Deletion Transformation

- Transform the problem of Deletion to the problem of deletion of a node with two Null Pointers.

2 Null Children

1 Null Child

Right:

Left:

Delete
Deletion Transformation

0 Null Child

Left:

Right:
Deletion(Top Down)

- Let's consider the problem of deleting a node with null children.
- We do not know if the node to be deleted is red or black, because we start from the root of the tree.

Idea: Make sure it will be a red node by making some color changes, rotations and double rotations on the top down traversal.
Deletion (Top Down)

- **FIRST STEP:**
  Transform the problem so that if we are at node X, then both X and its sibling T are black and their parent "P" is RED.
  OR we are done.

- **Case 1:** If there is only one or two nodes then the solution is obvious.

- **Case 2:** Both children of the root are black nodes.
Deletion (Top Down)

Case 3: At least one child of the root is red and the next node is black node.

Case 4: At least one child of the root is red and the next node is red node.

next subproblem

* if both children of node "a" are present.
But if it is a leaf node then we are done
Deletion (Top Down)

SECOND STEP:

Case 1: Both Children of X are black

Case 1.1: Both Children of T are black

Case 1.2: The right child of T is red RR (LL case is similar)
Deletion (Top Down)

Case 1.3: The left child of T is red RL (LR case is similar).

Case 2: At least one child of node X is red

Case 2.1: If we advance to the red node child of X
Deletion (Top Down)

Case 2.2: We would advance to the black child of X LL (or RR)

Case 2.3: We would advance to the black child of X LR (or RL)
Notes

NOTE THAT

"Next subproblem" means

BUT

"Next subproblem*" means

TRADE-OFFS

Top-down delete: Easier algorithm +
Good for Parallel Operations(O(log n)rot)

Bottom-up delete: Fewer color changes and
Constant # of Rotations
Delete 7

Before Delete

After Delete
Delete 11

Before delete:

12
   /   
  8     14
 /   \
2     10
|     /
1     3
|     | 
4     9
|     | 
5     11

After delete:

12
   /   
  4     14
 /   \
2     13
|     /
1     10
|     | 
3     6
|     | 
5     11

Additional code:

11
   /   
  8     14
 /   \
2     10
|     /
1     3
|     | 
4     9
|     | 
5     11

X
Delete 1

Before deletion:

```
          12
         /  
        4    14
       /    /  
      2    13   16
     /   /  
    1   3   6  10
   /   /   /  
  5   9  15
```

