Master Theorem

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a \geq 1$, $b > 1$, $k \geq 0$ and p is real number.

1) if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2) if $a = b^k$
   a) if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
   b) if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
   c) if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

3) if $a < b^k$
   a) if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b) if $p < 0$, then $T(n) = O(n^k)$.

Problem: 1

1) $T(n) = 3T(n/2) + n^2$.

Sol: $a = 3$, $b = 2$, $k = 2$, $p = 0$.

here

$a \quad b^k$

$3 \quad 2^2$.

$3 < 4$.

So this is case 3 & $p \geq 0$, So it fall under 3a → case.

$T(n) = \Theta(n^2 \log^0 n)$

$= \Theta(n^2)$.

2) $T(n) = 4T(n/2) + n^2$.

Sol: $a = 4$, $b = 2$, $k = 2$, $p = 0$.

$4 \quad 2^2$

$4 = 4$.

condition is 2, Now we have to check p, condition, is $p = 0$

So the condition is

2a)

$$T(n) = \Theta(n^{\log_2 4} \log n)$$
$$= \Theta(n^2 \log n)$$

3) $T(n) = T(n/2) + n^2$

$a=1, b=2, k=2, p=0$.

$1 < 2^2$.

This falls in 3a.

$$T(n) = \Theta(n^2 \log^0 n)$$
$$= \Theta(n^2).$$

4) $T(n) = 2^n T(n/2) + n^n$. $\longrightarrow$ in this master theorem cannot be. applied, as the form say

$$aT(n/b) + \Theta(n^k \log^p n)$$

so we cannot apply master theorem.

5) $T(n) = 16T(n/4) + n$.

Sol $a=16, b=4, k=1, P=0$.

$16 > 4$ $\rightarrow$ condition(1) $T(n) = \Theta(n^{\log_b a})$

$$= \Theta(n^{\log_4 16}) = \Theta(n^2).$$

6) $T(n) = 2T(n/2) + n\log n$

$a=2, b=2, k=1, P=1$.

$2 = 2$.

condition 2(a).

$$T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$$
$$= \Theta(n \log^2 n)$$
$$= \Theta(n \log^2 n).$$

7) $T(n) = 2T(n/2) + n/\log n$.

= we can re-write the following Equation in the below form.

$2T(n/2) + n\log^{-1} n$.

$a = 2,\ b = 2,\ k = 1,\ p = -1$

$2 = 2^1$

so its

2(b)

$T(n) = \Theta(n^{\log_b a} \log\log n)$

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

8) $T(n) = 2T(n/4) + n^{0.51}$

Sol $a = 2\ b = 4\ k = 0.51\ p = 0$.

$2 < 4^{0.51}$

3a → case.

$T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^{0.51})$.

9) $T(n) = 0.5T(n/2) + 1/n$. } this case cannot apply master Theorm as $a < 1$, ~~buts~~

$a = 0.5$

10) $T(n) = 6T(n/3) + n^2$

$a = 6,\ b = 3,\ k = 2,\ p = 1$

$6 < 3^2$

3a → case based on comparsiou

$T(n) = \Theta(n^k \log^p n)$

$= \Theta(n^2 \log^1 n)$

$= \Theta(n^2 \log n)$

11) $T(n) = 64T(n/8) - n^2 \log n$.

this also cannot solve using master theorm.
as the format doesn't match with

$$aT(n/b) + \Theta(n^k \log^p n)$$

12) $T(n) = 7T(n/3) + n^2$.

$a = 7,\ b = 3,\ k = 2,\ p = 0$.

$7 < 3^2$.

It's in condition 3 a.

$$T(n) = \Theta(n^k \log^p n)$$
$$= \Theta(n^2 \log^0 n)$$
$$= \Theta(n^2).$$

13) $T(n) = 4T(n/2) + \log n$.

$a = 4$. $b = 2$, $k = 0$, $p = 1$
since there is no, k-value in the equation

$4 > 2^0$
here it's case 1

$$T(n) = \Theta(n^{\log_b a})$$
$$T(n) = \Theta(n^{\log_b a})$$
$$= \Theta(n^2)$$

14)

$T(n) = \sqrt{2}\, T(n/2) + \log n$

$a = \sqrt{2}$ $b = 2$ $k = 0$ $p = 1$

$\sqrt{2} > 2^0$

1- case.

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 \sqrt{2}})$

$= \Theta(\sqrt{n})$

15)

$T(n) = 2T(n/2) + \sqrt{n}$

$a = 2,\ b = 2,\ k = 1/2,\ p = 0$

$2 > 2^{1/2}$ $T(n) = \Theta(n^{\log_2 2})$

$= \Theta(n)$

16) $T(n) = 3T(n/3) + \sqrt{n}$.

$a = 3$ $b = 3$ $k = 1/2,\ p = 0$

$3 > 3^{1/2} \rightarrow$ case 1

$T(n) = \Theta(n^{\log_3 3}) = \Theta(n)$.

Arrays

Data structures are classified as either linear or non-linear.
A data structure is said to be linear if its elements from a sequence, or in other words, a linear list.

There are two basic ways of representing such linear structures in memory.

↳ one way is to have the linear relationship between the elements represented by means of sequential memory locations.
These linear structures are called arrays ~~and~~.
~~form other main~~

The other way is to have the linear relationship between the elements represented by means of pointers or links.
These linear structures are called linked lists

The operations one normally performs on any linear structure whether it be an array or a linked list, include the following

a) Traversal → processing each element in the list.
b) Search. finding the location of the element with a given value or the record with a given key.
c) Insertion : Adding a new element to the list
d) deletion : Removing an element from the list
e) Sorting : Arranging the element in some type of order.
f) Merging : combing two lists into a single list.

The particular linear structure that one chooses for a given situation depends on the relative frequency with which one performs these different operations on the structure.

* Since arrays are usually easy to traverse, search and sort they are frequently used to store relatively permanent collections of data.

On the other hand, if the size of the structure and the data in the structure are constantly changing, then the array may not be as useful a structure as the linked list

Linear Array

A linear array is a list of a finite number n of homogeneous data elements (i.e data elements of the same type) Such that:

a) The elements of the array are referenced respectively by an index set consisting of consecutive numbers.

b) The elements of the array are stored respectively in successive memory locations.

The number n of elements is called the length or size of the array.

If not explicitly stated, we will assume the index set consists of the integers $1, 2, \ldots, n$.

* In general, the length or the number of data elements of the array can be obtained from the index set by the formula.

$$\text{Length} = UB - LB + 1.$$

where

UB is the largest index, called the upper bound,

& LB is the smallest index, called the lower bound of the array.

Note the length = UB when LB = 1

The elements of an array A may be denoted by the subscript notation

$$A_1, A_2, A_3, \ldots, A_n.$$

or by the bracket notation (used in C).

A[1], A[2], A[3], ... A[N].

We will usually use the subscript notation or the bracket notation.

Regardless of the notation, the number K in A[K] is called a ~~se~~ subscript or an index and A[K] is called a subscripted variable.

Note: that subscripts allow any element of A to be referenced by its relative position in A.

Ex:-1

Let DATA be a 6-element linear array of integers such that

DATA[1] = 247, DATA[2] = 56 DATA[3] = 429 DATA[4] = 135

DATA[5] = 87 DATA[6] = 156.

Sometimes we will denote such an array by simply writing

DATA: 247, 56, 429, 135, 87, 156.

The array DATA is frequently pictured as.

DATA

1	247
2	56
3	429
4	135
5	87
6	156

(a)

DATA

247	56	429	135	87	156
1	2	3	4	5	6

(b)

Ex:-

an automobile company uses an array AUTO to record the number of automobiles sold each year from 1932 through ~~tt~~ 1984.

Rather than beginning the index set with 1, it is more useful to begin the index set with 1932 so that

AUTO [K] = number of automobiles solid in the year K

Then LB = 1932 is the lower bound and UB = 1984 is the upper bound of AUTO

length = UB - LB + 1 = 1984 - 1932 + 1 = 55.

zli od is AUTO,

Pg: 1

Defining Arrays in C*/

```
#include <stdio.h>
main ()
{
  int a[10];  // 1
  for (int i = 0; i<10; i++)
   {
     a[i] = i;
   }
   print array(a);
  }
 ~~pa~~ void print array (int a[])
{
 for (int i = 0; i< 10; i++).
  {
   print f ("value in the array %d\n", a[i]);
  }
}
```

The above program helps to define an array.
Statement 1 defines an array of integers of the size 10, which means you can store 10 integers.

* when we define the array, the size should be known.
Subscripts are used to refer the Elements of the array where 0 is considered to be the lowest subscript always and the highest subscript is (size-1), which is 9 in this case.

we can refer to any element as a[0], a[1], a[2], ...

* Each programming language has its own rules for declaring arrays.
Each such declaration must give, implicity or Explicitly, three items of information.
1) the name of the array.
2) the data type of the array.
3) the index set of the array.

Ex:-
float DATA[6];
(Suppose data is a 6-element linear array containing real values. C-Language declares such an array as follows).

b) Consider the integer array AUTO with the lower bound LB=1932 and upper bound UB=1984.
In C, the lower bound of an array is always 0;
it cannot be customized to some other value.
Thus, to implement a scenario where LB=1932,
we need to use an offset value that logically represent the LB index value as 1932.

Disadvantages:-

↳ it is relatively expensive to insert and delete elements in an array

↳ Also, since an array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. (for this reason, arrays are called dense lists and are said to be static data structure).

Another way of storing a list in memory is to have each element in the list contain a field, called a link or pointer which contains the address of the next Element in the list.

Thus successive elements in the list need not occupy adjacent space in memory.

This will make it easier to insert & delete elements in the list.

Accordingly, if one were mainly interested in searching through data for inserting and deleting, as in word processing, one should not store the data in array but rather in a list using pointers.

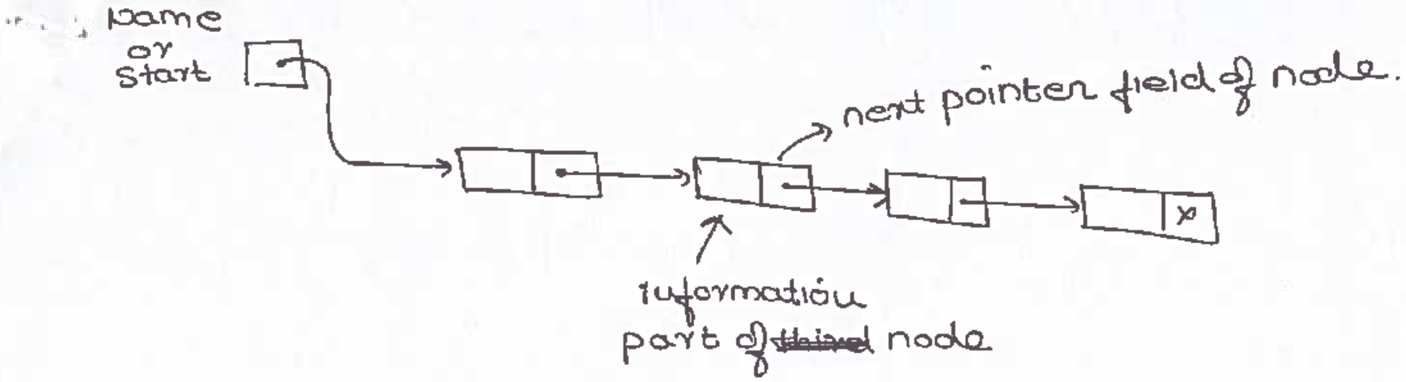
This latter type of data structure is called a linked list

~~and is the main s~~

Linked lists

A linked lists, or one-way list, is a linear collection of data elements, called nodes, where the linear order is given by means of pointers.

That is, each node is divided into two parts: the first part contains the information of the element, and the second part, called the link field or next pointer field, contains the address of the next node in the list.



The null pointer, denoted by x in the diagram.
The linked list also contains a list pointer variable - called START or Name - which contains the address of the first node in the list; hence there is an arrow drawn from START to the first node.
Clearly, we need only this address in START to trace through the list.
A special case is the list that has no-nodes. Such a list is called the null list or empty list and is denoted by the null pointer in the variable START.

Experimental Studies:

- write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- use a method like System.currentTimeMills() or clock function. to get an accurate measure of the actual running time.
- plot the results.

Limitations of experiments

Results may not be indicative of the running time as all inputs may not be included in the experiment

In order to compare two algorithms the same hardware & software environments must be used

- it is necessary to implement the algorithm, and the efficiency of implementation varies

Asymptotic notation.

O(n)

o(n)

Ω(n)

ω(n) → small omega.

Θ(n).

Edmond Landau

1877-1938,

Inventor of the asymptotic notation.

Donald E. Knuth.

1938 → is an american computer scientists.

He is the author of the multi-volume work the art of computer programming

He contributed to the development of the rigorous analysis of the computational complexity of algorithms and systematized formal mathematical techniques for it.

↳ Turing award, 1974.

↳ father of the analysis of algorithm.

Theoretical analysis

↳ uses a high-level description of the algorithm instead of an implementation.

↳ characterizes running times as a function of the input size, n.

↳ Takes into account all possible inputs

↳ Allows us to evaluate the speed of an algorithm independent of the hardware / software environment.

Pseudocode.

↳ High-level description of an algorithm.

↳ more structured than English prose.

↳ Less detailed than a program.

↳ preferred notation for describing algorithms.

↳ Hides program design issues.

Ex: find max element of an array

```
Algorithm arrayMax(A,n)
Input array A of n integers.
Output Maximum element of A.
    currentMax ← A[0]
    for i ← 1 to n-1 do
      if A[i] > currentMax then.
        currentMax ← A[i]
    return currentMax.
```

Pseudocode Details

Control flow
- if ... then ... [else ..]
- while .. do ..
- ... repeat .. until
- for .. do ..
- Indentation replaces braces

Expressions
- ← Assignment (like = in C/Java)
- = Equality testing (like == in C/Java)

n^2 superscripts and other mathematical formatting allowed.

- method declaration
  - Algorithm method (arg [, arg..])
  - Input..
  - output..

Primitive operations.

- Basic computation performed by an algorithm
- Identifiable in pseudocode.
- Largely independent from the programming language
- Exact definition not important (we will see why later).
- Assumed to take a constant amount of time in the RAM model.

Example:

- Assigning a value to a variable
- indexing into an array
- calling a method
- Returning from a method.

Example

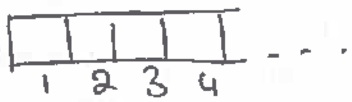
cnt ← cnt + 1
A[5].
mySort(A, n)
return(cnt).

RAM: The Random Access Machine

- for theoretical analysis, we assume RAM model for. our "theoretical" computer.

- our RAM model consists of:
  - a CPU
  - a potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character

- memory cells are numbered and accessing any cell in memory takes unit time.



Counting primitive operations

Algorithm arrayMax(A, n)	operations
currentMax ← A[0]	2.
for (i = 1; i < n; i++)	2n.
$\downarrow$ 1, $\downarrow$ n, $\downarrow$ n-1	
if A[i] > current then (n-1)	2(n-1)
currentMax ← A[i] (n-1)	2(n-1)
return currentMax	1
	6n-1.

1(n-1) + 1(n-1)
n-1+n-1
2n-2.
= 2(n-1)

2+2n+2n-2+2n-2+1
= 6n-1

1+n+n-1

Estimating Running time:

↳ Algorithm arrayMax executes $6n-1$ primitive operations in the worst case.

Define

a = Time taken by the fastest primitive operation

b = Time taken by the slowest primitive operation

↳ Let $T(n)$ be worst-case time of arrayMax. Then.

$$a(6n-1) \le T(n) \le b(6n-1).$$

↳ Hence, the running time $T(n)$ is bounded by two linear function.

Time Complexity- why should we care?

Prime?

2
3
5
7
11

example of prime number

Ram

for $i \leftarrow 2$ to $n-1$
  if i divides n
  n is not prime

1ms for a division
in worst case
$(n-2)$ times

$n = 11$ 9ms.

$n = 101 \longrightarrow$ 99ms

$T \propto n$

Time is proposition to input
here it $O(n)$
↑
Big O

Shyam

for $i \leftarrow 2$ to $\sqrt{n}$ [till the square root of n].
  if i divides n.
  n is not prime

worst case.
$(\sqrt{n}-1)$ times
$(3-1) = 2$ms

$(\sqrt{101}-1)$ times
$(10-1)$ times $= 9$ms.

$T \propto \sqrt{n}$.

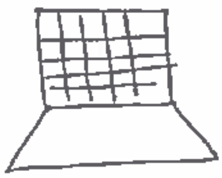
Time is proposition to $\sqrt{n}$ this is $O(\sqrt{n})$

How to analyze time complexity?

Running time depends upon.

1) Single vs multi processor
2) Read/write speed to memory.
3) 32-bits vs 64-bits
4) Input
   ↳ rate of growth of time Algorithm.

→ model machine [Hypotecial model].



→ Single processor
→ 32-bit
→ Sequential Execution
→ 1 unit time for assignment & return.
→ 1 unit time for arithmetical and logical operations

Other Example
To find the sum of integers

	Sum of list (A, n)	Cost	no of times
	{		
1.	Total = 0	1 (c_1)	1
2.	for i = 0 to n-1	2 (c_2)	n+1 [extra for false condition].
3.	total = total + A;	2 (c_3)	n
4.	return total	1 (c_4)	1

$$T_{\text{sum of list}} = 1 + 2(n+1) + 2n + 1$$
$$= 1 + 2(n+1) + 2n + 1$$
$$= 4n + 4.$$

$$T(n) = cn + c'$$

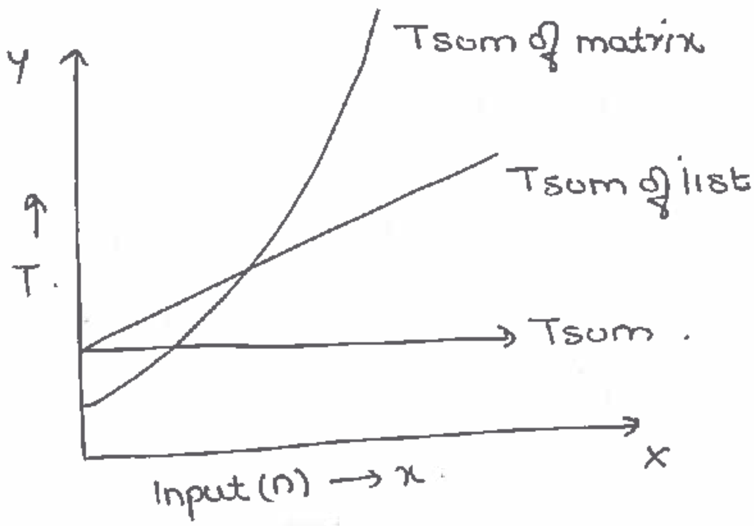
where $c = c_2 + c_3$.
$c' = c_1 + c_2 + c_4$.

