Data structure and Algorithms

Sorting – Giải thuật sắp xếp

Thanh-Hai Tran

Electronics and Computer Engineering
School of Electronics and Telecommunications

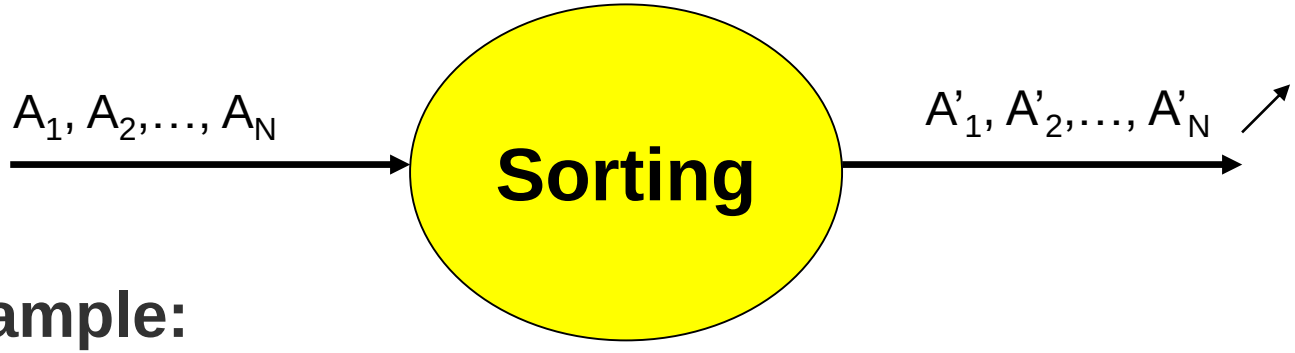
Hanoi University of Science and Technology
1 Dai Co Viet - Hanoi - Vietnam

Outline

- **Introduction**
- **Basic sorting algorithms**
  - Selection sort (Sắp xếp chọn)
  - Bubble sort (Sắp xếp nổi bọt)
  - Insertion (Sắp xếp chèn)
- **Advanced sorting algorithms**
  - Quick sort (Sắp xếp nhanh)
  - Heap sort (Sắp xếp vun đống)
  - Merge sort (Sắp xếp trộn)
- **Exercises**
- **References**

Introduction

- **Sorting problem:**
  - Given a set of N elements A_1, A_2, …, A_N.
  - Arrange the elements so that they are placed in some relevant order (ascending or descending)



- **Example:**
  - Input: `int A[] = {21, 34, 11, 9, 1, 0, 22};`
  - Output: `A[] = {0, 1, 9, 11, 21, 22, 34;`
- **Sorting types:**
  - Internal sorting: data stored in computer memory
  - External sorting: data stored in files

Sorting on Multiple Keys

- **In real-world applications, it is desired to sort arrays of records using multiple keys**
- **Examples:** Employees of a big company are sorted by Department first then name in alphabetical order.

Name	Department	Salary	Phone Number
Janak	Telecommunications	1000000	9812345678
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Divya	Computer Science	750000	9350123455

- **Notes:**
  - Data records can be sorted based on a property. Such a component or property is called a **sort key**
  - A sort key can be defined using two or more sort keys. In such a case, the first key is called the **primary sort key**, the second is known as the **secondary sort key**, etc

Sorting on Multiple Keys

- **In real-world applications, it is desired to sort arrays of records using multiple keys**
- **Examples:**
  - Telephone directories in which names are sorted by location, category (business or residential), and then in an alphabetical order
  - In a library, the information about books can be sorted alphabetically based on titles and then by authors' names
  - Customers' addresses can be sorted based on the name of the city and then the street

Practical considerations for Internal Sorting

- Number of sort key comparisons that will be performed
- Number of times the records in the list will be moved
- Best case performance
- Worst case performance
- Average case performance
- Stability of the sorting algorithm where stability means that equivalent elements or records retain their relative positions even after sorting is done

Classification of Sorting algorithms

- **Basic SA**: at each step,
  - Try to move the current element into its right position in the array
  - Don't care about other elements in the array

    7 12 8 9 7 11 10 ($O(n^2)$)
- **Advance SA**: at each step
  - Move the current element into its right position,
  - Re-arrange the remaining element to reduce the operations at further steps.
- **Notes:** ASAs run faster than BSAs but harder for implementation.

Selection sort

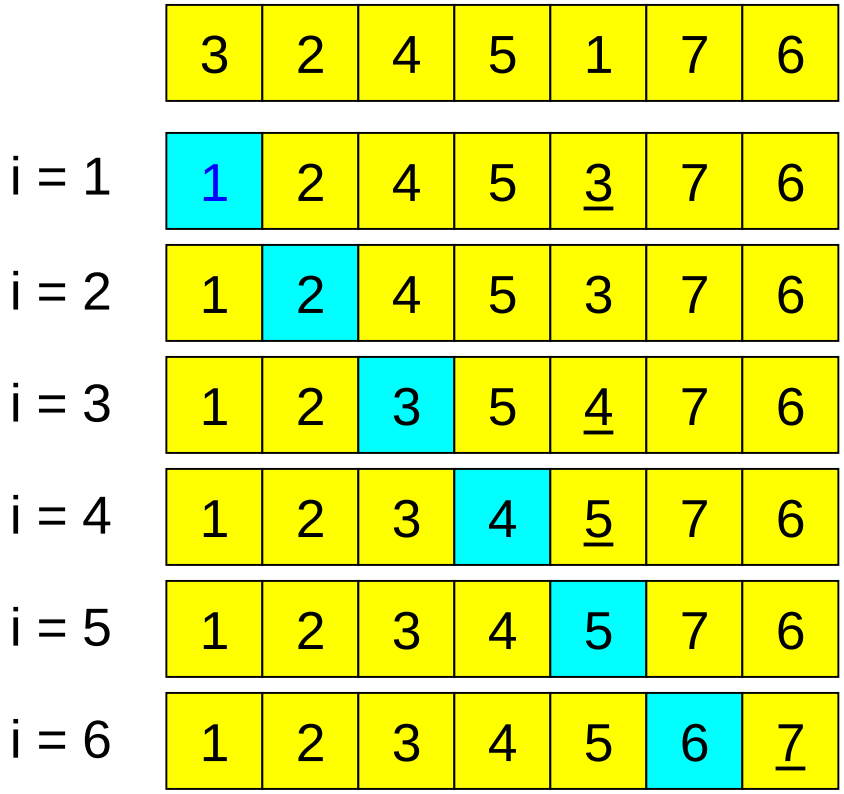
- **Idea:**
  - First find the smallest value in the array and place it in the first position.
  - Then, find the second smallest value in the array and place it in the second position.
  - Repeat this procedure until the entire array is sorted.
- **Algorithm:**

```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
      [END OF LOOP]
Step 4: EXIT
```

Selection sort

- **Give an array of 7 following elements**



Selection sort in C/C++

```
void selectionSort(int A[], int N) {
    int m;
    for (int i=0; i < N-1; i++){
        m=i;
        for (int k=i+1; k < N; k++)
            if (A[k] < A[m]) m=k;
        if (m != i) swap(A[i],A[m]);
    }
}
```

Complexity of selection sort

- **Selection sort is a sorting algorithm that is independent of the original order of elements in the array**
  - In Pass 1, selecting the element with the smallest value calls for scanning all **n** elements
  - Thus, **n–1** comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position.
  - In Pass 2, selecting the second smallest value requires scanning the remaining **n – 1** elements and so on
- **Then**

$$(n - 1) + (n - 2) + \ldots + 2 + 1$$
$$= n(n - 1) / 2 = O(n^2) \text{ comparisons}$$

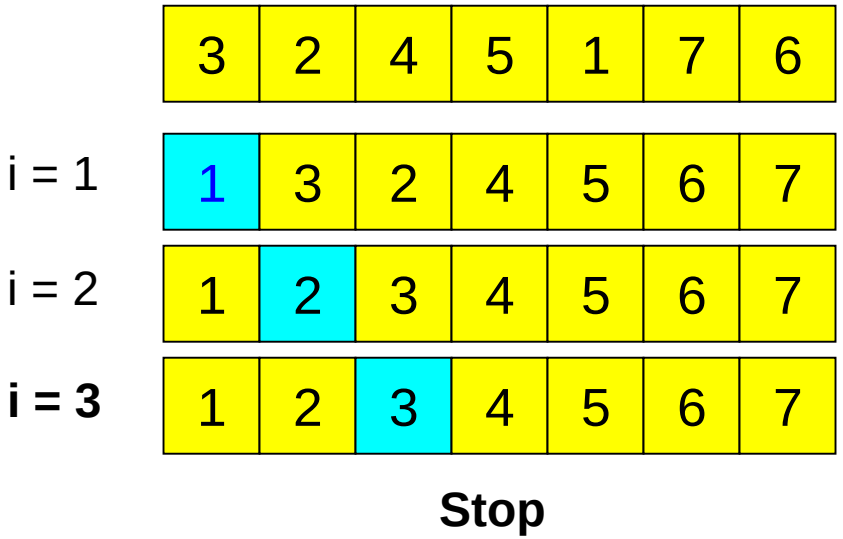
Notices of selection sort

- Best case:
  - The array was sorted
  - 0 swap operation, n(n-1)/2 comparison
- Worst case:
  - N-1 swap operation, n(n-1)/2 comparision
- Average: $O(n^2)$
- Notices:
  - It is simple and easy to implement.
  - It can be used for small data sets.
  - Selection sort is generally used for sorting files with very large objects (records) and small keys

Bubble sort

- **Idea:**
  - Repeatedly moving the smallest element to the lowest index position of the array segment
  - Consecutive adjacent pairs of elements in the array are compared with each other
  - If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one
  - This process will continue till the list of unsorted elements exhausts

Example



Bubble sort

```
i = 1;
sorted = False;
while (!sorted && i<N) {
    sorted = True;
    for (k=N-1;k>=i;k--)
        if (a_k > a_k+1) {
            swap(a_k, a_k+1);
            sorted = False;
        }
    i++;
}
```

Bubble sort in C/C++

```
void bubbleSort(int A[], int N) {
  int i = 0;
  bool sorted = false;
  while (!sorted && i<N-1) {
    sorted = true;
    for (k=N-2;k>=i;k--)
      if (A[k] > A[k+1]) {
        swap(A[k], A[k+1]);
        sorted = false;
      }
    i++;
  }
}
```

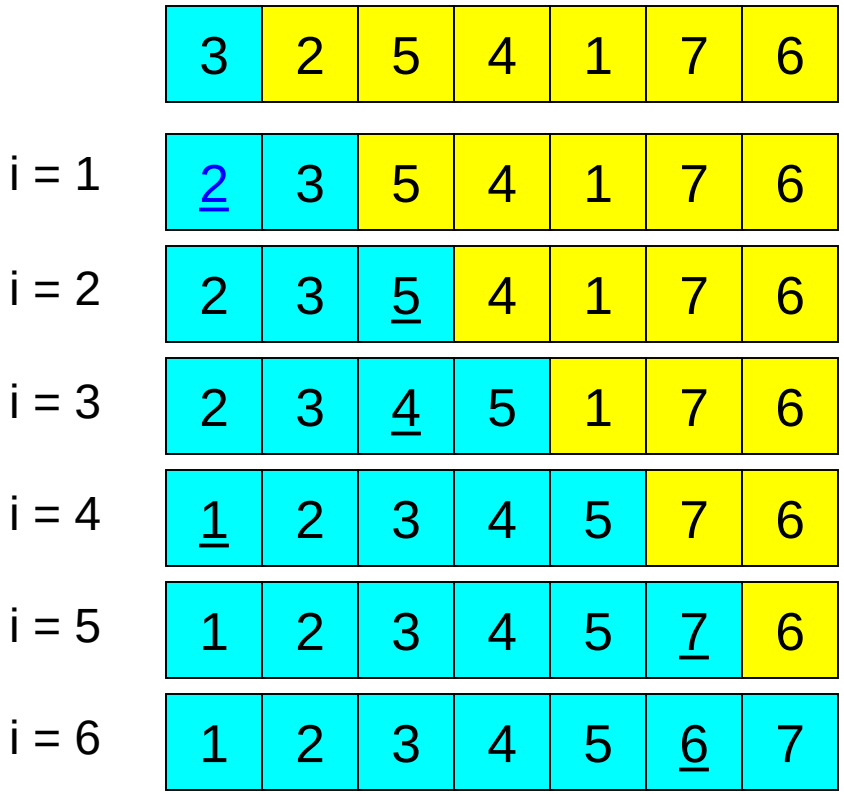
Notices about bubble sort

- **Best case:**
  - The array was sorted.
  - 0 swap, n(n-1)/2 comparisions
- **Worst case:**
  - n(n-1)/2 swaps and n(n-1)/2 comparisions
- **Average complexity:** $O(n^2)$

Insertion sort

- **Idea:** Insert each item into its proper place in the final list
- Technique:
  - The array of values to be sorted is divided into two sets.
  - Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.
  - The first element of the unsorted partition has array index 1 (if LB = 0).
  - During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Insertion sort



