```
-----------------------------------------------------------------------------
-----------------------------------------
//ARRAYLIST IMPLEMENTATION



#include<iostream>
using namespace std;
class ArrayList {
  private:
    int SIZE;
    int length;
    int pos;
    int * Array;
    int * curr;
  public:
    ArrayList() {
      SIZE=10;
      Array= new int[SIZE];
      length=0;
      pos=0;
      curr= Array;
    }
    ~ArrayList() {
      delete []Array;
      delete curr;
    }
    void printArray() {
      if(length>0) {
        head();
        for(int x=0; x<length; x++)
          cout<<*curr++<<"\t";
      } else cout<<"Array is Empty"<<endl;
    }
    void InsertElement(int val) {
      if(!IsFull()) {
        head();
        curr= curr +length;
        *curr= val;
        length++;
      } else {
        cout<<"Array is Full"<<endl;
      }
    }
    void InsertAtPos(int val, int pos) {
      if (!IsFull())
      if (pos<=length&&pos>0) {
        tail();
        for (int i=length; i>=pos; i-- ) {
          *(curr+1)= *curr;
          back(); //curr= curr-1;
        }
        next();//curr= curr+1;
        *(curr)= val;
        length++;
      } else if (pos>length && pos<=SIZE) {
        head();
        curr= curr+pos-1;
```

```cpp
        *curr= val;
        length++;
      } else
        cout<<"Invalid Position"<<endl;
    }
    void reverseArray() {
      int *p1, *pn, temp;
      p1= Array;
      pn= Array+length-1;
      int val= length/2;
      for (int i=0; i<val; i++) {
        temp= *p1;
        *p1= *pn;
        *pn= temp;
        p1++;
        pn--;
      }
    }
    void deleteElement(int n) {
      if (!IsEmpty()) {
        int *ptr= Array;
        for (int x=0; x<length; x++) {
          if(*ptr==n) {
            int *ptr2= ptr;
            for (int j=x; j<length; j++) {
              ptr2++;
              *ptr= *ptr2;
              ptr++;
            }
            length--;
            break;
          }
          ptr++;
        }
      } else cout<<"Array is Empty, Delete operation failed"<<endl;
    }
    void deleteElementAtPos(int pos) {
      if (!IsEmpty()) {
        if (pos<=SIZE && pos>0){
          head();  //curr= &Array[0]
          curr = curr+pos-1;
          for (int x=0; x<=length-pos; x++){
            *(curr)= *(curr+1);
            next(); //curr= curr+1;
          }
            length--;
        }
      } else cout<<"Array is Empty, Delete operation failed"<<endl;
    }
    bool IsFull() {
      if (length==SIZE)
        return true;
      else return false;
    }
    bool IsEmpty() {
      if (length==0)
        return true;
      else return false;
    }
    void head() {
      curr= Array;
    }
    void tail() {
      curr= Array+length-1;
    }

    void back() {
      curr= curr-1;
    }
    void next() {
      curr= curr+1;
    }
    int Length() {
      return length;
    }
    void emptylist() {
      head();
      for (int x=0; x<SIZE; x++) {
        *curr++=0;
      }
    }
    void sortArray() {
      int *p1;
      int *p2, *temp;
      //sorting - ASCENDING ORDER
      for(int i=0; i<SIZE; i++) {
        p1 = Array+i;
        for(int j=i+1; j<SIZE; j++) {
          p2 = Array+j;
          if(*p1>*p2) {
            *temp  =*p1;
            *p1=*p2;
            *p2=*temp;
          }
        }

      }

    }
    //    void reverseArray() {
//      if(length>0) {
//
//        int * temp= Array+length-1;
//        int * tempA= new int [length-1];
//        int *ptr= tempA;
//
//        for(int x=0; x<length; x++) {
//          *ptr= *temp;
//          ptr++;
//          temp--;
//        }
//        ptr = tempA;
//        temp= Array;
//        for(int x=0; x<length; x++) {
//          *temp= *ptr;
//          ptr++;
//          temp++;
//        }
//      }
//    }
};

int main () {
  ArrayList *obj= new ArrayList();

  obj->emptylist();
  obj->InsertElement(1);
  obj->InsertElement(2);
  obj->InsertElement(3);
  obj->InsertElement(4);

  obj->printArray();cout<<endl;
  obj->InsertAtPos(99,2);

  obj->printArray();cout<<endl;
   obj->deleteElementAtPos(2);
   obj->reverseArray();

  obj->printArray();cout<<endl;
//  obj->InsertElement(1);
//  obj->InsertElement(2);
//  obj->InsertElement(3);
//  obj->InsertElement(4);
//  obj->InsertElement(5);
//  obj->InsertElement(6);
//  obj->InsertElement(7);
//  obj->InsertAtPos(23,1);
//  obj->InsertElement(8);
//  obj->InsertElement(9);
//  obj->InsertElement(10);
//  obj->InsertElement(11);
//  obj->InsertElement(12);
//  obj->InsertElement(13);
  //obj->printArray();
//  obj->deleteElement(1);
//  obj->deleteElement(2);
  cout<<endl;

  //obj->deleteElementAtPos(4);

  cout<<endl;
  //obj->emptylist();
   // obj->reverseArrayAdvanced();
  cout<<endl;
  return 0;
}


------------------------------------------------------------------------------------------------------------------

//Single Linklist implementation


