Unit-1

what is algorithm:

An algorithm is defined as a finite set of instructions that, if followed, performs a particular task. All algorithms must satisfy the following criteria

Input. An algorithm has zero or more inputs, taken or collected from a specified set of objects.

Output. An algorithm has one or more outputs having a specific relation to the inputs.

Definiteness. Each step must be clearly defined; Each instruction must be clear and unambiguous.

Finiteness. The algorithm must always finish or terminate after a finite number of steps.

Effectiveness. All operations to be accomplished must be sufficiently basic that they can be done exactly and in finite length.

Algorithm specification:

Algorithm	Pseudocode
It is a step-by-step description of the solution.	It is an easy way of writing algorithms for users to understand.
It is always a real algorithm and not fake codes.	These are fake codes.
They are a sequence of solutions to a problem.	They are representations of algorithms.
It is a systematically written code.	These are simpler ways of writing codes.
They are an unambiguous way of writing codes.	They are a method of describing codes written in an algorithm.
They can be considered pseudocode.	They can not be considered algorithms
There are no rules to writing algorithms.	Certain rules to writing pseudocode are there.

Recursive Algorithms

A recursive algorithm calls itself which generally passes the return value as a parameter to the algorithm again. This parameter indicates the input while the return value indicates the output.

Recursive algorithm is defined as a method of simplification that divides the problem into sub-problems of the same nature. The result of one recursion is treated as the input for the next recursion. The repletion is in the self-similar fashion manner. The algorithm calls itself with smaller input values and obtains the results by simply accomplishing the operations on these smaller values. Generation of factorial, Fibonacci number series are denoted as the examples of recursive algorithms.

Example: Writing factorial function using recursion

```
intfactorialA(int n)
{
  return n * factorialA(n-1);
}
```

Performance analysis:

Performance analysis of an algorithm depends upon two factors i.e. amount of memory used and amount of compute time consumed on any CPU. Formally they are notified as complexities in terms of:

1.Space Complexity.

2.Time Complexity.

1.Space Complexity of an algorithm is the amount of memory it needs to run to completion i.e. from start of execution to its termination. Space need by any algorithm is the sum of following components:

Fixed Component: This is independent of the characteristics of the inputs and outputs. This part includes: Instruction Space, Space of simple variables, fixed size component variables, and constants variables.

Variable Component: This consist of the space needed by component variables whose size is dependent on the particular problems instances(Inputs/Outputs) being solved, the space needed by referenced variables and the recursion stack space is one of the most prominent components. Also this included the data structure components like Linked list, heap, trees, graphs etc.

Therefore the total space requirement of any algorithm 'A' can be provided as

Space(A) = Fixed Components(A) + Variable Components(A)

Among both fixed and variable component the variable part is important to be determined accurately, so that the actual space requirement can be identified for an algorithm 'A'. To identify the space complexity of any algorithm following steps can be followed:

1. Determine the variables which are instantiated by some default values.
2. Determine which instance characteristics should be used to measure the space requirement and this is will be problem specific.
3. Generally the choices are limited to quantities related to the number and magnitudes of the inputs to and outputs from the algorithms.
4. Sometimes more complex measures of the interrelationships among the data items can used.

Example: Space Complexity

Algorithm Sum(number,size)\\ procedure will produce sum of all numbers provided in 'number' list

```
{
    result=0.0;
    for count = 1 to size do                \\will repeat from 1,2,3,4,....size times
        result= result + number[count];
    return result;
}
```

In above example, when calculating the space complexity we will be looking for both fixed and variable components. here we have

Fixed components as 'result','count' and 'size' variable there for total space required is three(3) words.

Variable components is characterized as the value stored in 'size' variable (suppose value store in variable 'size 'is 'n'). because this will decide the size of 'number' list and will also drive the for loop. therefore if the space used by size is one word then the total space required by 'number' variable will be 'n'(value stored in variable 'size').

therefore the space complexity can be written as **Space(Sum) = 3 + n;**

2.Time Complexity of an algorithm(basically when converted to program) 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 the run/execution time . The compile time is independent of the instance(problem specific) characteristics. following factors effect the time complexity:

1. Characteristics of compiler used to compile the program.
2. Computer Machine on which the program is executed and physically clocked.
3. Multiuser execution system.
4. Number of program steps.

