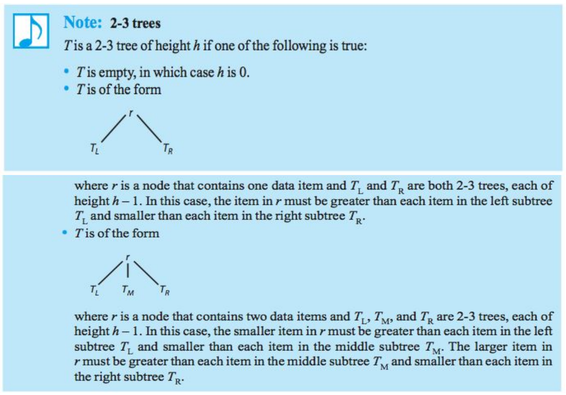
Brian Faure
RUID:150003563
NetID:bdf39
Prog. Methodology II
Project #2

My Decision: **-Project Type IA:** ***Investigating the height of trees***

---

1. Design and implement a 2-3 Balanced Search Tree. Use UML for your design. Please note you're not required to 'analyze' the requirements or the analyze the design. Your implementation must support the standard 'operational contract' as given in the interface for a 2-3 tree.

To fit the standard ‘operational contract’ the 2-3 Balanced Search Tree implementation **must** align with the following definitions provided in the “Data Abstraction and Problem Solving 6th Edition” on pages 569 & 570:


Note: **2-3 trees**

T is a 2-3 tree of height h if one of the following is true:

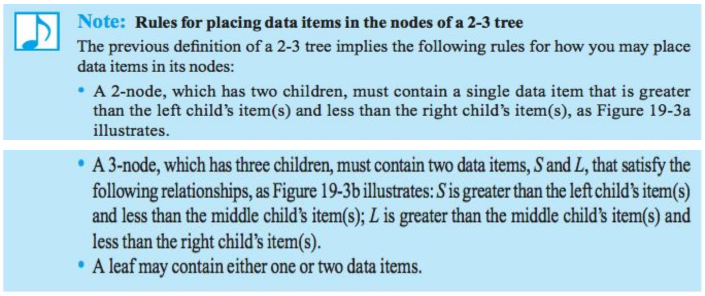
- T is empty, in which case h is 0.
- T is of the form

$$\begin{tikzcd} & r \arrow[dl, dash] \arrow[dr, dash] & \\ T_L & & T_R \end{tikzcd}$$

where r is a node that contains one data item and T_L and T_R are both 2-3 trees, each of height $h - 1$. In this case, the item in r must be greater than each item in the left subtree T_L and smaller than each item in the right subtree T_R.
- T is of the form

$$\begin{tikzcd} & r \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ T_L & T_M & T_R \end{tikzcd}$$

where r is a node that contains two data items and T_L, T_M, and T_R are 2-3 trees, each of height $h - 1$. In this case, the smaller item in r must be greater than each item in the left subtree T_L and smaller than each item in the middle subtree T_M. The larger item in r must be greater than each item in the middle subtree T_M and smaller than each item in the right subtree T_R.

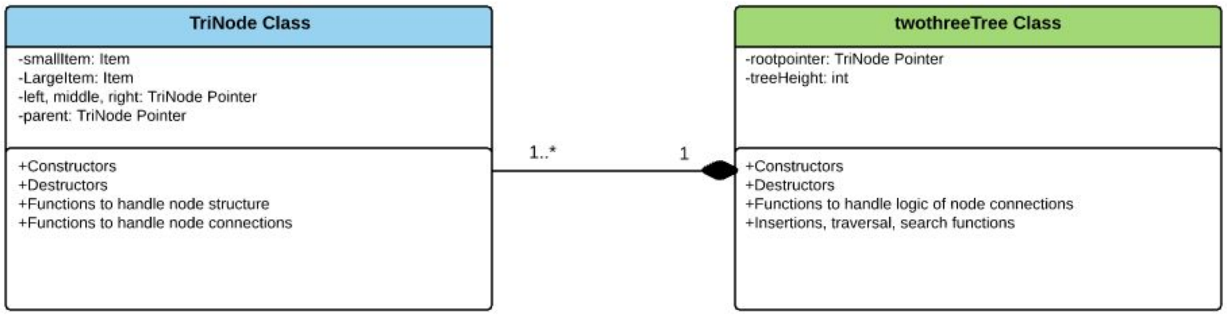

Note: **Rules for placing data items in the nodes of a 2-3 tree**

The previous definition of a 2-3 tree implies the following rules for how you may place data items in its nodes:

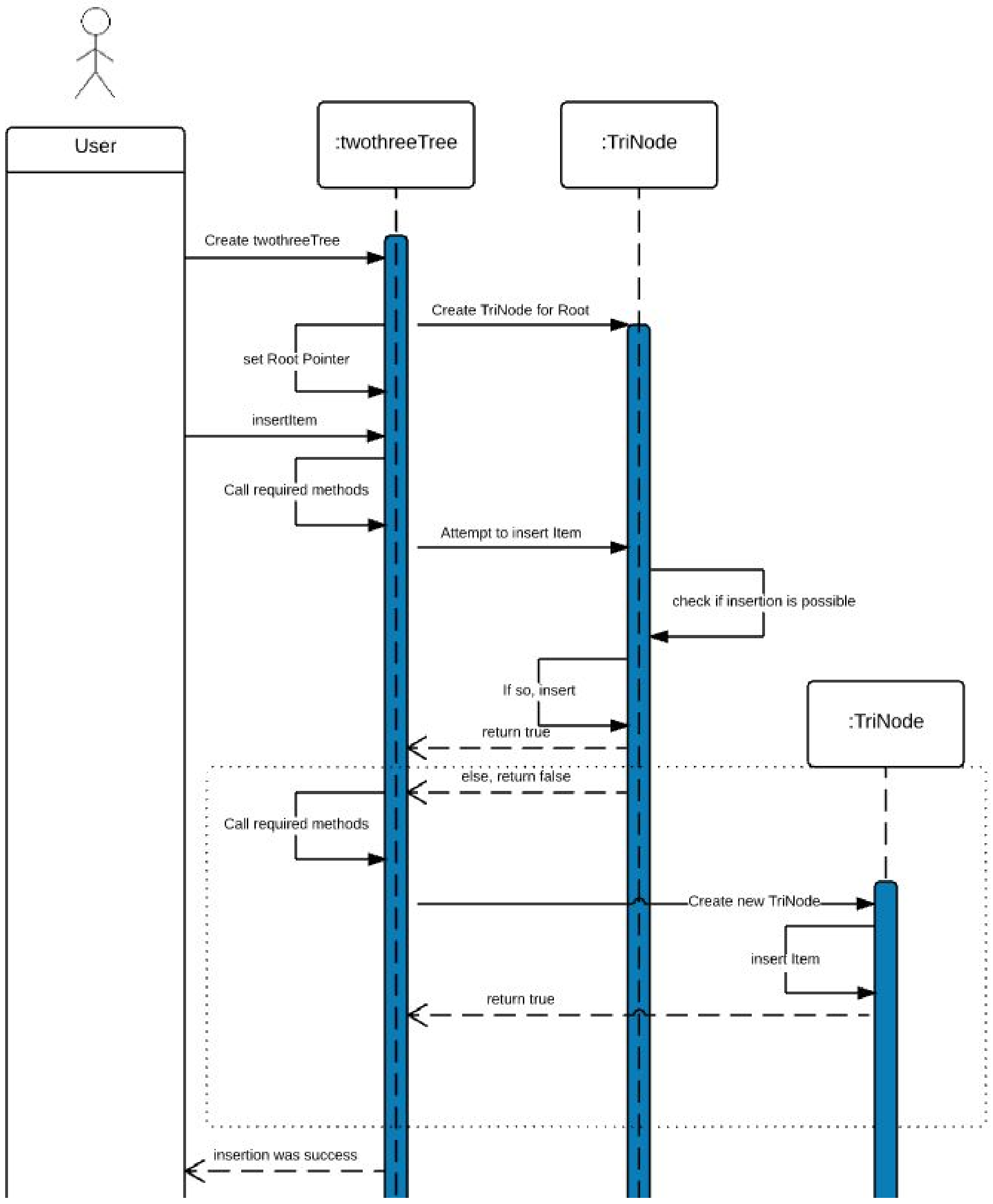
- A 2-node, which has two children, must contain a single data item that is greater than the left child’s item(s) and less than the right child’s item(s), as Figure 19-3a illustrates.
- A 3-node, which has three children, must contain two data items, S and L, that satisfy the following relationships, as Figure 19-3b illustrates: S is greater than the left child’s item(s) and less than the middle child’s item(s); L is greater than the middle child’s item(s) and less than the right child’s item(s).
- A leaf may contain either one or two data items.

To properly outline my thought process going into this project I have created several UML diagrams which include the interactions among the classes and functions present in the situation.

→ Basic Class Interaction:


TriNode Class
-smallItem: Item
-LargeItem: Item
-left, middle, right: TriNode Pointer
-parent: TriNode Pointer
+Constructors
+Destructors
+Functions to handle node structure
+Functions to handle node connections
1..*
1
twothreeTree Class
-rootpointer: TriNode Pointer
-treeHeight: int
+Constructors
+Destructors
+Functions to handle logic of node connections
+Insertions, traversal, search functions

→ Simplified twothreeTree creation and subsequent insertion:



Using the ideas I have developed in the process of creating these diagrams, along with the applicable information from the textbook, I will begin the creation of the 2-3 Tree in c++.

→ I have outlined several steps in my creation of the 2-3 Tree below.

Rev_1: Goal = Create all basic portions of 2-3 Tree code
See 23Tree_Rev1.cpp under Problem1 Folder...
-At this point I have created the following:
→ **TriNode<ItemType> class to hold the nodes present in the 2-3 Tree**
→ Defined Class member functions for TriNode including:
-TriNode constructor with a single input (create new TriNode with no

children and 1 item)
-TriNode constructor with input value and pointers to 3 children
-isLeaf() function to check if TriNode has no children
-isTwoNode() function to check if TriNode has 2 children
-isThreeNode() function to check if TriNode has 3 children
-getSmallItem() function to return small item held in TriNode
-getLargeItem() function to return larger item held in TriNode
-setSmallItem(input) to set the small item held in TriNode to input
-setLargeItem(input) to set the large item held in TriNode to input
-getLeftChildPtr() to return a pointer to the left child of the TriNode
-getMidChildPtr() to return a pointer to the mid child of the TriNode
-getRightChildPtr() to return a pointer to the right child of the TriNode
-setLeftChildPtr() to set new left child pointer for TriNode
-setMidChildPtr() to set new middle child pointer for TriNode
-setRightChildPtr() to set new right child pointer for TriNode

