Exercise 1

The video describing how the insertion of the different keys into an avl tree would happen can be seen in the file “Exercise1_AVlTree.mp4”,

Exercise 2

a. 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:

$$LR(T) = \frac{2^{h-1}}{2^h - 1 + 2^{h-1}}$$

$$h = 1, \frac{1}{2-1+1} = \frac{1}{2}$$

$$h = 2, \frac{2}{4-1+2} = \frac{2}{5}$$

$$h = 3, \frac{4}{8-1+4} = \frac{4}{11}$$

$$\lim_{h\to\infty} \frac{2^{h-1}}{2^h - 1 + 2^{h-1}} = \frac{1}{3}$$

H=1 is the base case since the tree must be nonempty and the limit of the equation is 1/3. This means that $LR(T) \leq 1/2$.

b. Since t is not an AVL tree the balance property is not necessarily complied with. Thus, we cant insure that t will have a depth of log(n).

Exercise 3

We implemented a parent variable to each node and tried to implement the iterators but could not get the files to link properly. Also the behavior of the iterators was not clear to us.

The parent was added in binary_search_tree.h and the insert method was changed to fit.

```
private:
  struct BinaryNode {
      Object element;
      BinaryNode *left;
      BinaryNode *right;
      BinaryNode *parent;
```

```
      BinaryNode(const Object& theElement, BinaryNode* lt, BinaryNode* rt,
BinaryNode* pt) :
          element {theElement}, left {lt}, right {rt}, parent {pt} { }
    };

    BinaryNode *root;

    /**
     * Internal method to insert into a subtree.
     * x is the item to insert.
     * t is the node that roots the subtree.
     * Set the new root of the subtree.
     */
    void insert(const Object& x, BinaryNode* &t) {
        if (t == nullptr)
            t = new BinaryNode{x, nullptr, nullptr, nullptr};
        else {
            if (x < t->element)
                if(t->left != nullptr)
                {
                    insert(x, t->left);
                } else
                {
                    std::cout << "test" << std::endl;
                    t->left = new BinaryNode{x, nullptr, nullptr, t};
                }
            else if (t->element < x)
                if(t->right != nullptr)
                {
                    insert(x, t->right);
                } else
                {
                    t->right = new BinaryNode{x, nullptr, nullptr, t};
                }
            else; // Duplicate; do nothing
        }
    }
```

The Set methods were also included in binary_search_tree.h

```
#include "set_itr.h"

    iterator Sinsert(const Object &x)
    {
        insert(x);
        return Sfind(x);
```

```
    }
    iterator Sfind(const Object &x) const
    {
        iterator out;
        BinaryNode *node = root;
        while (node != nullptr)
        {
            if (node->element == x)
            {
                return out{node};
            }
            else if (x < node->element)
            {
                node = node->left;
            }
            else if (x > node->element)
            {
                node = node->right;
            }
        }
        return iterator(nullptr);
    }

    iterator Serase(iterator &itr)
    {
        iterator out {itr->parent};
        remove(itr->element);
        return out;
    }
```

The iterators were included in set_itr.h

```
class iterator
{
private:
    BinaryNode *current;

public:
    friend class BinarySearchTree<Object>;

    // Public constructor for iterator.
    iterator()
    {
        current = nullptr;
    }
```

```cpp
    iterator(BinaryNode *p)
    {
        current = p;
    }

    const Object &operator*()
    {
        return current->element;
    }

    bool operator==(const iterator &rhs)
    {
        return current == rhs.current;
    }

    bool operator!=(const iterator &rhs)
    {
        return !(*this == rhs);
    }

    /* Pre-in/decrement. */
    iterator &operator++()
    {
        current = current->left;
        return *this;
    }

    iterator &operator--()
    {
        current = current->right;
        return *this;
    }


    /* Post-in/decrement. */
    iterator operator++(int)
    {
        iterator old = *this;
        ++(*this);
        return old;
    }

    iterator operator--(int)
    {
        iterator old = *this;
        --(*this);
        return old;
```

```
    }


};
```

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

a.

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

$$\lambda = 9/7$$

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	

$$\lambda = 9/17$$

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	

$$\lambda = 9/17$$

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.