Mục lục

LỜI GIỚI THIỆU

Đối với lĩnh vực công nghệ thông tin, các kiến thức về thuật giải và cấu trúc dữ liệu đặc biệt quan trọng. Chính vì vậy, bài giảng Cấu trúc dữ liệu và Giải thuật được biên soạn nhằm mục đích giảng dạy và học tập cho sinh viên khoa Công nghệ thông tin. Mục tiêu của bài giảng hệ thống lại những nội dung chủ yếu về cơ sở lập trình và kiến thức cơ bản của học phần này.

- Số tín chỉ: 4 (3 lý thuyết, 1 thực hành)
- Nội dung lý thuyết gồm 6 chương:
  - Chương 0. Ôn tập kiến thức về kỹ thuật lập trình (3 tiết)
  - Chương 1. Thuật toán và đánh giá độ phức tạp (3 tiết)
  - Chương 2. Các thuật toán tìm kiếm và Sắp xếp (12 tiết)
  - Chương 3. Danh sách liên kết (9 tiết)
  - Chương 4. Ngăn xếp và Hàng đợi (6 tiết)
  - Chương 5. Cây, cây nhị phân, cây cân bằng (9 tiết)
  - Chương 6. Bảng băm, hàm băm (3 tiết)
- Tài liệu tham khảo: [1], [2], [3], [4].

Bài 1

ÔN TẬP KỸ THUẬT LẬP TRÌNH (3 tiết)

1.1 Cách sử dụng biến con trỏ

Biến

Một biến khi được khai báo gồm ba thuộc tính cơ bản:

- Tên biến
- Giá trị của biến
- Địa chỉ của biến trong vùng nhớ

Vùng nhớ của một biến được cấp phát theo hai cách:

- Cấp phát tĩnh: đã xác định kích thước vùng nhớ trước khi biên dịch chương trình, được lưu trữ trong vùng nhớ Stack
  - Biến toàn cục *(global)* và biến tĩnh *(static)*: chạy chương trình.
  - Biến cục bộ *(local)*: gọi hàm.
- Cấp phát động: chỉ xác định vùng nhớ khi biên dịch chương trình, được lưu trữ trong vùng nhớ Heap
  - Sử dụng từ khóa `new` để tạo một vùng nhớ dữ liệu trong Heap.
  - Trong ngôn ngữ C#, gán giá trị biến là `null` để thu hồi vùng nhớ được cấp phát động
  - Vùng nhớ cấp phát động trong Heap được quản lý thông qua biến con trỏ.
- Biến con trỏ *(pointer)* là biến lưu địa chỉ của một kiểu dữ liệu nào đó.
- Cú pháp

```
<Kiểu dữ liệu> *<Tên con trỏ>;
```

```
1 int *p1;
2 float *p2;
3 SinhVien *sv;
```

- Chú ý, để sử dụng được biến con trỏ phải cấu hình theo các bước sau: mở Properties/Build của dự án –> chọn mục Allow unsafe code –> mã nguồn phải thêm từ khóa `unsafe` trước khi khai báo biến con trỏ.

```
using System;
namespace DSA2021
{
    unsafe class Program // unsafe
    {
        static void Main(string[] args)
        {
            int a = 10;
            int *b = &a; // pointer
            *b += 10;

            Console.WriteLine("value of a = {0}", a);
            // Console.WriteLine("address of a = {0}", b); //
                error
            Console.WriteLine("address of a = {0}", (uint)b);
        }
    }
}
```

1.2 Truyền tham số cho hàm

Tham số của một hàm có thể truyền theo một trong hai cách: truyền tham trị và truyền tham chiếu.

Truyền tham trị

- Giá trị của biến không thay đổi sau khi hàm thực hiện xong.

Truyền tham chiếu

- Giá trị của biến thay đổi sau khi hàm thực hiện xong, thường được dùng để hoán vị hai phần tử trong mảng.
- Hai loại truyền tham chiếu:
  - ref: không cần khởi tạo giá trị cho tham số
  - out: phải khởi tạo giá trị cho tham số

```
using System;
namespace Example
{
    class Program
    {
        static void Main(string[] args)
        {
            int a, b;
            Console.Write("a = ");
            a = Int32.Parse(Console.ReadLine());
```

```
            Console.Write("b = ");
            b = Int32.Parse(Console.ReadLine());
            // ...
            Console.WriteLine("a = {0}, b = {1}", a, b);
        }
    }
}
```

```
static void HoanVi1(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
static void HoanVi2(ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

1.3 Mảng một chiều

- Mảng là một dãy các phần tử được lưu trữ liên tiếp nhau trong vùng nhớ, cần khai báo số lượng phần tử trước khi sử dụng.
- Cú pháp:
  - Khai báo

    `type[] array;`
  - Truy xuất phần tử

    `array[i]`

Thao tác duyệt mảng

```
    int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
    int sum = 0;
    for (int i = 0; i < numbers.Length; i++)
    {
        sum += numbers[i];
    }
```

```
    foreach (int num in numbers)
    {
        sum += num;
    }
```

Các bài toán liên quan đến dữ liệu mảng:

- In mảng

- Tìm phần tử trong mảng
- Sắp xếp mảng theo thứ tự tăng hay giảm

1.4 Struct & Class

Kiểu cấu trúc (struct)

- Đây là một kiểu dữ liệu có cấu trúc và được kết hợp từ nhiều kiểu dữ liệu nguyên thủy khác nhau.
- Cú pháp:

```
struct StructName
{
    type fieldName1;
    type fieldName2;
}
```

Ví dụ 1.4.1. *Cấu trúc SinhVien quản lý các thông tin sau: Mssv, HoTen, NgaySinh, GioiTinh, QueQuan.*

```
1  struct SinhVien
2  {
3      public string MSSV;
4      public string HoTen;
5      public string NgaySinh;
6      public bool GioiTinh;
7      public string QueQuan;
8      public void Print()
9      {
10         Console.WriteLine(MSSV);
11         Console.WriteLine(HoTen);
12         Console.WriteLine(NgaySinh);
13         Console.WriteLine(GioiTinh);
14         Console.WriteLine(QueQuan);
15     }
16 }
```

```
1  //...
2  static void Main(string[] args)
3  {
4      SinhVien sv = new SinhVien();
5      sv.MSSV = "123456789";
6      //...
7      //Console.WriteLine(sv.MSSV);
8      sv.Print();
9  }
```

Kiểu đối tượng

- Một lớp đối tượng (class) là tập hợp các đối tượng có cùng thuộc tính và hành vi.

- Cú pháp:

```
class ClassName
{
    type fieldName1;
    type fieldName2;
}
```

Ví dụ 1.4.2. *Lớp đối tượng SinhVien quản lý các thông tin sau: Mssv, HoTen, NgaySinh, GioiTinh, QueQuan*

```
1 class SinhVien
2 {
3     public string MSSV;
4     public string HoTen;
5     public string NgaySinh;
6     public bool GioiTinh;
7     public string QueQuan;
8 }
```

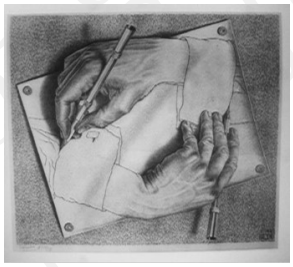
Chọn dữ liệu kiểu struct hay class?

- Struct: các biến được khai báo trong vùng nhớ stack.
- Class: các biến thành viên thuộc loại biến tham chiếu và được khai báo trong vùng nhớ heap.

1.5 Kỹ thuật đệ quy

Đệ quy

- Đệ quy *(recursion)* là vấn đề/bài toán được định nghĩa bằng chính nó.
- Một hàm được gọi là đệ quy, nếu bên trong thân của hàm đó có gọi lại chính nó một cách trực tiếp hay gián tiếp.



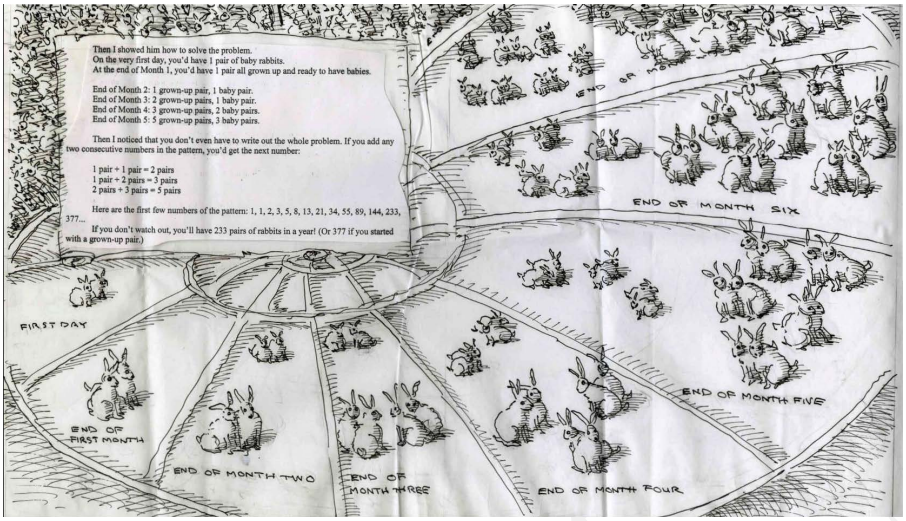
https://mcescher.com/gallery/back-in-holland

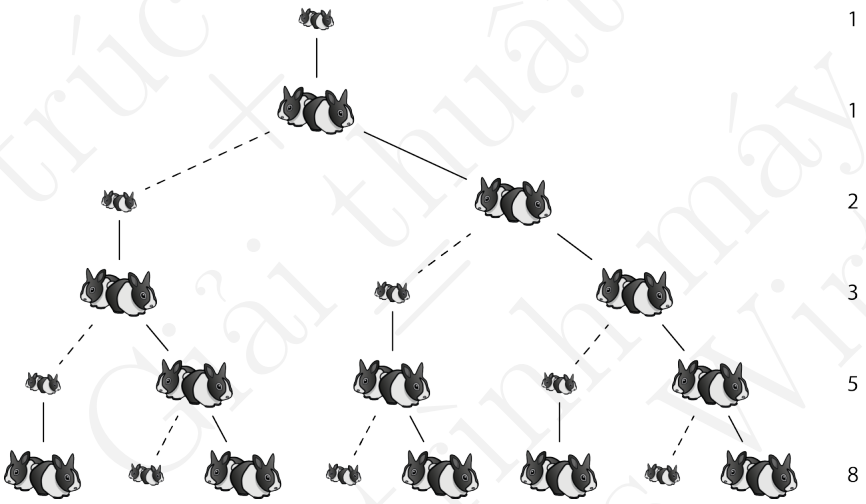
Ví dụ 1.5.1. *Bài toán nuôi thỏ*

- *Bắt đầu với một thỏ đực và một thỏ cái vừa mới chào đời.*
- *Thỏ đạt tới tuổi sinh sản sau một tháng.*
- *Thời gian mang thai của một con thỏ là một tháng.*
- *Sau khi tuổi sinh sản, thỏ cái đẻ đều đều mỗi tháng.*

- *Một thỏ cái sinh ra một thỏ đực và một thỏ cái.*
- *Không có thỏ chết.*

Hỏi sau một năm sẽ có bao nhiêu cặp thỏ?





https://curiosamathematica.tumblr.com/post/89242983089/the-life-and-numbers-of-fibonacci

Đệ quy

- Hàm đệ quy gồm hai phần
  - *Phần cơ sở*: điều kiện dừng quá trình gọi đệ quy.
  - *Phần đệ quy*: thân hàm chứa lời gọi đệ quy.
- *Bất kỳ một hàm đệ quy nào cũng phải có điều kiện dừng.*

Ví dụ 1.5.2. *Cho n là số nguyên không âm, tính $n!$ theo công thức*

$$f(n) = \begin{cases} 1 & , n = 0 \\ n \cdot f(n-1) & , n > 0 \end{cases}$$

Đệ quy tuyến tính *(Linear recursion)*

- Trong thân hàm có duy nhất một lời gọi hàm gọi lại chính nó một cách tường minh.

- Đệ quy đuôi *(Tail recursion)* là một trường hợp của đệ quy tuyến tính với lời gọi hàm nằm ở cuối mỗi hàm.

```
type TenHam(ThamSo)
{
    if (DieuKienDung)
    {
        ...;
        return GiaTri;
    }
    ...;
    TenHam(ThamSo);
}
```

Ví dụ 1.5.3. *Cho n là số nguyên không âm, $n!$ được định nghĩa như sau*

$$f(n) = \begin{cases} 1 & , n = 0 \\ n \cdot f(n-1) & , n > 0 \end{cases}$$

```
static long Factorial(int n)
{
    if (n == 0)
        return 1;
    return n * Factorial(n - 1);
}
```

Đệ quy nhị phân *(Binary recursion)*

- Trong thân hàm có hai lời gọi hàm gọi lại chính nó một cách tường minh.

```
type TenHam(ThamSo)
{
    if (DieuKienDung)
    {
        ...;
        return GiaTri;
    }
    ...;
    TenHam(ThamSo);
    ...;
    TenHam(ThamSo);
}
```

Ví dụ 1.5.4. *Dãy Fibonacci được định nghĩa như sau:*

$$f(n) = \begin{cases} 1 & , n = 0, 1 \\ f(n-1) + f(n-2) & , n > 1 \end{cases}$$

```
static long Fibonacci(int n)
{
```

```
3       if (n <= 1)
4           return 1;
5       return Fibonacci(n - 1) + Fibonacci(n - 2);
6   }
```

Đệ quy phi tuyến *(Nonlinear recursion)*

- Trong thân hàm có lời gọi hàm lại chính nó được đặt bên trong thân vòng lặp.

```
1   type TenHam(ThamSo)
2   {
3       if (DieuKienDung)
4       {
5           ...;
6           return GiaTri;
7       }
8       loop (DieuKieuLap)
9       {
10          ....;
11          TenHam(ThamSo);
12      }
13  }
```

Ví dụ 1.5.5. *Cho hàm $f(n)$ được định nghĩa như sau*

$$f(n) = \begin{cases} n & ,n \leq 4 \\ f(n-1) + f(n-2) + f(n-3) + f(n-4) & ,n > 4 \end{cases}$$

```
1   static long F(int n)
2   {
3       int i, result = 0;
4       if (n <= 4)
5           return n;
6       for (i = 1; i <= 4; i ++ )
7           result +=  F(n - i);
8       return result;
9   }
```

Đệ quy tương hỗ *(Mutual recursion)*

- Trong thân hàm 1 có lời gọi hàm tới hàm 2 và bên trong thân hàm 2 có lời gọi hàm đến hàm 1.

```
1   type TenHam1(ThamSo)
2   {
3       if (DieuKienDung)
4           return GiaTri;
5       TenHam2(ThamSo);
6   }
7
8   type TenHam2(ThamSo)
9   {
```

```
10      if (DieuKienDung)
11          return GiaTri;
12      TenHam1(ThamSo);
13  }
```

Ví dụ 1.5.6. *Cho n là một số nguyên không âm, hãy cho biết n là số chẵn hay lẻ?*

$$\begin{cases} n\,is\,even & ,n=0 \\ IsOdd\,(n-1) & ,n>0 \\ IsEven\,(n-1) & ,n>0 \end{cases}$$

trong đó, kết quả trả về 1 là số lẻ và 0 là số chẵn.

```
1   static bool IsOdd(int n)
2   {
3       if (n == 0)
4           return false;
5       return IsEven(n - 1);
6   }
```

```
1   static bool IsEven(int n)
2   {
3       if (n == 0)
4           return true;
5       return IsOdd(n - 1);
6   }
```

1.6 Bài tập cuối chương

1. Cho đoạn chương trình tính tổng từ $1+2+...+n$, hãy nhận xét về kết quả sau khi thực thi:

```
1       int s = 0;
2       int i = 0;
3       while ( i <= n )
4       {
5           i = i + 1;
6           s = s + i;
7       }
```

2. Cho hàm đệ quy giải bài toán Tháp Hà Nội như sau:

```
1   public static void HaNoiTower(int n, char from, char to,
        char via)
2   {
3       if (n == 1)
4           Console.WriteLine("Move 1 disk {0} to {1}", from,
                to);
5       else
6       {
7           HaNoiTower(n - 1, from, via, to);
```

```
8           HaNoiTower(1, from, to, via);
9           HaNoiTower(n - 1, via, to, from);
10      }
11  }
```

Giả sử $n = 3$ đĩa, cho biết phải thực hiện bao nhiêu bước chuyển đĩa từ cột A sang cột C? Liệt kê các bước.

3. Cho hàm đệ quy tính tích dãy số $P = n * (n-2) * (n-4) * \cdots$. Ví dụ, $n = 5 \Rightarrow P = 5 * 3 * 1, n = 8 \Rightarrow P = 8 * 6 * 4 * 2$.

```
1  public static long ProductError(int n)
2  {
3      if (n == 2)
4          return 2;
5      return n * ProductError(n - 2);
6  }
```

Hàm trên có đảm bảo tính dừng của một thuật toán hay không? Cho ví dụ và giải thích.

4. Cho hai số 6 và 46, hãy chạy từng bước thuật toán Euclid tính Ước chung lớn nhất và cho biết hàm đệ quy được gọi bao nhiêu lần.

5. Viết hàm đệ quy giải quyết các bài toán sau:

   (a) Tính tổng của n số nguyên dương đầu tiên

   $$S(n) = 1 + 2 + 3 + \cdots + n - 1 + n, n > 0.$$

   (b) Tính tổ hợp chập k của n phần tử

   $$C_n^k = \begin{cases} 1 & , k = 0 \vee k = n \\ C_{n-1}^k + C_{n-1}^{k-1} & , 0 < k < n \end{cases}$$

   (c) Tính $f(n)$

   $$f(n) = \begin{cases} n & , n \leq 3 \\ f(n-1) + 2f(n-2) + 3f(n-3) & , n > 3 \end{cases}$$

Bài 2

THUẬT TOÁN (3 tiết)

2.1 Giới thiệu thuật toán

Khái niệm cơ bản

Thuật toán *(algorithm - tên một nhà toán học người Trung Á là Abu Abd - Allah ibn Musa al'Khwarizmi, thường gọi là al'Khwarizmi)* là tập hợp hữu hạn các hướng dẫn rõ ràng để giải quyết một bài toán/vấn đề.
Trong lĩnh vực máy tính, thuật toán là một dãy *hữu hạn* các bước *không mập mờ* và *thực thi được*, quá trình hành động theo các bước này *phải dừng* và cho được *kết quả như mong muốn*.

Ví dụ 2.1.1. *Cho m, n là hai số nguyên dương, ước chung lớn nhất của m và n được định nghĩa theo công thức:*

$$gcd\,(m,n) = \{\forall m,n \in \mathcal{Z}, \exists g \in \mathcal{Z} : max\,\{g|m \wedge g|n\}\}\,.$$

trong đó, gcd (greastest common divisor) là ước chung lớn nhất.

- $gcd\,(3,5) = 1$
- $gcd\,(36,12) = 12$
- $gcd\,(27,15) = 3$

Thuật toán **Euclid** tìm ước chung lớn nhất của hai số nguyên dương.

- Bước 1: [Kiểm tra $m \geq n$] Nếu $m < n$ thì hoán vị m và n.
- Bước 2: [Tìm số dư r_i] Chia m cho n được số dư r_i, với $0 \leq r_i < n$ và $i \geq 0$.
- Bước 3: [Kiểm tra $r_i = 0$]
  - Nếu $r_i = 0$ thì thuật toán kết thúc. Trả về kết quả ước số chung lớn nhất là $n = (r_{i-1})$
  - Ngược lại, thực hiện bước 4.

- Bước 4: [Cập nhật giá trị m và n] Gán $m \leftarrow n$ và $n \leftarrow r$. Quay lại bước 2.

Ví dụ 2.1.2. *Tìm ước chung lớn nhất của 27 và 15.*

- *Gợi ý, kẻ bảng lưu trữ giá trị các bước thực hiện.*

	m	n	r
1			

Giải. • Khởi tạo $m = 27, n = 15$ và r là số dư của phép chia m cho n (hay $r = m \bmod n$).

- Thực hiện các bước theo thuật toán **Euclid** tìm ước chung lớn nhất của m và n như sau:

	m	n	r
1	27	15	12
2	15	12	3
3	12	3	0

- Kết luận: $gcd\,(27, 15) = 3$. □

2.1.1 Các tính chất của thuật toán

Một thuật toán nên thỏa các tính chất sau:

- Tính xác định = không mập mờ + thực thi được
- Tính hữu hạn
- Tính chính xác
- Đầu vào và đầu ra phải rõ ràng
- Tính hiệu quả
- Tính tổng quát

Tính xác định

- Mỗi bước của thuật toán phải được định nghĩa rõ ràng.

Ví dụ 2.1.3. *Thuật toán Euclid tìm ước chung lớn nhất của hai số m và n.*

- *Bước 1: [Kiểm tra $m \geq n$]*

Tính hữu hạn

- Thuật toán phải kết thúc sau một số bước thực hiện.

Ví dụ 2.1.4. *Thuật toán Euclid tìm ước chung lớn nhất của hai số nguyên dương m và n.*

- *Nếu $r \neq 0$ thì thuật toán tiếp tục thực hiện quá trình giảm giá trị r.*
- *Ngược lại $r = 0$, thuật toán kết thúc.*

Số bước thực hiện sẽ phụ thuộc giá trị 2 số m và n.

Tính chính xác

- Thuật toán phải tạo những giá trị đầu ra chính xác tương ứng với mỗi tập giá trị đầu vào.

Ví dụ 2.1.5. *Hãy nhận xét về tính chính xác của hai định nghĩa về số nguyên tố sau đây:*

- *Số nguyên tố là số nguyên dương chỉ chia hết cho* 1 *và chính nó.*
- *Số nguyên tố là số nguyên dương n bit không chia hết cho bất kỳ số nào có số bit $\leq \frac{n}{2}$ bit và lớn hơn* 1.

Giải. Số nguyên tố là số nguyên dương n bit không chia hết cho bất kỳ số nào có số bit $\leq \frac{n}{2}$ bit và lớn hơn 1?

- Xét $m = 9 = 1001_2$.
  - Vì $n = 4$ bit, nên các số nguyên biểu diễn dưới dạng nhị phân $\leq \frac{n}{2}$ bit và lớn hơn 1 gồm: $10_2 = 2, 11_2 = 3$.
  - Ta có, m *chia hết* 3.
  - Kết luận: m *không là số nguyên tố.*
- Xét $m = 25 = 11001_2$.
  - Vì $n = 5$ bit, nên các số nguyên biểu diễn dưới dạng nhị phân $\leq \frac{n}{2}$ bit và lớn hơn 1 gồm: $10_2 = 2, 11_2 = 3$.
  - Ta có, m không chia hết cho 2 và 3.
  - Có thể kết luận m *là số nguyên tố* ?
  - Thực tế, m chia hết cho $101_2 = 5$. □

Đầu vào và đầu ra của thuật toán (input, output)

- Đầu vào của thuật toán được lấy từ một tập xác định. Từ mỗi tập giá trị đầu vào, thuật toán sẽ tạo tập giá trị đầu ra tương ứng.

Ví dụ 2.1.6. *Thuật toán Euclid tìm ước chung lớn nhất của hai số nguyên dương m và n.*

- *Đầu vào: hai số nguyên dương m và n.*
- *Đầu ra: một số nguyên dương là ước chung lớn nhất của m và n.*

Tính hiệu quả

- Một thuật toán hiệu quả phải có độ phức tạp *thời gian* và *không gian* thực hiện *nhỏ hơn* các thuật toán khác.

Ví dụ 2.1.7. *Tính tổng* 1.000.000 *số nguyên dương đầu tiên.*

- *Cách 1: thực hiện thao tác lặp* 1.000.000 *lần, mỗi lần cộng với một số nguyên dương.*
- *Cách 2: sử dụng công thức Gauss tính tổng* n *số nguyên dương đầu tiên* $S = \frac{n(n+1)}{2}$.

Tính tổng quát

- Thuật toán phải áp dụng được cho mọi trường hợp của bài toán có dạng theo yêu cầu.

Ví dụ 2.1.8. *Giải phương trình bậc hai* $ax^2 + bx + c = 0, a \neq 0$ *dựa vào Delta.*

- *Thuật toán này luôn giải được với giá trị hệ số* a, b, c *bất kỳ.*

Thuật toán & Giải thuật

Giải thuật là khái niệm mở rộng/nâng cao của Thuật toán. Trong thực tế, nhiều bài toán không thể giải được theo một thuật toán và thỏa đầy đủ các tính chất mà thuật toán phải có. Nên người ta có thể chấp nhận bài toán với kết quả gần đúng nhưng thực hiện nhanh, thay vì bài toán tối ưu với kết quả rất tốt nhưng thời gian thực hiện rất chậm.

2.2 Phương pháp biểu diễn thuật toán

Một thuật toán có thể được thể hiện bởi nhiều phương pháp khác nhau như:

- Ngôn ngữ tự nhiên *(natural language)*
- Sơ đồ khối *(flowchart)*
- Mã giả *(pseudocode)*

Khái niệm

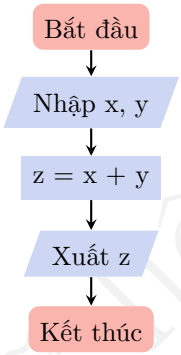
- Sử dụng ngôn ngữ bình thường để mô tả thuật toán.
- Thường viết theo dạng phân cấp 1, 1.1, 1.1.1
- Không có cấu trúc, dài dòng.

Ví dụ 2.2.1. *Nhập một số nguyên* n*. Cho biết* n *là số chẵn hay số lẻ.*

- *Bước 1: Nhập số nguyên* n*.*
- *Bước 2: Kiểm tra* $n \mod 2 = 0$*:*
  - *Bước 2.1: Nếu* $n \mod 2 = 0$*, thông báo* n *là số chẵn và chuyển qua bước 3.*
  - *Bước 2.2: Ngược lại, thông báo* n *là số lẻ.*
- *Bước 3: Kết thúc thuật toán.*

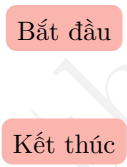
2.2.1 Sơ đồ khối

- Là công cụ trực quan để mô tả các thuật toán.
- Sử dụng các hình đại diện tương ứng với những thao tác trong thuật toán.



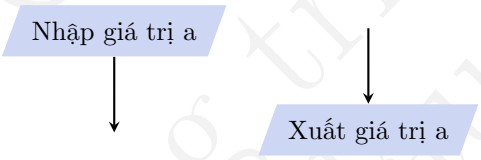
Điểm cuối

- Biểu diễn bằng hình ovan và được ghi chú *Bắt đầu* hay *Kết thúc*.
- Chỉ ra điểm bắt đầu hay kết thúc của thuật toán.



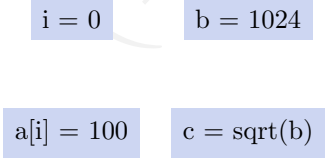
Đầu vào, đầu ra

- Biểu diễn bằng hình bình hành.
- Nhập, xuất các giá trị trong thuật toán.



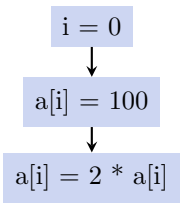
Thao tác xử lý (process)

- Biểu diễn bằng hình chữ nhật.
- Chứa nội dung các xử lý toán học, gán giá trị.



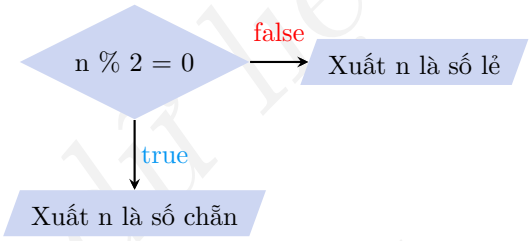
Thao tác tuần tự (sequence)

- Là một chuỗi các thao tác xử lý liên tiếp.
- Mũi tên thể hiện đường đi giữa các thao tác.



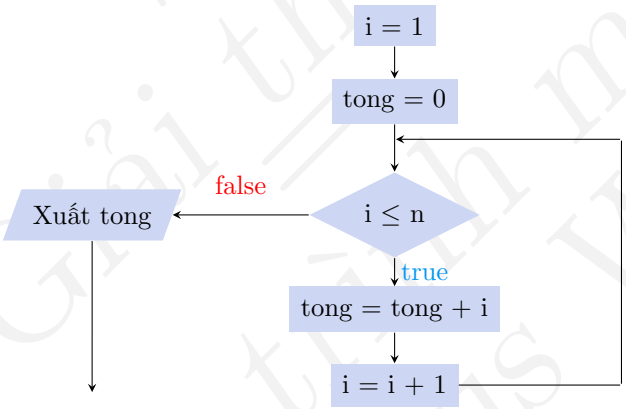
Thao tác chọn (rẽ nhánh)

- Biểu diễn bởi hình thoi.
- Có hai đường đi tương ứng với thỏa hay không thỏa điều kiện.

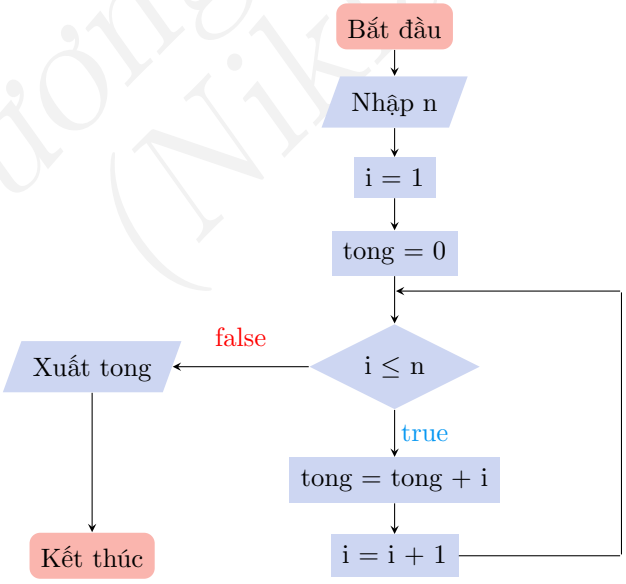


Thao tác lặp

- Được kết hợp từ nhiều thao tác liên tiếp nhau.
- Chia 2 loại: vòng lặp xác định và vòng lặp không xác định.



Ví dụ 2.2.2. *Vẽ sơ đồ khối mô tả các bước tính tổng n số nguyên dương đầu tiên.*



2.2.2 Mã giả

Mã giả *(pseudocode)* là phương pháp biểu diễn thuật toán có sử dụng cú pháp giống như một ngôn ngữ lập trình nào đó.

Ví dụ 2.2.3. *Tính tổng dãy số* $1+2+3+\cdots+n$ *với* $n>0$.

Thuật toán 2.1: Sum(n)
- Đầu vào: số nguyên dương n.
- Đầu ra: tổng các số nguyên dương từ 1 → n.

```
1   sum ← 0
2   if n > 0
3       for i ← 1 to n
4           sum ← sum + i
5   return sum
```

2.3 Độ phức tạp thuật toán

2.3.1 Các khái niệm cơ bản

Độ phức tạp

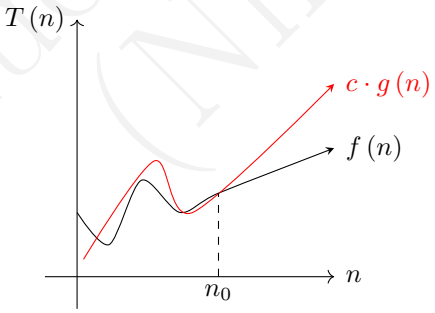
Độ phức tạp (complexity) của thuật toán là tổng các chi phí để thực hiện một thuật toán nào đó.

- Độ phức tạp thời gian (các phép tính toán)
- Độ phức tạp không gian (vùng nhớ lưu trữ khi thực hiện thuật toán)

Các ký hiệu tiệm cận

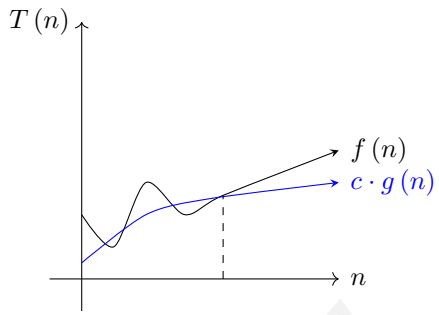
Định nghĩa 2.3.1 (Big-Oh). Hàm $f(n)$ là $O(g(n))$ nếu f có tỷ lệ tăng trưởng *(growth rate)* nhiều khi đến g:

$$\exists c, n_0 \in R^+, \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$



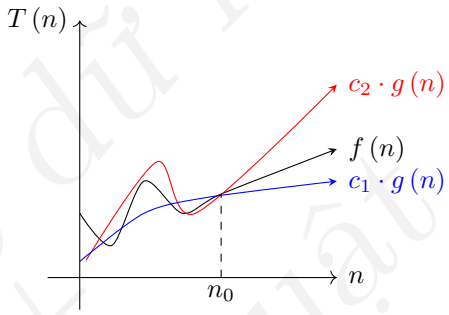
Định nghĩa 2.3.2 (Big-Omega). Hàm $f(n)$ là $\Omega(g(n))$ nếu f có tỷ lệ tăng trưởng ít khi đến g:

$$\exists c \in R^+, \forall n \in N : f(n) \geq c \cdot g(n).$$



Định nghĩa 2.3.3 (Big-Theta). Hàm $f(n)$ là $\Theta(g(n))$ nếu và chỉ nếu:

$$\exists c_1, c_2, n_0 \in R^+, \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$



Ví dụ 2.3.1. *Chứng minh rằng* $2n = O(n)$.

Chứng minh.

- Ta có, $f(n) = 2n \leq c \cdot n, \forall n \geq 1$
- Chọn $n_0 = 1$, ta lại có
$$\forall n \geq n_0, f(n) \leq c \cdot g(n)$$
với $c = 2$ và $g(n) = n$
- Suy ra, $f(n) = O(n)$.

Ví dụ 2.3.2. *Chứng minh rằng* $n^2 + 2n + 1 = O(n^2)$.

Chứng minh.

- Ta có, $f(n) = n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2, \forall n \geq 1$
- Chọn $n_0 = 1$, ta lại có
$$\forall n \geq n_0, f(n) \leq c \cdot g(n)$$
với $c = 4$ và $g(n) = n^2$
- Suy ra, $f(n) = O(n^2)$.

Để so sánh độ hiệu quả, ta phải đánh giá các thuật toán dựa trên tập mẫu xác định với cùng kích thước dữ liệu đầu vào.

Ví dụ 2.3.3. *Gọi số nguyên dương* n *là kích thước dữ liệu đầu vào của một thuật toán. Khi đó,* n *được xác định phụ thuộc vào từng bài toán như sau:*

- *Tìm kiếm, sắp xếp:* $n =$ *số phần tử của mảng.*
- *Xử lý chuỗi:* $n =$ *chiều dài chuỗi.*
- *Ma trận:* $n =$ *số chiều ma trận (trường hợp ma trận vuông),* $n \times n$ *phần tử.*
- *Đồ thị:* $n_V =$ *số đỉnh và* $n_E =$ *số cạnh của đồ thị.*

Các bước phân tích độ phức tạp thời gian

1. Xác định phép toán cơ sở.
2. Tính số lần thực hiện phép toán cơ sở của một hàm.

Công thức tính độ phức tạp thời gian

$$T(n) \approx c \cdot g(n).$$

Trong đó,

- $T(n)$: thời gian chạy của hàm,
- c: thời gian chạy của phép toán cơ sở,
- $g(n)$: số lần thực hiện các phép toán cơ sở của một hàm với n là kích thước dữ liệu đầu vào.

Độ phức tạp của thuật toán được phân thành một số lớp như bảng sau:

Bảng 2.1: Một số lớp độ phức tạp thời gian của thuật toán.

Độ phức tạp	Thuật ngữ
$O(1)$	Độ phức tạp hằng số
$O(\log n)$	Độ phức tạp logarit
$O(n)$	Độ phức tạp tuyến tính
$O(n \log n)$	Độ phức tạp $n \log n$
$O\left(n^b\right), b > 1$	Độ phức tạp đa thức
$O(b^n)$	Độ phức tạp hàm mũ
$O(n!)$	Độ phức tạp giai thừa

Ví dụ 2.3.4. *Giả sử, vòng lặp của một chương trình thực hiện trong thời gian 1 giây. Khi vòng lặp thực hiện 1.000.000 lần, hãy tính thời gian chạy của hai thuật toán có độ phức tạp lần lượt là* $O(n)$ *và* $O(\log n)$

Giải. Khi thực hiện vòng lặp nhiều lần, chúng ta thấy rõ sự khác biệt về thời gian thực hiện:

- $O(1.000.000) \approx 1.000.000$
- $O(\log 1.000.000) \approx 19,93$ □

QUY TẮC CỘNG: nếu $T_1(n) = O(g_1(n))$ và $T_2(n) = O(g_2(n))$ là thời gian thực hiện của 2 đoạn chương trình P_1 và P_2 thì thời gian thực hiện của 2 đoạn chương trình đó *nối tiếp nhau* là

$$T(n) = O(g_1(n) + g_2(n)).$$

Hay lấy giá trị của $g_i(n)$ lớn nhất

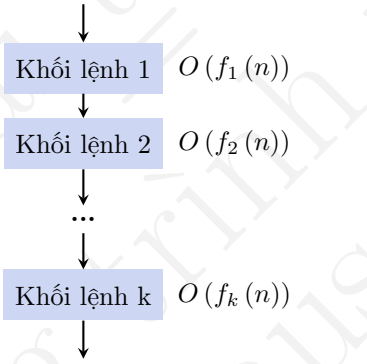
$$T(n) = O(max(g_1(n), g_2(n))).$$

QUY TẮC NHÂN: nếu $T_1(n) = O(g_1(n))$ và $T_2(n) = O(g_2(n))$ là thời gian thực hiện của 2 đoạn chương trình P_1 và P_2 thì thời gian thực hiện của 2 đoạn chương trình đó *lồng vào nhau* là

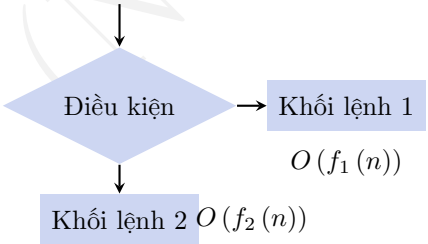
$$T(n) = O(g_1(n) \cdot g_2(n)).$$

Độ phức tạp thời gian

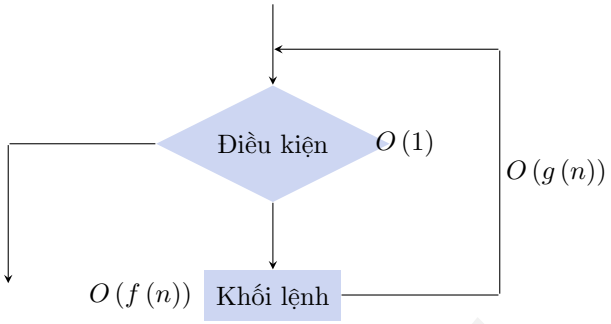
- Phép toán cơ sở (so sánh, gán, ...): $O(1)$.
- Các phép toán nối tiếp nhau: quy tắc cộng.
- Cấu trúc điều kiện (`if`): là thời gian lớn nhất sau `if` hay `else` và thời gian kiểm tra điều kiện (thường thời gian kiểm tra điều kiện là $O(1)$).
- Cấu trúc lặp (`for, while`): quy tắc nhân.



$$O(max\{f_1(n), f_2(n), \ldots, f_k(n)\})$$



$$O(max\{f_1(n), f_2(n)\})$$



$$O(f(n) \cdot g(n))$$

2.3.2 Độ phức tạp thuật toán không đệ quy

Ví dụ 2.3.5. *Tìm số lớn nhất trong mảng a gồm n phần tử cho trước.*

Thuật toán 2.2: `Max(a[], n)`
- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: trả về phần tử lớn nhất trong mảng a.

```
1    max ← a[0]
2    for i ← 1 to n - 1
3        if max < a[i]
4            max ← a[i]
5    return max
```

- Phép tính cơ sở: phép so sánh `max < a[i]`.
- Thời gian thực hiện: $T(n) = \sum_{i=1}^{n-1} 1 = n - 1 = O(n)$.

Độ phức tạp thuật toán không đệ quy

Ví dụ 2.3.6. *Kiểm tra các phần tử trong mảng A có trùng nhau hay không?*

Thuật toán 2.3: `IsDuplicate(a[], n)`
- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: trả về true/false.

```
1    for i ← 0 to n - 2
2        for j ← i + 1 to n - 1
3            if a[i] = a[j]
4                return true
5    return false
```

- Phép tính cơ sở: phép so sánh `a[i] = a[j]`.
- Thời gian thực hiện: $T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} n - 1 - i = \frac{n(n-1)}{2} = O\left(n^2\right)$.

2.3.3 Độ phức tạp thuật toán đệ quy

Đối với các thuật toán có sử dụng kỹ thuật đệ quy, độ phức tạp có thể được tính bằng những phương pháp sau:

- Phương pháp lặp (iteration method), còn được gọi là phương pháp thay thế.
  - Mở rộng quá trình đệ quy k lần. $k =?$
  - Thực hiện tính toán để tìm được công thức tính tổng.
  - Ước lượng công thức vừa tìm được để đưa về một lớp độ phức tạp thời gian của thuật toán.
- Thường sử dụng đối với các thuật toán sử dụng kỹ thuật chia để trị *(divide and conquer)*.

Trước khi tìm hiểu phương pháp lặp, chúng ta cần nhớ lại kiến thức về cấp số nhân.Cấp số nhân là một dãy số (*hữu hạn* hay *vô hạn*) với mỗi số hạng *(trừ số hạng đầu tiên)* đều bằng tích của số hạng đứng ngay trước nó và một số r không đổi.

- Số hạng thứ i của cấp số nhân:

$$a_i = ar^{i-1}, \quad i \geq 1.$$

- Số r được gọi là công bội của cấp số nhân.

  Ví dụ 2.3.7. *Các dãy số sau là một cấp số nhân.*
  - $1, 3, 9, 27$
  - $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$

- Giả sử có cấp số nhân với công bội r. Khi đó, tổng n số hạng đầu tiên:

$$S_n = a + ar^1 + ar^2 + \cdots + ar^n. \tag{2.1}$$

  - Dãy hữu hạn: nếu $r > 1$, thì

$$S_n = a\sum_{i=0}^{n} r^i = \frac{a\left(1 - r^n\right)}{1 - r}. \tag{2.2}$$

  - Dãy vô hạn: với $0 < r < 1$, thì

$$S = a\sum_{i=0}^{\infty} r^i = \frac{1}{1 - r} \tag{2.3}$$

Chúng ta sẽ áp dụng phương pháp lặp tính độ phức tạp của một số công thức truy hồi phổ biến.

Công thức 1

$$T\left(n\right) = \begin{cases} c_0 & , n = 0 \\ T\left(n-1\right) + cn & , n > 0 \end{cases}$$

Chứng minh.

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + c(n-1) + cn \\ &= T(n-3) + c(n-2) + c(n-1) + cn \\ &\dots \\ &= T(n-k) + c(n-k+1) + \dots + c(n-2) + c(n-1) + cn \\ &= T(n-k) + c \cdot \sum_{i=n-k+1}^{n} i \qquad , n \geq k \end{aligned}$$

Chứng minh.

- Giả sử $n = k$, thuật toán dừng đệ quy.

$$T(n) = T(0) + c \cdot \sum_{i=1}^{n} i = c_0 + c \cdot \frac{n(n+1)}{2}$$

- Do đó,

$$T(n) = \frac{n(n+1)}{2} \approx \frac{n^2}{2} = O\left(n^2\right).$$

Nhận xét

- Công thức 1 thường dùng cho chương trình đệ quy có vòng lặp duyệt qua dữ liệu nhập để bỏ một phần tử.

Công thức 2

$$T(n) = \begin{cases} c_0 & , n = 1 \\ T\left(\frac{n}{2}\right) + c & , n > 1 \end{cases}$$

Chứng minh.

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c = T\left(\frac{n}{8}\right) + 3c \\ &\dots \\ &= T\left(\frac{n}{2^k}\right) + k \cdot c \qquad , n \geq 2^k \end{aligned}$$

- Giả sử $n = 2^k$, thuật toán dừng đệ quy.
  $T(n) = T(1) + k \cdot c = c_0 + k \cdot c = k = \log n$
- Do đó,

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

Nhận xét

- Công thức 2 thường dùng cho chương trình đệ quy mà dữ liệu nhập được chia thành hai phần mỗi bước thực hiện.

Công thức 3

$$T(n) = \begin{cases} c_0 & , n = 1 \\ T\left(\frac{n}{2}\right) + n & , n > 1 \end{cases}$$

Chứng minh.

$$\begin{aligned} T(n) &= T\left(\tfrac{n}{2}\right) + n \\ &= T\left(\tfrac{n}{4}\right) + \tfrac{n}{2} + n \\ &= T\left(\tfrac{n}{8}\right) + \tfrac{n}{4} + \tfrac{n}{2} + n \\ &\dots \\ &= T\left(\tfrac{n}{2^k}\right) + \tfrac{n}{2^{k-1}} + \dots + \tfrac{n}{4} + \tfrac{n}{2} + n \qquad , n \geq 2^k \\ &= T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \frac{n}{2^i} \end{aligned}$$

- Giả sử $n = 2^k$, thuật toán dừng đệ quy.
  $T(n) = T(1) + 2^k \sum_{i=0}^{k-1} \frac{1}{2^i} = c_0 + 2^k \sum_{i=0}^{k-1} \frac{1}{2^i} = 2^k \sum_{i=0}^{k-1} \frac{1}{2^i}$
- Tính tổng số hạng của cấp số nhân của dãy số có dạng $\sum_{i=0}^{m} \frac{1}{2^i}$
  - Ta có, phần tử đầu tiên của chuỗi là $a = 1$ và công bội $r = \frac{1}{2}$.
  - Áp dụng công thức tính tổng của dãy số vô hạn $\sum_{i=0}^{\infty} \frac{1}{2^i} = \frac{1}{1-r} = 2$
- Do đó, $\sum_{i=0}^{k-1} \frac{1}{2^i} = \frac{1}{1-r} = 2$ ■
- Do đó,
$$T(n) = 2^k \cdot 2 = 2n = O(n).$$

Nhận xét

- Công thức 3 thường dùng cho chương trình đệ quy mà dữ liệu nhập được chia thành hai phần nhưng có thể kiểm tra mỗi phần tử của dữ liệu nhập.

Công thức 4

$$T(n) = \begin{cases} c_0 & , n = 1 \\ 2T\left(\frac{n}{2}\right) + n & , n > 1 \end{cases}$$

Chứng minh.

$$\begin{aligned} T(n) &= 2T\left(\tfrac{n}{2}\right) + n \\ &= 4T\left(\tfrac{n}{4}\right) + n + n \\ &= 8T\left(\tfrac{n}{8}\right) + n + n + n \\ &\dots \\ &= nT\left(\frac{n}{2^k}\right) + k \cdot n \qquad , n \geq 2^k \end{aligned}$$

- Giả sử $n = 2^k$, thuật toán dừng đệ quy.

$$\begin{aligned} T(n) &= 2^k \cdot T(1) + k \cdot 2^k \\ &= 2^k \cdot c_0 + k \cdot 2^k \\ &= n \cdot c_0 + n \log n \end{aligned}$$

- Do đó,

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

Nhận xét

- Công thức 4 thường dùng cho chương trình đệ quy mà duyệt tuyến tính dữ liệu nhập trước, trong hay sau khi được chia thành hai phần.

Công thức 5

$$T(n) = \begin{cases} c_0 & , n = 1 \\ 2T\left(\frac{n}{2}\right) + c & , n > 1 \end{cases}$$

Chứng minh.

$$\begin{aligned} T(n) &= 2T\left(\tfrac{n}{2}\right) + c \\ &= 4T\left(\tfrac{n}{4}\right) + 2c + c \\ &= 8T\left(\tfrac{n}{8}\right) + 4c + 2c + c \\ &\dots \\ &= nT\left(\frac{n}{2^k}\right) + c \cdot \sum_{i=0}^{k-1} 2^i \qquad , n \geq 2^k \end{aligned}$$

- Giả sử $n = 2^k$, thuật toán dừng đệ quy.

$$T(n) = 2^k \cdot T(1) + c \cdot \sum_{i=0}^{k-1} 2^i = 2^k \cdot c_0 + c \cdot \sum_{i=0}^{k-1} 2^i.$$

- Tính tổng số hạng của cấp số nhân hữu hạn có dạng $\sum_{i=0}^{m} 2^i$
  - Ta có, phần tử đầu tiên là $a = 1$ và công bội $r = 2$.
  - Áp dụng công thức tính tổng của dãy số hữu hạn $\sum_{i=0}^{m} 2^i = \frac{a(1 - r^m)}{1 - m} = 2^m - 1$.
- Do đó, $\sum_{i=0}^{k-1} 2^i = 2^{k-1} - 1$. ■
- Do đó,

$$T(n) = 2^k \cdot c_0 + 2^{k-1} \cdot c - c = n \cdot c_0 + \frac{n}{2} \cdot c - c = O(n).$$

Nhận xét

- Công thức 5 thường dùng cho chương trình đệ quy mà mỗi bước thực hiện dữ liệu được chia thành hai phần.

Cho $T(n)$ là công thức truy hồi của thuật toán:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{2.4}$$

với $a \geq 1, b > 1$. Nếu $f(n) \in O\left(n^d\right), d \geq 0$, thì

1. Nếu $a < b^d \Rightarrow T(n) = O\left(n^d\right)$.
2. Nếu $a = b^d \Rightarrow T(n) = O\left(n^d \log n\right)$.
3. Nếu $a > b^d \Rightarrow T(n) = O\left(n^{\log_b a}\right)$.

Trong đó,

- a: số bài toán con cần được xử lý.
- b: kích thước bài toán con (thường là 2).
- $f(n)$: chi phí chia bài toán con và chi phí tổng hợp kết quả.

Ví dụ 2.3.8.

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

- *Ta có,* $a = 4, b = 2, d = 3 \Rightarrow a < b^d$ *(trường hợp 1)*
- *Suy ra,* $T(n) = O\left(n^3\right)$.

Ví dụ 2.3.9.

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

- *Ta có,* $a = 4, b = 2, d = 2 \Rightarrow a = b^d$ *(trường hợp 2)*
- *Suy ra,* $T(n) = O\left(n^2 \log n\right)$.

Ví dụ 2.3.10.

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

- *Ta có,* $a = 2, b = 2, d = 0 \Rightarrow a > b^d$ *(trường hợp 3)*
- *Suy ra,* $T(n) = O\left(n^{\log_2 2}\right) = O(n)$.

2.4 Bài tập cuối chương

1. Chứng minh rằng:
   (a) $2n + 7 = O(n)$
   (b) $n^2 + 2n = O\left(n^2\right)$
   (c) $n^3 + 2n^2 + n + 1 = O\left(n^3\right)$
   (d) $(n+1)^3 = O\left(n^3\right)$

2. Áp dụng Định lý Chủ để tính độ phức tạp tính toán của các thuật toán sau:

   (a) Tính độ phức tạp tính toán của thuật toán Tháp Hà Nội *(Towers of Hanoi)*

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

   (b) Tính độ phức tạp tính toán của thuật toán tính $n!$

$$T(n) = \begin{cases} 1 & , n = 0 \\ nT(n-1) & , n > 0 \end{cases}$$

   (c) Sắp xếp trộn *(MergeSort)*

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

   (d) Strassen nhân hai ma trận

$$T(n) = \begin{cases} 1 & , n = 1 \\ 7T\left(\frac{n}{2}\right) + n^2 & , n > 1 \end{cases}$$

3. Cài đặt thuật toán Euclid tìm ước số chung lớn nhất của 2 số nguyên dương.

4. Cài hàm đệ quy tính x lũy thừa n, với n là một số nguyên.

5. Cho một mảng A lưu trữ dãy gồm n số nguyên dương đầu tiên. Cài đặt thuật toán kiểm tra mảng A có thiếu phần tử nào hay không? Yêu cầu thuật toán có độ phức tạp tính toán là $O(1)$. Ví dụ, mảng A lưu trữ 10 số nguyên dương đầu tiên:

$$A = \{1, 2, 3, 4, 5, 7, 8, 9, 10\}$$

   Kết quả thuật toán trả về là $false$ vì mảng thiếu số 6.

Bài 3

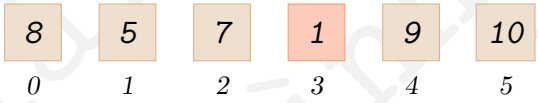
THUẬT TOÁN TÌM KIẾM (3 tiết)

3.1 Giới thiệu bài toán tìm kiếm

Khái niệm

- Sắp xếp là quá trình xử lý các phần tử của một danh sách, dãy số, dãy ký tự, ... theo đúng thứ tự (thỏa tiêu chuẩn nào đó).
- Trong bài toán sắp xếp, hai thao tác cơ bản là so sánh và gán giữa hai phần tử trong dãy.

Ví dụ 3.1.1. *Tìm phần tử có giá trị 1 trong dãy số:*



3.2 Tìm kiếm tuyến tính

3.2.1 Tìm kiếm tuyến tính/tuần tự *(linear search)*

Ý tưởng

Giải thuật lần lượt so sánh phần tử x cần tìm với phần thử thứ nhất, thứ hai, ... của dãy và dừng thực hiện khi:

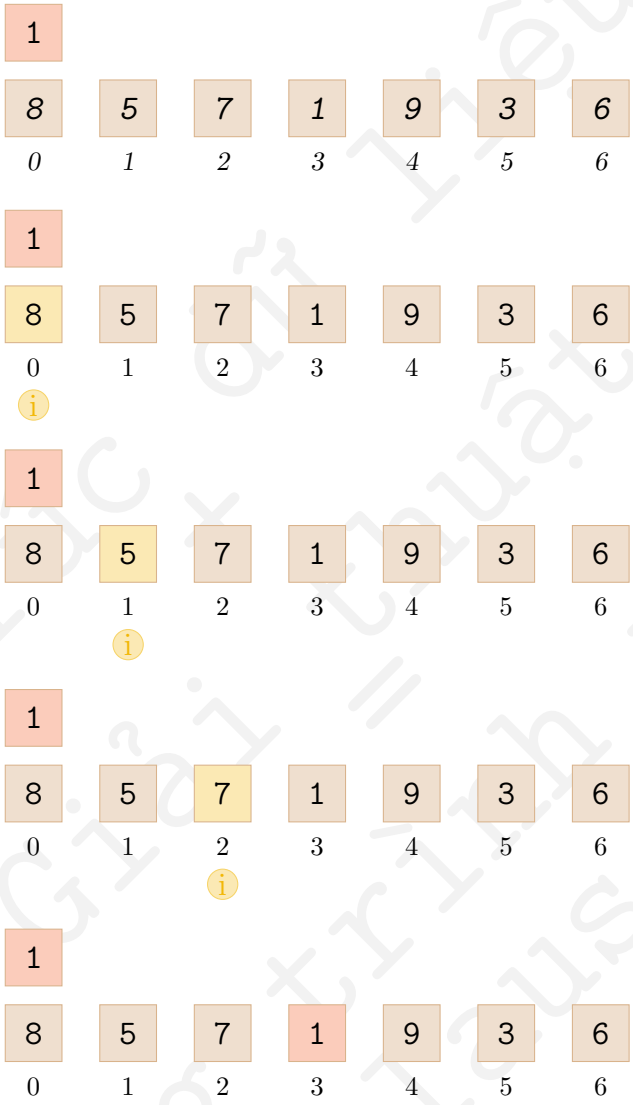
- Tìm được phần tử có khóa cần tìm.
- Duyệt hết các phần tử của dãy (không tìm thấy).

Thuật toán 3.1: `LinearSearch(a[], n, x)`
`- Đầu vào: mảng a gồm n phần tử và phần tử x.`
`- Đầu ra: vị trí của x hay -1 (không tìm thấy).`

```
1   i ← 0
2   while i < n and a[i] ≠ x
3       i ← i + 1
```

```
4       if i = n
5           return -1
6       else
7           return i
```

Ví dụ 3.2.1. *Cho dãy số a gồm 7 phần tử:* $8, 5, 7, 1, 9, 3, 6$. *Tìm phần tử có giá trị* 1.



Nhận xét

- Tại mỗi vòng lặp, giải thuật phải sử dụng 2 phép so sánh $i < n$ và $a[i] \neq x$.
- Cải tiến giải thuật bằng cách sử dụng phương pháp lính canh *(sentinel)* để loại bỏ điều kiện $i < n$.

Thuật toán 3.2: `LinearSearchEx(a[], n, x)`,
- Đầu vào: mảng a gồm n phần tử và phần tử x.
- Đầu ra: vị trí phần tử x hay -1 (không tìm thấy).

```
1       i ← 0
2       a[n] ← x
```

```
3       while a[i] ≠ x
4           i ← i + 1
5       if i = n
6           return -1
7       else
8           return i
```

Phương pháp lính canh

Trường hợp	Số phép so sánh
Tốt nhất	1
Xấu nhất	$n+1$
Trung bình	$\frac{n+1}{2}$
Độ phức tạp thời gian	$T(n) = O(n)$

3.3 Tìm kiếm nhị phân

Ý tưởng

Xét một dãy a *có thứ tự* gồm n phần tử: $a_0 < a_1 < a_2 < ... < a_{n-1}$.

- Nếu $a_i = x$, tìm thấy phần tử x tại vị trí i.
- Nếu $a_i > x$ thì chỉ có thể tìm x trong dãy con trái

$$a_0 \leq x \leq a_{i-1}.$$

- Ngược lại, $a_i < x$ thì chỉ có thể tìm x trong dãy con phải

$$a_{i+1} \leq x \leq a_{n-1}.$$

Sau mỗi lần so sánh, tiếp tục thực hiện tìm kiếm nhị phân với dãy con.

Phương pháp đệ quy

Thuật toán 3.3: BinarySearch(a[], left, right, x)
- Đầu vào: mảng a gồm các phần tử và phần tử x.
- Đầu ra: vị trí của x hay -1 (không tìm thấy).

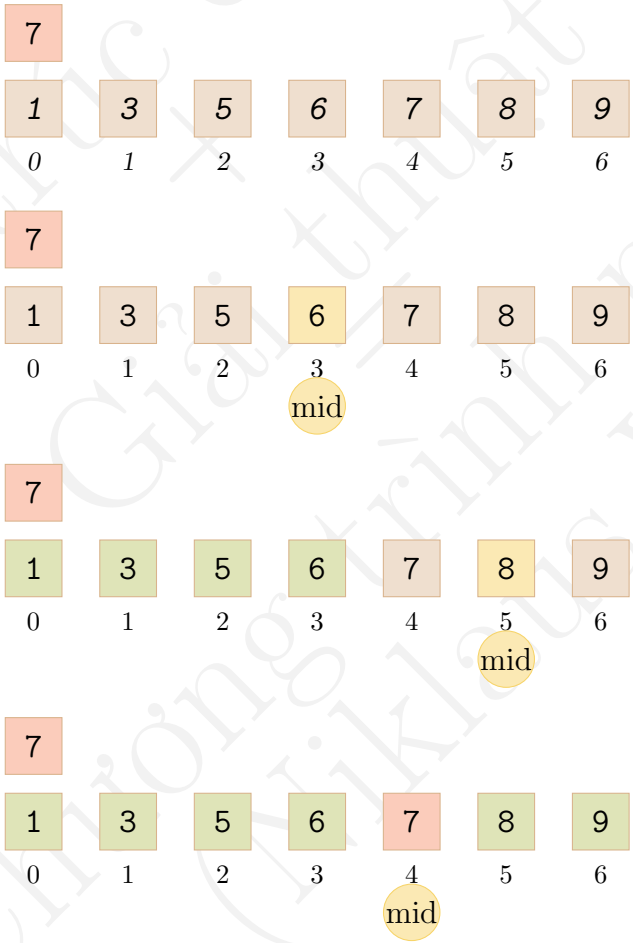
```
1       if left > right
2           return -1
3       mid ← (left + right) / 2
4       if a[mid] = x
5           return mid
6       else if a[mid] > x
7           BinarySearch(a, left, mid - 1, x)
8       else
9           BinarySearch(a, mid + 1, right, x)
```

Phương pháp lặp

Thuật toán 3.4: BinarySearch(a[], n, x)
- Đầu vào: mảng a gồm n phần tử và phần tử x.
- Đầu ra: vị trí của x hay -1 (không tìm thấy).

```
1     left ← 0, right ← n - 1
2     while left <= right
3         mid ← (left + right) / 2
4     if a[mid] = x
5         return mid
6     else a[mid] > x
7         right ← mid - 1
8     else
9         left ← mid + 1
10    return -1
```

Ví dụ 3.3.1. *Cho dãy số a đã sắp thứ tự gồm 7 phần tử:* $1, 3, 5, 6, 7, 8, 9$. *Tìm phần tử có giá trị 7.*



Độ phức tạp thuật toán

Thuật toán tìm kiếm nhị phân sử dụng kỹ thuật đệ quy có độ phức tạp như sau:

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

Trường hợp	Số phép so sánh
Tốt nhất	1
Xấu nhất	$\log_2 n$
Trung bình	$\frac{\log_2 n}{2}$
Độ phức tạp thời gian	$T(n) = O(\log_2 n)$

3.4 Bài tập cuối chương

1. Cho dãy các số nguyên $12, 2, 8, 5, 1, 6, 4, 15$. Áp dụng giải thuật tìm kiếm tuyến tính và tìm kiếm nhị phân tìm phần tử có giá trị 15.
2. Cho một dãy số gồm 32 phần tử đã có thứ tự. Giả sử phần tử cần tìm thuộc 1 trong 4 vị trí đầu tiên của dãy. Hãy cho biết giữa LinearSearch và BinarySearch, thuật toán nào số bước thực hiện ít hơn?
3. Cho dãy các số nguyên $1, 2, 4, 5, 6, 8, 12, 15$. Xây dựng giải thuật tìm kiếm tuyến tính đối với *dãy có thứ tự tăng dần.*
4. Xây dựng giải thuật tìm kiếm tam phân *(Ternary Search)* để tìm phần tử x trong dãy số đã có thứ tự.

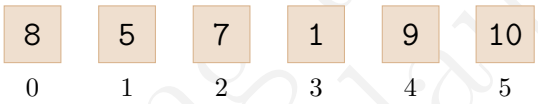
Buổi 4

CÁC THUẬT TOÁN SẮP XẾP CƠ BẢN (3 tiết)

4.1 Giới thiệu bài toán sắp xếp

Khái niệm

- Sắp xếp là quá trình xử lý các phần tử của một danh sách, dãy số, dãy ký tự, ... theo đúng thứ tự (thỏa tiêu chuẩn nào đó).
- Trong bài toán sắp xếp, hai thao tác cơ bản là so sánh và gán giữa hai phần tử trong dãy.



Định nghĩa 4.1.1. Xét dãy số a gồm n phần tử

$$a = \{a_0, a_1, ..., a_{n-1}\}$$

- Nếu $i < j$ và $a_i > a_j$ thì gọi đó là một nghịch thế (trường hợp tăng dần).
- Mảng không thứ tự: chứa nghịch thế.
- Mảng có thứ tự: không chứa nghịch thế.

4.2 Sắp xếp chọn trực tiếp

Ý tưởng

Cho dãy $a = \{a_0, a_1, ..., a_{n-1}\}$ gồm n phần tử, giải thuật là một vòng lặp thực hiện $n-1$ lần.
Tại mỗi lần lặp thứ $i = 0, 1, ..., n-2$

- Chọn phần tử nhỏ nhất trong dãy $a_i, a_{i+1}, ..., a_{n-1}$.
- Hoán vị phần tử được chọn với a_i.

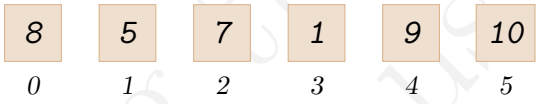
Thuật toán 4.1: `SelectionSort(a[], n)`
- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: mảng a có thứ tự tăng dần.

```
1   for i ← 0 to n - 2
2       min ← i
3       for j ← i + 1 to n - 1
4           if a[j] < a[min]
5               min ← j
6       Swap(a[i], a[min])
```

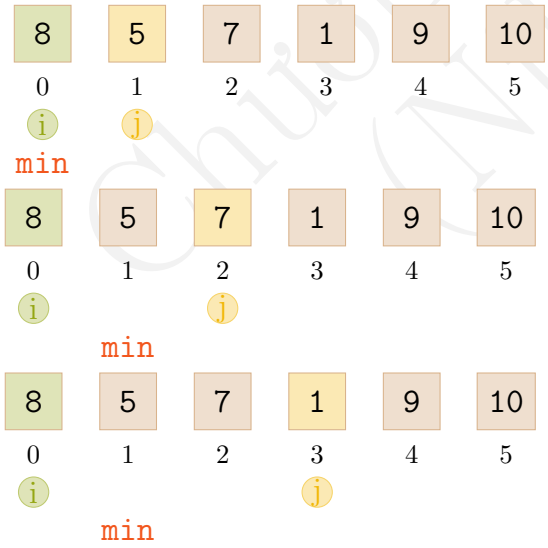
Giải thích

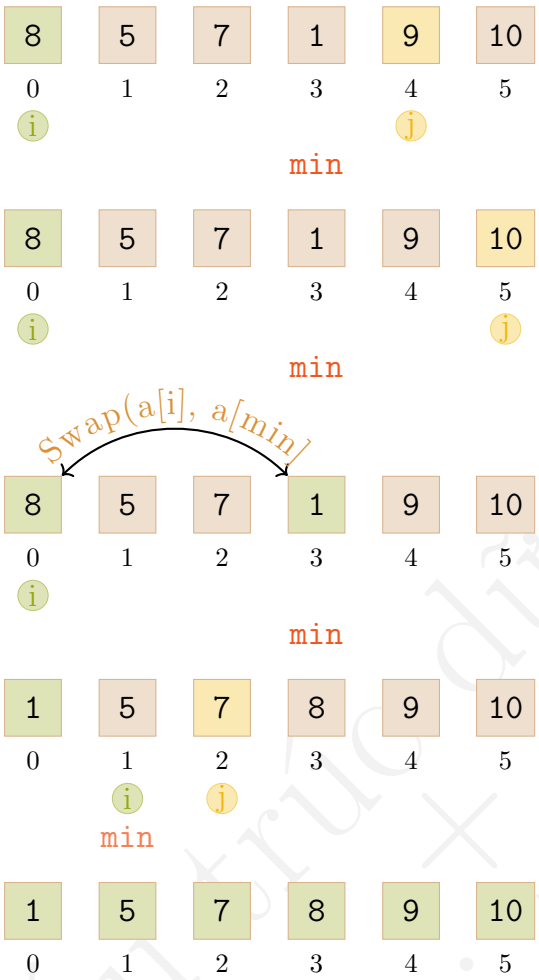
- *Dòng 2: vị trí phần tử có giá trị nhỏ nhất trong mảng chưa có thứ tự.*
- *Dòng 6: gọi hàm hoán vị 2 giá trị.*

Ví dụ 4.2.1. *Cho dãy số a gồm 6 phần tử:* $8, 5, 7, 1, 9, 10$. *Áp dụng giải thuật sắp xếp chọn trực tiếp sắp dãy a theo thứ tự tăng dần.*



Thuật toán sẽ thực hiện theo các bước sau đây:





Độ phức tạp của thuật toán

- Số phép so sánh

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

- Mỗi lần hoán vị thực hiện 3 phép gán.

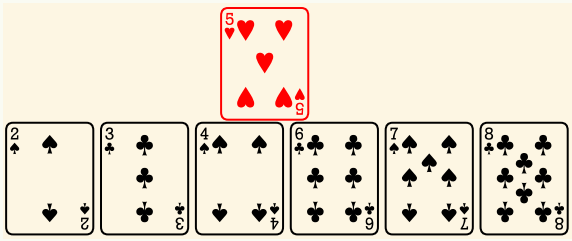
Trường hợp	Số phép so sánh	Số hoán vị
Tốt nhất	$\frac{n(n-1)}{2}$	$n-1$
Xấu nhất	$\frac{n(n-1)}{2}$	$n-1$
Độ phức tạp thời gian	$T(n) = O\left(n^2\right)$	

4.3 Sắp xếp chèn trực tiếp

Ý tưởng

Cho dãy $a = \{a_0, a_1, ..., a_{n-1}\}$ gồm n phần tử, giải thuật là một vòng lặp thực hiện $n-1$ lần.
Giả sử $a_0, a_1, ..., a_{i-1}$ đã có thứ tự, tại mỗi lần lặp thứ $i = 1, 2, ..., n-1$

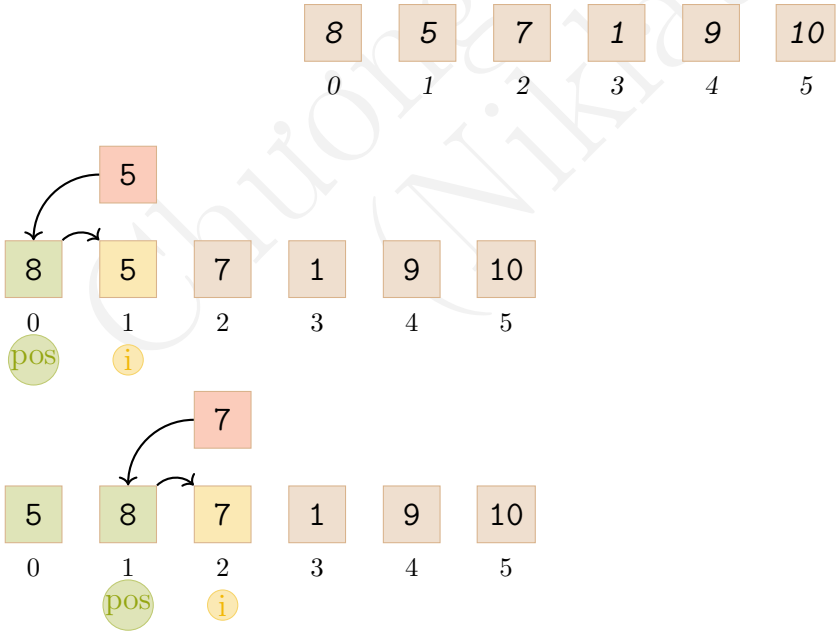
- Chèn phần tử a_i vào đúng vị trí trong dãy $a_0, a_1, ..., a_{i-1}$ để được dãy $a_0, a_1, ..., a_{i-1}, a_i$ có thứ tự.

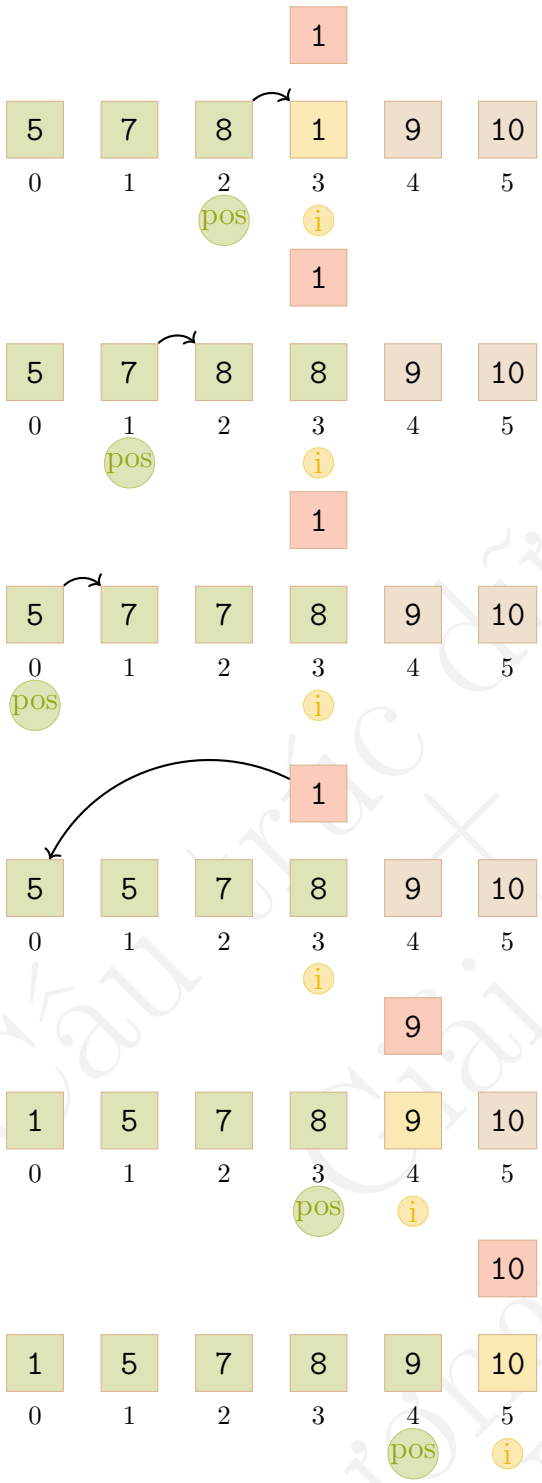


Thuật toán 4.2: InsertionSort(a[], n)
- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: mảng a có thứ tự tăng dần.

```
1    for i ← 1 to n - 1
2        x ← a[i]
3        pos ← i - 1
4        while pos ≥ 0 and a[pos] > x
5            a[pos + 1] ← a[pos]
6            pos ← pos - 1
7            a[pos + 1] ← x
```

Ví dụ 4.3.1. *Cho dãy số a gồm 6 phần tử:* $8, 5, 7, 1, 9, 10$. *Áp dụng giải thuật sắp xếp chèn trực tiếp sắp dãy a theo thứ tự tăng dần.*





- Phép so sánh: $a[pos] > x$.
  - Trường hợp tốt nhất
    $\sum_{i=1}^{n-1} 1 = n - 1.$
  - Trường hợp xấu nhất
    $\sum_{i=1}^{n-1} \sum_{pos=0}^{i-1} 1 = \sum_{i=1}^{n-1} (i-1) = \frac{n(n-1)}{2}.$
- Phép gán: $x = a[i]$, $a[pos+1] = a[pos]$, $a[pos+1] = x$.

– Trường hợp tốt nhất
$\sum_{i=1}^{n-1} 2 = 2\,(n-1).$

– Trường hợp xấu nhất
$\sum_{i=1}^{n-1}\left(\left(\sum_{pos=0}^{i-1} 1\right)+2\right) = \frac{n\,(n-1)}{2} + 2\,(n-1) = \frac{(n-1)\,(n+4)}{2}.$

Trường hợp	Số phép so sánh	Số phép gán
Tốt nhất	$n-1$	$2\,(n-1)$
Xấu nhất	$\frac{n(n-1)}{2}$	$\frac{(n-1)(n+4)}{2}$
Độ phức tạp thời gian	$T\,(n) = O\left(n^2\right)$	

4.4 Sắp xếp đổi chỗ trực tiếp

Cho dãy $a = \{a_0, a_1, ..., a_{n-1}\}$ gồm n phần tử, giải thuật là một vòng lặp thực hiện $n-1$ lần.

Tại mỗi lần lặp thứ $i = 0, 2, ..., n-2$

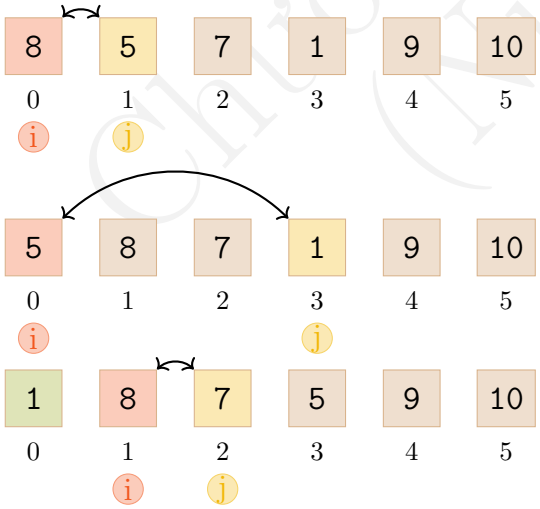
- Tìm tất cả nghịch thế chứa phần tử a_i.
- Đổi chỗ/hoán vị phần tử a_i và phần tử tương ứng trong nghịch thế.

Thuật toán 4.3: `InterchangeSort(a[], n)`
- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: mảng a có thứ tự tăng dần.

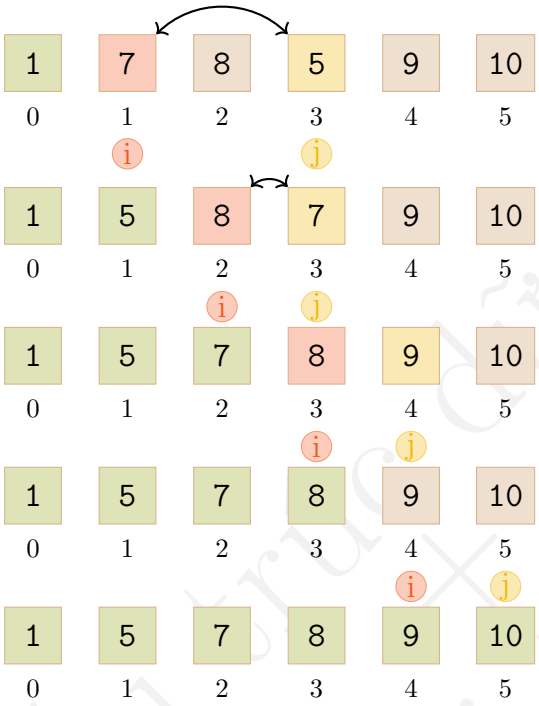
```
1    for i ← 0 to n - 2
2        for j ← i + 1 to n - 1
3            if a[j] < a[i]
4                Swap(a[i], a[j])
```

Giải thích

- *Dòng 3: kiểm tra nghịch thế giữa a[i] và a[j].*



Trường hợp	Số phép so sánh	Số hoán vị
Tốt nhất	$\frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
Độ phức tạp thời gian	$T(n) = O\left(n^2\right)$	



- Số phép so sánh

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

- Mỗi lần hoán vị thực hiện 3 phép gán.

4.5 Sắp xếp nổi bọt

Ý tưởng

Cho dãy $a = \{a_0, a_1, ..., a_{n-1}\}$ gồm n phần tử, giải thuật là một vòng lặp thực hiện $n-1$ lần.
Tại mỗi lần lặp thứ $i = 0, 1, ..., n-2$

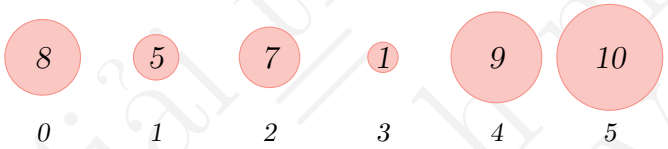
- Bắt đầu từ cuối/đầu dãy, tìm nghịch thế giữa a_j và a_{j-1}, với $j = n-1, n-2, ..., 1$.
- Hoán vị hai phần tử trong nghịch thế này.

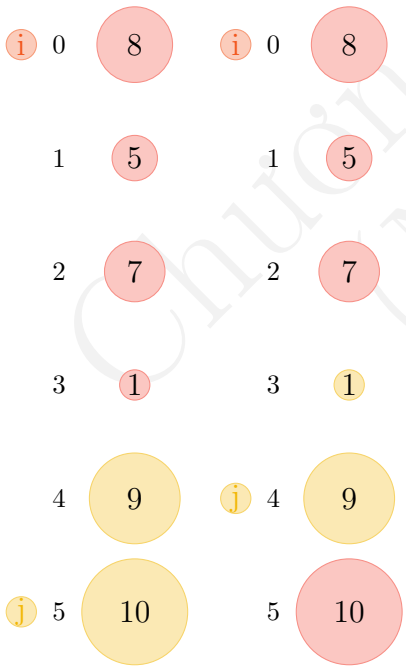
Phần tử nhỏ (nhẹ) sẽ nổi lên trên và phần tử lớn (nặng) sẽ chìm xuống đáy.

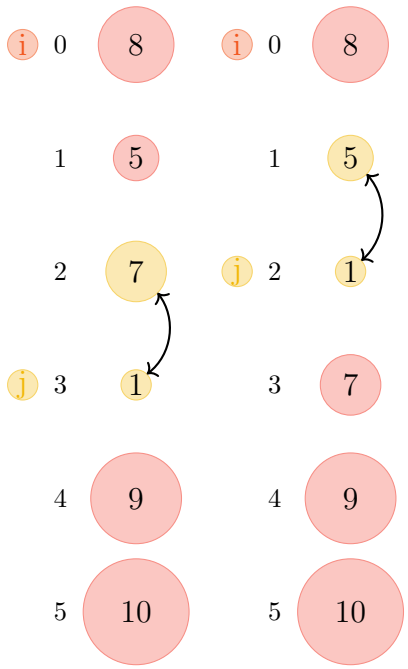
Thuật toán 4.4: BubbleSort(a[], n)
- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: mảng a có thứ tự tăng dần.

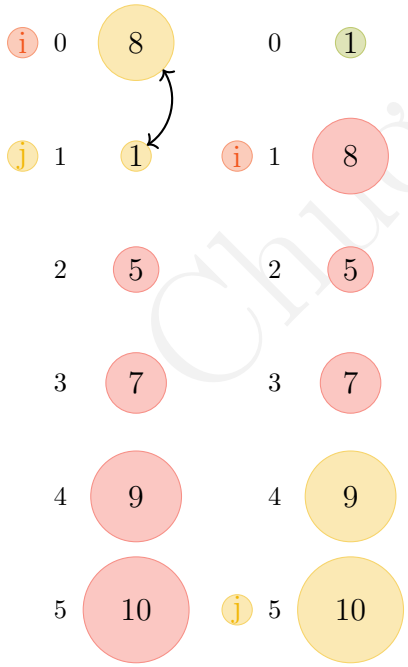
```
1    for i ← 0 to n - 2
2        for j ← n - 1 downto i
3            if a[j] < a[j - 1]
4                Swap(a[j], a[j - 1])
```

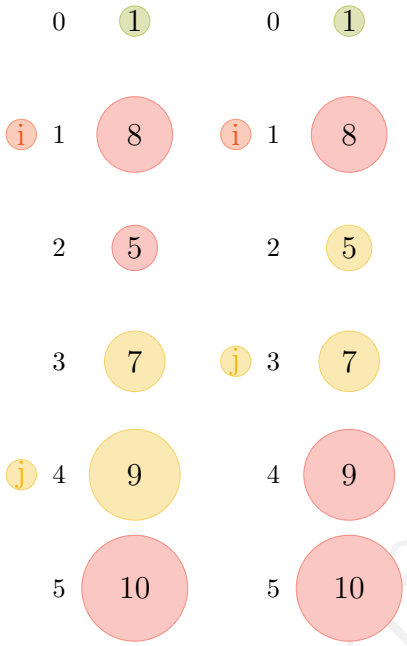
Ví dụ 4.5.1. *Cho dãy số a gồm 6 phần tử:* $8, 5, 7, 1, 9, 10$. *Áp dụng giải thuật sắp xếp nổi bọt sắp dãy a theo thứ tự tăng dần.*

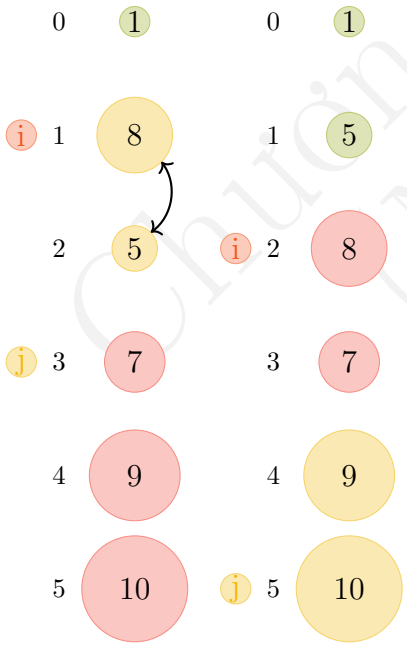


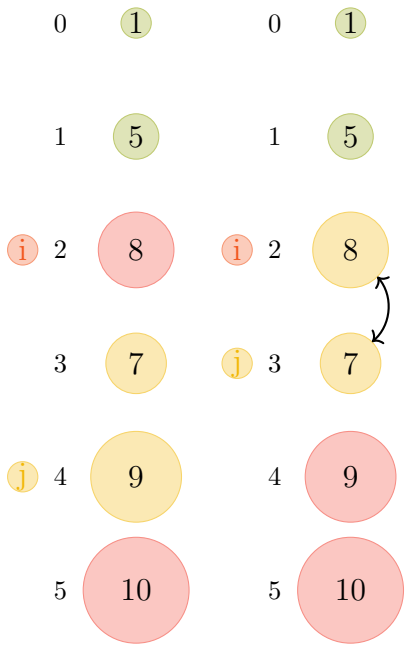


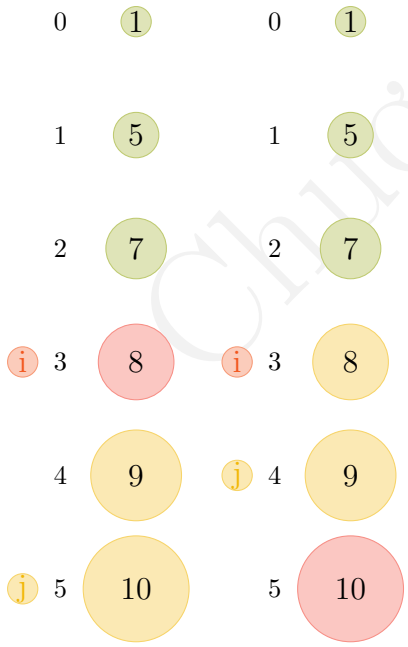


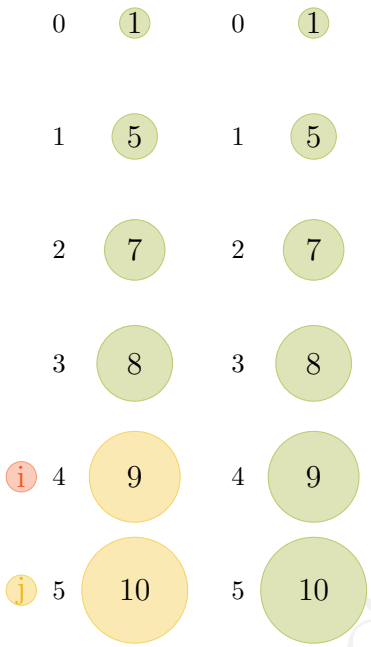












Nhận xét

- Không nhận biết được dãy đã có thứ tự hay có thứ tự từng phần.

- Các phần tử nhỏ được đưa về đúng vị trí rất nhanh, nhưng các phần tử lớn thì rất chậm.

- Số phép so sánh

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

- Mỗi lần hoán vị thực hiện 3 phép gán.

Trường hợp	Số phép so sánh	Số hoán vị
Tốt nhất	$\frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
Độ phức tạp thời gian	$T(n) = O\left(n^2\right)$	

4.6 Sắp xếp rung

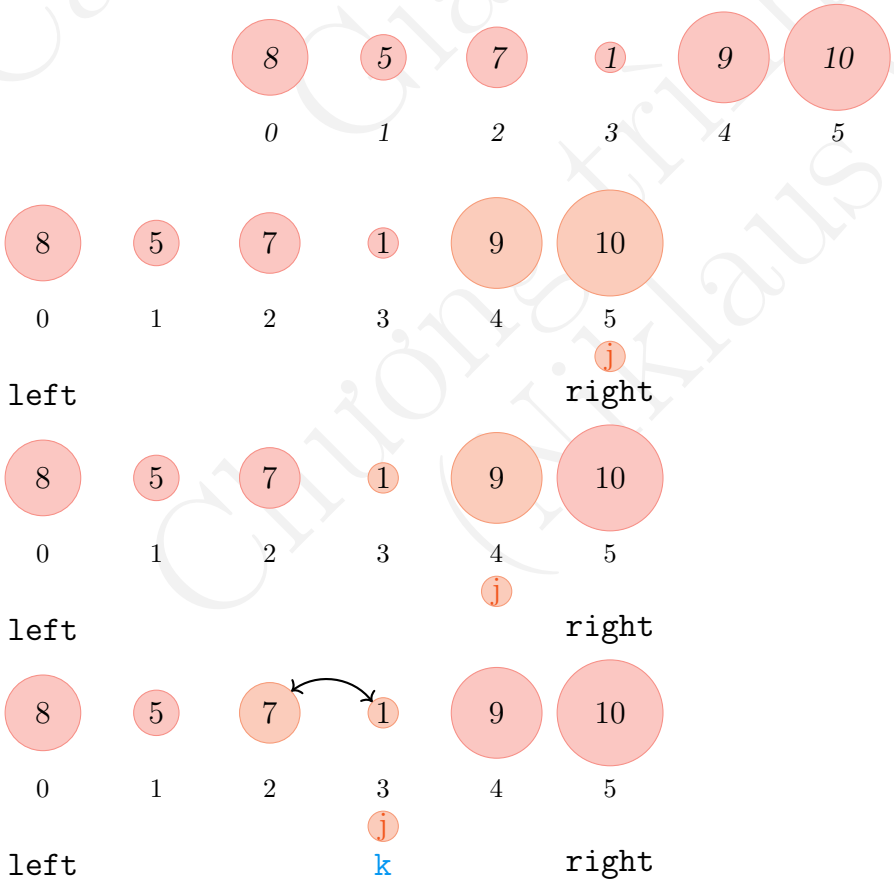
Ý tưởng

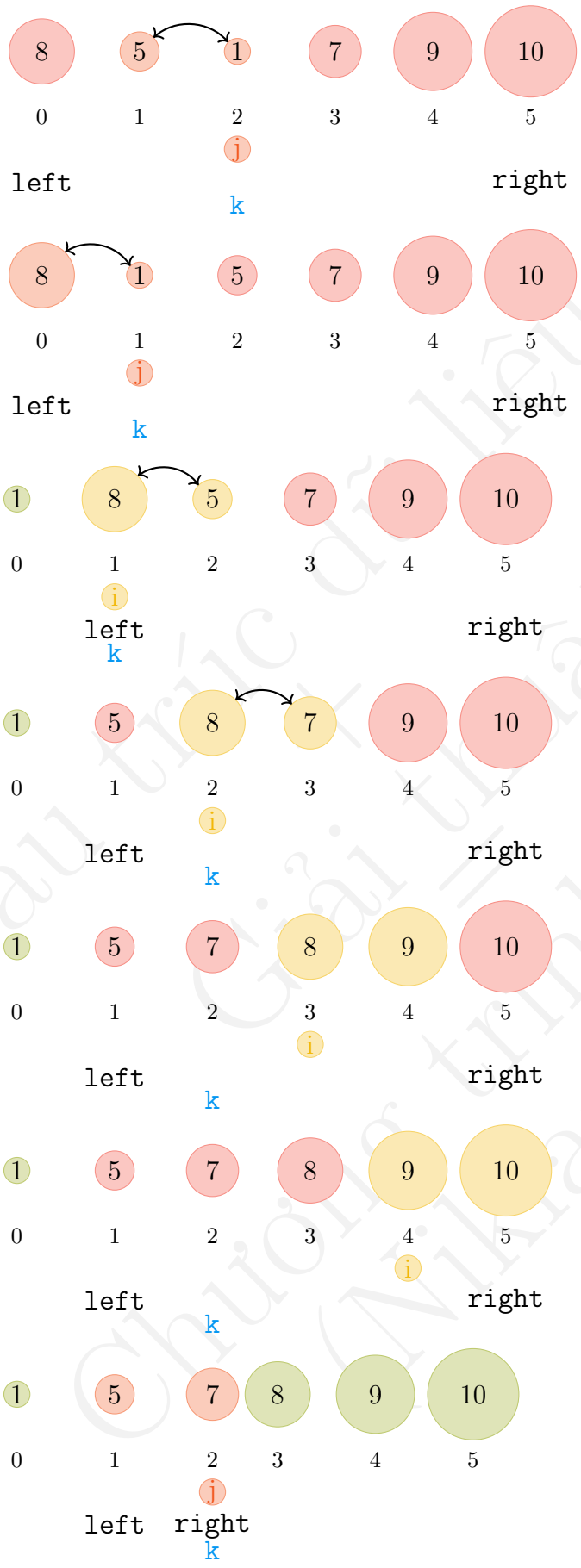
Tương tự như sắp xếp nổi bọt, nhưng khắc phục được khuyết điểm của giải thuật này bằng cách

- Lượt đi: xuất phát từ cuối dãy không thứ tự, đẩy phần tử nhỏ nhất về đầu dãy.
- Lượt về: xuất phát từ đầu dãy không thứ tự, đẩy phần tử lớn nhất về cuối dãy.
- Ghi nhận lại vị trí những đoạn đã có thứ tự và tiếp tục thực hiện thuật toán đối với dãy chưa có thứ tự (đoạn màu xanh dương/in đậm trong hình)

Ví dụ 4.6.1. *Cho dãy số a gồm 6 phần tử:* $8, 5, 7, 1, 9, 10$. *Áp dụng giải thuật sắp xếp rung sắp dãy a theo thứ tự tăng dần.*

Thuật toán thực hiện qua các bước sau:





- Trong trường hợp dãy số có thứ tự bộ phận thuật toán sẽ thực hiện số phép so sánh ít hơn Bubble Sort.

- Số phép hoán vị tương tự giải thuật Bubble Sort.

Trường hợp	Số phép so sánh	Số hoán vị
Tốt nhất	$n-1$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$
Độ phức tạp thời gian	$T(n) = O\left(n^2\right)$	

4.7 Bài tập cuối chương

1. Cho dãy ký tự `X, I, N, C, H, A, O` , hãy áp dụng các thuật toán đã học sắp dãy theo đúng thứ tự bảng chữ cái Latinh.
2. Trong các phương pháp sắp xếp: SelectionSort, InsertionSort, InterchangeSort và BubbleSort, phương pháp nào thực hiện sắp xếp nhanh nhất với một dãy đã có thứ tự? Giải thích.
3. Cho ví dụ minh họa ưu điểm của Shaker Sort so với Bubble Sort khi sắp xếp một dãy số.
4. Cho dãy số gồm n phần tử, hãy thiết kế thuật toán kiểm tra dãy có tồn tại hai số có tổng là một giá trị s cho trước hay không? Ví dụ, nếu dãy số gồm: 8, 5, 4, 1, 6, 3 và s là 10, thì thuật toán trả về true ($4 + 6 = 10$), ngược lại trả về false.
5. Bài toán di chuyển đĩa
   - Cho một dãy đĩa gồm 2 màu sáng và tối

     
   - Xây dựng giải thuật di chuyển các đĩa sáng về bên trái và đĩa tối về bên phải. Mỗi bước chỉ được hoán vị hai đĩa kề nhau.

     
   - Đếm số hoán vị hai đĩa khi thực hiện.

   Gợi ý: xem dãy đĩa như một dãy số, đĩa màu sáng có giá trị là 0 và đĩa tối có giá trị là 1.

Bài 5

CÁC THUẬT TOÁN SẮP XẾP NÂNG CAO (6 tiết)

5.1 Giới thiệu kỹ thuật Chia để trị

Chia để trị

Chia để trị (divide and conquer) là một phương pháp chia bài toán ban đầu thành nhiều bài toán con và quá trình này tiếp tục cho đến khi giải được bài toán con và kết quả sẽ được tổng hợp lại.
Các bước thực hiện:

- Chia bài toán: chia bài toán thành những bài toán con nhỏ hơn.
- Trị bài toán nhỏ: giải những bài toán con này.
- Tổng hợp: kết hợp lời giải của những bài toán con thành lời giải cho bài toán ban đầu.

Một số bài toán áp dụng kỹ thuật chia để trị: tìm kiếm nhị phân *(Binary-Search)*, sắp xếp nhanh *(QuickSort)*, sắp xếp trộn *(MergeSort)*, ⋯

5.2 Thuật toán sắp xếp nhanh (Quick Sort)

5.2.1 Ý tưởng

Ý tưởng

Sắp xếp nhanh *(Quick sort)* dựa vào ý tưởng chọn một phần tử chốt *(pivot)* $x = a_i$ trong dãy a để phân hoạch/chia dãy thành 2 dãy con

- Dãy bên trái: các phần tử nhỏ hơn x

$$a_0, a_1, \cdots, a_{i-1}$$

- Dãy bên phải: các phần tử lớn hơn x

$$a_{i+1}, a_{i+2}, \cdots, a_{n-1}$$

Tiếp tục thực hiện giải thuật đối với hai dãy con cho đến khi dãy con chỉ còn một phần tử *(xem như có thứ tự)*.

5.2.2 Thuật toán

Phương pháp đệ quy

Thuật toán 5.1: QuickSort(a[], left, right)
- Đầu vào: mảng a, vị trí left và right của mảng con đang xét.
- Đầu ra: mảng a có thứ tự tăng dần.