Therefore the again the time complexity consist of two components fixed(factor 1 only) and variable/instance(factor 2,3 & 4), so for any algorithm 'A' it is provided as:

Time(A) = Fixed Time(A) + Instance Time(A)

Here the number of steps is the most prominent instance characteristics and The number of steps any program statement is assigned depends on the kind of statement like

- comments count as zero steps,
- an assignment statement which does not involve any calls to other algorithm is counted as one step,
- for iterative statements we consider the steps count only for the control part of the statement etc.

Therefore to calculate total number program of program steps we use following procedure. For this we build a table in which we list the total number of steps contributed by each statement. This is often arrived at by first determining the number of steps per execution of the statement and the frequency of each statement executed. This procedure is explained using an example.

Asymptotic notation:

There are mainly three asymptotic notations:

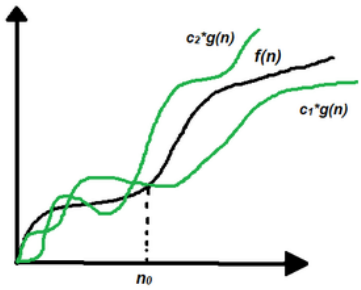
1. **Big-O Notation (O-notation)**
2. **Omega Notation (Ω-notation)**
3. **Theta Notation (Θ-notation)**

1. Theta Notation (Θ-Notation):

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.

.Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1* g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0



Theta notation

Mathematical Representation of Theta notation:

*Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1 * g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0}*

***Note:** Θ(g) is a set*

The above expression can be described as if f(n) is theta of g(n), then the value f(n) is always between c1 * g(n) and c2 * g(n) for large values of n (n ≥ n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

The execution time serves as both a lower and upper bound on the algorithm's time complexity.

It exist as both, most, and least boundaries for a given input value.

A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example**,** Consider the expression **$3n^3 + 6n^2 + 6000 = \Theta(n^3)$**, the dropping lower order terms is always fine because there will always be a number(n) after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.

Examples :

{ 100 , log (2000) , 10^4 } belongs to **Θ(1)**

{ (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **Θ(n)**

{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **$\Theta(n^2)$**

Note: Θ provides exact bounds.

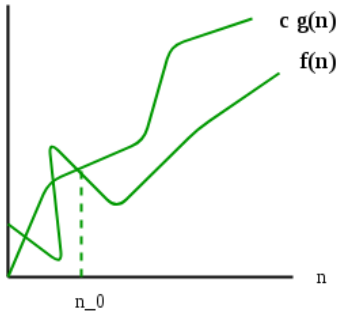
2. Big-O Notation (O-notation):

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

.It is the most widely used notation for Asymptotic analysis.
.It specifies the upper bound of a function.
.The maximum time required by an algorithm or the worst-case time complexity.
.It returns the highest possible output value(big-O) for a given input.
.Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time possible.

If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constant C and n0 such that, $0 \leq f(n) \leq cg(n)$ for all $n \geq n0$

It returns the highest possible output value (big-O)for a given input.
The execution time serves as an upper bound on the algorithm's time complexity.



f(n) = O(g(n))

Mathematical Representation of Big-O Notation:

O(g(n)) = { f(n): there exist positive constants c and n0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n0$ }

For example**,** Consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of the Insertion sort is $O(n^2)$.
Note: $O(n^2)$ also covers linear time.
If we use Θ notation to represent the time complexity of Insertion sort, we have to use two statements for best and worst cases:

- The worst-case time complexity of Insertion Sort is $\Theta(n^2)$.
- The best case time complexity of Insertion Sort is $\Theta(n)$.

The Big-O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

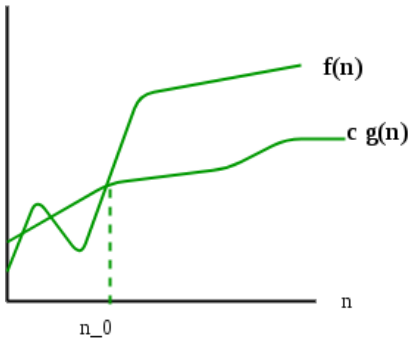
Examples :
{ 100 , log (2000) , 10^4 } belongs to **O(1)**
U { (n/4) , (2n+3) , (n/100 + log(n)) } belongs to **O(n)**
U { (n^2+n) , (2n^2) , (n^2+log(n))} belongs to **O(n^2)**
Note: Here, **U represents union**, we can write it in these manner because **O provides exact or upper bounds .**

3. Omega Notation (Ω-Notation):

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

The execution time serves as a lower bound on the algorithm's time complexity. It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Ω(g), if there is a constant c > 0 and a natural number n0 such that c*g(n) ≤ f(n) for all n ≥ n0



f(n) = Omega(g(n))

Mathematical Representation of Omega notation :

Ω(g(n)) = { f(n): there exist positive constants c and n0 such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in worst-case and sometimes in the average case.

