```
//binary search


#include<bits/stdc++.h>
using namespace std;
int main(){
   int arr[13]={0,1,2,4,4,6,7,8,9,10,11,12,13};
   int n = 13,loc;
   int beg=0;
   int en = 12;
   int mid = (beg+en)/2;
   int item = 4;
   while(beg<=en && arr[mid]!=item){
      if(item<arr[mid])
         en=mid-1;
      else beg=mid+1;
      mid=(beg+en)/2;
   }
   if(beg>en)cout<<"doesn't exist"<<endl;
   else cout<<mid <<endl;
}
```

```
//bubble sort


#include<bits/stdc++.h>
using namespace std;
int main(){
   int arr[13]={4,5,2,88,4,32,11,99,0,654,3,1,2};
   int n = 13;
   for(int i = 0; i<n-1;i++){
      for(int j=0;j<n-i-1;j++){
         if(arr[j]>arr[j+1])
            swap(arr[j],arr[j+1]);
      }
   }
   cout<<n<<endl;
   for(int i = 0; i< n;i++){
      cout<<arr[i]<<" ";
   }
}
```

```
//create, traverse, insert and delete linked list

#include<bits/stdc++.h>
using namespace std;
int size;
class node{
public:
   int data;
   node* next;
   node(int val){
      data = val;
      next = NULL;
   }
};

void insertAtHead(node* &head, int val){
   node* n = new node(val);
   if(head==NULL){
      head=n;
      size++;
      return;
   }
   n->next = head;
   head = n;
   size++;
}

void insertAtTail(node* &head, int val){
   node* n = new node(val);
```

```
    if(head==NULL){
        head=n;
        size++;
        return;
    }
    node* temp = head;
    while(temp->next!=NULL){
        temp=temp->next;
    }
    temp->next = n;
    size++;
}

void insertAtPos(node* &head, int val, int pos){

    if(pos==1){
        insertAtHead(head,val);
        return;
    }
    if(pos==size+1){
        insertAtTail(head,val);
        return;
    }
    node* n = new node(val);
    node* temp = head;
    int k = 1;
    while(temp!=NULL){
        k++;
        if(k==pos){
```

```
            break;
        }
        temp=temp->next;
    }
    if(pos>size+1){
        cout<<"No such position exist"<<endl;
        return;
    }
    n->next=temp->next;
    temp->next = n;
    size++;
}

bool search(node* head, int item){
    node* temp = head;
    while(temp!=NULL){
        if(temp->data == item){
            return true;
        }
        temp=temp->next;
    }
    return false;
}

void display(node* head){
    node* temp = head;
    while(temp!=NULL){
        cout<<temp->data<<" ";
        temp=temp->next;
```

```
    }
    cout<<endl;
}

void deletion(node* &head, int val){
    node* temp = head;
    if(head->data==val){
        head=head->next;
        return;
    }
    node* prev = NULL;
    while(temp->next!=NULL){
        if(temp->next->data==val){
            break;
        }
        temp=temp->next;
    }
    temp->next = temp->next->next;
}

node* reverse(node* &head){
    node* pre = NULL;
    node* next;
    node* cur = head;

    while(cur!=NULL){
        next = cur->next;
        cur->next = pre;
        pre = cur;
```

```
        cur = next;
    }
    return pre;
}

void sort(node* &head){
    node* i, *j;
    for(i=head;i->next!=NULL;i=i->next){
        for(j=i->next;j!=NULL;j=j->next){
            if(i->data>j->data){
                int temp = i->data;
                i->data = j->data;
                j->data = temp;
            }
        }
    }
}

int main(){
    node* head = NULL;
    insertAtTail(head,1);
    insertAtTail(head,2);
    insertAtTail(head,3);
    insertAtHead(head,4);
    display(head);
    insertAtPos(head,10,2);
    display(head);
    insertAtPos(head,11,1);
    display(head);
```

```
    insertAtPos(head,12,7);
    display(head);
    insertAtPos(head,13,7);
    display(head);
    insertAtPos(head,13,10);
    display(head);
    cout<<search(head,4)<<endl;
    cout<<search(head,14)<<endl;
    deletion(head,12);
    display(head);
    deletion(head,4);
    display(head);
    deletion(head,11);
    display(head);
    head = reverse(head);
    display(head);
    sort(head);
    display(head);
}
```

```
//evaluate_postfix

#include<bits/stdc++.h>
using namespace std;
int top,mxstk;
double STACK[1000];
class my_stack{
public:
   void push(double item){
      top = top +1;
      STACK[top]=item;
   }

   void pop(){
      top= top -1;
   }
   double topp(){
      return STACK[top];
   }
};

bool isOperand(char s){
   if(s>='0' && s<=57)
      return true;
   return false;
}

double operation(char s, double b, double a){
   if(s=='+'){
```

```
        return a+b;
    }
    if(s=='^'){
        return pow(a,b);
    }
    if(s=='-'){
        return a-b;
    }
    if(s=='/' && b!=0){
        return a/b;
    }
    if(s=='*'){
        return a*b;
    }
    return INT_MIN;
}

int main(){
    my_stack st; //creating object
    string s;
    getline(cin,s);
    for(int i=0;i<s.size();i++){
        if(s[i]==' ' || s[i]==',')
            continue;
        if(isOperand(s[i])){
            st.push(s[i]-'0');
        }
        else{
            double b = st.topp();
```

```
            st.pop();
            double a = st.topp();
            st.pop();
            double res = operation(s[i],b,a);
            st.push(res);
        }
    }
    double ans = st.topp();
    cout<<ans <<endl;
}
```

```
//evaluate_postfix_multidigit


#include<bits/stdc++.h>
using namespace std;
int top,mxstk;
double STACK[1000];
class my_stack{
public:
   void push(double item){
      top = top +1;
      STACK[top]=item;
   }

   void pop(){
      top= top -1;
   }
   double topp(){
      return STACK[top];
   }
};

bool isOperand(string s){
   if(s[0]>='0' && s[0]<=57)
      return true;
   return false;
}
double sto(string s){
   double num=0;
```

```
    int k = 1;
    for(int i=s.size()-1;i>=0;i--){
        num+= (s[i]-'0')*k*1.0;
        //cout<<num<<endl;
        k=k*10;
    }
    return num;
}

double operation(string s, double b, double a){
    if(s=="+"){
        return a+b;
    }
    if(s=="^"){
        return pow(a,b);
    }
    if(s=="-"){
        return a-b;
    }
    if(s=="/" && b!=0){
        return a/b;
    }
    if(s=="*"){
        return a*b;
    }
    return INT_MIN;
}

int main(){
```

```
    my_stack st; //creating object


    string s;
    while(1){
        if(s=="exit")break;
        cin>>s;
        if(isOperand(s)){
            double num = sto(s);
            st.push(num);
            //cout<<num<<endl;
        }
        else{
            double b = st.topp();
            st.pop();
            double a = st.topp();
            st.pop();
            double result = operation(s,b,a);
            st.push(result);
            cout<<result<<endl;
        }
    }
    double ans = st.topp();
    cout<<ans <<endl;
}
```

//factorial

```
#include<bits/stdc++.h>
using namespace std;

long long int fact(long long int n){
   if(n==1 || n==0)return n;
   return n*fact(n-1);
}

int main(){
   long long int n;
   cout<<"Enter the number: ";
   cin>>n;
   cout<<"Factorial of the number = "<<fact(n)<<endl;
}
```

