CS 271 Portfolio

Emma Steinman

December 2017

1 Introduction

In this portfolio I will cover my understanding of the core competencies of CS 271. Each section will begin with the competency and a short description, my perceived level of proficiency and explanation, followed by sample code, homeworks, and tests to support my claims. The scale I will be assessing myself on consists of the three following levels; proficiency, mastery, and mastery with distinction. Proficiency means a comprehensive understanding of the concept and when to apply it. Mastery implies more polish and expertise than proficiency; a deeper understanding, less frequent errors, and higher quality work. The last level, mastery with distinction, suggests an extremely high level of understanding with elegant work and an independent initiative to further study the topic.

2 Analysis of Algorithms

In this section I will discuss the analysis of algorithms including proving the correctness and the asymptotic time complexity. To prove the correctness of algorithms, you use pre-conditions, post-conditions, and loop invariants. I believe I have reached mastery in this section. I missed few to no points on the homeworks and exam in this area. A pre-condition is something that is true before the function, and a post-condition is something that is true after the function. A loop invariant is a statement that is true before and after each iteration of a loop. To prove a loop invariant is correct, you use a process that is very similar to induction proofs. You start with the statement of the loop invariant, similar to stating the Hypothesis in an induction proof. The Initialization step is similar to the Basis where you prove the invariant to be true for the most basic case. The Maintenance step is comparable to the Inductive step where you assume the invariant to be true before some arbitrary iteration and then prove it to be true for the next iteration. The last step, Termination, is not included in an induction proof, but you just prove the invariant is true after the loop ends. In addition to proving an algorithm is correct, analyzing the time complexity is also important because there are many ways to solve a problem, but some solutions are noticeably faster than others. Analyzing the algorithms

before implementing can indicate which is faster and save time instead of just running the code and timing it. Additionally, you may not want to just run the code because some functions are efficient on small inputs but very inefficient on larger sets and vice versa, and analyzing the algorithm beforehand can catch these differences, as well as show it is correct. It is helpful to know which time complexities are better than others, for example logarithmic time complexity is much more efficient than exponential time complexity. If you know the time complexity of common algorithms you can compare your code and see if yours is more efficient. For example, the most efficient searching algorithm has $\Theta(logn)$ so if your algorithm is linear time you should try to make it more efficient. Throughout the semester we analyzed both iterative and recursive algorithms. On Project 1 we proved loop invariants and time complexities for a linear search and for Bubble Sort. In the linear search algorithm, the loop invariant was "Before iteration i of the for loop, if v is in the set, the value v is in A[i..n]". Then I proved the loop invariant is true using the Initialization, Maintenence, and Termination steps. This loop invariant is true because when the algorithm finds the correct value it returns the index, thus if it is still going through the loop it hasn't found the index or else it would have returned. Using the three step method proves this for an arbitrary sequence, thus it is true for any input sequence. I then proved the time complexity using Θ notation. Since the algorithm will search through each element linearly until it finds the one it is looking for, the best case is the first element is the desired element in that case it would take $\Theta(1)$. The worst case is the element is not in the sequence and returns NIL, in which case the algorithm searches through n elements and it will take $\Theta(n)$. I then found the average time complexity which comes out to be $\Theta(n)$.

The second algorithm is Bubble Sort. This algorithm repeatedly swaps adjacent elements if they are not in order. It will traverse through the sequence n-1 times to ensure the list is in sorted order. There is a way to optimize the algorithm by jumping out of the loop early if no swaps have been made, but the algorithm we were given was not optimized. Since this algorithm has two for loops, there are two separate loop invariants. The loop invariant for the inner for loop is "Before each iteration j of the inner for loop, A[j..j+1] contains the elements previously in A[j..j+1] but in sorted order". I then proved this using the Initialization, Maintenance, and Termination steps. Ultimately this invariant is true because the loop is only swapping two elements at a time which will leave the elements in the same section of the array. The loop invariant for the outer loop is "Before each iteration i of the outer for loop, A[1..i] contains elements from A in sorted order". I then proved this using the Initialization, Maintenance, and Termination steps as well. This invariant is correct because the algorithm swaps items in place within the range so the items will stay within the given range but in sorted order from the inner loop. Then I proved the time complexity of Bubble Sort. If your algorithm is optimized, the best case time is $\Theta(n)$ which occurs when the array is already sorted, and it will do one pass through the loop and jump out because no swaps were made. However, our algorithm was not optimized, so it will go through the outer loop $n-1$ times and

the inner loop $n-1$ times as well, regardless of the amount of swaps happening. This comes out to be $\Theta(n^2)$. This is also the time complexity for the worst case, which occurs when the array is in reverse sorted order so the algorithm has to swap each time it finds an element. Since both best and worst case times are $\Theta(n^2)$, the average time complexity is $\Theta(n^2)$. In the case of asymptotic time complexities, quadratic time complexities are among the worst you can have, besides from exponential. For a sorting algorithm, the lowest time complexity you can have is $\Theta(nlogn)$ so this algorithm is not efficient.

On the first exam there were a few questions about loop invariants and time complexities. The first question, about heapSort, asked us to prove the loop invariant "Before iteration i of the for loop, the subarray `A[0..i]` is a max-heap containing the `i+1` smallest elements of `A`, and the subarray `A[i+1..n-1]` contains the `n-1` largest elements of A, in sorted order". I used the Initialization, Maintenance, and Termination steps to prove this is true. This invariant is true because before every loop, the previous section of the array was a max-heap so adding one more element and then calling heapify will maintain that property.

The second question involves finding the time complexity of a recursive algorithm. It gives the running time in a recurrence and I solved it using properties of logs and the substitution method. Then by induction I proved that it is correct for arbitrary values thus it will be true for any values. The second project I analyzed time complexities using the properties of asymptotic time complexities. Using the definitions of O, Ω,and Θ I proved that for asymptotic time complexities, the limit will always converge to the higher power of n. The third question asked to prove recurrences which is how you solve for time complexities of recursive algorithms. I solved these by using the substitution method and properties of summations.

Overall I feel as though I understand this topic extremely well including how to calculate and the importance of it. Time complexity will definitely help later in my career, whether it is during my time at Denison or later in my career.

INCLUDE EXAMS!!!11

Emma Steinman
CS 271 Project 1
December 16, 2017

1.b Before iteration i of the for loop, if v is in the set, the value v is in A[i..n].

1.c Initialization
Before the first iteration, $i = 1$, of the for loop, if v is in the set, v is in A[i..n]. A[i..n] is the entire set so it makes sense that if v is in the set, it is in A[i..n] so the loop invariant is true before the first iteration.
Maintenence
Assume the loop invariant is true before some iteration i. The loop invariant states that if v is in the set, v is in A[i..n]. If the function has not returned a value yet, it means it has not reached v in the set. There are two cases in the loop. The first is that $A[i] = v$. In this case, the function would return and the loop invariant would say the value v is in A[i..n] and it was at A[i] which holds true. The second case is that A[i] is not equal to v. In this case, i is incremented and the loop invariant will say, if v is in the set, v is in A[i+1..n]. Since v was not in A[i] or any previous places (since the function has not returned yet) then if it is in the set the value of v must be in the remainder of the set (A[i+1..n]) so the loop invariant holds true.

1.d Termination
The loop can end for two reasons.
Case 1 (return i)
If $A[i] = v$, search will return i and therefore the loop will end. In this case, the loop invariant says: if v is in A, it will be in A[i..n]. Since the function returned because $A[i] = v$, we know this is true because A[i] is in A[i..n].
Case 2 (v is not in A)
If search completes the entire for loop it is because A[i] was never equal to v; v is not in the set A. At the end of the loop, i increments one more time so $i = n+1$. The loop invariant will then say, After the for loop terminates, v will be in A[n+1..n]. Since this set doesn't exist, the statement is not false, therefore the loop invariant still holds.

1.e The best case time complexity is $\Theta(1)$. The best case would be the first item in the set is the value v, so the loop takes a constant amount of time to find it and return its index. The worst case time complexity is $\Theta(n)$. The worst case would be value v is not in the set, so the function has to complete the loop n times to check each item. The average time complexity is $\frac{\sum_{i=1}^{n+1} \Theta(i)}{n+1} = \frac{\Theta((n+1)*(n+2)/2)}{(n+1)} = \Theta(n)$

2.a Before each iteration j of the inner for loop, A[j..j+1] contains the elements previously in A[j..j+1] but in sorted order.
Initialization
Before the first iteration, $j = n$, A[n..n+1] contains the elements previously in A[n..n+1] but in sorted order. This set does not exist so it is not false, therefore it holds.
Maintenance
Assume the loop invariant is true before some iteration j. The loop invariant then states A[j..j+1] contains the elements previously in A[j..j+1] in sorted order. During iteration j, if A[j-1] is greater than A[j], they are swapped. Thus at the end of the iteration, A[j..j+1] will be the same elements but in sorted order, as the loop invariant states. At the beginning of the next iteration, the elements in A[j..j+1] which are the elements previously in A[j-1..j] are in sorted order because they have not been moved since the end of the last iteration. Thus the loop invariant maintenance step holds.

2.b Before iteration each iteration i of the outer for loop, A[1..i] contains elements from A in sorted order.
Initialization
Before the first iteration, $i = 1$, A[1..1] contains elements from A in sorted order. A[1..1] is just A[1] and a single element is always sorted, thus the loop invariant holds.
Maintenance
Assume the loop invariant is true before some iteration i. That says that all elements in A[1..i] are elements from A in sorted order. During that loop, the next smallest item will be swapped into place A[i]. At the end of the loop, the statement is true because A[1..i] still contains the same elements, but before the next iteration A[1..i] will be equivalent to A[1..i+1] which is in sorted order.
Termination (of inner loop)
At the end of the inner for loop, $j = i$, the termination condition states A[i..i+1] are elements from A in sorted order. Those are the next two smallest elements to be sorted so this proves the outer for loop, because the outer loop invariant claims that A[1..i] will be sorted, so this will 'sort' the next two items.

2.d The best case time is $\Theta(n^2)$which occurs when all of the items are already in sorted order.This is because the algorithm has to go through each item in the list for both loops, even if the items are already sorted. The worst case time is also $\Theta(n^2)$ because the algorithm does not behave differently if the items are already sorted.
CORRECTION: The best case time is actually $\Theta(n)$ if the algorithm is optimized to jump out of the loop early if no swaps were made.

Emma Steinman
CS 271 Project 2
December 16, 2017

1 For any two functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ means there exists constants c_1 and c_2 such that $g(n)$ creates an upper AND a lower bound for $f(n)$, or $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all values of n greater than some n_0. Thus, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ AND $f(n) = \Omega(g(n))$. $f(n) = O(g(n))$ states that $0 \leq f(n) \leq cg(n)$ or that some constant, c, times $g(n)$ creates an upper bound for $f(n)$. $f(n) = \Omega(g(n))$ states that $0 \leq cg(n) \leq f(n)$ or that for some constant, c, times $g(n)$ creates a lower bound for $f(n)$. Thus, in order to have $f(n) = \Theta(g(n))$ you must have $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Additionally, if you have $f(n) = \Theta(g(n))$ you know you have both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

2.a $n^2 + 3n - 20 = O(n^2)$
$0 \leq n^2 + 3n - 20 \leq cn^2$
$n^2 + 3n - 20 \leq 2n^2 n = 1 \Rightarrow 0 \leq -16 \leq 2X$
...
$n = 3 \Rightarrow 0 \leq -2 \leq 9X$
$n = 4 \Rightarrow 0 \leq 8 \leq 32\surd$
$c = 2$
$n_0 = 4$

2.b $n - 2 = \Omega(n)$
$0 \leq cn \leq n - 2$
$0 \leq \frac{1}{2}n \leq n - 2$
$n = 1 \Rightarrow 0 \leq \frac{1}{2} \leq -1X$
$n = 2 \Rightarrow 0 \leq 1 \leq 0X$
...
$n = 5 \Rightarrow 0 \leq \frac{5}{2} \leq 3\surd$
$c = \frac{1}{2}$
$n_0 = 5$

2.c $log_{10}n + 4 = \Theta(log_2 n)$
$0 \leq c_1 log_2 n \leq log_{10}n + 4 \leq c_2 log_2 n$
$0 \leq c_1 log_2 n \leq \frac{log_2 n}{log_2 10} + 4 \leq c_2 log_2 n$
$0 \leq \frac{1}{log_2 10} log_2 n \leq \frac{log_2 n}{log_2 10} + 4 \leq \frac{2}{log_2 10} log_2 n$
$c_1 = \frac{1}{log_2 10}$
$c_2 = \frac{2}{log_2 10}$
$n_0 = 1$

2.d $2^{n+1} = O(2^n)$
$0 \leq 2^{n+1} \leq c2^n$
$0 \leq 2^n + 1 \leq 3 * 2^n$
$n = 1 \Rightarrow 0 \leq 4 \leq 6\surd$
$n = 2 \Rightarrow 0 \leq 8 \leq 12\surd$
$c = 3$
$n_0 = 1$

2.e $ln(n) = \Theta(log_2 n)$
$0 \leq c_1 log_2 n \leq ln(n) \leq c_2 log_2 n$
$0 \leq c_1 log_2 n \leq log_e n \leq c_2 log_2 n$
$0 \leq c_1 log_2 n \leq \frac{log_2 n}{log_2 e} \leq c_2 log_2 n$
$c_1 = c_2 = \frac{1}{log_2 e}$
$n_0 = 1$

2.f $n^\epsilon = \Omega(lgn)$ for any $\epsilon > 0$
$0 \leq c * lgn \leq n^\epsilon$
$n = 1 \Rightarrow 0 \leq 0 \leq 1, 1, ..., 1$
$n = 2 \Rightarrow 0 \leq 1 \leq 2, 4, 8...2^\epsilon$
$c = 1$
$n_0 = 1$

3.a $T(n) = 2T(\frac{n}{2}) + n^3$
$= 2[2T(\frac{n/2}{2}) + (\frac{n}{2})^3] + n^3$
$= 4T(\frac{n}{4}) + 2(\frac{n}{2})^3 + n^3$
$= 4[2T(\frac{n/4}{2}) + (\frac{n}{4})^3] + 2(\frac{n}{2})^3 + n^3$
$= 8T(\frac{n}{8}) + 4(\frac{n}{4})^3 + 2(\frac{n}{2})^3 + n^3$
$= 8[2T(\frac{n/8}{8}) + (\frac{n}{8})^3] + 4(\frac{n}{4})^3 + 2(\frac{n}{2})^3 + n^3$
$= 16T(\frac{n}{16}) + (\frac{n}{8})^3 + 4(\frac{n}{4})^3 + 2(\frac{n}{2})^3 + n^3$
$= 2^i T(\frac{n}{2^i}) + \sum_{j=0}^{i-1} 2^j (\frac{n}{2^j})^3$
$\frac{n}{2^i} = 2$
$n = 2 * 2^i$
$\frac{n}{2} = 2^i$
$log_2 \frac{n}{2} = i$
$2^{log_2 \frac{n}{2}} T(\frac{n}{2^{log_2 \frac{n}{2}}}) + \sum_{j=0}^{(log_2 \frac{n}{2})-1} \frac{2^j n^3}{2^{3j}}$
$= \frac{n}{2} T(2) + n^3 \sum_{j=0}^{(log_2 \frac{n}{2})-1} \frac{1}{2^{2j}}$
$\frac{n}{2} T(2) + n^3 \sum_{j=0}^{\infty} (\frac{1}{2})^{2j}$
$< \frac{n}{2} T(2) + n^3 \frac{1}{1-\frac{1}{2}}$
$= \frac{n}{2} a + 2n^3$
Proof
Prove that $T(n) < \frac{n}{2} a + 2n^3$
For the base case, let $n = 2$. We need to show that $T(2) < \frac{n}{2} a + 2n^3$. $T(2) = a < \frac{n}{2} a + 2n^3$ which is correct by definition.
Now assume that $T(k) < \frac{n}{2} a + 2n^3$ for all values $k = 2, 3, ..., n-1$.
To show that $T(n) < \frac{n}{2} a + 2n^3$, we start with the definition of $T(n) = 2T(\frac{n}{2}) + n^3$.
Since $\frac{n}{2} < n - 1$ we can use the previous assumption to get the equation
$T(n) < 2[\frac{n}{4}]a + 2(\frac{n}{2})^3$
$= \frac{n}{4} a + 2(\frac{n^3}{8})$
$= \frac{n}{4} a + \frac{n^3}{4}$
which is less than $\frac{n}{2} a + 2n^3$■

3.b $T(n) = T(\frac{9n}{10}) + n$
$= [T(\frac{9}{10}(\frac{9}{10})n + \frac{9}{10}n] + n$
$= T(\frac{81}{100}n) + \frac{9}{10}n + n$

$= [T(\frac{9}{10}(\frac{81}{100})n + \frac{81}{100}n] + \frac{9}{10}n + n$
$= T(\frac{729}{1000}n) + \frac{81}{100}n + \frac{9}{10}n + n$
$= T((\frac{9}{10})^i n) + \sum_{j=0}^{i-1}(\frac{9}{10})^j n$
$(\frac{9}{10})^i n = 2$
$(\frac{9}{10})^i = \frac{2}{n}$
$(\frac{10}{9})^i = \frac{n}{2}$
$log_{\frac{10}{9}}\frac{n}{2} = i$
$T((\frac{9}{10})^{log_{\frac{10}{9}}\frac{n}{2}}) + \sum_{j=0}^{log_{\frac{10}{9}}\frac{n}{2}-1}(\frac{9}{10})^j n$
$< T(2) + n\sum_{j=0}^{\infty}(\frac{9}{10})^j$
$= a + n(\frac{1}{1-\frac{9}{10}})$
$= a + 10n$
Proof
Prove that $T(n) < a + 10n$.
For the base case, let $n = 2$. We need to show that $T(2) < a+10n.T(2) = a < a+10n$ which is correct by definition.
Now assume that $T(k) < a + 10n$ for all values of $k = 2, 3, ..., n - 1$.
To show that $T(n) < a + 10n$, we can start with the definition of $T(n) = T(\frac{9n}{10}) + n$.
Since $\frac{9}{10}n < n - 1$ we can use the previous assumption to get the equation
$T(n) < a + \frac{9}{10}(10n) + n$
$= a + 9n + n$
$= a + 10n.$
Thus, $T(n) < a + 10n$■

3.c $T(n) = 7T(\frac{n}{3}) + n^2$
$= 7[7T(\frac{n/3}{3}) + (\frac{n}{3})^2] + n^2$
$= 49T(\frac{n}{9}) + 7(\frac{n}{3})^2 + n^2$
$= 49[7T(\frac{n/9}{3}) + \frac{n}{9})^2] + 7(\frac{n}{3})^2 + n$
$= 7^3T(\frac{n}{3^3}) + 7^2(\frac{n}{3^2})^2 + 7(\frac{n}{3})^2 + n^2$
$= 7^3[7T(\frac{n/3^3}{3}) + (\frac{n}{3^3})^2] + 7^2(\frac{n}{3^2})^2 + 7(\frac{n}{3})^2 + n^2$
$= 7^4T(\frac{n}{3^4}) + 7^3(\frac{n}{3^3})^2 + 7^2(\frac{n}{3^2})^2 + 7(\frac{n}{3})^2 + n^2$
$= 7^iT(\frac{n}{3^i}) + \sum_{j=0}^{i-1} 7^j(\frac{n}{3^j})^2$
$\frac{n}{3^i} = 2$
$\frac{n}{2} = 3^i$
$log_3\frac{n}{2} = i$
$= 7^{log_3\frac{n}{2}}T(\frac{n}{3^{log_3\frac{n}{2}}} + \sum_{j=0}^{(log_3\frac{n}{2})-1} 7^j(\frac{n}{3^j})^2$
$= 7^{log_3\frac{n}{2}}T(2) + n^2\sum_{j=0}^{(log_3\frac{n}{2})-1}\frac{7^j}{3^2j}$
$< 7^{log_3\frac{n}{2}}a + 7n^2\sum_{j=0}^{\infty}(\frac{1}{3})^j$
$= 7^{log_3\frac{n}{2}}a + 7n^2(\frac{1}{1-\frac{1}{3}})$
$= 7^{log_3\frac{n}{2}}a + 7n^2(\frac{3}{2})$
$= 7^{log_3\frac{n}{2}}a + \frac{21}{2}n^2$
Proof Prove that $T(n) < 7^{log_3\frac{n}{2}}a + \frac{21}{2}n^2$.
For the base case, let $n = 2$. We need to show that $T(2) < 7^{log_3\frac{n}{2}}a + \frac{21}{2}n^2$. $T(2) = a < 7^{log_3\frac{n}{2}}a + \frac{21}{2}n^2$ which is correct by definition.
Now assume that $T(k) < 7^{log_3\frac{n}{2}}a + \frac{21}{2}n^2$ for all values of $k = 2, 3, ..., n - 1$.

To show that $T(n) < 7^{log_3 \frac{n}{2}} a + \frac{21}{2} n^2$, we can start with the definition of $T(n) = 7T(\frac{n}{3}) + n^2$. Since $\frac{n}{3} < n - 1$ we can use the previous assumption to get the equation
$T(n) < 7[7^{log_3 \frac{n/3}{2}} a + \frac{21}{2}(\frac{n}{3})^2] + n^2$
$= 7 * 7^{log_3 \frac{n/3}{2}} a + \frac{21}{2} * \frac{n^2}{9} + n^2$
$= 7a * 7^{log_3 \frac{n/3}{2}} + \frac{39n^2}{18} < 7^{log_3 \frac{n}{2}} a + \frac{21}{2} n^2$ because $\frac{21}{2} > \frac{39}{18}$. Thus, $T(n) < 7^{log_3 \frac{n}{2}} a + \frac{21}{2} n^2$■

3.d $T(n) = T(n^{\frac{1}{2}}) + 1$
$= T(n^{\frac{1}{4}}) + 2$
$= T(n^{\frac{1}{8}}) + 3$
$= T(n^{\frac{1}{16}}) + 4$
$= T(2^{\frac{1}{2^i}}) + i$
$n^{\frac{1}{2^i}} = 2$
$= log(n^{\frac{1}{2^i}}) = log(2)$
$= \frac{1}{2^i} log(n) = log(2)$
$= \frac{log(n)}{log(2)} = 2^i$
$= log(\frac{log(n)}{log(2)}) = log(2^i)$
$= log(\frac{log(n)}{log(2)}) = i(log(2))$
$= \frac{log(\frac{log(n)}{log(2)})}{log(2))} = i$
$= log_2(\frac{log(n)}{log(2)}) = i$
$= log_2(log_2(n)) = i$
$T(n^{\frac{1}{2^{log_2(log_2(n))}}}) + log_2(log_2(n))$
$= T(2) + log_2(log_2(n))$
$= a + log_2(log_2(n))$
Proof
Prove $T(n) = a + log_2(log_2(n))$.
For the base case, let $n = 2$. We need to show that $T(2) = a + log_2(log_2(n))$.
$T(2) = a + log_2(log_2(2))$
$= a + log_2(1)$
$= a$
which is true by definition.
Now assume that $T(k) < a + log_2(log_2(n))$ for all values of $k = 2, 3, ..., n - 1$.
To show that $T(n) < a + log_2(log_2(n))$ we start with the definition of $T(n) = T(n^{\frac{1}{2}}) + 1$.
Since $n^{\frac{1}{2}}$ we can use the previous assumption to get the equation
$T(n) = a + log_2(log_2(n^{\frac{1}{2}})) + 1$
$= a + log_2(\frac{1}{2} * log_2(n)) + 1$
$= a + log_2(\frac{1}{2}) + log_2(log_2(n)) + 1$
$= a + (-1) + log_2(log_2(n)) + 1$
$= a + log_2(log_2(n)) + 1$
which proves $T(n) = a + log_2(log_2(n))$■

3.e $T(n) = T(n - 1) + lg(n)$
$= T(n - 2) + lg(n - 1) + lg(n)$
$= T(n - 3) + lg(n - 2) + lg(n - 1) + lg(n)$
$= T(n - 4) + lg(n - 3) + lg(n - 2) + lg(n - 1) + lg(n)$

$= T(n-i) + \sum_{j=0}^{i-1} lg(n-j)$
$n - i = 2$
$= n - 2 = i$
$T(n-(n-2)) + \sum_{j=0}^{(n-2)-1} lg(n-j)$
$= T(2) + \sum_{j=0}^{n-3} lg(n-j)$
$\leq a + lg(n!)$
Proof
Prove that $T(n) \leq a + lg(n!)$
For the base case, let $n = 2$. We need to show that $T(2) < a + lg(n!)$. $T(2) = a \leq a + lg(n!)$.
Now assume that $T(k) \leq a + lg(n!)$ for all values of $k = 2, 3, ..., n-1$.
To show that $T(n) \leq a + lg(n!)$ we can start with the definition of $T(n) = T(n-1) + lg(n)$. Since $n-1$ is in the range of k, we can use the previous assumption to get the equation
$T(n) \leq a + lg((n-1)!) + lg(n)$
$= a + lg((n-1)! + n)$
$= a + lg(n!)$ which is equal.■

4 Prove $F_i = \frac{\phi^i - \hat{\phi^i}}{\sqrt{5}}$.
For the base case, let $i = 1 and 2$.
$F_1 = \frac{\phi^1 - \hat{\phi^1}}{\sqrt{5}} = \frac{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}}{\sqrt{5}} = \frac{1+\sqrt{5}-1+\sqrt{5}}{2\sqrt{5}} = \frac{2\sqrt{5}}{2\sqrt{5}} = 1$
$F_2 = \frac{\phi^2 - \hat{\phi^2}}{\sqrt{5}} = \frac{(\frac{1+\sqrt{5}}{2})^2 - (\frac{1-\sqrt{5}}{2})^2}{\sqrt{5}} = \frac{(1+2\sqrt{5}+5)-(1-2\sqrt{5}+5)}{4\sqrt{5}} = \frac{4\sqrt{5}}{4\sqrt{5}} = 1$
$F_3 = \frac{\phi^3 - \hat{\phi^3}}{\sqrt{5}} = \frac{(\frac{1+\sqrt{5}}{2})^3 - (\frac{1-\sqrt{5}}{2})^3}{\sqrt{5}} = \frac{(16+8\sqrt{5})-(16-8\sqrt{5})}{8\sqrt{5}} = \frac{16\sqrt{5}}{8\sqrt{5}} = 2$
To find the Fibonacci numbers you sum the previous two Fibonacci numbers. The sequence always begins with 1 so F_1 should equal 1, as it does above. To get F_3, you sum $F_1 + F_2 = 1 + 1 = 2$ which is what the formula yielded for F_3.
Now assume that $F_k = \frac{\phi^k - \hat{\phi^k}}{\sqrt{5}}$ for all values of $k = 1, 2, 3, ..., i-1$.
To prove that $F_i = \frac{\phi^i - \hat{\phi^i}}{\sqrt{5}}$ you need to show the previous two Fibonacci numbers sum to it.
$F_{i-2} + F_{i-1} = F_i = \frac{(\frac{1+\sqrt{5}}{2})^i - (\frac{1-\sqrt{5}}{2})^i}{\sqrt{5}}$
$F_{i-2} + F_{i-1} = \frac{(\frac{1+\sqrt{5}}{2})^{i-2} - (\frac{1-\sqrt{5}}{2})^{i-2}}{\sqrt{5}} + \frac{(\frac{1+\sqrt{5}}{2})^{i-1} + (\frac{1-\sqrt{5}}{2})^{i-1}}{\sqrt{5}}$
$= \frac{(\frac{1+\sqrt{5}}{2})^{i-2} + (\frac{1+\sqrt{5}}{2})^{i-1} + (\frac{1-\sqrt{5}}{2})^{i-2} + (\frac{1-\sqrt{5}}{2})^{i-1}}{\sqrt{5}}$
$= \frac{(\frac{1+\sqrt{5}}{2})^{i-2} * (1 + \frac{1+\sqrt{5}}{2}) + (\frac{1-\sqrt{5}}{2})^{i-2} * (1 - \frac{1+\sqrt{5}}{2})}{\sqrt{5}}$
$= \frac{(\frac{1+\sqrt{5}}{2})^{i-2} * (\frac{1+\sqrt{5}}{2})^2 + (\frac{1-\sqrt{5}}{2})^{i-2} * (\frac{1-\sqrt{5}}{2})^2}{\sqrt{5}}$(by properties of the conjugate)
$= \frac{(\frac{1+\sqrt{5}}{2})^i - (\frac{1-\sqrt{5}}{2})^i}{\sqrt{5}} = F_i$■

3 Linear Data Structures

In this section I will discuss linear data structures including List, Stack, and Queue ADTs, implemented with both dynamically allocated arrays and linked list data structures, including the proper usage and design. I believe I have reached the mastery with distinction level in this area. A linear data structure is a data structure where the elements are stored linearly. This means the only way to get to one element is through the element before it. The main linear ADTs are Lists, Stacks, and Queues. These can be implemented one of two ways, either with dynamically allocated arrays or linked lists. A dynamically allocated array allocates new memory when an instance variable is created, and if it runs out of space allocates more memory elsewhere, copies values over, and deletes the original memory. A linked list uses pointers to memory and to the next element so it does not have to allocate new memory if you add a new item. The benefit of using a linked list is that inserting and removing is generally much more efficient than an array, you just have to switch the pointers around to point to the correct node. Inserting and removing in an array is much less efficient; unless it is the last element, when you are inserting or removing you must recopy many items over to create or close the spot for the element. With both arrays and linked lists, there are many details you must be careful about while implementing. For arrays, you must be careful to keep track of the size and the capacity so you do not try to add an element to a full array. For linked lists, you must be very careful with the pointers so that you do not create a memory leak or cause a segmentation fault which are very common while using pointers. These data structures are useful for storing many types of information, however other types of data structures are more efficient when it comes to sorting or searching.

The first linear ADT is a List. Lists are unordered and indexed. You can implement a List with either a dynamically allocated array or a linked list. I have included excerpts from my code that demonstrates both. When implementing the dynamically allocated array List, you can see in the header file the instance variables including capacity and size. It is important to keep track of these so that throughout your code you can reference them so you do not try to insert something into a full array, or so you know how many items are currently in the list for iteration purposes. In the default constructor that you must allocate memory with the `new` command. Then you must set the capacity and the size of the array. The insert function is rather long; first you must check if the array is full. If it is you must create a new array with more memory. Either way you must copy all of the elements after the index of insertion into the spot over. The remove function is very similar but removes the item at the given position and then copies all of the elements one position to the left. The `operator[]` function is very simple, just `return array[i]`.

The other implementation of List is with a linked list using pointers instead of directly allocating memory for the structure. From the header file you can see that the only instance variables are a Node struct and the head of the list. While there are less variables to keep track of, pointers can be difficult to maintain.

The default constructor is simple, just set `head = NULL`. The `insert` function is as follows; a iterator `ptr` traverses the linked list until the given position at which it resets the pointers of the location to point to the iterator `qtr` which will become the newly inserted node. It also must set the pointer from `qtr` to the node that was originally after `ptr`. I only slightly altered the code from when I originally wrote it to throw an error instead of printing an error message. A more efficient `insert` method would be to just insert at the head each time, but then you cannot specify an index at which to insert. The `remove` function is similar with iterators and you reset the pointers to point from `qtr` (the node before the node to be deleted) to `ptr` (the node after the node to be deleted) and then `delete` the memory for the node to be deleted. The `operator[]` function is longer than that of the array List class, but not difficult. To reach position `n`, the iterator `ptr` traverses through the list `n` times and returns the value on which it ends.

Another linear ADT is a Stack. A Stack follows a last in first out method, where you can only access the item you most recently added to the Stack. This ADT uses the List class so the code is very succinct. No code is needed for the default constructor, when you call it it automatically creates the private variables. The `insert` method simply calls the `insert` method from the List class. There is no `remove` function for a Stack, but a `pop`. This function removes the first item in the Stack and returns it. It calls `remove` from the List class and then returns the value. The `clear` function just calls the function for the List class.

A Queue is also a linear data structure. It follows a first in first out method where you can only access the last element. It can be implemented using a List class and utilizing the `append` and `remove` functions.

Overall, I thoroughly understand this topic, the implementations, and the pros and cons of linear data structures. Since the elements in the structures are organized linearly, the time complexity of most functions are linear. Depending on how you implement them, `insert` and `remove` could be constant (in a linked list), which would then make `push` and `pop` constant for the Stack. However, with the elements organized this way, searching algorithms are not very efficient and the fastest is I DONT KNOW LOOK IT UP

```
//=========================================================================
// linear.cpp
// This file contains excerpts from projects involving linear data structures
// December 2017
//=========================================================================



//=========================================================================
// LIST AS ARRAY
//=========================================================================
template <class T>
class List
{
private:
        T               *array;
        int             DEFAULT_LIST_SIZE = 10;
        int             capacity;
        int     size;

public:

                                List                (void);             //default
                List                (const List<T> &c); //copy constructor
                ~List               (void);             //destructor
        bool            isEmpty                 (void);
        int             length              (void) const;
        T&                      operator[]              (int i);        //index operator
        string          toString            (void)const ;
        void            append              (T c);
        void            insert              (T, int);
        void            remove              (int);
        List<T>         operator+               (const List<T> c) const ;
        List<T>         operator=               (List<T> c) const ;
        void        clear                   (void);
  friend ostream& operator<<    (ostream &os, List<T> c)
  {
    for (int i = 0; i < c.size; i++)
      os << c[i] << " ";

    return os;
  }
};
class IndexError { };

//=========================================================================
// default constructor
//=========================================================================
template <class T>
                List<T>::List   (void)
{
        array = new T[DEFAULT_LIST_SIZE];
        capacity = DEFAULT_LIST_SIZE;
        size = 0;
}

//=========================================================================
// insert
// inserts indicated item at indicated position
//=========================================================================
template <class T>
void    List<T>::insert         (T item, int pos)
{
```

```
        if (pos > size or pos < 0)
        {
                //cout << "Error. Invalid index.\n";
                //exit(1);
    throw IndexError();
        }
        else if (size + 1 > capacity)
        {
                T *oldArray = array;
                capacity *= 2;
                array = new T[capacity];

                for (int i = 0; i < pos; i++)
                        array[i] = oldArray[i];

                array[pos] = item;

                for (int i = pos; i < size; i++)
                        array[i+1] = oldArray[i];

                size += 1;
                delete[] oldArray;
        }
        else
        {

                T *oldArray = array;
                array = new T[capacity];

                for (int i = 0; i < pos; i++)
                        array[i] = oldArray[i];

                array[pos] = item;

                for (int i = pos; i < size; i++)
                        array[i+1] = oldArray[i];

                size += 1;
                delete[] oldArray;
        }

}

//==========================================================================
// remove
// removes item from list at indicated position
//==========================================================================
template <class T>
void    List<T>::remove         (int pos)
{
        if (pos >= size or pos < 0)
        {
                //cout << "Error. Invalid index.\n";
                //exit(1);
    throw IndexError();
        }
        T *oldArray = array;
        array = new T[capacity];

        for (int i = 0; i < pos; i ++)
                array[i] = oldArray[i];

        for (int i = pos; i < size; i++)
```

```
                array[i] = oldArray[i+1];

        size -= 1;
        delete[] oldArray;
}

//========================================================================
// operator []
// returns item in list at indicated index
//========================================================================
template <class T>
T&              List<T>::operator[]     (int i)
{
        if (i >= size or i < 0)
        {
                //cout << "Error. Invalid index\n";
                //exit(1);
                throw IndexError();
        }
        return array[i];
}

//========================================================================
// clear
// removes items from list
//========================================================================
template <class T>
void    List<T>::clear                  (void)
{
        T *oldArray = array;
        array = new T[capacity];
        size = 0;
        delete[] oldArray;
}

//========================================================================
// LIST AS LINKED LIST
//========================================================================

template <class T>
class List
{
private:
        struct Node
        {
                T     data;
                Node *  next;
        };

        Node *head;

public:
                                List            (void);                                 //defa
ult constructor
                        List            (const List<T> &src);   //copy constructor
                        ~List           (void);                                 //destructor
  T&                    operator[]              (int x);
  List<T>               operator+       (const List<T> &l);
  List<T>               operator=       (const List<T> &src);
  bool                  isEmpty         (void);
  int                   length          (void);
  string                toString                (void);
  void                  append          (T x);
```

```
  void                  insert          (T i, int p);
  void                  remove          (int p);
  void                  clear           (void);

  friend ostream & operator<< ( ostream &os, List<T> &l)
  {
        Node *ptr = l.head;
        while (ptr != NULL)
        {
        os << ptr->data << " ";
        ptr = ptr->next;
        }
        return os;
  }

};
class IndexError { };
//========================================================================
//default constructor
//creates an empty list
//========================================================================
template <class T>
        List<T>::List (void)
{
        head = NULL;
}

//========================================================================
// insert
// inserts an item at the given position
//========================================================================

template <class T>
void    List<T>::insert (T i, int p)
{
        if (p > length() || p < 0)
        {
                //cout << "Error: invalid index.\n";
                //exit(1);                                              //prints error
 message and exits program
    throw IndexError();
        }
        Node *ptr = head;
        int count = 0;
        if (p == 0)
        {
                Node *qtr = new Node;
                qtr->data = i;
                qtr->next = ptr;
                head = qtr;
        }
        else
        {
                while (count < (p-1))                           //need to stop one before desi
red index to
                {                                                                       //make
 new node in the right place
                        ptr = ptr->next;
                        count++;                                                        //goes through
 list until desired index
                }
                Node *qtr = new Node;                           //sets new node equal to given
 item and
```

```
                qtr->data = i;                                          //pointing to next nod
e
                qtr->next = ptr->next;
                ptr->next = qtr;                                        //creates link to new
node
        }
}


//========================================================================
// remove
// removes item at given index
//========================================================================

template <class T>
void    List<T>::remove (int p)
{
        if (p >= length() || p < 0)
        {
                //cout << "Error: invalid index.\n";
                //exit(1);                                      //prints error message and qui
ts program
    throw IndexError();
        }
        Node *ptr = head;
        Node *qtr = head;
        Node *rm;
        int count = 0;
        if (p == 0)
        {
                rm = ptr;
                delete rm;
                head = ptr->next;
        }
        else
        {
                while (count < (p-1))
                {
                        ptr = ptr->next;                        //goes through list until inde
x right before p
                        qtr = qtr->next;
                        count++;
                }
                ptr = ptr->next;
                rm = ptr;
                ptr = ptr->next;
                qtr->next = ptr;
                delete rm;

        }
}

//========================================================================
//operator[]
//returns item in list at given index
//========================================================================

template <class T>
T&      List<T>::operator[]     (int x)
{
        if (x >= length() || x < 0)
        {
                // << "Error: Invalid index.\n";
```

```
                //exit(1);                                          //prints error if invalid inde
x
                throw IndexError();
        }
        int count = 0;
        Node *ptr = head;
        while (count < x)
        {
                ptr = ptr->next;                        //goes through list until it reaches g
iven index
                count++;
        }
        return ptr->data;                               //returns value at index
}

//=========================================================================
// clear
// deletes the links from the list
//=========================================================================
template <class T>
void List<T>::clear (void)
{
        Node *ptr, *qtr;
        ptr = head;
        qtr = head;
        while (ptr != NULL)
        {
                qtr = ptr->next;                        //sets node equal to item after ptr
        delete ptr;                                     //deletes ptr
        ptr = qtr;                                      //resets ptr to item originally after
        }
        delete ptr;                                             //deletes head
        head = NULL;
}

//=========================================================================
// STACK
//=========================================================================

template <class T>
class Stack
{
private:
        List<T> list;



public:
                        Stack                   (void);
                        ~Stack                  (void);
                        Stack                   (const Stack<T> &);
        void            push                    (T);
        T                       pop                                     (void);
        T                       peek                    (void);
        int                     length                  (void) const;
        bool            isEmpty                 (void) const;
        void            clear                   (void);


        Stack<T>        operator=               (const Stack<T> &);

        friend ostream & operator<< (ostream &os, const Stack<T> &s)
        {
```

```
                os << s.list;
                return os;
        }

};
//=============================================================================
// default constructor
//=============================================================================
template <class T>
                Stack<T>::Stack         (void)
{
        //secretly creates private variables, no code needed
}

//=============================================================================
// insert
//=============================================================================
template <class T>
void            Stack<T>::push          (T item)
{
        list.insert(item, 0);
}
//=============================================================================
// remove
//=============================================================================
template <class T>
T               Stack<T>::pop           (void)
{
        T temp = list[0];
        list.remove(0);
        return temp;
}
//=============================================================================
// clear
//=============================================================================
template <class T>
void    Stack<T>::clear         (void)
{
        list.clear();
}
```

4 Priority Queues and Binary Heaps

In this section I will discuss Priority Queues and Binary Heaps including the proper usage and design of a priority queue ADT implemented with a binary heap. I think I have reached the mastery level with this section; I fully understand binary heaps including their properties and accompanying algorithms. I also understand how to implement a priority queue with a binary heap and why that is efficient. A binary heap is an array object that can be viewed as organization of nodes where each node has at most two children, or a binary tree. A heap is either a max-heap or a min-heap. In a max-heap, each element is larger than both of its children, thus the root is the largest element in the heap. Conversely, in a min-heap, each element is smaller than its children and the root is the smallest element in the heap. There are several different parts to a binary heap; root, children, parent, leaf, and height. The root is the largest or smallest (depending on the implementation) element of the heap and array. The children of a node are the nodes it points to, and the parent of a node is the node that points to it. A leaf is a node with no children. The height of a node is the length of the longest path from that node to a leaf and the height of a heap is the height of the root. These terms are all necessary to understand for the algorithms associated with heaps. Heaps are actually implemented with an array, with pointers pointing to the proper position in the array instead of nodes like a binary search tree. As you can see in the code below, to find the child or parent, you just return some function of the given index because each parent or child is the same distance away from itself for every node. For simplicity purposes, I will describe a minimum heap throughout this section, but the same processes apply for maximum heaps with just a few changes. A min-heap must always maintain the min-heap property, which states that the root is the smallest element in the heap and the child of every node is larger than the node itself. In the code you can see the MinHeap has a few instance variables including an array, size, and capacity. Since it uses a dynamically allocated array, you must track the size and capacity so you do not try to insert an element into a full array. Also included in the code below are the accompanying algorithms that help to maintain the heap properties. The first is `buildHeap` which is called on an object and takes no parameters. This algorithm starts at the last element that is not a leaf and calls `heapify` on each element travelling back to the root. `heapify` is a function that fixes a heap so it satisfies the properties of a min-heap again. It has a precondition that the left and right children of the root are roots themselves of a min-heap. It finds the left and right children and then finds the minimum between the three elements. If one of the children is smaller than the root, the algorithm swaps it with the root and calls heapify recursively on the new root until the min-heap property is satisfied. The `swap` function is also included, it is a simple swap between indices of an array. The `buildHeap` starts calling `heapify` at the bottom and travelling back up towards the root because of the precondition for `heapify`. Since the children of the node on which you call `heapify` must be MinHeaps themselves, the only guaranteed MinHeap inside every single MinHeap are the leaves. Therefore, `buildHeap` starts by calling

at the bottom and travels up. Another accompanying algorithm for MinHeaps is `heapSort`. This algorithm takes a MinHeap as input. It takes the root and copies it into an array. Then it swaps the root with the largest element in the heap so that when you decrement `heapSize` the previous root is no longer in the MinHeap. Then it calls `heapify` on the new root to fix the heap so that it will maintain the min-heap properties. Question 5 on the first exam (included below) asked to show how `heapSort` works on a MaxHeap.

Since a min-heap always has the smallest element at the root, it can be used to implement a Priority Queue. The Priority Queue ADT allows a few operations including `extractMin, decreaseKey` and `insert`. To implement `extractMin` you swap the smallest element (the root) with the largest element so that when you decrease heapSize, the smallest element is no longer in the MinHeap or the Priority Queue. After you decrement heapSize, you call `heapify` on the new root so that the heap still maintains the min-heap properties and then return `A[heapSize]` which is the previous root. Since you can only access the smallest element in a Priority Queue, if you want a different element to come out first you need to use `decreaseKey`. If the given index is valid, you set the value at the index to the given key and the perform `swap` until the array is in min-heap order again. If you want to add a new value to the Priority Queue you use `insert`. If the heap is not full, you increment `heapSize`, set the last index equal to the given key, and then call `decreaseKey` to sort the new key into the correct place.

One use for a Priority Queue is Huffman Coding. Huffman Coding is a more efficient way to store a file which creates new compression codes depending on the frequency of the character. After reading in the text file you can sort the characters depending on the frequencies then insert them into a Priority Queue. Then call `extractMin` to pull out the characters and create the codes depending on the order in which they are removed. This algorithm can compress files much more efficiently because it creates shorter compression codes for the more frequent characters which appear more often than the longer compression codes.

Min-heaps have much more efficient running times than linear data structures with `insert` and `remove` running in $\Theta(n)$ each. `heapSort` runs in $\Theta(nlogn)$ time since `buildHeap` takes $\Theta(n)$ and each of the $n-1$ calls to `heapify` takes $\Theta(logn)$.