Examples :

{ (n^2+n) , (2n^2) , (n^2+log(n))} belongs to ***Ω(n^2)***
U *{ (n/4) , (2n+3) , (n/100 + log(n)) } belongs to* ***Ω(n)***
U *{ 100 , log (2000) , 10^4 } belongs to* ***Ω(1)***
Note: *Here,* ***U represents union,*** *we can write it in these manner because* ***Ω provides exact or lower bounds.***

Randomized Algorithms:

Randomized algorithm is a different design approach taken by the standard algorithms where few random bits are added to a part of their logic. They are different from deterministic algorithms; deterministic algorithms follow a definite procedure to get the same output every time an input is passed where randomized algorithms produce a different output every time they're executed. It is important to note that it is not the input that is randomized, but the logic of the standard algorithm.

Classification of Randomized Algorithms

Randomized algorithms are classified based on whether they have time constraints as the random variable or deterministic values. They are designed in their two common forms − Las Vegas and Monte Carlo.

- **Las Vegas** − The Las Vegas method of randomized algorithms never gives incorrect outputs, making the time constraint as the random variable. For example, in string matching algorithms, las vegas algorithms start from the beginning once they encounter an error. This increases the probability of correctness. Eg., Randomized Quick Sort Algorithm.
- **Monte Carlo** − The Monte Carlo method of randomized algorithms focuses on finishing the execution within the given time constraint. Therefore, the running time of this method is deterministic. For example, in string matching, if monte carlo encounters an error, it restarts the algorithm from the same point. Thus, saving time. Eg., Karger's Minimum Cut Algorithm

ELEMENTARY DATA STRUCTURES

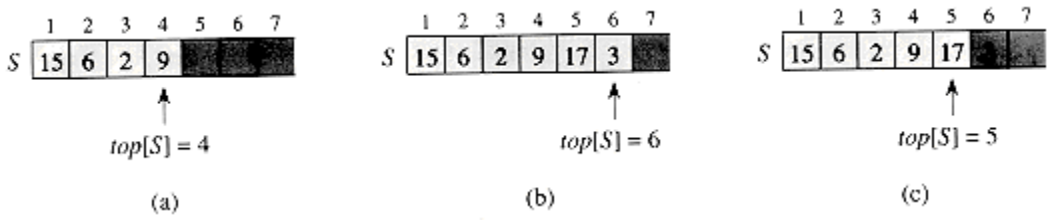
Although many complex data structures can be fashioned using pointers, we present only the rudimentary ones: stacks, queues, linked lists, and rooted trees.

Stacks

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. These names are allusions to physical stacks, such as the spring-loaded stacks of plates used in cafeterias. The order in which plates are popped from the stack is the reverse of the order in which they were pushed onto the stack, since only the top plate is accessible.

As shown in Figure 11.1, we can implement a stack of at most n elements with an array $S[1..n]$. The array has an attribute $top[S]$ that indexes the most recently inserted element. The stack consists of elements $S[1..top[S]]$, where $S[1]$ is the element at the bottom of the stack and $S[top[S]]$ is the element at the top.

When $top[S] = 0$, the stack contains no elements and is ***empty***. The stack can be tested for emptiness by the query operation STACK-EMPTY. If an empty stack is popped, we say the stack ***underflows***, which is normally an error. If $top[S]$ exceeds n, the stack ***overflows***. (In our pseudocode implementation, we don't worry about stack overflow.)