```

```


#include<iostream>
using namespace std;
class node {
  public:
    int data;
    node *next;
};

node *head= new node();
node *curr= new node();
int length=0;
void GoToHead() { // set curr pointer to head node;
  curr= head;
}

void insertNodeAtEnd(int val) { // This function will insert new node at the end.
  GoToHead();
  node *t= new node();
  while(curr->next!=NULL)
    curr= curr->next;
  t->data= val;
  t->next= NULL;
  curr->next= t;
  length++;
}
void AddNodeBeforeHead( int val) { // This function will insert new node as a head.
  GoToHead();
  node *t= new node();
  t->data= val;
  t->next= curr;
  head= t;
  length++;
}
void InsertAfterSpecificKey(int val, int key ) {
  node *t= new node();
  GoToHead();
  while (curr!=NULL) {
    if (curr->data==key) {
      t->data= val;
      t->next= NULL;
      t->next= curr->next;
      curr->next= t;
      length++;
      break;
    }
    curr= curr->next;
  }
}
void InsertBeforeSpecificKey(int val, int key ) {
  node *ptr=NULL;
  GoToHead();
  while (curr!=NULL) {
    if (curr->data==key) {
      node *t= new node();
      t->data= val;
      t->next= NULL;
      t->next= curr;
      ptr->next= t;
      length++;
      break;
    }
    ptr= curr;
    curr= curr->next;
  }
}
void printLinklist() {
  GoToHead();
  while(curr!=NULL) {
    cout<<curr->data<<"\t";
    curr= curr->next;
  }
}

void DeleteNodeUsingKey(int key) {
  GoToHead();
  node *prenode= new node();
  if(curr->data== key) {
    head= curr->next;
    delete curr;
    length--;
    return;
  } else
    while(curr!=NULL) {
      if(curr->data==key) {
        prenode->next= curr->next;
        delete curr;
        length--;
        break;
      }
      prenode= curr;
      curr=curr->next;
    }

}
void DeleteNodeUsingPos(int pos) {
  GoToHead();
  node *prenode= new node();
  if(pos>length) {
    cout<<"This Position dosenot exist"<<endl;
    return;
  } else if (pos==1 ) { // if we want to delet head node
    prenode= curr;
    head= curr->next;
    delete prenode;
    length--;
  } else {
    for (int x=1; x<pos; x++) {
      prenode= curr;
      curr= curr->next;
    }
    prenode->next= curr->next;
    delete curr;
```

```cpp
    length--;

  }
}

void InsertNodeUsingKey(int val, int key, bool isBefore) {
  if (isBefore)
    InsertBeforeSpecificKey( val, key);
  else
    InsertAfterSpecificKey( val, key);

}
void InsertNodeUsingPos(int val, int pos, bool isBefore) {
  GoToHead();
  if(pos>length) {
    cout<<"This Position dosenot exist"<<endl;
    return;
  } else if (pos==1 && isBefore ) { // if we want to insert before head
    AddNodeBeforeHead(val);
  } else {
    node *prenode= new node();
    for (int x=1; x<pos; x++) {
      prenode= curr;
      curr= curr->next;
    }
    if (isBefore) {
      node *t= new node();
      t->data= val;
      t->next= NULL;
      t->next= curr;
      prenode->next= t;

    } else {
      node *t= new node();
      t->data= val;
      t->next= NULL;
      t->next= curr->next;
      curr->next= t;
    }
  }

}
int main () {
  head->data= 1;
  head->next=NULL;

  insertNodeAtEnd(2);
  insertNodeAtEnd(3);
  insertNodeAtEnd(4);
  printLinklist();
  cout<<endl;

  InsertAfterSpecificKey(99, 2);
  printLinklist();
  cout<<endl;

  DeleteNodeUsingKey(99);
  printLinklist();
  cout<<endl;

  InsertBeforeSpecificKey(99, 2);
  printLinklist();
  cout<<endl;

  InsertNodeUsingPos(88,1,true);
  printLinklist();
  cout<<endl;

  DeleteNodeUsingPos(1);
  DeleteNodeUsingPos(2);

  printLinklist();
  cout<<endl;
  return 0;
}


------------------------------------------------------------------------------------------------------------------
//DOUBLY LINK



#include<iostream>
using namespace std;
class Node{
    public:
    int data;
    Node*next;
    Node*prev;
    Node(int s){data=s;
    next=prev=NULL;}
};

class DLinkList{
    private:
    Node*head;
    int length;
    public:
    DLinkList(){head=NULL;
    length=0;}
    void insertHead(int valve){Node*t=new Node(valve);
    if(head==NULL){head=t;
    return;}
    t->next=head;
    head->prev=t;
    head=t;
    length++;}
    // void insertEnd(int valve){//}
    void insertSpecific(int valve,int pos){
        if(pos<1||pos>length+1){cout<<"Invalid Position"<<endl;
        return;}
        Node*temp=head;
        Node*p=new Node(valve);
        if(pos==1){
            insertHead(valve);}
```

```cpp
        else{for(int i=1;i<pos;i++){
        temp=temp->next;}
        p->next=temp->next;
        p->prev=temp;
        temp->next->prev=p;
        temp->next=p;
        length++;
        }}
    void deletion(int valve){Node*temp;
    if(valve>length){
        cout<<"Invalid Pos"<<endl;
        return;
    }
    temp=head;
    if(valve==1){
        head=head->next;
        temp=head;
    }
    while(temp->next->data!=valve){
        temp=temp->next;}
        temp->next->next->prev=temp;
        temp->next=temp->next->next;}
    void print(){bool flag;
    cout<<"Press 0 to print in Ascending and 1 to print in Descending ";
    cin>>flag;
    if(flag==1){
    Node*curr=head;
    while(curr!=NULL){
    cout<<curr->data<<endl;
    curr=curr->next;}}
    if(flag==0){Node*curr=head;
    while(curr->next!=NULL){curr=curr->next;}
    while(curr!=NULL){
        cout<<curr->data<<endl;
        curr=curr->prev;}
    }}
};
int main(){DLinkList List1;
// List1.insertHead(2);
// List1.insertHead(3);
// List1.insertHead(9);
// List1.insertHead(10);
// List1.insertHead(12);
List1.insertSpecific(1,1);
List1.insertSpecific(2,1);
List1.insertSpecific(3,1);
List1.insertSpecific(4,1);
List1.insertSpecific(5,1);
List1.insertSpecific(6,1);
List1.print();
List1.print();
//cout<<endl;
//cout<<"To insert at end, give position 1 in the perimeter:"<<endl;
List1.deletion(5);
List1.print();}



