Week 2
Write a program to implement singly linked list? Write a program to implement different operations like adding a node at beginning, end, center, after a certain element, after a certain count of nodes in a linked list,deleting a node at beginning, end, center, after a certain element, after a certain count of nodes in a linked list. Write a program in C++ to reverse a linked list by changing the link in the nodes? Write a program to add two polynomials represented as linked list?

A program to implement singly linked list

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

class Node
{
public:
int info;
Node* next;
};
class List:public Node
{

Node *first,*last;
public:
List()
{
first=NULL;
last=NULL;
}
void create();
//void insert();
void display();
};
void List::create()
{
Node *temp;
temp=new Node;
int n;
cout<<"\nEnter an Element:";
cin>>n;
temp->info=n;
temp->next=NULL;
if(first==NULL)
{
first=temp;
last=first;
}

else
{
last->next=temp;
```

```
last=temp;
}
}

void List::display()
{
Node *temp=first;
if(temp==NULL)
{
cout<<"\nList is Empty";
}
while(temp!=NULL)
{
cout<<temp->info;
cout<<"-->";
temp=temp->next;
}
cout<<"NULL";
}
int main()
{
List l;
int ch;
while(1)
{
cout<<"\n**** MENU ****";
cout<<"\n1:CREATE\n2:DISPLAY\n3:EXIT\n";
cout<<"\nEnter Your Choice:";
cin>>ch;
switch(ch)
{
case 1:
l.create();
break;
case 2:
l.display();
break;
case 3:
return 0;
default:
cout<<"invalid choice";
}
}
return 0;
}
```

A program to implement different operations like adding a node at beginning, end, center, after a certain element, after a certain count of nodes in a linked list.

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

class Node
{
```

```
public:
int info;
Node* next;
};
class List:public Node
{

Node *first,*last;
public:
List()
{
first=NULL;
last=NULL;
}
void create();
void insert();
void delet();
void display();
void search();
};
void List::create()
{
Node *temp;
temp=new Node;
int n;
cout<<"\nEnter an Element:";
cin>>n;
temp->info=n;
temp->next=NULL;
if(first==NULL)
{
first=temp;
last=first;
}

else
{
last->next=temp;
last=temp;
}
}
void List::insert()
{
Node *prev,*cur;
prev=NULL;
cur=first;
int count=1,pos,ch,n;
Node *temp=new Node;
cout<<"\nEnter an Element:";
cin>>n;
temp->info=n;
temp->next=NULL;
cout<<"\nINSERT AS\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST NODES";
cout<<"\nEnter Your Choice:";
cin>>ch;
switch(ch)
{
```

```
case 1:
temp->next=first;
first=temp;
break;
case 2:
last->next=temp;
last=temp;
break;
case 3:
cout<<"\nEnter the Position to Insert:";
cin>>pos;
while(count!=pos)
{
prev=cur;
cur=cur->next;
count++;
}
if(count==pos)
{
prev->next=temp;
temp->next=cur;
}
else
cout<<"\nNot Able to Insert";
break;

}
}
void List::delet()
{
Node *prev=NULL,*cur=first;
int count=1,pos,ch;
cout<<"\nDELETE\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST NODES";
cout<<"\nEnter Your Choice:";
cin>>ch;
switch(ch)
{
case 1:
if(first!=NULL)
{
cout<<"\nDeleted Element is "<<first->info;
first=first->next;
}
else
cout<<"\nNot Able to Delete";
break;
case 2:
while(cur!=last)
{
prev=cur;
cur=cur->next;
}
if(cur==last)
{
cout<<"\nDeleted Element is: "<<cur->info;
prev->next=NULL;
last=prev;
```

```
}
else
cout<<"\nNot Able to Delete";
break;
case 3:
cout<<"\nEnter the Position of Deletion:";
cin>>pos;
while(count!=pos)
{
prev=cur;
cur=cur->next;
count++;
}
if(count==pos)
{
cout<<"\nDeleted Element is: "<<cur->info;
prev->next=cur->next;
}
else
cout<<"\nNot Able to Delete";
break;
}
}
void List::display()
{
Node *temp=first;
if(temp==NULL)
{
cout<<"\nList is Empty";
}
while(temp!=NULL)
{
cout<<temp->info;
cout<<"-->";
temp=temp->next;
}
cout<<"NULL";
}
void List::search()
{
int value,pos=0;
int flag=0;
if(first==NULL)
{
cout<<"List is Empty";
return;
}
cout<<"Enter the Value to be Searched:";
cin>>value;
Node *temp;
temp=first;
while(temp!=NULL)
{
pos++;
if(temp->info==value)
{
flag=1;
```

```
cout<<"Element"<<value<<"is Found at "<<pos<<" Position";
return;
}
temp=temp->next;
}
if(!flag)
{
cout<<"Element "<<value<<" not Found in the List";
}
}
int main()
{
List l;
int ch;
while(1)
{
cout<<"\n**** MENU ****";
cout<<"\n1:CREATE\n2:INSERT\n3:DELETE\n4:SEARCH\n5:DISPLAY\n6:EXIT\n";
cout<<"\nEnter Your Choice:";
cin>>ch;
switch(ch)
{
case 1:
l.create();
break;
case 2:
l.insert();
break;
case 3:
l.delet();
break;
case 4:
l.search();
break;
case 5:
l.display();
break;
case 6:
return 0;
}
}
return 0;
}
```

A program to add two polynomials represented as linked list

```
#include <iostream.h>
#include<conio.h>
#include<stdio.h>
class poly
{
private :
struct polynode
{
float coeff ;
```

```
int exp ;
polynode *link ;
} *p ;
public :
poly( ) ;
void poly_append ( float c, int e ) ;
void display_poly( ) ;
void poly_add( poly &l1, poly &l2 ) ;
~poly( ) ;
} ;
poly :: poly( )
{
p = NULL ;
}
void poly :: poly_append ( float c, int e )
{
polynode *temp = p ;
if ( temp == NULL )
{
temp = new polynode ;
p = temp ;
}
else
{
while ( temp -> link != NULL )
temp = temp -> link ;
temp -> link = new polynode ;
temp = temp -> link ;
}
temp -> coeff = c ;
temp -> exp = e ;
temp -> link = NULL ;
}
void poly :: display_poly( )
{
polynode *temp = p ;
int f = 0 ;

cout << endl ;
while ( temp != NULL )
{
if ( f != 0 )
{
if ( temp -> coeff > 0 )
cout << " + " ;
else
cout << " " ;
}
if ( temp -> exp != 0 )
cout << temp -> coeff << "x^" << temp -> exp ;
else
cout << temp -> coeff ;
temp = temp -> link ;
```

```
f = 1 ;
}
}
void poly :: poly_add ( poly &l1, poly &l2 )
{
polynode *z ;
if ( l1.p == NULL && l2.p == NULL )
return ;
polynode *temp1, *temp2 ;
temp1 = l1.p ;
temp2 = l2.p ;
while ( temp1 != NULL && temp2 != NULL )
{
if ( p == NULL )
{
p = new polynode ;
z = p ;
}
else
{
z -> link = new polynode ;
z = z -> link ;
}
if ( temp1 -> exp < temp2 -> exp )
{
z -> coeff = temp2 -> coeff ;
z -> exp = temp2 -> exp ;
temp2 = temp2 -> link ;
}
else
{
if ( temp1 -> exp > temp2 -> exp )
{
z -> coeff = temp1 -> coeff ;
z -> exp = temp1 -> exp ;
temp1 = temp1 -> link ;
}
else
{
if ( temp1 -> exp == temp2 -> exp )
{
z -> coeff = temp1 -> coeff + temp2 -> coeff ;
z -> exp = temp1 -> exp ;
temp1 = temp1 -> link ;
temp2 = temp2 -> link ;
}
}
}
}
while ( temp1 != NULL )
{
if ( p == NULL )
{
```

```
p = new polynode ;
z = p ;
}
else
{
z -> link = new polynode ;
z = z -> link ;
}
z -> coeff = temp1 -> coeff ;
z -> exp = temp1 -> exp ;
temp1 = temp1 -> link ;
}
while ( temp2 != NULL )
{
if ( p == NULL )
{
p = new polynode ;
z = p ;
}
else
{
z -> link = new  polynode ;
z = z -> link ;
}
z -> coeff = temp2 -> coeff ;
z -> exp = temp2 -> exp ;
temp2 = temp2 -> link ;
}
z -> link = NULL ;
}
poly :: ~poly( )
{
polynode *q ;
while ( p != NULL )
{
q = p -> link ;
delete p ;
p = q ;
}
}
void main( )
{
poly p1 ;
p1.poly_append ( 1.4, 5 ) ;
p1.poly_append ( 1.5, 4 ) ;
p1.poly_append ( 1.7, 2 ) ;
p1.poly_append ( 1.8, 1 ) ;
p1.poly_append ( 1.9, 0 ) ;
cout << "\nFirst polynomial:" ;
p1.display_poly( ) ;
poly p2 ;
p2.poly_append ( 1.5, 6 ) ;
p2.poly_append ( 2.5, 5 ) ;
```

```
p2.poly_append ( -3.5, 4 ) ;
p2.poly_append ( 4.5, 3 ) ;
p2.poly_append ( 6.5, 1 ) ;
cout << "\nSecond polynomial:" ;
p2.display_poly( ) ;
poly p3 ;
p3.poly_add ( p1, p2 ) ;
cout << "\nResultant polynomial: " ;
p3.display_poly( ) ;
getch();
}
```

A program in C++ to reverse a linked list by changing the link in the nodes

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>

class linklist
{
   private:
//structure containing data and link part
struct node
{
 int data;
 node *link;
}*p;

public:
 linklist();
void addatbeg(int num);
void reverse();
void display();
int count();
~linklist();
};
//initialize data members
linklist()::linklist()
{
 p=null;
}

//adds a new node at the beginning of the linked list
void linklist::addatbeg(int num)
{
 node *temp;
//add new node
temp=new node;
temp->data=num;
temp->link=p;
p=temp;
}
```

```
//reverse the linked list
void linklist::reverse()
{
 node *q,*r,*s;
q=p;
r=null;

//traverse the entire linked list
while(q!=null)
{
 s=r;
r=q;
q=q->link;
r->link=s;
}
p=r;
}
//display the contents of linked list
void linklist::display
{
 node *temp=p;
cout<<endl;
//traverse the entire linked list
while(temp!=null)
{
 cout<<temp->data<<"";
temp=temp->link;
}
}

//counts the number of nodes present in the linked list
int linklist::count()
{
 node *temp=p;
int c=0;
//traverse the entire linked list
while(temp!=null)
{
 temp=temp->link;
c++;
}
return c;
}

//deallocate memory
linklist::~linklist()
{
 node *q;
while(p!=null)
{
 q=p->link;
delete p;
```

```
p=q;
}
}
void main()
{
 linklist l;
l.addatbeg(7);
l.addatbeg(43);
l.addatbeg(17);
l.addatbeg(3);
l.addatbeg(23);
l.addatbeg(5);

cout<<”\nelements in the linked list before reversing”;
l.display();
cout<<”\nno.of elements in the list”<<l.count;

l.reverse();
cout<<”\nelements in the linked list after reversing”;
l.display();
cout<<”\nno.of elements in the list”<<l.count;

getch();
}
```

Week 3
Write a program in C++ to multiply two polynomials represented as linked lists? Write a program in C++ to implement a doubly linked list? Write a program to implement different operations like adding a node at beginning, end, center, after a certain element, after a certain count of nodes in a doubly linked list, deleting a node at beginning, end, center, after a certain element, after a certain count of nodes in a doubly linked list. Write a program to implement different operations of a circular linked list

A program in C++ to multiply two polynomials represented as linked lists

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>

class poly
{
private:
//structure containing data part and link part
struct polynode
{
float coeff;
int exp;
polynode *link;
}*p;
public:
poly();
void poly_append(float c,int e);
void display_poly();
void poly_multiply(poly &p1,poly &p2);
void padd(float c,int e);
~poly();
}
//initialise data member
poly::poly()
{
p=NULL;
}
//add a term to a polynomial
void poly::poly_append(float c,int e)
{
polynode *temp;
temp=p;
//create a new node if the list is empty
if(temp==NULL)
{
temp=new polynode;
```

```
p=temp;
}
else
{
//traverse the entire linke list
while(temp->link!=NULL)
temp=temp->link;
//create node at intermediate stages
temp->link=new polynode;
temp=temp->link;
}
//assign coefficient and exponent
temp->coeff=c;
temp->exp=e;
temp->link=NULL;
}
//Displays the contents of linked list representing a polynomial
void poly::display_poly()
{
polynode *temp=p;
int f=0;
//traverse till the end of the linked list
while(temp!=NULL)
{
if(f!=0)
{
//display sign of a term
if(temp->coeff>0)
cout<<"+";
else
cout<<"";
}
if(temp->exp!=0)
cout<<temp->coeff<<"x^"<<temp->exp;
else
cout<<temp->coeff;
temp=temp->link;
f=1;
}
}
//multiplies 2 polynomials
void poly::poly_multiply(poly &p1,poly &p2)
{
polynode *temp1,*temp2;
float coeff1,exp1;

//point to the starting of first linked list
temp1=p1.p;
//point to the starting of second linked list
temp2=p2.p;

if(temp1==NULL && temp2==NULL)
return;
```

```
//if one of the list is empty
if(temp1==NULL)
p=p2.p;
else
{
if(temp2==NULL)
p=temp1;
else   //if both linked list exist
{
//for each term of the first list
while(temp1!=NULL)
{
//multiply each term of the second linked list
//with a term of the first linked list
while(temp2!=NULL)
{
coeff1=temp1->coeff * temp2->coeff;
exp1=temp1->exp+temp2->exp;
temp2=temp2->link;

//add the new term to the resultant polynomial
padd(coeff1,exp1);
}
//reposition the pointer to the starting of the second ll
temp2=p2.p;
//go to the next node
temp1=temp1->link;
}
}
}
}
//adds a term to the polynomial in the descending
//order of the exponent
void poly::padd(float c,int e)
{
polynode *r,*temp;
temp=p;
//if list is empty or if the node is to be inserted before the 1st node
if(temp==NULL|| c>temp->exp)
{
r=new polynode;
r->coeff=c;
r->exp=e;
if(p==NULL)
{
r->link=NULL;
p=r;
}
else
{
r->link=temp;
p=r;
}
```

```
}
else
{
//traverse the entire linked list to
//search the position to insert a new node

while(temp!=NULL)
{
if(temp->exp==e)
{
temp->coeff+=c;
return;
}
if(temp->exp>c &&(temp->link->exp<c || temp->link==NULL))
{
r=new polynode;
r->coeff=c;
r->exp=e;
r->link=NULL;
temp->link=r;
return;
}
//go to next node
temp=temp->link;
}
r->link=NULL;
temp->link=r;
}
}
//deallocates memory
poly::~poly()
{
polynode *q;
while(p!=NULL)
{
q=p->link;
delete p;
p=q;
}
}
void main()
{
poly p1;
p1.poly_append(3,5);
p1.poly_append(2,4);
p1.poly_append(1,2);

cout<<"\nfirst polynomial"<<endl;
p1.display_poly();
poly p2;
p2.poly_append(1,6);
p2.poly_append(2,5);
p2.poly_append(3,4);
```

```
cout<<"\n second polynomial:"<<endl;
p2.display_poly();

poly p3;
p3.poly_multiply(p1,p2);

cout<<"\nresultant polynomial"<<endl;
p3.display_poly();
getch();
}
```