Example

1) Tsum = K → all the function of form some constant.
O(1)

2) Tsum of list = Cn + C' → all linear function O(n)
[Linear function]

3) Tsum of matrix = $an^2+bn+c \longrightarrow O(n^2)$
↳ set of all the function



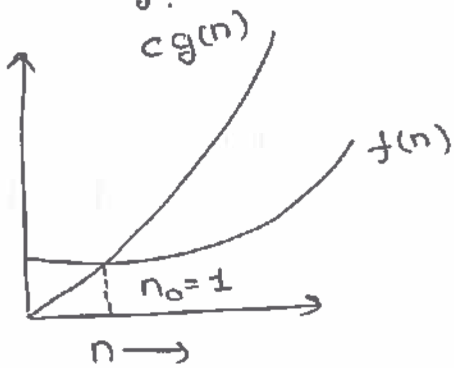
Time complexity analysis → asymptotic notations

Algo 1: $T(n) = 5n^2 + 7$
Algo 2: $T(n) = 17n^2 + 6n + 8$
} model machine

1) $n \to \infty$
2) no constant.

O - "big oh" notation → upper bound.

$O(g(n)) = \{ f(n)$: there exist constant c and n_0,
$f(n) \le c\,g(n)$, for $n \ge n_0 \}$.



$f(n) = 5n^2 + 2n + 1 \in O(n^2)$.

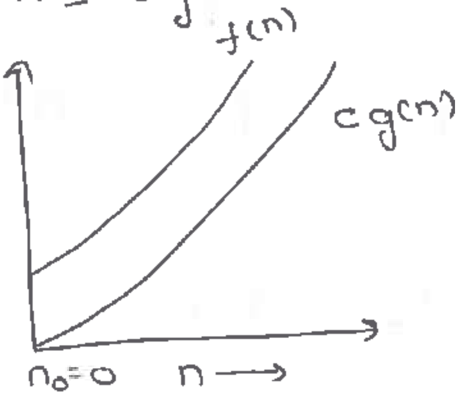
$g(n) = n^2$

$c = 5+2+1$; $f(n) \le 8n^2, n \ge 1$.

$n_0 = 1$,

Ω- omega notation - lower bound.

$\Omega(g(n)) = \{f(n)$: there exist constant c and n_0,
$cg(n) \le f(n)$ for $n \ge n_0 \}$



$f(n) = 5n^2 + \underbrace{2n+1}$

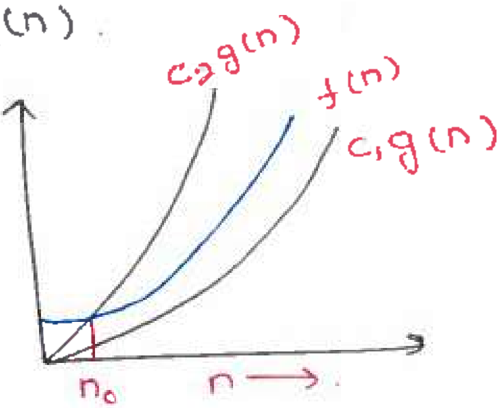
$g(n) = n^2$.

$c = 5$. $\quad 5n^2 \le f(n),\ n \ge 0$

$n_0 = 0$

Θ-theta notation. - Tight bound.

$\Theta(g(n)) = \{f(n)$: there exists constants c_1, c_2 and n_0
$c_1 g(n) \le f(n) \le c_2 g(n)$.
for $n \ge n_0 \}$



$f(n) = 5n^2 + 2n + 1 = \Theta(n^2)$.

$g(n) = n^2$.

$c_1 = 5,\ c_2 = 8$

$n_0 = 1$.

Time complexity analysis - some general rules.

we analyze time complexity for.

a) very large input size

b) worst-case scenario

$T(n) = n^3 + \underbrace{3n^2 + 4n + 2}_{\text{in}}$.

$\simeq n^3 \quad (n \to \infty)$

$cn^3 \quad O(n^3)$.

Rule: a) drop lower order terms.
b) drop constant multiplier

AVL (Adelson-Velskii and Landis) trees.

AVL-Trees:

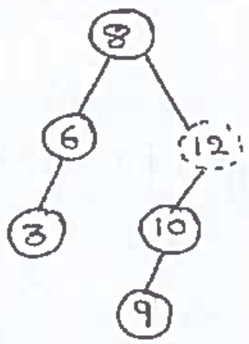
In HB(K), if $k=1$ (if balance factor is one), such a binary search tree is called an AVL Tree.

That means an AVL tree is a binary search tree with a balance condition: the difference between left subtree height and right subtree height is at most 1.

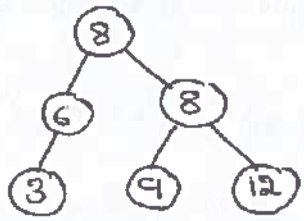
Properties of AVL Trees:

A binary tree is said to be an AVL Tree, if:

- it is a binary search tree, and
- for any node x, the height of left subtree of x and height of right subtree of x differ by at most 1.



(a)



(b)

Ex:- a is not an AVL Tree.
b is a AVL tree.

Minimum/Maximum Number of Nodes in AVL Tree:

for simplicity let us assume that the height of an AVL tree is h and $N(h)$ indicates the number of nodes in AVL tree with height h. To get the minimum number of nodes with height h, we should fill the tree with the min-number of node possible.

That means if we fill the left subtree with height $h-1$ then we should fill the right subtree with height $h-2$.

As a result, the min-number of nodes with height h is

$$N(h) = N(h-1) + N(h-2) + 1$$

In the above equation:

- $N(h-1)$ indicates the minimum number of nodes with height $h-1$.
- $N(h-2)$ indicates the minimum number of nodes with height $h-2$.

* In the above expression, "1" indicates the current node.

We can give $N(h-1)$ either for left subtree or right subtree.

Solving the above recurrence gives:

$$N(h) = O(1.618^h) \Rightarrow h = 1.44 \log n \approx O(\log n)$$

Where n is the number of nodes in AVL tree. Also, the above derivation says that the maximum height in AVL trees is $O(\log n)$

Similarly, to get maximum number of nodes, we need to fill both left and right subtrees with height $h-1$. As a result, we get:

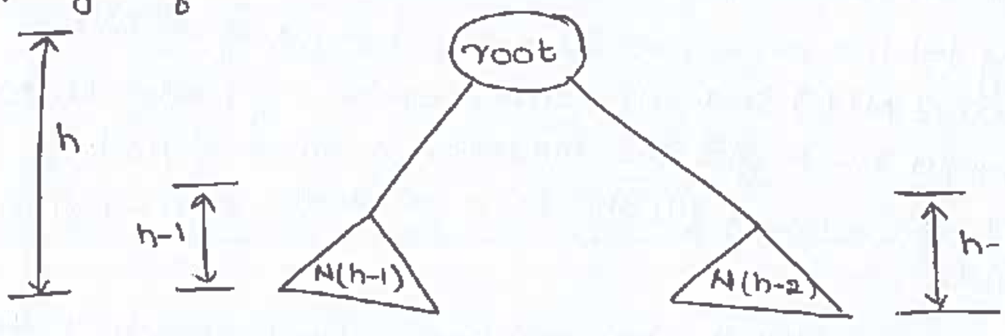
$$N(h) = N(h-1) + N(h-1) + 1 = 2N(h-1) + 1$$

The above Expression defines the case of full binary tree.

Solving the recurrence we get:

$$N(h) = O(2^h) \Rightarrow h = \log n \approx O(\log n).$$

$\therefore$ In both the cases, AVL tree property is ensuring that the height of an AVL tree with n-nodes is $O(\log n)$.



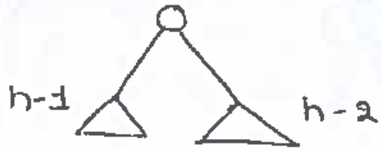
Height of an AVL Tree

fact: the height of an AVL tree storing 'n' keys is $O(\log n)$.

Proof: Let us bound $n(h)$: the minimum number of internal nodes of an AVL tree of height h.

- we easily see that $n(1) = 1$ and $n(2) = 2$.
- for $h > 2$, AVL tree with minimum no of nodes is such that both its subtrees are AVL trees with minimum no of nodes.
- Such an AVL tree contains the root node, one AVL subtree of height $n-1$ and another of height $n-2$

That is, $n(h) = 1 + n(h-1) + n(h-2)$.



- knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$

so $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(n-6)$ ...

by induction,

$n(h) > 2^i n(h-2i)$

choosing i such that $h-2i$ is either 1 or 2.

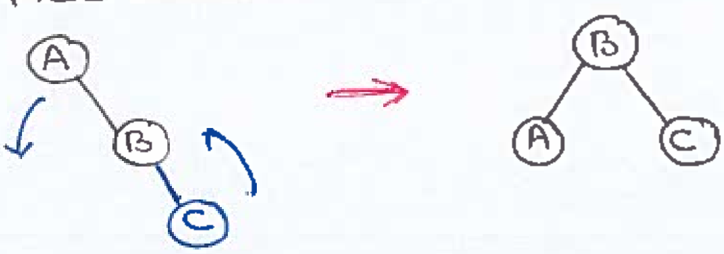
we get: $n(h) > 2^{h/2-1}$.

Taking logarithms: $h < 2\log n(h) + 2$.

Thus the height of an AVL Tree is $O(\log n)$.

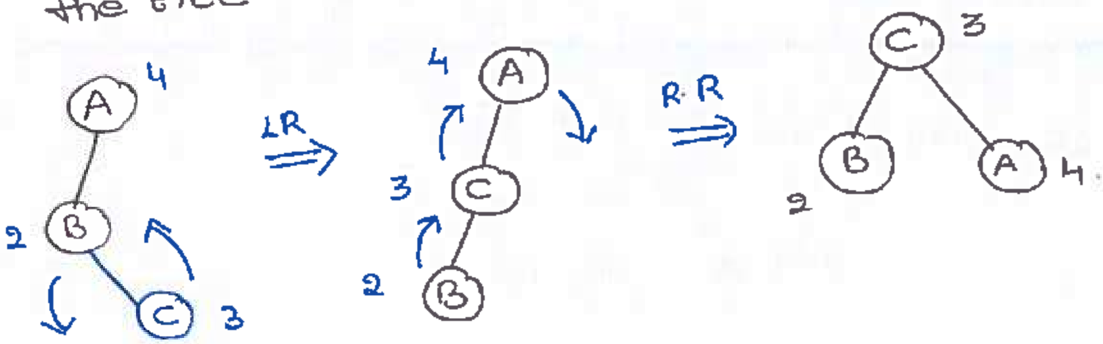
Rule to convert from non-AVL to AVL know as AVL-Rotations.

1) Left-Rotations: used when nodes is inserted into right subtree [of right subtree] after inserting new node tree becomes unbalanced.

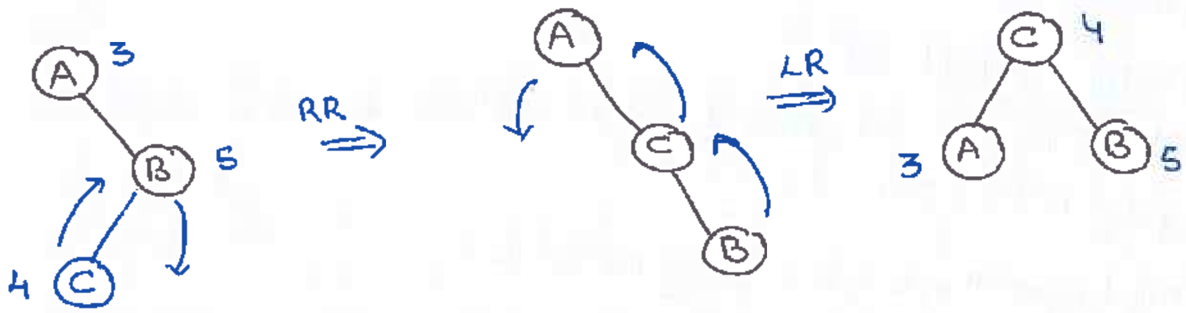


ii) Right Rotation :- node is inserted in the left of left-subtree.

iii) Left-Right Rotation (Double Rotation).
↳ node is inserted in the right of left-subtree and makes the tree unbalanced.



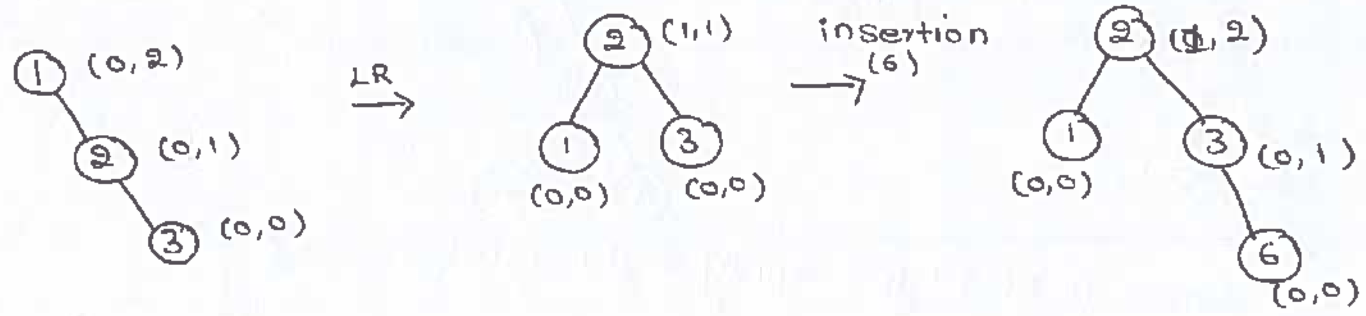
iv) Right-Left Rotation : node is inserted in the left of right subtree and make the tree unbalanced.



* LR ⇒ Left Rotation
RR ⇒ Right Rotation

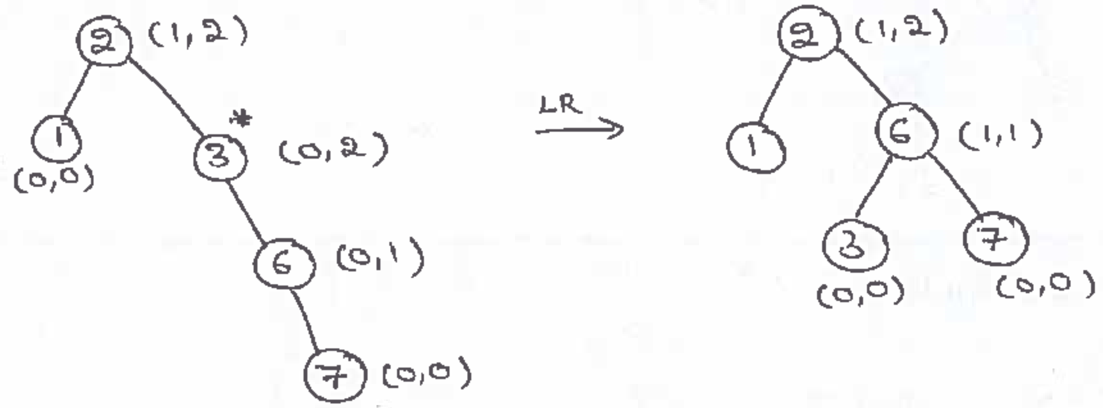
AVL Trees and Insertion :

1, 2, 3, 6, 7.



insertion (7)

5



[As inserting is
or right subtree]

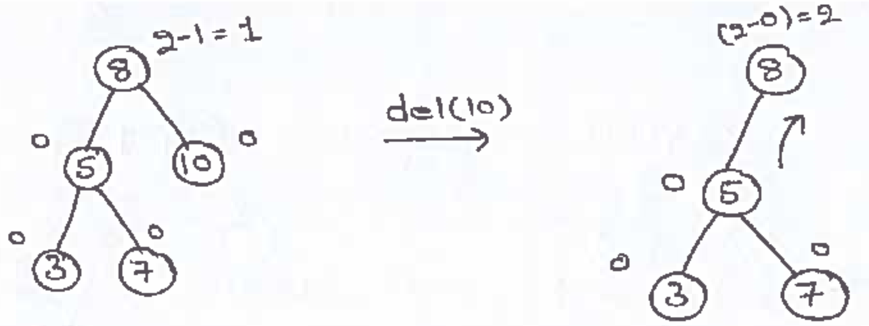
Delete an element from AVL Trees :

A) simple remove the node with all three case of BST.
   i) no - child
   ii) one - child
   iii) Both children available.

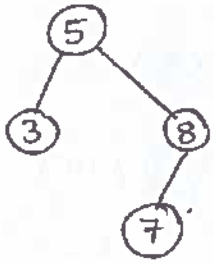
B) calculate B.F → again
C) A is nearest ancestor of deleted node.
D) if deleted node are from left subtree of A then it is called type L
   delete otherwise it is called type R delete.

i) type R.

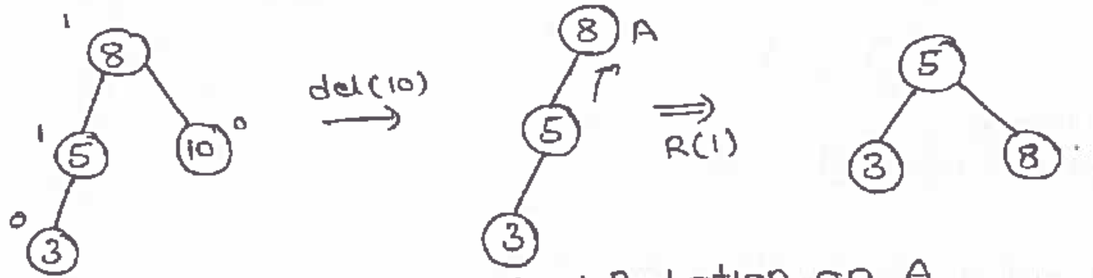
a) R(0)
   R(1)
   R(-1) } Balance of left sybling



Since it R(0) type apply right rotation on A.

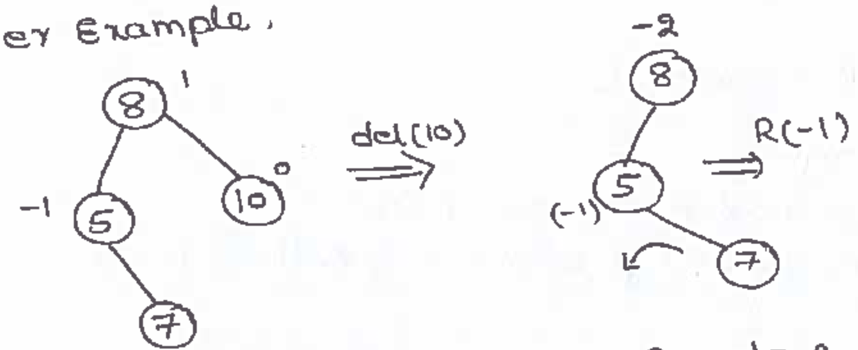


Another Example

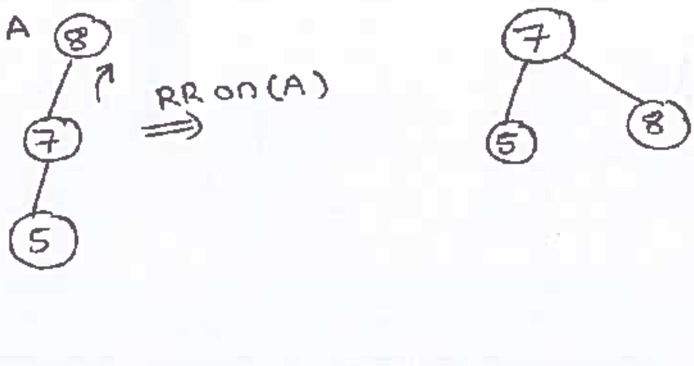


R(1) ⇒ Simple apply Right Rotation on A

Another Example.



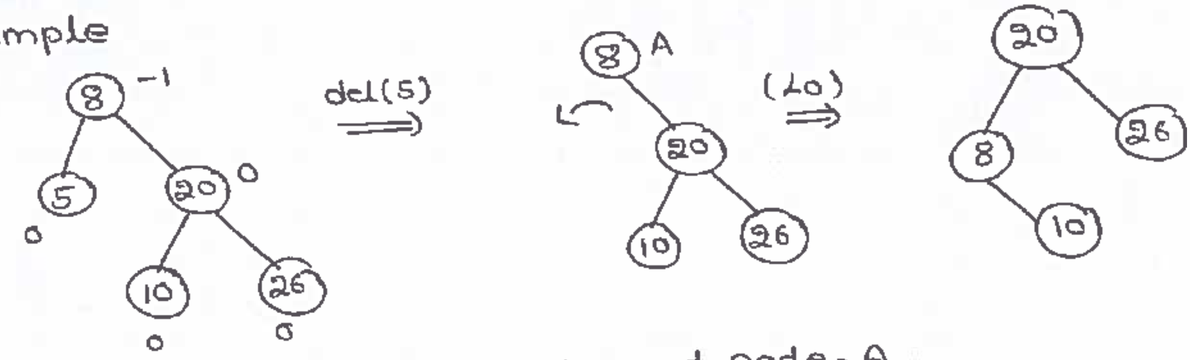
R(-1) ⇒ Left rotation on left child of node A & then Right rotation on node A.



Type - L

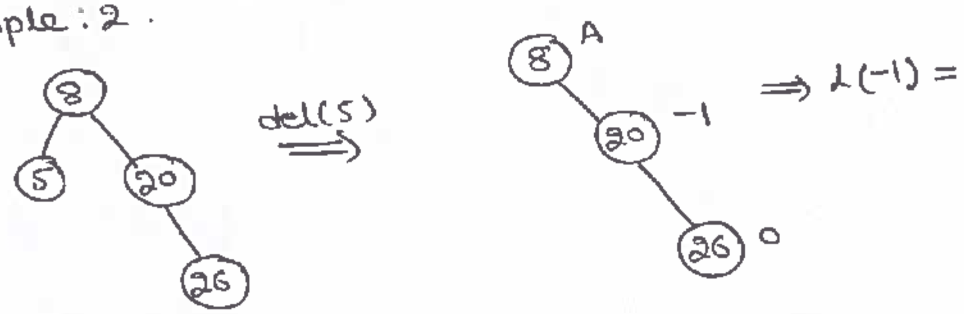
a) L(0)
b) L(1)
c) L(-1)

Example

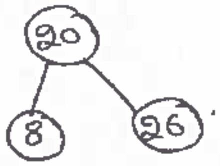


L_0 = case apply left Rotation at node - A .

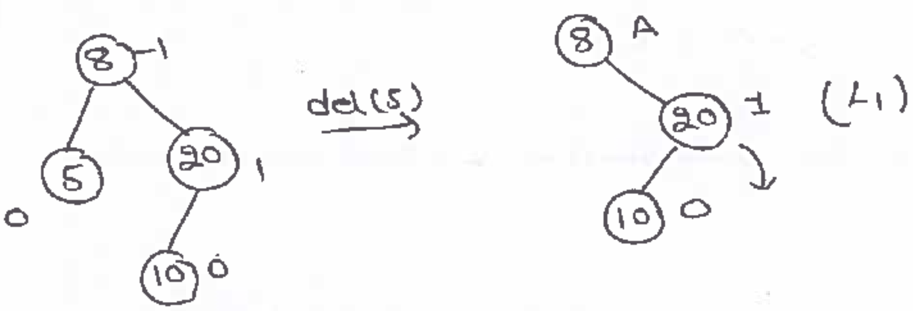
Example : 2 .



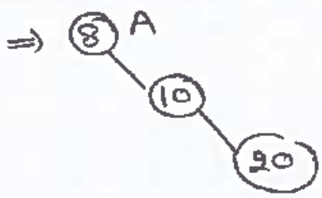
L(-1) = case apply left rotation at node A .



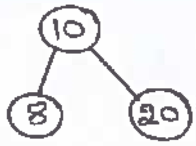
Example on AVL-deletion continues.



This is L(1) type .
in L_1 case we have to solve in two steps,
Step 1 : Right Rotation ~~and~~ at right child of 'A'



Now step 2: left rotation at node A.



Binary Search Trees (BSTs)

Why Binary Search Trees?

12 We have discussed different tree representations and in all of them we did not impose any restriction on the nodes data. As a result, to search for an element we need to check both in left subtree and in right subtree.

Due to this, the worst case complexity of search operation is $O(n)$.

We will discuss another variant of binary trees: Binary Search Trees (BSTs). As the name suggests, the main use of this representation is for searching. In this representation we impose restriction on the kind of data a node can contain.

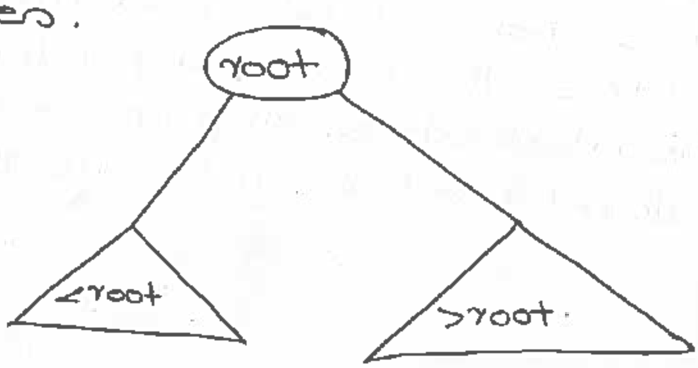
As a result, it reduces the worst case average search operation to $O(\log n)$.

Binary search tree property:

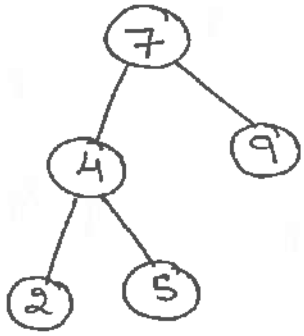
In binary search tree, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data.

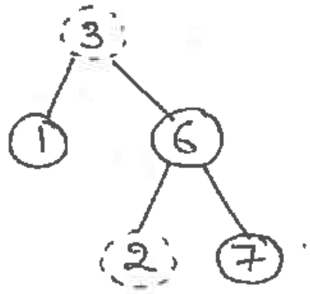
This is called binary search tree property, note that, this property should be satisfied at every node in the tree.

- The left subtree of a node contains only nodes with keys less the nodes key.
- The right subtree of a node contains only nodes with keys greater than the nodes key.
- Both the left and right subtrees must also be binary search trees.



Ex:- The left tree is a binary search tree and right tree is not binary search tree. [at node 6 it's not satisfying the binary search tree property].





Operations on Binary Search trees:-

Main operations: following are the main operations that are supported by binary search trees:

- ✓ find / find minimum / find maximum element in binary search trees.
- ✓ Inserting an element in binary search trees.
- ✓ Deleting an element from binary search trees.

Auxiliary operations:
Checking whether the given tree is a binary search tree or not

- finding k^{th} - smallest element in tree.
- Sorting the element of binary search tree and many more

Important Notes on Binary Search Trees:-

- Since root data is always between left subtree data and right subtree data, performing inorder traversal on binary search tree produces a sorted list.
- while solving problems on binary search trees, first we process left subtree, then root data and finally we process right subtree. This means, depending on the problem only the intermediate step (processing root data) changes and we do not touch the first and third steps.

If we are searching for an element and if the left subtree roots data is less than the element we want to search then skip it. Same is the case with right subtree.. Because of this, binary search trees take less time for searching an element than regular binary trees.

In other words, the binary search trees consider either left or right subtrees for searching an element but not both.

finding an element in Binary Search Trees.

find operation is straightforward in a BST. Start with the root and keep moving left or right using the BST property.

If the data we are searching is same as nodes then we return current nodes. If the data is not present, we end up in a NULL Link.

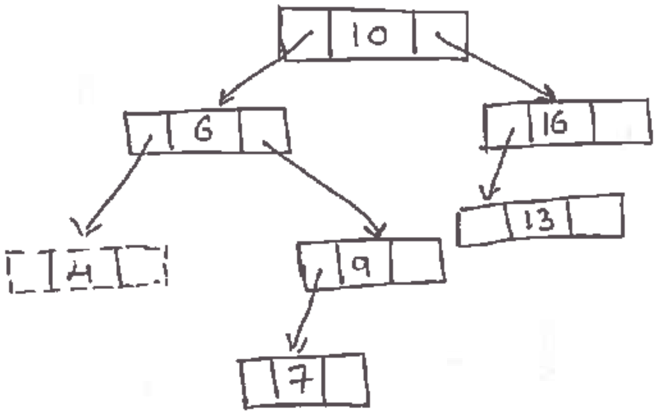
```
struct BinarySearchTreeNode *find(struct BinarySearchTree
                                   node *root, int data) {
    if (root == NULL)
        return NULL;
    if (data < root -> data)
        return find (root -> left, data);
    Else if (data > root -> data)
        return find (root -> right, data);
    return root;
}.
```

Time complexity: O(n), in worst case (when BST is a skew tree).

Space complexity: O(n), for recursive stack.

Finding minimum Element in Binary Search trees:

In BSTs, the minimum element, is the left-most node, which does not has left child. In the BST below, the minimum element is 4.




```
Struct BinarySearchTreenode *findmin(struct BinarySearchTree)
  {
  if(root==NULL)
     return NULL;
  Else if (root→left==NULL)
     return root;
   Else
     return findmin(root→left);
  }
```

Time Complexity: O(n) in the worst case (when BST is a left skew tree).

Space complexity: O(n) for recursive stack.

Finding Maximum Element in Binary Search Trees:-

In BSTs, the maximum element is the right-most node, which does not have right child. In the BST above the maximum element is 16.

```
findmax( ) {.
 if (root==NULL)
    return NULL;
 Else if (root→right==NULL).
    return root;
  else return findMax(root→right);
}.
```

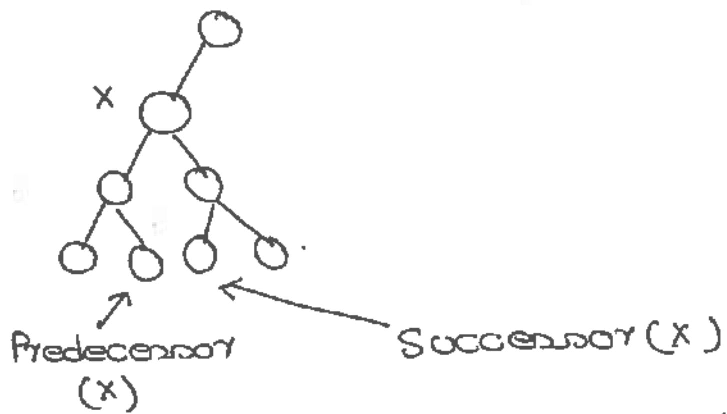
Time complexity: O(n) in the worst case (BST is a right skew)

Space complexity : O(n) for recursive stack.

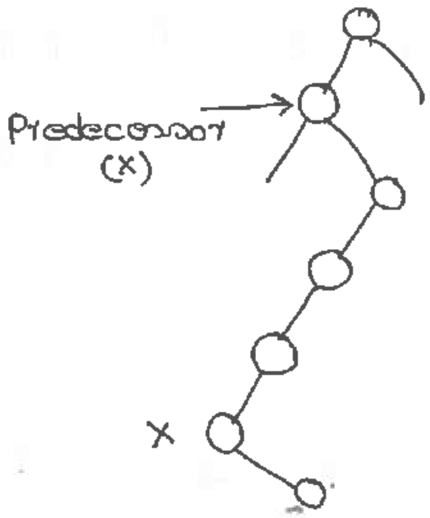
Where is Inorder Predecessor and Successor?

Where is the inorder predecessor and Successor of node x in a binary search tree assuming all Keys are distinct?

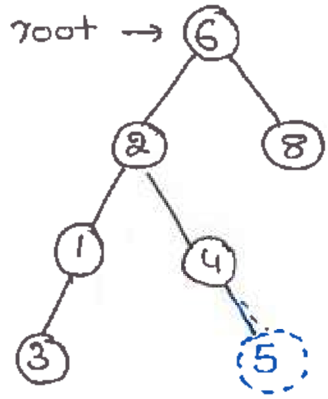
If x has two children then its inorder predecessor is the maximum value in its left subtree and its inorder successor the minimum value in its right subtree.



If it does not have a left child a node inorder predecessor is its first left ancestor.



Inserting an Element from Binary Search Tree:



To insert data into binary search tree, first we need to find the location for that element. We can find the location of insertion by following the same mechanism as that of find operation.

While finding the location if the data is already there then we can simply neglect and come out.

Otherwise, insert data at the last location on the path traversed.

As an example let us consider the above tree. The dotted node indicates the element (5) to be inserted, To insert 5, traverse the tree as using find fonction. At node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct location for insertion.

Time complexity: O(n), ~~if~~ in worst case

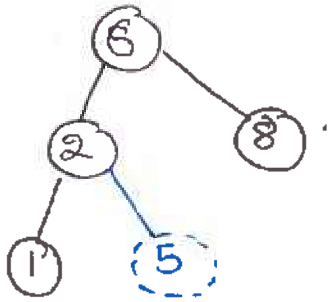
Space complexity: O(n),

Deleting an Element from Binary Search Tree:

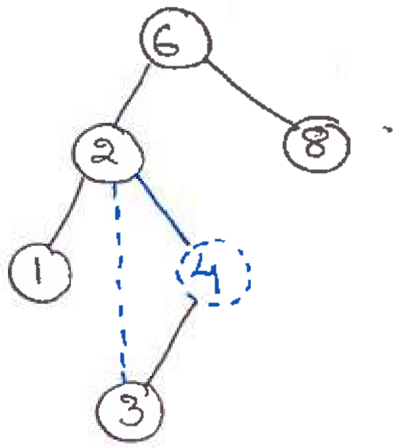
The deleting operation is more complicated than other operations. This is because the element to be deleted may not be the leaf node. In this operation also, first we need to find the location of the element which we want to delete.

Once we have found the node to be deleted, consider the following cases:

- If the element to be deleted is a leaf node: return NULL to its parent. That means make the corresponding child pointer NULL. In the tree below to delete 5, set NULL to its parent node 2.

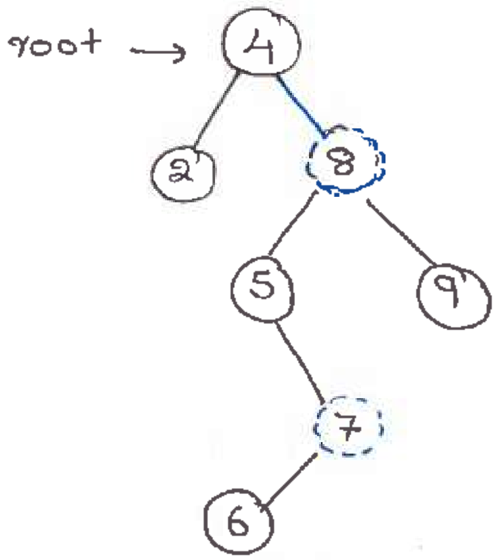


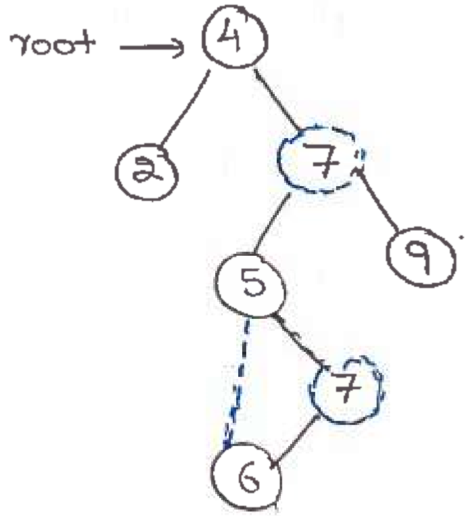
- If the element to be deleted has one child: In this case we just need to send the current node's child to its parent. In the tree below, to delete 4, 4 left subtree is set to its parent node 2.



- if the element to be deleted has both children: The general ~~sta~~ strategy is to replace the key of this node with the largest element of the left subtree and recursively delete the node (which is now empty). The largest node in the left subtree cannot have a right child, the second delete is an easy one. As an example, let us consider the following tree.

In the tree below, to delete 8, it is the right child of root. The key value is 8. It is replaced with the largest key in its left subtree (7), and then that node is deleted as before (second case)





Time complexity: O(n)

Space complexity: O(n) for recursive stack. for Iterative version, space complexity is O(1).

Balanced Binary Search Trees:-

we have seen different trees whose worst case complexity is O(n). where n is the number of nodes in the tree. This happens when the trees are skew trees. In this section we will try to reduce this worst case complexity to O(logn) by imposing restrictions on the heights.

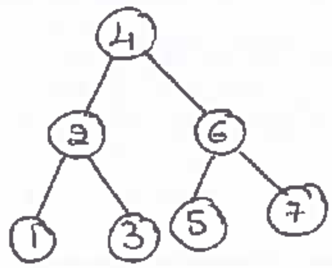
In general, the height balanced trees, are represented with HB(K), where K is the difference between left subtree height and right subtree height. Sometimes K is called balance factor.

Full Balanced Binary Search Trees:

In HB(K), if K = 0 (if Balance factor is zero), then we call such binary search trees as full balanced binary search trees.

That means, in HB(0) binary search tree, the difference between left subtree height and right subtree height should be at most zero.

This ensures that the tree is a full binary tree.



Graph and its representations:

graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u,v) called as edge.

The pair is ordered because (u,v) is not same as (v,u) in case of directed graph (di-graph). The pair of form (u,v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks.

The networks may include paths in a city or telephone network or circuit network.

graphs are also used in social networks like linkedin, facebook. for example, in facebook each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender and locale.

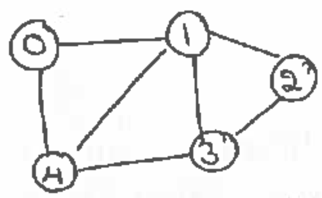
applications of graphs.

→ graphs can be used to model many types of relations and processes in physical, biological, social and information systems.

In computer science, graph are used to represent networks of communication, data organization, computational devices, the flow of computation, etc.,

for instance, the link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another.

following is an example of undirected graph with 5-vertices



175

following two are the most commonly used representations of graph.

1. Adjacency matrix
2. Adjacency list.
3. Edge list [Self-study]

There are other representations also like, ~~and~~ Incidence matrix and Incidence list. The choice of the graph representation is situation specific.

It totally depends on the type of operations to performed and ease of use.

Adjacency matrix

Adjacency matrix is a 2D array of size V×V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.

Adjacency matrix for undirected graph is always symmetric.

Adjacency matrix is also used to represent weighted graphs. if adj[i][j] = w, then there is an edge from vertex i to vertex j. with weight w.

The adjacency matrix for the above example graph is

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Pros:- Representation is easier to implement and follow, Removing an edge takes O(1) time. Queries like whether there is an edges from vertex 'u' to vertex 'v' are efficient and can be done in O(1).

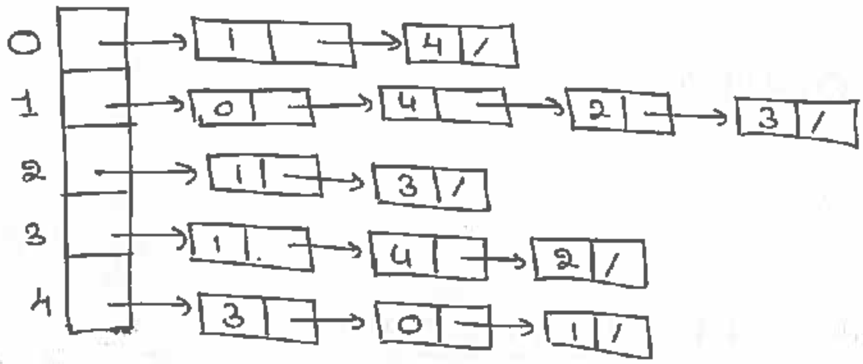
Cons: Consumes more space $O(V^2)$. Even if the graph is sparse (contains less number of edges). It consumes the same space

Adding a vertex is $O(V^2)$ time.

Adjacency Matrix

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ith vertex.

This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



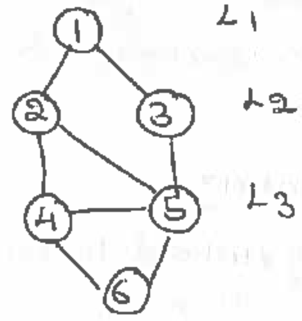
Pros: Saves space $O(|V|+|E|)$. In the worst case, there can be $C(V,2)$ number of edges thus consuming $O(V^2)$ space.

Adding a vertex is easier.

Cons: Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done $O(V)$.

Breadth first traversal or BFS for a graph:-

Idea: Traverse nodes in layer



Problem: Since we have cycles, each node will be visited infinite times.

Solution: use a boolean visited array

we use queues in implementation

	1	2	3	4	5	6
visited:	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1	~~0~~ 1

queue: ~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~.

print: 1, 2, 3, 4, 5, 6

Time complexity: O(V+E)

V: vertices
E: Edges.

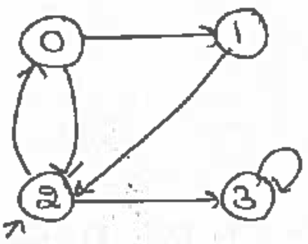
Depth first Traversal or DFS for a graph.

Depth first Traversal (or search) for a graph is similar to Depth first Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array

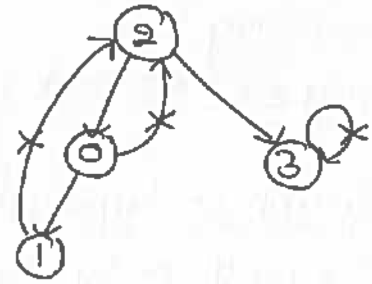
for Example:

in the following graph, we start traversal from vertex 2. when we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't make visited vertices, then 2 will be processed again and it will become a non-terminating process

A Depth first traversal of the following graph is
2, 0, 1, 3.

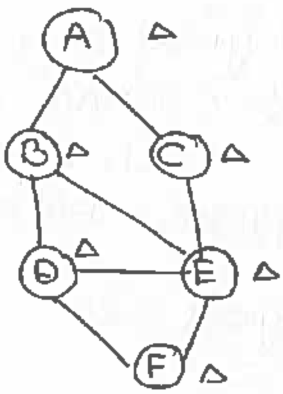


Start



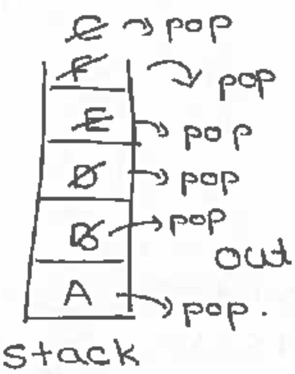
Time complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

Idea: to go forward (in depth) while there is any such possibility, if not then backtrack.



Problem: Since we have cycles, each node may be visited infinite times

Sol use a boolean visited array.



output:- A B D E F C

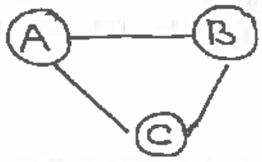
Time complexity: O(V+E).

Depth-first traversal Application

- Detecting cycle in a graph.
- Topological sorting
- finding strongly connected components
- path finding
- To test if a graph is bipartite
- solving puzzles with only one solution, such as mazes.

1) Detecting a cycle.

for every visited vertex 'v'.
if there is an adjacent 'u'
such that u is already visited
and u is not parent of v, then there is a cycle in graph.



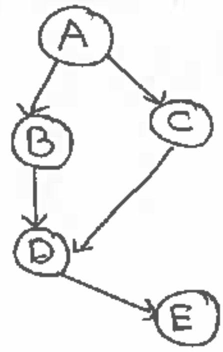
2) Topological sorting: (job scheduling)

for a DAG $G=(V,E)$ is a linear ordering of all its vertices such that if G contains an edges (u,v) then u appears before v in ordering.

Topological sort:

A B C D E

A C B D E

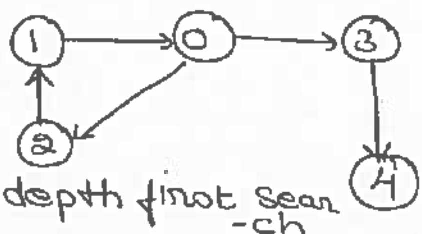


3) Strongly connected component:

A directed graph is strongly connected if there is a path between all pairs of vertices.

A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph.

for example, there are 3 SCC's in the following graph.



Algorithm:

Kosaraju's algorithm: uses two passes of depth first search

Tarjan's algorithm: uses a single pass of depth first search.

A) pathfinding: find a path between two given vertices u and v.

Algorithm :

i) call DFS(G, U) with u as the start vertex.

ii) use a stack S to keep track of the path between the ~~stack~~ start vertex and the current vertex.

iii) As soon as destination vertex v is encountered, return the path as the contents of the stack.

Application of Breadth first Traversal.

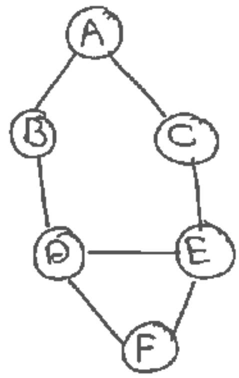
- shortest Path in a graph.
- web Crawler
- social network
- cycle Detection
- To test if a graph is bipartite
- Broadcasting in a network.
- ford-fulkerson algorithm.

1. Shortest path in an unweighted graph.

In unweighted graph, the shortest path is the path with least number of edges.

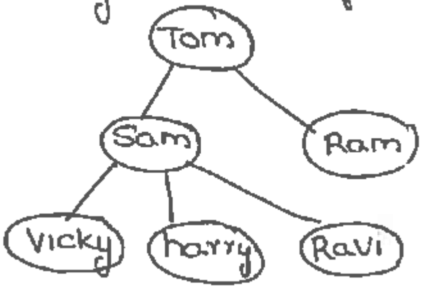
2) web crawler.

The idea is to start from source page and follow all links from source and keep doing same upto the required depth



3) Social network:

In social network, we can find people within a given distance 'k' from a person using Breadth first search till 'k' levels.



181

GPS Navigation systems:

↳ Breadth-first is used to find all neighboring location

Broadcasting in Network

In Networks, a broadcasted packet follows Breadth first search to reach all nodes.

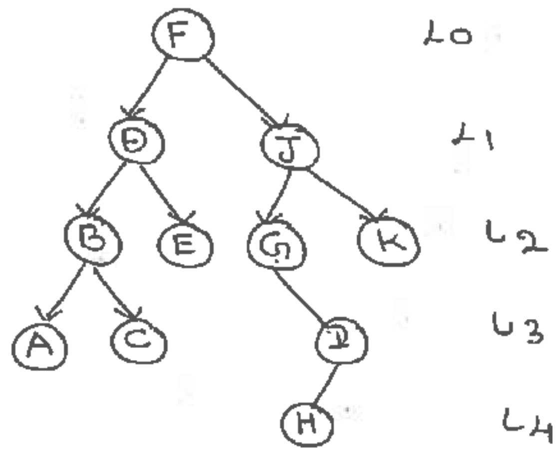
In Garbage Collection:

Breadth first search is used in copying garbage collection using cheney's algorithm

Binary tree traversal. [Recap - Needed].

Tree Traversal.

- → Breadth-first
- → Depth-first



Breadth-first

F, D, J, B, E, G, K, A, C, I, H
(level-order).

Depth-first

- root, left, right → Preorder.
- left, root, right → inorder
- left, right, root → Post order.

Preorder.

F, D, B, A, C, E, J, G, I, H, K.

Inorder:

A, B, C, D, E, F, G, H, I, J, K.

Post-order.

A, C, B, E, D, H, I, G, K, J, F.

183

Introduction:

Let us start our discussion with simple theory that will gives us an understanding of the greedy technique. in the game of chess,

every time we make a decision about a move, we have to also think about the future consequences,

whereas, in the game of Tennis (or Volleyball), our action is based on the immediate situation.

Greedy Strategy

Greedy algorithms work in stages. In each stage, a decision is made that is good at that point, without bothering about the future.

This means that some local best is chosen. It assumes that a local good selection makes for a global optimal Solution.

Elements of Greedy Algorithms

The two basic properties of optimal greedy algorithms are:

1) Greedy choice property
2) optimal substructure.

Greedy choice property

This property says that the globally optimal solution can be obtained by making a locally optimal solution (greedy). The choice made by a greedy algorithm may depend on earlier choices but not on the future. It iteratively makes one greedy choice after another and reduces the given problem to a smaller one.

optimal substructure

A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the subproblems.

That means we can solve subproblems and build up the solutions to solve larger problems.

Does Greedy Always Works?

Making locally optimal choices does not always work. Hence, Greedy Algorithms will not always give the best solutions.
We will see particular in the problems.

Advantages and Disadvantages of Greedy Method:

The main advantages of the Greedy method is that it is straight forward, easy ~~of~~ to understand and easy to code. In greedy algorithms, once we make a decision, we do not have to spend time re-examining the already computed values.
Its main disadvantage is that for many problems there is no greedy algorithm. That means, in many cases there is no guarantee that making locally optimal improvements in a locally optimal solution gives the optimal global solution.

Greedy Applications

- Sorting : Selection sort,
- priority queues : Heap sort
- Huffman coding compression algorithm
- prim's and kruskal's algorithm.
- Shortest path in weighted graph [Dijkstra's].
- coin change problem
- fractional knapsack problem.
- Disjoint sets - UNION by size and UNION by height (or rank)
- job scheduling algorithm.
- Greedy ~~algorithm~~ techniques can be used as an approximation algorithm for complex problems.

Problem 1:

Given an array F with size n. Assume the array content F[i] indicates the length of the i^{th} file and we want to merge all these files into one single file. check whether the following algorithm gives the best solution for this problem or not?

Algorithm: Merge the files contiguously. That means select the first two files and merge them. Then select the output of the previous merge and merge with the third file, and keep going..

Note: Given two files A and B with sizes m and n, the complexity of merging is $O(m+n)$.

Sol: This algorithm will not produce the optimal solution. for a counter example, let us consider the following file sizes array.

$$F = \{10, 5, 100, 50, 20, 15\}.$$

As per the above algorithm, we need to merge the first two files (10 and 5 size files), and as a result we get the following list of files. In the list below, 15 indicates the cost of merging two files with sizes 10 and 5.

$$\{15, 100, 50, 20, 15\}.$$

Similarly, merging 15 with the next file 100 produces: $\{115, 50, 20, 15\}$. for the subsequent steps the list becomes.

$\{165, 20, 15\}$, $\{185, 15\}$

$\{200\}$.

finally,

The total cost of merging = cost of all merging operations

$= 15 + 115 + 165 + 185 + 200$

$= 680$.

To see whether the above result is optimal or not, consider the order: $\{5, 10, 15, 20, 50, 100\}$. for this Example following the same approach, the total cost of merging

$= 15 + 30 + 50 + 100 + 200$

$= 395$

So, the given algorithm is not giving the best (optimal) Solution.

Huffman Coding Algorithm [

- It is a lossless data compression algorithm
- We assign variable-length codes to input characters, length of the assigned codes are based on the frequencies of corresponding characters.

The most frequent character gets the smallest code and the least frequent character gets the largest code.

- The variable-length codes assigned to input characters are Prefix codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character.
- This is How Huffman coding makes sure that there is no ambiguity when decoding the generated bit stream.
- Let us understand prefix codes with a counter example. Let there be four characters a, b, c, and d and their corresponding variable length codes be 00, 01, 0 and 1

This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. if the Compressed bit stream is 0001, the decompressed output may be "cccd" or "ccb" or "acd" or "ab".

- Application of Huffman coding

Huffman is widely used in all the mainstream compression formats that you might encounter - from GZIP, PKZIP (winzip etc) and BZIP2, to image formats such as JPEG and PNG.

* Brotli compression, released by google uses Huffman coding. Apart from that, Brotli also uses LZ77 & few other fundamental lossless compression

There are mainly two major parts in Huffman coding.

1) Build a Huffman tree from input characters.
2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman tree.

Input is array of unique characters along with their frequency of occurrences and output is Huffman tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (min heap is used as a priority queue. The value of frequency field is used to compare two nodes in min-heap. Initially, the least frequent character is at root)

2) Extract two nodes with the minimum frequency from the min-heap.

3) Create a new internal node with frequency equal to the sum of the two nodes frequencies make the first extracted nodes as its left child and the other extracted node as its right child. Add this node to the min-heap.

4) Repeat Steps #2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Huffman Coding Example

THE ESSENTIAL FEATURE [12 symbol].

A	2
B	
C	
D	
E	5
F	1
G	
H	1
I	1
J	
K	
L	1
M	
N	1
O	
P	
Q	
R	1
S	2
T	3
U	1
V	
W	
X	
Y	
Z	
_	2

The ␣ Essential ␣ FEATURE

↳ 66 binary number using Huffman.

