2 Week 5: February 26

2.1 Rank and Select Queries

Suppose we want to design an ordered dictionary data structure that not only supports the standard operations Find, Insert, Delete, Pred, and Succ, but also supports two additional operations:

- Rank(k): Return the *rank* of k: the number of items in the dictionary with value at most k.
- Select(r): Return the rth smallest item in the dictionary, that is, the item with rank r.

For example, if the dictionary holds the 26 letters of the alphabet, then Rank(J) should return the integer 10, and Select(18) should return the letter R.

We can support the standard operations using a standard binary search tree. With only a small modification to these data structures, we can support Rank and Select queries with exactly the same worst-case (or amortized) time as Find.

Specifically, we **augment** the binary search tree by storing an additional value v.size at each node v, which is equal to the size of the subtree rooted at v, in addition to the search key v.key and the child pointers v.left and v.right. Then the Rank and Select queries can be implemented recursively as follows; in both algorithms, the input argument v is (a pointer to) a node in the tree.

```
Rank(v, k):
    if v == NULL
        return 0

    if v.left == NULL
        leftsize ← 0
    else
        leftsize ← v.left.size

    if k == v.key
        return 1 + leftsize
    else if k < v.key
        return Rank(v.left, k)
    else
        return leftsize + 1 + Rank(v.right, k)

Select(v, r):
    if v == NULL or r < 0 or r > v.size
        explode

    if v.left == NULL
        leftsize ← 0
    else
        leftsize ← v.left.size

    if r == leftsize + 1
        return v
    else if r < leftsize + 1
        return Select(v.left, r)
    else
        Select(v.left, r - leftsize - 1)
```

The Rank algorithm follows the usual search path from the root to the node with key k. Whenever the search path turns to the right right, we know that both v and every node in the right subtree of v have keys smaller than k, so we have identified v.left.size $+ 1$ nodes smaller than k.

Conversely, in the Select algorithm, if $r > v$.left.size $+ 1$, then we know that the target of our search must have rank $r - v$.left.size $- 1$ in the right subtree of v. Thus Select also follows the standard binary search path to its target node.

These augmented trees are sometimes called *order statistic trees* by people who feel the need to name every single variant of every data structure.

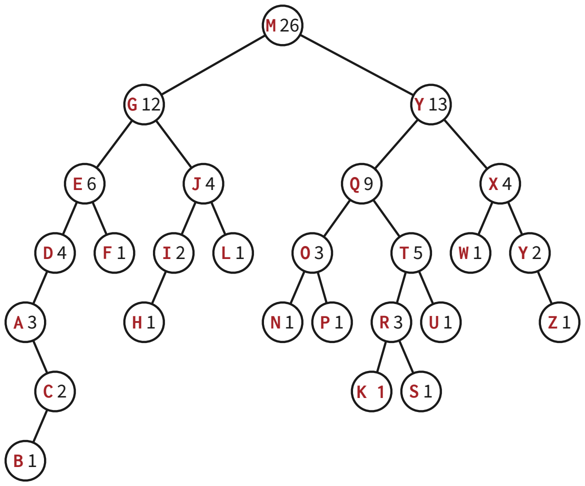

Figure 1: An order-statistic tree for the alphabet; each node stores a search key and the number of nodes in its subtree.

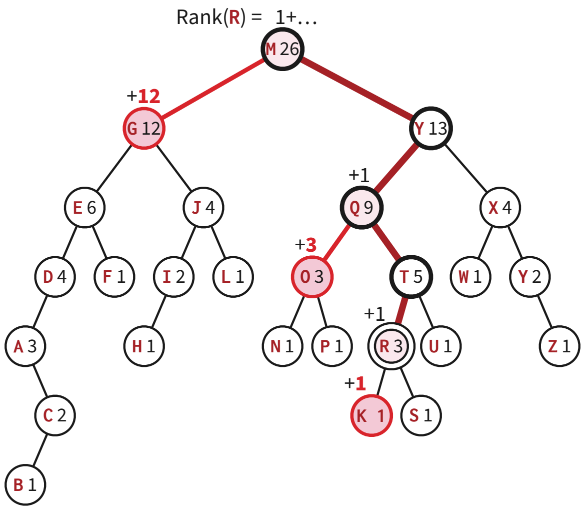

Figure 2: Answering the query Rank(R)

2.2 Construction and Updates

Of course we can't just magically assume that the $v.\mathsf{size}$ is correct; we have to maintain this value as nodes are inserted into or deleted from the subtree rooted at v.

