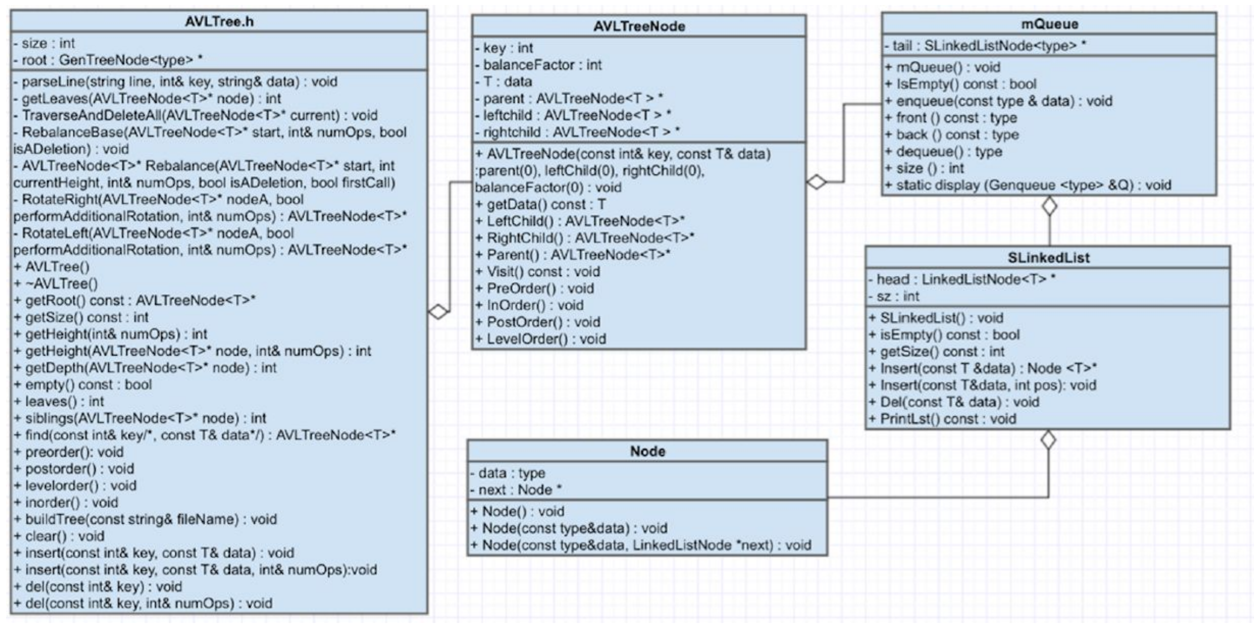
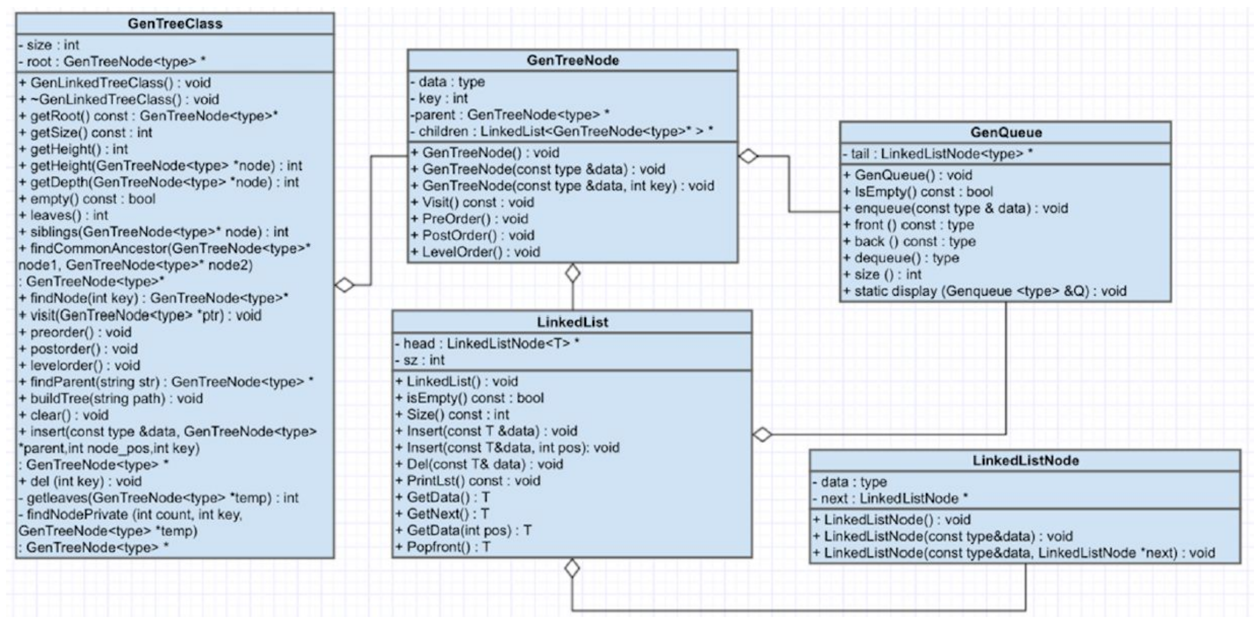
