19/02/2024

Introduction to Data Structure = organized data + allowed operations

DS is systematic way to organize data so that it can be used efficiently.

DS gives us a way to structure a data to appropriately manage the data in a such way we get meaningful information from it

Data Structure-

Data structure is representation of logical relationship existing between individual elements of data, in other words a data structure is a way of organising all data items that considers not only the elements stored but also their relationship to each other selection of a particular data structure stresses on the following two things -

1. The data structure must be rich enough in structure to reflect the relationship between existing data.
2. The structure should be simple enough so that we can process data effectively whenever required.

Data structure mainly specifies the following four things -

- Organization of data
- Accessing method
- Degree of Associativity
- Processing alternatives for information/methods

Application of data structure -

Area in which data structure are applied extensively are -

- Array is used to store an image as a bitmap.
- DBMS (Database management system)
- Compiler Design
- Network Analysis
- AI
- Simulation
- OS
- Graphics
- Statistical / Analysis Package
- Stack ds used in implementing redo and undo feature.

Data Structure Operations -

1) Creation
2) Traversal
3) Searching
4) Insertion
5) Deletion

Classification of Data Structure -

(1) According to Nature of Size

(a) Static DS
(b) Dynamic DS

(a) **Static DS -**
If we can store data upto a fix number for example arrays.

(b) **Dynamic DS -**
Allows the programmer to change its size during program execution.
Ex- Linked list, trees and graphs.

(2) According to the Nature of its Occurence

(a) Linear DS

(b) Non-Linear DS

(a) Linear DS-

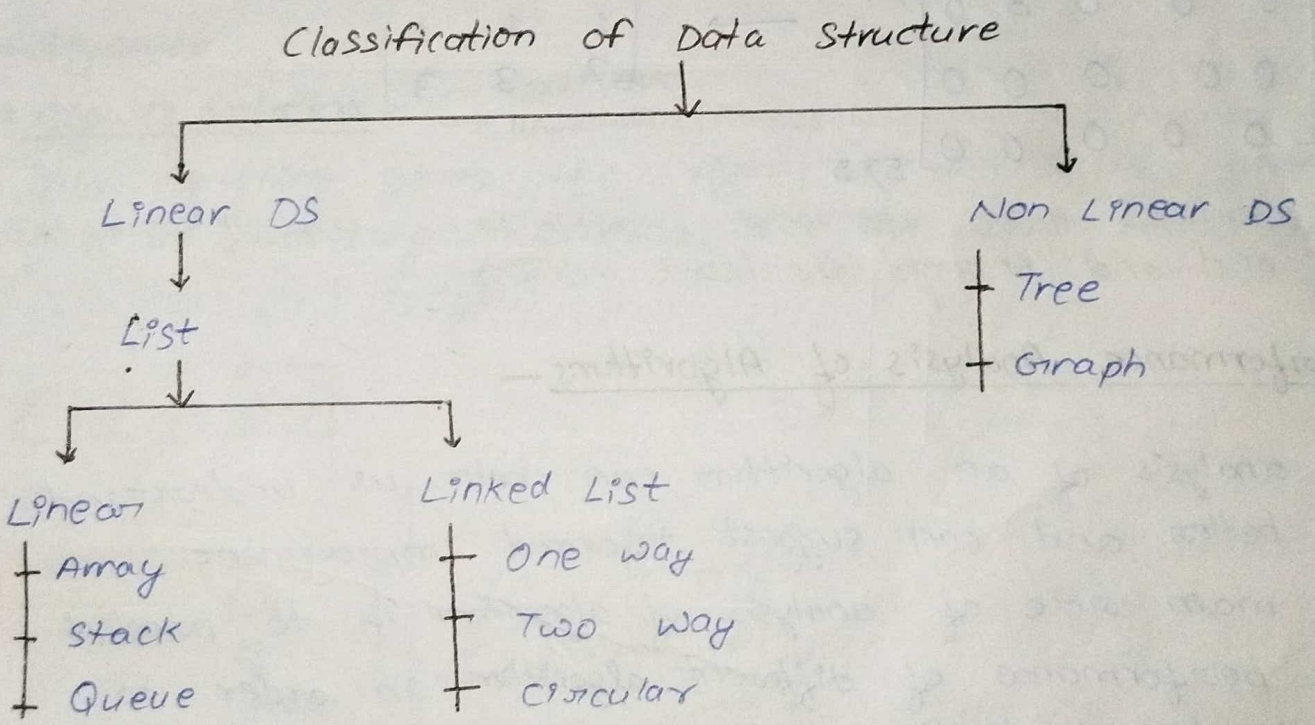
The arrangement of data is done linearly where each element consist of a unique predecessor and successor except for the first and last data element.

Ex- Array, Stack, queue and Linked list.
memory utilization is not efficient, single level involved, implementation is easy as compared to non-linear, data elements traversed in single run only.

(b) Non-Linear DS-

Arrangement of data is not in sequencial order. There exist a hierarchical relationship between the data elements. Linear manner of insertion and deletion ~~of~~ is not possible. There is no unique successor and predecessor.

Ex- Trees, graph etc.



Sparse Matrix –

Sparse matrix are the special type of matrices in which most of the elements are zeros (0s). Such matrices are not a good choice to implement as they require lot of memory and may also take unnecessary long computation time. These matrices can be represented using arrays by considering only the non-zero values and their position.

Example –

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 4 & 0 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}_{5\times5} \longrightarrow \begin{array}{c} \begin{array}{ccc} \text{Row} & \text{column} & \text{value} \end{array} \\ \begin{bmatrix} 0 & 0 & 4 \\ 0 & 2 & 2 \\ 1 & 1 & 1 \\ 2 & 3 & 3 \end{bmatrix}_{4\times3} \end{array}$$

Ideal DS takes the least possible time for all its operation and and consumes the least memory

Performance Analysis of Algorithms –

The analysis of an algorithm can help us understand it better and can suggest informed improvement. The main side of analysis of algorithm is to predict the performance of different algorithm in order to guide design decision.

Types of Analysis –

- **Worst case –**

Defines the input for which the algorithm takes longest time.

- **Best case-**

Defines the input for which algorithm takes the least time. It takes fastest time to complete.

- **Average case-**

Provide some prediction about the running time of the algorithm. The time may vary between best case and the ~~average~~ worst case.

Asymptotic Notation-

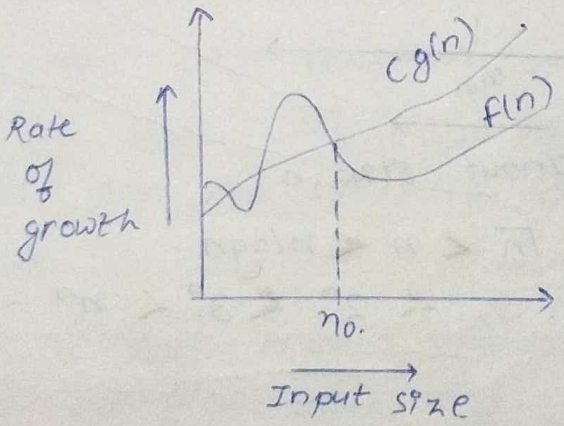
Having the expression for the best, average and worst cases, we need to identify upper and lower bounds. To represent these upper and lower bounds we need the following notations-

- ~~Big O Not~~

- **Big-O Notation-** → Worst case → Upperbound (Atmost)

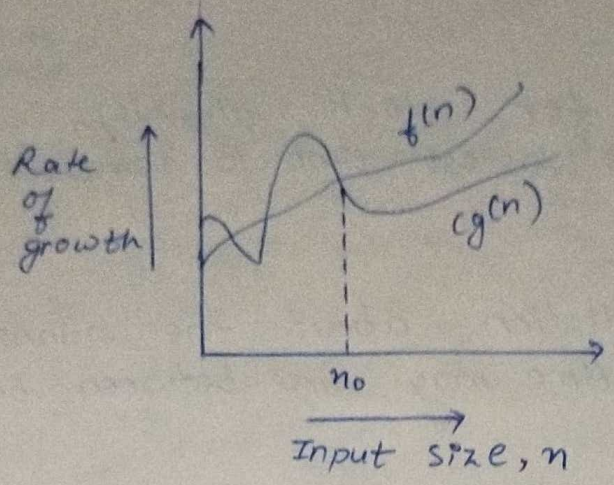
This notation gives the tight upperbound of given function, at larger value of n the upper bound of a function $f(n)$ is $cg(n)$.

$$0 \leq f(n) \leq cg(n)$$



- **Omega Ω Notation-** → best case → Lower Bound (At least)

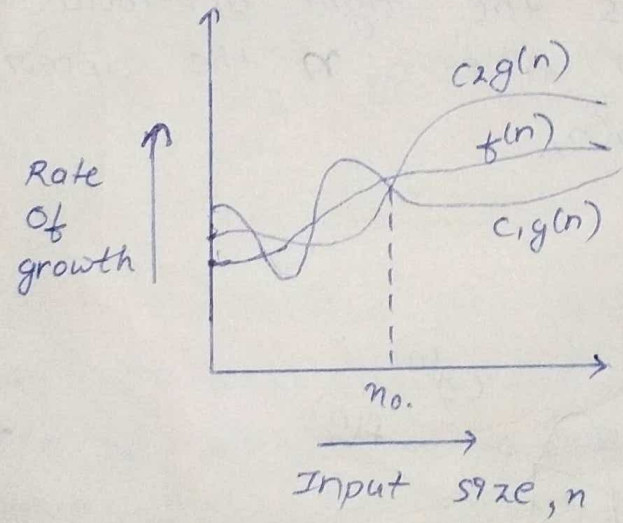
This notation gives the tighter lower bound this means that at larger values of n the tighter low bound of $f(n)$ is $cg(n)$. $0 \leq cg(n) \leq f(n)$



- **Theta Θ Notation** – → Average case → Exact time

This notation decide whether the upper bound and lower bound of a given algorithm or function are the same. The average running time of an algorithm is always between lower bound and the upper bound.

$$0 \leq c_1 g(n) \leq f(n)\ c_2 g(n)$$



$1 < \log n < \sqrt{n} < n < n\log n$
$< n^2 < n^3 \ldots < 2^n < 3^n \ldots < n^n$

Recursion-

It is a process when a function calls itself to solve a problem by breaking it into smaller problems, which are similar in nature to the original problem. All recursive function work in two phases - winding phase and unwinding phase. Winding phase begin's when the recursive function calls itself for the first time and each recursive call continues the winding phase. No return statement are executed in this phase, this phase terminates when the terminating condition become true. After this the unwinding phase begins and all the recursive function calls start returning in reverse order till the first instance of the function returns.

Unit-2 (Stack and Queue)

Stack-

A stack is a linear data structure in which all insertion and deletion are made at one end called the top of the stack.

A stack is an ordered collection of homogeneous data elements.

The insertion and deletion operations are given special names, the push operation insert the element in the stack and the pop operation deletes the element from the stack.

If there is not enough space, then the stack is said to be in overflow state and new elements can't be pushed. Similarly, before pop operation if stack is empty and pop operation is attempted, then stack is said to be in underflow stage state.

Algorithm for PUSH operation-

1. Start
2. a. If TOP $\geq$ MAX-1 then write "STACK OVERFLOW"
   b. STOP
3. Read DATA
4. Top ← Top+1
5. STACK[TOP] ← DATA
6. STOP

POP operation-

1. Start
2. a. If TOP<0, then write "STACK UNDERFLOW"
   b. STOP
3. STACK[TOP] ← Null
4. Top ← Top-1
5. STOP

Application of Stack-

- Reversal of string
- Checking validity of an expression containing nested parenthesis.
- Conversion of infix expression to postfix expression.
- Evaluation of postfix expression.
- Used in implementing undo and redo feature.
-

String Reversal

```
void main()
{
char str[MAX];
int i;
printf("Enter a string");
gets(str);
for(i=0; i<strlen(str); i++)
push(str[i]);
for (i=0; i<strlen(str); i++)
str[i] = pop();
printf("The reversed string is :");
puts(str);
}
```

Checking validity of an expression –

1. Initially take an empty stack.
2. Scan the symbols of an expression from left to right.
3. If the symbol is a left parenthesis push it on the stack.
4. If the symbol is right parenthesis –
   a. If the stack is empty print right parenthesis are more than left parenthesis, invalid expression.
      else
      pop an element from the stack
      If popped parenthesis does not match the parenthesis being scanned, print mismatch parenthesis, invalid expression.
5. After scanning all the symbols –
   If the stack is empty print valid expression.
   else
   print left parenthesis are more than right parenthesis, invalid expression.

Polish Notation –

Named after the Polish mathematician Jan Lukasiewicz refers to the notation in which the operator symbol is placed before its operands.

