CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

MỘT SỐ VẤN ĐỀ CƠ BẢN CỦA CẤU TRÚC DỮ LIỆU VÀO GIẢI THUẬT

Mục lục

- Khái niệm về kiểu và cấu trúc dữ liệu
- Thuật toán và một số vấn đề liên quan
- Phương pháp biểu diễn thuật toán
- Độ phức tạp của thuật toán
- Ví dụ mở đầu

Khái niệm về kiểu và cấu trúc dữ liệu

- Cấu trúc dữ liệu: Cách tổ chức dữ liệu trên máy tính để thuận tiện cho các tính toán
- Thuật toán: Một thủ tục xác định bao gồm 1 dãy các thao tác tính toán để thu được kết quả đầu ra với mỗi dữ liệu đầu vào xác định

Khái niệm về kiểu và cấu trúc dữ liệu

- Kiểu dữ liệu
  - Tập các giá trị
  - Tập các phép toán thao tác trên các giá trị
  - Biểu diễn và cài đặt trên máy tính
- Ví dụ kiểu số nguyên int
  - Tập giá trị -2^{31} đến $2^{31} - 1$
  - Tập phép toán: +, -, *, / mod
- Kiểu dữ liệu trừu tượng (Abstract Data Type - ADT)
  - Tập các giá trị
  - Tập tác thao tác
  - Chưa quan tâm đến biểu diễn và cài đặt trên máy tính

Thuật toán và một số vấn đề liên quan

- Các bài toán tối ưu tổ hợp
  - Lập lịch, lập lộ trình vận tải, thời gian biểu, lập kế hoạch sản xuất,. . .
- Xử lý ảnh, thị giác máy tính, học máy, phân tích dữ liệu
- Cơ sở dữ liệu
- Các bài toán quản lý bộ nhớ, lập lịch thực hiện tiến trình trong các hệ điều hành
- …

Phương pháp biểu diễn thuật toán

- Giả mã: Ngôn ngữ để biểu diễn thuật toán một cách thân thiện, ngắn gọn mà không cần viết chương trình (bằng 1 ngôn ngữ lập trình cụ thể)

```
max(a[1..n]) {
  m = a[1];
  for i = 2 to n do {
    if(m < a[i])
      m = a[i];
  }
  return m;
}
```

Độ phức tạp của thuật toán

- Phân tích độ phức tạp của thuật toán
  - Thời gian
  - Bộ nhớ
- Thực nghiệm
  - Viết chương trình hoàn chỉnh bằng một ngôn ngữ lập trình
  - Chạy chương trình với các bộ dữ liệu khác nhau
  - Đo thời gian thực hiện chương trình và vẽ biểu đồ
- Nhược điểm của phương pháp thực nghiệm
  - Cần lập trình
  - Kết quả thực nghiệm không bao quát được hết các trường hợp
  - Cùng 1 chương trình nhưng chạy trên 1 cấu hình máy khác nhau sẽ cho kết quả khác nhau

Độ phức tạp của thuật toán

- Phân tích độ phức tạp thời gian tính như là một hàm của kích thước dữ liệu đầu vào
- Kích thước dữ liệu đầu vào
  - Số bít cần để biểu diễn dữ liệu đầu vào
  - Mức cao: số phần tử của dãy, ma trận đầu vào
- Câu lệnh cơ bản
  - Thực hiện trong thời gian hằng số và không phụ thuộc kích thước dữ liệu đầu vào
  - Ví dụ: các phép toán cơ bản: +, -, *, /, so sánh,…

Độ phức tạp của thuật toán

- Phân tích độ phức tạp thời gian
  - Đếm số câu lệnh cơ bản được thực hiện như một hàm của kích thước dữ liệu đầu vào

```
sum(a[1..n]) {
  s = a[1];
  for i = 2 to n do {
    s = s + a[i];
  }
  return s;
}
```

Số câu lệnh cơ bản của hàm **`sum(a[1..n])`** là cỡ **`n`**

Độ phức tạp của thuật toán

- Ký hiệu tiệm cận (O lớn)
  - Dùng để viết ngắn gọn hàm độ phức tạp thời gian
  - Thể hiện độ tăng của hàm độ phức tạp thời gian theo kích thước dữ liệu đầu vào
  - $f(n) = O(g(n))$: độ tăng của $f(n)$ không vượt quá độ tăng của $g(n)$, nói cách khác $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
  - $f(n) = \Theta(g(n))$: độ tăng của $f(n)$ bằng độ tăng của $g(n)$, nói cách khác $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
  - Ví dụ
    - $2n^2 + 10^6 n + 5 = O(n^2)$
    - $10^3 n\log n + 2n + 10^4 = O(n\log n)$
    - $10^3 n\log n + 2n + 10^4 = O(n^3)$
    - $2^n + n^{10} + 1 = O(2^n)$
    - Độ phức tạp về thời của hàm **`sum(a[1..n])`** là O(**n**)

Độ phức tạp của thuật toán

- Câu lệnh cơ bản trong hàm **sort** là khối các câu lệnh so sanh và đổi chỗ 2 phần tử **a[i]** và **a[j]**
- Số câu lệnh cơ bản là **n(n-1)/2 =** $O(n^2)$
- Độ phức tạp về thời của hàm **sort(a[1..n])** là $O(n^2)$

```
sort(a[1..n]) {
  for i = 1 to n-1 do
    for j = i+1 to n do
      if(a[i] > a[j]) {
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
}
```

Độ phức tạp của thuật toán

- Trong nhiều trường hợp, với cùng kích thước dữ liệu đầu vào, các bộ dữ liệu khác nhau sẽ cho độ phức tạp về thời gian tính khác nhau
  - Thời gian tính trong tình huống tồi nhất (worst-case time complexity): Bộ dữ liệu cho thời gian tính lâu nhất
  - Thời gian tính trong tình huống tốt nhất (best-case time complexity): Bộ dữ liệu cho thời gian tính nhanh nhất
  - Thời gian tính trung bình: trung bình về thời gian tính của tất cả các bộ dữ liệu đầu vào với cùng kích thước xác định

Ví dụ mở đầu

- Bài toán dãy con cực đại
  - Đầu vào: Cho dãy $a = a_1, a_2, \ldots, a_n$. Một dãy con của a là dãy gồm một số liên tiếp các phần tử $a_i, a_{i+1}, \ldots, a_j$ và có trọng số là $a_i + a_{i+1} + \ldots + a_j$
  - Đầu ra: Tìm dãy con có trọng số lớn nhất của dãy a
  - Ví dụ: dãy a = 2, 4, -7, 5, 7, -10, 4, 3, dãy con cực đại của a là dãy 5, 7

Ví dụ mở đầu

- Thuật toán 1 (**subseq1**)
  - Duyệt tất cả các dãy con
  - Tính tổng các phần tử của mỗi dãy con và giữ lại dãy con có trọng số lớn nhất
  - Độ phức tạp $O(n^3)$

```
subseq1(a[1..n]){
  max = -∞;
  for i = 1 to n do{
    for j = i to n do{
      s = 0;
      for k = i to j do
        s = s + a[k];
      max = s > max ? s : max;
    }
  }
  return max;
}
```

Ví dụ mở đầu

- Thuật toán 2 (**subseq2**)
  - Duyệt tất cả các dãy con
  - Tính tổng các phần từ của mỗi dãy con và giữ lại dãy con có trọng số lớn nhất
  - Tận dụng tính chất liên tiếp để tính tổng dãy con dựa vào dãy con trước đó
  - Độ phức tạp $O(n^2)$

```
subseq1(a[1..n]){
  max = -∞;
  for i = 1 to n do{
    s = 0;
    for j = i to n do{
      s = s + a[k];
      max = s > max ? s : max;
    }
  }
  return max;
}
```

Ví dụ mở đầu

- Thuật toán 3 (**subseq3**)
  - Chia dãy đã cho thành 2 dãy con độ dài đều nhau
  - Tìm dãy con lớn nhất của dãy bên phải
  - Tìm dãy con lớn nhất của dãy bên trái
  - Tìm dãy con lớn nhất có 1 phần thuộc dãy con bên phải và 1 phần thuộc dãy con bên trái
  - Độ phức tạp O(*n*log*n*)

```
subseq3(a[1..n], l, r){
  if(l = r) return a[r];
  i = (l+r)/2;
  ml = subseq3(a,l,i);
  mr = subseq3(a,i+1,r);
  mlr = maxLeft(a,l,i) +
        maxRight(a,i+1,r);
  max = mlr;
  max = max < ml ? ml : max;
  max = max < mr ? mr : max;
  return max;
}
```

Ví dụ mở đầu

```
maxLeft(a[1..n], l, r){
  max = -∞;
  s = 0;
  for i = r downto l do{
    s = s + a[i];
    if(s > max) max = s;
  }
  return max;
}
```

```
maxRight(a[1..n], l, r){
  max = -∞;
  s = 0;
  for i = l to r do{
    s = s + a[i];
    if(s > max) max = s;
  }
  return max;
}
```

Ví dụ mở đầu

- Thuật toán 4 (**subseq4**)
  - Dựa trên quy hoạch động
  - S_i là trọng số dãy con lớn nhất của dãy $a_1, \ldots, a_i$ mà phần tử cuối cùng là a_i $(\forall i = 1,\ldots, n)$
  - $S_1 = a_1$
  - $S_i = \begin{cases} S_{i-1} + a_i, \text{ nếu } S_{i-1} > 0 \\ a_i, \text{ nếu } S_{i-1} \leq 0 \end{cases}$
  - Độ phức tạp O(n)

```
subseq4(a[1..n]){
  max = -∞;
  s = a[1];
  for i = 2 to n do{
    if(s > 0)
      s = s + a[i];
    else s = a[i];
    if(s > max) max = s;
  }
  return max;
}
```

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

CÁC LƯỢC ĐỒ THUẬT TOÁN QUAN TRỌNG

Các lược đồ thuật toán quan trọng

- Đệ quy
- Đệ quy có nhớ
- Đệ quy quay lui
- Nhánh và cận
- Tham lam
- Chia để trị
- Quy hoạch động

Đệ quy

- Một chương trình con (thủ tục/hàm) đưa ra lời gọi đến chính nó nhưng với dữ liệu đầu vào nhỏ hơn
- Tình huống cơ sở
  - Dữ liệu đầu vào nhỏ đủ để đưa ra kết quả một cách trực tiếp mà không cần đưa ra lời gọi đệ quy
- Tổng hợp kết quả
  - Kết quả của chương trình còn được xây dựng từ kết quả của lời gọi đệ quy và một số thông tin khác

Ví dụ: tổng 1 + 2 + … + n

```
int sum(int n) {
  if (n <= 1) return  1;
  int s = sum(n-1);
  return n + s;
}
```

Đệ quy

- Ví dụ: hằng số tổ hợp

$$C(k, n) = \frac{n!}{k!(n-k)!}$$

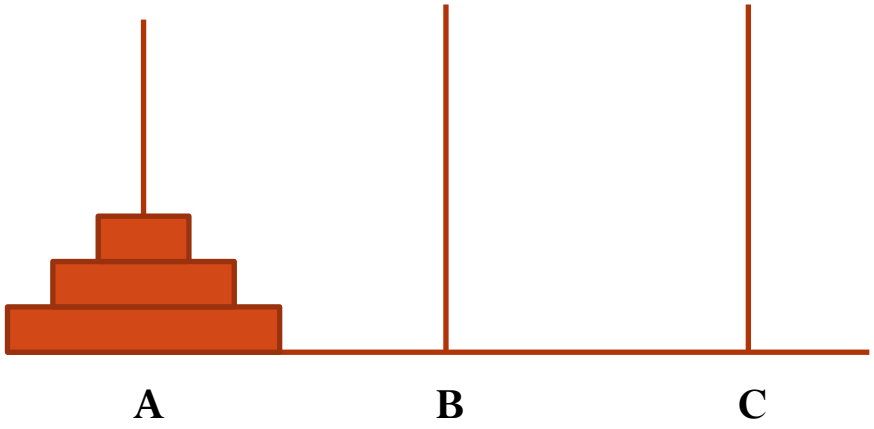
- $C(k, n) = C(k\text{-}1, n\text{-}1) + C(k, n\text{-}1)$
- Trường hợp cơ sở:

$$C(0, n) = C(n, n) = 1$$

```
int C(int k, int n) {
  if (k == 0 || k == n)
    return  1;
  int C1 = C(k-1,n-1);
  int C2 = C(k,n-1);
  return C1 + C2;
}
```

Đệ quy

- Bài toán tháp Hà Nội
  - Có n đĩa với kích thước khác nhau và 3 cọc A, B, C
  - Ban đầu n đĩa nằm ở cọc A theo thứ tự đĩa nhỏ nằm trên và đĩa lớn nằm dưới
  - Tìm cách chuyển n đĩa này từ cọc A sang cọc B, sử dụng cọc C làm trung gian theo nguyên tắc
    - Mỗi lần chỉ được chuyển 1 đĩa trên cùng từ 1 cọc sang cọc khác
    - Không được phép để xảy ra tình trạng đĩa to nằm bên trên đĩa nhỏ

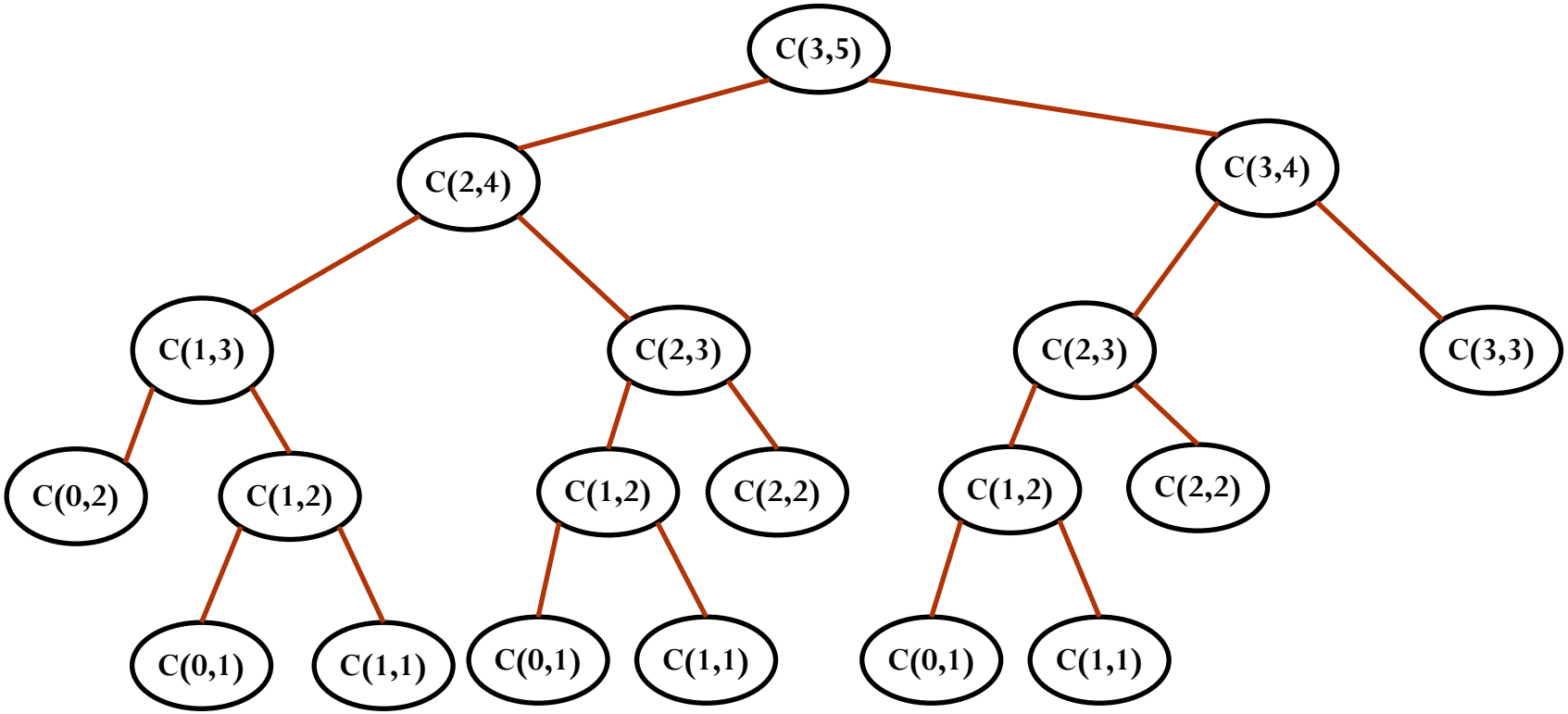


- Lời giải
- B1: A → B
- B2: A → C
- B3: B → C
- B4: A → B
- B5: C → A
- B6: C → B
- B7: A → B

Đệ quy

```
void move(int n, char A, char B, char C) {
  if(n == 1) {
    printf("Move 1 disk from %c to %c", A, B)
  }else{
    move(n-1, A, C, B);
    move(1, A, B, C);
    move(n-1, C, B, A);
  }
}
void main() {
   int n = 3;
   move(n, 'A', 'B', 'C');
}
```

Đệ quy


C(3,5)
C(2,4)
C(3,4)
C(1,3)
C(2,3)
C(2,3)
C(3,3)
C(0,2)
C(1,2)
C(1,2)
C(2,2)
C(1,2)
C(2,2)
C(0,1)
C(1,1)
C(0,1)
C(1,1)
C(0,1)
C(1,1)

Đệ quy có nhớ

- Khắc phục tình trạng một chương trình con với tham số xác định được gọi đệ quy nhiều lần
- Sử dụng bộ nhớ để lưu trữ kết quả của một chương trình con với tham số cố định
- Bộ nhớ được khởi tạo với giá trị đặc biệt để ghi nhận mỗi chương trình con chưa được gọi lần nào
- Địa chỉ bộ nhớ sẽ được ánh xạ với các giá trị tham số của chương trình con

```
int m[MAX][MAX];

int C(int k, int n) {
  if (k == 0 || k == n)
    m[k][n] = 1;
  if(m[k][n] < 0){
    m[k][n] = C(k-1,n-1) +
              C(k,n-1);
  }
  return m[k][n];
}

int main() {
  for(int i = 0; i < MAX; i++)
    for(int j = 0; j < MAX; j++)
       m[i][j] = -1;
  printf("%d\n",C(16,32));
}
```

Đệ quy quay lui

- Áp dụng để giải các bài toán liệt kê, bài toán tối ưu tổ hợp
- $A = \{(x_1, x_2, \ldots, x_n) \mid x_i \in A_i, \forall\, i = 1,\ldots, n\}$
- Liệt kê tất cả các bộ $x \in A$ thoả mãn một thuộc tính P nào đó
- Thủ tục TRY(k):
  - Thử các giá trị v có thể gán cho x_k mà không vi phạm thuộc tính P
  - Với mỗi giá trị hợp lệ v:
    - Gán v cho x_k
    - Nếu $k < n$: gọi đệ quy TRY(k+1) để thử tiếp giá trị cho x_{k+1}
    - Nếu $k = n$: ghi nhận cấu hình

Đệ quy quay lui

```
TRY(k)
  Begin
    Foreach v thuộc A_k
     if check(v,k) /* kiểm tra xem v có hợp lệ không */
       Begin
         x_k = v;
         if(k = n) ghi_nhan_cau_hinh;
         else TRY(k+1);
       End
  End
Main()
Begin
  TRY(1);
End
```

Đệ quy quay lui: liệt kê xâu nhị phân

- Mô hình hoá cấu hình:
  - Mảng `x[n]` trong đó `x[i]` $\in \{0,1\}$ là bít thứ `i` của xâu nhị phân

    (`i= 0, . . ., n-1`)

```
void printSolution(){
  for(int k = 0; k < n; k++)
    printf("%d",x[k]);
  printf("\n");
}

int TRY(int k) {
  for(int v = 0; v <= 1; v++){
    x[k] = v;
    if(k == n-1) printSolution();
    else TRY(k+1);
  }
}

int main() {
  TRY(0);
}
```

Đệ quy quay lui: liệt kê xâu nhị phân với ràng buộc

- Liệt kê các xâu nhị phân sao cho không có 2 bit 1 nào đứng cạnh nhau
- Mô hình hoá cấu hình:
  - Mảng `x[n]` trong đó `x[i]` $\in \{0,1\}$ là bít thứ `i` của xâu nhị phân

    (`i= 1, . . ., n`)
  - Thuộc tính *P*: không có 2 bít 1 nào đứng cạnh nhau

```
int TRY(int k) {
  for(int v = 0; v <= 1; v++){
    if(x[k-1] + v < 2){
      x[k] = v;
      if(k == n)
        printSolution();
      else TRY(k+1);
    }
  }
}

int main() {
  x[0] = 0;
  TRY(1);
}
```

Đệ quy quay lui: liệt kê các tổ hợp

- Liệt kê các tổ hợp chập k của 1, 2, …, n
- Mô hình hoá cấu hình:
  - Mảng $x[k]$ trong đó $x[i] \in \{1, \ldots, n\}$ là phần tử thứ i của cấu hình tổ hợp ($i = 1, \ldots, k$)
  - Thuộc tính P: $x[i] < x[i+1]$, với mọi $i = 1, 2, \ldots, k-1$

```
int TRY(int i) {
  for(int v = x[i-1]+1; v <= n-k+i;
          v++){
     x[i] = v;
     if(i == k)
        printSolution();
      else TRY(i+1);
     }
}

int main() {
  x[0] = 0;
  TRY(1);
}
```

Đệ quy quay lui: liệt kê các hoán vị kỹ thuật đánh dấu

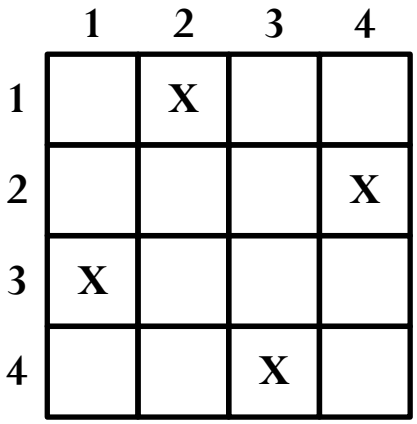
- Liệt kê các hoán vị của 1, 2, …, n
- Mô hình hoá cấu hình:
  - Mảng $x[1,\ldots,n]$ trong đó $x[i] \in \{1,\ldots,n\}$ là phần tử thứ i của cấu hình hoán vị ($i = 1,\ldots,n$)
  - Thuộc tính P:
    - $x[i] \neq x[j]$, với mọi $1 \leq i < j \leq n$
  - Mảng đánh dấu $m[v]$ = true (false) nếu giá trị v đã xuất hiện (chưa xuất hiện) trong cấu hình bộ phận, với mọi $v = 1, \ldots, n$

```
void TRY(int i) {
  for(int v = 1; v <= n; v++){
    if(!m[v]) {
      x[i] = v;
      m[v] = true; // đánh dấu
      if(i == n)
        printSolution();
      else TRY(i+1);
      m[v] = false;// khôi phục
    }
  }
}

void main() {
  for(int v = 1; v <= n; v++)
     m[v] = false;
  TRY(1);
}
```

Đệ quy quay lui: bài toán xếp hậu

- Xếp n quân hậu trên một bàn cờ quốc tế sao cho không có 2 quân hậu nào ăn được nhau
- Mô hình hoá
  - $x[1, \ldots, n]$ trong đó $x[i]$ là hàng của quân hậu xếp trên cột i, với mọi $i = 1, \ldots, n$
  - Thuộc tính P
    - $x[i] \neq x[j]$, với mọi $1 \leq i < j \leq n$
    - $x[i] + i \neq x[j] + j$, với mọi $1 \leq i < j \leq n$
    - $x[i] - i \neq x[j] - j$, với mọi $1 \leq i < j \leq n$



Lời giải $x = (3, 1, 4, 2)$

Đệ quy quay lui: bài toán xếp hậu

```
int check(int v, int k) {
  // kiểm tra xem v có thể gán được
  // cho x[k] không
  for(int i = 1; i <= k-1; i++) {
    if(x[i] == v) return 0;
    if(x[i] + i == v + k) return 0;
    if(x[i] - i == v - k) return 0;
  }
  return 1;
}
```

```
void TRY(int k) {
  for(int v = 1; v <= n; v++) {
    if(check(v,k)) {
      x[k] = v;
      if(k == n) printSolution();
      else TRY(k+1);
    }
  }
}
void main() {
    TRY(1);
}
```

Đệ quy quay lui: bài toán sudoku

- Điền các chữ số từ 1 đến 9 vào các ô trong bảng vuông 9x9 sao cho trên mỗi hàng, mỗi cột và mỗi bảng vuông con 3x3 đều có mặt đầy đủ 1 chữ số từ 1 đến 9

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	1	4	3	6	5	8	9	7
3	6	5	8	9	7	2	1	4
8	9	7	2	1	4	3	6	5
5	3	1	6	4	2	9	7	8
6	4	2	9	7	8	5	3	1
9	7	8	5	3	1	6	4	2

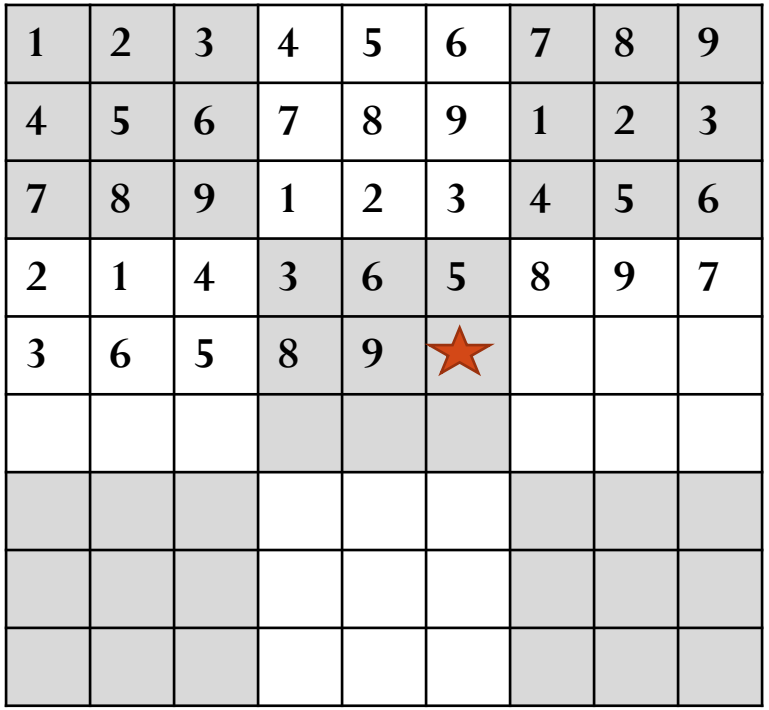
Đệ quy quay lui: bài toán sudoku

- Mô hình hoá
  - Mảng 2 chiều $x[0..8, 0..8]$
  - Thuộc tính P
    - $x[i, j_2] \neq x[i, j_1]$, với mọi $i = 0,\ldots,8$, và $0 \leq j_1 < j_2 \leq 8$
    - $x[i_1, j] \neq x[i_2, j]$, với mọi $j = 0,\ldots,8$, và $0 \leq i_1 < i_2 \leq 8$
    - $x[3I+i_1, 3J+j_1] \neq x[3I+i_2, 3J+j_2]$, với mọi $I, J = 0,\ldots, 2$, và $i_1, j_1, i_2, j_2 \in \{0,1, 2\}$ sao cho $i_1 \neq i_2$ hoặc $j_1 \neq j_2$

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	1	4	3	6	5	8	9	7
3	6	5	8	9	7	2	1	4
8	9	7	2	1	4	3	6	5
5	3	1	6	4	2	9	7	8
6	4	2	9	7	8	5	3	1
9	7	8	5	3	1	6	4	2

Đệ quy quay lui: bài toán sudoku

- Thứ tự duyệt: từ ô (0,0), theo thứ tự từ trái qua phải và từ trên xuống dưới



Đệ quy quay lui: bài toán sudoku

```
bool check(int v, int r, int c){
  for(int i = 0; i <= r-1; i++)
    if(x[i][c] == v) return false;
  for(int j = 0; j <= c-1; j++)
    if(x[r][j] == v) return false;
  int I = r/3;
  int J = c/3;
  int i = r - 3*I;
  int j = c - 3*J;
  for(int i1 = 0; i1 <= i-1; i1++)
    for(int j1 = 0; j1 <= 2; j1++)
      if(x[3*I+i1][3*J+j1] == v)
        return false;
  for(int j1 = 0; j1 <= j-1; j1++)
    if(x[3*I+i][3*J+j1] == v)
       return false;
  return true;
}
```

```
void TRY(int r, int c){
  for(int v = 1; v <= 9; v++){
    if(check(v,r,c)){
      x[r][c] = v;
      if(r == 8 && c == 8){
        printSolution();
      }else{
        if(c == 8) TRY(r+1,0);
        else TRY(r,c+1);
      }
    }
  }
}

void main(){
  TRY(0,0);
}
```

Đệ quy quay lui: bài tập

Cho số nguyên dương M, N và N số nguyên dương A_1, A_2, …, A_N. Liệt kê các nghiệm nguyên dương của phương trình

$$A_1X_1 + A_2X_2 + \ldots + A_NX_N = M$$

Đệ quy quay lui: bài tập

1. Liệt kê tất cả các cách chọn ra *k* phần tử từ 1, 2, …, *n* sao cho không có 2 phần tử nào đứng cạnh nhau cũng được chọn
2. Liệt kê tất cả các cách chọn ra *k* phần tử từ 1, 2, …, *n* sao cho không có 3 phần tử nào đứng cạnh nhau cùng đồng thời được chọn
3. Liệt kê tất cả các xâu nhị phân độ dài *n* trong đó không có 3 bit 1 nào đứng cạnh nhau
4. Liệt kê tất cả các cách phân tích số *N* thành tổng của các số nguyên dương
5. Giải bài toán Sudoku, xếp hậu sử dụng kỹ thuật đánh dấu
6. Giải bài toán Sudoku với 1 số ô đã được điền từ trước

Thuật toán nhánh và cận

- Bài toán tối ưu tổ hợp
  - Phương án $x = (x_1, x_2, \ldots, x_n)$ trong đó $x_i \in A_i$ cho trước
  - Phương án thoả mãn ràng buộc C
  - Hàm mục tiêu $f(x)$ → min (max)

Thuật toán nhánh và cận

- Bài toán người du lịch
  - Có n thành phố 1, 2, …, n. Chi phí đi từ thành phố i đến thành phố j là $c(i, j)$. Hãy tìm một hành trình xuất phát từ thành phố thứ 1, đi qua các thành phố khác, mỗi thành phố đúng 1 lần và quay về thành phố 1 với tổng chi phí nhỏ nhất
- Mô hình hoá
  - Phương án $x = (x_1, x_2, \ldots, x_n)$ trong đó $x_i \in \{1, 2, \ldots, n\}$
  - Ràng buộc C: $x_i \neq x_j$, $\forall\ 1 \leq i < j \leq n$
  - Hàm mục tiêu

$$f(x) = c(x_1, x_2) + c(x_2, x_3) + \ldots + c(x_n, x_1) \rightarrow \min$$

Thuật toán nhánh và cận

- Duyệt toàn bộ:
  - Liệt kê tất cả các phương án bằng phương pháp đệ quy quay lui
  - Với mỗi phương án, tính toán hàm mục tiêu
  - Giữ lại phương án có hàm mục tiêu nhỏ nhất

```
TRY(k)
  Begin
    Foreach v thuộc A_k
     if check(v,k)
       Begin
         x_k = v;
         if(k = n)
           ghi_nhan_cau_hinh;
           cập nhật kỷ lục f*;
         else TRY(k+1);
       End
  End
Main()
Begin
  TRY(1);
End
```

Thuật toán nhánh và cận

- Duyệt nhánh và cận:
  - Phương án bộ phận (a_1,…, a_k) trong đó a_1 gán cho x_1,… a_k gán cho x_k
  - Phương án (a_1,…, a_k, b_{k+1},…,b_n) là một phương án đầy đủ được phát triển từ (a_1,…,a_k) trong đó b_{k+1} gán cho x_{k+1},…, b_n được gán cho x_n
  - Với mỗi phương án bộ phận (x_1,…, x_k), hàm cận dưới $g(x_1,\ldots, x_k)$ có giá trị không lớn hơn giá trị hàm mục tiêu của phương án đầy đủ phát triển từ (x_1,…,x_k)
  - Nếu $g(x_1,\ldots,x_k) \geq f^*$ thì không phát triển lời giải từ (x_1,…,x_k)

```
TRY(k) {
  Foreach v thuộc A_k
      if check(v,k) {
        x_k = v;
        if(k = n) {
            ghi_nhan_cau_hinh;
            cập nhật kỷ lục f*;
         } {
            if g(x1,…,x_k) < f*
              TRY(k+1);
        }
      }
}
Main()
{ f* = ∞;
  TRY(1);
}
```

Thuật toán nhánh và cận giải bài toán người du lịch

- c_m là chi phí nhỏ nhất trong số các chi phí đi giữa 2 thành phố khác nhau
- Phương án bộ phận $(x_1,\ldots, x_k)$
  - Chi phí bộ phận $f = c(x_1, x_2) + c(x_2, x_3) + \ldots + c(x_{k-1}, x_k)$
  - Hàm cận dưới

    $g(x_1,\ldots, x_k) = f + c_m \times (n-k+1)$

Thuật toán nhánh và cận giải bài toán người du lịch

```
void TRY(int k){
  for(int v = 1; v <= n; v++){
    if(marked[v] == false){
      a[k] = v;
      f = f + c[a[k-1]][a[k]];
      marked[v] = true;
      if(k == n){
        process();
      }else{
        int g = f + cmin*(n-k+1);
        if(g < f_min)
          TRY(k+1);
      }
      marked[v] = false;
      f = f - c[a[k-1]][a[k]];
    }
  }
}
```

```
void process() {
  if(f + c[x[n]][x[1]] < f_min){
    f_min = f + c[x[n]][x[1]];
  }
}
void main() {
  f_min = 9999999999;
  for(int v = 1; v <= n; v++)
    marked[v] = false;
  x[1] = 1; marked[1] = true;
  TRY(2);
}
```

Thuật toán tham lam

- Được ứng dụng để giải một số bài toán tối ưu tổ hợp
- Đơn giản và tự nhiên
- Quá trình tìm lời giải diễn ra qua các bước
- Tại mỗi bước, ra quyết định dựa trên các thông tin hiện tại mà không quan tâm đến ảnh hưởng của nó trong tương lai
- Dễ đề xuất, cài đặt
- Thường không tìm được phương án tối ưu toàn cục

Thuật toán tham lam

- Lời giải được biểu diễn bởi tập S
- C biểu diễn các ứng cử viên
- select(C): chọn ra ứng cử viên tiềm năng nhất
- solution(S): trả về true nếu S là lời giải của bài toán
- feasible(S): trả về true nếu S không vi phạm ràng buộc nào của bài toán

```
Greedy() {
  S = {};
  while C ≠ ∅ and
        not solution(S){
    x = select(C);
    C = C \ {x};
    if feasible(S ∪ {x}) {
      S = S ∪ {x};
    }
  }
  return S;
}
```

Thuật toán tham lam

- Bài toán tập đoạn không giao nhau
  - Đầu vào: cho tập các đoạn thẳng $X = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ trong đó $a_i < b_i$ là toạ độ đầu mút của đoạn thứ i trên đường thẳng, với mọi $i = 1, \ldots, n$.
  - Đầu ra: Tìm tập con các đoạn đôi một không giao nhau có số đoạn lớn nhất

Thuật toán tham lam

- Tham lam 1
  - Sắp xếp các đoạn theo thứ tự không giảm của a_i được danh sách L
  - Lặp lại thao tác sau cho đến khi L không còn phần tử nào
    - Chọn đoạn (a_c, b_c) đầu tiên trong L và loại nó ra khỏi L
    - Nếu (a_c, b_c) không có giao nhau với đoạn nào trong lời giải thì đưa (a_c, b_c) vào lời giải

```
Greedy1() {
  S = {};
  L = Sắp xếp các đoạn trong X
   theo thứ tự không giảm của ai;
  while (X ≠ ∅) {
    (ac,bc) = chọn đoạn đầu tiên
       trong L;
    Loại bỏ (ac,bc) khỏi L;
    if feasible(S ∪ {(ac,bc)}) {
      S = S ∪ {(ac,bc)};
    }
  }
  return S;
}
```

