Portfolio

CS 234: Data Structures

Dr. Havill

By:

Ryan Anderson

Made With:

LaTeX

May 8, 2017

Contents

1 Introduction

Throughout this semester, I have made an impressive amount of progress in my programming abilities and analysis. Learning about various types of data structures has added to the tools I can use when programming in the future. In addition to the actual data structures, the ideas and skills learned alongside of them will surely carry me far in my computer science career. I learned a lot about the subjects contained in these nine competencies. The competencies include analysis of algorithms, linear data structures, priority queues and binary heaps, graphs, hash tables, binary search trees, object-oriented programming, and professional practice. Using class experiences, projects, and exams, I will demonstrate my mastery of each competency.

2 Analysis of Algorithms

This competency involves proving the correctness of algorithms using preconditions, postconditions, and loop invariants, along with analyzing the asymptotic time complexity of both iterative and recursive algorithms. With a combination of my original understanding and improvement in these topics, I believe that I achieved mastery with distinction for this competency. The topics included in this competency were relevant throughout the whole semester, so I had to stay sharp on remembering how to analyze algorithms.

2.1 Proving Correctness of Algorithms

Proving the correctness of an algorithm is the best way to understand how it works. Proving an algorithm requires multiple steps: finding the loop invariant, proving the loop invariant is correct using an initialization and maintenance step, and finding the termination condition to prove the algorithm's correctness. On Exam 1, the first problem dealt with the loop invariant for a simple function. My answer, which I scored 20/20 on, is shown below:

```
double(A, n)
    for i = 1 to n
        A[i] = A[i] * 2
```

a) Write a loop invariant for the `for` loop in this function.
Before each iteration `i`, the values from `A[1...i-1]` are double the values in the original array.

b) Prove that your loop invariant is correct.
Initialization: Before the first iteration, the values from `A[1...0]` have been doubled. This is true because we can say anything about this subarray since it goes backwards, and therefore is empty.
Maintenance: During iteration `i`, the value at `A[i]` is doubled, and our loop invariant can be extended to:
The values from `A[1...i]` are double the values in the original array.
This is the loop invariant for iteration `i+1`, so our loop invariant must be correct.

c) What is the termination condition of the loop invariant?
After the final iteration of the `for` loop, the values from `A[1..(n+1)-1]` or `A[1..n]` are double the values from the original array.

d) How does the termination condition of the loop invariant prove that the function is correct?
The termination condition says that all of the values in the array have been doubled,

which is the goal of the function.

2.2 Asymptotic Time Complexity

When writing extensive programs, it is important for the program to run efficiently. Understanding run times of algorithms helps us to write more efficient algorithms that will make our programs run faster. It is important to understand how to find time complexities of algorithms, as it helps programmers write code that is efficient.

One example that shows my expertise in time complexity is first question of Project 1. The problem was to write a linear search method, prove its loop invariant, and then analyze the run-time of the function. Only the pseudocode and time complexity sections of this question are included.

```
LinearSearch(A, v)
    for i = 1 to A.length
        if A[i] = v
            return i
    return NIL
```

Best Case: When $A[1] = v$. $T(n) = c_1 + c_2 + c_3$, $\Theta(n) = n^0$ or 1, where $n =$ A.length.

Average Case: If v is equally likely to be found each iteration, then the search will require:

$$\frac{1}{n+1}\sum_{i=1}^{n+1} i = \frac{1}{n+1} * \frac{(n+1)(n+2)}{2} = \frac{n+2}{2} = \Theta(n).$$

Worst Case: The item is never found, and therefore the loop runs through until the end, so it has $n+1 = \Theta(n)$ iterations.

