Data Structures

CS 271-Havill

Final Portfolio

Ansel Schiavone

May 6, 2016

Analyzing Algorithms

Throughout the course, algorithm analysis was a central focus, and I feel that it is an area I have improved greatly in. Although I began the semester with a basic understanding of pre/post conditions, time complexities, etc., I now possess the ability to successfully determine algorithm performance in best, worst, and average cases, write and prove loop invariants based on preconditions and postconditions, and accurately compare algorithms.

Possessing the ability to successfully determine and prove loop invariants is an essential skill in algorithm analysis. In project I, part 3 and 4, I determined the loop invariants of linear search and bubble sort, and proved their correctness through the use of the algorithm's termination condition (see Project I).

In my analysis of bubble sort, I determined the correct invariant of the inner loop to be that "before each iteration of the loop, it is true that $A[j]$ is the smallest element in $A[j...n]$". In order to prove any loop invariant correct, three separate cases must be proven; it must be shown that the loop invariant is true before every iteration of the loop. The first step is to prove that the loop invariant is true before the first iteration of the loop. In the case of bubble sort, before the first iteration of the loop, $j \leftarrow n$. Thus, if we consider the subarray $A[j...n]$, where $j = n$, it is clear that $A[j] = A[n]$ is the smallest element in $A[n]$. The next step in any loop invariant proof is to prove a maintenance step, thereby proving the loop invariant to be true for any subsequent iterations. The first step is to assume that the loop invariant holds for some iteration j. Then it must be shown that whatever operations are performed within the loop make it such that before the following iteration, the loop invariant is still true. In the case of the inner loop in bubble sort, I demonstrated how the algorithm compares the smallest element $A[j]$ to the $A[j-1]$, and if $A[j-1] > A[j]$, the two elements are swapped.Thus, because the final action taken by the body of the inner loop is to decrement j such that $A[j] \rightarrow A[j-1]$, the correctness of the loop invariant is maintained (see Project I, problem 4a for details). The final step is to state the termination condition of the loop. The inner loop of bubble sort terminates when $j = i$. Thus, we know that at the termination of the inner loop, $A[j] = A[i]$, which is the smallest element in $A[i...n]$.

In order to prove the overall correctness of an algorithm, it is necessary to determine and prove invariants of all loops, thereby allowing the creation of pre/post conditions. If the termination condition is correctly stated in terms of the loop invariant, it then becomes possible to determine the pre and post conditions of a particular loop. After completing a similar process as detailed above for the outer for loop, I determined that the termination condition of the outer for loop of bubble sort is that i must be greater than or equal to n, meaning that i has taken on every value of n. I was able to do this by utilizing the termination condition of the inner loop. Thus, I was able to prove the correctness of the bubble sort algorithm simply by using the termination condition of the outer loop. (see Project I, problem 4b and 4c for details). From here, I was able to determine that the pre-condition for the bubble sort algorithm is an inputted list, and the postcondition is that array is in increasing sorted order.

Being new to induction techniques, I occasionally used the wrong iteration value

in determining and proving the loop invariants. However, as the semester progressed and my grasp on proofs by induction became stronger, I was able to go over Project I again and correct these indexing mistakes. I now feel that I have a strong understanding of the purpose of loop invariants, and the importance of the initialization, maintenance, and termination steps.

Another important tool in algorithm analysis is the ability to analyze and determine the asymptotic time complexities of iterative and recursive algorithms. Understanding concepts such as O,Θ and Ω is essential to writing efficient software, and creates important distinctions between different algorithms and data structures. These definitions provide a method of measuring algorithm runtime. In Project II, I proved the upper, lower, or bounded runtime of six different functions, using the definitions of O,Θ and Ω (see Project II, problem 3). I correctly found constant values that proved the O,Θ or Ω notation of five of the six functions, erring only on a constant value in part c (I have since corrected this mistake). These proofs demonstrate the significance of asymptotic notation, showing how a function can be bounded above, below, or both, simply by the variance of a constant factor.

Determining the asymptotic runtime of iterative algorithms is generally straightforward, although it requires a firm understanding of loop behavior and termination conditions. For example, the basic bubble sort consists of a nested for loop. Since each loop runs on the order of n, the best case time complexity of basic bubble sort can be described as $\Theta(n^2)$. However, by including a secondary termination condition in the outer for loop, which terminates the entire algorithm when through an iteration of the outer loop no values are swapped, we see that the algorithm's best case runtime is $\Theta(n)$, because the outer loop will only iterate once, in the best case. This is the scenario in which the array is already in sorted order. Thus, understanding termination conditions is extremely important in analyzing iterative algorithms, as slight modifications can have drastic effects on runtime.

Analyzing recursive functions is a skill that I acquired during the course, which in turn has allowed me to be able to determine time complexities of various powerful yet common recursive algorithms. In Project 2, I successfully found upper bounds for several recursive functions, in each case proving correctness through proof by induction (see Project 2, part 4). The ability to find the general form for these recursive equations and then prove their correctness has provided me with the skills necessary to analyze time complexities of recursive algorithms such as quick sort and merge sort (see Merge Sort Analysis). In this particular example, I first found a general equation to describe the merge sort algorithm. Then, by determining the base case condition, I was able to put the general equation in terms of n, which I found to be $T(n) = an + cn \log_2 n$, implying that merge sort runs in $\Theta(n \log n)$. Using induction I was able to prove the correctness of this runtime (see Merge Sort Analysis - Proof).

Overall, algorithm analysis is an area that I now feel quite confident in. The methods of determining time complexities, loop invariants, and pre/post conditions, combined with my newly acquired proof techniques, have given me the ability to determine, compare and enhance the runtimes of both iterative and recursive algorithms, which is an important skill in any programming field.

Algorithm Design Techniques

Solving problems efficiently is perhaps the most fundamental aspect of computer programming. Throughout the course, my understanding of different techniques of problem solving increased greatly, and I now possess a firm grasp on the problem solving approaches of divide-and-conquer versus bottom-up dynamic programming.

The divide and conquer method of problem solving is quite common in computing, as the recursive abilities of modern computers allow for simple and easily implemented code solutions. In many cases, a recursive approach works quite effectively. In general, recursive solutions consist of three basic steps:

1. Divide - Separating a problem into multiple, smaller subproblems of the same type
2. Conquer - Solving subproblems, which is accomplished either by solving the subproblem directly if feasible, or by recursively dividing the subproblem until it is small enough to be solved efficiently.
3. Combine - Once a subproblem has been solved, it is combined with other solved subproblems to generate the correct aggregate solution.

As mentioned before, certain problems naturally lend themselves to being solved recursively. Many sorting algorithms rely on this divide and conquer method. For example, in Project 3 I implemented quick sort, which follows the steps mentioned above (see Project 3, timer.cpp). quick sort begins by a call to a partition method, which divides the array being sorted into two by randomly selecting a pivot point. Elements are then rearranged so that all values to the left of the pivot are less than the pivot value, and all values to the right are greater. This is the divide step. quick sort is then called recursively on the left and right subarrays, thereby dividing the array into smaller and smaller subproblems until the base case is reached, where the subarray consists of a single element. At this point the algorithm terminates, and the array is in sorted order (in the case of quick sort, the "combine" step is implicitly performed in the divide and conquer).

In certain problems, the divide and conquer method provides the solution in a highly efficient manner. quick sort runs on average in $\Theta(n \log n)$, which is significantly faster than an iterative sorting algorithm such as bubble sort ($\Theta(n^2)$).

However, in some problems, a recursive solution may perform exceptionally badly due to the huge amount of overlapping subproblems that must be solved. Recursive solutions to problems of this nature can take an enormous amount of time (on the order of $\Theta(2^n)$), and quickly become unsolvable from a practical standpoint. These problems generally involve optimization, in which a particular value must be optimized, with the solution of the problem being the method for achieving this optimization. For example, in Project 9 I was asked to design an algorithm which, given an input string, would compute the longest possible palindromic subsequence and output said palindrome (see

Project 9, writeup). The following recurrence depicts a solution to this problem:

$$LPS(i,j) = \begin{cases} 0 & \text{if } j < i \\ 1 & \text{if } j = i \\ LPS(i+1, j-1) + 2 & \text{if } V_i = V_j \\ MAX\{LPS(i+1, j), LPS(i, j-1)\} & \text{if } V_i \neq V_j \end{cases}$$

It is apparent by looking at this recurrence that solving the longest palindrome problem even for a small string requires many recursive calls. Furthermore, looking at the case where $V_i \neq V_k$, we can see that there is significant overlap, with many of the same subsequences of the string being analyzed multiple times. Thus, a recursive solution to this problem is extremely inefficient.

Rather, this problem begs for the application of dynamic programming. In short, dynamic programming is a "bottom up" approach to a problem that is too costly to solve with a divide and conquer method. Solving a problem through dynamic programming technique generally follows four basic steps:

1. Define the substructure of the optimal solution
2. Write a recurrence to define a solution
3. Compute the value of the optimal solution (and store via an accounting system)
4. Create the optimal solution via the accounting system created in step 3

In this particular example the optimal substructure was simply the largest palindromic sequence in a subsequence of the input string. Step two simply involves writing the recurrence detailed above. To complete step three, I had to devise a way in which to compute value of the optimal solution (in this case the length of the LPS) for any given subproblem, and store it in order to construct the solution in step 4. To do this, I made use of a matrix (two dimensional array), with length n by n, where n is the length of the input string. Please refer to Project 9, pal.cpp and observe the LPS_length method.

After filling in the matrix, the final step calls for the creation of the optimal solution using the matrix created in step 3. Devising a method to do this took some time, but I was able to create an algorithm that successfully constructed the longest palindromic subsequence. Because the n by n matrix corresponds to the length of the input string, each row and column index corresponds to a specific letter. By starting at the optimal value of the entire string (Matrix[0][n-1]) and working backwards towards the center, it was possible to determine the exact characters that were included in calculating the optimal value observing the recurrence case in which a value of two is added to the recursive call (see recurrence above). When this occurred in step three, this meant that two characters had found that optimized the LPS. Thus, if the optimal solution in any given matrix index was dependent on its southwest neighbor, it meant that the corresponding row and column indices were the indices of two characters in the LPS (see Project 9, pal.cpp for details). By calling this algorithm recursively, it became possible

to construct the LPS from the optimal values in the matrix. Most importantly, the functions for step three and four ran in $\Theta(n^2)$, greatly outperforming the potentially exponential time complexity of a recursive solution.

Although the LPS problem is just one instance of dynamic programming application, it serves as an excellent example of how in certain problems, bottom up approaches allow for an efficient solution to an otherwise extremely difficult problem to solve. However, it is important to remember that this is not always the case, and both divide-conquer and bottum-up paradigms play important roles in effective problem solving.

Linear Data Structures

Before this course, I had worked limitedly with the linear data structures List, Stack and Queue. However, throughout the semester, the importance of these ADTs became increasingly apparent as we began working with larger, more complex data structures.

The List data structure is perhaps the simplest form of linear data management. Lists can store any data type, assuming that that the type is defined. General list operations include the ability to determine whether the list is empty, insert a value into the list at a particular index (including at the end), iterate through values in the list, and remove a value from the list. The list must also have a distinct starting value, accessible to the previously mentioned methods.

There are two practical ways to implement a list in C++. The first is through the use of an array. In C++, an array is simply a pointer to a block of consecutive memory, which is divided into n parts, where n is the number of possible elements in the array. Because elements are stored consecutively in memory, any element in the array can be accessed in $\Theta(1)$ using its index. Thus, when a list is implemented with an array, accessing list elements is extremely efficient. However, one major drawback of the array implementation is that, since data must be stored consecutively in memory, inserting an element not at the end of the list will require that every element from the desired slot of insertion to the end of the array will will have to be shifted down by one, which will take on average $\Theta(n)$ time. Furthermore, because elements are stored consecutively, it is possible that the array will run out of space. However, in order for it to be a list, it must be able to store an undefined number of elements. Thus, the array implementation will require that a new block of memory of the desired size be dynamically allocated, and all elements of the old array be copied to the new array. This process takes $\Theta(n)$ time.

The alternative implementation of a list ADT is through the use of a linked list, which requires the use of a node ADT. Nodes simply contain a piece of data (can be of any type, so long as the node is a template class) which represents the list element, and one or more pointers that point to the subsequence and/or previous node (a list is considered "doubly linked" if each node contains a pointer to the previous node as well as one that points to the subsequent node). Unlike the array implementation, in a linked list elements are not stored consecutively in memory. Thus, in order to insert an element in the list, the pointer of the previous node must be changed to point to

the new node, and the new node's pointer must point to the next element in the list. Deletion of an element also follows this method. This process takes $\Theta(c)$, compared to the array implementation of $\Theta(n)$. Furthermore, since memory does not need to be stored consecutively, there is no case in which a block of memory larger than a single node must allocated to insert an element. The drawbacks of the linked list implementation is that there is no way to access elements through an index method (in the same manner as an array), meaning that the list must be stepped through one element at a time, starting at the head (or tail) to find the desired element, which takes $\Theta(n)$ in the worst case.

Throughout the course, I used both implementations of a list. In project 0, I used a linked list in my Set ADT (see Project 0, set.h). Because a set is simply defined as a collection of distinct objects, with an indeterminate number of elements, I elected to use a linked list in order to avoid having to dynamically allocate more memory should many elements be inserted into the set. Furthermore, because a set is unordered, it would be necessary to iterate through every element of the list (until found) in order to get or remove an element. This made the benefit of the array implementation's indexing irrelevant in this case.

In project 7, however, I elected to use a dynamically allocated array implementation in my Graph ADT (see Project 7, Graph.h). Graph requires a list of pointers to vertices to be stored (vertex array), which allowed for the graph methods to access vertices in the graph. Because the number of vertices in the graph is determined in the graph constructor by reading in an input file, and graph does not include an "insert vertex" operation, the number of elements in vertex array will never change. Thus, it would never be the case in which a new array would have to be allocated in order to store more elements. Furthermore, the graph vertices are indexed, so that the index of a vertex corresponds to its location in the vertex pointer list. Thus, using the array implementation, elements could be accessed in constant time.

Stacks and queues can also be implemented using either a dynamically allocated array or linked-list structure. A stack follows the last-in-first-out principle, meaning that at any given time, the stack methods should only have access to the most recently inserted element. A simple stack has two main methods, push and pop, which insert and delete elements, respectively. With an a dynamically allocated array implementation, the top of the stack is at index i. When a value is pushed, it is simply placed at slot $i+1$, and the top of the stack is incremented. The same method is used for pop. Since push and pop simply require access to indexed elements, both methods are $\Theta(c)$. In the linked-list implementation, a new node is simply added or removed from the top the stack by making the new node point to the node pointed to by head, and then making head point to the new node (in the case of push) or making head point to the value pointed to by the top of the stack and then deleting the top node. Push and pop both take constant time in this implementation as well. Thus, the difference between the two implementations lies in the size capacity. As was the case with list, the array implementation may run out of space requiring a costly reallocation and copy. However, if the number of elements is known, the array implementation avoids having to dynamically allocated and delete nodes. Thus, the choice between the two implementations is case dependent.

A queue follows a first-in-last-out principle. Thus, the queue must be able to insert a value at the tail (enqueue), and remove a value from the head (dequeue). Again, this data structure can be implemented using a linked list or an array. The method for enqueue and dequeue in the array implementation are very similar to that of the stack, and run in constant time. However, the performance of the linked-list implementation depends on the type of linked list used. If the list contains a tail node to which the last element points to, then elements can be inserted at the head and removed at the tail in constant time, as explained before. If the linked list does not have this tail node, then in order to dequeue, the entire list must be stepped through until the last element is found and deleted, which takes $\Theta(n)$. Again the linked list implementation avoids the case in which large blocks of memory must be allocated in order to insert more elements. Thus, the choice between implementations in a queue must depend on the number of potential elements to be added, as well as the type of linked-list available.

Priority Queues and Binary Heaps

Binary tree-based data structures were a central focus of this course. A binary tree is an acyclic, connected graph, where each node has at most two children. This basic concept is the foundation of many data structures implemented throughout the semester, each of which take advantage of the binary tree properties. Similar to stack, queue, and list, there are multiple ways to implement a binary tree. One way is to use a linked-list structure, using nodes vertices and pointers as edges. As discussed in the Linear Data Structures section, the benefit of this approach is the ability to insert an undefined number of elements to the tree without having to dynamically allocate new storage space. Another approach is to store vertices in an array. Because an array is stored continuously in memory, there is no need to create pointers between nodes. the children and parent of any vertex can be computed using the index i at which it is stored. Observe:

$$Left = 2i + 1$$

$$Right = 2i + 2$$

$$Parent = \left\lfloor \frac{i-1}{2} \right\rfloor$$

The heap data structure, which consists of a binary tree, utilizes the array implementation. A heap always contains a full binary tree, meaning that the second to last layer must always be full (every node in the third to last layer has two children), and all nodes in the bottom layer must be as far left as possible. Along with the binary tree, the heap ADT contains an integer variable "capacity" that represents the maximum number of elements that can be stored. Because of this capacity parameter, upon initialization, an array of size "capacity" can be declared, and this array will never have to be dynamically expanded to include more elements. However, this also means that the heap can become full, if the number of elements (heap size) is equal to capacity.

In project 3, I implemented a specific type of heap, called a minimum heap (see Project 3, heap.cpp & heap.h). A min-heap has all the same properties as a heap, with

the additional rule that all children must be "greater" than their parents. Thus, assuming the min-heap is not empty, the element stored at $A[0]$ is the "smallest" in the min-heap. The ability to assign order to elements in the heap is not within the scope of min-heap, as it simply assumes that whatever type element is, two instances of the type can be compared. In Project 3 I was simply using integers, however, in later projects the heap was used to store other data types, which will be discussed later. In order to maintain these essential properties, min-heap contains a recursive method called "heapify", whose postcondition is that all min-heap properties are maintained, and whose runtime is $\Theta(\log n)$. Therefore, to insert an element into the heap (assuming heap is not full), the new value must simply be inserted at the end of the array (which takes constant time because the end of the array is indexed by heap size), heap size is incremented then incremented (constant). Then, the newly inserted value is swapped with the first element of the array (constant time), and then heapify is be called ($\Theta(\log n)$). Thus, insert takes $\Theta(\log n)$. Similarly, to extract the smallest element in the min-heap, the first element must be swapped with the last (constant time), heap size must be decremented (constant), and heapify must be called ($\Theta(\log n)$). Thus, extracting the minimum value also takes $\Theta(\log n)$. These are the two main abilities of min-heap, whose use will become apparent in the following discussion of priority queues.

As mentioned before, in Project 3 I creating a min-heap. However, aside from standard unit testing, Project 3 did not involve direct application of the min-heap ADT. Rather, in Project 4, I implemented a min-priority queue data structure, which inherits a min-heap (see Project 4, MPQ.h). Although originally I incorrectly redefined the constructors (MPQ.h and MPQ.cpp have since been corrected), the min-priority queue successfully inherited all of min-heap's properties (except constructors). Similar to a standard queue, a priority queue is an ADT that stores a set of elements. However, instead of following FIFO properties, a priority queue extracts elements in order, based on a key value that is assumed to be included in the element. Similar to heap, there are two types of priority queues - minimum and maximum. A min-priority queue extracts elements in ascending order, using the extract minimum method inherited from min-heap. In fact, by definition of inheritance, the min-priority queue I created in project 4 *is* a min-heap, such that all inherited methods will have the same runtime.

In addition to the inherited methods, min-priority queue has a public "decrease key" function, which allows users to increase the priority of an element. Thus, a min-priority queue is very useful for managing a set of inserted elements and extracting them in increasing order. Because both "insert" and "extract minimum" run in $\Theta(\log n)$, min-priority queue performs better than inserting into a sorted array ($\Theta(n)$) and removing from a linked list ($\Theta(n)$). Of course the same is true for a max-priority queue, substituting "extract minimum" with "extract maximum". Thus, priority queues are ideal for situations in which you want to insert many values and extract them in ascending (descending) order.

After building a min-priority queue in Project 4, I put it to use by implementing a Huffman compression program (see Project 4, huffman.cpp), which requires multiple removals of the smallest element of a set, as well as multiple insertions. Thus, the min-priority queue was the choice method of storing said elements.

Graphs

Graphs are an extremely important topic in computer science. Because of their direct application to things such as network design, data structure analysis, etc., it is important to be able to represent a graph as an abstract data type. Doing so allows for important graph algorithms to be applied and analyzed in a C++ program.

There are two conventional ways to implement a graph ADT. The first is called an adjacency matrix, which consists of a two dimensional array of integers that detail the existence and weight of edges between vertices. Observe:

0 4 5
4 0 0
5 0 0

This adjacency matrix represents a graph with three vertices (because of the matrix's 3 by 3 size). Vertex 0 is connected to vertices 1 and 2 by an edge of weight 4 and 5, respectively. Vertices 1 and 2 are not connected. This simple matrix is an excellent way of representing dense graphs (graphs with many edges) because an edge connecting two vertices can be easily represented by two integers in the matrix, and most values will be nonzero.

The other common method of graph representation in an ADT is by an adjacency list. Similar to a hash table, an adjacency list is an array of pointers to linked lists. Each slot in the array represents a vertex, and the list rooted at that slot contains nodes that hold the index value of the adjacent vertices. If a graph is sparse, the adjacency list is generally preferable, because instead of having a two dimensional array (as is the case with the adjacency matrix), there is only a single array of size V, with few nodes in the attached linked lists (thus conserving memory). In Project 7, I implemented my graph class using an adjacency list because, for practical purposes, my test graphs were relatively sparse (see Project 7, graph.h).

Searching a graph is a fundamental graph operation. By "exploring" every edge in a graph, a searching algorithm discovers every vertex that can be reached from a particular starting vertex, finding the distance from the source to any given vertex in the graph (in the case in which a vertex cannot be reached from the source, a value of infinity is assigned as the distance). Depth-first search and breadth-first search are two basic searching algorithms that I became quite familiar with throughout the course. The basic idea behind depth-first search is, assuming a starting vertex s, to explore as far out from the source as possible until no new nodes can be discovered. Once this occurs, the algorithm backtracks until there are new nodes to explore. The algorithm does this by iterating through each vertex in the adjacency list, calling a recursive helper function that explores all the edges of the most recently "discovered" vertex (see Project 7, graph.cpp). The algorithm keeps track of the vertices by assigning a color to each node. A white node means that particular vertex has not been measured. A grey node means that it has been visited once, and that the algorithm has not yet returned to the node. Finally a black node is a a vertex that has been visited and all edges have been explored (see graph.cpp for details).

Breadth-first search uses an alternative method for exploring a graph. Instead of exploring as deep as possible each time, BFS first explores all vertices that are one step away from the source vertex, before moving on to the next level of exploration. Similar to DFS, BFS uses a coloring scheme to denote which nodes have been visited.

Another important graph operation is the ability to find a minimum spanning tree from a given source. Assuming that a graph is acyclic and connected, a spanning tree is a set of edges that connects all vertices in a graph. Thus, a *minimum* spanning tree is a set of edges that connects all vertices while minimizing the total edge weight of the set. There are two main algorithms we discussed throughout the course that complete this task. The first is Kruskal's algorithm, which makes use of a disjoint set data structure (see Project 7, disjoint_set.h & graph.cpp). Kruskal uses the disjoint set ADT to create forest of disjoint sets. Each set in the disjoint forest represents a tree in the graph. The algorithm proceeds by selecting edges in increasing order by weight (it does this by means of a minimum priority queue), and and then unioning the two trees to which the edge vertices belong (if they are not already the same, because otherwise algorithm would be creating cycles and thus not a tree) (see Project 7, graph.cpp). Upon completion of Kruskal, a minimum spanning tree has been established in the graph.

Prim's algorithm is similar to Kruskal's, and accomplishes the same goal. However, rather than forming multiple trees, Prim's algorithm establishes a single growing tree, starting at an arbitrary vertex. Using a min-priority queue, the algorithm stores vertices that are not connected to the growing tree in order of a key value, which represents the vertex's distance from a vertex in the tree. Prim selects a minimum vertex from the queue and iterates through the vertex's adjacency list. If the distance between the current tree and an adjacency is less than the weight of the edge connecting the vertex to the tree, and the adjacent vertex is in the priority queue (meaning it is not in the tree), then an edge is established between the adjacent vertex and the vertex already in the tree, thereby adding the adjacent vertex to the tree. The previous edge connecting the vertex to the graph is removed so as to not create cycles. This process ensures that a tree is created that spans the all vertices, and its total edge weight is minimized.

These graph methods that I have discussed have many important real world applications, both in computing and beyond. Being able to implement graph ADT and algorithms in C++ is an important step in being able to analyze and understand graphs in general. I feel that I have developed these skills throughout the course, and am now well-versed in many essential graph algorithms and implementations of graph ADTs.

Hash Tables

A very common ADT used in C++ and other programming is a dictionary (map). A dictionary stores key-value pairs (generally a large quantity), and allows users to get a pair back by simply passing a key value as a parameter. With this definition, it is clear as to why the name "dictionary" was chosen. Through abstraction, a dictionary user interacts with a dictionary only through the "get", "insert" and "remove" functions. However, the way in which a dictionary stores and manages data depends on the chosen back-end data structures. Proper selection of these underlying data structures is

extremely important, as the overall dictionary's performance will be dependent on the performance of the back-end.

One method for storing key-value pairs is through the use of a hash table, which I completed in Project 5 (see Project 5, hash_table.h & hash_table.cpp). Implemented correctly, a hash table can perform insertion and removal in constant time. The actual hash table architecture is fairly straightforward. A hash table contains a protected array of pointers to linked lists of type key type (key type here is simply the template type, because hash table is a template class). This array is protected as opposed to private, so that when hash table is inherited by dictionary, it can use the insert, get, and remove methods (see Project 5, dict.cpp).

Thus, the "table" can be thought of as a "vertical" array of pointers to "horizontal" linked lists, where each linked list's head is stored in the array. To insert an element into the table, the element is "mapped" to a particular slot in the array and then inserted at the head of the list rooted at the array slot. Accessing a particular slot in an array takes constant time, as well as inserting an element at the head of a linked list (see Linear Data Structures for explanation). Thus, inserting an element into a hash table takes $\Theta(c)$. For "get" and "removal", the process is quite similar; a particular array slot is mapped to using the key value (constant), and then the linked rooted at that slot is traversed through until the desired element is found (remember that deleting an element in a linked list takes constant time). While it is true that searching in a linked list takes $\Theta(n)$, it is important to observe that if we have an array with m slots, we have m linked lists. Thus, if m is significantly large, the number of elements in any particular linked list is approximately $\frac{n}{m}$ (assume for now that each linked list is equally utilized). Thus, if m is sufficiently large, we can say that delete and get run in $\Theta(\frac{n}{m}) = \Theta(c)$. Ideally, the data structure I just described performs insert, delete, and get in constant time. However, actualization of this performance is difficult.

In order to ensure that get and delete run in constant time, it vital that all linked lists in the table are of similar length. This means that, ideally, insert would map to each slot with equal frequency. However, this is a difficult task because the key value of a particular element must map to a particular slot consistently, in order for the key-value pair to be retrieved in a get or delete. Thus a hash function is required in order to map a key to an array slot. While this function is actually stored in the key type ADT, it must be tailored to the particular hash table being used so that all map values generated correspond to an array slot (typically the final step of a hash function is to mod some generated value k by the size of the array m, such that $0 \leq k \bmod m < m$). This value m is generally passed to the hash function as a parameter. A good hash function maps to every slot with near-equal frequency.

In Project 5, I was tasked with creating a hash table-inheriting dictionary to store a movie ADT. The movie class included two pieces of information, a movie title (key) and a cast list (value). I thus needed to create a method of mapping the movie ADT to a particular slot, using the movie's title as a key (see Project 5, movie.h). I theorized that there were many movies in the IMDB with similar titles. Thus, my overarching goal was to generate a value that was as far removed from the title as possible. I began by taking advantage of the ASCII values that are used to represent characters in C++. I decided

that by summing the ASCII value of each character in the title, I would generate a wide range of distinct numbers. Because I used a for loop to iterate through each character in the title, I decided to multiply the ASCII value of the character I was accessing on a particular iteration by the for loop's index value to generate an even wider range of values. My final step was to mod the final generated value by the number of slots in the hash table's array.

I found my hash function to be extremely effective in mapping movie titles evenly across array slots. In fact, the standard deviation of the mapping histogram was 9.97, which actually outperformed many randomly-generated distributions. Thus, the hash function I devised was extremely effective in minimizing the runtime of the hash table's insert and delete methods for reasons discussed previously. The hash function's performance also allowed the hash-based dictionary to run extremely efficiently, with insert, delete, and get running in $\Theta(c)$.

Binary Search Trees

Binary search trees are a very important data structure in C++ programming, and one that I spent significant time analyzing and implementing. As mentioned in the Binary Heap section, a binary tree is an acyclic, connected graph, where each node has at most two children. I also indicated in the heap section that there are multiple ways of implementing a binary tree. Unlike in Project 3 where I implemented a heap using an array, in Project 6 and Project 7 I built two binary search trees using nodes and pointers.

A binary search tree, along with satisfying the properties of a binary tree, must also satisfy the following condition:

for any node x in a binary search tree, the key value of every node in x's left subtree subtree is less than or equal to the key value of x, and the key value of every node in x's right subtree is greater than or equal to the key value of x.

Although a complete implementation of a binary search tree includes delete, successor/predecessor and various helper functions, I will only discuss the insert, search and minimum/maximum methods here, because these are the functions necessary for BST-based dictionary implementation (see Project 6, BST.h & BST.cpp for complete implementation).

Before discussing the methods, however, I will first prove the following theorem, which will assist in the analysis of BST methods:

Theorem

The height of a complete binary tree with n nodes is $h = \log_2(n+1) - 1$

PF
Base Case: $n = 1$

A complete binary tree with one node has $h = \log_2(1+1) - 1 = 0$.

Induction Hypothesis

Assume that a complete binary tree with k nodes has height $h = \log_2(k+1)-1$, *where* $k < n$

Induction Step

Consider a complete binary tree with n nodes. If we assume that x is the root node of said tree, then there exists two subtrees, one rooted at the left child of x and the other at the right child. Each of these subtrees have $\frac{n-1}{2}$ *nodes. Since* $n > \frac{n-1}{2}$, *we can assume that* $k = \frac{n-1}{2}$. *Thus, by the inductive hypothesis, each of these subtrees have* $h = \log_2(\frac{n-1}{2}+1)-1$ *Since x is the root of these two subtrees, we can say that the total height of the tree is* $h = \log_2(\frac{n-1}{2}+1)-1+1 = \log_2(\frac{n-1}{2}+1) = \log_2(\frac{n+1}{2}) = \log_2(n+1)-\log_2(2) = \log_2(n+1)-1$. *QED.*

Now that we have established that the height of a *complete* binary search tree is $h = \log_2(n+1) - 1$, we can begin discussing the runtimes of insert, search, and maximum/minimum. Assume for now that our BST is balanced. Search is perhaps the most intuitive method. Receiving a key value to search for and a node to start at as parameters, search simply checks if the value being searched for is the node parameter. If it is, then the algorithm terminates, returning the value of the node. Otherwise, the algorithm determines whether the value bing searched for is less than the current node. If it is, then search is simply called recursively on the left child of the current node. If not, search is called on the right child. Because we are assuming the tree is balanced, we will complete at most one comparison at each level of the tree. Thus, in a balanced tree, search runs at $\Theta(\log n)$. Similarly, the minimum and maximum methods will follow the left and right children, respectively, until the next child is null. Thus, given the binary search tree property, the minimum/maximum value in the tree has been found. These methods will take $\log_2(n+1) - 1$ comparisons, again assuming the tree is balanced. Finally, the insert method descends down through the tree like search, until the correct location for the node being inserted is found, which takes at most $\log_2(n+1) - 1$ steps. Once this location has found, it only takes constant time to insert the node and ensure the maintenance of the BST property. Thus, we see that insert and get (search) take $\Theta(\log n)$.

After constructing my binary search tree class, I declared a dictionary class that inherits insert, get (search), remove and empty methods from BST (see Project 6, dict.h).

As I did in Project 5, I used my BST-based dictionary to store movies from the IMDB database. However, unlike the hash-based dictionary, the BST-dictionary took

on the order of seconds (≈ 6.2) to insert all movies into the dictionary, where as the hash-based dictionary took on the order of milliseconds (≈ 0.04). Although insert in the hash-dictionary runs in $\Theta(c)$ versus the $\Theta(\log n)$ of BST-dictionary, a performance disparity of that magnitude was unprecedented.

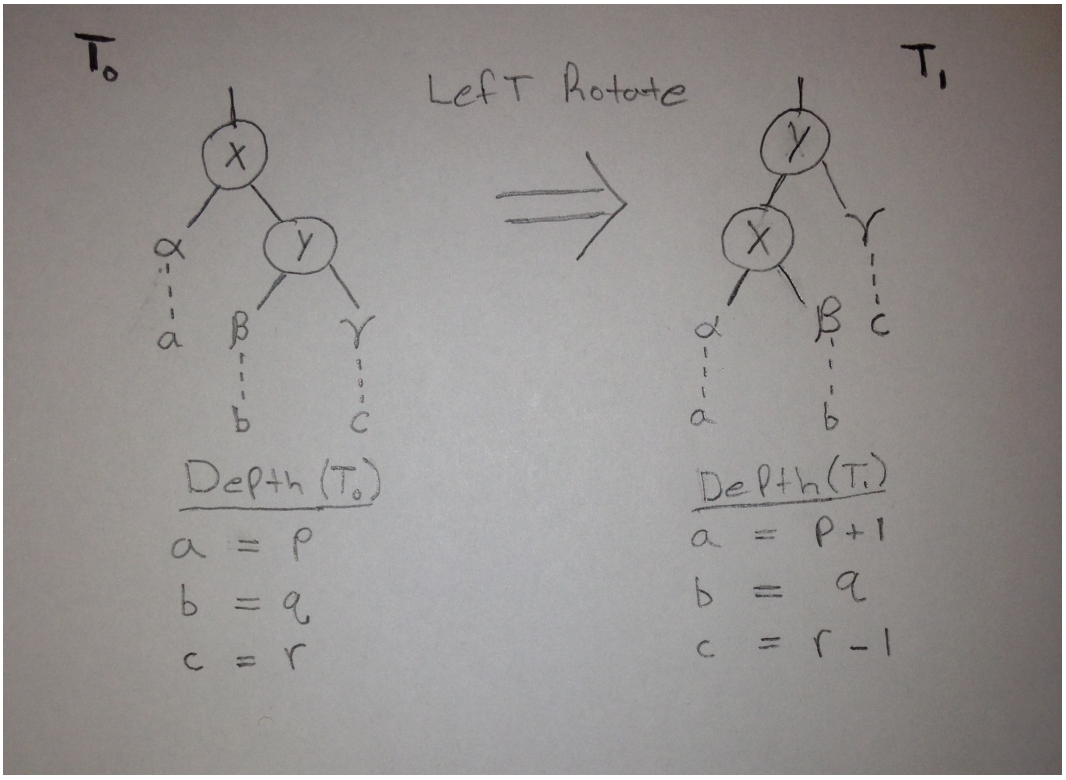
It turns out the reason for such poor performance of the BST-based movie dictionary is due to the fact that movie titles in the IMDB data base are in sorted alphabetical order. Because the key value of the movie ADT is the movie title, of each movie being inserted was less than all previously inserted movies. Thus insert, being concerned with maintaining the BST principle, was essentially just appending movies to the end of a linked list, having to iterate through every time insert was called. As this list grew, the runtime of insert became significantly slower, on the order of $\Theta(n)$. Thus, because the tree was "perfectly imbalanced" and incomplete, insert could not achieve $\Theta(\log n)$ as it did on a balanced tree.

This experience with the BST-dictionary taught me a very important lesson; when considering data structures for a particular task, it is essential to first engage in an in-depth analysis of possible candidates. Had I done so in Project 6, I would have realized how poorly a BST-based dictionary would perform with sorted data. Fortunately, the subsequent Project 8 provided me with the opportunity to implement an alternative data structure that would avoid such catastrophic consequences.

In project 8 I implemented a red-black tree data structure. Red-black trees (RBT) are a subcategory of binary search trees. The key difference (besides the addition of a color type to the node) is that RBT is a self-adjusting data structure , meaning that the tree structure is "approximately" balanced. A RBT must always maintain the following five properties:

1. Every node is either red or black
2. Root is always colored black
3. Leaves are black
4. If a node is red, both of its children are black
5. For each node, all paths from the node to descendent leaves contain the same number of black nodes

Because insertion and deletion are the only RBT methods that can change these properties, the post conditions of these two functions must include the maintenance of the five RBT properties. Although I did not implement the delete method, my RBT implementation successfully inserts values into the tree, while maintaining both the BST and RBT properties (see Project 8, RBT.cpp). Essentially insert in RBT is identical to BST, with two changes. First a newly inserted node is colored red. Then, an insert helper method "insert fix-up" is called (see Project 8, RBT.cpp). The postcondition of this method is the maintenance of the five RBT properties. This helper insertion helper method in turn relies on (among other things) two helper methods, right rotate and left rotate.



Left rotate and right rotate are sketched above (left rotate is moving from the left image to the right, and right rotate from the right to the left). These helper methods are responsible for the maintenance of property 5, which in turn is responsible (in coordination with the other properties) for maintaining the balance of the RBT.

After implementing the RBT, I build a RBT-based dictionary that inherits the insert and get methods. As I had done in Project 6, I again used this dictionary to insert movie from the IMDB database. Unlike the BST-based dictionary, inserting all movies from the IMDB file took approximately 0.5 seconds. This was much quicker than the BST-based dictionary. The reason for this, of course, is that the RBT is self-adjusting, and thus remained balanced despite the movies being inserted in alphabetical order. Thus, insertion of RBT achieved the $\Theta(\log n)$ that we generally expect from binary search trees.

Object-oriented Programming in C++

Object-oriented programming is an extremely powerful aspect of C++ programming language. The ability to design abstract data types is an extremely important tool in data structure construction, and was thus quite useful throughout the course.

Most data structures that I built throughout the semester consisted of template classes. In C++, template classes are abstract data types that do not specify the specific data type(s) with which the class will work. Instead, template classes use generic types,

which allow template ADTs to be built with a high level of abstraction. This is extremely useful, because generally programmers want to reuse previously created data structures in different applications than they were originally build for. Thus, along with their power to create a high level of abstraction while building ADTs, the template class method saves time and effort on behalf of the software engineer.

Throughout the course, the majority of data structures I built utilized template classes. For example, In Project 5, 6 & 8, I created three different data structures, each of which included a template class (see hash_table.h, BST.h and RBT.h). In each project, the final goal was to create a dictionary that inherited the data structure, and then use the dictionary to store each movie in the IMDB as a movie ADT (see movie_query.cpp). However, the IMDB database is extremely large, and the movie ADT somewhat complex. Thus, while constructing the data structure in each project I instead used a tester data type (see Project 5, 6 & 8, unit_test.cpp). This allowed me to build each data structure with a high level of abstraction, not having to worry about the movie ADT that would eventually become the data type stored in the structure. Once I had finished debugging each data structure using a tester data type, I was able to seamlessly implement a dictionary based on my template data structure that stored movie ADTs from the IMDB database.

Inheritance is another important aspect of dynamic programming in C++. In programming, inheritance refers to the ability of abstract data types to inherit methods and/or variables from other "base" classes. The "derived" class thus utilizes the same code as the base class for all inherited traits. Along with providing a high level of abstraction, the ability of a class to "inherit" attributes from another saves time and effort on the part of the software engineer by eliminating the need to rewrite multiple versions of similar or identical code.

Throughout the semester, I used inheritance in many of my projects, including Projects 5, 6, & 8, as mentioned above. However, the project in which I feel I learned the most about the concept of inheritance was Project 4. In this project, I first created a minimum-heap template class data structure. I was the tasked with creating a minimum-priority queue, which inherited various methods and variables from the min-heap (see Project 4, MPQ.h). Because this was the first time I had used inheritance in C++, upon initial completion of the project I erroneously defined the min-priority queue constructors in MPQ.cpp, whereas I should have been calling the min-heap constructors via the initializer lists in MPQ.h. However, I have since corrected this mistake. In the following projects that required inheritance, I made sure to correctly call the constructors of the base class in the initializer list of the derived class's .h file. Another important lesson I learned in Project 4 was the need to change any private methods/variables in the base class to protected if they were to be inherited by the derived class. In C++, items that are private to the base class will be private to the derived class, and thus unusable. Making these items protected instead allows the derived class to use them, but prevents any subsequent classes from doing so. This is a very important detail, and one that I successfully included in Project 4 and all subsequent projects.

Although inheritance in C++ provides many benefits, issues of polymorphism can create perplexing issues when using pointers with derived classes. Polymorphism

is when a call to a particular member function of a derived class erroneously calls the method of the base class. Generally the C++ compiler is able to avoid such confusion by determining which class is calling the method. However, when using pointers, the compiler cannot discern whether the method being called should be the method of the base class or the derived class. Clearly this can cause significant problems when the methods of the derived class vary from the base class. To overcome this issue, the built-in "virtual" method must be invoked. In general, the type of every variable calling a method must be determined upon program compilation. However, the virtual method allows the compiler to wait until the particular method is called before determining the calling class. For example, if Square is a derived class of Shape, and both classes have different "area" methods, the following code snippet would call the area method of Shape:

```
Square * S1 = new Square;
S1.area( );
```

However, by including a "virtual" before the declaration of area in Shape.h, the compiler will wait until area called in the program, thereby allowing it to know that $S1$ is a pointer to Square, and then Square's area must be called. By understanding when and how to implement this practice, issues of polymorphism in C++ can generally be avoided.

Professional Practices

Along with the extensive range of topics detailed in previous sections, this course has greatly improved my methods of professional practice. Throughout the semester, I never turned in a project past its due date, as I made it a habit to begin assignments as soon as they were made available. Furthermore, I never missed a day of class, and always made sure I was prepared to engage in lectures by completing the assigned daily reading. Whenever I had a question during in-class discussions, I always voiced my uncertainty so that my classmates and I were able to understand. When I had remaining questions after any particular class, I made it a point to meet with the professor and clarify.

In group projects, I always strove to create the best group dynamic possible, ensuring that every member, including myself, was involved in all steps of the project, and that no individual contributed significantly more or less effort than others. Working with randomly assigned partners was a particularly apt exercise in developing these group skills, which will undoubtedly pay high dividends through my remaining academic career and beyond.

All projects I turned in included adequate commenting in the proper format, and documentation when necessary. All written assignments were completed in the LatTeX typesetting system, as requested by the professor. Although I was new to the formatting system at the commencement of this course, I am now highly proficient in LaTeX typesetting, possessing the ability to create a professional-grade document of any type. In fact I have begun using LaTeX in courses outside of department courses, which demonstrates how the professional practices I have garnered over the semester translate to

other areas of academia and beyond (for an example of my LaTex ability, please see Portfolio - and analyze that recursion time ☺).

Although my official involvement with the Denison CS department will be concluded with the completion of this course, my time spent pursuing the study of computer science has been incredibly fruitful and enjoyable. Along with the technical and analytical skills that I have gained, I have acquired a deep appreciation for more intangible qualities, such as self-motivation, determination, and pride in one's work. Perhaps more than anything else, these skills will accompany me through the rest of my time as an undergraduate student, and into post-graduate academics and professional life.

Merge Sort Time Complexity

In this proof, we find the time complexity of Merge Sort and then prove its correctness.

Finding General Form

$$
\begin{aligned}
T(n) &= T(\frac{n}{2}) + T(\frac{n}{2}) + cn \\
&= 2(T(\frac{n}{2})) + cn \\
&= 2[2T(\frac{n}{4}) + c(\frac{n}{2})] + cn \\
&= 4T(\frac{n}{4}) + 2cn \\
&= 4[2T(\frac{n}{8} + c(\frac{n}{4})] + 2cn \\
&= 8T(\frac{n}{8}) + 3cn
\end{aligned}
$$

After several iterations, we observe distinct patters of growth in the equation, and we are able to begin construction of our general form:

$$= 2^i T(\frac{n}{2^i}) + icn$$

With the equation in general form, we see that Merge Sort will run until $2^i = n \Rightarrow T(1)$ when the algorithm reaches its base case. Thus:

$$2^i = n$$

$$i = \log_2 n$$

We can now write a definite general form of merge sort runtime by substituting in our value for i:

$$= 2^{\log_2 n} T(\frac{n}{2^{\log_2 n}}) + (\log_2 n)cn$$

Because $2^{\log_2 n} = n$ and $T(\frac{n}{2^{\log_2 n}})$ represents the base case, and therefore runs in constant time, we can simplify the general form to:

$$= an + cn \log_2 n$$

Thus, we can say that Merge Sort runs in $\Theta(n \log n)$.

