Addition of polynomials using single linked lists

*import* java.util.\*;  
*class* polyaddition{  
 node head;  
 *static class* node{  
 *int* coefficient;  
 *int* power;  
 node next;  
  
 *public* node(*int* coefficient, *int* power) {  
 *this*.coefficient = coefficient;  
 *this*.power = power;  
 }  
 }  
 *void* display(){  
 node temp = head;  
 *while* (temp!=*null*){  
 System.out.print(temp.coefficient+" x^ "+temp.power+" ");  
 *if* (temp.next != *null*)  
 System.out.print(" + ");  
 temp = temp.next;  
 }  
 System.out.println();  
 }  
 polyaddition addition(polyaddition list1,polyaddition list2){  
 polyaddition result = *new* polyaddition();  
 node temp1 = list1.head;  
 node temp2 = list2.head;  
 *while* (temp1!=*null* && temp2!=*null*){  
 *if*(temp1.power> temp2.power){  
 result.endinsert(temp1.coefficient, temp1.power);  
 temp1 = temp1.next;  
 }  
 *else if*(temp1.power< temp2.power){  
 result.endinsert(temp2.coefficient, temp2.power);  
 temp2 = temp2.next;  
 }  
 *else*{  
 *int* resco = temp1.coefficient+ temp2.coefficient;  
 result.endinsert(resco, temp2.power);  
 temp1 = temp1.next;  
 temp2 = temp2.next;  
 }  
 }  
 *while* (temp1!=*null*){  
 result.endinsert(temp1.coefficient, temp1.power);  
 temp1 = temp1.next;  
 }  
 *while* (temp2!=*null*){  
 result.endinsert(temp2.coefficient, temp2.power);  
 temp2 = temp2.next;  
 }  
 *return* result;  
 }  
 *void* endinsert(*int* coefficient,*int* power){  
 node inserted = *new* node(coefficient,power);  
 node temp = head;  
 *if* (head==*null*){  
 head=inserted;  
 }  
 *else* {  
 *while* (temp.next != *null*) {  
 temp = temp.next;  
 }  
 temp.next = inserted;  
 inserted.next = *null*;  
 }  
 }  
}  
*public class* question8 {  
 *public static void* main(String[] args) {  
 Scanner sc = *new* Scanner(System.in);  
 System.out.print("Enter the number of terms in 1st polynomial : " );  
 *int* size1 = sc.nextInt();  
 System.out.print("Enter the number of terms in 2nd polynomial : " );  
 *int* size2 = sc.nextInt();  
 polyaddition list1 = *new* polyaddition();  
 polyaddition list2 = *new* polyaddition();  
 System.out.println("Enter the terms of 1st polynomial ");  
 *for*(*int* i=0;i<size1;i++){  
 System.out.print("Enter coefficient and power : ");  
 *int* coef =sc.nextInt();  
 *int* power = sc.nextInt();  
 list1.endinsert(coef,power);  
 }  
 System.out.println("Enter the terms of 2nd polynomial ");  
 *for*(*int* i=0;i<size2;i++){  
 System.out.print("Enter coefficient and power : ");  
 *int* coef =sc.nextInt();  
 *int* power = sc.nextInt();  
 list2.endinsert(coef,power);  
 }  
 System.out.print("1st polynomial : ");  
 list1.display();  
 System.out.print("2nd polynomial : ");  
 list2.display();  
 polyaddition result = list1.addition(list1,list2);  
 System.out.println("Answer is ");  
 result.display();  
  
  
  
  
 }  
}

Doubly Linked List all functions

#include<stdio.h>

#include<stdlib.h>

struct node{

    struct node\*prev;

    int data;

    struct node\*next;

};

struct node\*DeleteAtEnd(struct node\*head){

    struct node\*p=head;

    if(head==NULL){

        printf("Can't delete because list is empty\n");

        return NULL;

    }

    else if(head->next==NULL){

        free(head);

        printf("Deleted only item in list the list is empty now\n");

        return NULL;

    }

    else{

        while(p->next!=NULL){

            p=p->next;

        }

    struct node\*q=p;

    p=p->prev;

    if(p!=NULL)

        p->next=NULL;

    free(q);

    return head;

    }

}

struct node\*DeleteAtFront(struct node\*head){

    if(head==NULL){

        printf("Can't delete because the list is empty\n");

        return NULL;

    }

    else if(head->next==NULL && head->prev==NULL){

        free(head);

        printf("List is empty now\n");

        return NULL;

    }

    else{

    struct node\*p=head;

    head=head->next;

    head->prev=NULL;

    free(p);

    return head;

    }

}

struct node\*InsertAtEnd(struct node\*head,int data){

    struct node\*ptr=(struct node\*)malloc(sizeof(struct node));

    ptr->data=data;

    if(head==NULL){

        ptr->prev=NULL;

        ptr->next=NULL;

        head=ptr;

        return head;

    }

    else{

        struct node\*p=head;

        while(p->next!=NULL){

        p=p->next;

        }

    ptr->prev=p;

    p->next=ptr;

    ptr->next=NULL;

    return head;

    }

}

struct node\*InsertAtFront(struct node\*head,int data){

    struct node\*ptr=(struct node\*)malloc(sizeof(struct node));

    ptr->data=data;

    if(head==NULL){

        ptr->prev=NULL;

        ptr->next=NULL;

        head=ptr;

        return head;

    }

    else{

    ptr->next=head;

    head->prev=ptr;

    head=ptr;

    return head;

    }

}

struct node\*InsertBeforePosition(struct node\*head,int checkData,int insData){

    struct node\*ptr=(struct node\*)malloc(sizeof(struct node));

    struct node\*p=head;

    ptr->data=insData;

    if(head==NULL){

        printf("List is empty,So can't insert anything\n");

        return NULL;

    }

    while(p->data!=checkData && p->next!=NULL){

        p=p->next;

    }

    struct node\*q=p->prev;

    if(p->next==NULL && p->data!=checkData){

        printf("No such element exists!\n");

        return head;

    }

    else if(p->prev==NULL){

        ptr->prev=NULL;

        ptr->next=head;

        head->prev=ptr;

        head=ptr;

        return head;

    }

    else{

        ptr->prev=p->prev;

        q->next=ptr;

        ptr->next=p;

        p->prev=ptr;

        return head;

    }

}

void Display(struct node\*head){

    struct node\*temp=head;

    if(head==NULL)

        printf("List is empty!\n");

    else{

    printf("Traversal of entire Linked List\n");

    while(temp!=NULL){

        printf("%d ",temp->data);

        temp=temp->next;

        }

    }

}

struct node\*DeleteAtPosition(struct node\*head,int checkData){

    struct node\*p=head;

    if(head==NULL){

        printf("No such element exists!\n");

        return NULL;

    }

    while(p->data!=checkData && p->next!=NULL){

        p=p->next;

    }

    if((p->next==NULL && p->data!=checkData)){

        printf("No such element exists!\n");

        return head;

    }

    else if(p->next==NULL && p->prev==NULL){

        free(p);

        printf("List is empty now\n");

        return NULL;

    }

    else if(p->next==NULL){

        head=DeleteAtEnd(head);

        return head;

    }

    else if(p->prev==NULL){

        head=DeleteAtFront(head);

        return head;

    }

    else if(p!=NULL){

        struct node\*r=p->prev;

        r->next=p->next;

        p->next->prev=p->prev;

        free(p);

        return head;

    }

}

int main(){

    struct node\*head=NULL;

    int flag=0;

    do {

    int ch;

    printf("\nEnter your choice:\n");

    printf("1)Insert At Front\n2)Insert At End\n3)Insert with a given value\n");

    printf("4)Delete At Front\n5)Delete At End\n6)Delete with a given element\n7)Exit\n");

    scanf("%d",&ch);

    switch(ch){

        case 1:

        {

            int ele;

            printf("Enter the element you want to insert:\n");

            scanf("%d",&ele);

            head=InsertAtFront(head,ele);

            printf("Current Status:\n");

            Display(head);

            break;

        }

        case 2:

        {

            int ele;

            printf("Enter the element you want to insert:\n");

            scanf("%d",&ele);

            head=InsertAtEnd(head,ele);

            printf("Current Status:\n");

            Display(head);

            break;

        }

        case 3:

        {

            int ele,check;

            printf("Enter the element you want to insert:\n");

            scanf("%d",&ele);

            printf("Enter the element before which you want to insert:\n");

            scanf("%d",&check);

            head=InsertBeforePosition(head,check,ele);

            printf("Current Status:\n");

            Display(head);

            break;

        }

        case 4:

        {

            head=DeleteAtFront(head);

            printf("Current Status:\n");

            Display(head);

            break;

        }

        case 5:

        {

            head=DeleteAtEnd(head);

            printf("Current Status:\n");

            Display(head);

            break;

        }

        case 6:

        {

            int check;

            printf("Enter the element which you want to delete:\n");

            scanf("%d",&check);

            head=DeleteAtPosition(head,check);

            printf("Current Status:\n");

            Display(head);

            break;

        }

        case 7:

        {

            flag=1;

            printf("Program finished\n");

            break;

        }

        default:

        {

            printf("Invalid choice!\n");

            break;

        }

    }

    }while(flag!=1);

    return 0;

}

Circular single linked list

struct node {

    int info;

    struct node\* next;

};

// Pointer to last node in the list

struct node\* last = NULL;

// Function to insert a node in the

// starting of the list

void insertAtFront(int data)

{

    // Initialize a new node

    struct node\* temp;

    temp = (struct node\*)malloc(sizeof(struct node));

    // If the new node is the only

    // node in the list

    if (last == NULL) {

        temp->info = data;

        temp->next = temp;

        last = temp;

    }

    // Else last node contains the

    // reference of the new node and

    // new node contains the reference

    // of the previous first node

    else {

        temp->info = data;

        temp->next = last->next;

        // last node now has reference

        // of the new node temp

        last->next = temp;

    }

}

void addatlast(int data)

{

    // Initialize a new node

    struct node\* temp;

    temp = (struct node\*)malloc(sizeof(struct node));

    // If the new node is the

    // only node in the list

    if (last == NULL) {

        temp->info = data;

        temp->next = temp;

        last = temp;

    }

    // Else the new node will be the

    // last node and will contain

    // the reference of head node

    else {

        temp->info = data;

        temp->next = last->next;

        last->next = temp;

        last = temp;

    }

}

void addatlast()

{

    // Stores number to be inserted

    int data;

    // Initialize a new node

    struct node\* temp;

    temp = (struct node\*)malloc(sizeof(struct node));

    // Input data

    printf("\nEnter data to be inserted : \n");

    scanf("%d", &data);

    // If the new node is the

    // only node in the list

    if (last == NULL) {

        temp->info = data;

        temp->next = temp;

        last = temp;

    }

    // Else the new node will be the

    // last node and will contain

    // the reference of head node

    else {

        temp->info = data;

        temp->next = last->next;

        last->next = temp;

        last = temp;

    }

}

// Function to insert after any

// specified element

void insertafter()

{

    // Stores data and element after

    // which new node is to be inserted

    int data, value;

    // Initialize a new node

    struct node \*temp, \*n;

    // Input data

    printf("\nEnter number after which"

           " you want to enter number: \n");

    scanf("%d", &value);

    temp = last->next;

    do {

        // Element after which node is

        // to be inserted is found

        if (temp->info == value) {

            n = (struct node\*)malloc(sizeof(struct node));

            // Input Data

            printf("\nEnter data to be"

                   " inserted : \n");

            scanf("%d", &data);

            n->info = data;

            n->next = temp->next;

            temp->next = n;

            // If temp is the last node

            // so now n will become the

            // last node

            if (temp == last)

                last = n;

            break;

        }

        else

            temp = temp->next;

    } while (temp != last->next);

}

void addatlast(int data)

{

    // Initialize a new node

    struct node\* temp;

    temp = (struct node\*)malloc(sizeof(struct node));

    // If the new node is the only

    // node in the list

    if (last == NULL) {

        temp->info = data;

        temp->next = temp;

        last = temp;

    }

    // Else the new node will be the

    // last node and will contain

    // the reference of head node

    else {

        temp->info = data;

        temp->next = last->next;

        last->next = temp;

        last = temp;

    }

}

// Function to delete the first

// element of the list

void deletefirst()

{

    struct node\* temp;

    // If list is empty

    if (last == NULL)

        printf("\nList is empty.\n");

    // Else last node now contains

    // reference of the second node

    // in the list because the

    // list is circular

    else {

        temp = last->next;

        last->next = temp->next;

        free(temp);

    }

}

void addatlast(int data)

{

    // Initialize a new node

    struct node\* temp;

    temp = (struct node\*)malloc(sizeof(struct node));

    // If the new node is the only

    // node in the list

    if (last == NULL) {

        temp->info = data;

        temp->next = temp;

        last = temp;

    }

    // Else the new node will be

    // last node and will contain

    // the reference of head node

    else {

        temp->info = data;

        temp->next = last->next;

        last->next = temp;

        last = temp;

    }

}

// Function to delete the last node

// in the list

void deletelast()

{

    struct node\* temp;

