



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



Cấu trúc dữ liệu và giải thuật

Nguyễn Khánh Phương

**Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn**

Nội dung khóa học

Chương 1. Các khái niệm cơ bản

Chương 2. Các cấu trúc dữ liệu cơ bản

Chương 3. Cây

Chương 4. Sắp xếp

Chương 5. Tìm kiếm



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



Chương 4. Sắp xếp

Nguyễn Khánh Phương

**Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn**

Bài toán sắp xếp

- **Sắp xếp (Sorting)** là quá trình tổ chức lại họ các dữ liệu theo thứ tự giảm dần hoặc tăng dần.
- Dữ liệu cần sắp xếp có thể là:
 - Số nguyên/thực.. (integers/float)
 - Xâu kí tự (character strings)
 - ...
- Khóa sắp xếp (sort key)
 - Là bộ phận của bản ghi xác định thứ tự sắp xếp của bản ghi trong họ các bản ghi.
 - Ta cần sắp xếp các bản ghi theo thứ tự của các khoá.
 - Ví dụ: khóa là tong = toan + ly + hoa

```
typedef struct{
    char *ma;
    struct{
        float toan, ly, hoa, tong;
    } DT;
} thisinh;

typedef struct node{
    thisinh data;
    struct node* next;
} node;
```

Các loại thuật toán sắp xếp

Sắp xếp trong (internal sort):

- Đòi hỏi họ dữ liệu được đưa toàn bộ vào bộ nhớ trong của máy tính
- Ví dụ:
 - insertion sort (sắp xếp chèn), selection sort (sắp xếp lựa chọn), bubble sort (sắp xếp nổi bọt)
 - quick sort (sắp xếp nhanh), merge sort (sắp xếp trộn), heap sort (sắp xếp vun đống), sample sort (sắp xếp dựa mẫu), shell sort (vỏ sò)

Sắp xếp ngoài (external sort):

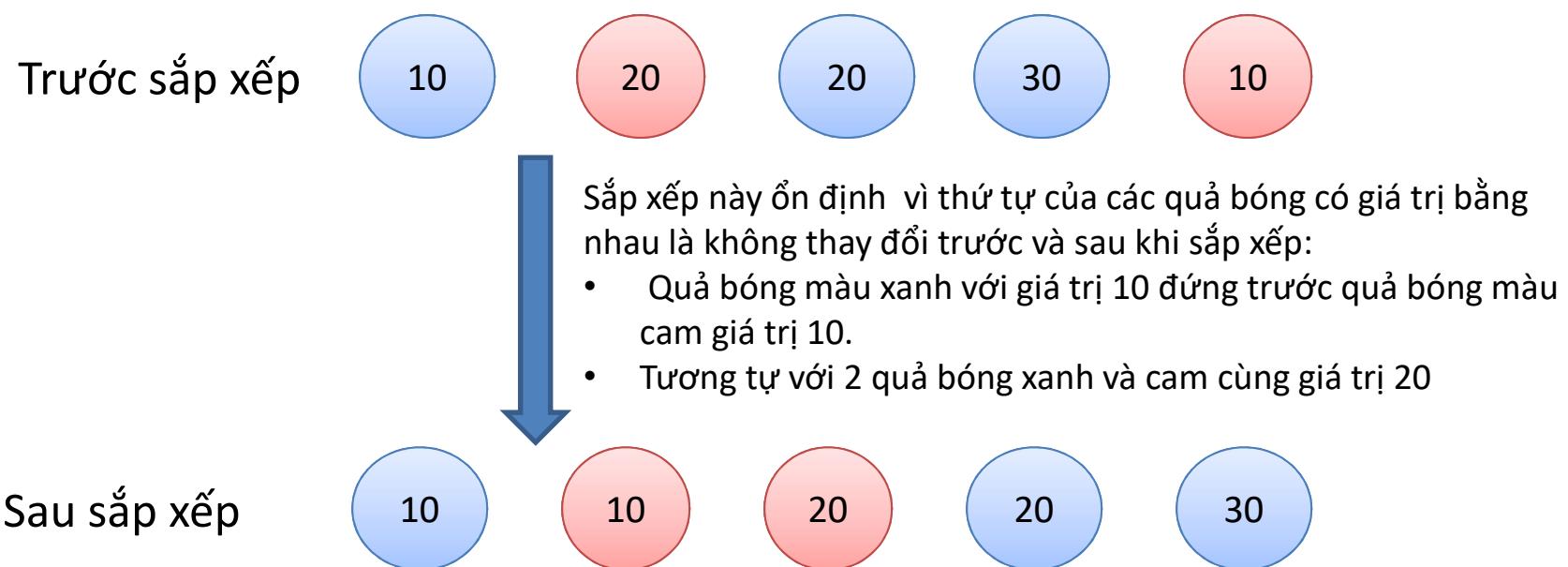
- Họ dữ liệu không thể cùng lúc đưa toàn bộ vào bộ nhớ trong, nhưng có thể đọc vào từng bộ phận từ bộ nhớ ngoài
- Ví dụ: Poly-phase mergesort (trộn nhiều đoạn), cascade-merge (thác nước), oscillating sort (đao động)

Sắp xếp song song (Parallel sort):

- Bitonic sort, Batcher even-odd sort.
- Smooth sort, cube sort, column sort.
- GPU sort.

Các đặc trưng của một thuật toán sắp xếp

- **Tại chỗ (in place):** nếu không gian nhớ phụ mà thuật toán đòi hỏi là $O(1)$, nghĩa là bị chặn bởi hằng số không phụ thuộc vào độ dài của dãy cần sắp xếp.
- **Ôn định (stable):** Nếu các phần tử có cùng giá trị vẫn giữ nguyên thứ tự tương đối của chúng như trước khi sắp xếp.



Bài toán sắp xếp

- Có hai phép toán cơ bản mà thuật toán sắp xếp thường phải sử dụng:
 - Đổi chỗ** (Swap): Thời gian thực hiện là $O(1)$

```
void swap( datatype *a, datatype *b) {  
    datatype *temp = *a; //datatype-kiểu dữ liệu của phần tử  
    *a = *b;  
    *b = *temp;  
}
```

```
void swap(int *a, int *b) {  
    int *temp = *a;  
    *a = *b;  
    *b = *temp;  
}
```

- So sánh:** Compare(a, b) trả lại true nếu a đi trước b trong thứ tự cần sắp xếp và false nếu trái lại.
- Phân tích thuật toán sắp xếp:** thông thường, các thuật toán sẽ sử dụng phép toán so sánh để xác định thứ tự giữa hai phần tử rồi thực hiện đổi chỗ nếu cần.
→ Khi phân tích thuật toán sắp xếp, ta sẽ chỉ cần đếm số phép toán so sánh và số lần dịch chuyển các phần tử (bỏ qua các phép toán khác không ảnh hưởng đến kết quả).

Bài toán sắp xếp

- Các thuật toán chỉ sử dụng phép toán so sánh để xác định thứ tự giữa hai phần tử được gọi là thuật toán sử dụng phép so sánh (*Comparison-based sorting algorithm*).
- Nếu có những thông tin bổ sung về dữ liệu đầu vào, ví dụ như:
 - Các số nguyên nằm trong khoảng $[0..k]$ trong đó $k = O(n)$
 - Các số thực phân bố đều trong khoảng $[0, 1]$

ta sẽ có thuật toán tốt hơn thuật toán sắp xếp chỉ dựa vào phép so sánh.

(Thuật toán thời gian tuyến tính: sắp xếp đếm (couting-sort), sắp xếp theo cơ số (radix-sort), sắp xếp đóng gói (bucket-sort))

Các thuật toán sắp xếp

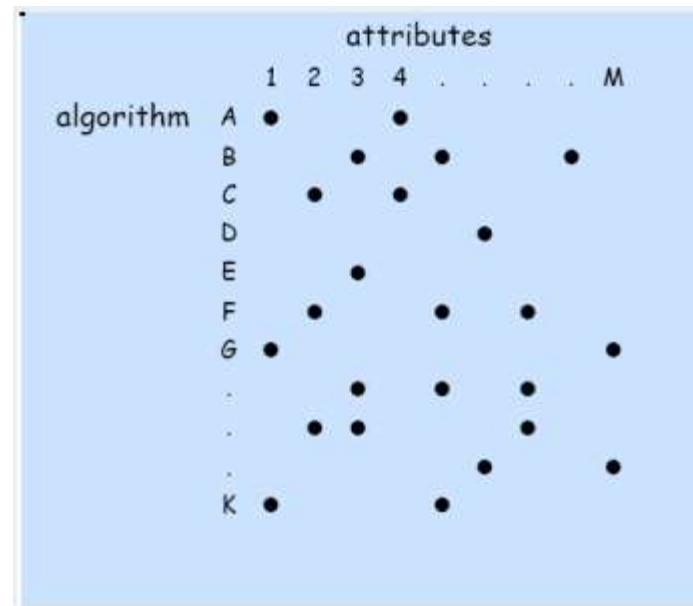
Khi so sánh các thuật toán, thông thường quan tâm đến:

- **Thời gian** chạy. Đối với các dữ liệu rất lớn, các thuật toán không hiệu quả sẽ chạy rất chậm và không thể ứng dụng trong thực tế.
- **Bộ nhớ**. Các thuật toán nhanh đòi hỏi đệ quy sẽ có thể phải tạo ra các bản copy của dữ liệu đang xử lí. Với những hệ thống mà bộ nhớ có giới hạn (ví dụ embedded system), một vài thuật toán sẽ không thể chạy được.
- **Độ ổn định (stability)**. Độ ổn định được định nghĩa dựa trên điều gì sẽ xảy ra với các phần tử có giá trị giống nhau.
 - Đối với thuật toán sắp xếp ổn định, các phần tử với giá trị bằng nhau sẽ giữ nguyên thứ tự trong mảng trước và sau khi sắp xếp.
 - Đối với thuật toán sắp xếp không ổn định, các phần tử có giá trị bằng nhau sẽ có thể có thứ tự bất kỳ.

Tiêu chí lựa chọn giải thuật

Nhiều yếu tố ảnh hưởng:

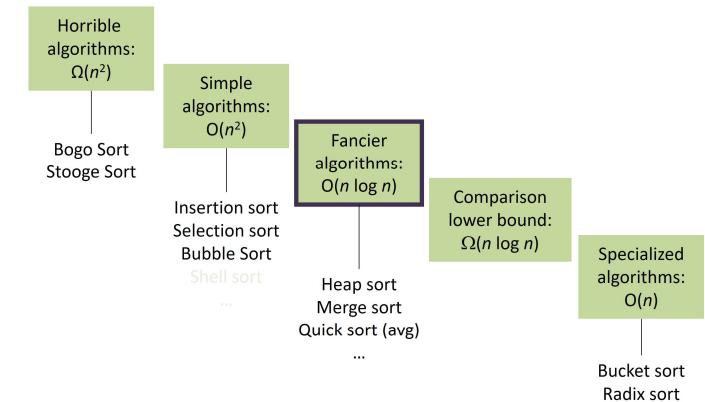
- Ôn định
- Danh sách liên kết hay mảng
- Đặc trưng của dữ liệu cần sắp xếp:
 - Nhiều khóa ?
 - Các khóa là phân biệt ?
 - Nhiều dạng khóa ?
 - Kích thước bản ghi lớn hay nhỏ ?
 - Dữ liệu được sắp xếp ngẫu nhiên?



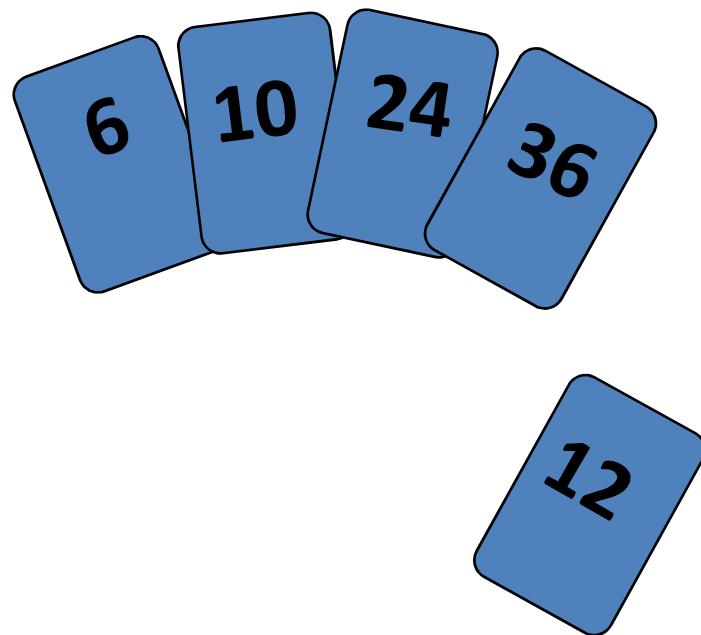
Không thể bao phủ tất cả các yếu tố

Nội dung

1. Sắp xếp chèn (Insertion sort)
2. Sắp xếp chọn (Selection sort)
3. Sắp xếp nổi bọt (Bubble sort)
4. Sắp xếp trộn (Merge sort)
5. Sắp xếp nhanh (Quick sort)
6. Sắp xếp vun đống (Heap sort)



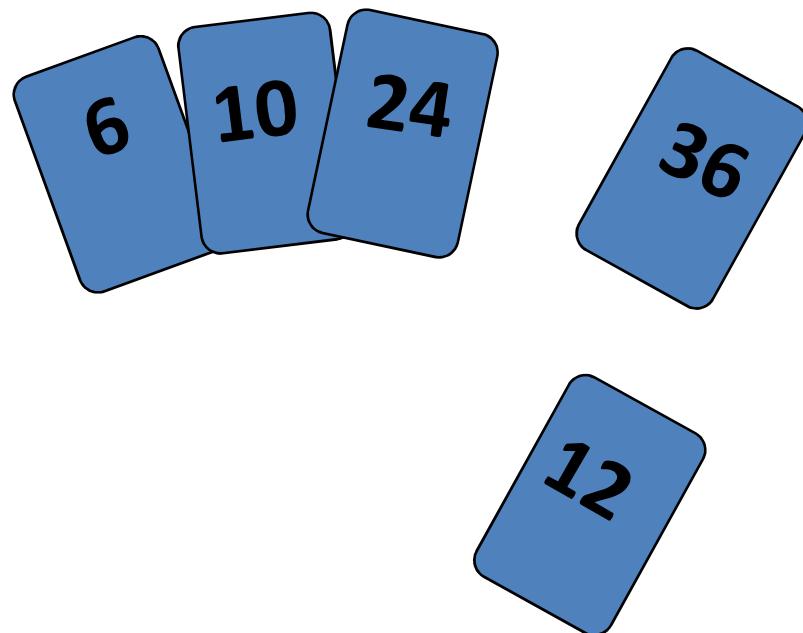
1. Sắp xếp chèn (Insertion sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

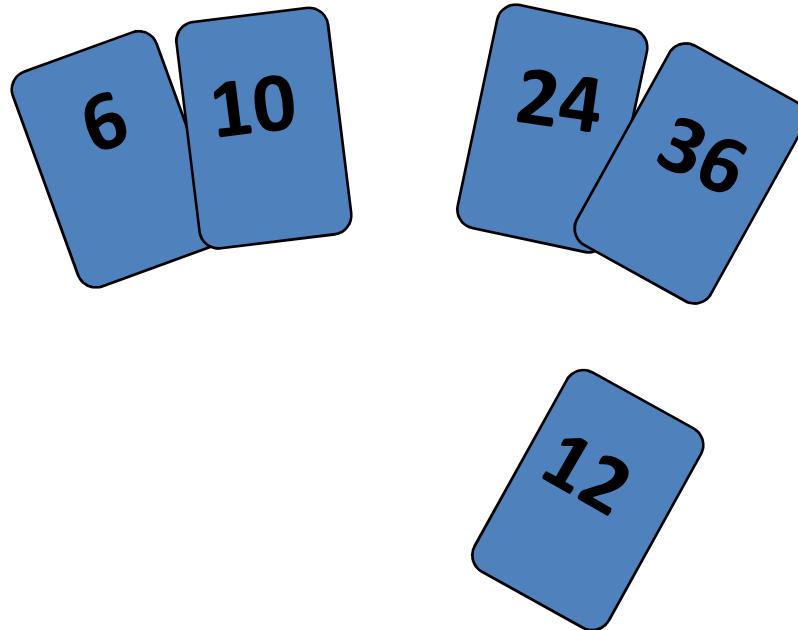
1. Sắp xếp chèn (Insertion sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

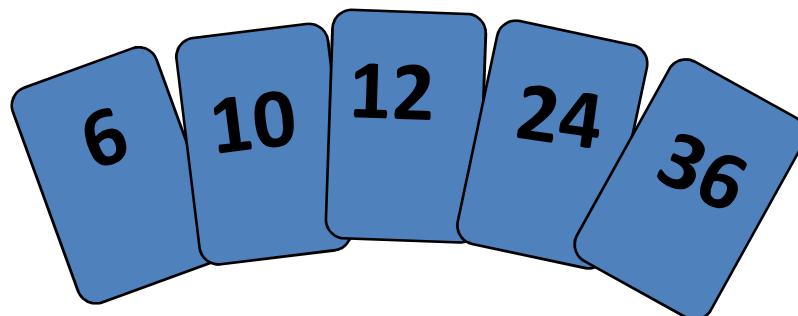
1. Sắp xếp chèn (Insertion sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

1. Sắp xếp chèn (Insertion sort)



Phỏng theo cách làm của người chơi bài khi cần "chèn" thêm một con bài vào bộ bài đã được sắp xếp trên tay.

Để chèn 12, ta cần tạo chỗ cho nó bởi việc dịch chuyển đầu tiên là 36 và sau đó là 24.

1. Sắp xếp chèn (Insertion sort)

Thuật toán:

- Mảng cần sắp xếp được chia làm 2 phần, *sorted* và *unsorted*:
 - Những phần tử nằm trong phần *sorted*: đã được sắp xếp
 - Những phần tử nằm trong phần *unsorted*: chưa được sắp xếp
 - Mỗi bước lặp: phần tử đầu tiên thuộc phần *unsorted* sẽ được chuyển sang phần *sorted*.
- Mảng có n phần tử sẽ cần $n-1$ bước lặp để sắp xếp xong

Sorted

Unsorted

23	78	45	8	32	56
----	----	----	---	----	----

Original array

23	78	45	8	32	56
----	----	----	---	----	----

After iteration 1

23	45	78	8	32	56
----	----	----	---	----	----

After iteration 2

8	23	45	78	32	56
---	----	----	----	----	----

After iteration 3

8	23	32	45	78	56
---	----	----	----	----	----

After iteration 4

8	23	32	45	56	78
---	----	----	----	----	----

After iteration 5

1. Sắp xếp chèn (Insertion sort)

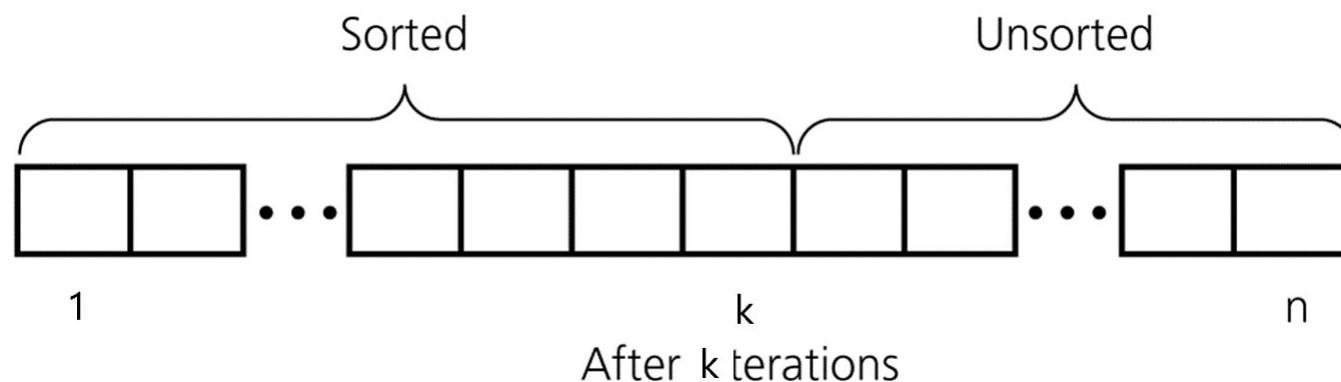
- **Thuật toán:**

- Tại bước $k = 1, 2, \dots, n$:

- đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.

Tại mỗi bước lặp k , có thể cần nhiều hơn một lần hoán đổi vị trí các phần tử để có thể đưa phần tử thứ k về đúng vị trí của nó trong dãy cần sắp xếp

Bước lặp k : liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó



Tính chất: Sau bước lặp k , k phần tử đầu tiên $a[1], a[2], \dots, a[k]$ đã được sắp thứ tự.

Cài đặt: Insertion Sort Algorithm

k=5: tìm vị trí đúng cho a[5]=14

```
void insertionSort(int a[], int size);
```

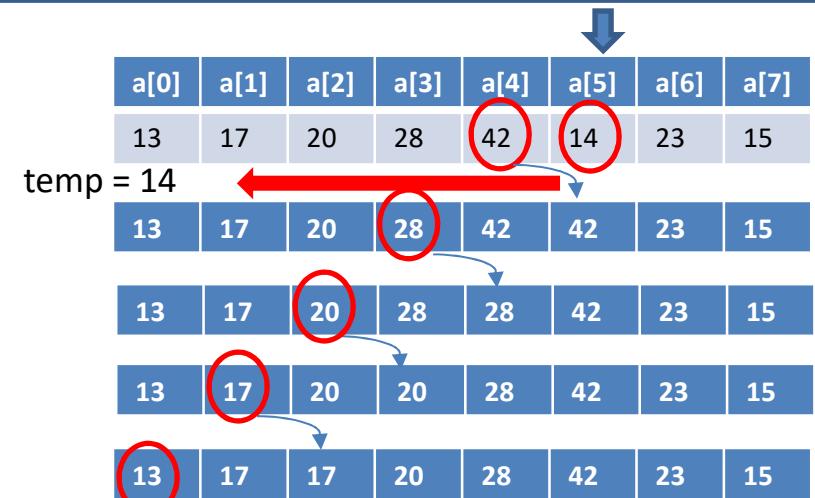
- Thuật toán:

- Tại bước $k = 1, 2, \dots, n$:

đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.

Tại mỗi bước lặp k , có thể cần nhiều hơn một lần hoán đổi vị trí các phần tử để có thể đưa phần tử thứ k về đúng vị trí của nó trong dãy cần sắp xếp

Bước lặp k : liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó



```
for (int k = 1; k < size; k++) {  
    int temp = a[k];  
    int pos = k;  
    /* bước lặp k: liên tục đổi chỗ phần tử thứ k với phần tử kề bên  
       trái nó chừng nào phần tử thứ k còn nhỏ hơn phần tử đó */  
    while (pos > 0 && a[pos-1] > temp) {  
        a[pos] = a[pos-1];  
        pos--;  
    } // end while
```

}

Cài đặt: Insertion Sort Algorithm

k=5: tìm vị trí đúng cho a[5]=14

```
void insertionSort(int a[], int size);
```

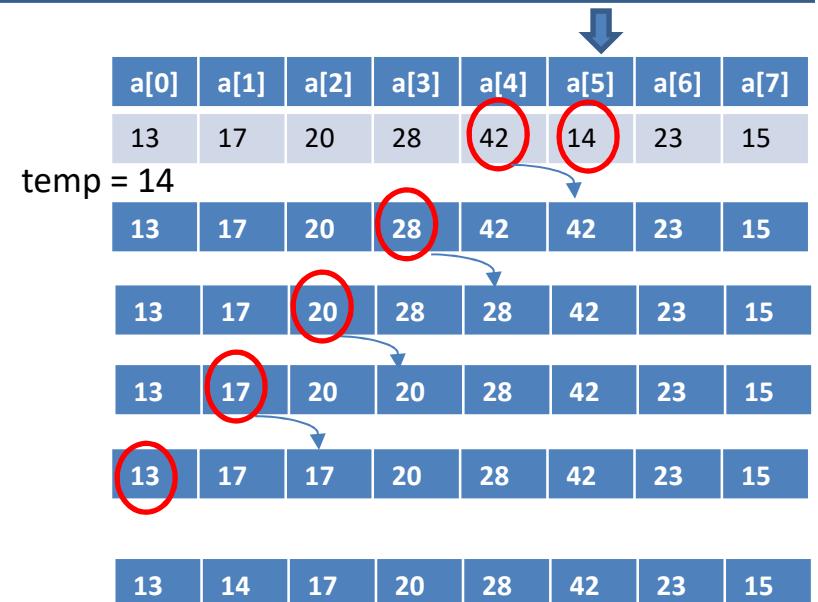
- Thuật toán:

- Tại bước $k = 1, 2, \dots, n$:

đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.

Tại mỗi bước lặp k , có thể cần nhiều hơn một lần hoán đổi vị trí các phần tử để có thể đưa phần tử thứ k về đúng vị trí của nó trong dãy cần sắp xếp

Bước lặp k : liên tục đổi chỗ phần tử thứ k với phần tử kề bên trái nó (phần tử ngay trước) chừng nào phần tử thứ k còn nhỏ hơn phần tử đó



```
for (int k = 1; k < size; k++) {  
    int temp = a[k];  
    int pos = k;  
    /* bước lặp k: liên tục đổi chỗ phần tử thứ k với phần tử kề bên  
       trái nó chừng nào phần tử thứ k còn nhỏ hơn phần tử đó */  
    while (pos > 0 && a[pos-1] > temp) {  
        a[pos] = a[pos-1];  
        pos--;  
    } // end while  
    // Chèn giá trị temp (=a[k]) vào đúng vị trí  
    a[pos] = temp;  
}
```

Cài đặt: Insertion Sort Algorithm

```
void insertionSort(int a[], int size) {
    int k, pos, temp;
    for (k=1; k < size; k++) {
        temp = a[k];
        pos = k;
        while ((pos > 0) && (a[pos-1] > temp)) {
            a[pos] = a[pos-1];
            pos = pos - 1;
        }
        a[pos] = temp;
    }
}

void main()
{
    int a[5] = {8,4,3,2,1};
    insertionSort(a,5);
    for (int i = 0; i<5; i++)
        printf("%d \n",a[i]);
}
```

Đánh giá độ phức tạp tính toán của Insertion sort: $O(n^2)$

- Sắp xếp chèn là tại chỗ và ổn định (In place and Stable)
- Phân tích thời gian tính của thuật toán
 - **Best Case:** 0 hoán đổi, $n-1$ so sánh (khi dãy đầu vào là đã được sắp)
 - **Worst Case:** $n^2/2$ hoán đổi và so sánh (khi dãy đầu vào có thứ tự ngược lại với thứ tự cần sắp xếp)
 - **Average Case:** $n^2/4$ hoán đổi và so sánh
- Là thuật toán sắp xếp tốt đối với dãy *đã gần được sắp xếp*
 - Nghĩa là mỗi phần tử đã đứng ở vị trí rất gần vị trí trong thứ tự cần sắp xếp

# of Sorted elements	Best case	Worst case
0	0	0
1	1	1
2	1	2
...
$n-1$	1	$n-1$
Số phép so sánh	$n-1$	$n(n-1)/2$

Đánh giá độ phức tạp tính toán của Insertion sort

Vòng lặp for:

- thực hiện $n-1$ lần
 - Gồm 5 câu lệnh (bao gồm cả câu lệnh gán và so sánh ở vòng lặp for)
- ➔ Tổng chi phí cho vòng lặp for: $5(n-1)$

Số lần vòng lặp while thực hiện phụ thuộc vào trạng thái sắp xếp các phần tử trong mảng:

- Best case: mảng đầu vào đã được sắp xếp ➔ do đó phép hoán đổi vị trí các phần tử trong mảng không được thực hiện lần nào.
 - Với mỗi giá trị k ở vòng lặp for, ta chỉ test điều kiện thực hiện vòng lặp while đúng 1 lần (2 thao tác = 1 phép so sánh $pos > 0$ và 1 phép so sánh phần tử $a[pos-1] > temp$), các câu lệnh trong thân vòng lặp while không bao giờ được thực hiện
 - Do đó, tổng chi phí cho vòng lặp while trong toàn bộ chương trình là $2(n-1)$ thao tác.
- Worst case: mảng đầu vào có thứ tự ngược với thứ tự cần sắp xếp
 - Vòng lặp for thứ k : vòng lặp while thực hiện tổng cộng $4k+1$ thao tác
 - Do đó, tổng chi phí cho vòng lặp while trong toàn bộ chương trình là $2n(n-1)+n-1$

➔ Time cost:

- Best case: $7(n-1)$
- Worst case: $5(n-1)+2n(n-1)+n-1$

```
void insertionSort(int a[], int n) {  
    int k, pos, temp;  
    for (k=1; k < n; k++) {  
        temp = a[k];  
        pos = k;  
        while ((pos > 0) && (a[pos-1] > temp))  
        {  
            a[pos] = a[pos-1];  
            pos = pos - 1; }  
        a[pos] = temp;  
    } }
```

Các thuật toán sắp xếp

1. Sắp xếp chèn (Insertion sort)
- 2. Sắp xếp chọn (Selection sort)**
3. Sắp xếp nổi bọt (Bubble sort)
4. Sắp xếp trộn (Merge sort)
5. Sắp xếp nhanh (Quick sort)
6. Sắp xếp vun đống (Heap sort)

2. Sắp xếp chọn (Selection sort)

- Mảng cần sắp xếp được chia làm 2 phần: *sorted* và *unsorted*
 - Những phần tử thuộc phần *sorted*: đã được sắp xếp
 - Những phần tử thuộc phần *unsorted*: chưa được sắp xếp
- Mỗi bước lặp: tìm phần tử nhỏ nhất thuộc phần *unsorted*, rồi đổi vị trí phần tử nhỏ nhất này với phần tử đầu tiên của phần *unsorted* ➔ Sau mỗi bước lặp (sau mỗi thao tác chọn và hoán đổi): vách ngăn hai phần *sorted* và *unsorted* dịch chuyển 1 vị trí sang phải, tức là số lượng phần tử thuộc phần *sorted* tăng lên 1, và thuộc phần *unsorted* giảm đi 1.
- Mảng gồm n phần tử sẽ cần $n-1$ bước lặp để thực hiện xong sắp xếp.

Sorted	Unsorted	
	23 78 45 8 32 56	Original array
	8 78 45 23 32 56	After iteration 1
	8 23 45 78 32 56	After iteration 2
	8 23 32 78 45 56	After iteration 3
	8 23 32 45 78 56	After iteration 4
	8 23 32 45 56 78	After iteration 5

Tại bước lặp i : tìm phần tử nhỏ nhất của phần *unsorted* (tức là các phần tử từ $a[i+1] \dots a[n-1]$), rồi chuyển nó lên vị trí thứ i của mảng

2. Sắp xếp chọn (Selection sort)

- Thuật toán
 - Tìm phần tử nhỏ nhất đưa vào vị trí 1
 - Tìm phần tử nhỏ tiếp theo đưa vào vị trí 2
 - Tìm phần tử nhỏ tiếp theo đưa vào vị trí 3
 - ...

```
void selectionSort(int a[], int n){  
    int i, j, index_min;  
    for (i = 0; i < n-1; i++) {  
        index_min = i;  
        //Tìm phần tử nhỏ nhất từ a[i+1] đến phần tử cuối cùng trong mảng  
        for (j = i+1; j < n; j++)  
            if (a[j] < a[index_min]) index_min = j;  
        //đưa phần tử a[index_min] vào vị trí thứ i:  
        swap(&a[i], &a[index_min]);  
    }  
}
```

i=0	1	2	3	4	5	6
42	13	13	13	13	13	13
20	20	14	14	14	14	14
17	17	17	15	15	15	15
13	42	42	42	17	17	17
28	28	28	28	28	20	20
14	14	20	20	20	28	23
23	23	23	23	23	23	28
15	15	15	17	42	42	42

```
void swap(int *a,int *b)  
{  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Đánh giá độ phức tạp tính toán của Selection sort: $O(n^2)$

- Thuật toán
 - Tìm phần tử nhỏ nhất đưa vào vị trí 1
 - Tìm phần tử nhỏ tiếp theo đưa vào vị trí 2
 - Tìm phần tử nhỏ tiếp theo đưa vào vị trí 3
 - ...

	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	20	14	14	14	14	14	14
17	17	17	15	15	15	15	15
13	42	42	42	17	17	17	17
28	28	28	28	28	20	20	20
14	14	20	20	20	28	23	23
23	23	23	23	23	23	28	28
15	15	15	17	42	42	42	42

Đánh giá độ phức tạp tính toán:

- **Best case:** 0 phép đổi chỗ, $n^2/2$ phép so sánh.
- **Worst case:** $n - 1$ phép đổi chỗ, $n^2/2$ phép so sánh.
- **Average case:** $O(n)$ phép đổi chỗ, $n^2/2$ phép so sánh.
- Ưu điểm nổi bật của sắp xếp chọn là số phép đổi chỗ là ít. Điều này là có ý nghĩa nếu như việc đổi chỗ là tốn kém.

Đánh giá độ phức tạp

```
void selectionSort(int a[], int n){  
    int i, j, index_min;  
    for (i = 0; i < n-1; i++) {  
        index_min = i;  
        //Tìm phần tử nhỏ nhất của phần unsorted: từ a[i+1] đến phần tử cuối cùng trong mảng  
        for (j = i+1; j < n; j++)  
            if (a[j] < a[index_min]) index_min = j;  
        //đưa phần tử a[index_min] vào vị trí thứ i:  
        swap(&a[i], &a[index_min]);  
    }  
}
```

```
void swap(int *a, int *b)  
{  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Hàm selectionSort: vòng lặp for ngoài thực hiện $n-1$ times.

Ta gọi hàm swap 1 lần tại mỗi bước lặp.

- ➔ Tổng phép hoán đổi (swap): $n-1$
- ➔ Tổng phép gán: $3*(n-1)$ vì mỗi phép swap có 3 phép đổi chỗ

Vòng lặp for trong thực hiện số lần = kích thước phần unsorted - 1 (từ $n-1$ tới 1), và mỗi vòng lặp thực hiện phép so sánh 1 lần, do đó tổng cộng:

- ➔ # phép so sánh = $(n-1) + (n-2) + \dots + 2 + 1 = n*(n-1)/2$
- ➔ Vậy, độ phức tạp của Selection sort là $O(n^2)$

Kết luận: Đánh giá độ phức tạp của Selection sort: $O(n^2)$

- Sắp xếp chọn là tại chỗ và ổn định (In place and Stable)
- Thời gian tính: $O(n^2)$ cho cả 3 trường hợp best/worse/average. Tức là thuật toán không phụ thuộc vào đặc tính của dữ liệu đầu vào.
- Mặc dù thuật toán sắp xếp chèn cần $O(n^2)$ phép so sánh, nhưng chỉ thực hiện $O(n)$ phép đổi chỗ.
 - Nên chọn sắp xếp chọn nếu việc đổi chỗ là tốn kém, còn phép so sánh ít tốn kém (trường hợp dữ liệu đầu vào: short keys, long records).

