Effectiveness of an AVL Tree

Joseph Garro (jmg289@uakron.edu)

Abstract:

The AVL Tree is an extension of the Binary Search Tree that adds mechanisms to make the structure self balancing, which keeps the height of the root node's subtrees approximately equal to provide searches with a fast time complexity. By comparing the execution time of the AVL Tree to two other data structures, the Skip List and STL List, it was determined that the AVL Tree had the fastest searches, but it experienced slower insertion and deletion times compared to the Skip List. Testing the effect of sorted versus unsorted input on the execution time of the operations supported by the three data structures, it was found that sorted input could result in a large decrease in execution time or no change at all. Evidently, it was found that sorted input drastically improved the performance of both the AVL Tree and Skip List. However, the difference between the Skip List's search execution time and AVL Tree's search execution time is so small (0.008 for a problem size N = 256000) that the faster insertion and deletion times provided by the Skip List make it a better overall choice to implement compared to the AVL Tree. Additionally, the implementation of the AVL Tree that was designed for this research was determined to have an overhead fraction of 2/3, making it an inefficient solution from a storage perspective. Therefore, it can be concluded that even though the AVL Tree provides fast search speeds, other data structures such as the Skip List can achieve comparable search speeds while also providing faster insertion and deletion times, making the AVL Tree an ineffective structure to use for most applications.

Introduction:

The purpose of this research is to determine the effectiveness of an "Adelson-Velskii and Landis" (Morris, 1998) Tree, or AVL Tree, compared to other data structures similar to it: The Skip List and the STL list. The goal was to determine the time complexities of the insert, delete, and search operations implemented in each data structure, as well as determining the space complexity of each data structure. The time complexity of each operation was obtained by observing how the runtime of each operation grew with respect to the problem size N. To determine the space complexity of each structure, the amount of space required to implement each data structure and its associated overhead was computed. This research also determined the effect of using sorted and unsorted input when performing each of the three operations, providing yet another metric by which the three data structures could be compared and ranked. To achieve the results, the AVL Tree was researched and a custom AVL Tree class was created. The code for a Skip List class was obtained from another student, Graham Campbell (glc31@uakron.edu), and the list was implemented using the STL's list container. These data structures were tested using a driver code that recorded the time taken by each structure's implementation of the three operations with respect to the problem size in order to quantitatively determine which data structure was the best at performing each operation. Therefore, the result of this research is that the time complexity for insertions, deletions, and searches were obtained for the AVL Tree, Skip List, and STL list, as well as each structure's associated space complexity and optimal type of input. This allows the best data structure among the three to be determined with respect to the type of operations that will be performed, the amount of space available, and the type of input.

Discussion:
Hypothesis: The AVL Tree should have the fastest searches among all the structures due to being based on a Binary Search Tree (BST), but the performance of inserts and deletions will likely suffer due to overhead incurred from the Tree's self balancing nature. Therefore, is is expected that the AVL Tree should perform slower than the Skip List for insertions and deletions. Meanwhile, the STL list is expected to be slower than both structures due to its unoptimized nature. Finally, the use of sorted input versus unsorted input should produce a measurable difference in the recorded run for each of the three operations, with sorted input likely resulting in faster run times compared to unsorted input.

An AVL Tree is based on a BST and contains a root node with 0, 1 or 2 children. Each child can then serve as a root node to its own subtree, allowing the structure to be recursively defined. As the AVL Tree is fundamentally a BST, it follows the Binary Search Tree property and requires the elements on the left subtree of a given node have a value less than the node's value and the values on the right subtree must have a value greater than or equal to it. The AVL Tree further extends the functionality of the BST to provide mechanisms which ensure that the AVL Tree remains balanced after every insertion and deletion of an element, preventing the AVL Tree from becoming lopsided. The AVL Tree is considered balanced if the height of two subtrees shared by a single parent have a difference no greater than two, as illustrated in the figure below.

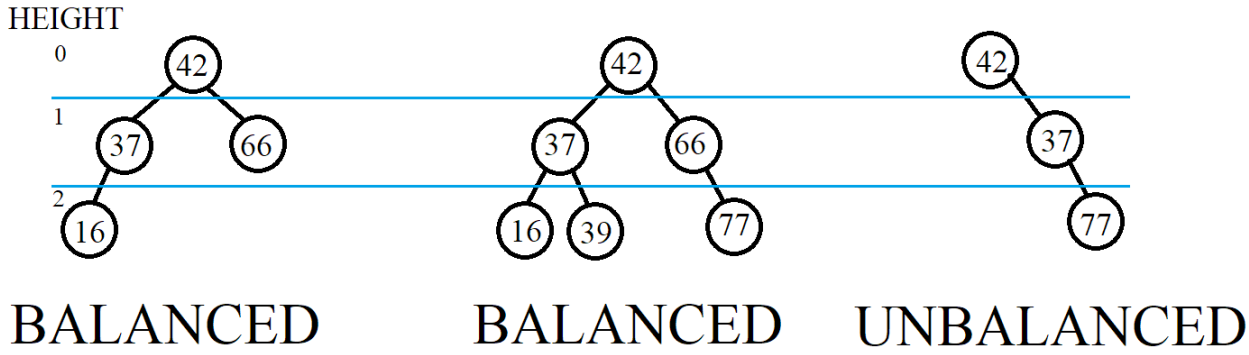

Figure 1: Example of balanced and unbalanced AVL Trees.

Beginning with the leftmost AVL Tree, the root node 42 has a height of 2. The left subtree of the root node has a height of 1 and the right subtree a height of 1. Thus, the difference in heights at the root node is 2 - 1 = 1, so the root node is balanced. Looking at the middle AVL Tree, the height of the root node's left subtree is 2 and the height of the right subtree 2, yielding a height difference of 0 so the root node is balanced. Repeating this process down each subtree yields the same result of being balanced. Finally, looking at the right AVL Tree, the root has no left subtree, so its height is 0. Meanwhile, the height of the root node's right subtree is 2, which produced a difference in height equal to 0 - 2 = -2. Thus, the root node is unbalanced.

The convention used to obtain the difference in height between subtrees in this research was to always subtract the height of the right node from the height of the left node. If the height of the node is always updated at insertion or deletion, this ensures that the difference between two subtrees can never exceed 2 and the AVL Tree will remain balanced as elements are inserted and deleted. From the convention of subtracting the height of the right subtree from the left, this difference in height, known as the balance factor, is also indicative of the subtree causing the

imbalance. Evidently, a balance factor of positive 2 requires that the height of the left subtree be 2 larger than the height of the right subtree, while a balance factor of -2 requires the opposite. Continuing the assumption that the heights of the subtrees and their respective nodes are always updated at insertion and deletion, any imbalance can be detected as soon as it arises. Supposing that unbalances occur due to insertions only, 4 possibilities exist for an unbalanced subtree and are illustrated in Figure 2 below.

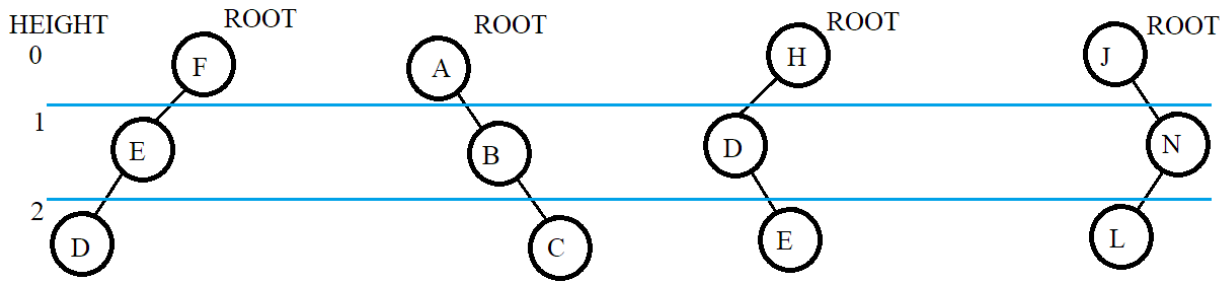

Figure 2: Example of insertions to unbalance AVL Trees.

As shown in Figure 2, it is the insertions into the row with a height corresponding to 2 that unbalance the AVL Tree, as the root node has a balance factor of 1 or -1 when only the nodes at the row corresponding to a height of 1 are present. These cases are labeled according to the path of the 2 nodes that cause the unbalance underneath the root node. From left to right, these are: left left imbalance (node D is the left child of node E which is the left child of root node F), right right imbalance (node C is the right child of node B which is the right child of root node A), left right imbalance (node E is the right child of node D which is the left child of root H), and right left imbalance (node L is the left child of node N which is the right child of root J). Additionally, any of the children of the root node can have up to 2 children but only the essential nodes are shown for simplicity. Furthermore, the node labeled "root" does not have to be the root of the entire AVL Tree (although this root can be unbalanced as well), and instead is the root of the unbalanced subtree. Each of the above imbalances requires a corresponding correction in order to rebalance the AVL Tree relative to the unbalanced subtree's root. Performing a correction also requires that the children of the nodes be appropriately moved, although this is not shown. The correction corresponding to each imbalance is illustrated and described briefly below.

Left Left Imbalance:

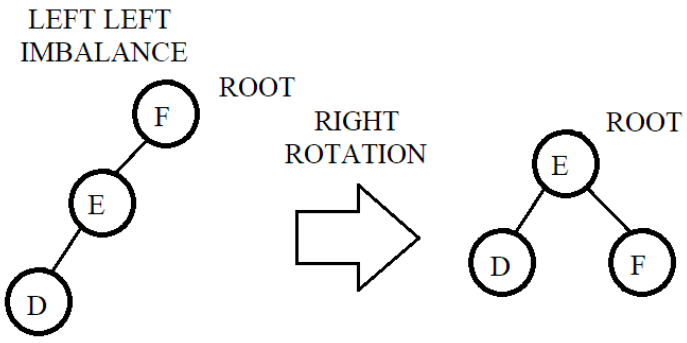

Figure 3: Depiction of performing a right rotation to correct a left left imbalance.

To correct a left left imbalance, the left child of the root node becomes the new root node, and the previous root node becomes the right child of the new root node. The resulting balanced AVL Tree obeys the BST property as the root node has a value greater than its left child and smaller than its right child.

Right Right Imbalance:

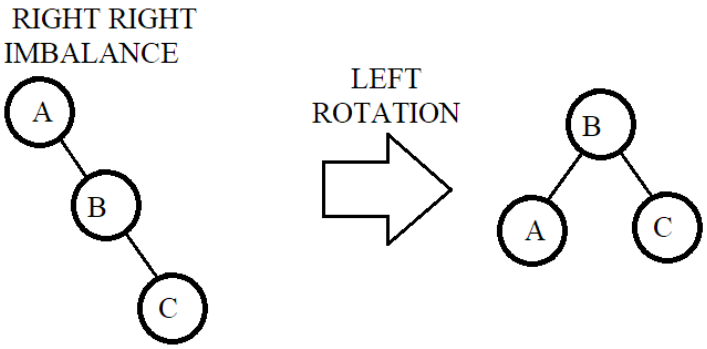

Figure 4: Depiction of performing a left rotation to correct a right right imbalance.

To correct a right right imbalance, the right child of the root node becomes the new root node, and the previous root node becomes the left child of the new root node. The resulting balanced AVL Tree obeys the BST property as the root node has a value greater than its left child and smaller than its right child.

Left Right Imbalance:

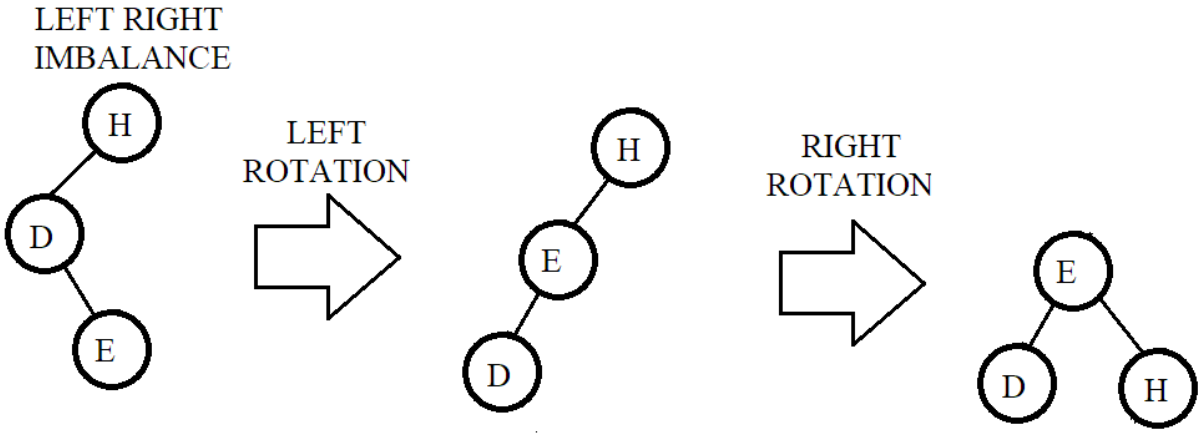

Figure 5: Depiction of performing a left rotation followed by a right rotation to correct a left right imbalance.

To correct a left right imbalance, a left rotation is performed on the left child of the root node to obtain the unbalanced AVL Tree shown in the middle of Figure 5. Then a right rotation is performed to obtain the balanced AVL Tree shown on the right. Again, the BST property is followed by the new balanced AVL Tree.

Right Left Imbalance:

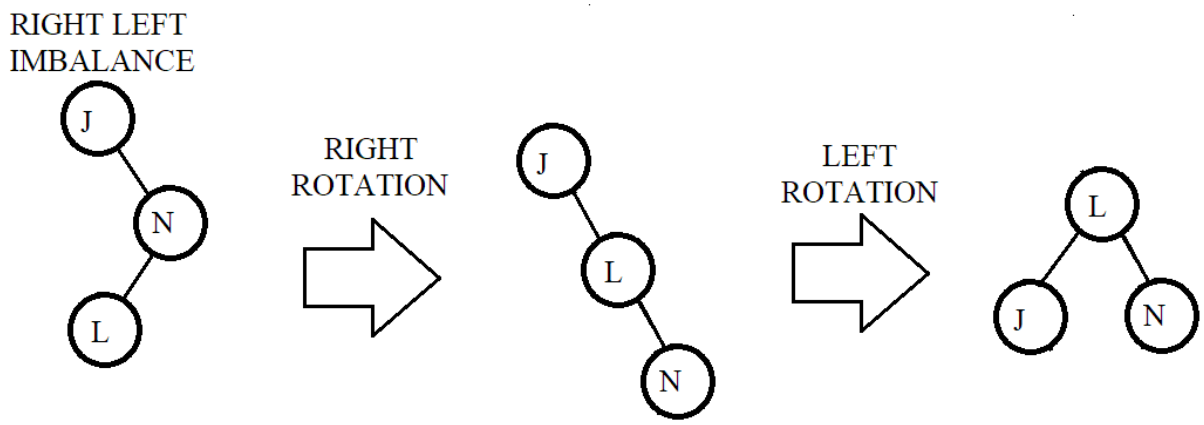

Figure 6: Depiction of performing a right rotation followed by a left rotation to correct a right left imbalance.

To correct a right left imbalance, a right rotation is performed on the right child of the root node to obtain the unbalanced AVL Tree shown in the middle of Figure 6. Then a left rotation is performed to obtain the balanced AVL Tree shown on the right. Once again, the BST property is followed by the new balanced AVL Tree.

Once it is determined that a correction is needed, the balance factor can once again be used to determine the required correction. Extending the idea of imbalances to occurring with the deletion of elements as well, the potential right and left subtrees that imbalance a given AVL Tree and their corrections are as follows:

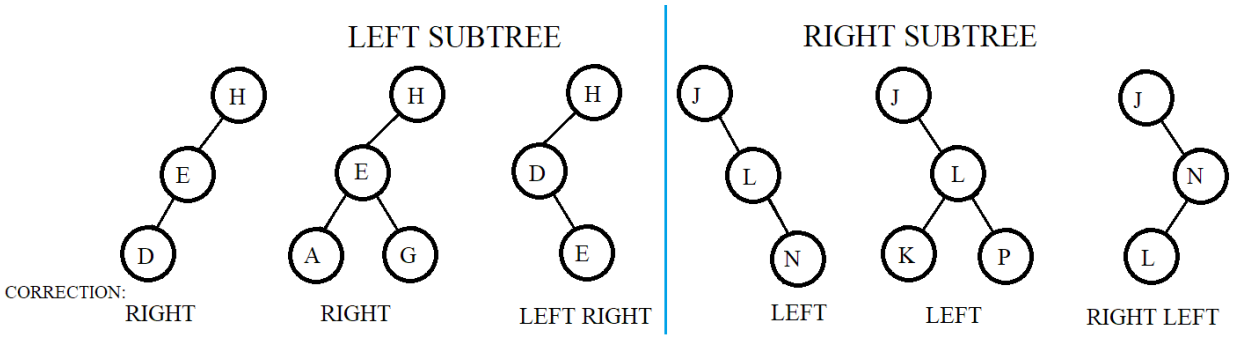

Figure 7: Depiction of potential imbalances and corresponding correction on the left and right subtrees of an AVL Tree.