```

```
------------------------------------------------------------------------------------------------------------------
//CIRCULAR LINK




#include<iostream>
using namespace std;
class node{
    public:
    int data;
    node*next;
    node(int valve){
        data=valve;
        next=NULL;
    }
};
class circular{
    public:
    node*head;
    int length;
    circular(){
        head=NULL;
        length=0;
    }
    void insert(int value){
        if(head==NULL){
            node*n=new node(value);
            head=n;
            head->next=head;
            return;
        }
        node*n=new node(value);
        node*temp=head;
        while(temp->next!=head){
            temp=temp->next;
        }
        n->next=head;
        head=n;
        temp->next=head;
        return;
    }
    // void deletion(int )
    void print(){
        node*temp;

        temp=head;
        while(temp->next!=head){
            cout<<temp->data;
            temp=temp->next;
        }
    }
    void deletion(){
        if(head==NULL){
            cout<<"nothing to delete";
            return;
        }
        node*temp=head;
        while(temp->next!=head){
            temp=temp->next;
        }
        head=head->next;
        temp->next=head;
    }
};
int main(){
    circular obj1;
    obj1.insert(5);
    obj1.insert(5);
    obj1.insert(5);
    obj1.insert(5);
    obj1.insert(5);
    obj1.deletion();
    obj1.deletion();
    obj1.deletion();
    // obj1.insert(5);
    obj1.print();
};



--------------------------------------------------------------------------------------------------------------------
//STACK USING ARRAY




#include<iostream>
using namespace std;
#define SIZE 100
class StackArr{
    private:
    int top;
    public:
    int arr[SIZE];
    StackArr(){
        top = -1;
        int arr[SIZE];
    }
    void pop(){
        if(top==-1){
            cout<<"Stack Underflows";
            return;
        }
        cout<<arr[top]<<endl;
        top--;
    }
    void push(int valve){
        if(top>SIZE){
            cout<<"Stack Overflows";
            return;
        }
        top++;
```

```cpp
        arr[top]=valve;
    }
    void display(){
        for(int i=top;top>=0;i--){
            cout<<arr[top]<<endl;
            top--;
        }
    }
    int peek(){
        if(top==-1){
            cout<<"Stack is empty"<<endl;
            return 0;
        }
        return arr[top];
    }
    void isEmpty(){
        if(top==-1){
            cout<<"Stack is empty"<<endl;
        }
        return;
    }
};
int main(){
    StackArr obj1;
    //obj1.isEmpty();
    //obj1.display();
    //obj1.peek();
    //obj1.push(2);
    //cout<<obj1.peek();
    obj1.push(4);
    obj1.push(7);
    obj1.push(8);
    //obj1.display();
    obj1.pop();
    obj1.pop();
    cout<<obj1.peek()<<endl;
    //obj1.pop();
    //obj1.pop()
    obj1.isEmpty();
    obj1.display();
}
------------------------------------------------------------------------------------------------------------------
//STACK USING LINKLIST



#include<iostream>
using namespace std;
class Node{
    public:
    int data;
    Node*next;
    Node(int valve){
        data=valve;
        next=NULL;
    }
};
```

```
class Stack{
    private:
    Node*head;
    int length;
    public:
    Stack(){
        head=NULL;
        length=0;
    }
    void push(int vault){
        Node*n=new Node(vault);
            n->next=head;
            n->data = vault;
            head=n;
        }
    void pop(){
        Node*temp=head;
        cout<<head->data<<endl;
        head=head->next;
        delete temp;
    }
    void peek(){
        cout<<head->data;
    }
    void IsEmpty(){
        if(head==NULL){
            cout<<"Empty";
        }
        else{
            cout<<"It is not empty"<<endl;
        }
    }
    void display(){
        if(head==NULL){
            cout<<"Stack is empty";
        }
        else{
        Node*temp=head;
        while(temp!=NULL){
            cout<<temp->data<<endl;
            temp=temp->next;
        }
    }
    }
};
int main(){
    Stack obj1;
    //obj1.push(5);
    //obj1.push(6);
    obj1.push(9);
    obj1.push(15);
    obj1.push(19);
    obj1.peek();
    cout<<endl;
    obj1.IsEmpty();
    obj1.display();
    obj1.pop();
    obj1.pop();
    //obj1.pop();
    //obj1.pop();
}
------------------------------------------------------------------------------------------------------------------
//QUEUE USING ARRAY



#include<iostream>
using namespace std;
class Queue{
    private:
    int *arr;
    int front;
    int rear;
    int size;
    int noofelements;
    public:
    Queue(int s){
        arr=new int[s];
        size=s;
        front=0;
        rear=-1;
        noofelements=0;
    }
    void enqueue(int val){
        if(isFull()){
            cout<<"Queue overflow"<<endl;
            return;
        }
        if(rear==(size-1))
        rear=0;
        else
        rear++;
        arr[rear]=val;
        noofelements++;
    }
    bool isFull(){
        if(noofelements==size)
        return true;
        else
        return false;
    }
    int dequeue(){
        if(isEmpty()){
            cout<<"Queue Underflow"<<endl;
            return 0;
        }
        int val=arr[front];
        if(front==(size-1))
        front=0;
        else
        front++;
        noofelements--;
        return val;
    }
    bool isEmpty(){
        if(noofelements==0)
        return true;
        else
        return false;
    }
    void definition(){
    }
};
int main(){
    Queue obj1(100);
    obj1.enqueue(4);
    obj1.enqueue(8);
    cout<<obj1.dequeue();
    cout<<obj1.dequeue();
    //cout<<obj1.dequeue();
}
------------------------------------------------------------------------------------------------------------------
//QUEUE USING LINK LIST




#include<iostream>
using namespace std;
class Node{
    public:
    int data;
    Node*next;
    Node(int valve){
        data=valve;
        next=NULL;
    }
};
class QueueL{
    private:
    Node*head;
    Node*front;
    Node*rear;
    int length;
    public:
    QueueL(){
        head=NULL;
        length=0;
    }
    void Enqueue(int vault){
        /*if(isFull()){
            cout<<"Queue overflows"<<endl;
            return 0;
        }*/
        Node *n=new Node(vault);
        if(head==NULL){
            head=n;
            front=head;
            rear=head;
            length++;
        }
        else{
            rear->next=n;
            rear=n;
            length++;
        }
    }
    bool isEmpty(){
        if(head==NULL)
        return true;
        else
        return false;
    }
    void Dequeue(){
        if(isEmpty()){
            cout<<"Queue Underflows";
            return;
        }
        Node*vamp;
        vamp=front;
        front=front->next;
        cout<<vamp->data;
        delete vamp;
    }
};
int main(){
    QueueL obj1;
    obj1.Enqueue(2);
    obj1.Enqueue(4);
    obj1.Dequeue();
    cout<<endl;
    obj1.Dequeue();
    cout<<endl;
}




------------------------------------------------------------------------------------------------------------------
//BST IMPLEMENTATION



#include<iostream>
#include <bits/stdc++.h>
using namespace std;
class Node{
    public:
    int data;
    Node*left;
    Node*right;
    Node(int data){
        this->data=data;
        left=right=NULL;
    }
};
class BinarySearchTree{
    public:
    Node*root;
    BinarySearchTree(){
        root=NULL;
    }
    bool searchNode(int num);
    Node*insert(Node*root,int val);
    void remove(Node*root,int val);
    void inOrderTraversal(Node*root);
    void preOrderTraversal(Node*root);
    void postOrderTraversal(Node*root);
    void makeDeletion(Node*&nodePtr);
    int getLeafCount(Node* node);
    void Merging(BinarySearchTree tree);
};
int main(){
    BinarySearchTree tree;
    BinarySearchTree Stree;
    tree.insert(tree.root,10);
    tree.insert(tree.root,8);
    tree.insert(tree.root,6);
    tree.insert(tree.root,9);
    tree.insert(tree.root,15);
    tree.insert(tree.root,14);
    tree.insert(tree.root,20);

    //tree.insert(tree.root,5);
    //tree.insert(tree.root,17);
    //tree.insert(tree.root,25);
    //tree.insert(tree.root,14);
    //tree.insert(tree.root,20);
    //Node*Anroot=tree.root->left->left;
    //tree.makeDeletion(tree.root);

    /*
    Stree.insert(tree.root,11);
    Stree.insert(tree.root,22);
    Stree.insert(tree.root,7);
    Stree.insert(tree.root,25);
    */

    Stree.Merging(tree);
    tree.makeDeletion(tree.root->left->left);
    cout<<"\n In-Order"<<endl;
    cout<<"Left---Root---Right"<<endl;
    tree.inOrderTraversal(tree.root);

    cout<<"\n Pre-Order"<<endl;
    cout<<"Root---Left---Right"<<endl;
    tree.preOrderTraversal(tree.root);

    cout<<"\n Post-Order"<<endl;
    cout<<"Left---Right---Root"<<endl;
    tree.postOrderTraversal(tree.root);
    cout<<"\n\nThe Tree Leaf Count Is: ";
    cout<<tree.getLeafCount(tree.root)<<"\t";
    cout<<endl;
    if(tree.searchNode(29)){
        cout<<"Value Found";
    }
    else{
        cout<<"Not found";
    }
    return 0;
}
bool BinarySearchTree::searchNode(int num){
  Node *nodePtr = root;
  while (nodePtr)
  {
    if (nodePtr->data == num)
      return true;
    else if (num < nodePtr->data)
      nodePtr = nodePtr->left;
    else
      nodePtr = nodePtr->right;
  }
  return false;
}
Node*BinarySearchTree::insert(Node*r,int val){
    if(r==NULL){
        Node*t=new Node(val);
        if(r==root){
            root=r=t;
        }
        else{
        r=t;}
        return r;
    }
    else if(val==r->data){
        cout<<"Duplicate Data: "<<val<<endl;
    }
    else if(val<r->data){
        r->left=insert(r->left,val);
    }
    else if(val>r->data){
        r->right=insert(r->right,val);
    }
    return r;
}
void BinarySearchTree::inOrderTraversal(Node*r){
    if(r==NULL){
        return;
    }
    inOrderTraversal(r->left);
    cout<<" "<<r->data<<" ->";
    inOrderTraversal(r->right);
}
void BinarySearchTree::preOrderTraversal(Node*r){
    if(r==NULL){
        return;
    }
    cout<<" "<<r->data<<" ->";
    inOrderTraversal(r->left);
    inOrderTraversal(r->right);
}
void BinarySearchTree::postOrderTraversal(Node*r){
    if(r==NULL){
        return;
    }
    inOrderTraversal(r->left);
    inOrderTraversal(r->right);
    cout<<" "<<r->data<<" ->";
}
void BinarySearchTree::makeDeletion(Node*&nodePtr)
{
  Node*tempNodePtr;
  if (nodePtr == NULL)
    cout << "Cannot delete empty node.\n";
  else if (nodePtr->right == NULL)
  {
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->left;
    delete tempNodePtr;
  }
    else if (nodePtr->left == NULL)
  {
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->right;
    delete tempNodePtr;
  }
  else
  {
    tempNodePtr = nodePtr->right;
    while (tempNodePtr->left)
    tempNodePtr = tempNodePtr->left;
    tempNodePtr->left = nodePtr->left;
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->right;
    delete tempNodePtr;
  }
}
int BinarySearchTree::getLeafCount(Node* root)
{
  if(root == NULL)
    return 0;
  if(root->left == NULL && root->right == NULL)
    return 1;
  else
    return getLeafCount(root->left)+getLeafCount(root->right);
}
void BinarySearchTree::Merging(BinarySearchTree tree){
    if(root==NULL){
        return;
    }
    else{
        tree.inOrderTraversal(tree.root->left);
        insert(root,root->data);
        tree.inOrderTraversal(tree.root->right);}
    }


-------------------------------------------------------------------------------------------------------------
//BST ADEEL IMPLEMENTATION