Learning to prove recurrences was instrumental in getting a true understanding of time complexities. Question three from Project 2 asked to find a tight upper bound for $T(n)$ and to prove a variety of recurrences using induction. I included part e from this question, which I believe was one of the more complex recurrences.

$$\begin{aligned} T(n) &= T(n-1) + lg(n) \\ &= T(n-2) + lg(n-1) + lg(n) \\ &... \\ &= T(n-i) + \Sigma_{j=0}^{i-1} lg(n-j) \end{aligned}$$

Scratch Work:

$$\begin{aligned} n - i &= 1 \\ i &= n-1 \\ \Rightarrow T(n) &= T(n-n+1) + \Sigma_{j=0}^{n-2} lg(n-j) \\ &= a + \Sigma_{j=0}^{n-2} lg(n-j) \\ &= a + lg(n!) \end{aligned}$$

Prove that $T(n) = T(n-1) + lg(n)$ by strong induction on n.
Proof: For the base case, let $n = 1$. So we get $T(1) = a + lg(1!) = a$. For the induction hypothesis, assume that $T(k) = a + lg(k!)$ for all $k = 1, 2, ..n-1$. For the induction step:

$$\begin{aligned} T(n) &= T(n-1) + lg(n) \\ &= a + lg((n-1)!) + lg(n) \\ &= a + lg((n-1)!(n)) \\ &= a + lg(n!)\blacksquare \end{aligned}$$

3 Linear Data Structures

A linear data structure is one whose elements form a sequence. This semester, the linear data structures we used most often were lists, stacks, and queues. Most often, these linear data structures are used within other data structures to build more complex data structures. The main goal of this class was to learn to not only program our own data structures, but to use those that we created in order to build other data structures. In our projects, linked lists and disjoint sets were very important data structures to use. I believe that through my use of these two linear data types in my projects, that I achieved mastery in my understanding of these versatile data structures.

3.1 Linked Lists

A linked list is a linear data structure that is an ordered set of data nodes, each of which have a pointer to its successor. Many of our projects had parts that were implemented using a linked list. Although we programmed a linked list back in CS 173, implementing it in our new data types gave a new challenge and truly tested my understanding of linked lists. One example where linked lists were used was in the programming of a hash table, which used an array of linked lists to store our movies that were queried. It was declared as a KeyType template, so that we could insert whatever data type we wanted into our table. It was initialized in our hash.h file as `List<KeyType> *table; // an array of List<KeyType>'s`. We initialized this array with size equal to the number of slots, in this case 1000. In the Hash Tables section, methods that utilize linked lists are shown.

3.2 Disjoint Sets

A disjoint set is a data structure that is a set of smaller sets that do not have any of the same elements, hence the name disjoint. A disjoint set is used to implement Kruskal's algorithm in the Graph class. A disjoint set supports three main operations: make set, union, and find set. Below I have included my code for findSet, which, when given a node pointer, returns the identifier of the set that contains the node. This method has one of the most efficient and elegant use of recursion. Since we only care about the root of the set, this function makes each node on the path point directly to the root. This idea is called path compression, which increases run-time efficiency.

```
//PreCondition:  x not NULL
//PostCondition:  return root of set
template <class T>
DSNode<T>* DisjointSets<T>::findSet(DSNode<T>* x) // destroys the set/array
{
    if (x==NULL)
        throw NotFoundError();
    if(x!=x->parent)
```

```
        x->parent = findSet(x->parent);
    return x->parent;
}
```

In order to implement a disjoint set class, we needed to write a specific node class for our disjoint set, which is included below.

```
ifndef DSNODE_H
define DSNODE_H
include <iostream>
include <sstream>
template <class T>
class DSNode
{
    public:
        DSNode(T* d)
        {
        parent = this;
        rank = 0;
        data = d;
        };
    DSNode<T>* parent;
    T* data;
    int rank;
    string toString() const
    {
        stringstream string;
        string « *data;
        return string.str();
    }
};
```

