Department of Information Technology

DATA STRUCTURES THROUGH C++
Lab Manual

(II B.tech -I Sem)

D.SRAVANTHI
Asst. Professor

J.B.Institute of Engineering & Technology
Yenkapally, Moinabad(Mandal)
Himathnagar(post),Hydreabad

INDEX

S.No	Name of the Programs
1	Implementation of stack ADT using arrays and performing push, pop and display
2	Implementation of queue ADT and performing insert, delete and display operations.
3	Implementation of stack ADT using linked list and performing push, pop and display Operations.
4	Implementation of queue using array and performing insert, delete and display Operations
5	Implementation of deque ADT using a doubly linked list.
6	Implementation of Binary Search Tree operations insert, delete search and Display.
7	To write c++ program that use recursive/non recursive functions to traverse the given. Binary tree
8	A) Implementation of breadth first search for given graph. B) Implementation of depth first search for a given graph.
9	A) Implementation of Merge Sort using template. B) Implementation of heap sort method in c++.
10	To write c++ program to perform insertion into AVLtree and deletion from an AVL tree
11	Implementation of dictionary functions using hashing.
12	Implementation of Knuth-Morris-Pratt algorithm
13	Implementation of various operations of B-tree

1. STACK ADT USING ARRAYS

Implementation of stack ADT using arrays and performing push, pop and display operations.

Program:

```
 #include<iostream.h>
enum bool
{
  true,false
}anyval;
template<class T>
class Stack
{
  T *t;
  int max,len;
  public:
  Stack(int m);
  ~Stack();
  bool Isempty();
  int Lenth();
  void Top();
  void Pop();
  void Push(T x);
  void Display();
};

//constructor definition

template<class T>
Stack<T>::Stack(int m)
{
  t=new T[m];len=-1;max=m;
}


//destructor definition
template<class T>
Stack<T>::~Stack()
{
  delete[]t;
}

//function used to find whether stack is empty or not
template<class T>
bool Stack<T>::Isempty()
{
  if(len==-1)
  return(true);
  else
  return(false);
}

//function which gives topmost element of stack
template<class T>
void Stack<T>::Top()
{
  if(len!=-1)
  cout<<"present top element is"<<t[len];
}
```

```
//function used to delete topmost element of stack
template<class T>
void Stack<T>::Pop()
{
 if(len==-1)
 cout<<"stack underflow condition"<<endl;
 else{
 cout<<t[len];len--;}
}

//function which inserts element in to stack
template<class T>
void Stack<T>::Push(T x)
{
 if(len==max-1)
 cout<<"stack overflow condition"<<endl;
 else{
 len++;
 t[len]=x;}
}
//function which gives length of stack
template<class T>
int Stack<T>::Lenth()
{
 return(len+1);
}

//function which displays elements of stack
template<class T>
void Stack<T>::Display()
{
 if(len!=-1)
 {
  for(int i=0;i<=len;i++)
  cout<<t[i]<<endl;
 }
 else
 cout<<"empty stack"<<endl;
}

void main()
{
 Stack<int>st(10);
 st.Push(1);
 st.Push(2);
 st.Push(3);
 st.Display();
 st.Pop();
 cout<<"len of stack"<<st.Lenth();
 cout<<st.Isempty();
 st.Top();
}
```

INPUT/OUTPUT

Sample Input:

Sample output:

1
2
3
3
len of stack is 2
1
present top
element is 2

2. QUEUE ADT USING ARRAYS

Implementation of queue using array and performing insert, delete and display operations.

Program:

```
# include < iostream.h >
enum bool
{
true,false
}anyval;
template<class T>
class queue
{
T *t;
int f,r,len,max;
public:
queue(int m,int a,int b);
~queue();
bool isempty();
int lenth();
void insert(T x);
void delet();
void display();
}

template<class T>
queue<T>::queue(int m,int a,int b)
{
 max=m;f=a;r=b;
 t=new T[m];
}

template<class T>
```

```
queue<T>::~queue()
{
delete[]t;
}
template<class T>
bool queue<T>::isempty()
{
if(f>r)
return(true);
else
return(false);
}

template<class T>
int queue<T>::lenth()
{
if(f>r)
return(0);
else
{
len=r-f+1;
return(len);
}
}

template<class T>
void queue<T>::insert(T x)
{
if(r==max-1)
cout<<"Queue overflow condition"<<endl;
else
{
r++;
t[r]=x;
}
}

template<class T>
void queue<T>::delet()
{
if(f>r)
cout<<"Queue underflow condition"<<endl;


else
{
cout<<"Deleted element is"<<t[f];
f++;
}
}

template<class T>
void queue<T>::display()
{
```

```
if(f>r)
cout<<"Queue is empty"<<endl;
else
{
for(int i=f;i<=r;i++)
cout<<t[i]<<endl;
}
}

void main()
{
queue<int>qu(10,0,-1);
qu.insert(1);
qu.insert(2);
qu.insert(3);
qu.display();
qu.delet();
cout<<"lenth of queue"<<qu.lenth();
cout<<qu.isempty();
}
```

INPUT/OUTPUT

Sample Input:

Sample output:

```
1
        2
        3
Deleted element is 3
Lenth of queue is 2
        1
```

3. STACK ADT USING LINKED LIST

Implementation of stack ADT using linked list and performing push, pop and display operations.

