



TRƯỜNG ĐẠI HỌC AN GIANG
KHOA CÔNG NGHỆ THÔNG TIN

CẤU TRÚC DỮ LIỆU & GIẢI THUẬT
Mã số: COS304

: Nguyễn Thái Dư

1

1

MỤC TIÊU

- ◆ Cần làm chủ:
 - Ngôn ngữ: C
- ◆ Mục tiêu:
 - Có hiểu biết tốt về CTDL và GT
 - Hiểu và cài đặt được các kiểu dữ liệu trùu tượng cơ bản
 - Nắm được các giải thuật về sắp xếp và tìm kiếm
 - Nắm được một số phương pháp thiết kế giải thuật
 - Rèn luyện cách phân tích một bài toán, Tìm ra giải thuật
 - Thể hiện cách phân tích qua NNLT cụ thể (**C, Java**)

3

3

Chương 0. GIỚI THIỆU

- ◆ **Chương 1:** Tổng quan về giải thuật và cấu trúc dữ liệu.
- ◆ **Chương 2:** Tìm kiếm và sắp xếp.
- ◆ **Chương 3:** Cấu trúc dữ liệu động.
- ◆ **Chương 4:** Cấu trúc cây.

2

2

Phương pháp học tập

- ◆ **Giảng viên:** Cung cấp bài giảng, bài tập, tài liệu tham khảo.
- ◆ **Sinh viên:**
 - Tự giác làm các bài tập
 - Đọc tài liệu tham khảo liên quan;
 - Trong giờ học PHẢI trả lời khi GV hỏi;
 - PHẢI đê điện thoại ở chế độ rung và KHÔNG nghe điện thoại trong lớp.
 - KHÔNG sử dụng máy tính trong giờ lý thuyết;
- ◆ **GV-SV:** Giải đáp thắc mắc - Trao đổi

4

4

Phân bố tiết của môn học

- ◆ Tổng cộng: 45 tiết
 - Lý thuyết: 30 tiết
 - Thực hành: 30 tiết

5

Tài liệu tham khảo

- ◆ Nhập môn Cấu trúc dữ liệu và thuật toán – Hoàng Kiêm (chủ biên), Trần Hạnh Nhi, Dương Anh Đức, 2003.
- ◆ Cấu trúc dữ liệu và giải thuật, Đỗ Xuân Lôi, NXB Khoa học và Kỹ thuật, 1995.
- ◆ Cấu trúc dữ liệu, Nguyễn Văn Linh (chủ biên), ĐH Cần Thơ, 2003.
- ◆ Giải thuật, Nguyễn Văn Linh (chủ biên), ĐH Cần Thơ, 2003.
- ◆ Data Structures and Algorithm Analysis in C, Mark Allen Weiss, 1992.
- ◆ Algorithms In C, Sedgewick, 1990.

6

Tài liệu tham khảo

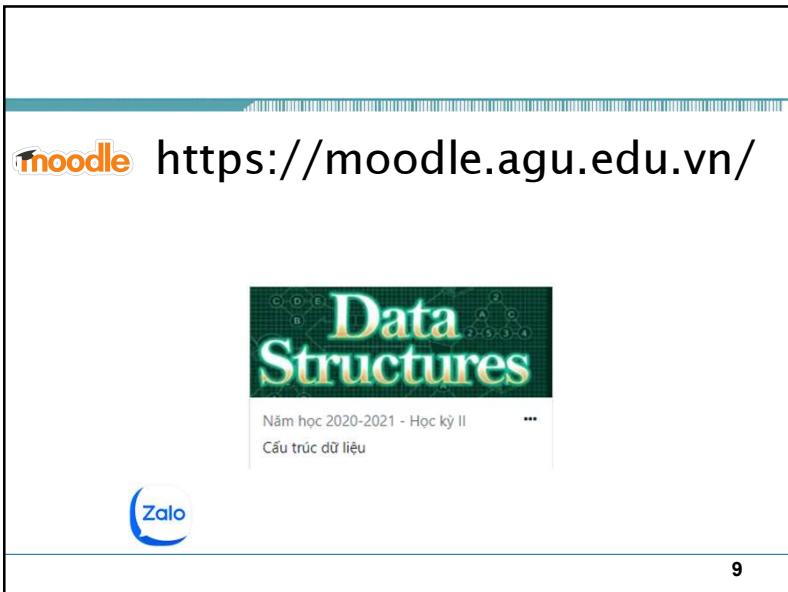
- ◆ Introduction to Algorithms 2nd, Thomas H. Cormen, 2001.
- ◆ Sedgewick Robert, Cẩm nang thuật toán, tập 1 và 2, bản dịch của Hoàng Hồng, NXB Khoa học và Kỹ thuật, 2001.
- ◆ Wirth Niklaus, Cấu trúc dữ liệu + Giải thuật = Chương trình, bản dịch của Nguyễn Quốc Cường, Nhà xuất bản Giáo dục, 1993.

7

Cách tính điểm môn học

- ◆ **Điểm môn học = 50% ĐGTX+ 50% Thi HK**
- ◆ **Điểm ĐGTX: Chuyên cần + Điểm bài kiểm tra [+tiểu luận]**
 - **Chuyên cần:** Vắng 1 buổi thực hành trừ 2,5 đ
 - Xung phong và trả lời đúng – 0,5-1đ. Tối đa: 2đ
- ◆ **Thi kết thúc học phần:**
 - Thời gian: 90-120 phút
 - Hình thức: Viết
 - Không sử dụng tài liệu

8



9



10

Chương 1. TỔNG QUAN CẤU TRÚC DỮ LIỆU

- ◆ Cấu trúc dữ liệu (Data Structures)
- ◆ Kiểu dữ liệu trừu tượng (Abstract Data Type - ADT)
- ◆ Giải thuật (Algorithms)
- ◆ Tính toán độ phức tạp của giải thuật (Computational complexity of algorithms)
- ◆ Phân tích giải thuật (Algorithm Analysis)

1

2

1

2

Kiểu dữ liệu trừu tượng (ADT)

- ◆ Một kiểu dữ liệu trừu tượng (Abstract Data Type - ADT) là tập hợp các đối tượng và được xác định hoàn toàn bởi các phép toán có thể biểu diễn trên các đối tượng đó.
- ◆ ADT là một mô hình toán của cấu trúc dữ liệu xác định kiểu dữ liệu được lưu trữ, các thao tác được hỗ trợ trên dữ liệu đó và kiểu của các tham số trong từng thao tác.

3

4

3

Cấu trúc dữ liệu (Data Structures)

- ◆ Cấu trúc dữ liệu dùng để tổ chức dữ liệu
 - Thường có nhiều hơn một thành phần
 - Có các thao tác hợp lý trên dữ liệu
 - Dữ liệu có thể được kết nối với nhau (ví dụ: array) như là một tập hợp.

Kiểu dữ liệu trừu tượng (ADT)

- ◆ Có hai loại ADT
 - Đơn/nguyên tử: int, char, ...
 - Có cấu trúc: array, struct,...
- ◆ Ngoài những ADT do ngôn ngữ lập trình cung cấp, người lập trình có tạo ra các ADT của riêng mình
- ◆ Trong C, các ADT do người dùng định nghĩa sẽ thông qua kiểu cấu trúc (struct), các thao tác được xây dựng bằng các hàm (functions)

4

Kiểu dữ liệu trừu tượng (ADT)

◆ Các lớp thao tác của một ADT

- Tạo lập đối tượng mới
- Biến đổi các đối tượng của ADT
 - Mang lại những thay đổi cần thiết cho đối tượng
- Quan sát
 - Cho biết trạng thái của đối tượng
- Chuyển đổi kiểu
 - Chuyển kiểu từ kiểu này sang kiểu khác
- Vào ra dữ liệu
 - Nhập/xuất giá trị cho đối tượng

5

Kiểu dữ liệu trừu tượng (ADT)

◆ Person

- Cấu thành bởi:
 - Họ tên
 - Ngày sinh
 - Nơi sinh
 - Phái
- Phép toán:
 - Tạo mới một person (với thông tin đầy đủ)
 - Hiển thị thông tin về một person
 -

6

Tại sao cần phải học Cấu trúc dữ liệu và Giải thuật?

◆ Giải thuật?

- Tại sao lại cần phải học giải thuật? Vai trò của giải thuật? Những vấn đề nào sẽ cần giải quyết bằng giải thuật?

◆ Giải thuật:

- Là một khái niệm quan trọng nhất trong tin học. Thuật ngữ này xuất phát từ nhà toán học Ả Rập Abu Ja'far Mohammed ibn Musa al Khowarizmi (khoảng năm 825)
- Thuật toán nổi tiếng nhất, có từ thời kỳ cổ Hy Lạp là thuật toán Euclid.
- Là một phương pháp giải quyết vấn đề thích hợp để cài đặt như một chương trình máy tính.

7

Tại sao cần phải học Cấu trúc dữ liệu và Giải thuật?

◆ Algorithm:

- A finite sequence of steps for solving a logical or mathematical problem or performing a task. (*The Microsoft Computer Dictionary, Fifth Edition*)
- A computable set of steps to achieve a desired result.
- Informally, an **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**. An algorithm is thus a sequence of computational steps that transform the input into the output (*Introduction to Algorithms, 2nd, Thomas H. Cormen, 2001*)

8

Tại sao cần phải học Cấu trúc dữ liệu và Giải thuật?

◆ Cấu trúc dữ liệu?

- Được tạo ra để phục vụ cho các giải thuật
- Phải hiểu cấu trúc dữ liệu để hiểu giải thuật \Rightarrow để giải quyết vấn đề
- Các giải thuật đơn giản có thể cần đến cấu trúc dữ liệu phức tạp
- Các giải thuật phức tạp có thể chỉ dùng các cấu trúc dữ liệu đơn giản

Cấu trúc dữ liệu + Giải thuật = Chương trình

9

Kiểu dữ liệu

◆ ĐN kiểu dữ liệu

◆ Các thuộc tính của 1 kiểu dữ liệu

- Tên kiểu dữ liệu
- Miền giá trị
- Kích thước lưu trữ
- Tập các toán tử, phép toán tác động trên kiểu dữ liệu

◆ Một số kiểu dữ liệu có cấu trúc cơ bản

- Kiểu chuỗi ký tự (string), Kiểu mảng (array)
- Kiểu mẫu tin (struct)
- Kiểu tập hợp (union)

10

Tại sao cần phải học Cấu trúc dữ liệu và Giải thuật?

◆ Dùng máy tính để:

- Giải quyết các vấn đề về tính toán?
- Việc giải quyết vấn đề nhanh hơn?
- Khả năng truy xuất nhiều dữ liệu hơn?

◆ Kỹ thuật vs. Thực thi/Giá

- Kỹ thuật có thể tăng khả năng giải quyết vấn đề bằng nhân tố hằng.
- Thiết kế giải thuật tốt có thể giúp giải quyết vấn đề tốt hơn nhiều và có thể rẻ hơn.
- Một siêu máy tính không thể giúp một giải thuật tồi làm việc tốt hơn

11

Một số tính chất chung của các thuật toán

◆ Đầu vào (input):

có các giá trị đầu vào xác định.

◆ Đầu ra (output):

từ tập các giá trị đầu vào, thuật toán sẽ tạo các

giá trị đầu ra, xem là nghiệm của bài toán.

◆ Tính xác định (definiteness):

các bước của thuật toán phải được

xác định một cách chính xác.

◆ Tính hữu hạn (finiteness):

một thuật toán chứa một số hữu hạn

các bước (có thể rất lớn) với mọi tập đầu vào.

◆ Tính hiệu quả (effectiveness):

mỗi bước phải thực hiện chính xác,

trong khoảng thời gian hữu hạn.

◆ Tính tổng quát (general):

thuật toán phải áp dụng được cho một

họ các vấn đề.

12

Ví dụ

◆ **Ví dụ:** Mô tả thuật toán tìm phần tử lớn nhất trong một dãy hữu hạn các số nguyên

- **Bước 1:** Đặt giá trị cực đại tạm thời bằng số nguyên đầu tiên.
- **Bước 2:** Nếu số nguyên kế tiếp lớn hơn giá trị cực đại tạm thời thì gán giá trị cực đại tạm thời bằng số nguyên đó.
- **Bước 3:** Lặp lại bước 2 nếu còn số nguyên trong dãy.
- **Bước 4:** Dừng khi không còn số nguyên trên dãy. Giá trị cực đại tạm thời sẽ là số nguyên lớn nhất trong dãy.

13

Ví dụ so sánh - Tìm tuyến tính và tìm nhị phân

```
// Tìm tuyến tính  
int linearSearch(int a[], int x, int n)  
{  
    int i = 0;  
    while (i < n) && (a[i] != x)  
        i++;  
    return (i == n);  
}
```

14

Ví dụ so sánh - Tìm tuyến tính và tìm nhị phân

```
// Tìm nhị phân  
int binarySearch(int a[], int x, int n)  
{  
    int left = 0, right = N - 1, middle;  
    do {  
        middle = (left + right) / 2;  
        if (a[middle] == x)  
            return TRUE;  
        else if (x < a[middle])  
            right = middle - 1;  
        else left = middle + 1;  
    } while (left <= right);  
    return FALSE;  
}
```

15

Ví dụ so sánh - Tìm tuyến tính và tìm nhị phân

- ◆ Đễ so sánh hai giải thuật, sử dụng $n = 32$
- Với tìm kiếm tuần tự, giả sử x không có trong mảng a , giải thuật phải xử lý đầy đủ n lần.
 - Với tìm kiếm nhị phân, dù x có hay không có trong a thì số lần tìm kiếm tối đa chỉ là $\log_2 n = 5$.

16

Ví dụ so sánh - Tìm tuyến tính và tìm nhị phân

n	Tuần tự	Nhị phân
256	256	8
1024	1024	10
1048576	1048576	20
4.294.967.296	4.294.967.296	32

17

17

Ví dụ so sánh - Fibonacci

```
// Tính Fibonacci(n)
int fib1(int n)
{
    if (n <= 1)
        return n;
    else
        return fib1(n - 1) + fib1(n - 2);
}
```

Gọi $T(n)$ là số lượng các F_i ($0 \leq i \leq n$) được tính trong giải thuật đệ qui: $T(n) > 2^{n/2}$

18

18

Ví dụ so sánh - Fibonacci

```
// Tính Fibonacci(n)
int fib2(int n)
{
    f[0] = 0;
    f[1] = 1;
    for (i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];
    return f[n];
}
```

19

19

Ví dụ so sánh - Fibonacci

n	n + 1	$2^{n/2}$	fib2()	fib1()
40	41	1,048,576	41 ns	1048 μ s
60	61	1.1×10^9	61 ns	1 s
80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 years
160	161	1.2×10^{24}	161 ns	3.8×10^7 years
200	201	1.3×10^{30}	201 ns	4×10^{13} years

20

20

Phân tích thuật toán

- ◆ Độ phức tạp về không gian (space complexity)
- ◆ Độ phức tạp về thời gian (time complexity)
- ◆ Độ phức tạp về giải thuật (complexity of algorithm)

21

21

Độ phức tạp về không gian (space complexity)

- ◆ Chiếm tài nguyên của máy:
 - Bộ nhớ
 - Sử dụng CPU
 - Băng thông
 - ...

22

22

Độ phức tạp về thời gian (time complexity)

- ◆ Tính hiệu quả của giải thuật được kiểm tra bằng phương pháp thực nghiệm, thông qua các bộ dữ liệu thử:
 - Phụ thuộc vào ngôn ngữ lập trình.
 - Trình độ, kỹ năng có được của người viết.
 - Phần cứng (máy tính) dùng để thử nghiệm.
 - Sự phức tạp của việc xây dựng một bộ dữ liệu thử.

23

23

Tiêu chuẩn đánh giá một giải thuật là tốt

- ◆ Một giải thuật được xem là tốt nếu nó đạt các tiêu chuẩn sau:
 - Thực hiện đúng.
 - Tốn ít bộ nhớ.
 - Thực hiện nhanh.
- ◆ Trong khuôn khổ môn học này, chúng ta chỉ quan tâm đến tiêu chuẩn **thực hiện nhanh**.

24

24

Độ phức tạp về giải thuật (complexity of algorithm)

- ◆ Mang tính hình thức.
- ◆ Phép đo độc lập với máy tính, ngôn ngữ máy tính, người lập trình, ... hay những "tiểu tiết": tăng/giảm chỉ số vòng lặp, sự khởi tạo, gán, ...
- ◆ Thời gian thực thi của một giải thuật sẽ tăng theo kích thước dữ liệu và thời gian này tỉ lệ với số lượng các thao tác cơ sở.
- ◆ Độ phức tạp của giải thuật là một hàm số trên kích thước dữ liệu.

25

25

Độ phức tạp về giải thuật (complexity of algorithm)

- ◆ Những thao tác cơ sở có thể là:
 - Phép so sánh
 - Phép chuyển dời
 - Phép toán số học, ...
- ◆ Thao tác cơ sở là thao tác mang lại hiệu quả cao nhất.
- ◆ Trong các giải thuật sắp xếp, hai thao tác cơ bản là: so sánh và chuyển dời.

26

26

Độ phức tạp giải thuật (Algorithm Complexity)

- ◆ Thời gian thực hiện chương trình (Running Time).
 - Đơn vị đo thời gian thực hiện:
 - $T(n)$ trong đó n là kích thước (độ lớn) của dữ liệu vào. $T(n) \geq 0$ với mọi $n \geq 0$.
 - Thông thường người ta tính thời gian thực hiện xấu nhất (the worst case):
 - $T(n)$ là thời gian lớn nhất để thực hiện chương trình đối với mọi dữ liệu vào có cùng kích thước n .

27

27

Tỷ suất tăng (growth rate)

- ◆ $T(n)$ có tỷ suất tăng $f(n)$ nếu tồn tại các hằng số C và N_0 sao cho $T(n) \leq Cf(n)$ với mọi $n \geq N_0$.
- ◆ Ví dụ:
 - Giả sử $T(0) = 1$, $T(1) = 4$, tổng quát $T(n) = (n+1)^2$
 - Đặt $N_0 = 1$ và $C = 4$ thì $\forall n \geq 1$,
 - Ta có $T(n) = (n+1)^2 \leq 4n^2$ với mọi $n \geq 1$, tức là tỷ suất tăng của $T(n)$ là n^2 .
- ◆ Ví dụ:
 - $T(n) = 3n^3 + 2n^2$ là n^3 . Cho $N_0 = 0$ và $C = 5$ ta có với mọi $n \geq 0$ thì $3n^3 + 2n^2 \leq 5n^3$

28

28

Khái niệm độ phức tạp của giải thuật

- ◆ Giả sử ta có hai giải thuật P1 và P2
 - P1: $T_1(n) = 100n^2$ (tỷ suất tăng là n^2)
 - P2 : $T_2(n) = 5n^3$ (tỷ suất tăng n^3).
- ◆ Giải thuật nào sẽ thực hiện nhanh hơn?
 - Với $n < 20$ thì P2 sẽ nhanh hơn P1 ($T_2 < T_1$)?
 - Với $n > 20$ thì P2 sẽ nhanh hơn P1 ($T_2 < T_1$)?

29

Khái niệm độ phức tạp của giải thuật

- ◆ Khái niệm:
 - Cho một hàm $T(n)$, $T(n)$ gọi là có độ phức tạp $f(n)$ nếu tồn tại các hằng C, N_0 sao cho $T(n) \leq Cf(n)$ với mọi $n \geq N_0$ (tức là $T(n)$ có tỷ suất tăng là $f(n)$) và kí hiệu $T(n)$ là $O(f(n))$ (đọc là “ô của $f(n)$ ”)
- ◆ Ví dụ: $T(n) = (n+1)^2$ có tỷ suất tăng là n^2 nên $T(n) = (n+1)^2$ là $O(n^2)$
- ◆ Chú ý: $O(C.f(n)) = O(f(n))$ với C là hằng số. Đặc biệt $O(C) = O(1)$
- ◆ hàm thể hiện độ phức tạp có các dạng thường gặp sau: $\log_2 n, n, n\log_2 n, n^2, n^3, 2^n, n!, n^n$.

30

Tính toán độ phức tạp (Computational Complexity)

- ◆ Qui tắc cộng
 - Nếu $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình P1 và P2; và $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$ thì thời gian thực hiện của đoạn hai chương trình đó **nối tiếp nhau** là $T(n) = O(\max(f(n), g(n)))$
 - Ví dụ: Lệnh gán $x := 15$ tốn một hằng thời gian hay $O(1)$, và lệnh đọc dữ liệu $scanf("%d", &x)$ tốn một hằng thời gian hay $O(1)$. Vậy thời gian thực hiện cả hai lệnh trên nối tiếp nhau là $O(\max(1, 1)) = O(1)$

31

Tính toán độ phức tạp (Computational Complexity)

- ◆ Qui tắc nhân:
 - Nếu $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình P1 và P2 và $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$ thì thời gian thực hiện của đoạn hai đoạn chương trình đó **lồng nhau** là $T(n) = O(f(n).g(n))$

32

Tính toán độ phức tạp (Computational Complexity)

◆ Qui tắc tổng quát để phân tích một chương trình:

- lệnh gán, scanf, printf là $O(1)$
- chuỗi tuần tự các lệnh được xác định bằng qui tắc cộng
=> Như vậy thời gian này = thời gian thi hành một lệnh nào đó lâu nhất.
- Thời gian thực hiện cấu trúc IF là thời gian lớn nhất thực hiện lệnh sau THEN hoặc sau ELSE và thời gian kiểm tra điều kiện. Thường thời gian kiểm tra điều kiện là $O(1)$.
- Thời gian thực hiện vòng lặp là tổng (trên tất cả các lần lặp) thời gian thực hiện thân vòng lặp. Nếu thời gian thực hiện thân vòng lặp không đổi thì thời gian thực hiện vòng lặp là tích của số lần lặp với thời gian thực hiện thân vòng lặp.

33

Tính toán độ phức tạp (Computational Complexity)

```
void BubbleSort(int list[], int n)
{
    int i,j,temp;
    (1)   for(i=0;i<(n-1);i++)
    (2)       for(j=0;j<(n-(i+1));j++)      O((n-i).1) = O(n-i)
    (3)           if(list[j] > list[j+1])      O(1)
    (4)               { temp = list[j];      O(1)
    (5)                   list[j] = list[j+1];      O(1)
    (6)                   list[j+1] = temp;      O(1)
    }
}
```

$$T(n) = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2} = O(n^2).$$

34

34

Khái niệm độ phức tạp của giải thuật

- ◆ Giả sử ta có hai giải thuật P1 và P2 với **thời gian thực hiện** tương ứng là $T_1(n) = 100n^2$ (với tỷ suất tăng là n^2) và $T_2(n) = 5n^3$ (với tỷ suất tăng là n^3).
- ◆ Khi $n > 20$ thì $T_1 < T_2$. Sở dĩ như vậy là do tỷ suất tăng của T_1 nhỏ hơn tỷ suất tăng của T_2 .
- ◆ Như vậy một cách hợp lý là ta xét tỷ suất tăng của hàm thời gian thực hiện chương trình thay vì xét chính bản thân thời gian thực hiện.
- ◆ Cho một hàm $T(n)$, $T(n)$ gọi là có **độ phức tạp** $f(n)$ nếu tồn tại các hằng C, N_0 sao cho $T(n) \leq Cf(n)$ với mọi $n \geq N_0$ (tức là $T(n)$ có tỷ suất tăng là $f(n)$) và kí hiệu $T(n)$ là $O(f(n))$ (đọc là “ô của $f(n)$ ”).

35

Khái niệm độ phức tạp của giải thuật

- ◆ Chú ý: $O(C.f(n)) = O(f(n))$ với C là hằng số. Đặc biệt $O(C) = O(1)$
- ◆ Các hàm thể hiện độ phức tạp có các dạng thường gặp sau: **log₂n, n, n log₂n, n², n³, 2ⁿ, n!, nⁿ**.
- ◆ Ba hàm cuối cùng ta gọi là dạng hàm mũ, các hàm khác gọi là hàm đa thức.
- ◆ Một giải thuật mà thời gian thực hiện có độ phức tạp là một hàm đa thức thì chấp nhận được, còn các giải thuật có độ phức tạp hàm mũ thì phải tìm cách cải tiến giải thuật.
- ◆ Trong cách viết, ta thường dùng logn thay thế cho log₂n cho gọn.

36

36

Ví dụ 1: Thủ tục sắp xếp “nổi bọt”

```
void BubbleSort(int a[n])
{
    int i,j,temp;
/*1*/ for(i= 0; i<=n-2; i++)
/*2*/     for(j=n-1; j>=i+1;j--)
/*3*/         if (a[j].key < a[j-1].key) {
/*4*/             temp = a[j-1];
/*5*/             a[j-1] = a[j];
/*6*/             a[j] = temp;
        }
}
```

37

Tính thời gian thực hiện của thủ tục sắp xếp “nổi bọt”

- ◆ Đây là chương trình sử dụng các vòng lặp xác định. Toàn bộ chương trình chỉ gồm một lệnh lặp {1}, lồng trong lệnh {1} là lệnh lặp {2}, lồng trong lệnh {2} là lệnh {3} và lồng trong lệnh {3} là 3 lệnh nối tiếp nhau {4}, {5} và {6}.
- ◆ Trước hết, cả ba lệnh gán {4}, {5} và {6} đều tốn O(1) thời gian, việc so sánh $a[j-1] > a[j]$ cũng tốn O(1) thời gian, do đó lệnh {3} tốn O(1) thời gian.
- ◆ Vòng lặp {2} thực hiện (n-i) lần, mỗi lần O(1) do đó vòng lặp {2} tốn $O((n-i).1) = O(n-i)$.
- ◆ Vòng lặp {1} có i chạy từ 1 đến n-1 nên thời gian thực hiện của vòng lặp {1} và cũng là độ phức tạp của giải thuật là

$$T(n) = \sum_{i=1}^{n-1} (n - i) = \frac{n(n - 1)}{2} = O(n^2)$$

38

Tính độ phức tạp của hàm tìm kiếm tuần tự

```
int linearSearch(int a[], int x, int n)
{
/* 1*/ int index = 0;
/* 2*/ while (index < n) {
/* 3*/     if (a[index] == x) return 1;
/* 4*/     index++;
}
/* 5*/ return 0;
}
```

39

Tính độ phức tạp của hàm tìm kiếm tuần tự

- ◆ Ta thấy các lệnh {1}, {2}, và {5} nối tiếp nhau, do đó độ phức tạp của hàm linearSearch chính là độ phức tạp lớn nhất trong 3 lệnh này. Dễ dàng thấy rằng ba lệnh {1}, {5} đều có độ phức tạp O(1) do đó độ phức tạp của hàm linearSearch chính là độ phức tạp của lệnh {2}. Lồng trong lệnh {2} là lệnh {3} và {4}. Lệnh {3}, {4} có độ phức tạp O(1).
- ◆ Lệnh {2} là một vòng lặp không xác định, nên ta không biết nó sẽ lặp bao nhiêu lần, nhưng trong trường hợp xấu nhất (tất cả các phần tử của mảng a đều khác x, ta phải xét hết tất cả các $a[i]$, i có các giá trị từ 0 đến n-1) thì vòng lặp {2} thực hiện n lần, do đó lệnh {2} tốn O(n). Vậy ta có $T(n) = O(n)$.

40

Độ phức tạp về giải thuật (complexity of algorithm)

```
void ExchangeSort(int n, int a[])
{
    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] < a[j])
                swap(a[i], S[j]);
}
```

$$T(n) = (n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2}$$

41

41

Độ phức tạp về giải thuật (complexity of algorithm)

```
void MatrixMult(int n, A[], B[], C[])
{
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            {
                C[i][j] = 0;
                for (k = 0; k < n; k++)
                    C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
}
```

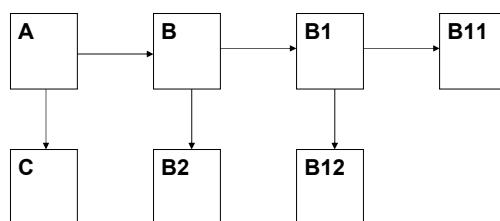
$$T(n) = n \times n \times n = n^3$$

42

42

Độ phức tạp của chương trình có gọi chương trình con không đệ quy

- Giả sử ta có một hệ thống các chương trình gọi nhau theo sơ đồ sau

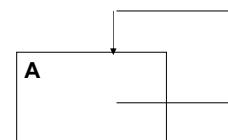


43

43

Phân tích các chương trình đệ qui

- Có thể thấy hình ảnh chương trình đệ quy A như sau:



- Để phân tích các chương trình đệ qui ta cần:
 - Thành lập phương trình đệ quy.
 - Giai phương trình đệ quy, nghiệm của phương trình đệ quy sẽ là thời gian thực hiện của chương trình đệ quy.

44

44

Chương trình đệ quy

- Để giải bài toán kích thước n, phải có ít nhất một trường hợp dừng ứng với một n cụ thể và lời gọi đệ quy để giải bài toán kích thước k ($k < n$).

- Ví dụ :** Chương trình đệ quy tính n!

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

- Trong ví dụ trên, n=0 là trường hợp dừng và $k=n-1$.

45

46

Thành lập phương trình đệ quy

- Phương trình đệ quy là một phương trình biểu diễn mối liên hệ giữa $T(n)$ và $T(k)$, trong đó $T(n)$ và $T(k)$ là thời gian thực hiện chương trình có kích thước dữ liệu nhập tương ứng là n và k, với $k < n$.

- Để thành lập được phương trình đệ quy, ta phải căn cứ vào chương trình đệ quy.

- Üng với trường hợp đệ quy dừng, ta phải xem xét khi đó chương trình làm gì và tốn hết bao nhiêu thời gian, chẳng hạn thời gian này là $c(n)$.

- Khi đệ quy chưa dừng thì phải xét xem có bao nhiêu lời gọi đệ quy với kích thước k ta sẽ có bấy nhiêu $T(k)$.

- Ngoài ra ta còn phải xem xét đến thời gian để phân chia bài toán và tổng hợp các lời giải, chẳng hạn thời gian này là $d(n)$.

45

46

Thành lập phương trình đệ quy

- Dạng tổng quát của một phương trình đệ quy sẽ là:

$$T(n) = \begin{cases} C(n) \\ F(T(k)) + d(n) \end{cases}$$

- $C(n)$ là thời gian thực hiện chương trình ứng với trường hợp đệ quy dừng.
- $F(T(k))$ là một đa thức của các $T(k)$.
- $d(n)$ là thời gian để phân chia bài toán và tổng hợp các kết quả.

47

Ví dụ về phương trình đệ quy của chương trình đệ quy tính n!

- Gọi $T(n)$ là thời gian tính $n!$.

- Thì $T(n-1)$ là thời gian tính $(n-1)!$.

- Trong trường hợp $n = 0$ thì chương trình chỉ thực hiện một lệnh return 1, nên tốn $O(1)$, do đó ta có $T(0) = C_1$.

- Trong trường hợp $n > 0$ chương trình phải gọi đệ quy $Giai_thua(n-1)$, việc gọi đệ quy này tốn $T(n-1)$, sau khi có kết quả của việc gọi đệ quy, chương trình phải nhân kết quả đó với n và return tích số.

