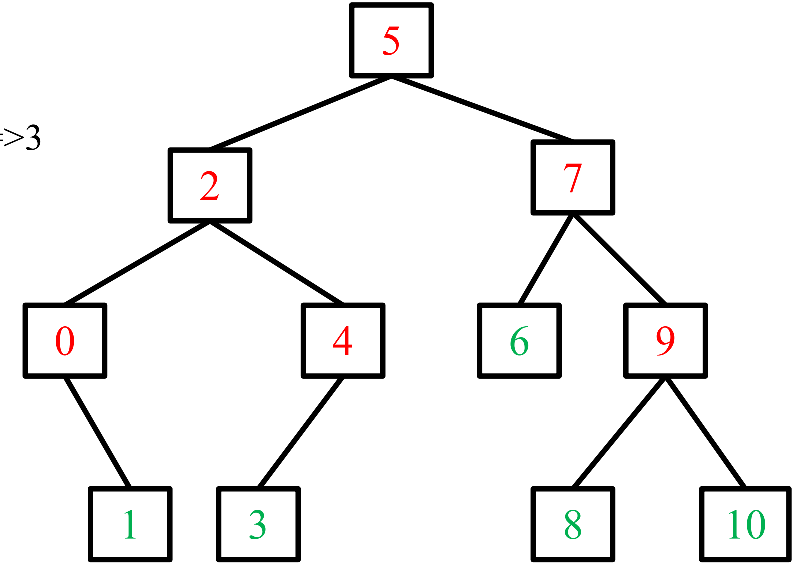
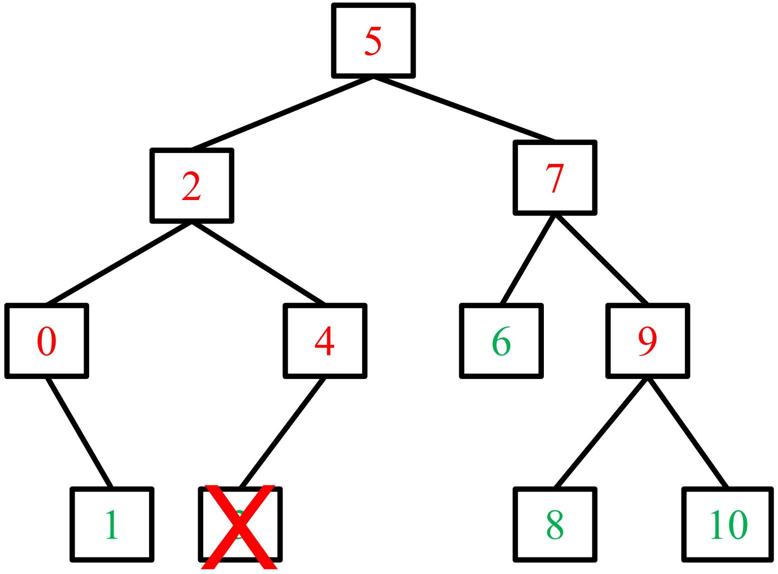
Binary Search Tree

- **Binary search tree property**
  - left subtree with key values < self key value
  - right subtree with key values $\geq$ self key value
- **Binary search**
  - search down only one subtree
    - e.g. find 3 : 5(3<5)=>2(3>2)=>4(3<4)=>3
- **Insert new nodes**
  - binary search & insert
    - 5>2>4>3>7>6>0>1>9>10>8
- **Remove existing nodes**



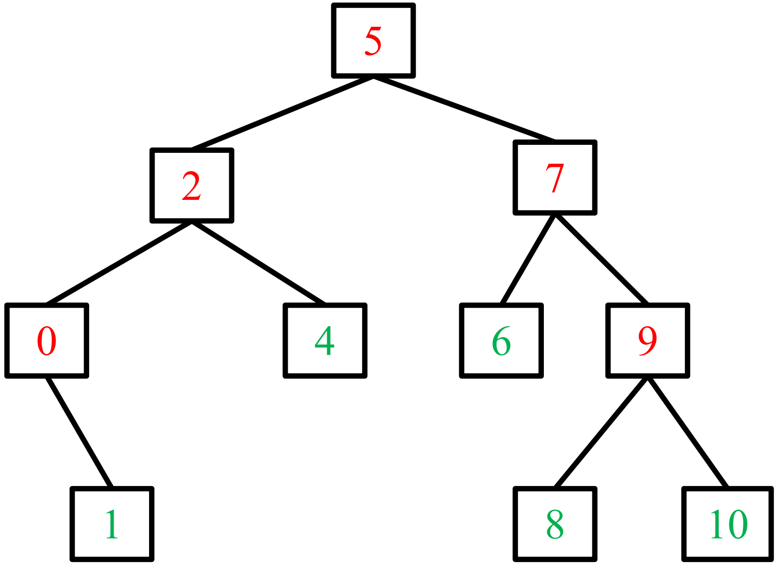
Binary Search Tree

- **Binary search tree property**
  - left subtree with key values < self key value
  - right subtree with key values ≥ self key value
- **Binary search**
  - search down only one subtree
- **Insert new nodes**
  - binary search & insert
- **Remove existing nodes**
  - Remove a leaf



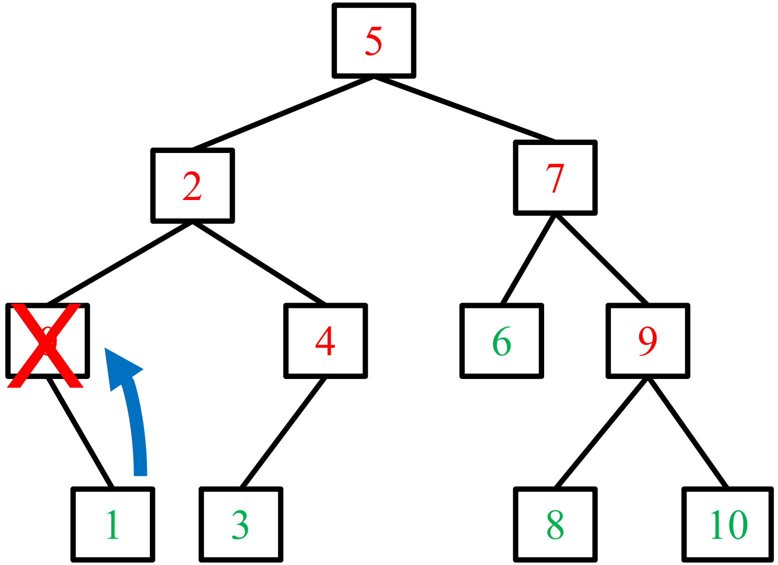
Binary Search Tree

- **Binary search tree property**
  - left subtree with key values < self key value
  - right subtree with key values $\geq$ self key value
- **Binary search**
  - search down only one subtree
- **Insert new nodes**
  - binary search & insert
- **Remove existing nodes**
  - Remove a leaf
    - trivial operation



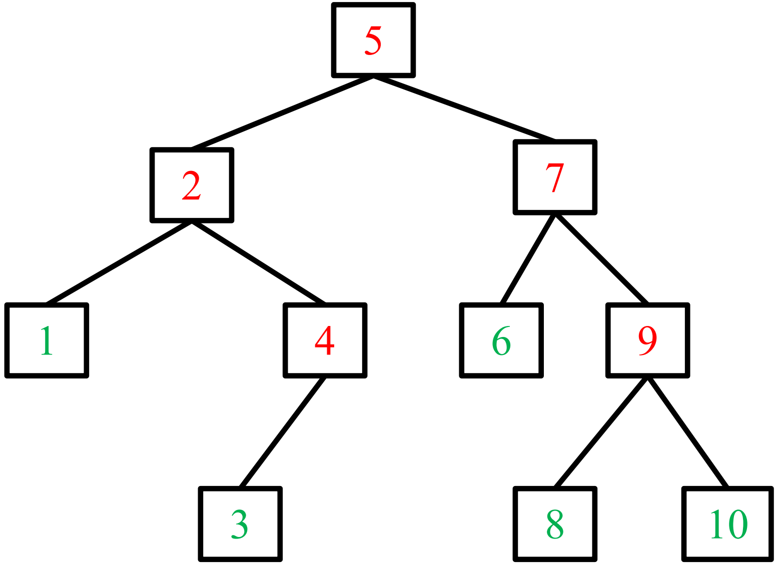
Binary Search Tree

- **Binary search tree property**
  - left subtree with key values < self key value
  - right subtree with key values $\geq$ self key value
- **Binary search**
  - search down only one subtree
- **Insert new nodes**
  - binary search & insert
- **Remove existing nodes**
  - Remove a node with only one child



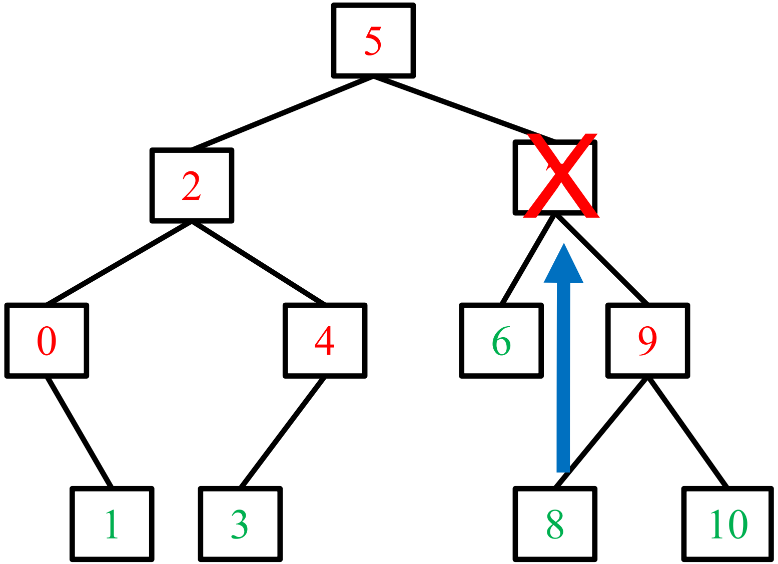
Binary Search Tree

- **Binary search tree property**
  - left subtree with key values < self key value
  - right subtree with key values $\geq$ self key value
- **Binary search**
  - search down only one subtree
- **Insert new nodes**
  - binary search & insert
- **Remove existing nodes**
  - Remove a node with only one child
    - Link its parent to its only child



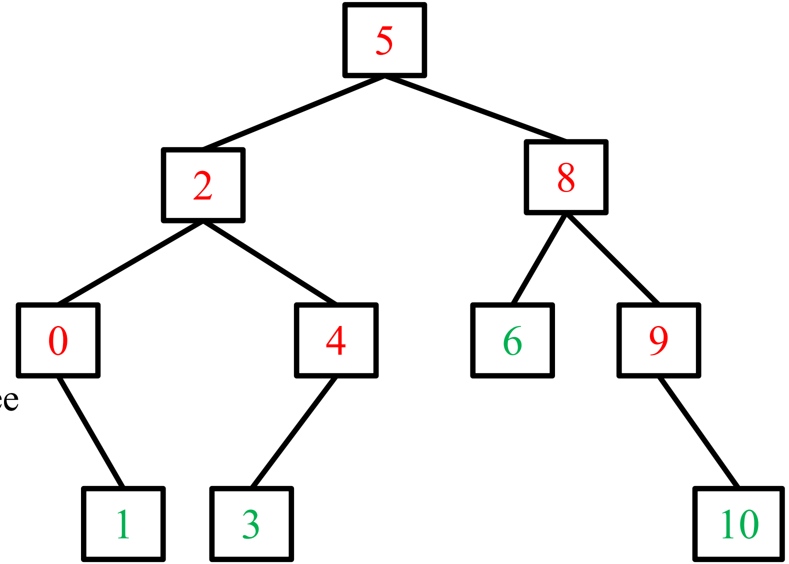
Binary Search Tree

- **Binary search tree property**
  - left subtree with key values < self key value
  - right subtree with key values ≥ self key value
- **Binary search**
  - search down only one subtree
- **Insert new nodes**
  - binary search & insert
- **Remove existing nodes**
  - Remove a node with both children



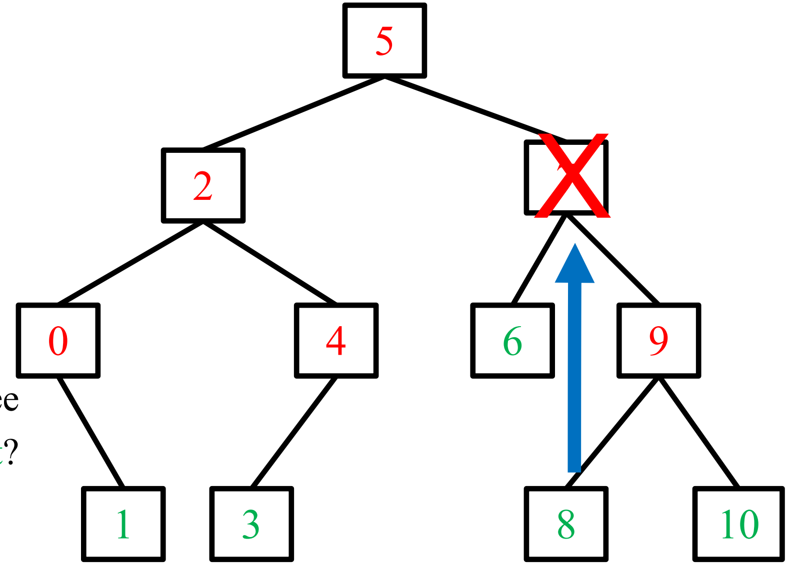
Binary Search Tree

- **Binary search tree property**
  - left subtree with key values $<$ self key value
  - right subtree with key values $\geq$ self key value
- **Binary search**
  - search down only one subtree
- **Insert new nodes**
  - binary search & insert
- **Remove existing nodes**
  - Remove a node with both children
    - replace it by the ‘min’ of its right subtree



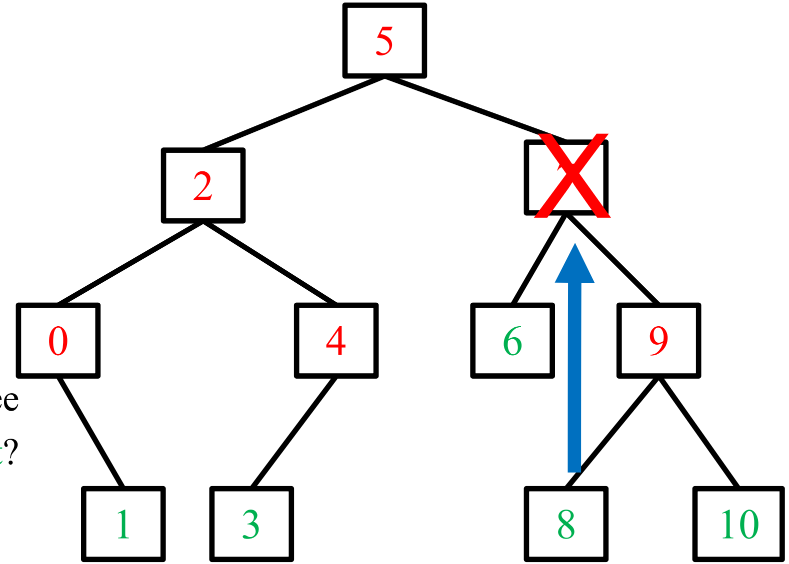
Binary Search Tree

- **Binary search tree property**
  - left subtree with key values < self key value
  - right subtree with key values $\geq$ self key value
- **Binary search**
  - search down only one subtree
- **Insert new nodes**
  - binary search & insert
- **Remove existing nodes**
  - Remove a node with both children
    - replace it by the 'min' of its right subtree
    - Why 'min' of right but not 'max' of left?



Binary Search Tree

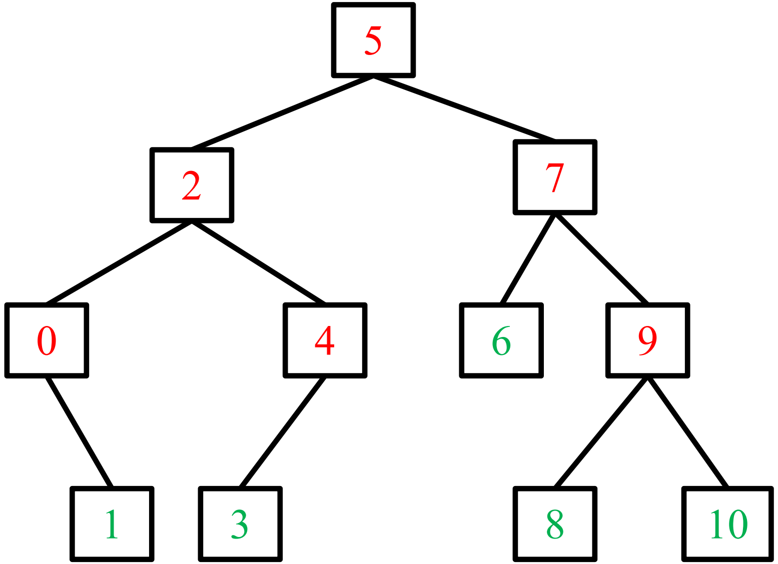
- **Binary search tree property**
  - left subtree with key values < self key value
  - right subtree with key values $\boxed{\geq}$ self key value
- **Binary search**
  - search down only one subtree
- **Insert new nodes**
  - binary search & insert
- **Remove existing nodes**
  - Remove a node with both children
    - replace it by the 'min' of its right subtree
    - Why 'min' of right but not 'max' of left?



Binary Search Tree

- **Binary search tree node**

structure abstraction



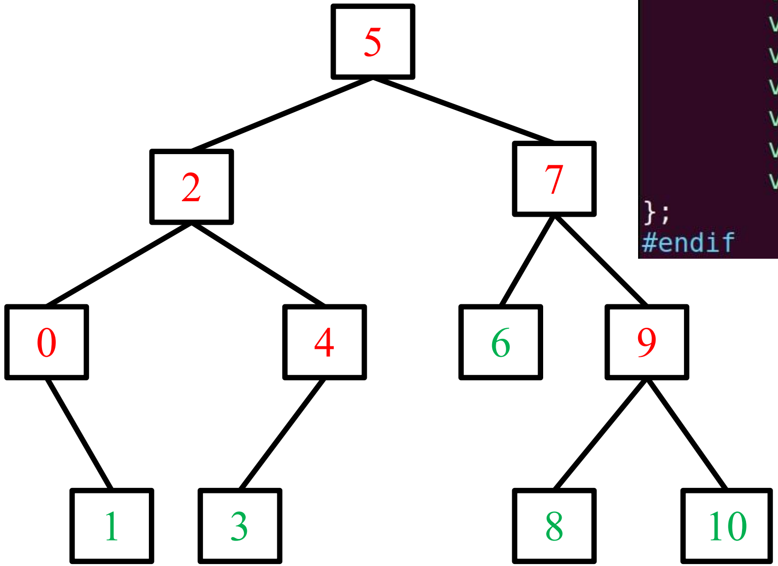

```
#ifndef __BSTNODE_H__
#define __BSTNODE_H__
#include <iostream>
#include "BNode.h"
template <typename Y, typename T> // Y {key} : T {element}
class BSTNode: public BNode<T>{
private:Y k; T e; // node's key & value
        BSTNode* cL; BSTNode* cR; // node's left child & right child
public: BSTNode(){cL=cR=NULL;} // constructor without initial values
        BSTNode(const Y& ki,const T& ei,BSTNode* L=NULL,BSTNode* R=NULL){
                k=ki;e=ei;cL=L;cR=R;} // constructor with initial values
        ~BSTNode(){}
        T& getE(){return e;}
        void setE(const T& ei){e=ei;}
        Y& getK(){return k;}
        void setK(const Y& ki){k=ki;}
        inline BSTNode* getL() const{return cL;}
        void setL(BNode<T>* b){cL=(BSTNode*)b;}
        inline BSTNode* getR() const{return cR;}
        void setR(BNode<T>* b){cR=(BSTNode*)b;}
        bool isLeaf(){return (cL==NULL)&&(cR==NULL);}
};
// ostream overloading, so that 'cout<<{BSTNode<Y,T> object}' can have meaning
template <typename Y,typename T>
std::ostream& operator<<(std::ostream& out,BSTNode<Y,T>& b){
        out<<b.getK()<<':'<<b.getE(); return out;}
#endif
```

Binary Search Tree

structure abstraction

- **Binary search tree**

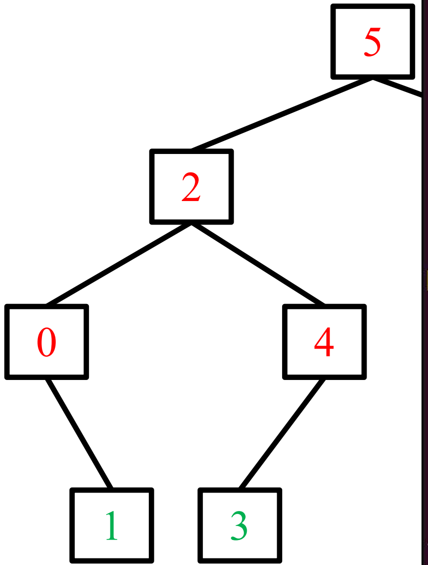
```
template <typename Y, typename T> class Dictionary{ // Dictionary abstract class
private:void operator=(const Dictionary&){}
        Dictionary(const Dictionary&){}
public: Dictionary(){}
        virtual ~Dictionary(){}
        virtual void clear()=0;
        virtual void insert(const Y& k, const T& e)=0; // k: key ; e: element
        virtual T remove(const Y& k)=0;
        virtual T remove()=0; // return an arbitrary record
        virtual T find(const Y& k) const=0;
        virtual int size()=0; // return the number of records in the dictionary
};
#endif
```



Binary Search Tree

structure abstraction

- **Binary search tree**

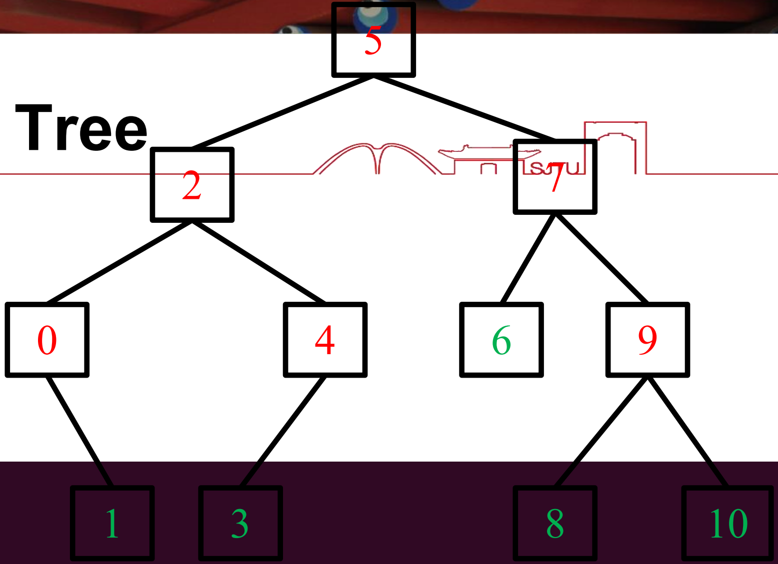



```
template <typename Y,typename T> // Y {key} : T {element}
class BST: public Dictionary<Y,T>{
private:BSTNode<Y,T>* r; int n; // root of BST; number of BST nodes
        // internal functions
        BSTNode<Y,T>* getm(BSTNode<Y,T>*); // get node with minimum key
        BSTNode<Y,T>* deletem(BSTNode<Y,T>*); // delete node with minimum key
        T findIn(BSTNode<Y,T>*,const Y&) const;
        BSTNode<Y,T>* insertIn(BSTNode<Y,T>*,const Y&,const T&);
        void clearIn(BSTNode<Y,T>*);
        BSTNode<Y,T>* removeIn(BSTNode<Y,T>*,const Y&);
        void indent(int) const;
        void printInorder(BSTNode<Y,T>*,int) const; // inorder printing by default
        void printPreorder(BSTNode<Y,T>*,int) const;
        void printPostorder(BSTNode<Y,T>*,int) const;
public: BST();
        ~BST();
        int size(); // return the number of BST nodes
        void clear();
        void insert(const Y& k,const T& e);
        T find(const Y& k) const;
        T remove(const Y& k); // remove a key-specified record
        T remove(); // remove an arbitrary record
        void print(int m) const; // m (print mode): -1 preorder, 1 postorder, otherwise inorder
};
```

Binary Search Tree

- Binary search tree

```
g++ demoBST.cpp -o _a; ./_a
```




```cpp
#include <iostream>
#include "BST.h"
using namespace std;

int main(){const int n0=11;int k,kBag[]={5,2,4,3,7,6,0,1,9,10,8};const char* ke;
        const char* eBag[]={"five","two","four","three","seven","six","zero","one","nine","ten","eight"};
        BST<int,const char*> aBST;for(int i=0;i<n0;i++){aBST.insert(kBag[i],eBag[i]);
                cout<<"Insert "<<kBag[i]<<" =>\n";aBST.print(0);cin.get();}
        cout<<"Preorder printing of BST:\n";aBST.print(-1);cin.get();
        cout<<"Postorder printing of BST:\n";aBST.print(1);cin.get();
        cout<<"Inorder printing of BST:\n";aBST.print(0);cin.get();
        ke=aBST.find(6);ke=(ke==NULL?"NOTHING":ke);cout<<"Search key 6 and have "<<ke<<endl;cin.get();
        ke=aBST.find(8);ke=(ke==NULL?"NOTHING":ke);cout<<"Search key 8 and have "<<ke<<endl;cin.get();
        ke=aBST.find(-1);ke=(ke==NULL?"NOTHING":ke);cout<<"Search key -1 and have "<<ke<<endl;cin.get();
        cout<<"Before removal =>\n";aBST.print(0);cin.get();cout<<"After removal of key 7 =>\n";
        ke=aBST.remove(7);aBST.print(0);cout<<ke<<" is removed\n";cin.get();
        cout<<"After default removal further =>\n";
        ke=aBST.remove();aBST.print(0);cout<<ke<<" is removed\n";cin.get();
        return 0;
}
```

THANK YOU

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY