SORTING ALGORITHMS (tt)

DATA STRUCTURES AND ALGORITHMS

Nội dung

- Các giải thuật sắp xếp nội:

1. Quick Sort

2. Heap Sort

3. Merge Sort

Quick Sort

- Được phát triển bởi Tony Hoare vào năm 1959
- Thuật toán Quicksort dựa trên chiến lược chia để trị (*divide-and-conquer)*.

Quick Sort: Ý tưởng

- Giải thuật QuickSort sắp xếp dãy $\mathbf{a_0, a_1 ..., a_{n-1}}$ dựa trên việc phân hoạch dãy ban đầu thành 3 phần :
  - Phần 1: Gồm các phần tử có giá trị bé hơn pivot
  - Phần 2: Gồm các phần tử có giá trị bằng pivot
  - Phần 3: Gồm các phần tử có giá trị lớn hơn pivot

  Với pivot là giá trị của một phần tử **tùy ý** trong dãy ban đầu.

Quick Sort: Ý tưởng

- Sau khi thực hiện phân hoạch, dãy ban đầu được phân thành 3 đoạn:
  - Đoạn 1: $a_k \leq$ pivot, với k = 0 .. J
  - Đoạn 2: $a_k =$ pivot, với k = j+1 .. i-1
  - Đoạn 3: $a_k \geq$ pivot, với k = i .. n-1

$a_k \leq$ pivot	a_k=pivot	pivot $\geq a_k$

Quick Sort: Ý tưởng

$a_k \leq pivot$	$a_k = pivot$	$pivot \geq a_k$

- Đoạn thứ 2 đã có thứ tự.
- Nếu các Đoạn 1 và 3 chỉ có 1 phần tử: đã có thứ tự → khi đó dãy con ban đầu đã được sắp.
- Nếu các Đoạn 1 và 3 có nhiều hơn 1 phần tử thì dãy ban đầu chỉ có thứ tự khi các đoạn 1, 3 được sắp.

  => Để sắp xếp các Đoạn 1 và 3, ta lần lượt tiến hành việc phân hoạch từng dãy con theo cùng phương pháp phân hoạch dãy ban đầu vừa trình bày …

Quick Sort: Chọn Pivot

- Chọn phần tử ngẫu nhiên
- Phần tử trung vị (Median) hay *Median-of-Three*
- Chọn phần tử đầu tiên, cuối cùng => shouldn’t
  - The popular, uninformed choice is to use the first element as the pivot. This is acceptable if the input is random, but if the input is presorted (or input with a large presorted section) or in reverse order, then the pivot provides a poor partition, because either all the elements go into $S1$ or they go into $S2$.
- ...

Quick Sort: Thuật toán

- **Bước 1**: Nếu left ≥ right thì Kết thúc; // dãy đã có thứ tự
- **Bước 2**: Phân hoạch (Partition) dãy a_{left} .. a_{right} thành các đoạn a_{left} .. a_j, a_{j+1} .. a_{i-1}, a_i .. a_{right}

  Đoạn 1: $a_{left} .. a_j \leq x$

  Đoạn 2: $a_{j+1} .. a_{i-1} = x$

  Đoạn 3: $a_i .. a_{right} \geq x$
- **Bước 3**: Sắp xếp đoạn 1: a_{left} .. a_j
- **Bước 4**: Sắp xếp đoạn 3: a_i .. a_{right}

Quick Sort: Giải thuật Partition

- **Bước 1**: Chọn tùy ý một phần tử a[k] trong dãy làm giá trị pivot

  (left ≤ k ≤ right): pivot **= a[k]; i = left; j = right;**
- **Bước 2**: Phát hiện và hiệu chỉnh cặp phần tử a[i], a[j] nằm sai chỗ:
  - Bước 2a: Trong khi (a[i] < pivot) i++;
  - Bước 2b: Trong khi (a[j] > pivot) j--;
  - Bước 2c: Nếu i ≤ j:

    Doicho(a[i], a[j]);

    i++, j--
- **Bước 3**: Nếu i ≤ j: Lặp lại Bước 2.

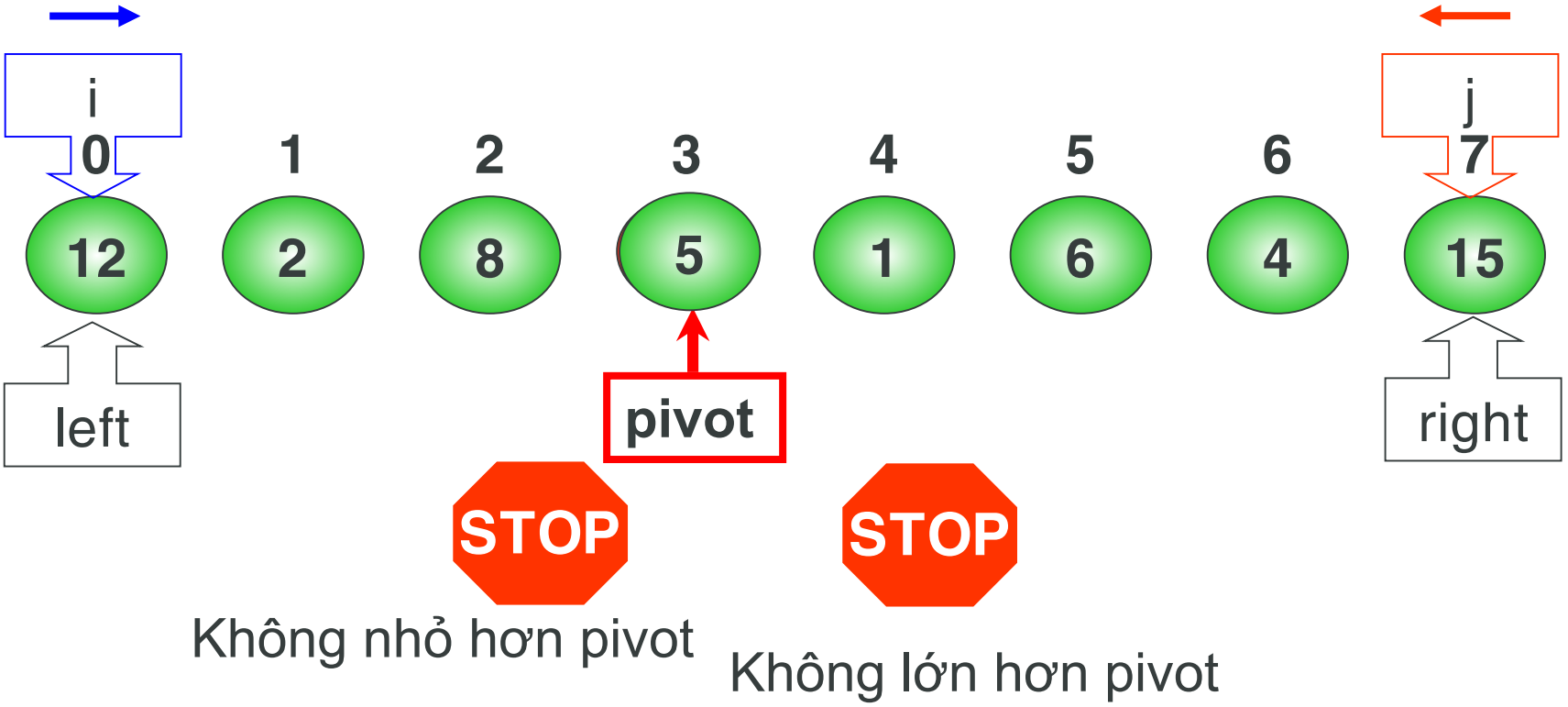
  Ngược lại: Dừng

Quick Sort: Code C/C++

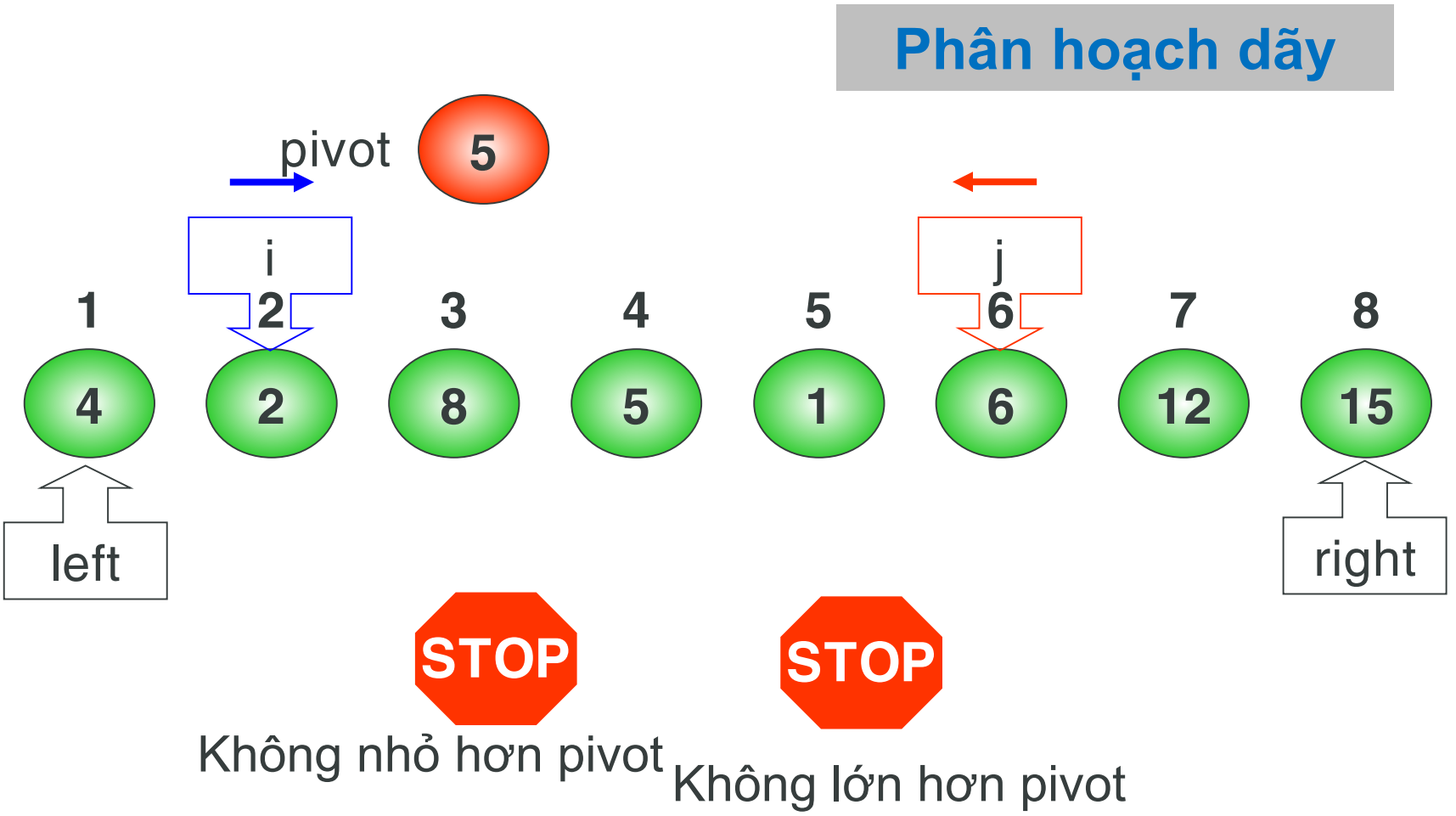
```
void Sort(int a[], int n) {
      QuickSort(a, 0, n - 1);
}

void QuickSort(int a[], int left, int right) {
    int i, j, pivot;
    pivot = a[(left + right) / 2];
    i = left; j = right;

    while (i <= j) {
        while (a[i] < pivot) i++;
        while (a[j] > pivot) j--;
        if (i <= j) {
            Swap(a[i], a[j]);
            i++; j--;
        }
    }

    if (left < j)
            QuickSort(a, left, j);
    if (i < right)
            QuickSort(a, i, right);
}
```

Quick Sort: Minh họa

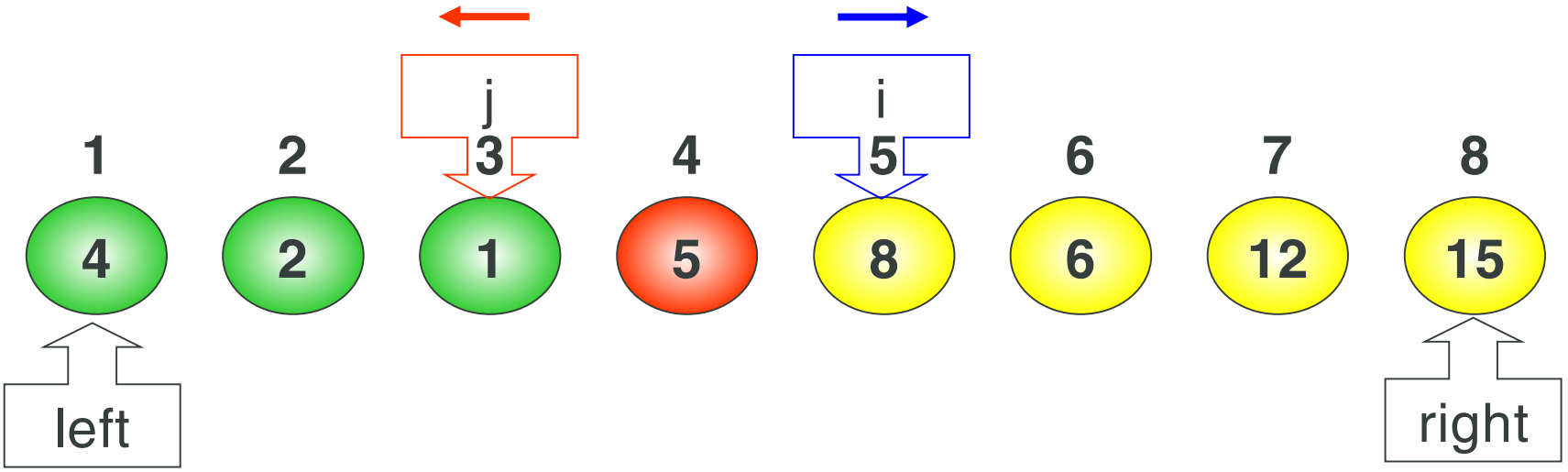
Phân hoạch dãy



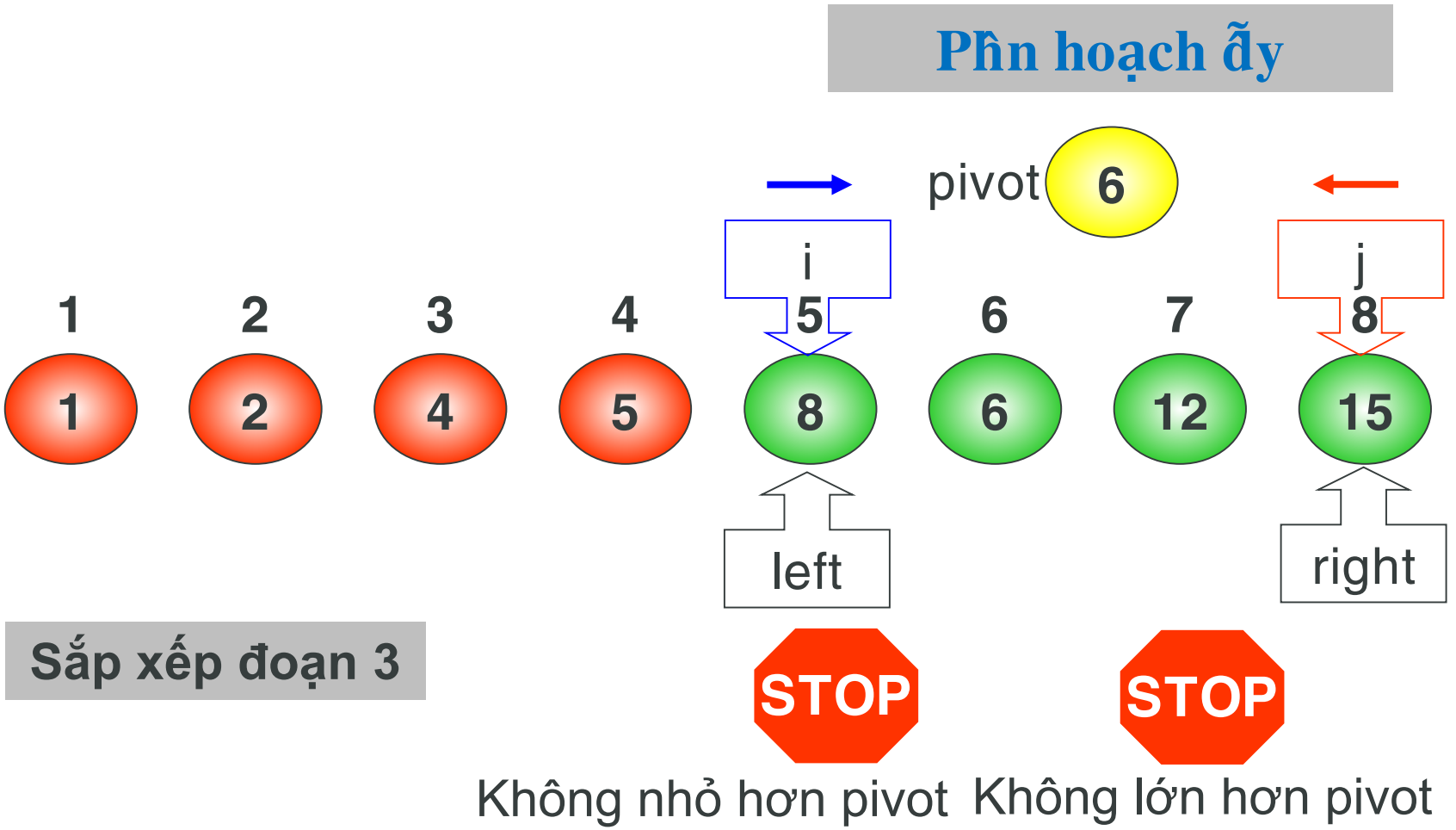
Quick Sort: Minh họa



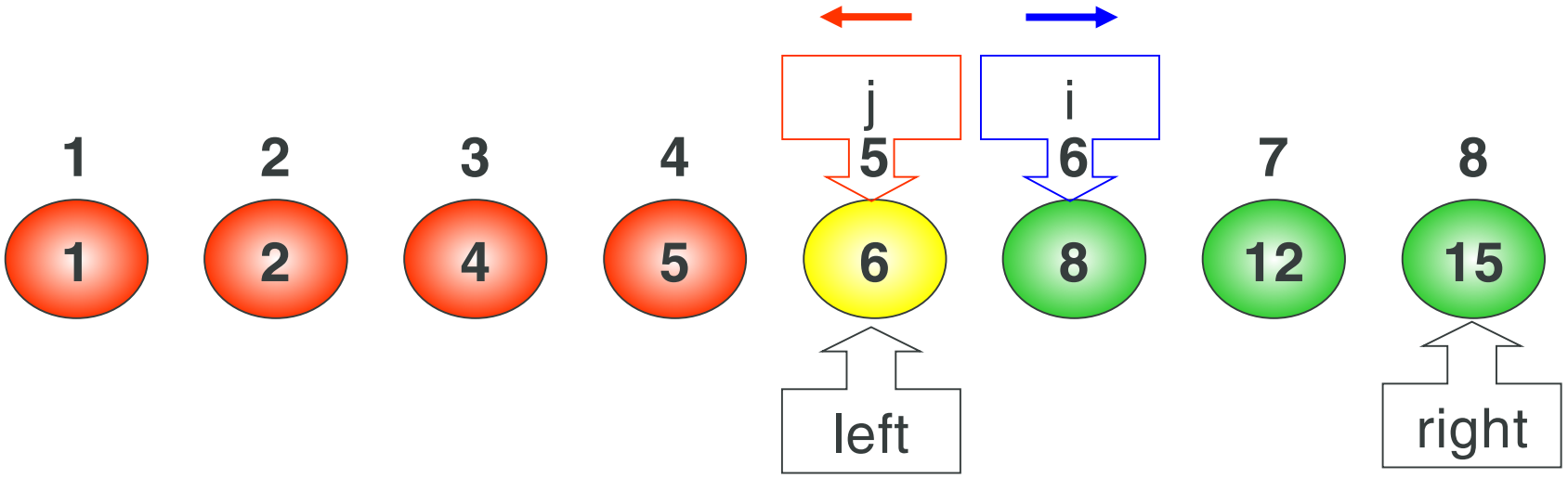
Quick Sort: Minh họa



Quick Sort: Minh họa



Quick Sort: Minh họa



Sắp xếp đoạn 3

Quick Sort: Độ phức tạp

- Quicksort sẽ kém hiệu quả khi dãy đã có thứ tự hoặc khi chọn phần tử đầu tiên làm phần tử pivot
- Quicksort là một trong những thuật toán hiệu quả nhất sử dụng từ khóa so sánh.

Trường hợp	Độ phức tạp
Tốt nhất	$nlog_2n$
Trung bình	$nlog_2n$
Xấu nhất	n^2

Nội dung

1. Quick Sort
2. **Heap Sort**
3. Merge Sort
4. Radix Sort

Heap Sort

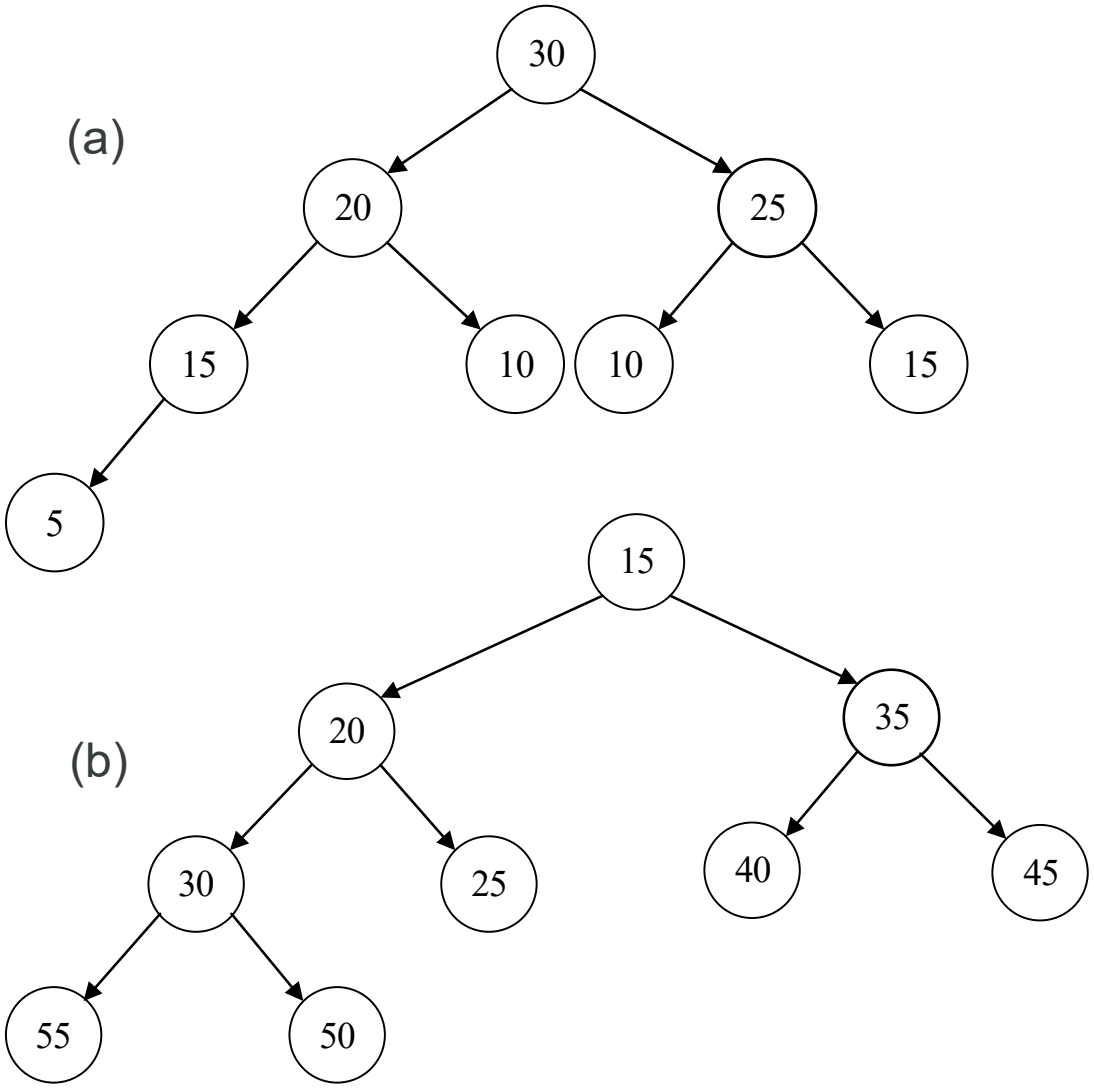
- Sắp xếp Heapsort dựa trên một cấu trúc dữ liệu được gọi là Binary Heap, được giới thiệu bởi J. W. J. Williams vào năm 1964.

