

OBJECTIVES

After the lesson, students can

1. Understand the sorting algorithms: selection sort, insertion sort, bubble sort
2. Implement these sorting algorithms

CONTENTS

1. Sorting problem

2. Insertion sort
3. Selection sort
4. Bubble sort

1. SORTING PROBLEM

- Sorting
 - The process of reorganizing data in descending or ascending order
- The data to be sorted can be:
 - Integer
 - String of characters
 - Objects
- Sort key
 - The record's part determines the sort order of the record in the list.
 - Records are arranged in order of keys.

1. SORTING PROBLEM

➤ Note:

- If sorting directly on records
 - → Requires moving record location → expensive.
- Build a key table consisting of records with only 2 fields:
 - "key" contains the key value,
 - "pointer" to write the address of the corresponding record.
- Sorting on the key table does not change the main table
- But the sequence of records in the key table allows determining the sequence of records in the main table.

1. SORTING PROBLEM

➤ General case

Input: Sequence of n numbers: $A = (a_1, a_2, \dots, a_n)$

Output: A permutation (a'_1, \dots, a'_n) of the sequence that satisfies

$$a'_1 \leq \dots \leq a'_n$$

1. SORTING PROBLEM

➤ Types of sorting algorithms

▪ Internal sort (Sắp xếp trong)

- Requires the data need to be sorted being entirely in the computer's internal memory

▪ External sort (Sắp xếp ngoài)

- The data need to be sorted cannot be put entirely into internal memory at the same time, but can be read into each part from external memory

1. SORTING PROBLEM

➤ Characteristics

▪ In place (tại chỗ)

- If the extra memory space required by the algorithm is $O(1)$, that is, it is bounded by a constant that does not depend on the length of the sequence to be sorted.

▪ Stable (ổn định)

- If elements have the same value, keep their relative order as before sorting.

1. SORTING PROBLEM

➤ Two basic operations are often used

▪ Swap (đổi chỗ): computing time $O(1)$

```
void swap( datatype &a, datatype &b){  
    datatype temp = a; //datatype- data type of element  
    a = b;  
    b = temp;  
}
```

▪ Compare: Compare(a, b) returns

- **true** if a is before b in the order to be arranged
- **false** otherwise.

1. SORTING PROBLEM

- Applications of sorting:
 - Database Administrator
 - Science and technology
 - Scheduling algorithms,
 - For example, designing translation programs, communication,...
 - Web search engine
 - and many other applications

CONTENTS

1. Sorting problem

2. Insertion sort

3. Selection sort

4. Bubble sort

2. INSERTION SORT

Algorithm:

- At step $k = 1, 2, \dots, n$, Insert the k^{th} element in the given array into the correct position in the sequence of the first k elements.
- The result is that after step k , the first k elements are sorted.

Explain:

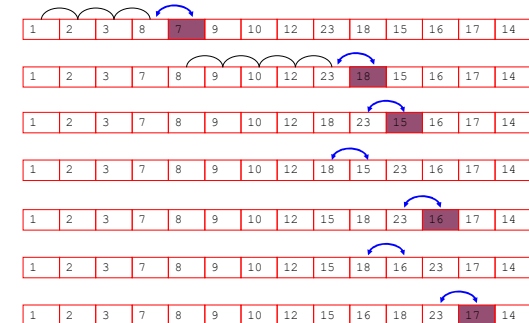
• At the beginning of iteration i of the outer "for" loop, the data from $a[0]$ to $a[i-1]$ is sorted.

• The "while" loop finds the position for the next element ($\text{last} = a[j]$) in the sequence of the first i elements.

```
void insertionSort(int a[], int array_size) {  
    int i, j, last;  
    for (i=1; i < array_size; i++) {  
        last = a[i];  
        j = i;  
        while ((j > 0) && (a[j-1] > last)) {  
            a[j] = a[j-1];  
            j = j - 1; } // end while  
        a[j] = last;  
    } // end for  
} // end of isort
```


