Rajdhani
DATE / /

Roll No. 23355

- Title : Assignment 4 : Expression Tree creation & Traversal

- Aim : To implement a expression tree using stack data structures.

- Problem statement :
Construct an expression tree for postfix expression and perform recursive and non-recursive Inorder, preorder and postorder traversal.

- Theory

- Concept of Nonlinear 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 involved. Therefore, we can't traverse all the elements in single run only.
- Non linear data structures are not easy to implement in comparison to linear data structure.
- But it ~~utr~~ utilizes computer memory efficiently in comparison to linear data structure.
- Examples of nonlinear datastructures:
Trees , graphs

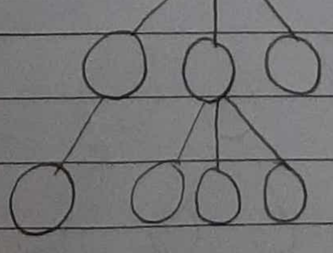


fig. Tree

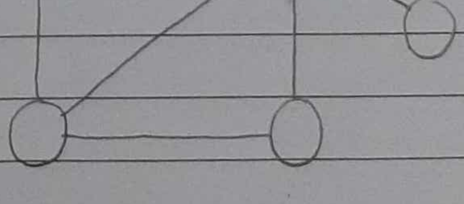


fig. Graph

- Binary tree

Tree in which any node can have atmost two branches i.e. at most 2 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'.

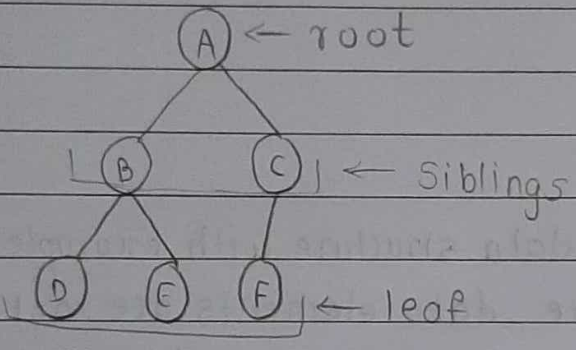


fig. binary Tree

Terminologies :

Root : Node without parent

Sibling : Nodes share the same parents

Internal nodes : Nodes with atleast 1 child

External nodes : Nodes without children

Ancestors of node : Parent, grand parent, grand-grandparents

Descendant of node : child, grand child, grand-grand child

Depth of node : Number of edges from root node

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.

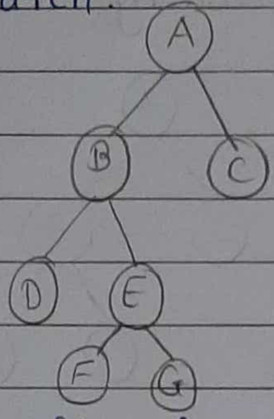


fig. full binary tree

Complete binary tree :

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

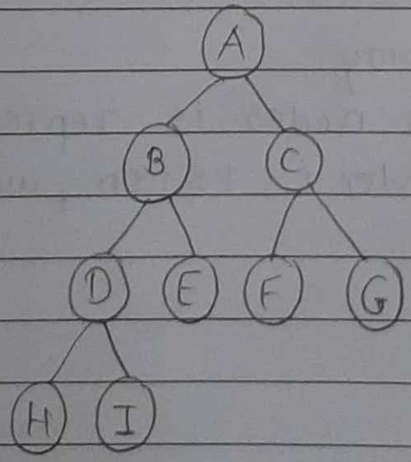


fig. Complete binary tree

- Binary tree ADT

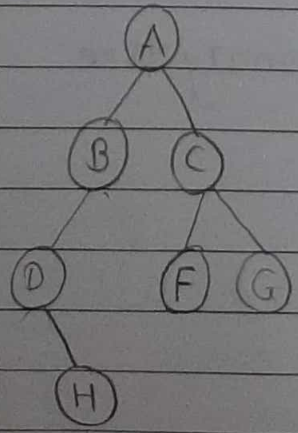
Structure Binary_Tree is a finite set of nodes either empty or consisting of rood node, left Binary_Tree and right Binary_Tree

Operations :

Bintree create ( )
boolean isEmpty ( )
Bintree MakeBT (bt1, item, bt2)
Bintree lChild (bt)
element data (bt)
Bintree rChild (bt)
void inorder (bt)
void preorder (bt)
void postorder (bt)

- Realization of ADT with Array

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 i
leftchild is at 2i
right child is at 2i+1



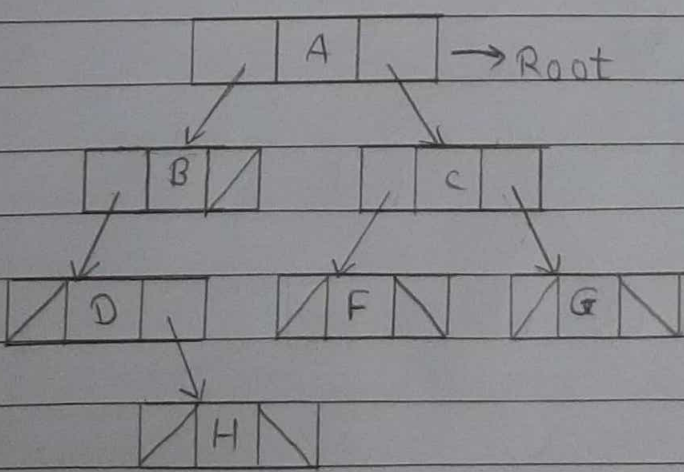
Array Representation:

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

- Realization of ADT with linked list
- Binary tree in 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:
  i) Data element
  ii) pointer that points towards left node
  iii) pointer that points towards right node.

left child	data	right child

Linked list representation:



Rajdhani
DATE / /

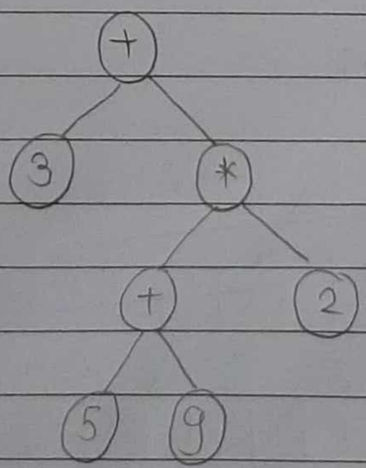
- Binary Tree applications :

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 and each leaf node corresponds to the operator.

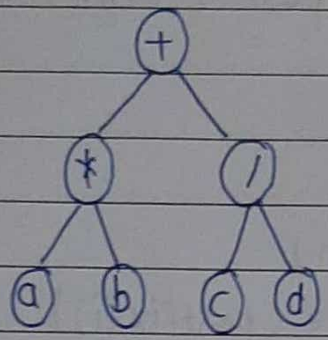
For example :
Infix : 3+((5+9)*2)



Rajdhani
DATE / /

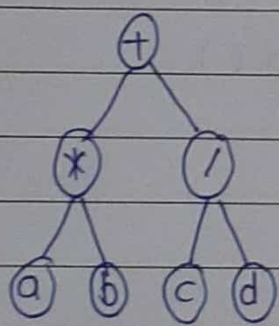
Example of prefix expression:

+ * a b / c d



Example of postfix expression

a b * c d / +



Applications of expression tree:

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

- Algorithm / Pseudocode :

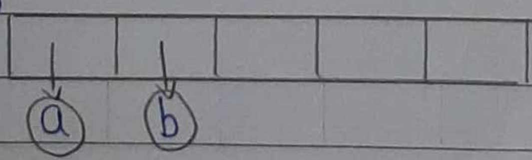
1) Expression tree creation from postfix expression

```
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()
```

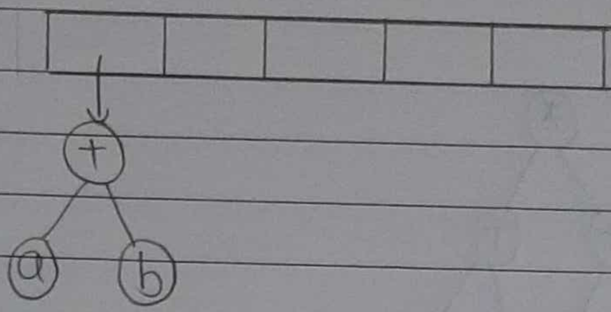
Example:
Postfix = ab+cd+*

Stack: 

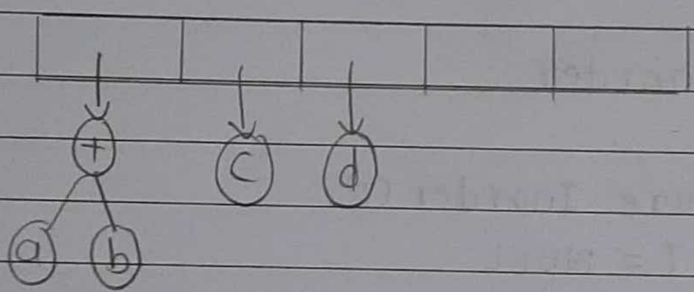
push (a)
push (b)



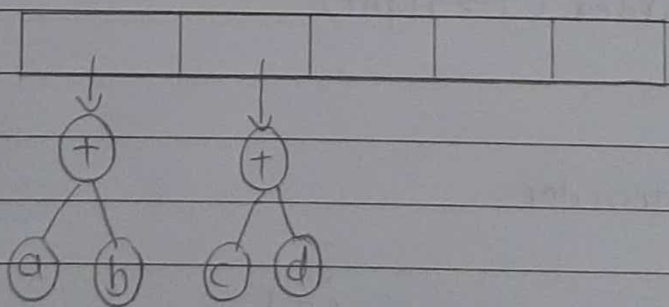
Next symbol is '+'. It pops two pointers from stack, a new tree is formed. pointer is pushed onto stack



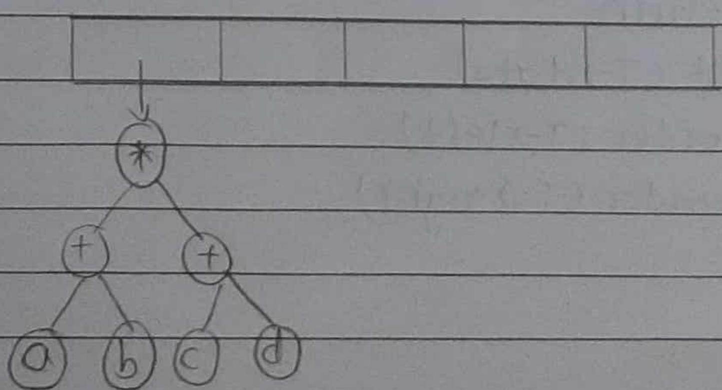
push (c)
push (d)



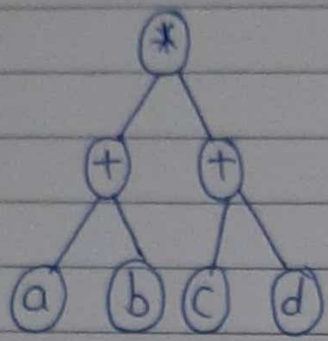
Next symbol is '+', pop two pointers . create new tree . push it into stack



Similar 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)
```

Rajdhani
DATE / /

iii) Recursive post order

```
Procedure Postorder (T)
If T = NULL
    return
Postorder (T->left)
Postorder (T->right)
print (T->data)
```

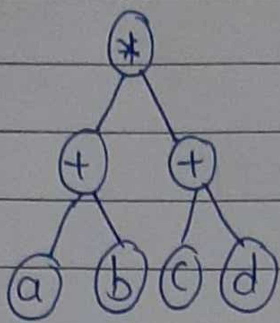
3) Nonrecursive traversal

i) Nonrecursive inorder

```
Procedur 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

    If top ≠ -1
        T = pop(S)
        print (T->data)
        T = T->right
```

Example:



Stack →

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

*	+	a		

a→left ≠ NULL ⇒ false
pop ()

Print ⇒ a
a→right ≠ NULL ⇒ false
pop()

print ⇒ a +

*				

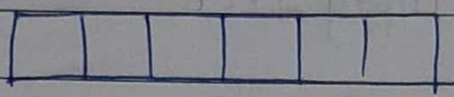
push (b)

*	b			

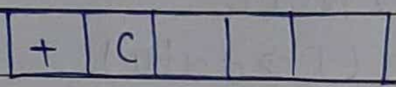
b→left ≠ NULL ⇒ false
pop ()
print ⇒ a + b

b → right ≠ NULL
pop ()

print => a+b *



push (+)
push (c)



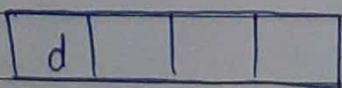
c → left ≠ NULL false
pop ()

print => a+b * c

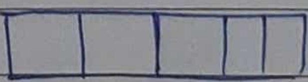
c → right ≠ NULL false
pop()

print => a+b * c+

push (d)



d → left ≠ NULL => false
pop ()

 → empty stack

print => a+b * c+d

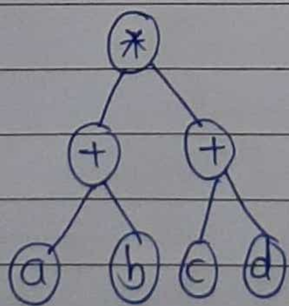
Rajdhani DATE / /

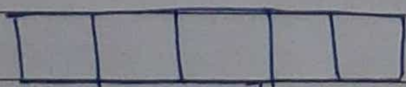
ii) Non-recursive preorder

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

a → left ≠ NULL false
pop ( )

*	+			

a → right ≠ NULL => false
pop ( )

*				

push (b)

*	b			

b → left ≠ NULL => false
pop ( )

b → right ≠ NULL => false
pop ( )

print => * + a b

push (+) print => * + a b +
push (c)
print => * + a b + c

c → left ≠ NULL => false
pop ( )

c → right ≠ NULL => false
pop ( )

push (d)

d				

d → left ≠ NULL false
~~≠~~ pop ( )

d → righ ≠ NULL false

Print => * + a b + c d

iii) Non recursive postorder:

```
Procedure Postorder (T)
// intstk is stack for flag
If T= NULL
    print "Empty stack"
    return
top=0
while T≠NULL OR top≠-1
    while T≠ NULL
        push (S, top, T)
        push (intstk, top, 1)
        T= T->left

    T= S.peep (S)
    If intstk[top] = 2
        print (T-> data)
        pop (S)
        T= NULL
    Else
        intstk [top] = 2
        T= T->right
```

- Test cases / validation

validation:
Number of operand and operator relationship

Test cases:

Sr. No.	Infix expression	Postfix expression	Prefix expression
1.	A+B*C	ABC*+	+A*CB
2.	A*B-C	AB*C-	-*ACB
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/	/*ABCB
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