    // If list is empty

    if (last == NULL)

        printf("\nList is empty.\n");

    temp = last->next;

    // Traverse the list till

    // the second last node

    while (temp->next != last)

        temp = temp->next;

    // Second last node now contains

    // the reference of the first

    // node in the list

    temp->next = last->next;

    last = temp;

}

void deleteAtIndex()

{

    // Stores the index at which

    // the element is to be deleted

    int pos, i = 1;

    struct node \*temp, \*position;

    temp = last->next;

    // If list is empty

    if (last == NULL)

        printf("\nList is empty.\n");

    // Else

    else {

        // Input Data

        printf("\nEnter index : ");

        scanf("%d", &pos);

        // Traverse till the node to

        // be deleted is reached

        while (i <= pos - 1) {

            temp = temp->next;

            i++;

        }

        // After the loop ends, temp

        // points at a node just before

        // the node to be deleted

        // Reassigning links

        position = temp->next;

        temp->next = position->next;

        free(position);

    }

}

Alternating split of two linked lists

/\*Program to alternatively split a linked list into two halves \*/

#include<stdio.h>

#include<stdlib.h>

#include<assert.h>

/\* Link list node \*/

struct Node

{

    int data;

    struct Node\* next;

};

/\* pull off the front node of the source and put it in dest \*/

void MoveNode(struct Node\*\* destRef, struct Node\*\* sourceRef) ;

/\* Given the source list, split its nodes into two shorter lists.

If we number the elements 0, 1, 2, ... then all the even elements

should go in the first list, and all the odd elements in the second.

The elements in the new lists may be in any order. \*/

void AlternatingSplit(struct Node\* source, struct Node\*\* aRef,

                            struct Node\*\* bRef)

{

/\* split the nodes of source to these 'a' and 'b' lists \*/

struct Node\* a = NULL;

struct Node\* b = NULL;

struct Node\* current = source;

while (current != NULL)

{

    MoveNode(&a, ¤t); /\* Move a node to list 'a' \*/

    if (current != NULL)

    {

    MoveNode(&b, ¤t); /\* Move a node to list 'b' \*/

    }

}

\*aRef = a;

\*bRef = b;

}

/\* Take the node from the front of the source, and move it to the front of the dest.

It is an error to call this with the source list empty.

Before calling MoveNode():

source == {1, 2, 3}

dest == {1, 2, 3}

After calling MoveNode():

source == {2, 3}

dest == {1, 1, 2, 3}

\*/

void MoveNode(struct Node\*\* destRef, struct Node\*\* sourceRef)

{

/\* the front source node \*/

struct Node\* newNode = \*sourceRef;

assert(newNode != NULL);

/\* Advance the source pointer \*/

\*sourceRef = newNode->next;

/\* Link the old dest off the new node \*/

newNode->next = \*destRef;

/\* Move dest to point to the new node \*/

\*destRef = newNode;

}

/\* UTILITY FUNCTIONS \*/

/\* Function to insert a node at the beginning of the linked list \*/

void push(struct node\*\* head\_ref, int new\_data)

{

/\* allocate node \*/

struct Node\* new\_node =

            (struct Node\*) malloc(sizeof(struct Node));

/\* put in the data \*/

new\_node->data = new\_data;

/\* link the old list off the new node \*/

new\_node->next = (\*head\_ref);

/\* move the head to point to the new node \*/

(\*head\_ref) = new\_node;

}

/\* Function to print nodes in a given linked list \*/

void printList(struct Node \*node)

{

while(node!=NULL)

{

printf("%d ", node->data);

node = node->next;

}

}

/\* Driver program to test above functions\*/

int main()

{

/\* Start with the empty list \*/

struct Node\* head = NULL;

struct Node\* a = NULL;

struct Node\* b = NULL;

/\* Let us create a sorted linked list to test the functions

Created linked list will be 0->1->2->3->4->5 \*/

push(&head, 5);

push(&head, 4);

push(&head, 3);

push(&head, 2);

push(&head, 1);

push(&head, 0);

printf("\n Original linked List: ");

printList(head);

/\* Remove duplicates from linked list \*/

AlternatingSplit(head, &a, &b);

printf("\n Resultant Linked List 'a' ");

printList(a);

printf("\n Resultant Linked List 'b' ");

printList(b);

getchar();

return 0;

}

Method-2

void AlternatingSplit(struct Node\* source, struct Node\*\* aRef,

                            struct Node\*\* bRef)