```
1  x ← a[(left + right) / 2]
2  i ← left
3  j ← right
4
5  Partition(a, i, j, x)
6  if left < j
7      QuickSort(a, left, j)
8  if i < right
9      QuickSort(a, i, right)
```

Giải thuật phân hoạch *(Partition)*

- Nếu dãy số còn phần tử chưa xét
  - Mỗi bước thứ i tìm phần tử lớn hơn x ở dãy con bên trái *(thực hiện từ đầu đến cuối dãy con)*.
  - Mỗi bước thứ j tìm phần tử nhỏ hơn x ở dãy con bên phải *(thực hiện từ cuối về đầu dãy con)*.
  - Hoán vị hai phần tử này.
- Sau khi phân hoạch, dãy a gồm các dãy con sau:
  - Dãy con 1: $a_{left}, \ldots, a_i < x$.
  - Dãy con 2: $a_{i+1}, \ldots, a_{j-1} = x$.

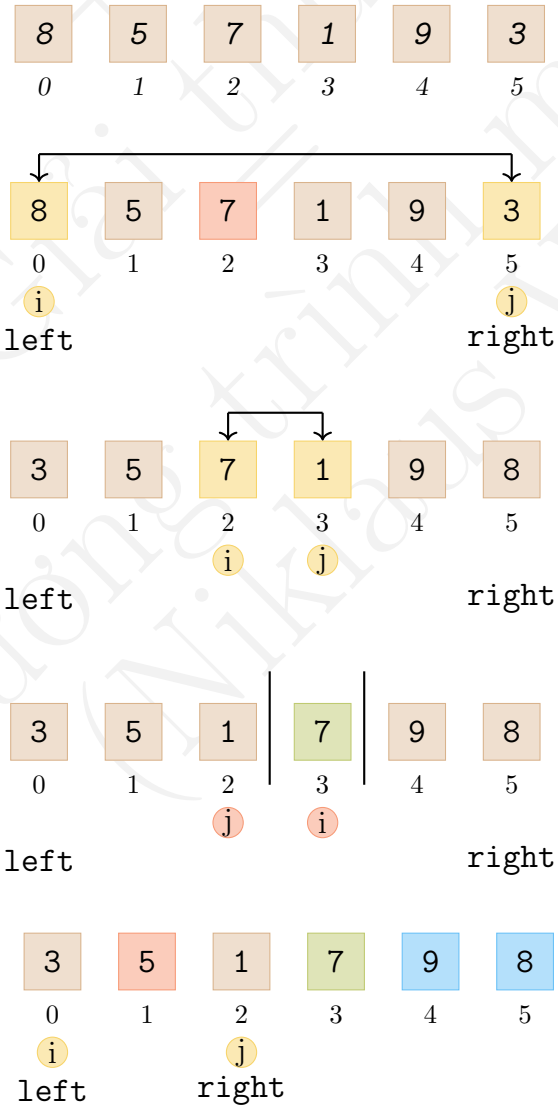
– Dãy con 3: $a_j, \ldots, a_{right} > x$.

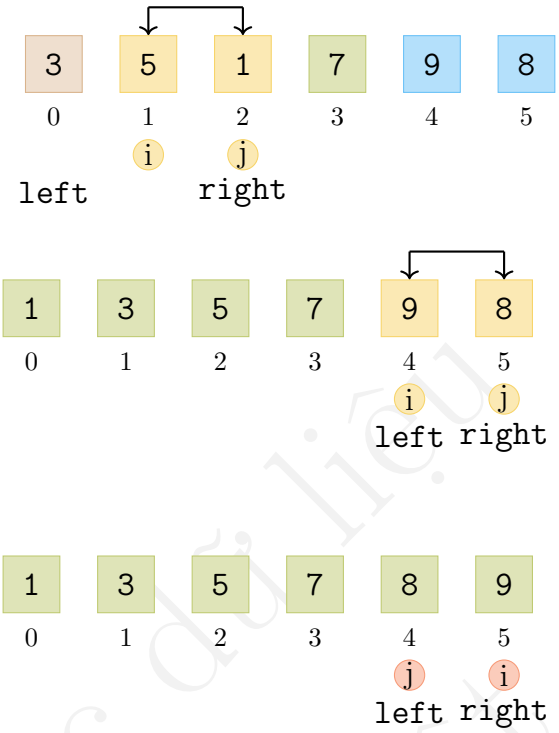
Thuật toán 5.2: `Partition(a[], i, j, x)`

- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: mảng a sau khi phân hoạch.

```
1     do
2     while a[i] < x
3         i ← i + 1
4     while a[j] > x
5         j ← j - 1
6     if i ≤ j
7         Swap(a[i], a[j])
8         i ← i + 1
9         j ← j - 1
10    while i < j
```

Ví dụ 5.2.1. *Cho dãy số a gồm 7 phần tử:* $8, 5, 7, 1, 9, 10$. *Áp dụng giải thuật sắp xếp nhanh (Quick Sort) sắp dãy a theo thứ tự tăng dần.*





Vấn đề chọn phần tử chốt

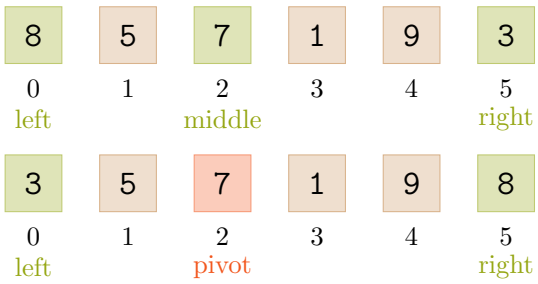
- Chọn phần tử trung vị của 3 phần tử trái, giữa, phải *(median of three)*
  - So sánh 3 phần tử: trái, giữa, phải của dãy.
  - Hoán vị các phần tử sao cho:
    - a_{left} = smallest
    - a_{middle} = median of three
    - a_{right} = largest
  - Chọn $median = a_{middle}$ là phần tử chốt.

Chọn phần tử trung vị của 3 phần tử

Thuật toán 5.3: `Median3(a[], left, right)`
- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: trả về phần tử trung vị của mảng a.

```
1     middle = (left + right) / 2
2
3     if a[left] > a[middle]
4         Swap(a[left], a[middle])
5     if a[left] > a[right]
6         Swap(a[left], a[right])
7     if a[middle] > a[right]
8         Swap(a[middle], a[right])
9
10    return a[middle]
```

Ví dụ 5.2.2. *Cho dãy số a gồm 7 phần tử:* $8, 5, 7, 1, 9, 10$. *Chọn phần tử pivot là trung vị của 3 phần tử: trái, giữa và phải.*



5.2.3 Độ phức tạp của thuật toán

- **Trường hợp tốt nhất**
  - Xảy ra khi mỗi lần phân hoạch chia dãy thành 2 phần bằng nhau.

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

trong đó,

* $2T\left(\frac{n}{2}\right)$: thời gian sắp thứ tự *2 dãy con*.
* $n-1$: số phép so sánh giữa x và $n-1$ phần tử khác.

Độ phức tạp thời gian

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

- **Trường hợp xấu nhất**
  - Xảy ra khi dãy đã có thứ tự. Mỗi lần phân hoạch chia dãy đang xét thành 2 dãy con
    * Một dãy gồm 1 phần tử.
    * Một dãy gồm $n-1$ phần tử.

$$T(n) = \begin{cases} 0 & , n = 1 \\ T(n-1) + (n-1) & , n > 1 \end{cases}$$

trong đó,

* $T(n-1)$: thời gian sắp thứ tự *1 dãy con*.
* $n-1$: số phép so sánh giữa x và $n-1$ phần tử khác.

Độ phức tạp thời gian

$$T(n) = O\left(n^2\right).$$

- **Trường hợp trung bình**
  - Xảy ra khi dãy đã có thứ tự. Mỗi lần phân hoạch chia dãy đang xét thành 2 dãy con khác rỗng.

$$T(n) = \begin{cases} 0 & , n \leq 1 \\ \frac{1}{n}\sum_{i=0}^{n-1}\left[T(i) + T(n-1-i)\right] + (n-1) & , n > 1 \end{cases}$$

trong đó,

* $T(i)$: thời gian sắp thứ tự dãy con thứ nhất.
* $T(n-1-i)$: thời gian sắp thứ tự dãy con thứ hai.

– Do đó,

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

5.3 Thuật toán sắp xếp trộn (MergeSort)

5.3.1 Ý tưởng

Ý tưởng

- Trong giải thuật sắp xếp trộn *(MergeSort)*, mỗi dãy $a_0, a_2, \cdots, a_{n-1}$ bất kỳ đều có thể coi như là một tập hợp các dãy con liên tiếp mà mỗi dãy con đều đã có thứ tự.
- Các dãy con này trộn với nhau sẽ tạo thành dãy có thứ tự.

Ví dụ 5.3.1. *Cho dãy số* $12, 2, 8, 5, 1, 6, 4, 15$ *gồm 5 dãy con có thứ tự:*

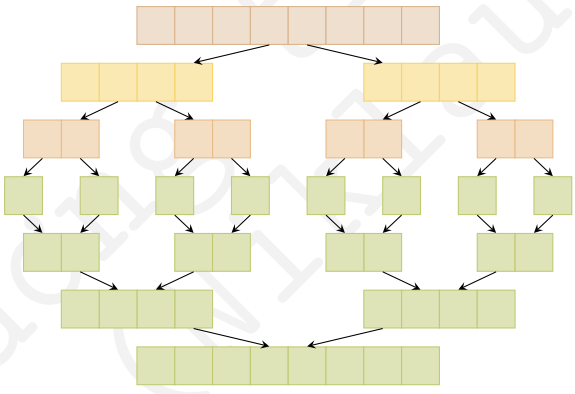
$$\{12\}, \{2, 8\}, \{5\}, \{1, 6\}, \{4, 15\}$$

Trộn các dãy con tạo thành dãy có thứ tự.

$$1, 2, 4, 5, 8, 6, 12, 15$$

Áp dụng phương pháp chia để trị

- Chia: chia dãy a gồm n phần tử thành hai dãy con có kích thước khoảng $\frac{n}{2}$.
- Trị: nếu dãy con chứa từ hai phần tử trở lên thì gọi đệ quy giải thuật MergeSort đối với dãy con đang xét.
- Tổng hợp: trộn hai dãy con với nhau thành một dãy có thứ tự.



5.3.2 Thuật toán

Thuật toán 5.4: `MergeSort(a[], left, right)`

```
- Đầu vào: mảng a, left và right tương ứng vị trí bắt đầu và kết
thúc của mảng đang xét.
- Đầu ra: mảng a có thứ tự tăng dần.
```

```
1     if left < right
2         middle ← (left + right) / 2
3     MergeSort(a, left, middle)
4     MergeSort(a, middle + 1, right)
5     Merge(a, left, middle, right)
```

Giải thích

- Dòng 1: nếu mảng nhiều hơn một phần tử thì chia thành 2 mảng con bên trái và bên phải tại phần tử giữa.
- Dòng 3, 4: gọi đệ quy giải thuật với mảng con bên trái và bên phải.
- Dòng 5: kết hợp 2 mảng con thành mảng có thứ tự.

Thuật toán 5.5: `Merge(a[], left, middle, right)`

- Đầu vào: mảng a, vị trí left, middle, right.
- Đầu ra: mảng a sau khi được trộn.

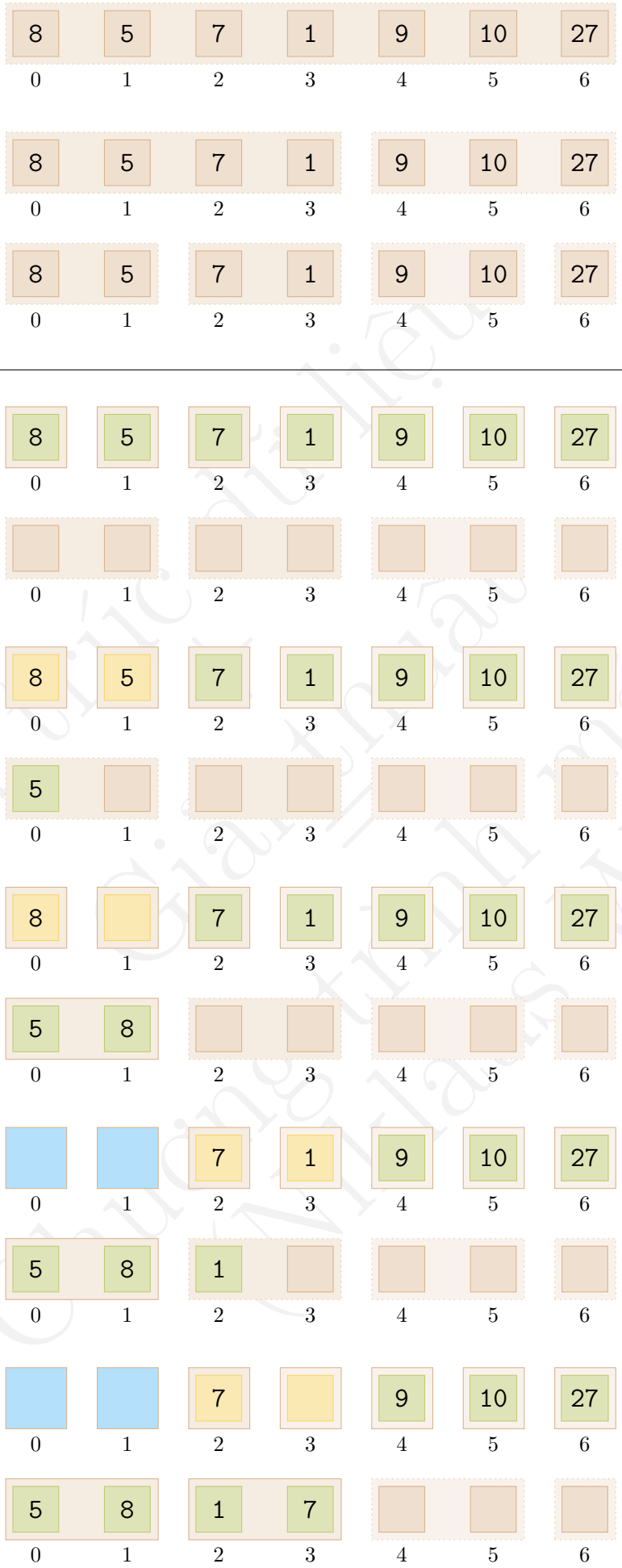
```
1     array b[left..right]
2     k ← left, i ← left, j ← middle + 1
3     while i ≤ middle or j ≤ right
4         if a[i] ≤ a[j]
5             b[k++] ← a[i++]
6         else
7             b[k++] ← a[j++]
8     while i ≤ middle
9         b[k++] ← a[i++]
10    while j ≤ right
11        b[k++] ← a[j++]
12    for k ← left to right
13        a[k] ← b[k]
```

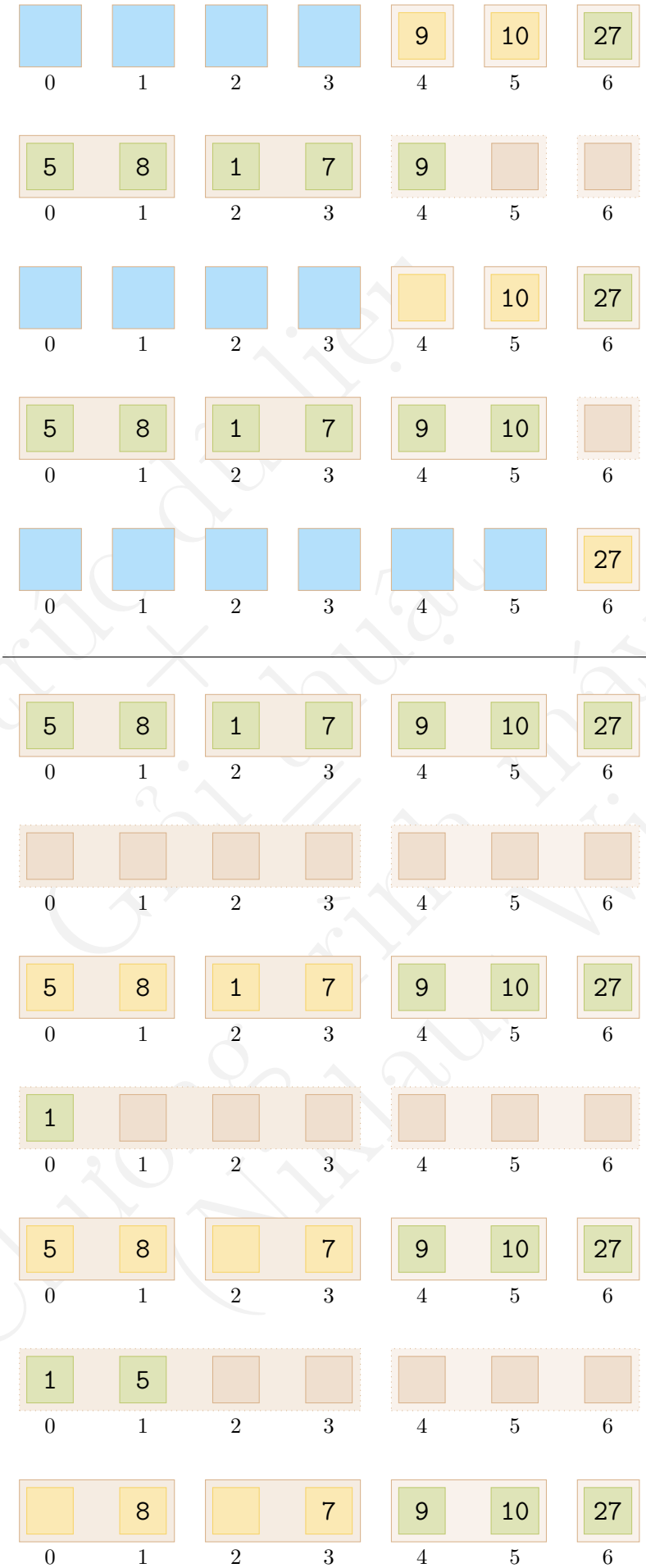
Giải thích

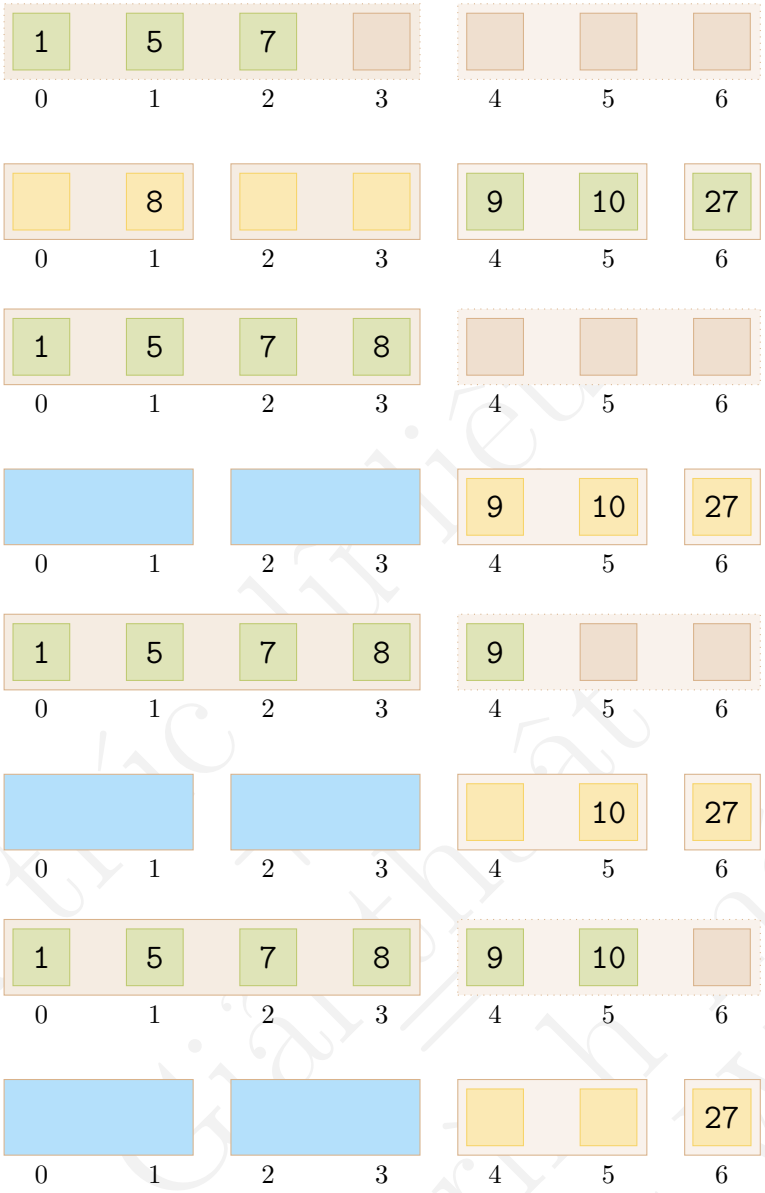
- Dòng 1: khai báo `b` là mảng tạm.
- Dòng 3 → 7: trường hợp hai mảng con bên trái và bên phải đều khác rỗng.
  - Dòng 4, 5: chép mảng bên trái vào `b`.
  - Dòng 6, 7: chép mảng bên phải vào `b`.
- Dòng 8, 9: chép phần còn lại của mảng bên trái vào `b`.
- Dòng 10, 11: chép phần còn lại của mảng bên phải vào `b`.
- Dòng 12, 13: chép mảng `b` vào mảng `a`. Kết thúc thuật toán.

Ví dụ 5.3.2. *Cho dãy số a gồm 7 phần tử:* $8, 5, 7, 1, 9, 10, 27$. *Áp dụng giải thuật MergeSort sắp dãy a theo thứ tự tăng dần.*

Thuật toán MergeSort được thực hiện từng bước như sau:


8 5 7 1 9 10 27
0 1 2 3 4 5 6
8 5 7 1 9 10 27
0 1 2 3 4 5 6
8 5 7 1 9 10 27
0 1 2 3 4 5 6
8 5 7 1 9 10 27
0 1 2 3 4 5 6
8 5 7 1 9 10 27
0 1 2 3 4 5 6
8 5 7 1 9 10 27
0 1 2 3 4 5 6
5
0 1 2 3 4 5 6
8 7 1 9 10 27
0 1 2 3 4 5 6
5 8
0 1 2 3 4 5 6
7 1 9 10 27
0 1 2 3 4 5 6
5 8 1
0 1 2 3 4 5 6
7 9 10 27
0 1 2 3 4 5 6
5 8 1 7
0 1 2 3 4 5 6


9 10 27
0 1 2 3 4 5 6
5 8 1 7 9
0 1 2 3 4 5 6
10 27
0 1 2 3 4 5 6
5 8 1 7 9 10
0 1 2 3 4 5 6
27
0 1 2 3 4 5 6
5 8 1 7 9 10 27
0 1 2 3 4 5 6
0 1 2 3 4 5 6
5 8 1 7 9 10 27
0 1 2 3 4 5 6
1
0 1 2 3 4 5 6
5 8 7 9 10 27
0 1 2 3 4 5 6
1 5
0 1 2 3 4 5 6
8 7 9 10 27
0 1 2 3 4 5 6


1 5 7
0 1 2 3 4 5 6
8
9 10 27
1 5 7 8
9 10 27
1 5 7 8 9
10 27
1 5 7 8 9 10
27
1 5 7 8 9 10 27
1 5 7 8 9 10 27
1 5 7 8 9 10 27
...
1 5 7 8 9 10 27

5.3.3 Độ phức tạp của thuật toán

Trường hợp xấu nhất/tốt nhất

- Hàm MergeSort: nếu dãy có hơn một phần tử thì mỗi lần thực hiện sẽ chia thành 2 dãy con có $\frac{n}{2}$ phần tử. Thời gian thực hiện của dãy con là $T\left(\frac{n}{2}\right)$.

- Hàm Merge: thời gian thực hiện là $O(n)$.

- Thời gian thực hiện thuật toán

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

- Nếu n là lũy thừa của 2 (trộn 2 dãy/2-way) thì

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

Chứng minh. Sử dụng phương pháp lặp để tìm độ phức tạp theo các bước sau:

$$\begin{aligned} T(n) &= 2T\left(\tfrac{n}{2}\right) + n \\ &= 4T\left(\tfrac{n}{4}\right) + n + n \\ &= 8T\left(\tfrac{n}{8}\right) + n + n + n \\ &\dots \\ &= nT\left(\frac{n}{2^k}\right) + k \cdot n \qquad , n \geq 2^k \end{aligned}$$

- Giả sử $n = 2^k$, thuật toán dừng đệ quy.

$$\begin{aligned} T(n) &= 2^k \cdot T(1) + k \cdot 2^k \\ &= 2^k + k \cdot 2^k \\ &= n + n \log n \end{aligned}$$

- Do đó,

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

5.4 Thuật toán sắp xếp vun đống (HeapSort)

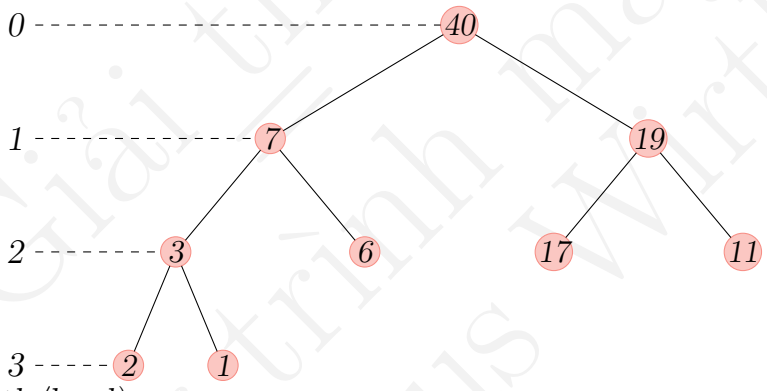
5.4.1 Các khái niệm cơ bản

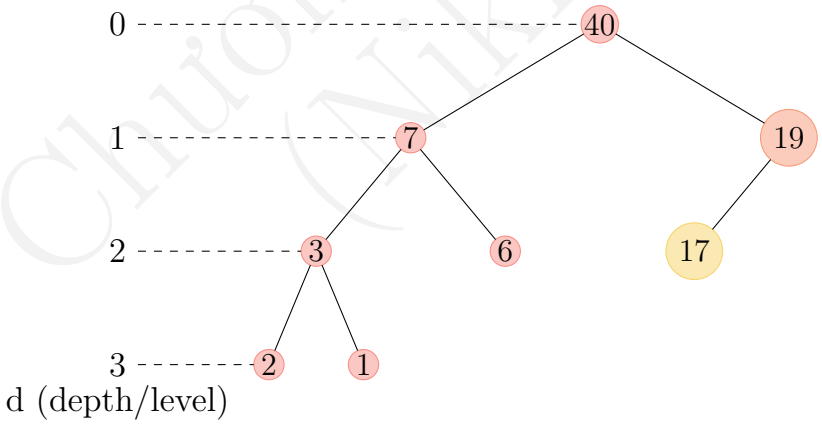
Khái niệm

Đống (heap) là một cây nhị phân đầy đủ thỏa các điều kiện sau:

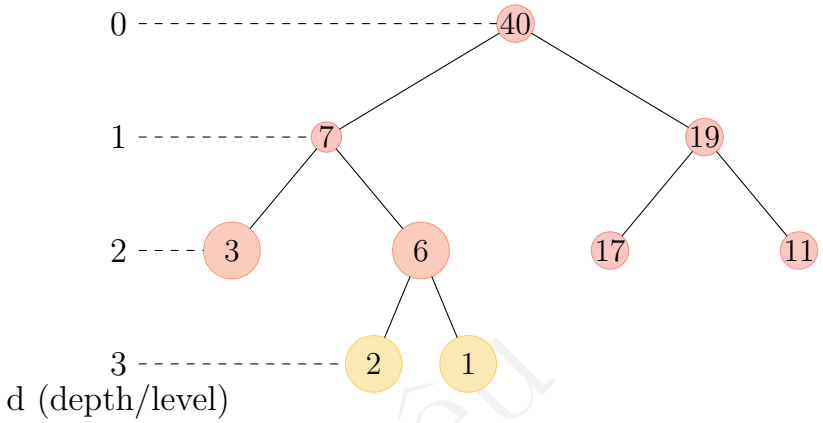
- Các nút từ mức 0 đến $d-1$ đều có đủ số lượng nút *(với d là mức của cây)*.
- Các nút ở mức d được thêm vào từ trái sang phải.
- Giá trị của một nút luôn lớn hơn hay bằng giá trị các nút con của nó (max-heap).
  Xét phần tử a_i
  - $a_i \geq a_{2i+1}$ và $a_i \geq a_{2i+2}$
  - Cặp phần tử liên đới là a_{2i+1} và a_{2i+2}.
  - Trường hợp đặt $j = 2 \cdot i + 1$ thì cặp phần tử liên đới là a_j và a_{j+1}.

Ví dụ 5.4.1.

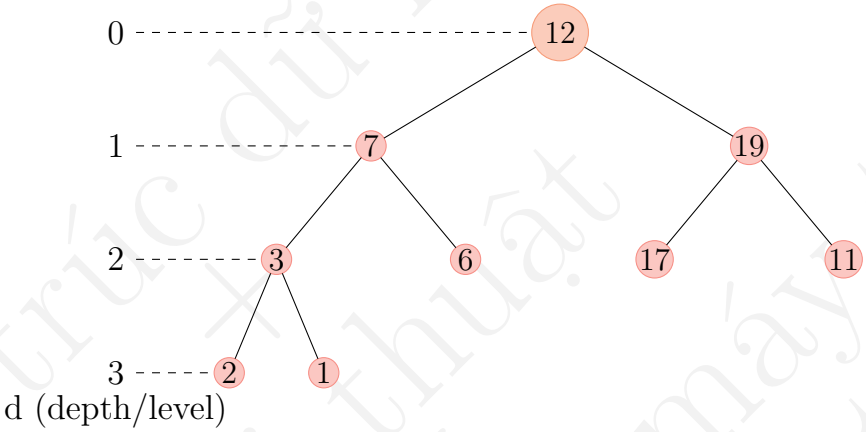




Tại mức $d-1$, số lượng nút không đảm bảo.



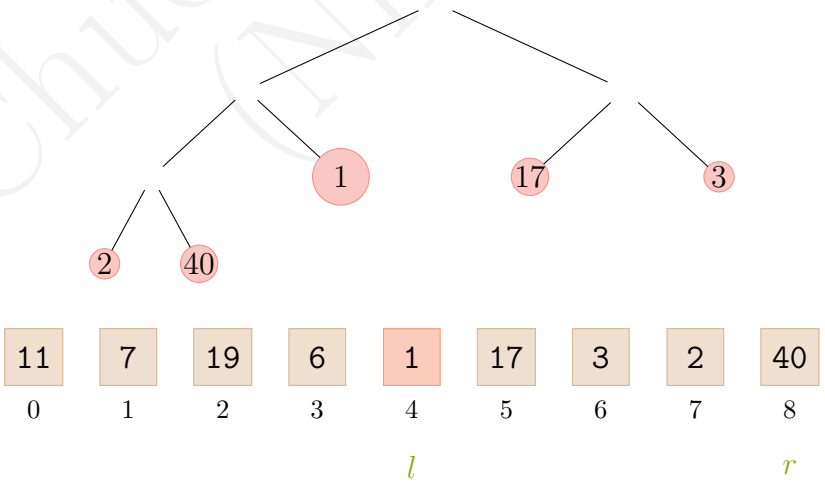
Tại mức d, các nút phải điền từ trái sang phải.



Nút cha $\geq$ nút con.

Một số tính chất của Heap

- Nếu $a_0, a_1, ..., a_r$ là một heap thì sau khi bỏ một số phần tử ở hai đầu, dãy còn lại vẫn là một heap.
- Nếu $a_0, a_1, ..., a_r$ là một heap thì phần tử a_0 có giá trị lớn nhất/nhỏ nhất *(trường hợp max-heap/min-heap)*.
- Cho dãy a gồm n phần tử. Nếu dãy $a_l, a_{l+1}, ..., a_r$ thỏa điều kiện $2l + 1 > r$ thì dãy đó là một heap tự nhiên *(với $l = \frac{n}{2}, r = n - 1$)*.



5.4.2 Ý tưởng

Ý tưởng

- Dựa vào cấu trúc heap *(đống)*, một max-heap *(đống cực đại)* thì phần tử ở **đỉnh** luôn có giá trị lớn nhất.
- HeapSort *(sắp xếp vun đống)* thực hiện qua hai giai đoạn:
  - Giai đoạn 1. Tạo (vun đống) và hiệu chỉnh dãy ban đầu thành heap.
  - Giai đoạn 2. Sắp xếp dãy dựa trên heap: thực hiện một vòng lặp
    - Lấy phần tử ở đỉnh của heap.
    - Hoán vị với phần tử cuối cùng của dãy số.
    - Hiệu chỉnh lại heap.

Tạo và hiệu chỉnh heap

- Theo tính chất 3, mọi dãy $a_l, a_{l+1}, ..., a_r$ thỏa điều kiện $2l + 1 > r$ là một heap tự nhiên (với $l = \frac{n}{2}, r = n - 1$).
- Do đó, giai đoạn tạo heap chỉ cần lần lượt ghép thêm các phần tử $a_{l-1}, a_{l-2}, \cdots a_0$ vào heap tự nhiên này.
- Mỗi lần ghép thêm một phần tử thực hiện thao tác hiệu chỉnh *(Heapify/Sift Down)* lại heap.

5.4.3 Thuật toán

Đầu tiên, Heap cần được khởi tạo từ một dãy ban đầu.

Thuật toán 5.6: `MakeHeap(a[], n)`
- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: mảng a là một max-heap.

```
1    l ← n / 2
2    l ← l - 1
3    while l ≥ 0
4        Heapify(a, l, n - 1)
5        l ← l - 1
```

Trong bước khởi tạo, thao tác hiệu chỉnh Heap sẽ được thực hiện nhằm đảm bảo các tính chất của một Heap.

Thuật toán 5.7: `Heapify(a[], l, r)`
- Đầu vào: mảng a và mảng con cần hiệu chỉnh $a_l, a_{l+1} \cdots a_r$.
- Đầu ra: mảng con đã hiệu chỉnh thành max-heap.