A Program in C++ to implement a doubly linked list

Implement different operations like adding a node at beginning, end, center, after a certain element, after a certain count of nodes in a doubly linked list, deleting a node at beginning, end, center, after a certain element, after a certain count of nodes in a doubly linked list

```
/*C++ Program to Implement Doubly Linked List*/
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
/*
* Node Declaration
*/

struct node
{
int info;
struct node *next;
struct node *prev;
}*start;

/*
Class Declaration
*/
class double_llist
{
public:
void create_list(int value);
void add_begin(int value);
void add_after(int value, int position);
void delete_element(int value);
void search_element(int value);
void display_dlist();
void count();
void reverse();
double_llist()
{
```

```
start = NULL;
}
};

/*
* Main: Conatins Menu
*/
int main()
{
int choice, element, position;
double_llist dl;
while (1)
{
cout<<endl<<"----------------------------"<<endl;
cout<<endl<<"Operations on Doubly linked list"<<endl;
cout<<endl<<"----------------------------"<<endl;
cout<<"1.Create Node"<<endl;
cout<<"2.Add at begining"<<endl;
cout<<"3.Add after position"<<endl;
cout<<"4.Delete"<<endl;
cout<<"5.Display"<<endl;
cout<<"6.Count"<<endl;
cout<<"7.Reverse"<<endl;
cout<<"8.Quit"<<endl;
cout<<"Enter your choice : ";
cin>>choice;
switch ( choice )
{
case 1:
cout<<"Enter the element: ";
cin>>element;
dl.create_list(element);
cout<<endl;
break;
case 2:
cout<<"Enter the element: ";
cin>>element;
dl.add_begin(element);
cout<<endl;
break;
case 3:
cout<<"Enter the element: ";
cin>>element;
cout<<"Insert Element after postion: ";
cin>>position;
dl.add_after(element, position);
cout<<endl;
break;
case 4:
if (start == NULL)
{
cout<<"List empty,nothing to delete"<<endl;
break;
```

```
}
cout<<"Enter the element for deletion: ";
cin>>element;
dl.delete_element(element);
cout<<endl;
break;
case 5:
dl.display_dlist();
cout<<endl;
break;
case 6:
dl.count();
break;
case 7:
if (start == NULL)
{
cout<<"List empty,nothing to reverse"<<endl;
break;
}
dl.reverse();
cout<<endl;
break;
case 8:
exit(0);
default:
cout<<"Wrong choice"<<endl;
}
}
return 0;
}

/*
* Create Double Link List
*/
void double_llist::create_list(int value)
{
struct node *s, *temp;
temp = new(struct node);
temp->info = value;
temp->next = NULL;
if (start == NULL)
{
temp->prev = NULL;
start = temp;
}
else
{
s = start;
while (s->next != NULL)
s = s->next;
s->next = temp;
temp->prev = s;
}
```

```
}

/*
* Insertion at the beginning
*/
void double_llist::add_begin(int value)
{
if (start == NULL)
{
cout<<"First Create the list."<<endl;
return;
}
struct node *temp;
temp = new(struct node);
temp->prev = NULL;
temp->info = value;
temp->next = start;
start->prev = temp;
start = temp;
cout<<"Element Inserted"<<endl;
}

/*
* Insertion of element at a particular position
*/
void double_llist::add_after(int value, int pos)
{
if (start == NULL)
{
cout<<"First Create the list."<<endl;
return;
}
struct node *tmp, *q;
int i;
q = start;
for (i = 0;i < pos - 1;i++)
{
q = q->next;
if (q == NULL)
{
cout<<"There are less than ";
cout<<pos<<" elements."<<endl;
return;
}
}
tmp = new(struct node);
tmp->info = value;
if (q->next == NULL)
{
q->next = tmp;
tmp->next = NULL;
tmp->prev = q;
}
```

```
else
{
tmp->next = q->next;
tmp->next->prev = tmp;
q->next = tmp;
tmp->prev = q;
}
cout<<"Element Inserted"<<endl;
}

/*
* Deletion of element from the list
*/
void double_llist::delete_element(int value)
{
struct node *tmp, *q;
/*first element deletion*/
if (start->info == value)
{
tmp = start;
start = start->next;
start->prev = NULL;
cout<<"Element Deleted"<<endl;
free(tmp);
return;
}
q = start;
while (q->next->next != NULL)
{
/*Element deleted in between*/
if (q->next->info == value)
{
tmp = q->next;
q->next = tmp->next;
tmp->next->prev = q;
cout<<"Element Deleted"<<endl;
free(tmp);
return;
}
q = q->next;
}
/*last element deleted*/
if (q->next->info == value)
{
tmp = q->next;
free(tmp);
q->next = NULL;
cout<<"Element Deleted"<<endl;
return;
}
cout<<"Element "<<value<<" not found"<<endl;
}
```

```
/*
* Display elements of Doubly Link List
*/
void double_llist::display_dlist()
{
struct node *q;
if (start == NULL)
{
cout<<"List empty,nothing to display"<<endl;
return;
}
q = start;
cout<<"The Doubly Link List is :"<<endl;
while (q != NULL)
{
cout<<q->info<<" <-> ";
q = q->next;
}
cout<<"NULL"<<endl;
}

/*
* Number of elements in Doubly Link List
*/
void double_llist::count()
{
struct node *q = start;
int cnt = 0;
while (q != NULL)
{
q = q->next;
cnt++;
}
cout<<"Number of elements are: "<<cnt<<endl;
}

/*
* Reverse Doubly Link List
*/
void double_llist::reverse()
{
struct node *p1, *p2;
p1 = start;
p2 = p1->next;
p1->next = NULL;
p1->prev = p2;
while (p2 != NULL)
{
p2->prev = p2->next;
p2->next = p1;
p1 = p2;
p2 = p2->prev;
}
```

```
start = p1;
cout<<"List Reversed"<<endl;
}
```

A Program to implement different operations of a circular linked list

```
/* C++ Program to Implement Circular Linked List operations*/
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

// Node Declaration

struct node
{
int info;
struct node *next;
}*last;

/*
* Class Declaration
*/
class circular_llist
{
public:
void create_node(int value);
void add_begin(int value);
void add_after(int value, int position);
void delete_element(int value);
void display_list();

circular_llist()
{
last = NULL;
}
};

/*
* Main :contains menu
*/
void main()
{
int choice, element, position;
circular_llist cl;
while (1)
{
cout<<endl<<"----------------------------"<<endl;
cout<<endl<<"Circular singly linked list"<<endl;
cout<<endl<<"----------------------------"<<endl;
cout<<"1.Create Node"<<endl;
cout<<"2.Add at beginning"<<endl;
cout<<"3.Add after"<<endl;
cout<<"4.Delete"<<endl;
cout<<"5.Display"<<endl;
```

```
cout<<"6.Quit"<<endl;
cout<<"Enter your choice : ";
cin>>choice;
switch(choice)
{
case 1:
cout<<"Enter the element: ";
cin>>element;
cl.create_node(element);
cout<<endl;
break;
case 2:
cout<<"Enter the element: ";
cin>>element;
cl.add_begin(element);
cout<<endl;
break;
case 3:
cout<<"Enter the element: ";
cin>>element;
cout<<"Insert element after position: ";
cin>>position;
cl.add_after(element, position);
cout<<endl;
break;
case 4:
if (last == NULL)
{
cout<<"List is empty, nothing to delete"<<endl;
break;
}
cout<<"Enter the element for deletion: ";
cin>>element;
cl.delete_element(element);
cout<<endl;
break;
case 5:
cl.display_list();
break;
case 6:
exit(1);
break;
default:
cout<<"Wrong choice"<<endl;
}
}
// return 0;
getch();
}

/*
* Create Circular Link List
*/
void circular_llist::create_node(int value)
{
struct node *temp;
temp = new(struct node);
```

```
temp->info = value;
if (last == NULL)
{
last = temp;
temp->next = last;
}
else
{
temp->next = last->next;
last->next = temp;
last = temp;
}
}



/*
* Insertion of element at beginning
*/
void circular_llist::add_begin(int value)
{
if (last == NULL)
{
cout<<"First Create the list."<<endl;
return;
}
struct node *temp;
temp = new(struct node);
temp->info = value;
temp->next = last->next;
last->next = temp;
}

/*
* Insertion of element at a particular place
*/
void circular_llist::add_after(int value, int pos)
{
if (last == NULL)
{
cout<<"First Create the list."<<endl;
return;
}
struct node *temp, *s;
s = last->next;
for (int i = 0;i < pos-1;i++)
{
s = s->next;
if (s == last->next)
{
cout<<"There are less than ";
cout<<pos<<" in the list"<<endl;
return;
}
}
temp = new(struct node);
temp->next = s->next;
```

```
temp->info = value;
s->next = temp;
/*Element inserted at the end*/
if (s == last)
{
last=temp;
}
}

/*
* Deletion of element from the list
*/
void circular_llist::delete_element(int value)
{
struct node *temp, *s;
s = last->next;
/* If List has only one element*/
if (last->next == last && last->info == value)
{
temp = last;
last = NULL;
free(temp);
return;
}
if (s->info == value)  /*First Element Deletion*/
{
temp = s;
last->next = s->next;
free(temp);
return;
}
while (s->next != last)
{
/*Deletion of Element in between*/
if (s->next->info == value)
{
temp = s->next;
s->next = temp->next;
free(temp);
cout<<"Element "<<value;
cout<<" deleted from the list"<<endl;
return;
}
s = s->next;
}
/*Deletion of last element*/
if (s->next->info == value)
{
temp = s->next;
s->next = last->next;
free(temp);
last = s;
return;
}
cout<<"Element "<<value<<" not found in the list"<<endl;
}
```

```
/* Display Circular Link List
*/
void circular_llist::display_list()
{
struct node *s;
if (last == NULL)
{
cout<<"List is empty, nothing to display"<<endl;
return;
}
s = last->next;
cout<<"Circular Link List: "<<endl;
while (s != last)
{
cout<<s->info<<"->";
s = s->next;
}
cout<<s->info<<endl;
}
```

Week 4
Write a program to implement various operations on an array based stack? Write a program to implement various operations on an stack represented using linked list. Write a program to demonstrate the use of stack in checking whether the arithmetic expression is properly parenthesized? Write a program to demonstrate the use of stack in converting an arithmetic expression from infix to postfix? Write a program to demonstrate the use of stack in evaluating an arithmetic expression in postfix notation?

A program to implement various operations on an array based stack

```
// stack --- Array implementation//
#include<stdio.h>
#include<conio.h>
#include<iostream.h>
const int max=10;
class stack
{
private:
int arr[max];
int top;
public:
stack();
void push(int item);
int pop();
};
//initialises data member
stack::stack()
{
```

```
top=-1;
}
//adds an element to the stack
void stack::push(int item)
{
if(top==max-1)
{
cout<<endl<<"stack is full";
return;
}
top++;
arr[top]=item;
}
//extracts an element from the stack
int stack::pop()
{
if(top==-1)
{
cout<<endl<<"stack is empty";
return NULL;
}
int data=arr[top];
top--;
return data;
}
void main()
{
stack s;
s.push(11);
s.push(23);
s.push(-8);
s.push(16);
s.push(27);
s.push(14);
s.push(20);
s.push(39);
s.push(2);
s.push(15);
s.push(7);

int i=s.pop();
cout<<"\nitem popoed:"<<i;

i=s.pop();
cout<<"\nitem popoed:"<<i;

i=s.pop();
cout<<"\nitem popoed:"<<i;

i=s.pop();
cout<<"\nitem popoed:"<<i;
getch();
}
```

Write a program to implement various operations on stack represented using linked list.

```
#include<stdio.h>
#include<conio.h>
#include<iostream.h>

class stack
{
private:
//structure containing data part and link part
struct node
{
int data;
node *link;
}*top;
public:
stack();
void push(int item);
int pop();
~stack();
};
//initialises data member
stack::stack()
{
top=NULL;
}
//adds a new node to the stack as linked list
void stack::push(int item)
{
node *temp;
temp=new node;
if(temp==NULL)
cout<<endl<<"stack is full";
temp->data=item;
temp->link=top;
top=temp;
}
//pops an element from the stack
int stack::pop()
{
if(top==NULL)
{
cout<<endl<<"stack is empty";
return NULL;
}
node *temp;
int item;

temp=top;
item=temp->data;
top=top->link;
```

```
delete temp;
return item;
}
//deallocates memory
stack::~stack()
{
if(top==NULL)
return;
node *temp;
while(top!=NULL)
{
temp=top;
top=top->link;
delete temp;
}
}
void main()
{
stack s;
s.push(14);
s.push(-3);
s.push(18);
s.push(29);
s.push(31);
s.push(16);
int i=s.pop();
cout<<"\nitem popped:"<<i;

i=s.pop();
cout<<"\nitem popped:"<<i;

i=s.pop();
cout<<"\nitem popped:"<<i;
getch();
}
```

A program to demonstrate the use of stack in converting an arithmetic expression from infix to postfix

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
const int max=50;
class infix
{
private:
char target[max],stack[max];
char *s,*t;
```

```
int top;
public:
infix();
void setexpr(char *str);
void push(char c);
char pop();
void convert();
int priority(char c);
void show();
};
//initialise data members
infix::infix()
{
top=-1;
strcpy(target,"");
strcpy(stack,"");
t=target;
s="";
}
//sets s to point to given expr
void infix::setexpr(char *str)
{
s=str;
}
//adds an operator to the stack
void infix::push(char c)
{
if(top==max)
cout<<"stack is full\n";
else
{
top++;
stack[top]=c;
}
}
//pops an operator from the stack
char infix::pop()
{
if(top==-1)
{
cout<<"stack is empty\n";
return-1;
}
else
{
char item=stack[top];
top--;
return item;
}
}
//converts the given expr from infix to postfix form
void infix::convert()
{
```

```
while(*s)
{
if(*s==' '||*s=='\t')
{
s++;
continue;
}
if(isdigit(*s)||isalpha(*s))
{
while(isdigit(*s)||isalpha(*s))
{
*t=*s;
s++;
t++;
}
}
if(*s=='(')
{
push(*s);
s++;
}
char opr;
if(*s=='*'||*s=='+'||*s=='/'||*s=='%'||*s=='-'||*s=='$')
{
if(top!=-1)
{
opr=pop();
while(priority(opr)>=priority(*s))
{
*t=opr;
t++;
opr=pop();
}
push(opr);
push(*s);
}
else
push(*s);
s++;
}
if(*s==')')
{
opr=pop();
while((opr)!='(')
{
*t=opr;
t++;
opr=pop();
}
s++;
}
}
while(top!=-1)
```

```
{
char opr=pop();
*t=opr;
t++;
}
*t='\0';
}
//returns the priority of an operator
int infix::priority(char c)
{
if(c=='$')
return 3;
if(c=='*'||c=='/'||c=='%')
return 2;
else
{
if(c=='+'||c=='-')
return 1;
else
return 0;
}
}
//displays the postfix form of given expr
void infix::show()
{
cout<<target;
}
void main()
{
char expr[max];
infix q;
cout<<"\nenter an expression in infix form";
cin.getline(expr,max);
q.setexpr(expr);
q.convert();
cout<<"the postfix expression is";
q.show();
getch();
}
```

Write a program to demonstrate the use of stack in evaluating an arithmetic expression in postfix notation

```
///evaluate postfix expression//

#include<stdio.h>

#include<conio.h>

#include<iostream.h>

#include<math.h>

#include<ctype.h>
```

```
const int max=50;
class postfix
{
private:
int stack[max];
int top,nn;
char *s;
public:
postfix();
void setexpr(char *str);
void push(int item);
int pop();
void calculate();
void show();
};
//initialses data members
postfix::postfix()
{
top=-1;
}
//sets s to point to the given expr
void postfix::setexpr(char *str)
{
s=str;
}
//adds digit to the stack
void postfix::push(int item)
{
if(top==max-1)
cout<<endl<<"stack is full";
else
```

```
{
top++;
stack[top]=item;
}
}
//pops digit from the stack
int postfix::pop()
{
if(top==-1)
{
cout<<endl<<"stack is empty";
return NULL;
}
int data=stack[top];
top--;
return data;
}
//evaluates the postfix expression
void postfix::calculate()
{
int n1,n2,n3;
while(*s)
{
//skip whitespace,if any
if(*s==' '||*s=='\t')
{
s++;
continue;
}
//if digit is encountered
if(isdigit(*s))
{
```

```
nn=*s-'0';
push(nn);
}
else
{
//if operator is encountered
n1=pop();
n2=pop();
switch(*s)
{
case'+':
n3=n2+n1;
break;
case'-':
n3=n2-n1;
break;
case'/':
n3=n2/n1;
break;
case'*':
n3=n2*n1;
break;
case'%':
n3=n2%n1;
break;
case'$':
n3=pow(n2,n1);
break;
default:
cout<<"unknown operator";
}
push(n3);
```

```
}
s++;
}
}
//displays the result
void postfix::show()
{
nn=pop();
cout<<"result is:"<<nn;
}
void main()
{
char expr[max];
cout<<"\nenter postfix expression to be evaluted:";
cin.getline(expr,max);
postfix q;
q.setexpr(expr);
q.calculate();
q.show();
getche();
}
```

Program to demonstrate the use of stack in checking whether the arithmetic expression is properly parenthesized

```
/*Program to check whether given expression have balanced parenthesis.
E.g., {[(){}]} ->Balanced ; {[([}]]} ->Not balanced.*/

#include <iostream.h>
#include <string.h>
#include<conio.h>

#define MAX_SIZE 100


/*We need a stack with basic operations push, pop, isEmpty, isFull.
This stack is required to store and pop the parenthesis for proper
comparision in LIFO manner */
```

```
class stack{
char data[MAX_SIZE];
int top;
public:
stack()
{
top = -1;
}
int push(char ch)
{
if (top < MAX_SIZE-1)
{
top++;
data[top] = ch;
return 1;
}
return 0;
}
char pop()
{
if(top>=0)
{
char ch = data[top];
top--;
return ch;
}
return NULL;
}
int isFull()
{
if(top >= MAX_SIZE-1)
return 1;
return 0;
}
int isEmpty()
{
if(top < 0)
return 1;
return 0;
}
};
/*
checkParenthesisMatch(char,char) would return false if given chars are not
appropriate.
*/
int checkParenthesisMatch(char ch1, char ch2)
{
int result = 0;
if(    (ch1 == ']' && ch2 == '[') ||
(ch1 == '}' && ch2 == '{') ||
(ch1 == ')' && ch2 == '(') )
result = 1;
return result;
}
/*
checkBalancedParenthesis(char[]) would return false if parenthesis are not
balanced.
```

```
*/
int checkBalancedParenthesis(char exp[])
{
int len = strlen(exp);
int result = 1;
stack st;
for(int i=0;i<len;i++)
{
if(    (exp[i] == '[') ||
(exp[i] == '{') ||
(exp[i] == '(') )
{
st.push(exp[i]);
}
else if((exp[i] == ']') ||
(exp[i] == '}') ||
(exp[i] == ')') )
{
if(checkParenthesisMatch(exp[i],st.pop()) == 0)
{
result = 0;
break;
}
}
}
/*There exists a case where even after successful result from above
statements
still the possibility of imbalance exists; e.g., [{}()[()]  -->Missed one
more
closing parenthesis. Check this case by checking stack space
*/
if(st.isEmpty()== 0)
result = 0;

return result;
}
void main()
{
clrscr();
char str[MAX_SIZE];
cout<<"Enter parenthesis expression that you wish to check for:"<<endl;
cin>>str;
if(checkBalancedParenthesis(str) == 1)
cout<<"Given expression is balanced";
else
cout<<"Given expression is not balanced";
getch();
}
```

Week 5
Write a program to demonstrate the implementation of various operations on a linear queue represented using a linear array. Write a program to demonstrate the implementation of various operations on a Circular queue represented using a linear array. Write a program to demonstrate the implementation of various operations on a queue represented using a linked list? *ASSIGNMENT:* Write a program to demonstrate the use of stack in implementing quicksort algorithm to sort an array of integers in ascending order?

Program to demonstrate the implementation of various operations on a linear queue represented using a linear array.

```
//implementation of queue using linear array//
#include<iostream.h>
#include<stdio.h>
#include<conio.h>

const int max=10;
class queue
{
private:
int arr[max];
int front,rear;

public:
queue();
void addq(int item);
int delq();
};

//initialize data members
queue::queue()
{
front=-1;
rear=-1;
}

//adds an element to the queue
void queue::addq(int item)
{
if(rear==max-1)
{
cout<<"\n queue is full";
return;
}
rear++;
arr[rear]=item;

if(front==-1)
front=0;
}

//removes an element from the queue
```

```
int queue::delq()
{
int data;
if (front==-1)
{
cout<<"queue is empty";
return NULL;
}

data=arr[front];
arr[front]=0;
if(front==rear)
front=rear=-1;
else
front++;
return data;
}

void main()
{
clrscr();
queue a;

a.addq(23);
a.addq(9);
a.addq(11);
a.addq(17);
a.addq(24);

int i=a.delq();
cout<<"\n item deleted"<<i;

i=a.delq();
cout<<"\nitem deleted"<<i;

i=a.delq();
cout<<"\nitem deleted"<<i;
getch();
}
```

Program to demonstrate the implementation of various operations on a Circular queue represented using a linear array.

```
// implementation of circular queue//
#include<iostream.h>
#include<stdio.h>
#include<conio.h>

const int max=10;
class queue
{
private:
int arr[max];
```

```
int front,rear;

public:
queue();
void addq(int item);
int delq();
void display();
};
//initialize
queue::queue()
{
front=rear=-1;
for(int i=0;i<max;i++)
arr[i]=0;
}

//add data member
void queue::addq(int item)
{
if((rear==max-1 && front==0)|| (rear+1==front))
{
cout<<"\n queue is full";
return;
}
if(rear==max-1)
rear=0;
else
rear++;
arr[rear]=item;
if(front==-1)
front=0;
}

//removes an element from the queue
int queue::delq()
{
int data;
if(front==-1)
{
cout<<"\nqueue is empty";
return NULL;
}
data=arr[front];
arr[front]=0;

if(front==rear)
{
front=-1;
rear=-1;
}
else
{
if(front==max-1)
```

```
front=0;
else
front++;
}
return data;
}

//displays elements in a queue
void queue::display()
{
cout<<endl;
for(int i=0;i<max;i++)
cout<<arr[i]<<"";
cout<<endl;
}

void main()
{
clrscr();
queue a;
a.addq(10);
a.addq(22);
a.addq(54);
a.addq(7);
a.addq(8);

cout<<"\nelements in the circular queue";
a.display();

int i=a.delq();
cout<<"item delete"<<i;

i=a.delq();
cout<<"item delete"<<i;

cout<<"\nelements in the circular queue after deletion";
a.display();

a.addq(12);
a.addq(13);
a.addq(14);
a.addq(15);

cout<<"elements after addition into c.queue";
a.display();

a.addq(32);
cout<<"elements after addition";
a.display();
getch();
}
```

Program to demonstrate the implementation of various operations on a queue represented using a linked list

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>

class queue
{
private:
struct node
{
int data;
node *link;
}*front,*rear;
public:
queue();
void addq(int item);
int delq();
~queue();
};

//initialises data memebers
queue::queue()
{
front=rear=NULL;
}
//Add an element to the queue
void queue::addq(int item)
{
node *temp;
temp=new node;
if(temp==NULL)
cout<<"\nqueue is full";
temp->data=item;
temp->link=NULL;
if(front==NULL)
{
rear=front=temp;
return;
}
rear->link=temp;
rear=rear->link;
}
//remove element from queue
int queue::delq()
{
if(front==NULL)
{
cout<<"\nqueue is empty";
return NULL;
}
node *temp;
```

```
int item;
item=front->data;
temp=front;
front=front->link;
delete temp;
return item;
}
queue::~queue()
{
if(front==NULL)
return;
node *temp;

while(front!=NULL)
{
temp=front;
front=front->link;
delete temp;
}
}
void main()
{
clrscr();
queue a;
a.addq(11);
a.addq(23);
a.addq(33);
a.addq(45);
a.addq(78);

int i=a.delq();
cout<<"\nitem extracted"<<i;

i=a.delq();
cout<<"\nitem extracted"<<i;
i=a.delq();
cout<<"\nitem extracted"<<i;
getch();
}
```

Week 6
Write a program in C++ to create a binary tree? Write a program to implement the traversal techniques of a binary tree? *ASSIGNMENT*: Write a program to demonstrate the use of multiple stacks?

Program in C++ to create a Binary tree & Binary Tree Traversals

```
#include<iostream.h>
#include<conio.h>

class Bintree
{
struct Node
{
int num;
struct Node *left,*right;
};

Node *root;

public:

Bintree()
{
root=NULL;
}

void create();
void inorder( Node * );
void preorder( Node * );
void postorder( Node * );

void display();

};

void Bintree :: create()
{
Node *prev,*temp,*nn;
nn=new Node;
nn->left=nn->right=NULL;
cout<<"\n\nEnter a number : ";
cin>>nn->num;

if( root==NULL )
{
root=nn;
return;
```

```
}

temp=root;
prev=NULL;

while( temp!=NULL )
{
prev=temp;

if( nn->num > temp->num )
temp=temp->right;
else
temp=temp->left;
}

if( nn->num > prev->num )
prev->right=nn;
else
prev->left=nn;
}

void Bintree :: display()
{
if( root == NULL )
{
cout<<"\n\nBinary tree is empty !!";
getch();
return;
}

cout<<"\n\n\nInorder display is..\n\n";
inorder( root );

cout<<"\n\n\npreorder display is..\n\n";
preorder( root );

cout<<"\n\n\npostorder display is..\n\n";
postorder( root );

getch();
}

void Bintree :: inorder( Node *r )
{
if( r!=NULL )
{
inorder( r->left );
cout<<r->num<<"  ";
inorder( r->right );
}
}

void Bintree :: preorder( Node *r )
```

```
{
if( r!=NULL )
{
cout<<r->num<<"  ";
preorder( r->left );
preorder( r->right );
}
}

void Bintree :: postorder( Node *r )
{
if( r!=NULL )
{
postorder( r->left );
postorder( r->right );
cout<<r->num<<"  ";

}
}

int main()
{
int ch;
Bintree b1;

do
{
clrscr();

cout<<"\n1.Create\n\n2.Display\n\n3.Exit\n\nEnter your chocie : ";
cin>>ch;

switch( ch )
{
case 1: b1.create();    break;

case 2: b1.display();   break;
}
}while( ch!=3 );

getch();
return 0;
}
```

Week 7
Write a program to delete a node in a binary search tree? Write a program to implement the different operations of an AVL tree? Write a program to implement the different operations of a threaded binary tree. Write a program to implement the different operations of a M-way search tree?

