Data Structure : Data structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Example of data structure are array, linked list, stack, Queue etc.

Basic Terminology :

Data : Data can be defined as an elementary value or the collection of values, for example, student's name and it's id are the data of student.

Group Items : Data item which have subordinate data items are called Group item. for example, name of a student can have first name and the last name.

Record : Record can be defined as the collection of various data items, for example, if we talk about the student entity, then it's name, course and marks can be grouped together to form the record of the student.

File : A file is a collection of various records of one type of entity. for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity : An entity represents the class of certain objects. it's contain various attributes. each attribute represents the particular property of that entity.

Field : Field is a single elementary unit of information representing the attribute of an entity.

Classification of Data Structure

2 types :

(i) Liniar Data structure
(ii) Non Liniar Data structure.

(i) Liniar Data Structure :

A data structure is called liniar if all of it's elements are arranged in the liniar order. In liniar structure, the elements are stored non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of liniar Data structure :

1. Array.
2. Linked list
3. stack
4. queues

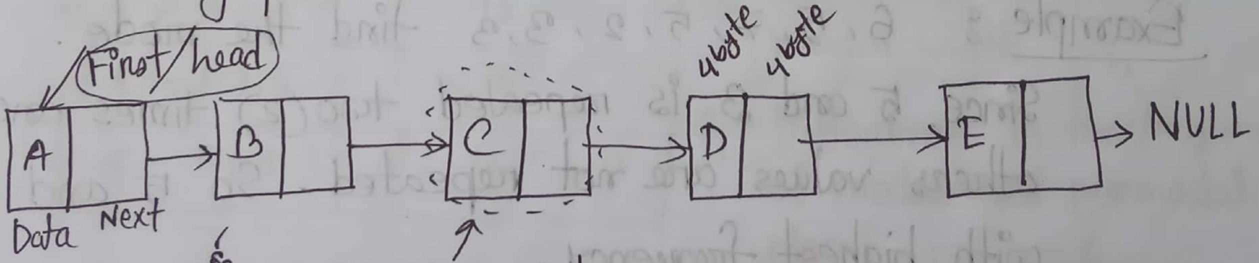
Arrays : An array is a collection of similar type of data item and data item is called the element of the array. The data type of the element may be any valid data-type like int, char, float, double etc.

example : a[0], a[1], a[2] . . . . . . a[99], a[100]

size = 100+1 = 101 (the array size will be static)

Linked list : Linked list is a liniar data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to it's adjacent node.

OR : A linked list is a liniar data structure in which the elements are not stored at contigious memory location. The element in a linked list are linked using pointer.



this is a node

A node contains two things

data	next

```
struct node
{
    int data;
    struct *next;
}
```

Array element এর : memory location consequtive হয় but link list এর memory location [illegible] where [illegible] । [illegible] data র সাথে next data র adress [illegible] থাকে।

need 4+4 byte — for value, next pointer (memory)

if(p->next = =NULL) not point anymore

Stack: stack is a liniar list in which insertion and deletion are allowed only at one end called top.

A stack data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves likes a real stack, For example: piles of plates or deck of cards etc. It follows First In last Out (FILO)

Queue: It is an abstract data structure, ~~similar to~~ in which elements can be inserted only at one end called front. it is an abstract data structure, similar to stack. Queue is opened at both ~~ends~~ end therefore it follows First In First Out (FIFO) methology for storing the data item.

(ii) Non linear Data Structure: This data structure does not form a sequence i.e. each item or element is connected with two or more other item in a non linear arrangement. The data elements were not arranged in sequential structure.

Types of non linear data structure

1. Tree
2. Graph.

Trees : Trees are multilevel data structures with a ~~the~~ hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called leaf node while the topmost node is called root node. Each nodes contains pointer to points adjacent nodes.

Tree data structure is based on the parent - child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have one parent except the root node. Trees can be classfied into many categories. ~~which will be~~

Graph : Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can be cycly while the tree cannot have the one

Operation on Data Structure :

Traversing :

Every data structure contain the set of data elements. Traversing the data structure means visiting each elements of the data structure in order to perform some specific operation like searching searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total ~~or~~ sum, Then we will devide that sum by 6 in order to find the average.

Insertion :

Insertion can be defined as the process of adding the elements to the data structure at any location.

→ If the size of data structure is n then we can only insert n-1 data element into it.

Deletion :

The process of removing an element from the data structure is called deletion. We can delete an element from the data structure at any random location.

→ If we try to delete an element from the empty data structure then underflow occurs.

Searching: The process of finding the location of an element within the data structure is called searching. There are two algorithm to perform searching: Linear search and Binary Search.

Sorting: The process of arranging the data structure in a specific order is known as Sorting. There are many algorithm that can be used to perform Sorting. Example: Insertion sort, Selection Sort, Bubble Sort etc.

Merging: When two list of List A and List B of size M and N respectively, of similar type of element, clubbed or joined to produce the third list, List C of size (M+N), then this process is called Merging.

Updation: We can also update element by replacing any element or value which we want.

Linked List

A link list in a linear collection of data elements (called nodes) where the linear order is given by means of pointers.

Each node is divided into 2 parts :

① 1st part : It contains the information of the elements.

② 2nd Part : It's called the link field or next pointer field which contains the adress of the next node in the list.

Uses of Linked list :

- The list is not required to be contiguously present in the memory and linked together to make a list. This achives optimized utilization of space.
- List size is limited to the memory size and doesn't ~~need~~ need to be declared in advance.
- Empty node cannot be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over Array

Arrays contain following limitations:

- The size of array must be known advance before using it in the program.
- Increasing size of array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all it's predecessors

USEFUL:

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because:

- It allocates the memory dynamically. All the nodes of link list are non-contiguously stored in the memory and linked together with the help of pointer.
- Sizing is no longer a problem since we donot need to define it's size at the time of declaration. List grows as per the programs demand and limited to the available memory space.

Advantage over array

1. Dynamic size
2. Ease of insertion/deletion.

Disadvantage of Linked list

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot to binary search with linked list efficiently with it's default implementation.
2. Extra memory space for a pointer requirde with each element of the list.
3. Not cache friendly. Since array elements are contigueous locations, there is locality of reference which is not there in case of linked list.

Basic Operations:

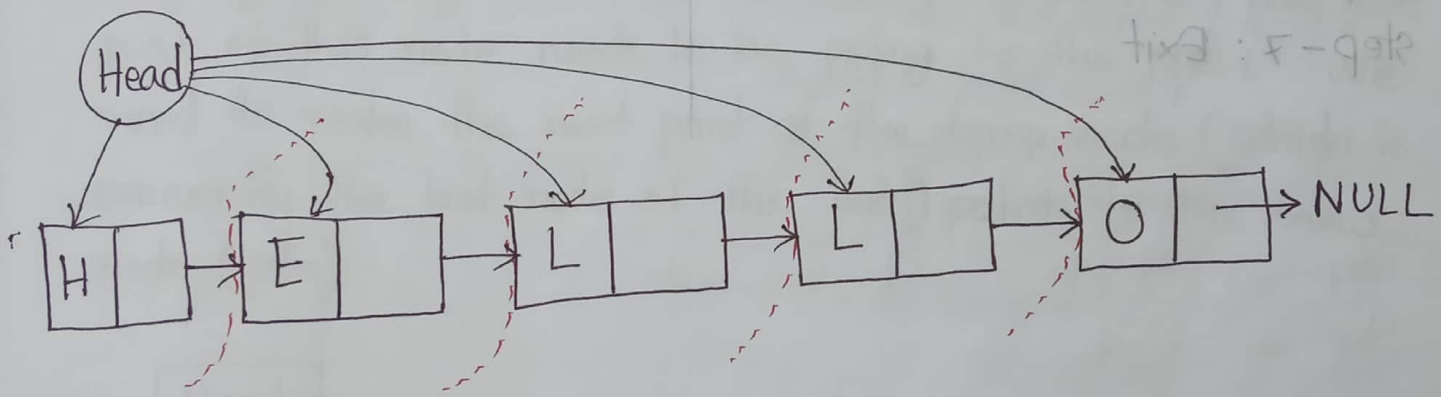
- Insert: Add a new node in the first, last or interior of the list
- Delete: Delete a node from " " " " " " "
- Search: Search a node containing particular value in the linked list.

A Node can be added in three ways

1) At the front of the linked list
2) After a given node.
3) At the end of the linked list.

Insert First :

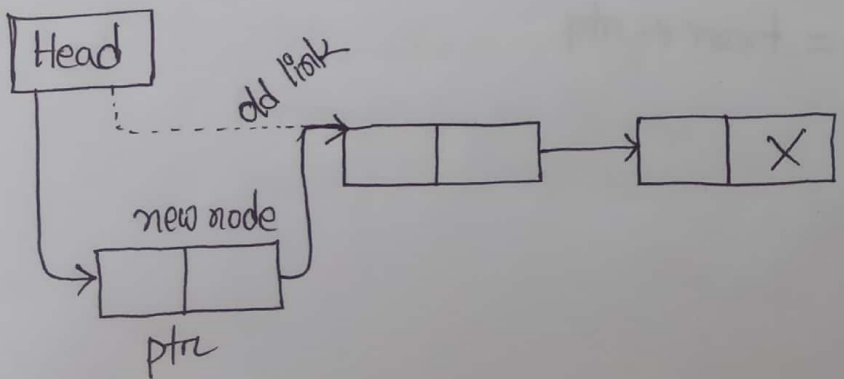
- To add a new node to the head of the linear linked list, we need to construct a new node that is pointed by pointer newitem.
- Assume there is a global variable, head which points to the first node in the list.
- The new nodes points to the first node in the list the head is then set to point to the new node.



Step-1 : Allocate the space for the new node and store data into the data part of the node.

Step-2 : Make the link part of the new node pointing to the existing first node of the list.

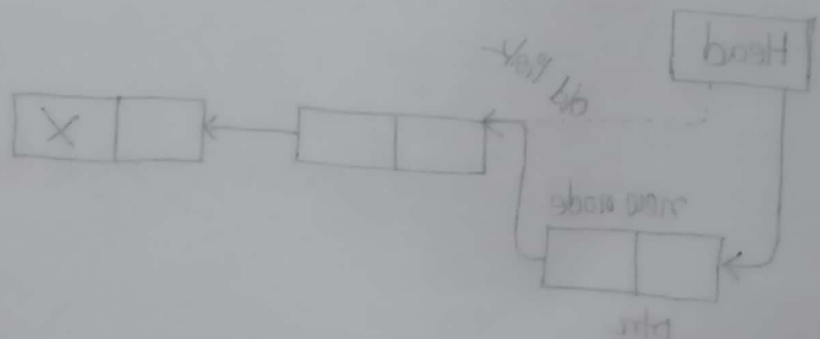
Step-3 : At the last, we need to make the new node as the first node of the list.



ptr → next = head
head = ptr

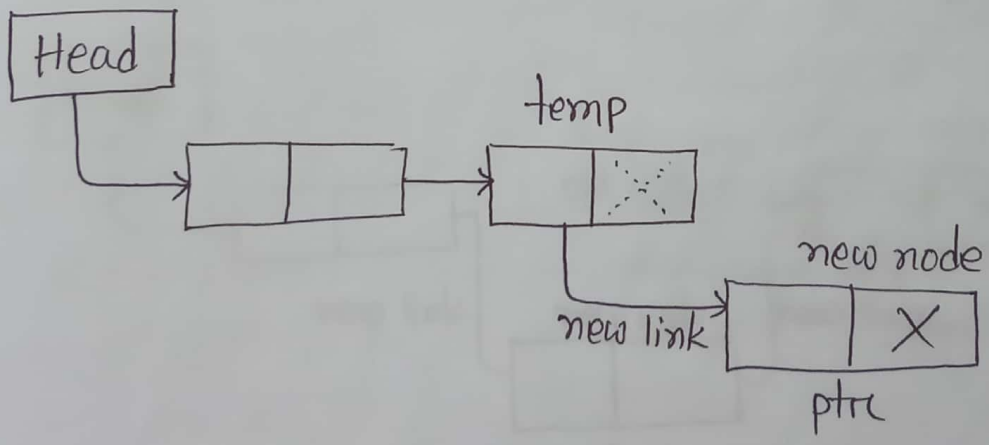
Algorithm:

Step-1 : If ptr = NULL
write overflow
Go to step 7
[end of if]
Step-2 : set new-node = ptr
Step-3 : set ptr = ptr → next
Step-4 : set new_node → data = value
Step-5 : set new_node → next = head
Step-6 : set head = New_node.
Step-7 : Exit



Insertion at the end:

- We need to declare a temporary pointer temp in order to traverse through the list. temp is made to point the first node of the list.
- Then, traverse through the entire linked list using the statements:
- At the end of the loop, the temp will be pointing to the last node of the list. Now allocate the space for the new node, and assign the item to it's data part. Since, the new node is going to be the last node of the list hence, the next part of this node needs to be poing to the NULL. We need to make the next part of the temp node (which is currently the last node of the list) point to the new node (ptr)



temp→next = ptr
ptr→next = NULL

Algorithm :

Step - 1 : If ptr = NULL
write overload
Go to step 10

Step - 2 : set new_node = ptr

Step - 3 : set ptr = ptr → next

Step - 4 : set new_node → data = value

Step - 5 : set new_node → next = NULL

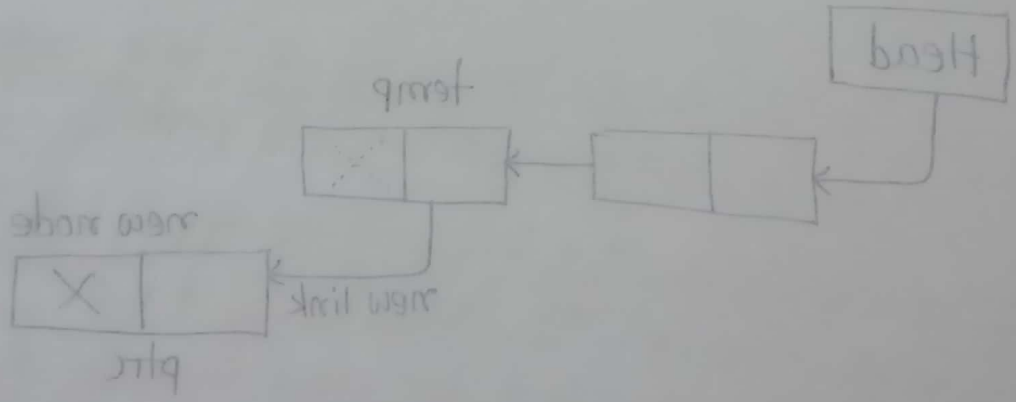
Step - 6 : set ptr = head

Step - 7 : Repeat 8 while
ptr → next != NULL

Step - 8 : set ptr = ptr → next
[end of loop]

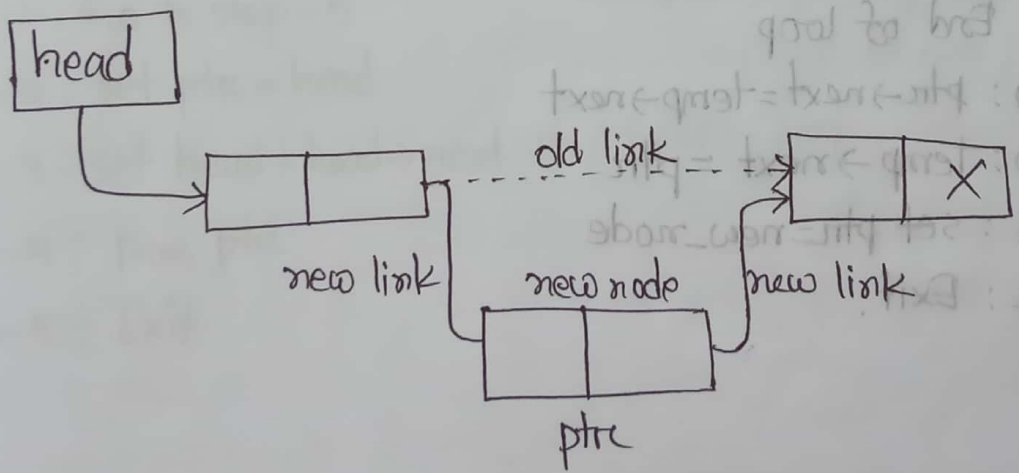
Step - 9 : set ptr → next = new_node

Step - 10 : Exit.



