DATA STRUCTURES: PORTFOLIO

December 18, 2017

Anamay Agnihotri
Denison University
CS 271: Data Structures

Contents

INTRODUCTION

CS 271 is a course about the fundamental methods of representing data and the algorithms that implement and use those data representation techniques. Data structures and algorithms that were a part of this class included linked lists, stacks, queues, trees, heaps, priority queues, hashing, searching, sorting, data compression, graphs, recursion.

Analysis topics included elementary big-O analysis, empirical measurements of performance, time/space trade-offs, and identifying differences among best, average, and worst case behaviors (time complexities) across various data structures. The objective of the course was to understand and identify efficient and effective data structures to solve a myriad of problems of varying complexity and utility.

Over the course of this portfolio, I will be going over some of the major learning outcomes of this class, along with identifying the key features across certain data structures that make them unique. I will begin with defining the parameters of analysis, namely the Big-O notation, followed by delving deeper into linear data structures, more complex data structures like Min PQs and heaps, Graphs, Hash Tables, BSTs and RBTs with the goal of understanding and analyzing their effectiveness in terms of data storage and efficiencies.

ANALYSIS OF ALGORITHMS

Competency: Mastery

An algorithm is defined as a computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. It is thus a sequence of computational steps that transform the input into the output. An algorithm is said to be correct if, for every input instance, it halts with the correct output. In computer science, we establish the validity of an algorithm by this halting property which is known as a loop invariant.

A loop invariant is a condition that is necessarily true immediately before and immediately after each iteration of a loop. These conditions are implicit within each algorithm and are known as pre and post conditions. A precondition is a condition or predicate that must always be true just prior to the execution of the algorithm. A postcondition is the condition that must always be true just after the execution of the algorithm. A loop invariant, thus, must satisfy both the pre and postconditions of the algorithm in order for the algorithm to be considered programmatically valid. Within each loop invariant, there are three major steps: the initialization, the maintenance and the termination steps.

The initialization condition holds true before the beginning of the loop while the maintenance step continues to hold at the end of each iteration. The termination condition ensures that the loop invariant ends at some point, making sure the algorithm is not in an endless spiral.
For example, below is the pseudo-code for a linear search algorithm.

```
LinearSearch(A,v)
for (int i=0 to A.length)
        if (A[i]==v)
                return i
return NIL
```

The loop invariant for this algorithm is that before each iteration i of the for loop, all elements in $A[0,...,i-1]$ are not equal to v.

To prove this loop invariant, we have to prove the individual initialization and maintenance steps (which can also be seen as pre and post conditions respectively)

Initialization: Before the $i^{th}(=0)$ iteration of the for loop, the loop invariant states that all the values in $A[0\dots i-1](=A[0\cdots-1]$ are not equal to the value of v. Since $A[0\cdots-1]$ is empty, the statement is vacuously true.

Maintenance: Assume that the loop invariant is true before some iteration j, that is, all elements in A[0...j-1] are not equal to the value of v. For iteration j, there are two possibilities. One, A[j] does not equal v. In this case, before the iteration j+1, we know that all values in $A[0\dots j]$ are not equal to v. Two, A[j] equals v. In this case, the for loop terminates and the j+1 iteration is never executed, thereby making values in $A[0\dots j-1]$ not equal to v, since A[j] equals v. Clearly, in either of these cases, the loop invariant remains true.

Once we prove the validity of an algorithm, it's efficiency is the next topic of analysis. In computer science, the Big O notation is used to describe the performance or complexity of an algorithm. It describes the worst-case scenario, and can be used to describe the execution time required or the space used by an algorithm. For a given problem of size n, which is usually the number of items, saying some equation f(n) = $O(g(n))$ means it is less than some constant multiple of g(n).

There is also the Big-Theta notation (Θ()) which is defined for a function f(n) belonging to the set $\Theta(g(n))$ if there exists positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$ for sufficiently large n. While the Big-O notation provides the upper bound for the time complexity of an algorithm, the Big-Theta provides both the upper and the lower bounds. The lower bound, specifically, is referred to as the Big-Omega notation.

Below is the proof showing the Big-Theta notation definition with respect to the bounds on the time complexity of the algorithm:

For any two functions f (n) and g(n), we have f(n) = $\Theta(g(n))$ if and only if f (n) = $O(g(n))$ and f (n) = $\Omega(g(n))$. (Theorem 3.1, Introduction to Algorithms 3rd Edition, pg 45)

Proof. By definition,
Θ(g(n)) = {f(n): ∃ positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ \forall n \ge n_0$}
O(g(n)) = {f(n): ∃ positive constants c and n_0 such that $0 \le f(n) \le cg(n)\ \forall n \ge n_0$}
Ω(g(n)) = {f(n): ∃ positive constants c and n_0 such that $0 \le cg(n) \le f(n)\ \forall n \ge n_0$}

To prove the above theorem, we need to show that:

$$\Theta(g(n)) \in O(g(n)) \wedge \Omega(g(n))$$

and

$$O(g(n)) \wedge \Omega(g(n)) \in \Theta(g(n))$$

Using the above stated definitions of $O(g(n))$ and $\Omega(g(n))$,
we can combine them such that,
$O(g(n)) \wedge \Omega(g(n))$
$= \{f(n) : \exists$ positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n)$ *and* $0 \le f(n) \le c_2 f(n) \forall n \ge n_0\}$
$= \{f(n) : \exists$ positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 f(n) \forall n \ge n_0\}$
Thus, $O(g(n)) \wedge \Omega(g(n)) \in \Theta(g(n))$.

Using the definition of $\Theta(g(n))$,
we can separate out the inequality such that,
$\Theta(g(n)) = \{f(n) : \exists$ positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n)$ and $0 \le f(n) \le c_2 g(n), \forall n \ge n_0\}$
Thus, $\Theta(g(n)) \in O(g(n)) \wedge \Omega(g(n))$.
Hence proved. □

We can also calculate the time complexity of algorithms using the above obtained definitions.

Prove that: $n^2 + 3n - 20 = O(n^2)$

Proof. By definition,
$O(n^2) = \{f(n) : \exists$ positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$ $\}$
We thus need to show that there exists a positive real number c and n_0 such that $0 \le n^2 + 3n - 20 \le cn^2$.
Let c = 3.
Thus, we need to show that:

$$0 \le n^2 + 3n - 20 \le 3n^2$$

n	$n^2 + 3n - 20$	$3n^2$
1	-16	3
2	-10	12
3	-2	27
4	8	48

As seen in the above table, for all values of $n > 4$, the definition is satisfied for c=3 and $n_0 = 4$
Hence proved. $\square$

Asymptotic complexities can also be applied to recursive algorithms. Below is an example of this.

