



**WOLLO UNIVERSITY**  
**KOMBOLCHA INSTITUTES OF TECHNOLOGY**  
College of Informatics

## **Analysis of Algorithms**

---

### **Chapter 1**

### **Introduction and Elementary Data Structures**

Belachew N.  
nbelay2112@gmail.com

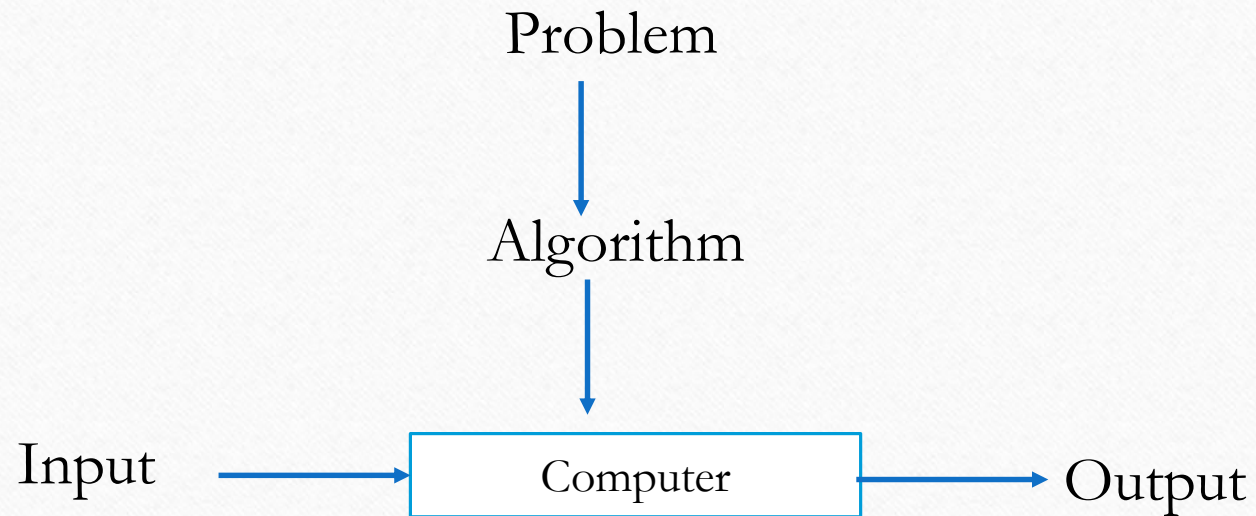
# Outline

- ✓ Introduction to Algorithms
- ✓ Analysis of algorithm
- ✓ Review of elementary data structures
- ✓ Stacks and Queues
- ✓ Priority Queues
- ✓ Hashing



# What is an algorithm?

- ✓ An algorithm is a finite set of well-defined instructions to be followed for solving a problem.



# Characteristics of an Algorithm

- ✓ All algorithms must satisfy the following criteria:
  - **Input:** Zero or more quantities are externally supplied.
  - **Output:** At least one quantity is produced.
  - **Definiteness:** Each instruction is clear and unambiguous.
  - **Finiteness:** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
  - **Effectiveness:** Steps are sufficiently simple and basic

# Study Of an Algorithm

- ✓ How to devise algorithm?
  - It's an ART.
- ✓ How to Validate algorithms?
  - Assure algorithm is working correctly.
- ✓ How to analyze algorithm?
  - Determining how much computing time and storage an algorithm require.
- ✓ How to test algorithm?
  - Testing the performance of algorithm.



# Algorithm Analysis

- ✓ Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.
- ✓ An algorithm is mainly analyzed to determine the execution time of the program and how much memory space required.
- ✓ Analysis of algorithm is important for the following reasons
  - Analysis is more reliable than the experimental testing
  - Analysis helps to select better algorithm
  - Analysis predicts performance
  - Analysis identifies the scope of improvement of algorithm

## ...cont'd

- ✓ The efficiency of the algorithm can be decided by measuring the performance of an algorithm.
- ✓ we can measure the performance of an algorithm by computing two factors:
  1. Amount of time required by an algorithm to execute.
  2. Amount of storage required by an algorithm.

# Space Complexity

- ✓ The space complexity of an algorithm is the amount of memory it needs to run to completion.
- ✓ To compute the space complexity we use two factors: constant and instance characteristics.
- ✓  $S(P) = c + SP$

Where  $SP$  is the instance characteristics and  $c$  is a constant.

- ✓ Space complexity must be taken seriously for **multiuser systems** and in situations where **limited memory is available**.



# Calculating the Space Complexity

- ✓ For calculating the space complexity, we need to know the value of memory used by different type of datatype variables.

```
// n is the length of array a[]
int sum(int a[], int n) {
    int x = 0;
    for(int i = 0; i < n; i++)
    {
        x = x + a[i];
    }
    return(x);
}
```

total memory requirement will be  $(4n + 12)$

# Time Complexity

- ✓ The time complexity of an algorithm is the amount of computer time it needs to run to completion.
- ✓ The time taken by a program is the sum of the **compile time** and **run time**.
- ✓ The compile time does not depend on the instance characteristics.

## ...cont'd

- ✓ The run time may depend on:
  - Specific Processor speed
  - Type of compiler used
  - Current Processor Load
  - Specific data for a particular run of the program (Logical size of the program)
    - Input Size
    - Input Properties
  - Available memory.
  - Operating Environment



# Analysis Rules

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1:
  - Assignment Operation
  - Single Input/Output Operation
  - Single Boolean Operations
  - Single Arithmetic Operations
  - Function Return

## ...cont'd

3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loops: Running time for a loop is equal to the running time for the statements inside the loop \* number of iterations.
  - The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops.
  - For nested loops, analyze inside out.
- ✓ Always assume that the loop executes the maximum number of iterations possible.