Insertion After Specified Node :

- Allocate the space for the new node and add the item to the data part of it.
- Now we just ~~new~~ need to make a few more link adjustments and our node at will be inserted at the specified position. since, at the end of the loop, the loop ~~of~~ pointer temp would be pointing to the node after which the new node will be inserted. Therefore, the next part of the new node ptr must contain the adress of the next part of the temp (since; ptr will be in between temp and the next of the temp)
- no, we just need to make the next part of the temp, point to the new ptr. This will be insert the new node ptr, at the specified position.

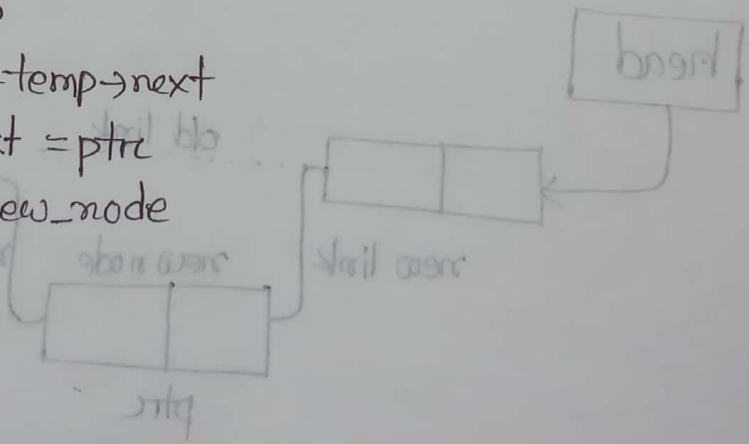


ptr → next = temp → next

temp → next → ptr.

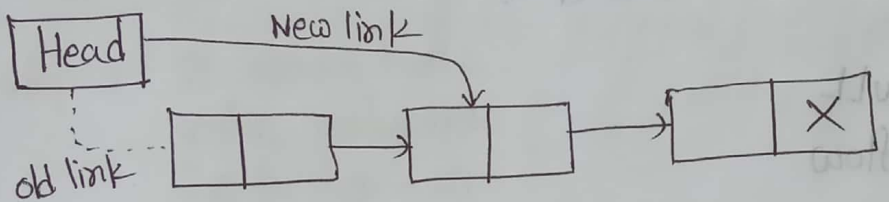
Algorithm :

Step-1 : If ptr = NULL
write overflow
Go to step(12)
step-2 : set new_node = ptr
step-3 : new_node → data = value
Step-4 : set temp = head
step-5 : set i = 0
step-6 : Repeat step-7 to 8 until
i < loc < li = "" > < /loc ,
Step-7 : temp = temp → next
step-8 : If temp = NULL
write "desired node
not present"
End of if
End of loop
step-9 : ptr → next = temp → next
step-10 : temp → next = ptr
step-11 : set ptr = new_node
step-12 : Exit.



Deletion at Beginning:

- Deletion a node from the begining of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is to be deleted. therefore, we just need to make the head, point to the next of the head.

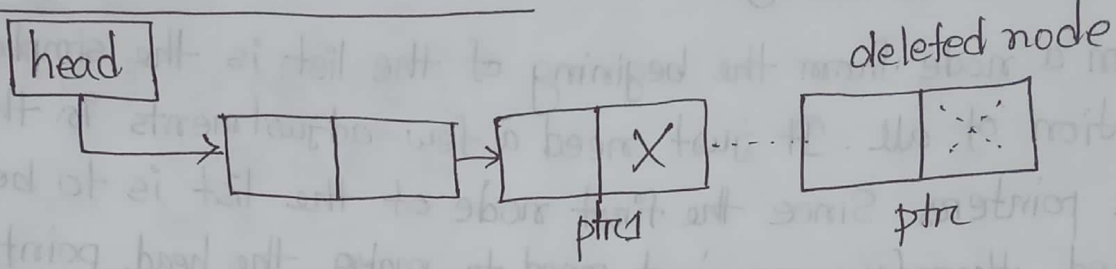


ptr = head
head = ptr → next
free (ptr)

Algorithm:

step-1 : If head = NULL
write underflow
Go to step-5
step-2 : set ptr = head
step-3 : set head = head → next
Step-4 : Free ptr
step-5 : Exit

Deletion at the end



ptr1 → next = NULL
free (ptr)

Algorithm

step-1 : If head = NULL
write underflow
go to step 8

step-2 : set ptr = head

step-3 : repeat step 4 and 5
while ptr → next != NULL

step-4 : set preptr = ptr

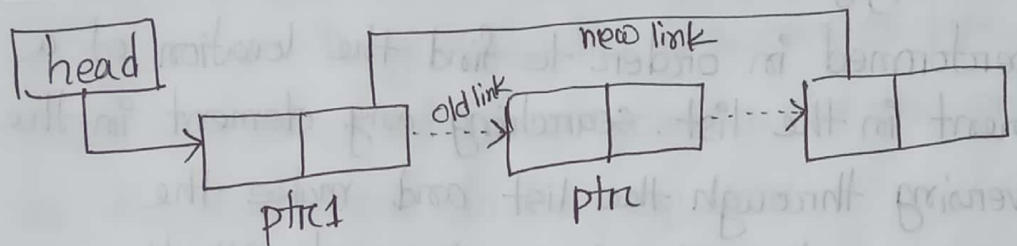
step-5 : set ptr = ptr → Next
[end of loop]

step-6 : set preptr → next = NULL

step-7 : Free ptr

step-8 : Exit

Deletion After the specified node



ptr1->next = ptr->next
free (ptr)

Algorithm :

step 1 : If head = NULL
write underflow
Go to step - 10
End of if

step-2 : set temp = head

step-3 : set i = 0

step-4 : Repeat step-5 to 8
until i<loc<li=""></loc<>

step-5 : temp1 = temp

step-6 : temp = temp->next

step-7 : If temp = NULL
write "desired node not present"
Go to step 11

step-8 : i = i+1
(end of Loop)

step-9 : temp1->next = temp->next

step-10 : free temp

step-11 : Exit

Searching in singly linked List

Searching is performed in order to find the location of a particular element in the list. searching any element in the list needs traversing through the list and make the comparison of every element of the list ~~tot~~ with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

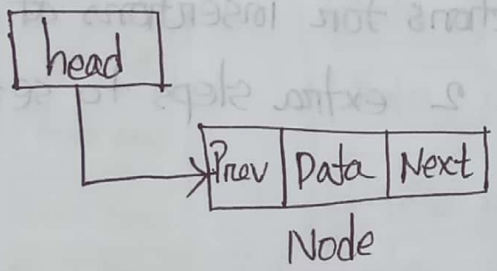
Algorithm:

step-1 : set ptr=head
step-2 : set i=0
step-3 : If ptr= NULL
write "Empty list"
Go to step 8
step-4 : repeat step 5 to 7
untill ptr!=NULL
step-5 : if ptr→data = item
write i+1
End of IF
step-6 : i = i+1
step-7 : ptr = ptr→next
[End of loop]
step-8 : Exit

Doubly linked list

Dubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.

Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the next node in sequence (next pointer, pointe to the previous node (previous pointer)



Disadvantages over singly linked list

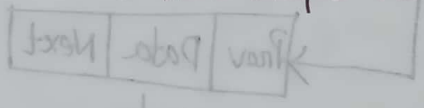
~~1. Every node of DLL require extra~~

1. A DLL can be traversed in both forword and backword direction.
2. The delete operation in DLL is morte efficient if pointer to the node to be deleted is given.
3. We can quickly insert a new node before a given node

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, Sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantage over singly linked list

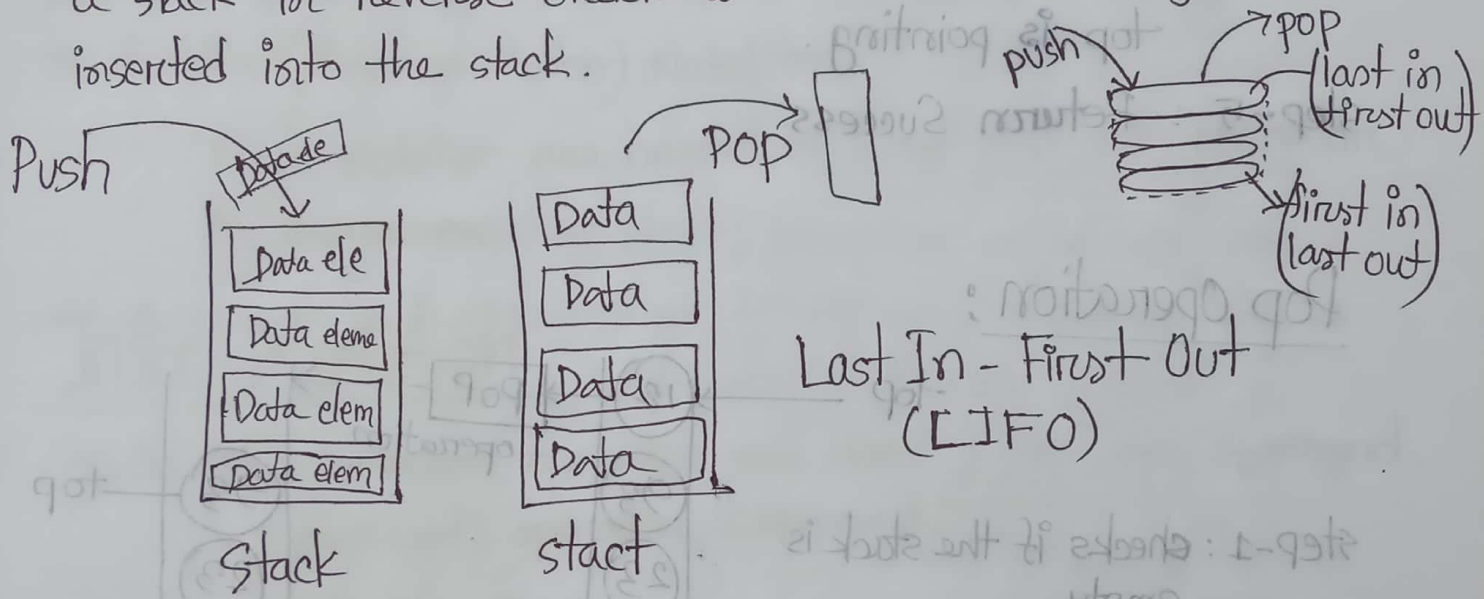
1. Every node of DLL require extra ~~space~~ space for an previous pointer. It is impossible to implement DLL with single pointer through (see this)
2. All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example, in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.



Stack

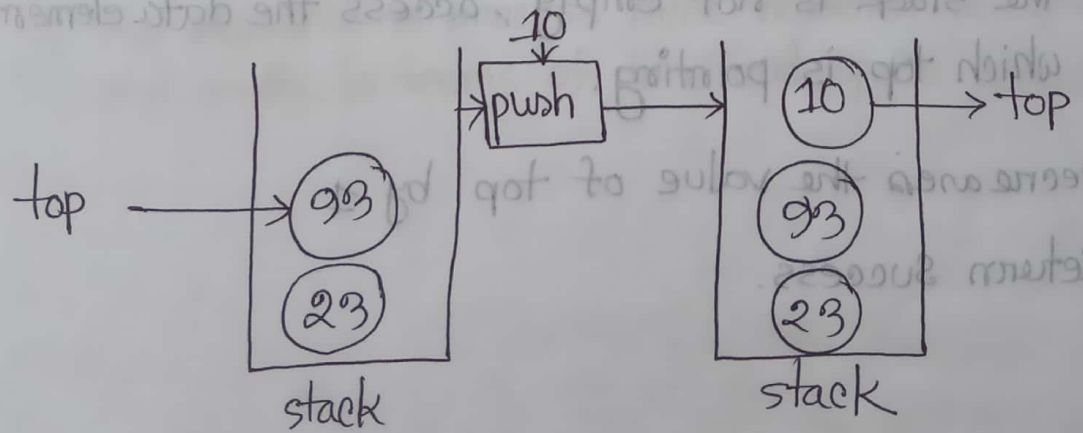
A stack is a list of element in which an element may be inserted or deleted only at one end, called the top of the stack.

This means, in particular, that elements are removed from a stack in reverse order of that in which they were inserted into the stack.



Push : push is the term used to insert an element into a stack

Pop : Pop is the term used to delete an element from a stack.



Push Operation :

Step-1 : check, if the stack is full

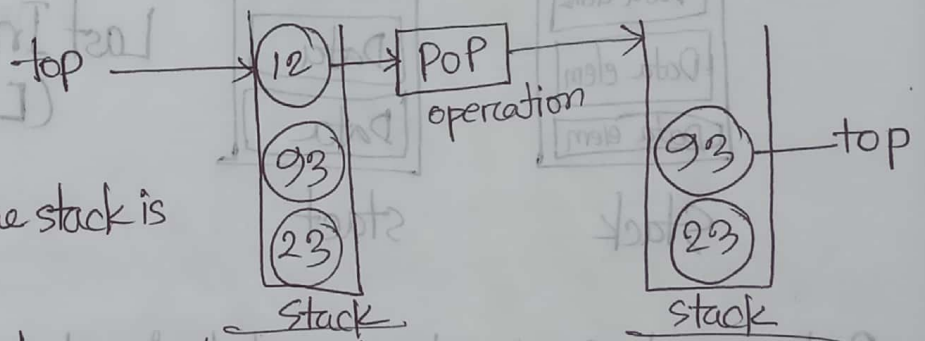
step-2 : If the stack is full, produces an error and exit.

step-3 : If the stack is not full, increments top to point next empty space

step-4 : Adds data element to the stack location where top is pointing.

step-5 : Return Success

Pop Operation :



step-1 : checks if the stack is empty.

step-2 : If the stack is empty, produces an error and exit.

step-3 : If the stack is not empty, access the data element at which top is pointing.

step-4 : Decreases the value of top by 1

step-5 : Return Success.

Notation

The way to write arithmatic expression is known as a notation. An arithmatic expression can be written in three different but equivalent notations. These notation are—

1. Infix notation
2. Prefix notation (Polish)
3. Postfix (Reverse-Polish) Notation

These notation are named as how they use operator in expression.