Insertion sort

```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:      SET TEMP = ARR[K]
Step 3:      SET J = K - 1
Step 4:      Repeat while TEMP <= ARR[J]
                    SET ARR[J + 1] = ARR[J]
                    SET J = J - 1
             [END OF INNER LOOP]
Step 5:      SET ARR[J + 1] = TEMP
         [END OF LOOP]
Step 6: EXIT
```

Insertion sort in C/C++

```
void insertionSort(int A[], int N) {
   int i,k,b;
    for (i=0;i < N-1;i++){
      k = i;
      b = A[i+1];
      while (k>=0 && b<A[k]){
          A[k+1] = A[k];
          k--;
      }
      A[k+1] = b;
    }
}
```

Notices about insertion sort

- **Best case:**
  - The array was sorted.
  - 0 swap, n(n-1)/2 comparisions
- **Worst case:**
  - n(n-1)/2 swaps and n(n-1)/2 comparisions
- **Average complexity:** $O(n^2)$

Summary about basic sorting algos

- Simple idea and implementation
- Average Complexity: $O(n^2)$

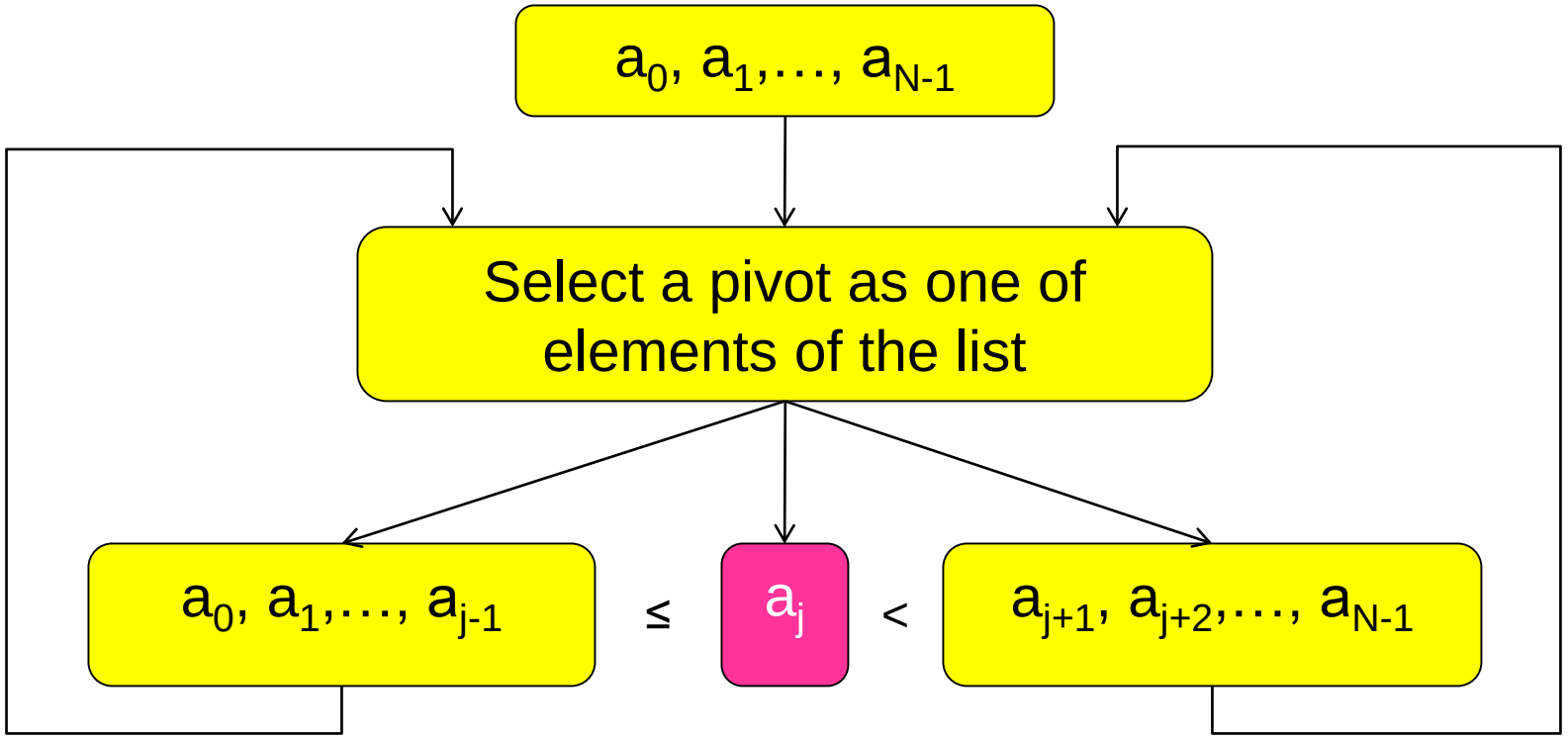
Quick sort

- **Developed by C. A. R. Hoare (1962)**
- **Average case:** O(nlogn)
- **Worst case:** $O(n^2)$
- Basically, the quick sort algorithm is faster than other O(nlogn) algorithms, because its efficient implementation can minimize the probability of requiring quadratic time
- Quick sort is also known as partition exchange sort
- Quick sort uses a divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays

Quick sort main steps

- Select an element pivot from the array elements.
- Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the *partition* operation.
- Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Quick sort


$a_0, a_1, \ldots, a_{N-1}$
Select a pivot as one of elements of the list
$a_0, a_1, \ldots, a_{j-1}$
≤
a_j
<
$a_{j+1}, a_{j+2}, \ldots, a_{N-1}$

Quick sort

- **Partition:** Divide the input A[left..right] into two parts:
  - A[left, p-1] includes elements equal or less than A[p]
  - A[p+1, right] includes elements equal or bigger than A[p]
- **Pivot selection:**
  - First or last element of the list
  - The element at the middle
  - The element at random position