//**Fibonacci**

```
#include<bits/stdc++.h>
using namespace std;

long long int fibo(long long int n){
   if(n==1 || n==0)return n;
   return fibo(n-1)+fibo(n-2);
}

int main(){
   long long int n;
   cout<<"Enter the number: ";
   cin>>n;
   cout<<"Fibanacci of the number = "<<fibo(n)<<endl;
}
```

```
//hashing 1


#include<bits/stdc++.h>
using namespace std;

int size,item;

void setvalue(int Hash[]){
   for(int i=0;i<size;i++){
      Hash[i]=-1;
   }
}

void display(int Hash[]){
   for(int i=0;i<size;i++){
      if(Hash[i]==-1){
         cout<<"none ";
      }
      else
      cout<<Hash[i]<<" ";
   }
   cout<<endl;
}

void linear_probing(int Hash[], int arr[], int n){
   int i,j;
   setvalue(Hash);
   for(i=0;i<n;i++){
```

```
    int hv = arr[i]%size;
    if(Hash[hv]==-1){
        Hash[hv]=arr[i];
    }
    else{
        for(j=1;j<size;j++){
            int t = (hv+j)%size;
            if(Hash[t]==-1){
                Hash[t]=arr[i];
                break;
            }
        }
    }
}
display(Hash);
int v = item%size;
for(j=0;j<size;j++){
    int t = (v+j)%size;
    if(Hash[t]==-1){
        cout<<"Doesn't exist"<<endl;
        break;
    }
    else if(Hash[t]==item){
        cout<<"Found at "<<t<<endl;
        break;
    }
    else continue;
}
if(j==size){
```

```
        cout<<"Doesn't exist"<<endl;
    }
}

void plus3_probing(int Hash[], int arr[], int n){
    int i,j;
    setvalue(Hash);
    for(i=0;i<n;i++){
        int hv = arr[i]%size;
        if(Hash[hv]==-1){
            Hash[hv]=arr[i];
        }
        else{
            for(j=1;j<size;j++){
                int t = (hv+j*3)%size;
                if(Hash[t]==-1){
                    Hash[t]=arr[i];
                    break;
                }
            }
        }
    }
    display(Hash);
    int v = item%size;
    for(j=0;j<size;j++){
        int t = (v+j*3)%size;
        if(Hash[t]==-1){
            cout<<"Doesn't exist"<<endl;
            break;
```

```
        }
        else if(Hash[t]==item){
            cout<<"Found at "<<t<<endl;
            break;
        }
        else continue;
    }
    if(j==size){
        cout<<"Doesn't exist"<<endl;
    }
}

void quadratic_probing(int Hash[], int arr[], int n){
    int i,j;
    setvalue(Hash);
    for(i=0;i<n;i++){
        int hv = arr[i]%size;
        if(Hash[hv]==-1){
            Hash[hv]=arr[i];
        }
        else{
            for(j=1;j<size;j++){
                int t = (hv+j*j)%size;
                if(Hash[t]==-1){
                    Hash[t]=arr[i];
                    break;
                }
            }
        }
```

```
    }
    display(Hash);
    int v = item%size;
    for(j=0;j<size;j++){
        int t = (v+j*j)%size;
        if(Hash[t]==-1){
            cout<<"Doesn't exist"<<endl;
            break;
        }
        else if(Hash[t]==item){
            cout<<"Found at "<<t<<endl;
            break;
        }
        else continue;
    }
    if(j==size){
        cout<<"Doesn't exist"<<endl;
    }
}

void chaining(int arr[], int n){
    int i,j;
    vector<int>Hash[size];
    for(i=0;i<n;i++){
        int hv = arr[i]%size;
        Hash[hv].push_back(arr[i]);
    }
    for(i=0;i<size;i++){
        int k =0;
```

```
        for(j=0;j<Hash[i].size();j++){
            cout<<Hash[i][j]<<" ";
            k++;
        }
        if(k==0)cout<<"none ";
        cout<<endl;
    }
    int k=0;
    for(i=0;i<size;i++){
        for(j=0;j<Hash[i].size();j++){
            if(Hash[i][j]==item){
                k++;
                break;
            }
        }
        if(k)break;
    }
    if(k)cout<<"found at "<<i<<endl;
    else cout<<"doesn't exist"<<endl;
}


int main(){
    int n,i;
    cin>>n>>size;
    int arr[n],Hash[size];
    for(i=0;i<n;i++){
        cin>>arr[i];
    }
```

```
    cout<<"Search item : ";
    cin>>item;
    linear_probing(Hash,arr,n);
    plus3_probing(Hash,arr,n);
    quadratic_probing(Hash,arr,n);
    chaining(arr,n);
    return 0;
}
```

```
//hashing 2


#include<bits/stdc++.h>
using namespace std;

int size,item;

void setvalue(int Hash[]){
   for(int i=0;i<size;i++){
      Hash[i]=-1;
   }
}

void display(int Hash[]){
   for(int i=0;i<size;i++){
      if(Hash[i]==-1){
         cout<<"none ";
      }
      else
      cout<<Hash[i]<<" ";
   }
   cout<<endl;
}

void linear_probing(int Hash[], int arr[], int n){
   int i,j;
   setvalue(Hash);
   for(i=0;i<n;i++){
```

```
    int hv = arr[i]%size;
    if(Hash[hv]==-1){
        Hash[hv]=arr[i];
    }
    else{
        for(j=1;j<size;j++){
            int t = (hv+j)%size;
            if(Hash[t]==-1){
                Hash[t]=arr[i];
                break;
            }
        }
    }
}
display(Hash);
int v = item%size;
for(j=0;j<size;j++){
    int t = (v+j)%size;
    if(Hash[t]==-1){
        cout<<"Doesn't exist"<<endl;
        break;
    }
    else if(Hash[t]==item){
        cout<<"Found at "<<t<<endl;
        break;
    }
    else continue;
}
if(j==size){
```

```
        cout<<"Doesn't exist"<<endl;
    }
}

void plus3_probing(int Hash[], int arr[], int n){
    int i,j;
    setvalue(Hash);
    for(i=0;i<n;i++){
        int hv = arr[i]%size;
        if(Hash[hv]==-1){
            Hash[hv]=arr[i];
        }
        else{
            for(j=1;j<size;j++){
                int t = (hv+j*3)%size;
                if(Hash[t]==-1){
                    Hash[t]=arr[i];
                    break;
                }
            }
        }
    }
    display(Hash);
    int v = item%size;
    for(j=0;j<size;j++){
        int t = (v+j*3)%size;
        if(Hash[t]==-1){
            cout<<"Doesn't exist"<<endl;
            break;
```

```
        }
        else if(Hash[t]==item){
            cout<<"Found at "<<t<<endl;
            break;
        }
        else continue;
    }
    if(j==size){
        cout<<"Doesn't exist"<<endl;
    }
}

void quadratic_probling(int Hash[], int arr[], int n){
    int i,j;
    setvalue(Hash);
    for(i=0;i<n;i++){
        int hv = arr[i]%size;
        if(Hash[hv]==-1){
            Hash[hv]=arr[i];
        }
        else{
            for(j=1;j<size;j++){
                int t = (hv+j*j)%size;
                if(Hash[t]==-1){
                    Hash[t]=arr[i];
                    break;
                }
            }
        }
```

```
    }
    display(Hash);
    int v = item%size;
    for(j=0;j<size;j++){
        int t = (v+j*j)%size;
        if(Hash[t]==-1){
            cout<<"Doesn't exist"<<endl;
            break;
        }
        else if(Hash[t]==item){
            cout<<"Found at "<<t<<endl;
            break;
        }
        else continue;
    }
    if(j==size){
        cout<<"Doesn't exist"<<endl;
    }
}

void chaining(int arr[], int n){
    int i,j;
    vector<int>Hash[size];
    for(i=0;i<n;i++){
        int hv = arr[i]%size;
        Hash[hv].push_back(arr[i]);
    }
    for(i=0;i<size;i++){
        int k =0;
```

```
        for(j=0;j<Hash[i].size();j++){
            cout<<Hash[i][j]<<" ";
            k++;
        }
        if(k==0)cout<<"none ";
        cout<<endl;
    }
    int k=0;
    for(i=0;i<size;i++){
        for(j=0;j<Hash[i].size();j++){
            if(Hash[i][j]==item){
                k++;
                break;
            }
        }
        if(k)break;
    }
    if(k)cout<<"found at "<<i<<endl;
    else cout<<"doesn't exist"<<endl;
}


int main(){
    int n,i;
    cin>>n>>size;
    int arr[n],Hash[size];
    for(i=0;i<n;i++){
        cin>>arr[i];
    }
```

```
    cout<<"Search item : ";
    cin>>item;
    linear_probing(Hash,arr,n);
    plus3_probing(Hash,arr,n);
    quadratic_probing(Hash,arr,n);
    chaining(arr,n);
    return 0;
}
```