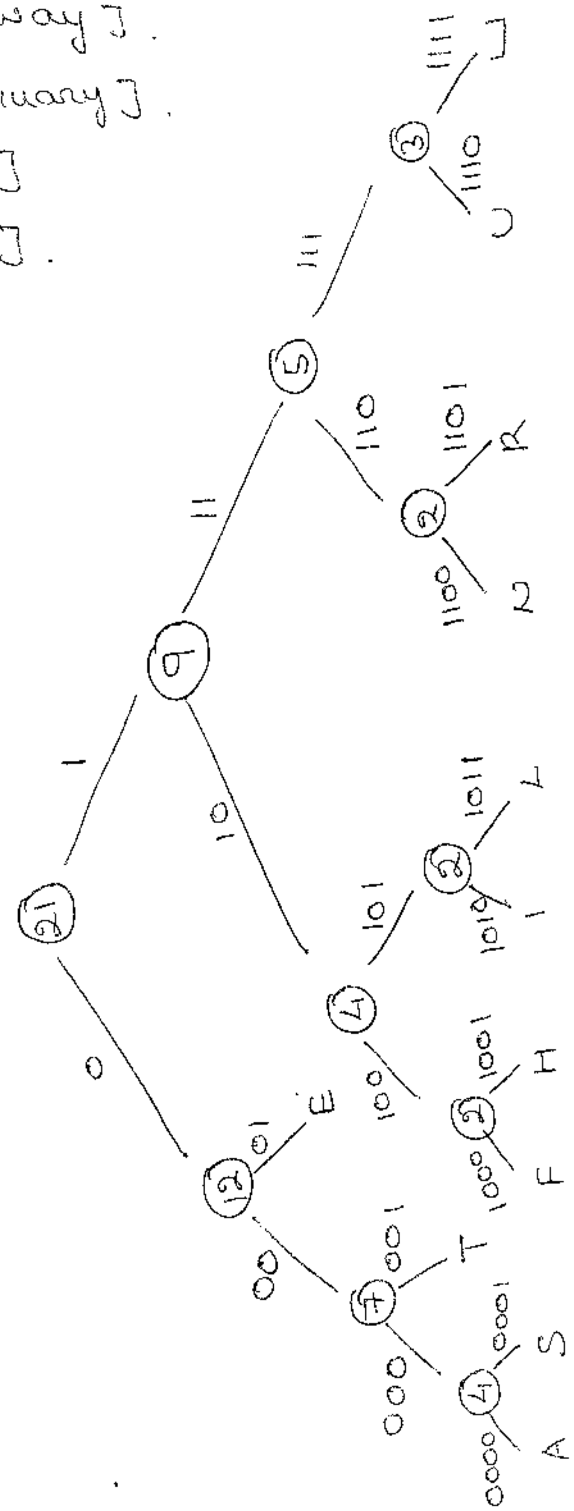
~~Ex~~

using ~~ascii~~ ascii value. [Naive way].

$21 \cdot 8 = 168$ [using 8-~~binary~~/bit binary].

$21 \cdot 5 = 105$ [only 5-bit binary]

$21 \cdot 4 = 84$ [only 4-bit binary].



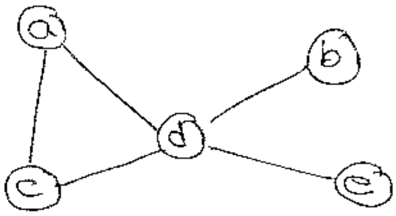
Minimal Spanning Tree:

The spanning tree of a graph is a subgraph, that contain all the vertices and is also a tree [which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be dis connected].

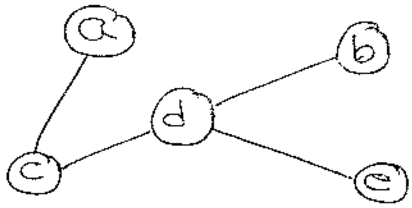
By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree.

A disconnected graph does not have any spanning tree, as it cannot be spanned to all its verticies

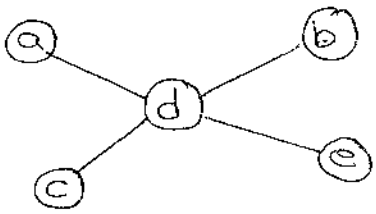
Example



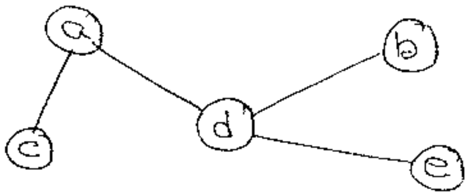
graph.



Spanning tree - 1.



Spanning tree - 2.



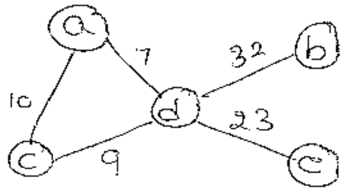
Spanning tree - 3.

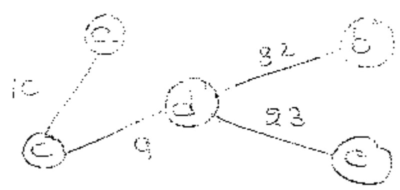
Weighted graphs

weighted graphs is a graph, in which each edge has a weight (some real number).

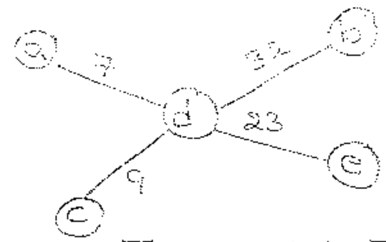
~~weighted~~ weight of a graph: The sum of the weights of all edges.

Example:

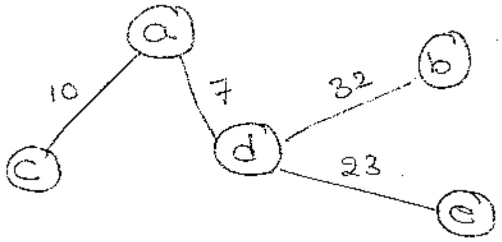




Tree 1. W = 74



Tree 2, W = 71.



Tree 3, W = 72.

Minimum spanning tree.

* A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. ~~In the above addressed example~~,

$\cancel{2\ 5^{5-2} = 5^{3}}$ $\cancel{2^{n}}$

General Properties of spanning Tree

we now understand that one graph can have more than one spanning tree. following are a few properties of the spanning tree connected to graph G.

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e the spanning tree is minimally connected.

↳ Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic

Application of Spanning Tree :-

Spanning tree is basically used to find a minimum path to connect all nodes in a graph.

Common application of spanning trees are.

↳ Civil network planning

↳ Computer Network Routing protocol.

↳ Cluster Analysis.

Let us ~~&~~ understand this through a small example. Consider City network as a hugh graph and ~~have~~ now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes This is where the spanning tree comes into picture.

Minimum Spanning-Tree Algorithm

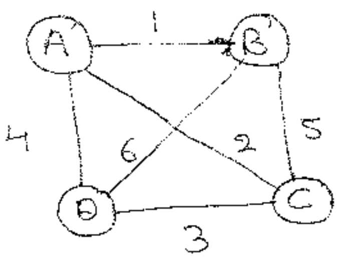
We shall learn about two most important spanning tree algorithms here.

↳ Kruskal's algorithm } Both are greedy algorithms.

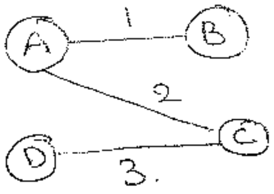
↳ Prim's algorithm

Prim's algorithm

It is an algorithm used to find a minimum cost spanning for connected weighted undirected graph.



$\Longrightarrow$

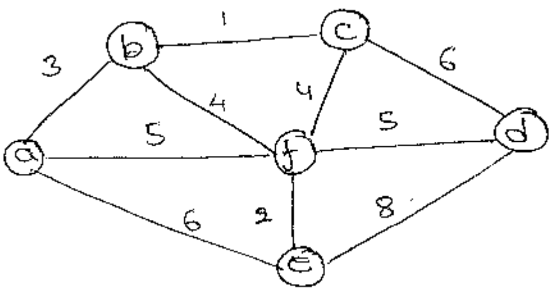


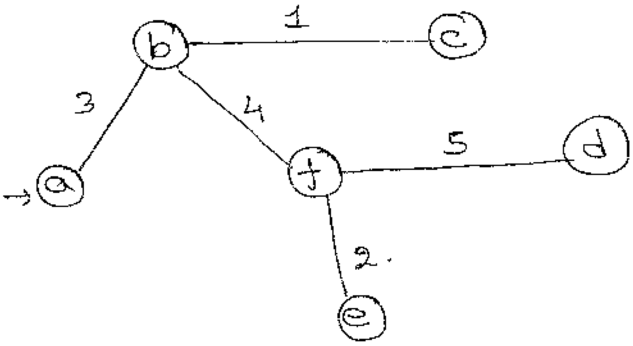
If we use Brute force way:

there are 16 possibilities, list out all possibilities and choose the smallest.

$$2^n = 2^4 = 16$$

Prim's Algorithm Example





Algorithm Prim (G).

$V_t \leftarrow \{v_0\}$ // set of visited vertices

$E_t \leftarrow \phi$

for $i \leftarrow 1$ to $|V|-1$ do

find minimum edge e between

vertices v and u such that v is in V_t and u is in $V - V_t$.

//Add u to V_t.

$V_t \leftarrow V_t \cup \{u\}$

// Add the edge to the spanning tree.

$E_t \leftarrow E_t \cup \{e\}$.

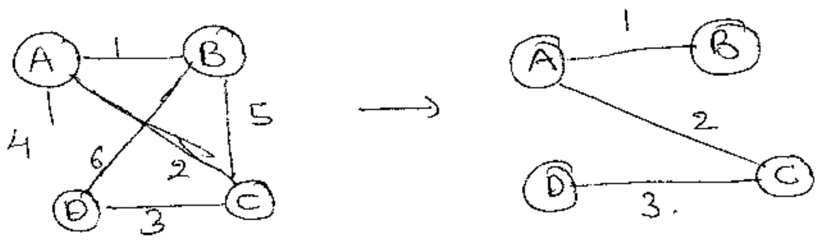
Time complexity

1) if the graph is represented as an adjacency matrix then the complexity of prims algorithm is V^2.

↳ If you use binary heap and adjacency list the complexity can be of the order of $E \log V$.

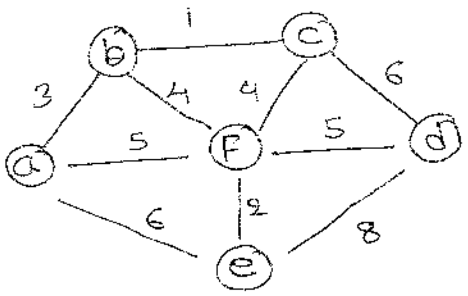
Krushkal's Algorithm

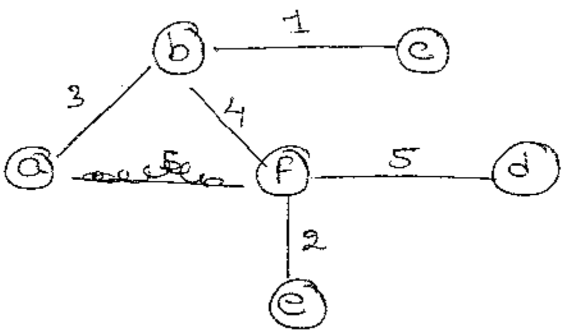
↳ It is an algorithm used to find a minimum cost spanning tree for connected weighted undirected graph.



Kruskal's Algorithm in real life.

No Loop formation





Kruskal's Algorithm

Sort E in ascending order of weights

$E_t \leftarrow 0$ // no Edge is selected.

encounter $\leftarrow$ 0 // no ~~edge~~ of edges selected.

$K \leftarrow 0$ → Variable

while encounter $< |V| - 1$ → till all the edges are connect

$K \leftarrow K + 1$

If $E_t \cup \{e_{ik}\}$ is acyclic

$E_t \leftarrow E_t \cup \{e_{ik}\}$

encounter += 1

return E_t.

Heart of the algo

Time complexity

* if the graph is represented as an adjacency matrix then the complexity of Kruskal algorithm is V^2.

* If you ~~are~~ use binary heap and adjacency list the complexity can be of the order of $E \log V$.

Guidelines for Asymptotic Analysis

There are some general rules to help us determine the running time of an algorithm.

1) **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of ~~it~~ iterations

```
// Executes n-times.
for (i=1; i<=n; i++)
```

Total time = a constant $c \times n = cn = O(n)$

2) **Nested Loops**: Analyze from inside out. Total running time is the product of the sizes of all the loops

```
// outer loop executed n times.
for (i=1; i<=n; i++) {
  // inner loop executed n-times.
  for (j=1; j<=n; j++).
      k=k+1;  // constant time.
}.
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

3) **Consecutive statements**

Add the time complexities of each statement.

```
x=x+1; // constant time.
  // executed n-times.
for (i=1; i<=n; i++).
    m=m+2; // constant time
// outer loop executed n-times.
  for (i=1; i<=n; i++) {.
// inner loop executed n-times.
  for (j=1; j<=n; j++ ).
      k=k+1; // constant time
}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$

4) If-then-else statements:

worst-case running time: the test, plus either the then part or the else part (which ever is the larger).

```
//test: constant
  if (length () == 0) {
    return false; // then part (constant + constant)*n
   for (int n=0; n< length(); n++) {
  //another if: Constant + constant (no else part)
      if (! list [n].equals (other list. list [n]))
           // constant
         return false;
      }
    }
```

Total number: $c_0 + c_1 + (c_2 + c_3)n = O(n)$.

5) Logarithmic complexity

An algorithm is $O(\log n)$ if it takes a constant time to cut the problem size by a fraction (usually by 1/2).

As an example let us consider the following program:

```
for (i=1; i<=n;) ~~i=i*2;~~
      i = i*2;
```

If we observe carefully, the value of i is doubling every time initially i=1 in next step i=2, and in subsequent steps i=4, 8 and so on.

Let us assume that the loop is executing some k-times

At k^{th} step $2^k = n$ & we come out of loop

Taking logarithm on both sides, gives

$\log(2^k) = \log n$

$k \log 2 = \log n$.

$k = \log n$ // if we assume base ~2.

Total number = $O(\log n)$.

Problem 1

what is the complexity of the program given below

```
void function (int n) {
    int i, j, k, count = 0;
    for (i = n/2; i <= n; i++)
    for (j = 1; j + n/2 <= n; j = j++)
        for (k = 1; k <= n; k = k*2)
            count++;
}.
```

Sol) consider the comments in the following function

```
void function (int n) {
    int i, j, k, count = 0;
    //outer loop executes n-times or n/2.
     for (i = n/2; i <= n; i++)
      // middle loop executes n/2 or n-times.
     for (j = 1; j+n/2 <= n; j = j++)
       //outer loop execute logn times.
         for (k = 1; k <= n; k = k*2).
             count ++;
       }.
```

The complexity of the above function is $O(n^2 \log n)$

Problem-2.

```
void function (int n) {
    int i, j, k, count = 0;
     for (i = n/2; i <= n; i++).
        for (j = 1; j <= n; j = 2*j)
           for (k = 1; k <= n; k = k*2).
               count ++;
   }.
```

Sol Consider the comments in the following function

```
Void function (int n) {.
  int i,j,K, count=0;
  // outer loop execute n/2 times.
  for (i=n/2; i<=n; i++)
    // middle loop executes logn times
    for (j=1; j<=n; j=2*j)
      // outer loop executes logn times
      for (K=1; K<=n; K=K*2)
        count++;
}.
```

The complexity of the above function is $O(n\log^2 n)$.

Problem-3.

find the complexity of the program.

```
function (int n) {
  if (n==1) return;
  for (int i=1; i<=n; i++) {
    for (int j=1; j<=n; j++) {.
      printf ("*");
      break;
    }
  }
}.
```

Sol Consider the comments in the function below.

```
function (int n) {
  // constant time.
  if (n==1) return;
  // outer loop execute n times.
  for (int i=1; i<=n; i++) {
    // inner loop executes only time due to break
                          statement
    for (int j=1; j<=n; j++) {
      printf("*");
      break;
  }
}
```

Problem-4

find the complexity of the recurrence

$$T(n) = \begin{cases} 2T(n-1)-1, & \text{if } n>0 \\ 1, & \text{otherwise} \end{cases}$$

Sol: Let us try solving this function with substitution.

$$T(n) = 2T(n-1) - 1.$$

$$= 2(2T(n-2)-1) - 1.$$

$$= 2^2 T(n-2) - 2 - 1.$$

$$= 2^2[2T(n-3)-1] - 2 - 1.$$

$$= 2^3 T(n-3) - 2^2 - 2^1 - 1.$$

$$= 2^3[2T(n-4)-1] - 2^2 - 2^1 - 2^0$$

$$= 2^4 T(n-4) - 2^3 - 2^2 - 2^1 - 2^0.$$

$$T(n) = 2^n T(n-n) - 2^{n-1} - 2^{n-2} \ldots 2^2 - 2^1 - 2^0.$$

$$= 2^n - 2^{n-1} - 2^{n-2} \ldots 2^2 - 2^1 - 2^0$$

$$= 2^n - [2^n - 1]. \quad [\text{Note: } 2^{n-1} + 2^{n-2} + \ldots + 2^0 = 2^n - 1].$$

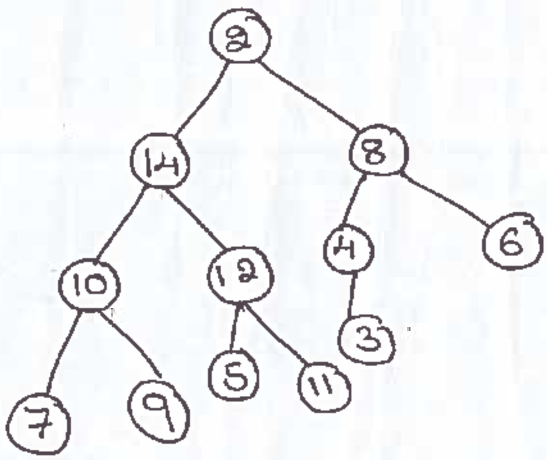
$$T(n) = 1$$

Time complexity is O(1). Note that while the recurrence relation looks exponential the solution to the recurrence relation here gives a different result.

Heapification

- Before discussing the method for building heap of an arbitrary complete binary tree, we discuss a simpler problem.
- Let us consider a binary tree in which left and right subtrees of the root satisfy the heap property but not the root.

See the following fig



- Now the question is how to transform the above tree into a heap?
- Swap the root and left child of root, to make the root satisfy the heap property.
- then check the subtree rooted at left child of the root is heap or not. 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.
- The following sequence of figures depicts the heapification process.