Partition algorithm

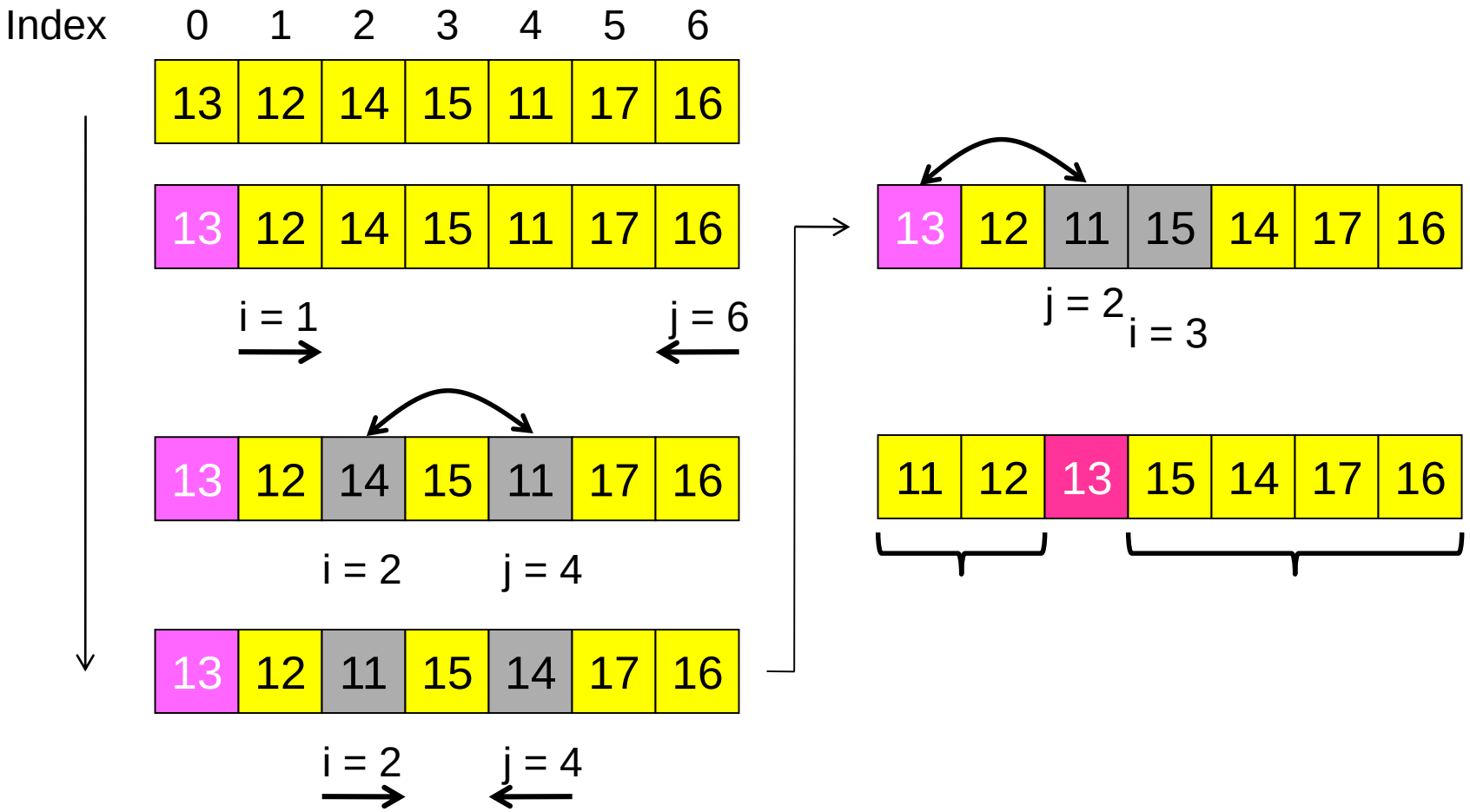
- **Input: (a_f, a_{f+1},…, a_h)**
- **Output: Dãy (a_f, a_{f+1},…, a_{j-1}, a_j, a_{j+1}, …, a_h) so that:**

  $(a_f, a_{f+1},\ldots, a_{j-1}) \le a_j < (a_{j+1}, \ldots, a_h)$
- **Algorithm:**
  - Select pivot c = a_f;
  - Init i = f+1; and j=h
  - Move the elements (not pivot) into their proper positions

```
While (i ≤ j) {
        while (a_i ≤ c) i++;
        while (a_j > c) j--;
        If (i<j) Swap (a_i, a_j);
}
```

  - Move the pivot into the right position: swap (a_f, a_j);
  - Stop when the list has only one element or empty

Example


Index
0 1 2 3 4 5 6
13 12 14 15 11 17 16
13 12 14 15 11 17 16
i = 1
j = 6
13 12 14 15 11 17 16
i = 2
j = 4
13 12 11 15 14 17 16
i = 2
j = 4
13 12 11 15 14 17 16
j = 2
i = 3
11 12 13 15 14 17 16

Quick sort in C/C++

```
void Partition(int A[], int first, int last){
        if (first>=last) return;
        int c=A[first];
        int i=first+1,j=last;
        while (i<=j){
                while (A[i]<=c && i<=j) i++;
                while (A[j]>c && i<=j) j--;
                if (i<j) swap(A[i],A[j]);
        }
        swap(A[first],A[j]);
        Partition(A, first,j-1);
        Partition(A, j+1,last);
}
```

```
void QuickSort(int A[], int N){
      Partition(A,0,N-1);
}
```

Quick sort

- **Best case:**
  - Every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size.
  - At the most, only log n nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is O(log n).
  - And because at each level, there can only be O(n), the resultant time is given as O(nlog n) time
- **Worst case:**
  - Occurs when the array is already sorted either in ascending or descending order) and the left-most element is chosen as the pivot
  - $O(n^2)$
- **The randomized version of the quick sort algorithm always has an algorithmic complexity of O(n log n)**

