Portfolio

CS 271: Data Structures

Taylor Heilman

May 7th, 2016

Professional Practice

Mastery with Distinction

Professional practice and computer science go hand in hand. If you fail to properly communicate with your partner during a project you will inevitably encounter problems. If you don't manage your time properly you will end up staying up all night to finish a project the night before it is due. As a member of the varsity basketball team, computer science major and math minor time management was a necessary skill to master. Because of these things it became obvious to me that whenever I walk into Olin, I need to be prepared to work effectively and efficiently . If I did not spend my time wisely I would end up falling behind in the classroom and on the basketball court. That is why I always begin projects multiple days before they are due and try my hardest to not leave work for the night before it is due.

When it comes to group projects I have no problem working with other students. If my partner and I are first time partners we exchange contact information so we can schedule times to work on the project together. I understand that other people's schedules won't always align with mine, so sometimes I might have to work on the project alone while my partner is busy or vise versa.

Many of the topics I'm covering fall under the category of respect. Do you have the respect for yourself to wake up early to work extra on a project or will you simply accept a passing grade? Do you have the respect for your classmates to show up to class having read the assigned material allowing you to actively participate in class? Do

you have the respect for your professor to pay attention in class and take notes? I feel like I can honestly answer all of these questions with a ‘yes’.

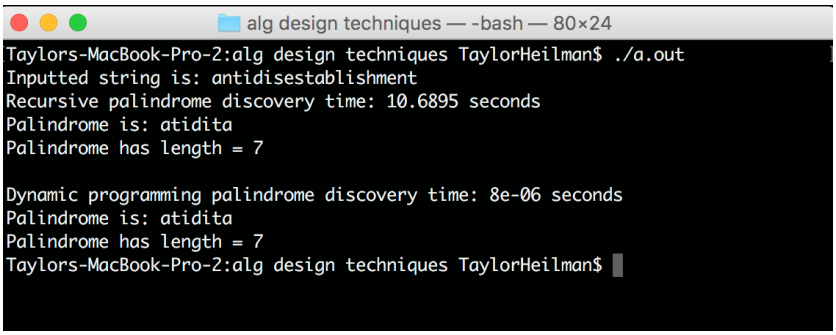
Lastly, my skills in LaTeX are far beyond standard. Between data structures and graph theory I had at least two typed up LaTeX files due per week, that’s around sixty multipage hand ins this semester using Latex. Including the work I did for proofs last semester I have completed over 100 projects using LaTeX.

All in all I believe my skills and ideology regarding professional practice are some of my most valuable qualities. A fews years down the road I think my leadership and communication skills will help me in the professional world rather than my coding skills.

Algorithm Design Techniques

Proficiency: Mastery

In **Project 1001** we were assigned the task of implementing an algorithm which found the longest palindrome subsequence in a string. For this project I showed the ability to not only design algorithms using divide & conquer techniques but also design algorithms using dynamic programming. To show the advantages of dynamic programming I implemented a recursive function to find the longest palindrome subsequence in a string. While this technique showed my ability to design recursive functions the actual function was severely inefficient. The recursive function computed the same problems multiple times, causing a poor run time. To improve efficiency I implemented a dynamic programming algorithm to compute longest palindrome subsequence. Unlike its recursive counterpart, the dynamic programming function used the problem's structure to its advantage. Instead of computing the same subproblems over and over again, the dynamic function used answers to smaller subproblems to formulate the answers to larger sub problems. This bottom up technique allowed for a far better runtime than the recursive solution.



When compared on a Mac Book Pro running OS X 10.11.4 the efficiency of the dynamic programming function was far superior than the recursive function. To compute the longest palindrome subsequence for a

twenty letter string it took the recursive function 10.6 seconds, while it took the dynamic programming function just 0.0000008 seconds. Attached is the write up from **Project 1001** showing how I developed the dynamic function using a recurrence that defines the value of an optimal solution as well as the code. From this recurrence I wrote pseudocode for the algorithm and finally I implemented the pseudocode for the final dynamic programming function.

In **Project 0110** I had to implement a binary search tree. Due to the structure of a binary search tree, the operations used in the class could be mostly implemented using recursion. Because of this I had to master the divide and conquer technique. When looking at the structure of a binary search tree it became obvious that you can find a desired answer by dividing the problem into smaller problems. An example of this is shown in the simple Minimum() function below. The Minimum() function is used to find the minimum key in the binary search tree. By definition of a binary search tree, all of keys in the left subtree of 'x' must be less than or equal to the key of 'x', hence the minimum key in the binary search tree is found by traversing left from the root until a node with no left child is found. This is easily done with recursion. We start at the root of the tree and check if the current node has a left child. This is our divide step. We divide the overall problem of finding the left

```
template <class KeyType>
KeyType* BinarySearchTree<KeyType>::minimum()
{
    return minimum_helper(root);
}


template <class KeyType>
KeyType *BinarySearchTree<KeyType>::minimum_helper(node* root)
{
    if(root == NULL)
    {
        cout << "empty tree" << endl;
        return NULL;
    }

    else if(root->left)
        return minimum_helper(root->left);

    else
        return root->key;
}
```

most element in the binary search tree into the smaller subproblem of whether a node has a left child or not. Using the outcomes of these smaller subproblems we eventually conquer the overall problem. If the current node has a left child we recursively call Minimum() on the left child. This step is repeated until we reach a node with no left child and return the current node. Attached are other recursive functions used to implement other functions in the binary search tree class.

Name: Taylor Heilman and Kevin Benson

Mon. May 2

CS 271: Project 1001

1. a.) The structure of an optimal solution to $f(m, n)$ would be a palindromic string of max length.

   b.) $f(m, n)$ represents the length of the longest palindrome subsequence from m^{th} character to the n^{th} character

$$f(m,n) \begin{cases} 1, & m = n \\ f(m+1, n-1) + 2, & a_m = a_n \\ max[f(m+1, n), f(m, n-1)] & else \end{cases}$$

   c.)

   Create a square array with dimensions ixj where both i and j are equal to the length of the input string.

   Make the diagonal indices = 1 (since a string of length 1 is a palindrome of itself) and all other indices = 0.

   Iterate through all columns and examine every item $c[i][j]$ above the diagonal. If the i^{th} and j^{th} characters in the input string are identical, the solution to subproblem f(i,j) is two greater than the solution to subproblem f(i+1, j-1), which is stored at index [i-1][j+1].

   If this is not the case, the value of the solution is the greater of either the item to the left or the item below the item in question.

   Our c++ implementation of this algorithm is provided below.

```
//==========================================================================
// Kevin Benson and Taylor Heilman
// May 2, 2016
// dynamic.cpp
//
// This file contains both the recursive and dynamically-programmed solutions
// to problem 15.2 of the textbook.
//==========================================================================
#include <string.h>
#include <iostream>
#include<stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <sys/time.h>
using namespace std;
//=======================================
// reverse helper function
//=======================================
string reverse(string s)
{
        int n = s.length();
        if (n == 0)
                return string("");
        return s[n - 1] + reverse(s.substr(0, n - 1));
}
//=======================================
// max helper function
//=======================================
int max (int x, int y) {
        if(x > y)
                return x;
        else
                return y;
}
//=======================================
// Recursive palindrome function
// Returns the length of the longest palindromic subsequence in s
//=======================================
string palindrome(string s, int i, int j){
        if(i==j){
                string c;
                c += s[i];
                return c;
        }
        if(i+1 == j and s[i] == s[j]){
                string  toreturn;
                toreturn = toreturn + s[i];
                toreturn = toreturn + s[j];
                return toreturn;
        }
        if(s[i] == s[j]){
                string toreturn;
                toreturn = toreturn + s[i];
                toreturn = toreturn + palindrome(s, i+1, j-1);
                toreturn = toreturn + s[j];
                return toreturn;
        }

        // Cannot use max to compare strings based on length, so do it the long way
        if((palindrome(s, i+1, j).length()) > (palindrome(s, i, j-1).length()))
                return palindrome(s, i+1, j);
        else
```

```
                return palindrome(s, i, j-1);
}

//========================================
// Dynamic Programming palindrome function
// Returns the length of the longest palindromic subsequence in s
//========================================
string dynamicPalindrome(string str)
{
        int l = str.length();
        int c[l][l];      // value table, which is square as we're funcitonally comparing two
identical strings
        // *** INITIALIZE THE TABLES ***

        for (int i = 0; i < l; i++){
                for (int j = 0; j < l; j++){
                        c[i][j] = 0;
                }
        }
        // *** FILL DIAGONAL OF TABLE WITH ONES ***
        for (int i = 0; i < l; i++){
                c[i][i] = 1;
        }
        for (int i = 1; i < l; i++)
        {
                for (int j = i-1; j >=0; j--)
                {
                        if (str[i] == str[j]){
                                c[i][j] = c[i-1][j+1] + 2;
                        }
                        else
                        {
                                c[i][j] = max(c[i-1][j], c[i][j+1]);
                        }
                }
        }

        int i = l-1;
        int j = 0;

        // Solution to subproblem f(m,n) is at index [m][m-n], so our final
        // solution is at the top right of our table
        int palLength = c[l-1][0];

        // Backtrace our array to find the palindrome itself
        string palindrome;
        while(c[i][j] != 0){
                if(c[i][j] != c[i-1][j] and c[i][j] != c[i][j+1]){
                        palindrome = palindrome + str[i];
                        i--;
                        j++;
                }
                else if (c[i][j] == c[i-1][j])
                        i--;
                else
                        j++;
        }
        // Reconstruct palindrome based on the parity of its length
        if(palLength % 2 == 0){
                palindrome = palindrome + reverse(palindrome);
        }
        else{
                for(int i = palindrome.length()-2; i >=0; i--){
```

```
                        palindrome += palindrome[i];
                }
        }

        return palindrome;
}

int main(){
        string q = "antidisestablishment";
        int i = q.length();

        // Recursive solution time analysis
        long diffSeconds, diffUSeconds;
        timeval timeBefore, timeAfter;
        gettimeofday(&timeBefore, NULL);

        string pal = palindrome(q, 0, i-1);

        gettimeofday(&timeAfter, NULL);
        diffSeconds = timeAfter.tv_sec - timeBefore.tv_sec;
        diffUSeconds = timeAfter.tv_usec - timeBefore.tv_usec;

        cout << "Recursive palindrome discovery time: " << diffSeconds + diffUSeconds/1000000.
0 << " seconds" << endl;
        cout << "palindrome is: " << pal << endl;
        cout << "palindrome has length = " << pal.length() << endl << endl;

        // Dynamic solution time analysis
        gettimeofday(&timeBefore, NULL);

        pal = dynamicPalindrome(q);

        gettimeofday(&timeAfter, NULL);

        diffSeconds = timeAfter.tv_sec - timeBefore.tv_sec;
        diffUSeconds = timeAfter.tv_usec - timeBefore.tv_usec;

        cout << "dynamic programming palindrome discovery time: " << diffSeconds + diffUSecond
s/1000000.0 << " seconds" << endl;
        cout << "palindrome is: " << pal << endl;
        cout << "palindrome has length = " << pal.length() << endl;

        return 0;
}
```

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include "bst.h"
using namespace std;



//========================================================================
// Insert()
// Use Recursion to Properly Insert an Item into the BST
//========================================================================

template <class KeyType>
void BinarySearchTree<KeyType>::insert(KeyType *x)
{
        if (root)
        {
                return insert_helper(root, x);
        }


        root = new node<KeyType>(x);
}


template <class KeyType>
void BinarySearchTree<KeyType>::insert_helper(*root, KeyType* key)
{
        //create new node
        if (root->key >= key)
        {

                if (root->left)
                        return insert_helper(root->left,key);

                else
                        root->left = new node<KeyType>(key);

        }

        else
        {
                if(root->right)
                {
                        return insert_helper(root->right, key);
                }

                else
                {
                        root->right = new node<KeyType>(key);
                }
        }
}

//========================================================================
// Get()
// Use Recursion to Properly Return First Element with Key Equal to k
//========================================================================

template <class KeyType>
KeyType *BinarySearchTree<KeyType>::get(const KeyType& x)
{
        return search_helper(root,key);
}
```

```
template <class KeyType>
KeyType *BinarySearchTree<KeyType>::get_helper(node* root, const KeyType& key)
{
        if(root->key == key)
                return root;

        else
        {
                if(root->key > key)
                {
                        root = root->right;

                        if(root == NULL)
                        {
                                cout << "not in tree" << endl;
                                return -1;
                        }

                        else
                                return get_helper(root,key);
                }

                else
                {
                        root = root->left;
                        if(root == NULL)
                        {
                                cout << "not in tree" << endl;
                                return -1;
                        }

                        else
                                return get_helper(root,key);

                }
        }
}



//====================================================================
// Minimum()
// Use Recursion to Properly Return the Minimum Element in the BST
//====================================================================

template <class KeyType>
KeyType* BinarySearchTree<KeyType>::minimum()
{
        return minimum_helper(root);
}


template <class KeyType>
KeyType *BinarySearchTree<KeyType>::minimum_helper(node* root)
{
        if(root == NULL)
        {
                cout << "empty tree" << endl;
                return NULL;
        }

        else if(root->left)
                return minimum_helper(root->left);

        else
                return root->key;
}

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

```
// Maximum()
// Use Recursion to Properly Return the Maximum Element in the BST
//=========================================================================

template <class KeyType>
KeyType* BinarySearchTree<KeyType>::maximum()
{
        return maximum_helper(root);
}


template <class KeyType>
KeyType *BinarySearchTree<KeyType>::maximum_helper(node* root)
{
        if(root == NULL)
        {
                cout << "empty tree" << endl;
                return NULL;
        }

        else if(root->right)
                return maximum_helper(root.right);
        else
                return root->key;
}


//=========================================================================
// inOrder()
// Use Recursion to Properly Return String of Elements From an Inorder Traversal
//=========================================================================


template <class KeyType>
void BinarySearchTree<KeyType>::inOrder()
{
        return inOrder_helper(root);
}


template <class KeyType>
void BinarySearchTree<KeyType>::inOrder_helper(node* root)
{
        if( root == NULL)
        {
                cout << "empty tree" << endl;
                return -1;
        }

        inOrder_helper(root->left);
        cout << root << key;
        inOrder_helper(root->right);
}




//=========================================================================
// preOrder()
// Use Recursion to Properly Return String of Elements From an Preorder Traversal
//=========================================================================



template <class KeyType>
void BinarySearchTree<KeyType>::preOrder()
{
        return preOrder_helper(root);
}
```

```
template <class KeyType>
void BinarySearchTree<KeyType>::preOrder_helper(node* root)
{
        if( root == NULL)
        {
                cout << "empty tree" << endl;
                return -1;
        }

        cout << root << key;
        preOrder_helper(root->left);
        preOrder_helper(root->right);
}


//=====================================================================
// inOrder()
// Use Recursion to Properly Return String of Elements From an Postorder Traversal
//=====================================================================


template <class KeyType>
void BinarySearchTree<KeyType>::postOrder()
{
        return postOrder_helper(root);
}


template <class KeyType>
void BinarySearchTree<KeyType>::postOrder_helper(node* root)
{
        if( root == NULL)
        {
                cout << "empty tree" << endl;
                return -1;
        }

        postOrder_helper(root->left);
        postOrder_helper(root->right);
        cout << root << key;
}


//=====================================================================
// Predecessor()
// Use Recursion to Properly Return the Predecessor of x
//=====================================================================

template <class KeyType>
KeyType* BinarySearchTree<KeyType>::predecessor(const KeyType& x)
{
        predecessor_helper(x, root);
}


template <class KeyType>
KeyType* BinarySearchTree<KeyType>::predecessor_helper(const KeyType& x, node* root)
{
        if (x->left)
                return maximum_helper(x->left);

        else
        {
                struct node *p = x->parent;
                while( p != NULL && x == p->left)
                {
                        x = p;
                        p = p->parent;
                }
```

```
                return p;

        }
}

//========================================================================
// Successor()
// Use Recursion to Properly Return the Successor of x
//========================================================================


template <class KeyType>
KeyType* BinarySearchTree<KeyType>::successor(const KeyType& x)
{
        successor_helper(x, root);
}


template <class KeyType>
KeyType* BinarySearchTree<KeyType>::successor_helper(const KeyType& x, node* root)
{
        if (x->right)           //x has a right child
                return minimum_helper(x->right);

        else                                    // successor is above x in tree
        {
                struct node *p = x->parent;
                while( p != NULL && x == p->right)
                {
                        x = p;
                        p = p->parent;
                }

                return p;
        }
}
```

Graphs

Proficiency: Mastery with Distinction

After this semester I feel as though my knowledge of graphs, their implementations and usages is one of my strongest subjects. Due to the fact that I was concurrently enrolled in Data Structures as well as Graph Theory I was able to get much practice implementing graphs and algorithms used on graphs. The two courses approached graphs with different ideology, however. Graph Theory focused mainly on traits of graphs and what makes them unique, while Data Structures focused mainly on designing an efficient way to implement graphs and graphical algorithms. In **Project 0111** we were assigned the task of implementing a weighted, directed graph. When implementing a graph class you have two main ways to represent a graph, a collection of adjacency lists or an adjacency matrix. Each representation has its advantages and disadvantages. An adjacency list offers a faster runtime for DFS and BFS compared to an adjacency matrix because of how the list stores data. An adjacency list is efficient with the data it stores in the fact that only edges found in the graph are stored in the list. An adjacency matrix on the other hand stores every possible connection, with a nonzero number denoting the connection with a weight represented by the number, and the number 0 denoting the connection to not exist. Because of these different representations, the traversal time of each implementation varies. To traverse an adjacency list you will look at each vertex in the graph and each edge of the graph. Thus the runtime of the list totals to O(n+m), where n is the number of vertices and m is the total number of edges. To traverse an adjacency matrix you look at n * n elements,

since you will look at every row and column in the matrix. The adjacency matrix has dimensions of n * n, hence the runtime for the matrix totals to $O(n^2)$.

