Justin Schafer

Jdschafer

CSIS 252

Advanced Storage Lab Assignment

**Problem Summary**

Utilizing the program from Lab 4, store people in different data structures.

**Problem Requirements**

* Store people who have names, email addresses, and social security numbers
* Compare and sort people by their social security numbers
* Add two additional data structures (BST and one of the ones in Chapter 7 or another pre-approved). The one from chapter 7 will be another list (unsorted, sorted, or indexes using a different base structure. It needs to implement one of the same interfaces you used in previous lab)
* Do a detailed analysis of the work done in your lessons learned

**System Design**

|  |  |
| --- | --- |
| UnsortedList extends List | SortedList extends List |
|  |  |
| +UnsortedList()  +UnsortedList(int initSize)  +add(Object element)  +remove(Object element): boolean found | +SortedList()  +SortedList(int initSize)  +add(Comparable element) |

|  |  |
| --- | --- |
| SSNGenerator | <<Interface>>  IOInterface |
| #random: int |  |
| #getRandom(): int random | getInt(int port)  getInt()  getString()  getStringLn()  writeString(String str)  writeStringLn(string str)  writeString(String str, int x, int y)  makeSound(String str)  turn(int direction)  move() |

|  |  |
| --- | --- |
| List | Person |
| #DEFCAP: int  #origCap : int  #list: Object[]  #numElements: int  #currentPos: int  #found: boolean  #location: int  #compares: int | -name: String  -email: String  -SSN: int |
| +List()  +List(int origCap)  #enlarge()  #find(Object target)  +size(): int numElements  +contains(Object element): boolean found  +remove(Object element): boolean found  +toString(): String listString  +reset()  +getNext(): Object next | +Person(int SSN, String name, String email)  +Person(String name, String email)  -getName(): String name  -getSSN(): int SSN  -getEmail(): String email  +compareTo(Person inc): int  +equals(Person inc): boolean  +toString(): String |

|  |  |
| --- | --- |
| <<interface>>  BSTInterface | BinarySearchTree  Implements BSTInterface |
| #INORDER: int  #PREORDER: int  #POSTORDER: int | #root: BSTNode  #found: boolean  #inOrderQueue: ArrayBndQueue  #preOrderQueue: ArrayBndQueue  #postOrderQueue: ArrayBndQueue |
| +isEmpty(): Boolean  +size(): int  +contains(Comparable element): boolean  +remove(Comparable element): boolean  +get(Comparable element): Comparable  +add(Comparable element): void  +reset(int orderType): int  +getNext(int orderType): Comparable | +BinarySearchTree()  +isEmpty(): boolean  +size(): int  +size2(): int  +contains(Comparable element): boolean  +remove(Comparable element): boolean  +get(Comparable element): Comparable  +add(Comparable element): void  +reset(int orderType): int  +getNext(int orderType): Comparable  -getPredecessor(BSTNode tree): Comparable  -removeNode(BSTNode tree): BSTNode  -inOrder(BSTNode tree): void  -preOrder(BSTNode tree): void  -postOrder(BSTNode tree): void |

|  |  |
| --- | --- |
| BSTNode | Queue |
| #BSTNode: left  #BSTNode: right  #Comparable: info | #head:int  #tail:int  #queue:Object[]  #MAX:int  #origCap:int  #elements:int |
| +BSTNode(Comparable info)  +setInfo(Comparable info): void  +getInfo(): Comparable  +setLeft(BSTNode link): void  +setRight(BSTNode link): void  +getLeft(): BSTNode  +getRight(): BSTNode | +Queue()  +enqueue(Object item)  +dequeue():Object item  +grow()  +isEmpty():boolean |

|  |  |
| --- | --- |
| CRefUnsortedList  Implements UnsortedListInterface | LLObjectNode |
| #list:LLObjectNode  #numElements:int  #found:boolean  #compares:int  #location:LLObjectNode  #previous:LLObjectNode  #currentPos:LLObjectNode | -link:LLObjectNode  -info:Object |
| +CRefUnsortedList()  +size():int  #find():void  +contains(Object element):boolean  +toString():String  +remove(Object element):boolean  +get(Object element):Object  +reset():void  +getNext():Object  +add(Object element):void | +LLObjectNode(Object info)  +setInfo(Object info):void  +getInfo():Object  +setLink(LLObjectNode link):void  +getLink():LLObjectNode |

|  |  |
| --- | --- |
| Storage | <<interface>>  UnsortedListInterface |
| -SSNGen:SSNGenerator  -unsortedList:UnsortedList  -sortedList:SortedList  -BSTree:BinarySearchTree  -CRefList:CRefUnsortedList  -storage:Storage |  |
| +Storage()  +main(String[] args): void  -runCRefList():void  -handleCRef():void  -runBSTree():void  -handleBSTree():void  -runSorted():void  -handleSorted():void  -runUnsorted():void  -handleUnsorted():void | +size():int  +contains(Object element):boolean  +remove(Object element):boolean  +get(Object element):Object  +toString():String  +reset():void  +getNext():Object  +add(Object element):void |

**Testing Report**

The IOInterface was never implemented due to lack of understanding and insufficient time so there’s no testing specific cases. Cases are generated at random allowing for a general observation to be made.

