AVL Trees (contd.)

Insertion

- If an insertion causes $T$ to become unbalanced, we travel up the tree from the newly created node until we find the first node $x$ such that its grandparent $z$ is unbalanced node.
- Since $z$ became unbalanced by an insertion in the subtree rooted at its child $y$,
  \[ \text{height}(y) = \text{height}(\text{siblings}(y)) + 2 \]
- Now to rebalance...

Restructure Algorithm

\textbf{Algorithm \textit{restructure}(x, y, z):}

\begin{itemize}
  \item \textbf{Input:} A node $x$ of a binary search tree $T$ that has both a parent $y$ and a grandparent $z$
  \item \textbf{Output:} Tree $T$ restructured by a rotation (either single or double) involving nodes $x$, $y$, and $z$.
\end{itemize}

Let $(a, b, c)$ be an in-order listing of the nodes $x$, $y$, and $z$.

Let $(T_0, T_1, T_2, T_3)$ be an in-order listing of the four sub-trees of $x$, $y$, and $z$.

Replace the sub-tree rooted at $x$ with a new sub-tree rooted at $b$.

Make $a$ the left child of $b$ and $T_0, T_1$ be the left and right sub-trees of $a$.

Make $c$ the right child of $b$ and $T_2, T_3$ be the left and right sub-trees of $c$. 
Restructuring (as Single Rotations)
• Single Rotations:

Restructuring (as Double Rotations)
• double rotations:

Restructure Algorithm (continued)
• Now create an Array of 8 elements. At rank 0 place the parent of z.

Cut() the 4 T trees and place them in their in-order rank in the array
Restructure Algorithm (continued)

• Now cut x, y, and z in that order (child, parent, grandparent) and place them in their in-order rank in the array.

```
1 2 3 4 5 6 7
```

• Now we can re-link these sub-trees to the main tree.

• Link in rank 4 (b) where the sub-tree’s root formerly

```
<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
```

Restructure Algorithm (continued)

• Link in ranks 2 (a) and 6 (c) as 4’s children.

```
<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
```

Restructure Algorithm (continued)

• Finally, link in ranks 1, 3, 5, and 7 as the children of 2 and 6.

```
<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
```

• Now you have a balanced tree!
Restructure Algorithm (continued)

• NOTE:
  – This algorithm for restructuring has the exact same effect as using the four rotation cases discussed earlier.
  – Advantages: no case analysis, more elegant

Removal

• We can easily see that performing a `removeAboveExternal(w)` can cause T to become unbalanced.
• Let z be the first unbalanced node encountered while traveling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.
• We can perform operation `restructure(x)` to restore balance at the sub-tree rooted at z.

Removal in an AVL Tree

• Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
• Example:
Rebalancing after a Removal

- Let $z$ be the first unbalanced node encountered while travelling up the tree from $w$. Also, let $y$ be the child of $z$ with the larger height, and let $x$ be the child of $y$ with the larger height.
- We perform $\text{restructure}(x)$ to restore balance at $z$.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of $T$ is reached.

Removal (contd.)

- NOTE: restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of $T$ is reached.

Running Times for AVL Trees

- a single restructure is $O(1)$
  - using a linked-structure binary tree
- find is $O(\log n)$
  - height of tree is $O(\log n)$, no restructures needed
- insert is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
  - One restructuring is sufficient to restore the global height balance property
- remove is $O(\log n)$
  - initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
  - Single re-structuring is not enough to restore height balance globally. Continue walking up the tree for unbalanced nodes.
Multi-Way Search Tree

- A multi-way search tree is an ordered tree such that
  - Each internal node has at least two children and stores \( d - 1 \) key-element items \((k_i, o_i)\), where \( d \) is the number of children
  - For a node with children \( v_1, v_2, \ldots, v_d \) storing keys \( k_1, k_2, \ldots, k_d \):
    - keys in the subtree of \( v_1 \) are less than \( k_1 \)
    - keys in the subtree of \( v_i \) are between \( k_{i-1} \) and \( k_i \) (\( i = 2, \ldots, d - 1 \))
    - keys in the subtree of \( v_d \) are greater than \( k_{d-1} \)
  - The leaves store no items and serve as placeholders