Observing figure 7, it is evident that a single right rotation occurs whenever a root node has a left child that has a left child. If the left child only has a right child, a left right rotation occurs. Likewise, on the right subtree, the root node's right child only having a left child signifies a right left rotation- all other possibilities result in a single left rotation. This fact is expressible by taking the balance factor of of the children of the root node's child causing the imbalance. Following Figure 7, the balance factor of node E on the two right rotation cases is 1 and 0 respectively. Similarly, the balance factor of node D on the left right rotation instance is -1. Thus,

a balance factor greater than or equal to 0 of the root node's first child node corresponds to a right rotation on the left subtree, and a balance factor of -1 a left right rotation. This is reversed for the right subtree, with node L for the 2 left rotation cases having a balance factor of -1 and 0 respectively. For the right left case, the balance factor of node N is 1. Hence, for the right subtree, if the child of the root node has a balance factor less than or equal to 0 a left rotation is required while a balance factor of 1 requires a right left rotation. This allows the appropriate correction to balance the AVL [sub]Tree to be determined by checking the balance factor of the root and then the balance factor of its first child, provided that the root was initially determined to be unbalanced.

The class created to implement the AVL Tree is called avlTree. As the AVL Tree is based on a BST, the code for the avlTree class was initially a BST that was modified to include the balancing features that define an AVL Tree. The initial BST code was based on two PowerPoint lectures obtained from taking Computer Science courses at the University of Akron: U3-M5 Binary Trees from Computer Science II and U1 ds05 Binary Trees from Data Structures. To implement the AVL Tree features, the U1 ds13 Advanced Tree Structures lecture from Data Structures was referenced, as well as various other online resources that discussed the implementation and functionality of AVL Trees. Additionally, the Data Structure Visualizations[1] website provided by David Galles, an associate professor for the Computer Science department at the University of San Francisco, was useful for visualizing AVL Tree operations and checking the validity of the created avlTree class.

The base of the AVL Tree is a private struct called avlNode which represents the nodes of the AVL Tree. The contents of the avlNode struct are shown below in Figure 8.

```
struct avlNode{
    int element;                                                    // data value
    int nHeight;                                                    // stores the height of the node
    avlNode* lc;                                                    // ptr to left child
    avlNode* rc;                                                    // ptr to right child

    // constructor to create avlNodes
    // Used for root node creation; Default parameters are values for node inserted into root
    avlNode(int element_ = 0, int nHeight_ = 0, avlNode* rc_ = nullptr, avlNode* lc_ = nullptr){
        element = element_;
        nHeight = nHeight_;
        lc = lc_;
        rc = rc_;
    }

    // destructor to destroy avlNodes
    ~avlNode(){
        // delete left and right pointers
        delete lc;
        delete rc;
    }
};
```

Figure 8: Contents of the avlNode struct in the avlTree class.

As shown above, each avlNode contains 2 integer variables and 2 avlNode pointers. The first integer variable is called element and is integer value held by the avlNode. If the avlTree class were to be templated to support custom data types, this would be replaced with the type that the

[1] https://www.cs.usfca.edu/~galles/visualization/AVLTree.html

AVL Tree would store. The second integer variable corresponds to the height of the node within the AVL Tree, which much be stored in each node so that balance factors can be calculated. The two avlNode pointers correspond to the 2 children that a given node can point to- a left child containing a value less than the node and a right child containing a value greater than or equal to it. A default constructor and destructor are also provided, with the default constructor having default arguments that simplify the creation of the root node of the entire AVL Tree. The avlTree class itself contains a single private member variable that is an avlNode pointer called root which points to the root of the AVL Tree. Otherwise, the avlTree class consists of the member functions that serve as the interface to the class and allow the manipulation of the avlTree and the avlNodes that it contains.

The behavior of the avlTree class follows that expected of a traditional AVL Tree. Upon insertion or deletion of an element, the modified node is checked for imbalance as well as every node on the path taken to reach it from the root node. The height of the AVL Tree implemented by the avlTree class follows the convention taught in class that an empty BST has a height of -1 and a BST consisting of just a root node a height of 0. Thus, the height of a node corresponds to how far removed it is from the root node. When performing a deletion, the maximum value on the root node's left subtree is selected to become the new root node instead of the minimum value on the root node's right subtree. This was how deletion was implemented by the Data Structure Visualizations tool and made debugging and testing the class possible. Finally, the avlTree class does not allow duplicate values to be stored in the AVL Tree. The logic preventing this can be easily removed, but the AVL may not balance properly when one or more of the repeated values are removed due to the implementation of delete as discussed later.

Code Analysis:
For the avlTree class, an empty avlTree is created with the default constructor. The result of this is that the AVL Tree's root pointer is set to nullptr. When the first node in the AVL Tree is created, the root node is set to point to the new node and its height is updated from -1 to 0. As the root node is not an actual avlNode, the value '-1' is not stored in it, but rather is the value returned by the member function responsible for returning the height of a node within the AVL Tree. Regarding the member functions in the avlTree class, the key functions are those responsible for searching, inserting, and deleting values, as well as the functions used to return and update the height of nodes, get a node's balance factor, rebalance a node, and to perform a rotation on a node. The following section explains each of these functions briefly, as well as determines the time complexity of each function with respect to the problem size N.

Analysis of the search for element function: find

```
bool avlTree::find(int element_){
    // tmpPtr to traverse AVLtree
    avlNode* tempPtr = root;
    while(tempPtr){
        // if value found return true
        if(tempPtr->element == element_){
                    return true;
        }
        // if value < root, know it must be in left subtree by BST property
        else if(element_ < tempPtr->element){
            tempPtr = tempPtr->lc;
        }
        else if(element_ > tempPtr->element){
        // if greater, must be in right subtree by BST property
            tempPtr = tempPtr->rc;
        }
    }
    // return false if not found
    return false;
}
```

Figure 9: Code for find function.

The find function accepts an integer argument that it attempts to find in the AVL Tree. The find function works by starting at the root node and checking whether it is equal to the desired value. If the value is equal, the function returns true. Otherwise, the function recursively calls itself until on the right of left child of the node until the value is found or a node with no children is encountered (meaning that the element is not in the AVL Tree). As this function runs continuously until the value is found or a nullptr is encountered, the worst case scenario is that the value is not found and is only stopped once reaching a node at the very bottom of the structure. Thus, the runtime of the find function is O(H), where H is the height of the binary Tree. Knowing that each level of the AVL Tree can store up to 2^H nodes, the height of the AVL Tree will be approximately log_2(N), so the runtime of the find function logarithmic => O(LOG(N)).

Analysis of the height retrieval function: getHeight

```
int avlTree::getHeight(avlNode *&parentPtr){
    // if nullptr, nHeight = -1
    if(parentPtr == nullptr)
        return -1;

    else
        // otherwise return height stored in specified node
        return parentPtr->nHeight;
}
```

Figure 10: Code for getHeight function.

The getHeight member function is a simple member function that accepts a pointer to an avlNode and returns the height of the node. As the AVL Tree contains nullptrs, these values must be carefully accessed to prevent program crashes. Hence, this function returns -1 if the pointer passed to it is a nullptr. This is how the empty avlTree can return a height of -1 when asked for it height. Otherwise, the height stored in the node is returned if the pointer points to a valid avlNode. As this function only checks equality and returns a value stored in struct that it is passed a direct pointer to, this function executes in constant time => O(1).

Analysis of the height computation function: updateHeight

```
void avlTree::updateHeight(avlNode *&parentPtr){
    // set new height to the max of the subtrees + 1
    parentPtr->nHeight = maxof(getHeight(parentPtr->lc), getHeight(parentPtr->rc)) + 1;
}
```

Figure 11: Code for updateHeight function.

The updateHeight function is called whenever a node is inserted or deleted in the AVL Tree. The function sets the height variable stored in the avlNode that is passed to it with the maximum value of the height of its subtrees. A value of 1 is added to this result to account for nullptrs having a height of -1. This function calls the getHeight() function on the node's 2 children and determines the maximum value. As this function only compares the result of these 2 calls and adds 1 in order to update the height of the node it was called on, it executes in constant time as well => O(1).

Analysis of the balance factor function: getBalanceFactor

```
int avlTree::getBalanceFactor(avlNode *&parentPtr){
    return (getHeight(parentPtr->lc) - getHeight(parentPtr->rc));
}
```

Figure 12: Code for getBalanceFactor function.

The getBalanceFactor function is a simple member function that returns the difference in heights between the 2 children of the node whose pointer it is passed. This function replaces the multitude of different height checking statements with a single function for cleaner code and easier testing and debugging. As this function returns the difference between the height of two nodes obtained using the getHeight function, it executes in constant time => O(1).

Analysis of the rotation functions:
singleLeftRotation, singleRightRotation, leftRightRotation and rightLeftRotation

The avlTree class has 4 member functions that perform rotations in order to correct height imbalances on nodes. The code for the singleLeftRotation and singleRightRotation functions are identical except for the pointers that they rewire. The singleRightRotation function is shown below as an example.

```
void avlTree::singleRightRotation(avlNode *&parentPtr){
    // get left child of parentPtr
    avlNode* tempLC = parentPtr->lc;
    // get right child of left child of parentPtr
    avlNode* tempLCRC = tempLC->rc;
    // set right child of left child of parentPtr to parentPtr
    tempLC->rc = parentPtr;
    // copy rhs of tempLc to lhs of new rightchild
    tempLC->rc->lc = tempLCRC;
    // update heights
    updateHeight(parentPtr);
    updateHeight(tempLC);
    // new parentPtr = tempLC == root of balanced subtree
    parentPtr = tempLC;
}
```

Figure 13: Code for singleRightRotation function.

