**CSE 330 Lab 8 Report**

Daniel Meyer

Data Structures

Fall 2017

**Status:** 100%

**Time Complexity:** O(n)

**Storage Complexity:** O(n)

**Source Code:** Pages 2 - 7

**Sample Run:** Page 8

/\*

Daniel Meyer

11-13-17

CSE 330

Fall 2017

Lab 8: Set

Problem: Implement a Set class of Nodes that is an ordered binary tree

Algorithm: A Set composed of Node class types holding a value as well

as a left and right child. The Set contains functions similar to the

STL set such as insert, erase, find, etc. An iterator is also

implemented to traverse the set.

Status: 100%

Time Complexity: O(n) Storage Complexity: O(n)

\*/

#ifndef SET\_H

#define SET\_H

// Set.h - an implementation of Set using Node and Set\_iterator

using namespace std;

template <class T> class Set;

template <class T> class Set\_iterator;

//Node class that contains a value as well as pointers to a parent, left child, and right //child nodes

template <class T>

class Node {

public:

Node() : value(0), parent(0), leftChild(0), rightChild(0) { }

Node(const T & x, Node \* p, Node \* lc, Node \* rc) :

value(x), parent(p), leftChild(lc), rightChild(rc) { }

// here copy constructor and assignment op don't make too much sense!

Node(const Node & n) :

value(n.value), parent(0), leftChild(0), rightChild(0) { }

Node & operator=(const Node & n) { value = n.value; return \*this; }

~Node() { delete leftChild; delete rightChild; } // this is very recursive, delete also calls the destructor of the object it is deleting

void insert(Node<T> \* newNode); // this is a helper func for Set::insert()

Node \* find(const T & x);

protected:

T value;

Node \* parent;

Node \* leftChild;

Node \* rightChild;

friend class Set<T>;

friend class Set\_iterator<T>;

};

//Insert function that checks current node and children for value

template <class T>

void Node<T>::insert(Node<T> \*newNode)

{

if (newNode->value <= value) {

if (leftChild != 0) {

leftChild->insert(newNode);

}

else {

newNode->parent = this;

leftChild = newNode;

}

}

else {

if (rightChild != 0) {

rightChild->insert(newNode);

}

else {

newNode->parent = this;

rightChild = newNode;

}

}

}

//Find function that checks current node and children for value

template <class T>

Node<T>\* Node<T>::find(const T &x)

{

if (x == value) {

return this;

}

if (x < value) {

if (leftChild == 0) {

return 0;

}

return leftChild->find(x);

}

if (rightChild == 0) {

return 0;

}

return rightChild->find(x);

}

//Ordered Set class in the form of a binary search tree of nodes

template <class T>

class Set {

public:

typedef Set\_iterator<T> iterator;

Set() : root(0), my\_size(0) { }

// the big three

Set(const Set<T> &);

~Set() { delete root; }

Set operator=(const Set &);

bool empty() const { return root == 0; }

unsigned int size() const { return my\_size; }

iterator insert(const T & x); // return an iterator to x if it already exists, otherwise insert and return an iterator to x

void erase(const iterator & it);

void erase(const T & x) { iterator i = find(x); erase(i); };

unsigned int count(const T & x) const; // returns 1 or 0 because this is a set, not a multi-set

iterator find(const T & x) const;

iterator begin() const; // for in-order traversal

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

protected:

Node<T> \* root;

unsigned int my\_size;

};

//Constructor setting new Set to contents of existing Set

template <class T>

Set<T>::Set(const Set<T> & op)

{

root = 0;

for (iterator i = op.begin(); i != op.end(); ++i)

insert(\*i);

}

//Overloaded equality operator that copies contents of a Set to surrent Set

template <class T>

Set<T> Set<T>::operator=(const Set<T> & op)

{

delete root;

root = 0;

for (iterator i = op.begin(); i != op.end(); ++i)

insert(\*i);

return \*this;

}

//Insert function that recursively calls Node::insert to insert a new Node

template <class T>

Set\_iterator<T> Set<T>::insert(const T &x)

{

Node<T> \*newNode = new Node<T>(x, 0, 0, 0);

if (count(x) > 0) {

return iterator(newNode);

}

if (root == 0) {

root = newNode;

}

else {

root->insert(newNode);

}

my\_size++;

return iterator(newNode);

}

//Function provided by K. Zemoudeh for erasing an element from the tree

template <class T>

void Set<T>::erase(const iterator & it)

{

if (root != 0 && it != end()) {

Node<T> \* p = it.n;

Node<T> \* left = p->leftChild;

Node<T> \* right = p->rightChild;

if (right != 0) {

// left slide

Node<T> \* left\_slide = right;

while (left\_slide->leftChild)

left\_slide = left\_slide->leftChild;

// connect the subtrees after left slide

left\_slide->leftChild = left;

if (left != 0)

left->parent = left\_slide;

// now connect right subtree to bypass p

right->parent = p->parent;

if (p->parent)

if (p->parent->leftChild == p)

p->parent->leftChild = right;

else p->parent->rightChild = right;

else // p was root

root = right;

}

else {

if (left == 0) {

if (p->parent)

if (p->parent->leftChild == p)

p->parent->leftChild = 0;

else p->parent->rightChild = 0;

else // p was root

root = 0;

}

else {

left->parent = p->parent;

if (p->parent)

if (p->parent->leftChild == p)

p->parent->leftChild = left;

else p->parent->rightChild = left;

else // p was root

root = left;

}

}

p->leftChild = 0;

p->rightChild = 0;

delete p;

my\_size--;

}

}

//Function that counts the number of times a value is in the tree

template <class T>

unsigned int Set<T>::count(const T &x) const

{

int counter = 0;

if (find(x) != 0) {

counter++;

}

return counter;

}

//Function that recursively calls Node::find until value is found or null

template <class T>

Set\_iterator<T> Set<T>::find(const T &x) const

{

if (root == 0) {

return iterator(0);

}

else {

return iterator(root->find(x));

}

}

//Function that returns a pointer to the left-most node

template <class T>

Set\_iterator<T> Set<T>::begin() const

{

iterator it(root);

while (it.n && (it.n)->leftChild) {

it.n = (it.n)->leftChild;

}

return it;

}

//Overloaded iterator class used to traverse the binary search tree

template <class T>

class Set\_iterator {

public:

Set\_iterator() : n(0) { }

Set\_iterator(Node<T> \* newNode) : n(newNode) { }

bool operator==(Set\_iterator it) const { return n == it.n; }

bool operator!=(Set\_iterator it) const { return n != it.n; }

Set\_iterator & operator++(); // inorder traversal, pre-increment

Set\_iterator operator++(int); // inorder traversal, post-increment

T & operator\*() { return n->value; }

Set\_iterator & operator=(Set\_iterator<T> it) { n = it.n; return \*this; }

protected:

Node<T> \* n;

friend class Set<T>;

};

//Overloaded pre-increment operator for inorder traversal

template <class T>

Set\_iterator<T>& Set\_iterator<T>::operator++()

{

if (n->rightChild) {

//n = n->parent;

n = n->rightChild;

while (n->leftChild) {

n = n->leftChild;

}

}

else {

Node<T> \*child = n;

n = n->parent;

while (n && n->rightChild == child) {

child = n;

n = n->parent;

}

}

return \*this;

}

//Overloaded post-increment operator for inorder traversal

template <class T>

Set\_iterator<T> Set\_iterator<T>::operator++(int)

{

Set\_iterator<T> temp = \*this;

++(\*this);

return temp;

}

#endif

**Sample Run:**

Script started on Thu 16 Nov 2017 03:26:33 PM UTC

To run a command as administrator (user "root"), use "sudo <command>".

See "man sudo\_root" for details.

]0;ubuntu@ubuntu: ~/Desktop[01;32mubuntu@ubuntu[00m:[01;34m~/Desktop[00m$ g++ Set.h

]0;ubuntu@ubuntu: ~/Desktop[01;32mubuntu@ubuntu[00m:[01;34m~/Desktop[00m$ g++ Set\_test.cpp

]0;ubuntu@ubuntu: ~/Desktop[01;32mubuntu@ubuntu[00m:[01;34m~/Desktop[00m$ ./a.out

All tests passed.

]0;ubuntu@ubuntu: ~/Desktop[01;32mubuntu@ubuntu[00m:[01;34m~/Desktop[00m$ exit

Script done on Thu 16 Nov 2017 03:26:53 PM UTC