7 Augmenting Red-Black Trees

7.1 Introduction
Suppose that you are asked to implement an ADT that is the same as a dictionary but has one additional operation:

\[ \text{operation: SIZE(Set } S) \text{: Returns the current size of the set} \]

If we try to implement this procedure without additional data in the data structure, the worst case running time would be \( \Theta(n) \) or worse.

But, if we add a size variable and increment it when we insert and decrement it when we delete, then the running time would be \( \Theta(1) \).

This is an example of augmenting a data structure.

7.2 Method
In this section we will look at three examples of augmenting red-black trees to support new queries. Any data structure can be augmented to provide additional functionality beyond the original ADT.

A red-black tree by itself is not very useful. All you can do is search the tree for a node with a certain key value. To support more useful queries we need to have more structure. When augmenting data structures, the following four steps are useful:

1. Pick a data structure to start with.
2. Determine additional information that needs to be maintained.
3. Check that the additional information can be maintained during each of the original operations (and at what additional cost, if any).
4. Implement the new operations.

7.3 Example 1
Let’s say we want to support the query \( \text{MIN}(R) \), which returns the node with minimum key-value in red-black tree \( R \).
One solution is to traverse the tree starting at the root and going left until there is no left-child. This node must have the minimum key-value. Since we might be traversing the height of the tree, this operation takes $O(\log n)$ time.

Alternatively, we can store a pointer to the minimum node as part of the data structure (at the root, for instance). Then, to do a query $\text{MIN}(R)$, all we have to do is return the pointer $(R.min)$, which takes time $O(1)$. The problem is that we might have to update the pointer whenever we perform $\text{INSERT}$ or $\text{DELETE}$.

- **INSERT($R$, $x$):** Insert $x$ as before, but if $key(x) < key(R.min)$, then update $R.min$ to point at $x$. This adds $O(1)$ to the running time, so its complexity remains $O(\log n)$.

- **DELETE($R$, $x$):** Delete $x$ as before, but if $x = R.min$, then update $R.min$: if $x$ was the minimum node, then it had no left child. Since it is a red-black tree, its right-child, if it has one, is red (otherwise property 3 would be violated). This right-child is $\text{succ}(x)$ and becomes the new minimum if it exists. If $x$ had no children, then the new minimum is the parent of $x$. Again we add $O(1)$ to the running time of $\text{DELETE}$ so it still takes $O(\log n)$ in total.

This is the best-case scenario. We support a new query in $O(1)$-time without sacrificing the running times of the other operations.

### 7.4 Example 2

Now we want to know not just the minimum node in the tree, but, for any $x$ in the tree, the minimum node in the subtree rooted at $x$. We’ll call this query $\text{SUBMIN}(R,x)$.

To achieve this query in time $O(1)$, we’ll store at each $x$ a pointer $x.min$, to the minimum node in its subtree. Again, we’ll have to modify $\text{INSERT}$ and $\text{DELETE}$ to maintain this information.

- **INSERT($R$, $x$):** Insert $x$ as before, but, for each $y$ that is an ancestor of $x$, if $key(x) < key(y.min)$, then update $y.min$ to point at $x$. This adds $O(\log n)$ to the running time, so its complexity remains $O(\log n)$.

- **DELETE($R$, $x$):** Delete $x$ as before, but, for each $y$ that is an ancestor of $x$, if $x = y.min$, then update $y.min$: if $x$ was the minimum node in a subtree, then it had no left child. Since it is a red-black tree, its right-child, if it has one, is red (otherwise property 3 would be violated). This right-child is $\text{succ}(x)$ and becomes the new minimum if it exists. If $x$ had no children, then the new minimum is the parent of $x$. Again we add $O(\log n)$ to the running time of $\text{DELETE}$ so it still takes $O(\log n)$ in total.

- **Fix-up:** The rotations (but not the recolorings) in the fix-up processes for $\text{INSERT}$ and $\text{DELETE}$ might affect the submins of certain nodes. Consider $\text{RotateRight}(T, y)$ where $x$ is the left child of $y$. The submin of $x$ will be in the subtree $A$ (or will be $x$ itself if $A$ is empty). This doesn’t change after the rotation. The submin of $y$, however, which used to be the same as $x$’s submin, is now in $B$ (or $y$ itself if $B$ is empty). So, we set $\text{SUBMIN}(R,y)$ to $y$ if $B$ is empty, or to $\text{SUBMIN}(R,z)$, where $z$ is the root of $B$. It takes just constant time to reset $\text{SUBMIN}(R,y)$, so rotations still take $O(1)$. The modification for a left rotation are symmetric.
7.5 Example 3

We want to support the following queries:

- $\text{RANK}(R,k)$: Given a key $k$, what is its ”rank”, i.e., its position among the elements in the red-black tree?
- $\text{SELECT}(R, r)$: Given a rank $r$, what is the key with that rank?

Example:
If $R$ contains the key-values 3,15,27,30,56, then $\text{RANK}(R,15) = 2$ and $\text{SELECT}(R,4) = 30$.

Here are three possibilities for implementation:

1. Use red-black trees without modification:

   - Queries: Simply carry out an inorder traversal of the tree, keeping track of the number of nodes visited, until the desired rank or key is reached. This requires time $\Theta(n)$ in the worst case.
   - Updates: No additional information needs to be maintained.
   - Problem: Query time is very long and this method does not take advantage of the structure of the Red-Black tree. We want to be able to carry out both types of queries in only $\Theta(\log n)$ time.

2. Augment red-black trees so that each node $x$ has an additional field $\text{rank}(x)$ that stores its rank in the tree.

   - Queries: Similar to $\text{SEARCH}$, choosing path according to $\text{key}$ or $\text{rank}$ field (depending on the type of query). This requires time $\Theta(\log n)$, just like $\text{SEARCH}$.
   - Updates: Carry out normal update procedure, then update the $\text{rank}$ field of all affected nodes. This can take time $\Theta(n)$ in the worst case, since any insertion or deletion affects the rank of every node with higher key-value.
   - Problem: We’ve achieved the $\Theta(\log n)$ query time we wanted, but at the expense of the update time, which has gone from $\Theta(\log n)$ to $\Theta(n)$. We would like all operations to have time at worst $\Theta(\log n)$.

3. Augment red-black trees so that each node has an additional field $\text{size}(x)$ that stores the number of nodes in the subtree rooted at $x$ (including $x$ itself).
Queries: We know that

\[ \text{rank}(x) = 1 + \text{number of nodes that come before } x \text{ in the tree}. \]

**RANK(R,k):** Given key \( k \), perform SEARCH on \( k \) keeping track of "current rank" \( r \) (which starts out as 0): when going left, \( r \) remains unchanged; when going right let \( r := r + \text{size}(\text{left}(x)) + 1 \). When \( x \) found such that \( \text{key}(x) = k \), output \( r + \text{size}(\text{left}(x)) + 1 \). Note that we did not deal with degenerate cases (such as when \( k \) does not belong to the tree), but it is easy to modify the algorithm to treat those cases.

**SELECT(R,r):** Given rank \( r \), start at \( x = R \) and work down, looking for a node \( x \) such that \( r = \text{size}(\text{left}(x)) + 1 \) (return that node once it is found). If \( r < \text{size}(\text{left}(x)) + 1 \), then we know the node we are looking for is in the left subtree, so we go left without changing \( r \). If \( r > \text{size}(\text{left}(x)) + 1 \), then we know the node we are looking for is in the right subtree, and that its relative rank in that tree is equal to \( r - (\text{size}(\text{left}(x)) + 1) \), so we change \( r \) accordingly and go right. Once again, we did not deal with degenerate cases (such as when \( r \) is a rank that does not correspond to any node in the tree), but they are easily accommodated with small changes to the algorithm.

Query time: \( \Theta(\log n) \), as desired, since both algorithms are essentially like SEARCH (tracing a single path down from the root).

Updates: **INSERT** and **DELETE** operations consist of two phases for red-black trees: the operation itself, followed by the fix-up process. We look at the operation phase first, and deal with the fix-up process afterwards.

**INSERT(R,x):** We can set \( \text{size}(x) := 1 \), and simply increment the \( \text{size} \) field for every ancestor of \( x \).

**DELETE(R,x):** Consider the node \( y \) that is actually removed by the operation (so \( y = x \) or \( y = \text{succ}(x) \)). We know the size of the subtree rooted at every node on the path from \( y \) to the root decreases by 1, so we simply traverse that path to decrement the size of each node.

We’ve shown how to modify the **INSERT** and **DELETE** operations themselves. If we show how to do rotations and keep the \( \text{size} \) fields correct, then we’ll know how to do the whole fix-up process, since each case just consists of a rotation and/or a recoloring (recoloring does not affect the \( \text{size} \) field of any node).

**Rotations:** Consider right rotations (left rotations are similar).

\[
\begin{align*}
\text{size}(y) &= \text{size}(A)+\text{size}(B)+\text{size}(C)+2 \\
\text{size}(x) &= \text{size}(A)+\text{size}(B)+1
\end{align*}
\]
The only size fields that change are those of nodes $x$ and $y$, and the change is easily computed from the information available. So each rotation can be performed while maintaining the size information with only a constant amount of extra work.

- Update time: We have only added a constant amount of extra work during the first phase of each operation, and during each rotation, so the total time is still $\Theta(\log n)$.

Now, we have finally achieved what we wanted: each operation (old or new) takes time $\Theta(\log n)$ in the worst-case.