Name:N. Hemashirisha
Rollno:2019103525

1. Implement depth first search using adjacency matrix.
   Code:

```
#include<stdio.h>
#include<string.h>
#define size 100
char stack[size];
int top=-1;


void push(char c){
    if(top==size){
        printf("Overflow\n");
    }
    else{
    stack[++top]=c;
    }
}

void pop(){
    if(top==-1){
        printf("Underflow\n");
    }
    else{
        top--;
    }
}
char peek(){
    return stack[top];
}

int notIn(char c){
    int i;
    for(i=0;i<=top;i++){
        if(c==stack[i]){
            return 0;
        }
    }
    return 1;
}
int notVisited(char closed[],char c,int n){
        int i;
```

```
        for(i=0;i<n;i++){
            if(c==closed[i]){
                return 0;
            }

        }
        return 1;
}
int dfs(int a[][20],int n){

       int i=0,j=0,k=0,l=0;
       char closed[n];
       char c='A';
       char path[n];
       push(c);
       while(top!=-1&&i<n&&i>=0){
          // display();
           c=peek();
           //char t='A';
           //i=c-65;
           path[k]=peek();
           pop();
           closed[l++]=path[k];
           k++;
           for(j=0;j<n;j++){
               //c='A'+j;
               c='A';
               if(a[i][j]==1){
                   c=c+j;
                   if(notIn(c)&&notVisited(closed,c,l)){
                       push(c);
                   }
               }
           }
           i++;
       }

       path[k]='\0';
       closed[k]='\0';
    if(strlen(path)!=n){
        printf("Its a disconnected graph\n");
    }
    printf("path: %s\n",path);
```

```
}

int main(){
    int a[20][20],n,i,j;
    printf("Enter the no.of nodes\n");
    scanf("%d",&n);
    printf("Enter the matrix\n");
    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            scanf("%d",&a[i][j]);
        }
    }
    char nodes[n];
    printf("The adjacency matrix:\n");
    printf("  |");
    for(i=0;i<n;i++){
        printf("%c ",'A'+i);
        nodes[i]='A'+i;
    }
    printf("\n");
    for(i=0;i<n;i++){
    printf("---");
    }
    printf("\n");
    for(i=0;i<n;i++){
        printf("%c |",'A'+i);
        for(j=0;j<n;j++){
            printf("%d ",a[i][j]);
        }
        printf("\n");
    }
    dfs(a,n);

    return 0;
}
```

Output:

```
C:\MinGW\bin\DSA\dsaLab>dfs
Enter the no.of nodes
5
Enter the matrix
0 1 0 1 0
1 0 1 0 1
0 1 0 0 1
1 0 0 0 0
0 1 1 0 0
The adjacency matrix:
  |A B C D E
--------------
A |0 1 0 1 0
B |1 0 1 0 1
C |0 1 0 0 1
D |1 0 0 0 0
E |0 1 1 0 0
path: ADECB

C:\MinGW\bin\DSA\dsaLab>dfs
Enter the no.of nodes
3
Enter the matrix
0 1 1
1 0 0
1 0 0
The adjacency matrix:
  |A B C
---------
A |0 1 1
B |1 0 0
C |1 0 0
path: ACB
```

Time complexity analysis:

n = Number of nodes

The push, pop and peek functions take constant time:
Worst case time complexity of push, pop and peek functions: O(1)

notVisited() function:
Worst case time complexity: O(n)

notIn() function:
Worst case time complexity: O(n)

Dfs() function:
Since there is a for loop inside a while loop:
Worst case time complexity: O(n^2)

Main function:
Here there are 2 nested for loops,
Worst case time complexity: O(n^2)

Over time complexity: O(n^2)

---

2. Implement Breadth first search using adjacency list

Code:

```
#include<stdio.h>

#define size 100

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

int q[size];
int rear=-1,front=-1;

//int front=-1,rear=-1;

struct node* insert(struct node *root,int c){
    struct node* ptr=(struct node*)malloc(sizeof(struct node));
    ptr->data=c;
    //ptr->next=NULL;
    if(root==NULL){

        root=ptr;
        ptr->next=NULL;
    }
    else{
        ptr->next=root;
        root=ptr;
    }
    return root;
}

void display(struct node *root){
    struct node *temp=root;
    while(temp!=NULL){
        printf("%d ",temp->data);
        temp=temp->next;
    }
    printf("\n");
```

```
}

void push(int c){
    if(front==-1&&rear==-1){
        front=0;
    }
    q[++rear]=c;
}
void pop(){
    if(front==-1){
        printf("Underflow\n");
    }
    else{
        if(front==rear){
            front=-1;
            rear=-1;
        }
        else{
            front++;
        }
    }
}

char peek(){
    return q[front];
}
int index(int c,int nodes[],int n){
    int i;
    for(i=0;i<n;i++){
        if(nodes[i]==c){
            return i;
        }
    }
}
int notVisited(int path[],int c,int n){
    int i;
    for(i=0;i<n;i++){
        if(c==path[i]){
            return 0;
        }
    }
    return 1;
}
```

```
int presentQ(int c){
    int i;
    for(i=front;i<=rear;i++){
        if(q[i]==c){
            return 0;
        }
    }
    return 1;
}

//void bfs()
void bfs(int nodes[],struct node* adjacencyList[],int n){
        int i=0,j,k=0,t=0,ind;
        int visited[n],path[n];
        int c=nodes[0];
        push(c);
        while(front!=-1&&i<n){
            c=peek();
            ind=index(c,nodes,n);
            path[k++]=peek();
            pop();
            struct node* temp=adjacencyList[ind];
            while(temp!=NULL){

if(notVisited(path,temp->data,k)&&presentQ(temp->data)){
                    push(temp->data);
                }
                temp=temp->next;
            }


        }
        printf("Path: \n");
        for(i=0;i<k;i++){
            printf("%d ",path[i]);
        }
        printf("\n");


}
```

```c
int main(){
    int n,i,j,k,n1;
    int c;
    printf("Enter no. of nodes\n");
    scanf("%d",&n);
    int nodes[n];
    struct node* adjacentNodes[n];
    //fflush(stdin);
    printf("Enter the nodes:\n");
    for(i=0;i<n;i++){
        scanf("%d",&nodes[i]);
    }

    //fflush(stdin);
    for(i=0;i<n;i++){
        printf("Enter the no.of adjacent nodes to %d \n",nodes[i]);
        scanf("%d",&n1);
        printf("Enter the adjacent nodes: \n");
        adjacentNodes[i]=(struct node*)malloc(sizeof(struct node));
        adjacentNodes[i]=NULL;

        for(j=0;j<n1;j++){
            //fflush(stdin);
            scanf("%d",&c);

            adjacentNodes[i]=insert(adjacentNodes[i],c);

        }
    }

    for(i=0;i<n;i++){
        printf("%d : ",nodes[i]);
        display(adjacentNodes[i]);
    }
    bfs(nodes,adjacentNodes,n);


    return 0;
}
```

```
C:\MinGW\bin\DSA\dsaLab>bfs
Enter no. of nodes
7
Enter the nodes:
1 2 3 4 5 6 7
Enter the no.of adjacent nodes to 1
2
Enter the adjacent nodes:
3 6
Enter the no.of adjacent nodes to 2
2
Enter the adjacent nodes:
3 7
Enter the no.of adjacent nodes to 3
4
Enter the adjacent nodes:
1 2 4 5
Enter the no.of adjacent nodes to 4
1
Enter the adjacent nodes:
3
Enter the no.of adjacent nodes to 5
1
Enter the adjacent nodes:
3
Enter the no.of adjacent nodes to 6
1
Enter the adjacent nodes:
1
Enter the no.of adjacent nodes to 7
1
Enter the adjacent nodes:
2
```

```
 1 : 6 3
 2 : 7 3
 3 : 5 4 2 1
 4 : 3
 5 : 3
 6 : 1
 7 : 2
 Path:
 1 6 3 5 4 2 7
```