Các thuật toán sắp xếp

1. Sắp xếp chèn (Insertion sort)
2. Sắp xếp chọn (Selection sort)
- 3. Sắp xếp nổi bọt (Bubble sort)**
4. Sắp xếp trộn (Merge sort)
5. Sắp xếp nhanh (Quick sort)
6. Sắp xếp vun đống (Heap sort)

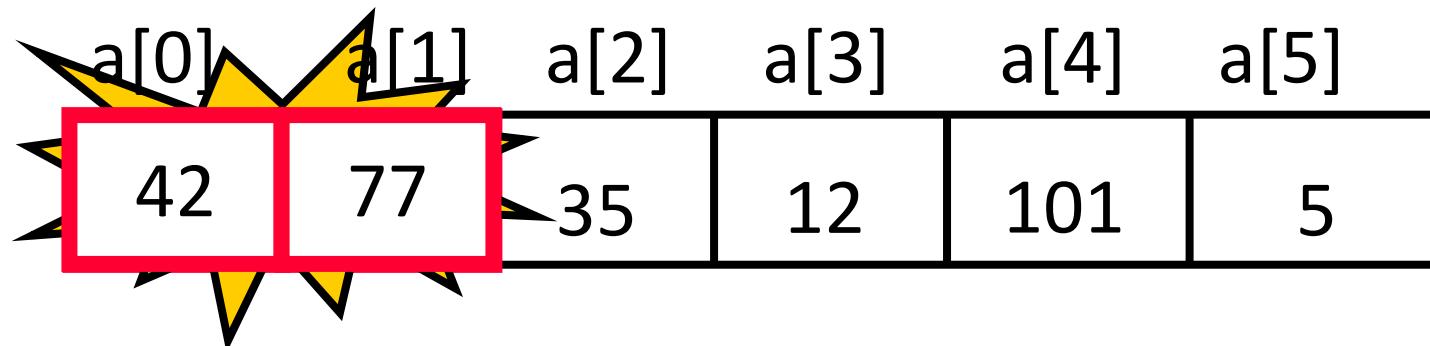
"Bubbling Up" phần tử lớn nhất

- Duyệt qua lần lượt các phần tử của mảng:
 - Từ trái sang phải (từ đầu dãy về cuối dãy)
 - “Bubble” phần tử có giá trị lớn nhất về cuối mảng bằng cách thực hiện các cặp lệnh (so sánh, hoán đổi)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
77	42	35	12	101	5

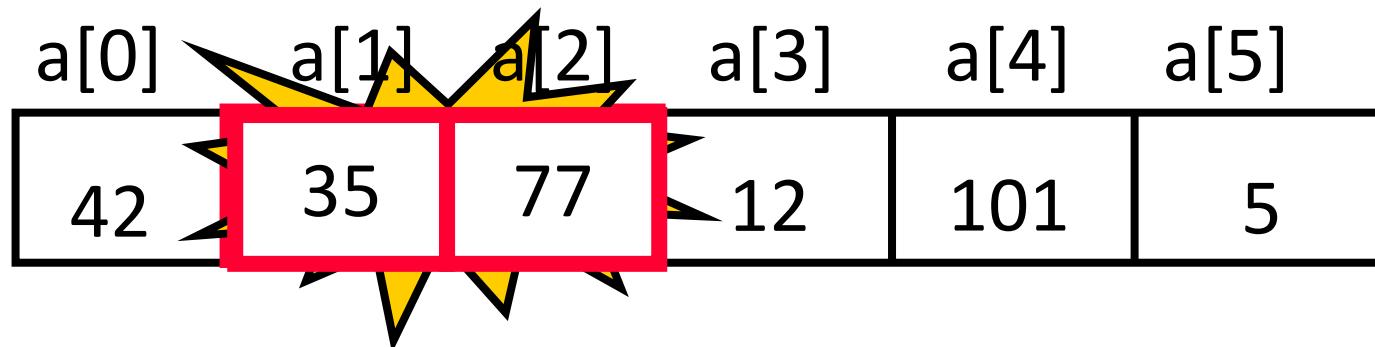
"Bubbling Up" phần tử lớn nhất

- Duyệt qua lần lượt các phần tử của mảng:
 - Từ trái sang phải (từ đầu dãy về cuối dãy)
 - “Bubble” phần tử có giá trị lớn nhất về cuối mảng bằng cách thực hiện các cặp lệnh (so sánh, hoán đổi)



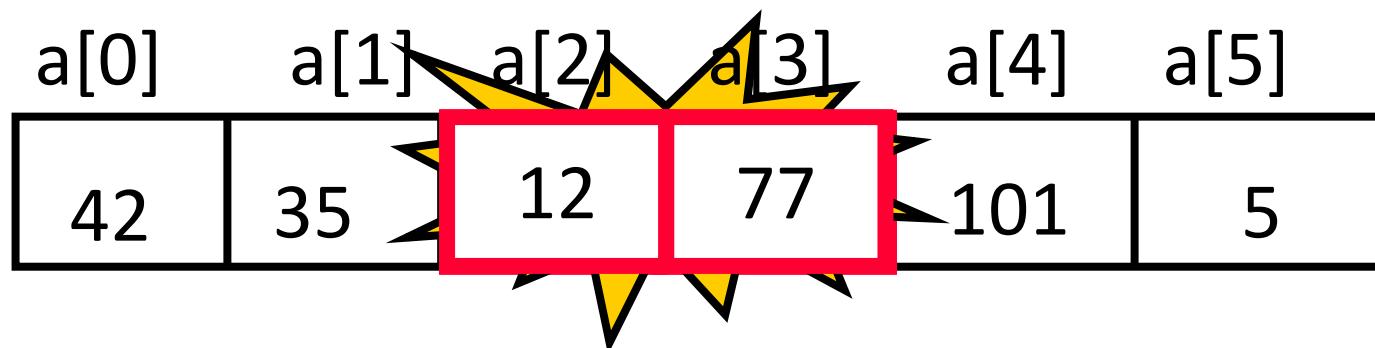
"Bubbling Up" phần tử lớn nhất

- Duyệt qua lần lượt các phần tử của mảng:
 - Từ trái sang phải (từ đầu dãy về cuối dãy)
 - “Bubble” phần tử có giá trị lớn nhất về cuối mảng bằng cách thực hiện các cặp lệnh (so sánh, hoán đổi)



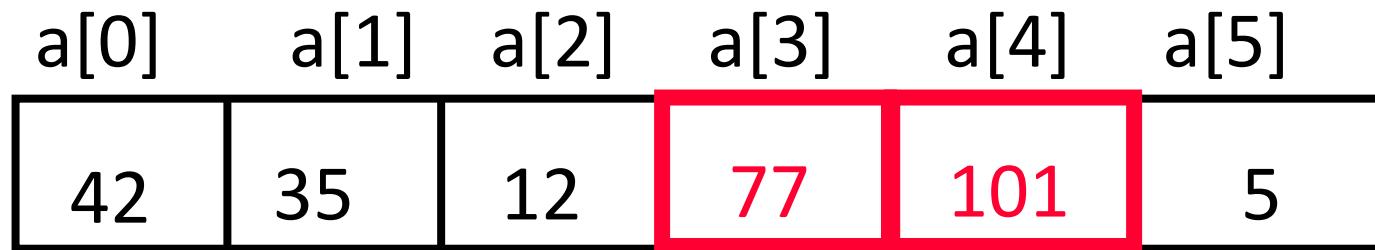
"Bubbling Up" phần tử lớn nhất

- Duyệt qua lần lượt các phần tử của mảng:
 - Từ trái sang phải (từ đầu dãy về cuối dãy)
 - “Bubble” phần tử có giá trị lớn nhất về cuối mảng bằng cách thực hiện các cặp lệnh (so sánh, hoán đổi)



"Bubbling Up" phần tử lớn nhất

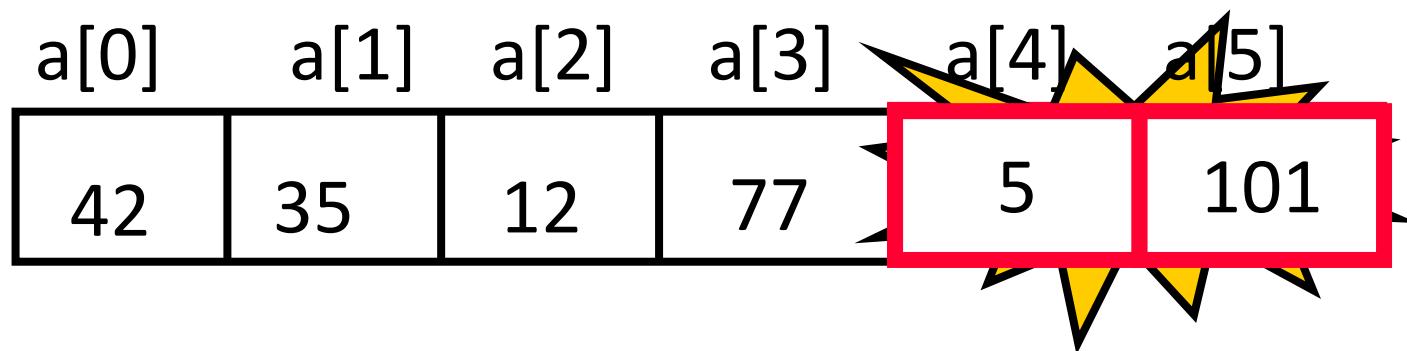
- Duyệt qua lần lượt các phần tử của mảng:
 - Từ trái sang phải (từ đầu dãy về cuối dãy)
 - “Bubble” phần tử có giá trị lớn nhất về cuối mảng bằng cách thực hiện các cặp lệnh (so sánh, hoán đổi)



Không cần swap

"Bubbling Up" phần tử lớn nhất

- Duyệt qua lần lượt các phần tử của mảng:
 - Từ trái sang phải (từ đầu dãy về cuối dãy)
 - “Bubble” phần tử có giá trị lớn nhất về cuối mảng bằng cách thực hiện các cặp lệnh (so sánh, hoán đổi)



"Bubbling Up" the Largest Element

- Duyệt qua lần lượt các phần tử của mảng:
 - Từ trái sang phải (từ đầu dãy về cuối dãy)
 - “Bubble” phần tử có giá trị lớn nhất về cuối mảng bằng cách thực hiện các cặp lệnh (so sánh, hoán đổi)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
42	35	12	77	5	101

Phần tử có giá trị lớn nhất đã nằm đúng vị trí

Chú ý

- Nhận thấy rằng mới chỉ có phần tử lớn nhất nằm đúng vị trí của nó trên mảng
- Các phần tử khác vẫn chưa nằm đúng vị trí
- Vì vậy ta cần **lặp lại tiến trình vừa thực hiện**

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
42	35	12	77	5	101

Phần tử có giá trị lớn nhất đã nằm đúng vị trí
của nó trong mảng

Cần lặp lại tiến trình “Bubble Up” bao nhiêu lần?

- Nếu mảng có n phần tử...
- Mỗi tiến trình “bubble up” ta chỉ di chuyển được 1 phần tử về đúng vị trí của nó trong mảng...
- Vậy ta cần **lặp lại tiến trình “bubble up” $n - 1$ lần** để đảm bảo tất cả n phần tử của mảng đã cho đã về đúng vị trí cần sắp xếp.

“Bubbling” tất cả các phần tử

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	
77	42	35	12	101	5	Mảng ban đầu
42	35	12	77	5	101	1 st iteration
35	12	42	5	77	101	2 nd iteration
12	35	5	42	77	101	3 rd iteration
12	5	35	42	77	101	4 th iteration
5	12	35	42	77	101	5 th iteration

$n - 1$

Ví dụ: Sắp xếp mảng: 9, 6, 2, 12, 11, 9, 3, 7

9, 6, 2, 12, 11, 9, 3, 7

Sắp xếp nổi bọt so sánh cặp các phần tử lần lượt từ trái sang phải, và đổi vị trí của chúng nếu cần thiết. Tại lần đầu tiên: phần tử thứ 1 được so sánh với phần tử thứ 2, vì nó lớn hơn, nên hai phần tử này được hoán đổi vị trí cho nhau.

Bubble Sort

9, 6, 2, 12, 11, 9, 3, 7
6, 9, 2, 12, 11, 9, 3, 7

Lúc này, cặp tiếp theo tiếp tục được so sánh. Vì $9 > 2$, nên hai phần tử này được đổi chỗ cho nhau.

Bubble Sort

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

Lần thứ 3: vì $9 < 12$, nên không cần đổi chỗ 2 phần tử này. Ta tiếp tục di chuyển đến cặp tiếp theo.

Bubble Sort

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

12 > 11 nên hai phần tử này đổi chỗ cho nhau.

Bubble Sort

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

Vì $12 > 9$, nên hai phần tử này đổi chỗ cho nhau

Bubble Sort

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

6, 2, 9, 11, 9, 12, 3, 7

Vì $12 > 3$ nên hai phần tử này đổi chỗ cho nhau.

Bubble Sort

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6, 2, 9, 11, 12, 9, 3, 7

6, 2, 9, 11, 9, 12, 3, 7

6, 2, 9, 11, 9, 3, 12, 7

Vì 12 > 7 nên hai phần tử này đổi chỗ cho nhau.

Bubble Sort

9, 6, 2, 12, 11, 9, 3, 7

6, 9, 2, 12, 11, 9, 3, 7

6, 2, 9, 12, 11, 9, 3, 7

6 2 0 12 11 0 3 7

Đã duyệt qua hết các phần tử của mảng ở lần lặp thứ 1. Lúc này số 12 nằm ở vị trí cuối cùng của dãy vì nó là phần tử có giá trị lớn nhất trong dãy.

Ta lại tiếp tục lần lặp thứ 2: duyệt mảng từ trái sang phải.

6, 2, 9, 11, 9, 12, 3, 7

6, 2, 9, 11, 9, 3, 12, 7

6, 2, 9, 11, 9, 3, 7, 12

Ví dụ: Sắp xếp mảng: 9, 6, 2, 12, 11, 9, 3, 7

1st iteration:

6, 2, 9, 11, 9, 3, 7, 12

2nd iteration:

2, 6, 9, 9, 3, 7, 11, 12

Chú ý: ở bước lặp thứ 2 này, ta không cần thực hiện so sánh hai phần tử cuối cùng vì phần tử cuối cùng (giá trị 12) đã nằm đúng vị trí của nó trong mảng. Như vậy, bước lặp thứ 2 này chỉ cần thực hiện 6 phép so sánh cặp 2 phần tử.

Bubble Sort

1st iteration:

6, 2, 9, 11, 9, 3, 7, 12

2nd iteration:

2, 6, 9, 9, 3, 7, 11, 12

3rd iteration:

2, 6, 9, 3, 7, 9, 11, 12

Ở bước lặp thứ 3: vì số 11 và 12 đã nằm đúng vị trí của nó trong mảng. Do đó bước lặp này chỉ cần thực hiện 5 phép so sánh cặp 2 phần tử.

Bubble Sort

1st iteration:

6, 2, 9, 11, 9, 3, 7, 12

2nd iteration:

2, 6, 9, 9, 3, 7, 11, 12

3rd iteration:

2, 6, 9, 3, 7, 9, 11, 12

4th iteration:

2, 6, 3, 7, 9, 9, 11, 12

Mỗi bước lặp, số phép so sánh cần thực hiện lại giảm đi một.

Ở bước lặp thứ 4: ta chỉ cần thực hiện 4 phép so sánh cặp 2 phần tử.

Bubble Sort

1st iteration:

6, 2, 9, 11, 9, 3, 7, 12

2nd iteration:

2, 6, 9, 9, 3, 7, 11, 12

3rd iteration:

2, 6, 9, 3, 7, 9, 11, 12

4th iteration:

2, 6, 3, 7, 9, 9, 11, 12

5th iteration:

2, 3, 6, 7, 9, 9, 11, 12

Mảng lúc này đã được sắp xếp, nhưng thuật toán không biết điều này và vẫn tiếp tục thực hiện cho đến khi tại 1 bước lặp nào đó không có phép hoán đổi nào thì thuật toán mới dừng.

Bubble Sort

1st iteration: 6, 2, 9, 11, 9, 3, 7, 12

2nd iteration: 2, 6, 9, 9, 3, 7, 11, 12

3rd iteration: 2, 6, 9, 3, 7, 9, 11, 12

4th iteration: 2, 6, 3, 7, 9, 9, 11, 12

5th iteration: 2, 3, 6, 7, 9, 9, 11, 12

6th iteration: 2, 3, 6, 7, 9, 9, 11, 12

Tại bước lặp thứ 6: không có phép hoán đổi nào được thực hiện, thuật toán “biết” là mảng đã được sắp xếp; vì vậy thuật toán kết thúc → không cần phải thực hiện đến bước cuối cùng ($n-1=8-1 = 7$) → tiết kiệm thời gian

Quiz: Sắp xếp nổi bọt

- Phần tử nào sẽ nằm đúng vị trí của nó trong mảng sau bước lặp đầu tiên của thuật toán?

Trả lời: Phần tử cuối cùng phải có giá trị lớn nhất.

- Số phép so sánh cần phải thực hiện thay đổi thế nào sau mỗi bước lặp của thuật toán?

Trả lời: Sau mỗi bước lặp, số phép so sánh cần thực hiện giảm đi 1.

- Khi nào thì thuật toán “biết” là mảng đã được sắp xếp xong?

Trả lời: Khi tại một bước lặp nào đó, không có phép đổi chỗ nào được thực hiện.

- Số lượng tối đa số lần phải thực hiện phép so sánh cho một mảng gồm 10 phần tử là bao nhiêu?

Trả lời: 9 phép so sánh ở bước lặp 1, rồi lần lượt đến 8, 7, 6, 5, 4, 3, 2, 1 ở các bước lặp tiếp theo → tổng cộng = 45.

3. Bubble Sort: cài đặt

Dùng thuật toán sắp xếp nổi bọt để sắp xếp mảng gồm n phần tử: $a[0], a[1], \dots, a[n-1]$:

- Thuật toán kết thúc khi nó nhận ra mảng đã được sắp xếp xong (@ bước lặp mà tại đó không có phép đổi chỗ nào được thực hiện)
- Số lượng bước lặp tối đa cần thực hiện = $n - 1$
 - Mỗi bước lặp i ($i=1, 2, \dots, n-1$): “bubble” phần tử có giá trị lớn nhất trong các phần tử từ $a[0] \dots a[n-i]$ về vị trí $(n-i)$

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Original array
77	42	35	12	101	5	
42	35	12	77	5	101	1 st iteration
35	12	42	5	77	101	2 nd iteration
12	35	5	42	77	101	3 rd iteration
12	5	35	42	77	101	4 th iteration
5	12	35	42	77	101	5 th iteration

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$	Original array
9	6	2	12	11	9	3	7	
6	2	9	11	9	3	7	12	1 st iteration
2	6	9	9	3	7	11	12	2 nd iteration
2	6	9	3	7	9	11	12	3 rd iteration
2	6	3	7	9	9	11	12	4 th iteration
2	3	6	7	9	9	11	12	5 th iteration
2	3	6	7	9	9	11	12	6 th iteration

Stop @ iteration 6th because the algorithm identifies that there does not occur any exchange

```
void bubbleSort(int a[], int n)
{
    bool sorted = false; /* sử dụng để ghi nhận bước lặp không có
    phép hoán đổi nào được thực hiện, tức là mảng đã sắp xếp xong*/
    for (int i = 1; i <= n-1; i++)
        if (sorted == false)
    {
        sorted = true;
        for (int j = 0; j <= n-i-1; j++)
            if (a[j] < a[j+1])
            {
                swap(&a[j], &a[j+1]);
                sorted = false; // đã thực hiện hoán đổi
            }
    }
}
```

3. Bubble Sort: cài đặt

```
void bubbleSort(int a[], int n)
{
    bool sorted = false; /*sử dụng để ghi nhận bước lặp không có
phép hoán đổi nào được thực hiện, tức là mảng đã sắp xếp xong*/
    for (int i = 1; i <= n-1; i++)
        if (sorted == false)
        {
            sorted = true;
            for (int j= 0; j <= n-i-1; j++)
                if (a[j] < a[j+1])
                {
                    swap(&a[j], &a[j+1]);
                    sorted = false; // đã thực hiện hoán đổi
                }
        }
}
```

3. Bubble Sort – Phân tích độ phức tạp

- **Best-case: O(n)**
 - Dãy đầu vào đã được sắp xếp.
 - Số phép toán đổi chỗ: 0 $\rightarrow O(1)$
 - Số phép toán so sánh: $(n-1) \rightarrow O(n)$
- **Worst-case: O(n^2)**
 - Dãy đầu vào có thứ tự ngược với thứ tự cần sắp xếp: vòng for ngoài thực hiện $n-1$ lần,
 - Số lượng phép đổi chỗ: $3*(1+2+...+n-1) = 3 * n*(n-1)/2 \rightarrow O(n^2)$
 - Số lượng phép so sánh: $(1+2+...+n-1) = n*(n-1)/2 \rightarrow O(n^2)$
- Average-case: **O(n^2)**
 - Cần phải xét tất cả các thuộc tính của dãy dữ liệu đầu vào.

Vậy, độ phức tạp của Bubble Sort là **O(n^2)**

```
void bubbleSort(int a[], int n)
{
    bool sorted = false; /*use to identify the iteration having no
                           exchanges, it means the array is sorted already */
    for (int i = 1; i <= n-1; i++)
        if (sorted == false)
        {
            sorted = true;
            for (int j= 0; j <= n-i-1; j++)
                if (a[j] < a[j+1])
                {
                    swap(&a[j],&a[j+1]);
                    sorted = false; // signal exchange
                }
        }
}
```

Nội dung

1. Sắp xếp chèn (Insertion sort)
2. Sắp xếp chọn (Selection sort)
3. Sắp xếp nổi bọt (Bubble sort)
4. Sắp xếp trộn (Merge sort)
5. Sắp xếp nhanh (Quick sort)
6. Sắp xếp vun đống (Heap sort)

Divide and conquer

Chia để trị (Divide and Conquer)

3 bước thực hiện:

- Divide: Phân rã bài toán đã cho thành bài toán cùng dạng với kích thước nhỏ hơn (gọi là bài toán con)
- Conquer: Giải các bài toán con một cách độc lập
- Combine: Tổng hợp lời giải của các bài toán con để thu được lời giải của bài toán ban đầu

Idea 1: Chia dãy đã cho thành hai nửa, sắp xếp nửa trái và nửa phải một cách đệ quy, sau đó **trộn (merge)** hai nửa lại để thu được dãy hoàn chỉnh đã sắp xếp → Mergesort

Idea 2 : Chia dãy đã cho thành hai tập: tập chứa các phần tử “nhỏ” và tập chứa các phần tử “lớn”, sau đó sắp xếp đệ quy hai tập này → Quicksort

Nội dung

1. Sắp xếp chèn (Insertion sort)
 2. Sắp xếp chọn (Selection sort)
 3. Sắp xếp nổi bọt (Bubble sort)
 4. **Sắp xếp trộn (Merge sort)**
 5. Sắp xếp nhanh (Quick sort)
 6. Sắp xếp vun đống (Heap sort)
- Devide and conquer

4. Sắp xếp trộn (Merge sort)

Bài toán: Cần sắp xếp mảng A[1 .. n]:

- **Chia (Divide)**
 - Chia dãy gồm n phần tử cần sắp xếp ra thành 2 dãy, mỗi dãy có $n/2$ phần tử
- **Trị (Conquer)**
 - Sắp xếp mỗi dãy con một cách đệ qui sử dụng **sắp xếp trộn**
 - Khi dãy chỉ còn một phần tử thì trả lại phần tử này
- **Tổ hợp (Combine)**
 - Trộn (Merge) hai dãy con được sắp xếp để thu được dãy được sắp xếp gồm tất cả các phần tử của cả hai dãy con

4. Merge Sort

MS(A , p , r)

if $p < r$ then

$q \leftarrow \lfloor(p + r)/2\rfloor$

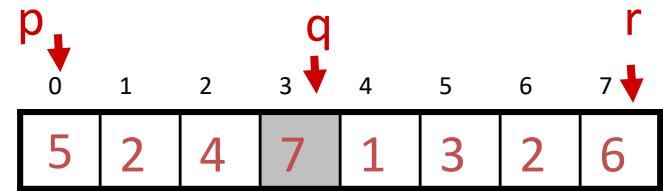
MS(A , p , q)

MS(A , $q + 1$, r)

MERGE(A , p , q , r)

endif

Lệnh gọi thực hiện thuật toán cho mảng S gồm n phần tử: MS(S , 0, $n-1$);



▷ Kiểm tra điều kiện neo

▷ Chia (Divide)

▷ Trị (Conquer)

▷ Trị (Conquer)

▷ Tổng hợp (Combine)

Ví dụ: Merge sort

MS(A , p , r)

if $p < r$ then

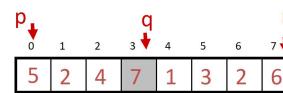
$q \leftarrow \lfloor (p + r) / 2 \rfloor$

MS(A , p , q)

MS(A , $q + 1$, r)

MERGE(A , p , q , r)

endif



▷ Kiểm tra điều kiện neo

▷ Chia (Divide)

▷ Trị (Conquer)

▷ Trị (Conquer)

▷ Tổ hợp (Combine)

Divide

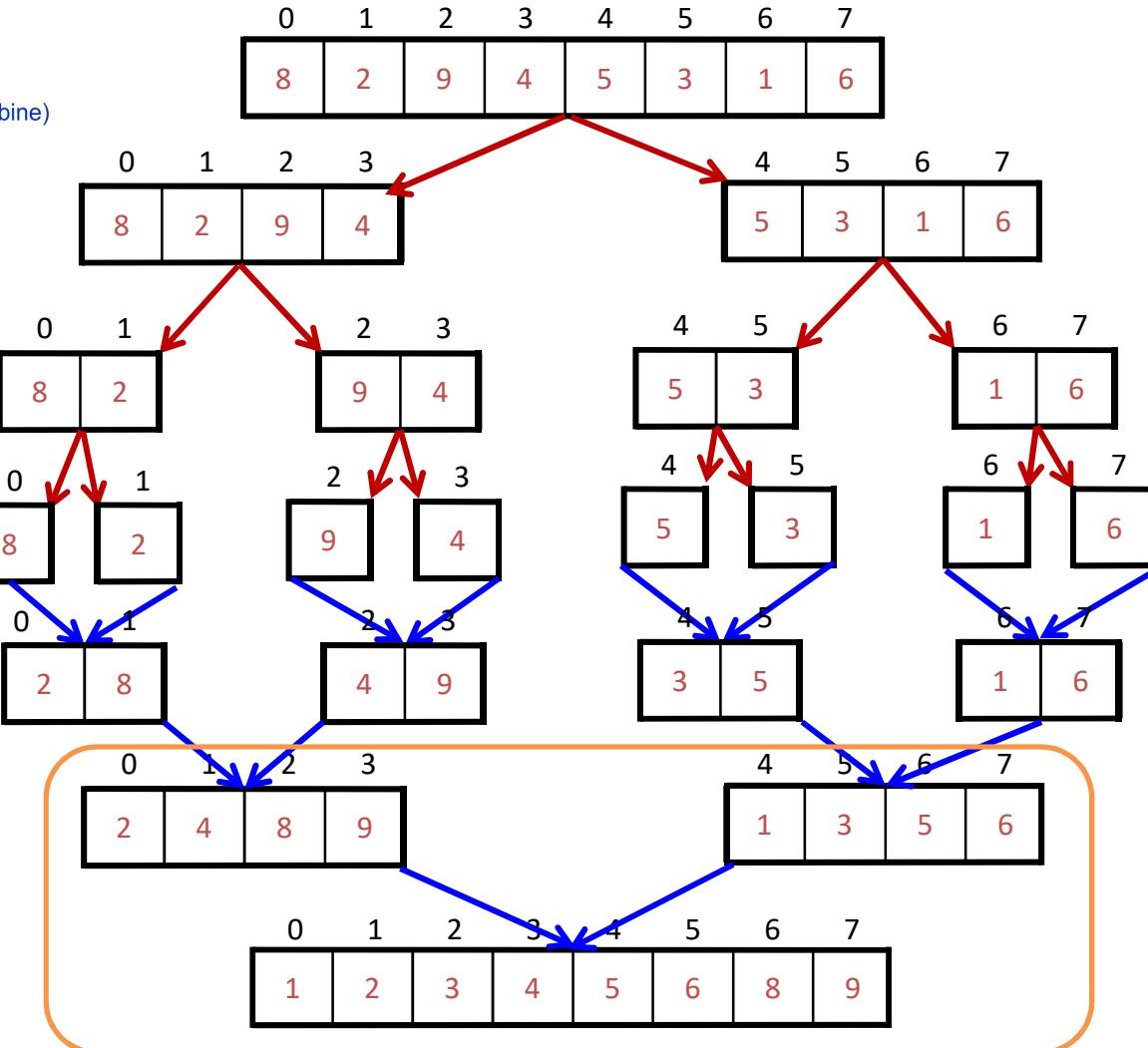
Divide

1 phần tử

Merge

Merge

Kết quả:



$MS(A, p, r)$

if $p < r$ then

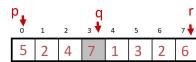
$q \leftarrow \lfloor (p + r)/2 \rfloor$

$MS(A, p, q)$

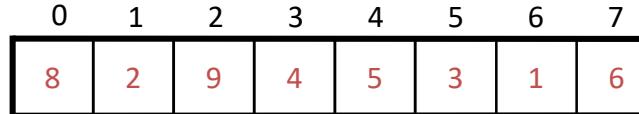
$MS(A, q + 1, r)$

MERGE(A, p, q, r)

endif



Ví dụ: Merge sort



$MS(A, 0, 7);$

Divide

$p=0; r=3; q = 1$
 $MS(A, 0, 1); MS(A, 2, 3);$
MERGE($A, 0, 1, 3$);



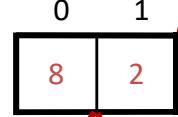
$MS(A, 4, 7)$



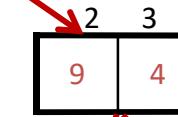
$p=4; r=7; q = 5$
 $MS(A, 4, 5); MS(A, 6, 7);$
MERGE($A, 4, 5, 7$);

Divide

$p=0; r=1; q = 0$
 $MS(A, 0, 0); MS(A, 1, 1);$
MERGE($A, 0, 0, 1$);

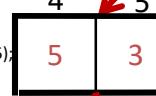


$MS(A, 0, 1)$



$MS(A, 2, 3)$

$p=2; r=3; q = 2$
 $MS(A, 2, 2); MS(A, 3, 3);$
MERGE($A, 2, 2, 3$);



$MS(A, 4, 5)$



$p=6; r=7; q = 6$
 $MS(A, 6, 6); MS(A, 7, 7);$
MERGE($A, 6, 6, 7$);

1 phần tử

$p=0; r=0;$



$p=1; r=1$



$p=2; r=2;$



$p=3; r=3$



$p=4; r=4$



$p=5; r=5$



$p=6; r=6$



$p=7; r=7$



Merge

MERGE($A, 0, 0, 1$)



$0 \quad 1$

MERGE($A, 2, 2, 3$)



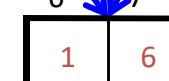
$2 \quad 3$

MERGE($A, 4, 4, 5$)



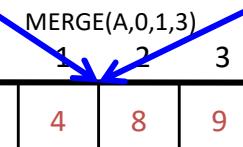
$4 \quad 5$

MERGE($A, 6, 6, 7$)



$6 \quad 7$

Merge



$0 \quad 1 \quad 2 \quad 3$

MERGE($A, 0, 3, 7$)



$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7$

Result:

4. Merge Sort

MS(A , p , r)

if $p < r$ then

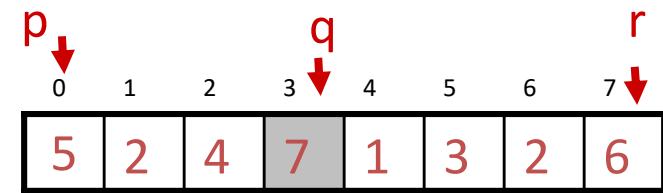
$q \leftarrow \lfloor(p + r)/2\rfloor$

$MS(A, p, q)$

$MS(A, q + 1, r)$

MERGE(A , p , q , r)

endif



▷ Kiểm tra điều kiện neo

▷ Chia (Divide)

▷ Trị (Conquer)

▷ Trị (Conquer)

▷ Tổng hợp (Combine)

Trộn 2 mảng đã sắp xếp theo thứ tự tăng dần $A[p]...A[q]$ và $A[q+1]...A[r]$ thành một mảng được sắp xếp theo thứ tự tăng dần

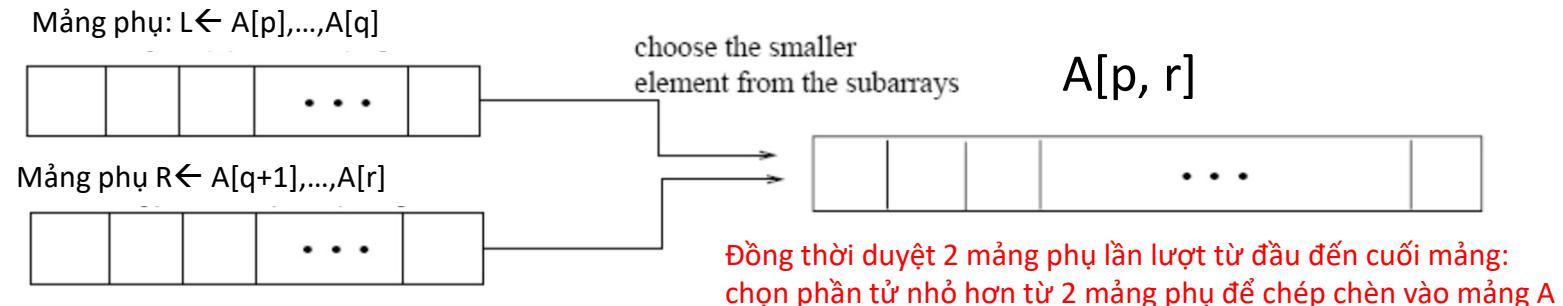
Ý tưởng trộn 2 dãy đã sắp xếp theo thứ tự tăng dần thành 1 dãy sắp xếp cũng theo thứ tự tăng dần

Mảng A gồm 2 nửa đã sắp xếp:

- Dãy 1: gồm các số từ $A[p] \dots A[q]$
- Dãy 2: gồm các số từ $A[q+1] \dots A[r]$

Cần trộn 2 dãy này với nhau để thu được 1 dãy gồm các số được sắp xếp theo thứ tự tăng dần.

Idea 1: dùng 2 mảng phụ L, R có kích thước bằng $\frac{1}{2}$ mảng A



Idea 2: dùng 1 mảng phụ tempA có kích thước bằng mảng A

Đồng thời duyệt qua lần lượt từng phần tử của

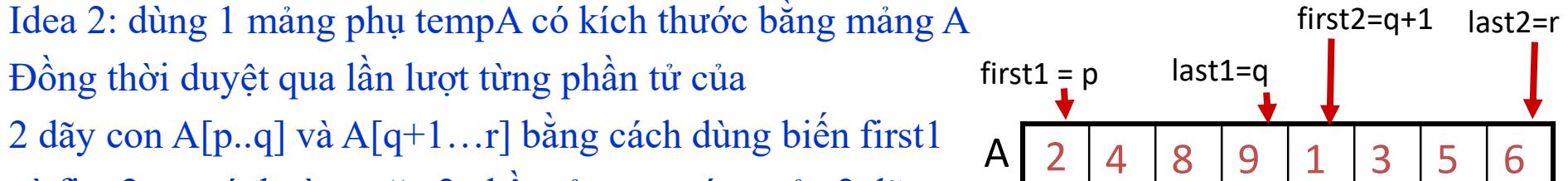
2 dãy con $A[p..q]$ và $A[q+1\dots r]$ bằng cách dùng biến first1

và first2: so sánh từng cặp 2 phần tử tương ứng của 2 dãy con,

chọn phần tử nhỏ hơn để chép vào mảng phụ tempA.