I was able to implement a graph class in both C++ and Python this semester. I also implemented Depth First Search in both Python and C++ (see attached). a Depth First Search is used to find a minimum spanning tree in a connected graph. A minimum spanning tree of graph G is a subgraph of G that is a tree and has minimum weighted edges. A minimum spanning tree has many applications such as network design and finding the most efficient routes (traveling salesman problem). Breadth First Search accomplishes the same outcome as DFS but in a different fashion. While both are greedy algorithms, DFS spans out as far as it can and once it can travel no further it back traces and checks to see if it can find new nodes to travel to. A BFS, unlike DFS, travels to all adjacent nodes first and then spreads out. Visually a BFS looks more like an expanding circle, while DFS looks like many lines sprouting out of a root. Attached are implementations of BFS and DFS in Python using adjacency lists. Kruskal's algorithm is another graph algorithm which finds a a minimum spanning tree. The main difference between Kruskal's algorithm and DFS/BFS is the fact that BFS and DFS stay connected throughout their steps, while Kruskal's algorithm simply picks the edge of minimum weight in a graph that doesn't create a cycle. All in all, I feel like my knowledge of graphs is in the category of Mastery with Distinction due to the fact that I was able to implement graphs and their algorithms in multiple ways. I was also able to study graphs in different ways, allowing me to understand of graphs from multiple perspectives.

Algorithm 8.1 (Breadth-First Search)

1. Let $S = \{v\}$ [v is the root]; $T = \emptyset$ [initially there are no edges in the tree].
2. $C = N(v)$ [C is the current set of vertices being processed].
3. $l(v) = 0$ [label root as vertex 0]; $p(v) = v$; $b^* = v$ [keeps track of current vertex we are branching from]; $i = 1$ [initializes variable i used to help label the vertices]; remove b^* from all adjacency lists.
4. For each $w \in N(b^*)$, place w in S and place edge $b{*}w \in T$; assign successive labels $l(w) = i$ and $p(w) = p(b^*), w$ to vertices of C. Add one to i after each vertex is labeled; remove w from all adjacency lists [this ensures that a vertex w gets labels $l(w)$ and $p(w)$ just once].
5. Define a new b^* to be the vertex x in C such that $l(x)$ is minimum; remove b^* from C, and return to step 4. If, however, C is empty, stop. If every vertex of G has been labeled, a spanning tree has been found. If not, then G is disconnected, but a spanning tree of the component containing the root has been found.

```
#Breadth First Search
def BFS(self,v):
    if v in self.Graph:
        #step 1
        graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
        S = [v]                            #list S
        label = []                          #list to hold labels, index refers to label
        T = []                               #empty list of edges
        #step 2
        C = graph[v]            #list of v's neighbors
        C.sort()
        #step 3
        label.append(v)  #label v as 0
        bstar = v        #bstar = vertex we're branching to

        for item in graph:
            if bstar in graph[item]:          #remove bstar from adjacency lists
                graph[item].remove(bstar)

        #step 4
        while(True):
            neighbors = graph[bstar]        #list of adjacent vertices

            for vert in label:
                if vert in neighbors:       #neighbors = adjacent vertices that haven't been visited
                    neighbors.remove(vert)

            for item in neighbors:          #iterate through neighbors
                S.append(item)          #add vertex to S
                S.sort()
                T.append((bstar,item))  #add edge to T
                label.append(item)      #label item

            graph2 = copy.deepcopy(graph)       #create copy of graph bc can't edit something you're iterating

            for thing in C:
                for item in graph:
                    if thing in graph2[item]:       #remove bstar from adjacency lists
                        graph2[item].remove(thing)


            graph = graph2          #update graph

            #step 5
            if (len(C) == 0):           #halting condition
                return T

            else:
                bstar = C[0]            # define a new bstar to be min vertex in C
                C.remove(bstar)
```

Algorithm 8.2 (Depth-First Search)

1. Let $S = \{v\}$ [v is the root], $T = \emptyset$ [initially there are no edges in the tree]; $b^* = v$ [our initial branch vertex]; $p^* = v$ $l(v) = 0$ [label root as vertex 0]; $i = 1$ [initializes variable i used to help label the vertices]; $U = V(G) - \{v\}$ [keeps track of unlabeled vertices].
2. While $N(b^*) \cap U \neq \emptyset$ [b^* has unlabeled neighbors] do
   begin
   label the next unlabeled neighbor w of b^* by i; place b^*w in T; remove w from U; let $p^* = b^*$ [helps in backtracking]; $b^* = w$ [new branch vertex]; $i = i + 1$ [increment i for future labeling].
   end.
3. $b^* = p^*$[backtrack].
4. If $b^* = v$ and $N(b^*) \cap U = \emptyset$ [v has no unlabeled neighbors], stop. A spanning tree for the component containing the root v has been found. Otherwise, repeat step 2. If $U = \emptyset$, then the tree found is a spanning tree of all of G. If $U \neq \emptyset$, then G is disconnected.

```
#Depth First Search
def DFS(self,v):
    if v in self.Graph:     #check that vertex v is in the graph
        #step 1
        graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
        T = []                                  #empty list of edges
        bstar = v                               #initial branch vertex
        pstar = v
        history = [v]                           #history of vertices that are used as bstar
        label = []                              #list to hold labels
        label.append(v)                         #label v as 0
        U1 = graph.keys()                       #keys of dict
        U = []                                  #list of unlabeled vertices

        for item in U1:
            U.append(item)
        U.remove(v)         #remove v since it is labeled
        U.sort()

        neighbors = graph[v]        #list of vertices adjacent to v
        neighbors.sort()
        intersection = list(set(U) & set(neighbors))        #U intersect neighbors
        intersection.sort()                                 #finds unlabeled niehgbors

        #step 2
        while (True):
            while (intersection):
                w = intersection[0]
                label.append(w)          #label neighbor of bstar
                T.append((bstar,w))      #add edge to T
                U.remove(w)              #remove labeled vertex from unlabeled list
                pstar = bstar            #help for backtracking
                bstar = w
                history.append(bstar)       #update history list
                neighbors = graph[bstar]    #get neighbors
                neighbors.sort()
                intersection = list(set(U) & set(neighbors))    #get intersect of U and neighbors
                intersection.sort()

            neighbors = graph[bstar]            #update neighbors
            neighbors.sort()
            intersection = list(set(U) & set(neighbors))     #used to check for halting condition
            intersection.sort()

            if (len(intersection) == 0):    #if U intersect neighbors = []
                num = history.index(bstar)
                bstar = history[num-1]      #bstar = pstar

            if (bstar == v and intersection == [] and len(U) == 0): #halting condition
                return T        #return spanning tree

            if (intersection != []):
                history.append(bstar)       #update history list
```

```
#Taylor Heilman
#an undirected simple graph
#Feb 11, 2016

import copy


class Graph(object):

    def __init__(self):
        #Graph constructor
        self.Graph = {}


    def add_vertices(self, vertices):
        #Add a list of vertices to the graph
        length = len(vertices)
        for i in range (length):
            if vertices[i] not in self.Graph:   #check if vertex exists in dictionary alr
eady
                self.Graph[vertices[i]] = []


    def delete_vertex(self, v):
        #Delete a vertex from the graph.
        if v in self.Graph:
            del self.Graph[v]           #delete key from dictionary
            for item in self.Graph:
                if v in self.Graph[item]:       #delete vertex from other keys' values
                    self.Graph[item].remove(v)




    def contract_edge(self, e):
        vertex1=e[0]        #first element of edge
        vertex2=e[1]        #second element of edge

        if vertex1 < vertex2:
            for item in (self.Graph[vertex2]):
                if item not in self.Graph[vertex1] and item != vertex1:
                    self.Graph[vertex1].append(item)    #copy vertex2's edges into vertex
1's

            for item in self.Graph:
                if vertex2 in self.Graph[item]:
                    self.Graph[item].remove(vertex2)    #remove vertex2 from other keys'
values

            del (self.Graph[vertex2])   #delete vertex2 from dictionary


        else:
            for item in (self.Graph[vertex1]):
                if item not in self.Graph[vertex2] and item != vertex2:
                    self.Graph[vertex2].append(item) #copy vertex1's edges into vertex2's

            for item in self.Graph:
                if vertex1 in self.Graph[item]:
                    self.Graph[item].remove(vertex1) #remove vertex1 from other keys' val
ues

            del (self.Graph[vertex1])  #delete vertex1 from dictionary




    def delete_edge(self, e):
        vertex1=e[0]        #first element of edge
        vertex2=e[1]        #second element of edge
```

```
        if vertex1 in self.Graph and vertex2 in self.Graph:
            if vertex1 in self.Graph[vertex2]:          #check if element1 is in element2
's values
                self.Graph[vertex2].remove(vertex1)     #remove element1 from values
if true
            if vertex2 in self.Graph[vertex1]:          #check if element2 is in element1
's values
                self.Graph[vertex1].remove(vertex2) #remove element2 from values if t
rue


    def vertices(self):
        #Return a list of nodes in the graph.
        return list(self.Graph.keys())

    def add_edges(self, edges):
        #Add a list of edges to the graph
        edges = list(edges)
        length1 = len(edges)
        for i in range (length1):
            temp = edges[i]
            first = temp[0]     #first vertex of pair
            second = temp[1]    #second vertex of pair

            if first not in self.Graph:         #Vertex1 is not in dictionary
                self.Graph[first] = [second]
            else:
                if second not in self.Graph[first]:     #Vertex1 in dictionary but Vertex
 2 isn't
                    self.Graph[first].append(second)

            if second not in self.Graph:        #Vertex2 is not in dictionary
                self.Graph[second] = [first]

            else:
                if first not in self.Graph[second]:     #Vertex2 is in dictionary but Ver
tex1 isn't
                    self.Graph[second].append(first)



    def edges(self):
        #Return a list of edges in the graph
        edge = []
        for vertex1 in self.Graph:
            for vertex2 in self.Graph[vertex1]:
                if (vertex2, vertex1) not in edge:      #check if inverse of edge is alre
ady in the list
                    edge.append((vertex1, vertex2))     #ex. if (u,v) is in list (v,u) wo
n't be appended


        return edge


    #Breadth First Search
    def BFS(self,v):
        if v in self.Graph:
            #step 1
            graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
            S = [v]                         #list S
            label = []                       #list to hold labels, index refers to label
            T = []                           #empty list of edges
            #step 2
            C = graph[v]            #list of v's neighbors
            C.sort()
            #step 3
            label.append(v)  #label v as 0
            bstar = v        #bstar = vertex we're branching to
```

```
            for item in graph:
                if bstar in graph[item]:          #remove bstar from adjacency lists
                    graph[item].remove(bstar)

            #step 4
            while(True):
                neighbors = graph[bstar]        #list of adjacent vertices

                for vert in label:
                    if vert in neighbors:       #neighbors = adjacent vertices that haven
't been visited
                        neighbors.remove(vert)

                for item in neighbors:          #iterate through neighbors
                    S.append(item)          #add vertex to S
                    S.sort()
                    T.append((bstar,item))  #add edge to T
                    label.append(item)      #label item

                graph2 = copy.deepcopy(graph)       #create copy of graph bc can't edit s
omething you're iterating

                for thing in C:
                    for item in graph:
                        if thing in graph2[item]:       #remove bstar from adjacency list
s
                            graph2[item].remove(thing)


                graph = graph2          #update graph

                #step 5
                if (len(C) == 0):           #halting condition
                    return T

                else:
                    bstar = C[0]            # define a new bstar to be min vertex in C
                    C.remove(bstar)




    #Depth First Search
    def DFS(self,v):
        if v in self.Graph:     #check that vertex v is in the graph
            #step 1
            graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
            T = []                                  #empty list of edges
            bstar = v                               #initial branch vertex
            pstar = v
            history = [v]                           #history of vertices that are used as
 bstar
            label = []                              #list to hold labels
            label.append(v)                         #label v as 0
            U1 = graph.keys()                       #keys of dict
            U = []                                  #list of unlabeled vertices

            for item in U1:
                U.append(item)
            U.remove(v)         #remove v since it is labeled
            U.sort()

            neighbors = graph[v]        #list of vertices adjacent to v
            neighbors.sort()
            intersection = list(set(U) & set(neighbors))        #U intersect neighbors
            intersection.sort()                                 #finds unlabeled niehgbor
s

            #step 2
```

```
            while (True):
                while (intersection):
                    w = intersection[0]
                    label.append(w)          #label neighbor of bstar
                    T.append((bstar,w))      #add edge to T
                    U.remove(w)              #remove labeled vertex from unlabeled list
                    pstar = bstar            #help for backtracking
                    bstar = w
                    history.append(bstar)       #update history list
                    neighbors = graph[bstar]    #get neighbors
                    neighbors.sort()
                    intersection = list(set(U) & set(neighbors))    #get intersect of U a
nd neighbors
                    intersection.sort()

                neighbors = graph[bstar]            #update neighbors
                neighbors.sort()
                intersection = list(set(U) & set(neighbors))     #used to check for halti
ng condition
                intersection.sort()

                if (len(intersection) == 0):    #if U intersect neighbors = []
                    num = history.index(bstar)
                    bstar = history[num-1]      #bstar = pstar

                if (bstar == v and intersection == [] and len(U) == 0): #halting conditio
n
                    return T        #return spanning tree

                if (intersection != []):
                    history.append(bstar)       #update history list




    def MaxDegC(self):
        order = list()      #list of vertices with degrees
        sort = list()       #sorted list of veritces based off of degrees
        colors = {}         #dictionary to keep track of color labels
        for index in range (len(self.Graph)):       #creates dictionary of colors for wor
st case scenario
            colors[index+1] = []                    #worst case = (every vertex has own c
olor)

        for vertex in self.Graph:
            order.append((len(self.Graph[vertex]), vertex))     #make list with (degree,
vertex)

        order.sort()        #sort in ascending order
        order.reverse()     #reverse order

        for index in range (len(order)):
            sort.append(order[index][1])        #remove degrees from list

        colors[1] = [sort[0]]             #color first vertex 1
        x = sort[0]
        sort.remove(x)                   #remove vertex


        while len(sort) > 0:               #while not all vertices are labeled
            j=1                             #set color = 1
            found = False
            w = sort[0]                         #set w = vertex with max degree
            while found == False:
```

```
                for label in colors:
                    if found == True:           #if a vertex can be labeled stop the for
loop
                        break
                    spot = 0        #keep track of # of neighbors checked
                    for neighbor in self.Graph[w]:
                        spot = spot+1  #checked one more neighbor
                        if neighbor in colors[label]:   #a neighbor vertex is already lab
eled with a num = label
                            j = 0
                            j = label+1                     #set j = one more than label
of neighboring vertex

                        elif spot == len(self.Graph[w]):        #all neighbors have been
checked
                            found = True
                            colors[j].append(w)     #label w with color j
                            x = sort[0]
                            sort.remove(x)      #remove from U




        ret = list()
        for i in colors:
            if (len(colors[i]) != 0):       #don't print out empy lists
                ret.append(colors[i])

        return ret



    def SeqC(self):
        colors = {}                             #dictionary to hold colors of vertices
        for index in range (len(self.Graph)):       #creates dictionary of colors for wor
st case scenario
            colors[index+1] = []                        #worst case = every vertex gets i
ts own color


        vertices = list()
        for vert in self.Graph:             #list of vertices in self.Graph
            vertices.append(vert)

        colors[1] = [vertices[0]]       #label first vertex 1

        labeled = list()                    #list of labeled vertices
        labeled.append(vertices[0])


        for vertex in self.Graph:           #iterate though all vertices in graph
            if vertex not in labeled:
                neighbors = self.Graph[vertex]      #list of adjacent vertices
                a = neighbors
                b = labeled                         #list of labeled vertices

                important = list(set(a) & set(b))   #list of labeled adjacent vertices

                if(len(important) == 0):        #no labeled adjacent vertices
                    colors[1].append(vertex)    #label vertex with 1

                else:
                    found = False
                    for label in colors:
                        if found == True:           #if a vertex can be labeled stop the
for loop
                            break
                        spot = 0        #keep track of # of neighbors checked
                        for neighbor in important:
                            spot = spot+1          #checked one more neighbor
```

```
                            if neighbor in colors[label]:   #a neighbor vertex is already
 labeled with a num = label
                                j = 0
                                j = label+1                     #set j = one more than la
bel of neighboring vertex

                            elif spot == len(important):        #all neighbors have been
checked
                                found = True
                                colors[j].append(vertex)     #label w with color j




        ret = list()
        for i in colors:
            if (len(colors[i]) != 0):       #don't print out empy lists
                ret.append(colors[i])

        return ret


    def isTree(self):       #modified DFS to check for cycles since we assume G is connec
ted
        keys = self.vertices()
        v = keys[0]             #arbitrarily pick a vertex for v
        #step 1
        graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
        S = [v]              #list
        T = []               #empty list of edges
        history = [v]        #history of vertices that are used as bstar
        bstar = v
        pstar = v
        label = []           #list to hold labels
        label.append(v)     #label v as 0
        U1 = graph.keys()   #keys of dict
        U = []              #list of unlabeled vertices
        for item in U1:
            U.append(item)
        U.remove(v)         #remove v since it is labeled
        U.sort()
        neighbors = graph[v]        #list of neighbors of v
        neighbors.sort()
        intersection = list(set(U) & set(neighbors))        #U intersect neighbors
        intersection.sort()
        #step 2
        while (True):
            while (intersection):
                w = intersection[0]
                label.append(w)          #label neighbor of bstar
                T.append((bstar,w))      #add edge to T
                U.remove(w)              #remove labeled vertex from unlabeled list
                pstar = bstar               #help for backtracking
                bstar = w
                history.append(bstar)       #update history list
                neighbors = graph[bstar]    #get niehgbors
                neighbors.sort()
                intersection = list(set(U) & set(neighbors)) #get intersect of U and neig
hbors
                intersection.sort()

        #MAIN EDIT
                #-------------------------------------------------------------------------
                cycle = list(set(history) & set(neighbors)) #take the intersection of the
 bstar's neighbors and history(visited vertices)
                if (len(cycle) >= 2):                    # IF 2 OR MORE NEIGHBORS OF CURR
ENT VERTEX HAVE BEEN VISITED
                    return (False)              #A CYCLE HAS BEEN FOUND, RETURN FALSE
                #-------------------------------------------------------------------------
```

```
            neighbors = graph[bstar]
            neighbors.sort()
            intersection = list(set(U) & set(neighbors))     #used to check for halting c
ondition
            intersection.sort()


            if (len(intersection) == 0):    #if U intersect neighbors = []
                num = history.index(bstar)
                bstar = history[num-1]      #bstar = pstar

            if (bstar == v and intersection == [] and len(U) == 0): #halting condition
                return True        #IF DFS COMPLETES NORMALLY, RETURN TRUE

            if (intersection != []):
                history.append(bstar)       #update history list




    def Center(self):
        if (self.isTree()):     #isTree returned True
            graph = copy.deepcopy(self.Graph)   #make a copy to delete vertices without c
hanging real graph
            leaves = []             #make a list of leaves found
            x = True
            while (x == True):
                for vertex in graph:       #iterate through vertices in graph
                    if (len(graph[vertex]) == 1):       #leaf found (has degree = 1)
                        leaves.append(vertex)           #get a list of leaves to delete


                for i in range (len(leaves)):   #iterate through leaves
                    del graph[leaves[i]]               #delete leaf from dict
                    for item in graph:
                        if leaves[i] in graph[item]:       #delete leaf from other keys'
values
                                graph[item].remove(leaves[i])


                leaves = []             #reset list for next iteration

                if (len(graph) <= 2):  #check if center/ centers have been found
                    for item in graph:
                        leaves.append(item) #reuse leaves list to return center/centers
                    return(leaves)

        else:
            return('Graph is not a Tree')           #isTree returned False



def main():
    #call functions from here
    G = Graph()


main()
```

```
// graph.h
// Graph header file
// Clay Sarafin & Taylor Heilman

#ifndef GRAPH_H
#define GRAPH_H

#include "list.h"
#include "ds.h"
#include "pq.h"



class Vertex{
public:
        Vertex();
        Vertex(int contents);                   //construct with value
        int data;                               //value of vertex
        char color;                             //w=white, g=grey

        Vertex* pred;                           //predecessor of node, for dfs()

        List<Vertex> connections;               //list of connections, for both dfs() and Krus
kal()
};


class Edge{
public:
        Edge();
        Edge(Vertex* vertexU, Vertex* vertexV, int w);

        Vertex* u;
        Vertex* v;

        DSNode<Vertex>* nodeU;
        DSNode<Vertex>* nodeV;

        int weight;


        bool operator<(const Edge& e){  //need this for selection sort
                return weight < e.weight;
        }
};

std::ostream& operator<< (std::ostream& os, const Edge& e){
        os << e.weight << "(" << e.u->data << "," << e.v->data << ")";
        return os;
}


class Graph{
public:
        Graph();
        Graph(std::string file);
        ~Graph();

        void dfs();
        void dfsVisit(Vertex * u);
        void Kruskal();
```

```
private:
        Vertex ** vertices;
        Edge ** edges;
        int capacity;
        int capacityEdge;
        int lengthEdge;

        void dealloc();
        //disallow copying
        Graph(const Graph& g) {};
        Graph& operator=(const Graph& g) {};
};

#endif

#include "graph.cpp"
```

```
// graph.cpp
// Graph class code
// Clay Sarafin & Taylor Heilman

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


/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                     Edge - CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/

/*----------------------------------------------
 * Default Constructor
 * PreConditions:       n/a
 * PostConditions:      empty Edge object created
---------------------------------------------*/
Edge::Edge(){
        weight = 0;
}


/*----------------------------------------------
 * Construct with all properties of the edge
 * PreConditions:       pointers to both vertices on the edge, and its weight
 * PostConditions:      Edge object with above properties will be created
---------------------------------------------*/
Edge::Edge(Vertex* vertexU, Vertex* vertexV, int w){
        u = vertexU;
        v = vertexV;
        weight = w;


}


/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                     Vertex - CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*----------------------------------------------
 * Default Constructor
 * PreConditions:       n/a
 * PostConditions:      "empty" Vertex object created
                        color 'w' denotes it has not been visited in dfs()
---------------------------------------------*/
Vertex::Vertex(){
        color = 'w';
        pred = NULL;
}

/*----------------------------------------------
 * Construct with contents
 * PreConditions:       integer denoting its contents
 * PostConditions:      creates vertex object with appropriate contents
---------------------------------------------*/
Vertex::Vertex(int contents){
        data = contents;
        color = 'w';
        pred = NULL;
}
```

```
/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                     Graph - (DE)CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*-----------------------------------------------
 * Default Constructor
 * PreConditions:       n/a
 * PostConditions:      creates empty Graph object
                        nothing dynamically allocated
----------------------------------------------*/
Graph::Graph(){
        capacity = 0;
        capacityEdge = 0;
        lengthEdge = 0;
}

/*-----------------------------------------------
 * Construct with file
 * PreConditions:       a string denoting the name of the text file
                        file has to be in the format of:
                                (number of vertices)
                                (adjacency matrix of wieghts)
                        ex:
                        5
                        0 1 2 0 0
                        1 0 1 1 0
                        2 1 0 0 3
                        0 1 0 0 1
                        0 0 3 1 0
 * PostConditions:      creates a graph based on the relationships defined in the text file
----------------------------------------------*/
Graph::Graph(std::string file){
        std::ifstream text(file, std::ifstream::in);
        std::string line;
        std::string str;
        std::string token;

        //read 1st line for capacity
        std::getline(text, str);
        capacity = stoi(str);
        vertices = new Vertex*[capacity];

        for (int i=0; i<capacity; i++)                  //never forget that you need to initia
lize every object in an array of pointers
                vertices[i] = new Vertex(i);
        capacityEdge = (capacity*capacity)-capacity;
        edges = new Edge*[capacityEdge];
        //matrix has total elements capacity^2,
        //nodes cannot be connected to itself (hence the diagonal of 0's) so capacity is also
subtracted

        lengthEdge = 0;

        //read rest of the file
        //read by line, then by character
        for (int i=0; i<capacity; i++){
                std::getline(text,line);
                std::istringstream iss(line);
                for (int j=0; j<capacity; j++){
                        std::getline(iss,token, ' ');
                        int weight = stoi(token);
                        if (weight != 0){
                                //create new edge based off of the value being read in, and th
```

```
e iterations it is at
                                edges[lengthEdge] = new Edge(vertices[i],vertices[j],weight);
                                vertices[i]->connections.append(vertices[j]);
                                lengthEdge++;
                        }
                }
        }
        text.close();
}

/*-----------------------------------------
 * Deconstructor
 * PreConditions:       n/a
 * PostConditions:      everything allocated in the Graph object will be deallocated
-----------------------------------------*/
Graph::~Graph(){
        dealloc();
}


/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                    Graph - FUNCTIONS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*-----------------------------------------
 * dfs()
 * PreConditions:       a valid graph, can be empty
 * PostConditions:      vertices visited will be printed out
                        will explore all of the elements of the spanning tree
                        if empty, nothing will bre printed out
-----------------------------------------*/
void Graph::dfs(){
        for (int i=0; i<capacity; i++)
                if (vertices[i]->color == 'w')
                        dfsVisit(vertices[i]);
        if (capacity != 0)
                cout << endl;
}

/*-----------------------------------------
 * dfsVisit()
 * PreConditions:       pointer to a vertex in the graph
 * PostConditions:      vertex will be colored in "grey", denoting that it has been visited
                        will go through all connections that haven't been visited
-----------------------------------------*/
void Graph::dfsVisit(Vertex * u){
        u->color = 'g';                                 //mark vertex as visited
        cout << u->data << " ";                         //print out vertex visited
        for(int i=0; i<u->connections.length(); i++)
                if (u->connections[i]->color == 'w'){   //check if vertex has not been visited
                        u->connections[i]->pred = u;
                        dfsVisit(u->connections[i]);    //recursivley do this if it hasn't bee
n visited
                }

}



/*-----------------------------------------
 * Kruskal()
 * PreConditions:       a valid graph
 * PostConditions:      will find and print out all edges in the Miniumum Spanning Tree (MST)
-----------------------------------------*/
```

```
void Graph::Kruskal(){
        List<Edge> a;                                           //contains all edges in MST
        DisjointSets<Vertex> vertexSet(capacity);
        DSNode<Vertex>* nodes[capacity];                        //all nodes created with all v
ertices
        for (int i=0; i<capacity; i++)
                nodes[i] = vertexSet.makeSet(vertices[i]);

        MinPriorityQueue<Edge> queue(lengthEdge);

        //go through every value in the nodes, and try to find the vertex each edge points to
        for (int i=0; i<capacity; i++){
                for (int j=0; j<lengthEdge; j++){
                        if (edges[j]->u == nodes[i]->data)
                                edges[j]->nodeU = nodes[i];
                        if (edges[j]->v == nodes[i]->data)
                                edges[j]->nodeV = nodes[i];
                }
        }
        //insert all edges into an Min Priority Queue, so all edges will be sorted by weight
        for(int i=0; i<lengthEdge; i++)
                queue.insert(edges[i]);
        //dequeue each element, check if
        for(int i=0; i<lengthEdge; i++){
                Edge * e = queue.extractMin();
                if (vertexSet.findSet(e->nodeU) != vertexSet.findSet(e->nodeV)){
                        a.append(e);
                        vertexSet.unionSets(e->nodeU, e->nodeV);
                }
        }

        //print out all edges in the list
        cout << a << endl;
}


/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                  Graph - PRIVATE FUNCTIONS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/


/*-------------------------------------------
 * dealloc()
 * PreConditions:       n/a
 * PostConditions:      will deallocate all memory in the Graph object
-------------------------------------------*/
void Graph::dealloc(){
        for (int i=0; i<capacity; i++)
                delete vertices[i];
        delete[] vertices;
        for (int i=0; i<lengthEdge; i++)
                delete edges[i];
        delete[] edges;
}
```

```
// test_graph.cpp
// Graph class tests
// Clay Sarafin & Taylor Heilman
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "graph.h"

using namespace std;


void mainTest(){
        Graph graph("test.txt");

        cout << "Output for Kruskal's Algorithm:" << endl;
        graph.Kruskal();

        cout << "Output for Depth First Search:" << endl;
        graph.dfs();
}


int main(){
        mainTest();
        return 0;
}
```