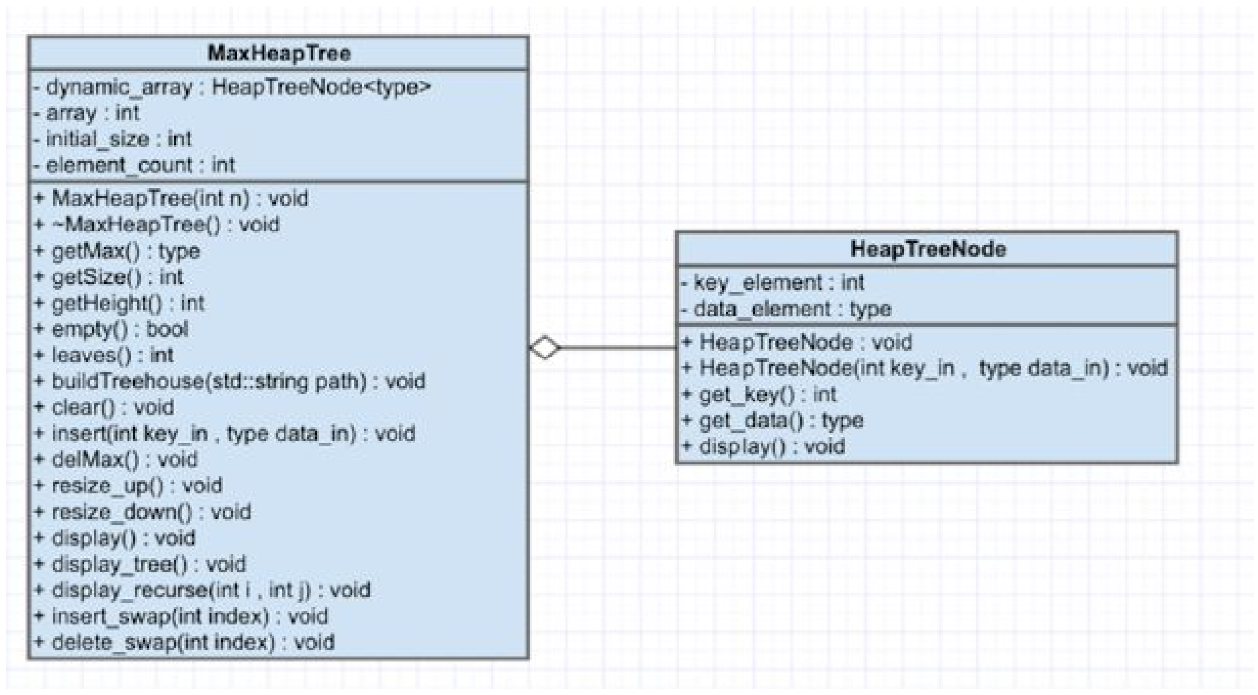
Team Project 3 Report - Trees, Trees, and more Trees