```
1    i ← l
2    j ← 2 * i + 1
3    x ← a[i]
4    while j <= r
5        if j < r and a[j] < a[j + 1]
6            j ← j + 1
```

```
7           if a[j] < x
8               return
9           else
10              a[i] ← a[j]
11              a[j] ← x
12              i ← j
13              j ← 2 * i + 1
```

Giải thích

- Dòng 2: hai phần tử liên đới của a_i là a_j và a_{j+1} .
- Dòng 4 → 13: nếu a_i có phần tử liên đới thì thực hiện:
  - Dòng 5, 6: tìm phần tử liên đới có giá trị lớn nhất.
  - Dòng 7, 8: nếu thỏa quan hệ liên đới. Kết thúc thao tác hiệu chỉnh.
  - Dòng 10, 11: ngược lại, hoán vị phần tử a_i với phần tử liên đới có giá trị lớn nhất.
  - Dòng 12, 13: xét hiệu chỉnh lan truyền sau khi thực hiện hoán vị.

Thuật toán HeapSort được mô tả như sau:

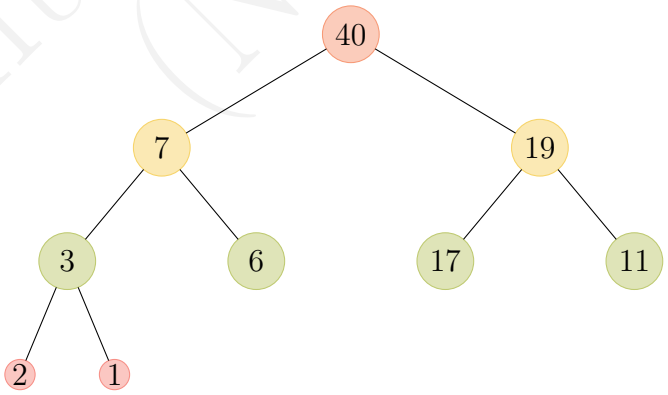
Thuật toán 5.8: HeapSort(a[], n)
- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: mảng a có thứ tự tăng dần.

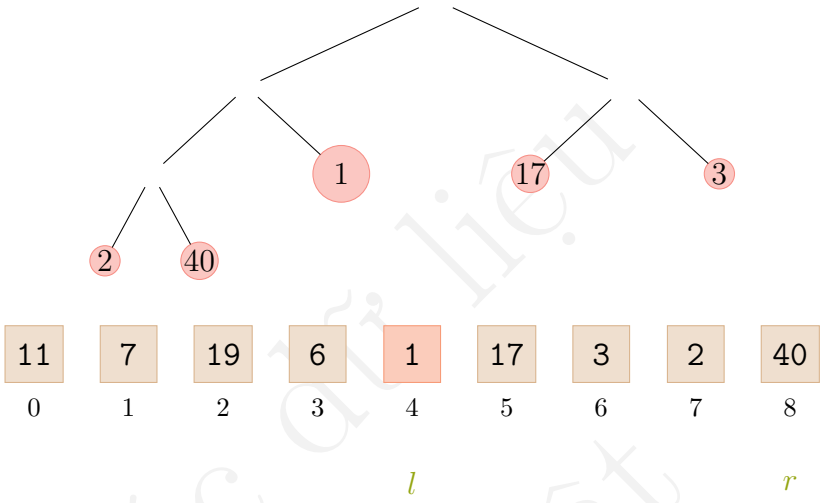
```
1   MakeHeap(a, n)
2   r ← n - 1
3   while r > 0
4       Swap(a[0], a[r])
5       r ← r - 1
6       Heapify(a, 0, r)
```

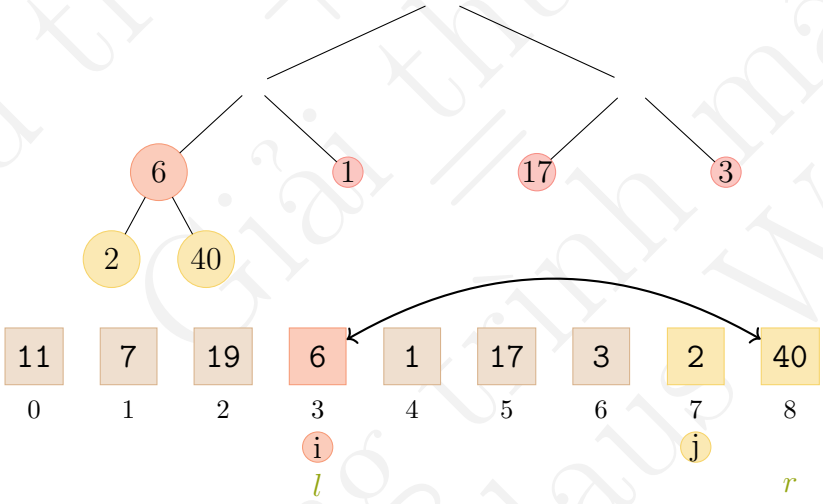
Giải thích

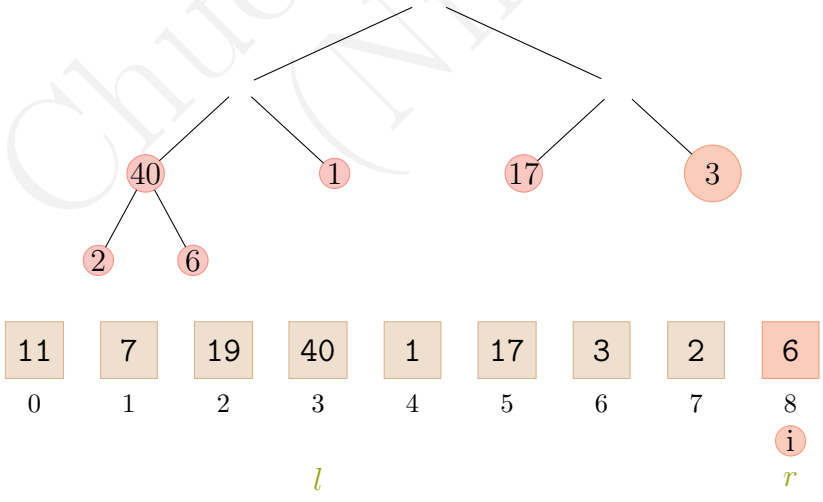
- Dòng 1: gọi hàm tạo max-heap từ mảng ban đầu.
- Dòng 3 → 6: mỗi lần lặp, hoán vị phần tử đầu và cuối mảng. Sau đó, gọi hàm hiệu chỉnh mảng tử $a_0, a_1, ..., a_{r-1}$.

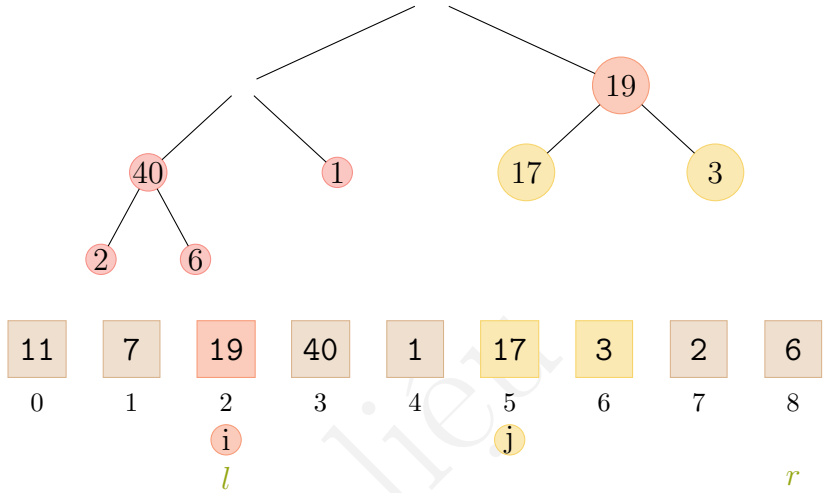
Biểu diễn heap bằng cây nhị phân đầy đủ

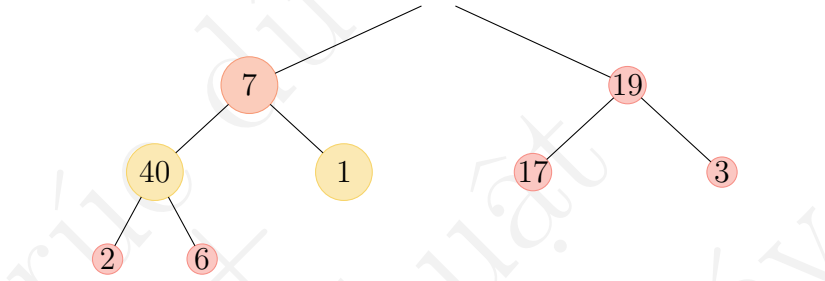


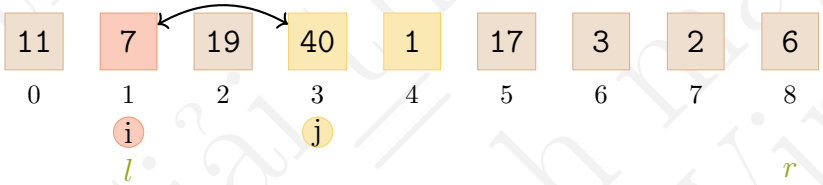


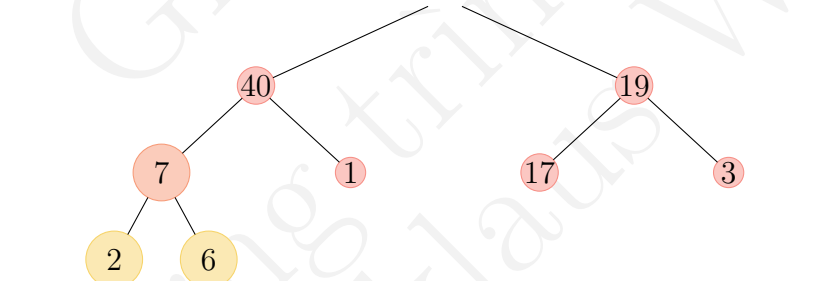


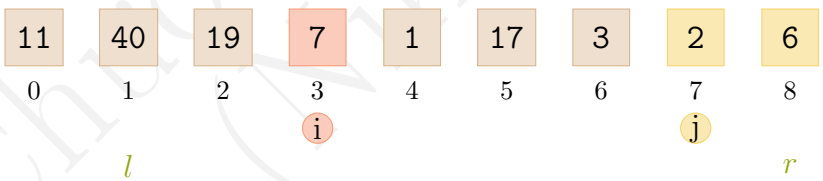


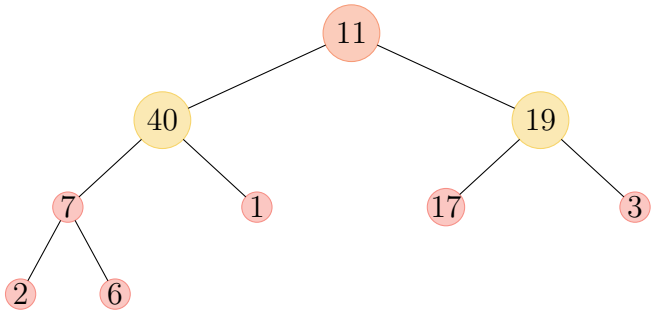


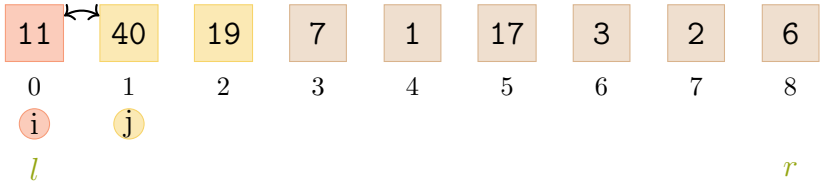


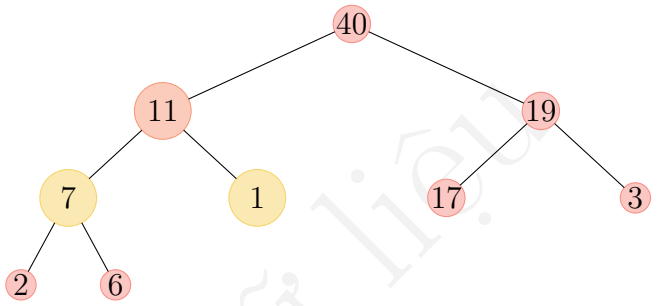


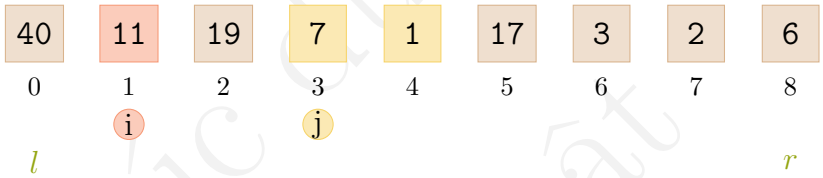


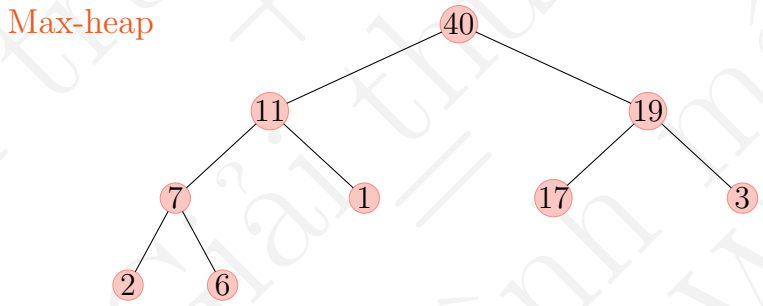




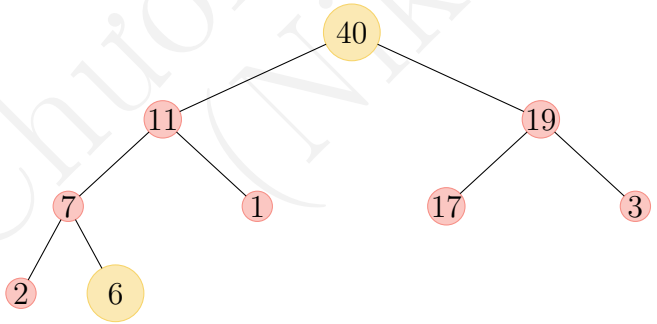


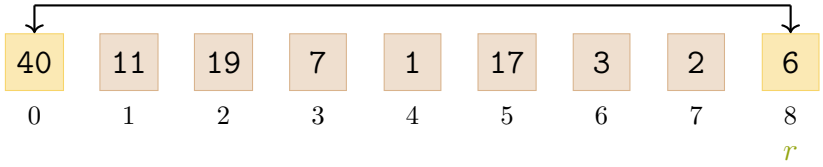


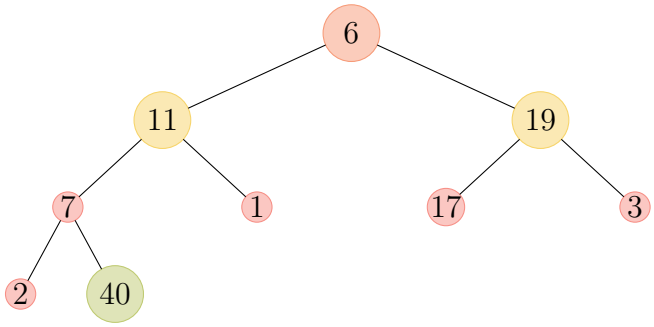


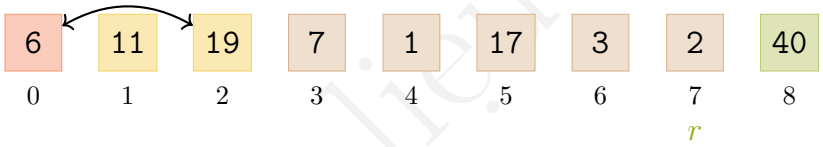


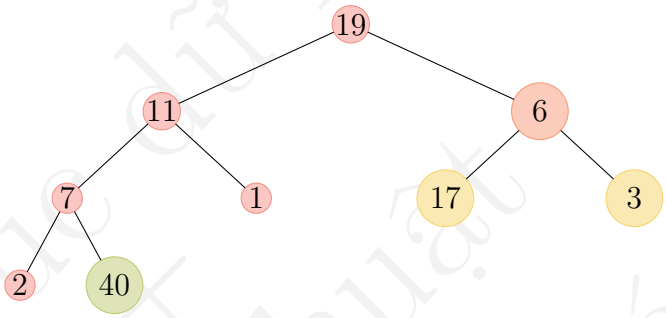
Sau khi tạo Heap, thuật toán sắp xếp sẽ được thực hiện theo các bước sau:

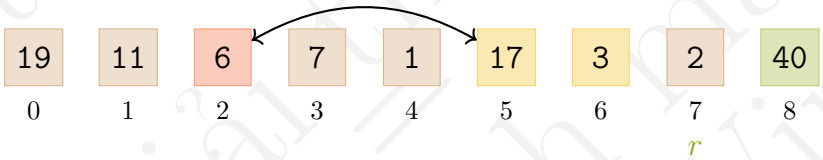


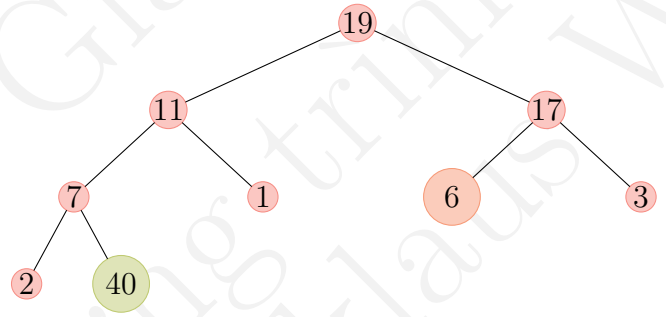


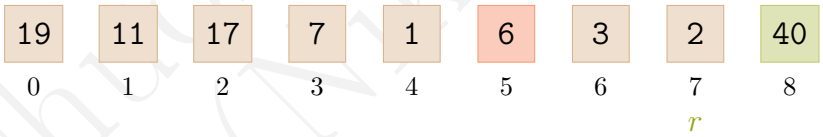


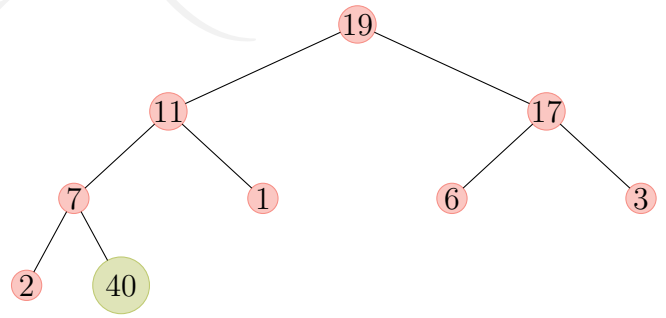


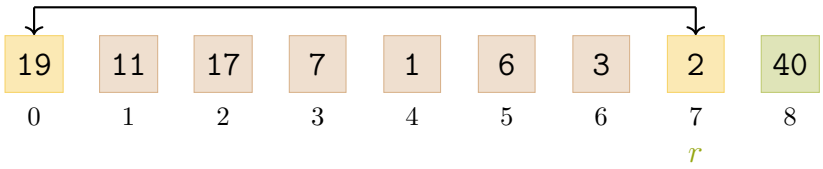


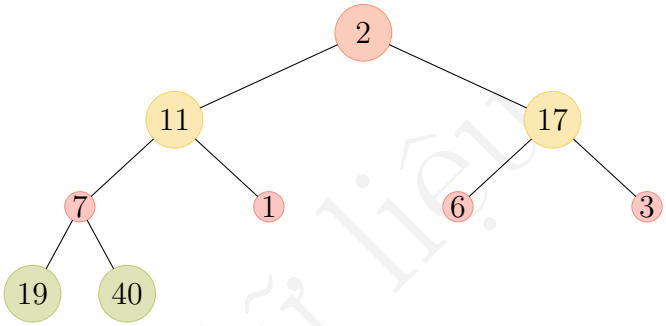


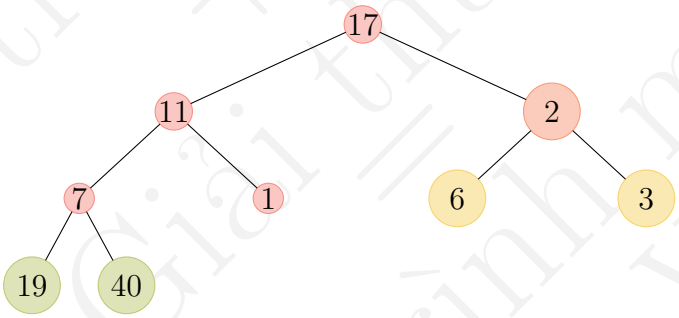


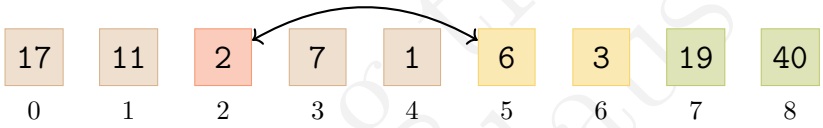


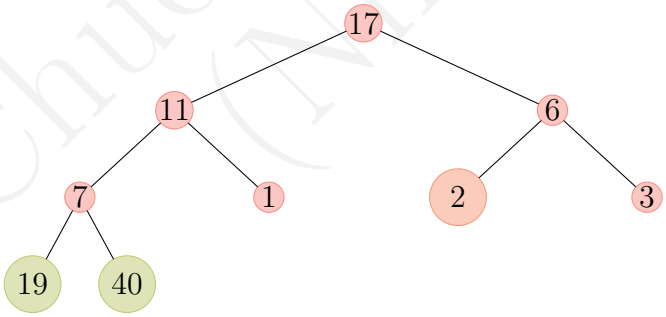


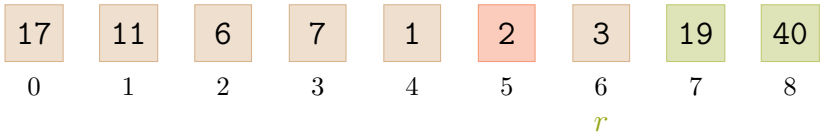


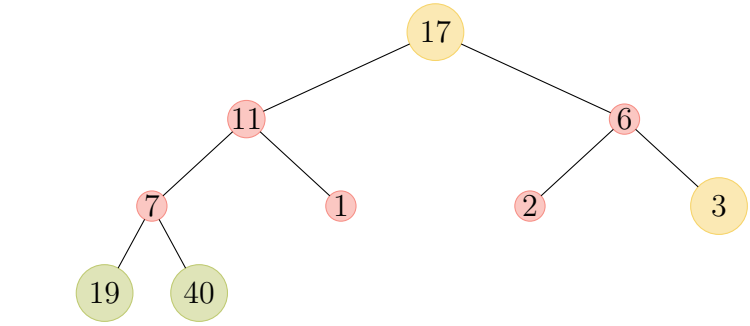


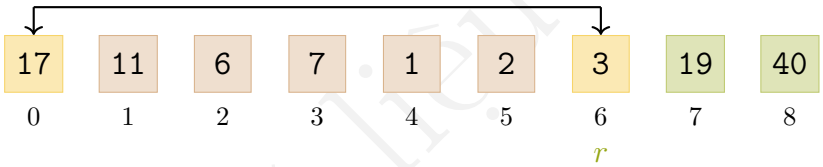


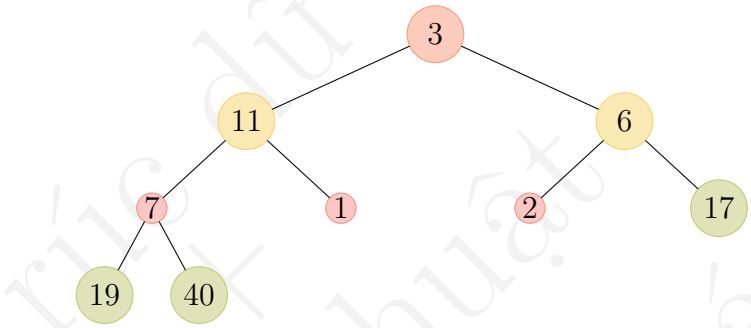


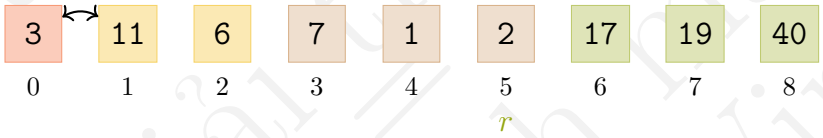


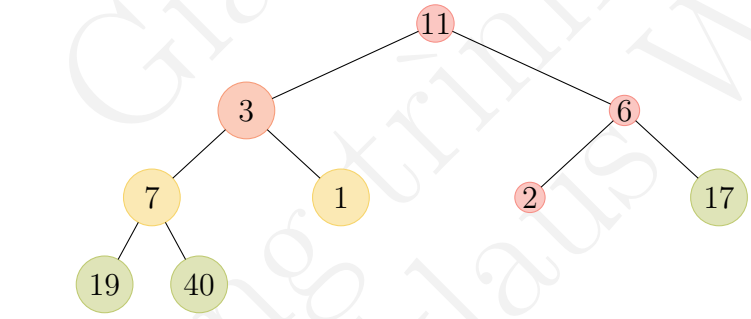


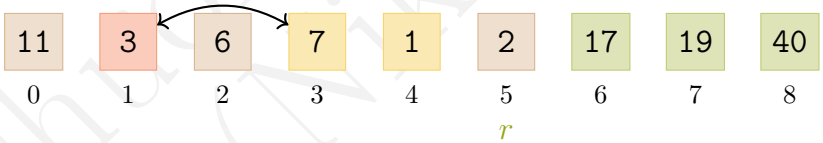


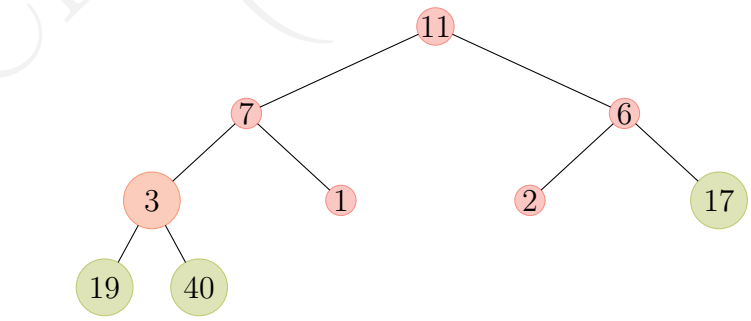


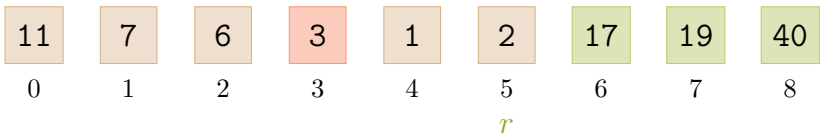


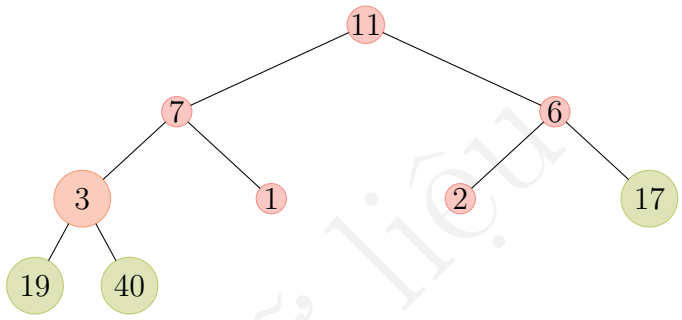


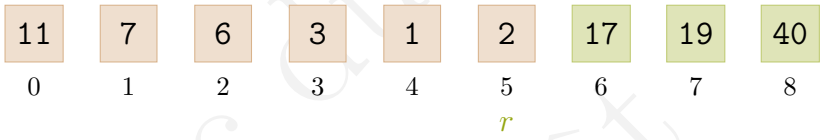


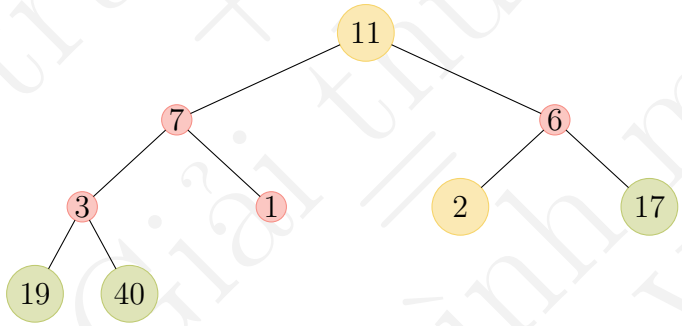


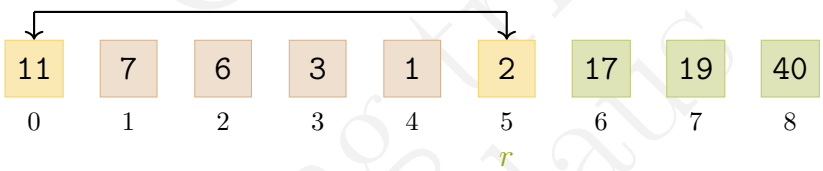


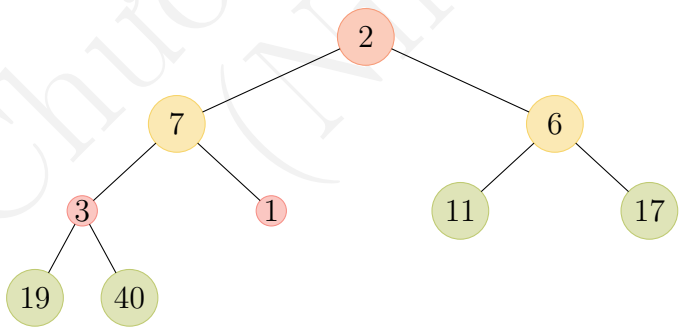


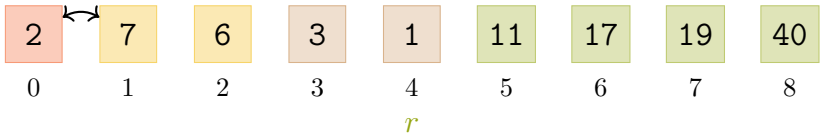


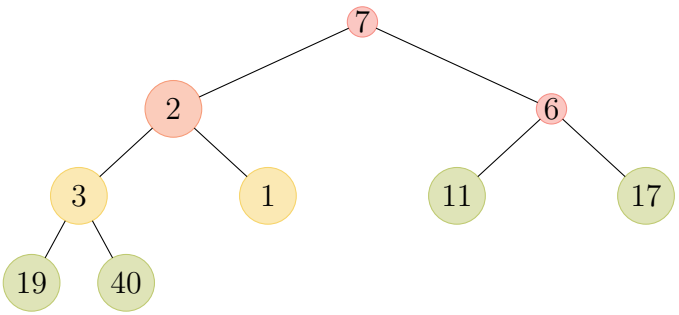


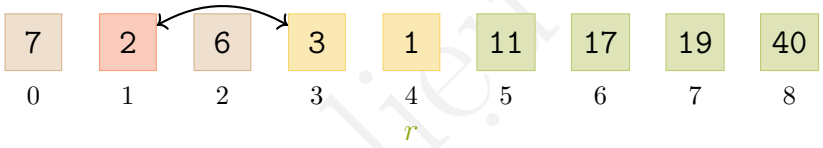


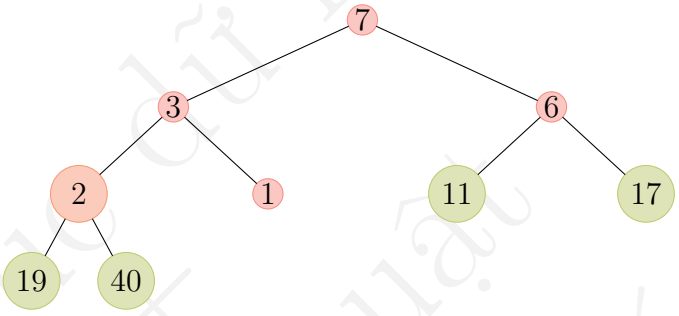


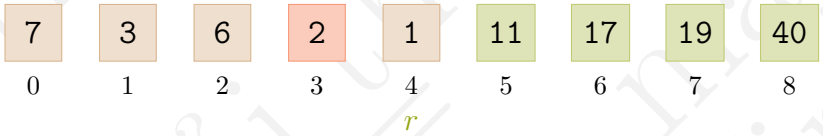


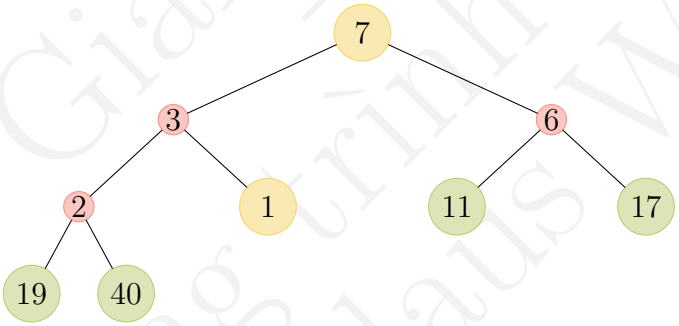


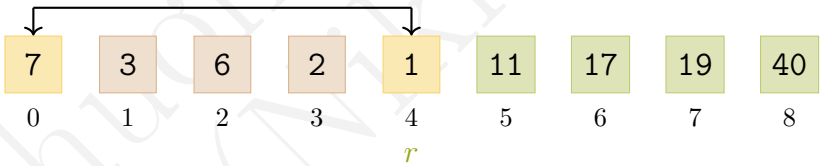


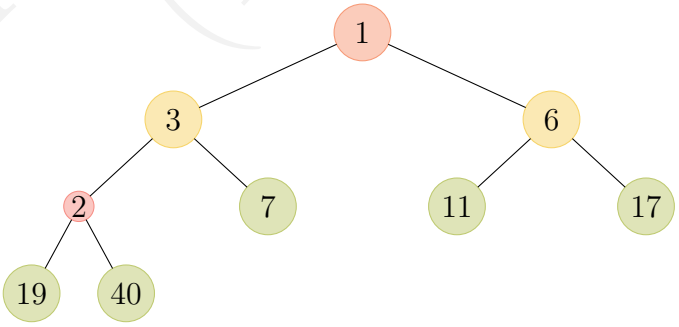


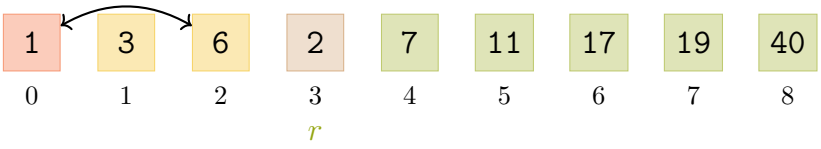


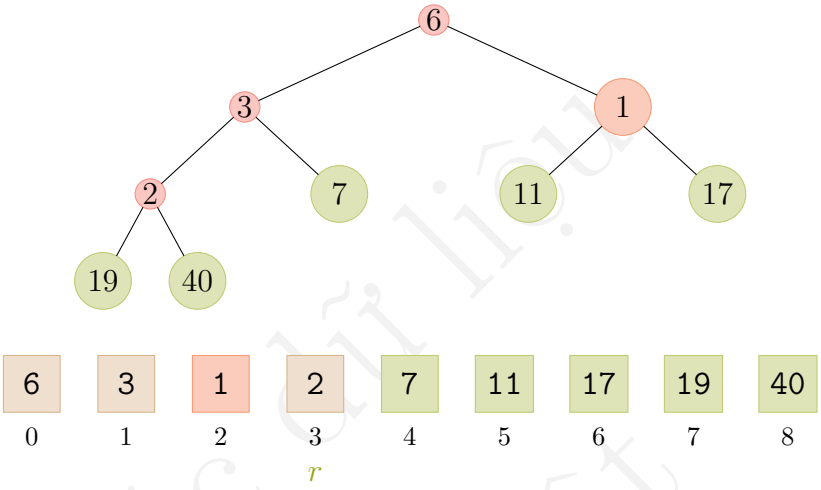


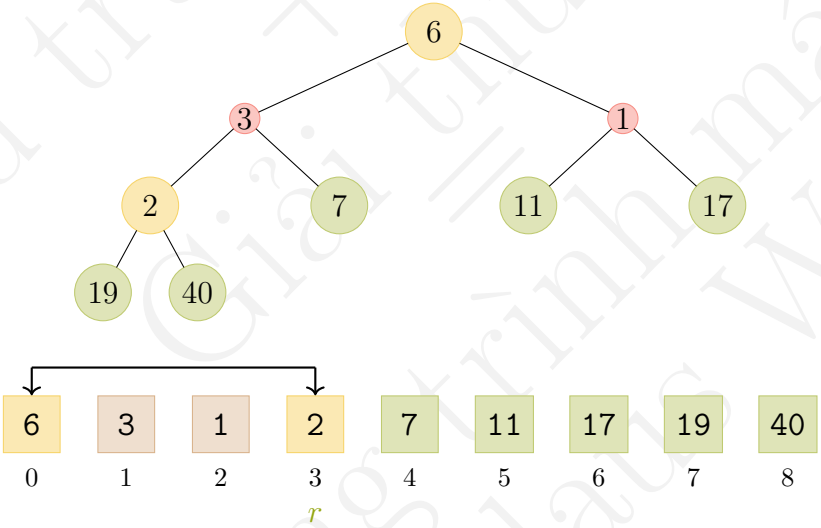


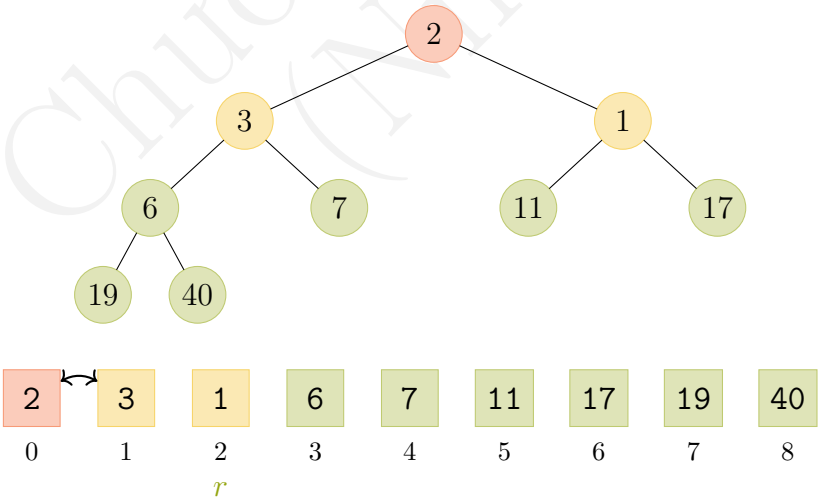


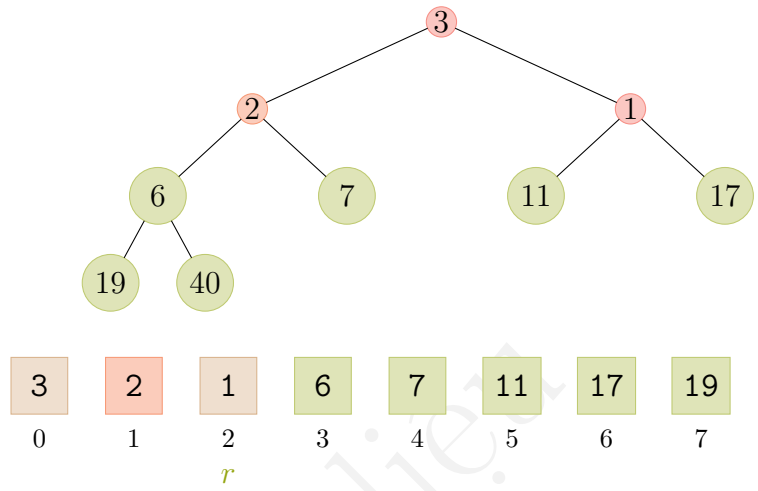


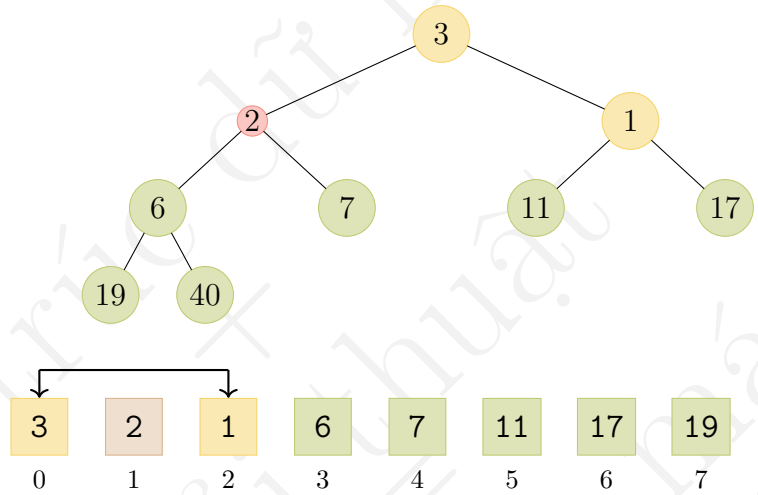


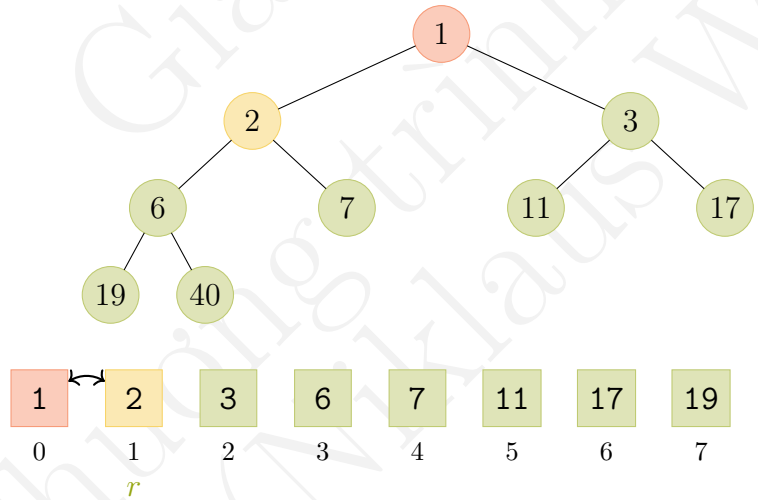


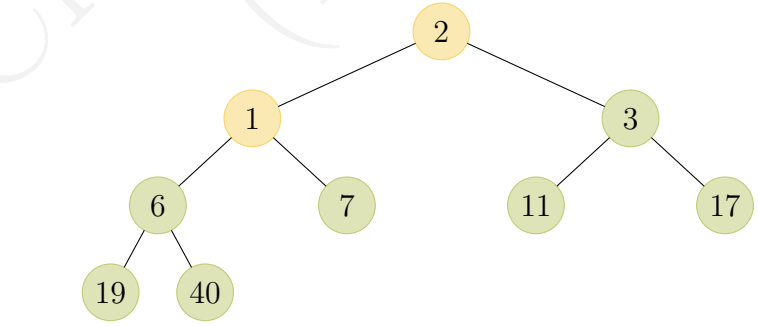


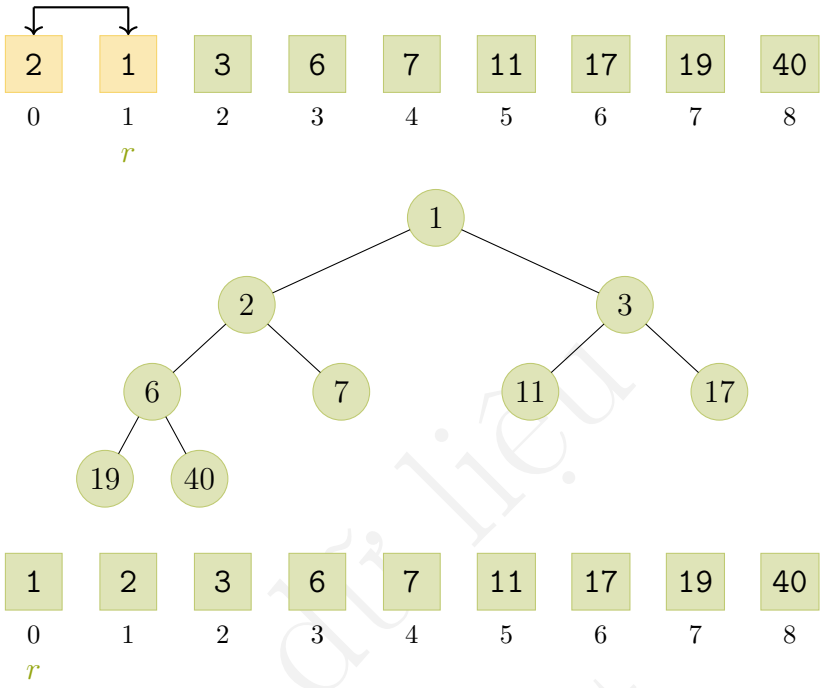




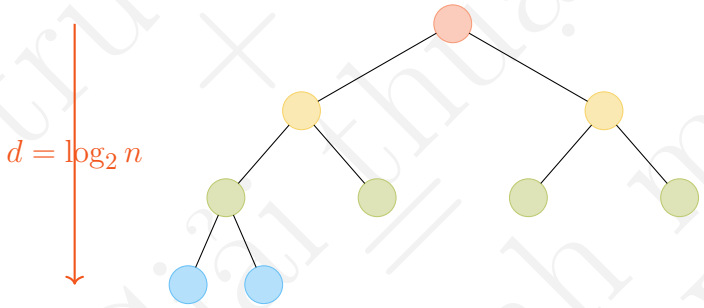








5.4.4 Độ phức tạp thuật toán



- Trường hợp xấu nhất, hàm hiệu chỉnh `Heapify` sẽ thực hiện từ nút gốc đến các nút lá. Cây có mức là d thì độ phức tạp thời gian của hàm hiệu chỉnh là

$$T(n) = O(d) = O(\log_2 n).$$

- Hiệu chỉnh heap là thao tác chính của thuật toán HeapSort.
  - Bước 1. Tạo heap $\frac{n}{2} - 1$.
  - Bước 2. Sắp xếp heap $n - 1$.
- Do đó, độ phức tạp thời gian của thuật toán HeapSort

$$T(n) = \left(\frac{3n}{2} - 2\right) \log_2 n = O(n \log n).$$

5.5 Bài tập cuối chương

1. Áp dụng giải thuật QuickSort sắp xếp dãy sau theo thứ tự tăng dần.

$$6, 5, 4, 3, 2, 1$$

- Chọn phần tử chốt tại vị trí đầu tiên
- Chọn phần tử chốt tại vị trí giữa như giải thuật đã học.

2. Áp dụng giải thuật sắp xếp nhanh xây dựng giải thuật chọn một phần tử nhỏ thứ k trong dãy không thứ tự.

3. Xây dựng giải thuật MergeSort trộn 3-dãy *(3-way)*. Áp dụng giải thuật sắp dãy sau theo thứ tự tăng dần

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

4. Giả sử Max-Heap là một dãy số gồm nhiều phần tử khác nhau, hãy cho biết phần tử nhỏ nhất nằm ở vị trí nào trong trường hợp biểu diễn bằng mảng, cấu trúc cây nhị phân?

5. Áp dụng thuật toán HeapSort sắp xếp dãy b và c theo thứ tự tăng dần.

   - $a = 1, 2, 3, 4, 5$.
   - $b = 5, 4, 3, 2, 1$.

   Hãy nhận xét sau khi thực hiện xong thuật toán.

Bài 6

DANH SÁCH LIÊN KẾT (6 tiết)

6.1 Giới thiệu cấu trúc dữ liệu động

Khái niệm

- Kiểu dữ liệu tĩnh
  - Là kiểu dữ liệu có kích thước *(số phần tử)* xác định *(không thay đổi trong vòng đời/chu kỳ sống)* như: số nguyên, số thực, ký tự, mảng, ...
  - Sử dụng phương pháp truy xuất trực tiếp *(direct access)* để truy xuất hay sửa một phần tử trong mảng.
  - Không có thao tác thêm và xóa một phần tử trên mảng.
- Kiểu dữ liệu động
  - Là kiểu dữ liệu có kích thước thay đổi.
  - Sử dụng phương pháp truy xuất tuần tự (sequential access) để thực hiện các thao tác thêm, sửa, xóa một phần tử.
  - Trong ngôn ngữ lập trình C và C++, kiểu dữ liệu con trỏ thường được dùng để cấp phát động một kiểu dữ liệu, mảng, cấu trúc, đối tượng.

6.2 Giới thiệu danh sách liên kết

Danh sách liên kết (Linked List)

- Là một dãy các nút (phần tử) được liên kết với nhau thông qua con trỏ liên kết.
- Các nút không cần lưu trữ liên tiếp nhau trong bộ nhớ.
- Kích thước của dãy có thể mở rộng!
- Thao tác thêm/xóa một nút không cần dịch chuyển các nút.

Ví dụ 6.2.1. *Danh sách liên kết chứa 4 phần tử như sau:*

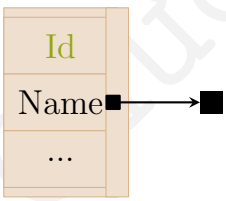
- Cấu trúc dữ liệu của một nút gồm
  - Thành phần dữ liệu.
  - Thành phần liên kết: con trỏ liên kết với nút kế tiếp (pNext) hoặc `NULL` nếu là nút cuối danh sách.