Reverse Polish Notation –

Refers to the notation in which operator symbol is placed after its operands.

In polish notation one never needs parenthesis to determine the order of operations in any arithmetic expression. The fundamental property of a polish notation is that the order in which the operations are to be performed is completely determined by the position of operators and operands in the expression.

Infix to postfix & Prefix Conversion –

Infix	(Polish Notation) Prefix	(Reverse Polish Notation) Postfix
a*b	*ab	ab*
a+b*c	+a*bc	abc*+
(a+b)*c	*+abc	ab+c*

Infix to postfix conversion –

(A+B)/(C-D)

AB+CD-/

A*B+C/D

AB*CD/+

A-B^C+D*E/(F+G)

⇒ ABC^-DEFG+/+

A+B/C-D*E+F

⇒ A+B/C-DE*+F
⇒ A+BC/-DE*+F
⇒ ABC/+-DE*F+
⇒ ABC/+DE*-F+

$A+(B*C-(D/E\hat{}F)*G)*H$ → Note Right → Left

$\Rightarrow A+(B*C-(DE\hat{}F/)*G)*H$

$\Rightarrow A+(B*C-(DEF\hat{}/)*G)*H$

$\Rightarrow A+(BC*-DEF\hat{}/*G)*H$

$\Rightarrow A+(BC*-DEF\hat{}/G*)*H$

$\Rightarrow A+(BC*DEF\hat{}/G*-)*H$

$\Rightarrow A+BC*DEF\hat{}/G*-H*$

$\Rightarrow ABC*DEF\hat{}/G*-H*+$

Alogrithm for conversion of Infix to Postfix using stack -

① Scan the expression from left to right. We shall get a symbol which may be an operand or a parenthesis or an operator. The symbol is treated as follows.

② (a) If the symbol is an operand it is added to the postfix ~~operat~~ expression.

(b) If the symbol is an opening parenthesis pust it onto the stack.

(c) If the symbol is an operator then check the top of the stack.

③ If the precedence of the operator at the top of the stack is higher or the same as the current operator then repeatedly it is popped and added to postfix expression otherwise it is pushed onto the stack.

④ If the symbol is the closing parenthesis then repeatedly pop from the stack and add each operator to the postfix expression until the corresponding opening parenthesis is encountered.

⑤ Remove the opening parenthesis from the stack.

- 5*(6+2)-(12/4)

Symbol Scanned	Stack	Postfix Expression
5		5
*	*	5
(	*(	5
6	*(	56
+	*(+	56
2	*(+	562
)	*	562+
-	-	562+*
(	-(	562+*
12	-(	562+*12
/	-(/	562+*12
)	-	562+*12/
		562+*12/-

- a+b*c+(d*e+f)*g

Symbol sconned	Stack	Postfix Expression
a		a
+	+	a
b	+	ab
*	+*	ab
c	+*	abc
+	+	abc*+
(	+(	abc*+
d	+(	abc*+d
*	+(*	abc*+d
e	+(*	abc*+de
+	+(+	abc*+de*
f	+(+	abc*+de*f
)	+	abc*+de*f+
*	+*	abc*+de*f+
g	+*	abc*+de*f+g*+

Evaluation of Postfix Expression –

1. Scan the symbol of array postfix one by one from left to right.
2. If an operand is encountered push it onto the stack.
3. If an operator is encountered
   - (a) Pop two elements from the stack where A is the top element and B is the next top element.
   - (b) Evaluate B ⊗ A, ⊗ → operator symbol
   - (c) Push the result in the stack.
4. After all the symbols of postfix have been scanned, pop the only element left in the stack and it is the required result of the evaluated postfix expression.

- 8+5*4 ⇒ 854*+

Symbol scanned	Action	stack
8	8 push	8
5	5 push	85
4	4 push	8 5 4
*	Pop 4&5, push 5*4	8 20
+	pop 20&8, push 8+20	28

- 7 5 3 2 ^ * 9 2 2 ^ - / + 6 4 * +

symbol scanned	Action	stack
7	7 push	7
5	5 push	7 5
3	3 push	7 5 3
2	2 push	7 5 3 2
^	pop 2&3, push 3^2	7 5 9
*	pop 9&5, push 5*9	7 45
9	~~push~~ 9 push	7 45 9
2	2 push	7 45 9 2
2	2 push	7 45 9 2 2
^	pop 2&2, push 2^2	7 45 9 4
-	pop 4&9, push 9-4	7 45 5
/	pop 5&45, push 45/5	7 9

+	pop 9 & 9, push 7+9	~~16~~ 16
6	6 push	~~2 16~~ 16 6
4	4 push	~~-2 16 4~~ 16 6 4
*	pop 4 & 6, push 6*4	~~-2 24~~ 16 24
+	pop 24 & -2, push -2+24	22 40

- 5 6 2 + * 12 4 / -

~~Stack~~ Symbol Scanned	Action	Stack
5	5 push	5
6	6 push	5 6
2	2 push	5 6 2
+	pop 2 & 6, push 6+2	5 8
*	pop 8 & 5, push 5*8	~~40~~
12	12 push	~~40~~ 12
4	4 push	~~40~~ 12 4
/	pop 4 & 12, push 12/4	~~40~~ 3
-	pop 3 & 40, push 40-3	37

- 2 3 1 * + 9 -

Symbol Scanned	Action	Stack
2	2 push	2
3	3 push	2 3
1	1 push	2 3 1
*	pop 1 & 3, push 3*1	2 3
+	pop 3 & 2, push 2+3	5
9	9 push	5 9
-	pop 9 & 5, push 5-9	-4

- 5 9 8 + 4 6 * + 7 - *

Symbol Scanned	Action	Stack
5	5 push	5
9	9 push	5 9
8	8 push	5 9 8
+	pop 8 & 9, push 9+8	5 17
4	4 push	5 17 4
6	6 push	5 17 4 6
*	pop 6 & 4, push 4*6	5 17 24
+	pop 24 & 17, push 17+24	5 41
7	7 push	5 41 7
-	pop 7 & 41, push 41-7	5 34
*	pop 34 & 5, push 34*5	170