4 Priority Queues and Binary Heaps

4.1 Minimum Priority Queue

The most commonly used linear data structure used throughout this course was a minimum priority queue. A min-PQ is a very versatile data structure, and has many applications. It was used for both the Huffman Coding Project (Project 4) and in the Graph Project (Project 8). In Project 4, we inherited methods and constructors from a min heap class that we had previously programmed. In Project 8, we used a min-PQ to keep track of vertices in Kruskal's algorithm. On our first exam, we were asked to write pseudocode for the four main functions of a minimum priority queue: `minimum(), extractMin(), decreaseKey(index, newKey),` and `insert(key).` unfortunately, I was not prepared, and did not score well. However, we were asked the same question on Exam 2, which I answered perfectly, as shown below. Based on my improvement between the two exams, I believe this shows my mastery in the understanding of minimum priority queues. Below are my answers to the question from Exam 2, as well as the run time for each function.

```
minimum() run time:  O(1)
    if h.heapsize == 0
        throw EmptyError()
    return h.A[0]
```

```
extractMin() run time: O(log n)
    if h.heapsize == 0
        throw EmptyError()
    min = h.A[0]
    h.A[0] = h.A[heapSize-1]
    heapSize-
    minHeapify(0)
    return min


decreaseKey(index, newKey) run time: O(log n)
    if h.heapsize == 0
        throw EmptyError()
    if h.A[index] < newKey
        throw KeyError()
    if index ≥ h.heapSize
        throw IndexError()
    h.A[index] = newKey
    while index > 0 & h.A[parent(index)] > h.A[index]
        swap(index, parent(index))
        index = parent(index)


insert(key) run time: O(log n)
    if h.capacity < h.heapSize + 1
        throw FullError
    h.A[heapSize] = key
    heapSize ++
    h.decreaseKey(heapSize-1, key)
```

4.2 Binary Heap

A binary heap data structure is an array object that can be represented as an almost complete binary tree. It is a complete binary tree up until the last row, where the leaves fill in from the left to the right. In my implementation, the heap is stored in an array. The tree is represented in the array's indices, with the parent of the node being at $\lfloor i/2 \rfloor$, its left child at $2i$, and its right child at $2i + 1$. These index formulas are used to give a representation of the heap as a binary tree. The values of the nodes in a min-heap, the type we studied in class, satisfy a heap property: the key of the parent must be $\leq$ the keys of both its children. Heaps have a practical application for implementing a minimum priority queue. It allows a min-PQ to `insert` and `extractMin` in at worst $O(\lg n)$ time, while allowing constant run time for the `minimum` method.

Below is the code for two methods from our heap project: `heapify` and `buildHeap`. `heapify` is used to maintain the heap property after we perform any of the four main priority queue methods. `buildHeap` is used to create a heap from an input array. `buildHeap` is an elegant method, because it builds the heap from the bottom up. It does this because the precondition of `heapify` is that the children of the index being heapified must both be min-heaps. Starting at $\lfloor \text{length}/2 \rfloor$ begins at the first node that is not a leaf, and therefore its children are always heaps since all leafs are heaps. As we move up the tree, the children of each node are heaps since we have already called `buildHeap` for each of its children. I believe that, between reading notes, this project, and exam questions, that I have mastered binary heaps with distinction.

```
Precondition:  this function assumes that the children of index are both min heaps,
and that A[index] may be smaller than its children.
Postcondition:  the binary tree rooted at index in now a min heap.

template <class KeyType>
void MinHeap<KeyType>::heapify(int index)
{
    int l = leftChild(index);
    int r = rightChild(index);
    int smallest;
    if (l < heapSize && A[l] < A[index])
        smallest = l;
    else
        smallest = index;
    if (r < heapSize && A[r] < A[smallest])
        smallest = r;
    if (smallest != index)
    {
        swap(index, smallest);
        heapify(smallest);
    }
}

BuildHeap
Precondition:  each node from i+1 to n is the root of a min heap.
Postcondition:  each node from i to n is now the root of a min heap.
template <class KeyType>
void MinHeap<KeyType>::buildHeap()
{
    for (int i=parent(heapSize); i>=0; i-)
        heapify(i);
}
```

