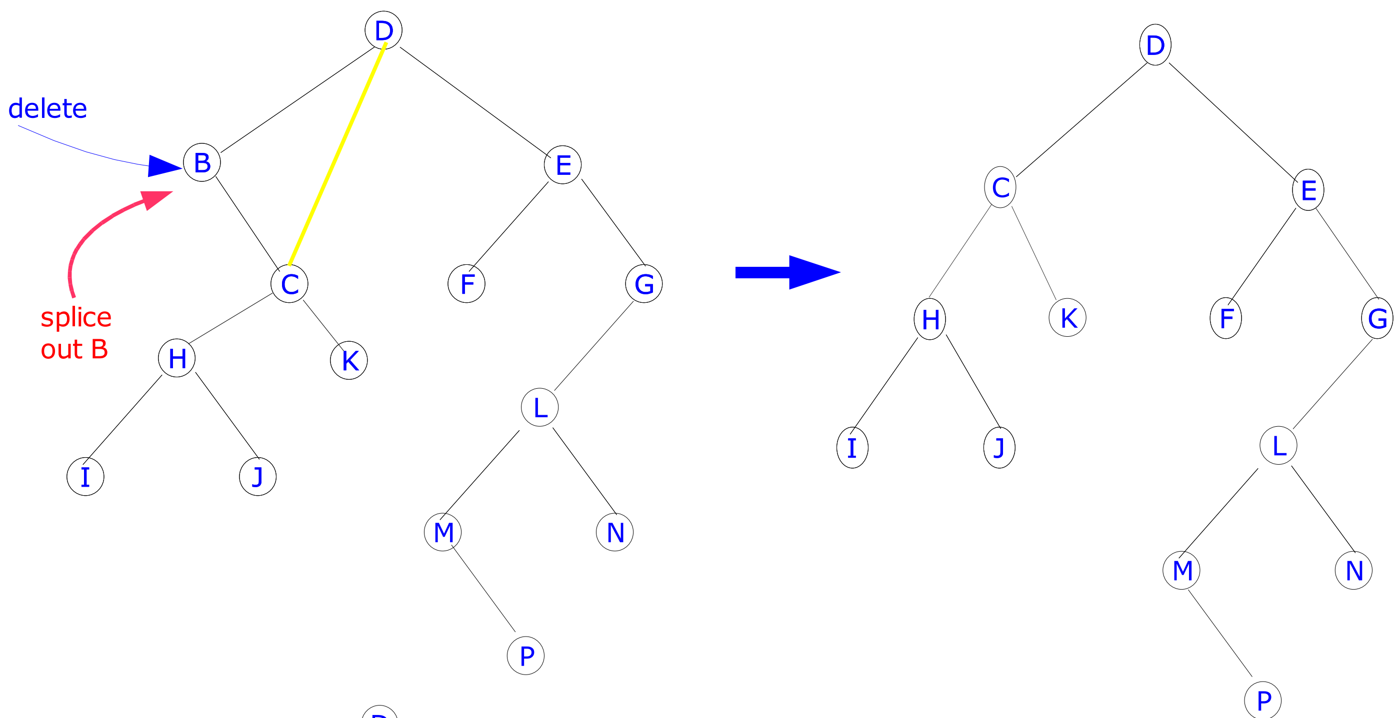
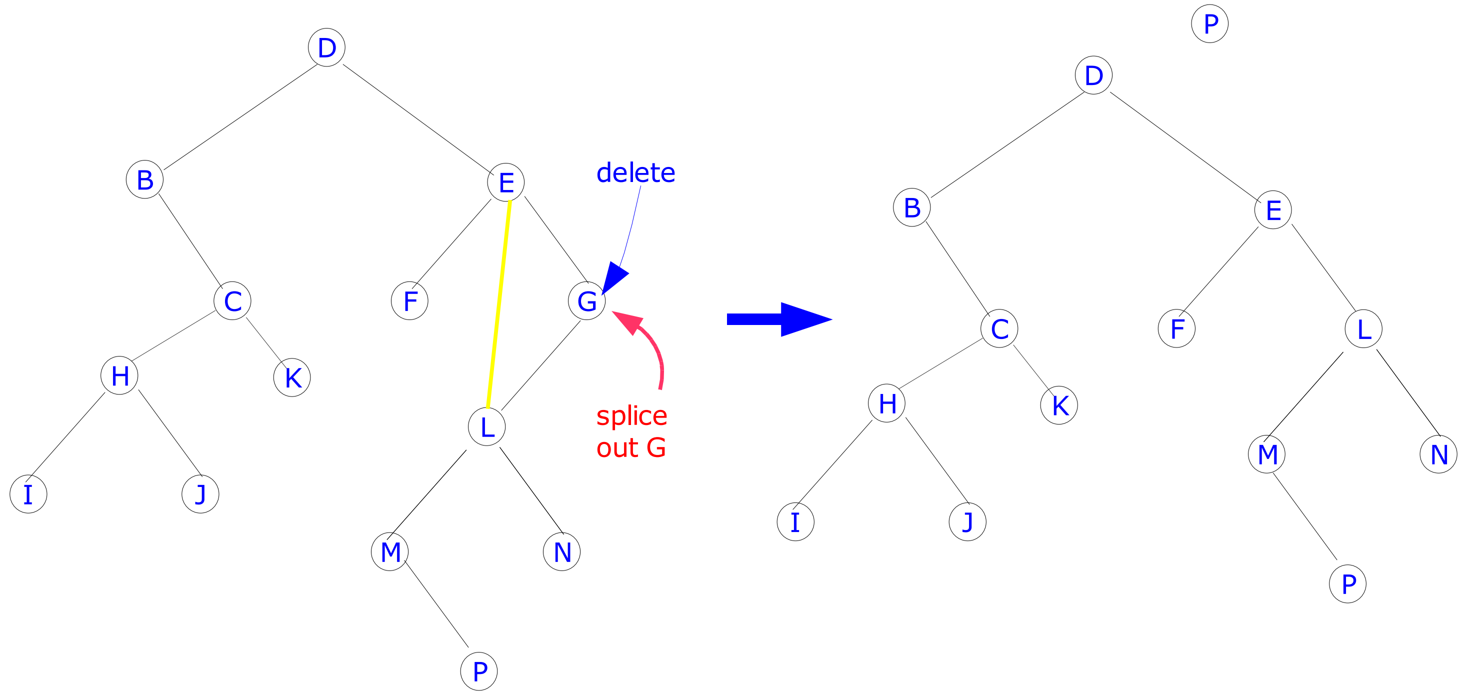
Binary Search Tree Deletion operation

1. To delete a leaf node, just delete it.

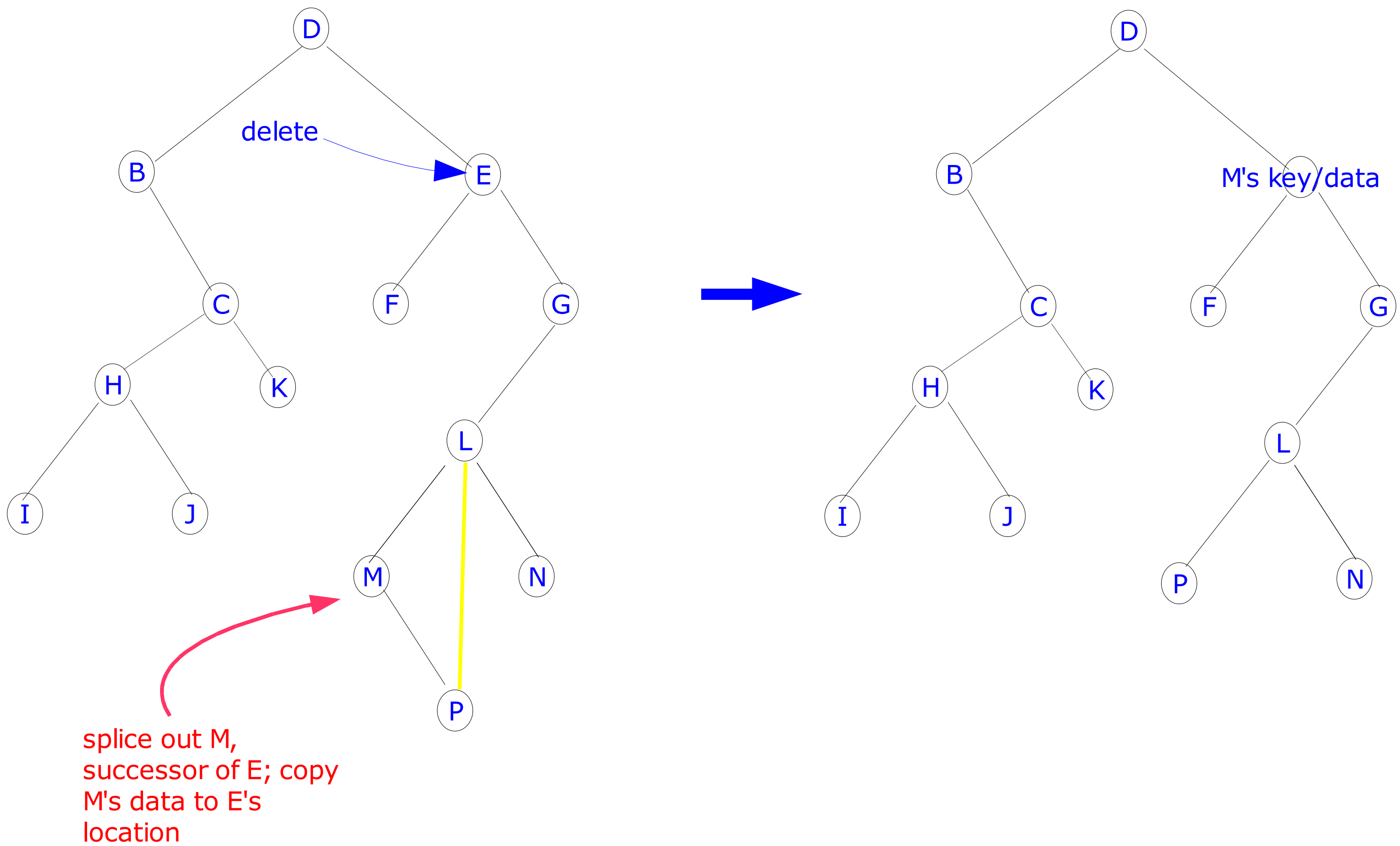
2. If the node to be deleted has only one child, splice that node out by connecting its parent and child as shown below:

(Note: in the figures below, the letters in the nodes are just labels, not KEY values)

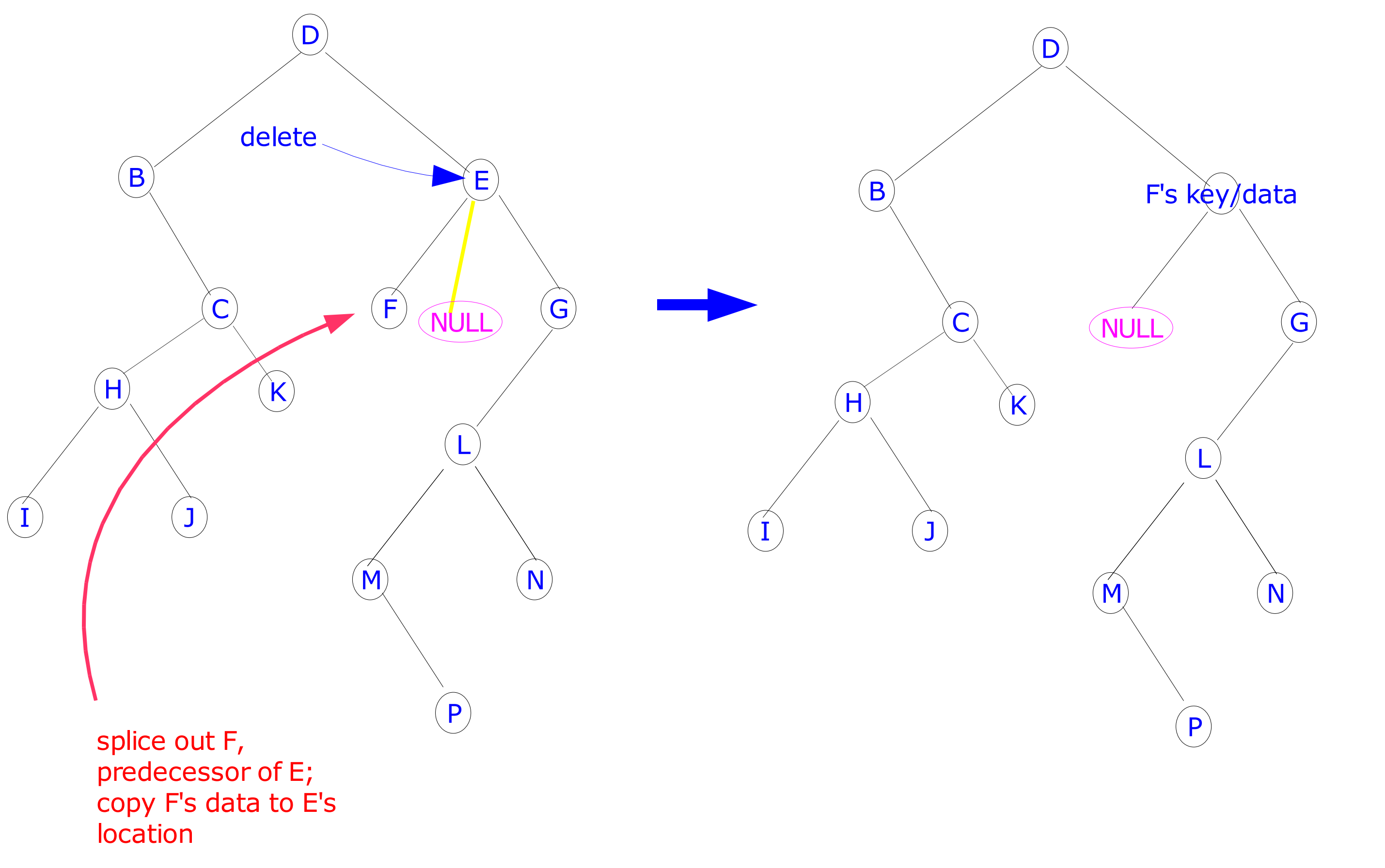




3. If the node to be deleted has two children, splice out its successor, and replace the key/data in the node to be deleted by the key/data from the spliced out successor:



Alternatively, you could do the same with the predecessor instead of the successor:



```
/*********Deletion Algorithm - From Carmen/Leiserson/Rivest/Stein************
delete(T, z):
//First determine node y to splice out
if left[z]==NULL or right[z]==NULL
   y=z   //node to splice out if z has AT MOST one child
else
   y=successor(z)    //node to splice out if z has two children

//set the non-NULL child of y to x, or set x to NULL if y has no children
if left[y] != NULL
   x=left[y]
else
   x=right[y]
//Now do the splicing
if x != NULL
   parent[x]=parent[y]
if parent[y]==NULL
   x=root[T]
else
   if y==left[parent[y]]
     x=left[parent[y]]
   else
     x=right[parent[y]]
//Splicing done
// if successor of z was the spliced out node, move y's key/data to z
if y != z
   key[y]=key[z]]
//copy non-key data too if present.
//return y if needed
*****************************************************/
```

Here is a complete Binary Search Tree program that implements the deletion operation.

```
/*BTree.cpp */
#include<iostream>
using namespace std;

class BTree
{
public:
  int key;  //data in the node
  BTree *left;   // Pointer to the left subtree.
  BTree *right;  // Pointer to the right subtree.
  BTree *parent; // Pointer to the parent node
  BTree();
  ~BTree();
  void insert(int key);   //insert a new node at a leaf position with the given int data
  void insert(BTree* leaf);  //insert a  given leaf node into the tree
  BTree *search(int key); //return NULL if no node has given int value, or a pointer
to the node that has the int value
  void destroy();         //clean up the whole tree
  const void preOrderPrint();
  const void inOrderPrint();
```

```
  const void postOrderPrint();
  BTree* find_Max();
  BTree* find_Min();
  int countNodes();
  int countLeafNodes();
  int find_depth();
  BTree* successor(int); //Returns pointer to successor node of this node
  BTree* predecessor(int) ; //Returns pointer to predecessor node of this node
  void remove(int val); //delete the node pointed to by the pointer Node.