## ...cont'd

5. Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

### Example1:

```
int count(){  
    int k=0;  
    cout<< "Enter an integer";  
    cin>>n;  
    for (i=0;i<n;i++)  
        k=k+1;  
    return 0;  
}
```

1 for the assignment statement: `int k=0`

1 for the output statement.

1 for the input statement.

In the for loop:

1 assignment,  $n+1$  tests, and  $n$  increments.

$n$  loops of 2 units for an assignment, and an addition.

1 for the return statement.



## ...cont'd

### Example2:

```
void func()
```

```
{
```

```
int x=0;
```

```
int i=0;
```

```
int j=1;
```

```
cout<< "Enter an Integer value";
```

```
cin>>n;
```

```
while (i<n){
```

```
    x++;
```

```
    i++;
```

```
}
```

```
while (j<n)
```

```
{
```

```
    j++;
```

```
}
```

```
}
```

$$T(n) = 1 + 1 + 1 + 1 + 1 + n + 1 + 2n + n + n - 1 = 5n + 5$$

# Asymptotic Analysis

- ✓ **Asymptotic analysis** is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.
- ✓ Generally, the complexity is investigated in three cases.
  1. **Best Case** – Minimum time required for program execution.
  2. **Average Case** – Average time required for program execution.
  3. **Worst Case** – Maximum time required for program execution.

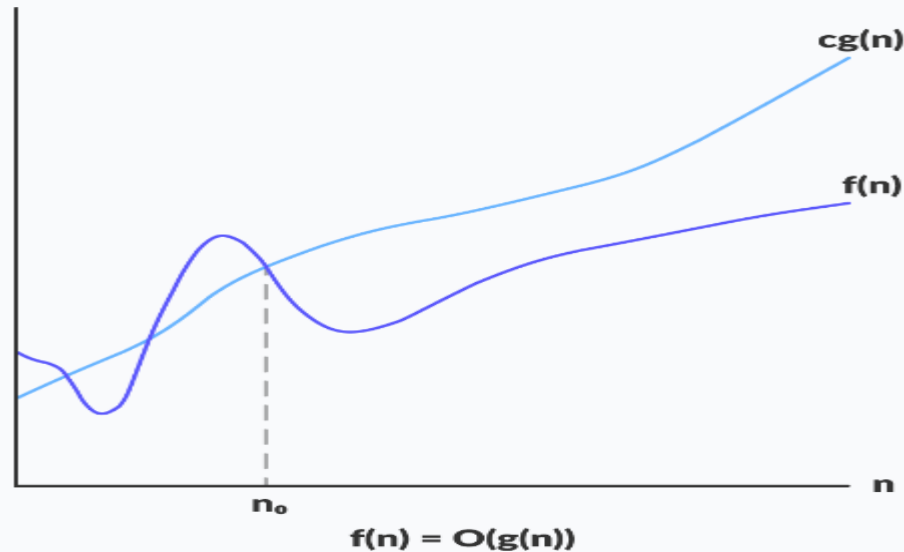
# Asymptotic Notations

- ✓ Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.
  - Big-Oh Notation ( $O$ )
  - Big-Omega Notation ( $\Omega$ )
  - Theta Notation ( $\Theta$ )



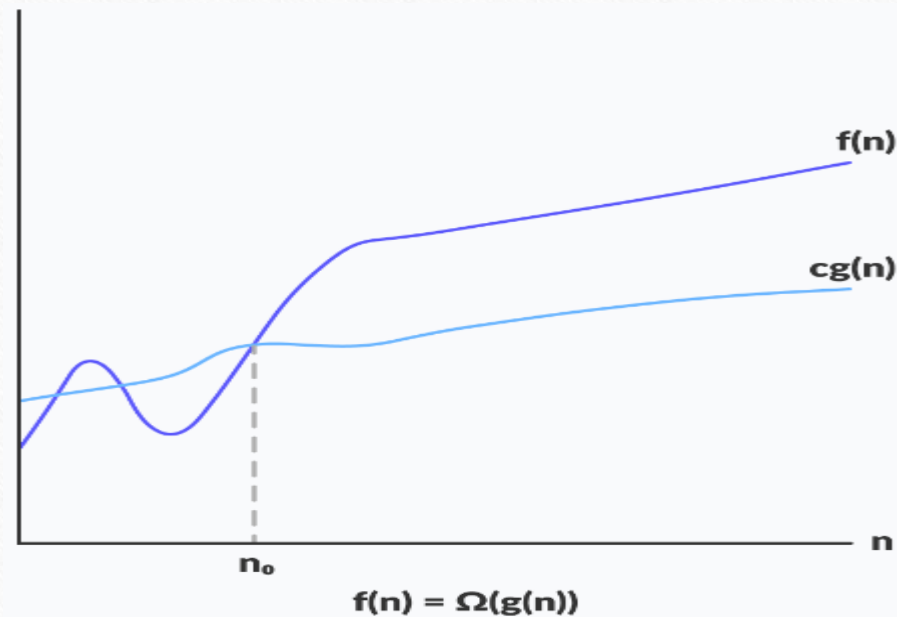
# Big-Oh notation(O)

- ✓ The notation  $O(n)$  is the formal way to express the upper bound of an algorithm's running time.
- ✓ It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