Overall, I feel as though I have a very comprehensive understanding of the binary heap structure and its application to Priority Queue. I understand the concept of the Huffman Coding but the code itself is a little confusing, although with review and practice I am confident I would be able to recreate it.

```
//=============================================================
// heapPQ.cpp
// This file contains excerpts from the MinHeap class,
// Priority Queue class, and Huffman Coding.
//=============================================================

//=============================================================
// MINHEAP
//=============================================================

template <class KeyType>
class MinHeap
{
  public:
    MinHeap(int n = DEFAULT_SIZE);
    MinHeap(KeyType initA[], int n);
    MinHeap(const MinHeap<KeyType>& heap);
    ~MinHeap();

    void heapSort(KeyType sorted[]);

    MinHeap<KeyType>& operator=(const MinHeap<KeyType>& heap);
    std::string toString() const;

  private:
    KeyType *A;     // array containing the heap
    int heapSize;   // size of the heap
    int capacity;   // size of A

        void heapify(int index);
    void buildHeap();
        int leftChild(int index) { return 2 * index + 1; }
        int rightChild(int index) { return 2 * index + 2; }
        int parent(int index) { return (index - 1) / 2; }
    void swap(int index1, int index2);
    void copy(const MinHeap<KeyType>& heap);
    void destroy();
};


//=============================================================
// buildHeap
// Builds a MinHeap
// Pre-Conditions:
//              none
// Post-Conditions:
//              the heap is definitely a Min-Heap
//=============================================================
template<class KeyType>
void  MinHeap<KeyType>::buildHeap()
{
        heapSize = capacity;
        for(int i = heapSize / 2 - 1; i >= 0; i--)
        {
                heapify(i);
        }
}

//=============================================================
// heapify
// Makes a heap into a min heap
// Pre-Conditions:
//              Both children must be roots of a Min-Heap
```

```
// Post-Conditions:
//              The heap is a Min-Heap (if the
//              Pre-Condition is satisfied)
//=============================================================
template<class KeyType>
void  MinHeap<KeyType>::heapify(int index)
{
        int l = leftChild(index);
        int r = rightChild(index);
        int min;
        if(l < heapSize && *(A[index]) > *(A[l]))
                min = l;
        else
                min = index;
        if(r < heapSize && *(A[min]) > *(A[r]))
                min = r;
        if(min != index)
        {
                swap(index, min);
                heapify(min);
        }
}


//=============================================================
// swap
// Swaps two items
// Pre-Conditions:
//              The indices are valid
// Post-Conditions:
//              The values at the indices
//              have been swapped
//=============================================================
template<class KeyType>
void  MinHeap<KeyType>::swap(int index1, int index2)
{
        KeyType* temp = A[index1];
        A[index1] = A[index2];
        A[index2] = temp;
}



//=============================================================
// heapSort
// Pre-Conditions:
//              The heap must be a MinHeap
// Post-Conditions:
//              sorted is now sorted in ascending order
//=============================================================
template<class KeyType>
void MinHeap<KeyType>::heapSort(KeyType* sorted[])
{
        sorted = new KeyType*[capacity];
        //buildHeap();
        for(int i = capacity - 1; i >= 0; i--)
        {
                sorted[i] = A[0];
                swap(0,i);
                heapSize--;
                heapify(0);
        }
        heapSize = capacity;
```

```
}

//============================================================
// PRIORITY QUEUE
//============================================================

//============================================================
// extractMin()
// removes and returns minimum value
// Pre-Conditions:
//              Calling object must be a minimum
//              priority queue or empty
// Post-Conditions:
//              heapSize will be one less (if not originally empty)
//              and will be a minimum priority
//              queue
//============================================================

template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::extractMin ( void )
{
        if(!empty())
        {
                swap(0,heapSize-1);     //swaps smallest value to end of pq
                heapSize--;
                heapify(0);
                return A[heapSize];                             //returns smallest value
        }
        else
        {
                return NULL;                            //returns NULL if pq is empty
        }
}

//============================================================
// decreaseKey(int index, KeyType* key)
// decreases the key of the given index to the given key
// Parameters:
//              int index               - the index to be decreased
//              KeyType* key    - the new key to be assigned
// Return Value: N/A
// Pre-Conditions:
//              The calling object must be a priority queue
// Post-Conditions:
//              The key of the given index (in the original pq)
//              will be decreased and the calling
//              object will be a minimum priority queue
//============================================================

template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey     (int index, KeyType* key)
{
                if(*key > *A[index])
                {
                        throw KeyError();                       //index is out of bounds

                }else{
                        A[index] = key;
                        while(index > 0 && *A[index] < *A[parent(index)])
                        {
                                swap(index, parent(index));     //swaps index with
                                index = parent(index);          //its parent
                        }
```

```
                }
}

//=============================================================
// insert(KeyType* key)
// inserts the given key into the minimum priority queue
// Parameters:
//              KeyType* key - the key to be inserted into the min
//              priority queue
// Return Value: N/A
// Pre-Conditions:
//              The calling object must be a minimum priority queue
// Post-Conditions:
//              The minimum priority queue will now contain the
//              given key and be a minimum priority queue
//=============================================================

template <class KeyType>
void MinPriorityQueue<KeyType>::insert (KeyType* key)
{
        if(heapSize == capacity)
        {
                throw FullError();                                      //if heap is full
        }else{
                heapSize++;
                A[heapSize - 1] = key;          //inserts key at end of array
                decreaseKey(heapSize - 1, key);//sorts key into correct
        }                                                                               //plac
e

        //sorted[heapSize] = key;
        //heapSize++;
        //this->heapSort(sorted);
}


//=============================================================
// HUFFMAN CODING
//=============================================================

//=============================================================
// creates a min priority queue from a text file
//=============================================================
MinPriorityQueue<Node> fileToMPQ ( string fileName, map<char,
        int> &frequencies )
{
        ifstream file(fileName.c_str());

        char c;
        int counter;
        if(file.is_open())
        {
                while(file.get(c))
                {
                        counter++;
                        frequencies[c]++;
                }
        }

        frequencies.erase(frequencies.begin());
        MinPriorityQueue<Node> nodes(frequencies.size());

        for(map<char, int>::iterator it = frequencies.begin();
```

```
        it != frequencies.end(); it++)
        {
                cout << it->first << " " << it->second << endl;
                Node c;
                c.character = it->first;
                c.freq = it->second;
                nodes.insert(&c);
                //string temp = nodes.toString();
                //cout << temp << endl;
        }

        return nodes;
}
//============================================================
// creates a huffman tree from a priority queue
//============================================================
Node buildHuffmanTree ( map<char, int> frequencies,
        MinPriorityQueue<Node> &nodes )
{
        int n = frequencies.size();
        for(int i = 0; i < n - 1; i++)
        {
                Node* z = new Node;
                z->left = nodes.extractMin();
                z->right = nodes.extractMin();
                if(z->left != NULL)
                        z->freq += z->left->freq;
                if(z->right != NULL)
                        z->freq += z->right->freq;
                z->character = '\0';
                nodes.insert(z);
        }
        Node *root = nodes.extractMin();
        Node x = *root;
        return x;
}
//============================================================
// creates huffman codes from huffman tree
//============================================================
void searchHuffmanTree ( map<char, int> frequencies, map<char,
        string> &huffCodes, Node* z, Node* root, string &s )
{
        char c;
        if(z->isLeaf())
        {
                c = z->character;
                huffCodes[c] = s;
        }
        else{
                if(z->left != NULL)
                {
                        s = s + "0";
                        searchHuffmanTree(frequencies, huffCodes, z->left, root, s);
                }
                if(z->right != NULL)
                {
                        s = s + "1";
                        searchHuffmanTree(frequencies, huffCodes, z->right, root, s);
                }
        }
        s = s.substr(0,s.size() - 1);
}
```

5 Graphs

In this section I will discuss graphs, ways to implement them, and the important algorithms associated with them. I think that I have reached the mastery level in this section. I have now had three semesters of dealing with graphs in computer science and have even learned about them in my math class. I fully understand the structure and organization of them, as well as most of the algorithms associated with them. A graph is a collection of vertices and edges which connect them. It can be directed or undirected, connected or unconnected, and cyclic or acyclic. The edges in a directed graph have direction and are pointing to a specific vertex while the edges in an undirected graph go in both directions. In a connected graph, there is a path to every vertex from any vertex, and in an unconnected graph that path does not exist for each vertex. If a graph is cyclic, there is a cycle and if there is not a cycle it is acyclic. The two main ways to implement a graph ADT are an adjacency matrix or an adjacency list. An adjacency matrix is a square matrix consisting of all the vertices and if there is a path from vertex v to vertex u the vu entry in the matrix contains the weight of the path or 0 if there is no path. An adjacency list is a list of lists where each element in the first list contains a vertex v and linked to that is another list of all the vertices it is connected to. Question 1 on exam 3 (included below) asked for an adjacency matrix and adjacency list representation for a given graph. If the graph is dense, meaning there are a lot of edges, a search on both of these implementations have about the same time complexity, $\Theta(n^2)$. In the matrix implementation, you have to traverse through every element in the matrix in the worst case, which is n^2. In the list implementation, you have to traverse through every element and its respective list of connected vertices, which in the worst case of a dense graph, is also n^2. However, if the graph is sparse (few edges), the list implementation is much more efficient. Even if there are not many edges, a matrix still has a entry in each spot in the matrix (0 for no path, or x for the weight of the path). Thus a search on this matrix in the worst case will still take $\Theta(n^2)$. But in a list implementation, if there are fewer edges, the respective lists of the vertices will be much shorter than n and take less than $\Theta(n2)$ time. In the code below, we use a list implementation.

The DFS and BST algorithms are two of the most common algorithms for graphs. Breadth First Search traverses through the graph from a specified source node and travels out one level at a time until there are no more nodes reachable from the source. Question 5 on exam 3 (included below) asked for psuedocode to perform BFS on a graph and determine if it is connected. The BFS algorithm starts by creating a priority queue. Then you travel to each vertex in the graph and set the color to white (implying it is undiscovered), the distance to ∞, and the predecessor to NULL. Then it enqueues the source vertex to the priority queue. While the priority queue is not empty, it dequeues a vertex from the priority queue and travels through each element in its adjacency list. If a vertex in its adjacency list is undiscovered/white, it changes the color to gray (to signify it has been discovered but not finished) and enqueues it to the priority queue. It sets the predecessor to the vertex we had dequeued and the distance

to the distance of the dequeued vertex plus 1. After each vertex in the dequeued vertex's adjacency list has been visited, it turns that node black to signify that level is complete and it repeats until the priority queue is empty (aka all of the reachable nodes have been discovered and finished). To determine if the graph is connected, I put this BFS algorithm in a for loop to call on every vertex in the graph. Before it repeats the call on a different source vertex, it checks the nodes in the graph. If any of the vertices have a predecessor of NULL, it means it was not reached by that call to BFS and the graph is not connected, or else it would have been reached. It then returns false and the function is over. Depth First Search traverses through the graph, depth first, and records the discovery time, the finish time, and the predecessor for each node. In the code below, it sets all of the vertices to white. It then traverses through each vertex in the graph and if it is undiscovered/white, it calls the helper function DFSVisit. This function sets the color of the given vertex to gray (discovered but not finished) and the discovery time to the counter variable `timee`. Then for each vertex in the given vertex's adjacency list, if it discovers a vertex that is already gray, it sets the `bool` cycles to true because that indicates there is a cycle. If it encounters a vertex that is white, it sets the predecessor of that vertex to the given vertex and performs a recursive call on that vertex. After the recursive calls are finished, it sets the vertex to black and the finished time to `timee`. Question 2 on exam 3 (included below) asked for a visualization of both BFS and DFS on given graphs. If you break both of these algorithms down, their time complexity is easy to calculate. BFS performs V enqueues and dequeues, which both run in $O(1)$, with V being the amount of vertices. It also scans the length of each adjacency list and the sum of the adjacency lists is at most E or the amount of edges so that runs in $O(E)$. Initialization of the graph also takes $O(V)$ so the total running time of BFS is linear to the adjacency list or $O(V + E)$. Excluding the calls to DFSVisit, DFS runs in $\Theta(V)$ time because each of the for loops are $O(1), V$ times. The main loop in DFSVisit iterates over the size of the adjacency list so it runs in $\Theta(E)$ thus the entire algorithm runs in $\Theta(V + E)$ time.

Other algorithms that are fundamental to graphs are Prim's, Kruskal's, Bellman-Ford, and Dykstra's. These algorithms can find the minimum spanning trees and shortest paths in a graph. Prim's algorithm maintains a priority queue where it inserts all of the vertices and sorts them by key. The key attribute is keeping track of the weight from the predecessor node to itself, and all of the keys, except for the root, are set to ∞ initially. The predecessors are all set to NULL. While the priority queue is not empty, it takes the minimum weight vertex and traverses through its adjacency list. It compares the keys of those nodes to the weights of the edge and if the other edge's weight is less than the key then it resets the key of the vertex. It repeats this until the priority queue is empty thus all of the vertices have been visited. If you trace each vertex back through it's predecessor then you can find the minimum spanning tree. Kruskal's algorithm also finds the minimum spanning tree. It puts all of the edges into a disjoint set and sorts them by non-decreasing weight. It then traverses through this set and adds the minimum-weight edges in non-decreasing

order if they connect two separate vertices, meaning at least one of the vertices is not already in the tree. The diagram included below illustrates both Kruskal's and Prim's and the order in which they run on the same graph, producing the same output.

Bellman-Ford and Dykstra's algorithms find shortest paths. Bellman-Ford goes through a graph and sets each vertex's predecessor to NULL and distance to ∞. It then goes through a loop $n-1$ times and checks if the distance from one node to another can be made smaller by going through a different path. The first time the loop goes through all of the distances will change because any weight is smaller than infinity. The next $n-2$ times the loop iterates, it compares the current distance to the distance it would have if it travelled through a different vertex. It finishes by checking for negative weight cycles. No algorithm can correctly handle negative weight cycles because by nature the weight will keep minimizing each time you go around the cycle. However, unlike Dykstra's, Bellman-Ford can detect these and returns false if there is a negative weight cycle. Dykstra's is similar to Prim's algorithm, it begins by traversing the graph and setting the distances to ∞ and the predecessors to NULL. It then creates a priority queue of all the vertices. It extracts the smallest weight vertex and if the distance through that edge is less than the current distance it updates the distance. It also creates a set S which is used for proofs. Bellman-Ford runs in $\Theta(nm)$ where n is the number of vertices and m is the number of edges. Dykstra's is much faster, running in $\Theta((m+n)logn)$. However, if there is a chance you will have a negative weight cycle you should use Bellman-Ford's or else your program will crash.

Overall, graphs are a concept I understand and with more practice I will be able to have a more firm grasp on some of the accompanying algorithms.

```
//=============================================================
// graph.cpp
// This file contains excerpts from the Graph, Vertex
// and Edge classes.
//=============================================================

//=============================================================
// VERTEX
//=============================================================

class Vertex
{
public:
  int id;
  Vertex* pred;
  List<Vertex> adj_list;
  char color;
  int disc;
  int fin;
  int key;

  bool    operator<     (const Vertex &x) {return this->key <= x.key;}
  bool    operator>     (const Vertex &x) {return this->key > x.key;}
  bool    operator==    (const Vertex &x) {return this->id == x.id;}

  Vertex& operator= (const Vertex& v);

          Vertex        (int name);
          Vertex        (Vertex* v);
          ~Vertex       (void);
private:
  friend ostream& operator<< (ostream& os, const Vertex& v)
  {
    os << "Vertex ID: " << v.id << endl;
    return os;
  }
};

//=============================================================
// Default Constructor
//=============================================================
Vertex::Vertex(int name)
{
  id = name;
  pred = NULL;
  color = 'w';
  disc = INT_MAX;
  fin = 0;
  key = INT_MAX;
}

//=============================================================
// Assignment Operator
//=============================================================
Vertex& Vertex::operator= (const Vertex& v)
{
  this->id = v.id;
  this->pred = v.pred;
  this->adj_list = v.adj_list;
  this->color = v.color;
  this->disc = v.disc;
  this->fin = v.fin;
```

```
  return *this;
}
//============================================================
// EDGE
//============================================================
class Edge
{
public:
  int u;    //start Vertex
  int v;    //end Vertex
  int weight;     //weight (u,v)

  Edge()
  {
    u = -1;
    v = -1;
    weight = 0;
  }

  ~Edge()
  {
  }
};
//============================================================
// GRAPH
//============================================================
class Graph
{
public:
          Graph         (string filename);
          Graph         (const Graph& g);
          ~Graph        (void);
  Graph   operator=     (const Graph& g);
  void    dfs           (void);
  bool    cycle         (void);
  void    Prim          (int root);


private:
  List<Vertex>  graph;
  List<Edge>    edges;
  bool          cycles = false;

  void    dfs_visit       (Vertex &u, int timee);
  bool    pqHelp          (Vertex s, MinPriorityQueue<Vertex> pq);
  int     findEdge        (Vertex u, Vertex v);
};

//============================================================
//default constructor
//Pre-Condition:
//  -file with matrix representation of a graph
//Post-Condition:
//  -a graph
//============================================================
Graph::Graph  (string filename)
{
  ifstream file;
  file.open(filename);                              //open file
  string line;
  getline(file, line);
  istringstream buffer(line);                 //read in number of
  int num_vert;                                       //vertices
```

```
  buffer >> num_vert;
  for (int i = 0; i < num_vert; i++)
  {
    Vertex *v = new Vertex(i);
    graph.append(v);                //append all vertices to graph
  }
  for (int j = 0; j < num_vert; j++)               //iterates rows
  {
    int srch_pt = 0;
    getline(file, line);
    for (int k = 0; k <num_vert; k++)
    {
      int space = line.find(" ", srch_pt);   //read in connections
      int weight;
      istringstream buffer (line.substr(srch_pt, space-srch_pt));
      buffer >> weight;
      srch_pt = space+1;

      if (weight != 0)
      {
        Edge *e = new Edge();                    //creates edges
        e->u = graph[j]->id;
        e->v = graph[k]->id;
        e->weight = weight;
        edges.append(e);
        graph[j]->adj_list.append(graph[k]);
      }
    }
  }
  //cout << graph;
  file.close();
}

//=============================================================
//dfs - depth first search
//Pre-Conditions
//  -graph
//Post-Conditions
//  -traversed graph
//=============================================================
void Graph::dfs (void)
{
  if (graph.length()==0)                //throw error if empty
    throw EmptyError();
  for (int i = 0; i < graph.length(); i++)
  {
    graph[i]->color = 'w';                 //set all to white
  }
  int timee = 0;
  //cout << timee << endl;
  for (int i = 0; i < graph.length(); i++)
  {
    if (graph[i]->color == 'w')
    {
      dfs_visit(*(graph[i]), timee);            //visit vertex
    }
  }
}
//=============================================================
//dfs_visit - depth first search
//=============================================================
void Graph::dfs_visit (Vertex& u, int timee)
{
```

```
  timee += 1;
  u.disc = timee;
  u.color = 'g';                                    //discover
  cout << "Visiting: " <<u << endl;
  for (int i = 0; i < u.adj_list.length(); i++)
  {
    if (u.adj_list[i]->color == 'g')
    {
      cycles = true;
    }
    if (u.adj_list[i]->color == 'w')
    {
      u.adj_list[i]->pred = &u;              //set predecessor
      dfs_visit(*u.adj_list[i], timee);//visit adjacent vertices
    }
  }
  u.color = 'b';               //all adjacent vertices visited
  timee += 1;
  u.fin = timee;
}

//=============================================================
//Prim's algorithm -- we tried but it's not happening
//Pre-Conditions
//  -undirected weighted graph
//Post-Conditions
//  -MST of graph
//=============================================================
void Graph::Prim (int root)
{
  MinPriorityQueue<Vertex> pq;                  //create min pq
  for (int i = 0; i < graph.length(); i++)
  {
    if (graph[i]->id != graph[root]->id)
    {
      graph[i]->key = INT_MAX;
      graph[i]->pred = NULL;
      pq.insert(graph[i]);             //insert vertices into pq
    }
  }
  graph[root]->key = 0;
  pq.insert(graph[root]);                  //insert root into pq
  cout << pq << endl;

  while (!pq.empty())
  {
    Vertex *u = pq.extractMin();          //find minimum weight
    cout << "MIN: " << u->id << endl;
    for (int j = 0; j < u->adj_list.length(); j++)
    {
      Vertex *v = u->adj_list[j];
      //cout << "weight of " << *v <<" " <<  v->key << endl;
      if (v->key > u->key && findEdge(u,v) < v->key)
      {

        v->pred = u;                        //update pred and key
        v->key = findEdge(u,v);
        //cout<<"reset key of "<<v->id<<"to "<<v->key<<endl;
      }
    }
  }
  for (int k = 0; k < graph.length(); k++)
  {
```

```
    cout << k << " weight: " << graph[k]->key <<graph[k]->pred <<endl;
  }
}
```