Time complexity analysis:
The push, pop, peek and insert functions take constant time:
Worst case time complexity of push, pop, peek and insert functions: O(1)

notVisited() function:
Worst case time complexity: O(n)

index() function:
Worst case time complexity: O(n)

presentQ() function:
Worst case time complexity: O(n)

bfs() function:
Worst case time complexity: O(e+n) where e is the no.of edges and n is the no.of nodes

Overall worst case time complexity: O(e+n)

---

3. Implement double threaded binary tree:

```c
#include<stdio.h>
#define size 100

struct node{
    int data;
    struct node *left;
    struct node *right;
    int leftThread,rightThread,leaf;
}*root=NULL;
int flag=1;
struct node *q[size];
int front=-1,rear=-1;

void enqueue(struct node* temp){
    if(front==-1){
        front=0;
    }
    q[++rear]=temp;
}
void dequeue(){
    if(front==-1){
        printf("Underflow\n");
    }
    else if(front==rear){
        front=-1;
        rear=-1;
    }
    else{
        front++;
    }
}
void insert(int c){
```

```
struct node *temp=(struct node*)malloc(sizeof(struct node));
temp->data=c;
temp->left=NULL;
temp->right=NULL;
temp->leftThread=0;
temp->rightThread=0;
temp->leaf=1;
struct node *ptr;
if(rear!=-1){
    ptr=q[front];
}
if(root==NULL){
    root=temp;
}
else if(rear%2==0){
   if(ptr->leftThread){

    ptr->leftThread=0;
    temp->leftThread=1;
    temp->left=ptr->left;
    }
    ptr->left=temp;
    temp->rightThread=1;
    temp->right=ptr;
    ptr->leaf=0;
}
else{
    if(ptr->rightThread){
    ptr->rightThread=0;
    temp->rightThread=1;
    temp->right=ptr->right;
    }
    ptr->right=temp;
    temp->leftThread=1;
    temp->left=ptr;
    ptr->leaf=0;

}

enqueue(temp);
if(rear%2==0&&rear!=0){
    dequeue();
}
```

```
}
struct node* leftMost(struct node *root2){
    struct node* tmp=root2;
    if(root2==NULL){
        return NULL;
    }
    while(tmp->leaf==0){
        tmp=tmp->left;
    }

    return tmp;
}
void Inorder(){
    struct node *ptr=leftMost(root);
    //int i=0;
    while(ptr!=NULL){
        printf("%d ",ptr->data);
        if(ptr->rightThread==1){
            ptr=ptr->right;
            }
        else{
             ptr=leftMost(ptr->right);
        }
    }
}

int main(){
    int tree,i,j,k,n;
    struct node *ptr=(struct node*)malloc(sizeof(struct node));
    ptr=NULL;
    printf("Enter no.of nodes\n");
    scanf("%d",&n);
    printf("Enter the nodes in order\n");
    for(i=0;i<n;i++){
        scanf("%d",&tree);
        insert(tree);
    }
    //printf("haaa\n");
    printf("Inorder display:\n");
    Inorder(root);
    printf("\n");
    return 0;
}
```

Output:

```
C:\MinGW\bin\DSA\dsaLab>tb
Enter no.of nodes
10
Enter the nodes in order
5 4 6 3 7 2 8 1 9 10
Inorder display:
1 3 9 4 10 7 5 2 6 8

C:\MinGW\bin\DSA\dsaLab>tb
Enter no.of nodes
5
Enter the nodes in order
1 2 3 4 5
Inorder display:
4 2 5 1 3
```

Time complexity analysis:
The enqueue, dequeue and the insert functions take constant time,
Worst time complexity: O(1)

leftMost() function:
Worst case complexity: O(d) where d is the depth of the tree

Inorder() function:
Worst case complexity: O(n) where n is the no.of. Nodes in the tree.

Overall time complexity:
Worst case time complexity: O(n)