          |
| <code>deleteBefore()</code> | $O(n)$       | $O(n)$          | $O(1)$          |
| <code>deleteAfter()</code>  | $O(n)$       | $O(1)$          | $O(1)$          |
| <code>next()</code>         | $O(1)$       | $O(1)$          | $O(1)$          |
| <code>previous()</code>     | $O(1)$       | $O(n)$          | $O(1)$          |

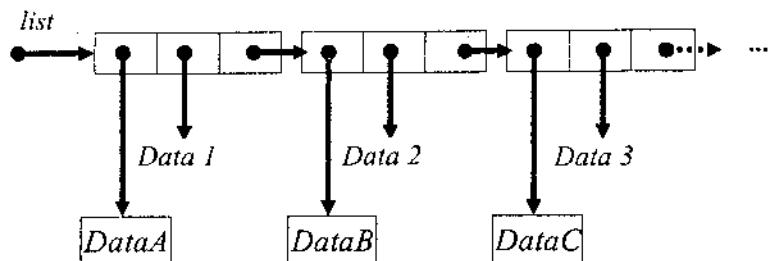
### 3.3.5. Một số biến thể của danh sách mốc nối

Có nhiều biến thể của danh sách mốc nối. Ta kể ra một số biến thể thường gặp:

- Danh sách mốc nối đa dữ liệu;
- Danh sách mốc nối vòng;
- Danh sách mốc nối đôi vòng;
- Danh sách mốc nối của các danh sách;
- Danh sách đa mốc nối.

Các phép toán cơ bản với các biến thể này được xây dựng tương tự như đối với danh sách mốc nối đơn và danh sách mốc nối kép mà ta xét ở trên.

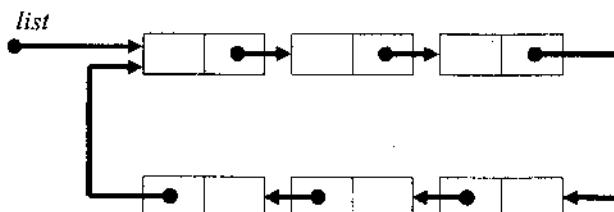
## Danh sách mốc nối đa dữ liệu



```
Struct {
 Node * next;
 void * item1;
 void * item2;
}
```

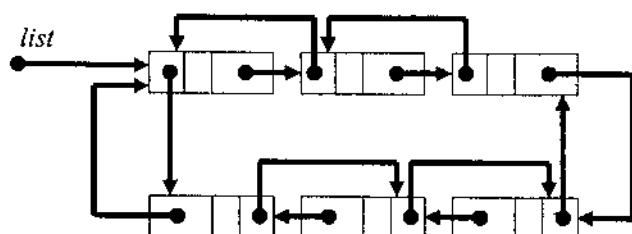
Có hai loại dữ liệu

## Danh sách nối vòng



```
Struct {
 DataType * item;
 Node * next;
}
```

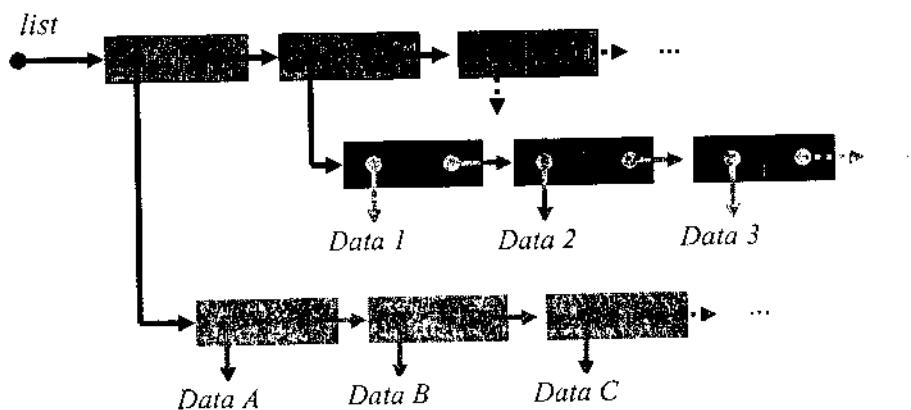
## Danh sách nối đôi vòng



```
Struct {
 void * item;
```

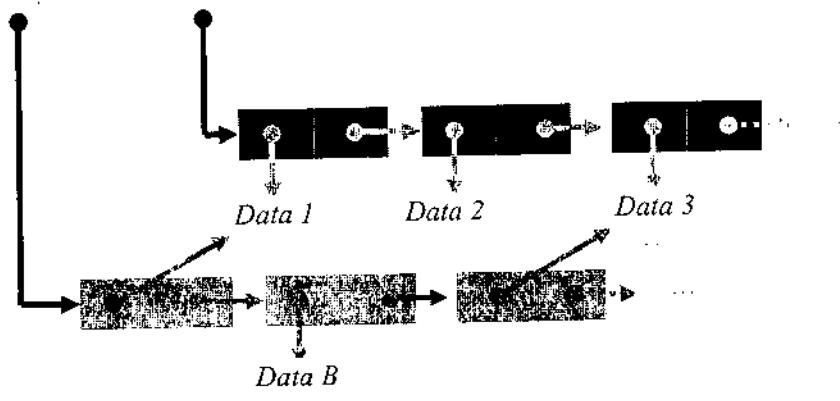
```
 Node * prev;
 Node * next;
}
```

### Danh sách mòc nối của các danh sách



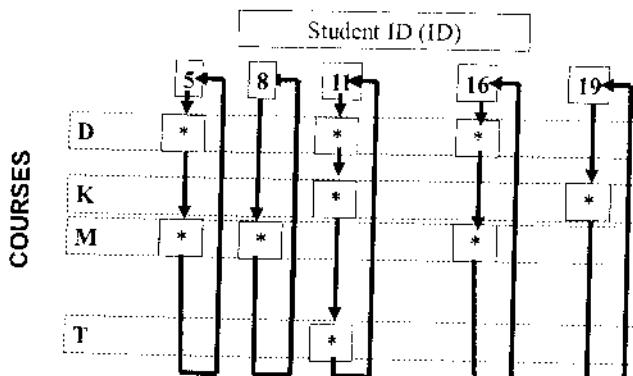
```
Struct {
 Node * item;
 Node * next;
}
```

### Danh sách đa mòc nối

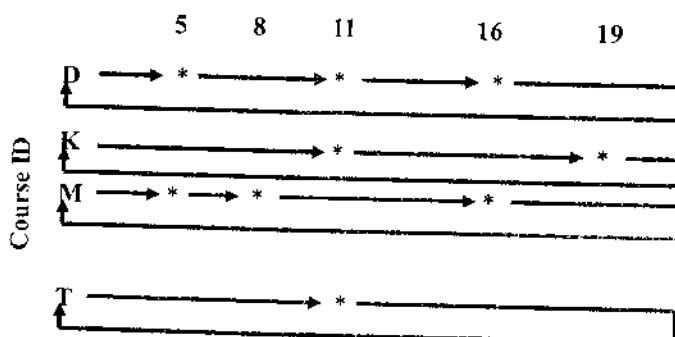


```
Struct {
 Node * item;
 Node * next;
}
```

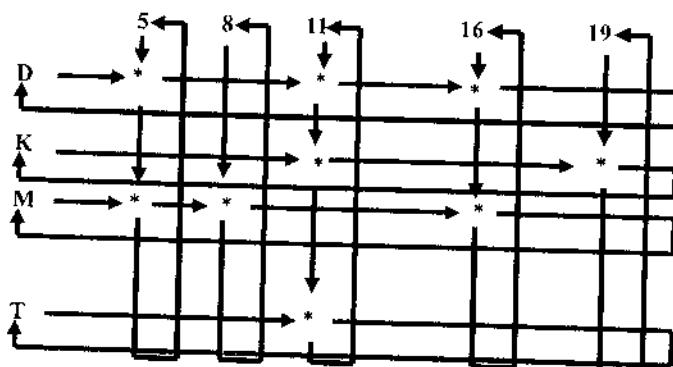
Ví dụ: Ta xét một ví dụ ứng dụng đa danh sách.



Mỗi một mã (ID) sinh viên có một list.



Mỗi môn học có một list.



Mỗi sinh viên có một list.

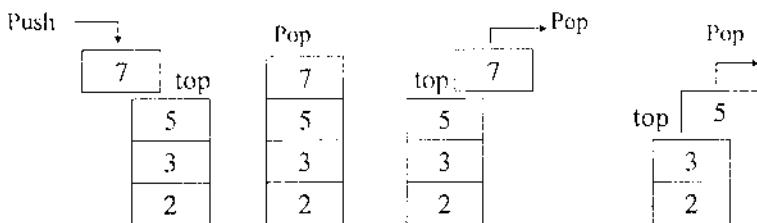
## Duyệt đa danh sách

- Chọn một danh sách tương ứng với một header, chẳng hạn “Mã Sinh Viên = 5”
- Lặp lại:
  - + Duyệt danh sách, tại mỗi nút đi theo danh sách môn học.
  - + Định vị được các đầu danh sách, chẳng hạn “Mã môn học = D”.
- Thu được: Sinh viên 5 đăng ký học môn D và M.

## 3.4. NGĂN XẾP

### 3.4.1. Kiểu dữ liệu trừu tượng ngăn xếp

Ngăn xếp là dạng đặc biệt của danh sách tuyến tính, trong đó các đối tượng được *nạp vào* (push) và *lấy ra* (pop) chỉ từ một đầu được gọi là *đỉnh* (top) của danh sách.



**Nguyên tắc:** Vào sau – Ra trước (Last-in, First-out, viết tắt là LIFO).

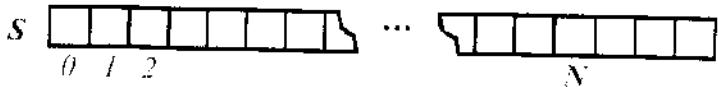
- Các phép toán cơ bản với ngăn xếp:
  - + push(object): bổ sung vào một phần tử, gọi tắt là *nạp vào*.
  - + pop(): loại bỏ và trả lại phần tử nạp vào sau cùng, gọi tắt là *lấy ra*.
- Các phép toán hỗ trợ:
  - + top(): trả lại phần tử nạp vào sau cùng mà không loại nó ra khỏi ngăn xếp.
  - + size(): trả lại số lượng phần tử được lưu trữ.
  - + boolean isEmpty(): nhận biết có phải ngăn xếp rỗng.

Ta xét hai cách tổ chức ngăn xếp:

- + Sử dụng mảng;
- + Sử dụng danh sách mốc nối.

### 3.4.2. Ngăn xếp dùng mảng

Cách đơn giản nhất để cài đặt ngăn xếp là dùng mảng ( $S$ ). Ta nạp các phần tử theo thứ tự từ trái sang phải. Có biến lưu giữ chỉ số của phần tử ở đầu ngăn xếp ( $N$ ).



**Algorithm size()**

**return** N + 1

**Algorithm pop()**

**if** isEmpty() **then**

        Error("EmptyStack")

**else**

        N  $\leftarrow$  N - 1

**return** S[N + 1]

Cài đặt ngăn xếp dùng mảng trên C

```

typedef Item;
static Item *s;
static int N;

void STACKinit(int maxN) {
 s = (Item *) malloc(maxN*sizeof(Item));
 N = 0; }

int STACKempty()
{
 return N==0; }

void STACKpush(Item item)
{
 s[N++] = item; }

Item STACKpop()
{
 return s[--N]; }

```

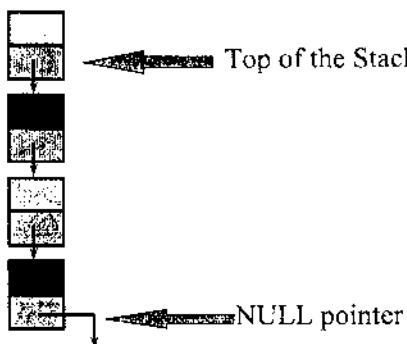
### 3.4.3. Cài đặt ngăn xếp với danh sách móc nối

Trong cách cài đặt ngăn xếp dùng danh sách móc nối, ngăn xếp được cài đặt như danh sách móc nối với các thao tác bổ sung và loại bỏ luôn làm việc với ô đầu tiên.

## Mô tả ngăn xếp

```
struct StackNode {
 float item;
 StackNode *next;
};

struct Stack {
 StackNode *top;
};
```



## Các phép toán cơ bản

Ta sẽ xét việc cài đặt các phép toán:

1. Khởi tạo:

```
Stack *StackConstruct();
```

2. Kiểm tra ngăn xếp rỗng:

```
int StackEmpty(const Stack* s);
```

3. Kiểm tra tràn ngăn xếp:

```
int StackFull(const Stack* s);
```

4. Nạp vào (Push): *Nạp phần tử vào đầu ngăn xếp*

```
int StackPush(Stack* s, float* item);
```

5. Lấy ra (Pop): *Trả lại giá trị của phần tử ở đầu ngăn xếp và loại bỏ nó*

```
float pop(Stack* s);
```

6. Dưa ra các phần tử của ngăn xếp:

```
void Disp(Stack* s);
```

## Khởi tạo ngăn xếp (Initialize Stack)

```
Stack *StackConstruct() {

 Stack *s;
 s = (Stack *)malloc(sizeof(Stack));
 if (s == NULL) {

 return NULL; // No memory
 }

 s->top = NULL;

 return s;
}
```

Hủy ngăn xếp

```
void StackDestroy(Stack *s) {
 while (!StackEmpty(s)) {
 StackPop(s);
 }
 free(s);
}
```

Các hàm hỗ trợ

```
/** Kiểm tra Stack rỗng */
int StackEmpty(const Stack *s) {
 return (s->top == NULL);
}

/** Thông báo Stack tràn */
int StackFull() {
 printf("\n NO MEMORY! STACK IS FULL");
 getch();
 return 1;
}
```

Đưa ra các phần tử của ngăn xếp

```
void disp(Stack* s) {
 StackNode* node;
 int ct = 0; float m;
 printf("\n\n DANH SACH CAC PHAN TU CUA STACK \n\n");
 if (StackEmpty(s)) {
 printf("\n\n >>>> EMPTY STACK <<<<\n");
 getch(); }
 else {
 node= s->top;
 do {
```

```

 m=node->item; printf("%8.3f ", m); ct++;
 if (ct % 9 == 0) printf("\n");
 node = node->next;
} while (!(node == NULL));
printf("\n");
}
}

```

### Bổ sung (Nạp vào) – Push

Cần thực hiện các thao tác sau:

- (1) Tạo nút mới cho item;
- (2) Móc nối nút mới đến nút ở đầu;
- (3) Đặt nút mới thành nút đầu mới.

```

int StackPush(Stack *s, float item) {
 StackNode *node;
 node = (StackNode *)malloc(sizeof(StackNode)); // (1)
 if (node == NULL) {
 StackFull();
 return 1; // Tràn Stack: hết bộ nhớ
 }
 node->item = item; // (1)
 node->next = s->top; // (2)
 s->top = node; // (3)
 return 0;
}

```

### Loại bỏ (Lấy ra) – Pop

Thuật toán:

1. Kiểm tra xem có phải ngăn xếp là rỗng;
2. Ghi nhớ địa chỉ của nút đầu hiện tại;
3. Ghi nhớ giá trị phần tử ở nút đầu;
4. Chuyển nút tiếp theo thành nút đầu mới (new top);
5. Giải phóng nút đầu cũ;
6. Trả lại giá trị phần tử ở nút đầu cũ.

### Cài đặt Pop trên C

```
float StackPop(Stack *s) {
 float item;
 StackNode *node;
 if (StackEmpty(s)) { // (1)
 // Empty Stack, can't pop
 return NULL;
 }
 node = s->top; // (2)
 item = node->item; // (3)
 s->top = node->next; // (4)
 free(node); // (5)
 return item; // (6)
}
```

### Chương trình thử nghiệm

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

struct StackNode {
 float item;
 StackNode *next;
};

struct Stack {
 StackNode *top;
};

Stack *StackConstruct();
void StackDestroy(Stack *s);
int StackEmpty(const Stack *s);
```

```
int StackFull(const Stack *s);
int StackPush(Stack *s, float *item);
float StackPop(Stack *s);

Stack *StackConstruct() {
 Stack *s;
 s = (Stack *)malloc(sizeof(Stack));
 if (s == NULL) {
 return NULL; // No memory
 }
 s->top = NULL;
 return s;
}

void StackDestroy(Stack *s) {
 while (!StackEmpty(s)) {
 StackPop(s);
 }
 free(s);
}

int StackEmpty(const Stack *s) {
 return (s->top == NULL);
}

int StackFull() {
 printf("\n NO MEMORY! STACK IS FULL"); getch();
 return 1;
}

int StackPush(Stack *s, float item) {
 StackNode *node;
 node = (StackNode *)malloc(sizeof(StackNode));
 if (node == NULL) {
```

```

 StackFull(); return 1;
}
node->item = item;
node->next = s->top;
s->top = node;
return 0;
}

float StackPop(Stack *s) {
 float item;
 StackNode *node;
 if (StackEmpty(s)) {
 // Empty Stack, can't pop
 return -100000;
 }
 node = s->top;
 item = node->item;
 s->top = node->next;
 free(node);
 return item;
}

void disp(Stack* s) {
 StackNode* node;
 int ct = 0; float m;

 printf("\n\n DANH SACH CAC PHAN TU CUA STACK \n\n");
 if (StackEmpty(s)) {
 printf("\n\n >>>> EMPTY STACK <<<<\n");
 getch();
 }
 else {
 node= s->top;
 do {

```

```

m=node->item;
printf("%8.3f ", m); ct++;
if (ct % 9 == 0) printf("\n");
node = node->next;
}
while (!(node == NULL));
printf("\n");
}

int main() {
int ch,n,i;
float m;
Stack* stackPtr;
while(1) {
printf("\n\n=====\\n");
printf("CHUONG TRINH THU STACK\\n");
printf("=====\\n");
printf(" 1.Create\\n 2.Push\\n 3.Pop\\n 4.Display\\n
5.Exit\\n");
printf("-----\\n");
printf("Chon chuc nang: ");
scanf("%d",&ch);
printf("\\n\\n");
switch(ch) {
case 1:
printf("Da khai tao STACK");
stackPtr = StackConstruct();
break;
case 2:
printf("Vao gia tri phan tu: ");
scanf("%f",&m);
printf("Gia tri phan tu nhap vao: %8.3f \\n ",m);
StackPush(stackPtr, m);
}
}
}

```

```

 break;

case 3:
 m=StackPop(stackPtr);
 if (m != -100000)
 (printf("\n\n Gia tri phan tu lay ra: %8.3f\n",m);
 getch());
 else {
 printf("\n\n 2 >>> Empty Stack, can't pop <<<\n");
 getch();
 }
 break;

case 4:
 disp(stackPtr);
 break;

case 5:
 printf("\n Bye! Bye! \n\n"); getch();
 exit(0);
 break;

default:
 printf("Wrong choice"); getch();
} //switch
) // end while
)

```

### 3.4.3. Một số ứng dụng của ngăn xếp

Các ứng dụng trực tiếp của ngăn xếp có thể kể đến là:

- Lịch sử duyệt trang trong trình duyệt Web;
- Dãy Undo trong bộ soạn thảo văn bản;
- Tổ chức chuỗi lệnh gọi;
- Kiểm tra tính hợp lệ của các dấu ngoặc trong biểu thức;
- Đổi cơ sở;
- Ứng dụng trong cài đặt chương trình dịch;
- Tính giá trị biểu thức;

- Quay lui;
- Khử đệ quy;
- ...

Các ứng dụng khác của ngăn xếp:

- Sử dụng như là cấu trúc dữ liệu hỗ trợ cho các thuật toán;
- Là thành phần của các cấu trúc dữ liệu khác.

Ta xét ứng dụng của ngăn xếp vào việc cài đặt thuật toán giải một số bài toán sau:

- Bài toán kiểm tra dấu ngoặc hợp lệ;
- Bài toán đổi cơ số;
- Bài toán tính giá trị biểu thức số học;
- Chuyển đổi biểu thức dạng trung tố về hậu tố;
- Khử đệ quy.

### **Ứng dụng 1: Bài toán biểu thức ngoặc hợp cách (Parentheses Matching)**

Mỗi “(”, “{” hoặc “[” phải cặp đôi với “)”, “}” hoặc “[”.

**Ví dụ:**

- correct: ( )( ) { ( [ ] ) }
- correct: ((( )( )) { ( [ ] ) })
- incorrect: )( )( ) { ( [ ] ) }
- incorrect: ( {[ ] } )
- incorrect: (

### **Thuật toán giải bài toán Parentheses Matching**

**Algorithm ParenMatch( $X, n$ ):**

**Input:** Mảng  $X$  gồm  $n$  ký hiệu, mỗi ký hiệu hoặc là dấu ngoặc, hoặc là biến, hoặc là phép toán số học, hoặc là con số.

**Output:** true khi và chỉ khi các dấu ngoặc trong  $Z$  có đôi.

$S$  = ngăn xếp rỗng;

**for**  $i=0$  to  $n-1$  **do**

**if** ( $X[i]$  là ký hiệu mở ngoặc)

        push( $S, X[i]$ ); // gấp dấu mở ngoặc thì đưa vào Stack

**else**

**if** ( $X[i]$  là ký hiệu đóng ngoặc) // gấp dấu đóng ngoặc thì so với dấu ở đầu

Stack

**if** isEmpty( $S$ )

**return false** {không tìm được cặp đôi}

```

if (pop(S) không đi cặp với dấu ngoặc trong X[i])
 return false {lỗi kiểu dấu ngoặc}

```

**if isEmpty(S)**

```
return true {mỗi dấu ngoặc đều có cặp}
```

```
else return false {có dấu ngoặc không tìm được cặp}
```

### Ứng dụng 2: Kiểm tra sánh đôi thẻ trong HTML (HTML Tag Matching)

Trong HTML, mỗi thẻ `<name>` phải đi cặp với thẻ `</name>`

#### Ví dụ:

```

<body>
<center>
<h1>The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>

 Will the salesman die?
 What color is the boat?
 And what about Naomi?

</body>

```

The Little Boat

The storm tossed the little boat  
like a cheap sneaker in an old  
washing machine. The three  
drunken fishermen were used to  
such treatment, of course, but not  
the tree salesman, who even as  
a stowaway now felt that he had  
overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Thuật toán giải bài toán kiểm tra tính sánh đôi thẻ trong HTML hoàn toàn tương tự như thuật toán giải bài toán ngoặc hợp lệ. Việc thiết kế và cài đặt chương trình giải bài toán đặt ra được coi là bài tập.

### Ứng dụng 3: Bài toán đổi cơ số

Bài toán: Viết một số trong hệ đếm thập phân thành số trong hệ đếm cơ số  $b$ .

#### Ví dụ:

$$(\text{cơ số } 8) \quad 28_{10} = 3 \cdot 8 + 4 = 34_8$$

$$(\text{cơ số } 4) \quad 72_{10} = 1 \cdot 64 + 0 \cdot 16 + 2 \cdot 4 + 0 = 1020_4$$

$$(\text{cơ số } 2) \quad 53_{10} = 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 = 110101_2$$

#### Thuật toán dùng ngăn xếp

**Input:** số trong hệ đếm thập phân  $n$ .

**Output:** số trong hệ đếm cơ số  $b$  tương ứng.

1. Chữ số phải nhất của  $n$  là  $n \% b$ . Nạp chữ số này vào stack.
2. Thay  $n$  bởi  $n/b$  (để tiếp tục xác định các chữ số còn lại).

3. Lặp lại các bước 1 – 2 đến khi còn số 0 ( $n/b = 0$ ).

4. Đẩy các ký tự ra khỏi ngăn xếp và in chúng.

**Ví dụ:**

|             |               |              |              |              |                   |
|-------------|---------------|--------------|--------------|--------------|-------------------|
|             |               |              |              |              | 6741 <sub>8</sub> |
| Empty stack | $n \% 8 = 1$  | $n \% 8 = 4$ | $n \% 8 = 7$ | $n \% 8 = 6$ |                   |
| $n = 3553$  | $n / 8 = 444$ | $n / 8 = 55$ | $n / 8 = 6$  | $n / 8 = 0$  |                   |
|             | $n = 444$     | $n = 55$     | $n = 6$      | $n = 0$      |                   |

#### Ứng dụng 4: Bài toán tính giá trị biểu thức số học

Xét việc tính giá trị của biểu thức số học trong đó có các phép toán hai ngôi: cộng, trừ, nhân, chia, luỹ thừa giữa các toán hạng (gọi là biểu thức số học trong ký pháp trung tố – infix notation). Thông thường, đối với biểu thức trong ký pháp trung tố, trình tự thực hiện tính biểu thức được chỉ ra bởi các cặp dấu ngoặc hoặc theo thứ tự ưu tiên của các phép toán. Vào năm 1920, Łukasiewicz (nhà toán học Ba Lan) đã đề xuất ký pháp Ba Lan cho phép không cần sử dụng các dấu ngoặc mà vẫn xác định được trình tự thực hiện các phép toán trong biểu thức số học.



Jan Łukasiewicz (1878 – 1956)

#### Ký pháp trung tố (Infix Notation)

- Mỗi phép toán hai ngôi được đặt giữa các toán hạng.
- Mỗi phép toán một ngôi (unary operator) đi ngay trước toán hạng.

**Ví dụ**

$$-2 + 3 * 5 \Leftrightarrow (-2) + (3 * 5)$$

Việc tính giá trị của biểu thức trung tố sẽ được thực hiện nhờ sử dụng hai ngăn xếp có kiểu dữ liệu khác nhau:

- Một ngăn xếp để giữ các toán hạng;
- Ngăn xếp kia giữ các phép toán.

Chúng ta sẽ thấy thuật toán đó sau khi xét thuật toán tính giá trị của biểu thức hậu tố và thuật toán chuyển biểu thức dạng trung tố về dạng hậu tố.

## Ký pháp hậu tố (Postfix Notation)

Còn được gọi là ký pháp đảo Ba Lan (Reverse Polish Notation, viết tắt là RPN), trong đó các toán hạng được đặt trước các phép toán. Chẳng hạn, biểu thức không cần dấu ngoặc trong RPN:

$$ab*c+$$

là tương đương với ký pháp trung tố:

$$a*b + c$$

### Ví dụ

| <i>infix</i>                                  | <i>postfix</i>                              |
|-----------------------------------------------|---------------------------------------------|
| $a*b*c*d*e*f$                                 | $ab*c*d*e*f*$                               |
| $1 + (-5) / (6 * (7+8))$                      | $1\ 5\ -\ 6\ 7\ 8\ +\ *\ /$                 |
| $(x/y - a*b) * ((b+x) - y)^y$                 | $x\ y\ /\ a\ b\ * -\ b\ x\ +\ y\ y\ ^ -\ *$ |
| $(x*y*z - x^2 / (y^2 - z^3) + 1/z) * (x - y)$ | $xy*z*x2^y2*z3^-/-1z/+xy - *$               |

### Tính giá trị biểu thức hậu tố

Thuật toán tính giá trị biểu thức hậu tố mô tả dưới đây sẽ sử dụng *ngăn xếp toán hạng*.

Giả sử ta có xâu gồm các toán hạng và các phép toán.

1. Duyệt biểu thức từ trái sang phải.
2. Nếu gặp toán hạng thì nạp (push) giá trị của nó vào ngăn xếp.
3. Nếu gặp phép toán thì thực hiện phép toán này với hai toán hạng được lấy ra (pop) từ ngăn xếp.
4. Nạp (push) giá trị tính được vào ngăn xếp (như vậy, ba ký hiệu được thay bởi một toán hạng).
5. Tiếp tục duyệt cho đến khi trong ngăn xếp chỉ còn một giá trị duy nhất – chính là kết quả của biểu thức.

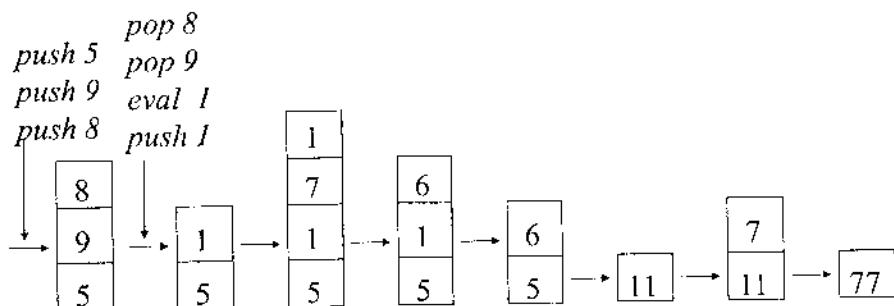
Thời gian tính là  $O(n)$  vì mỗi toán hạng và mỗi phép toán được duyệt qua đúng một lần.

### Ví dụ

Dãy đầu vào:

$$\begin{aligned} & 5\ 9\ 8 - 7\ 1 - * + 7\ * \\ & = 5(9 - 8)(7 - 1)* + 7* \end{aligned}$$

$$\begin{aligned}
 &= 5 ((9 - 8) * (7 - 1)) + 7 * \\
 &= (5 + ((9 - 8) * (7 - 1))) 7 * \\
 &= (5 + ((9 - 8) * (7 - 1))) * 7
 \end{aligned}$$



Thuật toán có thể mô tả hình thức hơn trong đoạn giả mã sau:

```

Tạo ngăn xếp rỗng S;
while (dòng vào khác rỗng){
 token = <phân tử tiếp theo của biểu thức>;
 if (token là toán hạng){
 push(S,token);
 }
 else if (token là phép toán){
 op2 = pop(S);
 op1 = pop(S);
 result = calc(token, op1, op2);
 push(S,result);
 }
} //end of while
return pop(S);

```

**Ví dụ:** Cài đặt PostfixCalcul tính giá trị biểu thức hậu tố đơn giản.

**Đầu vào:** Xâu chứa biểu thức hậu tố có độ dài không quá 80 ký tự. Các toán hạng và phép toán phân cách nhau bởi đúng một dấu cách.

**Kết quả:** Dưa ra giá trị của biểu thức.

**Hạn chế:** Toán hạng được giả thiết là: số nguyên không âm có một chữ số. Phép toán chỉ có: +, -, \*.

```
#include <stdio.h>
#include <stdlib.h>
```

```

#include <string.h>
#include <conio.h>
int stack [1000];
int top;

// Tinh giá trị của biểu thức
int eval (char *s);
// Kiểm tra có phải phép toán
int isop (char op);
// Thao tác đẩy ra của ngăn xếp
int pop (void);
// Thao tác đẩy vào của ngăn xếp
void push (int a);
// Thực hiện phép toán
int do_op (int a, int b, char op);

int main (void)
{char expression[80];
 int value;

 printf ("Nhập vào xau bieu thuc: ");
 gets(expression);
 printf ("\nBieu thuc nhap vao: %s", expression);
 value=eval (expression);
 printf ("\nGia tri cua bieu thuc = %i", value);
 getch();
 return 0;
}

int eval (char *s){
 char *ptr;
 int first, second, c;
 ptr = strtok (s, " ");
 top = -1;
 while (ptr) {
 if (isop (*ptr)) {

```

```

 second = pop(); first = pop();
 c = do_op (first, second, *ptr);
 push(c);
 }
 else { c = atoi(ptr); push(c);
 }
 ptr = strtok (NULL, " ");
}
return (pop ());
}

int do_op (int a, int b, char op)
{
 int ans;
 switch (op) {
 case '+':
 ans = a + b;
 break;
 case '-':
 ans = a - b;
 break;
 case '*':
 ans = a * b;
 break;
 }
 return ans;
}

int pop (void){
 int ret;
 ret = stack [top];
 top--;
 return ret;
}

void push (int a){

```

```

 top++;
 stack [top] = a;
}

int isop (char op){
 if (op == '+' || op == '-' || op == '*')
 return 1;
 else
 return 0;
}

```

### Chuyển biểu thức dạng trung tố về dạng hậu tố – (Infix to Postfix Conversion)

Để có thể tính được dễ dàng biểu thức trung tố, ta xét thuật toán chuyển đổi biểu thức dạng trung tố về dạng hậu tố.

Ta xét cách chuyển đổi biểu thức trung tố với các phép toán *cộng, trừ, nhân, chia, luỹ thừa và các dấu ngoặc* về dạng hậu tố.

Trước hết nhắc lại quy tắc tính giá trị biểu thức trung tố như sau:

- *Thứ tự ưu tiên*: Luỹ thừa; Nhân/Chia; Cộng/Trừ.
- *Quy tắc kết hợp*: Cho biết khi hai phép toán có cùng thứ tự ưu tiên thì cần thực hiện phép toán nào trước.

+ Luỹ thừa: phải qua trái. Ví dụ:  $2^2^3 = 2^{(2^3)} = 256$ .

+ Nhân/Chia: trái qua phải.

+ Cộng/Trừ: trái qua phải.

– *Dấu ngoặc* được ưu tiên hơn cả thứ tự ưu tiên và quy tắc kết hợp.

Thuật toán cơ bản là *operator precedence parsing*. Ta sẽ duyệt biểu thức từ trái qua phải. Khi gặp toán hạng, lập tức đưa nó ra. Khi gặp phép toán thì không thể đưa nó ra ngay được, vì toán hạng thứ hai còn chưa được xét. Vì thế, phép toán cần được cất giữ và sẽ được đưa ra đúng lúc. Sử dụng ngăn xếp để cất giữ phép toán đang xét nhưng còn chưa được đưa ra.

#### Ngăn xếp giữ phép toán

Xét biểu thức  $1 + 2 * 3 ^ 4 + 5$ , biểu thức hậu tố tương đương là  $1\ 2\ 3\ 4\ ^\ * \ +\ 5\ +$ .

Biểu thức  $1 * 2 + 3 ^ 4 + 5$  có biểu thức hậu tố tương đương là  $1\ 2\ *\ 3\ 4\ ^\ +\ 5\ +$ .

Trong cả hai trường hợp, khi phép toán thứ hai cần được xử lý thì trước đó chúng ta đã đưa ra  $1\ 2$  và có một phép toán đang nằm trong ngăn xếp. Câu hỏi là: *Các phép toán rời khỏi ngăn xếp như thế nào?*

Khi nào đưa phép toán ra khỏi ngăn xếp

– Phép toán rời khỏi ngăn xếp nếu các quy tắc về trình tự và kết hợp cho thấy nó cần được xử lý thay cho phép toán đang xét.

– Quy tắc cơ bản: Nếu phép toán đang xét có thứ tự ưu tiên thấp hơn so với phép toán ở đầu ngăn xếp, thì phép toán ở đầu ngăn xếp phải rời ngăn xếp.

$$\left| \begin{array}{c} 2 \\ 1 \end{array} \right| \left| \begin{array}{c} + \end{array} \right| * \Rightarrow \left| \begin{array}{c} 2 \\ 1 \end{array} \right| \left| \begin{array}{c} * \\ + \end{array} \right| \wedge$$

$$1 + 2 * 3 ^ 4 + 5 \qquad \qquad 1 + 2 * 3 ^ 4 + 5$$

$$\left| \begin{array}{c} 2 \\ 1 \end{array} \right| \left| \begin{array}{c} * \end{array} \right| + \Rightarrow \left| \begin{array}{c} * \\ 2 \\ 1 \end{array} \right| \left| \begin{array}{c} + \end{array} \right| \wedge$$

$$1 * 2 + 3 ^ 4 + 5 \qquad \qquad 1 * 2 + 3 ^ 4 + 5$$

## Cùng mức ưu tiên

Quy tắc kết hợp cho biết cần làm gì khi phép toán đang xét có cùng thứ tự ưu tiên với phép toán ở định ngăn xếp.

Nếu phép toán có tính kết hợp trái (left associative), thì phép toán ở định ngắn xếp cần đưa ra. Ví dụ, xét biểu thức trung tố “4 4 4” có dạng hậu tố là “4 4 – 4”.

Nếu phép toán có tính kết hợp phải (right associative), thì không đưa phép toán ở định ngăn xếp ra. Ví dụ, xét biểu thức trung tố “ $2^22^3$ ” có dạng hậu tố là “ $2\ 2\ 3\ ^\wedge\ ^\wedge$ ”.

4-4-4

$$\begin{array}{c} | & | & | \\ 2 & | & | \\ 2 & & \end{array} \xrightarrow{\hspace{1cm}} \begin{array}{c} | & | & | \\ 2 & | & | \\ 2 & & \end{array}$$

$\wedge$

$2^2 2^3$                      $2^2 2^3$

Có tình huống khi hàng loạt phép toán rời ngăn xếp. Xét biểu thức “ $1 + 2 * 3 ^ 4 + 5$ ”, với biểu thức hậu tố tương đương là “ $1\ 2\ 3\ 4\ ^\ * +\ 5\ +$ ”.

4  
3  
2  
1

+

Khi gấp phép toán + thứ hai, các phép toán  $\wedge$  \* + lần lượt được đưa ra khỏi ngăn xếp.

Như vậy, có thể xảy ra tình huống hàng loạt phép toán rời ngăn xếp đối với cùng một phép toán đang xét.

Ta cần xử lý dấu ngoặc như thế nào? Dấu mở ngoặc có thứ tự ưu tiên hơn phép toán khi nó được xem như là *ký tự dấu vào* (*input symbol*) (nghĩa là không có gì rời khỏi ngăn xếp). Dấu mở ngoặc có thứ tự ưu tiên thấp hơn phép toán khi nó ở *ngăn xếp*. Dấu đóng ngoặc sẽ đẩy phép toán ra khỏi ngăn xếp cho đến khi gấp dấu mở ngoặc rời khỏi ngăn xếp. Các phép toán sẽ được ghi ra còn các dấu ngoặc thì không được ghi ra.

### Thuật toán

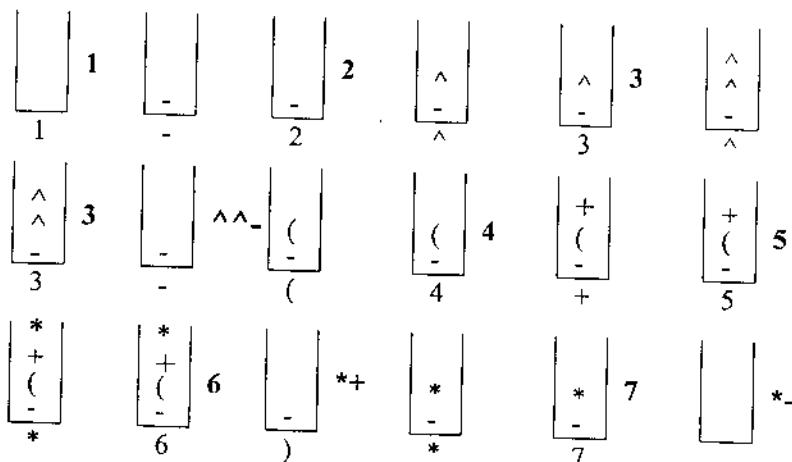
Duyệt biểu thức từ trái qua phải:

- Nếu gấp *toán hạng* (*Operands*): đưa ra tức thì.
- Nếu gấp *dấu mở ngoặc*: nạp nó vào ngăn xếp.
- Nếu gấp *dấu đóng ngoặc*: đẩy ký hiệu ra khỏi ngăn xếp cho đến khi gấp dấu mở ngoặc đầu tiên được đẩy ra.
  - Nếu gấp *phép toán* (*Operator*): đưa ra khỏi ngăn xếp tất cả các phép toán cho đến khi gấp phép toán có thứ tự ưu tiên thấp hơn hoặc gấp phép toán có tính kết hợp phải có cùng thứ tự ưu tiên, sau đó nạp phép toán đang xét vào ngăn xếp.
  - Khi duyệt hết biểu thức: đưa tất cả các phép toán còn lại ra khỏi ngăn xếp.

**Ví dụ.** Chuyển đổi biểu thức dạng trung tố về dạng hậu tố.

Infix:  $1 - 2 \wedge 3 \wedge 3 - (4 + 5 * 6) * 7$

Postfix:  $1\ 2\ 3\ 3\ \wedge\ \wedge\ -\ 4\ 5\ 6\ *\ +\ 7\ *\ -$



## Ứng dụng 5: Ngăn xếp và đệ quy

Một trong những ứng dụng quan trọng của ngăn xếp là nó được sử dụng để tổ chức thực hiện các thuật toán đệ quy. Có thể nói:

- Mỗi hàm đệ quy đều có thể cài đặt sử dụng ngăn xếp và lặp.
- Mỗi hàm lặp có sử dụng ngăn xếp đều có thể cài đặt sử dụng đệ quy.

Như ví dụ minh họa, ta xét hàm tính luỹ thừa được cài đặt đệ quy.

**Bài toán:** Cho  $x$  là số nguyên và  $n$  là số nguyên dương, cần tính  $x^n = x*x*...*x$  (có  $n$  thừa số  $x$ ). Thuật toán tính  $x^n$  trực tiếp theo định nghĩa đòi hỏi  $n$  phép nhân.

Thuật toán đệ quy: Nhận thấy rằng, nếu  $n$  là số chẵn thì:

$$x^n = (x^{n/2})^2 = (x^{n/2}) \times (x^{n/2})$$

Công thức trên cho ta chuyển việc tính  $x^n$  về việc tính luỹ thừa giảm đi một nửa của  $x$  ( $x^{n/2}$ ) và sau đó thực hiện một phép nhân. Với  $n$  lẻ ta có thể viết  $x^n = x^{(n-1)/2} \times x^{(n-1)/2} \times x$  và chuyển về việc tính tích luỹ thừa chẵn của  $x$ . Như vậy nếu  $n$  chẵn ta chuyển đến luỹ thừa  $n/2$  và nếu  $n$  lẻ ta chuyển đến luỹ thừa  $(n-1)/2$ .

Rõ ràng, sau không quá  $2\log n$  lần chuyển, ta đi đến luỹ thừa  $n = 0$ . Từ đó ta có thuật toán đòi hỏi thời gian  $O(\log n)$ .

Hàm tính luỹ thừa được cài đặt đệ quy như sau:

```
double power(double x, int n) {
 double tmp = 1;
 if (n > 0)
 {
 tmp = power(x, n/2);
 if (n % 2 == 0)
 tmp = tmp*tmp;
 else
 tmp = tmp*tmp*x;
 }
 return tmp;
}
```

Để thấy được hoạt động của hàm này ta xét việc thực hiện **power(2,5)**:

|           |            |                |            |                         |
|-----------|------------|----------------|------------|-------------------------|
| Lần gọi 1 | power(2,5) | $x = 2, n = 5$ | power(2,5) | $x = 2, n = 5$          |
|           | ↓          | $temp = 1$     | ↑          | $temp = 4 * 4 * 2 = 32$ |
| Lần gọi 2 | power(2,2) | $x = 2, n = 2$ | power(2,2) | $x = 2, n = 2$          |
|           | ↓          | $temp = 1$     | ↑          | $temp = 2 * 2 = 4$      |
| Lần gọi 3 | power(2,1) | $x = 2, n = 1$ | power(2,1) | $x = 2, n = 1$          |
|           | ↓          | $temp = 1$     | ↑          | $temp = 1 * 1 * 2 = 2$  |
| Lần gọi 4 | power(2,0) | $x = 2, n = 0$ | power(2,0) | $x = 2, n = 0$          |
|           |            | $temp = 1$     |            | $temp = 1$              |
|           |            | <i>Tiến</i>    |            | <i>Lùi</i>              |

Để tổ chức thực hiện thuật toán đệ quy, thông thường người ta xây dựng ngăn xếp cát giữ trạng thái của các lần gọi đệ quy. Ta có thể theo dõi trạng thái của ngăn xếp được xây dựng để thực hiện lệnh gọi hàm power(2,5) như sau:

|           |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
|-----------|--------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---|---|--|
| Lần gọi 4 | $\left\{ \begin{array}{l} tmp \\ n \\ x \end{array} \right.$ | <table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="padding: 5px;">1</td></tr> <tr><td style="padding: 5px;">0</td></tr> <tr><td style="padding: 5px;">2</td></tr> <tr><td style="padding: 5px;"> </td></tr> </table> | 1 | 0 | 2 |  |
| 1         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| 0         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| 2         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
|           |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| Lần gọi 3 | $\left\{ \begin{array}{l} tmp \\ n \\ x \end{array} \right.$ | <table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="padding: 5px;">1</td></tr> <tr><td style="padding: 5px;">1</td></tr> <tr><td style="padding: 5px;">2</td></tr> <tr><td style="padding: 5px;"> </td></tr> </table> | 1 | 1 | 2 |  |
| 1         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| 1         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| 2         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
|           |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| Lần gọi 2 | $\left\{ \begin{array}{l} tmp \\ n \\ x \end{array} \right.$ | <table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="padding: 5px;">1</td></tr> <tr><td style="padding: 5px;">2</td></tr> <tr><td style="padding: 5px;">2</td></tr> <tr><td style="padding: 5px;"> </td></tr> </table> | 1 | 2 | 2 |  |
| 1         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| 2         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| 2         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
|           |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| Lần gọi 1 | $\left\{ \begin{array}{l} tmp \\ n \\ x \end{array} \right.$ | <table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="padding: 5px;">1</td></tr> <tr><td style="padding: 5px;">5</td></tr> <tr><td style="padding: 5px;">2</td></tr> <tr><td style="padding: 5px;"> </td></tr> </table> | 1 | 5 | 2 |  |
| 1         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| 5         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
| 2         |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |
|           |                                                              |                                                                                                                                                                                                                                                      |   |   |   |  |

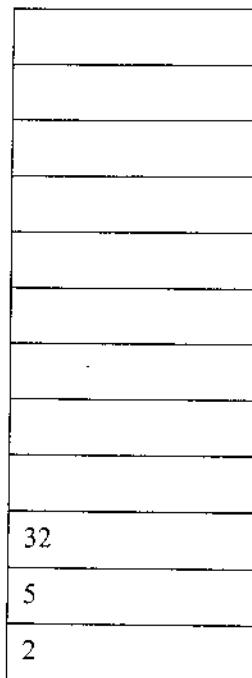
Ngăn xếp cho lần gọi 4

|                          |                                                                     |
|--------------------------|---------------------------------------------------------------------|
|                          |                                                                     |
|                          |                                                                     |
|                          |                                                                     |
|                          |                                                                     |
|                          |                                                                     |
| Trả lại cho<br>lần gọi 3 | $\left\{ \begin{array}{l} \text{tmp} \\ n \\ x \end{array} \right.$ |
| Lần gọi 2                | $\left\{ \begin{array}{l} \text{tmp} \\ n \\ x \end{array} \right.$ |
| Lần gọi 1                | $\left\{ \begin{array}{l} \text{tmp} \\ n \\ x \end{array} \right.$ |
|                          | 2                                                                   |
|                          | 1                                                                   |
|                          | 2                                                                   |
|                          | 1                                                                   |
|                          | 2                                                                   |
|                          | 2                                                                   |
|                          | 1                                                                   |
|                          | 5                                                                   |
|                          | 2                                                                   |

### *Ngăn xếp khi quay lại lần gọi 3*

### *Ngăn xếp khi quay lại lần gọi 2*

Trả lại cho  
lần gọi 1



### *Ngăn xếp khi quay lại lần gọi 1*

**Thuật toán lặp power(x,n) tính  $x^n$  sử dụng Stack**

module power(x, n)

{

create a Stack

initialize a Stack

*/\* Vòng lặp 1 \*/*

loop{

**if** ( $n == 0$ ) **then** { exit loop }

push n onto Stack

$$n = n/2$$

}

tmp = 1

*/\* Vòng lặp 2 \*/*

loop {

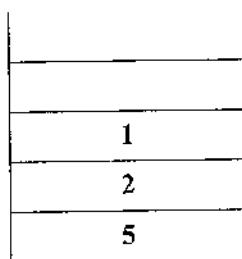
```

if (Stack is empty) then {return tmp}
pop n off Stack
if (n is even) {tmp = tmp*tmp}
else {tmp = tmp*tmp*x}
}
}

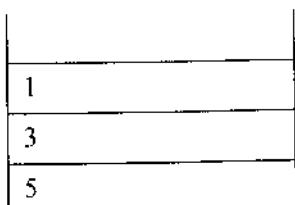
```

Ta có thể theo dõi việc thực hiện power(2,5):

– Trạng thái của Stack sau khi thực hiện vòng lặp 1:



– Trạng thái của Stack khi thực hiện vòng lặp 2:

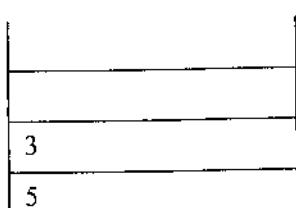


Stack

|            |   |
|------------|---|
| <i>tmp</i> | 1 |
| <i>n</i>   | 0 |
| <i>x</i>   | 2 |

Giá trị biến

### Đầu vòng lặp 2

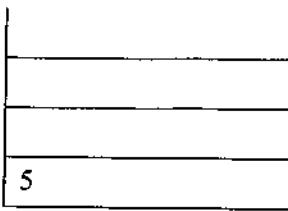


Stack

|            |   |
|------------|---|
| <i>tmp</i> | 2 |
| <i>n</i>   | 1 |
| <i>x</i>   | 2 |

Giá trị biến

### Sau lần lặp 1

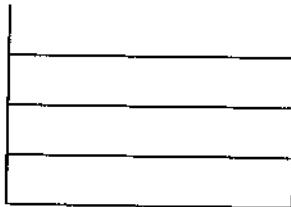


Stack

|            |   |
|------------|---|
| <i>tmp</i> | 4 |
| <i>n</i>   | 2 |
| <i>x</i>   | 2 |

Giá trị biến

Sau lần lặp 2



Stack

|            |    |
|------------|----|
| <i>tmp</i> | 32 |
| <i>n</i>   | 5  |
| <i>x</i>   | 2  |

Giá trị biến

Sau lần lặp 3

Cài đặt trên C có thể có dạng sau

```
double power(double x, int n){
 double tmp = 1;
 Stack theStack;
 initializeStack(&theStack);
 while (n != 0) {
 push(&theStack, n);
 n /= 2;
 }
 while (!stackEmpty(&theStack)) {
 n = pop(&theStack);
 if (n % 2 == 0) {
 tmp = tmp*tmp;
 }
 else {
 tmp = tmp*tmp*x;
 }
 }
 return tmp;
}
```

## 3.5. HÀNG ĐỢI

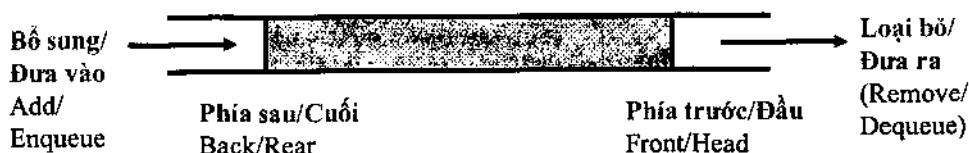
### 3.5.1. ADT hàng đợi

Hàng đợi là *danh sách có thứ tự* trong đó phép toán chèn luôn thực hiện chỉ ở một phía gọi là phía sau hay cuối (back or rear), còn phép toán xóa chỉ thực hiện ở phía còn lại gọi là phía trước hay đầu (front or head).

Thuật ngữ thường dùng cho hai thao tác chèn và xóa đối với hàng đợi tương ứng là đưa vào (enqueue) và đưa ra (dequeue).

Các phần tử được lấy ra khỏi hàng đợi theo quy tắc Vào trước – Ra trước (First-In–First–Out, viết tắt là FIFO). Vì thế hàng đợi còn được gọi là danh sách vào trước ra trước (FIFO list).

Các thuật ngữ liên quan đến hàng đợi được mô tả trong hình vẽ sau đây:



Hàng đợi có tính chất như các hàng đợi chờ được phục vụ trong thực tế.

#### Các phép toán

- $Q = \text{init}();$  Khởi tạo  $Q$  là hàng đợi rỗng.
- $\text{isEmpty}(Q);$  Trả lại "true" khi và chỉ khi hàng đợi  $Q$  là rỗng.
- $\text{isFull}(Q);$  Trả lại "true" khi và chỉ khi hàng đợi  $Q$  là tràn, cho biết ta đã sử dụng vượt quá kích thước tối đa dành cho hàng đợi.
- $\text{front}(Q);$  Trả lại phần tử ở phía trước (front) của hàng đợi  $Q$  hoặc gặp lỗi nếu hàng đợi rỗng.
- $\text{enqueue}(Q, x);$  Chèn phần tử  $x$  vào phía sau (back) hàng đợi  $Q$ . Nếu việc chèn dẫn đến tràn hàng đợi thì cần thông báo về điều này.
- $\text{dequeue}(Q, x);$  Xóa phần tử ở phía trước hàng đợi, trả lại  $x$  là thông tin chứa trong phần tử này. Nếu hàng đợi rỗng thì cần đưa ra thông báo lỗi.
- $\text{print}(Q);$  Đưa ra danh sách tất cả các phần tử của hàng đợi  $Q$  theo thứ tự từ phía trước đến phía sau.
- $\text{size}(Q);$  Trả lại số lượng phần tử trong hàng đợi  $Q$ .

#### Ví dụ

| Operation  | Output | Q      |
|------------|--------|--------|
| enqueue(5) | -      | (5)    |
| enqueue(3) | -      | (5, 3) |
| dequeue()  | 5      | (3)    |

|            |         |              |
|------------|---------|--------------|
| enqueue(7) | -       | (3, 7)       |
| dequeue()  | 3       | (7)          |
| front()    | 7       | (7)          |
| dequeue()  | 7       | 0            |
| dequeue()  | "error" | 0            |
| isEmpty()  | true    | 0            |
| size()     | 0       | 0            |
| enqueue(9) | -       | (9)          |
| enqueue(7) | -       | (9, 7)       |
| enqueue(3) | -       | (9, 7, 3)    |
| enqueue(5) | -       | (9, 7, 3, 5) |
| dequeue()  | 9       | (7, 3, 5)    |

### Ứng dụng của hàng đợi

#### Ứng dụng trực tiếp:

- Danh sách xếp hàng chờ mua vé tàu xe, chờ gửi xe, chờ được phục vụ ở hàng ăn, chờ mượn sách ở thư viện,...
- Chia sẻ các tài nguyên (ví dụ: máy in, CPU, bộ nhớ,...).
- Tổ chức thực hiện đa chương trình.
- ...

#### Ứng dụng gián tiếp:

- Cấu trúc dữ liệu hỗ trợ cho các thuật toán.
- Là thành phần của những cấu trúc dữ liệu khác.
- ...

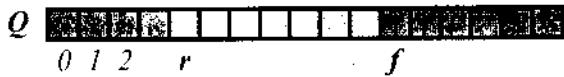
### 3.5.2. Cài đặt hàng đợi bằng mảng

Sử dụng mảng  $Q$  kích thước  $N$  theo thứ tự vòng tròn. Có hai biến để lưu trữ vị trí đầu và cuối (*front* and *rear*):

- $f$  chỉ số của phần tử ở đầu hàng đợi.
  - $r$  chỉ số của vị trí ở ngay sau vị trí phần tử cuối cùng của hàng đợi.
- Vị trí  $r$  được giữ là rỗng. Hình vẽ dưới đây mô tả hàng đợi:



Cấu hình bình thường



Cấu hình xoay vòng tròn

### Các phép toán đối với hàng đợi

Ta sử dụng phép toán số học theo modulo (phần dư của phép chia).

### Kích thước hàng đợi

#### Algorithm size()

```
return (N - f + r) mod N
```

#### Kiểm tra hàng đợi rỗng

#### Algorithm isEmpty()

```
return (f = r)
```

#### Phép toán chèn – Đưa vào (enqueue)

#### Algorithm enqueue(o)

```
if size() = N - 1 then
```

```
Error("FullQueue")
```

```
else
```

```
Q[r] ← o
```

```
r ← (r + 1) mod N
```

Phép toán enqueue phải đề ý đến lỗi tràn hàng đợi. Lỗi này cần được xử lý bởi người sử dụng.

#### Phép toán loại bỏ – Đưa ra (dequeue)

#### Algorithm dequeue()

```
if isEmpty() then
```

```
Error("EmptyQueue")
```

```
else
```

```
o ← Q[f]
```

```
f ← (f + 1) mod N
```

```
return o
```

Phép toán loại bỏ (dequeue) cần xử lý lỗi hàng đợi rỗng.

### 3.5.3. Cài đặt hàng đợi bởi danh sách mốc nối

Ta có thể cài đặt hàng đợi bởi danh sách mốc nối đơn hoặc đôi.

**Ví dụ:** Khai báo sau đây được sử dụng để mô tả hàng đợi bởi danh sách mốc nối đơn:

```

typedef struct _node {
 DataType element;
 struct _node *next; } node;

```

```

typedef struct { node *front; node *back; } queue;

```

Trong đó: DataType là kiểu dữ liệu của đối tượng cần lưu giữ, được khai báo trước.

Các phép toán đối với hàng đợi mô tả bởi danh sách mốc nối được cài đặt tương tự như đối với danh sách mốc nối đã trình bày ở trên. Ta sẽ không trình bày lại chi tiết ở đây. Chương trình sau đây minh họa cho việc cài đặt các phép toán cơ bản đối với hàng đợi.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Khai báo cấu trúc: Thông tin đi kèm trong mỗi nút là
// Họ tên và tuổi
typedef struct NODE * pNODE;
typedef struct NODE {
 char name[30+1];
 int age;
 pNODE link;
}NODE;

// Các hàm thực hiện các thao tác cơ bản
void enqueue(pNODE *front, pNODE *rear,pNODE *newNode);
pNODE dequeue(pNODE *front, pNODE *rear);

// Các hàm hỗ trợ minh họa cho Enqueue và Dequeue
void addQueue(pNODE *front, pNODE *rear);
void deleteQueue(pNODE *front, pNODE *rear);

// Hàm in ra tất cả các phần tử trong Queue
void printQueue(pNODE front, pNODE rear);

// Các hàm hỗ trợ nhập thông tin

```

```

int readInt(void);
double readDouble(void);
char *readString(char dest[], int len);

int main(void)
{
 /* Khởi tạo queue*/
 pNODE front=NULL, rear=NULL;

 char choice;

 do
 {
 printf("\n\n*****\n");
 printf(" DEMO QUEUE\n");
 printf("*****\n");
 printf("\n e = Enqueue to the queue, ");
 printf("\n d = Dequeue from the queue ");
 printf("\n p = Print the queue");
 printf("\n q = Exit ");
 fflush(stdin);
 printf("\n\n Go chu cai de chon chuc nang: ");
 choice=getchar();
 fflush(stdin);
 switch(choice) {
 case 'e': addQueue(&front,&rear);
 break;
 case 'd': deleteQueue(&front,&rear);
 break;
 case 'p': printQueue(front, rear);
 break;
 }
 } while (choice!='q');
 return 0;
}

```

```

void enqueue(pNODE *front, pNODE *rear,pNODE *newNode)
{
 if(*rear)
 {
 (*rear)->link=*newNode;
 *rear=*newNode;
 }
 else
 {
 *rear=*newNode;
 *front=*newNode;
 }
}

pNODE dequeue(pNODE *front, pNODE *rear)
{
 pNODE ptr=*front;
 if(*front)
 {
 *front=(*front)->link;
 if(*front==NULL)
 *rear=NULL;
 ptr->link=NULL;
 }
 return ptr;
}

void addQueue(pNODE *front, pNODE *rear)
{
 int age;
 char name[30+1]="";
 pNODE ptr;

 // Nạp thông tin cho một nút mới
}

```

```

while (strcmp(name,"") !=0)
{
 printf("\nGo ho ten sv: \n(Go <ENTER> da ket thuc nhap)");
 readString(name,30);

 if (strcmp(name,"") !=0)
 {
 printf("\nGo tuoi cua sinh vien:\n");
 age=readInt();

 ptr=new(NODE);
 strcpy(ptr->name,name);
 ptr->age=age;
 ptr->link=NULL;

 enqueue(front,rear,&ptr);
 }
}

void deleteQueue(pNODE *front, pNODE *rear)
{
 pNODE ptr=dequeue(front, rear);
 if (ptr) {
 printf("\nSinh vien %s, Tuoi %i, bi loai khoi
queue\n",ptr->name, ptr->age);
 delete ptr;
 ptr=NULL;
 }
 else printf("\nException! The stack is empty.\n");
}

void printQueue(pNODE front, pNODE rear)
{
 pNODE ptr;

```

```
if(front) {
 printf("\nThe following persons are in the
queue:\n");
 for (ptr=front;ptr!=NULL;ptr=ptr->link)
 {
 printf("\n%-20s %2d v.",ptr->name, ptr->age);
 }
 printf("\n");
}
else
 printf("The queue is empty\n");
}

int readInt()
{
 int number;

 while (scanf("%d", &number)!=1) {
 printf("\n*** Go vao so nguyen.");
 fflush(stdin);
 }
 fflush(stdin);
 return number;
}

double readDouble(void)
{
 double number;

 while (scanf("%lf", &number)!=1) {
 printf("\n*** Give a float type, please. ");
 fflush(stdin);
 }
 fflush(stdin);
 return number;
}
```

```

}

char *readString(char dest[], int len)
{
 if (fgets(dest, len, stdin) == 0) /* error in
reading*/
 dest[0]='\0';
 else if (dest[strlen(dest)-1] == '\n')
 dest[strlen(dest)-1]='\0';
 fflush(stdin);
 return dest;
}

```

### 3.5.4. Một số ví dụ ứng dụng hàng đợi

#### Ứng dụng 1: Chuyển đổi xâu chữ số thành số thập phân

Thuật toán được mô tả trong sơ đồ sau:

```

// Chuyển dãy chữ số trong Q thành số thập phân n
// Loại bỏ các dấu cách ở đầu (nếu có)

do {
 dequeue(Q, ch)
}
until (ch != blank)
// ch chứa chữ số đầu tiên
// Tính n từ dãy chữ số trong hàng đợi
n = 0; done = false;
do {
 n = 10 * n + số nguyên mà ch biểu diễn
 if (!isEmpty(Q))
 dequeue(Q, ch)
 else
 done = true
} until (done || ch != digit)
// Kết quả: n chứa số cần tìm

```

#### Ứng dụng 2: Nhận biết Palindrome

Ta gọi *palindrome* là xâu mà khi đọc nó từ trái qua phải cũng giống như đọc nó từ phải qua trái.

**Ví dụ:** Các xâu sau đây là *palindrome*:

- “NOON, DEED, RADAR, MADAM”
- “ABLE WAS I ERE I SAW ELBA”

Một trong những cách nhận biết một xâu cho trước có phải là *palindrome* hay không là ta đưa các ký tự của nó đồng thời vào một hàng đợi và một ngăn xếp. Sau đó lần lượt loại bỏ các ký tự khỏi hàng đợi và ngăn xếp rồi tiến hành so sánh:

- Nếu phát hiện sự khác nhau giữa hai ký tự, một ký tự được lấy ra từ ngăn xếp còn ký tự kia lấy ra từ hàng đợi, thì xâu đang xét không là *palindrome*.
- Nếu tất cả các cặp ký tự lấy ra trùng nhau thì xâu đang xét là *palindrome*.

**Ví dụ:** Xét hoạt động của thuật toán với xâu đầu vào là "RADAR".

Bước 1: Đưa “RADAR” vào Queue và Stack:

| Ký tự hiện thời | Queue<br>(tail ở bên phải) | Stack<br>(top ở bên trái) |
|-----------------|----------------------------|---------------------------|
| R               | R                          | R                         |
| A               | R A                        | A R                       |
| D               | R A D                      | D A R                     |
| A               | R A D A                    | A D A R                   |
| R               | R A D A R                  | R A D A R                 |

Bước 2: Xóa bỏ “RADAR” khỏi Queue và Stack:

| Queue<br>(head ở bên trái) | head của<br>Queue | top của<br>Stack | Stack<br>(top ở bên trái) |
|----------------------------|-------------------|------------------|---------------------------|
| R A D A R                  | R                 | R                | R A D A R                 |
| A D A R                    | A                 | A                | A D A R                   |
| D A R                      | D                 | D                | D A R                     |
| A R                        | A                 | A                | A R                       |
| R                          | R                 | R                | R                         |
| empty                      | empty             | empty            | empty                     |

Kết luận: Xâu "RADAR" là *palindrome*.

## BÀI TẬP CHƯƠNG 3

1. Xét cấu trúc dữ liệu mô tả danh sách nối đơn:

```
struct Node{
 int Inf;
 struct Node *Next; };
typedef struct Node LIST;
```

Hãy viết chương trình con trên C:

```
LIST * OddList(LIST * Linp);
```

nhận đầu vào là danh sách Linp, trả lại danh sách thu được từ danh sách đầu vào bởi việc loại bỏ các phần tử với trường inf là số chẵn.

2. Xét cấu trúc dữ liệu trên C để mô tả đa thức một biến:

```
struct PolyNode {
 int coeff; // chứa hệ số
 int pow; // chứa số mũ
 struct PolyNode *link;
}
typedef struct PolyNode Polynom;
```

Hãy viết hàm trên C:

```
Polynom * PolySum(Polynom* Poly1, Poly2);
```

nhận đầu vào là hai đa thức Poly1 và Poly2, tính và trả lại tổng của hai đa thức đã cho.

3. Xét cấu trúc dữ liệu mô tả danh sách nối đơn:

```
struct Node{
 int Inf;
 struct Node *Next; };
typedef struct Node LIST;
```

Cho dãy số nguyên được cất giữ dưới dạng danh sách nối đơn:

```
LIST * NumList;
```

Hãy viết chương trình con trên C:

```
void OddEven();
```

đếm xem trong dãy đã cho có bao nhiêu số chẵn, bao nhiêu số lẻ và đưa kết quả ra màn hình.

4. Xét cấu trúc dữ liệu trên C để mô tả đa thức một biến:

```
struct PolyNode {
 int coeff; // chứa hệ số
 int pow; // chứa số mũ
 struct PolyNode *link;
}
typedef struct PolyNode Polynom;
```

Hãy viết hàm trên C:

```
Polynom * PolyMult(Polynom* Poly1, Polynom* Poly2);
```

nhận đầu vào là hai đa thức *Poly1* và *Poly2*, tính và trả lại tích của hai đa thức đã cho.

5. Xét cấu trúc dữ liệu mô tả danh sách nối đơn:

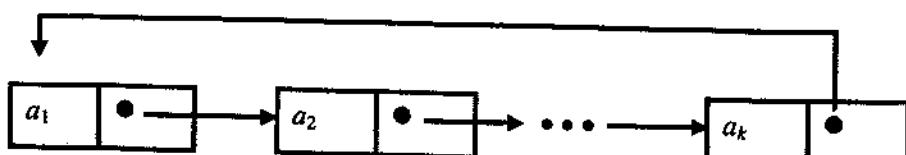
```
struct Node{
 int Inf;
 struct Node *Next; };
typedef struct Node LIST;
```

Hãy viết chương trình con trên C:

```
LIST * OddList(LIST * Linp);
```

nhận đầu vào là danh sách nối đơn với phần tử đầu tiên được trả bởi Linp, thực hiện việc loại bỏ tất cả các phần tử xuất hiện ở vị trí lẻ của danh sách rồi trả lại con trỏ đến đầu danh sách thu được. Giả thiết các phần tử trong danh sách được đánh số từ 1 bắt đầu từ phần tử ở đầu danh sách.

6. Danh sách nối vòng là danh sách với cách tổ chức được mô tả như trong hình vẽ dưới đây:



Xét cấu trúc dữ liệu mô tả danh sách nối vòng sau đây:

```
typedef struct Node {
 int data;
 struct Node* next;
} Node;
```

Hãy viết hàm trên C:

```
int Count(Node* ptr)
```

nhận đầu vào là con trỏ **ptr** trỏ đến một nút bất kỳ trong một danh sách nối vòng, trả lại số lượng nút trong danh sách. Giả thiết dữ liệu cất giữ trong các nút của danh sách là đôi một khác nhau.

7. Cho danh sách mốc nối được mô tả bởi cấu trúc dữ liệu:

```
struct ListNode {
 int info;
 struct ListNode * next; };
```

Giả thiết rằng các nút trong danh sách được sắp xếp theo thứ tự không giảm của trường **info**.

Viết hàm trên C:

```
ListNode * removeDuplicates(ListNode * list)
```

loại bỏ những phần tử lặp lại khỏi danh sách với phần tử đầu tiên được trả bởi **list**, trả lại con trỏ đến phần tử đầu tiên của danh sách thu được.

Ví dụ, nếu dãy số trong trường **info** của danh sách là (2, 2, 3, 3, 3, 5, 5, 5, 5, 6, 7) thì **removeDuplicates(list)** phải trả lại con trỏ đến phần tử đầu tiên của danh sách với dãy số tương ứng là (2, 3, 5, 6, 7).

8. Trong bài này chúng ta xét danh sách mốc nối mô tả danh sách các lớp sinh viên, mỗi lớp được biểu diễn bởi **ClassNode**; mỗi **ClassNode** chứa con trỏ đến danh sách các sinh viên trong lớp, thông tin về mỗi sinh viên được biểu diễn bởi **StudentNode**. **ClassNode** và **StudentNode** được định nghĩa như sau:

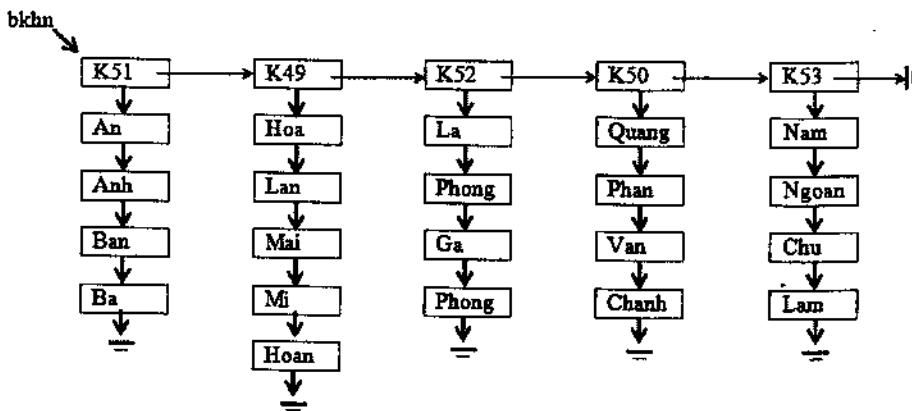
```
struct ClassNode
{
 string name; // tên lớp
 ClassNode * next; // con trỏ đến lớp tiếp theo trong
 // danh sách
 StudentNode * students; // danh sách sinh viên
};
```

```

struct StudentNode
{
 string name; // tên sinh viên
 StudentNode * next; // con trỏ đến sinh viên tiếp theo
};

```

**Ví dụ:** Trong ví dụ dưới đây ta có bkhn là con trỏ đến danh sách ClassNode gồm năm phần tử: K51, K49, K52, K50, K53; mỗi ClassNode lại chứa con trỏ đến danh sách sinh viên trong lớp.



a) Hãy viết hàm:

```
int NumStudents(ClassNode * ClPoint)
```

nhận đầu vào là con trỏ ClPoint, trả lại số sinh viên trong lớp được trả bởi ClPoint. Trong ví dụ nêu trên, NumStudents(bkhn->next->next) trả lại giá trị 4 (là số lượng sinh viên trong lớp K52).

b) Hãy viết hàm:

```
int NumClassStudents(ClassNode * list, string CName)
```

trả lại số lượng sinh viên trong lớp với tên lớp được cho trong CName. Xét ví dụ nêu ở trên, khi đó:

```
NumClassStudents(bkhn, "K49")
```

trả lại giá trị 5, còn:

```
NumClassStudents(bkhn, "K48")
```

trả lại giá trị 0 (vì lớp với tên "K48" không có trong danh sách).

9. Xét danh sách nối đôi được mô tả bởi cấu trúc dữ liệu:

```
struct dllNode
{
 int info;
 dllNode * next; // trỏ đến nút kế tiếp
 dllNode * prev; // trỏ đến nút đi trước
};
```

Viết chương trình con:

```
void Remove(dllNode * dlist, int key);
```

loại bỏ tất cả các phần tử có trường **info** bằng **key** khỏi danh sách với phần tử đầu tiên được trả bởi **dlist**.

10. Xét danh sách mốc nối được mô tả bởi cấu trúc dữ liệu

```
struct Node {
 int info;
 struct Node * next; // trỏ đến nút kế tiếp
};
```

Viết chương trình con:

```
void Shuffle(Node * list);
```

nhận đầu vào là danh sách mốc nối với phần tử đầu tiên được trả bởi **list**, thực hiện các việc sau đây:

- a) Sắp xếp lại các phần tử của danh sách đã cho sao cho: các nút chẵn phải đứng trước các nút lẻ và trong trường hợp ngược lại, thứ tự tương đối ban đầu của các nút là không thay đổi. Một nút được gọi là nút chẵn hay lẻ nếu nó đứng ở vị trí chẵn hay lẻ trong danh sách (vị trí của các nút trong danh sách được đánh số từ phần tử đầu tiên đến phần tử cuối cùng bắt đầu từ 1).

- b) Đưa ra màn hình danh sách thu được.

Ví dụ, nếu danh sách đã cho là (11, 13, 7, 9, 3, 10) thì kết quả phải đưa ra là (13, 9, 10, 11, 7, 3).

11. Xét danh sách nối đôi được mô tả bởi cấu trúc dữ liệu:

```
typedef struct dllNode
{
 int info;
 struct dllNode * next; // trỏ đến nút kế tiếp
 struct dllNode * prev; // trỏ đến nút đi trước
} dllNode;
```

Viết chương trình con trên C:

```
void Remove(dllNode * dlist, int key);
```

loại bỏ tất cả các phần tử có trường **info** bằng **key** khỏi danh sách với phần tử đầu tiên được trả bởi **dlist** và đưa ra màn hình danh sách thu được:

12. Xét danh sách nối đôi được mô tả bởi cấu trúc dữ liệu:

```
struct dllNode
{
 int info;
 dllNode * next; // trỏ đến nút kế tiếp
 dllNode * prev; // trỏ đến nút đi trước
};
```

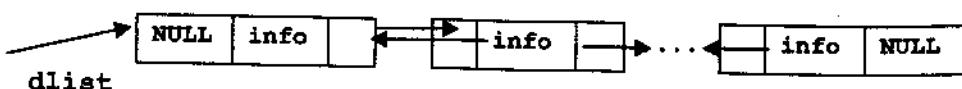
Viết chương trình con trên C:

```
void Remove(dllNode * dlist, int key);
```

loại bỏ tất cả các phần tử có trường **info** là bằng **key** khỏi danh sách với phần tử đầu tiên được trả bởi **dlist** và đưa ra màn hình danh sách thu được.