→ **twothreeTree<ItemType> class to manage the TriNode class**

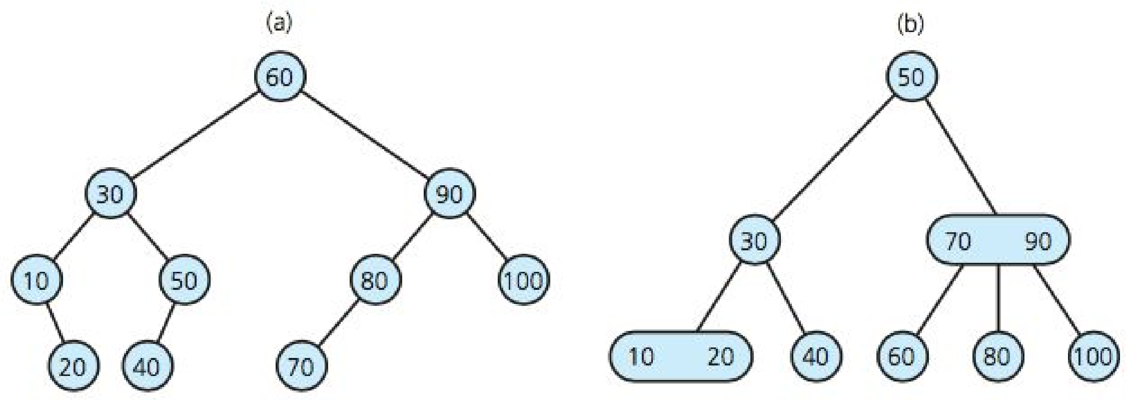
→ Defined Class member functions for twothreeTree including:
-twothreeTree constructor with input pointer to set rootPtr to input pointer
-setRootPtr(input) function to set rootPtr to new input

→ **Defined function called outputNode(TriNode<ItemType> temp, string) to output the contents of a TriNode given its input string, for example:**
-If the input string is "ALL", outputNode will attempt to output both the left and right items in the input TriNode
-If the input string is "RIGHT", outputNode will attempt to output the left item held in the TriNode
-If the input string is "LEFT", outputNode will attempt to output the right item held in the TriNode

→ **Defined function called findItem(twothreeTree<ItemType> temp, ItemType target) which is search implementation of the 2-3 Tree**
-findItem() will return true if it is able to find the target in the tree
-Otherwise it will return false

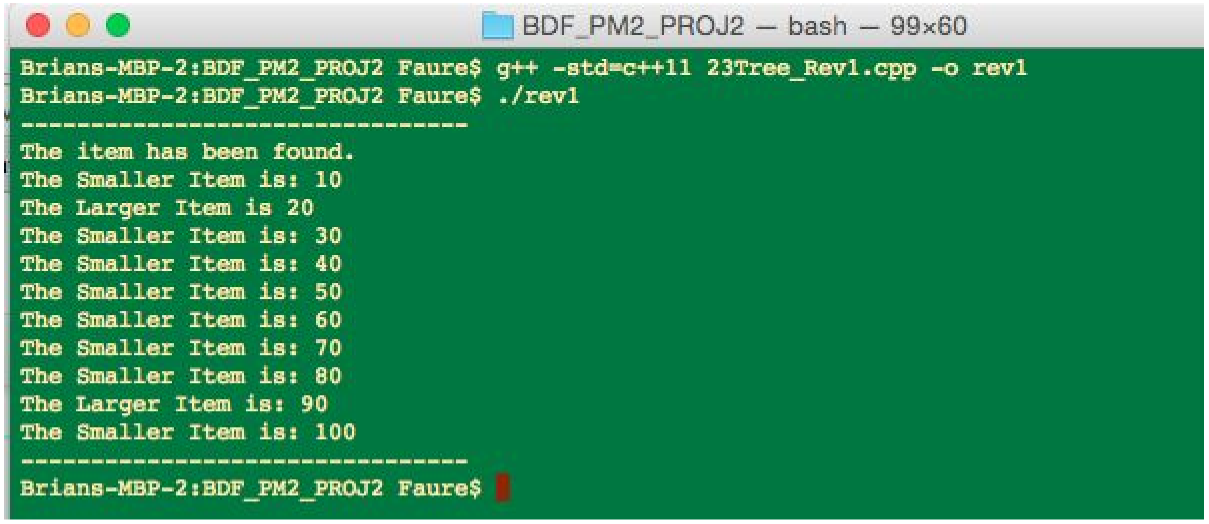
→ **Defined function called inorder(twothreeTree<ItemType> inputTree) which is the traversal implementation of the 2-3 Tree**
-Will output the contents of the inputTree in ascending sorted order as long as the 2-3 Tree is a correct 2-3 Tree

FIGURE 19-5 (a) A balanced binary search tree; (b) a 2-3 tree with the same entries



-Using 23Tree_Rev1.cpp I have created the 2-3 Tree in part (b) of the above figure to verify that my implementation is working correctly. See the main function of 23Tree_Rev1.cpp to check how I inserted values without having an insert function but the summary would be that it's definitely time-consuming.

-In the main function I have also called findItem to see if the value of 50 is present in the tree as well as calling an inorder traversal of the tree. The following is the terminal output…


BDF_PM2_PROJ2 — bash — 99×60

```
Brians-MBP-2:BDF_PM2_PROJ2 Faure$ g++ -std=c++11 23Tree_Rev1.cpp -o rev1
Brians-MBP-2:BDF_PM2_PROJ2 Faure$ ./rev1
--------------------------------
The item has been found.
The Smaller Item is: 10
The Larger Item is 20
The Smaller Item is: 30
The Smaller Item is: 40
The Smaller Item is: 50
The Smaller Item is: 60
The Smaller Item is: 70
The Smaller Item is: 80
The Larger Item is: 90
The Smaller Item is: 100
--------------------------------
Brians-MBP-2:BDF_PM2_PROJ2 Faure$ 
```

-As you can see, the findItem function worked perfectly
-As of now, the inorder traversal only outputs to the terminal and it did so correctly as the values it printed ARE in ascending order.
-So far my implementation is working correctly, the next function I will need to add is the insertItem function.

Rev_2: Goal = Create the insertItem function
-See 23Tree_Rev2.cpp under Problem1 Folder...

First Edit: Added public bool data item to TriNode<ItemType> called isLargeItem which is set to FALSE until a value is assigned to the largeItem private data member. This is used in several of the functions to assess whether or not the TriNode has 1 or 2 data items. The previous way it worked was that largeItem would be set to 0 until a value was loaded in but this restricted the 2-3 Tree to integer values. Everything else was already set up to allow for templates so I made this change to allow for all datatypes in the 2-3 Tree.

Second Edit: Add parent pointer private data item to TriNode class along with functions to get the parent pointer as well as set it. Predict that these will be needed in the insertion and removal functions.

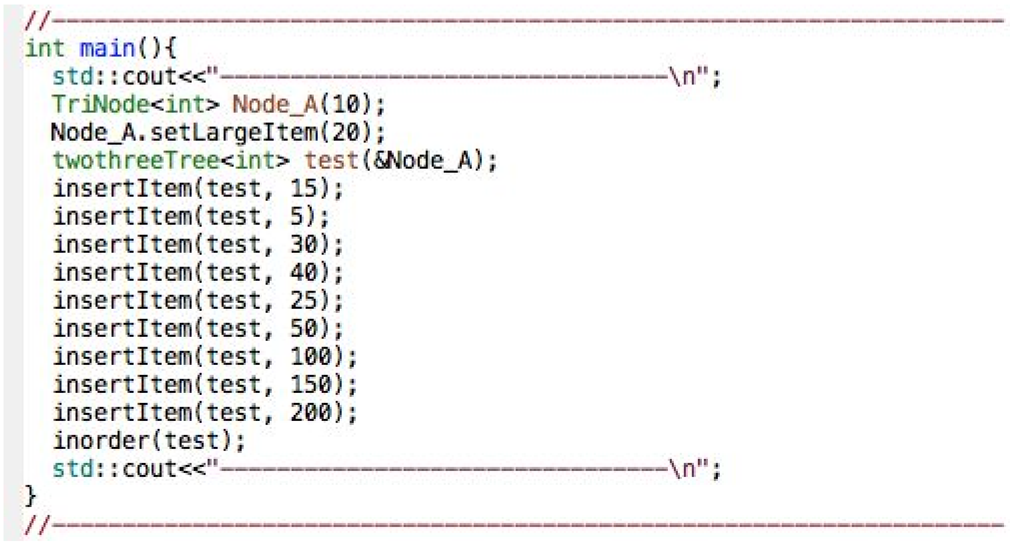
Third Edit: Added findLocation public member function to twothreeNode class. Different from the findItem function in that it takes in a TriNode pointer input to the currentNode it is inspecting and includes a third TriNode pointer input it carries through the operation. This third input is initialized before the function is called and at the end it will contain the location where a new item should be placed in the 2-3 Tree. In other words, this function is called in the insertItem function to figure out where to initially place the new item.

Fourth Edit: Add hasLargeItem() public member function to TriNode class to allow functions to know if the value stored in a TriNode largeItem spot is real (has been set) or has not yet been defined.

Fifth Edit: Added an extra data item to TriNode class called extraItem and functions to get and set this value named getExtraItem and setExtraItem as well as a hasExtraItem function that is similar to the hasLargeItem function. The addition of another data item allows the TriNode class to temporarily emulate a node with 3 data items that you might see in a 2-3-4 Tree. While building the insertItem function I quickly realized the requirement for this type of class addition because there needs to be a way for the node to hold another item occasionally when in the rebuilding phase of the tree. Along the same lines I have added an extra child pointer and helper functions for that data item because its existence is also required when using my implementation of insertItem (more specifically split, explained later).

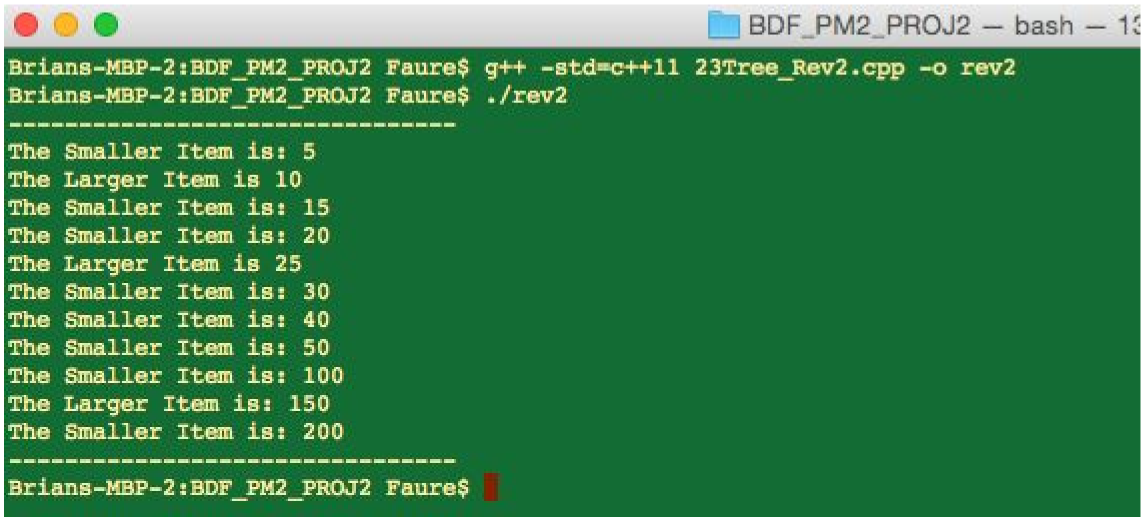
Sixth Edit: Finished the insertItem and split functions. insertItem takes an ItemType (probably int) input as well as a reference to a twothreeTree<ItemType>. The ItemType, newItem, input is used as the new value to be inserted and, when complete, the function sets the twothreeTree reference, inputTree, equal to the new tree it has created in the insertion process. The first function that insertItem calls is findLocation (described in Third Edit). In summary, findLocation adjusts a reference to a TriNode pointer such that after it is called, that TriNode pointer points to the location in the 2-3 Tree where the new value should be placed. After It has called findLocation, insertItem enters an if statement that decides whether or not it will need to split any nodes in the process of the insertion. If not, it uses the location provided by the findLocation function and just inserts the value. If so, insertItem calls the split function. split takes several inputs, the first is the pointer to the insertion location found in findLocation, this variable is named n within the split function. The second input to split is a reference to a twothreeTree, this variable is named obj within the split function. The third input to split is a reference to a newly created TriNode pointer, this variable is named tempNode within the split function. The last input is a reference to a newly created boolean, this variable is named condition within the split function. The last input decides whether or not insertItem uses the reference to the twothreeTree as its effective “output” or the reference to the TriNode pointer as its output. The *normal* output is the reference to the twothreeTree but when the split function is required to create a brand new root node for the tree, it must pass the output as a pointer to a TriNode item which is used to create a new twothreeTree. I included this as a function of trial and error where I checked whether or not the insertItem program was receiving a valid output from split. Within the split function there are several layers of complexity but it follows almost the exact structure found in the textbook on page 579.

