## Project 7

● Graded

**Student**
Devin Chen

**Total Points**
**100 / 100 pts**

**Autograder Score**
**100.0 / 100.0**

**Passed Tests**
Test compiles (5/5)
Tests Skill struct (5/5)
Tests SkillTree constructors (10/10)
Tests addSkill() (10/10)
Tests findSkill() (10/10)
Tests removeSkill() (10/10)
Tests clearTree() (5/5)
Tests calculateSkillPoints() (15/15)
Tests balance() (15/15)
Tests preorderDisplay() (15/15)

### Autograder Results

**Test compiles (5/5)**

Your program compiles!

**Tests Skill struct (5/5)**

Your program passed this test.

**Tests SkillTree constructors (10/10)**

Your program passed this test.

**Tests addSkill() (10/10)**

Your program passed this test.

**Tests findSkill() (10/10)**

Your program passed this test.

**Tests removeSkill() (10/10)**

Your program passed this test.

**Tests clearTree() (5/5)**

Your program passed this test.

**Tests calculateSkillPoints() (15/15)**

Your program passed this test.

**Tests balance() (15/15)**

Your program passed this test.

**Tests preorderDisplay() (15/15)**

Your program passed this test.

**Submitted Files**

**.gitignore** Download

```
1 .DS_Store
2 .vscode
3 *.log
4
5
```

**BinaryNode.cpp** Download

```cpp
//  Created by Frank M. Carrano and Timothy M. Henry.
//  Copyright (c) 2017 Pearson Education, Hoboken, New Jersey.
//  Adapted for CSCI 235000 Project 7 Spring 2024

/** @file BinaryNode.cpp */

#include "BinaryNode.hpp"
#include <cstddef>

template<class T>
BinaryNode<T>::BinaryNode()
    : item(nullptr), leftChildPtr(nullptr), rightChildPtr(nullptr)
{ }  // end default constructor

template<class T>
BinaryNode<T>::BinaryNode(const T& anItem)
    : item(anItem), leftChildPtr(nullptr), rightChildPtr(nullptr)
{ }  // end constructor

template<class T>
BinaryNode<T>::BinaryNode(const T& anItem,
                          std::shared_ptr<BinaryNode<T>> leftPtr,
                          std::shared_ptr<BinaryNode<T>> rightPtr)
    : item(anItem), leftChildPtr(leftPtr), rightChildPtr(rightPtr)
{ }  // end constructor

template<class T>
void BinaryNode<T>::setItem(const T& anItem)
{
  item = anItem;
}  // end setItem

template<class T>
T BinaryNode<T>::getItem() const
{
  return item;
}  // end getItem

template<class T>
bool BinaryNode<T>::isLeaf() const
{
  return ((leftChildPtr == nullptr) && (rightChildPtr == nullptr));
}

template<class T>
void BinaryNode<T>::setLeftChildPtr(std::shared_ptr<BinaryNode<T>> leftPtr)
{
  leftChildPtr = leftPtr;
}  // end setLeftChildPtr
```

```
50
51  template<class T>
52  void BinaryNode<T>::setRightChildPtr(std::shared_ptr<BinaryNode<T>> rightPtr)
53  {
54    rightChildPtr = rightPtr;
55  } // end setRightChildPtr
56
57  template<class T>
58  std::shared_ptr<BinaryNode<T>> BinaryNode<T>::getLeftChildPtr() const
59  {
60    return leftChildPtr;
61  } // end getLeftChildPtr
62
63  template<class T>
64  std::shared_ptr<BinaryNode<T>> BinaryNode<T>::getRightChildPtr() const
65  {
66    return rightChildPtr;
67  } // end getRightChildPtr
68
69
```

**BinaryNode.hpp** Download

```cpp
//  Created by Frank M. Carrano and Timothy M. Henry.
//  Copyright (c) 2017 Pearson Education, Hoboken, New Jersey.
//  Adapted for CSCI 235000 Project 7 Spring 2024

/** A class of nodes for a link-based binary tree.
 Listing 16-2.
 @file BinaryNode.hpp */

#ifndef BINARY_NODE_
#define BINARY_NODE_

#include <memory>

template<class T>
class BinaryNode
{
private:
   T item;          // Data portion
   std::shared_ptr<BinaryNode<T>> leftChildPtr;   // Pointer to left child
   std::shared_ptr<BinaryNode<T>> rightChildPtr;  // Pointer to right child

public:
   BinaryNode();
   BinaryNode(const T& anItem);
   BinaryNode(const T& anItem, std::shared_ptr<BinaryNode<T>> leftPtr,
std::shared_ptr<BinaryNode<T>> rightPtr);

   void setItem(const T& anItem);
   T getItem() const;

   bool isLeaf() const;

   std::shared_ptr<BinaryNode<T>> getLeftChildPtr() const;
   std::shared_ptr<BinaryNode<T>> getRightChildPtr() const;

   void setLeftChildPtr(std::shared_ptr<BinaryNode<T>> leftPtr);
   void setRightChildPtr(std::shared_ptr<BinaryNode<T>> rightPtr);
}; // end BinaryNode

#include "BinaryNode.cpp"

#endif

```

**BinarySearchTree.cpp** Download

```cpp
/*
BST class modified for Project 7
CSCI 235 Spring 2024
*/

#include "BinarySearchTree.hpp"
#include <vector>


/*CONSTRUCTRS*/

template <class T>
BinarySearchTree<T>::BinarySearchTree() : root_ptr_(nullptr)
{
} // end default constructor

template <class T>
BinarySearchTree<T>::BinarySearchTree(const T &root_item)
    : root_ptr_(std::make_shared<BinaryNode<T>>(root_item, nullptr, nullptr))
{
} // end constructor

template <class T>
BinarySearchTree<T>::BinarySearchTree(const BinarySearchTree &another_tree)
{
  root_ptr_ = copyTree(another_tree.root_ptr_); // Call helper method
} // end copy constructor



/*PUBLIC METHODS*/

 /** @return root_ptr_ **/
template <class T>
std::shared_ptr<BinaryNode<T>> BinarySearchTree<T>::getRoot() const
{
  return root_ptr_;
}

/** @return true if the BinarySearchTree is emtpy, false otherwise **/
template <class T>
bool BinarySearchTree<T>::isEmpty() const
{
  return root_ptr_ == nullptr;
} // end isEmpty


/** @return the height of the BST structure as the number of nodes on the longest path from root to leaf**/
```

```
49  template <class T>
50  int BinarySearchTree<T>::getHeight() const
51  {
52    return this->getHeightHelper(root_ptr_); // Call helper method
53  } // end getHeight
54
55
56  /** @return the number of Nodes in the BST structure**/
57  template <class T>
58  int BinarySearchTree<T>::getNumberOfNodes() const
59  {
60    return this->getNumberOfNodesHelper(root_ptr_); // Call helper method
61  } // end getNumberOfNodes
62
63
64  /** @param a new entry to be added to the BST
65      @post new entry is added to the BST retaining the
66            BST property, s.t. at any node, all Items in
67            its left subtree are < the item at that node
68            and all items in its right subtree are >
69            Note: > and < would need to be overloaded for self made data types
70      **/
71  template <class T>
72  void BinarySearchTree<T>::add(const T &new_entry)
73  {
74    std::shared_ptr<BinaryNode<T>> new_node_ptr = std::make_shared<BinaryNode<T>>(new_entry);
75    root_ptr_ = placeNode(root_ptr_, new_node_ptr);
76  } // end add
77
78
79    /** @param entry to be removed from the BST
80        @post entry is removed from the BST and retaining its
81            BST property, s.t. at any node, all Nodes in
82            its left subtree are < the item at that node
83            and all items in its right subtree are >**/
84  template <class T>
85  bool BinarySearchTree<T>::remove(const T &entry)
86  {
87    bool is_successful = false;
88    // call may change is_successful
89    root_ptr_ = removeValue(root_ptr_, entry, is_successful);
90    return is_successful;
91  } // end remove
92
93
94    /** @param entry to be found in the BST
95        @return true if entry is found in the BST, false otherwise**/
96  template <class T>
97  bool BinarySearchTree<T>::contains(const T &entry) const
98  {
99    return (findNode(root_ptr_, entry) != nullptr);
100 } // end contains
```

```
101
102
103 /**
104  * @param: sets the root pointer to the parameter
105  */
106 template <class T>
107 void BinarySearchTree<T>::setRoot(std::shared_ptr<BinaryNode<T>> new_root_ptr)
108 {
109   root_ptr_ = new_root_ptr;
110 }
111
112
113
114
115
116 /*PRIVATE METHODS*/
117
118
119
120
121  /** called by copy constructor
122      @param old_tee_root_ptr a pointer to the root of the tree to be copied
123      @post recursively copies every node in the tree pointed to by the parameter pointer
124      @return a pointer to the root of the copied subtree
125     **/
126 template <class T>
127 std::shared_ptr<BinaryNode<T>> BinarySearchTree<T>::copyTree(const
    std::shared_ptr<BinaryNode<T>> old_tee_root_ptr) const
128 {
129   std::shared_ptr<BinaryNode<T>> new_tree_ptr;
130
131   // Copy tree nodes during a preorder traversal
132   if (old_tee_root_ptr != nullptr)
133   {
134     // Copy node
135     new_tree_ptr = std::make_shared<BinaryNode<T>>(old_tee_root_ptr->getItem(), nullptr, nullptr);
136     new_tree_ptr->setLeftChildPtr(copyTree(old_tee_root_ptr->getLeftChildPtr()));
137     new_tree_ptr->setRightChildPtr(copyTree(old_tee_root_ptr->getRightChildPtr()));
138   } // end if
139
140   return new_tree_ptr;
141 } // end copyTree
142
143
144   /** called by getHeight
145      @param subtree_ptr a pointer to the root of the current subtree
146      @return the height of the BST structure
147      as the number of nodes on the longest path
148      from root to leaf**/
149 template <class T>
150 int BinarySearchTree<T>::getHeightHelper(std::shared_ptr<BinaryNode<T>> subtree_ptr) const
151 {
```

```
152   if (subtree_ptr == nullptr)
153     return 0;
154   else
155     return 1 + std::max(getHeightHelper(subtree_ptr->getLeftChildPtr()),
      getHeightHelper(subtree_ptr->getRightChildPtr()));
156 } // end getHeightHelper
157
158
159 /** called by getNumberOfNodes
160     @param subtree_ptr a pointer to the root of the current subtree
161     @return the number of nodes in the tree**/
162 template <class T>
163 int BinarySearchTree<T>::getNumberOfNodesHelper(std::shared_ptr<BinaryNode<T>> subtree_ptr)
      const
164 {
165   if (subtree_ptr == nullptr)
166     return 0;
167   else
168     return 1 + getNumberOfNodesHelper(subtree_ptr->getLeftChildPtr()) +
      getNumberOfNodesHelper(subtree_ptr->getRightChildPtr());
169 } // end getNumberOfNodesHelper
170
171
172 /** called by add(new_entry)
173      @param subtree_ptr a pointer to the subtree in which to place the new node
174      @param new_node_ptr a pointer to the new node to be added to the tree
175      @post recursively places the new node as a leaf retaining the BST property
176      @return a pointer to the root of the subtree in which node was placed
177     **/
178 template <class T>
179 std::shared_ptr<BinaryNode<T>> BinarySearchTree<T>::placeNode(std::shared_ptr<BinaryNode<T>>
      subtree_ptr, std::shared_ptr<BinaryNode<T>> new_node_ptr)
180 {
181   if (subtree_ptr == nullptr)
182     return new_node_ptr;
183   else
184   {
185     if (subtree_ptr->getItem() > new_node_ptr->getItem())
186       subtree_ptr->setLeftChildPtr(placeNode(subtree_ptr->getLeftChildPtr(), new_node_ptr));
187     else
188       subtree_ptr->setRightChildPtr(placeNode(subtree_ptr->getRightChildPtr(), new_node_ptr));
189     return subtree_ptr;
190   }
191 } // end placeNode
192
193
194     /** called by contains
195       @param subtree_ptr a pointer to the subtree to be searched
196       @param target a reference to the item to be found
197       @return a pointer to the node containing the target, nullptr if not found
198      **/
199 template <class T>
```

```
200 std::shared_ptr<BinaryNode<T>> BinarySearchTree<T>::findNode(std::shared_ptr<BinaryNode<T>>
    subtree_ptr, const T &target) const
201 {
202   // Uses a binary search
203   if (subtree_ptr == nullptr)
204     return subtree_ptr; // Not found
205   else if ((subtree_ptr->getItem() == target))
206     return subtree_ptr; // Found
207   else if (subtree_ptr->getItem() > target)
208     // Search left subtree
209     return findNode(subtree_ptr->getLeftChildPtr(), target);
210   else
211     // Search right subtree
212     return findNode(subtree_ptr->getRightChildPtr(), target);
213 } // end findNode
214
215
216 /** called by removeNode
217      @param node_ptr a pointer to the node to be removed
218      @param inorder_successor a reference to the  inorder successor (the smallest value in the left
    subtree) of the node to be deleted
219      @post removes the node containing the inorder successor
220      @return a pointer to the subtree after inorder successor node has been deleted
221     **/
222 template <class T>
223 std::shared_ptr<BinaryNode<T>>
    BinarySearchTree<T>::removeLeftmostNode(std::shared_ptr<BinaryNode<T>> node_ptr, T
    &inorder_successor)
224 {
225   if (node_ptr->getLeftChildPtr() == nullptr)
226   {
227     inorder_successor = node_ptr->getItem();
228     return removeNode(node_ptr);
229   }
230   else
231   {
232     node_ptr->setLeftChildPtr(removeLeftmostNode(node_ptr->getLeftChildPtr(), inorder_successor));
233     return node_ptr;
234   } // end if
235 } // end removeLeftmostNode
236
237
238
239    /** called by removeValue
240      @param node_ptr a pointer to the node to be removed
241      @post removed the node pointed to by parameter retaining the BST property
242      @return a pointer to the subtree after node has been removed
243     **/
244 template <class T>
245 std::shared_ptr<BinaryNode<T>>
    BinarySearchTree<T>::removeNode(std::shared_ptr<BinaryNode<T>> node_ptr)
246 {
```

```
247   // Case 1) Node is a leaf - it is deleted
248   if (node_ptr->isLeaf())
249   {
250     node_ptr.reset();
251     return node_ptr; // delete and return nullptr
252   }
253   // Case 2) Node has one child - parent adopts child
254   else if (node_ptr->getLeftChildPtr() == nullptr) // Has rightChild only
255   {
256     return node_ptr->getRightChildPtr();
257   }
258   else if (node_ptr->getRightChildPtr() == nullptr) // Has left child only
259   {
260     return node_ptr->getLeftChildPtr();
261   }
262   // Case 3) Node has two children: Find successor node.
263   else
264   {
265     // Traditional way to remove a value in a node with two children
266     T new_node_value;
267     node_ptr->setRightChildPtr(removeLeftmostNode(node_ptr->getRightChildPtr(),
      new_node_value));
268     node_ptr->setItem(new_node_value);
269     return node_ptr;
270   } // end if
271 } // end removeNode
272
273
274 /** called by remove
275       @param subtree_ptr a pointer to the subtree in which to look for the value to be removed
276       @param target the item to be removed
277       @param success a flag to indicate that item was successfully removed
278       @return a pointer to the subtree in which target is found
279      **/
280 template <class T>
281 std::shared_ptr<BinaryNode<T>>
    BinarySearchTree<T>::removeValue(std::shared_ptr<BinaryNode<T>> subtree_ptr, const T target,
    bool &success)
282 {
283   if (subtree_ptr == nullptr)
284   {
285     // Not found here
286     success = false;
287     return subtree_ptr;
288   }
289   if ((subtree_ptr->getItem() == target))
290   {
291     // Item is in the root of some subtree
292     subtree_ptr = removeNode(subtree_ptr);
293     success = true;
294     return subtree_ptr;
295   }
```

```
296   else
297   {
298     if (subtree_ptr->getItem() > target)
299     {
300       // Search the left subtree
301       subtree_ptr->setLeftChildPtr(removeValue(subtree_ptr->getLeftChildPtr(), target, success));
302     }
303     else
304     {
305       // Search the right subtree
306       subtree_ptr->setRightChildPtr(removeValue(subtree_ptr->getRightChildPtr(), target, success));
307     }
308     return subtree_ptr;
309   }
310 } // end removeValue
311
312
313
314
```

**BinarySearchTree.hpp** Download

```
/*
BST class modified for Project 7
CSCI 235 Spring 2024
*/

#ifndef BINARY_SEARCH_TREE_
#define BINARY_SEARCH_TREE_

#include "BinaryNode.hpp"
#include <iostream>

template <class T>
class BinarySearchTree
{
public:
  /*Constructors*/
  BinarySearchTree();                              //default constructor
  BinarySearchTree(const T &root_item);            //parameterized constructor
  BinarySearchTree(const BinarySearchTree &another_tree); //copy constructor

  /** @return root_ptr_ **/
  std::shared_ptr<BinaryNode<T>> getRoot() const;

  /** @return true if the BinarySearchTree is emtpy, false otherwise **/
  bool isEmpty() const;

  /** @return the height of the BST structure as the number of nodes on the longest path from root to leaf**/
  int getHeight() const;

  /** @return the number of Nodes in the BST structure**/
  int getNumberOfNodes() const;

  /** @param a new entry to be added to the BST
     @post new entry is added to the BST retaining the
           BST property, s.t. at any node, all Items in
           its left subtree are < the item at that node
           and all items in its right subtree are >
           Note: > and < would need to be overloaded for self made data types
    **/
  void add(const T &new_entry);

  /** @param entry to be removed from the BST
     @post entry is removed from the BST and retaining its
           BST property, s.t. at any node, all Nodes in
           its left subtree are < the item at that node
           and all items in its right subtree are >**/
  bool remove(const T &entry);

```

```
49
50    /** @param entry to be found in the BST
51        @return true if entry is found in the BST, false otherwise**/
52    bool contains(const T &entry) const;
53
54
55
56
57    /**
58     * @param: sets the root pointer to the parameter
59     */
60    void setRoot(std::shared_ptr<BinaryNode<T>> new_root_ptr);
61
62  protected:
63    std::shared_ptr<BinaryNode<T>> root_ptr_;
64
65    /** called by copy constructor
66        @param old_tee_root_ptr a pointer to the root of the tree to be copied
67        @post recursively copies every node in the tree pointed to by the parameter pointer
68        @return a pointer to the root of the copied subtree
69       **/
70    std::shared_ptr<BinaryNode<T>> copyTree(const std::shared_ptr<BinaryNode<T>> old_tee_root_ptr)
  const;
71
72
73    /** called by getHeight
74       @param subtree_ptr a pointer to the root of the current subtree
75       @return the height of the BST structure
76       as the number of nodes on the longest path
77       from root to leaf**/
78    int getHeightHelper(std::shared_ptr<BinaryNode<T>> subtree_ptr) const;
79
80    /** called by getNumberOfNodes
81       @param subtree_ptr a pointer to the root of the current subtree
82       @return the number of nodes in the tree**/
83    int getNumberOfNodesHelper(std::shared_ptr<BinaryNode<T>> subtree_ptr) const;
84
85    /** called by add(new_entry)
86        @param subtree_ptr a pointer to the subtree in which to place the new node
87        @param new_node_ptr a pointer to the new node to be added to the tree
88        @post recursively places the new node as a leaf retaining the BST property
89        @return a pointer to the root of the subtree in which node was placed
90       **/
91    std::shared_ptr<BinaryNode<T>> placeNode(std::shared_ptr<BinaryNode<T>> subtree_ptr,
  std::shared_ptr<BinaryNode<T>> new_node_ptr);
92
93
94    /** called by remove
95        @param subtree_ptr a pointer to the subtree in which to look for the value to be removed
96        @param target the item to be removed
97        @param success a flag to indicate that item was successfully removed
98        @return a pointer to the subtree in which target is found
```

```
99      **/
100   std::shared_ptr<BinaryNode<T>> removeValue(std::shared_ptr<BinaryNode<T>> subtree_ptr, const
      T target, bool &success);
101
102   /** called by removeValue
103       @param node_ptr a pointer to the node to be removed
104       @post removed the node pointed to by parameter retaining the BST property
105       @return a pointer to the subtree after node has been removed
106      **/
107   std::shared_ptr<BinaryNode<T>> removeNode(std::shared_ptr<BinaryNode<T>> node_ptr);
108
109
110   /** called by removeNode
111       @param node_ptr a pointer to the node to be removed
112       @param inorder_successor a reference to the  inorder successor (the smallest value in the left
      subtree) of the node to be deleted
113       @post removes the node containing the inorder successor
114       @return a pointer to the subtree after inorder successor node has been deleted
115      **/
116   std::shared_ptr<BinaryNode<T>> removeLeftmostNode(std::shared_ptr<BinaryNode<T>> node_ptr,
      T &inorder_successor);
117
118   /** called by contains
119       @param subtree_ptr a pointer to the subtree to be searched
120       @param target a reference to the item to be found
121       @return a pointer to the node containing the target, nullptr if not found
122      **/
123   std::shared_ptr<BinaryNode<T>> findNode(std::shared_ptr<BinaryNode<T>> subtree_ptr, const T
      &target) const;
124
125
126
127
128 };
129 #include "BinarySearchTree.cpp"
130 #endif
```

**Makefile** Download

```
CXX = g++
CXXFLAGS = -std=c++17 -g -Wall -O2

PROG ?= main
OBJS = main.o SkillTree.o

all: $(PROG)

.cpp.o:
	$(CXX) $(CXXFLAGS) -c -o $@ $<

$(PROG): $(OBJS)
	$(CXX) $(CXXFLAGS) -o $@ $(OBJS)

clean:
	rm -rf $(EXEC) *.o *.out main

rebuild: clean all

```

**README.md** Download

```
[![Review Assignment Due Date](https://classroom.github.com/assets/deadline-readme-button-24ddc0f5d75046c5622901739e7c5dd533143b0c8e959d652212380cedb1ea36.svg)](https://classroom.github.com/a/q5b9YE0w)
# Project7
The project specification can be found on Blackboard.

```

SkillTree.cpp Download

```cpp
/**
 * @file SkillTree.cpp
 * @author Devin Chen
 * @brief SkillTree Class
 * @date 5/6/2024
*/

#include "SkillTree.hpp"
#pragma once
/**
  * Default Constructor
  */
SkillTree::SkillTree(){}
/**
 * @param: A const reference to string: the name of a csv file
 * @post: The SkillTree is populated with Skills from the csv file
 * The file format is as follows:
 * id,name,description,leveled
 * Ignore the first line. Each subsequent line represents a Skill to be added to the SkillTree.
*/
SkillTree::SkillTree(const std::string &csv_file){
 std::ifstream input(csv_file);
        std::string ignoredline;
        std::getline(input, ignoredline); // Ignore first line
        std::string line;

        while (std::getline(input, line)){
            std::istringstream stream(line);
            std::string ID, NAME, DESCRIPTION, LEVEL, Temp;

            std::getline(stream, Temp, ',');
            ID = Temp;
            std::getline(stream, Temp, ',');
            NAME = Temp;
            std::getline(stream, Temp, ',');
            DESCRIPTION = Temp;
            std::getline(stream, Temp, ',');
            LEVEL = Temp;

            int id;
            bool leveled;

            if(std::stoi(LEVEL) == 0){
                leveled = false;
            }
            else{
                leveled = true;
            }

```

```
50            id = std::stoi(ID);
51
52            add(Skill(id, NAME, DESCRIPTION, leveled));
53        }
54    }
55    /**
56     * @param: A const reference to int representing the id of the Skill to be found
57     * @return: A pointer to the node containing the Skill with the given id, or nullptr if the Skill is not
      found
58     */
59    BinaryNode<Skill>* SkillTree::findSkill(const int &id){
60        if(isEmpty()){
61            return nullptr;
62        }
63        if(id == 0){
64            return nullptr;
65        }
66        std::shared_ptr<BinaryNode<Skill>> Temp = getRoot();
67        while(Temp!= nullptr){
68            if((Temp.get()->getItem()).id_ == id){
69                return Temp.get();
70            }
71            else if(id < (Temp.get()->getItem()).id_){
72                Temp = Temp.get()->getLeftChildPtr();
73            }
74            else{
75                Temp = Temp.get()->getRightChildPtr();
76            }
77        }
78        return nullptr;
79    }
80    /**
81      * @param: A const reference to Skill
82      * @post: The Skill is added to the tree (in BST order as implemented in the base class) only if it
      was not already in the tree. Note that all comparisons are id-based as implemented by the Skill
      comparison operators.
83      * @return: True if the Skill is successfully added to the SkillTree, false otherwise
84      */
85    bool SkillTree::addSkill(const Skill &skill){
86        if(findSkill(skill.id_) != nullptr){
87            return false;
88        }
89        else{
90            add(skill);
91            return true;
92        }
93    }
94    /**
95      * @param: A const reference to string: the name of a Skill
96      * @return: True if the Skill is successfully removed from the SkillTree, false otherwise
97      */
98    bool SkillTree::removeSkill(const std::string &skill_name){
```

```
99      std::shared_ptr<BinaryNode<Skill>> Main = removeskillhelper(getRoot(), skill_name);
100     if(Main != nullptr){
101         remove(Main.get()->getItem());
102         return true;
103     }
104     if(Main == nullptr){
105         return false;
106     }
107 
108 }
109 /**
110   * @param: A shared_ptr a node on the Binary tree and a string
111   * @post: Traverse the tree searching if the item's name in the node matches the string
112   * @return: Return the node when the item's name matches with the string.
113   */
114 std::shared_ptr<BinaryNode<Skill>> SkillTree::removeskillhelper(std::shared_ptr<BinaryNode<Skill>>
    node, std::string name){
115     if(node == nullptr){ // Base if the root is Null <- end of the tree and we didn't find it
116         return nullptr;
117     }
118     if(node.get()->getItem().name_ == name){// Node is found now returning it
119         return node;
120     }
121     else{// Node is not found search through left and right node
122         std::shared_ptr<BinaryNode<Skill>> left = removeskillhelper(node.get()->getLeftChildPtr(),
    name);
123         std::shared_ptr<BinaryNode<Skill>> right = removeskillhelper(node.get()->getRightChildPtr(),
    name);
124 
125         if(right != nullptr){
126             return right;
127         }
128         if(left != nullptr){
129             return left;
130         }
131     }
132     return nullptr;
133 }
134 /**
135   * @post: Clears the tree
136   */
137 void SkillTree::clear(){
138     setRoot(nullptr);
139 }
140 /**
141   * @param: A const reference to int representing the id of a Skill
142     * Note: For a Skill to be leveled up, its parent Skill must also be leveled up, thus the Skill points
    are the number of Skills that must be leveled up before and including the Skill with the given id.
143   * @return: an integer: the number of skill points needed to level up the Skill with the given id,
    starting from the root (i.e. the number of nodes from root to the given Skill).
144   * Include the Skill with the given id in the count. For example, if the tree contains the following
    Skills (represented by their ids):
```

```
145   *   5
146   *  / \
147   * 1   8
148   * and the function parameter queries for id 8, the function should return 2.
149   * Disregard the leveled_ field of the existing Skills in the tree.
150   * If the Skill with the given id is not found, return -1.
151   */
152 int SkillTree::calculateSkillPoints(const int &id){
153     if(findSkill(id) == nullptr){
154         return -1;
155     }
156     if(isEmpty()){
157         return -1;
158     }
159     int returnnum = 1;
160     std::shared_ptr<BinaryNode<Skill>> Temp = getRoot();
161     while(Temp!= nullptr){
162         if(Temp.get()->getItem().id_ == id){
163             break;
164         }
165         else if(Temp.get()->getItem().id_ < id){
166             Temp = Temp.get()->getLeftChildPtr();
167             returnnum++;
168         }
169         else{
170             Temp = Temp.get()->getRightChildPtr();
171             returnnum++;
172         }
173     }
174     return returnnum;
175 }
176 /**
177   * @post: prints the tree in preorder, in the format:
178   [id_] [name_]\n
179   [description_]\n
180   [leveled_]
181   */
182 void SkillTree::preorderDisplay(){
183     std::shared_ptr<BinaryNode<Skill>> node = getRoot();
184     preorderDisplayHelper(node);
185 }
186 /**
187   * @param: A shared_ptr a node on the Binary tree
188   * @post: Traverse the tree in preorder and displaing the item in format
189   [id_] [name_]\n
190   [description_]\n
191   [leveled_]
192   */
193 void SkillTree::preorderDisplayHelper(std::shared_ptr<BinaryNode<Skill>> node){
194           std::cout << "  " << node.get()->getItem().id_  << " " << node.get()->getItem().name_ << "\n";
195           std::cout << "  " << node.get()->getItem().description_  << "\n";
196           std::cout << "  " << node.get()->getItem().leveled_;
```

```
197     if(node != nullptr){//check if it is a nullptr so there won't be segfault
198         if(node.get()->getLeftChildPtr() != nullptr){//check if it has left ptr
199             preorderDisplayHelper(node.get()->getLeftChildPtr());
200         }
201         if(node.get()->getRightChildPtr() != nullptr){//check if it has right ptr
202             preorderDisplayHelper(node.get()->getRightChildPtr());
203         }
204 
205         //print current skill
206     }
207 }
208 /**
209  * @post: Balances the tree. Recall the definition of a balanced tree:
210  * A tree is balanced if for any node, its left and right subtrees differ in height by no more than 1. * All paths from root of subtrees to leaf differ in length by at most 1
211  * Hint: You may sort the elements and build the tree recursively using a divide and conquer approach
212  */
213 void SkillTree::balance(){
214     std::vector<std::shared_ptr<BinaryNode<Skill>>> vec = Inorder(root_ptr_);
215     setRoot(sortedArrayToBST(vec, 0 , vec.size()-1));
216 }
217 /**
218  * @param: A shared_ptr a node on the Binary tree, a pointer to a vecotor
219  * @post: Traverse Breath First search and push nodes into the vecotor
220  */
221  // adapted from :https://stackoverflow.com/questions/63720121/c-how-to-store-the-nodes-of-an-in-order-traversal-into-an-array
222 void SkillTree::InorderRecursive(std::shared_ptr<BinaryNode<Skill>> root, std::vector<std::shared_ptr<BinaryNode<Skill>>>& nodes)
223 {
224     if (root == NULL)
225         return;
226 
227     InorderRecursive(root->getLeftChildPtr(), nodes); //visit left sub-tree
228 
229     nodes.push_back(root); // Corrected this line to dereference root
230 
231     InorderRecursive(root->getRightChildPtr(), nodes); //visit right sub-tree
232 }
233 /**
234  * @param: A shared_ptr a node on the Binary tree
235  * @post: Calls InorderRecrusive
236  * @return: A vector containing Binary Nodes
237  */
238  // adapted from :https://stackoverflow.com/questions/63720121/c-how-to-store-the-nodes-of-an-in-order-traversal-into-an-array
239 std::vector<std::shared_ptr<BinaryNode<Skill>>>
    SkillTree::Inorder(std::shared_ptr<BinaryNode<Skill>> root)
240 {
241     std::vector<std::shared_ptr<BinaryNode<Skill>>> nodes;
242     InorderRecursive(root, nodes);
```

```
243     return nodes;
244 }
245 /**
246  * @param: Address to a vecotor containing nodes, int representing the start and end of the vector
247  * @post: Taking the middle of the vector and assigning it as the root. This way the BST is now able
    to be balanced.
248  * as each left and right will have the maximum distance of 1.
249  * @return A shared_ptr that is the new root.
250  */
251 // adapted from :
    https://github.com/mandliya/algorithms_and_data_structures/blob/master/tree_problems/sortedArr
252 std::shared_ptr<BinaryNode<Skill>>
    SkillTree::sortedArrayToBST(std::vector<std::shared_ptr<BinaryNode<Skill>>>& arr, int start, int end)
    {
253     if (start > end) {
254         return nullptr;
255     }
256     int mid = (start + end) / 2;
257     std::shared_ptr<BinaryNode<Skill>> node = std::make_shared<BinaryNode<Skill>>(*arr[mid]);
258     node->setLeftChildPtr(sortedArrayToBST(arr, start, mid - 1));
259     node->setRightChildPtr(sortedArrayToBST(arr, mid + 1, end));
260     return node;
261 }
262
```

SkillTree.hpp

Download

```cpp
/**
 * @file SkillTree.hpp
 * @author Devin Chen
 * @brief SkillTree Class
 * @date 5/6/2024
*/



#include "BinarySearchTree.hpp"
#include <string>
#include <fstream>
#include <sstream>
#include <string>
#include <iostream>
#pragma once

struct Skill
{
public:
    /**
     * @param: A const reference to Skill
     * @return: True if the id_ of the Skill is equal to that of the argument, false otherwise
     */
    bool operator==(const Skill &SKILL)const{
        return SKILL.id_ == id_;
    }
    /**
     * @param: A const reference to Skill
     * @return: True if the id_ of the Skill is less than that of the argument, false otherwise
     */
    bool operator<(const Skill &SKILL)const{
        return SKILL.id_ > id_;
    }
    /**
     * @param: A const reference to Skill
     * @return: True if the id_ of the Skill is greater than that of the argument, false otherwise
     */
    bool operator>(const Skill &SKILL)const{
        return SKILL.id_ < id_;
    }
    // Default constructor
    Skill():id_{0},name_{""},description_{""},leveled_{false}{}
    // Parameterized constructor
    /**
     * @param id: The unique identifier for the Skill
     * @param name: The name of the Skill
     * @param description: The description of the Skill
     * @param leveled: Whether or not the Skill is leveled up
```

```
50      */
51      Skill(int ID, std::string SKILLNAME, std::string DESCRIPTION, bool LEVEL){
52          id_ = ID;
53          name_ = SKILLNAME;
54          description_ = DESCRIPTION;
55          leveled_ = LEVEL;
56      }
57
58      int id_; // A unique identifier for the Skill
59      std::string name_; // The name of the Skill
60      std::string description_; // The description of the Skill
61      bool leveled_; // Whether or not the Skill is leveled up
62  };
63
64
65  class SkillTree : public BinarySearchTree<Skill>{
66
67  public:
68  /**
69   * Default Constructor
70   */
71      SkillTree();
72  /**
73   * @param: A const reference to string: the name of a csv file
74   * @post: The SkillTree is populated with Skills from the csv file
75   * The file format is as follows:
76   * id,name,description,leveled
77   * Ignore the first line. Each subsequent line represents a Skill to be added to the SkillTree.
78  */
79      SkillTree(const std::string &csv_file);
80  /**
81   * @param: A const reference to int representing the id of the Skill to be found
82   * @return: A pointer to the node containing the Skill with the given id, or nullptr if the Skill is not
    found
83  */
84      BinaryNode<Skill>* findSkill(const int &id);
85  /**
86    * @param: A const reference to Skill
87    * @post: The Skill is added to the tree (in BST order as implemented in the base class) only if it
    was not already in the tree. Note that all comparisons are id-based as implemented by the Skill
    comparison operators.
88    * @return: True if the Skill is successfully added to the SkillTree, false otherwise
89    */
90      bool addSkill(const Skill &skill);
91  /**
92    * @param: A const reference to string: the name of a Skill
93    * @return: True if the Skill is successfully removed from the SkillTree, false otherwise
94    */
95      bool removeSkill(const std::string &skill_name);
96  /**
97    * @post: Clears the tree
98    */
```

```
99      void clear();
100 /**
101   * @param: A const reference to int representing the id of a Skill
102     * Note: For a Skill to be leveled up, its parent Skill must also be leveled up, thus the Skill points are the number of Skills that must be leveled up before and including the Skill with the given id.
103   * @return: an integer: the number of skill points needed to level up the Skill with the given id, starting from the root (i.e. the number of nodes from root to the given Skill).
104   * Include the Skill with the given id in the count. For example, if the tree contains the following Skills (represented by their ids):
105   *   5
106   *  / \
107   * 1   8
108   * and the function parameter queries for id 8, the function should return 2.
109   * Disregard the leveled_ field of the existing Skills in the tree.
110   * If the Skill with the given id is not found, return -1.
111   */
112     int calculateSkillPoints(const int &id);
113 /**
114   * @post: Balances the tree. Recall the definition of a balanced tree:
115   * A tree is balanced if for any node, its left and right subtrees differ in height by no more than 1. * All paths from root of subtrees to leaf differ in length by at most 1
116   * Hint: You may sort the elements and build the tree recursively using a divide and conquer approach
117   */
118     void balance();
119 /**
120   * @post: prints the tree in preorder, in the format:
121   [id_] [name_]\n
122   [description_]\n
123   [leveled_]
124   */
125     void preorderDisplay();
126 /**
127   * @param: A shared_ptr a node on the Binary tree and a string
128   * @post: Traverse the tree searching if the item's name in the node matches the string
129   * @return: Return the node when the item's name matches with the string.
130   */
131     std::shared_ptr<BinaryNode<Skill>> removeskillhelper(std::shared_ptr<BinaryNode<Skill>> node, std::string);
132 /**
133   * @param: A shared_ptr a node on the Binary tree
134   * @post: Traverse the tree in preorder and displaing the item in format
135   [id_] [name_]\n
136   [description_]\n
137   [leveled_]
138   */
139     void preorderDisplayHelper(std::shared_ptr<BinaryNode<Skill>> node);
140 /**
141   * @param: A shared_ptr a node on the Binary tree
142   * @post: Calls InorderRecrusive
143   * @return: A vector containing Binary Nodes
144   */
```

```
145  // adapted from :https://stackoverflow.com/questions/63720121/c-how-to-store-the-nodes-of-an-in-order-traversal-into-an-array
146     std::vector<std::shared_ptr<BinaryNode<Skill>>>Inorder(std::shared_ptr<BinaryNode<Skill>> root);
147  /**
148   * @param: A shared_ptr a node on the Binary tree, a pointer to a vecotor
149   * @post: Traverse Breath First search and push nodes into the vecotor
150   */
151  // adapted from :https://stackoverflow.com/questions/63720121/c-how-to-store-the-nodes-of-an-in-order-traversal-into-an-array
152     void InorderRecursive(std::shared_ptr<BinaryNode<Skill>> root, std::vector<std::shared_ptr<BinaryNode<Skill>>>& nodes);
153  /**
154   * @param: Address to a vecotor containing nodes, int representing the start and end of the vector
155   * @post: Taking the middle of the vector and assigning it as the root. This way the BST is now able to be balanced.
156   * as each left and right will have the maximum distance of 1.
157   * @return A shared_ptr that is the new root.
158   */
159  // adapted from : https://github.com/mandliya/algorithms_and_data_structures/blob/master/tree_problems/sortedArr
160     std::shared_ptr<BinaryNode<Skill>> sortedArrayToBST( std::vector<std::shared_ptr<BinaryNode<Skill>>> & arr, int start, int end);
161  };
162
```

**SkillTree.o** Download

```
1  Large file hidden. You can download it using the button above.
```

**debug.csv** Download

```
1  id,name,description,leveled
2  2,skill2,d2,0
3  1,skill1,d1,0
4  3,skill3,d3,0
5  4,skill4,d4,0
6  5,skill5,d5,0
```

**main** Download

```
1  Large file hidden. You can download it using the button above.
```

**main.cpp** Download

```cpp
#include "SkillTree.hpp"

int main(){
    auto test = SkillTree("debug.csv");
    test.preorderDisplay();
    std::cout << "\n\n\n\n";
    test.balance();
    test.preorderDisplay();
}
```

**main.o** Download

Large file hidden. You can download it using the button above.

**skills.csv** Download

```csv
id,name,description,leveled
1,Terrifying Karaoke,Unleash a surprisingly awful rendition of a dragon's roar striking fear into enemies through sheer cringe. The embarrassment factor increases with each level.,0
2,Phantom Hug,Extend spectral tendrils for an unexpectedly warm and fuzzy hug immobilizing enemies and draining their will to fight. Hug intensity improves with each level.,0
3,Psychic Hotline,Offer psychic advice to foes leaving them so confused they question their life choices. The level of existential crisis induced increases with each level.,0
4,Mycotic Munchies,Consume MycoMorsels like a gourmet chef replenishing health and gaining a temporary boost in foodie satisfaction. Culinary delight improves with each level.,0
5,Flame Burp,Unleash a powerful breath attack with an unexpectedly comical fiery burp leaving enemies both scorched and amused. Burp radius and hilarity increase with each level.,0
6,Zombiewalk Dance,Lock eyes with a target and break into a mesmerizingly awkward dance reducing their combat skills through sheer embarrassment. Dance moves improve with each level.,0
7,Psychic Bubble Wrap,Create a protective bubble wrap barrier that pops loudly on impact deflecting attacks while providing unexpected auditory entertainment. Bubble wrap durability and popping volume improve with each level.,0
8,Fungi Fiasco,Release a burst of MycoMorsel spores that induce uncontrollable laughter in enemies. The comedic effect and laughter duration increase with each level.,0
9,Flight of the Absurd,Summon ethereal wings with unexpected cartoonish sound effects granting increased mobility and evasion. The absurdity of the wings improves with each level.,0
10,Ghostly Prank Phase,Enter a ghostly phase playing mischievous pranks on enemies like swapping their weapons or tying their shoelaces together. Prank duration and creativity improve with each level.,0
11,Reality Bypass,Inflict a mind-warping experience on a target making them question the very fabric of reality through a series of absurd scenarios. The absurdity and confusion increase with each level.,0
12,Mycotic Improv,Channel the essence of MycoMorsels to enhance other abilities with spontaneous improv comedy routines. The laughter-inducing potency improves with each level.,0
```