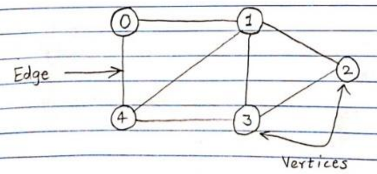
Assignment No. - 6

Title - Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent land marks as nodes and perform DFS and BFS on that.

Objective -
1. To identify directed and undirected graph.
2. To represent graph using adjacency matrix and list.
3. To traverse program to the graph.

Theory -
A graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as, A Graph consists of a Finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.



A graph is a data structure that consists of the following two components:
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u,v) called as edge. The pair is ordered because (u,v) is not the same as (v,u) in case of a directed graph (di-graph). The pair of the form (u,v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight / value / cost.

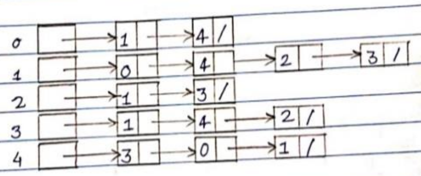
The following two are the most commonly used representation of a graph.
1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation - specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:
Adjacency Matrix is a 2D array of size VxV where V is the number of vertices in a graph. Let the 2D array be adj [][], a slot adj [i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj [i][j] = w, then there is an edge from vertex i to vertex j with weight w.

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

Adjacency list:
An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array []. An entry array [i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.



Breadth First Search or BFS is a graph traversal algorithm.
- It is used for traversing or searching a graph in a systematic fashion.
- BFS uses a strategy that searches in the graph in breadth first manner whenever possible.
- Queue data structure is used in the implementation of breadth first search.

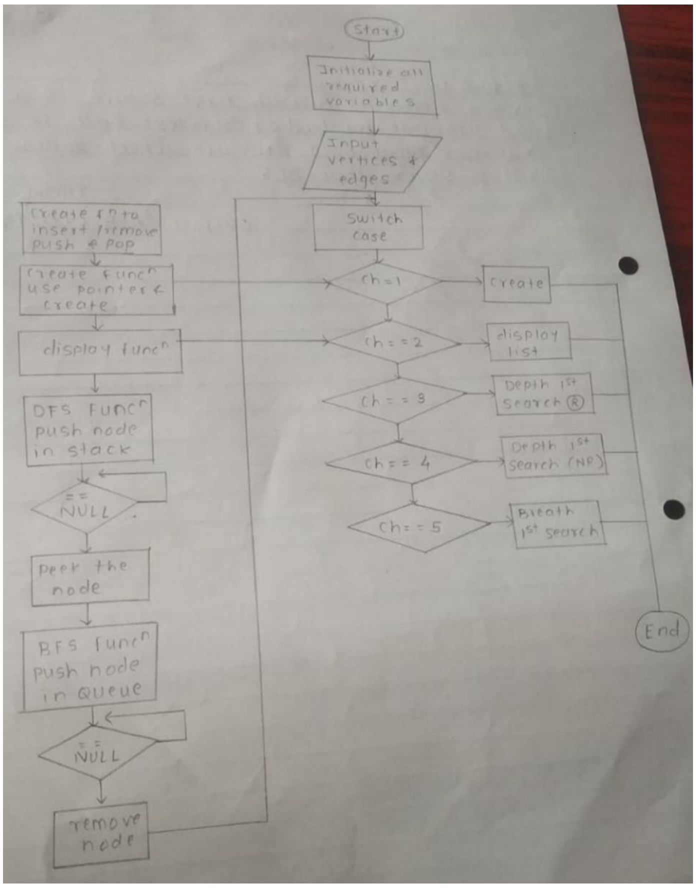
Breadth First Traversal (or search) for a graph is similar to Breadth First Traversal of a tree. Graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are rechable from the starting vertex.

Algorithm -

1) Create a recursive function that makes the index of node and a visited array.

2) Mark the current node as visited and print the node.

3) Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.




```
//Adjacency Matrix:  6 C13

//using adj matrix -BFS(Que)

#include<iostream>

#include<stdlib.h>

using namespace std;

int cost[10][10],i,j,k,n,qu[10],front,rear,v,visit[10],visited[10];

int stk[10],top,visit1[10],visited1[10];

int main()

{

    int m;

    cout <<"enter no of vertices";

    cin >> n;

    cout <<"enter no of edges";

    cin >> m;

    cout <<"\nEDGES \n";

  for(k=1;k<=m;k++)

  {

   cin >>i>>j;

   cost[i][j]=1;
```

```
  cost[j][i]=1;
 }
//display function
 cout<<"The adjacency matrix of the graph is:"<<endl;
  for(i=0;i<n;i++)
  {
  for(j=0;j<n;j++)
   {
  cout<<" "<<cost[i][j];
   }
  cout<<endl;
  }

  //for BFS
 cout <<"Enter initial vertex";
 cin >>v;
 cout <<"The BFS of the Graph is\n";
 cout << v;
 visited[v]=1;
 k=1;
 while(k<n)
   {
   for(j=1;j<=n;j++)
     if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)
     {
        visit[j]=1;
        qu[rear++]=j;
     }
     v=qu[front++];
     cout<<v << " ";
     k++;
     visit[v]=0; visited[v]=1;
   }
  //for DFS
 cout <<"Enter initial vertex";
 cin >>v;
```

```
 cout <<"The DFS of the Graph is\n";
 cout << v;
 visited[v]=1;
 k=1;
 while(k<n)
 {
  for(j=n;j>=1;j--)
  if(cost[v][j]!=0 && visited1[j]!=1 && visit1[j]!=1)
   {
      visit1[j]=1;
      stk[top]=j;
      top++;
   }
     v=stk[--top];
     cout<<v << " ";
     k++;
     visit1[v]=0; visited1[v]=1;
  }
}
```

OUTPUT

enter no of vertices 5

enter no of edges 4

EDGES

1

3

5

7

2

4

6

8

The adjacency matrix of the graph is:

0 0 0 0 0

0 0 0 1 0

0 0 0 0 1

0 1 0 0 0

0 0 1 0 0

Enter initial vertex 5

The BFS of the Graph is

50 0 0 0 Enter initial vertex 4

The DFS of the Graph is

42 4 0 0

Assignment No. 12

Title - Implementation of a direct access file - Insertion and deletion of a record from a direct access file.

Objective -
To understand concept of direct access file - Insertion and deletion.

Theory -
1) Different types of organizing the file -
i) Sequential file organization.
ii) Heap file organization
iii) Hash file organization
iv) B+ tree file organization
v) Clustered file organization

2) Direct access file organization :
i) Direct access file is also known as random access or relative file organization.
ii) In direct access file, all records are stored in direct access storage devices, such as hard disk. The records are randomly placed throughout the file.
iii) The records close not need to be in sequence becoz they are updated directly and rewritten back in the same location.
iv) This file organization is useful for immediate access to large amount of information. It is used in accessing large databases.
v) It is also called as hashing.

3) Difference betn sequential file and direct access

Sequential file access	Direct file access
i) Information in the file is processed in order one record after the other.	i) A field length logical record that allow the program to read & write record repidly in no particular order.
ii) When we used read command, it move ahead pointer, by one	ii) There is no restriction on the order of reading & writing for a direct access file.
iii) Data is entered in entry sequential order	iii) Data is entered in PRN number.
iv) Duplicate data be allowed.	iv) Duplicate data is not allowed
v) Access is slow	v) Access is faster than sequential access

4) Advantages of direct access file organization :
i) Direct access file helps in online transcation processing system like online railway reservation system.
ii) Indirect access file sorting of the records are not required.
iii) It access the desired records immediately.
iv) It update several files quickly.
v) It has better control over record allocation.