5 Graphs

5.1 Representation

A graph is made up of two parts: a set of vertices and set of edges, and is written as $G = (V, E)$. A graph can be either directed, which is when the one can only travel along the edges in one direction, or undirected, where one can travel both direction along each edge. Edges also can have a weight: a numerical value that can represent distance or cost, among many others. Graphs can be represented as either an adjacency list or an adjacency matrix. An adjacency list is an array containing each vertex at the head of a list, with each next node pointing to the nodes it is connected to. This representation is more efficient to use in a sparse graph, meaning that most of the nodes are connected. An adjacency matrix is represented as an n by n matrix, with the weights of the edges falling in the row and columns responding to the edge's end vertices. An adjacency matrix is more efficient for storing the weights of the edges and for dense graphs. Graphs can be practically used in many shortest path algorithms, such as in a GPS or calculating flight paths.

In our graph project, we used both representations. We read in an adjacency matrix from a text file, and used that to create our adjacency list. I believe that my code examples included in section 5.2 from Project 8 show my mastery of the subject. Not only does the actual code show my mastery, but the in depth descriptions of the pre and post conditions also support my cause.

5.2 DFS and BFS

Graphs have two main methods for visiting every node in a graph: Depth First Search and Breadth First Search. BFS takes in a source or starting vertex, and computes the shortest distance to each node. It traverses the whole graph by looking at all of vertex's adjacent vertices, before moving on to the next vertex.

DFS is the searching method that was implemented in our project. DFS explores edges connected to the most recently discovered vertex, before going down more adjacent paths. Once it encounters a node with no new adjacent vertices, it back tracks until there is another path to travel down. DFS keeps track of a vertex's discovery time and finish time. The discovery time is updated when a new vertex is first visited, and its finish time is updated when all its adjacent vertices have been visited. Below, I included my `DFS` and `DFS-Visit` methods.

```
Precondition:  the graph must be a valid, directed graph
Postcondition:  the order in which the vertices are discovered in, the discovery and finish times,
and the colors of the vertices in each step are all printed out.
Each vertex that is reachable from the start must have been visited, and therefore is black.

void Graph::DFS()
{
     timer = 0;
     cout « "\n" « endl;
     for (int i=0; i<vertices; i++)
     {
          for (int j=0; j<adjList[i].length(); j++) // initializes color
          {
               adjList[i][j]->color = "white";
               adjList[i][j]->pred = NULL;
          }
     }
     for (int i=0; i<vertices; i++)
     {
          for (int j=0; j<adjList[i].length(); j++)
          {
               if (adjList[i][j]->color = "white)
               {
                    cout « "Discovery Order:  \n" « endl;
                    DFSvisist(adjList[i][j]); // gets visited pathway
               }
          }
     }
     cout « "\n" « endl;
     for(int i =0; i<vertices;i++) //prints pathway
     {
          cout«"Vertex:  "«*adjList[i][0] «" Discovery Time:  "« adjList[i][0]->discovery « endl;
          cout « "Vertex:  " « *adjList[i][0] « " Finish Time:  " « adjList[i][0]->finish « endl;
     {
}
```