Program to delete a node in a binary search tree

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>

#define true 1
#define false 0

class btree
{
private:
struct btreenode
{
btreenode *leftchild;
int data;
btreenode *rightchild;
}*root;
public:
btree();
void buildtree(int num);
static void insert(btreenode **sr,int);
static void search(btreenode **sr,int num,btreenode **par,btreenode **x,int
*found);
void remove(int num);
static void rem(btreenode **sr,int num);
void display();
static void inorder(btreenode *sr);
~btree();
static void del(btreenode *sr);
};
//initialises data member
btree::btree()
{
root=NULL;
}
//calls insert to build tree
void btree::buildtree(int num)
{
insert(&root,num);
}
//insert a new node in a binary search tree
void btree::insert(btreenode **sr,int num)
{
if(*sr==NULL)
{
```

```
*sr=new btreenode;
(*sr)->leftchild=NULL;
(*sr)->data=num;
(*sr)->rightchild=NULL;
}
else
{
if(num<(*sr)->data)
insert(&((*sr)->leftchild),num);
else
insert(&((*sr)->rightchild),num);
}
}
//calls rem to delete node
void btree::remove(int num)
{
rem(&root,num);
}

void btree::rem(btreenode **sr,int num)
{
int found;
btreenode *parent,*x,*xsucc;
//if tree is empty
if(*sr==NULL)
{
cout<<"\n tree is empty";
return;
}
parent=x=NULL;
//call swearch to find the node to be deleted
search(sr,num,&parent,&x,&found);
if(found==false)
{
cout<<"\ndata to be deleted not found";
return;
}
//if the node to be deleted has two childern
if(x->leftchild!=NULL && x->rightchild!=NULL)
{
parent=x;
xsucc=x->rightchild;
while(xsucc->leftchild!=NULL)
{
parent=xsucc;
xsucc=xsucc->leftchild;
}
x->data=xsucc->data;
x=xsucc;
}
//if the node to be deleted has no child
if(x->leftchild==NULL && x->rightchild==NULL)
{
```

```
if(parent->rightchild==x)
parent->rightchild=NULL;
else
parent->leftchild=NULL;
delete x;
return;
}
//if the node to be deleted has only right child
if(x->leftchild==NULL && x->rightchild!=NULL)
{
if(parent->leftchild==x)
parent->leftchild=x->rightchild;
else
parent->rightchild=x->rightchild;
delete x;
return;
}
//if the node to be deleted has only left child
if(x->leftchild!=NULL && x->rightchild==NULL)
{
if(parent->leftchild==x)
parent->leftchild=x->leftchild;
else
parent->rightchild=x->leftchild;
delete x;
return;
}
}
//return the address of the node to be deleted
//address of its parent and whether the node is found or noot
void btree::search(btreenode **sr,int num,btreenode **par,btreenode **x,int
*found)
{
btreenode *q;
q=*sr;
*found=false;
*par=NULL;
while(q!=NULL)
{
if(q->data==num)
{
*found=true;
*x=q;
return;
}
*par=q;
if(q->data>num)
q=q->leftchild;
else
q=q->rightchild;
}
}
//calls inorder() to traverse tree
```

```
void btree::display()
{
inorder(root);
}
//traverse a binary search tree in a LDR
void btree::inorder(btreenode *sr)
{
if(sr!=NULL)
{
inorder(sr->leftchild);
cout<<sr->data<<"\t";
inorder(sr->rightchild);
}
}
//calls del to deallocate memory
btree::~btree()
{
del(root);
}
//deletes nodes of a binary tree
void btree::del(btreenode *sr)
{
if(sr!=NULL)
{
del(sr->leftchild);
del(sr->rightchild);
}
delete sr;
}
void main()
{
clrscr();
btree bt;
int req,i=0,num,a[]={11,9,13,8,10,12,14,15,7};
while(i<=8)
{
bt.buildtree(a[i]);
i++;
}
cout<<"binary tree before deletion:\n";
bt.display();

bt.remove(10);
cout<<"\n binary tree after deletion:\n";
bt.display();

bt.remove(14);
cout<<"\nbinary tree after deletion\n";
bt.display();

bt.remove(13);
cout<<"\nbinary tree after deletion\n";
bt.display();
```

```
getch();
}
```

Program to implement the different operations of an AVL tree

```
//Implementation of AVL TREE//
#include<iostream.h>
#include<conio.h>
#include<stdio.h>

#define false 0
#define true 1

struct avlnode
{
int data;
int balfact;
avlnode *left;
avlnode *right;
};
class avltree
{
private:
avlnode *root;
public:
avltree();
avlnode* insert(int data,int *h);
static avlnode* buildtree(avlnode *root,int data,int *h);
void display(avlnode *root);
avlnode* deldata(avlnode *root,int data,int *h);
static avlnode* del(avlnode *node,avlnode* root,int *h);
static avlnode* balright(avlnode *root,int *h);
static avlnode* balleft(avlnode *root,int *h);
void setroot(avlnode *avl);
~avltree();
static void deltree(avlnode *root);
};

//initialises data member
avltree::avltree()
{
root=NULL;
}
//insert an element in a binary tree by calling buildtree
avlnode* avltree::insert(int data,int *h)
{
root=buildtree(root,data,h);
return root;
}
//inserts an element into tree
avlnode* avltree::buildtree(avlnode *root,int data,int *h)
{
```

```
avlnode *node1,*node2;
if(root==NULL)
{
root=new avlnode;
root->data=data;
root->left=NULL;
root->right=NULL;
root->balfact=0;
*h=true;
return(root);
}
if(data<root->data)
{
root->left=buildtree(root->left,data,h);
//if left subtree is higher
if(*h)
{
switch(root->balfact)
{
case 1:
node1=root->left;
if(node1->balfact==1)
{
cout<<"\nright rotation";
root->left=node1->right;
node1->right=root;
root->balfact=0;
root=node1;
}
else
{
cout<<"\ndouble rotation,left then right";
node2=node1->right;
node1->right=node2->left;
node2->left=node1;
root->left=node2->right;
node2->right=root;
if(node2->balfact==1)
root->balfact=-1;
else
root->balfact=0;
if(node2->balfact==-1)
node1->balfact=1;
else
node1->balfact=0;
root=node2;
}
root->balfact=0;
*h=false;
break;
case 0:
root->balfact=1;
break;
```

```
case -1:
root->balfact=0;
*h=false;
}
}
}
if(data>root->data)
{
root->right=buildtree(root->right,data,h);
if(*h)
{
switch(root->balfact)
{
case 1:
root->balfact=0;
*h=false;
break;
case 0:
root->balfact=-1;
break;

case -1:
node1=root->right;
if(node1->balfact==-1)
{
cout<<"\nleft rotation";
root->right=node1->left;
node1->left=root;
root->balfact=0;
root=node1;
}
else
{
cout<<"\n double rotation,right then left";
node2=node1->left;
node1->left=node2->right;
node2->right=node1;
root->right=node2->left;
node2->left=root;

if(node2->balfact==-1)
root->balfact=1;
else
root->balfact=0;
if(node2->balfact==1)
node1->balfact=-1;
else
node1->balfact=0;
root=node2;
}
root->balfact=0;
*h=false;
}
```

```
}
}
return(root);
}
//prints data
void avltree::display(avlnode *root)
{
if(root!=NULL)
{
display(root->left);
cout<<root->data<<"\t";
display(root->right);
}
}
//to delete an item from the tree
avlnode* avltree::deldata(avlnode *root,int data,int *h)
{
avlnode *node;
if(root->data==13)
cout<<root->data;
if(root==NULL)
{
cout<<"no such data";
return(root);
}
else
{
if(data<root->data)
{
root->left=deldata(root->left,data,h);
if(*h)
root=balright(root,h);
}
else
{
if(data>root->data)
{
root->right=deldata(root->right,data,h);
if(*h)
root=balleft(root,h);
}
else
{
node=root;
if(node->right==NULL)
{
root=node->left;
*h=true;
delete(node);
}
else
{
if(node->left==NULL)
```

```
{
root=node->right;
*h=true;
delete(node);
}
else
{
node->right=del(node->right,node,h);
if(*h)
root=balleft(root,h);
}
}
}
}
}
return(root);
}
//deleted given node
avlnode* avltree::del(avlnode *succ,avlnode *node,int *h)
{
avlnode *temp=succ;
if(succ->left!=NULL)
{
succ->left=del(succ->left,node,h);
if(*h)
succ=balright(succ,h);
}
else
{
temp=succ;
node->data=succ->data;
succ=succ->right;
delete(temp);
*h=true;
}
return(succ);
}
//to balance the tree
avlnode* avltree::balright(avlnode *root,int *h)
{
avlnode *temp1,*temp2;
switch(root->balfact)
{
case 1:
root->balfact=0;
break;

case 0:
root->balfact=-1;
*h=false;
break;

case -1:
```

```
temp1=root->right;
if(temp1->balfact<=0)
{
cout<<"\nleft rotation";
root->right=temp1->left;
temp1->left=root;
if(temp1->balfact==0)
{
root->balfact=-1;
temp1->balfact=1;
*h=false;
}
else
{
root->balfact=temp1->balfact=0;
}
root=temp1;
}
else
{
cout<<"\ndouble rotation,right then left:";
temp2=temp1->left;
temp1->left=temp2->right;
temp2->right=temp1;
root->right=temp2->left;
temp2->left=root;

if(temp2->balfact==-1)
root->balfact=1;
else
root->balfact=0;
if(temp2->balfact==1)
temp1->balfact=-1;
else
temp1->balfact=0;
root=temp2;
temp2->balfact=0;
}
}
return(root);
}

//to balance the tree if left sub tree is higher
avlnode* avltree::balleft(avlnode *root,int *h)
{
avlnode *temp1,*temp2;
switch(root->balfact)
{
case -1:
root->balfact=0;
break;

case 0:
```

```
root->balfact=1;
*h=false;
break;

case1:
temp1=root->left;
if(temp1->balfact>=0)
{
cout<<"\nright rotation";
root->left=temp1->right;
temp1->right=root;
if(temp1->balfact==0)
{
root->balfact=1;
temp1->balfact=-1;
*h=false;
}
else
{
root->balfact=temp1->balfact=0;
}
root=temp1;
}
else
{
cout<<"\ndouble rotation,left the right";
temp2=temp1->right;
temp1->right=temp2->left;
temp2->left=temp1;
root->left=temp2->right;
temp2->right=root;

if(temp2->balfact==1)
root->balfact=-1;
else
root->balfact=0;

if(temp2->balfact==-1)
temp1->balfact=1;
else
temp1->balfact=0;
root=temp2;
temp2->balfact=0;
}
}
return(root);
}
//sets new the root node
void avltree::setroot(avlnode *avl)
{
root=avl;
}
//calls deltree to deallocate memory
```

```
avltree::~avltree()
{
deltree(root);
}
//deletes the tree
void avltree::deltree(avlnode *root)
{
if(root!=NULL)
{
deltree(root->left);
deltree(root->right);
}
delete(root);
}
void main()
{
clrscr();
avltree at;
avlnode *avl=NULL;
int h;

avl=at.insert(20,&h);
at.setroot(avl);

avl=at.insert(6,&h);
at.setroot(avl);

avl=at.insert(29,&h);
at.setroot(avl);

avl=at.insert(5,&h);
at.setroot(avl);

avl=at.insert(12,&h);
at.setroot(avl);

avl=at.insert(25,&h);
at.setroot(avl);

avl=at.insert(32,&h);
at.setroot(avl);

avl=at.insert(10,&h);
at.setroot(avl);

avl=at.insert(15,&h);
at.setroot(avl);

avl=at.insert(27,&h);
at.setroot(avl);

avl=at.insert(13,&h);
at.setroot(avl);
```

```
cout<<endl<<"avl tree:\n";
at.display(avl);
avl=at.deldata(avl,20,&h);
at.setroot(avl);
avl=at.deldata(avl,12,&h);
at.setroot(avl);

cout<<endl<<"avl tree after deletion of node:\n";
at.display(avl);
getch();
}
```

Write a program to implement the different operations of a threaded binary tree.

