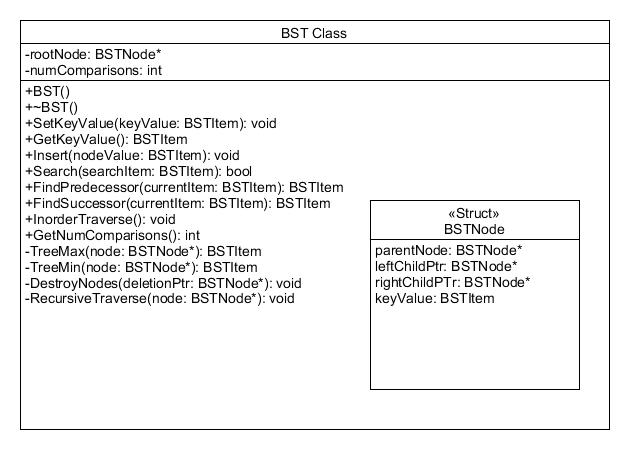
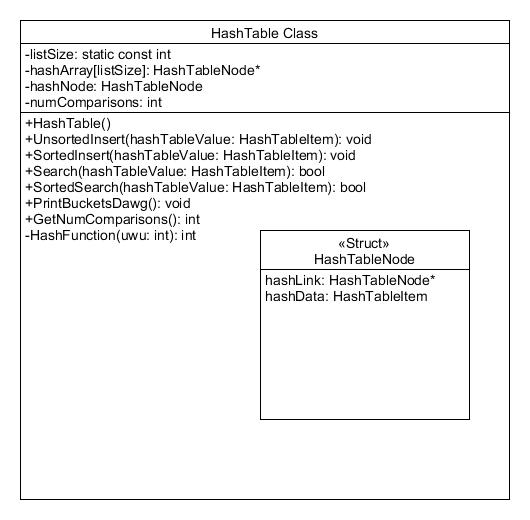
****

****

**Client**

**Data Structures Used:**

BSTNode

BSTItem\* parentNode

BSTItem\* leftChildPtr

BSTItem\* rightChildPtr

BSTItem keyValue

HashTableNode

HashTableItem\* hashLink

HashTableItem hashData

typedef BSTItem int

**Function: main()**

*l*ifstream inputFile

string fileName = “randomInts.txt”

int listSize

int\* ary

string numFromFileString

int numFromFileInt

output “Enter list size: “

read in list size

ary = new int[listSize]

open input file

for(int I = 0; I < listSize; i++)

getline(inputFile, numFromFileString)

numFromFileInt = stoi(numFromFileString, nullptr)

ary[i] = numFromFileInt

endfor

DoBSTOperations(ary, listSize)

DoHashTableUnsortedOps(ary, listSize)

DoHashTableSortedOps(ary, listSize)

close input file

**Function:** void DoBSTOperations(int ary[], int listSize)

Narrative: Does the necessary BST ops per program specs

Pre-condition: ary[] is populated with values and listSize > 0

Post-condition: BST ops have been performed and num comparisons have been noted

BST tree

output message denoting the insertion of items

for(int I = 0; I < listSize; i++)

tree.Insert(ary[i])

endfor

output message denoting the searching

for(int I = 0; I < listSize; i++)

tree.Search(ary[i])

endfor

output total BST comparisons

**Function:** void DoHashTableUnsortedOps(int ary[], int listSize)

Narrative: Does the necessary Hash Table ops (unsorted) per program specs

Pre-condition: ary[] is populated with values and listSize > 0

Post-condition: Hash Table ops have been performed and num comparisons have been noted

HashTable unsortedTable

output message denoting the insertion of items

for(int I = 0; I < listSize; i++)

unsortedTable.UnsortedInsert(ary[i])

endfor

output message denoting the searching

for(int I = 0; I < listSize; i++)

unsortedTable.Search(ary[i])

endfor

output total number of comparisons

**Function:** void DoHashTableSortedOps(int ary[], int listSize)

Narrative: Does the necessary Hash Table ops (sorted) per program specs

Pre-condition: ary[] is populated with values and listSize > 0

Post-condition: Hash Table ops have been performed and num comparisons have been noted

HashTable sortedTable

output message denoting the insertion of items

for(int I = 0; I < listSize; i++)

sortedTable.SortedInsert(ary[i])

endfor

output message denoting the searching

for(int I = 0; I < listSize; i++)

sortedTable.SortedSearch(ary[i])

endfor

output total number of comparisons

**Class:**  *BST*

BSTNode\* rootNode

**Method:** BST()

Narrative: the base constructor for the BST class

Pre-condition: none

Post-condition: a BST object has been created

use CTOR to assign NULL to rootNode

**Method:** ~BST()

Narrative: the destructor for the BST class

Pre-condition: none

Post-condition: a BST object has been deallocated properly

DestroyNodes(rootNode)

**Method:** void SetKeyValue(BSTItem keyValue)

Narrative: The getter for the key value

Pre-condition: none

Post-condition: key value has been returned

rootNode→keyValue = keyValue

**Method:** BSTItem GetKeyValue()

Narrative: The setter for the key value

Pre-condition: none

Post-condition: key value has been set

return rootNode->keyValue

**Method:** void Insert(BSTItem\* node1, BSTItem\* node2)

Narrative: inserts a node into the Binary Search Tree

Pre-condition: none

Post-condition: a node has been inserted into the proper location in the Binary Search Tree