Thuật toán tham lam

- Tham lam 1 không đảm bảo cho lời giải tối ưu, ví dụ

$$X = \{(1,11), (2,5), (6,10)\}$$

  - Greedy 1 sẽ cho lời giải là {(1,11)} trong khi lời giải tối ưu là {(2,5), (6,10)}

Thuật toán tham lam

- Tham lam 2
  - Sắp xếp các đoạn theo thứ tự không giảm của độ dài $b_i - a_i$ được danh sách L
  - Lặp lại thao tác sau cho đến khi L không còn phần tử nào
    - Chọn đoạn (a_c, b_c) đầu tiên trong L và loại nó ra khỏi L
    - Nếu (a_c, b_c) không có giao nhau với đoạn nào trong lời giải thì đưa (a_c, b_c) vào lời giải

```
Greedy2() {
  S = {};
  L = Sắp xếp các đoạn trong X
    theo thứ tự không giảm của
       bi- ai;
  while (X ≠ ∅) {
    (ac,bc) = chọn đoạn đầu tiên
       trong L;
    Loại bỏ (ac,bc) khỏi L;
    if feasible(S ∪ {(ac,bc)}) {
      S = S ∪ {(ac,bc)};
    }
  }
  return S;
}
```

Thuật toán tham lam

- Tham lam 2 không đảm bảo cho lời giải tối ưu, ví dụ

$$X = \{(1,5), (4,7), (6,11)\}$$

  - Greedy 1 sẽ cho lời giải là {(4,7)} trong khi lời giải tối ưu là {(1,5), (6,11)}

Thuật toán tham lam

- Tham lam 3
  - Sắp xếp các đoạn theo thứ tự không giảm của b_i được danh sách L
  - Lặp lại thao tác sau cho đến khi L không còn phần tử nào
    - Chọn đoạn (a_c, b_c) đầu tiên trong L và loại nó ra khỏi L
    - Nếu (a_c, b_c) không có giao nhau với đoạn nào trong lời giải thì đưa (a_c, b_c) vào lời giải
- Tham lam 3 đảm bảo cho lời giải tối ưu

```
Greedy3() {
  S = {};
  L = Sắp xếp các đoạn trong X
   theo thứ tự không giảm của bi;
  while (X ≠ ∅) {
    (ac,bc) = chọn đoạn đầu tiên
       trong L;
    Loại bỏ (ac,bc) khỏi L;
    if feasible(S ∪ {(ac,bc)}) {
      S = S ∪ {(ac,bc)};
    }
  }
  return S;
}
```

Thuật toán chia để trị

- Sơ đồ chung
  - Chia bài toán xuất phát thành các bài toán con độc lập nhau
  - Giải các bài toán con (đệ quy)
  - Tổng hợp lời giải của các bài toán con để xây dựng lời giải của bài toán xuất phát

Thuật toán chia để trị

- Bài toán tìm kiếm nhị phân: cho dãy $x[1..n]$ được sắp xếp theo thứ tự tăng dần và 1 giá trị y. Tìm chỉ số i sao cho $x[i] = y$

```
bSearch(x, start, finish, y) {
  if(start == finish) {
    if(x[start] == y)
      return start;
    else return -1;
  }else{
    m = (start + finish)/2;
    if(x[m] == y) return m;
    if(x[m] < y)
      return bSearch(x, m+1,finish,y);
    else
      return bSearch(x,start,m-1,y);
  }
}
```

Thuật toán chia để trị

- Bài toán dãy con cực đại
  - Đầu vào: dãy số nguyên $a_1, a_2, \ldots, a_n$
  - Đầu ra: tìm dãy con bao gồm các phần tử liên tiếp của dãy đã cho có tổng cực đại

```
int maxSeq(int* a, int l, int r){
  if(l == r) return a[l];
  int max;
  int mid = (l+r)/2;
  int mL = maxSeq(a,l,mid);
  int mR = maxSeq(a,mid+1,r);
  int mLR = maxLeft(a,l,mid) +
            maxRight(a,mid+1,r);
  max = mL;
  if(max < mR) max = mR;
  if(max < mLR) max = mLR;
  return max;
}
```

Thuật toán chia để trị

```
int maxLeft(int* a, int l, int r){
  int max = -9999999;
  int s = 0;
  for(int i = r; i >= l; i--){
    s += a[i];
    if(s > max) max = s;
  }
  return max;
}

int maxRight(int* a, int l, int r){
  int max = -99999999;
  int s = 0;
  for(int i = l; i <= r; i++){
    s += a[i];
    if(s > max) max = s;
  }
  return max;
}
```

```
void main() {
  readData();
  int rs = maxSeq(a,0,n-1);
  printf("result = %d",rs);
}
```

Thuật toán chia để trị (định lí thợ)

- Chia bài toán xuất phát thành a bài toán con, mỗi bài toán con kích thước n/b
- $T(n)$: thời gian của bài toán kích thước n
- Thời gian phân chia (dòng 4): $D(n)$
- Thời gian tổng hợp lời giải (dòng 6): $C(n)$
- Công thức truy hồi:

$$T(n) = \begin{cases} \Theta(1), & n \quad n0 \\ aT\left(\frac{n}{b}\right) + C(n) + D(n), & n > n0 \end{cases}$$

```
procedure D-and-C(n) {
1.  if(n ≤ n0)
2.    xử lý trực tiếp
3.  else{
4.    chia bài toán xuất phát
thành a bài toán con kích thước
n/b
5.    gọi đệ quy a bài toán con
6.    tổng hợp lời giải
7.  }
}
```

Thuật toán chia để trị

- Độ phức tạp của thuật toán chia để trị (định lí thợ)
- Công thức truy hồi:

  $T(n) = aT(n/b) + cn^k$, với các hằng số $a \geq 1$, $b > 1$, $c > 0$
- Nếu $a > b^k$ thì $T(n) = \Theta(n^{log_b a})$
- Nếu $a = b^k$ thì $T(n) = \Theta(n^k logn)$ với $logn = log_2 n$
- Nếu a $< b^k$ thì $T(n) = \Theta(n^k)$

Thuật toán quy hoạch động

- Sơ đồ chung
  - Chia bài toán xuất phát thành các bài toán con không nhất thiết độc lập với nhau
  - Giải các bài toán con từ nhỏ đến lớn, lời giải được lưu trữ lại vào 1 bảng
  - Bài toán con nhỏ nhất phải được giải 1 cách trực tiếp
  - Xây dựng lời giải của bài toán lớn hơn từ lời giải đã có của các bài toán con nhỏ hơn (truy hồi)
    - Số lượng bài toán con cần được bị chặn bởi đa thức của kích thước dữ liệu đầu vào
  - Phù hợp để giải hiệu quả một số bài toán tối ưu tổ hợp

Thuật toán quy hoạch động

- Bài toán dãy con cực đại
  - Đầu vào: dãy số nguyên $a_1, a_2, \ldots, a_n$
  - Đầu ra: tìm dãy con bao gồm các phần tử liên tiếp của dãy đã cho có tổng cực đại
- Phân chia
  - Ký hiệu P_i là bài toán tìm dãy con bao gồm các phần tử liên tiếp có tổng cực đại mà phần tử cuối cùng là a_i, với mọi $i = 1,\ldots, n$
  - Ký hiệu S_i là tổng các phần tử của lời giải của P_i, $\forall\ i = 1,\ldots, n$
  - $S_1 = a_1$
  - $S_i = \begin{cases} S_{i-1} + a_i, \text{ nếu } S_{i-1} > 0 \\ a_i, \text{ nếu } S_{i-1} \leq 0 \end{cases}$
  - Tổng các phần tử của dãy con cực đại của bài toán xuất phát là

$$\max\{S_1, S_2, \ldots, S_n\}$$

Thuật toán quy hoạch động

- Bài toán dãy con tăng dần cực đại
  - Đầu vào: dãy số nguyên $a = a_1, a_2, \ldots, a_n$ (gồm các phần tử đôi một khác nhau)
  - Đầu ra: tìm dãy con (bằng cách loại bỏ 1 số phần tử) của dãy đã cho tăng dần có số lượng phần tử là lớn nhất (gọi là dãy con cực đại)

Thuật toán quy hoạch động

- Phân chia
  - Ký hiệu P_i là bài toán tìm dãy con cực đại mà phần tử cuối cùng là a_i, với mọi $i = 1,\ldots, n$
  - Ký hiệu S_i là số phần tử của lời giải của P_i, $\forall\ i = 1,\ldots, n$
  - $S_1 = 1$
  - $S_i = \max\{1, \max\{S_j + 1 \mid j < i \wedge a_j < a_i\}\}$
  - Số phần tử của dãy con cực đại của bài toán xuất phát là

$$\max\{S_1, S_2, \ldots, S_n\}$$

```
void solve(){
  S[0] = 1;
  rs = S[0];
  for(int i = 1; i < n; i++){
    S[i] = 1;
    for(int j = i-1; j >= 0; j--){
      if(a[i] > a[j]){
        if(S[j] + 1 > S[i])
          S[i] = S[j] + 1;
      }
    }
    rs = S[i] > rs ? S[i] : rs;
  }
  printf("rs = %d\n",rs);
}
```

Thuật toán quy hoạch động

- Bài toán dãy con chung dài nhất
  - Ký hiệu $X = \langle X_1, X_2, \ldots, X_n \rangle$, một dãy con của X là dãy được tạo ra bằng việc loại bỏ 1 số phần tử nào đó của X đi
  - Đầu vào
    - Cho 2 dãy $X = \langle X_1, X_2, \ldots, X_n \rangle$ và $Y = \langle Y_1, Y_2, \ldots, Y_m \rangle$
  - Đầu ra
    - Tìm dãy con chung của X và Y có độ dài lớn nhất

Thuật toán quy hoạch động

- Bài toán dãy con chung dài nhất
  - Phân rã
    - Ký hiệu $S(i, j)$ là độ dài dãy con chung dài nhất của dãy $\langle X_1, \ldots, X_i \rangle$ và $\langle Y_1, \ldots, Y_j \rangle$, với $\forall\ i = 1, \ldots, n$ và $j = 1, \ldots, m$
    - Bài toán con nhỏ nhất
      - $\forall\ j = 1, \ldots, m$: $S(1, j) = \begin{cases} 1, \text{ nếu } X_1 \text{ xuất hiện trong } Y_1, \ldots, Y_j \\ 0, \text{ ngược lại} \end{cases}$
      - $\forall\ i = 1, \ldots, n$: $S(i, 1) = \begin{cases} 1, \text{ nếu } Y_1 \text{ xuất hiện trong } X_1, \ldots, X_i \\ 0, \text{ ngược lại} \end{cases}$
  - Tổng hợp lời giải

$$S(i, j) = \begin{cases} S(i\text{-}1, j\text{-}1) + 1, \text{ nếu } X_i = Y_j \\ \max\{S(i\text{-}1, j), S(i, j\text{-}1)\} \end{cases}$$

Thuật toán quy hoạch động

X	3	7	2	5	1	4	9

Y	4	3	2	3	6	1	5	4	9	7

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	1	1	1	1	1	1
2	0	1	1	1	1	1	1	1	1	2
3	0	1	2	2	2	2	2	2	2	2
4	0	1	2	2	2	2	3	3	3	3
5	0	1	2	2	2	3	3	3	3	3
6	1	1	2	2	2	3	3	4	4	4
7	1	1	2	2	2	3	3	4	5	5

Thuật toán quy hoạch động

```
void solve(){
  rs = 0;
  for(int i = 0; i <= n; i++) S[i][0] = 0;
  for(int j = 0; j <= m; j++) S[0][j] = 0;

  for(int i = 1; i <= n; i++){
    for(int j = 1; j <= m; j++){
      if(X[i] == Y[j]) S[i][j] = S[i-1][j-1] + 1;
      else{
        S[i][j] = S[i-1][j] > S[i][j-1] ?
                  S[i-1][j] : S[i][j-1];
      }
      rs = S[i][j] > rs ? S[i][j] : rs;
    }
  }
  printf("result = %d\n",rs);
}
```

Thuật toán quy hoạch động – Bài tập

- Cho dãy $a = a_1, a_2, \ldots, a_n$. Một dãy con của dãy a là một dãy thu được bằng cách loại bỏ 1 số phần tử khỏi a. Tìm dãy con của a là một cấp số cộng với bước nhảy bằng 1 có độ dài lớn nhất

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

DANH SÁCH TUYẾN TÍNH

Nội dung

- Định nghĩa danh sách tuyến tính
- Kiểu dữ liệu trừu tượng danh sách tuyến tính
- Mảng
- Danh sách liên kết
- Ngăn xếp
- Hàng đợi

Định nghĩa danh sách tuyến tính

- Các đối tượng được lưu trữ có thứ tự
- Các đối tượng có thể lặp lại giá trị
- Quan hệ trước-sau giữa các đối tượng

Kiểu dữ liệu trừu tượng danh sách tuyến tính

- Lưu trữ các đối tượng với quan hệ trước-sau
- Kí hiệu:
  - *L*: danh sách
  - *x*: đối tượng (phần tử của danh sách)
  - *p*: kiểu vị trí
  - END(*L*): hàm trả về vị trí sau vị trí của phần tử cuối cùng của danh sách *L*

Kiểu dữ liệu trừu tượng danh sách tuyến tính

- Thao tác
  - Insert(*x*, *p*, *L*): chèn phần tử *x* vào vị trí *p* trên danh sách *L*
  - Locate(*x*, *L*): trả về vị trí của phần tử x trên danh sách *L*
  - Retrieve(*p*, *L*): trả về phần tử ở vị trí p trong danh sách *L*
  - Delete(*p*, *L*): loại bỏ phần tử ở vị trí p trên danh sách *L*
  - Next(*p*, *L*): trả về vị trí tiếp theo của p trên danh sách *L*
  - Prev(*p*, *L*): trả về vị trí trước của p trên danh sách *L*
  - MakeNull(*L*): thiết lập danh sách *L* về danh sách rỗng
  - First(*L*): trả về vị trí đầu tiên của danh sách *L*

Kiểu mảng

- Các đối tượng được cấp phát liên tiếp nhau trong bộ nhớ
- Truy cập các đối tượng thông qua chỉ số (kiểu vị trí chính là giá trị nguyên xác định chỉ số)
- Việc thêm hoặc loại bỏ phần tử khỏi mảng cần thiết phải dồn mảng

Kiểu mảng

- Tổ chức dữ liệu
  - **`int a[100000];`** // bộ nhớ lưu trữ
  - **`int n;`** // số phần tử của mảng (bắt đầu từ chỉ số 0)

```
void insert(int x, int p) {
  for(int j = n-1; j >= p; j--)
    a[j+1] = a[j];
  a[p] = x; n = n + 1;
}
void delete(int p) {
  for(int j = p+1; j <= n-1; j++)
    a[j-1] = a[j];
  n = n – 1;
}
int retrieve(int p) {
  return a[p];
}
```

Kiểu mảng

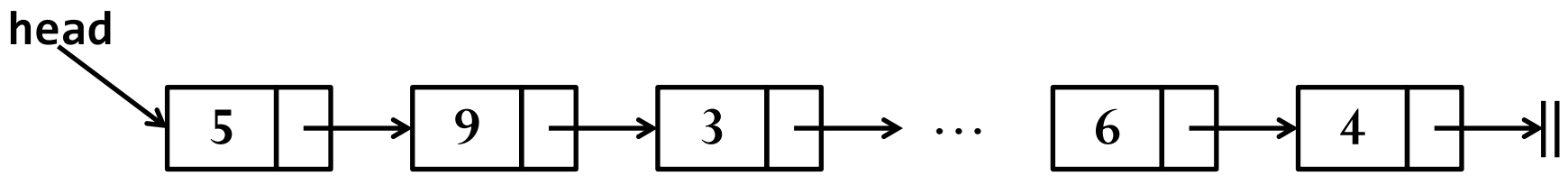
- Tổ chức dữ liệu
  - **`int a[100000];`** // bộ nhớ lưu trữ
  - **`int n;`** // số phần tử của mảng (bắt đầu từ chỉ số 0)

```
int locate(int x) {// vi trí đầu tiên
của x trong danh sách
  for(int j= 0; j <= n-1; j++)
    if(a[j] == x) return j;
  return -1;
}
void makeNull() {
  n = 0;
}
int next(int p) {
  if(0 <= p && p < n-1) return p+1;
  return -1;
}
int prev(int p) {
  if(p > 0 && p <= n-1) return p-1;
  return -1;
}
```

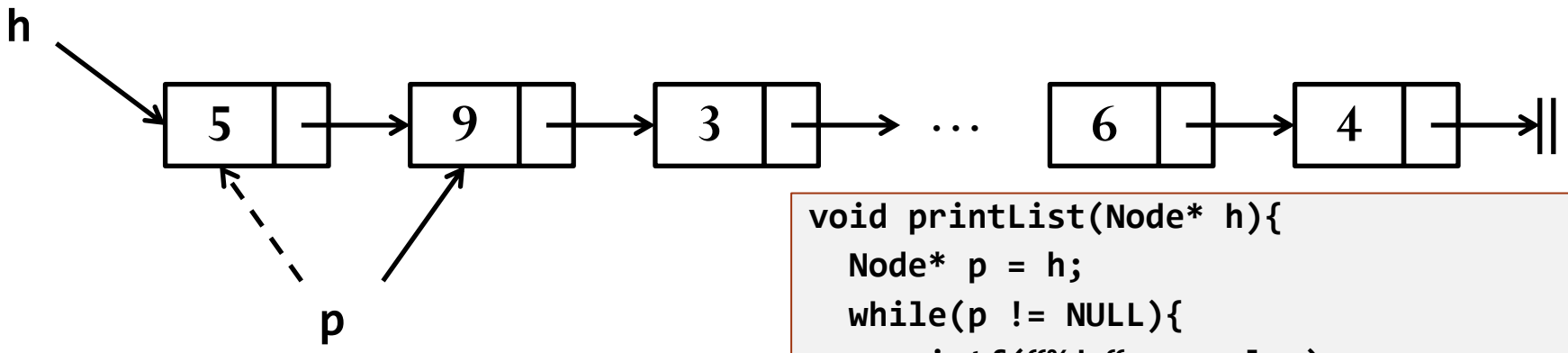
Danh sách liên kết nối đơn

- Mỗi phần tử, ngoài trường dữ liệu, có thêm con trỏ để trỏ đến phần tử tiếp theo

```
struct Node{
    int value;
    Node* next;
};
Node* head;
```

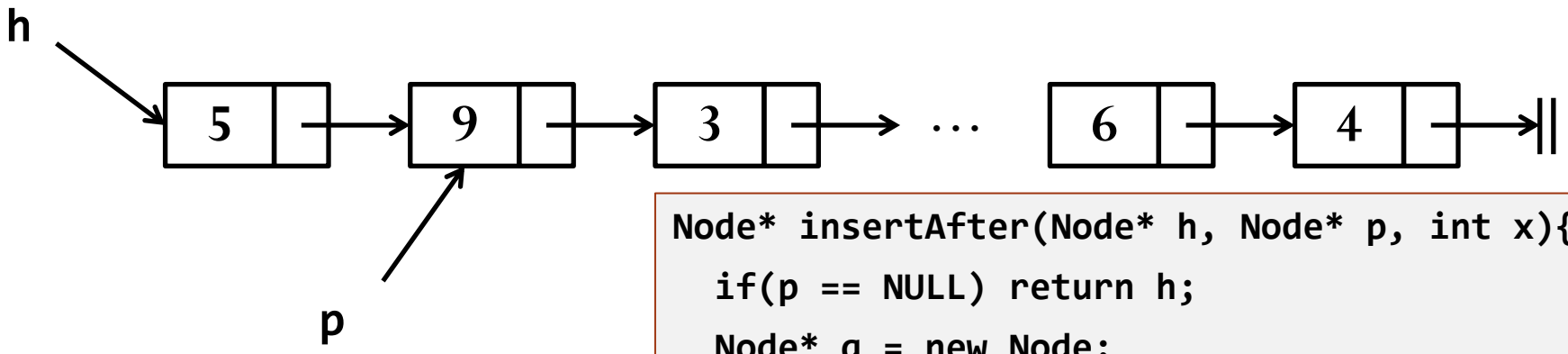


Danh sách liên kết nối đơn



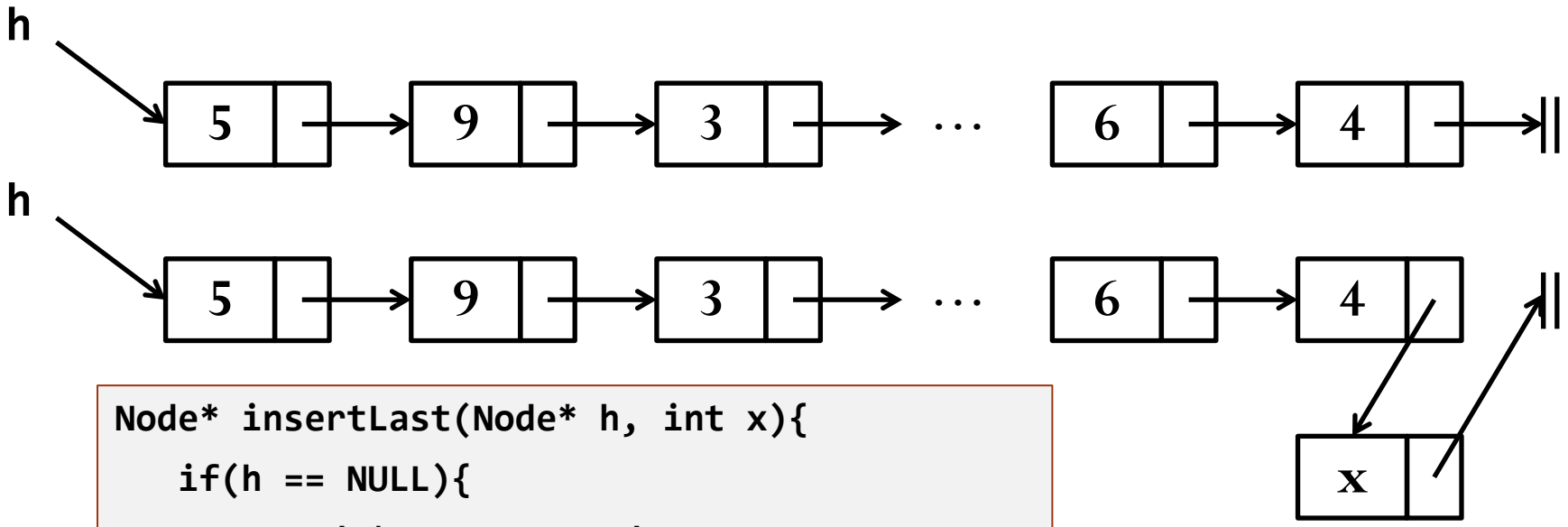

```
void printList(Node* h){
  Node* p = h;
  while(p != NULL){
    printf("%d ",p->value);
    p = p->next;
  }
}
Node* findLast(Node* h){
  Node* p = h;
  while(p != NULL){
    if(p->next == NULL) return p;
    p = p->next;
  }
  return NULL;
}
```

Danh sách liên kết nối đơn



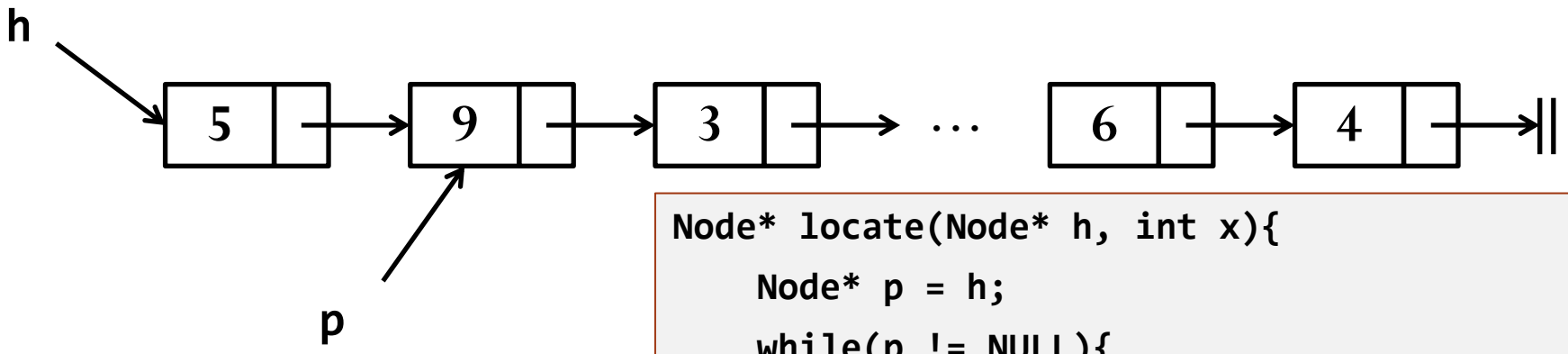

```
Node* insertAfter(Node* h, Node* p, int x){
  if(p == NULL) return h;
  Node* q = new Node;
  q->value = x; q->next = NULL;
  if(h == NULL) return q;
  q->next = p->next;
  p->next = q;
  return h;
}
```

Danh sách liên kết nối đơn



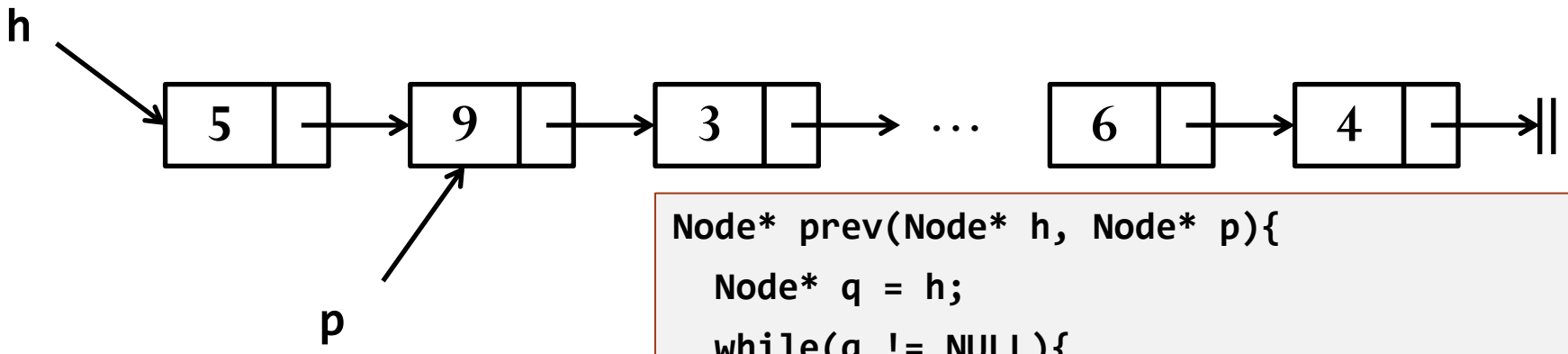

```
Node* insertLast(Node* h, int x){
   if(h == NULL){
        Node* q = new Node;
        q->value = x;
        q->next = NULL;
        return q;
    }
    Node* p = findLast(h);
    return insertAfter(h,p,x);
}
```

Danh sách liên kết nối đơn



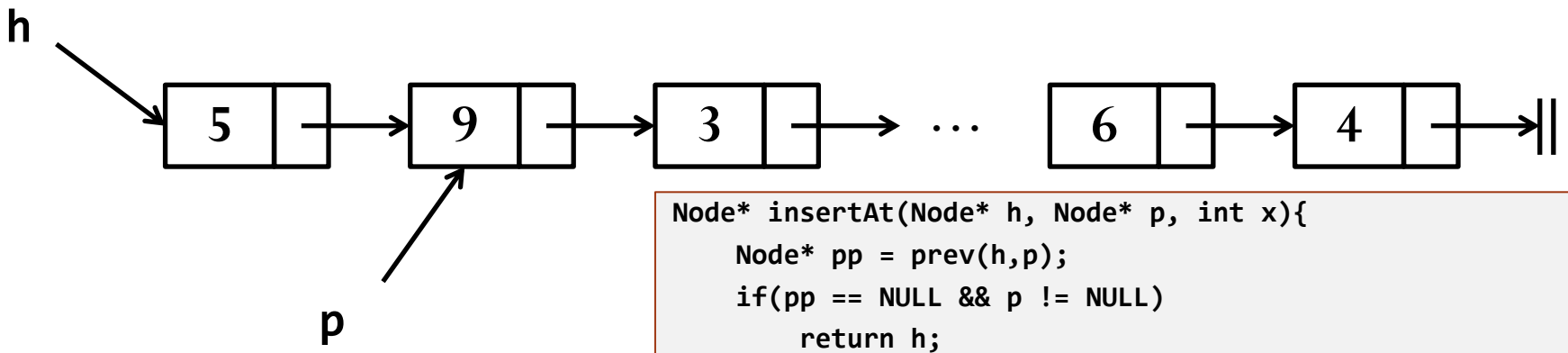

```
Node* locate(Node* h, int x){
    Node* p = h;
    while(p != NULL){
        if(p->value == x) return p;
        p = p->next;
    }
    return NULL;
}
```

Danh sách liên kết nối đơn



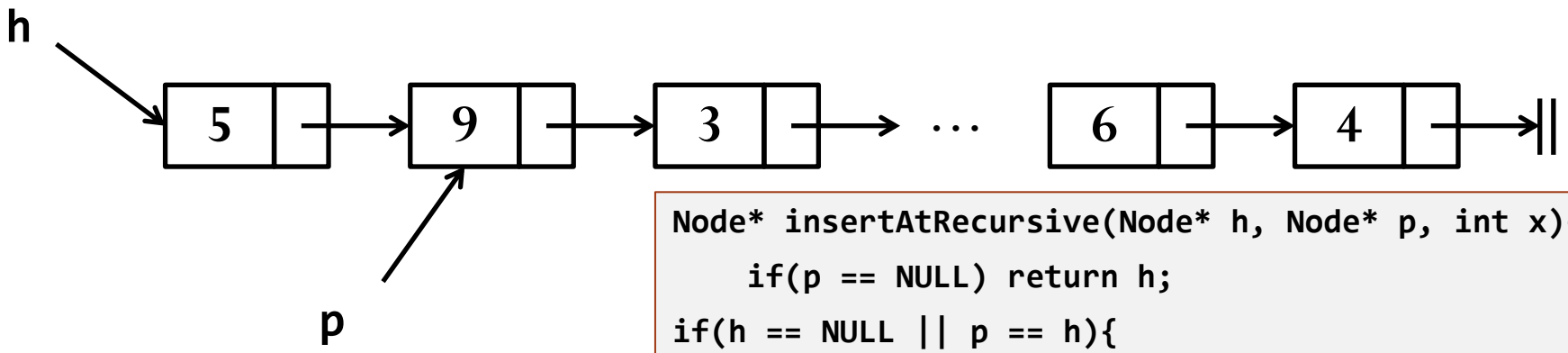

```
Node* prev(Node* h, Node* p){
  Node* q = h;
  while(q != NULL){
    if(q->next == p) return q;
    q = q->next;
  }
  return NULL;
}
```

Danh sách liên kết nối đơn



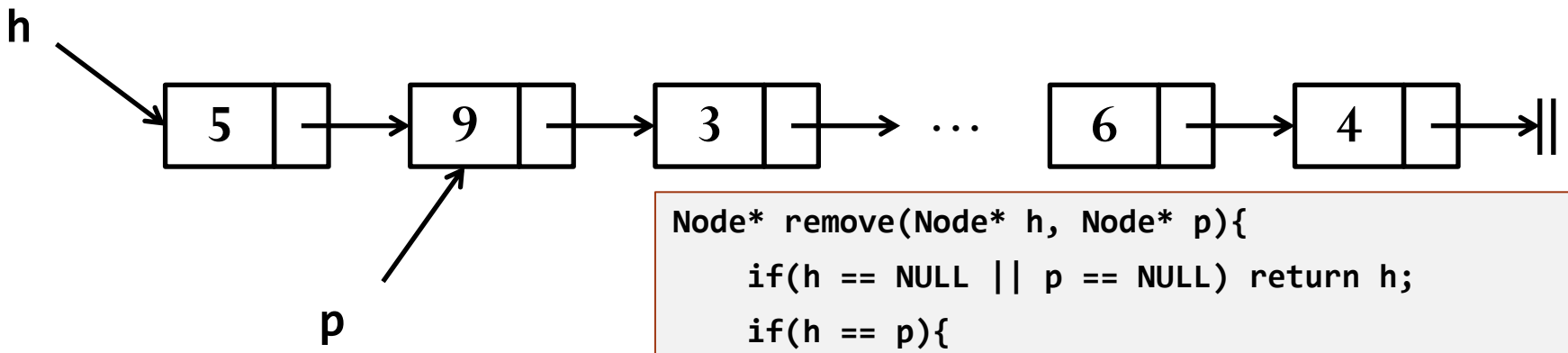

```
Node* insertAt(Node* h, Node* p, int x){
    Node* pp = prev(h,p);
    if(pp == NULL && p != NULL)
        return h;
    Node* q = new Node;
    q->value = x; q->next = NULL;
    if(pp == NULL){
        if(h == NULL){
            return q;
        }
        q->next = h;
        return q;
    }
    q->next = p;
    pp->next = q;
    return h;
}
```

Danh sách liên kết nối đơn



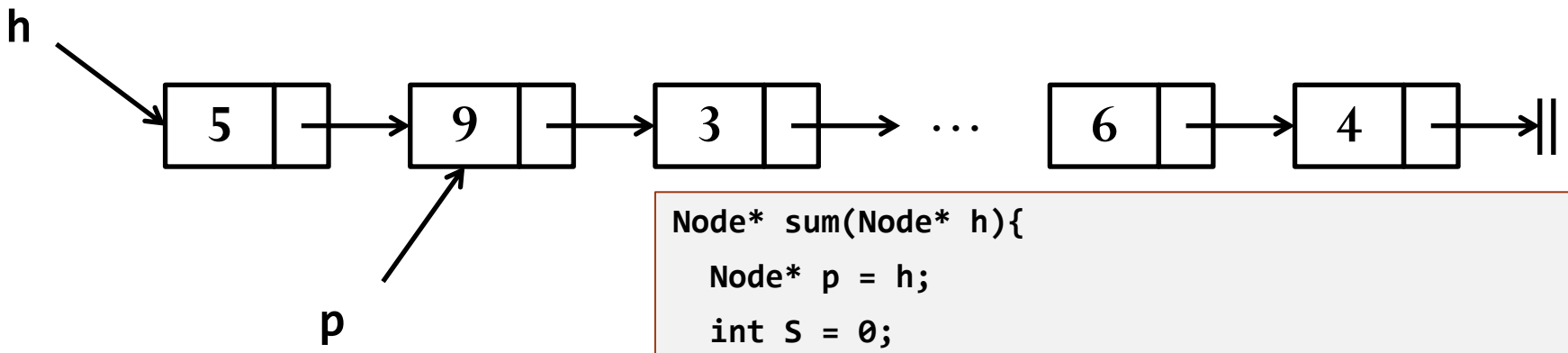

```
Node* insertAtRecursive(Node* h, Node* p, int x){
    if(p == NULL) return h;
if(h == NULL || p == h){
        Node* q = new Node;
        q->value = x;
        q->next = h;
        return q;
    }else{
        h->next = insertAtRecursive(h->next,p,x);
        return h;
    }
}
```

Danh sách liên kết nối đơn




```
Node* remove(Node* h, Node* p){
    if(h == NULL || p == NULL) return h;
    if(h == p){
        h = h->next;
        delete p;
        return h;
    }else{
        h->next = remove(h->next,p);
        return h;
    }
}
```