```
///threaded binary tree//
#include<iostream.h>
#include<conio.h>
#include<stdio.h>

enum boolean
{
false=0,
true=1
};
class ttree
{
private:
struct thtree
{
enum boolean left;
thtree *leftchild;
int data;
thtree *rightchild;
enum boolean right;
}*th_head;
public:
ttree();
void insert(int num);
void remove(int num);
static void search(thtree **sr,int num,thtree **par,thtree **x,int *found);
void inorder();
~ttree();
};

//initialises data member
ttree::ttree()
{
th_head=NULL;
}
```

```
//insert a node in a threaded binary tree
void ttree::insert(int num)
{
thtree *head=th_head,*p,*z;
//allocate a new node
z=new thtree;
z->left=true; //indicates a thread
z->data=num;
z->right=true;

//if tree is empty
if(th_head==NULL)
{
head=new thtree;
//the entire tree is treated as a left subtree of the head node
head->left=false;
//z becomes leftchild of the head node
head->leftchild=z;
head->data=-9999;
head->rightchild=head;
//right link will alwaz be pointing to itself
head->right=false;
th_head=head;
//left thread to head
z->leftchild=head;
//right thread to head
z->rightchild=head;
}
else //if tree is non-empty
{
p=head->leftchild;
//traverse till the threa is found attached to the head
while(p!=head)
{
if(p->data>num)
{
//checking for a thread
if(p->left!=true)
p=p->leftchild;
else
{
z->leftchild=p->leftchild;
p->leftchild=z;
//indicates a link
p->left=false;
z->right=true;
z->rightchild=p;
return;
}
}
else
{
if(p->data<num)
```

```
{
if(p->right!=true)
p=p->rightchild;
else
{
z->rightchild=p->rightchild;
p->rightchild=z;
//indicates a link
p->right=false;
z->left=true;
z->leftchild=p;
return;
}
}
}
}
}
}
//delete a node from binary searxh tree
void ttree::remove(int num)
{
int found;
thtree *parent,*x,*xsucc;
//fi tree is empty
if(th_head==NULL)
{
cout<<"\ntree is empty";
return;
}
parent=x=NULL;
//call to search function to find
//the node to be deleted
search(&th_head,num,&parent,&x,&found);
//if the node to deleted is not found
if(found==false)
{
cout<<"\ndata to be deleted,not found";
return;
}
//if the node to be deleted has two children
if(x->left==false&&x->right==false)
{
parent=x;
xsucc=x->rightchild;
while(xsucc->left==false)
{
parent=xsucc;
xsucc=xsucc->leftchild;
}
x->data=xsucc->data;
x=xsucc;
}
//if the node to be deleted has no child
```

```
if(x->left==true&&x->right==true)
{
//if node to be deleted is a root node
if(parent==NULL)
{
th_head->leftchild=th_head;
th_head->left=true;
delete x;
return;
}
if(parent->rightchild==x)
{
parent->right=true;
parent->rightchild=x->rightchild;
}
else
{
parent->left=true;
parent->leftchild=x->leftchild;
}
delete x;
return;
}
//if the node to be deleted has only right child
if(x->left==true&&x->right==false)
{
//node to be deleted is a root node
if(parent==NULL)
{
th_head->leftchild=x->rightchild;
delete x;
return;
}
if(parent->leftchild==x)
{
parent->leftchild=x->rightchild;
x->rightchild->leftchild=x->leftchild;
}
else
{
parent->rightchild=x->rightchild;
x->rightchild->leftchild=parent;
}
delete x;
return;
}
//if the node to be delted has only left child
if(x->left==false&&x->right==true)
{
//the node to be deleted is root node
if(parent==NULL)
{
parent=x;
```

```
xsucc=x->leftchild;
while(xsucc->right==false)
xsucc=xsucc->rightchild;

xsucc->rightchild=th_head;
th_head->leftchild=x->leftchild;
delete x;
return;
}
if(parent->leftchild==x)
{
parent->leftchild=x->leftchild;
x->leftchild->rightchild=parent;
}
else
{
parent->rightchild=x->leftchild;
x->leftchild->rightchild=x->rightchild;
}
delete x;
return;
}
}
//return the address of the node to be deleted address of its parent and
//whether the node is found or not
void ttree::search(thtree **root,int num,thtree **par,thtree **x,int *found)
{
thtree *q;
q=(*root)->leftchild;
*found=false;
*par=NULL;
while(q!=root)
{
//if the node to be deleted is found
if(q->data==num)
{
*found=true;
*x=q;
return;
}
*par=q;
if(q->data>num)
{
if(q->left==true)
{
*found=false;
x=NULL;
return;
}
q=q->leftchild;
}
else
{
```

```
if(q->right==true)
{
*found=false;
*x=NULL;
return;
}
q=q->rightchild;
}
}
}
//traverse the threaded binary tree in order
void ttree::inorder()
{
thtree *p;
p=th_head->leftchild;
while(p!=th_head)
{
while(p->left==false)
p=p->leftchild;
cout<<p->data<<"\t";
while(p->right==true)
{
p=p->rightchild;
if(p==th_head)
break;
cout<<p->data<<"\t";
}
p=p->rightchild;
}
}
ttree::~ttree()
{
while(th_head->leftchild!=th_head)
remove(th_head->leftchild->data);
}
void main()
{
ttree th;
th.insert(11);
th.insert(9);
th.insert(13);
th.insert(8);
th.insert(10);
th.insert(12);
th.insert(14);
th.insert(15);
th.insert(7);
cout<<"threaded binary tree before deletion:\n";
th.inorder();

th.remove(10);
cout<<"\nthreaded binary tree after deletion:\n";
th.inorder();
```

```
th.remove(14);
cout<<"\nthreaded binary tree after deltion:\n";
th.inorder();

th.remove(8);
cout<<"\nthreaded binary tree after deletion:\n";
th.inorder();

th.remove(13);
cout<<"\nthreaded binary tree after deltion:\n";
th.inorder();
getch();
}
```

Program to implement the different operations of a M-way search tree

```
/// m ordered btree///
#include<iostream.h>
#include<stdio.h>
#include<conio.h>

const int max=4;
const int min=2;

struct btnode
{
int count;
int value[max+1];
btnode *child[max+1];
};

class btree
{
private:
btnode *root;
public:
btree();
void insert(int val);
int setval(int val,btnode *n,int *p,btnode **c);
static btnode *search(int val,btnode *root,int *pos);
static int searchnode(int val,btnode *n,int *pos);
void fillnode(int val,btnode *c,btnode *n,int k);
void split(int val,btnode *c,btnode *n,int k,int *y,btnode **newnode);
void del(int val);
int delhelp(int val,btnode *root);
void clear(btnode *root,int k);
void copysucc(btnode *root,int i);
void restore(btnode *root,int i);
void rightshift(int k);
void leftshift(int k);
```

```
void merge(int k);
void show();
static void display(btnode *root);
static void deltree(btnode *root);
~btree();
};

//initialise data member
btree::btree()
{
root=NULL;
}
//inserts a value in the btree
void btree::insert(int val)
{
int i;
btnode *c,*n;
int flag;
flag=setval(val,root,&i,&c);
if(flag)
{
n=new btnode;
n->count=1;
n->value[1]=i;
n->child[0]=root;
n->child[1]=c;
root=n;
}
}
//sets the value in the node
int btree::setval(int val,btnode *n,int *p,btnode **c)
{
int k;
if(n==NULL)
{
*p=val;
*c=NULL;
return 1;
}
else
{
if(searchnode(val,n,&k))
cout<<endl<<"key value already exists"<<endl;
if(setval(val,n->child[k],p,c))
{
if(n->count<max)
{
fillnode(*p,*c,n,k);
return 0;
}
else
{
split(*p,*c,n,k,p,c);
```

```
return 1;
}
}
return 0;
}
}
//searches value in the node
btnode *btree::search(int val,btnode *root,int *pos)
{
if(root==NULL)
return NULL;
else
{
if(searchnode(val,root,pos))
return root;
else
return search(val,root->child[*pos],pos);
}
}
//searches for node
int btree::searchnode(int val,btnode *n,int *pos)
{
if(val<n->value[1])
{
*pos=0;
return 0;
}
else
{
*pos=n->count;
while((val<n->value[*pos])&& *pos>1)
(*pos)--;
if(val==n->value[*pos])
return 1;
else
return 0;
}
}
//adjust the value of node
void btree::fillnode(int val,btnode *c,btnode *n,int k)
{
int i;
for(i=n->count;i>k;i--)
{
n->value[i+1]=n->value[i];
n->child[i+1]=n->child[i];
}
n->value[k+1]=val;
n->child[k+1]=c;
n->count++;
}
//splits the node
void btree::split(int val,btnode *c,btnode *n,int k,int *y,btnode **newnode)
```

```
{
int i,mid;
if(k<=min)
mid=min;
else
mid=min+1;
*newnode=new btnode;
for(i=mid+1;i<=max;i++)
{
(*newnode)->value[i-mid]=n->value[i];
(*newnode)->child[i-mid]=n->child[i];
}
(*newnode)->count=max-mid;
n->count=mid;
if(k<=min)
fillnode(val,c,n,k);
else
fillnode(val,c,*newnode,k-mid);
*y=n->value[n->count];
(*newnode)->child[0]=n->child[n->count];
n->count--;
}

//delete value from the node
void btree::del(int val)
{
btnode *temp;
if(!delhelp(val,root))
cout<<endl<<"value"<<val<<"not found";
else
{
if(root->count==0)
{
temp=root;
root=root->child[0];
delete temp;
}
}
}
//helper function for del()
int btree::delhelp(int val,btnode *root)
{
int i;
int flag;
if(root==NULL)
return 0;
else
{
flag=searchnode(val,root,&i);
if(flag)
{
if(root->child[i-1])
{
```

```
copysucc(root,i);
flag=delhelp(root->value[i],root->child[i]);
if(!flag)
cout<<endl<<"value"<<val<<"not found";
}
else
clear(root,i);
}
else
flag=delhelp(val,root->child[i]);
if(root->child[i]!=NULL)
{
if(root->child[i]->count<min)
restore(root,i);
}
return flag;
}
}
//removes the value from the node and adjusts the values
void btree::clear(btnode *root,int k)
{
int i;
for(i=k+1;i<=root->count;i++)
{
root->value[i-1]=root->value[i];
root->child[i-1]=root->child[i];
}
root->count--;
}
//copies the successor of the value that is to be deleted
void btree::copysucc(btnode *root,int i)
{
btnode *temp=root->child[i];
while(temp->child[0])
temp=temp->child[0];
root->value[i]=temp->value[i];
}
//adjust the node
void btree::restore(btnode *root,int i)
{
if(i==0)
{
if(root->child[1]->count>min)
leftshift(1);
else
merge(1);
}
else
{
if(i==root->count)
{
if(root->child[i-1]->count>min)
rightshift(i);
```

```
else
merge(i);
}
else
{
if(root->child[i-1]->count>min)
leftshift(i+1);
else
{
if(root->child[i+1]->count>min)
leftshift(i+1);
else
merge(i);
}
}
}
}
//adjust the values and children while shifting the value from parent to
right child
void btree::rightshift(int k)
{
int i;
btnode *temp;
temp=root->child[k];
for(i=temp->count;i>0;i--)
{
temp->value[i+1]=temp->value[i];
temp->child[i+1]=temp->child[i];
}
temp->child[1]=temp->child[0];
temp->count++;
temp->value[1]=root->value[k];
temp=root->child[k-1];
root->value[k]=temp->value[temp->count];
root->child[k]->child[0]=temp->child[temp->count];
temp->count--;
}
//adjusts the values and children while shifting the value from parent to
left child
void btree::leftshift(int k)
{
btnode *temp;
temp=root->child[k-1];
temp->count++;
temp->value[temp->count]=root->value[k];
temp->child[temp->count]=root->child[k]->child[0];

temp=root->child[k];
root->value[k]=temp->value[1];
temp->child[0]=temp->child[1];
temp->count--;
for(int i=1;i<=temp->count;i++)
{
```

```
temp->value[i]=temp->value[i+1];
temp->child[i]=temp->child[i+1];
}
}
//merges two nodes
void btree::merge(int k)
{
btnode *temp1,*temp2;
temp1=root->child[k];
temp2=root->child[k-1];
temp2->count++;
temp2->value[temp2->count]=root->value[k];
temp2->child[temp2->count]=root->child[0];

for(int i=1;i<=temp1->count;i++)
{
temp2->count++;
temp2->value[temp2->count]=temp1->value[i];
temp2->child[temp2->count]=temp1->child[i];
}
for(i=k;i<root->count;i++)
{
root->value[i]=root->value[i+1];
root->child[i]=root->child[i+1];
}
root->count--;
delete temp1;
}
//calls display()
void btree::show()
{
display(root);
}
//display btree
void btree::display(btnode *root)
{
if(root!=NULL)
{
for(int i=0;i<root->count;i++)
{
display(root->child[i]);
cout<<root->value[i+1]<<"\t";
}
display(root->child[i]);
}
}
//deallocates memory
void btree::deltree(btnode *root)
{
if(root!=NULL)
{
for(int i=0;i<root->count;i++)
{
```

```
deltree(root->child[i]);
deltree(root->child[i]);
}
deltree(root->child[i]);
deltree(root->child[i]);
}
}
btree::~btree()
{
deltree(root);
}
void main()
{
clrscr();
btree b;
int arr[]={27,42,22,47,32,2,51,40,13};
int sz=sizeof(arr)/sizeof(int);
for(int i=0;i<sz;i++)
b.insert(arr[i]);
cout<<"btree of order 5:"<<endl;
b.show();

b.del(22);
b.del(11);

cout<<"\n\nb-tree deletion of values"<<endl;
b.show();
getch();
}
```

Week 8
Write a program in C++ to implement the different operations of a B+ tree? *ASSIGNMENT:* Write a program to implement the different operations of a B- tree?

Program in C++ to implement the different operations of a B+ tree

```
#include<iostream>
using namespace std;
struct BplusTree {
int *d;
BplusTree **child_ptr;
bool l;
int n;
}*r = NULL, *np = NULL, *x = NULL;
BplusTree* init()//to create nodes {
int i;
np = new BplusTree;
np->d = new int[6];//order 6
```

```
np->child_ptr = new BplusTree *[7];
np->l = true;
np->n = 0;
for (i = 0; i < 7; i++) {
np->child_ptr[i] = NULL;
}
return np;
}

void traverse(BplusTree *p)//traverse tree {
cout<<endl;
int i;
for (i = 0; i < p->n; i++) {
if (p->l == false) {
traverse(p->child_ptr[i]);
}
cout << " " << p->d[i];
}
if (p->l == false) {
traverse(p->child_ptr[i]);
}
cout<<endl;
}

void sort(int *p, int n)//sort the tree {
int i, j, t;
for (i = 0; i < n; i++) {
for (j = i; j <= n; j++) {
if (p[i] >p[j]) {
t = p[i];
p[i] = p[j];
p[j] = t;
}
}
}
}

int split_child(BplusTree *x, int i) {
int j, mid;
BplusTree *np1, *np3, *y;
np3 = init();
np3->l = true;
if (i == -1) {
mid = x->d[2];
x->d[2] = 0;
x->n--;
np1 = init();
np1->l = false;
x->l = true;
for (j = 3; j < 6; j++) {
np3->d[j - 3] = x->d[j];
np3->child_ptr[j - 3] = x->child_ptr[j];
np3->n++;
```