```
1 public class Node
2 {
3     int Info;
4     Node    pNext;
5 }
```

```
1 public class Node
2 {
3     Student Info;
4     Node    pNext;
5 }
```

Giải thích

- Dòng 4: Data là một kiểu dữ liệu: int, double, $\cdots$, hay là kiểu dữ liệu cấu trúc *(struct)* tự định nghĩa.
- Trong thực tế, thành phần dữ liệu *(Data)* thường là kiểu cấu trúc *(struct)*

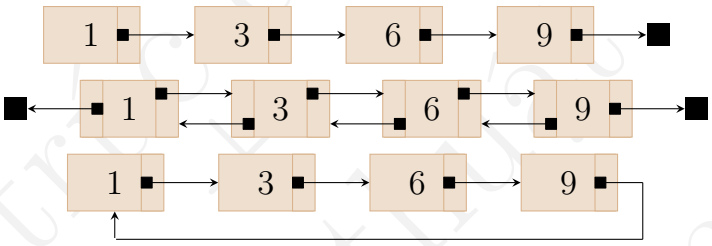



```
1 public class Student
2 {
3     public string Id;
4     public string Name;
```

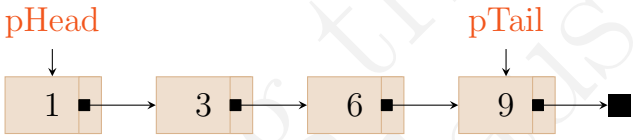
```
5       public bool Gender;
6   }
7
8   public class Node
9   {
10      public Student  Info;
11      public Node pNext;
12  }
```

Phân loại danh sách liên kết

- Danh sách liên kết đơn *(Single - Linked list)*: mỗi nút chỉ có 1 con trỏ liên kết (pNext).
- Danh sách liên kết đôi *(Double - Linked list)*: mỗi nút có 2 con trỏ liên kết (pPrev, pNext).
- Danh sách liên kết vòng *(Circular - Linked list)*: liên kết ở nút cuối cùng của danh sách chỉ đến nút đầu tiên trong danh sách.



- Một danh sách được quản lý bởi con trỏ đầu *(pHead)* lưu trữ địa chỉ nút đầu tiên.
- Trong thực tế, có trường hợp cần truy xuất nút cuối danh sách nên có thể sử dụng thêm con trỏ cuối *(pTail)* để quản lý địa chỉ nút cuối.
- pHead và pTail không phải là một nút mà chỉ là con trỏ trỏ đến một nút.




```
1   public class List
2   {
3       public Node pHead;
4       public Node pTail;
5   }
```

6.3 Các thao tác với danh sách liên kết đơn

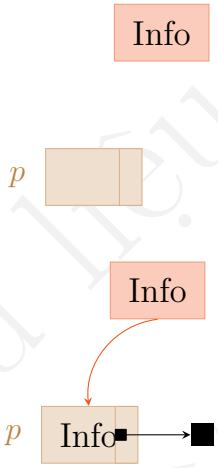
Một danh sách liên kết thường có những thao tác sau:

- Thao tác khởi tạo
- Thao tác thêm phần tử
- Thao tác xóa phần tử
- Thao tác duyệt

6.3.1 Thao tác khởi tạo

Thao tác khởi tạo một nút

- Khai báo một biến con trỏ *trỏ đến kiểu dữ liệu* danh sách đã được định nghĩa.
- Gán *thành phần dữ liệu* và *thành phần liên kết* cho biến này.

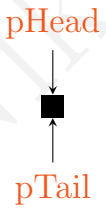



```
1 public static Node InitNode(Student std)
2 {
3     Node p = new Node();
4     p.Info = std;
5     p.pNext = null;
6     return p;
7 }
```

Thao tác khởi tạo danh sách

- Gán hai con trỏ pHead và pTail đến *NULL*.

```
1 public static void InitList()
2 {
3     pHead = null;
4     pTail = null;
5 }
```

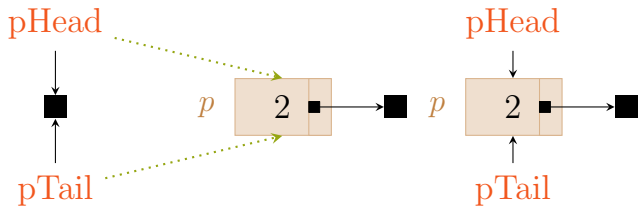


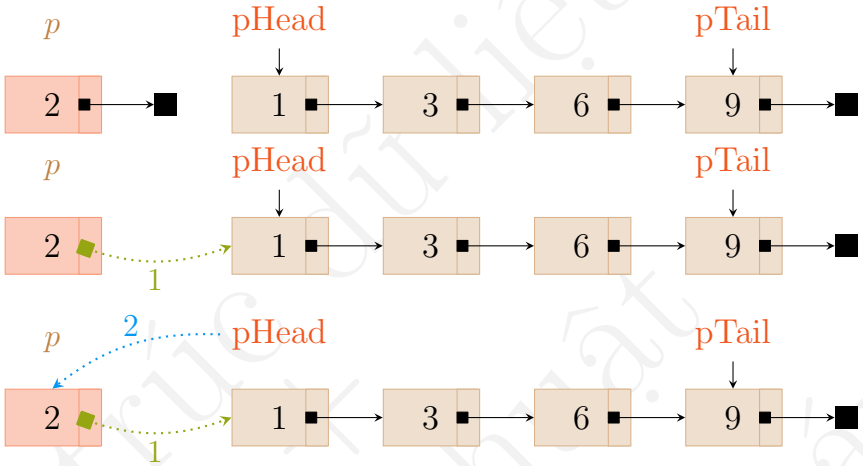
6.3.2 Thao tác thêm phần tử

Thuật toán thêm một phần tử vào danh sách liên kết có nhiều loại, phụ thuộc vào vị trí phần tử cần thêm.

Thao tác thêm đầu

Chú ý, trường hợp danh sách rỗng thì con trỏ đầu và cuối sẽ trỏ đến nút p chứa giá trị x.



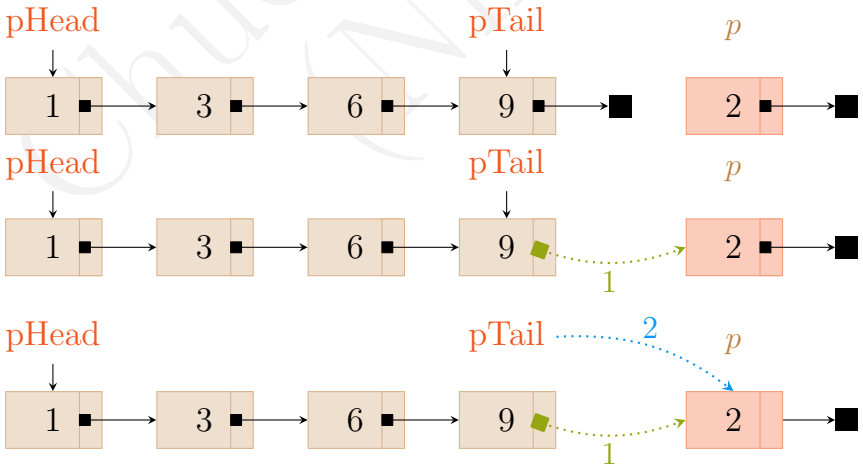


Thuật toán 6.1: `InsertHead(l, x)`

- Đầu vào: danh sách `l` và giá trị nút cần thêm.
- Đầu ra: danh sách `l` sau khi thêm đầu.

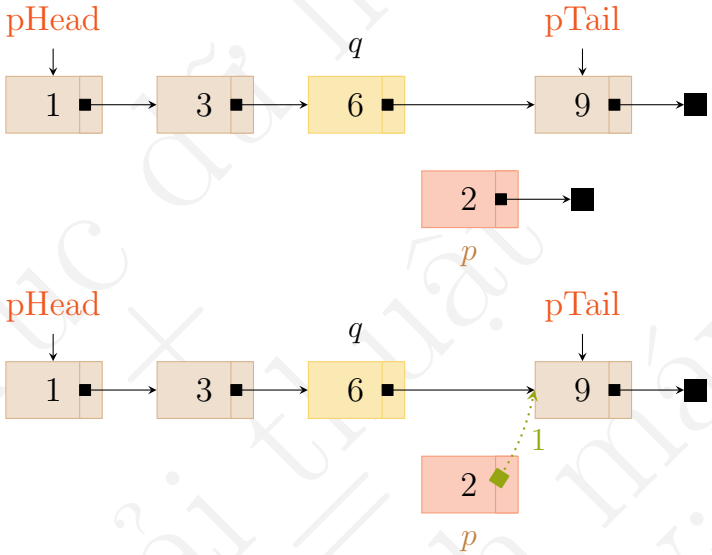
```
1 |   Khởi tạo nút p có giá trị x
2 |   if danh sách rỗng
3 |       pHead trỏ đến p
4 |       pTail trỏ đến p
5 |   else
6 |       p->pNext trỏ đến pHead
7 |       Cập nhật pHead
```

Thao tác thêm cuối

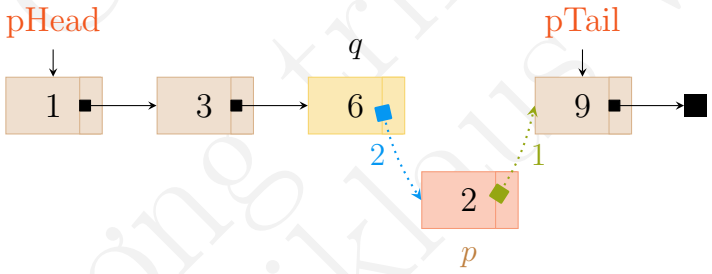


Thuật toán 6.2: InsertTail(l, x)
- Đầu vào: danh sách l và giá trị nút cần thêm.
- Đầu ra: danh sách l sau khi thêm cuối.

```
1  Khởi tạo nút p có giá trị x
2  if danh sách rỗng
3      pHead trỏ đến p
4      pTail trỏ đến p
5  else
6      Thêm p vào pTail->pNext
7      Cập nhật pTail
```



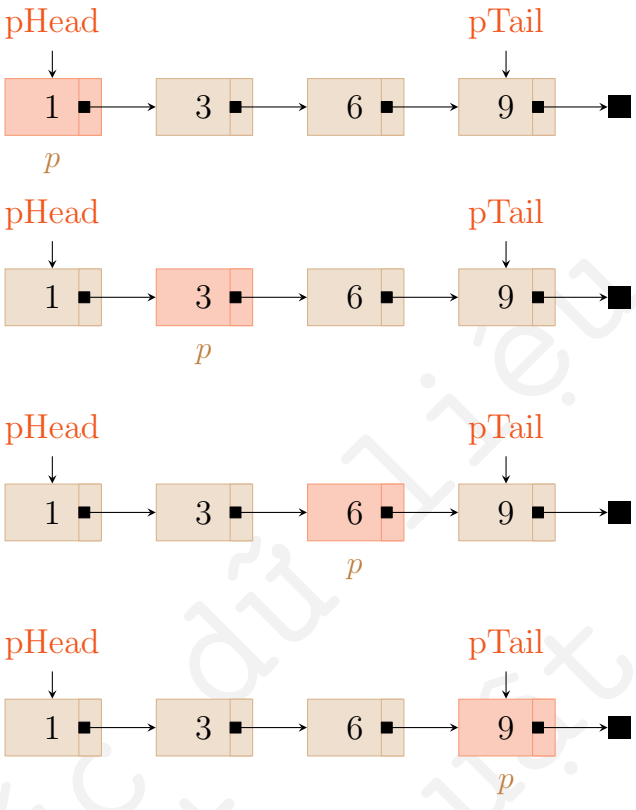
Thao tác thêm sau một nút khác



Chú ý trường hợp: nút q là nút cuối của danh sách.

Thuật toán 6.3: InsertAfter(l, q, x)
- Đầu vào: danh sách l, nút q và giá trị nút cần thêm.
- Đầu ra: danh sách l sau khi thêm một nút sau nút q.

```
1  Khởi tạo nút p có giá trị x
2  if q ≠ null
3      p->pNext trỏ đến q->pNext
4      q->pnext trỏ đến p
5  if q là nút cuối
6      Cập nhật pTail
```



6.3.3 Thao tác duyệt phần tử

Thuật toán 6.4: Traverse(l)
- Đầu vào: danh sách l.
- Đầu ra:

```
1 Khai báo nút p trỏ đến pHead
2 while chưa duyệt hết danh sách
3     // ...
4     p trỏ đến p->pNext
```

```
1  public void Traverse()
2  {
3      Node p = pHead;
4      while (p != null)
5      {
6          // In, tim gia tri
7          // ...
8
9          // Chuyen den nut ke tiep
10         p = p.pNext;
11     }
12 }
```

Giải thích

- Dòng 7: mã nguồn tương ứng với thao tác: tìm, in, ... các phần tử trong một danh sách.

Thao tác tìm kiếm

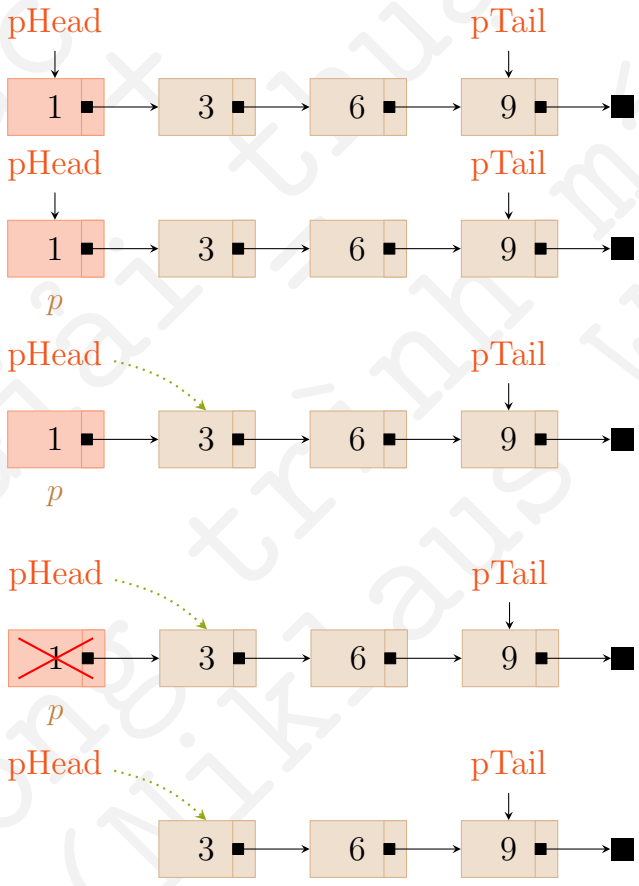
```
1  public Node Search(int x)
2  {
3      Node p = pHead;
4
5      while (p != null && p.Info != x)
6      {
7          p = p.pNext;
8      }
9      return p;
10 }
```

Giải thích

- Dòng 5: tương tự thao tác duyệt danh sách. Mỗi lần duyệt kiểm tra giá trị x với thành phần dữ liệu của nút đang xét.

6.3.4 Thao tác xóa phần tử

Thuật toán xóa một phần tử khỏi danh sách liên kết cũng phụ thuộc vào vị trí cần xóa. **Thao tác xóa đầu**



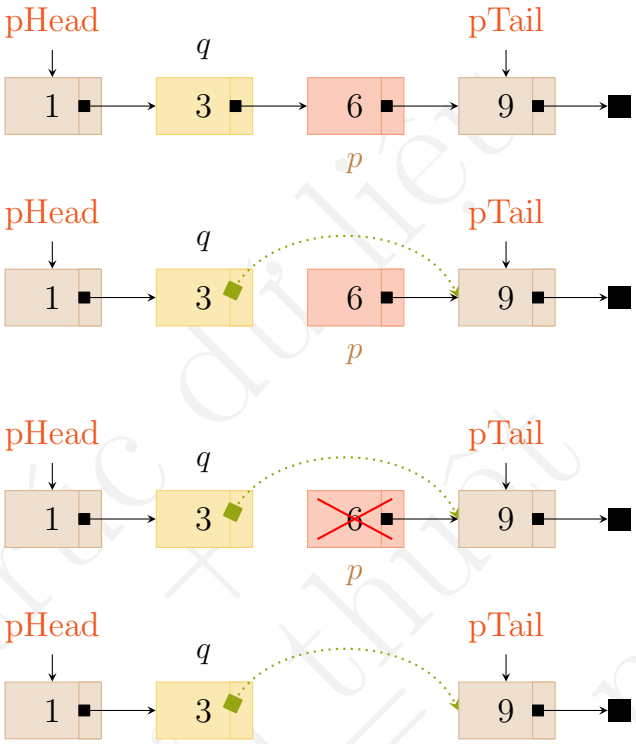
Chú ý hai trường hợp: danh sách rỗng và danh sách chỉ chứa một nút.

Thuật toán 6.5: RemoveHead(l)
- Đầu vào: danh sách l.
- Đầu ra: danh sách l sau khi xóa nút đầu.

```
1      if danh sách khác rỗng
2          p trỏ đến pHead
```

```
        pHead trỏ đến pHead->pNext
        Xóa nút p
        if danh sách rỗng
            Cập nhật pTail
```

Thao tác xóa sau một nút khác Giả sử nút xóa nút p sau nút q.

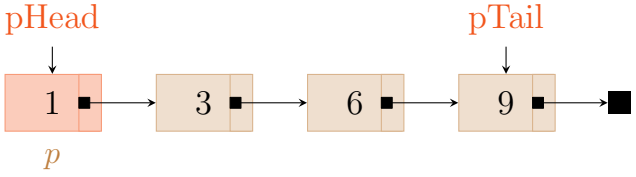


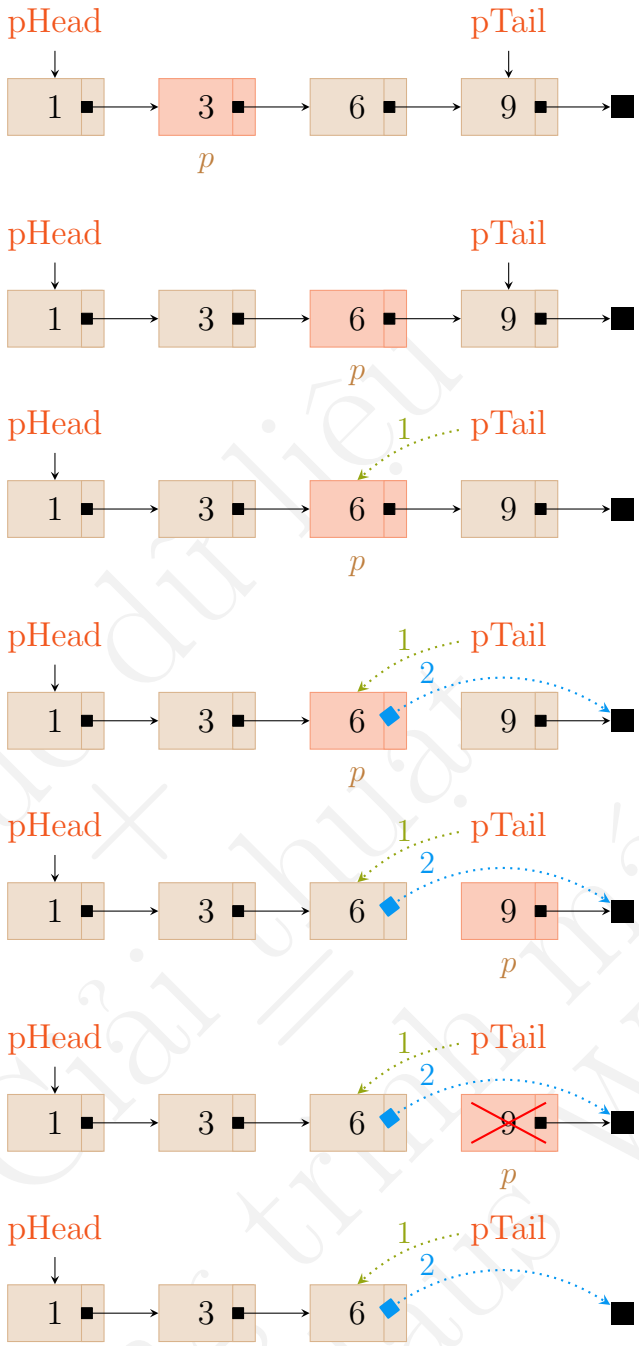
Chú ý trường hợp: nút q là nút kế cuối (nút p là nút cuối) của danh sách (cần cập nhật lại con trỏ pTail).

Thuật toán 6.6: RemoveAfter(l, q)
- Đầu vào: danh sách l và nút q.
- Đầu ra: danh sách l sau khi xóa nút.

```
    if q ≠ null
        p trỏ đến q->pNext
        if p ≠ null
            // Nếu p là nút cuối danh sách
            if p là phần tử cuối của danh sách
                Cập nhật pTail trước khi xóa p
            // Ngược lại, p không là nút cuối danh sách
            q->pNext trỏ đến p->pNext
            Xóa nút p
```

Thao tác xóa cuối



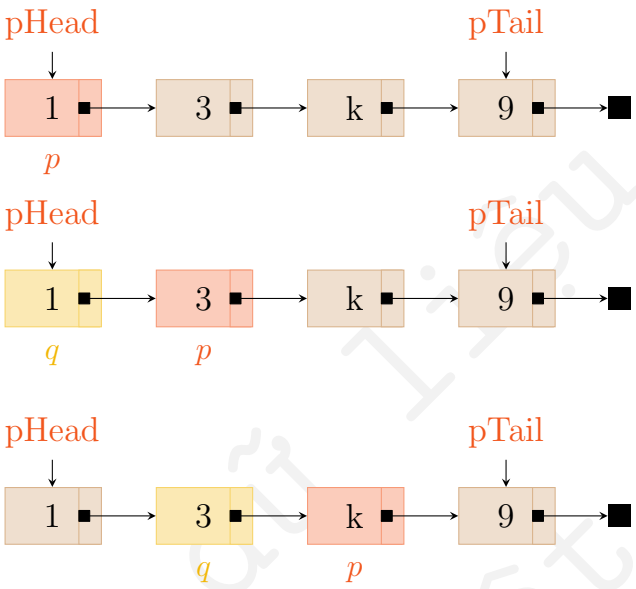


Chú ý hai trường hợp: danh sách rỗng và danh sách chỉ chứa một nút.

Thuật toán 6.7: RemoveTail(l)
- Đầu vào: danh sách l.
- Đầu ra: danh sách l sau khi xóa cuối.

```
1 |     if danh sách khác rỗng
2 |         p trỏ đến pHead
3 |     if danh sách chỉ chứa 1 nút p
4 |         Xóa nút p ···
5 |         Kết thúc
6 |     while p chưa trỏ đến vị trí kế cuối của danh sách
7 |         p trỏ đến p->pNext
8 |         Cập nhật pTail
9 |         Xóa nút p ···
```

Thao tác xóa nút có khóa k Giả sử, p là nút cần xóa có chứa khóa là k, thao tác xóa sẽ thực hiện theo các bước sau:



Chú ý các trường hợp:

- Nút có khóa k là nút đầu danh sách.
- Nút có khóa k là nút giữa danh sách.
- Nút có khóa k là nút cuối danh sách.

Thuật toán 6.8: RemoveNode(l, k)

- Đầu vào: danh sách l và giá trị k của nút cần xóa.
- Đầu ra: true hay false.

```
1  Lặp tìm nút p có giá trị k và nút q là nút trước của p ...
2      if p = null // TH1. Không tìm thấy p
3          return false
4      if q = null // TH2. Tìm thấy p và p là nút đầu danh sách
5          Thực hiện thao tác xóa đầu···
6      else q ≠ null // Tìm thấy p và q (q, p)
7          if p là nút cuối danh sách // TH3. p là nút cuối danh sách···

8              Cập nhật pTail
9          // TH4. p là nút giữa danh sách···
10         q->pNext trỏ đến p->pNext
11         Xóa nút p
12     return true
```

6.4 Mã nguồn danh sách liên kết

```
1  public class Student
2  {
3      public string Id;
```

```
4       public string Name;
5       public bool Gender;
6   }
7
8   public class Node
9   {
10      public Student Info;
11      public Node pNext;
12
13      public static Node InitNode(Student std)
14      {
15          Node p = new Node();
16          p.Info = std;
17          p.pNext = null;
18          return p;
19      }
20  }
21
22  public class List
23  {
24      public Node pHead;
25      public Node pTail;
26
27      public static void InitList()
28      {
29          pHead = null;
30          pTail = null;
31      }
32
33      public void InsertHead(int x)
34      {
35          Node p = Node.InitNode(x);
36
37          if (pHead == null)
38          {
39              pHead = p;
40              pTail = p;
41          }
42          else
43          {
44              p.pNext = pHead;
45              pHead = p;
46          }
47      }
48
49      public void InsertTail(int x)
50      {
51          Node p = Node.InitNode(x);
52
53          if (pHead == null)
```

```
54          {
55              pHead = p;
56              pTail = p;
57          }
58          else
59          {
60              pTail.pNext = p;
61              pTail = p;
62          }
63      }
64
65      public void InsertAfter(Node q, int x)
66      {
67          Node p = Node.InitNode(x);
68
69          if (q != null)
70          {
71              p.pNext = q.pNext;
72              q.pNext = p;
73              if (q == pTail)
74              {
75                  pTail = p;
76              }
77          }
78      }
79
80      public void Traverse()
81      {
82          Node p = pHead;
83          while (p != null)
84          {
85              // In, tim gia tri
86              // ...
87
88              // Chuyen den nut ke tiep
89              p = p.pNext;
90          }
91      }
92
93      public Node Search(int x)
94      {
95          Node p = pHead;
96
97          while (p != null && p.Info != x)
98          {
99              p = p.pNext;
100         }
101         return p;
102     }
103
```

```
104     public void RemoveHead()
105     {
106         Node p = new Node();
107         if (pHead != null) // TH. Danh sach khac rong
108         {
109             p = pHead;
110             pHead = pHead.pNext;
111             p = null;
112             if (pHead == null)
113             {
114                 pTail = null;
115             }
116         }
117     }
118 
119     public void RemoveTail()
120     {
121         Node p = new Node();
122         if (pHead != null) // Danh sach khac rong
123         {
124             p = pHead;
125             if (p == pTail) // TH1: 1 nut
126             {
127                 p = null;
128                 pHead = null;
129                 pTail = null;
130                 return;
131             }
132             // TH2: > 1 nut
133             while (p.pNext != pTail)
134             {
135                 p = p.pNext;
136             }
137             pTail = p;
138             pTail.pNext = null;
139             p = p.pNext;
140             p = null;
141         }
142     }
143 
144     public void RemoveAfter(Node q)
145     {
146         Node p = new Node();
147         if (q != null)
148         {
149             p = q.pNext;
150             if (p != null)
151             {
152                 if (p == pTail)
153                 {
```

```
154                     pTail = q;
155                 }
156                 q.pNext = p.pNext;
157                 p = null;
158             }
159         }
160     }
161
162     public bool RemoveNode(int k)
163     {
164         Node p = pHead;
165         Node q = null;
166         while (p != null)
167         {
168             if (p.Info == k)
169             {
170                 break;
171             }
172             q = p;
173             p = p.pNext;
174         }
175         if (p == null) // TH1. Khong tim thay p
176         {
177             return false;
178         }
179         if (q != null) // TH2: Tim thay p va p la nut dau
                danh sach
180         {
181             pHead = p.pNext;
182             if (pHead == null)
183             {
184                 pTail = null;
185             }
186         }
187         else // Tim thay p va q (q, p)
188         {
189             if (p == pTail) // TH3. p la phan tu cuoi danh
                    sach
190             {
191                 pTail = q;
192             }
193             // TH4. p la phan tu giua danh sach
194             q.pNext = p.pNext;
195             p = null;
196         }
197
198         return true;
199     }
200 }
```

6.5 Ứng dụng của danh sách liên kết

Xây dựng cấu trúc dữ liệu thích hợp để biểu diễn đa thức $P(x)$ có dạng như sau:

$$P(x) = c_1x^{e_1} + c_2x^{e_2} + \cdots + c_nx^{e_n}$$

với c_i là hệ số và e_i là số mũ, $1 \leq i \leq n$
Cài đặt hàm thực hiện các thao tác:

1. Thêm đơn thức vào cuối đa thức.
2. In đa thức
3. Tính giá trị đa thức với x cho trước.
4. **Điều chỉnh hàm in đa thức theo đúng định dạng:**

$$2x^2 - 3x + 5$$

5. **Tính tổng 2 đa thức.**

```
class DonThuc
{
    public int he_so;
    public int so_mu;
}
```

```
class Node
{
    public DonThuc don_thuc;
    public Node pNext;

    public static Node InitNode(int hs, int sm)
    {
        Node p = new Node();
        p.don_thuc = new DonThuc();

        p.don_thuc.he_so = hs;
        p.don_thuc.so_mu = sm;
        p.pNext = null;

        return p;
    }
}
```

```
class LinkedList
{
    public Node pHead;
    public Node pTail;

    public void InitList()
    {
        pHead = null;
```

```
9           pTail = null;
10      }
11
12      // 1. Them 1 don thuc vao cuoi cua da thuc
13      public void InsertTail(int hs, int sm)
14      {
15          Node p = Node.InitNode(hs, sm);
16          if (pHead == null)
17          {
18              pHead = p;
19              pTail = p;
20          }
21          else
22          {
23              pTail.pNext = p;
24              pTail = p;
25          }
26      }
27
28      // 2. In da thuc
29      public void Print()
30      {
31          Node p = pHead;
32          while (p != null)
33          {
34              Console.Write(p.don_thuc.he_so);
35              Console.Write("x");
36              Console.Write("^");
37              Console.Write(p.don_thuc.so_mu);
38              if (p != pTail)
39              {
40                  Console.Write("+");
41              }
42              p = p.pNext;
43          }
44      }
45
46      // 3. Tinh gia tri da thuc voi x cho truoc
47      public double Compute(int x)
48      {
49          double f = 0;
50
51          Node p = pHead;
52          while (p != null)
53          {
54              f +=  p.don_thuc.he_so
55              * Math.Pow(Convert.ToDouble(x),
56              Convert.ToDouble(p.don_thuc.so_mu));
57              p = p.pNext;
58          }
```

```
59
60          return f;
61      }
62
63      // 4.
64
65      // 5.
66
67  }
```

```
1   static void Main(string[] args)
2   {
3       LinkedList l = new LinkedList();
4       l.InitList();
5
6       l.InsertTail(2, 2); // 2x^2
7       l.InsertTail(-3, 1); // -3x^1
8       l.InsertTail(5, 0); // 5x^0
9
10      l.Print(); // 2x^2 + - 3x^1 + 5x^0
11      Console.WriteLine();
12      Console.WriteLine(l.Compute(2)); // 19
13
14      Console.ReadKey();
15  }
```

6.6 Bài tập cuối chương

1. Xây dựng cấu trúc dữ liệu thích hợp để biểu diễn đa thức $P(x)$ có dạng như sau:

$$P(x) = c_1 x^{e_1} + c_2 x^{e_2} + \cdots + c_n x^{e_n}$$

với c_i là hệ số và e_i là số mũ, $1 \leq i \leq n$
Các thao tác:

- Thêm đơn thức vào cuối đa thức.
- In đa thức.
- Tính giá trị đa thức với x cho trước.

2. Xây dựng cấu trúc dữ liệu thích hợp để quản lý danh sách sinh viên.
   - Dữ liệu mỗi sinh viên gồm các thông tin: MSSV, họ tên, giới tính, ngày sinh.
   - Các thao tác thực hiện với danh sách sinh viên gồm: thêm, xóa, tìm kiếm một sinh viên.

Bài 7

SẮP XẾP TRONG DANH SÁCH LIÊN KẾT (3 tiết)

7.1 Giới thiệu bài toán sắp xếp trên danh sách liên kết

Giới thiệu bài toán sắp xếp trên danh sách liên kết

- Sắp xếp là quá trình xử lý các phần tử của một dãy theo đúng thứ tự (thỏa tiêu chuẩn nào đó).
- Trong bài toán sắp xếp, hai thao tác cơ bản là so sánh và hoán vị giữa hai phần tử trong dãy.

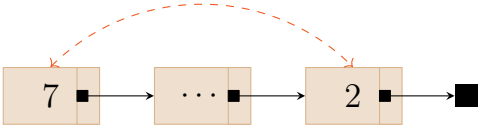
Hai phương pháp sắp xếp trên danh sách liên kết

- Hoán vị *thành phần dữ liệu (Info)* của một nút.
- Thay đổi *thành phần liên kết (pNext)* của một nút.

7.2 Phương pháp hoán vị thành phần dữ liệu

Phương pháp hoán vị thành phần dữ liệu

- Tương tự các thao tác sắp xếp trên mảng.
- Thao tác hoán vị chính là hoán vị thành phần dữ liệu của mỗi nút.
- *Thao tác hoán vị đòi hỏi thêm vùng nhớ trung gian chỉ thích hợp với dữ liệu có kích thước nhỏ.*



Nhắc lại thuật toán sắp xếp chọn *(Selection Sort)* thực hiện trên mảng

Thuật toán 7.1: `SelectionSort(a[], n)`

- Đầu vào: mảng a gồm n phần tử.
- Đầu ra: mảng a có thứ tự tăng dần.

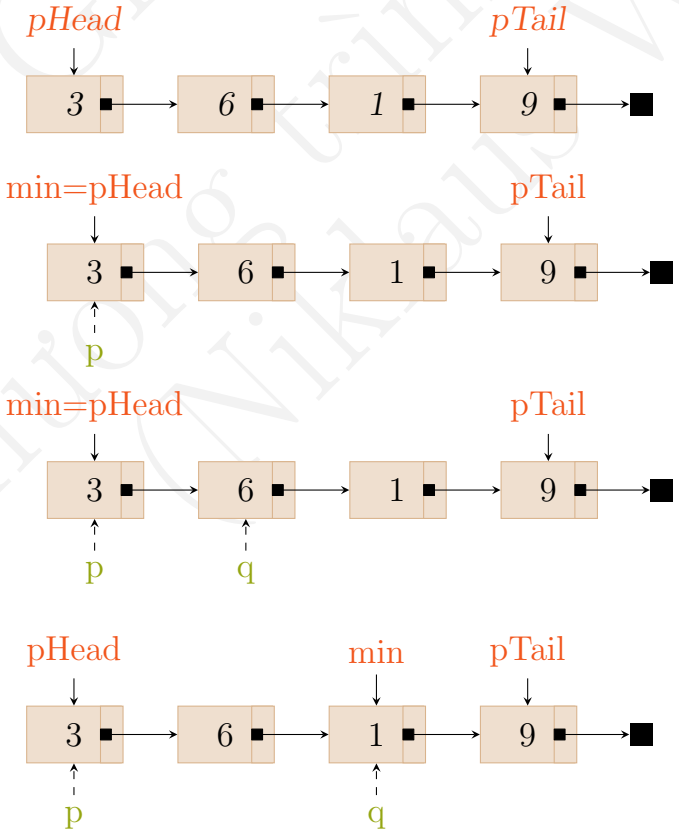
```
1   for i ← 0 to n - 2
2       min ← i
3       for j ← i + 1 to n - 1
4           if a[j] < a[min]
5           min ← j
6       Swap(a[i], a[min])
```

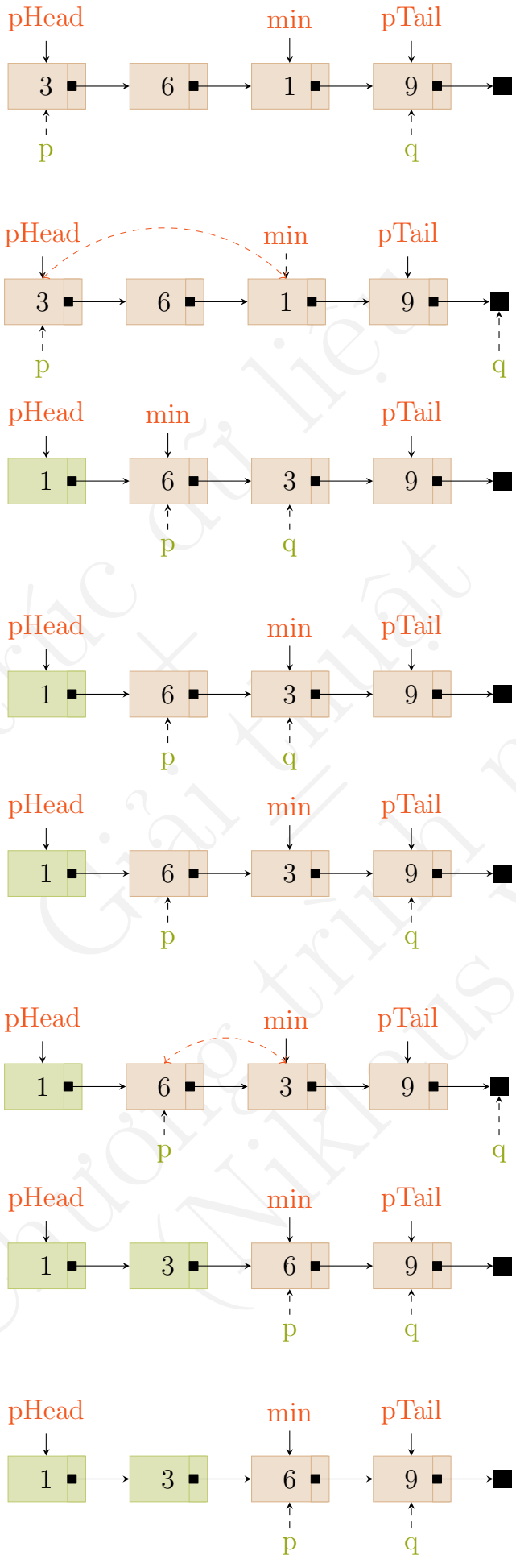
Thuật toán sắp xếp chọn *(Selection Sort)* thực hiện trên danh sách liên kết

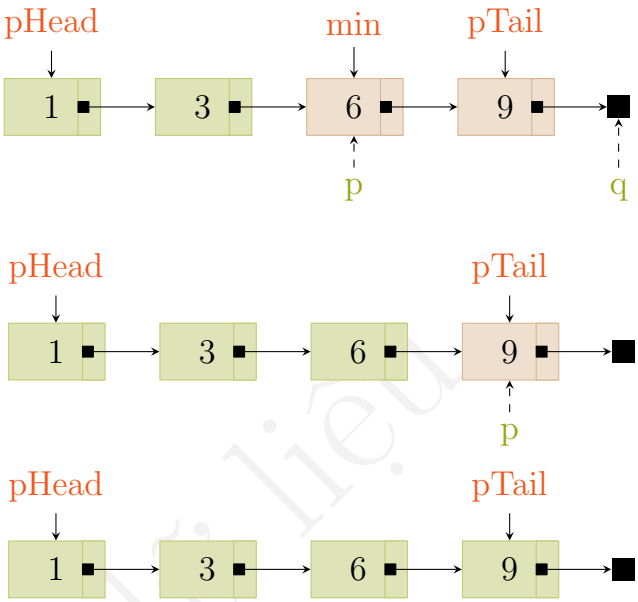
Thuật toán 7.2: `ListSeletionSort(l)`

```
1   // p la nut dau va p khac pTail
2   Khai báo p trỏ đến pHead
3   while p ≠ pTail
4       Khai báo nút min trỏ đến nút p
5       // q la nut sau p va q khac rong
6       Khai báo nút q trỏ đến nút sau p
7       while q ≠ null
8           if Info của nút q < Info của nút min
9               Nút min trỏ đến nút q
10      Gọi hàm hoán vị Info của hai nút p và min
11      p trỏ đến p->pNext
```

Ví dụ 7.2.1. *Sắp xếp danh sách liên kết theo thứ tự tăng dần của thành phần dữ liệu.*







Phương pháp hoán vị thành phần dữ liệu

```
1  public void ListSelectionSort()
2  {
3      Node p, q, min;
4      p = pHead;
5      while (p != pTail)
6      {
7          min = p;
8          q = p.pNext;
9          while (q != null)
10         {
11             if (q.Info < min.Info)
12                 min = q;
13             q = q.pNext;
14         }
15         Swap(p.Info, min.Info);
16         p = p.pNext;
17     }
18 }
```

7.3 Phương pháp thay đổi thành phần liên kết

Phương pháp thay đổi thành phần liên kết

- Khi sắp thứ tự, chỉ cần thực hiện thao tác *thay đổi thành phần liên kết pNext* giữa các nút.
- Kích thước của thành phần liên kết không phụ thuộc vào dữ liệu trong danh sách liên kết (4 byte, 8 byte, ... tùy thuộc hệ điều hành).
- Thuật toán sắp xếp QuickSort, MergeSort thực hiện hiệu quả với danh sách liên kết.

Thuật toán sắp xếp nhanh *(QuickSort)*

- Chọn x là *nút/phần tử đầu* danh sách làm phần tử chốt.
- Chia danh sách thành hai danh sách con:
  - Danh sách l_1: chứa các phần tử *nhỏ hơn hay bằng* x.
  - Danh sách l_2: chứa các phần tử *lớn hơn* x.
- Gọi đệ quy thực hiện với 2 danh sách l_1 và l_2.
- Danh sách l sắp theo thứ tự $l_1 \rightarrow x \rightarrow l_2$

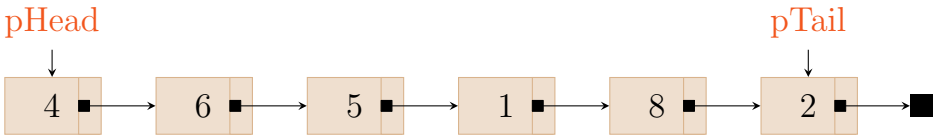
Thuật toán 7.3: ListQuickSort(l)

```
1   if danh sách chỉ chứa 1 nút
2       Dừng thuật toán.
3   x trỏ đến pHead // x la phan tu pivot
4   pHead trỏ đến nút sau x // loai x ra khoi danh sach
5   // 1. Chia: duyet tu dau den cuoi danh sach (da loai x)
6   while chưa duyệt hết danh sách
7       Tách p khỏi danh sách ...
8       if Info của p ≤ Info của x
9           Thêm p vào cuối danh sách l1
10      else
11          Thêm p vào cuối danh sách l2
12  // 2. Tri: goi de quy ham
13  Gọi đệ quy hàm ListQuickSort() với danh sách con l1
14  Gọi đệ quy hàm ListQuickSort() với danh sách con l2
15  // 3. Tong hop ket qua l: l1->x->l2
16  Gọi hàm ListAppend(l, l1, x, l2) nối danh sách l1, l2 va nút x
```

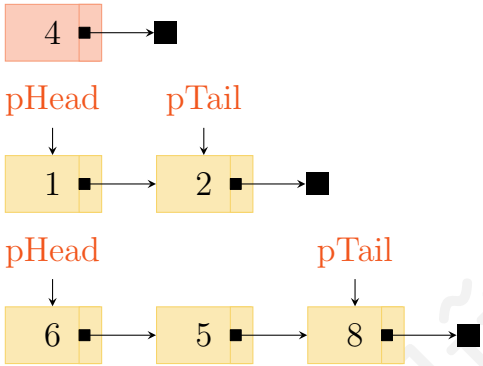
Thuật toán 7.4: ListAppend(l1, x, l2)

```
- Đầu vào: danh sách l1, l2 và nút x.
- Đầu ra: danh sách l sau khi nối (l1->x->l2).
```

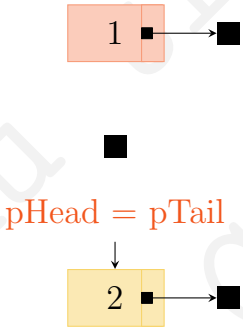
```
1   if danh sách l1 rỗng
2       pHead của l trỏ đến x
3   else
4       Cập nhật pHead của l
5       pTail->pNext của l1 trỏ đến x
6
7   x->pNext trỏ đến pHead của l2
8
9   if danh sách l2 rỗng
10      pTail của l trỏ đến x
11  else
12      Cập nhật pTail của l
```



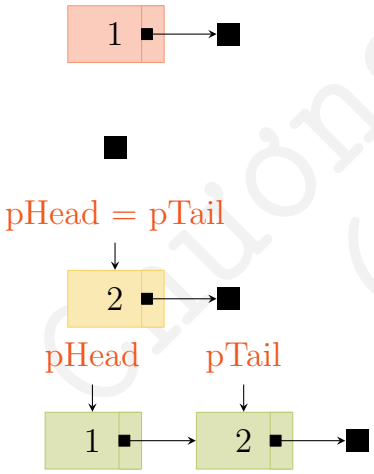
- $l = 4, 6, 5, 1, 8, 2$
- $x = 4$
- $l_1 = 1, 2$
- $l_2 = 6, 5, 8$



- $l_1 = 1, 2$
- $x = 1$
- $l_{11} = \emptyset$
- $l_{12} = 2$



- $l_1 = l_{11}|x|l_{12}$



- Cài đặt thêm hàm `InsertTail(Node p)` để thêm một nút vào danh sách liên kết.

- Cài đặt hàm `ListQuickSort()`.
- Cài đặt hàm `ListAppend(List l1, Node x, List l2)`.