- 5 9 3 / 2 1 + * + 6 2 / - 3 +

symbol Scanned	Action	Stack
5	5 push	5
9	9 push	59
3	3 push	593
/	pop 3 & 9, push 9/3	53
2	2 push	532
1	1 push	5321
+	pop 1 & 2, push 2+1	533
*	pop 3 & 3, push 3*3	59
+	pop 9 & 5, push 5+9	14
6	6 push	14 6
2	2 push	14 6 2
/	pop 2 & 6, push 6/2	14 3
-	pop 3 & 14, push 14-3	11
3	3 push	11 3
+	pop 3 & 11, push 11+3	14

- 4 3 2 ^ + 1 8 * 2 2 + / - 2 -

Symbol scanned	Action	Stack
4	4 push	4
3	3 push	4 3
2	2 push	4 3 2
^	pop 2 & 2, push 3^2	4 9
+	pop 9 & 4, push 4+9	13
1	1 push	13 1
8	8 push	13 1 8
*	pop 8 & 1, push 1*8	13 8
2	2 push	13 8 2
2	2 push	13 8 2 2
+	pop 2 & 2, push 2+2	13 8 4
/	pop 4 & 8, push 8/4	13 2
-	pop 2 & 13, push 13-2	11
2	2 push	11 2
-	pop 2 & 11, push 11-2	9

- 2 3 ^ 1 - 4 2 / 6 * + 3 1 + 2 / -

Symbol scanned	Action	stack
2	2 push	2
3	3 push	2 3
^	pop 3 & 2, push 2^3	8
1	1 push	8 1
-	pop 1 & 8, push 8-1	7
4	4 push	7 4
2	2 push	7 4 2
/	pop 2 & 4, push 4/2	7 2
6	6 push	7 2 6
*	pop 6 & 2, push 2*6	7 12
+	pop 12 & 7, push 7+12	19
3	3 push	19 3
1	1 push	19 3 1
+	pop 1 & 3, push 1+3	19 4
2	2 push	19 4 2
/	pop 2 & 4, push 4/2	19 2
-	pop 2 & 19, push 19-2	17

QUEUE

Queue is a linear list having homogeneous collection of elements which has two ends, one for insertion of elements and other for deletion for elements. The end where insertion takes place is called rear end and the end from where the elements are deleted is called frontend. The insertion operation is known as enqueue while deletion operation is called dequeue.

The first element inserted in the queue will be the first one that will be retrieved, hence it is called FIFO (First in First out) list.

Basic operations in Queue ⇒

- **Create Queue()** -
  To create an empty queue.
- **Enqueue()** -
  Add or store an item to the queue.
- **Dequeue()** -
  Access or remove an item from the queue.
- **Peek()** -
  Gets the element at the front of the queue without removing it.
- **IsFull()** -
  checks if the queue is full or not.
- **IsEmpty()** -
  checks if the queue is empty or not.

Array implementation of Queue-

- Initially when the queue is empty the values of both front and rear variables will be -1.
- For insertion value of rear variable is incremented by 1 and the element is inserted at the new rear position.
- For deletion the element at front position is deleted and the value of front variable is incremented by 1.
- When insertion is done in an initially empty queue that is when the value of front is -1 it is ~~made 0~~ updated 0.
- At any point of time number of elements in the queue is equal to rear - front + 1 except initial empty queue.
- When front equals to rear there is only one element present in the queue, except initial empty queue.
- When front becomes equals to rear + 1 the queue becomes empty therefore there are two conditions for underflow.
- Condition for overflow rear = MAX - 1

Algorithm for Queue Insertion -

1) Start
2) If REAR $\geq$ MAX-1 (or REAR = MAX-1) then write "QUEUE OVERFLOW" & stop
3) Read DATA
4) If FRONT = -1 then
   - 4.1) FRONT ← FRONT+1
   - 4.2) REAR ← REAR+1
   - 4.3) Queue[REAR] ← DATA

   Else
   - 4.4) REAR ← REAR+1
   - 4.5) Queue[REAR] ← DATA
5) Stop

Algorithm for Deletion from Queue-

1. Start
2. If FRONT = -1 OR FRONT = (REAR+1) then write "QUEUE UNDERFLOW" & Stop
3. If FRONT = REAR then
   - 3.1 Queue[FRONT] ← NULL
   - 3.2 FRONT ← -1
   - 3.3 REAR ← -1

   Else
   - 3.4 Queue[FRONT] ← NULL
   - 3.5 FRONT ← FRONT+1
4. Stop

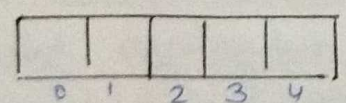
Types of Queue -

- Circular Queue
- Double Ended Queue
- Priority Queue

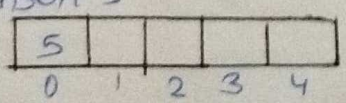
Circular Queue - (Ring buffer)

Circular queue is one in which insertion of new element is done at the very first location of the queue, if last location of the queue is full.

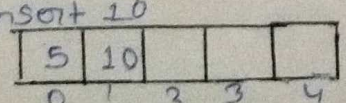
If the value of rear is MAX-1 then instead of incrementing rear we will make it zero and then perform insertion. Similarly when the value of front becomes MAX-1 it will be reset to 0.

ⓐ 

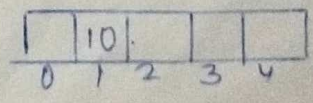
f = -1, r = -1

ⓑ Insert 5


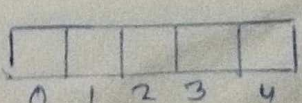
f = 0, r = 0

ⓒ Insert 10


f = 0, r = 1

ⓓ Delete


f = 1, r = 1

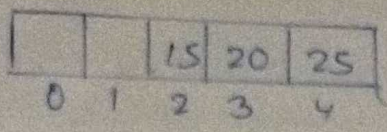
ⓔ Delete


f = 2, r = 1

At insertion value of r changes

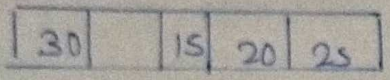
At deletion value of f changes

(f) Insert 15, 20, 25



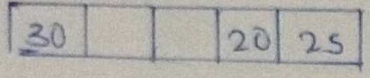
f = 2, r = 4

(g) Insert 30 (circular queue concept)