BSTNode\* tempNode = NULL

BSTNode\* rootPtr = rootNode

while(node1 != NULL)

tempNode = node1

if(node2.keyValue < node1.keyValue)

node1 = node1.leftChildPtr

else

node1 = node1.rightChildPtr

endif

endwhile

node2.parent = tempNode

if(tempNode == NULL)

node1 = node2

else

if(node2.keyValue < tempNode.keyValue)

tempNode.leftChildPtr = node2

else

tempNode.rightChildPtr = node2

endif

endif

**Method:** BSTItem\* Search(BSTItem\* searchNode, BSTItem searchItem)

Narrative: Searches for a specific item in the Binary Search Tree

Pre-condition: Binary Search Tree is populated with values

Post-condition: Either true is returned because the value was found or false is returned because the item is not in the Binary Search Tree

if(searchNode == NULL or searchItem == searchNode.keyValue)

return searchNode

endif

if(searchItem < searchNode.keyValue)

return Search(x.leftChildPtr, searchItem)

else

return Search(x.rightChildPtr, searchItem)

endif

**Method:** BSTItem FindPredecessor(BSTItem currentItem)

Narrative: Finds the predecessor to a specific item in the Binary Search Tree

Pre-condition: Binary Search Tree is populated with values, currentItem exists in the tree

Post-condition: the key value of the predecessor is returned

tempNode = NULL

if(currentItem.leftChildPtr != NULL)

return Tree-Max(currentItem.leftChildPtr)

else

tempNode = currentItem.parentNode

while(tempNode != NULL and currentItem == tempNode.leftChildPtr)

currentItem = tempNode

tempNode = tempNode.parentNode

endwhile

endif

**Method:** BSTItem FindSuccessor(BSTItem currentItem)

Narrative: Finds the successor to a specific item in the Binary Search Tree

Pre-condition: Binary Search Tree is populated with values, currentItem exists in the tree

Post-condition: the key value of the successor is returned, or -1 if there is no successor

tempNode = NULL

if(currentItem != NULL)

return Tree-Max(currentItem.rightChildPtr)

else

tempNode = currentItem.parentNode

while(tempNode != NULL and currentItem == tempNode.rightChildPtr)

currentItem = tempNode

tempNode = tempNode.parentNode

endwhile

endif

return tempNode

**Method:** void InorderTraverse(BSTItem\* node)

Narrative: outputs the key values of the Binary Search Tree in order from left-most child on the lowest level to the right-most child on the lowest level

Pre-condition: Binary Search Tree is populated with values

Post-condition: all nodes are output in order

if(node != NULL)

InorderTraverse(node.left)

output node

InorderTraverse(node.right)

endif

**Method:** void Delete(BSTNode\* tree, BSTNode\* deletionValue)

Narrative: Deletes an item from the Binary Search Tree

Pre-condition: Binary Search Tree is populated with data, keyValue exists in list prior to deletion

Post-condition: keyValue has been deleted from Binary Search Tree, all other values around keyValue have been reorganized via Transplant helper function

tempNode = NULL

if(deletionValue.leftChildPtr == NULL)

Transplant(tree, deletionValue, deletionValue.rightChildPtr)

else

if(deletionValue.rightChildPtr == NULL)

Transplant(tree, deletionValue, deletionValue.leftChildPtr)

else

tempNode = Tree-Min(deletionValue.rightChildPtr)

if(tempNode.parentNode != deletionValue)

Transplant(tree, tempNode, tempNode.rightChildPtr)

tempNode.rightChildPtr = deletionValue.rightChildPtr

tempNode.rightChildPtr.parentNode = deletionValue

endif

Transplant(tree, deletionValue, tempNode)

tempNode.leftChildPtr = deletionValue.leftChildPtr

tempNode.leftChildPtr.parentNode = tempNode

endif

endif

delete deletionValue

**Method:** void Transplant(BSTItem tree, BSTItem\* node1, BSTItem\* node2)

Narrative: A helper function for the Delete function of the Binary Search Tree class

Pre-condition: Delete has been called

Post-condition: A node has been transplanted to the proper location for deletion

if(node1.parentNode == NULL)

tree = node2

else

if(node1 == node1.parentNode.leftChildPtr)

node1.parentNode.leftChildPtr = node2

else

node1.parentNode.rightChildPtr = node2

endif

endif

if(node2 != NULL)

node2.parentNode = node1.parentNode

endif

**Method:** BSTNode\* TreeMax(BSTNode\* node)

Narrative: returns the max value of the binary search tree based on the given node

Pre-condition: none

Post-condition: the max value has been returned

if(node.rightChildPtr != NULL)

TreeMax(node.rightChildPtr)

else

return node

**Method:** BSTNode\* TreeMin(BSTNode\* node)

Narrative: returns the min value of the binary search tree based on the given node

Pre-condition: none

Post-condition: the min value has been returned

if(node.leftChildPtr != NULL)

TreeMin(node.leftChildPtr)

else

return node

**Method:** void DestroyNodes(BSTNode\* deletionPtr)

Narrative: a helper function for the destructor

Pre-condition: destructor has been called

Post-condition: all dynamic memory has been deallocated

if(deletionPtr != NULL)

if(deletionPtr→leftChildPtr != NULL)

DestroyNodes(deletionPtr→leftChildPtr)

if(deletionPtr→rightChildPTr != NULL)

DestroyNodes(deletionPTr→rightChildPtr)

delete deletionPtr

**Method:** void RecursiveTraverse(BSTNode\* node)

Narrative: a helper functiopn to traverse the BST

Pre-condition: InorderTraverse has been called

Post-condition: nodes have been output in proper order

if(node != NULL)

REcursiveTraverse(node→leftChildPtr)

output node→keyValue

RecursiveTraverse(node→rightChildPtr)

**Method:** int GetNumComparisons()

Narrative: Returns the total number of comparisons

Pre-condition: none

Post-condition: total number of comparisons returned

return numComparisons

**Class:**  *HashTable*

HashTableNode\* hashArray[listSize]

static const int listSize = 10

HashTableNode hashNode

**Method:** void UnsortedInsert(HashTableItem\* hashTableValue)

Narrative: Adds a node to the Hash Table in any order in the proper bucket

Pre-condition: none

Post-condition: hashTableValue has been inserted into the proper bucket

int assignmentValue

HashTableItem\* tempNode = NULL

assignmentValue = HashFunction(hashTableValue→hashData)

tempNode→hashData = hashTableValue

if(hashArray[assignmentValue] == NULL)

hashArray[assignmentValue] = tempNode

else

tempNode→hashLink = hashArray[assignmentValue]