  BTree* find_root(); //returns a pointer to the root node
};


BTree::BTree(){
  left=NULL;
  right=NULL;
  parent=NULL;
}

void BTree::insert(int key){
  BTree* bt=new BTree;  //new node
  bt->key=key;      //assign key to new node
  insert(bt);
}



void BTree::insert( BTree* leaf){
  if(!(this->key)){
    key=leaf->key;
  }
  else if ( (this->key) >= (leaf->key) ){
    if((this->left != NULL)){
      (this->left)->insert(leaf);
    }else{
      this->left=leaf;
      leaf->parent=this;
    }
  }
  else {
    if(this->right != NULL){
      (this->right)->insert(leaf);
    }else{
      this->right=leaf;
      leaf->parent=this;
    }
  }
}





const void BTree::inOrderPrint(){
  if( (left==NULL) && (right==NULL) ){
```

```
    if(key){cout<<key<<" ";}
  }
  else {
    if(left)left->inOrderPrint();
    cout<<key<<" ";
    if(right)right->inOrderPrint();
  }
}


const void BTree::preOrderPrint(){
  if( (left==NULL) && (right==NULL) ){
    if(key){cout<<key<<" ";}
  }
  else{
    cout<<key<<" ";
    if(left)left->preOrderPrint();
    if(right)right->preOrderPrint();
  }
}

const void BTree::postOrderPrint(){
  if( (left==NULL) && (right==NULL) ){
    if(key){cout<<key<<" ";}
  }
  else{
    if(left)left->postOrderPrint();
    if(right)right->postOrderPrint();
    cout<<key<<" ";
  }
}


BTree* BTree::search(int keyval){
  if(this->key==keyval){
    return this;
  }
  else if( (this->key) > keyval){ //search in left subtree
    if(this->left)
      return this->left->search(keyval);
    else return NULL;
  }else{
    if(this->right)
      return this->right->search(keyval);
    else return NULL;
  }
}



BTree* BTree::find_Max(){
  if(right == NULL){
    return this;
  }else{
    right->find_Max();
```

```
  }
}

BTree* BTree::find_Min(){
  if(left == NULL){
    return this;
  }else{
    left->find_Min();
  }
}

int BTree::countNodes(){
  int l, r;
  if((left==NULL)&&(right==NULL)){
    if(key)return 1; else return 0;
  }
  else{
    if(left)
      l=left->countNodes(); else l=0;
    if(right)
      r=right->countNodes(); else r=0;
    return 1+l+r;
  }
}


BTree* BTree::find_root(){ //return root of the tree
  if (this->parent == NULL)
    return this;
  else return (this->parent)->find_root();
}



int BTree::countLeafNodes(){
  int c=0,l=0,r=0;
  if((left==NULL)&&(right==NULL)){
    if(key) c++;
  }else{
    if(left)
      l=left->countLeafNodes();
    if(right)
      r=right->countLeafNodes();
  }
  return c+l+r;
}

int BTree::find_depth(){
  if(this==NULL)
    return 0;
  else{
    int leftDepth=left->find_depth();
    int rightDepth=right->find_depth();
    if (leftDepth>=rightDepth)
      return 1+leftDepth;
```

```
    else
      return 1+rightDepth;
  }
}




/***********Algorithm-Cormen/Leiserson/Rivest/Stein*********

TREE-SUCCESSOR(x)
if(right[x] != NULL)
   return find_min(right[x]);
y=parent[x];
while ( y != NULL and x == right[y]){
    x=y;
    y=parent[x];
}
return y
*********************************************/

BTree* BTree::successor(int val){
  BTree* np=this->search(val);
  if(np){               //should be a node in the tree
    if(np->right){      //node has a right subtree
      return (np->right)->find_Min();
    }
    else{ //go up the tree till you find a node which is a left child
      BTree* y= np->parent;
      BTree* x= np;
      while( (y != NULL) && (y->right==x)){
          x=y;
          y=x->parent;
      }
      if(y==NULL)
          return np; //np is Maximum element in the tree
      else
          return y;
    }
  }
  else return NULL; //node with given int value is not in the tree
}

BTree* BTree::predecessor(int val){
  BTree* np=this->search(val);
  if(np){               //should be a node in the tree
    if(np->left){      //node has a right subtree
      return (np->left)->find_Max();
    }
    else{ //go up the tree till you find a node which is a right child
      BTree* y= np->parent;
      BTree* x= np;
      while( (y != NULL) && (y->left==x)){
          x=y;
          y=x->parent;
```

```
    }
    if(y==NULL)
      return np; //np is Minimum element in the tree
    else
      return y;
   }
  }
  else return NULL; //node with given int value is not in the tree
}

/***Deletion Algorithm***************
****Introduction to Algorithms -Second Ed. Carmen/Leiserson/Rivest/Stein******
        delete(T, z):
        //First determine node y to splice out
        if left[z]==NULL or right[z]==NULL
        y=z   //node to splice out if z has only one child
        else
        y=successor(z)    //node to splice out if z has two children

        //set the non-NULL child of y to x, or set x to NULL if y has no children
        if left[y] != NULL
        x=left[y]
        else
        x=right[y]
        //Now do the splicing
        if x != NULL
        parent[x]=parent[y]
        if parent[y]==NULL
        x=root[T]
        else
        if y==left[parent[y]]
        left[parent[y]]=x
        else
        right[parent[y]]=x
        //Splicing done
        // if successor of z was the spliced out node, move y's key/data to z
        if y != z
        key[y]=key[z]
        //copy non-key data too if present.