//infix to postfix

```
#include<bits/stdc++.h>
using namespace std;
int top,mxstk;
double STACK[1000];
class my_stack{
public:
   void push(char item){
      top = top +1;
      STACK[top]=item;
   }

   void pop(){
      top= top -1;
   }
   double topp(){
      return STACK[top];
   }

   bool empty(){
      if(top==0){
         return true;
      }
      return false;
   }
};

bool isOperand(char s){
```

```
    if(s>='A' && s<='Z')
        return true;
    return false;
}

bool check(char pr, char ex){
    if(ex=='^'){
        return true;
    }
    if(ex=='/' || ex=='*'){
        if(pr=='^')
            return false;
        return true;
    }
    if(ex=='+' || ex=='-'){
        if(pr=='+' || pr=='-'){
            return true;
        }
        return false;
    }
    return false;
}

bool isOperator(char s){
    if(s=='+'){
        return true;
    }
    if(s=='^'){
        return true;
```

```
    }
    if(s=='-'){
        return true;
    }
    if(s=='/'){
        return true;
    }
    if(s=='*'){
        return true;
    }
    return false;
}

double operation(char s, double b, double a){
    if(s=='+'){
        return a+b;
    }
    if(s=='^'){
        return pow(a,b);
    }
    if(s=='-'){
        return a-b;
    }
    if(s=='/' && b!=0){
        return a/b;
    }
    if(s=='*'){
        return a*b;
    }
```

```
    return INT_MIN;
}

int main(){
    my_stack st; //creating object
    string s,p="";
    cin>>s;
    s.push_back(')');
    st.push('(');
    for(int i=0;i<s.size();i++){
        if(isOperand(s[i])){
            p.push_back(s[i]);
        }
        else if(s[i]=='('){
            st.push('(');
        }
        else if(isOperator(s[i])){
            while(!st.empty()){
                char c = st.topp();
                if(check(s[i],c)){
                    st.pop();
                    p.push_back(c);
                }
                else break;
            }
            st.push(s[i]);
        }
        else{
            while(!st.empty()){
```

```
            char c = st.topp();
            st.pop();
            if(c=='(')break;
            p.push_back(c);
          }
        }
    }
    cout<<p <<endl;
}
```

//**insert and delete_stack**

```
#include<bits/stdc++.h>
using namespace std;
int top,n;
int STACK[1001];
class my_stack{
public:
   void push(int item){
      if(top==n){
         cout<<"Overflow"<<endl;
         return;
      }
      top = top +1;
      STACK[top]=item;
   }

   void pop(){
      if(top==0){
         cout<<"Underflow"<<endl;
         return;
      }
      top= top -1;
   }
   bool empty(){
      if(top==0){
         return true;
      }
```

```
        return false;
    }
};

void display(){
    for(int i=1;i<=top;i++){
        cout<<STACK[i]<<" ";
    }
    cout<<endl;
}

int main(){
    my_stack st; //creating object
    n = 1000;
    for(int i = 1; i<=10;i++)
    {
        int x;
        cin>>x;
        st.push(x);
    }

    display();

    // The item to be pushed on stack

    int item;
    cin>>item;
    st.push(item);
    cin>>item;
```

```
    st.push(item);
    cin>>item;
    st.push(item);
    display();

    st.pop();
    display();
}
```

```
//linked list
#include<bits/stdc++.h>
using namespace std;

class node{
public:
   int data;
   node* next;
};

int main(){
   node* head = new node();
   node* one = NULL;
   node* two = NULL;
   node* three = NULL;
   one = new node();
   two = new node();
   three = new node();
   one->data = 4;
   two->data = 5;
   three->data = 1;
   one->next = two;
   two->next = three;
   head = one;
   while(head!=NULL){
      cout<<head->data<<endl;
      head = head->next;
   }
}
```

//**matrix multi**

```
#include<bits/stdc++.h>
using namespace std;
int main(){
   int r1,c1,r2,c2;
   cin>>r1>>c1>>r2>>c2;
   int a[r1][c1],b[r2][c2];
   int i,j,k;
   int ans[r1][c2];
   for(i=0;i<r1;i++){
      for(j=0;j<c1;j++){
         cin>>a[i][j];
      }
   }
   for(i=0;i<r2;i++){
      for(j=0;j<c2;j++){
         cin>>b[i][j];
      }
   }
   for(i=0;i<r1;i++){
      for(j=0;j<c2;j++){
         ans[i][j]=0;
         for(k=0;k<c1;k++){
            ans[i][j]+=a[i][k]*b[k][j];
         }
      }
   }
   for(i=0;i<r1;i++){
```

```
        for(j=0;j<c2;j++){
            cout<<ans[i][j]<<" ";
        }
        cout<<endl;
    }
}
```