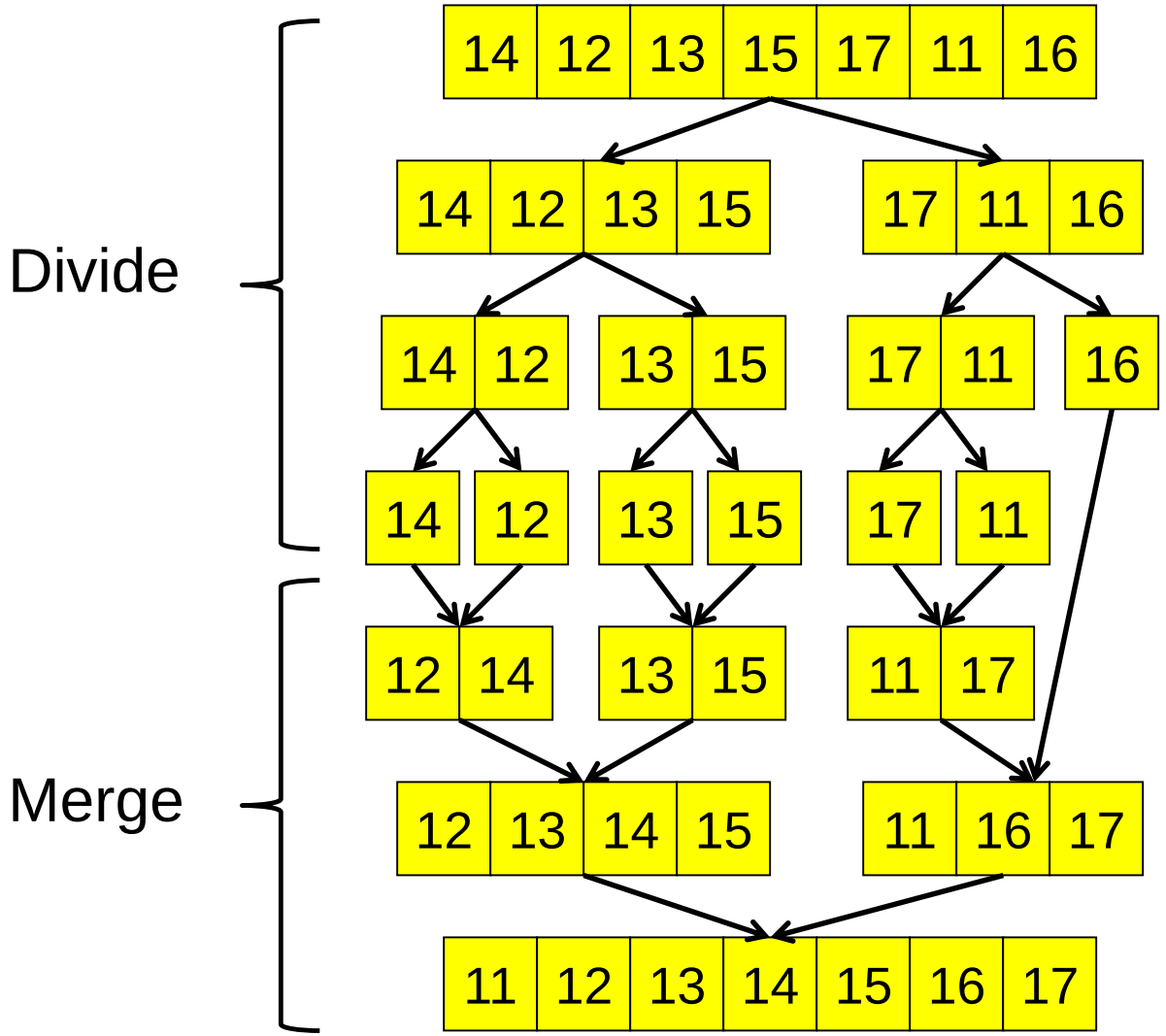
Merge Sort

- **Idea:** Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm
  - *Divide* means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements
  - *Conquer* means sorting the two sub-arrays recursively using merge sort
  - *Combine* means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

Merge sort

- **Basic steps:**
  - If the array is of length 0 or 1, then it is already sorted.
  - Otherwise, divide the unsorted array into two sub-arrays of about half the size.
  - Use merge sort algorithm recursively to sort each sub-array.
  - Merge the two sub-arrays to form a single sorted list

Merge sort example



Merge sort

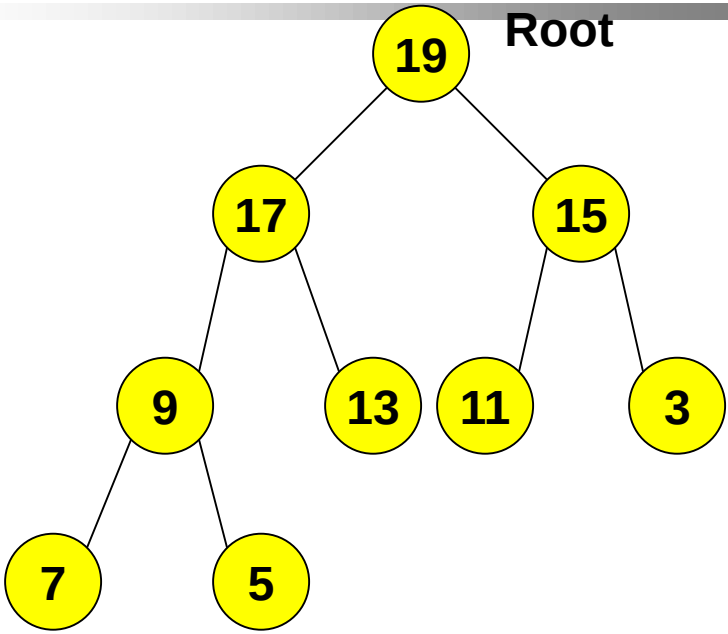
```
//Tron 2 day con ma da duoc sap xep trong A
//L1=A[m],A[m+1],...,A[n];
//L2=A[n+1],A[n+2],...,A[p]
void MergeArrays(int A[],int m, int n, int p){
    int i=m,j=n+1;
    while (i<j && j<=p){
        if (A[i]<=A[j]) i++;
        else {//chen Aj vao vi tri i
            int x=A[j];
            for (int k=j-1;k>=i;k--)
                A[k+1]=A[k];
            A[i]=x;
            i++; j++;
        }
    }
}
```

Merge sort

```
void Merge(int A[], int first, int last){
    if (first>=last) return;
    int m=(first+last)/2;
    Merge(A,first,m);
    Merge(A,m+1,last);
    MergeArrays(A,first,m,last);
}

void MergeSort(int A[], int N){
    if (N<2) return;
    Merge(A,0,N-1);
}
```

Heap Sort

- **Heap:** A complete tree, the root of every tree has maximal value
- **Build a heap:** convert a complete tree into a heap.
- **Heap sort:**
  - Step 1: Build a heap from elements of the array
  - Step 2: Repeatedly delete the root elements of the heap formed in step 1

