TRINITY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE
CPSC 215: Data Structures and Algorithms
Instructor: Dr. Chandranil Chakraborttii
Spring 2025

Graded Laboratory 2 (Thursday)

Academic Honesty Policy

In working on programming assignments, you may discuss broad issues of interpretation and understanding and general approaches to a solution. However, conversion to a specific solution or to program code must be your own work. Programming assignments are expected to be the work of the individual student, designed, and coded by him or her alone. Violations are easy to identify and will be dealt with promptly according to the Academic Integrity and Intellectual Dishonesty outlined in the Student Handbook.

- Copying another person's programs or encouraging or assisting another person to commit plagiarism is cheating. In particular, the following activities are strictly prohibited:
- Giving and receiving help in the actual development of code or writing of an assignment.
- Looking at another person's code or showing your code to another person.
- Sharing a copy of all or part of your code regardless of whether that copy is on paper or in a computer file.
- Turning in the work of any other person(s) (former students, friends, textbook authors, people on the Internet, etc.) and representing it as your own work.
- Fabricating compilation or execution results.
- Use of AI ChatBots for any help regarding the assignment is **strictly prohibited.**

The penalty for cheating is a failing grade (F) for the course, and the student is asked to appear at an Academic Dishonesty Hearing. In addition, the College also places a record of the incident in the student's permanent record. It is your responsibility to protect your work from unauthorized access. Do not discard copies of your programs in public places. Do not leave computers unattended and copies of output lying around.

Instructions:

- Please read the questions carefully
- You will need to use Generics as discussed in the class lectures.
- You can review all class materials for this assignment.
- Do not start coding right away! Think first about a strategy and commit your algorithm on a piece of paper. Use stepwise refinement strategy as you start coding.

Implementing an Augmented Binary Search Tree for Order Statistics

You are provided with a basic implementation of a Binary Search Tree (BST). In this assignment, your task is to extend that basic BST to support additional order-statistic operations. To accomplish this, you must augment each node with a field that keeps track of the size of its subtree (including itself). This extra information will allow you to efficiently perform the following advanced operations:

- **kthSmallest(k):** Returns the kth smallest element stored in the BST.
- **rank(key):** Returns the rank (i.e., the position in the sorted order) of a given key.

Additionally, you must ensure that the subtree size is correctly updated during all modifications to the tree (insertion and deletion).

How It Works

Node Augmentation

- **Subtree Size:**
    - Each node will include an extra integer field, size, which equals the number of nodes in the subtree rooted at that node (including the node itself).
    - When a node is created, its size should initially be 1.
    - On every insertion and deletion, update the size field of each ancestor accordingly.

Order-Statistic Operations

1. **Finding kth Smallest Element:**
    - Start at the root.
    - Let L be the size of the left subtree.
    - If k == L + 1, the current node is the kth smallest.
    - If k <= L, the kth smallest lies in the left subtree.
    - Otherwise, search in the right subtree for the element with rank k - (L + 1).
2. **Rank of an Element:**
    - The rank of a given key is its position in the in-order traversal of the BST.
    - When searching for the key, if you move to the right child, add the size of the left subtree (of the node you're leaving) plus one (for the node itself) to the rank accumulator.
    - If the key is not present in the tree, return an indicator (such as -1).

Example Scenario

Assume you insert the following keys in order:
{15, 10, 20, 8, 12, 17, 25}

- The resulting augmented BST would have every node's size correctly set. For example:
  - The root (15) will have a size of 7.
  - The left child (10) will have a size of 3, and so on.
- To find the **3rd smallest element**, your algorithm would use the left subtree sizes to traverse the BST and return the correct node.
- To find the **rank of key 17**, the algorithm would compute its in-order position considering the sizes of the left subtrees encountered during the search.