hashArray[assignmentValue] = tempNode

tempNode = NULL

delete tempNode

**Method:** void SortedInsert(HashTableItem\* hashTableValue)

Narrative: Adds a node to the Hash Table in sorted order into the proper bucket

Pre-condition: none

Post-condition: hashTableValue has been inserted into the proper bucket in sorted order

int assignmentValue

HashTableNode\* prevNode = NULL

HashTableNode\* nextNode = NULL

bool inList = false

HashTableNode\* item = new HashTableNode

item→hashData = hashTableValue

assignmentValue = HashFunction(hashTableValue→hashData)

nextNode = hashArray[assignmentValue]

if(hashArray[assignmentValue] == NULL)

insert hashTableValue into bucket at beginning of list

else

prevNode = nextNode

nextNode = nextNode→hashLink

while(nextNode != NULL && !inList && nextNode→hashData < item→hashData)

prevNode = nextNode

nextNode = nextNode→hashLink

if(prevNode == NULL)

hashArray[assignmentValue] = item

else

prevNode→hashLink = item

item = NULL

delete item

**Method:** bool Search(HashTableItem hashTableValue)

Narrative: Searches the Hash Table for a value

Pre-condition: none

Post-condition: either true is returned because the value has been found, or false is returned because the value is not in the Hash Table

bool found = false

HashTableNode\* tempNode

int comparisons

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

tempNode = hashArray[i]->hashLink

while(!found && tempNode != NULL)

if(tempNode→hashData == hashTableValue

found = true

endif

comparisons++

tempNode = tempNode→hashLink

endwhile

endfor

output number of comparisons

return found

**Method:** bool SortedSearch(HashTableItem hashTableValue)

Narrative: Searches the Hash Table for a value (sorteed)

Pre-condition: none

Post-condition: either true is returned because the value has been found, or false is returned because the value is not in the Hash Table

bool found = false

HashTableNode\* tempNode = new HashTableNode

int comparisons = 0

int assignmentValue = HashFunction(hashTableValue)

tempNode = hashArray[assignmentValue]

while(!found && tempNode != NULL && hashTableValue >= tempNode→hashData)

comparisons++

if(tempNode→hashData == hashTableValue)

found = true

tempNode = tempNode→hashLink

numComparisons += comparisons

return found

**Method:** intHashFunction(int uwu)

Narrative: Assigns a value to a bucket in the Hash Table

Pre-condition: none

Post-condition: value has been assigned to a bucket based on this function

return(uwu%listSize)

**Method:** void PrintBucketsDawg()

Narrative: A debugging function to test if the inserts are working properly or if they need to be dubugged

Pre-condition: none

Post-condition: Buckets have been printed to screen

HashTableNode\* tempNode = NULL

for(int I = 0; I < listSize; i++)

output bucket and bucket number

tempNode = hashArray[i]

while(tempNode != NULL)

output tempNode→hashData

tempNode = tempNode→hashLink

**Method:** int GetNumComparisons()

Narrative: Returns the total number of comparisons

Pre-condition: none

Post-condition: total number of comparisons returned

return numComparisons

**Client Code**

**Function Prototype: main()**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| M1 | Program successfully executing | Program executing properly after input is received | listSize = 50 | Program executes successfully | Yes |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Function Prototype:** void DoBSTOperations(int ary[], int listSize)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| DBSTO1 | Successful function execution | Is the function properly running with the given input | Ary[50] with random integers from list | Function successfully executes | Yes |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Function Prototype:** void DoHashTableUnsortedOps(int ary[], int listSize)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| DHTUO1 | Successful function execution | Is the function properly running with the given input | Ary[50] with random integers from list | Function successfully executes | Yes |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Function Prototype:** void DoHashTableSortedOps(int ary[], int listSize)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| DHSO1 | Successful function execution | Is the function properly running with the given input | Ary[50] with random integers from list | Function successfully executes | Yes |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Class:** BST

**Method Prototype:** BST()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| BST1 | Constructing an object | Successfully constructing a BST object | N/A | An object is successfully instantiated | Yes |
|  |  |  |  |  |  |

**Method Prototype:** ~BST()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| BST1 | Destroying an object | An object is properly deallocated | N/A | All dynamic memory is properly deallocated | Yes |
|  |  |  |  |  |  |

**Method Prototype:** void SetKeyValue(BSTItem keyValue)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key** | **Testing for** | **Test Case** | **Input/Test value** | **Expected Outcome** | **Observed Result** |
| SKV1 | Setting the key value | Key value is properly set | First integer in ary | Key value has been set in insertion | Yes |
|  |  |  |  |  |  |

**Method Prototype:** BSTItem GetKeyValue()

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

**Method Prototype:** void Insert(BSTItem nodeValue)

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

**Method Prototype:** bool Search(BSTItem searchItem)

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

**Method Prototype:** BSTItem FindPredecessor(BSTItem currentItem)

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

**Method Prototype:** BSTItem FindSuccessor(BSTItem currentItem)

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

**Method Prototype:** void InorderTraverse()

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

**Method Prototype:** int GetNumComparisons()

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

**Method Prototype:** BSTItem TreeMax(BSTNode\* node)

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

**Method Prototype:** BSTItem TreeMin(BSTNode\* node)

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

**Method Prototype:** void DestroyNodes(BSTNode\* deletionPtr)

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

**Method Prototype:** void RecursiveTraverse(BSTNode\* node)

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

**Class:** HashTable

**Method Prototype:** HashTable()

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

**Method Prototype:** void UnsortedInsert(HashTableItem hashTableValue)

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

**Method Prototype:** void SortedInsert(HashTableItem hashTableValue)

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

**Method Prototype:** bool Search(HashTableItem hashTableValue)

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

**Method Prototype:** bool SortedSearch(HashTableItem hashTableValue)

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

**Method Prototype:** void PrintBucketsDawg()

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

**Method Prototype:** int GetNumComparisons()

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

**Method Prototype:** int HashFunction(int uwu)

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

/\*

//Bruce Conrad

//BST.h

//This is the header file for the BST class

\*/

#ifndef BST\_H

#define BST\_H

#include <iostream>

#include <string>

using namespace std;

typedef int BSTItem;

class BST

{

public:

BST();

//Narrative: the base constructor for the BST class

//Pre-condition: none

//Post-condition: a BST object has been created

//BST(const BST& otherBST);

//Narrative: the copy constructor for the BST class

//Pre-condition: none

//Post-condition: a BST object has been deep copied

~BST();

//Narrative: the destructor for the BST class

//Pre-condition: none

//Post-condition: a BST object has been deallocated properly

void SetKeyValue(BSTItem keyValue);

//Narrative: The setter for the key value

//Pre-condition: none

//Post-condition: key value has been set

BSTItem GetKeyValue();

//Narrative: The getter for the key value

//Pre-condition: none

//Post-condition: key value has been returned

void Insert(BSTItem nodeValue);

//Narrative: inserts a node into the Binary Search Tree

//Pre-condition: none

//Post-condition: a node has been inserted into the proper location in the

//Binary Search Tree

bool Search(BSTItem searchItem);

//Narrative: Searches for a specific item in the Binary Search Tree

//Pre-condition: Binary Search Tree is populated with values

//Post-condition: Either true is returned because the value was found or

//false is returned because the item is not in the list

BSTItem FindPredecessor(BSTItem currentItem);

//Narrative: Finds the predecessor to a specific item in the

//Binary Search Tree

//Pre-condition: Binary Search Tree is populated with values,

//currentItem exists in the tree

//Post-condition: the key value of the predecessor is returned

BSTItem FindSuccessor(BSTItem currentItem);

//Narrative: Finds the successor to a specific item in the

//Binary Search Tree

//Pre-condition: Binary Search Tree is populated with values,

//currentItem exists in the tree

//Post-condition: the key value is returned, or -1 if there

//is no successor

void InorderTraverse();

//Narrative: outputs the key values of the Binary Search Tree in order

//from left-most child on the lowest level to the right-most child on the

//lowest level

//Pre-condition: Binary Search Tree is populated with values

//Post-condition: all nodes are output in order

//void Delete(BSTItem deletionValue);

//Narrative: Deletes an item from the Binary Search Tree

//Pre-condition: Binary Search Tree is populated with data,

//keyValue exists in list prior to deletion

//Post-condition: keyValue has been deleted from Binary Search Tree,

//all other value around keyValue have been reorganized via

//Transplant helper function

int GetNumComparisons();

//Narrative: Returns the total number of comparisons

//Pre-condition: none

//Post-condition: total number of comparisons is returned

private:

struct BSTNode

{

BSTNode\* parentNode;

BSTNode\* leftChildPtr;

BSTNode\* rightChildPtr;

BSTItem keyValue;

};

BSTNode\* rootNode;

int numComparisons = 0;

//void Transplant(BSTNode\* node1, BSTNode\* node2);

//Narrative: A helper function for the Delete function of the

//Binary Search Tree class

//Pre-condition: Delete has been called

//Post-condition: A node has been transplanted to the proper location

//for deletion

BSTItem TreeMax(BSTNode\* node);

//Narrative: returns the max value of the binary search tree

//based on the given node

//Pre-condition: none

//Post-condition: the max value has been returned

BSTItem TreeMin(BSTNode\* node);

//Narrative: returns the min value of the binary search tree based on

//the given node

//Pre-condition: none

//Post-condition: the min value has been returned

//void CopyIntoMe(const BSTNode\* otherBST);

//Narrative: a helper function for the copy constructor and

//overloaded assignment operator

//Pre-condition: copy constructor or overloaded assignment operator

//have been called

//Post-condition: an object has been copied into another object

void DestroyNodes(BSTNode\* deletionPtr);

//Narrative: a helper function for the destructor

//Pre-condition: destructor has been called

//Post-condition: all dynamic memory has been deallocated

void RecursiveTraverse(BSTNode\* node);

//Narrative: a helper function to traverse the BST

//Pre-condition: InorderTraverse has been called

//Post-condition: nodes have been output in proper order

};

#endif

/\*

//Bruce Conrad

//BST.cpp

//This is the implementation file for the BST class

\*/

#include "BST.h"

using namespace std;

BST::BST(): rootNode(NULL)

{

}

// BST::BST(const BST& otherBST)

// {

// delete rootNode;

// rootNode = NULL;

//

// rootNode = otherBST.rootNode;

// CopyIntoMe(otherBST.rootNode);

// }

BST::~BST()

{

DestroyNodes(rootNode);

}

void BST::SetKeyValue(BSTItem keyValue)

{

rootNode->keyValue = keyValue;

}

BSTItem BST::GetKeyValue()

{

return(rootNode->keyValue);

}

void BST::Insert(BSTItem nodeValue)

{

BSTNode\* item = new BSTNode;

item->keyValue = nodeValue;

BSTNode\* tempNode = rootNode;

BSTNode\* currNode = NULL;

while(tempNode != NULL)

{

currNode = tempNode;

if(item->keyValue < currNode->keyValue)

{

tempNode = tempNode->leftChildPtr;

}

else

{

tempNode = tempNode->rightChildPtr;

}

}

item->parentNode = currNode;

if(currNode == NULL)

{

rootNode = item;

}

else

{

if(item->keyValue < currNode->keyValue)

{

currNode->leftChildPtr = item;

}

else

{

currNode->rightChildPtr = item;

}

}

item = NULL;

delete item;

}

bool BST::Search(BSTItem searchItem)

{

BSTNode\* tempNode = NULL;

tempNode = rootNode;

bool found = false;

int comparisons = 0;

while(tempNode != NULL && !found)

{

if(tempNode->keyValue == searchItem)

{

comparisons++;

found = true;

}

if(searchItem > tempNode->keyValue)

{

comparisons++;

tempNode = tempNode->rightChildPtr;

}

else

{

comparisons++;

tempNode = tempNode->leftChildPtr;

}

}

numComparisons += comparisons;

return(found);

}

BSTItem BST::FindPredecessor(BSTItem currentItem)