Kết thúc vòng lặp, tất cả các phần tử của 2 dãy con đã được

đọc qua; khi đó mảng tempA chứa tất cả các phần tử của 2 dãy con, nhưng các phần tử đã được sắp xếp. Copy mảng tempA chèn vào mảng A



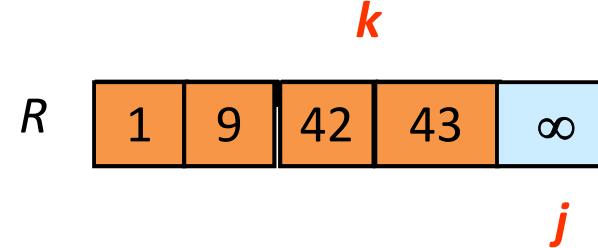
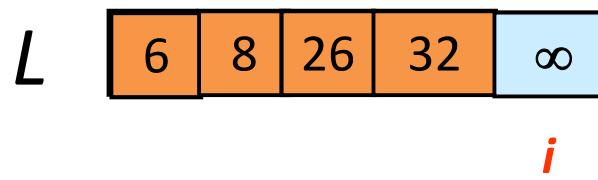
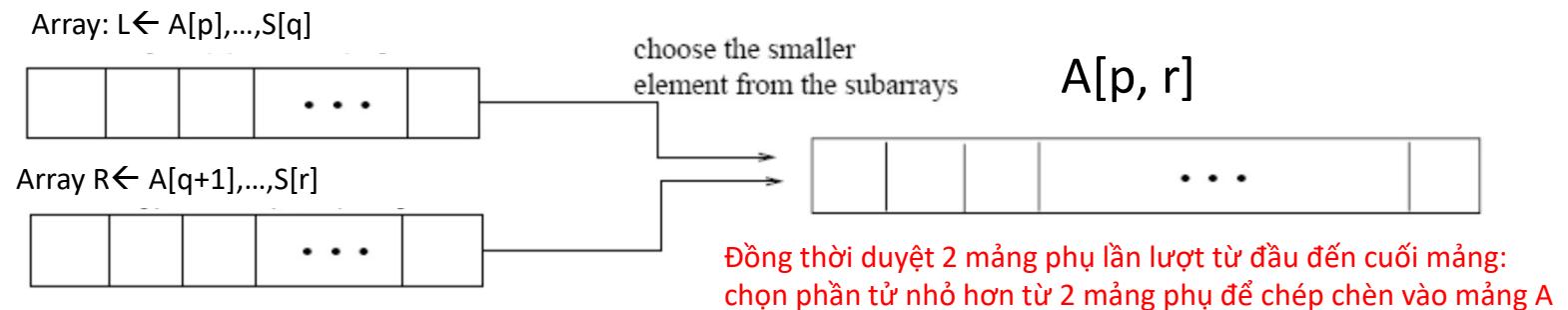
Trộn (Merge) – Minh họa Idea1

Mảng A gồm 2 nửa đã sắp xếp:

- Dãy 1: gồm các số từ $A[p] \dots A[q]$
- Dãy 2: gồm các số từ $A[q+1] \dots A[r]$

Cần trộn 2 dãy này với nhau để thu được 1 dãy gồm các số được sắp xếp theo thứ tự tăng dần.

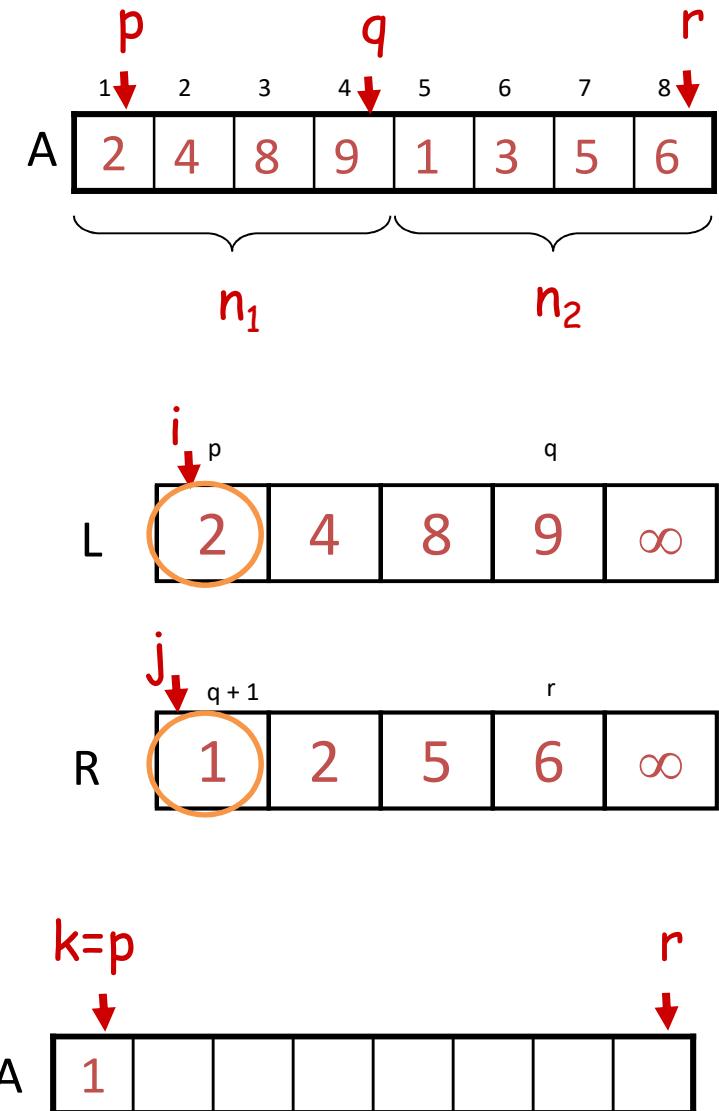
Dùng 2 mảng phụ L, R có kích thước bằng $\frac{1}{2}$ mảng A



Idea 1: Trộn (Merge) - Pseudocode

MERGE(A, p, q, r)

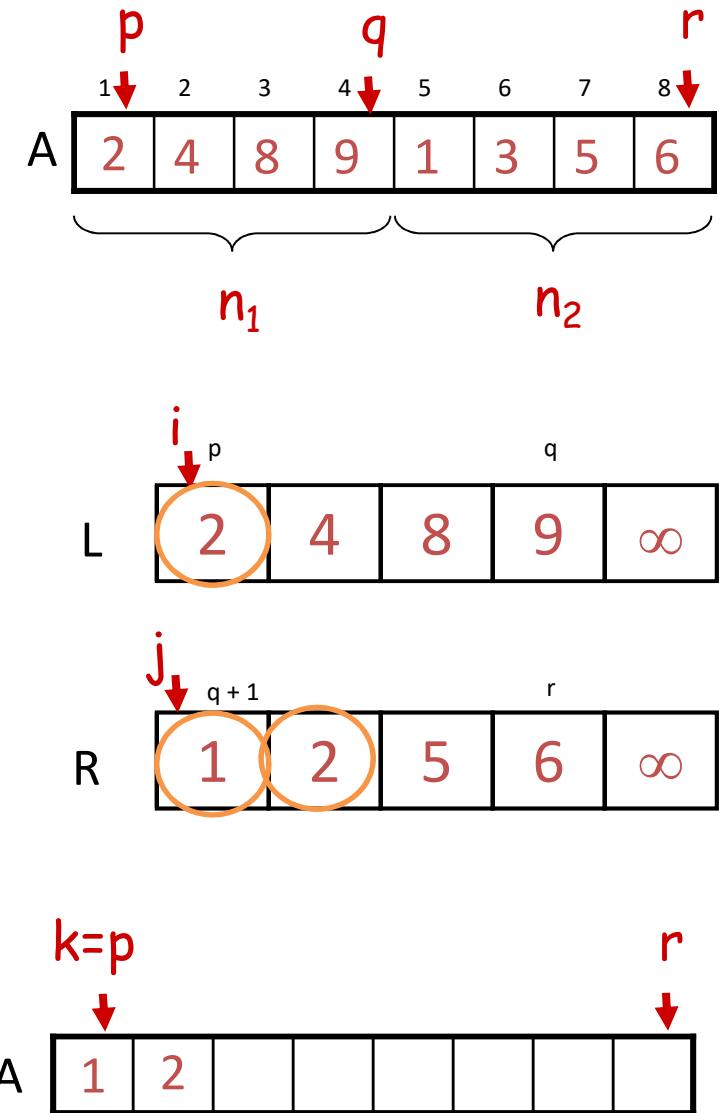
1. Tính n_1 và n_2
2. Sao n_1 phần tử đầu tiên vào $L[1 \dots n_1]$ và n_2 phần tử tiếp theo vào $R[1 \dots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ to r **do**
 6. **if** $L[i] \leq R[j]$
 then $A[k] \leftarrow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



Idea 1: Trộn (Merge) - Pseudocode

MERGE(A, p, q, r)

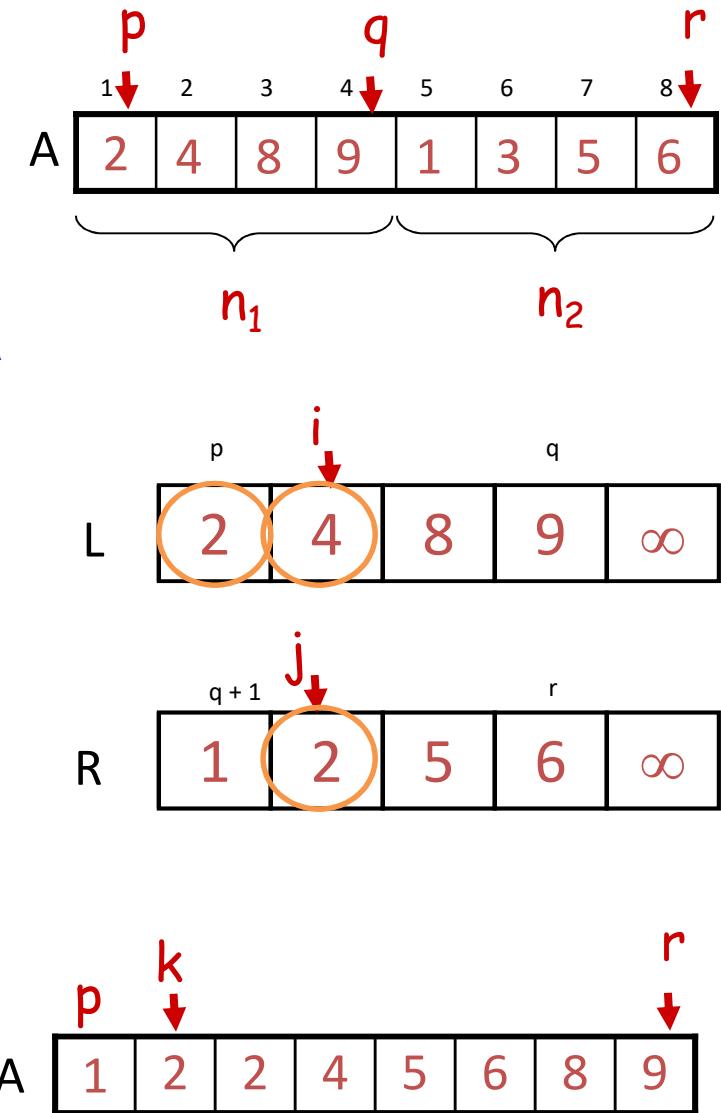
1. Tính n_1 và n_2
2. Sao n_1 phần tử đầu tiên vào $L[1 \dots n_1]$ và n_2 phần tử tiếp theo vào $R[1 \dots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ to r **do**
6. **if** $L[i] \leq R[j]$
7. **then** $A[k]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



Idea 1: Trộn (Merge) - Pseudocode

MERGE(A, p, q, r)

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



Đánh giá độ phức tạp tính toán của Merge sort: $O(n \log n)$

Thời gian tính của thủ tục Merge:

- Khởi tạo (tạo hai mảng con tạm thời L và R):
 - $O(n_1 + n_2) = O(n)$
- Đưa các phần tử vào mảng kết quả (vòng lặp for):
 - n lần lặp, mỗi lần đòi hỏi thời gian hằng số $\Rightarrow O(n)$
- Tổng cộng thời gian của trộn là: $O(n)$

Thời gian tính của sắp xếp trộn Merge Sort:

- Chia:** tính q như là giá trị trung bình của p và r : $O(1)$
- Trị:** giải đệ qui 2 bài toán con, mỗi bài toán kích thước $n/2 \Rightarrow 2T(n/2)$
- Tổ hợp:** TRỘN (MERGE) trên các mảng con cỡ n phần tử đòi hỏi thời gian $O(n)$

$$T(n) = \begin{cases} O(1) & \text{nếu } n=1 \\ 2T(n/2) + O(n) & \text{nếu } n > 1 \end{cases}$$

Suy ra: $T(n) = O(n \log n)$ (CM bằng qui nạp!
hoặc cây đệ quy)

MERGE(A, p, q, r)

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

MERGE-SORT(A, p, r)

if $p < r$

then $q \leftarrow \lfloor (p+r)/2 \rfloor$

MERGE-SORT(A, p, q)

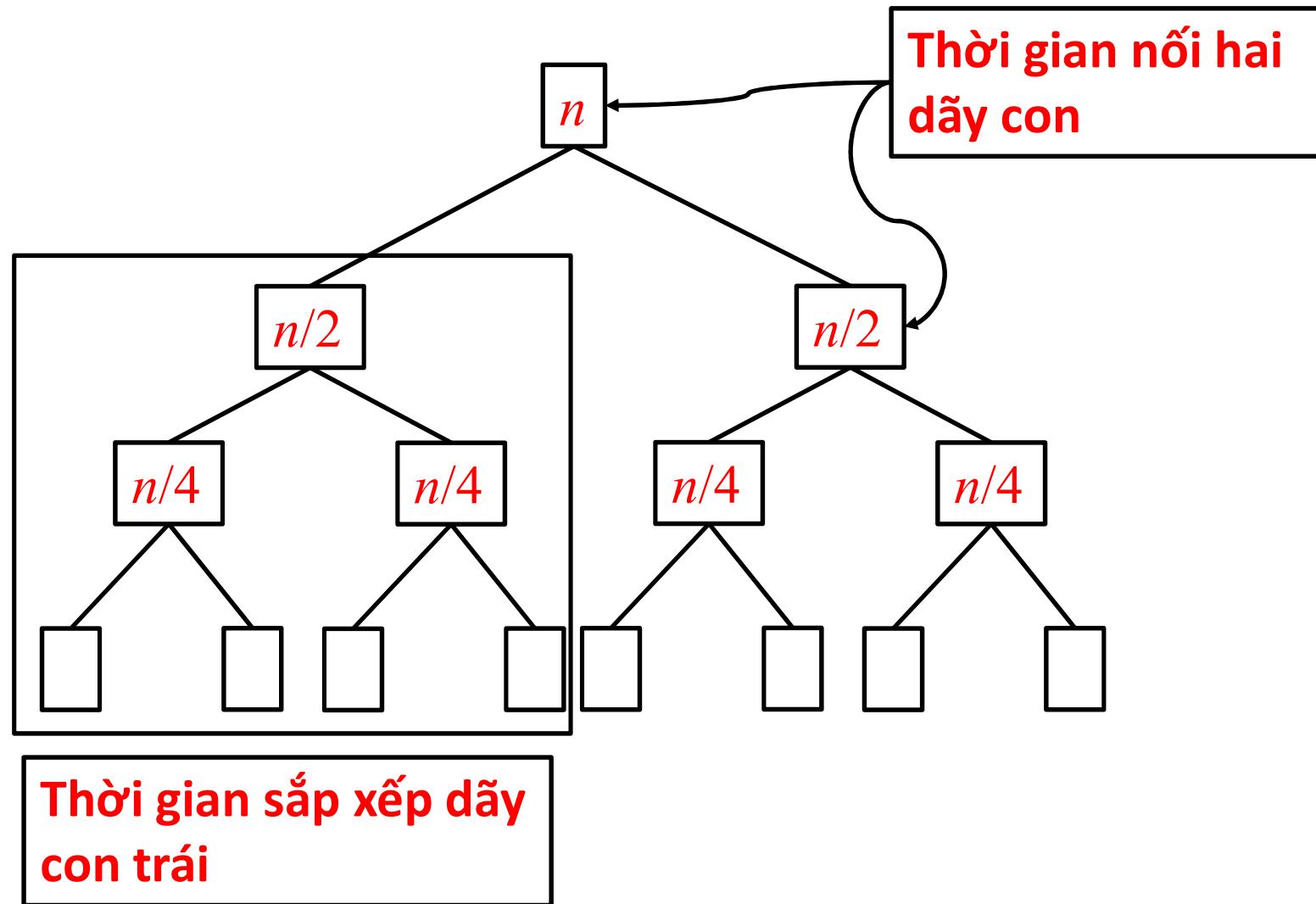
MERGE-SORT($A, q+1, r$)

MERGE(A, p, q, r)

endif

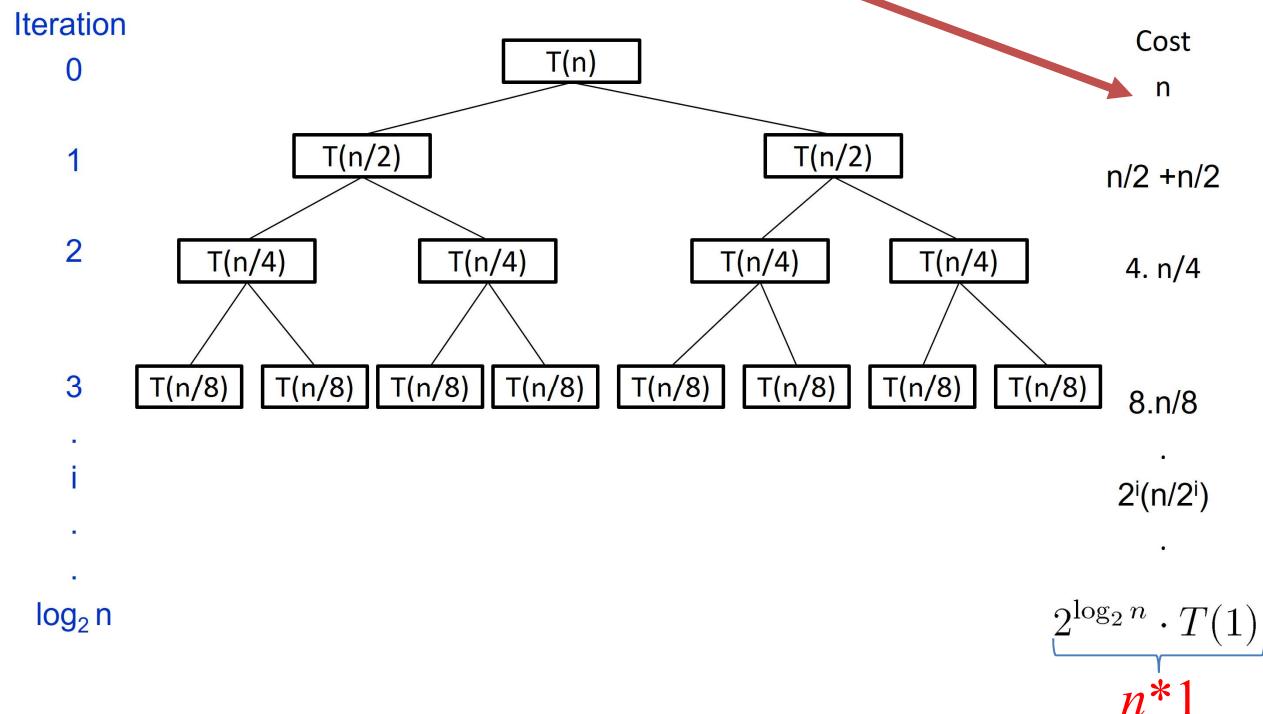
Ví dụ: Giải công thức đệ quy:

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



Ví dụ: Giải công thức đệ quy:

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



- Giá trị hàm $T(n)$ bằng tổng các giá trị tại tất cả các mức:

$$T(n) = \sum_{i=0}^{\log_2 n} 2^i \left(\frac{n}{2^i}\right)$$

- Vì mức cuối (sâu nhất) $\log_2 n$ có giá trị = n , ta có

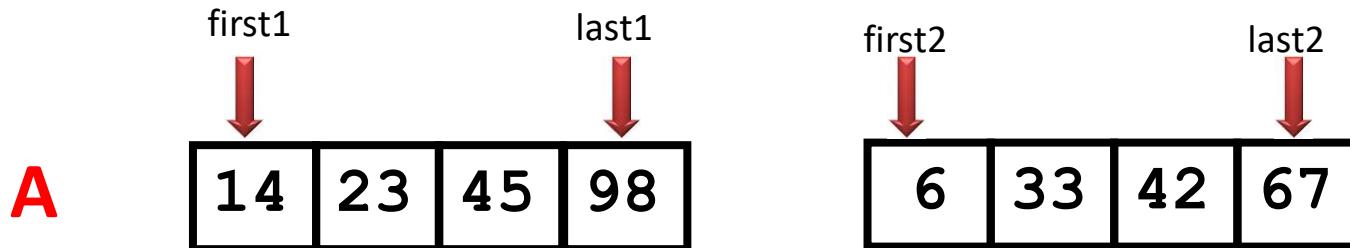
$$T(n) = n + \sum_{i=0}^{\log_2 n - 1} 2^i \frac{n}{2^i} = n(\log n) + n$$

Idea 2: Trộn (Merge)

```
#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge
```

A	14	23	45	98	6	33	42	67
---	----	----	----	----	---	----	----	----



```

#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge

```

Idea 2: Trộn (Merge)

A

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

Merge

```

#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge

```

Idea 2: Trộn (Merge)

A

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

tempA

6

Merge

```

#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge

```

Idea 2: Trộn (Merge)

A

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

tempA

6	14
---	----

Merge

```

#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge

```

Idea 2: Trộn (Merge)

A

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

tempA

6	14	23
---	----	----

Merge

```

#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge

```

Idea 2: Trộn (Merge)

A

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

tempA

6	14	23	33
---	----	----	----

Merge

```

#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge

```

Idea 2: Trộn (Merge)

A

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

tempA

6	14	23	33	42
---	----	----	----	----

Merge

```

#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge

```

Idea 2: Trộn (Merge)

A

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

tempA

6	14	23	33	42	45
---	----	----	----	----	----

Merge

```

#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge

```

Idea 2: Trộn (Merge)

A

14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

tempA

6	14	23	33	42	45	67
---	----	----	----	----	----	----

Merge

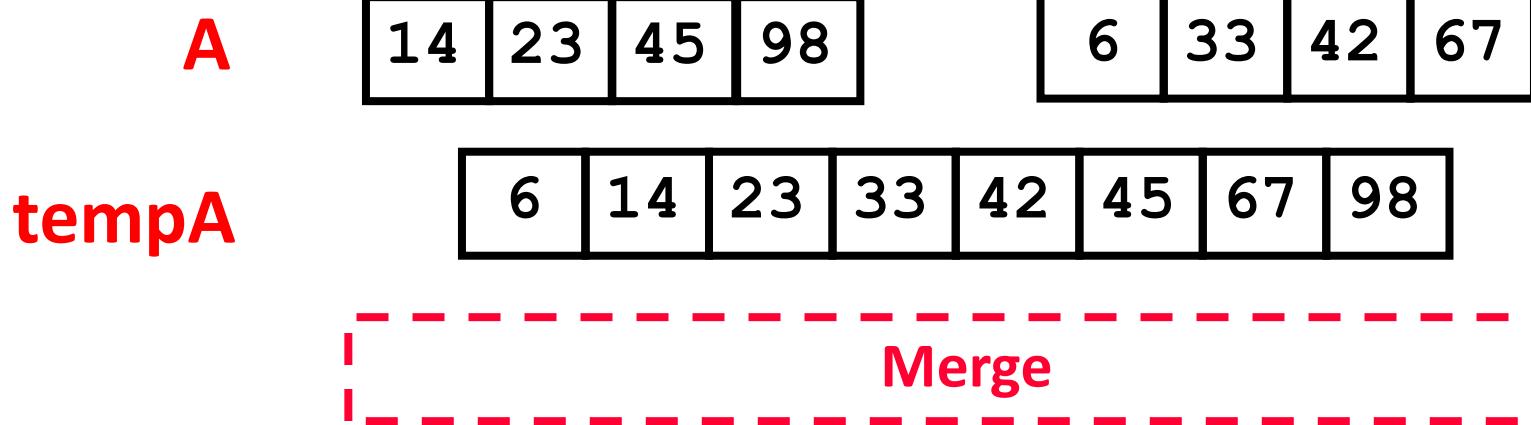
```

#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge

```

Idea 2: Trộn (Merge)



```

#define MAX_SIZE 500
void merge(int A[], int first, int mid, int last)
{
    int tempA[MAX_SIZE]; // mang phu
    int first1 = first; int last1 = mid;
    int first2 = mid + 1; int last2 = last;
    int index = first1;
    for (; (first1 <= last1) && (first2 <= last2); ++index)
    {
        if (A[first1] < A[first2])
            {tempA[index] = A[first1]; ++first1;}
        else
            { tempA[index] = A[first2]; ++first2;}
    }

    for (; first1 <= last1; ++first1, ++index)
        tempA[index] = A[first1]; // sao not day con 1
    for (; first2 <= last2; ++first2, ++index)
        tempA[index] = A[first2]; // sao not day con 2
    for (index = first; index <= last; ++index)
        A[index] = tempA[index]; // sao tra mang ket qua
} // end merge

```

```
void mergeSort(int A[], int first, int last)
{
    if (first < last)
    { // chia thành hai dãy con
        int mid = (first + last)/2;      // chỉ số điểm giữa
        // sắp xếp dãy con trái A[first..mid]
        mergeSort(A, first, mid);
        // sắp xếp dãy con phải A[mid+1..last]
        mergeSort(A, mid+1, last);
        // Trộn hai dãy con
        merge(A, first, mid, last);
    } // end if
} // end mergesort
```

```
void main()
{
    int a[5] = {8,4,3,2,1};
    mergeSort(a,0,4);
    for (int i = 0; i<5; i++)
        printf("%d \n",a[i]);
}
```

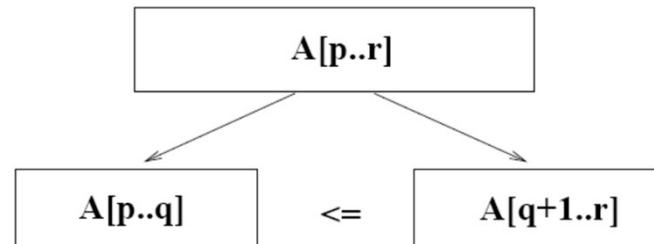
Nội dung

1. Sắp xếp chèn (Insertion sort)
 2. Sắp xếp chọn (Selection sort)
 3. Sắp xếp nổi bọt (Bubble sort)
 4. Sắp xếp trộn (Merge sort)
 5. **Sắp xếp nhanh (Quick sort)**
 6. Sắp xếp vun đống (Heap sort)
- Divide and conquer

5. Sắp xếp nhanh (Quick sort)

Thuật toán có thể mô tả đệ qui như sau (có dạng tương tự như merge sort):

1. **Neo đệ qui** (Base case). Nếu dãy chỉ còn không quá một phần tử thì nó là dãy được sắp và trả lại ngay dãy này mà không phải làm gì cả.
2. **Chia** (Divide):
 - Chọn một phần tử trong dãy và gọi nó là **phần tử chốt p** (pivot).
 - Chia dãy đã cho ra thành hai dãy con: Dãy con trái (L) gồm những phần tử \leq phần tử chốt, còn dãy con phải (R) gồm các phần tử \geq phần tử chốt. Thao tác này được gọi là "Phân đoạn" (Partition).

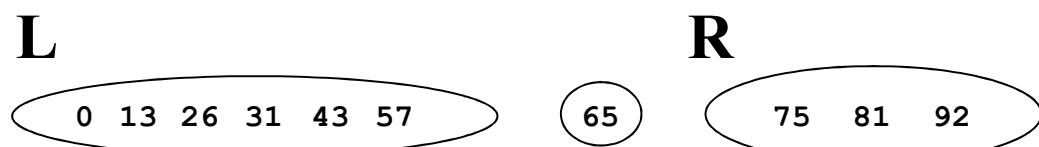
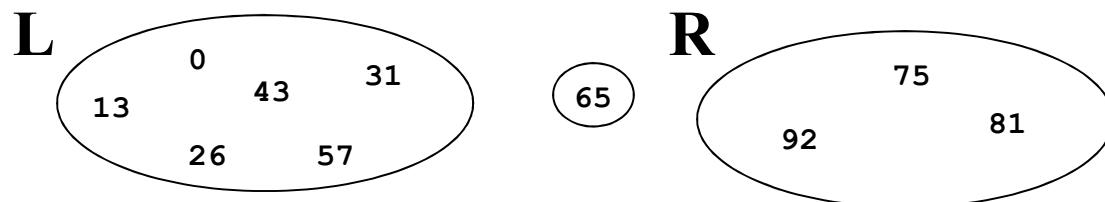
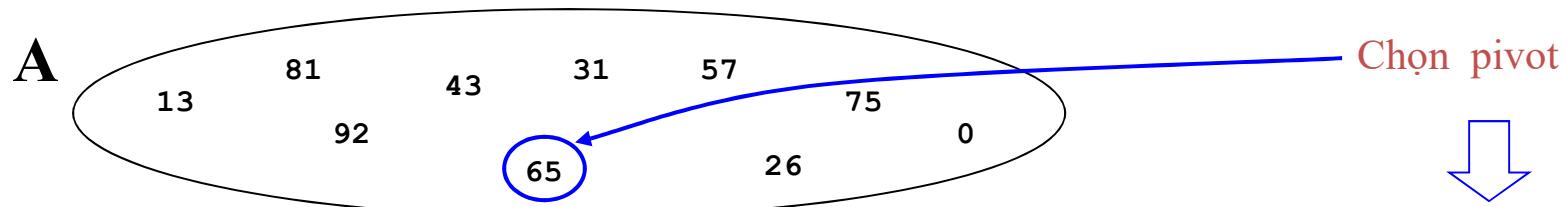


3. **Trị** (Conquer): Lặp lại một cách đệ qui thuật toán QuickSort đối với hai dãy con $L = A[p..q]$ và $R = A[q+1..r]$.
4. **Tổng hợp** (Combine): Dãy được sắp xếp là $L \ p \ R$.

Ngược lại với Merge Sort, trong Quick Sort thao tác chia là phức tạp, nhưng thao tác tổng hợp lại đơn giản.

Điểm mấu chốt để thực hiện Quick Sort chính là thao tác chia. Phụ thuộc vào thuật toán thực hiện thao tác này mà ta có các dạng Quick Sort cụ thể.

5. Sắp xếp nhanh (Quick sort): Ví dụ minh họa



Sơ đồ tổng quát sắp xếp nhanh (Quick sort)

- Sơ đồ tổng quát của QS có thể mô tả như sau:

Quick-Sort(A , $Left$, $Right$)

1. **if** ($Left < Right$) {
2. $Pivot = \text{Partition}(A, Left, Right);$
3. $\text{Quick-Sort}(A, Left, Pivot -1);$
4. $\text{Quick-Sort}(A, Pivot +1, Right);$
5. }

Hàm Partition(A , $Left$, $Right$) thực hiện chia $A[Left..Right]$ thành hai đoạn $A[Left..Pivot -1]$ và $A[Pivot+1..Right]$ sao cho:

- Các phần tử trong $A[Left..Pivot -1]$ là nhỏ hơn hoặc bằng $A[Pivot]$
- Các phần tử trong $A[Pivot+1..Right]$ là lớn hơn hoặc bằng $A[Pivot]$.

Lệnh gọi thực hiện thuật toán Quick-Sort(A , 1, n)

Sơ đồ tổng quát sắp xếp nhanh (Quick sort)

Knuth cho rằng khi dãy con chỉ còn một số lượng không lớn phần tử (theo ông là không quá 9 phần tử) thì ta nên sử dụng các thuật toán đơn giản để sắp xếp dãy này, chứ không nên tiếp tục chia nhỏ. Thuật toán trong tình huống như vậy có thể mô tả như sau:

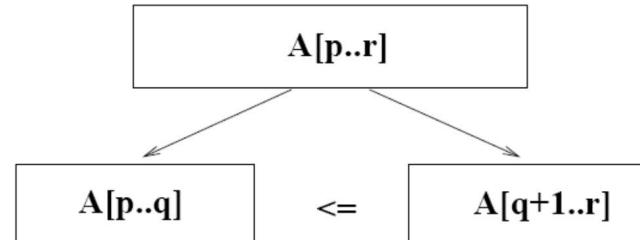
Quick-Sort(*A*, *Left*, *Right*)

1. **if** (*Right* - *Left* < n_0)
2. InsertionSort(*A*, *Left*, *Right*);
3. **else** {
4. *Pivot* = Partition(*A*, *Left*, *Right*);
5. Quick-Sort(*A*, *Left*, *Pivot* - 1);
6. Quick-Sort(*A*, *Pivot* + 1, *Right*);
7. }

Thao tác chia trong sắp xếp nhanh (Quick sort)

Chia (Divide):

- Chọn một phần tử trong dãy và gọi nó là **phần tử chốt p (pivot)**.
- Chia dãy đã cho ra thành hai dãy con: Dãy con trái (L) gồm những phần tử \leq phần tử chốt, còn dãy con phải (R) gồm các phần tử \geq phần tử chốt. Thao tác này được gọi là "Phân đoạn" (Partition).

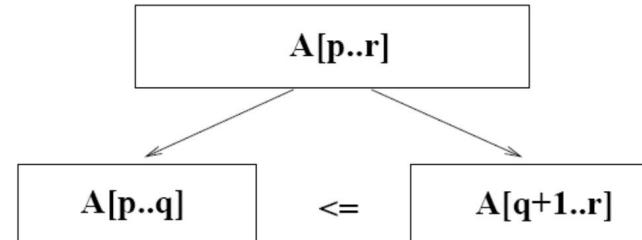


- Thao tác phân đoạn có thể cài đặt (tại chỗ) với thời gian $O(n)$.
- Việc chọn phần tử chốt có vai trò quyết định đối với hiệu quả của thuật toán. Tốt nhất nếu chọn được phần tử chốt là phần tử có giá trị trung bình trong danh sách (ta gọi phần tử như vậy là **trung vị/median**). Khi đó, sau $\log_2 n$ lần phân đoạn ta sẽ đạt tới danh sách với kích thước bằng 1. Tuy nhiên, điều đó rất khó thực hiện. Người ta thường sử dụng các cách chọn phần tử chốt sau đây:
 - Chọn phần tử trái nhất (đứng đầu) làm phần tử chốt.
 - Chọn phần tử phải nhất (đứng cuối) làm phần tử chốt.
 - Chọn phần tử đứng giữa danh sách làm phần tử chốt.
 - Chọn phần tử trung vị trong 3 phần tử đứng đầu, đứng giữa và đứng cuối làm phần tử chốt (Knuth).
 - Chọn ngẫu nhiên một phần tử làm phần tử chốt.

Thao tác chia trong sắp xếp nhanh (Quick sort)

Chia (Divide):

- Chọn một phần tử trong dãy và gọi nó là **phần tử chốt p (pivot)**.
- Chia dãy đã cho ra thành hai dãy con: Dãy con trái (L) gồm những phần tử \leq phần tử chốt, còn dãy con phải (R) gồm các phần tử \geq phần tử chốt. Thao tác này được gọi là "Phân đoạn" (Partition).