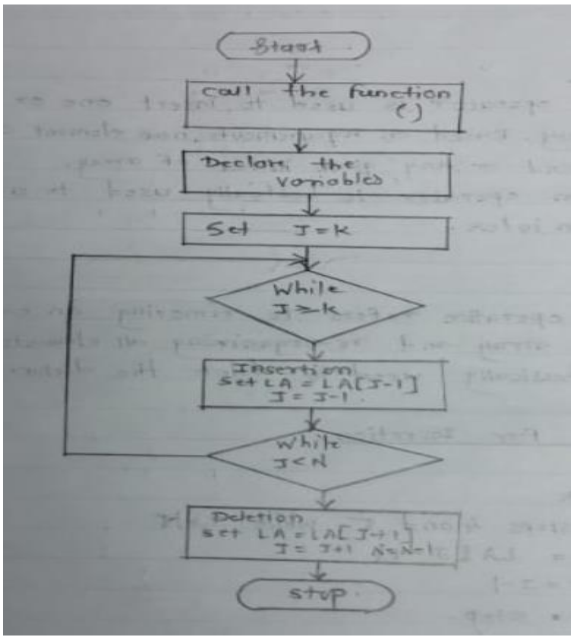
Disadvantages of direct access file organization :
i) Direct access file does not provides back up facility.
ii) It is expensive
iii) It has less storage space as compared to sequential file.

Insertion :
- Insertion operator is used to insert one or more data elements into an array. Based on requirements, new element can be added at the beginning, end or any given index of array.
- Insertion operator is basically used to add an element in the given index.

Deletion :
- Deletion operation refers to removing an existing element from the array and re-organizing all elements of an array.
- It is basically used to delete the data.

Algorithm for insertion -
1. Start
2. Set J = K
3. Repeat steps 4 & 5 while J = K
4. set LA = LA[J-1]
5. Set J=J-1
6. Stop