Prove that: $T(n) = 7T(n/3) + n^2$

Proof.

$$\begin{aligned}
T(n) &= 7T(n/3) + n^2 \\
&= 7(7T(n/9) + (n/3)^2) + n^2 \\
&= 49T(n/9) + 7(n/3)^2 + n^2 \\
&= 343T(n/27) + 49(n/9)^2 + 7(n/3)^2 + n^2 \\
&\textit{Observing this pattern:} \\
&= 7^i T(n/3^i) + \sum_{j=0}^{i-1} 7^j (n/3^j)^2 \\
&n/3^i = 2 \quad \therefore n = 2/3^i \quad \textit{i.e } i = log_3(n/2) \\
&= 7^{log_3(n/2)} T(n/3^{log_3(n/2)}) + \sum_{j=0}^{log_3(n/2)-1} 7^j (n/3^j)^2 \\
&= 7^{log_3(n/2)} T(2) + n^2 \sum_{j=0}^{log_3(n/2)-1} 7^j (1/3^{2j}) \\
&= a7^{log_3(n/2)} + n^2 \sum_{j=0}^{log_3(n/2)-1} (7/9)^j \ldots (T(n \le 2) = a \textit{ (constant time)}) \\
&< a7^{log_3(n/2)} + n^2 \sum_{j=0}^{\infty} (7/9)^j \\
&= a7^{log_3(n/2)} + n^2 (1/(1-7/9)) \\
&= a7^{log_3(n/2)} + n^2 (9/2) \\
&= a7^{log_3(n/2)} + cn^2 \ldots \textit{ (where c= 9/2)}
\end{aligned}$$

Now that we have a closed form to express $T(n)$,
Let us prove that $T(n) < a7^{log_3(n/2)} + cn^2$ where $c > 0$ by using mathematical induction on n.
For the basis step, let n=2.

$$\therefore T(2) < a7^{log_3(2/2)} + c2^2$$

$$a < a + 4c$$

This is clearly true.
For the induction hypothesis, let us assume that:

$$T(k) = a7^{log_3(k/2)} + ck^2$$

for all $k = 1, 2, 3 \ldots, n-1$

We have to show that,

$$\begin{aligned}
T(n) &= 7T(n/3) + n^2 \\
&< 7(a7^{log_3((n/3)/2)}) + c(n/3)^2) + n^2 \ldots \textit{ (from the Induction Hypothesis)} \\
&= a(7^{2+log_3(n/6)}) + 7c(n^2/9) + n^2 \\
&= a(7^{2+log_3(n/6)}) + n^2(7c/9 + 1) \\
&= a(7^{2+log_3(n/6)}) + n^2((7c + 9)/9) \\
&= a(7^{log_3 9+log_3(n/6)}) + Cn^2 \ldots \textit{(where C = (7c+9)/9)} \\
&= a(7^{log_3 3n/2}) + Cn^2 \\
&= a(7^{1+log_3 n/2}) + Cn^2 \\
&= a(7^{log_3 n/2}) + Cn^2
\end{aligned}$$

Thus, using mathematical induction, we have shown that:

$$T(n) < a7^{log_3(n/2)} + cn^2$$

Hence proved. □

Linear data structures

Competency: Mastery with Distinction

A data structure is said to be linear if the elements (data) being stored in them form a sequence. The most basic linear data structure is a list. In C++, this is referred to as an array where its size is fixed and thus occupies a set portion of memory during the initialization phase itself. Items can be inserted into each 'slot' of the list with reference to each index and can take linear time ($O(n)$) to execute depending on the ordering of the elements within the list. There is, however, a linked list implementation which can substantially improve the efficiency of certain operations.

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL. The below code (node.h) shows a sample implementation of a node. Since this nodes retain information about the next and the previous elements with respect to each node, insertions and deletions as take constant time ($O(1)$) compared to the $O(n)$ of a regular array. In a linked list, we can directly insert elements at the beginning since there is no fixed position of the nodes. To implement this, we make the head pointer point to the node and make the right pointer of the node point to the previous 'first' element, thereby not having to traverse the entire list to put in new items. During deletion, if the position of the node is known, we simply delete the node and connect the right pointer of the deleted node to the left pointer of the deleted node, thereby maintaining the connections across the data structure. In an array, we would have not only have to find the position of the element to delete but also have to move all the other elements in the correct orientation. For example if the first element in the array was being deleted, we would have to copy all the values coming after the first element a position to the left, making the time complexity $O(n)$

Stacks are another example of linear structures. A stack is a basic data structure that can be logically thought as linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack. In terms of insertions and deletion, stacks are similar to linked lists in complexity $O(1)$. Stacks are often implemented as linked lists with a Last In First Out (LIFO) format in C++.

Queues are another example of linear data structures that are implemented as linked lists but are First In First Out (FIFO). Because of the linked list implementation, they also perform insertions and deletions in constant time $O(n)$.

Clearly, a linked list has far ranging applications into different data types. The fact that C++ allows us to use data structures to improve and optimize other data structures is referred to as abstraction.

An abstract data type (ADT) is a mathematical model for data types, where a data type is defined by its behavior from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. This contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user. In this case, we use the linked list as an ADT to implement queues and stacks.

So far, we have discussed the time complexities of linear data structures with respect to insertion and deletion operations. In deletion, we assumed we knew the location of the item to delete but in reality, we would need to find the element programmatically. In all the above mentioned data structures (arrays, linked lists, queues and stacks), searches take linear time ($O(n)$). In order to find a specific element, we need to traverse the entire list which, in the worst-case, would take n steps. Thus, we need to have more diverse forms of data structures (beyond linear ones) that can perform faster and better searches as well as keep the efficiencies of other operations.

```
//======================================================================
// Node class for implementing singly/doubly linked lists
//======================================================================
class Node
{
        public:
                //Instance variables
                char ch;                        // keeping a character
                int freq;                       // and frequency
                Node *left;
                Node *right;
                bool hasc;              //flag to indicate if the node has a character

                Node(int);
                Node(char ch, int freq);
                bool operator<(const Node& node2)const;
                bool operator>(const Node& node2)const;
                std::string toString() const;
};

/*
 * Implementation notes: Constructor
 *--------------------------------------
 *      Pre: A character
 *      Post: Node initialized with the character
 */
Node::Node (int freq)
{
        left = right = NULL;
        this-> ch = '\0';
        this-> freq = freq;
        hasc = false;
}
/*
 * Implementation notes: Constructor
 *--------------------------------------
 *      Pre: character, frequency and L/R pointers to nodes
 *      Post: Node initialized with these elements and pointing accordingly
 */
Node::Node(char ch, int freq)
{
        this -> ch = ch;
        this -> freq = freq;
        left = NULL;
        right=NULL;
        hasc = true;
}
/*
 *  Overloading <
 *--------------------------------------
 *      Pre: Node
 *      Post: Returns a boolean if this frequency < node's frequency
 */
bool Node::operator<(const Node& node2)const
{
        return (freq < node2.freq);
}
/*
 *  Overloading >
 *--------------------------------------
 *      Pre: Node
 *      Post: Returns a boolean if this frequency < node's frequency
 */
bool Node::operator>(const Node& node2)const
{
        return (freq > node2.freq);
}
/*=========================================================================================
=======*/
```

PRIORITY QUEUES AND BINARY HEAPS

Competency: Mastery with Distinction

As mentioned in the above section, most linear data structures perform searches in linear time even though some operations like insertion and deletion can be performed in constant time. Thus, we use heaps as an ADT while implementing the priority queue data structure to optimize various operations.

A heap is a specialized tree-based data structure that satisfied the heap property: if B is a child node of A, then key(A) ≤ (or ≥) key(B). This implies that an element with the smallest (or greatest) key is always in the root node. Depending on the ordering, a heap is either a min-heap (smallest value at root) or a max-heap (largest value at root). As seen in the code snippet below ($heap_2$.cpp), each method within the heap data structure performs a specific function which makes it's implementation into other data structures robust and efficient.

As seen in the code for heaps below, the constructor calls the buildHeap() method which implicitly calls the heapify method. Heapify is the method that builds the heap structure by putting the smallest item to the top of the "tree" (can be implemented as a linked list or array) and restores the structure such that the parent is always smaller than the two child nodes. The buildHeap() method practically calls heapify on each (index of each) element and builds the heap. Moreover, it is important to realize that since each heap has a parent with two children, for n nodes, the real height is to the inverse order to 2, making the overall height of the tree log(n). This makes the worst case run-time complexity of heapify $Oln(n)$ as it has to only make log(n) comparisons while establishing the heap property. Clearly, for buildHeap(), which runs heapify n times, inserting each element will take constant time, making it's time complexity $O(n)$.

The Heapsort() method, however, is provides the real improvement in terms of efficiency. Heapsort helps us arrange all the elements within the heap in an ascending (in the below case) or descending order and thus makes sorting much faster compared to some other sorting algorithms which would take linear, if not quadratic time (selection sort, insertion sort, bubble sort). In the worst case, heapsort has a time complexity of $O(nlg(n))$ as it might have to call heapify (rather the swaps) on each element in the heap.