Hash Tables

Proficiency: Mastery

A hash table is a modified array with certain properties. These properties can vary depending on what type of hash table you are using. One thing universal to all hash tables is the use of a hash function. The hash function is what makes a hash table effective. The hash function determines where in the array an item will be stored. In a simple array an item might inserted into index 1, the next item into index 2 and so on. Hash table need not only the item to be inserted but also a key. A hash table takes the key of an item and inputs the key into the hash function. Based on the key the hash function will output the exact index in which the item should be inserted; this is called hashing. The goal of a hash function is to evenly distribute all elements into the allotted slots in the array. Creating a good function is an art. A normal hash function takes in the key, performs a series of operations, shifts and other almost “random” things to essentially wipe the identity away from the key. Lastly the hash function mods the final value by the number of slots, guaranteeing the hash value to reflect a valid index in the hash table. When making a hash function you want to try and avoid operations that will give the same result for different keys. For example, modding by a power of 10 or 2 will cause universal results, such as shifting the binary value. Ideally the hash function should not treat keys analogously. If the keys our hash function are of string type, similar keys such as “cat”, “bat”, “cats” should not have similar or equal hash values. Instead, the hash function should act randomly. This ideal idea of hashing is called uniform hashing and it is what all hash functions try to emulate. Uniform hashing means

if a hash table has ‘m’ slots and ’n’ elements to insert, the probability of an element to be inserted into a slot should be n/m. Unfortunately this ideal situation does not occur often and sometimes multiple keys, although different, get hashed to the same index known as a collision.

Chaining is one way of dealing with a collision. When a collision occurs chaining simply has a pointer in the index with multiple elements which points to a linked list. Then whenever a collision occurs the new item is added to the end of the list. An inefficient hash function may cause trouble for chaining. If many items chain to the same index the time it takes to find an item in a Hash Table will stop being constant and start being linear, since we may have to completely traverse a linked to find an item.

Another solution is linear probing. With linear probing if a collision occurs we check the next index to see if it is open. If the next index is available we place the item in it. In bad cases linear probing begins to act like chaining. When multiple items hash to the same index, we may have to traverse through the array to find the desired item

A final solution is quadratic probing. Quadratic probing is similar to linear pro, but instead of checking the next index we use a quadratic formula to see which index to check next. For example if our formula was “index = index + 4”, the first index we check is 4 away from the index we were hashed to. If that index is full the next index we check will be 12 away from the hashed index, then 28 indexes away. This way of probing stops the cluttering around “popular” indexes unlike linear probing.