```

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

class IntBinaryTree
{
private:
  struct TreeNode{
    int value;
    TreeNode *left;
    TreeNode *right;
  };
  TreeNode *root;

    // void tree_clear(TreeNode* nodeptr)
    // {
  // if (nodeptr != NULL) {
  //     tree_clear( nodeptr->left );
  //     tree_clear( nodeptr->right );
  //     delete nodeptr;
  // }
    // }
  void tree_clear(TreeNode *&);
  void deleteNode(int, TreeNode *&);
  void makeDeletion(TreeNode *&);
  void displayInOrder(TreeNode *);
public:
        IntBinaryTree()   // Constructor
    { root = NULL; }
  // ~IntBinaryTree() // Destructor
  //  { tree_clear(root); }
        // void tree_clear(TreeNode* nodeptr);
  void insertNode(int);
  bool searchNode(int);
  void remove(int);
  void showNodesInOrder(void)
    { displayInOrder(root); }
};
bool IntBinaryTree::searchNode(int num)
{
  TreeNode *nodePtr = root;

  while (nodePtr)
  {
    if (nodePtr->value == num)
      return true;
    else if (num < nodePtr->value)
      nodePtr = nodePtr->left;
    else
      nodePtr = nodePtr->right;
  }
  return false;
}
void IntBinaryTree::makeDeletion(TreeNode *&nodePtr)
{
  TreeNode *tempNodePtr;  // Temporary pointer, used in
                        // reattaching the left subtree.

  if (nodePtr == NULL)
```

```
    cout << "Cannot delete empty node.\n";
  else if (nodePtr->right == NULL)
  {
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->left; // Reattach the left child
    delete tempNodePtr;
  }
      else if (nodePtr->left == NULL)
  {
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->right; // Reattach the right child
    delete tempNodePtr;
  }
  // If the node has two children.
  else
  {
    // Move one node the right.
    tempNodePtr = nodePtr->right;
    // Go to the end left node.
    while (tempNodePtr->left)
      tempNodePtr = tempNodePtr->left;
    // Reattach the left subtree.
    tempNodePtr->left = nodePtr->left;
    tempNodePtr = nodePtr;
    // Reattach the right subtree.
    nodePtr = nodePtr->right;
    delete tempNodePtr;
  }
}



void IntBinaryTree::deleteNode(int num, TreeNode *&nodePtr)
{
  if (num < nodePtr->value)
    deleteNode(num, nodePtr->left);
  else if (num > nodePtr->value)
    deleteNode(num, nodePtr->right);
  else
    makeDeletion(nodePtr);
}

void IntBinaryTree::displayInOrder(TreeNode *nodePtr)
{
  if (nodePtr)
  {
    displayInOrder(nodePtr->left);
    cout<< nodePtr->value << endl;
    displayInOrder(nodePtr->right);
  }
}

void IntBinaryTree::insertNode(int num)
{
  TreeNode *newNode,  // Pointer to a new node
           *nodePtr;  // Pointer to traverse the tree

  // Create a new node
```

```cpp
  newNode = new TreeNode;
  newNode->value = num;
  newNode->left = newNode->right = NULL;

  if (!root)  // Is the tree empty?
    root = newNode;
  else
  {
    nodePtr = root;
            while (nodePtr != NULL)
    {      if (num < nodePtr->value)
      {      if (nodePtr->left)
          nodePtr = nodePtr->left;
        else
        {     nodePtr->left = newNode;
          break;
        }
      }
      else if (num > nodePtr->value)
      {      if (nodePtr->right)
          nodePtr = nodePtr->right;
        else
        {      nodePtr->right = newNode;
          break;
        }
      }
      else
      {   cout << "Duplicate value found in tree.\n";
                break;
      }
    }
  }
}


int main()
{
  IntBinaryTree tree;

  cout << "Inserting nodes.\n";
  tree.insertNode(5);
  tree.insertNode(8);
  tree.insertNode(3);
  tree.insertNode(12);
  tree.insertNode(9);
  if (tree.searchNode(3))
    cout << "3 is found in the tree.\n";
  else
    cout << "3 was not found in the tree.\n";

    // IntBinaryTree tree;

  // cout << "Inserting nodes. ";
  // tree.insertNode(5);
  // tree.insertNode(8);
  // tree.insertNode(3);
  // tree.insertNode(12);
  // tree.insertNode(9);
  // cout << "Done.\n";
}

------------------------------------------------------------------------------------------------------------------
// Binary Search Tree Implementation..  //SIR KHURRAM
// @KS.
#include<iostream>
using namespace std;

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

    Node* insert( Node* root, int val);
    Node* DeleteNodeInBST(Node* root,int data);
    Node* inOrderTraversal( Node* root);
    Node* preOrderTraversal( Node* root);
    Node* postOrderTraversal( Node* root);
    Node* merge( Node* r1, Node* r2);
    Node* FindMax(Node* root);
    int leafCount (Node* root);
    int treeHeight(Node *root);
};

int main (){
    BinarySearchTree tree1, tree2;

      tree1.insert(tree1.root,10);
    tree1.insert(tree1.root, 8);
    tree1.insert(tree1.root, 6);
    tree1.insert(tree1.root, 9);
    tree1.insert(tree1.root, 15);
    tree1.insert(tree1.root, 14);
    tree1.insert(tree1.root, 20);

//    tree.DeleteNodeInBST(tree.root ,9);


    cout<<"In Order Print (left--Root--Right)"<<endl;
    tree1.inOrderTraversal(tree1.root);

    cout<<"\n-----------------------"<<endl;
    cout<<"Pre Order Print (Root--left--Right)"<<endl;

    tree1.preOrderTraversal(tree1.root);

    cout<<"\n----------------------"<<endl;
    cout<<"Post Order Print (left--Right--Root)"<<endl;

    tree1.postOrderTraversal(tree1.root);
    cout<<"\n\nThe total leaf node in tree are: "<<
tree1.leafCount(tree1.root);

    cout<<"\n\nThe height of root node is : "<<
tree1.treeHeight(tree1.root);

    // Merge .

    tree2.insert(tree2.root, 7);
    tree2.insert(tree2.root, 33);

    tree1.merge(tree2.root, tree1.root);
    cout<<"\n\nAfter Merging"<<endl;
    cout<<"In Order Print (left--Root--Right)"<<endl;

    tree1.inOrderTraversal(tree1.root);
    cout<<"\n\nThe total leaf node in tree are: "<<
tree1.leafCount(tree1.root);

    cout<<"\n\nThe height of root node is : "<<
tree1.treeHeight(tree1.root);

    return 0;
}

Node* BinarySearchTree::FindMax(Node* r){

    while(r->right!=NULL){
        r= r->right;
    }
    return r;

}

Node* BinarySearchTree::insert(Node* r, int val ){

 if (r==NULL)
    {
        Node* t= new Node(val);

        if (r==root)
        root= r=t;
        else
        r=t;

        return r;
    }
//    else if (r->data== val){
//        //cout<<"Duplicate Record  "<<val;
//            return r;
//    }
    else if (val < r->data)
```

```
        r->left = insert(r->left , val );

    else if (val > r->data)
        r->right= insert( r->right,val);

}
Node * BinarySearchTree::DeleteNodeInBST(Node* root, int data)
{

    if(root==NULL)
     return root;
    else if(data<root->data)
        root->left = DeleteNodeInBST(root->left, data);
    else if (data> root->data)
        root->right = DeleteNodeInBST(root->right, data);
    else
    {
        //No child
        if(root->right == NULL && root->left == NULL)
        {
            delete root;
            root = NULL;
            return root;
        }
        //One child on left
        else if(root->right == NULL)
        {
            Node* temp = root;
            root= root->left;
            delete temp;
        }
        //One child on right
        else if(root->left == NULL)
        {
            Node* temp = root;
            root= root->right;
            delete temp;
        }
        //two child
        else
        {
            Node* temp = FindMax(root->left);
            root->data = temp->data;
            root->left = DeleteNodeInBST(root->left, temp->data);
        }
    }
    return root;
}


Node * BinarySearchTree::inOrderTraversal( Node* r){
     if (r == NULL)
        return NULL;
    /* first recur on left child */
    inOrderTraversal(r->left);
    /* then print the data of node */
    cout << " "<< r->data << " -> ";
    /* now recur on right child */
    inOrderTraversal(r->right);

}

Node* BinarySearchTree::preOrderTraversal( Node* r){
     if (r == NULL)
        return NULL;

    cout << " "<< r->data << " -> ";
    preOrderTraversal(r->left);
    preOrderTraversal(r->right);
}
Node* BinarySearchTree::postOrderTraversal( Node* r){
     if (r == NULL)
        return NULL;
    postOrderTraversal(r->left);
    postOrderTraversal(r->right);
    cout << " "<< r->data << " -> ";
}

int BinarySearchTree::leafCount(Node * r){
    int static count= 0;
    if(r == NULL)
        return 0;
    else if(r->left == NULL && r->right == NULL)
        return 1;

    return count + leafCount(r->left) + leafCount(r->right);
}

int BinarySearchTree::treeHeight(Node *root)
{
    int static l_height=0;
    int static r_height=0;
    if (root == NULL)
        return -1;
    else
    {
    l_height = treeHeight(root->left);
       r_height = treeHeight(root->right);
        if (l_height > r_height)
            return (l_height + 1);
        else
            return (r_height + 1);
    }
}
// This method will merge tree1 into tree2
Node * BinarySearchTree::merge( Node* r1, Node* r2){
     if (r1 == NULL)
        return NULL;
    /* first recur on left child */
    merge(r1->left, r2);

    insert(r2, r1->data);
    /* now recur on right child */
    merge(r1->right, r2);

}
```

```

------------------------------------------------------------------------------------------------------------------
//BST TO AVL



#include<iostream>
using namespace std;
class node{
  public:
    node *left;
    node*right;
    int data;
    int height;
    node(int data)
    {
      this->data=data;
      height=0;
      left=right=NULL;
    }
};
class AVLtree{
  private:
  node*root;
  void makeEmpty(node* t);
  node* insert(int x,node*t);
  node* singleleftrotate(node* &C);
  node* singlerightrotate(node*&C);

  node* doubleleftrightrotate(node* &C);
  node* doublerightleftrotate(node* &C);

  node*findmin(node*t);
  node*findmax(node *t);

  node *remove(int x,node*t);
  int height(node*t);
  int getBalance(node*t);
  void inorder(node *t);

  public:
    AVLtree()
    {
      root=NULL;
    }
    void insert(int x){
      root=insert(x,root);
    }
    void remove(int x)
    {
      root=remove(x,root);
    }
    void display()
    {
      inorder(root);
      cout<<endl;
    }
```

```


};

int main()
{
  AVLtree tree;
  tree.insert(3);
  tree.insert(4);
  tree.insert(5);
  tree.insert(6);
  tree.insert(7);
  tree.display();
  return 0;
}

node* AVLtree::singleleftrotate(node* &A)
{
node* newRoot = A->right;
A->right = newRoot->left;
newRoot->left = A;
A->height = max(height(A ->left), height(A ->right)) + 1;
newRoot ->height = max(height(newRoot->right), A->height) + 1;
return newRoot;
}

node* AVLtree::singlerightrotate(node* &C)
{
node* newRoot = C->left;
C->left = newRoot->right;
newRoot->right = C;
C->height = max(height(C ->left), height(C ->right)) + 1;
newRoot ->height = max(height(newRoot->left), C->height) + 1;
return newRoot;
}

node* AVLtree::doubleleftrightrotate(node*& t)
{
t->left = singleleftrotate(t->left);
return singlerightrotate(t);
}

node* AVLtree::doublerightleftrotate(node*& t)
{
t->right = singlerightrotate(t->right);
return singleleftrotate(t);
}

void AVLtree::inorder(node *t)
{
  if(t==NULL)
  return;
  inorder(t->left);
  cout<<t->data<<" ->";
  inorder(t->right);
}
int AVLtree::height(node* t)
{
  return(t==NULL ? -1 : t->height);
}
int AVLtree::getBalance(node*t)
{
  if(t==NULL)
  return 0;
  else
  return height(t->left) - height(t->right);
}


node *AVLtree::findmin(node *t)
{
  if(t==NULL)
  return NULL;
  else if(t->left==NULL)
    return  t;
  else
    return findmin(t->left);
}

node *AVLtree::findmax(node *t)
{
  if(t==NULL)
      return NULL;
  else if(t->right==NULL)
    return  t;
  else
    return findmax(t->right);
}
void AVLtree::makeEmpty(node* t) {
        if(t == NULL)
            return;
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
    }

node*  AVLtree::  insert(int x, node* t)
    {
        if(t == NULL)
        {
            t = new node (x);
        }
        else if(x < t->data)
        {
            t->left = insert(x, t->left);
            if(height(t->left) - height(t->right) == 2)
            {
                if(x < t->left->data)
                    t = singlerightrotate(t);
                else
                    t = doubleleftrightrotate(t);
            }
        }
        else if(x > t->data)
        {
            t->right = insert(x, t->right);
            if(height(t->right) - height(t->left) == 2)
            {
                if(x > t->right->data)
                    t = singleleftrotate(t);
                else
                    t = doublerightleftrotate(t);
            }
        }

        t->height = max(height(t->left), height(t->right))+1;
        return t;
    }

    node* AVLtree::remove(int x, node* t)
    {
        node* temp;

        // Element not found
        if(t == NULL)
            return NULL;

        // Searching for element
        else if(x < t->data)
            t->left = remove(x, t->left);
        else if(x > t->data)
            t->right = remove(x, t->right);