→ The following is a screenshot of the main function of 23Tree_Rev2.cpp:



```
//-----------------------------------------------------------------
int main(){
  std::cout<<"--------------------------------\n";
  TriNode<int> Node_A(10);
  Node_A.setLargeItem(20);
  twothreeTree<int> test(&Node_A);
  insertItem(test, 15);
  insertItem(test, 5);
  insertItem(test, 30);
  insertItem(test, 40);
  insertItem(test, 25);
  insertItem(test, 50);
  insertItem(test, 100);
  insertItem(test, 150);
  insertItem(test, 200);
  inorder(test);
  std::cout<<"--------------------------------\n";
}
//-----------------------------------------------------------------
```

→ Running the program creates the following terminal output:



```
Brians-MBP-2:BDF_PM2_PROJ2 Faure$ g++ -std=c++11 23Tree_Rev2.cpp -o rev2
Brians-MBP-2:BDF_PM2_PROJ2 Faure$ ./rev2
--------------------------------
The Smaller Item is: 5
The Larger Item is 10
The Smaller Item is: 15
The Smaller Item is: 20
The Larger Item is 25
The Smaller Item is: 30
The Smaller Item is: 40
The Smaller Item is: 50
The Smaller Item is: 100
The Larger Item is: 150
The Smaller Item is: 200
--------------------------------
Brians-MBP-2:BDF_PM2_PROJ2 Faure$ 
```

As can be seen by the terminal output, the insertItem function works just as it should because the contents when read in-order are sorted ascendingly.

While debugging the code, I have come across the need to add various helper functions and data members to both the TriNode and twothreeTree classes. The first of which was a complete overhaul of the insertItem function. At first (up until the final rev3) I had implemented an insertItem function which was not a member of either the TriNode *or* twothreeTree classes. This was working great until I got to Problem 3 which weighed heavily on the performance of the insertItem function. Now the insertItem function has updated logic and is a public member of the twothreeTree class.

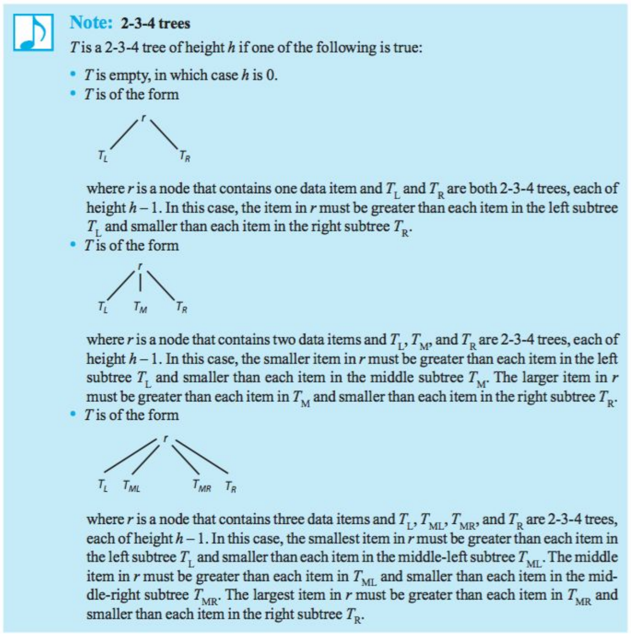
→ The final version of the implementation is saved as 23Tree_Final.cpp under the Problem1 folder.
→ The following table summarizes the functions available to the user. (Assume tempTree is an instance of twothreeTree)

User Available Functions	Source Class	Description
twothreeTree<ItemType> tempTree	twothreeTree	Default constructor, sets tempTree rootPtr to new TriNode<ItemType>
itn tempTree.getNodeCount()	twothreeTree	Return the number of separate nodes in the tree
bool tempTree.insertItem(ItemType)	twothreeTree	Insert an item into the tree if not already there, return false if found
int tempTree.getHeight()	twothreeTree	Return distance from root to leaf
void removeItem(tempTree, target)	twothreeTree	Remove item from tree if found (works in most cases)
void inorder(tempTree)	twothreeTree	Traverse tree inorder and print to terminal
bool findItem(tempTree, target)	twothreeTree	Returns true if item in tree, false otherwise

→ Even though there are several other functions available to the user, calling them may disrupt the organization of the twothreeTree

2. Repeat the above for another balanced search tree implementation in the book (2-3-4 Tree, AVL, RB tree). You can choose another but please provide an explanation.

The second balanced search tree I have decided to create is the 2-3-4 Tree, as described in the textbook from page 586 to page 592. This data structure differs from the 2-3 Tree in that its nodes are allowed to contain a maximum of 3 items and point to at most 4 children. The following screenshots outline the functionality of the 2-3-4 Tree.



Note: 2-3-4 trees

T is a 2-3-4 tree of height *h* if one of the following is true:

- *T* is empty, in which case *h* is 0.
- *T* is of the form

r; T_L T_R

where *r* is a node that contains one data item and T_L and T_R are both 2-3-4 trees, each of height $h - 1$. In this case, the item in *r* must be greater than each item in the left subtree T_L and smaller than each item in the right subtree T_R.

- *T* is of the form

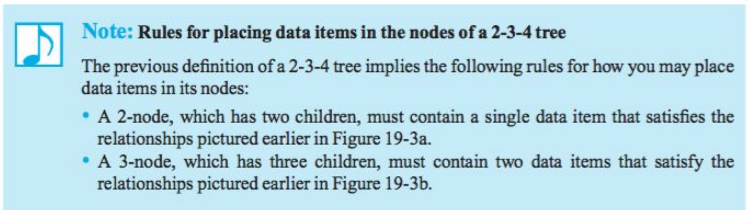
r; T_L T_M T_R

where *r* is a node that contains two data items and T_L, T_M, and T_R are 2-3-4 trees, each of height $h - 1$. In this case, the smaller item in *r* must be greater than each item in the left subtree T_L and smaller than each item in the middle subtree T_M. The larger item in *r* must be greater than each item in T_M and smaller than each item in the right subtree T_R.

- *T* is of the form

r; T_L T_{ML} T_{MR} T_R

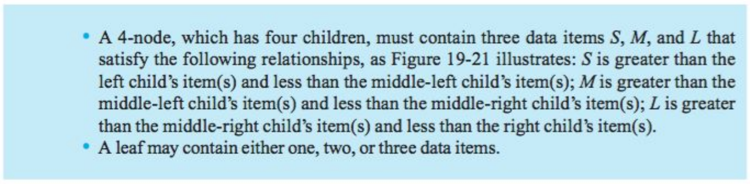
where *r* is a node that contains three data items and T_L, T_{ML}, T_{MR}, and T_R are 2-3-4 trees, each of height $h - 1$. In this case, the smallest item in *r* must be greater than each item in the left subtree T_L and smaller than each item in the middle-left subtree T_{ML}. The middle item in *r* must be greater than each item in T_{ML} and smaller than each item in the middle-right subtree T_{MR}. The largest item in *r* must be greater than each item in T_{MR} and smaller than each item in the right subtree T_R.



Note: Rules for placing data items in the nodes of a 2-3-4 tree

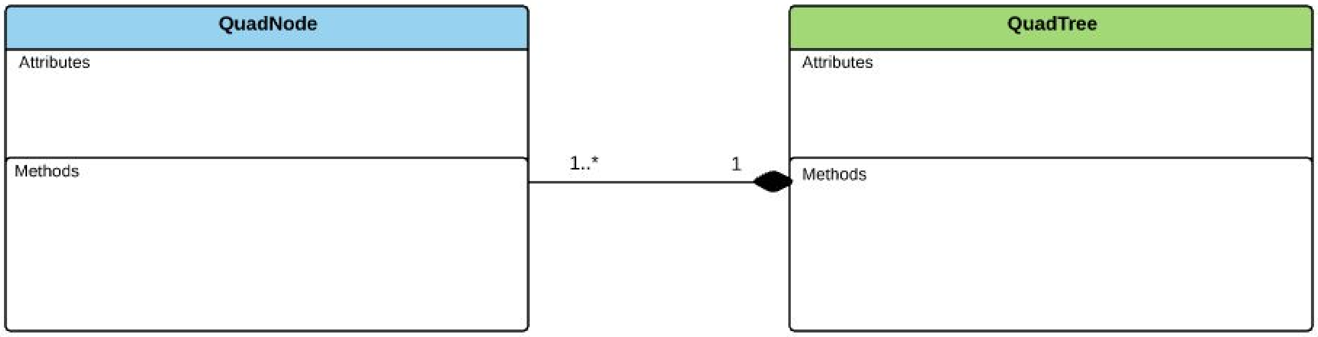
The previous definition of a 2-3-4 tree implies the following rules for how you may place data items in its nodes:

- A 2-node, which has two children, must contain a single data item that satisfies the relationships pictured earlier in Figure 19-3a.
- A 3-node, which has three children, must contain two data items that satisfy the relationships pictured earlier in Figure 19-3b.



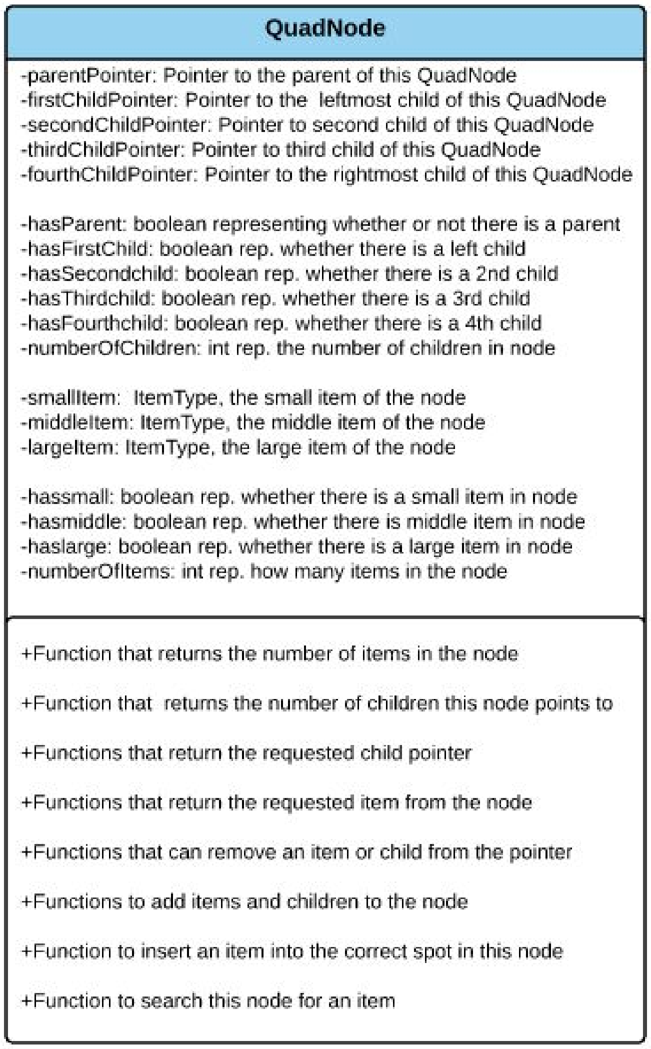
- A 4-node, which has four children, must contain three data items *S*, *M*, and *L* that satisfy the following relationships, as Figure 19-21 illustrates: *S* is greater than the left child's item(s) and less than the middle-left child's item(s); *M* is greater than the middle-left child's item(s) and less than the middle-right child's item(s); *L* is greater than the middle-right child's item(s) and less than the right child's item(s).
- A leaf may contain either one, two, or three data items.