//**infixEvaluation**

```
#include<bits/stdc++.h>
using namespace std;

int getPriority(char ch) {

   if(ch == '+' || ch == '-') return 1;
   else if(ch == '*' || ch == '/') return 2;
   else if(ch == '^') return 3;
   else if((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9') || (ch == '.')) return
0;
   else return -1;

}

string infixToPostFix(string infix) {
   stack<char>stk;
   int i = 0;

   string postfix = "";
   for(i = 0; infix[i]; i++) {
      char ch = infix[i];
      if(ch == '(') stk.push(ch);
      else if(ch == ')') {
         while(!stk.empty() && stk.top() != '(') {
            postfix += stk.top();
            postfix += ',';
            stk.pop();
```

```
        }
        stk.pop();
    }

    else {
        int priority = getPriority(ch);
        if(priority == 0) {
            while(getPriority(infix[i]) == 0) {
                postfix += infix[i];
                i++;
            }
            i--;
            postfix += ',';
        }
        else {
            if(stk.empty()) stk.push(ch);
            else {
            while(!stk.empty() && stk.top() != '(' && (priority <= getPriority(stk.top()))) {
                postfix += stk.top();
                postfix += ',';
                stk.pop();
            }
            stk.push(ch);
            }
        }
    }
}
while(!stk.empty()) {
    postfix += stk.top();
```

```
      postfix += ',';
      stk.pop();
   }
   postfix.erase(postfix.end()-1);
   return postfix;
}

double calculate(double a, double b, char ch) {
   switch(ch) {
      case '+':
         return a+b;
      case '-':
         return a-b;
      case '*':
         return a*b;
      case '/':
         return a/b;
      case '^':
         return pow(a, b);
   }
}

double postfixEvaluate(string postfix) {
   stack<double>stk;

   int i;
   for(i = 0; i < postfix[i]; i++) {
      char ch = postfix[i];
```

```
        if(ch == ',' || ch == ' ') continue;
        else if(ch >= '0' && ch <= '9') {
            string str = "";
            while(postfix[i] != ',' && postfix[i]) {
                str += postfix[i];
                i++;
            }
            double num = stod(str);
            stk.push(num);
        }
        else if(ch >= 'a' && ch <= 'z'  || ch >= 'A' && ch < 'Z') {

            double value;
            cout << "Enter the value of " << ch << " : ";
            cin >> value;
            stk.push(value);
        }

        else {
            double b = stk.top();
            stk.pop();
            double a = stk.top();
            stk.pop();
            stk.push(calculate(a, b, ch));
        }
    }
    return stk.top();
}
```

```
int main() {

    string infix = "a+5.0*(2+3)-b";
    string postfix = infixToPostFix(infix);

    cout << "Infix : " << infix << endl;
    cout << "Postfix : " << postfix << endl;

    double ans = postfixEvaluate(postfix);

    cout << "Answer : " << ans << endl;

    return 0;
}
```

```
//binary_search_tree


#include<iostream>

using namespace std;

class Node {
   public:
      int data;
      Node *left, *right;
   Node(int data) {
      this->data = data;
      left = right = NULL;
   }
};

Node * makeTree() {
   return NULL;
}

Node * insert(Node *root, int data) {
   if(root == NULL) root = new Node(data);
   else {
      if(data < root->data) root->left = insert(root->left, data);
      else root->right = insert(root->right, data);
   }
   return root;
}
```

```
void printTree(Node *root) {
   if(root == NULL) return;
   printTree(root->left);
   cout << root->data << " ";
   printTree(root->right);
}

bool searchTree(Node* root, int data) {
   if(root == NULL) return false;
   if(root->data == data) return true;
   else {
      if(root->data > data) searchTree(root->left, data);
      else searchTree(root->right, data);
   }
}

int minValue(Node *root) {
   if(root->left == NULL) return root->data;

   else return minValue(root->left);
}

Node* deleteItem(Node *root, int data) {
   if(root == NULL) return root;
   else if(data < root->data) root->left = deleteItem(root->left, data);
   else if(data > root->data) root->right = deleteItem(root->right, data);

   else {
```

```
        if(root->left == NULL && root->right == NULL) {
            delete root;
            root = NULL;
            return root;
        }
        else if(root->left == NULL) {
            Node *temp = root;
            root = root->right;
            return root;
        }
        else if(root->right == NULL) {
            Node *temp = root;
            root = root->left;
            delete temp;

            return root;
        }
        else {
            int tempValue = minValue(root->right);
            root->data = tempValue;

            root->right = deleteItem(root->right, tempValue);

            return root;
        }
    }
}

int main() {
```

```
    Node *root;
    root = makeTree();
    int a[] = {7, 5, 6, 3, 4, 9, 8, 2, 1, 10};

    // insert data in the tree
    for(int i = 0; i < 10; i++) {
        root = insert(root, a[i]);
    }

    // Print the whole tree inOrder order
    printTree(root);
    cout << endl;

    // Search an element in the tree...
    bool flag = searchTree(root, 11);
    cout << flag << endl;

    // delete a specific item from the tree
    deleteItem(root, 7);

    printTree(root);
    cout << endl;

    return 0;
}
```

```
//bst_inserting


#include <iostream>

using namespace std;

class Node
{
public:
   int data;
   Node *left, *right;
   Node(int item)
   {
      data = item;
      left = right = NULL;
   }
};

Node *insert(Node *root, int item)
{
   Node *currNode = root;
   Node *newNode = new Node(item);

   while (currNode)
   {
      if (currNode->data > item)
      {
         if (currNode->left)
```

```
            currNode = currNode->left;
        else
        {
            currNode->left = newNode;
            break;
        }
    }
    else if (currNode->data < item)
    {
        if (currNode->right)
            currNode = currNode->right;
        else
        {
            currNode->right = newNode;
            break;
        }
    }
    else
        return root;
  }
  return root ? root : newNode;
}

void postOrder(Node *root)
{
  if (root == NULL)
    return;
  postOrder(root->left);
  postOrder(root->right);
```

```
    cout << root->data << " ";
}

void inOrder(Node *root)
{
    if (root == NULL)
        return;
    inOrder(root->left);
    cout << root->data << ' ';
    inOrder(root->right);
}

int main()
{
    Node *root = NULL;
    int key[] = {7, 6, 4, 5, 9, 8, 10, 25, 12, 9, 11, 2, 3, 1};
    for (int item : key)
    {
        root = insert(root, item);
    }
    cout << root->data << endl;
    postOrder(root);
    cout << endl;
    inOrder(root);

    return 0;
}
```

```
//unique_bst

#include<iostream>
using namespace std;

#define SPACE 10

class Node {
   public:
   int data;
   Node *left, *right;

   Node(int data) {
      this->data = data;
      this->left = this->right = NULL;
   }
};

typedef Node node;

node *root = NULL;
int loc;

void insert(int data) {
   if(root == NULL) {
      root = new Node(data);
      return;
   }
   node *temp = root;
```

```
    loc = 1;
    while(temp != NULL) {
        if(temp->data > data) {
            if(temp->left != NULL) {
                temp = temp->left;
                loc *= 2;
            }
            else {
                temp->left = new Node(data);
                return;
            }
        }
        else if(temp->data < data) {
            if(temp->right != NULL) {
                temp = temp->right;
                loc = loc*2 + 1;
            }
            else {
                temp->right = new Node(data);
                return;
            }
        }
        else {
            cout << "Found at pos : " << loc << endl;
            return;
        }
    }
}
```

```
void printTree(node *root, int space) {
   if(root == NULL) return;

   space += SPACE;
   printTree(root->right, space);
   cout << endl;
   for(int i = SPACE; i < space; i++) cout << " ";
   // cout << endl;
   cout << root->data << endl;
   printTree(root->left, space);
}

int minNodeValue(node *root) {
   if(root->left == NULL) return root->data;
   return minNodeValue(root->left);
}

node* deleteNodeValue(node *root, int value) {

   if(root == NULL) return root;
   else if(value < root->data) root->left = deleteNodeValue(root->left, value);
   else if(value > root->data) root->right = deleteNodeValue(root->right, value);

   else {
      if(root->left == NULL && root->right == NULL) return NULL;
      else if(root->left == NULL) {
         node *temp = root->right;
         free(root);
         return temp;
```

```
        }
        else if(root->right == NULL) {
            node *temp = root->left;
            free(root);
            return temp;
        }
        int minValue = minNodeValue(root->right);
        root->data = minValue;
        root->right = deleteNodeValue(root->right, minValue);
    }
    return root;
}

void inorder(node *root) {
    if(root == NULL) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

int main() {

    while(true) {
        cout <<"Enter node (-1 for exit): ";
        int x;
        cin >> x;
        if(x == -1) break;
        insert(x);
        printTree(root, 10);
```

```
        cout << endl;
    }

    printTree(root, SPACE);

    cout << "Enter a value to delete : ";
    int x;
    cin >> x;
    root = deleteNodeValue(root, x);
    printTree(root, SPACE);

    cout << "Inorder result : ";
    inorder(root);

    return 0;
}
```

```
//BinaryTree

import java.util.Queue;
import java.util.LinkedList;

public class BinaryTree {

   Node root;

   class Node {
      int value;
      Node right, left;

      public Node(int value) {
         this.value = value;
      }
   }

   public void createBinaryTree() {
      root = null;
   }

   public void insert(int value) {
      Node newNode = new Node(value);

      if (root == null) {
         root = newNode;
      } else {
         Queue<Node> queue = new LinkedList<Node>();
```

```
        queue.add(root);

        while (!queue.isEmpty()) {
            Node currNode = queue.remove();
            if (currNode.left == null) {
                currNode.left = newNode;
                break;
            } else if (currNode.right == null) {
                currNode.right = newNode;
                break;
            } else {
                queue.add(currNode.left);
                queue.add(currNode.right);
            }
        }

    }
}

public boolean search(int value) {
    if (root == null)
        return false;
    Queue<Node> queue = new LinkedList<Node>();
    queue.add(root);

    while (!queue.isEmpty()) {
        Node currNode = queue.remove();

        if (currNode.value == value)
```

```
            return true;
        else {
            if (currNode.left != null)
                queue.add(currNode.left);
            else if (currNode.right != null)
                queue.add(currNode.right);
        }
    }
    return false;

}

public Node getDeepestNode() {

    Node currNode = root;

    Queue<Node> queue = new LinkedList<Node>();
    queue.add(root);
    while (!queue.isEmpty()) {
        currNode = queue.remove();
        if (currNode.left != null)
            queue.add(currNode.left);
        if (currNode.right != null)
            queue.add(currNode.right);
    }
    return currNode;
}

public void deleteDeepestNode() {
```

```
    if (root == null) {
        System.out.println("Tree is empty!");
        return;
    } else {
        Node currNode, prevNode;
        currNode = prevNode = null;
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(root);
        while (!queue.isEmpty()) {
            prevNode = currNode;
            currNode = queue.remove();
            if(currNode.left == null) {
                prevNode.right = null;
                return;
            }
            else if(currNode.left == null) {
                currNode.left = null;
                return;
            }
            queue.add(currNode.left);
            queue.add(currNode.right);
        }
    }
}

public void deleteNodeValue(int value) {
    if (root == null) {
        System.out.println("Tree is empty");
        return;
```

```
    } else {
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(root);

        while (!queue.isEmpty()) {
            Node currNode = queue.remove();
            if (currNode.value == value) {
                System.out.println(currNode.value + " Deleted");
                currNode.value = getDeepestNode().value;
                deleteDeepestNode();
                break;
            } else {
                if (currNode.left != null) {
                    queue.add(currNode.left);
                }
                if (currNode.right != null) {
                    queue.add(currNode.right);
                }
            }
        }
    }
}

public void inorder(Node root) {
    if (root == null)
        return;

    inorder(root.left);
    System.out.print(root.value + " ");
```

```
    inorder(root.right);
}

public void deleteBinaryTree() {
    root = null;
}

public static void main(String[] args) {
    BinaryTree bTree = new BinaryTree();

    bTree.createBinaryTree();
    bTree.insert(25);
    bTree.insert(26);
    bTree.insert(27);
    bTree.insert(28);
    bTree.insert(29);
    // System.out.println(bTree.root.value);

    System.out.print("Inorder : ");
    bTree.inorder(bTree.root);
    System.out.println();

    int value = 26;
    boolean s_value = bTree.search(value);
    if (s_value == true)
        System.out.println(value + " Found! in the Tree");
    else
        System.out.println(value + " Not found in the Tree");
```

```
        bTree.deleteNodeValue(25);
        System.out.print("Inorder Traversing : ");
        bTree.inorder(bTree.root);
        System.out.println();
        // System.out.println(bTree.root.value);
        bTree.deleteBinaryTree();
        bTree.deleteNodeValue(26);

        System.out.println("Tree was Deleted!");
    }
}
```

```
//binarytreeTraversal

import java.util.*;

public class Traversal {

   public Node root = null;
   class Node {
      int value;
      Node left, right;

      public Node(int value) {
         this.value = value;
         this.left = this.right = null;
      }
   }

   public void createTree() {
      root = null;
   }

   public void insert(int value) {
      Node newNode = new Node(value);
      if(root == null) {
         root = newNode;
         return;
      }
      else {
         Queue<Node> queue = new LinkedList<Node>();
```

```
        queue.add(root);
        while(!queue.isEmpty()) {
            Node tempNode = queue.remove();

            if(tempNode.left == null) {
                tempNode.left = newNode;
                return;
            }
            else if(tempNode.right == null) {
                tempNode.right = newNode;
                return;
            }
            else {
                queue.add(tempNode.left);
                queue.add(tempNode.right);
            }
        }
    }
}
// preorder traversing iterative method
public List<Integer> preOrderTraversing() {
    List<Integer> list = new ArrayList<Integer>();

    if(root == null) return list;

    Stack<Node> stack = new Stack<Node>();
    stack.push(root);

    while(!stack.isEmpty()) {
```

```
        Node node = stack.peek();
        list.add(node.value);
        stack.pop();

        if(node.right != null) stack.push(node.right);
        if(node.left != null) stack.push(node.left);
    }

    return (list);
}

public List<Integer> inOrderTraversing() {
    List<Integer> list = new ArrayList<Integer>();

    if(root == null) return list;

    Stack<Node> stack = new Stack<Node>();
    Node currNode = root;

    while(!stack.isEmpty() || currNode != null) {
        while(currNode != null) {
            stack.push(currNode);
            currNode = currNode.left;
        }
        currNode = stack.pop();
        list.add(currNode.value);
        currNode = currNode.right;
    }
    return list;
```

```
}

public List<Integer> postOrderTraversing() {
   List<Integer> list = new ArrayList<Integer>();
   if(root == null) return list;

   Stack<Node> stack = new Stack<Node>();

   Node currNode;
   stack.push(root);
   while(!stack.isEmpty()) {
      currNode = stack.pop();
      list.add(currNode.value);
      if(currNode.left != null) stack.push(currNode.left);
      if(currNode.right != null) stack.push(currNode.right);
   }
   return list;
}

public static void main(String[] args) {
   Traversal tree = new Traversal();
   tree.createTree();
   //int a[] = new int[10];
   int a[] = {1,2,3,4,5,6,7,8,9,10};

   for(int i = 0; i < a.length; i++) {
      tree.insert(a[i]);
   }
   System.out.print("PreOrder Traversing : ");
```

```
        List<Integer> preList = new ArrayList<Integer>();
        preList = tree.preOrderTraversing();

        for(int i = 0; i < preList.size();  i++) {
           System.out.print(preList.get(i) + " ");
        }

        System.out.println();
        System.out.print("Inorder Traversing : ");
        List<Integer> inList = new ArrayList<Integer>();
        inList = tree.inOrderTraversing();

        for(int i = 0; i < inList.size();  i++) {
           System.out.print(inList.get(i) + " ");
        }
        System.out.println();
        System.out.print("Post Order Traversing : ");
        List<Integer> postList = new ArrayList<Integer>();
        postList = tree.postOrderTraversing();

        for(int i = 0; i < postList.size();  i++) {
           System.out.print(postList.get(i) + " ");
        }
    }
}
```