- Hiệu quả của thuật toán phụ thuộc rất nhiều vào việc phần tử nào được chọn làm phần tử chốt:
 - Thời gian tính trong tình huống tồi nhất của QS là $O(n^2)$. Trường hợp xấu nhất xảy ra khi danh sách là đã được sắp xếp và phần tử chốt được chọn là phần tử trái nhất của dãy.
 - Nếu phần tử chốt được chọn ngẫu nhiên, thì QS có độ phức tạp tính toán là $O(n \log n)$.

Thuật toán phân đoạn: 2-way partitioning

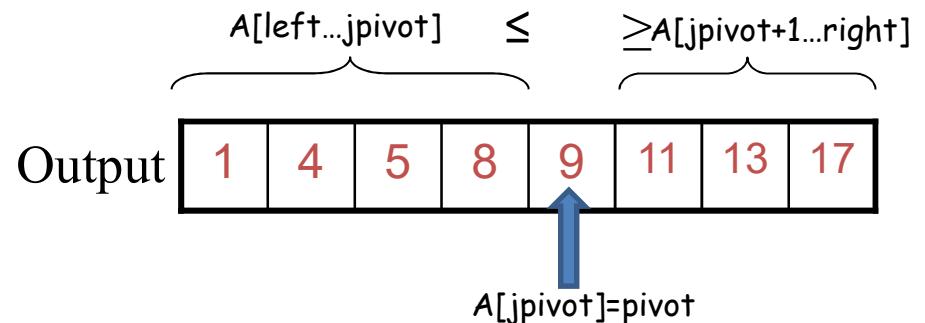
Ta xây dựng hàm **Partition(A, left, right)** làm việc sau:

- **Input:** Mảng $A[left .. right]$.
- **Output:** Phân bố lại các phần tử của mảng đầu vào dựa vào phần tử *pivot* được chọn và trả lại chỉ số *jpivot* thỏa mãn:
 - $a[jpivot]$ chứa giá trị của *pivot*,
 - $a[i] \leq a[jpivot]$, với mọi $left \leq i < jpivot$,
 - $a[j] \geq a[jpivot]$, với mọi $jpivot < j \leq right$.

trong đó *pivot* là giá trị phần tử chốt được chọn



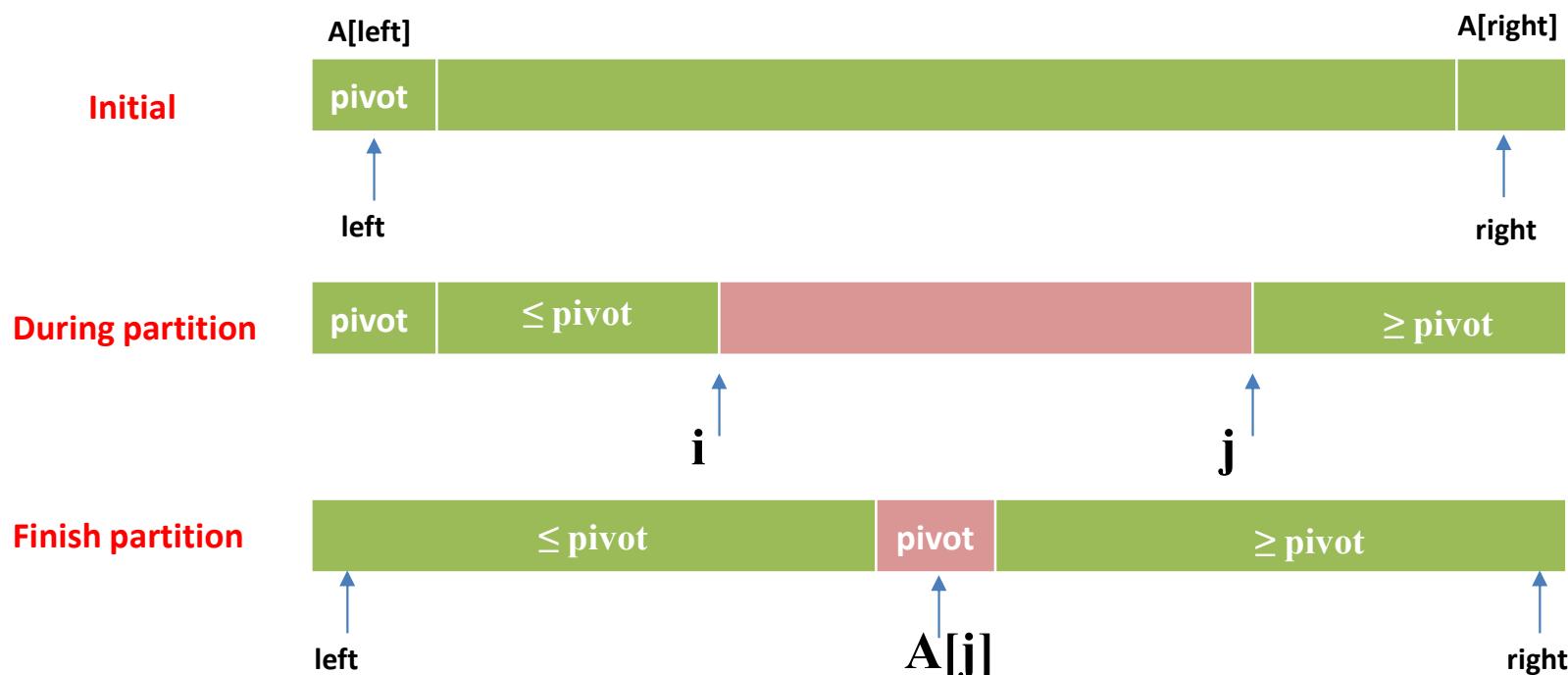
Ví dụ: chọn pivot là phần tử đầu tiên trong mảng



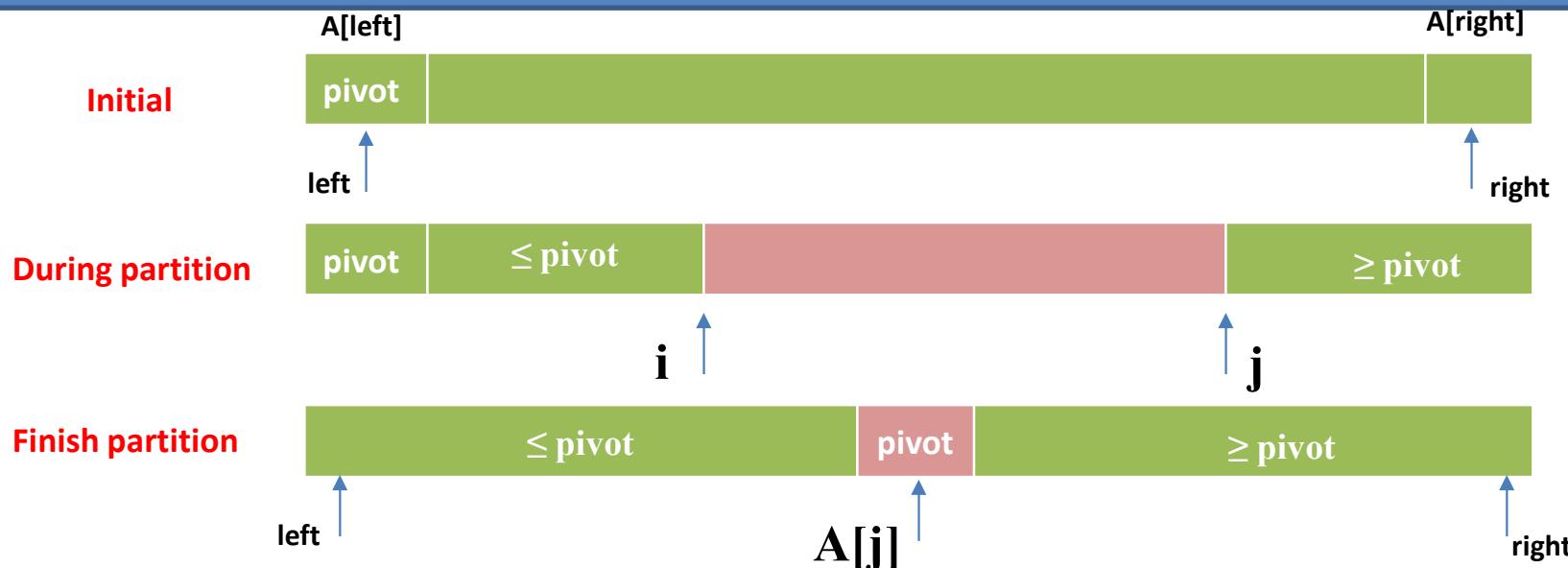
Phần tử chốt là phần tử đứng đầu: : 2-way partitioning

Hàm phân đoạn (Partition) dịch chuyển các phần tử của dãy cần sắp xếp $A[\text{left}].....A[\text{right}]$ để thu được một dãy mới thỏa mãn 3 điều kiện sau:

- Phần tử chốt pivot sẽ được đặt về đúng vị trí của nó (kí hiệu là vị trí thứ j) trong dãy sắp xếp
- Các phần tử từ $A[\text{left}]...A[j-1]$ đều \leq phần tử chốt
- Các phần tử từ $A[j+1]...A[\text{right}]$ đều \geq phần tử chốt



Phần tử chốt là phần tử đứng đầu: 2-way partitioning



- Chọn phần tử chốt là phần tử đứng đầu dãy $\text{pivot} = A[\text{left}]$ (phần tử này khi kết thúc hàm Partition, nó sẽ được đặt về vị trí j là vị trí thực sự của nó trong dãy A khi các phần tử được sắp xếp tăng dần)
- $i = \text{left} + 1$; Duyệt từ trái: Duyệt từ phần tử $A[i]$ về cuối dãy để tìm phần tử đầu tiên $\text{FIRST1} \geq \text{pivot}$
- $j = \text{right}$; Duyệt từ phải: Duyệt từ phần tử $A[j]$ về đầu dãy để tìm phần tử đầu tiên $\text{FIRST2} \leq \text{pivot}$
- SWAP ($\text{FIRST1}, \text{FIRST2}$) vì 2 phần tử này đang nằm không đúng vị trí của nó
- Tiếp tục duyệt từ trái và từ phải để hoán đổi vị trí các phần tử nếu cần thiết, dừng duyệt khi $i \geq j$
- Cuối cùng, $\text{SWAP}(A[\text{left}] = \text{pivot}, A[j])$ – hoán đổi vị trí phần tử chốt với phần tử cuối cùng của nửa trái
- $\text{return } j$

Ví dụ quá trình thực hiện hàm partition khi chọn phần tử chốt là phần tử đứng đầu dãy

- $i = \text{left} + 1$; Duyệt từ trái: Duyệt từ phần tử $A[i]$ về cuối dãy để tìm phần tử đầu tiên $\text{FIRST1} \geq \text{pivot}$
- $j = \text{right}$; Duyệt từ phải: Duyệt từ phần tử $A[j]$ về đầu dãy để tìm phần tử đầu tiên $\text{FIRST2} \leq \text{pivot}$
- SWAP ($\text{FIRST1}, \text{FIRST2}$) vì 2 phần tử này đang nằm không đúng vị trí của nó
- Tiếp tục duyệt từ trái và từ phải để hoán đổi vị trí các phần tử nếu cần thiết, dừng duyệt khi $i \geq j$
- Cuối cùng, $\text{SWAP}(A[\text{left}] = \text{pivot}, A[j])$ – hoán đổi vị trí phần tử chốt với phần tử cuối cùng của nửa trái

	i	j	pivot	left	A[]	right	right
				0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15		
initial values	0	16	K R A T E L E P U I M Q C X O S				
scan left, scan right	1	12	K R A T E L E P U I M Q C X O S	R			
exchange	1	12	K C A T E L E P U I M Q R X O S				
scan left, scan right	3	9	K C A T E L E P U I M Q R X O S	A T			
exchange	3	9	K C A I E L E P U T M Q R X O S				
scan left, scan right	5	6	K C A I E L E P U T M Q R X O S	E L	E P U		
exchange	5	6	K C A I E E L P U T M Q R X O S				
scan left, scan right	6	5	K C A I E E L P U T M Q R X O S	E L			
final exchange	6	5	E C A I E K L P U T M Q R X O S				
result	5		E C A I E K L P U T M Q R X O S				

Partitioning trace (array contents before and after each exchange)

Phần tử chốt là phần tử đứng đầu dãy:

Partition(*A*, *left*, *right*)

```
i = left; j = right + 1;  
pivot = A[left];  
while (true) do  
{  
    i = i + 1;  
    //Tìm từ trái sang phần tử đầu tiên  $\geq$  pivot:  
    while i  $\leq$  right and A[i] < pivot do i = i + 1;  
    j = j - 1;  
    //Tìm từ phải sang phần tử đầu tiên  $\leq$  pivot:  
    while j  $\geq$  left and pivot < A[j] do j = j - 1;  
    if (i  $\geq$  j) break;  
    swap(A[i] , A[j]);  
}  
swap(A[j], A[left]);  
return j;
```

pivot được chọn là phần tử đứng đầu

j là chỉ số (jpivot) cần trả lại, do
đó cần đổi chỗ a[left] và a[j]

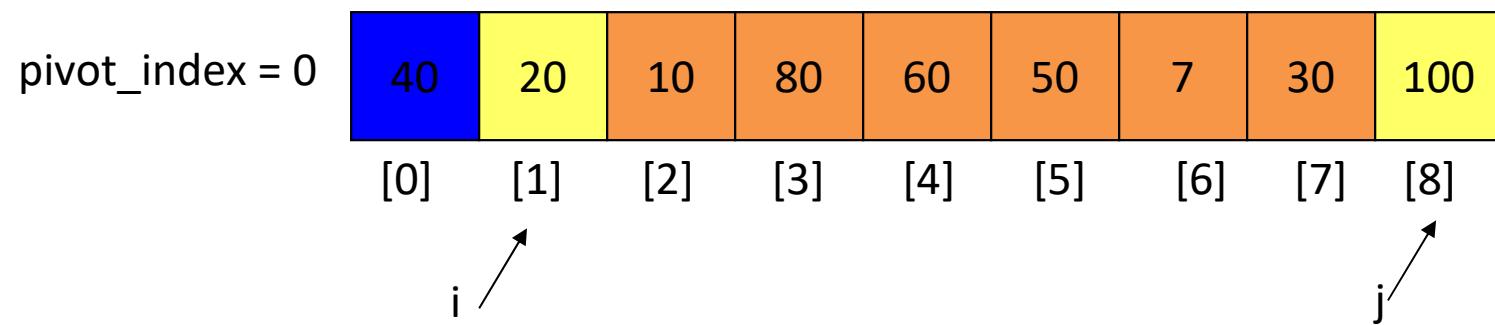


i

j

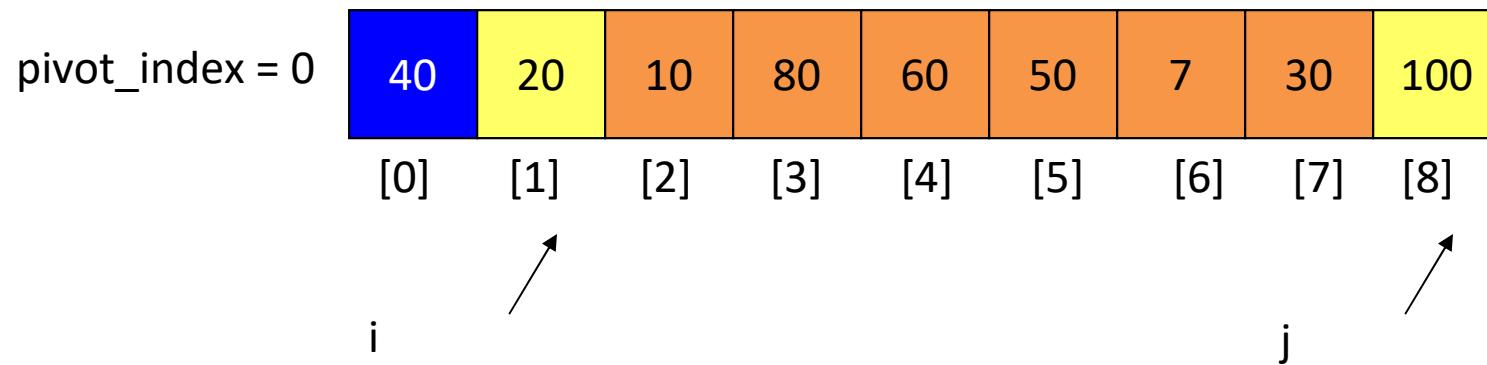
Sau khi chọn *pivot*, dịch các con trỏ *i* và *j* từ đầu và cuối mảng và
đổi chỗ cặp phần tử thoả mãn $a[i] \geq pivot$ và $a[j] \leq pivot$

Phần tử chốt là phần tử đứng đầu dãy:



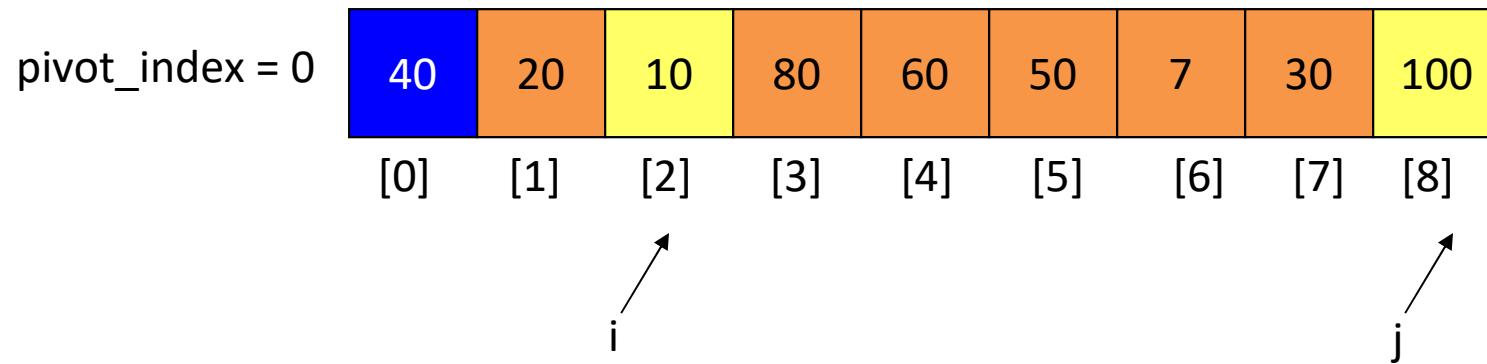
Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$



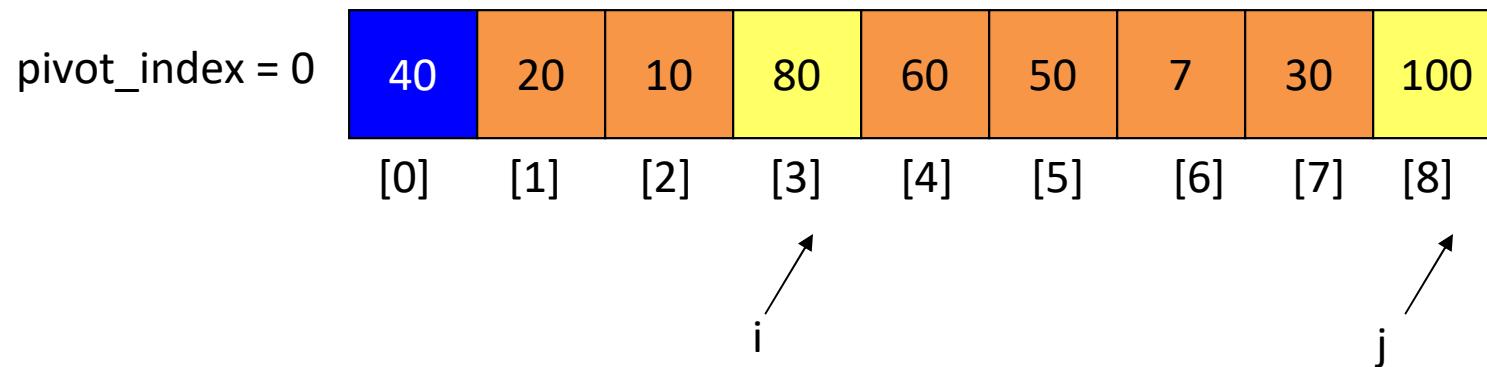
Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$



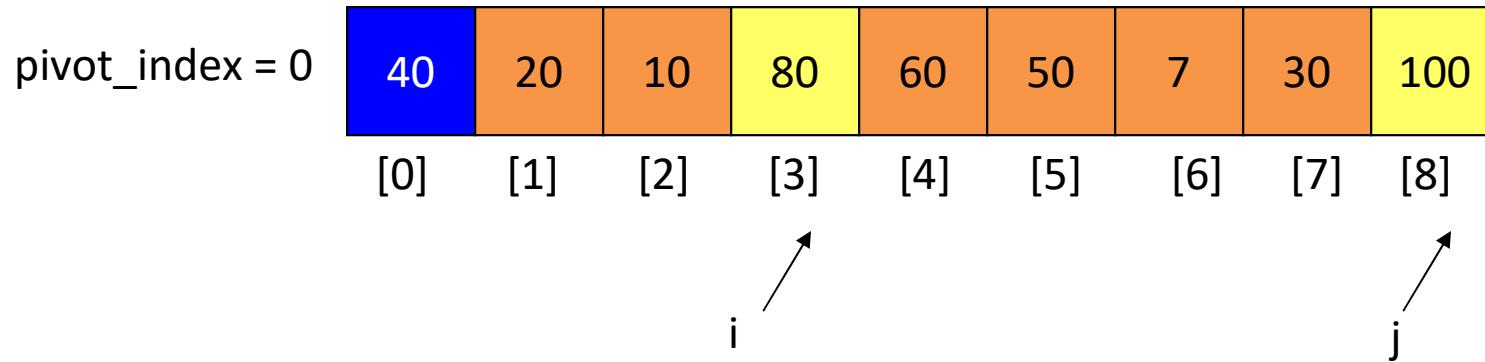
Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$



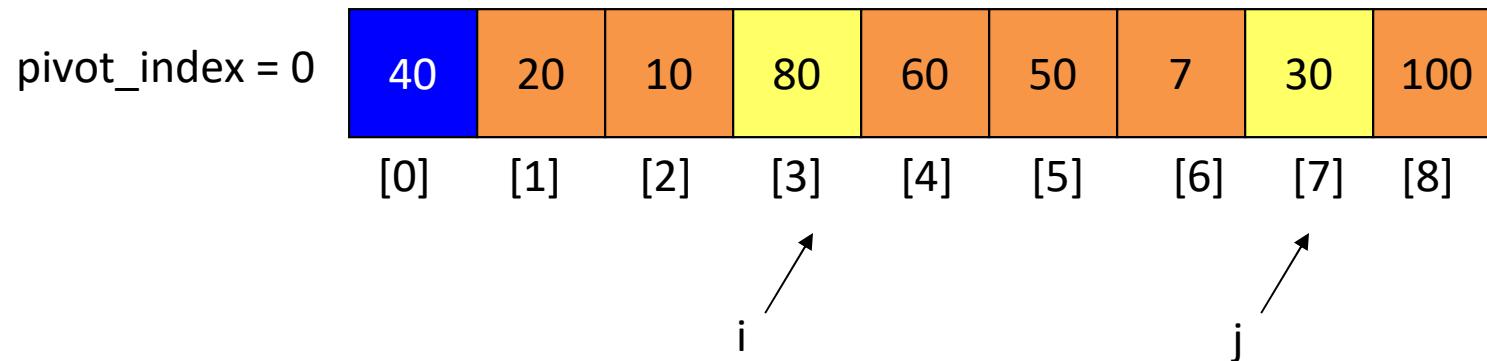
Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$



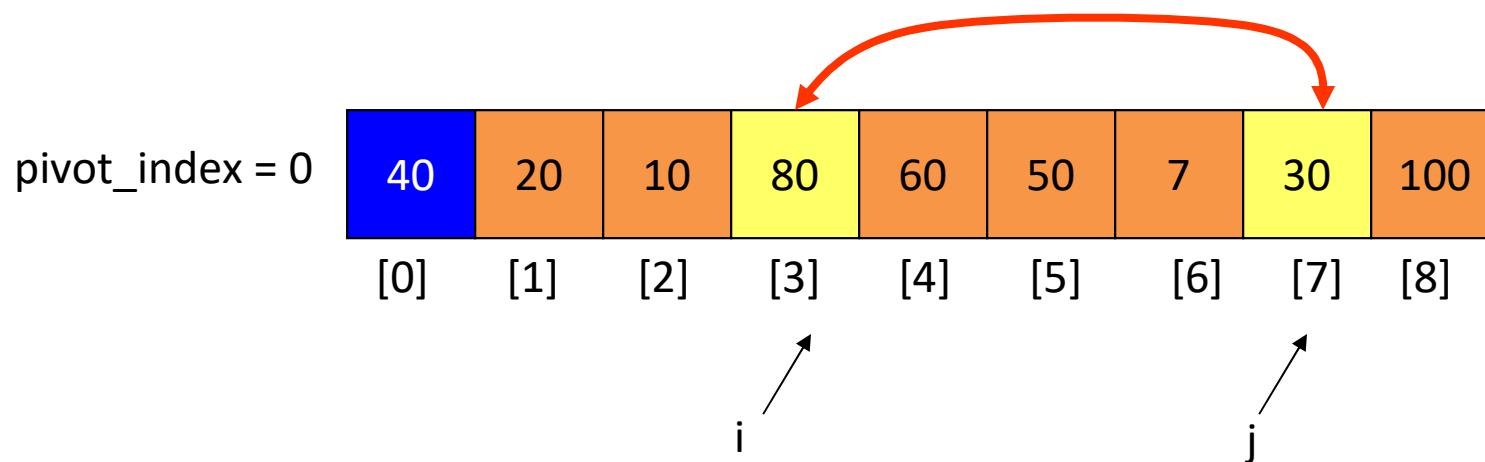
Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$



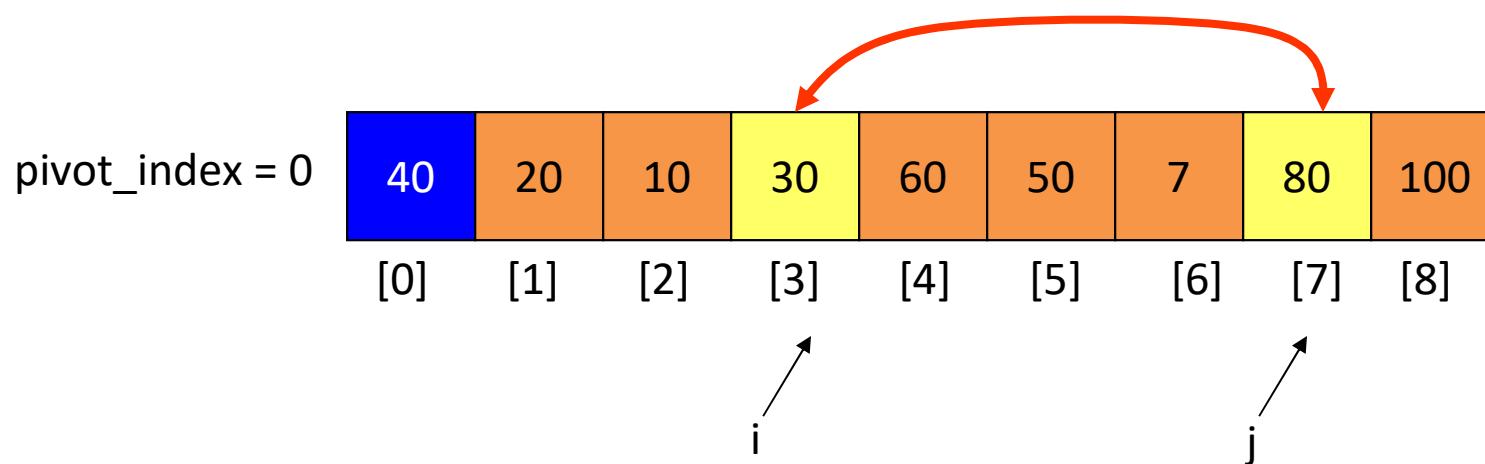
Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 $swap(A[i], A[j])$



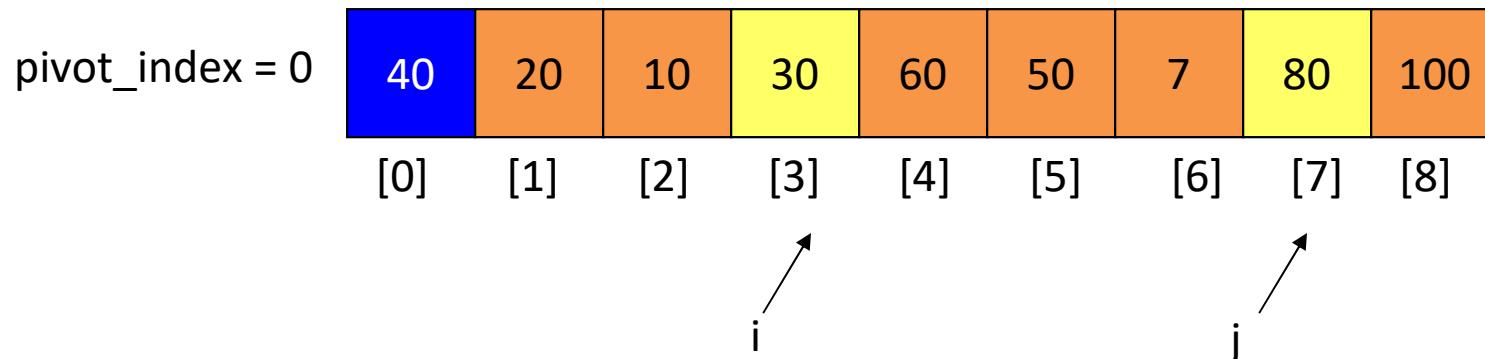
Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 $swap(A[i], A[j])$



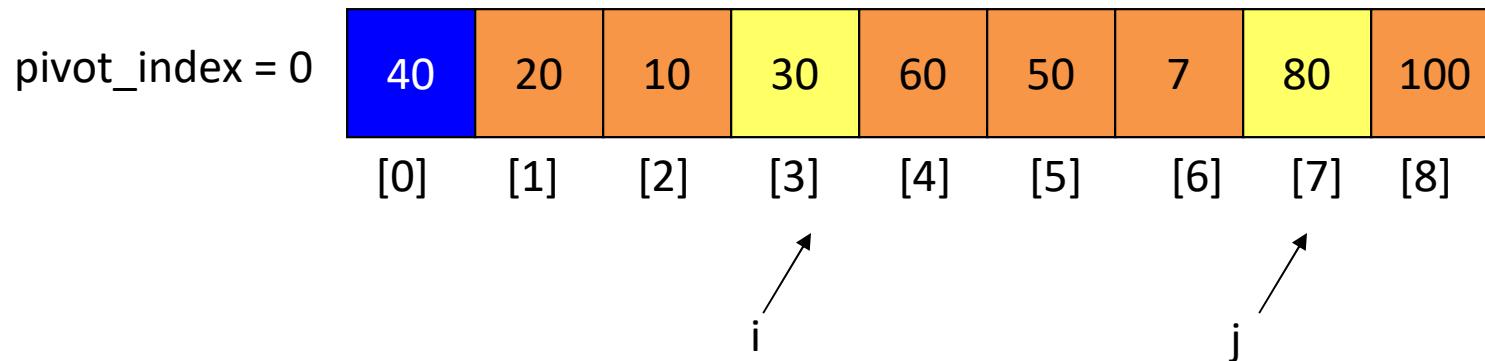
Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 swap($A[i]$, $A[j]$)
4. While $j > i$, go to 1.



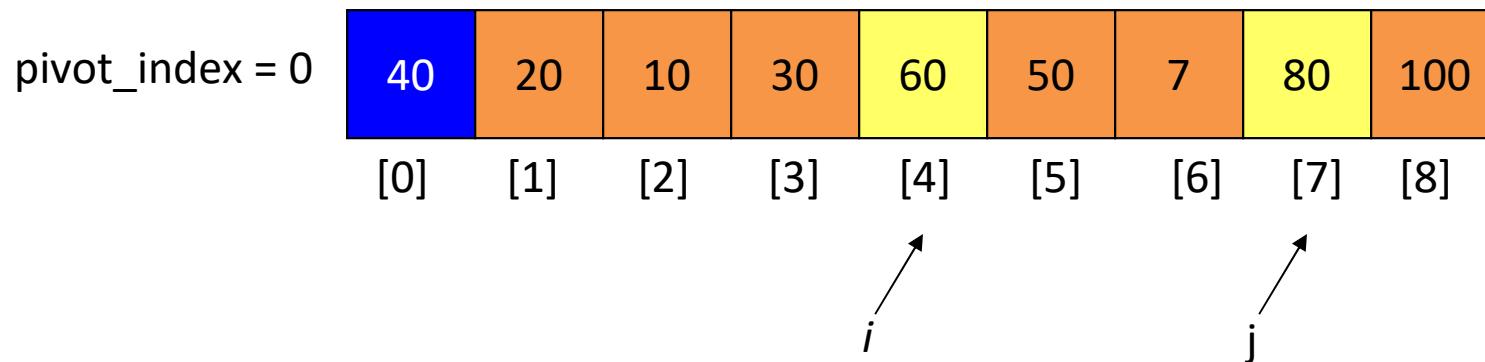
Phần tử chốt là phần tử đứng đầu

- 1. While $i \leq right$ and $A[i] < pivot$
 $++i$
- 2. While $j \geq left$ and $A[j] > pivot$
 $--j$
- 3. If $i < j$
 $swap(A[i], A[j])$
- 4. While $j > i$, go to 1.



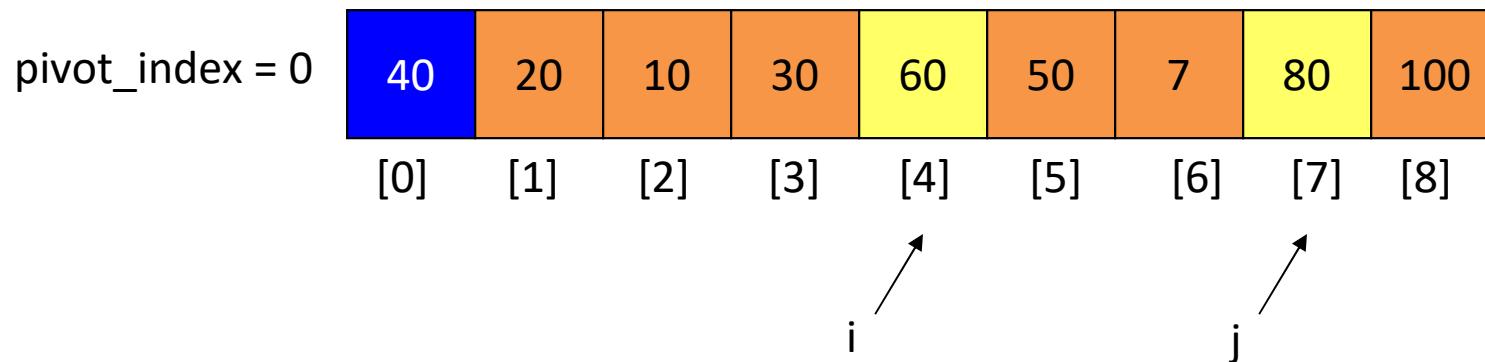
Phần tử chốt là phần tử đứng đầu

- 1. While $i \leq right$ and $A[i] < pivot$
 $++i$
- 2. While $j \geq left$ and $A[j] > pivot$
 $--j$
- 3. If $i < j$
 swap($A[i]$, $A[j]$)
- 4. While $j > i$, go to 1.