Định nghĩa cây Heap

- Cây **Binary Max-Heap** có tính chất: Là cây nhị phân hoàn chỉnh (complete binary tree) và giá trị của một node bất kỳ trên cây sẽ **luôn lớn hơn hoặc bằng** giá trị của các node con của nó (nếu có).
- Cây **Binary Min-Heap** có tính chất: Là cây nhị phân hoàn chỉnh (complete binary tree) và giá trị của một node bất kỳ trên cây sẽ **luôn nhỏ hơn hoặc bằng** giá trị của các node con của nó (nếu có).

Bài tập:

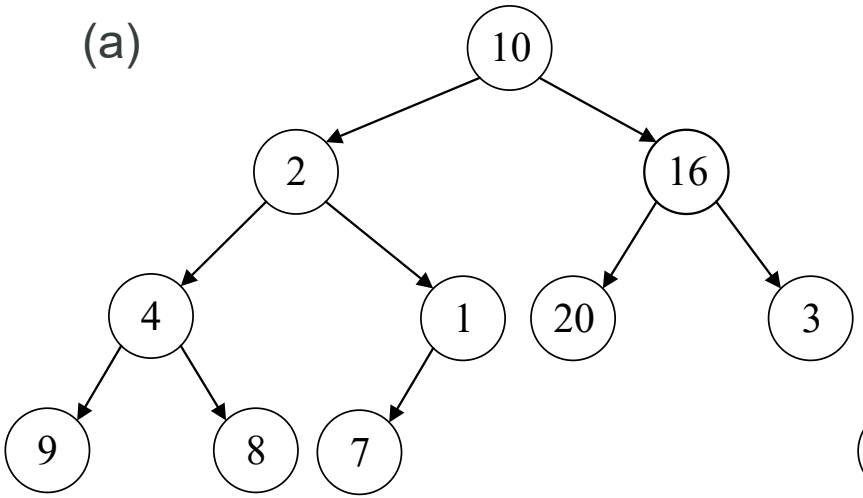
Câu 1. Cho biết cây Min-Heap, Max-Heap tương ứng trong 2 hình bên ?



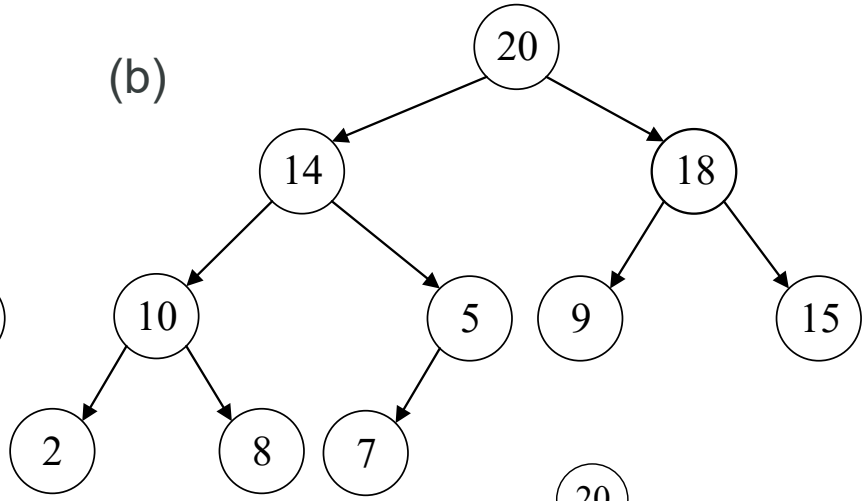
Bài tập: (tt)

Câu 2. Cho biết cây nào trong những hình bên dưới không thỏa tính chất Heap.

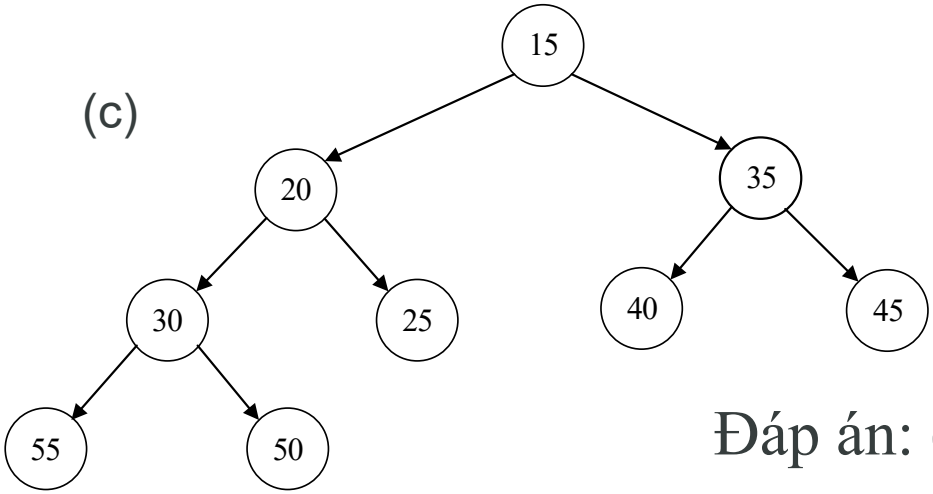
(a)



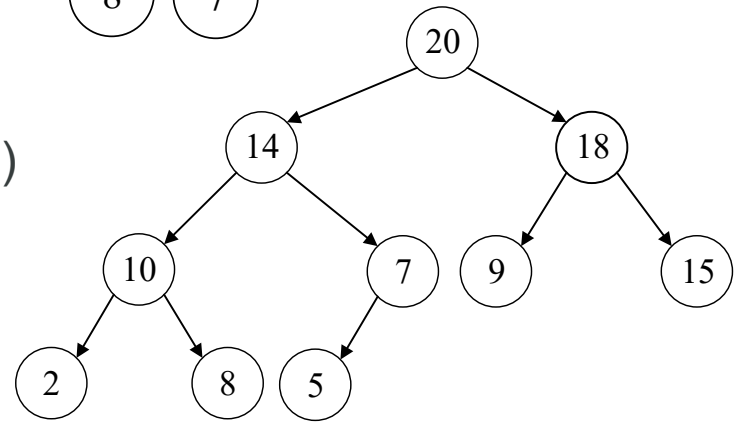
(b)



(c)



(d)

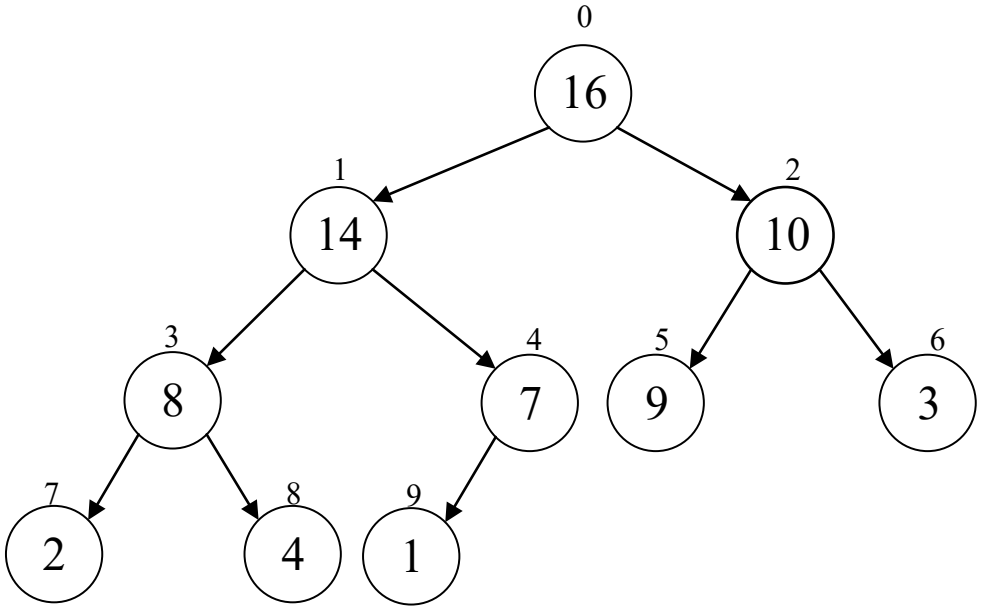


Đáp án: (a), (b)

Array-based Binary Heap

Binary Heap thường được biểu diễn bởi một mảng (array-based binary heap) thay vì một cây nhị phân (pointer-based binary heap).

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1



Array-based Binary Heap

Gốc của cây A là A[0].