Multi-Way Inorder Traversal

- We can extend the notion of inorder traversal from binary trees to multi-way search trees
- Namely, we visit item \((k, o)\) of node \( v \) between the recursive traversals of the subtrees of \( v \) rooted at children \( v_1 \) and \( v_2, \ldots, v_d \)
- An inorder traversal of a multi-way search tree visits the keys in increasing order
Multi-Way Searching

- Similar to search in a binary search tree
- A each internal node with children \( v_1, v_2, \ldots, v_d \) and keys \( k_1, k_2, \ldots, k_d \):
  - \( k = k_i (i = 1, \ldots, d - 1) \) the search terminates successfully
  - \( k < k_1 \) we continue the search in child \( v_1 \)
  - \( k_i < k < k_{i+1} (i = 2, \ldots, d - 1) \) we continue the search in child \( v_i \)
  - \( k > k_d \) we continue the search in child \( v_d \)
- Reaching an external node terminates the search unsuccessfully
- Example: search for 30

(2,4) Tree

- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node

Height of a (2,4) Tree

- Theorem: A (2,4) tree storing \( n \) items has height \( O(\log n) \)
  - Proof:
    - Let \( h \) be the height of a (2,4) tree with \( n \) items
    - Since there are at least 2^i items at depth \( i = 0, \ldots, h - 1 \) and no items at depth \( h \), we have
      \[ n \geq 2 \times 2^1 + 4 \times 2^2 + \ldots + 2^{h-1} = 2^h - 1 \]
    - Thus, \( h \leq \log (n + 1) \)
- Searching in a (2,4) tree with \( n \) items takes \( O(\log n) \) time
Insertion
- We insert a new item \((k, o)\) at the parent \(v\) of the leaf reached by searching for \(k\)
  - We preserve the depth property but
  - We may cause an overflow (i.e., node \(v\) may become a 5-node)
- Example: inserting key 30 causes an overflow

\[
\begin{array}{c}
27 & 30 & 32 & 35 \\
10 & 15 & 24 \\
2 & 8 & 12 & 18
\end{array}
\]

Overflow and Split
- We handle an overflow at a 5-node \(v\) with a split operation:
  - Let \(v_1 \ldots v_5\) be the children of \(v\) and \(k_1 \ldots k_4\) be the keys of \(v\)
  - Node \(v\) is replaced nodes \(v'\) and \(v''\)
    - \(v'\) is a 3-node with keys \(k_1\) and children \(v_1 v_2 v_3\)
    - \(v''\) is a 2-node with key \(k_4\) and children \(v_4 v_5\)
    - Key \(k_3\) is inserted into the parent \(u\) of \(v\) (a new root may be created)
- The overflow may propagate to the parent node \(u\)

Analysis of Insertion

\[\text{Algorithm insertItem}(k, o)\]
1. We search for key \(k\) to locate the insertion node \(v\)
2. We add the new item \((k, o)\) at node \(v\)
3. While overflow \((v)\)
   If isRoot \((v)\)
   Create a new empty root above \(v\)
   \(v \leftarrow \text{split}(v)\)
- Let \(T\) be a (2,4) tree with \(n\) items
  - Tree \(T\) has \(O(\log n)\) height
  - Step 1 takes \(O(\log n)\) time because we visit \(O(\log n)\) nodes
  - Step 2 takes \(O(1)\) time
  - Step 3 takes \(O(\log n)\) time because each split takes \(O(1)\) time and we perform \(O(\log n)\) splits
- Thus, an insertion in a (2,4) tree takes \(O(\log n)\) time
Deletion

- We reduce deletion of an item to the case where the item is at the node with leaf children.
- Otherwise, we replace the item with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter item.
- Example: to delete key 24, we replace it with 27 (inorder successor).