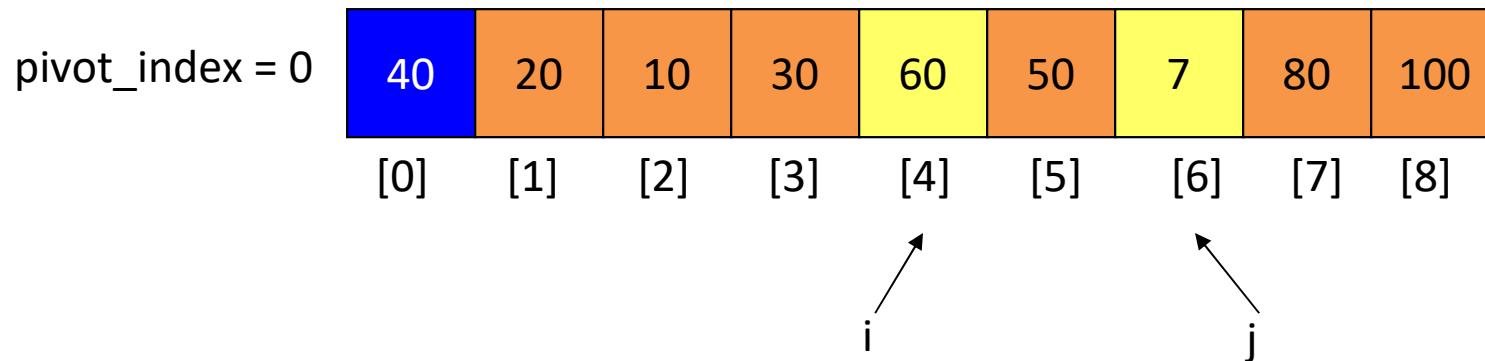
Phần tử chốt là phần tử đứng đầu

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
- 2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 swap($A[i]$, $A[j]$)
4. While $j > i$, go to 1.



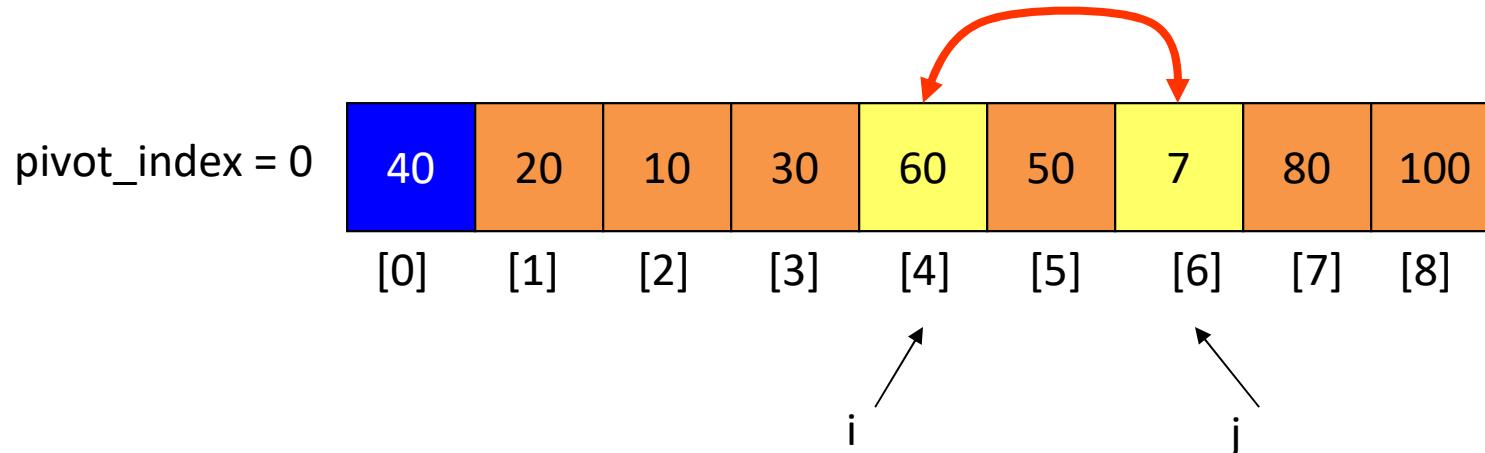
Phần tử chốt là phần tử đứng đầu

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
- 2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 swap($A[i]$, $A[j]$)
4. While $j > i$, go to 1.



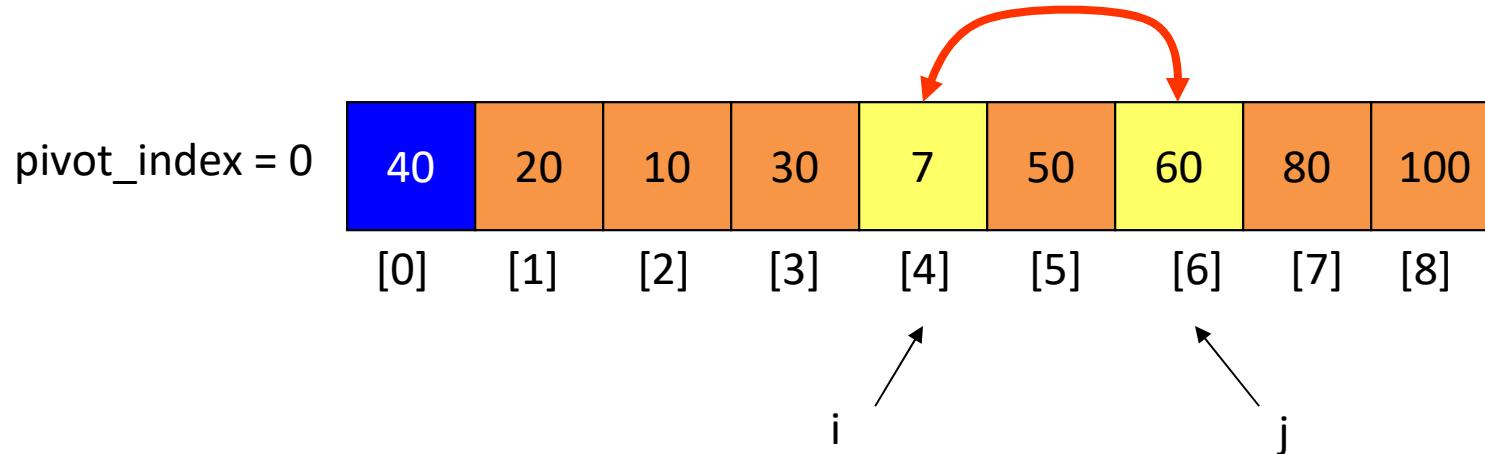
Phần tử chốt là phần tử đứng đầu

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$
- 3. If $i < j$
 $swap(A[i], A[j])$
4. While $j > i$, go to 1.



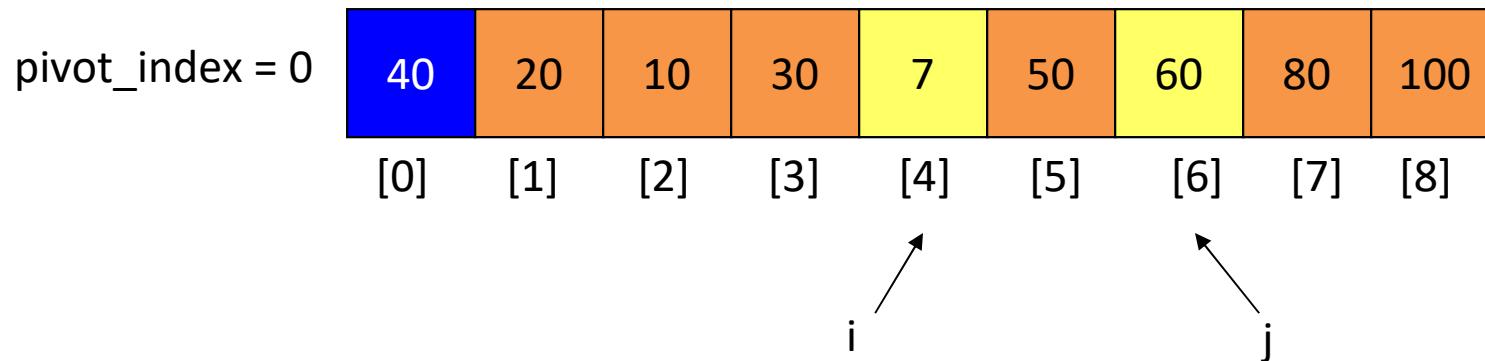
Phần tử chốt là phần tử đứng đầu

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$
- 3. If $i < j$
 $swap(A[i], A[j])$
4. While $j > i$, go to 1.



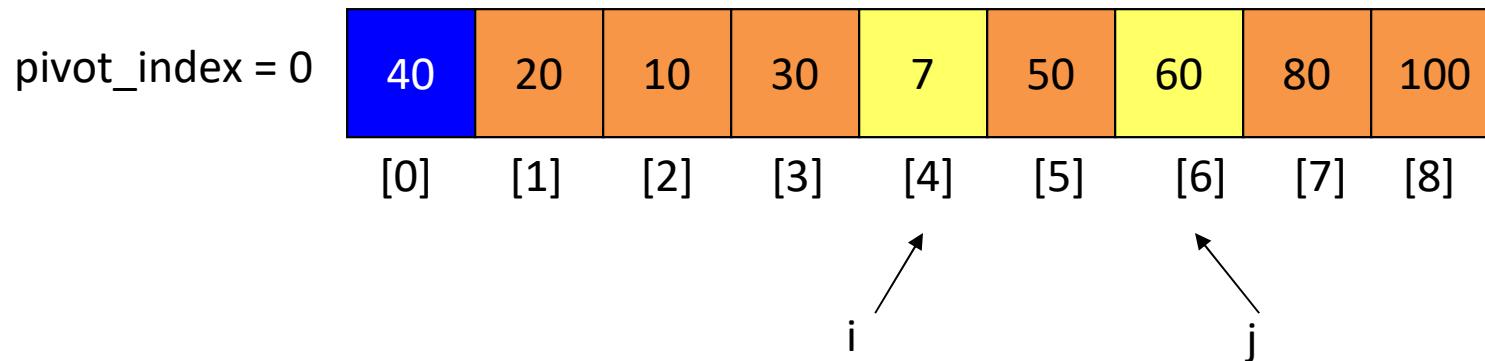
Phần tử chốt là phần tử đứng đầu

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 $swap(A[i], A[j])$
- 4. While $j > i$, go to 1.



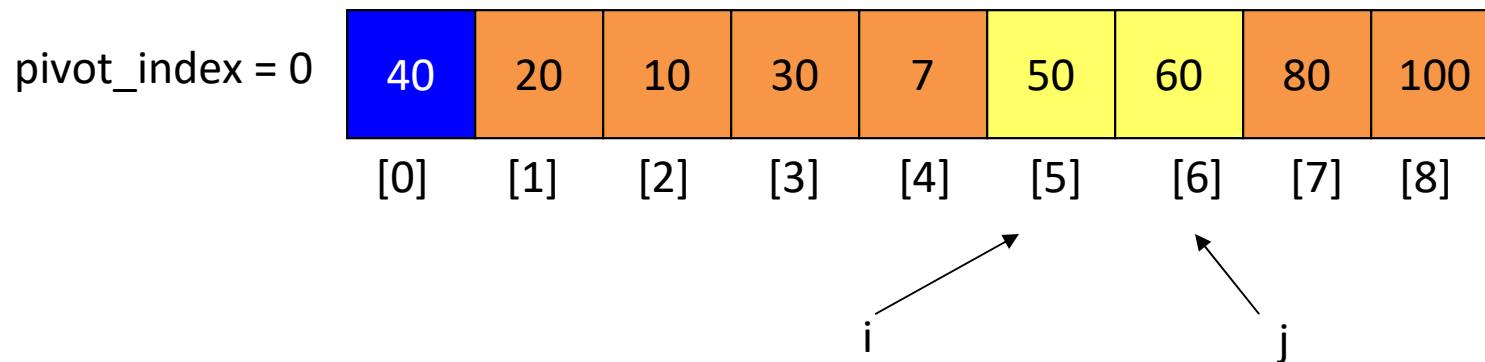
Phần tử chốt là phần tử đứng đầu

- 1. While $i \leq right$ and $A[i] < pivot$
 $++i$
- 2. While $j \geq left$ and $A[j] > pivot$
 $--j$
- 3. If $i < j$
 $swap(A[i], A[j])$
- 4. While $j > i$, go to 1.



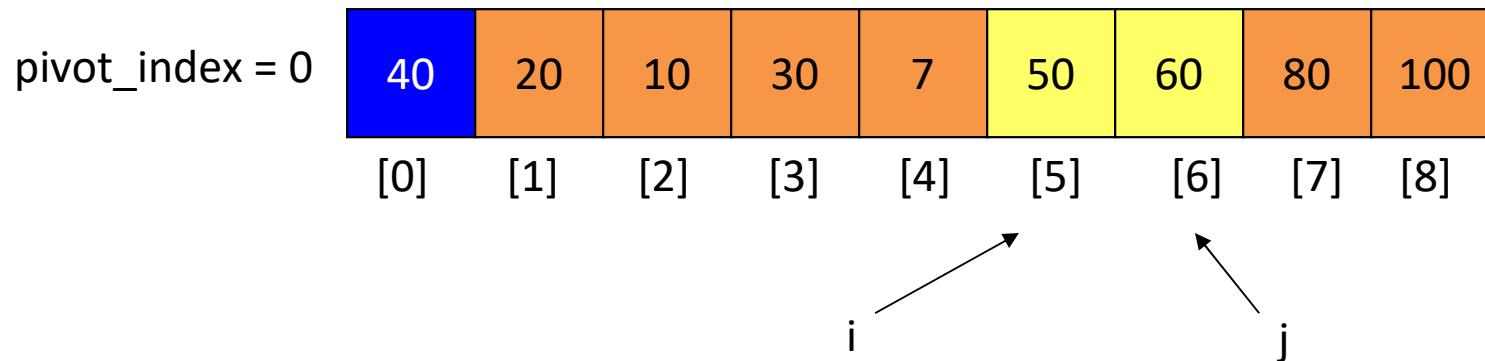
Phần tử chốt là phần tử đứng đầu

- 1. While $i \leq right$ and $A[i] < pivot$
 $++i$
- 2. While $j \geq left$ and $A[j] > pivot$
 $--j$
- 3. If $i < j$
 $swap(A[i], A[j])$
- 4. While $j > i$, go to 1.



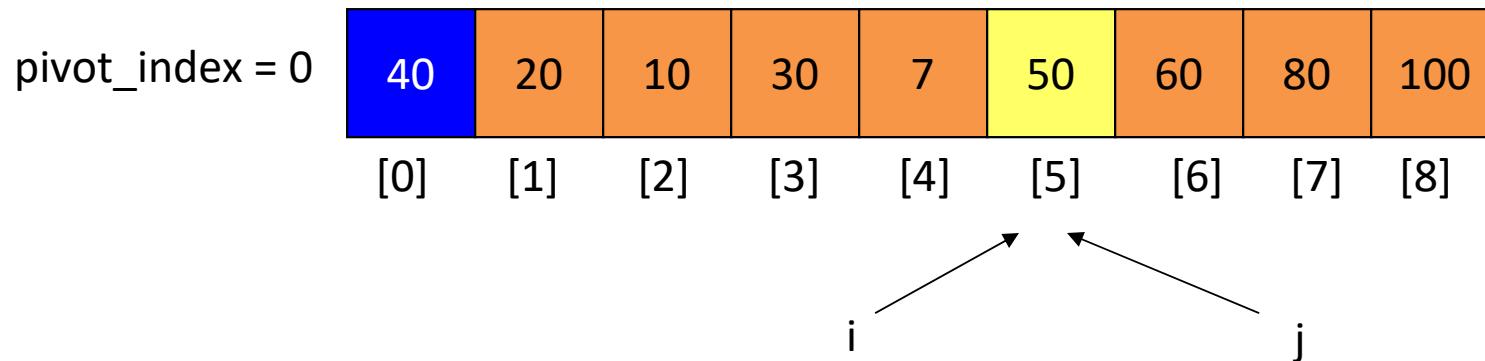
Phần tử chốt là phần tử đứng đầu

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
- 2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 swap($A[i]$, $A[j]$)
4. While $j > i$, go to 1.



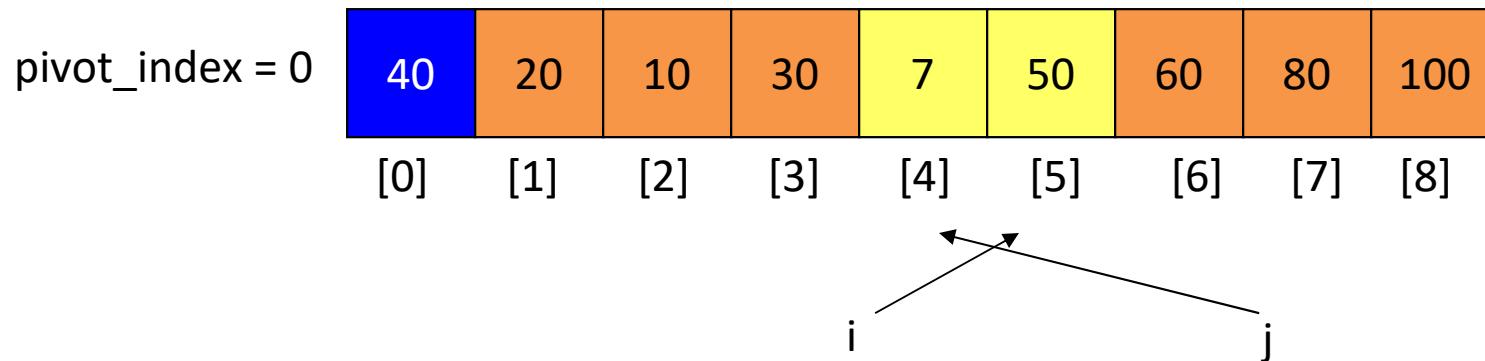
Phần tử chốt là phần tử đứng đầu

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
- 2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 swap($A[i], A[j]$)
4. While $j > i$, go to 1.



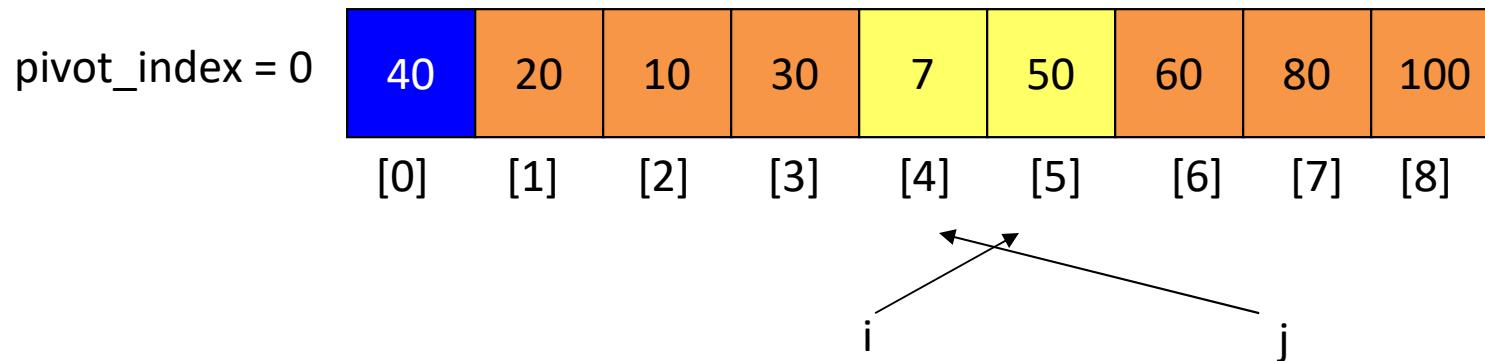
Phần tử chốt là phần tử đứng đầu

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
- 2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 swap($A[i]$, $A[j]$)
4. While $j > i$, go to 1.



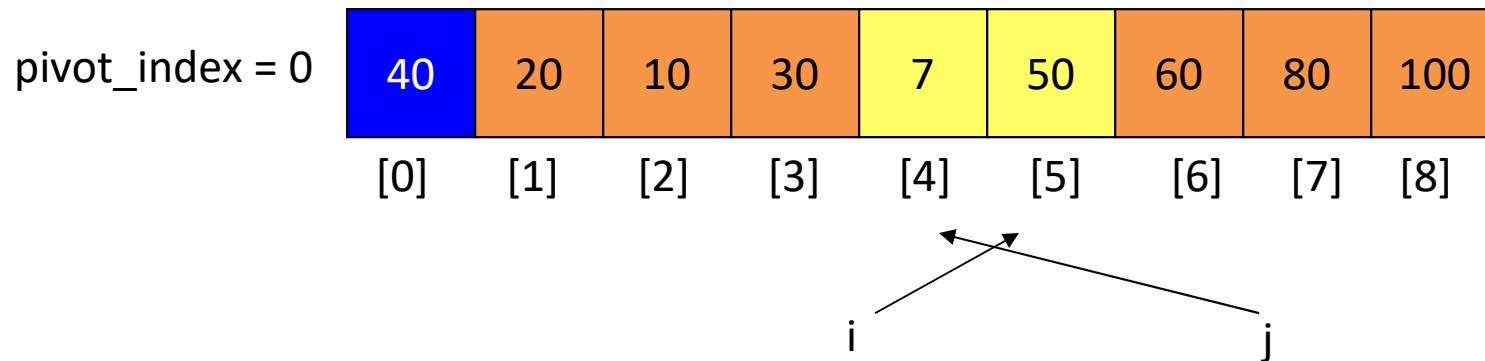
Phần tử chốt là phần tử đứng đầu

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$
- 3. If $i < j$
 $swap(A[i], A[j])$
4. While $j > i$, go to 1.



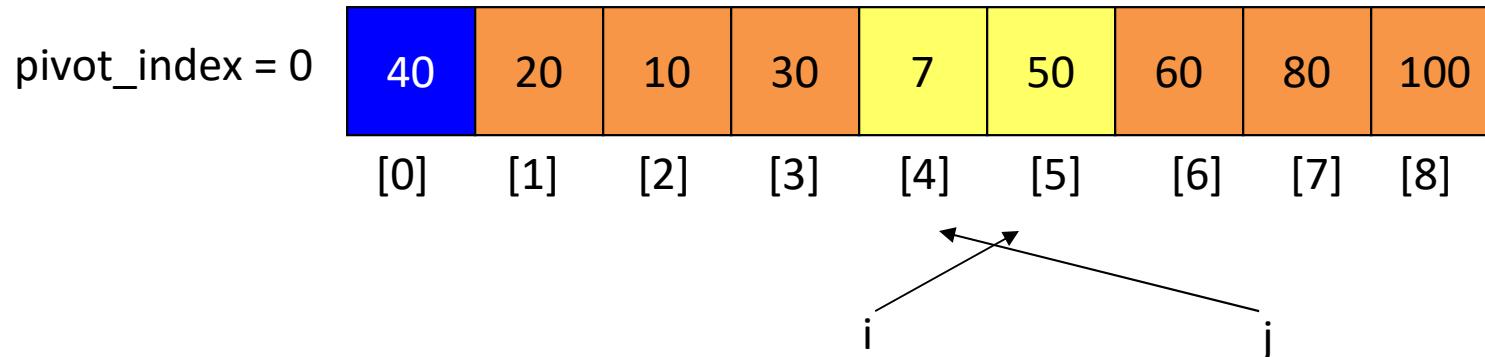
Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 $swap(A[i], A[j])$
- 4. While $j > i$, go to 1.



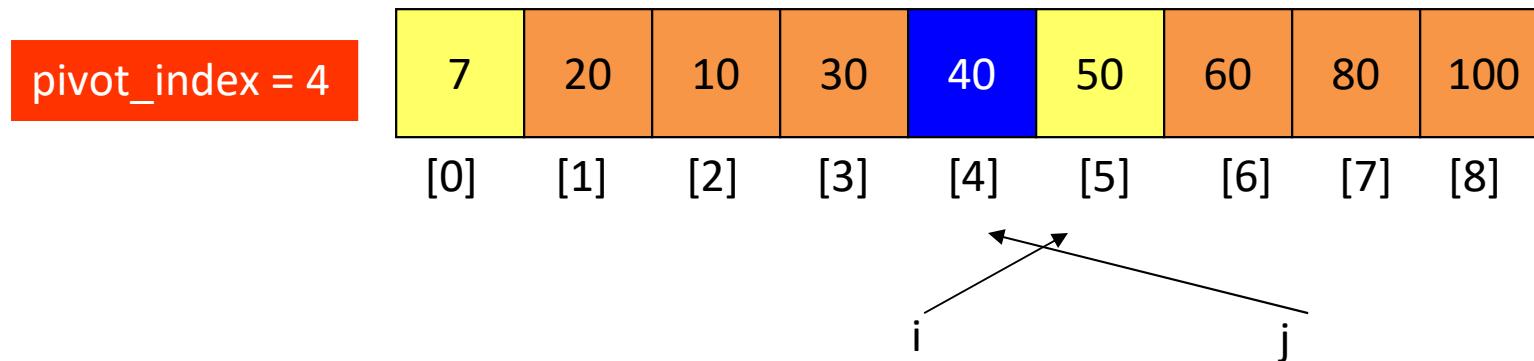
Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 swap($A[i]$, $A[j]$)
4. While $j > i$, go to 1.

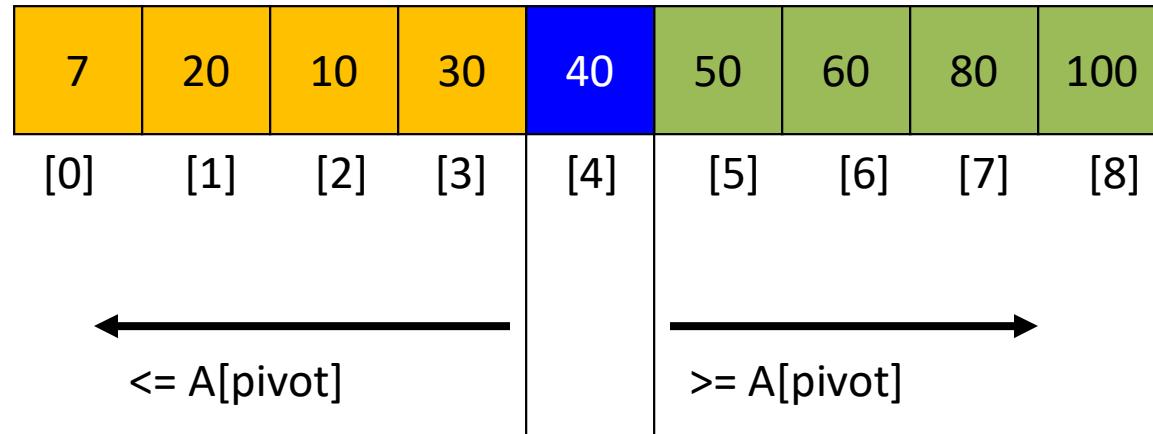


Phần tử chốt là phần tử đứng đầu dãy:

1. While $i \leq right$ and $A[i] < pivot$
 $++i$
2. While $j \geq left$ and $A[j] > pivot$
 $--j$
3. If $i < j$
 swap($A[i]$, $A[j]$)
4. While $j > i$, go to 1.
- 5. Swap($A[j]$, $A[pivot_index]$)



Dãy thu được sau khi gọi Partition(A,0,8)



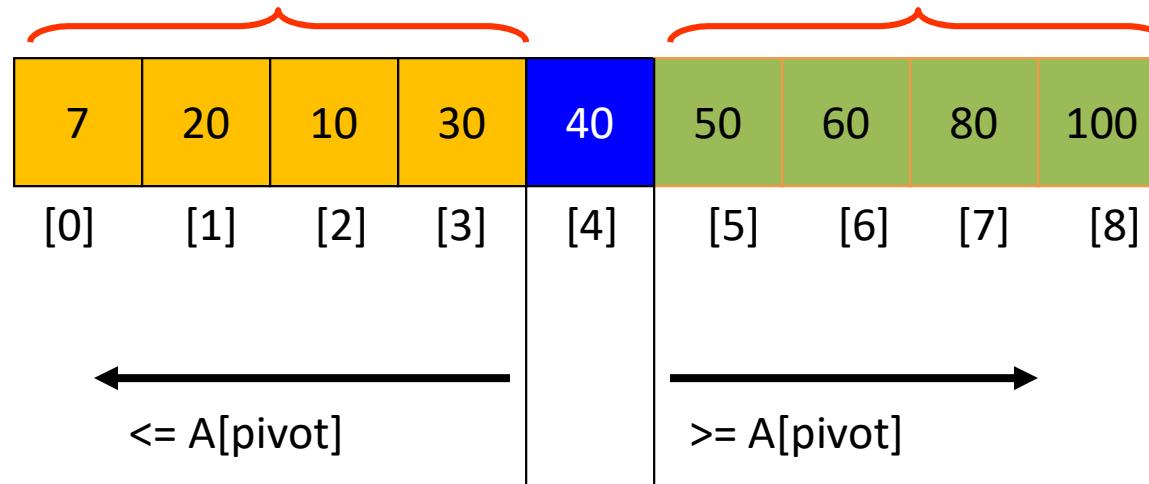
Quick-Sort($A, Left, Right$)

1. if ($Left < Right$) {
2. $Pivot = \text{Partition}(A, Left, Right);$
3. $\text{Quick-Sort}(A, Left, Pivot - 1);$
4. $\text{Quick-Sort}(A, Pivot + 1, Right);$
5. }

Quick-Sort($A, 0, 8$);

Partition($A, 0, 8$);

Recursion: Quicksort Sub-arrays



Quick-Sort($A, Left, Right$)

1. if ($Left < Right$) {
 2. Pivot = Partition($A, Left, Right$);
 3. Quick-Sort($A, Left, Pivot - 1$);
 4. Quick-Sort($A, Pivot + 1, Right$);
5. }

Quick-Sort($A, 0, 8$);

- 4 = Partition($A, 0, 8$);
 Quick-Sort($A, 0, 3$); →
 Quick-Sort($A, 5, 8$);

Source code QuickSort: Phân tử chốt là phân tử đứng đầu dãy

```
//QuickSort: Chon phan tu dau Lam phan tu chot:  
int Partition(int A[], int left, int right)  
{  
    int i = left; int j = right+1;  
    int pivot = A[left];  
    while (true)  
    {  
        //Tim tu trai sang phan tu dau tien >=pivot:  
        i = i + 1;  
        while (i <= right && A[i] < pivot) i=i+1;  
        //Tim tu phai sang phan tu dau tien <=pivot:  
        j -=1;  
        while (j >= left && pivot < A[j]) j =j-1;  
        if (i >= j) break;  
        swap(&A[i], &A[j]);  
    }  
    swap(&A[j], &A[left]);  
    return j;  
}
```

```
void swap(int *a, int *b)  
{  
    int temp =*a;  
    *a=*b;  
    *b=temp;  
}
```

```
//QuickSort: Chon phan tu dau Lam phan tu chot:  
int Partition(int A[], int left, int right)  
{  
    int i = left; int j = right+1;  
    int pivot = A[left];  
    while (true)  
    {  
        //Tim tu trai sang phan tu dau tien >=pivot:  
        while (A[++i] < pivot)  
            if (i==right) break;  
        //Tim tu phai sang trai phan tu dau tien < pivot  
        while (pivot < A[--j])  
            if (j==left) break; //khong can thiet vi phan tu chot a[left] acts as sentinel  
            if (i >= j) break;  
        swap(&A[i], &A[j]);  
    }  
    swap(&A[j], &A[left]);  
    return j;  
}
```

```
void QuickSort(int A[], int Left, int Right)  
{  
    int index_Pivot;  
    if (Left < Right )  
    {  
        index_Pivot =Partition(A,Left,Right);  
        QuickSort(A,Left,index_Pivot-1);  
        QuickSort(A, index_Pivot +1, Right);  
    }  
}
```

Source code QuickSort: Phần tử chốt là phần tử đứng giữa dãy

```
int PartitionMid(int A[], int left, int right)
{
    int pivot = A[(left + right)/2];
    while (left < right){
        //Tim tu trai sang phai phan tu dau tien >= pivot:
        while (A[left] < pivot) left++;
        //Tim tu phai sang trai phan tu dau tien <= pivot:
        while (A[right] > pivot) right--;
        if (left < right)
        {
            //doi cho 2 phan tu do cho nhau:
            swap(&A[left],&A[right]);
            left++;right--;
        }
    }
    return right;
}

void QuickSort(int A[], int Left, int Right)
{
    int index_Pivot;
    if (Left < Right )
    {
        index_Pivot =PartitionMid(A,Left,Right);
        QuickSort(A,Left,index_Pivot-1);
        QuickSort(A, index_Pivot +1, Right);
    }
}
```

Source code QuickSort: Phần tử chốt là phần tử đứng cuối dãy

```
int PartitionLast(int A[], int left, int right) {
    int pivot = A[right];
    int j = left - 1;
    for (int i = left; i < right; i++) {
        if (pivot >= A[i])
        {
            j = j + 1;
            swap(&A[i], &A[j]);
        }
    }
    A[right] = A[j + 1];
    A[j + 1] = pivot;
    return (j + 1);
}
```

```
void QuickSort(int A[], int Left, int Right)
{
    int index_Pivot;
    if (Left < Right )
    {
        index_Pivot =PartitionLast(A,Left,Right);
        QuickSort(A,Left,index_Pivot-1);
        QuickSort(A, index_Pivot +1, Right);
    }
}
```

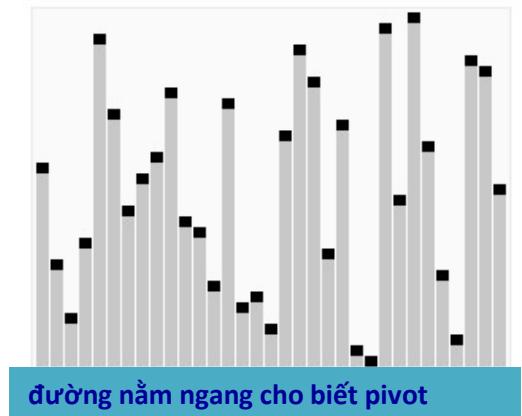
5. Sắp xếp nhanh (Quick sort)

- Thuật toán sắp xếp nhanh được phát triển bởi Hoare năm 1960 khi ông đang làm việc cho hãng máy tính nhỏ Elliott Brothers ở Anh, được ông dùng để dịch tiếng Nga sang tiếng Anh.
- QS có thời gian tính trung bình là $O(n \log n)$, tuy nhiên thời gian tính tồi nhất của nó lại là $O(n^2)$.
- QS là thuật toán sắp xếp tại chỗ, nhưng nó không có tính ổn định.
- QS khá đơn giản về lý thuyết, nhưng lại không dễ cài đặt.