2. INSERTION SORT


➤ Example

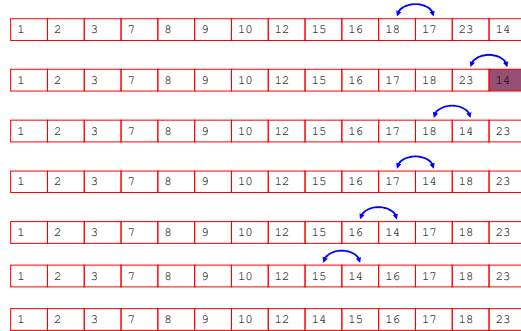


2. INSERTION SORT

➤ Example

13 swap operation: 

20 compare operation: 



2. INSERTION SORT

Characteristics of insertion sort

- In place and stable
- Computation time
 - Best Case: 0 swaps, $n-1$ comparisons (when input sequence is sorted)
 - Worst Case: $n^2/2$ swap and compare (when the input sequence has the opposite order to the order to be sorted)
 - Average Case: $n^2/4$ swaps and comparisons
 - In the best of situations, it's best
- The sorting algorithm is good for nearly sorted sequence
- Each element is located very close to the position in the order that needs to be sorted

CONTENTS

1. Sorting problem
2. Insertion sort
- 3. Selection sort**
4. Bubble sort

3. SELECTION SORT

➤ Algorithm

- Find the smallest element put into position 1
- Find the next smallest element and put it in position 2
- Find the next smallest element and put it in position 3
- ...

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

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

3. SELECTION SORT

➤ **Comment:**

- **Best case:** 0 swap, $n^2/2$ compare operations.
- **Worst case:** $n - 1$ swap and $n^2/2$ compare operations.
- **Average case:** $O(n)$ swap and $n^2/2$ compare operations.
- **Advantage:** less number of swap operations.
 - Makes sense if the swap operation is expensive.

3. SELECTION SORT

➤ **Example**

	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

CONTENTS

1. Sorting problem
2. Insertion sort
3. Selection sort
- 4. Bubble sort**

4. BUBBLE SORT

➤ **Algorithm**

- Starting from the beginning of the sequence, the algorithm compares each element with the element that comes after it and swaps them if they are not in the correct order.
- This process will be repeated until there is a traversal from the beginning to the end of the sequence without having to swap places (i.e. all elements are in the correct position).
- This approach pushes the largest element to the end of the sequence, while elements with smaller values are moved to the beginning of the sequence.

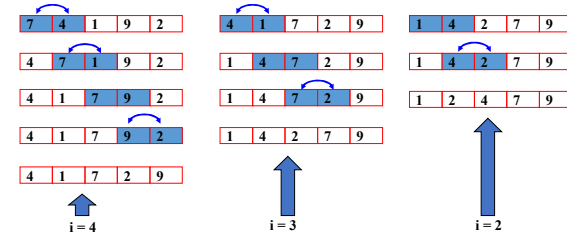
4. BUBBLE SORT

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

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

- Best case: 0 swap, $n^2/2$ compare.
- Worst case: $n^2/2$ swap and compare.
- Average case: $n^2/4$ swap and $n^2/2$ compare.

4. BUBBLE SORT



Note:

- Elements are indexed from 0.

▪ $n=5$

SUMMARY OF THREE BASIC SORTING ALGORITHMS

Number of compare operations:	Insertion	Bubble	Selection
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Number of swap operations:	Insertion	Bubble	Selection
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Chapter 7 – Sorting

Lesson 2. Merge sort

ONE LOVE. ONE FUTURE.

CONTENTS

1. Merge sort

1.1. Algorithm diagram

1.2. Computation time

1.3. Example

2. Implementation

1. MERGE SORT

1.1. Algorithm diagram

Divide

- Divide the sequence of n elements to be sorted into 2 sequences, each sequence has $n/2$ elements

Conquer

- Sort each subsequence recursively using **merge sort**
- When the sequence has only one element left, return this element

Combine

- Merge two sorted subsequences to obtain a sorted sequence containing all the elements of both subsequences

1. MERGE SORT

1.1. Algorithm diagram

MERGE-SORT(A, p, r)

if $p < r$

▷ Check the condition

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

Divide (chia)

MERGE-SORT(A, p, q)

▷ Conquer (tri)

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

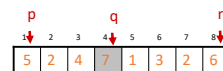
▷ Conquer (tri)

MERGE(A, p, q, r)

▷ Combine (tổ hợp)

endif

- The statement to execute the algorithm: MERGE-SORT(A, 1, n)