//**bTree_level_order_traversing**

```
#include<iostream>
#include<queue>
using namespace std;

class Node {
public:
   int data;
   Node *left, *right;
   Node(int data) {
      this->data = data;
      this->left = this->right = NULL;
   }
   Node() {

   }

   Node* insert(Node *root, int data) {
      if(root == NULL) {
         root = new Node(data);
         return root;
      }
      queue<Node *> q;
      q.push(root);

      while(!(q.empty())) {
         Node *currNode = q.front();
```

```
        q.pop();

        if(!currNode->left) {
            currNode->left = new Node(data);
            break;
        }
        else if(!currNode->right) {
            currNode->right = new Node(data);
            break;
        }
        q.push(currNode->left);
        q.push(currNode->right);
    }
    return root;
}

void inOrder(Node *root) {
    if(root == NULL) return;
    inOrder(root->left);
    cout << root->data << " ";
    inOrder(root->right);
}

void levelOrder(Node *root) {
    if(root == NULL) {
        cout << "tree is empty!"<< endl;
        return;
    }
    queue<Node *> q;
```

```
        q.push(root);
        while(!q.empty()) {
            Node *currNode = q.front();
            q.pop();

            cout << currNode->data << " ";
            if(currNode->left) q.push(currNode->left);
            if(currNode->right) q.push(currNode->right);
        }
        cout << endl << "level Order done!\n";
    }
};

int main() {
    Node tree;
    Node *root = NULL;
    root = tree.insert(root, 5);
    root = tree.insert(root, 9);
    root = tree.insert(root, 7);
    root = tree.insert(root,4);
    root = tree.insert(root, 2);
    root = tree.insert(root, 20);

    tree.inOrder(root);
    cout << endl;
    tree.levelOrder(root);

    return 0;
}
```

```
//binaryTree.cpp

#include <bits/stdc++.h>

using namespace std;

class Node
{
public:
   int data;
   Node *left, *right;
   Node(int data)
   {
      this->data = data;
      this->left = this->right = NULL;
   }
};
Node *root;

Node *makeTree()
{
   Node *newNode;
   newNode = NULL;
   return newNode;
}

void insert(int item)   // iterative inserting
{
   if (root == NULL)
```

```
    {
        root = new Node(item);
        return;
    }
    queue<Node *> q;
    q.push(root);

    while (!q.empty())
    {
        Node *temp = q.front();
        q.pop();

        if (!(temp->left))
        {
            temp->left = new Node(item);
            return;
        }
        else if (!(temp->right))
        {
            temp->right = new Node(item);
            return;
        }
        else
        {
            q.push(temp->left);
            q.push(temp->right);
        }
    }
}
```

```
void preOrder(Node *root)
{
   if (!root)
      return;
   cout << root->data << " ";
   preOrder(root->left);
   preOrder(root->right);
}

void postOrder(Node *root)
{
   if (!root)
      return;
   preOrder(root->left);
   preOrder(root->right);
   cout << root->data << " ";
}

void inOrder(Node *root)
{
   if (!root) return;
   preOrder(root->left);
   cout << root->data << " ";
   preOrder(root->right);

}

int main()
```

```
{
    root = makeTree();

    int key[10] = {1, 3, 5, 6, 8, 2, 7, 9, 4, 10};

    for (int i = 0; i < 10; i++)
        insert(key[i]);

    cout << "Pre Order : ";
    preOrder(root);
    cout << endl;
    cout << "Post Order : ";
    postOrder(root);
    cout << "In Order : ";
    inOrder(root);

    cout << endl;

    return 0;
}
```

//**implementation.c**

```
#include <stdio.h>
#include <stdlib.h>

struct Node
{
   int data;
   struct Node *left;
   struct Node *right;
} * root;
typedef struct Node node;

// Inorder travesal
void inOrderTraversal(node *root)
{
   if (root == NULL)
      return;

   inOrderTraversal(root->left);
   printf("%d ", root->data);
   inOrderTraversal(root->right);
}

// Preorder traversal
void preOrderTraversal(node *root)
{
   if (root == NULL)
```

```
        return;

    printf("%d ", root->data);
    preOrderTraversal(root->left);
    preOrderTraversal(root->right);
}

// Postorder traversal
void postOrderTraversal(node *root)
{
    if (root == NULL)
        return;

    postOrderTraversal(root->left);
    postOrderTraversal(root->right);
    printf("%d ", root->data);
}

// returning a dynamically allocated node with a given value.
node *getNode(int item)
{

    node *newNode;
    newNode = (node *)malloc(sizeof(node));
    newNode->data = item;
    newNode->left = NULL;
    newNode->right = NULL;

    return newNode;
```

```
}

// Add an element to the left child
node *addLeftChild(node *tree, int item)
{
   tree->left = getNode(item);
   return tree->left;
}

// Add an element to the right child
node *addRightChild(node *tree, int item)
{
   tree->right = getNode(item);
   return tree->right;
}

int main()
{

   root = getNode(2);  // set the root node

   node *rootLeft = addLeftChild(root, 5);   // add root nodes left child
   node *rootRight = addRightChild(root, 8);  // add root nodes right child

   node *rootLeft_left = addLeftChild(rootLeft, 10);  // add root-left nodes left child
   node *rootLeft_right = addRightChild(rootLeft, 11);   // add root-left nodes right child

   node *rootRight_left = addLeftChild(rootRight, 3);  // add roog-right nodes left child
   node *rootRight_right = addRightChild(rootRight, 4);  // add root-right nodes right child
```

```
    printf("\nInorder Traversing : ");
    inOrderTraversal(root);  // traverse inorder

    printf("\nPreOrder Traversing : ");
    preOrderTraversal(root);  // traverse preorder

    printf("\nPostOrder Traversing : ");
    postOrderTraversal(root);  // traverse postorder

    return 0;
}
```

```
//binarytreeiterative_inorder



#include<iostream>
#include<stack>
#include<queue>

using namespace std;

class Tree {
   public:
   int data;
   Tree *left, *right;
   Tree(int data) {
      this->data = data;
      this->left = this->right = NULL;
   }
};

// Tree *root = NULL;

Tree* make_Tree(Tree *root, int data) {
   if(root == NULL) {
      root = new Tree(data);
      return root;
   }
   queue<Tree*> q_tree;

   q_tree.push(root);
```

```
    while(!q_tree.empty()) {
        Tree *temp = q_tree.front();
        q_tree.pop();

        if(!temp->left) {
            temp->left = new Tree(data);
            break;
        }
        else if(!temp->right) {
            temp->right = new Tree(data);
            break;
        }
        else {
            q_tree.push(temp->left);
            q_tree.push(temp->right);
        }
    }
    return root;
}

void inorder(Tree *root) {

    if(root == NULL) return;

    stack<Tree*> stk_tree;

    // stk_tree.push(root);
    Tree *temp = root;
```

```
    while(!stk_tree.empty() || temp != NULL) {

        while(temp!= NULL) {
            stk_tree.push(temp);
            temp = temp->left;
        }
        temp = stk_tree.top();
        cout << temp->data << " ";
        stk_tree.pop();

        temp = temp->right;
    }
    cout << endl;
}

int main() {

    Tree* root = NULL;

    int data[] = {1,2,3,4,5,6,7,8,9};
    for(int i = 0; i < 9; i++) root = make_Tree(root, data[i]);

    cout << "In Order : ";
    inorder(root);

    return 0;

}
```

//binarytree**iterative_postorder**

```
#include<iostream>
#include<stack>
#include<queue>

using namespace std;

class Tree {
   public:
   int data;
   Tree *left, *right;
   Tree(int data) {
      this->data = data;
      this->left = this->right = NULL;
   }
};

// Tree *root = NULL;

Tree* make_Tree(Tree *root, int data) {
   if(root == NULL) {
      root = new Tree(data);
      return root;
   }
   queue<Tree*> q_tree;

   q_tree.push(root);
```

```
    while(!q_tree.empty()) {
        Tree *temp = q_tree.front();
        q_tree.pop();

        if(!temp->left) {
            temp->left = new Tree(data);
            break;
        }
        else if(!temp->right) {
            temp->right = new Tree(data);
            break;
        }
        else {
            q_tree.push(temp->left);
            q_tree.push(temp->right);
        }
    }
    return root;
}

void postorder(Tree *root) {
    if(root == NULL) return;

    stack<Tree*> stk_tree, temp_stk;
    stk_tree.push(root);

    while(!stk_tree.empty()) {
        Tree *temp = stk_tree.top();
        stk_tree.pop();
```

```
        temp_stk.push(temp);

        if(temp->left) stk_tree.push(temp->left);
        if(temp->right) stk_tree.push(temp->right);
    }

    while(!temp_stk.empty()) {
        cout << temp_stk.top()->data << " ";
        temp_stk.pop();
    }
    cout << endl;
}

int main() {

    Tree* root = NULL;

    int data[] = {1,2,3,4,5,6,7,8,9};
    for(int i = 0; i < 9; i++) root = make_Tree(root, data[i]);

    cout << "Post Order : ";
    postorder(root);

    return 0;

}
```

```
//binarytreeiterative_preOrder


#include<iostream>
#include<stack>
#include<queue>

using namespace std;

class Tree {
   public:
   int data;
   Tree *left, *right;
   Tree(int data) {
      this->data = data;
      this->left = this->right = NULL;
   }
};

Tree *root = NULL;

void make_Tree(int data) {
   if(root == NULL) {
      root = new Tree(data);
      return ;
   }
   queue<Tree*> q_tree;

   q_tree.push(root);
```

```
    while(!q_tree.empty()) {
        Tree *temp = q_tree.front();
        q_tree.pop();

        if(!temp->left) {
            temp->left = new Tree(data);
            return;
        }
        else if(!temp->right) {
            temp->right = new Tree(data);
            return;
        }
        else {
            q_tree.push(temp->left);
            q_tree.push(temp->right);
        }
    }
}

void preOrder() {
    if(root == NULL) return;

    stack<Tree*> stk_tree;
    stk_tree.push(root);

    while(!stk_tree.empty()) {
        Tree *temp = stk_tree.top();
        stk_tree.pop();
```

```
        cout << temp->data << " ";

        if(temp->right) stk_tree.push(temp->right);
        if(temp->left) stk_tree.push(temp->left);
    }
}

int main() {

    int data[] = {1,2,3,4,5,6,7,8,9};
    for(int i = 0; i < 9; i++) make_Tree(data[i]);

    preOrder();

    return 0;

}
```

1. //Graph

/
adjacency_list

```
#include<stdio.h>
#include <stdlib.h>
#define N 6

struct Node {
   int dest;
   struct Node* next;
};
typedef struct Node node;

struct Graph {
   node* head[N];
};
typedef struct Graph graph;

graph* createGraph(int n) {
   graph* g = (graph*)malloc(sizeof(graph));

   for(int i = 0; i < n; i++) {
      g->head[i] = NULL;
   }

   printf("Enter number of Edges : ");
   int E;
```

```
scanf("%d", &E);

for(int i = 0; i < E; i++) {
   int src, dest;
   printf("Enter source and destination : ");
   scanf("%d %d", &src, &dest);

   node *newNode1 = (node*)malloc(sizeof(node));
   newNode1->dest = dest;
   newNode1->next = NULL;
   if(g->head[src] == NULL) g->head[src] = newNode1;
   else {
      node *temp = g->head[src];
      while(temp->next != NULL) temp = temp->next;
      temp -> next = newNode1;
   }


   node *newNode2 = (node*)malloc(sizeof(node));
   newNode2->dest = src;
   newNode2->next = NULL;
   if(g->head[dest] == NULL) g->head[dest] = newNode2;
   else {
      node *temp = g->head[dest];
      while(temp->next != NULL) temp = temp->next;
      temp -> next = newNode2;
   }

}
```

```
    return g;
}

void printGraph(graph* g, int n) {
    for(int i = 0; i < n; i++) {
        printf("%d --> ", i);
        node *temp = g->head[i];
        while(temp != NULL) {
            printf("%d ", temp->dest);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {

    int n;
    printf("Enter number of Nodes : ");
    scanf("%d",&n);

    graph *graph1 = createGraph(n);

    printGraph(graph1, n);

    return 0;

}
```