        //return y if needed
*****************************************************/


void BTree::remove(int val){  //delete node containing val as its key
  BTree* z=this->search(val); //get hold of a pointer to the node having val as its
key
  BTree* y,*x;
  if(z){              //should be a node in the tree
   //y is the node to splice out, it is either the node with key=val, or its successor
   if( (z->left == NULL)||(z->right ==NULL)) //z has at most  one child
    y=z;
   else
    y=this->successor(val);  //z has two childern
   //set x to be either the only child of y or NULL. Note that successor can't have two
children
```

```
    if( y->left != NULL)
      x=y->left;
    else
      x=y->right;
    //###############Now do the splicing###########
    if (x != NULL){
      x->parent=y->parent;
    }
    if(y->parent == NULL){
      x=this->find_root();
    }
    else if (y == (y->parent)->left){ //y is a left child of its parent
      (y->parent)->left=x;
    }
    else {  //y is a right child of its parent
      (y->parent)->right=x;
    }
    //##################splicing done############
    // if successor of z was the spliced out node, move y's key/data to z
    if (y != z)
      z->key=y->key;
    //copy non-key data too if present.
    //return y if needed
  }

  else {
    cout<<"Not Found\n";
  }
}


int main(){
  BTree* bt=new BTree;
  bt->insert(12);
  bt->insert(5);
  bt->insert(10);
  bt->insert(21);
  bt->insert(13);
  bt->insert(3);
  bt->insert(15);
  bt->insert(22);
  bt->insert(7);
  cout<<"in-order->\t";bt->inOrderPrint();cout<<endl;
  cout<<"pre-order->\t"; bt->preOrderPrint(); cout<<endl;
  cout<<"post-order->\t"; bt->postOrderPrint(); cout<<endl;
  cout<<endl;
  cout<<"Max Value->"<<bt->find_Max()->key<<endl;
  cout<<"Min Value->"<<bt->find_Min()->key<<endl;
  cout<<"# of Nodes->"<<bt->countNodes()<<endl;
  cout<<"# of Leaf Nodes->"<<bt->countLeafNodes()<<endl;
  cout<<"Depth of the tree->"<<bt->find_depth()<<endl;

  cout <<"Successor of  10->"<< bt->successor(10)->key<<endl;
  int m;
```

```
 cout <<"Root is "<<(bt->find_root())->key<<endl;
 cout<<"Enter a value to remove\n";
 cin>>m;
 bt->remove(m);
 cout<<"Deleted "<<m<<"\n";
 cout<<"in-order->\t";bt->inOrderPrint();cout<<endl;
 cout<<"pre-order->\t"; bt->preOrderPrint(); cout<<endl;

 /*
   while (1){
   cout<<"Successor/predecessor of which element do you need?\n";
   cin>>m;

   if(bt->search(m)){
   cout <<"Successor of "<<m<<"->"<< bt->successor(m)->key<<endl;
   cout <<"Predecesssor of "<<m<<"->"<< bt->predecessor(m)->key<<endl;
   }
   else
   cout<<"Node not Found\n";
   }

 */


 /*
   while (1){
   cout<<"enter a value to serach\n";
   cin>>m;
   if(bt->search(m)){
   cout<<"Found\n";
   }else{
   cout<<"Not Found\n";
   }
   }
 */
}
```