Algorithm for deletion -
1. Start
2. set J=K
3. Repeat the steps 4 & 5 while J<N
4. Set LA[J] = LA[J+1]
5. Set J = J+1
6. Set N = N-1
7. Stop.




```
#include <stdio.h>
#include <string.h>
typedef struct person {
    char lastName[15];
    char firstName[15];
    char age[4];
} Person;
void blank100();
void tenEntry();
void printTen();
void updateRecord();
void deleteRecord();
int main(void) {

    blank100();
    tenEntry();
    printTen();
    updateRecord();
    printTen();
    deleteRecord();
    printTen();
}
// Creates 100 blank entries
void blank100() {
```

```
    FILE* wfPtr;
    if ((wfPtr = fopen("nameage.dat", "wb")) == NULL) {
        puts("Error opening nameage.dat for writing");
        return;
    }
    Person blankPerson = { "", "", "0" };
    for (int i0 = 0; i0 < 100; ++i0) {
        fwrite(&blankPerson, sizeof(Person), 1, wfPtr);
    }
    fclose(wfPtr);
}
// Fills ten entries
void tenEntry() {
    FILE* wfPtr;
    if ((wfPtr = fopen("nameage.dat", "wb")) == NULL) {
        puts("Error opening nameage.dat for writing");
        return;
    }
    Person p0 = { "Anderly", "Charley", "54" };
    Person p1 = { "Schultz", "Gilberto", "67" };
    Person p2 = { "Cuevas", "Abbigail", "34" };
    Person p3 = { "Rhodes", "Gregory", "83" };
    Person p4 = { "Taylor", "Giovanna", "78" };
    Person p5 = { "Glenn", "Kole", "54" };
    Person p6 = { "Reilly", "Dennis", "41" };
    Person p7 = { "Morrix", "Shannon", "27" };
    Person p8 = { "Herman", "Mekhi", "84" };
    Person p9 = { "Beltran", "Damari", "30" };
    fwrite(&p0, sizeof(Person), 1, wfPtr);
    fwrite(&p1, sizeof(Person), 1, wfPtr);
    fwrite(&p2, sizeof(Person), 1, wfPtr);
    fwrite(&p3, sizeof(Person), 1, wfPtr);
    fwrite(&p4, sizeof(Person), 1, wfPtr);
    fwrite(&p5, sizeof(Person), 1, wfPtr);
    fwrite(&p6, sizeof(Person), 1, wfPtr);
    fwrite(&p7, sizeof(Person), 1, wfPtr);
```

```
    fwrite(&p8, sizeof(Person), 1, wfPtr);
    fwrite(&p9, sizeof(Person), 1, wfPtr);
    fclose(wfPtr);
}
// Prints the first ten entries
void printTen() {
    FILE* rfPtr;
    if ((rfPtr = fopen("nameage.dat", "rb")) == NULL) {
        puts("Error opening nameage.dat for reading");
        return;
    }
    Person toScreen = { "", "", "0" };
    for (int i0 = 0; fread(&toScreen, sizeof(Person), 1, rfPtr) && i0 < 10; ++i0) {
        printf("%s, %s, %s\n", toScreen.lastName, toScreen.firstName, toScreen.age);
    }
    puts("");
    fclose(rfPtr);
}
// updates one record with a new age value
void updateRecord() {
    FILE* rpfPtr;
    if ((rpfPtr = fopen("nameage.dat", "rb+")) == NULL) {
        puts("Error opening nameage.dat r+");
    }
    Person newData = { "", "", "0" };
    printf("Enter existing lastName, firstName, newAge: ");
    char line[50];
    fgets(line, 50, stdin);
    char* token = strtok(line, ", \n");
    strcpy(newData.lastName, token);
    token = strtok(NULL, ", \n");
    strcpy(newData.firstName, token);
    token = strtok(NULL, ", \n");
    strcpy(newData.age, token);
    Person search = { "", "", "0" };
    while (fread(&search, sizeof(Person), 1, rpfPtr) && strcmp(search.lastName, newData.lastName)) {
```

```
    }
    if (strcmp(search.lastName, newData.lastName) == 0) {
        fseek(rpfPtr, -1 * (int)sizeof(Person), SEEK_CUR);
        fwrite(&newData, sizeof(Person), 1, rpfPtr);
    }
    else {
        puts("Record not found.\n");
    }
    fclose(rpfPtr);
}
// deletes one record
void deleteRecord() {
    FILE* fPtr;
    if ((fPtr = fopen("nameage.dat", "rb+")) == NULL) {
        puts("Error opening nameage.dat r+");
    }
    Person userEntry = { "", "", "0" };
    printf("Delete existing lastName, firstName, age: ");
    char line[50];
    fgets(line, 50, stdin);
    char* token = strtok(line, ", \n");
    strcpy(userEntry.lastName, token);
    token = strtok(NULL, ", \n");
    strcpy(userEntry.firstName, token);
    token = strtok(NULL, ", \n");
    strcpy(userEntry.age, token);
    Person search = { "", "", "0" };
    while (fread(&search, sizeof(Person), 1, fPtr) && strcmp(search.lastName, userEntry.lastName)) {
    }
    if (strcmp(search.lastName, userEntry.lastName) == 0) {
        fseek(fPtr, -1 * (int)sizeof(Person), SEEK_CUR);
        Person blank = { "", "", "" };
        fwrite(&blank, sizeof(Person), 1, fPtr);
    }
    else {
        puts("Record not found.\n");
```

```
    }
    fclose(fPtr);
}
```

OUTPUT

Anderly, Charley, 54

Schultz, Gilberto, 67

Cuevas, Abbigail, 34

Rhodes, Gregory, 83

Taylor, Giovanna, 78

Glenn, Kole, 54

Reilly, Dennis, 41

Morrix, Shannon, 27

Herman, Mekhi, 84

Beltran, Damari, 30

Enter existing lastName, firstName, newAge: Rhodes, Jeff, 44

Anderly, Charley, 54

Schultz, Gilberto, 67

Cuevas, Abbigail, 34

Rhodes, Jeff, 44

Taylor, Giovanna, 78

Glenn, Kole, 54

Reilly, Dennis, 41

Morrix, Shannon, 27

Herman, Mekhi, 84

Beltran, Damari, 30

Delete existing lastName, firstName, age: Rhodes, Abe, 3

Anderly, Charley, 54

Schultz, Gilberto, 67

Cuevas, Abbigail, 34

, ,

Taylor, Giovanna, 78

Glenn, Kole, 54

Reilly, Dennis, 41

Morrix, Shannon, 27

Herman, Mekhi, 84

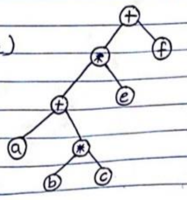
Beltran, Damari, 30

Assignment No-3 [Group-B]

Title ~~Theory~~ - Construct an expression tree from the given prefix expression eg. +--a*bc/def and traverse it using post order traversal (non recursive) and then delete the entire tree.