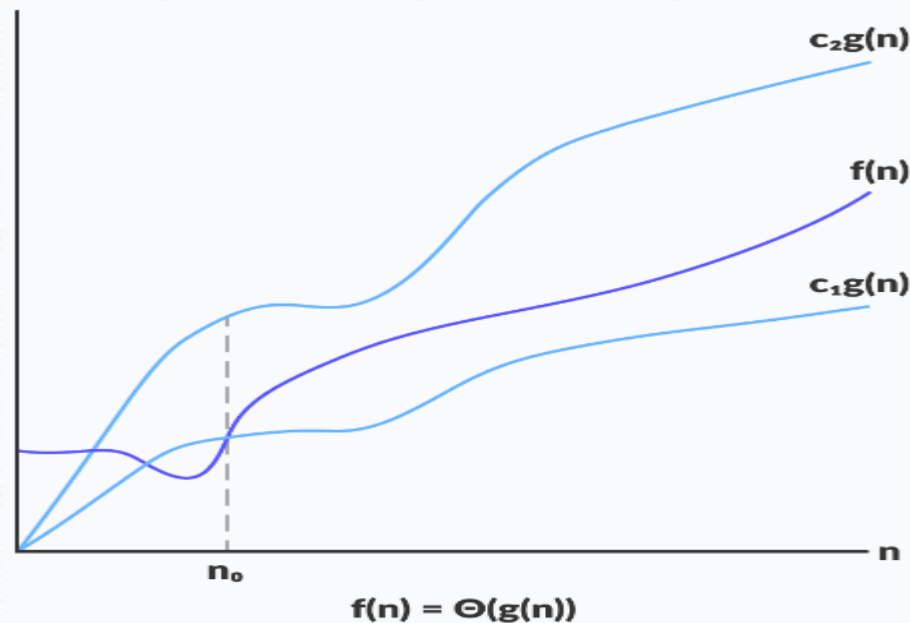
# Omega Notation( $\Omega$ )

- ✓ The notation  $\Omega(n)$  is the formal way to express the lower bound of an algorithm's running time.
- ✓ It measures the best case time complexity an algorithm can possibly take to complete.



# Theta Notation( $\Theta$ )

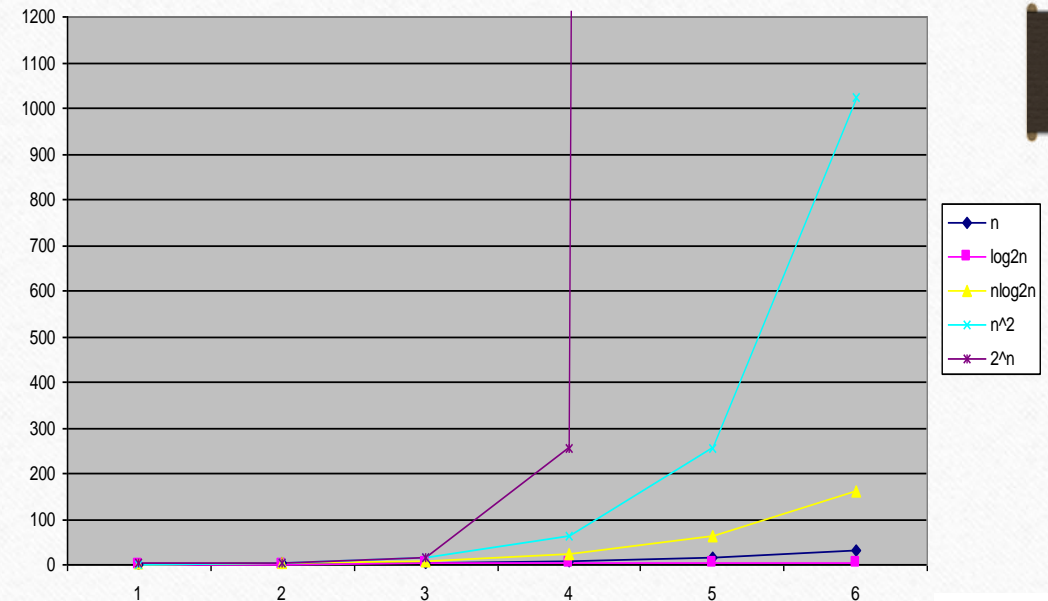
- ✓ Theta notation encloses the function from above and below.
- ✓ Since, it represents the upper and the lower bound of the running time of an algorithm.
- ✓ It is used for analyzing the average case complexity of an algorithm.





# Order of Growth

- ✓ An order of growth is a set of functions whose asymptotic growth behavior is considered equivalent.
- ✓ For example,  $2n$ ,  $100n$  and  $n + 1$  belong to the same order of growth, which is written  $O(n)$  in Big-Oh notation and often called linear because every function in the set grows linearly with  $n$ .
  - $g(n) = 1$  (growth is constant)
  - $g(n) = \log_2 n$  (growth is logarithmic)
  - $g(n) = n$  (growth is linear)
  - $g(n) = n \log_2 n$  (growth is faster than linear)
  - $g(n) = n^2$  (growth is quadratic)
  - $g(n) = 2^n$  (growth is exponential)



# Properties of Order of Growth

- ✓ If  $f_1(n)$  is order of  $g_1(n)$  and  $f_2(n)$  is order of  $g_2(n)$ , then

$$f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n))).$$

- ✓ Polynomials of degree  $m \in \Theta(n^m)$  maximum degree is considered from the polynomial.

Eg.  $a_1n^3 + a_2n^2 + a_3n + c$  has the order of growth  $\Theta(n^3)$ .

- ✓ Exponential function  $a^n$  have different orders of growth for different values of  $a$ .

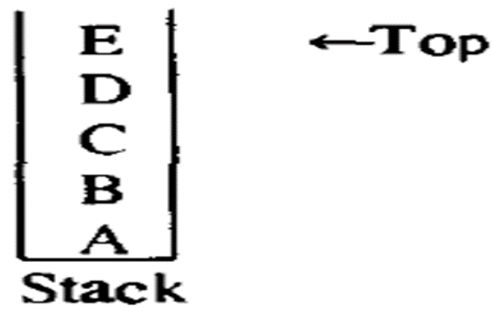
# Review of Elementary Data Structures

- ✓ One of the basic techniques for improving algorithms is to structure the data in such a way that the resulting operations can be efficiently carried out.
- ✓ We should be familiar with stacks and queue, binary trees, and graphs and be able to refer to the other structures as needed.