C.A.R. Hoare

January 11, 1934
ACM Turing Award, 1980
Photo: 2006



đường nằm ngang cho biết pivot

5. Sắp xếp nhanh (Quick sort)

- Worst case:
 - Số phép so sánh cần thực hiện $\sim n^2/2$
- Average Case: số phép so sánh cần thực hiện $\sim 1.39n \log n$
 - Số phép so sánh cần thực hiện nhiều hơn $\sim 39\%$ so với sắp xếp trộn trong trường hợp tồi nhất
 - Nhưng nhanh hơn sắp xếp trộn vì ít phải di chuyển các phần tử

Ước tính thời gian chạy

- Home computer: giả thiết có thể thực hiện 10^8 phép so sánh trong 1 giây
- Super computer: giả thiết máy tính có thể thực hiện 10^{12} phép so sánh trong 1 giây

	insertion sort (N^2)			mergesort ($N \log N$)			quicksort ($N \log N$)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Nội dung

- 1. Sắp xếp chèn (Insertion sort)
 - 2. Sắp xếp chọn (Selection sort)
 - 3. Sắp xếp nổi bọt (Bubble sort)
 - 4. Sắp xếp trộn (Merge sort)
 - 5. Sắp xếp nhanh (Quick sort)
 - 6. Sắp xếp vun đống (Heap sort)**
- $O(n^2)$
- $O(n \log n)$
- Divide and conquer

6. Sắp xếp vun đống (Heap sort)

6.1. Cấu trúc dữ liệu đống (heap)

6.2. Sắp xếp vun đống

6.3. Hàng đợi có ưu tiên (priority queue)

6. Sắp xếp vun đống (Heap sort)

6.1. Cấu trúc dữ liệu đống (heap)

6.2. Sắp xếp vun đống

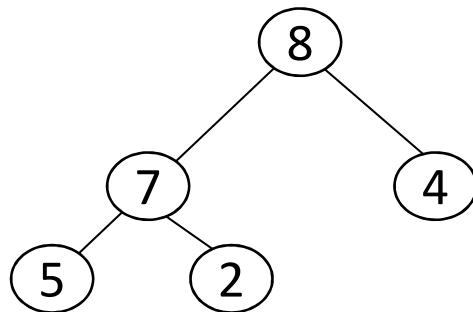
6.3. Hàng đợi có ưu tiên (priority queue)

Cấu trúc dữ liệu đống

- **Định nghĩa:** Đống (heap) là cây nhị phân có hai tính chất sau:
 - **Tính cấu trúc (Structural property):** tất cả các mức đều là đầy, ngoại trừ mức cuối cùng, mức cuối được điền từ trái sang phải
 - **Tính có thứ tự hay tính chất đống (heap property):** với mỗi nút x

$$\text{Parent}(x) \geq x : \text{max-heap}$$

OR $\text{Parent}(x) \leq x : \text{min-heap}$



Từ tính chất đống, suy ra:
“Gốc chứa phần tử lớn nhất của max-heap!”

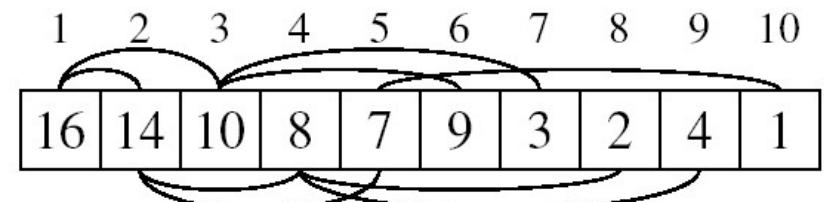
MAX-Heap

Đống là cây nhị phân được điền theo thứ tự

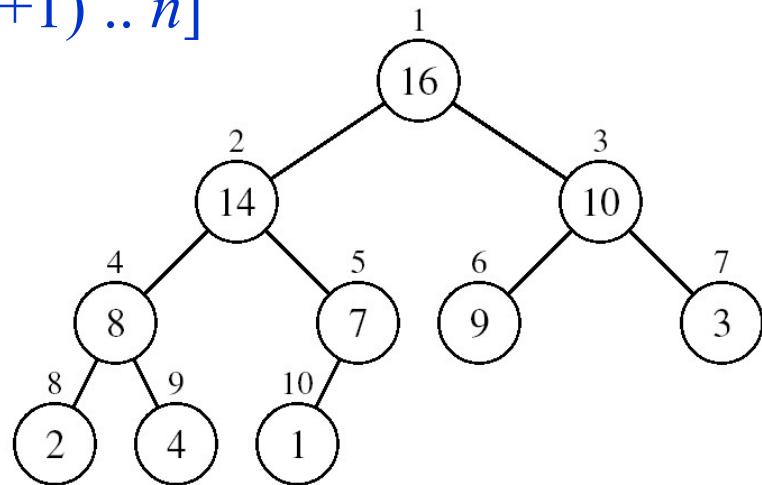
Cài đặt đống bởi mảng (array)

- Đống được cài đặt bởi mảng A .

- Gốc của cây là $A[1]$
- Con trái của $A[i] = A[2*i]$
- Con phải của $A[i] = A[2*i + 1]$
- Cha của $A[i] = A[\lfloor i/2 \rfloor]$
- Số lượng phần tử Heapsize[A] $\leq \text{length}[A]$



- Các phần tử thuộc mảng con $A[\lfloor n/2 \rfloor + 1] .. n]$ đều là lá của cây



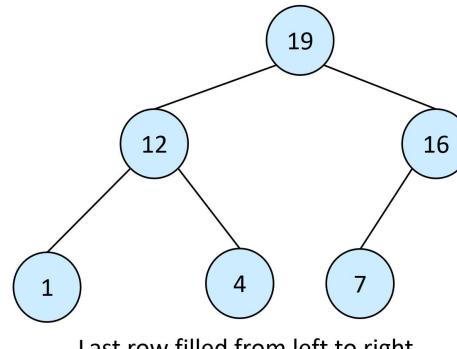
Hai dạng đống

- Đống max (Max Heap) – phần tử lớn nhất ở gốc
 - Sử dụng để sắp xếp mảng theo thứ tự tăng dần
 - Có tính chất max-heap: với mọi nút i , ngoại trừ gốc:
 $A[\text{PARENT}(i)] \geq A[i]$
- Đống min (Min Heap) – phần tử nhỏ nhất ở gốc
 - Sử dụng để sắp xếp mảng theo thứ tự giảm dần
 - Có tính chất min-heap: với mọi nút i , ngoại trừ gốc:
 $A[\text{PARENT}(i)] \leq A[i]$

MAX-HEAP EXAMPLE:

19	12	16	1	4	7
----	----	----	---	---	---

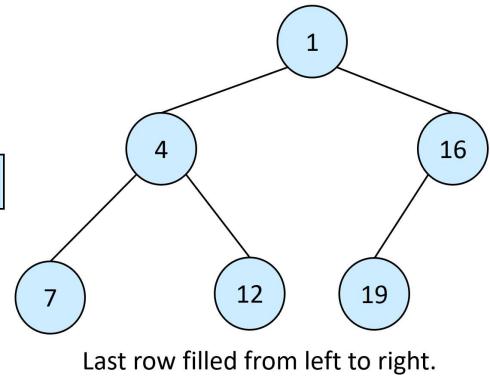
Max-heap as a binary tree:



MIN-HEAP EXAMPLE:

1	4	16	7	12	19
---	---	----	---	----	----

Min-heap as a binary tree:



Các phép toán đối với đống (Operations on Heaps)

- Khôi phục tính chất max-heap (Vun lại đống)
 - MAX-HEAPIFY
- Tạo max-heap từ một mảng không được sắp xếp
 - BUILD-MAX-HEAP

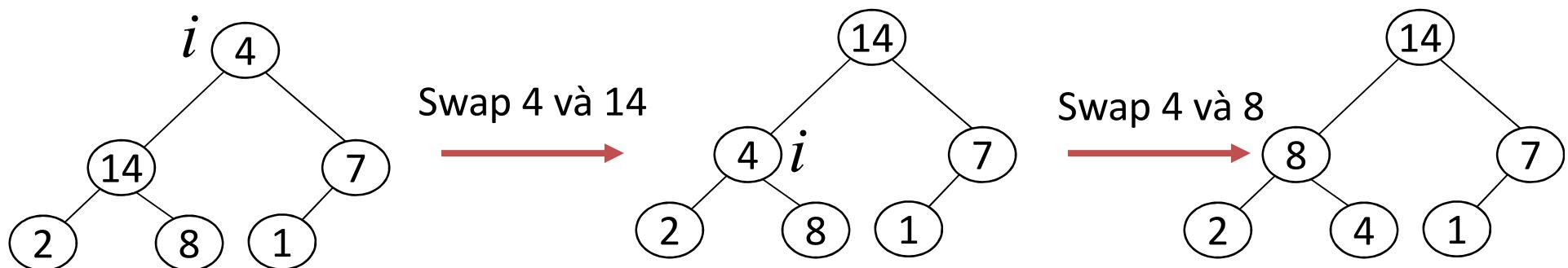
Các phép toán đối với đống (Operations on Heaps)

- Khôi phục tính chất max-heap (Vun lại đống)
 - MAX-HEAPIFY
- Tạo max-heap từ một mảng không được sắp xếp
 - BUILD-MAX-HEAP

Max-heap: MAX-HEAPIFY

– Tính chất max-heap: với mọi nút i , ngoại trừ gốc:
 $A[\text{PARENT}(i)] \geq A[i]$

- Giả sử có nút i với giá trị bé hơn con của nó (đã giả thiết là cây con trái và cây con phải của nút i đều là max-heaps), để loại bỏ sự vi phạm tính chất max-heap này, ta gọi thủ tục **MAX-HEAPIFY** tiến hành thực hiện cách bước sau:
 - Đổi chỗ nút i với nút con lớn hơn
 - Di chuyển xuống theo cây
 - Tiếp tục quá trình cho đến khi nút không còn bé hơn con



Không phải max-heap vì:

Tính chất Max-heap bị vi phạm
tại nút i

Không phải max-heap vì:

Tính chất Max-heap bị vi phạm
tại nút i

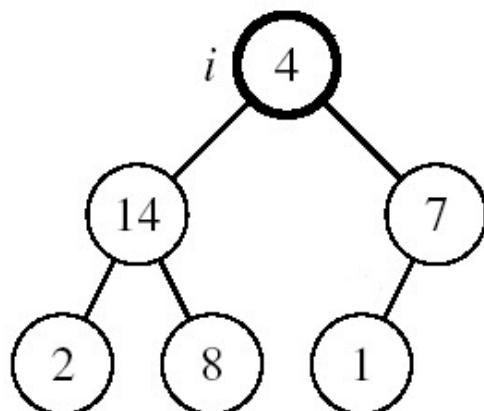
Max heap

MAX-HEAPIFY: khôi phục tính chất Max-Heap

- Giả thiết:
 - Cây con trái và phải của nút i đều là max-heaps
 - $A[i] <$ có thể nhỏ hơn các con của nó

Alg: MAX-HEAPIFY(A, i, n)

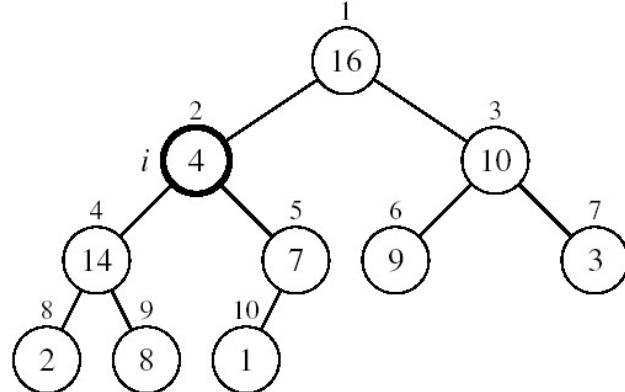
1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. **if** $l \leq n$ and $A[l] > A[i]$
4. **then** $\text{largest} \leftarrow l$
5. **else** $\text{largest} \leftarrow i$
6. **if** $r \leq n$ and $A[r] > A[\text{largest}]$
7. **then** $\text{largest} \leftarrow r$
8. **if** $\text{largest} \neq i$
9. **then** exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}, n$)



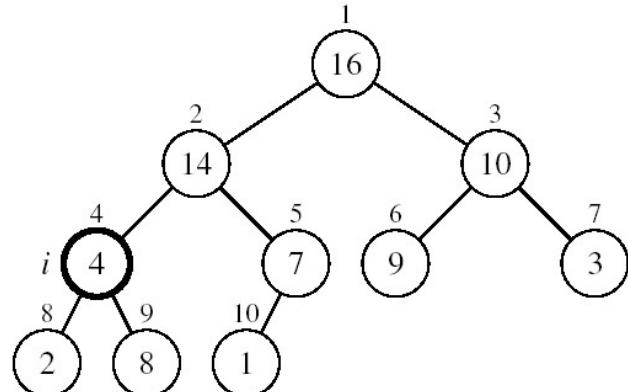
Ví dụ

MAX-HEAPIFY tiến hành thực hiện cách bước sau:

- Đổi chỗ nút i với nút con lớn hơn
- Di chuyển xuống theo cây
- Tiếp tục quá trình cho đến khi nút không còn bé hơn con



$A[2] \leftrightarrow A[4]$



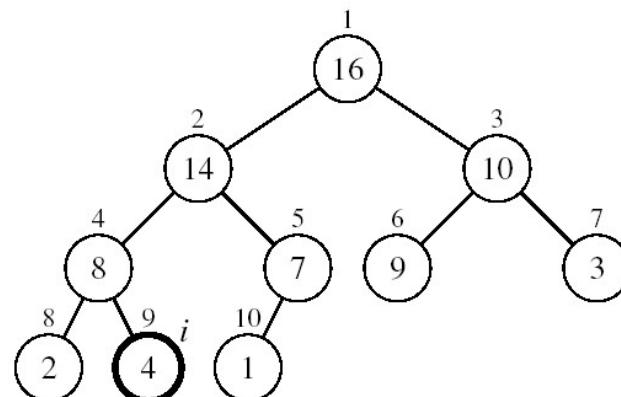
$A[4]$ vi phạm tính chất max-heap

$A[2]$ vi phạm tính chất max-heap

→ GỌI: **MAX-HEAPIFY(A, 2, 10);**

Để khôi phục tính chất max-heap

$A[4] \leftrightarrow A[9]$



MAX-HEAPIFY(A, 2, 10) kết thúc; và ta thu được một Max heap

Tính chất max-heap được khôi phục

Các phép toán đối với đống (Operations on Heaps)

- Khôi phục tính chất max-heap (Vun lại đống)
 - MAX-HEAPIFY
- **Tạo max-heap từ một mảng không được sắp xếp**
 - BUILD-MAX-HEAP

Tạo max-heap từ một mảng không được sắp xếp: BUILD-MAX-HEAP

Biến đổi mảng $A[1 \dots n]$ thành max-heap ($n = \text{length}[A]$) sao cho các phần tử thuộc mảng con $A[\lfloor n/2 \rfloor + 1] \dots n]$ là các lá:

- Do đó, để tạo đống, ta chỉ cần áp dụng MAX-HEAPIFY đối với các phần tử từ 1 đến $\lfloor n/2 \rfloor$

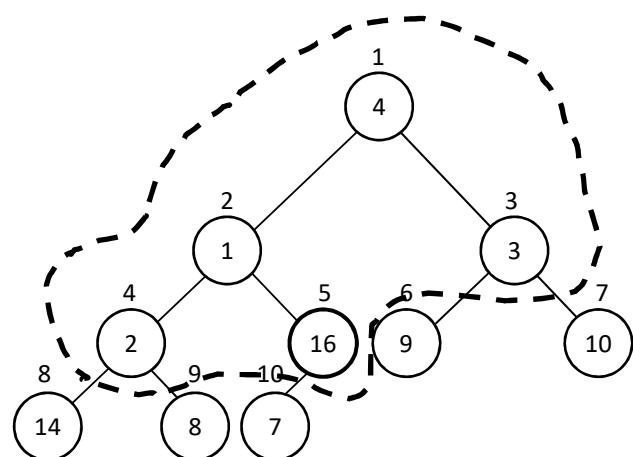
Alg: BUILD-MAX-HEAP(A)

1. $n = \text{length}[A]$
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)

Áp dụng MAX-HEAPIFY trên các nút trong $A[\lfloor n/2 \rfloor] \dots A[1]$

$A:$

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

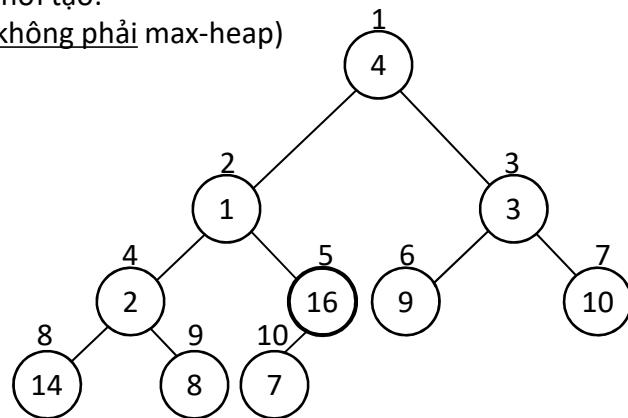


Ví dụ: cho mảng A, xây dựng max-heap biểu diễn A

A:

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

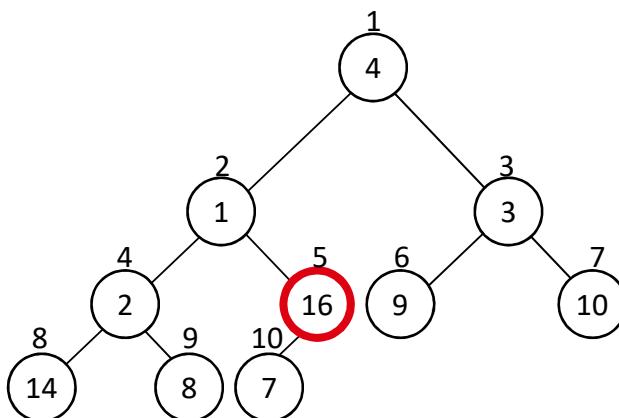
Khởi tạo:
(không phải max-heap)



Để chuyển cây này thành max-heap: ta cần áp dụng MAX-HEAPIFY trên tất cả các nút trong:
A[$\lfloor n/2 \rfloor$] ...A[1]

$$\lfloor 10/2 \rfloor = 5$$

i=5: Gọi MAX-HEAPIFY(A,5,10)



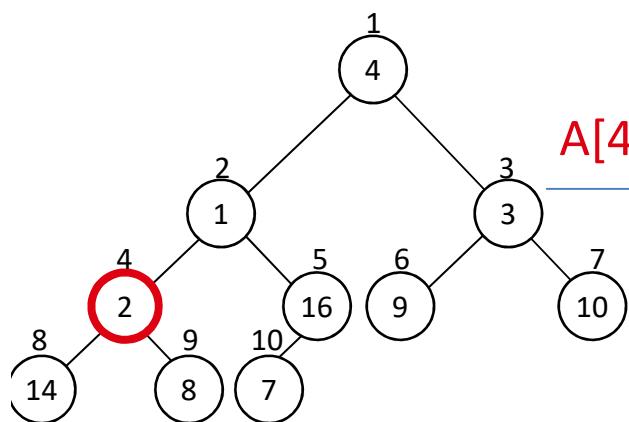
A[5] = 16 không lớn hơn các con của nó (A[10]=7)
→ ok; không cần làm gì

Ví dụ: cho mảng A, xây dựng max-heap biểu diễn A

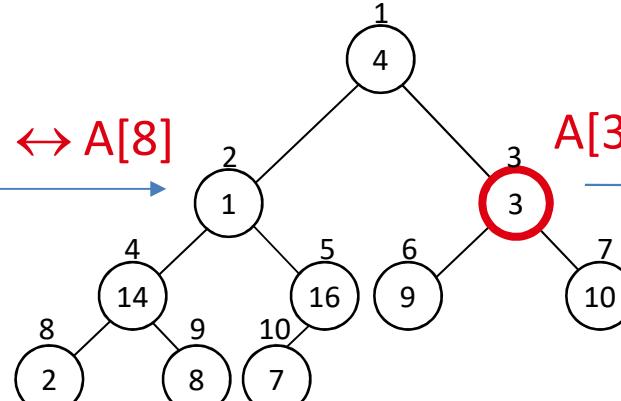
A:

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

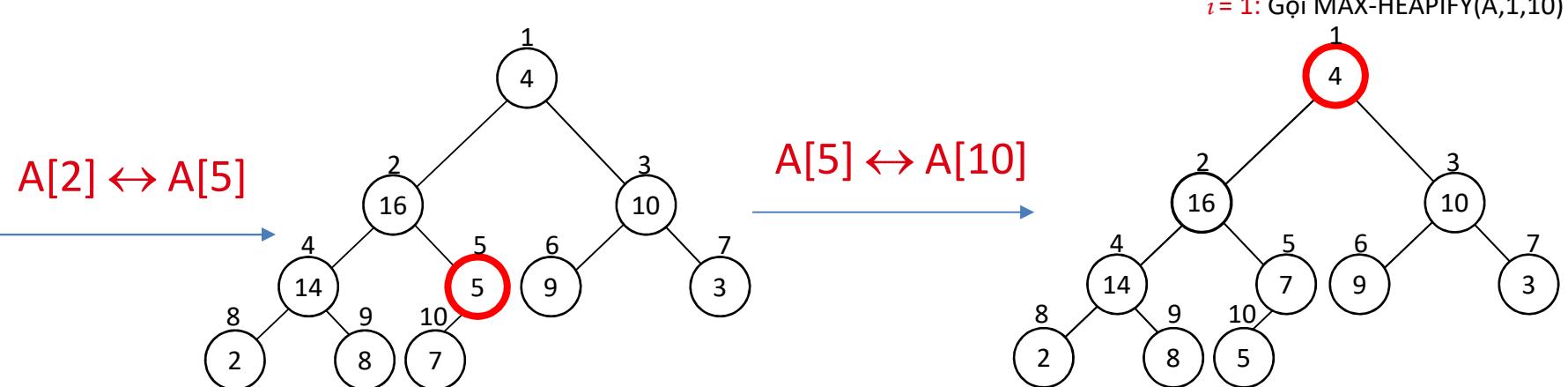
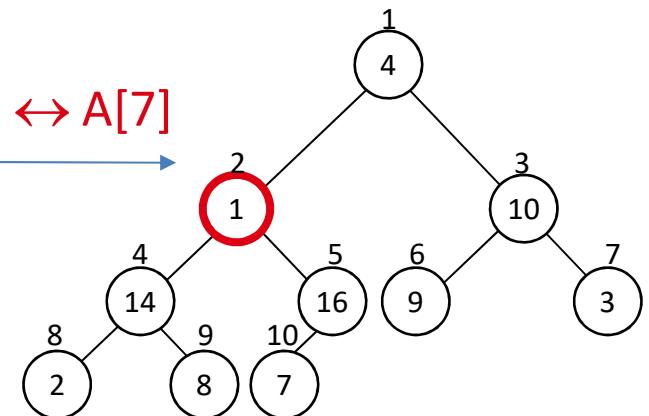
i = 4: Gọi MAX-HEAPIFY(A,4,10)



i = 3: Gọi MAX-HEAPIFY(A,3,10)



i = 2: Gọi MAX-HEAPIFY(A,2,10)

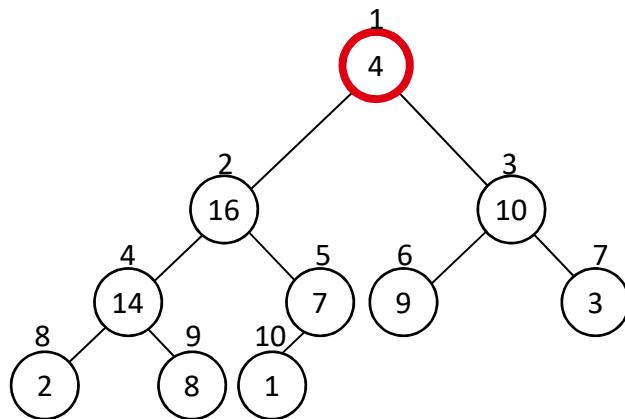


Ví dụ: cho mảng A, xây dựng max-heap biểu diễn A

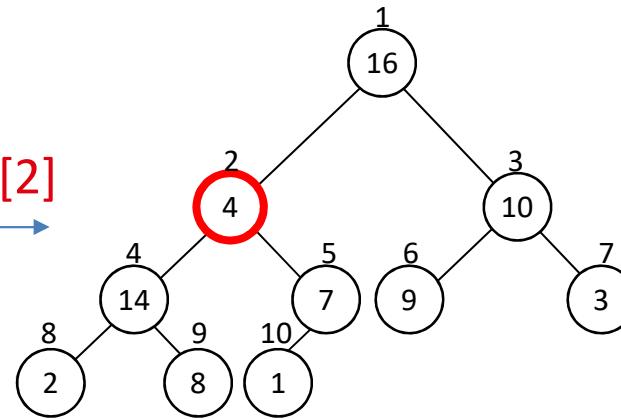
A:

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

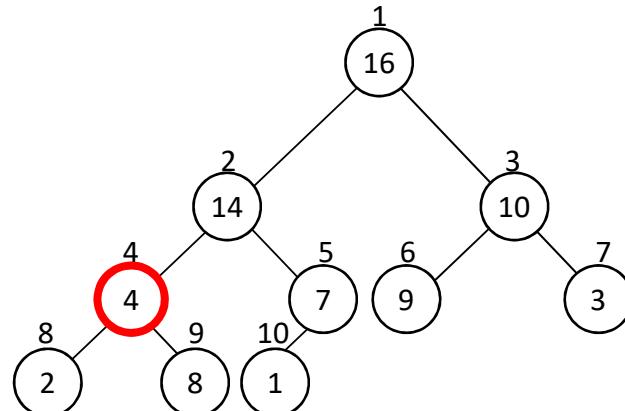
i = 1: Gọi MAX-HEAPIFY(A,1,10)



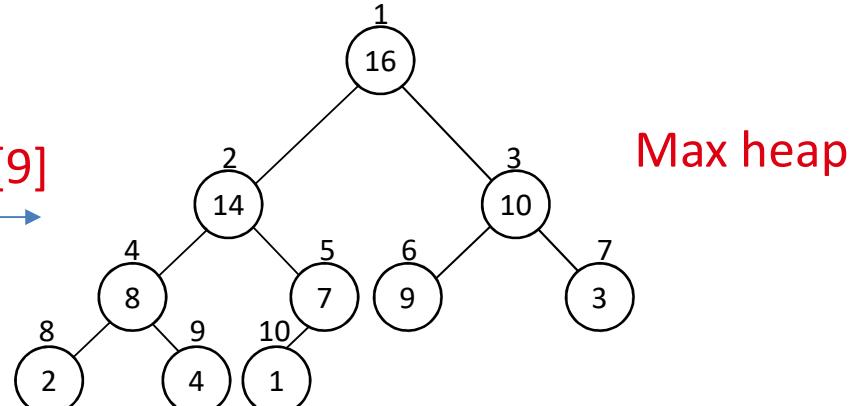
$A[1] \leftrightarrow A[2]$



$A[2] \leftrightarrow A[4]$



$A[4] \leftrightarrow A[9]$



6. Sắp xếp vun đống (Heap sort)

6.1. Cấu trúc dữ liệu đống (heap)

6.2. Sắp xếp vun đống

6.3. Hàng đợi có ưu tiên (priority queue)

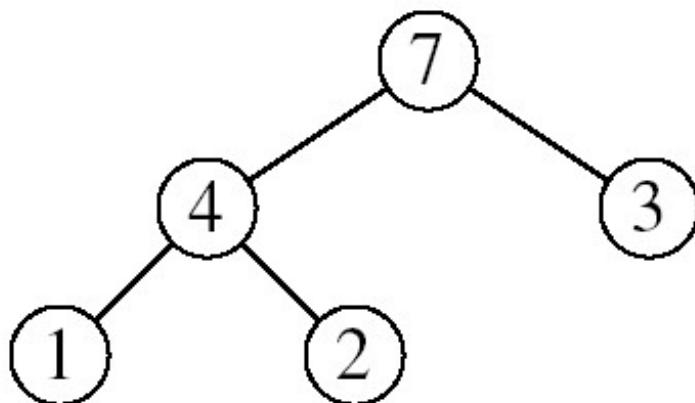
Sắp xếp vun đống

Mục đích: Sắp xếp mảng theo thứ tự tăng dần nhờ sử dụng đống

Sơ đồ thuật toán:

1. Sử dụng **BUILD-MAX-HEAP** để tạo đống max-heap từ mảng đã cho $A[1 \dots n]$.
2. Vì phần tử lớn nhất của mảng được lưu ở gốc $A[1]$: ta cần di chuyển nó về đúng vị trí của nó trong mảng: đổi chỗ gốc với phần tử cuối cùng của mảng $A[n]$.
3. Loại bỏ nút cuối n ra khỏi đống max-heap bằng cách giảm kích thước đống đi 1.
4. Phần tử mới đang nằm ở gốc có thể vi phạm tính chất max-heap. Vì vậy, cần gọi thủ tục **MAX-HEAPIFY** đối với nút gốc mới này để khôi phục tính chất max-heap.

Lặp lại quá trình (2-3-4) cho đến khi đống chỉ còn lại 1 nút

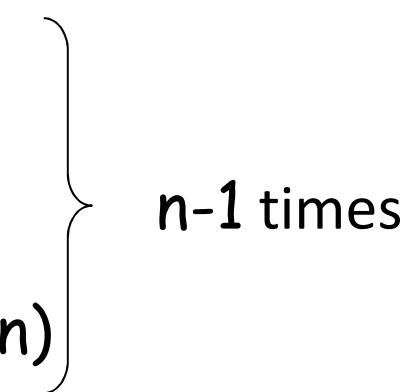


Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. $n = \text{length}[A]$
3. **for** $i \leftarrow n$ **downto** 2 {
4. **exchange** $A[1] \leftrightarrow A[i]$
5. MAX-HEAPIFY($A, 1, i - 1$)
6. }

Alg: HEAPSORT(A)

Alg: HEAPSORT(A)

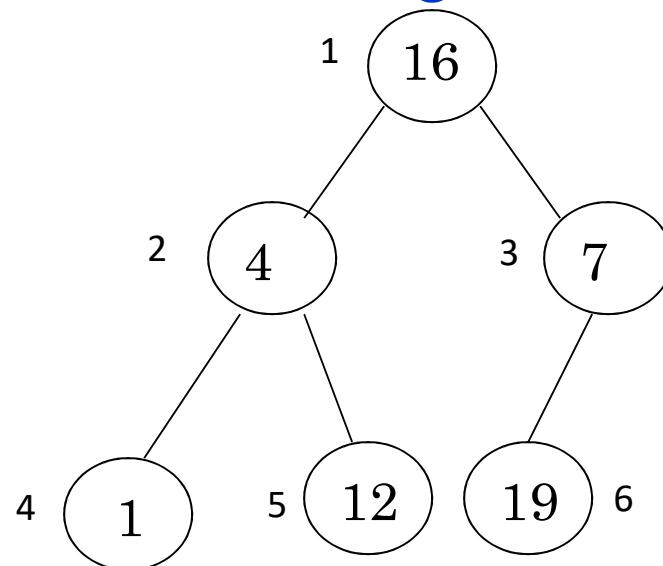
1. BUILD-MAX-HEAP(A) $O(n)$
 2. $n = \text{length}[A]$
 3. **for** $i \leftarrow n$ **downto** 2 {
4. exchange $A[1] \leftrightarrow A[i]$
5. MAX-HEAPIFY($A, 1, i - 1$) $O(\log_2 n)$
6. }
- Thời gian tính: $O(n \log_2 n)$
- 

Ví dụ: sắp xếp mảng A theo thứ tự tăng dần

A:

16	4	7	1	12	19
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]

- Bước 1: Biểu diễn mảng A bởi cây nhị phân đầy đủ



Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. $n = \text{length}[A]$
3. **for** $i \leftarrow n$ **downto** 2 {
4. exchange $A[1] \leftrightarrow A[i]$
5. MAX-HEAPIFY($A, 1, i - 1$)
6. }

- Bước 2: Gọi HEAPSORT(A):

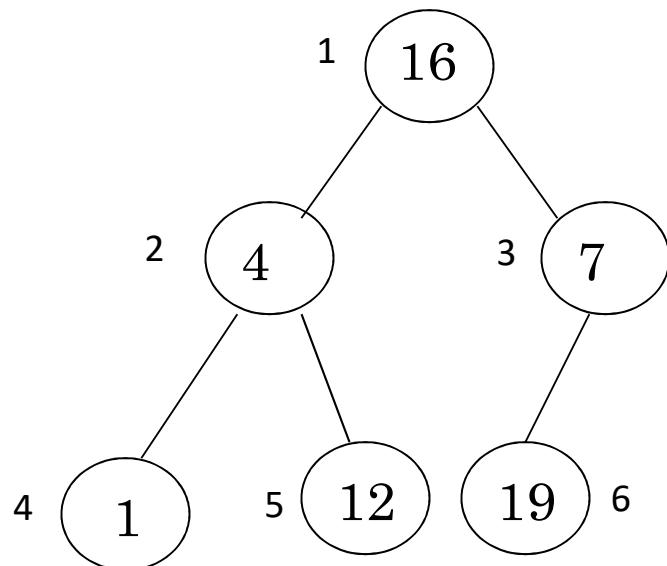
1. Gọi BUILD-MAX-HEAP(A): để đưa cây này về đống max-heap

Ví dụ: sắp xếp mảng A theo thứ tự tăng dần

- Bước 2: Gọi HEAPSORT(A):

- BUILD-MAX-HEAP(A)

- $n = 6$



Alg: BUILD-MAX-HEAP(A)

- $n = \text{length}[A]$
- $\text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto } 1$
- $\text{do MAX-HEAPIFY}(A, i, n)$

Để đưa cây này về dạng đống max-heap: ta cần áp dụng MAX-HEAPIFY trên tất cả các nút trong:
 $A[\lfloor n/2 \rfloor] \dots A[1]$

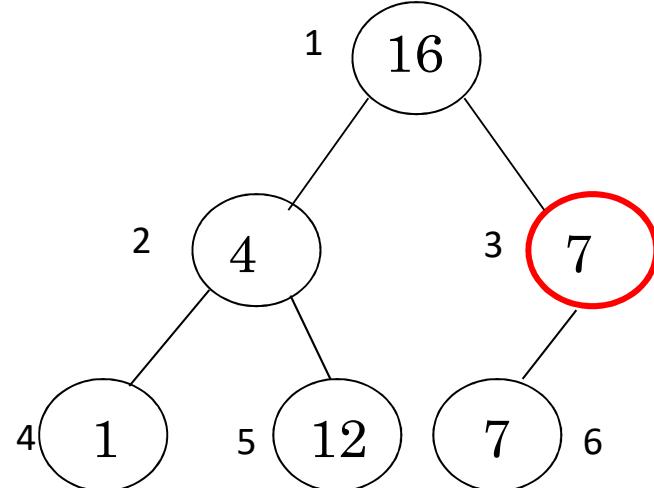
$$\lfloor 6/2 \rfloor = 3$$

BUILD-MAX-HEAP(A): gọi MAX-HEAPIFY trên tất cả các nút trong

MAX-HEAPIFY tiến hành thực hiện cách bước sau:

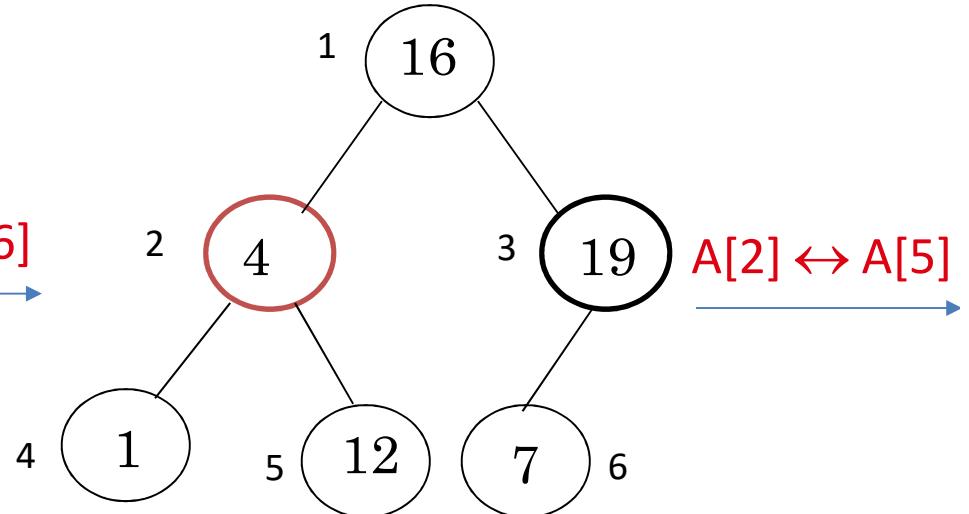
- Đổi chỗ nút i với nút con lớn hơn
- Di chuyển xuống theo cây
- Tiếp tục quá trình cho đến khi nút không còn bé hơn con

i = 3: Gọi MAX-HEAPIFY(A,3,6)

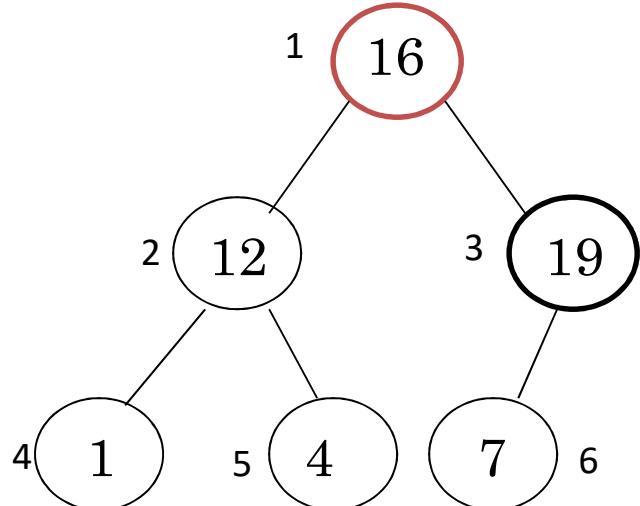


A[3] ↔ A[6]

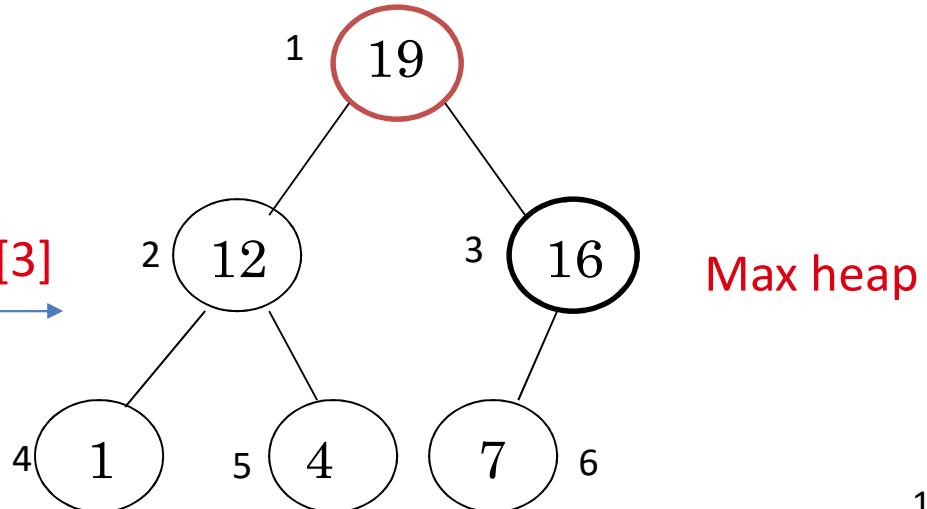
i = 2: Gọi MAX-HEAPIFY(A,2,6)



i = 1: Gọi MAX-HEAPIFY(A,1,6)



A[1] ↔ A[3]



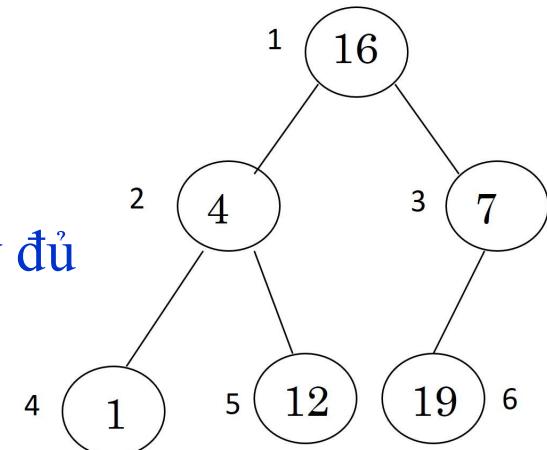
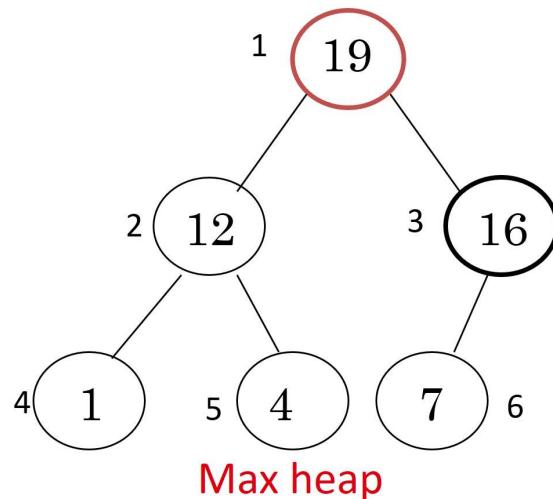
Ví dụ: sắp xếp mảng A theo thứ tự tăng dần

A:

16	4	7	1	12	19
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]

- Bước 1: Biểu diễn mảng A thành cây nhị phân đầy đủ
- Bước 2: Gọi HEAPSORT(A):

1. Gọi BUILD-MAX-HEAP(A): để biến đổi cây này về đống max-heap



Alg: HEAPSORT(A)

```
1. BUILD-MAX-HEAP(A)
2. n = length[A]
3. for i ← n downto 2 {
4.     exchange A[1] ↔ A[i]
5.     MAX-HEAPIFY(A, 1, i - 1)
6. }
```

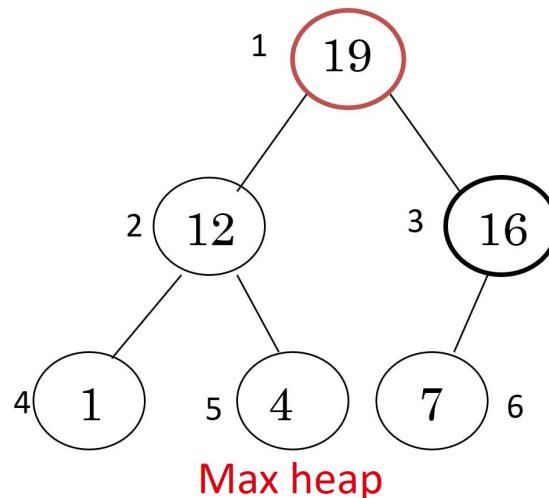
Ví dụ: sắp xếp mảng A theo thứ tự tăng dần

A:

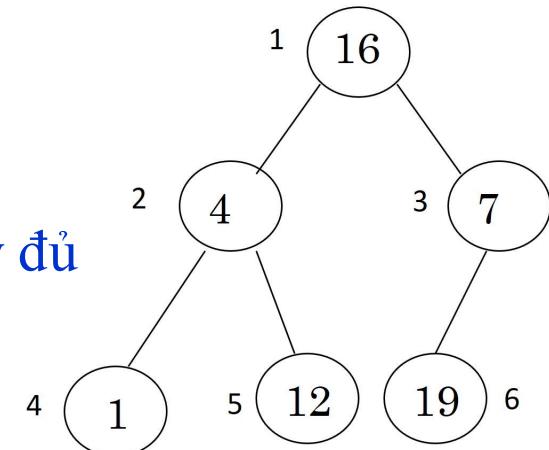
16	4	7	1	12	19
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]

- Bước 1: Biểu diễn mảng A thành cây nhị phân đầy đủ
- Bước 2: Gọi HEAPSORT(A):

1. Gọi BUILD-MAX-HEAP(A): để biến đổi cây này về đống max-heap



2. Thực hiện bước lines 3-6

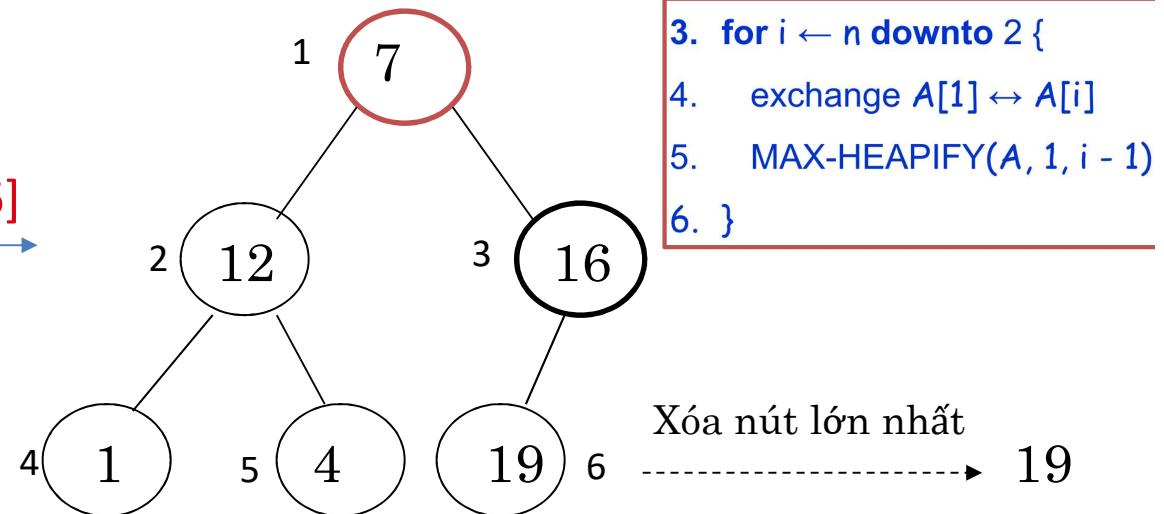
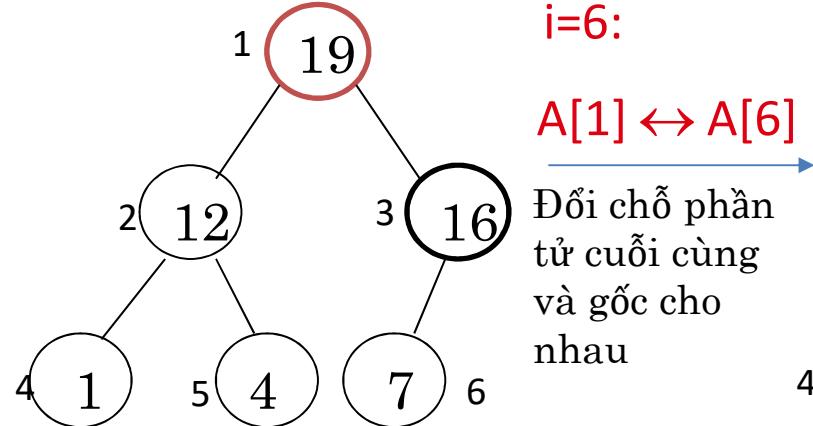


Alg: HEAPSORT(A)

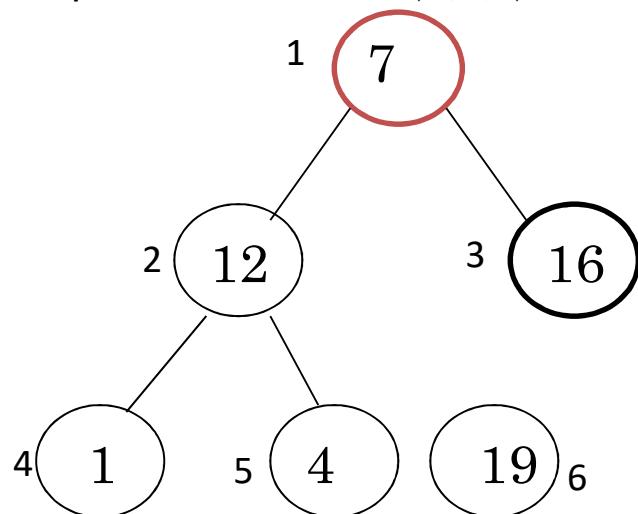
1. BUILD-MAX-HEAP(A)
2. $n = \text{length}[A]$
3. **for** $i \leftarrow n$ **downto** 2 {
4. exchange $A[1] \leftrightarrow A[i]$
5. MAX-HEAPIFY($A, 1, i - 1$)
6. }

Ví dụ:

Max-heap:



Gọi MAX-HEAPIFY(A,1,5)



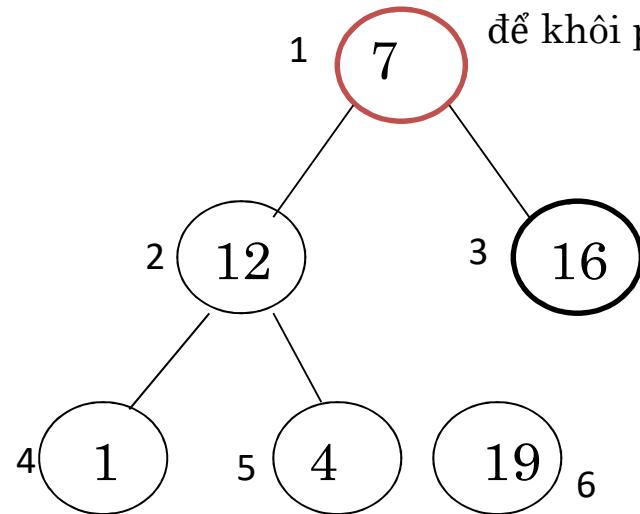
Array A:

7	12	16	1	4	19
---	----	----	---	---	----

Sorted:

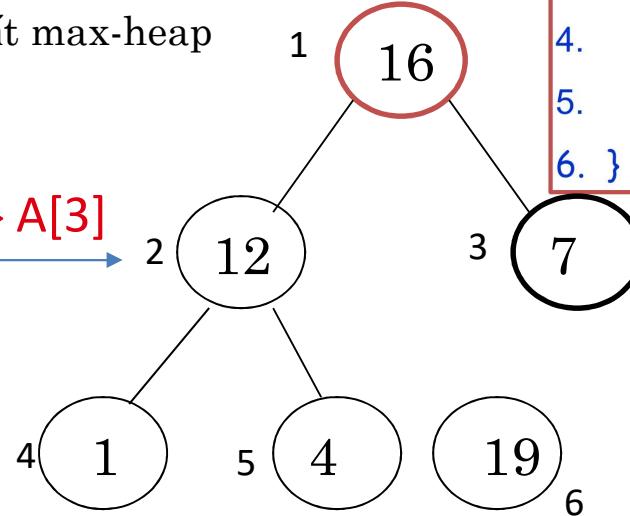
Ví dụ:

Gọi MAX-HEAPIFY($A, 1, 5$)



để khôi phục tính chất max-heap

$A[1] \leftrightarrow A[3]$



Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A)

2. $n = \text{length}[A]$

3. **for $i \leftarrow n$ down to 2 {**

4. **exchange $A[1] \leftrightarrow A[i]$**

5. **MAX-HEAPIFY($A, 1, i - 1$)**

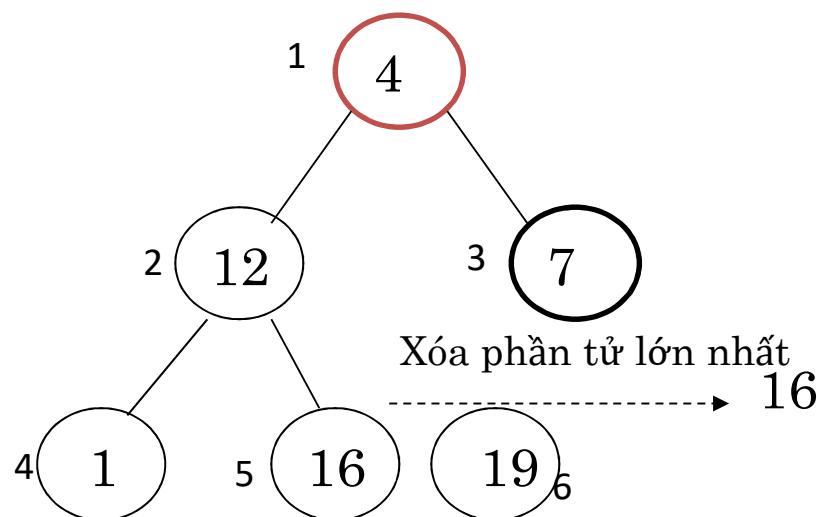
6. }

$i=5$:

$A[1] \leftrightarrow A[5]$

Đổi chỗ phần tử cuối và gốc cho nhau

Gọi MAX-HEAPIFY($A, 1, 4$)



Xóa phần tử lớn nhất

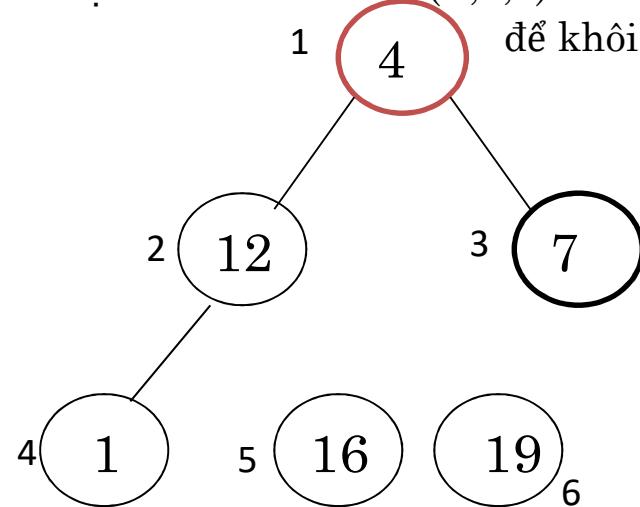
Array A:

4	12	7	1	16	19
---	----	---	---	----	----

Sorted:

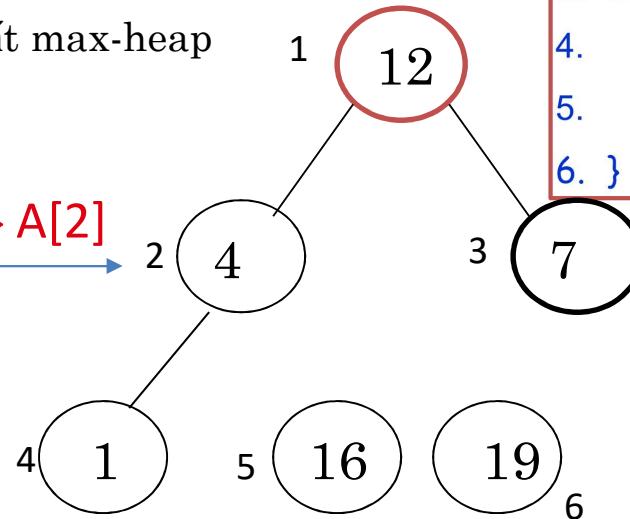
Ví dụ

Gọi MAX-HEAPIFY($A, 1, 4$)



để khôi phục tính chất max-heap

$A[1] \leftrightarrow A[2]$



Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A)

2. $n = \text{length}[A]$

3. **for** $i \leftarrow n$ **downto** 2 {

4. **exchange** $A[1] \leftrightarrow A[i]$

5. MAX-HEAPIFY($A, 1, i - 1$)

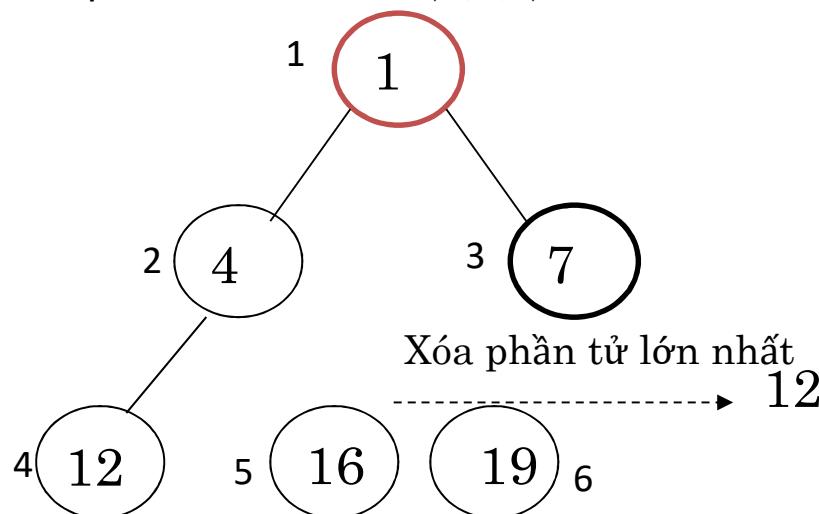
6. }

$i=4$:

$A[1] \leftrightarrow A[4]$

Đổi chỗ phần tử cuối và gốc cho nhau

Gọi MAX-HEAPIFY($A, 1, 3$)



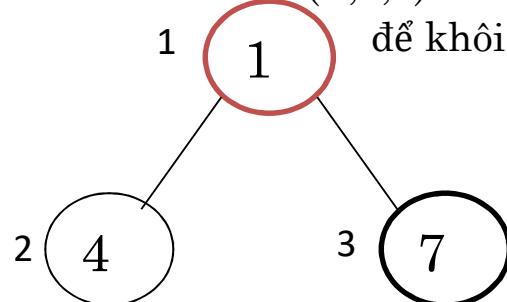
Array A:

1	4	7	12	16	19
---	---	---	----	----	----

Sorted:

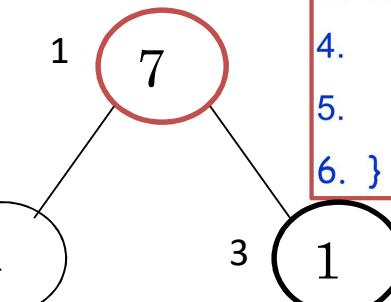
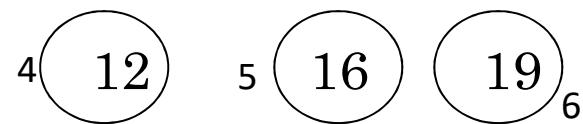
Ví dụ:

Call MAX-HEAPIFY($A, 1, 3$)



để khôi phục tính chất max-heap

$A[1] \leftrightarrow A[3]$



Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A)

2. $n = \text{length}[A]$

3. **for $i \leftarrow n$ down to 2 {**

4. **exchange $A[1] \leftrightarrow A[i]$**

5. **MAX-HEAPIFY($A, 1, i - 1$)**

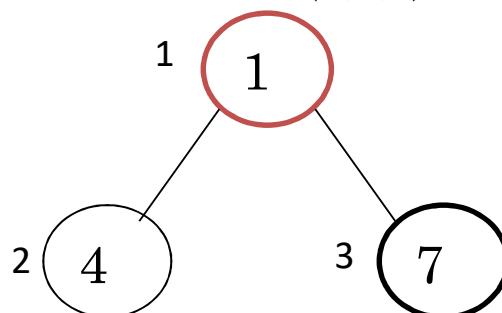
6. }

$i=3:$

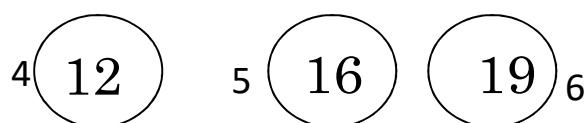
$A[1] \leftrightarrow A[3]$

Đổi chỗ phần tử cuối và gốc cho nhau

Call MAX-HEAPIFY($A, 1, 2$)



Xóa phần tử lớn nhất
7



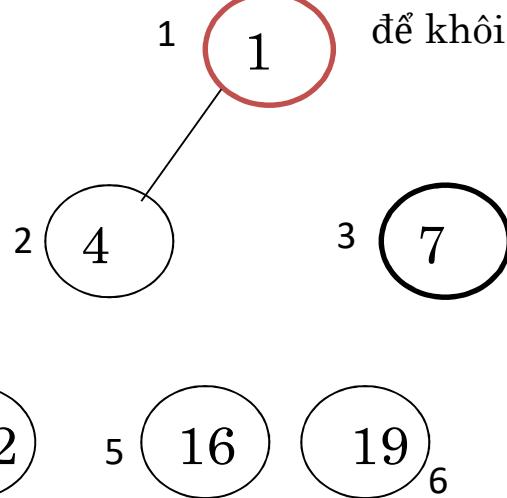
Array A:

1	4	7	12	16	19
---	---	---	----	----	----

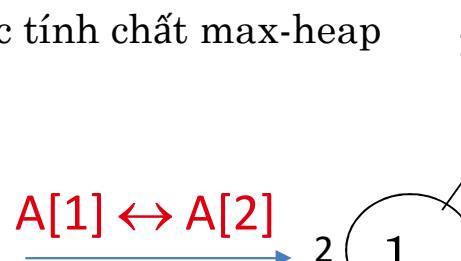
Sorted:

Ví dụ:

Gọi MAX-HEAPIFY($A, 1, 2$)



để khôi phục tính chất max-heap



Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A)

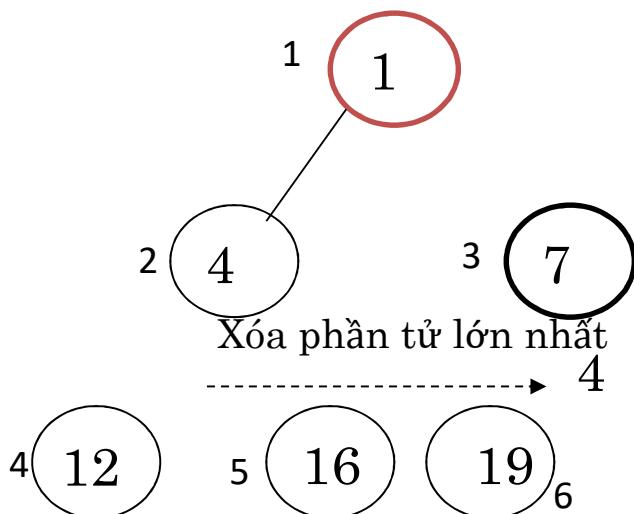
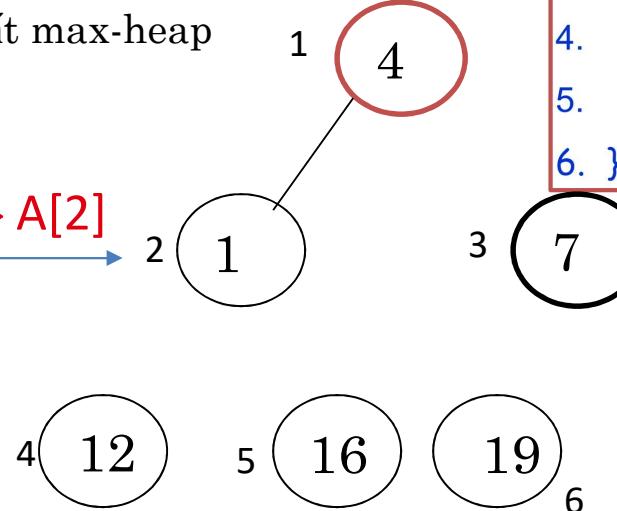
2. $n = \text{length}[A]$

3. **for $i \leftarrow n$ down to 2 {**
 4. **exchange $A[1] \leftrightarrow A[i]$**
 5. **MAX-HEAPIFY($A, 1, i - 1$)**
 6. **}**

$i=2:$

$A[1] \leftrightarrow A[2]$

Đổi chỗ phần tử cuối và gốc cho nhau



Mảng A:



Sorted:

Thu được mảng A được xếp theo thứ tự tăng dần

Bài tập 1: sử dụng thuật toán heap sort

- Sắp xếp các phần tử của mảng A theo thứ tự tăng dần

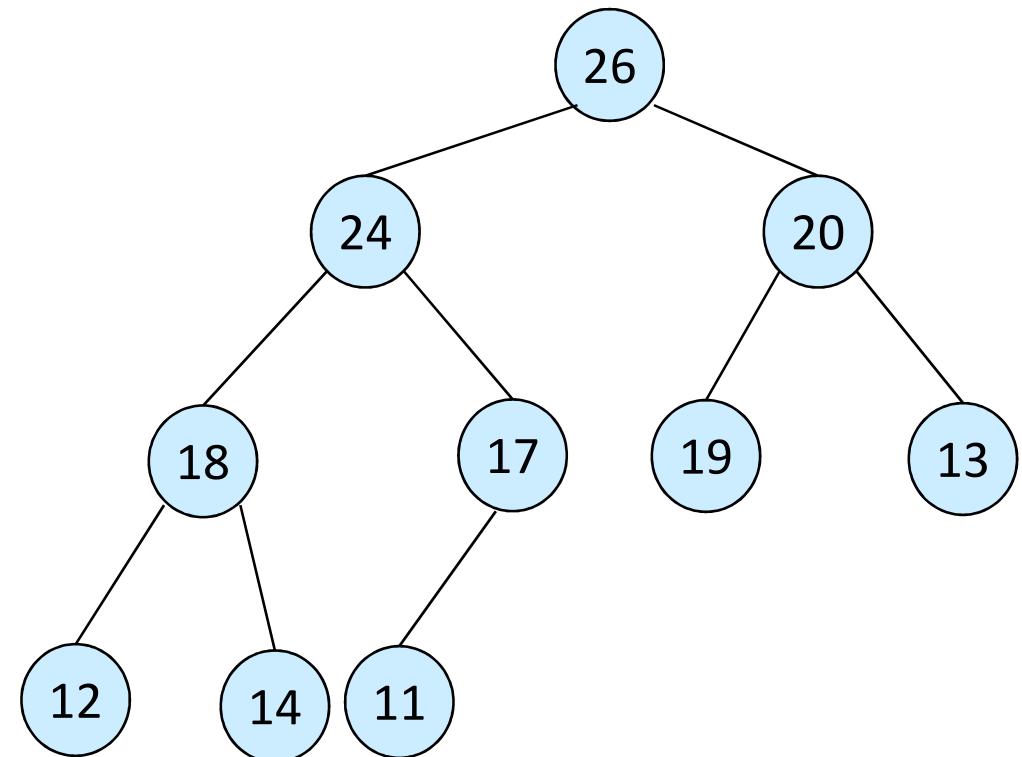
A:	7	4	3	1	2
	1	2	3	4	5

Bài tập 2: sử dụng thuật toán heap sort

- Sắp xếp các phần tử của mảng A theo thứ tự tăng dần

A:

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10



6. Sắp xếp vun đống (Heap sort)

6.1. Cấu trúc dữ liệu đống (heap)

6.2. Sắp xếp vun đống

6.3. Hàng đợi có ưu tiên (priority queue)

6.3. Hàng đợi có ưu tiên - Priority Queues

- Cho tập S thường xuyên biến động, mỗi phần tử x được gán với một giá trị gọi là khoá (hay độ ưu tiên). Cần một cấu trúc dữ liệu hỗ trợ hiệu quả các thao tác chính sau:
 - Insert (S, x) : Bổ sung phần tử x vào S
 - Max (S) : trả lại phần tử lớn nhất
 - Extract-Max (S) : loại bỏ và trả lại phần tử lớn nhất
 - Increase-Key (S, x, k) : tăng khoá của x thành k

Cấu trúc dữ liệu đáp ứng các yêu cầu đó là **hàng đợi có ưu tiên**.

Hàng đợi có ưu tiên có thể tổ chức nhờ sử dụng cấu trúc dữ liệu đống để cất giữ các khoá.

- **Chú ý:** Có thể thay "max" bởi "min".