Program:

```
#include<iostream.h>
template<class T>
class Stack
{
 struct node
 {
 T inf;
 node *link;
 }*head,*top,*cur;
 public:
 Stack();
 ~Stack();
 int Isempty();
 void Push(T x);
```

```
  void Pop();
  void Display();
  void Topele();
  int Lenth();
};

//constructor definition
template<class T>
Stack<T>::Stack()
{
  head=cur=top=NULL;
}

//destructor definition
template<class T>
Stack<T>::~Stack()
{
  cur=head;
  if(cur==NULL)
  return;
  else
  while(cur)
  {
  head=head->link;
  delete(cur);
  cur=head;
  }
}

//function used to find whether stack is empty or not
template<class T>
int Stack<T>::Isempty()
{
  if(head==NULL)
  return(1);
  else
  return(0);
}

//function used to insert element in to the stack
template<class T>
void Stack<T>::Push(T x)
{
  node *p=new node;
  p->inf=x;
  p->link=NULL;
  if(head==NULL)
  head=p;
  else
  {
  for(cur=head;cur->link;cur=cur->link);
  cur->link= p;
  top=p;
  }
```

```
}

//function used to delete topmost element from the stack
template<class T>
void Stack<T>::Pop()
{
 node *q;
 for(cur=head;cur->link;q=cur,cur=cur->link);
 q->link=NULL;
 top=q;
 cout<<"deleted element is:"<<cur->inf;
 delete(cur);
}

//displays the elements of the stack
template<class T>
void Stack<T>::Display()
{
 for(cur=head;cur!=NULL;cur=cur->link)
 cout<<cur->inf<<endl;
}

//function which gives topmost element of the stack
template<class T>
void Stack<T>::Topele()
{
 if(top)
 cout<<"top element is:"<<top->inf;
}
//function which gives lenth of the stack
template<class T>
int Stack<T>::Lenth()
{
 int len=0;
 for(cur=head;cur;len++,cur=cur->link);
 return(len);
}

void main()
{
 Stack<int>sll;    sll.Push(10);    sll.Push(20);   sll.Push(30);    sll.Display();
 sll.Pop();    cout<<sll.Isempty();   sll.Topele();    cout<<"lenth of stack:"<<sll.Lenth();
}
```

INPUT/OUTPUT

Sample Input:

Sample Output:

```
10
20
30
deleted element is 30
0
top element is 20
lenth of stack is 2
```

4. QUEUE ADT USING LINKED LIST

Implementation of queue ADT and performing insert, delete and display operations.

Program:

```
#include<iostream.h>
template<class T>
class Queue
{
 struct node
 {
 T inf;
 node *link;
 }*head,*f,*r,*cur;
 public:
 Queue();
 ~Queue();
 int Isempty();
 void Insert(T x);
 void Delet();
 void Display();
 void Lenth();
};

//constructor definition
template<class T>
Queue<T>::Queue()
{
  head=f=r=cur=NULL;
}

//destructor definition
template<class T>
Queue<T>::~Queue()
{
 cur=f;
 if(cur==NULL)
 return;
 else
 while(cur)
 {
 f=f->link;
 delete(cur);
 cur=f;
 }
}
//function used to find whether queue is empty or not
template<class T>
```

```
int Queue<T>::Isempty()
{
 if(f==NULL)
 return(1);
 else
 return(0);
}
//function used to insert elements at rear end of the queue
template<class T>
void Queue<T>::Insert(T x)
{
 node *p=new node;
 p->inf=x;
 p->link=NULL;
 if(head==NULL)
 {
 head=f=p;
 }
 else
 {
 for(cur=f;cur->link;cur=cur->link);
 cur->link=p;
 r=p;
 }
}
//function used to delete elements from front end of queue
template<class T>
void Queue<T>::Delet()
{
 cur=f;
 f=f->link;
 cur->link=NULL;
 cout<<"Deleted element is"<<cur->inf<<endl;
 delete(cur);
}

//function used to display elements of queue
template<class T>
void Queue<T>::Display()
{
 for(cur=f;cur!=NULL;cur=cur->link)
 cout<<cur->inf<<endl;
}

//function which gives length of queue
template<class T>
void Queue<T>::Lenth()
{
 int len=0;
 for(cur=f;cur!=r;cur=cur->link,len++);
 len++;
 cout<<"lenth of queue is"<<len<<endl;
}
```

```
void main()
{
 Queue<int>qull;
 qull.Insert(10);
 qull.Insert(20);
 qull.Insert(30);
 qull.Display();
 qull.Delet();
 qull.Display();
 cout<<qull.Isempty();
 qull.Lenth();
}
```

INPUT/OUTPUT

Sample Input:

Sample Output:

```
10
20
30
deleted element is 10
20
30
0
lenth of queue is 2
```

5. DOUBLE ENDED QUEUE ADT

Implementation of deque ADT using a doubly linked list.

Program:

```
#include <iostream.h>
class node
{
public:
int value;          //value stored in the node
node *next;         //pointer to next node
node *prev;         //pointer to previous node
};

class dlist
{
public:
node *front;       //pointer to front of list
node *back;        //pointer to back of list
dlist()
{
            front=NULL;
            back=NULL;
}
void insertFront(int value);
void insertBack(int value);
void removeFront();
void removeBack();
void printDListFront();
void printDListBack();
```

```
};

//insert a node before the front node
void dlist::insertFront (int value)
{
 node *newNode;
 if(this->front==NULL)
   {
            newNode=new node();
            this->front=newNode;
            this->back =newNode;
            newNode->prev=NULL;
            newNode->next=NULL;
            newNode->value=value;

    }
}

//insert a node after the last node
            void dlist::insertBack (int value)
            {
                        if(this->back==NULL)
                        {
                                    cout<<"insert at back";
                                    insertFront(value);
                        }

            }

//remove the front node
void dlist::removeFront ()
{
            removeNode(this->front);
}

//remove a back node
void dlist::removeBack  ()
{
            removeNode(this->back);

}
//Print the list from front
void dlist::printDListFront()
{
            node* curr2;
                        curr2= this->front;
                        cout<<"\n-----\n";
                        cout<<"Queue\n";
                        cout<<"-----\n";
                        //cout<<"size:"<<getQueueSize()<<endl;
            while(curr2!=NULL)
                        {
                                    cout<<" |"<<curr2->value<<"|";
                                    curr2=curr2->next;
```

```
                }
                cout<<endl;
}// print the Double Linked List from front

// print the Double Linked List from backwards
void dlist::printDListBack()
{
        node* curr2;
                curr2= this->back;
                cout<<"\n-----\n";
                cout<<"Queue\n";
                cout<<"-----\n";
                //cout<<"size:"<<getQueueSize()<<endl;
        while(curr2!=NULL)
                {
                        cout<<" |"<<curr2->value<<"|";
                        curr2=curr2->prev;
                }
                cout<<endl;
}// print the Double Linked List from back

void main()
{
        dlist *st ;
        st= new dlist();
st->insertBack(8);
st->printDListFront ();
st->insertBack(5);
st->printDListFront ();
st->insertBack(6);
st->printDListFront ();
st->insertFront(1) ;
st->printDListFront ();
st->insertFront(3) ;
st->printDListFront ();
st->insertBack(7);
st->printDListFront ();
st->removeFront();
st->printDListFront ();
st->removeBack();
st->printDListFront ();
}
```