As you can see my knowledge of Hash Tables is pretty extensive, therefor I feel the title of mastery is suitable for this topic.

```
#ifndef HASH_H
#define HASH_H

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

template <class KeyType>
class HashTable
{
 public:
    HashTable(int numSlots);
    ~HashTable();
    KeyType* get(KeyType& k);
    void insert(KeyType *k);
    void remove(KeyType& k) ;

        std::string toString(int slot);

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

#include "hash.cpp"

#endif
```

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include "hash.h"
using namespace std;




/*
 *              Construct a hashtable with slots equal to numSlots.
 *              Precondition: numSlots > 0
 *              Postcondition: Constructs a new Hashtable holding numSlots
 */
template <class KeyType>
HashTable<KeyType>::HashTable(int numSlots)
{
        slots = numSlots;
        table = new List<KeyType>[numSlots];
}

/*
 *              Destructor
 *              Precondition:
 *              Postcondition: table is deleted
 */
template <class KeyType>
HashTable<KeyType>::~HashTable()
{

        delete[] table;
}

/*
 *              Inserts a KeyType into the HashTable
 *              Precondition: k is a non-null object
 *              Postcondition: k is added to the hash table
 */
template <class KeyType>
void HashTable<KeyType>::insert(KeyType *k)
{
        int slot = k->hash(slots);
        table[slot].insert(0,k);
}

/*
 *              Get the value associated with k
 *              Precondition: k is a non-null object
 *              Postcondition: Returns the value associated with k, or null if not found
 */
template <class KeyType>
KeyType* HashTable<KeyType>::get(KeyType& k)
{
        int slot = k.hash(slots);
        int ind = (table[slot]).index(k);

        //throw error if k not found
        return (table[slot])[ind];

}
```

```
/*
 *              Removes a value from the hash table
 *              Precondition: k is a non-null object
 *              Postcondition: Remove k from the hashtable
 */
template <class KeyType>
void HashTable<KeyType>::remove(KeyType &k)
{
        //throw error if empty?

        table[k.hash(slots)].remove(k);
}

/*
 *              Generates a string representation of the hash table
 *              Precondition:
 *              Postcondition: Returns a string representation of the hash table
 */
template <class KeyType>
std::string HashTable<KeyType>::toString(int slot)
{

        string s = "(";

        for (int i = 0; i < table[slot].length(); i++)
        {
                stringstream str;
                string ss;
                s += "[";
                str << *((table[slot])[i]);
                str >> ss;
                s += (ss + "], ");
        }
        cout << s << "}" << endl;
        return s + "}";

}
```

```
#include <stdlib.h>
#include <fstream>
#include <cstdlib>
#include <string>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include "test.h"

using namespace std;

int main()
{

        Test T1, T2, T3, T4;
        T1.key = 1;             // 1
        T2.key = 2;             // 2
        T3.key = 3;             // 0
        T4.key = 4;             // 1



        //make hash table (&T1)
        HashTable<Test> *H1 = new HashTable<Test>(3);

        H1->insert(&T1);
        H1->insert(&T2);
        H1->insert(&T3);
        H1->insert(&T4);
        H1->remove(T1);
        H1->get(T2);



}
```

```
#ifndef _HASHDICTIONARY_H_
#define _HASHDICTIONARY_H_
#include <iostream>
#include <cstdlib>
#include "hash.h"

template <class KeyType>
class HashDictionary
{
public:
    int max_size;
    HashDictionary();
    ~HashDictionary();
    KeyType* get(KeyType& k);
    void insert(KeyType *x);
    void remove(KeyType& k);
    bool empty();
    int hash(string str); //should be in the DictionaryTest



private:

        HashTable<KeyType> *table;
};

#include "dict.cpp"

#endif // _HASHDICTIONARY_H_
```

```
#include <iostream>
#include <cstdlib>
#include "hash.h"


using namespace std;

template <class KeyType>
int HashDictionary<KeyType>::hash(string str)
{
   /* int hash = 5381;
    int c;
    const char *chr = str.c_str();

    while (c = *chr++)
        hash = ((hash << 5) + hash) + c;

    return abs(hash/10000000);*/

        int numslots = 1000;
        int len = str.length();
        unsigned int t = 0;


        for (int i = 0; i < len; i++)
        {
                t = ((25 * t + str[i]) *str[0] + (len/2) % len) % numslots;
        }
        return t;

}

template <class KeyType>
HashDictionary<KeyType>::HashDictionary()
{
    max_size = 1000;
    table = new HashTable<KeyType>(max_size);
}

template <class KeyType>
HashDictionary<KeyType>::~HashDictionary()
{
    table.~HashTable();
}

template <class KeyType>
KeyType *HashDictionary<KeyType>::get(KeyType& k)
{
    int index = hash(k.title);
    return table->get(k);
}

template <class KeyType>
bool HashDictionary<KeyType>::empty()
{
    if (table[0] == 0)
        return true;
    else
        return false;
}

template <class KeyType>
void HashDictionary<KeyType>::insert(KeyType *x)
```

```
{
    int index = hash(x->title);
    table->insert(x);
}

template <class KeyType>
void HashDictionary<KeyType>::remove(KeyType& k)
{
    int index = hash(k.title);
    table->remove(k);
}
```

```
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>
#include <string.h>
#include <sstream>

#include "dict.h"
using namespace std;
class Movie
{
public:
    string title;
    string year;
    string cast;



    Movie()
    {
        title = " ";
        cast = " ";
    }

    int hash (int numSlots)
        {
                        return (title.length() % numSlots);
        }

    Movie(string initTitle, string initCast)
    {
        title = initTitle;
        cast = initCast;
    }

    ~Movie()
    {

    };


    bool operator<(const Movie &other) const;
    bool operator==(const Movie &other) const;
    friend ostream &operator<<(ostream &strm, Movie &e);
};

bool Movie::operator==(const Movie &other) const
{
    return title == other.title;
}

bool Movie::operator<(const Movie &other) const
{
    return title < other.title;
}


ostream &operator<<(ostream &strm, Movie &e)
{
    strm << e.title;
    return strm;
```

```
}


int main()
{
    HashDictionary<Movie> *b = new HashDictionary<Movie>;
    Movie *movie;

    ifstream file("movies-mpaa.txt");
    string str;

   int start = clock();

    string mvieTitle;
    string theYear;


    while (getline(file, str))
    {
        Movie *e = new Movie();
        int j = 0;
        int k = 0;
        int l = 0;
        for (int i = 0; i < str.length(); i++)                          // This for loop gets
the title
        {
            if (str[i] == '(')
            {
                j = i-1;
                k = i;
                mvieTitle = str.substr(0, j);
                e->title = mvieTitle;
                break;
            }
        }

        for (int r = k; r < str.length(); r++)                          // Gets the year
        {
            if (str[r] == ')')
            {
                l = r;
                theYear = str.substr(k+1, 4);
                e->year = theYear;
                break;
            }

        }


        e->cast = str.substr(l+1, str.length()-l);
        b->insert(e);

    }

    int end = clock();
    cout << "it took " << end - start << " ticks, or " << ((float)end - start)/CLOCKS_PER_SEC
<< " seconds." << endl;


    bool done = false;
    int choice;
```

```
    while (!done)
    {
        cout << endl;
        cout << "MENU" << endl;
        cout << endl;
        cout << " 1. Search for a movie" << endl;
        cout << " 2. Delete a movie" << endl;
        cout << " 3. Insert a movie" << endl;
        cout << " 4. Quit" << endl;
        cout << endl;
        cout << "Choice: ";
        cin >> choice;

        cout << endl;
        if (choice == 1)
        {
            string movieTitle;
            cin.ignore(255, '\n');
            cout << "Enter the name of the movie: " << endl;
            getline(cin, movieTitle);
            movieTitle = movieTitle.substr(0, movieTitle.length());
            Movie *x = new Movie(movieTitle, " ");

            int start = clock();

            movie = b->get(*x);

            int end = clock();
            cout << "it took " << end - start << " ticks, or " << ((float)end - start)/CLOCKS_
PER_SEC << " seconds." << endl;


            if (movie != NULL)
            {
                cout << "Title: " << movie->title << '(' << movie->year << ')' << endl;
                cout << "Cast: " << movie->cast << endl;
            }
            else
            {
                cout << "not found" << endl;
            }

        }
        else if (choice == 2)
        {
            string movieTitle;
            cin.ignore(255, '\n');
            cout << "Title of Movie to Delete: ";
            getline(cin, movieTitle);
            Movie *x = new Movie(movieTitle, " ");

            timeval timeBefore, timeAfter;
            long diffSeconds, diffUSeconds;
            gettimeofday(&timeBefore, NULL);
            b->remove(*x);
            gettimeofday(&timeAfter, NULL);

            diffSeconds = timeAfter.tv_sec - timeBefore.tv_sec;
            diffUSeconds = timeAfter.tv_usec - timeBefore.tv_sec;
            cout << diffSeconds + diffUSeconds/1000000.0 << " seconds" << endl;

        }
        else if (choice == 3)
```

```
        {
            string movieTitle;
            string movieYear;
            string movieCast;
            cin.ignore(255, '\n');
            cout << "Title of movie to insert: ";
            getline(cin, movieTitle);
            cout << "Year of movie to insert: ";
            getline(cin, movieYear);
            cout << "Cast of movie to insert: ";
            getline(cin, movieCast);
            Movie *x = new Movie(movieTitle, movieCast);
            x->year = movieYear;
            b->insert(x);
        }
        else if (choice == 4)
        {
            done = true;
        }

        else
        {
                cout << "Please choose options 1, 2, 3, or 4." << endl;
                done = true;
        }
    }


}
```

Linear Data Structures

Proficiency: Mastery with Distinction

In the field of computer science knowledge of linear data structures is vital. Some of the most commonly used data structures are linear data structures. Lists, stacks, queues are all fundamental structures that are used and modified to be used in other data structures. The knowledge of how these data structures work, how they're implemented and proper usage is vital. Last year in CS 173 we thoroughly studied stacks, lists and queues. We implemented a stack using linked lists, then modified the class to use doubly linked lists and finally we implemented the stack using dynamically allocated arrays. All three implementations of the stack are attached. A stack is a last in first out structure, meaning the first item is added to the stack and the next items are added on top of the previous item (stacked on one another).It is easy to visualize a stack as a vertical container with a hole at the top. A Stack has four operations: pop(), peek(), push(x), and isEmpty(). Push(x) adds item x to the top of the stack. Pop() removes and returns the item at the top of the stack. Peek() does the same as pop() but does not remove the item. Lastly, isEmpty() returns a boolean value, true if the stack has no items in it and false else wise.

A queue, unlike a stack, is a first in first out data structure, meaning the first item added to the queue is first in line to leave the queue. A good visualization of a queue is a horizontal pipe; the first thing you push in the left side of the pipe will be the first thing that comes out of the right side of the pipe. A queue can be implemented almost identically to a stack, with a few changes here and there. Popular elementary ways of

implementing a stack are arrays, linked lists and doubly linked lists. A queue is like a line in the grocery store (hence why British call lines ‘queues’), the early you get your order in, the early you will be served. A queue has three operations: isEmpty(), enqueue(x), dequeue(). isEmpty() for a queue is the same as is empty for a stack. Enqueue(x) adds the item x to the end of the queue. Dequeue() removes the front item from the queue. A more advanced type of queue is the Minimum Priority Queue which we implemented this year using a heap. A Minimum Priority Queue is a type of queue where the the priority of one item may be more important than another. The important items are kept at the front of the queue, sort of like a celebrity cutting the line to get into a restaurant.

Last is the list ADT. A list class in computer is essentially just a computerized form of the way we use lists everyday. Unlike a queue and stack, when an item is added to a list has noImagine making a list of guests to invite to a party. You start with person 1, then add person 2 and so on. A list ADT is just a linear sequence of items, where an item can be accessed or edited by knowing its index in the list. A basic list has five operations: add(x, i), remove(x), append(x), isEmpty() and size(). Add(x, i) varies from append(x) in the fact that add(x, i) puts item x at the specified index i, while append adds item x to the end of the list. Remove(x) removes the first instance of x in the list. isEmpty() acts the same as queue and stack. Size() returns the amount of items in the list.

As you can see linear data structures are all somewhat similar, but each has unique characteristics that may make one ADT better suited for a problem than another. All of

these structures can be used to hold integers, strings, chars, etc. After implementing all of these ADTs I feel like I have more than masted Linear Data Structures.

```
/*
Taylor Heilman
list class using doubly linked lists
list.cc
*/

//#include "list.h"
#include <stdlib.h>
#include <iostream>
using namespace std;

//=========================================
//      Default Constructor
//=========================================
template<class T>
List<T>::List ( void )
{
        head = NULL;
        tail = NULL;
        size = 0;

}

//=========================================
//      Destructor
//=========================================
template<class T>
List<T>::~List ( void )
{
        dealloc();
}

//=========================================
//      Copy Constructor
//=========================================
template<class T>
List<T>::List ( const List<T>& source )
{
        copy(source);

}

//=========================================
//      Assignment Operator
//=========================================
template<class T>
List<T> & List<T>:: operator=   (const List<T>& source)
{
        if(this != &source)
        {
                dealloc();
                copy(source);
        }

        return *this;
}



 //=========================================
 //     Append
 //=========================================
template<class T>
void List<T>::append  (const T& x)
  {
          Node<T> * temp;
          temp = new Node<T>;

         if (head == NULL)              // appending to empty list
         {
```

```
                head = temp;
                tail = temp;
                temp -> item = x;
                temp -> next = NULL;
                temp -> prev = NULL;
                size++;
        }

        else                            // apending to end of list
        {
               tail->next = temp;
               temp->item = x;
               temp->prev = tail;
               temp->next = NULL;
               tail = temp;
               size++;

        }

  }



//=========================================
//      Insert
//=========================================
template<class T>
void List<T>::insert   (int index, const T & x)
{
        Node<T> * temp;
        temp = new Node<T>;

        if (index < 0 or index > size)          //indes out of bounds
        {
               delete temp;
               throw IndexError();

        }

        else if (size == 0)                     // empty list
        {
               temp->item = x;
               temp->prev = NULL;
               temp->next = NULL;
               head = temp;
               tail = temp;
               size++;

        }

        else if (index == 0 )                   // inserting to first spot
        {
                temp->item = x;
                temp->prev = NULL;
                temp->next = head;
                head->prev = temp;
                head = temp;
                size++;

        }
        else if (index == size)                 // inserting to last spot
        {
               append(x);
        }

        else                                    // inserting in middle
        {
               temp = head;
```

```
                for(int i=0; i < index; i++)
                {
                        temp = temp-> next;
                }

                Node<T> * temp2;
                temp2 = new Node<T>;
                temp2->next = temp;
                temp2->prev = temp->prev;
                temp2->item = x;
                temp->prev = temp2;
                temp2->prev->next = temp2;
                size++;

        }

}



//=========================================
//      String
//=========================================
template<class T>
string  List<T>::str()
 {

         string str = "";
         Node<T>*temp = head;
         char Reason [50];
         str += "[";
         while (temp != NULL)
         {
                 if (temp -> next != NULL)
                 {
                         sprintf(Reason,"%d", temp -> item);
                         str+= Reason;
                         str += ", ";
                 }

                 else
                         if(temp -> next == NULL)
                         {
                                 sprintf(Reason, "%d", temp -> item);
                                 str+= Reason;
                         }
                         temp = temp -> next;
         }
         str += "]";
         return str;
 }

//=========================================
//      Index
//=========================================
template<class T>
int List<T>::index  ( const T & x )
 {

         Node<T> * temp = head;
         int place = 0;

         while (temp != NULL and temp->item != x)
         {
                 temp = temp->next;
                 place++;
         }

         if (temp == NULL)      // if list is empty or item isn't in list
                 return -1;
```

```
         else
                 return place;  // return index of the item

}


//=========================================
//      Pop
//=========================================
template<class T>
T   List<T>::pop (int index)
{
        T x;
        int y = size -1;

        if ( head == NULL)                                  //empty list
        {
                throw IndexError();

        }
        else if (index == y)                    // popping last item
        {
                Node<T> * temp = tail;
                x = tail->item;
                tail = tail->prev;
                tail->next = NULL;
                delete temp;
                size--;
                return x;

        }

        else if( index == 0)                    // popping first item
        {
                Node<T> * temp = head;
                x = head -> item;
                head = head-> next;
                head->prev = NULL;
                delete temp;
                size--;
                return x;
        }


        else if(index > 0 and index < y)        // poppoing from middle of list
        {
                Node<T> * temp = head;
                for(int i=0; i<index; i++)
                {
                         temp = temp->next;
                }
                x = temp->item;
                (temp->prev)->next = temp->next;
                (temp->next)->prev = temp->prev;
                delete temp;
                size--;
                return x;


        }

        else                                    // no index given
        {
                Node<T> * temp = tail;
                x = tail->item;
                tail = tail  ->prev;
                tail->next = NULL;
                delete temp;
                size--;
                return x;
```

```
        }

}



//=========================================
//      Indexing Operator
//=========================================
template<class T>
T &  List<T>::operator[]  (int index)
{
        if (index < 0 or index > size-1)        //index out of bounds
                throw IndexError();
        else
        {
                Node<T> * temp = _find(index);
                return temp->item;

        }
}


//=========================================
//      resetForward
//=========================================
template<class T>
void   List<T>::resetForward(void)
{
        currentFwd = head;

}

//=========================================
//      next
//=========================================
template<class T>
T    List<T>::next()
{
        Node<T> * temp = currentFwd;
        if (temp == NULL)
                throw StopIteration();
        else
        {
                T z = currentFwd->item;
                currentFwd = currentFwd->next;
                return z;
        }
}

//=========================================
//      resetReverse
//=========================================
template<class T>
void  List<T>::resetReverse(void)
{
        currentRev = tail;
}

//=========================================
//      prev
//=========================================
template<class T>
T    List<T>::prev  (void)
{
        Node<T> * temp = currentRev;
        if (temp == NULL)
                throw StopIteration();
```

```
        else
        {
                T z = currentRev->item;
                currentRev = currentRev->prev;
                return z;
        }
}



//=========================================
//      copy
//=========================================
template<class T>
void  List<T>::copy  (const List<T>& source)
{
        Node<T> *snode, *node;                  // deep copy

        snode = source.head;
        if (snode)
        {
                node = head = new Node<T>(snode->item);
                snode = snode->next;
        }
        else
                head = NULL;
        while(snode)
        {
                node->next = new Node<T>(snode->item);
                node = node->next;
                snode = snode->next;
        }
        size = source.size;
}