Danh sách liên kết nối đơn

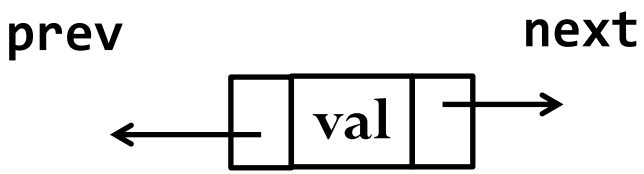


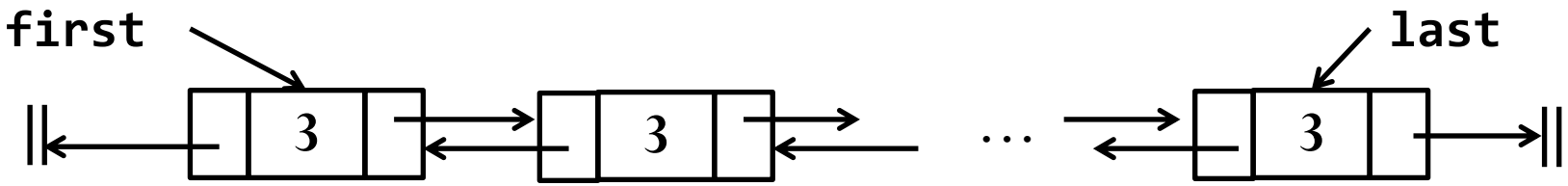

```
Node* sum(Node* h){
  Node* p = h;
  int S = 0;
  while(p != NULL) {
    S = S + p->value;
    p = p->next;
  }
  return S;
}
int sumRecursive(Node* h){
    if(h == NULL) return 0;
    return h->value + sumRecursive(h->next);
}
```

Danh sách liên kết nối đôi

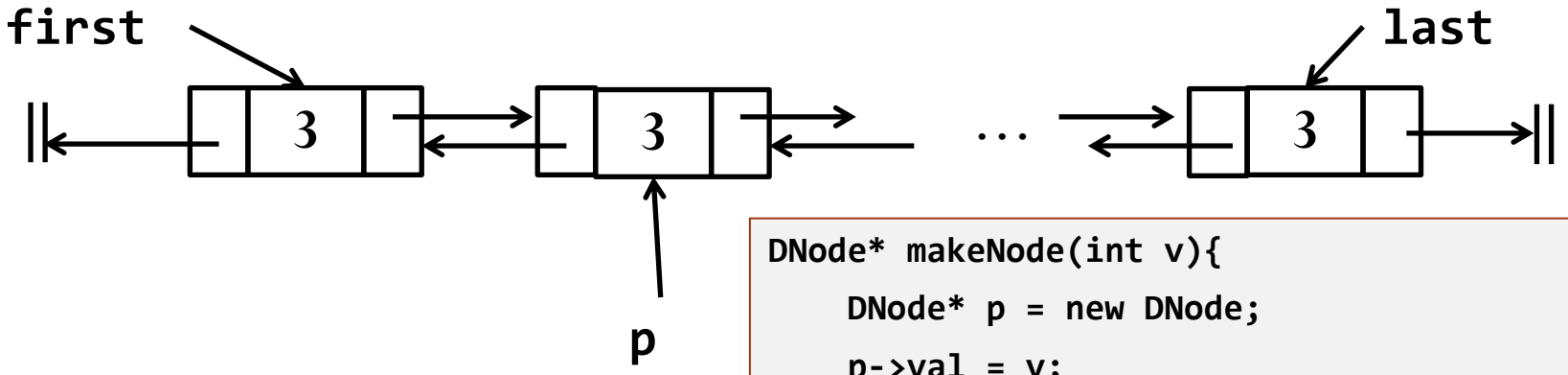
```
struct DNode{
    int val;
    DNode* prev;
    DNode* next;
};

DNode* first;
DNode* last;
```



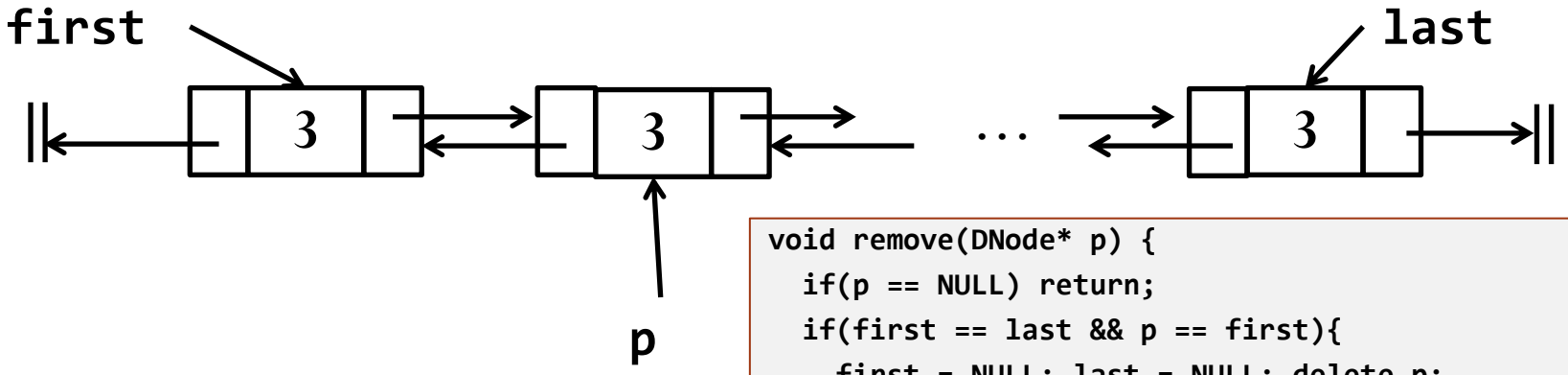


Danh sách liên kết nối đôi



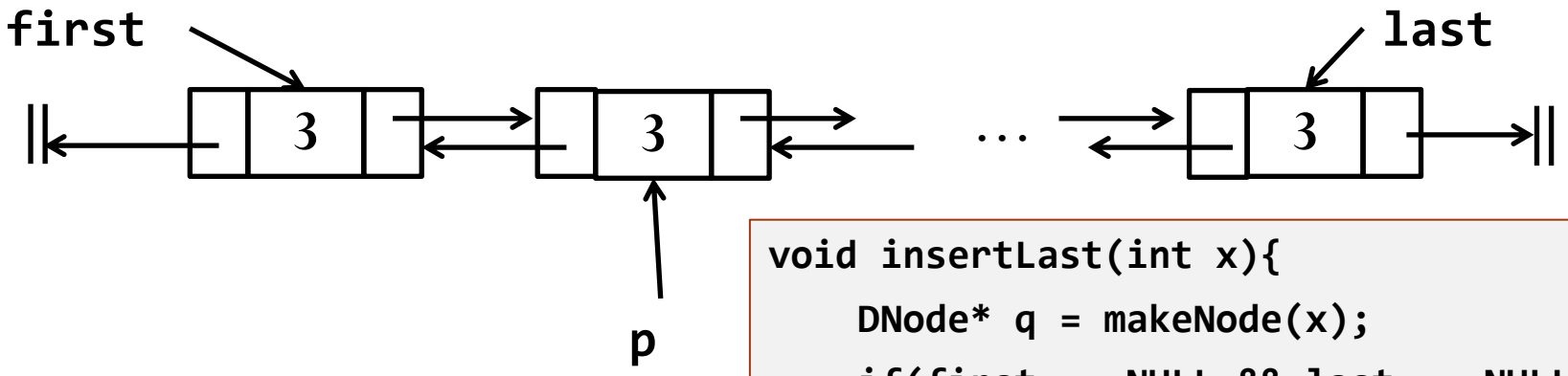

```
DNode* makeNode(int v){
    DNode* p = new DNode;
    p->val = v;
    p->next = NULL;
    p->prev = NULL;
    return p;
}
```

Danh sách liên kết nối đôi




```
void remove(DNode* p) {
  if(p == NULL) return;
  if(first == last && p == first){
    first = NULL; last = NULL; delete p;
  }
  if(p == first){
    first = p->next; first->prev = NULL;
    delete p; return;
  }
  if(p == last){
    last = p->prev; last->next = NULL;
    delete p; return;
  }
  p->prev->next = p->next;
  p->next->prev = p->prev;
  delete p;
}
```

Danh sách liên kết nối đôi




```
void insertLast(int x){
    DNode* q = makeNode(x);
    if(first == NULL && last == NULL){
        first = q;
        last = q;
        return;
    }
    q->prev = last;
    last->next = q;
    last = q;
}
```

Thư viện C++

- Kiểu list: http://www.cplusplus.com/reference/list/list/
- Phương thức
  - push_front()
  - push_back()
  - pop_front()
  - pop_back()
  - size()
  - clear()
  - erase()

```
#include <list>
#include <stdio.h>
using namespace std;

int main(){
  list<int> L;
  for(int i = 1; i <= 10; i++)
    L.push_back(i);
  list<int>::iterator it;
  for(it = L.begin(); it != L.end();
        it++){
     int x = *it;
     printf("%d ",x);
  }
}
```

Ngăn xếp (Stack)

- Danh sách các đối tượng
- Thao tác thêm mới và loại bỏ được thực hiện ở 1 đầu (đỉnh hay **top**) của danh sách (Last In First Out – LIFO)
- Thao tác
  - Push(x, S): chèn 1 phần tử x vào ngăn xếp
  - Pop(S): lấy ra 1 phần tử khỏi ngăn xếp
  - Top(S): truy cập phần tử ở đỉnh của ngăn xếp
  - Empty(S): trả về true nếu ngăn xếp rỗng

Ngăn xếp

- Ví dụ: kiểm tra tính hợp lệ của dãy ngoặc
  - [({})](): hợp lệ
  - ([} {}): không hợp lệ

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

using namespace std;

bool match(char a, char b){
    if(a == '(' && b == ')') return true;
    if(a == '{' && b == '}') return true;
    if(a == '[' && b == ']') return true;
    return false;
}
```

Ngăn xếp

- Ví dụ: kiểm tra tính hợp lệ của dãy ngoặc
  - [({})](): hợp lệ
  - ([} {}): không hợp lệ

```
bool solve(char* x, int n){
    stack<char> S;
    for(int i = 0; i <= n-1; i++){
        if(x[i] == '[' || x[i] == '(' || x[i] == '{'){
           S.push(x[i]);
        }else{
            if(S.empty()){
                return false;
            }else{
                char c = S.top();
                S.pop();
                if(!match(c,x[i])){
                    return false;
                }
            }
        }
    }
    return S.empty();
}
```

Hàng đợi (Queue)

- Danh sách các đối tượng với 2 đầu **head** và **tail**
- Thao tác thêm mới được thực hiện ở **tail**
- Thao tác loại bỏ được thực hiện ở **head** (First In First Out – FIFO)
- Thao tác
  - Enqueue(x, Q): chèn 1 phần tử x vào hàng đợi
  - Dequeue(Q): lấy ra 1 phần tử khỏi hàng đợi
  - Empty(Q): trả về true nêu hàng đợi rỗng

Hàng đợi

- Bài toán rót nước
  - Có 1 bể chứa nước (vô hạn)
  - Có 2 cốc với dung tích là a, b (nguyên dương) lít
  - Tìm cách đong đúng c (nguyên dương) lít nước

Hàng đợi

- Bài toán rót nước: a = 6, b = 8, c = 4

Bước	Thực hiện	Trạng thái
1	Đổ đầy nước vào cốc 1	(6,0)
2	Đổ hết nước từ cốc 1 sang cốc 2	(0,6)
3	Đổ đầy nước vào cốc 1	(6,6)
4	Đổ nước từ cốc 1 vào đầy cốc 2	(4,8)

Hàng đợi

- Bài toán rót nước: a = 4, b = 19, c = 21

Bước	Thực hiện	Trạng thái
1	Đổ đầy nước vào cốc 1	(4,0)
2	Đổ hết nước từ cốc 1 sang cốc 2	(0,4)
3	Đổ đầy nước vào cốc 1	(4,4)
4	Đổ hết nước từ cốc 1 sang cốc 2	(0,8)
5	Đổ đầy nước vào cốc 1	(4,8)
6	Đổ hết nước từ cốc 1 sang cốc 2	(0,12)
7	Đổ đầy nước vào cốc 1	(4,12)
8	Đổ hết nước từ cốc 1 sang cốc 2	(0,16)
9	Đổ đầy nước vào cốc 1	(4,16)
10	Đổ nước từ cốc 1 vào đầy cốc 2	(1,19)
11	Đổ hết nước ở cốc 2 ra ngoài	(1,0)

Hàng đợi

- Bài toán rót nước: a = 4, b = 19, c = 21 (tiếp)

Bước	Thực hiện	Trạng thái
12	Đổ hết nước từ cốc 1 sang cốc 2	(0,1)
13	Đổ đầy nước vào cốc 1	(4,1)
14	Đổ hết nước từ cốc 1 sang cốc 2	(0,5)
15	Đổ đầy nước vào cốc 1	(4,5)
16	Đổ hết nước từ cốc 1 sang cốc 2	(0,9)
17	Đổ đầy nước vào cốc 1	(4,9)
18	Đổ hết nước từ cốc 1 sang cốc 2	(0,13)
19	Đổ đầy nước vào cốc 1	(4,13)
20	Đổ hết nước từ cốc 1 sang cốc 2	(0,17)
21	Đổ đầy nước vào cốc 1	(4,17)

Hàng đợi

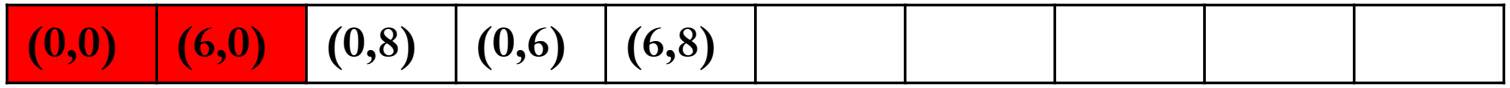
- Thiết kế thuật toán và cấu trúc dữ liệu
  - Trạng thái là bộ (x, y): lượng nước có trong cốc 1 và 2
  - Trạng thái ban đầu (0, 0)
  - Trạng thái kết thúc: $x = c$ hoặc $y = c$ hoặc $x + y = c$
  - Chuyển trạng thái
    - (1) Đổ đầy nước từ bể vào cốc 1: (a, y)
    - (2) Đổ đầy nước từ bể vào cốc 2: (x, b)
    - (3) Đổ hết nước từ cốc 1 ra ngoài: $(0, y)$
    - (4) Đổ hết nước từ cốc 2 ra ngoài: $(x, 0)$
    - (5) Đổ nước từ cốc 1 vào đầy cốc 2: $(x + y - b, b)$, nếu $x + y \geq b$
    - (6) Đổ hết nước từ cốc 1 sang cốc 2: $(0, x + y)$, nếu $x + y \leq b$
    - (7) Đổ nước từ cốc 2 vào đầy cốc 1: $(a, x + y - a)$, nếu $x + y \geq a$
    - (8) Đổ hết nước từ cốc 2 sang cốc 1: $(x + y, 0)$, nếu $x + y \leq a$

Hàng đợi

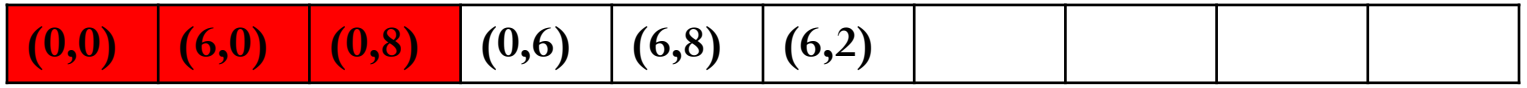
- Đưa (0,0) vào hàng đợi

- Lấy (0,0) ra và đưa (6,0), (0,8) vào hàng đợi

- Lấy (6,0) ra và đưa (0,6) và (6,8) vào hàng đợi

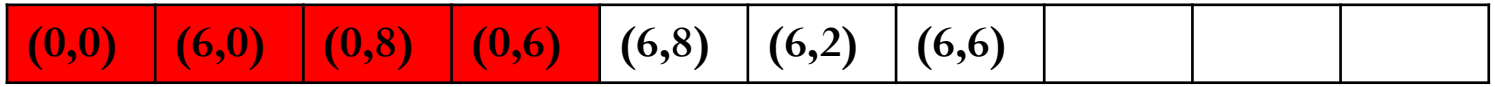


- Lấy (0,8) ra và đưa (6,2) vào hàng đợi

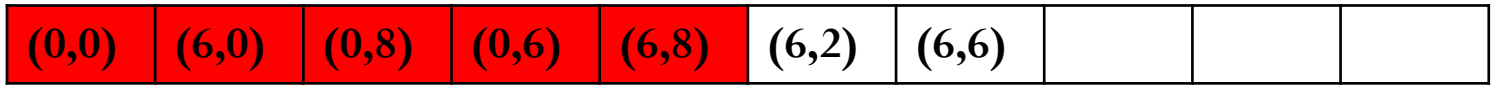


Hàng đợi

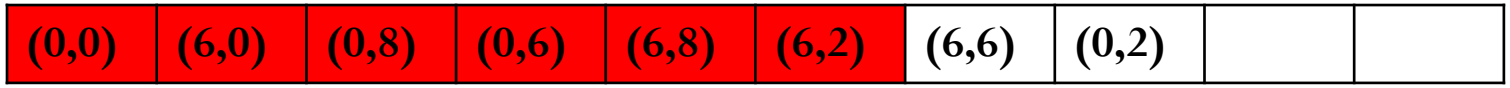
- Đưa (0,6) ra và đưa (6,6) vào hàng đợi



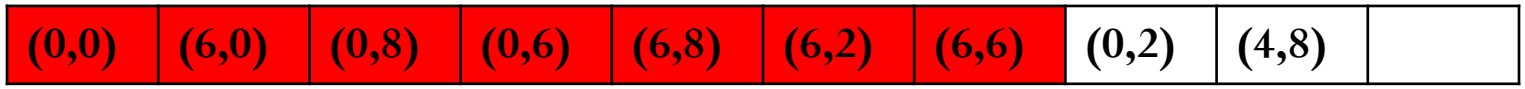
- Lấy (6,8) ra và không đưa trạng thái mới nào vào hàng đợi



- Lấy (6,2) ra và đưa (0,2) vào hàng đợi



- Lấy (6,6) ra và đưa (4,8) vào hàng đợi



Hàng đợi

- Thiết kế thuật toán và cấu trúc dữ liệu
  - Hàng đợi Q để ghi nhận các trạng thái được sinh ra
  - Mảng 2 chiều để đánh dấu trạng thái đã được xét đến
    - visited[x][y] = true, nếu trạng thái (x, y) đã được sinh ra
  - Ngăn xếp để in ra chuỗi các hành động để đạt được kết quả
  - Danh sách L để lưu các con trỏ trỏ đến các vùng nhớ được cấp phát động (phục vụ cho việc thu hồi bộ nhớ khi kết thúc chương trình)

Hàng đợi

- Khai báo cấu trúc dữ liệu

```
#include <stdio.h>
#include <stdlib.h>
#include <queue>
#include <stack>
#include <list>
using namespace std;

struct State{
  int x;
  int y;
  char* msg;// action to generate current state
  State* p;// pointer to the state generating current state
};

bool visited[10000][10000];
queue<State*> Q;
list<State*> L;
State* target;
int a,b,c;
```

Hàng đợi

- Các hàm khởi tạo mảng đánh dấu, đánh dấu trạng thái, kiểm tra trạng thái đích, giải phóng bộ nhớ

```
void initVisited(){
  for(int x = 0; x < 10000; x++)
    for(int y = 0; y < 10000; y++) visited[x][y] = false;
}
bool reachTarget(State* S){
    return S->x == c || S->y == c || S->x + S->y == c;
}
void markVisit(State* S){
    visited[S->x][S->y] = true;
}
void freeMemory(){
    list<State*>::iterator it;
    for(it = L.begin(); it != L.end(); it++){
        delete *it;
    }
}
```

Hàng đợi

- Hàm sinh trạng thái bởi hành động (3): đổ hết nước từ cốc 1 ra ngoài

```
bool genMove1Out(State* S){
    if(visited[0][S->y]) return false;
    State* newS = new State;
    newS->x = 0;
    newS->y = S->y;
    newS->msg = "Do het nuoc o coc 1 ra ngoai";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Hàng đợi

- Hàm sinh trạng thái bởi hành động (4): đổ hết nước từ cốc 2 ra ngoài

```
bool genMove2Out(State* S){
    if(visited[S->x][0]) return false;
    State* newS = new State;
    newS->x = S->x;
    newS->y = 0;
    newS->msg = "Do het nuoc o coc 2 ra ngoai";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Hàng đợi

- Hàm sinh trạng thái bởi hành động (5): đổ nước từ cốc 1 vào đầy cốc 2

```
bool genMove1Full2(State* S){
    if(S->x+S->y < b) return false;
    if(visited[S->x + S->y - b][b]) return false;
    State* newS = new State;
    newS->x = S->x + S->y - b;
    newS->y = b;
    newS->msg = "Do nuoc tu coc 1 vao day coc 2";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Hàng đợi

- Hàm sinh trạng thái bởi hành động (7): đổ nước từ cốc 2 vào đầy cốc 1

```
bool genMove2Full1(State* S){
    if(S->x+S->y < a) return false;
    if(visited[a][S->x + S->y - a]) return false;
    State* newS = new State;
    newS->x = a;
    newS->y = S->x + S->y - a;
    newS->msg = "Do nuoc tu coc 2 vao day coc 1";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Hàng đợi

- Hàm sinh trạng thái bởi hành động (6): đổ hết nước từ cốc 1 sang cốc 2

```
bool genMoveAll12(State* S){
    if(S->x + S->y > b) return false;
    if(visited[0][S->x + S->y]) return false;
    State* newS = new State;
    newS->x = 0;
    newS->y = S->x + S->y;
    newS->msg = "Do het nuoc tu coc 1 sang coc 2";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Hàng đợi

- Hàm sinh trạng thái bởi hành động (8): đổ hết nước từ cốc 2 sang cốc 1

```
bool genMoveAll21(State* S){
    if(S->x + S->y > a) return false;
    if(visited[S->x + S->y][0]) return false;
    State* newS = new State;
    newS->x = S->x + S->y;
    newS->y = 0;
    newS->msg = "Do het nuoc tu coc 2 sang coc 1";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Hàng đợi

- Hàm sinh trạng thái bởi hành động (1): đổ đầy nước từ bể vào cốc 1

```
bool genMoveFill1(State* S){
    if(visited[a][S->y]) return false;
    State* newS = new State;
    newS->x = a;
    newS->y = S->y;
    newS->msg = "Do day nuoc vao coc 1";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Hàng đợi

- Hàm sinh trạng thái bởi hành động (2): đổ đầy nước từ bể vào cốc 2

```
bool genMoveFill2(State* S){
    if(visited[S->x][b]) return false;
    State* newS = new State;
    newS->x = S->x;
    newS->y = b;
    newS->msg = "Do day nuoc vao coc 2";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;

}
```

Hàng đợi

- Hàm in chuỗi hành động để thu được kết quả

```
void print(State* target){
    printf("-----------RESULT------------\n");
    if(target == NULL) printf("Khong co loi giai!!!!!!!");
    State* currentS = target;
    stack<State*> actions;
    while(currentS != NULL){
        actions.push(currentS);
        currentS = currentS->p;
    }
    while(actions.size() > 0){
        currentS = actions.top();
        actions.pop();
        printf("%s, (%d,%d)\n",currentS->msg,currentS->x,
                currentS->y);
    }
}
```

Hàng đợi

- Khởi tạo, sinh trạng thái ban đầu và đưa vào hàng đợi
- Tại mỗi bước, lấy 1 trạng thái ra khỏi hàng đợi, sinh trạng thái mới và đưa vào hàng đợi

```
void solve(){
  initVisited();
  // sinh ra trang thai ban dau (0,0) va dua vao Q
  State* S = new State;
  S->x = 0; S->y = 0; S->p = NULL;
  Q.push(S); markVisit(S);
  while(!Q.empty()){
    State* S = Q.front(); Q.pop();
    if(genMove1Out(S)) break; // (0,y)
    if(genMove2Out(S)) break; // (x,0)
    if(genMove1Full2(S)) break; // (x+y-b,b) if(x+y >= b)
    if(genMoveAll12(S)) break; // (0,x+y) if(x+y <= b)
    if(genMove2Full1(S)) break; // (a,x+y-a) if (x+y >= a)
    if(genMoveAll21(S)) break;// (x+y,0) if(x+y <= a)
    if(genMoveFill1(S)) break;// (a,y)
    if(genMoveFill2(S)) break;// (x,b)
  }
}
```

Hàng đợi

- Hàm main
  - Thử nghiệm với bộ dữ liệu: a = 4, b = 7, c = 9

```
int main(){
    a = 4;
    b = 7;
    c = 9;
    target = NULL;
    solve();
    print(target);
    freeMemory();
}
```

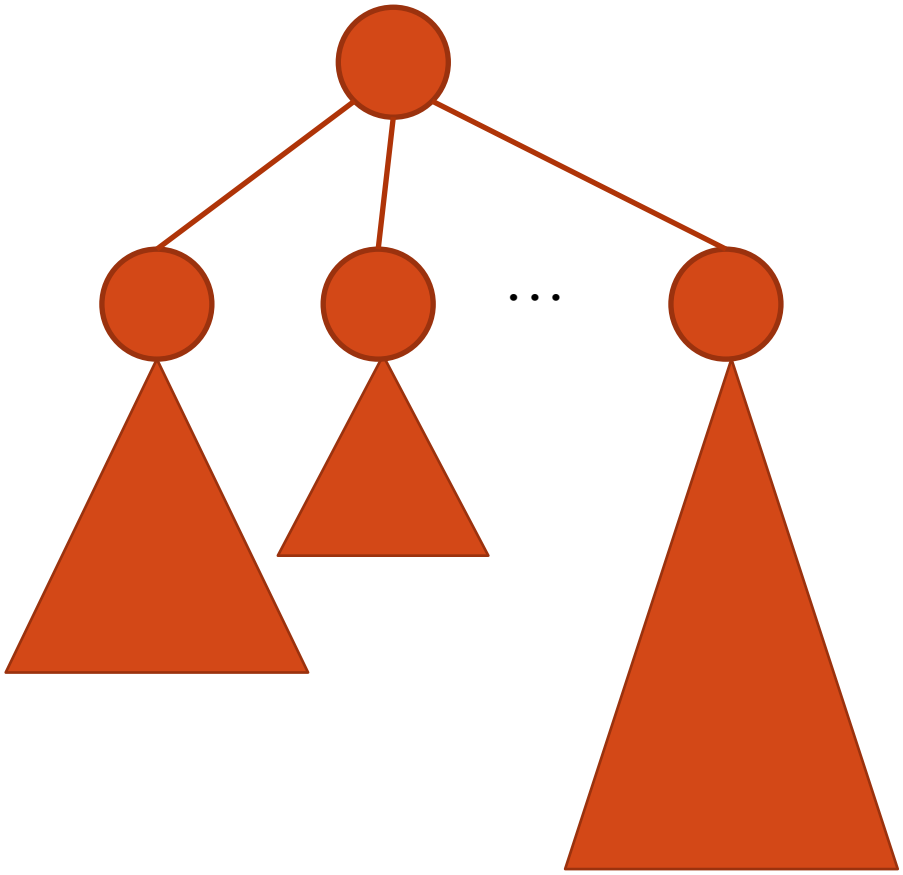
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

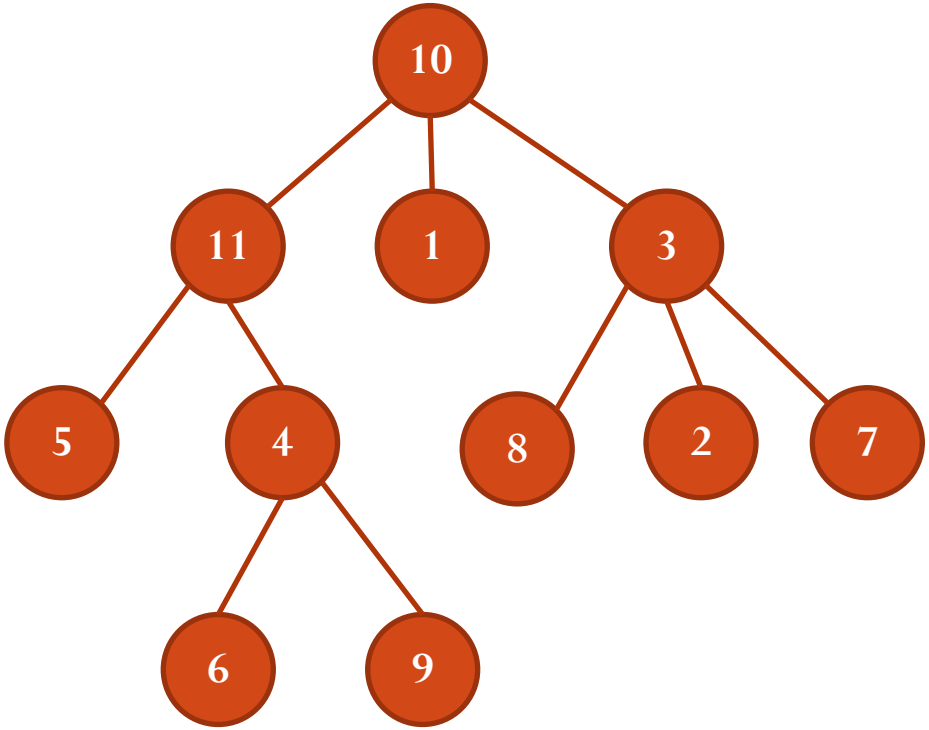
CÂY

Nội dung

- Định nghĩa cây
- Các khái niệm trên cây
- Các phép duyệt cây
- Cấu trúc lưu trữ
- Các thao tác trên cây

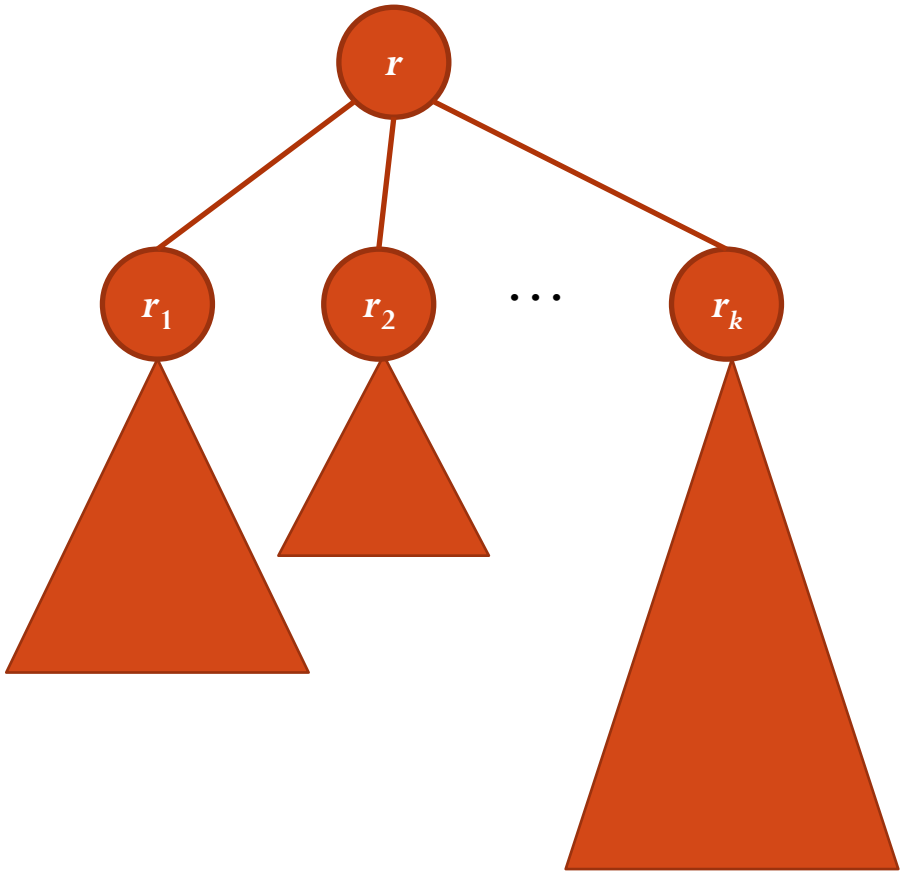
Định nghĩa cây





Định nghĩa cây

- Cấu trúc lưu trữ các đối tượng có quan hệ phân cấp
- Định nghĩa đệ quy
  - Bước cơ sở: r là 1 nút, cây T có nút gốc là r
  - Bước đệ quy:
    - Giả sử $T_1, T_2, \ldots, T_k$ là các cây có gốc là $r_1, r_2, \ldots, r_k$
    - Nút r
    - Liên kết các nút $r_1, r_2, .., r_k$ như la nút con của r sẽ tạo ra cây T



Các khái niệm trên cây

- Đường đi: dãy các nút $x_1, x_2, \ldots, x_q$ trong đó x_i là cha của x_{i+1}, $i = 1, 2, \ldots, q\text{-}1$. Độ dài đường đi là q-1
- Nút lá: không có nút con
- Nút trong: có nút con
- Nút anh em: 2 nút *u* và *v* là 2 nút anh em nếu có cùng nút cha
- Tổ tiên: nút *u* là tổ tiên của *v* nếu có đường đi từ *u* đến *v*
- Con cháu: nút *u* là con cháu của *v* nếu *v* là tổ tiên của *u*
- Độ cao: độ cao của 1 nút là độ dài đường đi dài nhất từ nút đó đến nút lá + 1
- Độ sâu: độ sâu của 1 nút *v* là độ dài đường đi duy nhất từ nút gốc tới *v*

Các phép duyệt cây

- Thăm các nút của 1 cây theo 1 thứ tự nào đó
- 3 phép duyệt cây cơ bản
  - Duyệt theo thứ tự trước
  - Duyệt theo thứ tự giữa
  - Duyệt theo thứ tự sau
- Xét cây T có cấu trúc
  - Nút gốc r
  - Cây con T_1 (gốc r_1), T_2 (gốc r_2), …, T_k (gốc r_k) theo thứ tự từ trái qua phải

Các phép duyệt cây

- Duyệt theo thứ tự trước
  - Thăm nút gốc
  - Duyệt cây T_1 theo thứ tự trước
  - Duyệt cây T_2 theo thứ tự trước
  - …
  - Duyệt cây T_k theo thứ tự trước

```
preOrder(r){
    if(r = NULL) return;
    visit(r);
    for each p = r1, r2, .., rk {
        preOrder(p);
    }
}
```

Các phép duyệt cây

- Duyệt theo thứ tự giữa
  - Duyệt cây T_1 theo thứ tự giữa
  - Thăm nút gốc r
  - Duyệt cây T_2 theo thứ tự giữa
  - …
  - Duyệt cây T_k theo thứ tự giữa

```
inOrder(r){
    if(r = NULL) return;
    inOrder(r1);
    visit(r);
    for each p = r2, .., rk {
        inOrder(p);
    }
}
```

Các phép duyệt cây

- Duyệt theo thứ tự sau
  - Duyệt cây T_1 theo thứ tự sau
  - Duyệt cây T_2 theo thứ tự sau
  - …
  - Duyệt cây T_k theo thứ tự sau
  - Thăm nút gốc

```
postOrder(r){
   if(r = NULL) return;
   for each p = r1, r2, .., rk {
      postOrder(p);
   }
   visit(r);
}
```

Cấu trúc lưu trữ cây

- Mảng:
  - Giả sử các nút trên cây được định danh 1, 2, …, *n*
  - *a*[1..*n*] trong đó *a*[*i*] là nút cha của nút *i*
  - Cấu trúc lưu trữ đơn giản, tuy nhiên khó cài đặt rất nhiều thao tác trên cây
- Con trỏ liên kết: với mỗi nút, lưu 2 con trỏ
  - Con trỏ trỏ đến nút con trái nhất
  - Con trỏ trỏ đến nút anh em bên phải

Cấu trúc lưu trữ cây

```
struct Node{
    int id;
    Node* leftMostChild;// con tro tro den nut con trai nhat
    Node* rightSibling;// con tro tro den nut anh em ben phai
};
Node* root;// con tro tro den nut goc
```

Các thao tác trên cây

- find(r, id): tìm nút có định danh là id trên cây có gốc là r
- insert(r, p, id): tạo một nút có định danh là id, chèn vào cuối danh sách nút con của nút p trên cây có gốc là r
- height(r, p): trả về độ cao của nút p trên cây có gốc là r
- depth(r, p): trả về độ sâu của nút p trên cây có gốc là r
- parent(r, p): trả về nút cha của nút p trên cây có gốc r
- count(r): trả về số nút trên cây có gốc là r
- countLeaves(r): trả về số nút lá trên cây có gốc là r

Phép toán trên cây

- Tìm một nút có nhãn cho trước trên cây

```
Node* find(Node* r, int v){
    if(r == NULL) return NULL;
    if(r->id == v) return r;
    Node* p = r->leftMostChild;
    while(p != NULL){
        Node* h = find(p,v);
        if(h != NULL) return h;
        p = p->rightSibling;
    }
    return NULL;
}
```