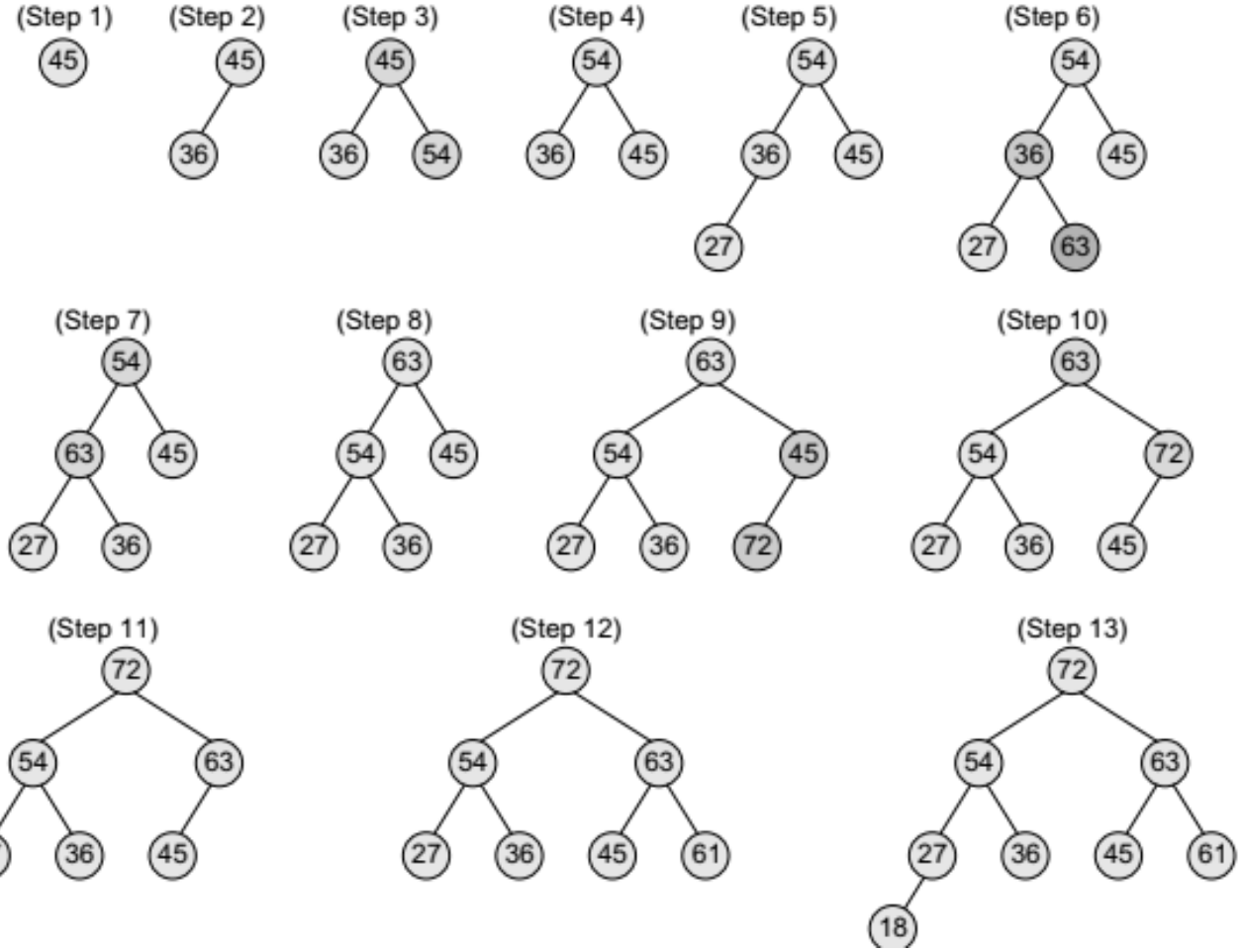


Building a heap

- Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:
  - 1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.
  - 2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well

Example

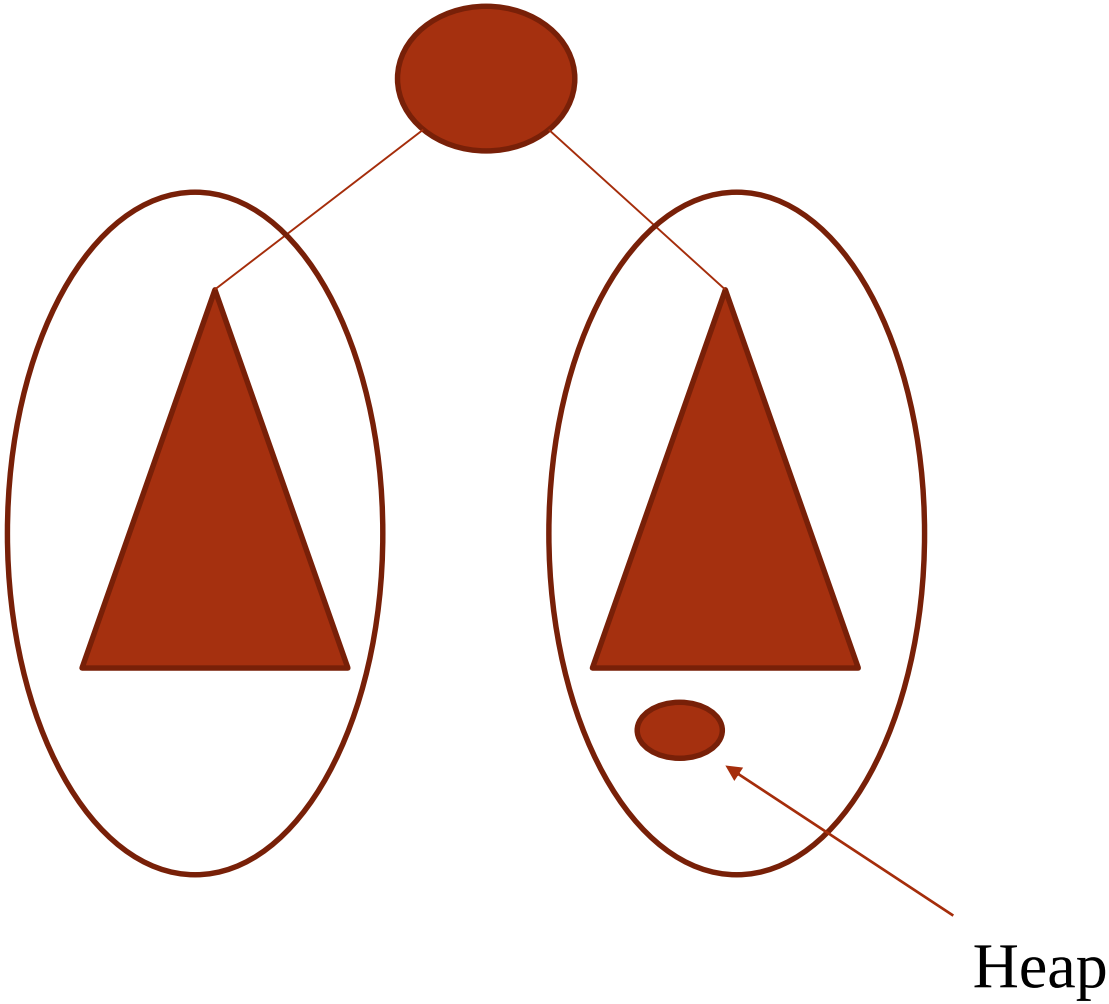
- Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18