Input Files

1. **AugmentedBST.java**
   - **Node Definition: [Completed]**
     Extend your basic BST node to include an additional field:
   - **BST Operations:**
     Ensure your insert, search, and delete methods update the size field appropriately.
   - **insert(key): [Need to Complete] [10 points]**
     Standard BST insertion but update the size for every node along the insertion path.
   - **search (key):** Standard BST search implementation **[Need to Complete]**
   - **delete(key): [Completed]**
     Standard BST deletion while maintaining correct subtree sizes.
   - **Order-Statistic Methods: [Need to Complete]**

     - **kthSmallest(int k): [25 points]**
       Return the kth smallest key by using the subtree sizes.
     - **rank(int key): [25 points]**
       Return the rank (position in sorted order) of a given key.
       Hint: Use an accumulator variable during the search.
   - **printInOrder(): [Need to Complete] [20 points]**
     Optionally, print the BST in-order along with each node's key and its subtree size for debugging.

2. **AugmentedBSTTester.java**

- **Test Operations: [Need to Complete] [20 points]**
  - Create an instance of your augmented BST.

    - Insert a set of keys (choose at least 10–15 keys with varied values to build a nontrivial tree).
    - Display the in-order traversal of the BST (including each node's size).
    - Perform kthSmallest(k) queries for value of **3**.
    - Perform rank(key) queries for keys that are in the tree and for some keys that are not present. [rankQueries]
    - After deletion, run a kth smallest query (3rd smallest) and a rank query (17)
- The tester should print clear, detailed output for each operation.

Implementation Details and Specifications

- **Initial Data Set:**
  Insert a set of at least 10–15 integer keys (e.g., {15, 10, 20, 8, 12, 17, 25, 6, 11, 13, 19, 22}) to create an interesting tree structure.
- **Augmenting Nodes:**
    - Every time a node is added or removed, update the size field.
    - Consider writing a helper method updateSize(Node node) that recalculates the size based on its children.
- **Order-Statistic Methods:**
    - **kthSmallest(k):**
      Implement using recursion or iteration, using the size of the left subtree to decide traversal.
    - **rank(key):**
      Implement using a similar traversal approach, summing up sizes as needed.
- **Testing:**
    - Ensure your in-order output shows keys in sorted order along with the correct sizes.
    - Include clear console output demonstrating the results of kth smallest and rank queries, as well as the effects of deletion on the augmented structure.

Visual Example

Assume you insert the keys in the following order into your augmented BST:

{15, 10, 20, 8, 12, 17, 25}

1. BST Construction and Node Augmentation

Each node in the augmented BST not only stores its key and children but also a size field representing the number of nodes in the subtree rooted at that node (including itself).

Step-by-step Insertion:

1. **Insert 15:**
   The tree is empty, so the node with key **15** becomes the root.
   - **Structure:**

     ```
     15 (size=1)
     ```

2. **Insert 10:**
   - Since **10 < 15**, it goes to the left of **15**.
   - Update the sizes:
     - Node **10** is created with size 1.
     - Node **15** now has size = 1 (itself) + 1 (left) = 2.
   - **Structure:**

     ```
       15 (size=2)
      /
     10 (size=1)
     ```

3. **Insert 20:**
   - Since **20 > 15**, it goes to the right of **15**.
   - Update sizes:
     - Node **20** is created with size 1.
     - Node **15** now: size = 1 (itself) + 1 (left subtree, node 10) + 1 (right subtree, node 20) = 3.
   - **Structure:**

     ```
        15 (size=3)
       /  \
     10 (1) 20 (1)
     ```

4. **Insert 8:**
   - Compare **8** with **15**: since **8 < 15**, go left.
   - Compare **8** with **10**: since **8 < 10**, go left.
   - Insert **8** as left child of **10**.
   - Update sizes:
     - Node **8** is new with size 1.
     - Node **10** now becomes: size = 1 (itself) + 1 (left child 8) = 2.
     - Node **15** now: size = 1 (itself) + 2 (from left, node 10) + 1 (node 20) = 4.
   - **Structure:**

     ```
           15 (size=4)
          /    \
     10 (size=2)    20 (size=1)
     ```

```
      /
    8 (size=1)
```

5. **Insert 12:**
   - Since **12 < 15**, go left.
   - Compare **12** with **10**: **12 > 10**, so it goes to the right of **10**.
   - Insert **12** with size 1.
   - Update sizes:
     - Node **10** becomes: size = 1 (itself) + 1 (left, node 8) + 1 (right, node 12) = 3.
     - Node **15** becomes: size = 1 (itself) + 3 (node 10) + 1 (node 20) = 5.
   - **Structure:**

```
          15 (size=5)
         /      \
  10 (size=3)  20 (size=1)
    /     \
 8 (1)     12 (1)
```

6. **Insert 17:**
   - Since **17 > 15**, go right.
   - Compare **17** with **20**: since **17 < 20**, go left of **20**.
   - Insert **17** as left child of **20** with size 1.
   - Update sizes:
     - Node **20** becomes: size = 1 (itself) + 1 (left, node 17) = 2.
     - Node **15** becomes: size = 1 (itself) + 3 (from node 10) + 2 (from node 20) = 6.
   - **Structure:**

```
             15 (size=6)
            /     \
     10 (size=3)   20 (size=2)
    /     \              /
 8 (1)     12 (1)    17 (1)
```

7. **Insert 25:**
   - Since **25 > 15**, go right.
   - Compare **25** with **20**: **25 > 20**, go right.
   - Insert **25** as right child of **20** with size 1.
   - Update sizes:
     - Node **20** now becomes: size = 1 (itself) + 1 (node 17) + 1 (node 25) = 3.
     - Node **15** updates: size = 1 (itself) + 3 (node 10) + 3 (node 20) = 7.
   - **Final Structure:**

```
              15 (size=7)
             /     \
      10 (size=3)       20 (size=3)
     /     \          /     \
  8 (1)     12 (1)  17 (1)     25 (1)
```

---

2. Order-Statistic Operations Using the Augmented Information

Finding kth Smallest Element

Assume you want to find the **3rd smallest element** in the tree:

1. **Start at the Root (15):**
   - Determine left subtree size:
     - Node **10** has size **3**.
   - Compare **k** (3) with left subtree size + 1:
     - Left subtree size = 3, so if k were 4, that would be the root.
     - Since **3 <= 3**, the kth smallest must be in the left subtree.
2. **Move to Node (10):**
   - Determine its left subtree size:
     - Node **8** has size **1**.
   - Now, compare **k** (still 3, because we are still counting from the beginning) with left subtree size + 1:
     - k = 3, left subtree (of node 10) size + 1 = 1 + 1 = 2.
     - Since **3 > 2**, the kth smallest element is not node 10.
     - Adjust **k:** subtract **(left size + 1)**; now **k = 3 - 2 = 1**.
     - Move to the right subtree of node 10.
3. **Move to Right Child (12) of Node 10:**
   - For node **12**, the left subtree is empty (size 0).
   - Here, k = 1; since left subtree size (0) + 1 equals 1, node **12** is the 1st smallest element in this subtree.
   - **Conclusion:** The 3rd smallest element in the entire BST is **12**.

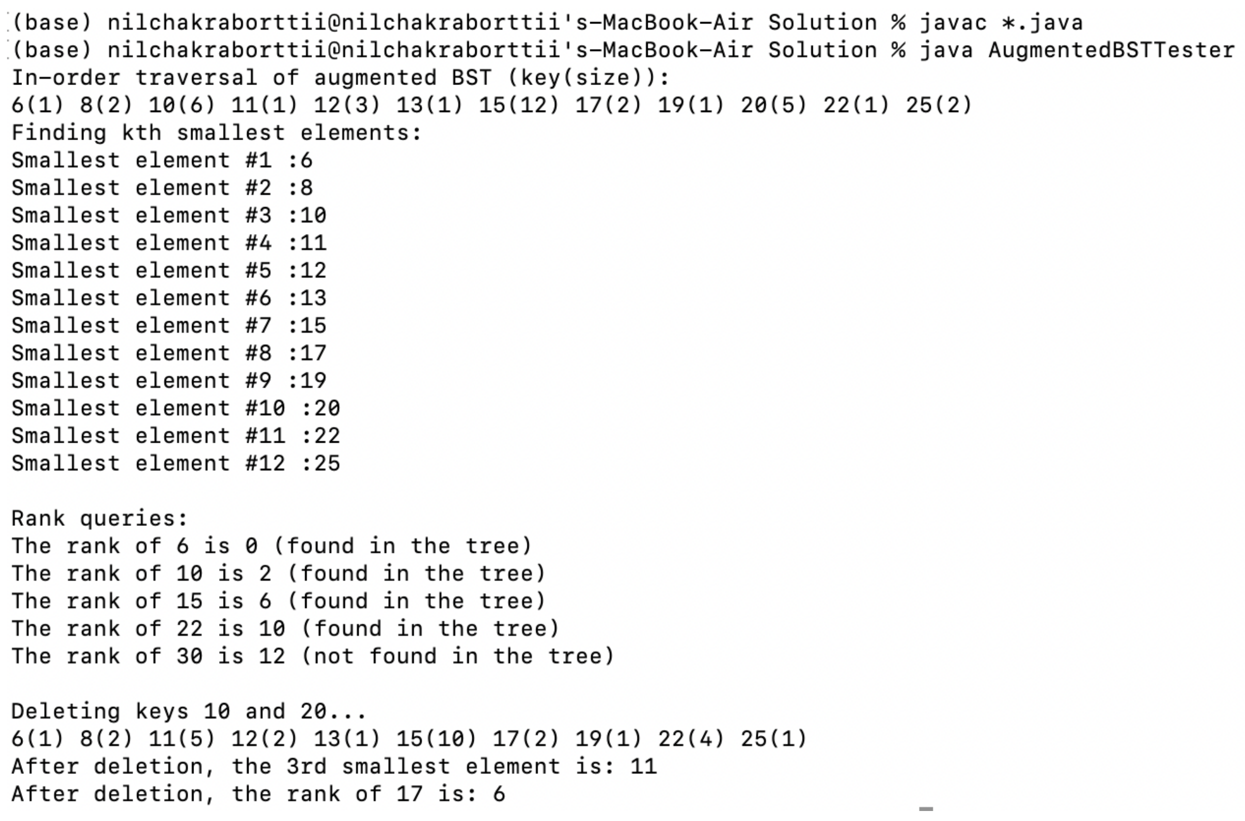
Rank of a Given Key

Assume you want to compute the **rank** (i.e., the in-order position) of key **17**:

1. **Start at the Root (15):**
   - Compare **17** with **15**: since **17 > 15**, then all nodes in the left subtree of **15** (which are 3 nodes: 10, 8, 12) plus the root node **15** itself come before **17**.
   - **Current rank accumulator:** 3 (left subtree of 15) + 1 (for 15) = **4**.
   - Move to the right subtree (node **20**) to search for key **17**.
2. **At Node 20:**

  - Compare **17** with **20**: since **17 < 20**, move to the left child of **20**.
  - Now, the rank becomes **4 + 0** (for the empty left subtree of node **17**) + **1** (for node **17** itself) = **5**.
3. **At Node 17 (Left Child of 20):**
  - We find **17**.
  - The rank for **17** is the accumulator we had; in this case, rank = **4** (i.e., in an in-order traversal, **17** is the 4th element).

Sample Output



```
(base) nilchakraborttii@nilchakraborttii's-MacBook-Air Solution % javac *.java
(base) nilchakraborttii@nilchakraborttii's-MacBook-Air Solution % java AugmentedBSTTester
In-order traversal of augmented BST (key(size)):
6(1) 8(2) 10(6) 11(1) 12(3) 13(1) 15(12) 17(2) 19(1) 20(5) 22(1) 25(2)
Finding kth smallest elements:
Smallest element #1 :6
Smallest element #2 :8
Smallest element #3 :10
Smallest element #4 :11
Smallest element #5 :12
Smallest element #6 :13
Smallest element #7 :15
Smallest element #8 :17
Smallest element #9 :19
Smallest element #10 :20
Smallest element #11 :22
Smallest element #12 :25

Rank queries:
The rank of 6 is 0 (found in the tree)
The rank of 10 is 2 (found in the tree)
The rank of 15 is 6 (found in the tree)
The rank of 22 is 10 (found in the tree)
The rank of 30 is 12 (not found in the tree)

Deleting keys 10 and 20...
6(1) 8(2) 11(5) 12(2) 13(1) 15(10) 17(2) 19(1) 22(4) 25(1)
After deletion, the 3rd smallest element is: 11
After deletion, the rank of 17 is: 6
```

Submission Guidelines:

- Make sure you completed all sections with “To do” statements and added JavaDoc in both classes.
- Upon the completion of your lab, upload your submission to the Moodle course website as a single Zip file. **[Must submit by 4:10 pm]**

———