7

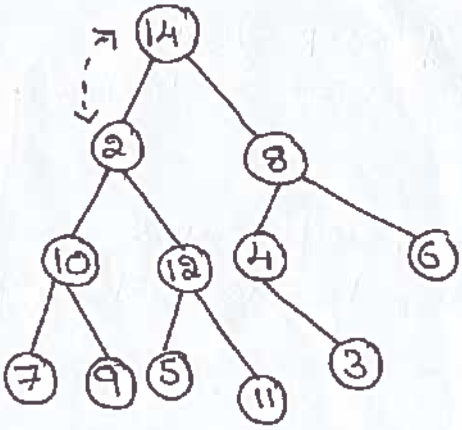


fig-1

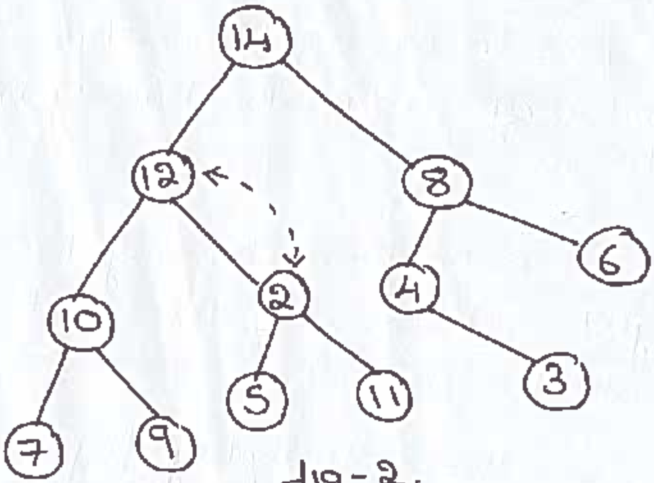
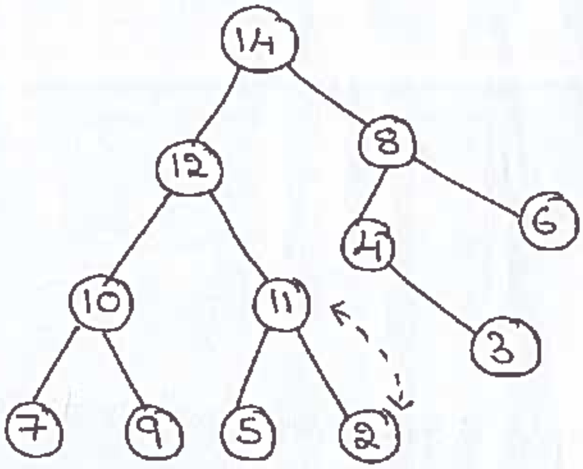


fig-2.



algorithm : Heapification (a, i, n)

Step 1: $left = 2^i$

Step 2: $right = 2^i + 1$

Step 3: if $(left < n)$ and $(a[left] > a[i])$ then

Step 4: maximum = left

Step 5: else

Step 6: maximum = i

Step 7: if $(right < n)$ and $(a[right] > a[maximum])$ then

Step 8: maximum = right

Step 9: if (maximum != i) then

Step 10: swap (a[i], a[maximum])

Step 11: heapfication (a, maximum, n)

- The time complexity of heapification is $O(\log n)$

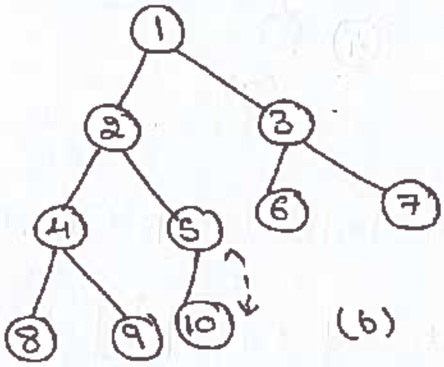
Build Heap

- Heap building can be done efficiently with bottom up fashion
- given an arbitrary complete binary tree, we can assume each leaf is a heap.
- Start building the heap from the parents of these leaves. i.e., heapify subtrees rooted at the parents of leaves.
- Then heapify subtrees rooted at their parents. continue this process till we reach the root of the tree.

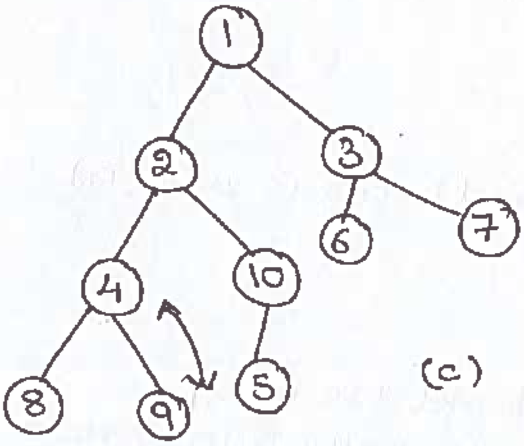
The following sequence of the figures illustrates the build heap procedure:

1	2	3	4	5	6	7	8	9	10

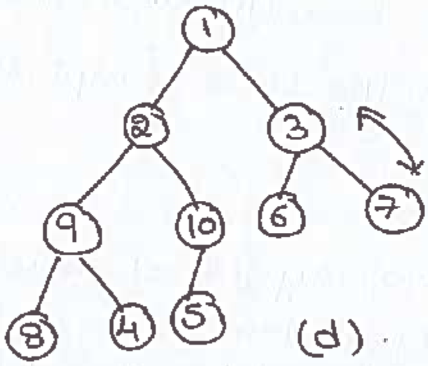
(a)



(b)

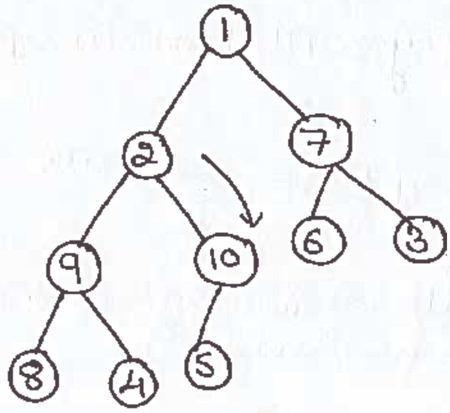


(c)

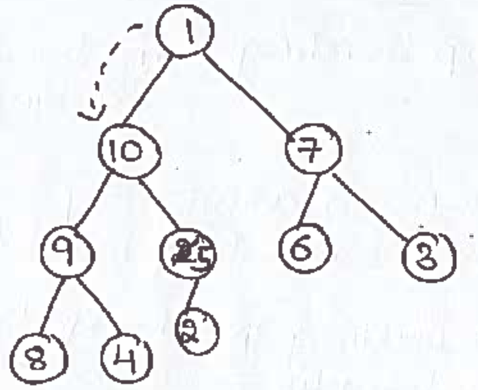


(d).

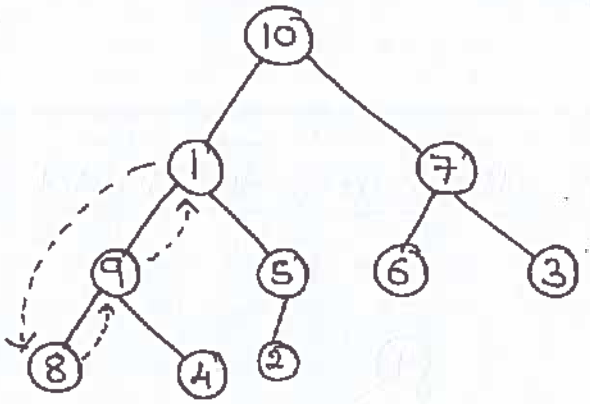
9



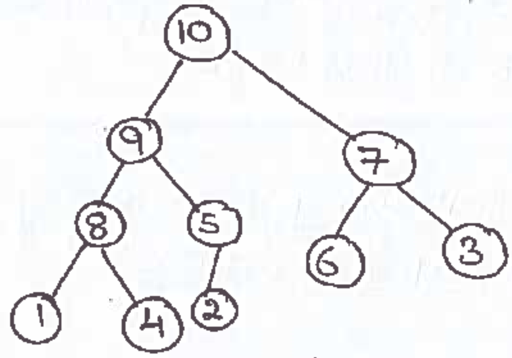
(e)



(f)



(g)



(h)

algorithm: build-heap (a, i, n)

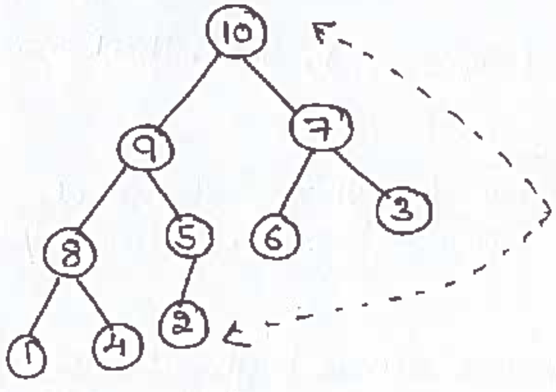
step 1: for $j = \lfloor \frac{i}{2} \rfloor$ down to 1 do.

step 2: heapification (a, j, n)

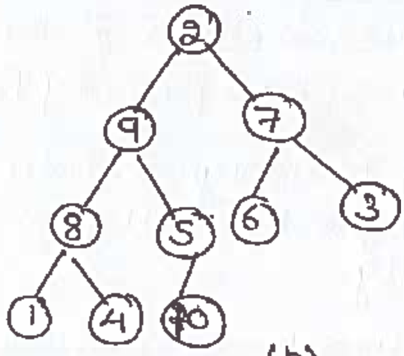
- The time complexity of the build heap is in $O(n)$

Heap sort

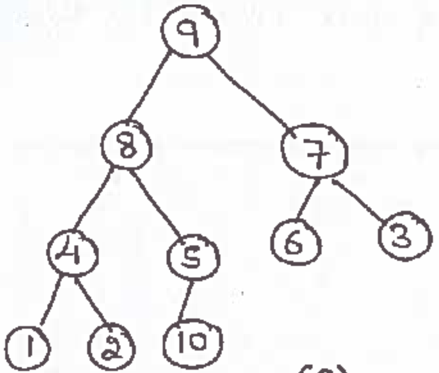
- given an array of n element, first we build the heap
- the largest element is at the root, but its position in sorted array should be at least. So, swap the root with the last
- we have placed the highest element in its correct position. we left with an array of n-1 elements. repeat the same of these remaining n-1 elements to place the next largest element in its correct position.
- Repeat the above step till all elements are placed in their correct positions.



(a)



(b)



(c)

10	9	7	8	5	6	3	1	4	2

(d)

2	9	7	8	5	6	3	1	4	10

Heapification

* pseudocode of the algorithm is given below.

<u>Algorithm Heap_Sort(a, i)</u>

```
build-heap (a, i)
   for j=1 down to 1 do
Swap (a[1], a[j])
 heapification (a, 1, j-1)
```

* The time complexity of the heap sort algorithm is in $O(n\log n)$.

Priority Queue

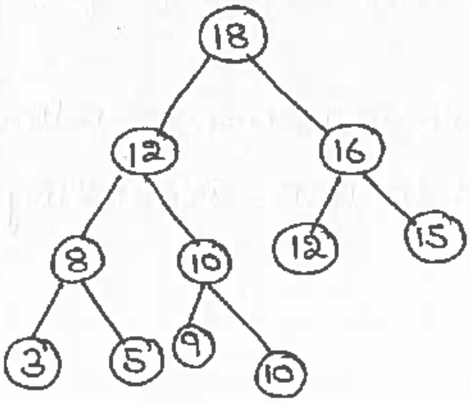
- Let consider a set S of elements ($S_1, S_2, \ldots$), such that each element S_i has priority P_i
- We want to design a data structure for these elements such that the highest priority element should be extracted/deleted efficiently.
- We can use heap for this purpose since highest element is always at the root, which can be extracted quickly.
- Pseudocode for extracting the maximum from the priority queue P is given below.
  Let the global variable size maintains the number of elements in P.

Algorithm : Max_Extract (P).

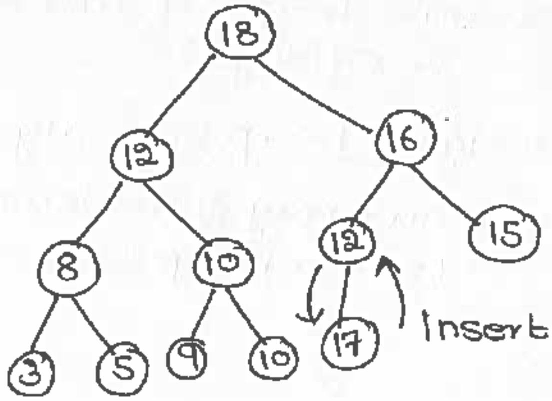
```
max = P[1]
P[1] = P[size]
size = size - 1
heapification (P, 1, size)
return (max)
```

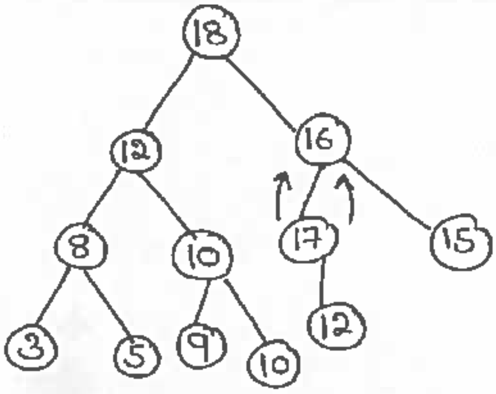
- After extracting the maximum, we have to maintain remaining elements in the priority queue. So we heapify before returning the maximum.
- Other operation to be supported is to insert an element into the priority queue.
- Inserting an element into the priority queue can be done easily.
- Inserting the new element as a new leaf, and push this up till it satisfies the heap property.

The following sequence of fig illustrates the inserting procedure.

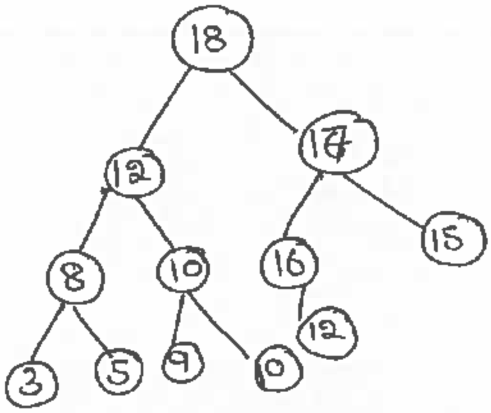


(a)





(c)



final heap.

The pseudo code is given below

Algorithm : Insert (P, x)

size = size + 1

i = size

while $(i > 1)$ and $(x > P[\lfloor \frac{i}{2} \rfloor])$ do

$P[i] = P[\lfloor \frac{i}{2} \rfloor]$

$i = \lfloor \frac{i}{2} \rfloor$

$P[i] = x$

13

Try/Solve the following problems/questions

1. write a program for heap sort.
2. Describe heapsort and show that its worst case performance is $O(n \log n)$
3. Design a heap sort algorithm to sort in non-ascending order.
4. given an array of n-elements sorted in non-ascending order. Is it a heap?

Introduction to tree.

What is a tree?

A tree is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes.

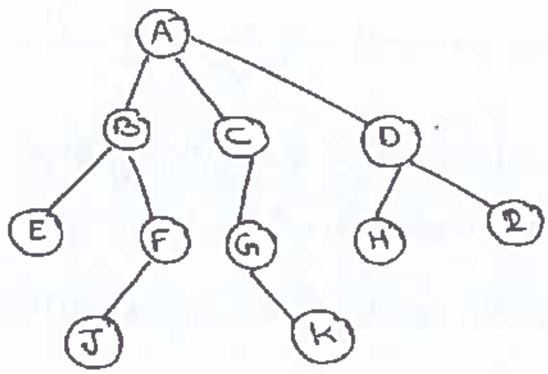
Tree is an example of a non-linear data structure.

A tree structure is a way of representing the hierarchical nature of a structure in a graphical form.

In trees ADT (Abstract Data Type), the order of the elements is not important.

If we need ordering information linear data structures like linked lists, stacks, queues etc. can be used.

Glossary



→ The root of a tree is the node with no parents. There can be at most one root node in a tree (node A in the above Example)

→ An edge refers to the link from parent to child (all links in the figure).

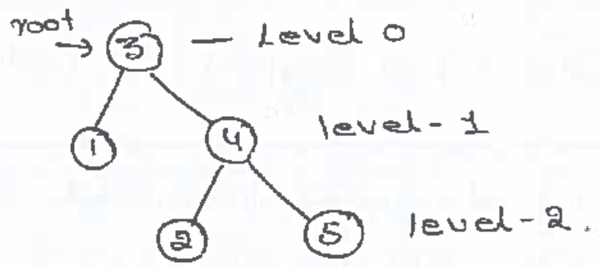
→ A node with no-children is called leaf node (E, J, K, H and I)

→ children of same parent are called siblings (B, C, D are the siblings of A, and E, F are the siblings of B).

- A node p is an ancestor of node q if there exists a path from root to q and p appears on the path.

  The node q is called a descendant of p for example A, C and G are the ancestors of K.

- The set of all nodes at a given depth is called the level of the tree (B, C and D are the same level). The root node is at level zero.



- The <u>depth</u> of a node is the length of the path from the root to the node (depth of G is 2, A-C-G).

- The height of a node is the length of the path from that node to the deepest node in the tree.

  A (rooted) tree with only one node (the root) has a height of zero.

  Ex: the height of B is 2 (B-F-J)

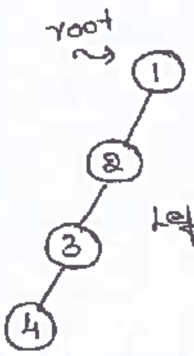
- Height of the tree is the maximum height among all the nodes in the tree and depth of the tree is the maximum depth among all the nodes in the tree.

  for a given tree, depth and height returns the same value But for individual nodes we may get different results.

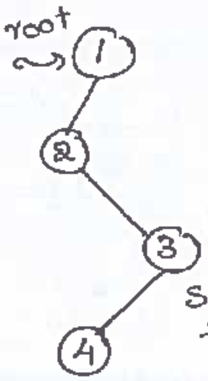
- If every node ~~has a~~ in a tree has only one child (except leaf nodes) then we call such trees <u>skew trees</u>

If every node has only left child then we call them <u>left skew trees</u>

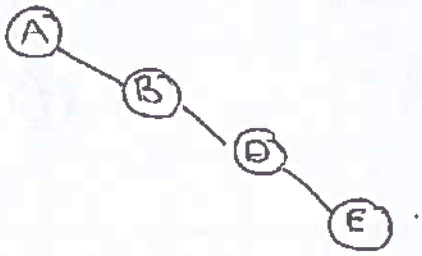
Similarly, if every node has only right child then we call them <u>right skew trees</u>.