As shown, the rotation just swaps the pointers to nodes in the unbalanced subtree in order to balance it. After the pointers are swapped, the heights of the 2 moved nodes are modified to stay current. As this function only reassigns pointers and updates heights, it and the singleLeftRotation function both run in constant time => O(1).

The other 2 balancing functions are the leftRightRotation and the rightLeftRotation functions, which use a combination of the singleLeftRotation and singleRightRotation functions to rotate the unbalance subtree. Only the code for the leftRightRotation function is shown as the rightLeftRotation function mirrors its format to execute its own rotation.

```
void avlTree::leftRightRotation(avlNode *&parentPtr){
    // single left rotation around left child of parentPtr
    singleLeftRotation(parentPtr->lc);
    // single right rotation around parentPtr
    singleRightRotation(parentPtr);
}
```

Figure 14: Code for leftRightRotation function.

As shown, the leftRightRotation function performs a single left rotation around the root node's left child, and then a single right rotation around the root node in order to balance the subtree. As this function makes 2 calls to 2 constant time functions, it executes in constant time => O(1).

Analysis of the rebalance function: rebalance

```
void avlTree::rebalance(int balanceFactor, avlNode *&parentPtr){
    // if balance factor is 2: unbalanced and LHS is larger
    if(balanceFactor == 2)
    {
        //2 possible rotations from here: single right or left right
        // single right if balance factor of parentPtr->lc >= 0, otherwise leftRight
        if(getBalanceFactor(parentPtr->lc) >= 0)
            singleRightRotation(parentPtr);
        else
            leftRightRotation(parentPtr);
    }
    // if balance factor is -2: unbalanced and RHS is larger
    if(balanceFactor == -2){
        //2 possible rotations from here: single left or right left
        // single left if balance factor of parentPtr->lc <= 0, otherwise rightLeft
        if(getBalanceFactor(parentPtr->rc) <= 0)
            singleLeftRotation(parentPtr);
        else
            rightLeftRotation(parentPtr);
    }
}
```

Figure 15: Code for rebalance function.

The rebalance function accepts two parameters, an integer representing the balance factor and the node that the balance factor corresponds to. This function is used in the format "rebalance(getBalanceFactor(avlNodePtr), avlNodePtr)" to conveniently pass the balance factor and the specific node to the rebalance function in a single function call. The rebalance function checks whether the balance factor is equal to 2 or -2 (depending on which subtree is unbalanced) and then determines the appropriate rotation to apply based on the balance factor of one of the parent node's children. As this function only utilizes comparisons and other functions that run in constant time, it must also run in constant time as well => O(1).

Analysis of the insert element function: insertAVLNode

```
void avlTree::insertAVLNode(avlNode *&newNode, avlNode *&parentPtr){
    // place at first nullptr found
    if(parentPtr == nullptr)
        parentPtr = newNode;

    // no duplicate values; return if value already exists
    // duplicates can be enabled by commenting this out, but multiple duplicate values do not delete well. The AVL tree balancing still works
    // but the leftmost maximum value is deleted and not the right most value on the LHS.
    else if(newNode->element == parentPtr->element)
        return;

    // otherwise find appropriate location to insert newNode
    // if newNode value less than parentPtr value, insert on LHS of parentPtr
    else if(newNode->element < parentPtr->element)
        insertAVLNode(newNode, parentPtr->lc);
    // otherwise, insert on RHS of parentPtr
    else if(newNode->element >= parentPtr->element)
            insertAVLNode(newNode, parentPtr->rc);

    // update height of parentPtr
    updateHeight(parentPtr);
    // check for unbalance after inserting new node. Subtrees are unbalanced if the difference between their heights is 2
    rebalance(getBalanceFactor(parentPtr), parentPtr);
}
```

Figure 16: Code for insertAVLNode function.

The insertAVLNode function accepts 2 avlNode pointer parameters. The first avlNode pointer corresponds to a pointer to a newly created node containing the desired value and the second avlNode pointer for the root of the AVL Tree that the node will be inserted into. This function recursively calls itself based on the values of the nodes currently in the AVL Tree in order to properly place the new node. The recursion stops once a valid nullptr has been found, as the nullptr can be overwritten to point to the new node. After insertion, the height of the new node's parent node is updated with the updateHeight function and then checked to see if it must be rebalanced. As this function is called recursively, every node on the path taken to get from the root of the AVL Tree to the newly inserted node has its height updated and its balance checked. This checking starts at the newly inserted node and works its way up, preserving the balance of the entire AVL Tree as a result. Thus, the runtime of the find function is O(P), where P is the length of the path taken to reach the newly inserted node from the root node. In the worst case, the path is equal to the height of the AVL Tree, so it has a time complexity proportional to the height of the AVL Tree, so it is logarithmic => O(LOG(N))

Analysis of the delete element function:

```
// recursive deletion of AVL nodes adapted from geeksforgeeks.org (https://www.geeksforgeeks.org/avl-tree-set-2-deletion/)
// and from Harish R (https://gist.github.com/Harish-R/097688ac7f48bcbadfa5)
avlTree::avlNode* avlTree::deleteAVLNode(int element_, avlNode *&parentPtr){
    // used for getting largest value
    avlNode* tempPtr = nullptr;

    // stop deletion + unwind stack if parentPtr == nullptr
    if(parentPtr == nullptr)
        return nullptr;

    // find element recursively so path to reach it can be retraced back to root to rebalance AVL tree
    // set lc of parentPtr to deletion of its left side if value of lc is less than element. Otherwise,
    // set rc of parenPtr to deletion of its right side
    else if(element_ < parentPtr->element)
        parentPtr->lc = deleteAVLNode(element_, parentPtr->lc);
    else if(element_ > parentPtr->element)
        parentPtr->rc = deleteAVLNode(element_, parentPtr->rc);

    // if 2 children, largest child on LHS becomes new parent
    else if(parentPtr->rc && parentPtr->lc){
        // find largest value on LHS
        tempPtr = getMaxAVLNode(parentPtr->lc);
        // set parentPtr to largest value on its LHS
        parentPtr->element = tempPtr->element;
        // delete max value from LHS by setting LHS of parentPtr equal top the deletion of itself and the max value
        parentPtr->lc = deleteAVLNode(parentPtr->element, parentPtr->lc);
    }
    // if right child does not exist, parent is set to left child
    else if(!parentPtr->rc)
            parentPtr = parentPtr->lc;

    // if left child does not exist, parentPtr set to right child
    else if(!parentPtr->lc)
        parentPtr = parentPtr->rc;

    // if nullptr, return because nullptr's do not have a height
    if(parentPtr == nullptr)
        return parentPtr;

    // update parentPtr height
    updateHeight(parentPtr);
    // now check for imbalance of subtrees
    rebalance(getBalanceFactor(parentPtr), parentPtr);

    // return parentPtr to repeat again
    return parentPtr;
    }
```

Figure 17: Code for deleteAVLNode function.

The code for the deleteAVLNode function is the most complex code in the avlTree class. The idea behind the code for this function is based off the flow of execution of two publicly available AVL Tree classes. The first AVL Tree class that was referenced to create the deleteAVLNode function was by Harish Ravichandran[2] and hosted on GitHub. The second AVL Tree class was described in an article on the website GeeksForGeeks[3] that discussed how to implement deletions in an AVL Tree. Both AVL Tree classes had deletion methods that began at the root node of the AVL Tree and used recursion to determine the element to delete (Ravichandran 2014, GeeksForGeeks 2019). Using recursion is much easier to implement than a pointer based deletion implementation as it provides a mechanism to trace the effect of the deletion up the path from the deleted node to the root node, ensuring that the AVL Tree can remain balanced as a result. Theoretically, the deletion could be implemented only using pointers, but this would require every node to have an additional pointer that points to its parent to allow easy traversal back up the AVL Tree to the root node. Initially, a pointer based deletion method was attempted but scrapped in favor of a deletion process inspired by both GeeksForGeek's and Ravichandran's deletion method.