Once again I have used UML diagrams to turn the theoretical functionality into a more physical representation of the interactions among various classes and functions. I will use a similar approach to the 2-3 Tree in that I plan on having 2 separate classes, one for the nodes and one for the tree. The diagram below illustrates how the two classes will be related.



An instance of QuadTree may be related to anywhere from 1 to infinity (theoretically) QuadNodes but its existence depends on these exact QuadNodes. Without the presence of the QuadNodes the QuadTree would be useless because it would not hold any actual data. In a similar respect, the QuadNode should not exist without the QuadTree (although it is entirely possible to organize the QuadNodes yourself, the point of the QuadTree is to keep the actual complexity veiled by its abstraction.

The two simplified class diagrams below present what I plan on incorporating within each class.


QuadNode

-parentPointer: Pointer to the parent of this QuadNode
-firstChildPointer: Pointer to the leftmost child of this QuadNode
-secondChildPointer: Pointer to second child of this QuadNode
-thirdChildPointer: Pointer to third child of this QuadNode
-fourthChildPointer: Pointer to the rightmost child of this QuadNode

-hasParent: boolean representing whether or not there is a parent
-hasFirstChild: boolean rep. whether there is a left child
-hasSecondchild: boolean rep. whether there is a 2nd child
-hasThirdchild: boolean rep. whether there is a 3rd child
-hasFourthchild: boolean rep. whether there is a 4th child
-numberOfChildren: int rep. the number of children in node

-smallItem: ItemType, the small item of the node
-middleItem: ItemType, the middle item of the node
-largeItem: ItemType, the large item of the node

-hassmall: boolean rep. whether there is a small item in node
-hasmiddle: boolean rep. whether there is middle item in node
-haslarge: boolean rep. whether there is a large item in node
-numberOfItems: int rep. how many items in the node

+Function that returns the number of items in the node

+Function that returns the number of children this node points to

+Functions that return the requested child pointer

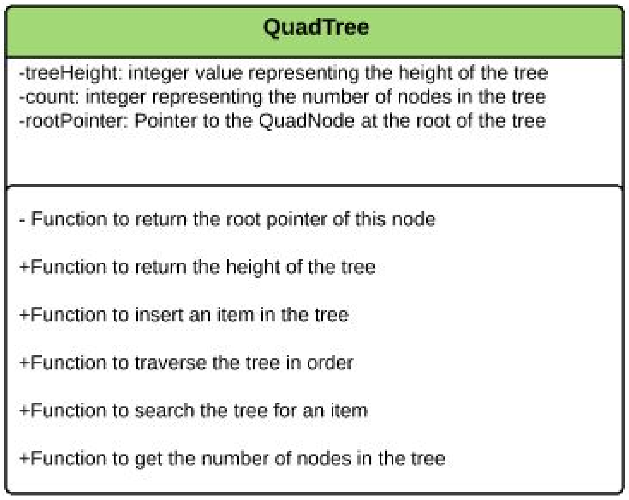
+Functions that return the requested item from the node

+Functions that can remove an item or child from the pointer

+Functions to add items and children to the node

+Function to insert an item into the correct spot in this node

+Function to search this node for an item


QuadTree

-treeHeight: integer value representing the height of the tree
-count: integer representing the number of nodes in the tree
-rootPointer: Pointer to the QuadNode at the root of the tree

- Function to return the root pointer of this node

+Function to return the height of the tree

+Function to insert an item in the tree

+Function to traverse the tree in order

+Function to search the tree for an item

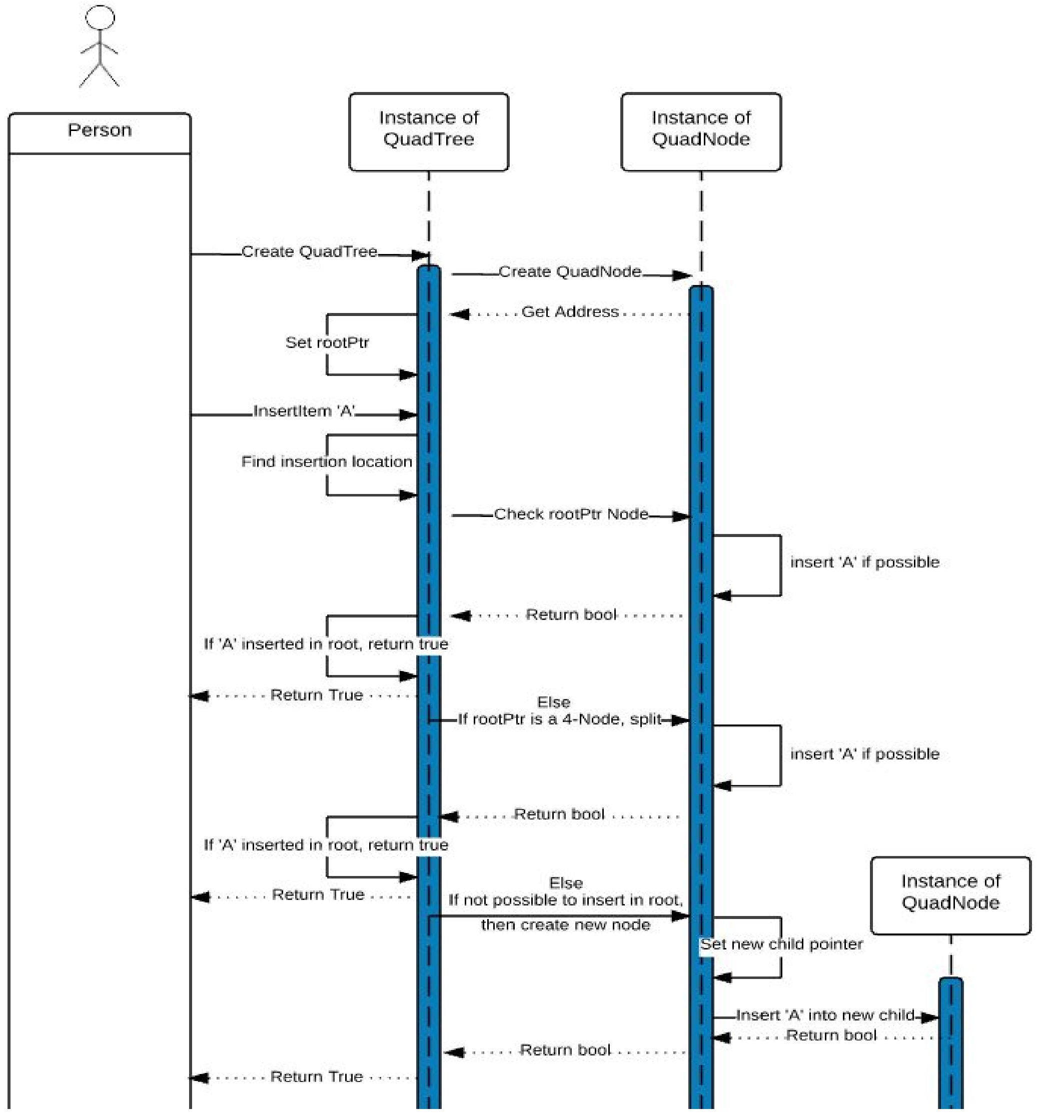
+Function to get the number of nodes in the tree

As can be seen from the sheer size of the QuadNode, almost all of the actual data is not held in the QuadTree but instead resides within the structure of the QuadNodes themselves. At first it may seem like this would slow down the functioning of the implementation of the tree because these big, heavy data containers are constantly being manipulated, but in my study of the problem I believe it to not be much of problem because the control of the QuadNodes is almost completely through the use of pointers.

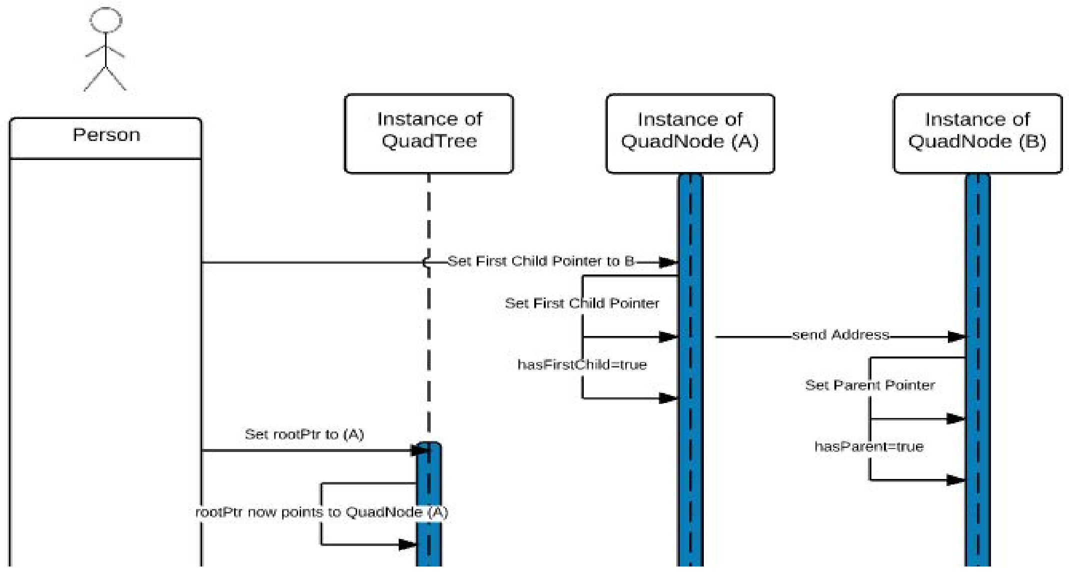
In the following diagrams I have outlined how I plan to implement some of the QuadTree functions. The insertItem function of the QuadTree Class will take in a single ItemType input. It will then call the findLocation function with the inputs being the newItem and the root pointer of the tree. The findLocation function will start at the root pointer, recursively checking if the current node is a four node. If it *is* a four node, the findLocation function will call a function in the QuadNode called splitFourNode which will turn the current four node into a combination of non-four nodes. On each step of the findLocation function, the program checks if the current node is a leaf, if so it attempts to insert the newItem, if this fails it means that the newItem is already in that node in which case the findLocation function returns false back to the insertItem function which then returns false to main. On the other hand, if the findLocation function is able to insert the newItem into the leaf, it returns true to insertItem which in turn returns true to main.