13. Xét danh sách nối đôi được mô tả bởi cấu trúc dữ liệu:

```
struct dllNode
{ int info;
 dllNode * next; // trỏ đến nút kế tiếp
 dllNode * prev; // trỏ đến nút đi trước
};
```



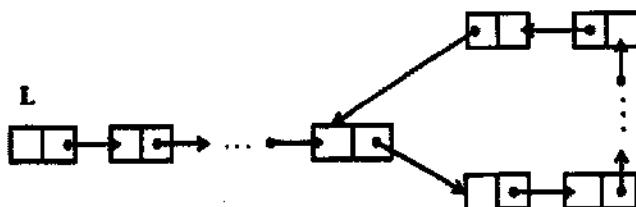
Viết chương trình con trên C:

```
void Remove(dllNode * dlist, int key);
```

loại bỏ tất cả các phần tử có trường **info** bằng **key** khỏi danh sách với phần tử đầu tiên được trả bởi **dlist** và đưa ra màn hình danh sách thu được.

14. Ta định nghĩa danh sách mốc nối bị quản là danh sách mốc nối mà trong đó con trỏ của phần tử cuối cùng không phải là con trỏ rỗng mà lại trỏ đến một phần tử nào đó trong danh sách (xem hình 1).

Cho danh sách mốc nối L. Hãy mô tả thuật toán (trong ngôn ngữ mô phỏng) xác định xem danh sách đã cho có bị quản hay không. Thuật toán đề xuất không được phép biến đổi danh sách đã cho, phải có thời gian tính là  $O(n)$  và sử dụng bộ nhớ phụ là  $O(1)$ , trong đó  $n$  là số phần tử của danh sách.



Hình 1. Danh sách mốc nối bị quản

15. Giả sử **push(a)** là thao tác nạp  $a$  vào ngăn xếp và **pop()** là thao tác lấy phần tử ra khỏi ngăn xếp. Xét việc thực hiện dãy thao tác sau đây đối với ngăn xếp ban đầu được khởi tạo rỗng:

**push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop()**.

Hãy đưa ra dãy phần tử của ngăn xếp (chỉ rõ vị trí đầu ngăn xếp) sau khi thực hiện mỗi thao tác.

16. Giả sử **enq(a)** là thao tác nạp  $a$  vào hàng đợi và **deq()** là thao tác lấy phần tử ra khỏi hàng đợi. Xét dãy thao tác sau đây đối với hàng đợi ban đầu được khởi tạo rỗng:

**enq(5), enq(3), deq(), enq(2), enq(8), deq(), deq(), enq(9), enq(1), deq(), enq(7), enq(6), deq(), deq(), enq(4), deq(), deq()**.

Hãy đưa ra dãy phần tử của hàng đợi (chỉ rõ vị trí đầu và cuối của hàng đợi) sau khi thực hiện mỗi thao tác.

17. Đối với mỗi một trong các kiểu cấu trúc dữ liệu sau đây: danh sách nối đơn (Singly-linked list); danh sách nối kép (Doubly-linked list); ngăn xếp dùng mảng (Stack); hàng đợi dùng mảng (Queue), hãy vẽ cấu trúc dữ liệu thu được sau khi lần lượt bổ sung các phần tử của dãy các khóa 4, 2, 6, 7, 6, 5 theo thứ tự trong dãy vào các cấu trúc này (bắt đầu từ cấu trúc rỗng).
18. Hãy trình diễn cách sử dụng ngăn xếp để chuyển biểu thức dạng trung tố sau đây về dạng biểu thức hậu tố:
- $a - b * c ^ d + f$
  - $1 + 2 + 3 * 4 + 5 - 6 * 7 + 8$
19. Hãy trình diễn cách tính giá trị của biểu thức hậu tố sau đây nhờ sử dụng ngăn xếp:
- $1 \ 2 \ + \ 3 \ 1 \ + \ * \ 1 \ 1 \ + \ 1 \ + \ /$
  - $3 \ 4 \ + \ 3 \ 5 \ + \ * \ 7 \ + \ 8 \ *$

# Chương 4

## CÂY

### 4.1. ĐỊNH NGHĨA VÀ CÁC KHÁI NIỆM

#### 4.1.1. Định nghĩa cây

Cây bao gồm các nút, có một nút đặc biệt được gọi là gốc (root) và các cạnh nối các nút. Cây được định nghĩa đê quy như sau:

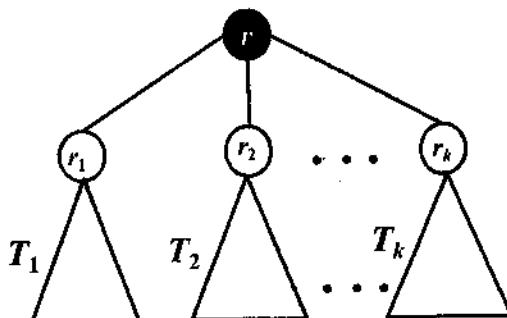
##### Định nghĩa cây

Bước cơ sở: Một nút  $r$  là cây và  $r$  được gọi là gốc của cây này.

Bước quy nạp: Giả sử  $T_1, T_2, \dots, T_k$  là các cây với gốc là  $r_1, r_2, \dots, r_k$ . Ta có thể xây dựng cây mới bằng cách đặt  $r$  làm cha (parent) của các nút  $r_1, r_2, \dots, r_k$ . Trong cây này  $r$  là gốc và  $T_1, T_2, \dots, T_k$  là các cây con của gốc  $r$ . Các nút  $r_1, r_2, \dots, r_k$  được gọi là con của nút  $r$ .

Chú ý: Nhiều khi để phù hợp, ta cần định nghĩa cây rỗng (null tree) là cây không có nút nào cả.

##### Cấu trúc đê quy của cây



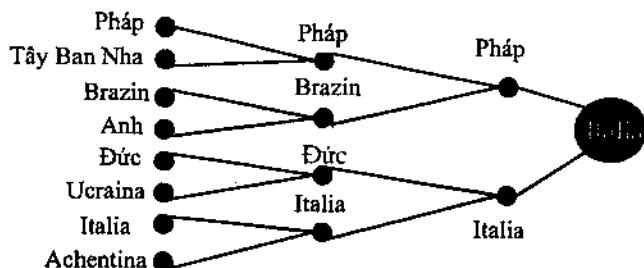
##### Cây trong thực tế ứng dụng

- Biểu đồ lịch thi đấu.
- Cây gia phả.
- Biểu đồ phân cấp quản lý hành chính.
- Cây thư mục.
- Cấu trúc của một quyển sách.

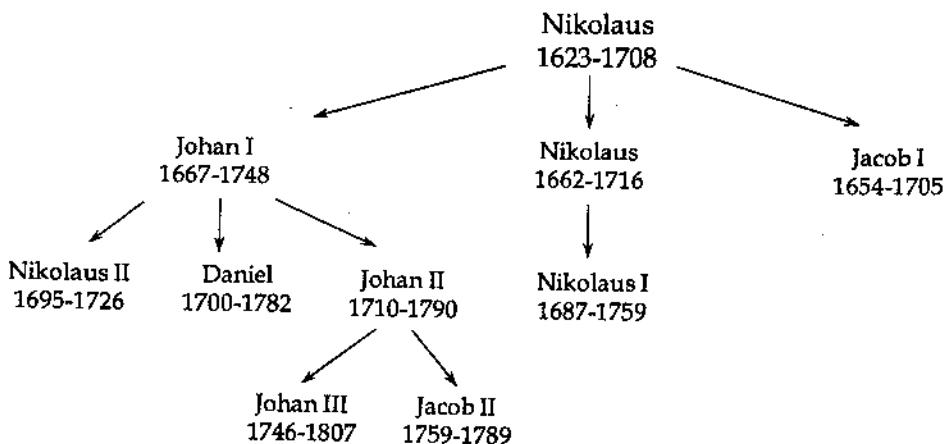
- Cây biểu thức.
- Cây phân hoạch tập hợp.
- ...

### Cây lịch thi đấu

Trong đời thường, cây rất hay được sử dụng để diễn tả lịch thi đấu của các giải thể thao theo thể thức đấu loại trực tiếp, chẳng hạn vòng 2 của World Cup.

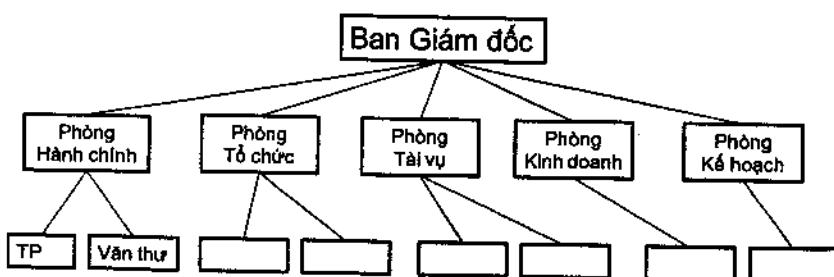


### Cây gia phả



### Cây gia phả của các nhà toán học dòng họ Bernoulli

### Cây phân cấp quản lý hành chính



## Cây thư mục



## Cấu trúc của sách

### Sách

#### C1

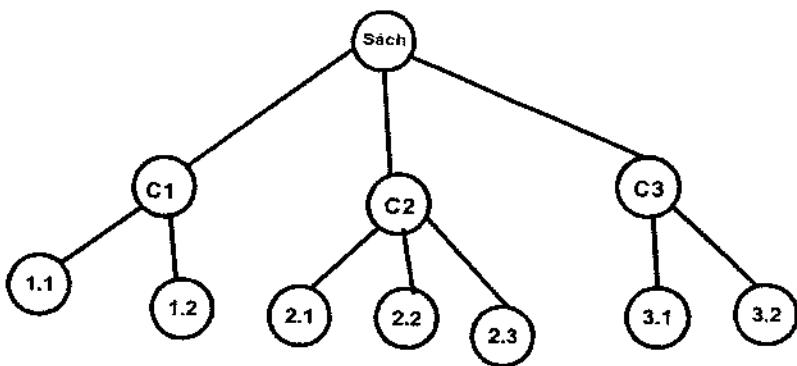
- 1.1.
- 1.2.

#### C2

- 2.1.
- 2.2.
- 2.3.

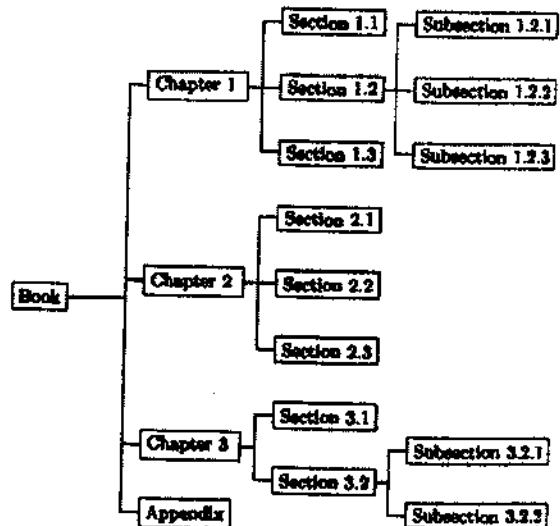
#### C3

- 3.1.
- 3.2.



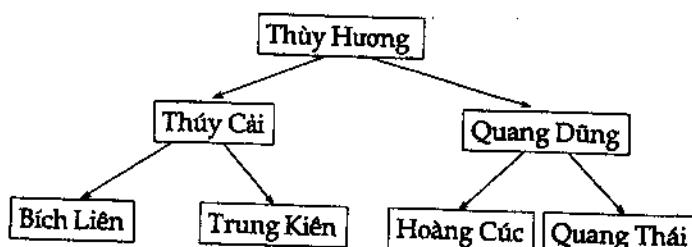
Cây mục lục

Book  
 Chapter 1  
   Section 1.1  
   Section 1.2  
     Subsection 1.2.1  
     Subsection 1.2.2  
     Subsection 1.2.3  
   Section 1.3  
 Chapter 2  
   Section 2.1  
   Section 2.2  
   Section 2.3  
 Chapter 3  
   Section 3.1  
   Section 3.2  
     Subsection 3.2.1  
     Subsection 3.2.2  
 Appendix

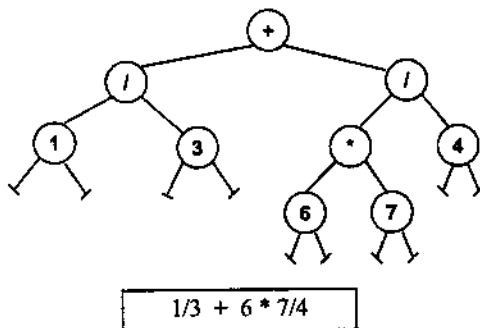


Cây gia phả ngược (Ancestor Tree)

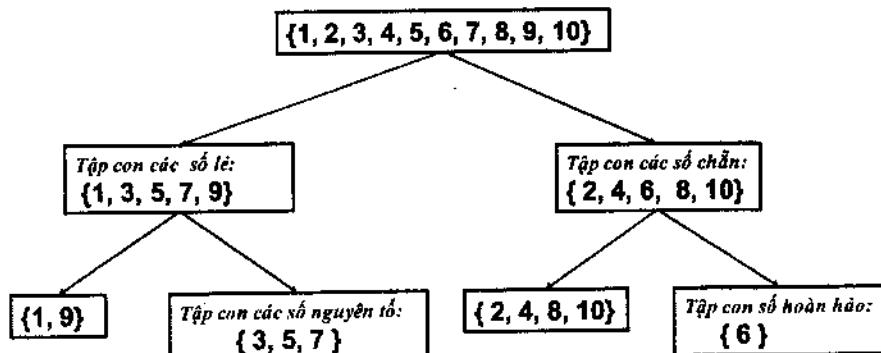
Cây phả hệ ngược: mỗi người đều có bố mẹ. Cây này là cây nhị phân (*binary tree*).



## Cây biểu thức (Expression Tree)



## Cây phân hoạch tập hợp



### 4.1.2. Các thuật ngữ chính

Các thuật ngữ chính liên quan đến cây là:

- Nút – node;
- Gốc – root;
- Lá – leaf;
- Con – child;
- Cha – parent;
- Tổ tiên – ancestors;
- Hậu duệ – descendants;
- Anh em – sibling;
- Nút trong – internal node;
- Chiều cao – height, chiều sâu – depth.

Nếu  $n_1, n_2, \dots, n_k$  là dãy nút trên cây sao cho  $n_i$  là cha của  $n_{i+1}$  với  $1 \leq i < k$ , thì dãy này được gọi là *đường đi* (path) từ nút  $n_1$  tới nút  $n_k$ . Độ dài (length) của đường đi bằng số lượng nút trên đường đi trừ bớt 1. Như vậy đường đi độ dài 0 là đường đi từ một nút đến chính nó.

Nếu có đường đi từ nút  $a$  tới nút  $b$ , thì  $a$  được gọi là *tổ tiên (ancestor)* của  $b$ , còn  $b$  được gọi là *hậu duệ (descendant)* của  $a$ .

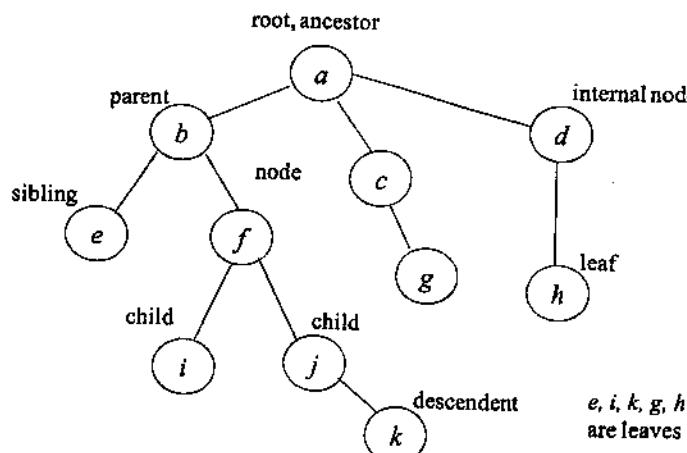
Tổ tiên (hậu duệ) của một nút khác với chính nó được gọi là tổ tiên (hậu duệ) chính thường (*proper*). Trong cây, gốc là nút không có tổ tiên chính thường và mọi nút khác trên cây đều là hậu duệ chính thường của nó.

Một nút không có hậu duệ chính thường được gọi là *lá (leaf)*.

Các nút có cùng cha được gọi là *anh em (sibling)*.

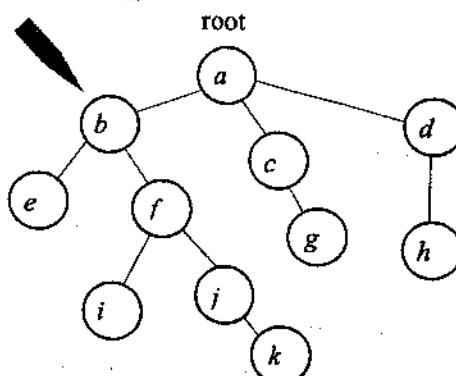
*Cây con (subtree)* của một cây là một nút cùng với tất cả các hậu duệ của nó.

*Chiều cao (height)* của nút trên cây bằng độ dài của đường đi dài nhất từ nút đó đến lá cộng 1. Chiều cao của cây (*height of a tree*) là chiều cao của gốc. *Độ sâu/mức (depth/level)* của nút bằng 1 cộng với độ dài của đường đi duy nhất từ gốc đến nó.

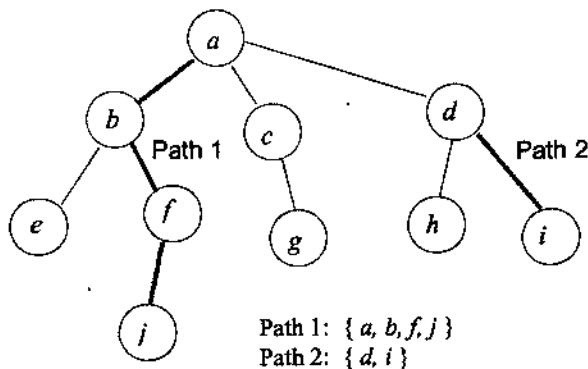


**Cây con (Subtree)**

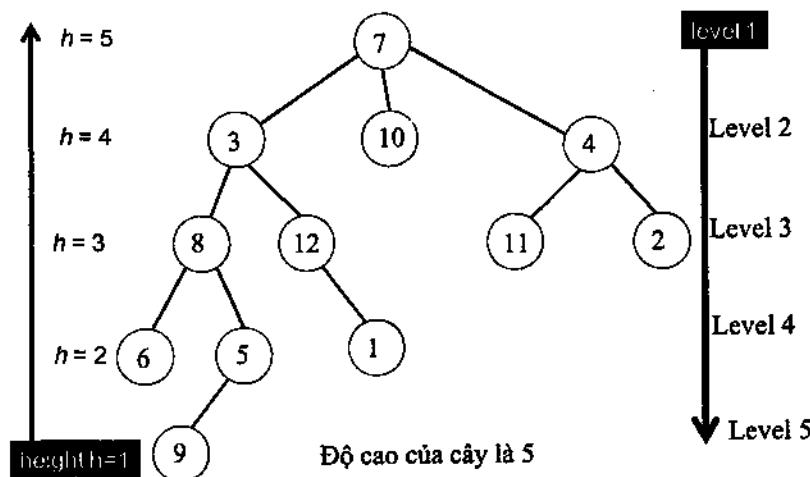
**Node and its descendants**



## Dường đi trên cây



## Độ cao (height) và độ sâu/mức (depth/level)

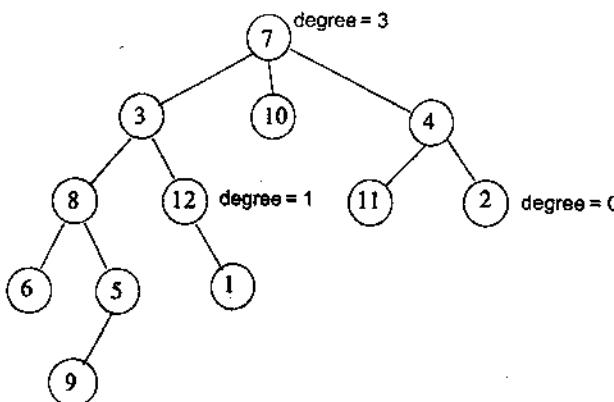


Ta quan sát cây trong tin học như thế nào?



## Bậc (Degree)

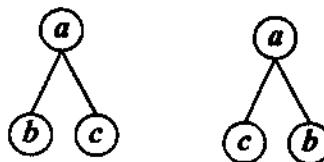
Số lượng con của nút  $x$  được gọi là *bậc (degree)* của  $x$ .



### 4.1.3. Cây có thứ tự (Ordered Tree)

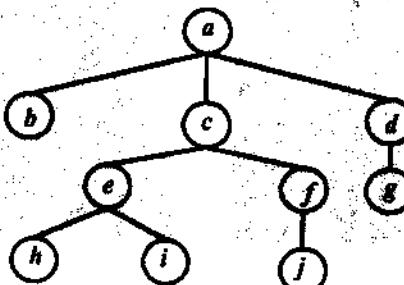
#### Thứ tự của các nút

Các con của một nút thường được xếp theo thứ tự *từ trái sang phải*. Như vậy hai cây trong hình sau đây là *khác nhau*, bởi vì hai con của nút  $a$  xuất hiện trong hai cây theo thứ tự khác nhau:



Cây với các nút được xếp thứ tự được gọi là *cây có thứ tự*. Ta sẽ xét chủ yếu là cây có thứ tự. Vì vậy, tiếp theo đây thuật ngữ cây là để chỉ cây có thứ tự. Khi muốn khẳng định không quan tâm đến thứ tự, ta sẽ phải nói rõ là *cây không có thứ tự*.

Thứ tự "từ trái sang phải" của các anh em (các con của cùng một nút) có thể tổng quát để so sánh hai nút không có quan hệ tổ tiên – hậu duệ. Quy tắc so sánh là: Nếu  $a$  và  $b$  là anh em và  $a$  ở bên trái  $b$  thì tất cả hậu duệ của  $a$  ở bên trái tất cả hậu duệ của  $b$ .



**Ví dụ:** Nút  $h$  ở bên phải nút  $b$ , ở bên trái các nút  $i, f, j, d, g$  và không ở bên trái cũng như bên phải của các nút tổ tiên của nó  $a, c, e$ .

### Xếp thứ tự các nút

Ta có thể xếp thứ tự các nút của cây theo nhiều cách. Có ba thứ tự quan trọng nhất, đó là Thứ tự trước, Thứ tự sau và Thứ tự giữa (Preorder, Postorder và Inorder).

Các thứ tự này được định nghĩa một cách đệ quy như sau:

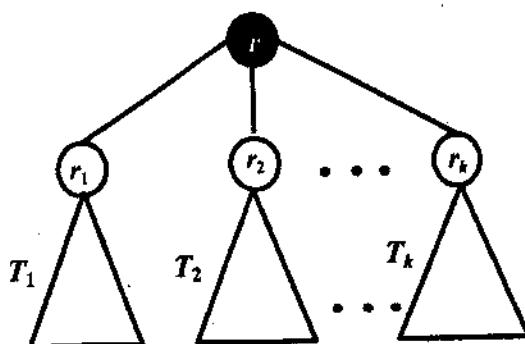
- Nếu cây  $T$  là rỗng, thì danh sách rỗng là danh sách theo thứ tự trước, thứ tự sau và thứ tự giữa của cây  $T$ .
- Nếu cây  $T$  có một nút, thì nút đó chính là danh sách theo thứ tự trước, thứ tự sau và thứ tự giữa của cây  $T$ .

– Trái lại, giả sử  $T$  là cây có gốc  $r$  với các cây con là  $T_1, T_2, \dots, T_k$ .

### Duyệt theo thứ tự trước – Preorder Traversal

*Thứ tự trước* (hay duyệt theo thứ tự trước – *preorder traversal*) của các nút của cây  $T$  là:

- Gốc  $r$  của  $T$ ;
- Tiếp đến là các nút của  $T_1$  theo thứ tự trước;
- Sau đó là các nút của  $T_2$  theo thứ tự trước;
- ...;
- Và cuối cùng là các nút của  $T_k$  theo thứ tự trước.



### Duyệt theo thứ tự sau – Postorder Traversal

*Thứ tự sau* của các nút của cây  $T$  là:

- Các nút của  $T_1$  theo thứ tự sau;
- Tiếp đến là các nút của  $T_2$  theo thứ tự sau;
- ...;
- Các nút của  $T_k$  theo thứ tự sau;
- Sau cùng là nút gốc  $r$ .

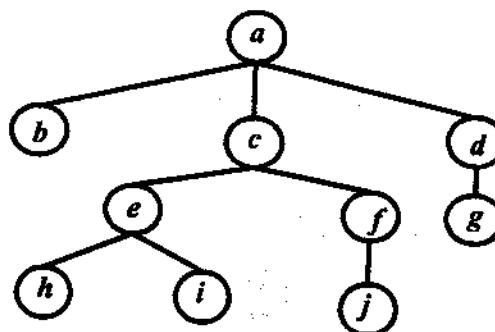
### Duyệt theo thứ tự giữa – Inorder Traversal

Thứ tự giữa của các nút của cây  $T$  là:

- Các nút của  $T_1$  theo thứ tự giữa;
- Tiếp đến là nút gốc  $r$ ;
- Tiếp theo là các nút của  $T_2, \dots, T_k$ , mỗi nhóm nút được xếp theo thứ tự giữa.

### Thuật toán duyệt theo thứ tự trước – Preorder Traversal

```
void PREORDER (nodeT r)
{
 (1) Đưa ra r ;
 (2) for (mỗi con c của r , nếu có, theo thứ tự từ trái sang) do
 PREORDER(c);
}
```



Ví dụ: Thứ tự trước của các đỉnh của cây trên hình vẽ là:

$a, b, c, e, h, i, f, j, d, g$

### Thuật toán duyệt theo thứ tự sau – Postorder Traversal

Thuật toán duyệt theo thứ tự sau thu được bằng cách đảo ngược hai thao tác (1) và (2) trong PREORDER:

```
void POSTORDER (nodeT r)
{
 for (mỗi con c của r , nếu có, theo thứ tự từ trái sang) do
 POSTORDER(c)
 Đưa ra r ;
}
```

Ví dụ: Dãy các đỉnh được liệt kê theo thứ tự sau của cây trong hình vẽ là:

$b, h, i, e, j, f, c, g, d, a$

## Thuật toán duyệt theo thứ tự giữa – Inorder Traversal

```
void INORDER (nodeT r)
```

```
{
```

```
 if (r là lá) Đưa ra r;
```

```
 else
```

```
{
```

```
 INORDER(đứa trái nhất của r);
```

```
 Đưa ra r;
```

```
 for (mỗi con c của r, ngoại trừ con trái nhất, theo thứ tự từ trái sang) do
```

```
 INORDER(c);
```

```
}
```

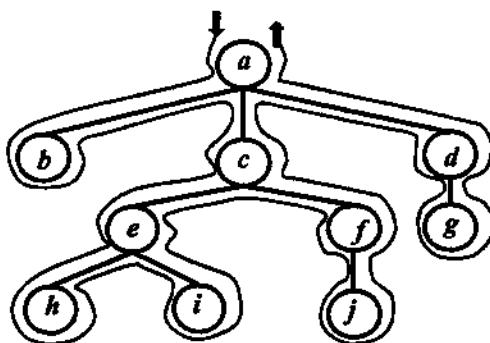
```
}
```

**Ví dụ:** Dãy các đỉnh của cây trong hình vẽ được liệt kê theo thứ tự giữa là:

*b, a, h, e, i, c, j, f, g, d*

### Xếp thứ tự các nút

Để nhớ cách đưa ra các nút theo ba thứ tự vừa trình bày, hãy hình dung là ta đi vòng quanh bên ngoài cây bắt đầu từ gốc, ngược chiều kim đồng hồ và sát theo cây nhất. Chẳng hạn, đường đi đó đối với cây trong các ví dụ đã xét ở trên như sau:



Đối với thứ tự trước, ta đưa ra nút mỗi khi đi qua nó.

Đối với thứ tự sau, ta đưa ra nút khi qua nó ở lần cuối trước khi quay về cha của nó.

Đối với thứ tự giữa, ta đưa ra lá ngay khi đi qua nó, còn những nút trong được đưa ra khi lần thứ hai được đi qua.

Chú ý rằng các lá được xếp theo thứ tự từ trái sang phải như nhau trong cả ba cách sắp xếp.

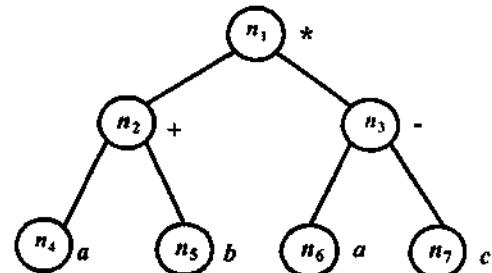
### 4.1.4. Cây có nhãn (Labeled Tree)

Thông thường người ta gán cho mỗi nút của cây một nhãn (*label*) hoặc một giá trị, cũng tương tự như chúng ta đã gán mỗi nút của danh sách với một phần tử.

Nghĩa là, nhãn của nút không phải tên gọi của nút mà là giá trị được cất giữ trong nó. Trong một số ứng dụng, ta có thể thay đổi nhãn của nút mà tên của nó vẫn được giữ nguyên.

**Ví dụ:** Xét cây có 7 nút  $n_1, \dots, n_7$ . Ta gán nhãn cho các nút như sau:

- Nút  $n_1$  có nhãn \*;
- Nút  $n_2$  có nhãn +;
- Nút  $n_3$  có nhãn -;
- Nút  $n_4$  có nhãn  $a$ ;
- Nút  $n_5$  có nhãn  $b$ ;
- Nút  $n_6$  có nhãn  $a$ ;
- Nút  $n_7$  có nhãn  $c$ .



### Cây biểu thức (Expression Tree)

Cây trong ví dụ vừa nêu có tên gọi là cây biểu thức:

$$(a + b)^*(a - c)$$

Quy tắc để cây có nhãn biểu diễn một biểu thức là:

– Mỗi nút lá có nhãn là toán hạng và chỉ gồm một toán hạng đó. Ví dụ nút  $n_4$  biểu diễn biểu thức  $a$ .

– Mỗi nút trong  $n$  được gán nhãn là phép toán. Giả sử  $n$  có nhãn là phép toán hai ngôi  $q$ , như + hoặc \*, con trái biểu diễn biểu thức  $E_1$  và con phải biểu diễn biểu thức  $E_2$ . Khi đó  $n$  biểu diễn biểu thức  $(E_1) q (E_2)$ . Ta có thể bỏ dấu ngoặc nếu như điều đó là không cần thiết.

**Ví dụ:** Nút  $n_2$  chứa toán hạng +, con trái và con phải của nó là  $a$  và  $b$ . Vì thế  $n_2$  biểu diễn  $(a) + (b)$ , hay  $a + b$ .

### Xác định quan hệ hậu duệ

Duyệt theo thứ tự trước và thứ tự sau cho thông tin hữu ích để xác định quan hệ tổ tiên, hậu duệ. Giả sử:

–  $postorder(n)$  là vị trí của nút  $n$  trong thứ tự sắp xếp các đỉnh của cây khi duyệt theo thứ tự sau.

–  $desc(n)$  là số lượng hậu duệ chính thường của nút  $n$ .

**Ví dụ:** Trong cây ở hình trên, thứ tự sau của các nút  $n_2, n_4$  và  $n_5$  tương ứng là 3, 1 và 2.

**Tính chất:** Trong thứ tự sau, các nút trong cây con gốc ở  $n$  được đánh số liên tiếp từ  $postorder(n) - desc(n)$  đến  $postorder(n)$ .

Do đó để kiểm tra nút  $x$  có phải là hậu duệ của nút  $y$  hay không, ta chỉ cần kiểm tra bất đẳng thức sau đúng hay sai:

$$postorder(y) - desc(y) \leq postorder(x) \leq postorder(y).$$

Tính chất tương tự cũng có thể phát biểu đối với duyệt theo thứ tự trước.

#### 4.1.5. ADT Cây

Trong chương này, chúng ta tìm hiểu cây vừa như kiểu dữ liệu trùu tượng vừa như kiểu dữ liệu. Một trong những ứng dụng quan trọng của cây là nó được dùng để thiết kế và cài đặt nhiều kiểu dữ liệu trùu tượng quan trọng khác như "Cây tìm kiếm nhị phân", "Tập hợp",...

Cũng như danh sách, đối với cây cũng có thể xét rất nhiều phép toán làm việc với nó. Dưới đây ta chỉ kể ra một số phép toán cơ bản.

– **parent( $n, T$ )**. Hàm này trả lại cha của nút  $n$  trong cây  $T$ . Nếu  $n$  là gốc (nó không có cha), trả lại  $\Lambda$ . Theo nghĩa này,  $\Lambda$  là nút rỗng ("null node") dùng để báo hiệu rằng chúng ta sẽ rời khỏi cây.

– **leftmost\_child( $n, T$ )** trả lại con trái nhất của nút  $n$  trong cây  $T$  và trả lại  $\Lambda$  nếu  $n$  là lá (không có con).

– **right\_sibling( $n, T$ )** trả lại em bên phải của nút  $n$  trong cây  $T$  (được định nghĩa như là nút  $m$  có cùng cha là  $p$  giống như  $n$ ) sao cho  $m$  nằm sát bên phải của  $n$  trong danh sách sắp thứ tự các con của  $p$ .

Ví dụ, với cây trong trang trước:

– **leftmost\_child( $n_2$ ) =  $n_4$ ;**

– **right\_sibling( $n_4$ ) =  $n_5$ , và RIGHT\_SIBLING ( $n_5$ ) =  $\Lambda$ .**

– **label( $n, T$ )** trả lại nhãn của nút  $n$  trong cây  $T$ . Tuy nhiên, ta không đòi hỏi cây nào cũng có nhãn.

– **create $i(v, T_1, T_2, \dots, T_i)$**  là họ các hàm, mỗi hàm cho một giá trị của  $i = 0, 1, 2, \dots$ .  $createi$  tạo một nút mới  $r$  với nhãn  $v$  và gắn cho nó  $i$  con, với các con là các gốc của cây  $T_1, T_2, \dots, T_i$  theo thứ tự từ trái sang. Trả lại cây với gốc  $r$ . Chú ý, nếu  $i = 0$ , thì  $r$  vừa là lá vừa là gốc.

– **root( $T$ )** trả lại nút là gốc của cây  $T$ , hoặc  $\Lambda$  nếu  $T$  là cây rỗng.

– **makenull( $T$ )** biến  $T$  thành cây rỗng.

##### Biểu diễn cây

Có nhiều cách biểu diễn cây. Ta giới thiệu qua ba cách biểu diễn cơ bản:

– Dùng mảng (Array Representation);

– Danh sách các con (Lists of Children);

– Dùng con trái và em phải (The Leftmost-Child, Right-Sibling Representation).

##### 4.1.5.1. Biểu diễn cây dùng mảng

Giả sử  $T$  là cây với các nút đặt tên là  $1, 2, \dots, n$ . Cách đơn giản để biểu diễn  $T$  là hỗ trợ thao tác **parent** bởi danh sách tuyến tính  $A$  trong đó mỗi phần tử  $A[i]$  chứa con trỏ đến cha của nút  $i$ . Riêng gốc của  $T$  có thể phân biệt bởi con trỏ rỗng.

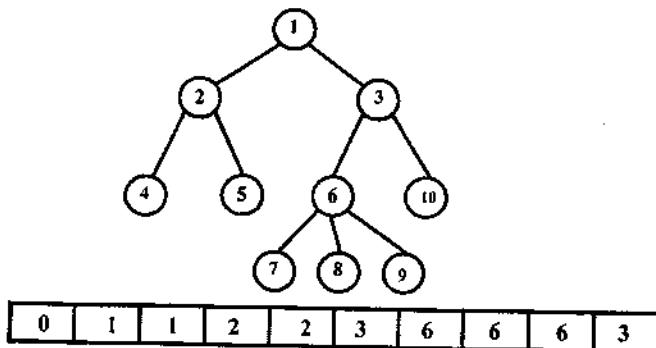
Khi dùng mảng, ta đặt  $A[i] = j$  nếu nút  $j$  là cha của nút  $i$  và  $A[i] = 0$  nếu nút  $i$  là gốc.

Cách biểu diễn này dựa trên cơ sở là mỗi nút của cây (ngoại trừ gốc) đều có duy nhất một cha. Với cách biểu diễn này, cha của một nút có thể xác định trong thời gian hằng số. Đường đi từ một nút đến tổ tiên của chúng (kể cả đến gốc) có thể xác định dễ dàng:

$n \leftarrow \text{parent}(n) \leftarrow \text{parent}(\text{parent}(n)) \leftarrow \dots$

Ta cũng có thể đưa thêm vào mảng  $L[i]$  để hỗ trợ việc ghi nhận nhãn cho các nút, hoặc biến mỗi phần tử  $A[i]$  thành bản ghi gồm hai trường: biến nguyên ghi nhận cha và nhãn.

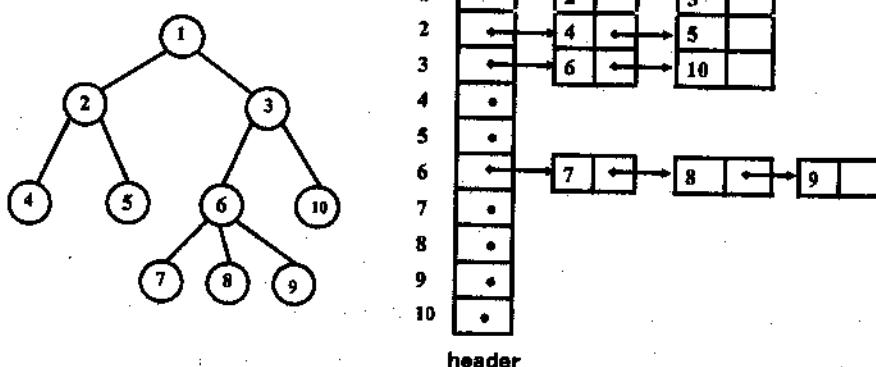
Ví dụ



**Hạn chế:** Cách dùng con trỏ cha không thích hợp cho các thao tác với con. Cho nút  $n$ , ta sẽ mất nhiều thời gian để xác định các con của  $n$  hoặc chiều cao của  $n$ . Hơn nữa, biểu diễn bởi con trỏ cha không cho ta thứ tự của các nút con. Vì thế các phép toán như `leftmost_child` và `right_sibling` là không xác định được. Do đó cách biểu diễn này chỉ dùng trong một số trường hợp nhất định.

#### 4.1.5.2. Danh sách các con (Lists of Children)

Trong cách biểu diễn này, với mỗi nút của cây ta cất giữ một danh sách các con của nó. Danh sách con có thể biểu diễn bởi một trong những cách biểu diễn danh sách đã trình bày trong chương trước. Tuy nhiên, chú ý rằng số lượng con của các nút rất khác nhau, nên danh sách móc nối thường là lựa chọn thích hợp nhất.



Cần có mảng con trả đến đầu các danh sách con của các nút 1, 2, ..., 10:

*header[i]* trả đến danh sách con của nút *i*.

Ví dụ: Có thể sử dụng mô tả sau đây để biểu diễn cây:

```
typedef ? NodeT; /* dấu ? cần thay bởi định nghĩa
kiểu phù hợp */

typedef ? ListT; /* dấu ? cần thay bởi định nghĩa
kiểu danh sách phù hợp */

typedef ? position;

typedef struct
{
 ListT header[maxNodes];
 labeltype labels[maxNodes];
 NodeT root;
} TreeT;
```

Ta giả thiết rằng gốc của cây được cất giữ trong trường *root* và 0 để thể hiện nút rỗng.

#### Cài đặt *leftmost\_child*

Dưới đây là minh họa cài đặt phép toán *leftmost\_child*. Việc cài đặt các phép toán còn lại được coi là bài tập.

```
NodeT leftmost_child (NodeT n, TreeT T)
/* trả lại con trái nhất của nút n trong cây T */
{
 ListT L; /* danh sách các con của n */
 L = T.header[n];
 if (empty(L)) /* n là lá */
 return(0);
 else return(retrieve (first(L), L));
}
```

Dùng con trái và em kế cận phải

(The Leftmost-Child, Right-Sibling Representation)

Rõ ràng, mỗi một nút của cây chỉ có thể có:

- Hoặc là không có con, hoặc có đúng một nút con cực trái (con cà);
- Hoặc là không có em kề cận phải, hoặc có đúng một nút em kề cận phải (right-sibling).

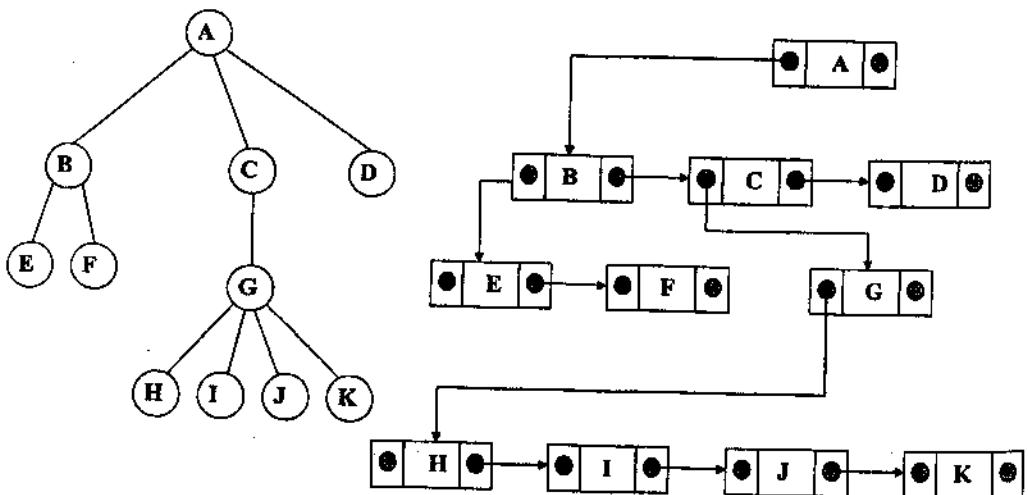
Vì vậy để biểu diễn cây, ta có thể lưu trữ thông tin về con cực trái và em kề cận phải của mỗi nút. Ta có thể sử dụng mô tả sau:

```

struct Tnode
{
 char word[20]; // Dữ liệu cắt giữ ở nút
 struct Tnode *leftmost_child;
 struct Tnode *right_sibling;
};

typedef struct Tnode treeNode;
treeNode Root;

```



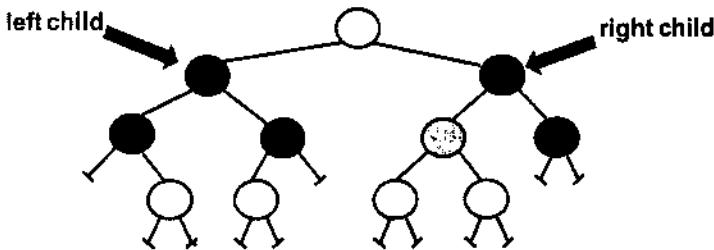
*Biểu diễn cây bởi con trái và em kề cận phải*

## 4.2. CÂY NHỊ PHÂN

### 4.2.1. Định nghĩa và tính chất

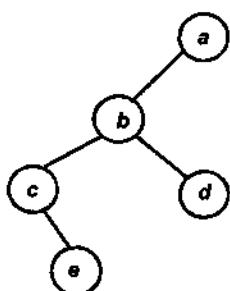
**Định nghĩa:** Cây nhị phân là cây mà mỗi nút có nhiều nhất hai con.

Vì mỗi nút chỉ có không quá hai con, nên ta sẽ gọi chúng là *con trái* và *con phải* (*left and right child*). Như vậy mỗi nút của cây nhị phân hoặc *không có con*, hoặc *chỉ có con trái*, hoặc *chỉ có con phải*, hoặc *có cả con trái và con phải*.

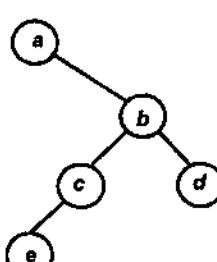


Vì ta phân biệt con trái và con phải, nên khái niệm cây nhị phân không trùng với cây có thứ tự định nghĩa ở 4.1.

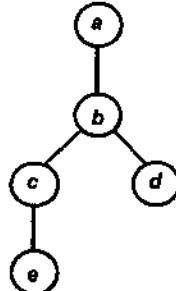
**Ví dụ:**



Binary tree  $T_1$



Binary tree  $T_2$



Vì thế, chúng ta sẽ không so sánh cây nhị phân với cây tổng quát.

### Tính chất của cây nhị phân

#### Bố đề 1:

- (i) Số đỉnh lớn nhất ở trên mức  $i$  của cây nhị phân là  $2^{i-1}$ ,  $i \geq 1$ .
- (ii) Một cây nhị phân với chiều cao  $k$  có không quá  $2^k - 1$  nút,  $k \geq 1$ .
- (iii) Một cây nhị phân có  $n$  nút có chiều cao tối thiểu là  $\lceil \log_2(n + 1) \rceil$ .

#### Chứng minh:

- (i) Bằng quy nạp theo  $i$ .

**Cơ sở:** Gốc là nút duy nhất trên mức  $i = 1$ . Như vậy số đỉnh lớn nhất trên mức  $i = 1$  là  $2^0 = 2^{1-1}$ .

**Chuyển quy nạp:** Giả sử với mọi  $j$ ,  $1 \leq j < i$ , số đỉnh lớn nhất trên mức  $j$  là  $2^{j-1}$ . Do số đỉnh trên mức  $i - 1$  là  $2^{i-2}$ , mặt khác theo định nghĩa, mỗi đỉnh trên cây nhị phân có không quá hai con, ta suy ra số lượng nút lớn nhất trên mức  $i$  không vượt quá hai lần số lượng nút trên mức  $i - 1$ , nghĩa là không vượt quá  $2 * 2^{i-2} = 2^{i-1}$ .

(ii) Số lượng nút lớn nhất của cây nhị phân chiều cao  $k$  không vượt quá tổng số lượng nút lớn nhất trên các mức  $i = 1, 2, \dots, k$ . Theo bố đề 1, số này không vượt quá:

$$\sum_{i=1}^k 2^{i-1} = 2^k - 1.$$

(iii) Cây nhị phân  $n$  nút có chiều cao thấp nhất  $k$  khi số lượng nút ở các mức  $i = 1, 2, \dots, k$  đều là lớn nhất có thể được. Từ đó ta có:

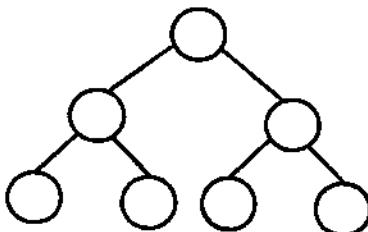
$$n = \sum_{i=1}^k 2^{i-1} = 2^k - 1, \text{ suy ra } 2^k = n + 1, \text{ hay } k = \lceil \log_2(n+1) \rceil.$$

### Cây nhị phân đầy đủ (full binary tree)

**Định nghĩa:** Cây nhị phân đầy đủ là cây nhị phân thỏa mãn:

- Mỗi nút lá đều có cùng độ sâu và;
- Các nút trong có đúng hai con.

**Ví dụ:** Cây nhị phân hoàn chỉnh được cho trong hình vẽ sau:



**Bổ đề 2:** Cây nhị phân đầy đủ với độ sâu  $n$  có  $2^n - 1$  nút.

**Chứng minh:** Suy trực tiếp từ bổ đề 1.

### Cây nhị phân hoàn chỉnh (Complete Binary Trees)

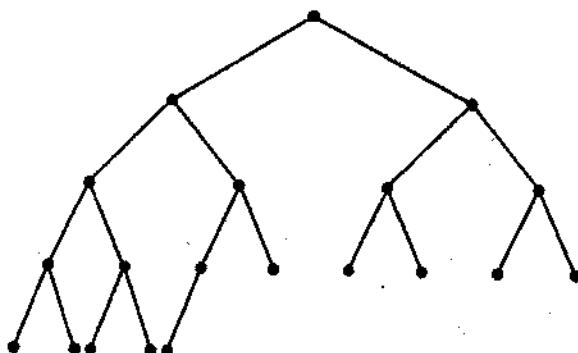
**Định nghĩa:** Cây nhị phân hoàn chỉnh là cây nhị phân độ sâu  $n$  thỏa mãn:

- Là cây nhị phân đầy đủ nếu không tính đến các nút ở độ sâu  $n$ ;
- Tất cả các nút ở độ sâu  $n$  lệch sang trái nhất có thể được.

**Bổ đề 3:** Cây nhị phân hoàn chỉnh độ sâu  $n$  có số lượng nút nằm trong khoảng từ  $2^{n-1}$  đến  $2^n - 1$ .

**Chứng minh:** Suy trực tiếp từ định nghĩa và bổ đề 1.

**Ví dụ:** Cây nhị phân hoàn chỉnh:



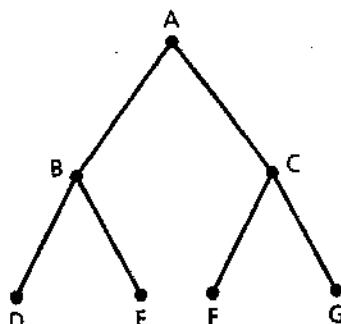
### Cây nhị phân cân đối (balanced binary tree)

**Định nghĩa:** Cây nhị phân được gọi là **cân đối** nếu chiều cao cây con trái và chiều cao cây con phải của mọi nút chênh lệch nhau không quá 1 đơn vị.

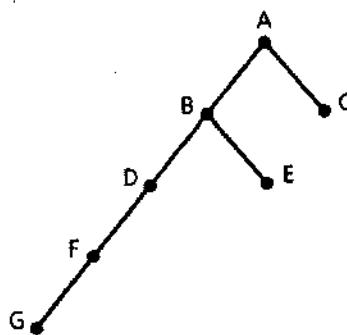
#### Nhận xét:

- Nếu cây nhị phân là đầy đủ thì nó là hoàn chỉnh.
- Nếu cây nhị phân là hoàn chỉnh thì nó là cân đối.

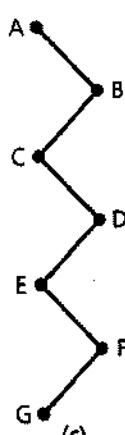
#### Ví dụ:



(a)



(b)



(c)

1. Cây nào là đầy đủ?
2. Cây nào là hoàn chỉnh?
3. Cây nào là cân đối?

#### 4.2.2. Biểu diễn cây nhị phân

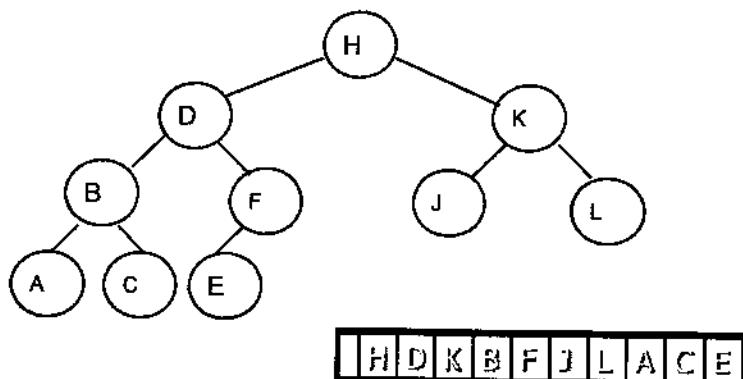
Ta xét hai phương pháp:

- Biểu diễn sử dụng mảng.
- Biểu diễn sử dụng con trỏ.

**Biểu diễn sử dụng mảng:** Hoàn toàn tương tự như trong cách biểu diễn cây tổng quát. Tuy nhiên, trong trường hợp cây nhị phân hoàn chỉnh, sử dụng cách biểu diễn này có thể cài đặt nhiều phép toán với cây rất hiệu quả.

Xét cây nhị phân hoàn chỉnh  $T$  có  $n$  nút, trong đó mỗi nút chứa một giá trị. Gán nhãn cho các nút của cây nhị phân hoàn chỉnh  $T$  từ trên xuống dưới và từ trái qua phải bằng các số  $1, 2, \dots, n$ . Đặt tương ứng cây  $T$  với *mảng*  $A$ , trong đó phần tử thứ  $i$  của  $A$  là giá trị cất giữ trong nút thứ  $i$  của cây  $T$ ,  $i = 1, 2, \dots, n$ .

## Biểu diễn mảng của cây nhị phân hoàn chỉnh



| <i>Để tìm</i>       | <i>Sử dụng</i> | <i>Hạn chế</i>   |
|---------------------|----------------|------------------|
| Con trái của $A[i]$ | $A[2*i]$       | $2*i \leq n$     |
| Con phải của $A[i]$ | $A[2*i + 1]$   | $2*i + 1 \leq n$ |
| Cha của $A[i]$      | $A[i/2]$       | $i > 1$          |
| Gốc                 | $A[1]$         | $A$ khác rỗng    |
| Thứ $A[i]$ là lá?   | True           | $2*i > n$        |

### Biểu diễn cây nhị phân dùng con trỏ

Mỗi nút của cây sẽ có con trỏ đến con trái và con trỏ đến con phải:



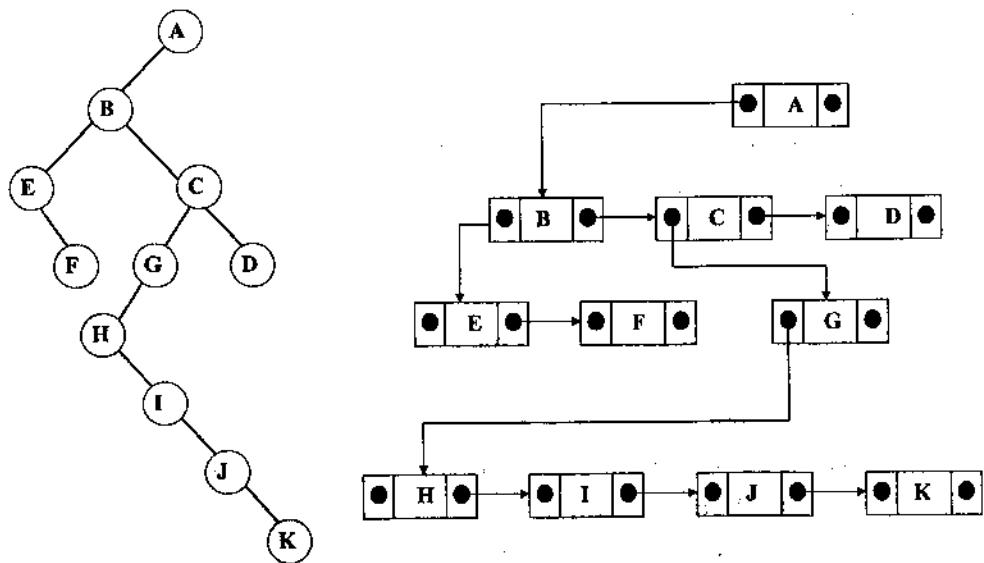
```

struct Tnode {
 DataType Item; // DataType - kiểu dữ liệu của phần tử
 struct Tnode *left;
 struct Tnode *right;
};

typedef struct Tnode treeNode;

```

## Ví dụ



### Các phép toán cơ bản

```
struct Tnode
{ char word[20]; // Dữ liệu của nút
 struct Tnode * left;
 struct Tnode *right;
};

typedef struct Tnode treeNode;

treeNode* makeTreeNode(char *word);
treeNode *RandomInsert(treeNode* tree,char *word);
void freeTree(treeNode *tree);
void printPreorder(treeNode *tree);
void printPostorder(treeNode *tree);
void printInorder(treeNode *tree);
int countNodes(treeNode *tree);
int depth(treeNode *tree);
```

### Tạo một nút (MakeTreeNode)

Thông số: Dữ liệu của nút cần bổ sung.

Các bước:

- Phân bộ bộ nhớ cho nút mới.
- Kiểm tra cấp phát bộ nhớ có thành công?
- Nếu đúng, đưa item vào nút mới.
- Đặt con trỏ trái và phải bằng NULL.

Đầu ra: con trỏ đến (là địa chỉ của) nút mới.

```
treeNode* makeTreeNode(char *word)
{
 treeNode* newNode = NULL;
 newNode = (treeNode*)malloc(sizeof(treeNode));
 if (newNode == NULL){
 printf("Out of memory\n");
 exit(1);
 }
 else {
 newNode->left = NULL;
 newNode->right= NULL;
 strcpy(newNode->word,word);
 }
 return newNode;
}
```

Cài đặt hàm tính số nút và độ sâu của cây

```
int countNodes(treeNode *tree) {
/* the function counts the number of nodes of a tree*/
if(tree == NULL) return 0;
else {
 int ld = countNodes(tree->left);
 int rd = countNodes(tree->right);
 return 1+ld+rd;
}
}

int depth(treeNode *tree) {
```

```

/* the function computes the depth of a tree */
if(tree == NULL) return 0;
int ld = depth(tree->left);
int rd = depth(tree->right);
/* if (ld > rd) return 1+ld; else return 1+rd; */
return 1 + (ld > rd ? ld : rd);
}

```

### Loại bỏ cây

```

void freeTree(treeNode *tree)
{
 if(tree == NULL) return;
 freeTree(tree->left);
 freeTree(tree->right);
 free(tree);
 return;
}

```

### Duyệt cây nhị phân

Duyệt cây nhị phân là cách duyệt có hệ thống các nút của cây. Tương tự cây tông quát, ta xét ba thứ tự duyệt cây nhị phân:

#### Thứ tự trước (Preorder) NLR

- Thăm nút (Visit a node);
- Thăm cây con trái theo thứ tự trước (Visit left subtree);
- Thăm cây con phải theo thứ tự trước (Visit right subtree).

#### Thứ tự giữa (Inorder) LNR

- Thăm cây con trái theo thứ tự giữa (Visit left subtree);
- Thăm nút (Visit a node);
- Thăm cây con phải theo thứ tự giữa (Visit right subtree).

#### Thứ tự sau (Postorder) LRN

- Thăm cây con trái theo thứ tự sau (Visit left subtree);
- Thăm cây con phải theo thứ tự sau (Visit right subtree);
- Thăm nút (Visit a node).

#### Duyệt theo thứ tự trước – NLR – Preorder Traversal

```

void printPreorder(treeNode *tree)
{
 if(tree != NULL)

```

```

 {
 printf("%s\n", tree->word);
 printPreorder(tree->left);
 printPreorder(tree->right);
 }
}

```

Duyệt theo thứ tự giữa – LNR – Inorder Traversal

```

void printInorder(treeNode *tree)
{
 if(tree != NULL)
 {
 printInorder(tree->left);
 printf("%s\n", tree->word);
 printInorder(tree->right);
 }
}

```

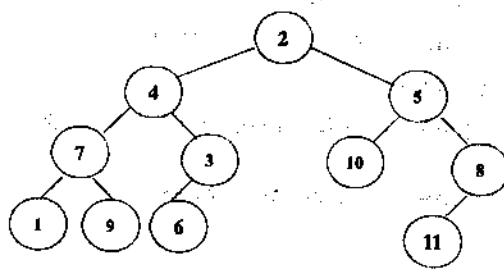
Duyệt theo thứ tự sau – LRN – Postorder Traversal

```

void printPostorder(treeNode *tree)
{
 if(tree != NULL)
 {
 printPostorder(tree->left);
 printPostorder(tree->right);
 printf("%s\n", tree->word);
 }
}

```

Ví dụ:

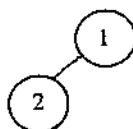


Preorder Traversal: 2, 4, 7, 1, 9, 3, 6, 5, 10, 8, 11

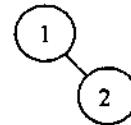
Inorder Traversal: 1, 7, 9, 4, 6, 3, 2, 10, 5, 11, 8

Postorder Traversal: 1, 9, 7, 6, 3, 4, 10, 11, 8, 5, 2

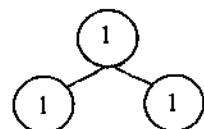
**Chú ý:** Ta có thể xây dựng được cây nhị phân mà thứ tự trước và thứ tự giữa hoặc thứ tự sau và thứ tự giữa là như nhau (xem hình vẽ dưới đây); nhưng không thể có cây nhị phân mà thứ tự trước và thứ tự sau là như nhau.



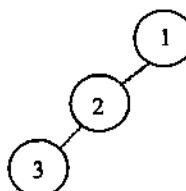
Pre: 12  
Post: 21  
In: 21



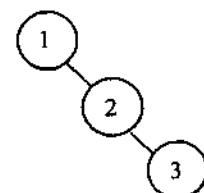
Pre: 12  
Post: 21  
In: 12



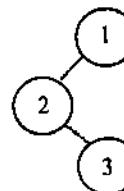
Pre: 123  
Post: 231  
In: 213



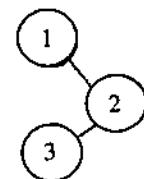
Pre: 123  
Post: 321  
In: 321



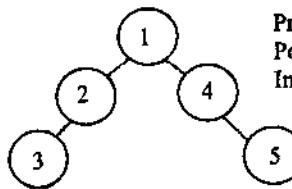
Pre: 123  
Post: 321  
In: 123



Pre: 123  
Post: 321  
In: 231



Pre: 123  
Post: 321  
In: 132



Pre: 12345  
Post: 32541  
In: 32145

### Chương trình minh họa

```

The program for testing binary tree traversal

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <process.h>
```

```

struct Tnode
{
 char word[20];
 struct Tnode * left;
 struct Tnode * right;
};

typedef struct Tnode treeNode;

treeNode* makeTreeNode(char *word);
treeNode *RandomInsert(treeNode* tree,char *word);
void freeTree(treeNode *tree);
void printPreorder(treeNode *tree);
void printPostorder(treeNode *tree);
void printInorder(treeNode *tree);
int countNodes(treeNode *tree);
int depth(treeNode *tree);

int main()
{
 treeNode *randomTree=NULL;
 char word[20] = "a";
 while(strcmp(word,"~")!=0)
 {
 printf("\nEnter item (~ to finish): ");
 scanf("%s", word);
 if (strcmp(word,"~")==0)
 printf("Cham dut nhap thong tin nut...\n");
 else randomTree=RandomInsert(randomTree,word);
 }

 printf("The tree in preorder:\n");
 printPreorder(randomTree);
 printf("*****\n");

 printf("The tree in postorder:\n");

```

```

printPostorder(randomTree);
printf("*****\n");

printf("The tree in inorder:\n");
printInorder(randomTree);
printf("*****\n");

printf("The number of nodes is:
%d\n",countNodes(randomTree));
printf("The depth of the tree is: %d\n",
depth(randomTree));

freeTree(randomTree);

getch();
return 0;
}

treeNode* makeTreeNode(char *word)
{
 treeNode* newNode = NULL;
 newNode = (treeNode*)malloc(sizeof(treeNode));
 if (newNode == NULL)
 {
 printf("Out of memory\n");
 exit(1);
 }
 else
 {
 newNode->left = NULL;
 newNode->right= NULL;
 strcpy(newNode->word,word);
 }
 return newNode;
}

```

```

treeNode *RandomInsert(treeNode *tree,char *word)
{
 treeNode *newNode, *p;

 newNode = makeTreeNode(word);
 if (tree == NULL) return newNode;
 if (rand()%2 ==0)
 {
 p=tree;
 while (p->left !=NULL) p=p->left;
 p->left=newNode;
 printf("Node %s is left child of %s
\n",word,(*p).word);
 }
 else
 {
 p=tree;
 while (p->right !=NULL) p=p->right;
 p->right=newNode;
 printf("Node %s is right child of %s
\n",word,(*p).word);
 }
 return tree;
}

void freeTree(treeNode *tree)
{
 if(tree == NULL) return;
 freeTree(tree->left);
 freeTree(tree->right);
 free(tree);
 return;
}

void printPreorder(treeNode *tree)
{

```

```

if(tree != NULL)
{
 printf("%s\n", tree->word);
 printPreorder(tree->left);
 printPreorder(tree->right);
}
}

void printPostorder(treeNode *tree)
{
 if(tree != NULL)
 {
 printPostorder(tree->left);
 printPostorder(tree->right);
 printf("%s\n", tree->word);
 }
}

void printInorder(treeNode *tree)
{
 if(tree != NULL)
 {
 printInorder(tree->left);
 printf("%s\n", tree->word);
 printInorder(tree->right);
 }
}

int countNodes(treeNode *tree)
{
/* the function counts the number of nodes of a tree*/
 if(tree == NULL) return 0;
 else
 {
 int ld = countNodes(tree->left);
 int rd = countNodes(tree->right);

```

```

 return 1+ld+rd;
 }
}

int depth(treeNode *tree)
{
 /* the function computes the depth of a tree */
 if(tree == NULL) return 0;
 int ld = depth(tree->left);
 int rd = depth(tree->right);
 /* if (ld > rd) return 1+ld; else return 1+rd; */
 return 1 + (ld > rd ? ld : rd);
}

```

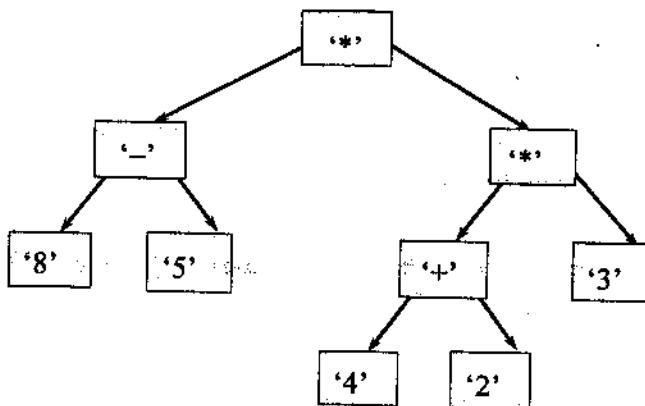
### 4.3. CÁC VÍ DỤ ÚNG DỤNG

#### 4.3.1. Cây nhị phân biểu thức (Binary Expression Trees)

Cây biểu thức là cây nhị phân trong đó:

1. Mỗi nút lá chứa một toán hạng.
2. Mỗi nút trong chứa một phép toán hai ngôi.
3. Các cây con trái và phải của nút phép toán biểu diễn các biểu thức con (subexpressions) cần được thực hiện trước khi thực hiện phép toán ở gốc của các cây con.

Ví dụ: Cây nhị phân bốn mức.



Cây nhị phân biểu thức  $(8 - 5) * (4 + 2)/3$

## Các mức thể hiện mức độ ưu tiên thực hiện phép toán

– Mức (độ sâu) của các nút trên cây cho biết trình tự thực hiện chúng (ta không cần sử dụng ngoặc để chỉ ra trình tự).

– Phép toán tại mức cao hơn của cây được thực hiện muộn hơn so với các mức dưới chúng.

– Phép toán ở gốc luôn được thực hiện sau cùng.

### Duyệt cây biểu thức

Sử dụng cây biểu thức ta có thể đưa ra biểu thức trong ký pháp trung tố, tiền tố và hậu tố:

– Duyệt cây biểu thức theo thứ tự trước (Preorder) cho ta ký pháp tiền tố (Prefix Notation).

– Duyệt cây biểu thức theo thứ tự giữa (Inorder) cho ta ký pháp trung tố (Infix Notation).

– Duyệt cây biểu thức theo thứ tự sau (Postorder) cho ta ký pháp hậu tố (Postfix Notation).

**Ví dụ:** Cây trong hình vẽ ở trên:

Infix:  $((8 - 5) * ((4 + 2) / 3))$

Prefix:  $* - 8 5 / + 4 2 3$

Postfix:  $8 5 - 4 2 + 3 / *$

### 4.3.2. Cây quyết định – Decision Trees

Cây quyết định là một cây mà mỗi đỉnh trong của nó tương ứng với một truy vấn đối với dữ liệu. Các cạnh của cây tương ứng với các khả năng trả lời cho câu hỏi. Mỗi lá của cây tương ứng với một đầu ra.

Cây quyết định là mô hình tính toán để giải nhiều bài toán ứng dụng.

Để tính toán với cây quyết định, chúng ta sẽ bắt đầu từ gốc và di chuyển theo một đường đi đến lá. Tại mỗi nút trung gian câu trả lời cho truy vấn đối với dữ liệu sẽ dẫn ta đến nút tiếp theo. Khi đạt đến lá ta thu được một đầu ra.

**Ví dụ: Bài toán tra từ điển.**

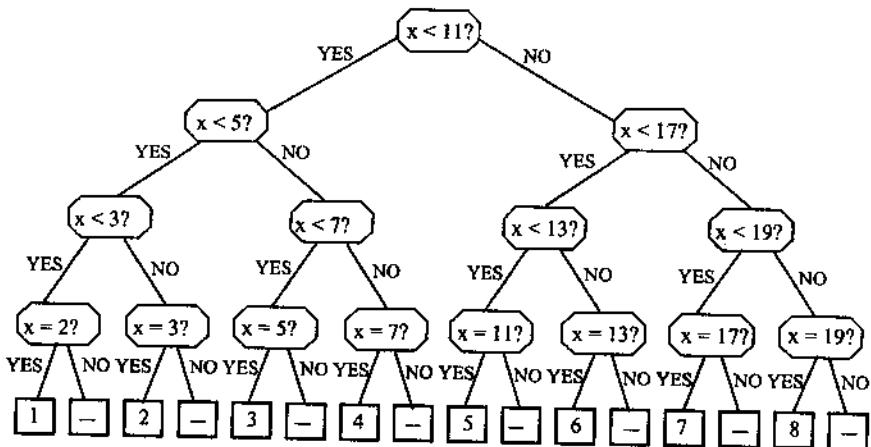
Cho mảng A gồm n số được sắp xếp theo thứ tự tăng dần và một số x. Cần xác định xem x có mặt trong mảng đã cho hay không. Nếu câu trả lời là khẳng định cần chỉ rõ vị trí của x trong mảng A.

Cây quyết định để giải bài toán này có dạng một cây nhị phân.

– Mỗi nút trong của cây quyết định chứa một câu hỏi dạng “ $x < k ?$ ”.

– Mỗi nút ở mức sát lá của cây quyết định chứa câu hỏi “ $x = A[i] ?$ ” có hai con, một con ứng với ‘i’ còn một con ứng với ‘-’ (không có).

**Ví dụ:** Xét  $A = (2, 3, 5, 7, 11, 13, 17, 19)$ . Cây quyết định có dạng sau đây:



Ta xác định thời gian của thuật toán cây quyết định là số truy vấn trên đường đi từ gốc đến lá. Như vậy, thời gian tính trong tình huống xấu nhất của thuật toán sẽ là độ cao của cây quyết định.

Nếu ta hạn chế chỉ được sử dụng phép so sánh để giải bài toán tra từ điển, thì rõ ràng hoạt động của mọi thuật toán giải bài toán tra từ điển đều có thể mô tả bởi cây quyết định.

Do đó nếu đánh giá được *cân dưới cho độ cao* của cây quyết định (thuật toán) giải bài toán tra từ điển thì ta cũng thu được *cân dưới cho độ phức tạp* của bài toán.

Giả thiết rằng: "Các truy vấn đối với dữ liệu phải đảm bảo đủ thông tin để có thể xác định được đầu ra có thể".

Nếu bài toán có  $N$  đầu ra có thể thì rõ ràng mọi cây quyết định đều có  $N$  lá (không loại trừ tình huống khi có một số lá tương ứng với cùng một đầu ra).

Từ bô đề 1, ta biết rằng: Nếu một cây có  $n$  lá và mỗi nút có nhiều nhất hai nút con (mỗi truy vấn có nhiều nhất hai câu trả lời) thì độ cao của cây ít nhất sẽ là:

$$\lceil \log n \rceil = \Omega(\log n).$$

Áp dụng kết quả này: Với bài toán tra từ điển với tập  $S$  gồm  $n$  phần tử, do bài toán có tất cả  $n + 1$  đầu ra có thể, nên mọi cây quyết định có  $n + 1$  lá và vì thế, có độ cao ít nhất là:

$$\lceil \log(n+1) \rceil = \Omega(\log n).$$

Suy ra: Cận dưới của độ phức tạp tính toán của bài toán tra từ điển chỉ sử dụng phép so sánh là  $\Omega(\log n)$ .

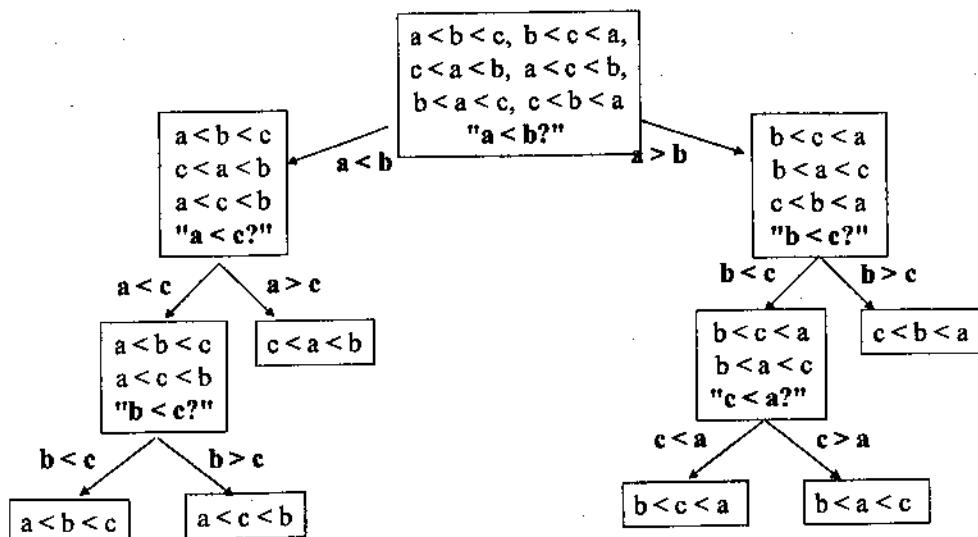
Như đã biết: Thuật toán tìm kiếm nhị phân có thời gian tính là  $O(\log n)$ . Từ các kết quả trên suy ra: "Thuật toán tìm kiếm nhị phân là thuật toán nhanh nhất trong số các thuật toán chỉ sử dụng phép so sánh để giải bài toán tra từ điển".