The idea behind GeeksForGeek's and Ravichandran's deletion method is that the deletion function can be used to return a pointer to a node which allows a pointer to be assigned to the function call (Ravichandran 2014, GeeksForGeeks 2019). Additionally, both deletion functions are responsible for finding the location of the node to delete, performing the deletion, and updating the heights and performing corrective actions on all nodes affected by the deletion. As stated previously, this is accomplished through recursive calls to both deletion methods which allow the path from the deleted node to the root of the AVL Tree to be updated from the bottom up. To start, both deletion functions are passed a pointer to a node and a value that will be recursively searched for and deleted if found. Both deletion methods first check whether the passed pointer is a nullptr and return a nullptr if true to stop the recursion. If the passed pointer is not a nullptr, the deletion function is called again and set equal to subtree of the parent node where the value would reside if it existed. This recursion creates a stack that is unwound to update all the heights along the path from the deleted node up to the root node to keep the AVL Tree balanced. Any node that will be removed can have 0, 1, or 2 children, requiring 3 possible cases to be handled. If the parent has 2 children, the largest child of the parents left child will replace the parent. If the parent has only 1 child, the existing child replaces the parent. For no children, the parent is just replaced with either of its nullptr valued children to delete its value. Thus, this portion of the deletion functions is responsible for finding and deleting the desired node. Following this, another check exists to see if the parent pointer is a nullptr. If it is, the parent pointer value gets returned and the deletion function will start working its way back up the path to the root node of the AVL Tree. Otherwise, the parent node has its height updated and is checked for imbalance. Finally, the pointer to the updated parent node is returned so that the instance of the deletion call which invoked it can continue. This assigns the pointer returned by the completed deletion function to the waiting pointer and allows that pointer to update its height and check for imbalance. This cycle repeats until the root node is reached under both deletion functions.

[2] https://gist.github.com/Harish-R/097688ac7f48bcbadfa5
[3] https://www.geeksforgeeks.org/avl-Tree-set-2-deletion/

The deleteAVLNode function is modeled on the logic of Ravichandran's and GeeksForGeeks' deletion function outlined above. First, the value of the passed pointer is checked to see if it is a nullptr to end the series of recursive calls. If not, the deletion function is called recursively on the appropriate side of the parent node. Another check for a nullptr parent pointer exists before the height of the parent pointer is updated and the node is checked for imbalance. Finally, the fully updated pointer is returned to satisfy the instance of the deleteAVLNode function that invoked it, allowing the deletion to be trace up from the deleted node to the root of the Tree. As for why the deletion of repeated values does not work properly, the deleteAVLNode function will search for the repeated value to remove it. Once found, the function will remove the first instance of the repeated value and not the rightmost instance that should be removed. This results in rebalances that do not match the results generated by the Data Structure Visualizations tool, so the structure is not functioning properly. This cannot be fixed as changing the logic to check for values greater than or equal to the parent node's right child will always result in a series of nullptrs being returned and checks for value equality create infinite loops that the deleteAVLNode function cannot escape. Likewise, checks for passed element and node value equality cannot be used as the deleteAVLNode function cannot distinguish between instances where the target value has been found or a repeated value is being deleted. Thus, the deleteAVLNode function would need to be rewritten in order to properly support the deletion of duplicate values.

Regarding the time complexity of the deletion function, it recursively calls itself until the desired node is found. Once found, it performs the deletion and then traverses back up the path it took up update the heights of all affected nodes and check for imbalances. Due to how the recursion is implemented, the deleteAVLNode function only needs to traverse to the deleted node in order to update all the nodes along the path between itself and the root node. Besides calls to itself, this function only performs checks for equality and calls to functions with constant time complexities, so its time complexity is proportional to the height of the AVL Tree, so it is logarithmic => O(LOG(N))

Testing Methodology:

The code to test the avlTree class was based off the sample project test code posted on Brightspace called "Complexity_Runtime_DS_Example". This code mas modified to include a switch statement that allows the user to select a particular test to perform on the 3 data structures, as well as the code to implement the desired tests. To determine the performance of each data structure, the provided "Complexity_Recorder.hpp" and "Complexity_Timer.hpp" files were used to record the time it took each data structure to complete the designated test with respect to the problem size N at the time. The range of the problem size was defined in terms of a lower bound equal to 100 and an upper bound equal to 25600. Once a test completed, the lower bound of the problem size was doubled and the test would be repeated with the larger problem size. This cycle was repeated until the lower bound of the problem size exceeded the upper bound, at which point the entire test was complete. Each test is performed a set number of times on each data structure. The number of times a test is repeated on a given data structure is defined by the constant "REPETITIONS" and the number of times to repeat all the tests and their repetitions is defined by the constant "NUMBER_OF_TRIALS".

Each data structure was subject to the same 6 tests:

1. The time to insert unsorted data
2. The time to delete unsorted data
3. The time to find unsorted data
4. The time to insert sorted data
5. The time to delete sorted data
6. The time to find sorted data

This method of testing produces results that demonstrate how the big three operations (insert, delete, and search) vary for each structure with respect to the problem size N and the type of input. The goal here is to not only determine which data structure can perform each operation the fastest, but also how the speed of each operation is dependent on whether the data fed into the structure is sorted or unsorted.

Two functions were created in the driver code, responsible for populating the data structures with values to perform each test. Regardless of the test, each data structure was fed the same data in the same order to keep the results as consistent as possible. The first function is called inputGeneration, which is used to create a sequence of non-duplicate values equal to the problem size N. This function is used to generate the input for the insertion and deletion tests. The code for this function is shown below in Figure 18:

```
void inputGeneration(std::vector<int> &vec, int N, bool randomize){
    // fill vector with N unique numbers from 0 to N-1 to generate a sorted list w/ unique values
    for(long long int i = 0; i < N; ++i)
        vec.push_back(i);
    // if randomize == 1, perform a random shuffle on the list
    if(randomize == 1)
        std::random_shuffle(vec.begin(), vec.end());
}
```

Figure 18: Code for inputGeneration function.

As shown above, the inputGeneration function requires a pointer to a vector, an integer, and a boolean value. The first parameter, the pointer to a vector, is the destination where the generated values will be placed. The second parameter is the current problem size. This function iterates a for loop from 0 to (N-1) and fills the vector with the value of the for loop's counter to create N unique values. The 3rd parameter is a boolean value that corresponds to the decision to randomize the data stored in the vector. If enabled, the std::random_shuffle function is used to randomize the order of the values stored in the vector. As a result, this vector is able to store sorted and unsorted input by passing a 0 or 1 respectively for the boolean parameter.

The second function created for the driver code is called findMeGeneration and was used to generate the input for the find tests. The code for the findMeGeneration function is shown below.

```
void findMeGeneration(std::vector<int> &vec, int N){
    // fill vector with N/2 unique numbers from 0 to N-1
    for(long long int i = 0; i < N/2; ++i)
        vec.push_back(std::rand() % N);

}
```

Figure 19: Code for findMeGeneration function.

The findMeGeneration function is like the inputGenerationFunction, as it accepts a pointer to a vector and an integer corresponding to the problem size N. However, the findMeGeneration function fills the vector with N/2 random values generated using the std::rand() function. The output of the rand function is guaranteed to fall within the input range for the problem size by using the modulo operator and the problem size N to restrict the result from 0 to N. Here, duplicate values are possible but unlikely due to the pseudorandomness of the rand function. However, repeated values for the find operation are acceptable as no data is added to or removed from the structure when performing a find operation.

Results:

This research yielded three results. The first result was the recorded run times of the insert, delete, and find operations relative to the problem size N for each data structure. The second result was how the run times of the three functions varied in response to sorted and unsorted input. The third result was then the space complexity for each of the three data structures.

To obtain the run times of the insert, delete, and find operations, the test code was executed on my personal desktop computer which has an Intel i7 9700k @4.8GHz and 32GB of 3200MHz RAM. The parameters for the test code were as follows: the number of repetitions was set to 4, the number of trials 3, the lower bound of the problem size 100, the upper bound of the problem size 25600, and the factor 20.

The runtime results of each test relative to the problem size are shown in the following figure.

Unsorted Insertion Test			
Problem Size (N)	AVL Tree	Skip List	STL List
1000	0.000	0.000	0.000
2000	0.001	0.001	0.000
4000	0.002	0.001	0.000
8000	0.004	0.003	0.001
16000	0.009	0.006	0.002
32000	0.020	0.014	0.003
64000	0.044	0.030	0.006
128000	0.100	0.071	0.013
256000	0.226	0.167	0.026

Sorted Insertion Test			
Problem Size (N)	AVL Tree	Skip List	STL List
1000	0.001	0.000	0.000
2000	0.001	0.000	0.000
4000	0.002	0.001	0.001
8000	0.004	0.002	0.001
16000	0.008	0.003	0.002
32000	0.017	0.006	0.003
64000	0.036	0.013	0.006
128000	0.078	0.025	0.013
256000	0.165	0.052	0.025

Unsorted Deletion Test			
Problem Size (N)	AVL Tree	Skip List	STL List
1000	0.000	0.000	0.003
2000	0.001	0.001	0.015
4000	0.001	0.001	0.059
8000	0.001	0.001	0.246
16000	0.003	0.002	1.060
32000	0.007	0.007	4.276
64000	0.014	0.013	17.116
128000	0.033	0.028	68.653
256000	0.074	0.060	285.059

Sorted Deletion Test			
Problem Size (N)	AVL Tree	Skip List	STL List
1000	0.000	0.000	0.004
2000	0.000	0.001	0.014
4000	0.001	0.000	0.059
8000	0.001	0.001	0.238
16000	0.003	0.002	0.992
32000	0.006	0.004	4.016
64000	0.013	0.008	16.140
128000	0.027	0.018	64.917
256000	0.059	0.037	267.286

Unsorted Find Test			
Problem Size (N)	AVL Tree	Skip List	STL List
1000	0.000	0.000	0.002
2000	0.000	0.000	0.009
4000	0.000	0.000	0.039
8000	0.001	0.001	0.167
16000	0.001	0.002	0.676
32000	0.002	0.005	2.690
64000	0.005	0.013	13.134
128000	0.012	0.028	50.001
256000	0.022	0.056	130.014

Sorted Find Test			
Problem Size (N)	AVL Tree	Skip List	STL List
1000	0.000	0.000	0.002
2000	0.000	0.000	0.009
4000	0.000	0.000	0.038
8000	0.001	0.001	0.163
16000	0.001	0.002	0.669
32000	0.002	0.004	2.643
64000	0.004	0.007	5.674
128000	0.009	0.014	11.533
256000	0.020	0.028	23.820

Figure 20: Results of insertion, deletion, and find tests on sorted and unsorted data for each data structure.