by: Manuel Martinez , John Santoro , and Jim Bui

UML Diagrams


AVLTree.h
- size : int
- root : GenTreeNode<type> *
- parseLine(string line, int& key, string& data) : void
- getLeaves(AVLTreeNode<T>* node) : int
- TraverseAndDeleteAll(AVLTreeNode<T>* current) : void
- RebalanceBase(AVLTreeNode<T>* start, int& numOps, bool isADeletion) : void
- AVLTreeNode<T>* Rebalance(AVLTreeNode<T>* start, int currentHeight, int& numOps, bool isADeletion, bool firstCall)
- RotateRight(AVLTreeNode<T>* nodeA, bool performAdditionalRotation, int& numOps) : AVLTreeNode<T>*
- RotateLeft(AVLTreeNode<T>* nodeA, bool performAdditionalRotation, int& numOps) : AVLTreeNode<T>*
+ AVLTree()
+ ~AVLTree()
+ getRoot() const : AVLTreeNode<T>*
+ getSize() const : int
+ getHeight(int& numOps) : int
+ getHeight(AVLTreeNode<T>* node, int& numOps) : int
+ getDepth(AVLTreeNode<T>* node) : int
+ empty() const : bool
+ leaves() : int
+ siblings(AVLTreeNode<T>* node) : int
+ find(const int& key/*, const T& data*/) : AVLTreeNode<T>*
+ preorder(): void
+ postorder() : void
+ levelorder() : void
+ inorder() : void
+ buildTree(const string& fileName) : void
+ clear() : void
+ insert(const int& key, const T& data) : void
+ insert(const int& key, const T& data, int& numOps):void
+ del(const int& key) : void
+ del(const int& key, int& numOps) : void
AVLTreeNode
- key : int
- balanceFactor : int
- T : data
- parent : AVLTreeNode<T > *
- leftchild : AVLTreeNode<T > *
- rightchild : AVLTreeNode<T > *
+ AVLTreeNode(const int& key, const T& data) :parent(0), leftChild(0), rightChild(0), balanceFactor(0) : void
+ getData() const : T
+ LeftChild() : AVLTreeNode<T>*
+ RightChild() : AVLTreeNode<T>*
+ Parent() : AVLTreeNode<T>*
+ Visit() const : void
+ PreOrder() : void
+ InOrder() : void
+ PostOrder() : void
+ LevelOrder() : void
mQueue
- tail : SLinkedListNode<type> *
+ mQueue() : void
+ IsEmpty() const : bool
+ enqueue(const type & data) : void
+ front () const : type
+ back () const : type
+ dequeue() : type
+ size () : int
+ static display (Genqueue <type> &Q) : void
SLinkedList
- head : LinkedListNode<T> *
- sz : int
+ SLinkedList() : void
+ isEmpty() const : bool
+ getSize() const : int
+ Insert(const T &data) : Node <T>*
+ Insert(const T&data, int pos): void
+ Del(const T& data) : void
+ PrintLst() const : void
Node
- data : type
- next : Node *
+ Node() : void
+ Node(const type&data) : void
+ Node(const type&data, LinkedListNode *next) : void


GenTreeClass
- size : int
- root : GenTreeNode<type> *
+ GenLinkedTreeClass() : void
+ ~GenLinkedTreeClass() : void
+ getRoot() const : GenTreeNode<type>*
+ getSize() const : int
+ getHeight() : int
+ getHeight(GenTreeNode<type> *node) : int
+ getDepth(GenTreeNode<type> *node) : int
+ empty() const : bool
+ leaves() : int
+ siblings(GenTreeNode<type>* node) : int
+ findCommonAncestor(GenTreeNode<type>* node1, GenTreeNode<type>* node2) : GenTreeNode<type>*
+ findNode(int key) : GenTreeNode<type>*
+ visit(GenTreeNode<type> *ptr) : void
+ preorder() : void
+ postorder() : void
+ levelorder() : void
+ findParent(string str) : GenTreeNode<type> *
+ buildTree(string path) : void
+ clear() : void
+ insert(const type &data, GenTreeNode<type> *parent,int node_pos,int key) : GenTreeNode<type> *
+ del (int key) : void
- getleaves(GenTreeNode<type> *temp) : int
- findNodePrivate (int count, int key, GenTreeNode<type> *temp) : GenTreeNode<type> *
GenTreeNode
- data : type
- key : int
-parent : GenTreeNode<type> *
- children : LinkedList<GenTreeNode<type>* > *
+ GenTreeNode() : void
+ GenTreeNode(const type &data) : void
+ GenTreeNode(const type &data, int key) : void
+ Visit() const : void
+ PreOrder() : void
+ PostOrder() : void
+ LevelOrder() : void
GenQueue
- tail : LinkedListNode<type> *
+ GenQueue() : void
+ IsEmpty() const : bool
+ enqueue(const type & data) : void
+ front () const : type
+ back () const : type
+ dequeue() : type
+ size () : int
+ static display (Genqueue <type> &Q) : void
LinkedList
- head : LinkedListNode<T> *
- sz : int
+ LinkedList() : void
+ isEmpty() const : bool
+ Size() const : int
+ Insert(const T &data) : void
+ Insert(const T&data, int pos): void
+ Del(const T& data) : void
+ PrintLst() const : void
+ GetData() : T
+ GetNext() : T
+ GetData(int pos) : T
+ Popfront() : T
LinkedListNode
- data : type
- next : LinkedListNode *
+ LinkedListNode() : void
+ LinkedListNode(const type&data) : void
+ LinkedListNode(const type&data, LinkedListNode *next) : void



