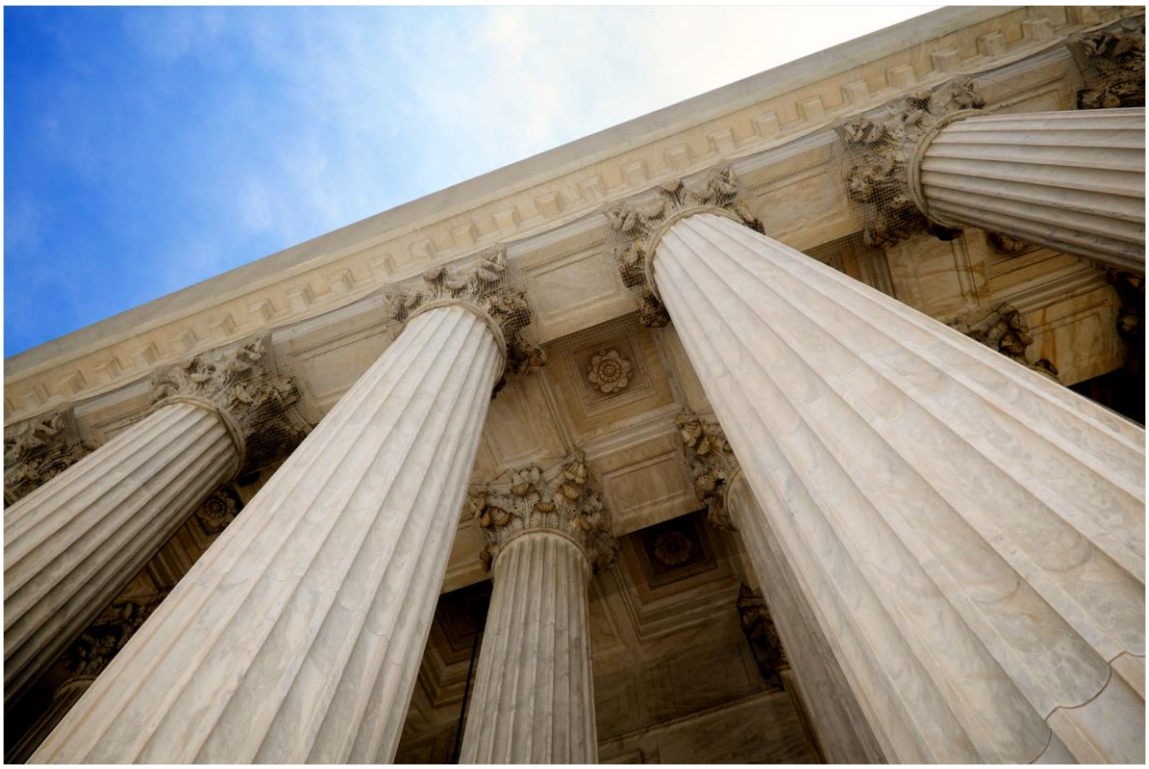


Computer Science 272 - Assignment 2

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**

```
    // preorderNext(x): return the node visited after node x in a pre-order
traversal of BT
    BTNode* preorderNext(BTNode* x) {
        if (x == NULL) return NULL; // return NULL if given a NULL ptr

        // create a vector to store the nodes that we visit
        vector<BTNode*> nodesVisited = {};

        // perform a basic preorder traversal to track each node we visit
        preorderNextInner(x, nodesVisited);

        // returning the node visited after x (index 1 since x is always the
first visited node with pre order)
        if (nodesVisited[1] != NULL) return nodesVisited[1];
        else return NULL;
    };

    void preorderNextInner(BTNode* x, vector<BTNode*>& nodesVisited) {
        if (x == NULL) return;
        // deal with x here for preorder traversal
        // we will use a vector to store nodes as we visit them
        nodesVisited.push_back(x);

        // recur through left subtree
        preorderNextInner(x->left, nodesVisited);
        // recur through right subtree
        preorderNextInner(x->right, nodesVisited);
    };
```

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

```
    // postorderNext(x): return the node visited after node x in a post-order
traversal of BT
    BTNode* postorderNext(BTNode* x) {
        if (x == NULL) return NULL; // return NULL if given a NULL ptr

        // create a vector to store the nodes that we visit
        vector<BTNode*> nodesVisited = {};

        // perform a basic postorder traversal to track each node we visit
        postorderNextInner(x, nodesVisited);

        // in a post-traversal, node X would be the last node visited, so there
would be no "node visited after node x"?
        // returning NULL instead
        return NULL;
    };

    void postorderNextInner(BTNode* x, vector<BTNode*>& nodesVisited) {
        if (x == NULL) return;

        // recur through left subtree
        postorderNextInner(x->left, nodesVisited);

        // recur through right subtree
        postorderNextInner(x->right, nodesVisited);

        // deal with x here for postorder traversal
        // we will use a vector to store nodes as we visit them
        nodesVisited.push_back(x);
    };
```

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

```
    // inorderNext(x): return the node visited after node x in a in-order
traversal of BT
    BTNode* inorderNext(BTNode* x) {
        if (x == NULL) return NULL; // return NULL if given a NULL ptr

        // create a vector to store the nodes that we visit
        vector<BTNode*> nodesVisited = {};
        // i'll use an additional variable here that will be used to keep track
of node x's index in our nodeVisited vector once it is reached
        unsigned int nodeXPos;
        // perform a basic inorder traversal to track each node we visit
        inorderNextInner(x, x, nodesVisited, nodeXPos);

        // we have stored node x's index in our vector when it was visited as
nodeXPos
        // return the next node visited after
        return nodesVisited[nodeXPos+1];
    };

    void inorderNextInner(BTNode* x, BTNode* sudoRoot, vector<BTNode*>&
nodesVisited, unsigned int& nodeXPos) {
        if (x == NULL) return;

        // recur through left subtree
        inorderNextInner(x->left, sudoRoot, nodesVisited, nodeXPos);

        // deal with x here for inorder traversal
        // we will use a vector to store nodes as we visit them
        nodesVisited.push_back(x);

        // if x == sudoRoot, then we have visited our initial node, store it's
index
        if (x == sudoRoot) {
            nodeXPos = nodesVisited.size() - 1;
        }

        // recur through right subtree
        inorderNextInner(x->right, sudoRoot, nodesVisited, nodeXPos);
    };
```

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.

```
// print postorder
void printPostorder(BTNode* u) {
    if (u == NULL) return;

    // first recur on left subtree
    printPostorder(u->left);

    // then recur on right subtree
    printPostorder(u->right);

    // now deal with the node
    cout << " -> " << u->x;
}

// print postorder
void printInorder(BTNode* u) {
    if (u == NULL) return;

    // first recur on left subtree
    printInorder(u->left);

    // now deal with the node
    cout << " -> " << u->x;

    // then recur on right subtree
    printInorder(u->right);
}

// print preorder
void printPreorder(BTNode* u) {
    if (u == NULL) return;

    // now deal with the node
    cout << " -> " << u->x;

    // first recur on left subtree
    printPreorder(u->left);

    // then recur on right subtree
    printPreorder(u->right);
}
```

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***

```
    // IsBinarySearchTree: tests whether a binary tree satisifies the search tree
order property at every node
    // http://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-
or-not/
    bool isBinarySearchTree(struct BTNode* root, struct BTNode* l=NULL, struct
BTNode* r=NULL)
    {
        // Base condition
        if (root == NULL)
            return true;

        // if left node exist that check it has
        // correct data or not
        if (l != NULL && root->x < l->x)
            return false;

        // if right node exist that check it has
        // correct data or not
        if (r != NULL && root->x > r->x)
            return false;

        // check recursively for every node.
        return isBinarySearchTree(root->left, l, root) &&
            isBinarySearchTree(root->right, root, r);
    }
```

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.

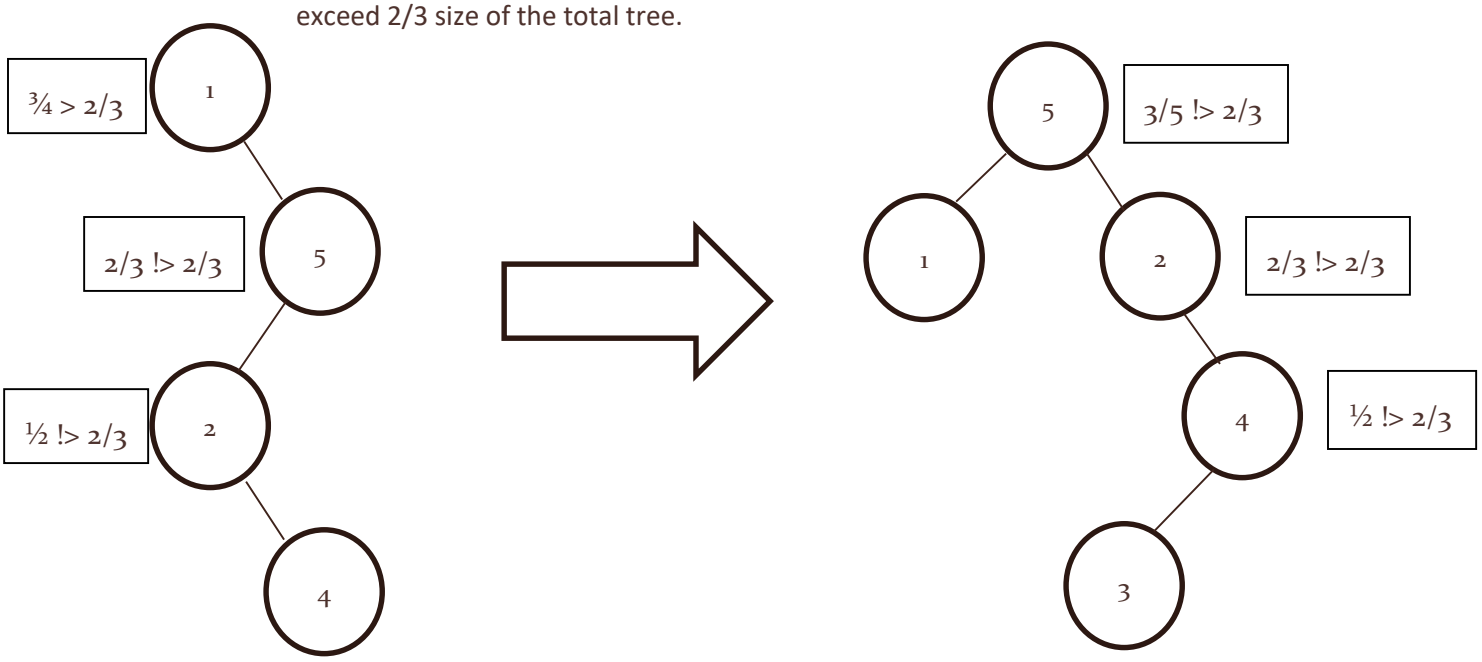
Lemma 8.3 states:

If we call rebuild(u) during an insertion, it is because u is a scapegoat. Suppose, without loss of generality, that

$$\frac{\texttt{size(u.left)}}{\texttt{size(u)}} > \frac{2}{3} .$$

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 node is 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.



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.

```
    // if (int x) does not exist in the hashtable, add it
    bool add(int x) {
        int index = find(x);
        if (index != -1) return false;

        // resize table
        if (2 * (this->numNonNull + 1) > this->tableSize) resize();

        index = hash(x);
        // start from index searching linearly for a place to store the new x
        while (this->table[index] != this->nullVal && this->table[index] != this-
>delVal) {
            // increment index, because we are ensuring occpancy levels,
            // we can count on eventually running into a null entry
            index = (index == this->tableSize -1) ? 0 : index + 1; // increment
index
        }

        // keep track of non null entries
        if (this->table[index] == this->nullVal) {
            this->numNonNull++;
        }

        // increment num elems and set x
        this->numElems++;
        this->table[index] = x;

        return true;
    }
```

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

```
    // returns an index (or hash) if it exists in the hashtable, otherwise
returns -1
    int find(int x) {
        int index = hash(x);

        while (this->table[index] != this->nullVal) {
            // return the index if it is the value we are searching for
            if (this->table[index] != this->delVal && x == this->table[index])
return index;

            // increment index, because we are ensuring occpancy levels,
            // we can count on eventually running into a null entry
            index = (index == this->tableSize - 1) ? 0 : index + 1; // increment
index
        }

        return -1;
    }
```

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

```
    // if (int x) exists in the hashtable remove it
    bool remove(int x) {
        int index = find(x); // search for an index
        if (index == -1) return false; // return false if it does not exist

        // decrement numElems
        this->numElems--;
        this->table[index] = this->delVal; // delete x

        return true;
    }
```

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

```
// k mod 13 hash function
int hash(int k) {
    return k % 13;
}
```

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.

```
    // preOrderNumber
    void preOrderNumber(BTNode* x) {
        if (x == NULL) return;

        preOrderNumber(x->left);

        // perform a basic preorder traversal to track each node we visit
        preOrderNumber(x->right);

        // if parent is greater than x, swap
        if (x->parent != NULL && x->parent->preorderX > x->preorderX) {
            int tmp1 = x->parent->preorderX;
            x->parent->preorderX = x->preorderX;
            x->preorderX = tmp1;
        }

        // if grandparent is greater than x, swap
        if (x->parent != NULL && x->parent->parent != NULL && x->parent->parent-
>preorderX > x->preorderX) {
            int tmp2 = x->parent->parent->preorderX;
            x->parent->parent->preorderX = x->preorderX;
            x->preorderX = tmp2;
        }

        // if parent->left is greater than x, swap
        if (x->parent != NULL && x->parent->left != NULL && x->parent->left-
>preorderX > x->preorderX) {
            int tmp3 = x->parent->left->preorderX;
            x->parent->left->preorderX = x->preorderX;
            x->preorderX = tmp3;
        }

        // if parent->right is less than x, swap
        if (x->parent != NULL && x->parent->right != NULL && x->parent->right-
>preorderX < x->preorderX) {
            int tmp4 = x->parent->right->preorderX;
            x->parent->right->preorderX = x->preorderX;
            x->preorderX = tmp4;
        }
    };
```

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

```
    // postOrderNumber
    void postOrderNumber(BTNode* x) {
        if (x == NULL) return;

        postOrderNumber(x->left);

        // if left is greater than x, swap
        if (x->left != NULL && x->left->postorderX > x->postorderX) {
            int tmp3 = x->left->postorderX;
            x->left->postorderX = x->postorderX;
            x->postorderX = tmp3;
        }

        // if right is less than x, swap
        if (x->right != NULL && x->right->postorderX < x->postorderX) {
            int tmp4 = x->right->postorderX;
            x->right->postorderX = x->postorderX;
            x->postorderX = tmp4;
        }

        postOrderNumber(x->right);

        // if parent is less 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 less 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;
        }
    };
```

- **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()**.