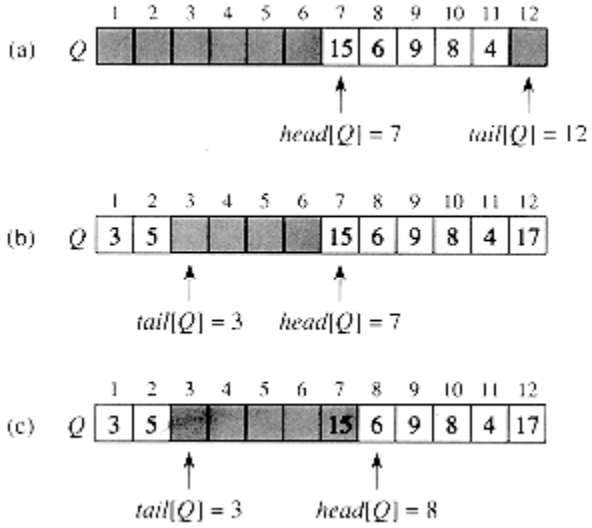


The stack operations can each be implemented with a few lines of code.

```
STACK-EMPTY(S)
1  if top [S] = 0
2     then return TRUE
3     else return FALSE
PUSH(S, x)
1  top[S] ← top [S] + 1
2  S [top[S]] ← x
POP(S)
1  if STACK-EMPTY(S)
2    then error "underflow"
3    else top [S] ← top [S] - 1
4         return S [top [S] + 1]
```

Queues

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE; like the stack operation POP, DEQUEUE takes no element argument. The FIFO property of a queue causes it to operate like a line of people in the registrar's office. The queue has a ***head*** and a ***tail***. When an element is enqueued, it takes its place at the tail of the queue, 1ike a newly arriving student takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the student at the head of the line who has waited the longest. (Fortunately, we don't have to worry about computational elements cutting into line.)



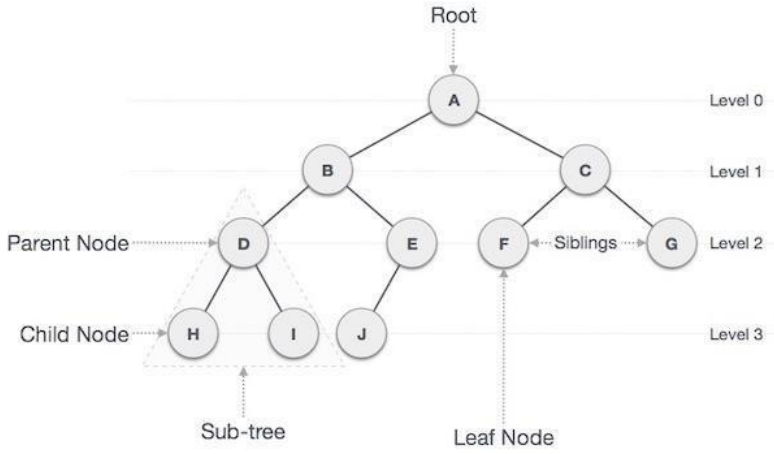
In our procedures ENQUEUE and DEQUEUE, the error checking for underflow and overflow has been omitted. (Exercise 11.1-4 asks you to supply code that checks for these two error conditions.)

```
ENQUEUE(Q, x)
1  Q [tail [Q]] ← x
2  if tail [Q] = length [Q]
3     then tail [Q] ← 1
```

4 **else** *tail* [*Q*] ← *tail* [*Q*] + 1
DEQUEUE(*Q*)
1 *x* ← *Q* [*head* [*Q*]]
2 **if** *head* [*Q*] = *length* [*Q*]
3 **then** *head* [*Q*] ← 1
4 **else** *head* [*Q*] ← *head* [*Q*] + 1
5 **return** *x*

Trees

A tree is a non-linear abstract data type with a hierarchy-based structure. It consists of nodes (where the data is stored) that are connected via links. The tree data structure stems from a single node called a root node and has subtrees connected to the root.



Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on

Types of Trees

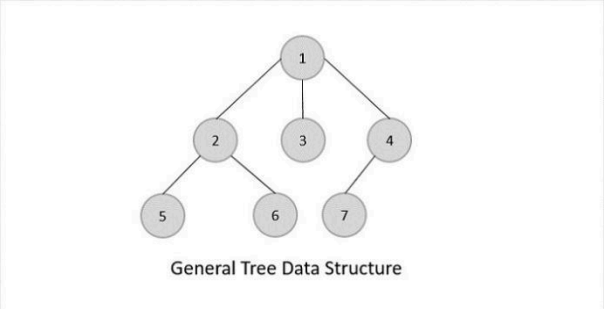
There are three types of trees −

- General Trees
- Binary Trees
- Binary Search Trees

General Trees

General trees are unordered tree data structures where the root node has minimum 0 or maximum ‘n’ subtrees.