INPUT/OUTPUT

Sample Input:

Sample Output:

```
----------
Queue
        ----------
        Size:1
        8
        ----------
        Queue
```

```
----------
Size:2
8 5
----------
Queue
----------
Size:3
6 8 5
----------
Queue
----------
Size:4
1 6 8 5
----------
Queue
----------
Size:5
3 1 6 8 5
----------
Queue
----------
Size:6
3 1 6 8 5 7
----------
Queue
----------
Size:5
1 6 8 5 7
----------
Queue
----------
size:4
1 6 8 5
```


6.BINARY SEARCH TREE OPERATIONS

Implementation of Binary Search Tree operations insert, delete search and display.

Program:

```
#include<iostream.h>
struct pair
{
        int key;
```

```
        int info;
};
struct bstnode
{
 pair ele;
 bstnode *left,*right;
 bstnode()
 {
        left=right=NULL;
 }
 bstnode(struct pair p)
 {
        this->ele=p;
        this->left=this->right=NULL;
 }
 bstnode(struct pair p,bstnode *lc,bstnode *rc)
 {
        this->ele=p;
        this->left=lc;
        this->right=rc;
 }
};
class bst
{
 public:
 bstnode *root;
 int size;
 bst()
 {
        root=NULL;
        size=0;
 }
 void insert(pair &tp);
 void find(int &k);
 void delet(int &k);
 void display(bstnode *node);
};
void bst::find(int &k)
{
 bstnode *p=root;
 while(p)
 {
        if(k<p->ele.key)
        p=p->left;
        else if(k>p->ele.key)
        p=p->right;
        else
        {
        cout<<"element corresponding to key :"
        <<p->ele.key<<" "<<p->ele.info <<endl;
        break;
   }
 }
}
```

```
void bst::insert(pair &tp)
{
 bstnode *p=root,*pp=NULL;
 while(p)
 {
 pp=p;
 if(tp.key<p->ele.key)
 p=p->left;
 else if(tp.key>p->ele.key)
 p=p->right;
 else{
 p->ele.info=tp.info;return;}
 }
 bstnode *nnode=new bstnode(tp);
 if(pp==NULL)
 {
        root=nnode;
        return;
 }
 else if(pp->ele.key<tp.key)
 pp->right=nnode;
 else
 pp->left=nnode;
 size++;
}
void bst::delet(int &k)
{
 bstnode *p=root,*pp=NULL;
 while((p)&&(p->ele.key!=k))
 {
        pp=p;
        if(k<p->ele.key)
        p=p->left;
        else
        p=p->right;
 }
 if(p==NULL)
 {
        cout<<"element is not found";
        return;
 }

 if((p->left)&&(p->right))
 {
        bstnode *s=p->left;
        bstnode *ps=p;
        while(s->right!=NULL)
        {
                ps=s;
                s=s->right;
        }
        bstnode *q=new bstnode(s->ele,p->left,p->right);
        if(pp==NULL)
        root=q;
```

```
            else
            if(p==pp->left)
            pp->left=q;
            else
            pp->right=q;
            if(ps==p)
            pp=ps;
            else
            pp=ps;
            delete(p);
            p=s;
  }
  bstnode *l;
  if(p->left!=NULL)
  l=p->left;
  else
  l=p->right;

  if(p==root)
  root=l;
  else
  {
            if(p==pp->left)
            pp->left=l;
            else
            pp->right=l;
  }
  size--;
  delete(p);
}

 void bst::display(bstnode *node)
 {
  if(node!=NULL)
  {
  display(node->left);
  cout<<node->ele.key<<" "<<node->ele.info<<endl;
  display(node->right);
  }
 }
 int main()
 {
  struct pair t;
  bst a;
  int ch;
  do{
  cout<<"1-insert,2-delete,3-find,4-display";
  cout<<"nter ur choice"<<endl;
  cin>>ch;
  switch(ch){
  case 1:cout<<"enter any element";
                              cin>>t.key>>t.info;
                              a.insert(t);break;
  case 2:cout<<"enter any key";
```

```
                        cin>>t.key;
                        a.delet(t.key);break;
  case 3:cout<<"enter any key";
                        cin>>t.key;a.find(t.key);
                        break;
  case 4:cout<<"key"<<" "<<"data"<<endl;
                        a.display(a.root);break;
  default:cout<<"invalid choice"<<endl;}
  }while(ch!=5); return(0);
}
```

INPUT/OUTPUT

```
1-inser 2-delete 3-find 4-display
enter ure choice
1
enter any element
10
enter ure choice
1
enter any element
20

enter ure choice
1
enter any element
30
enter ure choice
3
enter a key
20
Element is found
enter ure choice
2
enter any element
30
Element is deleted
enter ure choice
4
Elements in the tree are 20  10

enter ure choice
5
```

7.NON RECURSIVE TREE TRAVERSALS

To write c++ program that use non recursive functions to traverse the given binary tree.