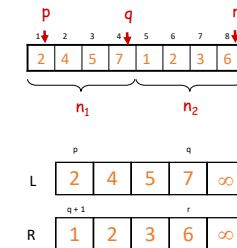


1. MERGE SORT

1.1. Algorithm diagram

MERGE(A, p, q, r)

- Calculate n_1 and n_2
- Copy n_1 first elements to $L[1 \dots n_1]$ and n_2 next elements to $R[1 \dots n_2]$
- $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
- $i \leftarrow 1$; $j \leftarrow 1$
- for $k \leftarrow p$ to r do
- if $L[i] \leq R[j]$
- then $A[k] \leftarrow L[i]$
- $i \leftarrow i + 1$
- else $A[k] \leftarrow R[j]$
- $j \leftarrow j + 1$



1. MERGE SORT

1.2. Computation time

- Init (create 2 subarrays L và R):
 - $\Theta(n_1 + n_2) = \Theta(n)$
- Populate the elements into the resulting array (final for loop):
 - n iterations, each iteration requires constant time $\Rightarrow \Theta(n)$
- The total computation time of merge sort:
 - $\Theta(n)$

1. MERGE SORT

1.2. Computation time

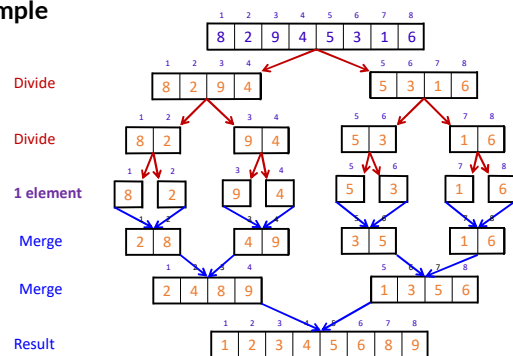
- **Divide:**
 - Calculate q as the average value of p and r: $D(n) = \Theta(1)$
- **Conquer:**
 - recursively solve 2 subproblems, each problem of size $n/2 \Rightarrow 2T(n/2)$
- **Combine:**
 - MERGE on subarrays of about n elements: requires $\Theta(n) \Rightarrow C(n) = \Theta(n)$

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

Master theorem: $T(n) = \Theta(n \log n)$

1. MERGE SORT

1.3. Example



CONTENTS

1. Merge sort

2. Implementation

2. IMPLEMENTATION

- Implement merge: Merge A[first..mid] and A[mid+1.. last]

```
void merge(DataType A[], int first, int mid, int last){   DataType tempA[MAX_SIZE];  
    // mảng phụ  
    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]; // copy the remaining of subarray 1  
    for (; first2 <= last2; ++first2, ++index)  
        tempA[index] = A[first2]; // copy the remaining of subarray 1  
    for (index = first; index <= last; ++index)  
        A[index] = tempA[index]; // copy the result array to A  
    // end merge
```

2. IMPLEMENTATION

```
void mergesort(DataType A[], int first, int last)  
{  
    if (first < last)  
    {  
        // divide into 2 subarrays  
        int mid = (first + last)/2; // index of the middle  
        // sort the left subarray A[first..mid]  
        mergesort(A, first, mid);  
        // sort the right subarray A[mid+1..last]  
        mergesort(A, mid+1, last);  
        // Merge 2 subarrays  
        merge(A, first, mid, last);  
    } // end if  
} // end mergesort
```



Chapter 7 – Sorting

Lesson 4. Quick sort

CONTENTS

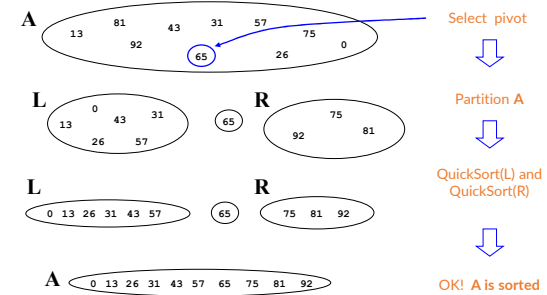
1. General diagram
2. Partition
3. Computation time of quick sort

1. GENERAL DIAGRAM

1. **Condition for base case:** If the sequence has no more than 1 element left, it is a sorted sequence and immediately return this sequence without having to do anything.
2. **Divide:**
 - Select an element in the sequence and call it the pivot element p .
 - Divide the given sequence into 2 subsequences:
 - The left subsequence (L) includes elements \leq key elements,
 - The right subsequence (R) includes elements $>$ key elements.
 - This operation is called "Partition".
3. **Conquer:** Repeat the algorithm recursively for two subsequences L and R
4. **Combine:** The order sequence is $L p R$.

