Lab 10 Notes

CS 61BL
Summer 2019

Lab 10
July 12, 2019

Name:

SID:

Please complete this worksheet during your lab, and turn it in to your TA by the end of your section. You are encouraged to work with your neighbors collaboratively.

Section Number: 01 [✓ 02] 03 04 05 06 07 08 09 10 11 12

1 Comparators

1.1 Suppose we want to sort an **int** array by how far each element is from the value 42. To accomplish this, we might choose to use a Comparator, a method reference, or a lambda expression. Fill in the blanks in the following code such that each line printed by the main method would display [42, 32, 52, 83, 0].

Hint: Try squaring to obtain the relative absolute difference between two numbers.

```
import java.util.*;
public class Comp42 implements Comparator<Integer> {
        public int compare(Integer o1, Integer o2) {
                return Integer.compare(Math.abs(o1-42), Math.abs(o2-42));
        }
        public static int compare42(Integer i1, Integer i2) {
                return ______________________________________________;
        }
        public static void main(String[] args) {
            Integer[] arrComparator = {0, 32, 42, 52, 83};
            Arrays.sort(arrComparator, new Comp42());
            System.out.println(Arrays.toString(arrComparator));
            Integer[] arrFunction = {0, 32, 42, 52, 83};
            Arrays.sort(arrFunction, Comp42::compare42);
            System.out.println(Arrays.toString(arrFunction));
            Integer[] arrLambda = {0, 32, 42, 52, 83};
            Arrays.sort(arrLambda, (o1, o2) -> Integer.compare(                    ));
            System.out.println(Arrays.toString(arrLambda));
        }
}
```

same
but i instead
of o

2 Collections

2.1 An important operation provided by `Collection` classes is the `contains` method: finding whether a given item is in the `Collection`. For certain tasks in which this operation needs to be efficient, we need to choose a good data structure.

Let's take a look at some of the data structures we've already looked at. For each of the questions below, the answer is one of the five options:

- a constant number
- proportional to k
- proportional to N
- proportional to $\log N$
- proportional to N^2

2.1 Suppose that N integers are stored in an `IntList` object. How many comparisons are necessary in the worst case to determine if a given integer k occurs in the sequence?

N

Not sorted

2.2 Suppose that N comparable objects are stored in an `ArrayList`. How many comparisons are necessary in the worst case to determine if a given object k occurs in the list?

N

sorted but can only look at next elem.

2.3 Suppose that N integers are stored in increasing order in an `IntList` object. How many comparisons are necessary in the worst case to determine if a given object k occurs in the list?

logN comparisons (but N calls to .next)

2.4 Suppose that N integers are stored in increasing order in an array. How many comparisons are necessary in the worst case to determine if a given integer k occurs in the sequence?

Sorted and can jump around

logN

⇒ binary earch

3 Binary Search

3.1 Consider applying the binary search algorithm to a sorted doubly linked list (a list where each node has references to both its previous and next node). The variables low, high, and mid would then be references to nodes in the list. Will this work?

In fact, binary search will not work in any linked list structure because we do not have a quick way of moving the low, high, or mid references. In addition, the contains operation would have to take time proportional to the number of elements in the collection. Which of the steps in the array binary search algorithm shown below would be the bottleneck that causes a decrease in performance if a sorted doubly linked list is used, relative to a sorted array for the contains operation?

1. Set *low* to 0 and *high* to the length of the array minus 1. The value we're looking for - we'll call it k - will be somewhere between position *low* and position *high* if it's in the array.
2. While $low \leq high$, do one of the following:
   (a) Compute *mid*, the middle of the range $[low, high]$, and see if that's k. If so, return *mid*.
   (b) Otherwise, we can cut the range of possible positions for k in half, by setting *high* to $mid - 1$ or by setting *low* to $mid + 1$, depending on the result of the comparison.
3. If the loop terminates with $low > high$, we know that k is not in the array, so we return some indication of failure.