6 Hash Tables

In this section I will discuss hash tables including how to implement a dictionary using a hash table, as well as hash functions and their importance in relation to universal hashing. I think I have reached the mastery level in this area. I understand the concept of a hash table and hash functions and how to use them in coding. While on the project we could not come up with a very efficient hash function, I understand the properties of an efficient function and what to avoid. A hash table is a data structure that optimizes a linked list, and in the best cases allows for searching in nearly constant time. It consists of an array and each entry in the array is the head of another linked list. There is an accompanying hash function which takes the element which you are inserting, and calculates a slot to enter it in. This function should be seemingly random and avoid patterns, but it also needs to give the same output if you enter the same input each time. If the hash table allows for chaining, the item will go to the slot calculated by the hash function, and if is empty it will insert it there. If there is already an element in the slot it will traverse to the end of the linked list and insert there. If the hash table does not allow for chaining it uses open addressing and it will first hash to the slot calculated by the hash function. If that slot is filled it will then use a probing method to find another slot in which to insert. Linear probing just goes from the calculated slot to the next slot and so on until it finds an open place. Quadratic probing produces another slot in which to attempt to insert. Double hashing is sometimes used to minimize chances of two inputs having the same output. Double hashing consists of going through two separate hash functions to calculate the slot to insert. With chaining, the worst case time-complexity for searching could be $O(n)$ if all of the elements hash to the same slot. However with simple uniform hashing both a successful and unsuccessful search can happen in $\Theta(1 + \alpha)$ where α is the number of elements examined in an unsuccessful search.

Better hash functions can minimize the number of elements chained to the same slot. There are several methods that are known to work and many which you should avoid. Some examples of methods you should avoid are ones that can have very obvious patterns. For example, if you are hashing strings, you should not hash by the first letter of the string because some letters are far more common than others, leading to an uneven hash table. Some methods that are known to work (although not well) are the multiplication and division methods. The division method takes a key k and maps it to $k \mathrm{mod} m$. It is best to avoid certain values of m including powers of 2. The best value for m to use is a prime number not too close to a power of 2 but close to the size of your table. The multiplication method is known for hashing as well. You first multiply the key k by a constant C, where C is inbetween 0 and 1, and mod by 1 to get the fractional portion. Then multiply by m and round down. For this method, powers of 2 are acceptable for m. The value for C is best if you use the Golden Ratio $\frac{\sqrt{5}-1}{2}$. An even better way to ensure you are calculating the most random (not actually random) value is to use Universal Hashing. In universal hashing there is a universal class of hash functions where they are chosen independently

of how they are going to be stored. In open addressing, if an element is already in the slot to which the element is hashed to, it uses a probing method to find another slot until one is available.

Question 8 on exam 2 (included below) asked to show visually how elements would be inserted into hash tables with and without chaining. Looking at the code below, the Hash Table class is not very difficult to implement using a linked list class. The default constructors allocate new memory with the appropriate number of slots. The get function uses the hash function (which is assumed to be in the class of KeyType) to find the slot of the item and returns the value located there. Insert uses the hash function to calculate the slot where the value should go and inserts it there. Remove uses the hash function to find the location at which to remove the value. Implementing a Dictionary class is very easy using a hash table, the class inherits HashTable and all of the functions are copied over inherently. A hash table is good for a dictionary because the keys are hashed to a certain slot and then when they need to be accessed the time-complexity, assuming you use a good hash function, is much lower than a linked list implementation.

Overall, I fully understand the concept of a hash table and how hash functions are used. I also understand the difference between chaining and universal hashing, a good and bad hash function, and how a hash table can be used to implement a Dictionary ADT.

```
//=============================================================
// Hash.cpp
// This file contains exceprts from the Hash Table class and
// Dictionary class.
//=============================================================

//=============================================================
// HASH TABLE
//=============================================================
template <class KeyType>
class HashTable
{
public:
                                          HashTable       (int numSlots);
                                          HashTable       (const HashTable<KeyType>& h);
                                          ~HashTable      ();
        KeyType*                    get                   (const KeyType& k) const;
        void                        insert          (KeyType *k);
        void                        remove          (const KeyType& k);
        HashTable<KeyType>& operator=   (const HashTable<KeyType>& h);
        std::string toString                              (int slot) const;

private:
        int                     slots;
        List<KeyType>   *table; // an array of List<KeyType>â\200\231s
};

//=============================================================
// default constructor (unspecified slots)
//=============================================================
template <class KeyType>
    HashTable<KeyType>::HashTable(void)
{
  slots = 10;
  table = new List<KeyType>[slots];
}

//=============================================================
// default constructor (specified slots)
//=============================================================
template <class KeyType>
    HashTable<KeyType>::HashTable(int numSlots)
{
    slots = numSlots;
    table = new List<KeyType>[slots];
}
//=============================================================
// get
// this method returns a pointer to the object in the hash
// table where the value resides
// parameters: const KeyType& k for which to find
// return value: KeyType* location
//=============================================================
template <class KeyType>
KeyType* HashTable<KeyType>::get(const KeyType& k) const
{
  int index = k.hash(slots);
  return table[index].get(k);
}

//=============================================================
// insert
// this method inserts a value into the hash table
```

```
// parameters: KeyType* k to insert
// return value: void
//=============================================================
template <class KeyType>
void HashTable<KeyType>::insert(KeyType* k)
{
  int slot = k->hash(slots);
  table[slot].insert(0,k);
}

//=============================================================
// remove
// this method removes a given value from the hash table
// parameters: const KeyType& k to remove
// return value: void
//=============================================================
template <class KeyType>
void HashTable<KeyType>::remove(const KeyType& k)
{
  int slot = k.hash(slots);
  table[slot].remove(k);
}
//=============================================================
// DICTIONARY
//=============================================================
template <class KeyType>
class Dictionary: public HashTable<KeyType>
{
/*
public:

              Dictionary  (){};          //default constructor
              ~Dictionary (void){};    //destructor
              Dictionary  (const Dictionary<KeyType> &d){};
  void        insert      (KeyType *k)
  {
    HashTable<KeyType>::insert(k);
  };
  void        remove      (const KeyType &k)
  {
    HashTable<KeyType>::remove(k);
  };
  KeyType *   get         (const KeyType &k) const
  {
    HashTable<KeyType>::get(k);
  };
  bool        Empty       (void)  const
  {
    HashTable<KeyType>::Empty();
  };

private:
  HashTable<KeyType> *d;
*/

public:

        Dictionary():HashTable<KeyType>(){};
        ~Dictionary(void) {};
Dictionary(const Dictionary<KeyType> &d):HashTable<KeyType>(d){};

//Inherited HashTable Class incorrectly, much simpler this way.
};
```

7 Binary Search Trees

In this section I will discuss Binary Search Trees, how to use them to implement a Dictionary ADT and the implementation of red-black trees. I think I have reached mastery with distinction in this area. I fully understand the concept of a binary search tree and a red black tree and the accompanying algorithms and the properties by which they are structured. I am very confident in my understanding of this topic and performed extremely well on the tests and exams covering these topics. A binary search tree is a binary tree that upholds the property that the left child of an element is less than the element itself and the right child of an element is greater than the element itself. Every node has at most two children. Similar to binary heaps, there are several definitions that are helpful when using binary search trees including root, children, parent, predecessor, successor, leaf, and height. Most of these are the same except predecessor and successor. The predecessor of a node is the element that comes immediately before it in sorted order, and the successor of a node is the element which comes immediately after it. If the element has a left subtree, the predecessor is the greatest item in the left subtree because it is the biggest element that is still smaller than the element itself. If the element does not have a left subtree, the predecessor is found by travelling up the tree to the right and then up one more level to the left. Similarly, if the element has a right subtree, the successor is the smallest element in the right subtree because it is the smallest element that is still bigger than the element itself. If the element does not have a right subtree, the successor is found by traversing up the tree and to the left as far as possible and then up one more level to the right. The predecessor and successor are used when an item is being removed from a binary search tree. Question 2 on exam 2 (included below) asked to prove that a complete binary tree with height h has $2^h - 1$ internal nodes. While the worst case height of a binary search tree is n which happens if the elements are inserted in increasing or decreasing order, most sequences are not in order. The average case height of a binary search tree is $logn$. Project 6 (included below) also asked to prove statements about the number of nodes or characteristics of the successor. Looking at the code below, the BST class is slightly complicated at a glance. The insert function starts at the root and takes iterators and depending on whether the element to be inserted is greater than or less than the elements in the tree it traverses to the correct location and inserts the value as a leaf. Qestion 5 on exam 2 (included below) asked to show visually how to insert nodes into a binary search tree. The remove function is slightly more difficult with multiple cases to consider. It first finds the location of the node to be deleted and its parent. If the node to be deleted has no children, it can just set the parent pointer to NULL and delete the node. However, if the node has children you have to be more careful. There are three cases if the node has children; one child to the right, one child to the left, or two children. If the node only has one child, whether it is to the right or left, find if the node is a right or left child itself. Then point the parent pointer in that direction to the one child of the node, then delete the node. If there are two children, you find the successor (the

next biggest node but it will still uphold the properties), and put it in the place of the deleted node. Question 6 on exam 2 (included below) asked to visually show how to remove nodes from a binary search tree. The maximum and minimum functions are easy, they just traverse the tree completely to the right for a maximum and completely to the left for a minimum. To find the predecessor of a node, you must first check if it has a left subtree. If it does, return the maximum of that tree. If it does not, traverse the tree up to the right as far as possible and then up one more to the left. Finding the successor is similar - if it has a right subtree, return the minimum of that tree. If it does not, traverse the tree in reverse to the left as far as possible and then one more to the right. Even though the average case of binary search trees is faster than a linked list, the worst case ($\Theta(n)$ is the same. Red-Black trees solve this problem, the worst case time complexity for a red-black tree is $\Theta(logn)$.

Red-Black trees are an instance of binary search trees with a few extra properties. The five properties of red-black trees are
1 - Every node is either red or black.
2 - The root is black.
3 - Every leaf is black.
4 - If a node is red, both of its children are black.
5 - For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
By maintaining these properties, red-black trees are inherently balanced and never have a height more than $logn$. Project 8 asked to prove that the longest path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest path from x to a descendant leaf. Looking at the code from Project 8 below most of the functions are very similar to the BST class with a few minor differences. The insert function starts off the same way, but after it inserts the node, it calls an auxiliary function called RBTFix on the new node. This function makes sure the 5 properties are maintained after an insert. It first checks if the parent of the node is red, because that could violate property (4). If the parent is red, it then splits into two cases depending on if the parent is a left child or a right child. If the parent is a left child, it finds the uncle of the node, which is the right child of the parent's parent. If the uncle is red, you can just change the color of the uncle to black to fix the black height. You then color the parent's parent red and change the value of current which could cause the while loop to run again, but usually there is only one iteration of the loop. If the uncle is black, you want the node to be a left child so if it isn't already it calls LeftRotate. After current is a left child, it changes the color of the parent to black and the parent's parent to red and calls RightRotate on the parent's parent to make the black heights equal which makes the path more balanced. The other case, if a parent is a right child, is symmetrical to the first case. At the end of the while loop, it colors the root black to uphold property (2) and the function returns. The LeftRotate function takes a node x and finds it's right child y. It then sets the appropriate pointers and makes the left subtree of y equal to x's right subtree. It then rotates the nodes so that y is moved into x's original place and the parent of x points to y. RightRotate is

symmetrical to this. Question 7 on exam 3 (included below) asked to visually show the insertion of nodes into a RBT. Since the tree is always balanced, insert and search will take $\Theta(logn)$ even in the worst case, which is much more efficient than a regular binary search tree.

Overall, I think Binary Search Trees are one of my favorite topics we covered this semester. The accompanying algorithms are very elegant and make complete sense to me. Red-Black trees are very cool how they are inherently balanced without every calculating the length of a subtree. I think I have a very thorough understanding of the topic and want to learn more about them.

Hannah Kerr and Emma Steinman
CS 271 Project 6
December 17, 2017

1.a To prove that a complete binary tree with height h will have $2^{h+1} - 1$ total nodes we will start with a base case where $h = 0$. A complete binary tree with height zero is just the root, which has $2^{0+1} - 1 = 1$ total nodes. Assume that all complete binary trees with heigh k, where $0 \leq k \leq h - 1$, have $2^{h+1} - 1$ total nodes. Now assume we have a complete binary tree with height h. Each of the subtrees of the root have height $h - 1$. Since this is in the range of k, we can apply the inductive hypothesis, so each subtree has $2^{(h-1)+1} - 1 = 2^h - 1$ total nodes. Thus there are $2(2^h - 1) + 1 = 2^{h+1} - 1$ total nodes in the complete binary tree with height h.

1.b To prove that a complete binary tree with n nodes will have $\frac{n-1}{2}$ internal nodes, we start with a base case where $n = 1$. A complete binary tree with one node is just the root, which is not an internal node if it has no children. Thus the complete binary tree has $\frac{1-1}{2} = 0$ internal nodes. Assume this is true for all complete binary trees with k nodes, where $0 \leq k \leq n - 1$. Now assume we have a complete binary tree with n nodes. Each of the subrees will have $\frac{n-1}{2}$ nodes and since this is in the range of k we can apply the inductive hypothesis. Thus each subtree will have $\frac{\frac{n-1}{2}-1}{2} = \frac{n-1-2}{4} = \frac{n-3}{4}$ internal nodes and the entire tree will have $2(\frac{n-3}{4})+1 = \frac{n-3}{2}+1 = \frac{n-3+2}{2} = \frac{n-1}{2}$ internal nodes in the complete binary tree with n nodes.

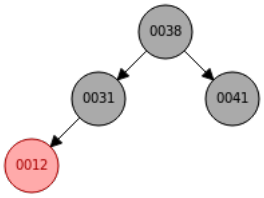
2 For a node x with an empty right subtree, to find the successor, y you find the ancestor up to the left as far as possible and then one up to the right. This, y will be the lowest ancestor of x whose left child is also an ancestor of x because the left child of the successor is as far up and to the left you can travel from x. The successor, y, is the lowest ancestor of x whose left child is also an ancestor of x because the left child of the left child of the successor will not be on the path travelling up and to the left from x.