5.3 Kruskal's and Prim's

Kruskal's algorithm and Prim's algorithm are methods that generate a minimum spanning tree from a graph. Prim's algorithm has the property that all edges in set A form a single tree. All vertices not yet in A are kept in a minimum priority queue. The vertices determine a cut in the graph, and each new edge is a light edge that crosses that cut. Kruskal's algorithm works in a slightly different way: Each vertex is initialized as its own tree in a forest, with cuts separating each tree. The algorithm finds the light edge that crosses any of these cuts. We first had to implement a disjoint set class for this algorithm, which is a set of linked lists. The pre and post conditions as well as my code for Kruskal's algorithm from Project 8 are below.

```
PreCondition:  graph must be valid with a valid adjacency list / weighted class
PostCondition:  Each vertex must be its own forest (done in first loop).
There will only be one forest for each part of a connected graph.
(minimum spanning tree)

void Graph::Kruskal()
{
    kruskal = new MinPriorityQueue<Edge<Vertex»;
    List<DSNode<Vertex» S;
    DisjointSets<Vertex> forest;
    List<Edge<Vertex» array;
    // makes each disjoint set
    for (int i=0; i<vertices; i++)
    {
        DSNode<Vertex> *v = forest.makeSet(adjList[i][0]);
        S.append(v);
    }
    for (int i=0; i<vertices; i++)
    {
        for (int j=0; j<vertices; j++)
        {
            if (matrix[i][j] != 0)
            {
                / sets up edges from matrix
                Edge<Vertex> *e = new Edge<Vertex>(adjList[i][0], adjList[j][0], matrix[i][j]);
                kruskal->insert(e); // inserts into the graph edges
            }
        }
    }
    int length = kruskal->length();
    for (int i=0; i<length; i++)
    {
        Edge<Vertex> *current = kruskal->extractMin();
        if (forest.findSet(S[(current->FirstNode->ID)-1]) != forest.findSet(S[(current->SecondNode->ID)-
        {
            // gathers minimum
            array.append(current);
            forest.unionSets(forest.findSet(S[(current->FirstNode->ID)-1]), forest.findSet(S[(current->S
            // union sets u and v to get graph
        }
    }
    cout « array « endl;
}
```

6 Hash Tables

6.1 Implementation

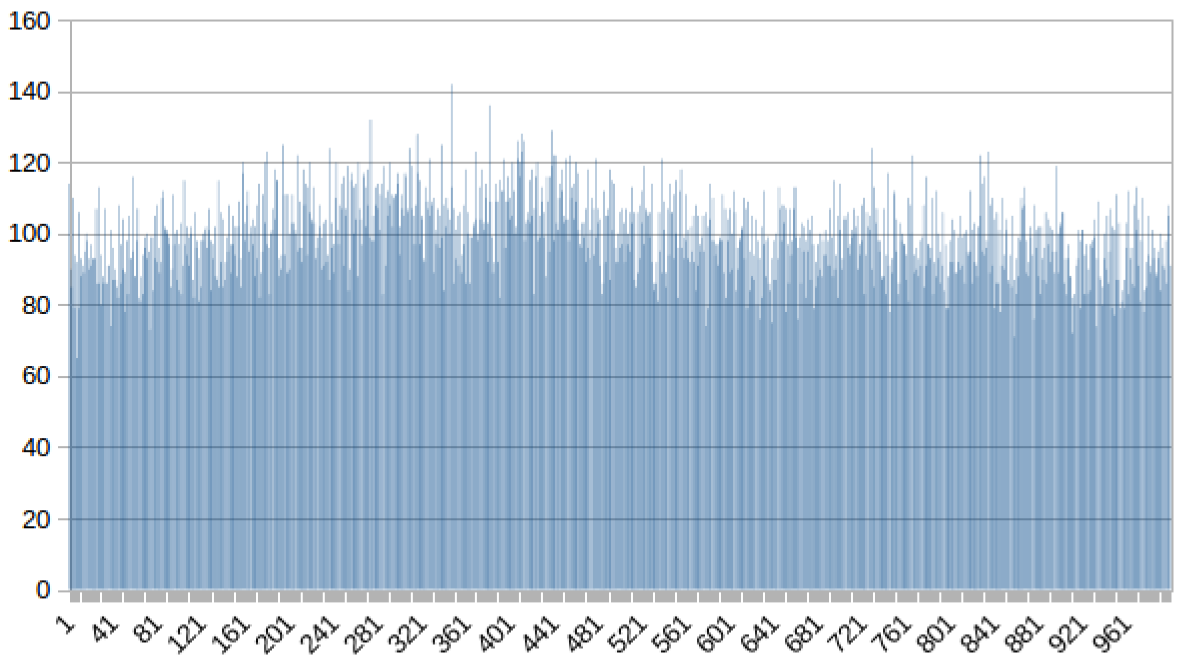
Hash tables are an extremely efficient data structure used to implement a dictionary. They use a hash function to simulate a pseudo-random distribution of the items into an array of linked lists. A hash table is one of the most efficient ways to implement a dictionary. It allows the dictionary to search and insert items in constant time. Below are the `get` and `insert` methods from my Project 6, with proper pre and post conditions. I believe that I obtained mastery with distinction for hash tables, not only because of the high grade I received on this project, but also with how efficiently our hash function ran, and how uniformly it distributed the movies.

```
// precondition:  k is in the hash table, returns NULL otherwise
// postcondition:  k is returned from the hash table
template <class KeyType>
KeyType* HashTable<KeyType>::get(const KeyType& k) const
{
    // getItem will return NULL if the item is not found, therefore
    // get will also return NULL if the item is not found.
    KeyType* key = table[k.hash(Slots)].getItem(k);
    return key;
}

// insert
// precondition:  k is of the same KeyType as the hash table, and the KeytType of k
has its own hash function
// postcondition:  the hash table contatains k
template <class KeyType>
void HashTable<KeyType>::insert(KeyType *k)
{
    table[k->hash(Slots)].insert(0, k);
}
```