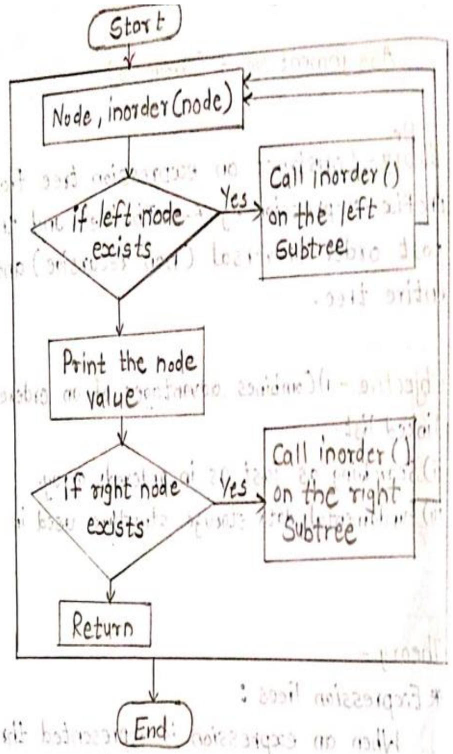
Objective - i) Combines advantages of an ordered array and a linked list.
ii) Searching as fast as in ordered array.
iii) Fundamental data storage structure used in programming.

Theory -
* Expression Trees :
When an expression is represented through a tree, it is known as expression tree. The leaves of an expression tree are operands, such as constants or variables names and all internal nodes contain operations. Figure gives an example of an expression tree.
(a+b*c)*e+f
A preorder traversal on the expression tree gives prefix equivalent of the expression. A postorder traversal on the expression tree gives postfix equivalent of the expression.
Prefix (expression tree of figure)
= +*+a*bcef
Postfix (expression tree of figure)
= abc*+e*f+



Flow...

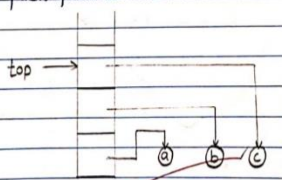
1) Inorder -



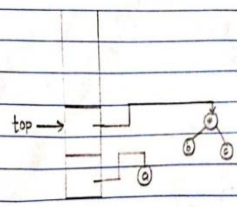
Constructing an expression tree
Here, we will discuss an algorithm for constructing an expression tree from a postfix expression. In case an expression tree is to be converted from infix expression, infix expression should be converted to postfix.

Algorithm:
We read our expression one symbol at a time. If the sybol is an operand, we create a one node tree and push a pointer to it onto a stack. If the symbol is an operator, we pop pointer to two trees T_2 and T_1 from the stack and form a new tree whose root is the operator and whose left and right childern point to T_1 and T_2 respectively. A pointer to this new tree is then pushed onto the stack.
As an example, suppose the input is abc*+e*f+

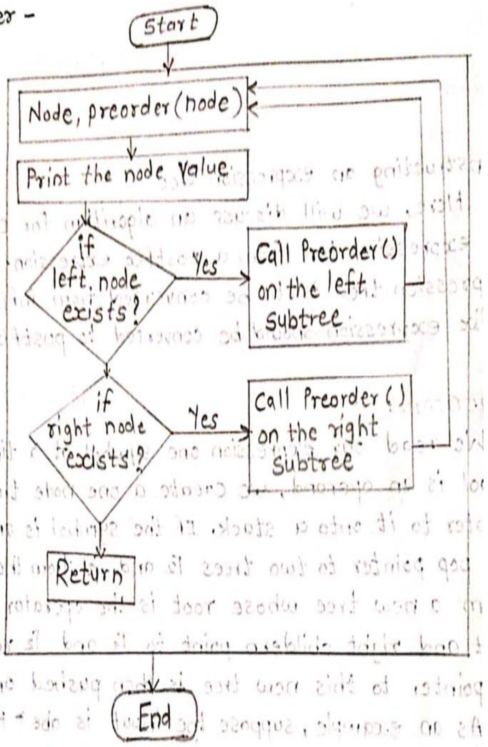
1) The first three symbols are operands, so we create one-node trees push pointers to them onto a stack.