Infix-Notation

1. In infix notation, operator are used in between operand.

   <operand> operator <operand>

   ex: a-b+c, 5+1

2. It is easy to read, write and speak but the same does not go well in computing devices.

3. An Algorith to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

1. operator is prefixed to operands. operator is written ahead of operands. ex : +ab
2. also known as polish notation
3. a+b ~ +ab
   infix prefix

<operator> <operand> <operand>

a*b+c ~ *ab+c ~ +*abc

Postfix Notation

1. Also known as reversed polish notation.
2. The operator is postfixed to the operand. the operator is written after the operands. ex: ab+
3. a+b ~ ab+
   infix postfix

<operand> <operand> <operator>

(*) 5+1 ~ 51+

(*) a*b+c

~ ab*+c

~ ab*c+

Precedence of the operator

(i) () { } [ ]

(ii) ^ → R-L

(iii) */ → L-R

(iv) +- → L-R

priority : ->+

1+2*5+30/5

= 1+10+30/5

= 1+10+6

= 11+6

= 17

2^2^3 (R-L)

= 2^8

= 256

((1+2)*5)+(30/5)

=

Evaluating Arithmetic Postfix Expression

Postfix expression: 234+*5*

Move	Token	stack
1	2	2
2	3	23
3	4	234
4	+	27 (3+4=7)
5	*	14 (2*7=14)
6	5	145
7	*	70 (14*5=70)

যেক্ষেত্রে operand এর চেয়ে operator একটি কম থাকবে। operand পেলে তাকে stack এ push করতে হবে, আর operator পেলে সেটি top most 2টি element নিয়ে উপর কাজ করবে

Consider the following infix expression P

$P = A+(B/C-(D*E\wedge F)+G)*H$

Character Scanned	stack	Postfix expression
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
/	(+(/	AB
C	(+(/	ABC
−	(+(-	ABC/
(	(+(-(	ABC/
D	(+(-(	ABC/D
*	(+(-(*	ABC/D
E	(+(-(*	ABC/DE
∧	(+(-(*∧	ABC/DE
F	(+(-(*∧	ABC/DEF
)	(+(-	ABC/DEF∧*
+	(+(+	ABC/DEF∧*−
G	(+(+	ABC/DEF∧*−G
)	(+	ABC/DEF∧*−G+
*	(+*	ABC/DEF∧*−G+
H	(+*	ABC/DEF∧*−G+H
)		ABC/DEF∧*−G+H*+

→ প্রথম step এ stact এ একটি extra "(" এবং Equation এর শেষে একটি ")" দিতে হবে।

→ operand পেলে expression এ সরাসরি push হবে

→ "(" " stact " push হবে

→ ")" " এর left এর "(" এর আগের পর্যন্ত সব expression এ চলে যাবে।

→ Operator এর ক্ষেত্রে: প্রথমত stack এ থাকবে, আর stact এ গিয়ে যদি পাশাপাশি আরেক operator পায় তাহলে তাদের precedence check করতে হবে, যার precedence সমান অথবা বেশি হলে expression এ চলে যাবে, আর যার কম সে stack এ থেকে যাবে।

Queue

A queue is a linear list of elements in which deletion can take place at one end, called the front, and insertions can take place only on the other end called the rear. The terem "front" and "rear" are used in describing a linear list only when it is implemented as a queue.

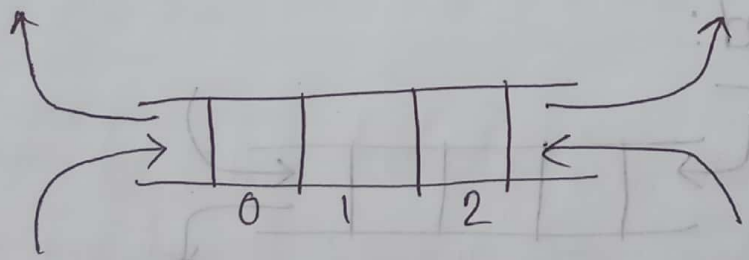
Also called First In - First Out (FIFO) since of the first element in a queue will be the first element out of the queue

Application of queue

i. Widely used ~~in~~ as waiting list for a single shared resource like printer, disc, CPU

(ii) Used to maintain the play list in media players in order to add and remove the songs from the play list.

(iii) Used to operating system to handle interrupts.



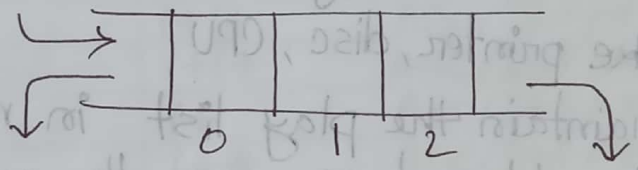
Deques

Deques : A deque (pronounced either "deck" or "dequeue") is a liniar list in which elements can be added or remove at either end but not in the middle. The term deque is a contraction of the name double-ended queue

Two types of deques: (i) input restricted
(ii) output "

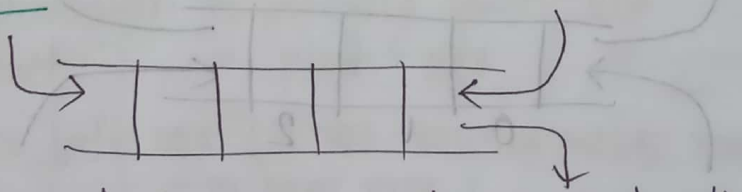
which are intermediate betwee a deque and a queue.

Input Restricted :



an input restricted deque is a deque which allows insertion at only one end of the list but allows deletion at both ends of the list.

Output Restricted :



an output restricted deque is a deque which allows deletion at only one end but allows insertion at both ends of the list.

Priority Queue

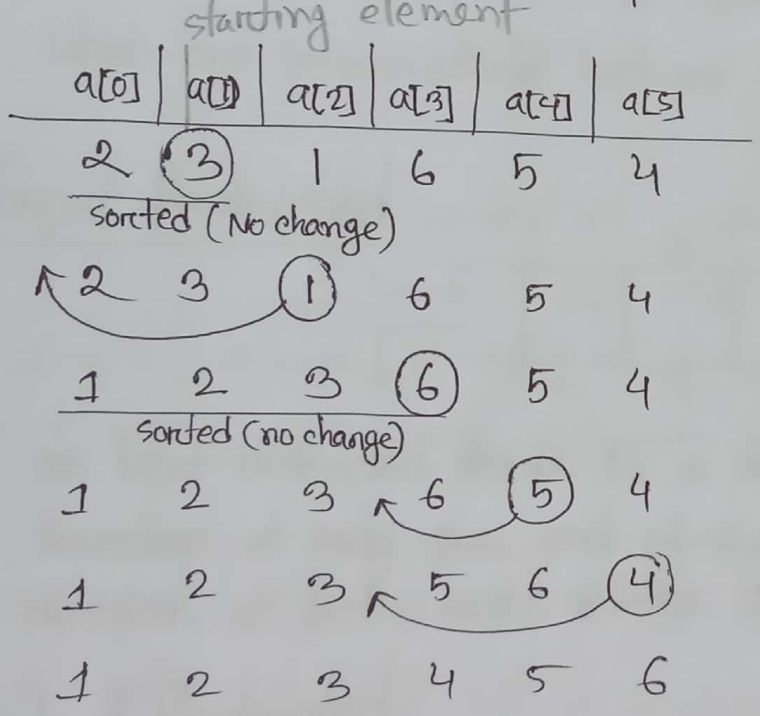
A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following roles:

1. An elements with the same priority are processed according to the order in which they were added to the queue.

Sorting Algorithm

Insertion Sort:

Let's a[0], a[1], a[2] ..... a[n-1] in an array. Insertion sort algorithm scans the element from a[1] to last element and inserting each element into it's proper position in the previously sorted subarray.



প্রতিটি element তার পূর্বের element-সমূহের অর্ডার থেকে সঠিক পজিশনটি বেছে নিয়ে সেই পজিশনে insert হবে আর সেই পজিশনের জানের element সমূহের পজিশন 1 করে বাড়বে।

Algorithm:

1. set a[0] = -.
2. Repeat step 3 to 5 for i = 1, 2, 3 ... N-1.
3. set temp = a[i], j = i-1
4. Repeat while: j >=0 && A[j] > temp
   (a) set a[j+1] = a[j] [moves element forward]
   (b) j--
   [end of loop]
5. set a[j+1] = temp // Insert elements in proper place.
   [end of step 2 loo]
6. Return

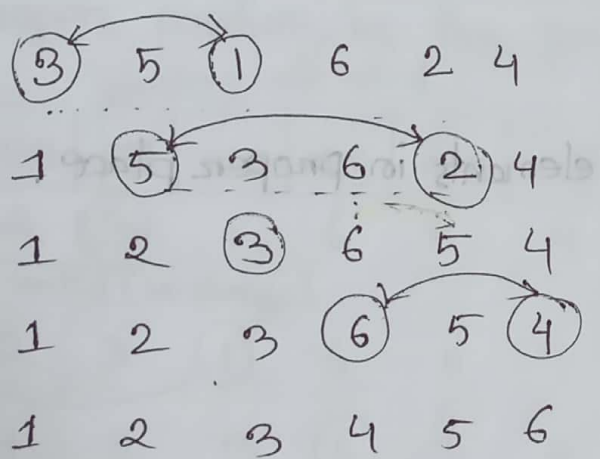
Complexity: Worst case:
Average case:

```
void insertion_sort (int A[], int n){
        int i, j, item;
        for (int i=1; i<n; i++){
            temp = a[i], j=i-1;
            while (j>=0 && a[j] > temp){
                A[j+1] = a[j];
                j--;
            }
            a[j+1] = temp;
        }
}
```

Selection Sort :

a[0], a[1] ..... a[n-1] is an array.

firstly find the smallest element in the list and put it in the 1st element (swap 1st element and smallest element). and secondly find 2nd lowest element then swap with 2nd element and so on.



সবচেয়ে ছোট element টি আগে খুঁজে বের করে 1st element এর সাথে swap করতে হবে। then 2nd, 3rd and so on

Algorithm

1. Repeat 2 to 4 for i=0, 1, 2 ..... n-2
2. set temp = i, j = i+1
3. repeat while j < n
   (a) if (a[j] < a[temp] then set temp = j;
   (b) j++; [end of loop]
4. If temp ≠ i then interchange a[i] and a[temp]
   [end of loop]
5. Exit

program:

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

// or swap (a[i], a[temp])

Complexity: $n+(n-1)+(n-2)+(n-3)+\cdots\cdots+1$

$= n(n-1)/2$

$= \frac{1}{2}(n^2-n$

$= \frac{1}{2}O(n^2-n)$

$\Rightarrow O(n^2)$

Quick Sort: (An Application of stacks)

Quick sort is an algorith of the devide-and-conquer type. That is the problem of sorting a set is reduced to the problem of sorting two smaller sets.

Lets a[] = {1, 5, 6, 3, 8, 4, 7, 2} be an array.

0	1	2	3	4	5	6	7

ব্যাখ্যাঃ কোনো একটি সংখ্যাকে pivot হিসেবে ধরে তাকে তার যথাস্থানে (অর্থাৎ তার ডানদিকের সংখ্যাগুলো তারচেয়ে বড় এবং বামদিকে গুলো তার চেয়ে ছোট হবে) বসাতে হবে। তাহলে এই কাজটি প্রতিবার করলে ২টি partition তৈরি হবে। প্রথমত array-র প্রথম element কে low এবং শেষ element কে high ধরে নিতে হবে এবং প্রতিবার a[high] কে pivot ধরতে হবে।

(1) 5 6 3 8 4 7 (2)

low — high & pivot

index →	0	1	2	3	4	5	6	7
	1	2	5	6	3	8	4	7

(partition 1) — partition 2

high = 7 =highi

low = 1+1 = 2

low 1 করে বসবে

low = low = 0

high = 1-1 = 0

high 1 করে বসবে।

first

(44) 33 11 55 77 90 40 60 99 22 88 (66)

last

pivot

Algorithm:

1. Set low = 0 and high = n-1
2. Repeat a to h while low < high
   (a) ~~Repeat while~~ set i = low, j = low, pivot = a[high]
   (b) Repeat c to d while j < high
   (c) If a[j] <= pitot then exchange a[i] and a[j] and i++
   (d) j++
   (e) exchange a[high] and a[i]
   (f) set p = i
   (g) set low = p, high = n-1 and repeat a to e
   (h) set low = 0, high = P and repeat a to e.
   [end of loop]
3. return
4. Exit

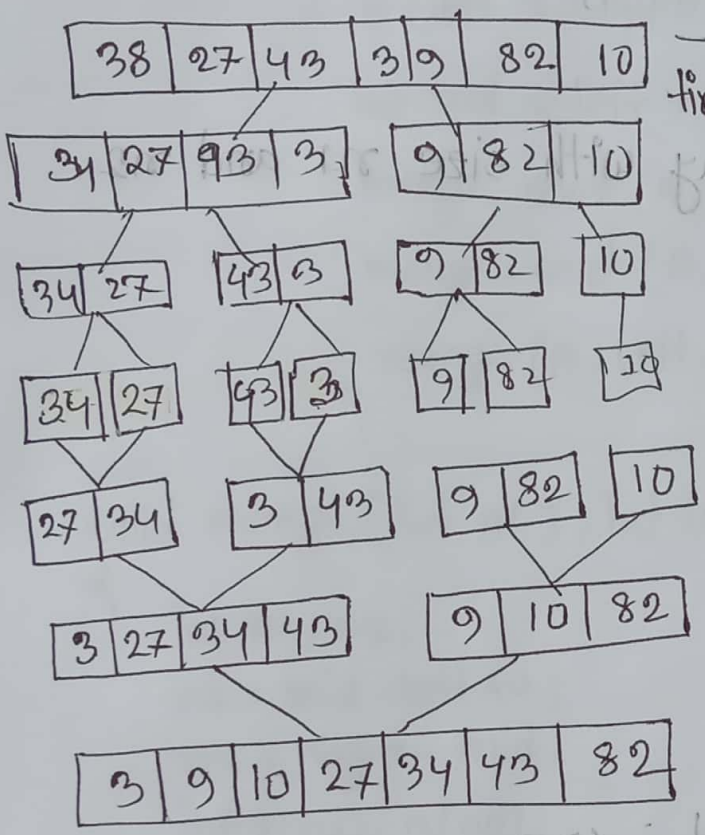
```
int Partition (int a[], int low, int high)
{
    int i, j, temp, pivot;
    pivot = a[high];
    for (i=low, j=low; j<high; j++)
    {
        if (a[j] <= pivot)
        {
            temp = a[j];
            a[j] = a[i]
            a[i] = temp;
            i++
        }
    }
    temp = a[high];
    a[high] = a[i];
    a[i] = temp;
    return i;
}
void quick_sort (int a[], int low, int high)
{
    int i;
    if (low < high)
    {
        i = partition (a, low, high);
        quick_sort (a, low, i-1);
        quick_sort (a, i+1, high);
    }
}
```

// swap (a[i], a[j])

// swap (a[high], a[i])

Merge Sort Algorithm

Devide and conquire Algorithm



time complexity: $O(n) \times \log_2 n$

element no. এর জন্য. — Level এর জন্য

Merge_Sort: (A, left, Righ) where A is an array left is the first element index and right is the last index.

1.

Merge: [A, left, mid, right]

1. set $n1 = mid - left + 1$, $n2 = right - mid$

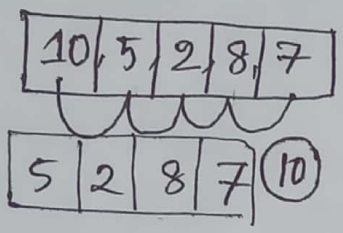
   [Here, A be an array]

2. Let l and r be two array with size $n1$ and $n2$.
3. set, $i=0, j=0$
4. while $(i < n1)$ repeat:
   (a) $l[i] = A[left + i]$
   (b) $i = i+1$
5. while $(j < n2)$ repeat:
   (a) $r[i] = A[mid+1+j]$
   (b) $j = j+1$
6. set $i=0, j=0, k=left$
7. Repeat while $i < n1$ and $j < n2$
   (a) If: $l[i] <= r[i]$ set $A[k] = l[i]$, $i = i+1$
   else: $A[k] = r[j]$, $j = j+1$
   (b) $k = k+1$
8. while $i < n1$ repeat:
   (i) $A[k] = l[i]$
   (ii) $i = i+1$, $k = k+1$
9. while $j < n2$ repeat:
   (i) $A[k] = r[j]$
   (ii) $j = j+1$, $k = k+1$
10. Return

```
void merge_sort(int A[], int left, int right){
        if (left >= right) return ;
        int mid = left +(right-left)/2 —এভাবে দিলে কখনো data type overflow হবেনা
        merge_sort (A, left, mid)
        merge_sort (A, mid+1, right)
        merge (A, left, mid, right);
}

void merge (int A[], int left, int mid, int right)
{
    int n1, n2;
    n1 = mid - left + 1;
    n2 = right - mid;
    int l[n1], r[n2];
    for(int i=0; i<n1; i++) l[i] = a[left+i];
    for(int j=0; j<n2; j++) r[j] = a[mid+1+i];
    int i=0, j=0, k=left;
    while(i<n1 && j<n2)
        {  if(l[i]<=r[j]) a[k] = l[i], i++;
           else a[k] = r[j], j++;
           k++;
        }
    while(i<n1){ a[k] = l[i]
                 i++, k++; }
    while(j<n2){ a[k] = r[i]
                 j++, k++
               }
}
```

Bubble Sort (is not a practical sorting Algo)

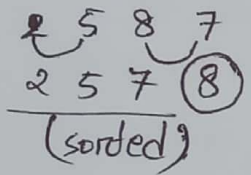


প্রথম থেকে প্রতিবার ২টি করে সংখ্যা তুলনা করে বড় সংখ্যাটি ২য় অবস্থানে ও ছোটটি ১ম অবস্থানে বসাবে।

10 সবার সাথে তুলনা করে সর্বশেষ অবস্থানে গিয়ে বসেছে।

এবার 10 এর হিসাব বাদ।

বাকী ৪ টা element এর মধ্যে আবার পর্যায়ক্রমে একইভাবে তুলনা করে বড়টি 10 এর আগের index এ বসাতে হবে।




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

complexity: $(n-1)+(n-2)+\cdots\cdots+2+1 = \frac{n(n-1)}{2} = \frac{n^2}{2} + O(n)$

$= O(n^2)$

Trees

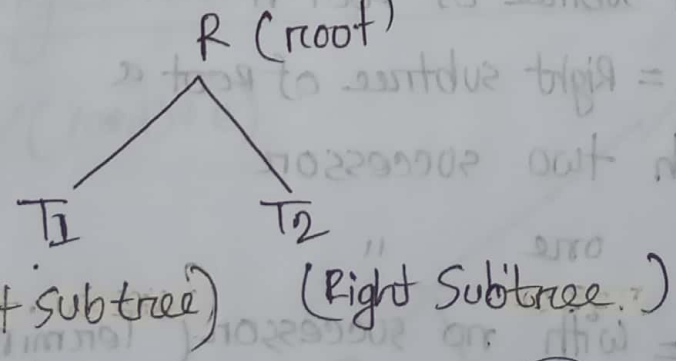
Trees is a nonlinear data structure. This structure is mainly used to represent data containing a hierarchical relationship between elements.

Binary Trees

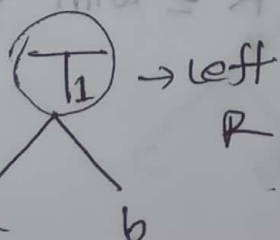
A binary tree T is defined as a finite set of elements called nodes such that:

1. T is empty (called the null tree or empty tree) or
2. T contains a distinguished node R, called the root of T and the remaing nodes of T form an ordered pair of disjoint binary trees T_1 and T_2

- If T does contain a root R then the two subtrees T_1 and T_2 are called respectively left and right subtree of R.



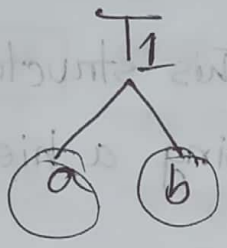
- If T_1 is non empty: Then



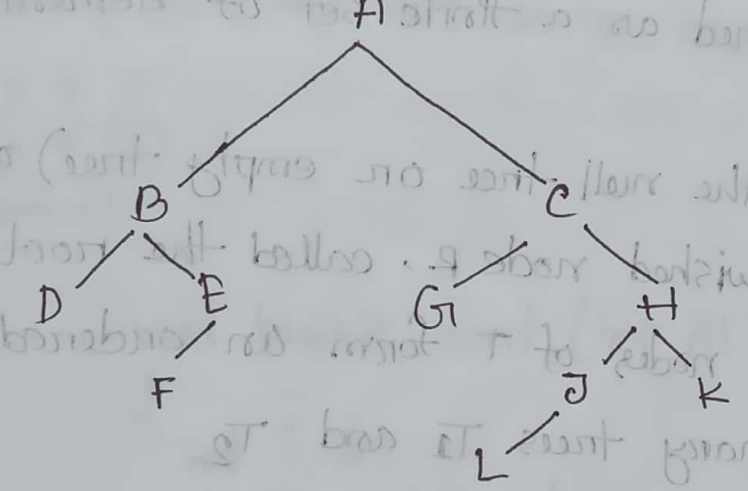
→ left successor of R

Right successor of R

• If a node with no successors is called terminal node



a and b both are terminal node.



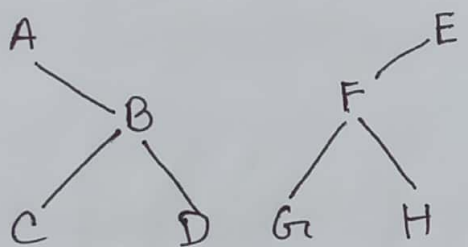
A = root
B = left successor of A
C = Right " " "
B, D, E, F = left subtree of Root A
C; G1, H, J, K, L = Right subtree of Root a
A, B, C, H = with two successor
E, J = " one "
D, F, G1, L, K = with no successor (Terminal node.)

Similar Binary Trees:

T and T' said to be similar if they have the same structure (or shape)

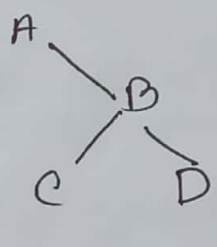
Copies Binary Trees

T and T' said to be copies if they are similar and they have the same contents as corresponding nodes

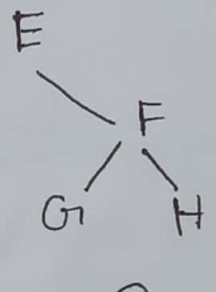


ⓐ

ⓑ



ⓒ



ⓓ

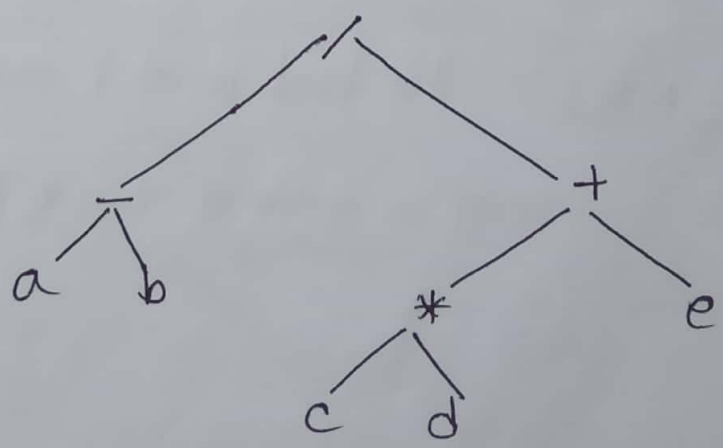
similar: ⓐ, ⓒ, ⓓ

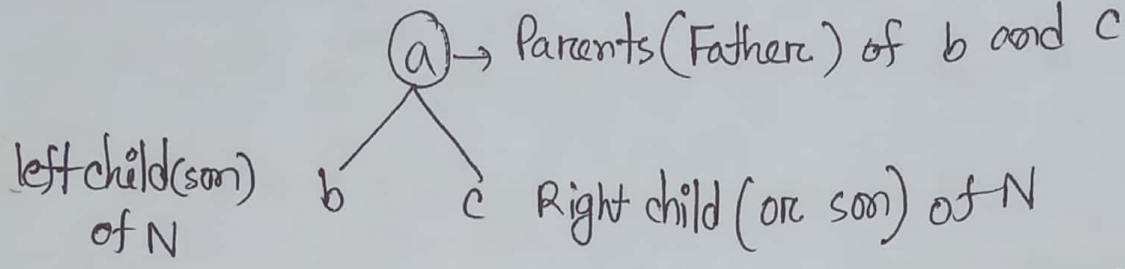
copies: ⓐ, ⓒ

Note: ⓑ is not similar or copies of anyone because the shape of ⓑ is different from ~~the~~ others.

Algebric Expressions

E = (a-b)/((c*d)+e)



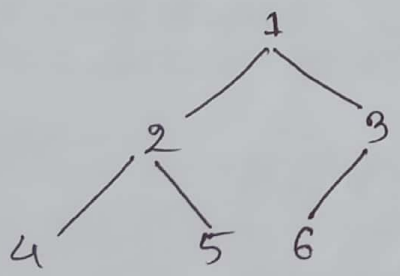


- b and c are siblings (or brothers)

Every node except root has a unique parent called the predecessor of N

Complete Binary Tree:

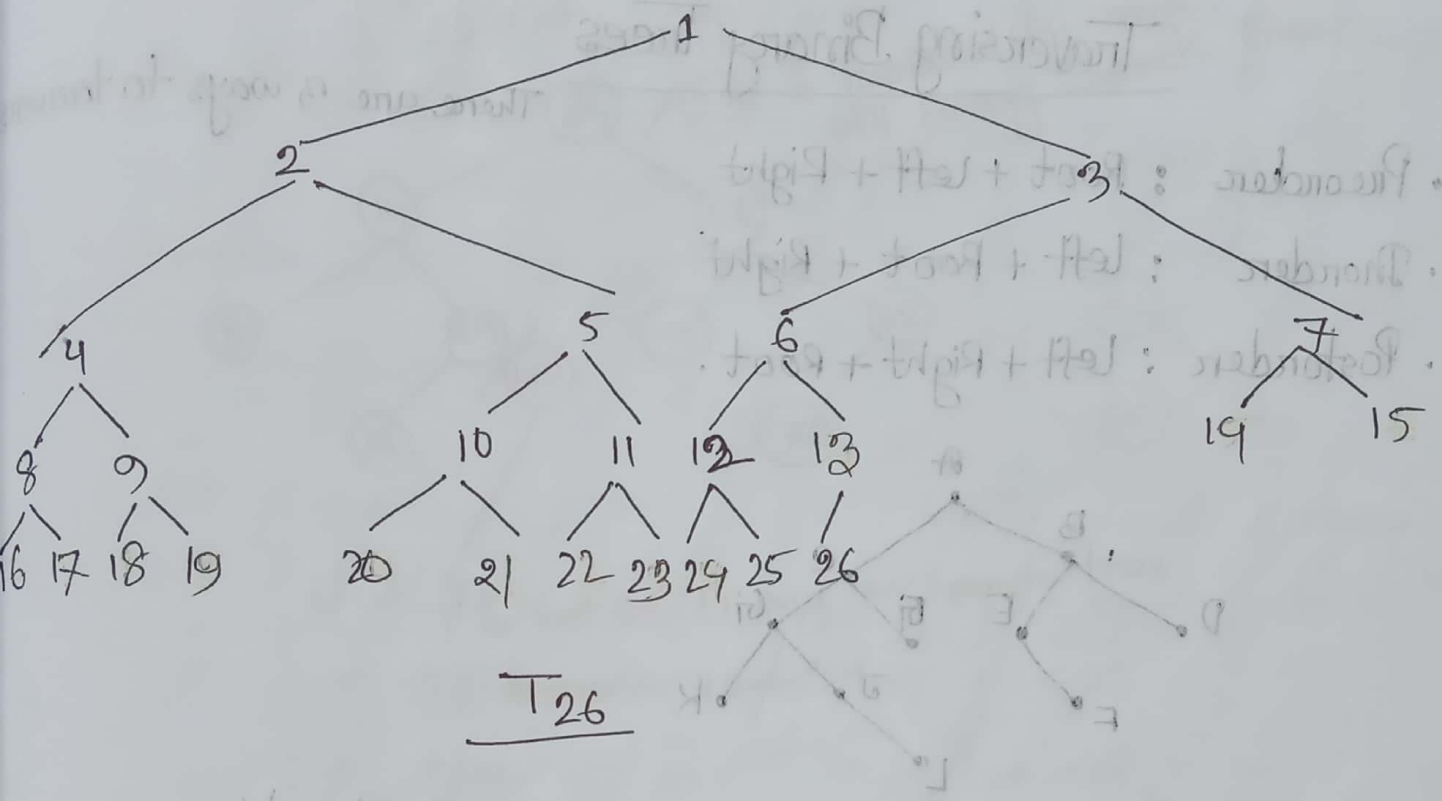
If any binary Tree is T then T is called complete Binary tree if each node of T has ~~either 0 or~~ atmost 2 T childreen. and all nodes at the last level as far left as possible.



Note: the level r of T can have at most 2^r nodes.

Level or depth of tree $D_n = \lceil \log_2 n + 1 \rceil$

where n = nodes number



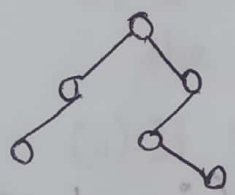
T_{26}

Find level: (of T_{26})

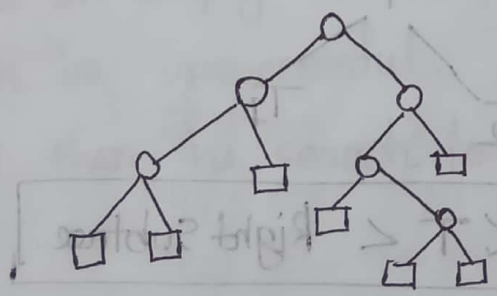
$\therefore D_n = \log_2 n + 1 = 5$

Extended Binary Tree: (2-Trees)

A binary tree T is said to be a 2-tree or an extended binary Tree if each ~~of~~ node of N has either 2 or 0 children. The nodes with 2 children called internal nodes and the nodes with 0 children are called external nodes.



Binary Tree



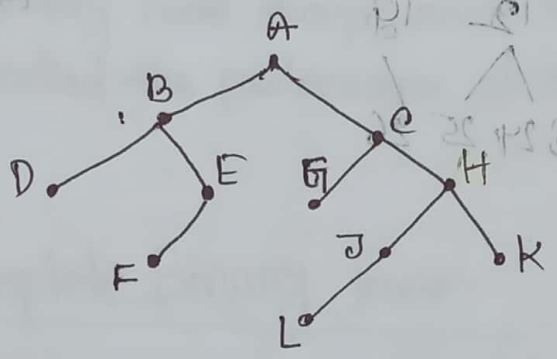
Extended 2-tree

New node are external node in the extended tree.

Traversing Binary Trees

There are 3 ways to traverse

- Preorder : Root + Left + Right
- Inorder : Left + Root + Right
- Postorder : Left + Right + Root.



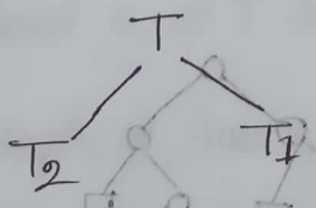
Preorder Traverse: A B D E F C G H J L K

Inorder " : D B F E A G C L J H K

Post order " : D F E B G L J K H C A

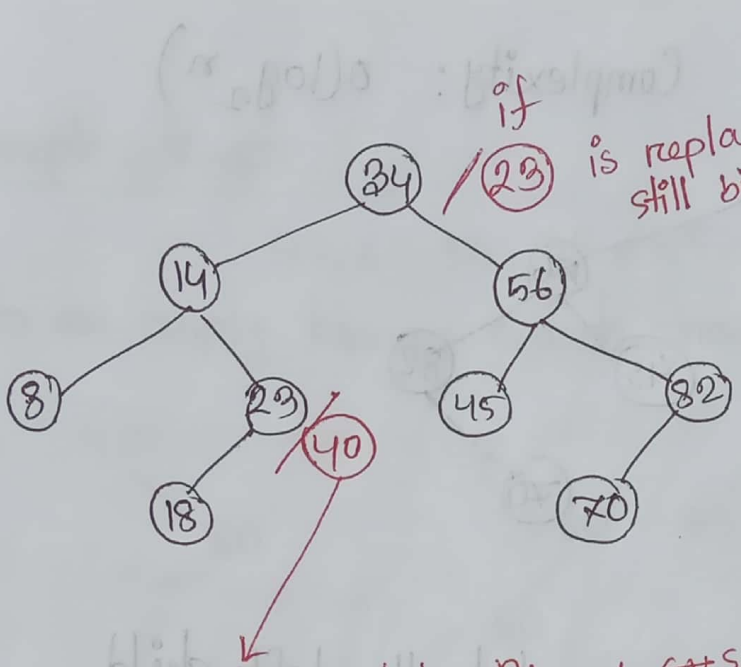
Binary Search Tree

Let, T is a binary Tree. T is call a binary search tree or binary sorted tree if each node N of T has the following property:



Left subtree < T < Right Subtree.

(inorder Traversal



if (23) is replaced by here; this would be still binary Search Tree

T is not be Binary ~~s~~ search tree because 40 ≠ 34

Searching any element in Binary search Tree:

~~Algorithm~~ Proceed:

~~(a) compare~~ Let find 'x' Find from Binary search Tree

(a) Compare x with root node N of the tree.

(i) If $x < N$ then proceed to the left child

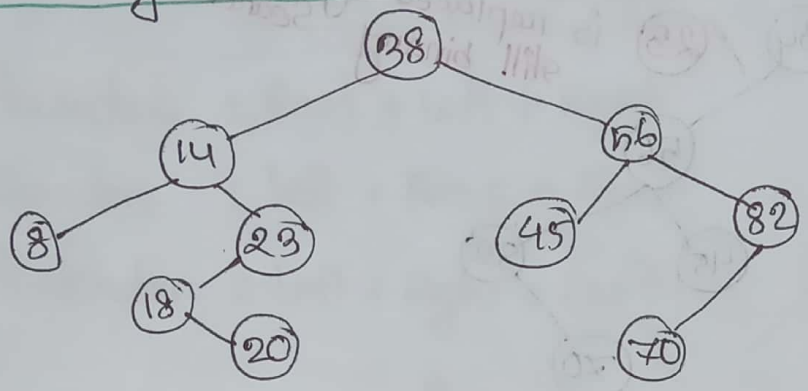
(ii) If $x > N$ then proceed to the right child.

(b) Repeat step (a) untill ~~x=N.~~ the following occurs:

(i) If we meet an empty subtree and $x \neq N$ then the search is unsuccessful.

(ii) If $x = N$ then the search is successful.

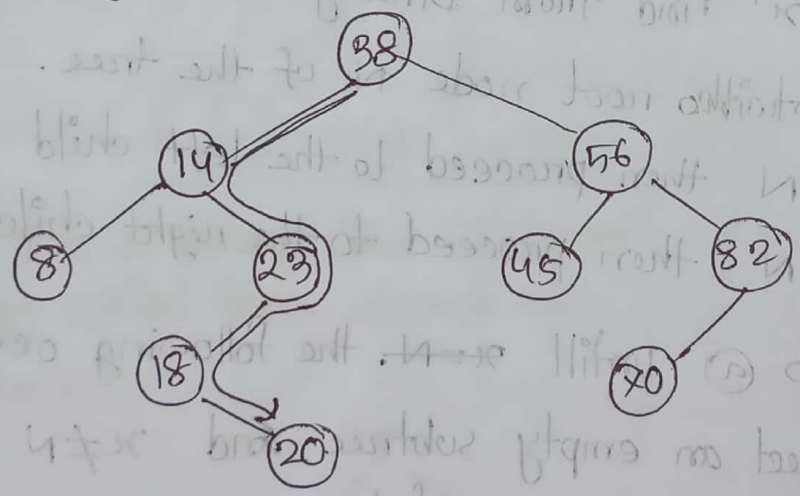
Searching element Complexity: $O(\log_2 n)$



Searching item = 20 :

20 < 38 processed to the left child
20 > 14 proceed " " Right "
20 < 23 " " " left "
20 = 20 (item) item found.

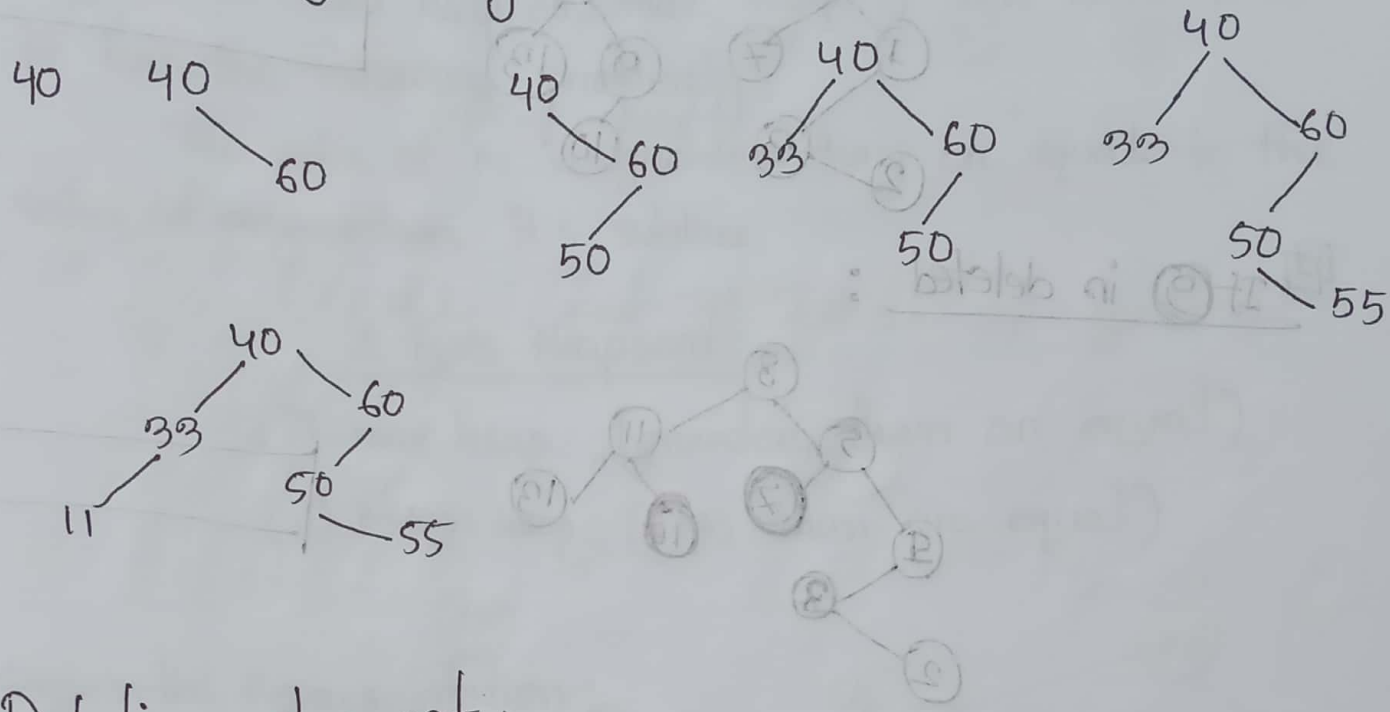
Inserting element: (inserting 20)



20 < 38 → left
20 > 14 → Right
20 < 23 → left
20 > 18 → Right

Example : 7.15

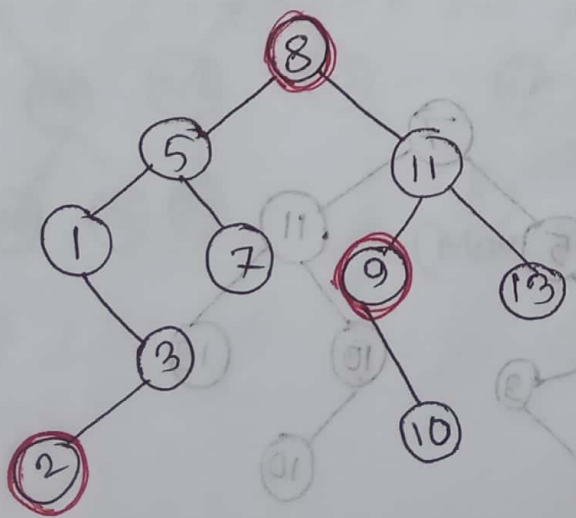
40, 60, 50, 33, 55, 11 inserting them into an empty binary search tree.



Deleting element :

if N is deleted : (3 case)

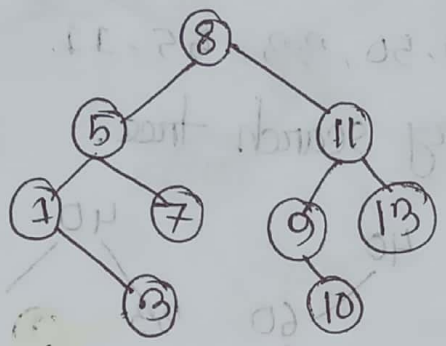
if (i) N has no children, n is easily deleted
(ii) N has 1 " : replacing that children in the location of N
(ii) If N has 2 children : find the nearest value of N and replace.



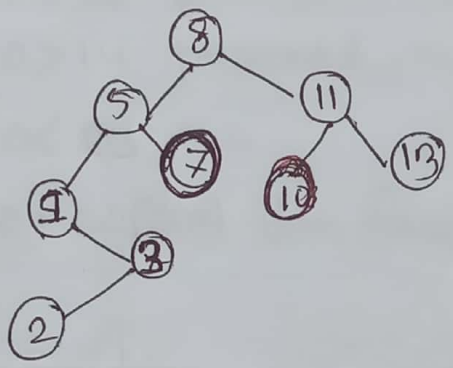
inorder:

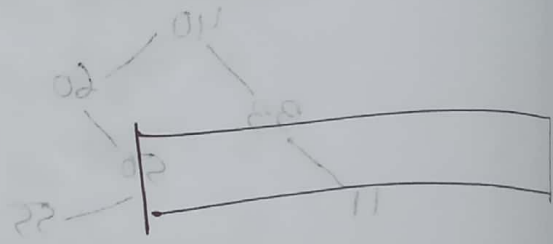
1 2 3 5 7 8 9 10 11 13

if (2) is deleted:



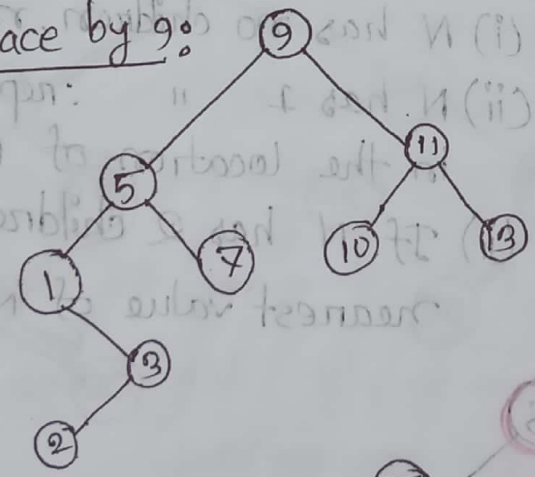
If (9) is deleted:

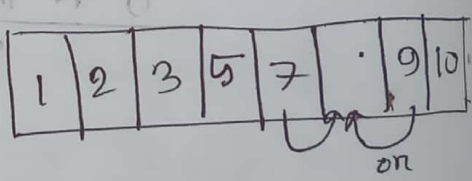


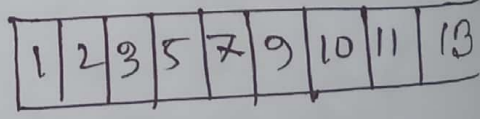


if (8) is deleted:

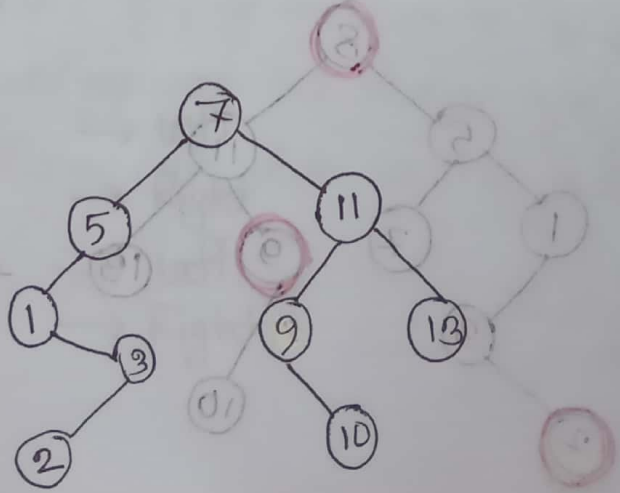
replace by 9:







or, replace by 7:



Heap Sort Algorithm :

Suppose H is a complete binary tree with n elements Than H in called heap or max Heap if each node N of H has the following property:

The value of N in greater than or equal to the value of ~~any other~~ it's subtree.

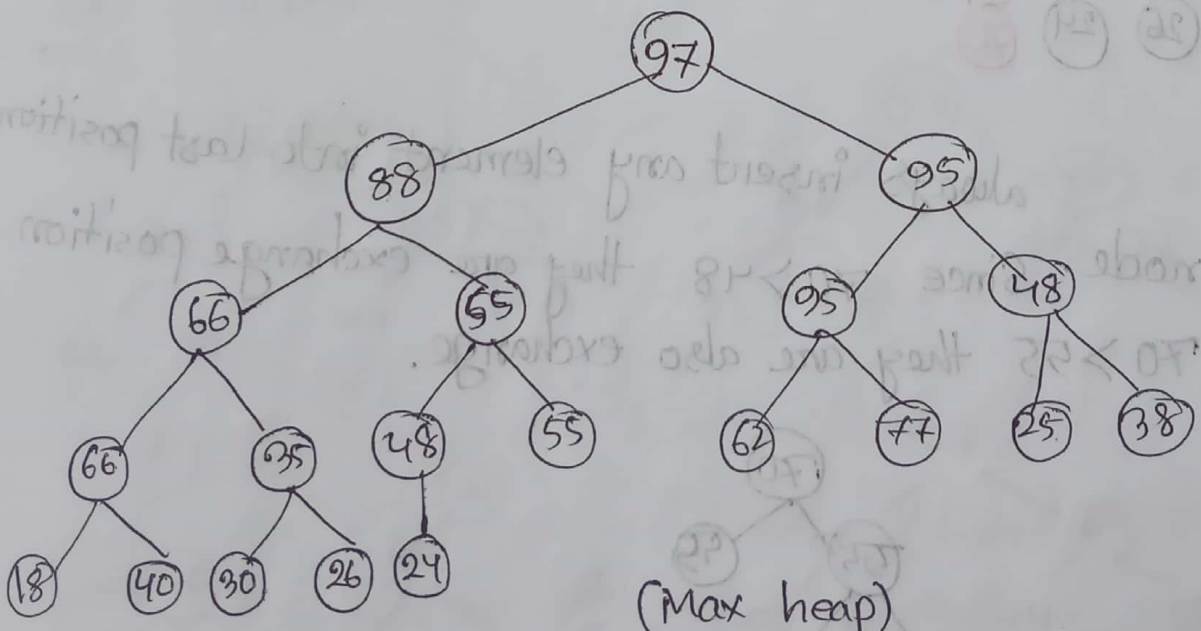
2 types Heapsort:

(i) max heap : (greater than or equal)

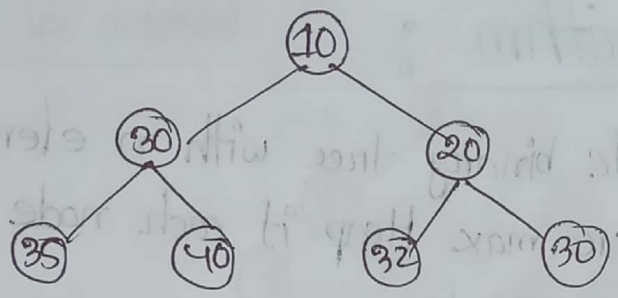
(ii) mean heap : (less than or equal)

Sequential Representation :

97 88 95 66 55 95 48 66 35 48 55 62 77 25 38 , 18, 40, 30
26, 24



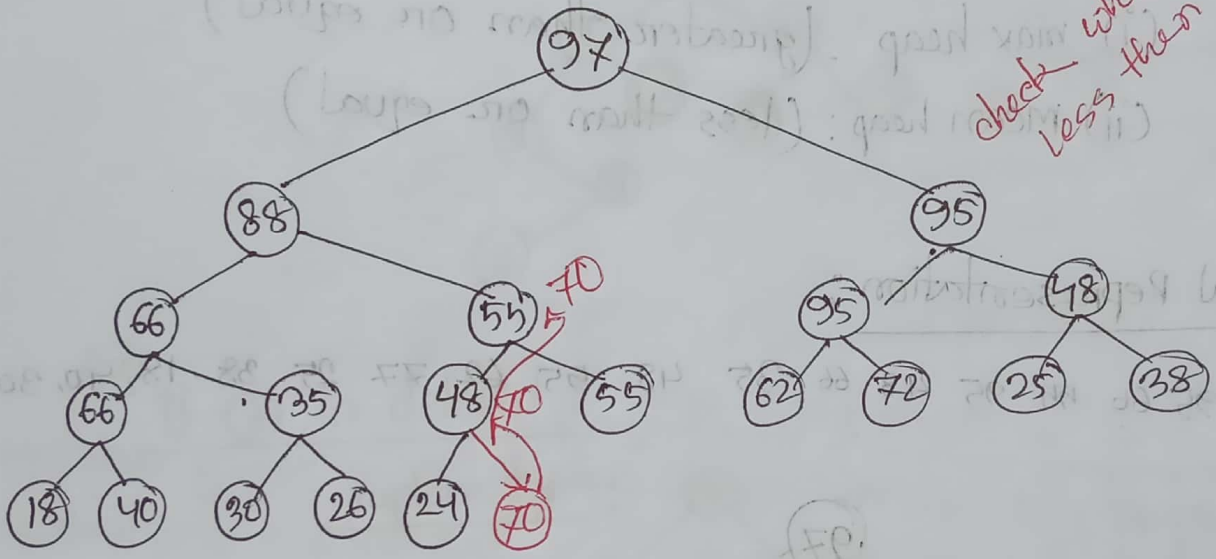
(Max heap)



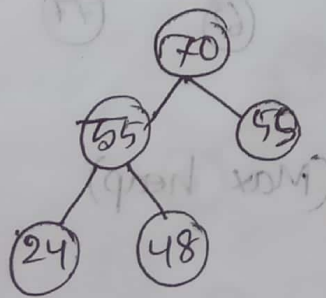
(Mean heap)

Inserting 70 to the 1st tree: O(log n)

check while parents is less than childreen.

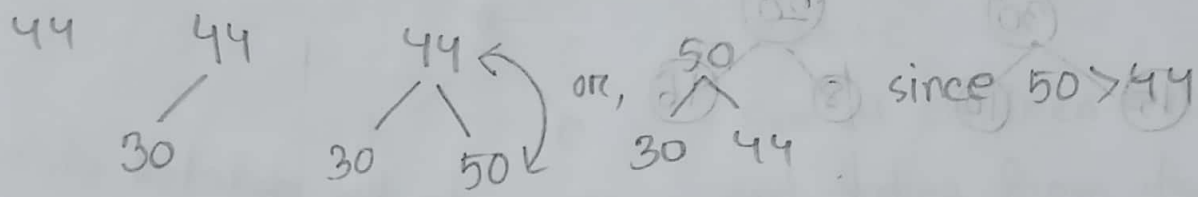


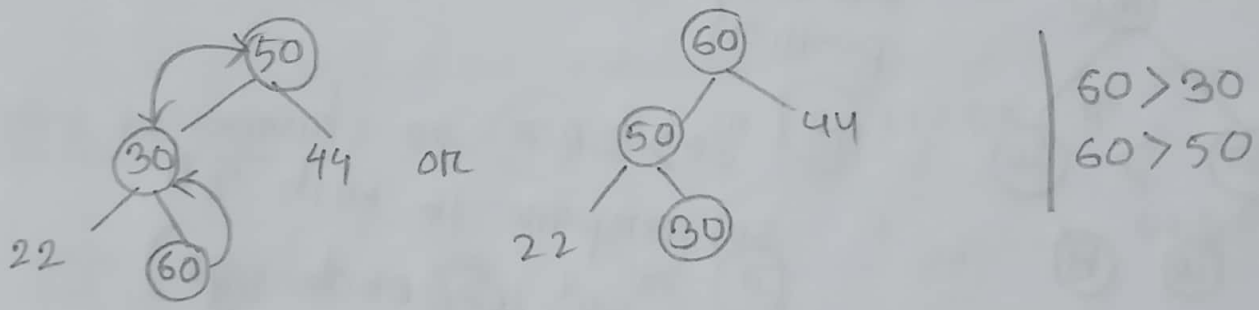
always insert any element into last position of the node. Since 70 > 48 they are exchange position. Again 70 > 55 they are also exchange.

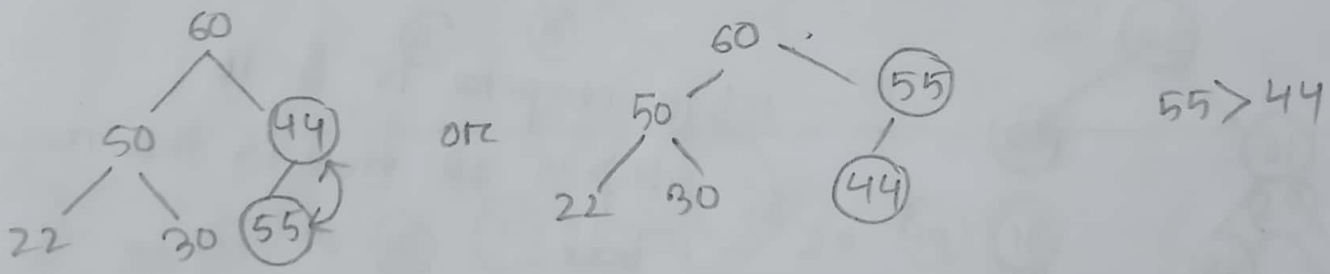


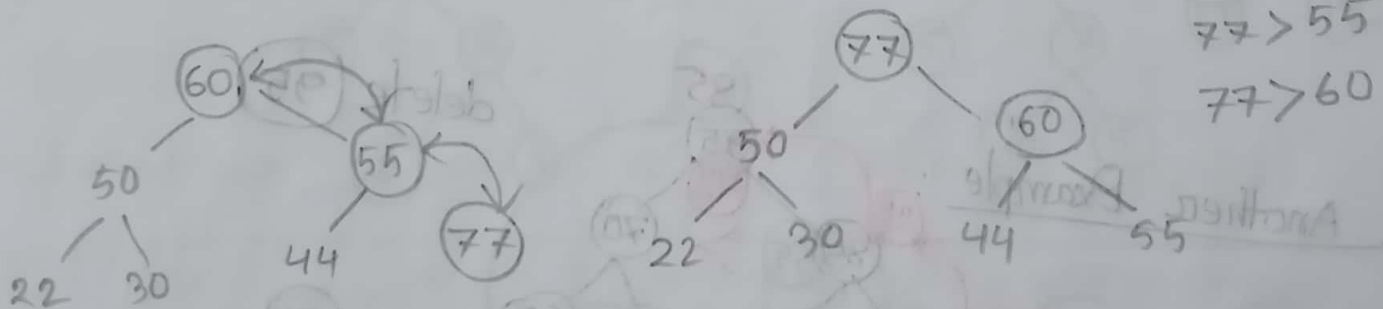
" Build a heap H "

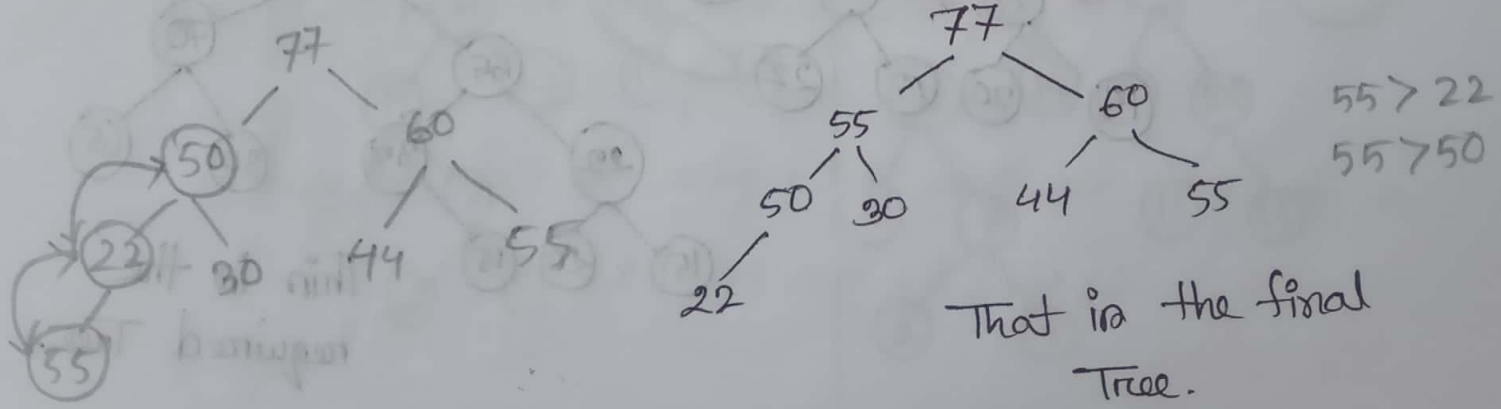
list of number 44, 30, 50, 22, 60, 55, 77, 55





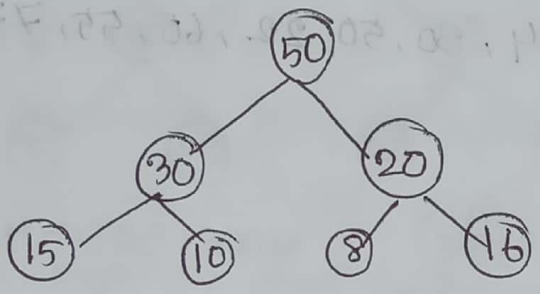




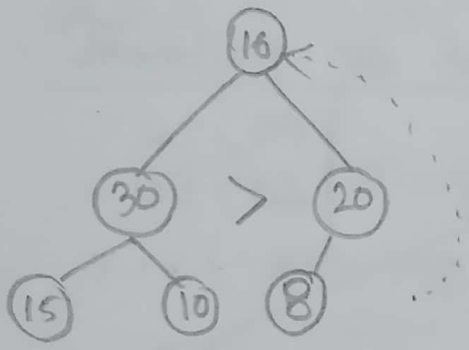


That is the final Tree.

Delete Element: (Delete 50 from the tree):



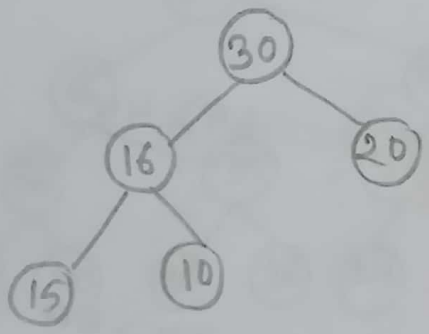
last element will go to the deleted element position



30 > 20

so, compare 16 with 30

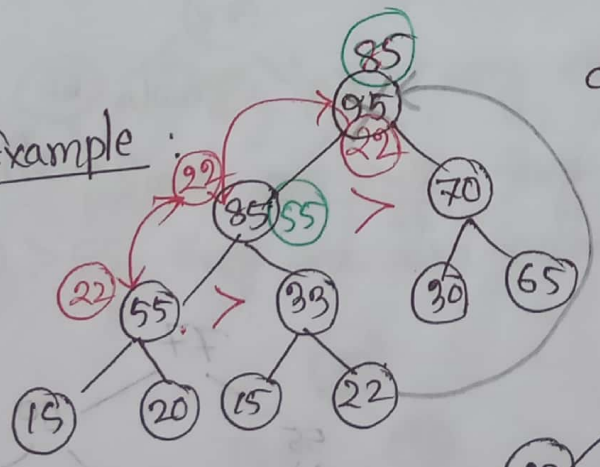
30 > 16 (exchange)



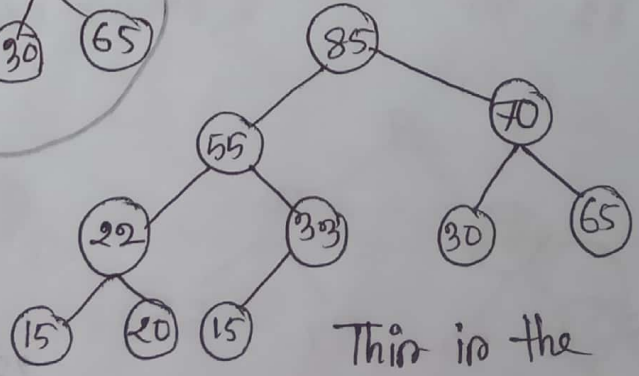
15 > 10 compare 15 & 16

16 ≯ 15 so this is the required Tree.