Building a heap from a BT

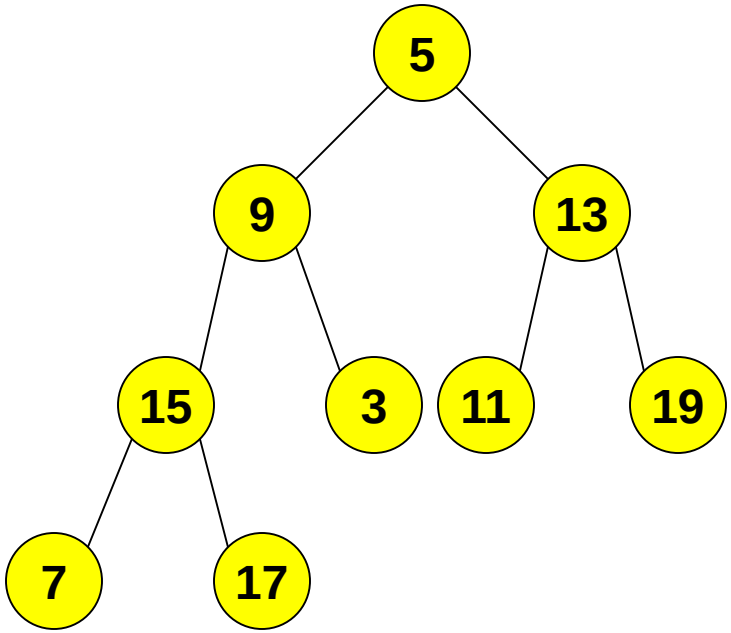
- **How to build a heap ?**
  - A Complete BT is a heap if its left sub-tree and right sub-tree are heap
  - A BT has n nodes then has $\lfloor n/2 \rfloor$ parent nodes
  - A leaf is a heap
- **To build a heap:**
  - Start from sub-tree with root are: $\lfloor n/2 \rfloor$; $\lfloor n/2 \rfloor$-1; $\lfloor n/2 \rfloor$-2; …; 1.
  - Start with sub-tree as leaf.

Recursive definition of heap

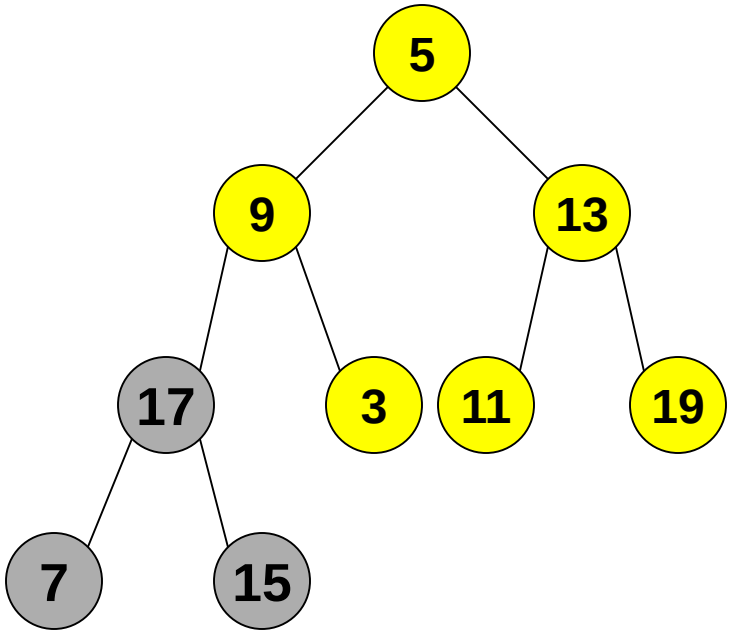


Example

Initial array: 5, 9, 13, 15, 3, 11, 19, 7, 17

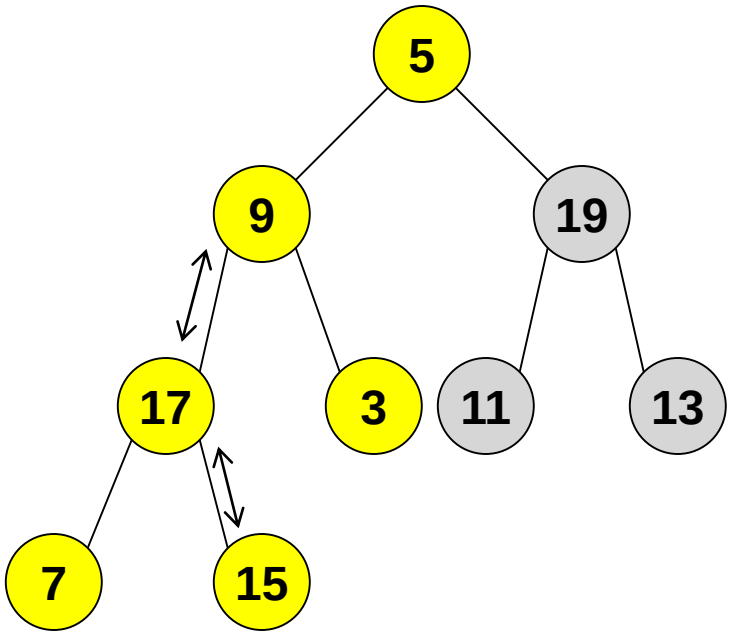


Binary Tree

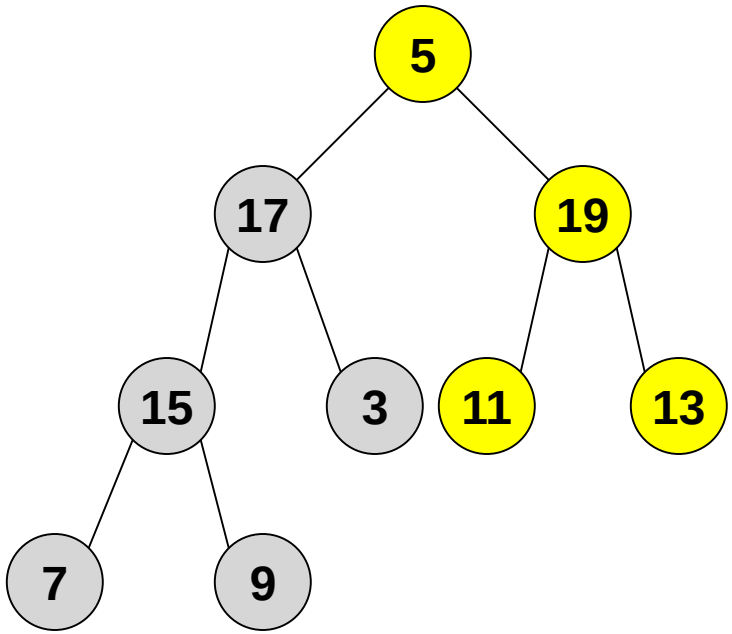


1.