41

1. GENERAL DIAGRAM



42

1. GENERAL DIAGRAM

➤ General diagram of Quick Sort

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); }

- Procedure Partition(A, Left, Right) divides A[Left..Right] into 2 subarrays A[Left..Pivot - 1] and A[Pivot + 1..Right] such that:
 - Elements of A[Left..Pivot - 1] \leq A[Pivot]
 - Elements of A[Pivot + 1..Right] \geq A[Pivot]
- Statement to execute the algorithm **Quick-Sort(A, 1, n)**

43

1. GENERAL DIAGRAM

- When the array only has a small number of elements left (eg: 9 elements)
- Use simple algorithms to sort this sequence, rather than continuing to divide it.
- Algorithm in this situation:

Quick-Sort(A, Left, Right)

1. **if** (Right - Left < n_0)
2. Insertion_sort(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); }

44

CONTENTS

1. General diagram

2. Partition

2.1. Select the pivot

2.2. The pivot is the first element

2.3. The pivot is the middle element

2.4. The pivot is the last element

3. Complexity of quick sort



45

2. PARTITION

- In QuickSort, divide operation consists of 2 subwork:
 - Select the pivot **p**.
 - Partition: divide the given array into 2 subarray
- The partition operation can be implemented (in place) with time $\Theta(n)$.
- The effectiveness of the algorithm depends greatly on which element is chosen as the pivot:
 - The calculation time in the worst case scenario of QS is $O(n^2)$.
 - Occurs when the array is sorted and the selected pivot is the leftmost element of the array.
 - If the pivot is chosen randomly -> QS has computational complexity of $O(n \log n)$.



46

2. PARTITION

- 2.1. Select the pivot
 - Plays a decisive role in the effectiveness of the algorithm.
 - Best: middle element in sorted list (median/median)
 - After $\log_2 n$ times the partition will reach an array of size 1.
 - However, it is difficult to do.



47

2. PARTITION

- 2.1. Select the pivot
 - Commonly used these ways to select elements:
 - Leftmost (first element).
 - Rightmost (last element).
 - Middle of the array.
 - **Median of the 3 elements, middle and last.**
 - An element randomly.



48

2. PARTITION

- 2.1. Select the pivot
 - Build the procedure **Partition(a, left, right)** do the following:
 - Input:** Array $a[\text{left} \dots \text{right}]$.
 - Output:** Redistribute the elements of the input array and return the j pivot index that satisfies:
 - $a[j_{\text{pivot}}]$ contains the original value of $a[\text{left}]$,
 - $a[i] \leq a[j_{\text{pivot}}]$, for all $\text{left} \leq i < \text{pivot}$,
 - $a[j] \geq a[j_{\text{pivot}}]$, for all $\text{pivot} < j \leq \text{right}$.



49

2. PARTITION

- 2.2. The pivot is the first element of the array

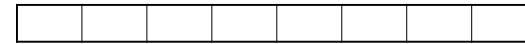
```

Partition(a, left, right)
  i = left; j = right + 1; pivot = a[left];
  while i < j do
    i = i + 1;
    while i ≤ right and a[i] < pivot do i = i + 1;
    j = j - 1;
    while j ≥ left and a[j] > pivot do j = j - 1;
    swap(a[i], a[j]);
  swap(a[i], a[j]); swap(a[j], a[left]);
  return j;

```

Annotations:

- $\text{pivot} = a[\text{left}]$: pivot is the first element
- j is the index (j_{pivot}) need to return, so swap $a[\text{left}]$ and $a[j]$



After selecting pivot, shift pointers i and j from the beginning and end of the array and swap pairs of elements that satisfy:
 $a[i] > \text{pivot}$ and $a[j] < \text{pivot}$



