Table of Contents

[UML 3](#_Toc529476539)

[Detailed Design 4](#_Toc529476540)

[Function: main() 4](#_Toc529476541)

[**Function:** CreateListRandomInts(int array[]) 5](#_Toc529476542)

[Function: shuffle(int values[], int size) 6](#_Toc529476543)

[Function: CreateTree(int length, int randArray[]) 6](#_Toc529476544)

[**Function:** CreateHashTable(int length, int temp[]) 7](#_Toc529476545)

[**Function:** CreateSortedHashTable (int length, int temp[]) 7](#_Toc529476546)

[**Function:** *BSTTrial* *(BST tree)* 8](#_Toc529476547)

[**Function:** *HashTableTrial* *(HashTable table)* 9](#_Toc529476548)

[**Function:** *SortedHashTableTrial(HashTable table)* 10](#_Toc529476549)

[**Class:** HashTable 11](#_Toc529476550)

[**Method:** *Hash(int key)* 11](#_Toc529476551)

[**Method:** void Insert(int key) 11](#_Toc529476552)

[**Method:** void InsertSorted(int key) 11](#_Toc529476553)

[**Method:** bool Search(int key) 12](#_Toc529476554)

[**Method:** int numCompares(int key) 12](#_Toc529476555)

[**Method:** int numComparesSorted(int key) 12](#_Toc529476556)

[**Method:** void PrintTable() 13](#_Toc529476557)

[**Method:** int ReturnLength() 13](#_Toc529476558)

[**Class:** LinkedList 14](#_Toc529476559)

[**Method:** void Insert(int data) 14](#_Toc529476560)

[**Method:** void InsertSorted(int data) 15](#_Toc529476561)

[**Method:** bool Search(int key) 16](#_Toc529476562)

[**Method:** int NumComparesSorted(int key) 16](#_Toc529476563)

[**Method:** int NumCompares(int key) 17](#_Toc529476564)

[**Method:** void PrintList() 17](#_Toc529476565)

[**Class:** BST 18](#_Toc529476566)

[**Method:** void InorderTraverse(NodePtr node) 18](#_Toc529476567)

[**Method:** void Insert(int item) 19](#_Toc529476568)

[**Method:** NodePtr Search(NodePtr x, int item) 20](#_Toc529476569)

[**Method:** NodePtr FindPredecessor(int item) 20](#_Toc529476570)

[**Method:** NodePtr FindSucessor(int item) 21](#_Toc529476571)

[**Method:** NodePtr NumComparisons(NodePtr x, int item, int& numComparisons) 22](#_Toc529476572)

[**Method:** NodePtr ReturnRoot() 22](#_Toc529476573)

[**Method:** int ValueofNode(NodePtr node) 23](#_Toc529476574)

[**Method:** int ReturnLength() 23](#_Toc529476575)

[**Method:** NodePtr FindMin(NodePtr node) 23](#_Toc529476576)

[**Method:** NodePtr FindMax(NodePtr node) 24](#_Toc529476577)

[**Method:** void BurnForest(NodePtr node) 24](#_Toc529476578)

[**Method:** void CreateTree(NodePtr node) 25](#_Toc529476579)

[Test Plan 26](#_Toc529476580)

[Function Prototype: main() 26](#_Toc529476581)

[Class: BST 27](#_Toc529476582)

[Class: HashTable 29](#_Toc529476583)

[Class Specs and Implementation 30](#_Toc529476584)

[Hash Table.h 30](#_Toc529476585)

[Hash Table.cpp 31](#_Toc529476586)

[LinkedList.h 32](#_Toc529476587)

[LinkedList.cpp 33](#_Toc529476588)

[BST.h 37](#_Toc529476589)

[BST.cpp 38](#_Toc529476590)

[NodeT.cpp(for BST) 42](#_Toc529476591)

[Client Source Code 43](#_Toc529476592)

[Building Program 52](#_Toc529476593)

[Test Plan Output 53](#_Toc529476594)

[Program Output 55](#_Toc529476595)

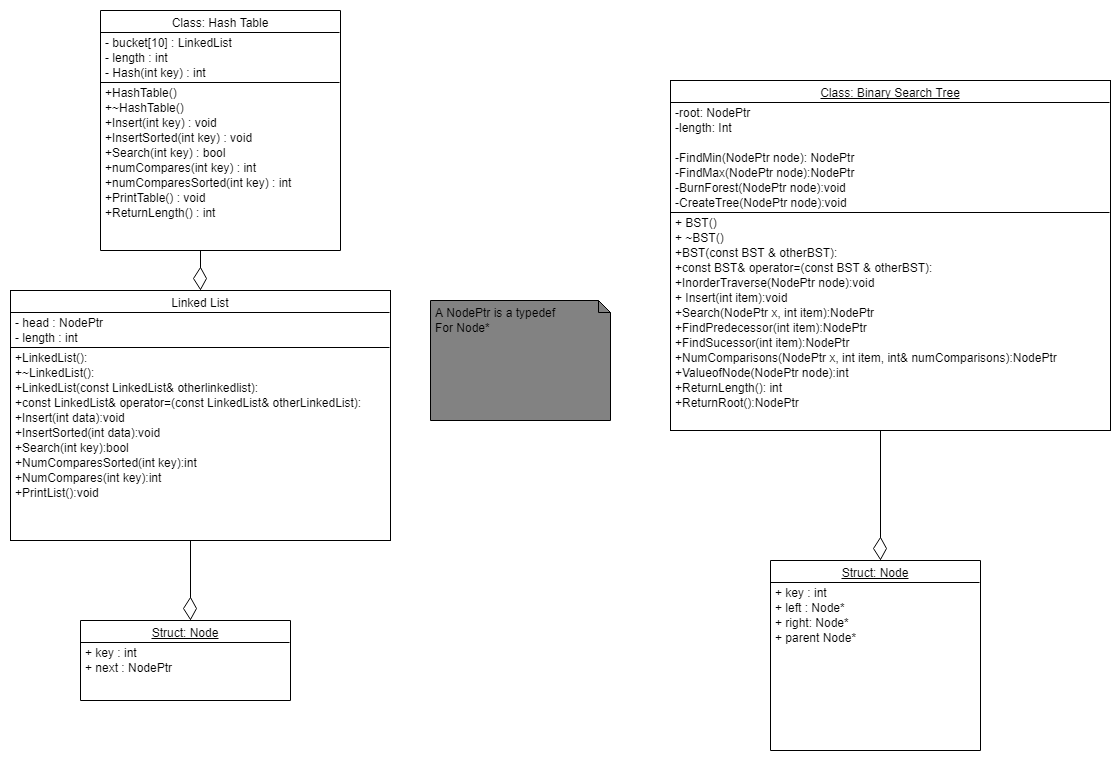
[#2 56](#_Toc529476596)

[#3 57](#_Toc529476597)

[#4 58](#_Toc529476598)

[#5 59](#_Toc529476599)

# UML



# Detailed Design

**Client**

**Data Structures Used:**

*Hashtable*

*BST*

*Array of integers*

## Function: main()

int arrRandomNums[250]