Example

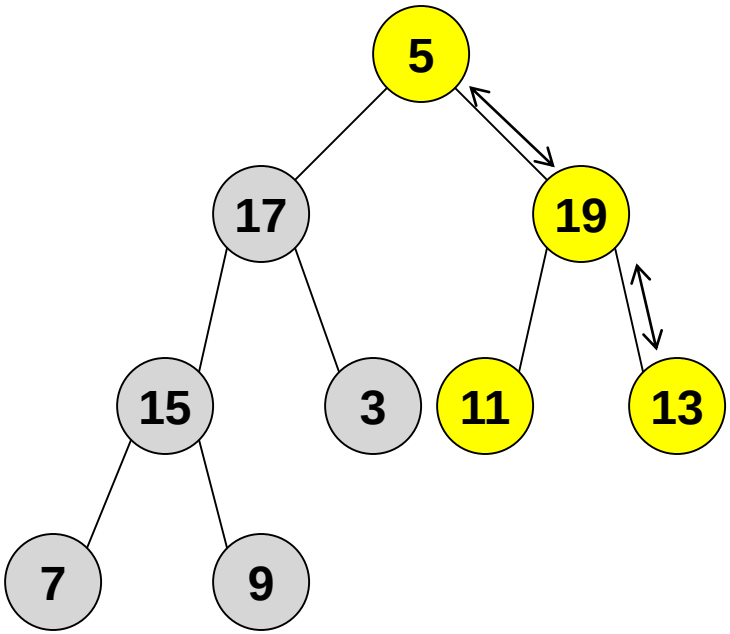


2.

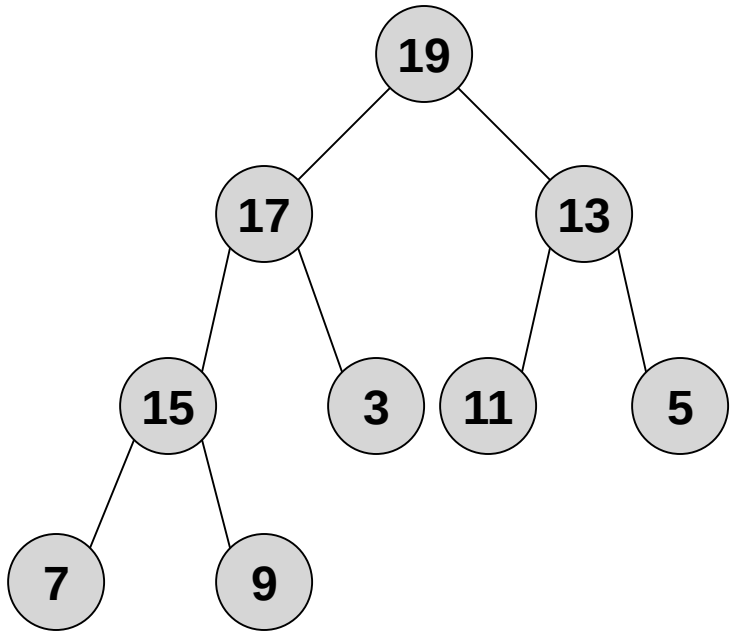


3.

Example



4.

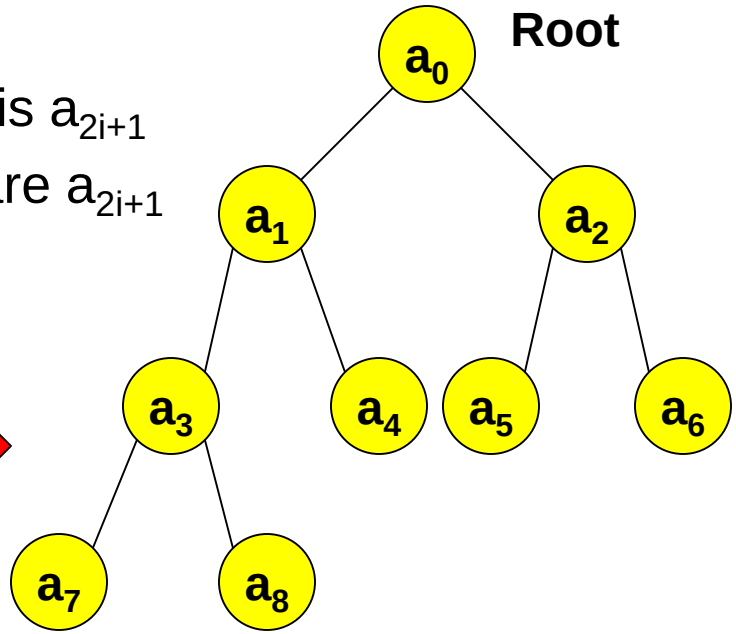


5.

Heap Sort

- **Relation between binary tree and the array (a_0, a_1,…, a_{n-1}):**
  - Consider each elements of arra as a node of the tree
  - An internal node a_i :
    - If it has 1 child then the child is a_{2i+1}
    - If it has 2 children then they are a_{2i+1} and a_{2i+2}

$a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8$ ⟺



MakeSubHeap

```
void MakeSubHeap(int A[], int r, int N){
   if (2*r+1 >= N) return;
   //Tim con ma co gia tri lon nhat
   int maxChid=A[2*r+1];
   int c=2*r+1;
   if (2*r+2<N)
       if (A[2*r+2]>A[2*r+1]){
             maxChid=A[2*r+2];
             c=2*r+2;
       }
   if (A[r]<A[c]) {
       swap(A[r],A[c]);
       MakeSubHeap(A,c,N);
   }
}
```

MakeHeap

```
void MakeHeap(int A[], int N){
   if (N<2) return;
   int m=N/2-1; //Vị trí nút trong ngoài cùng
   for (int i=m;i>=0;i--)
       MakeSubHeap(A,i,N);
}
```

```
void HeapSort(int A[], int N){
   if (N<2) return;
   MakeHeap(A,N);
   for (int i=1;i<N;i++){
       swap(A[0],A[N-i]);
       MakeSubHeap(A,0,N-i);
   }
}
```

HeapSort

```
void HeapSort(int A[], int N){
  if (N<2) return;
  MakeHeap(A,N);
  for (int i=1;i<N;i++){
     swap(A[0],A[N-i]);
     MakeSubHeap(A,0,N-i);
  }
}
```

Summary

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

References

- Chương 9 – Cấu trúc dữ liệu và giải thuật – Đỗ Xuân Lôi
- Slide– Đỗ Bích Diệp
- Slide– Nguyễn Thanh Bình
- Chapter 14 – Data Structure in C – Rema thareja