Phép toán trên cây

- Duyệt theo thứ tự trước

```
void preOrder(Node* r){
    if(r == NULL) return;
    printf("%d ",r->id);
    Node* p = r->leftMostChild;
    while(p != NULL){
        preOrder(p);
        p = p->rightSibling;
    }
}
```

Phép toán trên cây

- Duyệt theo thứ tự giữa

```
void inOrder(Node* r){
   if(r == NULL) return;
   Node* p = r->leftMostChild;
   inOrder(p);
   printf("%d ",r->id);
   if(p != NULL)
      p = p->rightSibling;
   while(p != NULL){
        inOrder(p);
        p = p->rightSibling;
   }
}
```

Phép toán trên cây

- Duyệt theo thứ tự sau

```
void postOrder(Node* r){
    if(r == NULL) return;
    Node* p = r->leftMostChild;
    while(p != NULL){
        postOrder(p);
        p = p->rightSibling;
    }
    printf("%d ",r->id);
}
```

Phép toán trên cây

- Đếm số nút trên cây

```
int count(Node* r){
    if(r == NULL) return 0;
    int s = 1;
    Node* p = r->leftMostChild;
    while(p != NULL){
        s += count(p);
        p = p->rightSibling;
    }
    return s;
}
```

Phép toán trên cây

- Đếm số nút lá trên cây

```
int countLeaves(Node* r){
    if(r == NULL) return 0;
    int s = 0;
    Node* p = r->leftMostChild;
    if(p == NULL) s = 1;
    while(p != NULL){
        s += countLeaves(p);
        p = p->rightSibling;
    }
    return s;
}
```

Phép toán trên cây

- Độ cao của một nút

```
int height(Node* p){
    if(p == NULL) return 0;
    int maxh = 0;
    Node* q = p->leftMostChild;
    while(q != NULL){
        int h = height(q);
        if(h > maxh) maxh = h;
        q = q->rightSibling;
    }
    return maxh + 1;
}
```

Phép toán trên cây

- Tìm nút cha của một nút

```
Node* parent(Node* p, Node* r){
    if(r == NULL) return NULL;
    Node* q = r->leftMostChild;
    while(q != NULL){
        if(p == q) return r;
        Node* pp = parent(p, q);
        if(pp != NULL) return pp;
        q = q->rightSibling;
    }
    return NULL;
}
```

Cây nhị phân

- Mỗi nút có nhiều nhất 2 nút con
- Hai con trỏ xác định nút con trái và nút con bên phải

```
struct BNode{
   int id;
   BNode* leftChild; // con trỏ đến nút con trái
   BNode* rightChild;// con trỏ đến nút con phải
};
```

- `leftChild = NULL`: có nghĩa nút hiện tại không có con trái
- `rightChild = NULL`: có nghĩa nút hiện tại không có con phải
- Có thể áp dụng sơ đồ thuật toán trên cây tổng quát cho trường hợp cây nhị phân

Thao tác trên cây nhị phân

- Duyệt theo thứ tự giữa
  - Duyệt theo thứ tự giữa cây con bên trái
  - Thăm nút gốc
  - Duyệt theo thứ tự giữa cây con bên phải

```
void inOrder(BNode* r) {
  if(r == NULL) return;
  inOrder(r->leftChild);
  printf("%d ",r->id);
  inOrder(r->rightChild);
}
```

Thao tác trên cây nhị phân

- Duyệt theo thứ tự trước
  - Thăm nút gốc
  - Duyệt theo thứ tự giữa cây con bên trái
  - Duyệt theo thứ tự giữa cây con bên phải

```
void preOrder(BNode* r) {
  if(r == NULL) return;
  printf("%d ",r->id);
  preOrder(r->leftChild);
  preOrder(r->rightChild);
}
```

Thao tác trên cây nhị phân

- Duyệt theo thứ tự sau
  - Duyệt theo thứ tự giữa cây con bên trái
  - Duyệt theo thứ tự giữa cây con bên phải
  - Thăm nút gốc

```
void postOrder(BNode* r) {
  if(r == NULL) return;
  postOrder(r->leftChild);
  postOrder(r->rightChild);
  printf("%d ",r->id);

}
```

Thao tác trên cây nhị phân

- Đếm số nút trên cây nhị phân

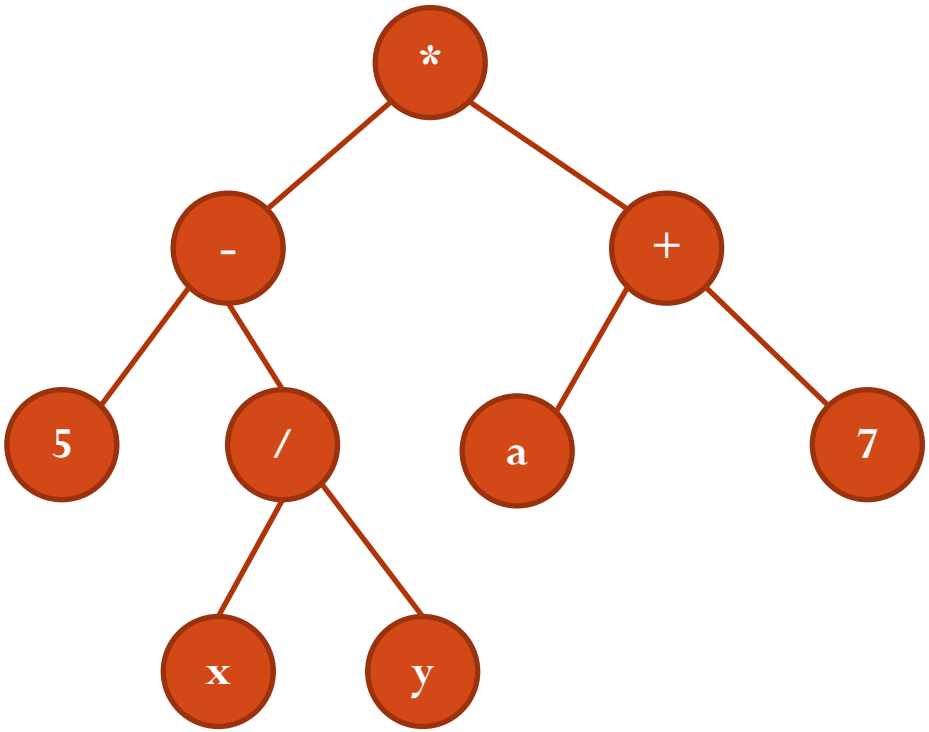
```
int count(BNode* r) {
  if(r == NULL) return 0;
  return 1 + count(r->leftChild) +
             count(r->rightChild);
}
```

Cây biểu thức

- Là cây nhị phân
  - Nút giữa biểu diễn toán tử
  - Nút lá biểu diện các toán hạng (biến, hằng)
- Biểu thức trung tố: dãy các nút được thăm trong phép duyệt cây theo thứ tự giữa:

  (5 - x/y) * (a + 7)
- Biểu thức hậu tố: dãy các nút được thăm trong phép duyệt cây theo thứ tự sau:

  5 x y / - a 7 + *



Tính giá trị của biểu thức hậu tố

- Khởi tạo stack S ban đầu rỗng
- Duyệt các phần tử của biểu thức hậu tố từ trái qua phải
- Nếu gặp toán hạng thì đẩy toán hạng đó vào S
- Nếu gặp toán tử ***op*** thì lần lượt lấy 2 toán hàng A và B ra khỏi S, thực hiện C = B ***op*** A, và đẩy C vào S
- Khi biểu thức hậu tố được duyệt xong thì giá trị còn lại trong S chính là giá trị của biểu thức đã cho

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

SẮP XẾP

Nội dung

- Giới thiệu bài toán sắp xếp
- Thuật toán sắp xếp chèn
- Thuật toán sắp xếp lựa chọn
- Thuật toan sắp xếp nổi bọt
- Thuật toán sắp xếp trộn
- Thuật toán sắp xếp nhanh
- Thuật toán sắp xếp vun đống

Giới thiệu bài toán sắp xếp

- Sắp xếp là việc đưa các phần tử của một dãy theo đúng thứ tự (không giảm hoặc không tăng) dựa vào 1 giá trị khoá
- Thiết kế thuật toán sắp xếp hiệu quả là một việc đặc biệt quan trọng do việc sắp xếp xuất hiện trong rất nhiều tình huống tính toán
- Hai thao tác cơ bản trong một thuật toán sắp xếp
  - Compare(a, b): so sánh khoá của 2 phần tử a và b
  - Swap(a, b): đổi chỗ 2 phần tử a và b cho nhau
- Không giảm tổng quát, giả sử cần sắp xếp dãy $a_1, a_2, \ldots, a_n$ theo thứ tự không giảm của giá trị

Giới thiệu bài toán sắp xếp

- Phân loại thuật toán sắp xếp
  - Sắp xếp *tại chỗ*: sử dụng bộ nhớ trung gian là hằng số, không phụ thuộc độ dài dãy đầu vào
  - Sắp xếp *ổn định*: duy trì thứ tự tương đối giữa 2 phần tử có cùng giá trị khoá (vị trí tương đối giữa 2 phần tử có cùng khoá không đổi trước và sau khi sắp xếp)
  - Thuật toán sắp xếp dựa trên so sánh: sử dụng phép so sánh để quyết định thứ tự phần tử (counting sort không phải là thuật toán sắp xếp dựa trên so sánh)

Thuật toán sắp xếp chèn (insertion sort)

- Thuật toán diễn ra qua các bước lặp $k = 2, 3, \ldots, n$
- Tại mỗi bước thứ k: chèn a_k vào đúng vị trí trong dãy đã được sắp $a_1, a_2, \ldots, a_{k-1}$ để thu được dãy được sắp đúng thứ tự
- Sau bước thứ k thì dãy $a_1, a_3, \ldots, a_k$ đã được sắp đúng thứ tự, dãy còn lại $a_{k+1}, \ldots, a_n$ giữ nguyên vị trí

```
void insertionSort(int A[], int N)
{
  // index tu 1 -> N
  for(int k = 2; k <= N; k++){
   int last = A[k];
   int j = k;
   while(j > 1 && A[j-1] >
        last){
     A[j] = A[j-1];
     j--;
   }
   A[j] = last;
  }
}
```

Thuật toán sắp xếp chèn (insertion sort)

- Ví dụ: 5, 7, 3, 8, 1, 2, 9, 4, 6

5	7	3	8	1	2	9	4	6

3	5	7	8	1	2	9	4	6

3	5	7	8	1	2	9	4	6

1	3	5	7	8	2	9	4	6

1	2	3	5	7	8	9	4	6

1	2	3	5	7	8	9	4	6

1	2	3	4	5	7	8	9	6

1	2	3	4	5	6	7	8	9

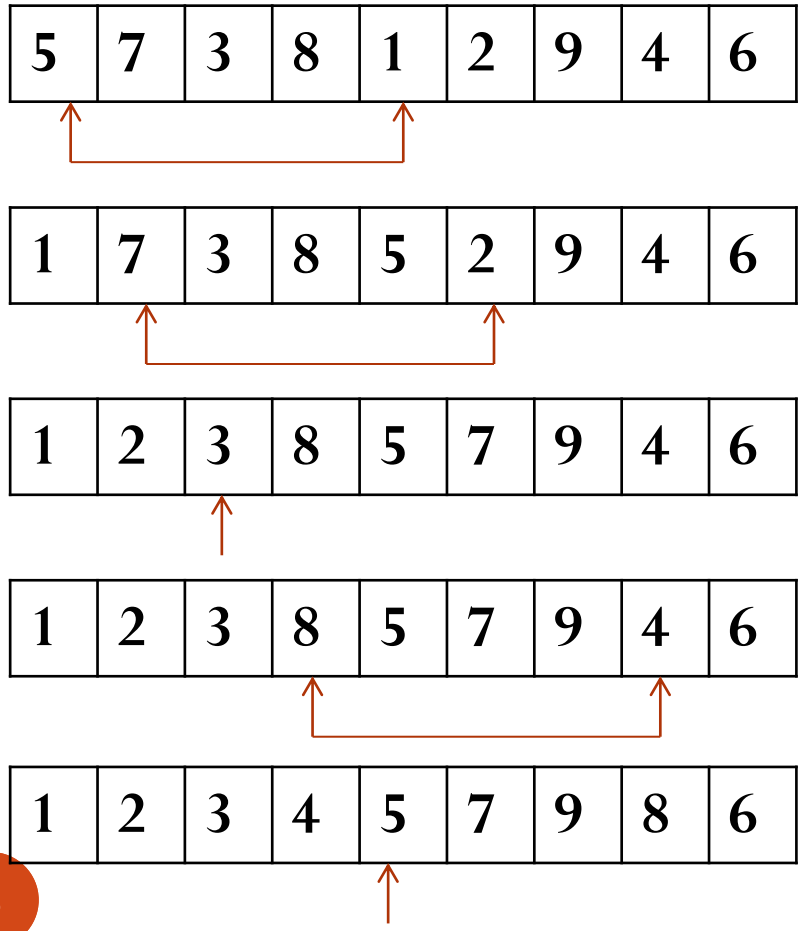
Thuật toán sắp xếp lựa chọn (selection sort)

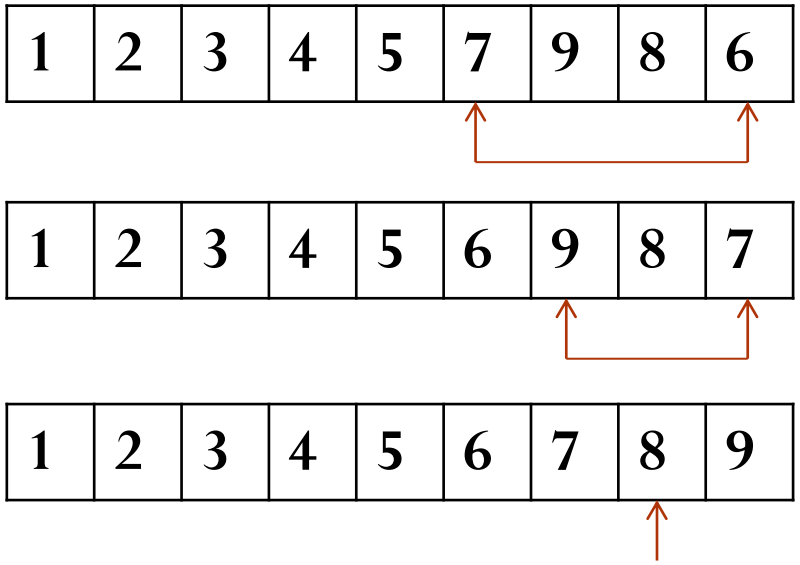
- Chọn số nhỏ nhất xếp vào vị trí thứ 1
- Chọn số nhỏ thứ 2 xếp vào vị trí thứ 2
- Chọn số nhỏ thứ 3 xếp vào vị trí thứ 3
- …

```
void selectionSort(int A[], int N) {
  // index tu 1 -> N
  for(int k = 1; k <= N; k++){
    int min = k;
    for(int j = k+1; j <= N; j++){
      if(A[min] > A[j]) min = j;
    }
    int tmp = A[min];
    A[min] = A[k];
    A[k] = tmp;
  }
}
```

Thuật toán sắp xếp lựa chọn (selection sort)

- Ví dụ: 5, 7, 3, 8, 1, 2, 9, 4, 6





Thuật toán sắp xếp nổi bọt (bubble sort)

- Duyệt dãy từ trái qua phải (hoặc từ phải qua trái)
  - Tại mỗi bước, so sánh 2 phần tử đứng cạnh nhau và tiến hành đổi chỗ 2 phần tử đó nếu phần tử trước lớn hơn phần tử sau
- Lặp lại quá trình trên khi nào trong dãy vẫn còn 2 phần tử đứng cạnh nhau mà phần tử trước lớn hơn phần tử sau

```
void bubleSort(int A[], int N) {
  // index tu 1 den N
  int swapped;
  do{
    swapped = 0;
    for(int i = 1; i < N; i++){
      if(A[i] > A[i+1]){
        int tmp = A[i];
        A[i] = A[i+1];
        A[i+1] = tmp;
        swapped = 1;
      }
    }
  }while(swapped == 1);
}
```

Thuật toán sắp xếp nổi bọt (bubble sort)

- Ví dụ: 5, 7, 3, 8, 1, 2, 9, 4, 6

5	**3**	**7**	**1**	**2**	**8**	**4**	**6**	**9**

3	**5**	**1**	**2**	**7**	**4**	**6**	**8**	**9**

3	**1**	**2**	**5**	**4**	**6**	**7**	**8**	**9**

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Thuật toán sắp xếp trộn (merge sort)

- Dựa trên chia để trị
- Chia dãy $a_1, \ldots, a_n$ thành 2 dãy con có độ dài bằng nhau
- Sắp xếp 2 dãy con bằng thuật toán sắp xếp trộn
- Trộn 2 dãy con đã được sắp với nhau để thu được dãy ban đầu được sắp thứ tự

```
void mergeSort(int A[], int L, int R) {
  if(L < R){
    int M = (L+R)/2;
    mergeSort(A,L,M);
    mergeSort(A,M+1,R);
    merge(A,L,M,R);
  }
}
```

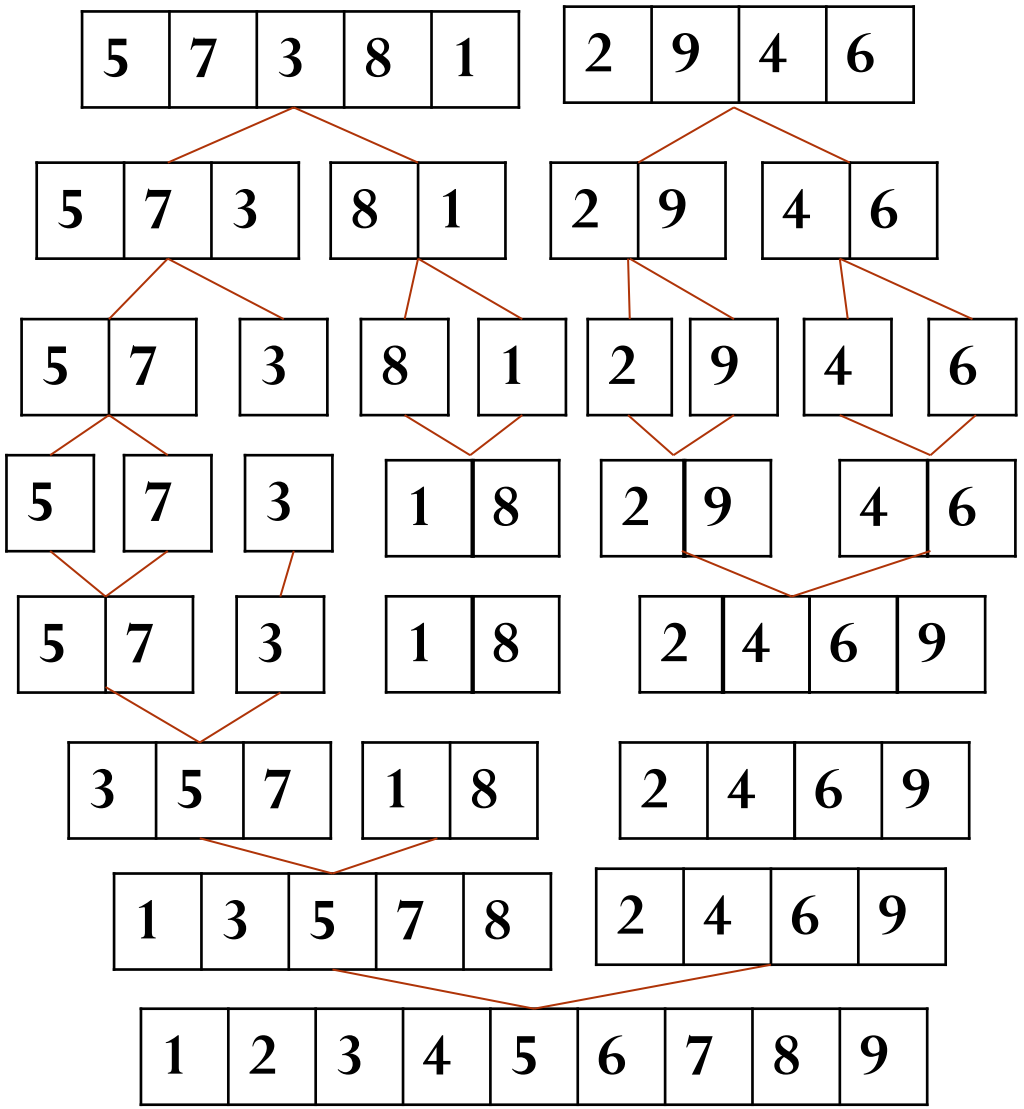
Thuật toán sắp xếp trộn (merge sort)

- Sử dụng mảng trung gian để lưu trữ tạm thời trong quá trình trộn

```
void merge(int A[], int L, int M, int R) {
// tron 2 day da sap A[L..M] va A[M+1..R]
  int i = L; int j = M+1;
  for(int k = L; k <= R; k++){
    if(i > M){ TA[k] = A[j]; j++;}
    else if(j > R){TA[k] = A[i]; i++;}
    else{
      if(A[i] < A[j]){
         TA[k] = A[i]; i++;
      }
      else {
         TA[k] = A[j]; j++;
      }
    }
  }
  for(int k = L; k <= R; k++) A[k] = TA[k];
}
```

Thuật toán sắp xếp trộn (merge sort)

- Ví dụ: 5, 7, 3, 8, 1, 2, 9, 4, 6



Thuật toán sắp xếp nhanh (quick sort)

- Chọn một phần tử bất kỳ dùng làm phần tử trụ (pivot)
- Sắp xếp lại dãy sao cho
  - Các phần tử đứng trước phần tử trụ sẽ không lớn hơn phần tử trụ
  - Các phần tử đứng sau phần tử trụ không nhỏ hơn phần tử trụ
- Khi đó phần tử trụ (có thể bị thay đổi vị trí) đã đứng đúng vị trí trong dãy khi được sắp thứ tự
- Tiến hành sắp xếp dãy con đứng trước và sau phần tử trụ bằng sắp xếp nhanh

Thuật toán sắp xếp nhanh (quick sort)

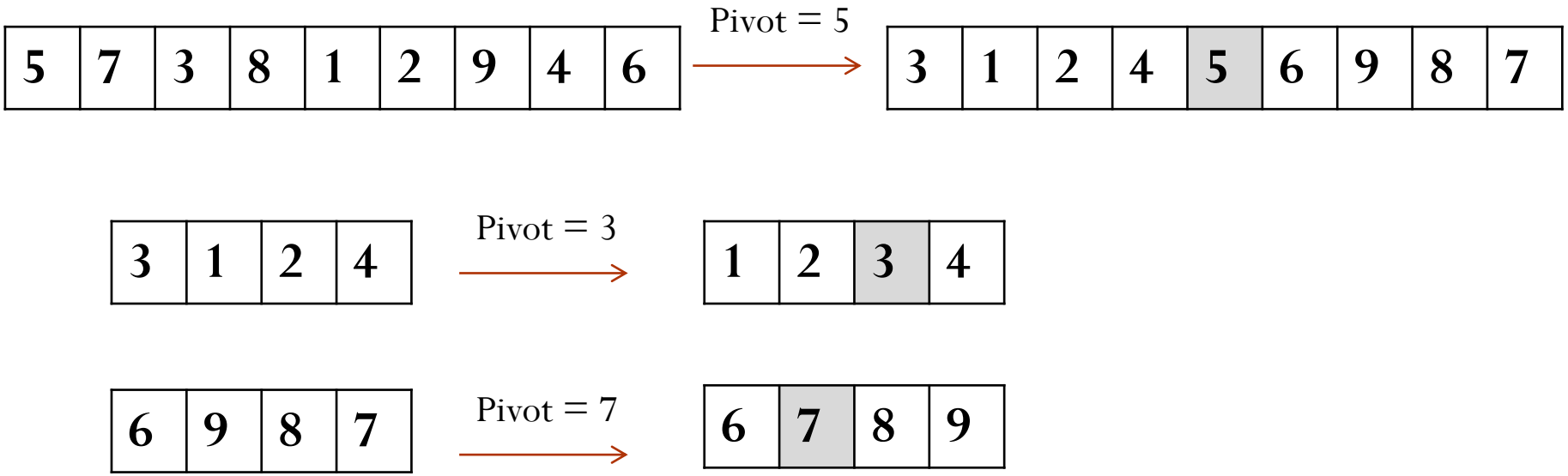
```
void quickSort(int A[], int L, int R) {
  if(L < R){
    int index = (L + R)/2;
    index = partition(A, L, R, index);
    if(L < index)
      quickSort(A, L, index-1);
    if(index < R)
      quickSort(A, index+1, R);
  }
}
```

Thuật toán sắp xếp nhanh (quick sort)

```
int partition(int A[], int L, int R, int indexPivot) {
  int pivot = A[indexPivot];
  swap(A[indexPivot], A[R]);
  int storeIndex = L;
  for(int i = L; i <= R-1; i++){
    if(A[i] < pivot){
      swap(A[storeIndex], A[i]);
      storeIndex++;
    }
  }
  swap(A[storeIndex], A[R]);
  return storeIndex;
}
```

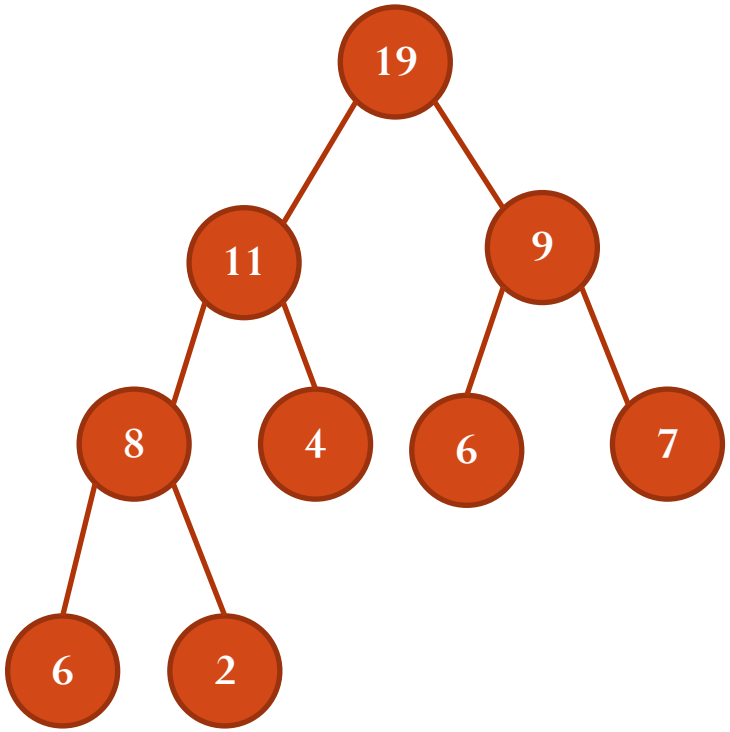
Thuật toán sắp xếp nhanh (quick sort)

- Ví dụ: 5, 7, 3, 8, 1, 2, 9, 4, 6



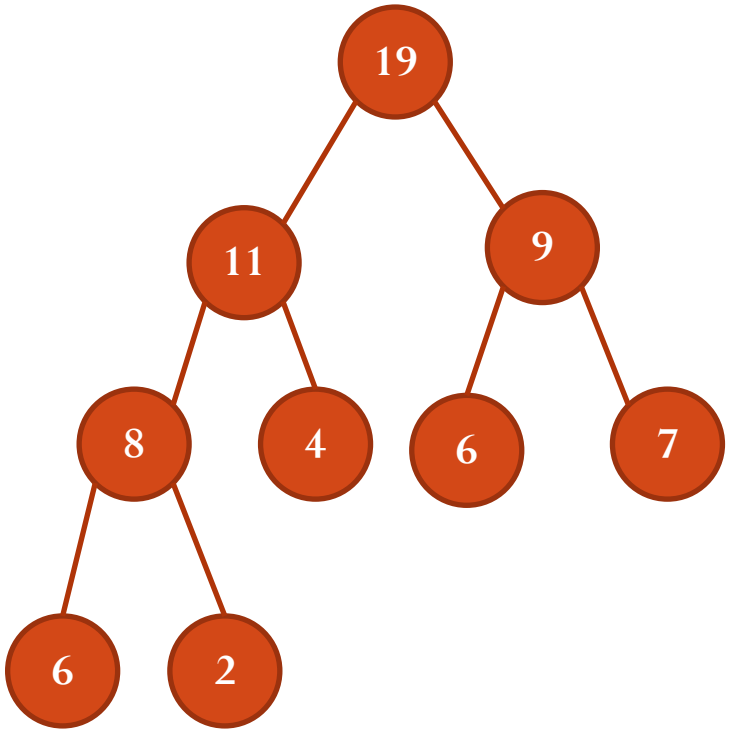
Thuật toán sắp xếp vun đống (heap sort)

- Cấu trúc đống (max-heap)
  - Cây nhị phân đầy đủ (complete tree)
  - Khoá của mỗi nút lớn hơn hoặc bằng khoá của 2 nút con (tính chất của max-heap)
- Ánh xạ từ dãy A[1…N] sang cây nhị phân đầy đủ
  - Gốc là A[1]
  - A[2i] và A[2i+1] là con trái và con phải của A[i]
  - Chiều cao của cây là logN + 1



Thuật toán sắp xếp vun đống (Heap Sort)

- Vun lại đống (heapify)
  - Tình trạng:
    - Tính chất max-heap ở A[i] bị phá vỡ
    - Tính chất max-heap ở các cây con của A[i] đã được thoả mãn
  - Vun lại đống để khôi phục tại tính chất max-heap trên cây gốc A[i]



Thuật toán sắp xếp vun đống (Heap Sort)

- Vun lại đống (heapify)
  - Chọn nút con lớn nhất
  - Đổi chỗ nút con và A[i] cho nhau nếu nút con này lớn hơn A[i] và vun lại đống bắt đầu từ nút con này

```
void heapify(int A[], int i, int N)
{
  int L = 2*i;
  int R = 2*i+1;
  int max = i;
  if(L <= N && A[L] > A[i])
    max = L;
  if(R <= N && A[R] > A[max])
    max = R;
  if(max != i){
    swap(A[i], A[max]);
    heapify(A,max,N);
  }
}
```

Thuật toán sắp xếp vun đống (Heap Sort)

- Sắp xếp vun đống
  - Xây dựng max-heap (thủ tục buildHeap)
  - Đổi chỗ A[1] và A[N] cho nhau
  - Vun lại đống bắt đầu từ A[1] cho A[1..N-1]
  - Đổi chỗ A[1] và A[N-1] cho nhau
  - Vun lại đống bắt đầu từ A[1] cho A[1..N-2]
  - …

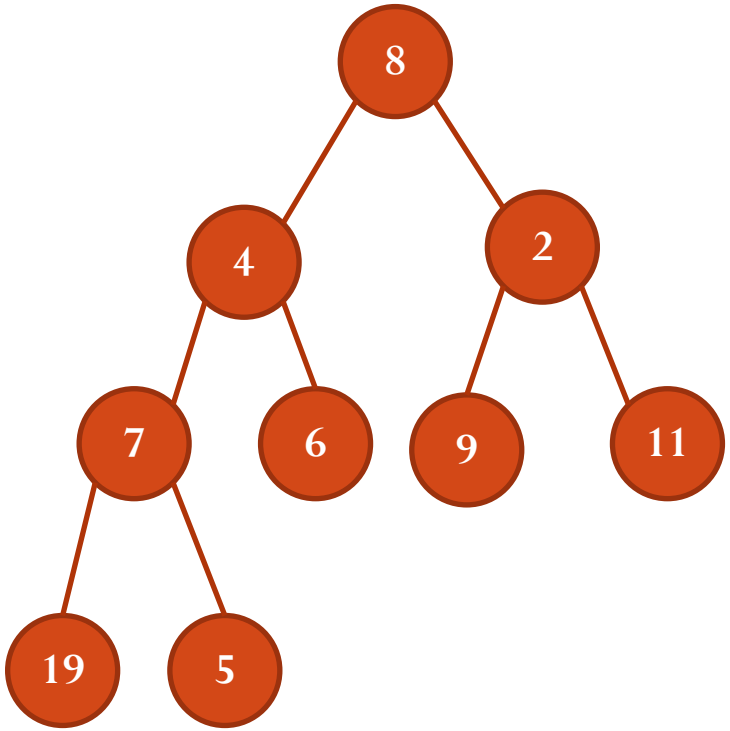
```
void buildHeap(int A[], int N) {
  for(int i = N/2; i >= 1; i--)
    heapify(A,i,N);
}
void heapSort(int A[], int N) {
  // index tu 1 -> N
  buildHeap(A,N);
  for(int i = N; i > 1; i--) {
    swap(A[1], A[i]);
    heapify(A, 1, i-1);
  }
}
```

Thuật toán sắp xếp vun đống (Heap Sort)

- Ví dụ: sắp xếp dãy sau theo thứ tự không giảm của khoá

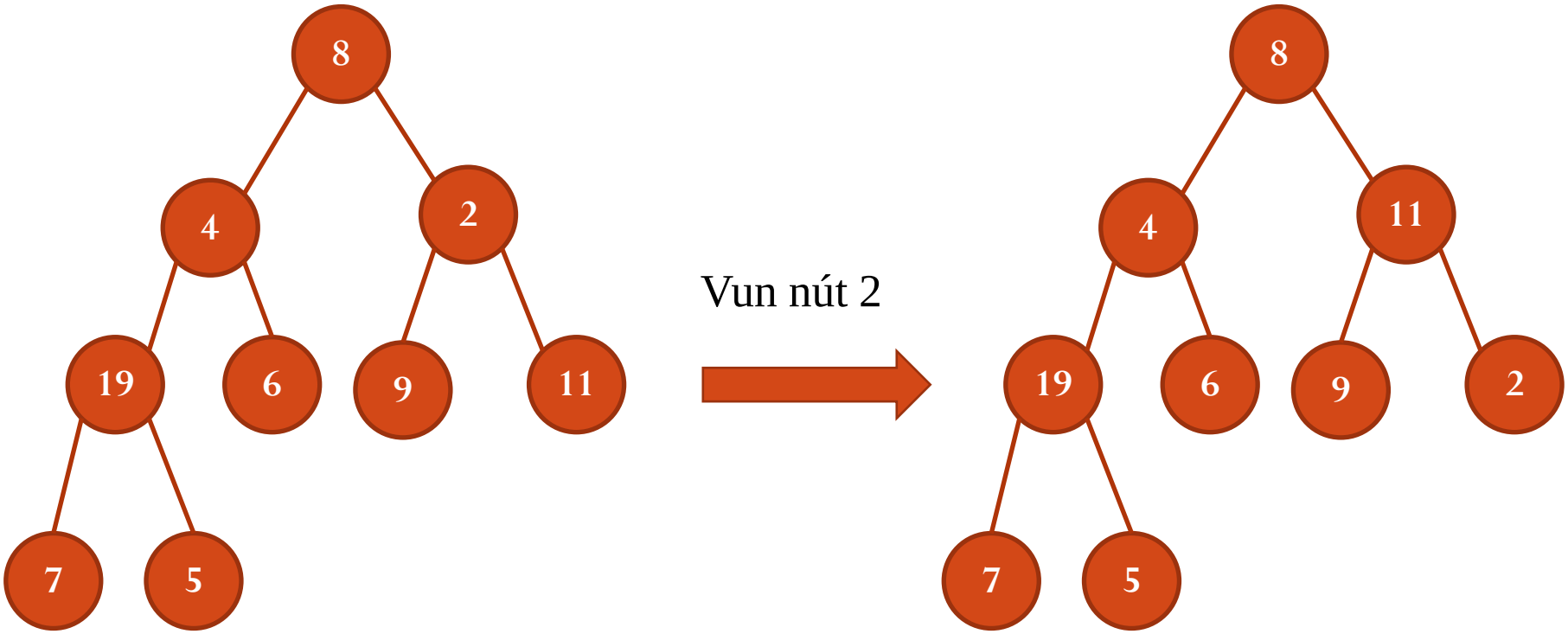
8, 4, 2, 7, 6, 9, 11, 19, 5

Cây nhị phân đầy đủ



8, 4, 2, 7, 6, 9, 11, 19, 5

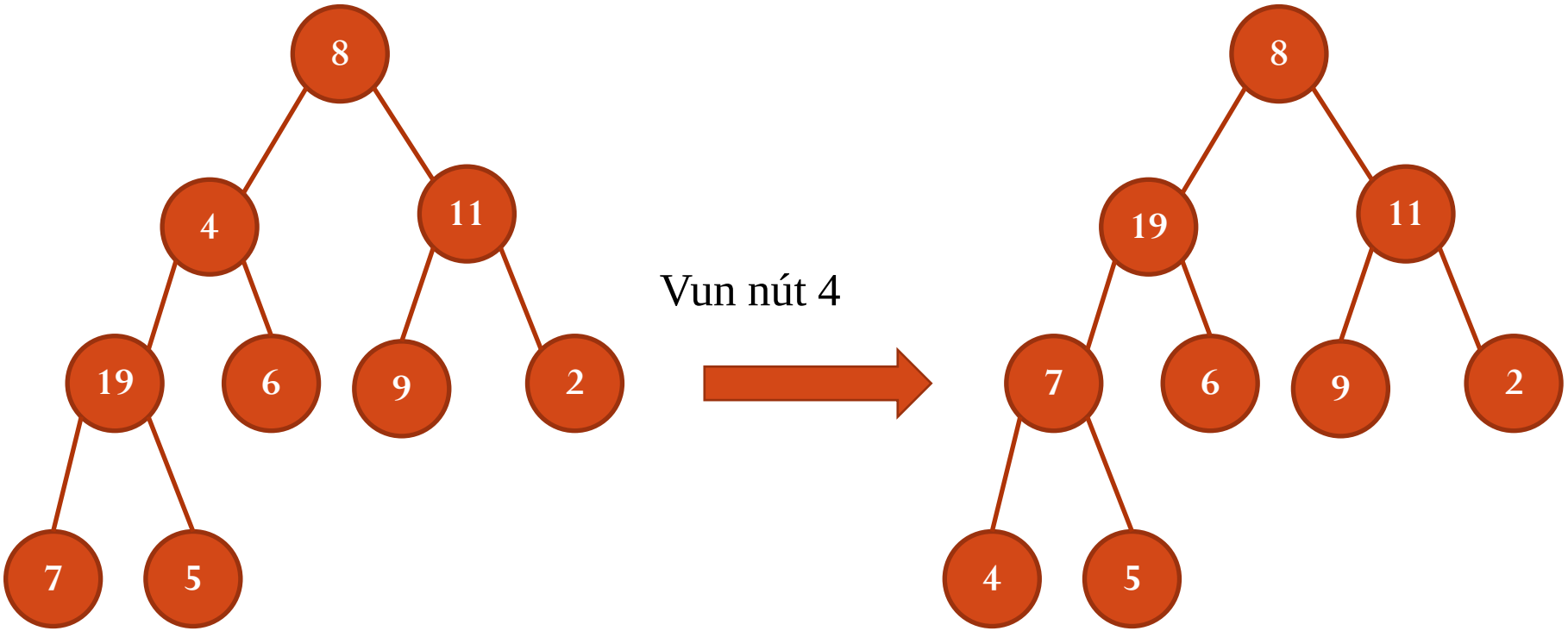
Thuật toán sắp xếp vun đống (Heap Sort)