CreateListRandomInts(arrRandomNums);

BST fifty = CreateTree(50, arrRandomNums)

BST onefifty = CreateTree(150, arrRandomNums)

BST twofifty = CreateTree(250, arrRandomNums)

HashTable hfifty = CreateHashTable(50, arrRandomNums)

HashTable honefifty = CreateHashTable(150, arrRandomNums)

HashTable htwofifty = CreateHashTable(250, arrRandomNums)

HashTable hsfifty = CreateSortedHashTable(50, arrRandomNums)

HashTable hsonefifty = CreateSortedHashTable(150, arrRandomNums)

HashTable hstwofifty = CreateSortedHashTable(250, arrRandomNums)

*BSTTrial(fifty)*

*BSTTrial(onefifty)*

*BSTTrial(twofifty)*

*HashTableTrial(hfifty)*

*HashTableTrial(honefifty)*

*HashTableTrial(htwofifty)*

*SortedHashTableTrial(hsfifty)*

*SortedHashTableTrial(hsonefifty)*

*SortedHashTableTrial(hstwofifty)*

### **Function:** CreateListRandomInts(int array[])

Narrative: This function creates a list of

Pre-condition: array is created that can hold 250 integers

Post-condition: 250 random unique integers with values between 100 and 500 will be created and placed in the array.

*int min = 100;*

*int max = 500;*

*if (max < min) {*

*int temp = min*

*min = max*

*max = temp*

*}*

*int range = (max - min);*

*int \*values = new int[range]*

*for (int i = 0; i <= range; i++) {*

*values[i] = min + i*

*}*

*shuffle(values, range + 1)*

*for (int i = 0; i < 250; i++) {*

*arrRandomNums[i] = values[i]*

*}*

*return*

### Function: shuffle(int values[], int size)

Narrative: Shuffles around the values in the array values

Pre-condition: values has been filled

Post-condition: values will be an unsorted randomized array of values

*srand(1)*

*for (int i = 0; i < (size \* 20); i++) {*

*int randvalue1 = (rand() % size) + 0*

*int randvalue2 = (rand() % size) + 0*

*int temp = values[randvalue1]*

*values[randvalue1] = values[randvalue2]*

*values[randvalue2] = temp*

*}*

### Function: CreateTree(int length, int randArray[])

Narrative: This will fill a binary search tree with the values in randArray.

Pre-condition: The length of randArray has been determined and randArray has been filled with values.

Post-condition: a BST is returned containing all values in randArray[]

*BST temp*

*for (int i = 0; i < length; i++) {*

*temp.Insert(arrRandomNums[i])*

*}*

*return temp*

### **Function:** CreateHashTable(int length, int temp[])

Narrative: This will fill a hash table with values in randArray

Pre-condition: The length of randArray has been determined and randArray has been filled with values.

Post-condition: a Hash Table is returned containing all values in randArray[]

*HashTable temphash;*

*for (int i = 0; i < length; i++) {*

*temphash.Insert(temp[i]);*

*}*

*return temphash;*

### **Function:** CreateSortedHashTable (int length, int temp[])

Narrative: This will fill a hash table with sorted buckets using values in randArray

Pre-condition: The length of randArray has been determined and randArray has been filled with values.

Post-condition: a Sorted Hash Table is returned containing all values in randArray[]

*HashTable temphash*

*for (int i = 0; i < length; i++) {*

*temphash.InsertSorted(temp[i]);;*

*}*

*return temphash*

### **Function:** *BSTTrial* *(BST tree)*

Narrative: This runs a trial on the binary tree that is passed into the function. The trial includes outputting the size of the BST and the average number of probes it takes to find a value between 100 and 500

Pre-condition: tree is a filled binary search tree

Post-condition: Binary search tree trial requirements (List size and Average number of probes) will be output to the screen.

*int numCompares[400]*

*for (int i = 0; i < 400; i++) {*

*int temp = 0*

*tree.NumComparisons(tree.ReturnRoot(), i + 100, temp)*

*numCompares[i] = temp*

*}*

*"Binary Search Tree implementation"*

*"List Size : " << tree.ReturnLength()*

*int averageNumProbes = 0*

*for (int i = 0; i < 400; i++) {*

*averageNumProbes += numCompares[i]*

*}*

*averageNumProbes = averageNumProbes / 400*

*"Average number of probes : " << averageNumProbes*

*return*

### **Function:** *HashTableTrial* *(HashTable table)*

Narrative: This runs a trial on the HashTable that is passed into the function. The trial includes outputting the size of the HashTable and the average number of probes it takes to find a value between 100 and 500

Pre-condition: table is a filled hash table

Post-condition: Hash Table trial requirements (List size and Average number of probes) will be output to the screen.

*int numCompares[400];*

*for (int i = 0; i < 400; i++) {*

*numCompares[i] = table.numCompares(i + 100);*

*}*

*"Unsorted Hash Table Implementation"*

*"List Size : " << table.ReturnLength()*

*int averageNumProbes = 0*

*for (int i = 0; i < 400; i++) {*

*averageNumProbes += numCompares[i]*

*}*

*averageNumProbes = averageNumProbes / 400*

*"Average number of probes : " << averageNumProbes*

*return*

### **Function:** *SortedHashTableTrial(HashTable table)*

Narrative: This runs a trial on the sorted HashTable that is passed into the function. The trial includes outputting the size of the sorted HashTable and the average number of probes it takes to find a value between 100 and 500

Pre-condition: table is a filled sorted hash table

Post-condition: Sorted Hash Table trial requirements (List size and Average number of probes) will be output to the screen.

*int numCompares[400];*

*for (int i = 0; i < 400; i++) {*

*numCompares[i] = table.numComparesSorted(i + 100);*

*}*

*"Unsorted Hash Table Implementation"*

*"List Size : " << table.ReturnLength()*

*int averageNumProbes = 0*

*for (int i = 0; i < 400; i++) {*

*averageNumProbes += numCompares[i]*

*}*

*averageNumProbes = averageNumProbes / 400*

*"Average number of probes : " << averageNumProbes*

*return*

## **Class:** HashTable

*LinkedList bucket[10]*

*int length*

*int Hash(int key) {return (key % 10);}*

### **Method:** *Hash(int key)*

Narrative: This takes the key and passes it through the hashing function returning the location in the bucket that the key should be placed in.