1. //Graph

/
adjacency_matrix

```
#include<iostream>

using namespace std;

// Undirected Graph

int main() {

   int vertex;
   cout << "Enter number of vertex : ";
   cin >> vertex;

   int edges;
   cout << "Enter Number of edges : ";
   cin >> edges;

   int adjacencyMatrix[vertex][vertex];
   for(int i = 0; i < vertex; i++) {
      for(int j = 0; j < vertex; j++) {
         adjacencyMatrix[i][j] = 0;
      }
   }

   // Enter edges for Undirected Graph
```

```
    cout << "Enter edges : \n";
    for(int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;
        if(u > vertex || v > vertex) {
            cout << "Invalid edge!";
            i--;
        }
        else {
        adjacencyMatrix[u][v] = 1;
        adjacencyMatrix[v][u] = 1;  // skip this line for directed graph
        }
    }

    // printing adjacency matrix...
    for(int i = 0; i < vertex; i++) {
        for(int j = 0; j < vertex; j++) {
            cout << adjacencyMatrix[i][j] << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

1. //Graph

/
bfs

```
#include<bits/stdc++.h>
using namespace std;

vector<int> bfs(vector<int>adjList[], int n) {
   vector<bool>vis(n, false);
   vector<int>bfs;
   queue<int> q;
   for(int i = 0; i < n; i++) {
      if(!vis[i]) {
         q.push(i);
         vis[i] = true;
         while(!q.empty()) {
            int node = q.front();
            q.pop();
            bfs.push_back(node);
            for(auto it : adjList[node]) {
               if(!vis[it]) {
                  q.push(it);
                  vis[it] = true;
               }
            }
         }
      }
   }
```

```
    return bfs;
}

int main() {
    cout << "Enter number of Nodes : ";
    int n;
    cin >> n;
    vector<int>adjList[n];
    cout << "Enter number of Edges : ";
    int e;
    cin >> e;
    cout << "Enter Edges : \n";
    for(int i = 0; i < e; i++) {
        int u, v;
        cin >> u >> v;
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }

    for(int i = 0; i < n; i++) {
        cout << i << " : ";
        for(auto i : adjList[i]) {
            cout << i << " ";
        }
        cout << endl;
    }
    cout << endl;
    vector<int>ans = bfs(adjList, n);
```

```
    for(auto it : ans) {
        cout << it << " ";
    }
    cout << endl;

    return 0;
}



/*

6 9
0 1
0 2
0 3
1 3
1 4
1 5
2 3
3 5
4 5

*/
```

1. Graph

/
dfs

```
#include<bits/stdc++.h>

using namespace std;

vector<int> dfs(vector<int>adjList[], int n) {
   vector<bool> visited(n, false);

   vector<int> dfs;

   stack<int> s;

   for(int i = 0; i < n; i++) {
      if(!visited[i]) {
         s.push(i);

         visited[i] = true;

         while(!s.empty()) {
            int node = s.top();
            s.pop();

            dfs.push_back(node);

            for(auto it : adjList[node]) {
```

```
                if(!visited[i]) {
                    s.push(i);
                    visited[i] = true;
                }
            }
        }
    }

    return dfs;

}

int main() {
    int n;
    cin >> n;

    vector<int>adjList[n];

    int edges;
    cin >> edges;

    for(int i = 0; i < edges; i++) {
        int u, v;
        cin >> u >> v;

        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }
```

```
    for(int i = 0; i < n; i++) {
        cout << i << " : ";

        for(auto j : adjList[i]) cout << j << " ";
        cout << endl;
    }
    cout << endl;

    vector<int>ans = dfs(adjList, n);

    cout << "DFS result : ";
    for(auto it : ans) cout << it << " ";

    cout << endl;

    return 0;
}
```

1. //Graph

/
linked_list_representation_of_graph_with_bfs_dfs

```
#include <iostream>
#include<queue>
#include<vector>
#include<stack>
using namespace std;
class Node
{
public:
   int data;
   int status;
   Node *next;
   class Adjacent *adjacent;

   Node(int data)
   {
      this->data = data;
      this->status = 0;
      this->next = NULL;
      this->adjacent = NULL;
   }
};

class Adjacent
{
```

```
public:
   class Node *node;
   Adjacent *next;
   Adjacent(Node *node)
   {
      this->node = node;
      this->next = NULL;
   }
};

typedef Node node;
typedef Adjacent adjcent;

node *start = NULL, *nodeptr;
adjcent *adjacentptr = NULL;

class Graph
{
public:
   void createNodeList(int v)
   {
      node *tail = start;
      for (int i = 0; i < v; i++)
      {
         if (i == 0)
         {
            start = new Node(i);
            tail = start;
         }
```

```
        else
        {
            tail->next = new Node(i);
            tail = tail->next;
        }
    }
}

void printNodeList()
{
    node *temp = start;

    while (temp != NULL)
    {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

node *findNodeForItem(int item)
{
    node *temp = start;
    while (temp != NULL)
    {
        if (temp->data == item)
            return temp;
        temp = temp->next;
    }
```

```
    return NULL;
}

void createGraph(int vertex)
{
    createNodeList(vertex);
    printNodeList();

    nodeptr = start;
    while (nodeptr != NULL)
    {
        adjacentptr = NULL;
        cout << "Enter connected nodes with " << nodeptr->data << " (-1) for end : ";
        while (true)
        {
            int item;
            cin >> item;
            if (item == -1)
                break;
            else if (findNodeForItem(item) != NULL)
            {
                node *temp = findNodeForItem(item);
                if (nodeptr->adjacent == NULL)
                {
                    nodeptr->adjacent = new Adjacent(temp);
                    adjacentptr = nodeptr->adjacent;
                }
                else
                {
```

```
            adjacentptr->next = new Adjacent(temp);
            adjacentptr = adjacentptr->next;
          }
        }
        else
        {
          cout << "Node not found! " << endl;
        }
      }
      nodeptr = nodeptr->next;
    }
}

void printAdjacencyList()
{
    nodeptr = start;
    while (nodeptr != NULL)
    {
      cout << nodeptr->data << " --> ";
      adjacentptr = nodeptr->adjacent;
      while (adjacentptr != NULL)
      {
        cout << adjacentptr->node->data << " ";
        adjacentptr = adjacentptr->next;
      }
      cout << endl;
      nodeptr = nodeptr->next;
    }
}
```

```
vector<int> bfs () {

   // vector<bool>vis(vertex, false);
   queue<node*>q;
   vector<int>bfs_result;
   q.push(start);
   start->status = 1;
   while(!q.empty()) {
      node *temp = q.front();
      q.pop();
      temp->status = 2;
      bfs_result.push_back(temp->data);
      adjacentptr = temp->adjacent;
      while(adjacentptr != NULL) {
         if(adjacentptr->node->status == 0) {
            q.push(adjacentptr->node);
            adjacentptr->node->status = 1;
         }
         adjacentptr = adjacentptr->next;
      }
   }
   return bfs_result;
}

vector<int> dfs () {
   vector<int>dfs_result;
   stack<node*>s;
```

```
    s.push(start);
    start->status = 1;
    while(!s.empty()) {
        node *temp = s.top();
        s.pop();
        dfs_result.push_back(temp->data);
        if(temp->status == 1) temp->status = 2;
        adjacentptr = temp->adjacent;
        while(adjacentptr != NULL) {
            if(adjacentptr->node->status == 0) {
                s.push(adjacentptr->node);
                adjacentptr->node->status = 1;
            }
            adjacentptr = adjacentptr->next;
        }
    }
    return dfs_result;
}

void reset() {
    nodeptr = start;
    while(nodeptr != NULL) {
        nodeptr->status = 0;
        nodeptr = nodeptr->next;
    }
}

