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\((k, v\.right())\)
INSERTION

- To perform operation \( \text{put}(k, v) \), we search for key \( k \) (using \( \text{Search}(k) \))
- 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 \texttt{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 \texttt{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 `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 $\text{get}(k)$, $\text{put}(k, v)$, and $\text{remove}(k)$ 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:
事实：AVL树存储n个键的高度为$O(\log n)$。

证明：让我们来求$n(h)$：一个高度为$h$的AVL树的内部节点的最小数目。

我们很容易看出$n(1) = 1$ 且 $n(2) = 2$。

对于$n > 2$，一个高度为$h$的AVL树包含根节点，一个高度为$h-1$的AVL子树和另一个高度为$h-2$的AVL子树。

那也就是说，$n(h) = 1 + n(h-1) + n(h-2)$。

知道$n(h-1) > n(h-2)$，我们可以得到$n(h) > 2n(h-2)$。因此

* $n(h) > 2n(h-2) > 4n(h-4) > 8n(n-6),...$ (通过归纳法)，
* $n(h) > 2^i n(h-2i)$

解决基础情况得到：$n(h) > 2^\frac{h-1}{2}$

取对数：$h < 2 \log n(h) + 2$

因此AVL树的高度是$O(\log n)$。
INSERTION IN AN AVL TREE

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example insert 54
- Might need to rebalance the tree!
  - Start from the newly inserted node walking to the parent
  - If an imbalance is detected, restructure that node and continue searching

Before Insertion

After Insertion, note the imbalance at node 78
TRINODE RESTRUCTURING

1. Let $z$ be the imbalanced node

2. Let $y$ be the taller child of $z$

3. Let $x$ be the tallest child of $y$ (or left child if the heights are equal)

4. Relabel $(z, y, x)$ as $(a, b, c)$ based on the order of the keys

5. Restructure the tree about node $b$ to become the topmost with $a$ on $b$'s left and $c$ on $b$'s right, the prior subtrees of $a$, $b$, $c$ get assigned to $a$ and $c$ in an inorder fashion

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 IN AN AVL TREE

Before restructuring with labeling

After restructuring
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

After deletion, note the imbalance at node 17
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 $\text{restructure}(x)$ 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.

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

• $\text{get}(k)$ takes $O(\log n)$ time – height of tree is $O(\log n)$, no restructures needed

• $\text{put}(k, v)$ takes $O(\log n)$ time
  • Initial find is $O(\log n)$
  • Restructuring up the tree, maintaining heights is $O(\log n)$

• $\text{remove}(k)$ 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 \( \text{splay}(x) \) 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
INTERVIEW QUESTION 1

• Given a sorted array with unique integer elements, write an algorithm to create a binary search tree with minimal height.
  • Hint: Recursion
INTERVIEW QUESTION 2

- Implement a function to check if a binary tree is a binary search tree.
EXAM 3 - PROGRAMMING

• Hack sheet – you can have a single 8 ½" x 11" paper with handwritten notes on both sides with you during the exam. You can put anything on it, but syntax of bounded type parameters might be useful.
• Can bring blank pieces of paper for scratch work.
• Internet only to access the API. Focus is on programming skills.
• Be sure to follow all naming conventions in the exam.
• Format – 2 questions and a bonus (there will be File IO and unit testing)
  • Q1 – Advanced Java Generics
  • Q2 – Java Priority Queues, Java Sets, or Java Maps
  • Bonus – ?
EXAM 3 - WRITTEN

• Separate Hack sheet – you can have a single 8 ½" x 11" paper with handwritten notes on both sides with you during the exam. You can put anything on it, but summary slides and the search tree processes make great candidates.

• Can bring blank sheets of paper for scratch work.

• Focus on algorithmic skills and programming concepts.

• Format – 5 questions and a bonus (will only have to write a single algorithm)
  • Q1 – T/F + fill-in-the-blank (similar to quizzes)
  • Q2 – Tracing with hash tables (similar to quizzes)
  • Q3 – Tracing with binary search trees (similar to quizzes)
  • Q4 – Write and/or analyze algorithm using/related to Priority Queues (similar to homework)
  • Q5 – Write and/or analyze algorithm using/related to Maps (similar to homework)
  • Bonus – ?