```cpp
// Taylor Pedretti - 005488635
// bibarySearchTree.h
// afater Mark A. Weiss, Chapter 4

// KV replaced exceptions with assert statements;

#ifndef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H

//#include "dsexceptions.h"
#include <cassert>
#include <algorithm>
using namespace std;


template <typename C>
class BinarySearchTree
{

 private:
  struct BinaryNode
    {
      C element;
      BinaryNode *left;
      BinaryNode *right;
      BinaryNode *parent;

    BinaryNode( const C & theElement, BinaryNode *lt, BinaryNode *rt,
              BinaryNode* par)
    : element{ theElement }, left{ lt }, right{ rt }, parent{par}
      { }

    BinaryNode( C && theElement, BinaryNode *lt, BinaryNode *rt,
              BinaryNode * par)
    : element{ std::move( theElement ) }, left{ lt }, right{ rt },parent{par}
      { }
  };

  public:

  class iterator
  {
  public:

    iterator () : current(nullptr) {}

    iterator (BinaryNode * t) :current(t) {}

    C & operator *() const
       {
              return current->element;
       }

    //prefix
    iterator & operator ++()
       {
              BinaryNode* parent;

              if (this->current == nullptr) {
                     /* '-> end iterator does not increment */
                     return *this;
              }
              parent = this->current->parent;
              /*
```

```
            * reaches root -> next is end()
            */
            if (parent == nullptr) {
                   this->current = nullptr;
                   return *this;
            }

            /*
            * left child -> go to right child
            * right child -> go to parent
            */
            if ((this->current == parent->left) && (parent->right != nullptr)) {
                   this->current = parent->right;
            }
            else {
                   this->current = this->current->parent;
                   return *this;
            }
            while (true) {
                   if (this->current->left != nullptr) {
                          /* '-> has left child node */
                          this->current = this->current->left;
                   }
                   else if (this->current->right != nullptr) {
                          /* '-> only right child node */
                          this->current = this->current->right;
                   }
                   else {
                          /* '-> has no children -> stop here */
                          return *this;
                   }
            }
     }

  //postfix
  iterator & operator ++(int)
     {
            iterator old(*this);
            ++(*this);
            return old;
     }

  bool operator ==(iterator other) const
  {
    return current == other.current;
  }

  bool operator != (iterator other) const
  {
    return current != other.current;
  }

protected:

  BinaryNode * current;

  // various internal functions ...
     // see Step 4 of Lab7
     bool is_root(BinaryNode *t)
     {
            //returns true when t is a pointer to the BinaryNode that is the "root"; the root is the
            //only BinaryNode that has nullptr as its parent;
            if (t->parent == nullptr)
                   return true;
```

```
            return false;
        }

        bool is_left_child(BinaryNode *t)
        {
            //returns true when t is a pointer to a BinaryNode that is the left child of its parent;
test
            //whether t's parent's left child is the same as t;
            //Fill in

            if (t->element == t->parent->left->element)
                return true;

            return false;
        }

        bool is_right_child(BinaryNode *t)
        {
            //analogous to is_left_child;

            if(t->element == t->parent->right->element)
                return true;

            return false;
        }

        BinaryNode * leftmost(BinaryNode *t)
        {
            //starting at t, follow the left children and return a pointer to the deepest leftmost
child;
            if (t->left == nullptr)
                return t;
            else
                leftmost(t->left);

            return t;
        }

        BinaryNode * follow_parents_until_left(BinaryNode *t)
        {
            //fill instarting at t, follow the parent links upwards until a BinaryNode is reached
which is a left child; return a pointer to this left child's parent;
            while (is_left_child(t) == false)
                t = t->parent;

            return t;
        }

        friend class BinarySearchTree<C>;

  };

 public:

    BinarySearchTree( ) : root{ nullptr }
    {
    }

    BinarySearchTree( const BinarySearchTree & rhs ) : root{ nullptr }
    {
        root = clone( rhs.root );
    }
```

```
BinarySearchTree( BinarySearchTree && rhs ) : root{ rhs.root }
{
    rhs.root = nullptr;
}

~BinarySearchTree( )
{
    makeEmpty( );
}

BinarySearchTree & operator=( const BinarySearchTree & rhs )
{
    BinarySearchTree copy = rhs;
    std::swap( *this, copy );
    return *this;
}

BinarySearchTree & operator=( BinarySearchTree && rhs )
{
    std::swap( root, rhs.root );
    return *this;
}

const C & findMin( ) const
{
  assert(!isEmpty());
    return findMin( root )->element;
}

const C & findMax( ) const
{

  assert(!isEmpty());
  return findMax( root )->element;
}

bool contains( const C & x ) const
{
    return contains( x, root );
}

bool isEmpty( ) const
{
    return root == nullptr;
}

void printTree( ostream & out = cout ) const
{
    if( isEmpty( ) )
        out << "Empty tree" << endl;
    else
        printTree( root, out );
}

void printInternal()
{
  printInternal(root,0);
}

void makeEmpty( )
{
    makeEmpty( root );
}
```

```
  void insert( const C & x )
  {
    insert( x, root, root   );
  }

  void insert( C && x )
  {
    insert( std::move( x ), root, root   );
  }

  void remove( const C & x )
  {
      remove( x, root );
  }

  iterator begin() const
  {
          BinaryNode *t = root;

          while (t->left != 0)
                 t = t->left;

          iterator beg(t);
          return beg;
  }

  iterator end() const
  {
          iterator end(0);
          return end;
  }

  void parent_check()
  {
    parent_check(root);
  }

private:

  BinaryNode *root;


  /**
   * Internal method to insert into a subtree.
   * x is the item to insert.
   * t is the node that roots the subtree.
   * Set the new root of the subtree.
   */
  void insert( const C & x, BinaryNode * & t, BinaryNode * & par )
  {
      if( t == nullptr )
       t = new BinaryNode{ x, nullptr, nullptr, par };
      else if( x < t->element )
       insert( x, t->left, t );
      else if( t->element < x )
       insert( x, t->right, t );
      else
          ;  // Duplicate; do nothing
  }

  /**
   * Internal method to insert into a subtree.
   * x is the item to insert.
```

```
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void insert( C && x, BinaryNode * & t, BinaryNode * & par )
{
    if( t == nullptr )
     t = new BinaryNode{ std::move( x ), nullptr, nullptr, par };
    else if( x < t->element )
     insert( std::move( x ), t->left, t );
    else if( t->element < x )
     insert( std::move( x ), t->right, t );
    else
        ;  // Duplicate; do nothing
}

/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the subtree.
 * Set the new root of the subtree.
 */
void remove( const C & x, BinaryNode * & t )
{
    if( t == nullptr )
        return;   // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode *oldNode = t;

       if (t->left != nullptr)
         {
          t->left->parent = t->parent;
          t = t->left;
         }
       else
         {
          if (t->right != 0)
            t->right->parent = t->parent;
          t = t->right;
         }

        delete oldNode;
    }
}

void parent_check(BinaryNode *t)
{
  if(t == nullptr)
   return;
  if (t->parent == nullptr)
   cout << t->element << " has parent null" << endl;
  else
   cout << t->element << " has parent " << t->parent->element << endl;
  parent_check(t->left);
  parent_check(t->right);
```

```
  return;
}


/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    if( t->left == nullptr )
        return t;
    return findMin( t->left );
}

/**
 * Internal method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */
BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != nullptr )
        while( t->right != nullptr )
            t = t->right;
    return t;
}


/**
 * Internal method to test if an item is in a subtree.
 * x is item to search for.
 * t is the node that roots the subtree.
 */
bool contains( const C & x, BinaryNode *t ) const
{
    if( t == nullptr )
        return false;
    else if( x < t->element )
        return contains( x, t->left );
    else if( t->element < x )
        return contains( x, t->right );
    else
        return true;    // Match
}

void makeEmpty( BinaryNode * & t )
{
    if( t != nullptr )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = nullptr;
}


void printTree( BinaryNode *t, ostream & out ) const
{
    if( t != nullptr )
    {
        printTree( t->left, out );
```

```
            out << t->element << endl;
            printTree( t->right, out );
        }
    }

    void printInternal(BinaryNode* t, int offset)
    {
      if (t == nullptr)
       return;

      for(int i = 1; i <= offset; i++)
       cout << "..";
      cout <<  t->element << endl;

      printInternal(t->left, offset + 1);
      printInternal(t->right, offset + 1);
    }


    BinaryNode * clone( BinaryNode *t ) const
    {
        if( t == nullptr )
            return nullptr;
        else
         return new BinaryNode{ t->element, clone( t->left ), clone( t->right ),
             clone(t->parent)};
    }
};

#endif
```

----------- MAIN.CPP

```
#include <iostream>
#include "BinarySearchTreeLab7.h"

using namespace std;

int main()
{
       BinarySearchTree<int> mybst;
       int next;
       for (int i = 1; i <= 10; i++)
       {
              cout << "Integer: ";
              cin >> next;
              cout << endl;
              mybst.insert(next);
       }
       cout << endl << "Values entered" << endl;
       mybst.printTree();
       cout << endl;
       mybst.printInternal();
       cout << endl << endl;
       cout << "And with iterators ..." << endl;
       BinarySearchTree<int>::iterator itr = mybst.begin();
       for (; itr != mybst.end(); ++itr)
              cout << *itr << endl;
       cout << endl << endl;
       cout << "Now doing some removals ..." << endl;
       for (int i = 1; i <= 3; i++)
```

```
	{
		cout << "Remove? ";
		cin >> next;
		cout << endl;
		mybst.remove(next);
	}
	cout << endl;
	mybst.printTree();
	cout << endl;
	mybst.printInternal();
	cout << endl << endl;
	cout << "And with iterators ..." << endl;
	itr = mybst.begin();
	for (; itr != mybst.end(); ++itr)
		cout << *itr << endl;
	cout << endl << endl;

	return 0;
}
```

OUTPUT

```
[005488635@csusb.edu@csevnc HW3]$ g++ Source.cpp
[005488635@csusb.edu@csevnc HW3]$ ./a.out
Integer: 1 3 5 7 9 2 4 6 8 0

Integer:
Integer:
Integer:
Integer:
Integer:
Integer:
Integer:
Integer:
Integer:

Values entered
0
1
2
3
4
5
6
7
8
9

1
..0
..3
....2
....5
......4
......7
........6
........9
..........8


And with iterators ...
0
2
4
6
8
9
7
5
3
1
```

Now doing some removals ...
Remove? 3 7 5

Remove?
Remove?

0
1
2
4
6
8
9

1
..0
..4
....2
....8
......6
......9

And with iterators ...
0
2
6
9
8
4
1

[005488635@csusb.edu@csevnc HW3]$