AVL Tree

Summary

AVL trees are balanced binary search trees. The advantage of using a balanced tree is that find operations have a runtime of O(log2(n)). In order to maintain the balance of the tree, rotations must be applied to the nodes during insertion and deletion. Rotation types include right-right, left-right, right-left, and left-left.

Data Members

All AVL tree members are private.

AVLTreeNode<T>* root:

This member contains the pointer to the root of the AVL tree.

int sz:

This variable stores the size of the AVL tree.

Methods

Private

void ParseLine(string line, int& key, string& data):

This helper method parses a line from a file and retrieves the key and data from it. O(1) is the running time since a constant number of operations are performed.

int GetLeaves(AVLTreeNode<T>* node):

This helper method ensures that leaves() functions properly. O(n) is the running time, where n is the number of elements in the tree.

void RebalanceBase(AVLTreeNode<T>* start, bool isADeletion):

This method serves as the base case for the rebalancing function. It contains a while loop, which moves the node start from its position as the parameter to the top of the tree. This function resolves all issues with imbalances, and has a runtime which varies based on the number of deletions that must be performed. In the best case, the running time is O(n), where n is the number of elements in the tree.

AVLTreeNode<T>* Rebalance(AVLTreeNode<T>* start, int currentHeight, bool isADeletion, bool firstCall):

This method iterates through each node from the starting node to the root. As it is doing so, it checks the node's balance factor using a variable currentHeight and the height of the subtree opposite to the direction that is being approached. The balance factor in the node in the tree is then checked, and if it is equal to +/-2, the method returns the imbalanced node. O(n) is the worst-case running time for this particular method.

AVLTreeNode<T>* RotateRight(AVLTreeNode<T>* nodeA, bool performAdditionalRotation):

This method performs either a right-right or a left-right rotation, depending on whether the node's child, from the path previously taken, has a balance factor which is less than 0. Right-right rotations are performed when node A has a balance factor of +2 (the left subtree is 2 links taller than the right subtree) and node B, which is node A's left child, has a balance factor that is greater than or equal to 0. The running time of this method is O(1) because a finite number of operations are performed. However, it can greatly extend the running time of the function because the traversal is altered.

AVLTreeNode<T>* RotateLeft(AVLTreeNode<T>* nodeA, bool performAdditionalRotation):

This method performs either a left-left or a right-left rotation, depending on whether the node's child, from the path previously taken, has a balance factor which is greater than 0.

Left-left rotations are performed when node A has a balance factor of -2 (the right subtree is 2 links taller than the left subtree) and node B, which is node A’s right child, has a balance factor that is less than or equal to 0. The running time of this method is O(1) because a finite number of operations are performed. However, it can greatly extend the running time of the function because the traversal is altered.

void TraverseAndDeleteAll(AVLTreeNode<T>* current):

This helper method ensures that clear() removes all of the elements in the tree. O(n) is the running time, where n is the number of elements in the tree.

Public

Constructor:

This method initializes an AVL tree with the default values. O(1) is the running time since a constant number of operations are performed.

Destructor:

This method frees all resources associated with the AVL tree and then deletes the object. O(n) is the runtime since n items must be deleted.

AVLTreeNode<T>* getRoot() const:

This method returns the root of the AVL tree. O(1) is the running time since a constant number of operations are performed.

int getSize() const:

This method returns the number of elements in the AVL tree. O(1) is the running time.

int getHeight():

This method returns the height of the AVL tree. O(n) is the running time since the method recursively searches all of the elements starting from the root.

int getHeight(AVLTreeNode<T>* node):

This method returns the height of a particular subtree of the AVL tree. O(n) is the running time, where n is the number of nodes in the subtree pointed to by node.

int getDepth(AVLTreeNode<T>* node):

This method returns the depth of a particular node in the AVL tree. To do so, it iterates from the node to its respective parents and then returns the number of operations performed. O(log2(n)) is the running time because AVL trees are balanced.

bool empty() const:

This method checks to see whether the root of the tree is a null pointer. If this is the case, then the tree is empty. O(1) is the running time since a constant number of operations are performed.

int leaves():

This method returns the number of leaves in the tree. Because it must check to ensure that each node has no children, its runtime is O(n), where n is the number of elements in the tree.

int siblings():

This method returns the number of children each of the nodes has. Due to the fact that this is an AVL tree, it may return 0, 1, or 2. Its runtime is O(1) since a constant number of operations are performed.

AVLTreeNode<T>* find(const int& key):

This method returns the first item in the AVL tree whose key is equal to the argument. The function works as follows:

1. If key < nodeKey, go to node’s left child.
2. If key > nodeKey, go to node’s right child.
3. If key == nodeKey, return this node.

Because AVL trees are balanced, the runtime of the algorithm is O(log2(n)), where n is the number of elements in the tree. The worst case scenario is one in which the element with the entered key does not exist (an underflow error will be thrown).

void Preorder():

This method calls helper functions in the tree’s node class to perform preorder traversal (VLR). Since all elements must be visited, O(n) is the running time.

void Postorder():

This method calls helper functions in the tree's node class to perform postorder traversal (LRV). Since all elements must be visited, O(n) is the running time.

void Levelorder():

This method calls helper functions in the tree's node class to perform level order traversal. This type of traversal uses a queue to store nodes and then print them off such that one can see them in order of level. Since all elements must be visited, O(n) is the running time.

void Inorder():

This method calls helper functions in the tree's node class to perform in order traversal (LVR). Since all elements must be visited, O(n) is the running time.

void Buildtree(string filename):

This method reads in a filename from the user and then inserts each item from the file into the tree. Due to the fact that imbalances during insertion must be resolved, O(n^2) is the approximate running time, where n is the number of elements in the file.

void clear():

This method clears all of the elements from the AVL tree. O(n) is the running time, where n is the number of elements in the tree.

void insert(const int& key, const T& data):

This method inserts an item into the tree and then performs the necessary rotations to keep the tree balanced. In the best case scenario, when the insertion does not cause an imbalance, the running time is O(n), where n is the number of items in the tree. (O(n) because the tree's balance must be checked using getHeight(...)). However, when there is an imbalance, the running time is much greater because rotations are required to resolve the imbalances, and rotations can cause changes in the tree's height. Based on empirical testing, the worst case running time seems to be O(n^2 / 2) or simply O(n^2).

void del(const int& key):

This method deletes an item from the AVL tree and then performs the necessary rotations to keep it balanced. In the best case scenario, when the deletion does not cause any imbalance in the tree, the running time is O(n), where n is the number of elements in the tree. However, the runtime is much greater when deletions cause imbalances, especially since there are cases in which more than one complex rotation must be performed to keep the tree balanced. Based on empirical testing, the worst case running time seems to be O(n^2 / 2) or O(n^2).

Max Heap Tree

A max binary heap tree is a particular type of implementation of a priority queue. The maximum key associated with each node is stored at the root of the tree. The tree stays a complete binary tree at all times, and all entries in the tree follow the heap property (no child has a key larger than its parent's key).