- When the tree is first constructed, we can compute $v.\mathsf{size}$ for all nodes v in $O(n)$ time using a single post-order traversal:
  - If v is a leaf, then $v.\mathsf{size} = 1$.
  - Otherwise, $v.\mathsf{size} = v.\mathsf{left.size} + 1 + v.\mathsf{right.size}1$.
- Whenever we insert a new node x into the tree, we add 1 to $v.\mathsf{size}$ for every node v on the search path to x.
- Whenever we delete a node x from the tree, we first subtract 1 from $v.\mathsf{size}$ for every node v on the search path to x.

These extra operations increase the construction time, insertion time, and deletion time by at most a small constant factor.

2.3 . . . In *Balanced* Binary Search Trees

We can support all the standard ordered dictionary operations in $O(\log n)$ time (possibly amortized and/or expected) using any *balanced* binary search tree: AVL tree, red-black tree, scapegoat tree, splay tree, treap, skip list, etc. (The query and updates times for some of these data structures, are amortized and/or randomized.) With some care, we can support Rank and Select in the same running time. The precise details depend on how the binary search tree is kept balanced. For example:

- If we are using a tree that maintains balance by rotations, such as an AVL tree or a splay tree, we recalculate $v.\mathsf{size}$ after any rotation that changes the children of v. Each rotation changes the children of at most two vertices, so this recalculation takes only $O(1)$ time per rotation.

- If we are using a splay tree, both Rank and Select should splay the target node before returning a result, just like Find.

2.4 Other Augmentation: Prefix Queries

This same augmentation idea generalizes to other kinds of queries. For example, Suppose each item in our ordered dictionary has a positive *weight*, which is independent of its search key. (We can store the weight of any item a field $v.\mathsf{wt}$ at the corresponding node v.) Consider the following terribly-named queries.

- $\mathsf{TotalWtLess}(k)$: Return the total weight of all items with less than k
- $\mathsf{TotalWtRank}(r)$: Return the total weight of items with the r smallest keys
- $\mathsf{RankByWt}(w)$: Find the largest rank r such that $\mathsf{TotalWtRank}(r) \leq w$
- $\mathsf{SelectByWt}(w)$: Find the largest search key k such that $\mathsf{TotalWtLess}(k) \leq w$

To support these queries, maintain for each node v the value $v.\mathsf{totalwt}$, which is the sum of the weights of nodes in the subtree of v. The new query algorithms are nearly identical to Rank and Select, the maintenance of the extra fields is almost identical to maintaining subtree sizes, and the query algorithms have the same running time as Find.

- $\mathsf{MaxWtLess}(k)$: Return the maximum weight among all items less than k

- MaxWtRank(k): Return the maximum weight of all items with rank at most r

To support these two queries, maintain for each node v the value v.maxwt, which is the maximum weight among all nodes in the subtree of v. Similarly, if we store v.minwt, we can support the corresponding minimum-weight queries

- MinWtLess(k): Return the minimum weight among all items less than k
- MinWtRank(k): Return the minimum weight of all items with rank at most r

The key feature of all these queries is that they are computing *efficiently decomposable* functions of *prefixes* of the item sequence. A function $f : 2^X \to Y$ over subsets of the set X is *efficiently decomposable* if, for any two disjoint subsets A and B, we have $f(A \cup B) = f(A) \diamond f(B)$, where $\diamond$ is some function we can compute in $O(1)$ time.