};
```

```
int main()
{

    cout << "Enter number of vertexes : ";
    int vertex;
    cin >> vertex;
    Graph graph;
    graph.createGraph(vertex);
    graph.printAdjacencyList();

    graph.reset();
    vector<int> res = graph.bfs();

    cout << "BFS result : ";
    for(auto i : res) cout << i << " ";
    cout << endl;

    graph.reset();
    vector<int> dfs = graph.dfs();

    cout << "DFS result : ";
    for(auto i : dfs) cout << i << " ";
    cout << endl;

    return 0;
}

/*
```

input :

4

1 2 3 -1

0 3 -1

0 3 -1

0 1 2 -1

output :

0 --> 1 2 3

1 --> 0 3

2 --> 0 3

3 --> 0 1 2

BFS result : 0 1 2 3

DFS result : 0 3 2 1

*/

1. //Graph

```
/
path_matrix
```

```
#include<bits/stdc++.h>
using namespace std;



int main() {

    cout << "Enter number of Nodes : ";
    int m;
    cin >> m;

    int matrix[m+1][m+1];
    memset(matrix, 0, sizeof(matrix));

    cout << "Enter number of edges : ";
    int n;
    cin >> n;

    cout << "Enter edges : \n";
    for(int i = 1; i <= n; i++) {
        int u, v;
```

```
    cin >> u >>v;
    matrix[u][v] = 1;
    // matrix[v][u] = 1;
}

cout << "Adjacency matrix : \n";

for(int i = 1; i <= m; i++) {
    for(int j = 1; j <= m; j++) {
        cout << matrix[i][j] << " ";
    }
    cout << endl;
}

// adjacents of v1, v2, v3 ... vm
// for(int i = 1; i <= m; i++) {
//     cout << "Adjacent of node " << i << " : ";
//     for(int j = 1; j <= m; j++) {
//         if(matrix[i][j]) cout << j << " ";
//     }
//     cout << endl;
// }
```

```
int powMatrix[m+1][m+1][m+1];
for(int i = 1; i <= m; i++) {
    for(int j = 0; j <= m; j++) powMatrix[1][i][j] = matrix[i][j];
}

for(int i = 2; i <= m; i++) {
    for(int j = 1; j <= m; j++) {
        for(int k = 1; k <= m; k++) {
            int temp = 0;
            for(int l = 1; l <= m; l++) {
                temp += powMatrix[i-1][j][l]*matrix[l][k];
            }
            powMatrix[i][j][k] = temp;
        }
    }
}

cout << endl << endl;
for(int i = 1; i <= m; i++) {
    for(int j = 1; j <= m; j++) {
        for(int k = 1; k <= m; k++) {
            cout << powMatrix[i][j][k] << " ";
        }
```

```
        cout << endl;
    }
    cout << endl << endl;
}

int Br[m+1][m+1];
memset(Br, 0, sizeof(Br));

for(int i = 1; i <= m; i++) {
    for(int j = 1; j <= m; j++) {
        for(int k = 1; k <= m; k++) {
            Br[j][k] += powMatrix[i][j][k];
            // Br[i][j] += powMatrix[k][i][j];
        }
    }
}

int cnt = 0;

cout << "Path Matrix : \n";
for(int i = 1; i <= m; i++) {
    for(int j = 1; j <= m; j++) {
        cout << Br[i][j] << " ";
```

```
            if(Br[i][j] == 0) cnt++;
        }
        cout << endl;
    }
    cout << endl;
    if(cnt == 0) cout << "Strongly Connected!" << endl;
    else cout << "Not Stronlgy Connected! " << endl;

    return 0;
}
```

1. //Graph

```
/
path_matrix_warshall
```

```
/*

1 0 1 0

0 1 0 1

0 0 1 1

1 1 0 0

*/

#include<iostream>

using namespace std;

int matrix[20][20];

int main() {

    cout << "Enter number of nodes : ";

    int m;

    cin >> m;
```

```
int matrix[m][m];


cout << "Enter matrix : " << endl;
for(int i = 0; i < m; i++) {
    for(int j = 0; j < m; j++) {
        cin >> matrix[i][j];
    }
}


cout << "Matrix (p[0]) is : \n";
for(int i = 0; i < m; i++) {
    for(int j = 0; j < m; j++) {
        cout << matrix[i][j] << " ";
    }
    cout << endl;
}


for(int k = 0; k < m; k++) {
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < m; j++) {
            matrix[i][j] = matrix[i][j]?matrix[i][j]:(matrix[i][k]&&matrix[k][j]);
        }
    }
```

```
    }


    cout << "Path matrix (p[4]) is : \n";
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < m; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }


}
```

1. //Graph

/
power_matrix

```
#include<bits/stdc++.h>
using namespace std;


int main() {

    cout << "Enter number of Nodes : ";
    int m;
    cin >> m;

    int matrix[m+1][m+1];
    memset(matrix, 0, sizeof(matrix));

    cout << "Enter number of edges : ";
    int n;
    cin >> n;

    cout << "Enter edges : \n";
    for(int i = 1; i <= n; i++) {
```

```
    int u, v;
    cin >> u >>v;
    matrix[u][v] = 1;
    // matrix[v][u] = 1;
}

cout << "Adjacency matrix : \n";

for(int i = 1; i <= m; i++) {
    for(int j = 1; j <= m; j++) {
        cout << matrix[i][j] << " ";
    }
    cout << endl;
}

// adjacents of v1, v2, v3 ... vm
for(int i = 1; i <= m; i++) {
    cout << "Adjacent of node " << i << " : ";
    for(int j = 1; j <= m; j++) {
        if(matrix[i][j]) cout << j << " ";
    }
    cout << endl;
}
```

```
int powMatrix[m+1][m+1][m+1];
for(int i = 1; i <= m; i++) {
    for(int j = 0; j <= m; j++) powMatrix[1][i][j] = matrix[i][j];
}

for(int i = 2; i <= m; i++) {
    for(int j = 1; j <= m; j++) {
        for(int k = 1; k <= m; k++) {
            int temp = 0;
            for(int l = 1; l <= m; l++) {
                temp += powMatrix[i-1][j][l]*matrix[l][k];
            }
            powMatrix[i][j][k] = temp;
        }
    }
}

for(int i = 2; i <= m; i++) {
    cout << "paths of length " << i << " : \n";
    int cnt = 0;
    for(int j = 1; j <= m; j++) {
        for(int k = 1; k <= m; k++) {
```

```
            if(powMatrix[i][j][k]) {
                cout << "From node " << j << " to " << k << endl;
                cnt++;
            }
        }
    }
    cout << "\ntotal number of path " << i << " is : " << cnt << endl;
}

cout << endl << endl;
for(int i = 1; i <= m; i++) {
    for(int j = 1; j <= m; j++) {
        for(int k = 1; k <= m; k++) {
            cout << powMatrix[i][j][k] << " ";
        }
        cout << endl;
    }
    cout << endl << endl;
}

return 0;
}
```

1. //Graph

```
/
shortest_path_matrix
```

```
#include<iostream>

using namespace std;

#define MAX 1000000000

int main() {

    cout << "Enter number of vertices : ";
    int v;
    cin >> v;

    int matrix[v][v];
    cout << "Enter the matrix : \n";
    int shortest_path_matrix[v][v];
    for(int i = 0; i < v; i++) {
        for(int j = 0; j < v; j++) {
            cin >> matrix[i][j];
            if(matrix[i][j] == 0) shortest_path_matrix[i][j] = MAX;
```

```
            else shortest_path_matrix[i][j] = matrix[i][j];
        }
    }

    // calculate shortest path matrix :
    for(int k = 0; k < v; k++) {

        cout << endl << endl;
        for(int i = 0; i < v; i++) {
            for(int j = 0; j < v; j++) {
                cout << shortest_path_matrix[i][j] << " ";
            }
        cout << endl;
        }

        for(int i = 0; i < v; i++) {
            for(int j = 0; j < v; j++) {
                shortest_path_matrix[i][j] = min(shortest_path_matrix[i][j],
(shortest_path_matrix[i][k]+shortest_path_matrix[k][j]));
            }
        }
    }

    cout << "\nShortest path matrix : \n";
```

```
    for(int i = 0; i < v; i++) {
        for(int j = 0; j < v; j++) {
            cout << shortest_path_matrix[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```