## Exercise 1

## Exercise 2

1. Since each node of the h-1 layer can only have one only child, we can use the number of nodes in that layer to find the max number of only childs. We also know that an only child cannot have a child itself, because it would break the AVL tree. Thus, we get at max 2^(h-1) only childs in any AVL tree. At the same time, we get a formula for the total number of nodes. Which is the number of only childs plus the entire tree above them.

Inserting this into LR(T) we get:

H=1 is the base case since the tree must be nonempty and the limit of the equation is 1/3. This means that .

1. Sdf

## Exercise 3

## Exercise 4

We added a public method to the AvlTree class and made the following implementation.

avl\_tree.tpp

template<typename Comparable>

bool AvlTree<Comparable>::verify() {

*return* verify(root);

}

avl\_tree.h

bool verify(AvlNode \*&t)

    {

*// We reached the bottom of a branch and returns.*

*// No further action is taken since the tree has already been verified including this branch*

*if* (t == nullptr) *// O(1)*

        {

*return* true;

        }

*// If the height on t is not one higher than one of the branches, then there is a mistake*

*if* (max(height(t->left), height(t->right)) + 1 != t->height) *// O(2)*

        {

*return* false;

        }

*// We check if the difference in height of the two subtrees are within 1*

*if* (height(t->left) - height(t->right) > ALLOWED\_IMBALANCE) *// O(2)*

*return* false;

*else* *if* (height(t->right) - height(t->left) > ALLOWED\_IMBALANCE) *// O(2)*

*return* false;

*// Verify the left and right child*

*if* (verify(t->left) && verify(t->right)) *// O(N/2) + O(N/2)*

        {

*return* true;

        }

*// If the two verifies above fail, the tree is not correct*

*return* false;

    }

## Exercise 5



|  |  |
| --- | --- |
| 0 | 28 |
| 1 | 15 |
| 2 |  |
| 3 | 17, 10 |
| 4 |  |
| 5 | 5, 19, 33, 12 |
| 6 | 20 |

b.

|  |  |
| --- | --- |
| 0 | 28 |
| 1 | 15 |
| 2 |  |
| 3 | 17 |
| 4 | 10 |
| 5 | 5 |
| 6 | 19 |
| 7 | 20 |
| 8 | 33 |
| 9 | 12 |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |
| 16 |  |

c.

|  |  |
| --- | --- |
| 0 | 28 |
| 1 | 15 |
| 2 |  |
| 3 | 17 |
| 4 | 10 |
| 5 | 5 |
| 6 | 19 |
| 7 | 20 |
| 8 |  |
| 9 | 33 |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 | 12 |
| 15 |  |
| 16 |  |

## Exercise 6

We compare each element of each left to the current max and return the max element at the end.

As stated in the code, the worst case complexity is O(N+M) since we will only reach the inner for-loop exactly N times and will look at each list exactly M times.

hash\_table\_chaining.tpp

template<typename HashedObj>

HashedObj HashTable<HashedObj>::findMax() {

*//Since we don't know where the first element is, we use a boolean to tell if we have found the first element*

    HashedObj current\_max;

    bool first\_element\_found = false;

*//We look at each list in the vector*

*for* (int i = 0; i < theLists.size(); i++)

*// O(M), where M is the size of our hashtable*

    {

*for* (typename list<HashedObj>::iterator it = theLists[i].begin(); it != theLists[i].end(); ++it)

*// O(N), where N is the number of elements we have inserted to the entire hash table.*

*// Therefore it is not O(NM) but O(N+M) since we have exactly N elements and will look at each only once.*

        {

*//For each element in each list we compare the current max to the current value*

*if*(first\_element\_found) *// O(1)*

            {

                current\_max = max(current\_max, \*it); *// O(1), max() is O(1)*

            } *else*{

                current\_max = \*it; *// O(1)*

                first\_element\_found = true; *// O(1)*

            }

        }

    }

*return* current\_max; *// O(1)*

}

To improve the findMax() we would sort the individual lists each time we insert a new element. This would make it easier to find the max element in each list thus reducing the worst case complexity to O(M). This would however increase the complexity of insert.