Opening Questions

Previously we discussed the order-statistic selection problem: given a collection of data, find the kth largest element. We saw that this is possible in $O(n)$ time for each kth element you are trying to find. The naïve method would just be to sort the data in $O(n \lg n)$ and then access each kth element in $O(1)$ time.

1. Which of these two methods would you use if you knew you would be asked to find multiple kth largest elements from a set of static data? If you're doing $\geq \lg n$ queries, it would be better to sort the data and then constantly choose the value based off of the index. If you're doing $< \lg n$, it would be better to run the $O(n)$ each time.

2. What if our collection of data is changing (dynamic), would either of these approaches work efficiently for a collection of data that has inserts and deletes happening? No, you would have to re-run either algorithm for either.

Augmenting Data Structures

For particular applications, it is useful to modify a standard data structure to support additional functionality. Sometimes the modification is as simple as by storing additional information in it, and by modifying the ordinary operations of the data structure to maintain the additional information.

Dynamic Order-Statistic Trees (Augmenting Balanced Trees)

1. What can we do with a binary search tree (and more efficiently with a balanced binary search tree)? We can insert and delete items, and if it's balanced, we can do that in $O(\lg n)$.

2. Consider the naïve method of finding the kth largest item is to sort the array and then access the kth item in $O(1)$ time. Can we do this with a (balanced) BST? What is we augment it (HINT: recall how counting sort worked)? A BST can do this by keeping track of the size of the subtree.

3. What is the recursive formula to find the size of a subtree at node x?

Algorithm 13.1 Size of a subtree at a node

```
1: function SubtreeSize(node)                    ▷ The size of null leaves is 0.
2:     size ← 1
3:     if node.left ≠ null then
4:         size ← size+ SubtreeSize(node.left)
5:     end if
6:     if node.right ≠ null then
7:         size ← size+ SubtreeSize(node.right)
8:     end if
9:     return size
10: end function
```

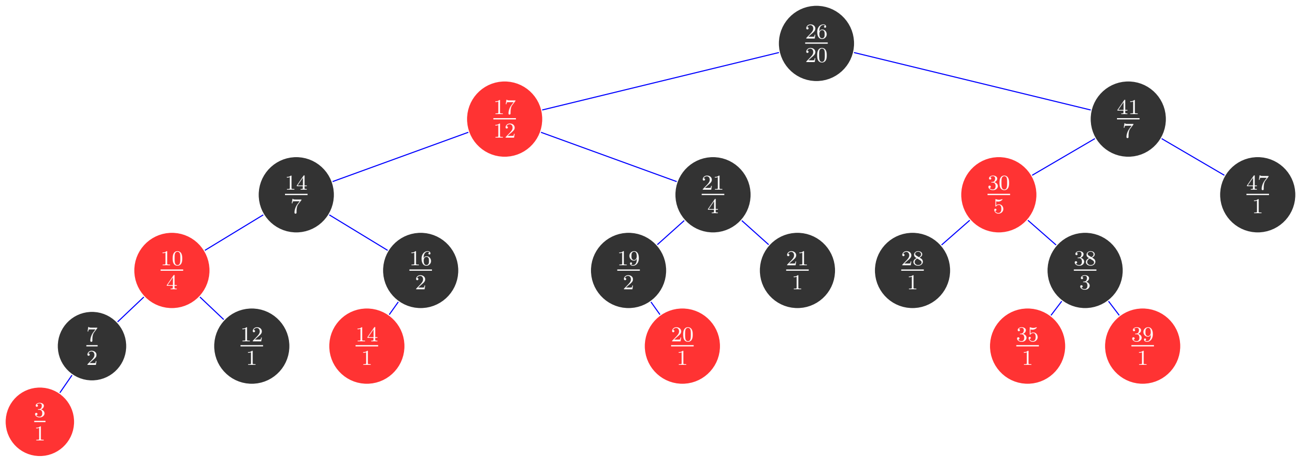

Figure 13.1: An order-statistic tree, which is an augmented red-black tree. In addition to its usual attributes, each node x has an attribute $x.size$, which is the number of nodes, other than the sentinel, in the subtree rooted at x.

4. Discuss in detail how would you keep the size at a node correct when you insert a new node, and possibly rotate to keep the tree balanced?

Algorithm 13.2 Left Rotation of a BST

```
function LEFT-ROTATE($T$, $x$)
    $y \leftarrow x.right$                                  ▷ Set $y$
    $x.right \leftarrow y.left$  ▷ Turn $y$'s left subtree into $x$'s right subtree
    if $y.left \neq T.nil$ then
        $y.left.p \leftarrow x$
    end if
    $y.p \leftarrow x.p$                          ▷ link $x$'s parent to $y$
    if x.p == T.nil then
        $T.root \leftarrow y$
    else if $x == x.p.left$ then
        $x.p.left \leftarrow y$
    else
        $x.p.right \leftarrow y$
    end if
    $y.left \leftarrow x$                             ▷ Put $x$ on $y$'s left
    $x.p \leftarrow y$
end function
```

Algorithm 13.3 Insert a node into a BST

```
function TREE-INSERT($T$, $z$)
    $y \leftarrow$ null
    $x \leftarrow T.root$
    while $x \neq$ null do
        $y \leftarrow x$
        if $z.key < x.key$ then
            $x = x.left$
        else
            $x = x.right$
        end if
    end while
    $z.p \leftarrow y$
    if $y ==$ null then
        $T.root \leftarrow z$                                  ▷ Tree $T$ w
    else if $z.key < y.key$ then
        $y.left \leftarrow z$
    else
        $y.right \leftarrow z$
    end if
end function
```

After you would add or delete the node from the BST, when you go back up the recursion, you can re-count the sizes after any rotations that may be necessary. (In any rotations, the only sizes that need to get updated are the nodes that are rotating, the size of the parent would stay the same.)

5. Discuss in general how would you keep the size at a node correct when you delete a node,

and possibly rotate to keep the tree balanced? You can maintain these sizes on deletions and rotations. As the Red-Black deletion rules says, if the node you're deleting has 0 or 1 children, then the node was in the search path. If the node has two children, you would have to find the successor and update the search path.

6. How can we use the augmented data at each node (size) in a balanced binary search tree to solve the kth largest item problem? We can use the size of a node to check how many values are less than or greater than it (by subtracting the size of the subtree from the size of the root), which can help us determine if the required rank is within that subtree. If not, we can cut out that subtree to reduce the runtime.

7. How can we use the augmented data at each node (size) in a balanced binary search tree so when given a pointer to a node in the tree, we can determine its rank (the index position of the node of the tree data in sorted order)?

Algorithm 13.4 Order Statistic with a BST for the ith smallest

```
function SelectSmallest($x$:Node, $i$:int)        ▷ Initial call: SelectSmallest($tree.root$, $i$)
    $y \leftarrow$ SubtreeSize($x.left$) $+1$
    if $i == y$ then
        return $x.value$
    else if $i < y$ then
        SelectSmallest($x.left$, $i$)
    else
        SelectSmallest($x.right$, $i - y$)
    end if
end function
```

Algorithm 13.5 Order Statistic with a BST for the kth largest

```
function SelectLargest($x$:Node, $k$:int)        ▷ Equivalent to SelectSmallest($x$, $n - k$)
    $y \leftarrow$ SubtreeSize($x.right$) $+1$
    if $k == y$ then
        return $x.value$
    else if $k < y$ then
        SelectLargest($x.right$, $k$)
    else
        SelectLargest($x.left$, $k - y$)
    end if
end function
```

8. Why not just use this approach always (not just with dynamically changing data) instead of the $O(n)$ one? The runtime to build and insert into the initial BST would be $O(n \lg n)$.

9. Can we use this approach on a regular BST? No, we need a balanced BST.