Left skew tree



Skew tree.



Right skew tree.

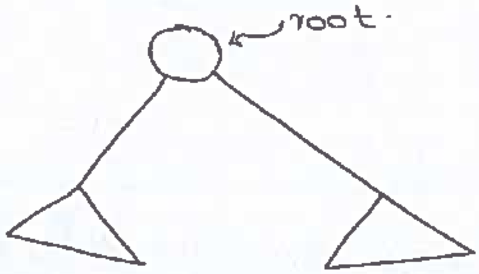
Binary trees

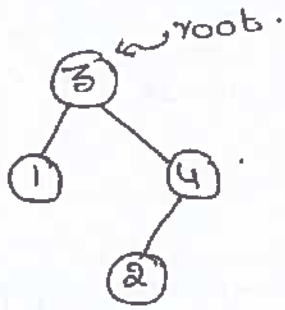
A tree is called binary tree if each node has zero child, one child or two children

Empty tree is also a valid binary tree.

we can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.

generic Binary tree

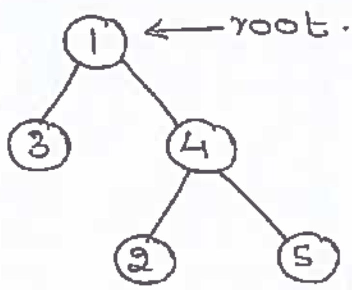




Type of binary trees:

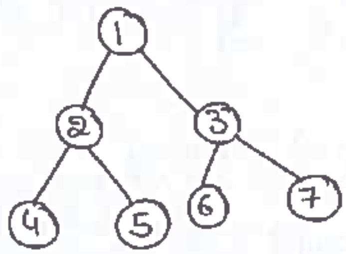
Strict Binary tree:

A binary tree is called strict binary tree if each node has exactly two children or no children

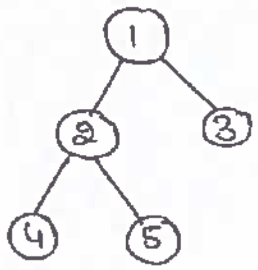


Full binary tree:

A binary tree is called full binary tree if each node has exactly two children and all leaf nodes are at the same level

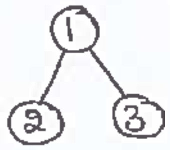
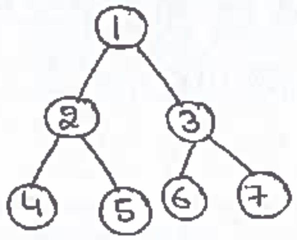


Complete Binary tree



Properties of Binary trees:

for the following properties, let us assume that the height of the tree is h. Also, assume that root node is at height

	height	number of nodes at level h
	$h=0$	$2^0 = 1$.
	$h=1$	$2^1 = 2$.
	$h=2$	$2^2 = 4$.

from the diagram we can infer the following properties:

↳ The number of nodes n in a full binary tree is $2^{h+1}-1$ since, there are h levels we need to add all nodes at each level

$$[2^0+2^1+2^2+\cdots+2^h = 2^{h+1}-1]$$

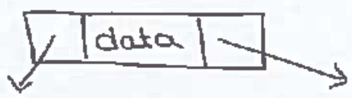
↳ The number of nodes n in a complete binary tree is between 2^h (minimum) and $2^{h+1}-1$ (maximum)

↳ The number of leaf nodes in a full binary is 2^h.

↳ The number of NULL links (wasted pointers) in a complete binary tree of n-nodes is $n+1$.

Structure of Binary Trees

Now let us define structure of the binary tree. for simplicity, assume that the data of the nodes are integers. one way to represent a node (which contain data) is to have two links which point to left and right children along with data fields as shown below:



Note: In trees, the default flow is from parent to children and it is not mandatory to show directed branches for our discussion.

Operations on Binary Trees.

Basic operations
- Inserting
- Deleting
- Searching
- Traversing

Auxiliary operations
- finding the size of the tree.
- finding the height of the tree
- finding maximum sum.

Applications of Binary trees:

following are the some of the applications where binary trees play an important role.

- Expression trees are used in compilers
- Huffman coding trees that are used in data compression algorithms.
- Binary search Tree (BST), which supports search, insertion and deletion on a collection of items in O(log n) [average].

Traversals.

A Traversal is a process that visits all the nodes in the tree.

Since a tree is a non-linear data structure, there is no unique traversal

we will consider several traversal algorithms with we group in the following two kinds.

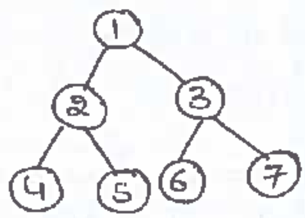
- depth-first traversal
- breadth-first traversal

There are three different types of depth-first traversals:

↳ pre order traversal: Visit the parent first & then left and right children.

↳ In-order traversal: Visit the left child, then the parent and the right child.

↳ post-order traversal: Visit left child, then the right child and then the parent.



Binary tree traversals.

↳ preorder traversal. (Root, left, Right).

→ Visit the root.

→ Traverse the left subtree in preorder.

→ Traverse the right subtree in pre-order.

Ex. 1, 2, 4, 5, 3, 6, 7.

↳ In-Order traversal (Left, Root, Right).

→ Traverse the left subtree in inorder

→ Visit the root

→ Traverse the right subtree in In-order

Ex: 4, 2, 5, 1, 6, 3, 7.

↳ Post-order traversal (left, right, root).

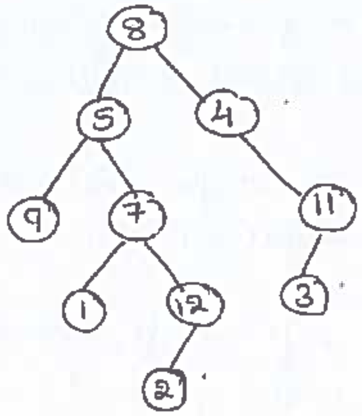
→ Traverse the left subtree in post-order.

→ Traverse the right subtree in post-order.

→ Visit the root.

Ex: 4, 5, 2, 6, 7, 3, 1.

Problem 1:



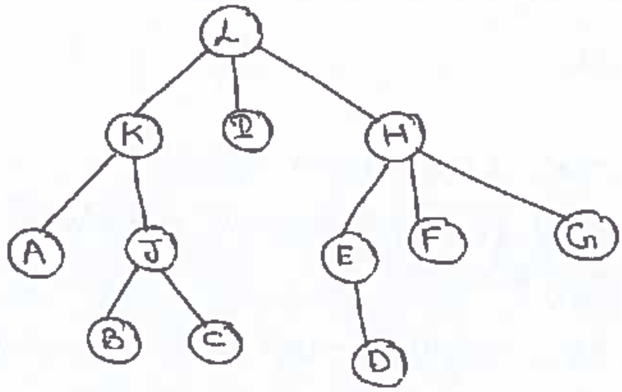
Pre-order: 8, 5, 9, 7, 1, 12, 2, 4, 11, 3.

Inorder: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

Postorder: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

level order: 8, 5, 4, 9, 7, 11, 1, 12, 3, 2.

Problem 2:



Preorder: L, K, A, J, B, C, I, H, E, D, F, G

Postorder: A, ~~B, C, J, K, I, D, E, F, G, B, C, D~~.
A, B, C, J, K, I, D, E, F, G, H, L.

level order/Breadth-first: L, K, I, H, A, J, E, F, G, B, C, D.

SSZG519- Introduction to Algorithms.

The importance of Algorithms:

Introduction:

The first Step towards an understanding of why the study and knowledge are so important is to define exactly what we mean by an algorithm.

According to the popular algorithms textbooks:

"introduction to algorithm (second edition by thomas H. cormen, charles E. Leiserson, Ronald L. Rivest, clifford stein),

"an algorithm is any well-defined computational procedure that takes some values, or set of values as input and produces some value, or set of values as output".

In-other words, algorithms are like road maps for accomplishing a given, well-defined task.

So, a chunk of code that calculates the terms of the fibonacci Sequence is an implementation, of a particular algorithm.

Some algorithms, like those that compute the fibonacci Sequences, are intuitive and may be innately embedded into our logical thinking & problem solving skills.

However, for most of us, complex algorithms are best studied so we can use them as building blocks for more efficient logical problem solving in the future.

In-fact, you may be surprised to learn just how many complex algorithms people use everyday when they check their e-mail or listen to music

Here we will introduce some basic ideas related to the analysis of algorithms

①

(2)

2000 discoveries that changed the world!!

↳ Johannes gutenberg : was a german black smith, gold smith, printer, and publisher. who introduced printing to Europe.

His introduction of mechanical movable type printing to Europe started the printing revolution.

Decimal System (600 AD)

The decimal ~~number~~ numeral system (also called base 10 or occasionally denary) has ten as its base. It is the numerical base most widely used by modern civilizations.

Al Khwarizmi (780 AD - 850 AD)

Al Khwarizmi contributions to mathematics, geography, astronomy, and cartography established the basis for innovation in algebra and trigonometry. His systematic approach to solving linear and quadratic equations led to algebra.

what is an algorithm?

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

- we can also view an algorithm as a tool for solving a well-specified computational problem.

Some vocabulary with an Example:

Problem: Sorting of given keys/numbers.

Input: A sequence of n-keys, $a_1, \dots, a_n$

output: The permutation (re-ordering) of the input sequence such that

$$a_1 \le a_2 \le a_3 \le \dots \le a_n .$$

Instance:

An Instance of sorting might be an array of names, likes {mike, Bob, sally...} or a list of number like {154, 245, 568..}.

Algorithm:

An algorithm is a procedure that takes any of the possible input instance and transforms it to the desired output.

Which algorithm is better?

There are many different algorithms for solving the problem

Ex: sorting

There are three desirable properties for a good algorithm we seek algorithms that are correct and efficient while being easy to implement.

Measure efficiency (asymptotic notation).

$O(n)$ - Big-O notation

$o(n)$ - Small/Little-o notation

$\Omega(n)$ - Big-Omega notation

$\Theta(n)$ - Big-Theta notation

Sorting Algorithms:-

Classification of sorting algorithms

1) Based on data sizes:

↳ External sort uses primary memory for the data currently being sorted and secondary storage for any data that will not fit in the primary memory.

↳ Internal sort is the sort in which all the data are held in the primary memory during the sorting process.

Based on information about data:

* Comparison based sorting: A comparison based algorithm orders a sorting array by weighing the values of one element against the value of other elements.

  Ex:- quick sort,
  merge, heap, bubbles and insertion sort.

* non-comparison based sorting:
  A non-comparison based algorithm sorts an array without consideration of pair wise data elements

  Ex: Bucket sort & radix sort.

* In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order.

The most-used orders are numerical order and lexicographical order.

The output must satisfy two conditions.

* The output is in non-decreasing order.
* The output is a permutation, or recordering of the input.

* <u>Randomized Algorithm</u>: is an algorithm that employs a degree of <u>randomness</u> as part of its logic

  The algorithm typically uses <u>uniformly random</u> bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits.

  * Quick sort → Any deterministic version of this algorithm requires $O(n^2)$ time to sort n-numbers

* If the algorithm selects <u>pivot</u> elements uniformly at random, it has a provably high probability of finishing in $O(n \log n)$

Data Structures :-

A data structure is a way to store and organize data in order to facilitate access and modifications

↳ No single data structure works well for all purposes, and so it is important to know the strengths & limitations of several of them.

* Changing a data structure in a slow program can work the same way an organ transplant does in a sick patient

In data structures, we will focus on each of the three fundamental abstract data Type

↳ Containers

↳ dictionaries & priority queues.

Contiguous vs linked Data structures.

Data structures can be neatly classified as either

Contiguous : contiguous-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.

- Linked data structures are composed of distinct chunks of memory bound together by pointers, and include lists, trees and graph adjacency lists.

* Arrays :-

The array is the fundamental contiguously-allocated data structure.

Array are structures of fixed-size data records such that each element can be efficiently located. by its index or (equivalently) address.

A good analogy likens an array to a street full of houses, where each array element is equivalent to a house, and the index is equivalent to the house number.

Advantages of contiguously-allocated arrays include:

- Constant-time access given the index
- Space-efficiency
- Memory locality

↳ Pointers and linked structures.

Pointers are the connections that hold the pieces of linked structures together.

Pointers represent the address of a location in memory. A variable storing a pointer to a given data item can provide more freedom than storing a copy of the item itself.

↳ Stacks & queues:

Container to denote a data structure that permits storage and retrieval of data items independent of content.

Container are distinguished by the particular retrieval order they support.

↳ The two important container

Stacks: Support retrieval by Last-in, first-out (LIFO) order.

Stacks are simple to implement and very efficient

queues: Support retrieval by (FIFO) order. queues are somewhat trickier to implement than stacks and thus are most appropriate for applications (like certain simulations) where the order is important.

Dictionaries:

The dictionary data type permits access to data items by content. you stick an item into a dictionary so you can find it when you need it.

The primary operations of dictionary support are.

Search (D, K) -

Insert (D, x) -

Delete (D, x) -

Binary Search trees:

we have seen data structures that allows fast search or flexible update, but not fast Search and flexible update

Ex: unsorted, doubly-linked lists supported insertion and deletion in O(1), but search took linear time in the worst case.

* Binary search requier that we have fast access to two elements
  - specifically the median elements above & below the given node.

Algorithm Techniques

↳ Divide and conquer (D&C).

D&C is an algorithm design paradigm based on multi-branched recursion

A divide and conquer, Algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.

The solution to the sub-problems are then combined to give a solution to the original problem.

Dynamic programming

Dynamic programming (dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler sub problems, solving each of those subproblems just once, & storing their solution - ideally, using a memory-based data structure.

* The next-time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, there by saving computation time at the expense of a modest expenditure in storage space.

Dynamic programming algorithm are often used to optimization.

A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for given problem.

↳ In comparison - greedy algorithm.

Matrix chain product:

Eg Example of dyamic programming

given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

greedy Approach.

Greedy Algorithm is an algorithm paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding global optimum.

In many problem, a greedy strategy doesnot in general produce an optimal solution

Ex: Travel salesman problem

Krusal's minimum spanning tree Algorithms. ⑨

↳ algorithm which finds an edge of the least possible weight that connects any two tree's in the forest

Prim's minimum spanning tree-Algorithm

It's a greedy algorithms that finds a minimum spanning tree for a weighted un-directed graph.

↳ This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Graph Algorithms:

Graphs algorithms are used to solve many real-life problems.

"In the real world, many problems are represented in terms of objects and connections between them. for Example, in an airline route map, we might be interested in questions like: "what's the fastest way to go from Hyderabad to New York?" or "what is the cheapest way to go from hyderabad to New York?" To answer these questions we need information about connections (airline routes) between objects (towns). Graphs are data structures used for solving these kinds of problems.

using the following Algorithms.

↳ shortest Path Algorithms:

— Dijkstra's

— floyd-Warshall's.

All pairs shortest Path Algorithm.

MST Algorithms:

Kruskual & Prims.

Graph traversal Methods: BFS & DFS.

P, NP, NP-Complete

Decision → No/Yes

class-P

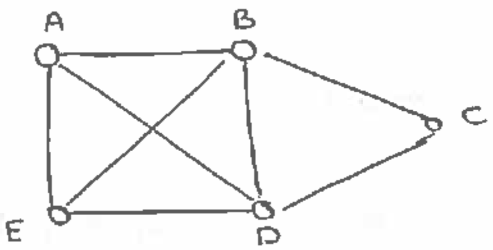
- Yes/No Problems with a polynomial-time algorithm.

Ex: $a_1, a_2, a_3 \dots a_n \rightarrow$ search $O(n)$, $O(n^2)$.

Class NP

- Yes/no problems with a polynomial-time "checking algorithm"

- more precisely, given a solution (E.g a subset of vertices) we can check in a polynomial time if that solution is what we are looking for (E.g is it a clique of size k?)



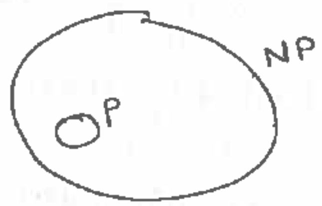
5-node
clique of size → 4
as all ABDE, all nodes are connected to each other,

* Is all problem in 'P' are there in NP?
i.e $P \subseteq NP$

$a_1, a_2, \dots a_n \rightarrow$ Sorting
$\Downarrow$
$a_1', a_2', \dots a_n' \rightarrow$ Sorting output checking
$\Downarrow$
$a_1 \le a_2 \le a_3 \le a_4 \dots \le a_{n-1} \le a_n$ [verify in a polynomial time]

$\therefore$ $P \subseteq NP$ [is true]

* Big open problem
prove P=NP



NP-Complete

NP-hard

A problem is NP hard, if all other problems in NP can be polynomially reduced to it.

NP- complete

A problem is NP-complete, if it is (a) in NP, and.

(b) NP-hard.

In-short:

NP-complete : the most difficult problem in NP.

(or)

NP- Complete: [Must they should be decision problem].

NP- complete is a complexity class which represents the set of all problem X in NP for which it is possible to reduce any problem Y to X in polynomial time

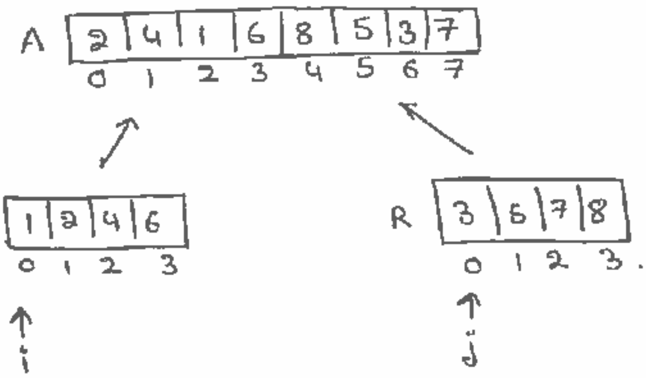
Ex: chinese postman problem.