**Bài toán sắp xếp:** Cho dãy  $n$  số phân biệt ( $a[1], a[2], \dots, a[n]$ ), tìm hoán vị  $\pi = (\pi(1), \pi(2), \dots, \pi(n))$  sao cho:

$$a[\pi(1)] < a[\pi(2)] < \dots < a[\pi(n)].$$

Giả sử ta chỉ sử dụng phép so sánh để thực hiện sắp xếp, khi đó mọi thuật toán sắp xếp đều có thể diễn tả trong mô hình cây quyết định. Cây quyết định là cây nhị phân thỏa mãn:

- Mỗi nút  $\equiv$  một phép so sánh " $a < b$ " (cũng có thể coi tương ứng với một không gian con các trình tự).
- Mỗi cạnh  $\equiv$  rẽ nhánh theo câu trả lời (true hoặc false).
- Mỗi lá  $\equiv$  1 trình tự sắp xếp.
- Nếu có  $n$  phần tử phân biệt cần sắp xếp thì Cây quyết định có  $n!$  lá, tức là tất cả các trình tự có thể.
- Đối với mỗi bộ dữ liệu, chỉ có một lá chứa trình tự sắp xếp cần tìm.



Cây quyết định cho bài toán sắp xếp ba số  $a, b, c$ .

Mỗi lá tương ứng với một trình tự sắp xếp ba số  $a, b, c$ .

Do có tất cả  $n!$  hoán vị, nên mọi cây quyết định giải bài toán sắp xếp đều có ít nhất  $n!$  lá, vì thế có độ cao ít nhất là  $\Omega(\log n!)$ .

Ta có:

$$\begin{aligned}\log n! &= \log 1 + \log 2 + \dots + \log n \\&\geq \log(n/2) + \log(n/2+1) + \dots + \log n \\&\geq (n/3) \log(n/2) \\&= (1/3)(n \log n - n \log 2) \\&= \Omega(n \log n).\end{aligned}$$

Như vậy, ta đã chứng minh được kết quả sau đây: "**Mọi thuật toán sắp xếp chỉ sử dụng phép so sánh đều đòi hỏi thời gian  $\Omega(n \log n)$ .**"

#### 4.3.3. Mã Huffman (Huffman Code)

Giả sử có một văn bản trên bảng chữ cái  $C$ . Với mỗi chữ cái  $c \in C$ , ta biết tần suất xuất hiện của nó trong văn bản là  $f(c)$ . Cần tìm cách mã hóa văn bản sử dụng ít bộ nhớ nhất. Có hai loại mã hóa hay dùng:

– Mã hóa với độ dài cố định (fixed length code), dễ mã hóa cũng như dễ giải mã, nhưng lại đòi hỏi bộ nhớ lớn.

– Mã phi tiền tố (prefix free code) là cách mã hóa mỗi ký tự  $c$  bởi một xâu nhị phân  $code(c)$  sao cho mã của một ký tự bất kỳ không phải là đoạn đầu của bất cứ mã của ký tự nào trong số các ký tự còn lại. Loại mã hóa này tuy đòi hỏi việc mã hóa và giải mã phức tạp hơn nhưng thông thường tỏ ra cần ít bộ nhớ hơn.

##### Mã hóa độ dài cố định

Để mã hóa 26 chữ cái tiếng Anh bởi mã nhị phân độ dài cố định, độ dài của xâu tối thiểu là  $\lceil \log 26 \rceil = 5$  bit.

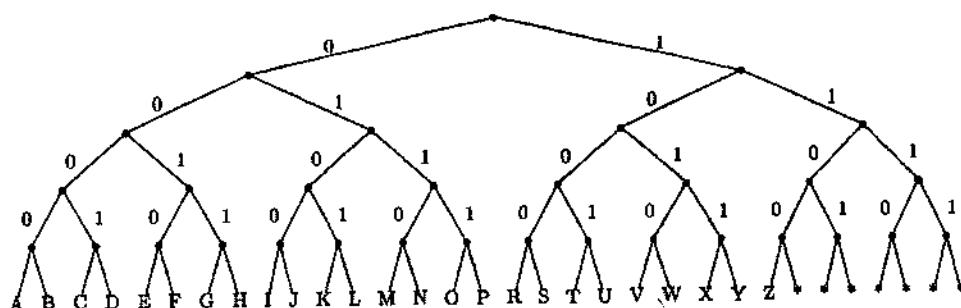
|       |   |       |   |       |   |       |   |
|-------|---|-------|---|-------|---|-------|---|
| 00001 | A | 01000 | H | 01111 | O | 10110 | V |
| 00010 | B | 01001 | I | 10000 | P | 10111 | W |
| 00011 | C | 01010 | J | 10001 | Q | 11000 | X |
| 00100 | D | 01011 | K | 10010 | R | 11001 | Y |
| 00101 | E | 01100 | L | 10011 | S | 11010 | Z |
| 00110 | F | 01101 | M | 10100 | T |       |   |
| 00111 | G | 01110 | N | 10101 | U |       |   |



David A. Huffman  
1925 – 1999

Các xâu từ 11011 đến 11111 không sử dụng.

## Cây mã hóa độ dài cố định



Mã hóa độ dài cố định là mã phi tiền tố.

### Morse Code

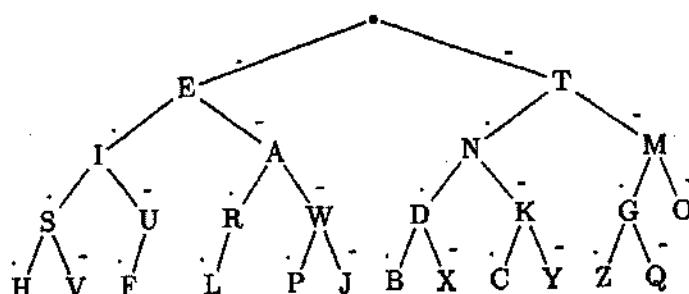
Căn cứ vào thống kê tần suất của các chữ cái trong tiếng Anh:

|               |        |
|---------------|--------|
| E             | 12,000 |
| T             | 9,000  |
| A, O, I, N, S | 8,000  |
| H             | 6,400  |
| R             | 6,200  |
| D             | 4,400  |
| L             | 4,000  |
| U             | 3,400  |

Morse đề xuất cách mã hóa:

|    |   |     |   |    |   |      |   |
|----|---|-----|---|----|---|------|---|
| .  | E | ..  | I | -  | T | ---  | M |
| -- | A | ... | S | -. | N | ...- | U |

### Cây mã hóa Morse



Mã hóa Morse không phải là phi tiền tố, do đó không thể giải mã. Ví dụ, việc giải mã xâu mã hóa không phải là duy nhất. Thông thường phải bổ sung dấu phân biệt các chữ cái, chẳng hạn, “..#..” → IU. Cây mã hóa là cây không đầy đủ.

### **Mã Huffman**

Mỗi mã phi tiền tố có thể biểu diễn bởi một cây nhị phân  $T$  mà mỗi lá của nó tương ứng với một chữ cái và cạnh của nó được gán cho một trong hai số 0 hoặc 1. Mã của một chữ cái  $c$  là một dãy nhị phân gồm các số gán cho các cạnh trên đường đi từ gốc đến lá tương ứng với  $c$ .

**Bài toán:** Tìm cách mã hóa tối ưu, tức là tìm cây nhị phân  $T$  làm tối thiểu hóa tổng độ dài có trọng số:

$$B(T) = \sum_{c \in C} f(c) \cdot \text{depth}(c),$$

trong đó:  $\text{depth}(c)$  là độ dài đường đi từ gốc đến lá tương ứng với ký tự  $c$ .

### **Ý tưởng thuật toán**

Chữ cái có tần suất nhỏ hơn cần được gán cho lá có khoảng cách đến gốc lớn hơn; chữ cái có tần suất xuất hiện lớn hơn cần được gán cho nút gần gốc hơn.

### **Mã Huffman: Thuật toán**

**procedure Huffman( $C, f$ );**

**begin**

$n \leftarrow |C|$ ;

$Q \leftarrow C$ ;

**for**  $i:=1$  to  $n-1$  **do**

**begin**

$x, y \leftarrow 2$  chữ cái có tần suất nhỏ nhất trong  $Q$ ; (\* Thao tác 1 \*)

            Tạo nút  $p$  với hai con  $x, y$ ;

$f(p) := f(x) + f(y)$ ;

$Q \leftarrow Q \setminus \{x, y\} \cup \{p\}$

(\* Thao tác 2 \*)

**end;**

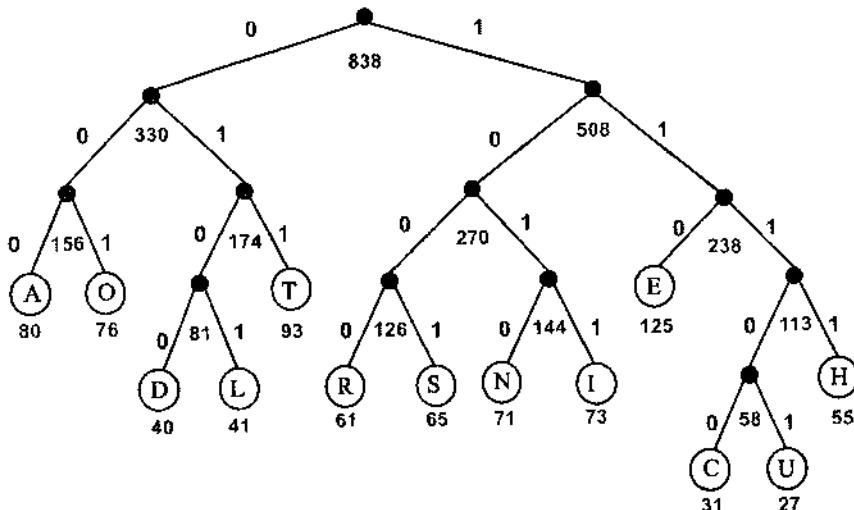
**end;**

Mã xây dựng theo thuật toán Huffman thường được gọi là mã Huffman (Huffman Code). Có thể cài đặt với thời gian  $O(n \log n)$  sử dụng priority queue để cài đặt  $Q$ .

**Ví dụ:** Xây dựng mã Huffman cho văn bản với tần suất của các ký tự trong văn bản cho trong bảng sau đây.

| Char | E   | T  | A  | O  | I  | N  | S  | R  | H  | L  | D  | C  | U  |
|------|-----|----|----|----|----|----|----|----|----|----|----|----|----|
| Freq | 125 | 93 | 80 | 76 | 72 | 71 | 65 | 61 | 55 | 41 | 40 | 31 | 27 |

Áp dụng thuật toán, ta thu được cây mã Huffman như trong hình vẽ sau đây:



Bảng mã của các chữ cái:

| Char | Freq | Fixed | Huff  |
|------|------|-------|-------|
| E    | 125  | 0000  | 110   |
| T    | 93   | 0001  | 011   |
| A    | 80   | 0010  | 000   |
| O    | 76   | 0011  | 001   |
| I    | 73   | 0100  | 1011  |
| N    | 71   | 0101  | 1010  |
| S    | 65   | 0110  | 1001  |
| R    | 61   | 0111  | 1000  |
| H    | 55   | 1000  | 1111  |
| L    | 41   | 1001  | 0101  |
| D    | 40   | 1010  | 0100  |
| C    | 31   | 1011  | 11100 |
| U    | 27   | 1100  | 11101 |
| Sum  | 838  | 3352  | 3036  |

### Mã Huffman: Giải mã

procedure Huffman\_Decode(B);

(\* B là xâu mã hóa văn bản theo mã hóa Huffman. \*)

begin

<Khởi động P là gốc của cây Huffman>

while < chưa đạt đến kết thúc của B> do

```

begin
 x ← bit tiếp theo trong xâu B;
 if x = 0 then P ← Con trái của P
 else P ← Con phải của P
 if (P là nút lá) then
 begin
 <Hiển thị ký hiệu tương ứng với nút lá>
 <Đặt lại P là gốc của cây Huffman>
 end;
 end;
end;

```

**Ví dụ:**

Giải mã xâu mã sau đây của đoạn văn bản được mã hóa theo mã Huffman vừa xây dựng ở trên:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Bắt đầu từ gốc, duyệt lần lượt từng ký tự của xâu mã, gấp 1 rẽ phải, gấp 0 rẽ trái, gấp lá đưa ra ký tự tương ứng với lá và quay lại từ gốc. Ta có:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

H

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|

E

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|

L

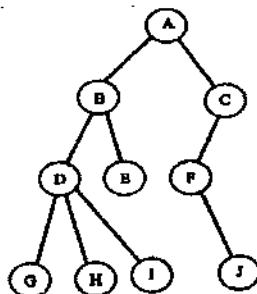
|   |   |   |
|---|---|---|
| 0 | 0 | 1 |
|---|---|---|

O

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| H |   |   |   | E |   |   |   | L |   |   |   | L |   |   |   | O |   |

## BÀI TẬP CHƯƠNG 4

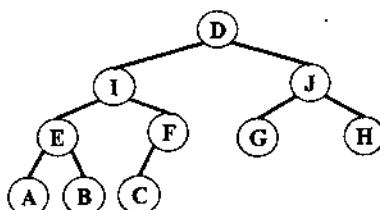
1. Xét cây cho trong hình sau:



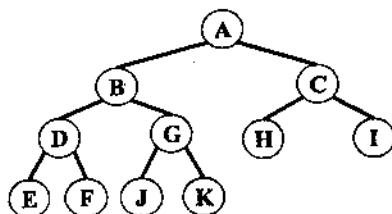
- a) Đưa ra thứ tự duyệt các nút của cây theo thứ tự trước.
  - b) Đưa ra thứ tự duyệt các nút của cây theo thứ tự giữa.
  - c) Đưa ra thứ tự duyệt các nút của cây theo thứ tự sau.
2. Vẽ cây nhị phân độ cao 3 mà duyệt theo thứ tự sau cho ta dãy khóa là:  
(10, 30, 50, 20, 40, 70, 60)

3. Đưa ra biểu diễn mảng của cây nhị phân hoàn chỉnh cho trong hình vẽ sau đây:

a)

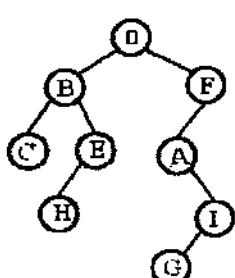


b)

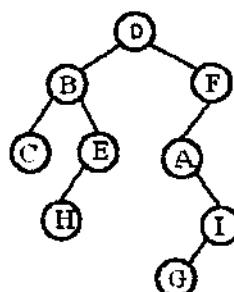


4. Cho cây nhị phân trong hình ở bài tập 1. Hãy đưa ra thứ tự các đỉnh xác định bởi duyệt cây theo thứ tự trước, thứ tự giữa và thứ tự sau.

a)

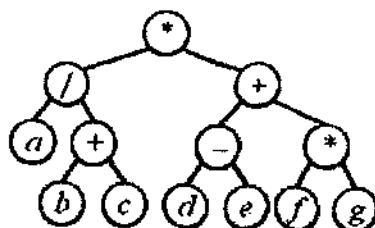


b)

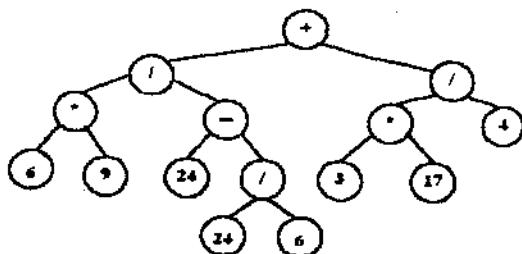


5. Hãy đưa ra các biểu thức số học mô tả bởi các cây biểu thức dưới đây trong ký pháp tiền tố, trung tố, hậu tố.

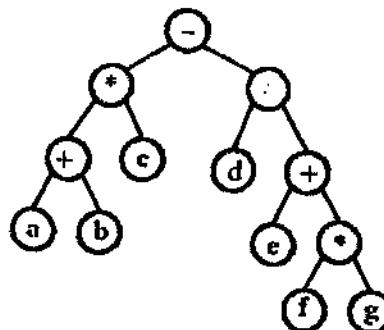
a)



b)



6. Cho cây biểu thức trong hình dưới đây:



- a) Hãy đưa ra biểu thức số học mô tả bởi cây đã cho trong ký pháp hậu tố.  
 b) Trình diễn thuật toán tính giá trị của biểu thức hậu tố cho trong câu 3a), trong đó:  $a = 5$ ,  $b = 10$ ,  $c = 7$ ,  $d = 20$ ,  $e = 5$ ,  $f = 3$ ,  $g = 5$ .

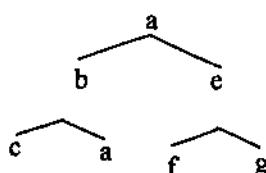
7. Xét cấu trúc dữ liệu mô tả cây nhị phân:

```

struct Node {
 char word;
 struct Node * left;
 struct Node * right; }

```

Hãy cho biết việc thực hiện thủ tục sau đây sẽ đưa ra kết quả gì, nếu x là con trỏ đến gốc của cây được cho trong hình dưới đây:

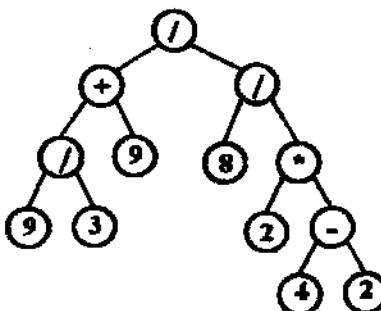


```

void t1(Node x) {
 if (x != null) {
 printf("%s", x->word);
 t1(x->left);
 t1(x->right);
 printf("%s", x->word);
 }
}

```

8. Cho cây biểu thức trong hình dưới:



- a) Hãy đưa ra biểu thức số học mô tả bởi cây đã cho trong ký pháp hậu tố.
- b) Trình diễn thuật toán tính giá trị của biểu thức hậu tố cho trong câu a)
9. Xây dựng cây mã Huffman để mã hóa một văn bản với tần suất xuất hiện của các ký hiệu được cho trong bảng thống kê sau đây:

| C | D  | E  | F | N  | O | R | Z  |
|---|----|----|---|----|---|---|----|
| 3 | 13 | 24 | 4 | 17 | 5 | 6 | 19 |

Đưa ra bảng mã của các ký hiệu và tổng số bít cần sử dụng để mã hóa văn bản.

10. Xây dựng cây mã Huffman để mã hóa một văn bản với tần suất xuất hiện của các ký hiệu được cho trong bảng thống kê sau đây (chỉ cần vẽ cây mã, không cho điểm việc trình bày các bước trung gian).

| C  | E   | F  | N  | R  | U  | Q | Y  |
|----|-----|----|----|----|----|---|----|
| 28 | 127 | 22 | 67 | 61 | 27 | 1 | 20 |

Đưa ra bảng mã của các ký hiệu và tổng số bít cần sử dụng để mã hóa văn bản.

11. Cho bảng mã Huffman của bảng chữ cái  $\Sigma = \{A, B, C, D, E\}$ .

| Chữ cái    | A  | B  | C  | D   | E   |
|------------|----|----|----|-----|-----|
| Mã Huffman | 11 | 01 | 00 | 101 | 100 |

- a) Vẽ cây mã Huffman sinh ra bảng mã ở trên.  
 b) Hãy giải mã xâu "001000111101001101100" được mã hóa theo bảng mã Huffman đã cho.

12. Xét cấu trúc dữ liệu trên C sau đây để mô tả cây nhị phân:

```
struct Tnode
{
 int key; // Dữ liệu của nút
 struct Tnode * left;
 struct Tnode * right;
};

typedef struct Tnode treeNode;
```

Hãy viết hàm trên C:

```
int EvenLeaf(treeNode *RootTree, int k);
trả lại số lượng nút lá với trường dữ liệu inf là số chẵn của cây với gốc được trả bởi RootTree.
```

13. Xét cấu trúc dữ liệu trên C sau đây để mô tả cây nhị phân:

```
struct Tnode
{
 int key; // Dữ liệu của nút
 struct Tnode * left;
 struct Tnode * right;
};

typedef struct Tnode treeNode;
```

Hãy viết hàm trên C:

```
int countNodes(treeNode *RootTree, int k);
```

trả lại số nút có trường key lớn hơn k trong cây được trả bởi RootTree.

14. Xét cấu trúc dữ liệu trên C sau đây để mô tả cây nhị phân:

```
struct Tnode
{
 int key; // Dữ liệu của nút
 struct Tnode * left;
```

```

 struct Tnode * right;
};

typedef struct Tnode treeNode;

```

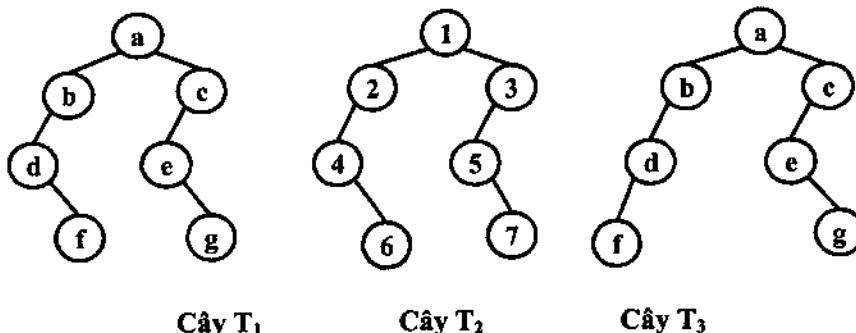
Hãy viết hàm trên C:

```
int countLeafs(treeNode *RootTree, int k);
```

trả lại số lượng lá của cây được trả bởi RootTree.

15. Hai cây nhị phân được gọi là *đẳng cấu* nếu như chúng có cấu trúc như nhau.

Ví dụ: xét ba cây T<sub>1</sub>, T<sub>2</sub> và T<sub>3</sub> cho trong hình vẽ dưới đây:



Khi đó ta có: hai cây T<sub>1</sub> và T<sub>2</sub> là *đẳng cấu*; hai cây T<sub>1</sub> và T<sub>2</sub> không *đẳng cấu* với cây T<sub>3</sub>.

Xét cấu trúc dữ liệu mô tả cây nhị phân:

```

struct Tnode {
 char word[20];
 struct Tnode *left;
 struct Tnode *right;
};

typedef struct Tnode BTNode;

```

Hãy viết hàm đệ quy trên C:

```
boolean isometric(BTNode *T1, BTNode *T2);
```

nhận đầu vào là hai cây nhị phân với gốc được trả bởi T1 và T2, trả lại giá trị true khi và chỉ khi hai cây này là *đẳng cấu*.

16. Giả sử cây nhị phân được mô tả bởi cấu trúc dữ liệu:

```
struct Node {
 int data; // Dữ liệu của nút
 struct Node * left;
 struct Node *right; };
typedef struct Node treeNode;
```

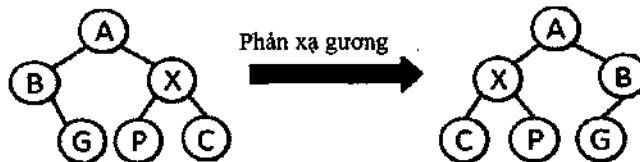
Hãy viết hàm trên C:

```
int OddSum(treeNode *root)
```

trả lại tổng của các số lẻ được cất giữ tại các nút của cây nhị phân với gốc được trả bởi root.

17. Ta gọi việc lấy phản xạ gương đối với cây nhị phân là việc hoán đổi cây con trái và cây con phải của mỗi nút trong cây đã cho.

Ví dụ:



Xét cấu trúc dữ liệu trên C để mô tả cây nhị phân sau đây:

```
struct TreeNode {
 struct TreeNode* leftPtr;
 struct TreeNode* rightPtr;
};
```

```
typedef struct TreeNode BTree;
```

Hãy viết hàm trên C:

```
BTree* mirror(BTree* nodePtr)
```

thực hiện việc lấy phản xạ gương đối với cây nhị phân được trả bởi nodeptr.

18. Giả sử cây nhị phân được mô tả bởi cấu trúc dữ liệu:

```
struct Node {
 int data; // Dữ liệu của nút
 struct Node * left;
```

```

 struct Node *right;);
typedef struct Node treeNode;

```

Hãy viết hàm trên C:

```
int Sum(treeNode *root);
```

trả lại giá trị là tổng của các giá trị số cất giữ tại các nút của cây nhị phân với gốc được trả bởi root.

19. Xét cấu trúc dữ liệu biểu diễn cây nhị phân:

```

struct TreeNode{
 int info;
 TreeNode * left;
 TreeNode * right;
};

```

a) Hãy viết hàm trên C:

```
int EvenMax(TreeNode *root)
```

nhận đầu vào root là con trỏ đến gốc của cây nhị phân tìm kiếm, trả lại giá trị info chẵn lớn nhất trong các nút của cây nhị phân đã cho.

- b) Gọi  $T(n)$  là thời gian tính của hàm trong câu 1a), hãy đưa ra đánh giá cho  $T(n)$ , trong đó  $n$  là số lượng nút của cây nhị phân đầu vào.

20. Xét cấu trúc dữ liệu mô tả cây nhị phân:

```

struct TreeNode {
 int info;
 TreeNode * left;
 TreeNode * right;);

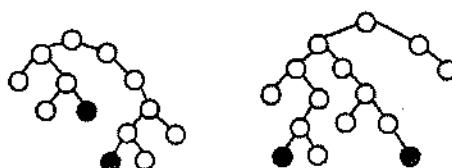
```

Viết hàm:

```
bool IsBST(TreeNode * root);
```

nhận đầu vào là con trỏ root đến gốc của cây, trả lại giá trị true khi và chỉ khi cây được trả bởi root là cây nhị phân cân bằng.

21. Ta gọi đường kính của cây là độ dài của đường đi dài nhất trong số các đường đi giữa hai lá bất kỳ của cây (độ dài của đường đi được xác định như số lượng nút trên nó).



Ví dụ, trong hình vẽ trên ta có hai cây đều có đường kính là 9, trong đó hai đầu mút của đường đi dài nhất được tô đậm. Đường đi dài nhất trong cây bên trái đi qua gốc, còn đường đi dài nhất trong cây bên phải không đi qua gốc.

Xét cấu trúc dữ liệu biểu diễn cây nhị phân:

```
struct TreeNode{
 int info;
 TreeNode * left;
 TreeNode * right;
};
```

a) Hãy viết hàm trên C:

```
int BT_Diam(TreeNode *root)
```

nhận đầu vào **root** là con trỏ đến gốc của cây nhị phân, trả lại đường kính của cây nhị phân đã cho.

b) Gọi  $T(n)$  là thời gian tính của hàm trong câu 1a), hãy đưa ra đánh giá cho  $T(n)$ , trong đó  $n$  là số lượng nút của cây nhị phân đầu vào.

22. Xét cấu trúc dữ liệu mô tả cây nhị phân:

```
struct TreeNode {
 int info;
 TreeNode * left;
 TreeNode * right; };
```

a) Viết hàm:

```
bool IsLeft(TreeNode * root, int key);
```

nhận đầu vào là con trỏ **root** đến gốc của cây, trả lại giá trị **true** khi và chỉ khi mọi nút của cây được trả bởi **root** đều có trường **info** nhỏ hơn **key**.

b) Ký hiệu  $n$  là số nút của cây. Đưa ra đánh giá thời gian tính của hàm **IsLeft** trong ký hiệu tiệm cận.

23. Xét cấu trúc dữ liệu mô tả cây nhị phân:

```
struct TreeNode {
 int info;
 struct TreeNode * left;
 struct TreeNode * right; };
```

a) Viết hàm:

```
bool IsLeft(TreeNode * root, int key);
```

nhận đầu vào là con trỏ **root** đến gốc của cây, trả lại giá trị **true** khi và chỉ khi mọi nút của cây được trả bởi **root** đều có trường **info** nhỏ hơn **key**.

b) Ký hiệu **n** là số nút của cây. Đưa ra đánh giá thời gian tính của hàm **IsLeft** trong ký hiệu tiệm cận.

24. Xét cấu trúc dữ liệu trên C sau đây để mô tả cây nhị phân:

```
struct Tnode
{ int key; // Dữ liệu của nút
 struct Tnode * left;
 struct Tnode * right;
};
typedef struct Tnode treeNode;
```

Hãy viết hàm trên C:

```
int EvenLeaf(treeNode *RootTree, int k);
```

trả lại số lượng nút lá với trường dữ liệu **inf** là số chẵn của cây với gốc được trả bởi **RootTree**.

# Chương 5

## CÁC THUẬT TOÁN SẮP XÉP

*D. Knuth: "40% thời gian hoạt động của các máy tính là dành cho sắp xếp!"*

### 5.1. BÀI TOÁN SẮP XÉP

#### 5.1.1. Bài toán sắp xếp

Sắp xếp (Sorting) là quá trình tổ chức lại họ các dữ liệu theo thứ tự giảm dần hoặc tăng dần (ascending or descending order).

Dữ liệu cần sắp xếp có thể là:

- Số nguyên (Integers);
- Xâu ký tự (Character strings);
- Đối tượng (Objects).

Khóa sắp xếp (Sort key) là bộ phận của bản ghi xác định thứ tự sắp xếp của bản ghi trong họ các bản ghi. Ta cần sắp xếp các bản ghi theo thứ tự của các khóa.

Chú ý:

- Việc sắp xếp tiến hành trực tiếp trên bản ghi đòi hỏi di chuyển vị trí bản ghi, có thể là thao tác rất tốn kém.
- Vì vậy, người ta thường xây dựng bảng khóa gồm các bản ghi chỉ có hai trường là (khóa, con trỏ):
  - + Trường "khóa" chứa giá trị khóa;
  - + Trường "con trỏ" để ghi địa chỉ của bản ghi tương ứng.
- Việc sắp xếp theo khóa trên bảng khóa không làm thay đổi bảng chính, nhưng trình tự các bản ghi trong bảng khóa cho phép xác định trình tự các bản ghi trong bảng chính.

Ta có thể hạn chế xét bài toán sắp xếp dưới dạng sau đây:

**Input:** Dãy  $n$  số  $A = (a_1, a_2, \dots, a_n)$ .

**Output:** Một hoán vị (sắp xếp lại)  $(a'_1, a'_2, \dots, a'_n)$  của dãy số đã cho thỏa mãn:

$$a'_1 \leq a'_2 \leq \dots \leq a'_n.$$

### Ứng dụng của sắp xếp

- Quản trị cơ sở dữ liệu (Database management);
- Khoa học và kỹ thuật (Science and engineering);
- Các thuật toán lập lịch (Scheduling algorithms), chặng hạn, thiết kế chương trình dịch (compiler design), truyền thông (telecommunication),...;
- Máy tìm kiếm web (Web search engine);

và nhiều ứng dụng khác...

### Các loại thuật toán sắp xếp

- Sắp xếp trong (internal sort): đòi hỏi họ dữ liệu được đưa toàn bộ vào bộ nhớ trong của máy tính.
- Sắp xếp ngoài (external sort): họ dữ liệu không thể cùng lúc đưa toàn bộ vào bộ nhớ trong, nhưng có thể đọc vào từng bộ phận từ bộ nhớ ngoài.

### Các đặc trưng của một thuật toán sắp xếp

- Tại chỗ (in place): nếu không gian nhớ phụ mà thuật toán đòi hỏi là  $O(1)$ , thì nghĩa là bị chặn bởi hằng số không phụ thuộc vào độ dài của dãy cần sắp xếp.
- Ôn định (stable): nếu các phần tử có cùng giá trị vẫn giữ nguyên thứ tự tương đối của chúng như trước khi sắp xếp.

### Có hai phép toán cơ bản mà thuật toán sắp xếp thường phải sử dụng

- Đổi chỗ (Swap): thời gian thực hiện là  $O(1)$ , chặng hạn sử dụng cài đặt trên C sau đây:

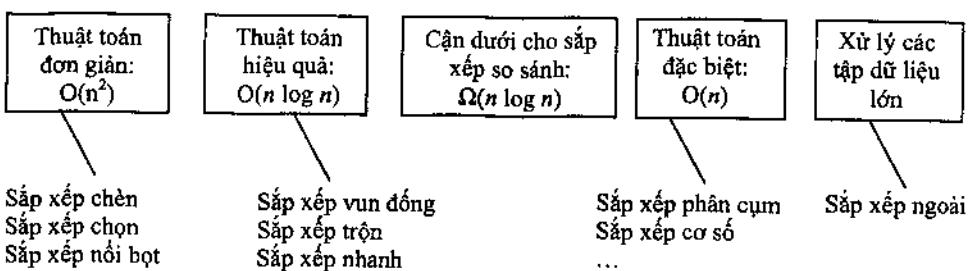
```
void swap(datatype &a, datatype &b) {
 datatype temp = a; //datatype-kiểu dữ liệu của phần tử
 a = b;
 b = temp;
}
```

- So sánh: hàm Compare(a, b) trả lại true nếu a đi trước b trong thứ tự cần sắp xếp và false nếu ngược lại.

Các thuật toán chỉ sử dụng phép toán so sánh để xác định thứ tự giữa hai phần tử được gọi là thuật toán sử dụng phép so sánh (*Comparison-based sorting algorithm*).

### 5.1.2. Giới thiệu sơ lược về các thuật toán sắp xếp

#### Phân loại các thuật toán sắp xếp



#### Các thuật toán sắp xếp dựa trên phép so sánh

| Thuật toán          | Trung bình (Average) | Tối nhất (Worst) | Bộ nhớ (Memory) | Ôn định (Stable) | Phương pháp (Method) |
|---------------------|----------------------|------------------|-----------------|------------------|----------------------|
| Bubble sort         | —                    | $O(n^2)$         | $O(1)$          | Có               | Hoán đổi             |
| Cocktail sort       | —                    | $O(n^2)$         | $O(1)$          | Có               | Hoán đổi             |
| Comb sort           | $O(n \log n)$        | $O(n \log n)$    | $O(1)$          | Không            | Hoán đổi             |
| Gnome sort          | —                    | $O(n^2)$         | $O(1)$          | Có               | Hoán đổi             |
| Selection sort      | $O(n^2)$             | $O(n^2)$         | $O(1)$          | Không            | Chọn                 |
| Insertion sort      | $O(n + d)$           | $O(n^2)$         | $O(1)$          | Có               | Chèn                 |
| Shell sort          | —                    | $O(n \log^2 n)$  | $O(1)$          | Không            | Chèn                 |
| Binary tree sort    | $O(n \log n)$        | $O(n \log n)$    | $O(n)$          | Có               | Chèn                 |
| Library sort        | $O(n \log n)$        | $O(n^2)$         | $O(n)$          | Có               | Chèn                 |
| Merge sort          | $O(n \log n)$        | $O(n \log n)$    | $O(n)$          | Có               | Trộn                 |
| In-place merge sort | $O(n \log n)$        | $O(n \log n)$    | $O(1)$          | Có               | Trộn                 |
| Heap sort           | $O(n \log n)$        | $O(n \log n)$    | $O(1)$          | Không            | Chọn                 |
| Smooth sort         | —                    | $O(n \log n)$    | $O(1)$          | Không            | Chọn                 |
| Quick sort          | $O(n \log n)$        | $O(n^2)$         | $O(\log n)$     | Không            | Phân chia            |
| Intro sort          | $O(n \log n)$        | $O(n \log n)$    | $O(\log n)$     | Không            | Lai                  |
| Patience sorting    | —                    | $O(n^2)$         | $O(n)$          | Không            | Chèn                 |

## Các thuật toán sắp xếp không chỉ dựa trên phép so sánh

| Name            | Average                  | Worst                            | Memory               | Stable | $n << 2^k?$ |
|-----------------|--------------------------|----------------------------------|----------------------|--------|-------------|
| Pigeonhole sort | $O(n+2^k)$               | $O(n+2^k)$                       | $O(2^k)$             | Có     | Có          |
| Bucket sort     | $O(n \cdot k)$           | $O(n^2 \cdot k)$                 | $O(n \cdot k)$       | Có     | Không       |
| Counting sort   | $O(n+2^k)$               | $O(n+2^k)$                       | $O(n+2^k)$           | Có     | Có          |
| LSD Radix sort  | $O(n \cdot k/s)$         | $O(n \cdot k/s)$                 | $O(n)$               | Có     | Không       |
| MSD Radix sort  | $O(n \cdot k/s)$         | $O(n \cdot (k/s) \cdot 2^s)$     | $O((k/s) \cdot 2^s)$ | Không  | Không       |
| Spread sort     | $O(n \cdot k / \log(n))$ | $O(n \cdot (k - \log(n)))^{0.5}$ | $O(n)$               | Không  | Không       |

Các thông số trong bảng:

$n$  – số phần tử cần sắp xếp,  $k$  – kích thước mỗi phần tử,

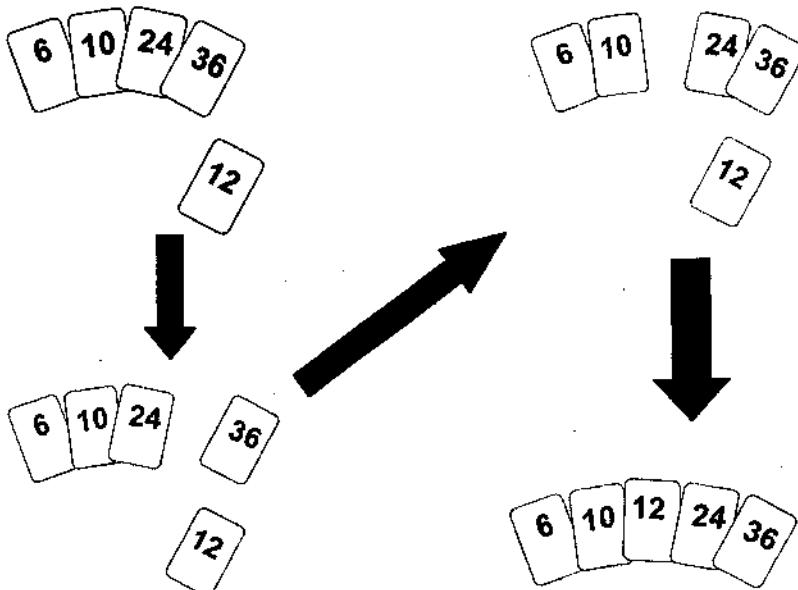
$s$  – kích thước bộ phận được sử dụng khi cài đặt.

Nhiều thuật toán được xây dựng dựa giả thiết  $n << 2^k$ .

## 5.2. BA THUẬT TOÁN SẮP XẾP CƠ BẢN

### 5.2.1. Sắp xếp chèn (Insertion Sort)

Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.



Để chèn 12, ta cần tạo chỗ cho nó bằng việc dịch chuyển đầu tiên là 36 và sau đó là 24.

### Thuật toán

- Tại bước  $k = 1, 2, \dots, n$ , đưa phần tử thứ  $k$  trong mảng đã cho vào đúng vị trí trong dãy gồm  $k$  phần tử đầu tiên.

- Kết quả là sau bước  $k$ ,  $k$  phần tử đầu tiên được sắp theo thứ tự.

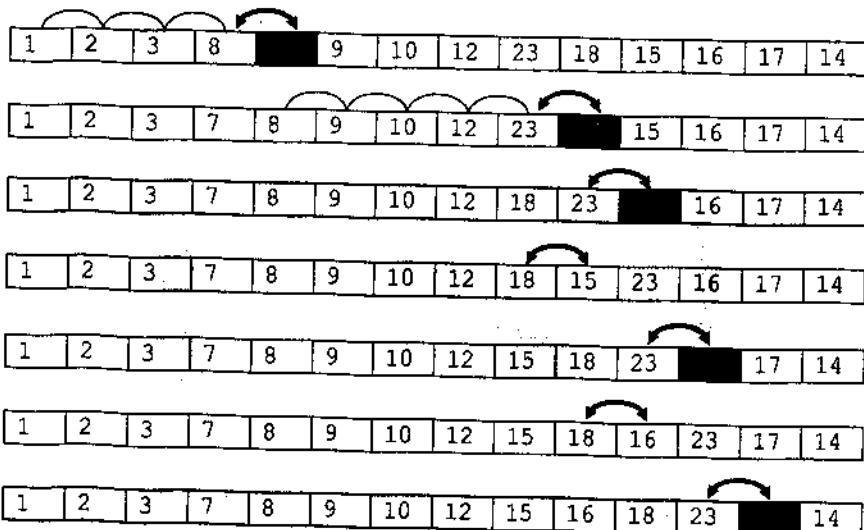
```
void insertionSort(int a[], int array_size) {
 int i, j, last;
 for (i=2; i < array_size; i++) {
 last = a[i];
 j = i;
 while ((j > 1) && (a[j-1] > last)) {
 a[j] = a[j-1];
 j = j - 1; }
 a[j] = last;
 } // end for
} // end of isort
```

Giải thích:

- Ở đầu lần lặp  $i$  của vòng "for" ngoài, dữ liệu từ  $a[1]$  đến  $a[i-1]$  là được sắp xếp.

- Vòng lặp "while" tìm vị trí cho phần tử tiếp theo ( $last = a[i]$ ) trong dãy gồm  $i$  phần tử đầu tiên.

Ví dụ: Insertion sort.



|   |   |   |   |   |   |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 7 | 8 | 9 | 10 | 12 | 15 | 16 | 18 | 17 | 23 | 14 |
| 1 | 2 | 3 | 7 | 8 | 9 | 10 | 12 | 15 | 16 | 17 | 18 | 23 |    |
| 1 | 2 | 3 | 7 | 8 | 9 | 10 | 12 | 15 | 16 | 17 | 18 | 14 | 23 |
| 1 | 2 | 3 | 7 | 8 | 9 | 10 | 12 | 15 | 16 | 17 | 14 | 18 | 23 |
| 1 | 2 | 3 | 7 | 8 | 9 | 10 | 12 | 15 | 16 | 14 | 17 | 18 | 23 |
| 1 | 2 | 3 | 7 | 8 | 9 | 10 | 12 | 15 | 14 | 16 | 17 | 18 | 23 |
| 1 | 2 | 3 | 7 | 8 | 9 | 10 | 12 | 14 | 15 | 16 | 17 | 18 | 23 |

13 phép đổi chỗ: ↘

20 phép so sánh: ↗

### Chứng minh tính đúng đắn của sắp xếp chèn

Bổ đề. Sau lần lặp  $i$  (từ 1 đến  $n$ ), các phần tử từ 0 đến  $i$  là được sắp theo thứ tự.

Chứng minh: Quy nạp theo  $i$ .

**Base case:** Khi  $i = 1$ , dãy gồm một phần tử  $a[1]$  là dãy được sắp.

**Inductive step:** Giả sử sau lần lặp  $i - 1$  ( $i > 1$ ), dãy  $a[1], \dots, a[i - 1]$  là được sắp theo thứ tự.

Ở lần lặp thứ  $i$  ta sẽ xếp phần tử mới  $a[i]$  (được gán vào last) vào đúng chỗ của nó trong dãy gồm  $i$  phần tử đầu tiên. Để ý rằng ta gán  $a[j]$  bằng  $a[j - 1]$  nếu  $a[j - 1]$  lớn hơn phần tử mới (last). Sau đó ta tiếp tục di chuyển sang trái, cho đến khi gặp phần tử đầu tiên nhỏ hơn last ( $a[j - 1] \leq \text{last}$ ) hoặc đạt đến vị trí phần tử đầu tiên của dãy ( $j = 1$ ) thì dừng và đưa phần tử mới (last) vào đúng chỗ ( $a[j] = \text{last}$ ). Theo giả thiết quy nạp, đoạn gồm  $i - 1$  phần tử đầu là được sắp theo thứ tự, còn theo lập luận vừa nêu thì phần tử mới thêm vào được xếp đúng chỗ, vì thế dãy thu được là được sắp theo thứ tự.

Bổ đề được chứng minh.

### Các đặc tính của Insertion Sort

– Sắp xếp chèn là tại chỗ và ổn định (In place and Stable).

– Phân tích thời gian tính của thuật toán:

+ *Best Case:* 0 hoán đổi,  $n - 1$  so sánh (khi dãy đầu vào đã được sắp).

+ *Worst Case:*  $n^2/2$  hoán đổi và so sánh (khi dãy đầu vào có thứ tự ngược lại với thứ tự cần sắp xếp).

+ *Average Case:*  $n^2/4$  hoán đổi và so sánh.

– Thuật toán này có thời gian tính trong tình huống tốt nhất là tốt nhất.

– Là thuật toán sắp xếp tốt đối với dãy đã gần được sắp xếp, nghĩa là mỗi phần tử đã đứng ở vị trí rất gần vị trí trong thứ tự cần sắp xếp.

### Thử nghiệm Insertion Sort

```
#include <stdlib.h>
#include <stdio.h>
void insertionSort(int a[], int array_size);
int a[1000];
int main() {
 int i, N;
 printf("\nGive n = "); scanf("%i", &N);
 srand(getpid()); // seed random number generator
 for (i = 0; i < N; i++) // fill array with random integers
 a[i] = rand();
 insertionSort(a, N); // perform insertion sort on array
 printf("\nOrdered sequence:\n");
 for (i = 0; i < N; i++)
 printf("%8i", a[i]);
 getch();
}
```

### 5.2.2. Sắp xếp lựa chọn (Selection Sort)

#### Thuật toán

- Tìm phần tử nhỏ nhất đưa vào vị trí 1.
- Tìm phần tử nhỏ tiếp theo đưa vào vị trí 2.
- Tìm phần tử nhỏ tiếp theo đưa vào vị trí 3.
- ...

```
void selectionSort(int a[], int n){
 int i, j, min, temp;
 for (i = 0; i < n-1; i++) {
 min = i;
 for (j = i+1; j < n; j++){
 if (a[j] < a[min]) min = j;
 }
 swap(a[i], a[min]);
 }
}
void swap(int &a,int &b)
{
```

```

int temp = a;
a = b;
b = temp;
}

```

### Phân tích thuật toán

– Best case: 0 đổi chỗ ( $n - 1$  như trong đoạn mã),  $n^2/2$  so sánh.

– Worst case:  $n - 1$  đổi chỗ và  $n^2/2$  so sánh.

– Average case:  $O(n)$  đổi chỗ và  $n^2/2$  so sánh.

– Ưu điểm nổi bật của sắp xếp chọn là số phép đổi chỗ ít. Điều này có ý nghĩa nếu như việc đổi chỗ là tốn kém.

### Ví dụ: Selection Sort.

| i=0  | 1    | 2    | 3    | 4    | 5    | 6    |
|------|------|------|------|------|------|------|
| 42 ← | 13   | 13   | 13   | 13   | 13   | 13   |
| 20   | 20 ← | 14   | 14   | 14   | 14   | 14   |
| 17   | 17   | 17 ← | 15   | 15   | 15   | 15   |
| 13 ← | 42   | 42   | 42 ← | 17   | 17   | 17   |
| 28   | 28   | 28   | 28   | 28 ← | 20   | 20   |
| 14   | 14 ← | 20   | 20   | 20 ← | 28 ← | 23   |
| 23   | 23   | 23   | 23   | 23   | 23 ← | 28 ← |
| 15   | 15   | 15 ← | 17 ← | 42   | 42   | 42   |

### 5.2.3. Sắp xếp nổi bọt (Bubble Sort)

Sắp xếp nổi bọt là phương pháp sắp xếp đơn giản thường được sử dụng như ví dụ minh họa trong các giáo trình nhập môn lập trình. Bắt đầu từ đầu dãy, thuật toán tiến hành so sánh mỗi phần tử với phần tử đi sau nó và thực hiện đổi chỗ, nếu chúng không theo đúng thứ tự. Quá trình này sẽ được lặp lại cho đến khi gặp lần duyệt từ đầu dãy đến cuối dãy mà không phải thực hiện đổi chỗ (tức là tất cả các phần tử đã đứng đúng vị trí). Cách làm này đã đẩy phần tử lớn nhất xuống cuối dãy, trong khi đó những phần tử có giá trị nhỏ hơn được dịch chuyển về đầu dãy.

Mặc dù thuật toán này đơn giản, nhưng nó là thuật toán kém hiệu quả nhất trong ba thuật toán cơ bản trình bày trong mục này. Vì thế, ngoài mục đích giảng dạy, sắp xếp nổi bọt rất ít khi được sử dụng. Giáo sư Owen Astrachan (Bộ môn Khoa học máy tính, Đại học Duke) còn đề nghị là không nên giảng dạy về thuật toán này.

Hàm trên C sau đây cài đặt thuật toán sắp xếp nổi bọt:

```

void bubbleSort(int a[], int n){
 int i, j;
 for (i = (n-1); i >= 0; i--) {
 for (j = 1; j <= i; j++) {

```

```

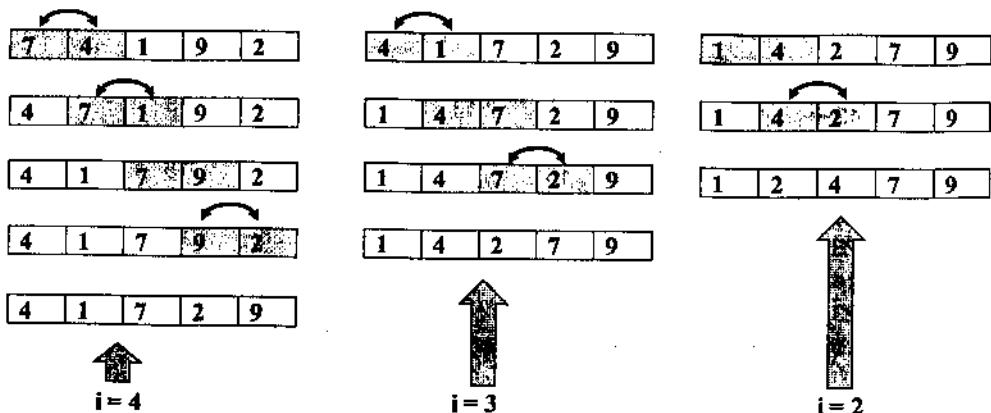
 if (a[j-1] > a[j])
 swap(a[j-1], a[j]);
 }
}

```

### Phân tích thuật toán

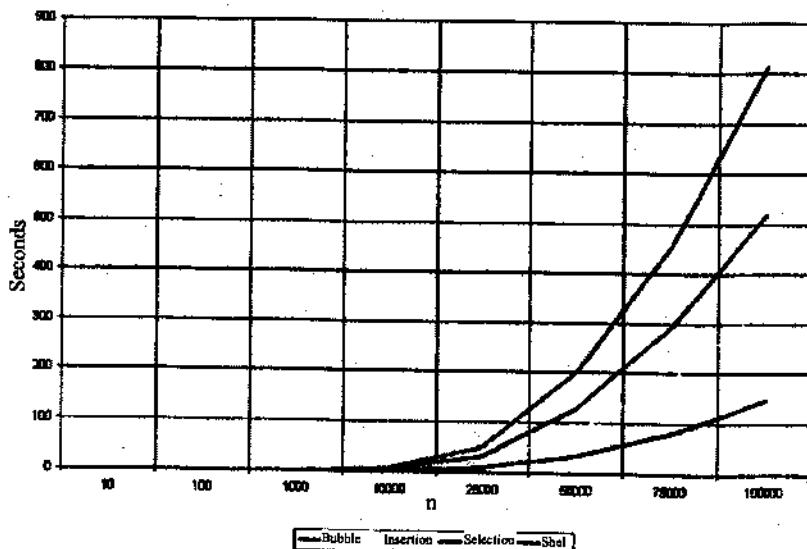
- Best case: 0 đổi chỗ,  $n^2/2$  so sánh.
- Worst case:  $n^2/2$  đổi chỗ và so sánh.
- Average case:  $n^2/4$  đổi chỗ và  $n^2/2$  so sánh.

### Ví dụ: Bubble Sort.



**Chú ý:** Các phần tử được đánh chỉ số bắt đầu từ 0,  $n = 5$ .

### So sánh ba thuật toán cơ bản



## Tổng kết ba thuật toán sắp xếp cơ bản

|                | Insertion     | Bubble        | Selection     |
|----------------|---------------|---------------|---------------|
| Best Case      | $\Theta(n)$   | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Average Case   | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Worst Case     | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Số lần đổi chỗ |               |               |               |
| Best Case      | 0             | 0             | $\Theta(n)$   |
| Average Case   | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n)$   |
| Worst Case     | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n)$   |

### 5.3. SẮP XẾP TRỘN (MERGE SORT)

**Bài toán:** Cần sắp xếp mảng A[1 ... n]. Thuật toán sắp xếp trộn được phát triển dựa trên chia để trị, bao gồm các thao tác sau:

#### - Chia (Divide)

Chia dãy gồm  $n$  phần tử cần sắp xếp ra thành hai dãy, mỗi dãy có  $n/2$  phần tử.

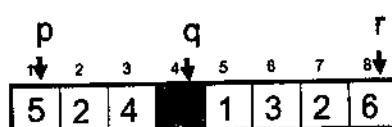
#### - Trị (Conquer)

Sắp xếp mỗi dãy con một cách đệ quy sử dụng *sắp xếp trộn*.

Khi dãy chỉ còn một phần tử thì trả lại phần tử này.

#### - Tổ hợp (Combine)

Trộn (Merge) hai dãy con được sắp xếp để thu được dãy được sắp xếp gồm tất cả các phần tử của cả hai dãy con.



**MERGE-SORT(A, p, r)**

if  $p < r$

// Kiểm tra điều kiện neo

then  $q \leftarrow \lfloor (p + r)/2 \rfloor$

// Chia (Divide)

```

 MERGE-SORT(A, p, q) // Trị (Conquer)
 MERGE-SORT(A, q + 1, r) // Trị (Conquer)
 MERGE(A, p, q, r) // Tô hợp (Combine)

endif

```

Lệnh gọi thực hiện thuật toán: MERGE-SORT(A, 1, n).

### Trộn (Merge)

#### MERGE(A, p, q, r)

1. Tính  $n_1$  và  $n_2$ .

2. Sao  $n_1$  phần tử đầu tiên vào  $L[1 \dots n_1]$  và  $n_2$  phần tử tiếp theo vào  $R[1 \dots n_2]$

3.  $L[n_1 + 1] \leftarrow \infty$ ;  $R[n_2 + 1] \leftarrow \infty$

4.  $i \leftarrow 1$ ;  $j \leftarrow 1$

5. **for**  $k \leftarrow p$  to  $r$  **do**

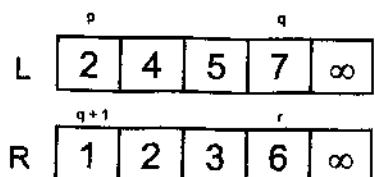
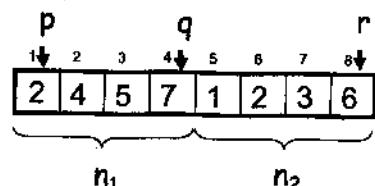
6.     **if**  $L[i] \leq R[j]$

7.       **then**  $A[k] \leftarrow L[i]$

8.           *i*  $\leftarrow i + 1$

9.       **else**  $A[k] \leftarrow R[j]$

*j*  $\leftarrow j + 1$



### Thời gian tính của trộn

– Khởi tạo (tạo hai mảng con tạm thời L và R):  $\Theta(n_1 + n_2) = \Theta(n)$ .

– Đưa các phần tử vào mảng kết quả (vòng lặp for cuối cùng): có  $n$  lần lặp, mỗi lần đòi hỏi thời gian hằng số, do đó thời gian cần thiết để thực hiện là  $\Theta(n)$ .

– Tổng cộng thời gian của trộn:  $\Theta(n)$ .

### Thời gian tính của sắp xếp trộn – MERGE SORT Running Time

– Chia: Tính  $q$  như là giá trị trung bình của  $p$  và  $r$ :  $D(n) = \Theta(1)$ .

– Trị: Giải đệ quy hai bài toán con, mỗi bài toán kích thước  $n/2 \Rightarrow 2T(n/2)$ .

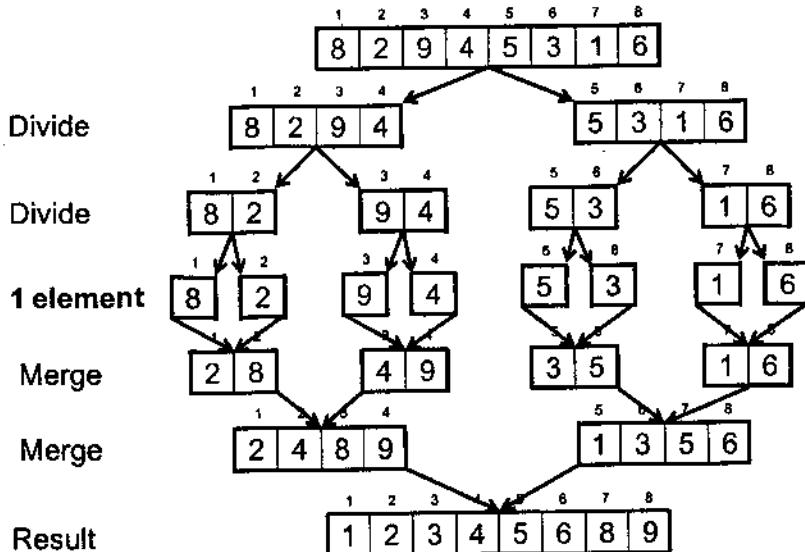
– Tô hợp: Trộn (MERGE) trên các mảng con cỡ  $n$  phần tử đòi hỏi thời gian  $\Theta(n) \Rightarrow C(n) = \Theta(n)$ .

Do đó ta có công thức đệ quy:

$$T(n) = \begin{cases} \Theta(1), & \text{nếu } n = 1 \\ 2T(n/2) + \Theta(n), & \text{nếu } n > 1 \end{cases}$$

Suy ra:  $T(n) = \Theta(n \log n)$  (chứng minh bằng quy nạp).

Ví dụ: Sắp xếp trộn.



Cài đặt Merge: Trộn A[first..mid] và A[mid+1.. last]

```

void merge(DataType A[], int first, int mid, int last){
 DataType tempA[MAX_SIZE]; // mảng phụ
 int first1 = first; int last1 = mid;
 int first2 = mid + 1; int last2 = last; int index =
first1;
 for (; first1 <= last1) && (first2 <= last2);
 ++index{
 if (A[first1] < A[first2])
 {
 tempA[index] = A[first1]; ++first1;
 }
 else
 { tempA[index] = A[first2]; ++first2; }
 for (; first1 <= last1; ++first1, ++index)
 tempA[index] = A[first1]; // sao nốt dãy con 1
 for (; first2 <= last2; ++first2, ++index)
 tempA[index] = A[first2]; // sao nốt dãy con 2
 for (index = first; index <= last; ++index)
 A[index] = tempA[index]; // sao trả mảng kết quả
 } // end merge
}

```

**Chú ý:** DataType – kiểu dữ liệu phần tử mảng.

```
void mergesort(DataType A[], int first, int last)
{
 if (first < last)
 {
 // chia thành hai dãy con
 int mid = (first + last) / 2; // chỉ số điểm giữa
 // sắp xếp dãy con trái A[first..mid]
 mergesort(A, first, mid);
 // sắp xếp dãy con phải A[mid+1..last]
 mergesort(A, mid+1, last);
 // Trộn hai dãy con
 merge(A, first, mid, last);
 } // end if
} // end mergesort
```

Lệnh gọi thực hiện merge sort(A, 0, n - 1).

## 5.4. SẮP XẾP NHANH (QUICK SORT)

### 5.4.1. Sơ đồ tổng quát

Thuật toán sắp xếp nhanh được phát triển bởi Hoare năm 1960 khi ông đang làm việc cho hãng máy tính nhỏ Elliott Brothers ở Anh. Theo thống kê tính toán, Quick Sort (viết tắt là QS) là thuật toán sắp xếp nhanh nhất hiện nay. QS có thời gian tính trung bình là  $O(n \log n)$ , tuy nhiên thời gian tính tồi nhất của nó lại là  $O(n^2)$ . QS là thuật toán sắp xếp tại chỗ, nhưng nó không có tính ổn định. QS khá đơn giản về lý thuyết, nhưng lại không dễ cài đặt.

Quick Sort là thuật toán sắp xếp được phát triển dựa trên kỹ thuật chia để trị. Thuật toán có thể mô tả đê quy như sau (có dạng tương tự như Merge Sort):

1. Neo đê quy (Base case): nếu dãy chỉ còn không quá một phần tử thì nó là dãy được sắp và trả lại ngay dãy này mà không phải làm gì cả.
2. Chia (Divide):
  - Chọn một phần tử trong dãy và gọi nó là phần tử chốt p (pivot).



C.A.R.Hoare

January 11, 1934

ACM Turing Award, 1980

Photo: 2006

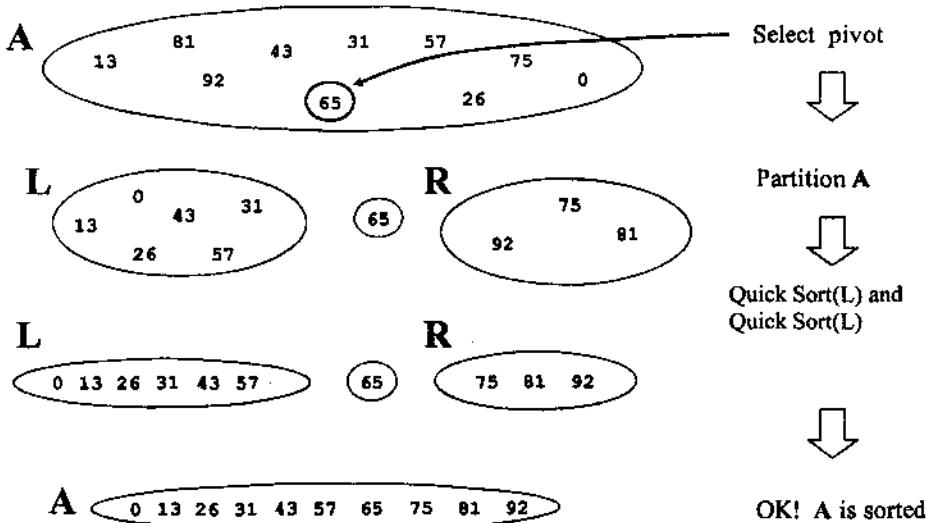
– Chia dãy đã cho ra thành hai dãy con: dãy con trái ( $L$ ) gồm những phần tử không lớn hơn phần tử chốt, còn dãy con phải ( $R$ ) gồm các phần tử không nhỏ hơn phần tử chốt. Thao tác này được gọi là "Phân đoạn" (Partition).

3. Trị (Conquer): Lặp lại một cách đệ quy thuật toán đối với hai dãy con  $L$  và  $R$ .

4. Tích hợp (Combine): Dãy được sắp xếp là  $L \ p \ R$ .

Ngược lại với Merge Sort, trong QS thao tác chia phức tạp, nhưng thao tác tổng hợp lại đơn giản. Điểm mấu chốt để thực hiện QS chính là thao tác chia. Phụ thuộc vào thuật toán thực hiện thao tác này mà ta có các dạng QS cụ thể.

### Các bước của Quick Sort



Sơ đồ tổng quát của QS có thể mô tả như sau:

**Quick Sort( $A, Left, Right$ )**

1. if ( $Left < Right$ ) {
2.  $Pivot = Partition(A, Left, Right);$
3.  $Quick\ Sort(A, Left, Pivot - 1);$
4.  $Quick\ Sort(A, Pivot + 1, Right)\}.$

Hàm  $Partition(A, Left, Right)$  thực hiện chia  $A[Left..Right]$  thành hai đoạn  $A[Left..Pivot - 1]$  và  $A[Pivot+1..Right]$  sao cho:

- Các phần tử trong  $A[Left..Pivot - 1]$  nhỏ hơn hoặc bằng  $A[Pivot]$
- Các phần tử trong  $A[Pivot+1..Right]$  lớn hơn hoặc bằng  $A[Pivot]$ .

Lệnh gọi thực hiện thuật toán Quick Sort ( $A, 1, n$ ).

Knuth cho rằng khi dãy con chỉ còn một số lượng không lớn phần tử (theo ông là không quá 9 phần tử) thì ta nên sử dụng các thuật toán đơn giản để sắp xếp dãy này, chứ không nên tiếp tục chia nhỏ. Thuật toán trong tình huống như vậy có thể mô tả như sau:

#### Quick Sort ( $A$ , $Left$ , $Right$ )

1. if ( $Right - Left < n_0$ )
2.     Insertion\_sort( $A$ ,  $Left$ ,  $Right$ );
3. else {
4.     *Pivot* = Partition( $A$ ,  $Left$ ,  $Right$ );
5.     Quick Sort ( $A$ ,  $Left$ ,  $Pivot - 1$ );
6.     Quick Sort ( $A$ ,  $Pivot + 1$ ,  $Right$ ).}

#### 5.4.2. Phép phân đoạn

Trong QS thao tác chia bao gồm hai công việc:

- Chọn phần tử chốt  $p$ .
- Phân đoạn: Chia dãy đã cho ra thành hai dãy con: dãy con trái ( $L$ ) gồm những phần tử không lớn hơn phần tử chốt, còn dãy con phải ( $R$ ) gồm các phần tử không nhỏ hơn phần tử chốt.

Thao tác phân đoạn có thể cài đặt (tại chỗ) với thời gian  $\Theta(n)$ . Hiệu quả của thuật toán phụ thuộc rất nhiều vào việc phần tử nào được chọn làm phần tử chốt.

- Thời gian tinh trong tình huống tồi nhất của QS là  $O(n^2)$ . Trường hợp xấu nhất xảy ra khi danh sách đã được sắp xếp và phần tử chốt được chọn là phần tử trái nhất của dãy.

– Nếu phần tử chốt được chọn ngẫu nhiên, thì QS có độ phức tạp tính toán là  $O(n \log n)$ .

#### Chọn phần tử chốt

Việc chọn phần tử chốt có vai trò quyết định đối với hiệu quả của thuật toán. Tốt nhất nếu chọn được phần tử chốt là phần tử đứng giữa trong danh sách được sắp xếp (ta gọi phần tử như vậy là *trung vị/median*). Khi đó, sau  $\log_2 n$  lần phân đoạn, ta sẽ đạt tới danh sách với kích thước bằng 1. Tuy nhiên, điều đó rất khó thực hiện. Người ta thường sử dụng các cách chọn phần tử chốt sau đây:

- Chọn phần tử trái nhất (đứng đầu) làm phần tử chốt.
- Chọn phần tử phải nhất (đứng cuối) làm phần tử chốt.
- Chọn phần tử đứng giữa danh sách làm phần tử chốt.
- Chọn phần tử trung vị trong ba phần tử đứng đầu, đứng giữa và đứng cuối làm phần tử chốt (Knuth).
- Chọn ngẫu nhiên một phần tử làm phần tử chốt.

## Thuật toán phân đoạn

Ta xây dựng hàm Partition(a, left, right) làm việc sau:

**Input:** Mảng  $a[left .. right]$ .

**Output:** Phân bố lại các phần tử của mảng đầu vào dựa vào phần tử  $pivot$  được chọn và trả lại chỉ số  $jpivot$  thỏa mãn:

- +  $a[jpivot]$  chứa giá trị của  $pivot$ ;
- +  $a[i] \leq a[jpivot]$ , với mọi  $left \leq i < pivot$ ;
- +  $a[j] \geq a[jpivot]$ , với mọi  $pivot < j \leq right$ .

trong đó:  $pivot$  là giá trị phần tử chốt được chọn.

Ta xét thuật toán thực hiện thao tác phân đoạn với các cách chọn phần tử chốt khác nhau.

**Phần tử chốt là phần tử đứng đầu**

**Partition(a, left, right)**

```
i = left; j = right + 1; pivot = a[left];
while i < j do i = i + 1;
while i ≤ right and a[i] < pivot do i = i + 1;
j = j - 1;
while j ≥ left and a[j] > pivot do j = j - 1;
swap(a[i], a[j]);
swap(a[i], a[j]); swap(a[j], a[left]);
return j;
```

**Phần tử chốt là phần tử đứng giữa**

**PartitionMid(a, left, right);**

```
i = left; j = right; pivot = a[(left + right)/2];
repeat
 while a[i] < pivot do i = i + 1;
 while pivot < a[j] do j = j - 1;
 if i <= j
 swap(a[i], a[j]);
 i = i + 1; j = j - 1;
 until i > j;
return j;
```

Cài đặt thuật toán phân đoạn trong đó pivot được chọn là phần tử đứng giữa là cách cài đặt mà TURBO C lựa chọn.

Cài đặt trên C thao tác phân đoạn với phần tử chốt là phần tử đứng cuối.

```

int PartitionR(int a[], int p, int r) {
 int x = a[r];
 int j = p - 1;
 for (int i = p; i < r; i++) {
 if (x >= a[i])
 { j = j + 1; swap(a[i], a[j]); }
 }
 a[r] = a[j + 1]; a[j + 1] = x;
 return (j + 1);
}

```

### Cài đặt Quick Sort

```

void swap(int &a,int &b)
{ int t = a;
 a = b;
 b = t;
}

int Partition(int a[], int L, int R)
{
 int i, j, p;
 i = L; j = R + 1; p = a[L];
 while (i < j) {
 i = i + 1;
 while ((i <= R)&&(a[i]<p)) i++;
 j--;
 while ((j >= L)&& (a[j]>p)) j--;
 swap(a[i] , a[j]);
 }
 swap(a[i], a[j]); swap(a[j], a[L]);
 return j;
}

void quick_sort(int a[], int left, int right)
{
 int p;

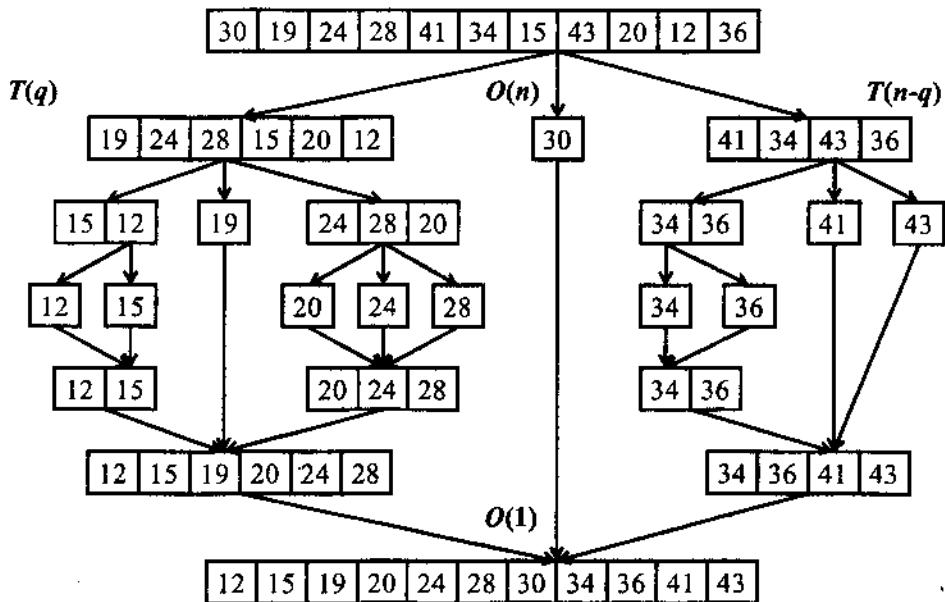
```

```

if (left < right)
{
 pivot = Partition(a, left, right);
 if (left < pivot)
 quick_sort(a, left, pivot-1);
 if (right > pivot)
 quick_sort(a, pivot+1, right); }
}

```

Ví dụ: Sơ đồ thực hiện sắp xếp nhanh



### 5.4.3. Độ phức tạp của sắp xếp nhanh

Thời gian tính của Quick Sort phụ thuộc vào việc phép phân chia là *cân bằng* (*balanced*) hay *không cân bằng* (*unbalanced*) và điều đó lại phụ thuộc vào việc phần tử nào được chọn làm chốt.

1. Phân đoạn không cân bằng: thực sự không có phần nào cả, do đó một bài toán con có kích thước  $n - 1$  còn bài toán kia có kích thước 0.
2. Phân đoạn hoàn hảo (Perfect partition): việc phân đoạn luôn được thực hiện dưới dạng phân đôi, như vậy mỗi bài toán con có kích thước  $c/n/2$ .
3. Phân đoạn cân bằng: việc phân đoạn được thực hiện ở đâu đó quanh điểm giữa, nghĩa là một bài toán con có kích thước  $n - k$  còn bài toán kia có kích thước  $k$ .

Ta sẽ xét từng tình huống.

### Phân đoạn không cân bằng (Unbalanced partition)

Công thức đệ quy cho thời gian tính trong tình huống này là:

$$T(n) = T(n - 1) + T(0) + \Theta(n),$$

$$T(0) = T(1) = 1.$$

Ta có:

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

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

$$T(n - 2) = T(n - 3) + (n - 2);$$

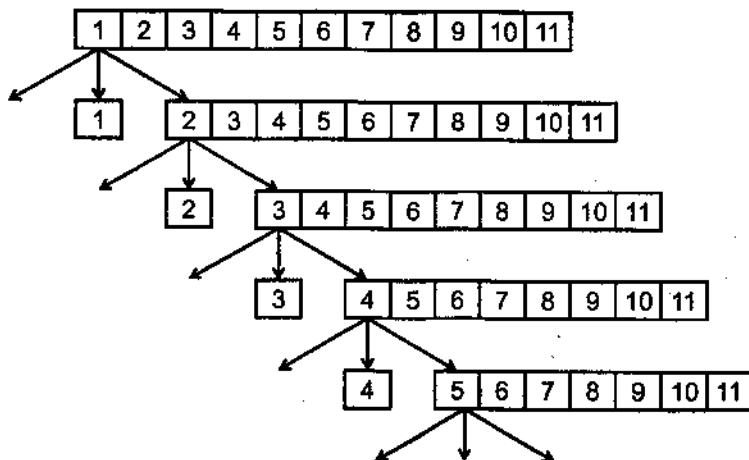
...

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

Cộng vế với vế các đẳng thức trên, sau khi đơn giản biểu thức ta thu được:

$$T(n) = T(1) + \sum_{i=2}^n i = O(n^2).$$

Ví dụ, khi phần tử chốt được chọn là phần tử đứng đầu, tình huống tồi nhất này xảy ra khi dãy đầu vào đã được sắp xếp được minh họa trong hình vẽ sau đây:



Công thức đệ quy cho thời gian tính trong tình huống này là:

$$T(0) = T(1) = 1,$$

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

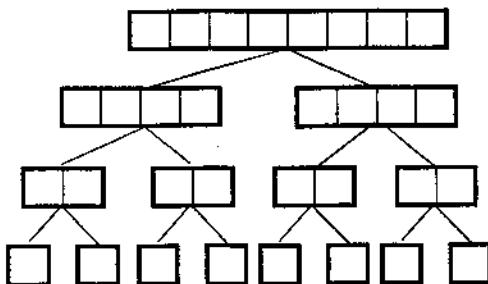
Từ đó,  $T(n) = \Theta(n^2)$ .

### Phân đoạn hoàn hảo – Perfect partition

Công thức đệ quy cho thời gian tính trong tình huống này là:

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + \Theta(n) \\ &= 2T(n/2) + \Theta(n). \end{aligned}$$

Từ đó,  $T(n) = \Theta(n \log n)$  (chứng minh bằng quy nạp toán học).