# Stacks and Queues

- ✓ One of the most common forms of data organization in computer programs is the ordered or linear list, which is often written as  $A = (a_1, a_2, \dots, a_n)$ .
- ✓ A **stack** is an ordered list in which all insertions and deletions are made at one end, called the top.
- ✓ A **queue** is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front.

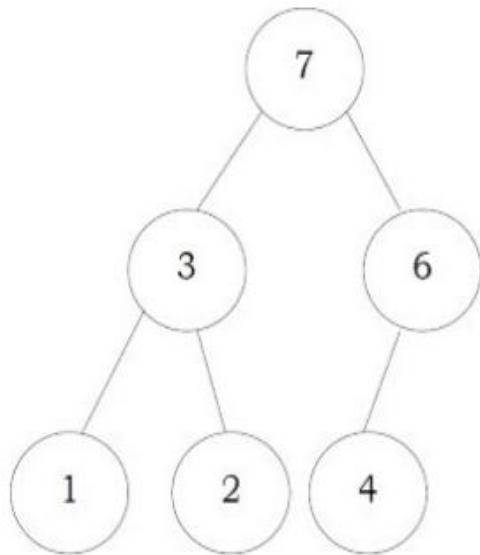


# PRIORITY QUEUES

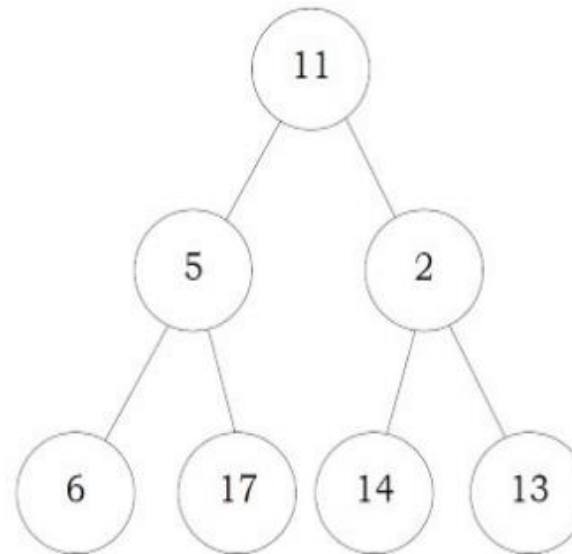
- ✓ Any data structure that supports the operations of search min (or max), insert, and delete min (or max, respectively) is called a priority queue.
- ✓ We might first consider using a queue since inserting new elements would be very efficient.
- ✓ But finding the largest element would necessitate a scan of the entire queue.
- ✓ A second suggestion would be to use a sorted list which is stored sequentially.
- ✓ But an insertion could require moving all of the items in the list.
- ✓ What we want is a data structure which allows both operations to be done efficiently.

# Heap

- ✓ A **heap** is a complete binary tree with some special properties.
- ✓ The basic requirement of a heap is that the value of a node must be  $\geq$  (or  $\leq$ ) than the values of its children.



heap



not a heap



# Heap Representation

- ✓ An efficient representation of the heap is using array.
- ✓ The root is stored at the first place, that is  $a[0]$ .
- ✓ The children of the node  $i$  is at the locations  $((2*i)+1)$  and  $((2*i) + 2)$ .

# ...cont'd

## ✓ Types of Heaps

✓ Based on the property of a heap we can classify heaps into two types:

1. **Max heap:** The value of a node must be greater than or equal to the values of its children
2. **Min heap:** The value of a node must be less than or equal to the values of its children

# Max heap

- ✓ The definition of a max heap implies that one of the largest elements is at the root of the heap.
- ✓ To insert an element into the heap, one adds it \"at the bottom\" of the heap and then compares it with its parent, grandparent, greatgrandparent, and so on, until it is less than or equal to one of these values.



...cont'd

```
Algorithm Insert( $a, n$ )  
{  
    // Inserts  $a[n]$  into the heap which is stored in  $a[1 : n - 1]$ .  
     $i := n$ ;  $item := a[n]$ ;  
    while  $((i > 1) \textbf{ and } (a[\lfloor i/2 \rfloor] < item))$  do  
    {  
         $a[i] := a[\lfloor i/2 \rfloor]$ ;  $i := \lfloor i/2 \rfloor$ ;  
    }  
     $a[i] := item$ ; return true;  
}
```

## ...cont'd

- ✓ To delete the maximum key from the max heap, we use an algorithm called Adjust.
- ✓ Adjust takes as input the array  $a[ ]$  and the integers  $i$  and  $n$ . It regards  $a[1: n]$  as a complete binary tree.
- ✓ If the subtrees rooted at  $2i$  and  $2i+1$  are already max heaps ,then Adjust will rearrange elements of  $a[ ]$  such that the tree rooted at  $i$  is also a max heap.
- ✓ The maximum element from the max heap  $a[1 : n]$  can be deleted by deleting the root of the corresponding complete binary tree.
- ✓ The last element of the array, that is,  $a[n]$ ,is copied to the root, and finally we call  $\text{Adjust}(a,1,n-1)$

## ...cont'd

### Algorithm Adjust( $a, i, n$ )

```
// The complete binary trees with roots  $2i$  and  $2i + 1$  are
// combined with node  $i$  to form a heap rooted at  $i$ . No
// node has an address greater than  $n$  or less than 1.
{
     $j := 2i$ ;  $item := a[i]$ ;
    while ( $j \leq n$ ) do
    {
        if (( $j < n$ ) and ( $a[j] < a[j + 1]$ )) then  $j := j + 1$ ;
            // Compare left and right child
            // and let  $j$  be the larger child.
        if ( $item \geq a[j]$ ) then break;
            // A position for  $item$  is found.
         $a[\lfloor j/2 \rfloor] := a[j]$ ;  $j := 2j$ ;
    }
     $a[\lfloor j/2 \rfloor] := item$ ;
}
```