```
x->d[j] = 0;
x->n--;
}
for (j = 0; j < 6; j++) {
x->child_ptr[j] = NULL;
}
np1->d[0] = mid;
np1->child_ptr[np1->n] = x;
np1->child_ptr[np1->n + 1] = np3;
np1->n++;
r = np1;
} else {
y = x->child_ptr[i];
mid = y->d[2];
y->d[2] = 0;
y->n--;
for (j = 3; j <6 ; j++) {
np3->d[j - 3] = y->d[j];
np3->n++;
y->d[j] = 0;
y->n--;
}
x->child_ptr[i + 1] = y;
x->child_ptr[i + 1] = np3;
}
return mid;
}

void insert(int a) {
int i, t;
x = r;
if (x == NULL) {
r = init();
x = r;
} else {
if (x->l== true && x->n == 6) {
t = split_child(x, -1);
x = r;
for (i = 0; i < (x->n); i++) {
if ((a >x->d[i]) && (a < x->d[i + 1])) {
i++;
break;
} else if (a < x->d[0]) {
break;
} else {
continue;
}
}
x = x->child_ptr[i];
} else {
while (x->l == false) {
for (i = 0; i < (x->n); i++) {
if ((a >x->d[i]) && (a < x->d[i + 1])) {
```

```
i++;
break;
} else if (a < x->d[0]) {
break;
} else {
continue;
}
}
if ((x->child_ptr[i])->n == 6) {
t = split_child(x, i);
x->d[x->n] = t;
x->n++;
continue;
} else {
x = x->child_ptr[i];
}
}
}
}
x->d[x->n] = a;
sort(x->d, x->n);
x->n++;
}

int main() {
int i, n, t;
cout<<"enter the no of elements to be inserted\n";
cin>>n;
for(i = 0; i < n; i++) {
cout<<"enter the element\n";
cin>>t;
insert(t);
}
cout<<"traversal of constructed B tree\n";
traverse(r);
}
```

Week 9
Write a program in C++ to implement the different operations of a B* tree? Write a program in C++ to Multi-dimensional binary search trees

Program in C++ to implement the different operations of a B* tree

```
// C++ program to implement B* tree

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

// This can be changed to any value -
// it is the order of the B* Tree
#define N 4

struct node {

// key of N-1 nodes
int key[N - 1];

// Child array of 'N' length
struct node* child[N];

// To state whether a leaf or not; if node
// is a leaf, isleaf=1 else isleaf=0
int isleaf;

// Counts the number of filled keys in a node
int n;

// Keeps track of the parent node
struct node* parent;
};

// This function searches for the leaf
// into which to insert element 'k'
struct node* searchforleaf(struct node* root, int k,
struct node* parent, int chindex)
{
if (root) {

// If the passed root is a leaf node, then
// k can be inserted in this node itself
if (root->isleaf == 1)
return root;

// If the passed root is not a leaf node,
// implying there are one or more children
else {
int i;
```

```
/*If passed root's initial key is itself g
reater than the element to be inserted,
we need to insert to a new leaf left of the root*/
if (k < root->key[0])
root = searchforleaf(root->child[0], k, root, 0);

else
{
// Find the first key whose value is greater
// than the insertion value
// and insert into child of that key
for (i = 0; i < root->n; i++)
if (root->key[i] > k)
root = searchforleaf(root->child[i], k, root, i);

// If all the keys are less than the insertion
// key value, insert to the right of last key
if (root->key[i - 1] < k)
root = searchforleaf(root->child[i], k, root, i);
}
}
}
else {

// If the passed root is NULL (there is no such
// child node to search), then create a new leaf
// node in that location
struct node* newleaf = new struct node;
newleaf->isleaf = 1;
newleaf->n = 0;
parent->child[chindex] = newleaf;
newleaf->parent = parent;
return newleaf;
}
}

struct node* insert(struct node* root, int k)
{
if (root) {
struct node* p = searchforleaf(root, k, NULL, 0);
struct node* q = NULL;
int e = k;

// If the leaf node is empty, simply
// add the element and return
for (int e = k; p; p = p->parent) {
if (p->n == 0) {
p->key[0] = e;
p->n = 1;
return root;
}
// If number of filled keys is less than maximum
```

```
if (p->n < N - 1) {
int i;
for (i = 0; i < p->n; i++) {
if (p->key[i] > e) {
for (int j = p->n - 1; j >= i; j--)
p->key[j + 1] = p->key[j];
break;
}
}
p->key[i] = e;
p->n = p->n + 1;
return root;
}

// If number of filled keys is equal to maximum
// and it's not root and there is space in the parent
if (p->n == N - 1 && p->parent && p->parent->n < N) {
int m;
for (int i = 0; i < p->parent->n; i++)
if (p->parent->child[i] == p) {
m = i;
break;
}

// If right sibling is possible
if (m + 1 <= N - 1)
{
// q is the right sibling
q = p->parent->child[m + 1];

if (q) {

// If right sibling is full
if (q->n == N - 1) {
struct node* r = new struct node;
int* z = new int[((2 * N) / 3)];
int parent1, parent2;
int* marray = new int[2 * N];
int i;
for (i = 0; i < p->n; i++)
marray[i] = p->key[i];
int fege = i;
marray[i] = e;
marray[i + 1] = p->parent->key[m];
for (int j = i + 2; j < ((i + 2) + (q->n)); j++)
marray[j] = q->key[j - (i + 2)];

// marray=bubblesort(marray, 2*N)
// a more rigorous implementation will
// sort these elements

// Put first (2*N-2)/3 elements into keys of p
for (int i = 0; i < (2 * N - 2) / 3; i++)
```

```
p->key[i] = marray[i];
parent1 = marray[(2 * N - 2) / 3];

// Put next (2*N-1)/3 elements into keys of q
for (int j = ((2 * N - 2) / 3) + 1; j < (4 * N) / 3; j++)
q->key[j - ((2 * N - 2) / 3 + 1)] = marray[j];
parent2 = marray[(4 * N) / 3];

// Put last (2*N)/3 elements into keys of r
for (int f = ((4 * N) / 3 + 1); f < 2 * N; f++)
r->key[f - ((4 * N) / 3 + 1)] = marray[f];

// Because m=0 and m=1 are children of the same key,
// a special case is made for them
if (m == 0 || m == 1) {
p->parent->key[0] = parent1;
p->parent->key[1] = parent2;
p->parent->child[0] = p;
p->parent->child[1] = q;
p->parent->child[2] = r;
return root;
}

else {
p->parent->key[m - 1] = parent1;
p->parent->key[m] = parent2;
p->parent->child[m - 1] = p;
p->parent->child[m] = q;
p->parent->child[m + 1] = r;
return root;
}
}
}
else // If right sibling is not full
{
int put;
if (m == 0 || m == 1)
put = p->parent->key[0];
else
put = p->parent->key[m - 1];
for (int j = (q->n) - 1; j >= 1; j--)
q->key[j + 1] = q->key[j];
q->key[0] = put;
p->parent->key[m == 0 ? m : m - 1] = p->key[p->n - 1];
}
}
}
}

/*Cases of root splitting, etc. are omitted
as this implementation is just to demonstrate
the two-three split operation*/
}
```

```
else
{
// Create new node if root is NULL
struct node* root = new struct node;
root->key[0] = k;
root->isleaf = 1;
root->n = 1;
root->parent = NULL;
}
}

// Driver code
int main()
{
/* Consider the following tree that has been obtained
from some root split:
6
/ \
1 2 4 7 8 9

We wish to add 5. This makes the B*-tree:
4 7
/ \ \
1 2 5 6 8 9

Contrast this with the equivalent B-tree, in which
some nodes are less than half full

4 6
/ \ \
1 2 5 7 8 9

*/

// Start with an empty root
struct node* root = NULL;
// Insert 6
root = insert(root, 6);

// Insert 1, 2, 4 to the left of 6
root->child[0] = insert(root->child[0], 1);
root->child[0] = insert(root->child[0], 2);
root->child[0] = insert(root->child[0], 4);
root->child[0]->parent = root;


// Insert 7, 8, 9 to the right of 6
root->child[1] = insert(root->child[1], 7);
root->child[1] = insert(root->child[1], 8);
root->child[1] = insert(root->child[1], 9);
root->child[1]->parent = root;
```

```
cout << "Original tree: " << endl;
for (int i = 0; i < root->n; i++)
cout << root->key[i] << " ";
cout << endl;
for (int i = 0; i < 2; i++) {
cout << root->child[i]->key[0] << " ";
cout << root->child[i]->key[1] << " ";
cout << root->child[i]->key[2] << " ";
}
cout << endl;


cout << "After adding 5: " << endl;

// Inserting element '5':

root->child[0] = insert(root->child[0], 5);

// Printing nodes

for (int i = 0; i <= root->n; i++)
cout << root->key[i] << " ";
cout << endl;
for (int i = 0; i < N - 1; i++) {
cout << root->child[i]->key[0] << " ";
cout << root->child[i]->key[1] << " ";
}

return 0;
}
```

A program in C++ to Multi-dimensional binary search trees

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

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
int point[k]; // To store k dimensional point
Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
struct Node* temp = new Node;

for (int i=0; i<k; i++)
```

```
temp->point[i] = arr[i];

temp->left = temp->right = NULL;
return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
// Tree is empty?
if (root == NULL)
return newNode(point);

// Calculate current dimension (cd) of comparison
unsigned cd = depth % k;

// Compare the new point with root on current dimension 'cd'
// and decide the left or right subtree
if (point[cd] < (root->point[cd]))
root->left  = insertRec(root->left, point, depth + 1);
else
root->right = insertRec(root->right, point, depth + 1);

return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
return insertRec(root, point, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
// Compare individual pointinate values
for (int i = 0; i < k; ++i)
if (point1[i] != point2[i])
return false;

return true;
}

// Searches a Point represented by "point[]" in the K D tree.
// The parameter depth is used to determine current axis.
bool searchRec(Node* root, int point[], unsigned depth)
{
// Base cases
if (root == NULL)
```

```
return false;
if (arePointsSame(root->point, point))
return true;

// Current dimension is computed using current depth and total
// dimensions (k)
unsigned cd = depth % k;

// Compare point with root with respect to cd (Current dimension)
if (point[cd] < root->point[cd])
return searchRec(root->left, point, depth + 1);

return searchRec(root->right, point, depth + 1);
}

// Searches a Point in the K D tree. It mainly uses
// searchRec()
bool search(Node* root, int point[])
{
// Pass current depth as 0
return searchRec(root, point, 0);
}

// Driver program to test above functions
int main()
{
struct Node *root = NULL;
int points[][k] = {{3, 6}, {17, 15}, {13, 15}, {6, 12},
{9, 1}, {2, 7}, {10, 19}};

int n = sizeof(points)/sizeof(points[0]);

for (int i=0; i<n; i++)
root = insert(root, points[i]);

int point1[] = {10, 19};
(search(root, point1))? cout << "Found\n": cout << "Not Found\n";

int point2[] = {12, 19};
(search(root, point2))? cout << "Found\n": cout << "Not Found\n";

return 0;
}
```

Week 10
Write a program in C++ to find the edges of a spanning tree using Prims Algorithm? Write a C++ program to illustrate the traversal of a graph using Depth First Search? Write a C++ program to illustrate the traversal of a graph using Breadth First Search? *ASSIGNMENT*: Write a program in C++ to implement the graph using different representations?

A program in C++ to find the edges of a spanning tree using Prims Algorithm

```
//A C++ program for Prim's Minimum Spanning Tree (MST) algorithm. The
//program is for adjacency matrix representation of the graph

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

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
// Initialize min value
int min = INT_MAX, min_index;

for (int v = 0; v < V; v++)
if (mstSet[v] == false && key[v] < min)
min = key[v], min_index = v;

return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
cout<<"Edge \tWeight\n";
for (int i = 1; i < V; i++)
cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
```

```
void primMST(int graph[V][V])
{
// Array to store constructed MST
int parent[V];

// Key values used to pick minimum weight edge in cut
int key[V];

// To represent set of vertices not yet included in MST
bool mstSet[V];

// Initialize all keys as INFINITE
for (int i = 0; i < V; i++)
key[i] = INT_MAX, mstSet[i] = false;

// Always include first 1st vertex in MST.
// Make key 0 so that this vertex is picked as first vertex.
key[0] = 0;
parent[0] = -1; // First node is always root of MST

// The MST will have V vertices
for (int count = 0; count < V - 1; count++)
{
// Pick the minimum key vertex from the
// set of vertices not yet included in MST
int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set
mstSet[u] = true;

// Update key value and parent index of
// the adjacent vertices of the picked vertex.
// Consider only those vertices which are not
// yet included in MST
for (int v = 0; v < V; v++)

// graph[u][v] is non zero only for adjacent vertices of m
// mstSet[v] is false for vertices not yet included in MST
// Update the key only if graph[u][v] is smaller than key[v]
if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
parent[v] = u, key[v] = graph[u][v];
}

// print the constructed MST
printMST(parent, graph);
}

int main()
{
/* Let us create the following graph
2 3
(0)--(1)--(2)
| / \ |
```

```
6| 8/ \5 |7
| / \ |
(3)-------(4)
9     */
int graph[V][V] = { { 0, 2, 0, 6, 0 },
{ 2, 0, 3, 8, 5 },
{ 0, 3, 0, 0, 7 },
{ 6, 8, 0, 0, 9 },
{ 0, 5, 7, 9, 0 } };

// Print the solution
primMST(graph);

return 0;
}
```

A C++ program to illustrate the traversal of a graph using Depth First Search?