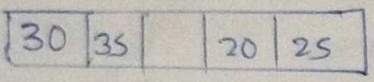
f = 2, r = 0

(h) Delete



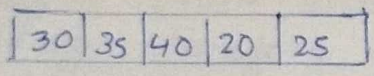
f = 3, r = 0

(i) Insert 35



f = 3, r = 1

(j) Insert 40



f = 3, r = 2

Algorithm –

- Inserting an Element to Circular Queue –

1. Start
2. If (FRONT == 0 && REAR = MAX-1) || (FRONT = REAR+1) then write "QUEUE OVERFLOW" & stop
3. Read DATA to insert
4. If (FRONT == -1)
   - 4.1. FRONT = 0
   - 4.2. REAR = 0
5. If (REAR = MAX-1) then REAR = 0 else REAR = REAR+1
6. QUEUE [REAR] = DATA
7. Stop

- Deleting an element from Circular Queue-

1. Start
2. If (FRONT == -1), then write QUEUE UNDERFLOW" & stop
3. If (FRONT == REAR) then
   - 3.1 C QUEUE[FRONT] = NULL
   - 3.2 FRONT = -1
   - 3.3 REAR = -1
4. If (FRONT == MAX-1)
   - 4.1 CQUEUE[FRONT] = NULL
   - 4.2 FRONT = 0

   else
   - 4.3 CQUEUE[FRONT] = NULL
   - 4.4 FRONT = FRONT + 1
5. Stop

Double ended queue- (Dequeue / Deque) - (Head-tail linked list)

In a double ended queue both insertion and deletion operations are performed at either end of the queue which means we can insert an element as well as delete an element from either rear end or front end. Double ended queue can be used as a stack and as a queue. Possible operations that can be performed on double ended queue are-

1. Add an element at the rear end.
2. Add an element at the front end.
3. Delete an element from the front end.
4. Delete an element from the rear end.

Types of double ended queue-

- Input Restricted Deque
- Output Restricted Deque

- **Input Restricted deque-**
  In this deque elements can be added at only one end but we can delete the elements from both ends.
  ~~In an output ries~~
- **Output Restricted deque-**
  In this deque deletion take place at only one place but insertion is allowed at both end.

Priority Queue-

A priority queue is a data structure in which each element has been assigned a value called priority of the element and element can be inserted or deleted not only at the ends but at any position.

- An element of higher priority is processed before element of lower priority.
- Two elements with same priority are processed to the order in which they were inserted in the queue.

There are two ways of implementing a priority queue-

① **Through Queue-** In this case insertion is simple because element is simply added at the rear end as usual. For performing deletion, first the element with highest priority searched and then it is ~~added~~ deleted.

② **Through sorted list-** In this case insertion is costly because the element is inserted at the proper place in the list based on its priority. Here deletion is easy because element with highest priority will always be in the beginning of the list.

If the priority queue is implemented using arrays then in both the above cases shifting of elements will be required so it is advantageous to use a linked list because insertion or deletion in between the linked list is more efficient.

Types of Priority Queue-

- **Ascending Priority Queue** - In this queue elements can be inserted in any order but while deleting the element always the smallest priority element is deleted first. ~~similarly~~
- **Descending Priority Queue** - Elements can be inserted in any order but always the largest priority element is deleted first.

Application of Queue-

- There are several algorithms that use queues to solve problem easily for example traversing a binary tree, breadth first search (BFS) etc.
- Round Robin technique for processor scheduling is implemented using queue.
- When the jobs are submitted to a network printer they are arranged in a queue and accessed accordingly.
- Every real life line is a queue.
- Used to manage incoming calls and ensure that they are handled in correct order.
- Used in operating system for semaphore, CPU scheduling, memory management, FCFS algorithm in which
- Used to manage ~~print~~ the order ~~not~~ print jobs are sent to the printer.
- Applied to add song at the end of the playlist.
- ATM booth line, Ticket counter line.
- Used in routers and switches

Unit-3

Linked-List

Limitations of Array-

- Memory storage space is wasted as the memory remains allocated to the array throughout the program execution.
- The size of the array can't be changed after it's declaration i.e. the size has to be known at the compiler time.
- The data in the array are separated in the computer memory by same distance which means that inserting an item inside the array requires shifting other data in this array.

These limitations can be overcome by using linked list data structure.

Linked-List-

It is an ordered collection of data elements called 'Nodes', where the linear order is maintained by means of links or pointer. Each node is divided into two part, the 1st part contains the information of the element or its value and the 2nd part contains address of the next node.

Node → A self referencial structure.

Key terms-

Node/Data field - Value of the element

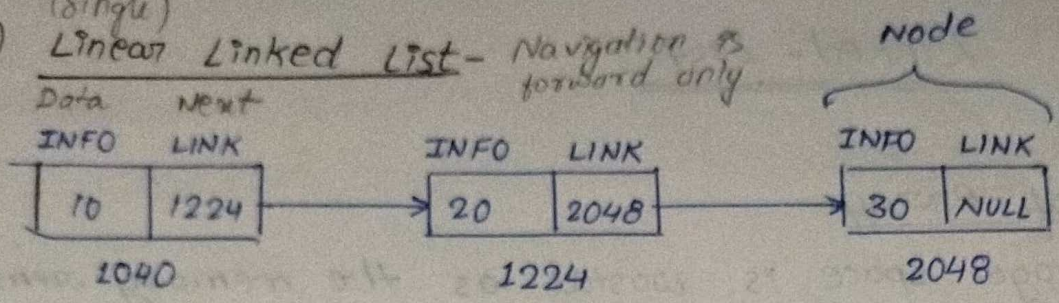
Link field - Address of the next node.
or
Next pointer Field

The no. of pointers is maintained depending on the requirement's usage. Based on this link list are classified into three categories -

- Linear LL
- Doubly LL
- Circular LL

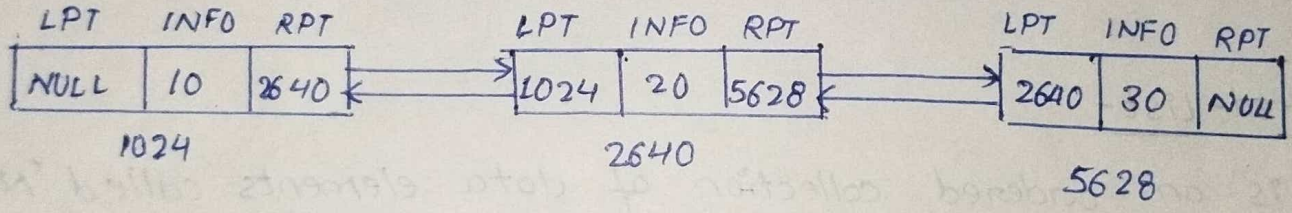
Types of Linked List-

① (Single) Linear Linked List- Navigation is forward only.