Program:

```
# include <conio.h>
# include <process.h>
# include <iostream.h>
# include <alloc.h>
struct node
{
        int ele;
        node *left;
        node *right;
};
typedef struct node *nodeptr;
class stack
{
        private:
                struct snode
                {
                        nodeptr ele;
                        snode *next;
                };
                snode *top;
        public:
                stack()
                {
                        top=NULL;
                }
                void push(nodeptr p)
                {
                        snode *temp;
                        temp = new snode;
                        temp->ele = p;
                        temp->next = top;
                        top=temp;
                }



void pop()
                {
                        if (top != NULL)
                        {
                        nodeptr t;
                        snode *temp;
                        temp = top;
                        top=temp->next;
                      delete temp;
```

```
                }
        }
        nodeptr topele()
        {
                if (top !=NULL)
                        return top->ele;
                else
                        return NULL;
        }
        int isempty()
        {
        return ((top == NULL) ? 1 : 0);
        }

};
class bstree
{
        public:
                void preordernr (nodeptr); void bstree::preordernr(nodeptr p)
{
        stack s;
        while (1)
        {
        if  (p != NULL)
        {
                cout<<p->ele<<"-->";
                s.push(p);
                p=p->left;
        }
        else
        if (s.isempty())
        {
                cout<<"Stack is empty";
                return;
        }
        else
        {
                nodeptr t;
                t=s.topele();
                p=t->right;
                s.pop();
        }
        }
}
void bstree::inordernr(nodeptr p)
{
        stack s;
        while (1)
        {
        if  (p != NULL)
        {
                s.push(p);
                p=p->left;
        }
```

```
        else
        {
        if (s.isempty())
        {
                cout<<"Stack is empty";
                return;
        }
        else
        {
                p=s.topele();
                cout<<p->ele<<"-->";
        }
        s.pop();
        p=p->right;
        }
        }
}
void bstree::postordernr(nodeptr p)
{
        stack s;
        while (1)
        {
        if  (p != NULL)
        {
                s.push(p);
                p=p->left;
        }
        else
        {
                if (s.isempty())
                {
                        cout<<"Stack is empty";
                        return;
                }
                else
                if (s.topele()->right == NULL)
                {
                        p=s.topele();
                        s.pop();
                        cout<<p->ele<<"-->";
                        if (p==s.topele()->right)
                        {
                                cout<<s.topele()->ele<<"-->";
                                s.pop();
                        }
                }
                if (!s.isempty())
                        p=s.topele()->right;
                else
                        p=NULL;
        }
        }
}
```

```
                    void inordernr(nodeptr);
                     void postordernr(nodeptr);
};

int main()
{
int ch,x,leftele,rightele;
bstree bst;
char c='y';
nodeptr root,root1,min,max;
root=NULL;
root1=NULL;

do
{
//      system("clear");
        clrscr();
        cout<<"Binary Search Tree ";
        cout<<"-------------------------";
cout<<"1.Preorder
        2.Inorder
        3.Postorder";
cout<<"Enter your choice :";
        cin>>ch;

        switch(ch)
        {
        case 1:Preorder
                if (root==NULL)
                        cout<<"Tree is empty";
                else
                {
                        cout<<"Preorder traversal (Non-Recursive) is :";
                        bst.preordernr(root);
case 2:.Inorder
        if (root==NULL)
        cout<<"Tree is empty";
        else
        {
                cout<<"Inorder traversal (Non-Recursive) is :";
                        bst.inordernr(root);

case 3:cout<<".Postorder";
                if (root==NULL)
                        cout<<"Tree is empty";
                else
                {
                        cout<<"Postorder traversal (Non-Recursive) is :";
                        bst.postordernr(root);
cout<<"Continue (y/n) ?";
        cin>>c;
        }while (c=='y' || c == 'Y');
        return 0;
}
```

INPUT/OUTPUT

Input:

1.preorder
2.inorder
3.postorder
enter ur choice
1
1.preorder
2.inorder
3.postorder
enter ur choice
2
1.preorder
2.inorder
3.postorder
enter ur choice :3

Output:

1 2 4 6 8 7 9 10 3 5
8 6 4 9 7 10 2 1 5 3
8 6 9 10 7 4 2 5 3 1

8.BREADTH FIRST SEARCH

Implementation of breadth first search for given graph.

Program:

```
#include<iostream.h>
#define G 9
#define false 0
#define true 1
int visited[G],v=1;
int q[G],w,front=0,rear=0,k;
void addq(int,int []);
void delq();
static int vertex[G][G]={{0},{0,2,8,3},{0,1,4,5},{0,1,7,6},
{0,2,8},{0,2,8},{0,8,3},{0,8,3},{0,4,5,1,6,7}};
void main()
{
 for(v=1;v<G;v++)
 visited[v]=false;
 for(v=1;v<G;v++)
 if(!visited[v])
 {
  addq(v,q);
  do
  {
  delq();
  visited[v]=true;
  for(w=1;w<G;w++)
  {
  if(vertex[v][w]==0)
  continue;
  if(!visited[vertex[v][w]])
  {
  visited[vertex[v][w]]=true;
  addq(vertex[v][w],q);
```

```
}
}
}while(front!=rear);
}
for(v=0;v<G-1;v++)
cout<<q[v]<<" ";
}
void addq(int v,int q[])
{
q[rear]=v;
rear++;
}
void delq()
{
front++;
if(front==G)
{
front=rear=0;
}
}
```

INPUT/OUTPUT

Sample Input:

Sample Output:

1 2 8 3 4 5 6 7

DEPTH FIRST SEARCH

Implementation of depth first search for a given graph.

of the adjacent vertices of this adjacent vertex.