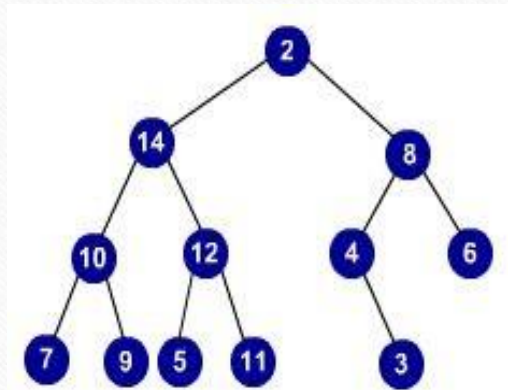
### Algorithm DelMax( $a, n, x$ )

```
// Delete the maximum from the heap  $a[1 : n]$  and store it in  $x$ .
{
    if ( $n = 0$ ) then
    {
        write ("heap is empty"); return false;
    }
     $x := a[1]$ ;  $a[1] := a[n]$ ;
    Adjust( $a, 1, n - 1$ ); return true;
}
```



# Heapification

- ✓ Let consider a binary tree in which left and right subtrees of the root are satisfying the heap property, but not the root. See the following figure.



- ✓ Now question is how to make the above tree into a heap?
- ✓ Swap the root and left child of root, to make the root satisfies the heap property.
- ✓ Then check the subtree rooted at left child of the root is heap or not.

## ...cont'd

- ✓ If it is, we are done. If not, repeat the above action of swapping the root with the maximum of its children.
- ✓ That is, push down the element at root till it satisfies the heap property.

```
Algorithm Heapify( $a, n$ )  
// Readjust the elements in  $a[1 : n]$  to form a heap.  
{  
    for  $i := \lfloor n/2 \rfloor$  to 1 step  $-1$  do Adjust( $a, i, n$ );  
}
```

# Heap Sort

- ✓ Heap sort operates by first converting the list in to a heap tree.
- ✓ Heap tree is a binary tree in which each node has a value greater than both its children (if any).
- ✓ It uses a process called "adjust to accomplish its task (building a heap tree) whenever a value is larger than its parent.
- ✓ The time complexity of heap sort is  $O(n \log n)$ .



# Algorithm

## 1. Construct a binary tree

- ✓ The root node corresponds to `Data[0]`.
- ✓ If we consider the index associated with a particular node to be  $i$ ,
  - then the left child of this node corresponds to the element with index  $2*i+1$  and the right child corresponds to the element with index  $2*i+2$ .
  - If any or both of these elements do not exist in the array, then the corresponding child node does not exist either.

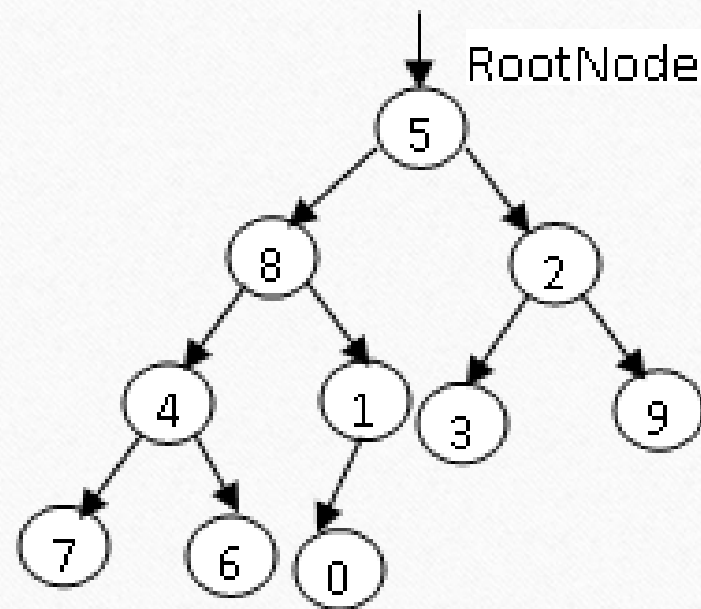
## 2. Construct the heap tree from initial binary tree using "adjust" process.

## 3. Sort by swapping the root value with the lowest, right most value and deleting the lowest, right most value and inserting the deleted value in the array in it proper position.