Gọi i là chỉ số của một node bất kỳ trong cây, ta ký hiệu sau:

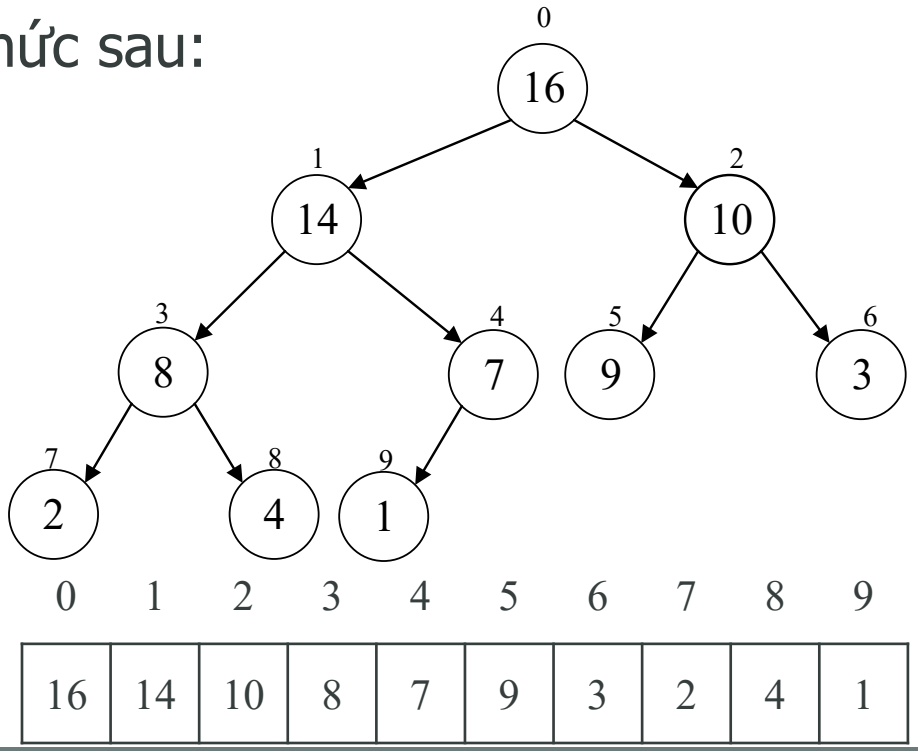
+ Chỉ số của node cha của i gọi là: PARENT(i),

+ Chỉ số của node con trái của i gọi là: LEFT(i)

+ Chỉ số của node con phải của i gọi là: RIGHT(i)

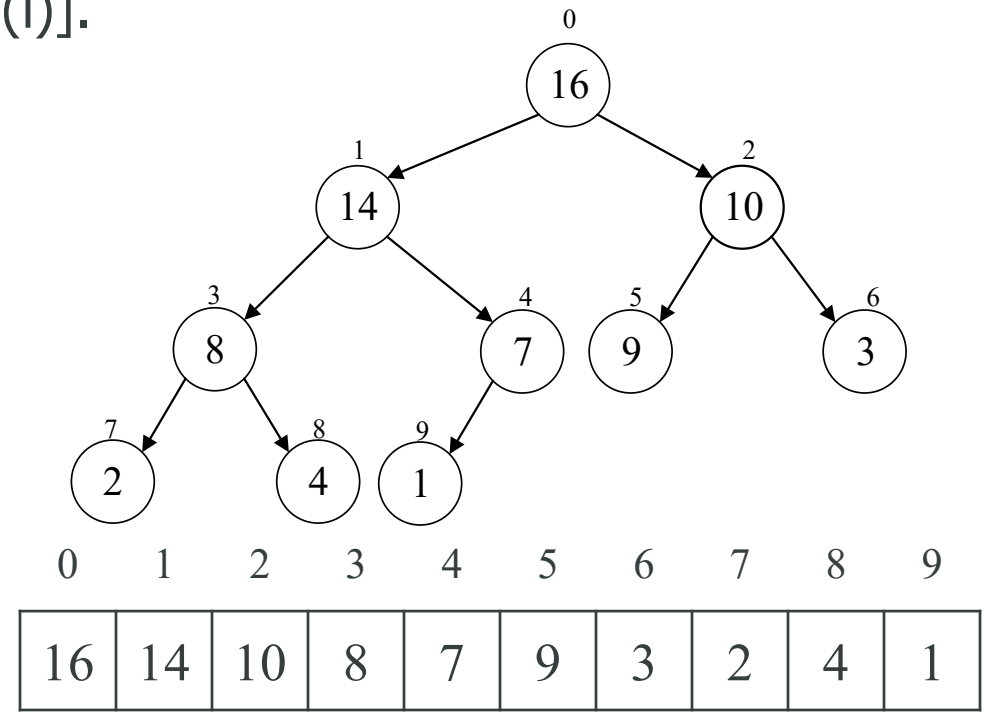
3 chỉ số này được tính qua công thức sau:

```
PARENT(i)
    return [(i-1)/2]
LEFT(i)
    return 2i+1
RIGHT(i)
    return 2i+2
```



Array-based binary heap

- Array-based binary heap phải thỏa mãn tính chất heap (*heap property):*
  - Với Binary Max-Heap: Tất cả các node i (không kể node gốc) thỏa A[i] ≤ A[PARENT(i)].
  - Với Binary Min-Heap: Tất cả các node i (không kể node gốc) thỏa A[i] ≥ A[PARENT(i)].



Bài tập: (tt)

Cho biết dãy nào sau đây thỏa tính chất Heap.

a) 20, 18, 15, 17, 5, 10

b) 20, 18, 15, 17, 15, 17, 14, 5

c) 20, 18, 15, 17, 15, 10, 14, 5

d) 20, 40, 30, 38, 45, 35

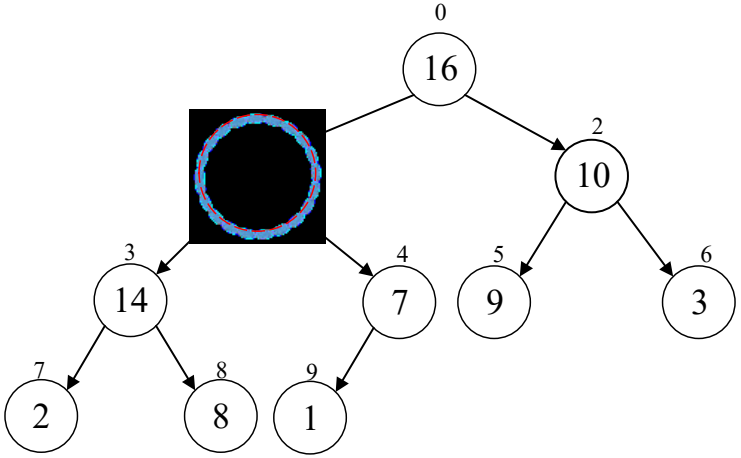
e) 20, 40, 30, 48, 45, 35

Duy trì tính chất Heap: HEAPIFY hay Re-Heap

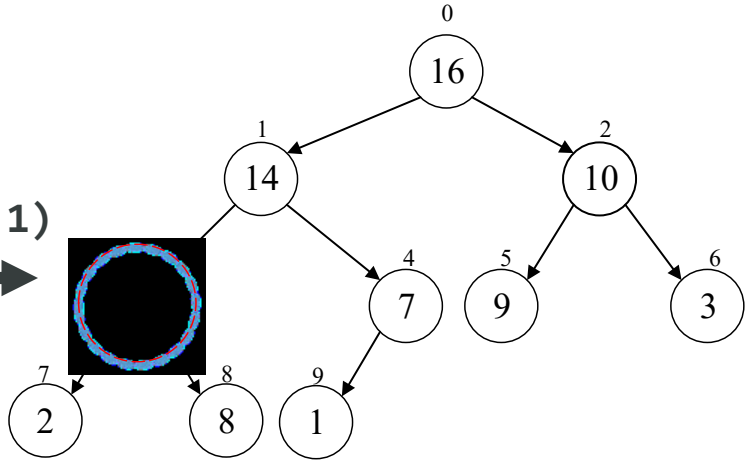
- HEAPIFY is to let the value at A[i] "float down" in the heap so that the subtree rooted at i becomes a heap:
- Thuật toán:

```
HEAPIFY(A, heap-size, i)
1.  l <- LEFT(i)
2.  r <- RIGHT(i)
3.  largest <- i
4.  if l < heap-size[A] and A[l] > A[i]
5.       then largest <- l
6.  if r < heap-size[A] and A[r] > A[largest]
7.       then largest <- r
8.  if largest ≠ i
9.       then exchange A[i] <-> A[largest]
10.      HEAPIFY(A, heap-size, largest )
```

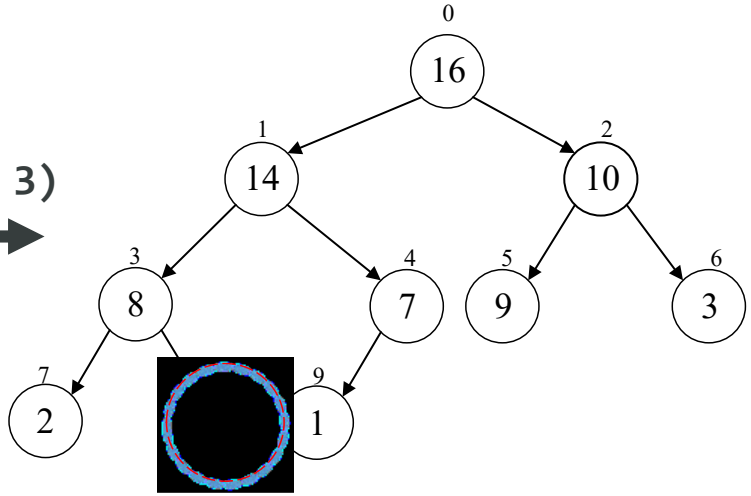
Duy trì tính chất Heap: Ví dụ trên cây Max-Heap



HEAPIFY(A, 10, 1)



HEAPIFY(A, 10, 3)



The action of

HEAPIFY(A, 10, 1)
HEAPIFY(A, 10, 3)
HEAPIFY(A, 10, 8)

where **heap-size[A]=10**

Xây dựng cây Heap: BUILD-HEAP

- **BUILD-HEAP** thực hiện chuyển đổi dãy A[0 .. n-1] thành dãy có tính chất heap (array-based binary heap). Với N là số lượng phần tử của A, ký hiệu length[A]=n. Quá trình đó sẽ sử dụng **HEAPIFY** trên các phần tử của mảng. Trong đó:
  - Các phần tử từ **A[n/2]** → **A[n-1]** là các node lá của cây nên không cần xét.
  - **BUILD-HEAP** sẽ thực hiện **HEAPIFY** tất cả các phần tử còn lại bắt đầu từ dưới lên trên (bottom-up manner). Nghĩa là từ **A[n/2-1] ngược lên A[0]**.

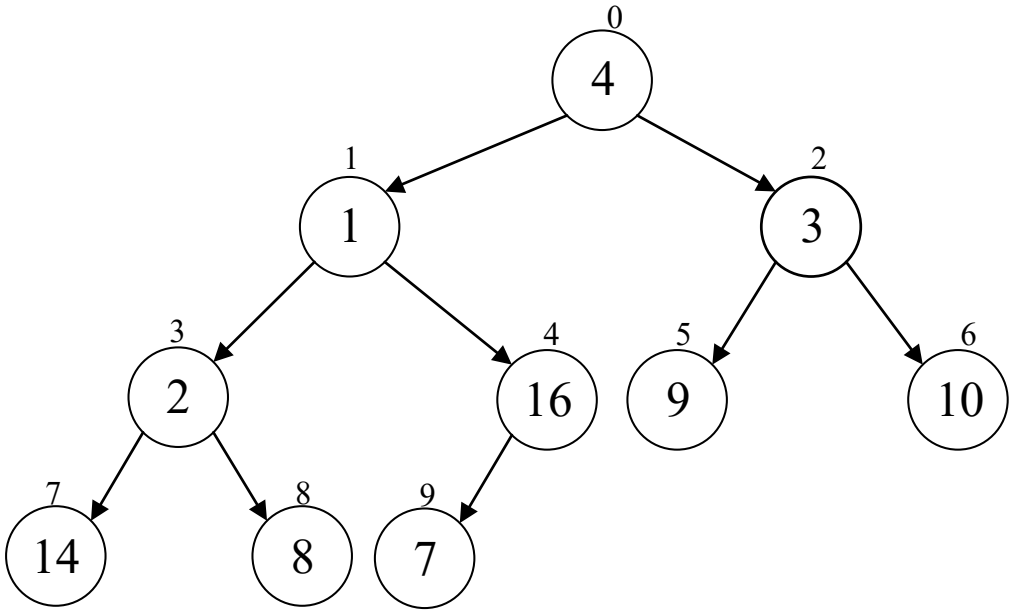
BUILD-HEAP(A)	
1.	heap-size[A] <- length[A]
2.	for i <- length[A]/2-1 downto 0 do
3.	HEAPIFY(A, heap-size, i)

BUILD-HEAP: Ví dụ

Xây dựng cây Heap cho dãy số sau:

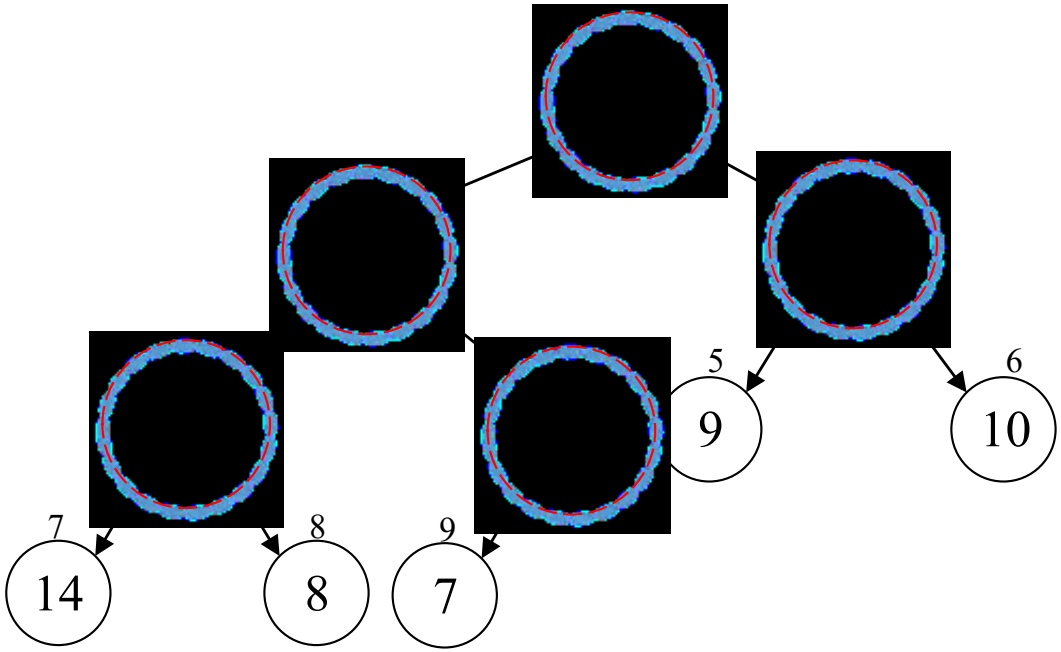
0	1	2	3	4	5	6	7	8	9
4	1	3	2	16	9	10	14	8	7

Ghi lại dãy số dưới dạng cây:

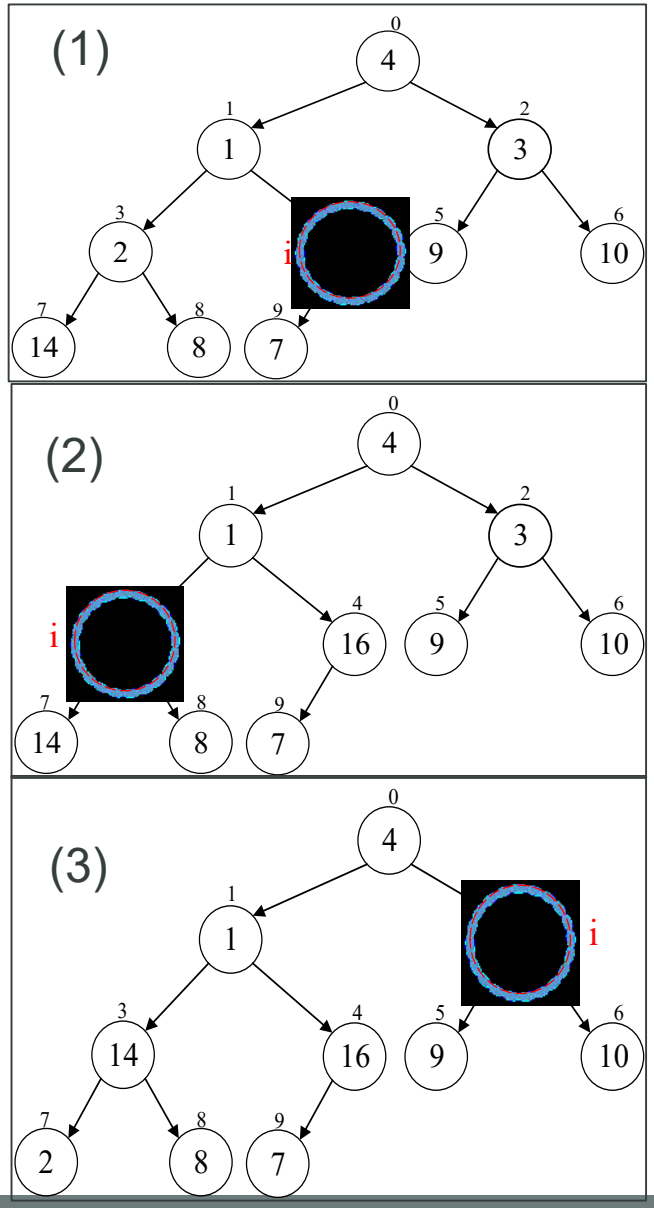


BUILD-HEAP: Ví dụ (tt)

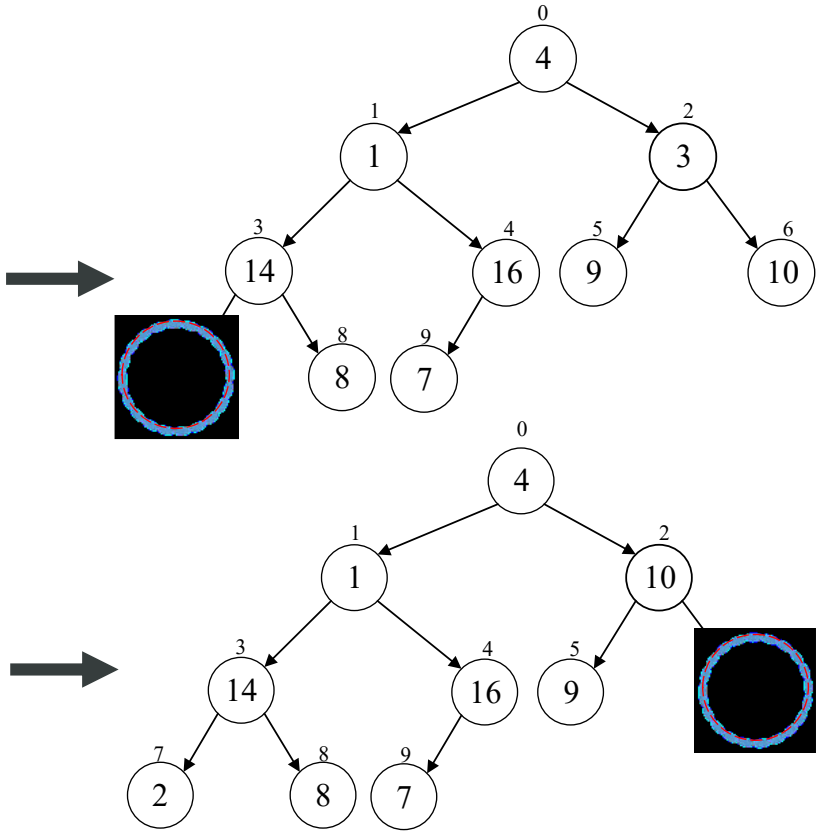
- Các giá trị trên dãy cần xem xét trong quá trình BUILD-HEAP lần lượt là 16, 2, 3, 1, 4:



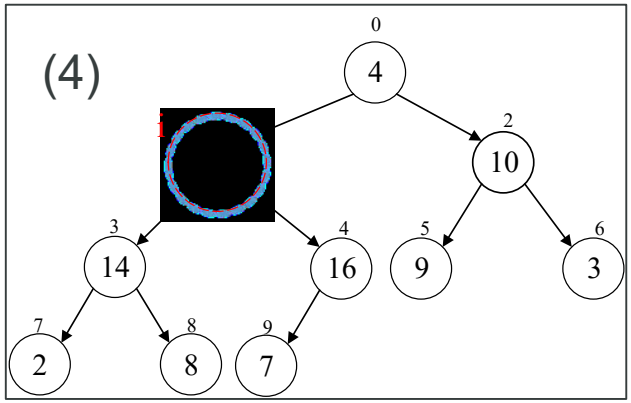
BUILD-HEAP: Ví dụ (tt)

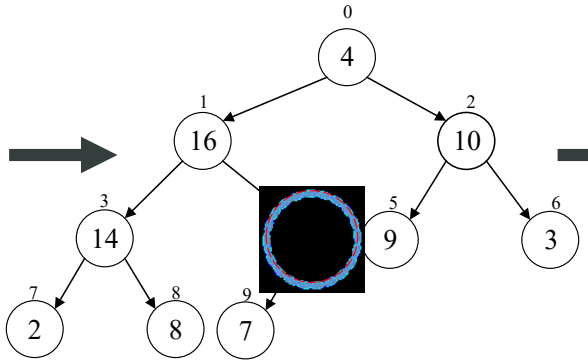


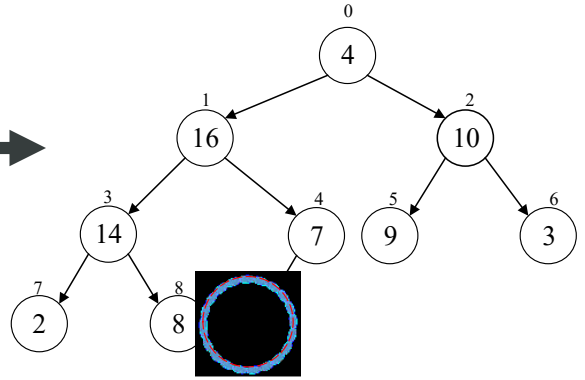
The operation of **BUILD-HEAP**, before the call **HEAPIFY**:
(1): A 10-element input array **A** and the binary tree it represents. The figure shows that the loop index **i** points to node 4 before the call **HEAPIFY(A, i)**.
(2): The data structure that results. The loop index **i** for the next iteration points to node 3.

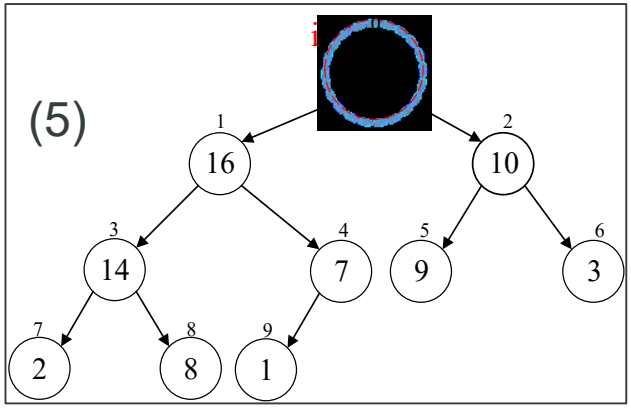


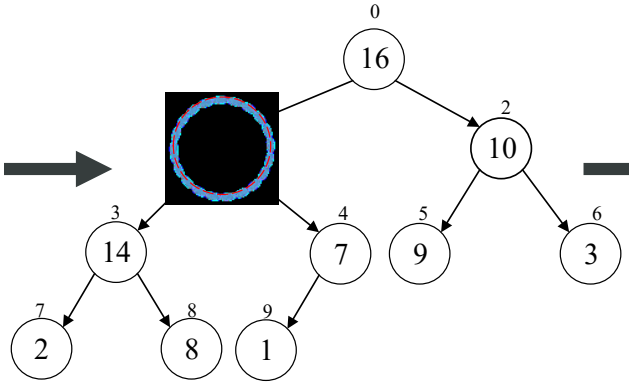
BUILD-HEAP: Ví dụ (tt)