Another Example:



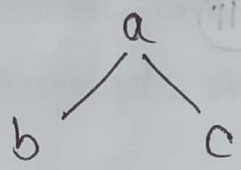
delete (95)



This is the required Tree.

HUFFMAN Algorithm

An extended binary Tree (which has either 0 or 2 children) which:



$a = b + c$

Sum of children = root

and finding from the minimum pair.

Let's be element: 22 5 11 19 2 11 25 5

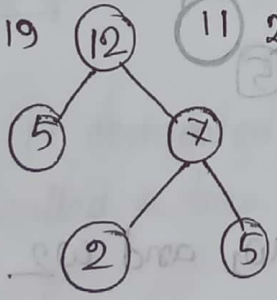
1

22 11 19 7 11 25 5

5+2=7

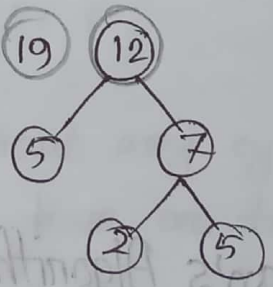
2

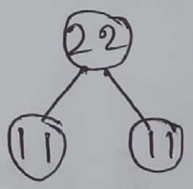
22 11 19 12 11 25



3

22 19 12 11 25

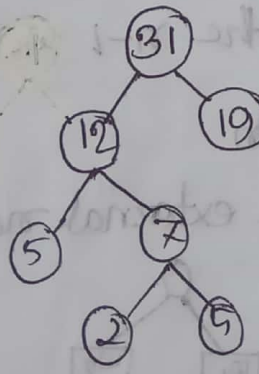


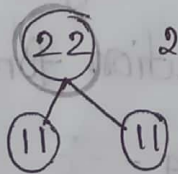


25

4

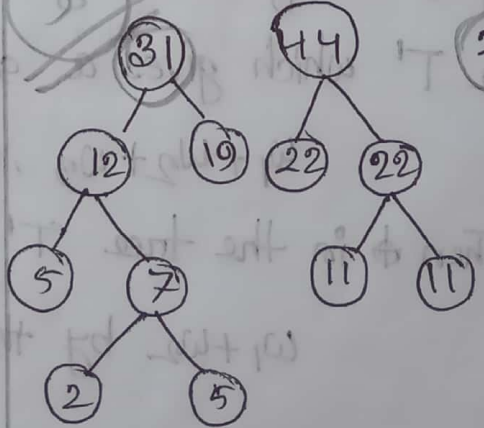
22





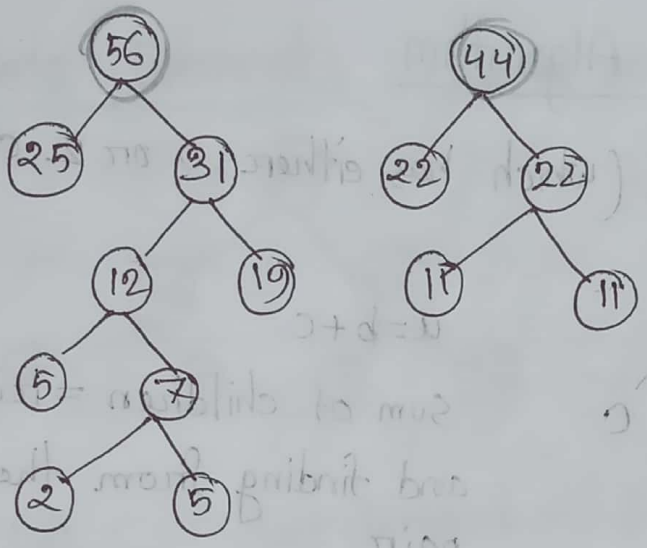
25

5

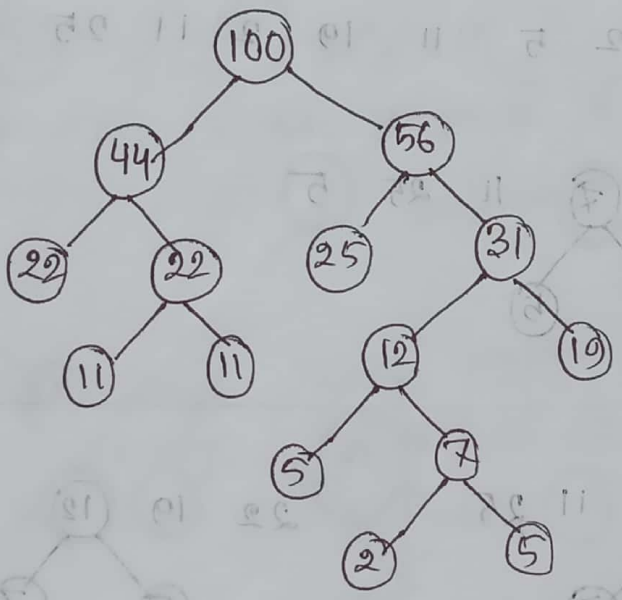


25

6



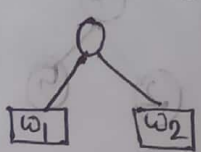
7



Hulfman's Algorithm: Suppose w_1 and w_2 are two minimum weights among the n given weight $w_1, w_2, \ldots w_n$. Find a Tree T' which gives a solution for the n-1 weights.

$w_1 + w_2, w_3, w_4 \ldots w_n$

Then in the tree T', replace the external node $w_1 + w_2$ by the subtree



Graph

A graph $G = (V, E)$ consists of V, a nonempty set of vertices (or nodes) and E, a set of edges.
Each edge has either one or two vertices associated with it, called its endpoints. An edge is said to connect its endpoints.

Connected graph :

A graph G is connected if and only if there is a simple path between any two nodes in G

Complete graph :

A Graph G is said to be complete if every node u in G is adjacent to every other node v in G. Denoted by k_n

total edge = $n(n-1)/2$

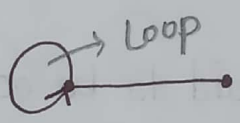
Tree graph : A connected graph T without any cycles is called a tree graph of free tree or tree.

Lebeled Graph: A graph G is said to be labeled if its edges are assigned data.

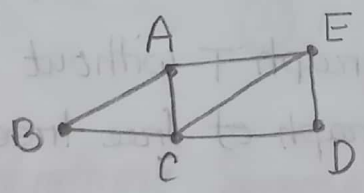
In particular G is said to be weighted if each edge e in G is assigned a nonnegative numerical value w(e) called the weight or length of e.

Multiple edge: Distinct edges e and e' are called multiple edges if they connect the same endpoint, that is e = [u, v] an e' = [u, v]. Multiple edge থাকলে multigraph বলে।

Loops: An edge e is called loop if it has identical endpoints if e = [u, u]

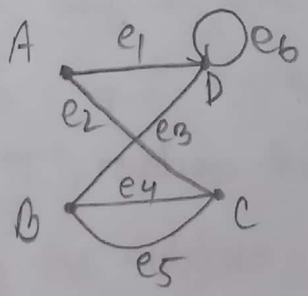


Example - 8.1 (a) making a connected graph with 5 nodes and 7 edges: [A, B], [B, C], [C, D], [E, E], [A, E], [C, E], [A, C]



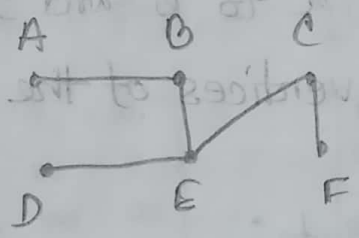
(graph)

(b) is not a graph but a multigraph, $e_4 = [B, C]$, $e_5 = [B, C]$ and has a loop $e_6 = [D, D]$

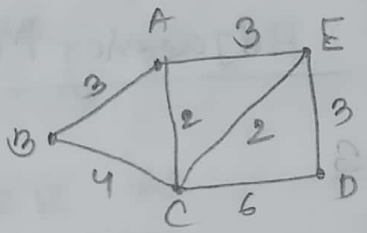


(multigraph)

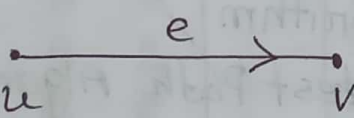
(c) is a tree with $m=6$ nodes and $m-1=5$ edges.
and there is unique simple path between any two nodes.



(d) is a weighted graph.



Directed graph: A directed graph (or digraph) (V, E) consists of a nonempty set of vertices V and a set of directed edges (or arcs) E. Each directed ~~graph~~ edge is associated with an ordered pair of vertices. The directed edge is associated with the ordered pair

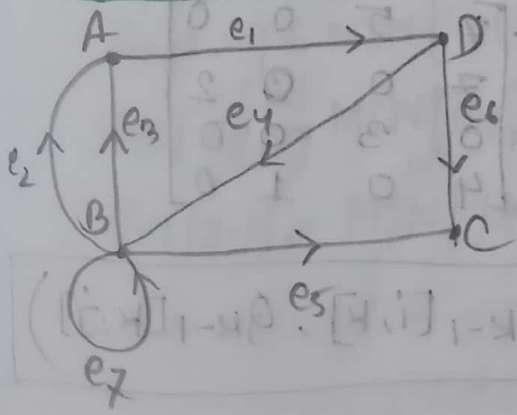


destination / terminal point of e

origin / initial point of e

begin at u and end at v

Example – 8.2



e_2, e_3 are parallel (begin and endpoint are same)

(DCBA ✗ path) Because · CB ✗

DBA ✓

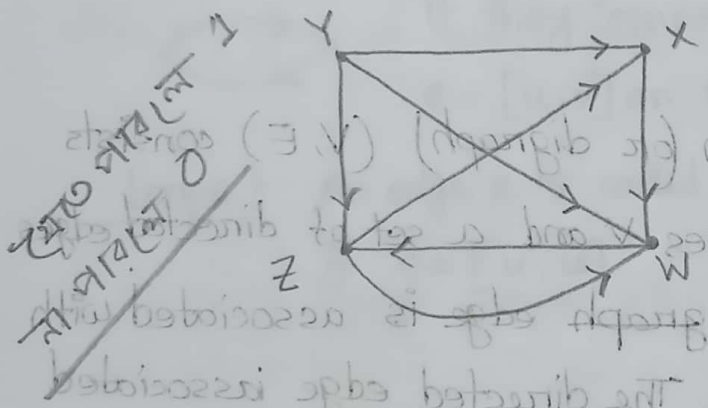
No path from C → so G is not strongly connected

e_7 is loop.

Strongly connected: A directed graph is strongly connected if there is a path from a to b and from b to a whenever a and b are vertices of the graph.

Adjacency Matrix

Example 8.3



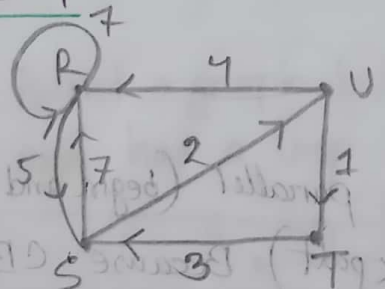
Adjacency Matrix, A =

	w	x	y	z
w	0	0	0	1
x	1	0	0	0
y	1	1	0	1
z	0	1	0	0

here, $v_1 = w$
$v_2 = x$
$v_3 = y$
$v_4 = z$

Warshall's Algorithm
Shortest Path Algorithm

Example 8.4



Assume $V_1 = R$
$V_2 = S$
$V_3 = T$
$V_4 = U$

$W =$

	R	S	T	U
R	7	5	0	0
S	7	0	0	2
T	0	3	0	0
U	4	0	1	0

Algo: $Q_K[i, j] = MIN(Q_{K-1}[i, j], Q_{K-1}[i, k], Q_{K-1}[k, j])$

$$Q_0 = \begin{bmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \infty & 1 & \infty \end{bmatrix} \qquad \begin{bmatrix} RR & RS & - & - \\ SR & - & - & SU \\ - & TS & - & - \\ UR & - & UT & - \end{bmatrix}$$

$$Q_1 = \begin{bmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & (9) & 1 & \infty \end{bmatrix} \qquad \begin{bmatrix} RR & RS & - & - \\ SR & - & - & SU \\ - & TS & - & - \\ UR & URS & UT & - \end{bmatrix}$$

RST

$$Q_2 = \begin{bmatrix} 7 & 5 & (\infty) & \infty \\ 7 & 12 & \infty & 2 \\ 10 & 3 & \infty & 5 \\ 4 & 9 & 1 & \infty \end{bmatrix} \qquad \begin{bmatrix} RR & RS & - & - \\ SR & SRS & - & SU \\ TSR & TS & - & - \\ UR & URS & UT & - \end{bmatrix}$$

$$Q_3 = \begin{bmatrix} 7 & 5 & \infty & 7 \\ 7 & 12 & \infty & 2 \\ 10 & 3 & \infty & 5 \\ 4 & (4) & 1 & 6 \end{bmatrix} \qquad \begin{bmatrix} RR & RS & - & RSU \\ SR & SRS & - & SU \\ TSR & TS & - & TSU \\ UR & UTS & UT & UTSU \end{bmatrix}$$

$$Q_4 = \begin{bmatrix} 7 & 5 & 8 & 7 \\ 7 & 12 & 3 & 2 \\ (9) & 3 & 6 & 5 \\ 4 & 4 & 1 & 6 \end{bmatrix} \qquad \begin{bmatrix} RR & RS & RSUT & RSU \\ SR & SRS & SUT & SU \\ TSUR & TS & TSUT & TSU \\ UR & UTS & UT & UTSU \end{bmatrix}$$

this is the shortest distance

$Q_1[4,2] = MIN[Q_0(4,2), Q_0(4,1)+Q_0(1,2)] = MIN(\infty, 4+5) = 9$

$Q_2[1,3] = MIN[Q_1(1,3), Q_1(1,2)+Q_1(2,3)] = MIN(\infty, 5+\infty) = \infty$

$Q_3[4,2] = MIN[Q_2(4,2), Q_2(4,3)+Q_2(3,2)] = MIN(9, 1+3) = 4$

$Q_4[3,1] = MIN[Q_3(3,1), Q_3(3,4)+Q_3(4,1)] = MIN(10, 5+4) = 9$

BFS – (Breadth-First Search)

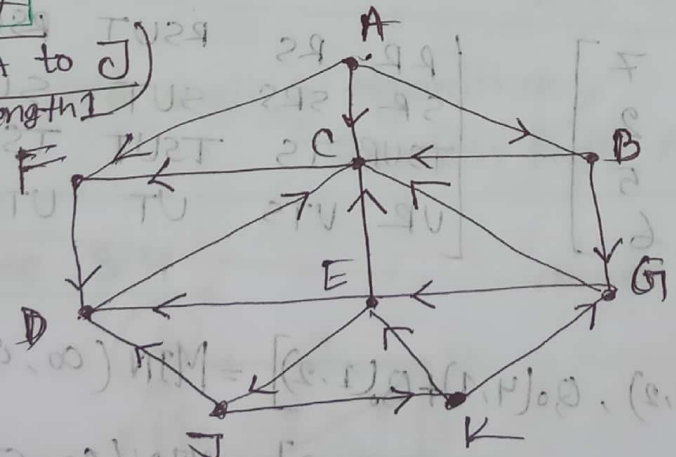
Discription: Firstly, Take a starting node from the graph. It can be any node. Then we examine all the neighbors of starting node. Then we examine all the neighbors of the neighbors of starting node and so on. No node is processed more than once. This is accomplished by using a queue to hold nodes that are waiting to be processed.