```
public void ListQuickSort()
{
    Node x, p;
    List l1, l2;
    if (pHead == pTail)
        return;
    // Khởi tạo nút p, danh sách l1, l2 ...
    x = pHead;
    pHead = x.pNext;
    while (pHead != NULL)
    {
        p = pHead;
        pHead = p.pNext;
        p.pNext = NULL;
        if (p.Info <= x.Info)
            InsertTail(l1, p);
        else
            InsertTail(l2, p);
    }
    ListQuickSort(l1);
    ListQuickSort(l2);
    ListAppend(l, l1, x, l2);
}
```

```
public void ListAppend(List l1, Node x, List l2)
{
    if (l1.pHead == null)
        pHead = x;
    else
    {
        pHead = l1.pHead;
        l1.pTail.pNext = x;
    }

    x.pNext = l2.pHead;

    if (l2.pHead == null)
        pTail = x;
    else
        pTail = l2.pTail;
}
```

7.4 Bài tập cuối chương

Cài đặt các thuật toán sắp xếp trên danh sách liên kết:

1. Thuật toán *SelectionSort, InterchangeSort*

2. Thuật toán *QuickSort*.

Bài 8

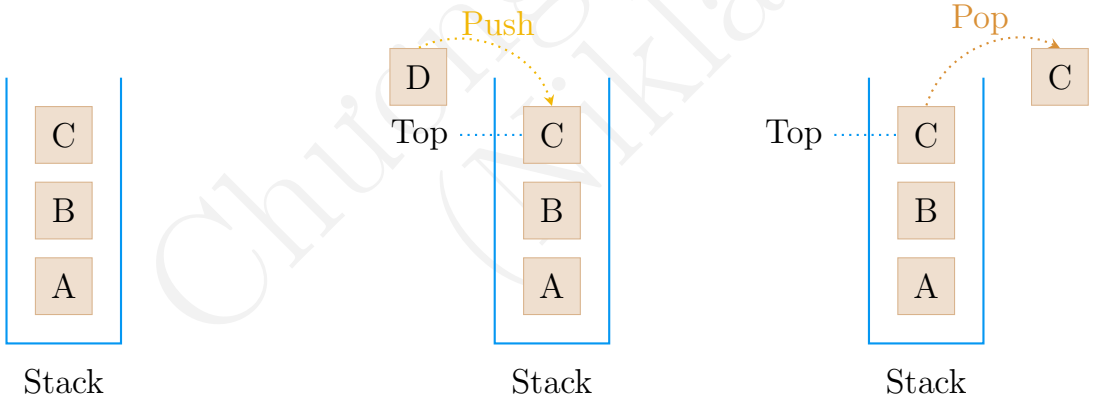
NGĂN XẾP & HÀNG ĐỢI (6 tiết)

8.1 Giới thiệu ngăn xếp

Khái niệm

Ngăn xếp (Stack) là một cấu trúc dữ liệu thường được sử dụng.

- Thực hiện theo cơ chế *LIFO (Last In, First Out) vào sau ra trước*.
- Dùng để lưu trữ các phần tử có thứ tự truy xuất *ngược* với thứ tự lưu trữ.



Hình 8.2: Hình minh họa ngăn xếp.

Các thao tác cơ bản trong ngăn xếp:

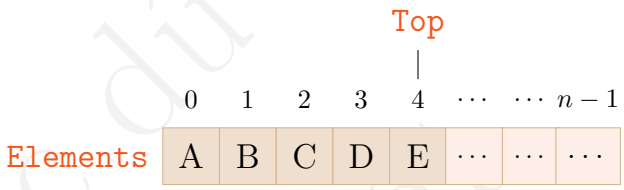
- Push: *thêm* phần tử vào *đỉnh* ngăn xếp.

- Pop: *xóa* phần tử tại *đỉnh* ngăn xếp.
- GetTop: *lấy* phần tử tại *đỉnh* ngăn xếp.
- Kiểm tra ngăn xếp rỗng, đầy.

8.2 Cài đặt ngăn xếp

8.2.1 Cài đặt ngăn xếp bằng mảng

- Một mảng 1 chiều kích thước n: lưu trữ phần tử từ vị trí $[0, ..., n-1]$.
- Một biến *top* kiểu số nguyên: cho biết vị trí đỉnh ngăn xếp. *Mặc định, ngăn xếp vừa khởi tạo* `top = -1`.



Cấu trúc của một ngăn xếp và hàm khởi tạo ngăn xếp

```
1  public class Stack
2  {
3      public int[] elements;
4      public int top;
5
6      public void InitStack()
7      {
8          elements = new int[MAX_SIZE];
9          top = -1;
10     }
11 }
```

Đối với một số thao tác, trước khi thực hiện cần phải kiểm tra ngăn xếp có rỗng hoặc đầy hay chưa?

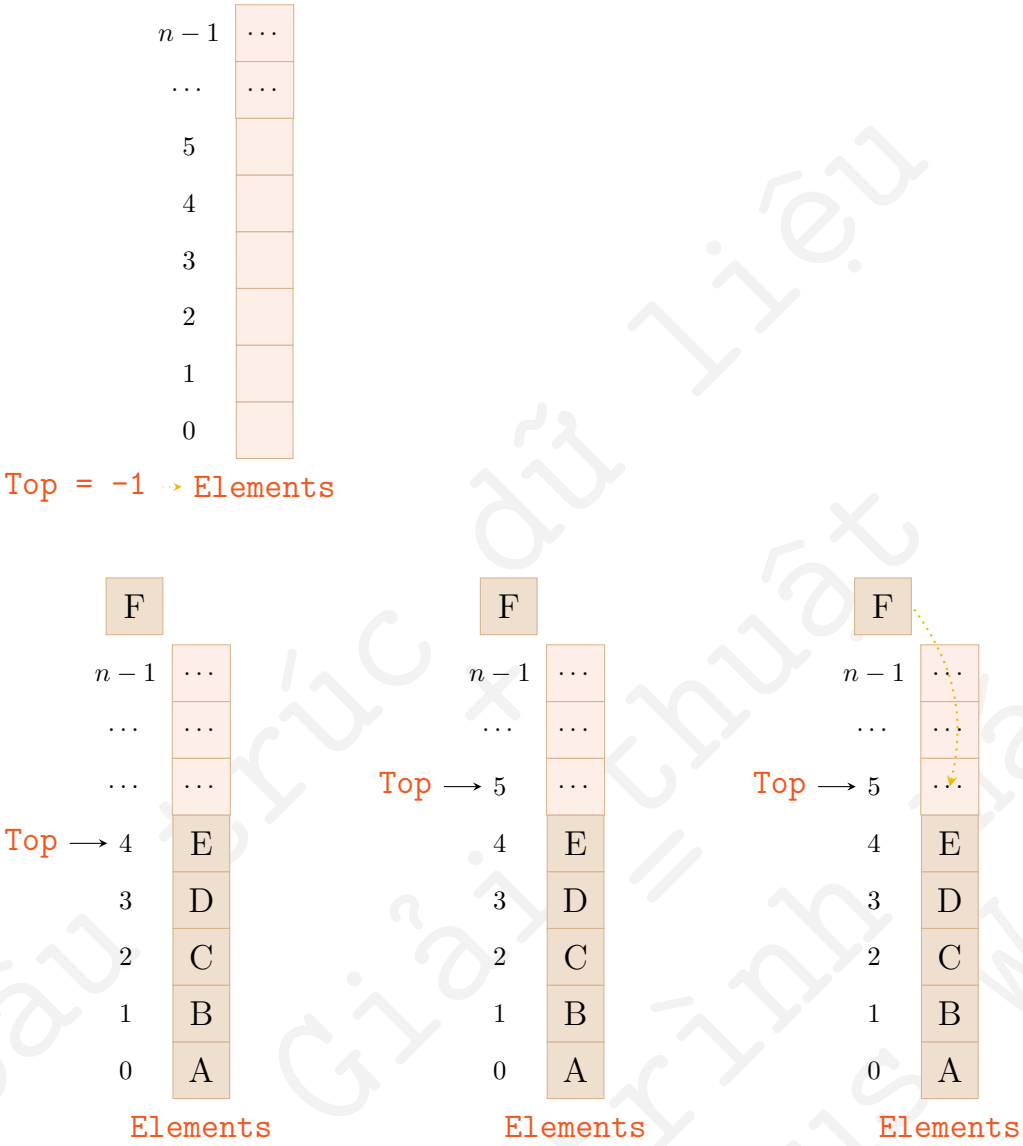
Thuật toán 8.1: IsEmpty(s)
- Đầu vào: ngăn xếp s.
- Đầu ra: true/false.

```
1      if top ≠ -1
2          return false
3      return true
```

Thuật toán 8.2: IsFull(s)
- Đầu vào: ngăn xếp s.
- Đầu ra: true/false.

```
1      if top  ≠ MAX_SIZE - 1
2          return false
3      return true
```

- Push là một trong hai thao tác cơ bản của ngăn xếp. Thao tác này chỉ thực hiện tại vị trí đỉnh của ngăn xếp.



Thuật toán 8.3: `Push(s, x)`

```
- Đầu vào: ngăn xếp s và phần tử x cần thêm.
- Đầu ra: ngăn xếp s sau khi thêm x.
```

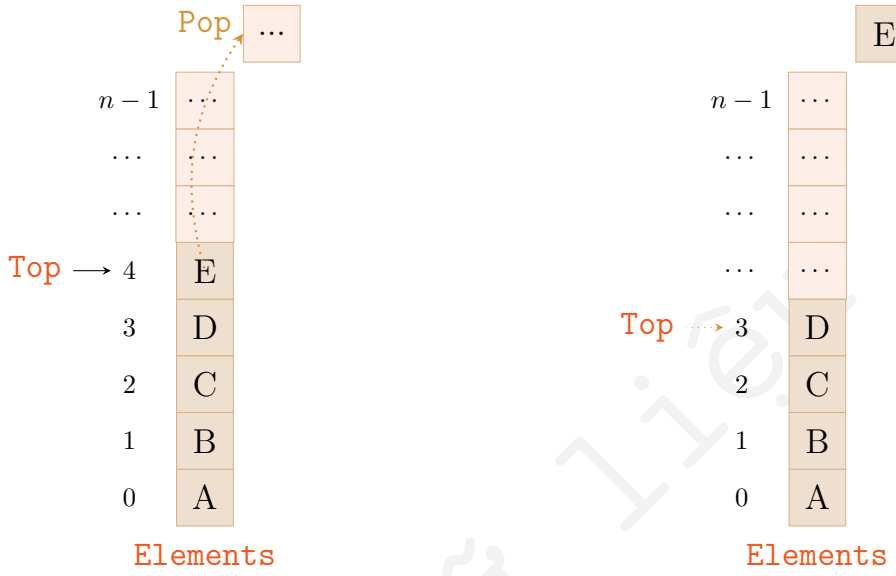
```
1    if ngăn xếp chưa đầy
2        top ← top + 1
3    elements[top] ← x
```

- Tương tự, thao tác Push cũng lấy và xóa phần tử tại đỉnh của ngăn xếp.

Thuật toán 8.4: `Pop(s)`

```
- Đầu vào: ngăn xếp s.
- Đầu ra: lấy ra và xóa phần tử khỏi đỉnh ngăn xếp
```

```
1    if ngăn xếp khác rỗng
2        x ← elements[top]
```




```
3           top ← top - 1 // Pop(s) ≠ GetTop(s)
4           return x
5       return STACK_EMPTY
```

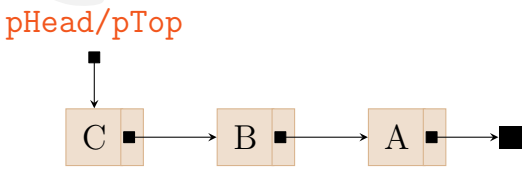
- Trong trường hợp chỉ cần lấy thông tin phần tử đỉnh mà không cần xóa khỏi ngăn xếp, chúng ta có thể sử dụng hàm sau đây:

Thuật toán 8.5: GetTop(s)
- Đầu vào: ngăn xếp s.
- Đầu ra: lấy ra phần tử đỉnh ngăn xếp

```
1 if ngăn xếp khác rỗng
2     x ← elements[top]
3     return x
4 return STACK_EMPTY
```

8.2.2 Cài đặt ngăn xếp bằng danh sách liên kết

- Cấu trúc dữ liệu một phần tử của ngăn xếp chứa thành phần dữ liệu và thành phần liên kết *(tương tự danh sách liên kết)*.
- Cấu trúc dữ liệu ngăn xếp chứa một con trỏ `pHead/pTop` trỏ đến phần tử *đầu/đỉnh* của ngăn xếp.
- Các thao tác thêm, xóa phần tử thực hiện ở *đầu/đỉnh* ngăn xếp.

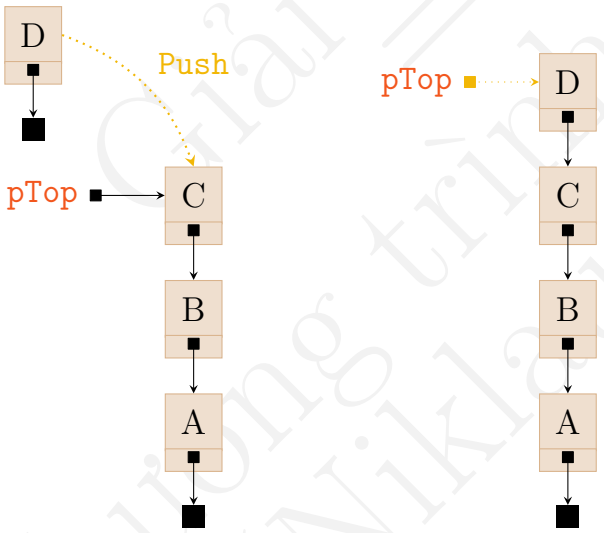


Cấu trúc của một phần tử trong ngăn xếp và hàm khởi tạo một nút trong ngăn xếp.

```
public class Node
{
    public int info;
    public Node pNext;

    public void InitNode(int x)
    {
        info = x;
        pNext = null;
    }
}
```

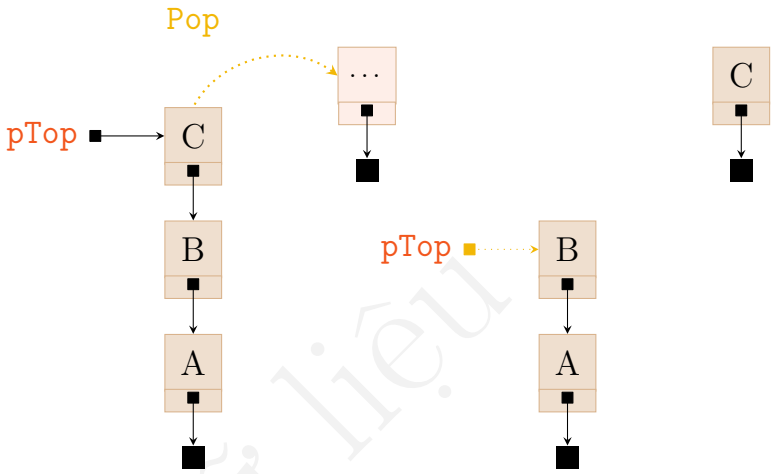
```
public class Stack
{
    public Node pTop;
    public int size;

    public void InitStack()
    {
        pTop = null;
        size = 0;
    }
}
```

Thao tác thêm phần tử




```
// Push phần tử vào đỉnh ngăn xếp
// tương tự thao tác thêm đầu danh sách liên kết (hàm InsertHead)
public void Push(Node p)
{
    if (p == null)
        return;
    p.pNext = pTop;
    pTop = p;
    size++;
}
```

Thao tác lấy phần tử




```
1  // Pop lấy ra phần tử từ đỉnh ngăn xếp và xóa khỏi ngăn xếp
2  // tương tự thao tác xóa đầu danh sách liên kết (RemoveHead)
3  public Node Pop()
4  {
5      if (pTop == null)
6          return STACK_EMPTY;
7      Node p = pTop;
8      int x = p.info;
9      pTop = pTop.pNext;
10     size--;
11     p = null; // delete p
12     return x;
13 }
```

```
1  // GetTop lấy phần tử tại đỉnh ngăn xếp, nhưng không xóa khỏi ngăn xếp
2  public int GetTop()
3  {
4      if (pTop == null)
5          return STACK_EMPTY;
6      return pTop.info;
7  }
```

8.3 Ứng dụng của ngăn xếp

Một số ứng dụng của ngăn xếp

- Sử dụng để khử đệ quy.
- Trong trình biên dịch, ngăn xếp được dùng để lưu trữ các thủ tục, biến, ...
- Tính giá trị biểu thức *(sử dụng ký pháp Ba Lan ngược)*.
- ...

8.3.1 Chuyển từ biểu thức dạng trung tố sang hậu tố

Biểu thức có thể được biểu diễn nhiều hình thức phụ thuộc vào thứ tự toán tử *(operator)* đối với các toán hạng *(operand)* như:

- Trung tố *(Infix)*
  Cú pháp:

  toán hạng toán tử toán hạng

- Hậu tố *(Postfix)* hay ký pháp Ba Lan ngược *(RPN-Reverse Polish notation)*
  Cú pháp:

  toán hạng toán hạng toán tử

Ví dụ 8.3.1. *Xét biểu thức* $x + y - z$

- *Biểu diễn dạng trung tố*
  - $x + (y - z)$
  - $(x + y) - z$
- *Biểu diễn dạng hậu tố*
  - $x\ y\ z - +$ $\Leftrightarrow$ $x + (y - z)$
  - $x\ y + z\ -$ $\Leftrightarrow$ $(x + y) - z$

Theo phương pháp Ký pháp Balan ngược, thao tác tính giá trị biểu thức thực hiện theo hai bước:

1. Chuyển từ trung tố sang hậu tố.
2. Tính giá trị biểu thức hậu tố.

Chuyển từ trung tố sang hậu tố

Thuật toán 8.6: `ConvertRPN(infix)`
- Đầu vào: infix (biểu thức trung tố)
- Đầu ra: postfix (biểu thức hậu tố)

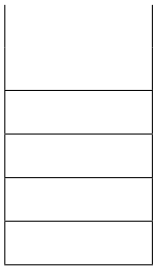
```
1   // Khoi tao stack rong
2   stack ← ∅
3
4   // Duyet infix: trai → phai
5   while infix ≠ ∅
6       // Doc 1 ky tu trong infix
7       read x from infix
8
9       // TH1: dau '('
10      if x = '('
11          Push(stack, x)
12      // TH2: dau ')'
```

```
13          else if x = ')'
14              y ← Pop(stack)
15              // Pop den khi gap dau '('
16              while y ≠ '('
17                  write y to postfix
18                  y ← Pop(stack)
19          // TH3: toan tu (+, -, *, /, ...)
20          else if x = operator
21              // Xet do uu tien tat ca toan tu trong stack
22              while GetPriority(GetTop(stack)) ≥ GetPriority(x)
23                  y ← Pop(stack)
24                  write y to postfix
25              Push(stack, x)
26          // TH4: toan hang (a, B, 1, ...)
27          else // x = operand
28          write x to postfix
29
30      // Lay ra tat ca ky tu trong stack
31      while stack ≠ ∅
32          y ← Pop(stack)
33          write y to postfix
```

Ví dụ 8.3.2. *Cho biểu thức* $2 * (7 + 3) - 8$. *Hãy áp dụng phương pháp Ký pháp Balan ngược biểu diễn biểu thức lại dạng hậu tố.*

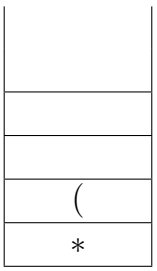
Ký tự	Trường hợp	Infix	Postfix
		$2 * (7 + 3) - 8$	

Ký tự	Trường hợp	Infix	Postfix
2	TH4	$* (7 + 3) - 8$	2

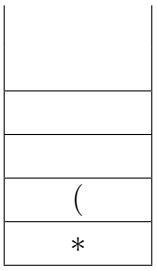


Ký tự	Trường hợp	Infix	Postfix
2	TH4	$* (7 + 3) - 8$	2
$*$	TH3	$(7 + 3) - 8$	2

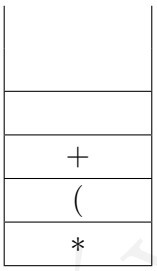
Ký tự	Trường hợp	Infix	Postfix
2	TH4	$*\ (7+3)-8$	2
$*$	TH3	$(7+3)-8$	2
(	TH1	$7+3)-8$	2



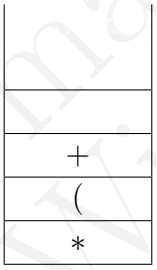
Ký tự	Trường hợp	Infix	Postfix
2	TH4	$*\ (7+3)-8$	2
$*$	TH3	$(7+3)-8$	2
(	TH1	$7+3)-8$	2
7	TH4	$+\ 3)-8$	2 7



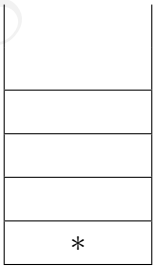
Ký tự	Trường hợp	Infix	Postfix
2	TH4	$*\ (7+3)-8$	2
$*$	TH3	$(7+3)-8$	2
(	TH1	$7+3)-8$	2
7	TH4	$+\ 3)-8$	2 7
$+$	TH3	$3)-8$	2 7



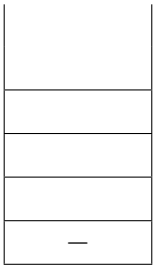
Ký tự	Trường hợp	Infix	Postfix
2	TH4	$*\ (7+3)-8$	2
$*$	TH3	$(7+3)-8$	2
(	TH1	$7+3)-8$	2
7	TH4	$+\ 3)-8$	2 7
$+$	TH3	$3)-8$	2 7
3	TH4	$)-8$	2 7 3



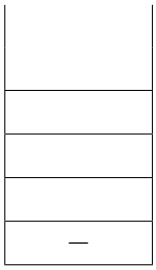
Ký tự	Trường hợp	Infix	Postfix
2	TH4	$*\ (7+3)-8$	2
$*$	TH3	$(7+3)-8$	2
(	TH1	$7+3)-8$	2
7	TH4	$+\ 3)-8$	2 7
$+$	TH3	$3)-8$	2 7
3	TH4	$)-8$	2 7 3
)	TH2	$-\ 8$	2 7 3 $+$



Ký tự	Trường hợp	Infix	Postfix
2	TH4	$*\ (7+3)-8$	2
$*$	TH3	$(7+3)-8$	2
(	TH1	$7+3)-8$	2
7	TH4	$+\ 3)-8$	2 7
$+$	TH3	$3)-8$	2 7
3	TH4	$)-8$	2 7 3
)	TH2	$-\ 8$	2 7 3 $+$
$-$	TH3	8	2 7 3 $+$ $*$



Ký tự	Trường hợp	Infix	Postfix
2	TH4	$*\ (7+3)-8$	2
$*$	TH3	$(7+3)-8$	2
(	TH1	$7+3)-8$	2
7	TH4	$+\ 3)-8$	2 7
$+$	TH3	$3)-8$	2 7
3	TH4	$)-8$	2 7 3
)	TH2	$-\ 8$	2 7 3 $+$
$-$	TH3	8	2 7 3 $+$ $*$
8	TH4		2 7 3 $+$ $*$ 8



Ký tự	Trường hợp	Infix	Postfix
2	TH4	$*\ (7+3)-8$	2
$*$	TH3	$(7+3)-8$	2
(	TH1	$7+3)-8$	2
7	TH4	$+\ 3)-8$	2 7
$+$	TH3	$3)-8$	2 7
3	TH4	$)-8$	2 7 3
)	TH2	$-\ 8$	2 7 3 $+$
$-$	TH3	8	2 7 3 $+$ $*$
8	TH4		2 7 3 $+$ $*$ 8
			2 7 3 $+$ $*$ 8 $-$

Ví dụ 8.3.3. *Cho các biểu thức dưới dạng trung tố, hãy áp dụng Ký pháp Balan ngược chuyển từ trung tố sang hậu tố và tính kết quả cho biểu thức vừa tìm được.*

1. $1+2*(9-3)/6$
2. $3*((6+5)-9)$

Kết quả

1.
   - Stack: .....
   - Biểu thức dạng trung tố: .....
   - Biểu thức dạng hậu tố: .....
   - ...
2.
   - Stack: .....
   - Biểu thức dạng trung tố: .....
   - Biểu thức dạng hậu tố: .....
   - ...

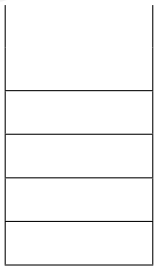
8.3.2 Tính biểu thức dạng hậu tố

Thuật toán 8.7: ComputeRPN(postfix)
- Đầu vào: postfix (biểu thức hậu tố)
- Đầu ra: giá trị biểu thức

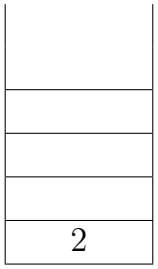
```
1   // Khoi tao stack rong
2   stack ← ∅
3
4   // Duyet postfix: trai → phai
5   while postfix ≠ ∅
6   // Doc 1 ky tu trong postfix
7   read x from postfix
8   // TH1: toan hang
9   if x = operand
10      Push(stack, x)
11  // TH2: toan tu (+, -, *, /, ...)
12  else
13      x2 ← Pop(stack)
14      x1 ← Pop(stack)
15      // x1, x2: operand; x: operator
16      y ←Calculate(x1, x2, x) // y ← x1 x x2
17      Push(stack, y)
18  return Pop(stack)
```

Ví dụ 8.3.4. *Cho biểu thức* 2 7 3 + ∗ 8 −. *Tính biểu thức dạng hậu tố.*

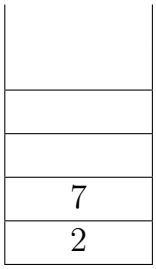
Ký tự	Trường hợp	Postfix
		2 7 3 + ∗ 8 −



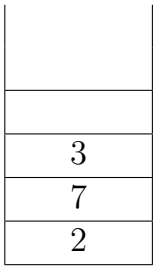
Ký tự	Trường hợp	Postfix
2	TH1	7 3 + ∗ 8 −



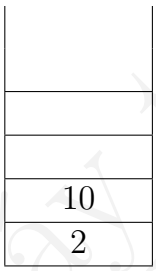
Ký tự	Trường hợp	Postfix
2	TH1	7 3 + ∗ 8 −
7	TH1	3 + ∗ 8 −



Ký tự	Trường hợp	Postfix
2	TH1	7 3 + ∗ 8 −
7	TH1	3 + ∗ 8 −
3	TH1	+ ∗ 8 −

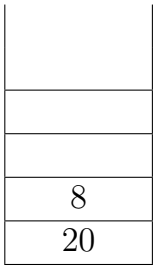


Ký tự	Trường hợp	Postfix
2	TH1	7 3 + ∗ 8 −
7	TH1	3 + ∗ 8 −
3	TH1	+ ∗ 8 −
+	TH2	∗ 8 −

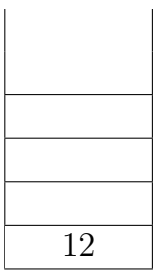


Ký tự	Trường hợp	Postfix
2	TH1	7 3 + ∗ 8 −
7	TH1	3 + ∗ 8 −
3	TH1	+ ∗ 8 −
+	TH2	∗ 8 −
∗	TH2	8 −

Ký tự	Trường hợp	Postfix
2	TH1	7 3 + ∗ 8 −
7	TH1	3 + ∗ 8 −
3	TH1	+ ∗ 8 −
+	TH2	∗ 8 −
∗	TH2	8 −
8	TH1	−



Ký tự	Trường hợp	Postfix
2	TH1	7 3 + ∗ 8 −
7	TH1	3 + ∗ 8 −
3	TH1	+ ∗ 8 −
+	TH2	∗ 8 −
∗	TH2	8 −
8	TH1	−
−	TH2	



8.4 Giới thiệu hàng đợi

Khái niệm

Hàng đợi (Queue) là một cấu trúc dữ liệu ngược với cấu trúc Ngăn xếp.

- Thực hiện theo cơ chế FIFO *(First In, First Out)* vào trước ra trước.
- Dùng để lưu trữ các phần tử có thứ tự truy xuất *đúng* với thứ tự lưu trữ *(vào trước, ra trước)*.

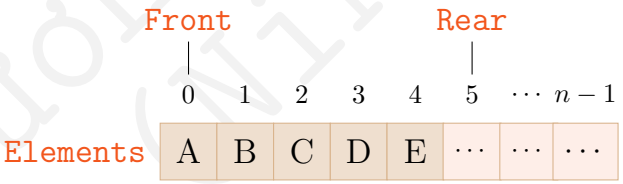
Hình 8.8: Hình minh họa hàng đợi.

- EnQueue: *thêm* phần tử vào *cuối* hàng đợi.
- DeQueue: *lấy và xóa* phần tử tại *đầu* hàng đợi.
- GetFront: *xem thông tin* phần tử tại *đầu* hàng đợi.
- Kiểm tra hàng đợi rỗng, đầy.

8.5 Cài đặt hàng đợi

8.5.1 Cài đặt hàng đợi bằng mảng

- Biến `Elements` là mảng 1 chiều kích thước n: lưu trữ phần tử từ vị trí $[0, ..., n-1]$.
- Biến `Front`, `Rear` kiểu số nguyên: cho biết vị trí đầu và cuối.
- Mặc định, hàng đợi vừa khởi tạo `Front = Rear = 0` và tất cả phần tử của mảng `Elements` gán bằng `QUEUE_EMPTY`.




```
1 public class Queue
2 {
3     public int[] elements;
4     public int front;
5     public int rear;
6
```

```
7       public void InitQueue()
8       {
9           elements = new int[MAX_SIZE];
10          front = rear = 0;
11      }
12  }
```

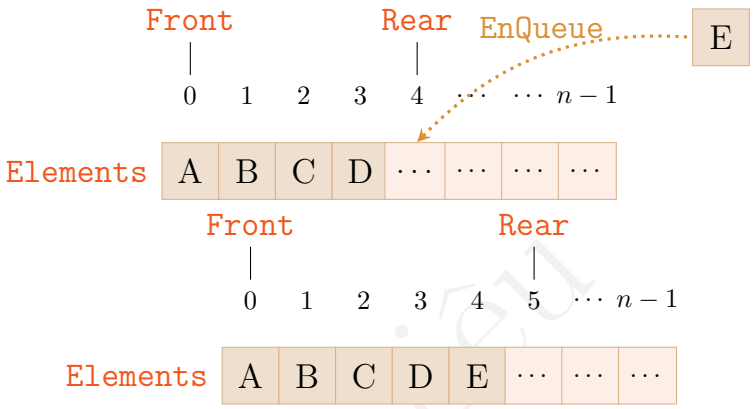
Thuật toán 8.8: IsEmpty(q)
- Đầu vào: hàng đợi q.
- Đầu ra: true/false.

```
1       if elements[front] ≠ QUEUE_EMPTY
2           return false
3       return true
```

Thuật toán 8.9: IsFull(q)
- Đầu vào: hàng đợi q.
- Đầu ra: true/false.

```
1       if elements[rear] = QUEUE_EMPTY
2           return false
3       return true
```

- Đối với hàng đợi, thao tác thêm một phần tử tại vị trí cuối cùng.



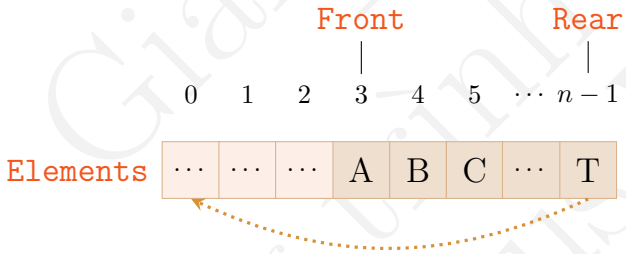
Thuật toán 8.10: `EnQueue(q, x)`

- Đầu vào: hàng đợi `q` và phần tử `x` cần thêm.
- Đầu ra: hàng đợi `q` sau khi thêm `x`.

```
1   if hàng đợi chưa đầy
2       elements[rear] ← x
3       rear ← rear + 1
4   if rear = MAX_SIZE
5       rear ← 0
```

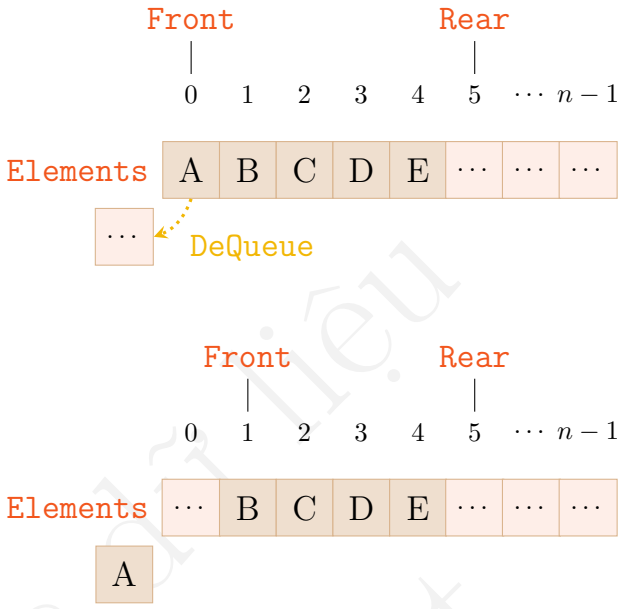
Xử lý vấn đề tràn giả

Sử dụng mảng như danh sách vòng.



- EnQueue: nếu đến cuối mảng, cập nhật Rear = 0.

- Ngược với ngăn xếp, thao tác lấy và xóa một phần tử tại vị trí đầu tiên.

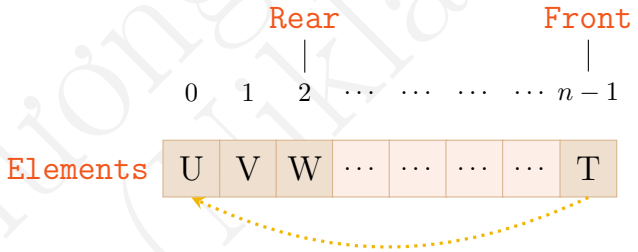


Thuật toán 8.11: DeQueue(q)
- Đầu vào: hàng đợi q.
- Đầu ra: phần tử đầu hàng đợi hay QUEUE_EMPTY (hàng đợi rỗng).

```
1  if hàng đợi khác rỗng
2      x ← elements[front]
3      elements[front] ← QUEUE_EMPTY
4      front ← front + 1
5  if front = MAX_SIZE
6      front ← 0
7      return x
8  return QUEUE_EMPTY
```

Xử lý vấn đề tràn giả

Sử dụng mảng như danh sách vòng.



- DeQueue: nếu đến cuối mảng, cập nhật Front = 0.
- Trong trường hợp chỉ cần lấy thông tin phần tử đầu hàng đợi, chúng ta dùng hàm sau:

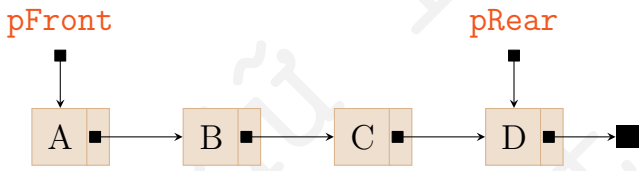
Thuật toán 8.12: GetFront(q)

```
1  if hàng đợi khác rỗng
```

```
2           return elements[front]
3       return QUEUE_EMPTY
```

8.5.2 Cài đặt hàng đợi bằng danh sách liên kết

- Cấu trúc dữ liệu một phần tử của hàng đợi chứa thành phần dữ liệu và thành phần liên kết *(tương tự danh sách liên kết)*.
- Cấu trúc dữ liệu hàng đợi chứa hai con trỏ `pFront` trỏ đến phần tử *đầu* và con trỏ `pRear` trỏ đến phần tử *cuối* của hàng đợi.
- Thao tác thêm thực hiện ở cuối và thao tác xóa thực hiện ở đầu hàng đợi.



Cấu trúc của một phần tử trong hàng đợi và hàm khởi tạo một nút trong hàng đợi.

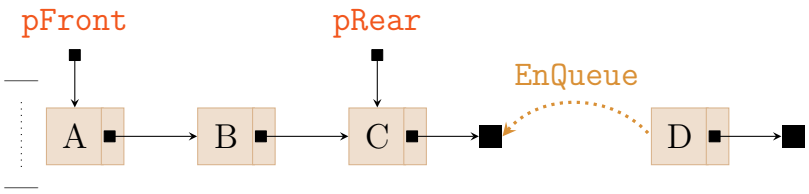
```
1  public class Node
2  {
3      public int info;
4      public Node pNext;
5
6      public void InitNode(int x)
7      {
8          info = x;
9          pNext = null;
10     }
11 }
```

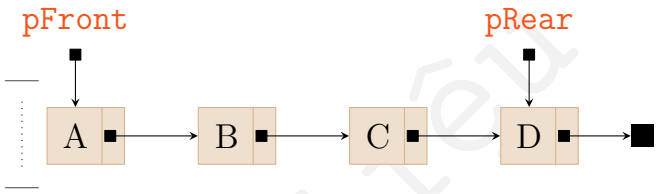
Cấu trúc của một hàng đợi và hàm khởi tạo hàng đợi

```
1  public class Queue
2  {
3      public Node pFront;
4      public Node pRear;
5      public int size;
6
7      public void InitQueue()
8      {
9          pFront = null;
10         pRear = null;
11         size = 0;
12     }
13 }
```

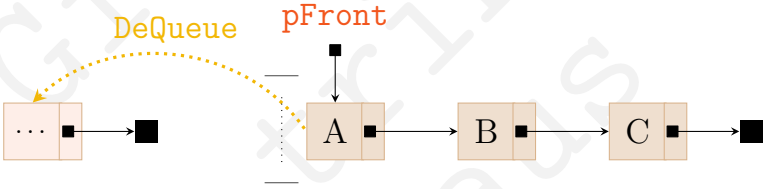
Thao tác thêm phần tử

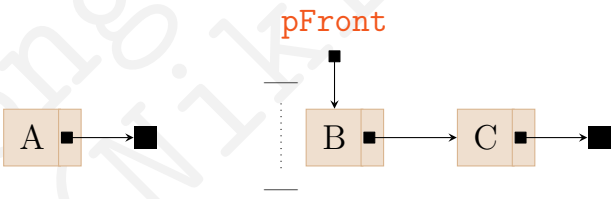
```
1  // Thêm phần tử vào hàng đợi (thêm cuối)
2  public void EnQueue(Node p)
```






```
3  {
4      if (p == null)
5      return;
6      if (pFront == null)
7      {
8          pFront = p;
9          pRear = p;
10     }
11     else
12     {
13         pRear.pNext = p;
14         pRear = p;
15     }
16     size++;
17 }
```





Thao tác lấy phần tử

```
1  // Lấy phần tử ra khỏi đầu hàng đợi (xóa đầu)
2  public int DeQueue()
3  {
4      if (pFront == null)
5          return QUEUE_EMPTY;
6      Node p = pFront;
```

```
7       int x = p.info;
8       pFront = pFront.pNext;
9       size--;
10      p = null; // delete p
11      return x;
12  }
```

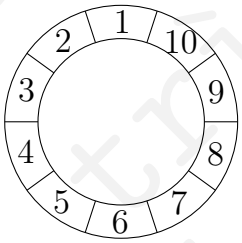
```
1   public int GetFront()
2   {
3       if (pFront == null)
4           return QUEUE_EMPTY;
5       return pFront.info;
6   }
```

8.6 Ứng dụng của hàng đợi

Một số ứng dụng của hàng đợi

- Trong một số thuật toán của lý thuyết đồ thị, hàng đợi được sử dụng để lưu dữ liệu khi thực hiện.
- Bài toán sản xuất và tiêu thụ.
- Quản lý bộ đệm (ví dụ: nhấn phím → bộ đệm → CPU xử lý).
- Xử lý các lệnh/tiến trình trong máy tính (ví dụ: hàng đợi máy in)

Ví dụ 8.6.1. *Bài toán Josephus mô tả như sau: cho n người đứng thành vòng tròn và một số nguyên m, với $m < n$.*



- *Bắt đầu vị trí s, bài toán sẽ đếm từng người theo một chiều nhất định. Sau khi có $m-1$ người được bỏ qua, người thứ m sẽ bị xử tử.*
- *Quy luật lặp lại đến khi còn $m-1$ người sống sót.*

Giả sử $n = 10$ và $m = 3$. Hãy cho biết $m-1$ người còn sống đứng vị trí nào?

Thuật toán 8.13: Josephus(n, m)
- Đầu vào: n là số người và m là một số nguyên
- Đầu ra: in ra thứ tự người bị xử tử