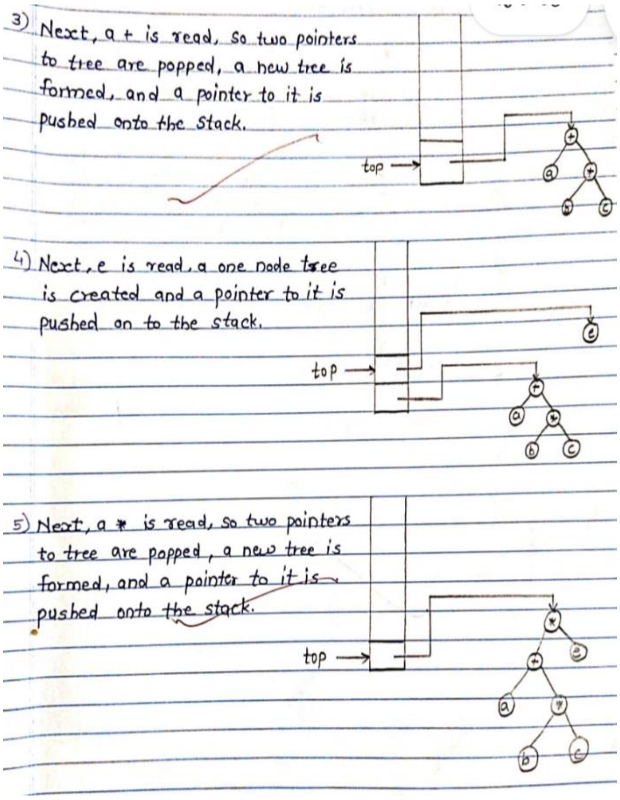


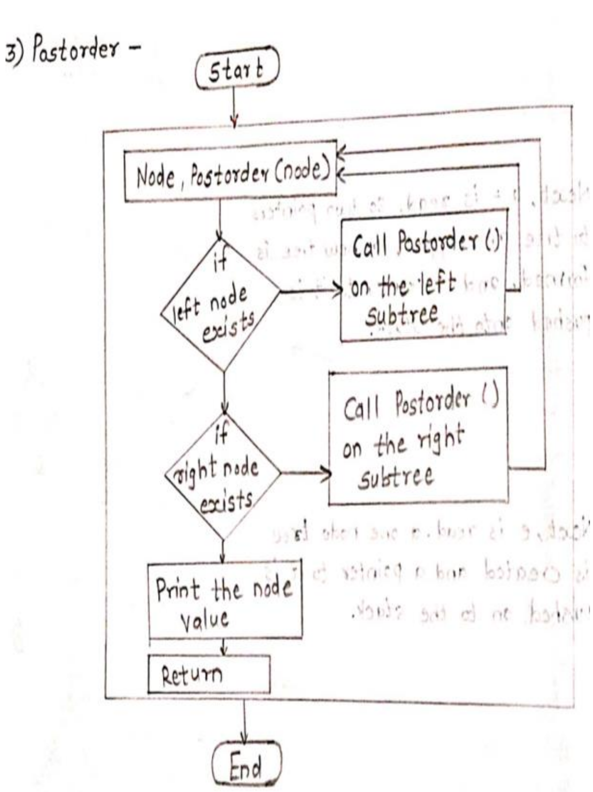
2) Next, 4* is read, so two pointers to tree are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