{

  struct Node aDummy;

  struct Node\* aTail = &aDummy; /\* points to the last node in 'a' \*/

  struct Node bDummy;

  struct Node\* bTail = &bDummy; /\* points to the last node in 'b' \*/

  struct Node\* current = source;

  aDummy.next = NULL;

  bDummy.next = NULL;

  while (current != NULL)

  {

    MoveNode(&(aTail->next), ¤t); /\* add at 'a' tail \*/

    aTail = aTail->next; /\* advance the 'a' tail \*/

    if (current != NULL)

    {

      MoveNode(&(bTail->next), ¤t);

      bTail = bTail->next;

    }

  }

  \*aRef = aDummy.next;

  \*bRef = bDummy.next;

}

**Problem statement:** Given a [linked list](https://en.wikipedia.org/wiki/Linked_list) split it into two halves, front and back. If the linked list has even number of elements then resultant linked lists will be equal. If the input linked list has odd number of elements then front linked list will has one element more than back linked list.

/\*C code to Split nodes of a linked list into the

front and back halves\*/

#include <stdio.h>

#include <stdlib.h>

typedef struct node

{

    int data;

    struct node \*next;

} NODE;

NODE \*newNode(int key)

{

    NODE \*temp = (NODE \*)malloc(sizeof(NODE));

    temp->data = key;

    temp->next = NULL;

    return temp;

}

void splitInHalf(NODE \*head, NODE \*\*front, NODE \*\*end)

{

    if (head == NULL)

    {

        \*front = head;

        \*end = NULL;

        return;

    }

    NODE \*slow = head, \*fast = head;

    if (head != NULL)

    {

        while (fast && fast->next)

        {

            fast = fast->next->next;

            slow = slow->next;

        }

    }

    \*front = head;

    \*end = slow->next;

    slow->next = NULL;

}

void printList(NODE \*temp)

{

    if (temp == NULL)

    {

        printf("List is empty!\n");

        return;

    }

    while (temp != NULL)

    {

        printf("%d--> ", temp->data);

        temp = temp->next;

    }

    printf("NULL\n");

}

int main()

{

    NODE \*head = NULL;

    NODE \*front = NULL;

    NODE \*end = NULL;

    head = newNode(1);

    head->next = newNode(2);

    head->next->next = newNode(3);

    head->next->next->next = newNode(4);

    head->next->next->next->next = newNode(5);

    head->next->next->next->next->next = NULL;

    printf("Linked List is\t");

    printList(head);

    splitInHalf(head, &front, &end);

    printf("\nFront half linked list is\t");

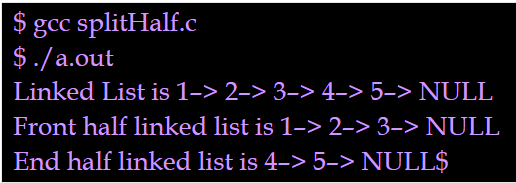
    printList(front);

    printf("\nEnd half linked list is\t\t");

    printList(end);

    return 0;

}



Priority queues

import java.util.\*;

class priorityQueues{

    int arr[];

    int priority[];

    int size;

    int front;

    int rear;

    priorityQueues(int size){

        this.size = size;

        arr = new int[size];

        priority = new int[size];

        front = 0;

        rear = -1;

    }

    boolean isFull(){

        if(rear == size - 1){

            return true;

        }

        return false;

    }

    boolean isEmpty(){

        if(front > rear){

            return true;

        }

        return false;

    }

    void enqueue(int value, int key){

        if(!isFull()){

            rear++;

            arr[rear] = value;

            priority[rear] = key;

            insertionSort();

            System.out.println(value + " has been enqueued.");

        }

        else{

            System.out.println("Queue is Full");

        }

    }

    void dequeue(){

        if(!isEmpty()){

            front++;

            System.out.println("\nFront element has been dequeued.");

        }

        else{

            System.out.println("\nQueue is Empty");

        }

    }

    int queueFront() {

        return (!isEmpty() ? arr[front] : 0);

    }

    int queueRear() {

        return (!isEmpty() ? arr[rear] : -1);

    }

    void display(){

        if(!isEmpty()){

            for(int i = front; i <= rear; i++){

                System.out.print(arr[i] + "(" + priority[i] + ")" + " ");

            }

            System.out.println();

        }

        else {

            System.out.println("\nQueue is Empty.");