50

2. PARTITION

2.2. Pivot is the first element

Example 1

Position:	0	1	2	3	4	5	6	7	8	9
Key (khóa):	9	1	11	17	13	18	4	12	14	5
Iter 1×while:	9	1	5	17	13	18	4	12	14	11
Iter 2×while:	9	1	5	4	13	18	17	12	14	11
Iter 3×while:	9	1	5	13	4	18	17	12	14	11
2 swap oper.:	4	1	5	9	13	18	17	12	14	11



51

2. PARTITION

2.2. Pivot is the first element

Example 1

Select pivot:	7	2	8	3	5	9	6
Partition: Pointer	7	2	8	3	5	9	6
$2 < \text{pivot}$	7	2	8	3	5	9	6
swap 6, 8	7	2	6	3	5	9	8
$3, 5 < 9$	7	2	6	3	5	9	8
End of partition	7	2	6	3	5	9	8
Put the pivot into position	5	2	6	3	7	9	8



52

2. PARTITION

• 2.3. Pivot is the middle element

```
PartitionMid(a, left, right);  
i = left; j = right; pivot = a[(left + right)/2];  
repeat  
    while a[i] < pivot do i = i + 1;  
    while pivot < a[j] do j = j - 1;  
    if i <= j  
        swap(a[i], a[j]);  
        i = i + 1; j = j - 1;  
until i > j;  
return j;
```

← pivot is selected as the middle element

53

2. PARTITION

2.4. Pivot is the last element

```
int PartitionR(int a[], int p, int r) {  
    int x = a[r];  
    int j = p - 1;  
    for (int i = p; i < r; i++) {  
        if (x >= a[i])  
            { j = j + 1; swap(a[i], a[j]); }  
    }  
    a[r] = a[j + 1]; a[j + 1] = x;  
    return (j + 1);  
}  
  
void quickSort(int a[], int p, int r) {  
    if (p < r) {  
        int q = PartitionR(a, p, r);  
        quickSort(a, p, q - 1);  
        quickSort(a, q + 1, r);  
    }  
}
```

54

2. PARTITION

▪ Implement QUICK SORT

```
int Partition(int a[], int L, int R)  
{  
    int i, j, p;  
    i = L; j = R + 1; p = a[L];  
    while (i < j) {  
        i = i + 1;  
        while ((i <= R) && (a[i] < p)) i++;  
        j--;  
        while ((j >= L) && (a[j] > p)) j--;  
        swap(a[i], a[j]);  
    }  
    swap(a[i], a[j]); swap(a[j], a[L]);  
    return j;  
}
```

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

```
void quick_sort(int a[], int left, int right)  
{  
    int p;  
    if (left < right)  
    {  
        p = Partition(a, left, right);  
        if (left < p)  
            quick_sort(a, left, p - 1);  
        if (right > p)  
            quick_sort(a, p + 1, right);  
    }  
}
```

55

CONTENTS

1. General diagram
2. Partition
3. Complexity of quick sort
 - 3.1. Unbalanced partition
 - 3.2. Perfect partition
 - 3.3. Balanced partition

56

3. COMPLEXITY OF QUICK SORT

- Depends on partition is **balanced or unbalanced** → depends on how to select the pivot.

- Unbalanced partition:** one subproblem has size $n - 1$ and the other has size 0.
- Perfect partition:** partition is always done in dichotomous form → each subproblem has size $n/2$.
- Balanced segmentation:** partition is done somewhere around the midpoint → one subproblem has size $n - k$ and the other has size k .



57

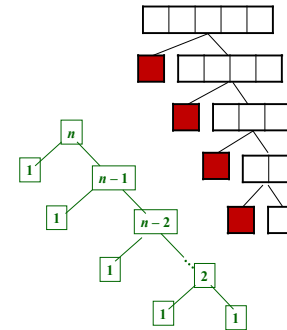
3. COMPLEXITY OF QUICK SORT

3.1. Unbalanced partition

Recursive formular:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

$$\left. \begin{aligned} T(0) &= T(1) = 1 \\ T(n) &= T(n-1) + n \\ T(n-1) &= T(n-2) + (n-1) \\ T(n-2) &= T(n-3) + (n-2) \\ &\dots \\ T(2) &= T(1) + (2) \\ T(n) &= T(1) + \sum_{i=2}^n i \\ &= O(n^2) \end{aligned} \right\}$$