6.2 Hash Functions

A hash function allows inserting in constant. The hash function is used to compute the slot from a key k. The key type of k has to have a hash function defined within it. A good hash function will give an approximately random distribution of the items in the table, with around n/m items per slot. n represents the number of total items in the table and m is the number of slots. Our hash function from Project 6 yielded a standard deviation of 10.8, and gave our movie query an average runtime of .087 seconds. How hash function takes each character and mulitplies it by 3^i, with i being the character in the word. This creates a very unique integer for almost every movie. We also check for overflow by checking if the previous sum is greater than the current sum, which means our sum grew above the maximum possible integer. The distribution of our hash table is shown below. It shows a very uniform distribution.



In addition, my understanding of hash tables extends to Exam 3 and the problems with bad hash functions. Question 2 asked: Why is $h(s) = (s[0] * s[1]) \bmod m$ a potentially problematic hash function for data with string-values keys s? My edits are marked in bold.

There are many possible string that begin with the same two characters. This hash function will cause all of those words to hash to the same spot, resulting in long linked lists in ~~each~~ **some** of the slots**, which slows down the run time of searching and deleting.** This will also result in a pattern**, which doesn't generate a pseudo-random distribution of the data.**

7 Binary Search Trees

A binary search tree is a tree in which each node has two children. The binary search tree property states: Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key $\leq x$.key. If y is a node in the right subtree of x, then y.key $\geq x$.key. A BST can be used to implement a dictionary, which was done in Project 5. Between Project 5 and Project 7, we got a 209/200, for implementing a remove function on our Red-Black Tree. I believe that, between the project scores and my BST and RBT test results, I achieved mastery with distinction on binary search trees. I would say that BSTs are the data structure that I understand the best. This understanding is not limited to the programming of a BST, but also my understanding of why they work. For example, question 2 from Exam 2, which asked to prove the following statement by induction: A complete binary tree with height h has $2^h - 1$ internal nodes. My in depth proof below demonstrates my understanding of this topic.

I will prove this statement using strong induction on h. For the base case, let $h = 0$, so a tree with height 0 has $2^0 - 1 = 0$ internal nodes, which is true because the only node is the root, which is a leaf. For the induction hypothesis, assume that a tree with height k has $2^k - 1$ internal nodes for all $k = 1, 2, ..., h-1$. If we take a tree T with height h, its subtrees located at the root's right and left children are also binary trees with height $h - 1$. So, by the induction hypothesis, each of the subtrees have $2^h - 1$ internal nodes. Therefore, the whole tree has a $(2^{h-1} - 1) + (2^{h-1} - 1) + 1 = 2^h - 1$ internal nodes. ■

