DSA Tutorial Week 3 - Trees and Search

Find All Source Code on the DSA Tutorial Github: DSA-Tutorial-Exercises

---

Problem 0 - Preparation *!examine before the tutorial!*

You are given the following set of numbers:

```
15, 23, 5, 7, 9, 25, 19, 17
```

a. Construct a binary search tree (BST) using these numbers. Begin by inserting them into the BST in the order given. Illustrate the process using pen and paper, showing the tree after each insertion.

b. Once your tree is constructed, perform the following operations:
   i. Insert an additional number, 13, into your BST and redraw the tree.
   ii. Delete a number of your choice from the BST and redraw the tree.
   iii. Perform an in-order traversal of the BST and list the resulting sequence of numbers.

c. After completing these operations, focus on binary search:
   i. Choose three numbers from your final tree: one that exists in the tree, one that does not, and the root number.
   ii. Using *binary search*, describe the process of searching for each of these numbers in your BST. Illustrate each step of the process.

d. Create an Array from BST: Convert your final BST into a sorted array. An in-order traversal of the BST will give you the elements in ascending order. Record this sorted array.
   i. Choose the same three numbers that you used for the binary search on the BST (one present in the tree, one absent, and the root number).
   ii. For each of these numbers, perform a binary search on the extracted array. Document each step of your search process, noting the middle element you compare with and the sub-array (or sub-section of the array) you choose to continue the search in after each comparison.

To verify your work and gain additional insights, use the online tool visualgo (https://visualgo.net/en/bst). Compare the tool's output with your own work to understand any discrepancies and to solidify your comprehension of BSTs and binary search.

Problem 1 - Binary Tree Maximum Path Sum

Write an algorithm to find the *maximum path sum* in a binary tree. A path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain *at least one node* and *does not need to go through the root*.

Example:

Consider the following binary tree:

```
      10
     /  \
    2   10
   / \    \
  20  1   -25
          /  \
         3    4
```

In this tree, some of the possible paths and their sums are:

Path: 20 → 2 → 10 → 10	Sum = 42
Path: 1 → 2 → 10	Sum = 13
Path: 10 → 10	Sum = 20
Path: 3 → -25 → 4	Sum = -18
Path: 3 → -25 → 10	Sum = -12

However, the maximum path sum for this tree is 42, which is achieved by the path 20 → 2 → 10 → 10.

1. Start by discussing the problem and finding an algorithm in pseudocode.
   How will it traverse the tree? What steps are needed?
   Consider the elements of the recursion, how does it branch, what does it return, what is the base-case?
2. When you have a grasp of the algorithm, implement the function `maxPathSum(TreeNode root)` in the attached **BinaryTreeMaximumPathSum.java** that returns the maximum path sum.
3. The function should use recursion to traverse the tree and calculate the sum of each possible path.
4. At each node, consider the maximum sum obtained by including that node and either of its left or right subtrees, or both.
5. The maximum path sum for a node should be the maximum of:
   a. The node's value.
   b. Node's value + maximum path sum of the left subtree.
   c. Node's value + maximum path sum of the right subtree.
   d. Node's value + maximum path sum of both subtrees.
6. Keep track of the global maximum path sum during the traversal.

Problem 2 - Receiver Roster[2]

Coach Bell E. is trying to figure out which of her football receivers to put in her starting lineup. In each game, Coach Bell wants to start the receivers who have the highest performance (the average number of points made in the games they have played), but has been having trouble because her data is incomplete, though interns do often add or correct data from old and new games. Each receiver is identified with a unique positive integer jersey number, and each game is identified with a unique integer time.

Describe a data structure supporting the following operations, each in ***worst-case O(log n)*** time, where n is the number of games in the database at the time of the operation. Assume that n is always larger than the number of receivers on the team.

`record(g, r, p)`	record **p** points for jersey **r** in game **g**
`clear(g, r)`	remove any record that jersey **r** played in game **g**
`ranked_receiver(k)`	return the jersey with the **k**th highest performance

There's no need to implement the methods, the goal is to find and design an appropriate data structure.

1. Design a data structure that meets the operation requirements.
2. Explain your choices and how they contribute to the efficiency of the system.
3. Discuss the potential challenges and how your design addresses them.

Pointers

- How should each node represent games and the performance of various receivers? Consider the information each node must store.
- How will you store data for each game and corresponding receiver performances? Think about efficient access and update of this data.
- A receiver's performance is an average across multiple games. How will you efficiently track and update this?
- What's the mechanism for inserting or updating a receiver's performance in a game? How does each operation affect the overall data structure?
- When recording or clearing data, how will you ensure the receiver's average performance is accurately updated?
- How can you efficiently determine the jersey number of the receiver with the ***k*th** highest performance? Consider how to track and sort performances.
- What primary and auxiliary data structures will you use? How will they interact with each other?

[2] Source:
https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/resources/mit6_006s20_prob4sol/

Problem 3 - Find Prime[3]

Consider a binary tree containing **N** integer keys whose values are all less than **K**, and the following `Find-Prime` algorithm that operates on this tree. The algorithm and supporting functions are given below. Your task is to analyze their ***time complexity***.

```
Function Find-Prime(T )
      x = Tree-Min(T )
      while x ≠ nil
            x = Tree-Successor(x)
            if Is-Prime(x.key)
            return x
      return x

Function Is-Prime(n)
      i = 2
      while i · i ≤ n
            if i divides n
                  return false
            i = i + 1
      return true
```

```
Function Tree-Successor(x)
      if x.right ≠ nil
            return Tree-Minimum(x.right)
      y = x.parent
      while y ≠ nil and x == y.right
            x = y
            y = y.parent
      return y

Function Tree-Minimum(x)
      while x.left ≠ nil
            x = x.left
      return x
```

Pointers

1. **Understand Each Function:** Begin by understanding what each function does: `Find-Prime` searches for prime numbers in a binary tree, `Is-Prime` checks if a number is prime, `Tree-Successor` finds the next node in in-order traversal, and `Tree-Minimum` finds the smallest node.
2. First analyze the functions' **individual time complexities**, starting with `Is-Prime`. Next, determine the complexity of `Tree-Minimum` and `Tree-Successor`. Consider the structure of the binary tree, particularly its height h.
3. `Find-Prime` calls `Tree-Successor` repeatedly and might call `Is-Prime` for each node. **Combine their complexities** to find the overall complexity of `Find-Prime`.
4. Consider the **best** and **worst-case** scenarios for the tree's structure (e.g., a balanced tree vs. a skewed tree).
5. Write down the ***final complexity*** and explain how you arrived at it.

Next, use the provided ***TreeNode.java*** and ***BinarySearchTree.java*** and implement the `find-prime, is-prime, tree-successor` and `tree-minimum` functions in Java.

[3] Source: https://www.inf.usi.ch/carzaniga/edu/algo19s/exercises.pdf