Proof

Conjecture:

$$T(n) = an + cn \log_2 n$$

Base Case:
$n = 1$ (Constant Time)

$$T(n) = a = a(1) + c(1) \log_2 1$$
$$= a + c * 0$$
$$= a$$

Induction Hypothesis:
Assume

$$T(k) = ak + ck \log_2 k$$

Induction Step:

$$T(n) = 2(\frac{n}{2}) + cn$$

By induction hypothesis:

$$= 2[a(\frac{n}{2}) + c(\frac{n}{2}) \log_2(\frac{n}{2})] + cn$$
$$= an + cn(\log_2 n - \log_2 2) + cn$$
$$= an + cn(\log_2 n - 1) + cn$$
$$= an + cn \log_2 n - cn + cn$$
$$= an + cn \log_2 n$$

Q.E.D.

```
// Set.h
// set.h
// A Set ADT.
// This implementation uses a linked list.

#ifndef SET_H
#define SET_H
#include <string>
#include <iostream>

using namespace std;

template <class Element>
class Node
{
  public:

   Element value;
   Node<Element> *next;

   Node(Element item)
   {
      value = item;
      next = NULL;
   }
};

template <class Element>
class Set;

template <class Element>
ostream& operator<<(ostream& stream, const Set<Element>& s);

template <class Element>
class Set
{
 public:

   Set();                                    // default constructor
   Set(const Set<Element>& s);               // copy constructor
   ~Set();                                   // destructor

   void insert(const Element& x);            // add x to the set
   void remove(const Element& x);            // remove x from the set
   int cardinality() const;                  // returns size of the set
   bool empty() const;                       // returns true if empty, false o/w
   bool contains(const Element& x) const;    // true if x is in set, false o/w

   bool operator==(const Set<Element>& s) const;           // equality operator
   bool operator<=(const Set<Element>& s) const;           // subset operator
   Set<Element>& operator+(const Set<Element>& s) const;   // union operator
   Set<Element>& operator&(const Set<Element>& s) const;   // intersection operator
   Set<Element>& operator-(const Set<Element>& s) const;   // difference operator

   Set<Element>& operator=(const Set<Element>& s);         // assignment operator

   string toString() const;
   // stream insertion operator
   friend ostream& operator<< (ostream& stream, const Set<Element>& s)
   {
        cout<< "{";
        Node<Element> *current = s.head;
        while(current != NULL)
```

```
        {
                stream<<current->value;
                if(current->next != NULL)
                {
                stream<<", ";
                }
                current=current->next;
        }
        cout<< "}"<<endl;
        return stream;
   }
 private:

   Node<Element> *head;
   int length;

   void copy(const Set<Element>& s);   // copy the set s to this set
   void destroy();                     // delete all elements in the set
};

#include "set.cpp"

#endif
```

Ansel Schiavone
May 2, 2016

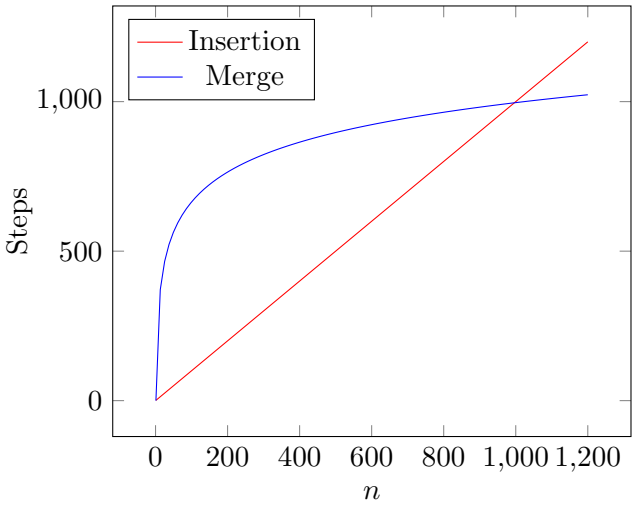
Computer Science 271
Project 1

---

1. If insertion sort runs in $4n^2$ steps and merge sort runs in $400nlog_2n$ steps, then, in order to determine the values of n for which merge sort is faster than insertion sort, we must find the integer value of n at which one algorithm overtakes the other. We can begin by setting the two equations equal to each other and simplifying.

$$4n^2 = 400nlog_2n$$
$$n^2 = 100nlog_2n$$
$$n = 100log_2n$$

Now, after simplifying both equations, we can graph the two equations and observe numbers into a chart and observe where one algorithm overtakes the other.



It is apparent that for small values of n, insertion sort is much more efficient, requiring fewer steps to sort the array. However, as n grows, merge sort becomes more efficient. The exact value of n at which merge overtakes insertion in efficiency is 996. Therefore - on this particular machine - for values of n between 1 and 996, insertion sort is more efficient, and for values greater than 996, merge sort is more efficient

2.

	1 second	1 minute	1 hour	1 day	1 year	1 century
$\lg n$	2^{1*10^6}	2^{6*10^7}	$2^{3.6*10^9}$	$2^{8.64*10}$	$2^{3.1536*10^{13}}$	$2^{3.156*10^{15}}$
$\sqrt{n}$	$1*10^{12}$	$36*10^{14}$	$12.96*10^{18}$	$74.64*10^{20}$	$99.45*10^{25}$	$99.45*10^{30}$
n	$1*10^6$	$6*10^7$	$3.6*10^9$	$8.64*10^{10}$	$3.1536*10^{13}$	$3.156*10^{15}$
$n \lg n$	62746	$28*10^5$	$13.3*10^9$	$27.6*10^8$	$79.8*10^{10}$	$68.6*10^{12}$
n^2	1000	7745	60000	293938	5616048	56160484
n^3	100	391	1532	4420	31595	146652
2^n	19	25	31	36	44	51
$n!$	9	11	12	13	16	17

3. a)
```
for(i = 0; i < n; i++)
{
    if(A[i] == v)
        return i;
}return NIL;
```

b) Before iteration j of the for loop, none of the values of subarray A[0... j-1] are equal to v and the elements in A[0... n-1] are the same as their initial values.

c) In order to prove the loop invariant to be true, we must first prove that before the first iteration of the for loop, where j = 0, none of the values of subarray A[0... j-1] are equal to v and the elements in subarray A[0... j-1] and A[j... n-1] are the same as their initial values. The first part of this statement is vacuously true because the subarray A[0...(0-1)] is empty, so there could not possibly be any values equal to v. The second part is also true because the subarray A[0...(0-1)] is empty and A[j... n-1] is the same as A[0... n-1]. Therefore, the loop invariant can be said to be true for the first iteration of the loop.

Now, assume that the loop invariant is true before some iteration of the for loop. Since (i < n) must be true in order to begin the loop, we know that A[i] is some value in the array A[0...n-1]. Furthermore, we know that none of the values of subarray A[0...i-1] are equal to v, because the loop terminates immediately if such a value is found. Finding A[i] = v is a termination condition of the loop. Therefore, given that we are at the start of some iteration of the loop, we know that none of the values of A[0..i] are equal to v. We also know that the values in A[0...i-1] and A[i...n-1] have remained unchanged, because i is simply incrementing every iteration until it is equal to n, in which case the loop is terminated.

By knowing the two termination conditions of this loop, we can prove that the linear search successfully performs its task. The two terminating conditions, when j $\geq$ n and when A[i] = v, ensure that the array is searched through until v is found, or the end of the list is reached. If the

former occurs, the value of i is returned, and if the latter, the value NIL.

d) Before the final iteration j of the for loop, none of the values of subarray A[0... j-1] are equal to v and the elements in A[0... n-1] are the same as their initial values. There are essentially two termination conditions for this linear search algorithm. First, if A[i] = v for some iteration, then the loop is immediately terminated and the value of i is returned. The second condition is if j $\geq$ n. Because the value n represents the number of elements in the array, n-1 iterations of the loop would effectively check every element in the array (A[0...n-1]). With the combination of these two termination conditions, we can conclude that after the termination of the for loop, either the value i where A[i] = v is being returned, or the value is NIL is being returned after every value in the array has been checked.

e) for(i = 0; i < n; i++) *n+1*
{
    if(A[i] == v) *n*
        return i; *+1*
}return NIL; *+1*

Worst case performance occurs for a linear search when it searches for an element that is not in the array. This means that the head of the for loop will executed exactly n+1 times, until the condition i < n is broken. Therefore, this implies that the comparison inside of the loop will execute exactly n times. The if-statement condition will never be true, so the "return i" line will not be executed. Once the for loop's i < n condition has been broken, the final line that will be executed is the "return NIL", which will run in constant time. Therefore, the time complexity equation can be described as:

$$T(n) = n + (n+1)$$
$$T(n) = 2n + 1$$

Therefore, the time complexity for a worst case linear search is $O(n)$, or linear time.

To calculate the average time complexity of the linear search algorithm, we must essentially add up all the possible amount of work done in each case, and divide that by n. This is because the value v could be at any element in the array (or may not be in the array at all). We can denote this as:

$$\frac{1}{n+1} * \sum_{i=0}^{n} (2i+1) = \frac{n+1}{2}$$

Therefore, the time complexity for the average linear search is $O(n)$ as well. Finally, lets consider the best case time complexity for the linear search. This

would be the scenario in which the element being searched for was at the start of the array, or A[0]. This would simply take constant time, because the body of the loop would only execute once, and only up until the body of the if-statement. Thus, the entire algorithm would be completed in three steps. Therefore, the best case scenario can be said to be *O(1)*, or constant time.

4. a) **Loop Invariant:**
Before each iteration of the inner for loop, we know that A[j] is the smallest element in A[j...n].

Proof:

In order to prove the loop invariant, we must begin by proving the base case. The base case, in this scenario, is the first iteration of the inner for loop. We know the value of j = n, because at the initialization of the first iteration of the inner loop, j is assigned the value of n. Therefore, the base case we must prove is that when j = n, A[j] is the smallest element in A[j...n]. Because j = n, we are essentially saying that A[n] is the smallest element in A[n]. Obviously this is true.

Now, we must prove that this loop invariant holds true for any iteration of the loop. Assume that the loop invariant before an iteration, j is true. Based on this assumption, we know that the A[j] is the smallest element in A[j...n]. From here we must observe what happens within the body of the inner for loop. We have a subarray A[j...n], where we know either A[j] is the smallest element in the subarray. The loop then compares A[j-1] and A[j]. If A[j-1] is greater than A[j], the two elements are swapped. If not, no action is completed. Therefore, we know that the smallest element in A[j-1...n] is A[j-1]. In the final part of the for loop (in the heading), j is decremented, so A[j] $\leftarrow$ A[j-1]. We have therefore shown how the invariant is successfully maintained through any iteration of the loop.

The termination condition of this inner loop occurs when j = i. Thus, we know that at the termination of this inner loop, we know that A[j]=A[i], which is the smallest element in A[i...n].

b) **Loop Invariant:**
Before each iteration of the outer for loop, we know that A[1...i] is in sorted order and all the elements in A[1...i-1] are less than [i...n].

Proof:

In order to prove the loop invariant, we must begin by proving the base case.

In this scenario, the base case is the first iteration of the outer for loop. We know that in the base case, i = 1. Therefore, we know that A[1...i] = A[1...1] = A[1]. It is therefore true to say that A[1...i] is in sorted order because A[1] is only one element. Furthermore, the second part of the statement is vacuously true, because A[1...1-1] is an empty array, meaning that its values are less than A[i...n]. Therefore, the loop invariant is true for the first iteration of the for loop.

Now, we must prove that this loop invariant is true for any iteration of the loop. Assume that the loop invariant before iteration i is true. Based on this assumption, we know that A[1...i-1] is in sorted order. We know that A[i] is the smallest element A[i...n], which we know will be greater than or equal to A[1...i-1], which is in sorted order. Therefore, by incrementing i at the end of the outer for loop, we know that A[1...i] $\leftarrow$ A[1...i-1] is in sorted order.

Finally, in order for the loop to terminate, i must be greater than or equal to n. This means that i has taken on every value of 1 through n.

c) In order to prove that the entire algorithm is correct, we need only to look at the termination condition of the outer for loop. The termination condition states that in order for the loop to stop iteration, i $\geq$ n. This implies that i has taken on every value between 1 and n, inclusive. However, when i = n, the loop does not iterate. Because the for loop iterates once for every value of i where i < n, we can say that for every value of A[i...n-1] is in sorted order, including A[1...n-1]. From this we can say that A[1...n-1] is in sorted order. Furthermore, based on the outer loop invariant, we know that all the elements in A[1...i] are lest than A[i+1...n]. Therefore, In the final iteration of the outer for loop when i = n-1 we can say all the elements in A[1...n-1] are less than the elements in A[n-1+1...n] = A[n]. Therefore, we can conclude that the entire array is in sorted order after the last iteration of the outer for loop.

d) To determine the worst case time complexities for bubble sort, we must consider a scenario that would require the maximum amount of steps to solve. It is guaranteed that the code inside the outer loop will run exactly n-1 times. Therefore, the outer for loop itself will run n times. For each iteration of the outer loop, the inner loop will run n-i times. Therefore, the total runtime of the inner loop can be described as the sum of n-i, where i takes on every value between 1 and n-1. The inside of the inner for loop runs in constant time, c. Therefore, the worst case time complexity of bubble sort can be described as:

$$T(n) = n + c * \sum_{i=1}^{n-1} n - i$$

$$= \frac{n(n+1)}{2} * c$$

$$= \frac{n^2+n}{2} * c$$

Therefore, the worst time complexity of bubble sort can be said to be:

$$T(n) = O(n^2)$$

For best time complexity, we have to consider a scenario in which the array can be sorted in the least amount of steps. First, however, lets consider what we know already. First, we know that the outer loop will still run n times, and the inside of the loop will still execute n-1 times. **Furthermore, we see that in this particular implementation of bubble sort, the inner loop will also always execute n-i times.** Therefore, we can describe best time complexity of bubble sort as:

$$T(n) = n + * \sum_{i=1}^{n-1} n - i$$

$$= \frac{n(n+1)}{2}$$

$$= \frac{n^2+n}{2}$$

Therefore, the best time complexity of this particular bubble sort is:

$$T(n) = O(n^2)$$

5. **Base Case:**

$$\sum_{i=0}^{0} 2^i = 2^{n+1} - 1$$

$$2^0 = 2^{0+1} - 1$$

$$1 = 1$$

Induction Hypothesis:

Assume that:

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

for any k = 0,1,2,3...

Induction Step:

We want to show that:

$$\sum_{i=0}^{k+1} 2^i = 2^{k+1+1} - 1$$

$$= \sum_{i=0}^{k} 2^i + 2^{k+1}$$

$$= 2^{k+1} - 1 + 2^{k+1}$$

$$= 2(2^{k+1}) - 1$$

$$= 2^{k+2} - 1$$

6. **Base Case:**

$$\sum_{i=1}^{1} i^2 = \frac{1(1+1)(2(1)+1)}{6}$$

$$1^2 = \frac{2*3}{6}$$

$$1 = 1$$

Induction Hypothesis:

Assume that:

$$\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$$

for any k = 1, 2, 3...

Induction Step:

We want to show that:

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)(k+1+1)(2(k+1)+1)}{6}$$

$$= \frac{k(k+1)(2k+1)}{6} + (k+1)^2$$

$$= \frac{k(k+1)(2k+1)}{6} + \frac{6(k+1)^2}{6}$$

$$= k+1(\frac{k(2k+1)+6(k+1)}{6})$$

$$= k+1(\frac{2k^2+7k+6}{6})$$

$$= k+1(\frac{2k^2+3k+4k+6}{6})$$

$$= k+1(\frac{(k+2)(2k+3)}{6})$$

$$= \frac{(k+1)(k+1+1)(2(k+1)+1)}{6}$$

Ansel Schiavone
May 3, 2016

Computer Science 271
Project 0010

---

1. The statement "The running time of algorithm A is at least $O(n^2)$" is meaningless. To demonstrate this, let's first take a look at the formal definition of $O(f(n))$.

   $O(f(n)) = \{g(n) :$ There exists constants $c > 0$ and $n_0 > 0$ such that $g(n) \leq c*f(n)$ for all $n \geq n_0\}$

   What this is saying is that $O(f(n)) = g(n)$, which implies that $f(n)$ is growing faster than some $g(n) \in O(f(n))$. Thus, for all values of $n \geq n_0$, $g(n) > f(n)$. Therefore, saying that an algorithm A is at least $O(n^2)$ is meaningless because if the running time of $A = O(n^2)$, then n^2 is strictly greater than A. However, the term "at least" implies that $A \geq n^2$. Therefore, $O(n^2)$ cannot possibly describe algorithm A.

2. Theorem: For any two functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

   Proof: In order to prove this theorem true, we must look at the definitions of $O(g(n))$, $\Theta(g(n))$, and $\Omega(g(n))$. Suppose $f(n) = O(g(n))$. This means that $f(n) \geq c * g(n)$ for all values of $n > n_0$, where c is some positive constant. $g(n)$ is essentially bounded above by $c * f(n)$. Now suppose $f(n) = \Omega(g(n))$. This means that $c * f(n) \leq g(n)$, for all values of $n > n_0$. $g(n)$ is bounded below by $c * f(n)$. Now, let us consider $\Theta(g(n))$. Suppose $f(n) = \Theta(g(n))$. This means that, where c_1 and c_2 are both positive constants, $c_1 * f(n) \leq g(n) \leq c_2 * f(n)$. Thus, $g(n)$ is growing equal to $f(n)$, differing only by the constant multipliers. Now, with the definitions in place, we can demonstrate $(f(n) = \Theta(g(n))) \Leftrightarrow (f(n) = O(g(n))$ and $f(n) = \Omega(g(n)))$. If $f(n) = \Theta(g(n))$, we know that $f(n)$ is growing at the same rate as $g(n)$. Thus, by varying the value of the constant, c, $c * f(n) \geq g(n)$, which, by definition means that $f(n) = \Omega(g(n))$ and $c * f(n) \leq g(n)$, which my definition means that $f(n) = O(g(n))$. Therefore, based on the previous definitions, we see that, if $f(n) = \Theta(g(n))$, then $c_1 * f(n) \leq g(n) \leq c_2 * f(n)$ for all $n \geq n_0$. Thus, $(f(n) = \Theta(g(n))) \Rightarrow (f(n) = O(g(n))$ and $f(n) = \Omega(g(n)))$. Conversely, we know that if $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$, we know that there exists some constant $c > 0$ and $n > n_0$ such that $f(n) \geq c*g(n)$, by definition of $\Omega(g(n))$. We also know that there exists some constant $c > 0$ and $n > n_0$ such that $f(n) \leq c * g(n)$, by definition of $O(g(n))$. This means that there exists three positive constants, c_1, c_2 and n_0 such that for all $n \geq N$, $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$, which, by definition, means that $f(n) = \Theta(g(n))$ Thus, we have proved the theorem $(f(n) = O(g(n))$ and $f(n) = \Omega(g(n))) \Leftrightarrow (f(n) = \Theta(g(n)))$.

3. a) $n^2 + 3n - 20 = O(n^2)$

Proof:

In order to prove this, we must choose values of $c > 0$ and $n_0 > 0$, such that $n^2 + 3n - 20 \leq c * (n^2)$ for all $n \geq n_0$. Let $c = 2$ and $n_0 = 1$. We know that $n^2 + 3n - 20 < 2 * n^2$ when $n = 1$. Furthermore, because $c > 1$, we know that $2n^2$ will grow faster than $n^2 + 3n - 20$. Thus, $n^2 + 3n - 20 = O(n^2)$.

b) $n - 2 = \Omega(n)$

Proof:

In order to prove this, we must choose values of $c > 0$ and $n_0 > 0$, such that $n - 2 \geq c * n$ for all $n > n_0$. Let $c = \frac{1}{2}$ and $n_0 = 4$. Thus, when $n = 4$, $\frac{1}{2}n = n - 2$. As $n \to \infty$, $n - 2$ will grow faster than $\frac{1}{2}n$. Therefore, we know that $n - 2 = \Omega(n)$.

c) $log_{10}n + 4 = \Theta(log_2 n)$

Proof:

In order to prove this, we must choose values of $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that $c_1 * log_2 n \leq log_{10}n + 4 \leq c_2 * log_2 n$ for all $n \geq n_0$. Let $c_1 = \frac{1}{4}$ and $c_2 = 2$ and $n_0 = 5$. When $n = 5$, $\frac{1}{2} * log_2 n < log_{10}n + 4 = c_2 * log_2 n$. As $n \to \infty$, $\frac{1}{2} * log_2 n < log_{10}n + 4 < c_2 * log_2 n$. Thus, we have proven that $log_{10}n + 4 = \Theta(log_2 n)$.

d) $2^{n+1} = O(2^n)$

Proof:

In order to prove this, we must choose values of $c > 0$ and $n_0 > 0$ such that $2^{n+1} \leq c_0 * 2^n$ for all $n_0 \geq n$. Let $c_0 = 3$ and $n_0 = 1$. When $n = 1$, $2^{n+1} < 3 * 2^n$. As $n \to \infty$, $2^{n+1} < 3 * 2^n$. Thus, we have proven that $2^{n+1} = O(2^n)$.

e) $\ln n = \Theta(log_2 n)$

Proof:

In order to prove this, we must choose values of $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that $c_1 * log_2 n \leq \ln n \leq c_2 * log_2 n$. Let $c_1 = \frac{1}{2}$, $c_2 = 1$, and $n_0 = 1$. When $n = 1$, $\frac{1}{2} * log_2 n = \ln n = 1 * log_2 n$. As $n \to \infty$, $\frac{1}{2} * log_2 n < \ln n < 1 * log_2 n$. Thus, we have proven that $\ln n = \Theta(log_2 n)$.

f) $n^{\in} = \Omega(\lg n)$ for any $\in > 0$

Proof:

Assume $n > 0$ and $\epsilon > 0$. Now, let us take the first derivative of n^ϵ and $lg(n)$.

$$f' = \epsilon n^{(\epsilon-1)}$$

$$g' = \frac{1}{ln(2)n}$$

Both derivatives are positive, indicating that both functions are increasing. We know this to be true, given the assumption that $n > 0$ and $\epsilon > 0$. Now, let us take the second derivative of both functions:

$$f'' = (\epsilon - 1)\epsilon n^{(\epsilon-2)} = \epsilon^2 - \epsilon n^{(\epsilon-2)}$$

$$g'' = -\frac{1}{ln(2)n^2}$$

Based on our assumption that $n > 0$ and $\epsilon > 0$, we can conclude that f'' is positive, because a positive to any power is still a positive, and a positive multiplied by a positive will always be a positive. This implies that the slope of f is increasing. Thus, we can conclude that f is increasing at an increasing rate. We observe, however, that g'' is negative. This implies that the slop of g is decreasing. Thus, we can conclude that g is increasing at a decreasing rate. Therefore, if n^ϵ is increasing at an increasing rate, and $lg(n)$ is increasing at a decreasing rate, we can assume that n^ϵ will overtake $lg(n)$ for sufficiently large values of n.