2 Terms:

1. Visiting a vertex
2. Exploration vertex (visiting Adjecency verte

Ex: 8.7.

(Want to fly A to J)
each edge has length 1



Adjacency lists

A: F, C, B
B: G, C
C: F, ~~D, E, G~~
D: C
E: D, C, J
F: D
G: C, E
J: D, K
K: E, G

Find the minimum path:

Note: when I select a vertex to visit, I should travers all the adjecency vertex of selected vertex.

QUEUE:

A	F	C	B	D	G	E	J	K

when find J, we stop as soon as

ORIG:

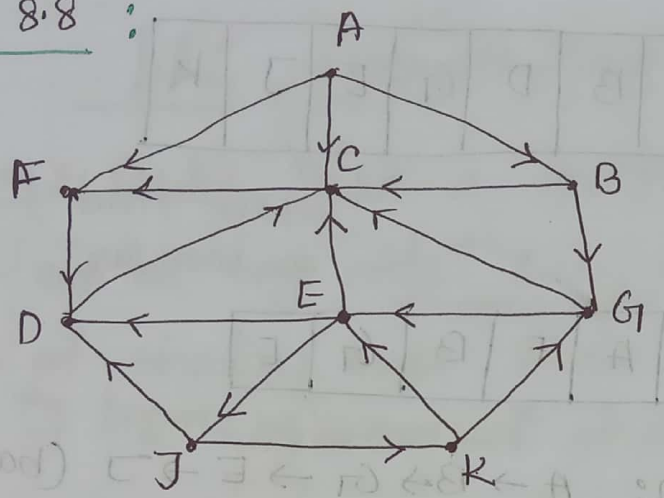
Ø	A	A	A	F	B	G	E

∴ required path: A → B → G → E → J (backtrack from J)

DFS (Depth First Search)

Firstly take a starting node from the graph. It can be any node. the process a neighbor of starting node, then a neighbor of neighbor of starting. Another way, The algorith start the root node (can be any node) and explores as far as possible along each branch before backtracking. This is accomplished by using STACK to hold nodes that are waiting to be processed.

Example 8.8 :



Want to find and print all the nodes reachable from the node J (including J itself)

Soln : starting node : J

STACK : | J | D | K | E | G | C | F |

stack
last in first out

steps: (i) Stack J

(ii) Print J, STACK D, K

(iii) Print K, STACK D, E, G

(iv) Print G, STACK D, E, C (E is already stack)

(v) Print C, STACK D, E, F

(vi) Print F, STACK D, E (F is already stack)

(vii) Print E, STACK D (C, D, G " " ")

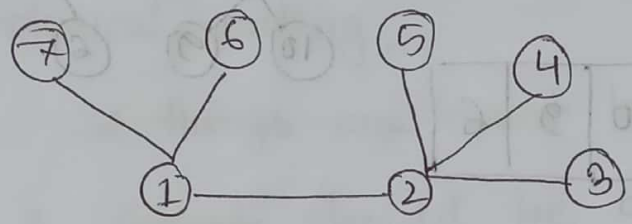
(viii) Print D, STACK (C " " a)

So, stack is now empty.

According the nodes: J, K, G, C, F, E, D are printed

Practice BFS & DFS

1

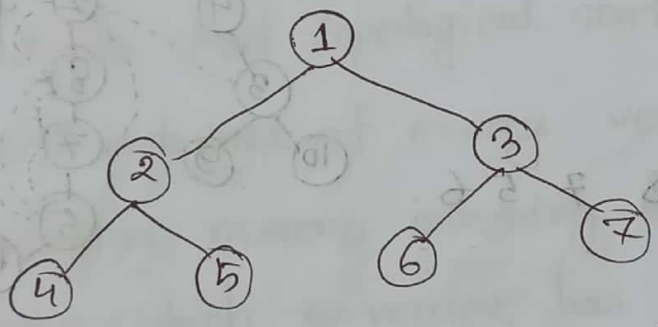


BFS: 1: 2, 6, 7
2: 3, 4, 5

so, 1, 2, 6, 7, 3, 4, 5

DFS: 1, 2, 3, 4, 5, 6, 7

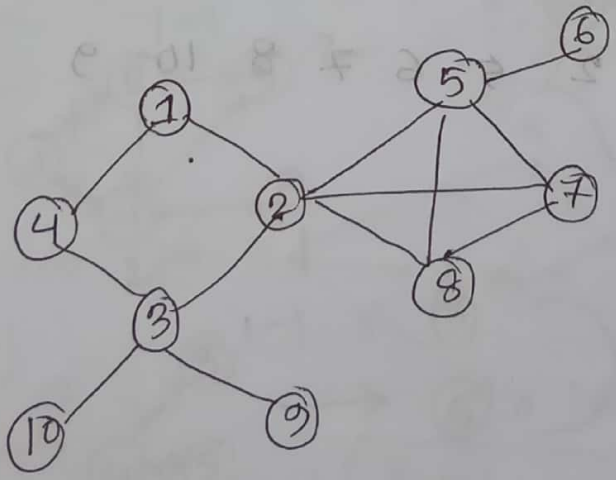
2



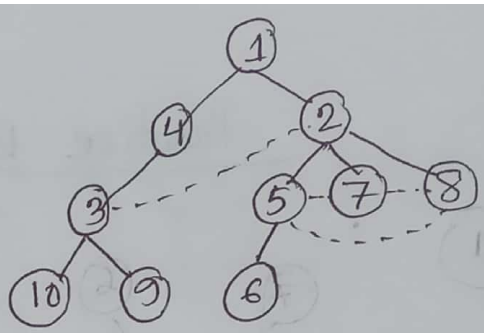
BFS: 1, 2, 3, 4, 5, 6, 7 (level order)

DFS: 1, 2, 4, 5, 3, 6, 7 (Pre order)

3



BFS spanning Tree:



QUEUE:

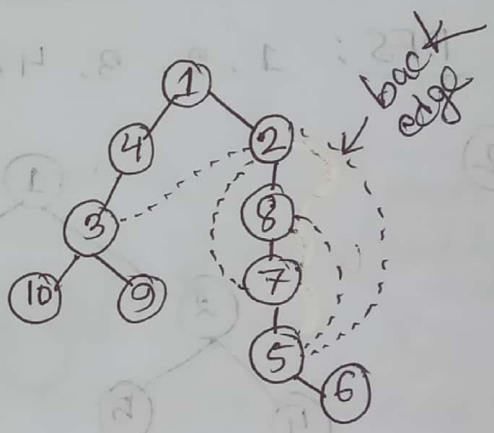
1	4	2	3	5	7	8	10	9	6

more valid Bfs:

(i) 1 2 4 8 5 7 3 6 10 9

(ii) 5 2 8 7 6 2 1 3 4 10

DFS spanning Tree:



Dfs: 1 4 3 10 9 2 8 7 5 6

more valid dfs:

(i) 1 2 8 7 5 6 3 10 9 4

(ii) 3 4 1 2 8 7 5 6 10 9

(iii) 3 4 10 2 5 6 7 8 10 9

Topological Sorting

For Topological sorting,

2 things are must be checked:

1. Graph should be directed
2. Graph should not contain cycle.

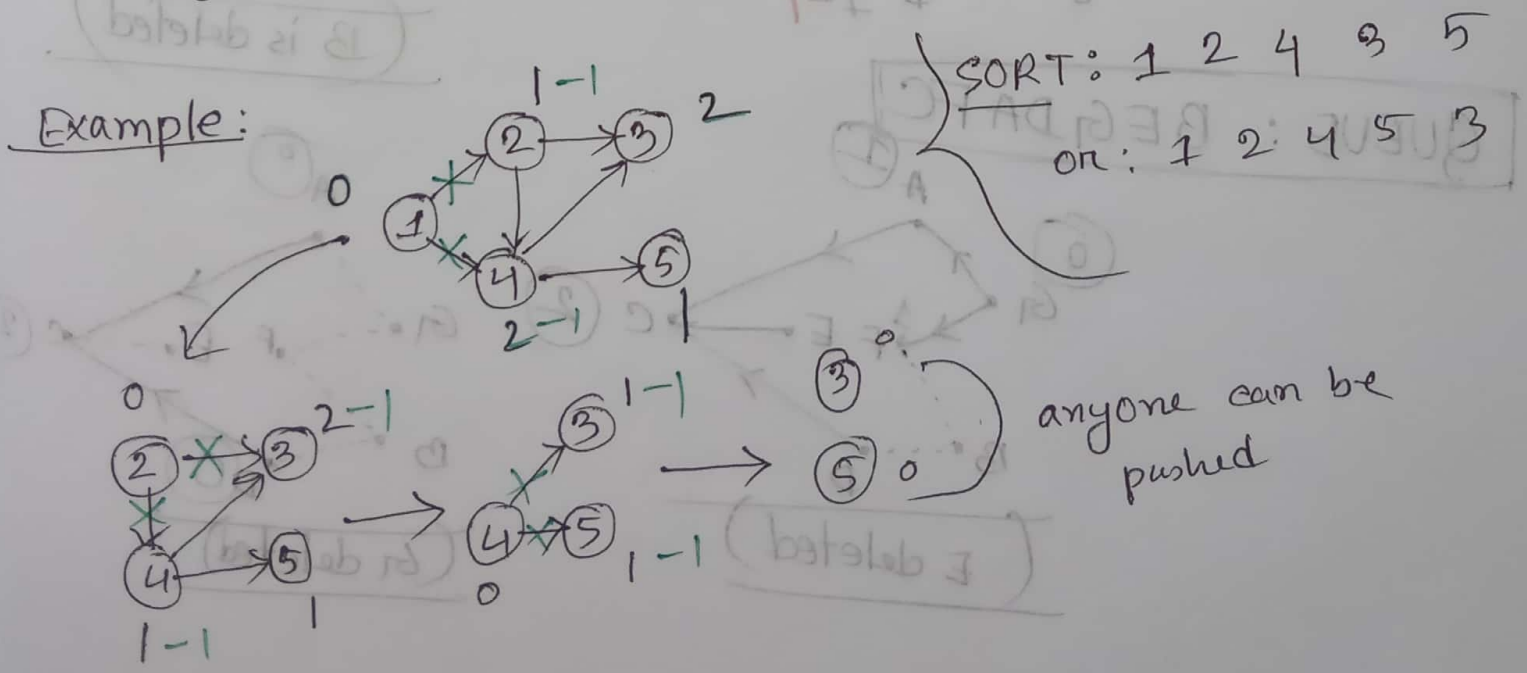
If A graph have these two property, then the graph have at least one topological sort.

Two step to find topological sort:

(i) Find indegree of every vertex. where indegree means incoming edge of a vertex.

(ii) chose which ~~w~~ vertex has indegree zero (0) and delete this.

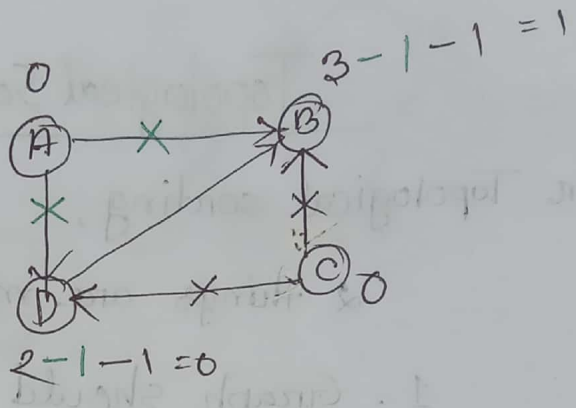
Example:



SORT: 1 2 4 3 5

or: 1 2 4 5 3

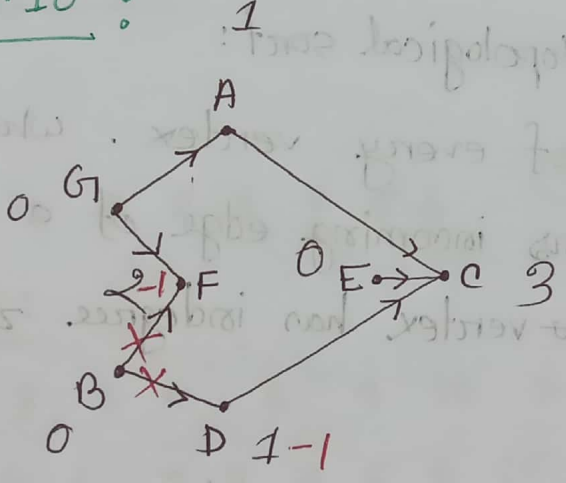
Another Example:

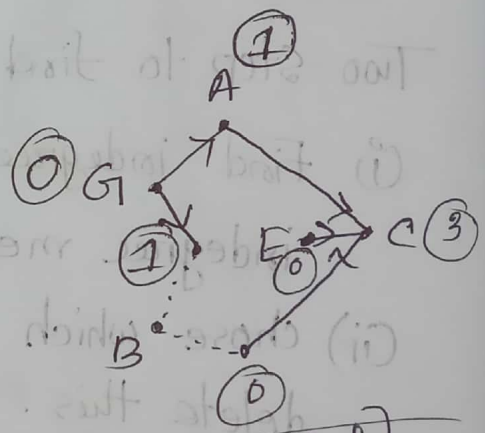


Case-1: A C D B

Case-2: C A D B

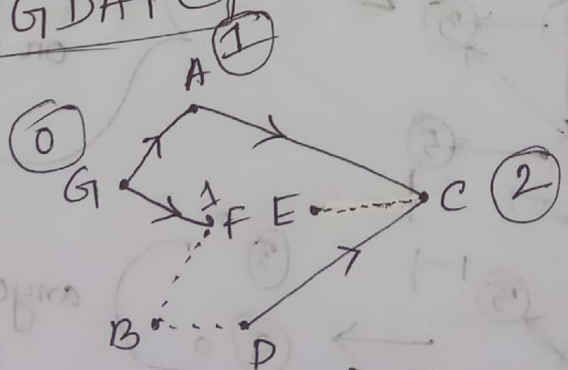
Example - 8.10 :



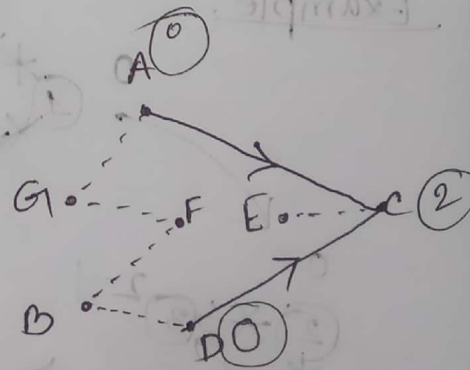


B is deleted

QUEUE : B E G D A F C



E deleted



G deleted