8, 4, 2, 19, 6, 9, 11, 7, 5

8, 4, 11, 19, 6, 9, 2, 7, 5

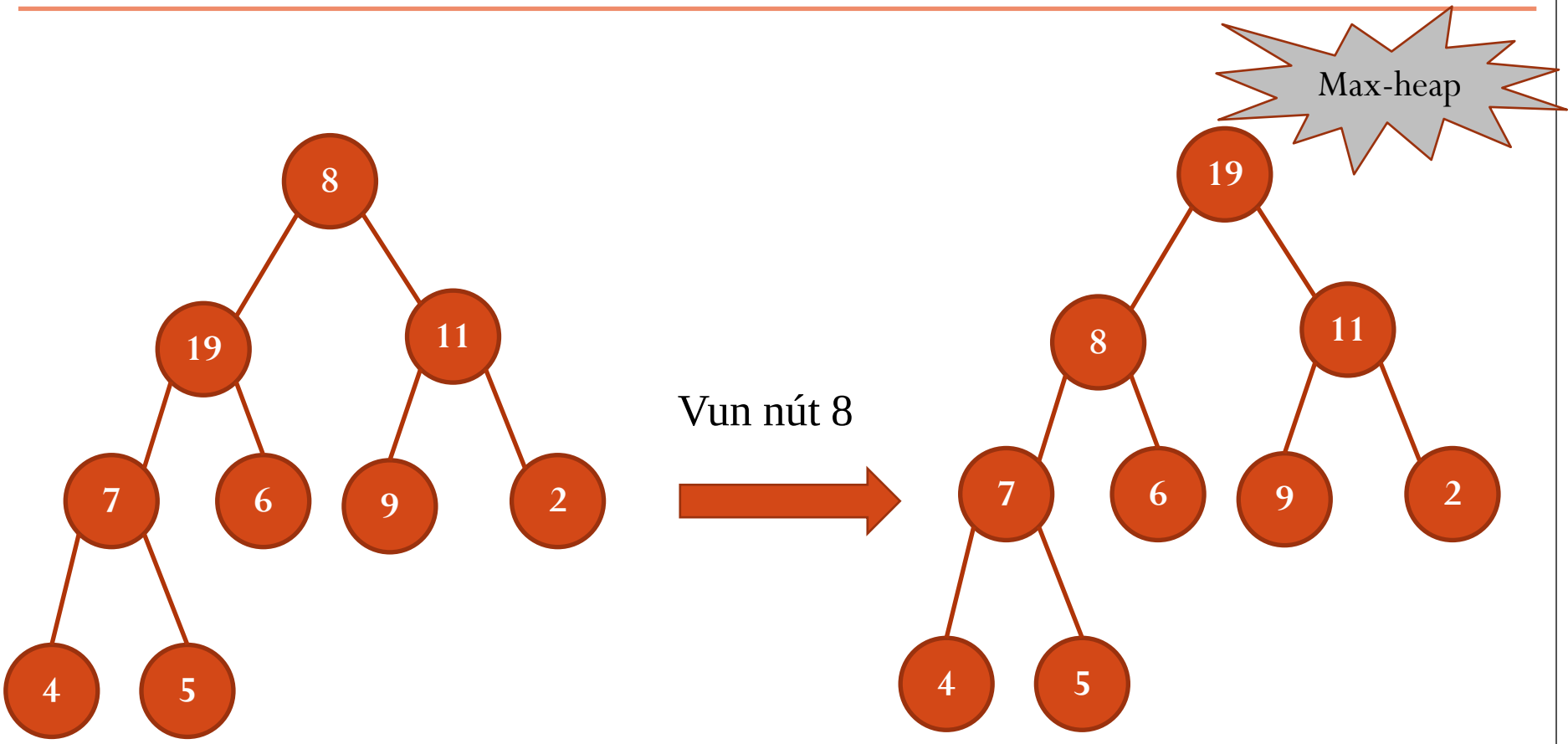
Thuật toán sắp xếp vun đống (Heap Sort)



8, 4, 11, 19, 6, 9, 2, 7, 5

8, 19, 11, 7, 6, 9, 2, 4, 5

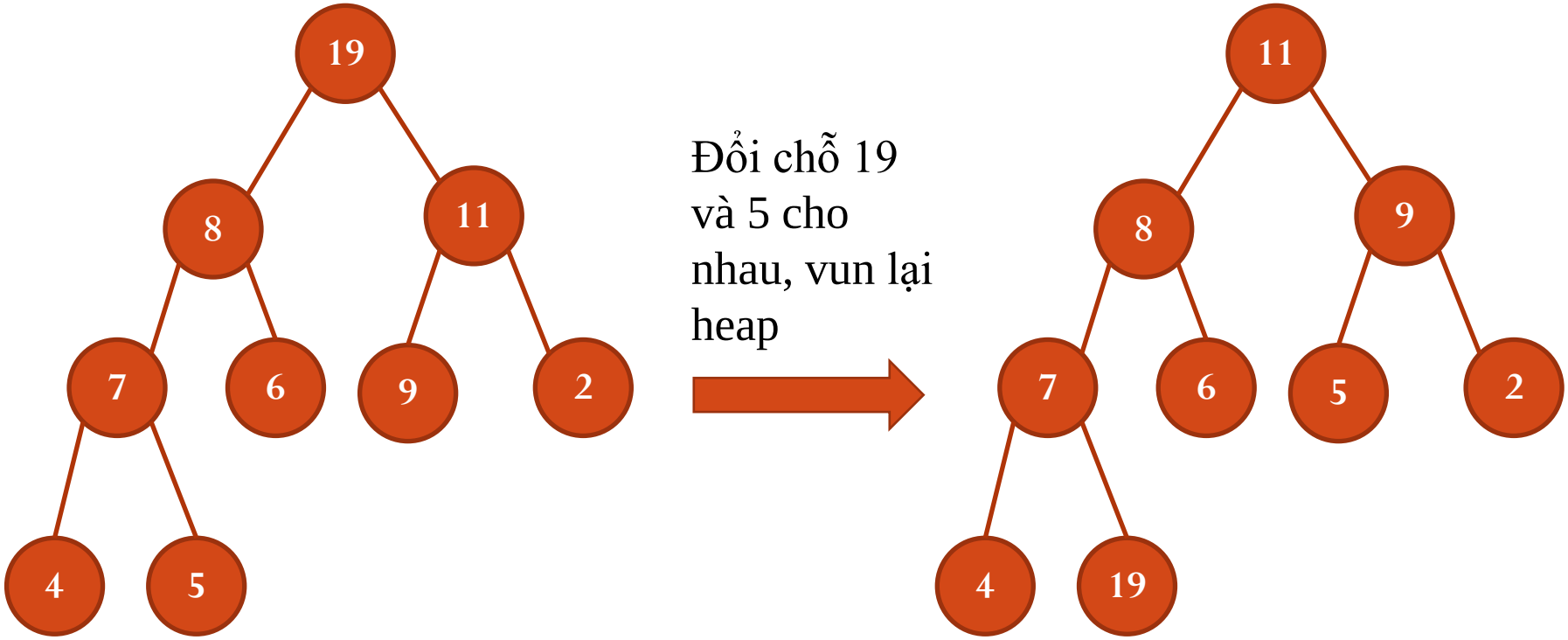
Thuật toán sắp xếp vun đống (Heap Sort)



8, 19, 11, 7, 6, 9, 2, 4, 5

19, 8, 11, 7, 6, 9, 2, 4, 5

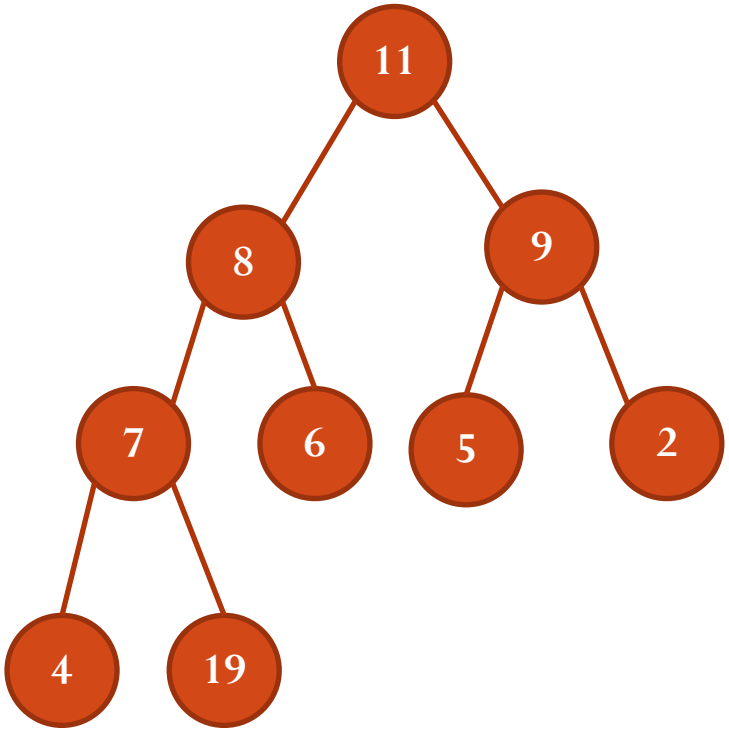
Thuật toán sắp xếp vun đống (Heap Sort)



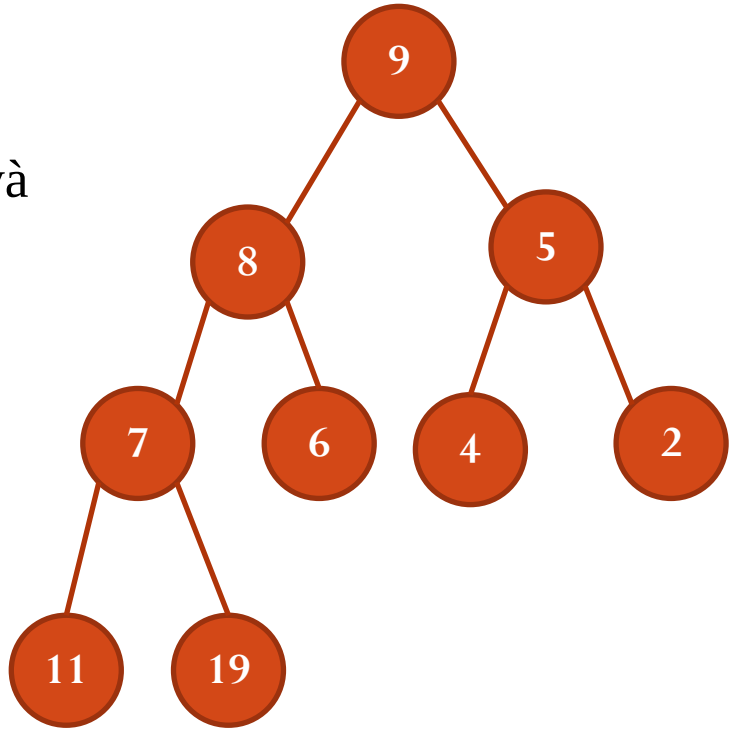
19, 8, 11, 7, 6, 9, 2, 4, 5

11, 8, 9, 7, 6, 5, 2, 4, 19

Thuật toán sắp xếp vun đống (Heap Sort)



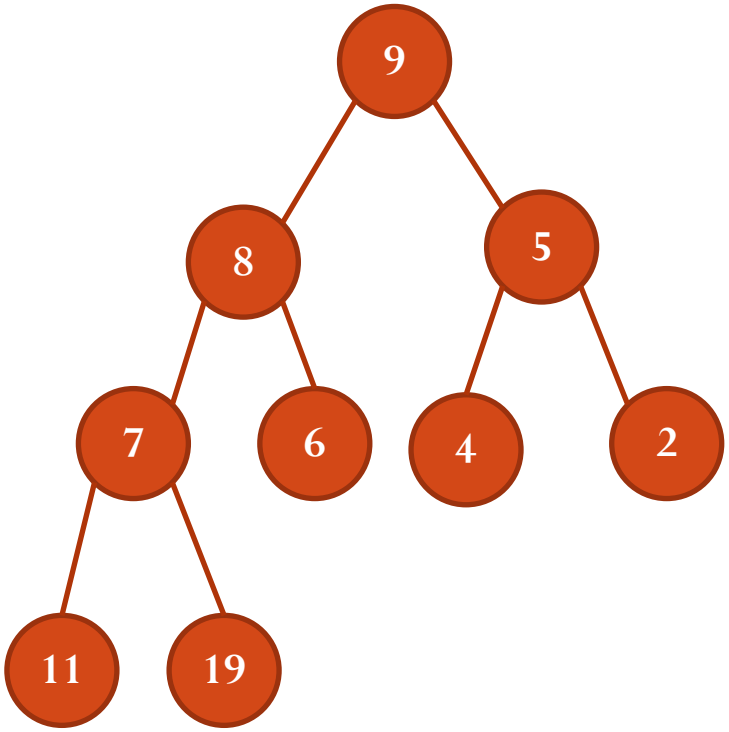
Đổi chỗ 11 và
4 cho nhau,
vun lại heap



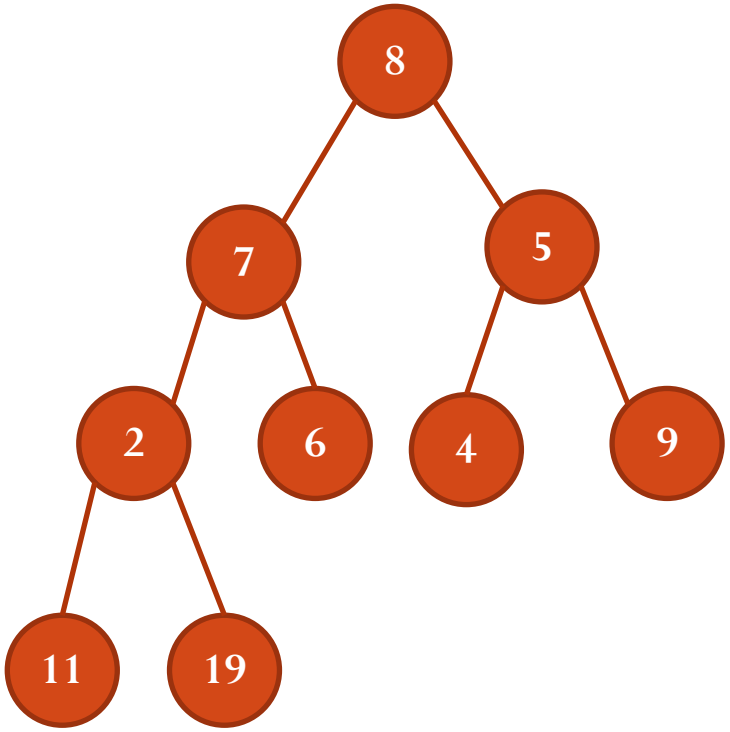
11, 8, 9, 7, 6, 5, 2, 4, 19

9, 8, 5, 7, 6, 4, 2, 11, 19

Thuật toán sắp xếp vun đống (Heap Sort)



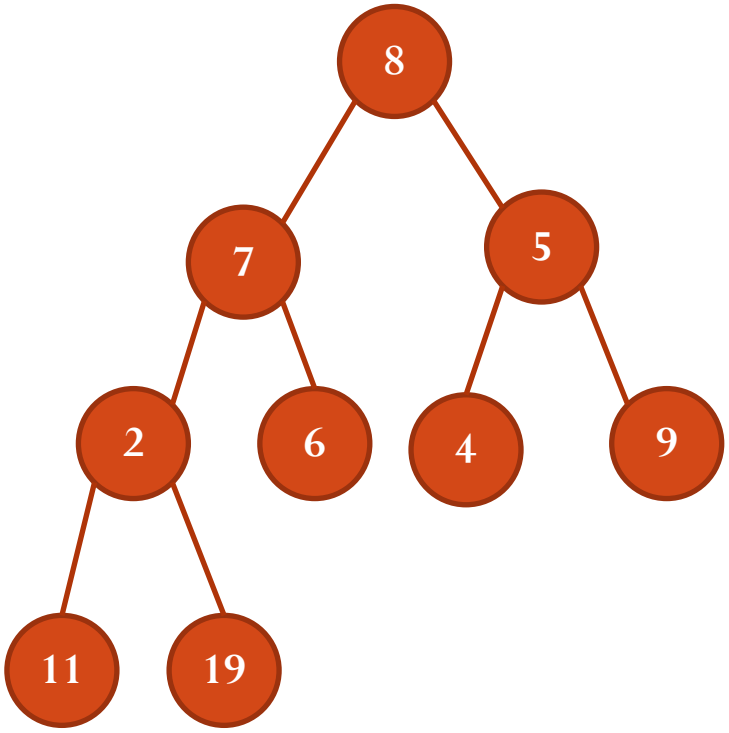
Đổi chỗ 9 và
2 cho nhau,
vun lại heap



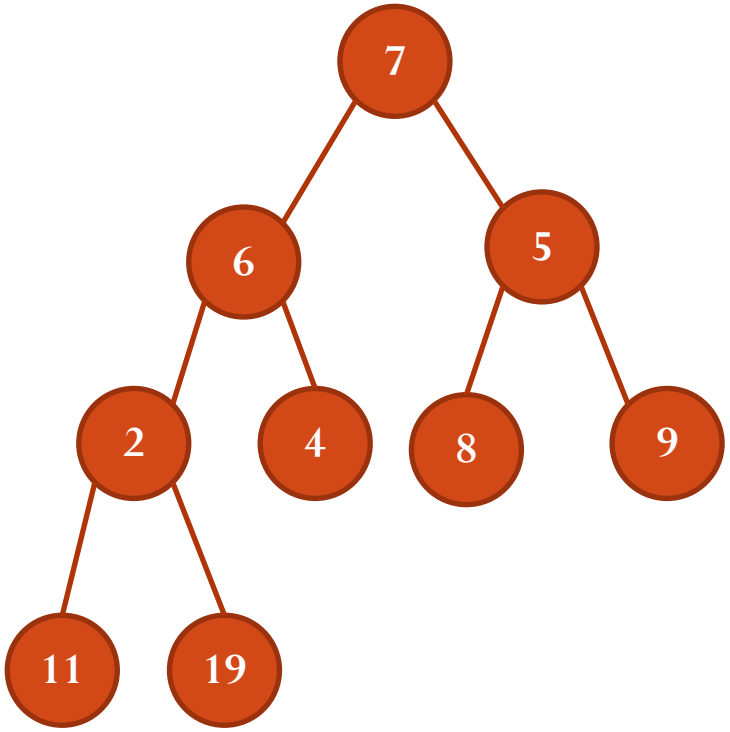
9, 8, 5, 7, 6, 4, 2, 11, 19

8, 7, 5, 2, 6, 4, 9, 11, 19

Thuật toán sắp xếp vun đống (Heap Sort)



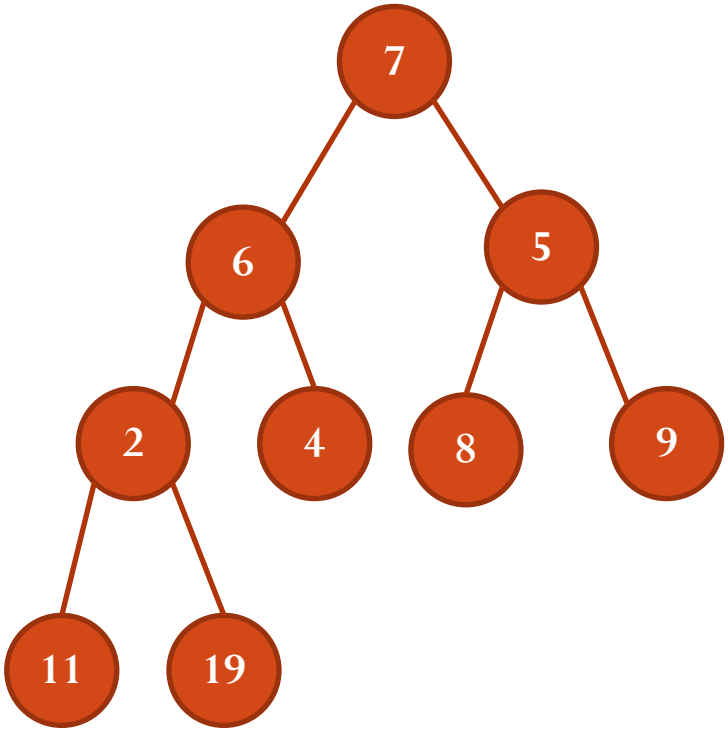
Đổi chỗ 8 và
4 cho nhau,
vun lại heap



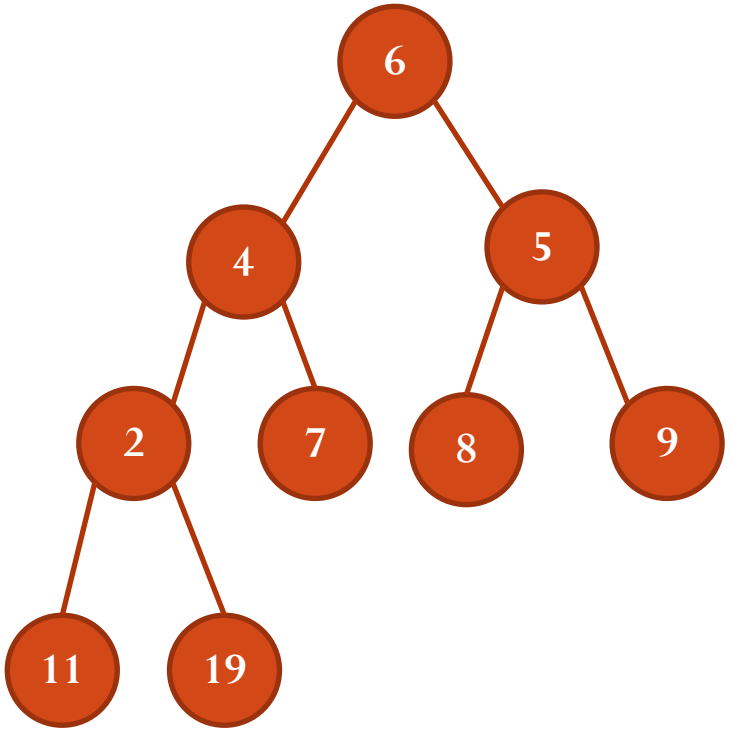
8, 7, 5, 2, 6, 4, 9, 11, 19

7, 6, 5, 2, 4, 8, 9, 11, 19

Thuật toán sắp xếp vun đống (Heap Sort)



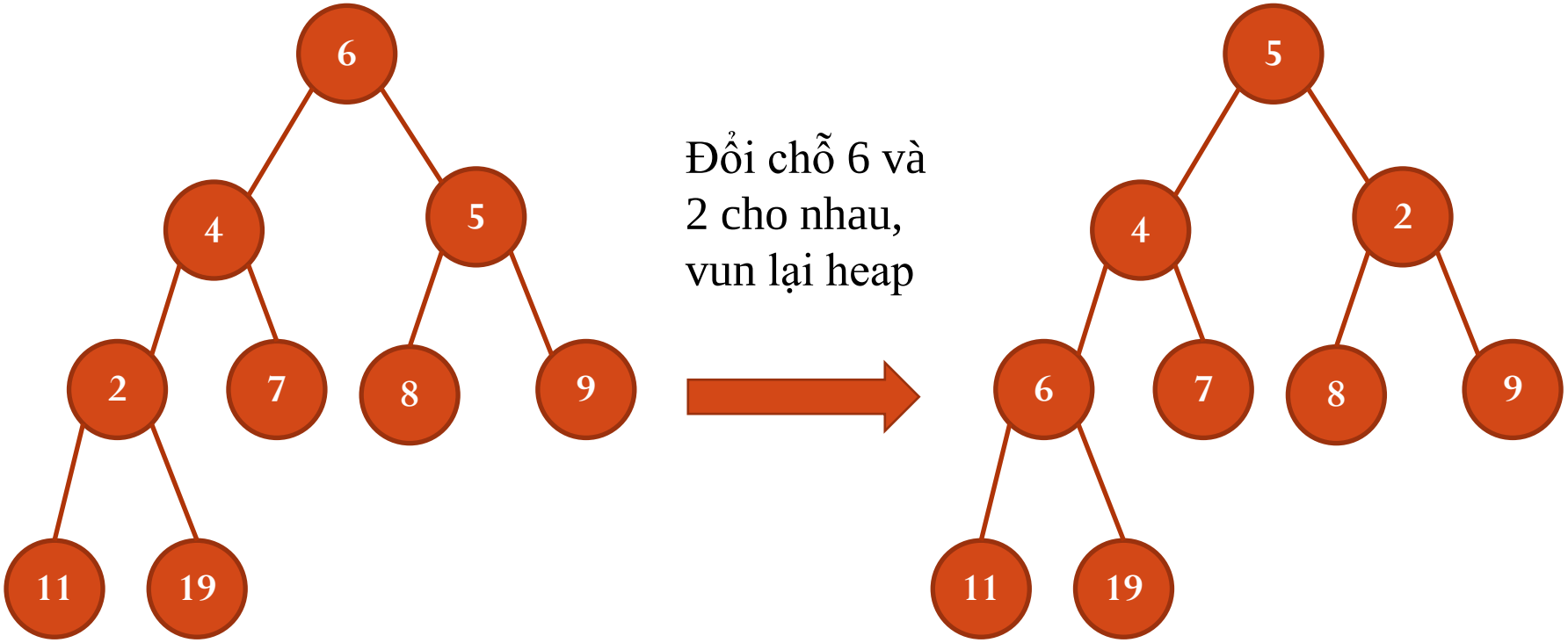
Đổi chỗ 7 và
4 cho nhau,
vun lại heap



7, 6, 5, 2, 4, 8, 9, 11, 19

6, 4, 5, 2, 7, 8, 9, 11, 19

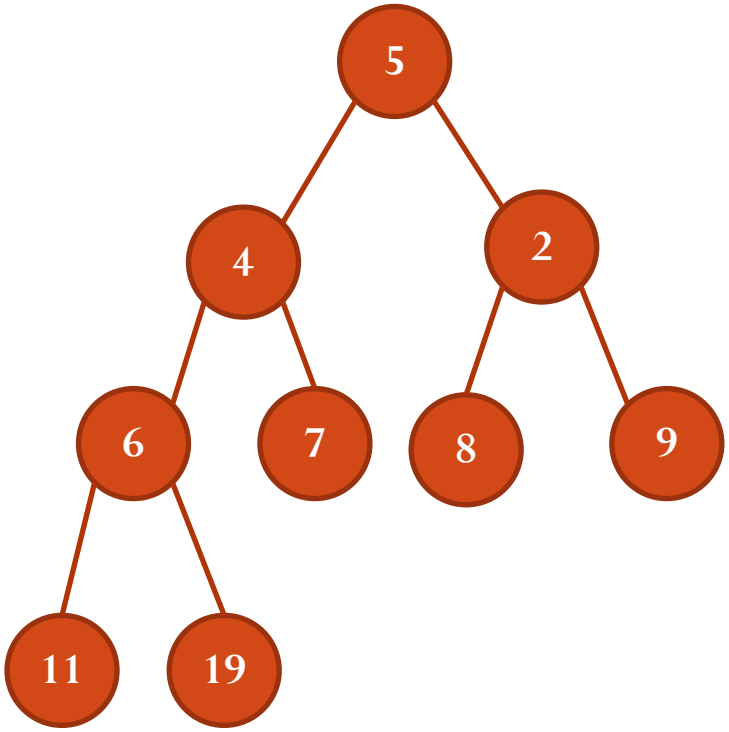
Thuật toán sắp xếp vun đống (Heap Sort)



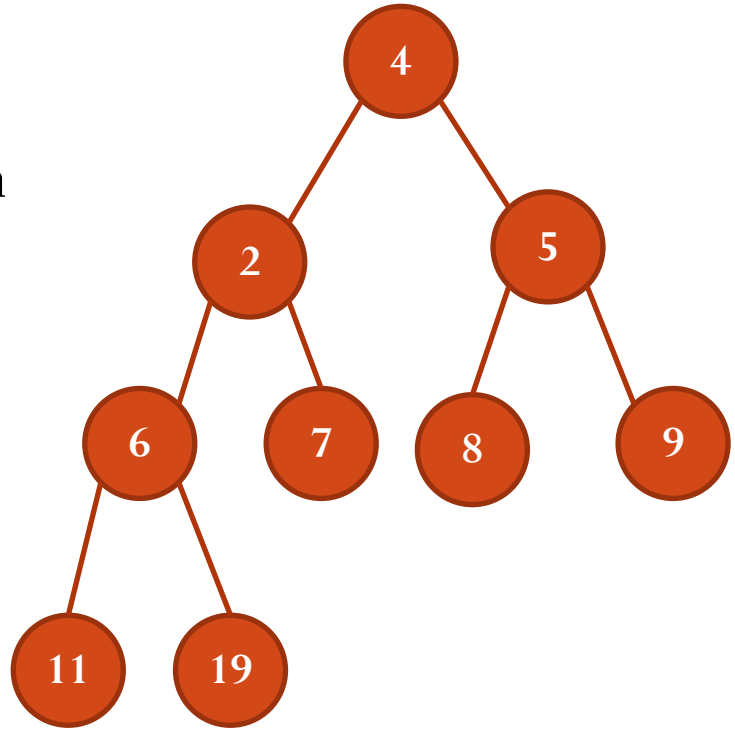
6, 4, 5, 2, 7, 8, 9, 11, 19

5, 4, 2, 6, 7, 8, 9, 11, 19

Thuật toán sắp xếp vun đống (Heap Sort)



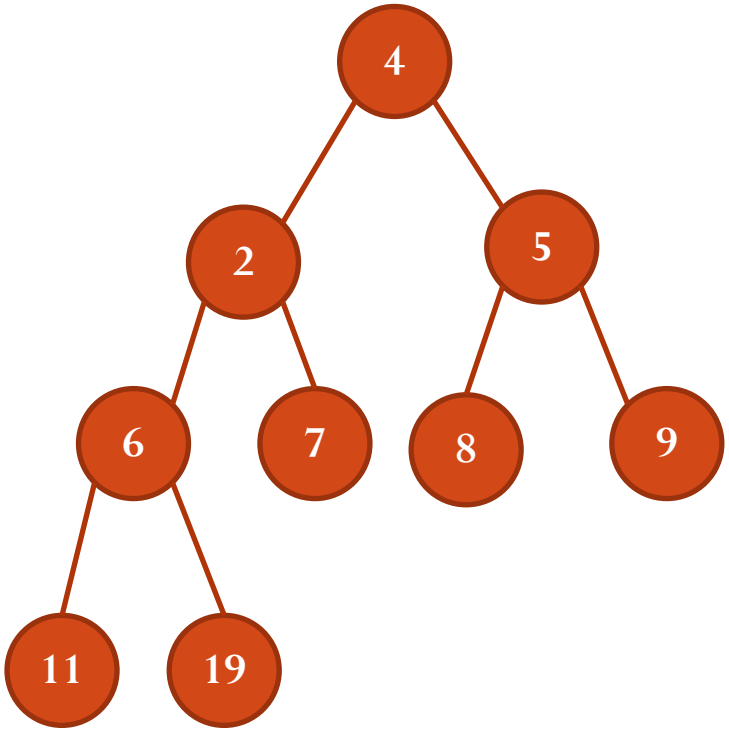
Đổi chỗ 5 và
2 cho nhau,
vun lại heap



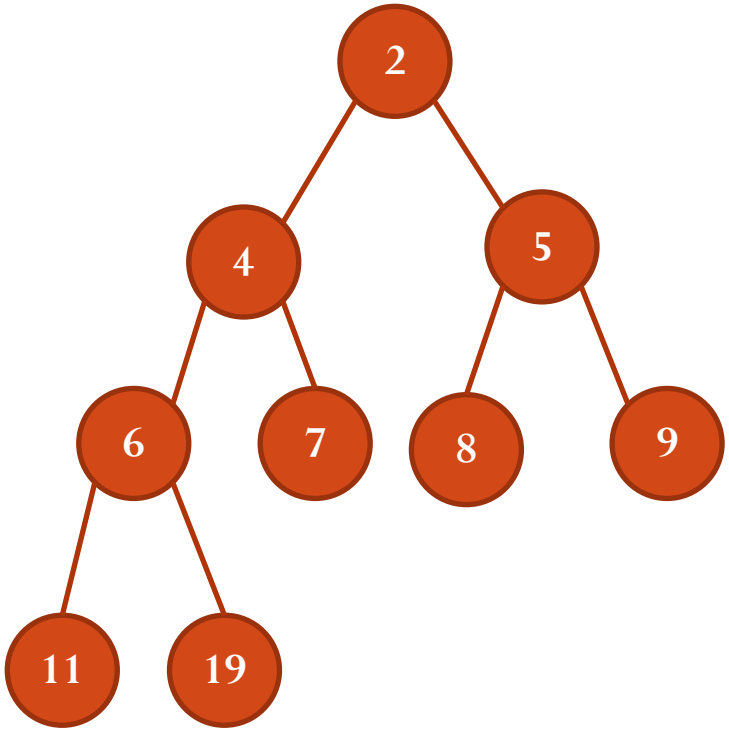
5, 4, 2, 6, 7, 8, 9, 11, 19

4, 2, 5, 6, 7, 8, 9, 11, 19

Thuật toán sắp xếp vun đống (Heap Sort)