//=========================================
//      dealloc
//=========================================
template<class T>
void   List<T>::dealloc     ()
{
        Node<T> * temp = head;

        while( temp != NULL )
        {
                head = head->next;
                delete temp;
                temp = head;
        }

        delete temp;
}



//=========================================
//      _find
//=========================================
template<class T>
Node<T>*  List<T>:: _find  (int index)
{
        Node<T> * temp = head;
        for(int i=0; i<index; i++)
                temp = temp->next;

        return temp;
}
```

```
//==========================================================================
// Matt Kretchmar
// April 1, 2015
// list.h
//
// This file contains the class definition for a List ADT class.
// ** Do not modify the Node<T> or List classes. **
// ** You will modify the StopIteration class at the bottom **
//
//==========================================================================
// List ADT
//
// The List class implements a sequence of stored items all of the same datatype.
// There are methods to add and remove items from the List, to query the List for
// an item, to index into the list at a specific location, and to iterate through
// the list.
//
// Default Constructor: creates an empty List (no items)
// Copy Constructor:    creates a new List that is an exact copy of an existing List.
// Destructor:          cleans up the memory for an existing List to be deleted.
// Assignment Operator: makes a copy of an existing List for the assigned List.
//
// length():            returns the number of items in the List.
// append(ItemType &x): adds item x to the end of the existing List.  Note that
//                      duplicate items are permitted.
// insert(i,x):         inserts item x at location i in the List.  The existing
//                      items are moved towards the end of the List to make room
//                      for the new item.  Valid values for i are 0 to length().
//                      If length() is the index, this will add the new item to the
//                      end of the list (such as in append).
// pop(i):              removes and returns item at index i from the list.  Valid
//                      values for i are 0 to length()-1.  The argument is optional
//                      will default to removing the last item in the list if i is not gi
ven.
// operator[i]:         access (by reference) the item at index i.  Valid values
//                      for i are 0 to length()-1.  The access by reference allows
//                      the user to change the value at this index.
// index(x):            returns the index of the first occurrence of item x in
//                      the List, returns -1 if x is not in the list.
// resetForward():      resets the forward iterator to the front of the list.
// resetReverse():      resets the backward iterator to the end of the list.
// next()               returns the value of the next item in the list using the
//                      forward iterator location.  The forward iterator is then
//                      moved to the next item.
// prev()               returns the value of the next item in the list using the
//                      backward iterator location.  The backward iterator is then
//                      moved to the next (previous) item.
// str()                Converts the List into a string, follows Python format.
//                      Example: "[1, 2, 3]" or "[]"
// cout <<              Overloads the cout << operator for printing. Follows the
//                      same format as in str().
//==========================================================================



#include <iostream>
#include <stdio.h>
#include <stdlib.h>
using namespace std;

#ifndef LIST_H
#define LIST_H

template <class T>        // where you can change the type of item stored in the list
struct Node
{
    T              item;                                    // data item stored in thi
s link
    Node *      next;                                    // pointer to next link in li
st
```

```
    Node *      prev;                                       // pointer to previous link i
n list

    Node () { next = prev = NULL; }                         // default constructor
    Node (const T & x) { next = prev = NULL;          // constructor with item
                                item = x; }
};

template <class T>
class List
{
  public:
    List();                                                 // default constructor
    List(const List<T>& source);                               // copy constructor
    ~List();                                                // destructor

    List<T> &      operator=   (const List<T>& source);           // assignment operator
    int         length      ()    const { return size; }    // return the length of the l
ist
    void        append      (const T& x);             // append an item to the end of the
list
    void        insert      (int index, const T& x); // insert an item in position index
    T    pop         (int index = -1);                // delete item at position index (or
 last
                                                            // item if no index given)
    T &  operator[]  (int index);                       // indexing operator
    int         index       ( const T &x );           // return the index of the first occ
urrence of x
    string      str();                                        // return the string represen
tation

    void        resetForward(void);                           // reset forward iterator to
the head of the list
    T    next();                                        // return the next item in the list
and advance
                                                            // forward iterator pointer
    void        resetReverse(void);                           // reset reverse iterator to
the tail of the list
    T    prev        (void);                           // return the prev item in the list
and advance
                                                            // reverse iterator pointer
 private:
    Node<T>     *head,                                        // head of the linked list
                *tail,                                        // tail of the linked list
                *currentFwd,                                  // current pointer for the fo
rward iterator
                *currentRev;                                  // current pointer for the re
verse iterator
    int         size;                                         // length of the list


    void        copy        (const List<T>& source);           // copy source list to thi
s list
    void        dealloc     ();                           // deallocate the list
    Node<T>*       _find       (int index);                   // return a pointer to the
 node in position index


    friend ostream&    operator<<  (ostream& os, const List<T>& l)
    {
        /*
         string str = "";
         Node<T> * temp = head;
         char Reason [50];
         str += "[";
         while (temp != NULL)
         {
                 if (temp -> next != NULL)
                 {
                         sprintf(Reason,"%d", temp -> item);
```

```
                        str+= Reason;
                        str += ", ";
                }

                else
                        if(temp -> next == NULL)
                        {
                                sprintf(Reason, "%d", temp -> item);
                                str+= Reason;
                        }
                        temp = temp -> next;
        }
        str += "]";
        return str;
        */                              //Returns error
    }

};

//================================================================================
// IndexError
// This class implements an exception for an indexing error.
//================================================================================
class IndexError {                                              // index error exception
public:
    IndexError() {};
    ~IndexError() {};
    const char *Reason () const { return "Index out of bounds."; }
};


//================================================================================
// StopIteration
// This class implements an exception for iterating (forward or backward) beyond
// the start/end of the list.
//================================================================================
class StopIteration {
public:
        StopIteration() {};
        ~StopIteration() {};
        const char *Reason () const { return "Iteration error \n System self destructing
in\n 3... \n 2... \n 1...";}

};       // stop iteration exception


#endif
#include "list.cc"
```

```
/*
Taylor Heilman
Stack class using linked lists
Stack2.cc
*/


#include "Stack2.h"
#include <stdlib.h>
#include <iostream>
using namespace std;

//=========================================
//                      Default Constructor
//=========================================
Stack::Stack ( void )
{
    Link * head = NULL;
    top = 0;

}

//=========================================
//                                Destructor
//=========================================

Stack::~Stack   ( void )
{
    Link * temp = head;
        while( temp != 0 ) {
        Link* next = temp->next;
        delete temp;
                temp = next;
}
        head = 0;
}

//=========================================
//                                Push
//=========================================
void        Stack::push    ( int item )
{
        Link * temp;
    temp = new Link;
    temp -> item = item;
    temp -> next = head;
    head = temp;
    top++;
}

//=========================================
//                                Pop
//=========================================
int         Stack::pop     ( void )
{
    if ( head == NULL )
    {
        cout << "Error: cannot pop from empty stack\n";
        exit(1);
    }

    else
        {

        Link * temp = head;
                head = head -> next;
                int x = temp -> item;
                delete temp;
                top--;
                return x;
```

```
        }


}

//=========================================
//                               Size
//=========================================
int         Stack::size    ( void )
{
    return top;
}
```

```
//=========================================
// Matt Kretchmar
// March 9, 2015
// Stack.h
//=========================================

#include <iostream>
using namespace std;

#ifndef STACK_H
#define STACK_H

#define DEFAULT_CAPACITY 5

class Stack
{
private:
    int top;        // index of top (empty) item
    int capacity;   // size of array for stack
    int *stack;     // dynamically allocated array
                    // to hold stack
public:
               Stack   ( void );
              ~Stack   ( void );
    void       push    ( int item );
    int        pop     ( void );
    int        size    ( void );
};

#endif
```

```
/*
Taylor Heilman
March 5, 2015

project6.cc
Project 6: Stacks With Dynamic Arrays
The goal of this project is to implement
stack behavior using dynamically allocated arrays.
*/


#include <iostream>
using namespace std;

int main ( void)
{
        int * list;             // pointer
        list = new int[5];      // dynamically allocating 5 places
        int capacity = 5;       // initialize capacity of allocated memory
        int length = 0;         // initialize length of array
        int num;
        char letter;            // p,q,s,x
        while (true)
        {
                if (length == capacity)  // Array is full
                {
                        int * tmp = new int [capacity + 5];  //Create a new, larger array

                        for(int i=0; i < length; i++)
                                tmp[i] = list[i];                       // copy old array
 into new, larger array

                        delete [] list;         // delete old array
                        list = tmp;             // change pointer to new array
                        capacity = capacity + 5;   // add 5 to capacity of array
                }



                cin >> letter;   // p, q, s, or x

                //=================================================================
                // Quit
                //=================================================================

                if (letter == 'q')  // Quit
                {
                        delete [] list;         // delete the allocated memory
                        exit(1);                        // quit program
                }

                //=================================================================
                // Push
                //=================================================================

                else if (letter == 'p')    // Push
                {
                        cin >> num;                             // value added to array
                        list[length] = num;             // set array index to input numbe
r
                        length  ++;                             // size of array increase
s by 1, move pointer up
                }


                //=================================================================
                // Pop
                //=================================================================

                else if (letter == 'x')         // Pop
```

```
                {
                        if (length == 0)        // empty array
                        {
                                delete [] list;         // delete the allocated memory
                                exit(1);                        // quit program
                        }
                        cout << list[length-1] << '\n' ;        // print number at top of
 stack
                        length--;                                                       /
/ length of array decreases by 1
                }


                //=====================================================================
                // Size
                //=====================================================================

                else if (letter == 's')         // Size
                {
                        cout << length << endl;   // Print amount of items in array
                }



                //=====================================================================
                // For Troubleshooting
                //=====================================================================


                //for (int j=0;j<length;j++)          // Print out the array
                //      cout <<  list[j] << endl;

                //cout << "capacity: " << capacity << '\n' ;  //See the size of allocated
 memory
        }



        return 0;
        }
```

Object-oriented programming in C++

Proficiency: Mastery

Inheritance and polymorphism are fundamental concepts in computer science. Although similar, inheritance and polymorphism vary. Inheritance is a simple idea. Say you made a class, called class A. Then while creating another class, class B, you notice that class A and class B share some similar concepts and maybe some functions in class A could help you class B out. Then any function in class A that is public in class A can be used in class B. Public means variables and functions declared as public are available from anywhere, for other classes and instances of the object. Private narrows the scope so your private variables and functions are visible in its own class only. Another option is protected. Protected is only used with inheritance. Functions and variables that are given a protected scope can only be accessed within the class and to any other classes that extend the base class. An example of us using inheritance is when we implemented a minimum priority queue ADT. We inherited from a MinHeap class and used functions in the MinHeap class in our minimum priority queue. Inheriting saves time because you an simply inherit functions from other classes instead of rewriting code.

Unlike many computer science terms, “polymorphism” means exactly what it does. ‘Poly’ means ‘many’ and ‘morphism’ means ‘to change or form’, So polymorphism is the ability to present the same interface for differing data types. So a polymorphic type has the ability to work with many different types. an example of polymorphism is

generic programming. With generic programming, code is written with no mention of a specific type, therefore it can be used transparently with a variety of types.

Most of our projects this year incorporated inheritance, so a variety of projects would be acceptable to attach. In project 1000 we made a Dictionary ADT that inherited from our red-black tree template class. In this project we used the structure and functions of a red black tree to store dictionary values.

```
// dict.h
// header file for dictoinary class
// Clay Sarafin & Taylor Heilman

#ifndef DICT_H
#define DICT_H

#include <iostream>

#include "rb.h"

using namespace std;

template <class KeyType>
class Dictionary : public RBTree<KeyType>{
        public:
                Dictionary() : RBTree<KeyType>() {};

                using RBTree<KeyType>::get;
                using RBTree<KeyType>::insert;
                using RBTree<KeyType>::count;
                using RBTree<KeyType>::inOrder;
                using RBTree<KeyType>::empty;


                std::string toString();
};

#include "dict.cpp"

#endif
```

```
// dict.cpp
//
// Clay Sarafin & Taylor Heilman

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


using namespace std;


/*-------------------------------------------
 * toString()
 * PreConditions:       inOrder from BST class does not have brackets, every entry is followed
 by ", "
                                and does not end with std::endl
                      type KeyType's operator<< is overloaded, and is formatted as "key:valu
e"
 * PostConditions:     string representation of the entire dictionary is printed out
-------------------------------------------*/

template <class KeyType>
std::string Dictionary<KeyType>::toString(){
        std::string str;
        str += "{";
        str += inOrder();
        str += "}";

        return str;
}
```

```
// 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();              // default constructor
    MinPriorityQueue(int n);         // construct an empty MPQ with capacity n
    MinPriorityQueue(const MinPriorityQueue<KeyType>& pq);  // copy constructor

    KeyType* minimum() const;                    // 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
    bool find(KeyType* key);
    string findCode(KeyType* key, int lenght);

    // 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
class IndexError { };     // MinPriorityQueue key exception

#include "pq.cpp"

#endif
```

88

call MinHeap constructors via initializer lists

Binary Search Trees

Proficiency: Mastery with Distinction

The binary tree structure is throughout computer science. A binary tree is a structure that starts a single root and has two children nodes (left and right). Every node in the tree has at most two children. Hence each row in a complete binary tree has 2 times more nodes than the row above it. This characteristic is what makes binary trees so appealing in computer science. A traversal from the root to a leaf in a complete binary tree will take log n time because the height of a complete tree is log n. Binary Search Trees are a binary trees with more rules. In a binary search tree the left child must be less than or equal to the parent node and the right child must be greater than the parent node. This additional characteristic makes searching a binary search tree very efficient. If you're looking for some node with value x, you simply compare the current node you're looking at with your desired value. If the current node has a larger value than the node you're looking for, you know your desired node is in the left subtree since it is smaller. Because the height of the binary search tree is at best log n, searching, inserting and deleting items from it take on average *O(log n)*. Unfortunately all three functions have a worst time complexity of *O(n).* This occurs if we're inserting a sorted list. If we first insert a 1, then a 2, then a 3, then a 4 and so on every new node is the right child of the previous node. In this case the height of the binary tree is n and not log n. A way to avoid this problem is using another binary tree- The Red Black Tree.

Red Black Trees are a type of binary search tree. The RBT has all the characteristics as a binary search tree as well as five more. These extra five

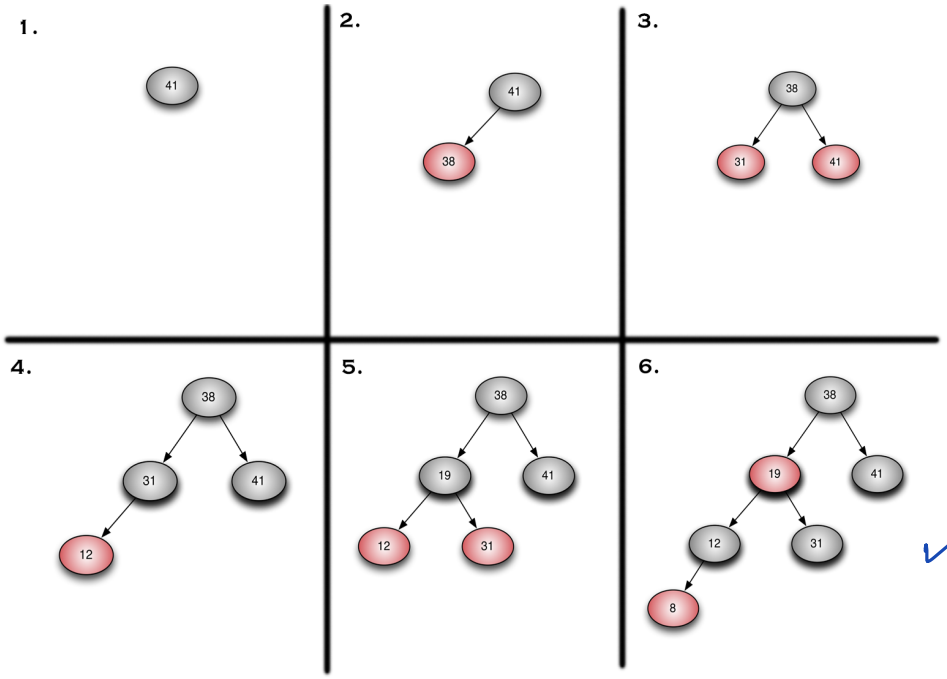
characteristics guarantee that a red black tree's height will never be more than 2log(n+1). The five characteristic are: Every node is red or black, the root of the tree is black, the leaves of the tree are black, if a node is red both children are black and for each node, all paths from the node to a descendant leaf contain the same amount of black nodes, this is known as the black height. When inserting nodes into a RBTree these characteristics might be violated, to make the tree's structure obey the rules of a RBTree we use a function called RBTreeFixup. This function targets the problem and fixes the tree by either rotating certain sub trees and or changing the colors of nodes. Because the height of a RBTree is never more than 2log(n+1) even the worst case of insert, delete and search is *O(log n).* This year we implemented a red black tree, attached is the code.

100

Name: Clay Sarafin & Taylor Heilman

CS 271 - proj1000

1. Let a, b and c be arbitrary nodes in subtrees in α, β, γ where α is a left subtree of node x, node y is the right child of x, β is a left subtree of y and γ is a right subtree of y. Let a have a depth of d, b have a depth of e and c have a depth of f. After a left rotation has been performed on node x the depths of a and c change while the depth of b stays the same. α remains the left the subtree of x, but node x is now the left child of y, therefore the depth of a increases by one, $d + 1$. γ remains the right subtree of y but y is now the parent of x instead of being the child of x, so the depth of c decreases by one, $f - 1$. β changes from the left subtree of y to the right subtree of x, hence the depth of b doesn't change, e.



2.

3. Suppose we have a non empty Red Black Tree. By the fifth property of a Red Black Tree we know for every node x, every path to a descendent leaf contains $bh(x)$, the black height of x, black nodes. Therefore the shortest path from x to a leaf would be at least of length $bh(x)$, where the path contains at least zero red nodes. According to the fourth property of a Red Black Tree we know that if a red node has children both must be black nodes, hence red nodes can not be children of another red node.

Therefore the longest path from x to a leaf would also contain $bh(x)$ black nodes. At most, every other node in the longest path is red because the path alternates between red and black nodes, so the length of the longest path is at most $2(bh(x))$. Hence 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.

4. See Attached

5. See Attached

6. We compared the insert time of our Binary Search tree, Hash Table and Red Black Tree. On average the Red Black Tree inserted the entire movie library in 0.2182 seconds. The Hash Table was a close second with an average time of 0.2236 and the Binary Search Tree took an average 3.076 seconds. The insert time for the Binary Search Tree was much longer than the Red Black Tree and Hash Table due to the ordering of the movie files. Since the movies in the movie library were in alphabetical order, ever subsequent movie added into the BST was added to the right child due to how the Binary Search tree determined the placement of strings (the movie titles). Since every node is connected to one other node the Binary Search Tree was essentially a linked list, therefore the insert time was close to $O(n^2)$. The running time to insert into the Hash Table is largely affected by the hash function. A poor hash function may cause a large amount of movies to be chained together, making the running time of inserting a movie close to $O(n)$. Our hash function was fairly effecient which allowed for a running time slightly larger than $O(1)$ The Red Black Tree had the fastest running time due to its characteristics which helps make the tree more complete and doesn't allow the problem that the Binary Search Tree encountered with constanly inserting to the right. Therefore the insert time for our Red Black Tree was $O(logn)$.

	Red Black Tree	Hash Table	Binary Search Tree
	0.174	0.192	3.087
	0.224	0.225	3.029
	0.253	0.249	3.114
	0.242	0.238	3.132
	0.198	0.214	3.018
Average	0.2182	0.2236	3.076