○ Step 1, because it takes extra time to figure out the length of the linked-list

○ Step 2a, because it takes extra time to check if we have found the key

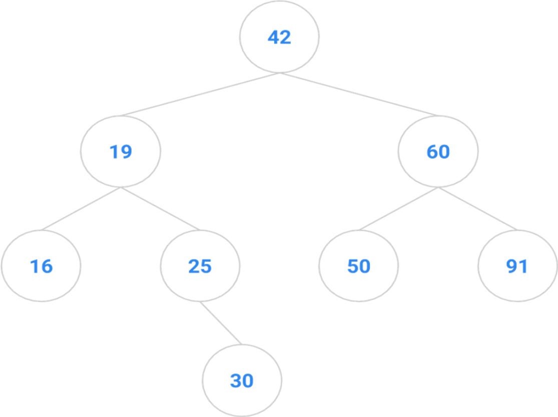
○ Step 2a, because it takes extra time to calculate the index of the middle node

● Step 2a, because takes extra time to access the middle nodes

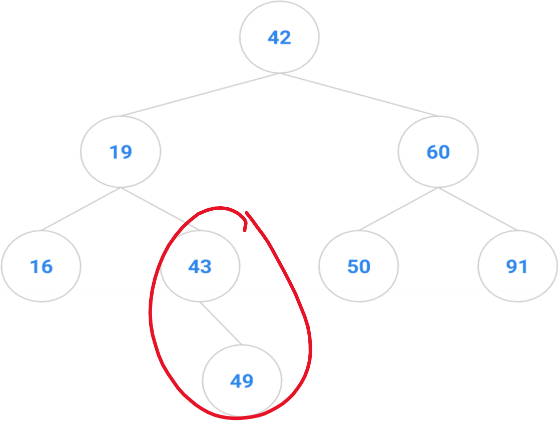
4 BST Identification

4.1 Now, let's practice identifying whether certain binary trees are binary search trees. For each of the following questions, discuss with your partner what nodes (can be multiple) must be removed from each tree to make it a binary search tree. Your answer should be in either of the following forms:

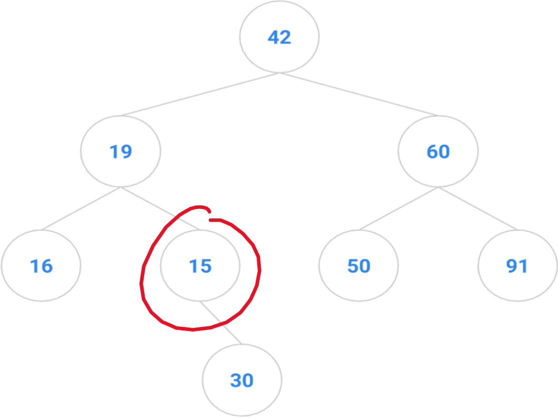
- space separated list of numbers
- "None" if it is already a BST



1. good



2. 43,49

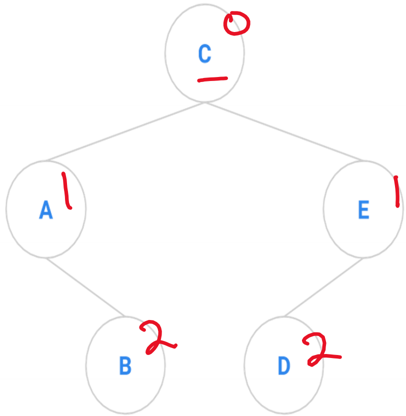


3. 15

5 Comparison Counting

5.1 How many comparisons between keys are necessary to produce the sample tree shown below? Ignore equality checks.

Remember: "However, to minimize restructuring of the tree and the creation of internal nodes, we choose in the following exercises to insert a new key only as a new leaf."



6

6 Binary Search Trees

6.1 Suppose we know the height `H` and number of nodes `N` of a BST. Can we determine whether or not this BST is minimum-BST-height without having to check the values of each node? Why or why not?

min height = $\log_2 N$, if height = $\log_2 N \Rightarrow$ min height

don't need to