If the findLocation function calls splitFourNode on the root of the tree, splitFourNode will actually create a new QuadNode for the root and move the middle item from the old four-node into this new root. It will then split the remaining smallItem and largeItem among two new children. This is the process by which the 2-3-4 Tree grows in height and ensures that the overall tree remains balanced at all times.

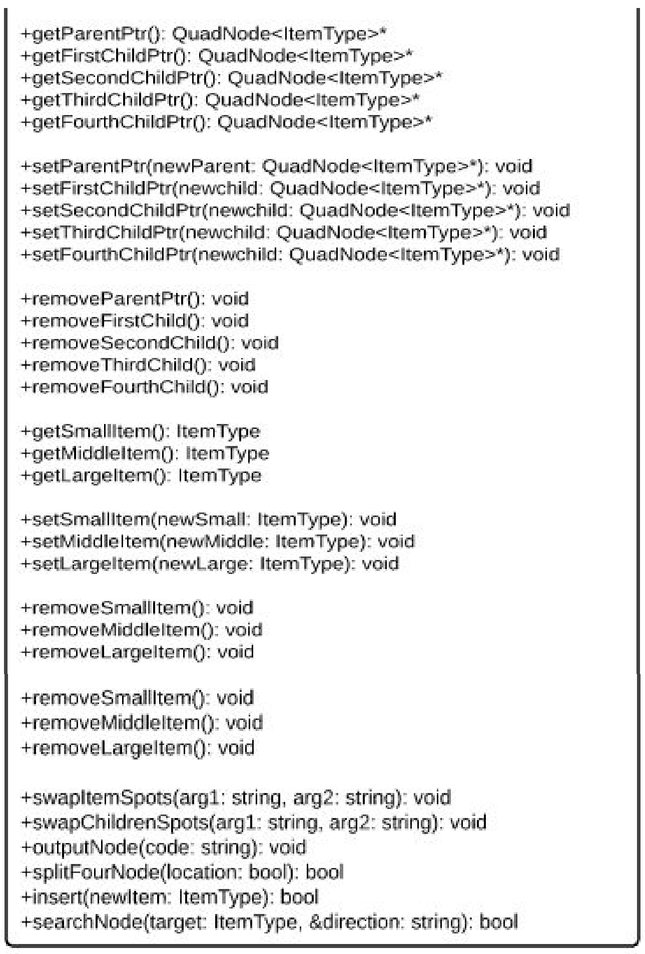
→ Example of construction of QuadTree and a consecutive insertItem call:



In the following example, I illustrate how the user can interact with QuadNodes themselves. In this case, the user creates two QuadNodes called “A” and “B”. The user then sets the firstChildPtr of A to point to B which automatically sets the parent pointer of “B” to point to “A”. This is not illustrative of how nodes are inserted into the tree because this would obviously cause the overall structure to be imbalanced but it shows how the nodes are linked with eachother. Similar to the 2-3 Tree, when a QuadNode sets its one of its child pointers, the child also must set its parent pointer to the initial node. In this example, the user then creates a QuadTree and sets its root pointer to node “A”, thus the new QuadTree is imbalanced but has a height of 2.

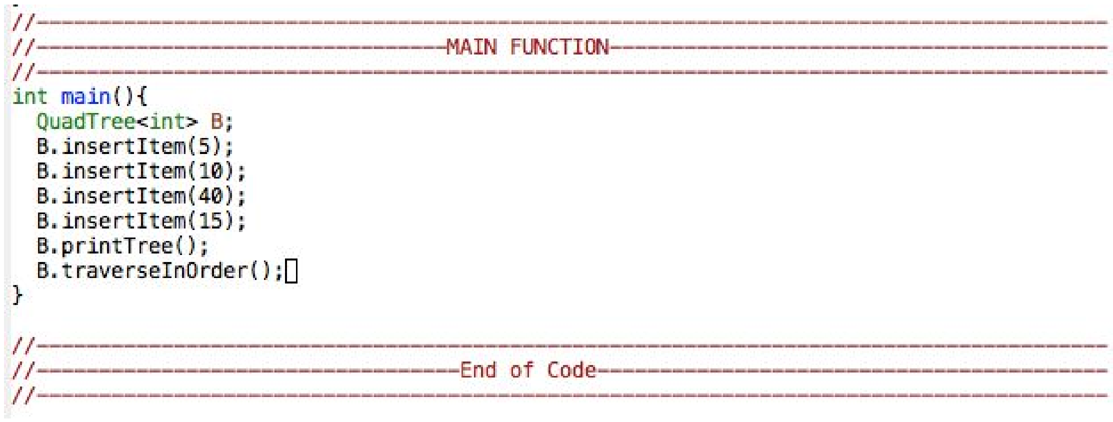


→ A more detailed QuadNode class representation:



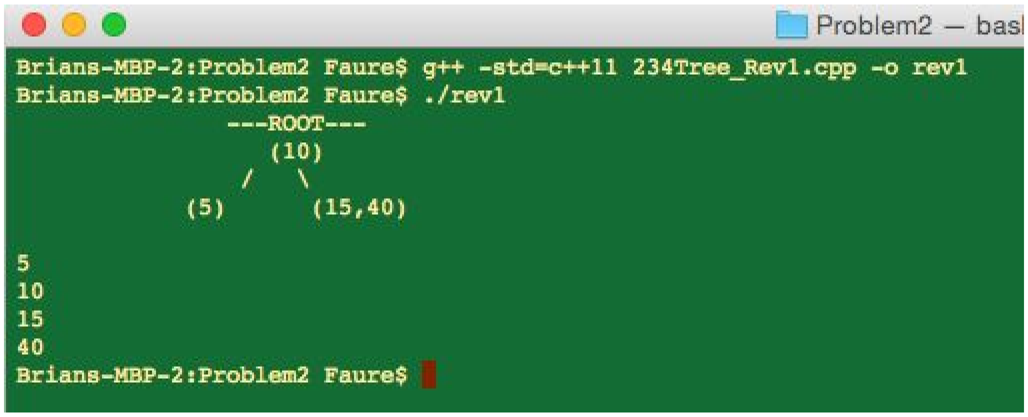
```
+getParentPtr(): QuadNode<ItemType>*
+getFirstChildPtr(): QuadNode<ItemType>*
+getSecondChildPtr(): QuadNode<ItemType>*
+getThirdChildPtr(): QuadNode<ItemType>*
+getFourthChildPtr(): QuadNode<ItemType>*

+setParentPtr(newParent: QuadNode<ItemType>*): void
+setFirstChildPtr(newchild: QuadNode<ItemType>*): void
+setSecondChildPtr(newchild: QuadNode<ItemType>*): void
+setThirdChildPtr(newchild: QuadNode<ItemType>*): void
+setFourthChildPtr(newchild: QuadNode<ItemType>*): void

+removeParentPtr(): void
+removeFirstChild(): void
+removeSecondChild(): void
+removeThirdChild(): void
+removeFourthChild(): void

+getSmallItem(): ItemType
+getMiddleItem(): ItemType
+getLargeItem(): ItemType

+setSmallItem(newSmall: ItemType): void
+setMiddleItem(newMiddle: ItemType): void
+setLargeItem(newLarge: ItemType): void

+removeSmallItem(): void
+removeMiddleItem(): void
+removeLargeItem(): void

+removeSmallItem(): void
+removeMiddleItem(): void
+removeLargeItem(): void

+swapItemSpots(arg1: string, arg2: string): void
+swapChildrenSpots(arg1: string, arg2: string): void
+outputNode(code: string): void
+splitFourNode(location: bool): bool
+insert(newItem: ItemType): bool
+searchNode(target: ItemType, &direction: string): bool
```

I have now created the entire 2-3-4 Tree implementation, see 234Tree_Rev1.cpp in the Problem2 folder for the source code. The following screenshots and brief descriptions outline the program functionality.



```
//-------------------------------------------------------------------------------------
//---------------------------------MAIN FUNCTION---------------------------------------
//-------------------------------------------------------------------------------------
int main(){
  QuadTree<int> B;
  B.insertItem(5);
  B.insertItem(10);
  B.insertItem(40);
  B.insertItem(15);
  B.printTree();
  B.traverseInOrder();
}

//-------------------------------------------------------------------------------------
//----------------------------------End of Code----------------------------------------
//-------------------------------------------------------------------------------------
```

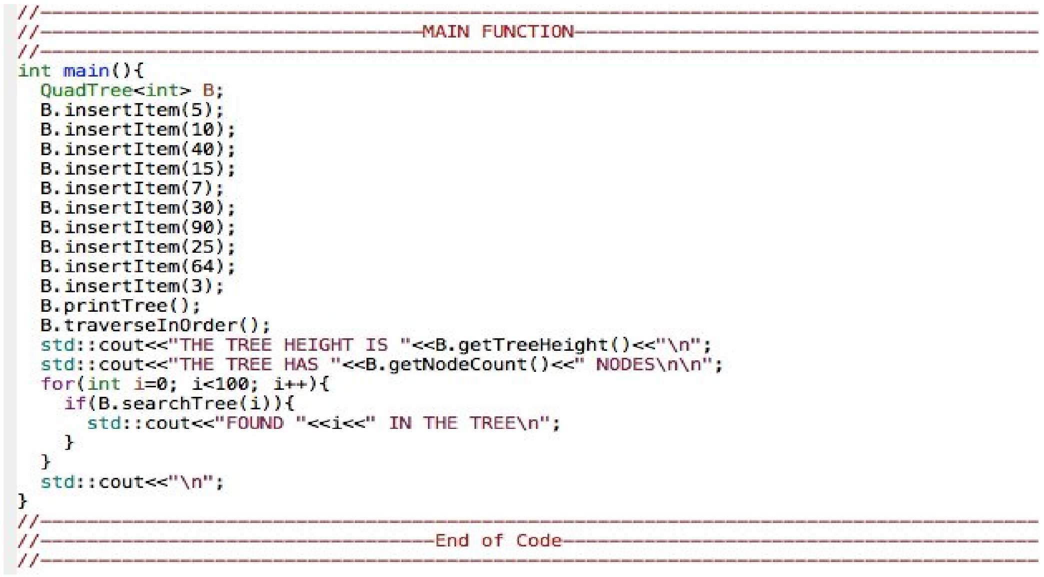
→ With this main function we are inserting 5, 10, 40, and 15 then calling the printTree() and traverseInOrder() functions to analyze the tree contents. The following screenshot is the terminal output when running this code.