        // Element found
        // With 2 children
        else if(t->left && t->right)
        {
            temp = findmin(t->right);
            t->data = temp->data;
            t->right = remove(t->data, t->right);
        }
        // With one or zero child
        else
        {
            temp = t;
            if(t->left == NULL)
                t = t->right;
            else if(t->right == NULL)
                t = t->left;
            delete temp;
        }
        if(t == NULL)
            return t;

        t->height = max(height(t->left), height(t->right))+1;

        // If node is unbalanced
        // If left node is deleted, right case
        if(height(t->left) - height(t->right) == 2)
        {
            // right right case
            if(height(t->left->left) - height(t->left->right) == 1)
                return singleleftrotate(t);
            // right left case
```

```
            else
                return doublerightleftrotate(t);
        }
        // If right node is deleted, left case
        else if(height(t->right) - height(t->left) == 2)
        {
            // left left case
            if(height(t->right->right) - height(t->right->left) == 1)
                return singlerightrotate(t);
            // left right case
            else
                return doubleleftrightrotate(t);
        }
        return t;
    }
------------------------------------------------------------------------------------------------------------------
//HASHIN LINEAR MAHAD



#include<iostream>
#include<string>

using namespace std;

class Students{
    public:
int rollNo;
// string name;

Students(){

}

};

class Hashtable {
Students **arr;
int size;
int count;
public:

Hashtable(int s){
size = s;
count = 0;
arr = new Students*[size];

for(int i =0 ; i<size ; i++)
arr[i] = NULL;
}

int hashin(int n){
 return n%size;
}


void insert(int key){  /// ,string value
```

```
    if(count == size){
    cout<<"hash is full";
    return;
    }

    int hashindex = hashin(key);
    while(arr[hashindex] != NULL){
        hashindex = (hashindex +1) %size;
    }
    arr[hashindex] = new  Students();
    arr[hashindex]->rollNo = key;
    // arr[hashindex]->name = value;
    count++;
}

int search (int key){
    if(count == 0){
        cout<< "empty";
    }
    int hashindex = hashin(key);
    int temp = hashindex;
    while(true){
        if(arr[hashindex] == NULL)
        hashindex = (hashindex +1)%size;
        else if(arr[hashindex]->rollNo != key)
        hashindex = (hashindex +1) %size;
        else
        break;

        if(hashindex == temp){
            temp = -1;
            break;
        }
  }
    if(temp == -1)
    cout<< "element not found";

     else
     cout<<"element found  ["<< arr[hashindex]->rollNo<<"]";


}

void deleteitem(int key){

    if(count == 0){
    cout<<"hash is empty";
    }

    int hashindex = hashin(key);
    int temp = hashindex;
    while(true){
       if(arr[hashindex] == NULL)
        hashindex = (hashindex +1)%size;
        else if(arr[hashindex]->rollNo != key)
        hashindex = (hashindex +1) %size;
        else
        break;

        if(hashindex == temp){
            temp = -1;
            break;
        }

    }
    if(temp == -1)
    cout<<"not found";

    else{
        delete arr[hashindex];
        arr[hashindex] = NULL;
    }

}

void displayitem(){

    for(int i = 0 ; i<size ; i++){
        if(arr[i]!= NULL)
        cout<<"Hash table ["<<i<<"] : key  "<<arr[i]->rollNo<<endl; // arr[i]->name
    }
}



// ~Hashtable(){

//     for(int i = 0 ; i<size ; i++){
//         if(arr[i]!= NULL){
//             cout<<"deleting key"<<arr[i]->rollNo<<"value" <<arr[i]->name<<endl;
//             delete arr[i];
//             arr[i] = NULL;
//         }
//     }
// }


};

int main(){

    Hashtable mt(25);

mt.insert(652 );
mt.insert(65402 );
mt.insert(65405 );
mt.insert(65403 );
mt.displayitem();
mt.getitem(6542);

return 0;
}
-------------------------------------------------------------------------------------------------------------------
```

```
//HASHING SIR KHURRAM



#include<iostream>
#include<list>
using namespace std;
class HashTable{
    int capacity;
    list<int> *table;
    public:
    HashTable(int V);
    void insertItem(int key, int data);
    void deleteItem(int key);
    int checkPrime(int n){
        int i;
        if(n==1 || n==0){
            return 0;
        }
        for(int i=2;i<n/2;i++){
            if(n%i==0){
                return 0;
            }
        }
        return 1;
    }
    int getPrime(int n){
        if(n%2==0){
            n++;
        }
        while(!checkPrime(n)){
            n+=2;
        }
        return n;
    }
    int hashFunction(int key){
        return (key%capacity);
    }
    void displayHash();
};
HashTable::HashTable(int c){
    int size=getPrime(c); //OR      int size=c*2
    this->capacity=size;
    table=new list<int>[capacity];
}
void HashTable::insertItem(int key,int data){
    int index=hashFunction(key);
    table[index].push_back(data);
}
void HashTable::deleteItem(int key){
    int index=hashFunction(key);
    list<int>::iterator i;
    for(i=table[index].begin();i!=table[index].end();i++){
        if(*i==key)
        break;
    }
    if(i!=table[index].end())
    table[index].erase(i);
```

```
}
void HashTable::displayHash(){
    for(int i=0;i<capacity;i++){
        cout<<"table[" <<i<<"]";
        for(auto x:table[i])
        cout<<" --> "<<x;
        cout<<endl;
    }
}
int main(){
    int key[]={231,321,212,321,433,262};
    int data[]={123,432,523,43,423,111};
    int size=sizeof(key)/sizeof(key[0]);

    HashTable h(size);
    //HashTable h(12);
    for(int i=0;i<size;i++){
        h.insertItem(key[i],data[i]);
    }
    h.deleteItem(12);
    h.displayHash();

    return 0;
}

------------------------------------------------------------------------------------------------------------------
//DUPLICATION OF NODES USING QUEUE



#include<iostream>
using namespace std;
class Node{
    public:
    int data;
    Node*next;
    Node(int valve){
        data=valve;
        next=NULL;
    }
};
class QueueL{
    private:
    Node*head;
    Node*front;
    Node*rear;
    int length;
    public:
    QueueL(){
        head=NULL;
        length=0;
    }
    int defination(){
        int var1=DequeueLastBackup();
        int var2=DequeueLastBackup();
        int var3=DequeueLastBackup();
        for(int i=0;i<var1;i++){
            Enqueue(var1);
        }
        for(int i=0;i<var2;i++){
            Enqueue(var2);
        }
        for(int i=0;i<var3;i++){
            Enqueue(var3);
        }
        return 0;
    }
    bool isFull(){
        if(head==NULL){
            return true;
        }
    }
    void Enqueue(int vault){
        /*if(isFull()){
            cout<<"Queue overflows"<<endl;
            return 0;
        }*/
        Node *n=new Node(vault);
        if(head==NULL){
            head=n;
            front=head;
            rear=head;
        }
        else{
            rear->next=n;
            rear=n;
        }
    }
    bool isEmpty(){
        if(front==NULL)
        return true;
        else
        return false;
    }
    void Dequeue(){
        if(isEmpty()){
            cout<<"Queue Underflows";
            return;
        }
        Node*vamp;
        vamp=front;
        front=front->next;
        cout<<vamp->data;
        delete vamp;
    }
    int DequeueBackup(){
        if(isEmpty()){
            cout<<"Queue Underflows";
            return 0;
        }
        Node*vamp;
        vamp=front;
        front=front->next;
        cout<<vamp->data;
        return vamp->data;
    }
    int DequeueLastBackup(){
        if(isEmpty()){
            cout<<"Queue Underflows";
            return 0;
        }
        Node*vamp,*vent;
        vamp=front;
        front=front->next;
        cout<<vamp->data;
        return vamp->data;
    }
    void Duplicate(){
        int var1=DequeueMana();
        int var2=DequeueMana();
        int var3=DequeueMana();
        for(int i=0;i<var1;i++){
            Enqueue(var1);
        }
        for(int i=0;i<var2;i++){
            Enqueue(var2);
        }
        for(int i=0;i<var3;i++){
            Enqueue(var3);
        }
    }
    int DequeueMana(){
        if(isEmpty()){
            cout<<"Queue Underflows";
            return 0;
        }
        Node*vamp;
        int mango;
        vamp=front;
        front=front->next;
        //cout<<vamp->data;
        mango=vamp->data;
        delete vamp;
        return mango;
    }
};
int main(){
    QueueL obj1;
    obj1.Enqueue(3);
    obj1.Enqueue(4);
    obj1.Enqueue(5);
    obj1.Duplicate();
    //obj1.Dequeue();
    //obj1.Dequeue();
    //obj1.defination();
    //int var1=obj1.DequeueMana();
    //int var2=obj1.DequeueMana();
    //obj1.Dequeue();
    //obj1.DequeueMana();
    //obj1.Dequeue();
    //int var1=obj1.DequeueBackup();
    //int var2=obj1.DequeueBackup();
    //int var3=obj1.DequeueBackup();
    //cout<<var1<<var2;
}
------------------------------------------------------------------------------------------------------------------
//HEAP MAH



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

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
  int *harr; // pointer to array of elements in heap
  int capacity; // maximum possible size of min heap
  int heap_size; // Current number of elements in min heap
  public:
  // Constructor
  MinHeap(int capacity);

  // to heapify a subtree with the root at given index
  void MinHeapify(int );

  int parent(int i) { return (i-1)/2; }

  // to get index of left child of node at index i
  int left(int i) { return (2*i + 1); }

  // to get index of right child of node at index i
  int right(int i) { return (2*i + 2); }

  // to extract the root which is the minimum element
  int extractMin();

  // Decreases key value of key at index i to new_val
  void decreaseKey(int i, int new_val);

  // Returns the minimum key (key at root) from min heap
  int getMin() { return harr[0]; }

  // Deletes a key stored at index i
  void deleteKey(int i);

  // Inserts a new key 'k'
  void insertKey(int k);
};

// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int cap)
{
  heap_size = 0;
  capacity = cap;
  harr = new int[cap];
```

```
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
  if (heap_size == capacity)
  {
    cout << "\nOverflow: Could not insertKey\n";
    return;
  }

  // First insert the new key at the end
  heap_size++;
  int i = heap_size - 1;
  harr[i] = k;

  // Fix the min heap property if it is violated
  while (i != 0 && harr[parent(i)] > harr[i])
  {
  swap(&harr[i], &harr[parent(i)]);
  i = parent(i);
  }
}

// Decreases value of key at index 'i' to new_val. It is assumed that
// new_val is smaller than harr[i].
void MinHeap::decreaseKey(int i, int new_val)
{
  harr[i] = new_val;
  while (i != 0 && harr[parent(i)] > harr[i])
  {
  swap(&harr[i], &harr[parent(i)]);
  i = parent(i);
  }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
  if (heap_size <= 0)
    return INT_MAX;
  if (heap_size == 1)
  {
    heap_size--;
    return harr[0];
  }

  // Store the minimum value, and remove it from heap
  int root = harr[0];
  harr[0] = harr[heap_size-1];
  heap_size--;
  MinHeapify(0);

  return root;
}


// This function deletes key at index i. It first reduced value to minus
// infinite, then calls extractMin()
void MinHeap::deleteKey(int i)
{
  decreaseKey(i, INT_MIN);
  extractMin();
}

// A recursive method to heapify a subtree with the root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
  int l = left(i);
  int r = right(i);
  int smallest = i;
  if (l < heap_size && harr[l] < harr[i])
    smallest = l;
  if (r < heap_size && harr[r] < harr[smallest])
    smallest = r;
  if (smallest != i)
  {
    swap(&harr[i], &harr[smallest]);
    MinHeapify(smallest);
  }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
}

// Driver program to test above functions
int main()
{
  MinHeap h(11);
  h.insertKey(3);
  h.insertKey(2);
  h.deleteKey(1);
  h.insertKey(15);
  h.insertKey(5);
  h.insertKey(4);
  h.insertKey(45);
  cout << h.extractMin() << " ";
  cout << h.getMin() << " ";
  h.decreaseKey(2, 1);
  cout << h.getMin();
  cout << endl;
  return 0;
}
------------------------------------------------------------------------------------------------------------------
//HEAP SIR KHURRAM



#include<iostream>
```

```
#include<assert.h>
using namespace std;
class MaxHeap{
    struct Node{
        int key;
        int value;
    };
    private:
    Node*arr;
    int capacity;
    int totalItems;
    void doubleCapacity(){
        if(this->arr==NULL){
            this->arr=new Node[1];
            this->capacity=1;
            return;
        }
        int newCapacity=capacity*2;
        Node*newArr=new Node[newCapacity];
        for(int i=0;i<this->totalItems;i++){
            newArr[i]=this->arr[i];
        }
        if(this->arr!=NULL)
        delete this->arr;
        this->capacity=newCapacity;
        this->arr=newArr;
    }
    void shiftUp(int index){
        if(index<1)
        return;
        int parent=(index-1)/2;
        if(this->arr[index].key>this->arr[parent].key){
            swap(this->arr[index],this->arr[parent]);
            shiftUp(parent);
        }
        return;
    }
    void shiftDown(int index){
        int maxIndex=-1;
        int lChildIndex=index*2+1;
        int rChildIndex=(index*2)+2;
        if(lChildIndex<totalItems){
            if(arr[index].key<arr[lChildIndex].key){
                maxIndex=lChildIndex;
            }
        }
        if(rChildIndex<totalItems){
            int newindex=(maxIndex==-1?index:maxIndex);
            if(arr[newindex].key<arr[rChildIndex].key){
                maxIndex=rChildIndex;
            }
        }
        if(maxIndex==-1)
        return;
        swap(arr[index],arr[maxIndex]);
        shiftDown(maxIndex);
    }
public:
    MaxHeap(){
        this->arr=NULL;
        this->capacity=0;
        this->totalItems=0;
    }
    MaxHeap(int _capacity){
        assert(_capacity>=1);
        this->arr=new Node[_capacity];
        this->capacity=_capacity;
        this->totalItems=0;
    }
    void insert(int key,int value){
        if(this->totalItems==this->capacity){
            doubleCapacity();
        }
        this->arr[totalItems].key=key;
        this->arr[totalItems].value=value;
        shiftUp(totalItems);
        this->totalItems++;
    }
    void getMax(int & value){
        assert(totalItems!=0);
        value=this->arr[0].value;
    }
    void deleteMax(){
        assert(totalItems!=0);
        swap(arr[0],arr[this->totalItems-1]);
        totalItems--;
        //shift down
        shiftDown(0);
    }
    bool isEmpty() const
    {
        return (totalItems==0);
    }
    void deleteAll(){
        if(this->arr!=NULL){
            delete[]arr;
            arr=NULL;
            this->capacity=0;
            this->totalItems=0;
        }
    }
    ~MaxHeap(){
        deleteAll();
    }
};
int main(){
    MaxHeap a;
    for(int i=1;i<=200;i++)
        a.insert(i,i);
    a.deleteAll();
    for(int i=201;i<=300;i++)
    a.insert(i,i);
    while(!a.isEmpty()){
        int s;
        a.getMax(s);
        cout<<s<<endl;
        a.deleteMax();
    }
}
------------------------------------------------------------------------------------------------------------------
//INFIX TO POSTFIX USING STACK



#include<iostream>

#include<stack>

using namespace std;

bool IsOperator(char);

bool IsOperand(char);

bool eqlOrhigher(char, char);

string convert(string);

int main()

{

string infix_expression, postfix_expression;

int ch;

do

{

cout << "Enter your expression ";

cin >> infix_expression;

postfix_expression = convert(infix_expression);

cout << "The Infix expression is.... "<<endl << infix_expression;

cout<<endl;

cout<<endl;

cout << "The Postfix expression is....."<<endl << postfix_expression;

cout<<endl;

cout<<endl;

cout << "Press 1 to enter new expression and 0 to stop the working ";

cout<<endl;

cin >> ch;
```

```

} while(ch == 1);

return 0;

}

bool IsOperator(char c)

{

if(c == '+' || c == '-' || c == '*' || c == '/' || c == '^' )

return true;

return false;

}

bool IsOperand(char c)
{

if( c >= 'A' && c <= 'Z')

return true;

if (c >= 'a' && c <= 'z')

return true;

if(c >= '0' && c <= '9')

return true;

return false;
}

int precedence(char op)

{

if(op == '+' || op == '-')

return 1;

if (op == '*' || op == '/')

return 2;

if(op == '^')

return 3;

return 0;

}

bool eqlOrhigher (char op1, char op2)
```

```

{

int p1 = precedence(op1);

int p2 = precedence(op2);

if (p1 == p2)

{

if (op1 == '^' )

return false;

return true;

}

return  (p1>p2 ? true : false);

}

string convert(string infix)

{

stack <char> S;

string postfix ="";

char ch;

S.push( '(' );

infix += ')';

for(int i = 0; i<infix.length(); i++)

{

ch = infix[i];

if(ch == ' ')

continue;

else if(ch == '(')

S.push(ch);

else if(IsOperand(ch))

postfix += ch;

else if(IsOperator(ch))

{
```

```

while(!S.empty() && eqlOrhigher(S.top(), ch))

{

postfix += S.top();

S.pop();

}

S.push(ch);

}

else if(ch == ')')

{

while(!S.empty() && S.top() != '(')

{

postfix += S.top();

S.pop();

}

S.pop();

}

}

return postfix;

}
-----------------------------------------------------------------------------------------------------------------
//AVL IMPLEMENTATION



#include<iostream>
using namespace std;
class node{
  public:
    node *left;
    node*right;
    int data;
    int height;
    node(int data)
    {
      this->data=data;
      height=0;
      left=right=NULL;
    }
```

```
};
class AVLtree{
  private:
  node*root;
  void makeEmpty(node* t);
  node* insert(int x,node*t);
  node* singleleftrotate(node* &C);
  node* singlerightrotate(node*&C);

  node* doubleleftrightrotate(node* &C);
  node* doublerightleftrotate(node* &C);

  node*findmin(node*t);
  node*findmax(node *t);

  node *remove(int x,node*t);
  int height(node*t);
  int getBalance(node*t);
  void inorder(node *t);

  public:
    AVLtree()
    {
      root=NULL;
    }
    void insert(int x){
      root=insert(x,root);
    }
    void remove(int x)
    {
      root=remove(x,root);
    }
    void display()
    {
      inorder(root);
      cout<<endl;
    }


};

int main()
{
  AVLtree tree;
  tree.insert(3);
  tree.insert(4);
  tree.insert(5);
  tree.insert(6);
  tree.insert(7);
  tree.display();
  return 0;
}

node* AVLtree::singleleftrotate(node* &A)
{
node* newRoot = A->right;
A->right = newRoot->left;
newRoot->left = A;
```

```
A->height = max(height(A ->left), height(A ->right)) + 1;
newRoot ->height = max(height(newRoot->right), A->height) + 1;
return newRoot;
}

node* AVLtree::singlerightrotate(node* &C)
{
node* newRoot = C->left;
C->left = newRoot->right;
newRoot->right = C;
C->height = max(height(C ->left), height(C ->right)) + 1;
newRoot ->height = max(height(newRoot->left), C->height) + 1;
return newRoot;
}

node* AVLtree::doubleleftrightrotate(node*& t)
{
t->left = singleleftrotate(t->left);
return singlerightrotate(t);
}

node* AVLtree::doublerightleftrotate(node*& t)
{
t->right = singlerightrotate(t->right);
return singleleftrotate(t);
}

void AVLtree::inorder(node *t)
{
  if(t==NULL)
  return;
  inorder(t->left);
  cout<<t->data<<" ->";
  inorder(t->right);
}
int AVLtree::height(node* t)
{
  return(t==NULL ? -1 : t->height);
}
int AVLtree::getBalance(node*t)
{
  if(t==NULL)
  return 0;
  else
  return height(t->left) - height(t->right);
}


node *AVLtree::findmin(node *t)
{
  if(t==NULL)
  return NULL;
  else if(t->left==NULL)
    return  t;
  else
    return findmin(t->left);
}

```

```
node *AVLtree::findmax(node *t)
{
  if(t==NULL)
      return NULL;
  else if(t->right==NULL)
    return  t;
  else
    return findmax(t->right);
}
void AVLtree::makeEmpty(node* t) {
        if(t == NULL)
            return;
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
    }

node*  AVLtree::  insert(int x, node* t)
    {
        if(t == NULL)
        {
            t = new node (x);
        }
        else if(x < t->data)
        {
            t->left = insert(x, t->left);
            if(height(t->left) - height(t->right) == 2)
            {
                if(x < t->left->data)
                    t = singlerightrotate(t);
                else
                    t = doubleleftrightrotate(t);
            }
        }
        else if(x > t->data)
        {
            t->right = insert(x, t->right);
            if(height(t->right) - height(t->left) == 2)
            {
                if(x > t->right->data)
                    t = singleleftrotate(t);
                else
                    t = doublerightleftrotate(t);
            }
        }

        t->height = max(height(t->left), height(t->right))+1;
        return t;
    }

    node* AVLtree::remove(int x, node* t)
    {
        node* temp;

        // Element not found
        if(t == NULL)
            return NULL;

        // Searching for element
        else if(x < t->data)
            t->left = remove(x, t->left);
        else if(x > t->data)
            t->right = remove(x, t->right);

        // Element found
        // With 2 children
        else if(t->left && t->right)
        {
            temp = findmin(t->right);
            t->data = temp->data;
            t->right = remove(t->data, t->right);
        }
        // With one or zero child
        else
        {
            temp = t;
            if(t->left == NULL)
                t = t->right;
            else if(t->right == NULL)
                t = t->left;
            delete temp;
        }
        if(t == NULL)
            return t;

        t->height = max(height(t->left), height(t->right))+1;

        // If node is unbalanced
        // If left node is deleted, right case
        if(height(t->left) - height(t->right) == 2)
        {
            // right right case
            if(height(t->left->left) - height(t->left->right) == 1)
                return singleleftrotate(t);
            // right left case
            else
                return doublerightleftrotate(t);
        }
        // If right node is deleted, left case
        else if(height(t->right) - height(t->left) == 2)
        {
            // left left case
            if(height(t->right->right) - height(t->right->left) == 1)
                return singlerightrotate(t);
            // left right case
            else
                return doubleleftrightrotate(t);
        }
        return t;
    }
```