CHAPTER 11
SEARCH TREES

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
BINARY SEARCH TREES

• A binary search tree is a binary tree storing entries \((k, e)\) (i.e., key-value pairs) at its internal nodes and satisfying the following property:
  • Let \(u\), \(v\), and \(w\) be three nodes such that \(u\) is in the left subtree of \(v\) and \(w\) is in the right subtree of \(v\). Then \(\text{key}(u) \leq \text{key}(v) \leq \text{key}(w)\)

• External nodes do not store items

• An inorder traversal of a binary search trees visits the keys in increasing order
To search for a key $k$, we trace a downward path starting at the root.

The next node visited depends on the outcome of the comparison of $k$ with the key of the current node.

If we reach a leaf, the key is not found.

Example: `get(4)`
- Call `Search(4, root)`

Algorithms for nearest neighbor queries are similar.

Algorithm `Search(k, v)`
Input: Key $k$, node $v$
Output: Node with key = $k$

1. if $v$.isExternal() then
2. return $v$
3. if $k < v$.key() then
4. return `Search(k, v.left())`
5. else if $k = v$.key() then
6. return $v$
7. else // $k > v$.key()
8. return `Search(k, v.right())`
To perform operation `put(k, v)`, we search for key `k` (using `Search(k)`).

Assume `k` is not already in the tree, and let `w` be the leaf reached by the search.

We insert `k` at node `w` and expand `w` into an internal node.

Example: insert 5.
EXERCISE
BINARY SEARCH TREES

• Insert into an initially empty binary search tree items with the following keys (in this order). Draw the resulting binary search tree
  • 30, 40, 24, 58, 48, 26, 11, 13
DELETION

• To perform operation $\text{remove}(k)$, we search for key $k$

• Assume key $k$ is in the tree, and let $v$ be the node storing $k$

• If node $v$ has a leaf child $w$, we remove $v$ and $w$ from the tree with operation $\text{removeExternal}(w)$, which removes $w$ and its parent

• Example: remove 4
DELETION (CONT.)

• We consider the case where the key $k$ to be removed is stored at a node $v$ whose children are both internal
  • we find the internal node $w$ that follows $v$ in an inorder traversal
  • we copy $w.key()$ into node $v$
  • we remove node $w$ and its left child $z$ (which must be a leaf) by means of operation $\text{removeExternal}(z)$

• Example: remove 3
EXERCISE
BINARY SEARCH TREES

• Insert into an initially empty binary search tree items with the following keys (in this order). Draw the resulting binary search tree
  • 30, 40, 24, 58, 48, 26, 11, 13

• Now, remove the item with key 30. Draw the resulting tree

• Now remove the item with key 48. Draw the resulting tree.
PERFORMANCE

• Consider an ordered map with $n$ items implemented by means of a binary search tree of height $h$
  - Space used is $O(n)$
  - Methods $get(k)$, $put(k, v)$, and $remove(k)$ take $O(h)$ time

• The height $h$ is $O(n)$ in the worst case and $O(\log n)$ in the best case
AVL TREES
AVL TREE DEFINITION

- AVL trees are balanced
- An AVL Tree is a binary search tree such that for every internal node $v$ of $T$, the heights of the children of $v$ can differ by at most 1

An example of an AVL tree where the heights are shown next to the nodes:
HEIGHT OF AN AVL TREE

• Fact: The height of an AVL tree storing \( n \) keys is \( O(\log n) \).

• Proof: Let us bound \( n(h) \): the minimum number of internal nodes of an AVL tree of height \( h \).
  
  • We easily see that \( n(1) = 1 \) and \( n(2) = 2 \).
  
  • For \( n > 2 \), an AVL tree of height \( h \) contains the root node, one AVL subtree of height \( h - 1 \) and another of height \( h - 2 \).
  
  • That is, \( n(h) = 1 + n(h - 1) + n(h - 2) \)
  
  • Knowing \( n(h - 1) > n(h - 2) \), we get \( n(h) > 2n(h - 2) \). So
    
    • \( n(h) > 2n(h - 2) > 4n(h - 4) > 8n(n - 6) \), ... (by induction),
    
    • \( n(h) > 2^i n(h - 2i) \)
  
  • Solving the base case we get: \( n(h) > 2^{\frac{h-1}{2}} \)
  
  • Taking logarithms: \( h < 2 \log n(h) + 2 \)
  
  • Thus the height of an AVL tree is \( O(\log n) \)
Insertion in an AVL Tree

- Insertion is as in a binary search tree.
- Always done by expanding an external node.
- Example insert 54:

Before Insertion:

```
44
/\    /
17 78 32
/\/  /\  /
48 50 88
```

After Insertion:

```
44
/\    /
17 78 32
/\/  /\  /
48 50 88
/\    /\  /
54 62 50 88
```
TRINODE RESTRUCTURING

- let \((a, b, c)\) be an inorder listing of \(x, y, z\)
- perform the rotations needed to make \(b\) the topmost node of the three

- **case 1**: single rotation (a left rotation about \(a\))

- **case 2**: double rotation (a right rotation about \(c\), then a left rotation about \(a\))

(Other two cases are symmetrical)
INSERTION EXAMPLE, CONTINUED

unbalanced...

...balanced
Restructuring
Single Rotations

single rotation

$T_0$

$T_1$

$T_2$

$T_3$

$a = x$

$b = y$

$c = z$

$T_0$

$T_1$

$T_2$

$T_3$

$a = x$

$b = y$

$c = z$

$T_0$

$T_1$

$T_2$

$T_3$

$a = x$

$b = y$

$c = z$

$T_0$

$T_1$

$T_2$

$T_3$
RESTRUCTURING
DOUBLE ROTATIONS
EXERCISE
AVL TREES

• Insert into an initially empty AVL tree items with the following keys (in this order). Draw the resulting AVL tree
  • 30, 40, 24, 58, 48, 26, 11, 13
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:

Before deletion of 32:

```
44
  /   \
17    62
  /  \\ /  \
32  50 78
  /   /  \
48 54 88
```

After deletion:

```
44
  /   \
17    62
  /  \\ /  \
50 78
  /   /  \
48 54 88
```
REBALANCING AFTER A REMOVAL

- Let \( z \) be the first unbalanced node encountered while travelling up the tree from \( w \) (parent of removed node). 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 `restructure(x)` to restore balance at \( z \).
REBALANCING AFTER A REMOVAL

- 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.
  - This can happen at most $O(\log n)$ times. Why?
EXERCISE
AVL TREES

• Insert into an initially empty AVL tree items with the following keys (in this order). Draw the resulting AVL tree
  • 30, 40, 24, 58, 48, 26, 11, 13

• Now, remove the item with key 48. Draw the resulting tree

• Now, remove the item with key 58. Draw the resulting tree
RUNNING TIMES FOR AVL TREES

- **A single restructure is** $O(1)$ — using a linked-structure binary tree
- **get($k$)** takes $O(\log n)$ time — height of tree is $O(\log n)$, no restructures needed
- **put($k, v$)** takes $O(\log n)$ time
  - Initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
- **remove($k$)** takes $O(\log n)$ time
  - Initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
OTHER TYPES OF SELF-BALANCING TREES

• **Splay Trees** – A binary search tree which uses an operation splay($x$) to allow for amortized complexity of $O(\log n)$

• **(2, 4) Trees** – A multiway search tree where every node stores internally a list of entries and has 2, 3, or 4 children. Defines self-balancing operations

• **Red-Black Trees** – A binary search tree which colors each internal node red or black. Self-balancing dictates changes of colors and required rotation operations