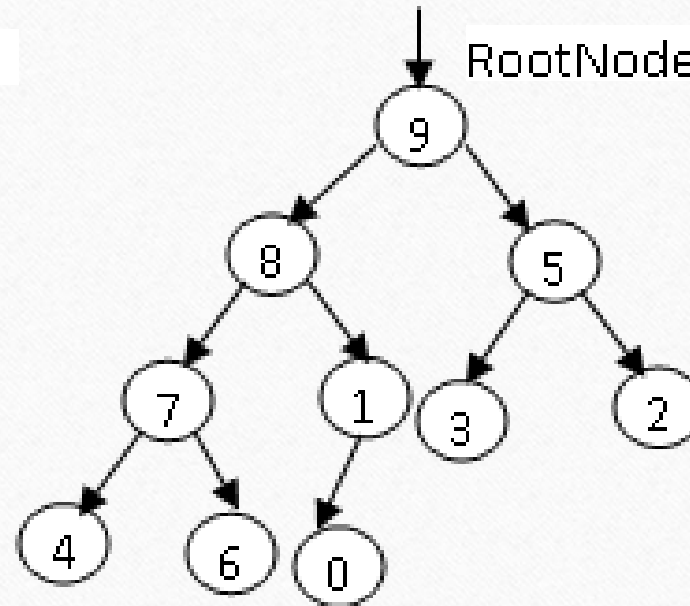
# Example

5	8	2	4	1	3	9	7	6	0
---	---	---	---	---	---	---	---	---	---

Construct the initial binary tree



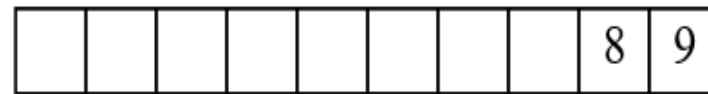
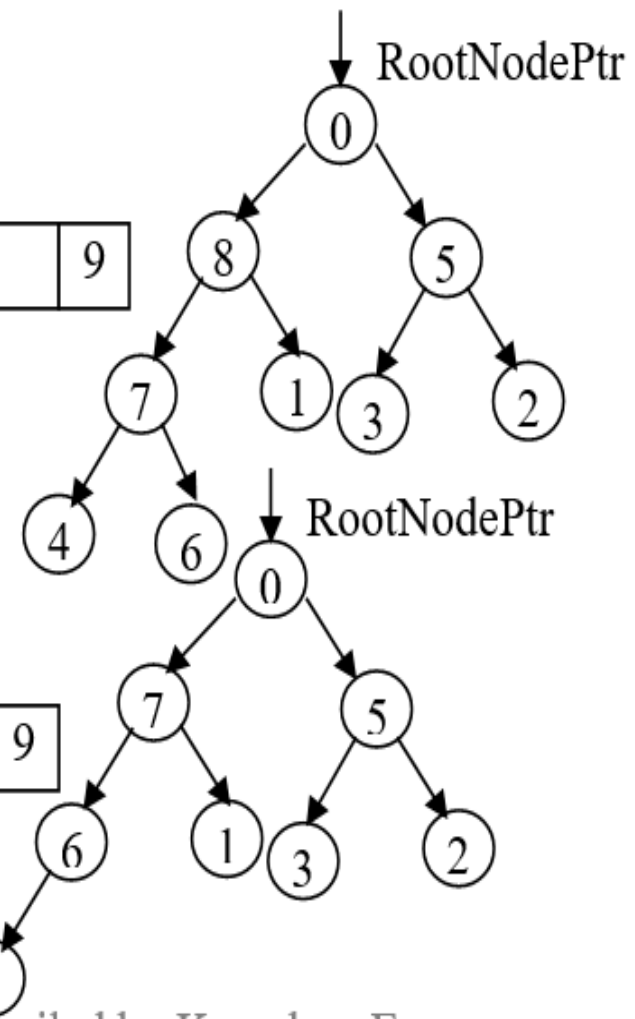
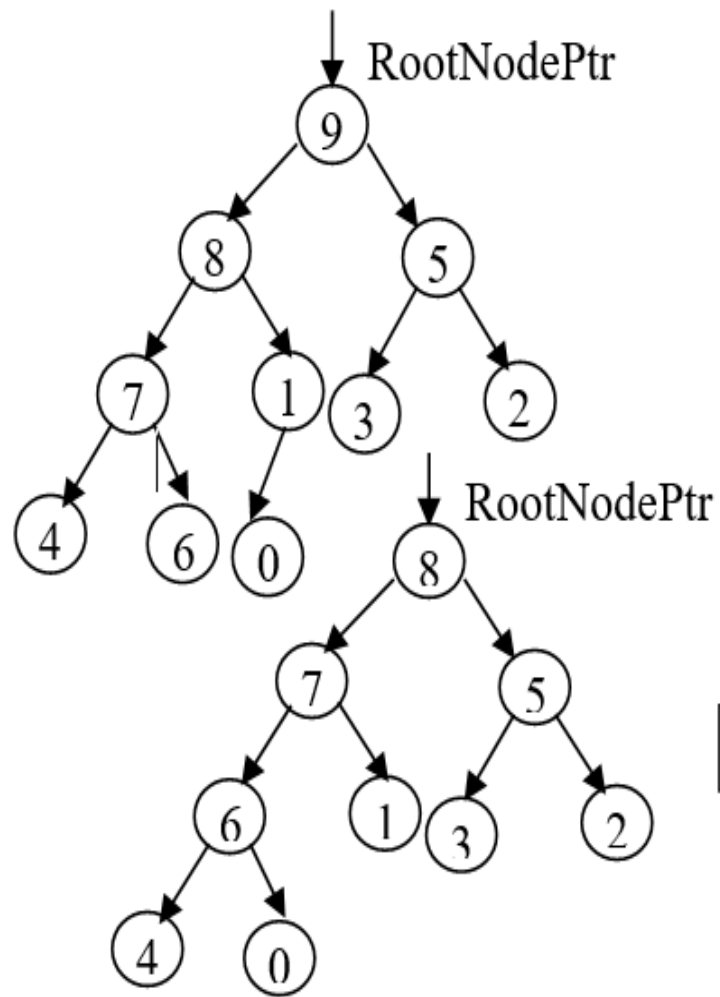
Construct the heap tree