7.1 Red Black Trees

Red Black Trees are a more efficient type of binary search tree. They are more efficient because they are able to adjust their structure to ensure that no one path is more than twice as long as another. It does this through ensuring that the five properties are held: 1. Every node must be either red or black. 2. The root must be black. 3. The leafs must be black. 4. Every red node must have two black children. 5. The black height of any simple path from the root to a child must be the same for the whole tree. RBTs can be used in place of a binary search tree to implement a dictionary, but because the tree must hold these rules, it stays balanced and therefore can search, insert, and delete much faster. Regular binary search trees can, in the worst case (when inserted all in order) can run at $O(n)$ time for all of a dictionary's main methods. However, an RBT-implemented dictionary can perform these methods in at worst $O(\lg n)$ time.

```
insert k into the tree
Precondition:  RBT is initialized
    All 5 RBT properties are held
Postcondition:  Key is inserted as a leaf in its correct position,
    and all 5 RBT properties are held

template <class KeyType>
void RBT<KeyType>::insert(KeyType *k)
{
    Node<KeyType>* parent = nil; //trailing pointer
    Node<KeyType>* current = root;
    while(current!= nil)
    {
        parent = current;
        if(*k<=*current->data)
        {
            current = current->leftChild;
        }
        else
            current = current->rightChild;
    }
    Node<KeyType>* newNode = new Node<KeyType>(k);
    newNode->parent = parent;
    if(parent == nil)
    {
        root = newNode;
    }
    else if(*(newNode->data) <= *(parent->data))
    {
        parent->leftChild = newNode;
    }
    else
    {
        parent->rightChild = newNode;
    }
    newNode->color = "red";
    newNode->leftChild = nil;
    newNode->rightChild = nil;
    insertFixup(newNode);
}
```

8 Object-Oriented Programming

8.1 Template Classes

Template classes are a way of programming a data structure to be versatile enough to hold any data type. It keeps the class abstract and does not force it to depend on the data type. A template class includes either `template <class KeyType>` or `template <class T>` above the class declaration and each method. Understanding the intuition and reasoning behind template classes was key to succeeding in the course. The majority of the data structures in our projects were implemented as template classes. For example, our hash table was a template class. This is because the dictionary has to also be abstract. This allowed us to create a data type that contained our own hash function within its methods. Although the end goal was to use our dictionary to execute a movie query, we tested the efficiency of our hash functions on integers and strings. I believe that because of its repetitive use throughout the semester, I achieved mastery of this subject, which helped me succeed in many other competencies in this course.

Many of our projects used nodes to store our data, which were able to hold any type of data. Because many of our methods included comparisons, we had to ensure that comparative operators were included in each KeyType we used. As shown in the code below from the Binary Search Tree class, we must compare the data stored in the node to the key that is passed into the method. This meant that we had to create == and < operators for our movie class. We implemented these operators by comparing the third character of the title. We chose the third character in order to prevent all of the movies being inserted in alphabetical order. Also, short movie titles, like *Up*, would throw an index out of range error.

```
template <class KeyType>
Node<KeyType>* BST<KeyType>::search(Node<KeyType> *node, KeyType key)
{
    if(node==NULL)
        return NULL;
    if(*(node->data)==key)
        return node;
    else if(key<*(node->data))
        return search(node->leftChild, key);
    else
        return search(node->rightChild, key);
}
```

8.2 Inheritance

Another concept in object-oriented programming is inheritance, which is when a class uses the same or similar methods as a parent class. Inheritance allows for simpler programming by letting the programmer re use methods. In addition, it extends the logic used in the parent class into the current class. For example, Binary Search Trees, Hash Tables, and Red Black Trees are all ways to implement a Dictionary class. Each one has its advantages and disadvantages. A dictionary's main methods include constructors, insert, remove, get, and empty, all of which were inherited from these classes in each of Project 5, 6, and 7. I included my Red Black Tree implementation of a Dictionary below. This program is extremely concise, which shows how much time and typing effective inheritance can save.

