1. Red-Black Trees

- Last time we discussed red-black trees:
  - Balanced binary trees—all elements in left (resp., right) subtree of node $x$ are $< x$ (resp., $> x$).
  
  $$
  \begin{array}{c}
  \circ \\
  <x \\
  \end{array}
  \begin{array}{c}
  \circ \\
  >x \\
  \end{array}
  $$

  - Every node is colored RED or BLACK and we maintained red-black invariant:

    * Root is BLACK.
    * A RED node can only have BLACK children.
    * Every path from the root to a leaf contains the same number of BLACK nodes.

- We saw how the red-black invariant guaranteed $O(\log n)$ height.
- We could reestablish the red-black invariant after an insertion or deletion in $O(\log n)$ time
  - $O(\log n)$ node recolorings (no structural changes).
  - At most three rotations:

    $$
    \begin{array}{c}
    \circ \\
    A \\
    \end{array}
    \begin{array}{c}
    \circ \\
    B \\
    \end{array}
    \begin{array}{c}
    \circ \\
    C \\
    \end{array}
    \leftarrow
    \begin{array}{c}
    \circ \\
    A \\
    \end{array}
    \begin{array}{c}
    \circ \\
    B \\
    \end{array}
    \begin{array}{c}
    \circ \\
    C \\
    \end{array}
    $$

- Red-black tree also supports SEARCH, SUCCESSOR, and PREDECESSOR in $O(\log n)$ as in binary search trees.
- We will now discuss how to develop data structures supporting other operations by augmenting red-black tree.
2 Augmented Data Structures

- We want to add an operation \( \text{SELECT}(i) \) to a red-black tree
  - We have previously seen how to select the \( i \)th element among \( n \) elements in \( O(n) \) time.
  - Can we support it faster if we have the elements stored in a data structure?
  - We can of course support the operation in \( O(1) \) time if we have the elements sorted in an array but what is we also want to be able to insert and delete elements?
- We augment every node \( x \) in red-black tree with a field \( \text{size}(x) \) equal to the number of nodes in the subtree rooted in \( x \)
  - \( \text{size}(x) = \text{size}(\text{left}(x)) + \text{size}(\text{right}(x)) + 1 \)

Example:

```
SELECT(x, i)
  r = \text{size}(\text{left}(x)) + 1
  \text{IF } i = r \text{ THEN Return } x
  \text{IF } i < r \text{ THEN Return SELECT(\text{left}(x), i)}
  \text{IF } i > r \text{ THEN Return SELECT(\text{right}(x), i-r)}
```

Example (\( \text{SELECT}(17) \)):

\( \implies \) Since we only follow one root-leaf path, the operation takes \( O(\log n) \) time.

- Actually, we can also use the field to perform the inverse operation in \( O(\log n) \) time: determining the \textit{rank} of the element in node \( x \):
\textbf{RANK}(x)
\begin{align*}
  r &= \text{size}(\text{left}(x)) + 1 \\
  y &= x \\
  \text{WHILE } y \neq \text{root of tree DO} \\
  \text{IF } y = \text{right}(\text{parent}(y)) \text{ THEN} \\
  & \quad r = r + \text{size}(\text{left}(\text{parent}(y))) + 1 \\
  & \quad y = \text{parent}(y) \\
  \text{FI} \\
  \text{OD} \\
  \text{Return } r
\end{align*}

- We need to maintain the extra field during updates:
  - \text{INSERT}(i):
    * Search down one root-leaf part as usual for position where \( i \) should be inserted.
    * Increment \( \text{size}(x) \) for all nodes \( x \) on root-leaf path (note that these are the only nodes for which the size field change).