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>

#define true 1
#define false 0
const int max=8;

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

class graph
{
private:
int visited[max];

public:
graph();
void dfs(int v,node **p);
node* getnode_write(int val);
void del(node *n);
};
//initialise data members
graph::graph()
{
for(int i=0;i<max;i++)
visited[i]=false;
}
//function that has implemented dfs algo
```

```
void graph::dfs(int v,node **p)
{
node *t;
visited[v-1]=true;
cout<<v<<"\t";
t=*(p+v-1);
while(t!=NULL)
{
if(visited[t->data-1]==false)
dfs(t->data,p);
else
t=t->next;
}
}
//creates a node
node* graph::getnode_write(int val)
{
node *newnode=new node;
newnode->data=val;
return newnode;
}
//deallocate memory
void graph::del(node *n)
{
node *temp;
while(n!=NULL)
{
temp=n->next;
delete n;
n=temp;
}
}
void main()
{
node *arr[max];
node *v1,*v2,*v3,*v4;
graph g;
v1=g.getnode_write(2);
arr[0]=v1;
v1->next=v2=g.getnode_write(8);
v2->next=NULL;
v1=g.getnode_write(1);
arr[1]=v1;
v1->next=v2=g.getnode_write(3);
v2->next=v3=g.getnode_write(8);
v3->next=NULL;

v1=g.getnode_write(2);
arr[2]=v1;
v1->next=v2=g.getnode_write(4);
v2->next=v3=g.getnode_write(8);
v3->next=NULL;
```

```
v1=g.getnode_write(3);
arr[3]=v1;
v1->next=v2=g.getnode_write(7);
v2->next=NULL;

v1=g.getnode_write(6);
arr[4]=v1;
v1->next=v2=g.getnode_write(7);
v2->next=NULL;

v1=g.getnode_write(5);
arr[5]=v1;
v2->next=NULL;

v1=g.getnode_write(4);
arr[6]=v1;
v1->next=v2=g.getnode_write(5);
v2->next=v3=g.getnode_write(8);
v3->next=NULL;

v1=g.getnode_write(1);
arr[7]=v1;
v1->next=v2=g.getnode_write(2);
v2->next=v3=g.getnode_write(3);
v3->next=v4=g.getnode_write(7);
v4->next=NULL;

cout<<endl;
g.dfs(1,arr);
for(int i=0;i<max;i++)
g.del(arr[i]);
getch();
}
```

A C++ program to illustrate the traversal of a graph using Breadth First Search

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>

#define true 1
#define false 0

const int max=8;
struct node
{
int data;
node *next;
};

class graph
```

```
{
private:
int visited[max];
int q[8];
int front,rear;
public:
graph();
void bfs(int v,node **p);
node* getnode_write(int val);
static void addqueue(int *a,int vertex,int *f,int *r);
static int deletequeue(int *q,int *f,int *r);
static int isempty(int *f);
void del(node *n);
};
//initialise data members
graph::graph()
{
for(int i=0;i<max;i++)
visited[i]=false;
front=rear=-1;
}
//function that implements bfs algo
void graph::bfs(int v,node **p)
{
node *u;
visited[v-1]=true;
cout<<v<<"\t";
addqueue(q,v,&front,&rear);
while(isempty(&front)==false)
{
v=deletequeue(q,&front,&rear);
u=*(p+v-1);

while(u!=NULL)
{
if(visited[u->data-1]==false)
{
addqueue(q,u->data,&front,&rear);
visited[u->data-1]=true;
cout<<u->data<<"\t";
}
u=u->next;
}
}
}
//create a node
node* graph::getnode_write(int val)
{
node *newnode=new node;
newnode->data=val;
return newnode;
}
//adds node to the queue
```

```
void graph::addqueue(int *a,int vertex,int *f,int *r)
{
if(*r==max-1)
{
cout<<"\nqueue overflow";
//return(0);
}
(*r)++;
a[*r]=vertex;
if(*f==-1)
*f=0;
}
//deletes a node from the queue
int graph::deletequeue(int *a,int *f,int *r)
{
int data;
if(*f==-1)
{
cout<<"\nqueue underflow";
return(0);
}
data=a[*f];
if(*f==*r)
*f=*r=-1;
else
(*f)++;
return data;
}
//checks if queue is empty
int graph::isempty(int *f)
{
if(*f==-1)
return true;
return false;
}
//deallocate memory
void graph::del(node *n)
{
node *temp;
while(n!=NULL)
{
temp=n->next;
delete n;
n=temp;
}
}
void main()
{
node *arr[max];
node *v1,*v2,*v3,*v4;
graph g;
v1=g.getnode_write(2);
arr[0]=v1;
```

```
v1->next=v2=g.getnode_write(3);
v2->next=NULL;
v1=g.getnode_write(1);
arr[1]=v1;
v1->next=v2=g.getnode_write(4);
v2->next=v3=g.getnode_write(5);
v3->next=NULL;
v1=g.getnode_write(1);
arr[2]=v1;
v1->next=v2=g.getnode_write(6);
v2->next=v3=g.getnode_write(7);
v3->next=NULL;

v1=g.getnode_write(2);
arr[3]=v1;
v1->next=v2=g.getnode_write(8);
v2->next=NULL;

v1=g.getnode_write(2);
arr[4]=v1;
v1->next=v2=g.getnode_write(8);
v2->next=NULL;

v1=g.getnode_write(3);
arr[5]=v1;
v1->next=v2=g.getnode_write(8);
v2->next=NULL;

v1=g.getnode_write(3);
arr[6]=v1;
v1->next=v2=g.getnode_write(8);
v2->next=NULL;

v1=g.getnode_write(4);
arr[7]=v1;
v1->next=v2=g.getnode_write(5);
v2->next=v3=g.getnode_write(6);
v3->next=v4=g.getnode_write(7);
v4->next=NULL;
cout<<endl;
g.bfs(1,arr);
for(int i=0;i<max;i++)
g.del(arr[i]);
getch();
}
```

Week 11
Write a C++ program to in C++ to find the shortest path in a graph using Dijkstra's Algorithm. Write a C++ program in C++ to implement Euler Graphs? Write a program in C++ to find the shortest path in a graph using Warshalls Algorithm. *ASSIGNMENT*: Write a C++ program to in C++ to find the shortest path in a graph using Modified Warshalls Algorithm.

A C++ program to find the shortest path in a graph using Dijkstra's Algorithm.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>

const int inf=9999;
void main()
{
int arr[4][4];
int cost[4][4]={ 7,5,0,0,7,0,0,2,0,3,0,0,4,0,1,0};
int i,j,k,n=4;

for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if(cost[i][j]==0)
arr[i][j]=inf;
else
arr[i][j]=cost[i][j];
}
}
cout<<"adjency matrix of cost of edges:\n";
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
cout<<arr[i][j]<<"\t";
cout<<"\n";
}
for(k=0;k<n;k++)
{
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if(arr[i][j]>arr[i][k]+arr[k][j])
arr[i][j]=arr[i][k]+arr[k][j];
}
}
}
cout<<"adjacency matrix of lowest cost between the vertices:\n";
for(i=0;i<n;i++)
{
```

```
for(j=0;j<n;j++)
cout<<arr[i][j]<<"\t";
cout<<"\n";
}
getch();
}
```

A C++ program to implement Euler Graphs

```
#include<iostream>
#include <list>

using namespace std;

// A class that represents an undirected graph
class Graph
{
int V;    // No. of vertices
list<int> *adj;    // A dynamic array of adjacency lists
public:
// Constructor and destructor
Graph(int V)   {this->V = V; adj = new list<int>[V]; }
~Graph() { delete [] adj; } // To avoid memory leak

// function to add an edge to graph
void addEdge(int v, int w);

// Method to check if this graph is Eulerian or not
int isEulerian();

// Method to check if all non-zero degree vertices are connected
bool isConnected();

// Function to do DFS starting from v. Used in isConnected();
void DFSUtil(int v, bool visited[]);
};

void Graph::addEdge(int v, int w)
{
adj[v].push_back(w);
adj[w].push_back(v);  // Note: the graph is undirected
}

void Graph::DFSUtil(int v, bool visited[])
{
// Mark the current node as visited and print it
visited[v] = true;

// Recur for all the vertices adjacent to this vertex
list<int>::iterator i;
for (i = adj[v].begin(); i != adj[v].end(); ++i)
if (!visited[*i])
DFSUtil(*i, visited);
```

```
}

// Method to check if all non-zero degree vertices are connected.
// It mainly does DFS traversal starting from
bool Graph::isConnected()
{
// Mark all the vertices as not visited
bool visited[V];
int i;
for (i = 0; i < V; i++)
visited[i] = false;

// Find a vertex with non-zero degree
for (i = 0; i < V; i++)
if (adj[i].size() != 0)
break;

// If there are no edges in the graph, return true
if (i == V)
return true;

// Start DFS traversal from a vertex with non-zero degree
DFSUtil(i, visited);

// Check if all non-zero degree vertices are visited
for (i = 0; i < V; i++)
if (visited[i] == false && adj[i].size() > 0)
return false;

return true;
}

/* The function returns one of the following values
0 --> If grpah is not Eulerian
1 --> If graph has an Euler path (Semi-Eulerian)
2 --> If graph has an Euler Circuit (Eulerian)  */
int Graph::isEulerian()
{
// Check if all non-zero degree vertices are connected
if (isConnected() == false)
return 0;

// Count vertices with odd degree
int odd = 0;
for (int i = 0; i < V; i++)
if (adj[i].size() & 1)
odd++;

// If count is more than 2, then graph is not Eulerian
if (odd > 2)
return 0;

// If odd count is 2, then semi-eulerian.
```

```
// If odd count is 0, then eulerian
// Note that odd count can never be 1 for undirected graph
return (odd)? 1 : 2;
}

// Function to run test cases
void test(Graph &g)
{
int res = g.isEulerian();
if (res == 0)
cout << "graph is not Eulerian\n";
else if (res == 1)
cout << "graph has a Euler path\n";
else
cout << "graph has a Euler cycle\n";
}

// Driver program to test above function
int main()
{
// Let us create and test graphs shown in above figures
Graph g1(5);
g1.addEdge(1, 0);
g1.addEdge(0, 2);
g1.addEdge(2, 1);
g1.addEdge(0, 3);
g1.addEdge(3, 4);
test(g1);

Graph g2(5);
g2.addEdge(1, 0);
g2.addEdge(0, 2);
g2.addEdge(2, 1);
g2.addEdge(0, 3);
g2.addEdge(3, 4);
g2.addEdge(4, 0);
test(g2);

Graph g3(5);
g3.addEdge(1, 0);
g3.addEdge(0, 2);
g3.addEdge(2, 1);
g3.addEdge(0, 3);
g3.addEdge(3, 4);
g3.addEdge(1, 3);
test(g3);

// Let us create a graph with 3 vertices
// connected in the form of cycle
Graph g4(3);
g4.addEdge(0, 1);
g4.addEdge(1, 2);
g4.addEdge(2, 0);
```

```
test(g4);

// Let us create a graph with all veritces
// with zero degree
Graph g5(3);
test(g5);

return 0;
}
```

A program in C++ to find the shortest path in a graph using Warshalls Algorithm

```
//C++ Program for Floyd Warshall Algorithm

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

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough
value.This value will be used for
vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm

void floydWarshall (int graph[][V])
{
/* dist[][] will be the output matrix
that will finally have the shortest
distances between every pair of vertices */

int dist[V][V], i, j, k;

/* Initialize the solution matrix same
as input graph matrix. Or we can say
the initial values of shortest distances
are based on shortest paths considering
no intermediate vertex. */

for (i = 0; i < V; i++)
for (j = 0; j < V; j++)
dist[i][j] = graph[i][j];

/* Add all vertices one by one to
the set of intermediate vertices.
---> Before start of an iteration,
```

```
we have shortest distances between all
pairs of vertices such that the
shortest distances consider only the
vertices in set {0, 1, 2, .. k-1} as
intermediate vertices.
----> After the end of an iteration,
vertex no. k is added to the set of
intermediate vertices and the set becomes {0, 1, 2, .. k} */

for (k = 0; k < V; k++)
{
// Pick all vertices as source one by one
for (i = 0; i < V; i++)
{
// Pick all vertices as destination for the
// above picked source
for (j = 0; j < V; j++)
{
// If vertex k is on the shortest path from
// i to j, then update the value of dist[i][j]
if (dist[i][k] + dist[k][j] < dist[i][j])
dist[i][j] = dist[i][k] + dist[k][j];
}
}
}

// Print the shortest distance matrix
printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
cout<<"The following matrix shows the shortest distances"
" between every pair of vertices \n";
for (int i = 0; i < V; i++)
{
for (int j = 0; j < V; j++)
{
if (dist[i][j] == INF)
cout<<"INF"<<"     ";
else
cout<<dist[i][j]<<"     ";
}
cout<<endl;
}
}


int main()
{
/* Let us create the following weighted graph
10
```

```
(0)------->(3)
|     /|\
5 |     |
|     | 1
\|/     |
(1)------->(2)
3     */
int graph[V][V] = { {0, 5, INF, 10},
{INF, 0, 3, INF},
{INF, INF, 0, 1},
{INF, INF, INF, 0}
};