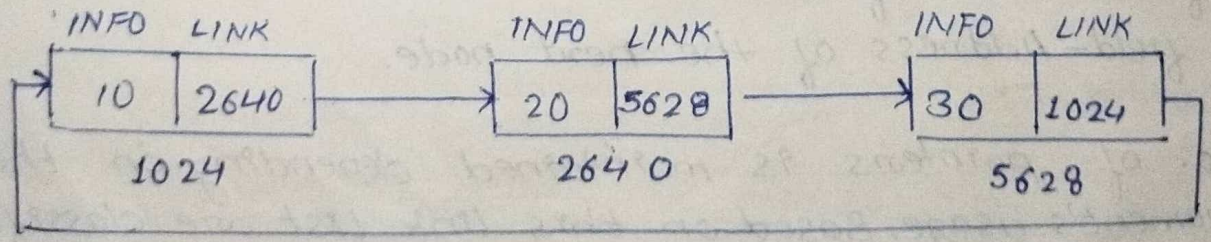
In linear linked list each node is divided into two parts. The first part contains information of the element and the second part contains address of the next node in the list. The last node's link field has a NULL pointer representing that there are no more nodes in the link list.

② Doubly Linked List- Forward and backward navigation is possible.



Doubly Linked list are also called two way list. Here two linked fields are maintain which help in accessing both successor and predecessor nodes.

③ Circular Linked List- Last element is linked to the first element.



It is just like a linear linked list in which second part of the last node contains the address of the first node in the list i.e. second field of the last node does not point to the NULL pointer rather it points back to the beginning of the list.

Advantages of Linked List -

- Linked list are dynamic data structures, they can grow or shrink during the execution of a program.
- The size is not fixed.
- Data can be stored in non-continuous memory loss.
- Insertion and deletion of nodes are easier and efficient.

Disadvantage of Linked List-

- Linked list need more memory because there is a special field called Link field which holds address of the next node.

Insertion in the Linked List-

- At the Beginning-

1. Start
2. Set TMP = (Node *) malloc (size of (node))
3. If TMP == NULL then write "LIST OVERFLOW" & stop.
4. Set TMP → INFO = DATA
5. Set TMP → LINK = START
6. Set START = TMP
7. Stop

Note- The order of step 5 and 6 is important, first we should make Link of TMP = START and after that only we should update start. If we reverse this order then the link of TMP will point to itself and we will stuck in infinite loop when it is processed.

- Insertion in an empty list -

When the list is empty value of start will be NULL. The new node that we are adding will be the only node in the list. Since it will be the first node, start should point to this node and the link of this node should be NULL.

- At the End -

1) Start
2. Set TMP = (node*) malloc (size of (node))
3. If TMP == NULL then write "LIST OVERFLOW" and stop.
4. Set TMP ⇒ INFO = DATA
5. PTR = START
6. while (PTR → LINK != NULL)
   6.1. PTR = PTR → LINK
7. PTR → LINK = TMP
8. TMP → LINK = NULL
9. STOP

- Insertion at a given position -

1. Begin
2. Set TMP = (node*) malloc (size of (node))
3. If TMP == NULL then print "LIST OVERFLOW" & stop.
4. Set TMP → INFO = DATA
5. Set counter COUNT = 1
6. If (POS == 1) then
   6.1 TMP → LINK = START
   6.2 START ⇒ TMP
7. while (COUNT = 1 && PTR → LINK != NULL)
   Set PTR = PTR → LINK
8. If PTR == NULL
   print "Less Nodes"
   Else
   TMP → LINK = PTR → LINK
   PTR → LINK = TMP
9. END

Deletion of Nodes from Linked List-

• Deletion of Nodes from Beginning -

1) Begin
2. If START == NULL then print "List is Empty" & stop.
3. Set PTR = START
4. Set START = START → LINK
5. Print "Deleted Node is :" PTR → INFO
6. Free (PTR)
7. End

• Deletion of Nodes from End-

1. ~~Deletio~~ Begin
2. If START == NULL then print "List is Empty" & stop.
3. Set PTR = START
4. while (PTR → LINK != NULL)
   4.1) TMP = PTR
   4.2) PTR = PTR → LINK
5. TMP → LINK = NULL
6. Print "Deleted Node is ;" PTR → INFO
7. Free(PTR)
8. End

• Deletion of Nodes from specific position-

1. Begin
2. If START == NULL then print "List is Empty" & stop.
3. Set count = 0
4. Set PTR = START
5. while (count < POS && PTR → LINK != NULL)

5.1) TMP = PTR

5.2) PTR = PTR → LINK

If PTR == NULL then print
"Position Not Found" & Stop

Else
TMP → LINK = PTR → LINK

6. Print "Deleted Node is:", PTR → INFO

7. Free (PTR)

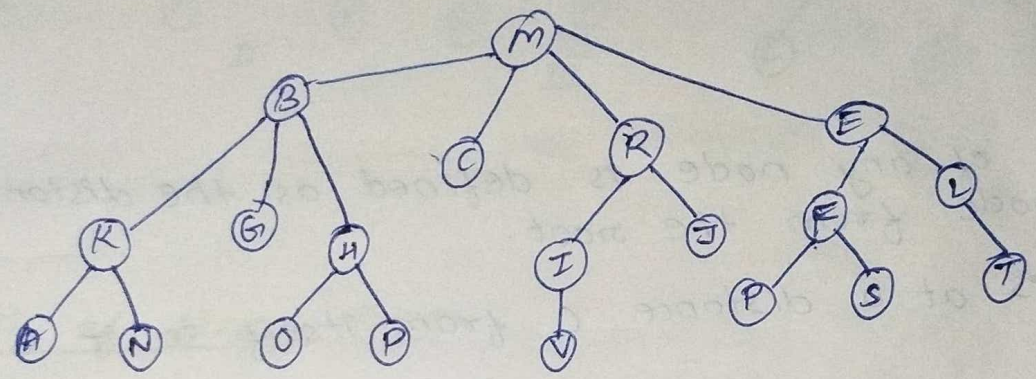
8. End

Unit-4

Tree

Tree-

It is a non linear data structure which represents hierarchical relationship among its elements. It is very useful for information retrieval and searching in it is very fast.



Tree Terminology-

Node- Each element of a tree is called a node Each node is represented by a circle.