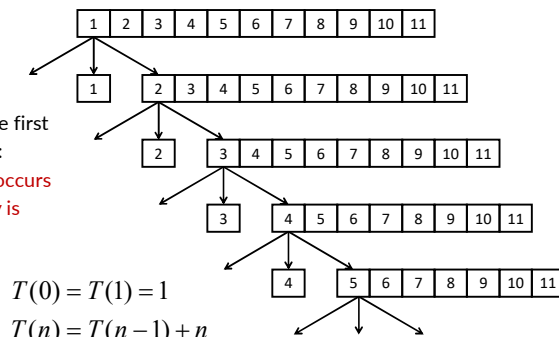


58

3. COMPLEXITY OF QUICK SORT

3.1. Unbalanced partition

- When the pivot is the first element of the array:
The worst situation occurs when the input array is already sorted



$$T(0) = T(1) = 1$$

$$T(n) = T(n-1) + n$$



59

3. COMPLEXITY OF QUICK SORT

3.2. Perfect partition

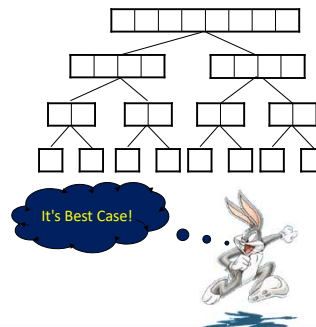
Recursive formula:

$$T(n) = T(n/2) + T(n/2) + \Theta(n)$$

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = \Theta(n \log n)$$

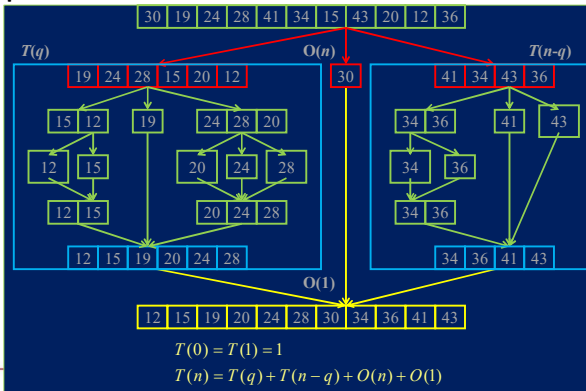
Master theorem !



60

3. COMPLEXITY OF QUICK SORT

3.3. Balance partition



61

3. COMPLEXITY OF QUICK SORT

3.3. Balance partition

- Assume that the pivot is chosen randomly from among the elements of the input array
- All of the following situations are equally likely:
 - Pivot is the smallest element of the array
 - Pivot is the second smallest element of the array
 - Pivot is the third smallest element of the array
 - ...
 - Pivot is the largest element of the array
 - Pivot is the first element of the array

62

Chapter 7 – Sorting

Lesson 5. Heap sort

ONE LOVE. ONE FUTURE.

63

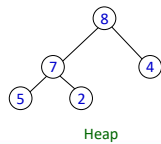
CONTENTS

- Heap data structure
- Heap sort

64

1. HEAP DATA STRUCTURE

- **Definition:** **Heap** is a complete binary tree with 2 properties:
 - **Structure:** all levels are filled, except the last level, which is filled from left to right.
 - **Ordered or heap property:** for each node x
 $\text{Parent}(x) \geq x$.
- The tree is implemented by array $A[i]$ of length $\text{length}[A]$. The number of elements is $\text{heapsize}[A]$



Từ tính chất đồng suy ra:
"Gốc chứa phần tử lớn nhất của đống!"