### Tình huống tổng quát

Ta có công thức đệ quy cho thời gian tính trong tình huống tổng quát là:

$$T(n) = T(q) + T(n - q) + \Theta(n).$$

Để đánh giá thuật toán trong tình huống xấu nhất, ta cần tìm:

$$T(n) = \max_{1 \leq q < n} \{ T(q) + T(n - q) \} + \Theta(n)$$

**Mệnh đề:**  $T(n) \leq cn^2 = O(n^2)$ .

**Chứng minh:** Quy nạp theo  $n$ .

**Base case:** Rõ ràng bồ đề đúng cho  $n = 1$ .

**Induction step:** Giả sử bồ đề đúng với mọi  $n < n'$ , ta chứng minh nó đúng cho  $n'$ .

Ta có:

$$\begin{aligned} T(n') &= \max_{1 \leq q < n'} \{ T(q) + T(n' - q) \} + \Theta(n') \\ &\leq cq^2 + c(n' - q)^2 + dn' \\ &= 2cq^2 + c(n')^2 - 2cqn' + dn' \end{aligned}$$

Để hoàn thành chứng minh, ta cần chỉ ra rằng:

$$2cq^2 + c(n')^2 - 2cqn' + dn' \leq c(n')^2,$$

và bất đẳng thức này tương đương với:

$$dn' \leq 2cq(n' - q)$$

Do  $q(n' - q)$  luôn lớn hơn  $n'/2$  (để thấy điều này là đúng có thể xét hai tình huống:  $q < n/2$  và  $n \geq q \geq n/2$ ), nên ta có thể chọn  $c$  đủ lớn để bất đẳng thức cuối cùng là đúng.

Mệnh đề được chứng minh.

Do trong tình huống phân đoạn không cân bằng, QS có thời gian tính  $\Theta(n^2)$ , nên từ mệnh đề suy ra thời gian tính trong tình huống tồi nhất của QS là  $O(n^2)$ .

### Thời gian tính trung bình của QS – Quick Sort Average Case

Giả sử rằng pivot được chọn ngẫu nhiên trong số các phần tử của dãy đầu vào. Tất cả các tình huống sau đây là đồng khả năng:

- Pivot là phần tử nhỏ nhất trong dãy.
- Pivot là phần tử nhỏ nhì trong dãy.
- ...
- Pivot là phần tử lớn nhất trong dãy.

Điều đó cũng là đúng khi pivot luôn được chọn là phần tử đầu tiên, với giả thiết dãy đầu vào là hoàn toàn ngẫu nhiên. Khi đó:

Thời gian tính trung bình =  $\Sigma$  (thời gian phân đoạn kích thước i)  $\times$  (xác suất phân đoạn có kích thước i).

Trong tình huống ngẫu nhiên, tất cả các kích thước là đồng khả năng, xác suất bằng  $1/n$ , do đó:

$$\begin{aligned}T(n) &= T(i) + T(n-i-1) + cn \\E(T(n)) &= \sum_{i=0}^{n-1} (1/n) [E(T(i)) + E(T(n-i-1)) + cn] \\E(T(n)) &\leq (2/n) \sum_{i=0}^{n-1} [E(T(i)) + cn]\end{aligned}$$

Giải công thức đệ quy này ta thu được:  $E(T(n)) = O(n \log n)$ .

## 5.5. SẮP XẾP VŨN ĐỒNG (HEAP SORT)

### 5.5.1. Cấu trúc dữ liệu đồng (Heap)

**Định nghĩa:** Đồng (heap) là cây nhị phân gần hoàn chỉnh có hai tính chất sau:

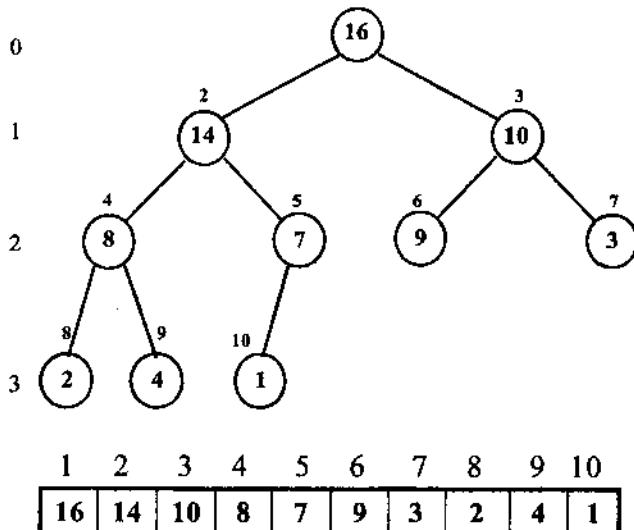
- Tính cấu trúc (Structural property): tất cả các mức đều là đầy, ngoại trừ mức cuối cùng, mức cuối được diền từ trái sang phải.
- Tính có thứ tự hay tính chất đồng (heap property): với mỗi nút x:

$$\text{Parent}(x) \geq x.$$

Đồng được cài đặt bởi mảng  $A[i]$  có độ dài  $length[A]$ . Số lượng phần tử là  $heapsize[A]$ . Từ tính chất đồng suy ra: “Gốc chứa phần tử lớn nhất của đồng!”. Như vậy có thể nói: Đồng là cây nhị phân được diền theo thứ tự.

### 5.5.1.1. Biểu diễn đống bởi mảng

Đống có thể cất giữ trong mảng A. Hình vẽ dưới đây minh họa cho đống:



Khi đó:

- Gốc của cây là  $A[1]$ ,
- Con trái của  $A[i]$  là  $A[2*i]$ ,
- Con phải của  $A[i]$  là  $A[2*i + 1]$ ,
- Cha của  $A[i]$  là  $A[\lfloor i/2 \rfloor]$ ,
- Heapsize[A] ≤ length[A],
- Các phần tử trong mảng con  $A[\lfloor n/2 \rfloor + 1] \dots n]$  là các lá.

Các hàm cơ bản được cài đặt dễ dàng. Ta có:

$$\text{parent}(i) = \lfloor i/2 \rfloor,$$

$$\text{left-child}(i) = 2i,$$

$$\text{right-child}(i) = 2i + 1.$$

### 5.5.1.2. Hai dạng đống

Đống max – Max-heaps (Phần tử lớn nhất ở gốc), có tính chất *max-heap*:

Với mọi nút i, ngoại trừ gốc:  $A[\text{parent}(i)] \geq A[i]$ .

Đống min – Min-heaps (phần tử nhỏ nhất ở gốc), có tính chất *min-heap*:

Với mọi nút i, ngoại trừ gốc:  $A[\text{parent}(i)] \leq A[i]$ .

Phản dưới đây ta sẽ chỉ xét đống max (max-heap). Đống min được xét hoàn toàn tương tự.

### **5.5.1.3. Các phép toán đối với đồng**

#### **Bổ sung và loại bỏ nút**

- Nút mới được bổ sung vào mức đáy (từ trái sang phải).
- Các nút được loại bỏ khỏi mức đáy (từ phải sang trái).

#### **Các phép toán đối với đồng**

- Khôi phục tính chất max-heap (Vun lại đồng): Max-Heapify.
- Tạo max-heap từ một mảng không được sắp xếp: Build-Max-Heap.

#### **Khôi phục tính chất đồng**

Giả sử có nút  $i$  với giá trị bé hơn con của nó. Giả thiết: cây con trái và cây con phải của  $i$  đều là max-heaps. Đề loại bỏ sự vi phạm này ta tiến hành như sau:

- Đổi chỗ với con lớn hơn;
- Di chuyển xuống theo cây;
- Tiếp tục quá trình cho đến khi nút không còn bé hơn con.

#### **Thuật toán khôi phục tính chất đồng**

Giả thiết: Cả hai cây con trái và phải của  $i$  đều là max-heaps. Chú ý là  $A[i]$  có thể bé hơn các con của nó.

##### **Max-Heapify ( $A, i, n$ )**

//  $n = \text{heapsize}[A]$

1.  $\text{left} \leftarrow \text{left-child}(i)$
2.  $\text{right} \leftarrow \text{right-child}(i)$
3. **if** ( $\text{left} \leq n$ ) and ( $A[\text{left}] > A[i]$ )
4.   **then**  $\text{largest} \leftarrow \text{left}$
5.   **else**  $\text{largest} \leftarrow i$
6. **if** ( $\text{right} \leq n$ ) and ( $A[\text{right}] > A[\text{largest}]$ )
7.   **then**  $\text{largest} \leftarrow \text{right}$
8. **if**  $\text{largest} != i$
9.   **then** Exchange( $A[i], A[\text{largest}]$ )
10. **Max-Heapify**( $A, \text{largest}$ )

#### **Thời gian tính của MAX-HEAPIFY**

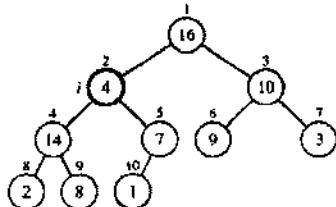
Nhận thấy rằng:

- Từ nút  $i$  phải di chuyển theo đường đi xuống phía dưới của cây. Độ dài của đường đi này không vượt quá độ dài đường đi từ gốc đến lá, nghĩa là không vượt quá  $h$ .
- Ở mỗi mức phải thực hiện hai phép so sánh.
- Do đó tổng số phép so sánh không vượt quá  $2h$ .

– Vậy, thời gian tính là  $O(h)$  hay  $O(\log n)$ .

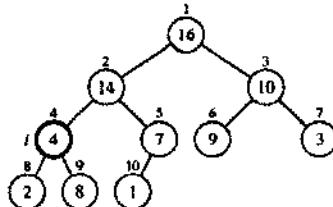
Do đó, thời gian tính của MAX-HEAPIFY là  $O(\log n)$ . Nếu viết trong ngôn ngữ chiêu cao của đồng thì thời gian này là  $O(h)$ .

Ví dụ:



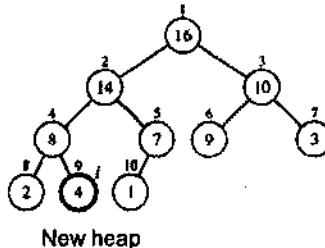
A[2] violates a heap property

$A[2] \leftrightarrow A[4]$



A[4] A[2] violates a heap property

$A[4] \leftrightarrow A[9]$



New heap

### Xây dựng đồng

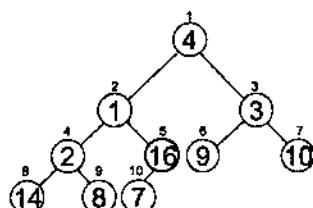
Vấn đề đặt ra là cần biến đổi mảng  $A[1 \dots n]$  thành max-heap ( $n = \text{length}[A]$ ). Vì các phần tử của mảng con  $A[\lfloor n/2 \rfloor + 1 \dots n]$  là các lá, do đó để tạo đồng ta chỉ cần áp dụng MAX-HEAPIFY đối với các phần tử từ 1 đến  $\lfloor n/2 \rfloor$ .

#### Algorithm: Build-Max-Heap(A)

1.  $n = \text{length}[A]$
2.  $\text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto } 1$
3.     do Max-Heappify( $A, i, n$ )

#### Alg: Build-Max-Heap(A)

1.  $n = \text{length}[A]$
2.  $\text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto } 1$
3.     do Max-Heappify( $A, i, n$ )



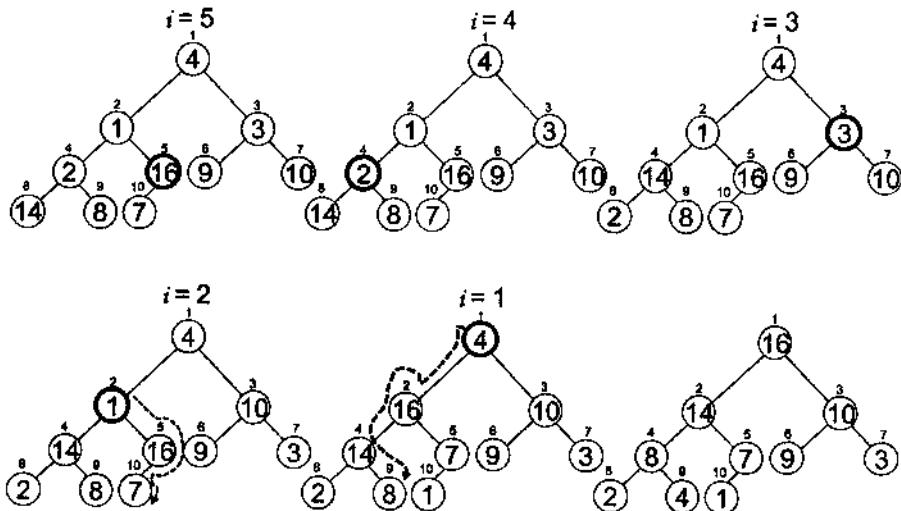
A: 

|   |   |   |   |    |   |    |    |   |   |
|---|---|---|---|----|---|----|----|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|----|---|----|----|---|---|

**Ví dụ:** Xét mảng A sau đây:

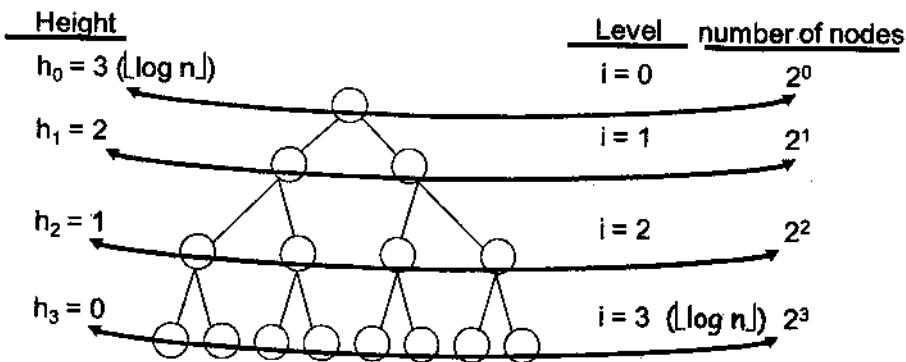
|   |   |   |   |    |   |    |    |   |   |
|---|---|---|---|----|---|----|----|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|----|---|----|----|---|---|

Chi cần áp dụng MAX-HEAPIFY đối với các phần tử từ 1 đến  $\lfloor 10/2 \rfloor = 5$ .



### Thời gian tính của Buil-Max-Heap

Heapify đòi hỏi thời gian  $O(h)$   $\Rightarrow$  chi phí của Heapify ở nút  $i$  tỷ lệ với chiều cao của nút  $i$  trên cây như minh họa trong hình sau đây:



Ký hiệu trong hình vẽ:

- $- h_i = h - i$ : chiều cao của đồng gốc tại mức  $i$ .
- $- n_i = 2^i$ : số lượng nút ở mức  $i$ .

Suy ra:

$$\begin{aligned} T(n) &= \sum_{i=0}^h n_i h_i && (\text{Chi phí Heapify tại mức } i) \quad (\text{số lượng nút trên mức này}) \\ &= \sum_{i=0}^h 2^i (h-i) && \text{Thay giá trị của } n_i \text{ và } h_i \text{ tính được ở trên} \\ &= \sum_{i=0}^h \frac{h-i}{2^{h-i}} 2^h && \text{Nhân cả tử và mẫu với } 2^h \text{ và viết } 2^i \text{ là } \frac{1}{2^{-i}} \\ &= 2^h \sum_{k=0}^h \frac{k}{2^k} && \text{Đổi biến: } k = h-i \\ &\leq n \sum_{k=0}^{\infty} \frac{k}{2^k} && \text{Thay tổng hữu hạn bởi tổng vô hạn} \\ &= O(n) && \text{Tổng số trên nhỏ hơn } 2 \end{aligned}$$

Vậy, thời gian tính của Build-Max-Heap là  $T(n) = O(n)$ .

### 5.5.2. Sắp xếp vun đồng

Sử dụng đồng ta có thể phát triển thuật toán sắp xếp mảng. Sơ đồ của thuật toán được trình bày như sau:

- Tạo đồng max-heap từ mảng đã cho;
- Đổi chỗ gốc (phần tử lớn nhất) với phần tử cuối cùng trong mảng;
- Loại bỏ nút cuối cùng bằng cách giảm kích thước của đồng đi 1;
- Thực hiện Max-Heapify đối với gốc mới;
- Lặp lại quá trình cho đến khi đồng chỉ còn một nút.

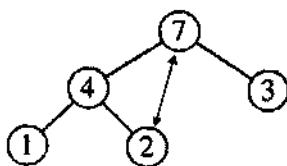
**Algorithm: HeapSort( $A$ )**

1. Build-Max-Heap( $A$ )
2. **for**  $i \leftarrow \text{length}[A]$  **downto** 2
3.     **do**  $\text{exchange } A[1] \leftrightarrow A[i]$
4.     Max-Heapify( $A$ , 1,  $i - 1$ )

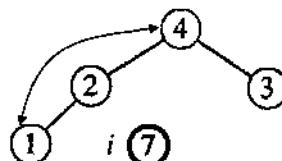
Thời gian tính của dòng 1 là  $O(n)$ . Thời gian tính của dòng 3 và 4 là  $O(\log n)$ .

Vòng lặp 2 lặp  $n - 1$  lần. Vậy, thời gian tính của Heap Sort là  $O(n \log n)$ .

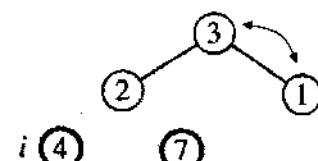
**Ví dụ:** Xét mảng  $A = [7, 4, 3, 1, 2]$ .



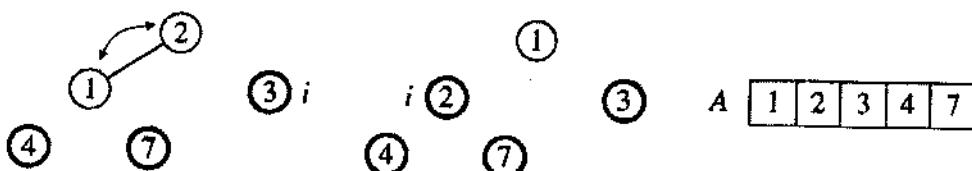
Max-Heapify ( $A, 1, 4$ )



Max-Heapify ( $A, 1, 3$ )



Max-Heapify ( $A, 1, 2$ )



Max-Heapify ( $A, 1, 1$ )

### 5.5.3. Hàng đợi có ưu tiên (Priority queue)

Cho tập  $S$  thường xuyên biến động, mỗi phần tử  $x$  được gán với một giá trị gọi là **khóa** (hay **độ ưu tiên**). Cần một cấu trúc dữ liệu hỗ trợ hiệu quả các thao tác chính sau:

- **Insert( $S, x$ )** – bổ sung phần tử  $x$  vào  $S$ .
- **Max( $S$ )** – trả lại phần tử lớn nhất.
- **Extract-Max( $S$ )** – loại bỏ và trả lại phần tử lớn nhất.
- **Increase-Key( $S, x, k$ )** – tăng khóa của  $x$  thành  $k$ .

Cấu trúc dữ liệu đáp ứng các yêu cầu đó là **hàng đợi có ưu tiên**. Hàng đợi có ưu tiên có thể tổ chức nhờ sử dụng cấu trúc dữ liệu đồng để cất giữ các khóa.

**Chú ý:** Có thể thay "max" bởi "min".

#### Các phép toán đối với hàng đợi có ưu tiên

Hàng đợi có ưu tiên có các phép toán cơ bản sau:

- **Insert( $S, x$ )**: bổ sung phần tử  $x$  vào tập  $S$ .
- **Extract-Max( $S$ )**: loại bỏ và trả lại phần tử của  $S$  với khóa lớn nhất.
- **Maximum( $S$ )**: trả lại phần tử của  $S$  với khóa lớn nhất.
- **Increase-Key( $S, x, k$ )**: tăng giá trị khóa của phần tử  $x$  lên thành  $k$  (Giả sử  $k \geq$  khóa hiện tại của  $x$ ).

#### Phép toán HEAP-MAXIMUM

**Chức năng:** Trả lại phần tử lớn nhất của đồng.

**Algorithm:** **Heap-Maximum( $A$ )**

**return  $A[1]$**

Thời gian tính:  $O(1)$ .

## Phép toán Heap–Extract–Max

**Chức năng:** Trả lại phần tử lớn nhất và loại bỏ nó khỏi đống.

**Thuật toán:**

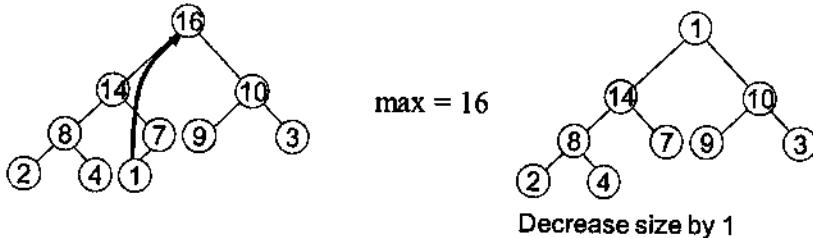
- Hoán đổi gốc với phần tử cuối cùng.
- Giảm kích thước của đống đi 1.
- Gọi Max–Heapify đối với gốc mới trên đống có kích thước  $n - 1$ .

**Algorithm:** **Heap–Extract–Max(A, n)**

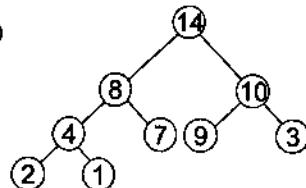
1. **if**  $n < 1$
2. **then** error “heap underflow”
3.  $\text{max} \leftarrow A[1]$
4.  $A[1] \leftarrow A[n]$
5. **Max–Heapify(A, 1, n-1)** // Vun lại đống
6. **return** max

Thời gian tính:  $O(\log n)$ .

**Ví dụ:** Heap–Extract–Max



Run Max-Heapify (A, 1, n - 1)



## Phép toán Heap–Increase–Key

**Chức năng:** Tăng giá trị khóa của phần tử  $i$  trong đống.

**Thuật toán:**

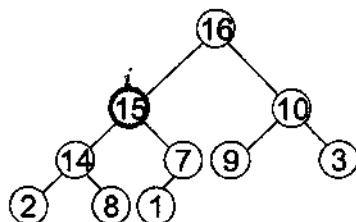
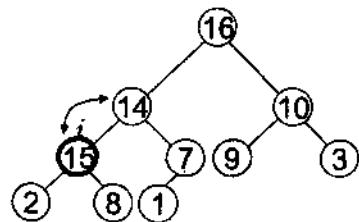
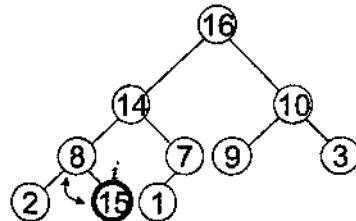
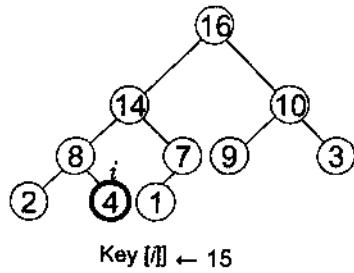
- Tăng khóa của  $A[i]$  thành giá trị mới.
- Nếu tính chất max–heap bị vi phạm: di chuyển theo đường đến gốc để tìm chỗ thích hợp cho khóa mới bị tăng này.

**Algorithm: Heap-Increase-Key(A, i, key)**

1. **if** key < A[i]
2.   **then** error "khóa mới nhỏ hơn khóa hiện tại"
3. A[i]  $\leftarrow$  key
4. **while** i > 1 **and** A[parent(i)] < A[i]
5.   **do** hoán đổi A[i]  $\leftrightarrow$  A[parent(i)]
6.    i  $\leftarrow$  parent(i)

Thời gian tính:  $O(\log n)$ .

**Ví dụ:** Heap-Increase-Key.



**Phép toán Max-Heap-Insert**

**Chức năng:** Chèn một phần tử mới vào max-heap.

**Thuật toán:**

- Mở rộng max-heap với một nút mới có khóa là  $-\infty$ .
- Gọi Heap-Increase-Key để tăng khóa của nút mới này thành giá trị của phần tử mới và vun lại đống.

**Algorithm: Max-Heap-Insert(A, key, n)**

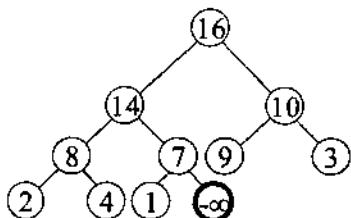
1. heap-size[A]  $\leftarrow$  n + 1
2. A[n + 1]  $\leftarrow -\infty$
3. Heap-Increase-Key(A, n + 1, key)

Running time:  $O(\log n)$ .

### Ví dụ: Max-Heap-Insert.

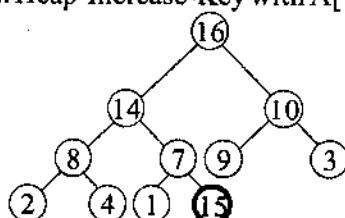
Insert 15:

- Start with  $-\infty$

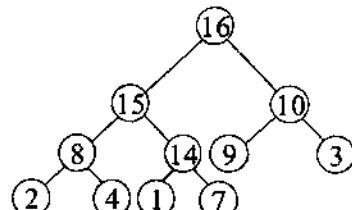
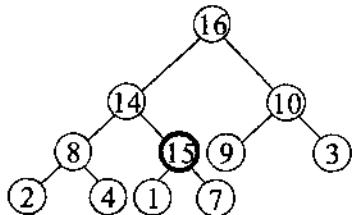


Increase key to 15

Run Heap-Increase-Key with  $A[11] = 15$



Max-Heapify



### Tổng kết

Chúng ta có thể thực hiện các phép toán sau đây với đồng:

| Phép toán           | Thời gian tính |
|---------------------|----------------|
| - Max-Heapify       | $O(\log n)$    |
| - Build-Max-Heap    | $O(n)$         |
| - Heap-Sort         | $O(n \log n)$  |
| - Max-Heap-Insert   | $O(\log n)$    |
| - Heap-Extract-Max  | $O(\log n)$    |
| - Heap-Increase-Key | $O(\log n)$    |
| - Heap-Maximum      | $O(1)$         |

### Cài đặt hàng đợi có ưu tiên bởi danh sách mốc nội

- Priority Queues sử dụng heaps:

- + Tìm phần tử lớn nhất:      Heap-Maximum       $O(1)$
- + Lấy và loại phần tử lớn nhất:      Heap-Extract-Max       $O(\log n)$
- + Bổ sung phần tử mới:      Heap-Insert       $O(\log n)$
- + Tăng giá trị phần tử:      Heap-Increase-Key       $O(\log n)$

- Priority Queues sử dụng danh sách mốc nội được sắp thứ tự:

- Tìm phần tử lớn nhất:       $O(1)$

- Lấy và loại phần tử lớn nhất:  $O(1)$
- Bỏ sang phần tử mới:  $O(n)$
- Tăng giá trị phần tử:  $O(n)$

### Có thể sắp xếp nhanh đến mức độ nào?

Trong chương 3 ta đã chứng minh được rằng: mọi thuật toán chỉ sử dụng phép so sánh có thời gian là  $\Omega(n \log n)$ . Do các thuật toán Heap Sort, Merge Sort có thời gian  $O(n \log n)$  nên chúng là các thuật toán nhanh nhất trong số các thuật toán chỉ sử dụng phép so sánh để giải bài toán sắp xếp.

## 5.6. CẬN DƯỚI CHO ĐỘ PHÚC TẠP TÍNH TOÁN CỦA BÀI TOÁN SẮP XẾP

### \* Mô hình sắp xếp

Giả sử rằng phép toán cơ bản mà ta được sử dụng là *so sánh hai số* ta có thể giảm bớt không gian tìm kiếm đi một nửa sau mỗi lần so sánh hai số. Giả sử, có  $N$  phần tử cần sắp xếp và không có hai phần tử nào trùng nhau. Có bao nhiêu thứ tự có thể?

**Ví dụ:** a, b, c ( $N = 3$ ):

Có sáu thứ tự có thể chính là tất cả các hoán vị có thể của ba phần tử:

(a b c), (a c b), (b a c), (b c a), (c a b), (c b a).

Đối với  $N$  phần tử:

–  $N$  khả năng chọn vị trí thứ nhất,  $(N - 1)$  khả năng chọn vị trí thứ hai,..., hai khả năng chọn vị trí sát cuối, một khả năng chọn vị trí cuối cùng.

– Vậy có tất cả  $N(N - 1)(N - 2)\cdots(2)(1) = N!$  thứ tự có thể.

**Cây quyết định:** Cây quyết định là cây nhị phân thỏa mãn:

– Mỗi nút  $\equiv$  một phép so sánh "a < b".

+ Cũng có thể coi tương ứng với một không gian con các trình tự.

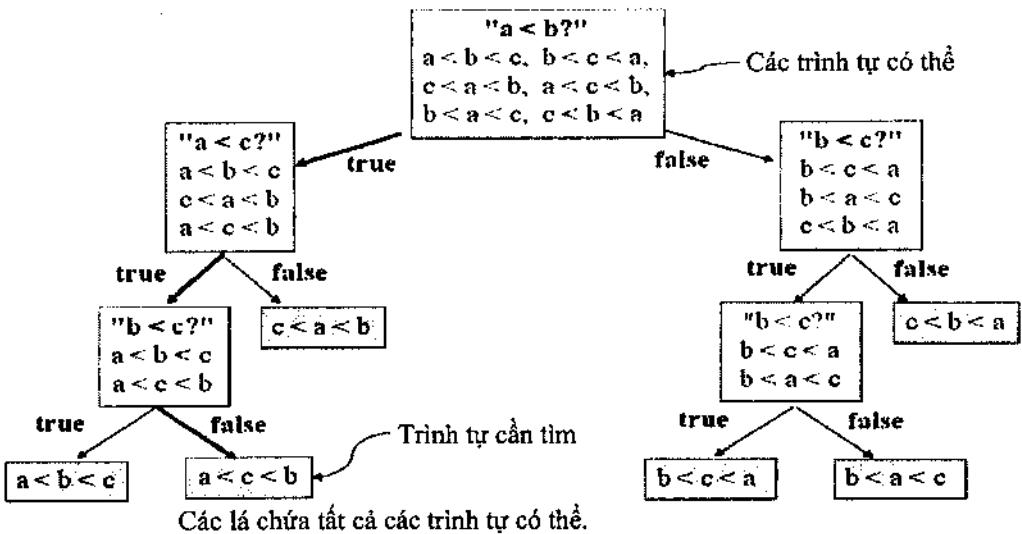
– Mỗi cạnh  $\equiv$  rẽ nhánh theo câu trả lời (true hoặc false).

– Mỗi lá  $\equiv$  một trình tự sắp xếp.

+ Cây quyết định có bao nhiêu lá, nếu có  $N$  phần tử phân biệt cần sắp xếp? Câu trả lời là  $N!$ , tức là tất cả các trình tự có thể.

Đối với mỗi bộ dữ liệu, chỉ có một lá chứa trình tự sắp xếp cần tìm.

**Ví dụ:** Cây quyết định với  $N = 3$ .



### Cây quyết định và sắp xếp

- Mỗi thuật toán sắp xếp đều có thể mô tả bởi cây quyết định.
- + Tìm lá nhờ di chuyển theo cây (tức là thực hiện phép so sánh).
- + Mỗi quyết định sẽ giảm không gian tìm kiếm đi một nửa.
- Thời gian tính trong tình huống tồi nhất  $\geq$  số phép so sánh nhiều nhất phải thực hiện:
  - + Số phép so sánh nhiều nhất cần thực hiện chính là độ dài đường đi dài nhất trên cây quyết định, tức là độ cao của cây.

**Mệnh đề:**  $\log(N!) = \Omega(N \log N)$ .

**Chứng minh:** Ta có:

$$\begin{aligned}
 \log(N!) &= \log(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)) \\
 &= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\
 &\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log \frac{N}{2} \\
 &\geq \frac{N}{2} \log \frac{N}{2} \\
 &= \Omega(N \log N).
 \end{aligned}$$

### Cận dưới cho độ phức tạp của bài toán sắp xếp

- Thời gian tính của mọi thuật toán chỉ dựa trên phép so sánh là  $\Omega(N \log N)$ .
- Do để giải bài toán sắp xếp, ta có thể sử dụng thuật toán sắp xếp vun đồng với thời gian  $O(N \log N)$ , nên cận trên cho bài toán sắp xếp là  $O(N \log N)$ .

– Từ đó suy ra: Độ phức tạp tính toán của bài toán sắp xếp chỉ dựa trên phép so sánh là  $\Theta(N \log N)$ .

Ta có thể phát triển thuật toán tốt hơn hay không nếu không hạn chế là chỉ được sử dụng phép so sánh?

## 5.7. CÁC PHƯƠNG PHÁP SẮP XẾP ĐẶC BIỆT

### 5.7.1. Mở đầu

Ta có thể làm tốt hơn sắp xếp chỉ sử dụng phép so sánh? Với những thông tin bổ sung (hay giả thiết) về đầu vào, chúng ta có thể làm tốt hơn sắp xếp chỉ dựa vào phép so sánh.

– Thông tin bổ sung/Giả thiết:

+ Các số nguyên nằm trong khoảng  $[0..k]$  trong đó  $k = O(n)$ .

+ Các số thực phân bố đều trong khoảng  $[0,1]$ .

– Ta trình bày ba thuật toán có thời gian tuyến tính:

+ Sắp xếp đếm (Counting Sort).

+ Sắp xếp theo cơ số (Radix Sort).

+ Sắp xếp đóng gói (Bucket Sort).

Để đơn giản cho việc trình bày, ta xét việc sắp xếp các số nguyên không âm.

### 5.7.2. Sắp xếp đếm (Counting Sort)

**Input:**  $n$  số nguyên trong khoảng  $[0..k]$  trong đó  $k$  là số nguyên và  $k = O(n)$ .

**Ý tưởng:** Với mỗi phần tử  $x$  của dãy đầu vào ta xác định hạng (*rank*) của nó như là số lượng phần tử nhỏ hơn  $x$ .

Một khi ta đã biết hạng  $r$  của  $x$ , ta có thể xếp nó vào vị trí  $r + 1$ .

**Ví dụ:** Nếu có 6 phần tử nhỏ hơn 17, ta có thể xếp 17 vào vị trí thứ 7.

**Lặp:** Khi có một loạt phần tử có cùng giá trị, ta sắp xếp chúng theo thứ tự xuất hiện trong dãy ban đầu (để có được tính ổn định của sắp xếp).

Cài đặt trên C

```
void countSort(int a[], int b[], int c[], int k) {
 // k - giá trị phần tử lớn nhất
 // Đếm: b[i] - số phần tử có giá trị i
 for (int i=0; i <= k; i++) b[i] = 0;
 for (int i=0; i<n; i++) b[a[i]]++;
 // Tính hạng: b[i] hạng của phần tử có giá trị i
```

```

for (i = 1; i <= k; i++)
 b[i] += b[i - 1];
// Sắp xếp
for (i = n-1; i >= 0; i--) {
 c[b[a[i]] - 1] = a[i];
 b[a[i]] = b[a[i]] - 1;
}

```

### Chương trình demo

```

/* include <iostream.h, stdlib.h,
stdio.h, conio.h, process.h, time.h */
int a[10000], c[10000]; int n;
void print(int a[], int n) {
 for (int i = 0; i < n; i++)
 printf("%8i", a[i]);
 printf("\n");
}
void countSort(int a[], int b[], int c[], int amax);
void main() {
 int i, k; // giá trị lớn nhất có thể của mảng A
 clrscr(); randomize();
 printf("\n n = "); scanf("%i", &n);
 printf("\n max = "); scanf("%i", &k);
 for (i = 0; i < n; i++) a[i] = rand() % k;
 printf("\n Day ban dau:\n"); print(a, n);
 int * b; int amax = 0;
 // Tính phần tử lớn nhất amax
 for (i = 0; i < n; i++)
 if (a[i] > amax) amax = a[i];
 countSort(a, b, c, amax);
 print(b, n);
}

```

```

b = new int[amax];
countSort(a, b, c, amax);
printf("\n Day ket qua:\n"); print(c,n);
// Verify result array
c[n]=32767;
for (i = 0; i < n; i++)
 if (c[i]>c[i+1]) {
 printf(" ?????? Loi o vi tri: %di ", i);
 getch(); }
printf("\n Correct!"); getch();
}

```

#### **Phân tích độ phức tạp của sắp xếp đếm**

- Vòng lặp for đếm  $b[i]$  – số phần tử có giá trị  $i$ , đòi hỏi thời gian  $\Theta(n + k)$ .
- Vòng lặp for tính hạng đòi hỏi thời gian  $\Theta(n)$ .
- Vòng lặp for thực hiện sắp xếp đòi hỏi thời gian  $\Theta(k)$ .

Tổng cộng, thời gian tính của thuật toán là  $\Theta(n + k)$ .

Do  $k = O(n)$ , nên thời gian tính của thuật toán là  $\Theta(n)$ , nghĩa là trong trường hợp này, nó là một trong những thuật toán tốt nhất. Điều đó không xảy ra nếu giả thiết  $k = O(n)$  không được thực hiện. Thuật toán sẽ làm việc rất tồi khi  $k \gg n$ .

Sắp xếp đếm không có tính tại chỗ.

#### **5.7.3. Sắp xếp theo cơ số (Radix Sort)**

**Giả thiết:** Đầu vào gồm  $n$  số nguyên, mỗi số có  $d$  chữ số.

**Ý tưởng của thuật toán:** Do thứ tự sắp xếp các số cần tìm cũng chính là thứ tự từ điển của các xâu tương ứng với chúng, nên để sắp xếp ta sẽ tiến hành  $d$  bước sau:

**Bước 1.** Sắp xếp các số theo chữ số thứ 1.

**Bước 2.** Sắp xếp các số trong dãy thu được theo chữ số thứ 2 (sử dụng sắp xếp ổn định).

...

**Bước d.** Sắp xếp các số trong dãy thu được theo chữ số thứ  $d$  (sử dụng sắp xếp ổn định).

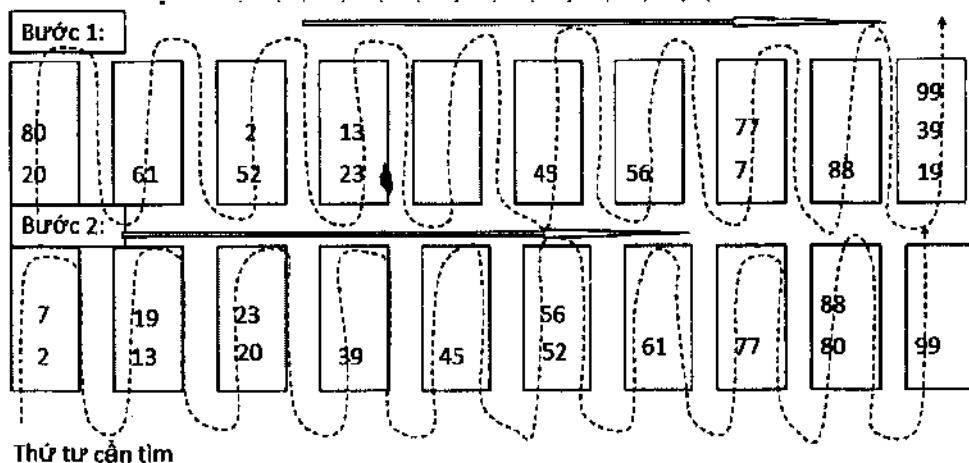
Giả sử các số được xét trong hệ đếm cơ số  $k$ . Khi đó mỗi chữ số chỉ có  $k$  giá trị, nên ở mỗi bước ta có thể sử dụng sắp xếp đếm với thời gian tính  $O(n + k)$ .

**Ví dụ:** Xét dãy số:

23, 45, 7, 56, 20, 19, 88, 77, 61, 13, 52, 39, 80, 2, 99

Việc sắp xếp bao gồm nhiều bước, bắt đầu từ chữ số trái nhất, tiếp đến là chữ số hàng chục, hàng trăm,...

Hình sau minh họa cho thao tác của thuật toán:



### Sơ đồ thuật toán Radix Sort

**Radix-Sort(A, d)**

1. **for**  $i \leftarrow 1$  to  $d$  **do**

2. Sử dụng sắp xếp ổn định để sắp xếp A theo chữ số thứ  $i$ .

### Phân tích độ phức tạp

– Thời gian tính: Nếu bước 2 sử dụng *sắp xếp đếm* thì thời gian tính của một lần lặp là  $\Theta(n + k)$ , do đó thời gian tính của thuật toán Radix Sort là  $T(n) = \Theta(d(n + k))$ . Thời gian này sẽ là  $\Theta(n)$  nếu  $d$  là hằng số và  $k = O(n)$ .

**Chú ý:** Để thực hiện sắp xếp ở mỗi lần lặp của thuật toán, phải sử dụng thuật toán sắp xếp ổn định, nếu không sẽ không có kết quả đúng.

### Thuật toán sắp xếp theo cơ số nhị phân

Ta xét các số *nguyên không âm* 32-bit và ta sẽ chuyển chúng về các số có bốn chữ số trong hệ đếm cơ số 256.

Giả sử:

$$p = a_{31} \times 2^{31} + a_{30} \times 2^{30} + \dots + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0$$

trong đó:  $a_i$  bằng 0 hoặc 1.

Khi đó trong hệ đếm cơ số 256 ta có:

$$p = b_3 \times 256^3 + b_2 \times 256^2 + b_1 \times 256^1 + b_0 \times 256^0$$

trong đó:

$$\begin{aligned}b_3 &= a_{31}2^7 + a_{30}2^6 + a_{29}2^5 + a_{28}2^4 + a_{27}2^3 + a_{26}2^2 + a_{25}2^1 + a_{24}2^0, \\b_2 &= a_{23}2^7 + a_{22}2^6 + a_{21}2^5 + a_{20}2^4 + a_{19}2^3 + a_{18}2^2 + a_{17}2^1 + a_{16}2^0, \\b_1 &= a_{15}2^7 + a_{14}2^6 + a_{13}2^5 + a_{12}2^4 + a_{11}2^3 + a_{10}2^2 + a_92^1 + a_82^0, \\b_0 &= a_72^7 + a_62^6 + a_52^5 + a_42^4 + a_32^3 + a_22^2 + a_12^1 + a_02^0.\end{aligned}$$

### Tính các chữ số

Nếu số nguyên  $n$  nằm trong phạm vi  $[0, 2^{31}]$ , ta có thể tính các chữ số  $b_0, b_1, b_2, b_3$  của nó nhờ sử dụng các phép toán & (bitwise) và  $>>$  (dịch phải – right shift) được cung cấp sẵn trong C (các phép toán này thực hiện rất hiệu quả) như sau:

$$\begin{aligned}b_0 &= n \& 255 \\b_1 &= (n >> 8) \& 255 \\b_2 &= (n >> 16) \& 255 \\b_3 &= (n >> 24) \& 255\end{aligned}$$

### Cài đặt trên C

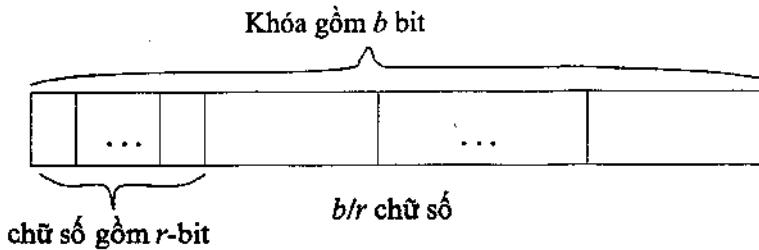
```
void radixsort(long a[], int n){
 int i, j, shift;
 long temp[1000];
 int bin_size[256], first_ind[256];
 for (shift=0; shift<32; shift+=8) {
 /* Tính kích thước mỗi cụm và sao chép các
 phần tử của mảng a vào mảng temp */
 for (i=0; i<256; i++) bin_size[i]=0;
 for (j=0; j<n; j++) {
 i=(a[j]>>shift)&255;
 bin_size[i]++;
 temp[j]=a[j];
 }
 /* Tính chỉ số vị trí bắt đầu của mỗi cụm */
 first_ind[0]=0;
 for (i=1; i<256; i++)
 first_ind[i]=first_ind[i-1]+bin_size[i-1];
 /* Sao chép phần tử của mảng temp vào cụm của nó
 trong mảng a */
 }
}
```

```

 for (j=0; j<n; j++) {
 i=(temp[j]>>shift)&255;
 a[first_ind[i]]=temp[j];
 first_ind[i]++;
 }
 } // end for shift
} // end radixsort
Chuong trinh chinh
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
void print(long a[], int n) {
 for (int i = 0; i < n; i++)
 printf("%8i", a[i]);
 printf("\n");
}
int main () {
 long data[1000];
 int i, N;
 printf("\n Nhap so luong phan tu n = ");
 scanf("%i",&N);
 // Randomdata
 for (i=0; i<N; i++) data[i]=rand();
 printf("\n Day dau vao:\n"); print(data, N);
 radixsort (data, N);
 printf("\n Day duoc sap thu tu:\n"); print(data, N);
 data[N]=9999999;
 for (i=0; i<N; i++)
 if (data[i]>data[i+1]) printf("\n ?????? ERROR:\n");
 getch();
}

```

## Chọn cơ số như thế nào?



Cho  $n$  số, mỗi số có  $b$  bit và số  $r \leq b$ . Khi đó Radix-Sort đòi hỏi thời gian  $\Theta((b/r)(n + 2^r))$ :

- Chọn  $d = b/r$  chữ số, mỗi chữ số gồm  $r$  bit có giá trị trong khoảng  $[0..2^r - 1]$ .
- Vì thế ở mỗi bước có thể sử dụng Counting Sort với  $k = 2^r - 1$ , đòi hỏi thời gian  $\Theta(n + k)$ .
  - Tất cả phải thực hiện  $d$  bước, do đó thời gian tổng cộng là  $\Theta(d(n + 2^r))$ , hay  $\Theta((b/r)(n + 2^r))$ .

Do đó:

- Với các giá trị của  $n$  và  $b$  cho trước, ta có thể chọn  $r \leq b$  để đạt mục tiêu của  $\Theta((b/r)(n + 2^r))$ .
- Nếu  $b \leq \log n$ , chọn  $r = \log n$ , ta thu được thuật toán với thời gian  $\Theta(n)$ .
- Nếu  $b > \log n$ , chọn  $r = \log n$ , ta thu được thuật toán với thời gian  $\Theta(bn/\log n)$ .

### 5.7.4. Sắp xếp phân cụm (Bucket Sort)

**Giả thiết:** Đầu vào gồm  $n$  số thực có phân bố đều trong khoảng  $[0..1]$  (các số có xác suất xuất hiện như nhau).

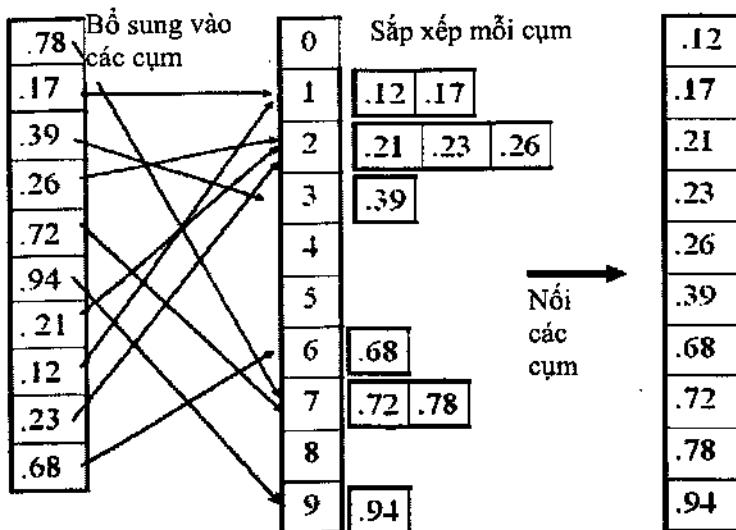
**Ý tưởng của thuật toán**

Chia đoạn  $[0..1]$  ra làm  $n$  cụm (buckets):

$$0, 1/n, 2/n, \dots, (n-1)/n.$$

- Đưa mỗi phần tử  $a_j$  vào đúng cụm của nó  $i/n \leq a_j < (i+1)/n$ .
- Do các số là phân bố đều nên không có quá nhiều số rơi vào cùng một cụm.
- Nếu ta chèn chúng vào các cụm (sử dụng sắp xếp chèn) thì các cụm và các phần tử trong chúng đều được sắp xếp theo đúng thứ tự.

**Ví dụ:** Sắp xếp phân cụm,  $n = 10$ , các cụm là  $0, 0.1, 0.2, \dots, 0.9$ .



## Sơ đồ thuật toán Bucket Sort

### Bucket Sort(A)

{  $A[0..n-1]$  là mảng đầu vào }

$B[0], B[1], \dots, B[n - 1]$  là các danh sách cụm }

1.  $n = \text{length}(A)$
2. **for**  $i = 0$  **to**  $n - 1$
3.   **do** Bổ sung  $A[i]$  vào danh sách  $B[\lfloor n * A[i] \rfloor]$
4. **for**  $i = 0$  **to**  $n - 1$

5.   **do** Insertion Sort( $B[i]$ )

6. Nối các danh sách  $B[0], B[1], \dots, B[n - 1]$  theo thứ tự:

$A[0..n - 1]$  là mảng đầu vào

$B[0], B[1], \dots, B[n - 1]$  là các danh sách cụm

### Phân tích thời gian tính của Bucket Sort

– Tất cả các dòng, ngoại trừ dòng 5 (Insertion Sort), đòi hỏi thời gian  $O(n)$  trong tình huống tồi nhất.

– Trong tình huống tồi nhất,  $O(n)$  số được đưa vào cùng một cụm, do đó thuật toán có thời gian tính  $O(n^2)$  trong tình huống tồi nhất.

– Tuy nhiên, trong tình huống trung bình, chỉ có một số lượng hằng số phần tử của dãy cần sắp xếp rơi vào trong mỗi cụm, vì thế thuật toán có thời gian tính trung bình là  $O(n)$  (ta công nhận sự kiện này).

**Mở rộng:** Sử dụng các sơ đồ chi số hóa khác nhau để phân cụm các phần tử.

## Tổng kết: Các thuật toán sắp xếp dựa trên phép so sánh

| Tên phương pháp | Trung bình    | Tối đa        | Tại chỗ | Ôn định |
|-----------------|---------------|---------------|---------|---------|
| Bubble Sort     | —             | $O(n^2)$      | Yes     | Yes     |
| Selection Sort  | $O(n^2)$      | $O(n^2)$      | Yes     | No      |
| Insertion Sort  | $O(n + d)$    | $O(n^2)$      | Yes     | Yes     |
| Merge Sort      | $O(n \log n)$ | $O(n \log n)$ | No      | Yes     |
| Heap Sort       | $O(n \log n)$ | $O(n \log n)$ | Yes     | No      |
| Quick Sort      | $O(n \log n)$ | $O(n^2)$      | No      | No      |

## BÀI TẬP CHƯƠNG 5

- Trình diễn trên sơ đồ hoạt động của thuật toán sắp xếp nhanh được áp dụng để sắp xếp theo thứ tự tăng dần dãy số sau đây: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Giả thiết phần tử chốt được chọn là phần tử đứng cuối.
- Trình diễn trên sơ đồ hoạt động của thuật toán sắp xếp nhanh được áp dụng để sắp xếp theo thứ tự giảm dần dãy số sau đây: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1. Giả thiết phần tử chốt được chọn là phần tử đứng đầu.
- Trình diễn thuật toán sắp xếp trộn (Merge Sort) thực hiện sắp xếp dãy số:  

$$<22, 15, 36, 44, 10, 3, 9, 13, 29, 25>.$$

- Cho dãy số:

$$S = (23, -20, 24, -13, 28, 26, 25, 21).$$

Trình diễn thuật toán sắp xếp nhanh (Quick Sort) thực hiện sắp xếp dãy  $S$ , trong đó phần tử chốt được chọn là phần tử đứng cuối.

- Cho mảng  $A = (16; 14; 10; 8; 7; 9; 3; 2; 1)$  biểu diễn một max-heap.
  - Hãy vẽ cây nhị phân tương ứng với max-heap đã cho.
  - Hãy trình bày các thao tác cần thực hiện trên cây để bổ sung thêm key = 15 vào max-heap nói trên để thu được một max-heap mới.

6.

- a) Vẽ cây nhị phân tương ứng với đồng min (min-heap) được tạo từ mảng chứa dãy số sau đây:

**40, 30, 65, 25, 35, 50, 76, 10, 28, 27, 33, 36, 34, 48, 60, 68, 80, 69.**

- b) Trình diễn thao tác giảm giá trị của nút với giá trị 80 xuống 13 trên min-heap thu được ở câu a).

7.

- a) Vẽ cây nhị phân tương ứng với đồng min (min-heap) được tạo từ mảng chứa dãy số sau đây:

**40, 30, 65, 25, 36, 50, 76, 34, 35**

- b) Trình diễn thao tác loại bỏ nút với giá trị 35 vào min-heap thu được ở câu a).

8. Cho đồng max (max-heap) gồm 10 phần tử có biểu diễn dưới dạng mảng là:

[50, 40, 35, 31, 32, 24, 18, 30, 20, 8]

- a) Vẽ cây nhị phân tương ứng với đồng đã cho.

- b) Trình diễn các thao tác cần thực hiện để bổ sung một nút với khóa 45 vào đồng đã cho.

9. Cho mảng A = (10; 12; 14; 13; 18; 19; 16; 15; 17; 19) biểu diễn một đồng min (min-heap).

- a) Hãy vẽ cây nhị phân tương ứng với min-heap đã cho.

- b) Hãy trình bày các thao tác cần thực hiện trên cây để bổ sung thêm key = 9 vào min-heap nói trên để thu được một min-heap mới. Hãy vẽ cây thu được sau từng thao tác.

10. Cho dãy số:

**46, 34, 83, 75, 25, 57, 93, 27, 17, 39, 52, 31, 93, 44**

- a) Vẽ min-heap thu được bằng cách chèn lần lượt các phần tử của dãy số vào heap rỗng.

- b) Vẽ max-heap thu được bằng cách chèn lần lượt các phần tử của dãy số vào heap rỗng.

11. Ta có thể xây dựng đống max (max-heap) đối với dãy số A bằng cách bổ sung dần các phần tử của A vào max-heap. Xét thuật toán xây dựng theo ý tưởng vừa nêu:

|                                             |                                                              |
|---------------------------------------------|--------------------------------------------------------------|
| <b>BUILD-MAX-HEAP_Ins (A)</b>               | <b>MAX-HEAP-INSERT (A, key)</b>                              |
| 1 heapsize[A] = 1                           | 1 heapsize[A] = heapsize[A] +                                |
| 2 for i = 2 to length[A] do                 | 2 i = heapsize[A]                                            |
| 3 MAX-HEAP-INSERT(A, A[i])                  | 3 A[i] = key                                                 |
| Build-Max-Heap trong giáo trình:            | 4 while (i > 1 and A[ $\lfloor i/2 \rfloor$ ] < A[i]) do     |
| <b>Build-Max-Heap (A)</b>                   | 5 exchange A[i] $\leftrightarrow$ A[ $\lfloor i/2 \rfloor$ ] |
| 1 n = length[A]                             | 6 i = $\lfloor i/2 \rfloor$                                  |
| 2 for i = $\lfloor n/2 \rfloor$ downto 1 do |                                                              |
| 3 Max-Heapify(A, i, n)                      |                                                              |

- a) Hỏi **BUILD-MAX-HEAP\_Ins** và **Build\_Max\_Heap** trong giáo trình có luôn tạo ra cùng một đống như nhau khi làm việc với cùng một đầu vào A hay không? Nếu câu trả lời là đúng hãy chứng minh và đưa ra phản ví dụ nếu trái lại.
- b) Đưa ra đánh giá thời gian tính trong tình huống tồi nhất của **BUILD-MAX-HEAP\_Ins** khi xây dựng đống gồm  $n$  phần tử.
12. Xét bài toán sau đây: "Cho hai tập  $X$  và  $Y$ , mỗi tập gồm  $n$  số nguyên và số nguyên  $k$ . Hỏi có thể tìm được một số  $x$  thuộc tập  $X$  và một số  $y$  thuộc tập  $Y$  sao cho  $x + y = k$  hay không?"
- a) Phát triển thuật toán hiệu quả để giải bài toán đặt ra (mô tả thuật toán trong giả ngôn ngữ).
- b) Chứng minh tính đúng đắn và đánh giá thời gian tính của thuật toán.
13. Xét bài toán sau đây: "Cho dãy  $n$  số nguyên không âm  $a_1, a_2, \dots, a_n$  và số nguyên  $k$ . Hỏi dãy đã cho có chứa hai số với tổng bằng  $k$  hay không? Nếu câu trả lời là có, hãy đưa ra hai số đó. Giả thiết  $n \leq 10^6$ ;  $k \leq 10^9$ ."
- a) Phát triển thuật toán hiệu quả để giải bài toán đặt ra (mô tả thuật toán trong giả ngôn ngữ).
- b) Chứng minh tính đúng đắn và đánh giá thời gian tính của thuật toán.
14. Cho dãy số:

46, 34, 83, 75, 25, 57, 93, 27, 17, 39, 52, 31, 93, 44

- a) Vẽ *min-heap* thu được bằng cách chèn lần lượt các phần tử của dãy số vào *heap* rỗng.

- b) Về *max-heap* thu được bằng cách chèn lần lượt các phần tử của dãy số vào *heap* rỗng.
15. Ta nói mảng gồm  $n$  số nguyên  $a[1..n]$  là chứa phần tử trội nếu tìm được phần tử xuất hiện trong mảng nhiều hơn  $n/2$  lần. Chẳng hạn, khi  $n = 7$ , mảng  $a = (1, 2, 1, 2, 1, 3, 1)$  chứa phần tử trội là 1, mảng  $a' = (1, 2, 3, 2, 3, 2, 1)$  không chứa phần tử trội. Hãy phát triển thuật toán kiểm tra xem mảng đã cho có chứa phần tử trội hay không. Chứng minh tính đúng đắn và đánh giá thời gian tính của thuật toán đề nghị.
16. Cho mảng gồm  $n$  số nguyên  $a[1..n]$ . Hãy phát triển thuật toán loại bỏ khỏi dãy thu được tất cả các số lặp lại để thu được mảng gồm các phần tử phân biệt. Chứng minh tính đúng đắn và đánh giá thời gian tính của thuật toán đề nghị.
17. Cho mảng gồm  $n$  số nguyên  $a[1..n]$ . Hãy phát triển thuật toán tìm hai số gần nhau nhất trong mảng đã cho. Chứng minh tính đúng đắn và đánh giá thời gian tính của thuật toán đề nghị.

# Chương 6

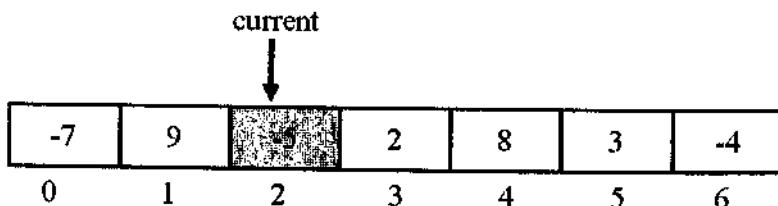
## TÌM KIẾM

### 6.1. TÌM KIẾM TUẦN TỤ VÀ TÌM KIẾM NHỊ PHÂN

Bài toán đặt ra là: Cho danh sách  $list[0..n-1]$  và phần tử  $target$ , ta cần tìm vị trí  $i$  sao cho  $list[i] = target$  hoặc trả lại giá trị  $-1$  nếu không có phần tử như vậy trong danh sách đã cho. Ta xét hai thuật toán cơ bản để giải quyết bài toán tìm kiếm đặt ra.

#### 6.1.1. Tìm kiếm tuần tự (Linear Search or Sequential Search)

Thuật toán tìm kiếm tuần tự được thực hiện từ ý tưởng trực tiếp sau đây: Bắt đầu từ phần tử đầu tiên, duyệt qua từng phần tử cho đến khi tìm được đích hoặc kết luận không tìm được.



Độ phức tạp:  $O(n)$ .

Lưu ý rằng, thuật toán tìm kiếm tuần tự:

- Không đòi hỏi các số được sắp thứ tự;
- Làm việc được với cả danh sách móc nối (Linked Lists).

Tìm kiếm tuần tự có thể cài đặt trên C như sau:

```
int linearSearch(dataArray list, int size, dataElem target)
{
 int i;
 for (i = 0; i < size; i++)
 {
 if (list[i] == target)
```

```

 {
 return i;
 }
}
return -1;
}

```

### Phân tích thời gian tính của thuật toán tìm kiếm tuần tự

Ta cần đánh giá thời gian tính tốt nhất, tồi nhất, trung bình của thuật toán với độ dài đầu vào là  $n$ . Rõ ràng thời gian tính của thuật toán có thể đánh giá bởi số lần thực hiện phép so sánh:

$$(*) \quad (a[i] == \text{target})$$

trong vòng lặp for.

+ Nếu  $a[1] = \text{target}$  thì phép so sánh (\*) phải thực hiện 1 lần. Do đó thời gian tính tốt nhất của thuật toán là  $\Theta(1)$ .

+ Nếu  $\text{target}$  không có mặt trong dãy đã cho, thì phép so sánh (\*) phải thực hiện  $n$  lần. Vì thế thời gian tính tồi nhất của thuật toán là  $\Theta(n)$ .

+ Cuối cùng, ta tính thời gian tính trung bình của thuật toán. Nếu  $\text{target}$  tìm thấy ở vị trí thứ  $i$  của dãy ( $\text{target} = a[i]$ ) thì phép so sánh (\*) phải thực hiện  $i$  lần ( $i = 1, 2, \dots, n$ ), còn nếu  $\text{target}$  không có mặt trong dãy đã cho thì phép so sánh (\*) phải thực hiện  $n$  lần. Từ đó suy ra số lần trung bình phải thực hiện phép so sánh (\*) là:

$$\begin{aligned} [(1 + 2 + \dots + n) + n] / (n + 1) &= [n + n(n + 1)/2] / (n + 1) \\ &= (n^2 + 3n) / [2(n + 1)]. \end{aligned}$$

Ta có:

$$n/4 \leq (n^2 + 3n) / [2(n + 1)] \leq n.$$

Vì vậy, thời gian tính trung bình của thuật toán là  $\Theta(n)$ .

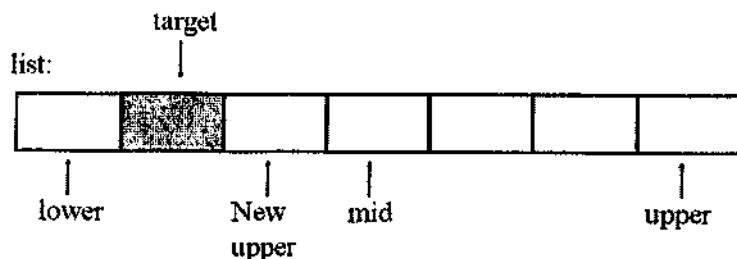
### 6.1.2. Tìm kiếm nhị phân (Binary Search)

Điều kiện để thực hiện thuật toán tìm kiếm nhị phân là:

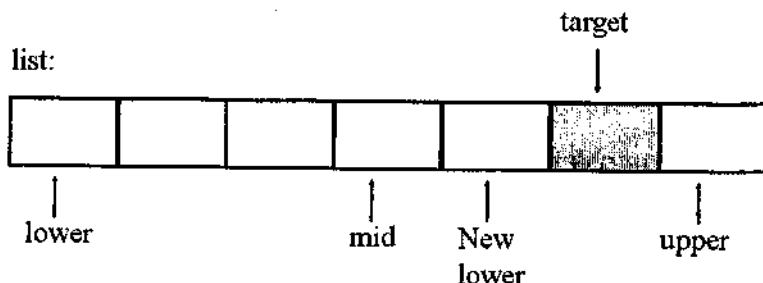
- Danh sách phải được sắp thứ tự không giảm;
- Phải cho phép trực truy.

Lưu ý rằng, điều kiện thứ hai cho thấy tìm kiếm nhị phân không áp dụng được với danh sách móc nối.

Tình huống 1: target < list[mid]



Tình huống 2: list[mid] < target



Cài đặt trên C

```
int binarySearch(dataArray list, int size, dataElem target)
{
 int lower = 0, upper = size - 1, mid;

 while (lower <= upper) {
 mid = (upper + lower)/2;
 if (array[mid] > target)
 { upper = mid - 1; }
 else if (array[mid] < target)
 { lower = mid + 1; }
 else
 { return mid; }
 }
 return -1;
}
```

Để thấy là: đoạn cần khảo sát có độ dài giảm đi một nửa sau mỗi lần lặp.

Như đã chứng minh trong chương 1, độ phức tạp của tìm kiếm nhị phân là  $O(\log n)$ .

### Ví dụ minh họa ứng dụng tìm kiếm nhị phân

**Dãy cấp ba.** Bộ gồm ba số nguyên ( $a_1, a_2, a_3$ ) được gọi là một cấp số cộng tăng nếu như:

$$a_2 - a_1 = a_3 - a_2 \text{ và } a_2 - a_1 > 0.$$

Bộ ba ( $a_1, a_2, a_3$ ) được gọi là đi trước bộ ba ( $b_1, b_2, b_3$ ) trong thứ tự từ điển nếu như một trong ba điều kiện sau đây được thực hiện:

- 1)  $a_1 < b_1$ ;
- 2)  $a_1 = b_1$  và  $a_2 < b_2$ ;
- 3)  $a_1 = b_1, a_2 = b_2$  và  $a_3 < b_3$ .

Dãy số nguyên  $c_1, c_2, \dots, c_n$  ( $|c_i| < 2^{31}$ ,  $i = 1, 2, \dots, n$ ) được gọi là **dãy cấp ba** nếu như có thể tìm được ba số hạng của nó để lập thành một bộ ba cấp số cộng tăng. Ví dụ: Dãy 3, 1, 5, 2, -7, 0, -1 là một dãy cấp ba, vì nó chứa bộ ba cấp số cộng tăng, chẳng hạn (1, 3, 5) hay (-7, -1, 5). Bộ ba (-7, -1, 5) là bộ ba cấp số cộng tăng đầu tiên theo thứ tự từ điển trong số các bộ ba cấp số cộng tăng của dãy đã cho.

**Yêu cầu:** Hãy kiểm tra xem dãy số nguyên cho trước có phải là dãy cấp ba hay không.

#### Thuật toán trực tiếp

- Sắp xếp dãy  $a[1..n]$  theo thứ tự không giảm.
- Duyệt tất cả các bộ ba  $a[i], a[j], a[k]$ ,  $1 \leq i < j < k \leq n$ .

```
for(i = 1; i < n-1; i++) {
 for(int j = i+1; j < n; j++) {
 long key = 2*a[j] - a[i];
 for (k=j+1; k<=n; k++)
 if (a[k]==key) {
 printf("\n YES %li %li %li", a[i], a[j], key);
 halt;
 }
 }
}
```

Thời gian tính:  $O(n^3)$ .

#### Thuật toán nhanh hơn

- Sắp xếp dãy  $a[1..n]$  theo thứ tự không giảm.

– Với mỗi bộ:

$$(a[i], a[j]), i = 1, \dots, n - 2; j = i + 1, \dots, n - 1,$$

tìm kiếm nhị phân giá trị key =  $2 * a[j] - a[i]$  trong khoảng từ  $j + 1$  đến  $n$ . Nếu tìm thấy thì  $(a[i], a[j], a[k])$  là một bộ ba thỏa mãn điều kiện đầu bài. Bộ ba đầu tiên tìm được sẽ là bộ ba cần đưa ra. Nếu không tìm được bộ ba nào thì dãy đã cho không phải là dãy cấp ba.

Thời gian tính:  $O(n^2 \log n)$ .

## 6.2. CÂY NHỊ PHÂN TÌM KIẾM (BINARY SEARCH TREE)

**Đặt vấn đề:** Ta cần xây dựng cấu trúc dữ liệu để biểu diễn các tập động (*dynamic sets*):

- Các phần tử có khóa (*key*) và thông tin đi kèm (*satellite data*).
- Tập động cần hỗ trợ các truy vấn (*queries*) như:
  - + *Search(S, k)*: tìm phần tử có khóa  $k$ ;
  - + *Minimum(S), Maximum(S)*: tìm phần tử có khóa nhỏ nhất, lớn nhất;
  - + *Predecessor(S, x), Successor(S, x)*: tìm phần tử kế cận trước, kế cận sau; đồng thời cũng hỗ trợ các thao tác biến đổi (*modifying operations*) như:
    - + *Insert(S, x)*: bổ sung (Chèn);
    - + *Delete(S, x)*: loại bỏ (Xóa).

Cây nhị phân tìm kiếm là cấu trúc dữ liệu quan trọng để biểu diễn tập động, trong đó tất cả các thao tác đều được thực hiện với thời gian  $O(h)$ , với  $h$  là chiều cao của cây.

### 6.2.1. Định nghĩa

Cây nhị phân tìm kiếm (viết tắt là BST) là cây nhị phân có các tính chất sau:

– Mỗi nút  $x$  (ngoài thông tin đi kèm) có các trường:

- + *left*: con trỏ đến con trái;
- + *right*: con trỏ đến con phải;

+ *parent*: con trỏ đến cha (trường này là tùy chọn);

+ **key**: khóa (thường giả thiết là khóa của các nút khác nhau từng đôi, trái lại nếu có khóa trùng nhau thì cần chỉ rõ thứ tự của hai khóa trùng nhau).

|             |            |              |               |
|-------------|------------|--------------|---------------|
| <i>left</i> | <i>key</i> | <i>right</i> | <i>parent</i> |
|-------------|------------|--------------|---------------|

– Tính chất BST (Binary-search-tree property)

Giả sử  $x$  là gốc của một cây con, khi đó:

+ Với mọi nút  $y$  thuộc cây con trái của  $x$ :  $key(y) < key(x)$ .

+ Với mọi nút  $y$  thuộc cây con phải của  $x$ :  $key(y) > key(x)$ .

Mọi nút  $y$  trong cây con trái  
đều có  $\text{key}(y) < \text{key}(x)$

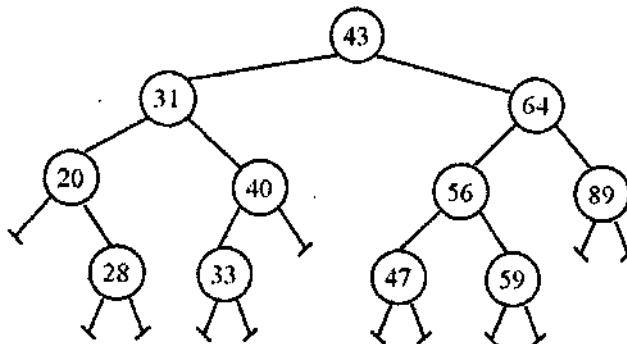
Mọi nút  $y$  trong cây con phải  
đều có  $\text{key}(y) > \text{key}(x)$



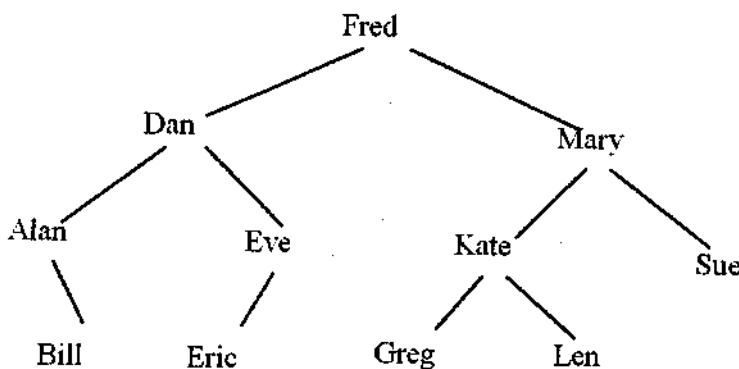
### Các phép toán với cây nhị phân tìm kiếm

- Tìm kiếm (Search): tìm kiếm một phần tử với khóa cho trước;
- Tìm cực tiểu, cực đại (Minimum, Maximum): tìm phần tử với khóa nhỏ nhất (lớn nhất) trên cây;
- Kế cận sau, kế cận trước (Predecessor, Successor): tìm phần tử kế cận sau (kế cận trước) của một phần tử trên cây;
- Chèn (Insert): bổ sung vào cây một phần tử với khóa cho trước;
- Xóa (Delete): loại bỏ khỏi cây một phần tử với khóa cho trước.

**Ví dụ:** Cây BST với khóa là số nguyên:

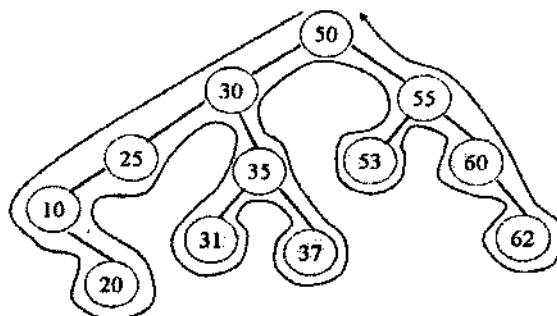


Cây BST với khóa là xâu ký tự:



## Tính chất của BST

- Duyệt BST theo thứ tự giữa sẽ đưa ra dãy khóa được sắp xếp. Ví dụ, duyệt cây:

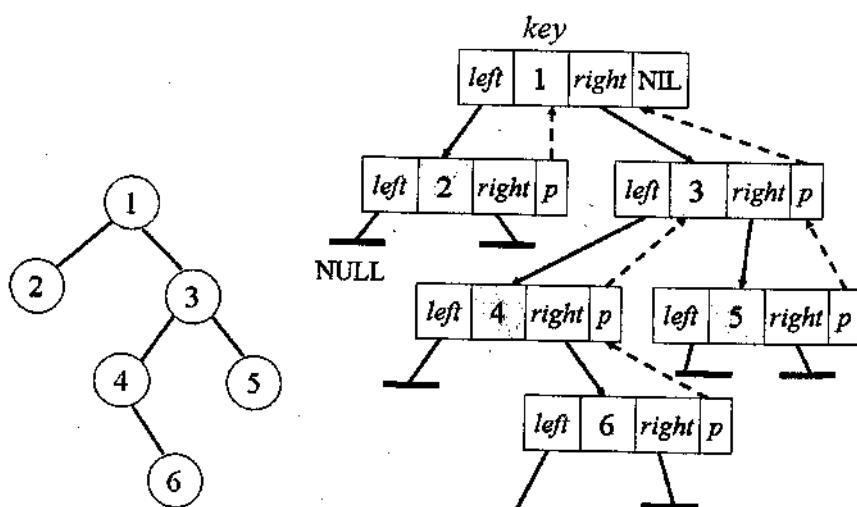


sẽ đưa ra dãy khóa được sắp xếp: 10, 20, 25, 30, 31, 35, 37, 50, 53, 55, 60, 62.