Program

```
#include<iostream.h>
#define G 9
#define false 0
#define true 1
int visited[G],v=1;
void traverse(int);
static int vertex[G][G]={{0},{0,2,8,3},{0,1,4,5},{0,1,7,6},
{0,2,8},{0,2,8},{0,8,3},{0,8,3},{0,4,5,1,6,7}};
void main()
{
 for(v=1;v<G;v++)
 visited[v]=false;
 for(v=1;v<G;v++)
 if(!visited[v])
   traverse(v);
 }
 void traverse(int v)
 {
  int w;
  visited[v]=true;
  if(v==1)
  cout<<v<<" ";
  for(w=1;w<G;w++)
  {
  if(vertex[v][w]==0)
  continue;
  if(!visited[vertex[v][w]])
  {
  cout<<vertex[v][w]<<" ";
  traverse(vertex[v][w]);
  }
  }
}
```


INPUT/OUTPUT

Sample Input:

Sample Output:

1 2 4 8 5 6 3 7

9. MERGE SORT USING TEMPLATES

Implementation of Merge Sort using template.

Program:

```
//  merge sort
#include <iostream.h>
template<class T>
void Merge(T c[], T d[], int l, int m, int r)
{// Merge c[l:m]] and c[m:r] to d[l:r].
  int i = l,    // cursor for first segment
      j = m+1,  // cursor for second
      k = l;    // cursor for result

  // merge until i or j exits its segment
  while ((i <= m) && (j <= r))
    if (c[i] <= c[j]) d[k++] = c[i++];
    else d[k++] = c[j++];

  // take care of left overs
```

```
  if (i > m) for (int q = j; q <= r; q++)
            d[k++] = c[q];
  else for (int q = i; q <= m; q++)
        d[k++] = c[q];
}

template<class T>
void MergePass(T x[], T y[], int s, int n)
{// Merge adjacent segments of size s.
  int i = 0;
  while (i <= n - 2 * s) {
    // merge two adjacent segments of size s
    Merge(x, y, i, i+s-1, i+2*s-1);
    i = i + 2 * s;
    }

  // fewer than 2s elements remain
  if (i + s < n) Merge(x, y, i, i+s-1, n-1);
  else for (int j = i; j <= n-1; j++)
        // copy last segment to y
        y[j] = x[j];
}

template<class T>
void MergeSort(T a[], int n)
{// Sort a[0:n-1] using merge sort.
  T *b = new T [n];
  int s = 1;  // segment size
  while (s < n) {
    MergePass(a, b, s, n); // merge from a to b
    s += s;
   MergePass(b, a, s, n); // merge from b to a
   s += s;
    }
}

void main(void)
{
  int y[10] = {10,7,8,9,4, 2, 3, 6, 5,1};
  MergeSort(y,10);
  Cout<<"Sorted elements are:
  for (int i=0; i< 10; i++) cout << y[i] << ' ';
  cout << endl
}
```

INPUT/OUTPUT

Sample Output:

Sorted elements are:1 2 3 4 5 6 7 8 9 10

9.B. HEAP SORT

Implementation of heapsort method in c++.

Program:

```
#include < iostream.h >
#include < conio.h >
int heapSize = 10;
void print(int a[]) {
  for (int i = 0; i <= 9; i++) {
    cout << a[i] << "-";
  }
  cout << endl;
}
int parent(int i) {
if(i==1)
return 0;
if(i%2==0)
    return ( (i / 2)-1);
else
    return ( (i / 2));
}
int left(int i) {
  return (2 * i) + 1;
}
int right(int i) {
  return (2 * i) + 2;
}
void heapify(int a[], int i) {
  int l = left(i), great;
  int r = right(i);
  if ( (a[l] > a[i]) && (l < heapSize)) {
    great = l;
  }
  else {
    great = i;
  }
  if ( (a[r] > a[great]) && (r < heapSize)) {
    great = r;
  }
  if (great != i) {
    int temp = a[i];
    a[i] = a[great];
    a[great] = temp;
    heapify(a, great);
  } }

void BuildMaxHeap(int a[]) {
```

```
 for (int i = (heapSize - 1) / 2; i >= 0; i--) {
  heapify(a, i);
  print(a);
 } }

void HeapSort(int a[]) {
 BuildMaxHeap(a);
 for (int i = heapSize; i > 0; i--) {
  int temp = a[0];
  a[0] = a[heapSize - 1];
  a[heapSize - 1] = temp;
  heapSize = heapSize - 1;
  heapify(a, 0);
 }
}
void main() {
 int arr[] = {
    2, 9, 3, 6, 1, 4, 5, 7, 0, 8};
 HeapSort(arr);
 print(arr);
}
```

INPUT/OUTPUT

Sample Output:

Sample Input:

0 1 2 3 4 5 6 7 8 9

++ topics

10. ADELSON VELSEKY LANDIS(AVL) TREE

To write c++ program to perform insertion into AVLtree and deletion from an AVL tree.

Program:

```
# include <iostream.h>
# include <stdlib.h>
# include <conio.h>
struct node
{
  int element;
  node *left;
  node *right;
  int height;
};
typedef struct node *nodeptr;
class bstree
{
    public:
          void insert(int,nodeptr &);
          void del(int, nodeptr &);
```

```
        void inorder(nodeptr);
};

//Inserting a node
void bstree::insert(int x,nodeptr &p)
{
  if (p == NULL)
  {
        p = new node;
        p->element = x;
        p->left=NULL;
        p->right = NULL;
        p->height=0;
        if (p==NULL)
                cout<<"Out of Space";
  }
  else
  {
        if (x<p->element)
        {
          insert(x,p->left);
          if ((bsheight(p->left) - bsheight(p->right))==2)
          {
            if (x < p->left->element)
                p=srl(p);
            else
                p = drl(p);
          }
        }
        else if (x>p->element)
        {
            insert(x,p->right);
            if ((bsheight(p->right) - bsheight(p->left))==2)
            {
                if (x > p->right->element)
                        p=srr(p);
                else
                        p = drr(p);
             }
        }
        else
                cout<<"Element Exists";
        }
        int m,n,d;
        m=bsheight(p->left);
        n=bsheight(p->right);
        d=max(m,n);
        p->height = d + 1;

}
//Deleting a node
void bstree::del(int x,nodeptr &p)
{
        nodeptr d;
```

```
        if (p==NULL)
          cout<<"Element not found ";
        else if ( x < p->element)
          del(x,p->left);
        else if (x > p->element)
          del(x,p->right);
        else if ((p->left == NULL) && (p->right == NULL))
        {
          d=p;
          free(d);
          p=NULL;
          cout<<" Element deleted !";
        }
        else if (p->left == NULL)
        {
         d=p;
         free(d);
         p=p->right;
         cout<<" Element deleted !";
        }
        else if (p->right == NULL)
        {
         d=p;
         p=p->left;
         free(d);
         cout<<" Element deleted !";
        }
        else
          p->element = deletemin(p->right);
}

//Inorder Printing
void bstree::inorder(nodeptr p)
{
        if (p!=NULL)
        {
          inorder(p->left);
          cout<<p->element<<"-->";
          inorder(p->right);
     }
}
int bstree::bsheight(nodeptr p)
{
        int t;
        if (p == NULL)
                return -1;
        else
        {
                t = p->height;
                return t;
        }
}

int main()
```

```
{
        clrscr();
        nodeptr root,root1,min,max;//,flag;
        int a,choice,findele,delele,leftele,rightele,flag;
        char ch='y';
        bstree bst;
        //system("clear");
        root = NULL;
        root1=NULL;
        cout<<"
                AVL Tree";
        cout<<"========";
        do
        {
cout<<"1.Insertion;
cout<<"2.Delete;

cout<<"3.Inorder;
                cout<<"Enter the choice:";
                cin>>choice;
                switch(choice)
                {
                case 1:
                        cout<<"New node's value ?";
                        cin>>a;
                        bst.insert(a,root);
                       break;
              case  2:cout<<"Delete Node ?";
                        cin>>delele;
                        bst.del(delele,root);
                        bst.inorder(root);
                        break;
               case 3:cout<<"Inorder Printing.... :";
                      bst.inorder(root);
                      break;

cout<<"Do u want to continue (y/n) ?";
cin>>ch;
        }while(ch=='y');


        return 0;
}
```

INPUT/OUTPUT

Sample Input: **Sample output:**

AVL tree 15 20

_____

_____

1.insert
2.delete
3.inorder

```
enter ur choice
1
enter new node's value
20
1.insert
2.delete
3.inorder
enter ur choice
1
enter new node's value
15
1.insert
2.delete
3.inorder
enter ur choice
1
enter new node's value
25
1.insert
2.delete
3.inorder
enter ur choice
2
1.insert
2.delete
3.inorder
enter ur choice
3
```


11. DICTIONARY USING HASHING

Implementation of dictionary functions using hashing.

Program:

```
#include<iostream.h>
enum bool
{
false,true
```

```
}anyval;
template<class K,class E>
class hashtable
{
 int D;
 E *ht;
 bool *empty;
 public:
 hashtable(int divisor)
 {
 D=divisor;
 ht=new E[D];
 empty=new bool[D];
 for(int i=0;i<0;i++)
 empty[i]=true;
 }
 ~hashtable()
 {
 delete []ht;
 delete []empty;
 }
 bool search(K &,E &);
 void insert(E &);
 void display();
 int hsearch(K &);
};
template<class K,class E>
void hashtable<K,E>::display()
{
 for(int i=0;i<0;i++)
 if(empty[i]==false)
 cout<<ht[i]<<endl;
}
template<class K,class E>
int hashtable<K,E>::hsearch(K &k)
{
 int i=k%D;
 int j=i;
 do
 {
 if(empty[j]||(ht[j]==k))
 return j;
 j=(j+1)%D;
 }while(j!=i);
 return j;
}
template<class K,class E>
bool hashtable<K,E>::search(K &k,E &e)
{
 int b=hsearch(k);
 if(empty[b]||(ht[b]!=k))
 return false;
 e=ht[b];
 return true;
```

```
}
template<class K,class E>
void hashtable<K,E>::insert(E &e)
{
  K k=e;
  int b=hsearch(k);
  if(ht[b]==k)
  {
  cout<<"duplicate";
  return;
  }
  if(empty[b])
  {
  empty[b]=false;
  ht[b]=e;
  }
}
void main()
{
int t;
hashtable<int,int>hb(10);
hb.insert(10);hb.insert(20);hb.insert(30);
hb.display();
bool b=hb.search(10,t);
if(b)
cout<<"10 is present"<<endl;
else
cout<<"10 is not present\n";
b=hb.search(40,t);
if(b)
cout<<"40 is present"<<endl;
else
cout<<"40 is not present"<<endl;
}
```


INPUT/OUTPUT

Sample Input:

Sample Output:

10 is present

40 is not present.

12.Implementation of Knuth-Morris-Pratt algorithm

```
#include <iostream.h>
#include<string.h>
#include <stdlib.h>

class KMP
{
 private:
 char *text;
 char *pattern;
 public:

 KMP()
 {
        *text=NULL;
        *pattern = NULL;
 }
 void get_input();
 void display();
 const char *kmp(const char *text,const char *pattern);
};

const char *KMP::kmp(const char *text,const char *pattern)
{
                int T[50];
                int i, j;
                const char *result = NULL;
                //no pattern to match
                if (pattern[0] == '\0')
                        return text;
          // construct the lookup table
                T[0] = -1;
                for (i=0; pattern[i] != '\0'; i++)
                {
                                T[i+1] = T[i] + 1;
                                while (T[i+1]>0 && pattern[i]!=pattern[T[i+1]-1])
                                        T[i+1] = T[T[i+1]-1] + 1;
                }

                // searching for pattern match
                for (i=j=0; text[i] != '\0'; )
                {
```

```
                                if (j < 0 || text[i] == pattern[j])
                                {
                                        ++i, ++j;
                                        if (pattern[j] == '\0')
                                        {
                                                result = text+i-j;//storing the
matched pattern
                                                break;
                                        }
                                }
                                else j = T[j];
                }
                return result;
}
void KMP::get_input()
{
 cout<<"\n Enter The text";
 cin>>text;
 cout<<"\n Enter pattern";
 cin>>pattern;
}
void KMP::display()
{
 cout<<"\n The pattern matched is ... ";
 cout<<kmp(text,pattern);
}
void main()
{
 KMP obj;
 obj.get_input();
 obj.display();
}
```

13.Implementation of various operations of B-tree

```
#include<iostream.h>
#include <stdio.h>
#include <string.h>
#include<conio.h>
#include <stdlib.h>
#define MAX 4 //maximum order m
#define MIN 2 //minimum allowed elements

typedef char Type[10];
typedef struct Btree
{
        Type  key;
} BT;

typedef struct treenode
{
                int count;
                BT entry[MAX+1];
                treenode *branch[MAX+1];
}node;
class B
{
 private:
 node *root;
 public:
 int LT(char *,char *);
 int EQ(char *,char *);
 node *Search(Type target,node *root,int *targetpos);
 int SearchNode(Type target,node *current,int *pos);
 node *Insert(BT New,node *root);
 int MoveDown(BT New,node *current,BT *med,node **medright);
 void InsertIn(BT med,node *medright,node *current,int pos);
 void Split(BT med,node *medright,node *current,int pos,BT *newmedian, node
**newright);
 void Delete(Type target, node **root);
 void Del_node(Type target, node *current);
 void Remove(node *current, int pos);
 void Successor(node *current, int pos);
 void Adjust(node *current, int pos);
 void MoveRight(node *current, int pos);
 void MoveLeft(node *current, int pos);
 void Combine(node *current, int pos);
 void InOrder(node *root);
};

int B::LT(char *a,char *b)
{
 if((strcmp(a,b)) < (0))
        return 1;
 else
        return 0;
```

```
}
int B::EQ(char *a,char *b)
{
 if((strcmp(a,b)) == (0))
          return 1;
 else
 return 0;
}
/*
Search: traverse B-tree in search of desired node
*/

node* B::Search(Type target, node *root, int *targetpos)
{
  if (root==NULL)
    return NULL;
  else if (SearchNode(target, root, targetpos))
    return root;
  else
    return Search(target, root->branch[*targetpos], targetpos);
}
/*
SearchNode: searches keys in node for target.
*/
int B::SearchNode(Type target,node *current, int *pos)
{
  if (LT(target, current->entry[1].key))
  {   /* searching the leftmost branch. */
    *pos = 0;
    return 0;
  }
  else
  {  /* searching through the keys.    */
    for(*pos = current->count;
     LT(target, current->entry[*pos].key) && *pos > 1; (*pos)--);
    return EQ(target, current->entry[*pos].key);
  }
}
/*
Insert: inserts entry into the B-tree.
*/
node *B::Insert(BT newentry,node *root)
{
  BT medentry; // node to be inserted as new root
  node *medright; // subtree on right of medentry
  node *New;  // required when the height of the tree increases
  if (MoveDown(newentry, root, &medentry, &medright))
  {
          New = new node;
          New->count = 1;
          New->entry[1] = medentry;
       New->branch[0] = root;
          New->branch[1] = medright;
          return New;
```

```
  }
 return root;
}
/*
 MoveDown: recursively move down tree searching for newentry.
*/

int B::MoveDown(BT New,node *current,BT *med,node **medright)
{
 int pos;
 if (current == NULL)
 {
  *med = New;/*new node*/
  *medright = NULL;
  return 1;
 }
 else
 {              /* Search the current node.    */
   if(SearchNode(New.key, current, &pos))
     cout<<"Duplicate key !!";
   if(MoveDown(New, current->branch[pos], med, medright))
     if (current->count < MAX)
      {
             //insertion of median key.
             InsertIn(*med, *medright, current, pos);
             return 0;
      }
     else
     {
      Split(*med, *medright, current, pos, med, medright);
      return 1;
    }
    return 0;
 }
}

/*
 InsertIn: inserts a key into a node
 */
void B::InsertIn(BT med,node *medright,node *current, int pos)

{
 /* index to move keys to make room for medentry   */
 int i;
 for (i = current->count; i > pos; i--)
 {
  /* Shift all the keys and branches to the right    */
  current->entry[i+1] = current->entry[i];
  current->branch[i+1] = current->branch[i];
 }
 current->entry[pos+1] = med;
 current->branch[pos+1] = medright;
 current->count++;
}
```

```
/*
 Split: splits a full node.
*/

void B::Split(BT med,node *medright,node *current, int pos,
                                    BT *newmedian,node **newright)
{
  int i;      /* used for copying from *current to new node   */
  int median; /* median position in the combined, overfull node   */
  if (pos <= MIN) /* Determine if new key goes to left or right half. */
    median = MIN;
  else
    median = MIN + 1;
 /* Get a new node and put it on the right.  */
  *newright = new node;
  for (i = median+1; i <= MAX; i++)
   { /* Move half the keys.  */
     (*newright)->entry[i - median] = current->entry[i];
     (*newright)->branch[i - median] = current->branch[i];
   }
   (*newright)->count = MAX - median;
   current->count = median;
   if (pos <= MIN)     /* Pushing in the new key. */
     InsertIn(med, medright, current, pos);
   else
     InsertIn(med, medright, *newright, pos - median);
   *newmedian = current->entry[current->count];
   (*newright)->branch[0] = current->branch[current->count];
   current->count--;
}

/*
 Delete: deletes target from the B-tree.
*/

void B::Delete(Type target, node **root)
{
  node *Prev;
  Del_node(target,*root);
  if ((*root)->count == 0)//empty root
  {
          Prev = *root;
          *root = (*root)->branch[0];
          free(Prev);
  }
}

/*
Del_node: look for target to delete.
*/

void B::Del_node(Type target,node *current)
{
  int pos;    /* location of target or of branch on which to search   */
```

```
 if (!current)
 {
  cout<“Item not in the B-tree.”;
  return;
 }
 else
 {
   if (SearchNode(target, current, &pos))
     if (current->branch[pos-1])
      {
           Successor(current, pos);
       /* replaces entry[pos] by its successor */
                    Del_node(current->entry[pos].key,current->branch[pos]);
      }
      else
           Remove(current, pos);
      /* removes key from pos of *current   */
   else
   /* Target was not found in the current node.*/
       Del_node(target, current->branch[pos]);
   if (current->branch[pos])
    if (current->branch[pos]->count < MIN)
          Adjust(current, pos);
 }
}

/*
Remove: delete an entry and the branch to its right.
*/

void B::Remove(node *current, int pos)
{
 int i;      /* index for moving entries */
 for (i = pos+1; i <= current->count; i++)
 {
  current->entry[i-1] = current->entry[i];
  current->branch[i-1] = current->branch[i];
 }
 current->count– –;
}
/*
Successor: finds and replaces an entry by its immediate successor.
*/
void B::Successor(node *current, int pos)
{
 node *leaf;     /* used to move down the tree to a leaf */
 /* Move to leftmost    */
 for (leaf=current->branch[pos];leaf->branch[0];
 leaf = leaf->branch[0]);
   current->entry[pos] = leaf->entry[1];
}

/*
Adjust: Adjusts the minimum number of entries.
```

```
*/

void B::Adjust(node *current, int pos)
{

 if (pos == 0)   /*  leftmost key  */
  if (current->branch[1]->count > MIN)
    MoveLeft(current, 1);
  else
    Combine(current, 1);
 else if (pos == current->count) /*  rightmost key */
  if (current->branch[pos-1]->count > MIN)
        MoveRight(current, pos);
  else
        Combine(current, pos);
 else if (current->branch[pos-1]->count > MIN)
        MoveRight(current, pos);
 else if (current->branch[pos+1]->count > MIN)
        MoveLeft(current, pos+1);
 else
     Combine(current, pos);
}

/*
 MoveRight: move a key to the right.
*/
void B::MoveRight(node *current, int pos)
{
                int i;
                node *t;

                t = current->branch[pos];
                for (i = t->count;i > 0;i – –)
                {
                /* Shift all keys in the right node one position.   */
                                t->entry[i+1] = t->entry[i];
                                t->branch[i+1] = t->branch[i];
                }
                 /* Move key from parent to right node.  */
                t->branch[1] = t->branch[0];
                t->count++;
                t->entry[1] = current->entry[pos];
                /* Move last key of left node into parent  */
                t = current->branch[pos-1];
                current->entry[pos] = t->entry[t->count];
                current->branch[pos]->branch[0] = t->branch[t->count];
                t->count--;
}

/*
 MoveLeft: move a key to the left.

 */
void B::MoveLeft(node *current, int pos)
```

```
{
        int c;
        node *t;
        t = current->branch[pos-1]; /* Move key from parent into left node. */
        t->count++;
        t->entry[t->count] = current->entry[pos];
        t->branch[t->count] = current->branch[pos]->branch[0];

        t = current->branch[pos];  /* Move key from right node into parent. */
        current->entry[pos] = t->entry[1];
        t->branch[0] = t->branch[1];
        t->count– –;
        for (c = 1; c <= t->count; c++) {
                /* Shift all keys in right node one position leftward.  */
                t->entry[c] = t->entry[c+1];
                t->branch[c] = t->branch[c+1];
        }
}

/*
Combine: combine adjacent nodes.
*/

void B::Combine(node *current, int pos)
{
        int c;
        node *right;
        node *left;

        right = current->branch[pos];
        left = current->branch[pos-1];     /* left node. */
        left->count++;          /* Insert the key from the parent.  */
        left->entry[left->count] = current->entry[pos];
        left->branch[left->count] = right->branch[0];

        for (c = 1; c <= right->count; c++)
    { /* Insert all keys from right node. */
          left->count++;
          left->entry[left->count] = right->entry[c];
          left->branch[left->count] = right->branch[c];
        }

        for (c = pos; c < current->count; c++)
          { /* Delete key from parent node. */
          current->entry[c] = current->entry[c+1];
          current->branch[c] = current->branch[c+1];
        }
        current->count--;
        free(right);    /* free the right node. */
}


/*
InOrder: inorder traversal of the B-Tree.
```

```
*/

void B::InOrder(node *root)
{
        int pos;

        if (root)
        {
                InOrder(root->branch[0]);
                for (pos = 1; pos <= root->count; pos++)
                {
                        cout<<"  "<<root->entry[pos].key;
                        InOrder(root->branch[pos]);
                }
        }
}

void main()
{
 int choice,targetpos;
 Type inKey;
 BT New;
 B obj;
 node *root, *target;
 root = NULL;
 while(1)
 {
  cout<<"\n\t\t Implementation of B-tree";
  cout<<"\n 1.Insert \n 2.Delete \n 3.Search \n 4.Display";
  cout<<"\n Enter Your choice";
  cin>>choice;
  switch(choice)
  {
   case 1:cout<<"Enter the Key to be inserted :";
          flushall();
          gets(New.key);
          root = obj.Insert(New, root);
          break;
   case 2:cout<<"Enter the Key to be deleted :";
          flushall();
                gets(New.key);
          cout<<"\n Deleting the desired item..."<<endl;
          obj.Delete(New.key, &root);
          break;
   case 3:cout<<"Enter the Key to be searched for :";
          flushall();
          gets(New.key);
          target = obj.Search(New.key, root, &targetpos);
          if (target)
          cout<<"The Searched Item: "<<target->entry[targetpos].key<<endl;
          else
          printf("Item is not present\n");
                break;
   case 4:cout<<"\n\nInOrder Traversal :\n";
```

```
                    obj.InOrder(root);
                        break;
    case 5:exit(0);
  }
 }
}
```