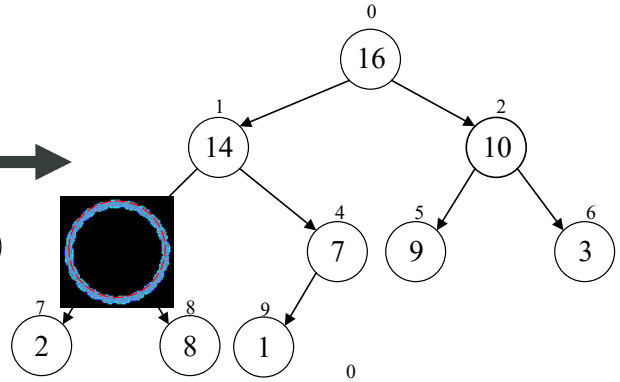




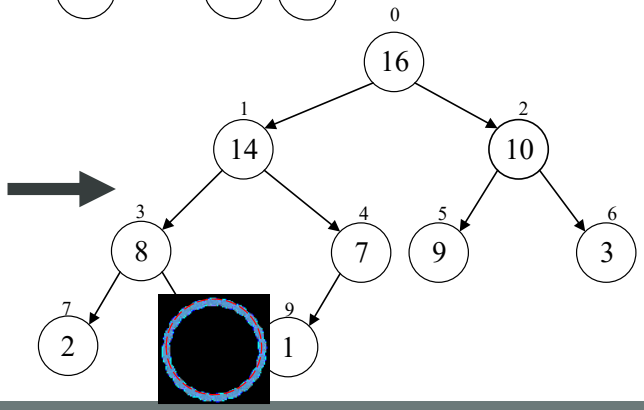








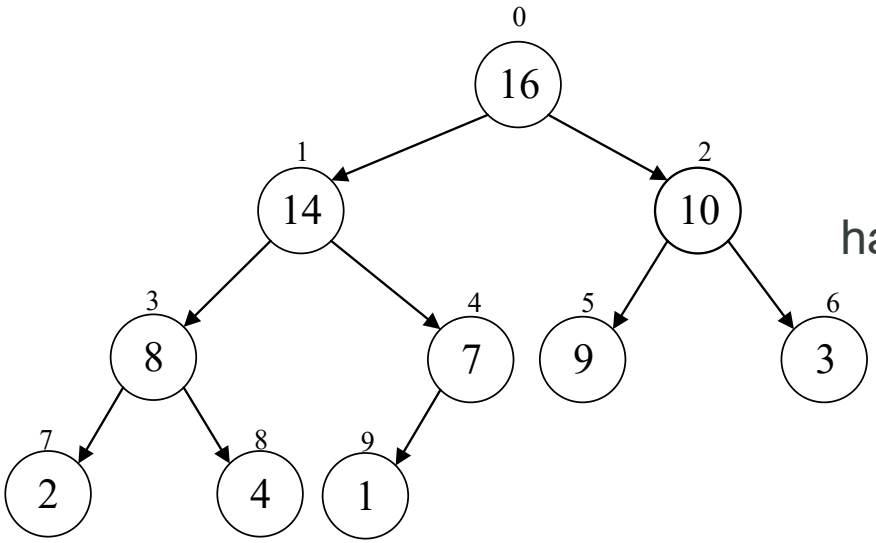
(3)-(5): Subsequent iterations of the for loop in **BUILD-HEAP**. Whenever **HEAPIFY** is called on a node, the two subtrees of that node are both heaps.



BUILD-HEAP: Ví dụ (tt)

	0	1	2	3	4	5	6	7	8	9
Dãy ban đầu:	4	1	3	2	16	9	10	14	8	7

The heap after **BUILD-HEAP** finishes :



hay:	0	1	2	3	4	5	6	7	8	9
	16	14	10	8	7	9	3	2	4	1

Heap Sort: Ý tưởng

- Bước 1: Sử dụng BUILD-HEAP để tạo dãy Heap cho dãy A[0..n-1] với n là chiều dài của dãy.
- Bước 2: Sau đó thực hiện đưa A[0] về cuối dãy bằng cách hoán vị với phần tử cuối cùng của dãy heap đang xét. Xét dãy heap mà đã loại bỏ phần tử cuối cùng. Tuy nhiên phần tử gốc A[0] của dãy heap hiện tại sẽ không còn giữ tính chất heap. Vì vậy ta cần HEAPIFY lại phần tử A[0] để đưa dãy hiện tại về dãy có tính chất heap. Thực hiện lặp lại các thao tác ở bước 2 trên cho tới khi dãy heap còn 1 phần tử.

Heap Sort: Thuật toán

- The heapsort algorithm then repeats this process for the heap of size **n-1** down to a heap of size 2.

HEAPSORT(A)	
1.	BUILD-HEAP(A)
2.	for i <- length[A]-1 downto 1 do
3.	exchange A[0] <-> A[i]
4.	heap-size[A] <- heap-size[A]-1
5.	HEAPIFY(A, heap-size, 0)

- The **HEAPSORT** procedure takes time O(nlogn), since the call to **BUILDHEAP** takes time O(n) and each of the **n-1** calls to **HEAPIFY** takes time O(logn).

HeapSort: Code C/C++

```
void Heapify(int a[], int heapSize, int i) {
    int childLeft  = i*2+1;
    int childRight = i*2+2;
    int max = i;
    if(childLeft<heapSize && a[max]<a[childLeft])
          max=childLeft;
    if(childRight<heapSize && a[max]<a[childRight])
          max= childRight;
    if(max != i) {
        swap(a[max], a[i]);
        Heapify(a, heapSize, max);
    }
}
```

HeapSort: Code C/C++ (tt)

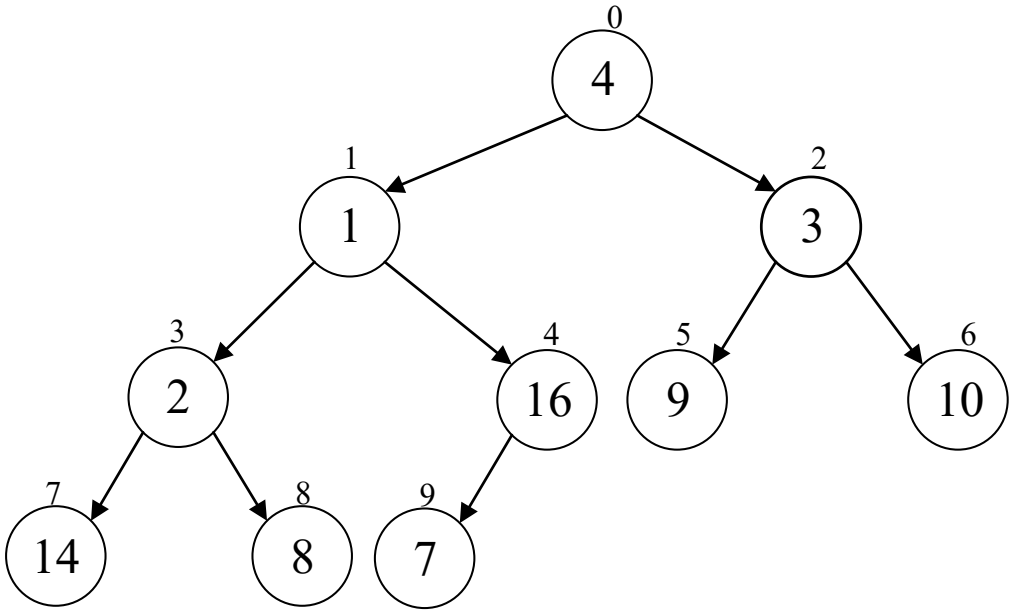
```
void buildHeap(int a[], int n) {
    int heapSize = n;
    for(int i = n/2-1; i>=0; i--)
          Heapify(a, heapSize, i);
}
void HeapSort(int a[], int n) {
    int heapSize;
    heapSize = n;
    buildHeap(a, n);
    for(int i = n-1; i>=1 ; i--) {
        swap(a[0], a[i]);
        heapSize -= 1;
        Heapify(a, heapSize, 0);
    }
}
```

Heap Sort: Minh họa

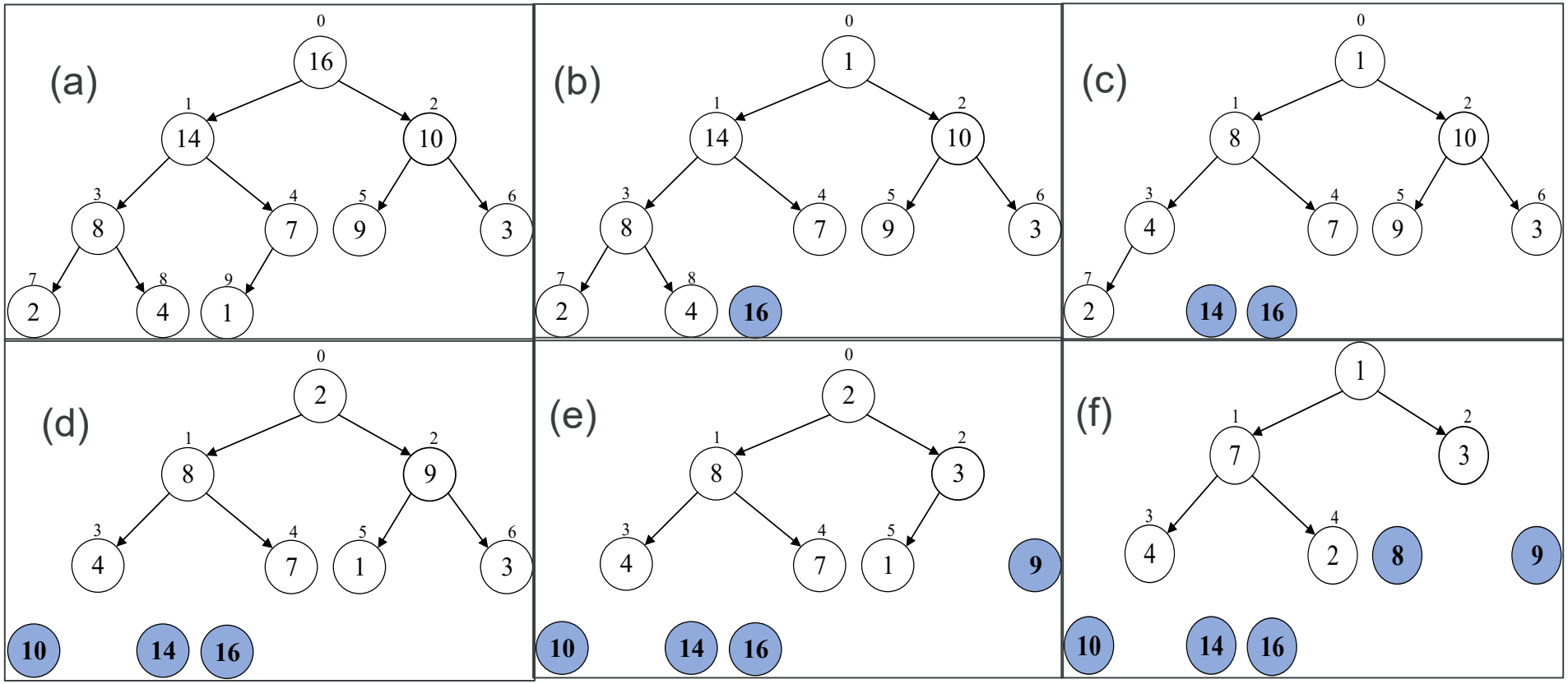
Sắp xếp dãy số sau bằng Heap Sort:

0	1	2	3	4	5	6	7	8	9
4	1	3	2	16	9	10	14	8	7

Ghi lại dãy số dưới dạng cây:

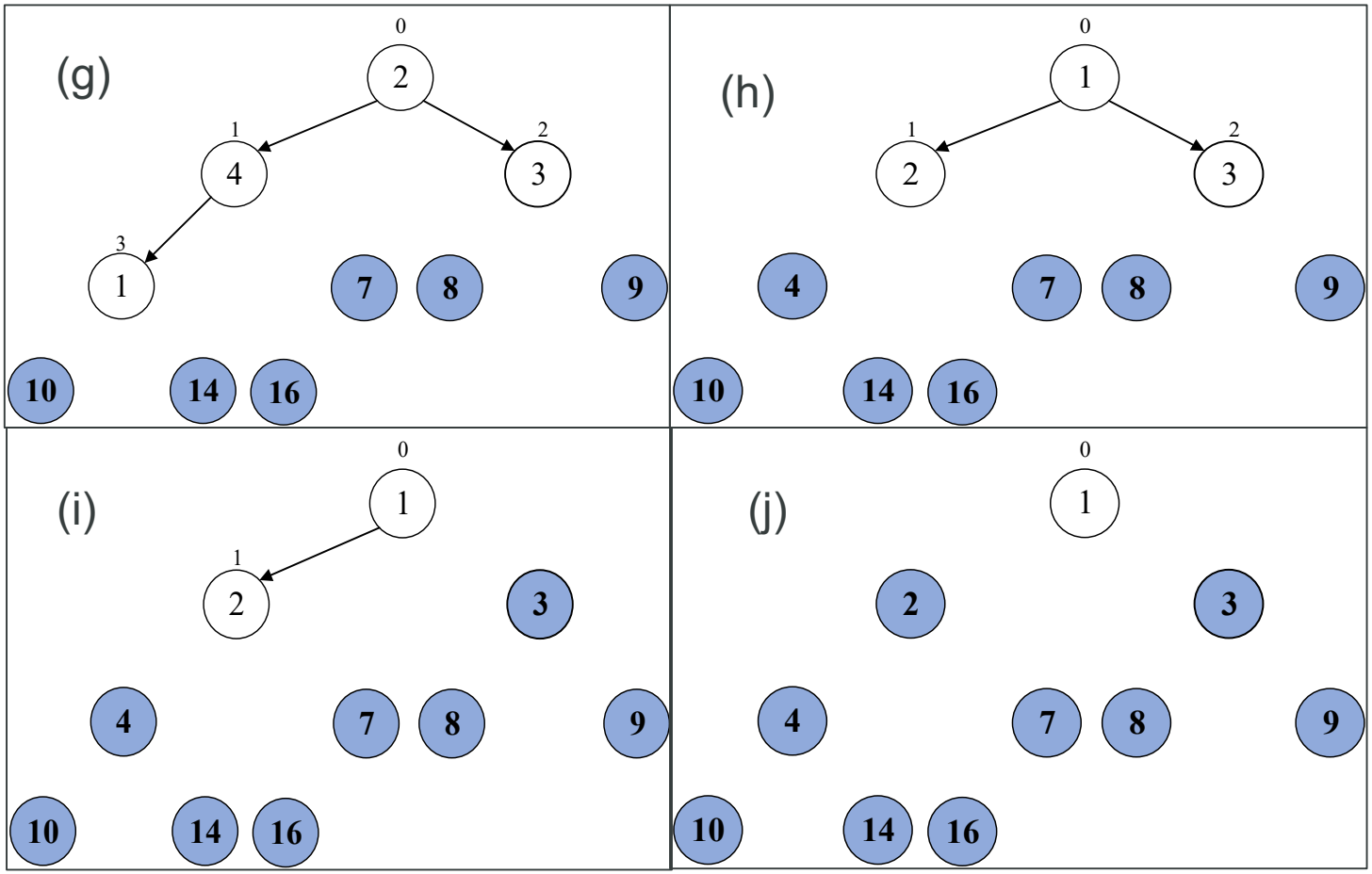


Heap Sort: Minh họa (tt)



Hình (a): The heap just after it has been built by **BUILD-HEAP.**
Hình (b)-(j): The heap just after calling of **HEAPIFY** in line 5.

Heap Sort: Minh họa (tt)



1	2	3	4	7	8	9	10	14	16

The resulting sorted array A.

Bài tập về nhà

1. Is the sequence (23, 17, 14, 6, 13, 10, 1, 5, 7, 12) a heap?
2. Illustrate the operation of HEAPIFY(A,3,) on the array A= (27, 17,3,16, 13, 10, 1,5,7, 12,4,8,9,0).
3. The code for HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient HEAPIFY that uses an iterative control construct (a loop) instead of recursion.
4. Illustrate the operation of BUILD-HEAP on the array A = (5, 3, 17, l0, 84, 19, 6, 22, 9)
5. Show that the running time of heapsort is Ω(nlgn).
6. Illustrate the operation of HEAPSORT on the array A = (5, 13, 2, 25, 7, 17, 20, 8, 4)

Heap Sort: Độ phức tạp

Trường hợp	Độ phức tạp
Tốt nhất	$nlog_2n$
Trung bình	$nlog_2n$
Xấu nhất	$nlog_2n$

Nội dung

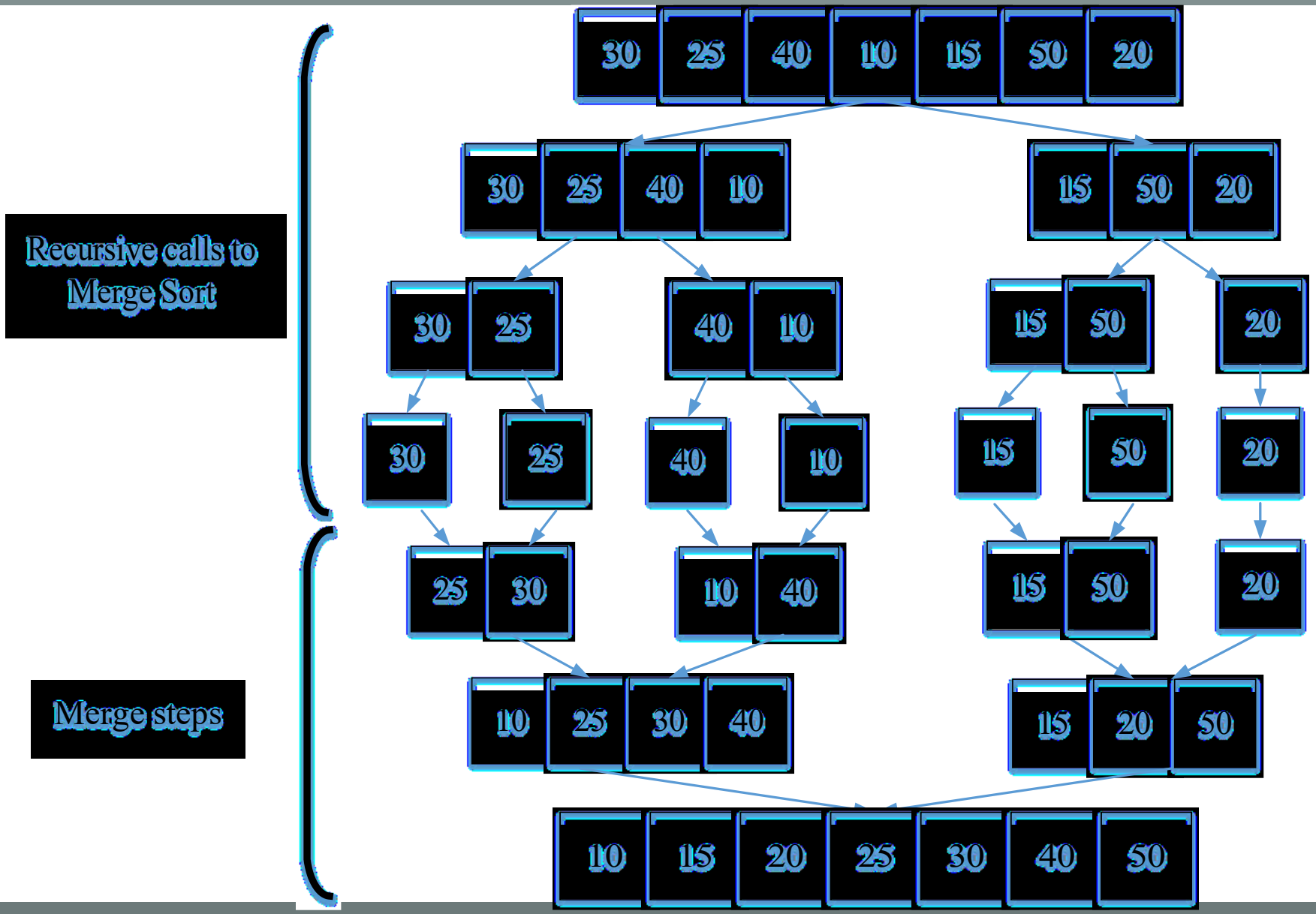
1. Quick Sort
2. Heap Sort
3. **Merge Sort**
4. Radix Sort

Merge Sort

- Thuật toán Mergesort áp dụng thuật toán chia để trị (divide and conquer) được phát minh bởi John von Neumann in 1945.
- Về mặt khái niệm, Thuật toán Mergesort hoạt động như sau:
  - Chia danh sách gồm n phần tử ban đầu thành n danh sách con (sublist), mỗi danh sách chứa 1 phần tử (danh sách 1 phần tử là danh sách có thứ tự).
  - Lặp lại việc trộn (merge) để tạo ra các danh sách con mới có thứ tự cho tới khi chỉ còn 1 danh sách con. Đây sẽ là danh sách được sắp xếp.

Top-down Merge Sort

Top-down Merge Sort: Minh họa



Top-down Merge Sort: Code C/C++

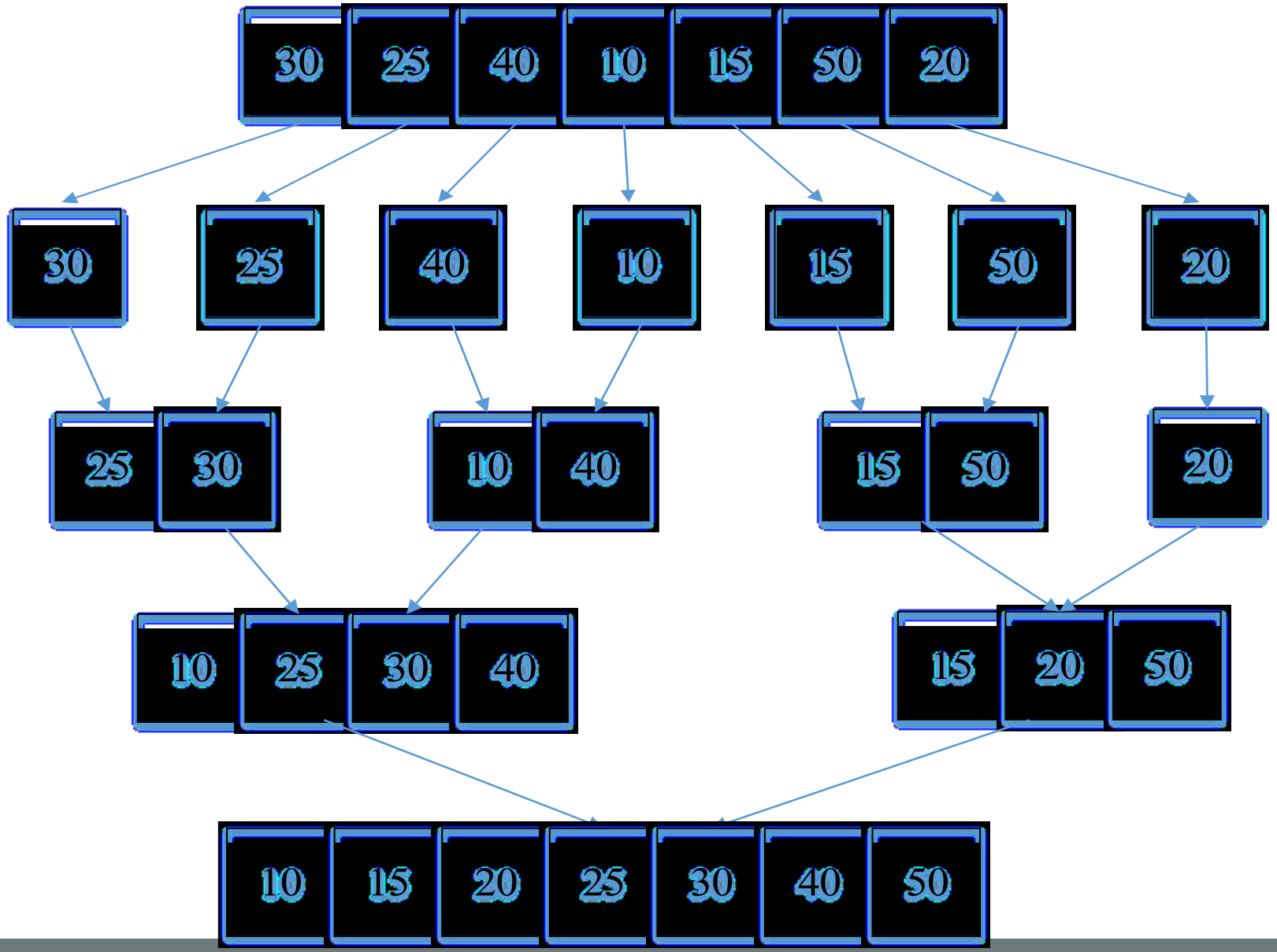
```
void MergeSort(int a[], int n) {
      TopDownMergeSort(a, 0, n - 1);
}

void TopDownMergeSort(int a[], int l, int r) {
      if (l < r) {
            int mid = (l + r) / 2;
            TopDownMergeSort(a, l, mid);
            TopDownMergeSort(a, mid + 1, r);
            TopDownMerge(a, l, r);
      }
}
```