```
// rb.h
// Red/Black Tree header file
// Clay Sarafin & Taylor Heilman


#ifndef RB_H
#define RB_H


template <class T>
class RBNode{
public:
        RBNode();
        RBNode(T* initValue);
        RBNode(char c);

        T* value;
        char color;

        RBNode<T>* parent;
        RBNode<T>* left;
        RBNode<T>* right;
};

template <class T>
class RBTree{
protected:
        int count;
        RBNode<T> *root;
        static RBNode<T> *nil;

        void createNode(RBNode<T>* node);

        RBNode<T>* copy(RBNode<T> *node, RBNode<T> * newP);
        void dealloc(RBNode<T> *ptr);

        RBNode<T> *getNode(RBNode<T> *ptr, const T& z);

        T *maximum_private(RBNode<T>* ptr);
        T *minimum_private(RBNode<T>* ptr);

        std::string inOrder_private(RBNode<T> *ptr, std::string str);
        std::string preOrder_private(RBNode<T> *ptr, std::string str);
        std::string postOrder_private(RBNode<T> *ptr, std::string str);


public:
        RBTree();
        RBTree(const RBTree<T>& rb);
        ~RBTree();

        RBTree<T>& operator=(const RBTree<T>& rb);

        bool empty();
        T* get(const T& z);

        void insert(T* z);
        void insertFixUp(RBNode<T>* z);
        void leftRotate(RBNode<T>* z);
        void rightRotate(RBNode<T>* z);


        T* maximum();
```

```
        T* minimum();

        T* successor(const T& z);
        T* predecessor(const T& z);

        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

        int getCount();
};

class EmptyError {};
class ExistError {};

#endif

#include "rb.cpp"
```

```
// rb.cpp
// Red/Black Tree implementation
// Clay Sarafin & Taylor Heilman


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

using namespace std;
template<class T>
RBNode<T> *RBTree<T>::nil = new RBNode<T>('b');

/*code is essentially a c/p from the BST class,
except with the appropiate names replaced (such as BSTNode -> RBNode)
as well as the appropriate NULL pointers replaced to nil*/

/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                     Node - (DE)CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*----------------------------------------------
 * Default Constructor
 * PreConditions:       n/a
 * PostConditions:      empty Node class
                        all pointers (value, left, right, parent) point nil
----------------------------------------------*/
template<class T>
RBNode<T>::RBNode(){
        value = NULL;
        left = NULL;
        right = NULL;
        parent = NULL;

        color = 'r';
}

/*----------------------------------------------
 * Construct with Item Pointer
 * PreConditions:       pointer to an value of type T
 * PostConditions:      Node class where the value pointer points to the value
                        all other pointers point to nil
----------------------------------------------*/
template<class T>
RBNode<T>::RBNode(T *initValue){
        value = initValue;
        left = NULL;
        right = NULL;
        parent = NULL;

        color = 'r';
}

/*----------------------------------------------
 * Construct with color
 * PreConditions:       char 'r' for red, 'b' for black
 * PostConditions:      Node class where the node has color c
----------------------------------------------*/
template<class T>
RBNode<T>::RBNode(char c){
        value = NULL;
        left = NULL;
```

```
        right = NULL;
        parent = NULL;

        color = c;
}

/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                        RBTree - (DE)CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*-------------------------------------------
 * Default Constructor
 * PreConditions:       n/a
 * PostConditions:      empty RBTree class created, where root points to nil
-------------------------------------------*/
template <class T>
RBTree<T>::RBTree(){
        count = 0;
        root = nil;
}


/*-------------------------------------------
 * Copy Constructor, operator=
 * PreConditions:       a RBTree
 * PostConditions:      RBTree is copied over to a new RBTree, and is in the exact same form
-------------------------------------------*/
template <class T>
RBTree<T>::RBTree(const RBTree<T>& rb){
        root = nil;
        root = copy(rb.root, nil);
}
template <class T>
RBTree<T>& RBTree<T>::operator=(const RBTree<T>& rb){
        dealloc();

        root = copy(rb.root, nil);

        return *this;
}

template <class T>
RBNode<T>* RBTree<T>::copy(RBNode<T> *node, RBNode<T> *newP){
        //will traverse through the RBTree to be copied w/ preOrder
        //instead of printing, the new node will be created and will
        //be assigned to the parent in the parameter
        if (node == nil)
                return nil;
        RBNode<T> *newNode = new RBNode<T>(node->value);
        newNode->color = node->color;
        newNode->parent = newP;
        newNode->left = copy(node->left, newNode);
        newNode->right = copy(node->right, newNode);

        return newNode;
}


/*-------------------------------------------
 * Deconstructor
 * PreConditions:       n/a
 * PostConditions:      all nodes that exists in the RBTree are deleted
-------------------------------------------*/
template <class T>
```

```
RBTree<T>::~RBTree(){
        dealloc(root);
}

template <class T>
void RBTree<T>::dealloc(RBNode<T> *ptr){
        if (ptr == nil)
                return;
        dealloc(ptr->left);
        dealloc(ptr->right);
        delete ptr;

        return;
}




/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                         RBTree - FUNCTIONS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*----------------------------------------------
 * createNode()
 * PreConditions:       a preexisting node
 * PostConditions:      pointers in the NULL that need to assigned to nil will be assigned to
nil
----------------------------------------------*/
template <class T>
void RBTree<T>::createNode(RBNode<T>* node){
        node->parent = nil;
        node->left = nil;
        node->right = nil;
}


/*----------------------------------------------
 * empty()
 * PreConditions:       n/a
 * PostConditions:      returns true if empty, false if not
----------------------------------------------*/
template <class T>
bool    RBTree<T>::empty(){
        return (count == 0);
        //alt approach: return (root == nil);
}

/*----------------------------------------------
 * get()
 * PreConditions:       tree is not empty
                        value of type T has the operator '==' and '<' overloaded
                        user has never modified what the value pointer points to in insert()
 * PostConditions:      returns a pointer to the value if it exists
                        returns nil if it does not
----------------------------------------------*/
template <class T>
T*      RBTree<T>::get(const T& z){
        if (empty())
                throw EmptyError();
        RBNode<T> *node = getNode(root, z);
        //node = nil when RBTree is empty, or if value doesn't exist
        //already covered the empty part, so logically the node doesn't exist
        if (node == nil)
                return NULL;            //does this matter now?
```

```
        return node->value;
}

template <class T>
RBNode<T>*      RBTree<T>::getNode(RBNode<T> *ptr, const T& z){
        if (ptr == nil)                 //BC: value not found, or tree is empty
                return nil;
        if (*(ptr->value) == z)                 //found value
                return ptr;
        if(z < *(ptr->value))                   //traverse tree to find value
                return getNode(ptr->left,z);
        else
                return getNode(ptr->right,z);

}

/*---------------------------------------------
 * insert()
 * PreConditions:       value of type T has the operator '<' overloaded
 * PostConditions:      value z is inserted into the tree
                        tree is still a valid RBTree
---------------------------------------------*/
template <class T>
void    RBTree<T>::insert(T *z){
        RBNode<T>* y = nil;
        RBNode<T>* x = root;

        RBNode<T>* newNode = new RBNode<T>(z);
        createNode(newNode);

        while (x != nil){ //traverse through tree to find appropriate place to insert node
                y = x;
                if (*(newNode->value) < *(x->value))
                        x = x->left;
                else
                        x = x->right;
        }

        newNode->parent = y;
        if (y == nil)                                   //tree is empty, make root point to ne
w node
                root = newNode;
        else if (*(newNode->value) < *(y->value))       //make appropriate subchild point to i
nserted node
                y->left = newNode;
        else
                y->right = newNode;

        insertFixUp(newNode);                           //call function to make the RBTree val
id again
        count++;
}

/*---------------------------------------------
 * insertFixUp()
 * PreConditions:       pointer to node inserted into the tree
 * PostConditions:      tree will be fixed to be a valid RBTree
---------------------------------------------*/
template <class T>
void RBTree<T>::insertFixUp(RBNode<T>* z){
        RBNode<T>* y;

        while (z->parent->color == 'r'){
```

```
                if (z->parent == z->parent->parent->left){
                        y = z->parent->parent->right;
                        if (y->color == 'r'){                   //only need to change colors
                                z->parent->color = 'b';
                                y->color = 'b';
                                z->parent->parent->color = 'r';
                                z = z->parent->parent;
                        }
                        else{
                                if (z == z->parent->right){     //some rotations necessary
                                        z = z->parent;
                                        leftRotate(z);
                                }
                                z->parent->color = 'b';
                                z->parent->parent->color = 'r';
                                rightRotate(z->parent->parent);
                        }
                }
                else{
                        y = z->parent->parent->left;
                        if (y->color == 'r'){                   //only have to change colors
                                z->parent->color = 'b';
                                y->color = 'b';
                                z->parent->parent->color = 'r';
                                z = z->parent->parent;
                        }
                        else{
                                if (z == z->parent->left){      //some rotations necessary
                                        z = z->parent;
                                        rightRotate(z);
                                }
                                z->parent->color = 'b';
                                z->parent->parent->color = 'r';
                                leftRotate(z->parent->parent);
                        }
                }
        }

        root->color = 'b';                                      //make root black in case it w
as changed to red
}

/*---------------------------------------------
 * leftRotate()
 * PreConditions:       pointer to node in the RBTree
 * PostConditions:      nodes in the tree will be rotated to the left such that
                        (with y being the right child of z):
                                left child of z is not modified
                                right child of y is not modified
                                z's right child is the left subtree of y
                                y's left child is z
---------------------------------------------*/
template <class T>
void RBTree<T>::leftRotate(RBNode<T>* z){
        RBNode<T>* child = z->right;

        z->right = child->left;
        if(child->left != nil)
                child->left->parent = z;
        child->parent = z->parent;
        if(z->parent == nil)
                root = child;
        else if (z == z->parent->left)
```

```
                z->parent->left = child;
        else
                z->parent->right = child;
        child->left = z;
        z->parent = child;
}

/*-----------------------------------------------
 * rightRotate()
 * PreConditions:       pointer to node in RBTree
 * PostConditions:      nodes in the tree will be rotated to the right such that
                        (with y being the left child of z):
                                right child is not modified
                                left child of y is not modified
                                z's left child is the right subtree of y
                                y's right child is z
-----------------------------------------------*/
template <class T>
void RBTree<T>::rightRotate(RBNode<T>* z){
        //code from leftRotate(), except left <-> right
        RBNode<T>* child = z->left;
        z->left = child->right;
        if(child->right != nil)
                child->right->parent = z;
        child->parent = z->parent;
        if(z->parent == nil)
                root = child;
        else if (z == z->parent->right)
                z->parent->right = child;
        else
                z->parent->left = child;
        child->right = z;
        z->parent = child;
}


/*-----------------------------------------------
 * maximum()
 * PreConditions:       tree is not empty
                        user has never modified what the value pointer points to in insert()
 * PostConditions:      returns pointer to the maximum value in the tree
-----------------------------------------------*/
template <class T>
T* RBTree<T>::maximum(){
        return maximum_private(root);
}

template <class T>
T* RBTree<T>::maximum_private(RBNode<T>* ptr){
        if (ptr == nil)//nothing exists in RBTree
                throw EmptyError();

        if (ptr->right == nil)
                return ptr->value;
        else
                return maximum_private(ptr->right);
}

/*-----------------------------------------------
 * minimum()
 * PreConditions:       tree is not empty
                        user has never modified what the value pointer points to in insert()
 * PostConditions:      returns pointer to the minimum value in the tree
```

```
-----------------------------------------*/
template <class T>
T* RBTree<T>::minimum(){
        return minimum_private(root);
}

template <class T>
T* RBTree<T>::minimum_private(RBNode<T>* ptr){
        if (ptr == nil)//nothing exists in RBTree
                throw EmptyError();

        if (ptr->left == nil)
                return ptr->value;
        else
                return minimum_private(ptr->left);
}

/*-----------------------------------------
 * sucessor()
 * PreConditions:       T z exists in RBTree
                        successor of z exists in the RBTree
                        user has never modified what the value pointer points to in insert()
 * PostConditions:      pointer to successor of z is returned
-----------------------------------------*/
template <class T>
T* RBTree<T>::successor(const T& z){
        RBNode<T> *ptr0 = getNode(root,z);
        if (ptr0 == nil)
                throw ExistError();
        RBNode<T> *ptr1;
        if (ptr0->right != nil)
                return minimum_private(ptr0->right);
        ptr1 = ptr0->parent;
        while((ptr1 != nil) && (ptr0 == ptr1->right)){
                ptr0 = ptr1;
                ptr1 = ptr1->parent;
        }

        if (ptr1 == nil)                //successor does not exist
                throw ExistError();

        return ptr1->value;
}

/*-----------------------------------------
 * predecessor()
 * PreConditions:       T z exists in RBTree
                        precedessor of z exists in the RBTree
                        user has never modified what the value pointer points to in insert()
 * PostConditions:      pointer to predecessor of z is returned
-----------------------------------------*/
template <class T>
T* RBTree<T>::predecessor(const T& z){
        RBNode<T> *ptr0 = getNode(root,z);
        if (ptr0 == nil)
                throw ExistError();
        RBNode<T> *ptr1;
        if (ptr0->left != nil)
                return maximum_private(ptr0->left);
        ptr1 = ptr0->parent;
        while((ptr1 != nil) && (ptr0 == ptr1->left)){
                ptr0 = ptr1;
                ptr1 = ptr1->parent;
```

```
        }

        if (ptr1 == nil)                //predecessor does not exist
                throw ExistError();

        return ptr1->value;
}

/*-------------------------------------------
 * inOrder(), preOrder(), and postOrder()
 * PreConditions:       n/a
 * PostConditions:      inOrder  : string representation of the values in the RBTree are in or
der
                        preOrder : string represenations of the values in RBTree are shown
                                        in pre order (starts with root, ends with max value)
                        postOrder: string represenations of the values in RBTree are shown
                                        in post order (starts with min value, ends with root)
                        all      : each value is followed by ", "
                                 : user has never modified what the value pointer points to in
 insert()
-------------------------------------------*/
template <class T>
std::string RBTree<T>::inOrder(){
        std::string str;
        str = inOrder_private(root,str);
        if (str == "")          //want to avoid using pop_back() on an empty string
                return str;
        else{
                str.pop_back();
                str.pop_back();
                return str;
        }
}
template <class T>
std::string RBTree<T>::inOrder_private(RBNode<T> *ptr, std::string str){
        if (ptr == nil){
                return str;
        }
        std::string str_null;   //null dummy string to help with recursion;
                                //if you pass in str as the parameter again, then
                                //it will append to that string in the funcion, then return th
at string
                                //it will then append it to the string in the initial call, wh
ich is something
                                //not desired

        //left, print, right
        str += inOrder_private(ptr->left, str_null);
        str += std::to_string(*(ptr->value));
        str += "-";
        str += ptr->color;
        str += ", ";
        str += inOrder_private(ptr->right, str_null);

        return str;
}


template <class T>
std::string RBTree<T>::preOrder(){
        std::string str;
        str = preOrder_private(root,str);
        if (str == "")
```

```
                return str;
        else{
                str.pop_back();
                str.pop_back();
                return str;
        }
}
template <class T>
std::string RBTree<T>::preOrder_private(RBNode<T> *ptr, std::string str){
        if (ptr == nil){
                return str;
        }
        std::string str_null;