        }

    }

    void insertionSort(){

        int i = rear;

        int j = i - 1;

        while(i > front && priority[i] > priority[j]){

            int temp = arr[i];

            arr[i] = arr[j];

            arr[j] = temp;

            temp = priority[i];

            priority[i] = priority[j];

            priority[j] = temp;

            i--;

            j--;

        }

    }

}

public class question6 {

    public static void main(String[] args) {

        Scanner sc=new Scanner(System.in);

        System.out.println("Enter the total number of elements: ");

        int n=sc.nextInt();

        priorityQueues priorityQueue = new priorityQueues(n);

        int el, k;

        for(int i=0;i<n;i++){

            System.out.println("\nEnter the element: ");

            el=sc.nextInt();

            System.out.println("Enter its key(priority): ");

            k=sc.nextInt();

            priorityQueue.enqueue(el, k);

            System.out.println("\nQueue so far: ");

            priorityQueue.display();

        }

        while(true){

            System.out.println("\nEnter 0 to dequeue, 1 to exit: ");

            // highest priority item dequeued first

            int ch=sc.nextInt();

            if(ch==0){

                System.out.println();

                priorityQueue.dequeue();

                System.out.println("Updated Queue: ");

                priorityQueue.display();

            }

            else if(ch==1){

                System.out.println("\nProgram Terminated !");

                break;

            }

        }

    }

}

DFS AND BFS TRAVERSALS

import java.util.\*;

class Graph

{

public

    static int time = 1;

    int ver;

    static boolean visitedbfs[];

    static boolean visiteddfs[];

    LinkedList<Integer> adj[];

    Graph(int ver)

    {

        this.ver = ver;

        adj = new LinkedList[ver];

        visitedbfs = new boolean[ver];

        visiteddfs = new boolean[ver];

        for (int i = 0; i < ver; i++)

        {

            adj[i] = new LinkedList<Integer>();

        }

    }

    void Add(int v, int w)

    {

        adj[v].add(w);

        adj[w].add(v);

    }

    void DFS(int ver, int start[], int end[])

    {

        start[ver] = time;

        time++;

        visiteddfs[ver] = true;

        System.out.print(ver + " ");

        Iterator<Integer> it = adj[ver].listIterator();

        while (it.hasNext())

        {

            int vis = it.next();

            if (!visiteddfs[vis])

            {

                DFS(vis, start, end);

            }

        }

        end[ver] = time;

        time++;

    }

    void print()

    {

        System.out.println("Adjacency List representation of Graph:");

        for (int i = 1; i < ver; i++)

        {

            System.out.print(i + ":");

            for (int x : adj[i])

            {

                System.out.print(x + " ");

            }

            System.out.println();

        }

    }

    void BFS(int src, int level[])

    {

        LinkedList<Integer> queue = new LinkedList();

        visitedbfs[src] = true;

        level[src] = 0;

        queue.add(src);

        while (queue.size() != 0)

        {

            int new\_src = queue.poll();

            System.out.print(new\_src + " ");

            Iterator<Integer> It = adj[src].listIterator();

            while (It.hasNext())

            {

                int neigh = It.next();

                if (!visitedbfs[neigh])

                {

                    visitedbfs[neigh] = true;

                    queue.add(neigh);

                    level[neigh] = level[src] + 1;

                }

            }

        }

    }

} public class DSA8

{

public

    static void main(String[] args)

    {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number of vertices:");

        int ver = sc.nextInt();

        Graph g = new Graph(ver + 1);

        System.out.println("Enter the number of edges:");

        int edge = sc.nextInt();

        int start[] = new int[ver + 1];

        int end[] = new int[ver + 1];

        int level[] = new int[ver];

        for (int i = 0; i < edge; i++)

        {

            System.out.println("Enter the first vertex");

            int src = sc.nextInt();

            System.out.println("Enter the second vertex");

            int des = sc.nextInt();

            g.Add(src, des);

        }

        System.out.println("Representation of graph");

        g.print();

        System.out.println("Enter the vertex from where you want to start traversal:");

        ;

        int startver = sc.nextInt();

        System.out.println();

        System.out.println("DFS TRAVERSAL");

        g.DFS(startver, start, end);

        System.out.println();

        for (int i = 1; i <= ver; i++)

            System.out.println("Vertex " + i + " START POINT " + start[i] + " END POINT " + end[i]);

        System.out.println("BFS TRAVERSAL");

        g.BFS(startver, level);

        System.out.println();

        System.out.println("Nodes" + " " + "Level");

        for (int i = 0; i < ver; i++)

            System.out.println(" " + i + " --> " + level[i]);

    }

}