- Thời gian để thực hiện phép nhân và return là một hằng C_2 . Vậy ta có phương trình:

$$T(n) = \begin{cases} C_1 & n = 0 \\ T(n-1) + C_2 & n > 0 \end{cases}$$

47

48

Giải thuật MergeSort

```
List MergeSort (List L; int n){
    List L1,L2
    if (n==1) RETURN(L);
    else {
        Chia đôi L thành L1 và L2, với độ dài n/2;
        RETURN(Merge(MergeSort(L1,n/2),MergeSort(L2,n/2)));
    };
}
```

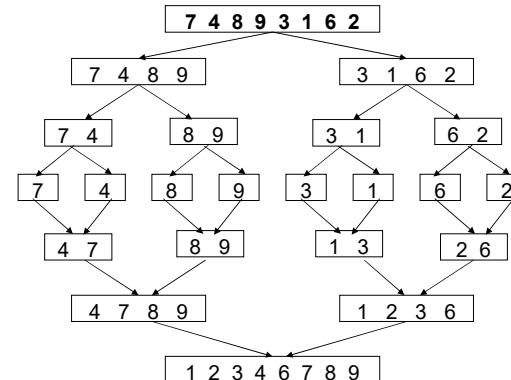
- ◆ Hàm MergeSort nhận một danh sách có độ dài n và trả về một danh sách đã được sắp xếp.
- ◆ Thủ tục Merge nhận hai danh sách đã được sắp L1 và L2 mỗi danh sách có độ dài n/2, trộn chúng lại với nhau để được một danh sách gồm n phần tử có thứ tự.

49

49

Mô hình minh họa Mergesort

- ◆ Sắp xếp danh sách L gồm 8 phần tử 7, 4, 8, 9, 3, 1, 6, 2



50

50

Phương trình đệ quy của giải thuật MergeSort

- ◆ Gọi $T(n)$ là thời gian thực hiện MergeSort một danh sách n phần tử
- ◆ Khi $T(1)$ là thời gian thực hiện MergeSort một danh sách 1 phần tử.
- ◆ Khi L có độ dài 1 ($n = 1$) thì chương trình chỉ làm một việc duy nhất là $return(L)$, việc này tốn $O(1) = C_1$ thời gian.
- ◆ Trong trường hợp $n > 1$, chương trình phải thực hiện gọi đệ quy MergeSort hai lần cho $L1$ và $L2$ với độ dài $n/2$ do đó thời gian để gọi hai lần đệ quy này là $2T(n/2)$.

51

51

Phương trình đệ quy của giải thuật MergeSort

- ◆ Ngoài ra còn phải tốn thời gian cho việc chia danh sách L thành hai nửa bằng nhau và trộn hai danh sách kết quả (Merge).
- ◆ Người ta xác định được thời gian để chia danh sách và Merge là $O(n) = C_2n$.
- ◆ Vậy ta có phương trình đệ quy như sau:

$$T(n) = \begin{cases} C_1 & n=1 \\ 2T\left(\frac{n}{2}\right) + C_2 n & n>1 \end{cases}$$

52

52

Giải phương trình đệ quy

- ◆ Có ba phương pháp giải phương trình đệ quy:
 - Phương pháp truy hồi.
 - Phương pháp đoán nghiệm.
 - Lời giải tổng quát của một lớp các phương trình đệ quy.

53

54

Các cấp thời gian thực hiện thuật toán (Typical growth rates)

Ký hiệu ô lớn (Big-O Notation) - Function	Tên gọi thông thường (name)
$O(1)$ hay C	Hằng (constant)
$O(\log n)$	Logarit (logarithmic)
$O(\log^2 n)$	Log-squared
$O(n)$	Tuyến tính (Linear)
$O(n \log n)$	n logarit
$O(n^2)$	Bình phương (Quadratic)
$O(n^3)$	Lập phương (Cubic)
$O(2^n)$	Mũ (exponential)

55

Các mức đánh giá

- ◆ Trường hợp tốt nhất (best case complexity)
- ◆ Trường hợp trung bình (average case complexity)
- ◆ Trường hợp xấu nhất (worst case complexity)

54

Qui tắc tính thời gian

- ◆ Luật 1: Cho các vòng lặp
 - Là thời gian lâu nhất của các lệnh trong vòng lặp
- ◆ Luật 2: Cho các vòng lặp lồng nhau
 - Phân tích từ trong ra ngoài. Thời gian thực hiện bằng tích thời gian các vòng lặp.
- ◆ Luật 3: Cho các lệnh liên tiếp nhau
 - Theo phương pháp cộng (max)

56

Qui tắc tính thời gian

◆ Thời gian : $O(n^2)$

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

◆ Thời gian $O(n^2)$ – phương pháp phép cộng (max)

```
for( i=0; i<n; i++ )  
    a[i] = 0;  
for( i=0; i<n; i++ )  
    for( j=0; j<n; j++ )  
        a[i] += a[j] + i + j;
```

57



Cảm ơn !

58

Chương 2. TÌM KIẾM VÀ SẮP XẾP

- ◆ Nhu cầu tìm kiếm và sắp xếp dữ liệu trong HTTT
- ◆ Các giải thuật tìm kiếm nội
 - Tìm kiếm tuyến tính
 - Tìm kiếm nhị phân
- ◆ Các giải thuật sắp xếp nội

1

Nhu cầu tìm kiếm và sắp xếp dữ liệu trong HTTT

- ◆ Nhu cầu?
- ◆ Các giải thuật tìm kiếm nội (Searching Techniques)
 - Tìm kiếm tuyến tính (Sequential Search)
 - Tìm kiếm nhị phân (Linear Search)

2

Mô tả bài toán

- ◆ Cho mảng A[1..n] các đối tượng, có các khóa key
- ◆ Chúng ta cần tìm trong mảng có phần tử nào có giá trị bằng x hay không?
- ◆ Lưu ý:
 - Khi cài đặt bằng ngôn ngữ C, do chỉ số mảng trong C bắt đầu từ 0 lên các giá trị chỉ số có thể chênh lệnh so với thuật toán
 - Để đơn giản: Dùng mảng các số nguyên làm cơ sở để cài đặt thuật toán.

3

Tìm kiếm tuyến tính (Sequential Search)

- ◆ Ý tưởng:
 - Đây là giải thuật tìm kiếm cổ điển
 - Thuật toán tiến hành so sánh x với lần lượt với phần tử thứ nhất, thứ hai,... của mảng a cho đến khi gặp được phần tử có khóa cần tìm.

4

Tìm kiếm tuyến tính

◆ Giải thuật

- **Bước 1:** i=1; //Bắt đầu từ phần tử đầu tiên
- **Bước 2:** So sánh a[i].key với x có 2 khả năng
 - a[i].key = x: Tìm thấy, Dừng;
 - a[i].key ≠ x: Sang bước 3;
- **Bước 3:**
 - i=i +1; //Xét tiếp phần tử kế tiếp trong mảng
 - Nếu i>n: hết mảng, không tìm thấy. Dừng
 - Ngược lại: lặp lại bước 2

5

6

Tìm kiếm tuyến tính – ví dụ

Tìm giá trị x =5, x=46, x=19

a	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	5	7	99	13	19	15	11	19	23	28	8	30	32	3	41	46

5

6

Tìm kiếm tuyến tính – cài đặt

◆ Cách 1

```
void LinearSearch(int a[], int N, int x)
{ int i, flag = 0;
  for(i=0;i<N;i++)
    if( a[i] == x)
    { printf("\nGiá trị %d ở vị trí %d trong mảng", x,i);
      flag =1; break;
    }
  if( flag == 0) printf("\nGiá trị %d không có trong mảng",
    x);
}
```

7

8

Tìm kiếm tuyến tính – cài đặt

◆ Cách 2

```
int LinearSearch (int a[], int N, int x)
{ int i=0;
  while ((i<N) && (a[i]!=x))
    i++;
  if (i==N)
    return -1 ; //Không có x, đã tìm hết mảng
  else
    return i; //Tìm thấy ở vị trí i
}
```

Tìm kiếm tuyến tính – cài đặt

◆ Cách 3

```
int LinearSearch (int a[], int N, int x)
{ int i=0;
  a[N] =x; //Thêm phần tử N+1
  while (a[i]!=x)
    ++
  if (i==N)
    return -1 ; //Không có x, đã tìm hết mảng
  else
    return i; //Tìm thấy ở vị trí i
}
```

9

Tìm kiếm tuyến tính – đánh giá

- ◆ Đánh giá giải thuật: Có thể ước lượng độ phức tạp của giải thuật tìm kiếm qua số lượng các phép so sánh được tiến hành để tìm ra x. Trường hợp giải thuật tìm tuyến tính, có:

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

- ◆ Vậy giải thuật tìm tuyến tính có độ phức tạp tính toán cấp n: $T(n) = O(n)$

10

Tìm kiếm tuyến tính (tuần tự)

◆ Nhận xét

- Không phụ thuộc vào thứ tự các phần tử trong mảng
- Một thuật toán có thể được cài đặt theo nhiều cách khác nhau, kỹ thuật cài đặt ảnh hưởng đến tốc độ thực hiện của thuật toán.

11

Tìm kiếm nhị phân (binary search)

- ◆ Bạn sẽ làm thế nào để tìm một tên chủ thuê bao trong danh bạ điện thoại, hoặc 1 từ (word) trong từ điển?

- Tìm nơi nào đó ở giữa (danh bạ, từ điển)
- So sánh nơi tên/từ nằm ở vị trí nào?
- Quyết định tìm kiếm ở nửa đầu hay nửa sau danh bạ.
- Lặp lại bước trên
- Đây chính là ý tưởng giải thuật tìm kiếm nhị phân (the binary search algorithm)**

12

Tìm kiếm nhị phân

◆ Giải thuật

- **Bước 1:** đặt left=1; right=N; //tìm kiếm tất cả các phần tử
- **Bước 2:** mid = (left+right)/2; //mốc so sánh
 - So sánh a[mid].key = x;
 - a[mid].key = x: Tìm thấy, Dừng;
 - a[mid].key > x : right = mid -1;
 - a[mid].key < x : left = mid +1;
- **Bước 3:**
 - Nếu left <= right
=> Tìm tiếp, lặp lại bước 2
 - Ngược lại: Dừng

13

13

Tìm kiếm nhị phân

Tìm trong mảng a, giá trị 36:

a	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	5	7	10	13	13	15	19	19	23	28	28	32	32	37	41	46

1. $(0+15)/2=7$; a[7]=19;
tìm trong 8..15
2. $(8+15)/2=11$; a[11]=32;
tìm trong 12..15
3. $(12+15)/2=13$; a[13]=37;
tìm trong 12..12
4. $(12+12)/2=12$; a[12]=32;
tìm trong 13..12 ...nhưng 13>12, => 36 không thấy

14

14

Tìm kiếm nhị phân

Tìm trong mảng a, giá trị 7:

a	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	5	7	10	13	13	15	19	19	23	28	28	32	32	37	41	46

1. $(0+15)/2=7$; a[7]=19;
tìm trong 0..6
2. $(0+6)/2=3$; a[3]=13;
tìm trong 0..2
3. $(0+2)/2=1$; a[1]=7;
Kết thúc

15

15

Tìm kiếm nhị phân – cài đặt

```
void BinarySearch(int a[], int N, int x)
{ int left, right, mid, flag = 0;
  left = 0; right = n-1;
  while(left <= right)
  {
    mid = (left+right)/2;
    if( a[mid] == x)
      {printf("\nGiá trị %d ở vị trí %d trong mảng", x,i);
       flag = 1; break; }
    else if(a[mid] < x) left = mid+1;
    else right = mid-1;
  } //while

  if( flag == 0) printf("\nGiá trị %d không có trong mảng", i);
}
```

16

16

Tìm kiếm nhị phân – cài đặt

```
int BinarySearch(int a[],int N, int x)
{ int left, right; mid ;
  left = 0; right= N-1;
  do
  {   mid = (left+right)/2;
      if( x==a[mid]) return mid; //thấy x tại vị trí mid
      else if(x< a[mid]) right= mid-1;
      else left= mid + 1;
  } while(left <= right);
  return -1;
}
```

17

17

Tìm kiếm nhị phân

◆ Nhận xét:

- Chỉ áp dụng cho dãy các phần tử đã có thứ tự
- Tiết kiệm thời gian hơn so với tìm kiếm tuần tự.
- Nếu dãy chưa được sắp xếp thứ tự?
 - Sắp xếp
 - Tìm kiếm
 - Tốn thời gian

18

18

Các giải thuật sắp xếp nội

- ◆ Mô tả bài toán
- ◆ Các phương pháp sắp xếp thông dụng
 - Phương pháp chọn trực tiếp – Selection Sort
 - Phương pháp chèn trực tiếp – Insertion sort
 - Phương pháp đổi chỗ trực tiếp – Interchange Sort
 - Phương pháp nổi bọt - Bubble Sort
 - Sắp xếp cây – Heap Sort
 - Sắp xếp với độ dài bước giảm dần – Shell Sort
 - Sắp xếp dựa trên phân hoạch – Quick Sort
 - Sắp xếp theo phương pháp trộn trực tiếp – Merge Sort
 - Sắp xếp theo phương pháp cơ sở - Radix Sort

19

19

Mô tả bài toán

- ◆ Sắp xếp là quá trình xử lý một danh sách các phần tử để đưa chúng về một thứ tự thỏa mãn tiêu chuẩn nào đó.
- ◆ Đối với bài giảng này:
 - Qui ước: sắp xếp thành mảng a, có N phần tử thành mảng có thứ tự tăng dần. Trong đó:
 - a[1] là phần tử có giá trị nhỏ nhất
 - a[N] là phần tử có giá trị lớn nhất

20

20

PP Chọn trực tiếp (Selection Sort)

◆ Ý tưởng

- Tìm phần tử có khóa nhỏ nhất trong mảng $a[1..N]$, giả sử đó là $a[k]$.
- Hóan đổi $a[k]$ với $a[1] \Rightarrow a[1]$ sẽ là phần tử nhỏ nhất
- Giả sử ta đã có $a[1].key \leq \dots \leq a[i-1].key$. Bây giờ ta tìm phần tử có khóa nhỏ nhất trong đoạn $[i..N]$
 - Giả sử tìm thấy $a[k]$, $i \leq k \leq N$.
 - Hóan đổi $a[k]$ với $a[i]$, ta được $a[1].key \leq \dots \leq a[i].key$
 - Lặp lại quá trình trên cho đến khi $i=N-1$, ta sẽ nhận được mảng a được sắp xếp

21

21

Selection Sort

◆ Giải thuật

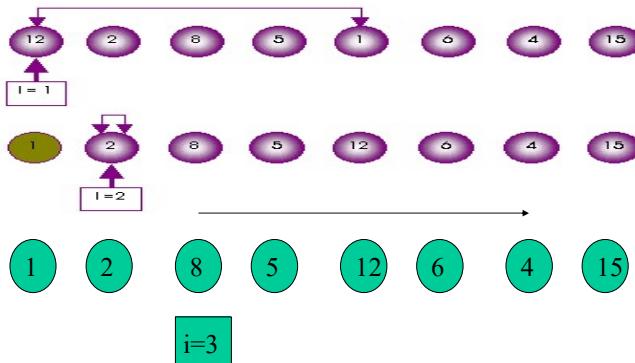
- Bước 1: $i=1$;
- Bước 2: Tìm phần tử $a[k]$ có khóa nhỏ nhất trong dãy hiện hành từ $a[i]$ đến $a[N]$
- Bước 3: Hóan vị $a[k]$ với $a[i]$
- Bước 4:
 - Nếu $i \leq N-1$ thì $i = i+1$; lặp lại bước 2.
 - Ngược lại: Dừng. //N-1 phần tử đã nằm đúng vị trí

22

22

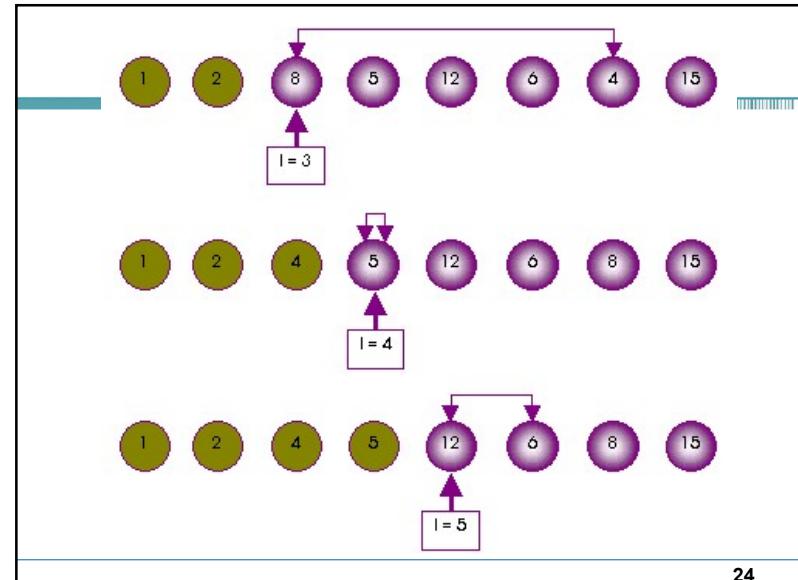
Selection Sort

◆ Cho dãy số: 12 2 8 5 1 6 4 15



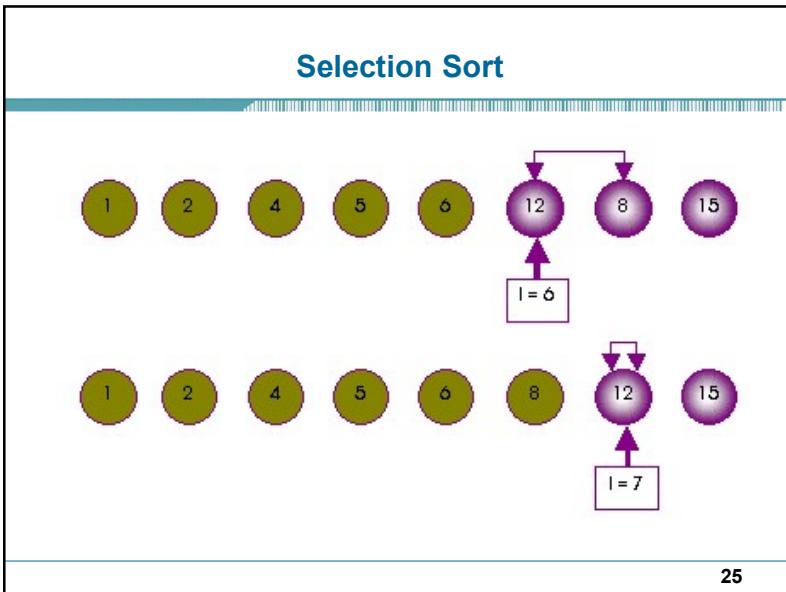
23

23

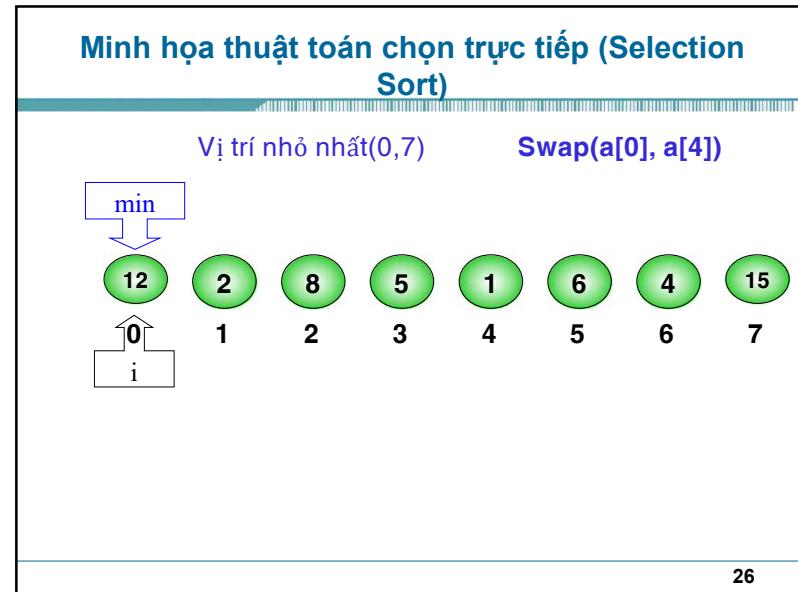


24

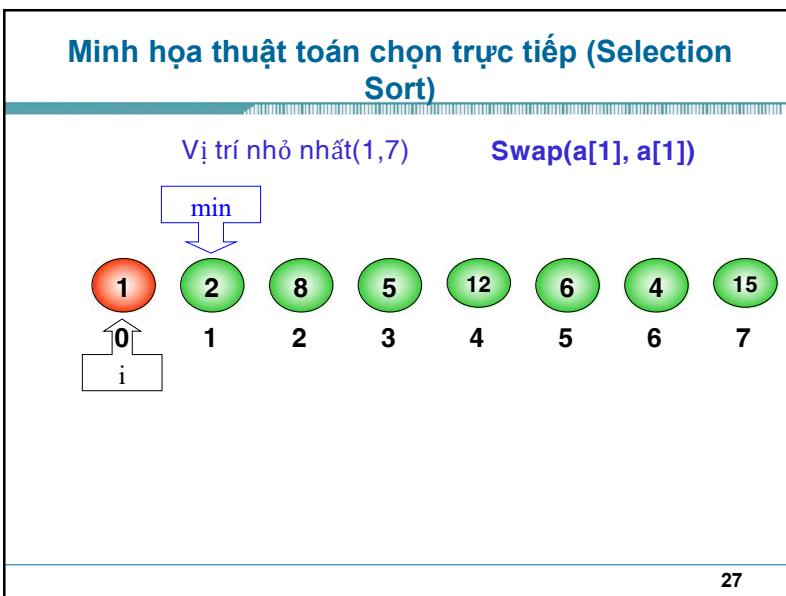
24



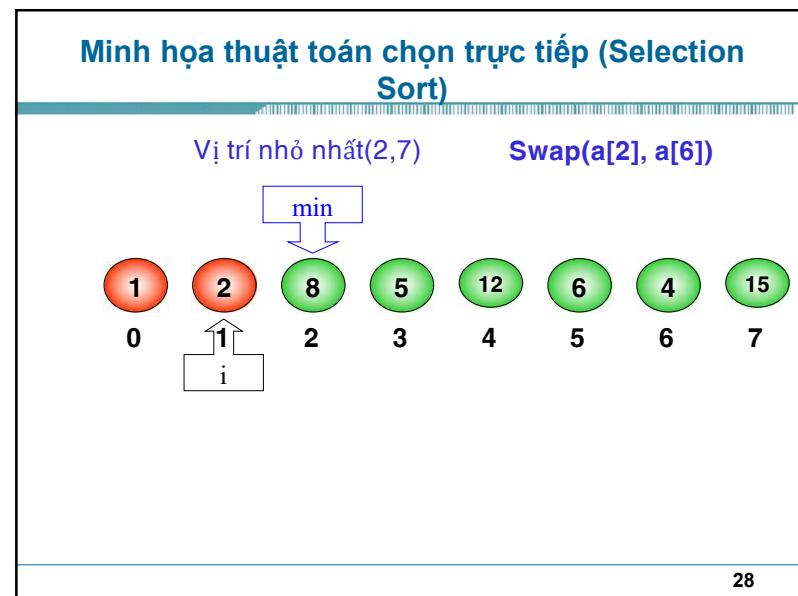
25



26



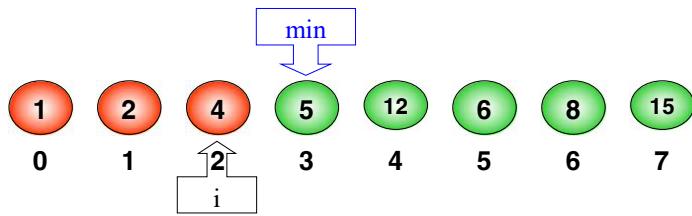
27



28

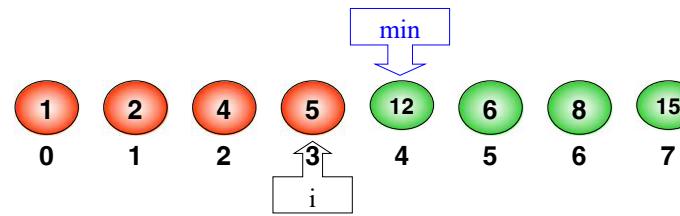
Minh họa thuật toán chọn trực tiếp (Selection Sort)

Vị trí nhỏ nhất(3, 7) Swap($a[3]$, $a[3]$)



Minh họa thuật toán chọn trực tiếp (Selection Sort)

Vị trí nhỏ nhất(4, 7) Swap($a[4]$, $a[5]$)

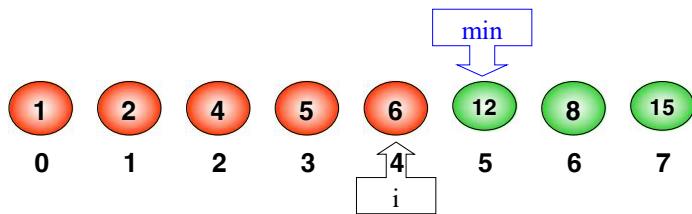


29

30

Minh họa thuật toán chọn trực tiếp (Selection Sort)

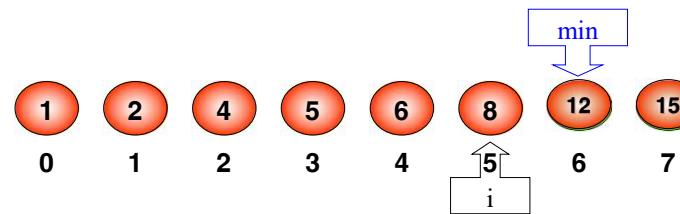
Vị trí nhỏ nhất(5, 7) Swap($a[5]$, $a[6]$)



31

Minh họa thuật toán chọn trực tiếp (Selection Sort)

Vị trí nhỏ nhất(6, 7)



32

Độ phức tạp của thuật toán Selection Sort

◆ Đánh giá giải thuật

$$\text{số lần so sánh} = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

Trường hợp	Số lần so sánh	Số phép gán
Tốt nhất	$n(n-1)/2$	0
Xấu nhất	$n(n-1)/2$	$3n(n-1)/2$

33

Selection Sort

◆ Cài đặt

```
void SelectionSort (int a[], int N)
{ int k; //chỉ số phần tử nhỏ nhất trong dãy
  for(int i=0; i<N-1; i++)
  {
    k = i;
    for(int j=i+1; j<N; j++)
      if (a[j] < a[k])
        k = j;// ghi nhận vị trí phần tử hiện nhỏ nhất
    hoandoi(a[k], a[i]);
  }
}
```

34

PP Chèn trực tiếp – Insertion Sort

◆ Ý tưởng

- Giả sử đoạn đầu của mảng $a[1..i-1]$ ($i \geq 2$) đã được sắp xếp, tức là ta có $a[1].key \leq \dots \leq a[i-1].key$
- Xen $a[i]$ vào vị trí thích hợp trong đoạn đầu $a[1..i-1]$ để nhận được đoạn đầu $a[1..i]$ được sắp xếp
- Lặp lại quá trình xen $a[i]$ như thế với i chạy từ 2 đến N , ta sẽ nhận được toàn bộ mảng $a[1..N]$ được sắp xếp.