4. a) $T(n) = 2T(n/2) + n^3$

General Form:

$$T(n) < an + 2n^3$$

Proof:
Base Case:

$$T(1) < an + 2n^3$$

$$a < a(1) + 2(1)^3$$

$$a < a + 2$$

Want to show:

$$T(n) < an + 2n^3$$

Hypothesis: Assume

$$T(k) < ak + 2k^3$$

Induction Step:

$$T(n) = 2T(\frac{n}{2}) + n^3$$
$$< 2\Big[a(\frac{n}{2}) + 2(\frac{n}{2})^3\Big] + n^3$$
$$= 2a(\frac{n}{2}) + 4(\frac{n}{2})^3 + n^3$$
$$= 2a(\frac{n}{2}) + 4(\frac{n^3}{2^3}) + n^3$$
$$= 2a(\frac{n}{2}) + \frac{4n^3}{8} + n^3$$
$$= an + \frac{3n^3}{2}$$

Thus, because $an + \frac{3n^3}{2} < an + 2n^3$, we have proved that $T(n) < an + 2n^3$ through the property of transitivity.

b) $T(n) = T(9n/10) + n$

General Form:

$$T(n) < a + 10n$$

Proof:
Base Case:

$$T(1) < a + 10n$$
$$a < a + 10(1)$$
$$a < a + 10$$

Want to Show:

$$T(n) < a + 10n$$

Assume:

$$T(k) < a + 10k$$

Induction Step:

$$T(n) = T(9n/10) + n$$
$$< \Big[a + 10(\frac{9n}{10})\Big] + n$$
$$= a + 9n + n$$
$$= a + 10n$$

Thus, we have proved that $T(n) < a + 10n$

c) $T(n) = 7T(n/3) + n^3$

General Form:

$$T(n) < 7^{log_3 n} * a + 4.5n^2$$

Proof:
Want to Show:

$$T(n) < 7^{log_3(n)} * a + 4.5(n)^2$$

Base Case:

$$T(1) < 7^{log_3 n} * a + 4.5(n)^2$$
$$a < 7^0 * a + 4.5(1)$$
$$a < a + 4.5$$

Assume:

$$T(k) < 7^{log_3 k} * a + 4.5(k)^2$$

Induction Step:

$$T(n) = 7T(n/3) + n^2$$
$$< 7\Big[7^{log_3 \frac{n}{3}} * a + 4.5(\frac{n}{3})^2\Big] + n^2$$
$$= 7^{1+log_3 \frac{n}{3}} * a + 3.5(n)^2 + n^2$$
$$= 7^{1+log_3 \frac{n}{3}} * a + 4.5(n)^2$$
$$= 7^{1+log_3 n - log_3 3} * a + 4.5(n)^2$$
$$= 7^{log_3 n} * a + 4.5(n)^2$$

Thus we have proved that $T(n) < 7^{log_3(n)} * a + 4.5(n)^2$.

d) $T(\sqrt{n}) + 1$

General Form:

$$2a + lg(lgn)$$

Proof:
Base Case:

$$T(2) = a + lg(lg2)$$
$$a < 2a$$

Assume:

$$T(k) < 2a + lg(lg(k)$$

Want to Show:

$$T(n) = T(\sqrt{n}) + 1$$

$$< 2a + lg(lg(n^{\frac{1}{2}}) + 1$$
$$= 2a + lg(\frac{1}{2}lg(n)) + 1$$
$$= 2a + lg(\frac{1}{2}) + lg(lg(n)) + 1$$
$$= 2a + lg(lg(n))$$

Thus, we have shown that $T(n) < 2a + lg(lg(n))$

e) $T(n) = T(n-1) + lgn$

General Form:

$$2a + nlg(n)$$

Proof:
Want to Show:

$$T(n) < a + nlg(n)$$

Base Case:

$$T(1) = 2a + 1 * lg(1)$$
$$a < 2a$$

Assume:

$$T(k) = 2a + klg(k)$$

Induction Step:

$$T(n) = T(n-1) + lgn$$
$$< \Big(2a + (n-1)lg(n-1)\Big) + lg(n)$$
$$= 2a + (n-1)lg(n-1) + lgn$$
$$= 2a + nlg(n-1) - lg(n-1) + lgn$$
$$= 2a + nlg(n-1) + lg\frac{n}{n-1}$$
$$\leq 2a + nlg\Big((n-1) * (\frac{n}{n-1})\Big)$$
$$= 2a + nlgn$$

Thus, we have proved that $T(n) < a + nlg(n)$.

5. **Proof:**

Assume:

$$F_k = \frac{(\frac{1+\sqrt{5}}{2})^k - (\frac{1-\sqrt{5}}{2})^k}{\sqrt{5}}$$

Want to show:

$$F_{k+1} = F_k + F_{k-1}$$

$$= \frac{(\frac{1+\sqrt{5}}{2})^k - (\frac{1-\sqrt{5}}{2})^k}{\sqrt{5}} + \frac{(\frac{1+\sqrt{5}}{2})^{k-1} - (\frac{1-\sqrt{5}}{2})^{k-1}}{\sqrt{5}}$$

$$= \frac{1}{\sqrt{5}}\Big((\frac{1+\sqrt{5}}{2})^k - (\frac{1-\sqrt{5}}{2})^k\Big) + \frac{1}{\sqrt{5}}\Big((\frac{1+\sqrt{5}}{2})^k \frac{1+\sqrt{5}}{2})^{-1} - (\frac{1-\sqrt{5}}{2})^k (\frac{1-\sqrt{5}}{2})^{-1}\Big)$$

$$= \frac{1}{\sqrt{5}}(\frac{1+(\sqrt{5}}{2})^k\Big(1 + \frac{1+\sqrt{5}}{2})^{-1}\Big) - \frac{1}{\sqrt{5}}(\frac{1-(\sqrt{5}}{2})^k\Big(1 + (\frac{1-\sqrt{5}}{2})^{-1}\Big)$$

Aside:

$$1+(\frac{1+\sqrt{5}}{2})^{-1} = \frac{1+\sqrt{5}}{1+\sqrt{5}} + (\frac{2}{1+\sqrt{5}}) = \Big(\frac{3+\sqrt{5}}{1+\sqrt{5}}\Big) * \frac{1-\sqrt{5}}{1-\sqrt{5}} = -2\frac{\sqrt{5}+1}{-4} = \frac{1+\sqrt{5}}{2}$$

Aside:

$$1+(\frac{1-\sqrt{5}}{2})^{-1} = \frac{1-\sqrt{5}}{1-\sqrt{5}} + (\frac{2}{1-\sqrt{5}}) = \Big(\frac{3-\sqrt{5}}{1-\sqrt{5}}\Big) * \frac{1+\sqrt{5}}{1+\sqrt{5}} = -2\frac{\sqrt{5}-1}{-4} = \frac{1-\sqrt{5}}{2}$$

Now, by inserting these new values:

$$F_{k+1}\frac{1}{\sqrt{5}}(\frac{1+(\sqrt{5}}{2})^k\Big(\frac{1+\sqrt{5}}{2}\Big) - \frac{1}{\sqrt{5}}(\frac{1-(\sqrt{5}}{2})^k\Big(\frac{1-\sqrt{5}}{2}\Big)$$

$$= \frac{(\frac{1+(\sqrt{5}}{2})^{k+1} - (\frac{1-(\sqrt{5}}{2})^{k+1}}{\sqrt{5}}$$

```
// heap.h
// a binary min heap

#ifndef HEAP_H
#define HEAP_H

#include <iostream>

const int DEFAULT_SIZE = 100;

template <class KeyType>
class MinHeap
{
  public:
    MinHeap(int n = DEFAULT_SIZE);          // default constructor
    MinHeap(KeyType initA[], int n);        // construct heap from array
    MinHeap(const MinHeap<KeyType>& heap);  // copy constructor
    ~MinHeap();                             // destructor

    void heapify(int index);          // heapify subheap rooted at index
    void buildHeap();                 // build heap
    void heapSort(KeyType sorted[]);  // heapsort, return result in sorted

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

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

    int leftChild(int index) { return 2 * index + 1; }  // return index of left child
    int rightChild(int index) { return 2 * index + 2; } // return index of right child
    int parent(int index) { return (index - 1) / 2; }   // return index of parent
    void swap(int index1, int index2);        // swap elements in A
    void copy(const MinHeap<KeyType>& heap);  // copy heap to this heap
    void destroy();                           // deallocate heap
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinHeap<KeyType>& heap);

#include "heap.cpp"

#endif
```

```
#include <iostream>
#include <sstream>
using namespace std;

/*Preconditions: a MinHeap instance must be declared
with a specified template type.

Postconditions: an array of size n*Keytype is dynamically
allocated, and capacity is set equal to n. heapSize is assigned
the value of 0 becuase no heap has been created.*/

template <class KeyType>
MinHeap<KeyType>::MinHeap(int n)//default constructor
{
        A = new KeyType [n];
        capacity = n;
        heapSize = 0;
}

/*Preconditions:a MinHeap instance must be declared
with a specified template type,and an array and matching
n value passed as parameters. (NOTE: n MUST be the size
of the array being passed).

Postconditions:an array of size n*Keytype is dynamically
allocated, and capacity is set equal to n.All elements in
the array parameter are assined to the newly allocated
array, A.heapSize is assigned the value of 0 becuase no heap
has been created.*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(KeyType initA[], int n)//construct heap from array
{
        A = new KeyType [n];
        for(int i=0; i<n; i++)
        {
                A[i]=initA[i];
        }
        capacity = n;
        heapSize = 0;
}

/*Preconditions: a MinHeap instance must be declared
with a specified template type and a previously declared
MinHeap instance of matching template type passed in parameter.

Postconditions:The MinHeap instance utilizing the copy constructor
will be exactly the same as the MinHeap passed into the parameter.*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(const MinHeap<KeyType>& heap)//copy constructor
{
        copy(heap);
}

/*Preconditions: a MinHeap instance must have been previously
declared with a specified template type.

Postcondition: The memory that was dynamically allocated for the
MinHeap is deallocated and returned to the heap.*/
template <class KeyType>
MinHeap<KeyType>::~MinHeap()
{
        destroy();
}
```

```
/*Preconditions: a MinHeap instance must have been previously
declared with a specified template type. This MinHeap instance
must then call heapify with an integer index parameter.

Postcondition:The index parameter is now the root of a
complete MinHeap.*/
template <class KeyType>
void MinHeap<KeyType>::heapify(int index)
{
        int left, right, smallest;
        left = leftChild(index);
        right = rightChild(index);
        if(left < heapSize && A[left]<A[index])
                smallest = left;
        else
                smallest = index;
        if(right<heapSize && A[right]<A[smallest])
                smallest=right;
        if(smallest != index)
        {
                swap(index, smallest);
                heapify(smallest);
        }
}

/*Precondition: a MinHeap instance must have been previously
declared with a specified template type. This MinHeap instance
must then call buildHeap.

Postcondition: the MinHeap array value A is now a complete
MinHeap.*/
template <class KeyType>
void MinHeap<KeyType>::buildHeap()
{
        heapSize=capacity;
        for(int i=(capacity/2)-1;i>=0;i--)
        {
                heapify(i);
        }
}


/*Precondition:a MinHeap instance must have been previously
declared with a specified template type. This MinHeap instance
must then call heapSort, with a previously declared array of
matching KeyType as a parameter.

Postcondition: The array sorted[] will now be in sorted descending
order.*/
template <class KeyType>
void MinHeap<KeyType>::heapSort(KeyType sorted[])
{
        buildHeap();
        for(int i = (capacity-1);i>=1; i--)
        {
                swap(0, i);
                heapSize--;
                heapify(0);
        }
        for(int j=0; j<capacity; j++)
        {
```

```
                sorted[j]=A[j];
        }
}

/*Precondition: two MinHeap instances must have been previously
declared with matching specified template types. One of these
instances must then be assigned to the other in the form of
H1=H2

Postcondition: H1 will now be exactly identical to H2, because
the assignment operator will return a copy of H2 to H1.*/
template <class KeyType>
MinHeap<KeyType>& MinHeap<KeyType>::operator=(const MinHeap<KeyType>& heap)
{
        if(this != &heap)
        {
                destroy();
                copy(heap);
        }
        return *this;
}


/*Precondition: a MinHeap instance must have been previously
declared with a specified template type.

Postcondition: a string value "string_heap" will be returned
that depicts the format of array A.*/
template <class KeyType>
string MinHeap<KeyType>::toString() const
{
        string string_heap;
        KeyType temp;
        string_heap = "[";
        for(int i=0; i<capacity; i++)
        {
                ostringstream convert;
                temp = A[i];
                convert << temp;
                string_heap+=convert.str();
                if(i!=(heapSize-1))
                {
                        if(i!=(capacity-1))
                                string_heap+=", ";
                }
                if(i==(heapSize-1))
                {
                        string_heap+="|";
                }
        }
        string_heap+="]";
        return string_heap;
}

/*Precondition: two MinHeap instances of matching types
must have been previously declared. copy must then be called
by another member function(because it is private) with H2 as
a parameter.

Postcondition: H1 will now have the exact same representation
as H2.*/
template <class KeyType>
void MinHeap<KeyType>::copy(const MinHeap<KeyType>& heap)
```

```
{
        A = new KeyType [heap.capacity];
        for(int i=0; i<heap.capacity; i++)
        {
                A[i]=heap.A[i];
        }
        capacity=heap.capacity;
        heapSize=heap.heapSize;

}

/*Precondition: a MinHeap instance must have been previously
declared with a specified template type. A member function must
then call the swap method with two inbound integer indexes passed
as parameters (because it is a private method).

Postcondition: the elements in MinHeap's A, A[index1] and
A[index2] will be swapped.*/
template <class KeyType>
void MinHeap<KeyType>::swap(int index1, int index2)
{
        KeyType temp;
        temp = A[index1];
        A[index1]=A[index2];
        A[index2]=temp;

}
/*Precondition: a MinHeap instance must have been previously
declared with a specified template type. A member function must
then call the destroy method (because it is a private method)

Postcondition: the dynamically allocated array A of the MinHeap
instance will be deallocated.*/
template <class KeyType>
void MinHeap<KeyType>::destroy()
{
        delete []A;
}

/*Precondition: a MinHeap instance must have been previously
declared with a specified template type.

Postcondition: a string representation of MinHeap's A array
will be returned to stream and subsequently outputted to
the terminal.*/
template <class KeyType>
ostream& operator<<(ostream& stream, const MinHeap<KeyType>& heap)
{
        stream<<heap.toString();
        return stream;
}
```

```
#include <sys/time.h>
#include <iostream>
#include <fstream>
#include "heap.h"

using namespace std;


int partition(int array[], int start, int end)
{
    int middle=(start+end)/2;//find middle of list
    int pivot_val=array[middle];//assign pivot_val to middle value
    int pivot_pos=middle;//assing pivot position to middle index
    for (int pos=start+1; pos<=end;pos++)//traverses list
    {
        if(array[pos]<pivot_val)//if value is less than pivot value...
        {
            //swap value with some value greater than pivot value
            swap(array[pivot_pos+1], array[pos]);
            swap(array[pivot_pos], array[pivot_pos+1]);
            pivot_pos++;
        }
    }
    return pivot_pos;
}



void quick_sort( int array[], int start, int end)
{
    if(start<end)//BASE CASE
    {
        int p = partition(array, start, end);//p is the partition point of the list
        quick_sort(array, start, p-1);//calls quick_sort recursively for both sides of the lis
t
        quick_sort(array, p+1, end);
    }
}


void bubble_sort(int n, int array[])
{
        int i, j;
        int temp;
        for(int i=0; i<n; i++)
                for(int j=0; j<(n-i-1); j++)
                {
                        if(array[j]>array[j+1])
                        {
                                temp = array[j];
                                array[j]=array[j+1];
                                array[j+1] = temp;
                        }
                }
}

void insertion_sort(int array[], int length)
{
        int j, i, temp;
        for(i=1; i<length; i++)
        {
                j=i;
                while(j>0 && array[j] < array[j-1])
```

```
                {
                        temp = array[j];
                        array[j]=array[j-1];
                        array[j-1]=temp;
                        j--;
                }
        }
}

int main()
{
        ofstream myfile;
        myfile.open("insertion_sort.txt");


        for(int i=0; i<=2000; i+=100)
        {
                int n = i*100;
                int array[n];
                //MinHeap<int> H1(n);
                timeval timeBefore, timeAfter;
                long diffSeconds, diffUSeconds;

                gettimeofday(&timeBefore, NULL);
                //H1.heapSort(array);
                //bubble_sort(n,array);
                //quick_sort(array,0,n-1);
                insertion_sort(array, n);
                gettimeofday(&timeAfter, NULL);
                diffSeconds = timeAfter.tv_sec - timeBefore.tv_sec;
                diffUSeconds = timeAfter.tv_usec - timeBefore.tv_usec;
                myfile<<n<<" "<<diffSeconds + diffUSeconds/1000000.0<<endl;
        }

        myfile.close();



}
```

```
// pq.h
// This MinPriorityQueue template class assumes that the class KeyType has
// overloaded the < operator and the << stream operator.

#ifndef PQ_H
#define PQ_H

#include <iostream>
#include "heap.h"

template <class KeyType>
class MinPriorityQueue : public MinHeap<KeyType>
{
  public:
    MinPriorityQueue():MinHeap(100){}//default constructor
    MinPriorityQueue(int n):MinHeap(n){}//consructor
    MinPriorityQueue(const MinPriorityQueue<KeyType>& pq):MinHeap<KeyType>::MinHeap(const MinH
eap<KeyType>& pq){}//copy contructor  // copy constructor

    KeyType* minimum();                          // return the minimum element
    KeyType* extractMin();                       // delete the minimum element and return it
    void decreaseKey(int index, KeyType* key);  // decrease the value of an element
    void insert(KeyType* key);                   // insert a new element
    bool empty() const;                               // return whether the MPQ is empty
    int length() const;                          //return the number of keys
    std::string toString() const;               // return a string representation of the MPQ

    // Specify that MPQ will be referring to the following members of MinHeap<KeyType>.

    using MinHeap<KeyType>::A;
    using MinHeap<KeyType>::heapSize;
    using MinHeap<KeyType>::capacity;
    using MinHeap<KeyType>::parent;
    using MinHeap<KeyType>::swap;
    using MinHeap<KeyType>::heapify;

    /* The using statements are necessary to resolve ambiguity because
       these members do not refer to KeyType.  Alternatively, you could
       use this->heapify(0) or MinHeap<KeyType>::heapify(0).
    */
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinPriorityQueue<KeyType>& pq);

class FullError { };    // MinPriorityQueue full exception
class EmptyError { };   // MinPriorityQueue empty exception
class KeyError { };     // MinPriorityQueue key exception

#include "MPQ.cpp"

#endif
```

```
using namespace std;



/*
Pre-Condition: *this is not empty

Post-Condition: The element with the smallest key in array
A is returned
*/
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::minimum()
{
        if(heapSize<=0)
                throw EmptyError();
        else
        {
                return A[0];
        }
}

/*
Pre-Condition: *this is not empty

Post-Condition: The element with the smallest key in array
A is returned, and then removed from the queue.
*/
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::extractMin()
{
        if(heapSize<=0)
                throw EmptyError();

        else
        {
                KeyType* min = A[0];
                A[0]=A[heapSize-1];
                heapSize--;
                heapify(0);
                return min;
        }


}

/*
Pre-Condition:new key parameter is less than the key
value at the index,

Post-Condition: the key value is assigned to the key
parameter value
*/

template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey(int index, KeyType* key)
{
        if(heapSize<=0)
                throw EmptyError();
        else if(*key>*A[index]||index>heapSize)
                throw KeyError();

        else
        {
```

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


/*
Pre-Condition: MPQ is not at maximum capacity,

Post-Condition:new Key gets inserted into the MPQ
*/
template <class KeyType>
void MinPriorityQueue<KeyType>::insert(KeyType* key)
{
        if(heapSize==capacity)
                throw FullError();

        else
        {
                A[heapSize]=key;
                heapSize++;
                int index=heapSize-1;
                while((index>=0)&&(*(A[index])<*(A[parent(index)])))
                {
                        swap(index, parent(index));
                        index=parent(index);
                }
        }
}

/*
Pre-Condition:An MPQ instance calls empty method

Post-Condition: Method returns true if MPQ is empty, false otherwise
*/
template <class KeyType>
bool MinPriorityQueue<KeyType>::empty() const
{
        if(heapSize==0)
                return true;
        else
                return false;
}

/*
Pre-Condition: An MPQ instance calls length method

Post-Condition:Method returns the length of the MPQ
*/
template <class KeyType>
int MinPriorityQueue<KeyType>::length() const
{
        return heapSize;
}

/*
Pre-Condition:An MPQ instance calls the toString method

Post-Condition:The toString method generates a string
```

```
containing and formatting all of the elements in the
queue. The method then returns this string.
*/
template <class KeyType>
string MinPriorityQueue<KeyType>::toString() const
{
        string string_heap;
        KeyType temp;
        string_heap = "[";
        for(int i=0; i<heapSize; i++)
        {
                ostringstream convert;
                temp = *A[i];
                convert << temp;
                string_heap+=convert.str();
                if(i!=(heapSize-1))
                {
                        if(i!=(capacity-1))
                                string_heap+=", ";
                }
                if(i==(heapSize-1))
                {
                        string_heap+="|";
                }
        }
        string_heap+="]";
        return string_heap;
}

/*
Pre-Condition:Using standard iostream syntax "cout<<MPQ",
the overloaded output operator is utilized

Post-Condition:A string representation of the MPQ is outputted
to the screen.
*/
template <class KeyType>
ostream& operator<<(ostream& stream, const MinPriorityQueue<KeyType>& pq)
{
        stream<<pq.toString();
        return stream;
}
```

```
/*=====================================================================
 * huffman.cpp
 * Authors: Bobby Craig, Ansel Schiavone, Chris Castillo
 * Proj4
 * CS271 - Dr. Jessen Havill
 *
 * A program to compress or decompress text using the Huffman compression
 * technique.
 *=====================================================================*/

#include <iostream>
#include <string>
#include <string.h>
#include <fstream>
#include <stdlib.h>
#include <stdio.h>
#include "node.h"
#include "MPQ.h"
#include <math.h>
using namespace std;

/*=====================================================================
 * GLOBAL ARRAYS
 *=====================================================================*/

int FREQUENCY [256];//global array that tracks frequency of characters
string CODES [256];//global array that contains huffman codes of characters

/*=====================================================================
 * bitConvert
 * A function to determine whether a char is a 1 or 0
 * Preconditions: a character of value 1 or 0
 * Postconditions: an int value corresponding to the character value
 *=====================================================================*/

int bitConvert(char ch) {
        int binHold;
        if ( ch == '1' )
        {
                return 1;
        }
        else
                return 0;
}

/*=====================================================================
 * convert
 * A function to
 * Preconditions:"convert" method takes in an integer value as a parameter
 * Postconditions: method returns a string value that is the binary
   representation of the integer parameter
 *=====================================================================*/

string convert(int val)
{
           unsigned int mask = 1 << 8-1; //mask 1000000000000
           string bin_string="";
           for(int i=0; i<8; i++)
           {
                   if ((val & mask)==0)
                           bin_string+="0";
                   else
                           bin_string+="1";
```

```
                mask >>=1;

            }
            return bin_string;
}

/*=====================================================================
 * generate_code
 * A function to recursively steps down Huff Tree, generating Huffman codes
 * Preconditions:
 * Postconditions:
 *=====================================================================*/

void generate_code(string code, Node* node_ptr) {
        if(node_ptr->character == 0)
        {
                generate_code(code+"0", node_ptr->left);
                generate_code(code+"1", node_ptr->right);
        }
        if(node_ptr->character != 0) //if child, record code
                CODES[node_ptr->character]=code;
}

/*=====================================================================
 * compress
 * A function to compress a text file into a .huff file
 * Preconditions: Main Function calls "compress" method if user enters
   a text file to be compressed in the terminal and the name of a .huff
   file to be written to.

 * Postconditions: A compressed version of the text file passed in as a
   parameter will be written to the .huff file in binary. This file will
   be significantly smaller than the original text file.
 *=====================================================================*/

void compress(char* text_file, char* huff_file)
{

        /*=====================================================================
        * Read In Text File
        *=====================================================================*/
        char ch;
        ifstream in_file;
        in_file.open(text_file);
        //CHECKING FOR ERROR OPENING FILE
        if(!in_file)
        {
                cout<<"Error opening file"<<endl;
                exit(0);
        }
        stringstream buffer;
        buffer << in_file.rdbuf();
        string decompress(buffer.str());
        for ( int i = 0; i < decompress.length(); i++ ) {
                FREQUENCY[decompress[i]]++; //FIND CHARACTER FREQUENCY
        }
        in_file.close();
        //MPQ WILL HOLD NODES OF ALL CHARACTERS WITH A NON-ZERO FREQUENCY
        MinPriorityQueue<Node> MPQ1;
        int count = 0;
        //LOOPS THROUGH FREQUENCY ARRAY, CREATING NODES FOR PRESENT CHARACTERS
        for(int i=0; i<256; i++)
        {
```

```
                if(FREQUENCY[i]>0)
                {
                        Node *node = new Node(i);
                        node->frequency=FREQUENCY[i];
                        MPQ1.insert(node);
                        count++;

                }
        }

        /*=====================================================================
        * Create Huffman Tree for Huffman Prefix Values
        *=====================================================================*/

        //PERFORMS HUFFMAN ALGORITHM ON PRIORITY QUEUE OF PRESENT CHARACTERS
        for(int i=1; i<count; i++)
        {
                Node* sum_node = new Node(0);//CREATES PARENT NODE
                Node* x;
                Node* y;
                sum_node->left= x = MPQ1.extractMin();
                sum_node->right= y = MPQ1.extractMin();

                //PARENT NODE FREQUENCY IS THE SUM OF ITS CHILDREN
                sum_node->frequency=(x->frequency)+(y->frequency);
                MPQ1.insert(sum_node);
        }
        Node* Huff = MPQ1.extractMin();
        generate_code("", Huff);

        /*=====================================================================
        * Write Codes to out_string
        *=====================================================================*/

        // OPEN FILES
        ofstream out_file;
        out_file.open(huff_file);
        in_file.open(text_file);

        // CHECK FOR ERROR WHEN OPENING
        if(!in_file)
        {
                cout<<"Error opening file"<<endl;
                exit(0);
        }
        string out_string = "";

        // OUTPUT TRANSLATION TABLE
        for ( int i = 0; i < 256; i++ ) {
                if ( !CODES[i].empty() ) {
                        out_string += convert(CODES[i].length());
                        out_string += CODES[i];
                        out_string += convert(i);
                }
        }
        // ADDITION OF 8 ZEROS TO INDICATE MOVE TO HUFF VALUES; ENDS TRANSLATION
        out_string += "00000000";

        // OUTPUT FILE CONTENTS
        stringstream buffer2;
        buffer2 << in_file.rdbuf();
        string decompress2(buffer2.str());
        for ( int i = 0; i < decompress2.length(); i++ ) {
```

```
                out_string += CODES[decompress2[i]];    // WRITE CODES TO OUT_STRING
        }

        // ADD 0s TO COMPLETE THE 8 BIT PATTERN
        // MAKES STRING A MULTIPLE OF 0
        int add = 8-(out_string.length() % 8);
        if ( out_string.length() % 8 != 0 ) {
                for ( int i = 0; i<add; i++ ) {
                        out_string += "0";
                }
        }

        /*=====================================================================
        * Write Codes to out_file
        *=====================================================================*/

        char end_char = add;
        // ADD VALUE TO SUBTRACT FROM END
        out_file << add;
        // CONVETRT OUT_STRING TO BINARY, THEN CHAR, THEN PUT IN OUT_FILE
        for ( int i = 0; i < out_string.length(); i += 8 ) {
                int runningVal = 0;
                for ( int j = 0; j < 8; j++ ) {
                        // CONVERT TO INT VALUE FROM BINARY
                        runningVal += bitConvert(out_string[i+j]) * pow(2,7-j);
                }
                // CONVERT INT TO CHAR VALUE
                char toFile = runningVal;
                out_file << toFile;
        }

        in_file.close();
        out_file.close();

}

/*=====================================================================
 * decompress
 * A function to decompress a .huff file into a .txt file
 * Preconditions: "decompress" is called by Main when a user requests
   a .huff file to be decompressed via the terminal. The user will enter
   the source huff file and the destination text file, which are passed
   into "decompress" as parameters.
 * Postconditions: A decompressed text file will be successfully decoded
   from the compressed huff file and written into a text file.
 *=====================================================================*/

void decompress(char * huff_file, char * text_file)
{
        // OPEN FILES AND DECLARE VARIABLES
        char ch;
        ifstream in_file;
        ofstream out_file;
        in_file.open(huff_file);
        out_file.open(text_file);

        // CHECK FOR ERROR OPENING
        if(!in_file)
        {
                cout<<"Error opening file"<<endl;
                exit(0);
        }
```

```
	// READ FIRST CHAR FOR VALUE TO REMOVE FROM BACK OF BINARY STRING
	int remove_from_back;
	in_file >> remove_from_back;

	// MAKESHIFT WAY OF READING IN BINARY STRING FROM FILE
	stringstream buffer;
	buffer << in_file.rdbuf();
	string decompress(buffer.str());
	string binary = "";
	// CONVERT CHAR STRING TO BINARY STRING
	for ( int z=0; z < decompress.length(); z++ ) {
		binary += convert(decompress[z]);
	}
	// ERASE VALUES FROM BACK; NO NEED FOR STRING LENGTH TO BE MULTIPLE
	// OF 8 ANYMORE
	binary.erase(binary.end()-remove_from_back, binary.end());

	int index = 0;
	// ENCODE TRANSLATION TABLE
	while(1)
	{
		string code= "";
		int LEN = 0;
		int ascii =0;
		// FIND THE LENGTH OF THE HUFFMAN VALUE FOLLOWING
		for(int i=7; i>-1; i--)
		{
			ch = binary[index++];
			int binHold;
			if ( ch == '1' )
			{
				binHold = 1;
			}
			else
				binHold = 0;
			LEN = LEN + binHold * pow(2,i);
		}
		// CHECK IF TERMINATOR TELLS US TO START TRANSLATING
		if(LEN==0)
			break;
		// ADDS NUMBER OF CHARACTERS FROM "FOR LOOP" ABOVE
		for(int j=0; j<LEN; j++)
		{
			ch = binary[index++];
			code=code+ch;
		}
		// ADDS THE ASCII VALUE
		for(int i=7; i>-1; i--)
		{
			ch = binary[index++];
			int binHold;
			if ( ch == '1' ) {
				binHold = 1;
			}
			else {
				binHold = 0;
			}
			ascii = ascii + binHold * pow(2,i);
		}
		// ADD THE VALUE TO THE GLOBAL ARRAY AT THE BEGINNING
		CODES[ascii]=code;
	}
```

```
        string newChar = "";
        // CONVERT THE HUFFMAN VALUES USING THE TRANSLATION TABLE
        // CREATED ABOVE.
        for (int j=index; j<binary.length(); j++) {
                newChar = newChar + binary[j];
                // LOOK THROUGH THE TRANSLATION TABLE
                for ( int i = 0; i < 256; i++ ) {
                        if (!strcmp(newChar.c_str(),CODES[i].c_str()) && !CODES[i].empty()) {
                                // IF VALUE FOUND, CONVERT TO CHAR, EMPTY STRING
                                // FOR NEXT VALUE, AND ADD THE CHAR TO THE OUT_FILE
                                char pushChar = i;
                                out_file << "" << pushChar;
                                newChar = "";
                        }
                }
        }

        out_file.close();
        in_file.close();
}


int main(int argc, char *argv[])
{
        string argv1(argv[1]);
        char * file_name1 = argv[2];
        char * file_name2 = argv[3];
        // PRINT HELP INFO IF USER NEEDS GUIDANCE
        if ( argv1 == "help" ) {
                cout << "help: ./huffman [-cd] [input] [output]\n";
                cout << "   Compress or decompress your file using the respective option.\n";
                cout << "   Using the Huffman Compression Technique, this program will\n";
                cout << "   compress or decompress an input file and return an output file\n";
                cout << "\n   Options:\n";
                cout << "     -c\tcompress a .txt file into a .huff file\n";
                cout << "     -d\tdecompress a .huff file into a .txt file\n";
                cout << "\n   Arguments:\n";
                cout << "     INPUT     for compression, .txt file; for decompression, .huff f
ile\n";
                cout << "     OUTPUT    for compression, .huff file; for decompression, .txt f
ile\n\n";
        }
        // CHECK FOR 4 ARGUMENTS TO PROCEED
        if ( argc != 4 )
        {
                exit(0);
        }
        // DIRECTIONAL CONTROL -- "-d" TO DECOMPRESS
        if ( argv1 == "-d")
        {
                decompress(file_name1, file_name2);
        }
        // DIRECTIONAL CONTROL -- "-c" TO COMPRESS
        else if ( argv1=="-c" )
        {
                compress(file_name1, file_name2);
        }
        // EXIT IF VALID COMMAND NOT GIVEN
        else
        {
                cout << "Valid command (-c or -d) not given." << endl;
                exit(0);
```

```
        }
}
```

```
/*#########################################################################
unit_test.cpp
Carol Vitellas & Ansel Schiavone
Project 5
CS 271: Data Structures
March 29th, 2016
#########################################################################*/

#include "hash_table.h"
#include "dict.h"
#include <iostream>
#include "test.h"
#include <string>
#include <assert.h>
#include "movie.h"
using namespace std;

/*#########################################################################
Insert Test: tests abilitiy to insert Test elements in HashTable
/*#########################################################################*/
void insert_test()
{
        Test T1(1), T2(2), T3(3), T4(4), T5(5), T6(6), T7(7), T8(8);
        HashTable<Test> H1(3);
        H1.insert(T1);
        H1.insert(T2);
        H1.insert(T3);
        H1.insert(T4);
        H1.insert(T5);
        H1.insert(T6);
        H1.insert(T7);
        H1.insert(T8);

        assert(H1.toString(0)=="3 6 " && "Insert");
        assert(H1.toString(1)=="1 4 7 " && "Insert");
        assert(H1.toString(2)=="2 5 8 " && "Insert");
}

/*#########################################################################
Get Test: tests abilitiy to getTest elements out of HashTable after they have
been inserted
/*#########################################################################*/
void get_test()
{       Test T1(1), T2(2), T3(3), T4(4), T5(1), T6(2), T7(3), T8(4);


        HashTable<Test> H1(3);
        H1.insert(T1);
        H1.insert(T2);
        H1.insert(T3);
        H1.insert(T4);

        assert(T1==H1.get(T5));
        assert(T2==H1.get(T6));
        assert(T3==H1.get(T7));
        assert(T4==H1.get(T8));

}
/*#########################################################################
Insert Test: tests abilitiy to remove Test elements from HashTable after they
have been inserted.
/*#########################################################################*/
void remove_test()
```

```
{       Test T1(1), T2(2), T3(3), T4(4), T5(5), T6(6), T7(7), T8(8);
        HashTable<Test> H1(3);
        H1.insert(T1);
        H1.insert(T2);
        H1.insert(T3);
        H1.insert(T4);
        H1.insert(T5);
        H1.insert(T6);
        H1.insert(T7);
        H1.insert(T8);

        assert(H1.toString(0)=="3 6 " && "Remove");
        assert(H1.toString(1)=="1 4 7 " && "Remove");
        assert(H1.toString(2)=="2 5 8 " && "Remove");

        H1.remove(T1);
        H1.remove(T2);
        H1.remove(T3);

        assert(H1.toString(0)=="6 ");
        assert(H1.toString(1)=="4 7 ");//getting segfault here
        assert(H1.toString(2)=="5 8 ");



}
/*#########################################################################
Dictionary Test: tests Dictionary (ihereted from HashTable) and its function-
-ality in storing and managing Test values
/*#########################################################################*/
void test_dict()
{
        Test T1(1), T2(2), T3(3), T4(4), T5(5), T6(6), T7(7), T8(8);
        Dictionary<Test> D1;
        assert(D1.empty()==true);
        D1.insert(T1);
        assert(D1.empty()==false);
        D1.remove(T1);
        assert(D1.empty()==true);

}

/*#########################################################################
Novie Test: tests Dictionary's functionality with Movie values
/*#########################################################################*/
void movie_test()
{
        Dictionary<Movie> D1;
        Movie M1, M2, M3;

        M1.key="Bruno";
        M2.key="Borat";
        M3.key="Pineapple Express";

        M1.cast_list.append("Ansel");
        M1.cast_list.append("Carol");

        M2.cast_list.append("Bill");
        M2.cast_list.append("Sean");

        D1.insert(M1);
        D1.insert(M2);
```

```
        assert(D1.get(M1)==M1);
        assert(D1.get(M2)==M2);

        assert(D1.empty()==false);

        D1.remove(M1);
        D1.remove(M2);

        assert(D1.empty()==true);
}

int main()
{
        insert_test();
        get_test();
        remove_test();
        test_dict();
        movie_test();
        return 0;
}
```

```
/*##############################################################################
hash.h
Carol Vitellas & Ansel Schiavone
Project 5
CS 271: Data Structures
March 29th, 2016
##############################################################################*/

#include "list.h"
#ifndef HASH_TABLE_H
#define HASH_TABLE_H

template <class KeyType>
class HashTable
{
public:
        HashTable(int numSlots);//constructor
        ~HashTable();//destructor

        KeyType get( KeyType& k);//returns KeyType that contains desired key
        void insert(KeyType k); //inserts k into hash table
        void remove( KeyType& k); //removes object that contains k from hash table

        std::string toString(int slot); //returns a string representation of a single hashtabl
e slot

protected:
        int num_slots;
        List<KeyType> *table;  // an array of List<KeyType>'s
};

class GetError { }; //get exception
class InsertError { }; //insert exception

#include "hash_table.cpp"

#endif
```

```
/*##############################################################################
movie.h
Carol Vitellas & Ansel Schiavone
Project 5
CS 271: Data Structures
March 29th, 2016
##############################################################################*/

#include "hash_table.h"
#include "list.h"
#include <iostream>
#include <cstdlib>

using namespace std;
#ifndef MOVIE_H
#define MOVIE_H

class Movie
{
public:
        string key;
        List<string> cast_list;
/*##############################################################################
CONSTRUCTOR
precondition: -
postcondition: a movie instance is created, and key and cast list are set to
empty.
##############################################################################*/
        Movie()
        {
                key = "";
                cast_list.insert(0,"");
        }
/*##############################################################################
HASH
precondition: *this is a Movie
postcondition: returns a mapping value based on the movie title (key)
##############################################################################*/
        int hash(int num_slots)
        {       int mapping =0;
                int int_key=0;
                int ascii=0;
                for(int i=0; i<key.length();i++)
                {
                        int ascii = key[i];
                        int_key+=(ascii*i);
                }
                mapping=(int_key % num_slots);
                return mapping;
        }
/*##############################################################################
INEQUALITY
precondition: *this and m are Movies
postcondition: returns boolean value of true if movies have different titles,
false if they are the same.
##############################################################################*/
        bool operator!=(const Movie& m) const
         {
                 bool un_equal = true;
                 if(this->key == m.key)
                 {
                         un_equal = false;
                 }
```

```
                return un_equal;
        }
/*#########################################################################
EQUALITY
precondition: *this and m are Movies
postcondition: returns boolean value of true if movies have the same title,
false if different.
/*#########################################################################*/
        bool operator==(const Movie& m) const
        {
                bool equal = true;
                if(this->key != m.key)
                {

                        equal = false;
                }
                return equal;
        }
/*#########################################################################
STREAM
precondition: m is a Movie and os is an ostream
postcondition: an ostream value is returned that contains the movie in the
format: (Title, [cast_member1, cast_member2,...])
/*#########################################################################*/
        friend ostream& operator<<(ostream& os, Movie& m)
        {
                os << "(" <<m.key << ", [";
                for(int i=0; i<m.cast_list.length();i++)
                {
                        if((i+1)==m.cast_list.length())
                                os<<m.cast_list[i]<<"])"<<endl;
                        else if(i==0)
                                os<<m.cast_list[i];
                        else
                                os<<", "<<m.cast_list[i];
                }
                return os;
        }

};

#endif
```

```
/*##############################################################################
dict.h
Carol Vitellas & Ansel Schiavone
Project 5
CS 271: Data Structures
March 29th, 2016
##############################################################################*/

#include "hash_table.h"
#ifndef DICT_H
#define DICT_H

template <class KeyType>
class Dictionary: public HashTable<KeyType>
{
public:
        /*Constructor - Utilizes HashTable Constructor, automatically
        setting number of slots to 100*/
        Dictionary() : HashTable<KeyType>::HashTable(100){};

        bool empty();//added method not included in base HashTable class

        /*Inherets the following elements from HashTable*/
        using HashTable<KeyType>::num_slots;
        using HashTable<KeyType>::get;
        using HashTable<KeyType>::insert;
        using HashTable<KeyType>::remove;
        using HashTable<KeyType>::toString;


};
#include "dict.cpp"
#endif
```

```
/*#############################################################################
BST.h
Carol Vitellas & Ansel Schiavone
Project 6
CS 271: Data Structures
April 5th, 2016
#############################################################################*/
#include <string>
#include <iostream>
#include <sstream>
using namespace std;
#ifndef BST_H
#define BST_H

template <class KeyType>
class Tree_Node
{
 public:

   KeyType *item;
   Tree_Node<KeyType> *right;
   Tree_Node<KeyType> *left;
   Tree_Node<KeyType> *parent;

   Tree_Node()
   {
        item = NULL;
        right = NULL;
        left = NULL;
        parent = NULL;
   }

   Tree_Node(KeyType* initItem)
   {
        item = initItem;
        right = NULL;
        left = NULL;
        parent = NULL;
   }
};

template <class KeyType>
class BST
{
public:
        BST();//constructor
        ~BST();//destructor
        BST(const BST<KeyType>& copy_tree);//copy constructor

        bool empty();                           // return true if empty; false o/w
        KeyType *get(const KeyType& k);         // return first element with key equal to k
        void insert(KeyType *k);                // insert k into the tree
        void remove(const KeyType& k);          // delete first element with key equal to k
        KeyType *maximum();                     // return the maximum element
        KeyType *minimum();                     // return the minimum element
        KeyType *successor(const KeyType& k);   // return the successor of k
        KeyType *predecessor(const KeyType& k); // return the predecessor of k
        std::string inOrder();          // return string of elements from an inorder traversal
        std::string preOrder();         // return string of elements from a preorder traversal
        std::string postOrder();        // return string of elements from a postorder traversa
l

        BST<KeyType> operator=(const BST<KeyType>& Tree);
```

```
protected:
        Tree_Node<KeyType>* root;
        Tree_Node<KeyType>* get_helper(Tree_Node<KeyType>* search_root, const KeyType &k);//ge
t helper function
        Tree_Node<KeyType>* minimum_helper(Tree_Node<KeyType>* x);//minimum helper function
        Tree_Node<KeyType>* maximum_helper(Tree_Node<KeyType>* x);//maximum helper function
        void inOrder_helper(Tree_Node<KeyType>* x, stringstream &order);//inorder helper funct
ion
        void preOrder_helper(Tree_Node<KeyType>* x, stringstream &order);//preorder helper fun
ction
        void postOrder_helper(Tree_Node<KeyType>* x, stringstream &order);//postorder helper f
unction
        void transplant(Tree_Node<KeyType>* victim, Tree_Node<KeyType>* replacement);//helper
function for remove
        void copy(Tree_Node<KeyType>* copy_root);
        void destroy(Tree_Node<KeyType>* x);
};

template <class KeyType>
std::ostream& operator<<(std::ostream& os, BST<KeyType>& Tree);

class GetError { }; //get exception
class MaxMinError { }; //minimum maximum exception

#include "BST.cpp"

#endif
```

```
/*##########################################################################
BST.cpp
Carol Vitellas & Ansel Schiavone
Project 5
CS 271: Data Structures
April 6th, 2016
##########################################################################*/

using namespace std;
#include <string>
#include <iostream>
#include <sstream>

/*##########################################################################
CONSTRUCTOR
precondition:
postcondition: a BST is declared, and its root is set to NULL.
/*##########################################################################*/
template <class KeyType>
BST<KeyType>::BST()//constructor
{
        this->root=NULL;
}


/*##########################################################################
Destructor
precondition: *this is a BST
postcondition: all nodes of the BST are deleted from memory
/*##########################################################################*/
template <class KeyType>
BST<KeyType>::~BST()//destructor
{
        destroy(root);

}

/*##########################################################################
Copy Constructor
precondition:
postcondition: a BST is declared, and is made a copy of Copy_Tree
/*##########################################################################*/
template <class KeyType>
BST<KeyType>::BST(const BST<KeyType>& copy_tree)//copy constructor
{
        root=NULL;
        copy(copy_tree.root);
}

/*##########################################################################
Empty
precondition: *this is a BST
postcondition: returns true if tree is empty, false if otherwise
/*##########################################################################*/
template <class KeyType>
bool BST<KeyType>::empty()
{
        if(this->root == NULL)
        {
                return true;
        }
```

```
        else
                return false;

}

/*#########################################################################
Get
precondition: *this is a BST
postcondition: returns pointer to KeyType k
/*#########################################################################*/
template <class KeyType>
KeyType* BST<KeyType>::get(const KeyType& k)
{
        return (get_helper(root,k)->item);
}

/*#########################################################################
Get-Helper
precondition: *this is a Bst
postcondition: returns pointer to node containing KeyType k
/*#########################################################################*/

template <class KeyType>
Tree_Node<KeyType>* BST<KeyType>::get_helper(Tree_Node<KeyType>* search_root, const KeyType &k
)
{
        if(search_root == NULL)
                throw GetError();
        if(k==*(search_root->item))//dereference search_root?
        {
                return search_root;
        }
        if(k<*(search_root->item))//comparing keyTypes
        {
                //cout<<"going left"<<endl;
                return get_helper(search_root->left, k);
        }
        else
        {
                //cout<<"going right"<<endl;
                return get_helper(search_root->right, k);
        }

}

/*#########################################################################
Insert
precondition: *this is a BST
postcondition: inserts KeyType pointer that points to k into tree
/*#########################################################################*/
template <class KeyType>
void BST<KeyType>::insert(KeyType* k)
{
        Tree_Node<KeyType>* z = new Tree_Node<KeyType>;
        z->item = k;
        Tree_Node<KeyType>* x = this->root;
        Tree_Node<KeyType>* y = NULL;
        while(x != NULL)
        {
                y=x;
                if(*z->item<*x->item)//dereference x to get item?
                        x=x->left;
                else
```

```
                        x=x->right;
        }
        z->parent = y;
        if(y == NULL)
                this->root = z;//assign address of z to root
        else if(*z->item < *y->item)//dereference y?
                y->left = z;//assign address of z to y.left? Dereference y?
        else
                y->right = z;//assign address of z to y.left? Dereference y?
}

/*#########################################################################
Remove
precondition: *this is a BST containing a node that contains a pointer to k
postcondition: node containing k* is removed from tree
/*#########################################################################*/
template <class KeyType>
void BST<KeyType>::remove(const KeyType& k)
{
        Tree_Node<KeyType>* y = NULL;
        Tree_Node<KeyType>* victimPtr=get_helper(this->root, k);
        if(victimPtr->left==NULL){
                transplant(victimPtr, victimPtr->right);
        }
        else if (victimPtr->right==NULL){
                transplant(victimPtr, victimPtr->left);
        }
        else
        {
                y = minimum_helper(victimPtr->right);
                if (y->parent == victimPtr)
                {
                        transplant(y,y->right);
                        y->right = victimPtr->right;
                        y->right->parent = y;
                }
                transplant(victimPtr,y);
                y->left = victimPtr->left;
                y->left->parent = y;
        }
        delete victimPtr;
}

/*#########################################################################
Transplant
precondition: transplant called by Remove method
postcondition: Subtree "victim" is replaced by "replacement"
/*#########################################################################*/
template <class KeyType>
void BST<KeyType>::transplant(Tree_Node<KeyType>* victim, Tree_Node<KeyType>* replacement)
{
        if(victim->parent == NULL)
                this->root=replacement;
        else if(victim == victim->parent->left)
                victim->parent->left=replacement;
        else
                victim->parent->right = replacement;
        if(replacement != NULL)
                replacement->parent= victim->parent;
}

/*#########################################################################
Maximum-helper
```

```
precondition: *this is a BST
postcondition: A pointer to the node containing pointer to largest KeyValue
in tree is returned.
/*#########################################################################*/
template <class KeyType>
Tree_Node<KeyType>* BST<KeyType>::maximum_helper(Tree_Node<KeyType>* x)
{
        if(x==NULL)
                throw MaxMinError();
        while(x->right != NULL)
                x=x->right;
        return x;
}

/*#########################################################################
Maximum
precondition: *this is a BST
postcondition: A pointer to largest KeyValue in tree is returned.
/*#########################################################################*/
template <class KeyType>
KeyType* BST<KeyType>::maximum()
{
        Tree_Node<KeyType>* min = maximum_helper(root);
        return min->item;
}

/*#########################################################################
Minimum-helper
precondition: *this is a BST
postcondition: A pointer to the node containing pointer to smallest KeyValue
in tree is returned.
/*#########################################################################*/
template <class KeyType>
Tree_Node<KeyType>* BST<KeyType>::minimum_helper(Tree_Node<KeyType>* x)
{
        if(x==NULL)
                throw MaxMinError();
        while(x->left != NULL)
                x=x->left;
        return x;
}

/*#########################################################################
Minimum
precondition: *this is a BST
postcondition: A pointer to smallest KeyValue in tree is returned.
/*#########################################################################*/
template <class KeyType>
KeyType* BST<KeyType>::minimum()
{
        Tree_Node<KeyType>* min = minimum_helper(root);
        return min->item;
}

/*#########################################################################
Successor
precondition: *this is a BST
postcondition: A KeyValue* of the successor of the node containinga k* is
returned.
/*#########################################################################*/
template <class KeyType>
KeyType* BST<KeyType>::successor(const KeyType& k)
{
```

```
        Tree_Node<KeyType>* nodePtr=get_helper(this->root, k);
        Tree_Node<KeyType>* successor = NULL;
        if(nodePtr->right != NULL)
        {
                successor = minimum_helper(nodePtr->right);
                return successor->item;
        }

        successor=nodePtr->parent;
        while(successor!=NULL && successor->right==nodePtr)
        {
                nodePtr = successor;
                successor = successor->parent;
        }
        return successor-> item;
}

/*#########################################################################
Predecessor
precondition: *this is a BST
postcondition: A KeyValue* of the predecessor of the node containinga k* is
returned.
/*#########################################################################*/
template <class KeyType>
KeyType* BST<KeyType>::predecessor(const KeyType& k)
{

        Tree_Node<KeyType>* nodePtr=get_helper(this->root, k);
        Tree_Node<KeyType>* successor = NULL;
        if(nodePtr->right != NULL)
        {
                successor = maximum_helper(nodePtr->left);
                return successor->item;
        }

        successor=nodePtr->parent;
        while(successor!=NULL && successor->left==nodePtr)
        {
                nodePtr = successor;
                successor = successor->parent;
        }
        return successor-> item;
}


/*#########################################################################
inOrder
precondition: *this is a BST
postcondition: a string representation of the tree "in order" is returned
/*#########################################################################*/
template <class KeyType>
string BST<KeyType>::inOrder()
{
        stringstream order;
        inOrder_helper(this->root, order);
        return order.str();
}

/*#########################################################################
inOrder-Helper
precondition: method called by inOrder
postcondition: stringstream value passed by reference contains next KeyValue
```

```
in tree in order
/*#########################################################################*/
template <class KeyType>
void BST<KeyType>::inOrder_helper(Tree_Node<KeyType>* x, stringstream &order)
{
        if(x!=NULL)
        {
                inOrder_helper(x->left,order);
                order<<*x->item<<" ";
                inOrder_helper(x->right,order);

        }
}

/*#########################################################################
preOrder
precondition: *this is a BST
postcondition: a string representation of the tree in "pre order" is returned
/*#########################################################################*/
template <class KeyType>
string BST<KeyType>::preOrder()
{
        stringstream order;
        preOrder_helper(this->root, order);
        return order.str();
}

/*#########################################################################
preOrder-Helper
precondition: method called by preOrder
postcondition: stringstream value passed by reference contains next KeyValue
in tree in preOrder
/*#########################################################################*/
template <class KeyType>
void BST<KeyType>::preOrder_helper(Tree_Node<KeyType>* x,  stringstream &order)
{
        if(x!=NULL)
        {
                order<<*x->item<<" ";
                preOrder_helper(x->left,order);
                preOrder_helper(x->right,order);
        }
}

/*#########################################################################
postOrder
precondition: *this is a BST
postcondition: a string representation of the tree in "post order" is returned
/*#########################################################################*/
template <class KeyType>
string BST<KeyType>::postOrder()
{
        stringstream order;
        postOrder_helper(this->root, order);
        return order.str();

}
/*#########################################################################
postOrder-Helper
precondition: method called by postOrder
postcondition: stringstream value passed by reference contains next KeyValue
in tree in postOrder
/*#########################################################################*/
```

```
template <class KeyType>
void BST<KeyType>::postOrder_helper(Tree_Node<KeyType>* x,  stringstream &order)
{
        if(x!=NULL)
        {
                postOrder_helper(x->left,order);
                postOrder_helper(x->right,order);
                order<<*x->item<<" ";

        }
}
/*##############################################################################
Destroy
precondition: *this is a BST, and copy has been called by copy constructor or
assignement operator
postcondition: *this BST becomes a copy of parameter (sub)tree
##############################################################################*/
template <class KeyType>
void BST<KeyType>::copy(Tree_Node<KeyType>* x)
{
        if(x != NULL)
        {
                //copy in preOrder
                this->insert(x->item);
                copy(x->left);
                copy(x->right);

        }
}

/*##############################################################################
Destroy
precondition: *this is a BST, and destroy has been called by destructor
postcondition: all nodes of the BST are deleted from memory
##############################################################################*/
template <class KeyType>
void BST<KeyType>::destroy(Tree_Node<KeyType>* x)
{
        if(x != NULL)
        {
                destroy(x->left);
                remove(*x->item);
                destroy(x->right);
        }
}
/*##############################################################################
Assignment
precondition: *this is a BST and parameter is a BST
postcondition: all nodes of the this BST are deleted from memory and this BST
becomes a copy of the parameter tree.
##############################################################################*/
template <class KeyType>
BST<KeyType> BST<KeyType>::operator=(const BST<KeyType>& Tree)
{
        if(root != Tree.root)
        {
                destroy(root);
                copy(Tree.root);
                return *this;
        }
}

/*##############################################################################
```

```
Stream
precondition: *this is an ostream object
postcondition: stream representation of tree in order is returned to object
/*#########################################################################*/
template <class KeyType>
std::ostream& operator<<(std::ostream& os, BST<KeyType>& Tree)
{
        os << Tree.inOrder();
        return os;
}
```

```
/*##############################################################################
dict.h
Carol Vitellas & Ansel Schiavone
Project 6
CS 271: Data Structures
March 6th, 2016
##############################################################################*/

#include "BST.h"
#ifndef DICT_H
#define DICT_H

template <class KeyType>
class Dictionary: public BST<KeyType>
{
public:
        /*Constructor - Utilizes BST Constructor*/
        Dictionary() : BST<KeyType>::BST(){};

        /*Inherets the following elements from BST*/
        using BST<KeyType>::empty;
        using BST<KeyType>::get;
        using BST<KeyType>::insert;
        using BST<KeyType>::remove;
        using BST<KeyType>::inOrder;


};

template <class KeyType>
std::ostream& operator<<(std::ostream& os, Dictionary<KeyType>& Dict)//overloaded stream opera
tor
{
        os<<Dict.inOrder();
        return os;
}
#endif
```

```
/*##############################################################################
test.cpp
Carol Vitellas & Ansel Schiavone
Project 6
CS 271: Data Structures
April 6th, 2016
##############################################################################*/

#include "BST.h"
#include <assert.h>
using namespace std;


void test_insert()
{
        BST<int> B1;
        int *a=new int;
        int *b=new int;
        int *c=new int;
        int *d=new int;
        int *e=new int;
        int *f=new int;
        int *g=new int;
        *a=4;
        *b=5;
        *c=3;
        *d=1;
        *e=2;
        *f=6;
        *g=7;
        B1.insert(a);
        B1.insert(b);
        B1.insert(c);
        B1.insert(d);
        B1.insert(e);
        B1.insert(f);
        B1.insert(g);
        assert(B1.inOrder()=="1 2 3 4 5 6 7 ");
        assert(B1.preOrder()=="4 3 1 2 5 6 7 ");
        assert(B1.postOrder()=="2 1 3 7 6 5 4 ");
}

void test_get()
{
        BST<int> B1;
        int *a=new int;
        int *b=new int;
        int *c=new int;
        int *d=new int;
        int *e=new int;
        int *f=new int;
        int *g=new int;
        *a=4;
        *b=5;
        *c=3;
        *d=1;
        *e=2;
        *f=6;
        *g=7;
        B1.insert(a);
        B1.insert(b);
        B1.insert(c);
        B1.insert(d);
```

```
        B1.insert(e);
        B1.insert(f);
        B1.insert(g);

        assert(B1.inOrder()=="1 2 3 4 5 6 7 ");
        assert(B1.preOrder()=="4 3 1 2 5 6 7 ");
        assert(B1.postOrder()=="2 1 3 7 6 5 4 ");
        assert(*B1.get(1)==1);
        assert(*B1.get(2)==2);
        assert(*B1.get(3)==3);
        assert(*B1.get(4)==4);
        assert(*B1.get(5)==5);
        assert(*B1.get(6)==6);
        assert(*B1.get(7)==7);

}

void test_minimum()
{
        BST<int> B1;
        int *a=new int;
        int *b=new int;
        int *c=new int;
        int *d=new int;
        int *e=new int;
        int *f=new int;
        int *g=new int;
        *a=4;
        *b=5;
        *c=3;
        *d=1;
        *e=2;
        *f=6;
        *g=7;
        B1.insert(a);
        B1.insert(b);
        B1.insert(c);
        B1.insert(d);
        B1.insert(e);
        B1.insert(f);
        B1.insert(g);
        assert(*B1.minimum()==1);
}
void test_maximum()
{
        BST<int> B1;
        int *a=new int;
        int *b=new int;
        int *c=new int;
        int *d=new int;
        int *e=new int;
        int *f=new int;
        int *g=new int;
        *a=4;
        *b=5;
        *c=3;
        *d=1;
        *e=2;
        *f=6;
        *g=7;
        B1.insert(a);
        B1.insert(b);
        B1.insert(c);
```

```
        B1.insert(d);
        B1.insert(e);
        B1.insert(f);
        B1.insert(g);
        assert(*B1.maximum()==7);

}
void test_remove()
{
        BST<int> B1;
        int *a=new int;
        int *b=new int;
        int *c=new int;
        int *d=new int;
        int *e=new int;
        int *f=new int;
        int *g=new int;
        *a=4;
        *b=5;
        *c=3;
        *d=1;
        *e=2;
        *f=6;
        *g=7;
        B1.insert(a);
        B1.insert(b);
        B1.insert(c);
        B1.insert(d);
        B1.insert(e);
        B1.insert(f);
        B1.insert(g);
        assert(B1.inOrder()=="1 2 3 4 5 6 7 ");
        assert(B1.preOrder()=="4 3 1 2 5 6 7 ");
        assert(B1.postOrder()=="2 1 3 7 6 5 4 ");

        B1.remove(4);
        assert(B1.inOrder()=="1 2 3 5 6 7 ");
        assert(B1.preOrder()=="5 3 1 2 6 7 ");
        assert(B1.postOrder()=="2 1 3 7 6 5 ");

        B1.remove(1);

        assert(B1.inOrder()=="2 3 5 6 7 ");
        assert(B1.preOrder()=="5 3 2 6 7 ");
        assert(B1.postOrder()=="2 3 7 6 5 ");

        B1.remove(2);
        B1.remove(3);
        B1.remove(5);
        B1.remove(6);

        assert(B1.inOrder()=="7 ");
        assert(B1.preOrder()=="7 ");
        assert(B1.postOrder()=="7 ");



}
void test_successor()
{
        BST<int> B1;
        int *a=new int;
        int *b=new int;
```

```
        int *c=new int;
        int *d=new int;
        int *e=new int;
        int *f=new int;
        int *g=new int;
        *a=4;
        *b=5;
        *c=3;
        *d=1;
        *e=2;
        *f=6;
        *g=7;
        B1.insert(a);
        B1.insert(b);
        B1.insert(c);
        B1.insert(d);
        B1.insert(e);
        B1.insert(f);
        B1.insert(g);
        assert(B1.inOrder()=="1 2 3 4 5 6 7 ");
        assert(B1.preOrder()=="4 3 1 2 5 6 7 ");
        assert(B1.postOrder()=="2 1 3 7 6 5 4 ");
        assert(*B1.successor(2)==3);
        assert(*B1.successor(1)==2);
}
void test_predecessor()
{
        BST<int> B1;
        int *a=new int;
        int *b=new int;
        int *c=new int;
        int *d=new int;
        int *e=new int;
        int *f=new int;
        int *g=new int;
        *a=4;
        *b=5;
        *c=3;
        *d=1;
        *e=2;
        *f=6;
        *g=7;
        B1.insert(a);
        B1.insert(b);
        B1.insert(c);
        B1.insert(d);
        B1.insert(e);
        B1.insert(f);
        B1.insert(g);
        assert(B1.inOrder()=="1 2 3 4 5 6 7 ");
        assert(B1.preOrder()=="4 3 1 2 5 6 7 ");
        assert(B1.postOrder()=="2 1 3 7 6 5 4 ");
        assert(*B1.predecessor(7)==6);
        assert(*B1.predecessor(4)==3);
}
void test_copy()
{
        BST<int> B1;
        int *a=new int;
        int *b=new int;
        int *c=new int;
        int *d=new int;
        int *e=new int;
```

```
        int *f=new int;
        int *g=new int;
        *a=4;
        *b=5;
        *c=3;
        *d=1;
        *e=2;
        *f=6;
        *g=7;
        B1.insert(a);
        B1.insert(b);
        B1.insert(c);
        B1.insert(d);
        B1.insert(e);
        B1.insert(f);
        B1.insert(g);
        assert(B1.inOrder()=="1 2 3 4 5 6 7 ");
        assert(B1.preOrder()=="4 3 1 2 5 6 7 ");
        assert(B1.postOrder()=="2 1 3 7 6 5 4 ");
        BST<int> B2(B1);
        assert(B2.inOrder()=="1 2 3 4 5 6 7 ");


}

void test_assignment()
{
        BST<int> B1;
        int *a=new int;
        int *b=new int;
        int *c=new int;
        int *d=new int;
        int *e=new int;
        int *f=new int;
        int *g=new int;
        *a=4;
        *b=5;
        *c=3;
        *d=1;
        *e=2;
        *f=6;
        *g=7;
        B1.insert(a);
        B1.insert(b);
        B1.insert(c);
        B1.insert(d);
        B1.insert(e);
        B1.insert(f);
        B1.insert(g);
        assert(B1.inOrder()=="1 2 3 4 5 6 7 ");
        assert(B1.preOrder()=="4 3 1 2 5 6 7 ");
        assert(B1.postOrder()=="2 1 3 7 6 5 4 ");
        BST<int> B2;
        B2=B1;

        assert(B2.inOrder()=="1 2 3 4 5 6 7 ");
        assert(B2.preOrder()=="4 3 1 2 5 6 7 ");
        assert(B2.postOrder()=="2 1 3 7 6 5 4 ");
}

int main()
{
        test_insert();
```

```
        test_get();
        test_minimum();
        test_maximum();
        test_remove();
        test_successor();
        test_predecessor();
        test_copy();
        test_assignment();

}
```

```
/*##############################################################################
movie_query.cpp
Carol Vitellas & Ansel Schiavone
Project 6
CS 271: Data Structures
April 6th, 2016
##############################################################################*/


using namespace std;
#include "list.h"
#include "movie.h"
#include "dict.h"
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <sys/time.h>


/*##############################################################################
SETUP
precondition: -
postcondition: a dictionary of all the movies stored in the IMDB file is returned
to main.
##############################################################################*/
Dictionary<Movie> setup()
{       ifstream in_file;
        string line;
        string tempString="";
        in_file.open("movie.txt");
        Dictionary<Movie> movieDict;
        /*Unpacks file line-by-line*/
        if(in_file)
        {
                List<string> movieList;
                while(getline(in_file, line))
                {
                        for(int i=0; i<line.length(); i++)
                        {
                                char x =line[i];

                                if(x==9) //Deliminates by "tab" character (9)
                                {
                                        movieList.append(tempString);
                                        tempString="";
                                }
                                else
                                {
                                        tempString+=x;
                                }
                        }
                        movieList.append(tempString);
                        tempString="";
                        //Movie list now contains all of a movie's information
                        Movie* tempMovie = new Movie;
                        tempMovie->key=movieList.pop(0);

                        int len = movieList.length();
                        for(int i=0; i<len; i++)
                        {       string castMember=movieList.pop(0);
                                tempMovie->cast_list.insert(0,castMember);
                        }
                        movieDict.insert(tempMovie);
```

```
                }
        }
        else
        {
                cout<<"Error opening file"<<endl;
                exit(0);
        }
        return movieDict;
}

/*##############################################################################
QUERY
precondition: dictionary of movies passed to function
postcondition: outputs title and cast list of particular movie entered by user.
##############################################################################*/
void query(Dictionary<Movie>& movieDict)
{       string movieString;
        Movie pseudoMovie;
        Movie realMovie;
        bool unfound = false;
        cout<<"Please enter a movie title: ";
        getline(cin, movieString);
        pseudoMovie.key=movieString;
        try{
                realMovie=*movieDict.get(pseudoMovie);
        }
        /*Catches error that occurs when searching
        for a movie that does not exist*/
        catch(const GetError& e)
        {
                cout<<"Movie title not found."<<endl;
                unfound = true;
        }
        if(!unfound)
                cout<<endl<<realMovie<<endl;
}

int main()

{
        timeval timeBefore, timeAfter;
        long diffSeconds, diffUSeconds;

        gettimeofday(&timeBefore, NULL);
        Dictionary<Movie> movieDict=setup();
        gettimeofday(&timeAfter, NULL);
        diffSeconds = timeAfter.tv_sec - timeBefore.tv_sec;
        diffUSeconds = timeAfter.tv_usec - timeBefore.tv_usec;
        cout<<diffSeconds + diffUSeconds/1000000.0<<endl;


        while(1)
        {       query(movieDict);
        }
}
```

```
/*#########################################################################
Graph.h
Megan Kenworthy & Ansel Schiavone
Project 7
CS 271: Data Structures
April 8th, 2016
#########################################################################*/
#include "list.h"
#include "pq.h"
#include "DisjointSets.h"
#include <string>

#ifndef GRAPH_H
#define GRAPH_H

/*#########################################################################
VERTEX

Vertex class is a essentially a node that contains a unique identifier beween
0 and n-1, as well as a color and various time variables that are used for the
DFS and Kruskal algorithms.
/*#########################################################################*/
template <class T>
class Vertex
{
public:
        Vertex();
        Vertex(int ids)
        {
                id = ids;       //sets id to input id
        }
        int id;
        string color;   //to be used in dfs
        Vertex* predv;  //used in dfs for pred vertex
        int discovery;
        int finish;
        friend std::ostream& operator<<(std::ostream& stream, Vertex<T>& v)//overloaded <<
        {
                stream<<"("<<v.id<<")";
                return stream;
        }
};

/*#########################################################################
EDGE

Edge class represents an edge (u,v) in the graph. Edge has pointers to vertices
v and u, as well as an integer varaible "weight" that represents the weight of
the edge between u and v. Edge also overloads all comparison operators, allowing
it to be used in a minimum priority queue.
/*#########################################################################*/
template <class T>
class Edge
{
public:
        Edge(int _weight, Vertex<T>* _u, Vertex<T>* _v)
        {
                weight = _weight;
                u = _u;
                v = _v;

        }
        int weight;
```

```
        Vertex<T>* u;
        Vertex<T>* v;

        bool operator==(const Edge<T>& e)
        {
                if(weight == e.weight)
                        return true;
                return false;
        }

        bool operator!=(const Edge<T>& e)
        {
                if(weight != e.weight)
                        return true;
                return false;
        }

        bool operator<(const Edge<T>& e)
        {
                if(weight < e.weight)
                        return true;
                return false;
        }

        bool operator>(const Edge<T>& e)
        {
                if(weight > e.weight)
                        return true;
                return false;
        }


        bool operator<=(const Edge<T>& e)
        {
                if(weight <= e.weight)
                        return true;
                return false;
        }

        bool operator>=(const Edge<T>& e)
        {
                if(weight >= e.weight)
                        return true;
                return false;
        }

        //overload << operator
        friend std::ostream& operator<<(std::ostream& stream, Edge<T>& e)
        {
                stream<<"("<<e.u->id<<","<<e.v->id<<")";
                return stream;
        }

};


/*############################################################################
GRAPH

Graph class recieves an input file in the form of a graph matrix. With this
matrix, Graph is able to represent a directed or undirected weighted graph
/*############################################################################*/
template <class T>
```

```
class Graph
{
public:
        Graph(string FILE);//constructor with file name
        ~Graph();//destructor
        Graph(const Graph<T>& g);//copy constructor
        void Kruskal();
        void dfs();
        Graph<T>& operator=(const Graph<T>& g);  // assignment operator

private:
        MinPriorityQueue<Edge<int> > edge_queue;
        int vertex_count;
        List<int> *adjacency;
        void dfs_helper(Vertex<T>* u, int &time);
        Vertex<T> ** vertex_array;
        void copy(const Graph<T>& g);
        void destroy();
};


#include "Graph.cpp"
#endif
```

```
/*##############################################################################
Graph.cpp
Megan Kenworthy & Ansel Schiavone
Project 7
CS 271: Data Structures
April 7th, 2016
##############################################################################*/
#include <string>
#include <fstream>
#include <stdlib.h>
#include <iostream>

using namespace std;

/*##############################################################################
CONSTRUCTOR
precondition: Graph class is declared with a string parameter, which is the
name of the text file from which the graph comes (graph matrix)
postcondition: Graph class is created, complete with adjacency list, list of
vertices, and minimum priority-queue of edge-weights
/*##############################################################################*/
template <class T>
Graph<T>::Graph(string FILE)
{
        vertex_count=0;
        int ** edge_matrix;
        int matrix_position = 0;
        int column_position = 0;
        ifstream myfile;
        string line;
        myfile.open(FILE.c_str());
        getline(myfile,line);   //gets number of vertices
        vertex_count= atoi(line.c_str());
        adjacency = new List<int>[vertex_count];
        vertex_array = new Vertex<T>*[vertex_count];

        //dynamically allocating edge matrix
        edge_matrix = new int*[vertex_count];
        for(int i=0; i< vertex_count; i++)
                edge_matrix[i] = new int[vertex_count];

        //reading vertex information from text file
        while(getline(myfile, line))
        {
                Vertex<T> * theVertex = new Vertex<T>(matrix_position);
                vertex_array[matrix_position] = theVertex;//inserting vertex into vertex_array
;
                column_position = 0;
                for(int i=0; i<line.length(); i++)
                {
                        if(line[i] != ' ')
                        {
                                char letter = line[i];
                                int numletter = letter - '0';
                                edge_matrix[matrix_position][column_position] = numletter;
                                column_position++;
                        }
                        if((line[i] !='0')&&(line[i] != ' '))
                        {
                                int *adj_int = new int;
                                *adj_int = column_position-1;
                                adjacency[matrix_position].insert(0, adj_int);
                        }
```

```
                }
                matrix_position++;
        }

        //Edge matrix converted into MPQ of Edges
        for(int i=0; i<vertex_count; i++)
        {
                for(int j=0; j<vertex_count; j++)
                {
                        if(edge_matrix[i][j]!=0)
                        {
                                //allocating edge, gathering information from edge matrix
                                Edge<int>* edgy = new Edge<int>(edge_matrix[i][j], vertex_arra
y[i], vertex_array[j]);
                                edge_queue.insert(edgy);//insert edge into MPQ
                        }
                }
        }

}

/*#########################################################################
COPY CONSTRUCTOR
precondition: *this is a Graph, and g is a Graph
postcondition: *this is a copy of g
/*#########################################################################*/
template <class T>
Graph<T>::Graph(const Graph<T>& g)
{
        copy(g);
}


/*#########################################################################
COPY
precondition: *this is a Graph, and g is a Graph
postcondition: *this is a copy of g
/*#########################################################################*/
template <class T>
void Graph<T>::copy(const Graph<T>& g)
{
        vertex_count = g.vertex_count;
        edge_queue = g.edge_queue;
        adjacency=g.adjacency;
        vertex_array = new Vertex<T>*[vertex_count];
        for(int i=0; i<vertex_count; i++)
        {
                vertex_array[i]=g.vertex_array[i];
        }
}

/*#########################################################################
DESTRUCTOR
precondition: *this is a graph
postcondition: vertex_array is freed from memory
/*#########################################################################*/
template <class T>
Graph<T>::~Graph()
{
        destroy();
}

/*#########################################################################
```

```
DESTROY
precondition: *this is a graph
postcondition: vertex_array is freed from memory
/*#########################################################################*/
template <class T>
void Graph<T>::destroy()
{
        for(int i=0; i<vertex_count; i++)
        {
                delete vertex_array[i];

        }
        delete [] vertex_array;
}

/*#########################################################################
DFS
precondition: *this is a graph, and vertex_array and adjacency_list have been
created and populated by constructor.
postcondition: all vertices in graph have been visited and are BLACK. Vertices
are printed in order that they are visited.
/*#########################################################################*/
template <class T>
void Graph<T>::dfs()
{

        for(int i=0; i<vertex_count; i++)
        {
                vertex_array[i]->color = "WHITE";
                vertex_array[i]->predv = NULL;
        }
        int time = 0;
        for(int i=0; i<vertex_count; i++)
        {
                if(vertex_array[i]->color == "WHITE")
                        dfs_helper(vertex_array[i], time);
        }
}

/*#########################################################################
DFS helper method (See DFS)
/*#########################################################################*/
template <class T>
void Graph<T>::dfs_helper(Vertex<T>* u, int &time)
{
        time++;
        u->discovery = time;
        cout<<"Vertex "<<u->id<<" : Time Discovered="<<u->discovery<<endl;
        u->color= "GRAY";
        for(int i=0; i< adjacency[u->id].length(); i++)
        {
                if(vertex_array[*adjacency[u->id][i]]->color=="WHITE")
                {
                        vertex_array[*adjacency[u->id][i]]->predv = u;
                        dfs_helper(vertex_array[*adjacency[u->id][i]],time);
                }
        }
        u->color = "BLACK";
        time++;
        u->finish = time;
        cout<<"Vertex "<<u->id<<" : Time Finished="<<u->finish<<endl;

}
```

```
/*#########################################################################
Kruskal
precondition: *this is a graph, and vertex_array and adjacency_list have been
created and populated by constructor.
postcondition: A minimum spanning tree has been established in A (set of edges)
and edges comprising MST are printed.
/*#########################################################################*/
template <class T>
void Graph<T>::Kruskal()
{
        List<Edge<T> > A;//set  of edges in MST
        DisjointSets<Vertex<T> > V(vertex_count);//disjoint set of vertices
        List<DSNode<Vertex<T> > > dsnode_list;
        for(int i=0; i<vertex_count; i++)
        {
                /*dsnode_list saves all DSNode pointers returned by makeSet in
                a linked list, to be used by findSet and unionSet, whose
                parameters require DSNode pointers*/
                dsnode_list.append(V.makeSet(vertex_array[i]));
        }
        int len = edge_queue.length();
        int count = 0;
        for(int i=0; i<len; i++)
        {
                Edge<T>* edge_ptr = edge_queue.extractMin();//extract edges in ascending order
 based on weight

                if(V.findSet(dsnode_list[edge_ptr->u->id]) != V.findSet(dsnode_list[edge_ptr->
v->id]))
                {
                                A.insert(count, edge_ptr);
                                count++;
                                V.unionSets(dsnode_list[edge_ptr->u->id], dsnode_list[edge_ptr
->v->id]);
                }

        }
        cout<<A;
}

/*#########################################################################
ASSIGNMENT OPERATOR
precondition: *this is a graph, and parameters g is a graph
postcondition: *this is now an exact copy of g.
/*#########################################################################*/
template <class T>
Graph<T>& Graph<T>::operator=(const Graph<T>& g)
{
        destroy();
        copy(g);
}
```

```
//RBT.h
//Project 1000
//Megan Kenworthy & Ansel Schiavone

#ifndef RBT_H
#define RBT_H

#include <iostream>
#include <stdio.h>
#include <stdlib.h>

#include "node.h"

template <class KeyType>
class RBT
{

        public:
                RBT();
                        //constructor
                RBT(const RBT<KeyType>& r);
        //copy constructor
                ~RBT();
                        //deconstructor
                RBT<KeyType> operator=(RBT<KeyType>& r);                                //equa
l operator

                bool empty();                                                           //retu
rn true if empty; false o/w
                KeyType *get(const KeyType& k);                         //return first element
 with key equal to k
                void insert(KeyType *k);                                        //insert k int
o the tree
                void insertFixup(Node<KeyType>* r);                     //ensures the properti
es of a RBT
                void leftRotate(Node<KeyType>* r);                      //performs left rotate
 around node r
                void rightRotate(Node<KeyType>* r);                     //performs right rotat
e around node r


                KeyType *maximum();                                                     //retu
rn the maximum element
                KeyType *minimum();                                                     //retu
rn the minimum element
                KeyType *successor(const KeyType& k);           //return the successor of k
                KeyType *predecessor(const KeyType& k);         //return the predecessor of k

                std::string inOrder();                                          //return strin
g of elements from an inorder traversal
                std::string preOrder();                                         //return strin
g of elements from a preorder traversal
                std::string postOrder();                                        //return strin
g of elements from postorder traversal
        private:
                Node<KeyType>* root;                                            //root
                Node<KeyType>* nnode;                                           //NIL node tha
t everything will point to

                Node<KeyType> *getHelper(Node<KeyType>* r, const KeyType& k);
                Node<KeyType>* copy(Node<KeyType>* r, Node<KeyType>* parp, Node<KeyType>* sour
cenil);
                void destroy(Node<KeyType>* r);
```

```
                Node<KeyType> *minhelper(Node<KeyType>* r);
                Node<KeyType> *maxhelper(Node<KeyType>* r);
                Node<KeyType> *successorHelper(const KeyType& k);
                Node<KeyType> *predecessorHelper(const KeyType& k);
                std::string inOrderHelper(Node<KeyType>* r);

                std::string preOrderHelper(Node<KeyType>* r);

                std::string postOrderHelper(Node<KeyType>* r);
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, RBT<KeyType>& k);

class IndexingError { };
class MaximumError { };
class MinimumError { };



#include "RBT.cpp"

#endif
```

```
//RBT.cpp
//Project 1000
//Megan Kenworthy & Ansel Schiavone

#include "node.h"

#include <string>
#include <iostream>
#include <sstream>
#include <stdlib.h>

using namespace std;


/*
 * Implementation notes: Red Black Tree Constuctor
 *-------------------------------------
 *      Pre:  None
 *      Post: Red Black Tree initialized with root set to nnode
 */
template <class KeyType>
RBT<KeyType>::RBT()
{
                nnode = new Node<KeyType>();    //allocate space for nnode
                nnode->color = "BLACK";                 //set nnode's color to black (will alw
ays be black)
                root=nnode;                                                     //sets root to nnode s
ince the tree is empty
}


/*
 * Implementation notes: Deconstructor
 *-------------------------------------
 *      Pre: A Red Black Tree "B"
 *      Post: The memory used for RBT "B" has been freed (deallocated)
 */
template <class KeyType>
RBT<KeyType>::~RBT()
{
        destroy(root);                                          //calls destroy function
}


/*
 * Implementation notes: Copy Constructor
 *-------------------------------------
 *      Pre: A Red Black Tree "B" and this Red Black Tree
 *      Post: B is an exact copy of this Red Black Tree
 */
template <class KeyType>
RBT<KeyType>::RBT(const RBT<KeyType>& b)
{
        root = copy(b.root, nnode, b.nnode);
}


/*
 * Implementation notes: Equal Operator
 *-------------------------------------
 *      Pre: A Red Black Tree b
 *      Post: RBT b is set equal to this RBT
 */
```

```
template <class KeyType>
RBT<KeyType> RBT<KeyType>::operator=(RBT<KeyType>& b)
{
        if(this != &b)                                                          //checking to
make sure you aren't setting this to itself
        {
                destroy(root);
                root = copy(b.root,nnode, b.nnode);
        }
        return *this;
}


/*
 * Implementation notes: Copy
 *-----------------------------------
 *      Pre: A nodpointer r to the root, nodepointer parp to the parent,
 *      and nodepointer sourcenil to to the source's RBT's nnode
 *      Post: all of the nodes in RBT A are copied to this RBT with the
 *  correct parent, item, and color attributes
 *  A nodepointer to the root of this new tree is returned
 */
template <class KeyType>
Node<KeyType>* RBT<KeyType>::copy(Node<KeyType>* r, Node<KeyType>* parp, Node<KeyType>* source
nil)
{
        if(r == sourcenil)                              //checking to see if the root of the i
nput is the same as its nnode
        {
                return nnode;                           //if so, returns nnode
        }

        Node<KeyType>* posernode = new Node<KeyType>(r->item);          //allocates space for
new node of r->item
        posernode->parent = parp;
        //sets parent and color
        posernode->color = r->color;

        posernode->leftChild = copy(r->leftChild, posernode, sourcenil);                //call
s recursively on left and right
        posernode->rightChild = copy(r->rightChild, posernode, sourcenil);              //subt
rees
        return posernode;
}


/*
 * Implementation notes: destroy
 *-----------------------------------
 *      Pre: A nodpointer to the root of a RBT A
 *      Post: All of the nodes in RBT A have been deleted
 */
template <class KeyType>
void RBT<KeyType>::destroy(Node<KeyType>* r)
{
        if (r != nnode)                                 //checks to see if the root equals nno
de
        {
                destroy(r->leftChild);          //calls recursively on left and right subtrees
                destroy(r->rightChild);
                delete r;                                       //deletes each node
        }
}
```

```
/*
 * Implementation notes: Insert
 *-------------------------------------
 *      Pre: A keytype pointer k (to be the item in node)
 *      Post: A node containing k is inserted into the Red Black Tree
 */
template <class KeyType>
void RBT<KeyType>::insert(KeyType *k)
{
        Node<KeyType>* z = new Node<KeyType>(k);

        Node<KeyType>* x = root;
        Node<KeyType>* y = nnode;

        while(x != nnode)                                               //iterates through whi
le x != nnode, meaning RBT isn't empty
        {
                y=x;                                                            //sets y to x

                if( (*z->item) < (*x->item) )           //if the item of what you are insertin
g is less than x item
                        x=x->leftChild;                                 //iterate through x's
left subtree
                else
                        x=x->rightChild;                                //if not iterate throu
gh x's right subtree
        }

        z->parent = y;                                                  //sets z's parent to y

        if(y == nnode)                                                  //if y still equals nn
ode, set the root to z
        {
                root = z;                                                       //assign addre
ss of z to root
        }

        else if( (*z->item) < (*y->item) )                      //checks to see if z's item is
 less than y
                y->leftChild = z;                                               //sets the lef
tchild to z

        else
        {
                y->rightChild = z;                                              //else, sets y
's rightchild to z
        }

        z->leftChild = nnode;                                           //sets z's left and ri
ght children to nnode
        z->rightChild = nnode;

        insertFixup(z);                                                         //calls insert
 fixup on z
}


/*
 * Implementation notes: Insert Fixup
 *-------------------------------------
 *      Pre: A pointer to node r in RBT
```

```
 *      Post: the node r is moved to it's correct position within the RBT
 *  ensuring that the root is black, and all red nodes have black children,
 *  and by calling on helper functions, all properties of the tree
 *  are maintained
 */
template <class KeyType>
void RBT<KeyType>::insertFixup(Node<KeyType>* r)
{
        Node<KeyType>* y;
        //initializes nodepointer y

        while (r->parent->color == "RED")                                       //goes
 through while as long as r's parent is red
        {

                //if r's parent is a leftchild
                if(r->parent == r->parent->parent->leftChild)
                {
                        y = r->parent->parent->rightChild;                      //set'
s y to r's grandparent's rightchild

                        if(y->color == "RED")
//if y is red then y->parent must be
                        {
                        //black to maintain the property that every
                                r->parent->color = "BLACK";
//red node must have two black children
                                y->color = "BLACK";
                                r->parent->parent->color = "RED";
                                r = r->parent->parent;
//moves r to it's granparent
                        }

                        else
                        {
                                if(r == r->parent->rightChild)                  //if r
 is a rightchild
                                {
                                        r = r->parent;
        //set r to it's parent and left rotate on r
                                        leftRotate(r);
                                }
                                r->parent->color = "BLACK";
//set r parent's color to black and grandparent's to red
                                r->parent->parent->color = "RED";
                                rightRotate(r->parent->parent);                 //righ
t rotate around r's grandparent
                        }
                }

                //if r's parent is a rightchild
                else
                {
                        y = r->parent->parent->leftChild;                       //set
y to r's grandparent's leftchild

                        if(y->color == "RED")
//if y is red
                        {
                                r->parent->color = "BLACK";
//set r's parent's color to black and y to black
                                y->color = "BLACK";
                                r->parent->parent->color = "RED";               //set
```

```
r's grandparent's color to red(now has 2 black c's)
                                r = r->parent->parent;
//r is not r's granparent
                        }

                        else
                        {
                                if(r == r->parent->leftChild)                           //if y
's color is plack and r is a leftchild
                                {
                                        r = r->parent;
        //set r to its parent and right rotate on r
                                        rightRotate(r);
                                }

                                r->parent->color = "BLACK";
//set r's parent to be black and grandparent to be red
                                r->parent->parent->color = "RED";
                                leftRotate(r->parent->parent);                          //left
 rotate on r's grandparent
                        }
                }
        }

        root->color = "BLACK";
//always sets root to black to maintain the property
}


/*
 * Implementation notes: Left Rotate
 *-----------------------------------
 *      Pre: A pointer to node r, representing the root of some subtree in an RBT
 *      Post: The subtree previously rooted at r is now balanced
 *  and rooted at r's right child
 */
template <class KeyType>
void RBT<KeyType>::leftRotate(Node<KeyType>* r)
{
                Node<KeyType>* y;                                                       //initializes
and sets y to r's right child
                y = r->rightChild;

                r->rightChild = y->leftChild;                   //turns y's left subtree into
r's right subtree

                if(y->leftChild != nnode)                               //if y's left child is
 not the nnode
                {
                        y->leftChild->parent = r;                       //set the left child's
 parent to r
                }

                y->parent = r->parent;                                  //set y's parent to r'
s parent

                if(r->parent == nnode)                                  //in the case that r's
 parent equals nnode
                {
                        root = y;                                                       //the
root points to y
                }
```

```
                else if(r == r->parent->leftChild)              //in the case that r is a left
child
                {
                        r->parent->leftChild = y;                       //set it's parent's le
ftchild to y
                }

                else
                {
                        r->parent->rightChild = y;                      //else, set the parent
's rightchild to y
                }

                y->leftChild = r;                                               //set y's left
child to r and r's parent to y
                r->parent = y;
}


/*
 * Implementation notes: Right Rotate
 *-------------------------------------
 *      Pre: A pointer to node r, representing the root of some subtree in an RBT
 *      Post: The subtree previously rooted at r is now balanced
 *  and rooted at r's left child
 */
template <class KeyType>
void RBT<KeyType>::rightRotate(Node<KeyType>* r)
{
                Node<KeyType>* y;                                               //initializes
y to be r's leftchild
                y = r->leftChild;

                r->leftChild = y->rightChild;                   //set's r's leftchild to be y'
s rightchild

                if(y->rightChild != nnode)                              //if y's rightchild is
n't the nil node then
                {
                        y->rightChild->parent = r;                      //y's rightchild's par
ent to point to r
                }

                y->parent = r->parent;                                  //set y's parent to r'
s parent

                if(r->parent == nnode)                                  //if r's parent is nno
de then root will equal y
                {
                        root = y;
                }

                else if(r == r->parent->rightChild)             //if r is a leftchild, then it
's parent's rightchild will be y
                {
                        r->parent->rightChild = y;
                }

                else                                                                    //if r
 is a rightchild, then it's parent leftchild will be y
                {
                        r->parent->leftChild = y;
                }
```

```
                y->rightChild = r;                                              //set y's righ
tchild to r and r's parent to y
                r->parent = y;
}


/*
 * Implementation notes: minHelper
 *------------------------------------
 *      Pre: A nodepointer r
 *      Post: A pointer to the node containing the minimum keytype is returned
 */
template <class KeyType>
Node<KeyType>* RBT<KeyType>::minhelper(Node<KeyType>* r)
{
        if(r==nnode)                                            //if the root is nnode, throw
an error because tree is empty
                throw MinimumError();
        if(r->leftChild == nnode)                       //if there is no left child, then r is
 the minimum
                return r;
        return minhelper(r->leftChild);         //call minhelper recursively on the left child

}

/*
 * Implementation notes: minimum
 *------------------------------------
 *      Pre: A Red Black Tree
 *      Post: A keytype pointer to the minimum item in the Red Black Tree
 */
template <class KeyType>
KeyType* RBT<KeyType>::minimum()
{
        Node<KeyType>* themin = minhelper(root);                //call minhelper to get pointe
r to the node with min item
        return themin->item;                                                    //return point
er to minimum item in the RBT
}


/*
 * Implementation notes: maxHelper
 *------------------------------------
 *      Pre: A nodepointer r
 *      Post: A pointer to the node containing the maximum keytype is returned
 */
template <class KeyType>
Node<KeyType>* RBT<KeyType>::maxhelper(Node<KeyType>* r)
{
        if(r==nnode)                                            //if the root is equal to nnod
e throw error as the tree is empty
                throw MaximumError();
        if(r->rightChild == nnode)                      //if there's no right child then retur
n r
                return r;
        return maxhelper(r->rightChild);        //call maxhelper recursively on the rightchild

}


/*
```

```
 * Implementation notes: maximum
 *-------------------------------------
 *      Pre: A Red Black Tree
 *      Post: A keytype pointer to the maximum item in the Red Black Tree
 */
template <class KeyType>
KeyType* RBT<KeyType>::maximum()
{
        Node<KeyType>* themax = maxhelper(root);               //call the helper function for
 recursion
        return themax->item;                                                    //return point
er to the macximum item in the RBT
}


/*
 * Implementation notes: Successor
 *-------------------------------------
 *      Pre: A keytype k
 *      Post: A keytype pointer to the successor of keytype k is returned
 */
template <class KeyType>
KeyType* RBT<KeyType>::successor(const KeyType& k)
{
        Node<KeyType>* succb = successorHelper(k);      //call helper function to recursively
search for successor
        return succb->item;                                                     //return point
er to the successor's item
}


/*
 * Implementation notes: successorhelper
 *-------------------------------------
 *      Pre: A keytype k
 *      Post: A nodepointer to the node containing the successor of k is returned
 */
template <class KeyType>
Node<KeyType>* RBT<KeyType>::successorHelper(const KeyType& k)
{

        Node<KeyType>* theNode = getHelper(root,k);                     //get pointer to node
we want to find the successor of
        if(theNode->rightChild != nnode)                                        //if it has a
right subtree, the successor is min of this subtree
        {
                return minhelper(theNode->rightChild);
        }

        Node<KeyType>* successor = theNode->parent;                     //set successor to the
node's parent

        //while successor isn't the nnode and thenode is equal to successor's righchild
        while( (successor!=nnode) && (theNode==successor->rightChild) )
        {
                theNode = successor;                                                    //set
the node to successor
                successor = successor->parent;                          //set successo
r to it's parent
        }
        return successor;
//return's the first leftchild of an ancestor of thenode
}
```

```
/*
 * Implementation notes: predecessor
 *-------------------------------------
 *      Pre: A keytype k
 *      Post: A keytype pointer to the predecessor of keytype k is returned
 */
template <class KeyType>
KeyType* RBT<KeyType>::predecessor(const KeyType& k)
{
        Node<KeyType>* predb = predecessorHelper(k);    //call helper function to recursively
search for predecessor
        return predb->item;                                                             //retu
rn pointer to item of predecessor
}

/*
 * Implementation notes: predecessorhelper
 *-------------------------------------
 *      Pre: A keytype k
 *      Post: A nodepointer to the node containing the predecessor of k is returned
 */
template <class KeyType>
Node<KeyType>* RBT<KeyType>::predecessorHelper(const KeyType& k)
{

        Node<KeyType>* theNode = getHelper(root,k);             //get the node that you want t
o find the predecessor of
        if(theNode->leftChild != nnode)                                 //if it has a left sub
tree, then pred is the max of left subtree
        {
                return maxhelper(theNode->leftChild);
        }

        Node<KeyType>* successor = theNode->parent;

        //while successor isn't nnode and thenode is equal to the left child
        while( (successor!=nnode) && (theNode==successor->leftChild) )
        {
                theNode = successor;                    //set thenode to successor and success
or to it's parent
                successor = successor->parent;
        }
        return successor;                                       //return pointer to the first
ancestor's rightchild
}


/*
 * Implementation notes: Empty
 *-------------------------------------
 *      Pre: A Red Black Tree "A"
 *      Post: The boolean returns false if A is not empty
 *  and true if the root is nnode, meaning the RBT has no elements
 */
template <class KeyType>
bool RBT<KeyType>::empty()
{
        if (root==nnode)                        //if root is nnode, it has no children and thu
s the tree is empty
                return true;
        else                                            //else return false
                return false;
```

```
}


/*
 * Implementation notes: Get
 *-------------------------------------
 *      Pre: A keytype K
 *      Post: the item pointer to keytype k is returned if found
 */
template <class KeyType>
KeyType* RBT<KeyType>::get(const KeyType& k)
{
                Node<KeyType>* temp = getHelper(root,k);        //will return a nodepointer
                return temp->item;                                                      //retu
rn item pointer
}


/*
 * Implementation notes: getHelper
 *-------------------------------------
 *      Pre: Nodepointer r and keytype k
 *      Post: A nodepointer to the node containing k is returned if k is found, and
 *  an IndexError is thrown otherwise
 */
template <class KeyType>
Node<KeyType>* RBT<KeyType>::getHelper(Node<KeyType>* r, const KeyType& k)
{
        if(r == nnode)
        {
                throw IndexingError();                                  //throw exception
        }

        if(*(r->item) == k)
        {
                return r;                                                               //if t
he item is equal to k return node r
        }

        if(*(r->item) > k)
                return getHelper(r->leftChild, k);              //if item > k call the leftChi
ld recursively
        else
                return getHelper(r->rightChild, k);             //if item < k call the rightCh
ild recursively
}


/*
 * Implementation notes: inOrder
 *-------------------------------------
 *      Pre: A Red Black Tree A
 *      Post: A string representation of RBT A returned with items in order (leftchild,parent,
rightchild)
 */
template <class KeyType>
std::string RBT<KeyType>::inOrder()
{
        string toorder = inOrderHelper(root);   //call helper function to use recursion
        return toorder;
}

/*
```

```
 * Implementation notes: inOrderhelper
 *------------------------------------
 *      Pre: A nodepointer r to RBT A
 *      Post: A string representation of RBT A returned with items in order (leftchild,parent,
rightchild)
 */
template <class KeyType>
std::string RBT<KeyType>::inOrderHelper(Node<KeyType>* r)
{
        stringstream goingout;
        if(r==nnode)                                                                    //if t
he tree is empty return empty string
        {
                return "";
        }

        else
        {
                goingout<<inOrderHelper(r->leftChild);          //inOrder traversal should be
left,root,right
                goingout<<*(r->item)<<": "<<r->color<<" ";
                goingout<<inOrderHelper(r->rightChild);
        }
                return goingout.str();
}

/*
 * Implementation notes: preOrder
 *------------------------------------
 *      Pre: A Red Black Tree A
 *      Post: A string representation of RBT A returned with items in pre order (parent,leftch
ild,rightchild)
 */
template <class KeyType>
std::string RBT<KeyType>::preOrder()
{
        string preorderit = preOrderHelper(root); //call helper function to use recursion
        return preorderit;
}

/*
 * Implementation notes: preOrderhelper
 *------------------------------------
 *      Pre: A nodepointer r to RBT A
 *      Post: A string representation of RBT A returned with items in pre order (parent,leftch
ild,rightchild)
 */
template <class KeyType>
std::string RBT<KeyType>::preOrderHelper(Node<KeyType>* r)
{
        stringstream goingout;
        if(r==nnode)                                                                    //is t
he root equals nnode, return empty string
                return "";
        else
        {
                goingout<<*(r->item)<<": "<<r->color<<" ";
                goingout<<preOrderHelper(r->leftChild);         //preOrder traversal should be
 root,left,right
                goingout<<preOrderHelper(r->rightChild);
        }
                return goingout.str();
}
```

```
/*
 * Implementation notes: postOrder
 *------------------------------------
 *      Pre: A Red Black Tree A
 *      Post: A string representation of RBT A returned with items in post order (leftchild,ri
ghtchild,parent)
 */
template <class KeyType>
std::string RBT<KeyType>::postOrder()
{
        string postorderit = postOrderHelper(root);             //call helper function to use
recursion
        return postorderit;
}


/*
 * Implementation notes: postOrderhelper
 *------------------------------------
 *      Pre: A nodepointer r to RBT A
 *      Post: A string representation of RBT A returned with items in post order (leftchild,ri
ghtchild,parent)
 */
template <class KeyType>
std::string RBT<KeyType>::postOrderHelper(Node<KeyType>* r)
{
        stringstream goingout;
        if(r==nnode)                                                                    //if t
he root equal nnode return empty string
                return "";
        else
        {

                goingout<<postOrderHelper(r->leftChild);
                goingout<<postOrderHelper(r->rightChild);       //postOrder traversal should b
e left,right,root
                goingout<<*(r->item)<<": "<<r->color<<" ";
        }
                return goingout.str();
}

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, RBT<KeyType>& k)
{
        stream<<k.inOrder();
        return stream;
}
```

```
//dict.h
//Project 1000
//Megan Kenworthy & Ansel Schiavone

#ifndef DICTIONARY_H
#define DICTIONARY_H

#include <iostream>
#include <stdio.h>
#include <stdlib.h>

#include "RBT.h"

//A templated dictionary class
template <class KeyType>
class Dictionary: public RBT<KeyType>
{
        public:

                Dictionary(): RBT<KeyType>() {}         //constructor calls on RBT constuctor


                using RBT<KeyType>::empty;                      //uses all necessary RBT metho
ds
                using RBT<KeyType>::get;
                using RBT<KeyType>::insert;
                using RBT<KeyType>::inOrder;

};

//ostream operator overloads to output inOrder string representation
template <class KeyType>
std::ostream& operator<<(std::ostream& stream, Dictionary<KeyType>& dict)
{
        stream<<dict.inOrder();
        return stream;
}

#endif
```

```
//test RBT
//Megan Kenworthy and Ansel Sciavone
//Project 1000

#include <string>
#include <iostream>
#include <cassert>
#include <cstdlib>
#include "RBT.h"

using namespace std;

//Tests to see if the empty function returns true when
//No items are in the tree, and false otherwise
void testEmpty()
{
                RBT<int> testtree;

                assert(testtree.empty() == true);

                int* p1 = new int;
                *p1 = 1;
                testtree.insert(p1);

                assert(testtree.empty() == false);
}

//Tests tp see if the get function returns
//the correct pointer to the element that
//we want to get, and checks to see if an
//error will be thrown if we try to get an
//element that is not in the tree
void testGet()
{
                RBT<int> testtree;

                int* p1 = new int;
                *p1 = 1;
                int* p2 = new int;
                *p2 = 2;
                int* p3 = new int;
                *p3 = 56;
                int* p4 = new int;
                *p4 = 22;
                int* p5 = new int;
                *p5 = 35;
                int* p6 = new int;
                *p6 = 8;

                testtree.insert(p1);
                testtree.insert(p2);
                testtree.insert(p3);
                testtree.insert(p4);
                testtree.insert(p5);
                testtree.insert(p6);


                assert(*(testtree.get(56)) == 56);
                assert(*(testtree.get(35)) == 35);


                try{
                        testtree.get(3);
```

```
                }
                catch(IndexingError e)
                {
                        cout<<"Error! Trying to get an element not in the tree"<<endl;
                }
}

//tests to see if the copy constructor and
//equal operator function properly after inserting
//elements into the tree
void testInsertcopandeq()
{

                RBT<int> tester;

                int* p1 = new int;
                *p1 = 1;
                int* p2 = new int;
                *p2 = 2;
                int* p3 = new int;
                *p3 = 56;
                int* p4 = new int;
                *p4 = 22;
                int* p5 = new int;
                *p5 = 35;
                int* p6 = new int;
                *p6 = 8;

                tester.insert(p1);
                tester.insert(p2);
                tester.insert(p3);
                tester.insert(p4);
                tester.insert(p5);
                tester.insert(p6);


                assert(tester.empty() == false);


                assert(tester.inOrder() == "1: BLACK 2: BLACK 8: RED 22: BLACK 35: RED 56: BLA
CK ");


                int* p7 = new int;
                *p7 = 10;

                tester.insert(p7);
                assert(tester.inOrder() == "1: BLACK 2: BLACK 8: RED 10: BLACK 22: RED 35: RED
 56: BLACK ");


                RBT<int> copytest(tester);
                assert(copytest.inOrder() =="1: BLACK 2: BLACK 8: RED 10: BLACK 22: RED 35: RE
D 56: BLACK ");


                int* a1 = new int;
                *a1 = 12;
                int* a2 = new int;
                *a2 = 44;
                int* a3 = new int;
                *a3 = 7;
                int* a4 = new int;
```

```
                *a4 = 21;
                int* a5 = new int;
                *a5 = 32;
                int* a6 = new int;
                *a6 = 1;

                tester.insert(a1);
                tester.insert(a2);
                tester.insert(a3);
                tester.insert(a4);
                tester.insert(a5);
                tester.insert(a6);

        RBT<int> tomakeeq;
                tomakeeq = tester;

                assert(tomakeeq.inOrder()=="1: BLACK 1: RED 2: BLACK 7: RED 8: BLACK 10: BLACK
 12: BLACK 21: RED 22: BLACK 32: RED 35: BLACK 44: RED 56: BLACK ");

                assert(*(tomakeeq.get(1))==1);
                assert(*(tomakeeq.get(44))==44);
                assert(*(tomakeeq.successor(8))==10);
                assert(*(tomakeeq.predecessor(7))==2);

                RBT<int> bebe;
                bebe.insert(a1);
                bebe.insert(a2);
                bebe = tomakeeq;
                assert(bebe.inOrder()=="1: BLACK 1: RED 2: BLACK 7: RED 8: BLACK 10: BLACK 12:
 BLACK 21: RED 22: BLACK 32: RED 35: BLACK 44: RED 56: BLACK ");

}


//Tests to see if the maximum function returns the
//pointer correctly to the maximum element in the tree,
//also checks to see if a maximum error will be thrown
//if the tree is empty
void testMax()
{
                RBT<int> testtree;

                try{
                        testtree.maximum();
                }
                catch(MaximumError e)
                {
                        cout<<"Error! Trying to find max in an empty tree"<<endl;
                }


                int* p1 = new int;
                *p1 = 11;
                int* p2 = new int;
                *p2 = 20;
                int* p3 = new int;
                *p3 = 5;
                int* p4 = new int;
                *p4 = 2;
                int* p5 = new int;
                *p5 = 85;
                int* p6 = new int;
                *p6 = 10;
```

```
		testtree.insert(p1);
		testtree.insert(p2);
		testtree.insert(p3);
		testtree.insert(p4);
		testtree.insert(p5);
		testtree.insert(p6);

		assert(*(testtree.maximum()) == 85);
}

//Tests to see if the minimum function returns the
//pointer correctly to the minimum element in the tree,
//also checks to see if a minimum error will be thrown
//if the tree is empty
void testMin()
{

		RBT<int> testtree;

		try{
			testtree.minimum();
		}
		catch(MinimumError e)
		{
			cout<<"Error! Trying to find min in an empty tree"<<endl;
		}

		int* p1 = new int;
		*p1 = 111;
		int* p2 = new int;
		*p2 = 2;
		int* p3 = new int;
		*p3 = 51;
		int* p4 = new int;
		*p4 = 28;
		int* p5 = new int;
		*p5 = 85;
		int* p6 = new int;
		*p6 = 34;

		testtree.insert(p1);
		testtree.insert(p2);
		testtree.insert(p3);
		testtree.insert(p4);
		testtree.insert(p5);
		testtree.insert(p6);

		assert(*(testtree.minimum()) == 2);
}

//tests to see if a pointer to the successor of
//a given element is returned properly
//and throws an indexing error if the given
//element has no successor (tree is just the root)
void testSuccessor()
{

		RBT<int> testtree;

		try{
			testtree.successor(1);
		}
```

```
                catch(IndexingError e)
                {
                        cout<<"Error! Trying to find successor of element not in tree"<<endl;
                }

                int* p1 = new int;
                *p1 = 11;
                int* p2 = new int;
                *p2 = 28;
                int* p3 = new int;
                *p3 = 32;
                int* p4 = new int;
                *p4 = 7;
                int* p5 = new int;
                *p5 = 16;
                int* p6 = new int;
                *p6 = 34;

                testtree.insert(p1);
                testtree.insert(p2);
                testtree.insert(p3);
                testtree.insert(p4);
                testtree.insert(p5);
                testtree.insert(p6);

                assert(*(testtree.successor(16)) == 28);
                assert(*(testtree.successor(32)) == 34);
}

//tests to see if a pointer to the predecessor of
//a given element is returned properly
//and throws an indexing error if the given
//element has no predecessor (tree is just the root)
void testPredecessor()
{
                RBT<int> testtree;

                try{
                        testtree.predecessor(1);
                }
                catch(IndexingError e)
                {
                        cout<<"Error! Trying to find predecessor of element not in tree"<<endl
;
                }

                int* p1 = new int;
                *p1 = 1;
                int* p2 = new int;
                *p2 = 2;
                int* p3 = new int;
                *p3 = 17;
                int* p4 = new int;
                *p4 = 7;
                int* p5 = new int;
                *p5 = 23;
                int* p6 = new int;
                *p6 = 52;

                testtree.insert(p1);
                testtree.insert(p2);
                testtree.insert(p3);
                testtree.insert(p4);
```

```
                testtree.insert(p5);
                testtree.insert(p6);

                assert(*(testtree.predecessor(7)) == 2);
                assert(*(testtree.predecessor(2)) == 1);
}

//tests to make sure that the in order string conversion
//returns all of the elements of the RBT in order
//the inOrder() is used in stream insertion, and as such
//is used in the above asserts
void testInOrder()
{
                RBT<int> testtree;

                int* p1 = new int;
                *p1 = 1;
                int* p2 = new int;
                *p2 = 2;
                int* p3 = new int;
                *p3 = 17;
                int* p4 = new int;
                *p4 = 7;
                int* p5 = new int;
                *p5 = 23;
                int* p6 = new int;
                *p6 = 52;

                testtree.insert(p1);
                testtree.insert(p2);
                testtree.insert(p3);
                testtree.insert(p4);
                testtree.insert(p5);
                testtree.insert(p6);

                assert(testtree.inOrder() == "1: BLACK 2: BLACK 7: BLACK 17: RED 23: BLACK 52:
 RED ");

                int *p7 = new int;
                *p7 = 4;
                testtree.insert(p7);

                assert(testtree.inOrder() == "1: BLACK 2: BLACK 4: RED 7: BLACK 17: RED 23: BL
ACK 52: RED ");
}

//tests the pre order string conversion to ensure
//that it is representing the tree through a
//string representation of a pre order traversal
//(parent, left, right)
void testPreOrder()
{
                RBT<int> testtree;

                int* p1 = new int;
                *p1 = 16;
                int* p2 = new int;
                *p2 = 2;
                int* p3 = new int;
                *p3 = 17;
                int* p4 = new int;
                *p4 = 8;
                int* p5 = new int;
```

```
                *p5 = 4;
                int* p6 = new int;
                *p6 = 52;

                testtree.insert(p1);
                testtree.insert(p2);
                testtree.insert(p3);
                testtree.insert(p4);
                testtree.insert(p5);
                testtree.insert(p6);

                assert(testtree.preOrder() == "16: BLACK 4: BLACK 2: RED 8: RED 17: BLACK 52:
RED ");
}

//tests the post order string conversion to ensure
//that it is representing the tree through a
//string representation of a post order traversal
//(left, right, parent)
void testPostOrder()
{
                RBT<int> testtree;

                int* p1 = new int;
                *p1 = 16;
                int* p2 = new int;
                *p2 = 2;
                int* p3 = new int;
                *p3 = 17;
                int* p4 = new int;
                *p4 = 8;
                int* p5 = new int;
                *p5 = 4;
                int* p6 = new int;
                *p6 = 52;

                testtree.insert(p1);
                testtree.insert(p2);
                testtree.insert(p3);
                testtree.insert(p4);
                testtree.insert(p5);
                testtree.insert(p6);

                assert(testtree.postOrder() == "2: RED 8: RED 4: BLACK 52: RED 17: BLACK 16: B
LACK ");
}


int main()
{
        testEmpty();
        testGet();
        testInsertcopandeq();
        testMax();
        testMin();
        testSuccessor();
        testPredecessor();
        testInOrder();
        testPreOrder();
        testPostOrder();

        return 0;
}
```

```
#include <iostream>
#include <string.h>
#include <cstdlib>
using namespace std;


string OGstring = "h";
const int n = 1;



void LPS_length(int B[][n])
{
        for(int i = 0; i<n; i++)
        {
                for(int k=0 ; k<n ; k++)
                {
                        B[i][k] = 0;
                }
        }
        for(int i=0; i<n; i++)
        {
                B[i][i]=1;
        }

        for(int l=0; l<n; l++)
        {
                for(int i=0; i<n-l; i++)
                {
                        for(int j=l; j<n; j++)
                        {

                                if(j<i)
                                {
                                        B[i][j]=0;
                                }
                                else if(i==j)
                                {
                                        B[i][j]=1;
                                }

                                else if(OGstring[i]==OGstring[j])
                                {
                                                B[i][j]=(B[i+1][j-1])+2;
                                }
                                else
                                {

                                        if(B[i][j-1]>B[i+1][j])
                                        {
                                                B[i][j]=B[i][j-1];
                                        }
                                        else
                                        {
                                                B[i][j]=B[i+1][j];
                                        }

                                }
                        }
                }
        }

}
```

```
char* getstring(int B[][n], int i, int j, char newstring[])
{
        //cout<<"this is real string"<<OGstring<<endl;
        if(i>=j)
        {
                newstring[i]=OGstring[i];
                cout<<"in the i eq j     "<<newstring[i]<<endl;
                //cout<<"in string if for   "<<newstring[i]<<endl;
                //cout<<"heres newstring   "<<newstring<<endl;
        }
        else if( (B[i+1][j-1])==0)
        {
                cout<<"only once"<<endl;
                newstring[i]=OGstring[j-1];
                getstring(B,i+1,j-1,newstring);
        }
        else
        {
                if((B[i][j-1]==B[i+1][j-1])&&(B[i+1][j-1]==B[i+1][j]))
                {

                        //cout<<"here"<<endl;
                        cout<<"this is og i    "<<OGstring[i]<<endl;
                        cout<<"this is og j    "<<OGstring[j]<<endl;
                        newstring[i]=OGstring[i];
                        newstring[j]=OGstring[j];
                        cout<<"newstring in here "<<newstring[i]<<endl;
                        getstring(B, i+1, j-1,newstring);

                }
                else
                {

                        if( (B[i][j-1]>B[i+1][j]))
                        {
                                cout<<"everything else"<<endl;
                                getstring(B,i,j-1, newstring);
                        }
                        else
                                getstring(B,i+1,j,newstring);
                }
        }
        //cout<<newstring;
        return newstring;

}
int main()
{

        char newstring[n];
        for(int i=0; i<n; i++)
        {
                newstring[i]='$';
        }
        int B[n][n];
        LPS_length(B);
                for(int i=0; i<n; i++)
        {
                for(int j=0; j<n; j++)
                {
                        cout<<B[i][j];
                }
                cout<<endl;
```

```
        }

        getstring(B,0,n-1, newstring);
        for(int j=0; j<n; j++)
        {
                if(newstring[j]!='$')
                        cout<<newstring[j];

        }
        cout<<endl;
        return 0;
}
```