```
1       for i ← 1 to n
2           EnQueue(q, i)
3
```

```
4       while q ≠ ∅
5           for i ← 1 to m - 1
6               EnQueue(q, DeQueue(q))
7               x ← DeQueue(q)
8               Print x
```

Giải thích

- Dòng 1 → 2: đưa tất cả người tham gia vào hàng đợi.
- Dòng 4 → 7: hàng đợi khác rỗng, bắt đầu đếm và thực hiện
  - Dòng 5 → 6: đưa $m-1$ người vào hàng đợi.
  - Dòng 7: chọn người vị trí m.
- Dòng 8: in thứ tự người bị chọn (trong đó, hai người ở vị trí cuối cùng sẽ sống sót)

8.7 Bài tập cuối chương

1. Cho một ngăn xếp rỗng, hãy lần lượt thực hiện các thao tác sau đây: push(1), push(5), push(2), push(7), pop(), pop(), push(9). Hãy vẽ ngăn xếp tương ứng với các thao tác trên.
2. Áp dụng ngăn xếp để thực hiện thao tác chuyển một số hệ thập phân *(cơ số 10)* sang hệ nhị phân *(cơ số 2)*.
3. Xét bài toán dùng chuyển biểu thức từ trung tố sang hậu tố bằng cấu trúc ngăn xếp. Nếu sau khi đọc 5 ký tự, ngăn xếp chứa lần lượt các ký tự: -, (, +. Hãy cho biết ký tự thứ 6 đọc được sẽ có kiểu là gì: toán tử, toán hạng, ngoặc đóng hay ngoặc mở? Giải thích.
4. Giả sử cài đặt hàng đợi bằng 2 ngăn xếp inStack và outStack. Hãy vẽ 2 ngăn xếp tương ứng với các thao tác EnQueue(1), EnQueue(5), EnQueue(2), EnQueue(7), DeQueue(), DeQueue(), EnQueue(9) của hàng đợi.
5. Áp dụng hàng đợi viết hàm cài đặt thuật toán Josephus.

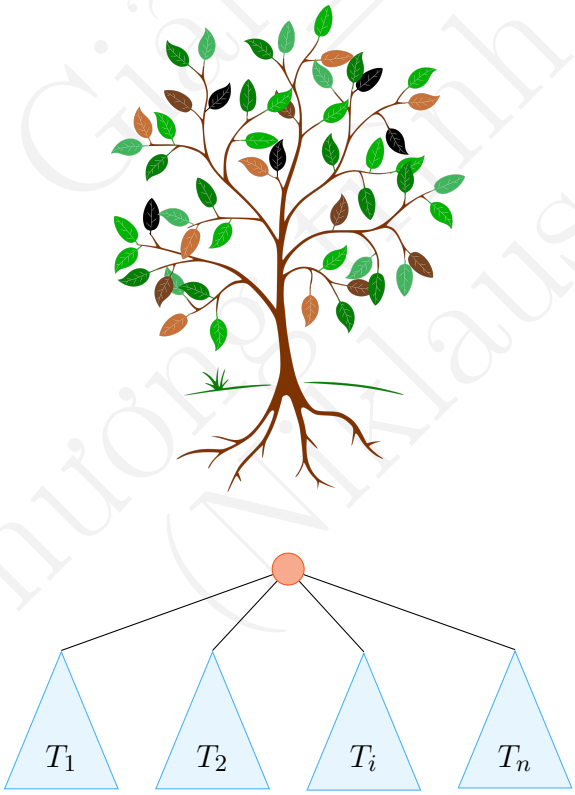
Bài 9

CẤU TRÚC CÂY (3 tiết)

9.1 Giới thiệu cấu trúc cây

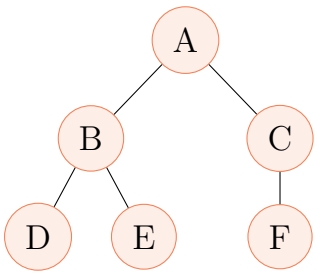
Các khái niệm

- Cây *(Tree)* là một tập hợp T gồm các nút *(phần tử)* của cây.
- Trong đó, có một nút đặc biệt được gọi là nút gốc *(root)*, các nút còn lại được chia thành những tập rời nhau $T_1, T_2, \dots T_n$ theo quan hệ phân cấp. Mỗi T_i cũng là một cây.



Ví dụ 9.1.1. *Cho cây biểu diễn dãy ký tự A, B, C, D, E, F.*

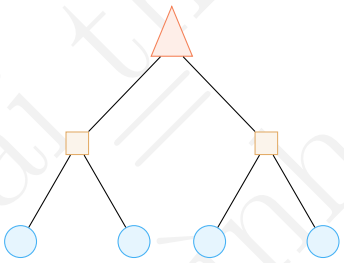
- Bậc *(degree)* của một nút: là số cây con của nút đó.



- Bậc của một cây: là bậc *lớn nhất* của các nút trong cây *(số cây con tối đa của một nút thuộc cây)*. Cây có bậc n thì gọi là cây n-phân.

(a) Bậc của cây T_1 là 2 *(cây nhị phân)*. (b) Bậc của cây T_2 là 3.

Hình 9.1: Bậc của cây.



- Nút *(node)*: là một phần tử trong cây.
- Nút cha *(parent)*.
- Nút con *(child)*.
- Nút anh em *(sibling)*: là những nút có cùng nút cha.
- Nút gốc *(root)*: là nút không có nút cha.
- Nút trong *(internal)*: là nút có bậc khác 0 và không phải là nút gốc.
- Nút lá *(leaf)*: là nút có bậc bằng 0.

Mức *(level)* hay chiều sâu *(depth)* của một nút

- Nếu p là nút gốc, thì
$$d(p) = 0.$$
- Ngược lại,
$$d(p) = d(parent(p)) + 1.$$

Chiều cao *(height)* của một nút

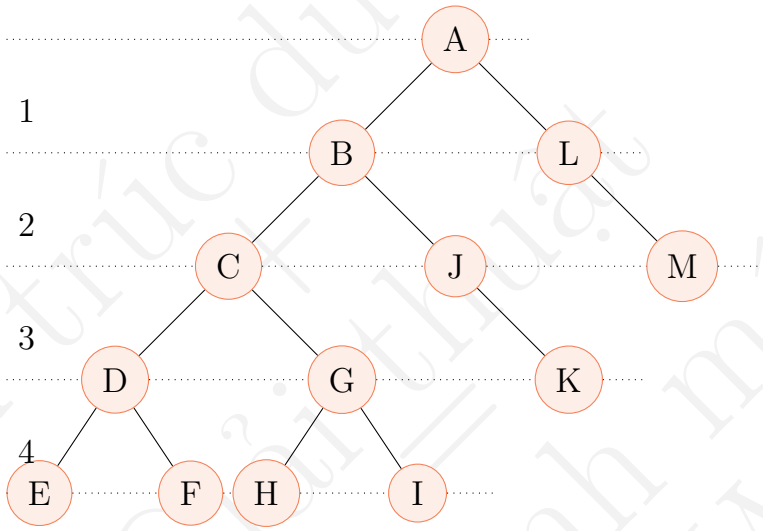
Là độ dài đường đi *(path)* hay số cạnh *(edge)* lớn nhất từ nút đó đến nút lá.

- Nếu p là nút lá, thì
$$height\,(p) = 0.$$
- Ngược lại,
$$height\,(p) = max\,\{height\,(T_i)\} + 1.$$

với T_i là các cây con của nút p.

Chiều cao của cây chính là chiều cao của *nút gốc*.

Ví dụ 9.1.2. *Cho cấu trúc cây như hình, hãy trả lời các câu hỏi sau:*



- Bậc của nút A?
- Bậc của nút M ?
- Bậc của cây ?
- Mức của nút C ?
- Mức của nút H ?
- Chiều cao của cây ?

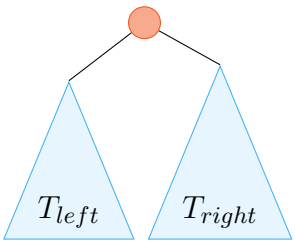
9.2 Giới thiệu cây nhị phân

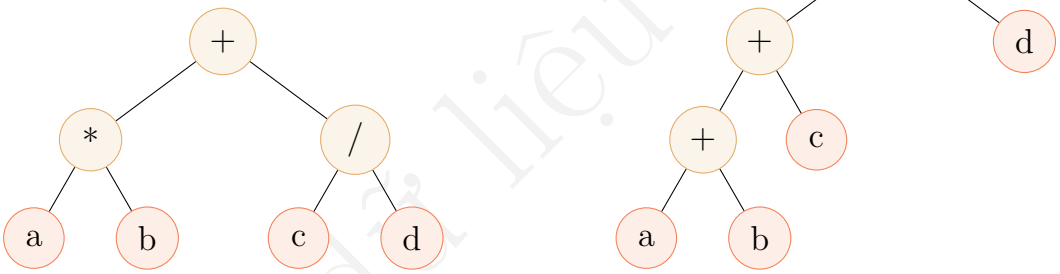
9.2.1 Các khái niệm về cây nhị phân

Khái niệm

Cây nhị phân *(binary tree)* là cây mà mỗi nút có *tối đa 2 cây con (cây con trái và cây con phải).*

Ví dụ 9.2.1. *Cây nhị phân mô tả biểu thức* $(a * b) + (c/d)$ và $(((a + b) + c) + d)$.



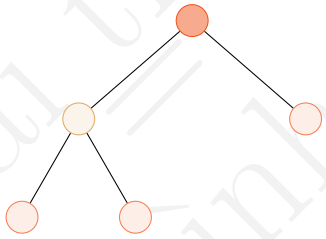


(a) Biểu thức $(a * b) + (c/d)$ (b) Biểu thức $(((a + b) + c) + d)$.

Hình 9.2: Biểu diễn biểu thức bằng cấu trúc cây.

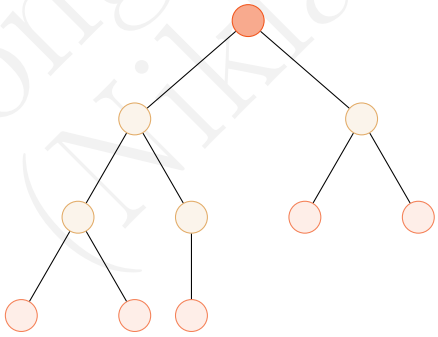
Full binary tree

- Là cây nhị phân mà mỗi nút trong *(internal)* có *đầy đủ 2 cây con*.



Complete binary tree

- Là cây nhị phân mà mỗi nút trong *(internal)* ở mức $d-1$ có *đầy đủ 2 cây con* và các cây con ở mức d được điền từ *trái sang phải*.

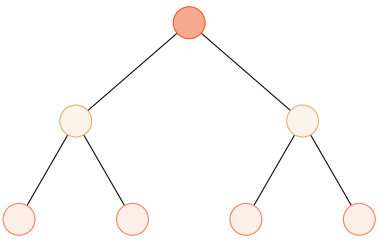


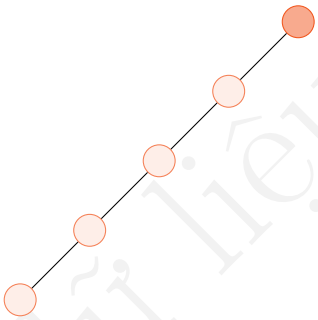
Perfect binary tree

- Là một *complete binary tree* mà tất cả nút lá *có cùng mức*.

Cây suy biến *(skewed tree)*

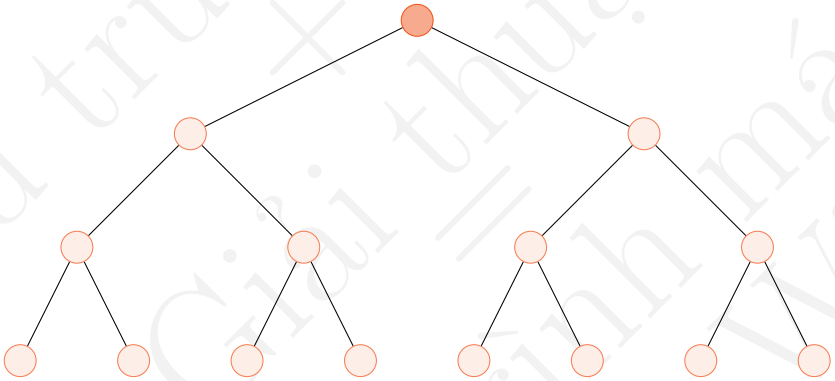
- Là cây nhị phân *bị lệch về một nhánh*.





9.2.2 Một số tính chất của cây nhị phân

Cho cây nhị phân T có chiều cao h



1. Số nút nằm ở mức d

$$n\left(d\right) \leq 2^d.$$

2. Số nút lá

$$n\left(h\right) \leq 2^{h-1}.$$

3. Số nút trong

$$n\left(h\right) \leq 2^h - 1.$$

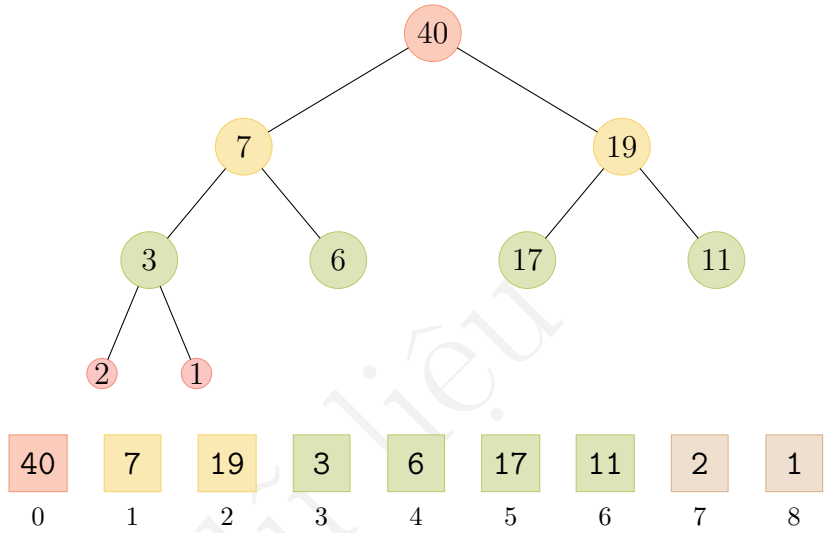
4. Tổng số nút

$$n\left(h\right) \leq 2^{h+1} - 1.$$

5. Chiều cao của cây

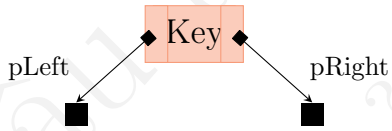
$$h \geq log_2\left(n+1\right) - 1 \approx log_2 n.$$

9.2.3 Cách lưu trữ cây nhị phân



Cấu trúc dữ liệu của một nút trong cây

- Thành phần dữ liệu: khóa *(key)* của một nút.
- Thành phần liên kết: con trỏ `pLeft` liên kết với *cây con trái* và con trỏ `pRight` liên kết với *cây con phải*.



- Chỉ cần một con trỏ trỏ đến nút gốc của cây.

Định nghĩa cấu trúc dữ liệu của một nút

```
class Node
{
    public int  Key;
    public Node pLeft;
    public Node pRight;
}
```

Khởi tạo giá trị của một nút gồm thành phần dữ liệu và 2 thành phần liên kết cây con trái, cây con phải

```
public static Node InitNode(int k)
{
    Node p = new Node();
    p.Key = k;
    p.pLeft = null;
    p.pRight = null;
    return p;
}
```

9.2.4 Các thao tác duyệt cây nhị phân

Duyệt cây

- Là thuật toán liệt kê danh sách tất cả các nút của một cây, mỗi nút chỉ một lần.
- Mỗi thuật toán duyệt cây khác nhau ở *thứ tự duyệt nút gốc* của *cây con đang xét*.
- Ba phương pháp duyệt cây:
  - Tiền thứ tự *(preorder)* hay **N**LR *(node-left-right)*.
  - Trung thứ tự *(inoder)* hay L**N**R *(left-node-right)*.
  - Hậu thứ tự *(postorder)* hay LR**N** *(left-right-node)*.

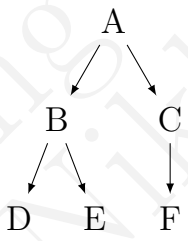
Thao tác duyệt

- Hàm duyệt cây chứa *hai lời gọi hàm đệ quy* tương ứng *cây con trái* và *cây con phải*.
- Ngoài ra, hàm duyệt sẽ chứa các thao tác xử lý tùy theo yêu cầu như: tìm kiếm, in, ... một nút.

Thuật toán 9.1: `Traverse(pRoot)`
- Đầu vào: cây nhị phân với gốc root.
- Đầu ra:

```
1 if cây con đang xét khác rỗng
2     Traverse(pRoot->pLeft) // Duyet cay con trai
3     Traverse(pRoot->pRight) // Duyet cay con phai
```

Ví dụ 9.2.2. *Duyệt cây T theo phương pháp tiền thứ tự, trung thứ tự và hậu thứ tự.*



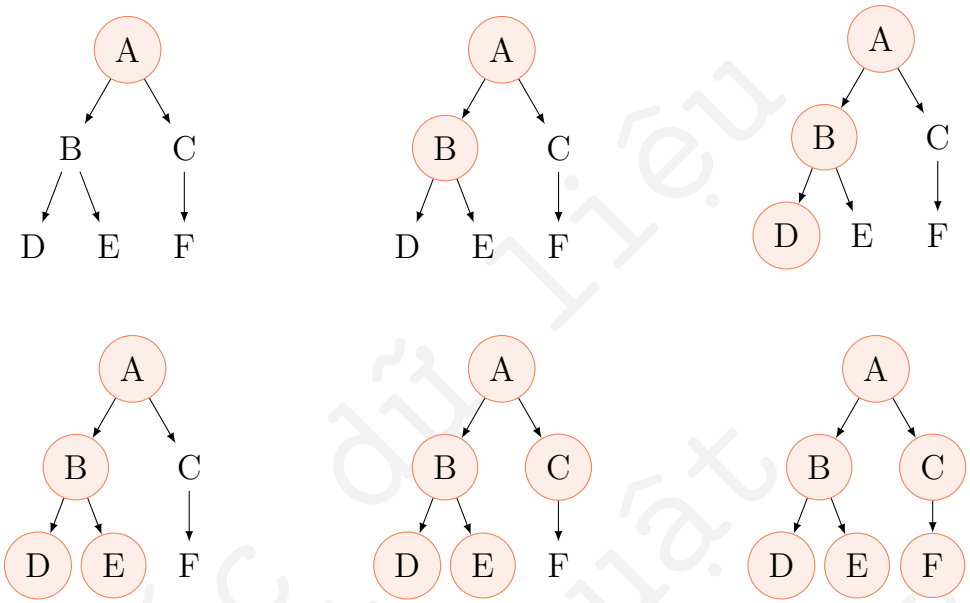
Hình 9.3: Cây T gồm 6 đỉnh A, B, C, D, E, F.

Duyệt tiền thứ tự *(NLR)*

```
1 public void NLR(ref Node root)
2 {
3     if (root != null)
4     {
5         // Thao tac xu ly nut dang xet ...
6         NLR(ref root.pLeft);
```

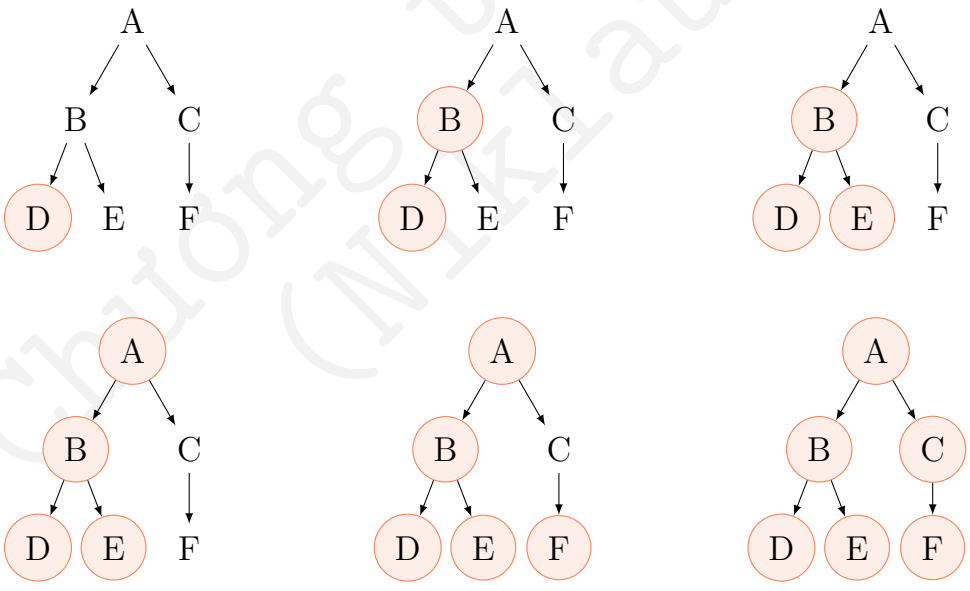
```
7           NLR(ref root.pRight);
8       }
9   }
```

Duyệt tiền thứ tự *(NLR)*



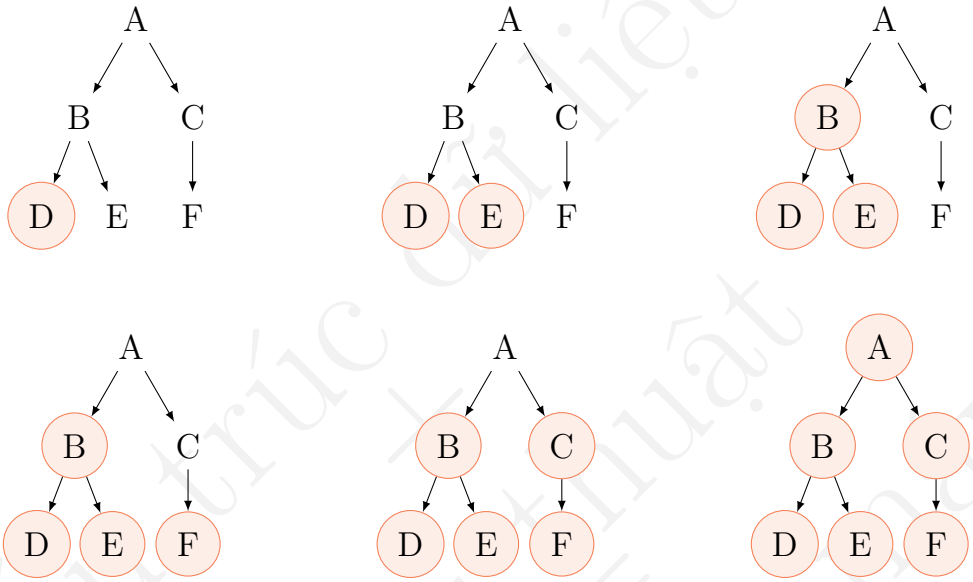
Duyệt trung thứ tự *(LNR)*

```
1   public void LNR(ref Node root)
2   {
3       if (root != null)
4       {
5           LNR(ref root.pLeft);
6           // Thao tac xu ly nut dang xet ...
7           LNR(ref root.pRight);
8       }
9   }
```



Duyệt hậu thứ tự *(LRN)*

```
1 public void LRN(ref Node root)
2 {
3     if (root != null)
4     {
5         LRN(ref root.pLeft);
6         LRN(ref root.pRight);
7         // Thao tac xu ly nut dang xet ...
8     }
9 }
```



9.3 Bài tập cuối chương

1. Cho dãy gồm các phần tử: 1, 2, 3, 4, 5, 6, 7.
   (a) Hãy vẽ cây NPTK sao cho *chiều cao nhỏ nhất có thể*.
   (b) Cho biết chiều cao của cây NPTK trên.
   (c) Duyệt cây theo thứ tự NLR, LNR, LRN.
   (d) Phương pháp duyệt nào cây NPTK có chiều cao nhỏ nhất có thể?
2. Cài đặt các hàm xử lý yêu cầu sau:
   (a) Tìm nút có khóa là k trong cây nhị phân..
   (b) Tính tổng giá trị các nút trong cây nhị phân.
   (c) Xác định số nút lá cây nhị phân.
   (d) Xác định chiều cao của cây nhị phân.

Bài 10

CÂY NHỊ PHÂN TÌM KIẾM (3 tiết)

10.1 Giới thiệu cây nhị phân tìm kiếm

Khái niệm

Cây nhị phân tìm kiếm - NPTK *(Binary Search Tree - BST)* là một *cây nhị phân* thỏa các điều kiện sau:

- Khóa của các nút thuộc cây con trái *nhỏ* hơn khóa nút gốc.
- Khóa của các nút thuộc cây con phải *lớn* hơn khóa nút gốc.
- Hai cây con trái và phải của nút gốc cũng là một cây NPTK.

Cây nhị phân tìm kiếm có một số đặc điểm sau:

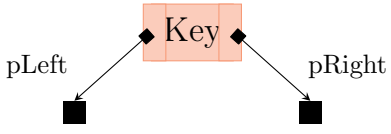
- Dữ liệu lưu trữ có thứ tự, hỗ trợ tìm kiếm tốt hơn danh sách liên kết, ngăn xếp, hàng đợi, ...
- Nếu tổng số nút trong cây NPTK là n thì chi phí tìm kiếm trung bình $log_2 n$.

10.2 Các thao tác trong cây NPTK

10.2.1 Thêm một nút vào NPTK

Trong cây NPTK, cấu trúc dữ liệu của một nút được định nghĩa bởi hai thành phần:

- Thành phần dữ liệu: khóa *(key)* của một nút.
- Thành phần liên kết: con trỏ `pLeft` liên kết với *cây con trái* và con trỏ `pRight` liên kết với *cây con phải*.



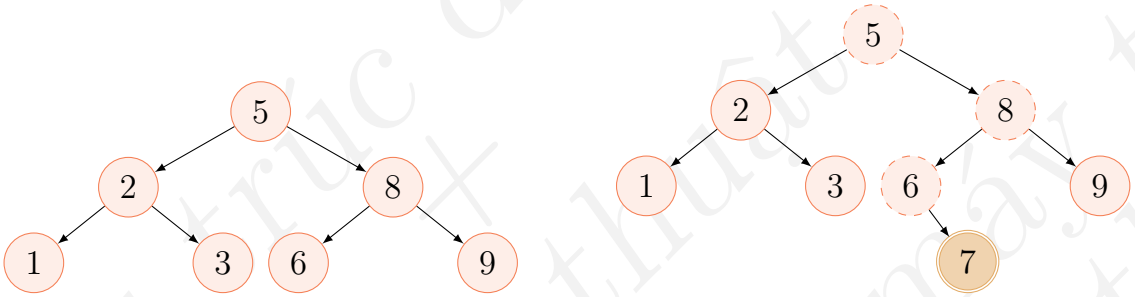
Đối với cây NPTK, cấu trúc dữ liệu chỉ cần lưu trữ thông tin về nút gốc của cây.

- Chỉ cần một con trỏ trỏ đến nút gốc của cây.

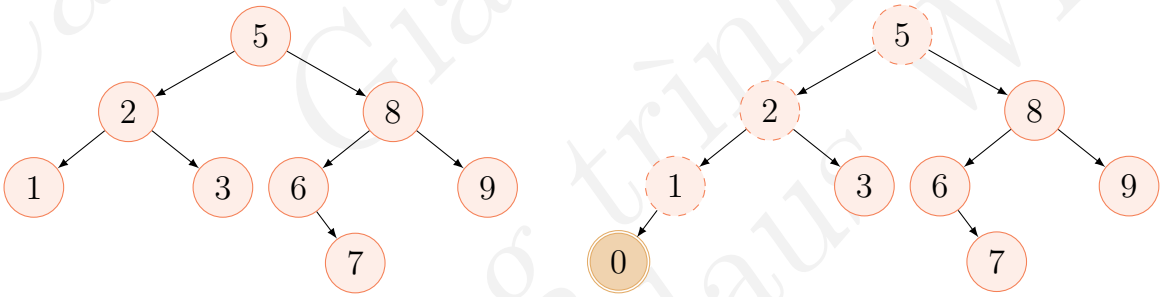
Thuật toán 10.1: InsertNode(t, k)
- Đầu vào: cây T và khóa k cần thêm.
- Đầu ra: cây T sau khi thêm k.

```
1  if cây rỗng // TH1. cay rong
2      Khởi tạo nút p có khóa k và cập nhật pLeft, pRight
3      pRoot trỏ đến p
4  // TH2. cay khac rong
5  else if pRoot->Key > k
6          Gọi đệ quy hàm InsertNode() với cây con bên trái
7  else if pRoot->Key < k
8      Gọi đệ quy hàm InsertNode() với cây con bên phải
```

Ví dụ 10.2.1. *Cho cây NPTK như hình sau, hãy thêm nút có khóa là 7 vào cây.*



Ví dụ 10.2.2. *Cho cây NPTK như hình sau, hãy thêm nút có khóa là 0 vào cây.*

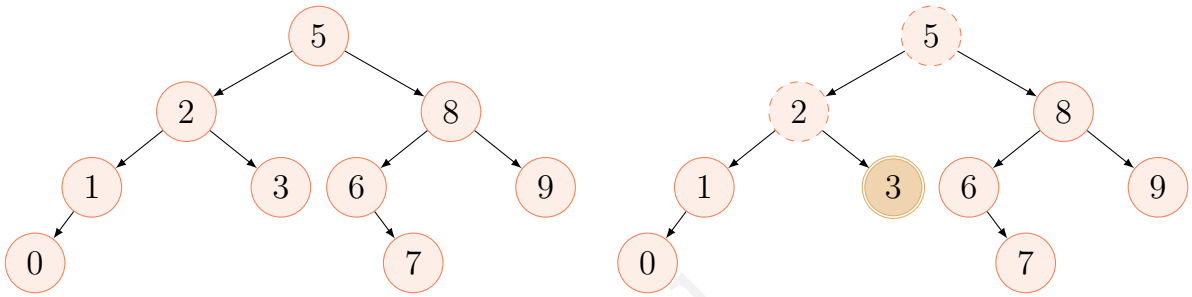


10.2.2 Thao tác tìm nút có khóa k trong cây NPTK

Thuật toán 10.2: SearchNode(t, k)
- Đầu vào: cây T và nút k cần tìm.
- Đầu ra: nút có khóa k hay NULL nếu không tìm thấy.

```
1  if cây khác rỗng
2      if pRoot->pKey = k
3          Trả về nút pRoot
4      else if pRoot->pKey > k
5          Gọi đệ quy hàm SearchNode() với cây con bên trái
6      else
7          Gọi đệ quy hàm SearchNode() với cây con bên phải
8  Trả về không tìm thấy
```

Ví dụ 10.2.3. *Cho cây NPTK, hãy tìm nút có khóa là 3.*

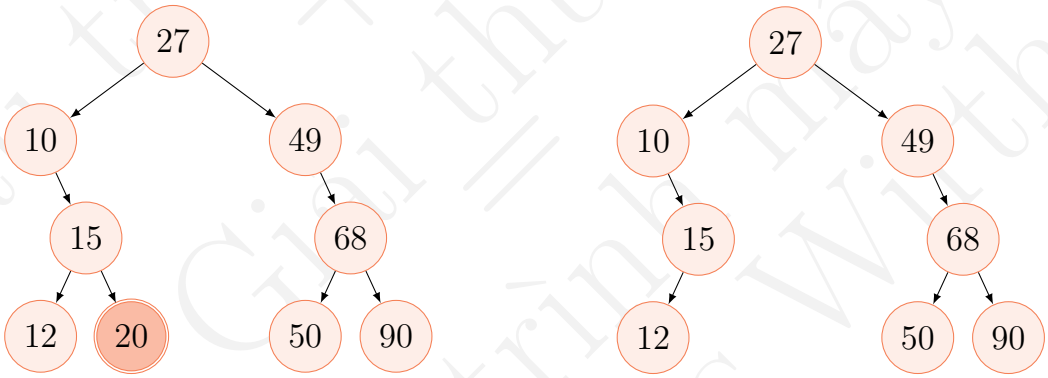


10.2.3 Thao tác xóa một nút trong cây NPTK

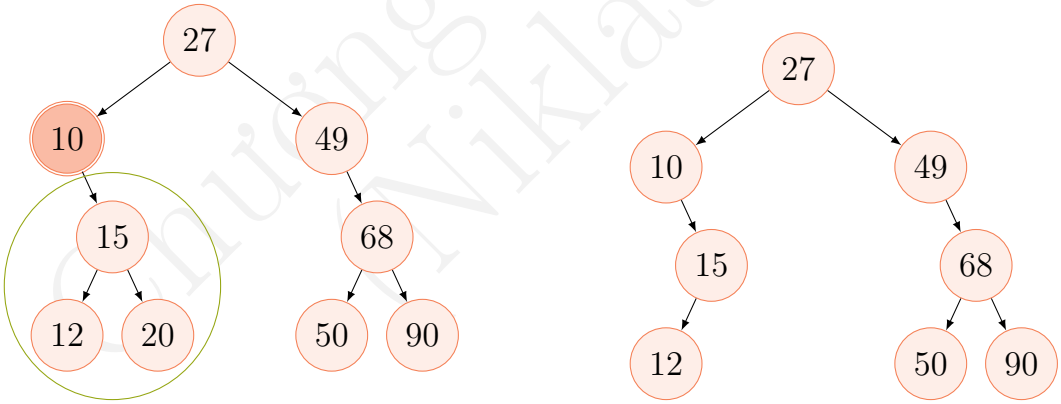
Trong cây NPTK, thao tác xóa một nút cần phải xét ba trường hợp:

- Trường hợp 1: nút khóa k là *nút lá*.
- Trường hợp 2: nút khóa k chỉ có *1* cây con trái hay phải.
- Trường hợp 3: nút khóa k chứa đầy đủ *2* cây con, thực hiện thao tác tìm *nút phải nhất của cây con trái* hay *nút trái nhất của cây con phải*.

Trường hợp 1: nút k là *nút lá*



Trường hợp 2: nút k chỉ có *1* cây con trái hay phải



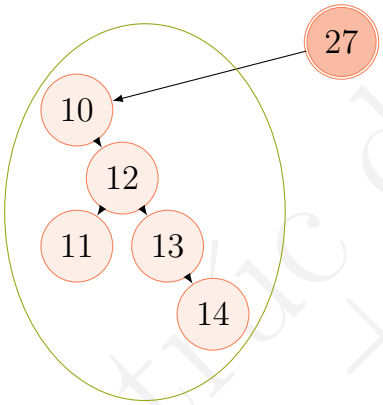
Trường hợp 3: nút k có đầy đủ *2* cây con
Thực hiện thao tác tìm *nút thay thế* trong 2 nút.

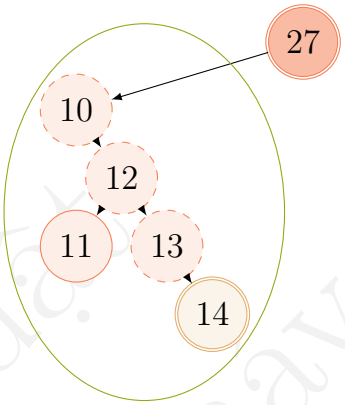
- Nút *phải nhất/lớn nhất* của *cây con trái*
- Nút *trái nhất/nhỏ nhất* của *cây con phải*.

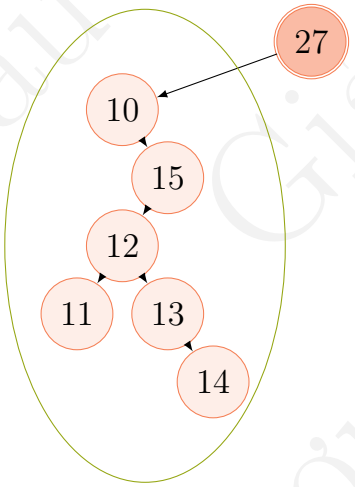
Tìm nút thay thế là nút phải nhất cây con trái

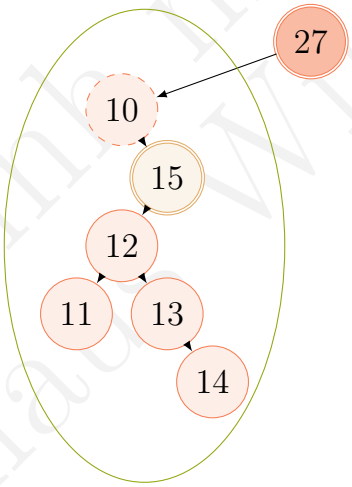
Thuật toán 10.3: SearchStandFor(t, p)
- Đầu vào: cây T.
- Đầu ra: nút p là nút thay thế (nút phải nhất/lớn nhất của cây T).

```
1 |     // Tìm theo nhánh bên phải của cây
2 |     if cây con phải nút đang xét khác rỗng
3 |         Gọi đệ quy hàm SearchStandFor() với cây con phải
4 |     else // Tim phan tu thay the
5 |         Chép dữ liệu của pRoot vào nút p ...
6 |         Lưu lại nhánh con trái (trường hợp nút p có cây con trái)
```

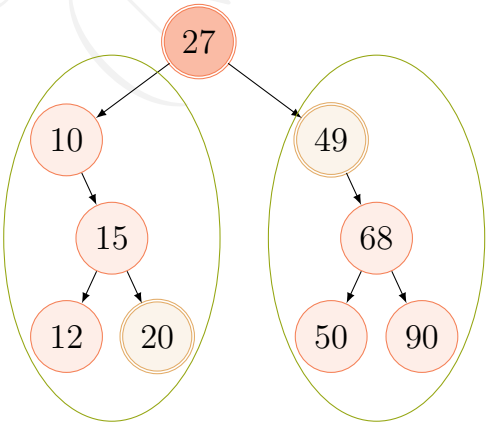








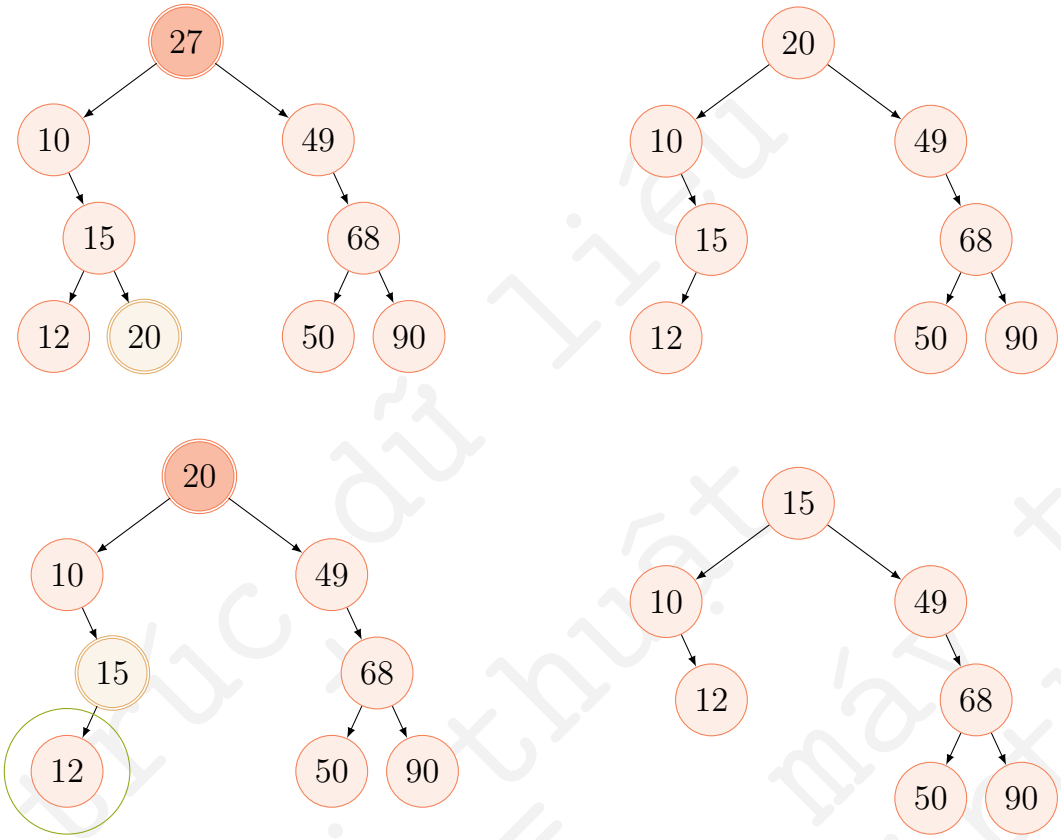
Xét trường hợp tìm nút thay thế là nút phải nhất cây con trái



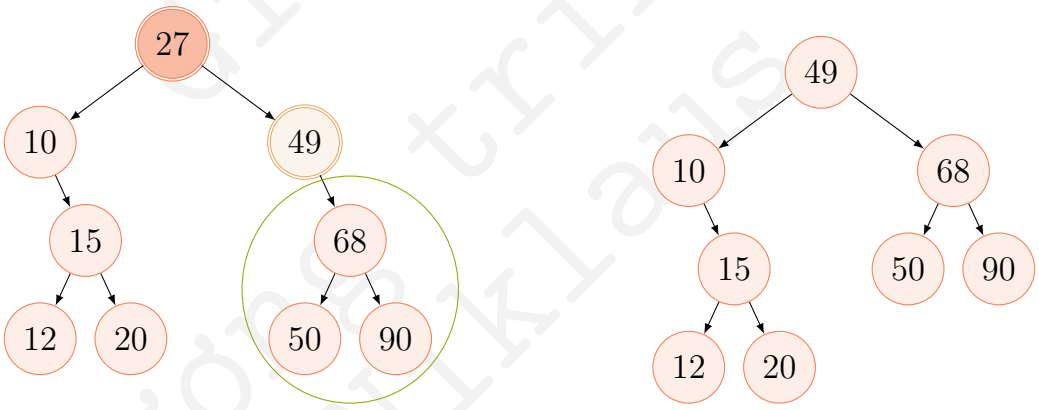
[!htbp]

Nút k có đầy đủ *2* cây con:

- Cách 1. Thực hiện thao tác tìm *nút phải nhất* của *cây con trái*.

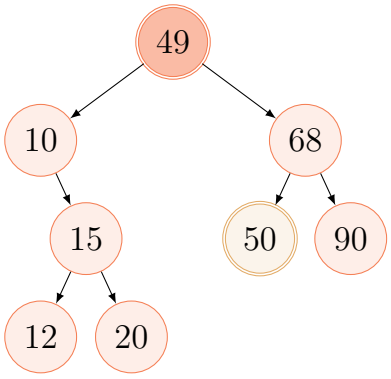


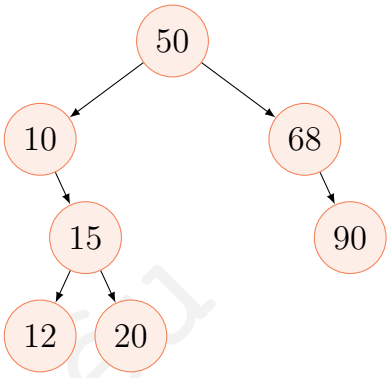
- Cách 2. Thực hiện thao tác tìm *nút trái nhất* của *cây con phải*.



Thuật toán 10.4: RemoveNode(t, k)
- Đầu vào: cây T và nút có khóa k cần xóa.
- Đầu ra: cây T sau khi xóa nút có khóa k.

```
1    if cây rỗng // TH1. Khong tim thay khoa k
2        Dừng thuật toán
3    if pRoot->Key > k
4        Gọi đệ quy hàm RemoveNode() với cây con trái
5    else if pRoot->Key < k
```






```
        Gọi đệ quy hàm RemoveNode() với cây con phải
    // TH2. Tim thay nut pRoot co khoa k
    else
        // Xoa nut p tuong ung 3 truong hop
        // TH2.1. nut la
        Khai báo nút p trỏ đến pRoot
        // TH2.2. nut co 1 cay con trai/phai
        if p chỉ có 1 cây con trái // nut co 1 cay con trai
            pRoot trỏ đến cây con trái của p
        else if p chỉ có 1 cây con phải // nut co 1 cay con phai
            pRoot trỏ đến cây con phải của p
        else // TH2.3. p co 2 cay con
            Gọi hàm tìm nút thay thế trước khi xóa nút có khóa k ...
            Xóa nút p
```

10.3 Mã nguồn của chương

```
public void InsertNode(ref Node root, Node p)
{
    if (root == null)
    {
        root = p;
    }
    else
    {
        if (root.Key > p.Key)
        {
            Insert(ref root.pLeft, p);
        }
        else if (root.Key < p.Key)
        {
            Insert(ref root.pRight, p);
        }
        else
        {
            return;
        }
    }
```

```
}
```

```
public Node SearchNode(ref Node root, int k)
{
    if (root != null)
    {
        if (root.Key == k) // TH1. Tim thay nut co khoa k
            return root;
        else if (root.Key > k) // Tim de quy cay con trai
            return SearchNode(ref root.pLeft, k);
        else // Tim de quy cay con phai
            return SearchNode(ref root.pRight, k);
    }
    // TH2. Khong tim thay
    return null;
}
```

```
public void SearchStandFor(ref Node root, Node p)
{
    if (root.pRight != null)
        SearchStandFor(ref root.pRight, p);
    else
    {
        p.Key = root.Key;
        p = root;
        root = root.pLeft;
    }
}
```

```
public void RemoveNode(Node root, int k)
{
    Node p = new Node();
    if (pRoot == null) // TH1. Khong tim thay nut co khoa k
        return;
    if (root.Key > k)
        RemoveNode(root.pLeft, k);
    else if (root.Key < k)
        RemoveNode(root.pRight, k);
    else // TH2. Tim thay nut co khoa k
    {
        // TH2.1. nut la
        p = root;
        // TH2.2 nut co 1 cay con trai/phai
        if (p.pRight == null) // nut co 1 cay con trai
            root = p.pLeft;
        else if (p.pLeft == null) // nut co 1 cay con phai
            root = p.pRight;
        // TH2.3. nut co 2 cay con
        else
            SearchStandFor(ref root.pLeft, p);

```