Now that we have the heap data structure, we implement a priority queue using the minheap as an ADT. The priority queue (as seen below (pq_1.cpp)) inherits (discussed in part 7-object oriented programming) the minheap constructors so that implementing heap methods can be easier for the priority queue. Minimum priority queues have four major methods that make the priority queue relevant, efficient and effective for multiple data structure applications.

A min pq has the minimum operation that returns the smallest element in the structure. Because of the structure of a min heap, the smallest element is always at the head of the heap, making minimum() a constant time operation $O(1)$. It also has an extractMin() method that takes out the smallest element in the heap which is very useful for implementing queues (dequeue operation). For extractMin() we first obtain the root of the heap, then swap it with the last element of the heap and reduce the size of the heap, thereby discarding the minimum element.However, to retain it's heap structure, we need to call heapify on the head element which makes the run-time of extractMin $Olg(n)$. It also has decreaseKey() method that can update the value in a given node and systematically restores the heap property by performing swaps (simplified heapify's). This operation has a $O(lg(n))$ run-time in the worst case (when it has to perform swaps till the top of the heap).

With respect to the earlier seen linear data structures, a heap ADT improves the insert implementation significantly. The pq adds the new element at the bottom of the heap and then calls decreaseKey which swaps the new added node with the pre-existing nodes until the heap property is satisfied. Since it is so very similar to decreaseKey, insertion has a worst case run-time of $O(lg(n))$ making it exponentially better than linear data structures $O(n)$.

```
//=====================================================================
// Heap implementation of pointer arrays.
//=====================================================================

//=========================================
//default constructor
// pre-condition: N/A
// post-condition: empty array
//=========================================
template <class KeyType>
MinHeap<KeyType>::MinHeap (int n)
{
  A = new KeyType[n]; // creates new array of size n
  capacity = n;
  heapSize = 0;
}

//=========================================
// constructor
// pre-condition: N/A
// post-condition: Converted heap from array
//=========================================
template <class KeyType>
MinHeap<KeyType>::MinHeap(KeyType initA[], int n)
{
   A = new KeyType[n];
   capacity = n;
   for (int i = 0 ; i < n; i++) // deep copy
                A[i] = initA[i];
   buildHeap();
}

//=========================================
// copy constructor
// pre-condition: existing heap
// post-condition: new heap created from exisiting one.
//=========================================
template <class KeyType>
MinHeap<KeyType>::MinHeap(const MinHeap<KeyType>& heap)
{
   copy(heap);  // calls the copy method
}

//=========================================
// assignment constructor
// pre-condition: existing heap
// post-condition: new heap created by assignment to existing one
//=========================================
template <class KeyType>
MinHeap<KeyType>&MinHeap<KeyType>::operator=(const MinHeap<KeyType>& heap)
{
  if (&heap != this)
  {
    destroy();
    copy(heap);
  }
        return *this;
}

//=========================================
//destructor
// pre-condition: array
// post-condition: N/A (deleted array)
//=========================================
template <class KeyType>
MinHeap<KeyType>::~MinHeap()
{
  destroy();  // deletes the array A
}
```

```
//==========================================
// heapSort
// pre-condition: heapified heap
// post-condition: sorted heap
//==========================================
template <class KeyType>
void MinHeap<KeyType>::heapSort(KeyType sorted[])
{
  for (int i = capacity-1; i >= 1; i--)
  {
    swap(0,heapSize-1); // swaps the positions of elements at the given index
    heapSize -- ;
    heapify(0);
  }
  if (capacity > 2)
  {
        if (capacity%2 == 0) // for even capacities
         {
                for (int i = 0; i < (capacity+1)/2;i++)
                        swap(i,capacity-1-i); // reverses the order
         }
        else
         {
                for (int i = 0; i <= (capacity-1)/2;i++) // for odd capacities
                        swap(i,capacity-1-i);
         }
  }
  if (capacity == 2)
          swap(0,1);
 }
}
//==========================================
//heapify
// pre-condition: heap and array
// post-condition: min-heap
//==========================================
template <class KeyType>
void MinHeap<KeyType>::heapify(int i)
{
    int left, right, smallest;
    left = leftChild(i);
    right = rightChild(i);
    if (left <= heapSize && A[left] < A[i])
      smallest = left;
    else
      smallest = i;
    if (right <= heapSize && A[right] < A[smallest])
      smallest = right;

    if (i != smallest)
    {
      swap(smallest,i);
      heapify(smallest);
    }
}
//==========================================
//buildHeap
// pre-condition: array
// post-condition: heap (of the array)
//==========================================
template <class KeyType>
void MinHeap<KeyType>:: buildHeap()
{
    heapSize = capacity-1;
    for (int i = capacity/2; i >= 0; i--)
        heapify(i);
}
/*===========================================================================
=======*/
```

```
//==============================================================
// Priority queue implementation using a heap ADT
//==============================================================
#define DEFAULT_SIZE2 1000

//==========================================
// default constructor
//==========================================
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(void) : MinHeap<KeyType> (DEFAULT_SIZE2)
{}

//==========================================
// construct an empty MPQ with capacity n
//==========================================
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(int n) : MinHeap<KeyType> (n)
{}

//==========================================
// Minimum
// returns the minimum element
//==========================================
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::minimum(void) const
{
        if (heapSize <= 0)
                throw EmptyError();
        else
                return A[0];
}

//==========================================
// extractMin
// deletes the minimum element and returns it
//==========================================
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::extractMin(void)
{
        if (heapSize < 0)
           throw EmptyError();
        else
        {
                KeyType *min;
                min = A[0];
                swap(0,heapSize-1);
                heapSize--;
                heapify(0);
                return min;
        }
}

//==========================================
// decreaseKey
// decrease the value of an element
//==========================================
template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey(int index, KeyType* key)
{
        if (key <= A[index])
        {
                A[index] = key;
                while (index > 0 and *A[parent(index)] > *A[index])
                {
                        swap(index,parent(index));
                        index = parent(index);
                }
        }
}
```

```
//============================================
// insert
// inserts a new element (key)
//============================================
template <class KeyType>
void MinPriorityQueue<KeyType>::insert(KeyType* key)
{
  if(heapSize <capacity)
  {
    heapSize++;
          A[heapSize-1] = key;
        decreaseKey(heapSize-1, key);
        }
}
/*==========================================================================================
=======*/
```

GRAPHS

Competency: Proficiency

A graph in data structures is defined as a set of items/pairs of connected by edges. Each item is called a vertex or a node. Formally, a graph is a set of vertices and a binary relation between vertices called adjacency. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges. Graphs can be either directed or undirected. A directed graph is a graph, where all the edges are directed from one vertex to another. An undirected graph is a graph where all the edges are bidirectional. There are 2 main ways to represent a graph: Adjacency List and Adjacency Matrix.

An adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a vertex in the graph. For each vertex U, we store an array of the vertices that are directly adjacent to it. We typically have an array of n adjacency lists where n is the number of vertices in the graph.

An adjacency matrix, sometimes also called the connection matrix, of a simple labeled graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position depending on whether the combination of vertices are connected or not. The matrix can also have values other than 0 or 1 to indicate the weights.

A graph can be used for multiple functions and help us understand how traversals and searches can occur. Below are some of the types of algorithms that can be implemented for graphs.

BFS is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Given a graph G = (V, E) and a distinguished source vertex s, BFS systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. The algorithm works on both directed and undirected graphs. To keep track of progress, BFS colors each vertex white, gray, or black. All vertices start out white and may become gray and then black based upon how and when they are visited. When a vertex is discovered the first time it becomes non-white. Gray and black vertices, therefore, are the discovered ones but are distinguished between based on the searching procedure in a breadth-first manner.

DFS is another search algorithm that, instead of going "wider", goes "deeper" in the graph whenever possible. DFS explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. The algorithm can pick a new source in case all nodes are not visited from the given source v. The algorithm repeats this entire process until it has discovered every vertex. DFS also colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely.The below code ($graph_1$.cpp) shows the implentation of DFS in C++ using a minimum priority queue.

Dijkstra's algorithm is an example of a greedy algorithm which solves the single-source shortest-path problem when all edges have nonnegative weights. The algorithm starts at the source vertex s, it grows a tree T that ultimately spans all vertices reachable from S. Vertices are added to T in order to of distance, i.e., first S, then the vertex closes to S, then the next closest, and so on.

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. It has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. Each step adds to the tree a light edge that connects A to an isolated vertex âĂŞ one on which no edge of A is incident. This rule adds only edges that are safe for A. Thus, when the algorithm terminates, the edges in A form a minimum spanning tree. In order to implement PrimâĂŹs algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A. The below code shows an implementation of Prim's.

```
//====================================================================
// Graph implementation for Depth First Search, Prim's algorithm and
// cycles.
//====================================================================

//==========================================
// Vertex constructor
// pre-condition: N/A
// post-condition: N/A
//==========================================
Vertex::Vertex (int a)
{
    value = a;
    ID = a;
    pred = NULL;
    discovered = -1;
    finished = -1;
    adj = new Vertex *[10000];
    size = 0;
    color = "white";
}

//==========================================
// Edge constructor
// pre-condition: N/A
// post-condition: N/A
//==========================================
Edge::Edge(Vertex *initial, Vertex *destination, int w)
{
    start = initial;
    end = destination;
    weight = w;
}

//==========================================
// Graph constructor
// pre-condition: text file
// post-condition: N/A
//==========================================
Graph::Graph(string text)
{
    //reading in matrix-format file
    ifstream input;
    input.open(text);
    string line;
    getline(input,line);
    numberVertex = atoi(line.c_str());

    // Storing vertices
    numberEdge = 0;
    storeVertex = new Vertex*[numberVertex];   //charater array
    for (int j = 0; j < numberVertex; j++)
        storeVertex[j] = new Vertex(j);       //to find which vertex

    // Storing edges
    int maxnumEdge;
    maxnumEdge = numberVertex*(numberVertex-1)/2;
    storeEdge  = new Edge*[maxnumEdge];   //charater array

    // redacted code to read in and store the matrix file

    input.close();
}

//==========================================
// findWeight
// Returns the weight between two vertices
// Pre-condition: Two vertices, u and v
// Post-condition: N/A
//==========================================
int Graph::findWeight(int u, int v)
{
  int record;
  for (int i = 0; i < numberEdge; i++)
  {
    if (storeEdge[i]->start->ID == u)
    {
       if (storeEdge[i]->end ->ID == v)
            record = i;
    }
  }
  return (storeEdge[record]->weight);
}

//===========================================
// DFS
// Performs depth-first search
//===========================================
void Graph::DFS()
{
    for (int i = 0; i< numberVertex ;i++)
    {
        storeVertex[i]->color = "white";
        storeVertex[i]->pred = NULL;
    }
    time = 0;    // used to keep track of discovery and finish times
    for (int j = 0; j < numberVertex; j++)
    {
        if (storeVertex[j]->color == "white")
            dfsVisit(storeVertex[j]);
    }

    MinPriorityQueue<Vertex> Foundem;
    for (int i=0;i< numberVertex;i++)
    {
      Foundem.insert(storeVertex[i]);
    }

    cout << "The vertices that are visited are:" << endl;

    for (int i=0;i<numberVertex;i++)
      cout << *Foundem.extractMin() << endl;  // overloads cout for Vertex
}

//===========================================
// dfsVisit
// Recursive helper function for DFS
//===========================================
void Graph::dfsVisit(Vertex *u)
{
    u->color = "gray";
    time = time + 1;
    u->discovered = time;
    for (int i = 0; i < u->size;i++)
    {
        if (u->adj[i]->color == "white")
        {
            u->adj[i]->pred = u;
            dfsVisit(u->adj[i]);
        }
    }
    u->color = "black";
    time++;
    u->finished = time;
}

//===========================================
// cycleVisit
// Recursive helper function for cycle
//===========================================
void Graph::cycleVisit(Vertex *u,bool &found_cycle)
{
    if (found_cycle == true)
        return;
    u->color = "gray";
    for (int i = 0; i < u->size; i++)
    {
      if (u->adj[i]->color == "gray")
      {
        found_cycle = true;
        return;
      }
      if (u->adj[i]->color == "white")
          cycleVisit(u->adj[i],found_cycle);
    }
    u->color = "black";
}

//===========================================
// cycle
// Returns boolean indicating the Graph
// has a cycle or not
//===========================================
bool Graph::cycle()
{
  bool found_cycle = false;   //initialized to non-cyclic

  for (int i = 0; i < numberVertex;i++)
    storeVertex[i] ->color = "white";

  for (int j = 0; j < numberVertex; j++)
  {
    if (storeVertex[j]->color == "white")
        cycleVisit(storeVertex[j],found_cycle);
    if (found_cycle == true)
        break;
  }
  return found_cycle;
}

//===========================================
// Prim
// Function to implement prim's algorithm
//===========================================
void Graph::Prim(int r)
{
    primVertex = new Vertex*[numberVertex];
    for (int i = 0; i< numberVertex ;i++)
    {
        storeVertex[i]->value = 1000000; //infinity
        storeVertex[i]->pred = NULL;
    }

    for (int i = 0; i < numberVertex; i++)
    {
        if (storeVertex[i]->ID == r)
        {
            storeVertex[i]->value = 0;
            primVertex[0] = new Vertex(r);
        }
        lastone.insert(storeVertex[i]);
    }

    int tracc = 0;
    while (!lastone.empty())   //while pq is not empty
    {
        Vertex *choseu;
        choseu = lastone.extractMin();

        for (int mm = 0; mm < choseu->size; mm++)
        {
            if ((lastone.find(choseu->adj[mm])) and
              (findWeight(choseu->ID,choseu->adj[mm]->ID) < choseu->adj[mm]->value))
            {
                choseu->adj[mm]->pred = choseu;
                choseu->adj[mm]->value = findWeight(choseu->ID,choseu->adj[mm]->ID);
                tracc = tracc+1;
                primVertex[tracc] = new Vertex(choseu->adj[mm]->ID);

                primVertex[tracc]->pred = choseu;
            }
        }
    }

    int record;
    for (int i = 0; i < tracc+1 ; i++)
    {
        record = primVertex[i]->ID;
        for (int j = i+1; j < tracc+1;j++)
        {
            if (primVertex[j]->ID == primVertex[i]->ID)
                primVertex[i]->ID = -1;
        }
    }

    cout << "Route: " ;
    for (int i = 1; i < tracc+1;i++)
    {
        if (primVertex[i]->ID >= 0)
            cout  << primVertex[i]->pred->ID<<"->" << primVertex[i]->ID  << "  ";
    }
}
/*===========================================================================*/
```

HASH TABLES

Competency: Mastery with Distinction

A Simple Hash Map (Hash Table) can be implemented in C++ as a Hash table data structure that can map keys to values. A hash table is made up of two parts: an array (which can be implemented as a linked list) and a mapping function, known as a hash function. The hash function is a mapping from the input space to the integer space that defines the indices of the array (where we could 'hash' each value). A hash function provides a way for assigning numbers to the input data such that the data can then be stored at the array index corresponding to the assigned number. However, the hash function could very well hash multiple elements into the same index. It doesn't guarantee that every input will map to a different output. This phenomenon of having to insert multiple elements into the same place in an array is known as a collision. For the purposes of this class, we have two methods to deal with collisions: chaining and open addressing.

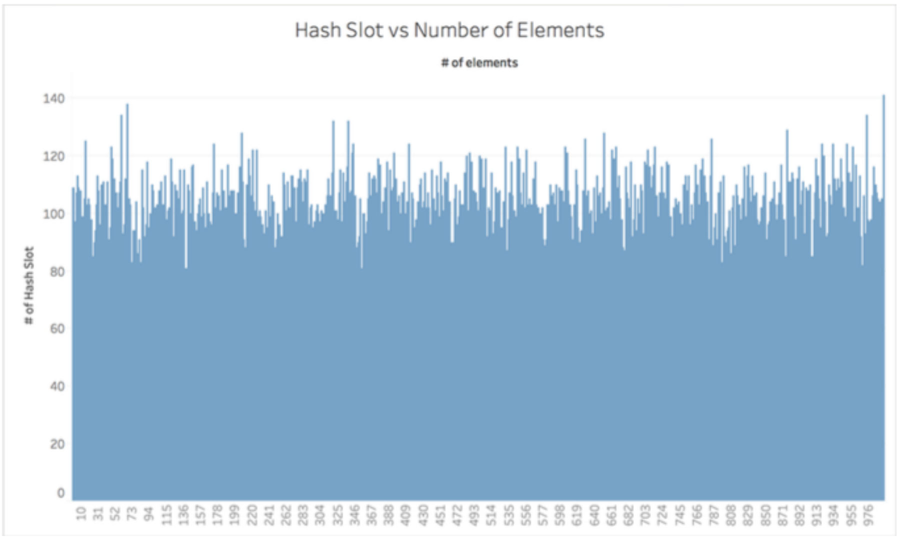
Chaining, instead of storing the data elements right into an array, stores the data in linked lists. Each slot in the array then points to one of these linked lists. When an element hashes to a value, it is added to the linked list at that index in the array. Because a linked list has no limit on length, collisions are no longer a problem. If more than one element hashes to the same value, then both are stored in that linked list.

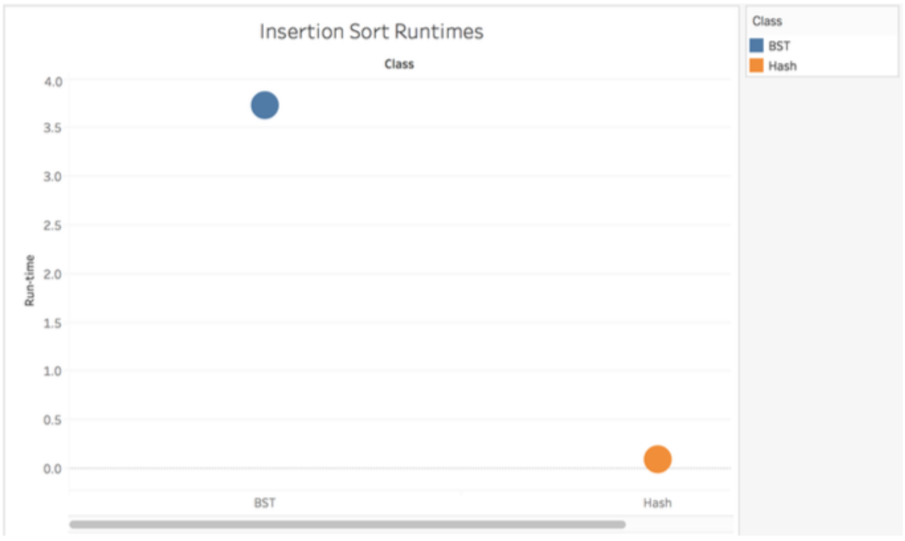
In open addressing, all elements occupy the hash table itself. Each table entry contains either an element of the dynamic set or NIL. In open addressing, the hash table can fill up so that no further insertions can be made. The benefit of open addressing is that it avoids pointers and instead computes the sequence of slots to be examined.

Moreover, there are multiple ways to design the hash functions that are used in the chaining process. A good hash function satisfies the assumption of simple uniform hashing; each key is equally likely to hash to any of the m slots,independent of other keys it can hash to. The two often used methods are the division and the multiplication methods wherein the division method determines the index based on a mod with a large number (ideally a large number close to a power of 2) while the multiplication method first multiplies the key by a constant within 0 and 1 and then, by doing a mod 1, extracts the fractional part. There is also the idea of universal hashing where slots are determined randomly, irrespective of the key value. The whole purpose of a hash function is to reduce value dependency and increase randomness in the hashing.

Moving on to the efficiency of hashing, both insertion and delete take constant time $O(1)$ as you only have to add elements based on the function to a linked list. The benefit of hashing, however, is in the searches, which, on average take $O(1 + n/m)$ where n is the number of elements and m is the number of slots. Once we know which slot the element is located in, we only have to look within that slot to find the element instead of the whole list.

As seen in the first graph below, for the purposes of my project, the hash function was relatively functional with a standard deviation of 12.6 in an insertion of over 10000 movie names. As the graph shows, the number of elements in each graph are fairly equal. The second graph is a better indication of the efficiency of the hash data structure. While a binary search tree (discussed in the next section) takes about 3.5 seconds to perform all the 10000 inserts, the hash table does the same in almost no time (literally).





```
//======================================================================
// Hash implementation of list
//======================================================================

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

//==========================================
// get method
// returns a pointer to the object in the hash table for a given key
//==========================================
template<class KeyType>
KeyType* HashTable<KeyType>::get( KeyType& k) const
{
  KeyType* track;
  int s = k.hash(slots);
  int i = 0;
  int remaincount = table[s].length();
  while ((*table[s][i] != k) && (remaincount !=0 ))
  {
      i++;
      remaincount--;
  }
  if (remaincount == 0)
    cout << "NOT FOUND" <<endl;
  else
    track = table[s][i];   //depends on list file
  return track;

}

//================================
// insert method
//=====================================
template<class KeyType>
void HashTable<KeyType>::insert(KeyType *k)
{
  int s = k->hash(slots);
  table[s].append(k);
}

//============================================
// remove method
//===========================================
template<class KeyType>
void HashTable<KeyType>::remove(KeyType& k)
{
  int remaincount = table[s].length();

  if (remaincount!=0)
    table[s].remove(k); // remove the first occurrence of the value item
}
/*====================================================================*/
```

BINARY SEARCH TREES

Competency: Mastery

Earlier we saw the heap data structure which was implemented as a tree structure. However, there are better, more efficiency tree-based data structures that can lead to better efficiencies and time complexities for certain operations (search, insert, delete etc). Two major kinds of tree data structures are Binary Search Trees (BSTs) and Red Black Trees (RBTs).

A binary tree is a data structure composed of parent nodes, or leaves, each of which stores data and also links to up to two other child nodes (leaves) which can be visualized spatially as below the first node with one placed to the left and with one placed to the right. This relationship between the leaves linked to and the linking leaf, also known as the parent node makes the binary tree an efficient data structure. By definition, the left child is the one with the lesser key value while the right child has an equal or greater key value. Since each node retains the property of the tree, it is possible to easily access and insert data in a binary tree using search and insert functions recursively called on successive leaves.

The code below (BST_5.h) shows the implementation of a BST in C++. Operations like insert, minimum, remove and maximum all take $O(lg(n))$ in terms of running time on average but can take $O(n)$ in the worst cases. To understand that we must see how insertion works in a binary tree. Starting from the root, when we insert a new value, we check the relative value of the new item and based on that the item is either compared to the left or the right subtree. Since for such comparisons, we always reduce the number of possible comparisons by 2 (assuming a balanced tree), making the time complexity $O(lg(n))$. The same logic applied to remove where if the node being removed does not has a left child, we swap the right subtree of the node with the node, else if the right child is empty, we swap the left child with the node (mind you, the left child is connected to the entire left subtree). If the node has both children, it finds the smallest value in the right subtree and swaps the value with it's right child. This restores the BST property. Since this also takes relative values into account, the time complexity is $O(lg(n))$. However, in a scenario where we insert an ascending ordered list on values, only the right subtree would ever get filled, cause the BST to practically be a linked list, causing the time complexity to rise to $O(n)$ in the worst case.

This is where Red Black Trees come into play. A red-black tree is a binary search tree in which each node is colored red or black such that it satisfies these properties:

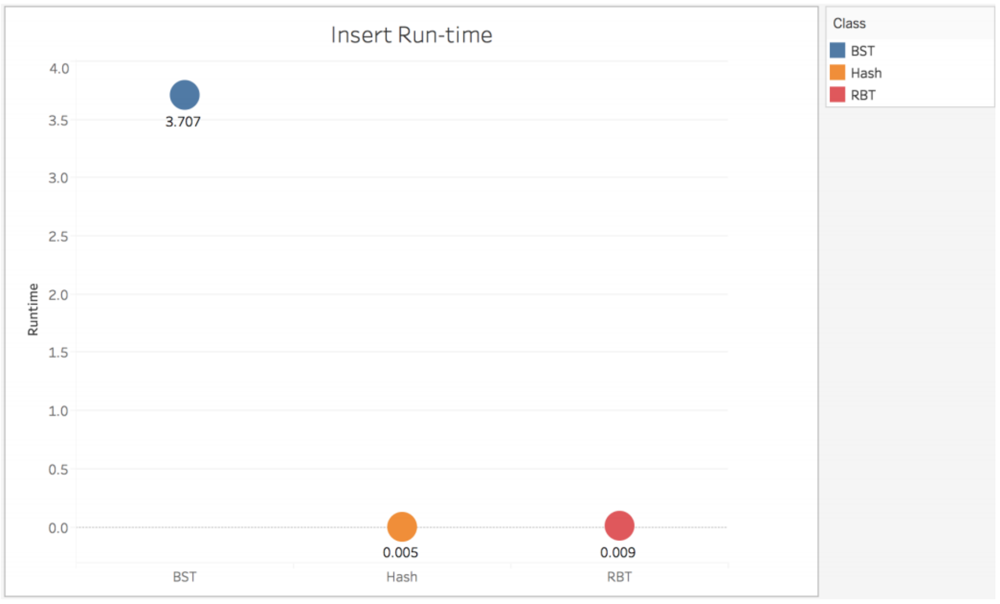
1. Every node is either red or black.
2. Every leaf is black.
3. A red node can only have black children.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

These properties ensure that the BST is always balanced, thereby getting rid of the extreme unbalancing that can result in the BST worst case. The below proof justifies why the height of a RBT is always balanced:

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

Proof. By property (5) of Red-Black Trees (RBTs), we know that for each node, each path from the node to a descendant leaf contains the same number of black nodes, which means they have the same black height. Thus, the number of black nodes for the longest path that starts at x should be same as the shortest path that starts at x. Therefore, the number of nodes of the shortest path will be equal to the black height. By definition, the maximum number of nodes of the longest path will be the sum of black height and max number of red nodes. By property (4) of RBTs, we know that if a node is red, both it's children are black. Therefore, the maximum number of red nodes should be equal to the number of black nodes (black height). Therefore, the longest path from x to descendant leaf will be $2*$black height, which is twice the length of the shortest path. □

As can be inferred from the below graph, for the same movies insertion, the RBT performs significantly better than a BST and is almost at par with the Hash table. Clearly, the balancing ability of RBT exponentially improves the time complexities of BST as all functions in RBTs perform in $O(lg(n))$ time.


Insert Run-time
Runtime
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
3.707
0.005
0.009
BST
Hash
RBT
Class
BST
Hash
RBT

```
//======================================================================
// BST data structure implementation
//======================================================================

template <class KeyType>
// node object to store the data
struct Node
{
    KeyType* data;
    Node *  left;
    Node *  right;
    Node *  parent;

    Node(KeyType* val)
    {
        this->data = val;
        this->left = NULL;
        this->right = NULL;
      this->parent = NULL;
    }
};

template <class KeyType>
class BST
{
        public:
                BST     ();
                ~BST    ();
                BST (const BST<KeyType> &bst);
    bool empty() const;//
                KeyType *get(const KeyType& k) const;
                void insert(KeyType *k);
                void remove(const KeyType& k);
                KeyType *maximum() const;
                KeyType *minimum() const;
                KeyType *successor(const KeyType& k) const;
                KeyType *predecessor(const KeyType& k) const;

                string inOrder() const;
                string preOrder() const;
                string postOrder() const;

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

        protected:
    Node<KeyType> *root;
                string traverseInOrder(Node<KeyType>*) const;
                string traversePreOrder(Node<KeyType>* ) const;
                string traversePostOrder(Node<KeyType>*) const;
                void  clear (Node<KeyType> *& );
                void  copy  (Node<KeyType> *&, Node<KeyType> * );
                void transplant(Node<KeyType>* u, Node<KeyType>* v);
    Node<KeyType>* minTree(Node<KeyType>* node) const;
    Node<KeyType>* maxTree(Node<KeyType>* node) const;
    Node<KeyType>* getNode(const KeyType& k) const;

};

//=========================================
// default constructor
//=========================================
template <class KeyType>
BST<KeyType>::BST ()
{
        root = NULL;
}

//=========================================
// copy constructor
//=========================================
template <class KeyType>
BST<KeyType>::BST (const BST<KeyType> &b)
{
    Node<KeyType> *ptr;
    copy(ptr, b.root); // calls a helper copy function
    root = ptr;
}

//===========================================
// insert
// inserts k into the BST
//===========================================
template <class KeyType>
void BST<KeyType>::insert(KeyType *k)
{
        Node<KeyType> *y = NULL;
        Node<KeyType> *x = root;
  Node<KeyType> *ptr = new Node<KeyType>(k);
        while (x != NULL)
        {
                y = x;
                if (*(ptr->data) < *(x->data))
                        x = x->left;
                else
                        x = x->right;
        }
  ptr->parent = y;
        if (y == NULL)
                root = ptr;
        else if (*(ptr->data) < *(y->data))
                y->left = ptr;
        else
                y->right = ptr;
}

//===========================================
// helper 'minTree' function
//===========================================
template <class KeyType>
Node<KeyType>* BST<KeyType>::minTree(Node<KeyType>* node) const
{
  if (node == NULL)
    return NULL;
  else
  {
    while (node->left != NULL)
      node = node->left;
    return node;
  }
}

//===========================================
// minimum
// returns min item
//===========================================
template <class KeyType>
KeyType* BST<KeyType>::minimum() const
{
  if (minTree(root) == NULL)
    return NULL;
  return minTree(root)->data;
}

//===========================================
// maximum
// returns max item
//===========================================
template <class KeyType>
KeyType* BST<KeyType>::maximum() const
{
  if (maxTree(root) == NULL) // calls a helper similar to minTree
    return NULL;
  return maxTree(root)->data;
}

//===========================================
// helper 'getNode' function
// pre-condition: existing BST
// post-condition: N/A
//===========================================
template<class KeyType>
Node<KeyType>* BST<KeyType>::getNode(const KeyType& k) const
{
  Node<KeyType> *ptr = root;
  bool search = false;
  while (ptr != NULL)
  {
    if (*ptr->data == k)
      {
        search = true;
        break;
      }
    else
    {
      if (*ptr->data < k)
          ptr = ptr->right;
      else
          ptr = ptr->left;
    }
  }
  if (search == false)
    return NULL;
  return ptr;
}

//===========================================
// get
// returns first item with key equal to k
// pre-condition: existing BST
// post-condition: BST
//===========================================
template<class KeyType>
KeyType* BST<KeyType>::get(const KeyType& k) const
{
  Node<KeyType> *find = getNode(k);
  if (find == NULL)
    return NULL;
  return find->data;
}

//===========================================
// successor
// returns the successor of k
// pre-condition: existing BST
// post-condition: BST
//===========================================
template <class KeyType>
KeyType* BST<KeyType>::successor(const KeyType& k) const
{
  Node<KeyType> *find = getNode(k);
  if (find == NULL)
    return NULL;
  else
  {
    if (*find->data == *maximum() || find == NULL)
      return NULL;
    if (find->right != NULL)
      return minTree(find->right)->data;

    Node<KeyType> *ptr = find->parent;
    while (ptr != NULL && find == ptr->right)
    {
      find = ptr;
      ptr = ptr->parent;
    }
    return ptr->data;
  }
}

//============================================
// predecessor
// returns the predecessor of k
// pre-condition: existing BST
// post-condition: BST
//============================================
template<class KeyType>
KeyType* BST<KeyType>::predecessor(const KeyType& k) const
{

  Node<KeyType> *find = getNode(k);
  if (find == NULL)
    return NULL;
  else
  {
    if (*find->data == *minimum())
      return NULL;
    if (find->left != NULL)
      return maxTree(find->left)->data;

    Node<KeyType> *ptr = find->parent;
    while (ptr != NULL && find == ptr->left)
    {
      find = ptr;
      ptr = ptr->parent;
    }
    return ptr->data;
  }
}

//============================================
// helper 'transplant' function
// pre-condition: existing BST
// post-condition: BST
//============================================
template<class KeyType>
void BST<KeyType>::transplant(Node<KeyType>* u, Node<KeyType>* v)
{
  if (u->parent == NULL)
    root = v;
  else if (u == u->parent->left)
    u->parent->left = v;
  else
    u->parent->right = v;
  if (v != NULL)
    v->parent = u->parent;
}

//============================================
// remove
// deletes first item with key equal to k
// pre-condition: existing BST
// post-condition: BST with removed k
//============================================
template<class KeyType>
void BST<KeyType>::remove(const KeyType& k)
{
  Node<KeyType> *del = getNode(k);
  if (del != NULL)
  {
    if (del->left == NULL)
```

```
      transplant(del, del->right);
    else if (del->right == NULL)
      transplant(del, del->left);
    else
    {
      Node<KeyType> *ptr = minTree(del->right);
      if (ptr->parent != del)
      {
        transplant(ptr, ptr->right);
        ptr->right = del->right;
        ptr->right->parent = ptr;
      }
      transplant(del, ptr);
      ptr->left = del->left;
      ptr->left->parent = ptr;
    }
  }
}
/*==========================================================================================
=======*/
```

```
//=====================================================================
// RedBlackTree implementation of Binary Search Trees
//=====================================================================

//==========================================
// default constructor
//==========================================
template <class KeyType>
  RedBlack<KeyType>::RedBlack()
{
  NIL = new Node<KeyType>();
  NIL->color = 'b'; // the color attribute
  root = NIL;
}

//==========================================
// leftRotate
//==========================================
template <class KeyType>
void RedBlack<KeyType>::leftRotate(Node<KeyType>* x)
{
  Node<KeyType>* y;
  y = x->right;
  if (x->parent!=NIL)
  {
    if (x == x->parent->left)
      x->parent->left = y;
    else
      x->parent->right=y;
  }
  else
    root = y;
  y->parent = x->parent;
  x->parent = y;
  x->right = y->left;
  y->left = x;
  x->right->parent = x;
}

//==========================================
// rightRotate
//==========================================
template <class KeyType>
void RedBlack<KeyType>::rightRotate(Node<KeyType>* x)
{
  Node<KeyType>* y;
  y = x->left;
  //x->parent = y;
  if (x->parent!=NIL)
  {
    if (x == x->parent->right)
      x->parent->right = y;
    else
      x->parent->left=y;
  }
  else
    root = y;
  y->parent = x->parent;
  x->parent = y;
  x->left = y->right;
  y->right = x;
  x->left->parent = x;

}

//==========================================
// insertHelp
//==========================================
template <class KeyType>
void RedBlack<KeyType>::insertHelp(Node<KeyType>* ptr)
```

```
{
  Node<KeyType>* y = NIL;
  Node<KeyType>* x = root;
  while (x != NIL)
  {
    y = x;
    if (*(ptr -> data) < *(x ->data))  //check if it is star
      x = x->left;
    else
      x = x->right;
  }
  ptr -> parent = y;
  if (y == NIL)
    root = ptr;   //tree was empty
  else if (*(ptr -> data) < *(y -> data))
    y ->left = ptr;    //left child
  else
    y ->right = ptr;  //right child
  ptr->left = NIL;
  ptr->right = NIL;
  ptr->color = 'r';
  insertFix(ptr);
}

//============================================
// insert
//============================================
template <class KeyType>
void RedBlack<KeyType>::insert(KeyType* k)
{
  Node<KeyType>* qtr = new Node<KeyType>();
  qtr->left = NIL;
  qtr->right= NIL;
  qtr->parent= NIL;
  qtr -> data = k;
  insertHelp(qtr);
}

//============================================
// get
//============================================
template <class KeyType>
KeyType* RedBlack<KeyType>::get(const KeyType& k) const
{
  Node<KeyType> *temp = getHelp(root,k);
  if(temp == NIL)
    return NIL->data ;
  return temp ->data;
}

//============================================
// removeFix
//============================================
template <class KeyType>
void RedBlack<KeyType>::removeFix(Node<KeyType>* x)
{
//  cout << "kakakka" <<endl;
  while ((x != root) and (x->color == 'b'))
  {
      if (x == x -> parent -> left)
      {
          Node<KeyType>* w;
          w = x-> parent->right;
          if (w->color == 'r')     //case1
          {
            w->color = 'b';
            x->parent->color='r';
            leftRotate(x->parent);
            w = x->parent -> right;
          }
          if ((w->left->color == 'b') and (w->right->color =='b'))  //case2
          {
            w->color = 'r';
            x = x->parent;
          }
          else
          {
            if (w->right->color == 'b')
            {
              w->left->color='b';
              w->color='r';
              rightRotate(w);
              w = x->parent->right;
            }
            w->color = x->parent->color;
            x->parent->color = 'b';
            w->right->color = 'b';
            leftRotate(x->parent);
            x = root;
          }
      }
      else
      {
        Node<KeyType>* w;
        w = x-> parent->left;
        if (w->color == 'r')     //case1
        {
          w->color = 'b';
          x->parent->color='r';
          rightRotate(x->parent);
          w = x->parent -> left;
        }
        if ((w->right->color == 'b') and (w->left->color =='b'))  //case2
        {
          w->color = 'r';
          x = x->parent;
        }
        else
        {
          if (w->left->color == 'b')
          {
            w->right->color='b';
            w->color='r';
            leftRotate(w);
            w = x->parent->left;
          }
          w->color = x->parent->color;
          x->parent->color = 'b';
          w->left->color = 'b';
          rightRotate(x->parent);
          x = root;
       }
     }
}
  x->color = 'b';
}

//===========================================
// remove
//===========================================
template <class KeyType>
void RedBlack<KeyType>::remove(const KeyType& k)
{
  Node<KeyType> *del = getHelp(root,k);
  Node<KeyType> *y = del;
  Node<KeyType> *x;
  char ccolor = y->color;

  if (del->left == NIL)
  {
     x = del->right;
     transplant(del, del->right); // same as the one called in BST.h
  }
  else if (del->right == NIL)
  {
     x = del->left;
     transplant(del, del->left);
  }
  else
   {
     y = minimumHelp(del->right);
     cout << *(y->data) <<endl;
     ccolor = y->color;
      x = y->right;

     if (y->parent == del)
     {
        x->parent = y;
      }
     else
     {
       transplant(y, y->right);
       y->right = del->right;
       y->right->parent = y;
     }

     transplant(del,y);
     y->left = del->left;
     y->left->parent = y;
     y->color = del->color;

    }
    if (ccolor == 'b')
      removeFix(x);
}

//===========================================
// successorHelp
//===========================================
template <class KeyType>
Node<KeyType>* RedBlack<KeyType>::successorHelp(Node<KeyType>* k)const
{
    Node<KeyType>* ptr;
    if (k->right != NIL)
      return minimumHelp(k->right);
    else
    {
      ptr = k->parent;
      while((ptr!=NIL) and (k == ptr->right))
      {
        k = ptr;
        ptr = ptr->parent;
      }
      return ptr;
    }
}

//===========================================
// successor
//===========================================
template <class KeyType>
KeyType* RedBlack<KeyType>:: successor(const KeyType& k) const
{
  if (empty())
    throw "Empty!";
  else
    return successorHelp(getHelp(root,k))->data;
}

//===========================================
// predecessorHelp
//===========================================
template <class KeyType>
Node<KeyType>* RedBlack<KeyType>::predecessorHelp(Node<KeyType>* k)const
{
  Node<KeyType>* ptr;
  if (k->left != NIL)
    return maximumHelp(k->left);
  else
  {
    ptr = k->parent;
    while((ptr!=NIL) and (k == ptr->left))
    {
      k = ptr;
      ptr = ptr->parent;
    }
    return ptr;
  }

}

//===========================================
// predecessor
//===========================================
template <class KeyType>
KeyType* RedBlack<KeyType>::predecessor(const KeyType& k) const
{
  return predecessorHelp(getHelp(root,k))->data;
}
/*=============================================================================*/
```

Object-oriented programming in C++

Competency: Mastery

C++ is primarily an object oriented successor to the C programming language. Object-oriented programming (OOP) refers to a type of computer programming in which we can not only define the data type of a data structure, but also the types of operations applicable to the data structure. A class in C++ is a user defined type or data structure declared with keyword class that has data and functions as its members whose access is governed by the three access specifiers: private, protected or public. The code for Binary Search Trees (BST_5.h) is an example where I used specific protected and public variables in order to effectively utilize the object oriented nature of C++. I also increasing created template classes for my projects and assignments so as to increase the scalability of my work. Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types wherein you can define the behavior of the class without actually knowing what datatype will be handled by the operations of the class.

Inheritance is also a major component of C++ (rather all OOPs). The capability of a class to derive properties and characteristics from another class is called inheritance. The code below demonstrates my ability to inherit one class from another. In the following code, I inherited the BST, RBT (in the code below) and Hash classes for the Movies class that made the the graph (shown in Binary Trees) possible since each movie insert worked differently for each data structure.

```
#include "redblack.h"

#ifndef DICTIONRBT_H
#define DICTIONRBT_H


template<class KeyType>
class dictionrbt : public RedBlack<KeyType>
{
        public:
                /*constructor, destructor and copy constructor
                inherited from BST<KeyType>*/
                //using RedBlack<KeyType>::root;
  //  using RedBlack<KeyType>:: NIL;

    using RedBlack<KeyType>::empty;
    using RedBlack<KeyType>::get;
    using RedBlack<KeyType>::remove;
    using RedBlack<KeyType>::insert;
                using RedBlack<KeyType>::inOrder;

};



#endif
```

Inheritance also reduces code redundancy and clustering which makes identifying issues and debugging a lot easier.

In OOP, polymorphism refers to the language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes. Simply put, it is the ability to use an operator or function in different ways so as to give different meaning or functions to the operators or functions. I achieved this in C++ using the overloading operators for certain class definitions, as seen in the code below. Not only did I use polymorphism for operations, but also used it for functions like the toString operator by overloading the cout stream.

```
//=============================================
//overload operators
//==============================================

bool movie::operator==(const movie &k) const
{
  return name == k.name;
}



bool movie::operator!=(const movie &k)const
{
  return name !=k.name;
}

bool movie::operator <(const movie &k)const
{
  return name < k.name;
}
bool movie::operator >(const movie &k)const
{
  return name > k.name;
}
```

Professional practice

Competency: Mastery with Distinction

CS 271 has been one of the most intense yet rewarding courses I have taken in my college life so far. The coursework involved a wide range of requirements: bi-weekly reading assignments, weekly projects along with the intermittent exams. Personally, I spent over two hours each day working on the class and the projects and believe performed relatively well in the class. While some of the initial projects (Huffman coding) were intellectually challenging and made me struggle throughout the weeks they were assigned, I believe I never let those variable affect my performance and quality of work. Over time, as I got more used to and versed with object oriented programming, I learned to code more effectively and efficiently and I think that showed in my work.

In terms of coding, I started off with minimal skill in terms of efficient coding practices and lacked attention to detail in terms of my testing and analysis. Over the course of the class, (hugely due to my partners) learned to add appropriate comments and thoroughly test my functions and methods before handing the work in.

The course also had a steep learning curve for me in terms of working effectively in groups. I must acknowledge Aixin, my consistent partner over the course, who helped me improve my ability to toggle between group and individual work even during a group project. I believe we made a good team and often contributed equally to most of the assignments and our work was always above average as our grades across the projects speak for themselves. Even when we were assigned different partners, I worked consistently and honestly with my partners.

I also believe I was an engaged and attentive student during class and tried to participate in the classroom discussions. I always made sure to clarify my doubts right away in class and diligently took notes throughout the semester.

Conclusion

Data structures have been a significant part of my life this semester. Having also taken CS-181 (Data Systems) and seeing the massive operational burdens that comes with big data, I have definitely developed a reverence for efficient and optimal data structures to store, process and analyze data. The projects throughout the semester were almost always challenging for me and I spent over 1-2 hours each day working through them in order to turn in quality work in a timely fashion. However, by the end of each project, I felt a heightened sense of accomplishment and had increased my understanding about the subject/topic.

I would specifically like to point out the Huffman coding project since I believe the project rigorously tested my limits, both in terms of intellectual capacity as well as mental tenacity. Like the other CS classes I have taken so far, CS 271 has helped me not only in terms of my Computer Science skills but has also taught me to be patient, persistent and attentive to details. Being an algorithms class, it was crucial to pay attention in the classroom as missing out on certain details in class directly affected my ability to transfer the algorithms to C++ implementation. This not only made me more attentive and focused but also encouraged me to indulge in diligent note taking and daily review of concepts. The reading notes were also a great resource to test my learning on a daily basis.

As mentioned throughout the portfolio, I made a few mistakes initially in the class but I believe I learned from them quick and overall performed well in the class. I am confident of my understanding of Algorithm analysis and thus gave myself a Mastery in competency. In terms of linear data structures, I believe I have a solid understanding of their fundamentals and understand their purpose and usage very well; thus I gave myself a Mastery with Distinction. The same applies to Heaps and Priority queues (especially after successfully attempting Huffman coding); I feel can safely give myself a Mastery with Distinction. Graphs have been my weakest area of learning so far and thus I gave myself a Proficiency competency in it. I will continue to work on improving my knowledge with regards to graphs in the future. For the hash tables project, we successfully managed to reduce the standard deviation to 12.6. Moreover I firmly understand the utilization of hash functions and thus gave myself a Mastery with Distinction. While I did very well on both the BST and RBT projects, there are a few areas of improvement for me and thus I gave myself a Mastery in them. In terms of my understanding of OOP in C++, I believe I gained a slow but increasing grasp in the utility of inheritance and polymorphism and thus gave myself a Mastery in it. Lastly, I believe I have been an excellent group partner and showed significant work ethic and dedication towards the class and thus gave myself a Mastery with Distinction. I hope my evaluation of my skill sets, combined with all the above evidence, resonates with you as well.