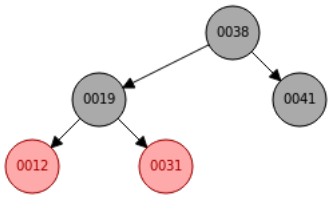
If x is the child of its successor y, it still upholds the properties that the successor is the lowest ancestor of x whose left child is also an ancestor of x because every node is an ancestor to itself.

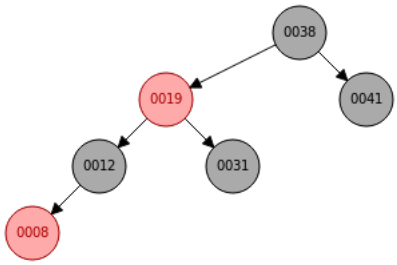
Alternatively, if y is the successor of x, x is the predecessor of y. To find the predecessor when the left subtree exists, you find the greatest item in the left subtree. x will be in the left subtree because it exists and the predecessor will be less than the successor. Once in the left subtree, travelling all the way to the right will give you x because as the predecessor of y, x is the greatest element in the left subtree.

Emma Steinman and Colin Smith
CS 271 Project 8
December 17, 2017

1. Insert {41,38,31,12,19,8}







2. Hypothesis: The longest path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest path from x to a descendant leaf.
   Proof:

   By definition of a red-black tree the number of black nodes on the path from a node to a descendant leaf must be the same for all paths.
   By definition of a red-black tree the children of a red node must be black.
   The shortest possible path for a given red-black tree is one where every node is black because there are no reds adding to the length.
   The longest possible path for a given red-black tree is one where every black node has a red child. This causes there to be the maximum number of red nodes, because every black node will be followed by a red node. This adds the maximum number of reds to the length.
   If every black node is followed by a red node there are as many red nodes as black nodes and therefore the path is twice the length of a path consisting only of black nodes. We know the path consisting of only black nodes is the shortest possible path for a given red-black tree. So, the longest possible path for a given red-black tree is twice the shortest possible path for the red-black tree. Because all paths in any given red-black tree are within these bounds, any given longest path in a red-black tree is at most twice the shortest path in the same red-black tree.

```
//=============================================================
// BST.cpp
// This file contains excerpts from the Binary Search Tree
// class and the Red Black Tree class.
//=============================================================

//=============================================================
// BST
//=============================================================

//=============================================================
// insert
// inserts the value given as the parameter into the correct
// node in the bst.
// pre-condition: It is assumed that the bst is valid
// post-condition: The bst is valid after insertion
//=============================================================
template <class KeyType>
void BST<KeyType>::insert (KeyType *k)
{
        Node<KeyType> *K = new Node<KeyType>;           //node to be
        K->data = k;                                                    //inserted
        K->left = NULL;
        K->right = NULL;
        Node<KeyType> *qtr = NULL;
        Node<KeyType> *ptr = root;
        while (ptr != NULL)
        {
                qtr = ptr;
                if (*(K->data) < *(ptr->data))
                        ptr = ptr->left;
                else
                        ptr = ptr->right;
        }
        if (qtr == NULL)
        {
                root = K;
        }
        else if (*(K->data) < *(qtr->data))
        {
                qtr->left = K;
        }
        else
                qtr->right = K;
}
//=============================================================
// remove
// removes and deletes the node of the first instance of the
// value passed as a parameter
// pre-condition: the bst is valid
// post-condition: the bst is still valid after removal
//=============================================================
template <class KeyType>
void BST<KeyType>::remove (const KeyType& k)
{
        Node<KeyType> *K = find(root, k);
        Node<KeyType> *parent = findParent(root, k);
        if (K->left == NULL && K->right ==NULL)                 //case 1:
        {                                                                       //no c
hildren
                if (*(parent->data) < *(K->data))
                        parent->right = NULL;
                else
```

```
                        parent->left = NULL;
                delete K;
        }
        else if (K->left == NULL && K->right != NULL) //case 2: one
        {                                                                       //chil
d (right)
                if (*(K->right->data) < *(parent->data))
                {
                        parent->left = K->right;
                        delete K;
                }
                else
                {
                        parent->right = K->right;
                        delete K;
                }
        }
        else if (K->left != NULL && K->right == NULL)   //one child
        {
// (left)
                if (*(K->left->data) < *(parent->data))
                {
                                parent->left = K->left;
                                delete K;
                }
                else
                {
                        parent->right = K->left;
                        delete K;
                }
        }
        else                            //case 3: two children
        {
                Node<KeyType> *succ = find(root, *successor(k));
                KeyType temp = *(succ->data);
                remove(*(succ->data));
                *(K->data) = temp;
        }
}

//============================================================
// maximum
// returns a KeyType pointer to the maximum element in the bst
//============================================================
template<class KeyType>
KeyType* BST<KeyType>::maximum (void) const
{
        Node<KeyType> *ptr = root;
        if (ptr == NULL)
        {
                cout << "Error: empty tree" << endl;
                //exit(1);
        }
        while (ptr->right != NULL)
                ptr = ptr->right;
        return ptr->data;
}
//============================================================
// minimum
// returns a KeyType pointer to the minimum element in the bst
//============================================================
template<class KeyType>
KeyType* BST<KeyType>::minimum (void) const
```

```
{
        Node<KeyType> *ptr = root;
        if (ptr == NULL)
        {
                cout << "Error: empty tree" << endl;
                //exit(1);
        }
        while (ptr->left != NULL)
    ptr = ptr->left;
  return ptr->data;
}
//============================================================
// successor
// returns a KeyType pointer to the successor of the first
// instance of the KeyType parameter value
//============================================================
template<class KeyType>
KeyType* BST<KeyType>::successor  (const KeyType& k) const
{
        Node<KeyType> *K = find(root, k);               //case 1: find
        if (K->right != NULL)   //smallest value in right subtree
        {
                K = K->right;
                while (K->left != NULL)
                {
                        K = K->left;
                }
                return K->data;
        }
        Node<KeyType> *parent = findParent(root, k);
        while (parent != NULL && K == parent->right)
        {
                K = parent;
                parent = findParent(root, *(parent->data));
        }
        return parent->data;
}
//============================================================
// predecessor
// returns a KeyType pointer to the predecessor of the first
// instance of the KeyType parameter value
//============================================================
template<class KeyType>
KeyType* BST<KeyType>::predecessor  (const KeyType& k) const
{
        Node<KeyType> *K = find(root, k);
        if (K->left != NULL)
        {
                K = K->left;
                while (K->right != NULL)
                {
                        K = K->right;
                }
                return K->data;
        }
        Node<KeyType> *parent = findParent(root, k);
        while (parent != NULL && K == parent->left)
        {
                K = parent;
                parent = findParent(root, *(parent->data));
        }
        return parent->data;
}
```

```
//=============================================================
// RBT
//=============================================================

template <class KeyType>
class Node
{
public:
        KeyType         *data;
        Node                    *left;
        Node                    *right;
        Node                    *p;
        string          color;

        Node            (void)                          //default constructor
        {
                data = NULL;
                left = NULL;
                right = NULL;
                p = NULL;
                color = "red";
        };
        Node            (KeyType *item) //constructor with item
        {
                data = item;
                left = NULL;
                right = NULL;
                p = NULL;
                color = "red";
        };
        Node            (KeyType *item, Node<KeyType>* nil)//nil pointer
        {                                                                                       //cons
tuctor
                data = item;
                left = nil;
                right = nil;
                p = nil;
                color = "red";
        };


};

template <class KeyType>
class RBT
{
protected:
        void    RBTFix                          (Node<KeyType> *current);
        void    LeftRotate                      (Node<KeyType> *z);
        void    RightRotate                     (Node<KeyType> *z);
        string  inOrderHelper   (Node<KeyType> *z,stringstream &s);
        string  preOrderHelper  (Node<KeyType> *z, stringstream &s);
        string  postOrderHelper (Node<KeyType> *z, stringstream &s);
        Node<KeyType>*  copy                                            (Node<KeyType> *z);
        void                                                    clear           (Node<KeyType> *z);
        Node<KeyType>*  find    (Node<KeyType> *r, KeyType k) const;

public:

        Node<KeyType> * root;     // root pointer for red black tree
        Node<KeyType> * nil;    // null pointer for leaf nodes

                        RBT                     (void);
```

```
                ~RBT                            (void);
                        RBT             (const RBT<KeyType> & r);

    bool                Empty                   (void) const;
    KeyType             *get                    (const KeyType& k);
    void                insert                  (KeyType *k);
    void                remove                  (const KeyType& k);
    KeyType             *maximum                (void) const;
    KeyType             *minimum                (void) const;
    KeyType             *successor              (const KeyType& k)const;
    KeyType             *predecessor            (const KeyType& k)const;
    string              inOrder                 (void) const;
    string              preOrder                (void) const;
    string              postOrder               (void) const;

};


class EmptyError {};


//=============================================================
//insert
//inserts item into RBT
//Pre-Condition:
//Post-Condition:
//Parameters: KeyType *k - pointer to item to be inserted
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::insert    (KeyType *k)
{
        Node<KeyType> *parent = nil;
        Node<KeyType> *current = root;
        Node<KeyType> *newNode = new Node<KeyType>(k, nil);
        //newNode->left = nil;
        //newNode->right = nil;
        while (current != nil)          //find where to insert newNode
        {
                parent = current;
                if (*newNode->data > *current->data)
                {
                        current = current->right;
                }
                else
                {
                        current = current->left;
                }
        }
        if (current == nil)                     //newNode inserted at root
        {
                root = newNode;
        }
        if (current == current->p->left)        //newNode should be
        {                                                                       //inserted to
left
                parent->left = newNode;
        }
        else
        {
                parent->right = newNode;                //inserted to right
        }
```

```
        RBTFix(newNode);                //fix black heights and balance tree


}
//=============================================================
//LeftRotate
//rotates the node left
//Pre-Condition:
//Post-Condition:
//Parameters: Node *current - pointer to node to be rotated
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::LeftRotate(Node<KeyType> *z)
{
        Node<KeyType> y = z->right;     //sets y to z's right child
        z->right = y->left;             //sets z's right subtree to y's
        if (y->left != nil)                             //left subtree
        {
                y->left->p = z;         //sets y's left subtree parent to z
        }
        y->p = z->p;                            //sets y's parent to z's parent
        if (z->p == nil)                                                //case 1: z is root
        {
                root = y;
        }
        else if (z == z->p->left)               //case 2: z is left child
        {
                z->p->left = y;
        }
        else                                                    //case 3: z is right child
        {
                z->p->right = y;
        }
        y->left = z;                                                    //put z on y's left
        z->p = y;
}
//=============================================================
//RightRotate
//rotates node right
//Pre-Condition:
//Post-Condition:
//Parameters:  Node *current - pointer to node to be rotated
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::RightRotate(Node<KeyType> *z)
{
        Node<KeyType> y = z->left;              //sets y to z's left child
        z->left = y->right;             //sets y's right subtree to
        if (y->right != nil)            //z's left subtree
        {
                y->right->p = z;
        }
        y->p = z->p;
        if (z->p == nil)                                //case 1: z is root
        {
                root = y;
        }
        else if (z == z->p->left)               //case 2: z is left child
        {
                z->p->left = y;
        }
```

```
        else                                    //case 3: z is right child
        {
                z->p->right = y;
        }
        y->right = z;                           //put z on y's right
        z->p = y;
}
//=============================================================
//RBTFix
//"fixes" RBT to uphold RBT properties after an insert
//Pre-Condition:
//Post-Condition:
//Parameters: Node *current - pointer to node to be rotated
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::RBTFix    (Node<KeyType> *current)
{
        while (current->p->color == "red")
        {
                if(current->p == current->p->p->left)//parent is l child
                {
                        Node<KeyType> *uncle = current->p->p->right;
                        if (uncle->color == "red")      //if uncle red can just
                        {                                                       //change color
 of parent
                                current->p->color = "black";//and uncle to fix
                                uncle->color = "black";                 //black height
                                current->p->p->color = "red";
                                current = current->p->p;
                        }
                        else
                        {
                                if (current == current->p->right)
                                {
                                        current = current->p;   //makes current a
                                        LeftRotate(current);    //left child
                                }
                                current->p->color = "black";
                                current->p->p->color = "red";//fixes colors
                                RightRotate(current->p->p);
                        }
                }
                else                    //parent is right child
                {
                        Node<KeyType> *uncle = current->p->p->left;
                        if (uncle->color == "red")
                        {                                                       //switches col
or
                                current->p->color = "black";//of uncle
                                uncle->color = "black"; //and parent to fix
                                current->p->p->color = "red";//black height
                                current = current->p->p;
                        }
                        else
                        {
                                if (current == current->p->left)
                                {
                                        current = current->p;
                                        RightRotate(current);//makes current a right
                                }                                                       //chil
d
                                current->p->color = "black";
```

```
                                current->p->p->color = "red";//fixes colors
                                LeftRotate(current->p->p);

                            }
                        }
                }
                root->color = "black";          //fixes root color to black
}
```

8 Object Oriented Programming

In this section I will discuss Object-oriented programming in C++ including how to implement template classes and using inheritance. I think I have reached the proficiency level in this section. I understand template classes very well and I know how to use inheritance, but I do not fully understand how it works, I just know what to type so that it does work. Similarly, I know how to overload operators and why you need to, but I don't fully understand the concept of Polymorphism. Throughout the semester we have been using template classes for object-oriented programming. Included below are several header files for template classes. Template classes are extremely useful because they can be used for any data type, even one you make like the Movie class we made for several projects. We were able to create several different classes for data structures that we then used to implement a Dictionary ADT which we used with our Movie class as the KeyType. If the class was not templated, we would've had to create a new Dictionary or data structure class every time we wanted to insert a different type. All of our projects this semester were creating objects. In an object-oriented language, an instance of the class is called an object. For example in the code below, when a movie type is constructed, it is considered an object. This object has attributes which are the private variables. For example in the RBT header, the Node class has many attributes for each object.

Along with template classes, we did a lot of inheritance this semester. Inheritance differs from simply `using` another class. When you are `using` another class, the object contains an object from that class, but when you inherit the object is an object from that class. There is little code involved, you have to use the correct formatting though. Looking at the code below, when I first implemented the dictionary from the Hash Table, I was making it too complicated. Instead, you can use the format that is not commented out and the class will inherit those functions. It also inherits all protected variables. The protected section is the same as the private section except for when you inherit the class it turns private and you can not use a private section in an inherited class.

Also included in many of our classes were elements of polymorphism. Often we needed to overload operators to do what we wanted. For example in the Movie class, we overloaded the comparison operators to compare the titles of the movies. This allowed the movies to be sorted by title in the dictionary rather than the cast or another way. We almost always overload the cout operator to print the elements of our object that we want. Other elements of polymorphism include having multiple functions with the same names that take different input. This allows for as much abstraction as possible for the user because they can use what they think is the same function for different inputs, when really there are several function definitions under the same name.