\[ \begin{array}{l}
\text{Old tree:} \\
\text{10 15 24} \\
\text{2 8} \\
\text{12 18} \\
\text{32 35} \\
\text{New tree:} \\
\text{10 15 27} \\
\text{2 8} \\
\text{12 18} \\
\text{32 35} \\
\end{array} \]

Underflow and Fusion

- Deleting an item from a node \( v \) may cause an underflow, where node \( v \) becomes a 1-node with one child and no keys.
- To handle an underflow at node \( v \) with parent \( u \), we consider two cases.
  - Case 1: the adjacent siblings of \( v \) are 2-nodes.
    - Fusion operation: we merge \( v \) with an adjacent sibling \( w \) and move an item from \( u \) to the merged node \( v' \).
    - After a fusion, the underflow may propagate to the parent \( u \).

\[ \begin{array}{l}
\text{Old tree:} \\
\text{2 5 7} \\
\text{10} \\
\text{4 9} \\
\text{6 8} \\
\text{2} \\
\text{New tree:} \\
\text{2 5 7} \\
\text{10} \\
\text{4 8} \\
\text{6} \\
\end{array} \]

Underflow and Transfer

- To handle an underflow at node \( v \) with parent \( u \), we consider two cases.
  - Case 2: an adjacent sibling \( w \) of \( v \) is a 3-node or a 4-node.
    - Transfer operation: we move a child of \( w \) to \( v \), move an item from \( u \) to \( v \), move an item from \( w \) to \( u \).
    - After a transfer, no underflow occurs.
Analysis of Deletion

- Let $T$ be a (2,4) tree with $n$ items
  - Tree $T$ has $O(\log n)$ height
- In a deletion operation
  - We visit $O(\log n)$ nodes to locate the node from which to delete the item
  - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - Each fusion and transfer takes $O(1)$ time
- Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time

Red-Black Tree

- A Binary Search Tree.
- Every node in this tree is colored in either Red or Black.
- A historically popular alternative to the AVL tree.
- Operation on red-black trees take $O(\log n)$ time in the worst case.
Red-Black Tree

- **Root Property:** The root is black
- **External Property:** Every external node is black
- **Internal Property:** The children of a red node are black
- **Depth Property:** All the external nodes have the same black depth

![Red-Black Tree Diagram]

Height of a Red-Black Tree

- **Theorem:** A red-black tree storing $n$ items has height $O(\log n)$
- **Proof:**
  - **Depth Property:** All external nodes have same black depth $d$.
    - If all nodes were black then
      - $d \leq \log(n+1)$
    - **Internal node Property:** The children of a red node are black.
      - i.e. $h \leq 2d$
    - Thus
      - $\log(n+1) \leq h \leq 2 \log(n+1)$
    - So height is $O(\log n)$

By the above theorem, searching in a red-black tree takes $O(\log n)$ time

Insertion

- We execute the insertion algorithm for binary search trees
- Let the newly inserted node is root then color it black else color it red
- We preserve the root, external, and depth properties
  - If the parent of the node is black, we also preserve the internal property and we are done.
  - Else (parent is red) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree
- Example: Sequence 6, 3, 8, 4

![Insertion Diagram]
Remedying a Double Red

- Consider a double red with child \( z \) and parent \( v \), and let \( w \) be the sibling of \( v \)

Case 1: \( w \) is black
  - Restructure: Same as done for AVL trees.

Restructuring

- A restructuring remedies a child-parent double red when the parent red node has a black sibling
- The internal property is restored and the other properties are preserved

Restructuring (cont.)

- There are four restructuring configurations depending on whether the double red nodes are left or right children
Remedying a Double Red

- Consider a double red with child $z$ and parent $v$, and let $w$ be the sibling of $v$

Case 2: $w$ is red
  - The double red corresponds to an overflow
  - Recolor and continue up

Recoloring

- A recoloring remedies a child-parent double red when the parent red node has a red sibling
- The parent $v$ and its sibling $w$ become black and the grandparent $u$ becomes red, unless it is the root
- The double red violation may propagate to the grandparent $u$