In this project, the tree is represented as a dynamic array, starting with the array location 1 as the root. The 0th position of the array is bored and ignored. The children of each respective nth node is 2n and 2n + 1. The parent of each node can be found with n / 2, except for the root, which is an orphan and has no parent.

MaxHeapTree(int)

This is the constructor. It creates an array from the given input and sets the private values as approriate. This will be O(1).

getMax()

This returns the value of the item with the maximum key, stored at the root of the tree. This will be O(1).

getSize()

This returns how many items are in the tree. This will be O(1).

getHeight()

This returns the height of the tree using an equation based off the premise that the tree is a complete binary tree at all times. This will be O(1).

empty()

This checks if whether the tree has elements or not. This will be O(1).

leaves()

This returns how many leaves (nodes with no children) the tree has. The equation is based off the premise that the tree is a complete binary tree at all times. This will be O(1).

buildTreehouse(string)

This reads in a text file and inserts it into the tree using the insert() operator. This will be O(nlog_2(n)), depending on how many items are in the text file and the worst case scenario of inserting.

clear()

This deletes the tree and resets it to the initial value. This will be O(n).

insert(int , type)

First the array is resized as necessary using resize_up(). This then takes in a int key and a type data and inserts it into a tree at the bottom of the list. It then compares the node to its parent to see if the heap property is satisfied. If not, it is then recursively swapped with the parent until it satisfies the heap property using insert_swap(). In the worst case scenario, the node will travel until it reaches the root, making this O(log_2(n)).

delMax()

This "pops" the value with the highest key, i.e. the root. It is done by swapping the root node with the last item in the array. The last item is deleted, and the root is then checked to see if the heap property is preserved. If not, it recursively fixes it using delete_swap(). The worst case scenario is that the root must travel to the bottom of the tree, making this O(log_2(n)). The array is then resized as necessary using resize_down().

resize_up()

This doubles the array size by creating a new array twice the size of the original and copying all the original nodes over. It is then set as the main array. This will have a running time of O(n).

resize_down()

This halves the array size by creating a new array with half the capacity (barring the lower limits of 2 and initial value) and copying the original nodes over. It is then set as the main array. This will have a running time of O(n).

display()

This simply displays the contents of the array.

insert_swap(int)

This is a recursive function to fix the heap property. It checks if the parent is larger, if not, the two are swapped, and the function is called again on the new position. This happens until the heap property is obtained. In the worst case, the node will travel until it is the root, making the run time O(log_2(n)).

delete_swap(int)

This is the same as insert_swap(), however it checks the children, swapping with the child that is larger of the two. In the worst case, the node will travel from the root all the way down the tree, making the run time O(log_2(n)).

General Tree

General Tree is build of various classes. It contain the GenTreeClass, GenTreeNode Class, LinkedList, LinkedListNode, and GenQueue. The LinkedList is used to store a Tree nodes children as instructed by the project documents. It hold the children and its siblings. The GenQueue is used for level order traversal. The tree is a set of notes that hold key, value, parent and a set of linked list of children. Here are the functions utilize for the GenTreeClass.

GenLinkedTreeClass():root(0),size(0)

This constructor initialized the root node and size of tree to null pointers. This will create a a tree instance. This is O(1).

GenLinkedTreeClass()

This is the tree class destructor. It will delete all the elements store in the tree as well as the root. This is O(1).

GenTreeNode<type>* getRoot() const

This method will return the root of the tree. It simply returns the root node. This is O(1).

int getSize() const

This method return the size of the current tree. The value size is a data member of the class tree. It will be updated as necessary in the inset method refer to below. This is O(1).

int getHeight()

The get height method returns the height of the tree according to the root node. This function called the sister method getHeight(root) that will calculate the height and return an int. This is O(n).

int getHeight(GenTreeNode<type> *node)

This get height method will have a parameter of a genTreeNode. This function will calculate the height of the tree by going through the node to its deepest dependent and determining the height from the parameter node. This is O(n).

int getDepth(GenTreeNode<type> *node)

The get depth method will take a parameter of GenTreeNode and will calculate the depth of that node by counting the steps to get from that node to the tree. This function will return an int with the depth. This is O(n).

bool empty() const

The empty function will be of bool and will return true if the tree is empty. The function will check to see if the root is empty. This is O(1).

int leaves()