35

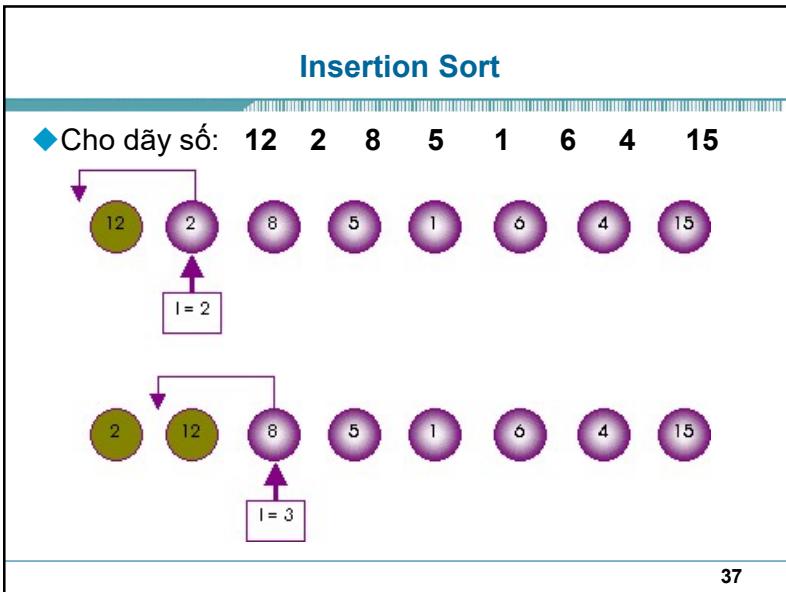
Insertion Sort

◆ Giải thuật

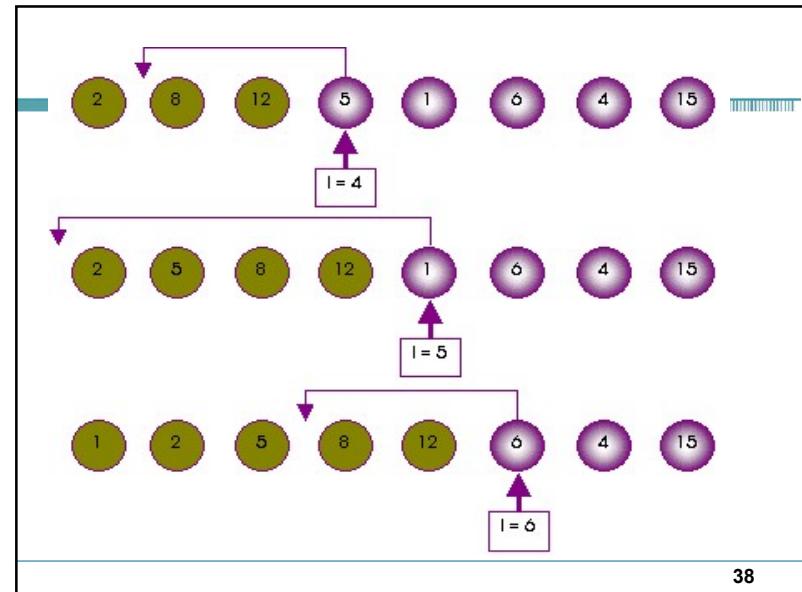
- Bước 1: $i=2$; //Giả sử đoạn $a[1]$ đã được sắp xếp
- Bước 2: $x=a[i].key$; Tìm vị trí pos thích hợp trong đoạn $a[1]$ đến $a[i-1]$ để chèn $a[i]$ vào
- Bước 3: Dời chỗ các phần tử từ $a[pos]$ đến $a[i-1]$ sang phải 1 vị trí để dành chỗ cho $a[i]$
- Bước 4: $a[pos].key = x$; //đoạn $a[1..a[i]]$ đã sắp xếp
- Bước 5:
 - $i=i+1$;
 - Nếu $i \leq N$: lặp lại bước 2
 - Ngược lại: Dừng

36

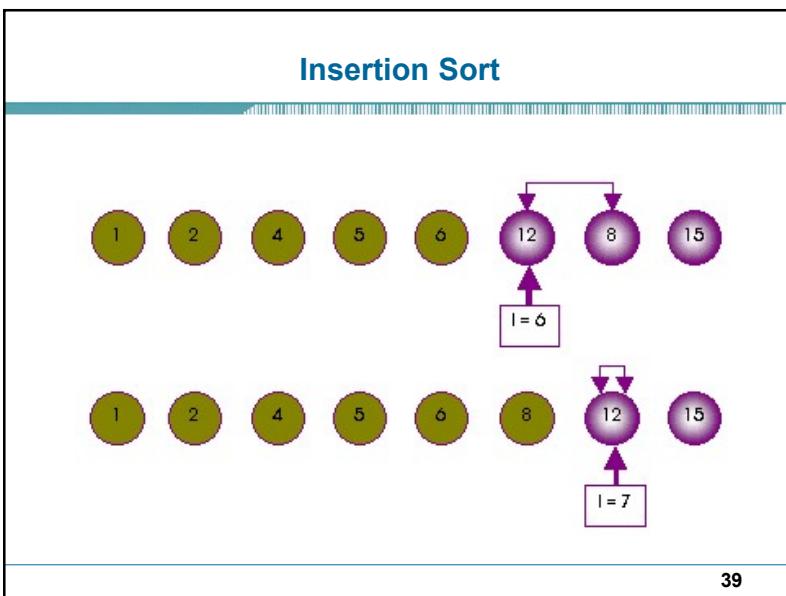
36



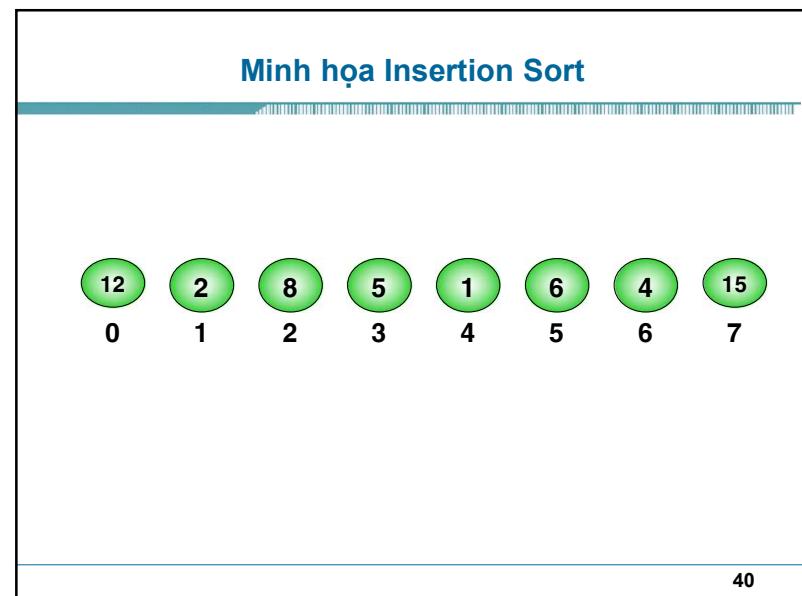
37



38



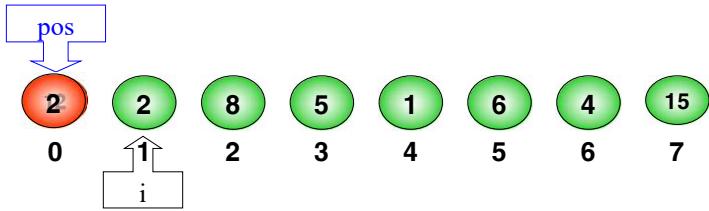
39



40

Minh họa Insertion Sort

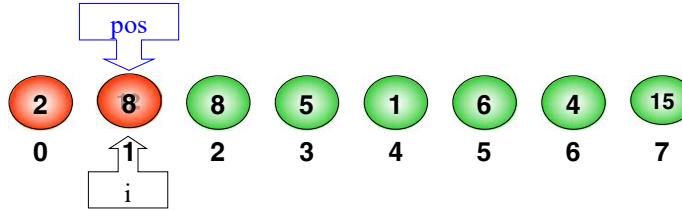
Insert a[1] into (0,0)



41

Minh họa Insertion Sort

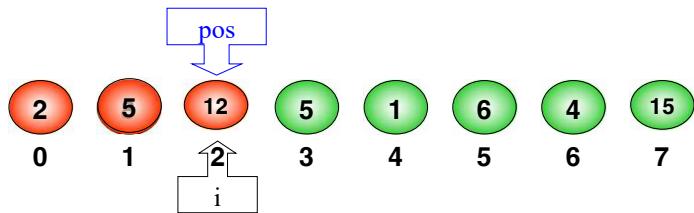
Insert a[2] into (0, 1)



42

Minh họa Insertion Sort

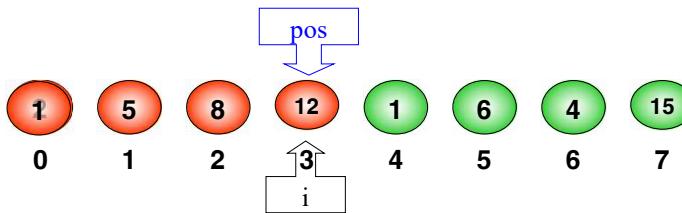
Insert a[3] into (0, 2)



43

Minh họa Insertion Sort

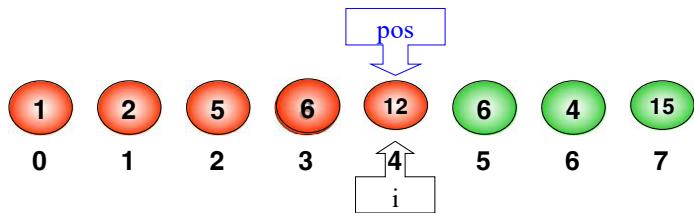
Insert a[4] into (0, 3)



44

Minh họa Insertion Sort

Insert a[5] into (0, 4)

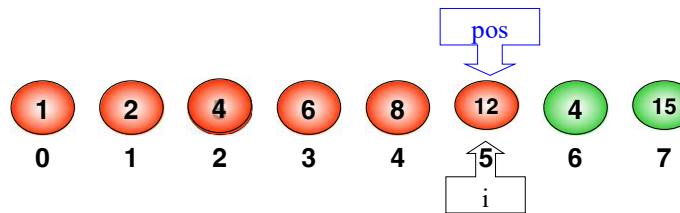


X

45

Minh họa Insertion Sort

Insert a[6] into (0, 5)

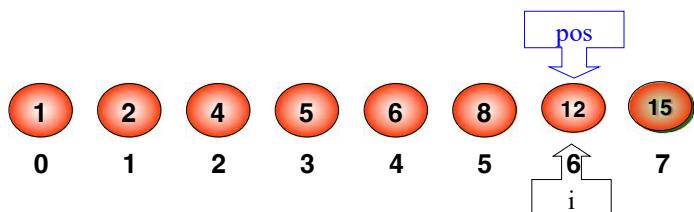


X

46

Minh họa Insertion Sort

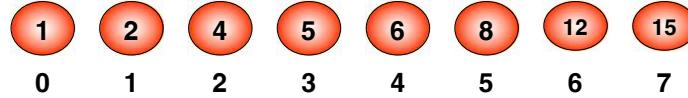
Insert a[8] into (0, 6)



X

47

Minh họa Insertion Sort



48

48

Độ phức tạp Insertion Sort

Trường hợp	Số phép so sánh	Số phép gán
Tốt nhất	$\sum_{i=1}^{n-1} 1 = n-1$	$\sum_{i=1}^{n-1} 2 = 2(n-1)$
Xấu nhất	$\sum_{i=1}^{n-1} (i-1) = \frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1} (i+1) = \frac{n(n+1)}{2} - 1$

49

Insertion Sort

◆ Cài đặt 1

```
void InsertionSort (int a[], int N)
{ int pos, i;
  int x; //lưu giá trị a[i] tránh bị đè khi dời chỗ các phần tử
  for(int i=1; i<N; i++) //đoạn a[0] đã sắp xếp
  {   x= a[i]; pos = i-1;
      while ((pos >= 0) && (a[pos] > x)) //Tìm vị trí chèn x;
          {   a[pos+1] = a[pos];
              pos--;
          }
      a[pos+1]=x;//chèn x vào
  }
}
```

50

Insertion Sort

◆ Cài đặt 2: Dành cho các dãy đã sắp xếp

```
void BinaryInsertionSort (int a[], int N)
{ int left, right, mid, i; int x;
  for(int i=1; i<N; i++) //đoạn a[0] đã sắp xếp
  {   x= a[i]; left = 1; right = i-1;
      while (left <= right) //Tìm vị trí chèn x;
          {   mid = (left + right) /2;
              if (x < a[mid])      right = mid -1;
              else left = mid +1;
          }
      for (int j = i-1; j>= left ; j--) a[j-1] = a[j];
      a[left] = x; //chèn x vào dãy
  }
}
```

51

PP Đổi chỗ trực tiếp (Interchange Sort)

◆ Ý tưởng

- Bắt đầu từ đầu dãy, tìm các phần tử có khóa nhỏ hơn nó, hoán đổi phần tử tìm được và phần tử đầu tiên
- Tiếp tục, thực hiện với phần tử thứ 2,...

52

51

52

Interchange Sort

◆ Giải thuật

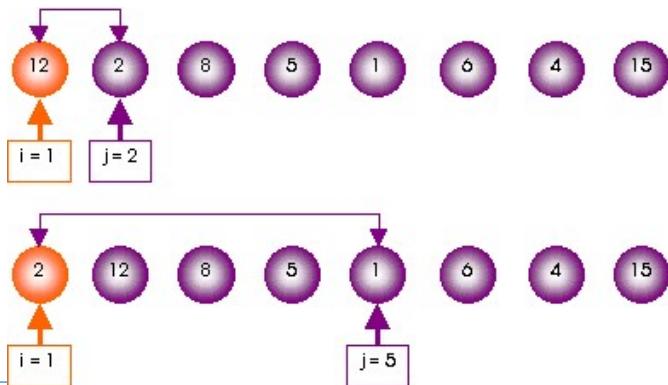
- Bước 1: $i=1$; //bắt đầu từ phần tử đầu dãy
- Bước 2: $j = i + 1$; //tìm các phần tử $a[j].key < a[i].key$, $j > i$
- Bước 3: trong khi $j \leq N$ thực hiện
 - nếu $a[j].key < a[i].key$ thì Hóan đổi($a[i]$, $a[j]$)
 - $j = j + 1$;
- Bước 4: $i = i+1$;
 - nếu $i < N$: lặp lại bước 2
 - ngược lại: Dừng

53

53

Interchange Sort

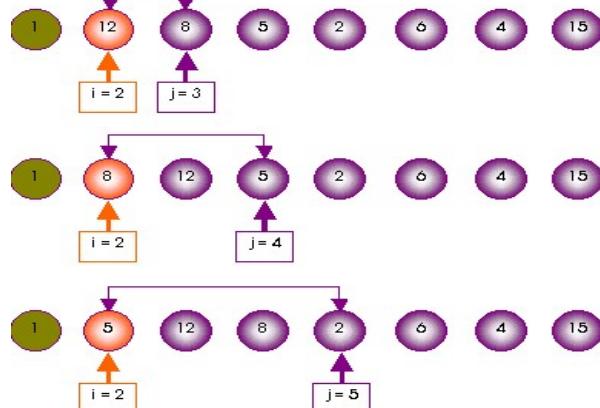
◆ Cho dãy số: 12 2 8 5 1 6 4 15



54

54

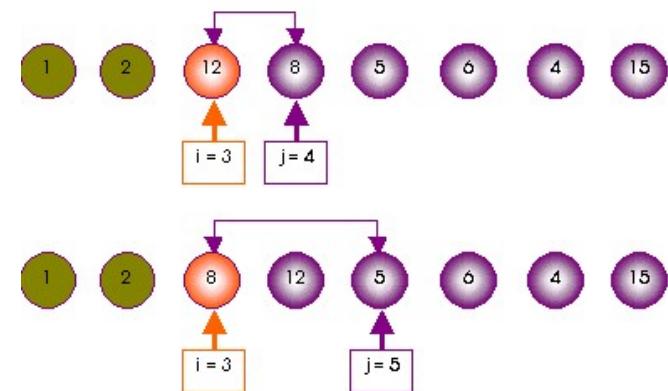
Interchange Sort



55

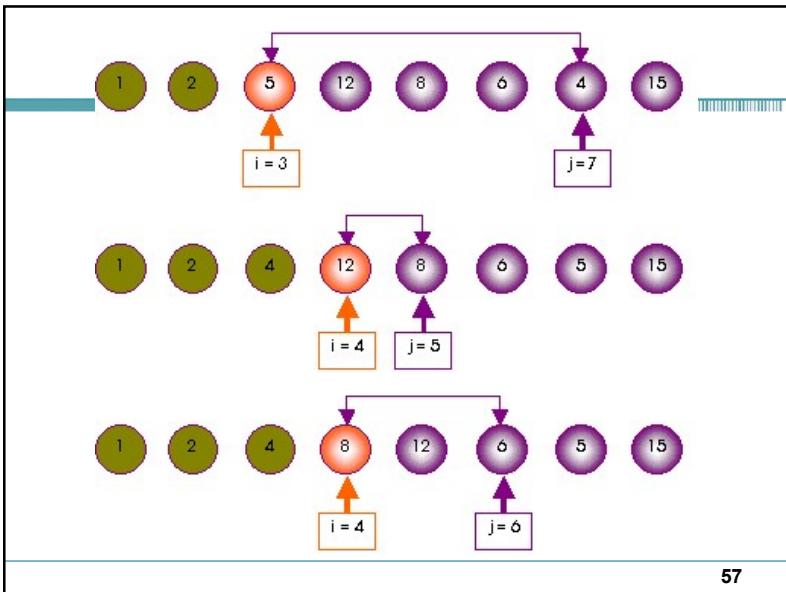
55

Interchange Sort

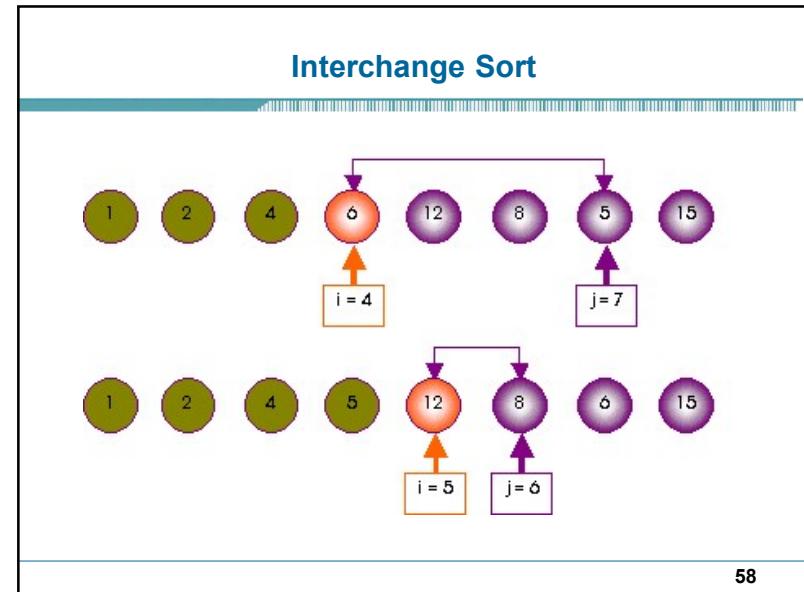


56

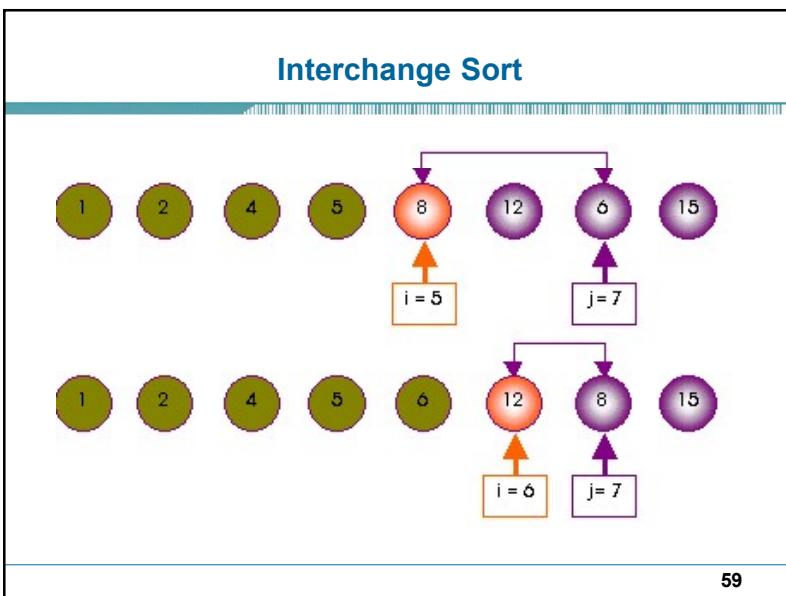
56



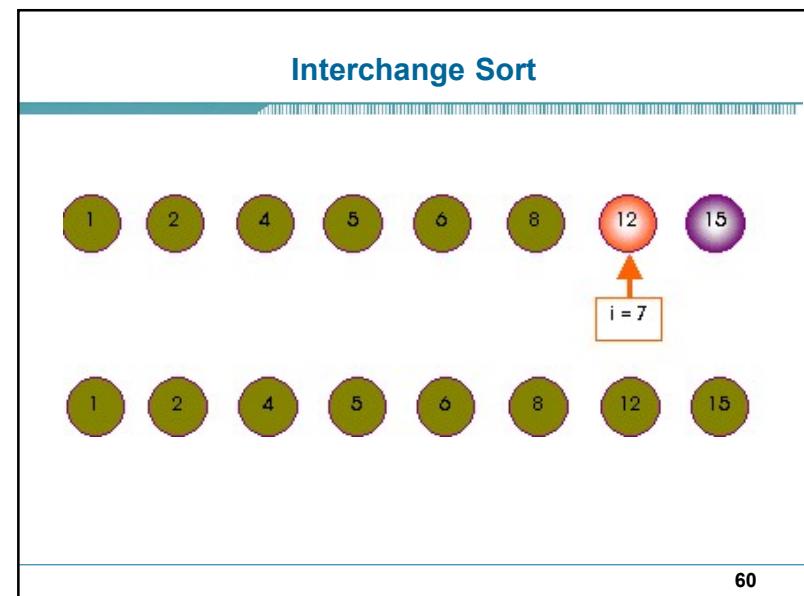
57



58

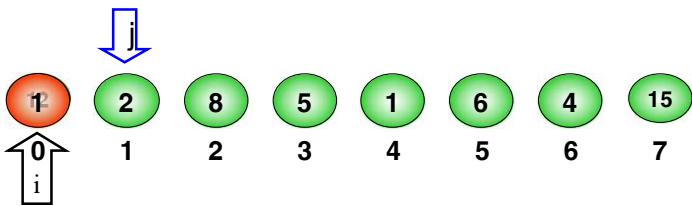


59



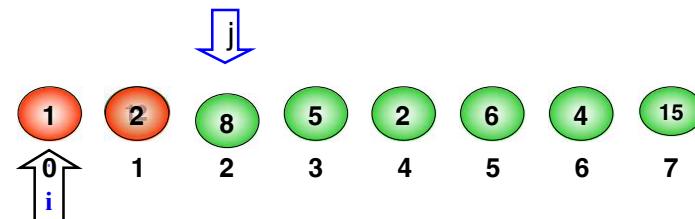
60

Minh họa thuật toán Interchange Sort



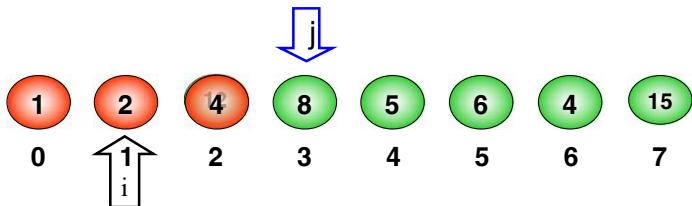
61

Minh họa thuật toán Interchange Sort



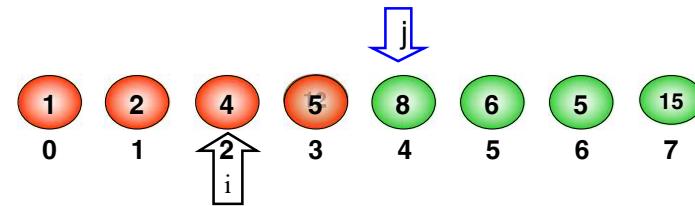
62

Minh họa thuật toán Interchange Sort



63

Minh họa thuật toán Interchange Sort



64

Minh họa thuật toán Interchange Sort



65

65

Độ phức tạp của thuật toán Interchange Sort

Trường hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất	$\sum_{i=1}^{n-1} (n-i+1) = \frac{n(n-1)}{2}$	0
Xấu nhất	$\frac{n(n-1)}{2}$	$\sum_{i=1}^{n-1} (n-i+1) = \frac{n(n-1)}{2}$

66

66

Interchange Sort

◆ Cài đặt

```
void InterchangeSort( int a[], int N)
{
    int i,j;
    for(i=0; i< N-1; i++)
        for(j = i+1; j<N; j++)
            if(a[j] < a[i]); //nếu sai vị trí thì đổi chỗ
            Hoanvi(a[j], a[i])
}
```

67

67

PP Nối bọt (Bubble Sort)

◆ Ý tưởng

- Xuất phát từ cuối dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ hơn trong cặp phần tử đó về vị trí đúng đầu dãy hiện hành, sau đó sẽ không xét nó ở các bước tiếp theo, do vậy ở lần xử lý thứ i sẽ có đầu dãy là i
- Qua quá trình sắp, mẫu tin nào có khóa “nhẹ” sẽ được nối lên trên.

68

68

Bubble Sort

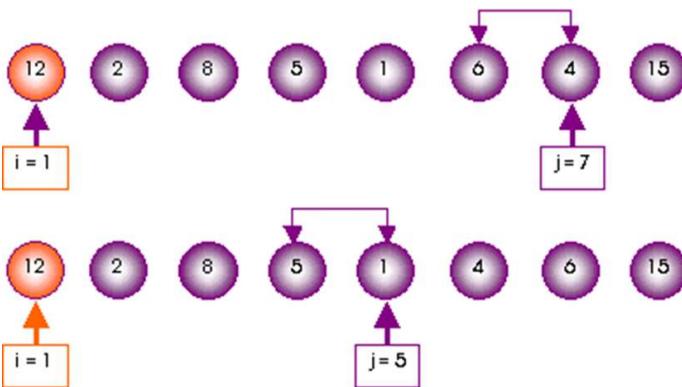
◆ Giải thuật

- Bước 1: $i=1$; //lần xử lý đầu tiên
- Bước 2: $j=N$; //Duyệt từ cuối dãy về vị trí i trong khi ($j > i$) thực hiện
 - nếu $a[j].key < a[j-1].key$ thì Hoanvi($a[j]$, $a[j-1]$); //xét cặp phần tử kế cận
 - $j = j - 1$;
- Bước 3: $i = i+1$; //lần xử lý kế tiếp
 - nếu $i > N - 1$: hết dãy, Dừng
 - Ngược lại: lặp lại bước 2

69

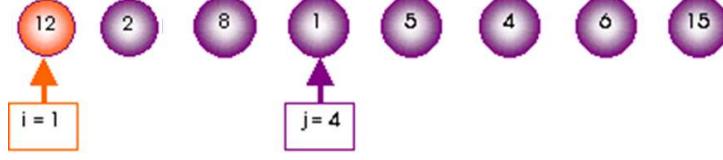
Bubble Sort

◆ Cho dãy số: 12 2 8 5 1 6 4 15



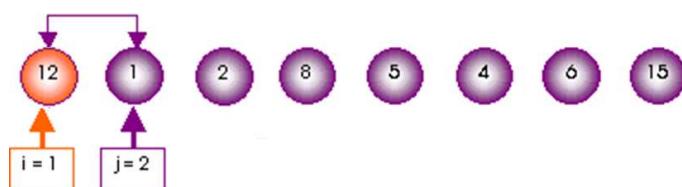
70

Bubble Sort

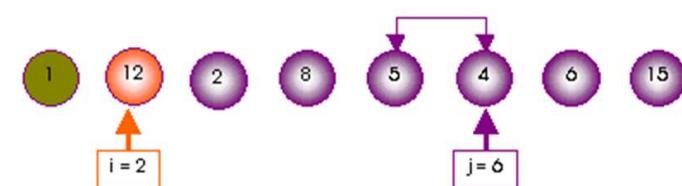


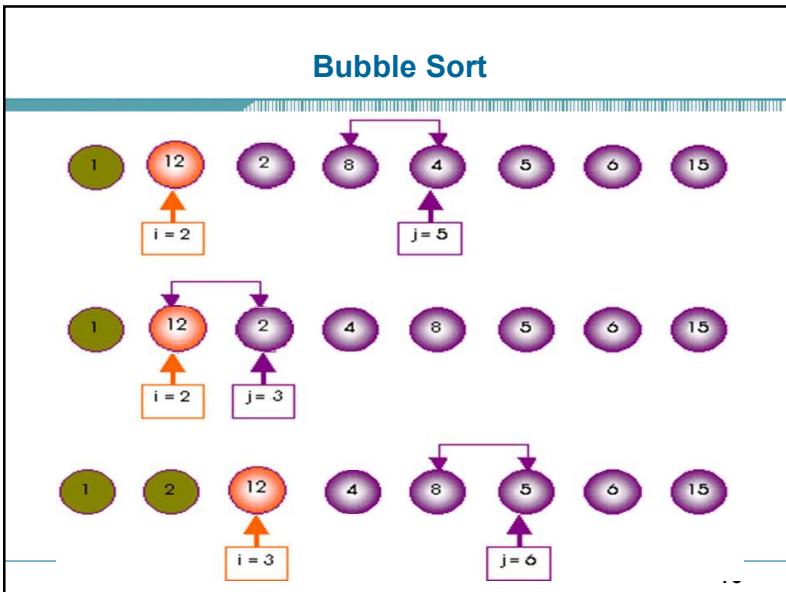
71

Bubble Sort

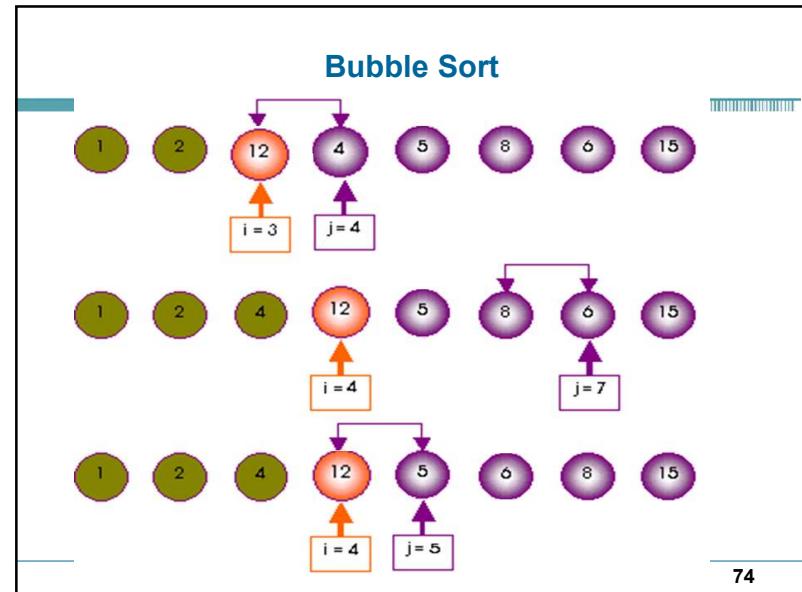


72

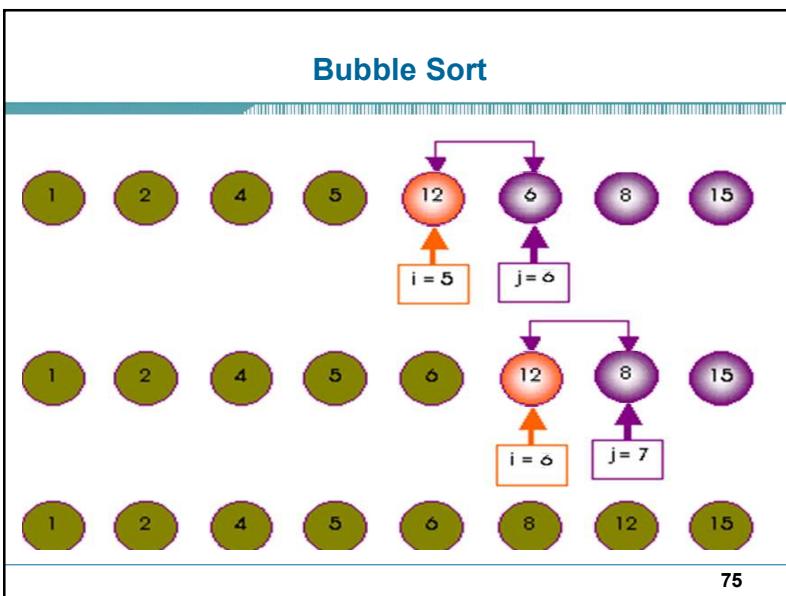




73



74

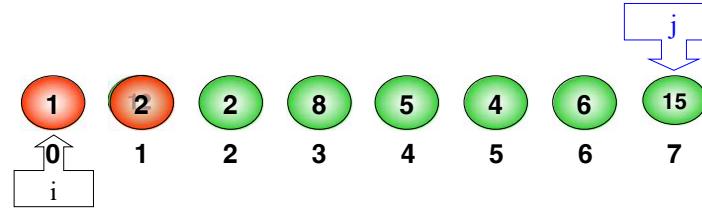


75



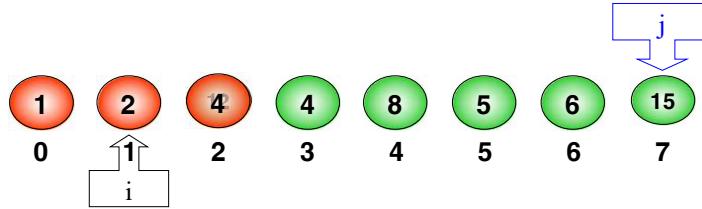
76

Minh họa Bubble Sort



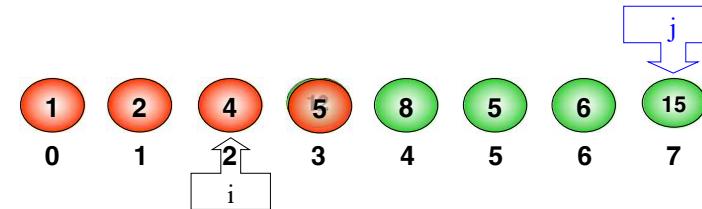
77

Minh họa Bubble Sort



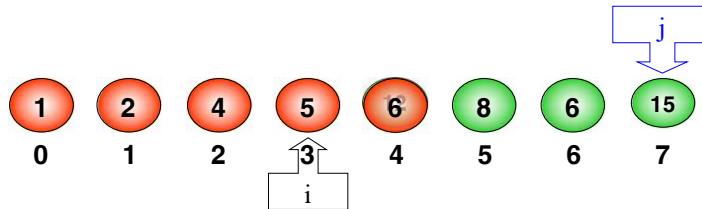
78

Minh họa Bubble Sort



79

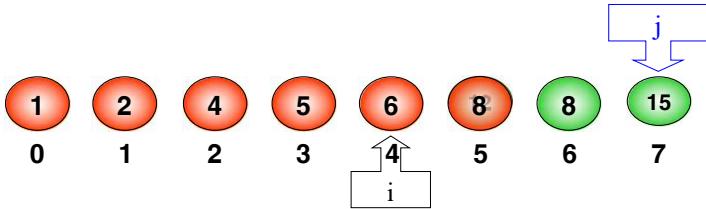
Minh họa Bubble Sort



80

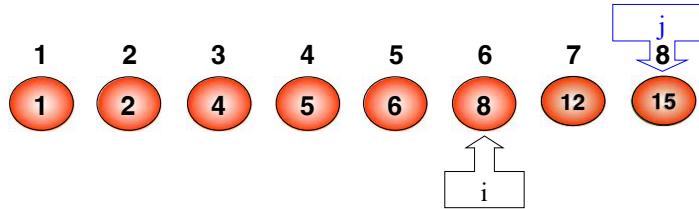
80

Minh họa Bubble Sort



81

Minh họa Bubble Sort



82

Độ phức tạp Bubble Sort

Trường hợp	Số lần so sánh	Số lần hoán vị
Tốt nhất	$\sum_{i=1}^{n-1} (n - i + 1) = \frac{n(n - 1)}{2}$	0
Xấu nhất	$\frac{n(n - 1)}{2}$	$\sum_{i=1}^{n-1} (n - i + 1) = \frac{n(n - 1)}{2}$

83

Bubble Sort

◆ Cài đặt

```
void BubbleSort(int a[], int N)
{ int i,j;
  for(i=0; i<(N-1);i++)
    for(j=N-1; j> i; j--)
      if(a[j] < a[j-1])
      {
        int t=a[j];
        a[j]=a[j-1];
        a[j-1]=t;
      }
}
```

84

Bubble Sort

◆ Nhận xét:

- Không nhận diện được dãy đã có thứ tự hay thứ tự từng phần.
- Các phần tử nhỏ được đưa về đúng vị trí rất nhanh, trong khi các phần tử lớn được đưa về vị trí đúng rất chậm

85

85

Quick Sort

- ◆ Là giải thuật dựa theo giải thuật Chia để trị (*divide-and-conquer recursive algorithm*).
- ◆ Để sắp xếp dãy a_1, a_2, \dots, a_n giải thuật QuickSort dựa trên việc phân hoạch dãy ban đầu thành 2 phần
 - Dãy con 1: Gồm các phần tử $a_1 \dots a_i$ có giá trị không lớn hơn x
 - Dãy con 2: Gồm các phần tử $a_i \dots a_n$ có giá trị không nhỏ hơn x
 - Với x là giá trị của phần tử tùy ý ban đầu. Sau khi thực hiện phân hoạch, dãy ban đầu được ban đầu thành 3 phần
 - $a_k < x$, với $k=1..i-1$;
 - $a_k = x$, với $k=i..j$
 - $a_k > x$, với $k=j+1..n$

86

86

Quick Sort

◆ Sau khi thực hiện phân hoạch, dãy ban đầu được phân thành 3 phần

1. $a_k < x$, với $k = 1..i$
2. $a_k = x$, với $k = i..j$
3. $a_k > x$, với $k = j..N$

$a_k < x$	$a_k = x$	$a_k > x$
-----------	-----------	-----------

87

87

Quick Sort

- ◆ Dãy con thứ 2 đã có thứ tự;
- ◆ Nếu các dãy con 1 và 3 chỉ có 1 phần tử: đã có thứ tự.
-> khi đó dãy ban đầu đã được sắp.