// Print the solution
floydWarshall(graph);
return 0;
}
```

Week 12
Write a program in C++ to implement Hamilton Graphs? Write a program to C++ to implement Kruskals Algorithm? Write a program to C++ to find the cycles in a graph? *ASSIGNMENT:* Write a program in C++ to implement Planner Graphs?

A program in C++ to implement Hamilton Graphs

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

// Number of vertices in the graph
#define V 5

void printSolution(int path[]);

/* A utility function to check if
the vertex v can be added at index 'pos'
in the Hamiltonian Cycle constructed
so far (stored in 'path[]') */

bool isSafe(int v, bool graph[V][V],
int path[], int pos)
{
/* Check if this vertex is an adjacent
vertex of the previously added vertex. */
if (graph [path[pos - 1]][ v ] == 0)
```

```
return false;

/* Check if the vertex has already been included.
This step can be optimized by creating
an array of size V */
for (int i = 0; i < pos; i++)
if (path[i] == v)
return false;

return true;
}

/* A recursive utility function
to solve hamiltonian cycle problem */

bool hamCycleUtil(bool graph[V][V],
int path[], int pos)
{
/* base case: If all vertices are
included in Hamiltonian Cycle */
if (pos == V)
{
// And if there is an edge from the
// last included vertex to the first vertex
if (graph[path[pos - 1]][path[0]] == 1)
return true;
else
return false;
}

// Try different vertices as a next candidate
// in Hamiltonian Cycle. We don't try for 0 as
// we included 0 as starting point in hamCycle()
for (int v = 1; v < V; v++)
{
/* Check if this vertex can be added
// to Hamiltonian Cycle */
if (isSafe(v, graph, path, pos))
{
path[pos] = v;

/* recur to construct rest of the path */
if (hamCycleUtil (graph, path, pos + 1) == true)
return true;

/* If adding vertex v doesn't lead to a solution,
then remove it */
path[pos] = -1;
}
}

/* If no vertex can be added to
Hamiltonian Cycle constructed so far,
```

```
then return false */
return false;
}

/* This function solves the Hamiltonian Cycle problem
using Backtracking. It mainly uses hamCycleUtil() to
solve the problem. It returns false if there is no
Hamiltonian Cycle possible, otherwise return true
and prints the path. Please note that there may be
more than one solutions, this function prints one
of the feasible solutions. */
bool hamCycle(bool graph[V][V])
{
int *path = new int[V];
for (int i = 0; i < V; i++)
path[i] = -1;

/* Let us put vertex 0 as the first vertex in the path.
If there is a Hamiltonian Cycle, then the path can be
started from any point of the cycle as the graph is undirected */
path[0] = 0;
if (hamCycleUtil(graph, path, 1) == false )
{
cout << "\nSolution does not exist";
return false;
}

printSolution(path);
return true;
}

/* A utility function to print solution */
void printSolution(int path[])
{
cout << "Solution Exists:"
" Following is one Hamiltonian Cycle \n";
for (int i = 0; i < V; i++)
cout << path[i] << " ";

// Let us print the first vertex again
// to show the complete cycle
cout << path[0] << " ";
cout << endl;
}

// Driver Code
int main()
{
/* Let us create the following graph
(0)--(1)--(2)
| / \ |
| / \ |
| / \ |
```

```
(3)-------(4) */
bool graph1[V][V] = {{0, 1, 0, 1, 0},
{1, 0, 1, 1, 1},
{0, 1, 0, 0, 1},
{1, 1, 0, 0, 1},
{0, 1, 1, 1, 0}};

// Print the solution
hamCycle(graph1);

/* Let us create the following graph
(0)--(1)--(2)
| / \ |
| / \ |
| / \ |
(3) (4) */
bool graph2[V][V] = {{0, 1, 0, 1, 0},
{1, 0, 1, 1, 1},
{0, 1, 0, 0, 1},
{1, 1, 0, 0, 0},
{0, 1, 1, 0, 0}};

// Print the solution
hamCycle(graph2);

return 0;
}

*//
Output:
Solution Exists: Following is one Hamiltonian Cycle
0  1  2  4  3  0

Solution does not exist*/
```

A program to C++ to find the cycles in a graph

```
// A C++ Program to detect cycle in a graph
#include<iostream>
#include <list>
#include <limits.h>

using namespace std;

class Graph
{
int V;    // No. of vertices
list<int> *adj;    // Pointer to an array containing adjacency lists
bool isCyclicUtil(int v, bool visited[], bool *rs);  // used by isCyclic()
public:
Graph(int V);   // Constructor
void addEdge(int v, int w);   // to add an edge to graph
```

```
bool isCyclic();    // returns true if there is a cycle in this graph
};

Graph::Graph(int V)
{
this->V = V;
adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
adj[v].push_back(w); // Add w to v's list.
}

bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
if(visited[v] == false)
{
// Mark the current node as visited and part of recursion stack
visited[v] = true;
recStack[v] = true;

// Recur for all the vertices adjacent to this vertex
list<int>::iterator i;
for(i = adj[v].begin(); i != adj[v].end(); ++i)
{
if ( !visited[*i] && isCyclicUtil(*i, visited, recStack) )
return true;
else if (recStack[*i])
return true;
}

}
recStack[v] = false;  // remove the vertex from recursion stack
return false;
}

// Returns true if the graph contains a cycle, else false.

bool Graph::isCyclic()
{
// Mark all the vertices as not visited and not part of recursion
// stack
bool *visited = new bool[V];
bool *recStack = new bool[V];
for(int i = 0; i < V; i++)
{
visited[i] = false;
recStack[i] = false;
}

// Call the recursive helper function to detect cycle in different
// DFS trees
```

```
for(int i = 0; i < V; i++)
if (isCyclicUtil(i, visited, recStack))
return true;

return false;
}

int main()
{
// Create a graph given in the above diagram
Graph g(4);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 0);
g.addEdge(2, 3);
g.addEdge(3, 3);

if(g.isCyclic())
cout << "Graph contains cycle";
else
cout << "Graph doesn't contain cycle";
return 0;
}
```

A program to C++ to implement Kruskals Algorithm

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>

const int max=5;
struct lledge
{
int v1,v2;
float cost;
lledge *next;
};

int stree[max];
int count[max];
int mincost;

lledge* create(int cr1,int vr2,int cs);
lledge* kminstree(lledge *root,int n);
int getrval(int i);
void combine(int i,int j);
void del(lledge *root);

lledge* kminstree(lledge *root,int n)
```

```
{
lledge *temp=NULL;
lledge *p,*q;
int noofedges=0;
int i,p1,p2;

for(i=0;i<n;i++)
stree[i]=i;
for(i=0;i<n;i++)
count[i]=0;
while((noofedges<(n-1))&&(root!=NULL))
{
p=root;
root=root->next;
p1=getrval(p->v1);
p2=getrval(p->v2);
if(p1!=p2)
{
combine(p->v1,p->v2);
noofedges++;
mincost+=p->cost;
if(temp==NULL)
{
temp=p;
q=temp;
}
else
{
q->next=p;
q=q->next;
}
q->next=NULL;
}
}
return temp;
}
int getrval(int i)
{
int j,k,temp;
k=i;
while(stree[k]!=k)
k=stree[k];
j=i;
while(j!=k)
{
temp=stree[j];
stree[j]=k;
j=temp;
}
return k;
}
void combine(int i,int j)
{
```

```
if(count[i]<count[j])
stree[i]=j;
else
{
stree[j]=i;
if(count[i]==count[j])
count[j]++;
}
}
void del(lledge *root)
{
lledge *temp;
while(root!=NULL)
{
temp=root->next;
delete root;
root=temp;
}
}
void main()
{
clrscr();
lledge *temp,*root;
int i;
root=new lledge;

root->v1=4;
root->v2=3;
root->cost=1;
temp=root->next=new lledge;

temp->v1=4;
temp->v2=2;
temp->cost=2;
temp->next=new lledge;

temp=temp->next;
temp->v1=3;
temp->v2=2;
temp->cost=3;
temp->next=new lledge;

temp=temp->next;
temp->v1=4;
temp->v2=1;
temp->cost=4;
temp->next=new lledge;

root=kminstree(root,max);
for(i=1;i<max;i++)
cout<<"\nstree["<<i<<"]->"<<stree[i];
cout<<"\nthe minimum cost spanning tree is"<<mincost;
del(root);
```

```
getch();
}
```

Week 13
Write a C++ program to implement hashing techniques? Write a C++ program to create Binomial and Leftist heaps? *ASSIGNMENT:* Write a C++ program to demonstrate the concept of rehashing? Write a C++ program to create Max and Min heaps?

A C++ program to implement hashing techniques

```
// CPP program to implement hashing with chaining

#include<iostream>
#include <list>
using namespace std;

class Hash
{
int BUCKET;    // No. of buckets

// Pointer to an array containing buckets
list<int> *table;
public:
Hash(int V);  // Constructor

// inserts a key into hash table
void insertItem(int x);

// deletes a key from hash table
void deleteItem(int key);

// hash function to map values to key
int hashFunction(int x) {
return (x % BUCKET);
}

void displayHash();
};

Hash::Hash(int b)
{
this->BUCKET = b;
table = new list<int>[BUCKET];
}

void Hash::insertItem(int key)
```

```
{
int index = hashFunction(key);
table[index].push_back(key);
}

void Hash::deleteItem(int key)
{
// get the hash index of key
int index = hashFunction(key);

// find the key in (inex)th list
list <int> :: iterator i;
for (i = table[index].begin();
i != table[index].end(); i++) {
if (*i == key)
break;
}

// if key is found in hash table, remove it
if (i != table[index].end())
table[index].erase(i);
}

// function to display hash table
void Hash::displayHash() {
for (int i = 0; i < BUCKET; i++) {
cout << i;
for (auto x : table[i])
cout << " --> " << x;
cout << endl;
}
}


int main()
{
// array that contains keys to be mapped
int a[] = {15, 11, 27, 8, 12};
int n = sizeof(a)/sizeof(a[0]);

// insert the keys into the hash table
Hash h(7);   // 7 is count of buckets in
// hash table
for (int i = 0; i < n; i++)
h.insertItem(a[i]);

// delete 12 from hash table
h.deleteItem(12);

// display the Hash table
h.displayHash();

return 0;
```

```
}
*/Output:
0
1 --> 15 --> 8
2
3
4 --> 11
5
6 --> 27
*/
```

A C++ program to create Binomial and Leftist heaps

```
//C++ program for leftist heap / leftist tree
#include <iostream>
#include <cstdlib>
using namespace std;

// Node Class Declaration
class LeftistNode
{
public:
int element;
LeftistNode *left;
LeftistNode *right;
int dist;
LeftistNode(int & element, LeftistNode *lt = NULL,
LeftistNode *rt = NULL, int np = 0)
{
this->element = element;
right = rt;
left = lt,
dist = np;
}
};

//Class Declaration
class LeftistHeap
{
public:
LeftistHeap();
LeftistHeap(LeftistHeap &rhs);
~LeftistHeap();
bool isEmpty();
bool isFull();
int &findMin();
void Insert(int &x);
void deleteMin();
void deleteMin(int &minItem);
void makeEmpty();
void Merge(LeftistHeap &rhs);
```

```
LeftistHeap & operator =(LeftistHeap &rhs);
private:
LeftistNode *root;
LeftistNode *Merge(LeftistNode *h1,
LeftistNode *h2);
LeftistNode *Merge1(LeftistNode *h1,
LeftistNode *h2);
void swapChildren(LeftistNode * t);
void reclaimMemory(LeftistNode * t);
LeftistNode *clone(LeftistNode *t);
};

// Construct the leftist heap
LeftistHeap::LeftistHeap()
{
root = NULL;
}

// Copy constructor.
LeftistHeap::LeftistHeap(LeftistHeap &rhs)
{
root = NULL;
*this = rhs;
}

// Destruct the leftist heap
LeftistHeap::~LeftistHeap()
{
makeEmpty( );
}

/* Merge rhs into the priority queue.
rhs becomes empty. rhs must be different
from this.*/
void LeftistHeap::Merge(LeftistHeap &rhs)
{
if (this == &rhs)
return;
root = Merge(root, rhs.root);
rhs.root = NULL;
}

/* Internal method to merge two roots.
Deals with deviant cases and calls recursive Merge1.*/
LeftistNode *LeftistHeap::Merge(LeftistNode * h1,
LeftistNode * h2)
{
if (h1 == NULL)
return h2;
if (h2 == NULL)
return h1;
if (h1->element < h2->element)
return Merge1(h1, h2);
```

```
else
return Merge1(h2, h1);
}

/* Internal method to merge two roots.
Assumes trees are not empty, and h1's root contains
smallest item.*/
LeftistNode *LeftistHeap::Merge1(LeftistNode * h1,
LeftistNode * h2)
{
if (h1->left == NULL)
h1->left = h2;
else
{
h1->right = Merge(h1->right, h2);
if (h1->left->dist < h1->right->dist)
swapChildren(h1);
h1->dist = h1->right->dist + 1;
}
return h1;
}

// Swaps t's two children.
void LeftistHeap::swapChildren(LeftistNode * t)
{
LeftistNode *tmp = t->left;
t->left = t->right;
t->right = tmp;
}

/* Insert item x into the priority queue, maintaining
heap order.*/
void LeftistHeap::Insert(int &x)
{
root = Merge(new LeftistNode(x), root);
}

/* Find the smallest item in the priority queue.
Return the smallest item, or throw Underflow if empty.*/
int &LeftistHeap::findMin()
{
return root->element;
}

/* Remove the smallest item from the priority queue.
Throws Underflow if empty.*/
void LeftistHeap::deleteMin()
{
LeftistNode *oldRoot = root;
root = Merge(root->left, root->right);
delete oldRoot;
}
```

```
/* Remove the smallest item from the priority queue.
Pass back the smallest item, or throw Underflow if empty.*/
void LeftistHeap::deleteMin(int &minItem)
{
if (isEmpty())
{
cout<<"Heap is Empty"<<endl;
return;
}
minItem = findMin();
deleteMin();
}

/* Test if the priority queue is logically empty.
Returns true if empty, false otherwise*/
bool LeftistHeap::isEmpty()
{
return root == NULL;
}

/* Test if the priority queue is logically full.
Returns false in this implementation.*/
bool LeftistHeap::isFull()
{
return false;
}

// Make the priority queue logically empty
void LeftistHeap::makeEmpty()
{
reclaimMemory(root);
root = NULL;
}

// Deep copy
LeftistHeap &LeftistHeap::operator =(LeftistHeap & rhs)
{
if (this != &rhs)
{
makeEmpty();
root = clone(rhs.root);
}
return *this;
}

// Internal method to make the tree empty.
void LeftistHeap::reclaimMemory(LeftistNode * t)
{
if (t != NULL)
{
reclaimMemory(t->left);
reclaimMemory(t->right);
delete t;
```

```
}
}

// Internal method to clone subtree.
LeftistNode *LeftistHeap::clone(LeftistNode * t)
{
if (t == NULL)
return NULL;
else
return new LeftistNode(t->element, clone(t->left),
clone(t->right), t->dist);
}

//Driver program
int main()
{
LeftistHeap h;
LeftistHeap h1;
LeftistHeap h2;
int x;
int arr[]= {1, 5, 7, 10, 15};
int arr1[]= {22, 75};

h.Insert(arr[0]);
h.Insert(arr[1]);
h.Insert(arr[2]);
h.Insert(arr[3]);
h.Insert(arr[4]);
h1.Insert(arr1[0]);
h1.Insert(arr1[1]);

h.deleteMin(x);
cout<< x <<endl;

h1.deleteMin(x);
cout<< x <<endl;

h.Merge(h1);
h2 = h;

h2.deleteMin(x);
cout<< x << endl;

return 0;
}
```