Overall, this is probably the section I am least confident about. I understand how to implement things and why they are useful, but I do not fully understand what is actually happening when you type the things you are supposed to type. I hope with more practice and research I will understand what is going on.

```
//=============================================================
// header.cpp
// This file contains several header files
//=============================================================

//=============================================================
// LIST AS ARRAY HEADER
//=============================================================
template <class T>
class List
{
private:
        T               *array;
        int             DEFAULT_LIST_SIZE = 10;
        int             capacity;
        int     size;

public:

                                List                (void);             //default
                List                (const List<T> &c);
                ~List               (void);             //destructor
        bool            isEmpty                 (void);
        int             length              (void) const;
        T&                      operator[]              (int i);        //index operator
        string          toString            (void)const ;
        void            append              (T c);
        void            insert              (T, int);
        void            remove              (int);
        List<T>         operator+               (const List<T> c) const ;
        List<T>         operator=               (List<T> c) const ;
        void        clear                   (void);
  friend ostream& operator<<    (ostream &os, List<T> c)
  {
    for (int i = 0; i < c.size; i++)
      os << c[i] << " ";

    return os;
  }
};
class IndexError { };

//=============================================================
// MINHEAP HEADER
//=============================================================

template <class KeyType>
class MinHeap
{
  public:
    MinHeap(int n = DEFAULT_SIZE);
    MinHeap(KeyType initA[], int n);
    MinHeap(const MinHeap<KeyType>& heap);
    ~MinHeap();

    void heapSort(KeyType sorted[]);

    MinHeap<KeyType>& operator=(const MinHeap<KeyType>& heap);
    std::string toString() const;

  private:
    KeyType *A;     // array containing the heap
    int heapSize;   // size of the heap
```

```
    int capacity;   // size of A

        void heapify(int index);
    void buildHeap();
        int leftChild(int index) { return 2 * index + 1; }
        int rightChild(int index) { return 2 * index + 2; }
        int parent(int index) { return (index - 1) / 2; }
    void swap(int index1, int index2);
    void copy(const MinHeap<KeyType>& heap);
    void destroy();
};

//=============================================================
// HASH TABLE HEADER
//=============================================================
template <class KeyType>
class HashTable
{
public:
                                            HashTable       (int numSlots);
                                            HashTable       (const HashTable<KeyType>& h);
                                            ~HashTable      ();
        KeyType*                        get                 (const KeyType& k) const;
        void                            insert          (KeyType *k);
        void                            remove          (const KeyType& k);
        HashTable<KeyType>& operator=   (const HashTable<KeyType>& h);
        std::string toString                                (int slot) const;

private:
        int                     slots;
        List<KeyType>   *table; // an array of List<KeyType>â\200\231s
};


//=============================================================
// RBT HEADER
//=============================================================

template <class KeyType>
class Node
{
public:
        KeyType         *data;
        Node                    *left;
        Node                    *right;
        Node                    *p;
        string          color;

        Node            (void)                          //default constructor
        {
                data = NULL;
                left = NULL;
                right = NULL;
                p = NULL;
                color = "red";
        };
        Node            (KeyType *item) //constructor with item
        {
                data = item;
                left = NULL;
                right = NULL;
                p = NULL;
                color = "red";
```

```
        };
        Node            (KeyType *item, Node<KeyType>* nil)//nil pointer
        {                                                                       //cons
tuctor
                data = item;
                left = nil;
                right = nil;
                p = nil;
                color = "red";
        };


};

template <class KeyType>
class RBT
{
protected:
        void    RBTFix                          (Node<KeyType> *current);
        void    LeftRotate                      (Node<KeyType> *z);
        void    RightRotate                     (Node<KeyType> *z);
        string  inOrderHelper   (Node<KeyType> *z,stringstream &s);
        string  preOrderHelper  (Node<KeyType> *z, stringstream &s);
        string  postOrderHelper (Node<KeyType> *z, stringstream &s);
        Node<KeyType>*  copy                                    (Node<KeyType> *z);
        void                                            clear           (Node<KeyType> *z);
        Node<KeyType>*  find    (Node<KeyType> *r, KeyType k) const;

public:

        Node<KeyType> * root;     // root pointer for red black tree
        Node<KeyType> * nil;    // null pointer for leaf nodes

                        RBT                     (void);
                ~RBT                            (void);
                        RBT             (const RBT<KeyType> & r);

    bool                Empty                   (void) const;
    KeyType             *get                    (const KeyType& k);
    void                insert                  (KeyType *k);
    void                remove                  (const KeyType& k);
    KeyType             *maximum                (void) const;
    KeyType             *minimum                (void) const;
    KeyType             *successor              (const KeyType& k)const;
    KeyType             *predecessor            (const KeyType& k)const;
    string              inOrder                 (void) const;
    string              preOrder                (void) const;
    string              postOrder               (void) const;

};


class EmptyError {};
//=============================================================
// MOVIE CLASS
//=============================================================

class Movie
{
public:
  string title;
  string cast;
```

```
  bool operator< ( Movie& a) const
  {
    return this->title <= a.title;
  };
  bool operator> (const Movie& a) //overloading comparison oper.
  {                               //to compare key specifically
    return this->title > a.title;
  };
  bool operator== (const Movie& a)
  {
    return this->title == a.title;
  };
  bool operator!= (const Movie& a)
  {
    return this->title != a.title;
  };

  friend ostream & operator<< (ostream &os, const Movie &mov)
  {                                        //overloading
      os << mov.cast;                                          //cout operator
      return os;
  };
};
//=============================================================
//movieDict
//takes string of text file as parameter and reads in and
//separates
//and creates a dictionary of movie titles and cast
//lists and returns
//said dictionary
//=============================================================
Dictionary<Movie> movieDict(string movieFile)
{
  ifstream file;
  file.open(movieFile);
  string line;
  Dictionary<Movie> movieDictionary;
  while (file)
  {
    getline(file, line);
    int index = line.find('\t');
    Movie *mov = new Movie;
    mov->title = line.substr(0, index);
    mov->cast = line.substr(index+1);
    //cout << mov->title << endl;
    movieDictionary.insert(mov);
  }
  file.close();
  return movieDictionary;
}

//=============================================================
// DICTIONARY
//=============================================================
template <class KeyType>
class Dictionary: public HashTable<KeyType>
{
/*
public:

              Dictionary  (){};         //default constructor
              ~Dictionary (void){};    //destructor
              Dictionary  (const Dictionary<KeyType> &d){};
```

```
  void        insert      (KeyType *k)
  {
    HashTable<KeyType>::insert(k);
  };
  void        remove      (const KeyType &k)
  {
    HashTable<KeyType>::remove(k);
  };
  KeyType *   get         (const KeyType &k) const
  {
    HashTable<KeyType>::get(k);
  };
  bool        Empty       (void)  const
  {
    HashTable<KeyType>::Empty();
  };

private:
  HashTable<KeyType> *d;
*/

public:

        Dictionary():HashTable<KeyType>(){};
        ~Dictionary(void) {};
Dictionary(const Dictionary<KeyType> &d):HashTable<KeyType>(d){};

//Inherited HashTable Class incorrectly, much simpler this way.
};
```

9 Professional Practice

Professional practice is extremely important to learn because it can greatly impact your future. Employers are looking to hire employees who can turn in projects on time, work with others, and communicate results effectively. Additional skills such as the LaTeX and Linux systems are valuable to employers and can set you apart from the large pool of applicants. I believe I have reached the mastery with distinction level in this area. I am very deadline oriented and have developed the skills to manage my many heavy course-loads. I keep track of all of my assignments so that I do not fall behind and run out of time to complete an assignment. I have developed effective communication skills as well, which is also important to employers. Even if an employee finishes a project on time, if they cannot explain it to others then they are not as appealing as a candidate who can explain their work and process to others. Commenting on code is important so that others can understand your code without spending a lot of time figuring it out. I think it takes practice to figure out what is important and what is unnecessary to comment, and how to determine pre- and post-conditions, but over the course of three semesters I believe I have developed the proper conventions. As seen below, commenting includes commenting throughout the function and the pre- and post- conditions. However, some functions are self-explanatory and over-commenting is just as bad as under-commenting. You can assume that anyone looking at your code has at least a basic knowledge of programming, or else all the comments in the world may not help them. The code shown below shows a mix of commenting styles and things that are unecessary to comment. Unit testing is also a professional skill I have learned throughout the semester. Testing your code for various circumstances and attempting to break it is very good practice because finding a bug before you send it out to multiple people saves everyone else's time and makes you look better. Included below are some sample unit tests for the set class that I wrote. I tested many different combinations of data types and all of the functions. I utilized the assert statement so that when you run your unit test, nothing should happen. If you get error statements then something is wrong and you know to fix it. When writing actual software or more intricate code many more test cases would be developed and by many other people, but it is good to get in the habit and learn how early on.

Proper emailing etiquette can give a good first impression, while poor etiquette can make a lasting bad impression. Proper etiquette includes using appropriate language, a clear message, and timely responses. Through emailing the professor with questions and other students to collaborate on projects, I have improved my communication through emails. Additionally learning to use LaTeX is a very important skill which I believe will help me in my future endeavors. LaTeX is advantageous to a word processor because you have control over both the layout and the content. It is also easier to include mathematical equations and other elements that are difficult to produce in a word processor. As seen below, I am able to write mathematical functions, and insert pictures. Overall, this course has taught me more than just technical skills, but invaluable

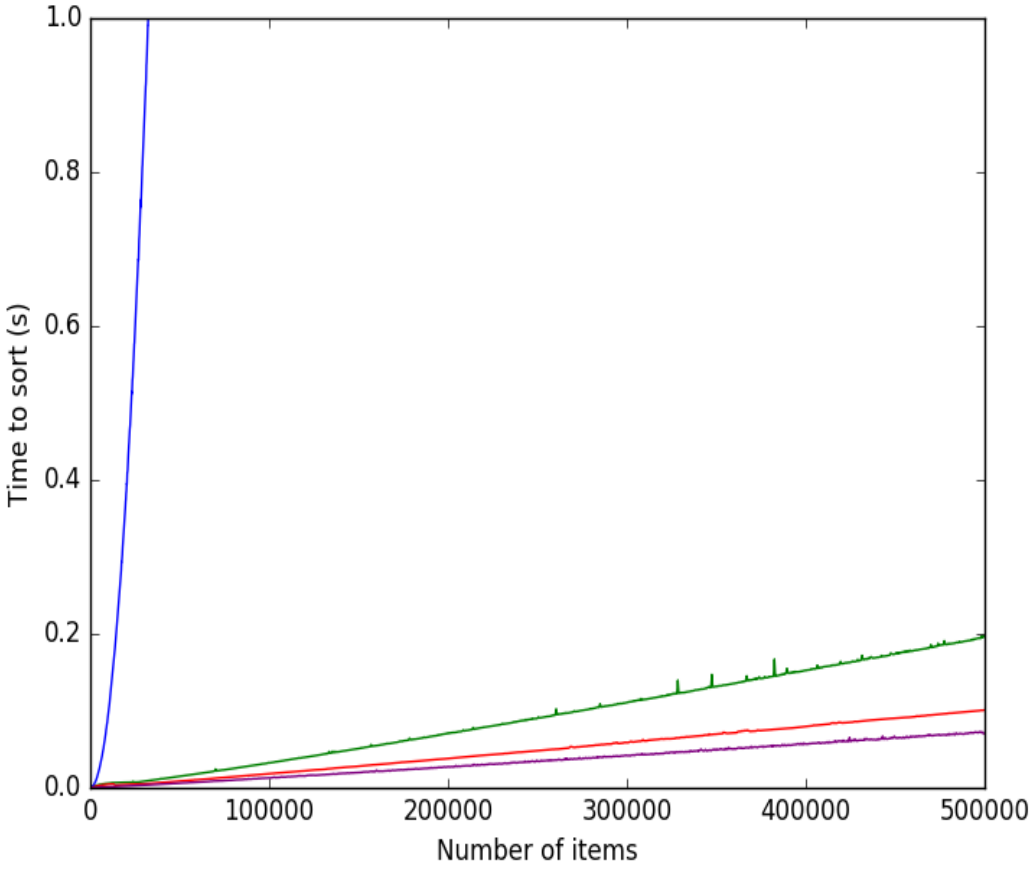
professional skills that are much harder to learn. I think these will benefit me in my future when I am in the process of finding a job.

Colin Smith and Emma Steinman
CS 271 Project 5
December 16, 2017

1.a By definition $T(n) = T(k) + T(n-k-1) + cn : n \geq 2$ and $T(n) = a : n \leq 1$
Our hypothesis is $T(n) = (\frac{c}{2(k+1)})n^2 + (\frac{T(k)}{k+1} + \frac{c}{2})n + a$. The base case occurs when $n \leq 1$.
$RHS : (\frac{c}{2(k+1)})0^2 + (\frac{T(k)}{k+1} + \frac{c}{2})0 + a = 0 + 0 + a = a$
$LHS : T(0) = a$
$RHS = LHS$
Assume $T(n) = (\frac{c}{2(k+1)})n^2 + (\frac{T(k)}{k+1} + \frac{c}{2})n + a$ up to $n-1$. Now in the inductive step we prove that $T(n) = T(k) + T(n-k-1) + cn = (\frac{c}{2(k+1)})n^2 + (\frac{T(k)}{k+1} + \frac{c}{2})n + a$.
Due to the inductive assumption because $k \in \mathbb{N} \Leftrightarrow n-k-1 \leq n-1$:
$T(n) = T(k) + T(n-k-1) + cn = T(k) + (\frac{c}{2(k+1)})(n-k-1)^2 + (\frac{T(k)}{k+1} + \frac{c}{2})(n-k-1) + a + cn$
$= T(k) + (\frac{c}{2(k+1)})n^2 - (\frac{c}{2(k+1)})2nk - (\frac{c}{2(k+1)})2n + (\frac{c}{2(k+1)})k^2 + (\frac{c}{2(k+1)})2k + (\frac{c}{2(k+1)}) + (\frac{T(k)}{k+1} + \frac{c}{2})n - (\frac{T(k)}{k+1} + \frac{c}{2})k - (\frac{T(k)}{k+1} + \frac{c}{2}) + a + cn$
$= [(\frac{c}{2(k+1)})n^2 + (\frac{T(k)}{k+1} + \frac{c}{2})n + a] + T(k) - (\frac{c}{2(k+1)})2nk - (\frac{c}{2(k+1)})2n + (\frac{c}{2(k+1)})k^2 + (\frac{c}{2(k+1)})2k + (\frac{c}{2(k+1)}) - (\frac{T(k)}{k+1} + \frac{c}{2})k - (\frac{T(k)}{k+1} + \frac{c}{2}) + cn$
$Note : [(\frac{c}{2(k+1)})n^2 + (\frac{T(k)}{k+1} + \frac{c}{2})n + a]$ will be called S until the end of the inductive step to keep things concise
$S + T(k) - \frac{cnk}{k+1} - \frac{cn}{k+1} + \frac{ck^2}{2(k+1)} + \frac{2ck}{2(k+1)} + \frac{c}{2(k+1)} - \frac{T(k)k}{k+1} - \frac{c}{2} + cn$
$S + T(k) - \frac{cn(k+1)}{k+1} + \frac{c(k^2+2k+1)}{2(k+1)} - \frac{T(k)(k+1)}{k+1} - \frac{c(k+1)}{2} + cn$
$S + T(k) - cn + \frac{c(k+1)^2}{2(k+1)} - T(k) - \frac{c(k+1)^2}{2(k+1)} + cn$
S
$S = (\frac{c}{2(k+1)})n^2 + (\frac{T(k)}{k+1} + \frac{c}{2})n + a$
Thus we have proven our n case as we have shown $T(n) = (\frac{c}{2(k+1)})n^2 + (\frac{T(k)}{k+1} + \frac{c}{2})n + a$

1.b $\Theta(n^2) = 0 \leq c_1 n^2 \leq (\frac{c}{2(k+1)})n^2 + (\frac{T(k)}{k+1} + \frac{c}{2})n + a \leq c_2 n^2$
$= \frac{1}{3(k+1)}n^2 \leq (\frac{c}{2(k+1)})n^2 + (\frac{T(k)}{k+1} + \frac{c}{2})n + a \leq 2(c+a)n^2$ $\frac{1}{3}n^2 \leq (\frac{c}{2}n^2 + \frac{T(k)}{k+1} + \frac{c}{2})n + a \leq 2(c+a)n^2$
For $n = 1$
$\frac{1}{3(k+1)} \leq \frac{c}{2(k+1)} + \frac{T(k)}{k+1} + \frac{c}{2} + a \leq 2(c+a)$
Since k can be at least 0 and at most $n-1$, prove for both.
for $k = 0$
$\frac{1}{3} \leq \frac{c+2T(0)+c(1)}{2(0+1)} \leq 2(c+a)$
$\frac{1}{3} \leq \frac{c+2a+c}{2} \leq 2(c+a)$
$\frac{1}{3} \leq c + a \leq 2(c+a)$ Which is true for all c
For $k = n-1$
$\frac{1}{3(n-1)+1}n^2 \leq \frac{c}{2((n-1)+1)}n^2 + (\frac{T(n-1)}{(n-1)+1} + \frac{c}{2})n + a \leq 2(c+a)n^2$
$\frac{n^2}{3n} \leq \frac{cn}{2} + (\frac{T(n-1)}{n} + \frac{c}{2})n + a \leq 2cn^2 + 2an^2$
$\frac{n}{3} \leq \frac{cn}{2} + T(n-1) + \frac{cn}{2} + a \leq 2cn^2 + 2an^2$
$\frac{n}{3} \leq cn + T(n-1) + a \leq 2cn^2 + 2an^2$ Which is true for any c and n

