Analysis of Algorithms I: Augmenting Data Structures

Xi Chen

Columbia University

In many situations, we need to design our own data structures for features not supported by the standard ones. It is usually easier to augment existing data structures than designing a new one from scratch. We will use red-black trees to demonstrate ideas useful in augmenting data structures. Red-black tree: Insert, Delete, Search, Min, Max, Predecessor and Successor all in $O(\lg n)$ time in the worst case. Can we augment it to support order statistics:

1. Select(i): return the node with the ith smallest key.
2. Rank(x): given a node x, return the rank of $x.k$ (i.e., return i such that $x.k$ is the ith smallest key in the tree).

also in $O(\lg n)$ worst-case running time?

Idea: For each node x, add a new field/attribute x.size to store

# of internal nodes (excluding the nils) in the subtree of x.

We call it the subtree size of x. They satisfy:

1. If x is a nil leaf, then $x.\text{size} = 0$;
2. If x is an internal node (and thus, has two children), then

$$x.\text{size} = x.\text{left.size} + x.\text{right.size} + 1$$

that is, the size of the left subtree plus the size of the right subtree plus x itself.

First, we show that, given a red-black tree in which each node x has this extra field x.size, then Select can be done efficiently in worst-case $O(\lg n)$ running time.

To find the node with the ith smallest key:

1. Set x to be the root and $h = x.\text{left.size} + 1$ Because of the Binary Search Tree property, all keys in the left subtree are $\leq x.k$ and all keys in the right subtree are $\geq x.k$. Since there are x.left.size many keys in the left subtree, we know that $x.k$ must be the hth smallest key in the tree.
2. $i = h$: simply return x
3. $i > h$: find the the $(i - h)$th smallest key in the right subtree
4. $i < h$: find the ith smallest key in the left subtree

The Rank operation can be done similarly in $O(\lg n)$ time, given a red-black tree in which each node maintains its subtree size. A quick question: Why not just add an extra field x.rank in each node x? This certainly makes the Rank operation much more efficient: $O(1)$, while we can still handle Select in $O(\lg n)$. Recall in Select from the last slide, h is essentially the rank of x.

However, the rank of a node (its key, more exactly) is very fragile. Remember the goal here is to maintain a dynamic set. So between order statistics queries, there might be insertions and deletions. If we insert a new key, it may change the rank of many nodes in the tree. Worst-case: If we insert a new key smaller than all keys currently in the red-black tree, then all the ranks increase by 1, which would cost us $\Omega(n)$ time to update the ranks. But we will see that the cost of updating subtree sizes when inserting (or deleting, see the textbook) a new node is much lower.

In RB-Insert, we start by calling BST-Insert to insert the new node, which we denote by z. By the end of BST-Insert, most of the nodes in the tree have its subtree size remain unchanged. The only nodes for which we need to update the size field are those on the path from z to the root, and we need to increase each of them by 1. So it only takes us $O(\lg n)$ time, because a red-black tree has $O(\lg n)$ depth. By the end, every node x in the tree has the correct size field x.size.

Then RB-Insert continues! The key here is that RB-Insert never changes the tree structure in Case 1 (recoloring only), so the subtree size of each node remains the same. RB-Insert only changes the tree structure in Case 2/3 by rotation (and the total number of rotations performed is ≤ 2). We show next that the damage caused by a rotation is minimal: the size fields can be fixed in $O(1)$ time.

We follow Figure 14.2 where a left-rotation is performed at y in the tree on the right. Check that all the nodes have the same subtree size before and after the rotation, except x and y. We can use the following code to fix the damage caused by a left-rotation:

$$y.\text{size} = x.\text{size}$$
$$x.\text{size} = 1 + x.\text{left.size} + x.\text{right.size}$$

This clearly can be done in $O(1)$ time.

To summarize, here is how we augment RB-Insert to update subtree sizes during the insertion of a new node: After calling BST-Insert, we increment the size field of every node on the path from the root to the new node (and set the size field of the new node to be 1). When we perform a rotation in Case 2/3, we update the size field of the two nodes involved properly. As a result, the running time of the new RB-Insert is still $O(\lg n)$ in the worst case. Same story for RB-Delete because the only structural changes in the tree come from rotations.

Next we give a more involved example, called Interval trees (very useful in practice), of augmenting data structures. Goal: A data structure that maintains a dynamic set of intervals. In addition to Insert and Delete, the data structure also needs to support Interval-Search (to be defined later).