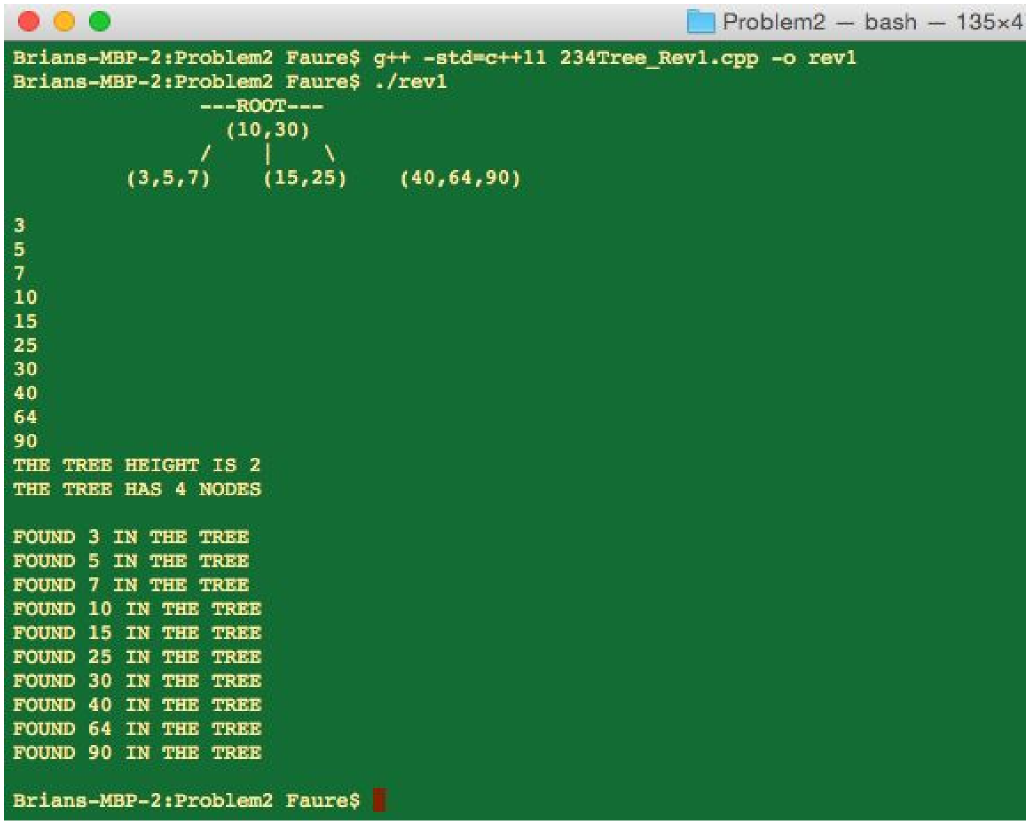
```
Brians-MBP-2:Problem2 Faure$ g++ -std=c++11 234Tree_Rev1.cpp -o rev1
Brians-MBP-2:Problem2 Faure$ ./rev1
               ---ROOT---
                  (10)
                 /   \
            (5)        (15,40)

5
10
15
40
Brians-MBP-2:Problem2 Faure$
```

→ As can be seen by the picture, the terminal now contains two main components. The tree diagram on the top is the product of the printTree() QuadTree member function and the list of elements comes from the traverseInOrder() QuadTree member function. The usefulness of the printTree() function is debatable but it was mainly used by myself while debugging the code and helped me to visualize how the QuadTree was arranging its QuadNodes. The list of items is now in sorted order, revealing that the traverseInOrder function is working correctly and that the QuadTree has its elements placed in their correct positions.

→ With the next example, I will include the searchTree(), the getTreeHeight() and the getNodeCount() functions to provide clear examples of their accuracy. See the main function below for more details.



```
//---------------------------------------------------------------------------------------
//---------------------------------MAIN FUNCTION-----------------------------------------
//---------------------------------------------------------------------------------------
int main(){
  QuadTree<int> B;
  B.insertItem(5);
  B.insertItem(10);
  B.insertItem(40);
  B.insertItem(15);
  B.insertItem(7);
  B.insertItem(30);
  B.insertItem(90);
  B.insertItem(25);
  B.insertItem(64);
  B.insertItem(3);
  B.printTree();
  B.traverseInOrder();
  std::cout<<"THE TREE HEIGHT IS "<<B.getTreeHeight()<<"\n";
  std::cout<<"THE TREE HAS "<<B.getNodeCount()<<" NODES\n\n";
  for(int i=0; i<100; i++){
    if(B.searchTree(i)){
      std::cout<<"FOUND "<<i<<" IN THE TREE\n";
    }
  }
  std::cout<<"\n";
}
//---------------------------------------------------------------------------------------
//----------------------------------End of Code------------------------------------------
//---------------------------------------------------------------------------------------
```



```
Brians-MBP-2:Problem2 Faure$ g++ -std=c++11 234Tree_Rev1.cpp -o rev1
Brians-MBP-2:Problem2 Faure$ ./rev1
                ---ROOT---
                  (10,30)
                 /   |   \
          (3,5,7)    (15,25)    (40,64,90)

3
5
7
10
15
25
30
40
64
90
THE TREE HEIGHT IS 2
THE TREE HAS 4 NODES

FOUND 3 IN THE TREE
FOUND 5 IN THE TREE
FOUND 7 IN THE TREE
FOUND 10 IN THE TREE
FOUND 15 IN THE TREE
FOUND 25 IN THE TREE
FOUND 30 IN THE TREE
FOUND 40 IN THE TREE
FOUND 64 IN THE TREE
FOUND 90 IN THE TREE

Brians-MBP-2:Problem2 Faure$
```

→ As the number of insertions into the tree increases, the printTree() function becomes less accurate because it is difficult to manage the spacing between nodes when the traversal of a tree is, by default, a recursive process. Even still, the printTree() function excels here in providing a visual example of how the tree will be structured. The getTreeHeight() function has output a height of 2, verified by the printTree() function and the number of elements in the QuadTree. The getNodeCount function has output a total number of nodes = 4, which once again is accurate and verified by the printTree(). The searchTree function in this example

has been set to search the QuadTree for all integers from 0 to 99 and has only returned true on those integers which are actually found in the QuadTree itself, verifying its functionality.
→ As I explained earlier, the QuadTree implementation is saved as 234Tree_Rev1.cpp, the only file in the Problem2 folder. The list of available functions can be summarized as follows:

User Available Functions	Source Class	Description
QuadTree()	QuadTree	The default constructor, sets rooptr to new empty QuadNode
QuadTree(QuadNode<ItemType>* root)	QuadTree	Constructor, sets rootptr to input QuadNode pointer
void setRootPtr(QuadNode<ItemType>* newRoot)	QuadTree	Sets rootptr to QuadNode pointer input
int getTreeHeight()	QuadTree	Returns integer height of QuadTree
void traverseInOrder()	QuadTree	Prints contents of QuadTree to console in ascending order
void insertItem(ItemType newItem)	QuadTree	Does nothing if item already in tree, else inserts in correct location
bool searchTree(ItemType target)	QuadTree	Returns true if target found in QuadTree, false otherwise
void printTree()	QuadTree	Prints diagram of QuadTree to console, faulty for large QuadTrees
int getNodeCount()	QuadTree	Returns the number of QuadNodes in the QuadTree

→ Note that the user can interact directly with all of the member functions of the QuadNodes but this is not guaranteed to work as expected because the QuadNode member functions do not self-organize and the resultant QuadTree would almost always be incorrect.

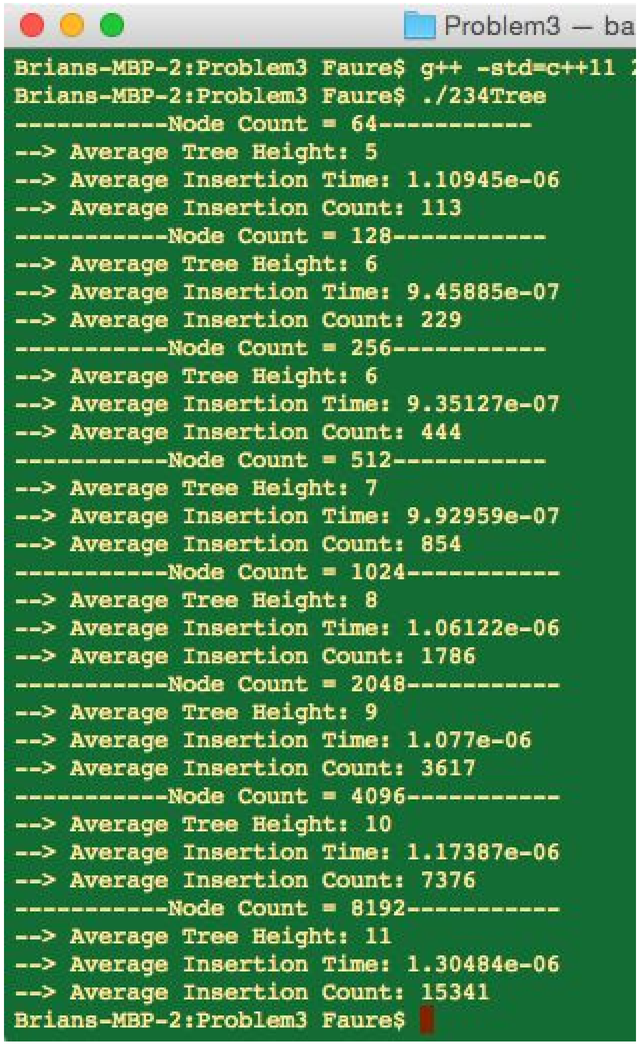
→ The final version of the 234 Tree is saved as 234Tree_Rev1.cpp under the Problem2 folder.

3. Generate 100 random trees of size (=number of nodes) ranging from 64 to 8192 (factor of 2 difference):

→ (i) & (ii) Combined in the following disscussions
→ See the Problem3 folder for the adjusted 23Tree.cpp and 234Tree.cpp files.

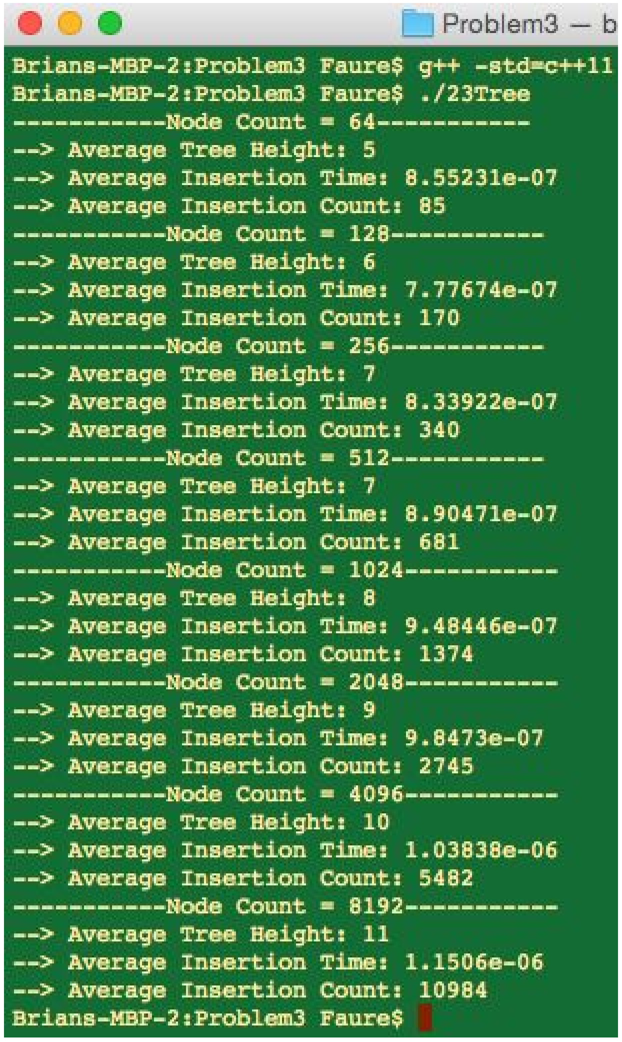
For both the 24Tree and 234Tree, I have copied the source code created new files under the Problem3 folder. These implementations contain a different main function that enables them to iterate through the different tree sizes ranging from 64 to 8192, in increments of a factor of 2, and perform exactly 100 tests at that tree size. I have used the C++ standard library rand() function to create the inputs, to ensure the numbers are truly random, I set the seed for the rand() function equal to the clock() at that exact time on each pass-through. For each tree size I have recorded the average number of successful insertions (unsuccessful meaning the item is already in the tree) to reach the required node size as well as the average time per insertion. For the integer inputs, I have set the range from 1 to 100,000 to ensure the process is not slowed down by a large amount of repeat insertions. The results of this process are summarized in the screenshots below…