Đổi chỗ 4 và
2 cho nhau,
vun lại heap



4, 2, 5, 6, 7, 8, 9, 11, 19

2, 4, 5, 6, 7, 8, 9, 11, 19

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

TÌM KIẾM

Nội dung

- Giới thiệu bài toán tìm kiếm
- Tìm kiếm tuần tự
- Tìm kiếm nhị phân
- Cây nhị phân tìm kiếm
- Cây nhị phân tìm kiếm cân bằng
- Tìm kiếm xâu mẫu
- Ánh xạ và bảng băm

GIỚI THIỆU BÀI TOÁN TÌM KIẾM

- Cần tìm kiếm 1 phần tử nào đó trong một tập dữ liệu
- Bài toán tìm kiếm xuất hiện rất phổ biến trong các bài toán tính toán cũng như các phần mềm ứng dụng
- Tập dữ liệu cần được lưu trữ một cách có cấu trúc để việc tìm kiếm được nhanh chóng và hiệu quả

TÌM KIẾM TUẦN TỰ

- Tập dữ liệu được lưu trữ một cách tuyến tính và không có thông tin gì thêm
- Duyệt lần lượt các phần tử của tập dữ liệu và so sánh với khoá đầu vào

```
sequentialSearch(X[], int L, int R,
    int Y) {
  for(i = L; i <= R; i++)
     if(X[i] = Y) return i;
  return -1;
}
```

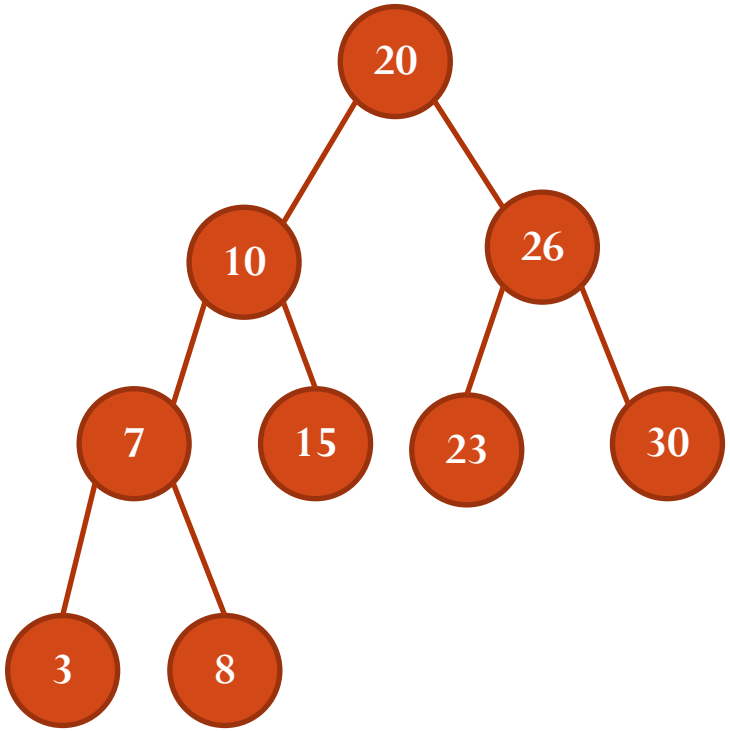
TÌM KIẾM NHỊ PHÂN

- Tập dữ liệu được lưu trữ một cách tuyến tính theo thứ tự không giảm của khoá
- Chia để trị
  - Chia dãy cần tìm thành 2 nửa bằng nhau
  - So sánh khoá đầu vào với phần tử ở giữa và quyết định tiếp tục tìm kiếm nửa bên trái hoặc nửa bên phải tuỳ thuộc vào kết quả so sánh

```
binarySearch(X[], int L, int R,
    int Y) {
  if(L = R){
    if(X[L] = Y) return L;
    return -1;
  }
  int mid = (L+R)/2;
  if(X[mid] = Y) return mid;
  if(X[mid] < Y)
   return binarySearch(X,mid+1,R,Y);
  return binarySearch(X,L,mid-1,Y);
}
```

CÂY NHỊ PHÂN TÌM KIẾM
Binary Search Tree - BST

- Cấu trúc dữ liệu lưu trữ các đối tượng dưới dạng cây nhị phân
  - Khoá của mỗi nút lớn hơn khoá của các nút của cây con trái và nhỏ hơn hoặc bằng khoá của các nút của cây con phải




```
struct Node{
  int key;
  Node* leftChild;
  Node* rightChild;
};
Node* root;
```

CÂY NHỊ PHÂN TÌM KIẾM

- Các thao tác
  - `Node* makeNode(int v)`: tạo ra một nút có khoá v và trả về con trỏ trỏ đến nút tạo được
  - `Node* insert(Node* r, int v)`: tạo một nút mới có khoá v và chèn vào BST có gốc là r
  - `Node* search(Node* r, int v)`: tìm và trả về con trỏ trỏ đến nút có khoá là v trên BST có gốc r
  - `Node* findMin(Node* r)`: trả về con trỏ trỏ đến nút có khoá nhỏ nhất
  - `Node* del(Node* r, int v)`: loại bỏ nút có khoá v ra khỏi cây nhị phân tìm kiếm gốc được trỏ bởi r

CÂY NHỊ PHÂN TÌM KIẾM

```
Node* makeNode(int v) {
  Node* p = new Node;
  p->key = v;
  p->leftChild = NULL;
  p->rightChild = NULL;
  return p;
}
```

CÂY NHỊ PHÂN TÌM KIẾM

```
Node* insert(Node* r, int v) {
  if(r == NULL)
    r = makeNode(v);
  else if(r->key > v)
    r->leftChild = insert(r->leftChild,v);
  else if(r->key <= v)
    r->rightChild = insert(r->rightChild,v);
  return r;
}
```

CÂY NHỊ PHÂN TÌM KIẾM

```
Node* search(Node* r, int v) {
  if(r == NULL)
    return NULL;
  if(r->key == v)
    return r;
  else if(r->key > v)
    return search(r->leftChild, v);
  return search(r->rightChild, v);
}
```

CÂY NHỊ PHÂN TÌM KIẾM

```
Node* findMin(Node* r) {
  if(r == NULL)
    return NULL;
  Node* lmin = findMin(r->leftChild);
  if(lmin != NULL) return lmin;
  return r;
}
```

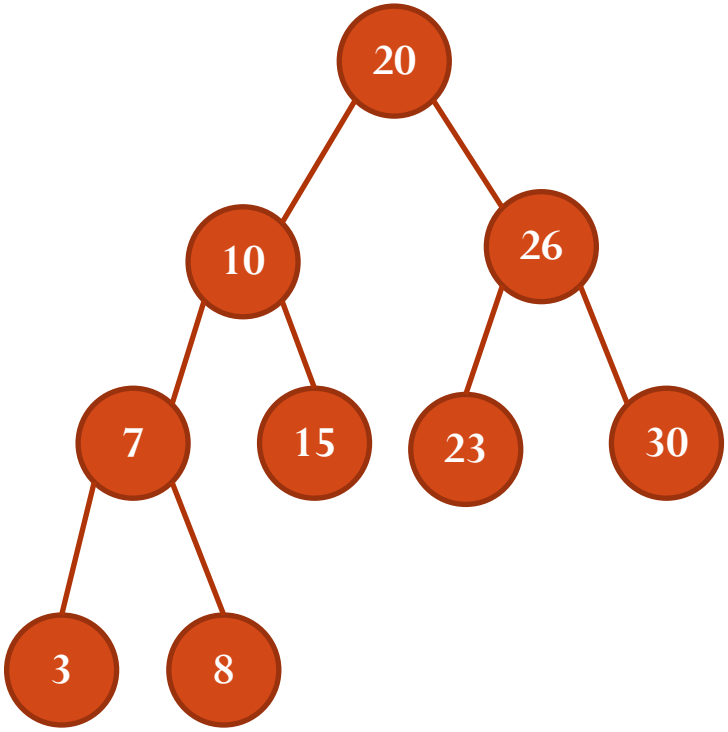
CÂY NHỊ PHÂN TÌM KIẾM

```
Node* del(Node* r, int v) {
  if(r == NULL)
    return NULL;
  if(v < r->key) r->leftChild = del(r->leftChild, v);
  if(v > r->key) r->rightChild = del(r->rightChild, v);
  if(r->leftChild != NULL && r->rightChild != NULL){
    Node* tmp = findMin(r->rightChild);
    r->rightChild = del(r->rightChild, tmp->key);
  }else{
    Node* tmp = r;
    if(r->leftChild == NULL) r = r->rightChild;
    else r = r->leftChild;
    delete tmp;
  }
  return r;
}
```

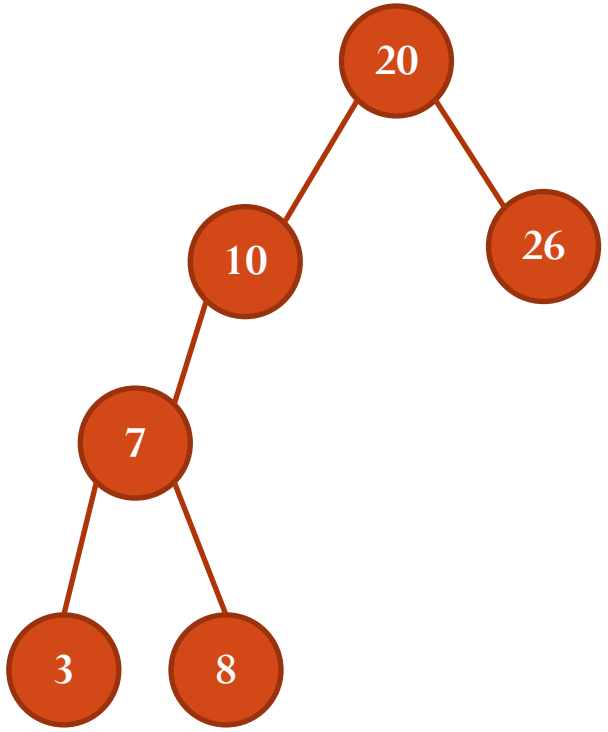
CÂY NHỊ PHÂN TÌM KIẾM

- Cây nhị phân tìm kiếm cân bằng (AVL)
  - Là một BST
  - Chênh lệch độ cao của nút con trái và con phải của mỗi nút không quá 1
  - Độ cao của cây logN (N là số nút)
  - Mỗi thao tác thêm, loại bỏ nút trên cây AVL cần bảo tồn tính cân bằng của cây

CÂY NHỊ PHÂN TÌM KIẾM



Cây AVL



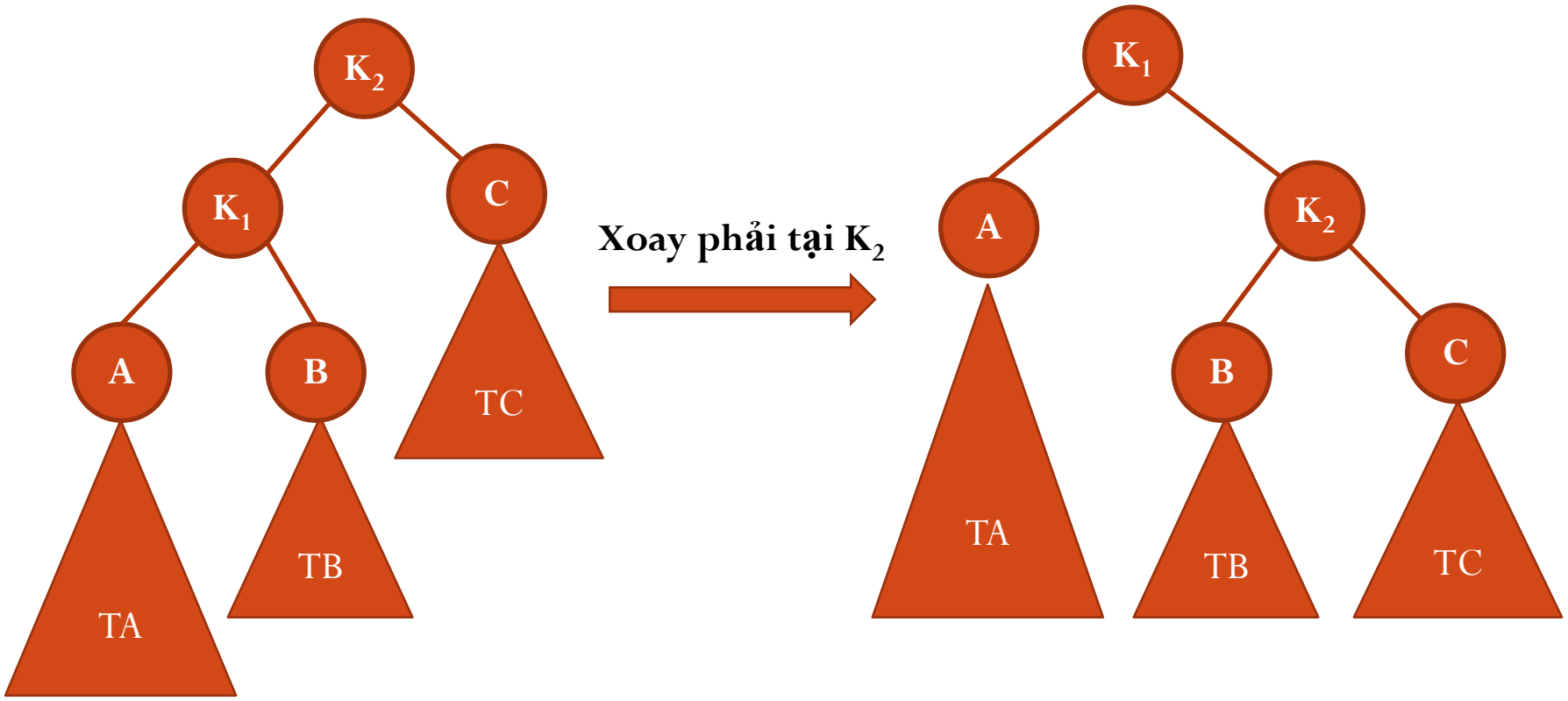
Cây BST nhưng không là AVL

CÂY NHỊ PHÂN TÌM KIẾM

- Mỗi thao tác loại bỏ hoặc thêm mới 1 nút trên AVL có thể làm mất tính cân bằng
  - Chênh lệch độ cao giữa 2 nút con của mỗi nút cùng lắm là 2 đơn vị
  - Thực hiện các phép xoay để khôi phục lại thuộc tính cân bằng

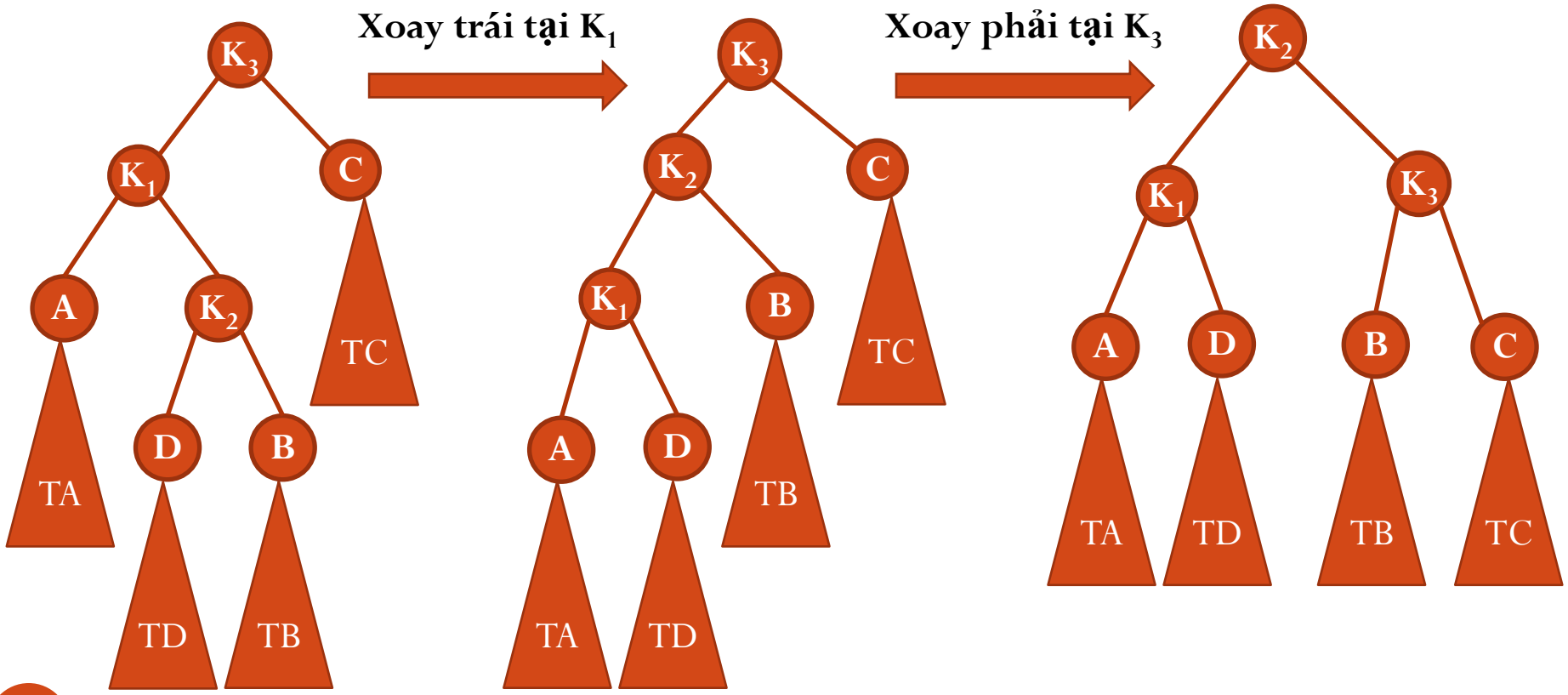
CÂY NHỊ PHÂN TÌM KIẾM

Trường hợp 1: chênh lệch độ cao của K_1 và C là 2, độ cao của B và C bằng nhau, độ cao của A hơn độ vào của B là 1 đơn vị



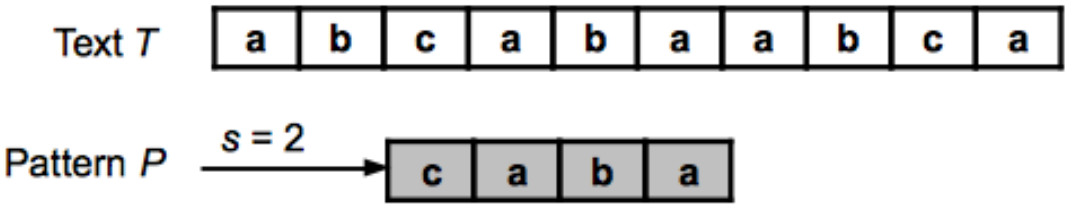
CÂY NHỊ PHÂN TÌM KIẾM

Trường hợp 2:



TÌM KIẾM XÂU MẪU

- Tìm sự xuất hiện của các xâu trong 1 văn bản cho trước
  - Cho văn bản T được biểu diễn bởi 1 mảng các ký tự $T[1..N]$ và 1 xâu (pattern) $P[1..M]$.
  - Cần tìm tất cả các vị trí xuất hiện của P trong T
  - P được gọi là xuất hiện trong T với độ lệch s nếu $P[i] = T[s+i]$ với mọi $i = 1,\ldots, M$



- Ứng dụng
  - Trình soạn thảo văn bản
  - Trích chọn thông tin
  - Xử lý chuỗi ADN

TÌM KIẾM XÂU MẪU

- Thuật toán trực tiếp
- Thuật toán Boyer Moore
- Thuật toán Rabin-Karp
- Thuật toán KMP (Knuth-Morris Pratt)

TÌM KIẾM XÂU MẪU

- Thuật toán trực tiếp
  - Xâu mẫu trượt từ trái qua phải của T
  - So khớp được thực hiện từ trái qua phải
  - Khi gặp trường hợp không khớp (mismatch) thì thực hiện dịch chuyển mẫu P ***một*** vị trí sang phải trên T

```
naiveSM(P, T) {
  foreach s = 0 . . N-M do
    i = 1;
    while i <= M && P[i] = T[i+s] do
      i = i + 1;
    endwhile
    if i > M then output(s);
  endfor
}
```

TÌM KIẾM XÂU MẪU

- Thuật toán Boyer Moore
  - Trượt xâu mẫu từ trái qua phải
  - Đối sánh: phải qua trái
  - Sử dụng thông tin tiền xử lý để bỏ qua càng nhiều ký tự càng tốt
  - Tiền xử lý xâu mẫu *P*
    - Last[*x*]: vị trí bên phải nhất xuất hiện ký tự *x* trong *P*
  - Khi tình trạng không khớp xảy ra với ký tự tồi là *x* (ký tự trên *T*), *P* sẽ được trượt 1 khoảng:

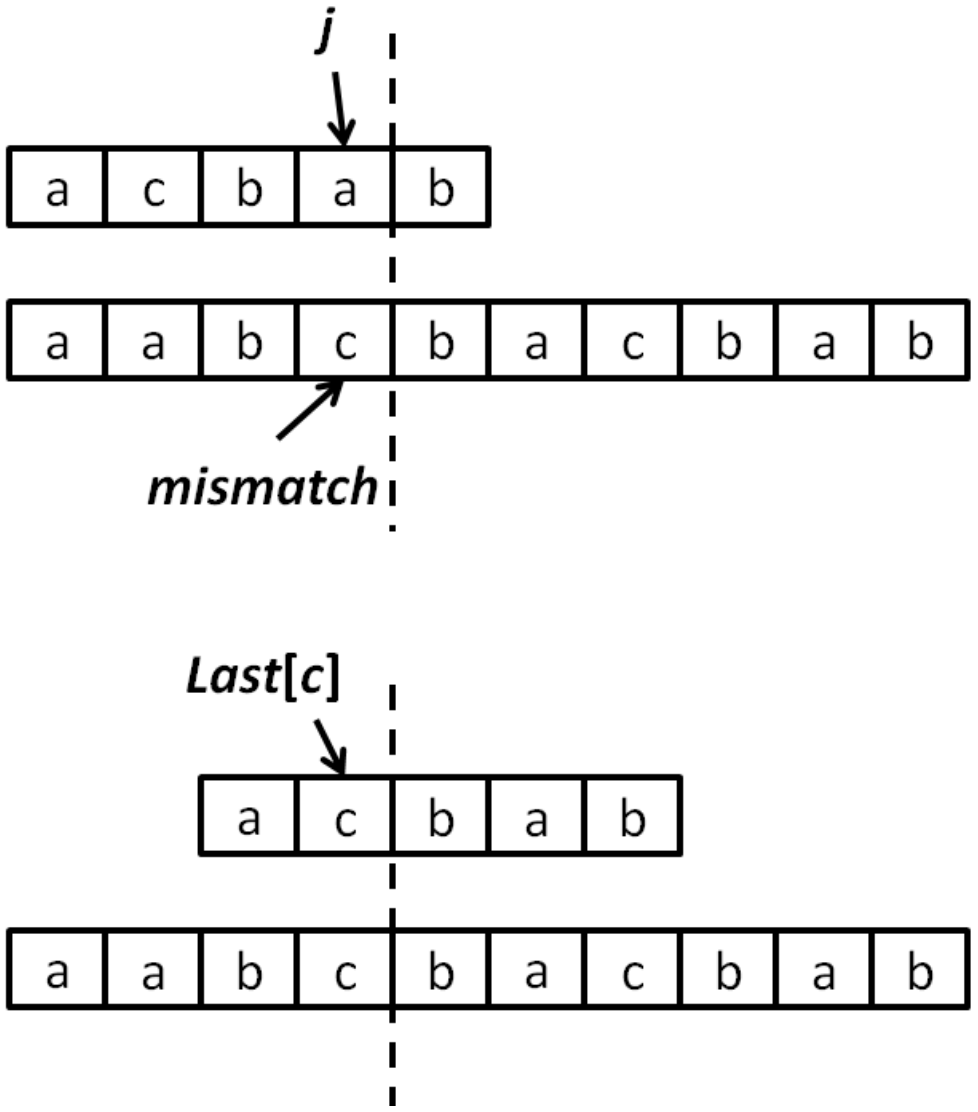
$$\max\{j - \text{Last}[x], 1\}$$

trong đó *j* là chỉ số hiện tại (xảy ra không khớp) trên *P* khi so khớp ký tự từ phải qua trái

```
computeLast(P) {
  for c = 0..255 do last[c] = 0;
  for i = m downto 1 do {
    if(last[P[i]] = 0) last[P[i]] = i;
  }
}
boyerMoore(P, T){
  s = 0;
  while(s <= N-M){
    j = M;
    while(j > 0 && T[j+s] = P[j])
      j = j-1;
    if(j = 0){
      output(s); s = s + 1;
    }else{
      k = last[T[j+s]];
      s = s + max(j-k,1);
    }
  }
}
```

TÌM KIẾM XÂU MẪU

- Thuật toán Boyer Moore



TÌM KIẾM XÂU MẪU

- Thuật toán Rabin-Karp
  - Mỗi ký tự trong bảng chữ cái được biểu diễn bởi 1 số nguyên không âm nhỏ hơn d (d là độ dài của bảng chữ cái)
  - Đổi xâu $P[1..M]$ sang giá trị số nguyên dương

$$p = P[1]*d^{M-1} + P[2]*d^{M-2} + \ldots + P[M]*d^{0}$$

  - Đối sánh mẫu bằng cách so sánh 2 giá trị số nguyên dương tương ứng
  - Sử dụng lược đồ Horner để tăng tốc độ tính toán
  - Đổi xâu con $T[s+1 .. s+M]$ sang số

$$T_s = T[s+1]*d^{M-1} + T[s+2]*d^{M-2} + \ldots + T[s+M]*d^{0}$$

  - T_{s+1} có thể được tính toán hiệu quả dựa vào T_s (được tính trước đó)

$$T_{s+1} = (T_s - T[s+1]*d^{M-1})*d + T[s+M+1]$$

TÌM KIẾM XÂU MẪU

- Thuật toán Rabin-Karp
  - Nhược điểm
    - Khi *M* lớn thì việc chuyển đổi xâu sang số mất thời gian đáng kể,
    - Có thể gây ra tràn số đối với kiểu dữ liệu cơ bản của ngôn ngữ lập trình

⇒ Cách giải quyết: thực hiện phép chia lấy đối với giá trị số dư cho Q

  - Khi 2 số dư khác nhau có nghĩa 2 giá trị số khác nhau và 2 xâu tương ứng cũng khác nhau
  - Khi 2 số dư bằng nhau, tiến hành đối sánh từng ký tự như cách truyền thống

TÌM KIẾM XÂU MẪU

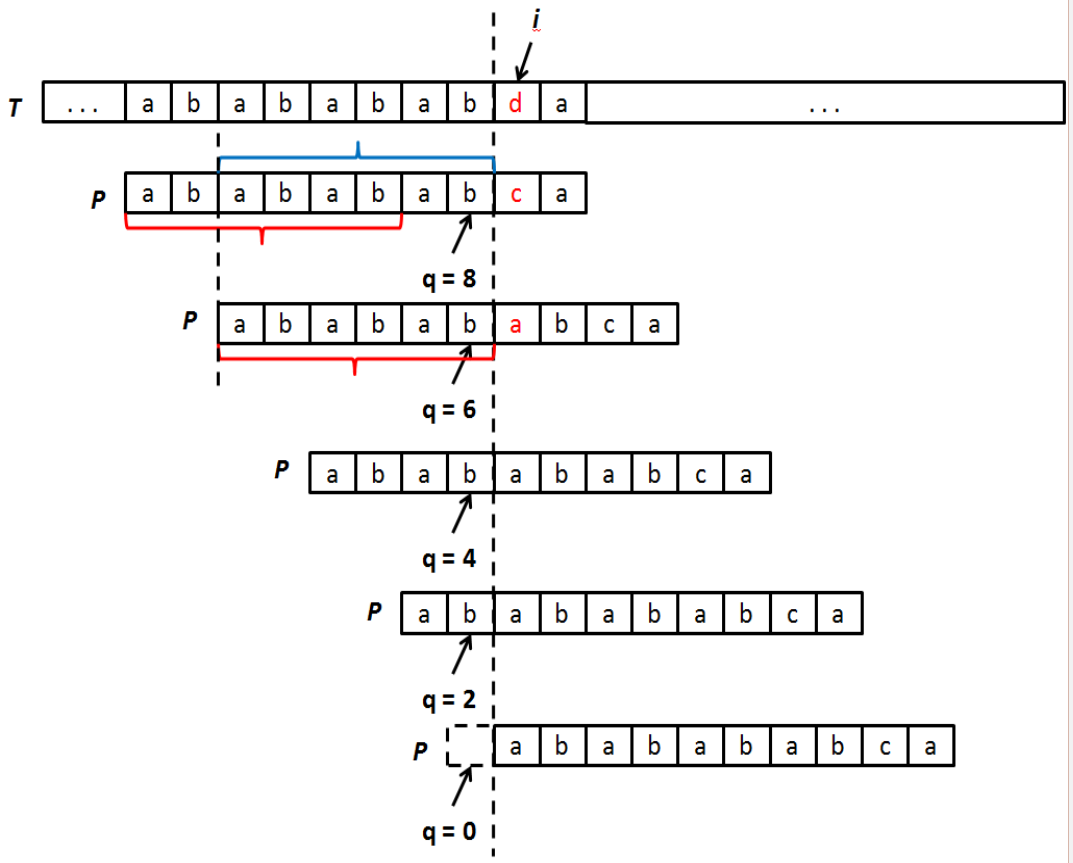
- Thuật toán KMP (Knuth-Morris Pratt)
  - Đối sánh: từ trái qua phải
  - Trượt: từ trái qua phải
  - π[q]: độ dài của tiền tố dài nhất cũng đồng thời là hậu tố ***ngặt*** của xâu P[1..q]

q	1	2	3	4	5	6	7	8	9	10
$P[q]$	a	b	a	b	a	b	a	b	c	a
$\pi[q]$	0	0	1	2	3	4	5	6	0	1

```
computePI(P){
  pi[1] = 0;
  k = 0;
  for q = 2..M do {
    while(k > 0 && P[k+1] != P[q])
      k = pi[k];
    if P[k+1] = P[q] then
      k = k + 1;
    pi[q] = k;
  }
}
```

TÌM KIẾM XÂU MẪU

- Thuật toán KMP (Knuth-Morris Pratt)



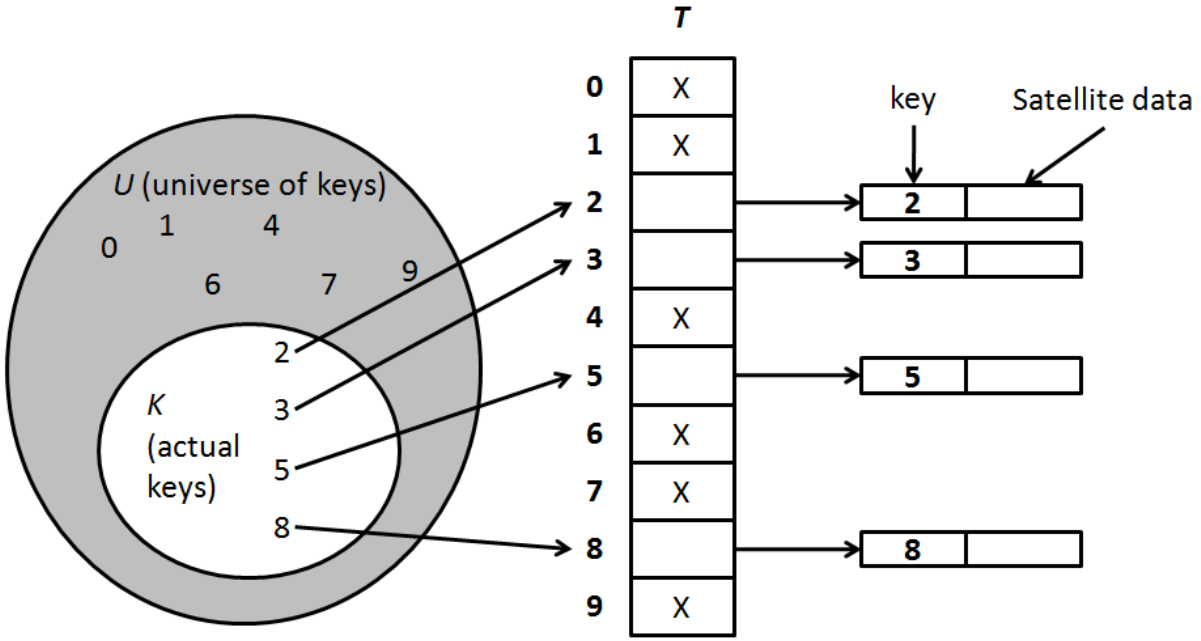

```
kmp(P, T){
  q = 0;
  for i = 1..N do {
    while q > 0 && P[q+1] != T[i]
      q = pi[q];
    if P[q+1] = T[i]
      q = q + 1;
    if(q = M){
      output(i-M+1);
      q = pi[q];
    }
  }
}
```

ÁNH XẠ VÀ BẢNG BĂM

- Từ điển: cấu trúc dữ liệu để ánh xạ một đối tượng với 1 đối tượng khác
  - put(*k*,*v*): thiết lập ánh xạ giữa khóa *k* và giá trị *v*
  - get(*k*): trả về giá trị tương ứng với khóa *k*
- Để cài đặt từ điển có thể dùng
  - Cây nhị phân tìm kiếm
  - Bảng băm

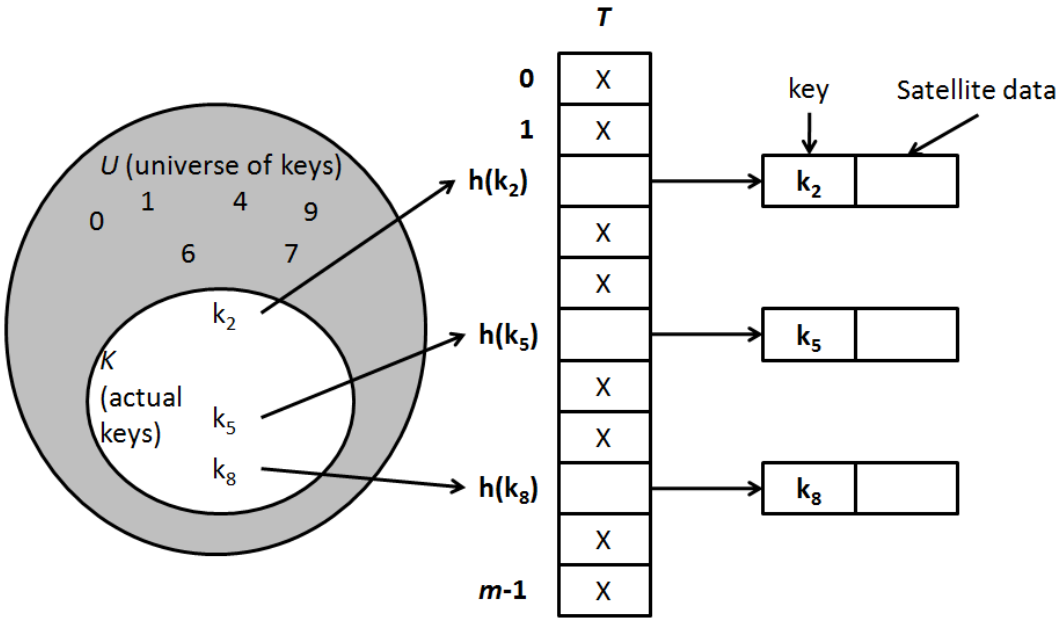
ÁNH XẠ VÀ BẢNG BĂM

- Phương pháp địa chỉ trực tiếp
  - Giá trị của khóa *k* sẽ là địa chỉ trực tiếp trong bảng nơi lưu trữ giá trị tương ứng với *k* trong từ điển
  - Ưu điểm: nhanh, đơn giản
  - Nhược điểm: Hiệu quả sử dụng bộ nhớ kém khi không gian khóa sử dụng có giá trị khóa rất khác nhưng số lượng lại ít