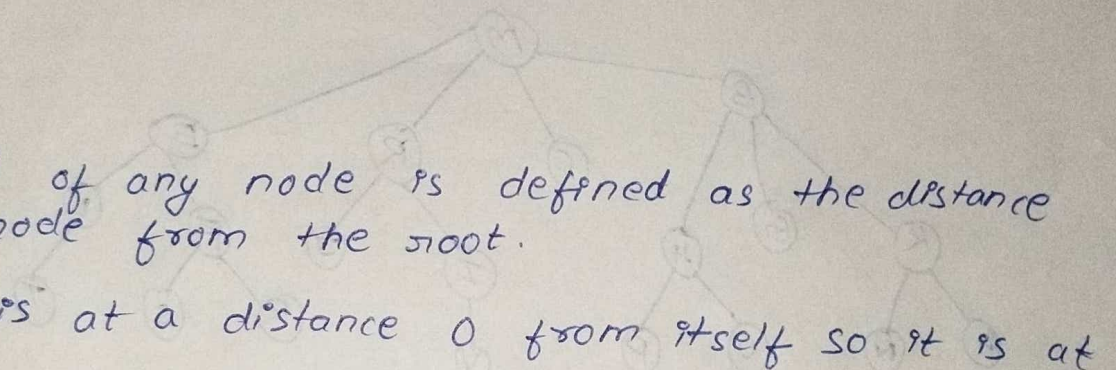
Edges- The lines connecting the nodes are called edges or branches.

Parent Node- The immediate predecessor of a node is called parent node

Child Node- All the immediate successor of a node are called its child node.

Root Node- This is a ~~specially~~ highest node in the tree structure, and has no parent.

Leaf Node- A node that doesn't have any child is called leaf node or terminal node.



Level- Level of any node is defined as the distance of that node from the root.

Root node is at a distance 0 from itself so it is at level zero.

Height - The total no. of level in a tree is the height of the tree. So height is equal to one more than the largest level number of the tree. It is also sometimes known as depth of the tree.

Degree- The no. of sub trees or children of a node is called its degree.

Recursive definition of the tree-

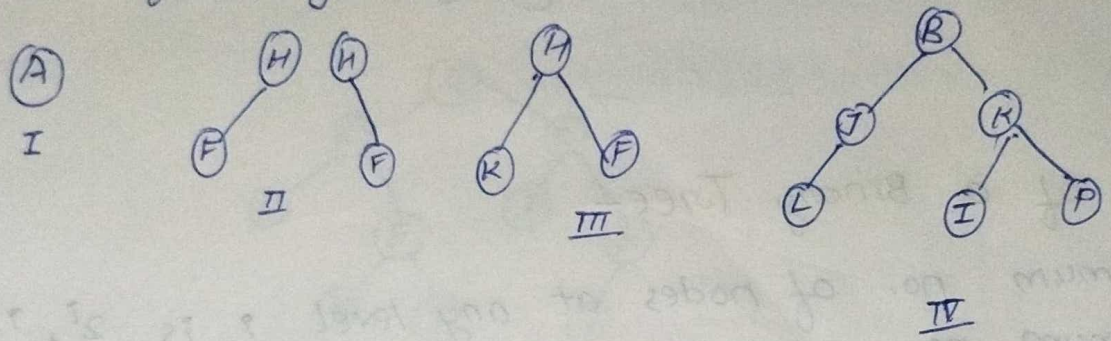
A tree is a finite set of nodes such that -

- There is a distinguish node called root node
- The remaining nodes are partioned into $n \geq 0$ disjoint sets. $T_1, T_2 \ldots T_n$, where each of these sets is a tree.
- The sets T_1, T_2 upto T_n are the sub trees of the root.

Binary Tree- In a binary tree no node can have more than two children that is a node can have 0, 1 or 2 children.

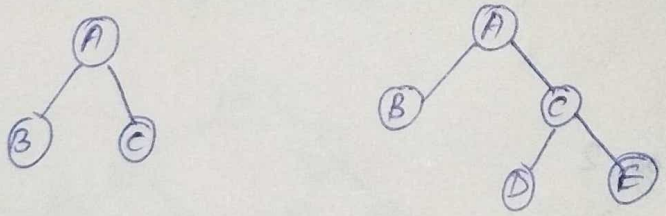
Recursive definition of Binary Tree- A binary tree is a finite set of nodes such that it is either empty or consist of a distinguish node called root and remaining nodes are partioned into two disjoint sets T_1 and T_2 and both of them are binary trees.

Examples of binary tree-

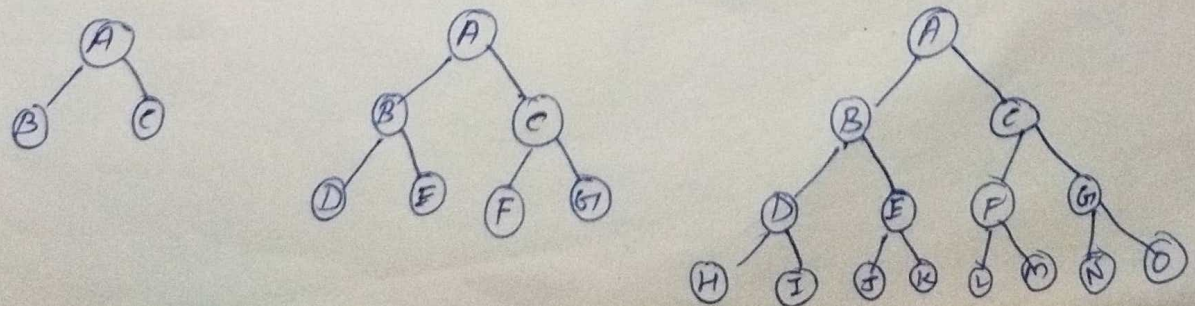


Types of Binary Tree-

- Strictly Binary Tree-
A binary tree is a strictly binary tree if each node in the tree is either a leaf (0 child) node or exactly has two children i.e. there is no node with one child.