234 Tree



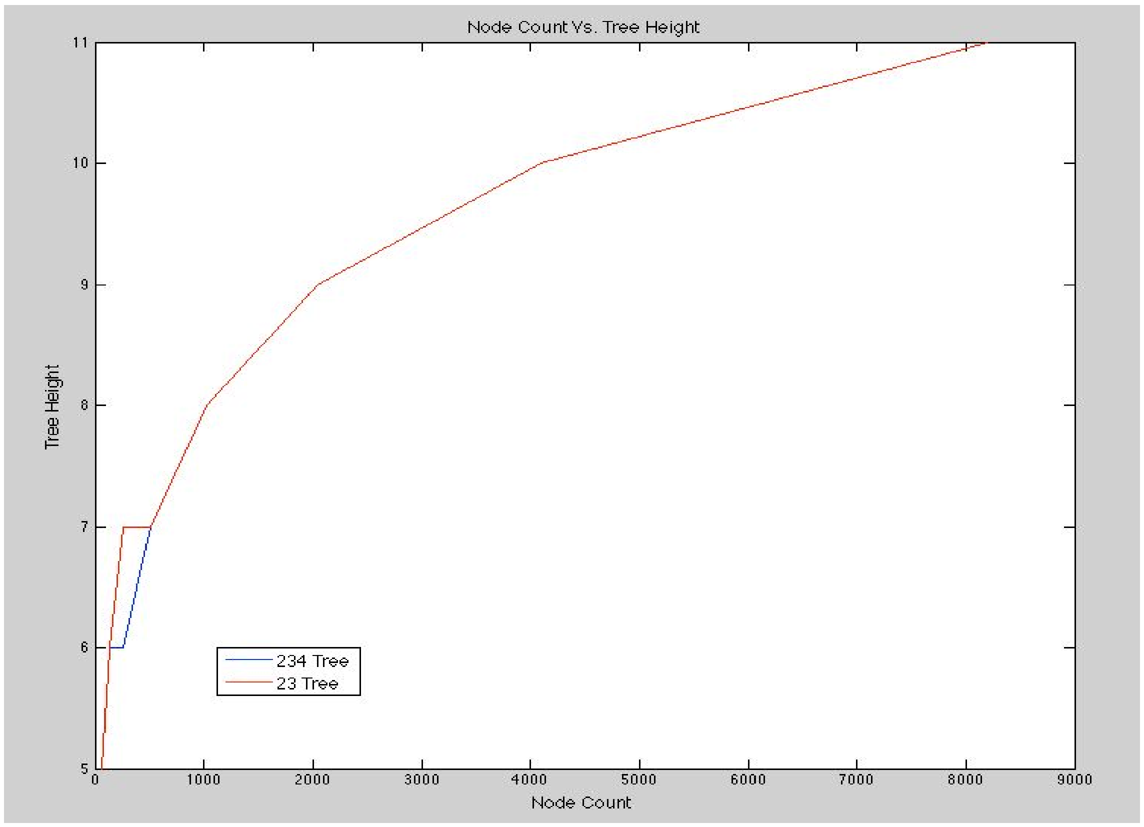
```
Brians-MBP-2:Problem3 Faure$ g++ -std=c++11
Brians-MBP-2:Problem3 Faure$ ./234Tree
-----------Node Count = 64-----------
--> Average Tree Height: 5
--> Average Insertion Time: 1.10945e-06
--> Average Insertion Count: 113
-----------Node Count = 128-----------
--> Average Tree Height: 6
--> Average Insertion Time: 9.45885e-07
--> Average Insertion Count: 229
-----------Node Count = 256-----------
--> Average Tree Height: 6
--> Average Insertion Time: 9.35127e-07
--> Average Insertion Count: 444
-----------Node Count = 512-----------
--> Average Tree Height: 7
--> Average Insertion Time: 9.92959e-07
--> Average Insertion Count: 854
-----------Node Count = 1024-----------
--> Average Tree Height: 8
--> Average Insertion Time: 1.06122e-06
--> Average Insertion Count: 1786
-----------Node Count = 2048-----------
--> Average Tree Height: 9
--> Average Insertion Time: 1.077e-06
--> Average Insertion Count: 3617
-----------Node Count = 4096-----------
--> Average Tree Height: 10
--> Average Insertion Time: 1.17387e-06
--> Average Insertion Count: 7376
-----------Node Count = 8192-----------
--> Average Tree Height: 11
--> Average Insertion Time: 1.30484e-06
--> Average Insertion Count: 15341
Brians-MBP-2:Problem3 Faure$
```