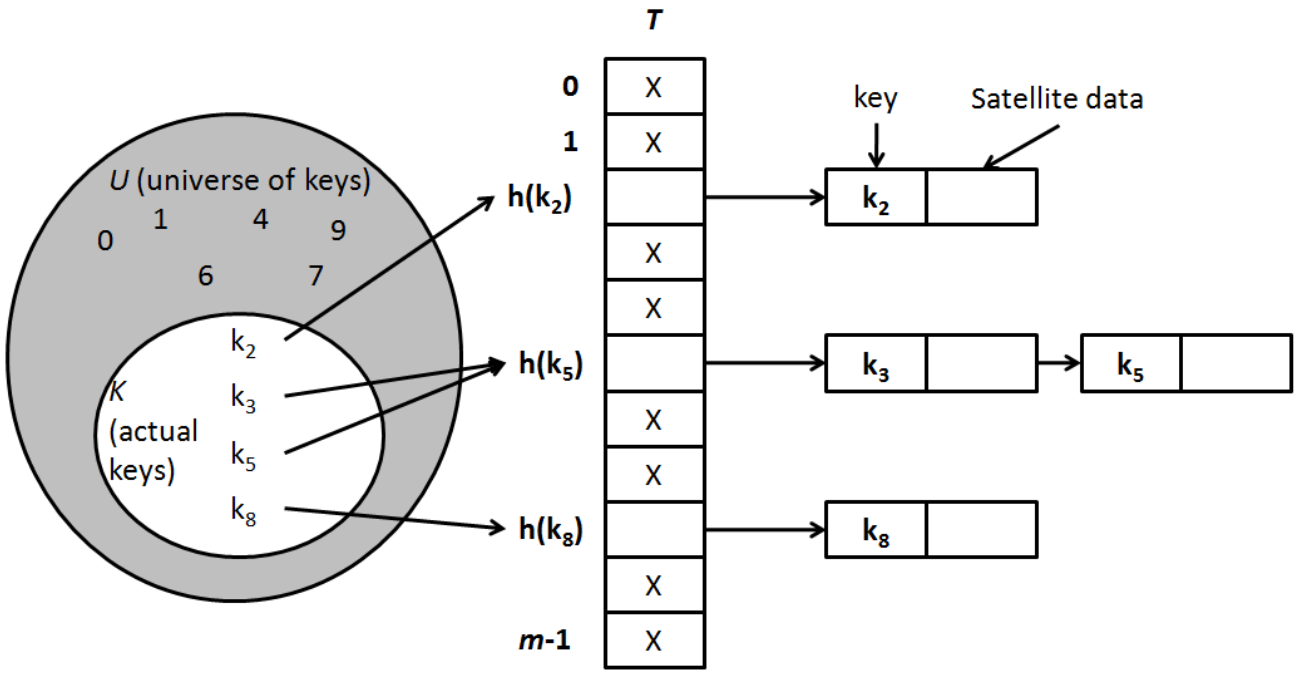
ÁNH XẠ VÀ BẢNG BĂM

- Phương pháp hàm băm
  - Với mỗi khóa *k*, hàm *h*(*k*) sẽ đưa ra địa chỉ trong bảng nơi lưu trữ giá trị tương ứng với *k*
  - Hàm *h*(*k*) phải đơn giản và tính toán phải hiệu quả



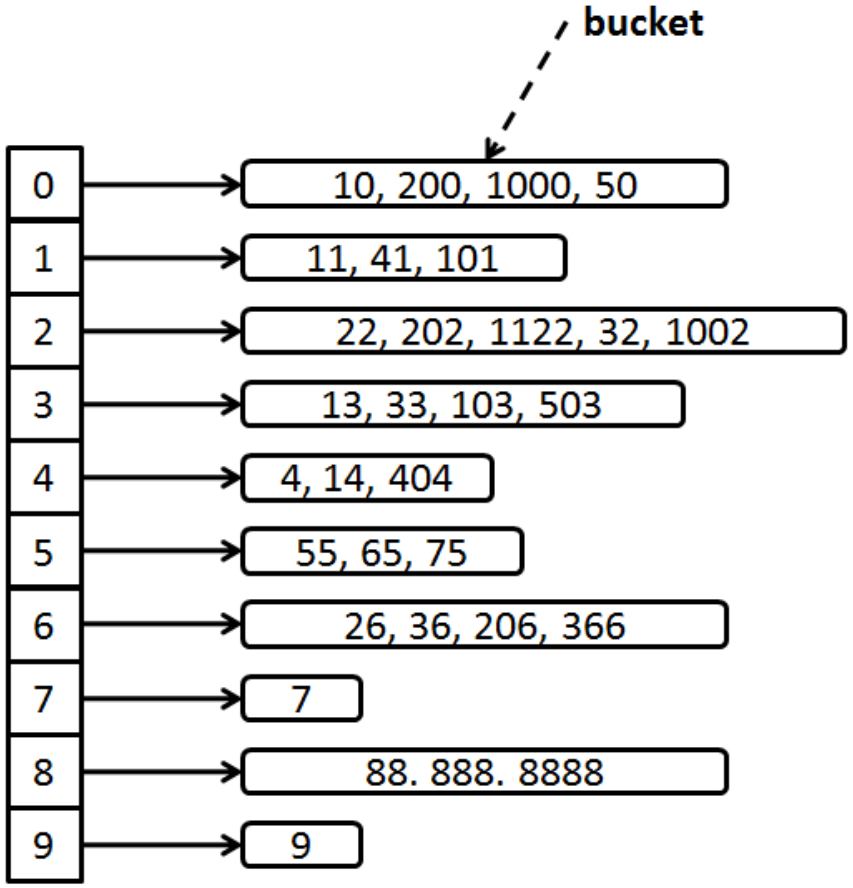
ÁNH XẠ VÀ BẢNG BĂM

- Phương pháp hàm băm
  - Xung đột: hai khoá khác nhau cho hai giá trị hàm băm bằng nhau
  - Giải pháp: các đối tượng cho cùng giá trị hàm băm sẽ được nhóm theo chuỗi (danh sách liên kết, cây hoặc bảng băm cấp 2)



ÁNH XẠ VÀ BẢNG BĂM

- Hàm băm phổ biến: hàm chia lấy dư (mod)
  - Giả thiết khóa là các số nguyên
  - h(k) = k mod m trong đó m là số phần tử của bảng lưu trữ



CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

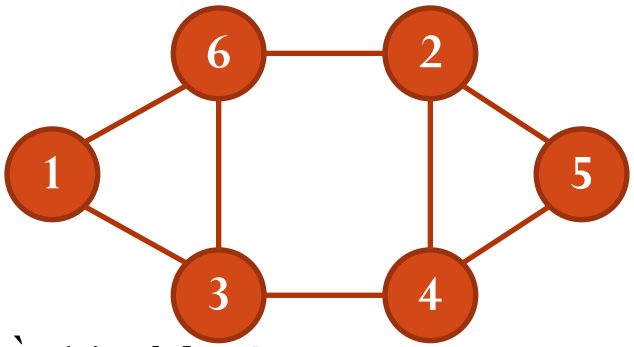
ĐỒ THỊ VÀ ỨNG DỤNG

Đồ thị và ứng dụng

- Khái niệm và định nghĩa
- Biểu diễn đồ thị
- Duyệt đồ thị
- Đồ thị Euler và đồ thị Hamilton
- Cây khung nhỏ nhất của đồ thị
- Đường đi ngắn nhất trên đồ thị

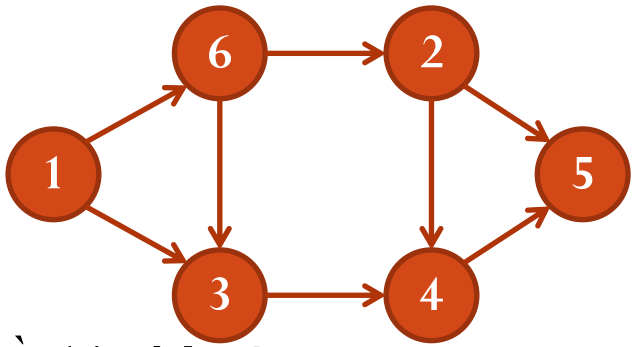
Khái niệm và định nghĩa

- Đồ thị là một đối tượng toán học mô hình hoá các thực thể và các liên kết giữa các thực thể đó
- Đồ thị $G = (V, E)$ trong đó V là tập đỉnh và E là tập cạnh (cung)
- Với mỗi $(u, v) \in E$: ta nói v kề với u



Đồ thị vô hướng

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(1, 3), (1,6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 6), (4, 5)\}$



Đồ thị vô hướng

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(1, 3), (1,6), (2, 4), (2, 5), (6, 2), (3, 4), (6, 3), (4, 5)\}$

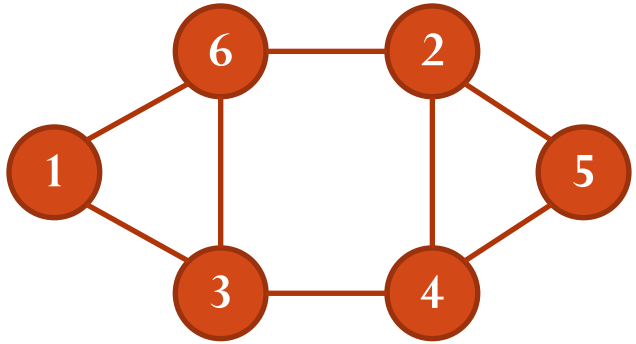
Khái niệm và định nghĩa

- Bậc của một đỉnh trên đồ thị vô hướng là số đỉnh kề với đỉnh đó:

$$deg(v) = \#\{u \mid (u, v) \in E\}$$

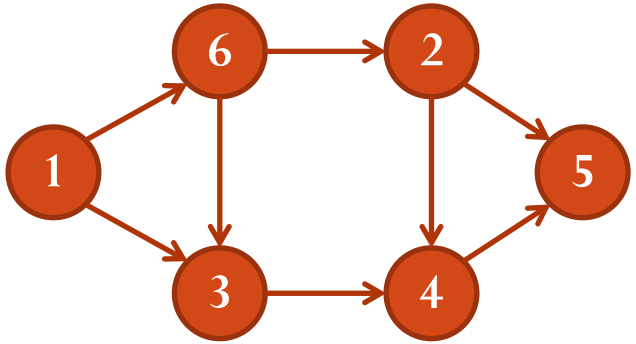
- Bán bậc vào (ra) của một đỉnh trên đồ thị có hướng là số cung đi vào (ra) đỉnh đó:

$$deg^-(v) = \#\{u \mid (u, v) \in E\},\ deg^+(v) = \#\{u \mid (v, u) \in E\}$$



đồ thị vô hướng

$deg(1) = 2$, $deg(4) = 3$

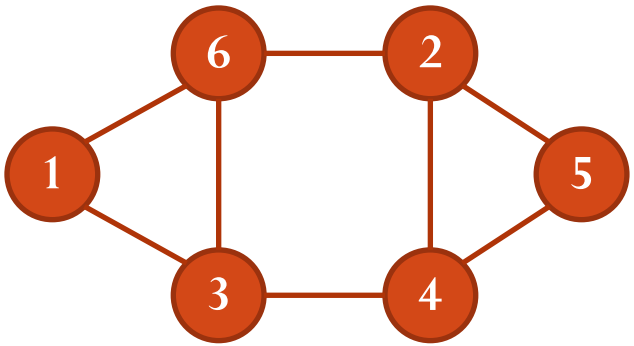


đồ thị có hướng

$deg^-(1) = 0$, $deg^+(1)=2$

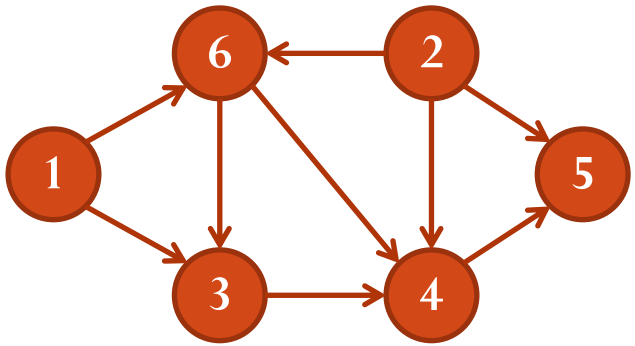
Khái niệm và định nghĩa

- Cho đồ thị $G=(V, E)$ và 2 đỉnh $s, t \in V$, đường đi từ s đến t trên G là dãy $s = x_0, x_1, \ldots, x_k = t$ trong đó $(x_i, x_{i+1}) \in E$, với $\forall i = 0, 1, \ldots, k\text{-}1$



Đường đi từ 1 đến 5:

- 1, 3, 4, 5
- 1, 6, 2, 5

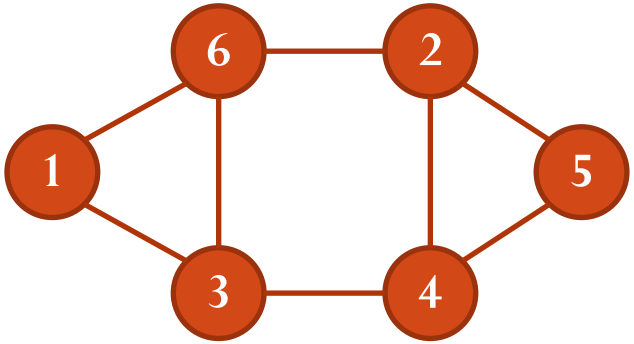


Đường đi từ 1 đến 5:

- 1, 3, 4, 5
- 1, 6, 4, 5

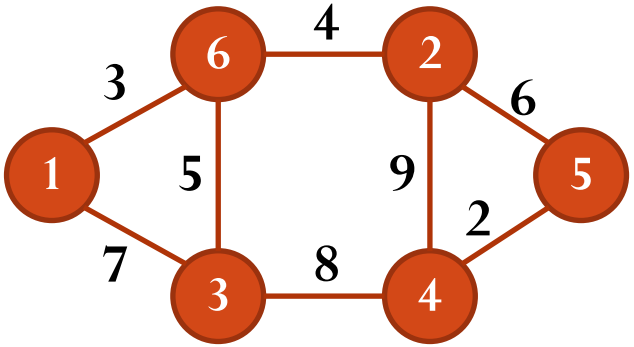
Biểu diễn đồ thị

- Ma trận kề



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

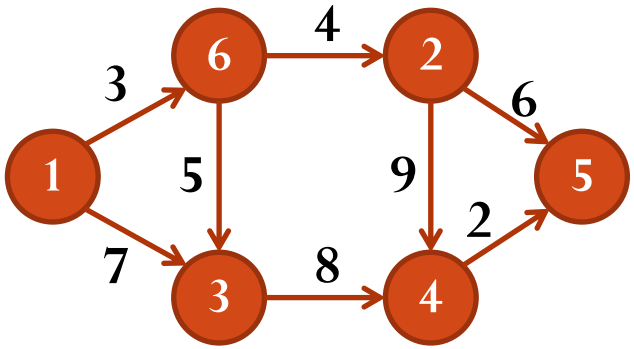
- Ma trận trọng số



	1	2	3	4	5	6
1	0	0	7	0	0	3
2	0	0	0	9	6	4
3	7	0	0	8	0	5
4	0	9	8	0	2	0
5	0	6	0	4	0	0
6	3	4	5	0	0	0

Biểu diễn đồ thị

- Danh sách kề
  - Với mỗi $v \in V$, $A(v)$ là tập các bộ (v, u, w) trong đó w là trọng số của cung (v, u)
  - $A(1) = \{(1, 6, 3), (1, 3, 7)\}$
  - $A(2) = \{(2, 4, 9), (2, 5, 6)\}$
  - $A(3) = \{(3, 4, 8)\}$
  - $A(4) = \{(4, 5, 2)\}$
  - $A(5) = \{\}$
  - $A(6) = \{(6, 3, 5), (6, 2, 4)\}$



Duyệt đồ thị

- Duyệt các đỉnh của đồ thị theo một thứ tự nào đó
- Các đỉnh được duyệt (thăm) đúng 1 lần
- Hai phương pháp cơ bản:
  - Duyệt theo chiều sâu
  - Duyệt theo chiều rộng

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

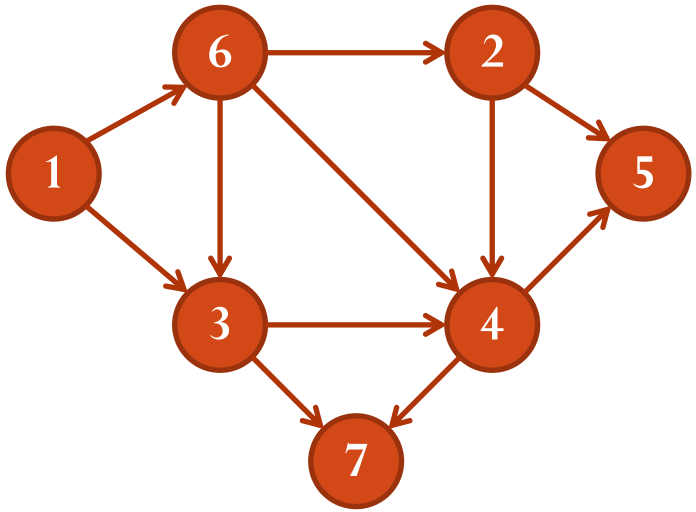
- DFS(*u*): duyệt theo chiều sâu bắt đầu từ đỉnh *u*
  - Nếu tồn tại đỉnh *v* trong danh sách kề của *u* chưa được thăm thì tiến hành thăm *v* và gọi DFS(*v*)
  - Nếu tất cả các đỉnh kề với *u* đã được thăm thì DFS quay trở lại đỉnh *x* mà từ đó thăm *u* để tiến hành thăm các đỉnh khác kề với *x* mà chưa được thăm. Lúc này đỉnh *u* được gọi là *đã duyệt xong*
- Cấu trúc dữ liệu: với mỗi đỉnh *v* của đồ thị
  - *p*(*v*): là đỉnh từ đó thăm *v*
  - *d*(*v*): thời điểm *v* được thăm nhưng chưa duyệt xong
  - *f*(*v*): thời điểm đỉnh *v* đã được duyệt xong
  - color(*v*)
    - WHITE: chưa thăm
    - GRAY: đã được thăm nhưng chưa duyệt xong
    - BACK: đã duyệt xong

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

```
DFS(u) {
  t = t + 1;
  d(u) = t;
  color(u) = GRAY;
  foreach (đỉnh v kề với u) {
    if(color(v) = WHITE) {
      p(v) = u;
      DFS(v);
    }
  }
  t = t + 1;
  f(u) = t;
  color(u) = BLACK;
}
```

```
DFS() {
  foreach (đỉnh u thuộc V) {
    color(u) = WHITE;
    p(u) = NIL;
  }
  foreach(đỉnh u thuộc V) {
    if(color(u) = WHITE) {
       DFS(u);
    }
  }
}
```

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d							
f							
p	-	-	-	-	-	-	-
color	W	W	W	W	W	W	W

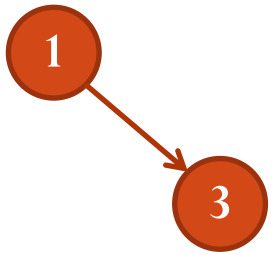
Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

DFS(1)

đỉnh	1	2	3	4	5	6	7
d	1						
f							
p	-	-	-	-	-	-	-
color	G	W	W	W	W	W	W

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

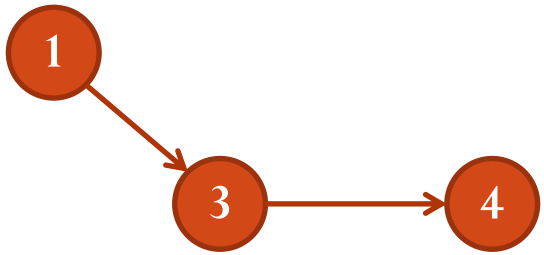
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2				
f							
p	-	-	1	-	-	-	-
color	G	W	G	W	W	W	W

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

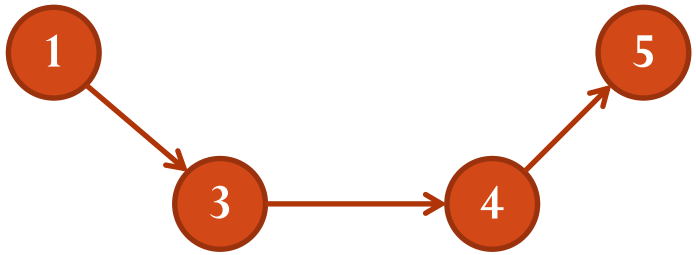
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3			
f							
p	-	-	1	3	-	-	-
color	G	W	G	G	W	W	W

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

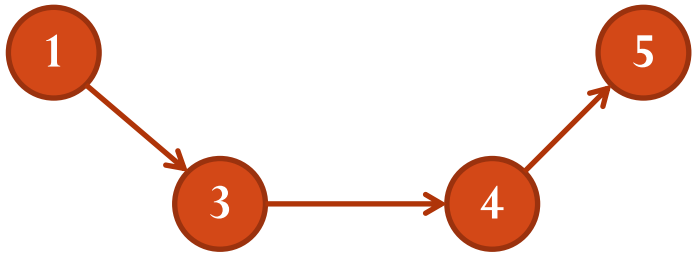
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3	4		
f							
p	-	-	1	3	4	-	-
color	G	W	G	G	G	W	W

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

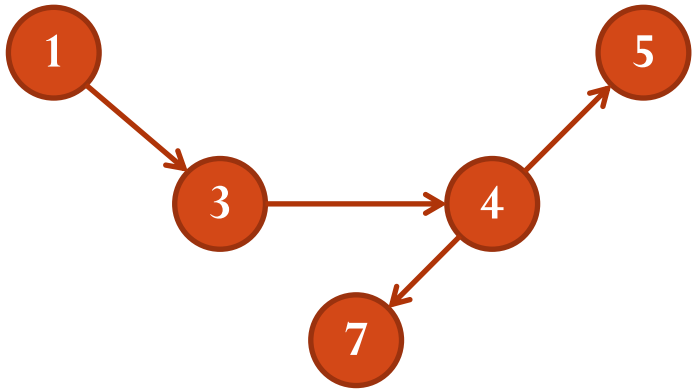
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3	4		
f					5		
p	-	-	1	3	4	-	-
color	G	W	G	G	B	W	W

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

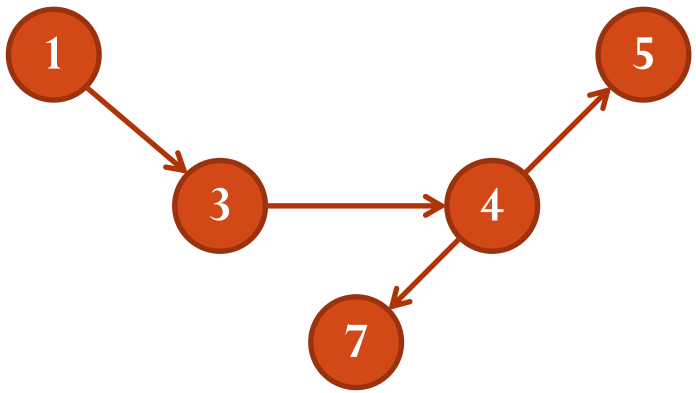
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3	4		6
f					5		
p	-	-	1	3	4	-	4
color	G	W	G	G	B	W	G

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

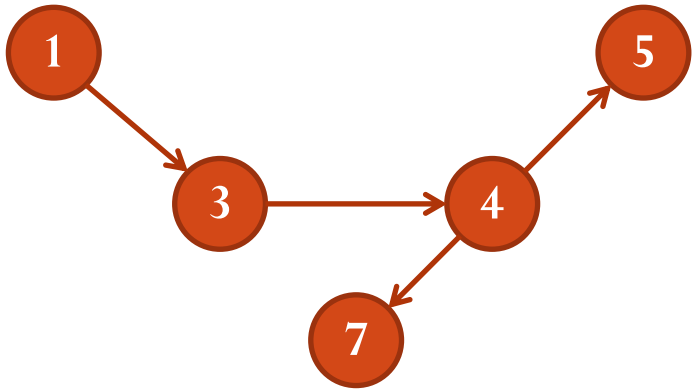
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3	4		6
f					5		7
p	-	-	1	3	4	-	4
color	G	W	G	G	B	W	B

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

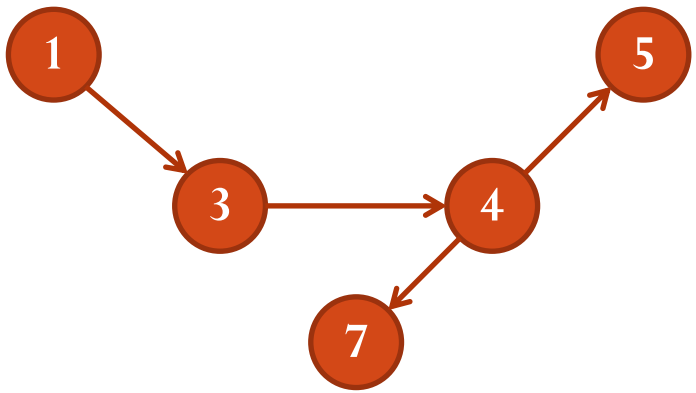
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3	4		6
f				8	5		7
p	-	-	1	3	4	-	4
color	G	W	G	B	B	W	B

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

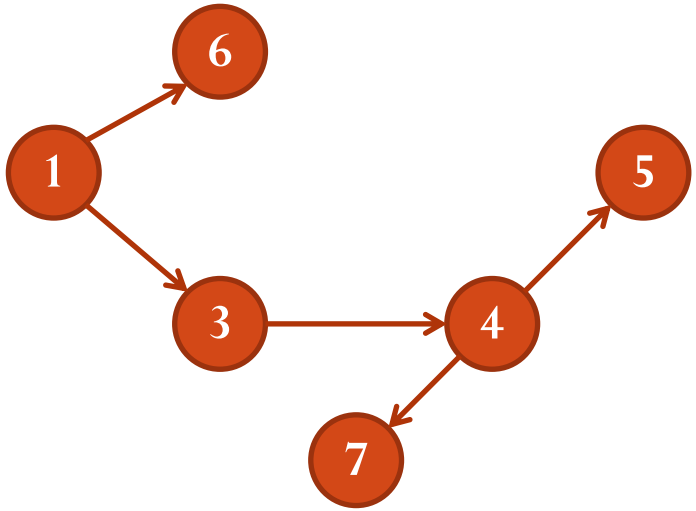
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3	4		6
f			9	8	5		7
p	-	-	1	3	4	-	4
color	G	W	B	B	B	W	B

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

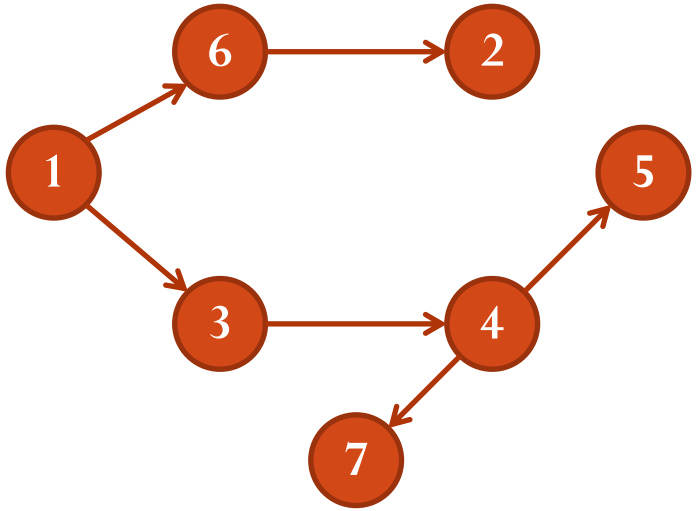
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1		2	3	4	10	6
f			9	8	5		7
p	-	-	1	3	4	1	4
color	G	W	B	B	B	G	B

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

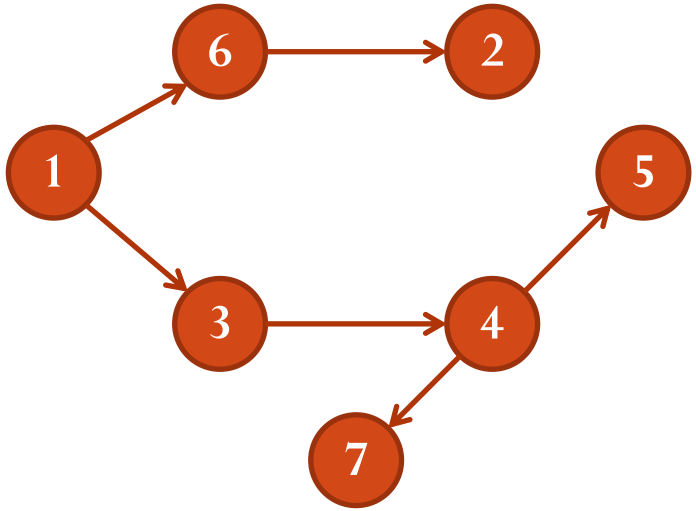
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1	11	2	3	4	10	6
f			9	8	5		7
p	-	6	1	3	4	1	4
color	G	G	B	B	B	G	B

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

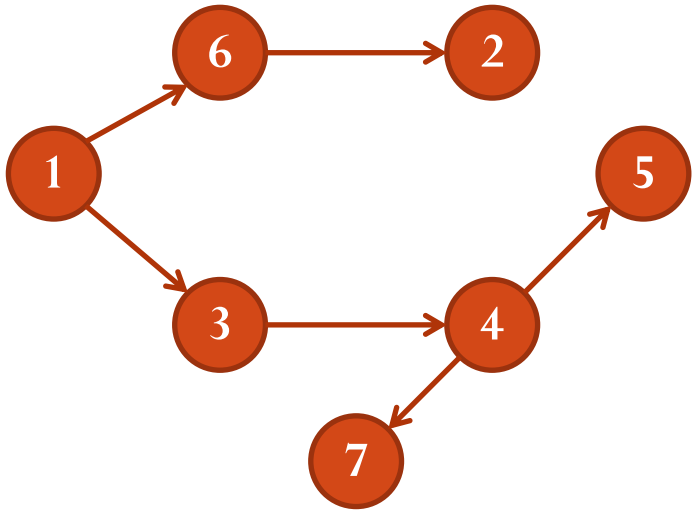
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1	11	2	3	4	10	6
f		12	9	8	5		7
p	-	6	1	3	4	1	4
color	G	B	B	B	B	G	B

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

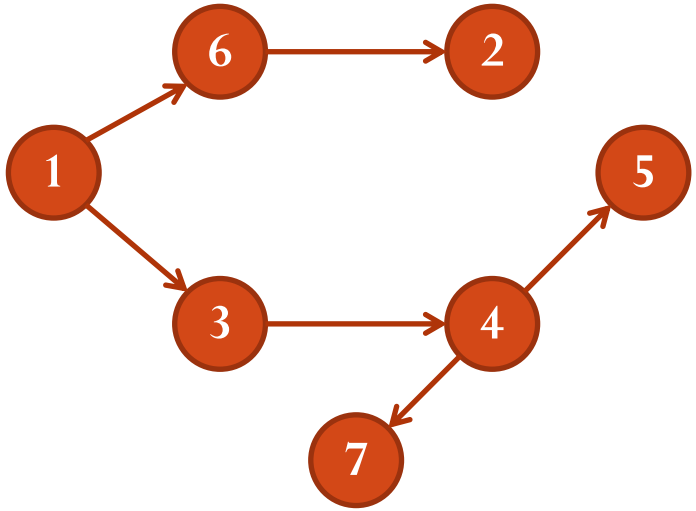
DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1	11	2	3	4	10	6
f		12	9	8	5	13	7
p	-	6	1	3	4	1	4
color	G	B	B	B	B	B	B

Duyệt đồ thị theo chiều sâu
Depth First Search - DFS

DFS(1)



đỉnh	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d	1	11	2	3	4	10	6
f	14	12	9	8	5	13	7
p	-	6	1	3	4	1	4
color	B	B	B	B	B	B	B

Duyệt đồ thị theo chiều rộng
Breadth First Search - BFS

- BFS(*u*): duyệt theo chiều rộng xuất phát từ đỉnh *u*
  - Thăm các đỉnh *u*
  - Thăm các đỉnh kề với u mà chưa được thăm (gọi là các đỉnh mức 1)
  - Thăm các đỉnh kề với các đỉnh mức 1 mà chưa được thăm (gọi là các đỉnh mức 2)
  - Thăm các đỉnh kề với các đỉnh mức 2 mà chưa được thăm (gọi là các đỉnh mức 3)
  - . . .
- Sử dụng cấu trúc hàng đợi (queue) để cài đặt

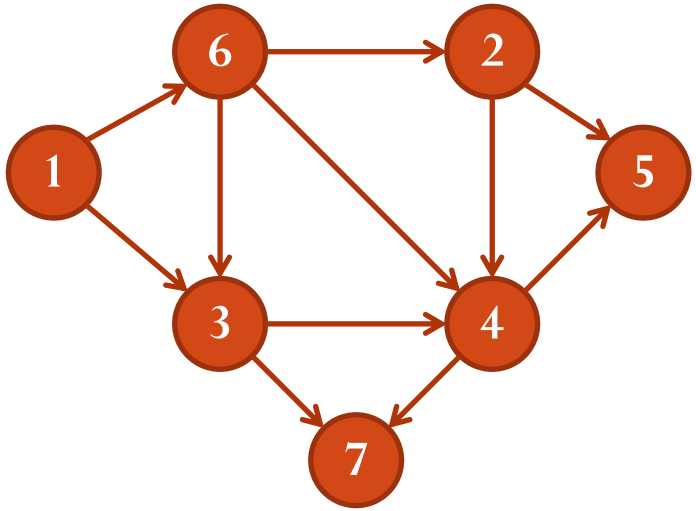
Duyệt đồ thị theo chiều rộng
Breadth First Search - BFS

```
BFS(u) {
  d(u) = 0;
  khởi tạo hàng đợi Q;
  enqueue(Q,u);
  color(u) = GRAY;
  while(Q khác rỗng) {
    v = dequeue(Q);
    foreach(x kề với v) {
      if(color(x) = WHITE){
        d(x) = d(v) + 1;
        color(x) = GRAY;
        enqueue(Q,x);
      }
    }
  }
}
```

```
BFS() {
  foreach (đỉnh u thuộc V) {
    color(u) = WHITE;
    p(u) = NIL;
  }
  foreach(đỉnh u thuộc V) {
    if(color(u) = WHITE) {
       BFS(u);
    }
  }
}
```

Duyệt đồ thị theo chiều rộng
Breadth First Search - BFS

BFS(1)

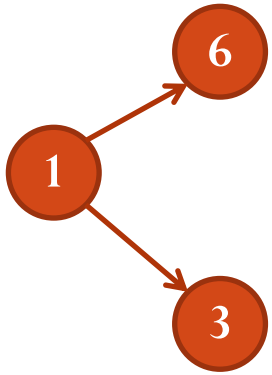


Duyệt đồ thị theo chiều rộng
Breadth First Search - BFS

BFS(1)

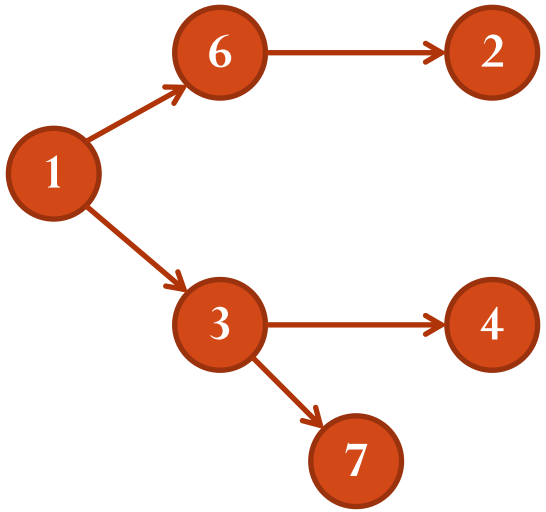
Duyệt đồ thị theo chiều rộng
Breadth First Search - BFS

BFS(1)