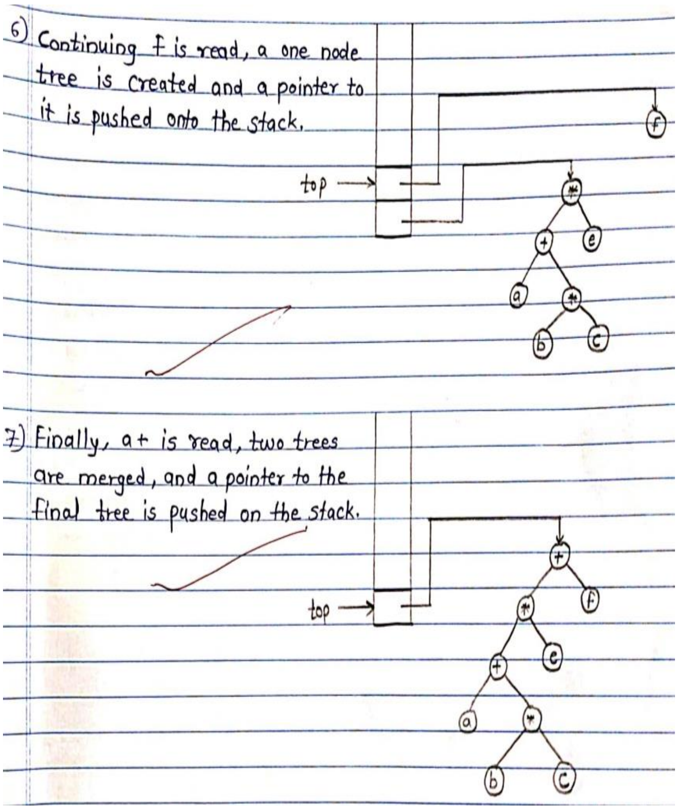


2) Preorder -










```
#include <iostream>

using namespace std;

#include<string.h>

struct node
```

```
{
char data;
node *left;
node *right;
};
class tree
{
char prefix[20];
public: node *top;
void expression(char []);
void display(node *);
void non_rec_postorder(node *);
void del(node *);
};
class stack1
{
node *data[30];
int top;
public:
stack1()
{top=-1;
}
int empty()
{
if(top==-1)
return 1;
return 0;
}
void push(node *p)
{
data[++top]=p;
}
node *pop()
{
return(data[top--]);
}
```

```
};
void tree::expression(char prefix[])
{char c;
stack1 s;
node *t1,*t2;
int len,i;
len=strlen(prefix);
for(i=len-1;i>=0;i--)
{top=new node;top->left=NULL;
top->right=NULL;
if(isalpha(prefix[i]))
{
top->data=prefix[i];
s.push(top);
}
else if(prefix[i]=='+'||prefix[i]=='*'||prefix[i]=='-'||prefix[i]=='/')
{
t2=s.pop();
t1=s.pop();
top->data=prefix[i];
top->left=t2;
top->right=t1;
s.push(top);
}
}
top=s.pop();
}
void tree::display(node * root)
{
if(root!=NULL)
{
cout<<root->data;
display(root->left);display(root->right);
}
}
void tree::non_rec_postorder(node *top)
```

```
{
stack1 s1,s2;
/*stack s1 is being used for flag . A NULL data
implies that the right subtree has not been visited */
node *T=top;
cout<<"\n";
s1.push(T);
while(!s1.empty())
{
T=s1.pop();
s2.push(T);
if(T->left!=NULL)
s1.push(T->left);
if(T->right!=NULL)
s1.push(T->right);
}
while(!s2.empty())
{
top=s2.pop();
cout<<top->data;
}}
void tree::del(node* node)
{if (node == NULL) return;
/* first delete both subtrees */
del(node->left);
del(node->right);
/* then delete the node */
cout<<" Deleting node:"<<node->data;
free(node);
}
int main()
{
char expr[20];
tree t;
cout<<"Enter prefix Expression: ";
cin>>expr;
```

```
cout<<expr;
t.expression(expr);
//t.display(t.top);
//cout<<endl;
t.non_rec_postorder(t.top);
// t.del(t.top);
// t.display(t.top);
}
```

OUTPUT

```
Enter prefix Expression: +--a*bc/def
+--a*bc/def
abc*-de/-f+
```