Lecture 4: Balanced Binary Search Trees

Lecture Overview

- The importance of being balanced
- AVL trees
  - Definition
  - Balance
  - Insert
- Other balanced trees
- Data structures in general

Readings

CLRS Chapter 13.1 and 13.2 (but different approach: red-black trees)

Recall: Binary Search Trees (BSTs)

- rooted binary tree
- each node has
  - key
  - left pointer
  - right pointer
  - parent pointer

See Fig. 1

Figure 1: Heights of nodes in a BST
- BST property (see Fig. 2).
- height of node = length (≠ edges) of longest downward path to a leaf (see CLRS B.5 for details).

The Importance of Being Balanced:
- BSTs support insert, min, delete, rank, etc. in \(O(h)\) time, where \(h = \text{height of tree} \) (= height of root).
- \(h\) is between \(\lg(n)\) and \(n\): Fig. 3.

- balanced BST maintains \(h = O(\lg n) \) ⇒ all operations run in \(O(\lg n)\) time.
AVL Trees:

Definition

AVL trees are self-balancing binary search trees. These trees are named after their two inventors G.M. Adel’son-Vel’skii and E.M. Landis.[1]

An AVL tree is one that requires heights of left and right children of every node to differ by at most ±1. This is illustrated in Fig. 4.

Figure 4: AVL Tree Concept

In order to implement an AVL tree, follow two critical steps:

- Treat nil tree as height −1.

- Each node stores its height. This is inherently a DATA STRUCTURE AUGMENTATION procedure, similar to augmenting subtree size. Alternatively, one can just store difference in heights.

A good animation applet for AVL trees is available at [this link]. To compare Binary Search Trees and AVL balancing of trees use code provided [here].

Balance:

The balance is the worst when every node differs by 1.
Let \( N_h = \min (\# \text{ nodes}) \).

\[
\Rightarrow N_h = N_{h-1} + N_{h-2} + 1 \\
> 2N_{h-2} \\
\Rightarrow N_h > 2^{h/2} \\
\Rightarrow h < \frac{1}{2} \lg h
\]

Alternatively:

\[
N_h > F_n \quad (n^{th} \text{ Fibonacci number}) \\
\text{In fact,} N_h = F_{n+2} - 1 \quad \text{(simple induction)} \\
F_h = \frac{\phi^h}{\sqrt{5}} \quad \text{(rounded to nearest integer)} \\
\text{where } \phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \quad \text{(golden ratio)} \\
\Rightarrow \max h \approx \log_\phi(n) \approx 1.440 \lg(n)
\]

AVL Insert:

1. insert as in simple BST
2. work your way up tree, restoring AVL property (and updating heights as you go).

Each Step:

- suppose \( x \) is lowest node violating AVL
- assume \( x \) is right-heavy (left case symmetric)
- if \( x \)'s right child is right-heavy or balanced: follow steps in Fig. 5
- else follow steps in Fig. 6
- then continue up to \( x \)'s grandparent, greatgrandparent . . .
Figure 5: AVL Insert Balancing

Figure 6: AVL Insert Balancing
**Example:** An example implementation of the AVL Insert process is illustrated in Fig. 7.

![Insertion Process Diagram]

**Figure 7:** Illustration of AVL Tree Insert Process

**Comment 1.** In general, process may need several rotations before an Insert is completed.

**Comment 2.** Delete(-min) harder but possible.
Balanced Search Trees:

There are many balanced search trees.

<table>
<thead>
<tr>
<th>Balanced Search Trees</th>
<th>Authors/Authors</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVL Trees</td>
<td>Adel’son-Velsii and Landis</td>
<td>1962</td>
</tr>
<tr>
<td>B-Trees/2-3-4 Trees</td>
<td>Bayer and McCreight</td>
<td>1972 (see CLRS 18)</td>
</tr>
<tr>
<td>BB[α] Trees</td>
<td>Nievergelt and Reingold</td>
<td>1973</td>
</tr>
<tr>
<td>Red-black Trees</td>
<td>CLRS Chapter 13</td>
<td></td>
</tr>
<tr>
<td>Splay-Trees</td>
<td>Sleator and Tarjan</td>
<td>1985</td>
</tr>
<tr>
<td>Skip Lists</td>
<td>Pugh</td>
<td>1989</td>
</tr>
<tr>
<td>Scapegoat Trees</td>
<td>Galperin and Rivest</td>
<td>1993</td>
</tr>
<tr>
<td>Treaps</td>
<td>Seidel and Aragon</td>
<td>1996</td>
</tr>
</tbody>
</table>

**Note 1.** Skip Lists and Treaps use random numbers to make decisions fast with high probability.

**Note 2.** Splay Trees and Scapegoat Trees are “amortized”: adding up costs for several operations $\implies$ fast on average.
Splay Trees

Upon access (search or insert), move node to root by sequence of rotations and/or double-rotations (just like AVL trees). Height can be linear but still $O(\log n)$ per operation “on average” (amortized)

Note: We will see more on amortization in a couple of lectures.

Optimality

- For BSTs, cannot do better than $O(\log n)$ per search in worst case.
- In some cases, can do better e.g.
  - in-order traversal takes $\Theta(n)$ time for $n$ elements.
  - put more frequent items near root

A Conjecture: Splay trees are $O(\text{best BST})$ for every access pattern.

- With fancier tricks, can achieve $O(\log \log u)$ performance for integers $1 \cdots u$ [Van Ernde Boas; see 6.854 or 6.851 (Advanced Data Structures)]

Big Picture:

Abstract Data Type (ADT): interface spec.

  e.g. Priority Queue:

  - $Q = \text{new-empty-queue}()$
  - $Q.\text{insert}(x)$
  - $x = Q.\text{deletemin}()$

vs.

Data Structure (DS): algorithm for each op.

There are many possible DSs for one ADT. One example that we will discuss much later in the course is the “heap” priority queue.