**Testing Instructions**

Run the program from the Storage.java file to be presented with information regarding the randomly generated test cases in the console.

**Management Report**

SDR: Completed over the weekend of 3/28/15, 2 hours  
 Revised on 4/8/15, 1 hour

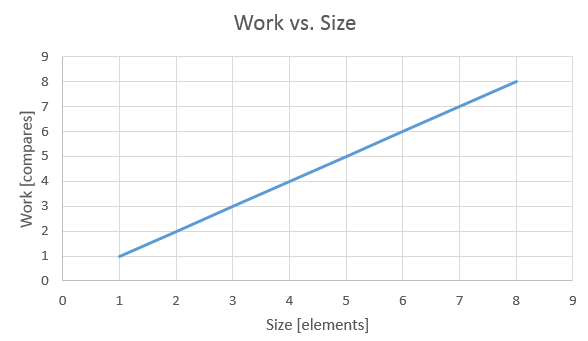
Lab: Estimated 5 hours  
 Actually took around 7 hours

**Lessons Learned**

Last lab we compared the work vs. size of the unsorted list and sorted list ADT’s. This time, we’ve added the BST and CRefList.

For the sake of simplicity, let’s say Social Security Numbers are only up to 2 digits big. We’ll go ahead and use the sample data (43, 21, 95, 45, 2, 5, 24, 97) as our example Social Security Numbers.

For Unsorted List, the conclusion is pretty straight forward. We’ll always have N compares where N is the number of data samples we’re given. That’s because we’ll always be able to add the new data to the end of the list and not have to worry about sorting it. The graph of this input would look like this:



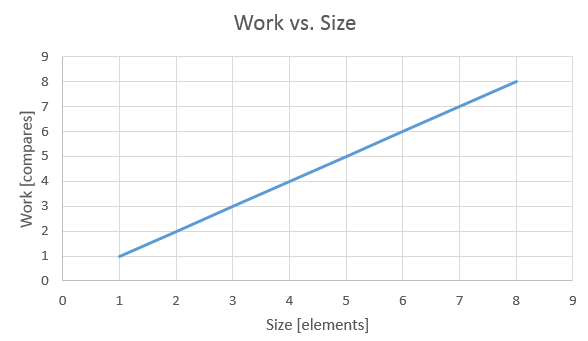
The compares will always match the size since you’re always being added to the end. This means we have O(N).

For Sorted List it gets more complicated. Now we have to take into account that the amount of compares per add could vary greatly since we’re looking for the right spot to insert our data. This though does not mean that the number of compares will ever exceed the number of elements. The graph for this data would look like this:



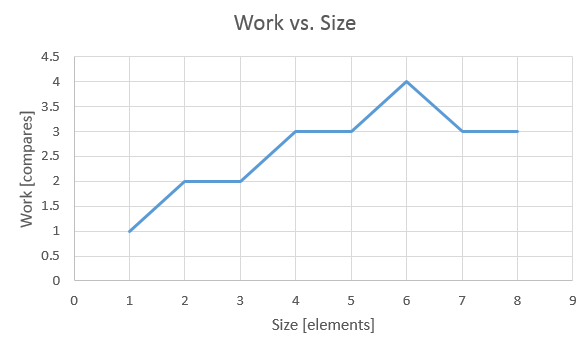
From this graph we can tell when data that doesn’t get sorted to the front comes in, that’s whenever we have a value that doesn’t sit in a flat line. Sorted List may be more work, but it is also much more tidy and easier to find certain objects. This means we have O(N) since we can’t guarantee we’ll always be adding value to the beginning of the list.

CRefList functions in the same way as Unsorted List if we choose to not care what order the elements are in, therefore we can draw the same conclusions about CRefList that we do about Unsorted List. The graph, as we have stated, will look like this:



Since everything will be added to the end the number of compares will steadily rise the number of elements contains in our list. This means we have O(N).

Binary Search Tree is where things start to get a little fun. Due to the nature of which things are sorted in a BST, it means that in an ideal case our work will always be Log2(N) which is phenomenal for us. We can cut out work load down drastically and make much more efficient finds and adds to allow cPU time to be spent on other processes. The graph for the BST will end up looking like this:



BST allows us to mirror the efficiency of the binary search (that name seems familiar) function which operates by starting in the middle and narrowing down the search field by constantly dividing it in half. That is exactly what BST is doing and is why it’s extremely efficient. This means we achieve O(log2(N)) using BST.

The best option in this case would be to use BST since it’s not only more efficient than all the other options, but it’s also sorted. This makes everything much more efficient in the long run.

**Future Improvements**

Implement the functionality of the IOInterface to allow for specific test cases to be loaded into the program and tested. This will also allow for a deeper analysis of the work done for each ADT.

**Appendix**