$a_k < x$	$a_k = x$	$a_k > x$
-----------	-----------	-----------

88

88

Quick Sort

- ◆ Dãy con thứ 2 đã có thứ tự;
- ◆ Nếu các dãy con 1 và 3 có nhiều hơn 1 phần tử thì dãy ban đầu chỉ có thứ tự khi các dãy con 1, 3 được sắp.
- ◆ Để sắp xếp dãy con 1 và 3, ta lần lượt tiến hành việc phân hoạch từng dãy con theo cùng phương pháp phân hoạch dãy ban đầu vừa trình bày .

$a_k < x$	$a_k = x$	$a_k > x$
-----------	-----------	-----------

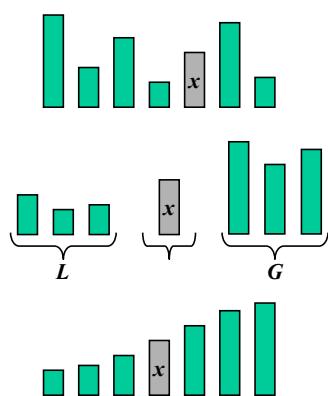
89

90

89

90

Quick Sort



91

92

91

92

Quick Sort

- ◆ **Ý tưởng:** sắp xếp mảng a gồm n phần tử $a[1..n]$
 1. Xác định một phần tử bất kỳ làm **chốt (pivot)**:
 2. Mảng được phân hoạch thành 2 phần bằng cách: $a[k]$
 - Chuyển tất cả những phần tử có **khóa nhỏ hơn** chốt sang bên trái chốt (**L**): $a[1]...a[k-1] < a[k]$
 - Chuyển tất cả những phần tử có **khóa lớn hơn** chốt sang bên phải chốt (**G**): $a[k+1]...a[n] \geq a[k]$
 3. Sắp xếp độc lập đệ qui bằng thuật toán trên với **L, G**
Giải thuật kết thúc khi mảng không có phần tử nào hoặc có 1 phần tử

90

90

Quick Sort

Giải thuật phân hoạch dãy $a[l..r]$ thành 2 dãy con

- ◆ **Bước 1:** Chọn tùy ý một phần tử $a[k]$ trong dãy là giá trị chốt (mốc, pivot), $l \leq k \leq r$; $x=a[k]$, $i=l$, $j=r$.
- ◆ **Bước 2:** Phát hiện và hiệu chỉnh cặp $a[i], a[j]$ nằm sai chỗ:
 - **Bước 2a:** Trong khi $(a[i] < x)$ $i++$
 - **Bước 2b:** Trong khi $(a[j] > x)$ $j- -$
 - **Bước 2c:** Nếu $i < j$ // $a[i] \geq x \geq a[j]$ mà $a[j]$ đứng sau $a[i]$
Hoán đổi $(a[i], a[j])$
- ◆ **Bước 3:**
 - Nếu $i < j$: Lặp lại bước 2 //chưa xét hết mảng
 - Nếu $i \geq j$: Dừng

92

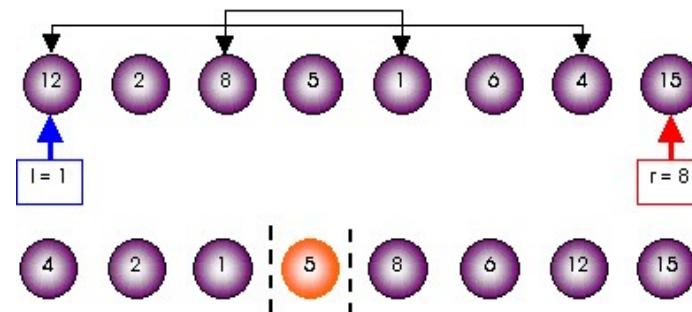
Quick sort

- ◆ Giải thuật để sắp xếp dãy a_i, a_{i+1}, \dots, a_r :
Có thể phát biểu một cách đẽ qui như sau
- ◆ Bước 1: Phân hoạch dãy a_i, a_{i+1}, \dots, a_r thành các dãy con:
 - Dãy con 1: $a_i \dots a_j < x$
 - Dãy con 2: $a_{j+1} \dots a_{i-1} = x$
 - Dãy con 3: $a_i \dots a_r > x$
- ◆ Bước 2:
 - Nếu ($i < j$) //dãy con 1 có nhiều hơn 1 phần tử
Phân hoạch dãy $a_i \dots a_j$
 - Nếu ($i < r$) //dãy con 3 có nhiều hơn 1 phần tử
Phân hoạch dãy $a_i \dots a_r$

93

Quick sort – ví dụ

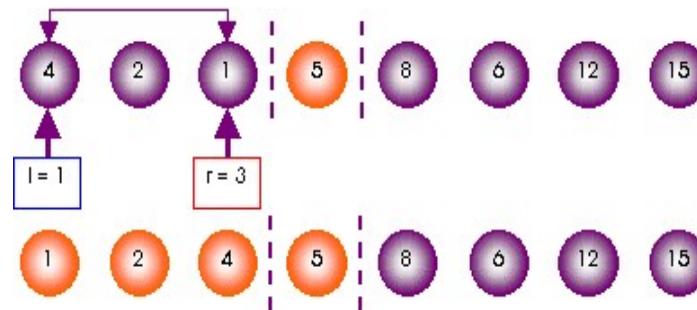
- ◆ Cho dãy số: 12 2 8 5 1 6 4 15
- ◆ Phân hoạch đoạn $i = 1, r = 8$: $x = A[4] = 5$



94

Quick sort – ví dụ

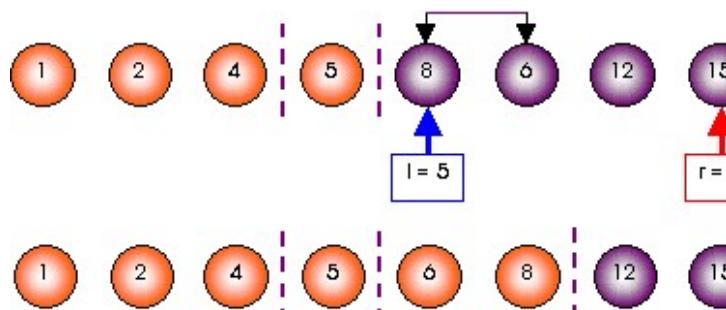
- ◆ Phân hoạch đoạn $i = 1, r = 3$: $x = A[2] = 2$



95

Quick sort – ví dụ

- ◆ Phân hoạch đoạn $i = 5, r = 8$: $x = A[6] = 6$



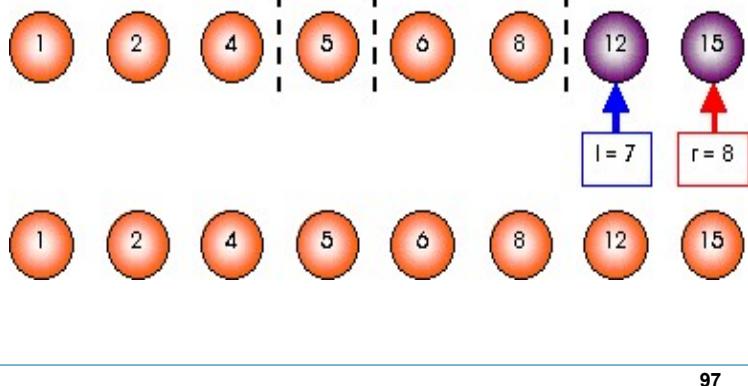
96

95

96

Quick sort – ví dụ

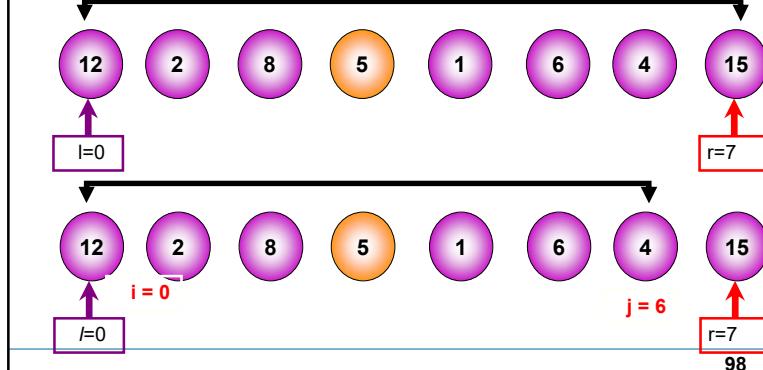
◆ Phân hoạch đoạn $l = 7, r = 8$: $x = A[7] = 6$



97

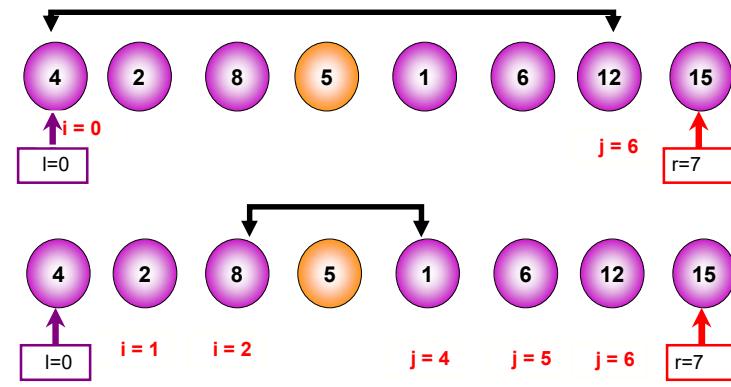
Quick sort – ví dụ

◆ dãy số: 12 2 8 5 1 6 4 15
Phân hoạch đoạn $l = 0, r = 7$: $x = a[3] = 5$



98

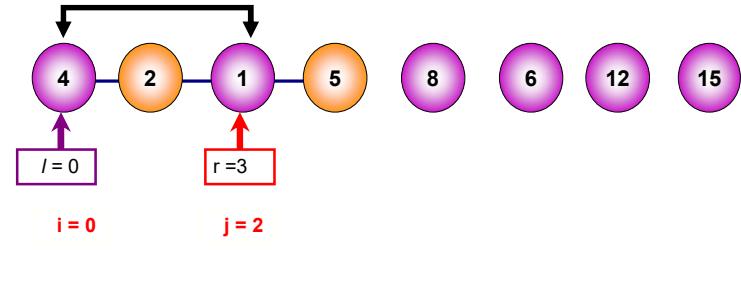
Quick sort – ví dụ



99

Quick sort – ví dụ

◆ Phân hoạch đoạn $l = 0, r = 2$: $x = a[1] = 2$



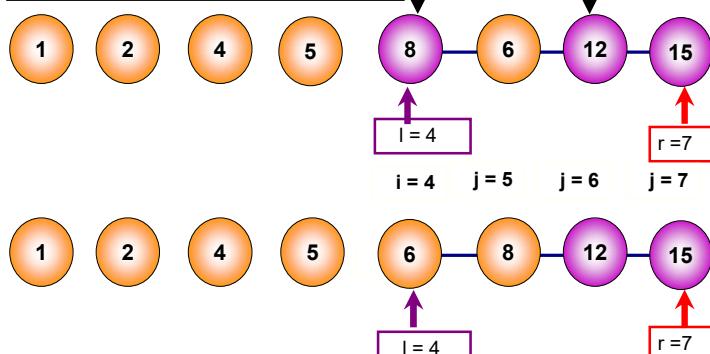
100

99

100

Quick sort – ví dụ

◆ Phân hoạch đoạn $l = 4, r = 7$:



101

Quick sort – ví dụ

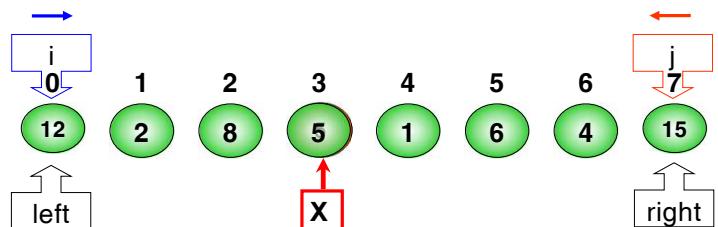
◆ Phân hoạch đoạn $l = 6, r = 7$:



102

Minh họa Quick Sort

➤ Phân hoạch đoạn $[0, 7]$

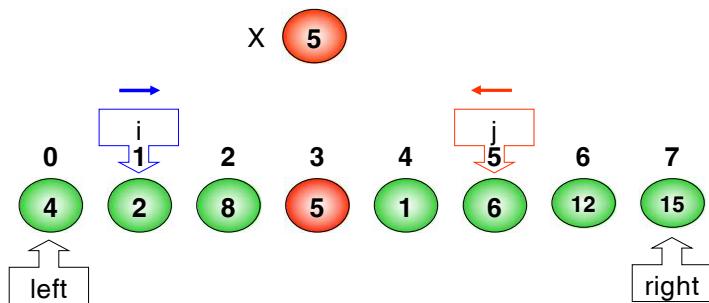


103

103

Minh họa Quick Sort

➤ Phân hoạch đoạn $[0, 7]$

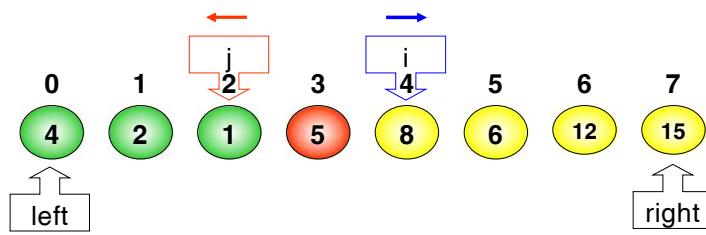


104

104

Minh họa Quick Sort

➤ Phân hoạch đoạn [0,2]

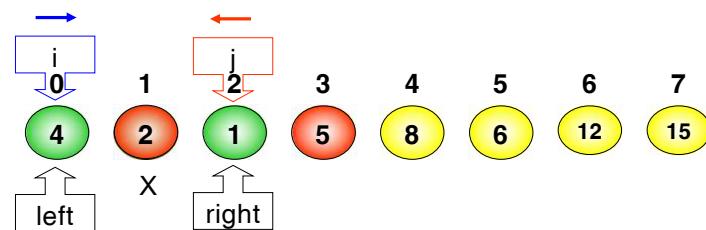


105

105

Minh họa Quick Sort

➤ Phân hoạch đoạn [0,2]

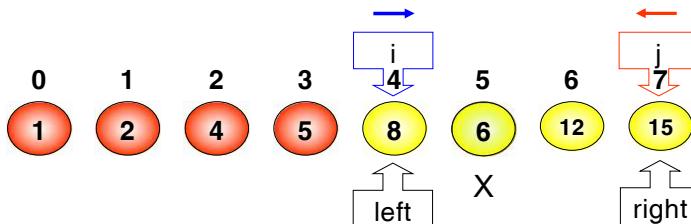


106

106

Minh họa Quick Sort

➤ Phân hoạch đoạn [4,7]

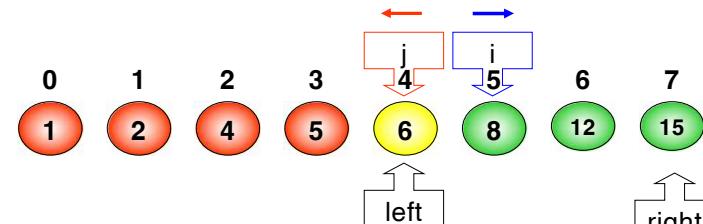


107

107

Minh họa Quick Sort

➤ Phân hoạch đoạn [5,7]

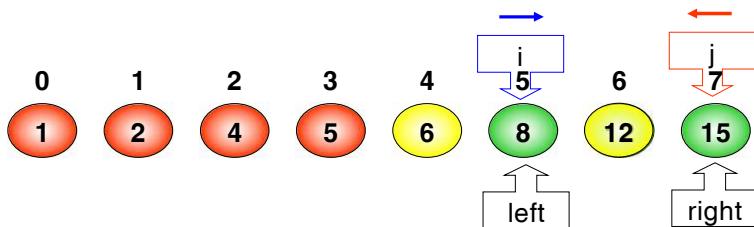


108

108

Minh họa Quick Sort

➤ Phân hoạch đoạn [5,7]



109

Minh họa Quick Sort



110

Độ phức tạp Quick Sort

Trường hợp	Độ phức tạp
Tốt nhất	$n * \log(n)$
Trung bình	$n * \log(n)$
Xấu nhất	n^2

111

Quick Sort – Cài đặt

```
void QuickSort(int a[], int l, int r)
{ int x,i,j;
  x = a[(l+r)/2] ; //mốc
  i = l; j = r;
  do {
    while (a[i]< x) i++;
    while (a[j]> x) j--;
    if (i <= j)
      { Hoandoi(a[i], a[j]);
        i++; j--;
      }
  } while (i <= j)
```

```
if ( l < j)
  QuickSort(a, l, j);
if (i < r)
  QuickSort(a, i, r);
}
```

112

111

112

Merge Sort

- ◆ **Ý tưởng:** Giải thuật Merge sort sắp xếp dãy a_1, a_2, \dots, a_n dựa trên nhận xét sau:
 - Mỗi dãy a_1, a_2, \dots, a_n đều có thể coi như là một tập hợp các dãy con liên tiếp đã sắp thứ tự.
 - Ví dụ dãy 12, 2, 8, 5, 1, 6, 4, 15 có thể coi như gồm 5 dãy con không giảm (12); (2, 8); (5); (1, 6); (4, 15).
 - Dãy đã có thứ tự coi như có 1 dãy con.
 - **Cách tiếp cận:**
 - tìm cách làm giảm số dãy con không giảm.

113

113

Merge Sort

- ◆ Tách dãy $a[1..n]$ thành 2 dãy phụ theo nguyên tắc phân phối đều luân phiên
- ◆ Trộn từng cặp dãy con của 2 dãy phụ thành một dãy con của dãy ban đầu
- ◆ Lặp lại qui trình trên cho đến khi nhận được một dãy con không giảm

114

114

Merge Sort

- ◆ Giải thuật
 - **Bước 1:** $k=1$; //chiều dài dãy con trong bước hiện hành
 - **Bước 2:** Tách dãy a_1, a_2, \dots, a_n thành 2 dãy b,c theo nguyên tắc luân phiên từng nhóm k phần tử
 - $b = a_1, \dots, a_k, a_{2k+1}, \dots, a_{3k}, \dots$
 - $c = a_{k+1}, \dots, a_{2k}, a_{3k+1}, \dots, a_{4k}, \dots$
 - **Bước 3:** Trộn từng cặp dãy con gồm k phần tử của 2 dãy b, c vào 1
 - **Bước 4:** $k=k*2$
 - Nếu $k < n$ thì trở lại bước 2
 - Ngược lại: Dừng

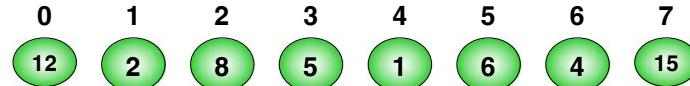
115

115

Minh họa Merge Sort

➤ $k=1$

➤ Phân phối luân phiên



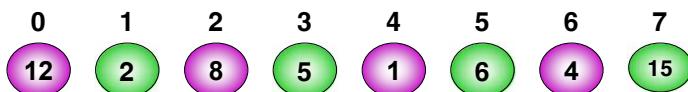
116

116

Minh họa Merge Sort

➤ $k = 1$

➤ Phân phối luân phiên

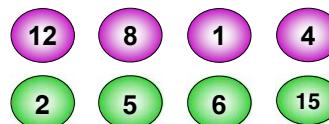


117

Minh họa Merge Sort

➤ Trộn từng cặp đường chạy

0 1 2 3 4 5 6 7



118

117

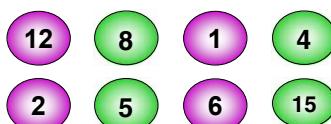
118

Minh họa Merge Sort

➤ $k = 1$

➤ Trộn từng cặp đường chạy

0 1 2 3 4 5 6 7



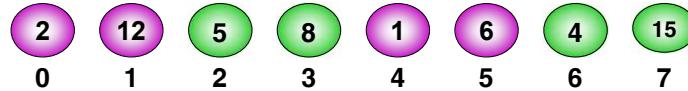
119

119

Minh họa Merge Sort

➤ $k = 2$

➤ Phân phối luân phiên



120

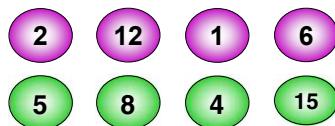
120

Minh họa Merge Sort

➤ $k = 2$

➤Trộn từng cặp đường chạy

0 1 2 3 4 5 6 7



121

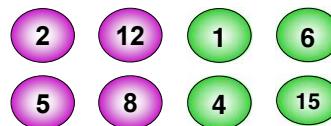
122

Minh họa Merge Sort

➤ $k = 2$

➤Trộn từng cặp đường chạy

0 1 2 3 4 5 6 7

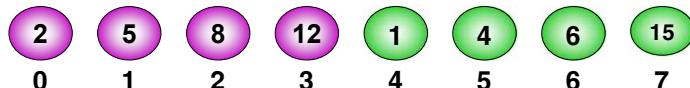


122

Minh họa Merge Sort

➤ $k = 4$

➤Phân phối luân phiên



123

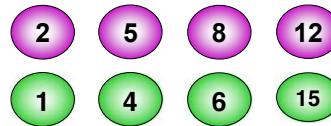
124

Minh họa Merge Sort

➤ $k = 4$

➤Trộn từng cặp đường chạy

0 1 2 3 4 5 6 7



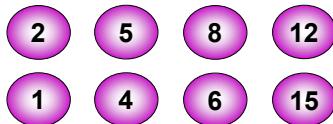
124

Minh họa Merge Sort

➤ $k = 4$

➤Trộn từng cặp đường chạy

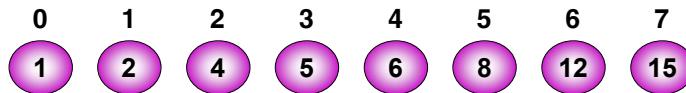
0 1 2 3 4 5 6 7



125

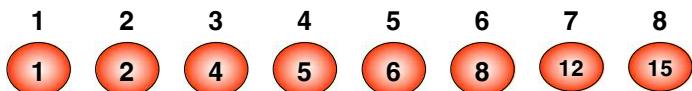
Minh họa Merge Sort

➤ $k = 8$



126

Minh họa Merge Sort



127

Sắp xếp cây - Heap sort

◆ Giải thuật Sắp xếp cây

- ◆ PPSX chọn trực tiếp không tận dụng được các thông tin đã có được do các phép so sánh ở bước $i-1$ khi tìm phần tử nhỏ nhất ở bước i ,
- ◆ Mấu chốt để giải quyết vấn đề vừa nêu là phải tìm ra được một cấu trúc dữ liệu cho phép tích lũy các thông tin về sự so sánh giá trị các phần tử trong qua trình sắp xếp.

128

128

Heap sort

- Giả sử dữ liệu cần sắp xếp là dãy số : 5 2 6 4 8 1
được bố trí theo quan hệ so sánh và tạo thành sơ đồ dạng cây như sau:

Mức 0:

8

Mức 1:

6

8

Mức 2:

5

6

$-\infty$

Mức 3:

5

2

6

4

8

1

129

129

Heap sort

- Heap Sort tận dụng được các phép so sánh ở bước i-1 mà thuật toán sắp xếp chọn trực tiếp không tận dụng được
- Để làm được điều này Heap sort thao tác dựa trên cây.

130

Thuật toán sắp xếp Heap Sort

- Ở cây trên, phần tử ở mức i chính là phần tử lớn trong cặp phần tử ở mức i +1, do đó phần tử ở nút gốc là phần tử lớn nhất.
- Nếu loại bỏ gốc ra khỏi cây, thì việc cập nhật cây chỉ xảy ra trên những nhánh liên quan đến phần tử mới loại bỏ, còn các nhánh khác thì bảo toàn.
- Bước kế tiếp có thể sử dụng lại kết quả so sánh của bước hiện tại.
- Vì thế độ phức tạp của thuật toán $O(n \log_2 n)$

131

131

Heap sort

- Định nghĩa Heap :

Giả sử xét trường hợp sắp xếp tăng dần, khi đó Heap được định nghĩa là một dãy các phần tử a_1, a_2, \dots, a_r thoả các quan hệ sau với mọi $i \in [l, r]$:

- $a_i \geq a_{2i}$
- $a_i \geq a_{2i+1}$
 $\{(a_i, a_{2i}), (a_i, a_{2i+1})\}$ là các cặp phần tử liên đới }

132

132

Heap sort

Heap có các tính chất sau :

- ◆ **Tính chất 1:** Nếu a_1, a_2, \dots, a_r là một heap thì khi cắt bỏ một số phần tử ở hai đầu của heap, dãy con còn lại vẫn là một heap.
- ◆ **Tính chất 2:** Nếu a_1, a_2, \dots, a_n là một heap thì phần tử a_1 (đầu heap) luôn là phần tử lớn nhất trong heap.
- ◆ **Tính chất 3:** Mọi dãy a_1, a_{l+1}, \dots, a_r với $2l > r$ là một heap.

133

133

Các bước thuật toán

- ◆ **Giai đoạn 1:** Hiệu chỉnh dãy số ban đầu thành heap
- ◆ **Giai đoạn 2:** Sắp xếp dãy số dựa trên heap:
 - **Bước 1:** Đưa phần tử lớn nhất về vị trí đúng ở cuối dãy:
 $r = n-1$; Swap (a_1, a_r);
 - **Bước 2:** Loại bỏ phần tử lớn nhất ra khỏi heap:
 $r = r-1$;
Hiệu chỉnh phần còn lại của dãy từ a_1, a_2, \dots, a_r thành một heap.
 - **Bước 3:**
Nếu $r > 1$ (heap còn phần tử): Lặp lại Bước 2
Ngược lại : Dừng

134

134

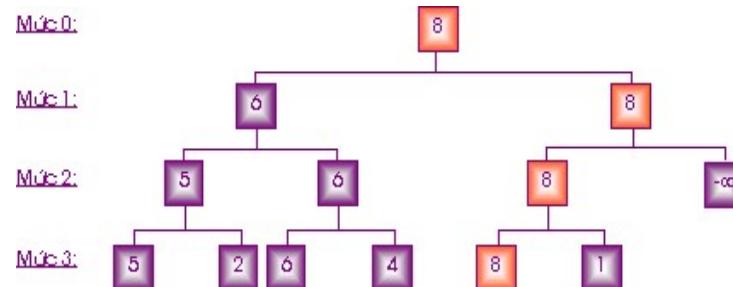
Heap sort

- ◆ Trong đó một phần tử ở mức i chính là phần tử lớn trong cặp phần tử ở mức $i+1$
- ◆ Phần tử ở mức 0 (nút gốc của cây) luôn là phần tử lớn nhất của dãy.
- ◆ Nếu loại bỏ phần tử gốc ra khỏi cây (nghĩa là đưa phần tử lớn nhất về đúng vị trí), thì việc cập nhật cây chỉ xảy ra trên những nhánh liên quan đến phần tử mới loại bỏ, còn các nhánh khác được bảo toàn, nghĩa là bước kế tiếp có thể sử dụng lại các kết quả so sánh ở bước hiện tại.