Pre-condition: none

Post-condition: the location that the key should be placed in is returned

*return (key % 10)*

### **Method:** void Insert(int key)

Narrative: Inserts a value into the bucket

Pre-condition: key has a value and the buckets have been initialized

Post-condition: node with the value key is added to the table. Length is updated

bucket[Hash(key)].Insert(key)

length++

return

### **Method:** void InsertSorted(int key)

Narrative: Adds sorted functionality to the hashtable class, same as insert but sorted buckets

Pre-condition: same as Insert

Post-condition: same as Insert

bucket[Hash(key)].InsertSorted(key)

length++

return

### **Method:** bool Search(int key)

Narrative: returns if key was found in the table

Pre-condition: the table is initialized

Post-condition: returns true if key is found in table

return bucket[Hash(key)].Search(key)

### **Method:** int numCompares(int key)

Narrative: searches the table for key

Pre-condition: table initialized

Post-condition: returns amount of compares made to attempt finding key

return bucket[Hash(key)].NumCompares(key)

### **Method:** int numComparesSorted(int key)

Narrative: Same as numCompares but sorted

Pre-condition: Same as numCompares but sorted

Post-condition: Same as numCompares but sorted

return bucket[Hash(key)].NumComparesSorted(key)

### **Method:** void PrintTable()

Narrative: Prints the contents of each bucket

Pre-condition: table initialized

Post-condition: information printed to screen

for (int i = 0; i < 10; i++) {

"Bucket " << i << " : "

bucket[i].PrintList()

endl

}

endl

return

### **Method:** int ReturnLength()

Narrative: Returns the amount of elements in the hash table

Pre-condition: length has a value

Post-condition: length is returned

*Return length*

## **Class:** LinkedList

*struct LLNode {*

*int key*

*NodePtr next*

*}*

*NodePtr head*

*int length*

### **Method:** void Insert(int data)

Narrative: Inserts a node into the linked list at the head

Pre-condition: data has a value and list initialized

Post-condition: a node with key = data inserted into list, length updated

NodePtr temp = new LLNode

temp->key = data

temp->next = nullptr

length++

if (!head) {

head = temp

return

}

temp->next = head

head = temp

return

### **Method:** void InsertSorted(int data)

Narrative: Same as Insert but inserts into sorted location in list

Pre-condition: same as Insert

Post-condition: Same as Insert

*NodePtr temp = new LLNode*

*NodePtr current*

*temp->key = data*

*temp->next = nullptr*

*length++*

*if (!head || head->key >= temp->key) {*

*temp->next = head*

*head = temp*

*}*

*else {*

*current = head*

*while (current->next != nullptr && current->next->key < temp->key)*

*current = current->next;*

*temp->next = current->next*

*current->next = temp*

*}*

*return*

### **Method:** bool Search(int key)

Narrative: Searches for a node with value of key

Pre-condition: list initialized

Post-condition: returns true if key found in list else return false

NodePtr currPtr = head

int numCompares = 0

while (currPtr != nullptr && currPtr->key != key) {

currPtr = currPtr->next

}

return (currPtr)

### **Method:** int NumComparesSorted(int key)

Narrative: Searches for an item in the sorted list with value key and keeps track of the number of compares and returns that value

Pre-condition: list initalized

Post-condition: number of comparisons returned

*NodePtr currPtr = head*

*int numCompares = 0*

*while (currPtr != nullptr && currPtr->key < key) {*

*currPtr = currPtr->next*

*numCompares++*

*}*

*return numCompares*

### **Method:** int NumCompares(int key)

Narrative: Searches for an item in the list with value key and keeps track of the number of compares and returns that value

Pre-condition: list initalized

Post-condition: number of comparisons returned

*NodePtr currPtr = head*

*int numCompares = 0*

*while (currPtr != nullptr && currPtr->key != key) {*

*numCompares++*

*currPtr = currPtr->next*

*}*

*return numCompares*

### **Method:** void PrintList()

Narrative: Prints the contents of the list

Pre-condition: list initialized

Post-condition: values of each node printed to screen

*NodePtr currPtr = head*

*while (currPtr != nullptr) {*

*cout << currPtr->key << " "*

*currPtr = currPtr->next*

*}*

*return*

## **Class:** BST

*Node \*root;*

*int length;*

*struct Node{*

*int key*

*struct Node \*left = nullptr*

*struct Node \*right = nullptr*

*struct Node \*parent = nullptr*

*}*

### **Method:** void InorderTraverse(NodePtr node)

Narrative: prints the contents of the tree in sorted order with node as root of the tree

Pre-condition: tree initialized

Post-condition: contents of the tree printed to screen

*if(node != NULL){*

*InorderTraverse(node->left)*

*cout << node->key << " "*

*InorderTraverse(node->right)*

*}*

*return*

### **Method:** void Insert(int item)

Narrative: Inserts a node in the tree with value of item

Pre-condition: tree initialized

Post-condition: node with key of item inserted into tree, length updated

*NodePtr temp = new Node*

*temp->key = Item*

*NodePtr y = NULL*

*NodePtr x = root*

*while (x != NULL) {*

*y = x*

*if (temp->key < x->key)*

*x = x->left*

*else*

*x = x->right*

*}*

*temp-> parent = y*

*if (y == NULL)*

*root = temp*

*else if (temp->key < y->key)*

*y->left = temp*

*else*

*y->right = temp*

*length++*

*return*

### **Method:** NodePtr Search(NodePtr x, int item)

Narrative: Searches for an item in the list with x as root recursively

Pre-condition: tree initialized

Post-condition: if item is not found a null pointer is returned. Otherwise node with key = item is returned

*if( (x == NULL) || (x->key == item)) {*

*return x*

*}*

*if (item < x->key)*

*return Search(x->left, item)*

*else*

*return Search(x->right, item)*

### **Method:** NodePtr FindPredecessor(int item)

Narrative: finds the predecessor of the node with key = item in the tree

Pre-condition: item is in the list

Post-condition: node that is predecessor of node is returned or null if predecessor is not found

NodePtr x = Search(root, item)

if (x == NULL)

return NULL

if (x->left != NULL)

return FindMax(x->left)

NodePtr y = x->parent

while (y != NULL && x == y->left) {

x = y

y = y->parent

}

return y

### **Method:** NodePtr FindSucessor(int item)

Narrative: finds the sucessor of the node with key = item in the tree

Pre-condition: item is in the list

Post-condition: node that is sucessor of node is returned or null if successor is not found

*NodePtr x = Search(root, item)*

*if (x == NULL)*

*return NULL*

*if (x->right != NULL)*

*return FindMin(x->right)*

*NodePtr y = x->parent*

*while (y != NULL && x == y->right) {*

*x = y*

*y = y->parent*

*}*

*return y*

### **Method:** NodePtr NumComparisons(NodePtr x, int item, int& numComparisons)

Narrative: Calculates the number of comparisons it takes to find item in tree with x as root. Uses numComparisons as an output parameter

Pre-condition: initialized tree

Post-condition: numComparisons will have the value of the number of comparisons this function has made.