        //print, left, right
        str += std::to_string(*(ptr->value));
        str += "-";
        str += ptr->color;
        str += ", ";
        str += preOrder_private(ptr->left, str_null);
        str += preOrder_private(ptr->right, str_null);

        return str;
}

template <class T>
std::string RBTree<T>::postOrder(){
        std::string str;
        str = postOrder_private(root,str);
        if (str == "")
                return str;
        else{
                str.pop_back();
                str.pop_back();
                return str;
        }
}
template <class T>
std::string RBTree<T>::postOrder_private(RBNode<T> *ptr, std::string str){
        if (ptr == nil){
                return str;
        }
        std::string str_null;

        //left, right, print
        str += postOrder_private(ptr->left, str_null);
        str += postOrder_private(ptr->right, str_null);
        str += std::to_string(*(ptr->value));
        str += "-";
        str += ptr->color;
        str += ", ";

        return str;
}



/*-------------------------------------------
 * getCount()
 * PreConditions:       n/a
 * PostConditions:      number of nodes that exists in the RBTree is returned
-------------------------------------------*/
template <class T>
```

```
int RBTree<T>::getCount(){
        return count;
}
```

```
// test_rb.cpp
// Red/Black Tree tests
// Clay Sarafin & Taylor Heilman

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

#include "rb.h"

using namespace std;

int *zero  = new int(0);
int *one   = new int(1);
int *two   = new int(2);
int *three = new int(3);
int *four  = new int(4);
int *five  = new int(5);
int *six   = new int(6);
int *seven = new int(7);
int *eight = new int(8);
int *nine  = new int(9);

void insertTest(){
        RBTree<int> tree;

        tree.insert(five);
        tree.insert(zero);
        assert(tree.getCount() == 2);
        assert(tree.preOrder() == "5-b, 0-r");  //tests for empty tree, left subtree
        tree.insert(nine);
        assert(tree.preOrder() == "5-b, 0-r, 9-r");     //test for right subtree;
        //add everything else
        tree.insert(one);
        tree.insert(two);
        tree.insert(three);
        tree.insert(four);
        tree.insert(six);
        tree.insert(seven);
        tree.insert(eight);
        assert(tree.preOrder() == "5-b, 1-r, 0-b, 3-b, 2-r, 4-r, 7-r, 6-b, 9-b, 8-r");
}

void getTest(){
        RBTree<int> tree;

        try{
                tree.get(*one);
                assert(false);
        }
        catch(EmptyError){
        }

        tree.insert(one);
        tree.insert(two);
        tree.insert(three);
        tree.insert(four);
        tree.insert(six);
        tree.insert(seven);
        tree.insert(eight);
```

```
        //test for root
        assert(tree.get(*one) == one);
        //test for non-leaf
        assert(tree.get(*two) == two);
        //test for leaf
        assert(tree.get(*eight) == eight);      //not sure what's going on here

        //test for nonexisting node in tree;
        assert(tree.get(*five) == NULL);
}

void maxMinTest(){
        RBTree<int> tree;

        //test for EmptyError exceptions
        try{
                tree.minimum();
                assert(false);
        }
        catch(EmptyError exception){
        }
        try{
                tree.maximum();
                assert(false);
        }
        catch(EmptyError exception){
        }

        tree.insert(zero);
        tree.insert(one);
        tree.insert(two);

        //test min & max
        assert(tree.maximum() == two);
        assert(tree.minimum() == zero);
}

void predSuccTest(){
        RBTree<int> tree;

        //test ExistError exceptions
        try{
                tree.predecessor(0);
                assert(false);
        }
        catch(ExistError exception){
        }
        try{
                tree.successor(0);
                assert(false);
        }
        catch(ExistError exception){
        }

        tree.insert(four);
        tree.insert(two);
        tree.insert(three);
        tree.insert(one);
        tree.insert(seven);
        tree.insert(five);
        tree.insert(nine);
```

```
        assert(tree.predecessor(2) == one);
        assert(tree.successor(2) == three);

        assert(tree.predecessor(4) == three);
        assert(tree.successor(4) == five);

        assert(tree.predecessor(9) == seven);
        //test EmptyError exception for an item that doesn't exist
        try{
                tree.successor(9);;
                assert(false);
        }
        catch(ExistError exception){
        }

        assert(tree.predecessor(3) == two);
        assert(tree.successor(3) == four);

}

void printTest(){
        RBTree<int> tree;
        assert(tree.inOrder() == "");
        assert(tree.preOrder() == "");
        assert(tree.postOrder() == "");

        tree.insert(zero);
        tree.insert(one);
        tree.insert(two);

        assert(tree.inOrder() == "0-r, 1-b, 2-r");
        assert(tree.preOrder() == "1-b, 0-r, 2-r");
        assert(tree.postOrder() == "0-r, 2-r, 1-b");

        RBTree<int> tree2;

        tree2.insert(four);
        tree2.insert(two);
        tree2.insert(three);
        tree2.insert(one);
        tree2.insert(seven);
        tree2.insert(five);
        tree2.insert(nine);
        /*
                        3-b
                     /       \
                2-b             5-r
             /               /       \
        1-r             4-b             7-b
                                            \
                                                9-r
        */
        assert(tree2.inOrder() == "1-r, 2-b, 3-b, 4-b, 5-r, 7-b, 9-r");
        assert(tree2.preOrder() == "3-b, 2-b, 1-r, 5-r, 4-b, 7-b, 9-r");
        assert(tree2.postOrder() == "1-r, 2-b, 4-b, 9-r, 7-b, 5-r, 3-b");

}

void copyTest(){
        RBTree<int> tree0;

        tree0.insert(four);
        tree0.insert(two);
```

```
        tree0.insert(three);
        tree0.insert(one);
        tree0.insert(seven);
        tree0.insert(five);
        tree0.insert(nine);

        RBTree<int> tree1(tree0);

        assert(tree1.inOrder() == "1-r, 2-b, 3-b, 4-b, 5-r, 7-b, 9-r");
        assert(tree1.preOrder() == "3-b, 2-b, 1-r, 5-r, 4-b, 7-b, 9-r");
        assert(tree1.postOrder() == "1-r, 2-b, 4-b, 9-r, 7-b, 5-r, 3-b");
}

int main(){
        insertTest();
        getTest();
        maxMinTest();
        predSuccTest();
        printTest();
        copyTest();

        delete zero;
        delete one;
        delete two;
        delete three;
        delete four;
        delete five;
        delete six;
        delete seven;
        delete eight;
        delete nine;
        return 0;
}
```

```
// dict.h
// header file for dictoinary class
// Clay Sarafin & Taylor Heilman

#ifndef DICT_H
#define DICT_H

#include <iostream>

#include "rb.h"

using namespace std;

template <class KeyType>
class Dictionary : public RBTree<KeyType>{
        public:
                Dictionary() : RBTree<KeyType>() {};

                using RBTree<KeyType>::get;
                using RBTree<KeyType>::insert;
                using RBTree<KeyType>::count;
                using RBTree<KeyType>::inOrder;
                using RBTree<KeyType>::empty;


                std::string toString();
};

#include "dict.cpp"

#endif
```

```
// dict.cpp
//
// Clay Sarafin & Taylor Heilman

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


using namespace std;


/*-------------------------------------------
 * toString()
 * PreConditions:       inOrder from BST class does not have brackets, every entry is followed
 by ", "
                                and does not end with std::endl
                      type KeyType's operator<< is overloaded, and is formatted as "key:valu
e"
 * PostConditions:     string representation of the entire dictionary is printed out
-------------------------------------------*/

template <class KeyType>
std::string Dictionary<KeyType>::toString(){
        std::string str;
        str += "{";
        str += inOrder();
        str += "}";

        return str;
}
```

Priority Queues and Binary heaps

Proficiency: Mastery with Distinction

Heaps are a tree based data structure. Each tree is made up of nodes and each node has at most two children. A binary heap is efficient because it is either a complete tree, or in the row of nodes that is not complete, the left most node up to a point is filled. This allows for an insert and delete time of *O(log n).* A heap can either be a Max or Min heap. In a min heap, the children of a node must be bigger than the parent. In a max heap, the children must be smaller than the parent. This means that the root either holds the max or min value in the heap, depending upon the type of heap. From now on I speak in terms of a min heap. This characteristic makes heaps great from implementing Priority Queues. When a new node is inserted into a heap, the node is immediately placed at the last spot in the heap (the lowest, right most node). Obviously after an insert the characteristic of children being bigger than its parent may be affected. To fix this we call a secondary function called heapify to make the tree into a min heap again. To get the newly inserted node to the correct position heapify compares the node to its parent. If the child is smaller than its parent they switch spots and heapify is recursively called until the node is properly positioned. Because the height of a min heap is alway log n, heapify runs on *O(log n)*. A Heap's structure is also useful for sorting, since the smallest is always the root. To sort a sequence of numbers into ascending order simply build a min heap, swap the root with the last node in the heap, append the old root into a new list and delete it from the heap, then replace the root with

its largest child, and do the same to the nodes below the new root. As I said previously, heaps makes great priority queues.

A priority queue is a queue with elements where each element has a priority associated with it. A min heap is used to represent a min priority queue. The element with the lowest (most important ex. #1 customer) priority value is the root of the min heap. Min priority queues are used often in business and for graphing algorithms such as Prim's algorithm. For Prim's algorithm the weight of an edge is used as the priority of that edge, hence the least weighted edges are examined first. Min priority queues allow us to find the minimums of thins, while max priority do the opposite. This year we implemented a min priority queue by inheriting a min heap. Using the heap's tree structure we then implemented Huffman coding by generating prefix codes using the unique path from a root to a leaf in tree's. As you can see heaps and priority queues have many uses and are a vital structure to know.

```
// 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();              // default constructor
    MinPriorityQueue(int n);         // construct an empty MPQ with capacity n
    MinPriorityQueue(const MinPriorityQueue<KeyType>& pq);  // copy constructor

    KeyType* minimum() const;                    // 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
    bool find(KeyType* key);
    string findCode(KeyType* key, int lenght);

    // 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
class IndexError { };     // MinPriorityQueue key exception

#include "pq.cpp"

#endif
```

88

call MinHeap constructors via initializer lists

```
#include "pq.h"
```

pq.cpp included from pq.h in template class

```
/*
 *      Default Constructor
 *      Precondition:
 *      Postcondition:
 */
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue()
{

}



/*
 *      Construct an empty MinPriority Queue with capacity n
 *      Precondition:
 *      Postcondition:
 */
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(int n)
{
        MinHeap<KeyType> heap(n);
}



/*
 *      Copy Constructor
 *      Precondition: MinPriorityQueue pq  must be a legitimate MinPriorityQueue
 *      Postcondition: The target MinPriorityQueue is a copy of the other MinPriorityQue
 */
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(const MinPriorityQueue<KeyType>& pq)
{
        heapSize = pq.heapSize;
        capacity = pq.capacity;

        A = new KeyType*[pq.capacity];

        for (int i = 0; i < pq.capacity; i++)
        {
                A[i] = pq.A[i];
        }
}
```

call MinHeap copy constructor

```
/*
 *      Return the Minimum Element
 *      Precondition: A valid MinPriorityQueue with Size >= 0
 *      Postcondition: Returns the smallest element in the Queue
 */
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::minimum() const
{
        if (heapSize <= 0)
        {
                throw EmptyError();
        }
```

```
        else
        {
                return A[0];
        }
}



/*
 *      Delete the Minimum Element and return it
 *      Precondition: A valid MinPriorityQueue with Size >= 0
 *      Postcondition: Returns the smallest element, deletes said element, and keeps a valid M
in Heap
 */
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;
        }
}




/*
 *      Decrease the value of an element
 *      Precondition: A valid MinPriorityQueue with Size >= 0,
 *      Postcondition: The element at the inputted index has the value of the inputted key
 */
template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey(int index, KeyType* key)
{
        if (index >= heapSize || index < 0)
        {
                throw IndexError();
        }
        if (key > A[index])
        {
                throw KeyError();
        }
        else
        {
                A[index] = key;
                while (index > 0 && *A[parent(index)] > *A[index])
                {
                        swap(index, parent(index));
                        index = parent(index);
                }
        }
```

```
}



/*
 *      Insert a New Element
 *      Precondition: a valid MinPriorityQueue with Heapsize != capacity
 *      Postcondition: a valid MinPriorityQueue containing the inputted key
 */
template <class KeyType>
void MinPriorityQueue<KeyType>::insert(KeyType* key)
{
        if (heapSize == capacity)
        {
                throw FullError();
        }
        else
        {
                A[heapSize] = key;
                heapSize++;
                int index = heapSize-1;
                decreaseKey(index, key);


        }
}


template <class KeyType>
bool MinPriorityQueue<KeyType>::find(KeyType* key)
{
        for(int i = 0; i < heapSize; i++)
        {
                if (A[i]->name == key->name)
                        return true;

        }

        return false;
}


template <class KeyType>
string MinPriorityQueue<KeyType>::findCode(KeyType* key, int length)
{
        string codenum;
        //cout << heapSize << endl;
        for(int i = 0; i < length; i++)
        {
                //cout << "name = " << A[i]->name << endl;
                if (A[i]->name == key->name)
                {
                        //cout << "got here" << endl;
                        codenum = A[i]->code;
                        return codenum;
                }
        }
        return "NIF";
}
```

these are not needed & facilitate an inefficient algorithm — see below.

```
/*
 *      Return whether the MPQ is empty
 *      Precondition: A valid MinPriorityQueue with Size >= 0
 *      PostCondition:  returns true id heapsize = 0, returns false if heapsize > 0
 */
template <class KeyType>
bool MinPriorityQueue<KeyType>::empty() const
{
        return heapSize == 0;
}



/*
 *      Return the numbers of keys
 *      Precondtion: a Valid MPQ
 *      Postcondition: returns the length of MPQ
 *
 */
template <class KeyType>
int MinPriorityQueue<KeyType>::length() const
{
        return heapSize;
}



/*
 *      return a string representation of the MPQ
 *      Precondtion: a valid MPQ
 *      Postcondition: a printed string of the MPQ in a list
 */

template <class KeyType>
std::string MinPriorityQueue<KeyType>::toString() const
{
        std::ostringstream sstream;

        sstream << "[";
        for (int i = 0; i < heapSize; i++)
        {
                sstream << *(A[i])  << " ";
        }

        std::string s = sstream.str();
        string st = s.substr(0, s.size()-1);
        st.append("]");
        return st;
}
```

```
// test_pq.cpp
// Testing if our stuff works
#include <iostream>
#include <stdlib.h>
#include <assert.h>
#include "pq.h"

using namespace std;

void test_pqInsert()
{
        MinPriorityQueue<int> west(10);
        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        assert(west.toString() == "[2,5,24,14]");
        cout << "Insert Assertion Passed" << endl;
}

void test_pqMin()
{
        MinPriorityQueue<int> west(10);
        try
        {
                west.minimum();
                assert(false);
        }
        catch(EmptyError x)
        {
                cout<< "caught" << endl;
        }

        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        assert(*west.minimum() == 2);
        cout << "Minimum Assertion Passed" << endl;
}

void test_pqExtractMin()
{
        MinPriorityQueue<int> west(10);
        try
        {
                west.extractMin();
```

```
                assert(false);
        }
        catch(EmptyError y)
        {
                cout<< "caught2" << endl;
        }


        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        assert(*west.extractMin() == 2);
        assert(west.toString() == "[5,14,24]");
        cout << "Extract Minimum Assertion Passed" << endl;
}

void test_pqEmpty()
{
        MinPriorityQueue<int> west(10);
        assert(west.empty() == true);
        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        assert(west.empty() == false);
        cout << "Empty Assertion Passed" << endl;
}

void test_pqLength()
{
        MinPriorityQueue<int> west(10);
        assert(west.length() == 0);
        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        assert(west.length() == 4);
        cout << "Length Assertion Passed" << endl;
```

```
}

void test_pqCopy()
{
        MinPriorityQueue<int> west(10);
        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        MinPriorityQueue<int> east(west);
        assert(east.toString() == "[2,5,24,14]");
        cout << "Copy Assertion Passed" << endl;
}


void test_pqDecreasekey()
{
        MinPriorityQueue<int> west(10);
        int *x = new int;
        *x = 5;
        int *p = new int;
        *p = 24;
        int *s = new int;
        *s = 14;
        int *q = new int;
        *q = 2;
        int *z = new int;
        *z = 9;
        try
        {
                west.decreaseKey(9,z);
                assert(false);
        }
        catch(IndexError t)
        {
                cout << "caught3" << endl;
        }
        west.insert(x);
        west.insert(s);
        west.insert(p);
        west.insert(q);
        MinPriorityQueue<int> east(west);
        cout << west.toString() << endl;
        assert(east.toString() == "[2,5,24,14]");
        cout << "Decreasekey Assertion Passed" << endl;


}


int main()
{
        test_pqInsert();
        test_pqMin();
        test_pqExtractMin();
```

```
        test_pqEmpty();
        test_pqLength();
        test_pqCopy();
        test_pqDecreasekey();
        return 0;
}
```

```
#include "pq.h"

using namespace std;

class Node {
  public:
        char name;                  // character
    int f;                                  // frequency
        Node *left, *right;             // pointers to the left and right child
    string code;                    // prefix code

        Node()              //default constructor
    {
        name = (char) 0;
        code = "";
        f = 0;
        left = right = 0;
    }

    Node(int ff, char n = (char) 0)     //constructor that creates a node
    {
        name = n;
        f = ff;
        left = right = 0;
        code = "";
    }

    ~Node()                             //destructor
    {
        delete left;
        delete right;
    }


    bool operator<(Node &comp)
    {
        return f < comp.f;
    }

    bool operator>(Node &comp)
    {
        return f > comp.f;
    }

    bool operator==(Node &comp)
    {
        return f  == comp.f;
    }


        static void display(Node*, bool);
        static void encode(Node*);


    friend ostream& operator <<(ostream&, Node&);
};
```

NULL

Don't deallocate in destructor because not allocated in constructor.