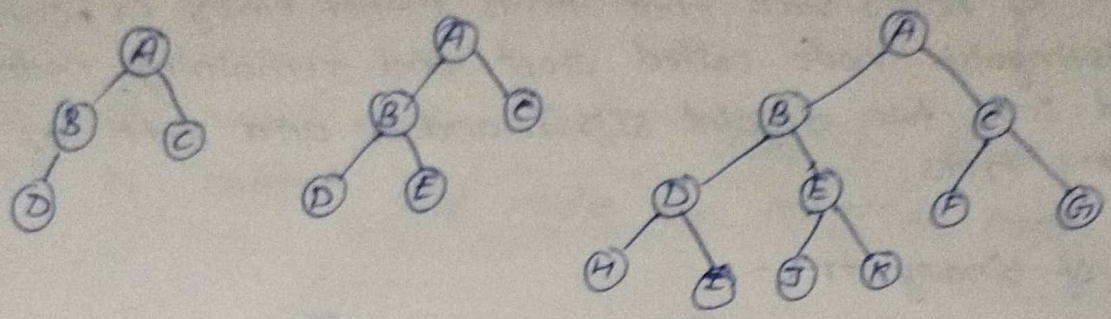


- Full binary tree-
A binary tree is a full binary tree if all the levels have maximum no. of nodes.



- **Complete Binary Tree-**

A complete binary tree is a binary tree where all the levels have maximum no. of nodes except possibly the last level.



Properties of a Binary Tree-

- The maximum no. of nodes at any level i is 2^i, $i \geq 0$
- The maximum no. of nodes possible in a binary tree of height h is $2^h - 1$
- The minimum number of nodes possible in a binary tree of height h is equal to h.
- If a binary tree contains n nodes then its maximum height possible is n and minimum height possible is $\lceil \log_2(n+1) \rceil$

Proof -

$$n \leq 2^h - 1$$

$$n + 1 \leq 2^h$$

$$\log_2(n+1) \leq h \log_2 2$$

$$h \geq \log_2(n+1)$$

- In a binary tree if n is the total no. of nodes and e is the total no. of edges then $e = n - 1$

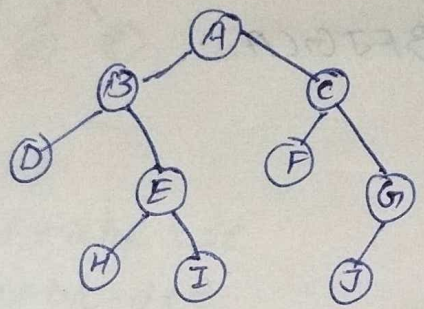
Tree traversal-

- Pre order (NLR)
- Inorder (LNR)
- Post order (LRN)

• Pre order (NLR)-

1. Visit the root node(N)
2. Traverse the left subtree(L)
3. Traverse the right subtree(R)

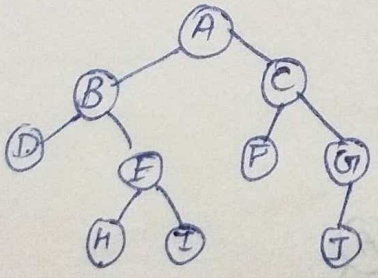
Ex-



Pre : ABDEHICFGJ

• Inorder (LNR)-

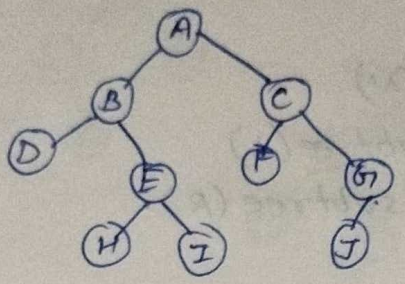
1. Traverse the left subtree(L)
2. Visit the root node(N)
3. Traverse the right subtree(R)



Inorder : DBHEIAFCJG

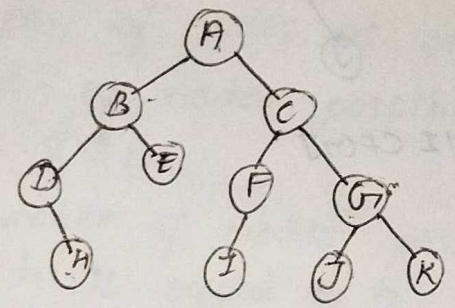
• Post order (LRN) -

1. Traverse the left subtree (L)
2. Traverse the right subtree (R)
3. Visit the root node (N)



Post order : DHIEBFJGCA

Q-



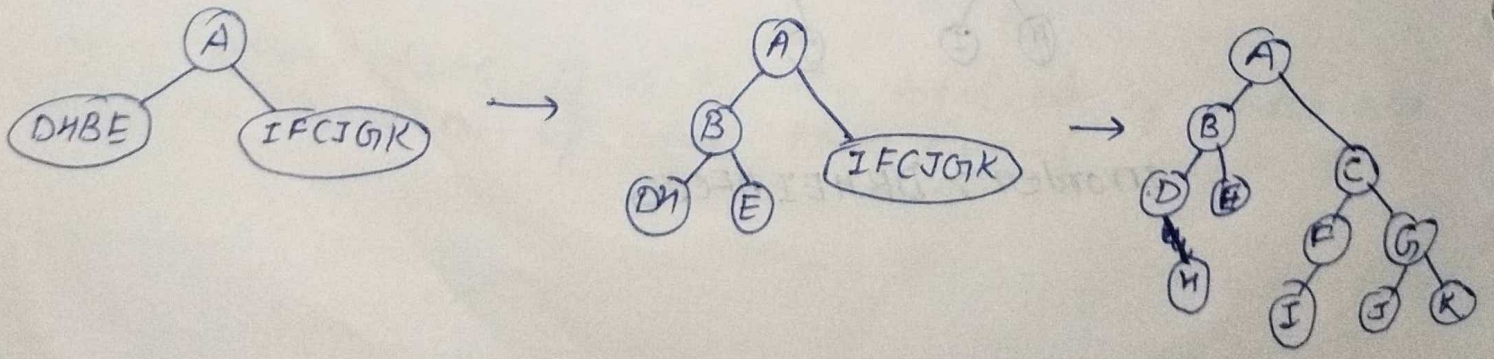
Pre : ABDHECFIGJK

In order : DHBEAIFCJGK

Post order : DHDEBIFJKGCA

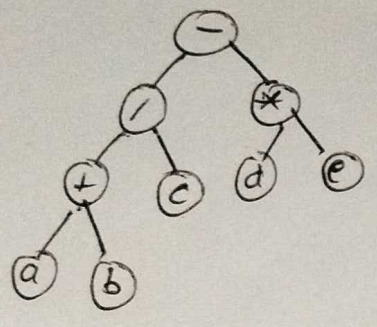
Q- Pre : ABDHECFIGJK

In order : DHBEAIFCJGK



```
Struct tree
{
int data;
Struct tree *lchild;
Struct tree *rchild;
};
```

Q-



Pre order : -/+abc*de

In order : a+b/c-d*e

Past order : ab+c/de*-