135

135

Heap sort

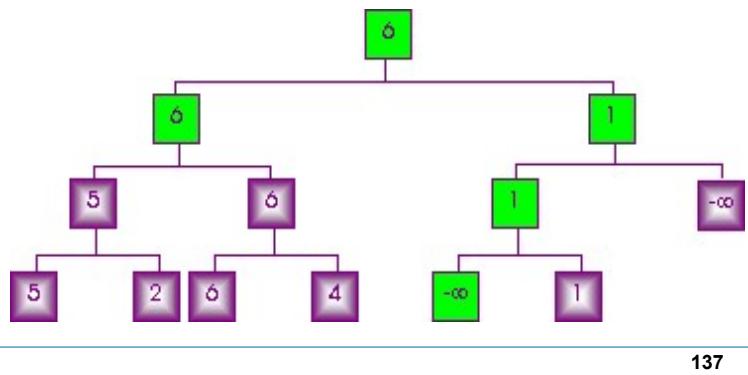


136

136

Heap sort

- ◆ Loai bỏ 8 ra khỏi cây và thế vào các chỗ trống giá trị -
◦ để tiện việc cập nhật lại cây :



137

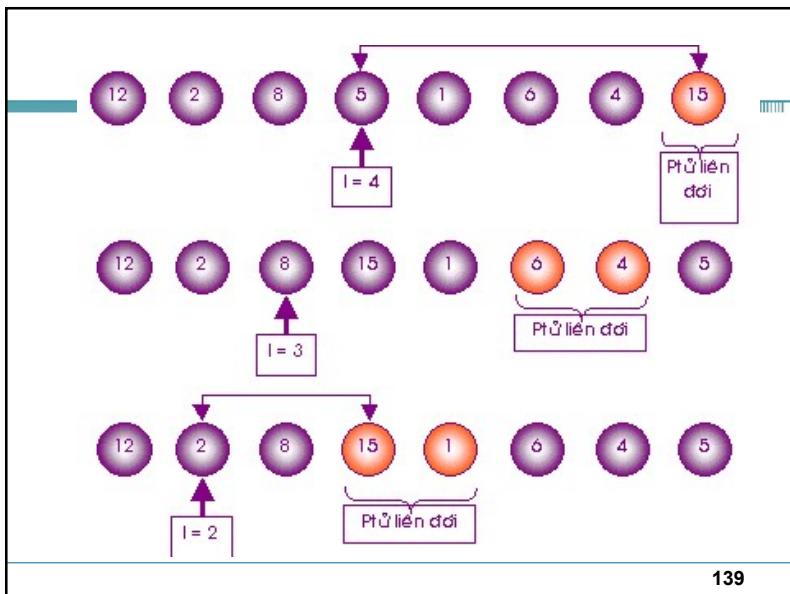
137

Heap sort – ví dụ

- ◆ Cho dãy số: 12 2 8 5 1 6 4 15
- ◆ Giai đoạn 1: hiệu chỉnh dãy ban đầu thành heap

138

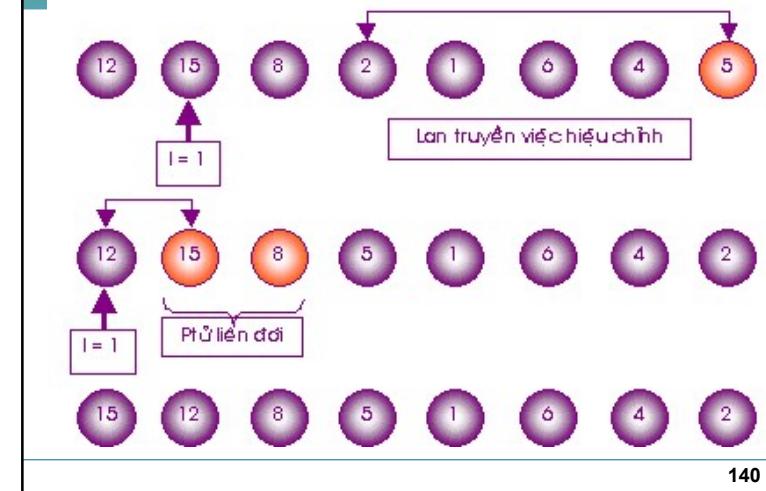
138



139

139

Heap sort – ví dụ

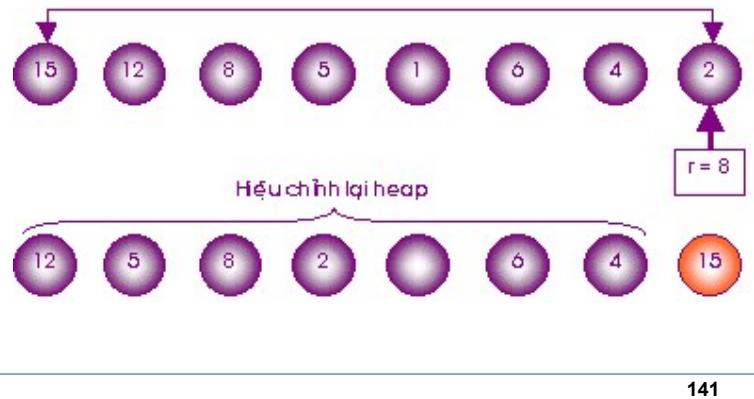


140

140

Heap sort – ví dụ

◆ Giai đoạn 2: Sắp xếp dãy số dựa trên heap :



141

Heap sort – ví dụ

Hỗ trợ chỉnh lại heap

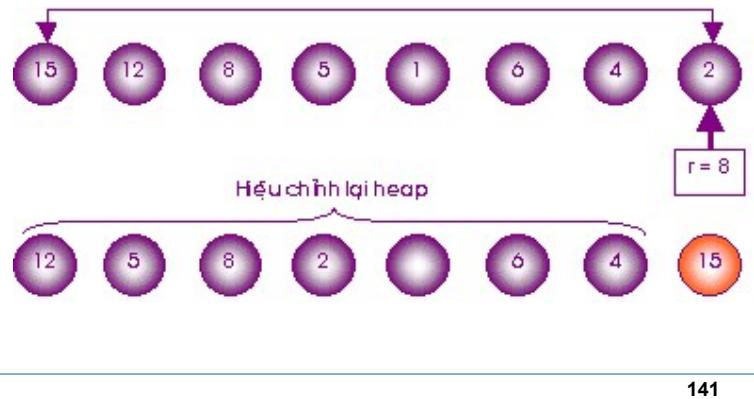


142

◆ thực hiện tương tự cho $r=5, 4, 3, 2$ ta được



144



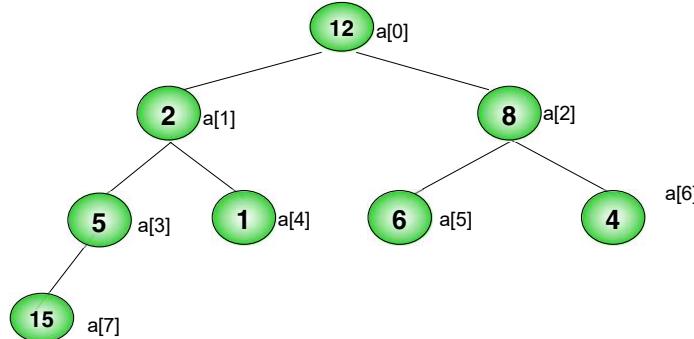
143

143

144

Heap sort

◆ Cho dãy số : 12 2 8 5 1 6 4 15
 0 1 2 3 4 5 6 7



145

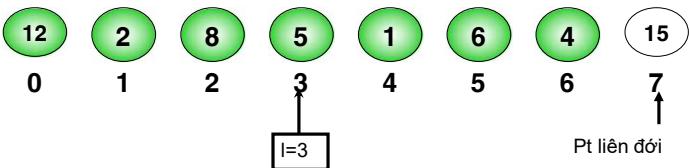
Minh họa Heap Sort

◆ **Heap:** Là một dãy các phần tử a_l, a_{l+1}, \dots, a_r thoả các quan hệ với mọi $i \in [l, r]$:

- $a_i \geq a_{2i+1}$
- $a_i \geq a_{2i+2} // (a_i, a_{2i}), (a_i, a_{2i+2})$ là các cặp phần tử liên đới

◆ Cho dãy số : 12 2 8 5 1 6 4 15

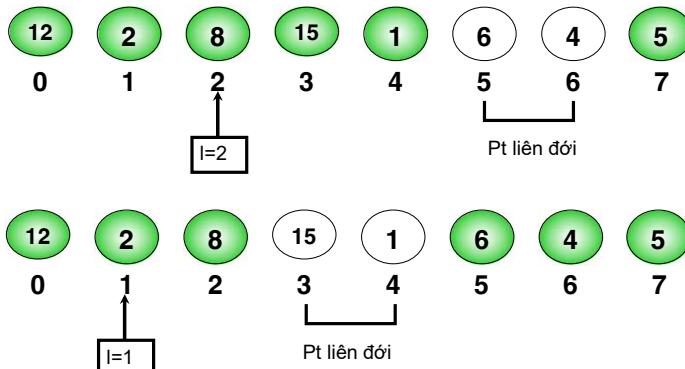
➤ Giai đoạn 1: Hiệu chỉnh dãy ban đầu thành Heap



146

145

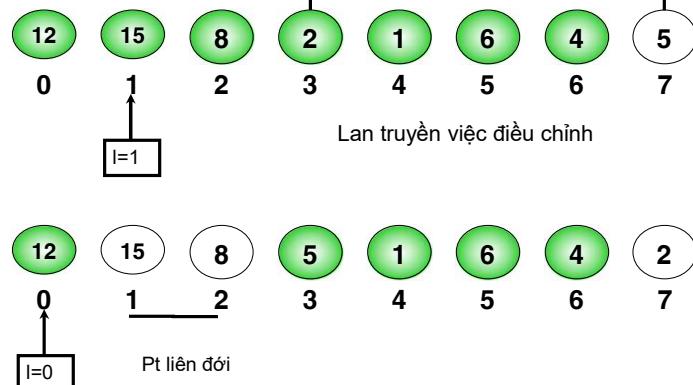
Minh họa Heap Sort



147

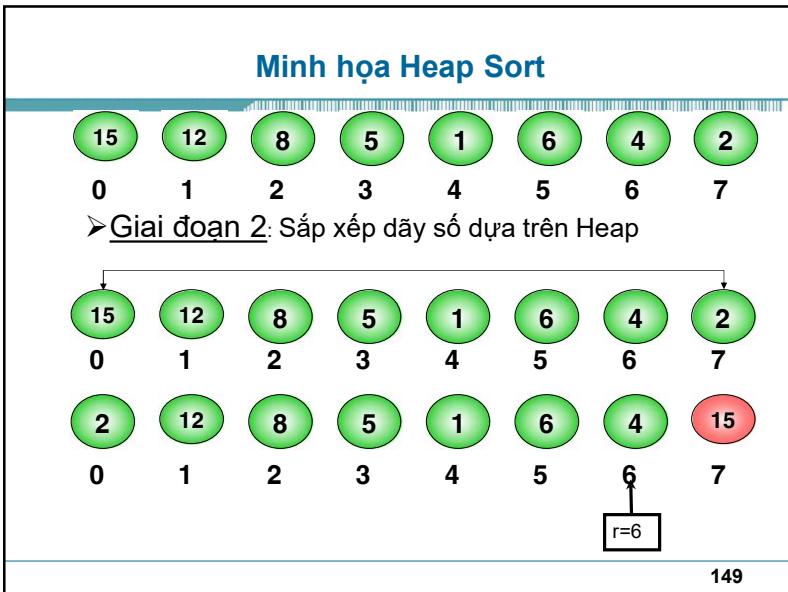
147

Minh họa Heap Sort

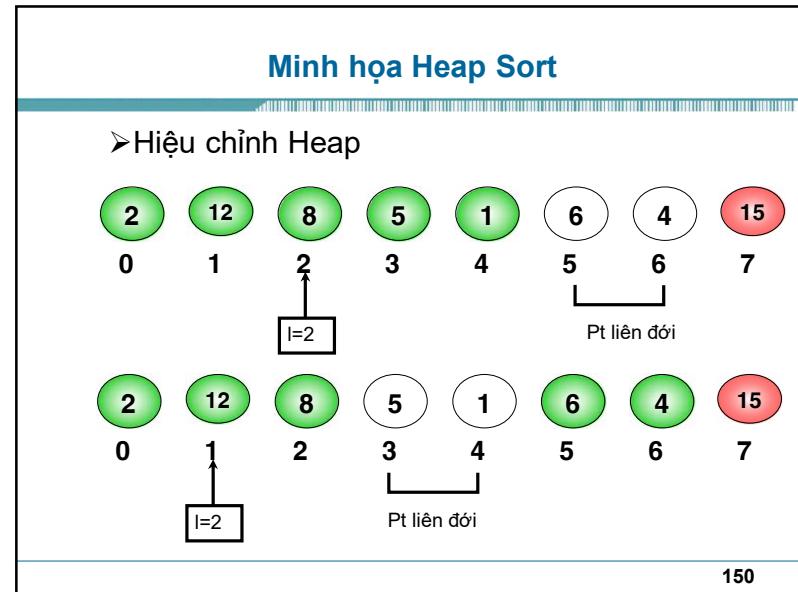


148

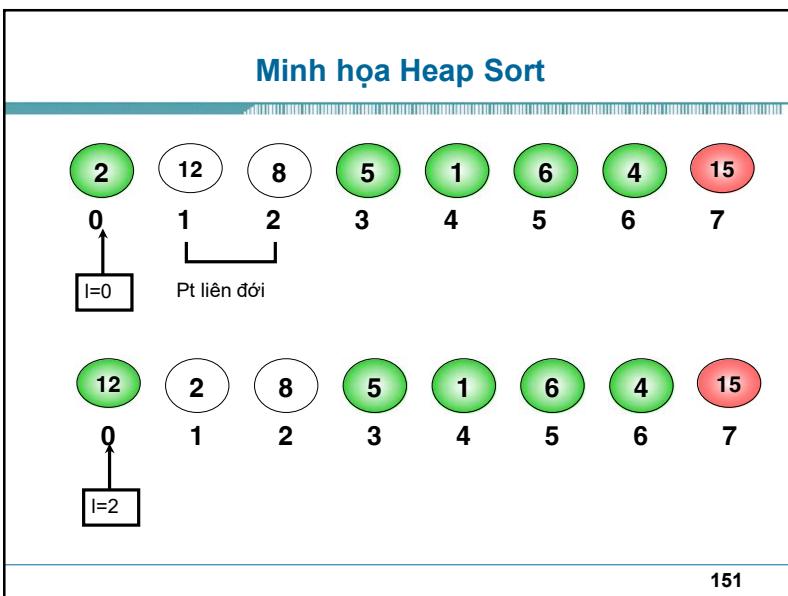
148



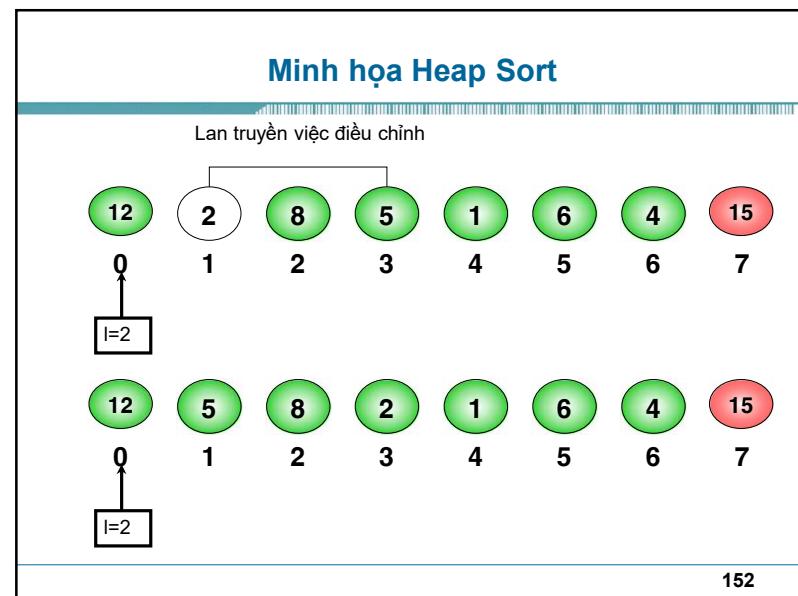
149



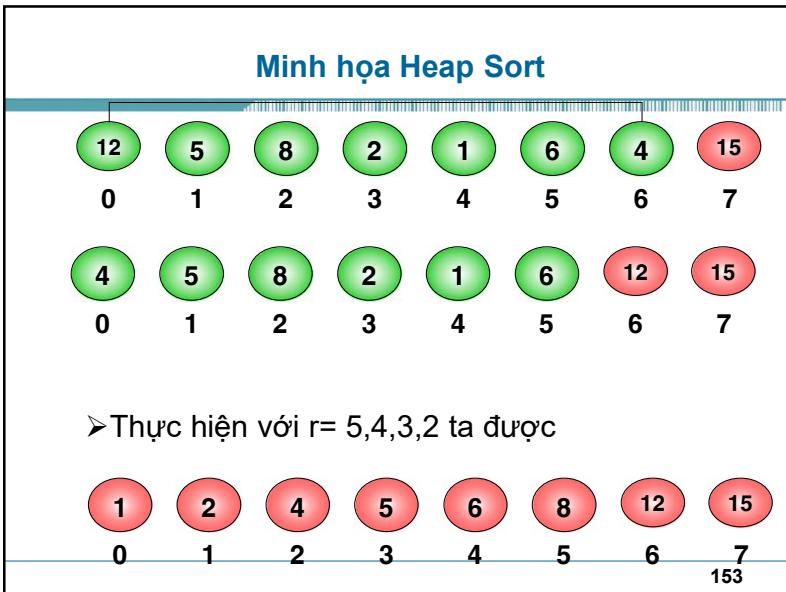
150



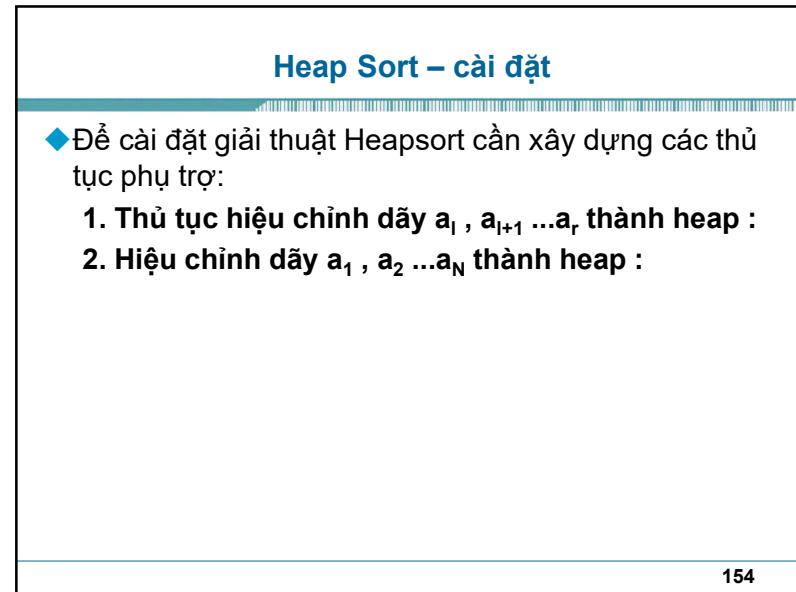
151



152



153



154

Thủ tục hiệu chỉnh dãy $a_l, a_{l+1} \dots a_r$ thành heap

- Giả sử có dãy $a_l, a_{l+1} \dots a_r$, trong đó đoạn $a_{l+1} \dots a_r$, đã là một heap.
- Ta cần xây dựng hàm hiệu chỉnh $a_l, a_{l+1} \dots a_r$ thành heap.
- Lần lượt xét quan hệ của một phần tử a_i nào đó với các phần tử liên đới của nó trong dãy là a_{2i} và a_{2i+1} , nếu vi phạm điều kiện quan hệ của heap, thì đổi chỗ a_i với phần tử liên đới thích hợp của nó.
- Lưu ý việc đổi chỗ này có thể gây phản ứng dây chuyền:
- void Shift (int a[], int l, int r)**

155

Thủ tục hiệu chỉnh dãy $a_l, a_{l+1} \dots a_r$ thành heap

```

void Shift (int a[ ], int l, int r )
{ int x,i,j;
  i = l; j = 2*i+1; // (ai , aj ), (ai , aj+1 ) là các phần tử liên đới
  x = a[i];
  while (j<=r)
  {
    if (j<r) // nếu có đủ 2 phần tử liên đới
      if (a[j]<a[j+1])// xác định phần tử liên đới lớn nhất
        j = j+1;
    if (a[j]<x) return;// thoả quan hệ liên đới, dừng.
    else
    {
      a[i] = a[j];
      i = j;// xét tiếp khả năng hiệu chỉnh lan truyền
      j = 2*i+1;
      a[i] = x;
    }
  }
}

```

156

155

156

Hiệu chỉnh dãy $a_1, a_2 \dots a_N$ thành heap

- ◆ Cho một dãy bất kỳ a_1, a_2, \dots, a_r , theo tính chất 3, ta có dãy $a_{n/2+1}, a_{n/2+2} \dots a_n$ đã là một heap. Ghép thêm phần tử $a_{n/2}$ vào bên trái heap hiện hành và hiệu chỉnh lại dãy $a_{n/2}, a_{n/2+1}, \dots, a_r$ thành heap, ..

```
void CreateHeap(int a[], int N )
{
    int l;
    l = (N-1)/2; // a[l] là phần tử ghép thêm
    while (l >= 0)
    {
        Shift(a,l,N-1);
        l = l -1;
    }
}
```

157

Heap Sort – cài đặt

- ◆ Khi đó hàm Heapsort có dạng sau :

```
void HeapSort (int a[], int N)
{
    int r;
    CreateHeap(a,N)
    r = N-1; // r là vị trí đúng cho phần tử nhỏ nhất
    while(r > 0)
    {
        Hoanvi(a[0],a[r]);
        r = r -1;
        Shift(a,0,r);
    }
}
```

158

Sắp xếp với độ dài bước giảm dần - Shell sort

- ◆ ShellSort: là một PP cải tiến của PP chèn trực tiếp.
- ◆ Ý tưởng: phân chia dãy ban đầu thành những dãy con gồm các phần tử ở cách nhau h vị trí:
- Dãy ban đầu : a_1, a_2, \dots, a_n được xem như sự xen kẽ của các dãy con sau :
 - Dãy con thứ nhất : $a_1 \ a_{h+1} \ a_{2h+1} \dots$
 - Dãy con thứ hai : $a_2 \ a_{h+2} \ a_{2h+2} \dots$
 -
 - Dãy con thứ h : $a_h \ a_{2h} \ a_{3h} \dots$

159

Shell sort

- ◆ Tiến hành sắp xếp các phần tử trong cùng dãy con sẽ làm cho các phần tử được đưa về vị trí đúng tương đối một cách nhanh chóng,
- ◆ Sau đó giảm khoảng cách h để tạo thành các dãy con mới (tạo điều kiện để so sánh một phần tử với nhiều phần tử khác trước đó không ở cùng dãy con với nó) và lại tiếp tục sắp xếp...
- ◆ Thuật toán dừng khi $h = 1$.
- ◆ Chọn h ?

160

Shell sort

Các bước tiến hành như sau:

- ◆ Bước 1: Chọn k khoảng cách $h[1], h[2], \dots, h[k]$; $i = 1$;
- ◆ Bước 2: Phân chia dãy ban đầu thành các dãy con cách nhau $h[i]$ khoảng cách. Sắp xếp từng dãy con bằng phương pháp chèn trực tiếp;
- ◆ Bước 3: $i = i+1$;
Nếu $i > k$: Dừng
Ngược lại : Lặp lại Bước 2.

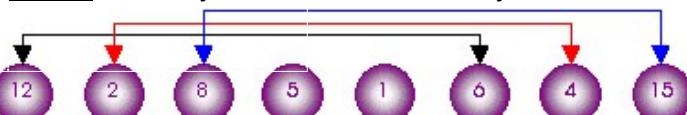
161

Shell sort – ví dụ

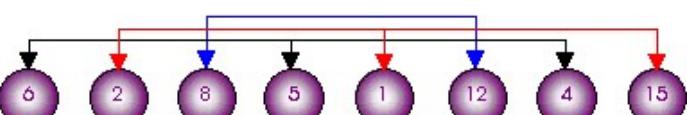
◆ Cho dãy số: 12 2 8 5 1 6 4 15

◆ Giả sử chọn các khoảng cách là 5, 3, 1

◆ $h = 5$: xem dãy ban đầu như các dãy con



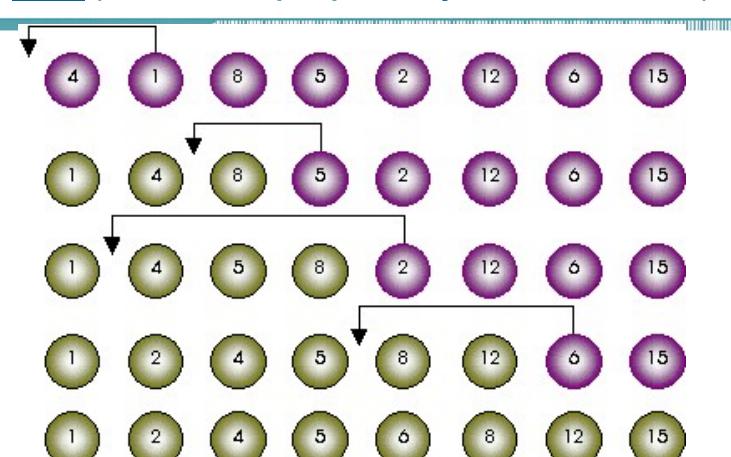
◆ $h = 3$: (sau khi đã sắp xếp các dãy con ở bước trước)



162

161

$h = 1$: (sau khi đã sắp xếp các dãy con ở bước trước)



163

Minh họa Shell Sort

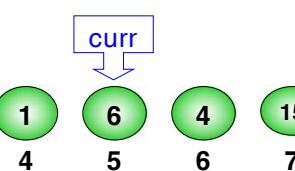
len = 5

joint



$h = (5, 3, 1); k = 3$

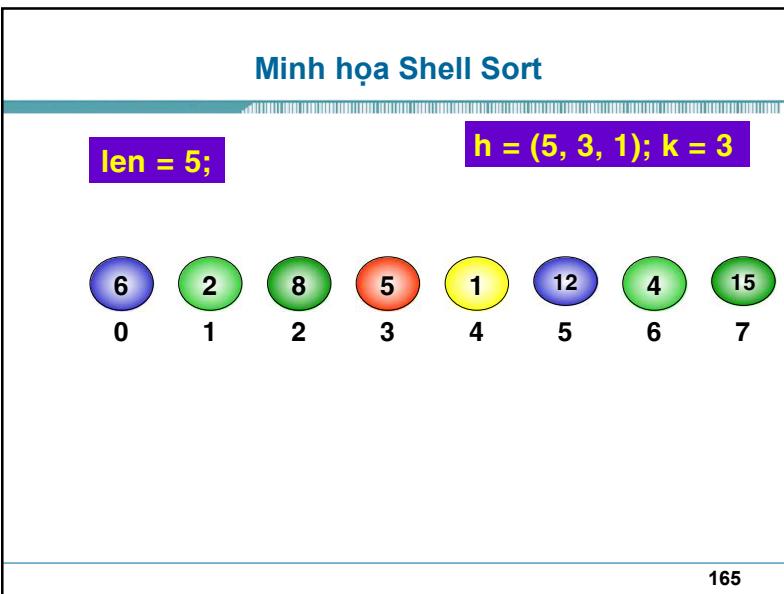
curr



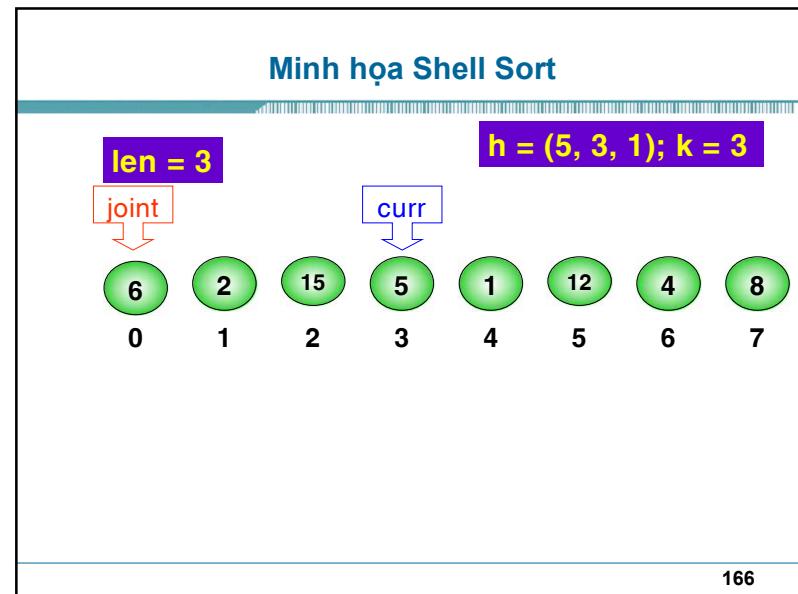
164

163

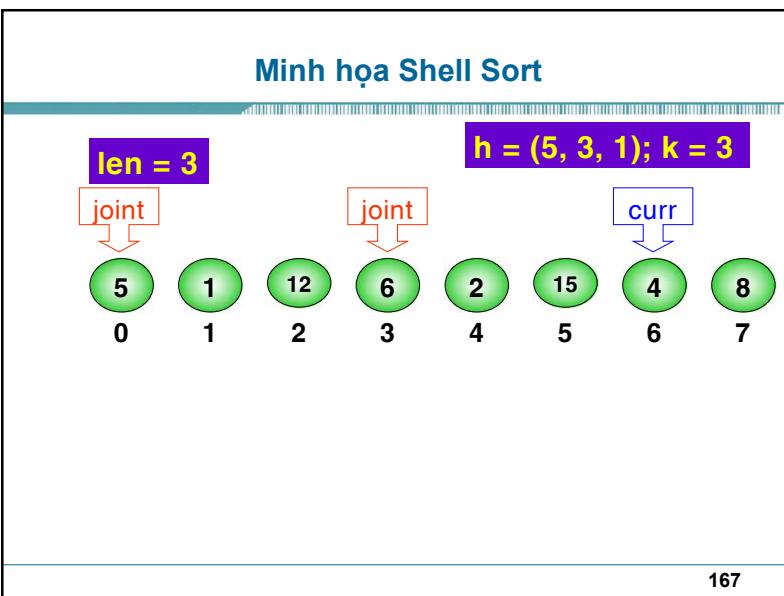
164



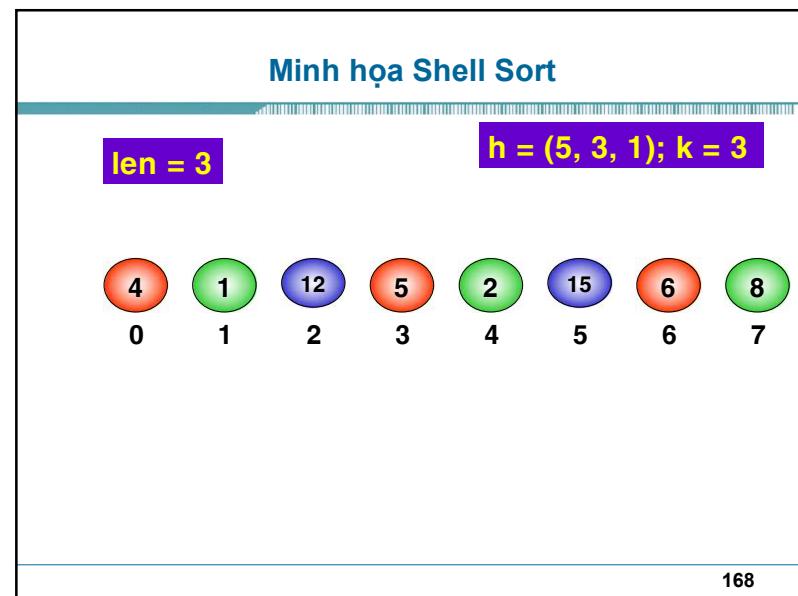
165



166

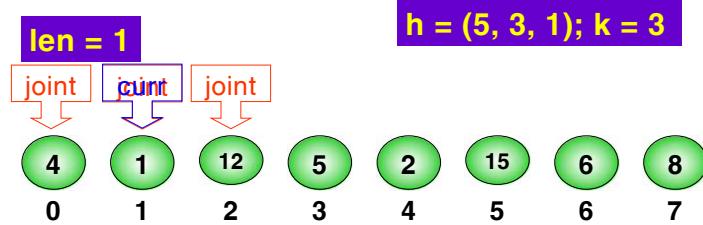


167



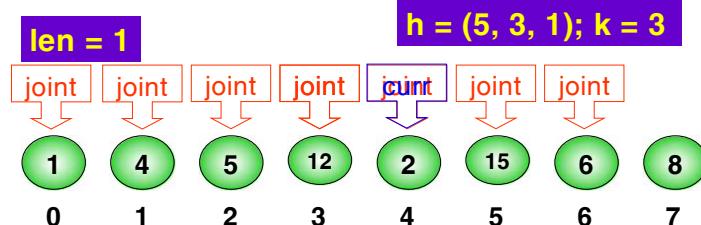
168

Minh họa Shell Sort



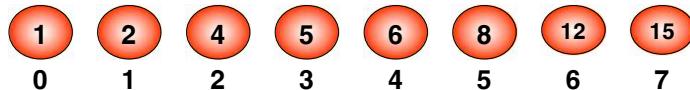
169

Minh họa Shell Sort



170

Minh họa Shell Sort



171

Cài đặt

- ◆ Giả sử đã chọn được dãy độ dài $h[1], h[2], \dots, h[k]$, thuật toán ShellSort có thể được cài đặt như sau :

```
void ShellSort(int a[], int N, int h[], int k)
```

```
{ int step,i,j;
```

```
int x,len;
```

```

for (step = 0 ; step <k; step++) (1)
{   len = h[step];
    for (i = len; i <N; i++) // (2)
    {
        x = a[i];
        j = i-len; // a[j] đứng kè trước a[i] trong cùng dãy con
    }
}

```

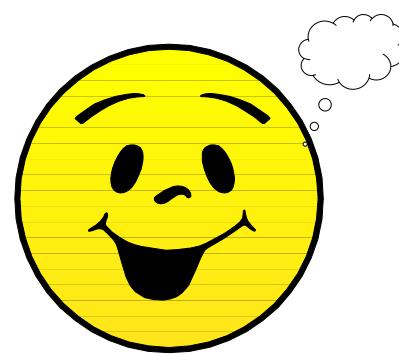
172

172

Cài đặt

```
while ((x<a[j])&&(j>=0) //sắp xếp dãy con chứa x
{
    // bằng phương pháp chèn trực tiếp
    a[j+len] = a[j];
    j = j - len;
}
a[j+len] = x;
} //for (2)
} //for (1)
} //end
```

173



Cảm ơn !

174

Chương 3. CẤU TRÚC DỮ LIỆU ĐỘNG

- ◆ Đặt vấn đề
- ◆ Kiểu dữ liệu Con trỏ
- ◆ Danh sách liên kết (link list)
- ◆ Danh sách đơn (xâu đơn)
- ◆ Tổ chức danh sách đơn theo cách cấp phát liên kết
- ◆ Một số cấu trúc dữ liệu dạng danh sách liên kết khác
 - Danh sách liên kết kép
 - Hàng đợi hai đầu (double-ended queue)
 - Danh sách liên kết có thứ tự (odered list)
 - Danh sách liên kết vòng
 - Danh sách có nhiều mồi liên kết
 - Danh sách tổng quát

1

Đặt vấn đề

- ◆ Biến không động (biến tĩnh, biến nửa tĩnh)
 - Được khai báo tường minh
 - Tồn tại trong phạm vi khai báo
 - Được cấp phát vùng nhớ trong vùng dữ liệu (Data) hoặc là Stack
 - Kích thước không thay đổi trong suốt quá trình sống

2

Đặt vấn đề

- ◆ Biến động
 - Biến không được khai báo tường minh
 - Có thể cấp phát hay giải phóng bộ nhớ khi cần
 - Vùng nhớ của biến được cấp phát trong Heap
 - Kích thước có thể thay đổi trong quá trình sống

3

Con trỏ (Pointers)

- ◆ Khai báo: **Dạng *Con trỏ**

```
char *c int *i; float *f;
typedef int *intPointer;
intPointer p;//tuong duong int* p;
hoặc int *p;
```
- ◆ Ví dụ:
 - Lập chương trình định nghĩa một số nguyên có giá trị bằng 1 và dùng một con trỏ p để chỉ số nguyên này. Sau đó in lên màn hình giá trị của số nguyên này bằng 2 cách
 - Không dùng con trỏ
 - Thông qua con trỏ

4

3

4

Con trỏ (tt)

```
#include <stdio.h>
void main()
{ int *pa, a=1;
  printf("a=%d \n", a);

  pa=&a;
  printf("a=%d \n", *pa);
}

Kết quả: a=1;
          a=1;
```

5

5

VD1



int a=1;
int *pa;
pa=&a;//pa trỏ đến a
1. Printf("%d",a);// 1
2. Printf("%d",*pa);// 1
3. Printf("%d",pa);// 00E0
4. Printf("%d",&a);// 00E0
5. Printf("%d",&pa);// 00E4

6

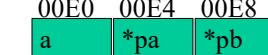
6

```
int a;
int *pa;// pa chưa có vùng nhớ, *pa=100: sai
pa=&a;// Dung, lúc này pa có vùng nhớ
pa=a;// sai

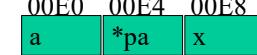
-----
a: biến a
pa: chứa địa chỉ của a
&a: địa chỉ của a
*pa: giá trị của vùng nhớ mà pa chứa//= a
&pa: địa chỉ của con trỏ pa
```

7

7



int a=1;
int *pa, *pb;
pa=&a;//pa trỏ đến a
pb=pa;
*pa=10;
*pb=100;
1. Printf("%d",a);//
2. Printf("%d",*pa);//
3. Printf("%d",*pb);//
4. Printf("%d",pa);//
5. Printf("%d",pb);//



int a=10, x=100
int *pa;
pa=&a;
Printf("%d",pa);//00E0
Printf("%d",*pa);//10
pa=&x;
Printf("%d",pa);//00E8
Printf("%d",*pa);//100

8

8

<pre> 00E0 00E4 00E8 00EC a *pa *pb x int *pa; int *pb; int a=20, x=200; pa=&x; pb=&a; int n=pa-pb;//n=(00EC-00E0)/4=3 </pre>	<pre> 00E0 00E4 00E8 00EC a *pa *pb x Int *pa; Int *pb; Int a=20; Int x=200; pa=&a;//Pa trả trên 00E0 pa=pa+3;// pa trả đến 00EC Printf("%d",*pa);// 200 </pre>
9	

9

Con trỏ - Các phép tính về con trỏ	
<pre> void main() { char *a; char c[]="Pointers"; int i; a=c;// tuong đương a=&c[0]; for(i=0; i<7; i++) printf("%c", *(a++)); printf("\n%c\n", *a); a=c; for(i=0; i<7; i++) printf("%c", *(++a)); printf("\n%c\n", *a); } </pre>	<p>Kết quả:</p> <p>C[1]=o After a+1, *a=P After a++, *a=o</p>

10

10

Con trỎ - Các phép tính về con trỎ	
<pre> void main() { char *a; char c[]="Pointers"; int i; a=c;// tuong đương a=&c[0]; for(i=0; i<7; i++) printf("%c", *(a++)); printf("\n%c\n", *a); a=c; for(i=0; i<7; i++) printf("%c", *(++a)); printf("\n%c\n", *a); } </pre>	<p>Kết quả:</p> <p>Pointer s ointers s</p>

11

Con trỎ - Các phép tính về con trỎ	
<p>◆ Lưu ý:</p> <ul style="list-style-type: none"> ▪ Ví dụ: <pre> char *cp;// 1byte int *ip;// 4 byte double *dp;// 8 byte </pre> <ul style="list-style-type: none"> ▪ Kết quả: giá trị con trỏ tăng lên bao nhiêu đơn vị? <pre> cp++ ip++ dp+=5 </pre>	

12

12

Con trỏ và mảng

◆ Khai báo:

```
int *p;  
int a[4]={1,2,3,4};  
p=&a[0];//p=a;  
// p trỏ đến a[0]  
print("a[2]=%d", *(p+2));// in ra 3  
p=p+2;  
print("a[2]=%d", *p); // in ra 3  
hay p=a; print("a[2]=%d", *(p+=2));
```

13

13

Con trỏ dùng như mảng

```
int *p={1,2,3,4};  
print("%d", *(p+2));  
char *p="con tro";  
char *p[3]={"Fortran", "Pascal", "Lisp"};
```

Khi đó:

```
p[0];  
p[1];  
p[2];  
printf("%s", p[2]);
```

14

14

Con trỏ dùng như mảng

◆ Ví dụ:

- Lập chương trình dùng con trỏ để định nghĩa mảng 3 dãy chữ "Fortral", "Pascan", "List". Sửa chúng thành "Fortran", "Pascal", "Lisp" và dùng printf để kiểm tra kết quả.

15

15

Con trỏ dùng như mảng

```
include <sddio.h>;  
void main()  
{  char *p[3]={"Fortral", "Pascan", "List"};  
   *(p[0]+6)='n';  
   *(p[1]+5)='l';  
   *(p[2]+3)='p';  
   printf("%s \n %s \n %s\n", p[0], p[1], p[2]);  
// in ra  
  Fortran  
  Pascal  
  Lisp  
}
```

16

16

Con trỏ dùng như mảng

◆ Ví dụ:

- Lập chương trình nhận từ bàn phím một số lượng từ 3 đến 10 dữ liệu số nguyên. Sau đó dùng lệnh printf để đưa chúng lên màn hình. Dùng 2 cách
 - Cách dùng **mảng**
 - Cách dùng con trỏ kết hợp với **malloc**

17

17

Con trỏ dùng như mảng

```
#define MAX 10;
void main()
{ int a[MAX], i, n;
printf("So luong du lieu"); scanf("%d", &n);
for(i=0; i<n; i++)
{ printf("du lieu %d =",i);
scanf("%d", &a[i]);
}
for(i=0; i<n; i++)
printf("a[%d] =%d \n", i,a[i]);
}
```

18

18

Con trỏ dùng như mảng

```
void main()
{ int *p, i, n;
printf("So luong du lieu"); scanf("%d", &n);
p =(int*) malloc(n*sizeof(int));
//Hoặc p =(int*) calloc(n,sizeof(int));
for(i=0; i<n; i++)
{ printf("du lieu %d =",i);
scanf("%d", p+i);
}
for(i=0; i<n; i++)
printf("a[%d] =%d \n", i,*(p+i));
free(p);
}
```

19

19

Con trỏ và cấu trúc

◆ Ví dụ:

- Định nghĩa cấu trúc có các thành phần:
 - **Name**: mảng ký tự
 - **Age**: kiểu số nguyên
- Viết chương trình nhập vào danh sách 3 người có thông tin như trên, sau đó in thông tin lên màn hình.

20

Con trỏ và cấu trúc

```
1 struct PERSON_INFO
2 {
3     char name[10];
4     int age;
5 };
6 int n;
7 PERSON_INFO list[3];
8 PERSON_INFO *p;
9 p=list;
10 for(int i=0;i<3;i++)
11 {
12     printf("Name: ");fflush(stdin); gets((p+i)->name);
13     printf("Age: ");scanf("%d",&((p+i)->age));
14 }
15 for(int i=0;i<3;i++)
16 {
17     printf("Name: %s Age: %d", (p+i)->name, (p+i)->age);
18 }
```

21

Con trỏ vạn năng

- ◆ Là con trỏ có thể chỉ bất kỳ loại dữ liệu nào (chữ, số nguyên, số thực, ...) trong chương trình
- ◆ Con trỏ vạn năng được định nghĩa như sau
void *p;

22

22

Con trỏ vạn năng

```
void main()
{ void *p;
  char c='a'; int n='1'; float r =0.5;
  p=&c;
  printf("p= %c\n", *((char*)) p);
  p=&n;
  printf("p= %d\n", *((int*)) p);
  p=&r;
  printf("p= %1.1f\n", *((float*)) p);
}
```

23

Con trỏ kép

```
void main()
{ char *p[3]={"Fortran", "Pascal", "Lisp"};
  char **pp;
  pp=p;
  printf("%s\n %s\n %s\n", *pp, *(pp+1), *(pp+2));
}
```

24

24



Chương 3. CẤU TRÚC DỮ LIỆU ĐỘNG

- ◆ Đặt vấn đề
- ◆ Kiểu dữ liệu Con trỏ
- ◆ Danh sách liên kết (link list)
- ◆ Danh sách đơn
- ◆ Tổ chức danh sách đơn theo cách cấp phát liên kết
- ◆ Một số cấu trúc dữ liệu dạng danh sách liên kết khác
 - Danh sách liên kết kép
 - Hàng đợi hai đầu (double-ended queue)
 - Danh sách liên kết có thứ tự (odered list)
 - Danh sách liên kết vòng
 - Danh sách có nhiều mối liên kết
 - Danh sách tổng quát

1

Khái niệm danh sách

◆ Khái niệm danh sách

- Mô hình toán học của danh sách là một tập hợp hữu hạn các phần tử có cùng một kiểu, tổng quát gọi là kiểu phần tử (ElementType).
- Ta biểu diễn danh sách như là một chuỗi các phần tử của nó: a_1, a_2, \dots, a_n với $n \geq 0$.
 - Nếu $n=0$ ta nói danh sách rỗng (empty list).
 - Số phần tử của DS gọi là độ dài của danh sách.
- Một tính chất quan trọng của danh sách là các phần tử của danh sách có thứ tự tuyến tính theo vị trí (position) xuất hiện của các phần tử.

2

Các phép toán trên danh sách

- ◆ **INSERT_LIST(x,p,L)**: xen phần tử x (kiểu ElementType) vào vị trí p (kiểu position) trong danh sách L.
- ◆ **LOCATE(x,L)**: thực hiện việc định vị phần tử có nội dung x đầu tiên trong danh sách L. Nếu x không có trong danh sách thì vị trí sau phần tử cuối cùng của danh sách được trả về, tức là ENDLIST(L).
- ◆ **RETRIEVE(p,L)**: lấy giá trị của phần tử ở vị trí p (kiểu position) của danh sách L.

3

Các phép toán trên danh sách

- ◆ **DELETE_LIST(p,L)**: thực hiện việc xoá phần tử ở vị trí p (kiểu position) của danh sách.
- ◆ **NEXT(p,L)**: cho kết quả là vị trí của phần tử sau phần tử p; nếu p là phần tử cuối cùng trong danh sách L thì NEXT(p,L) cho kết quả là ENDLIST(L).
- ◆ **PREVIOUS(p, L)**: cho kết quả là vị trí của phần tử đứng trước phần tử p trong danh sách.

4

Các phép toán trên danh sách

- ◆ **FIRST(L)**: cho kết quả là vị trí của phần tử đầu tiên trong danh sách.
- ◆ **EMPTY_LIST(L)**: cho kết quả TRUE nếu danh sách rỗng, ngược lại nó cho giá trị FALSE.
- ◆ **MAKENULL_LIST(L)**: khởi tạo một danh sách rỗng.

5

Các hình thức tổ chức danh sách

◆ Mối liên hệ giữa các phần tử được ngầm hiểu

- Mỗi phần tử có một chỉ số và ngầm hiểu rằng a_{i+1} nằm sau a_i . Do đó các phần tử phải **nằm cạnh nhau trong bộ nhớ**.
- **Số lượng phần tử cố định**. Không có thao tác thêm và hủy mà chỉ có thao tác dời chỗ.
- **Truy xuất ngẫu nhiên** đến từng phần tử nhanh chóng.
- **Phí bộ nhớ** do không biết trước kích thước.
- Ví dụ: **mảng một chiều**.

6

Các hình thức tổ chức danh sách

◆ Mối liên hệ giữa các phần tử rõ ràng

- Mỗi phần tử ngoài thông tin bản thân còn có thêm **liên kết** (địa chỉ) **đến phần tử kế tiếp**.
- Các phần tử **không cần phải sắp xếp cạnh nhau trong bộ nhớ**.
- Việc **truy xuất** đến một phần tử này đòi hỏi phải **thông qua một phần tử khác**.
- Tùy nhu cầu, các phần tử sẽ liên kết theo nhiều cách khác nhau tạo thành danh sách liên kết **đơn, kép, vòng**.

7

Cài đặt danh sách bằng mảng (DS đặc)

- ◆ Ta có thể cài đặt danh sách bằng mảng như sau: *dùng một mảng để lưu giữ liên tiếp các phần tử của danh sách từ vị trí đầu tiên của mảng*.

Chỉ số	0	1	...	Last-1	...	Maxlength-1
Nội dung phần tử	Phần tử thứ 1	Phần tử thứ 2	...	Phần tử cuối cùng trong danh sách	...	

8

Cài đặt danh sách bằng mảng (tt)

```
#define MaxLength 100
    //Số nguyên thích hợp để chỉ độ dài của danh sách
typedef ... ElementType; //kiểu của phần tử trong danh
    sách
typedef int Position; //kiểu vị trí của các phần tử
typedef struct {
    ElementType Elements[MaxLength];
        //mảng chứa các phần tử của danh sách
    Position Last; //giữ độ dài danh sách
} List;
```

9

Cài đặt danh sách bằng mảng (tt)

◆ Khởi tạo danh sách rỗng

- Danh sách rỗng là một danh sách không chứa bất kỳ một phần tử nào (hay độ dài danh sách bằng 0).
- Với cách khai báo trên, trường Last chỉ vị trí của phần tử cuối cùng trong danh sách và đó cũng độ dài hiện tại của danh sách.

```
void Makenull_List(List *L) void Makenull_List(List L)
{
    L->Last=0;
}
```

L.Last=0;
→DS không bị thay đổi

10

Cài đặt danh sách bằng mảng (tt)

◆ Kiểm tra danh sách rỗng

- Danh sách rỗng là một danh sách mà độ dài của nó bằng 0.

```
int IsEmpty_List(List L)
{
    return (L.Last==0);
}

int IsEmpty_List(List *L)
{
    return L->Last==0;
}
```

11

Cài đặt danh sách bằng mảng (tt)

◆ Chèn phần tử có nội dung x vào vị trí p trong danh sách sẽ có các trường hợp sau:

◆ Mảng đầy

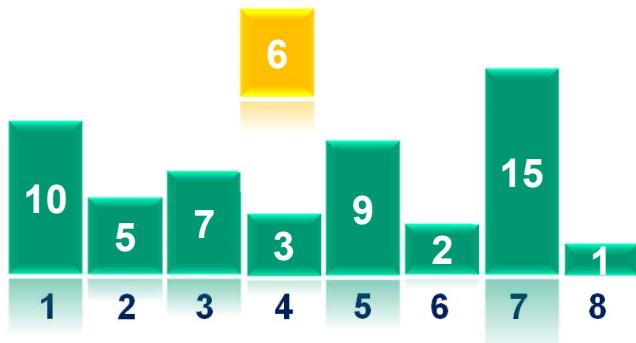
◆ Ngược lại ta tiếp tục xét:

- Nếu p không hợp lệ (p>last+1 hoặc p<1)- Lỗi
- Nếu vị trí p hợp lệ thì tiến hành chèn theo các bước:
 - Dời các phần tử từ vị trí p đến cuối DS ra sau 1 vị trí.
 - Độ dài danh sách tăng 1.
 - Đưa phần tử mới vào vị trí p.
 - Chèn phần tử x vào vị trí p.

12

Cài đặt danh sách bằng mảng (tt)

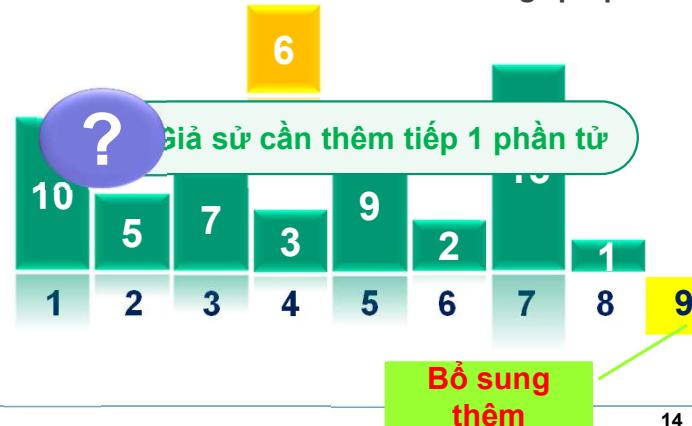
◆ Cho một mảng có N=8 phần tử



13

Cài đặt danh sách bằng mảng (tt)

* Làm sao để chèn thêm số 6 vào mảng tại vị trí 5



14

Cài đặt danh sách bằng mảng (tt)

```
void Insert_List(ElementType X, Position P, List *L)
{
    if (L->Last==MaxLength)
        printf("Danh sach day");
    else if ((P<1) || (P>L->Last+1))
        printf("Vi tri khong hop le");
    else
    {
        Position Q;
        /*Dời các phần tử từ vị trí p (chỉ số trong mảng là p-1)
         *đến cuối danh sách sang phải 1 vị trí*/
        for(Q=(L->Last-1)+1;Q>P-1;Q--)
            L->Elements[Q]=L->Elements[Q-1];
        L->Elements[P-1]=X; //Đưa x vào vị trí p
        L->Last++; //Tăng độ dài danh sách lên 1
    }
}
```

15

Cài đặt danh sách bằng mảng (tt)

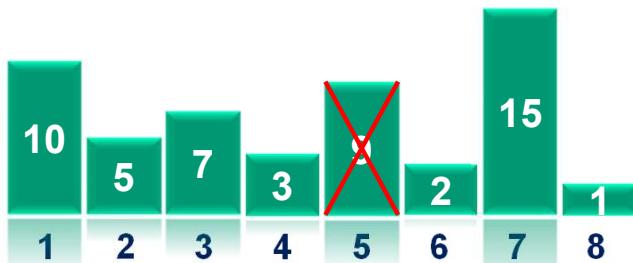
◆ Xóa phần tử ra khỏi danh sách

- Xoá một phần tử ở vị trí p ra khỏi danh sách L ta làm công việc ngược lại với xen.
- Trước tiên kiểm tra vị trí phần tử cần xóa xem có hợp lệ hay chưa? Nếu p>L.last hoặc p<1 thì đây không phải là vị trí hợp lệ.
- Nếu vị trí hợp lệ thì phải dời các phần tử từ vị trí p+1 đến cuối DS lên trước một vị trí và độ dài danh sách giảm đi 1 phần tử.

16

Cài đặt danh sách bằng mảng (tt)

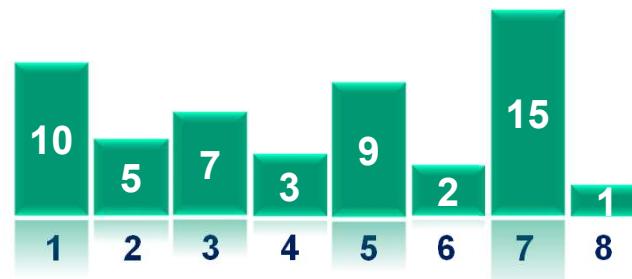
◆ Làm sao để xóa phần tử 9 ?



17

Cài đặt danh sách bằng mảng (tt)

◆ Làm sao để xóa phần tử 9 ?



18

Cài đặt danh sách bằng mảng (tt)

```
void Delete_List(Position P, List *L)
{ if ((P<1) || (P>L->Last))
    printf("Vi tri khong hop le");
else if (EmptyList(*L))
    printf("Danh sach rong!");
else {
    Position Q;
    /*Dời các phần tử từ vị trí p+1 (chỉ số trong mảng là p)
       đến cuối danh sách sang trái 1 vị trí*/
    for(Q=P-1;Q<L->Last-1;Q++)
        L->Elements[Q]=L->Elements[Q+1];
    L->Last--;
}
```

19

Cài đặt danh sách bằng mảng (tt)

◆ Định vị một phần tử trong danh sách

- Để định vị vị trí phần tử đầu tiên có nội dung x trong danh sách L, ta tiến hành dò tìm từ đầu danh sách.
 - Nếu tìm thấy x thì vị trí của phần tử tìm thấy được trả về.
 - Nếu không tìm thấy thì hàm trả về vị trí sau vị trí của phần tử cuối cùng trong danh sách, tức là ENDLIST(L) (ENDLIST(L)= L.Last+1).
 - Trong t/hợp có nhiều phần tử cùng giá trị x trong danh sách thì vị trí của phần tử được tìm thấy đầu tiên sẽ được trả về.

20

Cài đặt danh sách bằng mảng (tt)

```
Position Locate(ElementType X, List L)
{
    Position P;
    int Found = 0;
    P = First(L); //vị trí phần tử đầu tiên
    /*trong khi chưa tìm thấy và chưa kết thúc
     danh sách thì xét phần tử kế tiếp */
    while ((P != EndList(L)) && (Found == 0))
        if (Retrieve(P,L) == X) Found = 1;
        else P = Next(P, L);
    return P;
}
```

21

21

Cài đặt danh sách bằng mảng (tt)

```
void main()
{
    List L;
    ElementType X;
    Position P;
    Makenull_List(&L); //Khởi tạo danh sách rỗng
    Read_List(&L);
    printf("\nDanh sach vua nhap: ");
    Print_List(L); // In danh sach len man hinh
}
```

23

23

Cài đặt danh sách bằng mảng (tt)

◆ Lưu ý : Các phép toán sau phải thiết kế trước Locate

```
Position First(List L) ElementType Retrieve(Position P, List L)
{
    {
        return L.Elements[P];
    }
    return 1;
}
Position EndList(List L)
Position Next(Position P, List L)
{
    {
        return P+1;
    }
    return L.Last+1;
}
```

22

22

Cài đặt danh sách bằng mảng (tt)

```
printf("Phan tu can them: ");scanf("%d",&X);
printf("Vi tri can them: "); scanf("%d",&P);
Insert_List(X,P,&L);
printf("DS sau khi themla: "); Print_List(L);
printf("Phan tu can xoa: ");scanf("%d",&X);
P=Locate(X,L);
Delete_List(P,&L);
printf("Danh sach sau khi xoa %d la: ",X); Print_List(L);
}
Position Locate(ElementType X, List L)
{
    for(int i=0;i<L.Last-1;i++)
        if(L.Elements[i]==X)
            {
                return i;
            }
    return -1;
}
void Print_List(List L)
{
    for(int i=0;i<L.Last-1;i++)
        printf("%d", L.Elements[i]);
}
```

24

24



Chương 3. CẤU TRÚC DỮ LIỆU ĐỘNG

- ◆ Đặt vấn đề
- ◆ Kiểu dữ liệu Con trỏ
- ◆ **Danh sách liên kết (link list)**
- ◆ Danh sách đơn (xâu đơn)
- ◆ Tổ chức danh sách đơn theo cách cấp phát liên kết
- ◆ Một số cấu trúc dữ liệu dạng danh sách liên kết khác
 - Danh sách liên kết kép
 - Hàng đợi hai đầu (double-ended queue)
 - Danh sách liên kết có thứ tự (ordered list)
 - Danh sách liên kết vòng
 - Danh sách có nhiều mối liên kết
 - Danh sách tổng quát

1

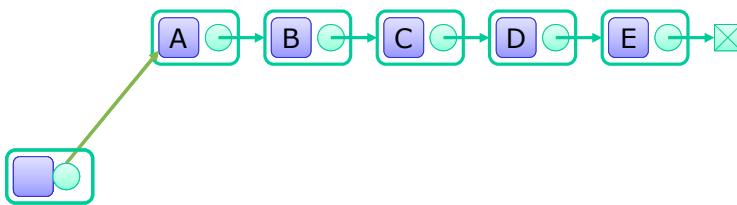
DANH SÁCH LIÊN KẾT

- ◆ Danh sách liên kết?
 - Bao gồm các thành phần có cấu trúc giống nhau.
 - Mỗi cấu trúc gồm: thành phần dữ liệu và con trỏ chỉ tới phần tử kế tiếp trong danh sách – Gọi là con trỏ *next*

2

Các loại danh sách liên kết

◆ Danh sách liên kết đơn



Header: Thường ký hiệu là L

3

Các loại danh sách liên kết

◆ Danh sách liên kết kép (Doubly Linked List)



4

Các loại danh sách liên kết

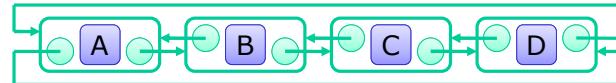
◆ Danh sách liên kết đơn vòng (Circular Linked List)



5

Các loại danh sách liên kết

◆ Danh sách liên kết kép vòng (Circular Doubly Linked List)



6

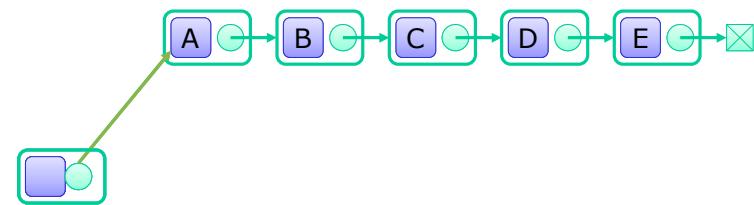
Danh sách liên kết

◆ Nhận xét

- Số nút không cố định, thay đổi tùy nhu cầu nên đây là cấu trúc động.
- Thích hợp thực hiện các thao tác **chèn** và **hủy** vì không cần phải dời nút mà **chỉ cần sửa các liên kết cho phù hợp**. Thời gian thực hiện không phụ thuộc vào số nút danh sách.
- Tốn bộ nhớ chứa con trỏ liên kết pNext.
- **Truy xuất tuần tự** nên mất thời gian.