```
#include "node.h"
#include <stdlib.h>
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
using namespace std;


string codes;
Node *root;
string head;

/*
 *              Searches the Minimum Priority Queue for a node that matches the character give
n and adds its code to the codes global variable
 *              Precondition: A valid MPQ must exist, and name must be a valid char
 *              Postcondition: The correct corresponding code of the char given is added to th
e codes global variable
 */
void searchName(char name, Node* node, bool leaf = 1)
{

    if (node == 0)
    {
        return ;
    }

    searchName(name, node->left, leaf);

    if (leaf == 1)
    {
        if (node->name == name)         //no children, is a leaf
        {
                codes += node->code;
        }
    }

    searchName(name,node->right, leaf);
}




/*
 *              Creates the character key that appears in our compressed file
 *              Precondition: A valid Huffman coded MPQ exists
 *              Postcondition: The header string is filled with character keys
 */
void header(Node* node, bool leaf = 1)
{
    if (node == 0)
    {
        return;
    }

    header(node->left, leaf);

    if (leaf == 1)
    {
        if (node->left == 0 && node->right == 0)        //no children, is a leaf
        {
```

```
                string f = to_string(node->f);
            char name = node->name;
            string code = node->code;
            head = head + name + "[" + f + "]" + "(" + code + ")";
        }
    }

    header(node->right, leaf);
}
```

```
/*
 *              Decompressed the given sourcefile, and creates a new file with the contents of
 the decompressed file
 *              Precondition: compressed is a valid sourcefile, and output is a valid name of
an output file
 *              Postcondition: the source file is decompressed and the output file contains th
e decompressed contents
 */
void decompress(string compressed, string output)               //compressed file, output file
{
        string str;
        string decomp;
        bool child;
        int i = 0;
        int j=0;
        int bit;
        string fcode;
        string bits;
        unsigned char buffer;

        ifstream file(compressed);              // sourcefile

        while(getline(file, str))
        {
                decomp += str;          // read in source file
        }


        int unique =0;
        int realnum;
        int t = 0;
        string name;
        string code;

        while(t<1)
        {

                if (decomp[i+1] = '[')
                {
                        name = name+decomp[i];                  //NAME
```

? use well-named boolean to make this clearer

```
                i=i+2;
                while(decomp[i] != ']')
                {
                        int num = num + decomp[i];
                        i++;
                }

                i=i+2;
                while(decomp[i] != ')')
                {
                        code = code+decomp[i];
                        i++;
                }
                code += ',';

        }
unique++;
        i++;


        if( decomp[i] == '$' &&  decomp[i+1] == '$')
                t = 1;
}

int namelen = name.length();
string aname[unique];
string acode[unique];
for (int h = 0; h < namelen; h++)
{

        aname[h] =  name[h];
}



int comma =0;
for (int j = 0; j < unique; j++)
{
        string codearray = "";
        while(code[comma] != ',')
        {
                codearray = codearray + code[comma];
                comma++;
        }
        acode[j] =  codearray;
        comma++;
}



int len2 = decomp.length();


i = i+2;

for( i; i< len2; i++)
{
        buffer = decomp[i];

        for(j = 0; j < 8; j++)
        {
                bit = buffer >>7;
                string stringbit = to_string(bit);
```

This is very hard to follow — use better variable names & comments!

```
                        bits = bits+ stringbit;
                        buffer = buffer << 1;
                }

        }


        ofstream outputFile;
        outputFile.open (output);


        int count = 0;
        int bitsL = bits.length();
        while (count < bitsL)
        {
                bool match = false;
                string check = "";
                while (!match)
                {
                        if ((bits[count + 1] != '0') && (bits[count + 1] !=  '1'))      //If t
he input is over, quit the loop
                        {
                                return;
                        }
                        check = check + bits[count];
                        for (int k = 0; k < unique; k++)
                        {
                                if (check == acode[k])
                                {
                                        outputFile << aname[k];
                                        match = true;
                                }
                        }
                        count++;
                }
        }

}




/*
 *              Compresses the give sourcefile given the root node
 *              Precondition: A valid sourcefile and root node is passed int
 *              Postcondition: The output file is a correctly compressed version of the source
 file.
 */
void compress(Node* node, string input, int length, string output)
{
        char item;
        unsigned char buffer = 0;
        int i = 0;
        int count = 0;
        int num = 0;
        ofstream outputFile;
        outputFile.open (output);                       //should be *argv[3]
        string precode;

        header(root);           //creates header
        head = head + '$';
```

```
        head = head + '$';

        outputFile << head;             //send header to output file
        outputFile << "\n";


        while(input[i] != ''')                          //traversing source file
        {

                item = input[i];
                searchName(item, node, 1);                              //adds
 input[i] code to 'codes'

                i++;
        }



        int eight=  codes.length();
        int remainder = eight % 8;
        eight = 8-remainder;


        i = 0;          //reset i


        while(codes[i] == '0' || codes[i] == '1')               //iterate through codes string
        {
                if (codes[i] == '1')
                {
                        buffer = buffer << 1;           //shift left
                        buffer = buffer | 1;                    //add 1 to end of bufffer
                        count ++;
                        num++;
                }

                else
                {
                        buffer = buffer << 1;           //shift left
                        buffer = buffer | 0;            // add 0 to end of buffer
                        count ++;
                        num++;
                }


                if(count == 8)                          //8 bits of buffer have been filled
                {
                        outputFile << buffer;           //add buffer to mid file
                        buffer = 0;
                        count = 0;
                }

                i++;
        }


        i = 0;
        if (count > 0 )
        {
                for (i=0; i< eight; i++)
                {
                        buffer = buffer << 1;           //shift left
```

↑ root

- Codes should not be global – side effect makes code harder to follow
- codes can be a really long string – problem for large files

-3

Factor out common code

does nothing

```
                    buffer = buffer | 0;                    // add 0 to end of buffer
                }
                outputFile << buffer;           //add buffer to mid file with extra 0's
        }

        outputFile.close();

}
```

```
/*
 *              Creates a string representation of the tree
 *              Precondition: a valid tree with a root node exists
 *              Postcondition: A correct representation of the tree is displayed
 */
void Node::display(Node* node, bool leaf = 1)
{
    if (node == 0)
    {
        return;
    }

    display(node->left, leaf);

    if (leaf == 1)
    {
        if (node->left == 0 && node->right == 0)        //no children, is a leaf
        {
            cout << *node << ", ";
        }
    }
    else
    {
        cout << *node << ", ";
    }

    display(node->right, leaf);
}
```

Should be in a node.cpp file.

```
/*
 *              Creates the codes for each character from the tree
 *              Precondition: a valid tree with a root exists
 *              Postcondition: Every node in the tree has a correct code made of 0's and 1's
 */
void Node::encode(Node* node)           // determines prefix code for each char
{
        if (node == 0)
        {
        return;
    }
```

```
    if (node->left != 0)        //checking node isn't a leaf
    {
        node->left->code = node->code + "0";    //left child adds code 0
                encode(node->left);                                         //call encode(
);
    }
    if (node->right != 0)       //checking node isn't a leaf
    {
        node->right->code = node->code + "1";  //right child adds code 1
        encode(node->right);
    }
}

/*
 *              Overloads the << operator in order to print out a string representation of the
 node
 */
ostream& operator <<(ostream &out, Node &node)
{
    out << node.name << "(" << node.f << ")" << ":" << node.code;
    return out;
}




/*
 *              The huffman code that creates the tree out of the MPQ
 *              Precondition: a valid MPQ is inputted
 *              Postcondition: A compressed file is created
 */

                                // pq with nodes and chars  # of nodes  input file    output f
ile
void HuffmanCode(MinPriorityQueue<Node>* q, int length, string input, string output)
{

    for (int j = 0; j < length - 1; j++)
    {
        Node *left = q->extractMin();                   //Node x = smallest freq char in PQ
        Node *right  = q->extractMin();                 //Node y = 2nd smallest freq char in P
Q
        Node *z = new Node(left->f + right->f, '#');    // parent node of nodeLeft and nodeRig
ht
        z->left = left;                             //connected by 0 edge
        z->right = right;                           // connected by 1 edge
        q->insert(z);                                                   // insert into PQ
    }

    root = q->extractMin();             //most frequent char in file


   cout << "Full tree (inorder):\n";
   Node::display(root, 0);                                  //Tree with nodes


    Node::encode(root);                                     // determine prefix codes based off tr
ee

        cout << "\nHuffman Code:\n";
        Node::display(root);
    cout << "\n";
```

```
        compress(root, input, length, output);

}



/*
 *              Creates the nodes for the MPQ out of the input file
 *              Precondition: A valid sourcefile exists
 *              Postcondition: A valid compressed outfile with the given name is created
 */
void begin(string input, string output)
{
        int length = 0;
        int i = 0;
        char sentinal = '‘';
        string str;
        string source;

        ifstream file(input);           // sourcefile

        while(getline(file, str))
        {
                source += str;          // read in source file
        }
        source += "‘";          //insert sentinal character

        while(source[i] != sentinal)
        {
                length++;       // find length of file
                i++;

        }

    MinPriorityQueue<Node> *t = new MinPriorityQueue<Node> (length); //create new PQ
        int unique = 0;         // amount of unique chars in source file


        for(int j=0; j<length; j++)
        {
                char name = source[j];                  //looking at character in source at in
dex j
                Node *n = new Node(0, name);    // create new node
                if (!t->find(n))                                // check if node is already in
 PQ
                {
                        int frequency = 0;                      // node isn't in PQ
                        for(int k = j;k<length;k++)
                        {
                                if(name == source[k])           // find frequency of char in s
ource file
                                {
                                        frequency++;
                                }
                        }
                        t->insert(new Node(frequency, name));   //insert node into PQ with fre
```

problematic if file won't fit in memory!

source.length() ?

This is really inefficient — just pass over input once & don't search MPQ.

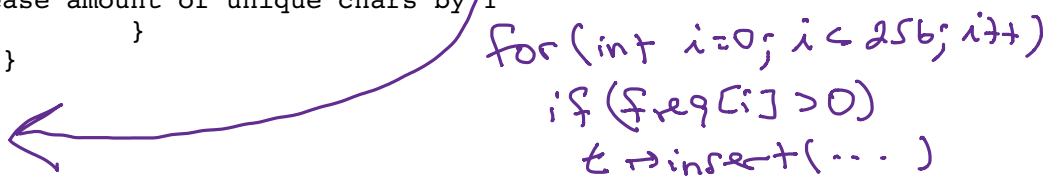
memory leak!

int freq[256];

freq[name]++;

```
q, char
                        unique++;
// increase amount of unique chars by 1
                }
        }

        HuffmanCode(t, unique, source, output); //call HuffmanCode (PQ t, chars #, source, out
put)
}
```




```
int main(int argc, char** argv)
{
        if (argc != 4)
        {
                return 1;       //not properly inputted
        }

        else
        {
                string arg1 = argv[1];
                if(arg1 == "-c")        //encode .txt file, and creating .huff file
                {
                        string sourceFile = argv[2];
                        string outputFile = argv[3];
                        begin(sourceFile, outputFile);
                }
                else if (arg1 == "-d")  //decompress, decode
                {
                        string sourceFile = argv[2];
                        string outputFile = argv[3];
                        decompress(sourceFile, outputFile);
                }
                else
                {
                        return 1; // improper input
                }
        }


        return 0;
}
```

Analysis of Algorithms

Proficiency: Mastery

It is an important skill to be able to examine an algorithm and be able to tell the runtime of said algorithm. This is a skill that we've been working on throughout the year. To determine run time an algorithm you ignore any operation that takes constant time and only focus on the raw components of the algorithm. This way of analyzing run time focuses more on the algorithm and less on the speed of the computer it is being run on. To find the asymptotic runtime of an algorithm we examine the algorithm when the size of the problem goes to infinity. This is known as Big O notation. Once we find the asymptotic runtime of an algorithm we can prove its correctness using induction. Like any induction proof, we solve the base case of the algorithm, with its input n is minimum. Then we do the induction hypothesis where we assume that for arbitrary n, $T(n) \leq n$. Lastly we prove $T(n+1) \leq n+1$. Using this format we are able to prove the runtime of algorithms. We can prove loops in a similar fashion.

A loop invariant is a property of a loop that holds true before and after each iteration of the loop. To prove a loop invariant we first show that the invariant holds prior to the first iteration. Then we assume the invariant holds after some iteration k of the loop. If we show that the invariant holds after the k+1 iteration of the loop we have proved the loop invariant using induction. While examining and proving algorithms is easy to do in practice, messy, complicated code is a reality we all have dealt with. To help make code more understandable it is important to comment with proper pre and post conditions.

A precondition is a condition that must be true of the parameters of a function **before** running the code in the function. A post condition is a condition that is true **after** the code has been run.

```
/*-------------------------------------------
 * link()
 * PreConditions:       pointers to nodes x and y
 * PostConditions:      a link between the nodes x and y is formed
-------------------------------------------*/
template<class T>
void DisjointSets<T>::link(DSNode<T>* x, DSNode<T>* y){
        if (inForest(x) == false)
                throw NotFoundError();
        if (inForest(y) == false)
                throw NotFoundError();
        //compare ranks of x and y; connect based on rank (smaller rank gets connected to larg
er rank)
        //if ranks are the same, make add 1 to x's rank
        if (x->dsrank < y->dsrank)
                x->parent = y;
        else{
                y->parent = x;
                if (x->dsrank == y->dsrank)
                        x->dsrank = x->dsrank + 1;
        }
}
```

An example of proper pre and post conditions is above. As long as the link function gets pointers to nodes x and y (precondition) the post condition will always be true. In every project completed this year, every function was properly commented with pre and post conditions. This allows editors of the code to understand what we, the programmers, were thinking as we were coding. Attached are examples of proving a loop invariant and proving time complexity.

1-29-16 Proving loop invariant

Proofs
- Initialization (Base case)
- Maintenance (Induction hypothesis & step)

State termination condition

Insertion Sort:

While loop: Before each iteration of the while loop, $A[i+1..j] \geq key$ & $A[i+2..j] \leftarrow A[i+1..j-1]$

for loop: Before iteration j of the for loop, $A[0..j-1]$ contains sorted elements originally in $A[0..j-1]$

Initialization - Prove that before the first iteration of the while loop, $A[j-1+1..j] \geq key$ and $A[j-1+2..j] = A[j+1..j] \leftarrow A[j-1+1..j-1] = A[j..j-1]$

The first part is true because $A[j] = key$.
The second part is vacuously true.

Maintenance - Assume the loop invariant is true before some iteration of the while loop. that is, that $A[i+1..j] \geq key$ & $A[i+2..j] \leftarrow A[i+1..j-1]$

Since we are starting an iteration of this loop, we know that $A[i] > key$ is true
Combing this w/ our assumption, we know $A[i..j] \geq key$
Also, since $A[i+1] = A[i]$, by combining this with the 2nd part of our assumption, we know that $A[i+1..j] \leftarrow A[i..j-1]$

Before iteration j of the for loop, A[0...j-1] contains the elements originally in A[0...j-1] in sorted order.

Proof:

Initialization:

Before the first iteration A[0...-1-1] contains the elements originally in A[0...-1-1] in sorted order. True

Maintenance:

Assume loop invariant is true before some iteration j, that is A[0...j-1] contains the elements originally in A[0...j-1] in sorted order. In particular A[0...i] & A[i+1...j] is sorted. By the loop invariant of the while loop, A[i+2...j] ≤ A[i+1...j-1]; Therefor A[i+2...j] is sorted. Also by the inner loop invariant, key ≤ A[i+2]
After the while loop, either:

(a) A[i] ≤ key, since A[i+1] = key & A[i] ≤ key ≤ A[i+2], A[0...j] is sorted

(b) i=-1, It must be that A[0...j] > key
Since A[0] = key, we know that A[0...j] is sorted

Before the next iteration, j is incremented. Therefore A[0...j-1] is sorted

Termination: A[0...n-1] contains ..... in sorted order

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

Proof By Induction

Base: let $n=1$

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

Induc. Hyp: Assume that $T(k) = ak + ck\log_2 k$ for all $k = 1, 2 \cdots n-1$

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

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

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

$= an + cn(\log_2(n) - \log_2(2)) + cn$

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

$= an + cn\log_2(n)$

Fin