**README**

Daniel Agu

Assignment 2

CS 3345

10/25/2021

**3.21**

To print out an error message that is likely to reflect the probable cause, the program would need to identify the cause while popping form the stack. The causes could be either the wrong symbol or an empty stack while trying to pop. Once the cause is identified call a method that either prints out the wrong symbol was on the stack signifying it is unbalanced or a message signifying that the stack was empty.

**4.37**

My algorithm for finding the nodes from a binary search tree in a specified range is called BSTRange. BSTRange defines a subclass called tree node which the binary search tree is built off of. The subclass has set and get methods that return and set the node value and the node’s left and right children. All methods in this class are O(1). Within the class of BSTRange there is one class variable root which represents the root of the binary search tree. Lastly we have the algorithm that goes through the given tree to find the correct nodes in the range. Because the tree is using inorder traversal recursively the runtime is O(K). Furthermore, the average depth of a binary search tree is O(Log N). Therefore, because all other methods in the class and subclass are O(1) the complexity is O(K + Log N).

**5.3**

My algorithm for computing the number of collisions required in a long random sequence of insertions using linear probing, quadratic probing, and double hashing is called CollisionCounter. CollisionCounter defines a subclass called element which are to be entered into the hash table. This subclass was created to hold and integer and solves the problem of entering a 0 to be hashed when a default int array is full of 0’s. Because the method only holds and int and has one set and get method the runtime is O(1). In the CollisionCounter class there are three variables to keep track of the number of collisions for each collision resolution method. There are then a method that implements each of the collision resolution methods. Each of the methods follow the same structure at best case where the original hashing function (h mod k where k is table size) without offset maps to an empty slot in the hash table thus making insertion O(1). At worst case if the original slot in the table is empty then a while loop is entered changing the offset and rehashing each time until an empty slot is found. The linear probing method increments the offset by 1, the quadratic probing method increments the offset by 1 then squares it, and the double hash increments the offset by 1 then multiplies it by the second hashing function ((7 – h mod 7) mod k where k is table size). This process has a run time of O(N). The rest of the methods are get methods with O(1) complexity along with the print method which has O(N) complexity thus making the entire class O(N) at the worst case.

**5.5**

My reimplementation of the separate chaining hash tables using singly linked lists is called ChainingHashTable. Defines to subclasses, Node and SinglyLinkedList. The subclass Node is used to build the single linked list and contains set and get methods for the next node in the list along with the value of the node. The complexity is O(1). The subclass SinglyLinkedList utilizes nodes to make a linked list and contains one class variable called head that represents the front of the list. The subclass contains the methods clear, contains, addNode, removeNode, printList, and get/set methods. The clear method is self-explanatory and clears the linked list. The contains method traverses through the list to see if a value is in the list or not and has a complexity of O(N) at worst case if the element is not in the list. The addNode method adds a node to the end of the linked list and has two cases. The best case where the list is empty, and the complexity is O(1) or the case where the list is not empty, and the method must traverse the list to add the node to the end which has O(N) complexity. The removeNode is the exact same construct as addNode where the best case is if the node to be removed is the head thus making the runtime O(1). Otherwise, the list must be traversed until the node is found which at the worst case is at the end of the list making the runtime O(N) at the worst case. The get and set methods for head are O(1) and the print method prints the list with O(N) complexity. Lastly the class ChainingHashTable contains the array of SinglyLinkedLists along with a variable that keeps track of the number of elements in the table. Both of those methods have get and set methods with O(1) complexity. Continuing, the class contains the methods isEmpty, insert, remove, hashFunction, and printTable. The isEmpty checks to see if the hash table is empty and has a complexity of O(1). Insert inserts the value entered into the hash table whether that be in an empty spot or at the end of a list. This method also has O(1) complexity. Remove makes sure the value to be removed is in the table first then removes it from the table with O(1) complexity. Lastly hash function takes in a value returning the hashed index with O(1) complexity and print table prints the hash table with O(N) complexity. Overall, all the methods of the subclasses along with the actual class have a worst-case complexity of O(N).

**Instructions**

For testing the methods uncomment the specified class to test in the main class (Assignment 2). The data file is included for 5.3 and further explanation is included in the comments. If an infinite loop occurs on CollisionCounter that means that one of the numbers with the double hashing has hit a loop where the indexes are repeating themselves so it can’t go in but that is supposed to happen.