```
23            p = null;
24        }
25    }
```

10.4 Bài tập cuối chương

1. Cho dãy gồm 9 phần tử: 5, 9, 3, 1, 8, 7, 4, 6, 2.
   - Lần lượt thêm các phần tử trên vào cây NPTK.
   - In cây nhị phân tìm kiếm theo 3 phương pháp duyệt cây: NLR, LNR, LRN.
   - Thêm vào phần tử 10.
   - Xóa phần tử 5.
2. Đối với thao tác tìm kiếm trong cây NPTK, hãy cho biết độ phức tạp trong trường hợp xấu nhất và trung bình? Giải thích.
3. Viết lại các thuật toán duyệt cây NLR, LNR, LRN *(sử dụng phương pháp lặp thay cho kỹ thuật đệ quy).*
4. Viết thuật toán tìm kiếm phần tử nhỏ nhất/lớn nhất trong cây NPTK.
5. Viết thuật toán kiểm tra một cây NPTK có phải là cây suy biến hay không?

Bài 11

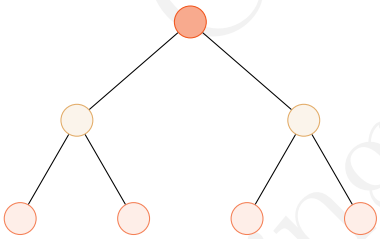
CÂY CÂN BẰNG (3 tiết)

11.1 Giới thiệu cây cây bằng

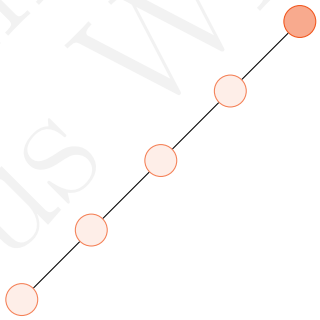
Xét chiều cao của một cây NPTK có ba trường hợp xảy ra:

- Chiều cao cây con trái *bằng* chiều cao cây con phải
- Chiều cao cây con trái *lớn hơn* chiều cao cây con phải
- Chiều cao cây con trái *nhỏ hơn* chiều cao cây con phải

Các thao tác thêm, xóa nút có thể làm chiều cao cây con trái hay phải lệch nhau nhiều, dẫn đến trường hợp cây suy biến. Điều này khiến cấu trúc cây NPTK không còn hiệu quả.



(a) Cây nhị phân hoàn hảo.



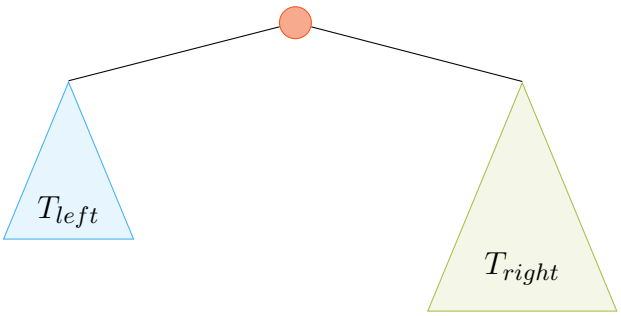
(b) Cây nhị phân suy biến.

Khái niệm

Định nghĩa 11.1.1. Cây cân bằng (AVL Tree - gọi theo tên các tác giả Adelson Velsky, và Landis) là một cây NPTK thỏa điều kiện tại tất cả các đỉnh chiều cao cây con trái và cây con phải lệch nhau tối đa là 1.

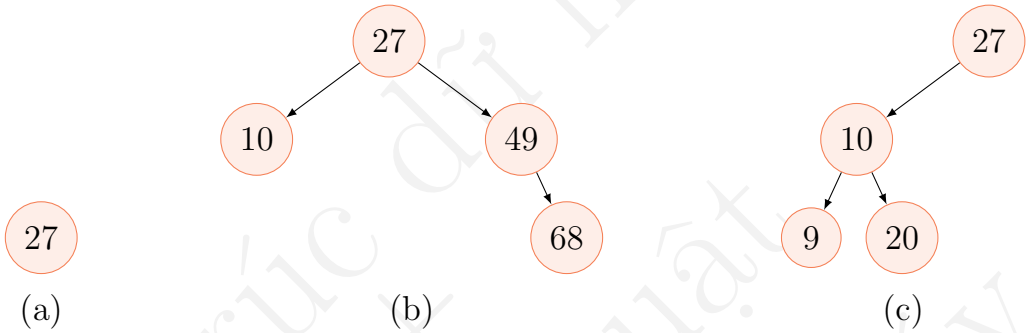
Định nghĩa 11.1.2. Chỉ số cân bằng (CSCB) của một nút là hiệu của chiều cao cây con trái và cây con phải của nó.

- CSCB(p) = 0 <=> Chiều cao cây trái (p) = Chiều cao cây phải (p)
- CSCB(p) = 1 <=> Chiều cao cây trái (p) < Chiều cao cây phải (p)



- CSCB(p) =-1 <=> Chiều cao cây trái (p) > Chiều cao cây phải (p)

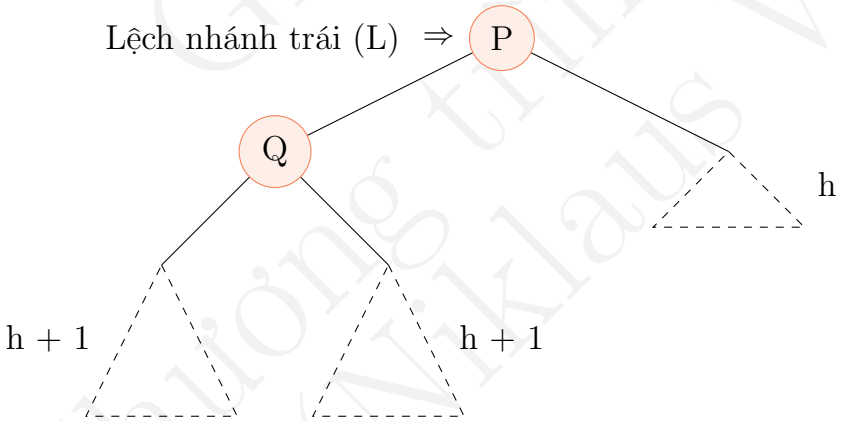
Ví dụ 11.1.1. *Cây cân bằng*

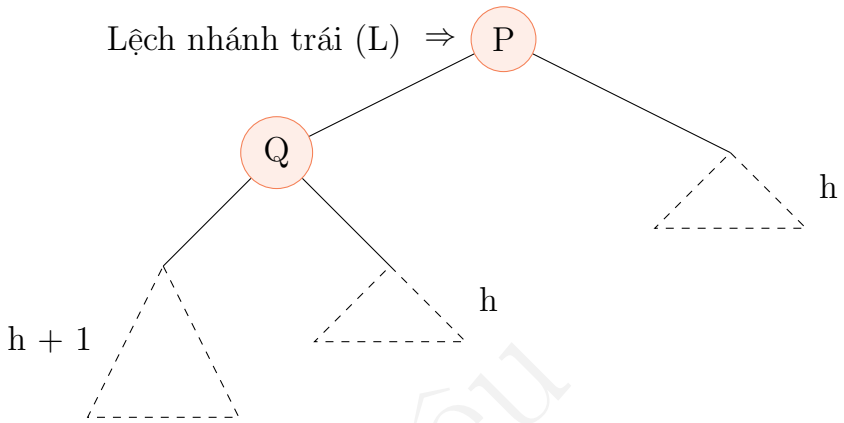


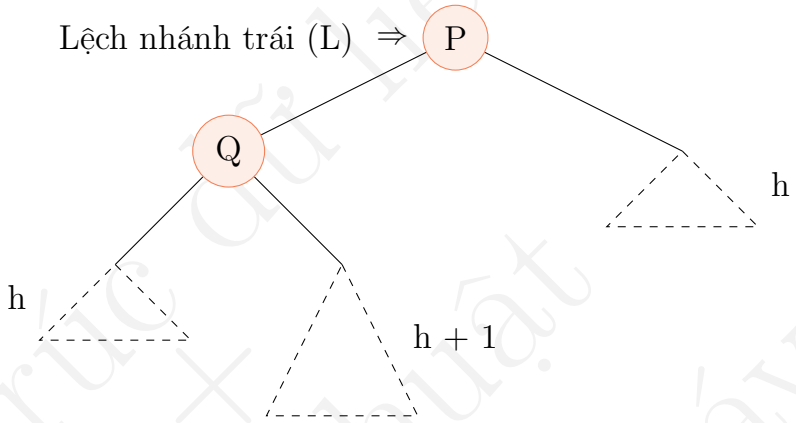
Hình 11.2: Cây (a), (b) là cây AVL, cây (c) không phải cây AVL.

11.2 Các trường hợp mất cân bằng

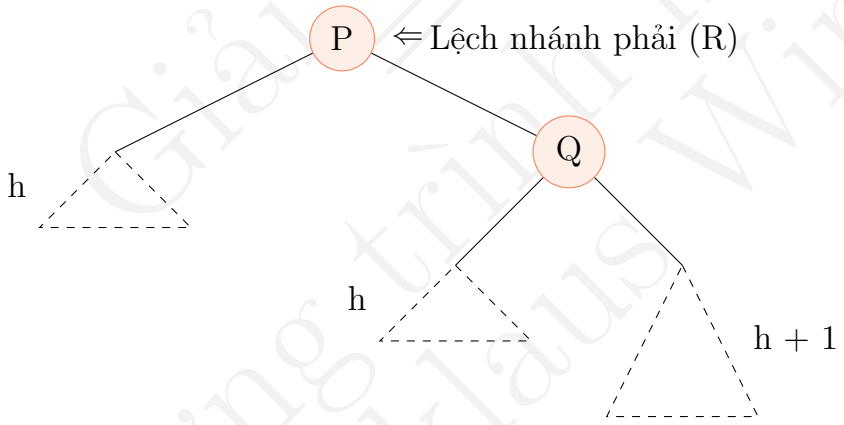
- Cây T lệch về bên trái (3 trường hợp con)

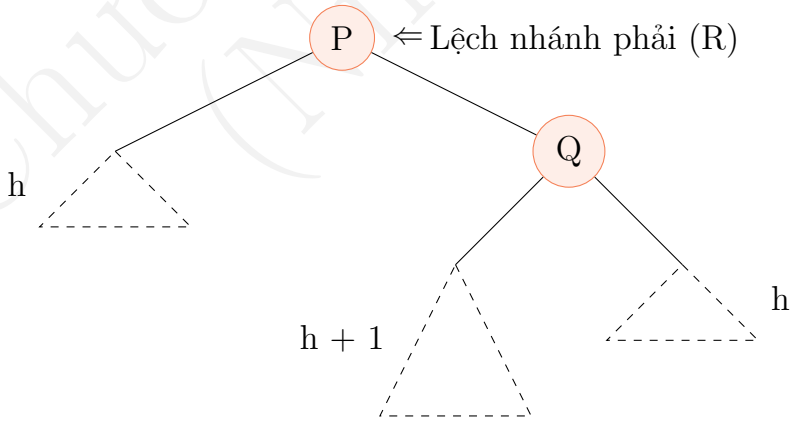


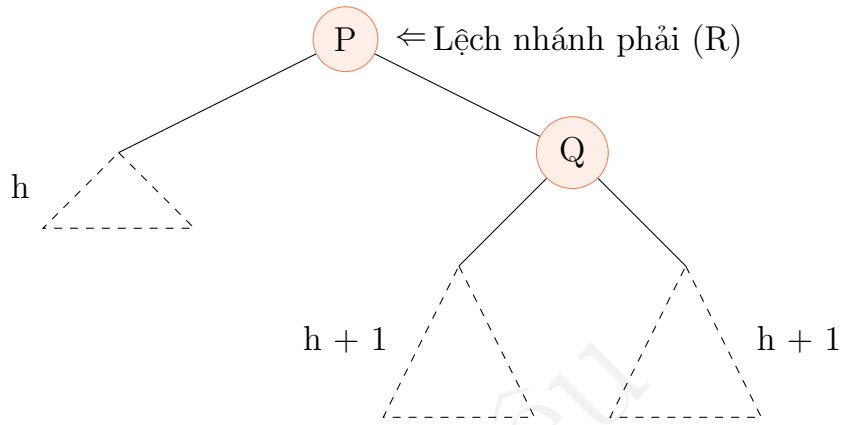




- Cây T lệch về bên phải (3 trường hợp con)







11.3 Các thao tác với cây AVL

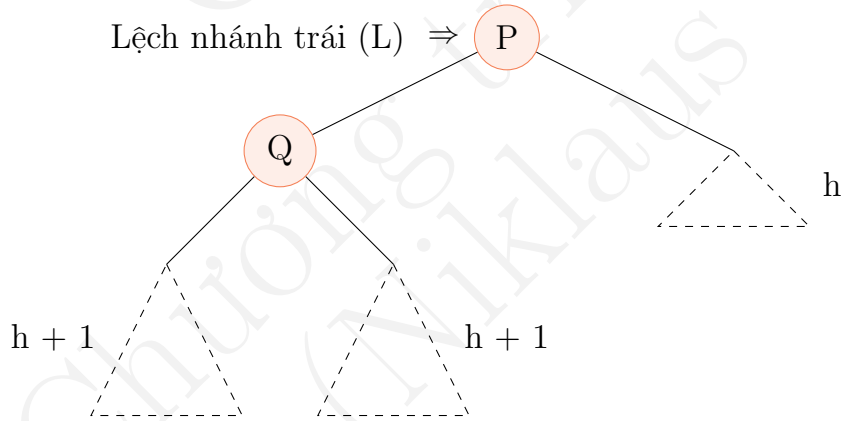
Cây AVL cũng có các thao tác tương tự cây NPTK và có thể thao tác để xử lý trường hợp mất cân bằng.

- Thêm phần tử vào cây AVL
- Xóa phần tử khỏi cây AVL
- Tìm phần tử trong cây AVL
- Cân bằng lại cây AVL nếu cây mất cân bằng

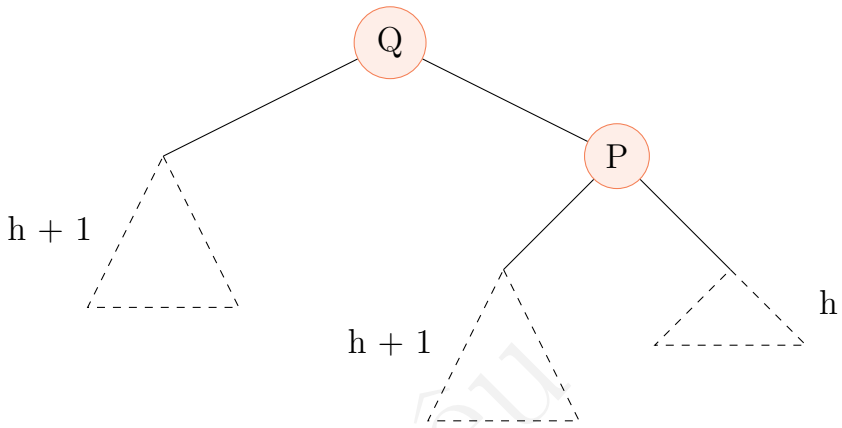
11.3.1 Cân bằng lại cây trường hợp lệch trái

Tương ứng với trường hợp cân mất cân bằng do bị lệch trái thì thực hiện cân bằng lại bằng một số phép quay sau đây.

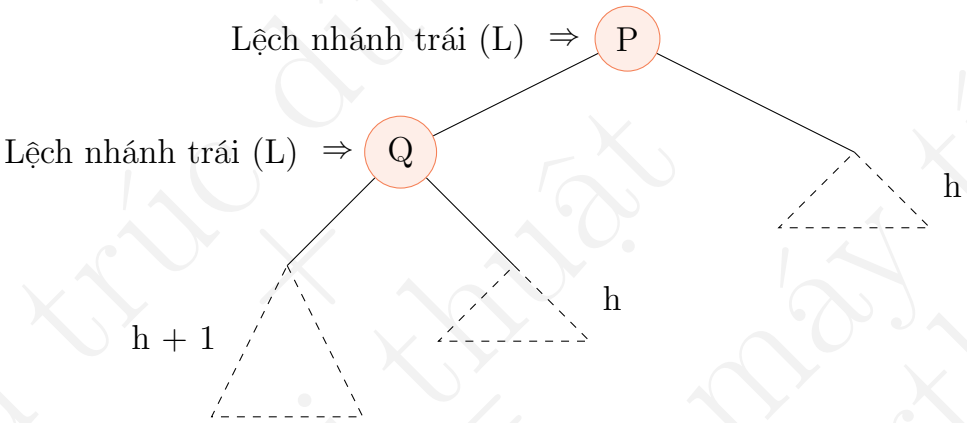
- **Trường hợp cây lệch trái P, nhưng tại Q không bị lệch:**



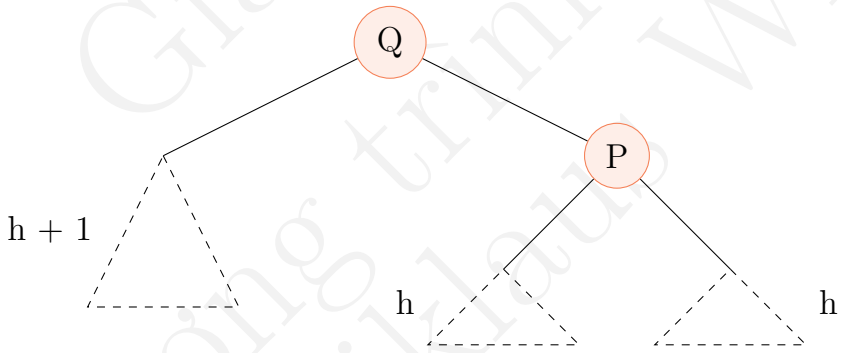
Thực hiện phép quay Phải: quay phải tại P.



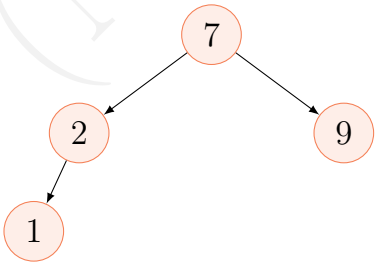
- **Trường hợp cây lệch trái tại hai nút P và Q:**



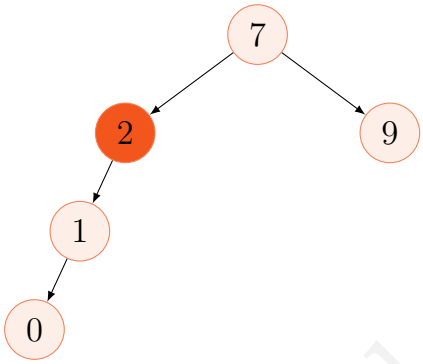
Thực hiện phép quay Phải: quay phải tại P.

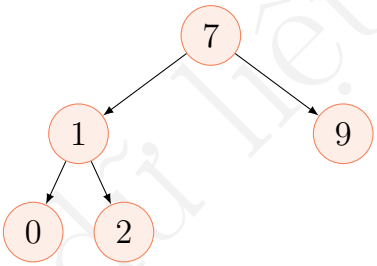


Ví dụ 11.3.1. *Cho cây AVL như hình, thêm nút 0 vào cây.*

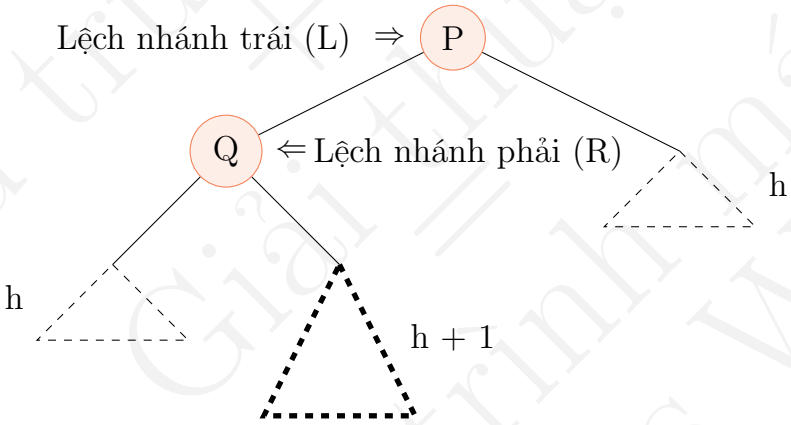


Giải.

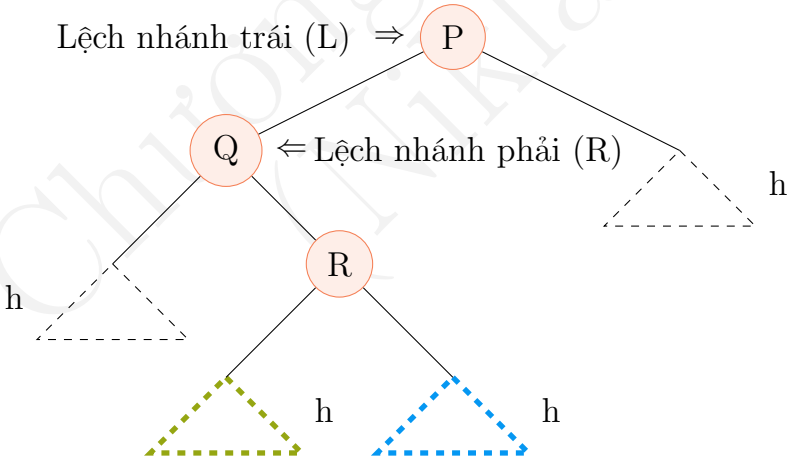




- **Trường hợp cây lệch trái tại P và lệch phải tại Q:**

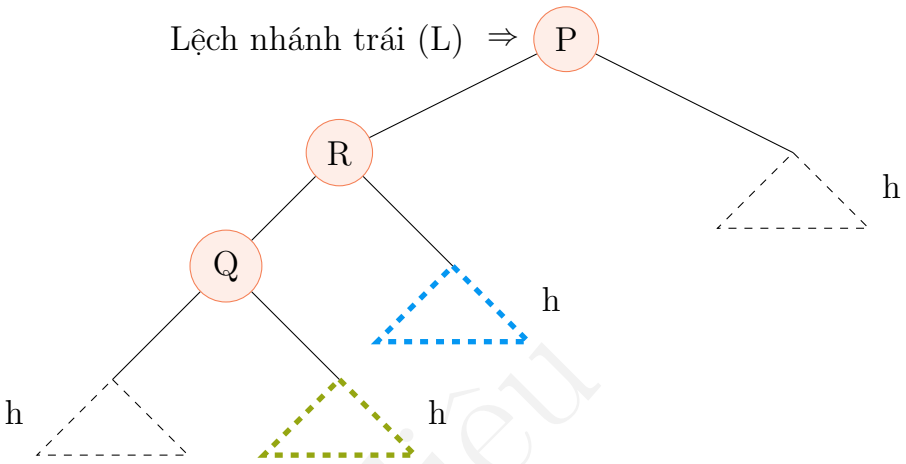


Giả sử, cây con bên phải của Q là một cây con với nút gốc là R. Hai cây con của nút R có chiều cao là h.

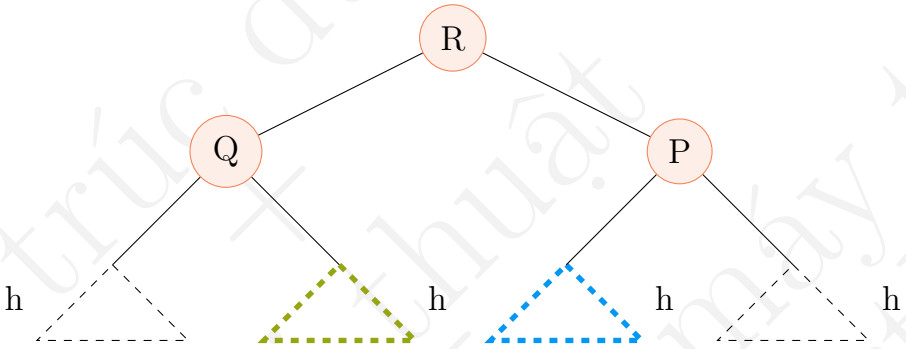


Thực hiện phép quay Trái-Phải

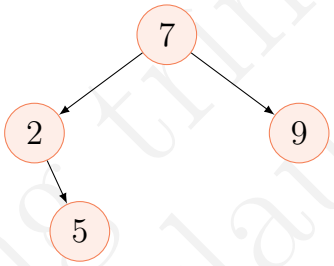
- Bước 1: quay trái tại Q

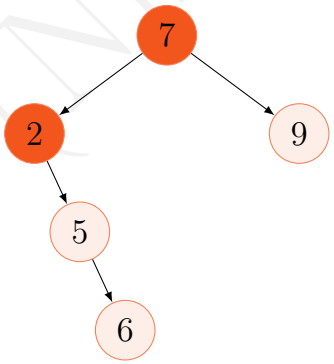


- Bước 2: quay phải tại P.

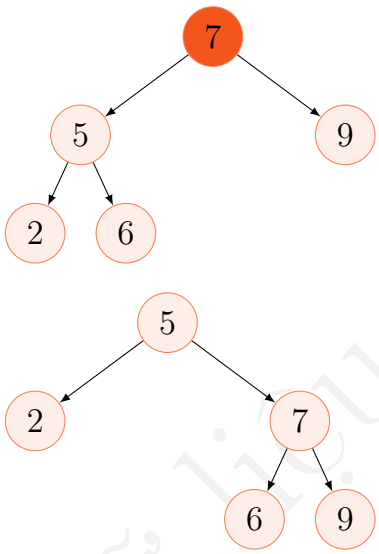


Ví dụ 11.3.2. *Cho cây AVL như hình, thêm nút 6 vào cây.*





Giải.



11.3.2 Cân bằng lại cây trường hợp lệch phải

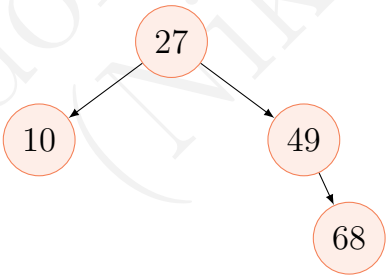
Những trường hợp lệch phải thì thao tác quay ngược lại với các trường hợp lệch trái.

11.4 Độ phức tạp thuật toán

- Các phép quay có thời gian thực hiện là $O(1)$.
- AVL là cây nhị phân thêm vào một số phép quay để xử lý các trường hợp mất cân bằng. Thao tác thêm và xóa một nút cần phải cân bằng lại (có thể lan truyền từ nút vừa thêm đến nút gốc). Do cây AVL có chiều cao h nên thao tác thêm và xóa có độ phức tạp thời gian là $O(\log n)$

11.5 Bài tập cuối chương

1. Cho dãy gồm các phần tử: 6, 43, 7, 42, 59, 63, 11, 21, 56, 54, 27, 20, 36. Vẽ cây AVL sau khi lần lượt thêm các phần tử vào cây.
2. Hãy so sánh thời gian thực hiện thao tác tìm kiếm của cây NPTK và cây AVL.
3. Cho cây AVL như hình sau: Hãy cho ví dụ khi thêm một phần tử vào cây AVL



tương ứng các trường hợp:

(a) Cây không bị mất cân bằng
(b) Cây mất cân bằng cần thực hiện một phép quay
(c) Cây mất cân bằng cần thực hiện hai phép quay

Bài 12

BẢNG BĂM - HÀM BĂM (3 tiết)

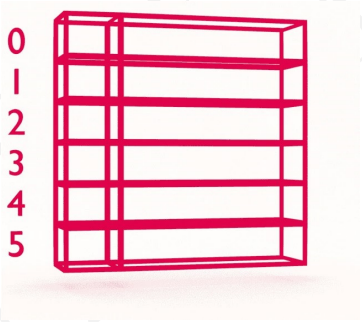
12.1 Giới thiệu bảng băm

Khái niệm

Bảng băm *(Hash Table)* là một cấu trúc dữ liệu phổ biến trong việc lưu trữ dữ liệu chưa có thứ tự.

- Mỗi mẩu tin của dữ liệu sẽ được đưa qua một hàm băm để lưu vào chỉ mục tương ứng trong bảng băm.
- Các thao tác thêm, xóa, tìm kiếm trên bảng băm được thực hiện với độ phức tạp tuyến tính.

Hàm băm (Hash) là phép biến đổi/ánh xạ khóa của một mẩu tin thành một chỉ mục trong bảng băm.

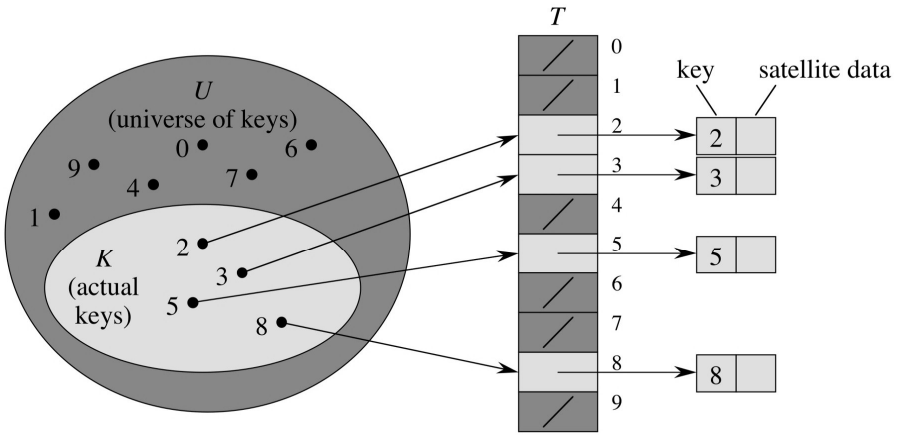


Định nghĩa 12.1.1. Phép biến đổi khóa là một ánh xạ từ tập hợp vũ trụ $\mathcal{U}$ của tất cả các khóa vào tập $S \in \mathcal{U}$.

$$\begin{array}{rcl} h : \mathcal{U} & \rightarrow & S = \{0, 1, \ldots, m-1\} \\ k & \mapsto & h(k) \end{array} \tag{12.1}$$

Trong đó,

- $\mathcal{U}$ là tập vũ trụ (chứa số lượng rất lớn các khóa)
- S là tập các chỉ mục trong bảng băm
- $h(k)$ là một hàm băm cho trước



Hình 12.1: Mô tả cách lưu trữ của bảng băm.

Khóa của hàm băm:

- Phải là một số nguyên không dấu
- Có thể dạng số (số thực cần chuyển thành số nguyên) hay chuỗi (sau khi chuyển thành mã ASCII)

12.2 Phân loại hàm băm

Một hàm băm tốt phải thỏa các điều kiện sau:

- Tính toán nhanh.
- Các khóa được phân bố đều trong bảng.
- Ít xảy ra đụng độ.
- Xử lý được các loại khóa có kiểu dữ liệu khác nhau.

12.2.1 Hàm băm sử dụng phương pháp chia lấy phần dư (modulo)

Hàm băm sử dụng phương pháp chia lấy phần dư (modulo) được định nghĩa bởi:

$$h(k) = k \bmod m \tag{12.2}$$

Trong đó,

- k là khóa cần lưu trữ

- m là kích thước/số lượng chỉ mục của bảng băm

Chọn m sẽ ảnh hưởng đến giá trị của $h(k)$:

- Nếu chọn $m = 2^n$ thì giá trị của $h(k)$ sẽ là n bit cuối cùng của k trong biểu diễn nhị phân.
- Nếu chọn $m = 10^n$ thì giá trị của $h(k)$ sẽ là n chữ số cuối cùng của k trong biểu diễn thập phân.
- Hai cách trên giá trị của $h(k)$ chỉ phụ thuộc vào n bit (n chữ số cuối). Thường chọn m là số nguyên tố để $h(k)$ phụ thuộc vào khóa.

12.2.2 Hàm băm sử dụng phương pháp nhân

Tác giả Knuth đã đề xuất hàm băm có công thức như sau:

$$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor \tag{12.3}$$

Trong đó,

- k là khóa
- m là kích thước bảng băm
- A là hằng số với $0 < A < 1$

Theo tác giả Knuth, chọn m và A như sau:

- Thường chọn $m = 2^n$
- Giá trị A thường chọn $A = \frac{\sqrt{5}-1}{2} \approx 0.61803398874989$

12.3 Các thao tác với bảng băm

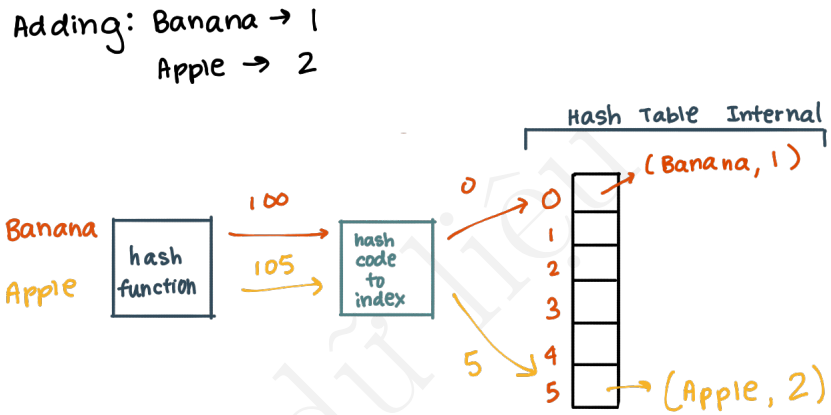
Một số thao tác cơ bản trong bảng băm:

- Khởi tạo bảng băm
- Thêm phần tử vào bảng băm
- Xóa phần tử khỏi bảng băm
- Tìm kiếm một phần tử trong bảng băm

12.4 Các phương pháp xử lý đụng độ của hàm băm

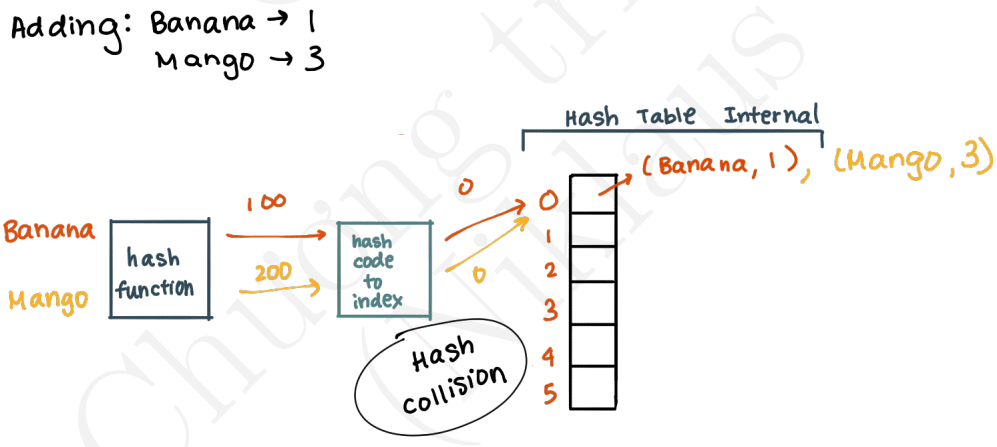
Một số phương pháp xử lý đụng độ của hàm băm:

- Phương pháp tạo dây chuyền/kết nối trực tiếp (Direct chaining): mỗi chỉ mục của bảng có chứa một danh sách liên kết đơn. Các chỉ mục này lưu địa chỉ phần tử đầu tiên của danh sách.
- Phương pháp dùng địa chỉ mở (Open addressing): nếu xảy ra đụng độ, thì tìm chỉ mục kế tiếp cho đến khi tìm thấy hoặc chỉ mục trống (không tìm thấy). Các chỉ mục của bảng băm lưu một phần tử duy nhất.
  - Phương pháp dò tuyến tính (Linear probing)
  - Phương pháp dò bậc 2 (Quadratic probing)
  - Phương pháp băm kép (Double hashing)



Nguồn: *https://guides.codepath.com/compsci/Hash-Tables*

Hình 12.2: Bảng băm không xảy ra đụng độ.



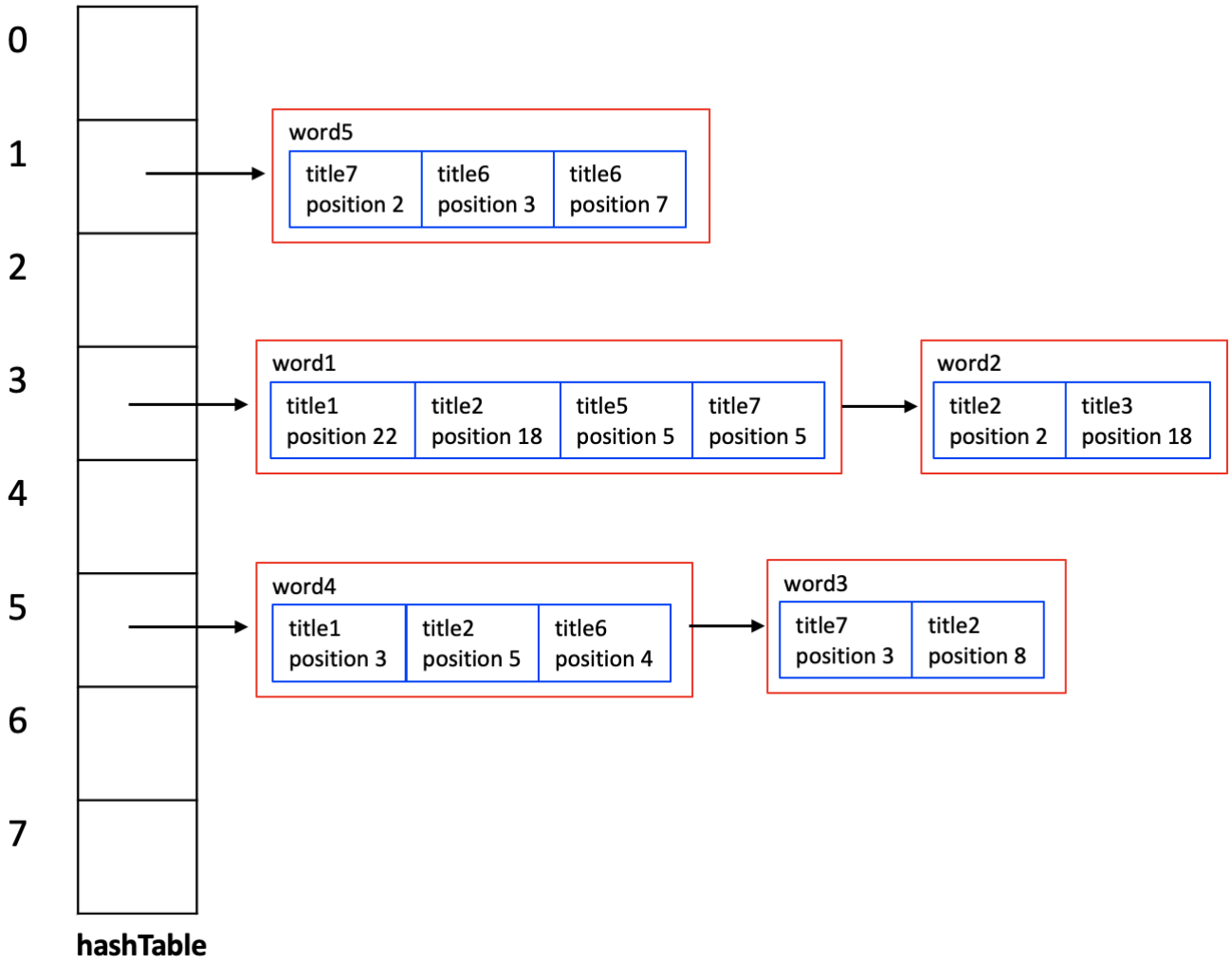
Nguồn: *https://guides.codepath.com/compsci/Hash-Tables*

Hình 12.3: Bảng băm **xảy ra đụng độ** khi hai khóa cùng chỉ mục.

12.4.1 Phương pháp nối kết trực tiếp

- Các chỉ mục của bảng băm sẽ được **băm** thành m danh sách liên kết. Nên bảng băm sẽ gồm m chỉ mục chứa địa chỉ đầu của các danh sách liên kết.
- Tại mỗi chỉ mục của bảng băm sẽ có một danh sách liên kết chứa các khóa khác nhau nhưng có cùng chỉ mục.

Ví dụ 12.4.1. *Phim được lưu trữ thông tin gồm tiêu đề phim và mô tả tóm tắt nội dung. Tiêu đề cũng như mô tả của bộ phim có thể chứa một hoặc nhiều từ. Trong bảng băm, mỗi địa chỉ là các từ trong mô tả phim và các giá trị là số lần xuất hiện của từ (mỗi bộ phim mà từ đó xuất hiện và vị trí tương ứng của nó trong mô tả).*



Nguồn: *https://ds.cs.rutgers.edu/assignment-rumdb/*

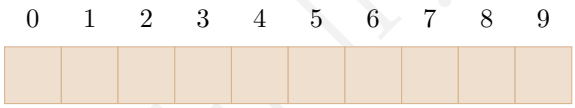
Hình 12.4: Xử lý đụng độ của hàm băm bằng phương pháp nối kết trực tiếp.

12.4.2 Kỹ thuật địa chỉ mở sử dụng phương pháp dò tuyến tính

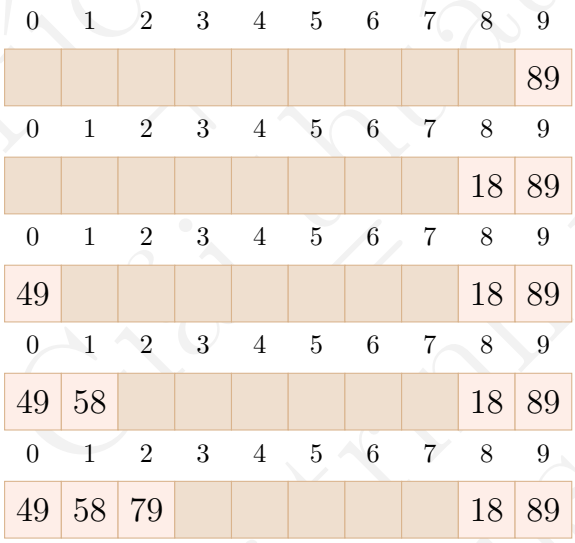
- Khi xảy ra đụng độ, tiếp tục dò địa chỉ phần tử kế tiếp, nếu địa chỉ trống thì thêm vào.
- Sử dụng một hàm băm tốt $h(k)$ để định nghĩa m hàm băm $h_i(k)$:

$$h_i(k) = (h(k) + i) \bmod m, 0 \leq i \leq m - 1 \tag{12.4}$$

Ví dụ 12.4.2. *Cho bảng băm chứa 10 chỉ mục. Hãy lần lượt thêm các khóa 89, 18, 49, 58, 79 vào bảng băm (trường hợp đụng độ xử lý bằng phương pháp dò tuyến tính).*

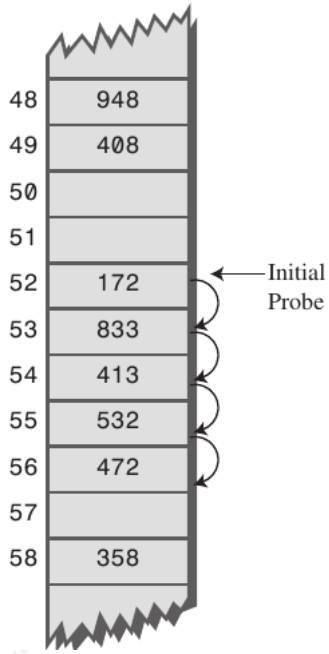


Giải. Hàm băm được định nghĩa như sau: $h_i(k) = (h(k) + i) \bmod 10$. □



Nhận xét

Phương pháp này có xu hướng lưu các chỉ mục gần nhau tạo thành một phân cụm chính (Primary clustering).



Hình 12.5: Bảng băm có phân cụm chính bắt đầu từ chỉ mục 52.

12.4.3 Kỹ thuật địa chỉ mở sử dụng phương pháp dò bậc 2

- Tương tự phương pháp dò tuyến tính, nhưng sử dụng hàm băm là một hàm bậc 2:

$$h_i(k) = (h(k) + i^2) \bmod m, 0 \leq i \leq m-1 \tag{12.5}$$

Ví dụ 12.4.3. *Cho bảng băm chứa 10 chỉ mục. Hãy lần lượt thêm các khóa 89, 18, 49, 58, 79 vào bảng băm (trường hợp đụng độ xử lý bằng phương pháp dò bậc 2).*

Giải. Hàm băm được định nghĩa như sau: $h_i(k) = (h(k) + i^2) \bmod 10$. □

0	1	2	3	4	5	6	7	8	9
									89

0	1	2	3	4	5	6	7	8	9
								18	89

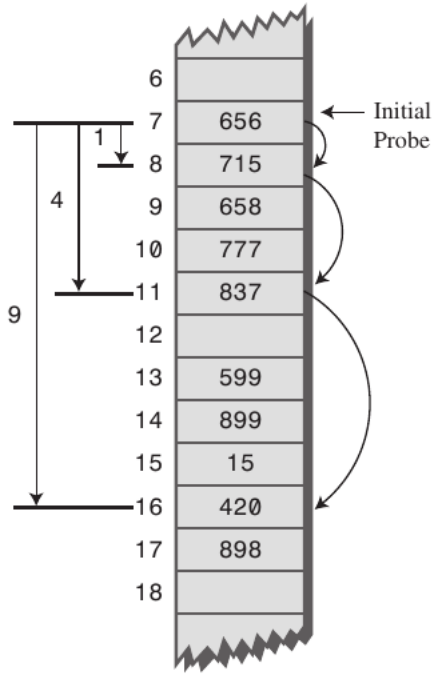
0	1	2	3	4	5	6	7	8	9
49								18	89

0	1	2	3	4	5	6	7	8	9
49		58						18	89

0	1	2	3	4	5	6	7	8	9
49		58	79					18	89

Nhận xét

Phương pháp này có xu hướng lưu các chỉ mục cách xa nhau tạo thành một phân cụm phụ (Secondary clustering), hiệu quả hơn phân cụm chính trong phương pháp dò tuyến tính.



Hình 12.6: Bảng băm có phân cụm phụ theo bước nhảy 1, 4 ,9, ...

12.4.4 Kỹ thuật địa chỉ mở sử dụng phương pháp băm kép

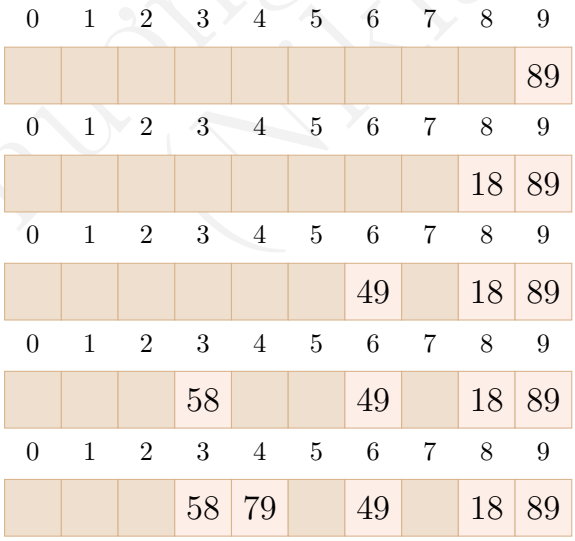
- Kết hợp hai hàm băm $h(k), g(k)$ độc lập để định nghĩa m hàm băm:

$$h_1(k) = (h(k) + ig(k)) \bmod m, 0 \leq m \leq m-1. \tag{12.6}$$

- Khóa được phân bố đều hơn phương pháp dò tuyến tính.

Ví dụ 12.4.4. *Cho bảng băm chứa 10 chỉ mục và hai hàm băm* $h_1(k) = k \bmod 10$ *và* $h_2(k) = 7 - (k \bmod 7)$. *Hãy lần lượt thêm các khóa 89, 18, 49, 58, 79 vào bảng băm.*

Giải. Hàm băm được định nghĩa như sau: $h_i(k) = ((k \bmod 10) + 7 - (k \bmod 7)) \bmod 10$. □



12.4.5 Độ phức tạp của thuật toán

Đối với thao tác tìm kiếm, bảng băm hoàn hảo *(hàm băm phân phối đều các khoá vào các vị trí trong bảng băm)* có sử dụng kỹ thuật địa chỉ mở dò tuyến tính có độ phức tạp thuật toán là:

- Trường hợp xấu nhất: $O(n)$
- Trường hợp trung bình: hiệu quả phụ thuộc trên mức độ đầy α với $\alpha = n/m$, là tỷ số giữa số chỉ mục đã sử dụng và kích thước của bảng băm.
  - Tìm thành công:
  $$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \tag{12.7}$$
  - Tìm thất bại:
  $$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) \tag{12.8}$$

12.5 Bài tập cuối chương

1. Cho bảng băm kích thước 11 chỉ mục và hàm băm $h(k) = (5k + 4) \bmod 11$. Thêm lần lượt các khóa 3, 9, 2, 1, 14, 6, 25 vào bảng băm. Nếu xảy ra đụng độ, xử lý bằng phương pháp nối kết trực tiếp.
2. Cho bảng băm kích thước 11 chỉ mục và hàm băm $h(k) = k \bmod 11$. Thêm lần lượt các khóa 0, 1, 8, 9, 52, 44, 56, 53, 61, 64 vào bảng băm. Xử lý trường hợp đụng độ theo các cách sau:
   a) Phương pháp dò tuyến tính
   b) Phương pháp dò bậc 2
   c) Phương pháp sử dụng băm kép
3. Trong bảng băm, số lượng khóa có thể ít hơn chỉ mục hay không? Trường hợp nào?
4. Cho bảng băm lưu trữ các khóa như hình. Hãy cho biết mức độ đầy α của bảng băm là bao nhiêu?

0	1	2	3	4	5	6	7	8	9
49		58	79					18	89

5. Viết mã giả đoạn chương trình thực hiện thao tác xác định chỉ mục còn trống trong bảng băm tương ứng các cách sau:
   a) Phương pháp dò tuyến tính
   b) Phương pháp dò bậc 2
   c) Phương pháp sử dụng băm kép

Tài liệu tham khảo

[1] Dương Anh Đức, Trần Hạnh Nhi, *Nhập môn Cấu trúc dữ liệu và Thuật toán*. Đại học Khoa học tự nhiên TP Hồ Chí Minh, 2003.

[2] Donald E. Knuth, *The Art of Computer Programming, Volume 3*. Addison-Wesley, 1998.

[3] Robert Sedgewick, *Algorithms in C*. Addison-Wesley, 1990.

[4] Niklaus Wirth, *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.