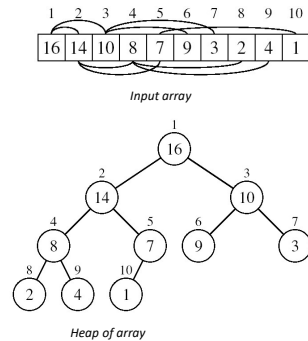
Như vậy có thể nói:
Đống là cây nhị phân được điền theo thứ tự

HUST

65

1. HEAP DATA STRUCTURE

- Represent heap by array
- The heap could be stored in array A .
 - Root of tree is $A[1]$
 - Left child of $A[i]$ is $A[2*i]$
 - Right child of $A[i]$ is $A[2*i + 1]$
 - Parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$
 - $\text{Heapsize}[A] \leq \text{length}[A]$
- Elements of $A[\lfloor n/2 \rfloor + 1] \dots n$ are leaves

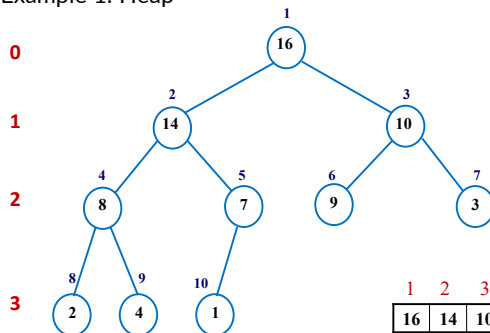


HUST

66

1. HEAP DATA STRUCTURE

- Example 1: Heap



$\text{parent}(i) = \lfloor i/2 \rfloor$
 $\text{left-child}(i) = 2i$
 $\text{right-child}(i) = 2i + 1$

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

HUST

67

1. HEAP DATA STRUCTURE

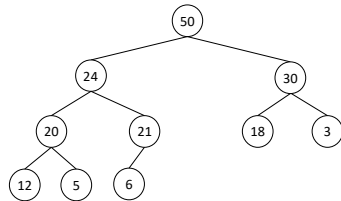
- Two type of heap
- **Max-heaps** (largest element at the root), with *max-heap* property:
 - For every node i , except the root:
 $A[\text{parent}(i)] \geq A[i]$
- **Min-heaps** (smallest element at the root), with *min-heap* property:
 - For every node i , except the root:
 $A[\text{parent}(i)] \leq A[i]$
- In the following we will only consider the max-heap. The min pile is considered completely similar.

HUST

68

1. HEAP DATA STRUCTURE

- Two types of heap
 - New nodes added to the bottom level (from left to right)
 - Nodes are removed from the bottom level (from right to left)



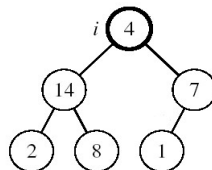
Example max heap

1. HEAP DATA STRUCTURE

- Operations on heap
 - Recover the max-heap
 - Max-Heapify
 - Create max-heap from unsorted array
 - Build-Max-Heap

1. HEAP DATA STRUCTURE

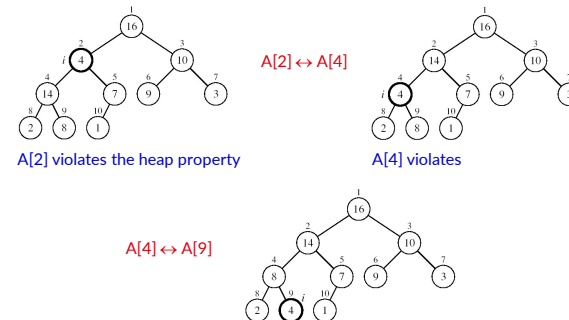
- Recover the heap property
 - Suppose there is a node i with a value smaller than its child
 - The assumption is: The left subtree and right subtree of i are both max-heaps
 - To eliminate this violation, proceed as follows:
 - Switch places with the larger child
 - Move down along the tree
 - Continue the process until the node is no longer smaller than the child



Node i violates the heap property

1. HEAP DATA STRUCTURE

- Example 2

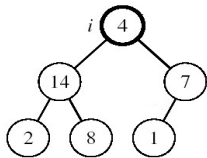


1. HEAP DATA STRUCTURE

➤ The algorithm to recover the heap property:

▪ Assume:

- Both left and right subtrees of node i are max-heaps
- $A[i]$ could be smaller than its children



```

Max-Heapify( $A, i, n$ )
//  $n = \text{heapsize}[A]$ 
1.  $l \leftarrow \text{left-child}(i)$ 
2.  $r \leftarrow \text{right-child}(i)$ 
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  $\text{Exchange}(A[i], A[\text{largest}])$ 
10. Max-Heapify( $A, \text{largest}, n$ )
    
```

1. HEAP DATA STRUCTURE

➤ Computation time of MAX-HEAPIFY

▪ We can see that:

- From node i , one must move down the tree.
- The length of this path does not exceed the length of the path from the root to the leaf, that is, it does not exceed h .
- At each level, two comparisons must be made.
- Therefore, the total number of comparisons does not exceed $2h$.
- So, the calculation time is $O(h)$ or $O(\log n)$.