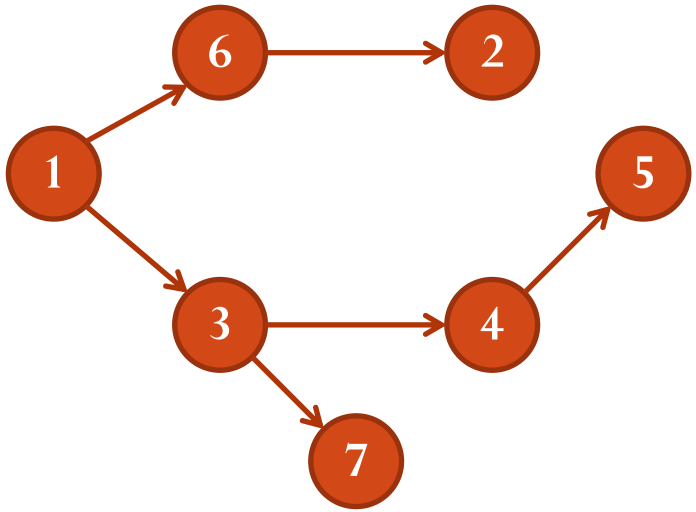
Duyệt đồ thị theo chiều rộng
Breadth First Search - BFS

BFS(1)



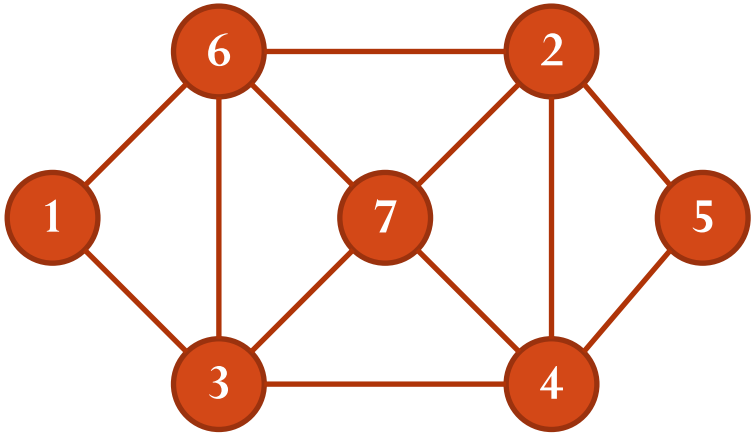
Duyệt đồ thị theo chiều rộng
Breadth First Search - BFS

BFS(1)



Đồ thị Euler và đồ thị Hamilton

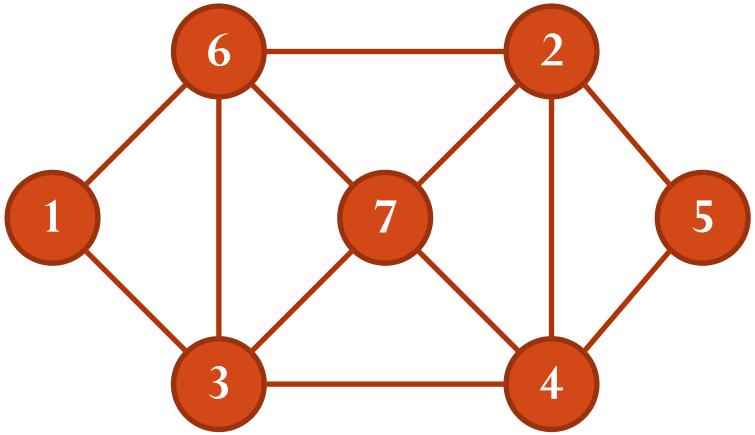
- Cho đồ thị vô hướng G = (V, E)
  - Chu trình Euler trên G là chu trình đi qua tất cả các cạnh, mỗi cạnh đúng 1 lần.
  - Chu trình Hamilton là chu trình đi qua tất cả các đỉnh, mỗi đỉnh đúng 1 lần (trừ đỉnh xuất phát)
- Đồ thị chứa chu trình Euler được gọi là đồ thị Euler
- Đồ thị chứa chu trình Hamilton được gọi là đồ thị Hamilton



- Chu trình Euler: 1, 6, 3, 7, 6, 2, 5, 4, 2, 7, 4, 3, 1
- Chu trình Hamilton: 1, 6, 2, 5, 4, 7, 3, 1

Đồ thị Euler và đồ thị Hamilton

- Định lí: Đồ thị vô hướng là đồ thị Euler khi và chỉ khi nó là đồ thị liên thông trong đó các đỉnh có bậc là số chẵn
- Định lí: Đồ thị vô hướng *n* đỉnh trong đó mỗi đỉnh có bậc lớn hơn hoặc bằng *n*/2 là đồ thị Hamilton



Đồ thị Euler và đồ thị Hamilton

- Thuật toán tìm chu trình Euler, sử dụng ngăn xếp
- Đồ thị đầu vào $G = (V, E)$ trong đó $A(x)$ là tập các đỉnh kề với đỉnh x

```
euler(G = (V, E)) {
  khởi tạo stack S, CE;
  chọn v là một đỉnh bất kỳ thuộc V;
  push(S,v);
  while(S khác rỗng) {
    x là nút ở đỉnh của S;
    if(A(x) khác rỗng) {
      chọn y là một đỉnh bất kỳ ∈ A(x);
      push(S,y);
      loại bỏ cạnh (x,y) khỏi G;
    }else{
      x = pop(S); push(CE,x);
    }
  }
  thứ tự các đỉnh trong CE tạo chu trình euler
}
```

Đồ thị Euler và đồ thị Hamilton

- Thuật toán tìm chu trình Hamilton, sử dụng duyệt đệ quy quay lui
- Đồ thị đầu vào $G = (V, E)$ trong đó
  - $V = \{1, 2, \ldots, n\}$
  - $A(v)$ là tập các đỉnh kề với đỉnh v
- Mô hình hoá: mảng $x[1..n]$
- Mảng đánh dấu mark[v] = true nếu đỉnh v đã được dùng và mark[v] = false, ngược lại

```
TRY(k) {
  for(v ∈ A(x[k-1]) {
    if(not mark[v]) {
      x[k] = v;
      mark[v] = true;
      if(k == n) {
        if(v ∈ A(x[1]) {
          ghi nhận chu trình Hamilton trong x;
        }
      }else{
        TRY(k+1);
      }
      mark[v] = false;
    }
  }
}
```

Cây khung nhỏ nhất trên đồ thị

- Cho đồ thị vô hướng liên thông $G = (V, E, w)$.
  - Mỗi cạnh $(u, v) \in E$ có $w(u, v)$ là trọng số
  - Nếu $(u, v) \in E$ thì $w(u, v) = \infty$
- Đồ thị là đồ thị vô hướng, liên thông và không có chu trình chứa tất cả các đỉnh của G.
- Cây $T = (V, F)$ trong đó $F \subseteq E$ gọi là một cây khung của G
  - Trọng số $w(T) = \sum_{e \in F} w(e)$
- Tìm cây khung của G có trọng số nhỏ nhất
- Ứng dụng
  - Thiết kế mạng truyền thông

Cây khung nhỏ nhất trên đồ thị

- Thuật toán PRIM tìm cây khung nhỏ nhất
- Ý tưởng chính
  - Xây dựng cây bằng cách kết nạp lần lượt các cạnh vào cây T theo chiến lược tham lam
  - Ban đầu, tập đỉnh V_T của cây chỉ gồm 1 đỉnh được chọn ngẫu nhiên
  - Mỗi bước, chọn đỉnh (chưa thuộc V_T) gần T nhất để kết nạp đỉnh và cạnh vào cây T
- Cấu trúc dữ liệu
  - Với mỗi đỉnh $v \notin V_T$
    - $d(v)$ là khoảng cách từ v đến V_T:

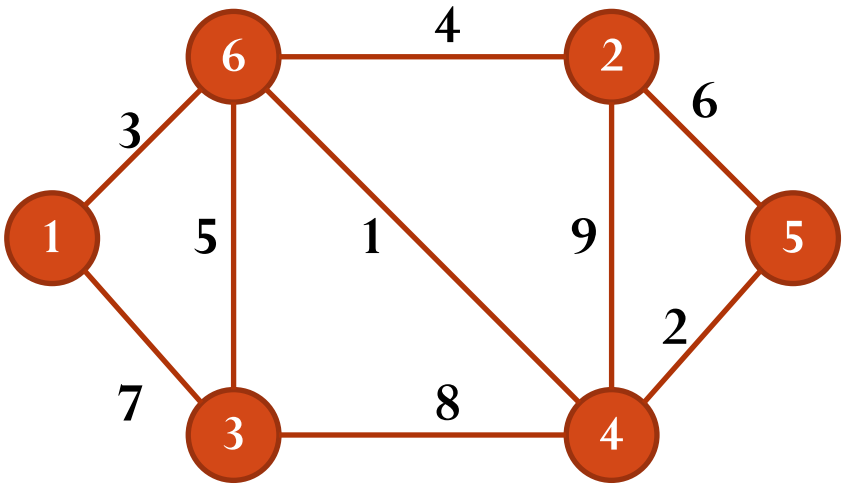
$$d(v) = \min\{w(v, u) \mid u \in V_T, (u, v) \in E\}$$

    - *near(v)*: đỉnh $\in V_T$ có $w(v, \text{near}(v)) = d(v)$;

Cây khung nhỏ nhất trên đồ thị

```
PRIM(G = (V,E,w)) {
  chọn s là một đỉnh nào đó của V;
  for(v ∈ V) {
     d(v) = w(s,v); near(v) = s;
  }
  E_T = {}; V_T = {s};
  while(|V_T| ≠ |V|) {
    v = chọn đỉnh ∈ V \ V_T có d(v) nhỏ nhất;
    V_T = V_T∪{v}; E_T = E_T ∪ {(v, near(v))};
    for(x ∈ V \ V_T) {
      if(d(x) > w(x,v) {
        d(x) = w(x,v);
        near(x) = v;
      }
    }
  }
  return (V_T, E_T);
}
```

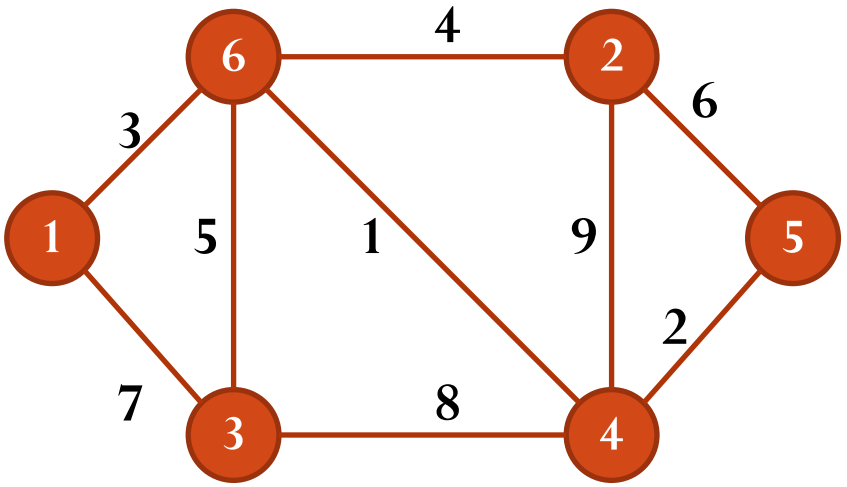
Cây khung nhỏ nhất trên đồ thị



- Mỗi ô ứng với đỉnh v của bảng chứa nhãn $(d(v), \text{near}(v))$
- Đỉnh xuất phát $s = 1$

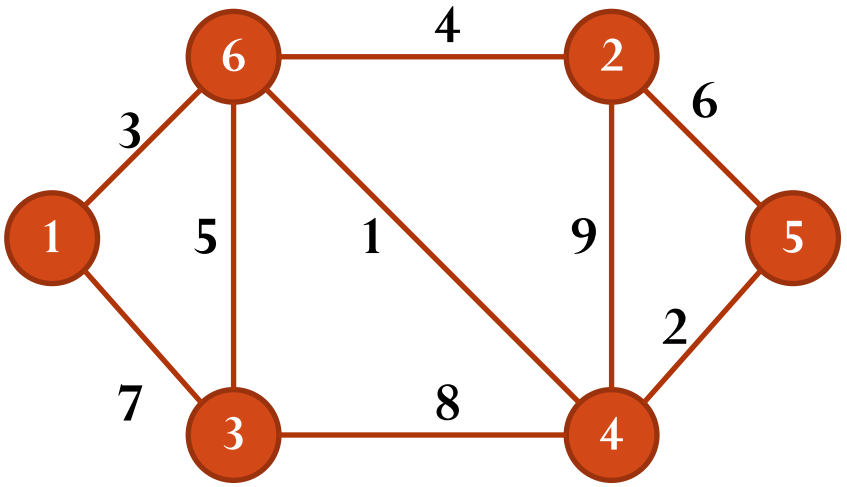
	1	2	3	4	5	6	E_T
Khởi tạo	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1)**	
Bước 1	-						
Bước 2	-						
Bước 3	-						
Bước 4	-						
Bước 5	-						

Cây khung nhỏ nhất trên đồ thị



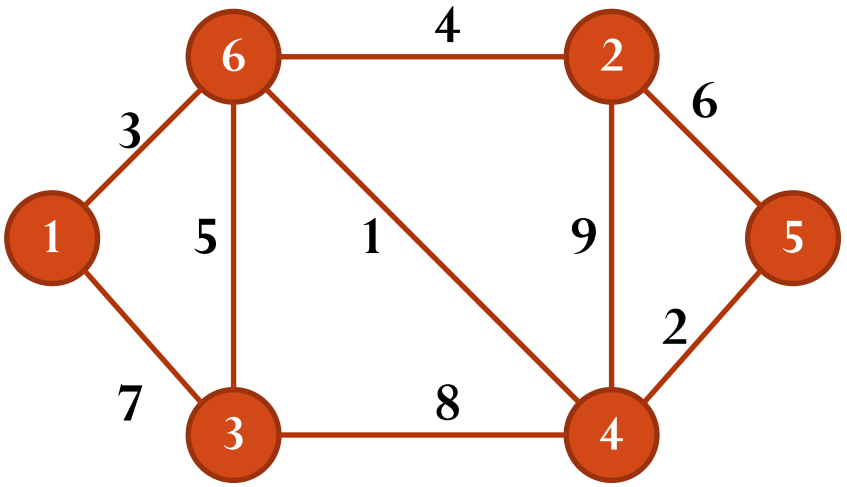
	1	2	3	4	5	6	E_T
Khởi tạo	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***	**(1,6)**
Bước 1	-	**(4,6)**	**(5,6)**	**(1,6)**	**(∞,1)**	-	
Bước 2	-					-	
Bước 3	-					-	
Bước 4	-					-	
Bước 5	-					-	

Cây khung nhỏ nhất trên đồ thị



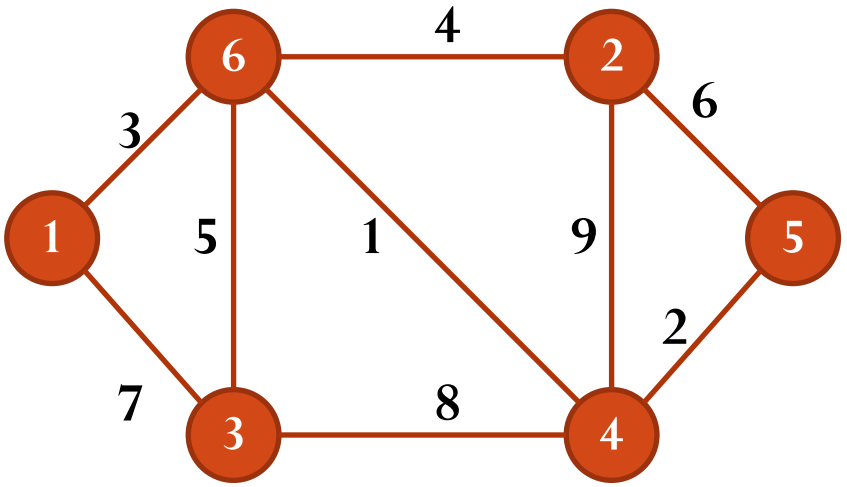
	1	2	3	4	5	6	E_T
Khởi tạo	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***	**(1,6)**
Bước 1	**-**	**(4,6)**	**(5,6)**	**(1,6) ***	**(∞,1)**	**-**	**(1,6), (4,6)**
Bước 2	**-**	**(4,6)**	**(5,6)**	**-**	**(2,4)**	**-**	
Bước 3	**-**			**-**		**-**	
Bước 4	**-**			**-**		**-**	
Bước 5	**-**			**-**		**-**	

Cây khung nhỏ nhất trên đồ thị



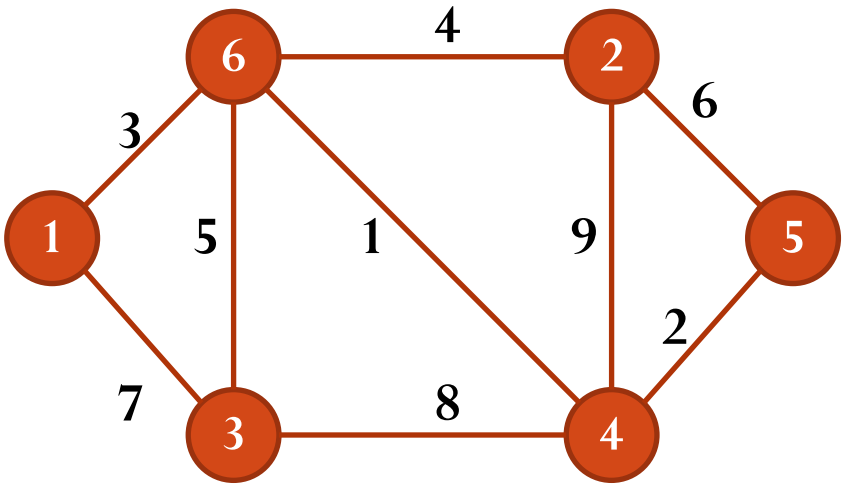
	1	2	3	4	5	6	E_T
Khởi tạo	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1) *	(1,6)
Bước 1	-	(4,6)	(5,6)	(1,6) *	(∞,1)	-	(1,6),(4,6)
Bước 2	-	(4,6)	(5,6)	-	(2,4) *	-	(1,6),(4,6),(4,5)
Bước 3	-	(4,6)	(5,6)	-	-	-	
Bước 4	-			-	-	-	
Bước 5	-			-	-	-	

Cây khung nhỏ nhất trên đồ thị



	1	2	3	4	5	6	E_T
Khởi tạo	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1) *	(1,6)
Bước 1	-	(4,6)	(5,6)	(1,6) *	(∞,1)	-	(1,6),(4,6)
Bước 2	-	(4,6)	(5,6)	-	(2,4) *	-	(1,6),(4,6),(4,5)
Bước 3	-	(4,6) *	(5,6)	-	-	-	(1,6),(4,6),(4,5),(2,6)
Bước 4	-	-	(5,6)	-	-	-	
Bước 5	-	-		-	-	-	

Cây khung nhỏ nhất trên đồ thị



	1	2	3	4	5	6	E_T
Khởi tạo	(0,1)	(∞,1)	(7, 1)	(∞, 1)	(∞, 1)	(3,1)	
Bước 1	-	(4,6)	(5,6)	(1,6) *	(∞,1)	-	(1,6)
Bước 2	-	(4,6)	(5,6)	-	(2,4) *	-	(1,6), (4,6)
Bước 3	-	(4,6) *	(5,6)	-	-	-	(1,6), (4,6), (4,5)
Bước 4	-	-	(5,6) *	-	-	-	(1,6), (4,6), (4,5), (2,6)
Bước 5	-	-	-	-	-	-	(1,6), (4,6), (4,5), (2,6), (3,6)

Cây khung nhỏ nhất trên đồ thị
Cài đặt thuật toán PRIM

```
#include <stdio.h>
#include <set>
#include <map>
#include <stack>
using namespace std;
struct Arc{
          int nod;
          int w;
};
set<int> V;// set of nodes
map<int, set<Arc*> > A;// A[v] is the set of adjacent arcs of v

// data structure for prim
map<int, int> d;
map<int, int> near;
set<int> S;
```

Cây khung nhỏ nhất trên đồ thị
Cài đặt thuật toán PRIM

```
int findMin(){
  // find a node v of NonFixed having minimum d[v]
  int min = 1000000;
  int v_min = -1;
  for(set<int>::iterator p = S.begin(); p != S.end(); p++){
    int v = *p;
    if(d[v] < min){
      min = d[v];
      v_min = v;
    }
  }
  return v_min;
}
```

Cây khung nhỏ nhất trên đồ thị
Cài đặt thuật toán PRIM

```
void prim(int s){
  // initialization
  for(set<int>::iterator pi = V.begin(); pi != V.end(); pi++){
    int x = *pi;  d[x] = 100000000;
  }
  d[s] = 0;
  for(set<Arc*>::iterator ps = A[s].begin(); ps != A[s].end(); ps++){
    Arc* a = *ps;
    int x = a->nod;  int w = a->w;
    d[x] = w; near[x] = s;
  }
  for(set<int>::iterator pi = V.begin(); pi != V.end(); pi++){
    int v = *pi;
    if(v != s)
      S.insert(v);
  }
  . . .
```

Cây khung nhỏ nhất trên đồ thị
Cài đặt thuật toán PRIM

```
  //LOOP
  while(S.size() > 0){
    int v = findMin();
    printf("select edge (%d,%d) with d = %d\n",v,near[v],d[v]);
    S.erase(v);
    // update d of non-fixed nodes
    for(set<Arc*>::iterator pv = A[v].begin(); pv != A[v].end(); pv++){
      Arc* a = *pv;
      int x = a->nod;
      int w = a->w;
      if(d[x] > w){
        d[x] = w;
        near[x] = v;
      }
    }
  }
}
```

Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra

- Cho đồ thị vô hướng liên thông $G = (V, E, w)$.
  - Mỗi cạnh $(u, v) \in E$ có $w(u, v)$ là trọng số không âm
  - Nếu $(u, v) \notin E$ thì $w(u, v) = \infty$
- Cho s là một đỉnh thuộc V, tìm đường đi ngắn nhất từ s đến tất cả các đỉnh còn lại

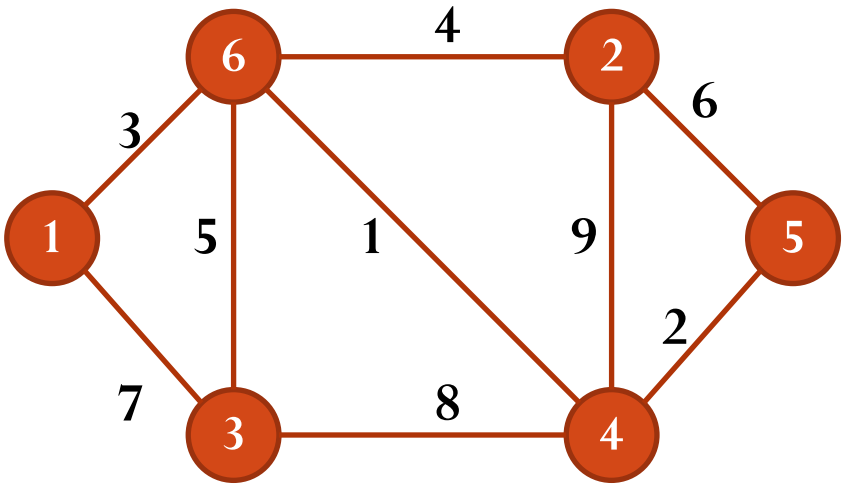
Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra

- Ý tưởng thuật toán Dijkstra:
  - Với mỗi đỉnh $v \in V$, duy trì:
    - $\mathscr{P}(v)$ là đường đi cận trên của đường đi ngắn nhất từ s đến v
    - $d(v)$: trọng số của $\mathscr{P}(v)$
    - $p(v)$: đỉnh trước đỉnh v trên $\mathscr{P}(v)$
  - Khởi tạo
    - $\mathscr{P}(v) = \langle s, v \rangle$, $d(v) = w(s,v)$, $p(v) = s$
  - Làm tốt cận trên
    - Mỗi khi phát hiện có đỉnh u sao cho $d(v) > d(u) + w(u, v)$ thì cập nhật
      - $d(v) = d(u) + w(u, v)$
      - $p(v) = u$

Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra

```
Dijkstra(G = (V,E,w)) {
  for(v ∈ V) {
     d(v) = w(s,v); p(v) = s;
  }
  S = V \ {s};
  while(S ≠ {}) {
    u = chọn đỉnh ∈ S có d(u) nhỏ nhất;
    S = S \ {u};
    for(v ∈ S) {
      if(d(v) > d(u) + w(u,v) {
        d(v) = d(u) + w(u,v);
        p(v) = u;
      }
    }
  }
}
```

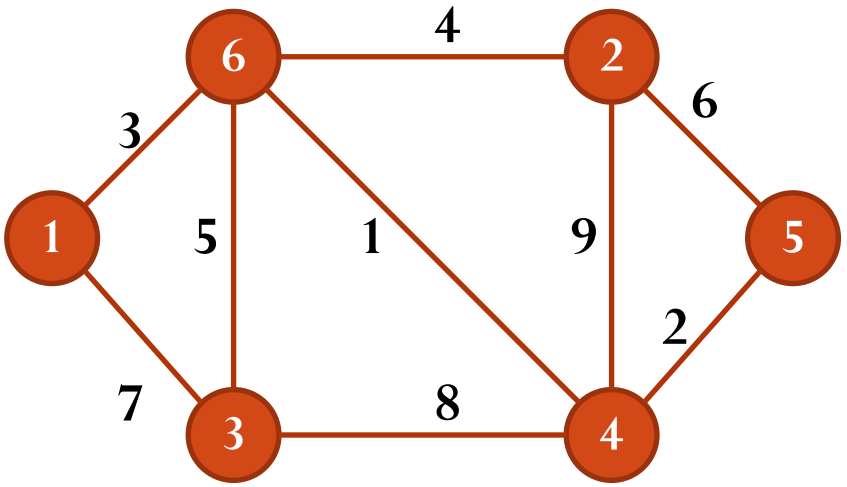
Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra



- Mỗi ô ứng với đỉnh v của bảng chứa nhãn $(d(v), p(v))$
- Đỉnh xuất phát $s = 1$

	1	2	3	4	5	6
Khởi tạo	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1)**
Bước 1	-					
Bước 2	-					
Bước 3	-					
Bước 4	-					
Bước 5	-					

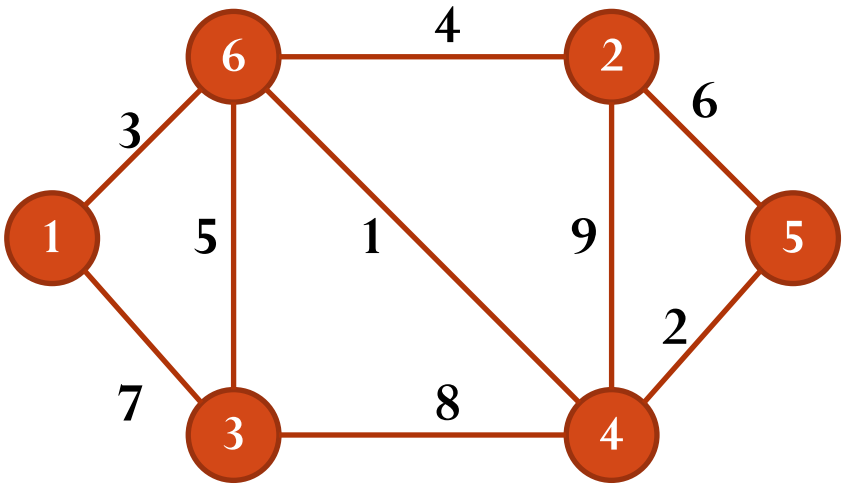
Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra



- Mỗi ô ứng với đỉnh v của bảng chứa nhãn $(d(v), p(v))$
- Đỉnh xuất phát $s = 1$

	1	2	3	4	5	6
Khởi tạo	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***
Bước 1	-	**(7,6)**	**(7,1)**	**(4,6)**	**(∞,1)**	-
Bước 2	-					-
Bước 3	-					-
Bước 4	-					-
Bước 5	-					-

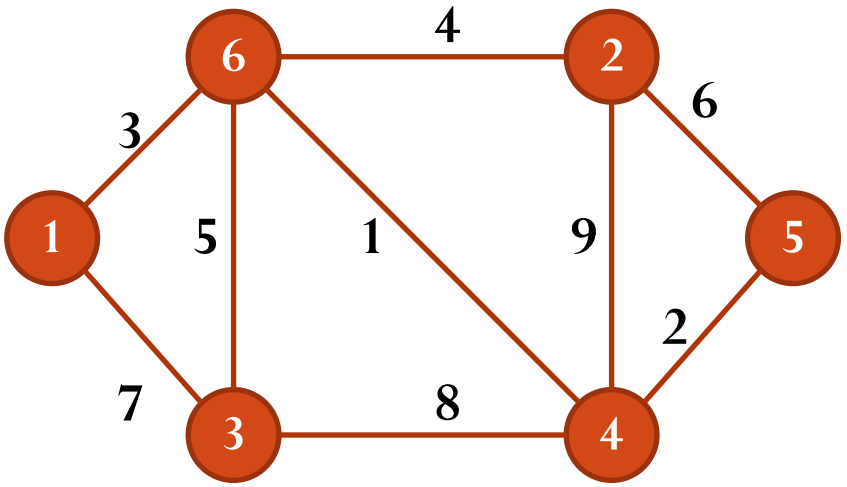
Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra



- Mỗi ô ứng với đỉnh v của bảng chứa nhãn $(d(v), p(v))$
- Đỉnh xuất phát $s = 1$

	1	2	3	4	5	6
Khởi tạo	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***
Bước 1	-	**(7,6)**	**(7,1)**	**(4,6) ***	**(∞,1)**	-
Bước 2	-	**(7,6)**	**(7,1)**	-	**(6, 4)**	-
Bước 3	-			-		-
Bước 4	-			-		-
Bước 5	-			-		-

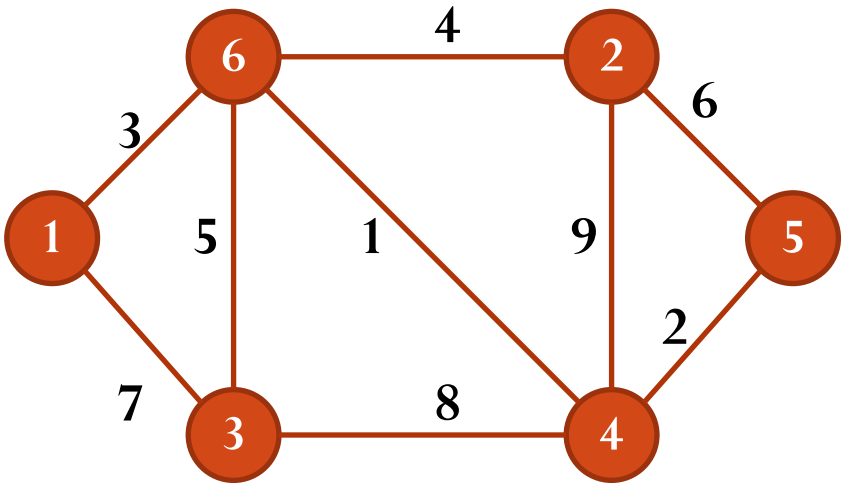
Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra



- Mỗi ô ứng với đỉnh v của bảng chứa nhãn $(d(v), p(v))$
- Đỉnh xuất phát $s = 1$

	1	2	3	4	5	6
Khởi tạo	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***
Bước 1	-	**(7,6)**	**(7,1)**	**(4,6) ***	**(∞,1)**	-
Bước 2	-	**(7,6)**	**(7,1)**	-	**(6, 4) ***	-
Bước 3	-	**(7,6)**	**(7,1)**	-	-	-
Bước 4	-			-	-	-
Bước 5	-			-	-	-

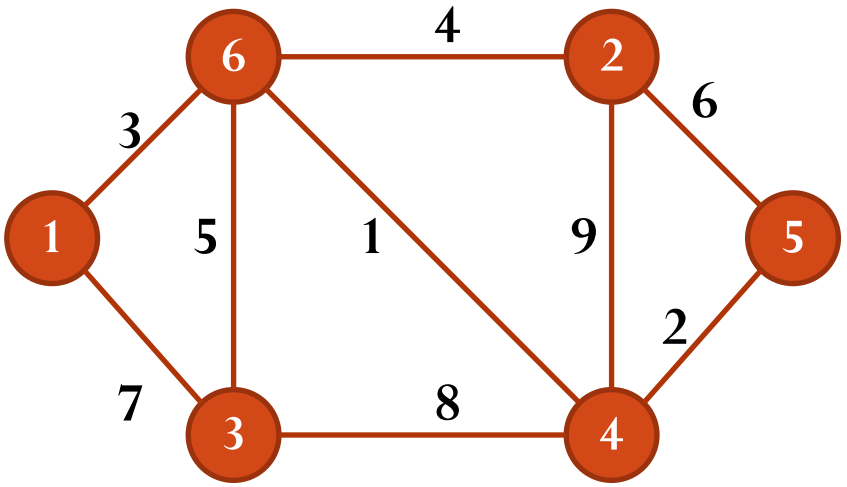
Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra



- Mỗi ô ứng với đỉnh v của bảng chứa nhãn $(d(v), p(v))$
- Đỉnh xuất phát $s = 1$

	1	2	3	4	5	6
Khởi tạo	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***
Bước 1	-	**(7,6)**	**(7,1)**	**(4,6) ***	**(∞,1)**	-
Bước 2	-	**(7,6)**	**(7,1)**	-	**(6, 4) ***	-
Bước 3	-	**(7,6)**	**(7,1) ***	-	-	-
Bước 4	-	**(7,6)**	-	-	-	-
Bước 5	-		-	-	-	-

Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra



- Mỗi ô ứng với đỉnh v của bảng chứa nhãn $(d(v), p(v))$
- Đỉnh xuất phát $s = 1$

	1	2	3	4	5	6
Khởi tạo	**(0,1)**	**(∞,1)**	**(7, 1)**	**(∞, 1)**	**(∞, 1)**	**(3,1) ***
Bước 1	-	**(7,6)**	**(7,1)**	**(4,6) ***	**(∞,1)**	-
Bước 2	-	**(7,6)**	**(7,1)**	-	**(6, 4) ***	-
Bước 3	-	**(7,6)**	**(7,1) ***	-	-	-
Bước 4	-	**(7,6) ***	-	-	-	-
Bước 5	-	-	-	-	-	-

Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra

```
#include <stdio.h>
#include <set>
#include <map>
#include <stack>
using namespace std;
struct Arc{
  int nod;
  int w;
};

set<int> V;// set of nodes
map<int, set<Arc*> > A;// A[v] is the set of adjacent arcs of v

// data strcuture for dijkstra
map<int, int> d;
map<int, int> p;
set<int> S;
```

Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra

```
int findMin(){
  // find a node v of NonFixed having minimum d[v]
  int min = 1000000;
  int v_min = -1;
  for(set<int>::iterator p = S.begin(); p != S.end(); p++){
    int v = *p;
    if(d[v] < min){
      min = d[v];
      v_min = v;
    }
  }
  return v_min;
}
```

Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra

```
void dijkstra(int s){
  // initialization
  for(set<int>::iterator pi = V.begin(); pi != V.end(); pi++){
    int x = *pi;  d[x] = 100000000;
  }
  d[s] = 0;
  for(set<Arc*>::iterator ps = A[s].begin(); ps != A[s].end(); ps++){
    Arc* a = *ps;
    int x = a->nod;  int w = a->w;
    d[x] = w; p[x] = s;
  }
  for(set<int>::iterator pi = V.begin(); pi != V.end(); pi++){
    int v = *pi;
    if(v != s)
      S.insert(v);
  }
  . . .
```

Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra

```
  //LOOP
  while(S.size() > 0){
    int v = findMin();
    S.erase(v);
    // update label of nodes in S
    for(set<Arc*>::iterator pv = A[v].begin(); pv != A[v].end(); pv++){
      Arc* a = *pv;
      int x = a->nod;
      int w = a->w;
      if(d[x] > d[v] + w){
        d[x] = d[v] + w;
        p[x] = v;
      }
    }
  }
}
```

Đường đi ngắn nhất trên đồ thị
Thuật toán Dijkstra

```
void printPath(int s, int v){
  stack<int> S;
  int x = v;
  while(x != s){
    S.push(x);
    x = p[x];
  }
  printf("%d ",s);
  while(!S.empty()){
    int  x = S.top(); S.pop();
    printf("%d ",x);
  }
  printf("\n");
}
```