Since using inheritance was a big tool in the three projects already mentioned as well as in Project 4, where we inherited from a min heap to implement a minimum priority queue, I can say with confidence that my understanding of this topic is mastery with distinction. Inheritance is a fundamental part of programming that I hope to carry with me into the professional world.

```
template <class KeyType>
class Dict:  public RBT<KeyType>
{
    public:
        Dict():RBT<KeyType>() {}
        Dict(const Dict &d):RBT<KeyType>(d){}

        void insert(KeyType *k) {if (get(*k)==NULL){RBT<KeyType>::insert(k);}}
        void remove(KeyType k) {RBT<KeyType>::remove(k);}
        KeyType* get(const KeyType k) {return RBT<KeyType>::get(k);}
        bool empty() {return RBT<KeyType>::empty();}
};
```

9 Professional Practice

As much as I learned about coding and data structures in this class, I feel that I learned many skills that are not directly related to programming as well; skills I can take with me into the real world. After all, that is what college is about, preparing for life after graduation. These skills include time management, teamwork, and mastery of LaTeX.

9.1 Time Management

When it comes to being a student-athlete, nothing is more important than time management. Fitting in practices, classwork, eating healthy, and getting enough sleep is no simple task. Having reading notes due almost every class helped me to stay on top of the readings. This semester, not once did I struggle to finish an assignment the hours before the due date. Earlier in my college career, I learned the hard way about the negative consequences of my procrastination. This semester however, I learned how to start early, and work in steps to avoid a last minute panic. At first, I had to force myself to start my projects early, but as the semester went on, it started to come more naturally. In addition, I never felt the need to be up extremely late the night before an exam studying, because I always started preparing early. Staying on top of my assignments this course was crucial, and something that I mastered with distinction.

9.2 Teamwork and Communication

When working as a programmer for a career, it is highly unlikely to be working alone on a project. Most real-world programs are written my a team of coders. For this reason, I am glad that most of our projects were partner projects. It gave me experience working collaboratively on a program. In addition, having us switch up partners for the last project forced me to adapt to working with someone new. Brett and I worked great together, but in the real world, I will not always be lucky enough to work with someone who I get along extremely well with. This is why it is important to learn how to work together with anybody, which I believe I mastered over this semester.

In addition, my communication skills have also improved, which includes not just working together but also in commenting my programs. Comments are a very important part of the program, because it allows me to help myself when it comes to error checking. Commenting can also include writing the pre and post conditions of an algorithm, as well as a short description of what the algorithm accomplishes right above its declaration. In the beginning of the year, I would be searching through my code for an error, and would come across lines of code that I could not easily discern what I was trying to accomplish. As the year went on, I learned how to comment my code as I wrote it. I believe that this helped me and my partner understand our code better, as well as make the grading process easier.

Commenting code is good practice for the professional world because oftentimes someone else will read the code I wrote for error checking and testing. Writing comments to explain the code to them is a huge help. The improvement in my commenting skills and partner programming shows my mastery with distinction when it comes to teamwork and communication.

9.3 LaTeX

When I was first introduced to LaTeX last semester, I was a bit skeptical of its practicality. I remember doing a homework assignment regarding sets, and trying to remember to type a backslash before the brackets gave me headaches. Once I got the hang of the syntax, LaTeX has become an extremely useful tool to add to my arsenal. It is very helpful when it comes to writing technical documents. For this course, I used it for all of my reading notes, answering questions for projects, and to write this portfolio. Outside of class, I learned many new features that are supported by LaTeX, and even used it for parts of my project in my Data Analytics 101 course and to write my resume. Because of its many practical uses, I believe that I have mastered with distinction the art of LaTeX, to the extent that I felt confident in including it under my skills on my resume.

10 Conclusion

This course has been an extremely valuable learning experience for me. Many of the data structures we learned about are very common and useful in the real world. I hope that my knowledge gained in this course will help me be successful whenever I find a job. When I was interviewing for an internship this summer, I found that I could answer many of the questions using information I had gathered from this class. In addition, I found that the topics in this course were actually interesting, which made learning them much easier. Through my grade in this class and for the reasons above, I can say with confidence that I have mastered this course. Being able to enjoy learning in this course further solidifies my overall understanding and interest in the competencies provided in this course.