*numComparisons++*

*if ((x == NULL) || (x->key == item)) {*

*return x*

*}*

*if (item < x->key)*

*return NumComparisons(x->left, item, numComparisons)*

*else*

*return NumComparisons(x->right, item, numComparisons)*

### **Method:** NodePtr ReturnRoot()

Narrative: helps client use root of a tree for functions

Pre-condition: tree initialized

Post-condition: root returned

*Return root;*

### **Method:** int ValueofNode(NodePtr node)

Narrative: returns the key of the node

Pre-condition: node is either null or is pointing to a current node

Post-condition: -1 if node is null or value of key is returned

*if(node)*

*return node->key*

*return -1*

### **Method:** int ReturnLength()

Narrative: returns the amount of nodes in the tree

Pre-condition: none

Post-condition: number of nodes returned

*Return length*

### **Method:** NodePtr FindMin(NodePtr node)

Narrative: Finds the minimum value in the tree with node as root

Pre-condition: tree is initialized

Post-condition: node with the lowest key returned

*NodePtr current = node*

*while (current->left != NULL)*

*current = current->left*

*return current*

### **Method:** NodePtr FindMax(NodePtr node)

Narrative: Finds the max value in the tree with node as root

Pre-condition: tree is initialized

Post-condition: node with the highest key returned

*NodePtr current = node*

*while (current->right != NULL)*

*current = current->right*

*return current*

### **Method:** void BurnForest(NodePtr node)

Narrative: Helper function for the destructor. Visits every node and deletes recursively

Pre-condition: tree initialized

Post-condition: all nodes deleted

*if (node != NULL){*

*BurnForest(node->left)*

*BurnForest(node->right)*

*delete node*

*}*

### **Method:** void CreateTree(NodePtr node)

Narrative: this inserts into the current tree with all of the values in the tree of node with node as the root

Pre-condition: otherlist is initalized

Post-condition: all values of tree of node inserted.

*if (node != NULL) {*

*Insert(node -> key);*

*CreateTree(node->left);*

*CreateTree(node->right);*

*}*

*return;*

# Test Plan

**Client Code**

## Function Prototype: main()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Function Prototype:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

## Class: BST

**Method Prototype: insert**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| BTI1 | Insert | Empty Tree | 1 | Inserted |  |
| BTI2 |  | Tree with values | 50 | inserted |  |

**Method Prototype:** equals operator/copy constructor

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| T= | Not deleting original list |  | 20 30 40 50 60 70 80 |  |  |
| Tpass | Not deleting original list |  | 20 30 40 50 60 70 80 |  |  |

**Method Prototype: Search**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| BS1 | Search | Item In List | 20 | Found |  |
| BS2 |  | Item not in list | 0 | Not found |  |

**Method Prototype: findPredecessor**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| BPRE1 | Predecessor | None | 20 | -1 |  |
| BPRE2 |  | Middle | 50 | 40 |  |
| BPRE3 |  | Top | 80 | 70 |  |

**Method Prototype: findSucessor**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| BPOST1 | successor | Bottom | 20 | 30 |  |
| BPOST2 |  | Middle | 50 | 60 |  |
| BPOST3 |  | none | 80 | -1 |  |

## Class: HashTable

**Method Prototype: insert**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| HTI1 | Insert | EmptyList | 1 | Inserted |  |
| HTI2 |  | List with items | 80 60 70 40 20 30 50 | inserted |  |
|  |  | Sorted | 80 60 70 40 20 30 50 | 20 30 40 50 60 70 80 |  |

**Method Prototype:** overload = / copy construct

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| TH= | Original list not destroyed | 80 60 70 40 20 30 50 | 80 60 70 40 20 30 50 | Not destroyed |  |
| THpass |  |  | 80 60 70 40 20 30 50 | Not destroyed |  |

**Method Prototype: Search**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| HS1 | Item in list | In list | 20 | Was found |  |
| HS2 |  | Not in list | 0 | Not found |  |

# Class Specs and Implementation

## Hash Table.h

#pragma once

#ifndef HASHTABLE\_H

#define HASHTABLE\_H

#include"LinkedList.h"

class HashTable

{

public:

HashTable();

~HashTable();

void Insert(int key);

void InsertSorted(int key);

bool Search(int key);

int numCompares(int key);

int numComparesSorted(int key);

void PrintTable();

int ReturnLength() { return length; }

private:

LinkedList bucket[10];

int length;

int Hash(int key) {return (key % 10);}

};

#endif

## Hash Table.cpp

#include "HashTable.h"

#include <iostream>

HashTable::HashTable(){

length = 0;

}

HashTable::~HashTable(){

for (LinkedList i : bucket) {

i.~LinkedList();

}

}

void HashTable::Insert(int key) {

//Narrative: Inserts a value into the bucket

// Pre - condition : key has a value and the buckets have been initialized

// Post - condition : node with the value key is added to the table.Length is updated

bucket[Hash(key)].Insert(key);

length++;

return;

}

void HashTable::InsertSorted(int key) {

//Narrative: Adds sorted functionality to the hashtable class, same as insert but sorted buckets

// Pre - condition : same as Insert

// Post - condition : same as Insert

bucket[Hash(key)].InsertSorted(key);

length++;

return;

}

bool HashTable::Search(int key) {

//Narrative: returns if key was found in the table

// Pre - condition : the table is initialized

// Post - condition : returns true if key is found in table

return bucket[Hash(key)].Search(key);

}

int HashTable::numCompares(int key) {

//Narrative: searches the table for key

// Pre - condition : table initialized

// Post - condition : returns amount of compares made to attempt finding key

return bucket[Hash(key)].NumCompares(key);

}

int HashTable::numComparesSorted(int key) {

//Narrative: Same as numCompares but sorted

// Pre - condition : Same as numCompares but sorted

// Post - condition : Same as numCompares but sorted

return bucket[Hash(key)].NumComparesSorted(key);

}

void HashTable::PrintTable(){

//Narrative: Prints the contents of each bucket

// Pre - condition : table initialized

// Post - condition : information printed to screen

for (int i = 0; i < 10; i++) {

std::cout << "Bucket " << i << " : ";

bucket[i].PrintList();

std::cout << std::endl;

}

std::cout << std::endl;

return;

}

## LinkedList.h

#pragma once

#ifndef LINKEDLIST\_H

#define LINKEDLIST\_H

class LinkedList

{

private :

struct LLNode;

typedef LLNode\* NodePtr;

public:

LinkedList();

~LinkedList();

LinkedList(const LinkedList& otherlinkedlist);

const LinkedList& operator=(const LinkedList& otherLinkedList);

void Insert(int data);

void InsertSorted(int data);

bool Search(int key);

int NumComparesSorted(int key);

int NumCompares(int key);

void PrintList();

private:

struct LLNode {

int key;

NodePtr next;

};

NodePtr head;

int length;

};

#endif

## LinkedList.cpp

#include "LinkedList.h"

#include <iostream>

LinkedList::LinkedList(){

length = 0;

head = nullptr;

}

LinkedList::~LinkedList(){

NodePtr temp;

NodePtr curr = head;

while (curr != nullptr) {

temp = curr;

curr = curr->next;

delete temp;

}

head = nullptr;

length = 0;

return;

}

LinkedList::LinkedList(const LinkedList& otherLinkedList) {

NodePtr from;

NodePtr to;

length = otherLinkedList.length;

if (otherLinkedList.head == nullptr)

head = nullptr;

else {

from = otherLinkedList.head;

head = new LLNode;

head->key = from->key;

to = head;

from = from->next;

while (from != nullptr) {

to->next = new LLNode;

to = to->next;

to->key = from->key;

from = from->next;