Merge sort

we have talked about.

1) selection sort
2) Bubble sort
3) Insertion Sort

} $O(n^2)$



code // Pseudocode.

```
merge (L,R,A)
{
  nL ← length (L) // length of the element in L array
  nR ← length (R) // length of the     "      " R array
  i←0; j←0; K←0;
  while (i<nL && j<nR) // when the i & j are smaller then the.
                         length of nL, nR then we enter the
                         loop.
  {
    if (L[i] <= R[j]) // to find the smallest element
                         in L & R → array.
    {
      A[K] ← L[i] // update A[K] with element at L[i]
      i ← i+1; // increment i
    }
    else.
    {
      A[K] ← R[j] //update A[K] with element at R[j]
      j←j+1; //increment j
    }
    k←k+1; // increment K.
  }
```

```
while (i<nL) // when the first loop false.
 {
  A[K] ← L[i];
   i ← i+1;
    K ← K+1;
 }
 while (j<RL)
  {
   A[K] ← R[j];
    j ← j+1;
    k ← k+1;
   }
 }
```

Code of merge sort // Pseudocode

```
merge sort (A)
 {
  n ← length (A)
  if (n<2)
      return
  mid ← n/2.
  left ← array of size (mid)
  Right ← array of size (n-mid)
 for i ← 0 to mid - 1 // fill the element in left.
    left [i] ← A [i]
 for i ← mid to n-1 // fill the element in right
    right [i-mid] ← A [i]
  merge sort (left)   // Recursive call for sort left list
  merge sort (right) // "       "     "  "  Right list
  merge sort (left, right, A)

 }
```

Time complexity → pseudocode.

Some properties

1) Divide & conquer.
2) Recursive
3) Stable.
4) Not-in-place.
   ↳ $O(n)$ → Space complexity
   but we should use $\Theta(n)$ → theta notation.

5) $\Theta(n\log n)$ time complexity

Time complexity

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ 2T(n/2) + c'n, & \text{if } n > 1 \end{cases}$$

$$T(n) = 2T(n/2) + c'n$$

$$= 2\{2T(n/4) + c'n/2\} + c'n$$

$$= 4T(n/4) + 2c'n$$

$$= 4\{2T(n/8) + c'.n/4\} + 2c'n$$

$$= 8T(n/8) + 3c'n$$

$$= 16T(n/16) + 4c'n$$

$$= 2^k T(n/2^k) + kc'n$$

$n/2^k = 1 \Rightarrow 2^k = n.$

$\Rightarrow k = \log_2 n.$

$\Rightarrow 2^{\log_2 n} T(1) + \log_2 n . c'. n$

$\Rightarrow nc + \underbrace{c'}_{\text{constant}} n\log n.$

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

Space complexity

If we don't clear the extra space then, we will $\Theta(n\log n)$ space.

If, we clean the extra space then, we will use $\Theta(n)$ space.

*Extraspace.

$n \downarrow n/2 \downarrow n/4 \downarrow \cdots \downarrow 1$

$$\frac{n}{2^k} = 1.$$

$k = \log_2 n$

If we do not clear Extra memory for left and right

$$\underbrace{n}_{n\text{-element}} \times \underbrace{\log n}_{\text{for level}} = \Theta(n\log n).$$

Merge sort:

$O(n\log n)$: worst case running time.

$O(n)$ – Space complexity

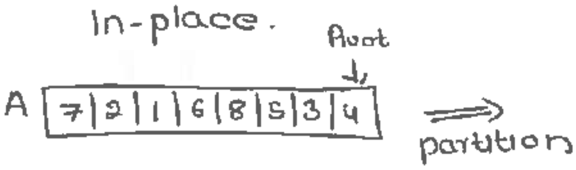
Not-in-place / Extra-Memory.

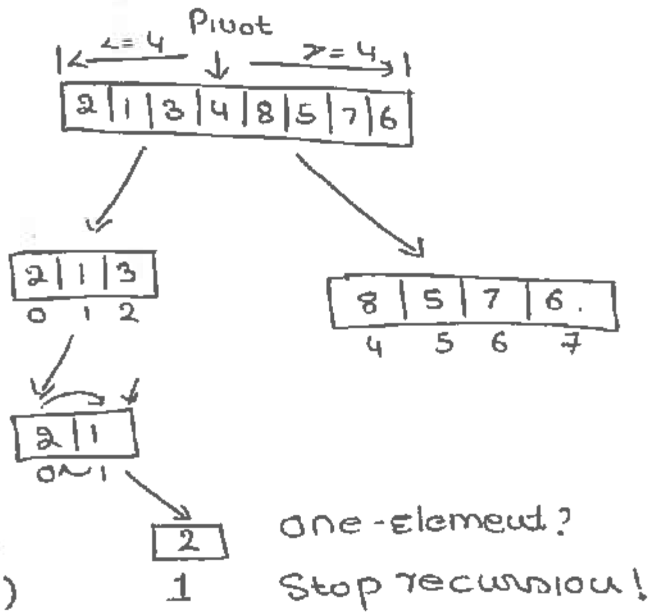
Quick Sort

Quicksort: $O(n\log n)$ - Average case running time

$O(n^2)$ - worst case running time.

In-place.






```
quicksort(A, start, end)
{
  if (start < end)
  Pindex ← partition(A, start,
                    end)
  quicksort (A, Start, Pindex-1).
  quicksort (A, Pindex+1, end)
}.
```

```
Partition (A, Start, end)
{
  Pivot ← A[end]
  Pindex ← start.
  for i ← start to end-1
  {
    if (a[i] <= pivot)
    {
      Swap (A[i], A[index])
      pindex ← pindex+1
    }
  }
  Swap (A[pindex], A[end]).
```

Analysis of quick sort

1) Divide and conquer
2) Recursive
3) Not stable.
4) Time complexity $O(n \log n) \longrightarrow$ best or Average case.

$O(n^2) \longrightarrow$ worst case
↳ can be avoided

```
Quick Sort(A, Start, end)
{ if (start < end)                        ---> c1
  {
    Pindex <- Partition(A, Start, end)    ---> an+b'
    (constant)
    QuickSort (A, Start, Pindex-1)        ---> T(n/2)
    QuickSort (A, Pindex+1, end)          ---> T(n/2)
  }
}

Partition(A, Start, end)
{
  Pivot <- A[end]                ---- (1)
  Pindex <- Start                ---- (2)
  for i <- start to end-1
  {
    if (A[i] <= pivot)
    {
      Swap (A[i], A[Pindex])           } a
      Pindex <- Pindex + 1
    }
  }
  Swap (A[Pindex], A[end])       ---- (3)
  return Pindex                  ---- (4)
}.
```

$T = an + b$
$= O(n)$

Basecase:

$$T(n) = 2T(n/2) + an + b' + c_1$$

$$T(n) = 2T(n/2) + c \cdot n \quad \text{if } n > 1$$

$$c_1, \quad \text{if } n = 1$$

$$T(1) = c_1$$

$$T(n) = 2T(n/2) + c \cdot n.$$

$$= 2\{2T(n/4) + c \cdot n/2\} + c.n$$

$$= 4T(n/4) + 2cn$$

$$= 8T(n/8) + 3cn$$

$$= 2^k T(\underline{n/2^k}) + kcn$$

↳ write T in T(1) terms.

$$n/2^k \Rightarrow 1 = 2^k = n.$$

$$= k = \log_2 n$$

$$= 2^{\log_2 n} T(1) + c \cdot n \log_2 n$$

$$= nc_1 + c \cdot n \log n$$

$$= O(n \log n) \text{ or } \Theta(n \log n).$$

Worst-case

↳ unbalanced partition, then we have worst-case

```
Quicksort(A, Start, end)
{ if (start < end) ———— c1
  { PIndex ← Partition (A, Start, end)
                          ↳ an+b'
    Quicksort(A, start, PIndex-1) ⟶ T(n-1)
    Quicksort (A, PIndex+1, end)
                  ↳ Some constant.
  }
}
```

Worst-case

$T(n) = T(n-1) + c.n$ } The worst case is when we have un-balanced partition

$T(1) = c_1$

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

$= \{T(n-2) + c(n-1) + cn\}$.

$= T(n-2) + 2cn - c \Rightarrow T(n-3) + c(n-2) + 2cn - c$

$= T(n-3) + 3cn - 3c$

$= T(n-4) + 4cn - 6c$

$= T(n-k) + kcn - \underbrace{(1+2+3+\cdots+k-1)c}_{k(k-1)/2.}$

$= T(\underline{n-k}) + kcn - \frac{k(k-1)}{2}.c$

↳ To write in $T(1)$ terms.

$n-k=1 \Rightarrow k=n = T(1) + cn^2 - \frac{n(n-1)}{2}.c$

$= T(n) = c_1 + \frac{cn(n+1)}{2.}$

$= \frac{cn^2}{2} + \frac{cn}{2} + c = O(n^2)$.

STACKS

what is a stack?

A stack is a simple data structure used for ~~stro~~ storing data (similar to linked lists). In a stack, the order in which the data arrives is important.

A pile of plates in a cafeteria is a good example of a stack. The plates are added to the stack as they are cleaned and they are placed on the top.

when a plate, is required it is taken from the top of the stack.

The first plate placed on the stack is the last one to be used.

Definition:-

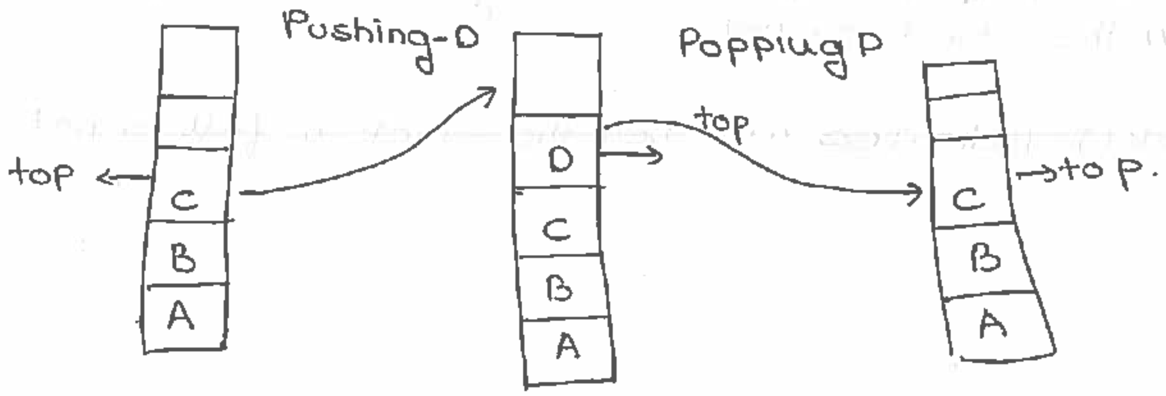
A stack is an ordered list in which insertion and deletion are done at one end, called top. The last element inserted is the first one to be deleted. Hence, it is called the Last in first out (LIFO) or first in last out (FILO) list.

Special names are given to the two changes that can be made to a stack.

when an element is inserted in a stack, the concept is called push, and when an element is removed from the stack, the concept is called pop.

Trying to pop out an empty stack is called underflow and trying to push an element in a full stack is called overflow

Generally, we treat them as exceptions. As an example, consider the snapshots of the stack.



How Stacks are Used.

Consider a working day in the office. Let us assume a developer is working on a long-term project. The manager then gives the developer a new task which is more important.

The developer puts the long-term project aside and begins work on the new task. The phone rings, and this is the highest priority as it must be answered immediately.

The developer pushes the present task into the pending tray and answers the phone.

When the call is complete the task that was abandoned to answer the phone is retrieved from the pending tray and work progresses. To take another call, it may have to be handled in the same manner, but eventually the new task will be finished, and the developer can draw the long-term project from the pending tray and continue with that.

STACK ADT

The following operations make a stack an ADT. For simplicity, assume the data is an integer type.

Main stack operations

↳ push (int data): Inserts data onto stack.

↳ int Pop(): Removes and returns the last inserted element from the stack.

Auxiliary stack operations

↳ int Top(): Returns the last inserted element without removing it.

int size(): Returns the number of elements stored in the stack.

int IsEmptyStack(): Indicates whether any elements are stored in the stack or not.

int IsFullStack(): Indicates whether the stack is full or not

Exceptions

Attempting the execution of an operation may sometimes cause an error condition, called an exception. Exceptions are said to be "thrown" by an operation that cannot be executed. In the stack ADT, operations pop and top cannot be performed if the stack is empty.

Attempting the execution of pop (top) on an empty stack throws an Exception.

Trying to push an Element in a full stack throws an Exception.

Applications

following are some of the applications in which stacks play an important role.

Direct Applications

- Balancing of symbols.
- Infix-to-postfix conversion.
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- finding of spans (finding spans in stock markets, refer to Problems section)
- page-visited history in a web browser [Back Buttons]
- undo-sequence in a text editor.
- matching Tags in HTML and XML.

Indirect applications

- Auxiliary data structure for other algorithms. [Ex: Tree Traversal algorithms]
- Component of other data structures (Ex: simulating queues, refer Queues chapter)

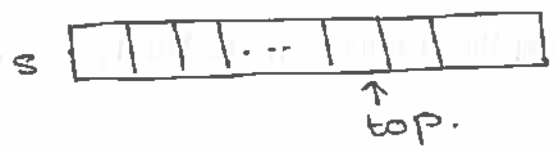
Implementation

There are many ways of implementing stack ADT; below are the commonly used methods.

- Simple array based implementation
- Dynamic array based implementation
- Linked lists implementation.

Simple Array Implementation

This implementation of stack ADT uses an array. In the array we add elements from left to right and use a variable to keep track of the index of the top element.



The array storing the stack elements may become full. A push operation will then throw a full stack exception. Similarly, if we try deleting an element from an empty stack it will throw stack empty exception.

Performance & limitations

Performance:

Let n be the number of elements in the stack. The complexity of stack operations with this representation can be given as:

Space complexity (for n push operations)	$O(n)$
Time complexity of push()	$O(1)$.
Time complexity of pop()	$O(1)$
Time complexity of size	$O(1)$.
Time complexity of IsEmptyStack()	$O(1)$
IsFullStack()	$O(1)$
DeleteStack()	$O(1)$.

Queues

what is a Queue?

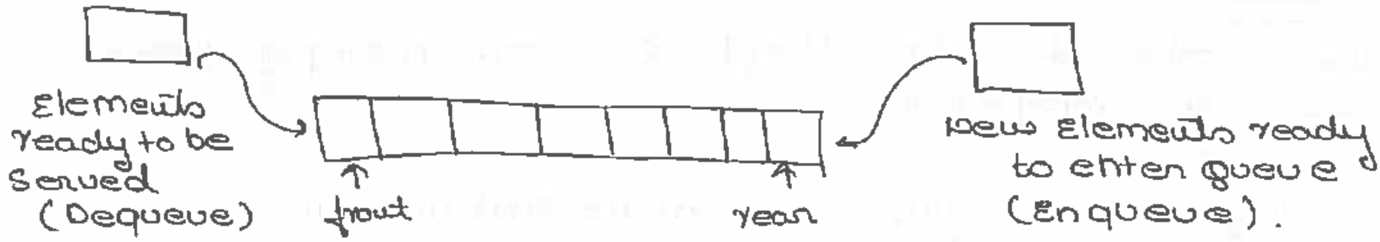
A queue is a data structure used for storing data (similar to linked lists and stacks). In queue, the order in which data arrives is important. In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

Definition:-

A queue is an ordered list in which insertions are done at one end (rear) and deletions are done at other end (front). The first element to be inserted is the first one to be deleted. Hence, it is called first in first out (FIFO) or last in last out (LILO) list.

Similar to stacks, special names are given to the two changes that can be made to a queue. when an element is inserted in a queue, the concept is called Enqueue, and when an element is removed from the queue, the concept is called Dequeue

Dequeueing an Empty queue is called underflow and Enqueuing an element in a full queue is called overflow. Generally, we treat them as exceptions. As an Example, consider the snapshot of the queue.



How are Queues used?

The concept of a queue can be explained by observing a line at a reservation counter. when we enter the line we stand at the end of the line and the person who is at the front of the line is the one who will be served next. He will exit the queue and be served.

Queue ADT

The following operations make a queue an ADT. Insertions and deletions in the queue must follow the FIFO scheme.
for simplicity we assume the elements are integers.

Main Queue operations

↳ Enqueue (int data): Inserts an element at the end of the queue

↳ int DeQueue() : Removes and returns the element at the front of the queue.

Auxiliary Queue Operations

int front(): Returns the element at the front without removing it.

int Queue Size(): Returns the number of elements stored in the queue.

int Is Empty Queue(): Indicates whether no elements are stored in the queue or not.

Exceptions

Similar to other ADTs, executing DeQueue on an empty queue throws an "Empty Queue Exception" and executing Enqueue on a full queue throws "full Queue Exception".

~~Applications~~
~~Similar to other ADTs, Executing DeQueue on an empty queue throws an "Empty Queue~~

Applications

following are some of the applications that use queues.

Direct applications.

↳ operating systems schedule jobs (with equal priority) in the order of arrival (e.g, a print queue).

↳ Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.

↳ Multiprogramming
↳ Asynchronous data transfer (file IO, pipes, sockets).
↳ waiting times of customers at call center.
↳ Determing number of cashiers to have at a supermarket.

Indirect Applications
↳ Auxiliary data structure for algorithms
↳ Component of other data structures.

Implementation:
There are many ways (similar to stacks) of implementing queue operations and some of the commonly used methods are listed below.

↳ simple circular array based implementation.
↳ Dynamic circular array based implementation
↳ Linked list implementation.

Why Circular Arrays
First, let us see whether we can use simple arrays for implementing queues as we have done for stacks. We know that, in queues, the insertions are performed at one end and deletions are performed at the other end. After performing some insertions and deletions the process becomes ~~eas~~ easy to understand.

In the Example shown below, it can be seen clearly that the initial slots of the array are getting wasted. So simple array implementation for queue is not efficient. To solve this problem we assume the arrays as circular arrays. That means, we treat the last element and the first array elements as contiguous. With this representation, if there are any free slots at the beginning, the rear pointer can easily go to its next free slots.