Các phép toán đối với hàng đợi có ưu tiên

Operations on Priority Queues

- Hàng đợi có ưu tiên (max) có các phép toán cơ bản sau:
 - **Insert (S, x)**: bổ sung phần tử x vào tập S
 - **Extract-Max (S)**: loai bỏ và trả lại phần tử của S với khoá lớn nhất
 - **Maximum (S)**: trả lại phần tử của S với khoá lớn nhất
 - **Increase-Key (S, x, k)**: tăng giá trị của khoá của phần tử x lên thành k
(Giả sử $k \geq$ khoá hiện tại của x)

Các phép toán đối với hàng đợi có ưu tiên

Operations on Priority Queues

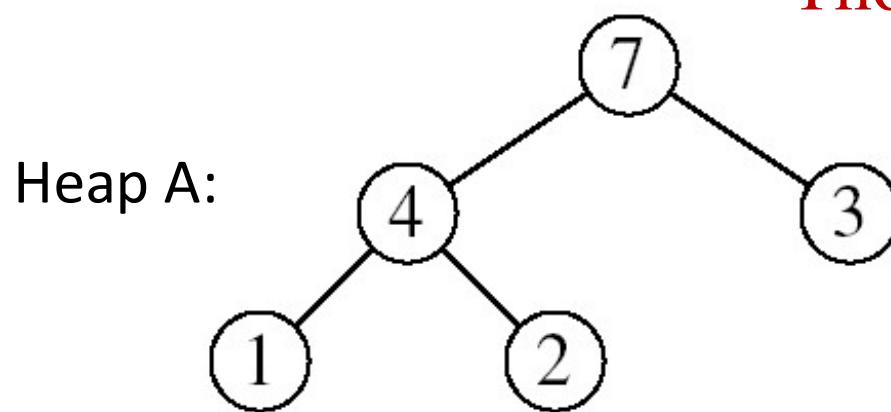
- Hàng đợi có ưu tiên (min) có các phép toán cơ bản sau:
 - **Insert (S, x)**: bổ sung phần tử x vào tập S
 - **Extract-Min (S)**: loai bỏ và trả lại phần tử của S với khoá nhỏ nhất
 - **Minimum (S)**: trả lại phần tử của S với khoá nhỏ nhất
 - **Decrease-Key (S, x, k)**: giảm giá trị của khoá của phần tử x xuống thành k (Giả sử $k \leq$ khoá hiện tại của x)

Phép toán HEAP-MAXIMUM

Chức năng: Trả lại phần tử lớn nhất của đống

Alg: Heap-Maximum(A)

1. **return** $A[1]$



Thời gian tính: $O(1)$

Heap-Maximum(A) trả lại 7

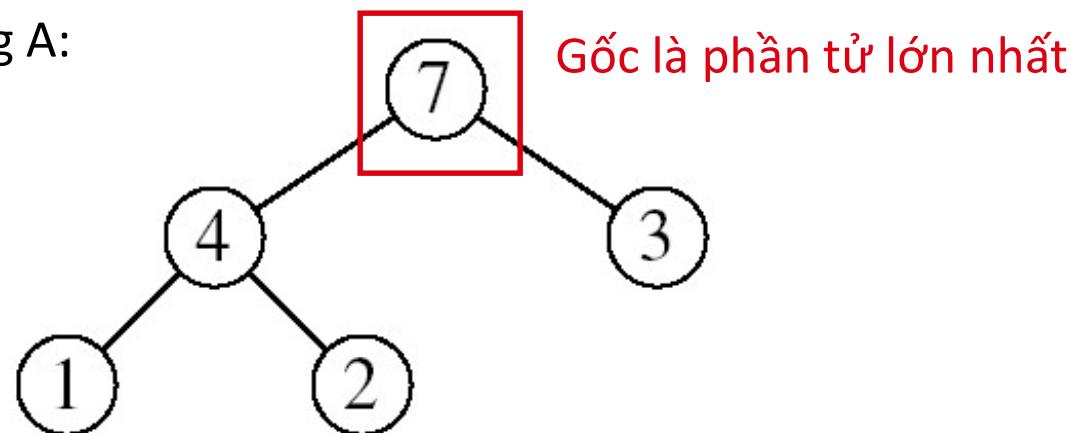
Phép toán Heap-Extract-Max

Chức năng: Trả lại phần tử lớn nhất và loại bỏ nó khỏi đống

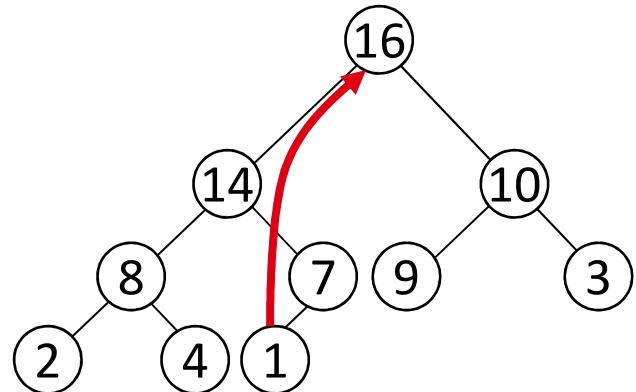
Thuật toán:

- Hoán đổi gốc với phần tử cuối cùng ($A[0]$ và $A[n-1]$)
- Giảm kích thước của đống đi 1
- Gọi Max-Heapify đối với gốc mới trên đống có kích thước $n-1$

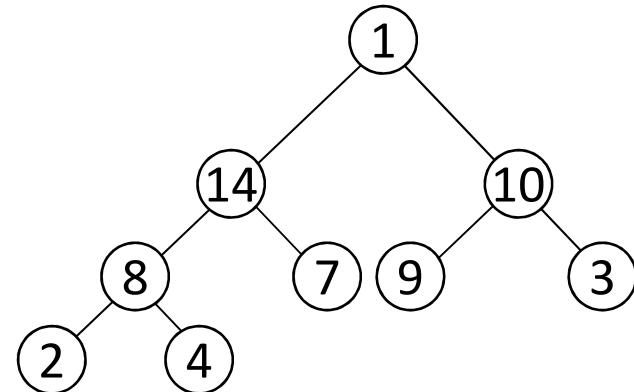
Đống A:



Ví dụ: Heap-Extract-Max

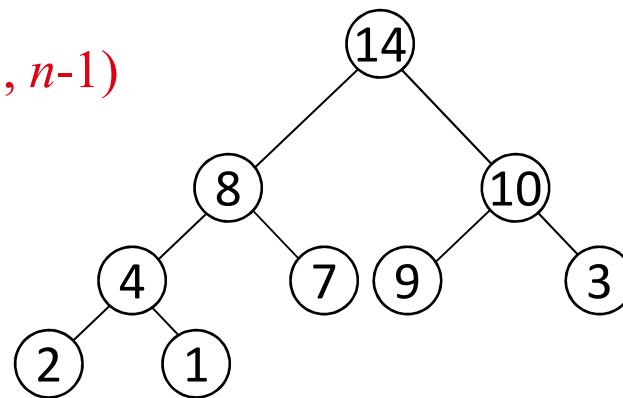


max = 16



Kích thước đống giảm đi 1

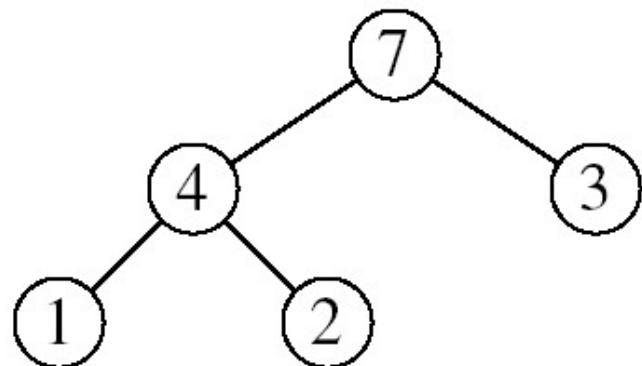
Thực hiện Max-Heapify($A, 1, n-1$)



Heap-Extract-Max

Alg: Heap-Extract-Max(A, n)

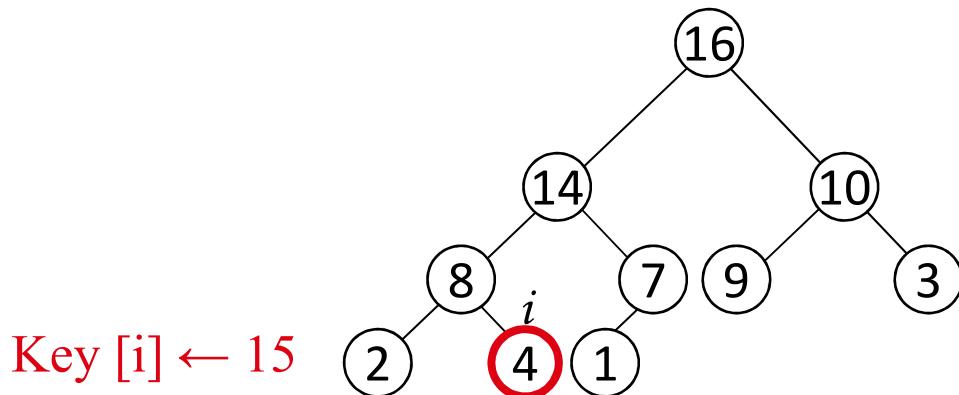
1. **if** $n < 1$
2. **then error** “heap underflow”
3. $\text{max} \leftarrow A[1]$
4. $A[1] \leftarrow A[n]$
5. Max-Heapify($A, 1, n-1$) // Vun lại đống
6. **return** max



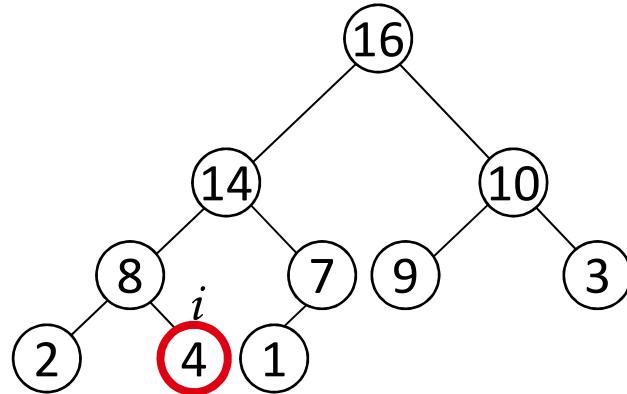
Thời gian tính: $O(\log n)$

Phép toán Heap-Increase-Key

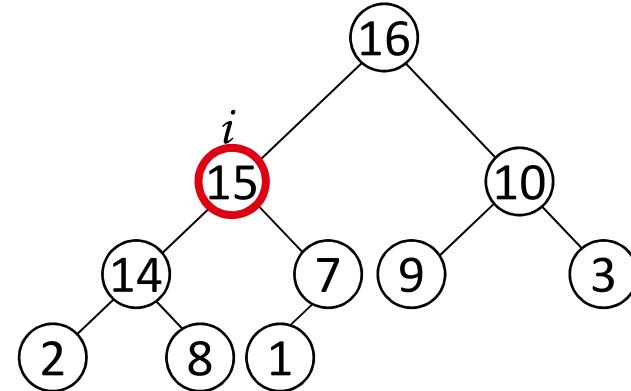
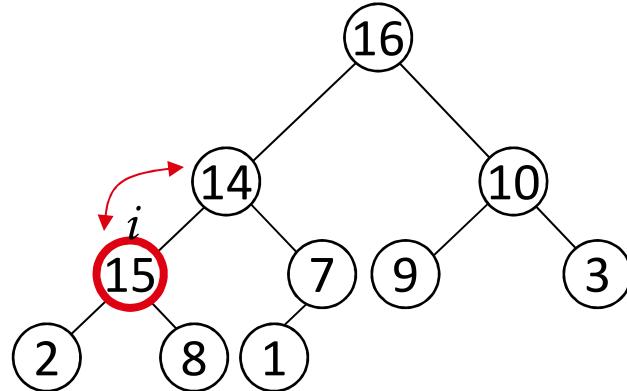
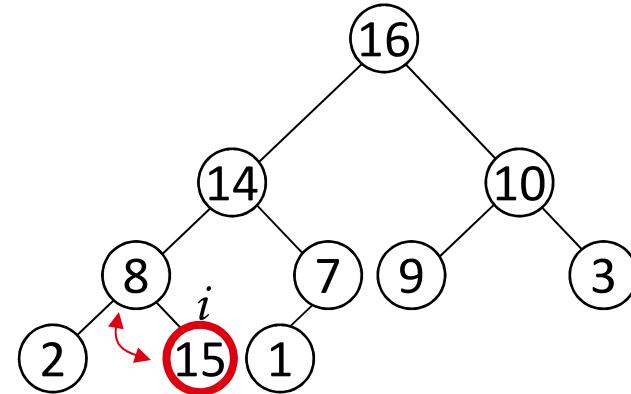
- **Chức năng:** Tăng giá trị khoá của phần tử i trong đống
- **Thuật toán:**
 - Tăng khoá của $A[i]$ thành giá trị mới
 - Nếu tính chất max-heap bị vi phạm: di chuyển theo đường đến gốc để tìm chỗ thích hợp cho khoá mới bị tăng này



Ví dụ: Heap-Increase-Key



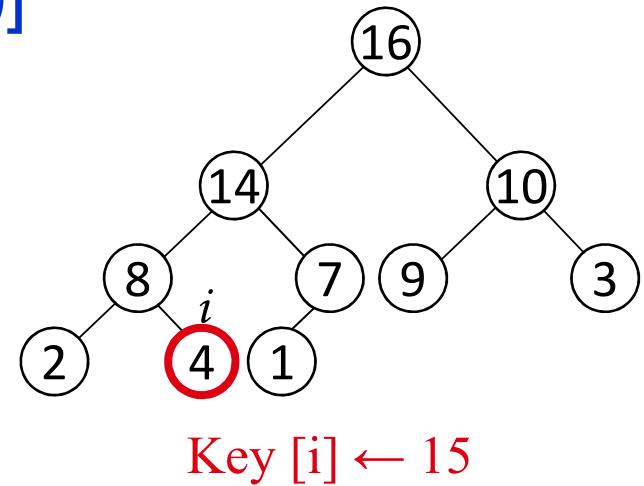
$Key[i] \leftarrow 15$



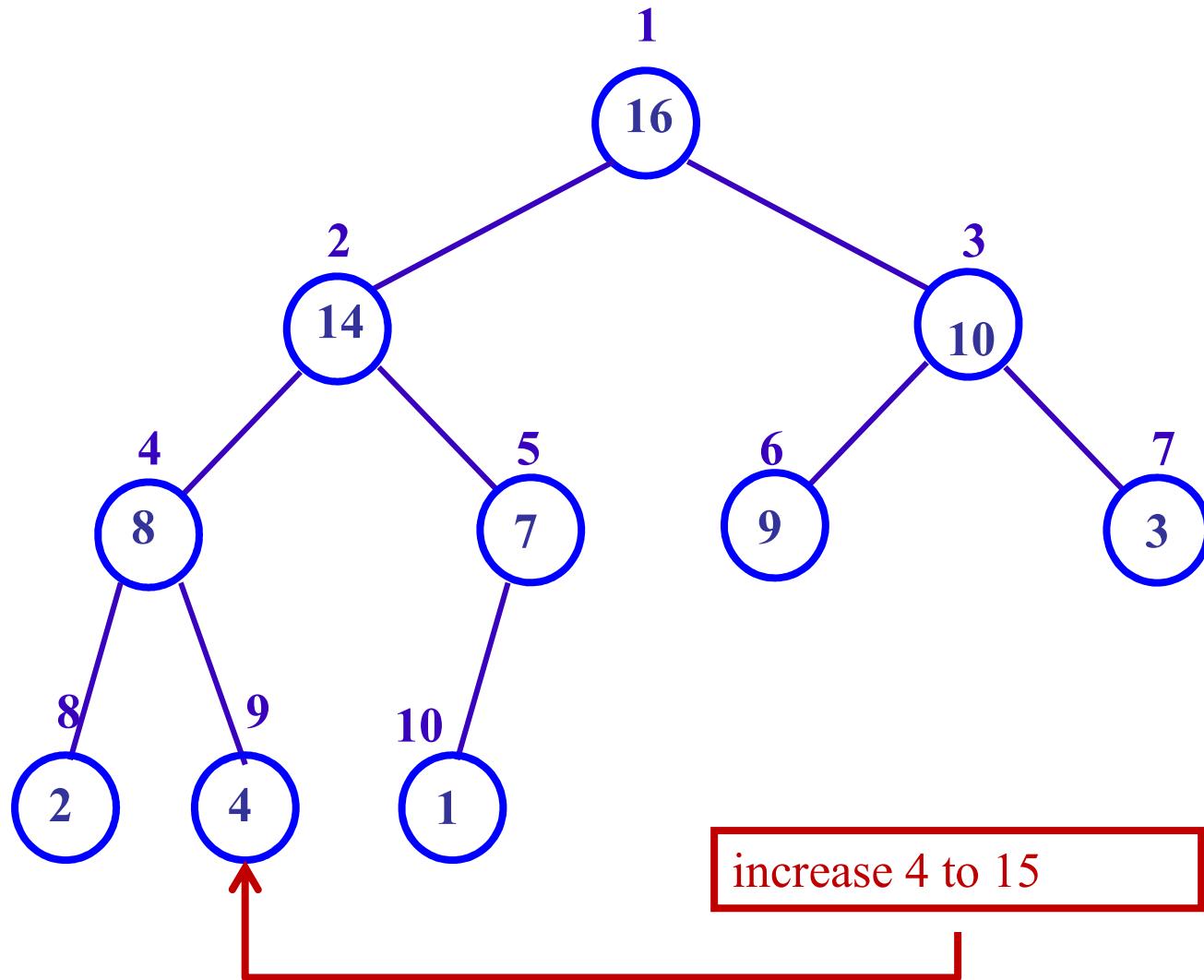
Heap-Increase-Key

Alg: Heap-Increase-Key(A, i, key)

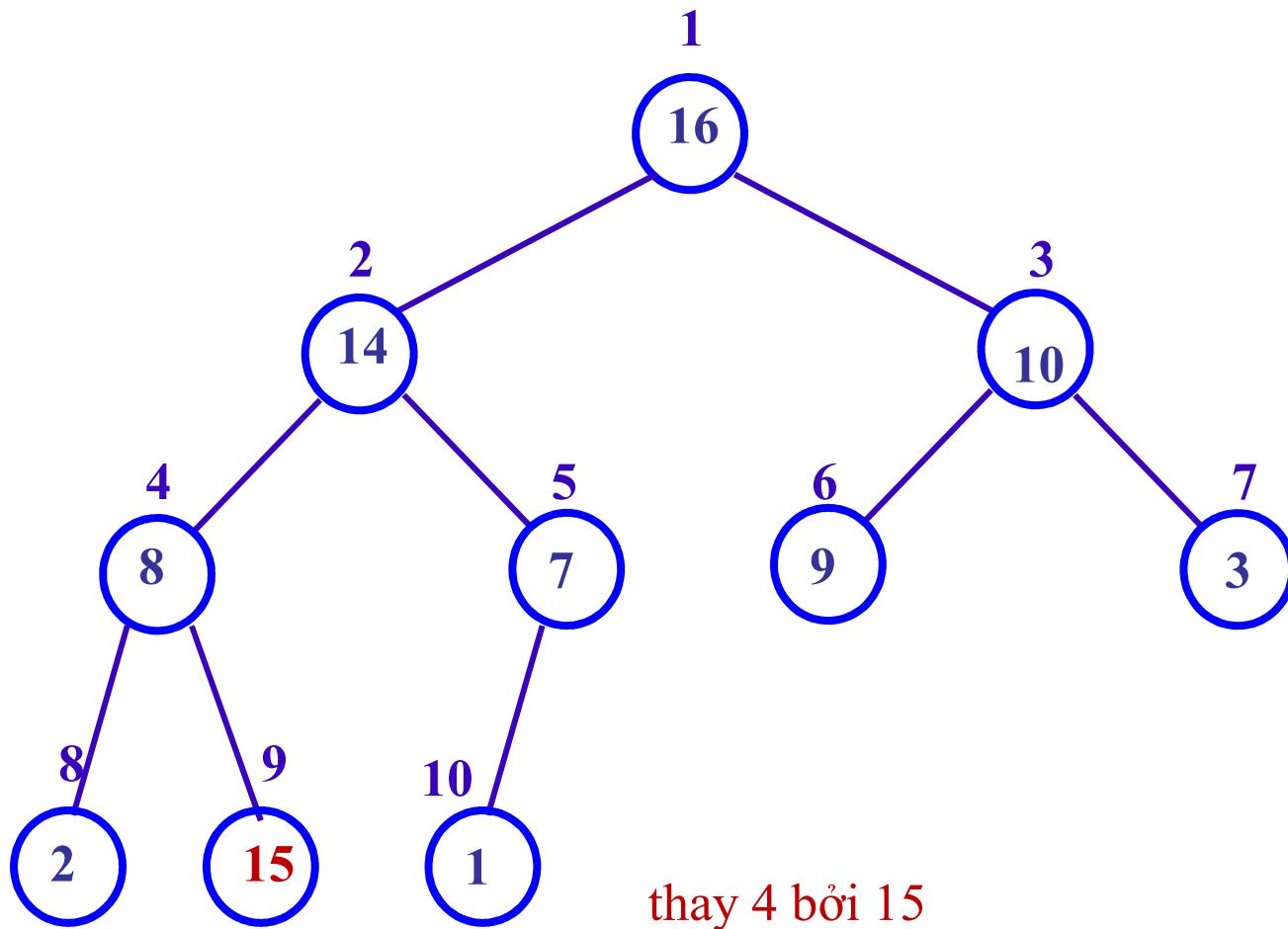
1. **if** $key < A[i]$
 2. **then error** “khoá mới nhỏ hơn khoá hiện tại”
 3. $A[i] \leftarrow key$
 4. **while** $i > 1$ and $A[\text{parent}(i)] < A[i]$
 5. **do** hoán đổi $A[i] \leftrightarrow A[\text{parent}(i)]$
 6. $i \leftarrow \text{parent}(i)$
- Thời gian tính: $O(\log n)$



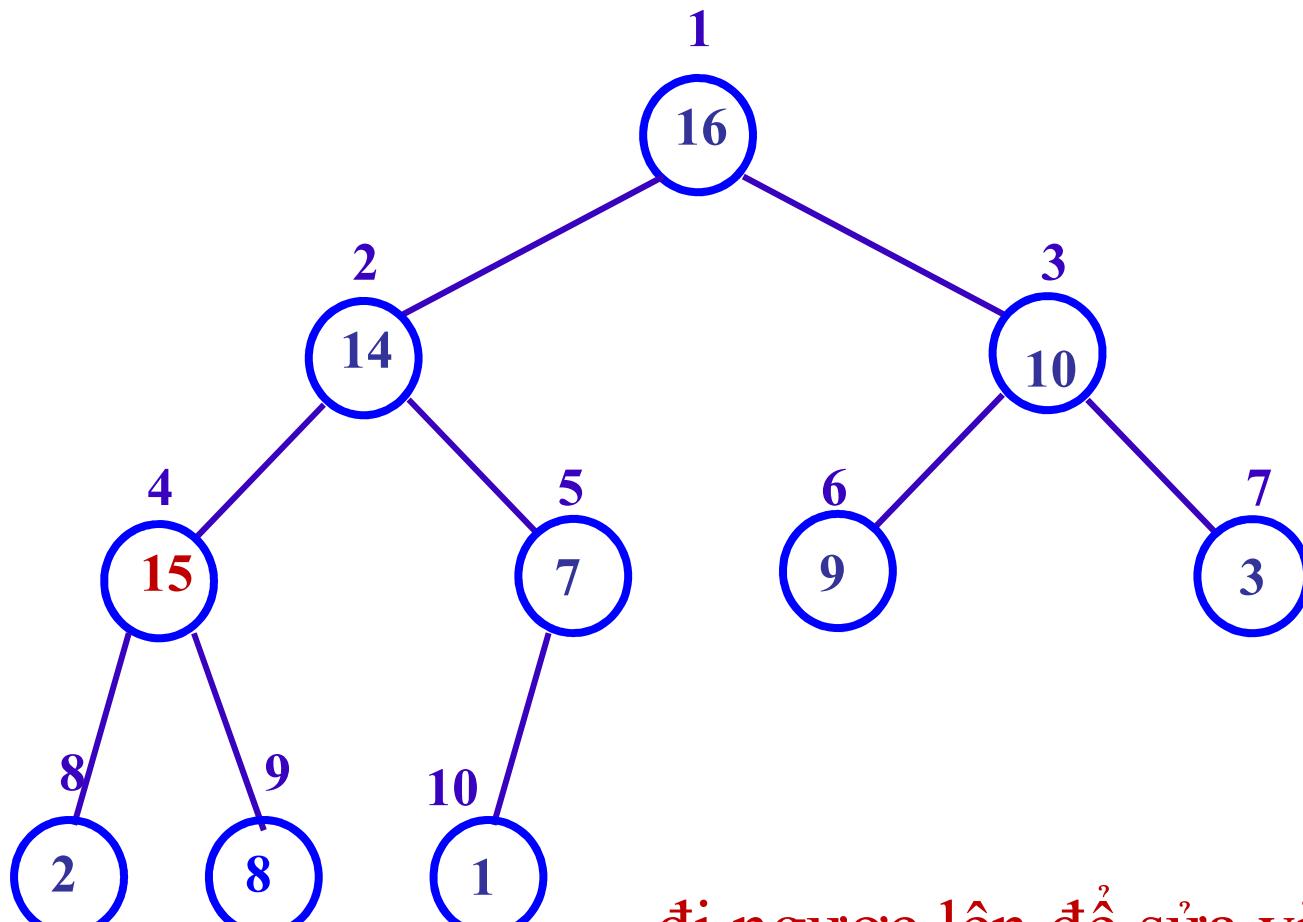
Ví dụ: Heap-Increase-Key (1)



Ví dụ: Heap-Increase-Key (2)

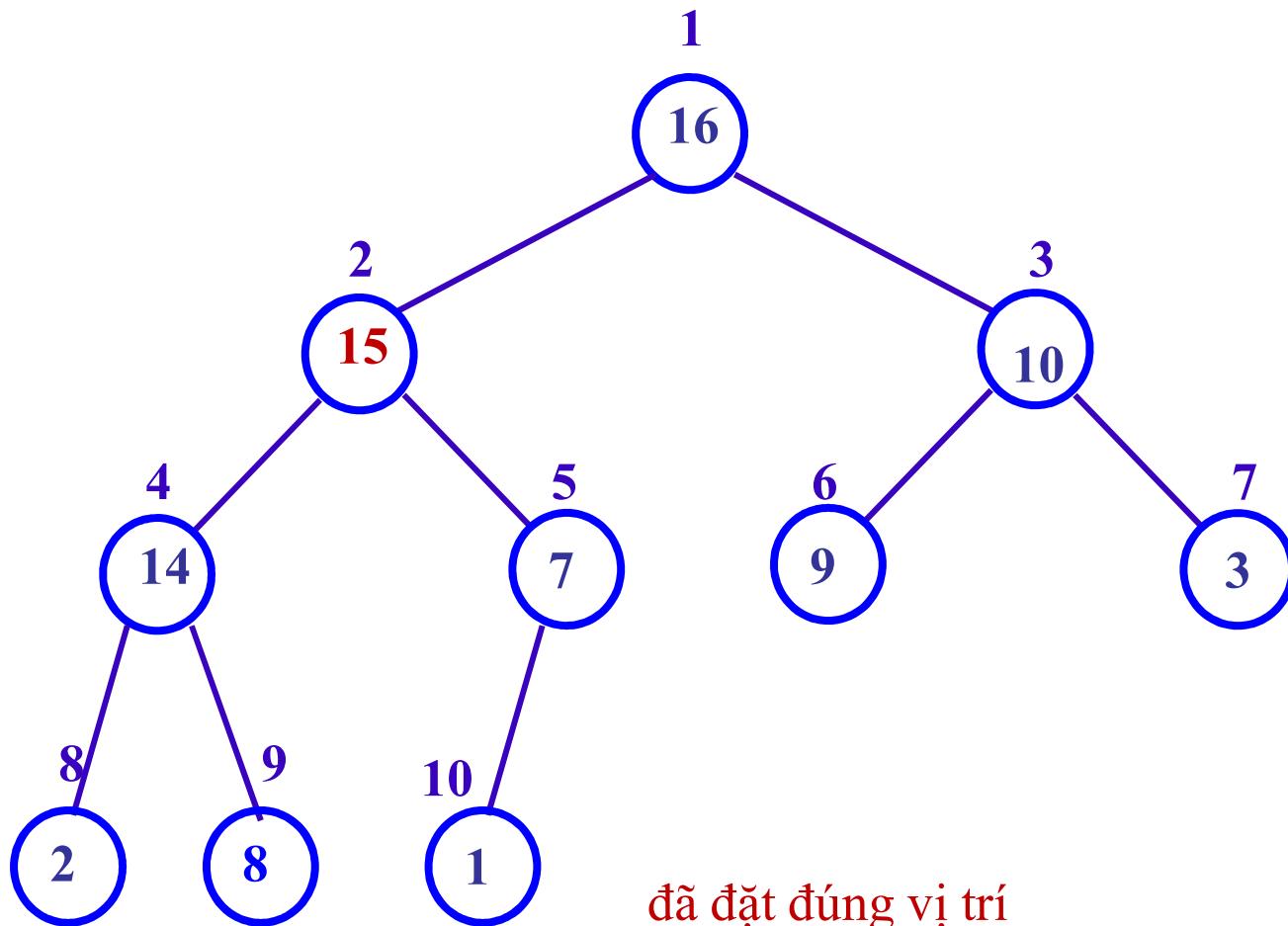


Ví dụ: Heap-Increase-Key (3)



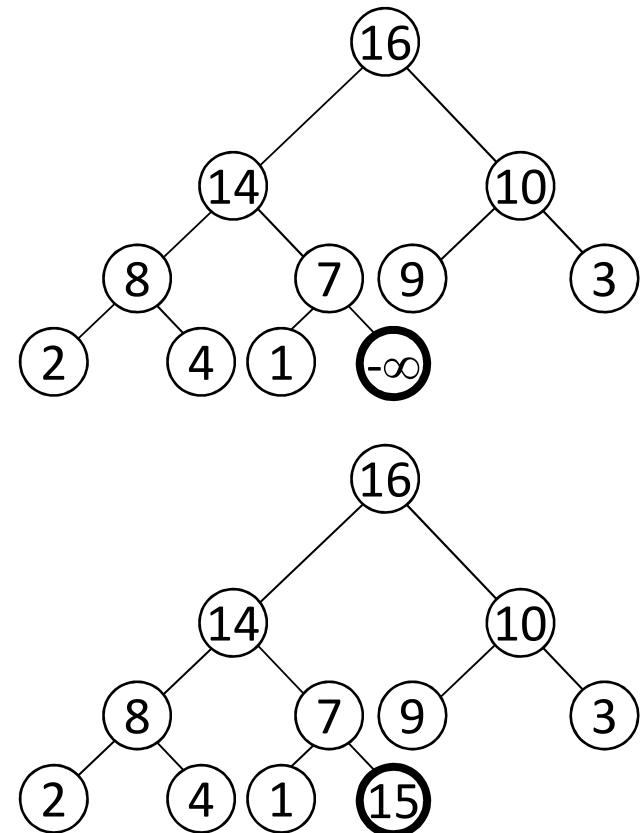
đi ngược lên để sửa vi phạm

Ví dụ: Heap-Increase-Key (4)



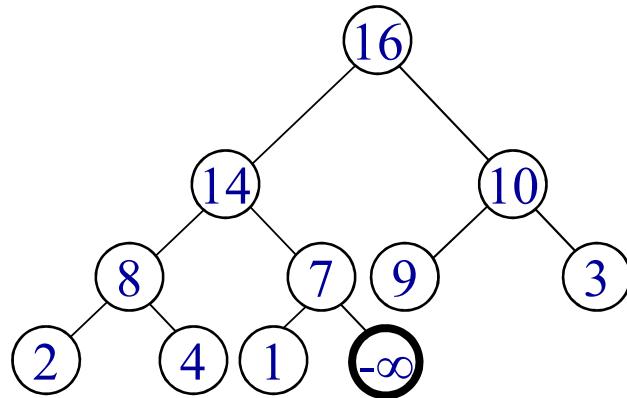
Phép toán Max-Heap-Insert

- **Chức năng:** Chèn một phần tử mới vào max-heap
- **Thuật toán:**
 - Mở rộng max-heap với một nút mới có khoá là $-\infty$
 - Gọi Heap-Increase-Key để tăng khoá của nút mới này thành giá trị của phần tử mới và vun lại đống

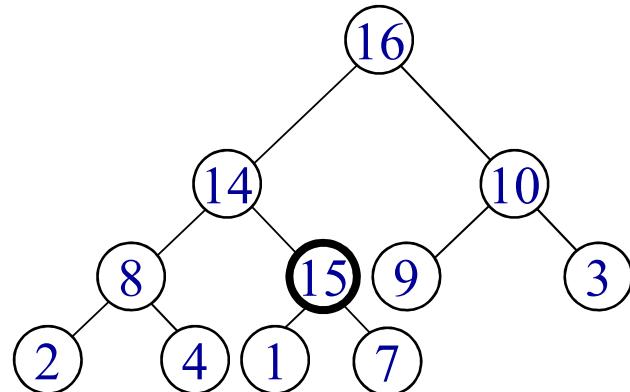
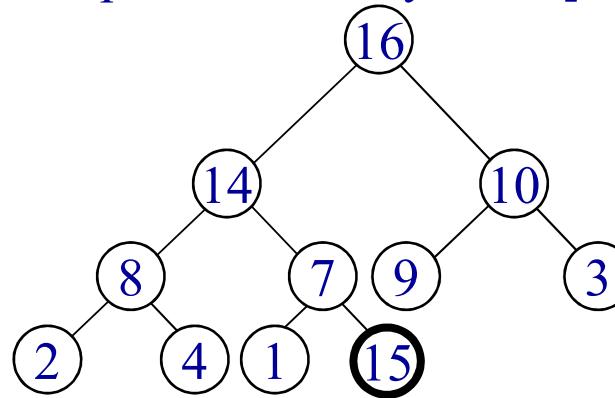


Ví dụ: Max-Heap-Insert

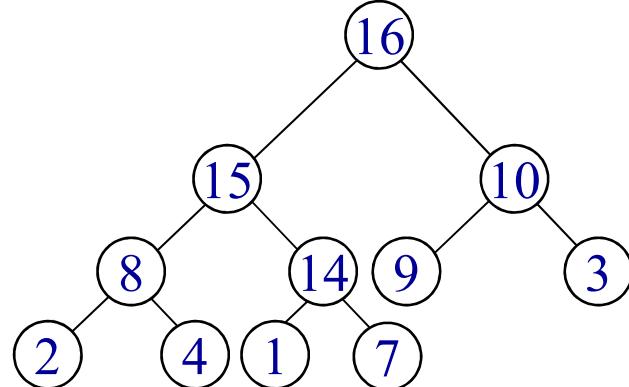
Chèn giá trị 15:
- Bắt đầu với $-\infty$



Tăng khoá thành 15
Gọi Heap-Increase-Key với $A[11] = 15$



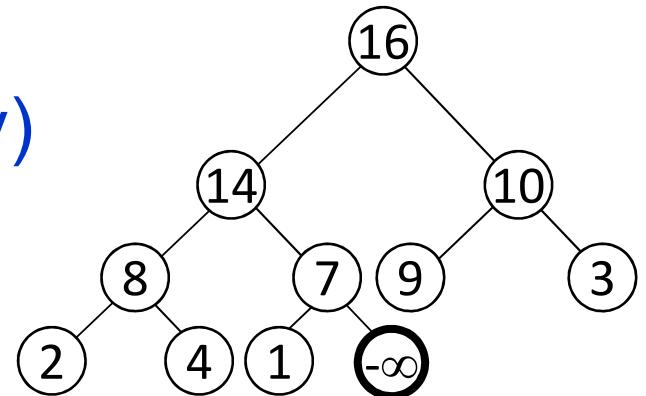
Vun lại đống
với phần tử
mới bổ sung



Max-Heap-Insert

Alg: Max-Heap-Insert(A , key, n)

1. $\text{heap-size}[A] \leftarrow n + 1$
2. $A[n + 1] \leftarrow -\infty$
3. Heap-Increase-Key(A , $n + 1$, key)



Running time: $O(\log n)$

Tổng kết

- Chúng ta có thể thực hiện các phép toán sau đây với đống:

Phép toán

- Max-Heapify
- Build-Max-Heap
- Heap-Sort
- Max-Heap-Insert
- Heap-Extract-Max
- Heap-Increase-Key
- Heap-Maximum

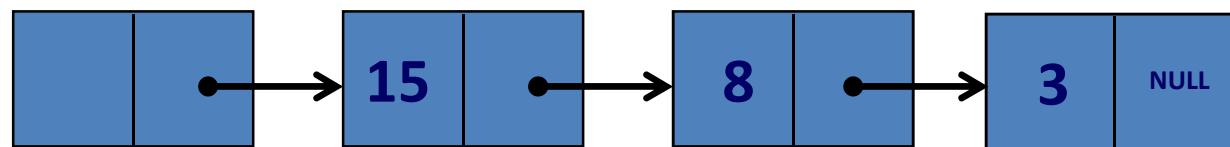
Thời gian tính

- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(\log n)$
- $O(\log n)$
- $O(\log n)$
- $O(1)$

Cài đặt hàng đợi có ưu tiên bởi danh sách mốc nối

- Priority Queues sử dụng heaps:
 - Tìm phần tử lớn nhất: Heap-Maximum $O(1)$
 - Lấy và loại phần tử lớn nhất: Heap-Extract-Max $O(\log n)$
 - Bổ sung phần tử mới: Heap-Insert $O(\log n)$
 - Tăng giá trị phần tử: Heap-Increase-Key $O(\log n)$
- Priority Queues sử dụng danh sách mốc nối được sắp thứ tự:
 - Tìm phần tử lớn nhất: $O(1)$
 - Lấy và loại phần tử lớn nhất: $O(1)$
 - Bổ sung phần tử mới: $O(n)$
 - Tăng giá trị phần tử: $O(n)$

header



Tổng kết:

Các thuật toán sắp xếp dựa trên phép so sánh

Name	Average	Worst	In place	Stable
Bubble sort	—	$O(n^2)$	Yes	Yes
Selection sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion sort	$O(n + d)$	$O(n^2)$	Yes	Yes
Merge sort	$O(n \log n)$	$O(n \log n)$	No	Yes
Heapsort	$O(n \log n)$	$O(n \log n)$	Yes	No
Quicksort	$O(n \log n)$	$O(n^2)$	Yes	No