}

to->next = nullptr;

}

return;

}

const LinkedList& LinkedList::operator=(const LinkedList& otherLinkedList) {

NodePtr from;

NodePtr to;

length = otherLinkedList.length;

if (otherLinkedList.head == nullptr)

head = nullptr;

else {

from = otherLinkedList.head;

head = new LLNode;

head->key = from->key;

to = head;

from = from->next;

while (from != nullptr) {

to->next = new LLNode;

to = to->next;

to->key = from->key;

from = from->next;

}

to->next = nullptr;

}

return \*this;

}

void LinkedList::Insert(int data) {

//Narrative: Inserts a node into the linked list at the head

// Pre - condition : data has a value and list initialized

// Post - condition : a node with key = data inserted into list, length updated

NodePtr temp = new LLNode;

temp->key = data;

temp->next = nullptr;

length++;

if (!head) {

head = temp;

return;

}

temp->next = head;

head = temp;

return;

}

void LinkedList::InsertSorted(int data) {

//Narrative: Same as Insert but inserts into sorted location in list

// Pre - condition : same as Insert

// Post - condition : Same as Insert

NodePtr temp = new LLNode;

NodePtr current;

temp->key = data;

temp->next = nullptr;

length++;

if (!head || head->key >= temp->key) {

temp->next = head;

head = temp;

}

else {

current = head;

while (current->next != nullptr && current->next->key < temp->key)

current = current->next;

temp->next = current->next;

current->next = temp;

}

return;

}

bool LinkedList::Search(int key) {

//Narrative: Searches for a node with value of key

// Pre - condition : list initialized

// Post - condition : returns true if key found in list else return false

NodePtr currPtr = head;

int numCompares = 0;

while (currPtr != nullptr && currPtr->key != key) {

currPtr = currPtr->next;

}

return (currPtr);

}

int LinkedList::NumCompares(int key) {

//Narrative: Searches for an item in the list with value key and keeps track of the number of compares and returns that value

// Pre - condition : list initalized

// Post - condition : number of comparisons returned

NodePtr currPtr = head;

int numCompares = 0;

while (currPtr != nullptr && currPtr->key != key) {

numCompares++;

currPtr = currPtr->next;

}

return numCompares;

}

int LinkedList::NumComparesSorted(int key) {

//Narrative: Searches for an item in the sorted list with value key and keeps track of the number of compares and returns that value

// Pre - condition : list initalized

// Post - condition : number of comparisons returned

NodePtr currPtr = head;

int numCompares = 0;

while (currPtr != nullptr && currPtr->key < key) {

currPtr = currPtr->next;

numCompares++;

}

return numCompares;

}

void LinkedList::PrintList() {

//Narrative: Prints the contents of the list

// Pre - condition : list initialized

// Post - condition : values of each node printed to screen

NodePtr currPtr = head;

while (currPtr != nullptr) {

std::cout << currPtr->key << " ";

currPtr = currPtr->next;

}

return;

}

## BST.h

#ifndef BST\_H

#define BST\_H

#include "NodeT.cpp"

class BST {

public:

BST();

~BST();

BST(const BST & otherBST);

const BST& operator=(const BST & otherBST);

void InorderTraverse(NodePtr node);

void Insert(int item);

NodePtr Search(NodePtr x, int item);

NodePtr FindPredecessor(int item);

NodePtr FindSucessor(int item);

NodePtr NumComparisons(NodePtr x, int item, int& numComparisons);

//void Delete(int item); //Bonus Points for Transpose helper function

NodePtr ReturnRoot(); //This is used so that we can use the root of the tree in other functions

int ValueofNode(NodePtr node);

int ReturnLength() { return length; }

private:

Node \*root;

int length;

NodePtr FindMin(NodePtr node);

NodePtr FindMax(NodePtr node);

void BurnForest(NodePtr node); //Recursive Destructor Helper Function

void CreateTree(NodePtr node); //Recursive Copy Constructer Helper Function

};

#endif

## BST.cpp

#include "BST.h"

#include<iostream>

BST::BST()

{

root = NULL;

length = 0;

}

BST::~BST() {

BurnForest(root);

root = NULL;

}

BST::BST(const BST & otherBST) {

root = NULL;

length = 0;

CreateTree(otherBST.root);

return;

}

const BST& BST::operator=(const BST & otherBST) {

root = NULL;

length = 0;

CreateTree(otherBST.root);

return \*this;

}

void BST::Insert(int Item){

//Narrative: Inserts a node in the tree with value of item

// Pre - condition : tree initialized

// Post - condition : node with key of item inserted into tree, length updated

NodePtr temp = new Node;

temp->key = Item;

NodePtr y = NULL;

NodePtr x = root;

while (x != NULL) {

y = x;

if (temp->key < x->key)

x = x->left;

else

x = x->right;

}

temp-> parent = y;

if (y == NULL)

root = temp;

else if (temp->key < y->key)

y->left = temp;

else

y->right = temp;

length++;

return;

}

void BST::InorderTraverse(NodePtr node) {

//Narrative: prints the contents of the tree in sorted order with node as root of the tree

// Pre - condition : tree initialized

// Post - condition : contents of the tree printed to screen

if(node != NULL){

InorderTraverse(node->left);

std::cout << node->key << " ";

InorderTraverse(node->right);

}

return;

}

NodePtr BST::Search(NodePtr x, int item){

//Narrative: Searches for an item in the list with x as root recursively

// Pre - condition : tree initialized

// Post - condition : if item is not found a null pointer is returned.Otherwise node with key = item is returned

if( (x == NULL) || (x->key == item)) {

return x;

}

if (item < x->key)

return Search(x->left, item);

else

return Search(x->right, item);

}

NodePtr BST::NumComparisons(NodePtr x, int item, int& numComparisons) {

//Narrative: Calculates the number of comparisons it takes to find item in tree with x as root.Uses numComparisons as an output parameter

// Pre - condition : initialized tree

// Post - condition : numComparisons will have the value of the number of comparisons this function has made.

//

numComparisons++;

if ((x == NULL) || (x->key == item)) {

return x;

}

if (item < x->key)

return NumComparisons(x->left, item, numComparisons);

else

return NumComparisons(x->right, item, numComparisons);

}

NodePtr BST::FindSucessor(int item) {

//Narrative: finds the sucessor of the node with key = item in the tree

// Pre - condition : item is in the list

// Post - condition : node that is sucessor of node is returned or null if successor is not found

NodePtr x = Search(root, item);

if (x == NULL)

return NULL;

if (x->right != NULL)

return FindMin(x->right);

NodePtr y = x->parent;

while (y != NULL && x == y->right) {

x = y;

y = y->parent;

}

return y;

}

NodePtr BST::FindPredecessor(int item){

//Narrative: finds the predecessor of the node with key = item in the tree

// Pre - condition : item is in the list

// Post - condition : node that is predecessor of node is returned or null if predecessor is not found

NodePtr x = Search(root, item);

if (x == NULL)

return NULL;

if (x->left != NULL)

return FindMax(x->left);

NodePtr y = x->parent;

while (y != NULL && x == y->left) {

x = y;

y = y->parent;

}

return y;

}

NodePtr BST::ReturnRoot() {

//Narrative: helps client use root of a tree for functions

// Pre - condition : tree initialized

// Post - condition : root returned

return root;

}

NodePtr BST::FindMin(NodePtr node) {

//Narrative: Finds the minimum value in the tree with node as root

// Pre - condition : tree is initialized

// Post - condition : node with the lowest key returned

NodePtr current = node;

while (current->left != NULL)

current = current->left;

return current;

}

NodePtr BST::FindMax(NodePtr node) {

//Narrative: Finds the max value in the tree with node as root

// Pre - condition : tree is initialized

// Post - condition : node with the highest key returned

NodePtr current = node;

while (current->right != NULL)

current = current->right;

return current;

}

void BST::BurnForest(NodePtr node) {

//Narrative: Helper function for the destructor.Visits every node and deletes recursively

// Pre - condition : tree initialized

// Post - condition : all nodes deleted

if (node != NULL){

BurnForest(node->left);

BurnForest(node->right);

delete node;

}

}

void BST::CreateTree(NodePtr node) {

//Narrative: this inserts into the current tree with all of the values in the tree of node with node as the root

// Pre - condition : otherlist is initalized

// Post - condition : all values of tree of node inserted.

if (node != NULL) {

Insert(node -> key);

CreateTree(node->left);

CreateTree(node->right);

}

return;

}

int BST::ValueofNode(NodePtr node) {

//Narrative: returns the key of the node

// Pre - condition : node is either null or is pointing to a current node

// Post - condition : -1 if node is null or value of key is returned

if(node)

return node->key;

return -1;

}

## NodeT.cpp(for BST)

#ifndef NODE

#define NODE

/\*

\* Node Declaration

\*/

struct Node

{

int key;

struct Node \*left = nullptr;

struct Node \*right = nullptr;

struct Node \*parent = nullptr;

};

typedef Node\* NodePtr;

#endif

# Client Source Code

#include<iostream>

#include<random>

#include<time.h>

#include <vld.h>

#include"BST.h"

#include"HashTable.h"

using namespace std;

void shuffle(int values[], int size);

void CreateListRandomInts(int arrRandomNums[]);

BST CreateTree(int length, int arrRandomNums[]);

void BSTTrial(BST tree);

HashTable CreateHashTable(int length, int temp[]);

HashTable CreateSortedHashTable(int length, int temp[]);

void HashTableTrial(HashTable table);

void SortedHashTableTrial(HashTable table);

//BinarySearchTreeTests

void TestInsert(int temp[]);

void CreateTestArray(int temp[]);

void testPass(BST test);

void testSearch(BST test);

void testSuccPre(BST test);

BST CreateTestBST(int temp[]);

//Hash Table Tests

void HTestInsert(int temp[]);

void HTestPass(HashTable test);

void HtestSearch(HashTable test);

int main(){

/\*This block is for testing purposes only\*/

int testValues[7];

CreateTestArray(testValues);

//Binary Search Tree

TestInsert(testValues);

//Testing = operator

cout << "T=:" << endl;

BST test = CreateTestBST(testValues);

test.InorderTraverse(test.ReturnRoot());

cout << endl << endl;

//Testing passing through function

cout << "Tpass:" << endl;

testPass(test);

test.InorderTraverse(test.ReturnRoot());

cout << endl << endl;

//Testing Search

testSearch(test);

//Testing Predecessor and Successor

testSuccPre(test);

//Hash Table

HTestInsert(testValues);

//Testing = and passing

cout << "TH=:" << endl;

HashTable Htest = CreateHashTable(7,testValues);

Htest.PrintTable();

cout << endl << endl;

cout << "THpass:" << endl;

HTestPass(Htest);

Htest.PrintTable();

cout << endl << endl;

HtestSearch(Htest);

/\* Trials \*/

int arrRandomNums[250];

CreateListRandomInts(arrRandomNums);

//BST Trials

BST fifty = CreateTree(50, arrRandomNums);

BST onefifty = CreateTree(150, arrRandomNums);

BST twofifty = CreateTree(250, arrRandomNums);

BSTTrial(fifty);

BSTTrial(onefifty);

BSTTrial(twofifty);

//Hash Table Trials

HashTable hfifty = CreateHashTable(50, arrRandomNums);

HashTable honefifty = CreateHashTable(150, arrRandomNums);

HashTable htwofifty = CreateHashTable(250, arrRandomNums);

HashTableTrial(hfifty);

HashTableTrial(honefifty);

HashTableTrial(htwofifty);

//Sorted Hash Table Trials

HashTable hsfifty = CreateSortedHashTable(50, arrRandomNums);

HashTable hsonefifty = CreateSortedHashTable(150, arrRandomNums);

HashTable hstwofifty = CreateSortedHashTable(250, arrRandomNums);

SortedHashTableTrial(hsfifty);

SortedHashTableTrial(hsonefifty);

SortedHashTableTrial(hstwofifty);

system("pause");

return 0;

}

void shuffle(int values[], int size) {

//Narrative: Shuffles around the values in the array values

// Pre - condition : values has been filled

// Post - condition : values will be an unsorted randomized array of values

// Seed our random number generator.

srand(1);

// Create large number of swaps

// This example takes the size and times it by 20 for the number of swaps

for (int i = 0; i < (size \* 20); i++) {

// Generate random values for subscripts, not values!

int randvalue1 = (rand() % size) + 0;

int randvalue2 = (rand() % size) + 0;

int temp = values[randvalue1];

values[randvalue1] = values[randvalue2];

values[randvalue2] = temp;

}

}

void CreateListRandomInts(int arrRandomNums[]) {

//Narrative: This function creates a list of

// Pre - condition : array is created that can hold 250 integers

// Post - condition : 250 random unique integers with values between 100 and 500 will be created and placed in the array.

int min = 100;

int max = 500;

// If the values for the range was reversed, swap them.

if (max < min) {

int temp = min;

min = max;

max = temp;

}

int range = (max - min);

// Create our new array of size "range"

int \*values = new int[range];

// Load some counting values into our array

for (int i = 0; i <= range; i++) {

values[i] = min + i;

}

// Now shuffle the array values randomly.

shuffle(values, range + 1);

// Spit out the values

for (int i = 0; i < 250; i++) {

arrRandomNums[i] = values[i];

}

return;

}

void CreateTestArray(int temp[]) {

temp[0] = 50;

temp[1] = 30;

temp[2] = 20;

temp[3] = 40;

temp[4] = 70;

temp[5] = 60;

temp[6] = 80;

}

BST CreateTestBST(int temp[]) {

/\* Creating Tree \*/

/\* 50 \*/

/\* / \ \*/

/\* 30 70 \*/

/\* / \ / \ \*/

/\* 20 40 60 80 \*/

/\* \*/

BST tempBST;

for (int i = 0; i < 7; i++)

tempBST.Insert(temp[i]);

return tempBST;

}

void TestInsert(int temp[]) {

BST testInsert;

//Insert a 1 into an empty tree

cout << "BTI 1:" << endl;

testInsert.Insert(1);

testInsert.InorderTraverse(testInsert.ReturnRoot());

cout << endl << endl;

//Inserting multiple items into the tree

cout << "BTI 2:" << endl;

for (int i = 0; i < 7; i++)

testInsert.Insert(temp[i]);

testInsert.InorderTraverse(testInsert.ReturnRoot());

cout << endl << endl;

return;

}

void testPass(BST test) {

return;

}

void testSearch(BST test) {

cout << "BS1" << endl;

if (test.Search(test.ReturnRoot(), 20))

cout << "20 Was found " << endl << endl;

else

cout << "20 Was not found" << endl << endl;

cout << "BS2" << endl;

if (test.Search(test.ReturnRoot(), 0))

cout << "0 Was found " << endl << endl;

else

cout << "0 Was not found" << endl << endl;

return;

}

void testSuccPre(BST test) {

cout << "BPRE1: " << endl;

cout << test.ValueofNode(test.FindPredecessor(20));

cout << endl << endl;

cout << "BPRE2: " << endl;

cout << test.ValueofNode(test.FindPredecessor(50));

cout << endl << endl;

cout << "BPRE3: " << endl;

cout << test.ValueofNode(test.FindPredecessor(80));

cout << endl << endl;

cout << "BPOST1: " << endl;

cout << test.ValueofNode(test.FindSucessor(20));

cout << endl << endl;

cout << "BPOST2: " << endl;

cout << test.ValueofNode(test.FindSucessor(50));

cout << endl << endl;

cout << "BPOST3: " << endl;

cout << test.ValueofNode(test.FindSucessor(80));

cout << endl << endl;

return;

}

void HTestInsert(int temp[]) {

HashTable test;

HashTable testSorted;

//Insert a 1 into empty tree

cout << "HTI 1" << endl;

test.Insert(1);

test.PrintTable();

cout << endl << endl;

//Inserting multiple items into tree

cout << "HTI 2" << endl;

for (int i = 0; i < 7; i++) {

test.Insert(temp[i]);

}

test.PrintTable();

cout << endl << endl;

cout << "HTI 3" << endl;

for (int i = 0; i < 7; i++) {

testSorted.InsertSorted(temp[i]);

}

testSorted.PrintTable();

cout << endl << endl;

}

HashTable CreateHashTable(int length, int temp[]) {

//Narrative: This will fill a hash table with values in randArray

// Pre - condition : The length of randArray has been determined and randArray has been filled with values.

// Post - condition : a Hash Table is returned containing all values in randArray[]

HashTable temphash;

for (int i = 0; i < length; i++) {

temphash.Insert(temp[i]);

}

return temphash;

}

HashTable CreateSortedHashTable(int length, int temp[]) {

//Narrative: This will fill a hash table with sorted buckets using values in randArray

// Pre - condition : The length of randArray has been determined and randArray has been filled with values.

// Post - condition : a Sorted Hash Table is returned containing all values in randArray[]

HashTable temphash;

for (int i = 0; i < length; i++) {

temphash.InsertSorted(temp[i]);

}

return temphash;

}

void HTestPass(HashTable test) {

return;

}

void HtestSearch(HashTable test) {

cout << "HS1" << endl;

if (test.Search(20))

cout << "20 Was found " << endl << endl;

else

cout << "20 Was not found" << endl << endl;

cout << "HS2" << endl;

if (test.Search(0))

cout << "0 Was found " << endl << endl;

else

cout << "0 Was not found" << endl << endl;

return;

}

BST CreateTree(int length, int arrRandomNums[]) {

//Narrative: This will fill a binary search tree with the values in randArray.

// Pre - condition : The length of randArray has been determined and randArray has been filled with values.

// Post - condition : a BST is returned containing all values in randArray[]

BST temp;

for (int i = 0; i < length; i++) {

temp.Insert(arrRandomNums[i]);

}

return temp;

}

void BSTTrial(BST tree) {

//Narrative: This runs a trial on the binary tree that is passed into the function.

// The trial includes outputting the size of the BST and the average number of probes

// it takes to find a value between 100 and 500

//

// Pre - condition : tree is a filled binary search tree

// Post - condition : Binary search tree trial requirements(List size and Average number of probes) will be output to the screen.

int numCompares[400];

for (int i = 0; i < 400; i++) {

int temp = 0;

tree.NumComparisons(tree.ReturnRoot(), i + 100, temp);

numCompares[i] = temp;

//cout << numCompares[i] << endl;

}

cout << endl << endl;

cout << "Binary Search Tree implementation" << endl;

cout << "List Size : " << tree.ReturnLength() << endl ;

int averageNumProbes = 0;

for (int i = 0; i < 400; i++) {

averageNumProbes += numCompares[i];

}

averageNumProbes = averageNumProbes / 400;

cout << "Average number of probes : " << averageNumProbes << endl << endl;

return;

}

void HashTableTrial(HashTable table) {

// Narrative: This runs a trial on the HashTable that is passed into the function.

// The trial includes outputting the size of the HashTable and the average number

// of probes it takes to find a value between 100 and 500

// Pre - condition : table is a filled hash table

// Post - condition : Hash Table trial requirements(List size and Average number of probes) will be output to the screen.

int numCompares[400];

for (int i = 0; i < 400; i++) {

numCompares[i] = table.numCompares(i + 100);

//cout << numCompares[i] << endl;

}

//table.PrintTable();

cout << endl << endl;

cout << "Unsorted Hash Table Implementation" << endl;

cout << "List Size : " << table.ReturnLength() << endl;

int averageNumProbes = 0;

for (int i = 0; i < 400; i++) {

averageNumProbes += numCompares[i];

}

averageNumProbes = averageNumProbes / 400;

cout << "Average number of probes : " << averageNumProbes << endl << endl;

return;

}

void SortedHashTableTrial(HashTable table) {

//Narrative: This runs a trial on the sorted HashTable that is passed into the function.

// The trial includes outputting the size of the sorted HashTable and the average

// number of probes it takes to find a value between 100 and 500

// Pre - condition : table is a filled sorted hash table

// Post - condition : Sorted Hash Table trial requirements(List size and Average number of probes) will be output to the screen.

int numCompares[400];

for (int i = 0; i < 400; i++) {

numCompares[i] = table.numComparesSorted(i + 100);

//cout << numCompares[i] << endl;

}

//table.PrintTable();

cout << endl << endl;

cout << "Sorted Hash Table Implementation" << endl;

cout << "List Size : " << table.ReturnLength() << endl;

int averageNumProbes = 0;

for (int i = 0; i < 400; i++) {

averageNumProbes += numCompares[i];

}

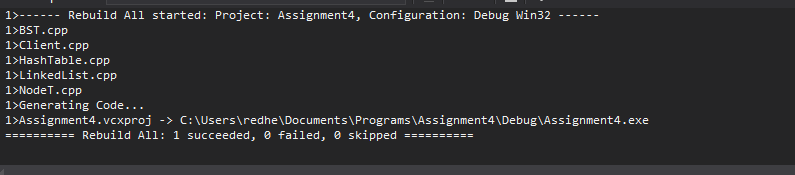
averageNumProbes = averageNumProbes / 400;

cout << "Average number of probes : " << averageNumProbes << endl << endl;

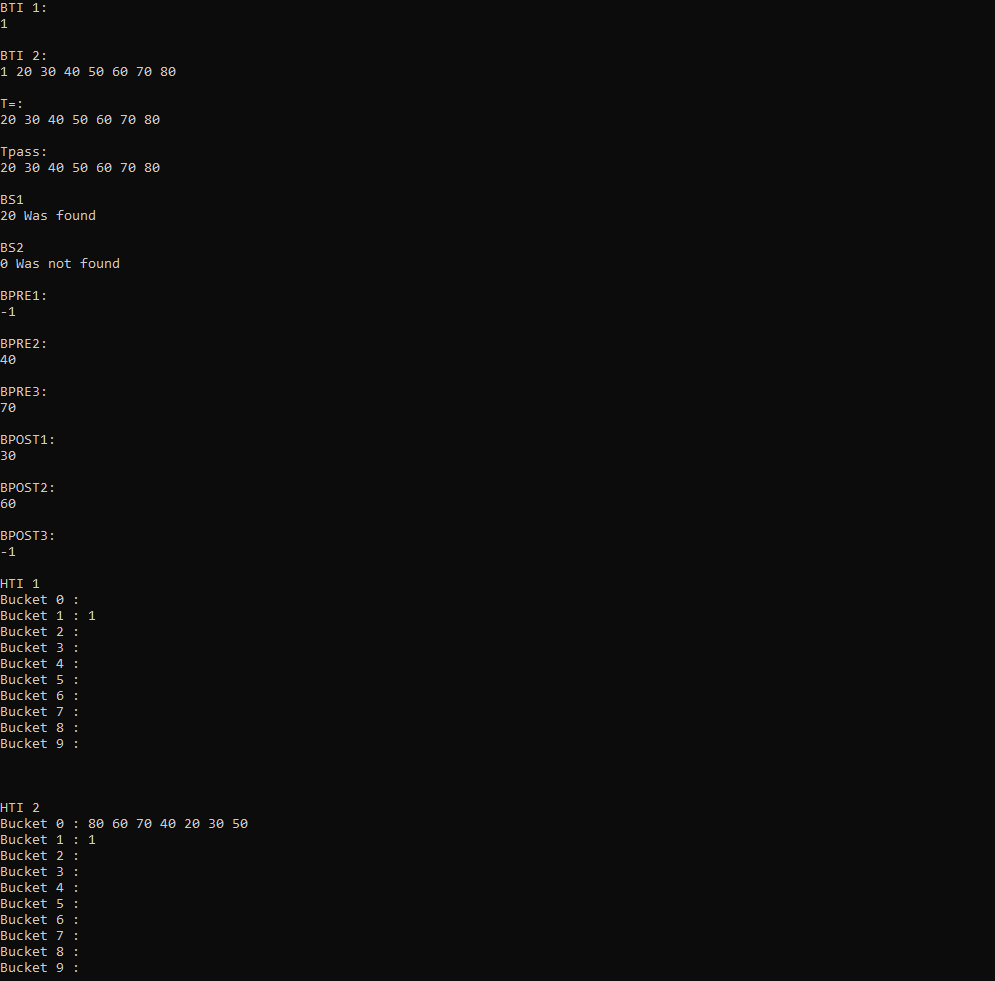
return;

}

# Building Program

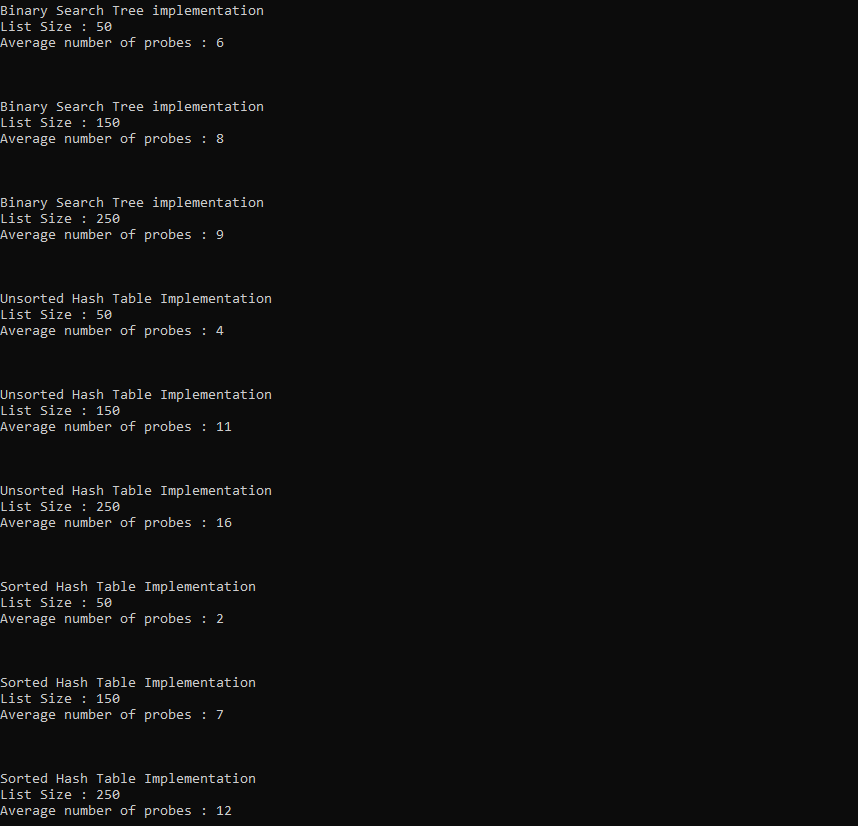


# Test Plan Output





# Program Output



# #2

# #3

Based upon the data collected, as the size of the lists get larger the better it is to use a binary search tree but if you have a larger bucket or a better hashing algorithm then it is better to use a hash table for searching for nodes. The reason for this is as each bucket in a hash table fills, the more nodes you have to check to see if it is the node you are looking for.

# #4

The best way to implement a InorderTraverse function for a hash table would be to collect the nodes in the buckets and place them in an entirely different data structure such as an array, sort that and then traverse the sorted structure printing them out. The reason that this is so hard is that even if it is a sorted hash table, the next sorted item in the table may be in another bucket so you theoretically would have to search all of the buckets for the next sorted item until you get every item in the list.

# #5

The advantages of using a BST would be faster sorted traversal, fast accessing of nodes although this is normally slower than a hash table.

The advantages of a hash table would be almost instant access of any node that you would be searching for. The disadvantage is if you need to access the sorted version of the values.