The leaves method will call a private function in GenTreeClass that will calculate the amount of leaves the tree has. This is O(n).

int getleaves(GenTreeNode<type> *temp)

This method is only called by the leaves method. It has a parameter of GenTreeNode to calculate the leaves of the tree. The function will traverse through the tree until finding a null child and this determines that this node is in fact a leave by definition. A variable count will be return to leaves with the amount of leaves in the tree. This is O(n).

int siblings(GenTreeNode<type>* node)

This sibling class will take a parameter of GenTreeNode and will find the sibling of that node. The class will simply call the getSize() function of the children since it will be a linked list and that number is returned as the number of siblings. This is O(n).

GenTreeNode<type>* findCommonAncestor(GenTreeNode<type>* node1, GenTreeNode<type>* node2)

This class will find the common ancestors of two node. It will take two GenTreeNodes and will get the depth of each. From there it will then compare the sizes of both nodes and bring them to the same depth. If the nodes match then the nodes have the common ancestor. This node ancestor will be return. This is O(n).

GenTreeNode<type>* findNode(int key)

This method will take a key argument and will try to find this key in the tree. The function calls a private function FindNodePrivate which will return the node that matched the key in the argument. It will then return that node. This is O(n).

GenTreeNode<type> * findNodePrivate (int count, int key, GenTreeNode<type> *temp)

This function will only be called by the find node function. It will take in argument a count int and key int and a GenTreeNode. The function will traverse through the list to find the correct node given the key in the argument. This node will be returned to the original function. This is O(n).

void visit(GenTreeNode<type> *ptr)

This function will take a GenTreeNode argument and will visit the data at that node. It will display that nodes data. This is O(n).

void preorder()

This function will traverse the tree in a preorder fashion. The function calls a preorder function in the GenTreeNode class. The following shows the code use to traverse the list. This is O(n).

```
if (this != nullptr)
{
        this->Visit();
        LinkedList<GenTreeNode<type> *> * tmp = this->children;

        int sz = this->children->getSize();
        int counter = 0;

        for (counter = 0; counter < sz;counter++){
                tmp->getData(counter)->PreOrder();
        }
}
```

void postorder()

The post order function will traverse the tree in a post order fashion. It will call a post order function in the GenTreeNode class. similar to preorder it will traverse the list but visit the node after the for loop. This is O(n).

void levelorder()

This function will traverse the tree in a level order fashion. It will call a lever order function in the GenTreeNode class that will perform the traversal. The traverse is similar to the pre and post order traversals but differs in that it uses a queue. A queue is used to enqueue and dequeue all the tree node while visiting them. Here is the structure for the level order function. This is O(n).

```
if (this == nullptr)
        return;
GenQueue< GenTreeNode<type>* > Q;
Q.enqueue(this);

while (!Q.isEmpty()){
```

```
                GenTreeNode<type> *n = Q.dequeue();
                n->Visit();

                int sz = n->children->getSize();
                int counter = 0;

                for (counter = 0; counter < sz;counter++){
                        Q.enqueue(n->children->getData(counter));
                }
        }
```

GenTreeNode<type> * findParent(string str)

This function was originally used but then discarded. This function used a string from the text file to locate the parent and return a newly created tree node of that data. This function serve no purpose once the build tree function was modified. This is O(n).

void buildTree(string path)

This function takes for an argument a path string which is uses to determined the various data member of a tree node. By using string function we can determine the key, data, and position of the information given on the tree. This is O(n).

void clear()

This function will simply clear the tree root and all its children. This is O(1).

void insert(const type &data, GenTreeNode<type> *parent,int node_pos,int key)

The inset method will insert a new tree node using the data, genTreeNode parent and key information as the parameters. It will first check to see if parent is null and if so, will make the new root the current passed variables. It will then check to see if the parents children is empty, and if so will update the the parent and will insert the child. And if none of the above it will update the parent and inset the new child using the temp node and node position. The size of the tree is updated at each of these steps. This is O(n).

void del (int key)

This function will take a key to delete a specific node on the tree. The function will call find node function to locate the node and then it will delete the node. If this node has any children, then the children will replace the parent node deleted and will have its own children. Since this function call the find node it will be O(n).