PAGE:
DATE: / /

Assignment No. 4

Name: Bhavana Batna
Roll no.: 23107
Class: SE 09
Batch: E 09

1] Title: Expression tree creation and traversal.

2] Aim: To implement a expression tree using stack data structure.

3] Problem Statement: Construct an expression tree for postfix expression and perform recursive & non-recursive, inorder, preorder and postorder traversal.

4] Theory:

- Concept of non-linear data structure with example:
  - Data structures where data elements are not arranged linearly or sequentially are called non-linear data structures.
  - In non-linear data structure, single level is not evolved. Therefore, we can't traverse all the elements in single run only.
  - Non-linear data structure, are not easy to implement in comparision to linear data structure.
  - But it utilizes computer memory efficiently in comparision to linear data structure.
  - Examples of non-linear data structures:
    Eg: Trees, graphs.

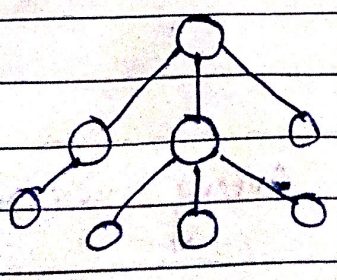


Fig: Tree

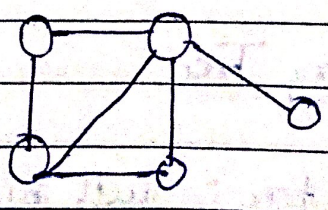


Fig: Graph

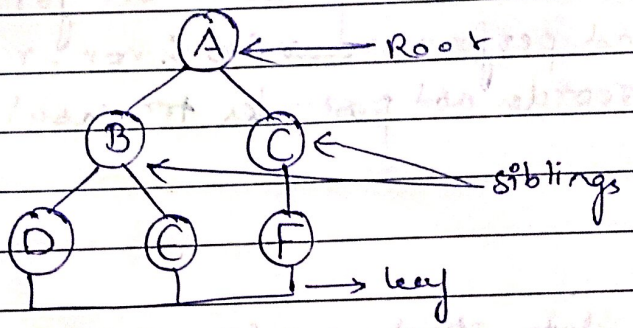
Rainbow
PAGE:
DATE: / /

- **Binary Tree :-**

Tree in which any node can have atmost two branches, i.e., at most two children, is a binary tree.

Definition:
A binary tree is a finite set of nodes that is either empty or consist of a root and two disjoint binary trees called 'left subtree' and 'right subtree'.

Fig:



Binary Tree

Terminologies:

1) Root : Node without parent
2) Sibling : Nodes share same parents.
3) Internal nodes : Nodes with atleast one child.
4) External nodes : Nodes without children.
5) Ancestors of node : Parent, grandparent, grandparents.
6) Descendant of node : child, grand child, grand-grand children.
7) Depth of node : Number of edges from root node.
8) Height of tree : Maximum depth of any node.

- **Full Binary Tree :**

A binary tree is full binary tree if every node has zero or two children.

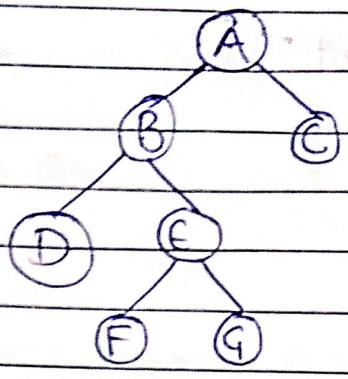


Fig: Full Binary Tree

- **Complete Binary Tree:**

A complete binary tree is a binary tree which is completely filled, with the possible exception of bottom level which is filled from left to right.

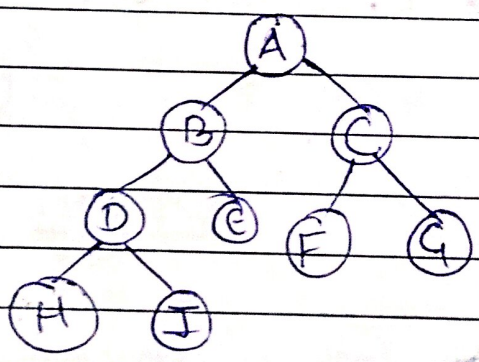


Fig: Complete Binary tree.

- **Binary Tree ADT:**

Structure Binary Tree is a finite set of nodes either empty or consisting of root node, left binary tree & right binary tree.

Operations:

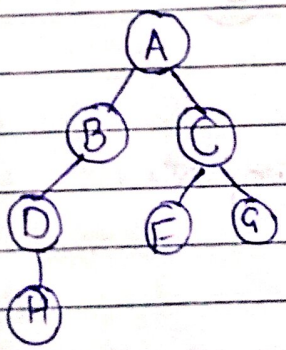
Bintree create()
boolean isEmpty()

Rainbow
PAGE:
DATE: / /

Bintree MakeBT(bt1, item, bt2)
Bintree lchild(bt)
element data(bt)
element data(bt)
Bintree rchild(bt)
Void inorder(bt)
Void preorder(bt)
void postorder(bt)

- Realization of ADT with binary:

- If a binary tree with 'n' nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have:
parent is at $\lfloor i/2 \rfloor$
left child is at $2i$
right child is at $2i+1$



Array representation:

1	2	3	4	5	6	7	8	9
A	B	C	D		F	G		H

- Realization of ADT with Linked list:

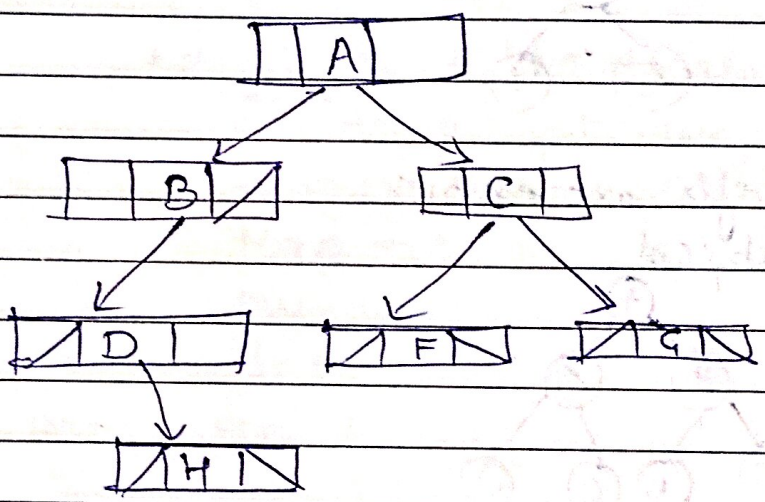
-Binary Tree is linked representation are stored in memory as linked lists. These lists are linked to each other through parent-child relationship associated with trees.

- Each node has three parts:

1) Data element
2) Pointer that points towards left node.
3) Pointer that points towards right node.

left child	data	right child

Linked list representation:



- Binary Tree Applications:

1) A binary tree is useful data structure when two way decisions must be made at each point in a process. Examples: Finding duplicates in a list of numbers.

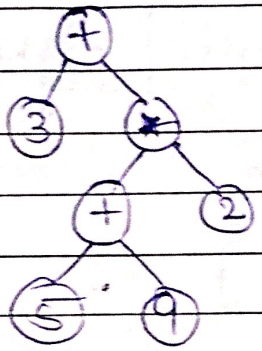
2) A binary tree can be used for representing an expression containing operands (leaf) and operators (internal nodes).

- Expression Tree concepts:

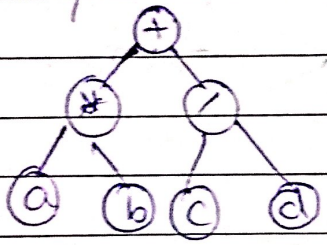
- An expression tree is a representation of expression arranged in a tree like data structure.

- It is a binary tree in which internal nodes corresponds to the operator & each leaf node corresponds to the operators.

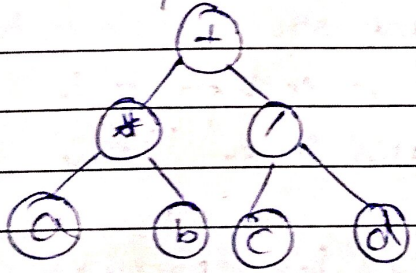
For eg: Infix : 3 + ((5+9)*2)



Example of prefix expression:
+ * ab / cd



Example of postfix expression:
ab * cd / +



- Applications of expression Tree :

1) Evaluation of airthmetic expression.
2) Expression conversion i.e. infix to postfix or prefix

- Algorithm/pseudocode:

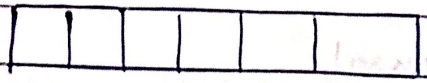
i) Expression Tree creation from postfix expression:

```
ET * create_ET(postfix[])
    for i=0 to postfix.length
        if postfix[i] = operand
            ET *temp = getNode(postfix[i])
            push(temp)
        else if postfix[i] = operator
            ET *temp = getNode(postfix[i])
            temp -> right = pop()
            temp -> left = pop()
            push(temp)
    end of for.
    return pop().
```

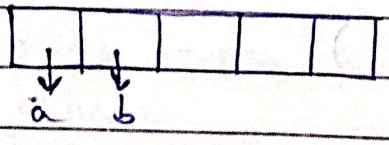
Example:

Postfix = ab+cd+x

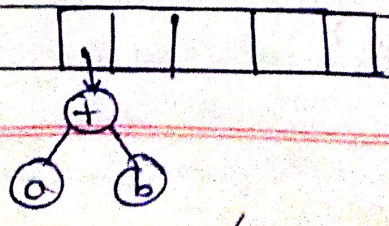
stack :-



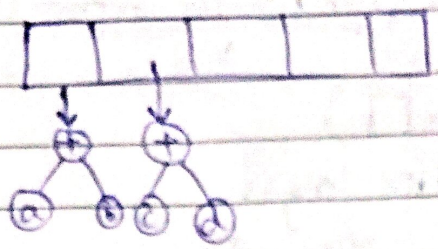
push(a)
push(b)



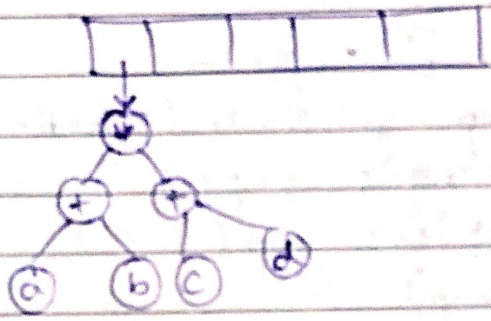
Next symbol is '+'. It pops two pointers from stack, a new tree is formed pointer is pushed onto stack.



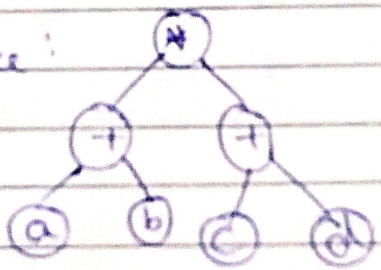
push (c)
push (d)



Similarly, for *



∴ Created Tree :



2) Recursive Traversal :

i) Recursive inorder –

```
Procedure Inorder (T)
if T = NULL
    return
Inorder (T → left)
print (T → data)
inorder (T → right)
```

ii) Recursive preorder:

```
Procedure preorder (T)
if T = NULL
    return
print (T → data)
preorder (T → left)
preorder (T → right)
```

iii) Recursive postorder:

```
Procedure Postorder (T)
if T = NULL
    return
Postorder (T → left)
Postorder (T → right)
print (T → data)
```

3) Non-recursive Traversal:

i) Non-recursive inorder:

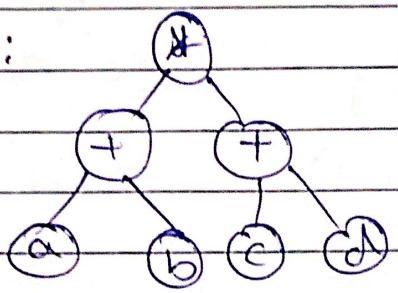
```
Procedure inorder (T)
// S & top denotes stack & associative top
if T = NULL
    print "Empty Tree"
    return
top = 0
while T ≠ NULL or top ≠ -1
    while T ≠ NULL
    push (S, top, T)
    T = T → left
```

Rainbow

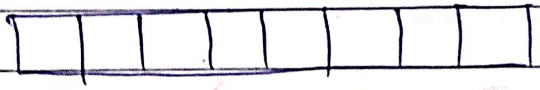
PAGE:
DATE: / /

if top ≠ -1
T = top(s)
print(T → data)
T = T → right

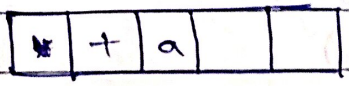
Example:



Stack →



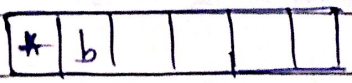
push (*)
push (+)
push (a)



a → left ≠ NULL ⇒ false
pop()

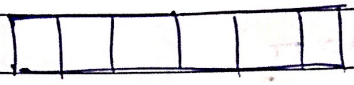
print a+

push (b)

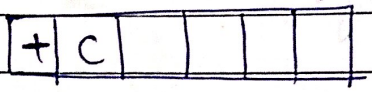


b → left ≠ NULL ⇒ false
pop()
print ⇒ a + b
b → right ≠ NULL
pop()

print ⇒ a + b *



push (+)
push (c)



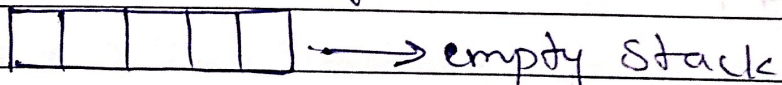
c → left ≠ NULL false
pop ()

print ⇒ a + b * c
c → right ≠ NULL false
pop()

print ⇒ a + b * c +

push (d)

d → left ≠ NULL ⇒ false
pop()



→ empty Stack

print ⇒ a + b * c + d.

ii) Non-recursive preorder:

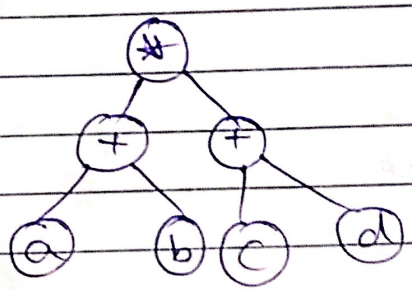
```
Procedure Preorder(T)
if T = NULL
    print "Empty Tree"
    return
top = 0
```

Rainbow

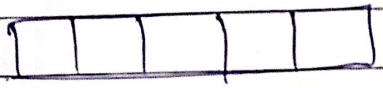
PAGE :
DATE : / /

```
while T ≠ NULL or top ≠ -1
    while T ≠ NULL
        print (T → data)
        push (S, top, T)
        T = T → left
    if top ≠ -1
    T = pop()
    T = T → right.
```

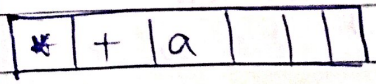
Example:



Stack:

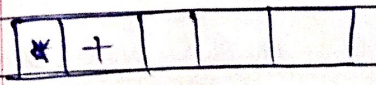


print ⇒ *
push (*)
print ⇒ * +
push (+)

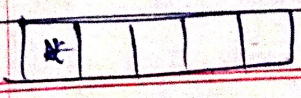


print ⇒ * + a
push (a)

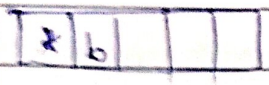
a → left ≠ NULL false
pop()



a → right ≠ NULL ⇒ false
pop()

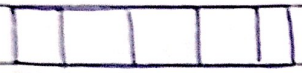


push (b)



b → left ≠ NULL ⇒ false
pop()

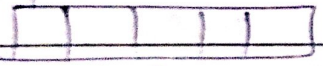
b → right ≠ NULL ⇒ false
pop()



print ⇒ * + ab

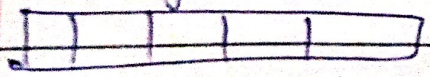
push (+) print ⇒ * +ab +
push (c)
print → * +ab + c

c → left ≠ NULL ⇒ false
pop()
c → right ≠ NULL ⇒ false
pop()



push (d)

d → left ≠ NULL ⇒ false
pop()
d → right ≠ NULL ⇒ false



print ⇒ * +ab + cd

Rainbow

PAGE:
DATE: / /

iii) Non-recursive postorder:

```
Procedure Postorder(T)
// int stack is stack
If T = NULL
    print "Empty stack"
    return
top = 0
while T ≠ NULL
    push(s top, T)
    Push(int stk, top, 1)
    T = T → left.
T = peep(s)
if int stk[top] = 2
    print(T → data)
    pop(s)
    T = NULL.
else
    int stk[top] = 2
    T = T → right
```

- Test cases / Validations:

1) Validation:

Number of operands & operators relationship.

2) Test cases:

Rainbow
PAGE:
DATE: / /

	Infix Expression	Postfix Expression	Prefix Expression
1)	A + B * C	ABC * +	+ A * CB
2)	A * B - C	AB * C -	- * A CB
3)	A ^ B - C	AB ^ C -	- ^ ABC
4)	A + B * C ^ E	ABCE ^ * +	+ A * B ^ CE
5)	A - B * C + A	ABC * - A +	- + A * BCA
6)	(A + B) / (C + D) ^ E ^ F - D * F - D	AB + CD + EF ^ ^ / DF * - D -	/ + AB + CD ^ EF * DFD
7)	A + B + C	AB + C +	+ + ABC
8)	A * B / C	AB * C /	/ * ACD
9)	A ^ B ^ C	ABC ^ ^	^ A ^ BC

- **Conclusion:**

Using Binary tree, it is possible to build an expression tree. Traversal of tree gives expression in various forms, i.e. inorder traversal for infix expression preOrder traversal for prefix expression, postorder traversal for postfix expression.