The General trees have no constraint placed on their hierarchy. The root node thus acts like the superset of all the other subtrees.



General Tree Data Structure

Binary Trees

Binary Trees are general trees in which the root node can only hold up to maximum 2 subtrees: left subtree and right subtree. Based on the number of children, binary trees are divided into three types.

Full Binary Tree

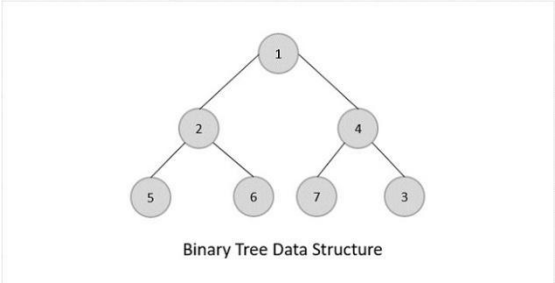
- A full binary tree is a binary tree type where every node has either 0 or 2 child nodes.

Complete Binary Tree

- A complete binary tree is a binary tree type where all the leaf nodes must be on the same level. However, root and internal nodes in a complete binary tree can either have 0, 1 or 2 child nodes.

Perfect Binary Tree

- A perfect binary tree is a binary tree type where all the leaf nodes are on the same level and every node except leaf nodes have 2 children.

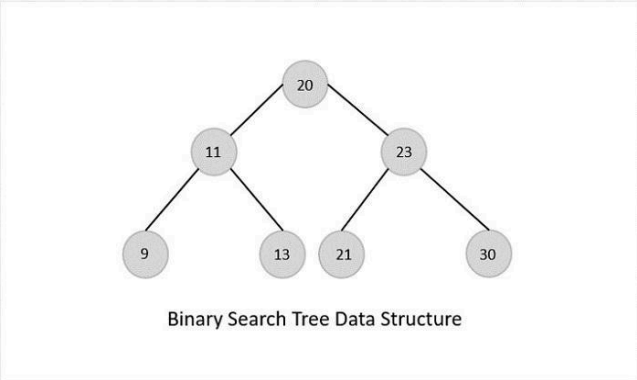


Binary Tree Data Structure

Binary Search Trees

Binary Search Trees possess all the properties of Binary Trees including some extra properties of their own, based on some constraints, making them more efficient than binary trees.

The data in the Binary Search Trees (BST) is always stored in such a way that the values in the left subtree are always less than the values in the root node and the values in the right subtree are always greater than the values in the root node, i.e. left subtree < root node ≤ right subtree.



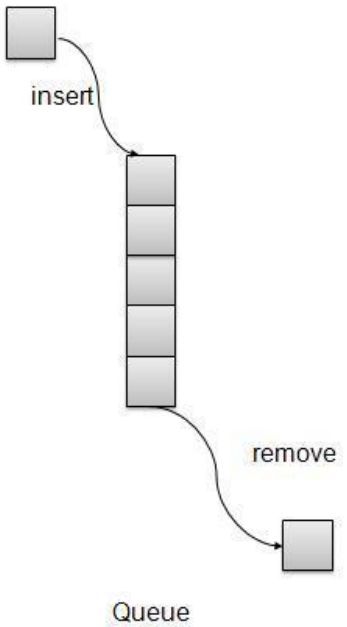
Binary Search Tree Data Structure

Priority Queue:

priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

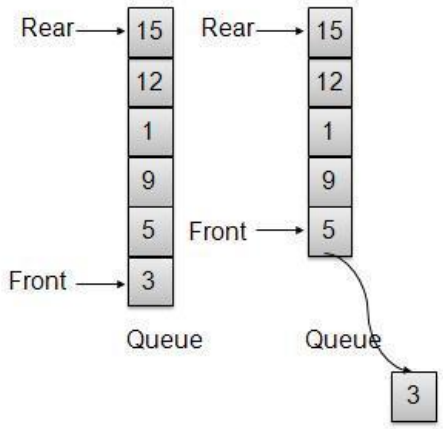
- **insert / enqueue** − add an item to the rear of the queue.
- **remove / dequeue** − remove an item from the front of the queue.
- Priority Queue Representation



Queue

We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.
- Remove / Dequeue Operation
- Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.



- One Item removed from front