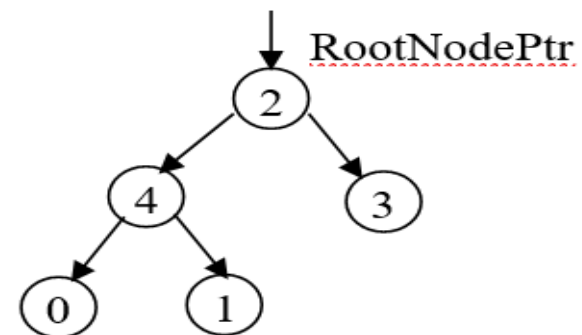
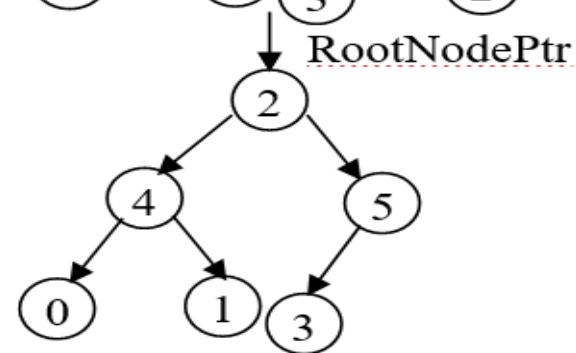
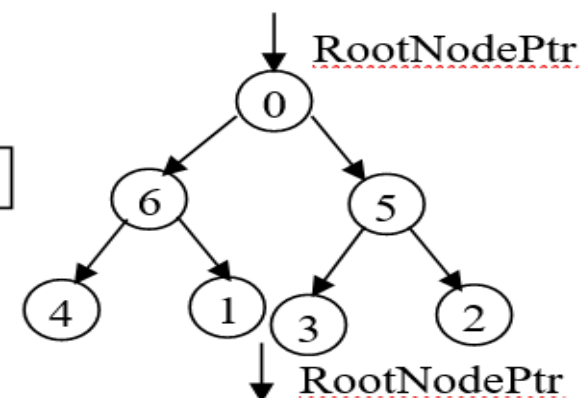
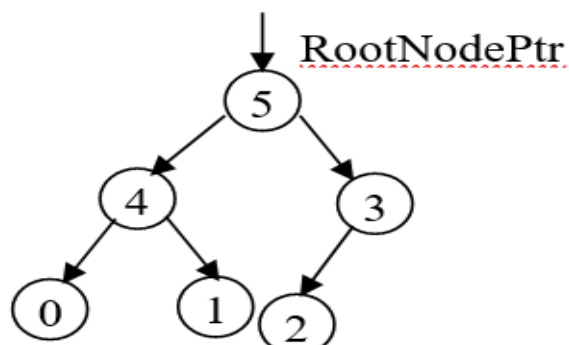
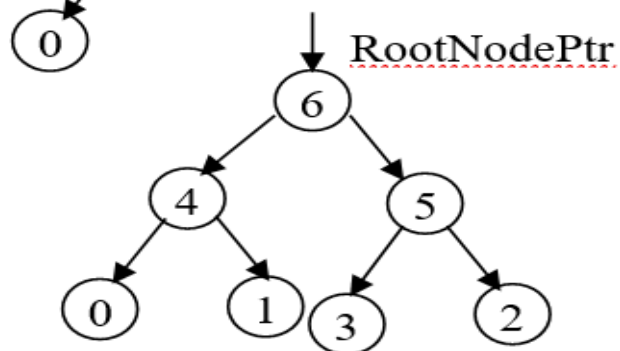
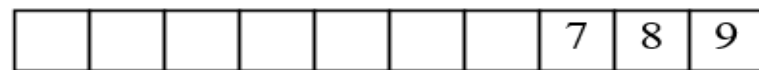
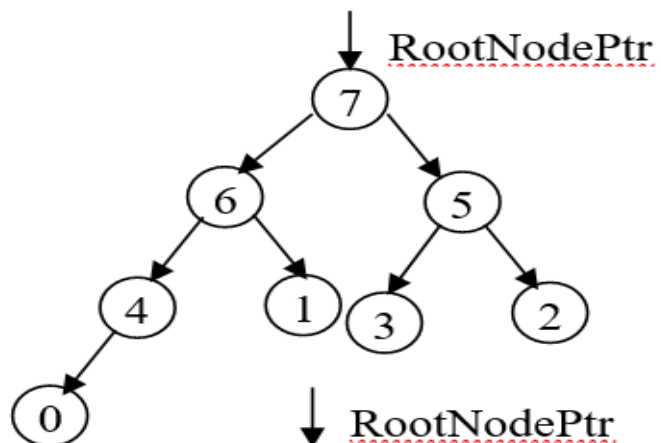