Here an (closed) interval $[t_1, t_2]$ is an ordered pair of real numbers with $t_1 \leq t_2$. (For example, t_1 and t_2 are the starting and ending time of a process.) We will use i to denote an interval. When $i = [t_1, t_2]$, we denote t_1 by i.low and t_2 by i.high.

We need a data structure to support Insert (inserting an interval into the set), Delete (deleting an interval), and Interval-Search:

Given an interval i, find an interval that overlaps with i.

We say two intervals i and i' overlap if

$$i.\text{low} \leq i'.\text{high} \quad \text{and} \quad i.\text{high} \geq i'.\text{low}$$

And they do not overlap if

$$\text{either} \quad i.\text{low} > i'\text{high} \quad \text{or} \quad i.\text{high} < i'.\text{low}$$

We augment red-black trees to support Interval-Search as follows:

1. Each node x contains an interval $x.\text{int}$, and we use the low endpoint of $x.\text{int}$ as the key of x in the red-black tree.
2. In addition to an interval (and of course, a color because it is a red-black tree), each node x also stores the largest high endpoint in the subtree rooted at x, in $x.\,\text{max}$. (This is the place where things become less intuitive but we will see it works out later.)
3. Check that RB-Insert (and RB-Delete as well) can still be augmented to update the max fields while inserting a new node, with worst-case $O(\lg n)$ running time.

Next we show that, given an interval red-black tree in which each node has a max field, Interval-Search can be done in $O(\lg n)$ time.
Interval-Search (i): search for an interval that overlaps with i

1. Set x to be the root
2. If x.int and i overlap
3. return x
4. else if x.left.max $\geq i$.low
5. recursively call Interval-Search (i) in the left subtree
6. else
7. recursively call Interval-Search (i) in the right subtree

To prove the correctness of Interval-Search, we need to show that the algorithm always makes a "correct" decision of going left or right, if x.int and i do not overlap (if they overlap we are done). The tricky thing here is the meaning of a "correct" decision. Here it means when the algorithm decides to go left or right, there must be an interval that overlaps with i in that subtree if there is one such interval in the whole tree. So the algorithm never makes a "wrong" decision by going into a subtree with no interval that overlaps with i while such intervals exist in the other subtree.

We let x denote the root, and let

$$L = \{i' \in \text{left subtree of the root that overlaps with } i\}$$
$$R = \{i' \in \text{right subtree of the root that overlaps with } i\}$$

Lemma

If $x.int$ and i do not overlap and the algorithm decides to go right, then $L = \emptyset$.

Lemma

If $x.int$ and i do not overlap and the algorithm decides to go left, then $L \cup R \neq \emptyset$ implies that $L \neq \emptyset$.

The first lemma is usually what we would expect: If the algorithm decides to go right, it means all intervals that overlap with i (if any) must lie in the right subtree because none of them lies in the left subtree as the first lemma indicates. The second lemma is the tricky part. If the algorithm decides to go left, it does not mean that R is empty. It only means that if there is at least one interval that overlaps with i in the tree, then some of them must lie in the left subtree so we may just narrow the search down to the left subtree. So in either cases, the algorithm moves to a subtree with at least one interval that overlaps with i, if such an interval exists in the tree. The correctness follows from these two lemmas.

Proof of the first lemma is straight-forward. Let $y = x.\text{left}$. Because $y.\text{max} < i.\text{low}$ and because $y.\text{max}$ is the largest high endpoint among all intervals in the left subtree, we have

$$i'.\text{high} \leq y.\text{max} < i.\text{low}, \quad \text{for any interval } i' \text{ in the left subtree}$$

Therefore, none of the intervals in the left subtree overlaps with i.

For the second lemma, we show that $L = \emptyset$ implies $R = \emptyset$. Assume $L = \emptyset$. By the definition of $y.\max$, there is an interval i' in the left subtree with $i'.\text{high} = y.\max \geq i.\text{low}$. But $L = \emptyset$ implies that i' does not overlap with i. This means that i' must satisfy

$$i'.\text{low} > i.\text{high}$$

On the other hand, because we use the low endpoint as the key of the tree. Every interval i'' in the right subtree satisfies

$$i''.\text{low} \geq i'.\text{low} > i.\text{high}$$

and thus, none of the intervals in the right subtree overlap with i.