Augmented Search Trees
(CLRS 14)

1 Red-Black Trees

• Last time we discussed red-black trees:
  – Balanced binary trees—all elements in left (right) subtree of node $x$ are $< x$ ($> x$).
    
    ```
    +---+
    | x |
    +---+
    |   |
    +---+---+
    | < e | > e |
    +---+---+
    +---+
    +---+
    ```
  
  – Every node is colored RED or BLACK and we maintained red-blue 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-blue invariant guaranteed $O(\log n)$ height.

• We could reestablish the red-blue invariant after an insertion or deletion in $O(\log n)$ time
  – $O(\log n)$ node recolorings (no structural changes).
  – $O(1)$ rotations:

  ```
  AxByC
  +---+
  | A | B |
  +---+
  | C |
  +---+
  AxByC
  ```

• 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 `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 $size(x)$ equal to the number of nodes in the subtree rooted in $x$
  - $size(x) = size(left(x)) + size(right(x)) + 1$

Example:

- We can use this field to implement `Select(i)`:

  ```
  Select(x, i)
  r = size(left(x)) + 1
  IF $i = r$ THEN Return $x$
  IF $i < r$ THEN Return Select(left(x), $i$)
  IF $i > r$ THEN Return Select(right(x), $i-r$)
  ```

Example (Select(17)):
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 “opposite” operation in $O(\log n)$ time—determining the rank of the element in node $x$:

\[
\text{Rank}(x) \quad r = \text{size(left}(x)) + 1 \\
y = x \\
\text{WHILE } y \neq \text{root of tree DO} \\
\quad \text{IF } y = \text{right(parent}(y)) \text{ THEN} \\
\quad \quad r = r + \text{size(left(parent}(y)))+1 \\
\quad \text{FI} \\
\quad y = \text{parent}(y) \\
\text{OD} \\
\text{Return } r
\]

Example (Rank of element 38):

- We need to maintain the extra field during updates:
  - 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 (these are the only nodes for which the size field change).

Example (Insertion of element 32)

\[
\begin{array}{c}
\text{14} \\
\text{2} \phantom{1} \\
\text{17} \\
\text{12} \\
\text{15} \\
\text{21} \\
\text{19} \\
\text{20} \\
\text{14} \\
\text{12} \\
\text{26} \\
\text{21} \\
\text{41} \\
\text{8} \\
\text{47} \\
\text{1} \\
\text{32} \\
\text{1} \\
\text{3} \\
\text{1} \\
\text{7} \\
\text{2} \\
\text{12} \\
\text{1} \\
\text{14} \\
\text{1} \\
\text{20} \\
\text{1} \\
\text{35} \\
\text{2} \\
\text{39} \\
\end{array}
\]

* Rebalancing using Red-black tree rules—recall that we do \( O(\log n) \) recolorings and \( O(1) \) rotations:
  * Color change rules do not affect extra field
  * Rotations do affect size extra fields but we can still easily perform a rotation in \( O(1) \) time

\[
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\end{array}
\quad \rightarrow 
\quad
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\end{array}
\]

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

\[\downarrow\]
INSERT performed in \( O(\log n) \) time.

- **DELETE**(\( i \)):
  * Find element to delete and decrement size field on one root-leaf path (recall that conceptually we always delete a node with at most one child).
  * Rebalance using rotations.

\[\downarrow\]
DELETE performed in \( O(\log n) \) 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 \( x \) ⇒
  - 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:

A field $f$ in a red-black tree can be maintained in $O(\log n)$ time during updates if $f(x)$ can be computed using only information in $x$, $\text{left}(x)$ and $\text{right}(x)$ (including $f(\text{left}(x))$ 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 so solve the following problem:

  - Maintain a set of $n$ intervals $[i_1, i_2]$ such that one of the intervals containing a query point $q$ (if any) can be found efficiently.


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

  - Red-black tree with intervals in nodes
    * Key is left endpoint
  - Node $x$ augmented with maximal right endpoint in subtree rooted in $x$

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:

  1. We first check if $q$ is contained in interval stored in root $r$—if it is we are done.
  2. 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).
  3. If $q$ is to the right of left endpoint of interval in $r$ we have two cases:
     
     (a) If $\max(\text{left}(r)) > q$ there must be a segment in left subtree containing $q$ and we recurse left.
     
     (b) If $\max(\text{left}(r)) < q$ there is no segment in left subtree containing $q$ and we recurse right.

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

Example: Query with $q = 23$: