

Computer Science 272 - Assignment 2

Hash Tables & Binary Trees

Alex Di Vito | Data Structures & Algorithms | June 24, 2017

# **Question 1**

*Unit tests for Question 1 begin on* ***line 54*** *of* ***tests/BinaryTree.cpp***

* **PreOrderNext(x):** located on **line 110** of **include/BinaryTree.hpp**



* **PostOrderNext(x):** starts on **line 137** of **include/BinaryTree.hpp**



* **InOrderNext(x):** starts on **line 166** of **include/BinaryTree.hpp**



## **Question 1 Reflection**

Discovering <http://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/> was instrumental in my ability to understand this problem correctly. I understand now that the order in which we visit nodes in a Binary Tree is easily controlled by the order of where we place our business logic while recursively traversing each node’s left and right subtree. Whether we would like to traverse these nodes preorder, inorder or postorder, we simply need to move the lines of code pertinent to our business logic to the correct position.

For example, the following snippet shows three methods that all provide the same basic functionality of printing out each node value in a Binary Tree as it is traversed, however, this basic functionality could be replaced with more complicated business logic quite easily.



# **Question 2**

*Unit tests for Question 2 begin on* ***line 103*** *of* ***tests/BinaryTree.cpp***

* **IsBinarySearchTree(root, left, right):** returns true if the Binary Tree satisfies the search property, otherwise returns false.
  + *Begins on* ***line 89*** *of* ***include/BinaryTree.hpp***

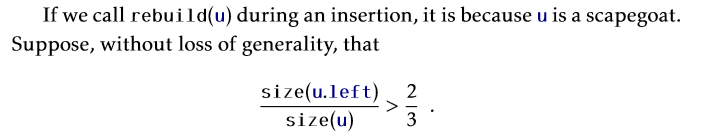
****

## **Question 2 Reflection**

Again, <http://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/> was an extremely helpful resource to which I based my implementation off of for this question. Essentially, this method traverses down both subtrees of each node and determines, if each left node contains an integer that is less than its parent and if each right node contains an integer that is greater than its parent. If any node does not satisfy these conditions, then the entire Binary Tree is not considered searchable.

# **Question 3**

Given a **Scapegoat Binary Tree**, as elements are added to the tree, we can ensure that the tree remains balanced by storing credits on each node as we traverse down the tree in order to add the new node. These credits are considered to be numeric values that we can use to ensure that the tree remains balanced and provides the minimum total tree height possible.

*Lemma 8.3 states:*

Using this formula, we can determine if any node **u** is a **scapegoat** by comparing the **left** or **right** **child** subtrees**.** As we traverse down the tree, each node **u** is compared with its children and deemed a **scapegoat** if any children contain more than 2/3 of the entire size of the tree/subtree. When this is detected, we rebuild the tree beginning with the scapegoat node to provide a more balanced Binary Tree.

When the elements **1, 5, 2, 4, 3** are added to a **ScapeGoat,** the following occurs. The first three nodes: **1, 5,** **2** are added successfully, but when the fourth nodeis added, the root node becomes a **scapegoat** because its right tree contains more than **2/3** of the total space in the tree. The tree is rebuilt using an **inorder** traversal and **5** becomes the new root. Finally, the last node **3** can be added without needing a call to the **rebuild** method as none of the nodes children exceed 2/3 size of the total tree.

¾ > 2/3

1

5

2

4

3

1

5

4

2

2/3 !> 2/3

½ !> 2/3

½ !> 2/3

2/3 !> 2/3

3/5 !> 2/3

## **Question 3 Reflection**

I found this video extremely useful for this question: <https://www.youtube.com/watch?v=La-xYKoKwjY&t=1s>. I understand Scapegoat Binary Trees do not maintain a perfectly balanced Binary Tree, but are deterministic to maintain a Binary Tree that is balanced such that each node’s children contain more than two thirds of the total subtree.

# **Question 4**

*Unit tests for Question 4 begin on* ***line 50*** *of* ***tests/HashTable.cpp***

* **Add(x):** If x already exists in the hash table, return false, otherwise add the new element and resize the hash table if necessary.

****

* **Find(x)**: If x is not found, return -1, otherwise return the hashKey for the found x.



* **Remove(x):** If x is not found return false, otherwise remove the element and resize the table if necessary and return true.



* **Hash(x):** Hash x using k mod 13 hash function.



## **Question 4 Reflection**

I was able to utilize Pat Morin’s textbook implementation relatively easily for my Hash Table. At first, one thing that kind of confused me about hash tables was if and how they would be superior to using the **Map** data structure, or the **Tuple** from python or any other **key-value** store data type since **Hash Tables** have to deal with collisions. I came to the conclusion that perhaps why hash tables are great data structures is because they can be easily used as buffers in the sense that they allow for an infinite number of keys, which can be ordered in some type of sorted manner while also allowing you to easily resize the **Hash Table** as needed.

# **Question 5**

*Unit tests for Question 5 begin on* ***line 14*** *of* ***tests/SubBinaryTree.cpp***

* **PreOrderNumber(x):** Recursively travels the Binary Tree and assigns the appropriate **preorderX** values to each node that it encounters.

****

* **PostOrderNumber(x)**: Recursively travels the Binary Tree and assigns the appropriate **postorderX** values to each node that it encounters.



* **InOrderNumber(x)**: Recursively travels the Binary Tree and assigns the appropriate **inorderX** values to each node that it encounters.

// inOrderNumber

void inOrderNumber(BTNode\* x, bool IsLeftTree = true) {

if (x == NULL) return;

inOrderNumber(x->left, IsLeftTree);

if (IsLeftTree) {

// if parent is smaller than x, swap

if (x->parent != NULL && x->parent->postorderX < x->postorderX) {

int tmp1 = x->parent->postorderX;

x->parent->postorderX = x->postorderX;

x->postorderX = tmp1;

}

// if grandparent is smaller than x, swap

if (x->parent != NULL && x->parent->parent != NULL && x->parent->parent->postorderX < x->postorderX) {

int tmp2 = x->parent->parent->postorderX;

x->parent->parent->postorderX = x->postorderX;

x->postorderX = tmp2;

}

} else {

// if parent is larger than x, swap

if (x->parent != NULL && x->parent->postorderX < x->postorderX) {

int tmp1 = x->parent->postorderX;

x->parent->postorderX = x->postorderX;

x->postorderX = tmp1;

}

// if grandparent is larger than x, swap

if (x->parent != NULL && x->parent->parent != NULL && x->parent->parent->postorderX < x->postorderX) {

int tmp2 = x->parent->parent->postorderX;

x->parent->parent->postorderX = x->postorderX;

x->postorderX = tmp2;

}

}

// if parent->right is less than x, swap

if (x->parent != NULL && x->parent->right != NULL && x->parent->right->postorderX < x->postorderX) {

int tmp4 = x->parent->right->postorderX;

x->parent->right->postorderX = x->postorderX;

x->postorderX = tmp4;

}

// if parent->left is greater than x, swap

if (x->parent != NULL && x->parent->left != NULL && x->parent->left->postorderX > x->postorderX) {

int tmp3 = x->parent->left->postorderX;

x->parent->left->postorderX = x->postorderX;

x->postorderX = tmp3;

}

inOrderNumber(x->right, false);

}

## **Question 5 Reflection**

I definitely had the most trouble with this problem. First, I was not able to correctly extend my original Binary Tree class. Unfortunately, C++ is not my main programming language, and after much effort and even asking a question on stack overflow, and getting stuck again. I decided to just copy and paste the original **BinaryTree.hpp** and adapted from there. In order to ensure that my algorithms produced the correct output, I wrote unit tests beginning on **line 31** of **SubBinaryTree.cpp** which prints out the before and after states in the console. I was able to successfully get the correct output for **PreOrderNumber()** and **PostOrderNumber()** but struggled to get the exact output desired for **InOrderNumber()**.