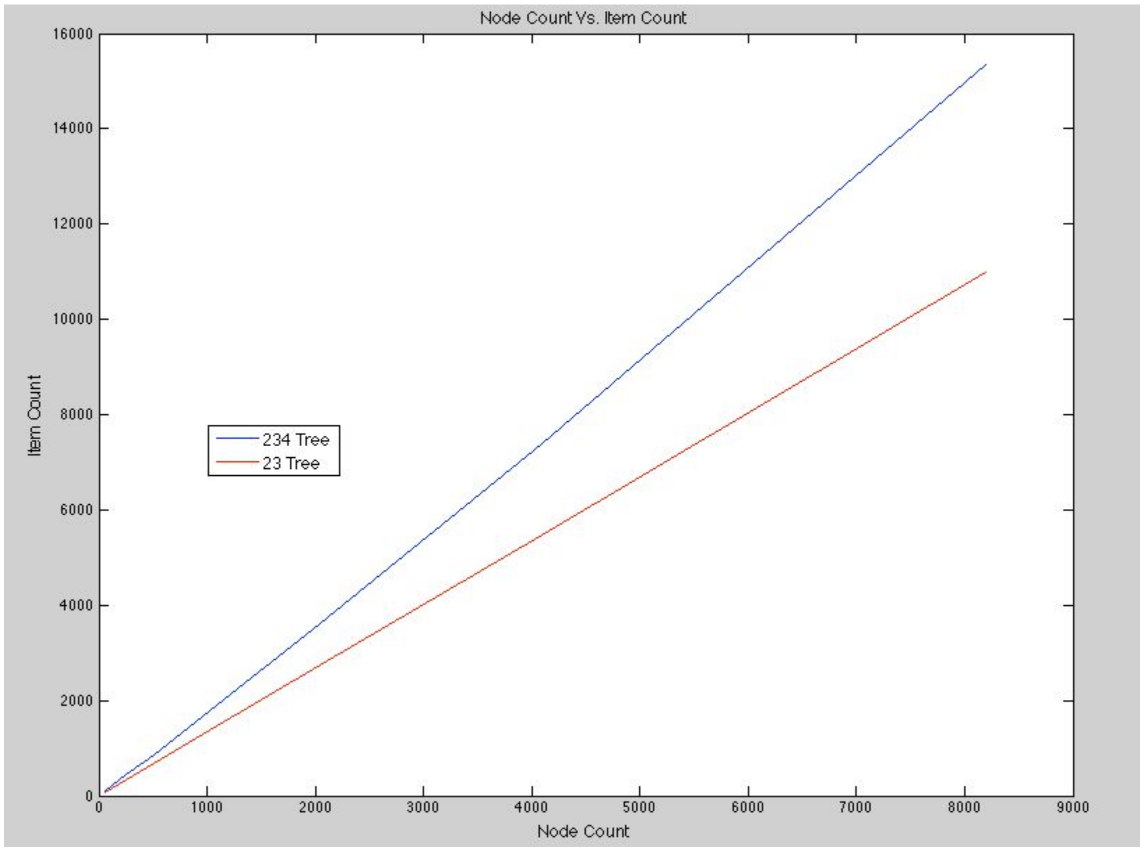
23 Tree

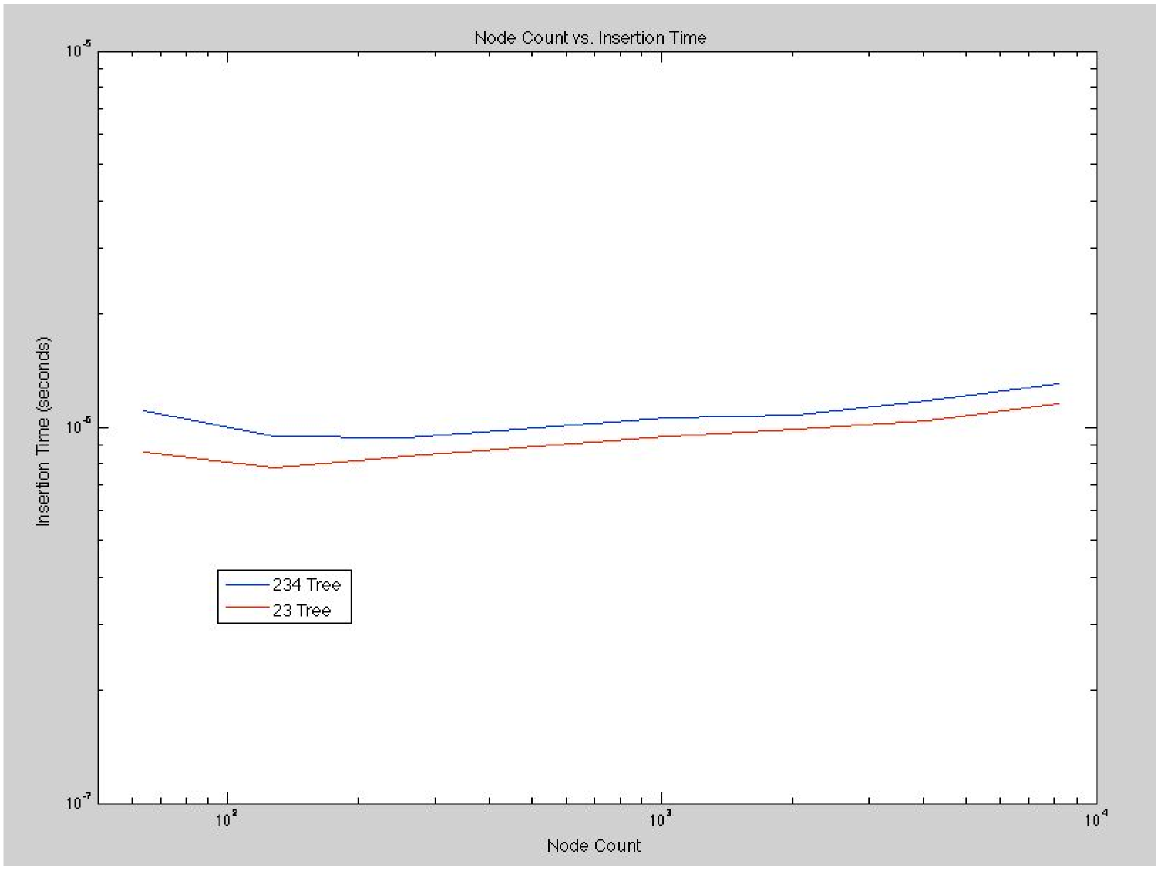


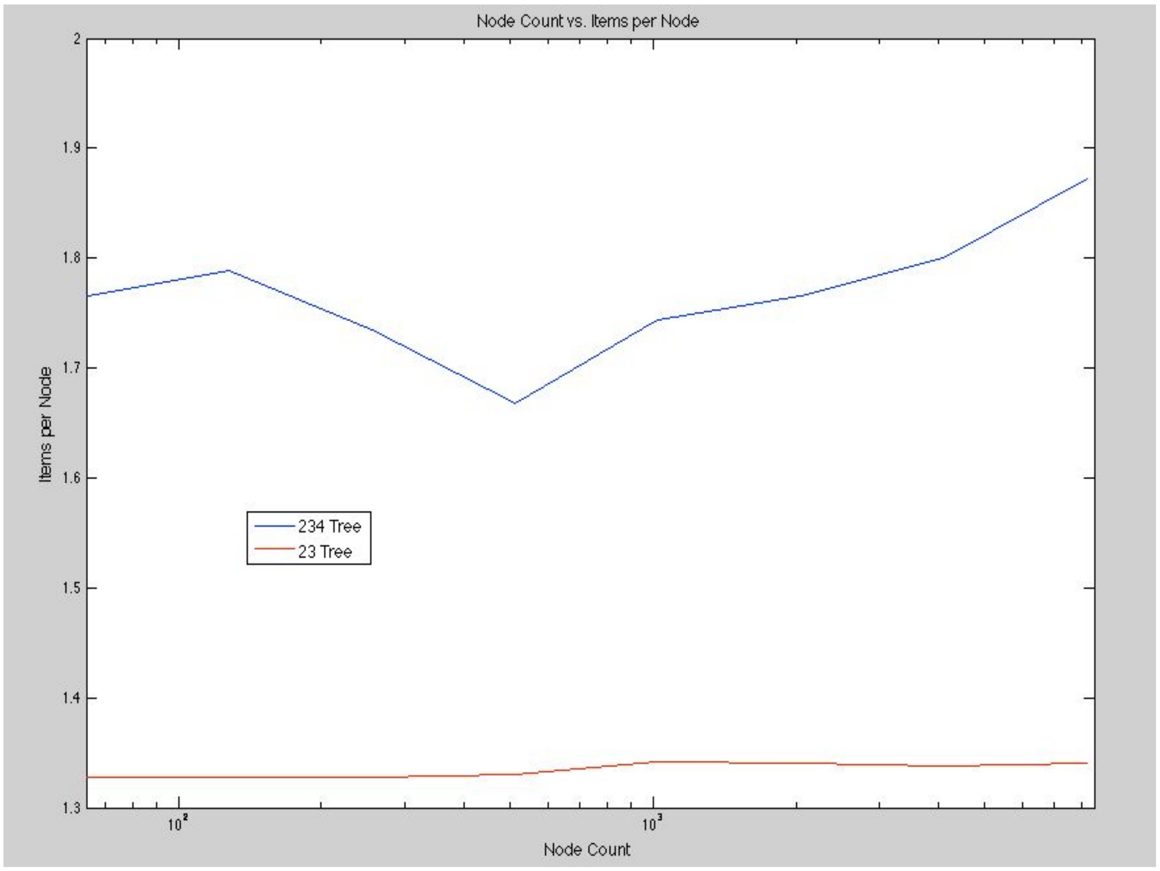
```
Brians-MBP-2:Problem3 Faure$ g++ -std=c++11
Brians-MBP-2:Problem3 Faure$ ./23Tree
-----------Node Count = 64-----------
--> Average Tree Height: 5
--> Average Insertion Time: 8.55231e-07
--> Average Insertion Count: 85
-----------Node Count = 128-----------
--> Average Tree Height: 6
--> Average Insertion Time: 7.77674e-07
--> Average Insertion Count: 170
-----------Node Count = 256-----------
--> Average Tree Height: 7
--> Average Insertion Time: 8.33922e-07
--> Average Insertion Count: 340
-----------Node Count = 512-----------
--> Average Tree Height: 7
--> Average Insertion Time: 8.90471e-07
--> Average Insertion Count: 681
-----------Node Count = 1024-----------
--> Average Tree Height: 8
--> Average Insertion Time: 9.48446e-07
--> Average Insertion Count: 1374
-----------Node Count = 2048-----------
--> Average Tree Height: 9
--> Average Insertion Time: 9.8473e-07
--> Average Insertion Count: 2745
-----------Node Count = 4096-----------
--> Average Tree Height: 10
--> Average Insertion Time: 1.03838e-06
--> Average Insertion Count: 5482
-----------Node Count = 8192-----------
--> Average Tree Height: 11
--> Average Insertion Time: 1.1506e-06
--> Average Insertion Count: 10984
Brians-MBP-2:Problem3 Faure$
```

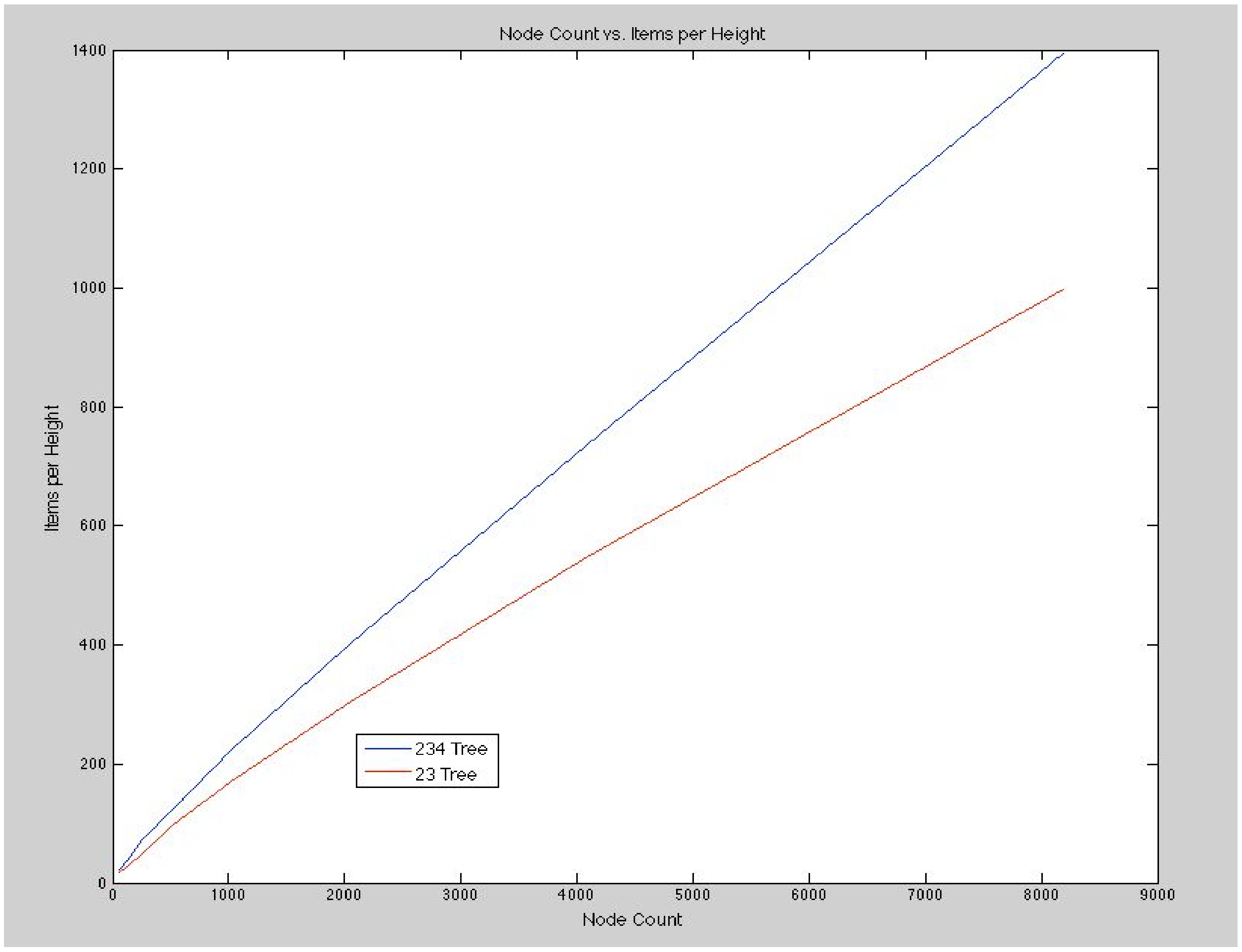
→ Using MATLAB I have created the following plots to analyze this data.






Node Count vs. Insertion Time
Insertion Time (seconds)
Node Count
10^{-5}
10^{-6}
10^{-7}
10^{2}
10^{3}
10^{4}
234 Tree
23 Tree


Node Count vs. Items per Node
Items per Node
Node Count
2
1.9
1.8
1.7
1.6
1.5
1.4
1.3
10^{2}
10^{3}
234 Tree
23 Tree



From these plots, we can learn a lot about the relative performances of the 23 Tree and the 234 Tree. In the first plot it is clear that the relationship between Node Count and Tree Height is very similar for both trees. The only deviation being that the 23 Tree added a 7th row right before the 234 Tree. Given that we know the 234 Tree should have a faster insertion time, this may lead one to believe there is cause for. The only problem being that that analysis does not include the actual number of items stored in the respective trees. For example, looking at the second plot we can see that there is a clear difference between the number of items stored in the 23 Tree vs. the 234 Tree. Towards the higher node counts, the 234 Tree is storing thousands of data items more than the 23 Tree at the same respective height.

The third plot verifies the fact that the 234 Tree is taking longer to insert items than the 23 Tree at the same Node Count, this is because, given the same number of nodes, the 234 Tree will have to make more comparisons when selecting its insertion point. Had the two trees had the same number of items, the 234 Tree would have been faster. The fourth plot illustrates what I have been explaining in that the 234 Tree can store more items per node than the 23 Tree. The fourth plot also shows that as the Node Count increases, both the 23 Tree and the 234 Tree tend to “fill out” their nodes to a higher extent; in other words, the more nodes there are in the tree, the higher the average items per node will be. Lastly, the fifth plot shows the growth of the number of nodes per unit height on each tree. Towards the higher end of the node count, the advantages of the 234 Tree become more clear.

(iii). When given the task of deciding upon a certain Tree data structure to implement, there are many circumstances that must be analyzed. For example, if you were an employee at Google and you had access to nearly limitless computing power and you were given the option between choosing a 234 Tree and a 23 Tree, you would go through several thought processes. The first being to decide whether or not you want your data structure to be quickly indexed and very quick to call upon. Your answer to this would be a clear ‘yes’. The other question is whether or not it is okay for your data structure to take up an enormous amount of room on a server or database, if you worked at Google this might not be a problem because you have personal access to an entire data center, thus your answer to this question would be another ‘yes’. You have found your solution; choose the 234 Tree. On the other hand, if you were an application designer and you were trying to decide between the 234 Tree and the 23 Tree to act as a data structure on a new mobile app you’re writing, you might be stuck at a crossroads. You want the enhanced performance of the 234

Tree yet you know you will not be provided with enough memory to implement such a design given your program will be running on mobile phones. If you were making this decision you would have to choose the 23 Tree and find ways to make it work in your implementation.

In summary, there are many reasons why one would want to choose one type of Balanced Search Tree over another and they all weigh in to the current situation to allow the person in charge to make the right decision.