After deletion:

```
          12
         /  
        4    14
       /    /  
      2    13   16
     /   /  
    1   3   6  10
   /   /   /  
  5   9  15
```

```
          12
         /  
        4    14
       /    /  
      2    13   16
     /   /  
    1   3   6  10
   /   /   /  
  5   9  15
```

```
Note that at some point one needs to recolor the root from red to black.
Delete 1
Red Black Trees

- Balanced Binary Search Trees, 80’s
  - Henk Olivie, PhD thesis, Belgium
  - Robert E. Tarjan, Current form
  - Sedgwick (top-down)
  - Best ones deletion have at most a constant number of rotations,
  - Height is at most $2 \log n$
Other Balanced Trees

- AVL trees, 60’s (first balanced trees)
  - Adelson-Velskii and Landis (Russian)
  - Every node has $| h_l - h_r | \leq 1$, where $h_l$ is height of left subtree and $h_r$ is height of right subtree.
  - Height of tree is at most $1.7 \log n$
  - Deletion need $\Omega(n)$ rotations
  - Procedures more complex
Other Balancing Techniques

- Weighted balanced trees, 70’s
  - Every node has \( \frac{\min\{n_l, n_r\}}{\max\{n_l, n_r\}} \leq 1/2 \), where \( n_l \) is number of nodes in the left subtree and \( n_r \) is number of nodes in the right subtree.

- 2-3 trees: All leaves at the same level and every node has two or three children.

- B-trees: All leaves at the same level and every node has a huge number of children. This type of trees are good when storing large number of elements in a tree and storing nodes in disks.
Incorrect Balancing Techniques

- Number balanced trees
  - For every node $|n_l - n_r| \leq 1$.
  - Does not support insert and delete in $O(\log n)$ time.
**Concatenate two Red-Black trees**

- Given RB-trees $L_1$ and $L_2$ such that all elements in $L_1$ are smaller than all elements in $L_2$, concatenate all values into one RB-tree in $O(\log n)$ time, ($n$ is total number of elements in the trees).

- Assume $\text{Rank}(L_1) \geq \text{Rank}(L_2)$.

- Delete the smallest element in $L_2$. Call the node where it is stored node $j$ and the resulting RB-tree is $L'_2$.

- Clearly, $\text{Rank}(L_1) \geq \text{Rank}(L'_2)$.
Initial Step

- Let $r = \text{Rank}(L'_{2})$.
- Traverse the rightmost path of $L_{1}$ until you find the first node with Rank $r$.
- Insert the junction node $j$ and $L'_{2}$ as shown below.

Time is proportional to difference in ranks +1

$O(\log n)$
• We now need to adjust the coloring. Promotion means color changes.
Additional Operations

- $\beta$ has to be black because its right child has rank $r$ and $\beta$ has rank $r+1$.
  - Case 1: $\beta_l$ is Red then make $j$ Red and do a color change.

- Since $\beta$ is now red you need to reach above for problems. If the parent of $\beta$ is black then we are done. If it is the root, make it black. Otherwise we are in a case similar to Insert.
Case 2 and Subcase 2.1

- **Case 2:** $\beta_i$ is Black. (Note that the root of $L'_2$ must be black because it is an RB-tree. However Subcases 2.2 and 2.3 consider the case when the root of $L'_2$ is Red. We do this because we want the code to also work for the concatenation operation that arise when performing the operation Split that we define later one. For those concatenation operations the color of the root of $L'_2$ could be Red.)
Subcase 2.1

- **Subcase 2.1:** $\alpha$ is black and root of $L'_2$ is black. Make $j$ red and we are done.
### Subcase 2.2

- **Subcase 2.2:** $\alpha$ is red and root of $L'_2$ is red. Make $j$ red and then rotate.

- Need to continue searching above $j$ because root of subtree is now red
Subcase 2.3

- **Subcase 2.3**: $\alpha$ is black and root of $L'_2$ is red. Make $j$ red, perform a rotation and we are done.
Subcase 2.4

- Subcase 2.4: $\alpha$ is red and root of $L'_2$ is black. Make $j$ red, perform a double rotation and we are done.
Split operation

Given a red-Black tree and key $y$ construct 2 Red-Black trees. One with keys $\leq y$ and the other with keys $> y$. You may destroy the old tree. We now show how to implement this in $O(\log n)$ time, where $n$ is the number of elements in the tree.

Step 1: First find node $y$.

In the following example $y$ is greater than the value at $p_1, p_4, p_6,$ and $p_7$. But less than $p_2, p_3, p_5$ and $p_8$.

Perform all the concatenations shown below. Specifically, Let $T_1 = \text{Concatenate} (\beta_2, p_2, \beta_3)$, $T_2 = \text{Concatenate} (\beta_5, p_3, T_1)$,
and $T_3 = \text{Concatenate} (\beta_8, p_5, T_2)$. $T_4 = \text{Concatenate} (\beta, p_8, T_3)$. Tree $T_4$ is the RB-tree with all the elements $> y$. Then insert let $S_1$ be the tree with $y$ inserted to $\alpha$. Let $S_2 = \text{Concatenate} (\alpha_7, p_7, S_1)$, $S_3 = \text{Concatenate} (\alpha_6, p_6, S_2)$, $S_4 = \text{Concatenate} (\alpha_4, p_4, S_3)$, $S_5 = \text{Concatenate} (\alpha_1, p_1, S_4)$, Tree $S_5$ is the RB-tree with all the elements $\leq y$.

The total time complexity is $O(\log n)$ because the time complexity for the concatenation for two trees and a junction node takes time proportional to the difference in ranks of the two trees.
Concatenate with junction node

Total time for Split = \( \sigma \) difference in ranks
Other operations

- min: find smallest $O(\log n)$
- max: find largest $O(\log n)$
- $k_{th}$ smallest: $O(\log n)$

\[
\begin{align*}
  k &= \alpha L + 1 \\
  k &\leq \alpha L \\
  k &> q\alpha L + 1
\end{align*}
\]

Search for $k_{th}$ smallest

- Mutiset: All the above can be made to work with multisets within the same time bounds.