{

BSTNode\* tempNode = NULL;

if(rootNode->leftChildPtr != NULL)

{

return(TreeMin(rootNode->leftChildPtr));

}

else

{

tempNode = rootNode->parentNode;

while(tempNode != NULL && currentItem == tempNode->leftChildPtr->keyValue)

{

tempNode = tempNode->parentNode;

}

}

return(tempNode->keyValue);

}

BSTItem BST::FindSuccessor(BSTItem currentItem)

{

BSTNode\* tempNode = NULL;

if(rootNode->rightChildPtr != NULL)

{

return(TreeMax(rootNode->rightChildPtr));

}

else

{

tempNode = rootNode->parentNode;

while(tempNode != NULL && currentItem == tempNode->rightChildPtr->keyValue)

{

tempNode = tempNode->parentNode;

}

}

return(tempNode->keyValue);

}

void BST::InorderTraverse()

{

RecursiveTraverse(rootNode);

}

void BST::RecursiveTraverse(BSTNode\* node)

{

if(node != NULL)

{

RecursiveTraverse(node->leftChildPtr);

cout << node->keyValue << endl;

RecursiveTraverse(node->rightChildPtr);

}

}

// void BST::Delete(BSTItem deletionValue)

// {

// BSTNode\* tempNode = NULL;

// tempNode = rootNode;

//

// if(tempNode->leftChildPtr == NULL)

// {

// Transplant(tempNode, tempNode->rightChildPtr);

// }

//

// else

// {

// if(tempNode->rightChildPtr == NULL)

// {

// Transplant(tempNode, tempNode->leftChildPtr);

// }

//

// else

// {

//

// tempNode->keyValue = TreeMin(tempNode->rightChildPtr);

//

// if(tempNode->parentNode != NULL && tempNode->parentNode->keyValue != deletionValue)

// {

// Transplant(tempNode, tempNode->rightChildPtr);

// tempNode->rightChildPtr->keyValue = deletionValue;

// tempNode->rightChildPtr->parentNode = tempNode;

// }

//

// Transplant(tempNode->rightChildPtr, tempNode);

// tempNode->leftChildPtr->keyValue = deletionValue;

// tempNode->leftChildPtr->parentNode = tempNode;

// }

// }

//

// tempNode = NULL;

// delete tempNode;

// }

// void BST::Transplant(BSTNode\* node1, BSTNode\* node2)

// {

// if(node1->parentNode == NULL)

// {

// rootNode = node2;

// }

//

// else

// {

// if(node1 == node1->parentNode->leftChildPtr)

// {

// node1->parentNode->leftChildPtr = node2;

// }

// else

// {

// node1->parentNode->rightChildPtr = node2;

// }

// }

//

// if(node2 != NULL)

// {

// node2->parentNode = node1->parentNode;

// }

// }

BSTItem BST::TreeMax(BSTNode\* node)

{

if(node->rightChildPtr != NULL)

{

TreeMax(node->rightChildPtr);

}

else

{

return node->keyValue;

}

}

BSTItem BST::TreeMin(BSTNode\* node)

{

if(node->leftChildPtr != NULL)

{

TreeMin(node->leftChildPtr);

}

else

{

return node->keyValue;

}

}

// void BST::CopyIntoMe(const BSTNode\* otherBST)

// {

// if(otherBST != NULL)

// {

// Insert(otherBST->keyValue);

// CopyIntoMe(otherBST->leftChildPtr);

// CopyIntoMe(otherBST->rightChildPtr);

// }

// return;

// }

void BST::DestroyNodes(BSTNode\* deletionPtr)

{

if(deletionPtr != NULL)

{

if(deletionPtr->leftChildPtr != NULL)

{

DestroyNodes(deletionPtr->leftChildPtr);

}

if(deletionPtr->rightChildPtr != NULL)

{

DestroyNodes(deletionPtr->rightChildPtr);

}

delete deletionPtr;

}

}

int BST::GetNumComparisons()

{

return(numComparisons);

}

/\*

//Bruce Conrad

//BST.h

//This is the header file for the HashTable class

\*/

#ifndef HASHTABLE\_H

#define HASHTABLE\_H

#include <iostream>

#include <string>

using namespace std;

typedef int HashTableItem;

class HashTable

{

public:

HashTable();

void UnsortedInsert(HashTableItem hashTableValue);

//Narrative: Adds a node to the Hash Table in any order in the

//proper bucket

//Pre-condition: none

//Post-condition: hashTableValue has been inserted into

//the proper bucket

void SortedInsert(HashTableItem hashTableValue);

//Narrative: Adds a node to the Hash Table in sorted order into

//the proper bucket

//Pre-condition: none

//Post-condition: hashTableValue has been inserted into the

//proper bucket

bool Search(HashTableItem hashTableValue);

//Narrative: Searches the Hash Table for a value (unsorted)

//Pre-condition: none

//Post-condition: either true is returned because the value is found,

//or false is returned because the value is not in the Hash Table

bool SortedSearch(HashTableItem hashTableValue);

//Narrative: Searches the Hash Table for a value (sorted)

//Pre-condition: none

//Post-condition: either true is returned because the value is found,

//or false is returned because the value is not in the Hash Table

void PrintBucketsDawg();

//Narrative: A debugging function to test if the inserts are properly

//working or if they need to be debugged

//Pre-condition: none

//Post-condition: Buckets have been printed to screen

int GetNumComparisons();

//Narrative: Returns the total number of comparisons

//Pre-condition: none

//Post-condition: total number of comparisons is returned

private:

struct HashTableNode

{

HashTableNode\* hashLink;

HashTableItem hashData;

};

static const int listSize = 10;

HashTableNode\* hashArray[listSize];

HashTableNode hashNode;

int numComparisons = 0;

int HashFunction(int uwu);

//Narrative: Assigns a value to a bucket in the Hash Table

//Pre-condition: none

//Post-condition: value has been assigned to a bucket based

//on this function

};

#endif

/\*

//Bruce Conrad

//BST.h

//This is the implementation file for the HashTable class

\*/

#include "HashTable.h"

using namespace std;