1. Graph of Sort Times



Blue = Insertion Sort
Green = Heap Sort
Red = Merge Sort
Purple = Quick Sort

In this plot you can see that quick sort is the fastest sorting algorithm, followed by merge sort, followed by heap sort, with insertion sort being the slowest. We tested all of our sorting algorithms on randomized arrays ranging from size 0 to 500,000, by increments of 500 for Heap Sort, Merge Sort, and Quick Sort and from 0 to 50000 in increments of 100 for Insertion Sort. This plot follows the time complexities we found for each of these sorting algorithms. Insertion sort has time complexity $\Theta(n^2)$ which is much slower than the next fastest sorting algorithm, heap sort. Heap sort, merge sort, and quick sort all have time complexity of $\Theta(nlogn)$. However, because Θ is purely asymptotic it does not take into account constants which make certain algorithms faster or slower even though they are asymptotically similar; therefore, though Heap Sort, Merge Sort, and Quick Sort are all $\Theta(nlogn)$, due to constants Quick Sort is the quickest.

```
//=============================================================
// Professional.cpp
// This file contains exercepts hilighting my commenting
// style.
//=============================================================
//=============================================================
// SET CLASS
//=============================================================

//=============================================================
//operator ==
//returns a boolean value indicating if the two sets are equal
//=============================================================
template <class Element>
bool    Set<Element>::operator==        (const Set<Element> & s) const
{
        if (length != s.length)
                return false;

        else
        {
                Node<Element> * ptr = head;

                while (ptr != NULL)
                {
                        if (!s.contains(ptr->data))
                                return false;
                        ptr = ptr->next;
                }

                return true;
        }
}

//=============================================================
//operator <=
//returns a boolean value indicating if the set is a subset
//of another set
//=============================================================
template <class Element>
bool    Set<Element>::operator<=        (const Set<Element> & s) const
{
        if (s.length == 0)
                return true;                                    //empty set is always
                                                                        //a subset
        Node<Element> * ptr = head;

        while (ptr != NULL)
        {
                if (!s.contains(ptr->data))
                        return false;

                ptr = ptr->next;
        }

        return true;
}

//=============================================================
//operator +
//returns the union of two sets
//=============================================================
template <class Element>
```

```
Set<Element>&   Set<Element>::operator+ (const Set<Element> & s) const
{
        Set<Element> *s1 = new Set();
        Node<Element> * ptr = head;

        while (ptr != NULL)                             //inserts elements from
        {                                                               //first set
                s1->insert(ptr->data);
                ptr = ptr->next;
        }

        Node<Element> * qtr = s.head;
        while (qtr != NULL)                             //inserts elements from
        {                                                               //second set
                s1->insert(qtr->data);
                qtr=qtr->next;
        }

        return *s1;

}
//=================================================================
// RBT CLASS
//=================================================================

//=================================================================
//insert
//inserts item into RBT
//Pre-Condition:
//Post-Condition:
//Parameters: KeyType *k - pointer to item to be inserted
//Return Value: void
//=================================================================
template <class KeyType>
void            RBT<KeyType>::insert    (KeyType *k)
{
        Node<KeyType> *parent = nil;
        Node<KeyType> *current = root;
        Node<KeyType> *newNode = new Node<KeyType>(k, nil);
        //newNode->left = nil;
        //newNode->right = nil;
        while (current != nil)          //find where to insert newNode
        {
                parent = current;
                if (*newNode->data > *current->data)
                {
                        current = current->right;
                }
                else
                {
                        current = current->left;
                }
        }
        if (current == nil)                     //newNode inserted at root
        {
                root = newNode;
        }
        if (current == current->p->left)        //newNode should be
        {                                                                               //inserted to
left
                parent->left = newNode;
        }
        else
```

```
        {
                parent->right = newNode;                //inserted to right
        }

        RBTFix(newNode);                //fix black heights and balance tree


}
//=============================================================
//LeftRotate
//rotates the node left
//Pre-Condition:
//Post-Condition:
//Parameters: Node *current - pointer to node to be rotated
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::LeftRotate(Node<KeyType> *z)
{
        Node<KeyType> y = z->right;     //sets y to z's right child
        z->right = y->left;             //sets z's right subtree to y's
        if (y->left != nil)                             //left subtree
        {
                y->left->p = z;         //sets y's left subtree parent to z
        }
        y->p = z->p;                            //sets y's parent to z's parent
        if (z->p == nil)                                                //case 1: z is root
        {
                root = y;
        }
        else if (z == z->p->left)               //case 2: z is left child
        {
                z->p->left = y;
        }
        else                                                    //case 3: z is right child
        {
                z->p->right = y;
        }
        y->left = z;                                                    //put z on y's left
        z->p = y;
}
//=============================================================
//RightRotate
//rotates node right
//Pre-Condition:
//Post-Condition:
//Parameters:  Node *current - pointer to node to be rotated
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::RightRotate(Node<KeyType> *z)
{
        Node<KeyType> y = z->left;              //sets y to z's left child
        z->left = y->right;             //sets y's right subtree to
        if (y->right != nil)            //z's left subtree
        {
                y->right->p = z;
        }
        y->p = z->p;
        if (z->p == nil)                                //case 1: z is root
        {
                root = y;
        }
```

```
        else if (z == z->p->left)               //case 2: z is left child
        {
                z->p->left = y;
        }
        else                                    //case 3: z is right child
        {
                z->p->right = y;
        }
        y->right = z;                           //put z on y's right
        z->p = y;
}
//=============================================================
//RBTFix
//"fixes" RBT to uphold RBT properties after an insert
//Pre-Condition:
//Post-Condition:
//Parameters: Node *current - pointer to node to be rotated
//Return Value: void
//=============================================================
template <class KeyType>
void            RBT<KeyType>::RBTFix    (Node<KeyType> *current)
{
        while (current->p->color == "red")
        {
                if(current->p == current->p->p->left)//parent is l child
                {
                        Node<KeyType> *uncle = current->p->p->right;
                        if (uncle->color == "red")      //if uncle red can just
                        {                                                       //change color
 of parent
                                current->p->color = "black";//and uncle to fix
                                uncle->color = "black";                 //black height
                                current->p->p->color = "red";
                                current = current->p->p;
                        }
                        else
                        {
                                if (current == current->p->right)
                                {
                                        current = current->p;   //makes current a
                                        LeftRotate(current);    //left child
                                }
                                current->p->color = "black";
                                current->p->p->color = "red";//fixes colors
                                RightRotate(current->p->p);
                        }
                }
                else                    //parent is right child
                {
                        Node<KeyType> *uncle = current->p->p->left;
                        if (uncle->color == "red")
                        {                                                       //switches col
or
                                current->p->color = "black";//of uncle
                                uncle->color = "black"; //and parent to fix
                                current->p->p->color = "red";//black height
                                current = current->p->p;
                        }
                        else
                        {
                                if (current == current->p->left)
                                {
                                        current = current->p;
```

```
                                        RightRotate(current);//makes current a right
                                    }                                                   //chil
d
                                    current->p->color = "black";
                                    current->p->p->color = "red";//fixes colors
                                    LeftRotate(current->p->p);

                                }
                            }
                        }
                        root->color = "black";          //fixes root color to black
}


//=========================================================
// SET TEST CASES
//=========================================================


//=========================================================
// tests default constructor
//=========================================================

void test1 (void)
{
        Set<int> s1;
        string str = s1.toString();
        assert(str=="{}");

}

//=========================================================
// tests insert
//=========================================================

void test2 (void)
{
        Set<int> s1;
        s1.insert(1);
        s1.insert(4);
        s1.insert(89);
        s1.insert(3);
        string str = s1.toString();
        assert(str=="{1, 4, 89, 3}");
}

//=========================================================
// tests copy constructor
//=========================================================

void test3 (void)
{
        Set<int> s1;
        s1.insert(4);
        s1.insert(44);
        s1.insert(55);
        Set<int> s2(s1);
        string str = s2.toString();

        assert(str == "{4, 44, 55}");
}

//=========================================================
```

```
// tests remove
//============================================================

void test4 (void)
{
        Set<char> s1;
        s1.insert('s');
        s1.insert('e');
        s1.insert('a');
        s1.insert('g');
        s1.insert('z');

        s1.remove('a');

        string str = s1.toString();
        assert(str == "{s, e, g, z}");
}

//============================================================
// tests cardinality
//============================================================

void test5 (void)
{
        Set<int> s1;
        s1.insert(4);
        s1.insert(8);
        s1.insert(12);
        s1.insert(16);
        int length = s1.cardinality();
        assert (length == 4);
}

//============================================================
// tests ==
//============================================================
void test6 (void)
{
        Set<char> s1;
        Set<char> s2;
        for (char letter = 'a';letter<='z';letter++)
        {
                s1.insert(letter);
                s2.insert(letter);
        }

        if (s1==s2)
                return;

        else
                cout << "Test 6 failed." << endl;
}
//============================================================
// tests contains
//============================================================

void test7 (void)
{
        Set<int> s1;
        for (int i = 0; i < 8; i++)
                s1.insert(i);
        if (s1.contains(3))
                return;
```

```
        else
                cout << "Test 7 failed" << endl;
}


//=============================================================
// tests union
//=============================================================

void test8 (void)
{
        Set<int> s1;
        for (int i = 0; i<6; i++)
                s1.insert(i);
        Set<int> s2;
        for (int i = 6; i<11; i++)
                s2.insert(i);
        Set<int> s3;
        s3.insert(5);
        s3.insert(3);
        s3.insert(6);
        s3 = s1+s2;
        string str = s3.toString();
        assert(str == "{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}");
}

//=============================================================
//tests copy to an empty list
//=============================================================

void test9 (void)
{
        Set<int> s1;
        for (int i = 0; i < 6; i++)
                s1.insert(i);
        Set<int> s2(s1);
        string str = s2.toString();
        assert(str == "{0, 1, 2, 3, 4, 5}");
}

//=============================================================
//tests operator =
//=============================================================

void test10     (void)
{
        Set<char> s1;
        for (char a = 'a'; a<='z'; a++)
                s1.insert(a);
        Set<char> s2;
        s2 = s1;
        string str = s2.toString();
        assert(str == "{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x
, y, z}");
}


//=============================================================
//tests operator &
//=============================================================

void test11     (void)
{
```

```
        Set<char> s1;
        for (char a = 'a'; a<='e'; a++)
                s1.insert(a);
        Set<char> s2;
        for (char f = 'c'; f<= 'j'; f++)
                s2.insert(f);
        Set<char> s3;
        s3 = s1 & s2;
        string str = s3.toString();
        assert(str == "{c, d, e}");
}

//============================================================
//tests remove with item not in list
// TERMINAL
//============================================================

void test12     (void)
{
        Set<char> s1;
        for (char a = 'a'; a < 'r'; a++)
                s1.insert(a);
        s1.remove('z');


}


//============================================================
//tests operator -
//============================================================

void test13     (void)
{
        Set<int> s1;
        Set<int> s2;
        for (int i = 0; i <= 10; i++)
                s1.insert(i);
        for (int i = 0; i <=10; i+=2)
                s2.insert(i);
        Set<int> s3;
        s3 = s1-s2;
        string str = s3.toString();
        assert(str == "{1, 3, 5, 7, 9}");
}

//============================================================
//tests operator = with items in set previously
//============================================================

void test14     (void)
{
        Set<int> s1;
        for (int i = 0; i < 10; i++)
                s1.insert(i);
        Set<int> s2;
        for (int j = 0; j < 3; j++)
                s2.insert(j);
        s2 = s1;
        string str = s2.toString();
        assert(str=="{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}");
}
```

```
//=============================================================
//tests operator<=
//=============================================================
void test15     (void)
{
        Set<int> s1;
        for (int i = 0; i < 10; i++)
                s1.insert(i);
        Set<int> s2;
        for (int j = 0; j < 3; j++)
                s2.insert(j);
        if (s2 <= s1)
                return;
        else
                cout << "test 15 failed" << endl;
}

//=============================================================
//tests insert with existing item
//TERMINAL
//=============================================================

void test16     (void)
{
        Set<int> s1;
        for (int i = 0; i < 10; i++)
                s1.insert(i);
        for (int j = 5; j < 15; j++)
                s1.insert(j);
        string str = s1.toString();
        assert(str=="{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}");
}

//=============================================================
//tests <= with empty set
//=============================================================

void test17     (void)
{
        Set<int> s1;
        for (int i = 0; i < 10; i++)
                s1.insert(i);
        Set<int> s2;
        if (s1 <= s2)
                return;
        else
                cout << "test 17 failed" << endl;
}

//=============================================================
//tests intersection between two different sets
//=============================================================

void test18()
{
        Set<int> s1;
        for (int i = 0; i<6; i++)
                s1.insert(i);
        Set<int> s2;
        for (int i = 6; i<11; i++)
                s2.insert(i);
        Set<int> s3;
        s3 = s1&s2;
```

```
        string str = s3.toString();
        assert(str == "{}");
}

//=============================================================
//tests difference between two different sets
//=============================================================

void test19()
{
        Set<int> s1;
        for (int i = 0; i<6; i++)
                s1.insert(i);
        Set<int> s2;
        for (int i = 6; i<11; i++)
                s2.insert(i);
        Set<int> s3;
        s3 = s1-s2;
        string str = s3.toString();
        assert(str == "{0, 1, 2, 3, 4, 5}");
}

//=============================================================
//tests cardinality of empty set
//=============================================================

void test20     (void)
{
        Set<char> s1;
        assert(s1.cardinality() == 0);
}

//=============================================================
//tests difference operator with a bigger second set
//=============================================================

void test21     (void)
{
        Set<int> s1;
        for (int i = 0; i<6; i++)
                s1.insert(i);
        Set<int> s2;
        for (int i = 0; i<11; i++)
                s2.insert(i);
        Set<int> s3 = s1 - s2;
        string str = s3.toString();
        assert(str == "{}");
}

//=============================================================
//tests == with non equal sets
//=============================================================

void test22     (void)
{
        Set<int> s1;
        for (int i = 0; i<6; i++)
                s1.insert(i);
        Set<int> s2;
        for (int i = 6; i<11; i++)
                s2.insert(i);
        if (s1 == s2)
                cout << "Test 22 failed" << endl;
```

```
}

//============================================================
//tests string set
//============================================================

void test23 (void)
{
        Set<string> s1;
        s1.insert("Emma");
        s1.insert("Eliza");
        s1.insert("Evelyn");
        assert(s1.toString()=="{Emma, Eliza, Evelyn}");
}

//============================================================
// tests contains when item not in set
//============================================================

void test24 (void)
{
        Set<int> s1;
        for (int i = 0; i<6; i++)
                s1.insert(i);
        bool cont = s1.contains(6);
        assert(cont==0);
}
```

10 Conclusion

Overall, I have really enjoyed this course. Towards the beginning of the semester, I knew what we were learning but it wasn't until about halfway through when it truly clicked about how all of the topics related. After that, I really appreciated the different projects where we were writing classes to perform the same end function, but the implementation and the running times were different. I really enjoyed how the course was set up, we cover basic topics like time complexity at the very beginning of the semester and revisit throughout the course so it is emphasized and you truly understand it. After this epiphany, I think that my proficiency in the topics increased, and what we had done previously made more sense as well. I can approach computer science from a much different perspective than I had before this course and I am intrigued to learn more. I am very proud of myself overall and how far I have come, both from coming into college with zero coding experience, and from the beginning of the semester when I was learning things but didn't fully understand how they related or why they were important. I now know that searches in $\Theta(n^2)$ are bad and that the fastest search algorithm can be implemented in $\Theta(nlogn)$. I also understand the differences between data structures and how they relate to ADTs, and why it is more efficient to use some than others. I have really enjoyed this course and the topics covered and I hope my excitement for the subject continues to grow in the coming semesters and years.