As shown in Figure 20, the AVL Tree had the slowest insertions of the three structures. Conversely, the STL list had the fastest insertions followed by the Skip List. Concerning the time to delete elements, the Skip List was the fastest but was closely followed by the AVL Tree. As the problem size increased, the STL list began to take an incredibly long amount of time. Looking at the results of the search test, the AVL Tree outperformed the other structures by a wide margin. The Skip List has relatively close searching performance to the AVL Tree while the STL list lagged severely behind.

An interesting observation from the data in Figure 20 is how the use of sorted versus unsorted input affected the test results. Sorted input greatly increased the insertion speeds of the AVL Tree and Skip List compared to unsorted input. The STL list saw no change here. Looking at the deletion test, sorted input once again improved the performance of the AVL Tree and Skip List, with even the STL list seeing a slight improvement for larger problem sizes. Finally, the use of sorted input had little effect on the find test when performed on the AVL Tree. However, sorted input resulted in find test results that were approximately 2 times faster for the Skip List and 5.5 times faster for the STL list. From these results, it is evident that the use of sorted input vs unsorted input can greatly impact the runtime of some, or all of the operations performed on each data structure. Therefore, the type of input that will be used constitutes another important factor to consider when selecting a particular data structure to address a specific problem.

Plotting the runtime for each test with respect to the problem size yields the plots shown in figures 21 through 26.

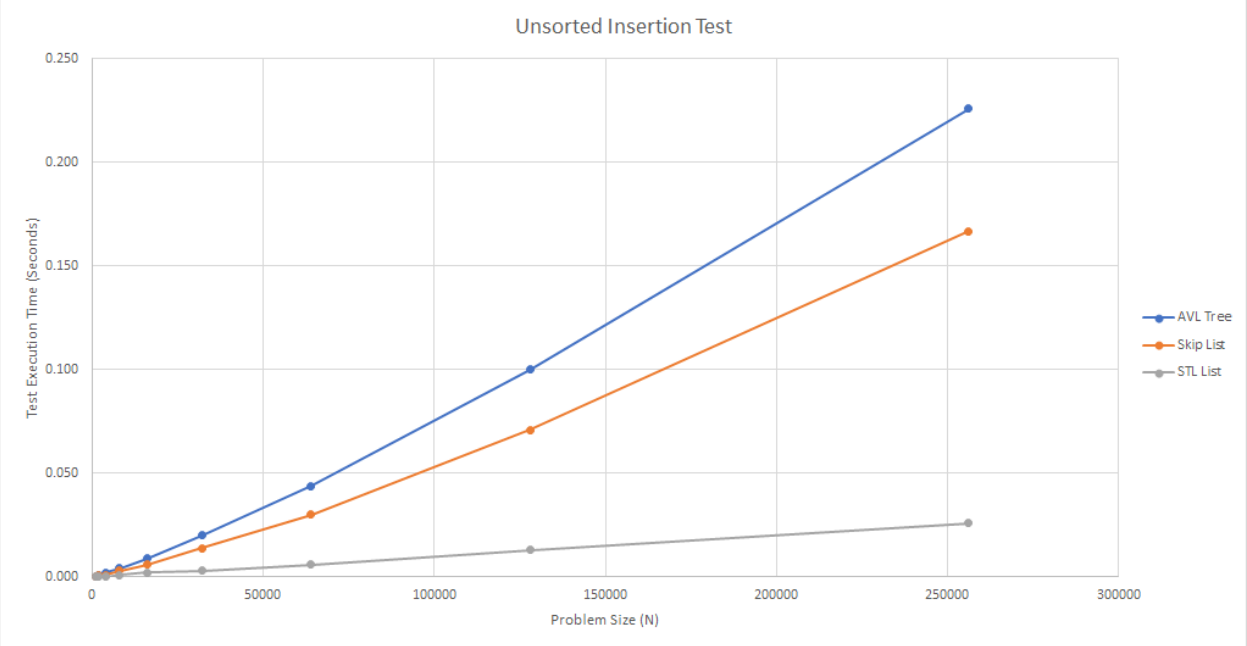

Figure 21: Plot of the runtime of the unsorted insertion test for all 3 data structures as a function of the problem size N.

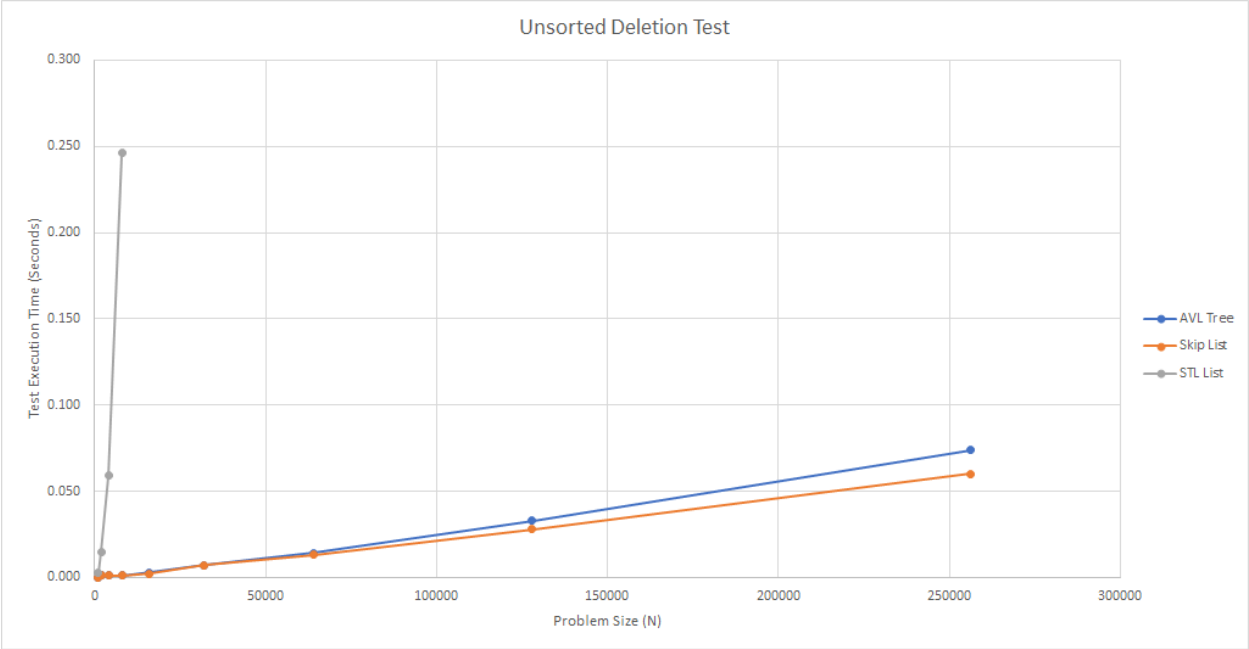

Figure 22: Plot of the runtime of the unsorted deletion test for all 3 data structures as a function of the problem size N. The extreme runtime vales for the STL list are not plotted.