Top-down Merge Sort: Code C/C++ (tt)

```
void TopDownMerge(int* a, int l, int r) {
    int *b, nb = r - l + 1; // Tạo mảng b lưu giá trị mảng a[l->r]
    b = new int[nb];
    Copy(b, a, l, r);

    int mid = (nb-1) / 2;
    // i0 là phần từ đầu tiên của dãy con thứ nhất trong b[0->mid]
    // i1 là phần từ đầu tiên của dãy con thứ nhất trong b[mid+1..nb-1]
    int i0 = 0, i1 = mid + 1;

    for (int j = l; j <= r; j++) {
      if (i0 <= mid && (i1 >= nb || b[i0] < b[i1]))
            a[j] = b[i0++];
      else
            a[j] = b[i1++];
    }
}
void Copy(int b[], int a[], int l, int r) {
    int nb = r - l + 1;
    for (int i = 0; i < nb; i++)
         b[i] = a[i+l];
}
```

Bottom-up Merge Sort

Bottom-up Merge Sort: Minh họa



Bottom-up Merge Sort: Code C/C++ (tt)

```
void BottomUpMergeSort(int a[], int n) {
     int *b = new int[n];
     Copy(b, a, 0, n - 1);

     for (int width = 1; width < n; width = 2 * width) {
       for (int i = 0; i < n; i = i + 2 * width)
            BottomUpMerge(a, i, min(i+width, n), min(i+2*width, n)-1, b);
       Copy(a, b, 0, n - 1);
}
```

Bottom-up Merge Sort: Code C/C++

```
void BottomUpMerge(int a[], int iLeft, int iRight, int iEnd, int b[]) {
    int i = iLeft, j = iRight;
    // While there are elements in the left or right runs...
    for (int k = iLeft; k <= iEnd; k++)
    // If left run head exists and is <= existing right run head.
      if (i < iRight && (j > iEnd || a[i] <= a[j]))
            b[k] = a[i++];
      else b[k] = a[j++];
}
```

Merge Sort: Độ phức tạp

Trường hợp	Độ phức tạp
Tốt nhất	$nlog_2n$
Trung bình	$nlog_2n$
Xấu nhất	$nlog_2n$

Nội dung

- Các giải thuật sắp xếp nội:

1. Chọn trực tiếp - Selection Sort
2. Đổi chỗ trực tiếp - Interchange Sort
3. Nổi bọt - Bubble Sort
4. Shaker Sort
5. Chèn trực tiếp - Insertion Sort
6. Chèn nhị phân - Binary Insertion Sort
7. Shell Sort
8. Quick Sort
9. Heap Sort
10. Merge Sort
11. **Radix Sort**

Radix Sort

- Nếu như trong các thuật toán khác, cơ sở để sắp xếp luôn là việc so sánh giá trị của 2 phần tử thì Radix Sort lại dựa trên nguyên tắc phân loại thư của bưu điện. Vì lý do đó Radix Sort còn có tên là Postman's sort.
- Radix Sort không hề quan tâm đến việc so sánh giá trị của phần tử mà bản thân việc phân loại và trình tự phân loại sẽ tạo ra thứ tự cho các phần tử.
- Radixsort:
  - Xem mỗi dữ liệu là 1 chuỗi ký tự
  - Trước tiên **phân nhóm** các dữ liệu theo ký tự bên phải cùng: đặt các dữ liệu vào chung một nhóm theo ký tự bên phải cùng.
  - Sau đó **kết nối** các nhóm lại với nhau
  - Lập lại việc **phân nhóm** và **kết nối** cho tất cả các vị trí trong các dữ liệu từ phải qua trái.
  - Sau khi xét tất cả các vị trí, ta có dãy đã sắp xếp.

Radix Sort: Ý tưởng

- Mô phỏng lại qui trình trên, để sắp xếp dãy $a_0, a_1, ..., a_{n-1}$, giải thuật Radix Sort thực hiện như sau:
  - Trước tiên, ta có thể giả sử mỗi phần tử a_i trong dãy $a_0, a_1, ..., a_{n-1}$ là một số nguyên có tối đa m chữ số.
  - Ta phân loại các phần tử lần lượt theo các chữ số hàng đơn vị, hàng chục, hàng trăm, … tương tự việc phân loại thư theo tỉnh thành, quận huyện, phường xã, ….

Radix Sort: Thuật toán (tt)

- Bước 1:// k cho biết chữ số dùng để phân loại hiện hành
  - k = 0; // k = 0: hàng đơn vị; k = 1: hàng chục; …
- Bước 2: //Tạo các lô chứa các loại phần tử khác nhau
  - Khởi tạo 10 lô B_0, B_1, …, B_9 rỗng;

Radix Sort: Thuật toán (tt)

- Bước 3:
  - For i = 0 .. n-1 do
    - Đặt a_i vào lô B_t với t: chữ số thứ k của a_i;
- Bước 4:
  - Nối B_0, B_1, …, B_9 lại (theo đúng trình tự) thành a.
- Bước 5:
  - k = k+1; Nếu k < m thì trở lại bước 2. Ngược lại: Dừng

Radix Sort: Minh họa (tt)

12	0701									
11	1725									
10	0999									
9	9170									
8	3252									
7	4518									
6	7009									
5	1424									
4	0428									
3	1239									0999
2	8425						1725			4518 7009
1	7013	9170	0701	3252	7013	1424	8425			0428 1239
CS	A	0	1	2	3	4	5	6	7	8 9

Radix Sort: Minh họa (tt)

12	0999										
11	7009										
10	1239										
9	4518										
8	0428										
7	1725										
6	8425										
5	1424										
4	7013			0428							
3	3252			1725							
2	0701	7009	4518	8425							
1	9170	0701	7013	1424	1239		3252		9170		0999
CS	A	0	1	2	3	4	5	6	7	8	9

Radix Sort: Minh họa (tt)

12	0999										
11	9170										
10	3252										
9	1239										
8	0428										
7	1725										
6	8425										
5	1424										
4	4518										
3	7013					0428					
2	7009	7013		3252		8425			1725		
1	0701	7009	9170	1239		1424	4518		0701		0999
CS	A	0	1	2	3	4	5	6	7	8	9

Radix Sort: Minh họa (tt)

12	0999										
11	1725										
10	0701										
9	4518										
8	0428										
7	8425										
6	1424										
5	3252										
4	1239										
3	9170	0999	1725								
2	7013	0701	1424						7013		
1	7009	0428	1239		3252	4518			7009	8425	9170
CS	A	0	1	2	3	4	5	6	7	8	9

Radix Sort: Minh họa (tt)

12	**9**170										
11	**8**425										
10	**7**013										
9	**7**009										
8	**4**518										
7	**3**252										
6	**1**725										
5	**1**424										
4	**1**239										
3	**0**999										
2	**0**701										
1	**0**428										
CS	A	0	1	2	3	4	5	6	7	8	9

Radix Sort: Code C/C++ (tt)

```
void RadixSort(int arr[], int n) {
    // Find the maximum number to know number of digits
    int m = GetMax(arr, n);

    // Do counting sort for every digit. Note that instead of passing digit
    // number, exp is passed. exp is 10^i where i is current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
            CountingSort(arr, n, exp);
}

int GetMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Radix Sort: Code C/C++ (tt)

```
// A function to do counting sort of arr[] according to the digit represented by exp.
void CountingSort(int arr[], const int n, int exp) {
    int *output = new int[n]; // output array
    int i, count[10];
    memset(count, 0, sizeof(count)); // or count[10] = { };

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++) count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains actual position of this digit in output[]
    for (i = 1; i < 10; i++) count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to curent digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

Radix Sort: Độ phức tạp

- **Độ phức tạp**: $O(n)$

RadixSort: VD2

mom, dad, god, fat, bad, cat, mad, pat, bar, him	original list
(dad,god,bad,mad) (mom,him) (bar) (fat,cat,pat) by rightmost letter	group strings
dad,god,bad,mad,mom,him,bar,fat,cat,pat	combine groups
(dad,bad,mad,bar,fat,cat,pat) (him) (god,mom) by middle letter	group strings
dad,bad,mad,bar,fat,cat,pat,him,god,mom	combine groups
(bad,bar) (cat) (dad) (fat) (god) (him) (mad,mom) (pat) strings by first letter	group
bad,bar,cat,dad,fat,god,him,mad,mom,par groups (SORTED)	combine

RadixSort: VD3

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix sort	n	n
Treesort	n^2	n * log n
Heapsort	n * log n	n * log n

Câu hỏi và Bài tập

1. Trình bày ý tưởng của 3 thuật toán sắp xếp Quick sort, Merge sort, Radix sort?

2. Hãy trình bày những ưu khuyết điểm của 3 thuật toán sắp xếp ở câu 1? Theo bạn cách khắc phục những nhược điểm này là như thế nào?

3. Sử dụng hàm random trong C để tạo ra một dãy M có 1.000 số nguyên. Vận dụng 3 thuật toán sắp xếp ở câu 1 để sắp xếp các phần tử của mảng M theo thứ tự tăng dần về mặt giá trị. Với cùng một dữ liệu như nhau, cho biết thời gian thực hiện các thuật toán?

Câu hỏi và Bài tập

4. Thông tin về một sinh viên bao gồm: Mã số (là một số nguyên dương), Họ và đệm (là một chuỗi có tối đa 20 ký tự), Tên sinh viên (là một chuỗi có tối đa 10 ký tự), Ngày, tháng, năm sinh (là các số nguyên dương), Điểm trung bình (là các số thực có giá trị từ 0.00 ? 10.00).

Viết chương trình nhập vào danh sách sinh viên (ít nhất là 10 sinh viên, không nhập trùng mã giữa các sinh viên với nhau), sau đó vận dụng các thuật toán sắp xếp để sắp xếp danh sách sinh viên theo thứ tự tăng dần theo Mã sinh viên. In danh sách sinh viên sau khi sắp xếp ra màn hình.

Chúc các em học tốt!