Izaak Kersey & Alyssa Ingerson

Help received: Textbook, Notes, Class code

CIS-343 Project

Accomplishments/Challenges

We had many accomplishments on this project we were able to successfully implement all the methods requested in the generic binary search tree class and were able to add hash maps to each node in the binary tree to get it to store duplicates in each node of the tree. We were able to implement the insert, contains, count, removeOne, and and removeAll methods, as well as a main method to test all the methods. For problem 2, we were able to implement all 3 different strategies on the box storage problem and provided an analysis and running times for all 3 methods. We had some difficulties with this project as well. We especially had difficulties implementing the different strategies for organizing the items into boxes in problem 2, particularly with sorting and traversing the priority queues. We also had some issues with sorting the items with the heaviest ones first, but we were able to solve this by entering the weights into an array and sorting them in descending order using a for loop. We also had difficulties coming up with and implementing a 3rd approach to storing the items in the boxes, since the first two were already good approaches to the problem.

Contribution

Cadet Alyssa Ingerson:

- Binary Search Tree hashmap implementation
- Contains, removeOne, removeAll methods for Binary Search Tree

- Strategy 1 for problem 2
- Strategy 2 for problem 2

Cadet Izaak Kersey:

- Box and Item classes
- Accomplishments and challenges writeup
- Insert and count methods in Binary Search Tree
- Strategy 3 for problem 2

Both:

- Running Time Analysis and algorithms

Running Time and Algorithm Analysis

Problem 1:

- `public boolean insert(AnyType x)`
    - Algorithm: The insert method in the Binary Search Tree adds Item x into the tree and increases the count for x in the BinaryNode's hash map. First, the item is passed into the method as well as the binary node, which starts at the root and recursively goes through all the nodes until the correct position for the item is found. The binary node is checked to see if it is null, and if it is then a new binary node is created, and the value is set as a key for the new binary node's hash map. Next, the element of the binary node, which is a hash map, is set to a Map.Entry so that the item to be inserted and the key for the entry can be compared. Next, the result of the comparison (compareTo method is used) is stored in the

compareResult variable. The comparison result will be checked to determine the next steps to take. If the two values are equal, then that means that the value to be inserted is a duplicate, so entry's value, which is the counter for the key, is incremented by 1. If the value to be inserted is less than the entry's key, then the insert method will be recursively called on the binary node's left child. If the value to be inserted is greater than the entry's key, then the insert method will be recursively called on the binary node's right child. The method will return the binary node T at the end.

  - Running time: If n items are inserted, the best case for the insert method is O(logn) if the tree is balanced and there are no duplicates. The worst case for the insert method is O(n * logn) if the tree is skewed to one side and there are duplicates.

- `public boolean contains(AnyType x)`
  - Algorithm: The contains method is like the insert method in terms of logic. The hash map of the binary node is set to Map.entry once again so that the key can be compared to the value passed into method. The method starts at the root and works its way down to find the expected key. If binary node being searched is null, then the method will return false. If the passed value and the key in the binary node's hash map are equal, then this means that the value is present in the tree and the method returns true. If the passed value is greater than the key, then the contains method will be recursively called on the binary node's right child. If the passed value is less than the key, then the contains method will be recursively

called on the binary node's left child. If the value is never found, then the method will return false.

  - Running Time: The best case for the contains method is O(logn) if the tree is balanced. Since the contains method only sees if there is at least one instance of item, there is no need to search the duplicates, which means that the worst case running time would be O(n) if the tree is skewed to one side.

- `public int count(AnyType x)`
  - Algorithm: The count method is also like the insert method and contains method. The hash map of the binary node is set to Map.entry once again so that the key can be compared to the value passed into method. The method starts at the root and works its way down to find the expected key. If binary node being searched is null, then the method will return 0. If the passed value and the key in the binary node's hash map are equal, then the method will return the value (counter) of the key which tracks the number of duplicates. If the passed value is greater than the key and the right child of the node is not null, then the count method is recursively called on the binary node's right child. If the passed value is less than the key and the left child of the node is not null, then the count method is recursively called on the binary node's left child. If the value is never found, then the method will return 0.
  - Running Time: The best case for the count method is O(logn) if the tree is balanced. The worst case is O(n) in the case that the tree is skewed to one side. Once the key is found, then the method just returns the value/counter of that key which is constant.

- `public boolean removeOne(AnyType x)`
    - Algorithm: The removeOne method is like the other methods aside from a few differences. If the binary node is null, then the method will return true because the value already doesn't exist in the tree. If the passed value is equal to the node's key, then the method will check if the value/counter of the key is greater than 1. If it is, then the value/counter will be decremented by one. If it is only 1, then the key will be removed altogether, then the method will return true. If the passed value is greater than the key, then the removeOne method will be recursively called on the binary node's right child. If the passed value is less than the key, then the removeOne method will be recursively called on the binary node's left child. If the search or removal fails for some reason, then the method will return false.
    - Running time: The best case is O(logn) if the tree is balanced and there are no duplicates, so the value is removed completely. The worst case is O (n * logn) if the tree is skewed to one side and there are duplicates to search for.
- `public boolean removeAll(AnyType x)`
    - Algorithm: The removeAll method is almost identical to the removeOne method except that the method does not check how many duplicates there are and just removes the key from the tree completely regardless of the value/frequency count.
    - Running time: The best case is O(logn) if the tree is is balanced. The worst case is O(n) if the tree is skewed to one side. Duplicates do not matter, so there is no need to search for them.

Problem 2:

- Strategy 1: Scan the items in the order given; place each new item in the most-filled box that can accept it without overflowing. Use a priority queue to determine the box that an item goes in.
    - Algorithm: An ArrayList of items is initialized, as well as a scanner for the user to input the weights for items and a PriorityQueue with an anonymous comparator that prioritizes boxes based on how full they are. The program will keep prompting the user to enter weights for the items until they enter a -1 which will cause the loop to exit. In each loop, a new item is initialized with the entered weight, and the item is added to the items ArrayList. Then, the program moves to a for loop which will traverse the length of the items ArrayList. In this loop, the items will be sorted into boxes in the Priority queue depending on how much space is available in the boxes.
    - Running time: the running time for this strategy is O(2n) because the items are added to the ArrayList in linear time, and then they are added to the boxes one by one.
- Strategy 2: sort the items, placing the heaviest item first; then use the strategy above
    - Algorithm: the algorithm for this strategy is identical to the first one except that the items are sorted in descending weight order, then added to the boxes.This is accomplished using arrays.
    - Running time: Due to the increase in complexity caused by the sorting of the items by weight, the running time is $O(4n + n^2)$.