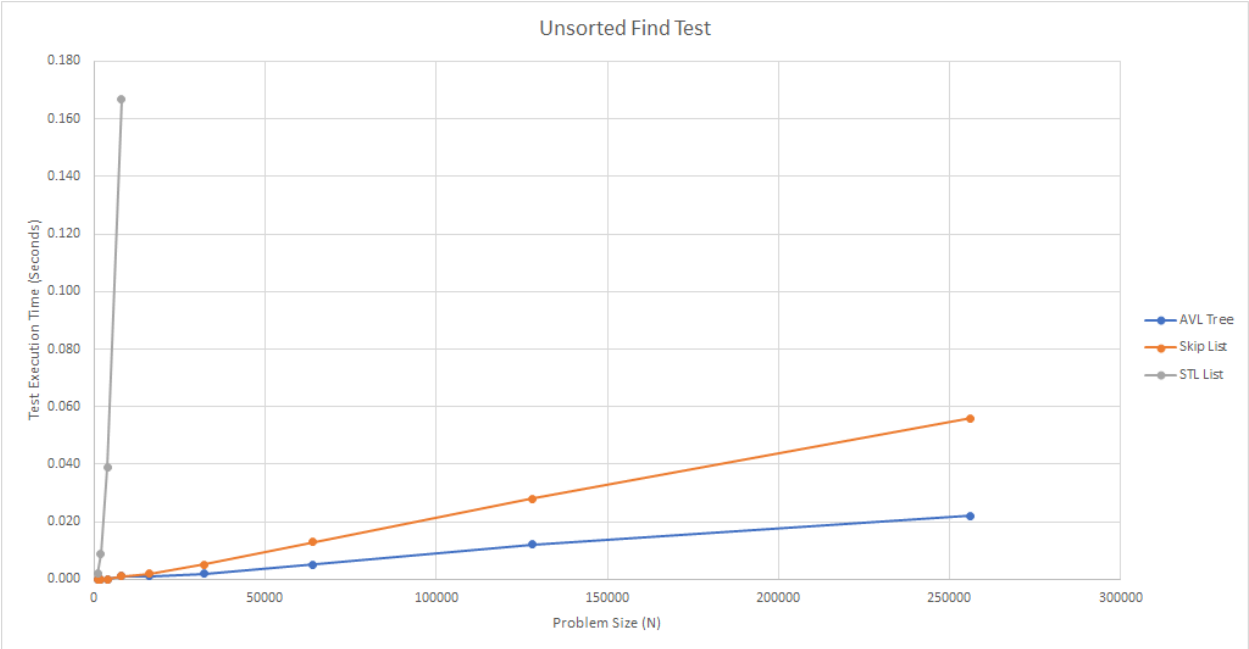

Figure 23: Plot of the runtime of the unsorted find test for all 3 data structures as a function of the problem size N. The extreme runtime vales for the STL list are not plotted.

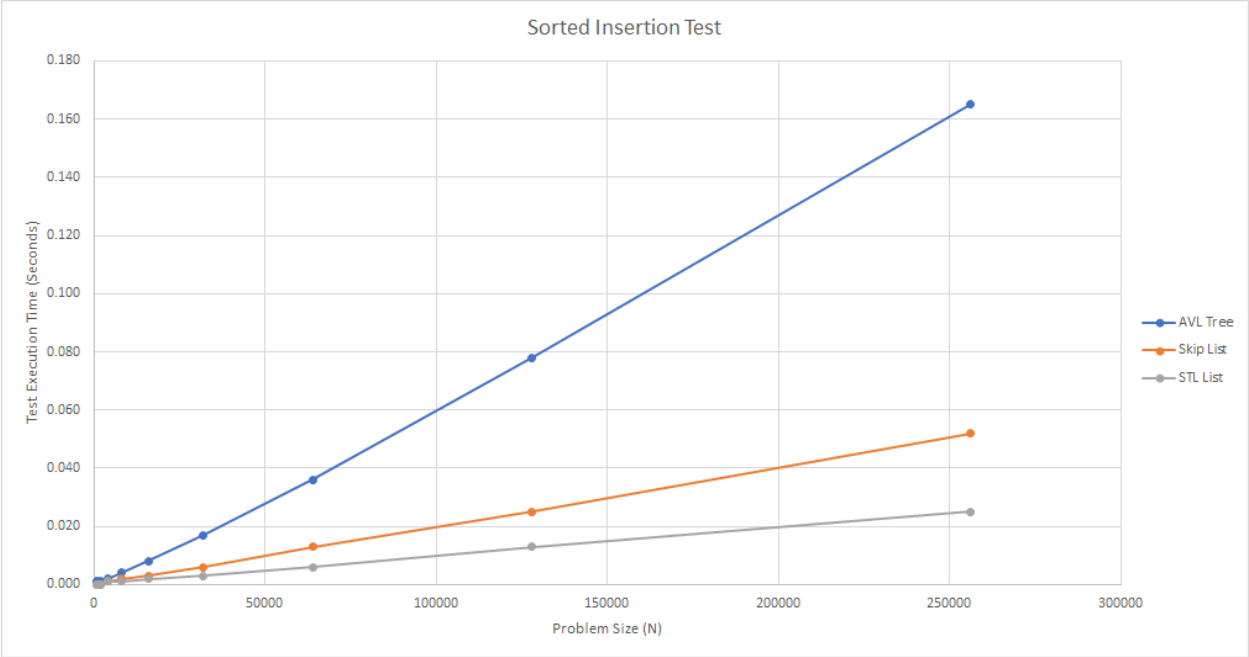

Figure 24: Plot of the runtime of the sorted insertion test for all 3 data structures as a function of the problem size N.

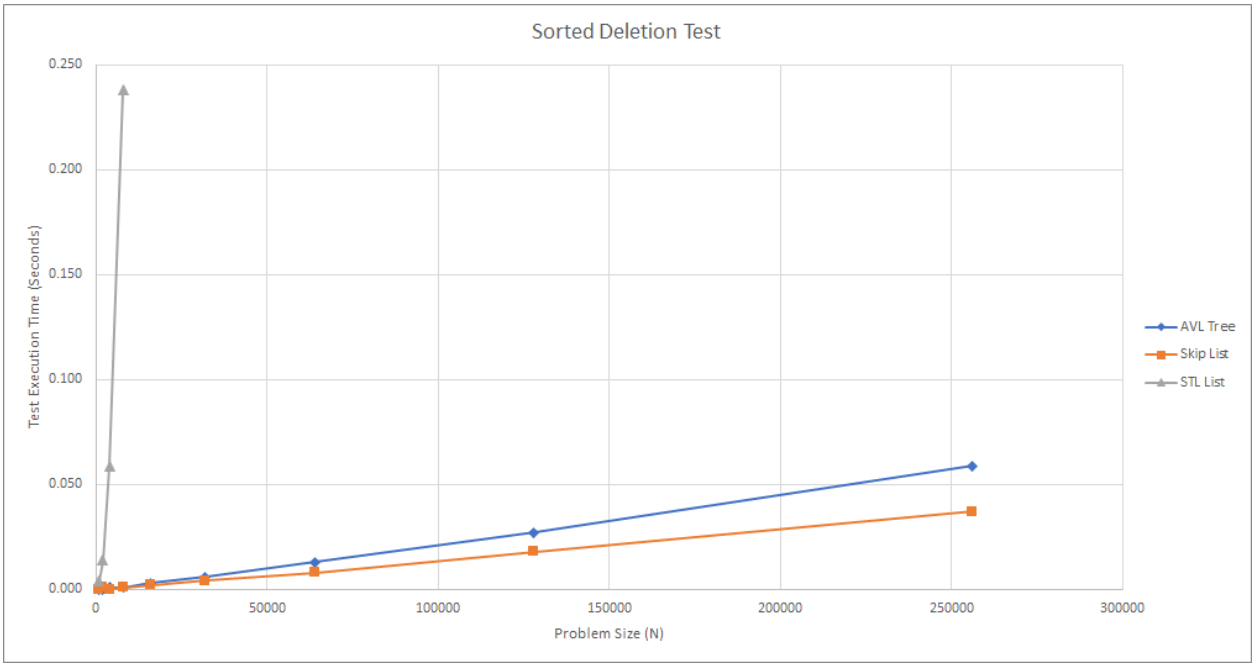

Figure 25: Plot of the runtime of the sorted deletion test for all 3 data structures as a function of the problem size N. The extreme runtime vales for the STL list are not plotted.