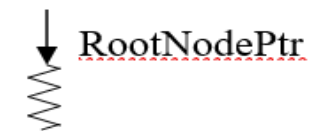
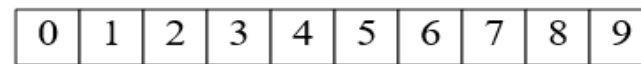
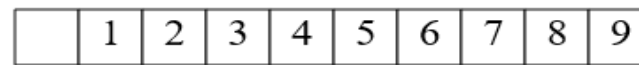
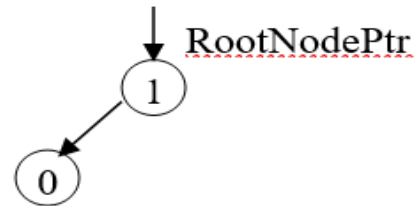
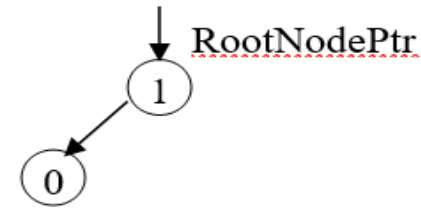
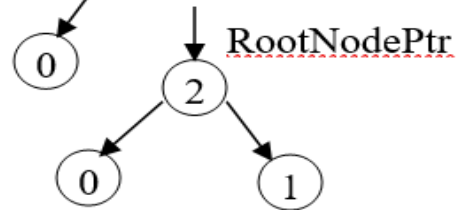
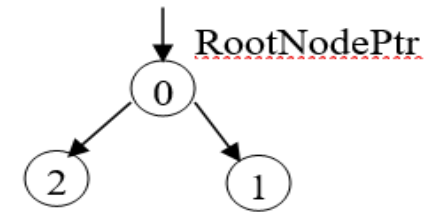
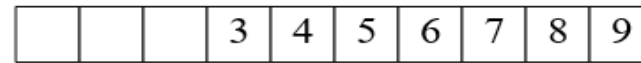
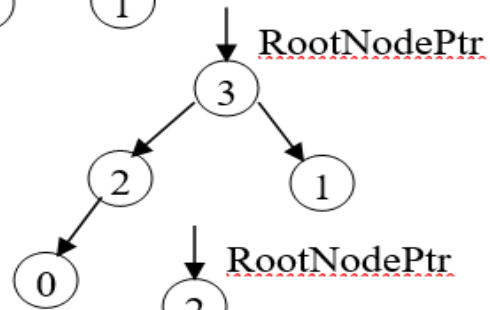
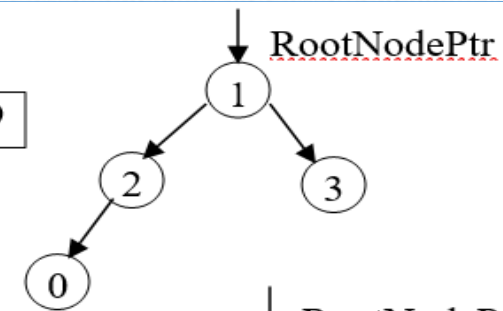
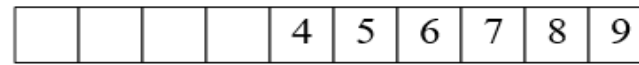
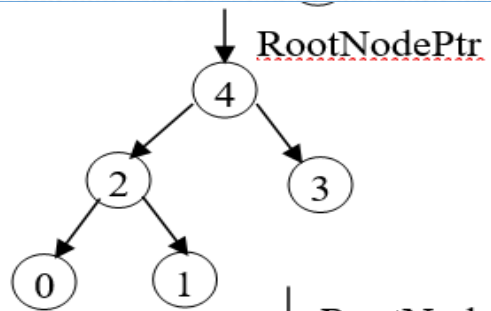
## ...cont'd

- ✓ Swap the root node with the lowest, right most node and delete the lowest, right most value;
- ✓ insert the deleted value in the array in its proper position;
- ✓ adjust the heap tree; and
- ✓ repeat this process until the tree is empty.











# Hashing

- ✓ Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- ✓ Hashing is also known as Hashing Algorithm or Message Digest Function.
- ✓ It is a technique to convert a range of key values into a range of indexes of an array.
- ✓ It is used to facilitate the next level searching method when compared with the linear or binary search.
- ✓ Hashing allows to update and retrieve any data entry in a constant time  $O(1)$ .
- ✓ Constant time  $O(1)$  means the operation does not depend on the size of the data.
- ✓ Hashing is used with a database to enable items to be retrieved more quickly.
- ✓ It is used in the encryption and decryption of digital signatures.

## ...cont'd

- ✓ A hash function generates a signature from a data object.
- ✓ Hash functions have security and data processing applications.
- ✓ A **hash table** is a data structure where the storage location of data is computed from the key using a hash function.
- ✓ A hash table is a collection of items which are stored in such a way as to make it easy to find them later.
- ✓ Each position of the hash table, often called a **slot**, can hold an item and is named by an integer value starting at 0.

## ...cont'd

- ✓ For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty.
- ✓ Figure below shows a hash table of size  $m=11$ .
- ✓ In other words, there are  $m$  slots in the table, named 0 through 10.

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None



## ...cont'd

- ✓ The mapping between an item and the slot where that item belongs in the hash table is called the hash function.
- ✓ The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and  $m-1$ .
- ✓ Given a collection of items, a hash function that maps each item into a unique slot is referred to as a perfect hash function.
- ✓ If we know the items and the collection will never change, then it is possible to construct a perfect hash function.
- ✓ Unfortunately, given an arbitrary collection of items, there is no systematic way to construct a perfect hash function.

## ...cont'd

- ✓ Luckily, we do not need the hash function to be perfect to still gain performance efficiency.
- ✓ One way to always have a perfect hash function is to increase the size of the hash table so that each possible value in the item range can be accommodated.
- ✓ This guarantees that each item will have a unique slot.
- ✓ Although this is practical for small numbers of items, it is not feasible when the number of possible items is large.

## ...cont'd

- ✓ Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :
- ✓ Hash Key = Key Value % Number of Slots in the Table
- ✓ Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	$26 \% 10 = 6$	6
70	$70 \% 10 = 0$	0
18	$18 \% 10 = 8$	8
31	$31 \% 10 = 1$	1
54	$54 \% 10 = 4$	4
93	$93 \% 10 = 3$	3

0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table



## ...cont'd

- ✓ In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by,  $\lambda = \text{No. of items} / \text{table size}$ . For example,  $\lambda = 6/10$ .
- ✓ It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.
- ✓ Constant amount of time  $O(1)$  is required to compute the hash value and index of the hash table at that location.

# Reading Assignment

- ✓ Take the above example, if we insert next item 40 in our collection, it would have a hash value of 0 ( $40 \% 10 = 0$ ).
- ✓ But 70 also had a hash value of 0, it becomes a problem. This problem is called as Collision or Clash. Collision creates a problem for hashing technique.
- ✓ **How to solve collision in hash table?**

# Review Questions

1. What is an algorithm?
2. Describe some important characteristics of algorithms?
3. Why analysis algorithms?
4. List commonly used asymptotic notations and explain with example?
5. Explain the difference between max heap and min heap
6. how to construct heap tree from unsorted array.
7. Write at least three applications of stack and queue.



# Assignment 10%

1. Describe recursive algorithms with example.
2. What is recurrence relations?
3. List three techniques/ways to solve/analysis recurrence algorithms and discuss with example.

# End of Ch.1

---

Questions, Ambiguities, Doubts, ... ???