Example (Insertion of element 32)

\begin{itemize}
  \item Rebalancing using Red-black tree rules. Recall that we do \( O(\log n) \) recolorings and \( O(1) \) rotations:
    \begin{itemize}
      \item Color change rules do not affect extra field
      \item Rotations do affect size extra fields but we can still easily perform a rotation in \( O(1) \) time
    \end{itemize}
\end{itemize}

\[
\begin{align*}
  \text{size}(y') &= \text{size}(\text{root}(B)) + \text{size}(\text{root}(C)) + 1 \\
  \text{size}(x') &= \text{size}(y) = \text{size}(\text{root}(A)) + \text{size}(\text{root}(B)) + \text{size}(\text{root}(C)) + 2
\end{align*}
\]

\[\Rightarrow \text{ INSERT performed in } O(\log n) \text{ time.}\n\]
- \text{DELETE}(i):

3
* Find element to delete and decrement size field on one root-leaf path.
* Rebalance using rotations

\[ \implies \text{DELETE performed in } O(\log n) \text{ time.} \]

- Note: The key to maintaining the size field during updates is that the field of node \( x \) only depend on the field of the children of \( x \) \( \implies \)
  - Insertion or deletion only affect one root-leaf path.
  - Rotations can be handled in \( O(1) \) time locally.

- In general we can easily prove the following theorem about augmenting a red-black tree, which appears as CLRS Theorem 14.1:

\[ \text{A field } f \text{ in a red-black tree can be maintained in } O(\log n) \text{ time during updates if } f(x) \text{ can be computed using only the information in nodes } x, \left\{ \text{left}(x), \text{right}(x) \right\}, \text{ including } f(\text{left}(x)) \text{ and } f(\text{right}(x)). \]

- When changing field in a node \( x \), \( f \) can only change for the \( O(\log n) \) ancestors of \( x \) on the path to the root.
- Rotations can be handled in \( O(1) \) time locally.

3 Interval Tree

- We now consider a slightly more complicated augmentation. We want to solve the following problem:

  - Maintain a set of \( n \) intervals \([i_1, i_2]\) such that, given a query point \( q \), \textit{at least one} of the intervals (if any) containing \( q \) can be found efficiently. (Note: this particular query only requires that one interval be found.)

Example: A set of intervals. A query with \( q = 9 \) returns \([6, 10]\) or \([8, 9]\).

![Diagram of intervals]

- To solve the problem we use the so-called “Interval tree”:

  - Red-black tree with intervals in nodes
    * The left endpoint of the interval is used as the key value in the search tree.
    - Node \( x \) is augmented with the maximal right endpoint in the subtree rooted at \( x \)

Example: Interval tree on intervals from previous figure:
- We can maintain the interval tree dynamically during insertions and deletions in $O(\log n)$ time
  - because augmented field in $x$ only depends on augmented fields in the children of $x$ (and the interval stored in $x$).
  - $\max(x) = \max(\text{right endpoint}(x), \max(\text{left}(x)), \max(\text{right}(x)))$

- We can also answer a query in $O(\log n)$ time:
  - We first check if $q$ is contained in interval stored in root $r$. If it is we are done.
  - Next we check if $q$ is on left side of left endpoint of interval in $r$. If it is we recursively search in left subtree ($q$ cannot be contained in any interval in right subtree).
  - If $q$ is to the right of left endpoint of interval in $r$ we have two cases:
    * If $\max(\text{left}(r)) > q$ there must be a segment in left subtree containing $q$ and we recurse left.
    * If $\max(\text{left}(r)) < q$ there is no segment in left subtree containing $q$ and we recurse right.

```
QUERY(x, q)
  IF q contained in x interval THEN Return x
  IF max(\text{left}(x)) \geq q THEN
    Return Query(\text{left}(x), q)
  ELSE
    Return Query(\text{right}(x), q)
  FI
```

$\implies$ We search down one root-leaf path $\implies O(\log n)$ time.

- Example: Search for $q = 23$. Follow left path from root.

- What happens if we want to find all the intervals containing the query point $q$?
  - Example: Search for $q = 19$.
  - Worse-case: First modify all intervals stored above the leaves in the red-black tree so that the intervals extend to 100. (For ex, [0, 100], [6, 100], [15, 100], [17, 100], [19, 100], [26, 100].) Now search for $q = 99$. The search goes to each of these $\Theta(\log n)$ nodes.

$\implies$ If there are $k$ intervals output, the query time is $\Theta((k + 1) \log n)$. NOT OPTIMAL!

- We want $O(k + \log n)$ time.

- Solution: Interval trees, as developed by Edelsbrunner and McCreight. See notes from DeBerg, Section 14.1.