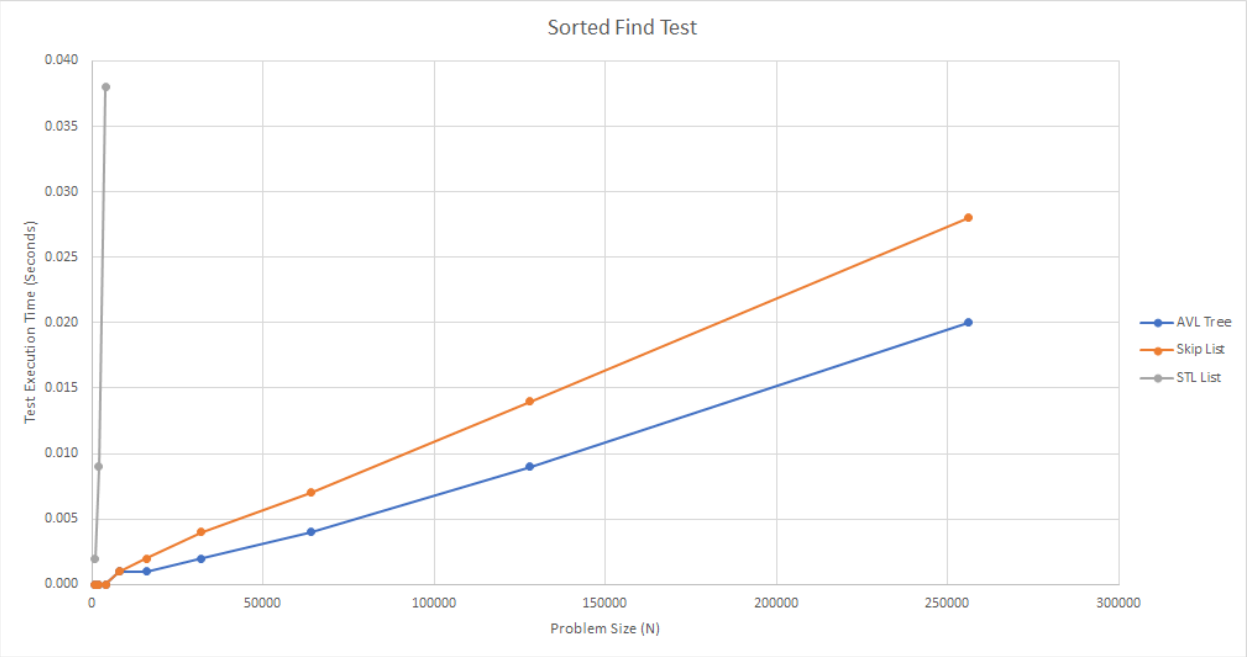

Figure 26: Plot of the runtime of the sorted find test for all 3 data structures as a function of the problem size N. The extreme runtime vales for the STL list are not plotted.

As shown in the above plots, the STL list had the fastest insertion times while the AVL Tree's insertion times were the slowest. This is likely due to the extra overhead occurred by the AVL Tree having to check that an insertion does not upset its balance and the cost of performing rebalancing operations. The STL list has exponentially slow deletion runtimes while the Skip List had quick deletion runtimes. The AVL Tree lagged slightly behind the Skip List for both deletions and insertions. Finally, concerning searching, the AVL Tree is the fastest followed by the Skip List. The STL list is once again exponentially slower here.

With the run times of the insert, delete, and find operations shown as a function of the problem size N and the effect of sorted vs unsorted input able to be observed, the final result of this research is the space complexity associated with each data structure. To begin, the AVL Tree is just a modified Binary Search Tree so the same method of determining the overhead fraction can be applied. The AVL Tree consists of N nodes that all contain 2 integer variables and 2 pointers as shown in Figure 8. Thus, the formula to calculate the overhead is (2P+2D) = 2P. Assuming that each integer, D, is 4 bytes and each pointer 8, the overhead fraction becomes (2*8)/(2*4+2*8) = 2/3. To compare the AVL Tree to an array, assume G is the maximum number of elements that can be stored in an array. The amount of space to implement an AVL Tree with n nodes is n(2P+2D) while the array requires D*G space. Setting these 2 equations equal yields n(2P+2D) = D*G. Solving for n yields the value at which the AVL Tree is more efficient than the array, which occurs for values of n less than (D*G)/(2P+2D). In instances where n exceeded this fraction, storing the data in the array is more space effective, but the fast deletion and search provided by the AVL Tree may outweigh the cost of being less space efficient. Finally, as the

AVL Tree allocates a single node for each element that it stores, the size of the AVL Tree grows proportionally to the size of the problem, so the space complexity of the AVL Tree is O(N).

Concerning the space complexity of the Skip List, it is accepted as having a space complexity of O(N) (Garg, 2012). This trait is also shared by the STL list, as the number of nodes that it contains grows proportionally to the size of the problem, so its memory usage must as well. Hence, the STL list also has a space complexity of O(N). Finally, the overhead of the STL list can be determined by the overhead equation for linked lists. As the STL list is a doubly linked list, it contains 2 pointers per node and 1 data element. Assuming that the size of the data is A and the size of the pointers is B, the total space required to implement the STL list is n(2B + A). Now, assuming that an array stores X elements of the same type as A, the array requires XA space. Equating the two equations yields n(2B+A)=X*A. Solving for n yields the number of elements at which the STL list is more space efficient than its array based implementation, which is any value of n less than (X*A)/(2B+A).

Conclusion:

In conclusion, this experiment demonstrated the affect that sorted versus unsorted input can have on the performance of a data structure. As shown by the table in Figure 20, the use of unsorted data in all 3 test cases produced slower execution times compared to the use of sorted data for corresponding problem sizes. The STL List experienced a minimal difference with insertions and deletions between the sorted and unsorted input, but the speed of the search was shown to be approximately 5.5 times faster with sorted input than unsorted. Concerning the Skip List, sorted insertions were executed nearly 3 times faster than unsorted insertions for larger problem sizes, while sorted deletion were nearly 1.5 times faster than unsorted deletions. Similarly, sorted searches performed approximately 2 times faster than unsorted searches. Finally, the AVL Tree saw a slight performance improvement when sorted data was used over unsorted data, with sorted insertions performing approximately 1.3 times faster than unsorted insertions and sorted deletions performing approximately 1.2 times faster than unsorted deletions for larger problem sizes. The speed of the AVL Tree's search was largely unaffected using sorted input versus unsorted, which makes sense as the AVL Tree is designed for optimized searches.

Comparing the 3 data structures with each other, the AVL Tree had the fastest sorted and unsorted searches and the sorted Skip List was approximately 1.5 times slower. However, the Skip List had faster insertion and deletion speeds than the AVL Tree for both sorted and unsorted input. Similarly, the STL List had the fastest insertions regardless of the type of input, but its deletion speed and search speed were lacking. This suggests that each data structure is best suited to the type of operations that will be performed on the data that is contained. For instance, if data is just to be stored in bulk and infrequently accessed, the STL list will provide the fastest insertion times. Likewise, if the information is going to be repeatedly searched for after insertion, the AVL Tree would be a good choice due to its fast search times. However, even though the

Skip List has a slightly slower search speed with sorted data, its much faster insertion and deletion speed compared to the AVL Tree may be desirable and worth the slight penalty to search times. Thus, the Skip List is the best all-around structure for both sorted and unsorted input due to having fast insertion, deletion, and search times. The use of sorted data only decreases the amount of time that each operation takes and brings the insertion and search speeds of the Skip List closer to the structures that excel in each operation, being the STL List and AVL Tree, respectively.

Regarding the space complexity of all 3 data structures, the space required to implement each data structures grows linearly with respect to the problem size, so each data structure has a space complexity of O(N). The AVL Tree has an overhead fraction of 2/3 which means it is not very space efficient, but this is a necessary tradeoff for problems requiring a solution with fast search times. Compared to array based implementations, the AVL Tree, Skip List, and STL List all have extra overhead due to their extensive use of pointers, which could result in them becoming less space efficient than their array equivalents at very small problem sizes. Once again though, for applications that require any of the data structures due to their performance with one or more operation, they become a tradeoff between performance and the space required to implement them.

As shown by the results of this research, a variety of factors need to be considered when selecting a data structure to address a specific problem. This selection presents a tradeoff between the space required to implement the data structure and the performance promised as a result, with the application of the data structure deciding which is the most important factor. Knowing the type of input that the data structure will handle, whether it is sorted or unsorted, presents another factor that must be considered when selecting a data structure as it largely impacts the performance of the structure, with this research showing that sorted input either results in a large boost in performance or no change at all.

References:

Crissey Renwald, W. (n.d.). *U3-M5 Binary Trees.pptx* [PDF]. Akron: University of Akron Department of Computer Science.

Obtained from taking the CSII Course

Crissey Renwald, W. (n.d.). *U1 ds05 Binary Trees* [PDF]. Akron: University of Akron Department of Computer Science.

Obtained from taking the Data Structures Course

Crissey Renwald, W. (n.d.). *U1 ds13 Advanced Tree Structures* [PDF]. Akron: University of Akron Department of Computer Science.

Obtained from taking the Data Structures Course

Galles, D. (2011). AVL Tree. Retrieved March 13, 2021, from https://www.cs.usfca.edu/~galles/visualization/AVLTree.html

Garg, D., Dr. (2012, June 7). *Skip Lists* [PDF]. Gdeepak.com.

GeeksForGeeks. (2019, August 07). AVL Tree: Set 2 (Deletion). Retrieved March 12, 2021, from https://www.geeksforgeeks.org/avl-Tree-set-2-deletion/

Morris, J. (1998). 8.3 AVL Trees. Retrieved March 13, 2021, from https://www.cs.auckland.ac.nz/software/AlgAnim/AVL.html

Ravichandran, H. (2014, October 18). AVL Tree implementation in C++. self balancing Tree. Retrieved March 12, 2021, from https://gist.github.com/Harish-R/097688ac7f48bcbadfa5