7

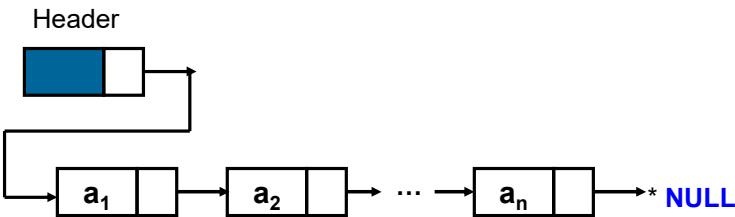
Danh sách liên kết đơn



Header : L

8

DANH SÁCH LIÊN KẾT ĐƠN (tt)



9

DANH SÁCH LIÊN KẾT ĐƠN

Khai báo DANH SÁCH QUẢN LÝ SỐ NGUYÊN

typedef int ElementType; //kiểu của phần tử trong danh sách
struct Node

```

{
    ElementType Element; //Chứa nội dung của phần tử
    Node      *Next; //con trỏ chỉ đến phần tử kế tiếp
};
  
```

```

typedef Node *      PtrToNode;
typedef PtrToNode   Position; //kiểu vị trí
typedef PtrToNode   List;     //Danh sách
  
```

10

Danh sách quản lý sinh viên

```

struct SinhVien
{
    char Ten[32];
    int Tuoi;
    float diemtb;
};

typedef SinhVien ElementType; //kiểu của phần tử trong danh sách
struct Node
{
    ElementType Element; //Chứa nội dung của phần tử
    Node      *Next; //con trỏ chỉ đến phần tử kế tiếp
};

typedef Node *      PtrToNode;
typedef PtrToNode   Position; //kiểu vị trí
typedef PtrToNode   List;     //Danh sách
  
```

11

DANH SÁCH LIÊN KẾT ĐƠN (tt)

- ◆ Để quản lý danh sách ta chỉ cần một biến giữ địa chỉ ô chứa phần tử đầu tiên của danh sách. Biến này gọi là *chỉ điểm đầu danh sách (Header)* .
- ◆ Header là một ô đặc biệt:
 - Trường dữ liệu của ô này là **rỗng**,
 - Trường con trỏ Next trỏ tới ô chứa phần tử đầu tiên thật sự của danh sách. Nếu danh sách rỗng thì Header->next trỏ tới **NULL**.
- ◆ Cần phân biệt rõ giá trị của một phần tử và vị trí (position) của nó trong cấu trúc trên.

12

DANH SÁCH LIÊN KẾT ĐƠN (tt)

- Tạo danh sách rỗng

```
void Makenull_List(List &L)
{
    L = new Node;
    L->Next = NULL;
}

- Kiểm tra một danh sách rỗng
```

```
int IsEmpty_List( List L )
```

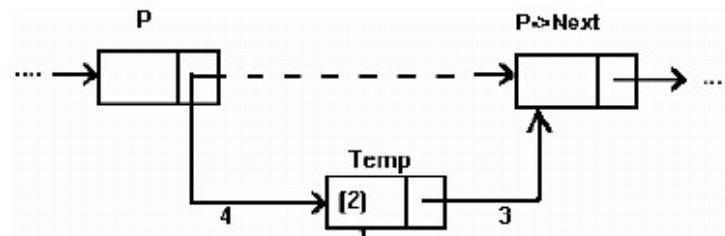
```
{
    return L->Next == NULL;
}
```

13

14

DANH SÁCH LIÊN KẾT ĐƠN (tt)

◆ Chèn một phần tử vào danh sách :



14

DANH SÁCH LIÊN KẾT ĐƠN (tt)

◆ Chèn một phần tử vào danh sách

- Chèn vào đầu danh sách
- Chèn vào cuối danh sách
- Chèn vào danh sách sau một phần tử q

15

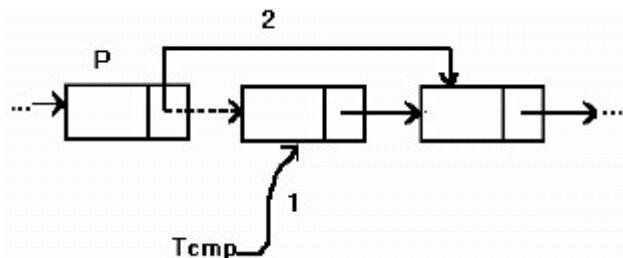
16

DANH SÁCH LIÊN KẾT ĐƠN (tt)

```
void Insert_List( ElementType X,Position P, List L )
{
    Position TmpCell;
    TmpCell = new Node;
    if( TmpCell == NULL )
        printf( "Out of space!!!" );// Không khởi tạo được d.sách
    else
    {
        TmpCell->Element = X;
        TmpCell->Next = P->Next;
        P->Next = TmpCell;
    }
}
```

DANH SÁCH LIÊN KẾT ĐƠN (tt)

Xóa một phần tử khỏi danh sách :



17

DANH SÁCH LIÊN KẾT ĐƠN (tt)

Xóa phần tử có giá trị X trong danh sách

void Delete_List(ElementType X, List L)

```
{  
    Position P, TmpCell;  
    P = FindPrevious( X, L );  
    if( !IsLast( P, L ) )  
    {  
        TmpCell = P->Next;  
        P->Next = TmpCell->Next;  
        free( TmpCell );// xoa vung nho  
    }  
}
```

18

Tìm vị trí trước phần tử có giá trị X

```
Position FindPrevious(ElementType X, List L)  
{  
    if(IsEmpty_List())  
        return NULL;  
    Position P=L;  
    while (P->Next!=NULL)  
    {  
        if(P->Next->Element==X) return P;  
        else P=P->Next;  
    }  
    return NULL;  
}  
  
int IsLast(Position P, List L)  
{  
    return ( P->next == NULL );  
}
```

19

DANH SÁCH LIÊN KẾT ĐƠN (tt)

◆ Định vị một phần tử trong danh sách liên kết

Position Locate(ElementType X, List L)

```
{  
    Position P;  
    P = L->Next;//node đầu danh sách  
    while( P != NULL && P->Element != X )  
        P = P->Next;  
    return P;  
}
```

20

DANH SÁCH LIÊN KẾT ĐƠN (tt)

◆ Xác định nội dung phần tử:

```
ElementType Retrieve( Position P )
{
    return P->Element;
}
```

21

Hàm tính tổng giá trị của các phần tử trong DSLK số nguyên

```
int TinhTong(List L)
{
    int Tong=0;
    Position P=L->Next;
    while (P!=NULL)
    {
        Tong=Tong+P->Element;
        P=P->Next;
    }
    return Tong;
}

for(int Tong=0, Position P=L->Next; P!=NULL; P=P->Next)
    Tong=Tong+P->Element;
return Tong;
```

23

In danh sách

In danh sách quản lý số nguyên

Cài đặt = con trỏ

```
void Print_List(List L)
{
    Position P=L->Next;
    while (P!=NULL)
    {
        printf("%d ", P->Element);
        P=P->Next;
    }
}
```

In danh sách quản lý sinh viên

Cài đặt = con trỏ

```
void Print_List(List L)
{
    Position P=L->Next;
    while (P!=NULL)
    {
        printf("Ten :%s ", P->Element.Ten);
        printf("Tuoi :%d ", P->Element.Tuoi);
        printf("Diem :%f ", P->Element.DiemTB);
        P=P->Next;
    }
}
```

22

22

Hàm đếm số phần tử trong DSLK số nguyên

```
int Dem(List L)
{
    int d=0;
    Position P=L->Next;// Bắt đầu tại nút đầu tiên của DSLK
    while (P!=NULL) // Kiểm tra nút có khác NULL không
    {
        d++;
        P=P->Next; // Chuyển sang nút kế tiếp
    }
    return d;
}
```

24

24

Hàm in các phần tử chẵn trong DSLK số nguyên

```
void InSoChan(List L)
{
    Position P=L->Next;
    while (P!=NULL)
    {
        if(P->Element%2==0) // % là ký hiệu chia lấy dư
            printf("%d ", P->Element);

        P=P->Next;
    }
}
```

25

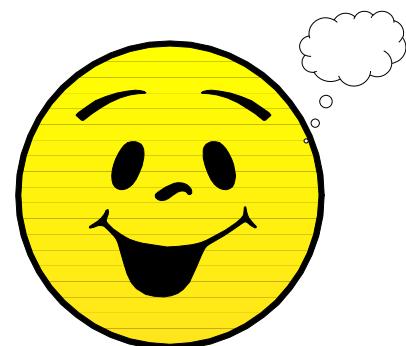
25

DANH SÁCH LIÊN KẾT ĐƠN (tt)

void Makenull_List (List &L);	x
int IsEmpty_List (List L);	x
int IsLast (Position P);	x
Position Locate (ElementType X, List L);	x
void Delete_List (ElementType X, List L);	x
Position FindPrevious (ElementType X, List L);	x
void Insert_List (ElementType X, Position P, List L);	x
void Delete_All_List (List L);	
Position Header (List L);	
Position First (List L);	
Position Advance (Position P);	
ElementType Retrieve (Position P);	x
void Read_List (List L);	
void Write_List (List L);	x

26

26



Cảm ơn !

27

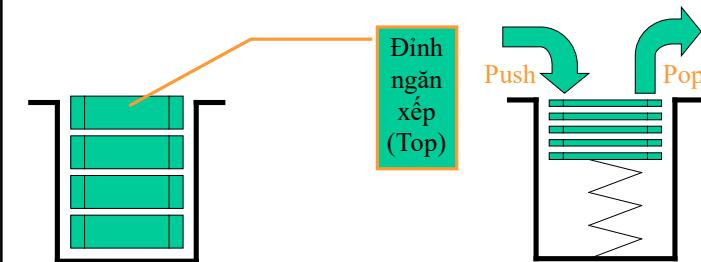
NGĂN XẾP (STACK)

- ◆ Ngăn xếp (Stack): là một danh sách mà việc thêm vào hoặc loại bỏ một phần tử chỉ thực hiện tại một đầu của danh sách, đầu này gọi là đỉnh (TOP) của ngăn xếp.
- ◆ Stack là một cấu trúc có tính chất “vào sau - ra trước” hay “vào trước – ra sau” (**LIFO** (last in - first out) hay **FILO** (first in – last out)).

1

1

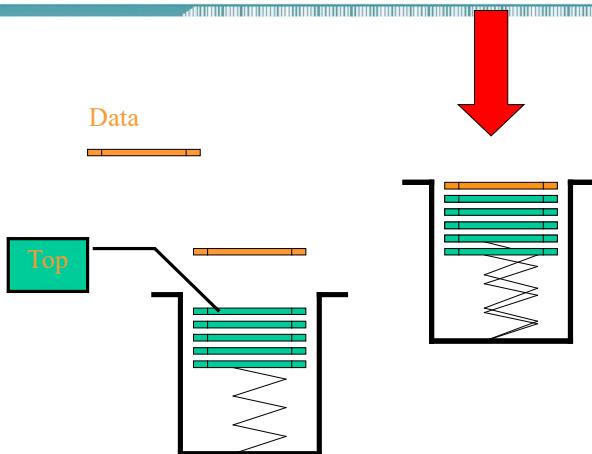
NGĂN XẾP (STACK)



2

2

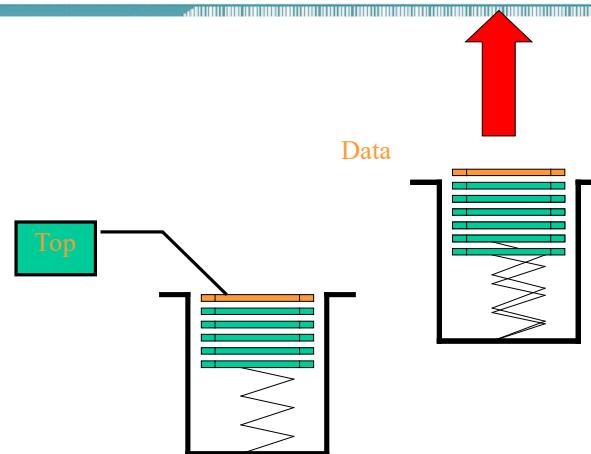
Minh họa thao tác PUSH



3

3

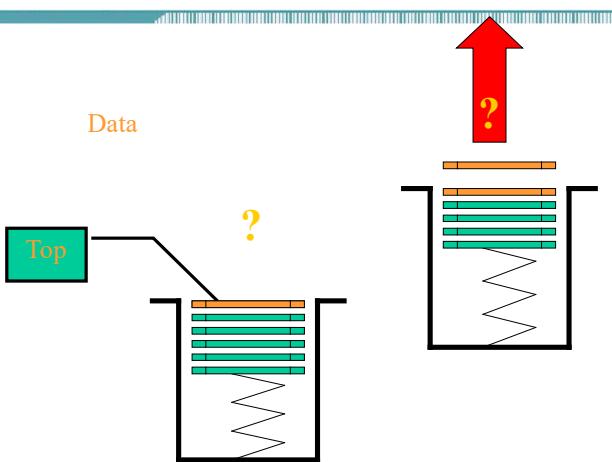
Minh họa thao tác POP



4

4

Minh họa thao tác TOP



5

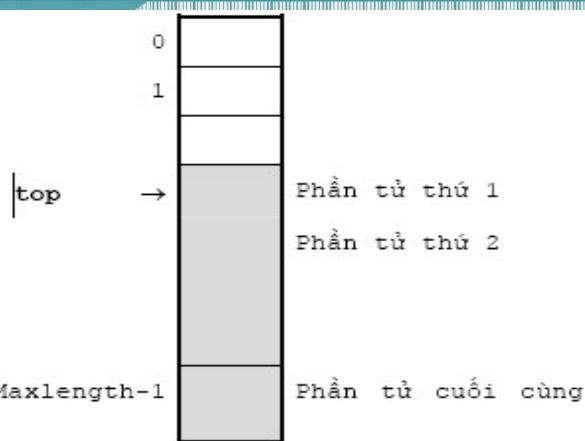
NGĂN XẾP (STACK)

◆ Các phép toán trên ngăn xếp

- **MAKENULL_STACK(S)**: tạo một ngăn xếp rỗng.
- **TOP(S)** hàm trả về phần tử tại đỉnh ngăn xếp.
- **POP(S)** xoá một phần tử tại đỉnh ngăn xếp.
- **PUSH(x,S)** thêm một phần tử x vào đầu ngăn xếp.
- **EMPTY_STACK(S)** kiểm tra ngăn xếp

6

Cài đặt ngăn xếp bằng mảng



7

Cài đặt ngăn xếp bằng mảng

◆ Khai báo ngăn xếp

```
#define MaxLength ... //độ dài của mảng  
typedef ... ElementType; //kiểu các phần tử trong ngăn xếp  
typedef struct  
{  
    ElementType Elements[MaxLength];  
    //Lưu nội dung của các phần tử  
    int Top; //giữ vị trí đỉnh ngăn xếp  
} Stack;
```

8

Cài đặt ngăn xếp bằng mảng

◆ Tạo ngăn xếp rỗng

- Ngăn xếp rỗng là ngăn xếp không chứa bất kỳ một phần tử nào, do đó đỉnh của ngăn xếp không được phép chỉ đến bất kỳ vị trí nào trong mảng.

```
void Makenull_Stack(Stack *S)
{
    S->Top=MaxLength;
}
```

9

Cài đặt ngăn xếp bằng mảng

◆ Kiểm tra ngăn xếp rỗng

```
int IsEmpty_Stack(Stack S)
{
    return S.Top==MaxLength;
}
```

◆ Kiểm tra ngăn xếp đầy

```
int IsFull_Stack(Stack S)
{
    return S.Top==0;
}
```

10

Cài đặt ngăn xếp bằng mảng

◆ Lấy nội dung phần tử tại đỉnh của ngăn xếp

```
ElementType Top(Stack S)
{
    if (!IsEmpty_Stack(S))
        return S.Elements[S.Top];
    else printf("Loi! Ngan xep rong");
}
```

11

Cài đặt ngăn xếp bằng mảng

- ◆ Chú ý Nếu ElementType không thể là kiểu kết quả trả về của một hàm thì ta có thể viết Hàm Top như sau:

```
void Top2(Stack S, ElementType *X)
{
    //Trong đó x sẽ lưu trữ nội dung phần tử
    //tại đỉnh của ngăn xếp
    if (!IsEmpty_Stack(S))
        *X = S.Elements[S.Top];
    else printf("Loi: Ngan xep rong ");
}
```

12

```

void Top3(Stack S, Elementtype &X)
{
    //Trong đó x sẽ lưu trữ nội dung phần tử
    //tại đỉnh của ngăn xếp
    if (!IsEmpty_Stack(S))
        X = S.Elements[S.Top];
    else printf("Loi: Ngan xep rong ");
}

```

13

Tham chiếu

```

int main()
{
    int a =100;
    int b=200;
    HoanVi(a,b);
    printf("%d %d",a,b);
}
Void HoanVi(int a, int b)
{
    int t=a;
    a=b;
    b=t;
}
sai

```

```

int main()
{
    int a =100;
    int b=200;
    HoanVi(&a, &b);
    printf("%d %d",a,b);
}

void HoanVi(int *a, int *b)
{
    int t=*a;
    *a=*b;
    *b=t;
}
đúng

```

14

Tham chiếu – C

```

int main()
{
    int a =100;
    int b=200;
    HoanVi(&a, &b);
    printf("%d %d",a,b);
}

void HoanVi(int *a, int *b)
{
    int t=*a; *a=*b;
    *b=t;
}

```

Tham chiếu – C++

```

int main()
{
    int a =100;
    int b=200;
    HoanVi(a, b);
    printf("%d %d",a,b);
}

void HoanVi(int &a, int &b)
{
    int t=a; a=b; b=t;
}

```

15

void Init(**int** &a);

```

int main()
{
    int x=0;
    Init(x);
    printf("%d", x);// 200
    return 0;
}

void Init(int &a)
{
    a=200;
}

```

16

Cài đặt ngăn xếp bằng mảng

◆ Xóa phần tử ra khỏi ngăn xếp

```
void Pop(Stack *S)
{
    if (!IsEmpty_Stack(*S))
        S->Top=S->Top+1;
    else printf("Loi! Ngan xep rong!");
}
```

17

Cài đặt ngăn xếp bằng mảng

◆ Thêm phần tử vào ngăn xếp

```
void Push(ElementType X, Stack *S)
{
    if (IsFull_Stack(*S))
        printf("Loi! Ngan xep day!");
    else{
        S->Top=S->Top-1;
        S->Elements[S->Top]=X;
    }
}
Void Print_Stack(Stack S)
{
    for(int i=S.Top;i<MaxLenght-1;i++)
        printf("%d", S.Elements[i]);
}
```

18

Cài đặt ngăn xếp bằng con trỏ

◆ Khai báo ngăn xếp

```
typedef ... ElementType;
struct Node
{
    ElementType Element;
    Node  *Next;
};
typedef struct Node *PtrToNode;
typedef PtrToNode Position;
typedef PtrToNode Stack;
```

19

Cài đặt ngăn xếp bằng con trỏ

◆ Tạo ngăn xếp rỗng

```
void Makenull_Stack( Stack &S )
{
    S = new Node;
    S->Next = NULL;
}
◆ Kiểm tra ngăn xếp rỗng
int IsEmpty_Stack( Stack S )
{
    return S->Next == NULL;
}
```

20

19

20

Cài đặt ngăn xếp bằng con trỏ

- ◆ **Lấy nội dung phần tử tại đỉnh của ngăn xếp**

```
ElementType Top( Stack S )
{
    return S->Next->Element;
}
```

21

Cài đặt ngăn xếp bằng con trỏ

- ◆ **xóa phần tử ra khỏi ngăn xếp**

```
void Pop(Stack S )
{
    Position P, TmpCell;
    P = Header(S);
    if( P->Next!=NULL )
    {
        TmpCell = P->Next;
        P->Next = TmpCell->Next;
        free( TmpCell );
    }
}
```

22

Cài đặt ngăn xếp bằng con trỏ

- ◆ **Thêm phần tử vào ngăn xếp**

```
void Push( ElementType X, Stack S )
{
    Position TmpCell, P;
    P = Header(S);
    TmpCell = new Node;
    if( TmpCell == NULL )
        printf( "Out of space!!!" );
    else{
        TmpCell->Element = X;
        TmpCell->Next = P->Next;
        P->Next = TmpCell;
    }
}
```

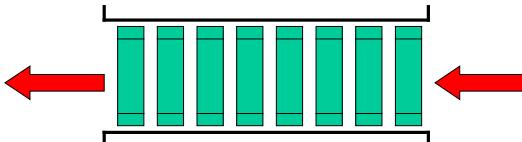
Position Header (stack S)
{
 return S;
}

23

HÀNG ĐỢI (QUEUE)

- ◆ Hàng đợi (queue): là một danh sách đặc biệt mà phép thêm vào chỉ thực hiện tại một đầu của danh sách, gọi là cuối hàng (REAR), còn phép loại bỏ thì thực hiện ở đầu kia của danh sách, gọi là đầu hàng (FRONT).

- ◆ Queue còn được gọi là cấu trúc **FIFO** (first in - first out) hay "vào trước - ra trước".



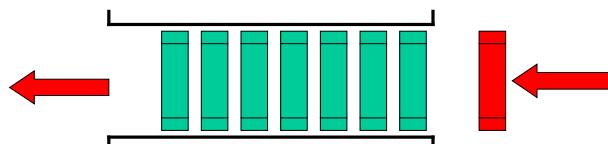
24

23

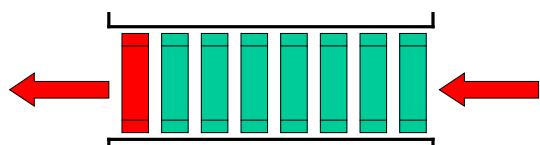
24

HÀNG ĐỢI (QUEUE)

- ◆ Minh họa thao tác EnQueue



- ◆ Minh họa thao tác DeQueue



25

Cài đặt hàng bằng mảng

- ◆ Các phép toán cơ bản trên hàng

- **Makenull_Queue(Q)** khởi tạo một hàng rỗng.
- **Front(Q)** hàm trả về phần tử đầu tiên của hàng Q.
- **EndQueue(x,Q)** thêm phần tử x vào cuối hàng Q.
- **Delete_Queue(Q)** xoá phần tử tại đầu của hàng Q.
- **IsEmpty_Queue(Q)** hàm kiểm tra hàng rỗng.
- **IsFull_Queue(Q)** hàm kiểm tra hàng đầy.

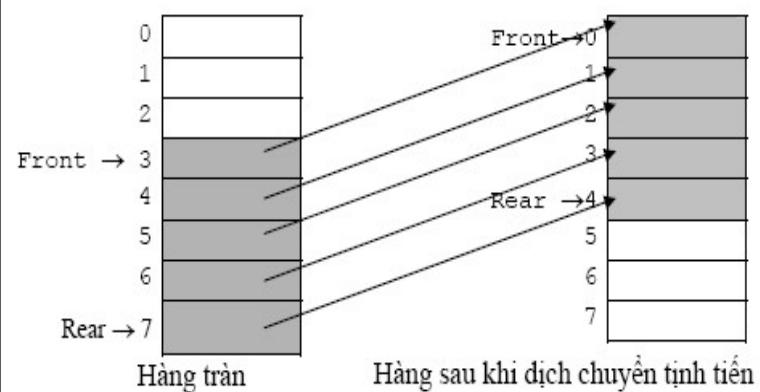
26

Cài đặt hàng bằng mảng

- ◆ Ta dùng một mảng để chứa các phần tử của hàng.
- ◆ khởi đầu phần tử đầu tiên của hàng được đưa vào vị trí thứ 1, phần tử thứ 2 vào vị trí thứ 2 của mảng...
- ◆ Giả sử hàng có **n** phần tử, ta có **front=0** và **rear=n-1**. Khi xoá một phần tử front tăng lên 1, khi thêm một phần tử rear tăng lên 1.
- ◆ Đến một lúc nào đó ta không thể thêm vào hàng được nữa (**rear=maxlength-1**) dù mảng còn nhiều chỗ trống (các vị trí trước front) trường hợp này ta gọi là **hàng bị tràn** Trong trường hợp toàn bộ mảng đã chứa các phần tử của hàng ta gọi là **hàng bị đầy**.

27

Cài đặt hàng bằng mảng



28

Cài đặt hàng bằng mảng

◆ Cách khắc phục hàng bị tràn

- Dời toàn bộ hàng lên front -1 vị trí, cách này gọi là di chuyển tịnh tiến. Trong trường hợp này ta luôn có front<=rear.
- Xem mảng như là một vòng tròn nghĩa là khi hàng bị tràn nhưng chưa đầy ta thêm phần tử mới vào vị trí 0 của mảng, thêm một phần tử nữa thì thêm vào vị trí 1 (nếu có thể)...Rõ ràng cách làm này front có thể lớn hơn rear. Cách khắc phục này gọi là dùng mảng xoay vòng

29

29

Cài đặt hàng bằng mảng theo phương pháp tịnh tiến

```
#define MaxLength ... //chiều dài tối đa của mảng
typedef ... ElementType;
//Kiểu dữ liệu của các phần tử trong hàng
typedef struct
{
    ElementType Elements[MaxLength];
    //Lưu trữ nội dung các phần tử
    int Front, Rear; //chỉ số đầu và đuôi hàng
} Queue;
```

30

30

Cài đặt hàng bằng mảng theo phương pháp tịnh tiến

◆ Tạo hàng rỗng

```
void Makenull_Queue(Queue *Q)
{ Q->Front=-1; Q->Rear=-1; }
```

◆ Kiểm tra hàng rỗng

```
int IsEmpty_Queue(Queue Q)
{ return Q.Front===-1; }
```

◆ Kiểm tra đầy

```
int isFull_Queue(Queue Q)
{ return (Q.Rear-Q.Front+1)==MaxLength; }
```

31

31

Cài đặt hàng bằng mảng theo phương pháp tịnh tiến

◆ Xóa phần tử ra khỏi hàng

```
void Delete_Queue(Queue *Q)
{ if (!IsEmpty_Queue(*Q))
    {
        Q->Front=Q->Front+1;
        if (Q->Front > Q->Rear)
            Makenull_Queue(Q);
        //Dat lai hang rong
    }
    else printf("Loi: Hang rong!");
}
```

32

32

Cài đặt hàng bằng mảng theo phương pháp tịnh tiến

◆ Thêm phần tử vào hàng

```
void EnQueue(ElementType X, Queue *Q){  
    if (!IsFull_Queue(*Q))  
    {  
        if (IsEmpty_Queue(*Q)) Q->Front=0;  
        if (Q->Rear==MaxLength-1)  
        { //Di chuyen tinh tien ra truoc Front -1 vi tri  
            int dem=0;  
            for(int i=Q->Front;i<=Q->Rear;i++)  
            {  
                Q->Elements[i-Q->Front]=Q->Elements[i];  
                dem++;  
            }  
            //Xac dinh vi tri Rear moi  
            Q->Rear=Q->Rear-dem;  
            Q->Front=0;  
        }  
    }  
}
```

33

Cài đặt hàng bằng mảng theo phương pháp tịnh tiến

```
//Tang Rear de luu noi dung moi  
Q->Rear=Q->Rear+1;  
Q->Element[Q->Rear]=X;  
} //!IsEmpty_Queue(Q)  
else printf("Loi: Hang day!");  
}
```

34

33

34

Stack

1/ 12 20 5 9 ** 5 25 40 22 * 7 * 9 * 15 *** 6

Ký hiệu * thay cho thao tác Pop

◆ Kết quả

2/ 1 2 3 *** 5 * 7 8 9 *** 10 11 1 2 3 **

Kết quả

3/ S I * I ** H A A A ** P A S *** P P S * Y

Kết quả: HAPPY

4/ EAS*Y**QUE***SI***I*ON

Kết quả: ON

35

Queue

1/ EAS*Y**QUE***SI***I*ON

Kết quả: ON

2/ S I * I ** H A A A ** P A S *** P P S * Y

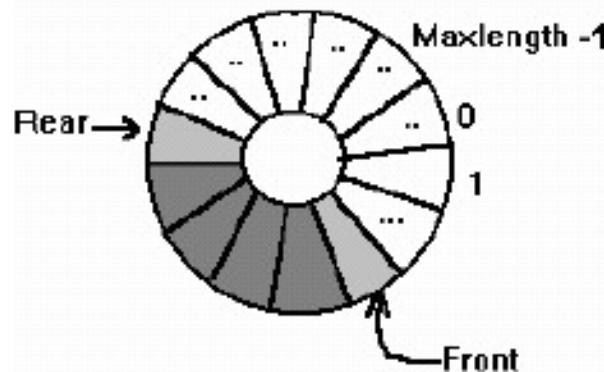
Kết quả: SPPSY

36

35

36

Cài đặt mảng với mảng xoay vòng



37

37

Cài đặt hàng với mảng xoay vòng

```
#define MaxLength ... //chiều dài tối đa của mảng
typedef ... ElementType;
    //Kiểu dữ liệu của các phần tử trong hàng
typedef struct
{
    ElementType Elements[MaxLength];
    //Lưu trữ nội dung các phần tử
    int Front, Rear; //chỉ số đầu và đuôi hàng
} Queue;
```

39

39

Cài đặt hàng với mảng xoay vòng

- ◆ Với phương pháp này khi hàng bị tràn, tức là **rear=maxlength-1**, nhưng chưa đầy, tức là **front>0**, thì ta thêm phần tử mới vào vị trí 0 của mảng và cứ tiếp tục như vậy vì từ 0 đến front-1 là các vị trí trống.
- ◆ Các phần khai báo cấu trúc dữ liệu, tạo hàng rỗng, kiểm tra hàng rỗng **giống như phương pháp di chuyển tịnh tiến**.

38

38

Cài đặt hàng với mảng xoay vòng

- ◆ **Tạo hàng rỗng:** Lúc này front và rear không trỏ đến vị trí hợp lệ nào trong mảng vậy ta có thể cho **front** và **rear đều bằng -1**.

```
void Makenull_Queue(Queue *Q)
{
    Q->Front=-1;
    Q->Rear=-1;
}
```

40

40

Cài đặt hàng với mảng xoay vòng

◆ Kiểm tra hàng rỗng

```
int IsEmpty_Queue(Queue Q)
{
    return Q.Front== -1;
}
```

41

Cài đặt hàng với mảng xoay vòng

◆ Xóa một phần tử ra khỏi hàng

◆ Khi xóa một phần tử ra khỏi hàng, ta xóa tại vị trí đầu hàng và có thể xảy ra các trường hợp sau:

- Nếu hàng rỗng thì báo lỗi không xóa;
 - Ngược lại,
 - Nếu hàng chỉ còn 1 phần tử thì khởi tạo lại hàng rỗng;
 - Ngược lại, thay đổi giá trị của Q.Front.
- (Nếu Q.front != Maxlength-1 thì
đặt lại Q.front = q.Front +1;
Ngược lại Q.front=0)

43

Cài đặt hàng với mảng xoay vòng

◆ Kiểm tra hàng đầy

▪ Hàng đầy nếu toàn bộ các ô trong mảng đang chứa các phần tử của hàng. Với phương pháp này thì front có thể lớn hơn rear. Ta có hai trường hợp hàng đầy như sau:

- **Trường hợp Q.Rear=Maxlength-1 và Q.Front =0**
- **Trường hợp Q.Front =Q.Rear+1.**

◆ Để đơn giản ta có thể gom cả hai trường hợp trên lại theo một công thức như sau:

$$(Q.rear-Q.front +1) \text{ mod Maxlength} =0$$

```
int IsFull_Queue(Queue Q) {
    return (Q.Rear-Q.Front+1) % MaxLength==0;
}
```

42

Cài đặt hàng với mảng xoay vòng

Cài đặt hàng với mảng xoay vòng

```
void DeQueue(Queue *Q)
```

```
{ if (!IsEmpty_Queue(*Q))
{
    //Nếu hàng chỉ chứa một phần tử thì khởi tạo hàng lại
    if (Q->Front==Q->Rear) Makenull_Queue(Q);
    else Q->Front=(Q->Front+1) % Maxlength;
    //tăng Front lên 1 đơn vị
}
else printf("Loi: Hang rong!");
}
```

44

43

44

Cài đặt hàng với mảng xoay vòng

◆ Thêm một phần tử vào hàng

- Khi thêm một phần tử vào hàng thì có thể xảy ra các trường hợp sau:
 - Trường hợp hàng đầy thì báo lỗi và không thêm;
 - Ngược lại, thay đổi giá trị của Q.rear
 - Nếu Q.rear =maxlength-1 thì đặt lại Q.rear=0;
 - Ngược lại Q.rear =Q.rear+1 và đặt nội dung vào vị trí Q.rear mới.