### 6.2.2. Biểu diễn cây nhị phân tìm kiếm

Ta sẽ sử dụng cấu trúc cây nhị phân (Binary Tree Structure) để biểu diễn cây nhị phân tìm kiếm.

Hình dưới đây minh họa cho cách biểu diễn này:



Ví dụ 1:

```
struct TreeNodeRec {
 int key;
 struct TreeNodeRec* leftPtr;
 struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

Ví dụ 2:

```
#define MAXLEN 15

struct TreeNodeRec {
 char key[MAXLEN];
 struct TreeNodeRec* leftPtr;
 struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

Ví dụ 3: Quản lý sách trong thư viện. Giả sử ta có mô tả bản ghi thông tin về sách:

```
struct BookRec {
 char* author;
 char* title;
 char* publisher;

 /* etc.: other book information. */
};

typedef struct BookRec Book;
```

Cấu trúc Binary Search Tree Node để lưu trữ thông tin về sách là:

```
struct TreeNodeRec
{
 Book info;

 struct TreeNodeRec* leftPtr;
 struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

Trong phần tiếp theo ta sẽ sử dụng cấu trúc dưới đây để mô tả các thao tác với BST:

```
struct TreeNodeRec {
 float key;
```

```

 struct TreeNodeRec* leftPtr;
 struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;

```

### 6.2.3. Các phép toán

Các phép toán cơ bản:

- **makeTreeNode(value)**: tạo một nút với khóa cho bởi value.
- **search(nodePtr, k)**: tìm kiếm nút có giá trị khóa bằng k trên BST trả bởi nodePtr;
- **find\_min(nodePtr)**: trả lại nút có khóa nhỏ nhất trên BST trả bởi nodePtr.
- **find\_max(nodePtr)**: trả lại nút có khóa lớn nhất trên BST trả bởi nodePtr.
- **Successor(nodePtr, x)**: trả lại nút kế cận sau của nút x.
- **Predcessor(nodePtr, x)**: trả lại nút kế cận trước của nút x.
- **Insert(nodePtr, float item)**: chèn một nút với khóa cho bởi item vào BST trả bởi nodePtr.
- **delete(nodePtr, item)**: xóa nút có giá trị khóa bằng item trên BST trả bởi nodePtr.

Ngoài ra có thể sử dụng các hàm cơ bản đối với cây nhị phân như:

```

+ void printInorder(nodePtr);
+ void printPreorder(nodePtr);
+ void printPostorder(nodePtr);
+

```

Mô tả trên C các hàm cài đặt các phép toán cơ bản đối với BST:

```

struct TreeNodeRec
{
 float key;
 struct TreeNodeRec* leftPtr;
 struct TreeNodeRec* rightPtr;
};

typedef struct TreeNodeRec TreeNode;
TreeNode* makeTreeNode(float value);
TreeNode* find_min(TreeNode * T);

```

```
TreeNode* find_max(TreeNode* T);
TreeNode* insert(TreeNode* nodePtr, float item);
TreeNode* search(TreeNode* nodePtr, float item);
void printInorder(const TreeNode* nodePtr);
void printPreorder(const TreeNode* nodePtr);
void printPostorder(const TreeNode* nodePtr);
```

Tạo một nút mới (MakeNode)

– Đầu vào: phần tử cần chèn.

Các bước cần thực hiện:

- + Cấp phát bộ nhớ cho nút mới;
- + Kiểm tra lỗi cấp phát;
- + Nếu cấp phát được thì đưa phần tử vào nút mới;
- + Đặt con trái và phải là NULL.

– Đầu ra: con trỏ tới (địa chỉ của) nút mới.

```
TreeNode* makeTreeNode(float value) {
 TreeNode* newNodePtr = NULL;
 newNodePtr = (TreeNode*)malloc(sizeof(TreeNode));
 if (newNodePtr == NULL)
 {
 fprintf(stderr, "Out of memory\n");
 exit(1);
 }
 else
 {
 newNodePtr->key = value;
 newNodePtr->leftPtr = NULL;
 newNodePtr->rightPtr = NULL;
 }
 return newNodePtr;
}
```

Tìm phần tử nhỏ nhất, lớn nhất: findMin, findMax

Việc tìm phần tử nhỏ nhất (lớn nhất) trên cây nhị phân tìm kiếm có thể được thực hiện nhờ việc di chuyển trên cây mà không cần so sánh khóa. Cụ thể:

- Để tìm phần tử nhỏ nhất trên BST, ta đi theo con trái cho đến khi gặp NULL.
  - Để tìm phần tử lớn nhất trên BST, ta đi theo con phải cho đến khi gặp NULL.
- Thuật toán có thể mô tả trong các sơ đồ sau:

**find-min(T)**

1. **while** *left[T]* ≠ NULL
2.   **do** *T* ← *left[T]*
3. **return** *T*

**find-max(T)**

1. **while** *right[T]* ≠ NULL
2.   **do** *T* ← *right[T]*
3. **return** *T*

Cài đặt đệ quy của thuật toán find-min có thể thực hiện như sau:

```
TreeNode* find_min(TreeNode * T) {
 /* luôn di theo con trái */
 if (T == NULL) return(NULL);
 else
 if (T->leftPtr == NULL) return(T);
 else return(find_min(T->leftPtr));
}
```

Cài đặt đệ quy của thuật toán find-max có thể thực hiện như sau:

```
TreeNode* find_max(TreeNode* T) {
 /* luôn di theo con phải */
 if (T != NULL)
 while (T->rightPtr != NULL)
 T = T->rightPtr;
 return(T);
}
```

### Ké cận trước và Ké cận sau (Predecessor và Successor)

Ké cận sau (Successor) của nút *x* là nút *y* sao cho *key[y]* là khóa nhỏ nhất còn lớn hơn *key[x]*. Ké cận sau của nút với khóa lớn nhất là NIL.

Ké cận trước (Predecessor) của nút *x* là nút *y* sao cho *key[y]* là khóa lớn nhất còn nhỏ hơn *key[x]*. Ké cận trước của nút với khóa nhỏ nhất là NIL.

Việc tìm kiếm kế cận sau/trước được thực hiện mà không cần thực hiện so sánh khóa.

**Tìm kế cận sau:** Có hai tình huống:

1. Nếu  $x$  có con phải thì kế cận sau của  $x$  sẽ là nút  $y$  với khóa key[y] nhỏ nhất trong cây con phải của  $x$  (nói cách khác  $y$  là nút trái nhất trong cây con phải của  $x$ ).

– Để tìm  $y$  có thể dùng  $\text{find\_min}(x \rightarrow \text{rightPtr})$ :  $y = \text{find\_min}(x \rightarrow \text{rightPtr})$ .

– Hoặc bắt đầu từ gốc của cây con phải luôn đi theo con trái cho đến khi gặp nút không có con trái thì chính là nút  $y$  cần tìm.

2. Nếu  $x$  không có con phải thì kế cận sau của  $x$  là tổ tiên gần nhất có con trái hoặc là  $x$  hoặc là tổ tiên của  $x$ . Để tìm kế cận sau:

– Bắt đầu từ  $x$  cần di chuyển lên trên (theo con trỏ parent) cho đến khi gặp nút  $y$  có con trái đầu tiên thì dừng:  $y$  là kế cận sau của  $x$ .

– Nếu không thể di chuyển tiếp được lên trên (tức là đã đến gốc) thì  $x$  là nút lớn nhất (vì thế  $x$  không có kế cận sau).

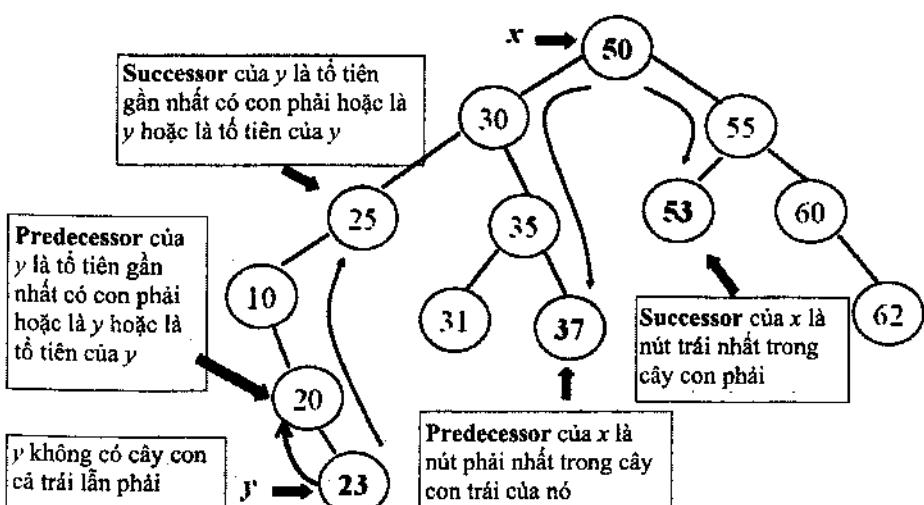
**Tìm kế cận trước:** Tương tự như tìm kế cận sau, có hai tình huống:

– Nếu  $x$  có con trái thì kế cận trước của  $x$  sẽ là nút  $y$  với khóa key[y] lớn nhất trong cây con trái của  $x$  (nói cách khác  $y$  là nút phải nhất trong cây con trái của  $x$ ):

$y = \text{find\_max}(x \rightarrow \text{leftPtr})$

– Nếu  $x$  không có con trái thì kế cận trước của  $x$  là tổ tiên gần nhất có con phải hoặc là  $x$  hoặc là tổ tiên của  $x$ .

Hình vẽ dưới đây minh họa cho việc tìm kế cận sau và kế cận trước:



## Thuật toán tìm kiếm trên BST

Để tìm kiếm một khóa trên cây ta tiến hành như sau:

– Nếu khóa cần tìm nhỏ hơn khóa của nút hiện tại thì tiếp tục tìm kiếm ở cây con trái.

– Trái lại, nếu khóa cần tìm lớn hơn khóa của nút hiện tại thì tiếp tục tìm kiếm ở cây con phải.

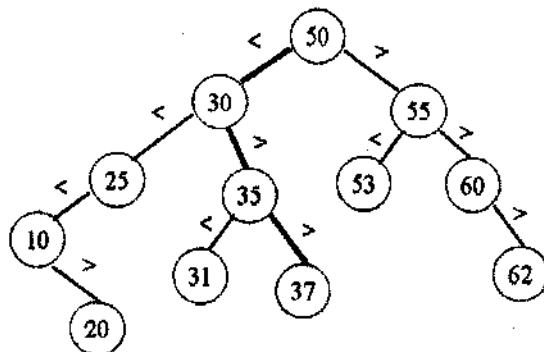
Kết quả tìm kiếm cần đưa ra:

– Nếu tìm thấy (nghĩa là khóa cần tìm bằng khóa của nút hiện tại), thì trả lại con trỏ đến nút chứa khóa cần tìm.

Ngược lại, trả lại con trỏ NULL.

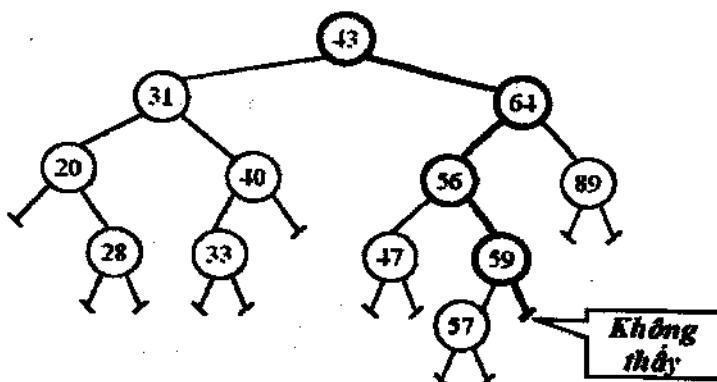
Như vậy, việc tìm kiếm đòi hỏi di chuyển trên cây. Do độ dài của đường di chuyển không vượt quá độ cao của cây, nên thời gian tìm kiếm sẽ là  $O(h)$ , trong đó  $h$  là chiều cao của cây.

**Ví dụ:** Tìm kiếm trên BST: Tìm 37.



Sau bốn phép so sánh ta tìm được nút với khóa 37 (xem đường tô đậm trên cây).

**Ví dụ:** Tìm 60 trên cây BST sau đây.



Sau bốn phép so sánh ta gặp NULL, khóa 60 không có trên cây.

Cài đặt trên C:

```
TreeNode* search(TreeNode* nodePtr, float target)
{
 if (nodePtr != NULL)
 {
 if (target < nodePtr->key)
 {
 nodePtr = search(nodePtr->leftPtr, target);
 }
 else if (target > nodePtr->key)
 {
 nodePtr = search(nodePtr->rightPtr, target);
 }
 }
 return nodePtr;
}
```

Đoạn chương trình gọi có thể có dạng sau:

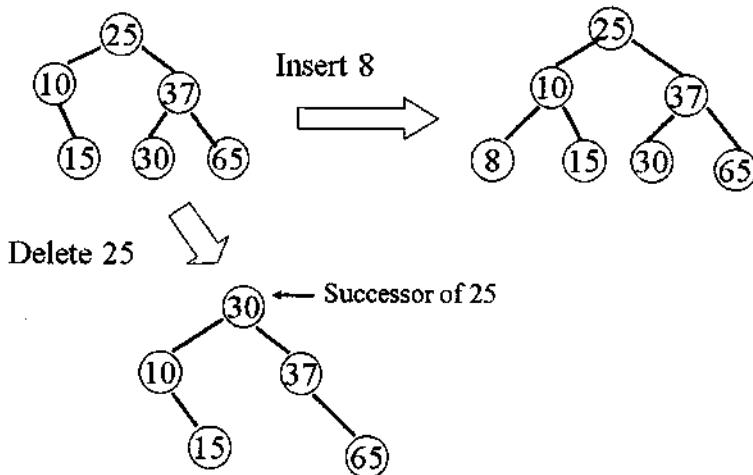
```
/* ... một số lệnh ... */
printf("Enter target ");
scanf("%f", &item);
if (search(rootPtr, item) == NULL)
{
 printf("Không tìm thấy\n");
}
else
{
 printf("Tim thấy\n");
}

/* ... các lệnh khác ... */
```

## Thao tác bô sung và loại bỏ trên BST

Việc bô sung thêm một nút (hoặc loại bỏ một nút) cần được thực hiện sao cho sau khi bô sung (loại bỏ), cây thu được vẫn là cây BST.

Ví dụ:



### Bô sung (Insertion)

#### Thuật toán bô sung (Insert)

– Tạo nút mới chứa phần tử cần chèn.

– Di chuyển trên cây để tìm cha của nút mới: So sánh khóa của phần tử cần chèn với khóa của nút đang xét (bắt đầu là gốc của cây), nếu khóa của phần tử cần chèn lớn hơn (nhỏ hơn) khóa của nút đang xét thì rẽ theo con phải (con trái) của nút đang xét. Nếu gặp NULL thì dừng, nút đang xét là cha cần tìm.

– Gắn nút mới như là con của nút cha tìm được. Chú ý rằng nút mới luôn là lá.

#### Cài đặt đệ quy của thao tác bô sung

– Thông số đầu vào:

+ Con trỏ đến nút đang xét (thoạt tiên là nút gốc);

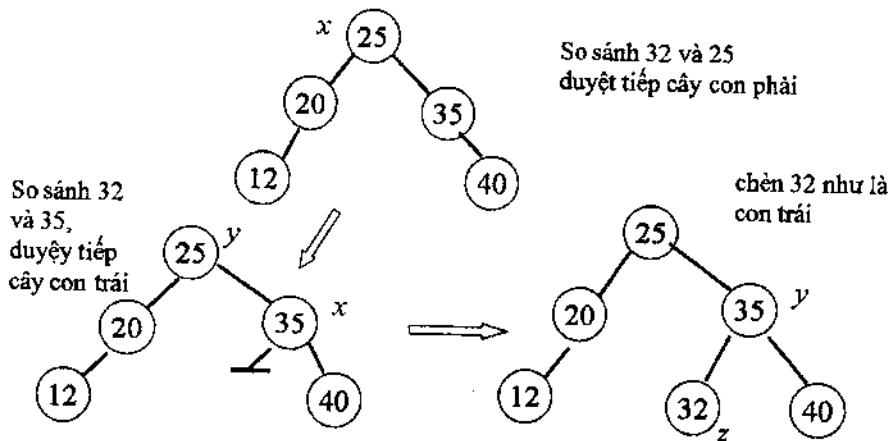
+ Phần tử cần bô sung.

– Nếu nút hiện thời là NULL:

+ Tạo nút mới và trả lại nó;

+ Trái lại, nếu khóa của phần tử bô sung nhỏ hơn (lớn hơn) khóa của nút hiện thời, thì tiếp tục quá trình với nút hiện thời là nút con trái (con phải).

Ví dụ: Bổ sung  $z = 32$ .



### Cài đặt trên C

```
TreeNode* insert(TreeNode* nodePtr, float item)
{
 if (nodePtr == NULL)
 {
 nodePtr = makeTreeNode(item);
 }
 else if (item < nodePtr->key)
 {
 nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
 }
 else if (item > nodePtr->key)
 {
 nodePtr->rightPtr = insert(nodePtr->rightPtr,
item);
 }
 return nodePtr;
}
```

Thời gian tính:  $O(h)$ , trong đó  $h$  là độ cao của BST.

Hàm gọi đến Insert có thể là:

```
/* ... còn các lệnh khác ở đây ... */
printf("Enter number of items ");
scanf("%d", &n);
```

```

for (i = 0; i < n; i++) {
 scanf("%f", &item);
 rootPtr = insert(rootPtr, item);
}

```

/\* ... và còn những lệnh tiếp tục ... \*/

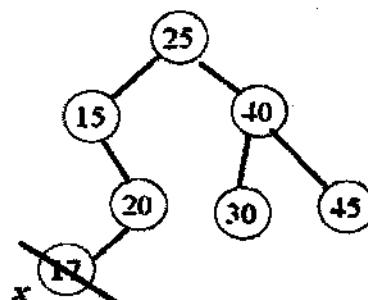
### Thao tác loại bỏ – delete

Khi loại bỏ một nút, cần phải đảm bảo cây thu được vẫn là cây nhị phân. Vì thế, khi xóa cần phải xét cẩn thận các con của nó. Có bốn tình huống cần xét:

- Tình huống 1: Nút cần xóa là lá.
- Tình huống 2: Nút cần xóa chỉ có con trái.
- Tình huống 3: Nút cần xóa chỉ có con phải.
- Tình huống 4: Nút cần xóa có hai con.

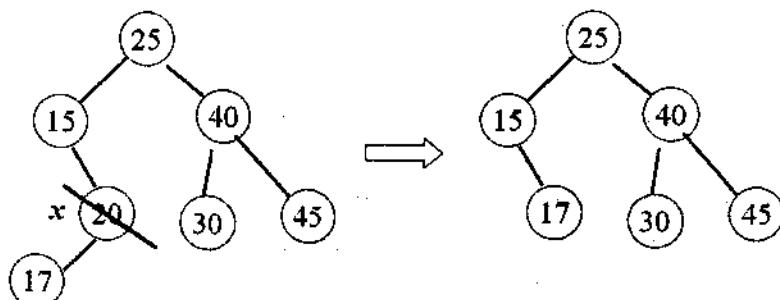
**Tình huống 1:** Nút cần xóa  $x$  là lá (leaf node).

**Thao tác:** Chữa lại nút cha của  $x$  có con rỗng.



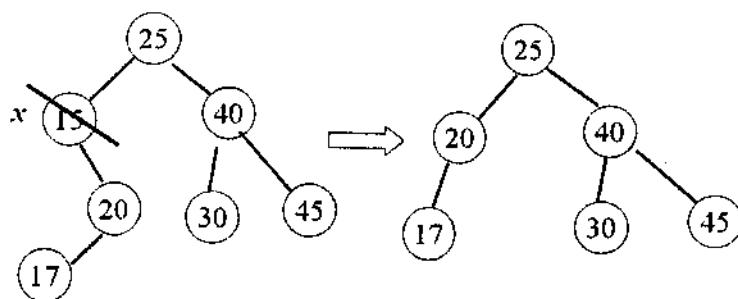
**Tình huống 2:** Nút cần xóa  $x$  có con trái mà không có con phải.

**Thao tác:** Gắn cây con trái của  $x$  vào cha.



**Tình huống 3:** Nút cần xóa  $x$  có con phải mà không có con trái.

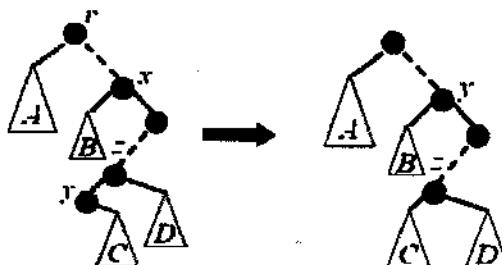
**Thao tác:** Gắn cây con phải của  $x$  vào cha.



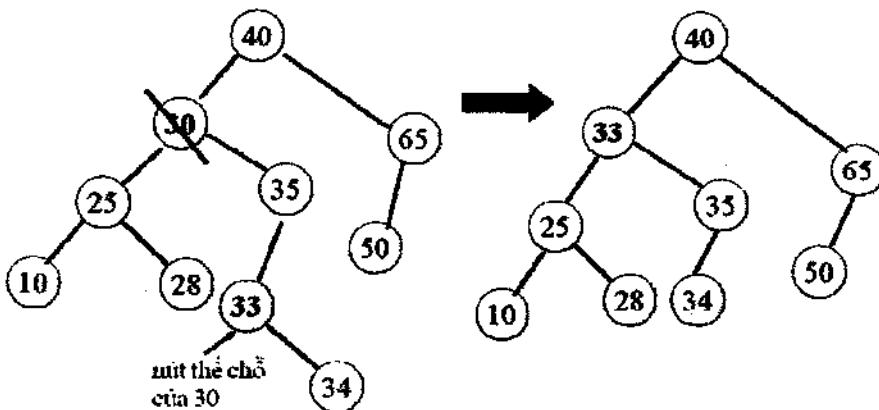
**Tình huống 4:** Nút  $x$  có hai con.

**Thao tác:**

- Chọn nút  $y$  để thế vào chỗ của  $x$ , nút  $y$  sẽ là successor của  $x$ .  $y$  là giá trị nhỏ nhất còn lớn hơn  $x$ , nói cách khác  $y$  là giá trị nhỏ nhất ở cây con phải của  $x$ .
- Gỡ nút  $y$  khỏi cây.
- Nối con phải của  $y$  vào cha của  $y$ .
- Cuối cùng, thay thế  $y$  vào nút cần xóa.



**Ví dụ:**



Hàm trên C dưới đây minh họa cài đặt thao tác xóa phần tử có  $key = x$ .

```
TreeNode* delete(TreeNode * T, float x) {
 TreeNode tmp;
 if (T == NULL) printf("Not found\n");
 else if (x < T->key) /* di bên trái */
 T->leftPtr = delete(T->leftPtr, x);
 else if (x > T->key) /* di bên phải */
 T->rightPtr = delete(T->rightPtr, x);
 else /* tìm được phần tử cần xóa */
 if (T->leftPtr && T->rightPtr) {
 /* Tình huống 4: phần tử thẻ chỗ là
 phần tử min ở cây con phải */
 tmp = find_min(T->right);
 T->key = tmp->key;
 T->rightPtr = delete(T->key, T->rightPtr);
 }
 else
 { /* có 1 con hoặc không có con */
 tmp = T;
 if (T->leftPtr == NULL)
 /* chỉ có con phải
 hoặc không có con */
 T = T->rightPtr;
 else
 if (T->rightPtr == NULL)
 /* chỉ có con trái */
 T = T->leftPtr;
 free(tmp);
 }
 return(T);
}
```

### Sắp xếp nhờ sử dụng BST

Do duyệt cây BST theo thứ tự giữa cho ta dãy khóa được sắp xếp, nên ta có thể ứng dụng cây BST để giải quyết bài toán sắp xếp. Để sắp xếp dãy phần tử, ta có thể thực hiện như sau:

– Xây dựng cây BST tương ứng với dãy số đã cho bằng cách bổ sung (insert) các phần tử của dãy số vào cây nhị phân tìm kiếm bắt đầu từ cây rỗng.

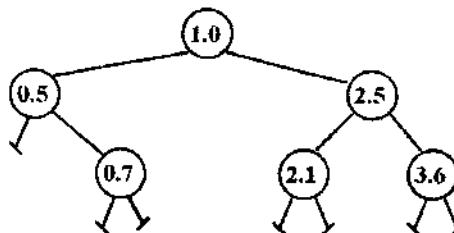
– Duyệt cây BST thu được theo thứ tự giữa để đưa ra dãy được sắp xếp.

### Ví dụ

Sắp xếp dãy sau nhờ sử dụng BST:

|     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|
| 1.0 | 2.5 | 0.5 | 0.7 | 3.6 | 2.1 |
|-----|-----|-----|-----|-----|-----|

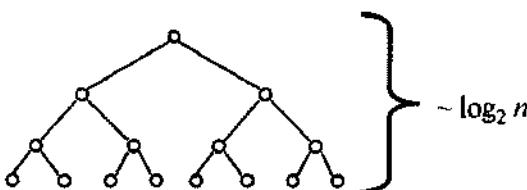
Xây dựng cây BST:



Duyệt cây thu được theo thứ tự giữa cho ta dãy được sắp xếp.

### Phân tích hiệu quả của sắp xếp nhờ sử dụng cây BST

– Tính huống trung bình:  $O(n \log n)$ , bởi vì chèn phần tử thứ  $(i + 1)$  tốn quãng  $\log_2(i)$  phép so sánh.



– Tính huống tồi nhất:  $O(n^2)$ , bởi vì bổ sung phần tử thứ  $(i + 1)$  tốn quãng  $i$  phép so sánh. Ví dụ, bổ sung dãy: 1, 3, 7, 9, 11, 15.

### Độ phức tạp trung bình của các thao tác với BST

Người ta chỉ ra được rằng độ cao trung bình của BST là:

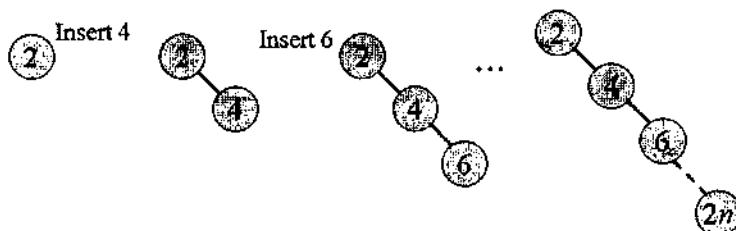
$$h = O(\log n).$$

Từ đó suy ra độ phức tạp trung bình của các thao tác với BST là:

|           |       |               |
|-----------|-------|---------------|
| Insertion | ..... | $O(\log n)$   |
| Deletion  | ..... | $O(\log n)$   |
| Find Min  | ..... | $O(\log n)$   |
| Find Max  | ..... | $O(\log n)$   |
| BST Sort  | ..... | $O(n \log n)$ |

## Thời gian tính tồi nhất

Tất cả các phép toán cơ bản (Search, Successor, Predecessor, Minimum, Maximum, Insert, Delete) trên BST với chiều cao  $h$  đều có thời gian tính là  $O(h)$ . Trong tình huống tồi nhất  $h = n$ , ví dụ hình vẽ sau đây minh họa cho tình huống này.



Vấn đề đặt ra là có cách nào đảm bảo  $h = O(\log n)$ ? Như đã biết, cây nhị phân có  $n$  nút có độ cao tối thiểu là  $\log n$ . Do đó, tình huống tốt nhất xảy ra khi ta dụng được BST là cây nhị phân đầy đủ (cây có độ cao thấp nhất có thể được). Có hai cách tiếp cận nhằm đảm bảo độ cao của cây là  $O(\log n)$ :

- Luôn giữ cho cây cân bằng tại mọi thời điểm (AVL Trees).
- Thỉnh thoảng kiểm tra lại xem cây có "quá mất cân bằng" hay không và nếu điều đó xảy ra thì cần tìm cách cân bằng lại nó (Splay Trees [Tarjan]).

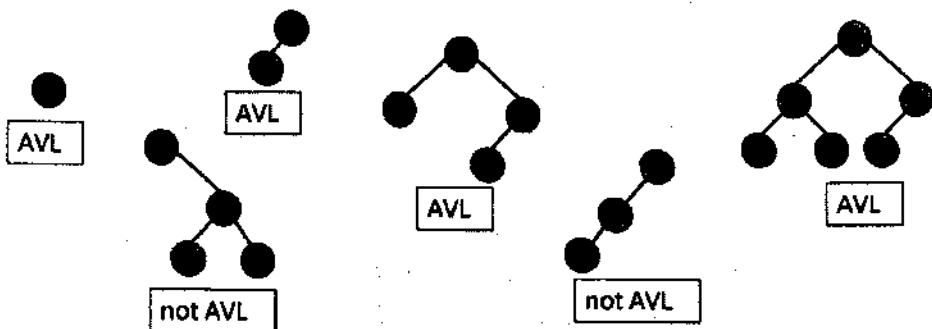
Trong mục tiếp theo ta trình bày cách tiếp cận thứ nhất – cây AVL.

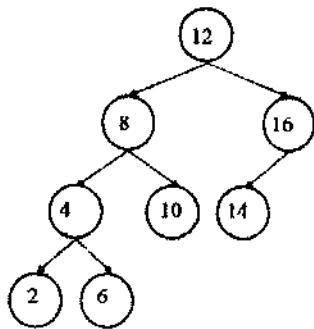
## 6.3. CÂY NHỊ PHÂN TÌM KIÉM CÂN BẰNG – CÂY AVL

### 6.3.1. Định nghĩa

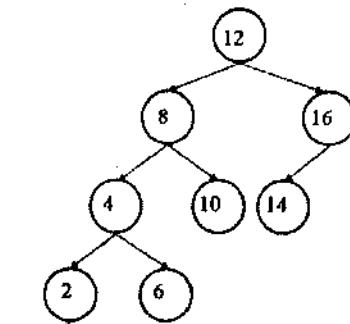
**Định nghĩa:** Cây AVL là cây nhị phân tìm kiếm thỏa mãn *tính chất AVL* sau đây: chiều cao của cây con trái và cây con phải của gốc chỉ sai nhau không quá 1 và cả cây con trái và cây con phải đều là cây AVL.

Ví dụ:





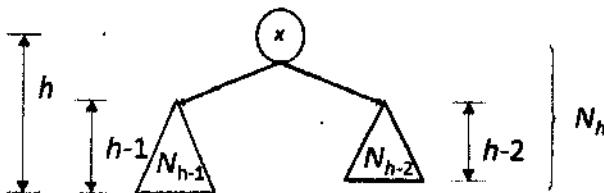
Cây AVL



1  
Không là cây AVL  
(nút 8 và 12 không thỏa  
mãn tính chất AVL)

**Tính chất AVL (tính chất cân bằng):** Chênh lệch giữa độ cao của cây con trái và độ cao của cây con phải của một nút bất kỳ trong cây không quá 1.

**Chú ý:** Cây nhị phân đầy đủ và hoàn chỉnh có tính chất này, nhưng cây có tính chất AVL không nhất thiết phải là cây đầy đủ hay hoàn chỉnh.



Số nút của cây:  $N_h = N_{h-1} + N_{h-2} + 1$ .

Cây AVL được Adelson-Velskii và Landis đề xuất từ năm 1962:

– Cây AVL là cây BST được giữ sao cho luôn ở trạng thái cân bằng (cân đối). Nếu việc bổ sung hay loại bỏ dẫn đến mất tính cân bằng của cây thì cần tiến hành khôi phục ngay lập tức.

– Các thao tác: tìm kiếm, bổ sung, loại bỏ đều có thể thực hiện với cây AVL có  $n$  nút trong thời gian  $O(\log n)$  (cả trong tình huống trung bình lẫn tồi nhất).

Để dễ hình dung, tạm bỏ qua khóa của các nút và sử dụng các ký hiệu / \ - // // để thể hiện “yếu tố cân bằng” của nút:

\ : phải nặng hơn

$$h(\text{right subtree}) = 1 + h(\text{left subtree})$$

/ : trái nặng hơn

$$h(\text{left subtree}) = 1 + h(\text{right subtree})$$

- : bằng nhau

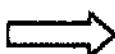
$$h(\text{right subtree}) = h(\text{left subtree})$$

// : mất cân bằng phải

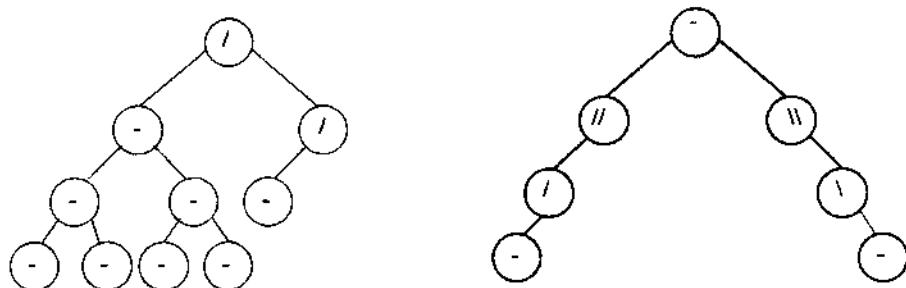
$$h(\text{right subtree}) > 1 + h(\text{left subtree})$$

// : mất cân bằng trái

$$h(\text{left subtree}) > 1 + h(\text{right subtree})$$



**Ví dụ:** Ta không chỉ ra giá trị của các khóa. Giả thiết là chúng thỏa mãn tính chất BST.



**Chú ý:** Cây AVL không nhất thiết phải là cây đầy đủ và cũng không nhất thiết phải là cây có số mức ít nhất. Thao tác bổ sung và loại bỏ được thực hiện như BST. Tuy nhiên, sau mỗi lần thực hiện thao tác, *nếu như xảy ra mất cân bằng, chúng ta cần hiệu chỉnh cây để đảm bảo cây vẫn có tính chất AVL.*

### Kiểm tra tính AVL

**Định nghĩa:** Chiều cao (height) của nút  $x$ , ký hiệu là  $h(x)$ , được định nghĩa như sau:

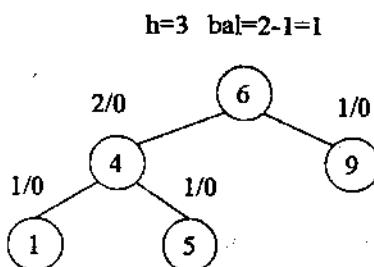
- $h(x) = 1$ , nếu  $x$  là lá;  $h(x) = 0$ , nếu  $x = \text{NULL}$ .
- $h(x) = \max\{h_{left}(x), h_{right}(x)\} + 1$ , nếu  $x$  là nút trong, trong đó  $h_{left}(x)$  ( $h_{right}(x)$ ) là chiều cao của cây con trái (phải) của  $x$ .

**Định nghĩa:** Hệ số cân bằng (balance factor) của nút  $x$ , ký hiệu là  $\text{bal}(x)$ , bằng hiệu giữa chiều cao của cây con phải và cây con trái của  $x$ .

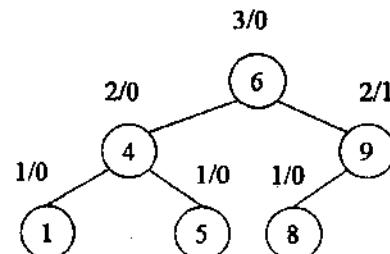
Như vậy, cây nhị phân tìm kiếm là cây AVL nếu như hệ số cân bằng của mỗi nút của nó là 0, 1 hay 2.

**Ví dụ:** Chiều cao của nút.

Tree A (AVL)

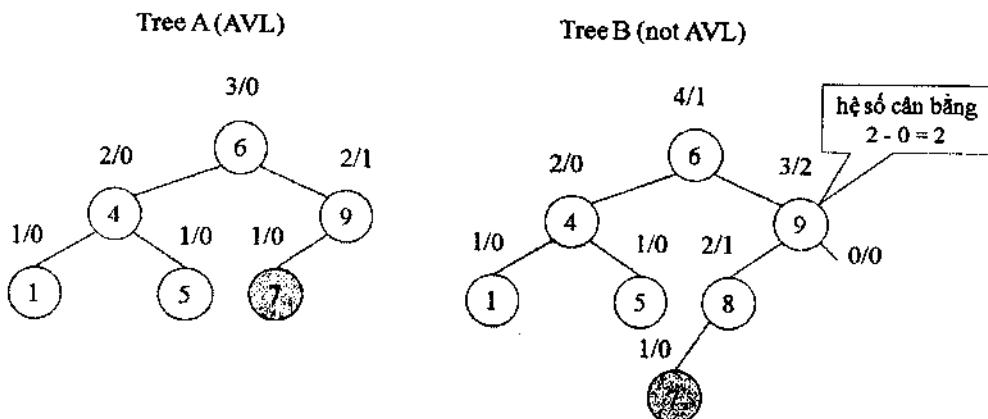


Tree B (AVL)



Ký hiệu: chiều cao của nút:  $h(x)$ , hệ số cân bằng:  $bal = h_{left}(x) - h_{right}(x)$ , chiều cao của nút rỗng = 0.

Chiều cao của nút sau khi chèn thêm nút 7:



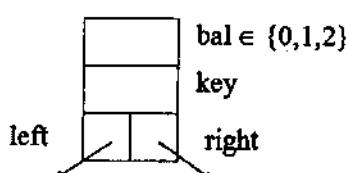
### Biểu diễn cây AVL

```
struct TreeNode
```

```
{ int bal;
 float key;
 struct TreeNode* left;
 struct TreeNode* right;
};
```

```
typedef struct TreeNode AvlTree;
```

Cấu trúc của một nút được minh họa trong hình vẽ sau:



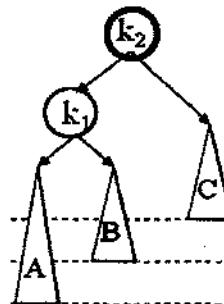
### Khôi phục tính cân bằng của cây

- Trước khi thực hiện thao tác cây là cân bằng.
- Sau khi thực hiện thao tác bổ sung hoặc loại bỏ (insertion or deletion operation), cây có thể trở thành mất cân bằng. Như vậy, cần xác định cây con mất cân bằng.

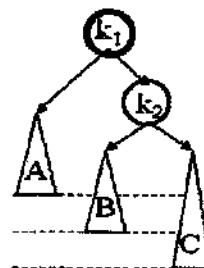
– Chiều cao của một cây con bất kỳ chỉ có thể tăng hoặc giảm nhiều nhất là 1. Vì thế, nếu xảy ra mất cân bằng thì chênh lệch chiều cao giữa hai cây con chỉ có thể là 2.

Có bốn tình huống với chênh lệch chiều cao là 2.

**Tình huống 1:** Cây con trái cao hơn cây con phải nguyên do bởi cây con trái của con trái.

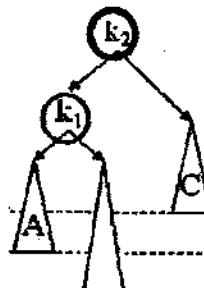


**Tình huống 2:** Cây con phải cao hơn cây con trái nguyên do bởi cây con phải của con phải.

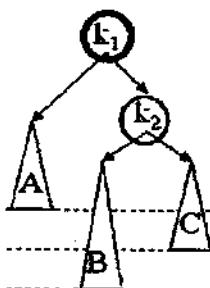


Trong tình huống 1 và 2 ta, khôi phục cân bằng nhờ sử dụng phép quay đơn: quay phải hoặc quay trái (single rotation: right rotation or left rotation).

**Tình huống 3:** Cây con trái cao hơn cây con phải nguyên do bởi cây con phải của con trái.

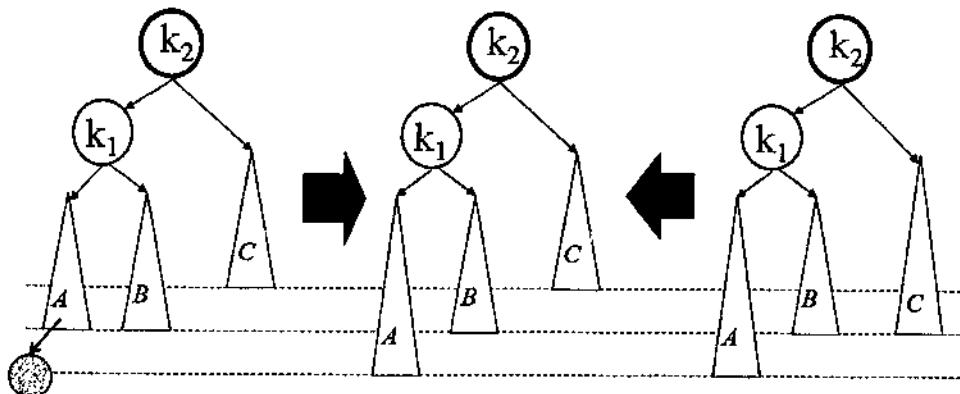


**Tình huống 4:** Cây con phải cao hơn cây con trái nguyên do bởi cây con trái của con phải.



Trong tình huống 3 và 4, ta khôi phục cân bằng nhờ sử dụng phép quay kép: quay phải rồi quay trái hoặc quay trái rồi quay phải (double rotation: right-left rotation or left-right rotation).

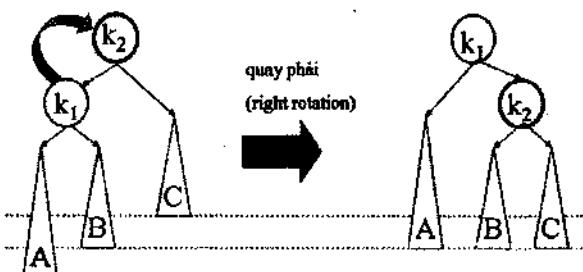
**Ví dụ:** Hình vẽ sau đây minh họa tình huống khi các thao tác bổ sung hoặc loại bỏ có thể dẫn đến tình huống 1.



**Insertion** – Chiều cao của cây con  $A$  có thể tăng thêm 1 sau khi bổ sung.

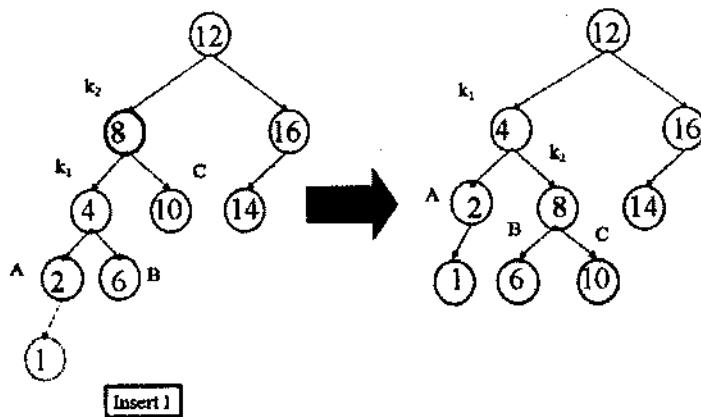
**Deletion** – Chiều cao của cây con  $C$  có thể giảm đi 1 sau khi loại bỏ.

**Xử lý tình huống 1: Thực hiện phép quay phải (right rotation)**

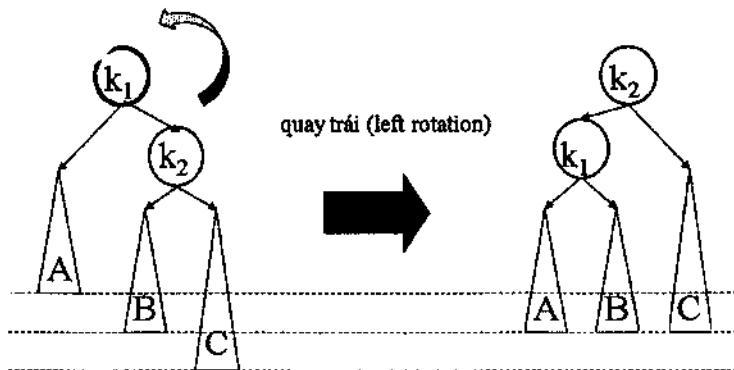


- Thời gian  $O(1)$  và khôi phục được cân bằng.
- Insertion – Chiều cao của cây con  $A$  giảm đi 1. Sau khi quay, cây có độ cao như trước khi thực hiện bù sung.
- Deletion – Chiều cao của cây con  $C$  tăng thêm 1. Sau khi quay, chiều cao của cây giảm đi 1 so với trước khi loại bỏ.

**Ví dụ:** Tinh huống 1 xảy ra khi thực hiện bù sung.

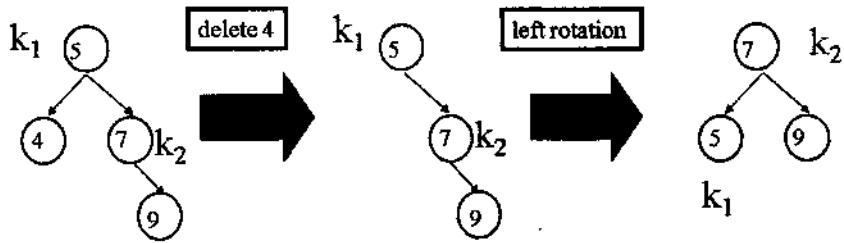


**Xử lý tình huống 2: Thực hiện phép quay trái (left rotation)**

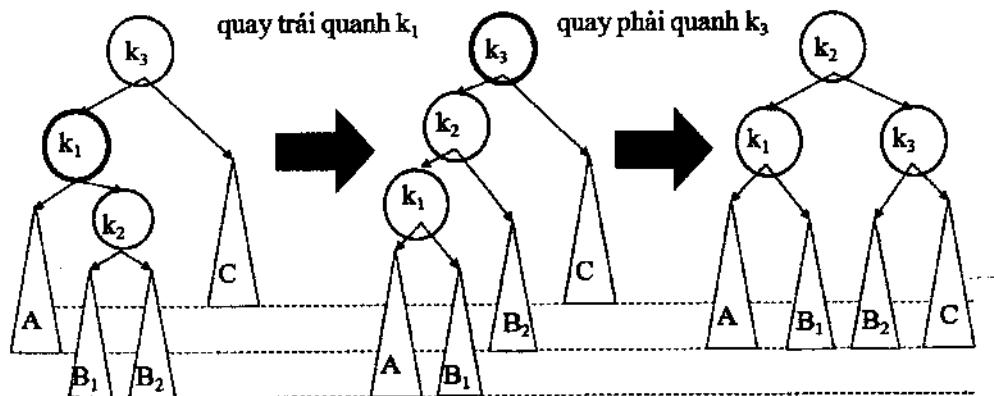


Phân tích giống tình huống 1.

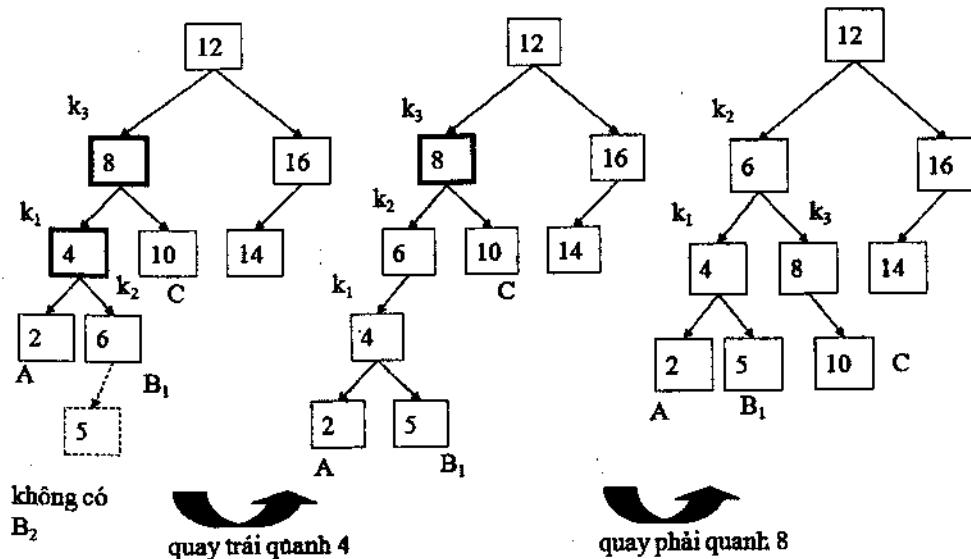
**Ví dụ:** Tinh huống 2 xảy ra sau khi thực hiện loại bỏ.



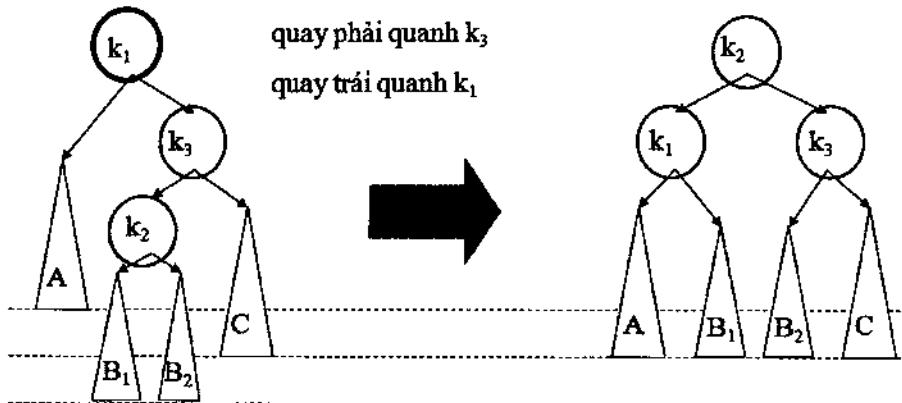
Xử lý tình huống 3: Thực hiện phép quay kép, quay trái rồi quay phải



Ví dụ: Tình huống 3 sau khi thực hiện bổ sung.



#### Xử lý tình huống 4: Thực hiện phép quay kép, quay phải rồi quay trái



Phân tích tương tự như tình huống 3.

#### 6.3.2. Các thao tác với cây AVL

##### Thuật toán thực hiện thao tác bổ sung

- Chèn nút mới được thực hiện giống như trong cây BST.
- Lần theo đường đi từ nút mới bổ sung về gốc, với mỗi nút  $x$  kiểm tra  $|h_{left}(x) - h_{right}(x)| \leq 1$ .
- Nếu đúng, thì tiếp tục với  $parent(x)$ . Nếu ngược lại, cần khôi phục cân bằng nhờ thực hiện phép quay đơn hoặc quay kép.

**Chú ý:** Khi ta đã khôi phục được tính cân bằng của một nút  $x$ , thì đồng thời cũng khôi phục được tính cân bằng của tất cả các tổ tiên của nó. Nghĩa là việc khôi phục chỉ phải tiến hành với không quá một nút vi phạm.

Do việc khôi phục cân bằng của một nút đòi hỏi thời gian  $O(1)$ , nên thao tác bổ sung đòi hỏi thời gian  $O(h)$ .

##### Thuật toán thực hiện thao tác loại bỏ (Deletion)

- Thực hiện loại bỏ nút  $x$  giống như đối với cây BST.
- Tiếp đến duyệt từ nút lá mới đến gốc.
- Với mỗi nút  $x$  trên đường đi, kiểm tra tính cân bằng của nó. Nếu  $x$  là cân bằng thì tiếp tục với  $parent(x)$ . Ngược lại, khôi phục cân bằng của nút  $x$  nhờ thực hiện các phép quay.
- Khác với thuật toán chèn, sau khi khôi phục cân bằng của nút  $x$  ta phải tiếp tục quá trình cho đến khi gặp gốc của cây, bởi vì việc khôi phục cân bằng của nút  $x$  không đảm bảo khôi phục được tính cân bằng của tổ tiên của nó.

Dễ thấy việc khôi phục cân bằng của một nút đòi hỏi thời gian  $O(1)$ , còn đường đi tới gốc có độ dài không quá  $h$ , nên thao tác loại bỏ đòi hỏi thời gian  $O(h)$ .

**Chú ý:** Do cài đặt thao tác xóa đối với cây AVL khá phức tạp, nên nếu việc loại bỏ không phải làm thường xuyên thì có thể sử dụng ý tưởng "*lazy deletion*": chỉ đánh dấu xóa chứ không cần thực sự xóa.

### Phân tích phức tạp

Ta sẽ đưa ra đánh giá chiều cao  $h$  của cây AVL với  $n$  nút. Ký hiệu  $T_h$  là cây AVL chiều cao  $h$  có số lượng nút ít nhất.  $T_h$  phải chứa gốc và một cây con với độ cao  $h-1$  còn cây con kia có độ cao  $h-2$ . Do đó, nếu ký hiệu  $N_h$  là số lượng nút của  $T_h$  thì  $N_h$  phải thỏa mãn công thức đệ quy sau đây:

$$N_h = \begin{cases} 1, & \text{khi } h=0 \\ 2, & \text{khi } h=1 \\ N_{h-1} + N_{h-2} + 1 & \text{khi } h \geq 2 \end{cases}$$

Từ đó, chứng minh bằng quy nạp, ta có:

$$N_{h+1} = N_{h-1} + N_h + 1 \geq F_{h+2} - 1 + F_{h+1} = F_{h+3} - 1,$$

trong đó:  $F_h$  là số Fibonacci thứ  $h$ .

Số Fibonacci thứ  $n$  được tính bởi công thức:

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \bar{\phi}^n), \quad \phi = \frac{1+\sqrt{5}}{2}, \quad \bar{\phi} = \frac{1-\sqrt{5}}{2}.$$

Ta có:

$$\begin{aligned} N_h &\geq F_{h+2} - 1 \Rightarrow N_h \geq \frac{1}{\sqrt{5}}\phi^{h+2} - 2 \quad (|\bar{\phi}| < 1) \\ &\Rightarrow (N_h + 2)\sqrt{5} \geq \phi^{h+2} \\ &\Rightarrow \log_\phi((N_h + 2)\sqrt{5}) \geq h + 2 \\ &\Rightarrow h \leq \log_\phi((N_h + 2)\sqrt{5}) - 2 \\ &\Rightarrow h \leq 1.440 \log_2(N_h + 2) - 0.328 \end{aligned}$$

Mặt khác, cây AVL với  $n$  nút có độ cao thấp nhất là  $\log(n+1)$ , khi nó là cây đầy đủ, tức là ta có:

$$h \geq \log(n+1).$$

Vậy ta có bất đẳng thức sau đây đối với độ cao của cây AVL có  $n$  nút:

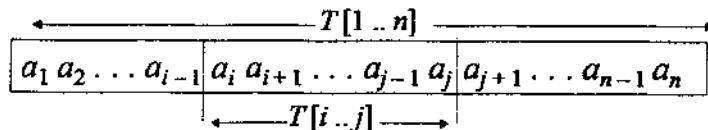
$$\log(n+1) \leq h \leq 1.44 \log(n+2) - 0.328.$$

Do đó,  $h = O(\log n)$ . Từ bất đẳng thức cuối cùng này ta suy ra khẳng định: "Tất cả các phép toán với cây AVL đều được thực hiện với thời gian  $O(\log n)$ ".

## 6.4. TÌM KIẾM XÂU MẪU (STRING SEARCHING)

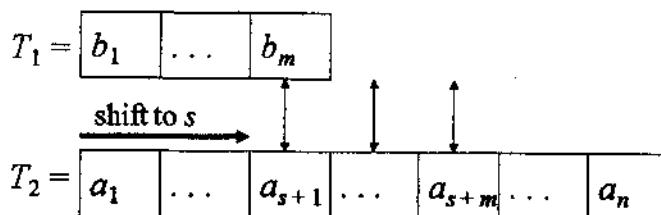
### 6.4.1. Phát biểu bài toán

**Xâu (Strings):** Xâu là dãy ký hiệu lấy từ bảng ký hiệu (alphabet)  $\Sigma$ . Ký hiệu  $T[i..j]$  là xâu con của xâu  $T$  bắt đầu từ vị trí  $i$  và kết thúc ở vị trí  $j$ .



**Trượt (Shifts):** Giả sử  $T_1$  và  $T_2$  là hai xâu, trong đó  $|T_1| = m$  và  $|T_2| = n$ . Ta nói  $T_1$  xuất hiện nhờ trượt đến  $s$  trong  $T_2$  nếu:

$$T_1[1..m] = T_2[s+1..s+m].$$



**Vị trí khớp và không khớp:** Giả sử  $T_1$  và  $T_2$  là hai xâu. Nếu  $T_1$  xuất hiện nhờ trượt đến  $s$  trong  $T_2$  thì  $s$  được gọi là vị trí khớp của  $T_1$  trong  $T_2$ , ngược lại  $s$  được gọi là vị trí không khớp.

#### Bài toán tìm kiếm xâu mẫu – (The String Matching Problem)

Cho xâu  $T$  độ dài  $n$  ( $T$  được gọi là văn bản) và xâu  $P$  độ dài  $m$  ( $P$  được gọi là xâu mẫu *pattern*). Bài toán đặt ra là: Tìm tất cả các vị trí khớp của  $P$  trong  $T$ .

#### Các ứng dụng của bài toán:

- Trong thu thập thông tin (information retrieval).
- Trong soạn thảo văn bản (text editing).
- Trong tính toán sinh học (computational biology).

...

**Ví dụ:**  $T = 000010001010001$  và  $P = 0001$ , các vị trí khớp là:

-  $s = 1$

+  $T = 000010001010001$

+  $P = \quad \quad \quad 0001$

-  $s = 5$

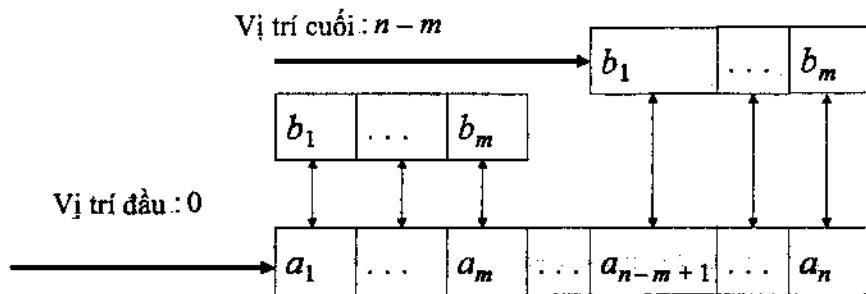
+  $T = 000010001010001$

+  $P = \quad \quad \quad 0001$

- s = 11  
 + T = 000010001010001  
 + P = 0001

### 6.4.2. Thuật toán trực tiếp – Naïve algorithm

Ý tưởng: Trượt đến từng vị trí  $s = 0, 1, \dots, n - m$ , với mỗi vị trí kiểm tra xem xâu mẫu có xuất hiện ở vị trí đó hay không.



```

void NaiveSM(char *P, int m, char *T, int n) {
 int i, j;
 /* Searching */
 for (j = 0; j <= n - m; ++j) {
 for (i = 0; i < m && P[i] == T[i + j]; ++i);
 if (i >= m) OUTPUT(j);
 }
}

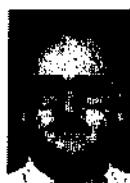
```

Thời gian tính trong tình huống tồi nhất =  $O(nm)$ .

Ví dụ: T = 000010001010001 và P = 0001.

|                     |                          |
|---------------------|--------------------------|
| T = 000010001010001 |                          |
| s=0      P = 0001   | $\Rightarrow$ không khớp |
| s=1      P = 0001   | $\Rightarrow$ khớp       |
| s=2      P = 0001   | $\Rightarrow$ không khớp |
| s=3      P = 0001   | $\Rightarrow$ không khớp |
| s=4      P = 0001   | $\Rightarrow$ không khớp |
| s=5      P = 0001   | $\Rightarrow$ khớp       |
| s=6      P = 0001   | $\Rightarrow$ không khớp |
| ...                 |                          |

### 6.4.3. Thuật toán Boyer–Moore



Robert-Stephen Boyer



J. Strother Moore

#### Boyer–Moore Algorithm

- Làm việc tốt khi  $P$  dài và  $\Sigma$  lớn.
- Chúng ta sẽ xét sự khớp nhau bằng cách duyệt từ phải qua trái.  
Trước hết hãy quan sát lại thuật toán trực tiếp làm việc theo thứ tự duyệt này.

#### Thuật toán trực tiếp cải biến

for  $s \leftarrow 0$  to  $n - m$  do

$j \leftarrow m$

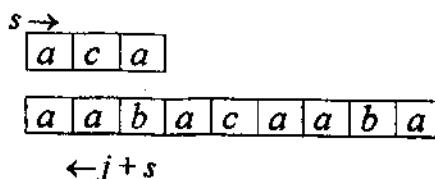
while  $j > 0$  and  $T[j + s] = P[j]$  do

$j \leftarrow j - 1$

if  $j = 0$  then

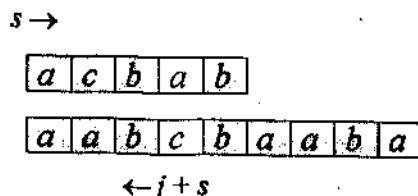
print  $s$  “là vị trí khớp”

#### Xét ví dụ



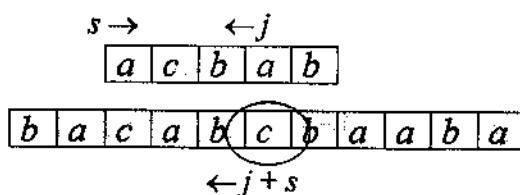
Nhận thấy rằng,  $b$  không xuất hiện ở bất cứ đâu trong  $P$ , vì thế có thể trượt  $P$  bỏ qua  $b$ .

#### Ví dụ khác



Ký hiệu “ $c$ ” xuất hiện ở vị trí 2 trong  $P$ , vì thế có thể trượt  $P$  sao cho hai ký hiệu “ $c$ ” được đóng thảng.

**Ký tự tồi:** Giả sử  $P[j] \neq T[j + s]$ , khi đó ta gọi  $T[j + s]$  là ký tự tồi (*bad character*). Sử dụng ký tự tồi ta có thể tránh được việc đóng hàng không đúng.



**Hàm Last:**  $T[j + s]$  xuất hiện ở vị trí nào trong  $P$ ? Ta xác định hàm *last* như sau:  
Nếu  $c$  xuất hiện trong  $P$  thì đặt:

$\text{last}(c) = \text{chỉ số lớn nhất (bên phải nhất) của vị trí xuất hiện của } c \text{ trong } P,$   
ngược lại đặt:

$$\text{last}(c) = 0.$$

**Tăng vị trí dịch chuyển:** Giả sử ký tự tồi được đóng ở vị trí  $j$  của  $P$ .

### Tình huống 1

- Ký tự tồi có mặt trong  $P$  và  $\text{last}(c) < j$ .
- Khi đó có thể trượt đến:  $s \leftarrow s + (j - \text{last}(c))$ .

### Tình huống 2

- Ký tự tồi có mặt trong  $P$  và  $\text{last}(c) > j$ .
- Khi đó:  $s \leftarrow s + 1$ .

### Tình huống 3

- Ký tự tồi không có mặt trong  $P$  và vì thế  $\text{last}(c) = 0$ .
- Khi đó:  $s \leftarrow s + (j - \text{last}(c))$  hay  $s \leftarrow s + j$ .

### Boyer-Moore Algorithm

$s \leftarrow 0$

**while**  $s \leq n - m$  **do**

$j \leftarrow m$

**while**  $j > 0$  **and**  $T[j + s] = P[j]$  **do**

$j \leftarrow j - 1$

**if**  $j = 0$  **then**

        print  $s$  "là vị trí khớp"

$s \leftarrow s + 1$

**else**

$k \leftarrow \text{last}(T[j + s])$

$s \leftarrow s + \max(j - k, 1)$

## Ví dụ

pattern

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| a | c | a | b | a | c |
|---|---|---|---|---|---|

text

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | a | b | a | c | b | d | c | a | a | c | a | a | c | a | b | a | c |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Tính last cho mỗi ký hiệu trong bảng ký hiệu  $\Sigma = \{a, b, c, d\}$ :

$$\text{last}(a) = 5, \text{last}(c) = 6, \text{last}(b) = 4, \text{last}(d) = 0.$$

### Thời gian tính

- Việc tính hàm last đòi hỏi thời gian  $O(m + |\Sigma|)$ .
- Tình huống tồi nhất không khác gì thuật toán trực tiếp, nghĩa là đòi hỏi thời gian  $O(nm + |\Sigma|)$ .
- Ví dụ tình huống tồi nhất xảy ra khi:
  - + Pattern:  $ba^{m-1}$
  - + Text:  $a^n$
- Thuật toán làm việc kém hiệu quả đối với bảng  $\Sigma$  nhỏ.

### 6.4.4. Thuật toán Rabin–Karp

**Ý tưởng:** Coi mẫu  $P[0..m-1]$  như là khóa và chuyển đổi nó thành số nguyên  $p$ .

Tương tự như vậy ta cũng chuyển đổi các xâu con của  $T[]$  thành các số nguyên:

- For  $s = 0, 1, \dots, n - m$ , chuyển  $T[s \dots s + m - 1]$  thành số nguyên tương đương  $t_s$ . Khi đó: Mẫu xuất hiện ở vị trí  $s$  khi và chỉ khi  $p = t_s$ .

Nếu ta có thể tính  $p$  và  $t_s$  nhanh chóng, thì bài toán tìm kiếm xâu mẫu quy đắn về bài toán so sánh  $p$  với  $n - m + 1$  số nguyên.

Ta cần tính  $p$  như thế nào? Giả sử  $\Sigma = \{0, 1\}$ . Ta có:

$$p = P[0] + 2^{m-1}P[1] + \dots + 2P[m-2] + P[m-1].$$

Sử dụng sơ đồ Horner:

$$p = P[m-1] + 2*(P[m-2] + 2*(P[m-3] + \dots + 2*(P[1] + 2*P[0]))\dots),$$

ta có thể cài đặt trên C đoạn chương trình tính  $p$  như sau:

```
p = 0;
for (i=0; i<m; i++)
 p = 2*p + P[i];
```

Việc này đòi hỏi thời gian  $O(m)$ , nếu giả thiết các phép toán số học được thực hiện với thời gian  $O(1)$ .

Tương tự, tính  $(n - m + 1)$  số nguyên  $t_s$  từ xâu văn bản:

```
for (s = 0; s <= n-m; s++) {
 t[s] = 0;
 for (i = 0; i < m; i++)
 t[s] = 2*t[s] + T[s+i];
}
```

Việc này đòi hỏi thời gian  $O((n - m + 1) m)$ , với giả thiết các phép toán số học được thực hiện với thời gian  $O(1)$ . Công đoạn này là công đoạn tốn kém.

Để ý rằng có thể sử dụng  $t[s - 1]$  để tính  $t[s]$ :

$$t[s - 1] = 2^{m-1} T[s - 1] + 2^{m-2} T[s] + \dots + 2 T[s + m - 3] + T[s + m - 2],$$
$$t[s] = 2^{m-1} T[s] + 2^{m-2} T[s - 2] + \dots + 2 T[s + m - 2] + T[s + m - 1],$$

nên có thể thực hiện việc tính các số  $t_s$  hiệu quả hơn như sau:

```
t[0] = 0;
offset = 1;
for (i = 0; i < m; i++)
 offset = 2*offset;
for (i = 0; i < m; i++)
 t[0] = 2*t[0] + T[i];
for (s = 1; s <= n-m; s++)
 t[s] = 2*(t[s-1] - offset*T[s-1]) + T[s+m-1];
```

Việc này đòi hỏi thời gian  $O(n + m)$ , với giả thiết các phép toán số học được thực hiện với thời gian  $O(1)$ .

Khó khăn này sinh khi thực hiện thuật toán là nếu  $m$  lớn thì các phép toán số học cần thực hiện tốn rất nhiều thời gian, chứ không phải là  $O(1)$  như đã giả thiết. Trong trường hợp  $m$  lớn, ta thường phải cài đặt phép tính số học với số nguyên lớn.

Để giải quyết khó khăn này có thể sử dụng tính toán theo modulo. Giả sử  $q$  là số nguyên tố sao cho  $2q$  có thể cắt giữ trong một từ máy. Điều này đảm bảo tất cả các tính toán đều được thực hiện nhờ sử dụng tính toán số học với độ chính xác đơn.

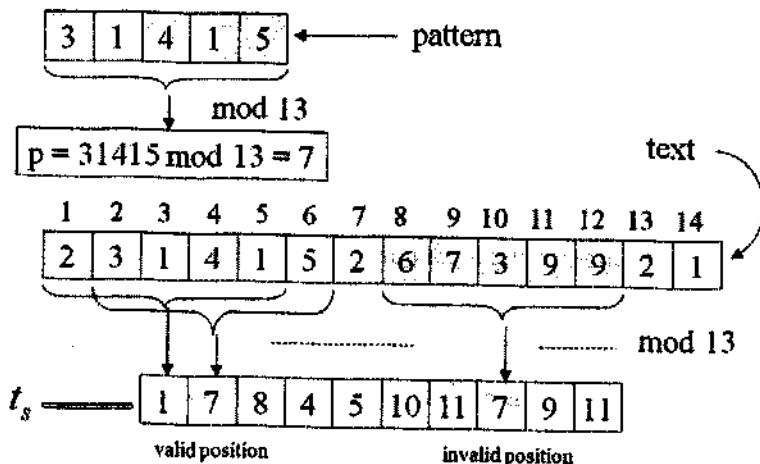
Tính  $p$  và  $t_s$

```
p = 0;
for (i=0; i<m; i++)
 p = (2*p + P[i]) % q;
t[0] = 0;
offset = 1;
for (i = 0; i < m; i++)
 offset = 2*offset % q;
for (i = 0; i < m; i++)
 t[0] = (2*t[0] + T[i]) % q;
for (s = 1; s <= n-m; s++)
 t[s] = (2*(t[s-1] - offset*T[s-1]) + T[s+m-1]) % q;
```

### Chú ý

- Nếu sử dụng tính toán theo modulo, khi  $p = t_s$  với  $s$  nào đó, chúng ta không còn chắc chắn được rằng  $P[0 \dots m-1]$  là bằng  $T[s \dots s+m-1]$ .
- Vì thế, sau khi kiểm tra được  $p = t_s$ , ta cần tiếp tục so sánh  $P[0 \dots m-1]$  với  $T[s \dots s+m-1]$  để có thể khẳng định  $s$  đúng là vị trí khớp.
- Thời gian trong tình huống tồi nhất của thuật toán vẫn là  $O(nm)$ . Đó là tình huống khi  $P = a^m$  còn  $T = a^n$ . Tuy nhiên trong thực tế ứng dụng, ta loại được rất nhiều phép so sánh xâu mẫu.

### Ví dụ



## 6.4.5. Thuật toán Knuth–Morris–Pratt (KMP)



Donald Knuth



James Morris



Vaughan Pratt

Ý tưởng chính là sử dụng hàm hỗ trợ (*prefix function*).

Thuật toán có độ phức tạp là  $O(n+m)$ .

### 6.4.5.1. Hàm tiền tố (prefix function)

#### Prefix và Suffix

Xâu  $W$  được gọi là **prefix** của xâu  $X$  nếu  $X = WY$  với một xâu  $Y$  nào đó, ký hiệu là  $W \subset X$ .

**Ví dụ:**  $W = ab$  là prefix của  $X = abefac$ , trong đó  $Y = efac$ .

Xâu rỗng, ký hiệu là  $\epsilon$ , là prefix của mọi xâu.

Xâu  $W$  được gọi là **suffix** của xâu  $X$  nếu  $X = YW$  với một xâu  $Y$  nào đó, ký hiệu là  $W \supset X$ .

**Ví dụ:**  $W = cdaa$  là suffix của  $X = acbecdaa$ , trong đó  $Y = acbe$ .

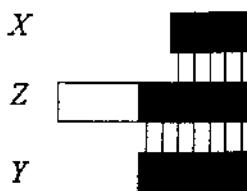
Xâu rỗng  $\epsilon$  là suffix của mọi xâu.

#### Overlapping Suffix

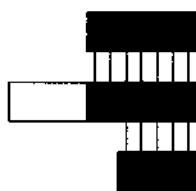
**Bố đề:** Giả sử  $X \supset Z$  và  $Y \supset Z$ . Khi đó:

- nếu  $|X| \leq |Y|$ , thì  $X \supset Y$ ;
- nếu  $|X| \geq |Y|$ , thì  $Y \supset X$ ;
- nếu  $|X| = |Y|$ , thì  $X = Y$ .

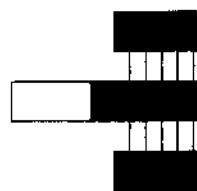
**Chứng minh:** Hiển nhiên. Hình vẽ dưới đây minh họa cho bố đề.



a)



b)



c)

## Dịch chuyển tối thiểu

Vấn đề đặt ra là: Biết rằng prefix  $P[1..q]$  của xâu mẫu khớp với đoạn  $T[(s+1) \dots (s+q)]$ , tìm giá trị nhỏ nhất  $s' > s$  sao cho:

$$P[1..k] = T[(s'+1) \dots (s'+k)], \text{ trong đó } s'+k = s+q.$$

Khi đó, tại vị trí  $s'$ , không cần thiết so sánh  $k$  ký tự đầu của  $P$  với các ký tự tương ứng của  $T$ , vì ta biết chắc rằng chúng là khớp nhau.

**Prefix function:**  $\pi[q]$  là độ dài của prefix dài nhất của  $P[1..m]$ , đồng thời là suffix thực sự của  $P[1..q]$ , nghĩa là:

$$\pi[q] = \max \{ k : k < q \text{ và } P[1..k] \text{ là suffix của } P[1..q] \}.$$

**Ví dụ:**

Xét xâu mẫu  $P = ababababca$ . Bảng dưới đây cho giá trị của hàm tiền tố:

|          |   |   |   |   |   |   |   |   |   |    |
|----------|---|---|---|---|---|---|---|---|---|----|
| $i$      | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $P[i]$   | a | b | a | b | a | b | a | b | c | a  |
| $\pi[i]$ | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 0 | 1  |

Tính giá trị của hàm prefix

**Compute-Prefix-Function( $P$ )**

```
1 $m \leftarrow \text{length}[P]$
2 $\pi[1] \leftarrow 0$
3 $k \leftarrow 0$
4 for $q = 2$ to m
5 do while $k > 0$ and $P[k+1] \neq P[q]$
6 do $k \leftarrow \pi[k]$
7 if $P[k+1] = P[q]$
8 then $k \leftarrow k+1$
9 $\pi[q] \leftarrow k$
10 return π
```

Thời gian tính:  $\Theta(m)$ .

### 6.4.5.2. KMP Algorithm

**KMP-Matcher( $T, P$ )** //  $n = |T|$  and  $m = |P|$

$\pi \leftarrow \text{Compute-Prefix-Function}(P)$