HashTable::HashTable()

{

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

{

hashArray[i] = NULL;

}

}

void HashTable::UnsortedInsert(HashTableItem hashTableValue)

{

int assignmentValue;

HashTableNode\* tempNode = new HashTableNode;

assignmentValue = HashFunction(hashTableValue);

tempNode->hashData = hashTableValue;

if(hashArray[assignmentValue] == NULL)

{

hashArray[assignmentValue] = tempNode;

}

else

{

tempNode->hashLink = hashArray[assignmentValue];

hashArray[assignmentValue] = tempNode;

}

tempNode = NULL;

delete tempNode;

}

void HashTable::SortedInsert(HashTableItem hashTableValue)

{

int assignmentValue;

HashTableNode\* prevNode = NULL;

HashTableNode\* nextNode = NULL;

bool inList = false;

HashTableNode\* item = new HashTableNode;

item->hashData = hashTableValue;

assignmentValue = HashFunction(hashTableValue);

nextNode = hashArray[assignmentValue];

if(hashArray[assignmentValue] == NULL)

{

hashArray[assignmentValue] = item;

}

else

{

prevNode = nextNode;

nextNode = nextNode->hashLink;

while(nextNode != NULL && !inList && nextNode->hashData < item->hashData)

{

prevNode = nextNode;

nextNode = nextNode->hashLink;

}

item->hashLink = nextNode;

if(prevNode == NULL)

{

hashArray[assignmentValue] = item;

}

else

{

prevNode->hashLink = item;

}

}

item = NULL;

delete item;

}

bool HashTable::Search(HashTableItem hashTableValue)

{

bool found = false;

HashTableNode\* tempNode = new HashTableNode;

int comparisons = 0;

int assignmentValue = HashFunction(hashTableValue);

tempNode = hashArray[assignmentValue];

while(!found && tempNode != NULL)

{

comparisons++;

if(tempNode->hashData == hashTableValue)

{

found = true;

}

tempNode = tempNode->hashLink;

}

numComparisons += comparisons;

return(found);

}

bool HashTable::SortedSearch(HashTableItem hashTableValue)

{

bool found = false;

HashTableNode\* tempNode = new HashTableNode;

int comparisons = 0;

int assignmentValue = HashFunction(hashTableValue);

tempNode = hashArray[assignmentValue];

while(!found && tempNode != NULL && hashTableValue >= tempNode->hashData)

{

comparisons++;

if(tempNode->hashData == hashTableValue)

{

found = true;

}

tempNode = tempNode->hashLink;

}

numComparisons += comparisons;

return(found);

}

int HashTable::HashFunction(int uwu)

{

return(uwu % listSize);

}

void HashTable::PrintBucketsDawg()

{

HashTableNode\* tempNode = NULL;

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

{

cout << "Bucket: " << i << endl << endl;

tempNode = hashArray[i];

while(tempNode != NULL)

{

cout << tempNode->hashData << endl;

tempNode = tempNode->hashLink;

}

cout << endl;

}

}

int HashTable::GetNumComparisons()

{

return(numComparisons);

}

/\*

//Bruce Conrad

//The Test Client for the BST class.

\*/

#include <iostream>

#include <string>

#include "BST.h"

using namespace std;

void TestInsert(int testValues[], int testListSize, BST& rootNode);

void TestSearch(int searchValue, BST& tester);

void TestFindSuccessor(BST& tester, int successItem);

void TestFindPredecessor(BST& tester, int predItem);

void TestInorderTraverse(BST& tester);

void TestDelete(int deletionValue, BST& tester);

int main()

{

int testArray[10] = {10, 8, 6, 3, 42, 56, 78, 1, 99, 666};

BST testNode;

cout << "Testing Insert!" << endl << endl;

TestInsert(testArray, 10, testNode);

cout << "Done!" << endl << endl;

cout << "Testing Search!" << endl << endl;

TestSearch(666, testNode);

cout << "Done!" << endl << endl;

cout << "Testing FindSuccessor!" << endl << endl;

TestFindSuccessor(testNode, 42);

cout << "Done!" << endl << endl;

cout << "Testing FindingPredeccessor!" << endl << endl;

TestFindPredecessor(testNode, 8);

cout << "Done!" << endl << endl;

cout << "Testing InorderTraverse!" << endl << endl;

TestInorderTraverse(testNode);

cout << "Done!" << endl << endl;

cout << "Testing Delete!" << endl << endl;

TestDelete(6, testNode);

cout << "Done!" << endl << endl;

return 0;

}

void TestInsert(int testValues[], int testListSize, BST& rootNode)

{

int insertionValue;

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

{

insertionValue = testValues[i];

rootNode.Insert(insertionValue);

}

}

void TestSearch(int searchValue, BST& tester)

{

bool rv;

rv = tester.Search(searchValue);

if(rv)

{

cout << "Key value is in list!" << endl;

}

else

{

cout << "Value is not in Binary Search Tree!" << endl;

}

}

void TestFindSuccessor(BST& tester, int successItem)

{

tester.FindSuccessor(successItem);

}

void TestFindPredecessor(BST& tester, int predItem)

{

tester.FindPredecessor(predItem);

}

void TestInorderTraverse(BST& tester)

{

tester.InorderTraverse();

}

void TestDelete(int deletionValue, BST& tester)

{

tester.Delete(deletionValue);

}

/\*

//Bruce Conrad

//This is the Test Driver for the HashTable class.

\*/

#include <iostream>

#include <string>

#include "HashTable.h"

using namespace std;

void TestSortedInsert(int ary[], int arySize, HashTable& tester1);

void TestUnsortedInsert(int ary[], int arySize, HashTable& tester2);

void TestSearch (HashTable& tester1);

int main()

{

int testerArray[10] = {1,5,9,66,42,88,3,57,52,4};

HashTable tester1;

HashTable tester2;

cout << "Before Sorted Insert!" << endl;

TestSortedInsert(testerArray, 10, tester1);

cout << "After Sorted Insert!" << endl;

cout << "Before Unsorted Insert!" << endl;

TestUnsortedInsert(testerArray, 10, tester2);

cout << "After Unsorted Insert!" << endl;

tester1.PrintBucketsDawg();

tester2.PrintBucketsDawg();

cout << "Before Search!" << endl;

TestSearch(tester2);

cout << "After Search!" << endl;

return 0;

}

void TestSortedInsert(int ary[], int arySize, HashTable& tester1)

{

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

{

tester1.SortedInsert(ary[i]);

cout << "Inserted: " << ary[i] << endl;

cout << ary[i]%arySize << endl;

}

}

void TestUnsortedInsert(int ary[], int arySize, HashTable& tester2)

{

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

{

tester2.UnsortedInsert(ary[i]);

}

}

void TestSearch(HashTable& tester1)

{

tester1.UnsortedInsert(62);

tester1.PrintBucketsDawg();

tester1.Search(42);

}

/\*

//Bruce Conrad

//Random Shuffle code that will be output to a file

\*/

#include <iostream>

#include <fstream>

#include <time.h>

using namespace std;

void Shuffle(int ary[], int size);

void SendToFile(ofstream& outFile, string fileName, int ary[], int size);

int main()

{

int min, max;

ofstream outFile;

string fileName = "randomInts.txt";

cout << "Enter the min integer value of number range: ";

cin >> min;

cout << "Enter the max integer value of number range: ";

cin >> max;

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 <= range; i++)

{

cout << "Next random value: " << values[i] << endl;

}

SendToFile(outFile, fileName, values, range);

return 0;

}

void Shuffle(int ary[], int size)

{

srand((int)time(NULL));

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

{

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

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

int temp = ary[randvalue1];

ary[randvalue1] = ary[randvalue2];

ary[randvalue2] = temp;

}

}

void SendToFile(ofstream& outFile, string fileName, int ary[], int size)

{

outFile.open(fileName.c\_str());

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

{

outFile << ary[i] << endl;

}

outFile.close();

}

/\*

//Bruce Conrad

//This is the Client for the BST and Hash Table Program.

\*/

#include <iostream>

#include <string>

#include <fstream>

#include "BST.h"

#include "HashTable.h"

using namespace std;

void DoBSTOperations(int ary[], int listSize);

//Narrative: Does the necessary BST ops per program specs

//Pre-condition: ary[] is populated with values and listSize > 0

//Post-condition: BST ops have been performed and num comparisons have been noted

void DoHashTableUnsortedOps(int ary[], int listSize);

//Narrative: Does the necessary Hash Table ops (unsorted) per program specs

//Pre-condition: ary[] is populated with values and listSize > 0

//Post-condition: Hash Table ops have been performed and num comparisons have been noted

void DoHashTableSortedOps(int ary[], int listSize);

//Narrative: Does the necessary Hash Table ops (sorted) per program specs

//Pre-condition: ary[] is populated with values and listSize > 0

//Post-condition: Hash Table ops have been performed and num comparisons have been noted

int main()

{

ifstream inputFile;

string fileName = "randomInts.txt";

int listSize;

int\* ary;

string numFromFileString;

int numFromFileInt;

cout << "Enter list size: ";

cin >> listSize;

ary = new int[listSize];

inputFile.open(fileName.c\_str());

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

{

getline(inputFile, numFromFileString);

numFromFileInt = stoi(numFromFileString, nullptr);

ary[i] = numFromFileInt;

}

DoBSTOperations(ary, listSize);

DoHashTableUnsortedOps(ary, listSize);

DoHashTableSortedOps(ary, listSize);

inputFile.close();

return 0;

}

void DoBSTOperations(int ary[], int listSize)

{

BST tree;

cout << "Inserting the items!" << endl;

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

{

tree.Insert(ary[i]);

}

cout << "Searching for the values!" << endl;

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

{

tree.Search(ary[i]);

}

cout << "Total BST Comparisons: " << tree.GetNumComparisons() << endl;

}

void DoHashTableUnsortedOps(int ary[], int listSize)

{

HashTable unsortedTable;

cout << "Inserting the items!" << endl;

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

{

unsortedTable.UnsortedInsert(ary[i]);

}

cout << "Searching for the values!" << endl;

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

{

unsortedTable.Search(ary[i]);

}

cout << "Total Unsorted Hash Table Comparisons: ";

cout << unsortedTable.GetNumComparisons() << endl;

}

void DoHashTableSortedOps(int ary[], int listSize)

{

HashTable sortedTable;

cout << "Inserting the items!" << endl;

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

{

sortedTable.SortedInsert(ary[i]);

}

cout << "Searching for the values!" << endl;

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

{

sortedTable.SortedSearch(ary[i]);

}

cout << "Total Sorted Hash Table Comparisons: ";

cout << sortedTable.GetNumComparisons() << endl;

}

Script started on 2019-11-01 16:07:01-0400

]777;notify;Command completed;exit]0;s789220c@cslab162:~/CSCI385/Prog4]7;file://cslab162/home/CS/s789220c/CSCI385/Prog4[16:07] s789220c@cslab162:~/CSCI385/Prog4 $ c++ -c BST.cpp HashTable.cpp

]777;notify;Command completed;c++ -c BST.cpp HashTable.cpp]0;s789220c@cslab162:~/CSCI385/Prog4]file://cslab162/home/CS/s789220c/CSCI385/Prog4[16:07] s789220c@cslab162:~/CSCI385/Prog4 $ c++ BST.o HashTable.o Client.cpp -o Client

]777;notify;Command completed;c++ BST.o HashTable.o Client.cpp -o Client]0;s789220c@cslab162:~/CSCI385/Prog4]7;file://cslab162/home/CS/s789220c/CSCI385/Prog4[16:07] s789220c@cslab162:~/CSCI385/Prog4 $ ./Client

Enter list size: 50

Inserting the items!

Searching for the values!

Total BST Comparisons: 372

Inserting the items!

Searching for the values!

Total Unsorted Hash Table Comparisons: 157

Inserting the items!

Searching for the values!

Total Sorted Hash Table Comparisons: 110

]777;notify;Command completed;./Client]0;s789220c@cslab162:~/CSCI385/Prog4]7;file://cslab162/home/CS/s789220c/CSCI385/Prog4[16:07] s789220c@cslab162:~/CSCI385/Prog4 $ exit

exit

Script done on 2019-11-01 16:07:32-0400

Script started on 2019-11-01 16:07:37-0400

]777;notify;Command completed;exit]0;s789220c@cslab162:~/CSCI385/Prog4]7;file://cslab162/home/CS/s789220c/CSCI385/Prog4[16:07] s789220c@cslab162:~/CSCI385/Prog4 $ ./Client

Enter list size: 150

Inserting the items!

Searching for the values!

Total BST Comparisons: 1382

Inserting the items!

Searching for the values!

Total Unsorted Hash Table Comparisons: 1239

Inserting the items!

Searching for the values!

Total Sorted Hash Table Comparisons: 723

]777;notify;Command completed;./Client]0;s789220c@cslab162:~/CSCI385/Prog4]7;file://cslab162/home/CS/s789220c/CSCI385/Prog4[16:07] s789220c@cslab162:~/CSCI385/Prog4 $ exit

exit

Script done on 2019-11-01 16:07:45-0400

Script started on 2019-11-01 16:07:52-0400

]777;notify;Command completed;exit]0;s789220c@cslab162:~/CSCI385/Prog4]7;file://cslab162/home/CS/s789220c/CSCI385/Prog4[16:07] s789220c@cslab162:~/CSCI385/Prog4 $ ./Client

Enter list size: 250

Inserting the items!

Searching for the values!

Total BST Comparisons: 2517

Inserting the items!

Searching for the values!

Total Unsorted Hash Table Comparisons: 3310

Inserting the items!

Searching for the values!

Total Sorted Hash Table Comparisons: 1945

]777;notify;Command completed;./Client]0;s789220c@cslab162:~/CSCI385/Prog4]7;file://cslab162/home/CS/s789220c/CSCI385/Prog4[16:07] s789220c@cslab162:~/CSCI385/Prog4 $ exit

exit

Script done on 2019-11-01 16:08:00-0400

2)

3) Based on the data, the Sorted Hash Table seems to be the most efficient container in terms of searching for data values. The Binary Search Tree is the weakest by comparison because it does not differentiate which values go where in the tree directly. It instead simply sends the values to the left child or right child, requiring more comparisons until the value is found if it is even in the list at all. By contrast, the Hash Tables dictate which bucket a value goes into, regardless of being sorted or unsorted. This means you can look through a specific bucket based on the value you are searching for, and if it does not exist in that bucket, it is not in the Hash Table. The Sorted Hash Table makes this even easier, as if you pass where the value should be in the list, you can immediately exit the search because it is guaranteed that the value is not in the list.

4)

A Red-Black Tree is a binary search tree that contains an extra attribute, which is the color. This will either be red or black. It has the following properties: each node will either be red or black, every leaf is black, if a node is red, then it’s children are black, and every simple path from a node to a descendant leaf contains the same amount of black nodes [1]. An AVL Tree is a tree that has the following properties: the sub-trees of every node differ in height by at most one, and every sub-tree is an AVL Tree [2]. Finally, a B-Tree is a tree that is optimized for large data blocks. It has the following properties: B-Tree nodes have more than two children, and a B-Tree node may contain more than a single element [3].

For a Red-Black Tree balancing is essentially guaranteed by design, based on the properties it contains. The overall complexity and search times are O(logn) [1]. For the AVL Tree, balance is also guaranteed, and the search time and time to re-balance the tree are both O(logn) [2]. Finally, for the B-Tree, because it is designed to handle large amounts of data, balance is guaranteed, and the worst-case time for adding or removing an element is O(logn) [3]. These alternative approaches appear to be better than a basic Binary Search Tree because they utilize extra properties to maintain balancing and order. Because of this factor, these trees are far better for analyzing larger amounts of data with fewer comparisons than the basic Binary Search Tree.

5)

A treap is a special type of Binary Search Tree that uses a heap to implement the tree. It can be maintained either via a max heap or a min heap [4]. Depending on the type of treap, the properties will vary. For a max order heap implementation, the following properties are true: left subtree nodes of the root contain data fields less than the root data field; right subtree nodes or the root have data fields greater than the root data field; the left and right subtrees are Binary Search Trees; the priority value of the root is higher than its children; and the left and right subtrees are also max heaps [4]. For a min heap, the following properties are true: the left subtree nodes have data fields less than the root data field; the right subtree nodes have data fields greater than the data field of the root; the left and right subtrees are Binary Search Trees, the priority of the root is less than the priority of the children, and the left and right subtrees are also min heaps [4].

Based on the properties of a treap, their purpose is to handle large quantities of data while maintaining a simplified sorting order. This allows the data management to be significantly easier in comparison to a normal Binary Search Tree as it utilizes heap properties to maintain sorted order. By using the heap properties, it may be assumed that the treap can heapify-up or heapify-down, which saves the program execution time when compared to a Binary Search Tree’s delete (which utilizes Transplant until the node is at the end of the list). This significantly reduces the overall cost, therefore increasing efficiency and allowing the use of much larger datasets when compared to a Binary Search Tree.

# Bibliography

|  |  |
| --- | --- |
| [1] | J. Morris, "8.2 Red-Black Trees," 1998. [Online]. Available: https://www.cs.auckland.ac.nz/software/AlgAnim/red\_black.html. |
| [2] | J. Morris, "8.3 AVL Trees," 1998. [Online]. Available: https://www.cs.auckland.ac.nz/software/AlgAnim/AVL.html. |
| [3] | D. Tang, "CS241--Lecture Notes: B-Trees," 2012. [Online]. Available: https://www.cpp.edu/~ftang/courses/CS241/notes/b-tree.htm. [Accessed 1 11 2019]. |
| [4] | Freedman, "Treaps," [Online]. Available: http://faculty.cs.niu.edu/~freedman/340/340notes/340treap.htm. |