45

45

Cài đặt hàng với mảng xoay vòng

```
void EnQueue(ElementType X,Queue *Q)
{
    if (!IsFull_Queue(*Q))
    {
        if (IsEmpty_Queue(*Q)) Q->Front=0;
        Q->Rear=(Q->Rear+1) % MaxLength;
        Q->Elements[Q->Rear]=X;
    }
    else printf("Loi: Hang day!");
}
```

46

46

Cài đặt hàng bằng con trỏ

◆ Khai báo

```
typedef int ElementType;
struct Node
{
    ElementType Element;
    Node *Next;
};
typedef struct Node *PtrToNode;
typedef PtrToNode Queue;
typedef PtrToNode Position;
```

47

47

Cài đặt hàng bằng con trỏ (tt)

◆ Các hàm

```
void Makenull_Queue( Queue &Q );
int IsEmpty_Queue( Queue Q );
void EnQueue( ElementType X, Queue Q );
void DeQueue( Queue Q );
ElementType Front(Queue Q); // Như hàm Top trong stack
Position Header( Queue Q );
```

48

48

Cài đặt hàng bằng con trỏ (tt)

◆ Tạo hàng rỗng

```
void Makenull_Queue( Queue &Q )
{
    Q = new Node;
    Q->Next = NULL;
}
```

◆ Kiểm tra hàng rỗng

```
int IsEmpty_Queue( Queue Q )
{
    return Q->Next == NULL;
}
```

49

Cài đặt hàng bằng con trỏ (tt)

◆ Chèn phần tử X vào cuối hàng

```
void EnQueue( ElementType X, Queue Q )
{
    Position TmpCell, P;
    P = Header(Q);
    while(P->Next != NULL) //Tim vi tri cuoi hang doi
        P=P->Next;
    TmpCell = new Node;
    if( TmpCell == NULL ) printf( "Out of space!!!" );
    TmpCell->Element = X;
    TmpCell->Next = P->Next;
    P->Next = TmpCell;
}
```

50

Cài đặt hàng bằng con trỏ (tt)

◆ Xóa phần tử đầu hàng

```
void DeQueue(Queue Q )
{
    Position P, TmpCell;
    P = Header(Q);
    if( P->Next!=NULL )
    {
        TmpCell = P->Next;
        P->Next = TmpCell->Next;
        free( TmpCell );
    }
}
```

51

Cài đặt hàng bằng con trỏ (tt)

◆ Trả về vị trí phần tử Header

```
Position Header( Queue Q )
{
    return Q;
}
```

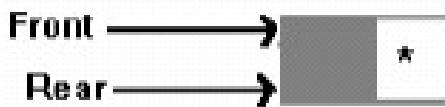
◆ Trả về giá trị phần tử đầu hàng

```
ElementType Front( Queue Q )
{
    return Q->Next->Element;
}
```

52

Cài đặt hàng bằng danh sách liên kết

- ◆ Cách tự nhiên nhất là dùng hai con trỏ front và rear để trỏ tới phần tử đầu và cuối hàng. Hàng được cài đặt như một danh sách liên kết có Header là một ô thực sự,
- ◆ Khi hàng rỗng Front và Rear cùng trỏ về 1 vị trí đó chính là ô header



53

Cài đặt hàng bằng danh sách liên kết (con trỏ)

◆ Khai báo cần thiết

```
typedef int ElementType;
struct Node
{
    ElementType Element;
    Node *Next;
};
typedef struct Node *PtrToNode;
typedef PtrToNode Position;
```

```
struct QueueList
{
    Node *Front;
    Node *Rear;
};
typedef QueueList Queue;
```

54

54

Cài đặt hàng bằng danh sách liên kết (con trỏ)

- ◆ Khởi tạo hàng rỗng

```
void Makenull_Queue(Queue &Q)
{
    Q.Front=Q.Rear=NULL;
}
```

55

Cài đặt hàng bằng danh sách liên kết (con trỏ)

- ◆ Kiểm tra hàng rỗng

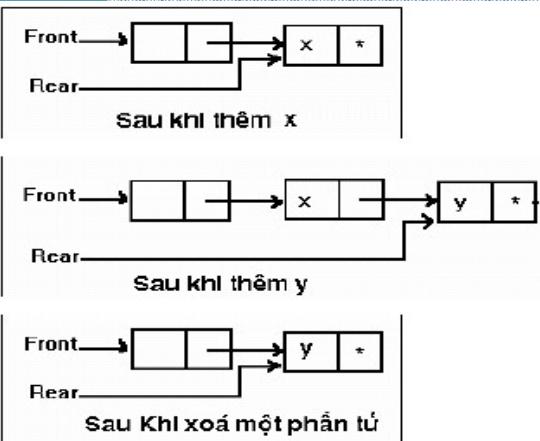
- Hàng rỗng nếu Front và Rear chỉ cùng một vị trí là ô Header hoặc Header chỉ đến NULL

```
ElementType IsEmpty_Queue(Queue Q)
{
    return(Q.Front==NULL);
}
```

56

56

Cài đặt hàng bằng danh sách liên kết (con trỏ)



57

57

Cài đặt hàng bằng danh sách liên kết (con trỏ)

◆ Thêm một phần tử vào hàng

- Thêm một phần tử vào hàng ta thêm vào sau Rear ($\text{Rear} \rightarrow \text{next}$), rồi cho Rear trỏ đến phần tử mới này. Trường next của ô mới này trỏ tới **NULL**.

58

58

Cài đặt hàng bằng danh sách liên kết (con trỏ)

```
void EnQueue(Queue &Q, ElementType X)
{
    Position p;
    p = new Node;
    if(p==NULL) exit(1);
    p->Element=X;
    p->Next=NULL;
    if(IsEmpty_Queue(Q))
        { Q.Front=Q.Rear=p; }
    else
        { Q.Rear->Next=p;
          Q.Rear=p;
        }
}
```

59

59

Cài đặt hàng bằng danh sách liên kết (con trỏ)

◆ Xóa một phần tử ra khỏi hàng

- Thực chất là xoá phần tử nằm ở vị trí đầu hàng do đó ta chỉ cần cho front trỏ tới vị trí kế tiếp của nó trong hàng.

60

60

Cài đặt hàng bằng danh sách liên kết (con trỏ)

```
ElementType DeQueue(Queue &Q)
{ ElementType X;
Position p=Q.Front;
if(IsEmpty_Queue(Q)) return -1;
X=p->Element;
Q.Front=p->Next;
free(p);
return X;
}
```

61

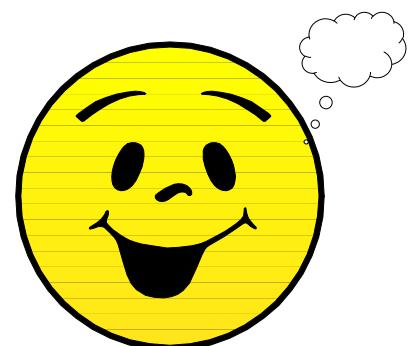
61

Stack-Giai thừa

```
Int GT(int n)           Int Gt(int n)
{
    int a=0,b=0;          {
    a=n*2;b=n*3;          int gt=1;
    if (n==0 || n==1)      for int i=1→ n
                           gt=gt*i;
    return 1;              }
    return n*GT(n-1);      }
} Ví dụ n=5;n=1 triệu //1.000.000
5, 10, 15
4,8,12
3,6,9
2,
1 a b
```

62

62



Cảm ơn !

63

CÂY VÀ CÂY NHỊ PHÂN

- ◆ **Định nghĩa:** Cây là một tập hợp T các phần tử (gọi là nút của cây) trong đó có 1 nút đặc biệt được gọi là gốc, các nút còn lại được chia thành những tập rời nhau T_1, T_2, \dots, T_n theo quan hệ phân cấp trong đó T_i cũng là một cây. Mỗi nút ở cấp i sẽ quản lý một số nút ở cấp $i+1$. Quan hệ này người ta còn gọi là quan hệ cha-con.

1

CÁC THUẬT NGỮ CƠ BẢN TRÊN CÂY

- ◆ Mỗi quan hệ *cha - con* (parenthood): để xác định hệ thống cấu trúc trên các nút.
- ◆ Mỗi nút biểu diễn một phần tử trong tập hợp đang xét
- ◆ Mỗi *quan hệ cha con* được biểu diễn theo qui ước *nút cha* ở dòng trên *nút con* ở dòng dưới và được nối bởi *một đoạn thẳng*.

2

CÁC THUẬT NGỮ CƠ BẢN TRÊN CÂY

- ◆ **Bậc 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.
- ◆ **Nút gốc:** Là nút không có nút cha.
- ◆ **Nút lá:** Là nút có bậc bằng 0 .
- ◆ **Nút nhánh:** Là nút có bậc khác 0 và không phải là gốc.
- ◆ Mỗi nút, trừ nút gốc, có duy nhất một nút cha. Một nút có thể có nhiều nút con hoặc không có nút con nào.

3

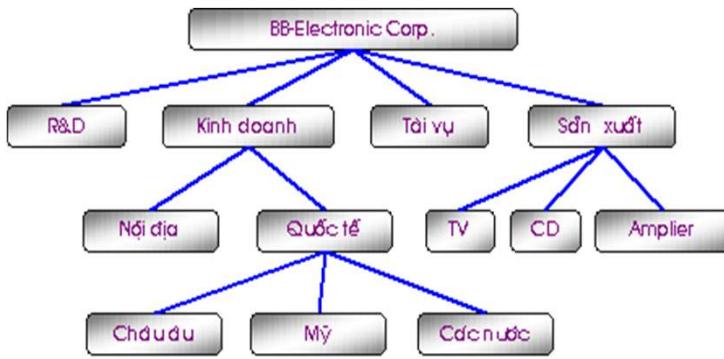
CÁC THUẬT NGỮ CƠ BẢN TRÊN CÂY

- ◆ **Mức của một nút:**
 - Mức (gốc (T)) = 0.
 - Gọi $T_1, T_2, T_3, \dots, T_n$ là các cây con của T_0
 - $\text{Mức}(T_1) = \text{Mức}(T_2) = \dots = \text{Mức}(T_n) = \text{Mức}(T_0) + 1$.
- ◆ **Độ dài đường đi từ gốc đến nút x:**
Là số nhánh cần đi qua kể từ gốc đến x.
- ◆ **Độ dài đường đi trung bình:**
 $PI = PT/n$ (n là số nút trên cây T).
- ◆ **Rừng cây:** Là tập hợp nhiều cây trong đó thứ tự các cây là quan trọng.

4

Một số ví dụ về đối tượng các cấu trúc dạng cây

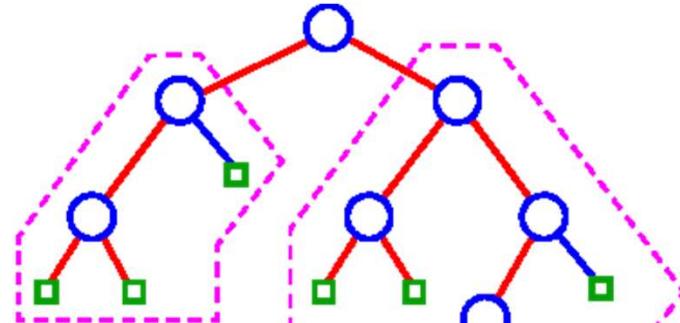
Sơ đồ tổ chức của một công ty



5

6

CÂY NHỊ PHÂN (BINARY TREES)



7

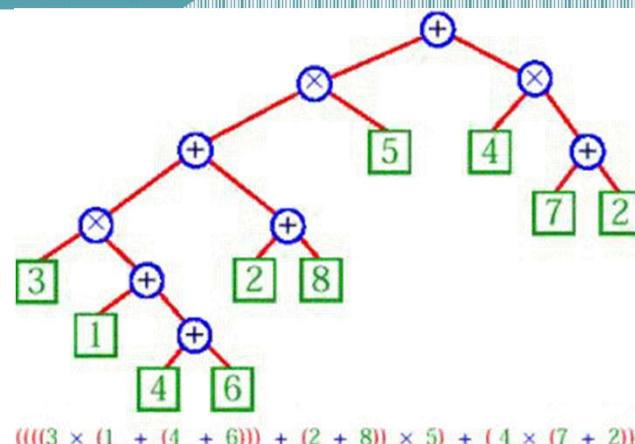
CÂY NHỊ PHÂN (BINARY TREES)

Định nghĩa

- Cây nhị phân là cây rỗng hoặc là cây mà mỗi nút có tối đa hai nút con.
- Các nút con của cây được phân biệt thứ tự rõ ràng
 - một nút con gọi là nút con trái
 - một nút con gọi là nút con phải.
 - Ta quy ước vẽ nút con trái bên trái nút cha và nút con phải bên phải nút cha, mỗi nút con được nối với nút cha của nó bởi một đoạn thẳng.

6

CÂY NHỊ PHÂN (BINARY TREES)



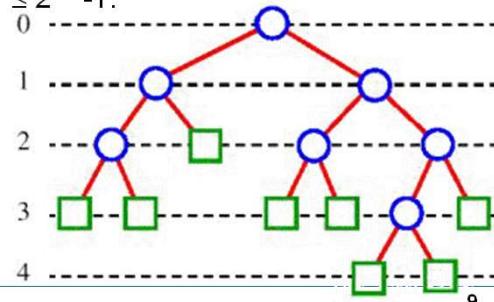
8

7

8

Một số tính chất của cây nhị phân

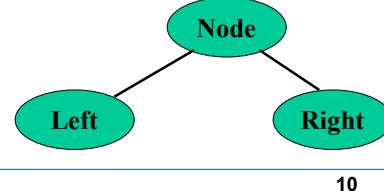
- ◆ Số nút nằm ở mức $l \leq 2^l$
- ◆ Số nút lá $\leq 2^h$, với h là chiều cao của cây = mức lớn nhất của nút trên cây.
- ◆ Chiều cao của cây $h \geq \log_2(\text{số nút trong cây})$.
- ◆ Số nút trong cây $\leq 2^{h+1}-1$.



9

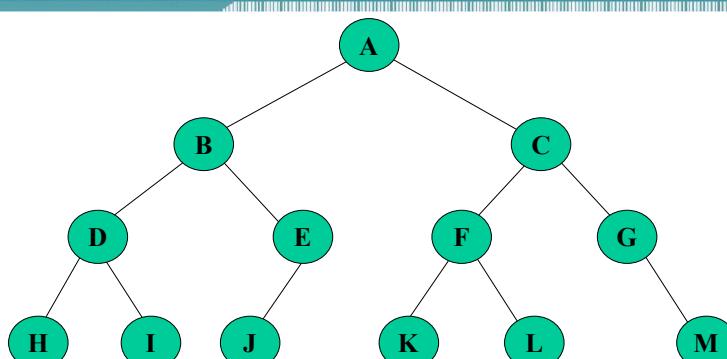
Cây nhị phân

- ◆ Duyệt cây nhị phân
 - **Duyệt tiền tự (Node-Left-Right)**: duyệt nút gốc, duyệt tiền tự con trái rồi duyệt tiền tự con phải.
 - **Duyệt trung tự (Left-Node-Right)**: duyệt trung tự con trái rồi đến nút gốc sau đó là duyệt trung tự con phải.
 - **Duyệt hậu tự (Left-Right-Node)**: duyệt hậu tự con trái rồi duyệt hậu tự con phải sau đó là nút gốc.



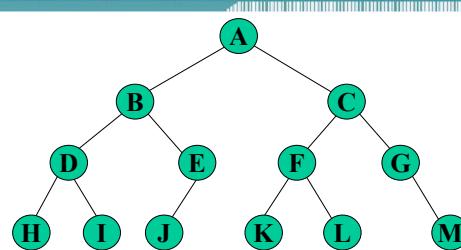
10

Cây nhị phân



11

Cây nhị phân



Các danh sách duyệt cây nhị phân

Tiền tự	A B D H I E J C F K L G M
Trung tự	H D I B J E A K F L C G M
Hậu tự	H I D J E B K L F M G C A

12

Cài đặt cây nhị phân

```
typedef int ElementType;
struct TreeNode;
typedef struct TreeNode *Node;
typedef struct TreeNode *Tree;
//Khai bao cay nhi phan
struct TreeNode
{
    ElementType Element;
    Node Left;      //Con tro Trai
    Node Right;     //Con tro Phai
};
```

13

Cài đặt cây nhị phân

◆ **Tạo cây rỗng :** Cây rỗng là một cây mà không chứa một nút nào cả.

```
Tree MakeEmpty(Tree T)
{
    if(T!=NULL)
    {
        MakeEmpty(T->Left);
        MakeEmpty(T->Right);
        free(T);
    }
    return NULL;
}
```

14

Cài đặt cây nhị phân

◆ Kiểm tra cây rỗng

```
int IsEmpty_Tree(Tree T)
{
    return (T==NULL);
}
```

15

Cài đặt cây nhị phân

◆ Xác định con trái của một nút

```
Node LeftChild(Tree p)
{
    if (p!=NULL)
        return p->Left;
    else
        return NULL;
}
```

◆ Xác định con phải của một nút

```
Node RightChild(Tree p)
{
    if (p!=NULL)
        return p->Right;
    else
        return NULL;
}
```

16

15

16

Cài đặt cây nhị phân

◆ Kiểm tra nút lá:

- Nếu nút là nút lá thì nó không có bất kỳ một con nào cả nên khi đó con trái và con phải của nó cùng bằng NULL

```
int IsLeaf(Tree p)
{   if(p!=NULL)
    return(LeftChild(p)==NULL)
    &&(RightChild(p)==NULL);
else
    return 0;
}
```

17

18

```
return(LeftChild(p)==NULL)
&&(RightChild(p)==NULL)
```

◆ return (x%2==0)

Tương đương

◆ Tương đương

If(x%2==0)

```
&(LeftChild(p)==NULL)&
&(RightChild(p)==NULL)
```

return 1;

Else

Else

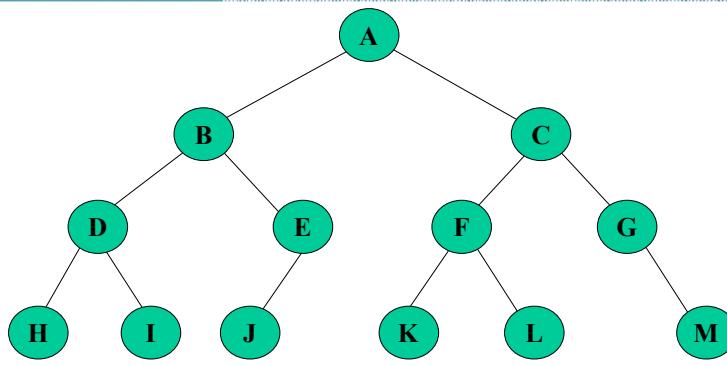
return 0;

return 0;

17

18

Cây nhị phân



19

Cài đặt cây nhị phân

◆ Xác định số nút của cây

```
int nb_nodes(Tree T)
{
    if(IsEmpty_Tree(T))
        return 0;
    else
        return 1 + nb_nodes(LeftChild(T))
            + nb_nodes(RightChild(T));
}
```

19

20

Cài đặt cây nhị phân

◆ Thủ tục duyệt tiền tự

```
void PreOrder(Tree T)
{
    if(T!=NULL)
        printf("\t%d",T->Element);
    if (LeftChild(T)!=NULL)
        PreOrder(LeftChild(T));
    if (RightChild(T)!=NULL)
        PreOrder(RightChild(T));
}
```

21

Cài đặt cây nhị phân

◆ Thủ tục duyệt trung tự

```
void InOrder(Tree T)
{
    if (LeftChild(T)!=NULL)
        InOrder(LeftChild(T));
    printf("\t%d",T->Element);
    if (RightChild(T)!=NULL)
        InOrder(RightChild(T));
}
```

22

Cài đặt cây nhị phân

◆ Thủ tục duyệt hậu tự

```
void PosOrder(TTree T)
{
    if (LeftChild(T)!=NULL)
        PosOrder(LeftChild(T));
    if (RightChild(T)!=NULL)
        PosOrder(RightChild(T));
    printf("\t%d ",T->Element);
}
```

23

CÂY TÌM KIẾM NHỊ PHÂN (BINARY SEARCH TREES)

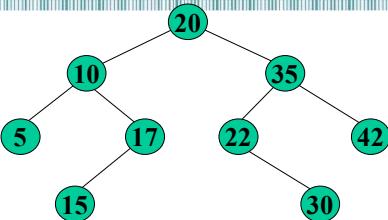
◆ Định nghĩa

- Cây tìm kiếm nhị phân (TKNP) là cây nhị phân mà khoá tại mỗi nút cây lớn hơn khoá của tất cả các nút thuộc cây con bên trái và nhỏ hơn khoá của tất cả các nút thuộc cây con bên phải.

- ◆ Lưu ý: khoá của nút được tính dựa trên một trường nào đó, ta gọi là trường khoá. Trường khoá phải chứa các giá trị có thể so sánh được, tức là nó phải lấy giá trị từ một tập hợp có thứ tự.

24

Binary Search Trees - BST



một cây TKNP có khoá là số nguyên (với quan hệ thứ tự trong tập số nguyên).

Trung tự: 5 10 15 17 20 22 30 35 42

Tiền tự: 20 10 5 17 15 35 22 30 42

Hậu tự: 5 15 17 10 30 22 42 35 20

25

26

Binary Search Trees - BST

◆ **Qui ước:** Cũng như tất cả các cấu trúc khác, ta coi cây rỗng là cây TKNP

◆ Nhận xét:

- Trên cây TKNP không có hai nút cùng khoá.
- Cây con của một cây TKNP là cây TKNP.
- Khi duyệt trung tự (InOrder) cây TKNP ta được một dãy có thứ tự tăng. Chẳng hạn duyệt trung tự cây trên ta có dãy: 5, 10, 15, 17, 20, 22, 30, 35, 42.

25

26

BST – Cài đặt

◆ Có thể áp dụng các cách cài đặt như đã trình bày trong phần cây nhị phân.

◆ Một cách cài đặt cây TKNP thường gặp là cài đặt bằng con trỏ. Mỗi nút của cây là một mẫu tin (struct) có ba trường:

- Khoá (Key)
- Nút trái (Left)
- Nút phải (Right)

(nếu nút con vắng mặt ta gán con trỏ bằng NULL)

27

28

Cài đặt cây nhị phân

```
typedef ... ElementType;
struct TreeNode;
typedef struct TreeNode *Node;
typedef struct TreeNode *Tree;
//Khai bao cay nhi phan
struct TreeNode
{
    ElementType Element;
    Node Left;      //Con tro Trai
    Node Right;     //Con tro Phai
};
```

27

28

BST – Cài đặt

◆ Tìm kiếm một nút có khóa cho trước trên cây TKNP

Để tìm kiếm 1 nút có khóa x trên cây TKNP, ta tiến hành từ nút gốc bằng cách so sánh khóa của nút gốc với khóa x.

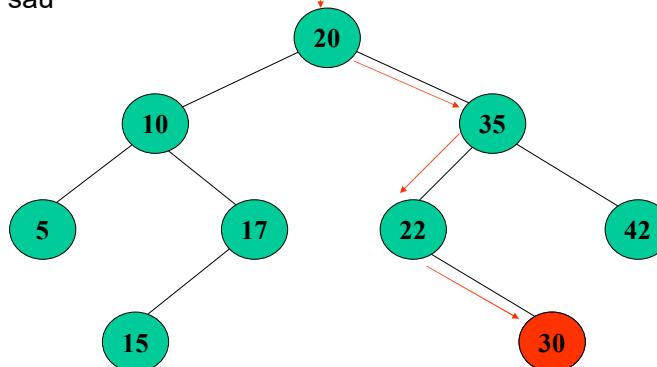
- **Nếu nút gốc bằng NULL** thì không có khóa x trên cây.
- **Nếu x bằng khóa của nút gốc** thì giải thuật dừng và ta đã tìm được nút chứa khóa x.
- **Nếu x lớn hơn khóa của nút gốc** thì ta tiến hành (một cách đệ qui) việc tìm khóa x trên cây con bên phải.
- **Nếu x nhỏ hơn khóa của nút gốc** thì ta tiến hành (một cách đệ qui) việc tìm khóa x trên cây con bên trái.

29

29

BST – Cài đặt

◆ Ví dụ: tìm nút có khóa x= 30 trong cây ở trong hình sau



30

30

BST – Cài đặt

◆ Hàm dưới đây trả về kết quả là con trỏ trả về nút chứa khóa x hoặc NULL nếu không tìm thấy khóa x

Node Search(ElementType X, Tree T)

```
{  
    if(T == NULL)  
        return NULL;  
    if(X < T->Element)  
        return Search( X, T->Left);  
    else if(X > T->Element)  
        return Search(X, T->Right);  
    else  
        return T;  
}
```

31

31

BST – Thêm một nút có khóa cho trước vào cây TKNP

◆ Thêm một nút có khóa cho trước vào cây TKNP

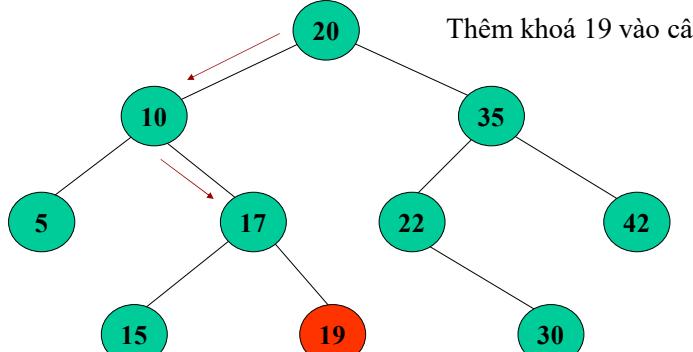
Ta tiến hành từ nút gốc bằng cách so sánh khóa của nút gốc với khóa x.

- **Nếu nút gốc bằng NULL** thì khóa x chưa có trên cây, do đó ta thêm một nút mới chứa khóa x.
- **Nếu x bằng khóa của nút gốc** thì giải thuật dừng, trường hợp này ta không thêm nút.
- **Nếu x lớn hơn khóa của nút gốc** thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên phải.
- **Nếu x nhỏ hơn khóa của nút gốc** thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên trái.

32

32

BST–Thêm một nút có khóa cho trước vào cây TKNP



33

33

BST–Thêm một nút có khóa cho trước vào cây TKNP

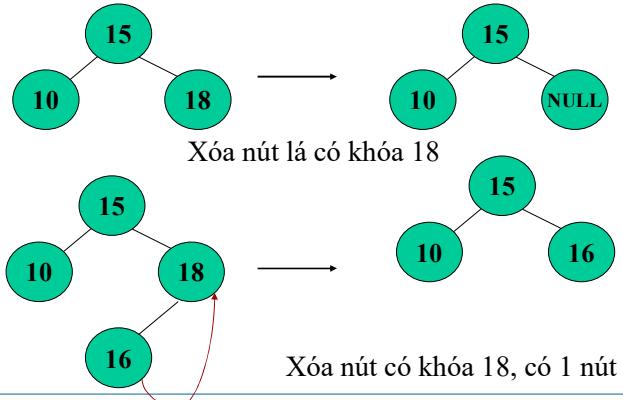
```
Tree Insert(ElementType X,Tree T)
{ if(T==NULL)
  { T= (TreeNode*) malloc(sizeof(struct TreeNode) );
    //tương đương với: T=new Node;
    if(T==NULL) printf("Out of space!");//Loi
    else
    { T->Element = X;
      T->Left = T->Right = NULL;
    }
  }
  else if(X < T->Element)
    T->Left = Insert(X, T->Left);
  else if(X > T->Element)
    T->Right = Insert(X,T->Right);
  return T;
}
```

34

34

BST–Xóa một nút có khóa cho trước ra khỏi cây TKNP

◆ Xóa một nút có khóa cho trước ra khỏi cây TKNP

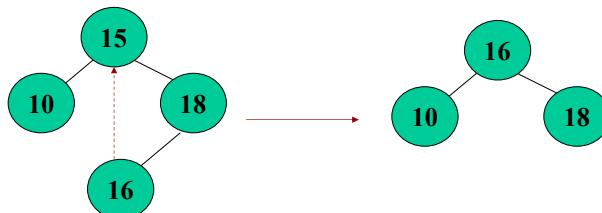


35

35

BST–Xóa một nút có khóa cho trước ra khỏi cây TKNP

◆ Xóa một nút 15 có 2 nút con



36

36

BST–Xóa một nút có khóa cho trước ra khỏi cây TKNP

- ◆ **Giải thuật xóa một nút có khóa cho trước**
- ◆ Nếu không tìm thấy nút chứa khoá x thì giải thuật kết thúc.
- ◆ Nếu tìm gặp nút N có chứa khoá x, ta có ba trường hợp sau
 - **Nếu N là lá** ta thay nó bởi NULL.
 - **N chỉ có một nút con** ta thay nó bởi nút con của nó.
 - **N có hai nút con** thay nó bởi nút lớn nhất trên cây con trái của nó (nút cực phải của cây con trái) hoặc là nút bé nhất trên cây con phải của nó (nút cực trái của cây con phải).

37

37

BST–Xóa một nút có khóa cho trước ra khỏi cây TKNP

- ◆ Trong giải thuật sau, ta thay x bởi khoá của nút cực trái của cây con bên phải rồi ta xóa nút cực trái này.
- ◆ Hàm sau trả về khoá của nút cực trái, đồng thời xoá nút này.

```
Node FindMin(Tree T)
{ if(T==NULL)
    return NULL;
else if(T->Left == NULL)
    return T;
else
    return FindMin(T->Left);}
```

38

38

BST–Xóa một nút có khóa cho trước ra khỏi cây TKNP

```
Tree Delete(ElementType X,Tree T)
{ Node TmpCell;
if(T== NULL) printf("Element not found");
else
    if (X < T->Element)
        T->Left = Delete(X, T->Left);
    else
        if(X > T->Element)
            T-> Right = Delete(X, T->Right);
    //else
```

39

39

BST–Xóa một nút có khóa cho trước ra khỏi cây TKNP

```
else //Nút có 2 con
if(T->Left!=NULL && T->Right!=NULL)
    { TmpCell = FindMin(T->Right);
    T->Element = TmpCell->Element;
    T->Right = Delete(T->Element, T->Right);
    }
else
    { TmpCell = T;
    if (T->Left == NULL)          T = T->Right;
    else if (T->Right == NULL)   T = T->Left ;
    free(TmpCell); //Xoa nut
    }
return T;
```

40

40



Cảm ơn !

◆ Xóa nút có giá trị 83

- Do nút này có hai con
- Và yêu cầu giữ cây con trái
- Nên thay nó bằng con cực trái của cây con phải, chính là nút có giá trị 85
- Sau đó do nút 85 có 1 con, Nên thay nó bằng con, Chính là nút có giá trị 88