▪ Conclusion: Calculation time of MAX-HEAPIFY is $O(\log n)$

▪ If written in heap height language, this time is $O(h)$

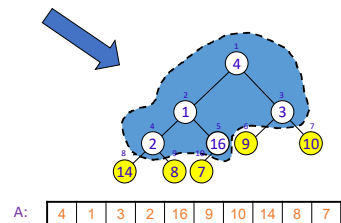
1. HEAP DATA STRUCTURE

➤ Computation time of MAX-HEAPIFY

- Change the array $A[1 \dots n]$ to max-heap ($n = \text{length}[A]$)
- Because elements of subarray $A[\lfloor n/2 \rfloor + 1 \dots n]$ are leaves
- So to create heap, we only apply MAX-HEAPIFY on elements from 1 to $\lfloor n/2 \rfloor$

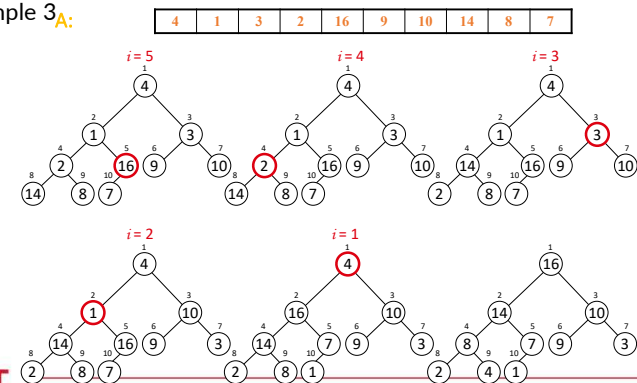
Alg: Build-Max-Heap(A)

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



1. HEAP DATA STRUCTURE

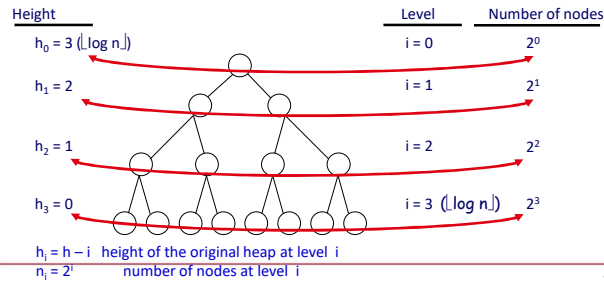
▪ Example 3:



1. HEAP DATA STRUCTURE

➤ Computation time of Build-Max-Heap

- Heapify requires $O(h) \Rightarrow$ cost of Heapify at node i is proportional to the height of node i in the tree
- $$\Rightarrow T(n) = \sum_{i=0}^h n_i h_i = \sum_{i=0}^h 2^i (h-i) = O(n)$$



CONTENTS

1. Heap data structure

2. Heap sort

2. HEAP SORT

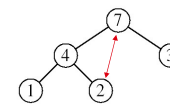
➤ Algorithm diagram:

- Create a max-heap from the given array
- Swap the root (largest element) with the last element in the array
- Remove the last node by reducing the size of the heap by 1
- Perform **Max-Heapify** on the new root
- Repeat the process until the pile has only 1 node left

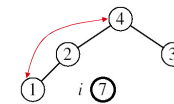
2. HEAP SORT

▪ Example 4

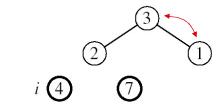
$A = [7, 4, 3, 1, 2]$



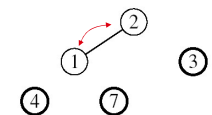
Max-Heapify($A, 1, 4$)



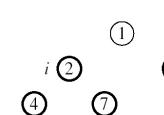
Max-Heapify($A, 1, 3$)



Max-Heapify($A, 1, 2$)



Max-Heapify($A, 1, 1$)



2. HEAP SORT

• *Algorithm:* HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. Max-Heapify(A, 1, $i - 1$)

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

- Computation time: $O(n \log n)$

