CS61B Spring 2016 Secret Section 4 Solutions

Mitas Ray, Khalid Shakur

Week of 2/25

1 Generics

Use Java's `compareTo()` method to write a generic function to find the greatest object in an array. Do not worry about exceptions `compareTo()` may throw and assume the input list is not empty.

```
public static <T> T findGreatest(T[] a) {
  //  int compareTo(T o)
  //  Compares this object with the specified object for order. Returns a negative
      integer, zero, or a positive integer as this object is less than, equal to,
     or greater than the specified object.
    T maxItem = a[0];
  for (int i = 1; i < a.length; i++) {
    if (maxItem.compareTo(a[i]) < 0) {
      maxitem = a[i];
    }
  }
  return maxItem;
}
```

2 Iterators

(a) Name and explain the purpose/use of the three methods that are outlined by the Java iterator interface.

hasNext() - Allows you to check if the iterator has more elements to return from calling next.
next() - Return the next element in the iteration.
remove() - Removes the last element returned by this iterator.

(b) Suppose that for some reason the Java iterator interface did not include the `hasNext()` method, and it was bad practice to implement one. How might you account for the fact that next() will at some point throw a `NoSuchElementException` exception without crashing your program? Why is it a good idea that `hasNext()` exists?
When calling the next method, it would be necessary to catch the NoSuchElementException exception. It is a good idea that hasNext() exists so that we can avoid handling the exception, and instead have a simple manner of predicting the end of iterators. Abstraction!

3 Runtime Analysis

Suppose we have an array of N strings of length N. What would be the runtime of a function that reversed each individual string's characters and the ordering of the strings in the array.

N * N + N or just N * N

4 Which Data Structure?

For each of the following parts, choose which data structure would best fits the constraints given and write it on the line provided. Choose from **Balanced Search Tree**, **Disjoint Sets**, and **HashSet**.

(a) Balanced Search Tree We want to store a set of strings for which we care about the alphabetical order.

Explanation: Out of the three data structures listed, only Balanced Search Trees maintains an ordering amongst the elements in a natural way.

(b) Hash Map We want to store a set of strings for which we want to perform lookup in constant time.

Explanation: Since we want to store a set of strings, and do not care about their connectedness to other elements, we can discard Disjoint Sets. This leaves us with Balanced Search Tree or HashSet. Out of the these two data structures, only HashSets allow for a constant time lookup of a certain element.

(c) Balanced Search Tree We want to store a large set of numbers, but we have space constraints.

Explanation: Since we want to store a set of numbers, and do not care about their connectedness to other elements, we can discard Disjoint Sets. This leaves us with Balanced Search Tree or HashSet.

(d) Disjoint Sets We want to represent the relationships between a group of people and determine if two people are friends. Two people can either be friends or enemies. If person A and person B are friends, then person A is also friends with all the friends of person B.

Explanation: Since we care about the connectedness between pairs of elements, the only data structure out of the three listed that handles connectedness naturally is Disjoint Sets.

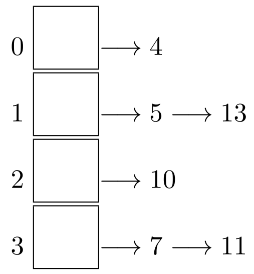
5 HashSet

(a) Currently we have 6 items in our HashSet. How many more elements do we need to insert before we upsize the HashSet from 4 buckets to 8 buckets if we upsize whenever the load factor factor reaches 2? See figure of HashSet below. Write your answer on the blank below.

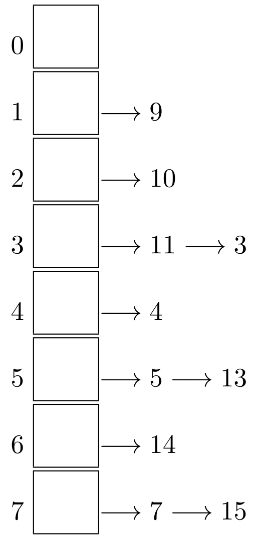
2

Explanation: Currently we have four buckets. Since we upsize whenever our load factor reaches 2, we know that we will upsize when the number of elements in the HashSet becomes 8. Since we currently have 6 items, inserting 2 more items will cause the HashSet to upsize.

(b) Draw the new HashSet beside the one below after inserting the integers 3, 14, 15, 9. The hashCode of an integer returns the integer itself. Follow the same principle of resizing as described in part (a).



Solution:



Explanation: Since we have added four new elements, looking on our answer from part (a), we need to upsize to eight buckets. We then reassign the integers based on the mod value of the integer with the number of buckets. We perform this operation for all existing integers. We then add the new integers into the HashSet. Note that the order of the numbers in each bucket does not matter.

(c) Currently we have 10 items in our HashSet, after adding four items from part (b). How many elements do we need to remove from the HashSet if we downsize by halving the number of buckets, whenever the load factor reaches 0.5? Use the HashSet you drew in part (b) as the current HashSet. Write your answer on the blank below.

6

Explanation: Currently we have eight buckets. Since we downsize whenever our load factor reaches 0.5, we know that we will downsize when the number of elements in the HashSet becomes 4. Since we currently have 10 items, removing 6 more items will cause the HashSet to downsize.

6 Binary Search Tree

Fill in the blank lines below for the method `isBinarySearchTree()`. This method checks whether a Binary Tree is a Binary Search Tree by following the property that the left child is less than the parent and the right child is greater than the parent.

```java
public class BinaryTree {

  public int value;
  public BinaryTree left;
  public BinaryTree right;

  public BinaryTree(int value) {
    this.value = value;
    left = null;
    right = null;
  }

  public boolean isBinarySearchTree() {
    // check if leaf node

    if (left == null && right == null) {
      return true;
    }

    // check left value

    if (left != null) {
      if (left.value > value) {
        return false;
      }
    }

    // check right value

    if (right != null) {
      if (right.value < value) {
        return false;
      }
    }

    if (left == null) {
      return right.isBinarySearchTree();
    } else if (right == null) {
      return left.isBinarySearchTree();
    } else {
      return right.isBinarySearchTree() && left.isBinarySearchTree();
    }

  }
}
```