```

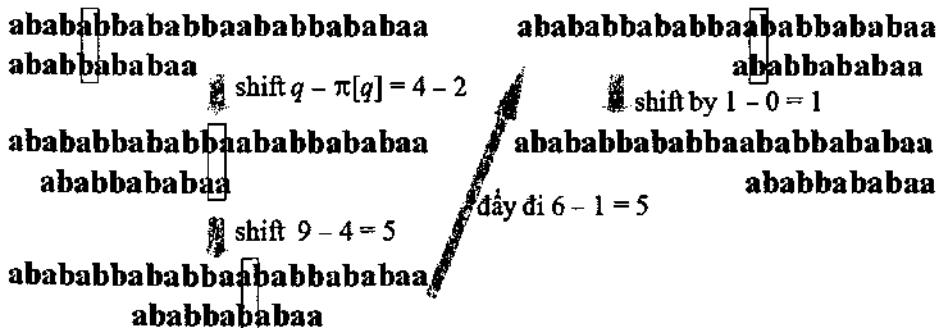
 $q \leftarrow 0$
for $i \leftarrow 1$ to n
 do while $q > 0$ and $P[q+1] \neq T[i]$
 do $q \leftarrow \pi[q]$
 if $P[q+1] = T[i]$
 then $q \leftarrow q+1$
 if $q = m$
 then print "Pattern xuất hiện ở vị trí" $i - m$
 $q \leftarrow \pi[q]$

```

Thời gian tính  $\Theta(m+n)$ .

**Ví dụ:** Thực hiện thuật toán KMP.

|           |                         |
|-----------|-------------------------|
| $i$       | 1 2 3 4 5 6 7 8 9 10 11 |
| $P[1..i]$ | a b a b b a b a b a a   |
| $\pi[i]$  | 0 0 1 2 0 1 2 3 4 3 1   |



## 6.5. BẢNG BĂM (MAPPING AND HASHING)

### 6.5.1. Đặt vấn đề

Cho bảng  $T$  và bản ghi  $x$ , với khóa và dữ liệu đi kèm, ta cần hỗ trợ các thao tác sau:

- Insert ( $T, x$ );
- Delete ( $T, x$ );
- Search( $T, x$ ).

Ta muốn thực hiện các thao tác này một cách nhanh chóng mà không phải thực hiện việc sắp xếp các bản ghi. Bảng băm là cách tiếp cận giải quyết vấn đề đặt ra.

Trong mục này ta sẽ chỉ xét khóa là các số nguyên dương (có thể rất lớn).

## Ứng dụng

- Xây dựng chương trình dịch của ngôn ngữ lập trình: Ta cần thiết lập bảng ký hiệu trong đó khóa của các phần tử là dãy ký tự tương ứng với các từ định danh (identifiers) trong ngôn ngữ lập trình.
- Bảng băm là cấu trúc dữ liệu hiệu quả để cài đặt các từ điển (dictionaries).
- Mặc dù trong tình huống xấu nhất, việc tìm kiếm đòi hỏi thời gian  $O(n)$  giống như danh sách mòc nối, nhưng trên thực tế bảng băm làm việc hiệu quả hơn nhiều. Với một số giả thiết khá hợp lý, việc tìm kiếm phần tử trong bảng băm đòi hỏi thời gian  $O(1)$ .
- Bảng băm có thể xem như sự mở rộng của mảng thông thường. Việc địa chỉ hóa trực tiếp trong mảng cho phép truy nhập đến phần tử bất kỳ trong thời gian  $O(1)$ .

### 6.5.2. Địa chỉ trực tiếp – Direct Addressing

Giả thiết rằng:

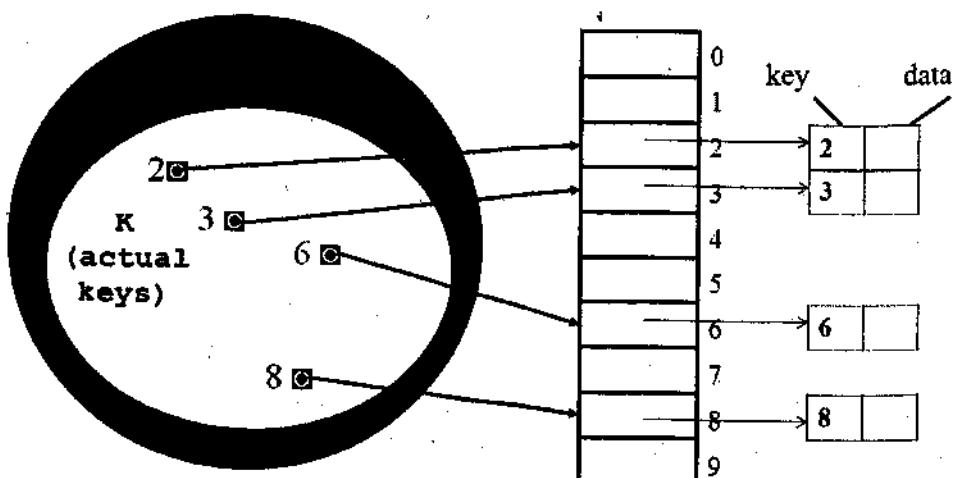
- Các khóa là các số trong khoảng từ 0 đến  $m - 1$ ;
- Các khóa là khác nhau từng đôi.

Ý tưởng: Thiết lập mảng  $T[0..m - 1]$  trong đó:

- $T[i] = x$ , nếu  $x \in T$  và  $\text{key}[x] = i$ ;
- $T[i] = \text{NULL}$  nếu ngược lại.

$T$  được gọi là bảng địa chỉ trực tiếp (*direct-address table*), các phần tử trong bảng  $T$  sẽ được gọi là các ô.

Ví dụ: Tạo bảng địa chỉ trực tiếp  $T$ . Mỗi khóa trong tập  $U = \{0, 1, \dots, 9\}$  tương ứng với một chi số trong bảng. Tập  $K = \{2, 3, 5, 8\}$  gồm các khóa thực có xác định các ô trong bảng chứa con trỏ đến các phần tử. Các ô khác (được tô màu) chứa con trỏ NULL.



Các phép toán được cài đặt một cách trực tiếp:

– DIRECT-ADDRESS-SEARCH( $T, k$ )

return  $T[k]$

– DIRECT-ADDRESS-INSERT( $T, x$ )

$T[key[x]] = x$

– DIRECT-ADDRESS-DELETE( $T, x$ )

$T[key[x]] = \text{NULL}$

Thời gian thực hiện mỗi phép toán đều là  $O(1)$ .

#### Hạn chế của phương pháp địa chỉ trực tiếp

Phương pháp địa chỉ trực tiếp làm việc tốt nếu như biên độ  $m$  của các khóa là tương đối nhỏ. Nếu các khóa là các số nguyên 32-bit thì sao?

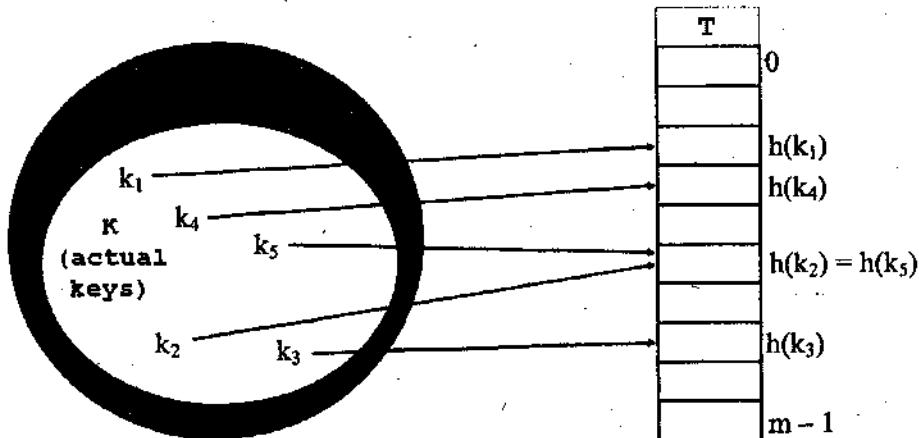
– Vấn đề 1: bảng địa chỉ trực tiếp sẽ phải có  $2^{32}$  (hơn 4 tỷ) phần tử.

– Vấn đề 2: ngay cả khi bộ nhớ không phải là vấn đề, thì thời gian khởi tạo các phần tử là NULL cũng rất tốn kém.

Cách giải quyết: Ánh xạ khóa vào khoảng biến đổi nhỏ hơn  $0 \dots m - 1$ . Ánh xạ này được gọi là hàm băm (*hash function*).

#### 6.5.3. Hàm băm (Hash Functions)

Khi sử dụng hàm băm, vấn đề này sinh lại là xung đột (*collision*), khi nhiều khóa được đặt tương ứng với cùng một ô trong bảng địa chỉ  $T$ .



**Giải quyết xung đột:** Ta cần giải quyết xung đột như thế nào? Xét hai cách tiếp cận chính để giải quyết xung đột:

– Cách giải quyết 1: Phương pháp địa chỉ mở (open addressing);

– Cách giải quyết 2: Tạo chuỗi (chaining).

#### 6.5.3.1. Địa chỉ mở (Open Addressing)

Trong phương pháp địa chỉ mở, tất cả các phần tử đều được cất giữ vào bảng. Do đó mỗi ô của bảng hoặc là chứa khóa hoặc là NULL. Ý tưởng chính của phương pháp này là:

– Để thực hiện bổ sung, nếu ô tìm được là bận, ta sẽ tiến hành khảo sát lần lượt (hay còn gọi là dò thử) các ô của bảng cho đến khi tìm được ô rỗng để nạp khóa vào. Thay vì tìm tuần tự theo thứ tự  $0, 1, \dots, m - 1$  (sẽ đòi hỏi thời gian  $\Theta(n)$ ), dãy các vị trí thử sẽ phụ thuộc vào khóa được bổ sung. Để xác định các ô dò thử, ta sẽ mở rộng định nghĩa hàm băm.

– Khi tìm kiếm (search), ta sẽ tìm dọc theo dãy các phép dò thử giống như dãy dò thử khi thực hiện chèn phần tử vào bảng.

- + Nếu tìm được phần tử với khóa đã cho thì trả lại nó;
- + Nếu tìm được con trỏ NULL, thì phần tử cần tìm không có trong bảng.

Ta mở rộng định nghĩa hàm băm như sau:

$$h: U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}.$$

Trong phương pháp địa chỉ mở ta đòi hỏi, với mỗi khóa  $k$ , dãy dò thử:

$$\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$$

phải là một hoán vị của  $\langle 0, 1, \dots, m-1 \rangle$ , do đó mỗi vị trí trong bảng sẽ được xét như là một ô để chứa khóa mới khi ta tiến hành bổ sung vào bảng.

Việc bổ sung khóa  $k$  vào bảng được mô tả trong đoạn giả mã sau đây:

##### HASH-INSERT( $T, k$ )

```
1 i ← 0
2 repeat j ← h(k,i)
3 if T[j] = NIL
4 then T[j] ← k
5 return j
6 else i ← i + 1
7 until i = m
8 error "hash table overflow"
```

Thuật toán tìm kiếm khóa  $k$  trong bảng được mô tả trong đoạn giả mã sau:

##### HASH-SEARCH( $T, k$ )

```
1 i ← 0
2 repeat j ← h(k,i)
3 if T[j]=j
4 then return j
```

```

5 $i \leftarrow i + 1$
6 until $T[j] = \text{NIL}$ or $i = m$
7 return NIL

```

Việc loại bỏ gấp nhiều khó khăn hơn. Thông thường ta sẽ đánh dấu loại bỏ chứ không thực hiện loại bỏ thực sự.

Có ba kỹ thuật dò thử thường được sử dụng:

- Dò tuyến tính (Linear Probing);
- Dò toàn phương (Quadratic Probing);
- Băm kép (Double Hashing).

Các kỹ thuật này đều đảm bảo  $\langle h(k, 1), h(k, 2), \dots, h(k, m) \rangle$  là hoán vị của  $\langle 0, 1, \dots, m - 1 \rangle$  đối với mỗi khóa  $k$ .

#### Dò tuyến tính (Linear probing)

Cho hàm băm  $h': U \rightarrow \{0, 1, \dots, m - 1\}$ , phương pháp dò tuyến tính sử dụng hàm băm mở rộng sau:

$$h(k, i) = (h'(k) + i) \bmod m,$$

với  $i = 0, 1, \dots, m - 1$ . Cho khóa  $k$ , ô đầu tiên được thử là  $T[h'(k)]$ . Tiếp đến ta sẽ thử ô  $T[h'(k) + 1]$ ,  $T[h'(k) + 2], \dots$  cho đến ô  $T[m - 1]$ .

#### Dò toàn phương (Quadratic probing)

Sử dụng hàm băm mở rộng:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

trong đó: (giống như dò tuyến tính)  $h'$  là hàm băm ban đầu,  $c_1$  và  $c_2 \neq 0$  là các hằng số,  $i = 0, 1, \dots, m - 1$ .

#### Hàm băm kép (Double hashing)

Hàm băm kép là phương pháp tốt nhất có thể sử dụng trong phương pháp địa chỉ mở vì dãy phép thử tạo được bởi nó có tính chất giống như một hoán vị ngẫu nhiên. Trong phương pháp hàm băm kép, ta sử dụng hàm băm mở rộng:

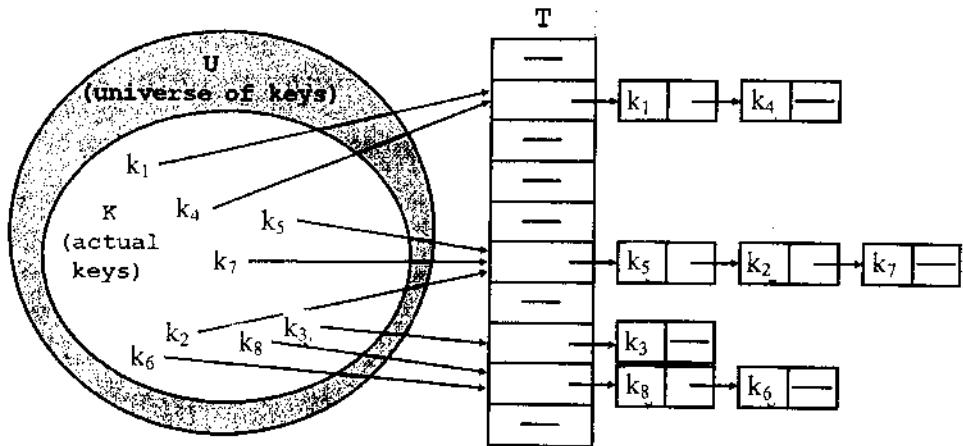
$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m,$$

trong đó:  $h_1$  và  $h_2$  là các hàm băm hỗ trợ.

Chú ý là phương pháp địa chỉ mở áp dụng tốt khi ta làm việc với tập cố định (chỉ có bổ sung mà không có chèn). Ví dụ: khi kiểm lỗi chính tả (spell checking). Lưu ý rằng bảng có kích thước không cần lớn hơn  $n$  quá nhiều.

#### 6.5.3.2. Tạo chuỗi (Chaining)

Theo phương pháp này, ta sẽ tạo danh sách mốc nối để chứa các phần tử được gắn với cùng một vị trí trong bảng.



### Chú ý:

- Ta cần thực hiện bổ sung phần tử như thế nào? (Như bổ sung vào danh sách mốc nối).
- Ta cần thực hiện loại bỏ phần tử như thế nào? Có cần sử dụng danh sách nối đôi để thực hiện xóa một cách hiệu quả hay không? Không, vì thông thường chuỗi có độ dài không lớn.
- Thực hiện tìm kiếm phần tử với khóa cho trước như thế nào?

### Phân tích phương pháp chuỗi

Giả sử rằng thực hiện điều kiện *simple uniform hashing*: Mỗi khóa trong bảng là đồng khả năng được gán với một ô bất kỳ. Cho  $n$  khóa và  $m$  ô trong bảng, ta định nghĩa nhân tử nạp:

(load factor)  $\alpha = n/m =$  số lượng khóa trung bình trên một ô.

Khi đó có thể chứng minh được rằng:

- Chi phí trung bình để phát hiện một khóa không có trong bảng là  $O(1 + \alpha)$ .
- Chi phí trung bình để phát hiện một khóa có trong bảng là  $O(1 + \alpha)$ .

Do đó chi phí tìm kiếm là  $O(1 + \alpha)$ .

Như vậy chi phí của thao tác tìm kiếm là  $O(1 + \alpha)$ . Nếu số lượng khóa  $n$  tỷ lệ với số lượng ô trong bảng thì  $\alpha$  có giá trị  $O(1)$ . Nói cách khác, ta có thể đảm bảo chi phí tìm kiếm mong đợi là hằng số nếu ta đảm bảo  $\alpha$  là hằng số.

### Chọn hàm băm

Rõ ràng việc chọn hàm băm tốt sẽ có ý nghĩa quyết định. Khi chọn hàm băm ta cần quan tâm đến hai vấn đề sau:

- Thời gian tính của hàm băm là bao nhiêu?
- Thời gian tìm kiếm sẽ như thế nào?

Từ đó, hai yêu cầu quan trọng đối với hàm băm là:

- Phải phân bổ đều các khóa vào các ô.
- Không phụ thuộc vào khuôn mẫu trong dữ liệu.

### Hash Functions – Phương pháp chia (The Division Method)

Ta xác định hàm băm theo công thức:

$$h(k) = k \bmod m,$$

nghĩa là: gắn  $k$  vào bảng có  $m$  ô nhờ sử dụng ô xác định bởi phần dư của phép chia  $k$  cho  $m$ .

Dễ thấy là nếu  $m$  là luỹ thừa của 2 (chẳng hạn  $2^p$ ), thì  $h(k)$  chính là  $p$  bit cuối của  $k$ . Còn nếu  $m$  là luỹ thừa 10 (chẳng hạn  $10^p$ ) thì  $h(k)$  chỉ phụ thuộc vào  $p$  chữ số cuối của  $k$ . Vì thế, thông thường người ta chọn kích thước bảng  $m$  là số nguyên tố không quá gần với luỹ thừa của 2 (hoặc 10).

### Hash Functions – Phương pháp nhân (The Multiplication Method)

Fương pháp nhân để xây dựng hàm băm được tiến hành theo hai bước. Đầu tiên ta nhân  $k$  với một hằng số  $A$ ,  $0 < A < 1$  và lấy phần thập phân của  $kA$ . Sau đó, ta nhân giá trị này với  $m$  rồi lấy phần nguyên của kết quả:

- Chọn hằng số  $A$ ,  $0 < A < 1$ .
- $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$  (phần thập phân của  $kA$ ).

Cách chọn các thông số:

- Chọn  $m = 2^p$ .
- Chọn  $A$  không quá gần với 0 hoặc 1. Knuth khuyên rằng:

Hãy chọn  $A = (\sqrt{5} - 1)/2$ .

## BÀI TẬP CHƯƠNG 6

- a) Vẽ cây nhị phân tìm kiếm thu được bởi việc, bắt đầu từ cây rỗng, bổ sung lần lượt các khóa từ dãy khóa sau đây:  
**40, 30, 65, 25, 35, 50, 76, 10, 28, 27, 33, 36, 34, 48, 60, 68, 80, 69.**  
b) Trình diễn thao tác loại bỏ nút với khóa 30 trên cây nhị phân tìm kiếm thu được.  
c) Trình diễn thao tác loại bỏ nút với khóa 65 trên cây nhị phân tìm kiếm thu được.
- a) Vẽ cây nhị phân tìm kiếm thu được bởi việc, bắt đầu từ cây rỗng, bổ sung lần lượt các khóa từ dãy khóa sau đây:  
**40, 30, 65, 25, 35, 50, 76, 10, 28, 33, 36, 34, 48, 60, 68, 69, 80.**

- b) Trình diễn thao tác loại bỏ nút với khóa 65 trên cây nhị phân tìm kiếm thu được ở câu a).
3. a) Trình diễn việc xây dựng cây nhị phân tìm kiếm thu được bởi việc, bắt đầu từ cây rỗng, bổ sung lần lượt các khóa từ dãy khóa sau đây:

**8, 4, 3, 6, 23, 7, 39, 28, 25, 29.**

- b) Trình diễn thao tác loại bỏ nút với khóa 28 trên cây nhị phân tìm kiếm thu được.
4. a) Vẽ cây nhị phân tìm kiếm thu được khi lần lượt chèn các phần tử của dãy số nguyên:

**3, 1, 4, 6, 9, 2, 8, 5, 7, 0**

- vào cây nhị phân tìm kiếm ban đầu là rỗng (không cần giải thích).
- b) Hãy vẽ cây nhị phân tìm kiếm thu được sau hai lần loại bỏ gốc của cây nhị phân tìm kiếm trong a) (không cần giải thích).
5. Xét cấu trúc dữ liệu trên C để mô tả cây nhị phân tìm kiếm sau đây:

```
struct TreeNode {
 float key;
 struct TreeNode* leftPtr;
 struct TreeNode* rightPtr;
};
```

```
typedef struct TreeNode BSTree;
```

- a) Hãy viết các hàm trên C sử dụng các cấu trúc dữ liệu trên để thực hiện các thao tác sau đây với cây nhị phân tìm kiếm:
- Tạo một nút mới:

```
BSTree* makeTreeNode (float value);
```

- Bổ sung một nút mới vào cây nhị phân tìm kiếm:

```
BSTree* insert (BSTree* nodePtr, float item);
```

- b) Vẽ cây nhị phân tìm kiếm đối với tập các khóa S = {3, 2, 5, 4, 7, 6, 1} thu được nhờ thực hiện bổ sung lần lượt các khóa theo thứ tự đã cho vào cây nhị phân được khởi tạo ban đầu là rỗng.
6. a) Tính giá trị của hàm tiền tố trong thuật toán Knuth–Morris–Pratt (prefix function)  $\pi$  đối với xâu mẫu P = "abccbabbcaabcbabccbabcba" với giả thiết bảng chữ cái  $\Sigma = \{a, b, c\}$ .

Điền các giá trị tính được vào bảng sau:

| i        | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|----------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| P[i]     | a | b | c | c | b | a | b | b | c | a  | a  | b  | c  | b  | a  | b  | c  | c  | b  | a  | b  | c  | b  | a  |
| $\pi[i]$ |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

- b) Tính giá trị của hàm tiền tố trong thuật toán Knuth–Morris–Pratt (prefix function)  $\pi$  đối với xâu mẫu  $P = "aaaaaaaaabbbbbbbbaaaaaaaaa"$  với giả thiết bảng chữ cái  $\Sigma = \{a, b, c\}$ .

Điền các giá trị tính được vào bảng sau:

| i        | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|----------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| P[i]     | a | a | a | a | a | a | a | b | b | b  | b  | b  | b  | b  | b  | a  | a  | a  | a  | a  | a  | a  | a  |    |
| $\pi[i]$ |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

7. Cho bảng  $T$  kích thước 11 ô và tập khóa  $S = \{5, 40, 18, 22, 16, 30, 27\}$  cần được nạp vào bảng sử dụng hàm băm  $h(k) = k \% 11$ . Hãy vẽ bảng  $T$  sau khi tất cả các giá trị khóa trong tập  $S$  được cắt giữ vào bảng  $T$ , sử dụng kỹ thuật tạo chuỗi để xử lý xung đột.
8. Cho bảng  $T$  kích thước 11 ô và tập khóa  $S = \{5, 40, 18, 22, 16, 30, 27\}$  cần được nạp vào bảng sử dụng hàm băm  $h(k) = k \% 11$ . Hãy vẽ bảng  $T$  sau khi tất cả các giá trị khóa trong tập  $S$  được cắt giữ vào bảng  $T$ , sử dụng kỹ thuật dò tuyển tính để xử lý xung đột.
9. Xét cấu trúc dữ liệu biểu diễn cây nhị phân:

```
struct TreeNode{
 int info;
 TreeNode * left;
 TreeNode * right;
};
```

- a) Hãy viết hàm trên C:

```
int MaxOdd(TreeNode *root)
```

nhận đầu vào root là con trỏ đến gốc của cây nhị phân tìm kiếm, trả lại giá trị **info** lẻ lớn nhất trong các nút của cây nhị phân đã cho.

- b) Gọi  $T(n)$  là thời gian tính của hàm trong câu 1a), hãy đưa ra đánh giá cho  $T(n)$ , trong đó  $n$  là số lượng nút của cây nhị phân đầu vào.

10. Xét cấu trúc dữ liệu mô tả cây nhị phân:

```
struct TreeNode {
 int info;
 TreeNode * left;
 TreeNode * right; };
```

Viết hàm:

```
bool IsBST(TreeNode * root);
```

nhận đầu vào là con trỏ **root** đến gốc của cây, trả lại giá trị **true** khi và chỉ khi cây được trả bởi **root** là cây nhị phân cân bằng.

11. Xét cấu trúc dữ liệu biểu diễn cây nhị phân tìm kiếm sau đây:

```
typedef struct _bstreenode {
 int k; /* khóa */
 struct _bstreenode
 left, / cây con trái */
 right; / cây con phải */
} bstreenode, *bstree;
```

a) Hàm sau đây thực hiện công việc gì đối với cây nhị phân tìm kiếm?

```
int *search_bstree (bstree *t, int k) {
 while (t) {
 if (t->k == k) return &(t->k); /* tìm thấy */
 if (k <= t->k)
 t = t->left; /* di chuyển theo con trái */
 else
 t = t->right; /* di chuyển theo con phải */
 }
 return NULL;
}
```

Giải thích kết quả thực hiện của hàm.

Hãy viết lại hàm trên dưới dạng hàm đệ quy.

b) Hàm sau đây thực hiện công việc gì đối với cây nhị phân tìm kiếm?

```
void insert_bstree (bstree *t, int k) {
 bstree p;
 p = (bstree) malloc (sizeof (bstreenode));
 p->left = NULL;
 p->right = NULL;
 p->k = k;
 while (*t) {
 if (k <= (*t)->k)
 t = &(*t)->left;
 else
 t = &(*t)->right;
 }
 *t = p;
}
```

Giải thích kết quả thực hiện của hàm.

Hãy viết lại hàm trên dưới dạng hàm đệ quy.

12. Trong bài này giả sử cây nhị phân được mô tả bởi cấu trúc dữ liệu sau:

```
struct TreeNode
{
 string info;
 TreeNode * left;
 TreeNode * right;
 TreeNode * parent;
};
```

- a) Viết hàm trả lại nút có trường **info** nhỏ nhất của cây với gốc được trả bởi **root**:

```
TreeNode* Minimum(TreeNode * root)
```

Đưa ra đánh giá thời gian tính trong ký hiệu  $O$  lớn của hàm.

- b) Viết hàm trả lại nút có trường **info** lớn nhất của cây với gốc được trả bởi **root**:

```
TreeNode* Maximum(TreeNode * root);
```

Đưa ra đánh giá thời gian tính trong ký hiệu  $O$  lớn của hàm.

c) Viết hàm xác định nút kế cận sau của một nút được trả bởi **tNode** trên cây nhị phân tìm kiếm.

```
TreeNode* Successor(TreeNode * t);
```

Đưa ra đánh giá thời gian tính trong ký hiệu  $O$  lớn của hàm.

Trong các bài tập tiếp theo ta giả thiết rằng thông tin cát giữ tại nút của cây là số nguyên thay cho xâu, nghĩa là ta sẽ sử dụng cấu trúc dữ liệu:

```
struct TreeNode
{
 int info;
 TreeNode * left;
 TreeNode * right;
 TreeNode * parent;
};
```

d) Viết hàm không đệ quy **isBST** và hàm đệ quy **isBSTrec** nhận đầu vào **root** là con trỏ đến gốc của cây nhị phân, trả lại giá trị **true** khi và chỉ khi cây nhị phân đã cho là cây nhị phân tìm kiếm.

```
bool IsBST(TreeNode* root);
```

```
bool IsBSTrec(TreeNode* root);
```

e) Viết hàm *FindKthInOrder*

```
TreeNode * FindKthInOrder(TreeNode * root, int k);
```

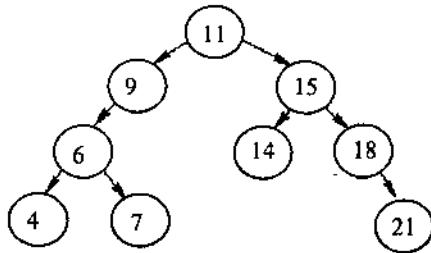
nhận đầu vào là con trỏ **root** đến gốc của cây nhị phân tìm kiếm và số nguyên dương **k**, trả lại con trỏ đến nút có trường **info** nhỏ thứ **k** trong cây nhị phân tìm kiếm, trả lại **NULL** nếu cây có ít hơn **k** nút. Như vậy nếu **k = 1**, hàm cần trả lại con trỏ đến nút có trường **info** là nhỏ nhất, nếu **k = 2** hàm trả lại con trỏ đến nút có trường **info** là nhỏ nhì,...

Ví dụ, đối với cây nhị phân trong hình vẽ dưới đây:

*FindKthInOrder(t, 4)* trả lại nút với trường **info** là 9;

*FindKthInOrder(t, 8)* trả lại nút với trường **info** là 18;

*FindKthInOrder(t, 12)* trả lại **NULL** bởi vì chỉ có 9 nút trên cây.



Có thể sử dụng hàm *count* đã trình bày trong bài giảng để đếm số nút trong cây:

```

int count(TreeNode * root)
{
 if (root == NULL) return 0;
 return 1 + count(root->left) + count(root->right);
}

```

Đưa ra đánh giá thời gian tính trong tình huống tồi nhất của hàm.

Đánh giá thời gian tính của hàm sẽ như thế nào nếu giả sử rằng cây là cân bằng?

# Chương 7

## ĐỒ THỊ VÀ CÁC THUẬT TOÁN ĐỒ THỊ

### 7.1. ĐỒ THỊ

Đồ thị  $G$  là cấu trúc rời rạc bao gồm hai tập:

- Tập đỉnh  $V(G)$  là tập hữu hạn khác rỗng;
- Tập cạnh  $E(G)$  là tập hữu hạn có thể là tập rỗng các cặp  $(u, v)$ ,  $u, v \in V$ .

Ta sẽ ký hiệu là  $G = (V, E)$ .

Phụ thuộc vào kiểu của cạnh nối và số lượng cạnh nối giữa hai đỉnh mà ta phân biệt các loại đồ thị khác nhau.

#### 7.1.1. Các loại đồ thị

**Định nghĩa:** Đơn (đa) đồ thị vô hướng  $G = (V, E)$  là cặp gồm:

- Tập đỉnh  $V$  là tập hữu hạn phần tử, các phần tử gọi là các **đỉnh**;
- Tập cạnh  $E$  là tập (hợp) các bộ không có thứ tự dạng  $(u, v)$ ,  $u, v \in V$ ,  $u \neq v$ . Các phần tử của  $E$  được gọi là các **cạnh**.

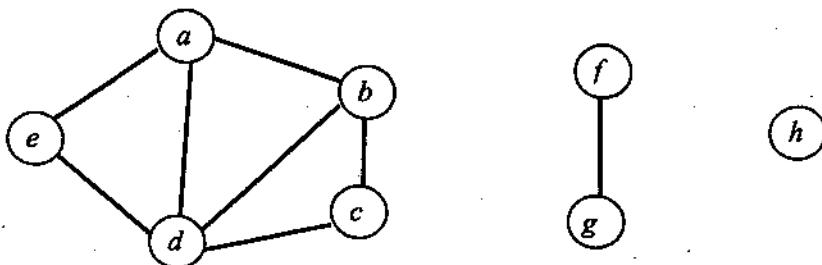
**Ví dụ:**

Đơn đồ thị vô hướng  $G_1 = (V_1, E_1)$ ,

trong đó:

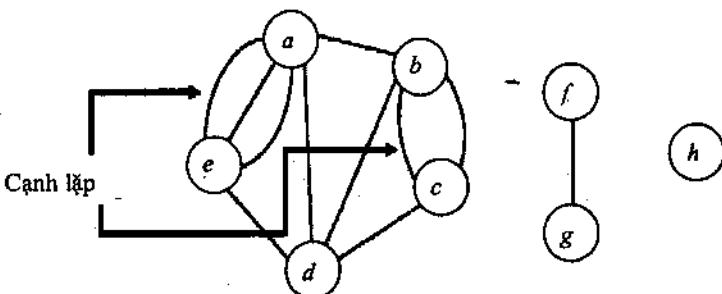
$$V_1 = \{a, b, c, d, e, f, g, h\},$$

$$E_1 = \{(a, b), (b, c), (c, d), (a, d), (d, e), (a, e), (d, b), (f, g)\}.$$



Đơn đồ thị vô hướng  $G$ ,

Đa đồ thị vô hướng  $G_2 = (V_2, E_2)$ ,  
 trong đó:  $V_2 = \{a, b, c, d, e, f, g, h\}$ ,  
 $E_2 = \{(a,b), (b,c), (c,d), (d,e), (e,a), (a,c), (a,d), (d,b), (f,g)\}$ .

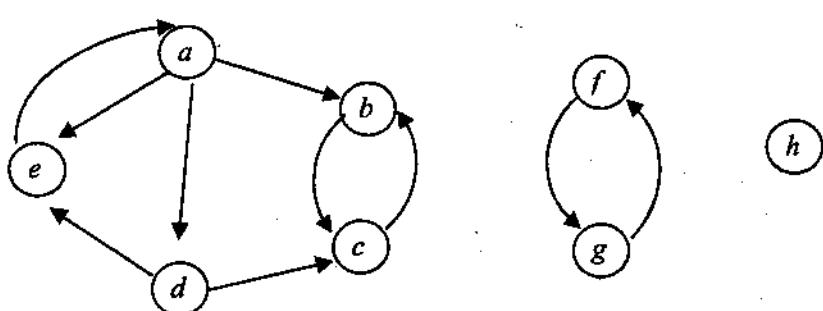


**Đa đồ thị vô hướng  $G_2$**

**Định nghĩa:** Đơn (đa) đồ thị có hướng  $G = (V, E)$  là cặp gồm:  
 – Tập đỉnh  $V$  là tập hữu hạn phần tử, các phần tử gọi là các **định**;  
 – Tập cung  $E$  là tập (hợp) các bộ có thứ tự dạng  $(u, v)$ ,  $u, v \in V$ ,  $u \neq v$ . Các phần tử của  $E$  được gọi là các cung hoặc cạnh có hướng.

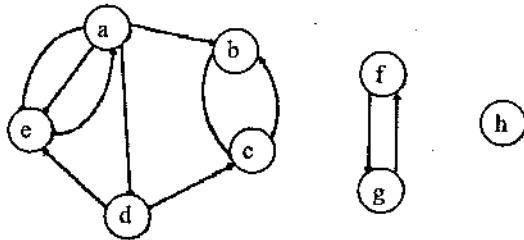
**Ví dụ:**

Đơn đồ thị có hướng  $G_3 = (V_3, E_3)$ ,  
 trong đó:  $V_3 = \{a, b, c, d, e, f, g, h\}$ ;  
 $E_3 = \{(a,b), (b,c), (c,b), (d,c), (a,d), (a,e), (d,e), (e,a), (f,g), (g,f)\}$



**Đơn đồ thị có hướng  $G_3$**

Đa đồ thị có hướng  $G_4 = (V_4, E_4)$ ,  
 trong đó:  $V_4 = \{a, b, c, d, e, f, g, h\}$ ,  
 $E_4 = \{(a,b), (b,c), (c,b), (d,c), (a,d), (a,e), (a,e), (d,e), (e,a), (f,g), (g,f)\}$ .



**Đa đồ thị có hướng  $G_4$**

Bảng sau đây nêu hai đặc trưng để phân biệt các loại đồ thị.

| Loại                | Kiểu cạnh | Có cạnh lặp? |
|---------------------|-----------|--------------|
| Đơn đồ thị vô hướng | Vô hướng  | Không        |
| Đa đồ thị vô hướng  | Vô hướng  | Có           |
| Đơn đồ thị có hướng | Có hướng  | Không        |
| Đa đồ thị có hướng  | Có hướng  | Có           |

Chúng ta cần các thuật ngữ liên quan đến mối quan hệ giữa các đỉnh và các cạnh của đồ thị sau: *kề nhau, nối, đầu mút, bậc, bắt đầu, kết thúc, bán bậc vào, bán bậc ra, ...*

Cho  $G$  là đồ thị vô hướng và giả sử  $e = (u,v)$  là cạnh của  $G$ . Khi đó ta nói:

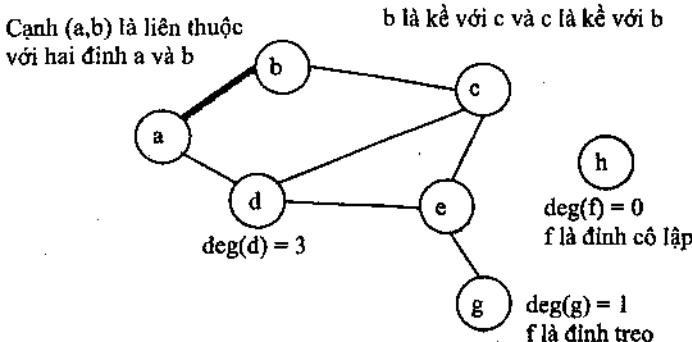
- $u, v$  là *kề nhau/lân cận/nối* với nhau (adjacent/neighbors/connected).
- Cạnh  $e$  là *liên thuộc* với hai đỉnh  $u$  và  $v$ .
- Cạnh  $e$  *nối (connect)*  $u$  và  $v$ .
- Các đỉnh  $u$  và  $v$  là các *đầu mút (endpoints)* của cạnh  $e$ .

Cho  $G$  là đồ thị có hướng và giả sử  $e = (u,v)$  là cạnh của  $G$ . Ta nói:

- $u$  và  $v$  là *kề nhau*,  $u$  là *kề tới*  $v$ ,  $v$  là *kề từ*  $u$ .
- $e$  *đi ra khỏi*  $u$ ,  $e$  *đi vào*  $v$ .
- $e$  *nối*  $u$  với  $v$ ,  $e$  *đi từ*  $u$  *tới*  $v$ .
- Đỉnh *đầu (initial vertex)* của  $e$  là  $u$ .
- Đỉnh *cuối (terminal vertex)* của  $e$  là  $v$ .

**Định nghĩa:** Giả sử  $G$  là đồ thị vô hướng,  $v \in V$  là một đỉnh nào đó. *Bậc* của đỉnh  $v$ , ký hiệu là  $\deg(v)$ , là số cạnh kề với nó. Đỉnh bậc 0 được gọi là *đỉnh cô lập (isolated)*. Đỉnh bậc 1 được gọi là *đỉnh treo (pendant)*.

### Ví dụ:



Ta có kết quả nổi tiếng sau đây về bậc của các đỉnh.

**Định lý về các cái bắt tay:** Giả sử  $G$  là đồ thị vô hướng (đơn hoặc đa) với tập đỉnh  $V$  và tập cạnh  $E$ . Khi đó:

$$\sum_{v \in V} \deg(v) = 2|E|.$$

**Chứng minh:** Trong tổng ở vế trái, mỗi cạnh  $e = (u,v) \in E$  được tính hai lần: trong  $\deg(u)$  và  $\deg(v)$ .

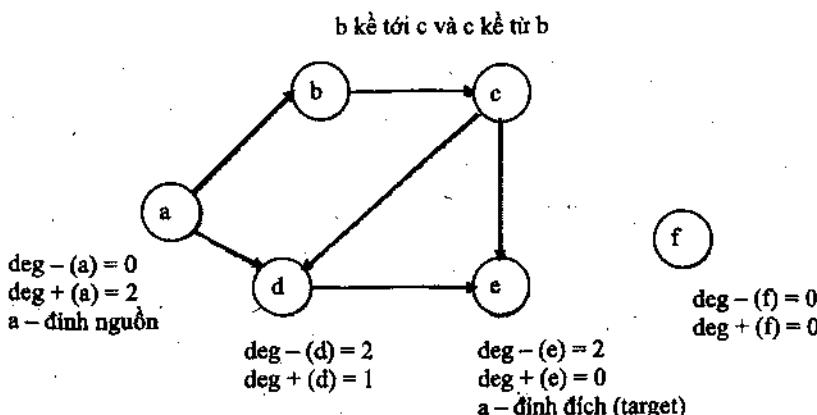
Từ định lý ta dễ dàng chứng minh hệ quả sau.

**Hệ quả:** Trong một đồ thị vô hướng bất kỳ, số lượng đỉnh bậc lẻ (đỉnh có bậc là số lẻ) bao giờ cũng là số chẵn.

**Định nghĩa:** Cho  $G$  là đồ thị có hướng,  $v$  là đỉnh của  $G$ .

- *Bán bậc vào (in-degree)* của  $v$ ,  $\deg^-(v)$ , là số cạnh đi vào  $v$ .
- *Bán bậc ra (out-degree)* của  $v$ ,  $\deg^+(v)$ , là số cạnh đi ra khỏi  $v$ .
- *Bậc của  $v$* ,  $\deg(v) := \deg^-(v) + \deg^+(v)$ , là tổng của bán bậc vào và bán bậc ra của  $v$ .

### Ví dụ:



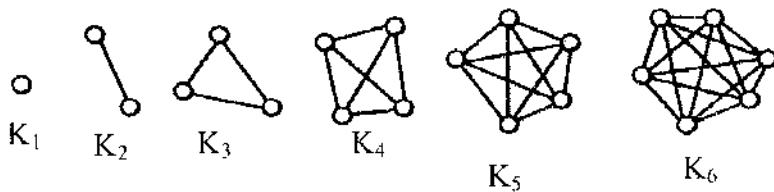
**Định lý:** Giả sử  $G$  là đồ thị có hướng (có thể là đơn hoặc đa) với tập đỉnh  $V$  và tập cạnh  $E$ . Khi đó:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = \frac{1}{2} \sum_{v \in V} \deg(v) = |E|.$$

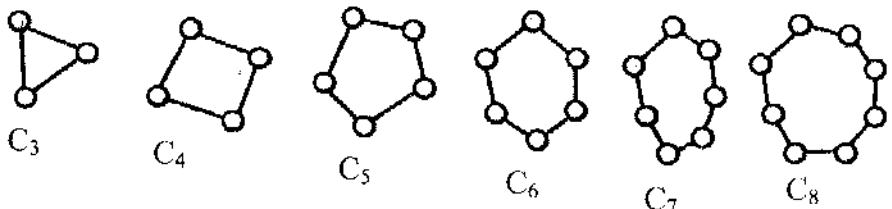
Chú ý: khái niệm bậc của đỉnh là không thay đổi cho dù ta xét đồ thị vô hướng hay có hướng.

### Một số dạng đơn đồ thị vô hướng đặc biệt

**Đồ thị đầy đủ:** Với  $n \in \mathbb{N}$ , đồ thị đầy đủ  $n$  đỉnh,  $K_n$ , là đơn đồ thị vô hướng với  $n$  đỉnh trong đó giữa hai đỉnh bất kỳ luôn có cạnh nối:  $\forall u, v \in V: u \neq v \Leftrightarrow \{u, v\} \in E$ .

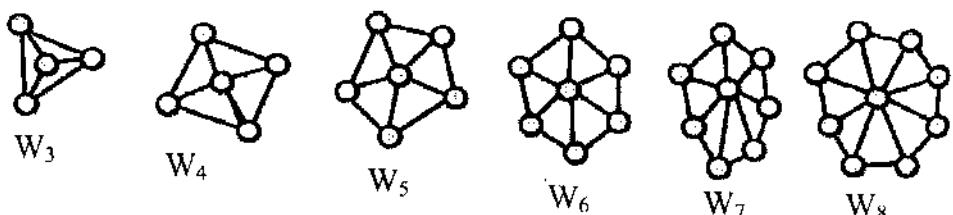


**Chu trình:** Với  $n \geq 3$ , chu trình  $n$  đỉnh,  $C_n$ , là đơn đồ thị vô hướng với  $V = \{v_1, v_2, \dots, v_n\}$  và  $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$ .



**Bánh xe:** Với  $n \geq 3$ , bánh xe  $W_n$ , là đơn đồ thị vô hướng thu được bằng cách bổ sung vào chu trình  $C_n$  một đỉnh  $v_{hub}$  và  $n$  cạnh nối:

$$\{\{v_{hub}, v_1\}, \{v_{hub}, v_2\}, \dots, \{v_{hub}, v_n\}\}.$$



**Định nghĩa:** Đồ thị  $G = (V, E)$  là hai phía nếu và chỉ nếu:

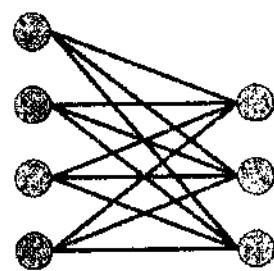
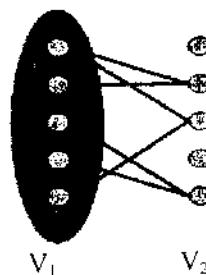
$V = V_1 \cup V_2$  với  $V_1 \cap V_2 = \emptyset$  và  $\forall e \in E: \exists v_1 \in V_1, v_2 \in V_2: e = \{v_1, v_2\}$ .

Nói cách khác, có thể phân hoạch tập đỉnh thành hai tập sao cho mỗi cạnh nối hai đỉnh thuộc hai tập khác nhau.

**Đồ thị hai phía đầy đủ:** Với  $m, n \in \mathbb{N}$ , đồ thị hai phía đầy đủ  $K_{m,n}$  là đồ thị hai phía trong đó  $|V_1| = m, |V_2| = n$  và  $E = \{(v_1, v_2) | v_1 \in V_1 \text{ và } v_2 \in V_2\}$ .

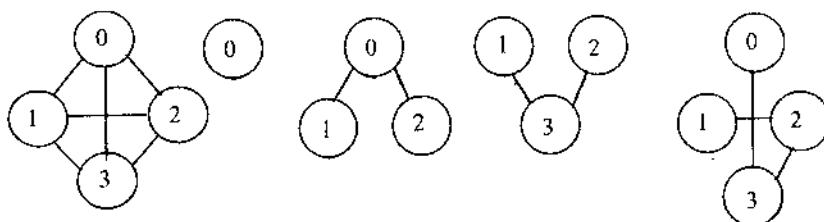
$K_{m,n}$  có  $m$  đỉnh ở tập bên trái,  $n$  đỉnh ở tập bên phải và mỗi đỉnh ở phần bên trái được nối với mỗi đỉnh ở phần bên phải.

**Đồ thị con:** Đồ thị con của đồ thị  $G = (V, E)$  là đồ thị  $H = (W, F)$  trong đó  $W \subseteq V$  và  $F \subseteq E$ .

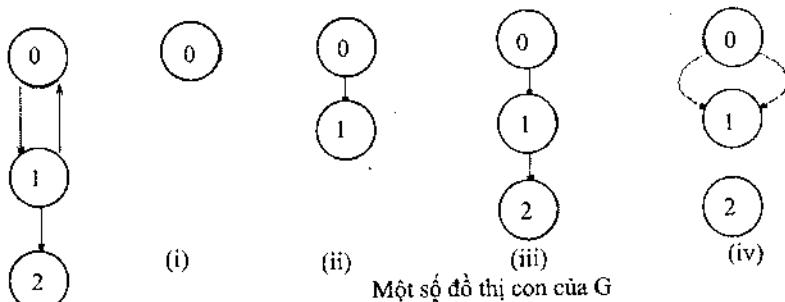


$K_{4,3}$

**Ví dụ:**



Đồ thị  $K_4$  và một số đồ thị con của nó



Đồ thị có hướng G và một số đồ thị con của nó

### 7.1.2. Đường đi, chu trình và tính liên thông của đồ thị

**Định nghĩa:** Đường đi  $P$  độ dài  $n$ , từ đỉnh  $u$  đến đỉnh  $v$ , trong đó  $n$  là số nguyên dương, trên đồ thị  $G = (V, E)$  là dãy:

$$P: \quad x_0, x_1, \dots, x_{n-1}, x_n,$$

trong đó:  $u = x_0, v = x_n, (x_i, x_{i+1}) \in E, i = 0, 1, 2, \dots, n-1$ .

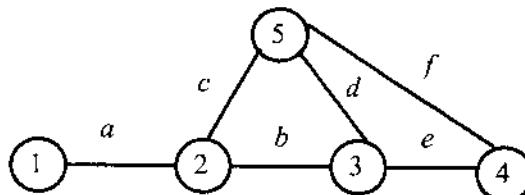
Đường đi nói trên còn có thể biểu diễn dưới dạng dãy các cạnh:

$$(x_0, x_1), (x_1, x_2), \dots, (x_{n-1}, x_n).$$

Đỉnh  $u$  gọi là *đỉnh đầu*, đỉnh  $v$  gọi là *đỉnh cuối* của đường đi.

Đường đi gọi là *đường đi đơn* nếu không có đỉnh nào bị lặp lại trên nó. Đường đi gọi là *đường đi cơ bản* nếu không có cạnh nào bị lặp lại trên nó.

**Ví dụ:** Xét đồ thị cho trong hình vẽ sau:



Dãy 5, 2, 3, 4 (hoặc 5, c, 2, b, 3, e, 4) là đường đi đơn.

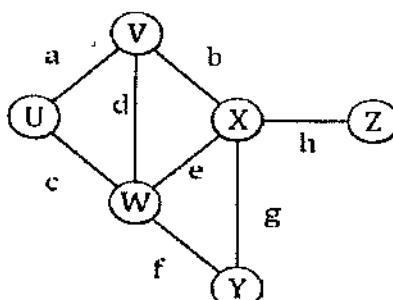
Dãy 1, 2, 5, 3, 4 (hoặc 1, a, 2, c, 5, d, 3, e, 4) là đường đi đơn.

Dãy 1, 2, 5, 3, 4, 5 là đường đi cơ bản nhưng không là đường đi đơn.

Dãy 1, 2, 3, 5, 2, 3, 4 là đường đi nhưng không là đường đi cơ bản.

**Định nghĩa:** Đường đi cơ bản có đỉnh đầu trùng với đỉnh cuối (tức là  $u = v$ ) được gọi là *chu trình*. Chu trình được gọi là *đơn* nếu như ngoại trừ đỉnh đầu trùng với đỉnh cuối, không có đỉnh nào bị lặp lại.

**Ví dụ:** Xét đồ thị cho trong hình vẽ sau:



Ta có:

–  $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$  là chu trình đơn;

–  $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$  là chu trình nhưng không là chu trình đơn.

**Định nghĩa:** Đồ thị vô hướng được gọi là *liên thông* nếu luôn tìm được đường đi nối hai đỉnh bất kỳ của nó.

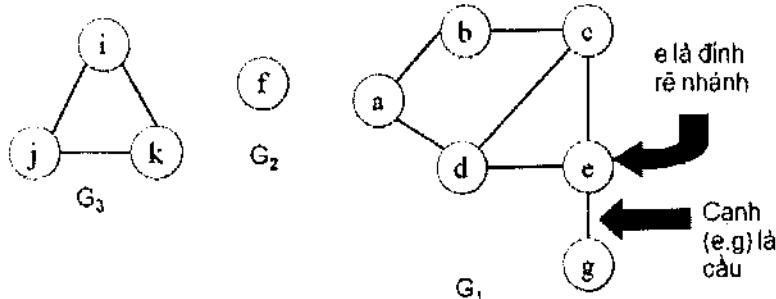
**Mệnh đề:** Luôn tìm được đường đi đơn nối hai đỉnh bất kỳ của đồ thị vô hướng liên thông.

**Thành phần liên thông (Connected component):** Đồ thị con liên thông cực đại của đồ thị vô hướng  $G$  được gọi là thành phần liên thông của nó.

**Đỉnh rẽ nhánh (cut vertex):** là đỉnh mà việc loại bỏ nó làm tăng số thành phần liên thông của đồ thị.

**Cầu (bridge):** Cạnh mà việc loại bỏ nó làm tăng số thành phần liên thông của đồ thị.

**Ví dụ:**



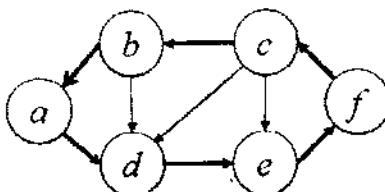
**Đồ thị  $G$  và ba thành phần liên thông  $G_1$ ,  $G_2$  và  $G_3$**

**Định nghĩa:** Đồ thị có hướng được gọi là *liên thông mạnh (strongly connected)* nếu như luôn tìm được đường đi nối hai đỉnh bất kỳ của nó.

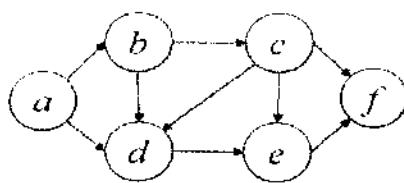
Đồ thị có hướng được gọi là *liên thông yếu (weakly connected)* nếu như đồ thị vô hướng thu được từ nó bởi việc bỏ qua hướng của tất cả các cạnh của nó là đồ thị vô hướng liên thông.

Để thấy nếu  $G$  là liên thông mạnh thì nó cũng là liên thông yếu, nhưng điều ngược lại không luôn đúng.

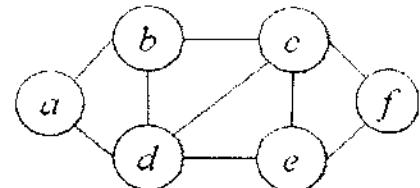
**Ví dụ:**



**Đồ thị có hướng liên thông mạnh  $G$ ,**



Đồ thị có hướng liên thông yếu  $G_2$



Đồ thị vô hướng tương ứng với  $G_1$  và  $G_2$

### Các phép toán cơ bản đối với đồ thị

- Khởi tạo đồ thị (create).
- Hủy đồ thị (destroy).
- Nhận số cạnh của đồ thị (get number of edges).
- Nhận số đỉnh của đồ thị (get number of vertices).
- Cho biết đồ thị là có hướng hay vô hướng (isdirected).
- Bổ sung cạnh (insert an edge); loại bỏ cạnh (remove an edge).
- Có cạnh nối giữa hai đỉnh.
- Duyệt các đỉnh kề của một đỉnh cho trước.

Những thao tác phức tạp hơn đối với đồ thị là các bài toán xử lý đồ thị:

- Tính giá trị của một số đặc trưng số của đồ thị (số liên thông, sắc số,...).
- Tìm một số tập con cạnh đặc biệt (cặp ghép, bè, chu trình, cây khung,...).
- Tìm một số tập con đỉnh đặc biệt (phủ đỉnh, phủ cạnh, tập độc lập,...).
- Trả lời truy vấn về một số tính chất của đồ thị (song liên thông, phẳng,...).
- Các bài toán tối ưu trên đồ thị: Bài toán cây khung nhỏ nhất, Bài toán đường đi ngắn nhất, Bài toán luồng cực đại trong mạng,...
- ...

## 7.2. BIỂU DIỄN ĐỒ THỊ

Có nhiều cách biểu diễn đồ thị. Việc lựa chọn cách biểu diễn phụ thuộc vào từng bài toán cụ thể cần xét, từng thuật toán cụ thể cần cài đặt. Có hai vấn đề chính cần quan tâm khi lựa chọn cách biểu diễn:

- Bộ nhớ mà cách biểu diễn đó đòi hỏi.
- Thời gian cần thiết để trả lời các truy vấn thường xuyên đối với đồ thị trong quá trình xử lý đồ thị. Chẳng hạn: "Có cạnh nối hai đỉnh  $u, v$ ?", "Liệt kê các đỉnh kề của đỉnh  $v$ ?",...

### 7.2.1. Biểu diễn đồ thị bởi ma trận

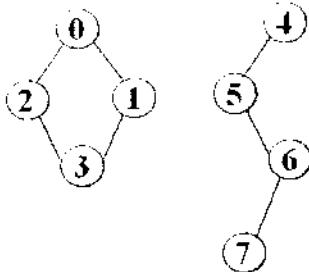
#### Ma trận kề (Adjacency matrix)

Cho đồ thị  $G = (V, E)$ . Ma trận kề của đồ thị  $G$  là ma trận  $A$  kích thước  $|V| \times |V|$ , trong đó mỗi cạnh  $e = (u, v)$  tương ứng với phần tử khác 0 ở vị trí  $(u, v)$  của ma trận, nghĩa là:

$$A[u, v] = \begin{cases} 1, & (u, v) \in E, \\ 0, & (u, v) \notin E. \end{cases}$$

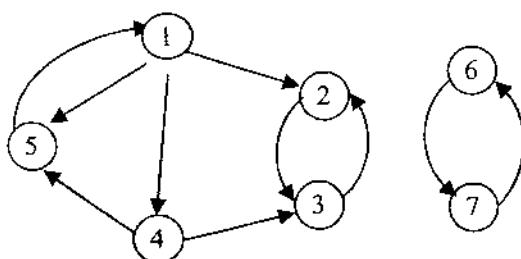
Bộ nhớ đòi hỏi là:  $\Theta(|V|^2)$ .

**Ví dụ:**



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Đồ thị vô hướng  $G$  và ma trận kề



$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Đồ thị có hướng  $G$  và ma trận kề

#### Tính chất của ma trận kề

Một trong những ưu điểm nổi bật của ma trận kề là để trả lời câu hỏi " $(u, v)$  có phải là cạnh của đồ thị hay không?", ta chỉ mất thời gian  $O(1)$ .

Giả sử  $A$  là ma trận kề của đồ thị vô hướng. Khi đó:

- $A$  là ma trận đối xứng:  $A = A^T$  ( $a_{ij} = a_{ji}$ ).
- $\deg(v) = \text{Tổng các phần tử trên dòng } v \text{ của } A$ .
- Nếu ký hiệu  $A^k = (a^{(k)}[u, v])$  thì  $a^{(k)}[u, v]$  là số lượng đường đi từ  $u$  đến  $v$  đi qua không quá  $k - 1$  đỉnh trung gian.

Giả sử  $A$  là ma trận kề của đồ thị có hướng. Khi đó:

- Bán bậc vào của đỉnh  $v$ :  $\deg^-(v) = \text{Tổng các phần tử trên cột } v \text{ của } A$ .
- Bán bậc ra của đỉnh  $v$ :  $\deg^+(v) = \text{Tổng các phần tử trên dòng } v \text{ của } A$ .

**Chú ý:** Khái niệm ma trận kề có thể mở rộng để biểu diễn đa đồ thị vô hướng:

$a_{uv}$  – số lượng cạnh nối hai đỉnh  $u$  và  $v$ .

### Ma trận trọng số

Trong trường hợp đồ thị có trọng số trên cạnh, thay vì ma trận kề, để biểu diễn đồ thị, ta sử dụng ma trận trọng số.

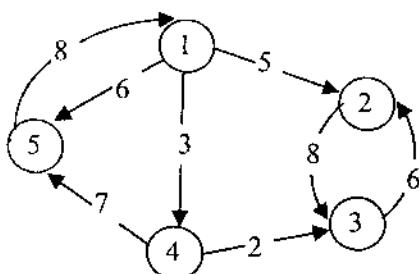
$$C = c[i, j], i, j = 1, 2, \dots, n,$$

với

$$c[i, j] = \begin{cases} c(i, j), & \text{nếu } (i, j) \in E \\ \theta, & \text{nếu } (i, j) \notin E, \end{cases}$$

trong đó:  $\theta$  là giá trị đặc biệt để chỉ ra một cặp  $(i, j)$  không là cạnh, tùy từng trường hợp cụ thể, có thể được đặt bằng một trong các giá trị sau:  $0, +\infty, -\infty$ .

**Ví dụ:** Hình vẽ dưới đây cho đồ thị có trọng số (trọng số là số viết trên mỗi cạnh) và ma trận trọng số của nó.



Đồ thị có trọng số  $G$

$$A = \begin{bmatrix} \theta & 5 & \theta & 3 & 6 \\ \theta & \theta & 8 & \theta & \theta \\ \theta & 6 & \theta & \theta & \theta \\ \theta & \theta & 2 & \theta & 7 \\ 8 & \theta & \theta & \theta & \theta \end{bmatrix}$$

Ma trận trọng số của  $G$

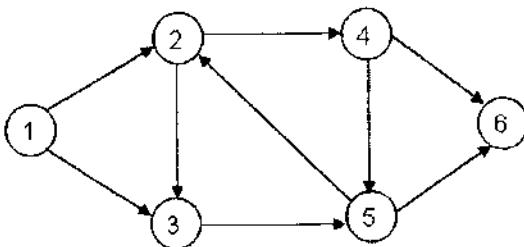
### Ma trận liên thuộc đỉnh cạnh

Xét  $G = (V, E)$ , ( $V = \{1, 2, \dots, n\}$ ,  $E = \{e_1, e_2, \dots, e_m\}$ ), là đơn đồ thị có hướng. Ma trận liên thuộc đỉnh cạnh  $A = (a_{ij}; i = 1, 2, \dots, n; j = 1, 2, \dots, m)$  được xây dựng theo quy tắc:

$$a_{ij} = \begin{cases} 1, & \text{nếu } i \text{ là đỉnh đầu của cung } e_j \\ -1, & \text{nếu } i \text{ là đỉnh cuối của cung } e_j \\ 0, & \text{nếu } i \text{ không là đầu mút của cung } e_j \end{cases}$$

Ma trận liên thuộc đỉnh – cạnh là một trong những cách biểu diễn rất hay được sử dụng trong các bài toán liên quan đến đồ thị có hướng mà trong đó phải xử lý các cung của đồ thị.

**Ví dụ:** Xét đồ thị:



Ma trận liên thuộc đỉnh cạnh của nó là:

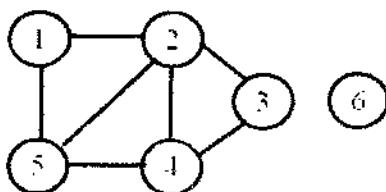
$$A = \begin{bmatrix} (1,2) & (1,3) & (2,3) & (2,4) & (3,5) & (4,5) & (4,6) & (5,2) & (5,6) \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & -1 \\ 3 & 0 & -1 & -1 & 0 & 1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & -1 & 0 & 1 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix}$$

### 7.2.2. Biểu diễn đồ thị bởi danh sách kề

Khi cần biểu diễn đồ thị thưa (sparse graphs) tức là khi  $|E| = O(|V|)$ , ta thường sử dụng danh sách kề để biểu diễn đồ thị. Danh sách kề của đỉnh  $v$  là tập:

$$\text{Adj}(v) = \{u; (u, v) \in E\}.$$

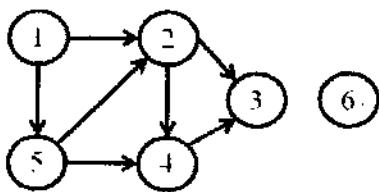
**Ví dụ:** Các hình vẽ dưới đây cho danh sách kề của đồ thị vô hướng  $G_1$  và đồ thị có hướng  $G_2$ :



Đồ thị  $G_1$

| $v$ | $Adj(v)$   |
|-----|------------|
| 1   | 2, 5       |
| 2   | 1, 3, 4, 5 |
| 3   | 2, 4       |
| 4   | 2, 3, 5    |
| 5   | 1, 2, 4    |
| 6   |            |

Danh sách kề của  $G_1$



Đồ thị  $G_2$

| $v$ | $Adj(v)$ |
|-----|----------|
| 1   | 2, 5     |
| 2   | 3, 4     |
| 3   |          |
| 4   | 3        |
| 5   | 2, 4     |
| 6   |          |

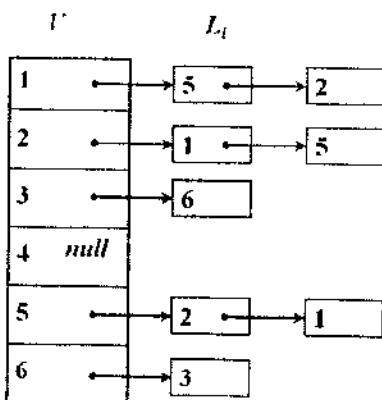
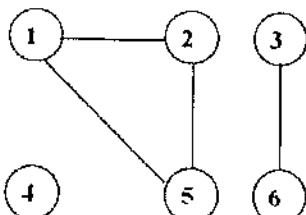
Danh sách kề của  $G_2$

Để cài đặt danh sách kề, ta có thể sử dụng  $n$  danh sách liên kết, mỗi danh sách chứa các đỉnh kề của một đỉnh của đồ thị.

**Ví dụ:** Biểu diễn đồ thị vô hướng  $G = (V, E)$  bởi danh sách kề:

$$V = \{1, 2, 3, 4, 5, 6\},$$

$$E = \{(1,2), (1,5), (2,5), (3,6)\}.$$

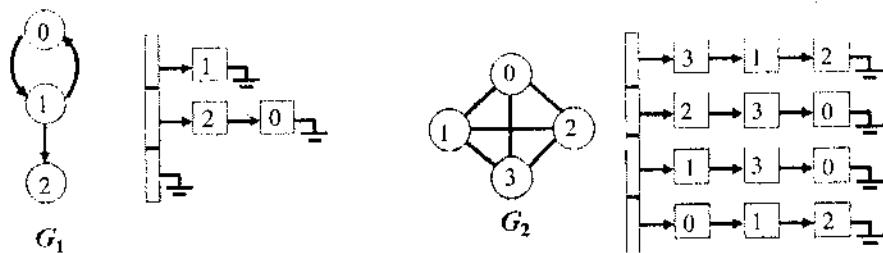


Có thể sử dụng mô tả trên C sau đây để biểu diễn danh sách kè:

```
#define MAX_VERTICES 500
typedef struct node *node_ptr;
typedef struct node {
 int vertex;
 node_ptr link;
} node;
node_ptr graph[MAX_VERTICES];
```

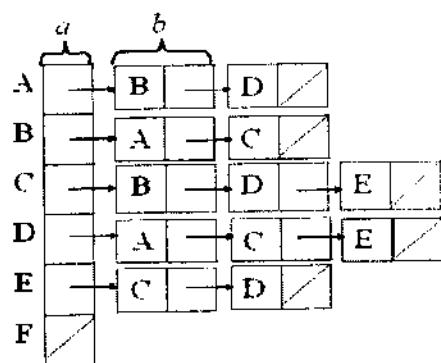
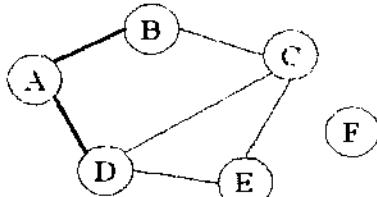
### Ví dụ:

Hình vẽ dưới đây mô tả danh sách kè của đồ thị có hướng  $G_1$  và đồ thị vô hướng  $G_2$ .



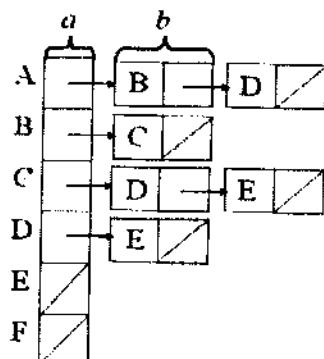
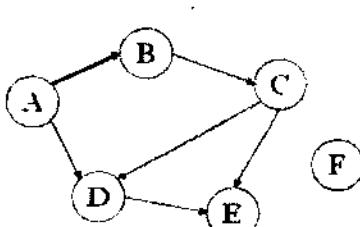
### Phân tích yêu cầu bộ nhớ

- Đối với đồ thị vô hướng: Giả sử mỗi con trỏ đòi hỏi  $a$  đơn vị bộ nhớ và mỗi nút của danh sách đòi hỏi  $b$  đơn vị bộ nhớ.



Khi đó biểu diễn đồ thị vô hướng bởi danh sách kè đòi hỏi  $a|V| + 2b|E|$  đơn vị bộ nhớ.

Dồi với đồ thị có hướng: Giả sử mỗi con trỏ đòi hỏi  $a$  đơn vị bộ nhớ và mỗi nút của danh sách đòi hỏi  $b$  đơn vị bộ nhớ.



Khi đó biểu diễn đồ thị vô hướng bởi danh sách kè đòi hỏi  $a|V| + b|E|$  đơn vị bộ nhớ.

Như vậy tổng cộng bộ nhớ để biểu diễn đồ thị bởi danh sách kè là:  $\Theta(|V| + |E|)$ .

**Chú ý:** Đòi hỏi bộ nhớ này thường nhỏ hơn nhiều so với đòi hỏi bộ nhớ của cách biểu diễn ma trận kè ( $|V|^2$ ), nhất là đối với đồ thị thưa (đó là đồ thị thỏa mãn  $|E| \leq k|V|$  với  $k < 10$ ). Do phần lớn các đồ thị trong thực tế ứng dụng là đồ thị thưa, nên cách biểu diễn này được sử dụng nhiều nhất trong ứng dụng.

### Danh sách cạnh

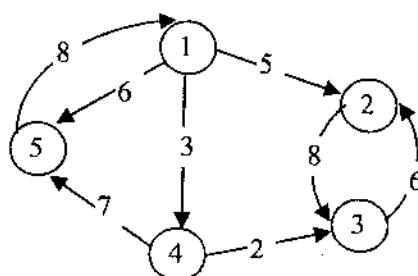
Với mỗi cạnh  $e = (u, v)$ , ta cần giữ:

$$\text{dau}[e] = u, \text{cuoi}[e] = v.$$

Nếu đồ thị có trọng số trên cạnh, thì cần có thêm một biến cát giữ  $c[e]$ .

**Ví dụ:**

Xét đồ thị có trọng số:



Danh sách cạnh được cho trong bảng sau:

| $e$ | $dau[e]$ | $cuoi[e]$ | $e[e]$ |
|-----|----------|-----------|--------|
| 1   | 1        | 5         | 6      |
| 2   | 5        | 1         | 8      |
| 3   | 4        | 5         | 7      |
| 4   | 1        | 4         | 3      |
| 5   | 1        | 2         | 5      |
| 6   | 4        | 3         | 2      |
| 7   | 2        | 3         | 8      |
| 8   | 3        | 2         | 6      |

Mặc dù cách biểu diễn này không thuận tiện cho các thao tác với đồ thị, nhưng đây là cách chuẩn bị dữ liệu thường gặp nhất cho các đồ thị thực tế.

### Những thao tác cơ bản thường gặp khi xử lý đồ thị

- incidentEdges( $v$ ) – duyệt các đỉnh kề của đỉnh  $v$ .
- areAdjacent( $v, w$ ) – trả lại giá trị true khi và chỉ khi hai đỉnh  $v, w$  là kề nhau.
- insertVertex( $z$ ) – bổ sung đỉnh  $z$ .
- insertEdge( $v, w, e$ ) – bổ sung cạnh  $e = (v, w)$ .
- removeVertex( $v$ ) – loại bỏ đỉnh  $v$ .
- removeEdge( $e$ ) – loại bỏ cạnh  $e$ .

Bảng sau đây cho ta đánh giá bộ nhớ cần sử dụng và thời gian thực hiện các thao tác cơ bản đối với các cách biểu diễn đồ thị khác nhau, với giả thiết đồ thị là đơn đồ thị vô hướng có  $n$  đỉnh và  $m$  cạnh:

| Thao tác                | Danh sách cạnh | Danh sách kề             | Mã trận kề |
|-------------------------|----------------|--------------------------|------------|
| Bộ nhớ                  | $n + m$        | $n + m$                  | $n^2$      |
| incidentEdges( $v$ )    | $m$            | $\deg(v)$                | $n$        |
| areAdjacent( $v, w$ )   | $m$            | $\min(\deg(v), \deg(w))$ | 1          |
| insertVertex( $z$ )     | 1              | 1                        | $n^2$      |
| insertEdge( $v, w, e$ ) | 1              | 1                        | 1          |
| removeVertex( $v$ )     | $m$            | $\deg(v)$                | $n^2$      |
| removeEdge( $e$ )       | 1              | 1                        | 1          |

Việc đưa ra đánh giá thời gian đối với các loại đồ thị khác được thực hiện hoàn toàn tương tự.

### 7.3. CÁC THUẬT TOÁN DUYỆT ĐỒ THỊ

Ta gọi *duyệt đồ thị* (Graph Searching hoặc Graph Traversal) là việc duyệt qua mỗi đỉnh và mỗi cạnh của đồ thị.

#### Ứng dụng

- Xây dựng các thuật toán khảo sát các tính chất của đồ thị;
- Là thành phần cơ bản của nhiều thuật toán.

Cần xây dựng thuật toán hiệu quả để thực hiện việc duyệt đồ thị. Ta xét hai thuật toán duyệt cơ bản:

- Tìm kiếm theo chiều rộng (Breadth First Search – BFS);
- Tìm kiếm theo chiều sâu (Depth First Search – DFS).

#### Ý tưởng chung

Trong quá trình thực hiện thuật toán, mỗi đỉnh ở một trong ba trạng thái:

- Chưa thăm (thể hiện bởi màu trắng);
- Đã thăm nhưng chưa duyệt xong (thể hiện bởi màu xám);
- Đã duyệt xong (thể hiện bởi màu đen).

Quá trình duyệt được bắt đầu từ một đỉnh  $v$  nào đó. Ta sẽ khảo sát các đỉnh đạt tới được từ  $v$ :

- Thoát đầu mỗi đỉnh đều có màu trắng (chưa thăm – not visited).
- Đỉnh đã được thăm sẽ chuyển thành màu xám (trở thành đã thăm nhưng chưa duyệt xong).
- Khi tất cả các đỉnh kề của một đỉnh  $v$  đã được thăm, đỉnh  $v$  sẽ có màu đen (đã duyệt xong).

#### 7.3.1. Thuật toán tìm kiếm theo chiều rộng (BFS)

- Input: Đồ thị  $G = (V, E)$ , có hướng hoặc vô hướng và đỉnh xuất phát  $s \in V$ .
- Output:

+ Với mọi  $v \in V$ ,

+  $d[v]$  = khoảng cách từ  $s$  đến  $v$ .

+  $\pi[v]$  – đỉnh đi trước  $v$  trong đường đi ngắn nhất từ  $s$  đến  $v$ .

+ Xây dựng cây BFS gốc tại  $s$  chứa tất cả các đỉnh đạt đến được từ  $v$ .

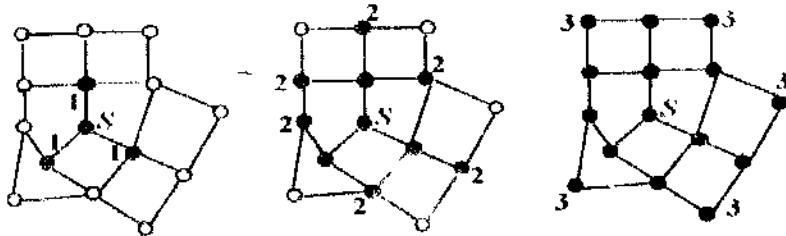
Ta sẽ sử dụng màu để ghi nhận trạng thái của đỉnh trong quá trình duyệt:

- *Trắng (White)* – chưa thăm.

- *Xám (Gray)* – đã thăm nhưng chưa duyệt xong.

- *Đen (Black)* – đã duyệt xong.

Ví dụ:



Tìm kiếm theo chiều rộng bắt đầu từ đỉnh  $s$

#### BFS\_Visit( $s$ )

1.     **for**  $u \in V - \{s\}$
2.         **do**  $\text{color}[u] \leftarrow \text{white}$
3.          $d[u] \leftarrow \infty$
4.          $\pi[u] \leftarrow \text{NULL}$
5.  $\text{color}[s] \leftarrow \text{gray}$
6.  $d[s] \leftarrow 0$
7.  $\pi[s] \leftarrow \text{NULL}$
8.  $Q \leftarrow \emptyset$
9.  $\text{enqueue}(Q, s)$
10.      **while**  $Q \neq \emptyset$
11.         **do**  $u \leftarrow \text{dequeue}(Q)$
12.             **for**  $v \in \text{Adj}[u]$
13.                 **do if**  $\text{color}[v] = \text{white}$
14.                     **then**  $\text{color}[v] \leftarrow \text{gray}$
15.                      $d[v] \leftarrow d[u] + 1$
16.                      $\pi[v] \leftarrow u$
17.                      $\text{enqueue}(Q, v)$
18.          $\text{color}[u] \leftarrow \text{black}$

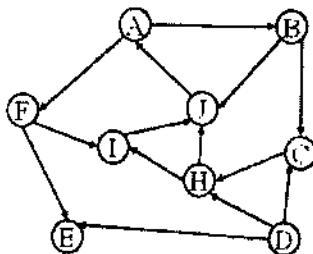
Thuật toán tìm kiếm theo chiều rộng trên đồ thị  $G$

#### BFS( $G$ )

1. **for**  $u \in V[G]$
2.     **do**  $\text{color}[u] \leftarrow \text{white}$
3.      $\pi[u] \leftarrow \text{NULL}$
4.  $\text{time} \leftarrow 0$

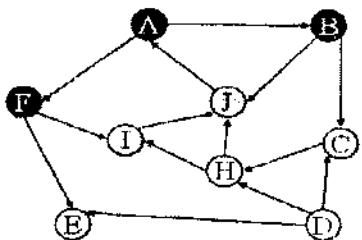
5. **for**  $u \in V[G]$
6.     **do if**  $\text{color}[u] = \text{white}$
7.         **then BFS-Visit( $u$ )**

**Ví dụ:** Xét đồ thị cho trong hình vẽ sau đây:



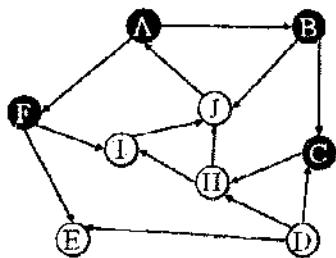
Ta xét việc thực hiện BFS( $G, A$ ):

| <i>Đồ thị G</i> | <i>Hàng đợi Q</i>               |
|-----------------|---------------------------------|
|                 | $Q = \{A\}$                     |
|                 | Khảo sát A<br>$Q = \{B, F\}$    |
|                 | Khảo sát B<br>$Q = \{F, C, G\}$ |



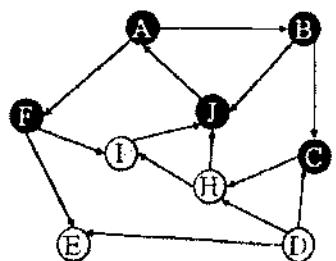
Khảo sát F

$$Q = \{C, J, E, I\}$$



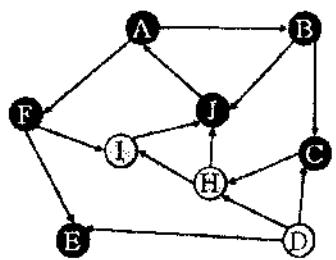
Khảo sát C

$$Q = \{J, E, I, H\}$$



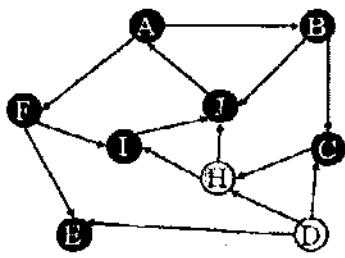
Khảo sát J

$$Q = \{E, I, H\}$$



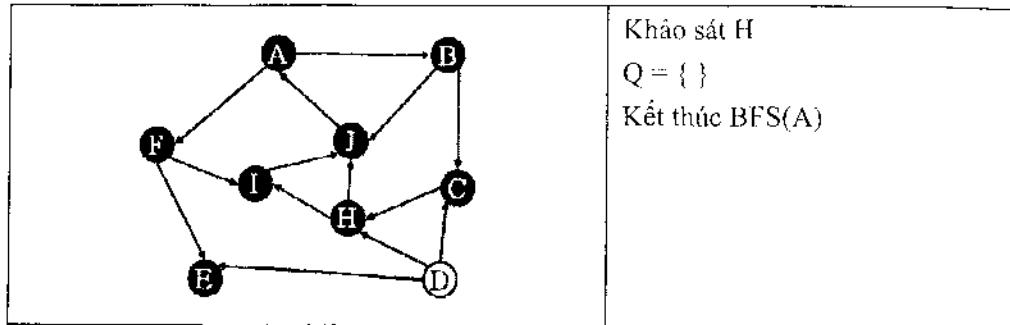
Khảo sát E

$$Q = \{I, H\}$$



Khảo sát I

$$Q = \{H\}$$



Ta có định lý sau đây khẳng định tính đúng đắn của BFS.

#### Dịnh lý:

1. BFS cho phép đến thăm tất cả các đỉnh  $v \in V$  đạt đến được từ  $s$ .
2. Khi thuật toán kết thúc,  $d[v]$  cho ta độ dài đường đi ngắn nhất (theo số cạnh) từ  $s$  đến  $v$ .
3. Với mỗi đỉnh  $v$  đạt đến được từ  $s$ ,  $\pi[v]$  cho ta đỉnh đi trước đỉnh  $v$  trong đường đi ngắn nhất từ  $s$  đến  $v$ .

#### Cây tìm kiếm theo chiều rộng (Breadth-first Tree)

- Đối với đồ thị  $G = (V, E)$  với đỉnh xuất phát  $s$ , ký hiệu  $G_\pi = (V_\pi, E_\pi)$  là đồ thị với:
  - +  $V_\pi = \{v \in V : \pi[v] \neq \text{NULL}\} \cup \{s\}$ .
  - +  $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$ .
- Đồ thị  $G_\pi$  được gọi là cây BFS( $s$ ):

  - +  $V_\pi$  chứa tất cả các đỉnh đạt đến được từ  $s$ ;
  - + Với mọi  $v \in V_\pi$ , đường đi từ  $s$  đến  $v$  trên  $G_\pi$  là đường đi ngắn nhất từ  $s$  đến  $v$  trên  $G$ .

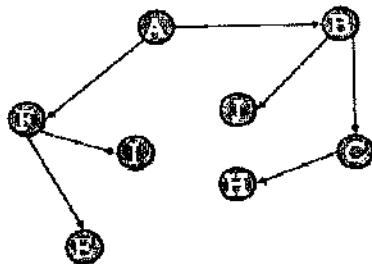
Các cạnh trong  $E_\pi$  được gọi là các *cạnh của cây*.

$$|E_\pi| = |V_\pi| - 1.$$

#### Độ phức tạp của BFS

- Thuật toán loại bỏ mỗi đỉnh khỏi hàng đợi đúng một lần, do đó thao tác DeQueue thực hiện đúng  $|V|$  lần.
- Với mỗi đỉnh, thuật toán duyệt qua tất cả các đỉnh kề của nó và thời gian xử lý mỗi đỉnh kề như vậy là hằng số. Như vậy thời gian thực hiện câu lệnh if trong vòng lặp while bằng hằng số nhân với số cạnh kề với đỉnh đang xét.
- Do đó tổng thời gian thực hiện việc duyệt qua tất cả các đỉnh bằng một hằng số nhân với số cạnh  $|E|$ .
- Thời gian tổng cộng:  $O(|V|) + O(|E|) = O(|V| + |E|)$ , hay  $O(|V|^2)$ .

**Ví dụ:** Cây BFS(A) trong ví dụ vừa trình bày ở trên có dạng:



### 7.3.2. Thuật toán tìm kiếm theo chiều sâu (DFS)

*Input:*  $G = (V, E)$  – đồ thị vô hướng hoặc có hướng.

*Output:* Với mỗi  $v \in V$ :

$d[v]$  = thời điểm bắt đầu thăm (v chuyển từ màu trắng sang xám);

$f[v]$  = thời điểm kết thúc thăm (v chuyển từ màu xám sang đen);

$\pi[v]$ : đỉnh từ đó ta đến thăm đỉnh  $v$ .

*Rừng tìm kiếm theo chiều sâu* (gọi tắt là rừng DFS – Forest of depth-first trees):

$G_\pi = (V, E_\pi)$ ;

$E_\pi = \{(\pi[v], v) : v \in V \text{ và } \pi[v] \neq \text{null}\}$ .

**Thuật toán tìm kiếm theo chiều sâu bắt đầu từ đỉnh  $u$**

**DFS-Visit( $u$ )**

1.  $color[u] \leftarrow \text{GRAY}$
2.  $time \leftarrow time + 1$
3.  $d[u] \leftarrow time$
4. **for**  $v \in Adj[u]$
5.     **do if**  $color[v] = \text{WHITE}$
6.         **then**  $\pi[v] \leftarrow u$
7.         DFS-Visit( $v$ )
8.      $color[u] \leftarrow \text{BLACK}$
9.      $f[u] \leftarrow time \leftarrow time + 1$

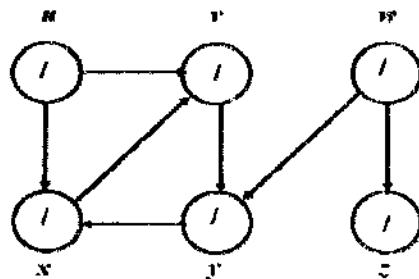
**Thuật toán tìm kiếm theo chiều sâu trên đồ thị  $G$**

**DFS( $G$ )**

1. **for**  $u \in V[G]$
2.     **do**  $color[u] \leftarrow \text{white}$
3.      $\pi[u] \leftarrow \text{NULL}$
4.      $time \leftarrow 0$

5. for  $u \in V[G]$
6.     do if  $\text{color}[u] = \text{white}$
7.         then DFS-Visit( $u$ )

**Ví dụ:** Xét việc thực hiện DFS( $G$ ) trên đồ thị cho trong hình vẽ sau đây:



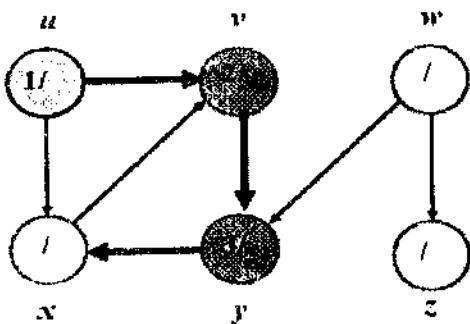
– DFS( $G$ ) sẽ gọi thực hiện DFS( $u$ ) và DFS( $w$ ).

– Cặp số viết trong mỗi đỉnh  $v$  là  $d[v]/f[v]$ .

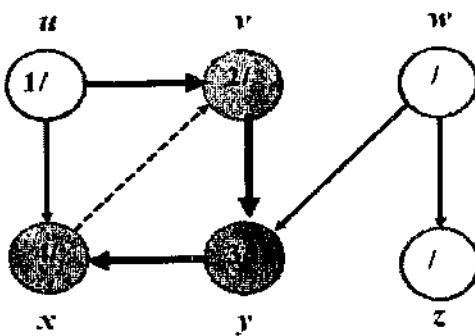
– Các cạnh đậm là các cạnh của rừng tìm kiếm.

Quá trình thực hiện thuật toán được trình bày trong các hình minh họa sau đây:

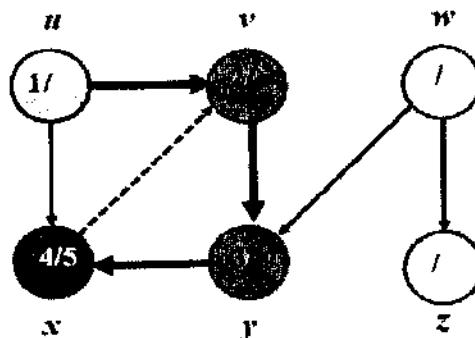
|  |                               |
|--|-------------------------------|
|  | DFS( $u$ ): Thăm đỉnh $u$<br> |
|  | DFS( $v$ ): Thăm đỉnh $v$<br> |



DFS(y): Thăm đỉnh  $y$



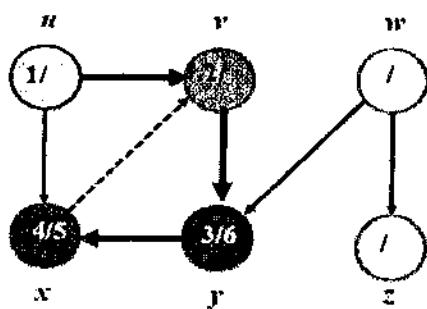
DFS(x): Thăm đỉnh  $x$



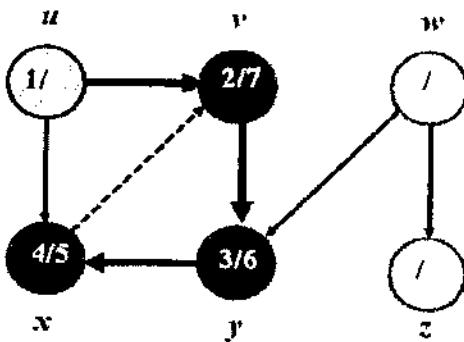
Kết thúc thăm đỉnh  $x$



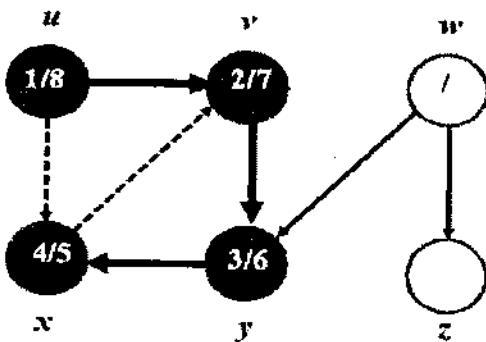
Kết thúc thăm đỉnh  $y$

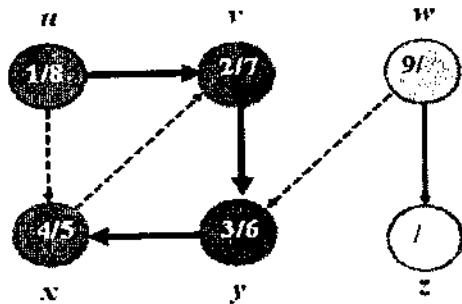


Kết thúc thăm đỉnh  $v$

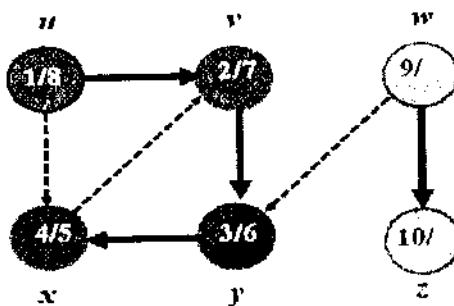


Kết thúc thăm đỉnh  $u$

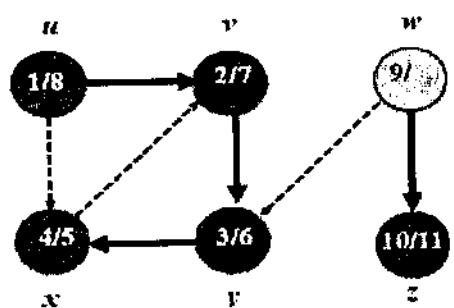




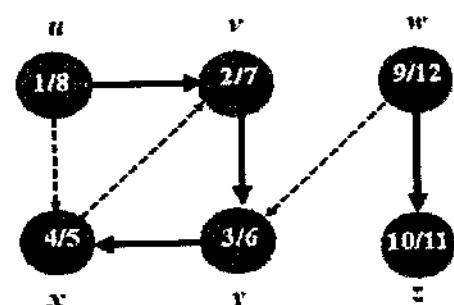
DFS( $w$ ): Thăm đỉnh  $w$



DFS( $z$ ): Thăm đỉnh  $z$



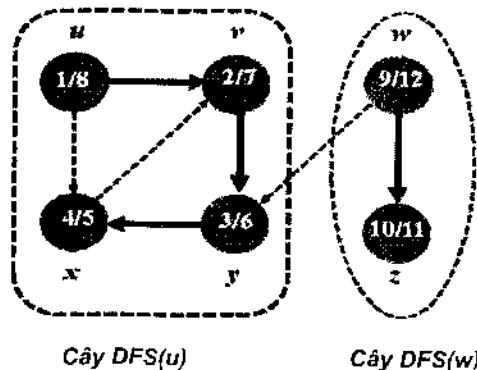
Kết thúc thăm đỉnh  $z$



Kết thúc thăm đỉnh  $w$

Kết thúc DFS( $G$ )

Rừng tìm kiếm gồm hai cây: cây DFS( $u$ ) và cây DFS( $w$ ):



### Các tính chất của DFS

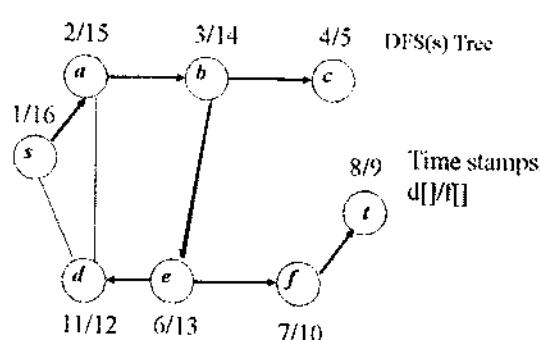
- Rừng DFS phụ thuộc vào thứ tự các đỉnh được duyệt trong các vòng lặp for duyệt đỉnh trong DFS( $G$ ) và DFS\_Visit( $u$ ).
- Để gỡ đệ quy có thể sử dụng ngăn xếp và có thể nói, điểm khác biệt cơ bản của DFS với BFS là các đỉnh đang được thăm trong DFS được cắt giữ vào ngăn xếp thay vì hàng đợi trong BFS.
- Các khoảng thời gian thăm  $[d[v], f[v]]$  của các đỉnh có cấu trúc lồng nhau (*parenthesis structure*).

### Cấu trúc lồng nhau (parenthesis structure)

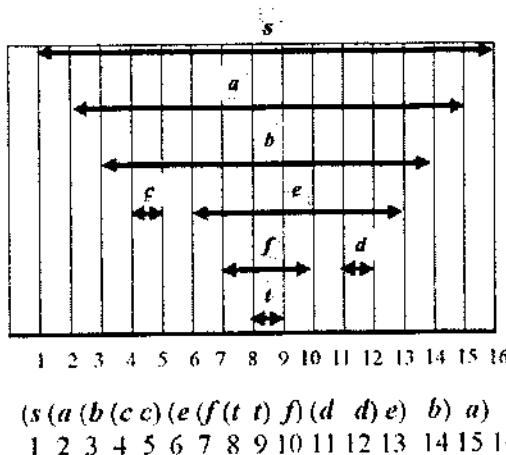
**Định lý:** Với mọi  $u, v$ , chỉ có thể xảy ra một trong các情形 sau:

1.  $d[u] < f[u] < d[v] < f[v]$  hoặc  $d[v] < f[v] < d[u] < f[u]$  (nghĩa là hai khoảng thời gian thăm của  $u$  và  $v$  là rời nhau) và khi đó  $u$  và  $v$  không có quan hệ tổ tiên – hậu duệ.
2.  $d[u] < d[v] < f[v] < f[u]$  (nghĩa là khoảng thời gian thăm của  $v$  lồng trong khoảng thời gian thăm của  $u$ ) và khi đó  $v$  là hậu duệ của  $u$ .
3.  $d[v] < d[u] < f[u] < f[v]$  (nghĩa là khoảng thời gian thăm của  $u$  lồng trong khoảng thời gian thăm của  $v$ ) và khi đó  $u$  là hậu duệ của  $v$ .

**Ví dụ:** Xét việc thực hiện thuật toán tìm kiếm theo chiều sâu trên đồ thị cho trong hình vẽ dưới đây:



Hình vẽ sau đây minh họa cho định lý về cấu trúc lồng nhau:



### Độ phức tạp của DFS

– Thuật toán thăm mỗi đỉnh  $v \in V$  đúng một lần, do đó tổng thời gian thăm đỉnh là  $\Theta(|V|)$ .

– Với mỗi đỉnh  $v$  duyệt qua tất cả các đỉnh kề, với mỗi đỉnh kề thực hiện thao tác với thời gian hằng số. Do đó việc duyệt qua tất cả các đỉnh mất thời gian là:

$$\sum_{v \in V} |\text{Adj}[v]| = \Theta(|E|).$$

Tổng cộng:  $\Theta(|V|) + \Theta(|E|) = \Theta(|V|+|E|)$ , hay  $\Theta(|V|^2)$ .

Như vậy, DFS có cùng độ phức tạp như BFS.

### Phân loại cạnh

DFS tạo ra một cách phân loại các cạnh của đồ thị đã cho:

– **Cạnh của cây (Tree edge):** là cạnh mà theo đó từ một đỉnh ta đến thăm một đỉnh mới.

– **Cạnh ngược (Back edge):** đi từ con cháu (descendant) đến tổ tiên (ancestor).

– **Cạnh tới (Forward edge):** đi từ tổ tiên đến hậu duệ.

– **Cạnh vòng (Cross edge):** cạnh nối hai đỉnh không có quan hệ họ hàng.

Để nhận biết cạnh  $(u, v)$  thuộc loại cạnh nào, ta dựa vào màu của đỉnh  $v$  khi lần đầu tiên cạnh  $(u, v)$  được khảo sát. Cụ thể, nếu màu của đỉnh  $v$  là:

– Trắng, thì  $(u, v)$  là cạnh của cây;

– Xám, thì  $(u, v)$  là cạnh ngược;

– Đen, thì  $(u, v)$  là cạnh tới hoặc vòng. Trong trường hợp này, để phân biệt cạnh tới và cạnh vòng, ta cần xét xem hai đỉnh  $u$  và  $v$  có quan hệ họ hàng hay không nhờ sử dụng kết quả của định lý về cấu trúc lồng nhau.

Nhiều ứng dụng của DFS chỉ đòi hỏi nhận biết cạnh của cây và cạnh ngược. Hơn nữa, đối với đồ thị vô hướng, DFS chỉ sản sinh ra hai loại cạnh này như chỉ ra trong khẳng định của định lý sau đây.

**Định lý:** Nếu  $G$  là đồ thị vô hướng, thì DFS chỉ sản sinh ra cạnh của cây và cạnh ngược.

#### Chứng minh:

Giả sử  $(u,v) \in E$ , không giảm tông quát giả sử  $d[u] < d[v]$ . Khi đó  $v$  phải trở thành đã duyệt xong trước khi  $u$  trở thành đã duyệt xong.

– Nếu  $(u,v)$  được khảo sát lần đầu tiên theo hướng  $u \rightarrow v$ , thì trước thời điểm khảo sát  $v$  phải có màu trắng và do đó  $(u,v)$  là cạnh của cây.

– Nếu  $(u,v)$  được khảo sát lần đầu tiên theo hướng  $v \rightarrow u$ ,  $u$  phải có màu xám tại thời điểm khảo sát cạnh này và do đó nó là cạnh ngược.

### 7.4. MỘT SỐ ỨNG DỤNG CỦA TÌM KIẾM TRÊN ĐỒ THỊ

Các thuật toán tìm kiếm trên đồ thị BFS và DFS được ứng dụng để giải nhiều bài toán trên đồ thị, chẳng hạn như:

- Tìm đường đi giữa hai đỉnh  $s$  và  $t$  của đồ thị;
- Kiểm tra tính liên thông, liên thông mạnh của đồ thị;
- Xác định các thành phần liên thông, song liên thông, liên thông mạnh;
- Tính hai phía của đồ thị;
- Tính phẳng của đồ thị.
- ...

Trong mục này ta sẽ xét một số ứng dụng đơn giản.

#### 7.4.1. Bài toán đường đi

Bài toán đặt ra là: "Cho đồ thị  $G = (V,E)$  và hai đỉnh  $s, t$  của nó. Hỏi có tồn tại đường đi từ  $s$  đến  $t$  hay không? Trong trường hợp câu trả lời là khẳng định cần đưa ra một đường đi từ  $s$  đến  $t$ ."

Để giải bài toán này, ta có thể thực hiện `DFS_Visit(s)` hoặc `BFS_Visit(s)`. Kết thúc, nếu đỉnh  $t$  được thăm thì câu trả lời là khẳng định và khi đó để đưa ra đường đi từ  $s$  đến  $t$ , ta sử dụng biến ghi nhận  $\pi[v]$ :

$$s \leftarrow \pi[s] \leftarrow \pi[\pi[s]] \leftarrow \dots \leftarrow t.$$

Nếu  $t$  không được thăm, ta khẳng định là không có đường đi cần tìm.

**Chú ý:** Đường đi tìm được từ  $s$  đến  $t$  theo `BFS_Visit(s)` là đường đi ngắn nhất (theo số cạnh).

### 7.4.2. Bài toán liên thông

Bài toán này được xét đối với đồ thị vô hướng và có hướng, được gọi tương ứng là bài toán liên thông và bài toán liên thông mạnh.

**Bài toán liên thông:** Cho đồ thị vô hướng  $G = (V, E)$ . Hãy kiểm tra xem đồ thị  $G$  có phải liên thông hay không. Nếu  $G$  không là liên thông, cần đưa ra số lượng thành phần liên thông và danh sách các đỉnh của từng thành phần liên thông.

Để giải bài toán này, ta chỉ việc thực hiện DFS( $G$ ) (hoặc BFS( $G$ )). Khi đó số lần gọi thực hiện BFS\_Visit() (DFS\_Visit()) sẽ chính là số lượng thành phần liên thông của đồ thị. Việc đưa ra danh sách các đỉnh của từng thành phần liên thông đòi hỏi phải đưa thêm vào biến ghi nhận xem mỗi đỉnh được thăm ở lần gọi nào trong BFS( $G$ ) (DFS( $G$ )).

**Bài toán liên thông mạnh:** Cho đồ thị có hướng  $G = (V, E)$ . Hãy kiểm tra xem đồ thị  $G$  có phải liên thông mạnh hay không?

Kết quả sau đây cho phép quy dẫn bài toán cần giải về bài toán đường đi.

**Mệnh đề:** Đồ thị có hướng  $G = (V, E)$  là liên thông mạnh khi và chỉ khi luôn tìm được đường đi từ một đỉnh  $v$  đến tất cả các đỉnh còn lại và luôn tìm được đường đi từ tất cả các đỉnh thuộc  $V \setminus \{v\}$  đến  $v$ .

**Chứng minh:** Điều kiện cần là hiển nhiên. Để chứng minh điều kiện đủ, ta chứng minh luôn tìm được đường đi nối hai đỉnh  $u, w$  bất kỳ của đồ thị. Thực vậy, do có đường đi từ  $V \setminus \{v\}$  đến  $v$ , nên suy ra có đường đi từ  $u$  đến  $v$  và do luôn có đường đi từ  $v$  đến mọi đỉnh còn lại nên có đường đi từ  $v$  đến  $w$ . Từ đó suy ra ta có đường đi từ  $u$  đến  $w$  đi thông qua  $v$ . Vậy, đồ thị là liên thông mạnh.

#### Đồ thị đảo hướng (đồ thị chuyển vị)

Cho đồ thị có hướng  $G = (V, E)$ . Ta gọi đồ thị đảo hướng (đồ thị chuyển vị) của đồ thị  $G$  là đồ thị có hướng  $G^T = (V, E^T)$ , với  $E^T = \{(u, v) : (v, u) \in E\}$ , nghĩa là tập cung  $E^T$  thu được từ  $E$  bởi việc đảo ngược hướng của tất cả các cung. Để thấy nếu  $A$  là ma trận kè của  $G$  thì ma trận chuyển vị  $A^T$  là ma trận kè của  $G^T$  (điều này giải thích tên gọi đồ thị chuyển vị). Dễ dàng xây dựng được danh sách kè của đồ thị  $G^T$  từ danh sách kè của đồ thị  $G$  sau thời gian  $\Theta(|V| + |E|)$ .

Từ mệnh đề ta suy ra có thể sử dụng thuật toán sau đây để giải quyết bài toán liên thông mạnh.

#### Thuật toán kiểm tra tính liên thông mạnh

- Chọn  $v \in V$  là một đỉnh tùy ý.
- Thực hiện DFS( $v$ ) trên  $G$ . Nếu tồn tại đỉnh  $u$  không được thăm thì  $G$  không liên thông mạnh và thuật toán kết thúc, trái lại thực hiện tiếp.
- Thực hiện DFS( $v$ ) trên  $G^T = (V, E^T)$ . Nếu tồn tại đỉnh  $u$  không được thăm thì  $G$  không liên thông mạnh, trái lại  $G$  là liên thông mạnh.

Dễ dàng thấy rằng thuật toán có thể cài đặt với thời gian  $\Theta(|V| + |E|)$ .

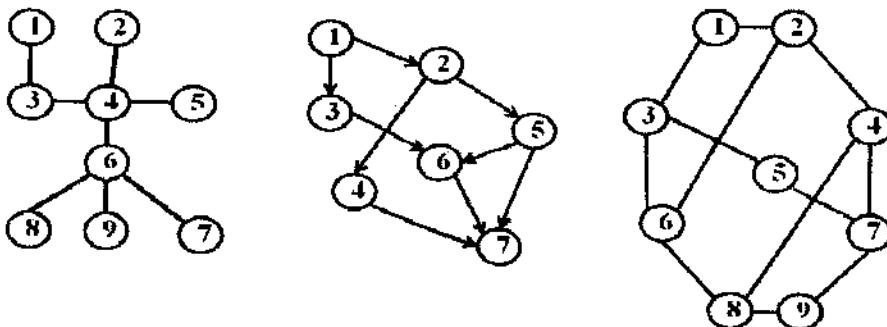
### 7.4.3. Đồ thị không chứa chu trình và bài toán sắp xếp tópô

#### Chu trình trong đồ thị

Trong mục này ta xây dựng thuật toán khảo sát chu trình trong đồ thị. Cả hai thuật toán BFS và DFS đều có thể sử dụng làm cơ sở để phát triển các thuật toán như vậy. Đồ thị không chứa chu trình còn được gọi là đồ thị phi chu trình (acyclic). Đồ thị có hướng không có chu trình sẽ được gọi là DAG. Đôi khi ta còn quan tâm đến chu trình ngắn nhất (có ít cạnh nhất).

**Định nghĩa:** Đối với đồ thị chứa chu trình, độ dài của chu trình ngắn nhất được gọi là *chu vi* của đồ thị. Nếu đồ thị không chứa chu trình thì chu vi của nó được đặt bằng  $+\infty$  (hoặc  $|V|+1$ ).

**Ví dụ:** Ta có ba đồ thị. Đồ thị thứ nhất là phi chu trình, đồ thị thứ hai không có chu trình có hướng, đồ thị thứ ba có chu vi là 4.



Kết quả sau đây sẽ là cơ sở để xây dựng thuật toán xác định xem đồ thị có chứa chu trình hay không.

**Mệnh đề:** Đồ thị có hướng  $G$  là không chứa chu trình khi và chỉ khi DFS thực hiện đối với  $G$  không phát hiện ra cạnh ngược.

**Chứng minh:**

$\Rightarrow)$  Nếu  $G$  không chứa chu trình thì không thể có cạnh ngược. Điều này là hiển nhiên vì sự tồn tại cạnh ngược kéo theo sự tồn tại chu trình.

$\Leftarrow)$  Ta phải chứng minh: Nếu không có cạnh ngược thì  $G$  là á chu trình. Ta chứng minh bằng lập luận phản đòn:  $G$  có chu trình  $\Rightarrow \exists$  cạnh ngược. Gọi  $v$  là đỉnh trên chu trình được thăm đầu tiên và  $u$  là đỉnh đi trước  $v$  trên chu trình. Khi  $v$  được thăm, các đỉnh khác trên chu trình đều là đỉnh trắng. Ta phải thăm được tất cả các đỉnh đạt được từ  $v$  trước khi quay trở lại từ DFS-Visit(). Vì thế cạnh  $u \rightarrow v$  được duyệt từ đỉnh  $u$  về tố tiên  $v$  của nó, do đó  $(u, v)$  là cạnh ngược.

Mệnh đề được chứng minh.

Như vậy, để kiểm tra xem đồ thị  $G$  có chứa chu trình hay không, ta chỉ cần thực hiện DFS( $G$ ). Nếu không phát hiện cạnh ngược thì đồ thị là phi chu trình, nếu trái lại đồ thị là chứa chu trình.

Độ phức tạp của thuật toán sẽ là độ phức tạp của DFS:  $O(|V|+|E|)$ .

**Chú ý:** Đối với đồ thị vô hướng, có thể chứng minh rằng thời gian tính của thuật toán kiểm tra xem đồ thị vô hướng có chứa chu trình hay không là  $O(|V|)$  dựa trên mệnh đề sau đây:

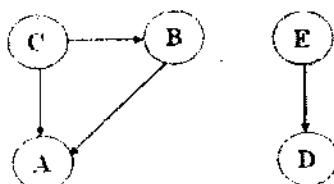
**Mệnh đề:** Đơn đồ thị vô hướng với  $n$  đỉnh và  $n$  cạnh chắc chắn chứa chu trình.

### Bài toán sắp xếp tông (Topological Sort)

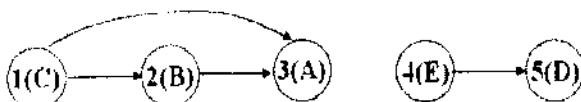
**Bài toán đặt ra là:** Cho đồ thị có hướng không có chu trình  $G = (V, E)$ . Hãy tìm cách sắp xếp các đỉnh sao cho nếu có cạnh  $(u,v)$  thì  $u$  phải đi trước  $v$  trong thứ tự đó (nói cách khác, cần tìm cách đánh số các đỉnh của đồ thị sao cho mỗi cung của đồ thị luôn hướng từ đỉnh có chỉ số nhỏ hơn đến đỉnh có chỉ số lớn hơn).

Bài toán sắp xếp tông có nhiều ứng dụng trong lý thuyết lập lịch.

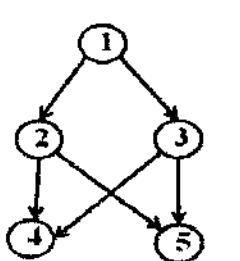
**Ví dụ:** Xét đồ thị:



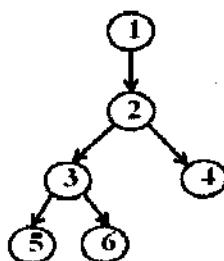
Một cách đánh số (sắp xếp tông) là:



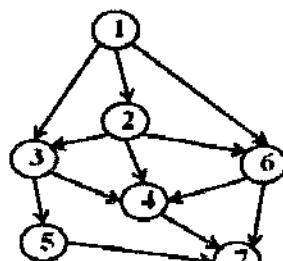
Tiếp theo là ba DAG và một số thứ tự sắp xếp tông:



- 1, 2, 3, 4, 5
- 1, 2, 3, 5, 4
- 1, 3, 2, 4, 5
- 1, 3, 2, 5, 4



- 1, 2, 4, 3, 5, 6
- 1, 2, 4, 3, 6, 5
- 1, 2, 3, 4, 5, 6
- 1, 2, 3, 4, 6, 5



- 1, 2, 3, 5, 6, 4, 7
- 1, 2, 3, 6, 4, 5, 7
- 1, 2, 3, 6, 4, 7, 5
- 1, 2, 6, 3, 4, 5, 7

...

...

Kết quả sau đây là cơ sở để phát triển thuật toán sắp xếp topô.

**Mệnh đề:** Nếu có cung  $(u, v)$  thì  $f[u] > f[v]$  trong DFS.

### Chứng minh:

Khi cung  $(u, v)$  được khảo sát thì  $u$  có màu xám. Khi đó  $v$  có thể có một trong ba màu: xám, trắng, đen.

- Nếu  $v$  có màu xám  $\Rightarrow (u, v)$  là cạnh ngược  $\Rightarrow$  Tồn tại chu trình?
- Nếu  $v$  có màu trắng  $\Rightarrow v$  trở thành con cháu của  $u \Rightarrow f[v] < f[u]$ .
- Nếu  $v$  có màu đen  $\Rightarrow v$  đã duyệt xong  $\Rightarrow f[v] < f[u]$ .

Mệnh đề được chứng minh.

Từ bồ đề suy ra, để sắp xếp topô, chỉ việc thực hiện DFS và thứ tự cần tìm chính là thứ tự các đỉnh được sắp xếp theo thứ tự giảm dần của  $f[v]$ . Ta đi đến thuật toán sau:

### Thuật toán sắp xếp topô

Thuật toán có thể mô tả ngắn tắt như sau: Thực hiện  $\text{DFS}(G)$ , khi mỗi đỉnh được duyệt xong, ta đưa nó vào đầu danh sách liên kết (điều đó có nghĩa là những đỉnh kết thúc thăm càng muộn thì sẽ càng ở gần đầu danh sách hơn). Danh sách liên kết thu được khi kết thúc  $\text{DFS}(G)$  sẽ cho ta thứ tự cần tìm.

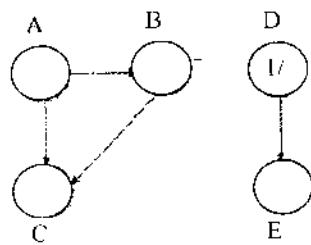
#### TopoSort(G)

1. **for**  $u \in V$   $\text{color}[u] = \text{white}$ ; // khởi tạo
2.  $L = \text{new(linked\_list)}$ ; // khởi tạo danh sách liên kết rỗng  $L$
3. **for**  $u \in V$
4. if ( $\text{color}[u] == \text{white}$ )  $\text{TopVisit}(u)$ ;
5. **return**  $L$ ; //  $L$  cho thứ tự cần tìm

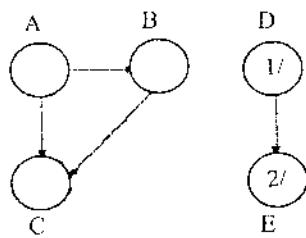
1. **TopVisit( $u$ ) {** // Bắt đầu tìm kiếm từ  $u$
2.  $\text{color}[u] = \text{gray}$ ; // Đánh dấu  $u$  là đã thăm
3. **for**  $v \in \text{Adj}(u)$
4. if ( $\text{color}[v] == \text{white}$ )  $\text{TopVisit}(v)$ ;
5. Nạp  $u$  vào đầu danh sách  $L$  //  $u$  đã duyệt xong

Thời gian tính của  $\text{TopoSort}(G)$  là  $O(|V| + |E|)$ .

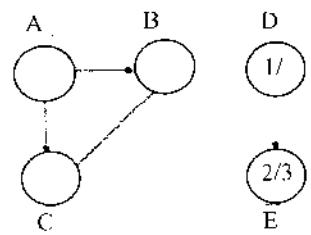
Ví dụ:



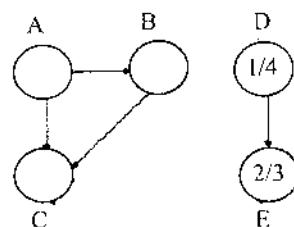
Linked list:  $\emptyset$



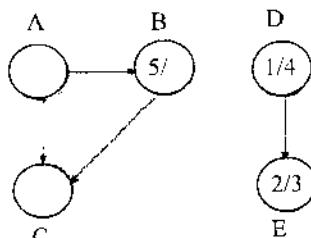
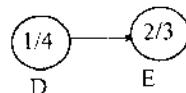
Linked list:  $\emptyset$



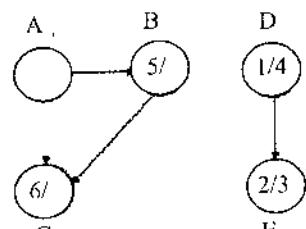
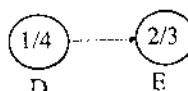
Linked List:



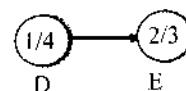
Linked List:

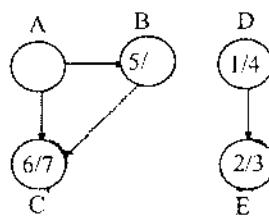


Linked List:

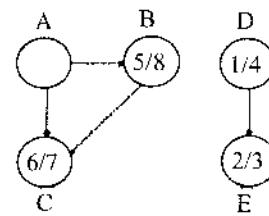
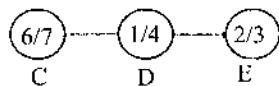


Linked List:

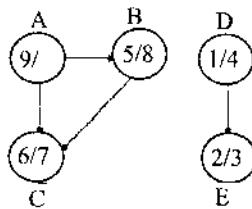
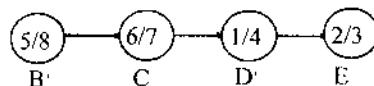




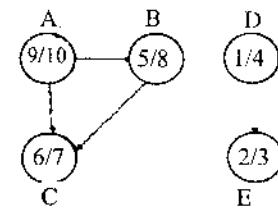
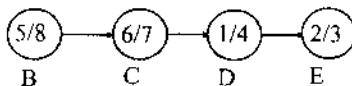
Linked List:



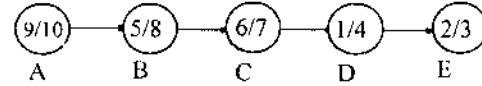
Linked List:



Linked List:

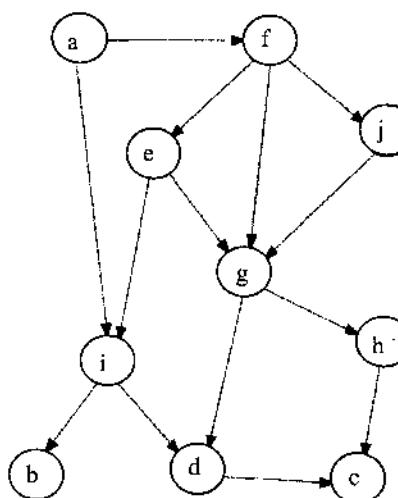


Linked List:

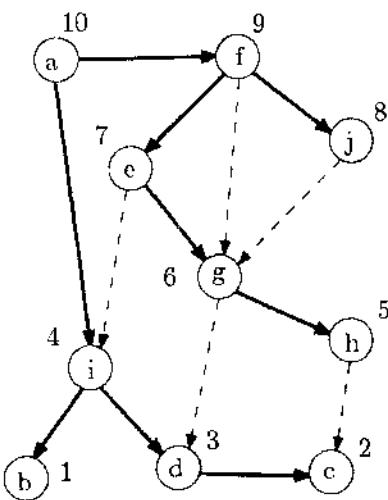


Kết thúc thuật toán ta thu được thứ tự cần tìm là 1(A), 2(B), 3(C), 4(D), 5(E).

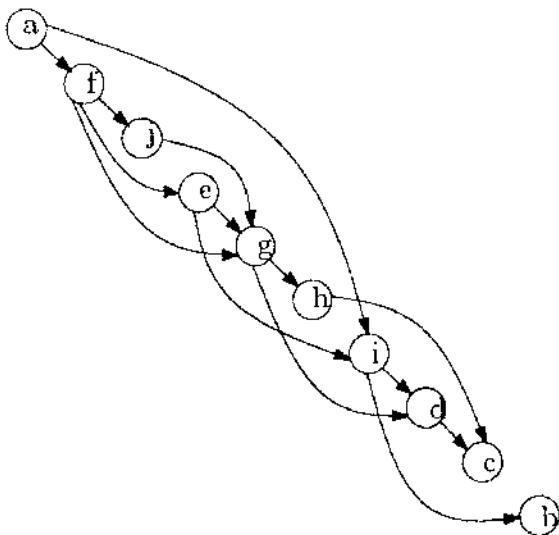
**Ví dụ:** Xét đồ thị:



Kết quả thực hiện DFS(G): Các số viết bên cạnh các đỉnh cho biết thứ tự kết thúc thăm của chúng:



Ta thu được thứ tự tópô cần tìm:



### Thuật toán xóa dần đỉnh

Một thuật toán khác để thực hiện sắp xếp tópô được xây dựng dựa trên mệnh đề sau:

**Mệnh đề:** Giả sử  $G$  là đồ thị có hướng không có chu trình. Khi đó:

- 1) Mọi đồ thị con  $H$  của  $G$  đều là đồ thị phi chu trình.
- 2) Bao giờ cũng tìm được đỉnh có bán bậc vào bằng 0.

### **Chứng minh:**

Khẳng định 1) là hiển nhiên.

Ta chứng minh khẳng định 2) bằng phản chứng. Giả sử không tìm được đỉnh có bán bậc vào bằng 0. Khi đó bắt đầu từ một đỉnh  $v$  tùy ý, do bán bậc vào của  $v$  khác 0 nên ta tìm được đỉnh  $u$  sao cho  $(u, v) \in E$ . Tiếp theo, do  $\deg(u) \neq 0$  nên phải tìm được  $w$  sao cho  $(w, u) \in E$ . Nếu  $w$  trùng với  $v$  thì ta thu được chu trình và điều đó mâu thuẫn với giả thiết  $G$  không có chu trình. Ngược lại, ta sẽ tiếp tục lập luận trên đối với  $w$  và đến được đỉnh mới  $z$ . Nếu  $z$  trùng với một trong số các đỉnh đã đi qua  $(v, u, w)$  thì ta thu được chu trình, còn ngược lại lập luận sẽ được tiếp tục đối với  $z$ . Do số đỉnh của đồ thị là hữu hạn nên dãy  $v, u, w, z, \dots$  không thể kéo dài vô hạn, đến một lúc nào đó ta sẽ gặp phải đỉnh mà trước đó đã đi qua, nghĩa là thu được chu trình và điều đó trái với giả thiết  $G$  là đồ thị phi chu trình.

Mệnh đề được chứng minh.

Từ mệnh đề ta suy ra thuật toán xóa dần đỉnh để thực hiện sắp xếp topô sau đây: Thoạt tiên, tìm các đỉnh có bán bậc vào bằng 0. Rõ ràng ta có thể đánh số chúng theo một thứ tự tùy ý bắt đầu từ 1. Tiếp theo, loại bỏ khỏi đồ thị những đỉnh đã được đánh số cùng các cung di ra khỏi chúng, ta thu được đồ thị mới cũng không có chu trình và thủ tục được lặp lại với đồ thị mới này. Quá trình đó sẽ được tiếp tục cho đến khi tất cả các đỉnh của đồ thị được đánh số.

### **TopoSort2(G)**

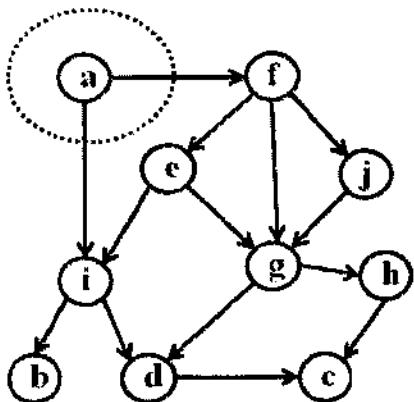
```

for $v \in V$ do $Vao[v] = 0$;
for $u \in V$ do // Tính $Vao[v] =$ bán bậc vào của v
 for $v \in Adj(u)$ do
 $Vao[v] = Vao[v] + 1$
 $Q = \emptyset$; // Khởi tạo hàng đợi rỗng Q
for $v \in V$ do
 if $Vao[v] = 0$ then Enqueue(Q, v)
 num = 0;
 while $Q \neq \emptyset$ do {
 $u = Dequeue(Q)$
 num = num + 1
 NR[u] = num
 for $v \in Adj(u)$ do {
 $Vao[v] = Vao[v] - 1$;
 if $Vao[v] = 0$ then Enqueue(Q, v)
 }
 }
}

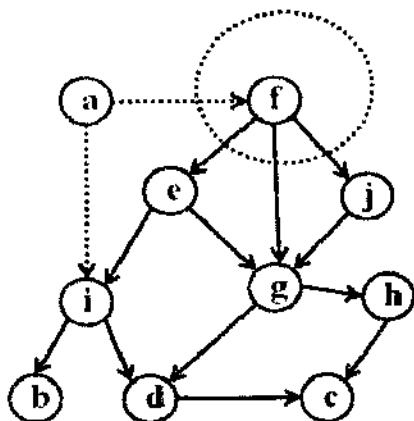
```

Kết thúc thuật toán NR[v] cho chỉ số của đỉnh v trong thứ tự sắp xếp tôpô cần tìm.

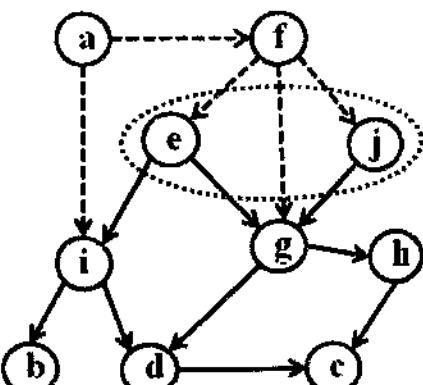
Ví dụ:



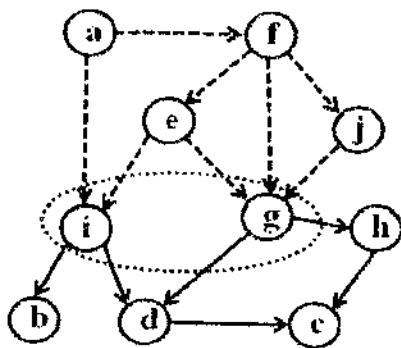
Đỉnh a có bán bậc vào bằng 0. Đánh số đỉnh a là 1. Xóa bỏ các cung (a,i), (a,f).



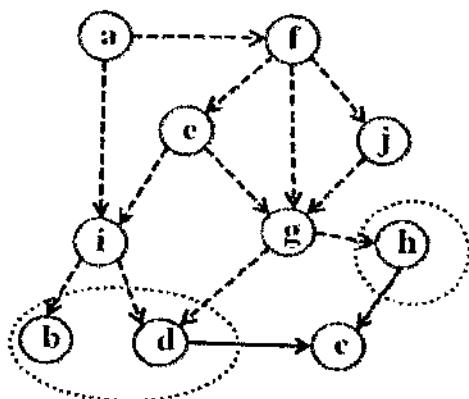
Đỉnh f có bán bậc vào bằng 0. Đánh số đỉnh f là 2. Xóa bỏ các cung (f,e), (f,g), (f,j).



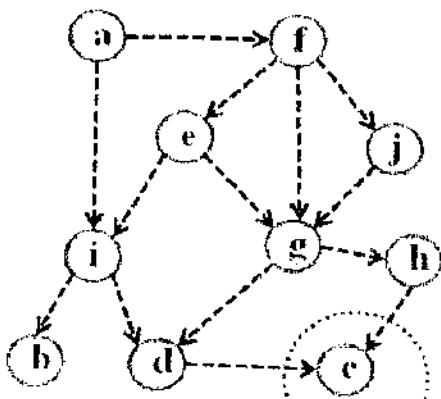
Đỉnh e và j có bán bậc vào bằng 0. Đánh số đỉnh e và j bởi 3 và 4 theo thứ tự tùy ý. Chẳng hạn đánh số j là 3 còn e là 4. Xóa bỏ các cung (e,i), (e,g), (j,g).



Dịnh i và g có bán bậc vào bằng 0.  
Đánh số đỉnh i là 5, g là 6. Xóa bỏ các cung (i,b), (i,d), (g,d), (g,h).



Dịnh b, d và h có bán bậc vào bằng 0. Đánh số đỉnh b là 6, d là 7 và h là 8. Xóa bỏ các cung (d,c), (b,c).



Đánh số đỉnh cuối cùng c là 9. Kết thúc.

#### 7.4.4. Bài toán tô màu đỉnh đồ thị

**Định nghĩa:** Cho đồ thị vô hướng  $G = (V,E)$  và số nguyên dương  $k$ . Ta nói đồ thị  $G$  có thể tô được bởi  $k$  màu nếu có thể tìm được cách gán cho mỗi đỉnh của đồ thị một trong  $k$  màu để cho sao cho không có hai đỉnh kề nhau nào bị tô bởi cùng một màu.

Số  $k$  nhỏ nhất để đồ thị  $G$  có thể tô được bởi  $k$  màu được gọi là *sắc số* của đồ thị và được ký hiệu là  $\chi(G)$ .

Bài toán đặt ra là: *Cho đồ thị vô hướng  $G$  và số nguyên dương  $k$ . Hỏi  $G$  có phải là tô được bởi  $k$  màu hay không?*

Hiện nay vẫn chưa có thuật toán hiệu quả để giải quyết bài toán khi  $k \geq 3$ . Tuy nhiên trong trường hợp  $k = 2$ , thuật toán hiệu quả để giải quyết có thể xây dựng dựa trên các thuật toán tìm kiếm trên đồ thị. (Bài toán trong trường hợp  $k = 2$  có thể phát biểu như bài toán nhận biết đồ thị hai phía: "Hỏi đồ thị vô hướng  $G$  có phải là đồ thị hai phía hay không?").

Ý tưởng của thuật toán có thể trình bày như sau: Không giâm tông quát, ta giả sử  $G$  là liên thông (trái lại, ta sẽ áp dụng thuật toán trình bày dưới đây với từng thành phần liên thông của  $G$ ). Chọn  $v$  là một đỉnh tùy ý, tô  $v$  bởi màu xanh. Tiếp đến ta tô màu tất cả các đỉnh kè của  $v$  bởi màu đỏ. Các đỉnh kè của các đỉnh màu đỏ lại được tô màu xanh,... Quá trình sẽ được tiếp tục cho đến khi gặp phải một trong hai tình huống sau đây:

1. Tất cả các đỉnh đều được tô màu. Khi đó ta thu được một cách tô màu  $G$  bởi hai màu. Thuật toán kết thúc.

2. Phát hiện ra một đỉnh đã tô màu xanh (đỏ) nay lại phải tô lại bởi màu đỏ (xanh). Khi đó ta kết luận đồ thị không thể tô bởi hai màu và kết thúc thuật toán.

Việc duyệt qua các đỉnh kè của một đỉnh có thể dễ dàng thực hiện nhờ các thuật toán BFS hoặc DFS. Do đó dễ dàng cài đặt thuật toán vừa mô tả dựa vào sơ đồ BFS hoặc DFS.

Ta xét cách cài đặt thuật toán dựa trên DFS. Ta thay đổi ý nghĩa của biến Color:

- Color[v] = 0 nếu đỉnh  $v$  chưa thăm;
- Color[v] = 1 nếu đỉnh  $v$  có màu xanh;
- Color[v] = 2 nếu  $v$  có màu đỏ.

Trong hàm DFS( $u$ ):

- Nếu Color[v] = 0, thì gán cho đỉnh  $v$  màu đối với màu của  $u$ , nghĩa là:

$$\text{Color}[v] = 3 - \text{Color}[u];$$

– Nếu đỉnh  $v$  đã được tô màu thì màu của nó phải là  $3 - \text{Color}[u]$ , tức là điều kiện  $\text{Color}[v] + \text{Color}[u] = 3$  phải được thực hiện, nếu ngược lại không có cách tô màu cần tìm.

Hàm Bi-Color( $u$ ) trả lại giá trị true nếu thành phần liên thông chứa đỉnh  $u$  tô được bởi hai màu và trả lại false nếu ngược lại:

**Bi-Color( $u$ )**

1. **for**  $v \in \text{Adj}[u]$
2.   **do if**  $\text{Color}[v] = 0$  **then** {
3.          $\text{Color}[v] \leftarrow 3 - \text{Color}[u]$

```

4. if !Bi-Color(v) then return false
5. }
6.
7. else
8. if Color [v] + Color[u] != 3 then return false
9. return true

```

**Thuật toán tô màu đồ thị  $G$  bởi hai màu**

**Bi-Color\_Graph( $G$ )**

```

1. for $u \in V[G]$
2. do color[u] = 0
3. for $u \in V[G]$
4. do if color[u] = 0
5. then if !Bi-Color(u) then return false // Đồ thị G không tô được
 bởi hai màu
6. return true // Đồ thị G tô được bởi 2 màu

```

#### 7.4.5. Bài toán xây dựng bao đóng truyền ứng của đồ thị

**Định nghĩa:** Bao đóng truyền ứng của đồ thị có hướng  $G = (V,E)$  là đồ thị có hướng  $G^* = (V,E^*)$  với tập đỉnh là tập đỉnh của đồ thị  $G$  và tập cạnh

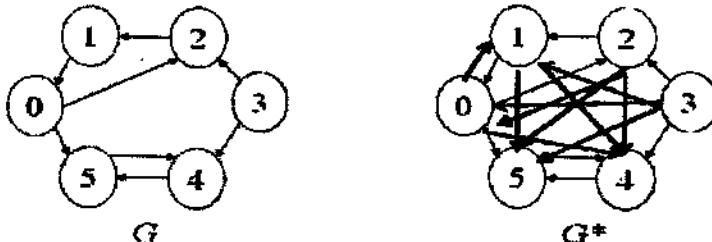
$$E^* = \{(u,v) | \text{có đường đi từ } u \text{ đến } v \text{ trên } G\}.$$

**Bài toán:** Cho đồ thị có hướng  $G$ , tìm bao đóng truyền ứng  $G^*$ .

Bài toán xây dựng bao đóng truyền ứng có ứng dụng quan trọng trong lý thuyết cơ sở dữ liệu quan hệ.

**Ví dụ:**

Hình vẽ dưới đây cho đồ thị  $G$  và bao đóng truyền ứng của nó  $G^*$ .



Nếu đồ thị  $G$  cho bởi ma trận kè thì để giải quyết bài toán đặt ra, ta sẽ đi xây dựng ma trận kè của  $G^*$ . Thuật toán nổi tiếng để giải quyết bài toán xây dựng bao đóng truyền ứng là thuật toán Warshall sau đây:

### Thuật toán Warshall

//  $n = |V|$ , Các đỉnh đánh số từ 0 đến  $n - 1$

for ( $i = 0; i < n; i++$ )

    for ( $s = 0; s < n; s++$ )

        for ( $t = 0; t < n; t++$ )

            if ( $A[s][i] \&& A[i][t]$ )

$A[s][t] = 1;$

**Mệnh đề:** Thuật toán tìm được bao đóng truyền ứng với thời gian  $O(|V|^3)$ .

**Chứng minh:** Ta chứng minh thuật toán tìm được bao đóng truyền ứng bằng quy nạp theo bước lặp  $i$ .

Lần lặp 1: Ma trận có 1 ở vị trí  $(s,t)$  khi và chỉ khi có đường đi  $s-t$  hoặc  $s=0=t$ .

Lần lặp thứ  $i$ : Gán phần tử ở vị trí  $(s, t)$  giá trị 1 khi và chỉ khi có đường đi từ  $s$  đến  $t$  trong đồ thị không chứa đỉnh với chỉ số lớn hơn  $i$  (ngoại trừ hai mút).

Khi đó ở lần lặp thứ  $i + 1$ :

– Nếu có đường đi từ  $s$  đến  $t$  không chứa đỉnh có chỉ số lớn hơn  $i - A[s,t]$  đã có giá trị 1.

– Nếu có đường đi từ  $s$  đến  $i + 1$  và đường đi từ  $i + 1$  đến  $t$  và cả hai đều không chứa đỉnh với chỉ số lớn hơn  $i$  (ngoại trừ hai mút) thì  $A[s,t]$  được gán giá trị 1.

Rõ ràng thời gian tính của thuật toán là  $O(|V|^3)$ .

Ta có thể cải tiến thuật toán bằng cách bổ sung thêm câu lệnh if trước vòng lặp for trong cùng và đi đến thuật toán sau đây:

### Thuật toán Warshall cải tiến

//  $n = |V|$ , Các đỉnh đánh số từ 0 đến  $n - 1$

for ( $i = 0; i < n; i++$ )

    for ( $s = 0; s < n; s++$ )

        if  $A[s][i]$

            for ( $t = 0; t < n; t++$ )

                if  $A[i][t]$

$A[s][t] = 1;$

Cải tiến này chỉ có tác dụng tăng hiệu quả thực tế của thuật toán mà không thay đổi được đánh giá thời gian tính trong tình huống tồi nhất của thuật toán.

### Áp dụng DFS tìm bao đóng truyền ứng

Do việc xây dựng tập cạnh  $E$  của bao đóng truyền ứng có thể dẫn về bài toán tìm đường đi từ một đỉnh  $u$  bất kỳ đến tất cả các đỉnh còn lại, nên ta có kết quả sau:

**Mệnh đề:** Sử dụng DFS ta có thể xác định bao đóng truyền ứng sau thời gian  $O(|V|^*(|E|+|V|))$ .

## Chứng minh

DFS cho phép xác định tất cả các đỉnh đạt đến được từ một đỉnh cho trước  $u$  sau thời gian  $O(|E|+|V|)$  nếu ta sử dụng biểu diễn đồ thị bởi danh sách kè.

Do đó, để xác định bao đóng truyền ứng ta thực hiện DFS với mỗi  $u \in V$  ( $|V|$  lần).

Thời gian tính của thuật toán rõ ràng là  $O(|V|^*(|E|+|V|))$ .

## Kinh nghiệm tính toán

Để thấy được hiệu quả của các thuật toán đề xuất, ta tiến hành thực nghiệm đối với hai loại đồ thị:

- Đồ thị thưa:  $|E| = 10|V|$ , số đỉnh sẽ lần lượt là 25, 50, 125, 250, 500.
- Đồ thị dày 250 đỉnh với số cạnh lần lượt là 5000, 10000, 25000, 50000, 100000.

Bảng sau đây thống kê thời gian tính của các thuật toán vừa đề xuất trong hai tình huống nêu trên:

| Đồ thị thưa ( $ E  = 10 V $ ) |      |      |      |    | Đồ thị dày (250 đỉnh) |     |     |     |     |
|-------------------------------|------|------|------|----|-----------------------|-----|-----|-----|-----|
| V                             | W    | W*   | A    | L  | E                     | W   | W*  | A   | L   |
| 25                            | 0    | 0    | 1    | 0  | 5000                  | 289 | 203 | 177 | 23  |
| 50                            | 3    | 1    | 2    | 1  | 10000                 | 300 | 214 | 184 | 38  |
| 125                           | 35   | 24   | 23   | 4  | 25000                 | 309 | 226 | 200 | 97  |
| 250                           | 275  | 181  | 178  | 13 | 50000                 | 315 | 232 | 218 | 337 |
| 500                           | 2222 | 1438 | 1481 | 54 | 100000                | 326 | 246 | 235 | 784 |

Trong đó:

W : Thuật toán Warshall;

W\* : Thuật toán Warshall cải tiến;

A : DFS sử dụng ma trận kè;

L : DFS sử dụng danh sách kè.

Kết quả tính toán trên cho thấy:

- Thuật toán Warshall cải tiến có thời gian tính giảm rõ rệt so với chưa cải tiến;
- Đối với đồ thị thưa, thuật toán DFS sử dụng danh sách kè hiệu quả hơn hẳn các thuật toán khác;
- Đối với đồ thị dày, thuật toán DFS sử dụng ma trận kè là hiệu quả nhất.

## 7.5. BÀI TOÁN CÂY KHUNG NHỎ NHẤT

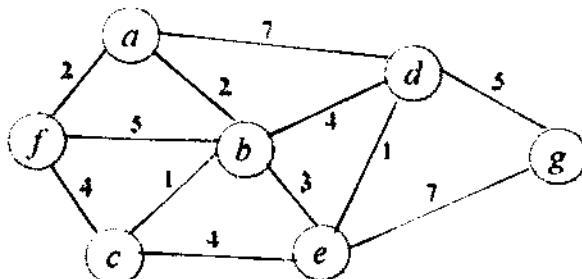
Giả sử  $G = (V, E)$  là đồ thị vô hướng liên thông có trọng số trên cạnh  $c(e)$ ,  $e \in E$ .

**Định nghĩa:** Cây  $T = (V, E_T)$  với  $E_T \subseteq E$ , được gọi là cây khung của  $G$ . Độ dài của cây khung  $T$  là tổng trọng số trên các cạnh của nó:

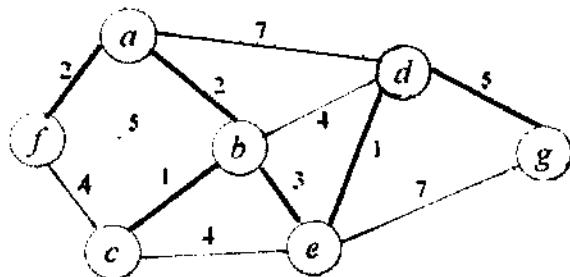
$$c(T) = \sum_{e \in E_T} c(e).$$

Bài toán đặt ra là tìm cây khung có độ dài nhỏ nhất.

**Ví dụ:** Xét đồ thị  $G$ :



Cây khung nhỏ nhất được thể hiện bởi các cạnh đậm có độ dài 14:



### 7.5.1. Thuật toán Kruskal

Thuật toán sẽ xây dựng tập cạnh  $E_T$  của cây khung nhỏ nhất  $T = (V, E_T)$  theo từng bước. Trước hết sắp xếp các cạnh của đồ thị  $G$  theo thứ tự không giảm của độ dài. Bắt đầu từ tập  $E_T = \emptyset$ , ở mỗi bước ta sẽ lần lượt duyệt trong danh sách cạnh đã sắp xếp, từ cạnh có độ dài nhỏ đến cạnh có độ dài lớn hơn, để tìm ra cạnh mà việc bồi sung nó vào tập  $E_T$  không tạo thành chu trình trong tập này. Thuật toán sẽ kết thúc khi ta thu được tập  $E_T$  gồm  $n - 1$  cạnh. Cụ thể, thuật toán có thể mô tả như sau:



Joseph Bernard Kruskal, Jr.

(1928 – 2010)

### Kruskal\_Algorithm

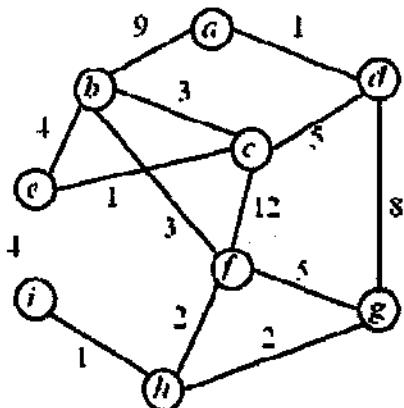
```

 $E_T := \emptyset;$
while $|E_T| < (n-1)$ and ($E \neq \emptyset$) do
{
 Chọn e là cạnh có độ dài nhỏ nhất trong E;
 $E := E \setminus \{e\};$
 if ($E_T \cup \{e\}$ không chứa chu trình) then $E_T := E_T \cup \{e\};$
}
if ($|E_T| < n-1$) then Đồ thị không liên thông;

```

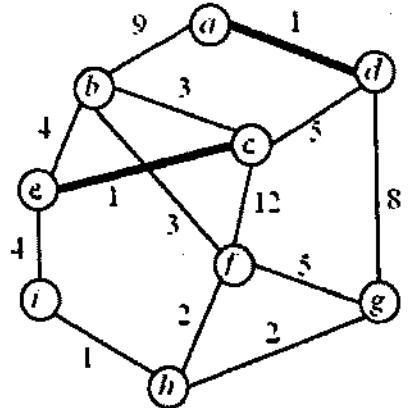
**Ví dụ:**

Tìm cây khung nhỏ nhất cho đồ thị sau đây:



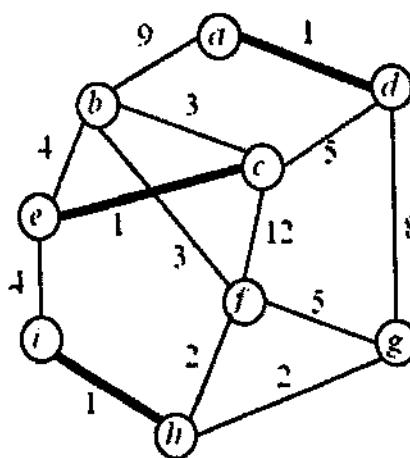
Quá trình thực hiện thuật toán được mô tả trong bảng sau đây:

|  |                                       |
|--|---------------------------------------|
|  | Xét cạnh (a,d).<br>$E_T = \{(a,d)\}.$ |
|--|---------------------------------------|



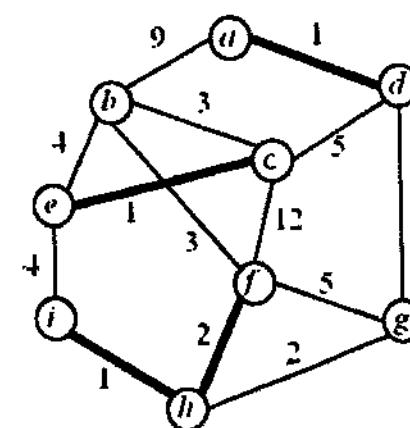
Xét cạnh (e,c).

$$E_T = \{(a,d), (e,c)\}.$$



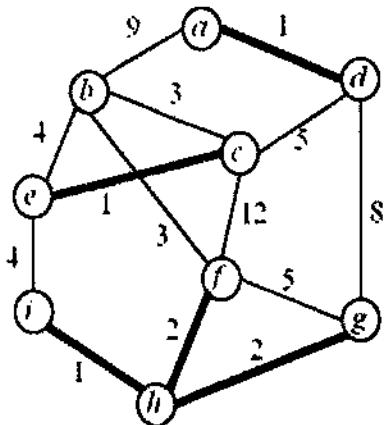
Xét cạnh (i,h).

$$E_T = \{(a,d), (e,c), (i,h)\}.$$



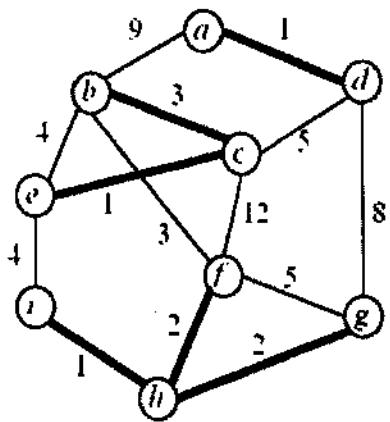
Xét cạnh (h,f).

$$E_T = \{(a,d), (e,c), (i,h), (h,f)\}.$$



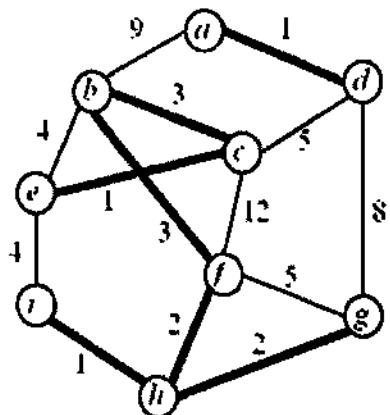
Xét cạnh (h,g).

$$E_T = \{(a,d), (e,c), (i,h), (h,f), (h,g)\}.$$



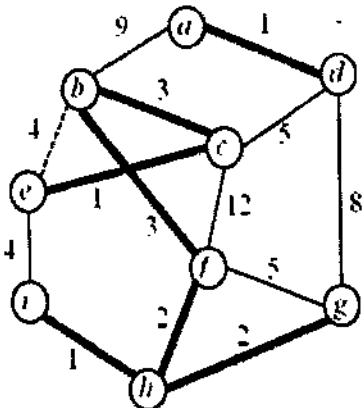
Xét cạnh (b,c).

$$E_T = \{(a,d), (e,c), (i,h), (h,f), (h,g), (b,c)\}.$$



Xét cạnh (b,f).

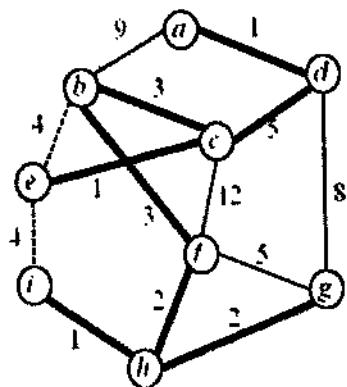
$$E_T = \{(a,d), (e,c), (i,h), (h,f), (h,g), (b,c), (b,f)\}.$$



Xét cạnh (b,e).

Cạnh (b,e) không được chọn.

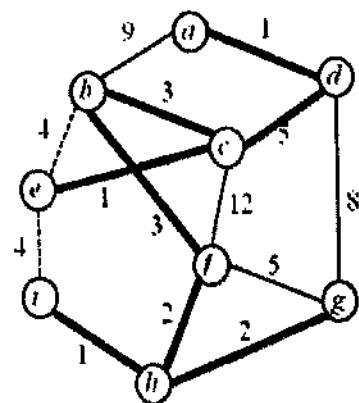
$$E_T = \{(a,d), (e,c), (i,h), (h,f), (h,g), (b,c), (b,f)\}.$$



Xét cạnh (i,e).

Cạnh (i,e) không được chọn.

$$E_T = \{(a,d), (e,c), (i,h), (h,f), (h,g), (b,c), (b,f)\}.$$



Xét cạnh (c,d).

$$E_T = \{(a,d), (e,c), (i,h), (h,f), (h,g), (b,c), (b,f), (c,d)\}.$$

Thuật toán kết thúc.

Tập cạnh của cây khung nhỏ nhất:

$$E_T = \{(a,d), (e,c), (i,h), (h,f), (h,g), (b,c), (b,f), (c,d)\}$$

Độ dài của cây khung nhỏ nhất:

$$c(T) = 1 + 1 + 1 + 2 + 2 + 3 + 5 = 15$$

### 7.5.2. Cấu trúc dữ liệu biểu diễn phân hoạch

Trong thuật toán, bước đòi hỏi khôi lượng tính toán lớn là bước sắp xếp các cạnh của đồ thị theo thứ tự không giảm của độ dài. Đồi với đồ thị có  $m$  cạnh, bước này đòi hỏi thời gian  $O(m \log m)$ . Tuy nhiên, để xây dựng cây khung nhỏ nhất với  $n - 1$  cạnh, nói chung ta không cần phải sắp thứ tự toàn bộ các cạnh mà chỉ cần xét phần trên của dãy đó chứa  $p < m$  cạnh. Do đó thay vì sắp xếp toàn bộ dãy cạnh ta sẽ sử dụng heap-min. Theo cách làm này, để tạo đồ thị đầu tiên ta mất thời gian  $O(m)$ , việc vun lại đồ thị sau khi lấy ra phần tử ở gốc đòi hỏi thời gian  $O(\log m)$ . Vì vậy, với cài tiến này, thuật toán sẽ đòi hỏi thời gian  $O(m + p \log m)$  cho việc sắp xếp các cạnh. Trong việc giải các bài toán thực tế, số  $p$  thường nhỏ hơn rất nhiều so với  $m$ .

Vấn đề thứ hai trong việc thể hiện thuật toán Kruskal là việc lựa chọn cạnh để bổ sung đòi hỏi phải có một thủ tục kiểm tra tập cạnh  $E_T \cup \{e\}$  có chứa chu trình hay không. Để ý rằng, các cạnh trong  $E_T$  ở các bước lặp trung gian sẽ tạo thành một rừng. Cạnh  $e$  cần khảo sát sẽ tạo thành chu trình với các cạnh trong  $E_T$  khi và chỉ khi cả hai đỉnh đầu của nó thuộc vào cùng một cây con của rừng nói trên. Do đó, nếu cạnh  $e$  không tạo thành chu trình với các cạnh trong  $E_T$ , thì nó phải nối hai cây khác nhau trong  $E_T$ . Vì thế, để kiểm tra xem có thể bổ sung cạnh  $e$  vào  $E_T$  hay không, ta chỉ cần kiểm tra xem nó có nối hai cây khác nhau trong  $T$  hay không. Một trong các phương pháp hiệu quả để thực hiện việc kiểm tra này là ta sẽ phân hoạch tập các đỉnh của đồ thị ra thành các tập con không giao nhau, mỗi tập xác định bởi một cây con trong  $T$  (được hình thành ở các bước do việc bổ sung cạnh vào  $T$ ).

Như vậy, để giải quyết vấn đề thứ hai này, ta phải xây dựng hai thủ tục: Kiểm tra xem hai đỉnh  $u, v$  của cạnh  $e = (u, v)$  có thuộc vào hai tập con khác nhau hay không và trong trường hợp câu trả lời là khẳng định, nối hai tập con tương ứng thành một tập. Cấu trúc dữ liệu biểu diễn các tập không giao nhau có thể sử dụng để đáp ứng yêu cầu này.

#### Cấu trúc dữ liệu mô tả các tập không giao nhau

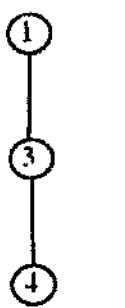
##### (Disjoint Set Data Structures)

Cho tập  $V = \{1, 2, \dots, n\}$ . Chúng ta cần xây dựng cấu trúc dữ liệu mô tả các tập con trong phân hoạch của tập  $V$  và các phép toán cơ bản sau đây đối với cấu trúc dữ liệu:

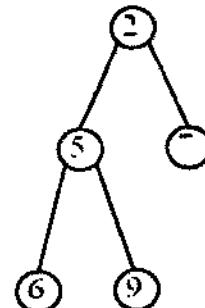
- Makeset( $x$ ) – tạo một tập con chứa duy nhất phần tử  $x$ .
- Union( $x, y$ ) – thay hai tập chứa  $x$  và  $y$  bởi tập là hợp của chúng.
- Find( $x$ ) – trả lại tên của tập con chứa  $x$ . Để đặt tên cho các tập con, ta có thể sử dụng một phần tử nào đó của nó.

Trước hết để biểu diễn mỗi tập con  $X \subset V$ , chúng ta sẽ sử dụng cấu trúc cây có gốc: chọn một phần tử nào đó của  $X$  làm gốc (tên của tập con  $X$  chính là phần tử tương ứng với gốc), mỗi phần tử  $x \in X$  sẽ có một biến trỏ  $\text{parent}[x]$  trỏ đến cha của nó, nếu  $x$  là gốc thì  $\text{parent}[x] = x$ .

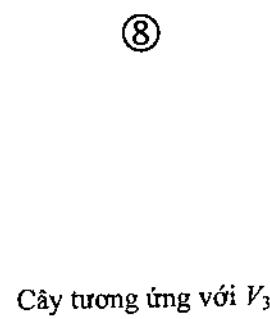
**Ví dụ:** Giả sử có  $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ ,  $V_1 = \{1, 3, 4\}$ ,  $V_2 = \{2, 5, 6, 7, 9\}$ ,  $V_3 = \{8\}$ . Ta có ba cây mô tả ba tập  $V_1$ ,  $V_2$ ,  $V_3$ .



Cây tương ứng với  $V_1$



Cây tương ứng với  $V_2$



Cây tương ứng với  $V_3$

Mảng parent để biểu diễn rỗng gồm ba cây tương ứng với  $V_1$ ,  $V_2$ ,  $V_3$ :

| i      | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|--------|---|---|---|---|---|---|---|---|---|
| parent | 1 | 2 | 1 | 3 | 2 | 5 | 2 | 8 | 5 |

Như vậy, thủ tục MakeSet(x) có thể viết như sau:

```

procedure MakeSet(x)
begin
parent[x] := x;
end;

```

Để tìm tên của tập con chứa phần tử x, ta dùng hàm sau:

```

function Find(x);
begin
 while x ≠ parent[x] do x = parent[x];
 Find := x;
end;

```

Để nối tập con chứa x và tập con chứa y, chúng ta có thể chia lại biến trỏ của gốc của cây chứa x để nó trỏ đến gốc của cây con chứa y. Điều đó được thực hiện nhờ thủ tục sau:

```

procedure Union(x, y);
begin
 u := Find(x); (* Tìm u là gốc của cây con chứa x *)
 v := Find(y); (* Tìm v là gốc của cây con chứa y *)
 parent[u] := v;
end;

```

Có thể thấy thời gian tính của hàm Find(x) phụ thuộc vào độ cao của cây chứa x. Trong trường hợp cây có  $k$  đỉnh và được biểu diễn như một đường đi (xem cây  $V_1$ ) thì độ cao của cây sẽ là  $k - 1$ . Từ đó suy ra hàm Find(x) có đánh giá thời gian tính là  $O(n)$ .

Liệu có cách nào để giảm độ cao của các cây con? Có một cách thực hiện rất đơn giản: khi nối hai cây chúng ta sẽ điều chỉnh con trỏ của gốc của cây con có ít đỉnh hơn, chứ không thực hiện việc nối một cách tùy tiện. Để ghi nhận số phần tử của một cây, chúng ta sẽ sử dụng thêm biến Num[v] chứa số phần tử của cây con với gốc tại v. Khi đó cần sửa lại thủ tục MakeSet(x) và Union(x, y) như sau:

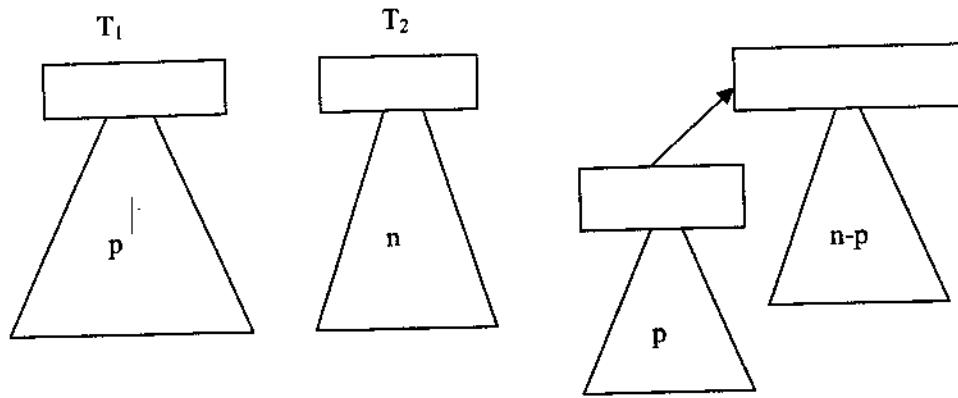
```
procedure MAKESET(x)
begin
 parent[x] := x; Num[x]:=1;
end;

procedure Union(x, y);
begin
 u:= Find(x); (* Tìm u là gốc của cây con chứa x *)
 v:= Find(y); (* Tìm v gốc của cây con chứa y *)
 if Num[u] <= Num[v] then
 begin
 parent[u] := v; Num[u]:= Num[u]+Num[v];
 end
 else
 begin
 parent[v] := u;
 Num[v]:= Num[u]+Num[v];
 end
 end;
end;
```

Ta có kết quả sau đây.

**Bố đề 2:** Giả sử quá trình thực hiện nối cây bắt đầu từ các cây chỉ có một đỉnh. Khi đó độ cao của các cây xuất hiện khi thực hiện thủ tục nối không vượt quá  $\log n$ .

**Chứng minh:** Ta chứng minh bằng quy nạp theo  $n$  là số đỉnh của cây. Rõ ràng bố đề là đúng cho  $n = 1$ . Giả sử bố đề là đúng cho cây với số đỉnh ít hơn  $n$ . Xét cây  $T$  có  $n$  đỉnh. Do cây  $T$  được gộp từ hai cây với số đỉnh ít hơn  $n$ , nên nếu gọi  $T_1$  và  $T_2$  là hai cây với số đỉnh tương ứng là  $m$  và  $n - p$  gộp thành cây  $T$ , thì ta có  $p \leq n/2$ .



Từ giả thiết quy nạp, ta suy ra cây  $T$  có độ cao là:

$$\begin{aligned} & \max \{\text{dept}(T_1) + 1, \text{depth}(T_2)\} \\ & \leq \max \{[\log m] + 2, [\log(n-m)] + 1\} \\ & \leq \max \{[\log m] + 2, [\log n] + 1\} \\ & = [\log n] + 1. \end{aligned}$$

Bỏ đê được chứng minh.

Từ bỏ đê suy ra các thao tác Find và Union được thực hiện với thời gian  $O(\log n)$  nhờ sử dụng cách nối cây trình bày sau cùng.

Thuật toán Kruskal với cấu trúc dữ liệu vừa trình bày có thể mô tả như sau:

### Kruskal\_UF

// Khởi tạo

1.  $E_T \leftarrow \emptyset$

2. **for**  $v \in V$  **do**

3.        Make-Set( $v$ );

4. Sắp xếp các cạnh trong  $E$  theo thứ tự không giảm của trọng số

// Vòng lặp

5. **for**  $(u,v) \in E$  **do**

6.        **if**  $\text{Find}(u) \neq \text{Find}(v)$

7.        **then**  $E_T \leftarrow E_T \cup \{(u,v)\};$

8.        Union( $u,v$ );

9. **if** ( $|E_T| < n-1$ )

**then** *Đồ thị không liên thông;*

**else**  $E_T$  là tập cạnh của cây khung nhỏ nhất;

### **Phân tích thời gian tính của thuật toán Kruskal\_UF**

- Dòng 1 – 3:  $O(|V|)$ .
- Dòng 4 (sắp xếp danh sách cạnh):  $O(|E| \log |E|)$ .
- Dòng 6 – 8 (các thao tác với phân hoạch phải lặp lại không quá  $|E|$  lần):  $O(|E| \log |E|)$ .

Tổng cộng:  $O(|E| \log |E|)$ .

Đối với đồ thị thưa, thuật toán nên sử dụng cách cài đặt vừa trình bày để giải bài toán cây khung nhô nhất.

## **7.6. BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT**

Cho đồ thị có trọng số trên cạnh. Trọng số trên cạnh của đồ thị có thể là chi phí để khảo sát cạnh đó. Chẳng hạn, nếu đồ thị biểu diễn sơ đồ giao thông (các đỉnh tương ứng với các nút giao thông, các cạnh tương ứng với các đoạn đường phố nối các cặp nút giao thông), thì trọng số trên cạnh có thể là khoảng cách, chi phí đi lại hoặc thời gian đi lại giữa hai nút giao thông.

Đối với đồ thị như vậy, chúng ta thường phải trả lời cho các câu hỏi như:

- Đi từ A đến B theo đường nào là nhanh nhất?
- Đi từ A đến B theo đường nào là rẻ nhất?
- Đi từ A đến B theo đường nào là ngắn nhất?

Mỗi câu hỏi như vậy là đầu vào cho cùng một bài toán: Bài toán đường đi ngắn nhất (shortest path problem).

Để phát biểu một cách chính xác hơn bài toán đường đi ngắn nhất, ta cần đưa vào một số khái niệm.

**Định nghĩa:** Cho đồ thị có hướng  $G = (V, E)$  với trọng số trên cạnh  $c(e)$ ,  $e \in E$ . Giả sử  $s, t \in V$  và  $P(s, t)$  là đường đi từ  $s$  đến  $t$  trên đồ thị:

$$P(s, t) : s = v_0, v_1, \dots, v_{k-1}, v_k = t.$$

Ta gọi độ dài của đường đi  $P(s, t)$  là tổng trọng số trên các cung của nó, tức là nếu ký hiệu  $\rho(P(s, t))$ , thì theo định nghĩa:

$$\rho(P(s, t)) = \sum_{e \in P(s, t)} c(e) = \sum_{i=0}^{k-1} c(v_i, v_{i+1}).$$

Ta gọi đường đi ngắn nhất từ  $s$  đến  $t$  là đường đi có độ dài nhỏ nhất trong số tất cả các đường đi từ  $s$  đến  $t$  trên đồ thị.

Người ta thường sử dụng ký hiệu  $\delta(s, t)$  để chỉ độ dài của đường đi ngắn nhất từ  $s$  đến  $t$  và gọi nó là khoảng cách từ  $s$  đến  $t$ .

Có ba dạng bài toán đường đi ngắn nhất cơ bản.

Bài toán 1: Tìm đường đi ngắn nhất giữa hai đỉnh cho trước.

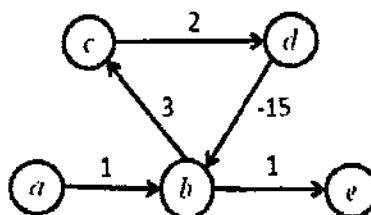
Bài toán 2: Tìm đường đi ngắn nhất từ một đỉnh nguồn  $s$  đến tất cả các đỉnh còn lại.

Bài toán 3: Tìm đường đi ngắn nhất giữa hai đỉnh bất kỳ.

Trước hết, dễ thấy các bài toán được dẫn ra theo thứ tự từ đơn giản đến phức tạp hơn. Một khía cạnh khác cũng có thể thấy ngay là nếu ta có thuật toán để giải một trong ba bài toán thì thuật toán đó cũng có thể sử dụng để giải hai bài toán còn lại.

Đường đi ngắn nhất giữa hai đỉnh nào đó có thể không tồn tại. Chẳng hạn, nếu không có đường đi từ  $s$  đến  $t$ , thì rõ ràng cũng không có đường đi ngắn nhất từ  $s$  đến  $t$ . Ngoài ra, nếu đồ thị chứa cạnh có trọng số âm thì có thể xảy ra tình huống: độ dài đường đi giữa hai đỉnh nào đó có thể làm nhỏ tùy ý. Ta xét ví dụ sau đây.

Ví dụ: Xét đồ thị có trọng số.



Xét đường đi từ  $a$  đến  $b$ :

$$a, k(b, c, d, b), e.$$

Nghĩa là ta đi  $k$  lần vòng theo chu trình

$$C: b, c, d, b$$

trước khi đến  $e$ . Độ dài của đường đi này bằng:

$$c(a, b) + k \rho(C) + c(b, e) = 2 - 10k \rightarrow -\infty, \text{ khi } k \rightarrow +\infty.$$

Tổng quát, nếu trên đường đi từ  $s$  đến  $t$  ta có thể đi vòng quanh một chu trình có độ dài âm (chu trình như vậy được gọi là chu trình âm) thì độ dài của đường đi từ  $s$  đến  $t$  có thể làm nhỏ tùy ý. Trong trường hợp này đường đi ngắn nhất từ  $s$  đến  $t$  là không tồn tại.

Dễ thấy rằng, nếu như trọng số trên các cạnh là các số không âm:  $c(e) \geq 0, e \in E$ , thì không tồn tại chu trình âm và nếu có đường đi từ  $s$  đến  $t$  thì luôn có đường đi ngắn nhất từ  $s$  đến  $t$ .

Trong mục tiếp theo chúng ta sẽ trình bày thuật toán Dijkstra để giải bài toán 2.

### 7.6.1. Thuật toán Dijkstra

Thuật toán Dijkstra được đề xuất để giải bài toán: Tìm đường đi ngắn nhất từ một đỉnh nguồn  $s$  đến tất cả các đỉnh còn lại trên đồ thị với trọng số không âm trên cạnh.

Trong quá trình thực hiện thuật toán, với mỗi đỉnh  $v$  ta sẽ lưu trữ nhãn của đỉnh gồm các thông tin sau:

- $k[v]$ : biến bun có giá trị đúng nếu ta đã tìm được đường đi ngắn nhất từ  $s$  đến  $v$ , ban đầu biến này được khởi tạo giá trị false.

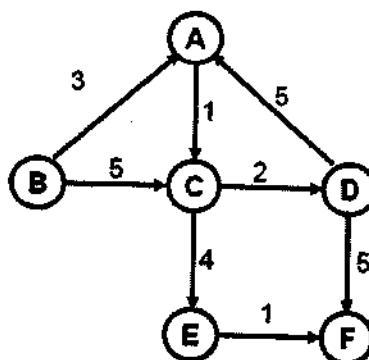
- $d[v]$ : khoảng cách ngắn nhất hiện biết từ  $s$  đến  $v$ . Ban đầu biến này được khởi tạo giá trị  $+\infty$  đối với mọi đỉnh, ngoại trừ  $d[s]$  được đặt bằng 0.

- $p[v]$ : là đỉnh đi trước đỉnh  $v$  trong đường đi có độ dài  $d[v]$ . Ban đầu, các biến này được khởi tạo rỗng (chưa biết).

Thuật toán lặp lại các thao tác sau đây cho đến khi tất cả các đỉnh được khảo sát xong (nghĩa là  $k[v] = \text{true}$  với mọi  $v$ ):

1. Trong tập đỉnh với  $k[v] = \text{false}$ , chọn đỉnh  $v$  có  $d[v]$  là nhỏ nhất.
2. Đặt  $k[v] = \text{true}$ .
3. Với mỗi đỉnh  $w$  kề với  $v$  và có  $k[w] = \text{false}$ , ta kiểm tra  $d[w] > d[v] + c(v, w)$ . Nếu đúng thì đặt lại  $d[w] = d[v] + c(v, w)$  và đặt  $p[w] = v$ .

**Ví dụ:** Tìm đường đi ngắn nhất từ đỉnh B đến tất cả các đỉnh còn lại trên đồ thị sau:



Bảng sau đây cho thông tin về quá trình thực hiện thuật toán. Chú ý là: khi nhãn của đỉnh không còn thay đổi, trong bảng sẽ bỏ qua không chép lại để dễ theo dõi.



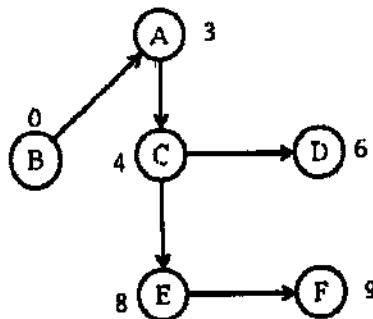
Edsger W. Dijkstra

1930 – 2002

Giải thưởng Turing, 1972

| Bước lặp | A        |   |   | B |   |   | C        |   |   | D        |   |   | E        |   |   | F        |   |   |
|----------|----------|---|---|---|---|---|----------|---|---|----------|---|---|----------|---|---|----------|---|---|
|          | d        | p | k | d | p | k | d        | p | k | d        | p | k | d        | p | k | d        | p | k |
| Khởi tạo | $\infty$ | - | F | 0 | - | F | $\infty$ | - | F |
| 1        | 3        | B | F | 0 | - | T | 5        | B | F | $\infty$ | - | F | $\infty$ | - | F | $\infty$ | - | F |
| 2        | 3        | B | T |   |   |   | 4        | A | F | $\infty$ | - | F | $\infty$ | - | F | $\infty$ | - | F |
| 3        |          |   |   |   |   |   | 4        | A | T | 6        | C | F | 8        | C | F | $\infty$ | - | F |
| 4        |          |   |   |   |   |   |          |   |   | 6        | C | T | 8        | C | F | 11       | D | F |
| 5        |          |   |   |   |   |   |          |   |   |          |   |   | 8        | C | T | 11       | D | F |
| 6        |          |   |   |   |   |   |          |   |   |          |   |   |          |   |   | 9        | E | T |

Tập các cạnh  $\{(p[v], v) : v \in V - \{B\}\}$  tạo thành một cây được gọi là *cây đường đi ngắn nhất* từ đỉnh B đến tất cả các đỉnh còn lại. Cây này được cho trong hình vẽ sau:



Số viết bên cạnh mỗi đỉnh v là độ dài đường đi ngắn nhất từ đỉnh B đến nó ( $d[v]$ ).

**Chú ý:** Thuật toán Dijkstra có thể làm việc với cả đồ thị vô hướng lẫn có hướng.

### 7.6.2. Cài đặt thuật toán với các cấu trúc dữ liệu

Để cài đặt thuật toán Dijkstra, chúng ta sử dụng bộ nhớ của các đỉnh:

Nhận của mỗi đỉnh v gồm ba thành phần cho biết các thông tin: đã tìm được đường đi ngắn nhất từ đỉnh nguồn đến v hay chưa, khoảng cách (độ dài đường đi) từ s đến v hiện biết và đỉnh đi trước đỉnh v trong đường đi tốt nhất hiện biết. Các thành phần này sẽ được cất giữ tương ứng trong các biến  $k[v]$ ,  $d[v]$  và  $p[v]$ .

Cách cài đặt trực tiếp của thuật toán Dijkstra có thể mô tả như sau:

### Dijkstra\_Table(G, s)

```
1. for u ∈ V do {
2. d[u] ← infinity;
3. p[u] ← NIL;
4. k[u] ← FALSE;
5. }
6. d[s] ← 0; // s là đỉnh nguồn
7. T = V;
8. while T ≠ ∅ do {
9. u ← đỉnh có d[u] là nhỏ nhất trong T;
10. k[u]=TRUE;
11. T = T-{u};
12. for (v ∈ Adj(u)) && !k[v] do
13. if d[v] > d[u] + c[u, v] {
14. d[v] = d[u] + c[u, v];
15. p[v] = u;
16. }
17. }
```

Dễ dàng nhận thấy rằng Dijkstra\_Table(G, s) đòi hỏi thời gian  $O(|V|^2+|E|)$ .

Do tại mỗi bước ta cần tìm ra đỉnh với nhãn khoảng cách nhỏ nhất, nên để thực hiện thao tác này một cách hiệu quả, ta sẽ sử dụng hàng đợi có ưu tiên. Dưới đây ta mô tả thuật toán Dijkstra với hàng đợi có ưu tiên:

### Dijkstra\_Heap(G, s)

```
1. for u ∈ V do {
2. d[u] ← infinity;
3. p[u] ← NIL;
4. k[u] ← FALSE;
5. }
6. d[s] ← 0; // s là đỉnh nguồn
7. Q ← Build_Min_Heap(d[V]); // Khởi tạo hàng đợi có
uu tiên Q từ
 // d[v] = (d[v], v∈V)
```

```

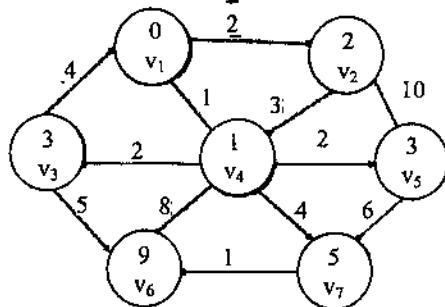
8. while Not Empty(Q) do {
9. u ← Extract-Min(Q); // loại bỏ gốc của Q và đưa vào u
10. k[u]=TRUE;
11. for (v ∈ Adj(u)) && !k[v] do
12. if d[v] > d[u] + c[u, v] {
13. d[v] = d[u] + c[u, v];
14. p[v] = u;
15. Decrease-Key(Q, v, d[v]);
16. }
17. }

```

### Ví dụ:

Bảng dưới đây minh họa quá trình thực hiện thuật toán vừa mô tả để tìm đường đi ngắn nhất từ đỉnh  $v_1$  đến tất cả các đỉnh còn lại.

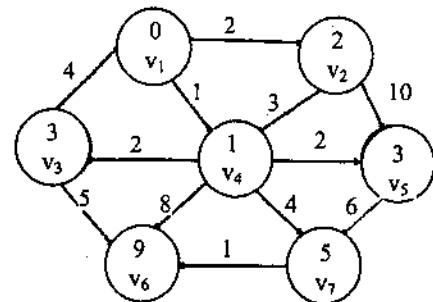
| Hàng đợi có ưu tiên |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|----------------|-------|---|-------|----------|-------|----------|-------|----------|-------|----------|-------|----------|-------|----------|
| Khởi tạo            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
|                     | <table border="1"> <thead> <tr> <th>v</th><th>d<sub>v</sub></th></tr> </thead> <tbody> <tr> <td><math>v_1</math></td><td>0</td></tr> <tr> <td><math>v_2</math></td><td><math>\infty</math></td></tr> <tr> <td><math>v_3</math></td><td><math>\infty</math></td></tr> <tr> <td><math>v_4</math></td><td><math>\infty</math></td></tr> <tr> <td><math>v_5</math></td><td><math>\infty</math></td></tr> <tr> <td><math>v_6</math></td><td><math>\infty</math></td></tr> <tr> <td><math>v_7</math></td><td><math>\infty</math></td></tr> </tbody> </table>                                                                                     | v | d <sub>v</sub> | $v_1$ | 0 | $v_2$ | $\infty$ | $v_3$ | $\infty$ | $v_4$ | $\infty$ | $v_5$ | $\infty$ | $v_6$ | $\infty$ | $v_7$ | $\infty$ |
| v                   | d <sub>v</sub>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_1$               | 0                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_2$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_3$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_4$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_5$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_6$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_7$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
|                     | <table border="1"> <thead> <tr> <th>v</th><th>d<sub>v</sub></th></tr> </thead> <tbody> <tr> <td><math>v_1</math></td><td>1</td></tr> <tr> <td><math>v_2</math></td><td><math>\infty</math></td></tr> <tr> <td><math>v_3</math></td><td><math>\infty</math></td></tr> <tr> <td><math>v_4</math></td><td><math>\infty</math></td></tr> <tr> <td><math>v_5</math></td><td><math>\infty</math></td></tr> <tr> <td><math>v_6</math></td><td><math>\infty</math></td></tr> <tr> <td><math>v_7</math></td><td><math>\infty</math></td></tr> </tbody> </table> <p>Xử lý đỉnh <math>v_1</math>: Chưa nhãn <math>v_2</math> và <math>v_4</math>.</p> | v | d <sub>v</sub> | $v_1$ | 1 | $v_2$ | $\infty$ | $v_3$ | $\infty$ | $v_4$ | $\infty$ | $v_5$ | $\infty$ | $v_6$ | $\infty$ | $v_7$ | $\infty$ |
| v                   | d <sub>v</sub>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_1$               | 1                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_2$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_3$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_4$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_5$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_6$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |
| $v_7$               | $\infty$                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |   |                |       |   |       |          |       |          |       |          |       |          |       |          |       |          |



| v     | $d_v$ |
|-------|-------|
| $v_2$ | 2     |
| $v_3$ | 3     |
| $v_5$ | 3     |
| $v_7$ | 5     |
| $v_6$ | 9     |

Xử lý đỉnh  $v_4$

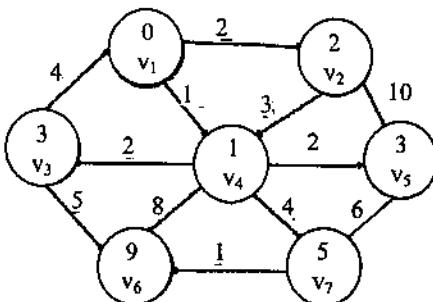
Chưa nhăn  $v_3, v_5, v_6, v_7$ .



| v     | $d_v$ |
|-------|-------|
| $v_3$ | 3     |
| $v_5$ | 3     |
| $v_7$ | 5     |
| $v_6$ | 9     |

Xử lý đỉnh  $v_2$

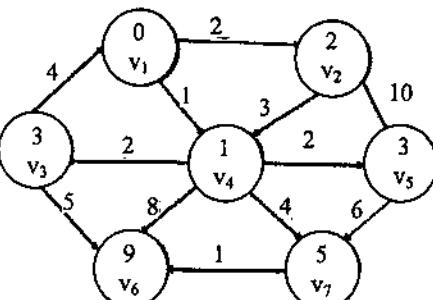
- $v_4$ : đã xong.
- $v_5$ : không sửa được nhăn.



| v     | $d_v$ |
|-------|-------|
| $v_5$ | 3     |
| $v_7$ | 5     |
| $v_6$ | 8     |

Xử lý đỉnh  $v_3$

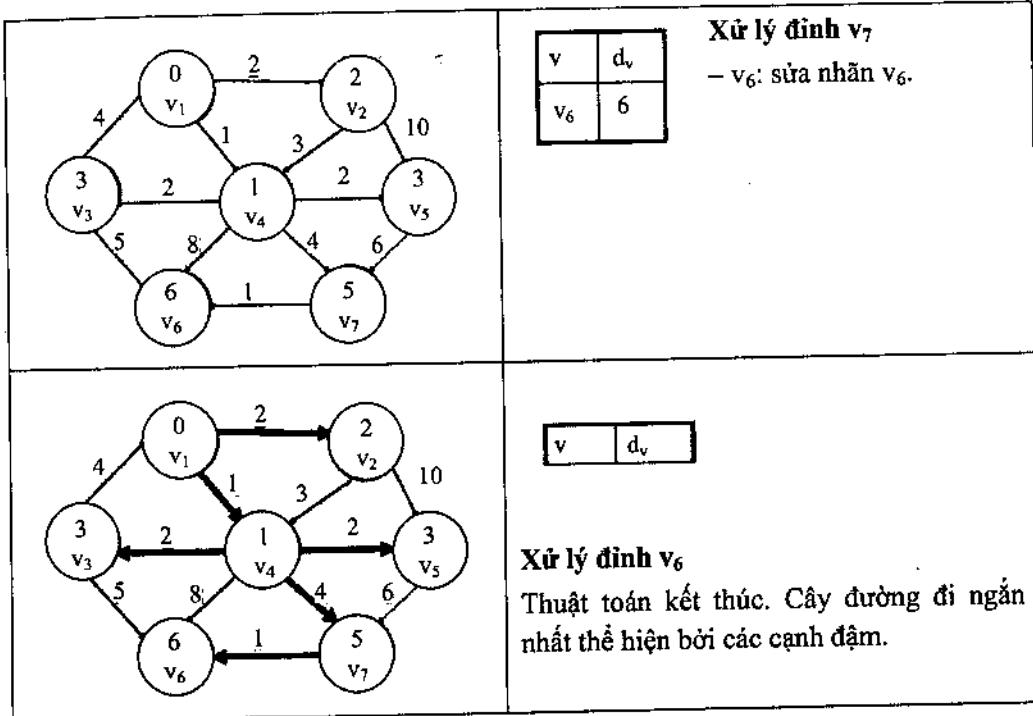
- $v_1$ : đã xong.
- $v_6$ : không sửa được nhăn.



| v     | $d_v$ |
|-------|-------|
| $v_7$ | 5     |
| $v_6$ | 8     |

Xử lý đỉnh  $v_5$

- $v_7$ : không sửa được nhăn.



### Phân tích thời gian tính của thuật toán

- Vòng lặp for ở dòng 1 đòi hỏi thời gian  $O(|V|)$ .
- Việc khởi tạo đồng min đòi hỏi thời gian  $O(|V|)$ .
- Vòng lặp while ở dòng 8 lặp  $|V|$ , lần do đó thao tác Extract-Min thực hiện  $|V|$  lần và đòi hỏi thời gian  $O(|V| \log |V|)$ .
  - Thao tác Decrease-Key ở dòng 15 phải thực hiện không quá  $O(|E|)$  lần. Do đó thời gian thực hiện thao tác này trong thuật toán là  $O(|E| \log |V|)$ .

Vậy tổng cộng thời gian tính của thuật toán là  $O((|E| + |V|) \log |V|)$ .

Nếu sử dụng cấu trúc dữ liệu đồng Fibonacci, ta có thể thu được cài đặt của thuật toán Dijkstra với thời gian tính  $O(|E| + |V| \log |V|)$ .

Kinh nghiệm tính toán cho thấy cài đặt thuật toán Dijkstra sử dụng đồng (heap) chỉ tỏ ra hiệu quả hơn so với cách cài đặt trực tiếp (Dijkstra\_Table) đối với những đồ thị thưa có số lượng đỉnh lớn hơn 5000. Chẳng hạn, với đồ thị có:

- 2000 đỉnh, 1 triệu cạnh: cài đặt với đồng chậm hơn từ 2 đến 3 lần;
- 100000 đỉnh, 1 triệu cạnh: cài đặt với đồng nhanh hơn 500 lần;
- 1 triệu đỉnh, 2 triệu cạnh: cài đặt với đồng nhanh hơn 10000 lần.

Bài toán đường đi ngắn nhất là một trong những bài toán có nhiều ứng dụng trong thực tiễn. Dưới đây ta đưa ra một ứng dụng của bài toán đường đi ngắn nhất.

### Ứng dụng: Bài toán "Canh gác bảo tàng"

Viện bảo tàng X tổ chức cuộc trưng bày giới thiệu một số bức tranh gốc của danh họa Leonardo da Vinci. Những bức tranh của các danh họa thường là mục tiêu của nhiều tên trộm cắp chuyên nghiệp. Vì vậy, Ban Giám đốc bảo tàng cần giải quyết bài toán bảo vệ an toàn cho các bức tranh độc nhất vô nhị này. Theo kế hoạch, cuộc triển lãm sẽ diễn ra trong khoảng thời gian  $n$  giờ. Thời điểm bắt đầu triển lãm được tính là 0. Có  $m$  vệ sỹ có nghiệp vụ cao có thể thuê để canh gác các bức tranh. Để đơn giản, các vệ sỹ sẽ được gán số hiệu từ 1 đến  $m$ . Vệ sỹ  $i$  chấp nhận đứng canh trong khoảng thời gian từ thời điểm  $s_i$  đến thời điểm  $t_i$  ( $0 \leq s_i < t_i \leq n$ ) với tiền công là  $c_i$ ,  $i = 1, 2, \dots, m$ .

**Yêu cầu:** Hãy giúp Ban Giám đốc lựa chọn thuê các vệ sỹ trong số  $m$  vệ sỹ để tại bất cứ thời điểm nào trong thời gian diễn ra triển lãm cũng luôn có ít nhất một vệ sỹ đứng canh, đồng thời tổng chi phí phải trả cho các vệ sỹ được thuê là nhỏ nhất.

**Dữ liệu:** Vào từ file văn bản GALERY.INP:

- Dòng đầu tiên chứa hai số nguyên dương  $n$  và  $m$  ( $n, m \leq 10^5$ );
- Dòng thứ  $i$  trong số  $m$  dòng tiếp theo chứa ba số nguyên không âm  $s_i, t_i, c_i$  ( $0 < c_i \leq 10^5$ ),  $i = 1, 2, \dots, m$ .

Ta sẽ chỉ ra rằng bài toán đặt ra có thể quy đổi về bài toán đường đi ngắn nhất trên đồ thị.

Xây dựng đồ thị  $G = (V, E)$  với trọng số trên cạnh:

- $V = \{0, 1, \dots, n\}$ : mỗi đỉnh của đồ thị tương ứng với thời điểm;
- $E = \{e_i = (s_i, t_i) : i = 1, 2, \dots, m\} \cup \{f_k = (k, k-1) : k = 1, 2, \dots, n\}$ ;
- $c(e_i) = c_i$ ,  $i = 1, 2, \dots, m$ ;  $c(f_k) = 0$ ,  $k = 1, 2, \dots, n$ .

Như vậy đồ thị  $G$  có  $n$  đỉnh, và  $n + m$  cạnh.

Ví dụ, khi  $n = 8$ ,  $m = 5$  và thông tin về 5 vệ sĩ được cho trong bảng sau đây:

| Vệ sỹ | $s_i$ | $t_i$ | $c_i$ |
|-------|-------|-------|-------|
| 1     | 0     | 3     | 5     |
| 2     | 2     | 8     | 3     |
| 3     | 3     | 5     | 9     |
| 4     | 4     | 8     | 2     |
| 5     | 7     | 8     | 11    |

ta có đồ thị  $G$  như sau:



Dễ thấy mỗi cách thuê vé số đáp ứng yêu cầu đầu bài tương ứng với một đường đi từ đỉnh 0 đến đỉnh n và ngược lại mỗi đường đi từ đỉnh 0 đến đỉnh n tương ứng với một cách thuê vé số đáp ứng các yêu cầu đặt ra. Rõ ràng độ dài của đường đi chính là tổng tiền công thuê vé số trong cách thuê vé số tương ứng.

Do đó, bài toán đặt ra được dẫn về tìm đường đi ngắn nhất từ đỉnh 0 đến đỉnh n trên đồ thị G.

### Phân tích cài đặt

– Theo đề bài: Đồ thị có  $10^5$  đỉnh và  $2*10^5$  cạnh.

– Vì thế thời gian tính của cách cài đặt:

+ Dùng Dijkstra Table:  $O(|V|^2)$ ,

tức là quãng  $10^{10}$ .

+ Dùng Dijkstra với hàng đợi có ưu tiên:  $O((|E| + |V|) \log |V|)$ ,

tức là quãng  $(3*10^5) \log 10^5 < 60*10^5$ .

Nhận thấy là  $10^{10}/(60*10^5) = 100000/60 = 10000/6 > 1666$ .

Vì vậy nên sử dụng cách cài đặt sử dụng hàng đợi có ưu tiên.

## BÀI TẬP CHƯƠNG 7

- Xét đồ thị có hướng G với các đỉnh được đánh số từ 1 đến 7 được cho bởi ma trận kè A sau đây:

$$A = \begin{vmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{vmatrix}$$

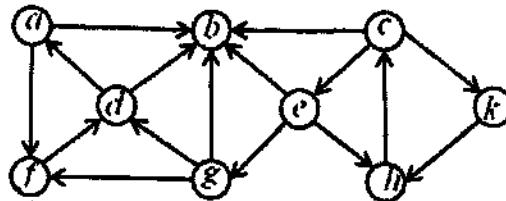
a) Đưa ra danh sách kè của đồ thị đã cho.

b) Thực hiện tìm kiếm theo chiều sâu trên đồ thị G (giả thiết là khi duyệt các đỉnh của đồ thị và các đỉnh trong danh sách kè của một đỉnh, ta duyệt theo thứ tự chỉ số tăng dần). Hãy đưa ra danh sách các cạnh của cây, cạnh ngược, cạnh tới, cạnh vòng.

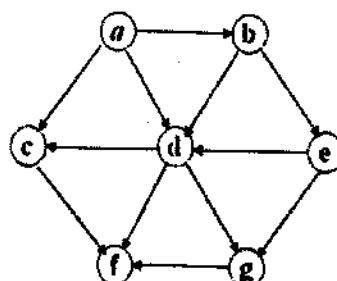
2. Xét đồ thị có hướng  $G$  với các đỉnh được đánh số từ 1 đến 7 được cho bởi ma trận kề  $A$  sau đây:

$$A = \begin{vmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{vmatrix}$$

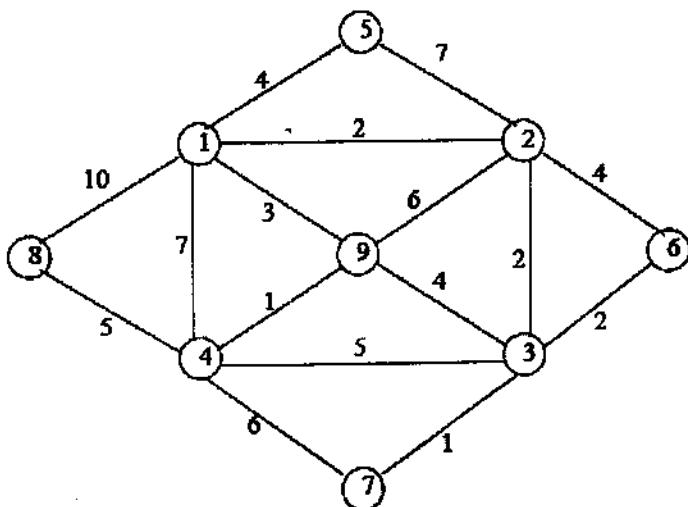
- a) Đưa ra danh sách kề của đồ thị đã cho.
- b) Thực hiện tìm kiếm theo chiều sâu trên đồ thị  $G$  (giả thiết là khi duyệt các đỉnh của đồ thị và các đỉnh trong danh sách kề của một đỉnh, ta duyệt theo thứ tự chỉ số tăng dần). Hãy đưa ra danh sách các cạnh của cây, cạnh ngược, cạnh tới, cạnh vòng.
3. Xét đồ thị có hướng  $G$  trong hình vẽ sau:



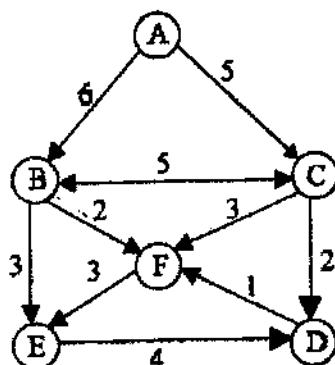
- a) Đưa ra ma trận kề của đồ thị.
- b) Hãy thực hiện thuật toán tìm kiếm theo chiều sâu trên  $G$  (giả thiết là khi duyệt đỉnh, ta duyệt theo thứ tự trong bảng chữ cái), yêu cầu đưa ra các kết quả sau:
- Vẽ rãnh DFS thu được.
  - Đưa ra phân loại cạnh cho các cạnh.
4. Xét đồ thị có hướng không có chu trình  $G$  trong hình vẽ sau:



- a) Đưa ra cách biểu diễn của đồ thị đã cho dưới dạng danh sách kề.
- b) Hãy trình diễn thuật toán xóa dần cung để sắp xếp các đỉnh của  $G$  theo thứ tự topô.
- c) Vẽ bao đóng truyền ứng của  $G$ .
5. Cho đồ thị có hướng  $G = (V, E)$  với trọng số không âm trên các cung. Giả sử  $e = (u, v)$  là một cung của đồ thị  $G$ .
- Mô tả thuật toán (trong giả ngôn ngữ) tìm chu trình với độ dài nhỏ nhất chứa cung  $e$  đã cho.
  - Đưa ra đánh giá thời gian tính của thuật toán đề xuất.
6. Xét đồ thị vô hướng với trọng số trên cạnh cho trong hình vẽ sau đây:



- a) Đưa ra ma trận trọng số của đồ thị.
- b) Tìm cây khung nhỏ nhất của đồ thị bằng thuật toán Kruskal.
7. Xét đồ thị có hướng với trọng số trên cạnh cho trong hình vẽ sau đây:



- a) Đưa ra ma trận trọng số của đồ thị.
- b) Tìm đường đi ngắn nhất từ đỉnh A đến tất cả các đỉnh còn lại trên đồ thị đã cho theo thuật toán Dijkstra.
8. Đơn đồ thị vô hướng  $G = (V, E)$  với  $n$  đỉnh ( $|V| = n$ ) và  $m$  cạnh ( $|E| = m$ ) được coi là có dạng *tựa cây* nếu như có thể tìm được một cạnh trong  $E$  mà việc loại bỏ nó khỏi đồ thị  $G$  sẽ dẫn đến một cây.
- a) Hãy đề xuất thuật toán (mô tả trong giả ngôn ngữ) nhận biết đơn đồ thị vô hướng  $G = (V, E)$  có phải là có dạng tựa cây hay không?
- b) Chứng minh tính đúng đắn và đánh giá thời gian tính của thuật toán đề xuất. Chú ý là đồ thị đầu vào có thể có số cạnh ( $m$ ) lớn hơn rất nhiều so với số đỉnh ( $n$ ). Giả thiết đồ thị được cho bởi danh sách kè.
9. Bài toán cây khung lớn nhất: Cho đồ thị vô hướng liên thông  $G = (V, E)$  với trọng số trên cạnh  $c(e)$ ,  $e \in E$ . Trong số tất cả các cây khung của  $G$ , hãy tìm cây khung với trọng số lớn nhất. Hãy chỉ ra cách áp dụng thuật toán Kruskal để giải bài toán đặt ra.
10. Cho đồ thị vô hướng liên thông  $G = (V, E)$  với trọng số trên cạnh  $c(e)$ ,  $e \in E$ . Hỏi cây khung nhỏ nhất của đồ thị  $G$  có thay đổi hay không nếu như ta cộng thêm trọng số của tất cả các cạnh của đồ thị với cùng một hằng số  $C$ . Nếu đúng hãy chứng minh, nếu sai hãy đưa ra phản ví dụ.
11. Xét bài toán "Phân loại văn bản cổ" sau đây:

Trong một lần khai quật một khu di tích được xác định là một thư viện cổ cách chúng ta hàng nghìn năm, các nhà khảo cổ phát hiện được  $n$  văn bản cổ. Ký hiệu  $V$  là tập gồm  $n$  văn bản cổ này. Theo những câu truyện truyền thuyết được người dân địa phương kể lại, các nhà khảo cổ biết rằng các văn bản cổ này thuộc  $k$  loại ngôn ngữ. Vấn đề đặt ra cho các nhà khảo cổ là phải xác định xem mỗi văn bản thuộc loại ngôn ngữ nào.

Cụ thể, các nhà khảo cổ cần giải bài toán sau đây: Mỗi một văn bản cổ là một xâu gồm không quá 100 ký hiệu. Cho xâu  $x$ , ta gọi xâu con của  $x$  là xâu thu được từ nó bởi việc xóa đi một số ký hiệu và giữ nguyên thứ tự của các ký hiệu còn lại. Một xâu  $z$  được gọi là xâu con chung của hai xâu  $x$  và  $y$  nếu như nó vừa là xâu con của  $x$  vừa là xâu con của  $y$ . Số lượng ký hiệu trong một xâu được gọi là độ dài của nó. Ta gọi độ phân biệt của hai xâu  $x$  và  $y$ , ký hiệu  $\rho(x, y)$ , là hiệu giữa tổng độ dài của hai xâu  $x$ ,  $y$  và hai lần độ dài của xâu con chung dài nhất của chúng.

Ví dụ: Nếu  $x = \text{'aaa'}$  còn  $y = \text{'aca'}$  thì do độ dài của xâu con chung dài nhất của hai xâu  $x$  và  $y$  là 2, nên ta có  $\rho(x,y) = 3 + 3 - 2*2 = 2$ .

Giả sử  $P$  và  $Q$  là hai tập con không giao nhau của tập  $V$  gồm  $n$  văn bản cổ đã cho, ta gọi độ tương đồng của  $P$  và  $Q$  là số:

$$d(P,Q) = \min \{ \rho(x,y) : x \in P, y \in Q \}.$$

Rõ ràng có một tương ứng 1-1 giữa một cách phân loại các văn bản đã cho theo  $k$  ngôn ngữ với một cách phân hoạch tập các văn bản cổ đã cho  $V$  ra thành  $k$  tập con:

$$V = V_1 \cup V_2 \cup \dots \cup V_k;$$

$$V_i \cap V_j \neq \emptyset, i \neq j.$$

Ta gọi độ chính xác của cách phân hoạch này là số:

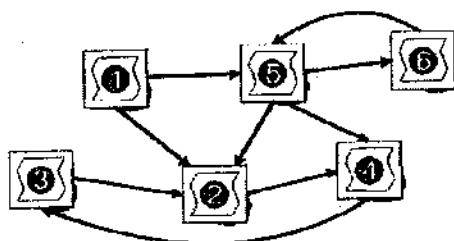
$$\min \{ d(V_i, V_j) : 1 \leq i < j \leq k \}.$$

Bài toán phân loại văn bản cổ đòi hỏi phải tìm cách phân hoạch tập các văn bản cổ đã cho  $V$  ra thành  $k$  tập con  $V_1, V_2, \dots, V_k$  với độ chính xác lớn nhất.

Hãy chỉ ra cách quy dẫn bài toán nói trên về bài toán cây khung nhỏ nhất và phát triển thuật toán giải bài toán đặt ra. Đánh giá thời gian tính của thuật toán đề xuất.

12. Trong mạng xã hội, mỗi trang web được tổ chức trên một máy tính thành viên và cung cấp dịch vụ truy nhập tới một số trang web khác. Để truy nhập tới một trang web nào đó không có trong danh mục kết nối trực tiếp của mình, người dùng phải truy nhập tới trang web khác có kết nối với mình, dựa vào danh mục dịch vụ của trang web này để chuyển tới trang web khác theo tùy chọn, cứ như thế cho đến khi tới được trang web mình cần. Thời gian để truy nhập tới một trang web phụ thuộc chủ yếu vào số lần mở trang web trong quá trình truy nhập. Như vậy, người dùng cần chủ động chọn lộ trình truy nhập hợp lý.

Sau một thời gian làm việc trên mạng, Sáng – một thành viên nhiệt thành đã tích lũy kinh nghiệm, tạo một cơ sở dữ liệu cho biết từ một trang web có thể đi tới những trang web nào trong mạng. Trong cơ sở dữ liệu, các trang web được đánh số từ 1 đến  $n$  và có  $m$  bản ghi, mỗi bản ghi có dạng cặp có thứ tự  $(u, v)$  cho biết trang web  $u$  có kết nối tới trang web  $v$  ( $1 \leq u, v \leq n, u \neq v$ ). Cơ sở dữ liệu chưa được chuẩn hóa, vì vậy có thể chứa các cặp  $(u, v)$  giống nhau.



Trang web của Sáng có số hiệu là  $s$ . Dựa vào cơ sở dữ liệu, Sáng có thể xác định lô trình truy nhập nhanh nhất (tức số lần phải mở trang web là ít nhất) từ trang web  $s$  tới trang web  $u$  bất kỳ. Tuy vậy, ở mạng xã hội, mọi chuyện đều có thể xảy ra: một khu vực nào đó bị mất điện, máy của một thành viên bị hỏng, trang web đang bị đóng để nâng cấp,... Kết quả là một vài trang web nào đó có thể tạm thời không hoạt động. Như vậy, nếu từ  $s$  có ít nhất hai lô trình nhanh nhất khác nhau tới  $u$  thì khả năng thực hiện truy nhập được một cách nhanh nhất tới  $u$  là lớn hơn so với những trang web chỉ có duy nhất một lô trình nhanh nhất. Hai lô trình gọi là khác nhau nếu có ít nhất một trang web có ở lô trình này mà không có ở lô trình kia hoặc cả hai lô trình cùng đi qua những trang web như nhau nhưng theo các trình tự khác nhau. Những trang web mà từ  $s$  tới đó có ít nhất hai lô trình nhanh nhất khác nhau được gọi là  **ổn định đối với  $s$** . Trang web mà từ  $s$  không có lô trình tới nó là không  **ổn định đối với  $s$** .

Ví dụ, với mạng nêu ở hình trên ( $n = 6, m = 9$ ), các trang web 4 và 3 là ổn định với  $s = 1$  (từ 1 tới 4 có hai lô trình nhanh nhất:  $1 \rightarrow 2 \rightarrow 4$  và  $1 \rightarrow 5 \rightarrow 4$ , từ 1 tới 3 cũng có hai lô trình nhanh nhất:  $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$  và  $1 \rightarrow 5 \rightarrow 4 \rightarrow 3$ ).

Hãy phát triển thuật toán xác định số lượng trang web ổn định đối với  $s$ . Đánh giá thời gian tính của thuật toán đề xuất.

13. Chương trình trên C trình bày dưới đây minh họa cài đặt một số ứng dụng của DFS đối với đồ thị có hướng:

- Dưa ra phân loại cạnh khi duyệt đồ thị theo DFS.
  - Kiểm tra đồ thị có chứa chu trình?
  - Dưa ra thứ tự sắp xếp tópô nếu đồ thị có hướng, không có chu trình.
- Chương trình thực hiện nhập đồ thị vào từ file văn bản có tên "TOPO\_ORDER.INP" có cấu trúc như sau:
- Dòng đầu tiên chứa số đỉnh của đồ thị;
  - Các dòng tiếp theo mô tả cạnh của đồ thị, mỗi dòng chứa hai số là hai đầu mút của một cạnh (các đỉnh của đồ thị được đánh số từ 1 đến  $n$ ).

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <conio.h>
```

```
// Các hằng sử dụng
```

```

#define TREE_EDGE 1
#define BACK_EDGE 2
#define CROSS_EDGE 3
#define FORWARD_EDGE 4
#define INT_MAX 999999999

#define WHITE 0
#define GRAY 1
#define BLACK 2

int n, m, callCount, at, atPos = 0;
int* parent;
int* color;
int* distance;
int* topological_order;
bool isAcyclic;

// Cấu trúc một nút trong danh sách kề
struct ListIt {
 int vertex;
 int type;
 ListIt *p_next; };

// Định nghĩa kiểu côn trỏ
typedef ListIt* pListIt;
pListIt* adjacencyList; // Biểu diễn đồ thị bởi danh sách kề
pListIt* inverseList; // Biểu diễn đồ thị bởi danh sách kề ngược

bool add(pListIt &dest, int val);
void read();
void DFS(int vertex);
void DFS_inver(int vertex);
void Waiting();
void inverse(pListIt*& from, pListIt*&to);

```

```

// Ta cần sắp xếp các đỉnh trong danh sách kề
// theo thứ tự tăng dần của chỉ số
bool add(pListIt &dest, int val){
 // tạo nút
 pListIt p;
 p = (pListIt) malloc(sizeof(ListIt));
 p->vertex = val;
 p->type = 0;
 if(!dest) // chèn nút đầu tiên
 {
 p->p_next = NULL;
 dest = p;
 } else
 {
 pListIt find = dest;
 pListIt at = NULL;
 // Tìm nút lớn hơn đầu tiên và chèn nút mới vào
 // trước nó
 while(find && find->vertex <= val){
 if(find->vertex == val) // không chèn cạnh lặp
 {
 free(p);
 return false;
 }
 at = find;
 find = find->p_next; }
 // insert at
 if (at) // chèn vào vị trí tìm được
 {
 p->p_next = at->p_next;
 at->p_next = p;
 } else // chèn vào đầu
 {
 p->p_next = dest;
 dest = p; }
 } return true;
}

void read() {
 // Đọc dữ liệu vào từ file
 freopen("TOPO_ORDER.INP","r",stdin);
}

```

```

scanf("%d", &n); // đọc vào số lượng đỉnh

// cấp phát bộ nhớ cho mảng con trỏ
adjacencyList = (pListIt*) malloc(n+1, sizeof(pListIt));
color = (int*)malloc(n+1, sizeof(int));
parent = (int*)malloc(n+1, sizeof(int));
distance = (int*)malloc(n+1, sizeof(int));
topological_order = (int*)malloc(n+1, sizeof(int));

// Đọc vào danh sách cạnh
int i, from, to;
m = 0;
for (i = 1; ; i++) {
 scanf("%d %d", &from, &to); // from -> to
 if(feof(stdin)) break;
 add(adjacencyList[from], to);
 ++m;
 //add(adjacencyList[to], from);
 // Nếu đồ thị là vô hướng thì cần có thêm lệnh
này
}
}

// DFS xuất phát từ đỉnh vertex
void DFS(int vertex) {
 pListIt p;
 color[vertex] = GRAY; // visited
 for (p = adjacencyList[vertex]; p != NULL; p = p ->
p_next)
 if (!color[p -> vertex]) {
 parent[p->vertex] = vertex;
 distance[p->vertex] = distance[vertex] + 1;
 p->type = TREE_EDGE;
 DFS(p -> vertex);
 }
 else {

```

```

 if(color[p->vertex] == GRAY)
 { p->type = BACK_EDGE;
 if (isAcyclic) {
 isAcyclic = false;

 // Đưa ra chu trình
 printf("\n \n Graph contains the following
cycle: ");
 int l = vertex;
 while(l != p->vertex) {
 printf(" %d ", l);
 l = parent[l];

 } printf(" %d ", l);
 }
 }
 else
 if (distance[p->vertex] > distance[vertex])
 {
 p->type = FORWARD_EDGE ;
 }
 else
 p->type = CROSS_EDGE ;
 }
 color[vertex] = BLACK;

 // đưa vào danh sách đỉnh theo thứ tự sắp xếp
tôpô
 ++atPos;
 topological_order[n-atPos] = vertex;
}

void Waiting()
{
 printf("\n Press any key to continue..."); getch();
}

```

```

int main() {
 int i;
 isAcyclic = true;
 printf("\n CHUONG TRINH THUC HIEN DFS VA UNG
DUNG\n");
 printf(" ======\n");
 read();
 printf("\n Da doc du lieu vao tu file...\n");
 Waiting();
 at = 0;
 memset(color, WHITE, (n+1)*sizeof(int));
 memset(parent, 0, (n+1)*sizeof(int));
 memset(topological_order, 0, (n+1)*sizeof(int));

// DFS(G)
 for (i = 1; i <= n; i++)
 if (!color[i]) {
 callCount++; // counting the number of
connected component
 // in undirected graph
 DFS(i);
 }

 if (isAcyclic) {
 printf(" \n\nGraph is acyclic.");
 printf(" \n\nTopological Order: \n");
 for (i = 0; i != n; ++i)
 printf(" %d" , topological_order[i]);
 }

// Phân loại cạnh
 printf("\n\n Tree edges: \n");
 pListIt p;
 for(i = 1; i <= n; ++i)
 for (p = adjacencyList[i]; p; p = p->p_next) {
 if (p->type == TREE_EDGE) {
 printf(" %d - %d ; ", i, p->vertex);
 }
 }
}

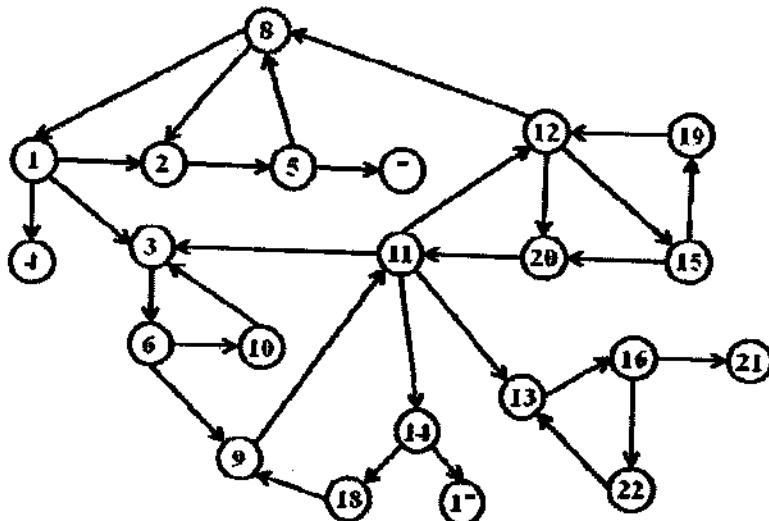
```

```

printf(" \n\n Back edges: \n");
for(i = 1; i <= n; ++i)
 for (p = adjacencyList[i]; p; p = p->p_next){
 if (p->type == BACK_EDGE){
 printf(" %d - %d ; ", i, p->vertex);
 }
 }
printf("\n\n Forward edges: \n");
for(i = 1; i <= n; ++i)
 for (p = adjacencyList[i]; p; p = p->p_next) {
 if (p->type == FORWARD_EDGE){
 printf(" %d - %d ; ", i, p->vertex);
 }
 }
printf("\n\n Cross edges: \n");
for(i = 1; i <= n; ++i)
 for (p = adjacencyList[i]; p; p = p->p_next) {
 if (p->type == CROSS_EDGE){
 printf(" %d - %d ; ", i, p->vertex);
 }
 }
printf("\n\n ***** THE END *****\n");
Waiting();
return 0;
}

```

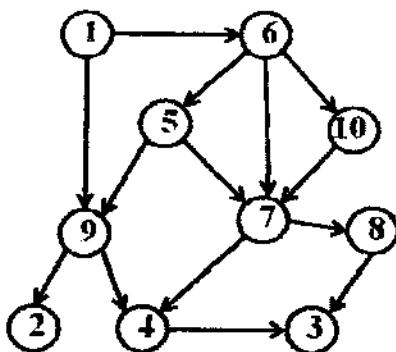
- a) Hãy gõ lại chương trình, theo dõi hoạt động của chương trình trên các bộ dữ liệu tương ứng với các đồ thị cho trong các hình vẽ sau đây:



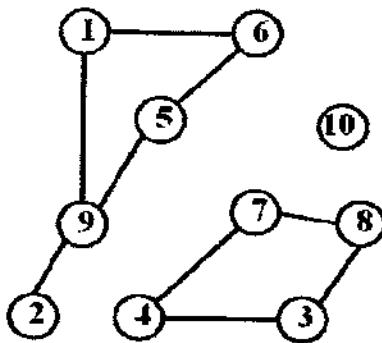
File dữ liệu TOPO\_ORDER.INP có nội dung sau:

```
22
1 2
1 3
1 4
2 5
5 7
5 8
3 6
6 9
6 10
8 2
8 1
9 11
10 3
11 12
11 13
11 14
11 3
12 8
12 15
13 16
12 20
14 17
14 18
15 19
15 20
16 21
16 22
18 9
19 12
20 11
22 13
```

Hãy chuẩn bị file dữ liệu tương ứng với đồ thị dưới đây, chạy chương trình và theo dõi kết quả:



b) Hãy điều chỉnh lại chương trình để nó thực hiện tìm kiếm theo chiều sâu trên đồ thị vô hướng và ứng dụng vào bài toán liên thông. Chạy thử chương trình với đồ thị sau đây:



## TÀI LIỆU THAM KHẢO

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Algorithms*. Addison-Wesley, Reading, MA, 1975.
2. D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Second edition, Addison-Wesley, Reading, MA, 1997.
3. Robert Sedgewick. *Algorithms in C*. Third Edition. Addison-Wesley, 1998.
4. Robert Sedgewick. *Algorithms in C++, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*. 3th Edition, Addison-Wesley, 1999.
5. Robert Sedgewick. *Algorithms in C++ Part 5: Graph Algorithms (3rd Edition)*. 3th Edition, Addison-Wesley, 2002.
6. Michael T. Goodrich, Roberto Tamassia, David M. Mount. *Data Structures and Algorithms in C++*. 704 pages. Wiley, 2003.
7. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. *Introduction to Algorithms*. Second Edition, MIT Press, 2001. (Có bản dịch tiếng Việt).
8. A. Levitin. *The design & analysis of algorithms*. Addison Wesley, 2003.
9. R. Johnsonbaugh, M. Schaefer. *Algorithms*. Pearson Education, 2004.
10. Đỗ Xuân Lôi. *Cấu trúc dữ liệu và giải thuật*. Nhà xuất bản Đại học Quốc gia, Hà Nội, 2005.

## PHỤ LỤC

# BẢNG THUẬT NGỮ ANH – VIỆT

Phụ lục này dẫn ra một số thuật ngữ tiếng Anh được sử dụng, theo thứ tự xuất hiện trong giáo trình. Để tiện tra cứu thuật ngữ Việt – Anh, các thuật ngữ này được dẫn ra trong giáo trình ngay lần xuất hiện đầu tiên.

| Tiếng Anh              | Tiếng Việt                      |
|------------------------|---------------------------------|
| <b>Chương 1</b>        |                                 |
| divide and conquer     | chia đế trị                     |
| dynamic programming    | quy hoạch động                  |
| tractable              | dễ giải                         |
| intractable            | khó giải                        |
| nonsolvable            | không giải được                 |
| pseudo language        | giả ngôn ngữ                    |
| upper bound            | cận trên                        |
| lower bound            | cận dưới                        |
| pigeonhole sorting     | sắp xếp kiều nhốt chim vào lồng |
| asymptotic notation    | ký hiệu tiệm cận                |
| transitivity           | truyền ứng                      |
| symmetry               | đối xứng                        |
| transpose symmetry     | đối xứng chuyển vị              |
| <b>Chương 2</b>        |                                 |
| basic step             | bước cơ sở                      |
| recursive step         | bước đệ quy                     |
| root                   | đầu                             |
| combine                | tổng hợp                        |
| merge sort             | (thuật toán) sắp xếp trộn       |
| backtracking algorithm | thuật toán quay lui             |

### Chương 3

|                              |                                        |
|------------------------------|----------------------------------------|
| data type                    | kiểu dữ liệu                           |
| built-in data type           | kiểu dữ liệu dựng sẵn                  |
| integer numeric type         | kiểu dữ liệu số nguyên                 |
| floating point numeric type  | kiểu dữ liệu số thực                   |
| primitive type               | kiểu nguyên thủy                       |
| array type                   | kiểu mảng                              |
| abstract data type           | kiểu dữ liệu trừu tượng                |
| implementation               | cài đặt                                |
| data structure               | cấu trúc dữ liệu                       |
| cell                         | ô                                      |
| array                        | mảng                                   |
| record structure             | cấu trúc bản ghi                       |
| record                       | bản ghi                                |
| dictionary ADT               | kiểu dữ liệu trừu tượng từ điển        |
| pointer                      | con trỏ                                |
| base data structures         | cấu trúc dữ liệu cơ sở                 |
| linear data structures       | cấu trúc dữ liệu tuyến tính            |
| non linear data structures   | cấu trúc dữ liệu phi tuyến             |
| object                       | đối tượng                              |
| linear list                  | danh sách tuyến tính                   |
| indirect addressing          | địa chỉ không trực tiếp                |
| singly linked list           | danh sách mốc nối đơn                  |
| circularly linked list       | danh sách mốc nối vòng                 |
| doubly linked list           | danh sách mốc nối kép                  |
| linked list-multiple data    | danh sách mốc nối đa dữ liệu           |
| circular linked lists        | danh sách nối vòng                     |
| circular doubly linked lists | danh sách nối đôi vòng                 |
| linked lists of lists        | danh sách mốc nối của các danh sách    |
| multiply linked lists        | danh sách đa mốc nối                   |
| stack                        | ngăn xếp                               |
| array-based stack            | ngăn xếp cài đặt bởi mảng              |
| linked stack                 | ngăn xếp cài đặt bởi danh sách mốc nối |
| infix notation               | ký pháp trung tố                       |
| postfix notation             | ký pháp hậu tố                         |

|                     |                                     |
|---------------------|-------------------------------------|
| random-access       | trực truy                           |
| insert              | chèn (bô sung)                      |
| delete              | xóa (loại bỏ)                       |
| find                | tìm                                 |
| push                | nạp vào (thao tác đối với ngăn xếp) |
| pop                 | lấy ra (thao tác đối với ngăn xếp)  |
| last-in, first-out  | vào sau, ra trước                   |
| top                 | đỉnh (của ngăn xếp)                 |
| first in, first out | vào trước, ra trước                 |
| enqueue             | đưa vào (thao tác với hàng đợi)     |
| dequeue             | đưa ra (thao tác với hàng đợi)      |
| front               | phía đầu (của hàng đợi)             |
| rear                | phía đuôi (của hàng đợi)            |

#### Chương 4

|                 |               |
|-----------------|---------------|
| tree            | cây           |
| root            | đầu           |
| child           | con           |
| ancestor tree   | cây phả hệ    |
| binary tree     | cây nhị phân  |
| expression tree | cây biểu thức |
| ancestors       | tổ tiên       |
| descendants     | hậu duệ       |
| leaf            | lá            |
| sibling         | anh em        |
| internal node   | nút trong     |
| height          | chiều cao     |
| depth           | chiều sâu     |
| subtree         | cây con       |
| degree          | bậc           |
| ordered tree    | cây có thứ tự |
| preorder        | thứ tự trước  |
| postorder       | thứ tự sau    |
| inorder         | thứ tự giữa   |
| labeled tree    | cây có nhãn   |
| left child      | con trái      |
| right child     | con phải      |

|                        |                         |
|------------------------|-------------------------|
| full binary tree       | cây nhị phân đầy đủ     |
| complete binary tree   | cây nhị phân hoàn chỉnh |
| balanced binary tree   | cây nhị phân cân đối    |
| binary expression tree | cây nhị phân biểu thức  |
| decision tree          | cây quyết định          |

### Chương 5

|                                    |                                             |
|------------------------------------|---------------------------------------------|
| sort key                           | khóa sắp xếp                                |
| database management                | quản trị cơ sở dữ liệu                      |
| internal sort                      | sắp xếp trong                               |
| external sort                      | sắp xếp ngoài                               |
| in place                           | tính tại chỗ (của thuật toán sắp xếp)       |
| stable                             | tính ổn định (của thuật toán sắp xếp)       |
| swap                               | phép hoán đổi                               |
| comparison-based sorting algorithm | thuật toán sắp xếp chỉ sử dụng phép so sánh |
| insertion sort                     | sắp xếp chèn                                |
| selection sort                     | sắp xếp lựa chọn                            |
| bubble sort                        | sắp xếp nổi bọt                             |
| merge sort                         | sắp xếp trộn                                |
| quick sort                         | sắp xếp nhanh                               |
| partition                          | phân đoạn                                   |
| pivot element                      | phản tử chốt                                |
| median element                     | phản tử trung vị                            |
| perfect partition                  | phân đoạn hoàn hảo                          |
| unbalanced partition               | phân đoạn không cân bằng                    |
| heap sort                          | sắp xếp vùn đống                            |
| heap property                      | tính chất đống                              |
| max-heap                           | đống max                                    |
| min-heap                           | đống min                                    |
| priority queue                     | hàng đợi có ưu tiên                         |
| counting sort                      | sắp xếp đếm                                 |
| radix sort                         | sắp xếp theo cơ số                          |
| bucket sort                        | sắp xếp phân cụm                            |

### Chương 6

|                                    |                                          |
|------------------------------------|------------------------------------------|
| linear search or sequential search | tìm kiếm tuyến tính hay tìm kiếm tuần tự |
| binary search                      | tìm kiếm nhị phân                        |
| binary search tree                 | cây nhị phân tìm kiếm                    |

|                          |                           |
|--------------------------|---------------------------|
| dynamic set              | tập động                  |
| satellite data           | thông tin đi kèm          |
| queries                  | các truy vấn              |
| modifying operations     | các thao tác biến đổi     |
| predecessor              | kẻ cận trước              |
| successor                | kẻ cận sau                |
| binary tree structure    | cấu trúc cây nhị phân     |
| left rotation            | quay trái                 |
| right rotation           | quay phải                 |
| double rotation          | quay kép                  |
| right-left rotation      | quay phải rồi quay trái   |
| left-right rotation      | quay trái rồi quay phải   |
| string searching problem | bài toán tìm kiếm xâu mẫu |
| shift                    | phép trượt                |
| prefix                   | tiền tố                   |
| suffix                   | hậu tố                    |
| prefix function          | hàm tiền tố               |
| mapping and hashing      | ánh xạ và băm             |
| dictionaries             | (các) từ điển             |
| direct addressing        | địa chỉ trực tiếp         |
| actual keys              | khóa thực có              |
| hash function            | hàm băm                   |
| collision                | xung đột                  |
| open addressing method   | phương pháp địa chỉ mở    |
| chaining method          | phương pháp tạo chuỗi     |
| linear probing           | dò tuyến tính             |
| quadratic probing        | dò toàn phương            |
| double hashing           | băm kép                   |
| simple uniform hashing   | băm phân bố đều           |
| division method          | phương pháp chia          |
| multiplication method    | phương pháp nhân          |

### Chương 7

|              |            |
|--------------|------------|
| graph        | đồ thị     |
| simple graph | đơn đồ thị |
| multigraph   | đa đồ thị  |
| vertex       | đỉnh       |

|                                      |                                          |
|--------------------------------------|------------------------------------------|
| vertice set                          | tập đỉnh                                 |
| edge                                 | cạnh                                     |
| edge set                             | tập cạnh                                 |
| isolated vertice                     | đỉnh cô lập                              |
| pendant vertice                      | đỉnh treo                                |
| path                                 | đường đi                                 |
| simple path                          | đường đi đơn                             |
| cycle                                | chu trình                                |
| connected component                  | thành phần liên thông                    |
| cut vertex                           | đỉnh rẽ nhánh                            |
| bridge                               | cầu                                      |
| strongly connected                   | liên thông mạnh                          |
| weakly connected                     | liên thông yếu                           |
| adjacency matrix                     | ma trận kè                               |
| sparse graph                         | đồ thị thưa                              |
| adjacency list                       | danh sách kè                             |
| graph searching (or graph traversal) | duyệt đồ thị                             |
| breadth first search                 | tìm kiếm theo chiều rộng                 |
| depth first search                   | tìm kiếm theo chiều sâu                  |
| parenthesis structure                | cấu trúc lồng nhau                       |
| tree edge                            | cạnh của cây                             |
| back edge                            | cạnh ngược                               |
| forward edge                         | cạnh tới                                 |
| cross edge                           | cạnh vòng                                |
| directed acyclic graph               | đồ thị có hướng không có chu trình       |
| topological sort                     | sắp xếp tópô                             |
| chromatic number                     | sắc số                                   |
| transitive closure                   | bao đóng truyền ứng                      |
| spanning tree                        | cây khung (cây bao trùm)                 |
| minimum spanning tree problem        | bài toán cây khung nhỏ nhất              |
| disjoint set data structures         | cấu trúc dữ liệu các tập không giao nhau |
| shortest path problem                | bài toán đường đi ngắn nhất              |
| negative cycle                       | chu trình âm                             |

# CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

NHÀ XUẤT BẢN BÁCH KHOA – HÀ NỘI  
Ngõ 17 Tạ Quang Bửu – Hai Bà Trưng – Hà Nội  
ĐT: 04. 38684569; Fax: 04. 38684570  
[www.nxbbk.hust.edu.vn](http://www.nxbbk.hust.edu.vn)

*Chịu trách nhiệm xuất bản và nội dung:*  
*Giám đốc – Tổng Biên tập: GVC. TS. PHÙNG LAN HƯƠNG*

*Phản biện:* TS. ĐỖ PHAN THUẬN  
TS. NGUYỄN KHANH VĂN  
*Biên tập:* ĐỖ THANH THÙY  
*Sửa bản in:* VŨ THỊ HẰNG  
*Trình bày bìa:* TRIỆU VĂN NAM

---

In 500 cuốn khổ 16 x 24 cm tại Xưởng thực hành kỹ thuật in ĐHBK – Hà Nội.  
Số đăng ký KHXB: 58 – 2013/CXB/28 – 01/BKHN; ISBN: 9786049112782, do Cục  
Xuất bản cấp ngày 10/1/2013.  
Số QĐXB: 181/QĐ – ĐHBK – BKHN ngày 18/7/2013.  
In xong và nộp lưu chiểu quý III năm 2013.