Lecture 6: Balanced Binary Search Trees

Lecture Overview

• The importance of being balanced
• AVL trees
  – Definition and balance
  – Rotations
  – Insert
• Other balanced trees
• Data structures in general

Recall: Binary Search Trees (BSTs)

• rooted binary tree
• each node has
  – key
  – left pointer
  – right pointer
  – parent pointer

See Fig. 1

• 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, delete, min, max, next-larger, next-smaller, etc. in $O(h)$ time, where $h =$ height of tree (= height of root).

- Height $h$ of the tree should be $O(\lg n)$.
  
  The tree in Fig. 3 looks like a linked list. We have achieved $O(n)$ not $O(\lg n)$!!

- $h$ is between $\lg n$ and $n$: Fig. 4.

- balanced BST maintains $h = O(\lg n) \Rightarrow$ all operations run in $O(\lg n)$ time.
AVL Trees: Adel’son-Vel’skii & Landis 1962

For every node, require heights of left & right children to differ by at most ±1.

- treat nil tree as height -1
- each node stores its height (DATA STRUCTURE AUGMENTATION) (like subtree size) (alternatively, can just store difference in heights)

This is illustrated in Fig. 3
Balance:

Worst when every node differs by 1 — let $N_h = (\text{min.}) \ # \ \text{nodes in height}-h \ \text{AVL tree}$

$$N_h = N_{h-1} + N_{h-2} + 1 \quad (1 \ for \ the \ root \ node)$$

$$> 2N_{h-2}$$

$$\Rightarrow N_h > 2^{h/2}$$

$$\Rightarrow h < 2 \lg N_h$$

Alternatively:

$N_h > F_h \ (n\text{th Fibonacci number})$

- In fact $N_h = F_{n+1} - 1$ (simple induction)

- $F_h = \frac{\varphi^h}{\sqrt{5}}$ rounded to nearest integer where $\varphi = \frac{1 + \sqrt{5}}{2} \approx 1.618$ (golden ratio)

- $\Rightarrow \ max. \ h \approx \log_\varphi n \approx 1.440 \lg n$

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 (takes $O(\log n)$ time to discover violating node)

• assume $x$ is right-heavy (left case symmetric)

• if $x$’s right child is right-heavy or balanced: follow steps in Fig. 6. Note that $A \leq x \leq B \leq y \leq C$ both before and after rotation satisfying BST property.

![Diagram of AVL Insert Balancing](image)

Figure 6: AVL Insert Balancing

• else: follow steps in Fig. 7

• At most 2 rotations are required as in Figure 7 to restore balance.
Example: An example implementation of the AVL Insert process is illustrated in Fig. 8.

Comment 1. Delete(-min) is similar — harder but possible. Delete requires $O(\log n)$ rotations, not just 2 as in insert.

Insert(23)
x = 29: left-left case

Done

Insert(55)

x = 65: left-right case

Done

Figure 8: Illustration of AVL Tree Insert Process
AVL sort:

- insert each of $n$ items into AVL tree $\Theta(n \lg n)$
- in-order traversal $\Theta(n)$
  $$\Theta(n) \quad \Theta(n \lg n)$$

Balanced Search Trees:

There are many balanced search trees.

- AVL Trees
- B-Trees/2-3-4 Trees
- BB[$\alpha$] Trees
- Red-black Trees
- (A) — Splay-Trees
- (R) — Skip Lists
- (A) — Scapegoat Trees
- (R) — Treaps

\begin{align*}
\text{AVL Trees} & \quad \text{Adel’son-Velsii and Landis 1962} \\
\text{B-Trees/2-3-4 Trees} & \quad \text{Bayer and McCreight 1972 (see CLRS 18)} \\
\text{BB[$\alpha$] Trees} & \quad \text{Nievergelt and Reingold 1973} \\
\text{Red-black Trees} & \quad \text{CLRS Chapter 13} \\
\text{(A) — Splay-Trees} & \quad \text{Sleator and Tarjan 1985} \\
\text{(R) — Skip Lists} & \quad \text{Pugh 1989} \\
\text{(A) — Scapegoat Trees} & \quad \text{Galperin and Rivest 1993} \\
\text{(R) — Treaps} & \quad \text{Seidel and Aragon 1996}
\end{align*}

\begin{itemize}
\item (R) = use random numbers to make decisions fast with high probability
\item (A) = “amortized”: adding up costs for several operations $\implies$ fast on average
\end{itemize}

For example, Splay Trees are a current research topic — see 6.854 (Advanced Algorithms) and 6.851 (Advanced Data Structures)

Big Picture:

Abstract Data Type (ADT): interface spec.

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.

\begin{align*}
\text{Priority Queue ADT} & \quad \text{heap} & \quad \text{AVL tree} \\
Q = \text{new-empty-queue()} & \quad \Theta(1) & \quad \Theta(1) \\
Q.\text{insert}(x) & \quad \Theta(\lg n) & \quad \Theta(\lg n) \\
x = Q.\text{deletemin()} & \quad \Theta(\lg n) & \quad \Theta(\lg n) \\
x = Q.\text{findmin()} & \quad \Theta(1) & \quad \Theta(\lg n) \rightarrow \Theta(1)
\end{align*}
The improvement for \texttt{findmin()} can be made by using augmentation to store, for every node \( x \), the minimum key in \( x \)’s subtree. Call this \( x.\text{minval} \). Then, we can simply look at the root’s \( \text{minval} \).

<table>
<thead>
<tr>
<th>Predecessor/Successor ADT</th>
<th>heap</th>
<th>AVL tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S = \text{new-empty()} )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>( S.\text{insert}(x) )</td>
<td>( \Theta(\lg n) )</td>
<td>( \Theta(\lg n) )</td>
</tr>
<tr>
<td>( S.\text{delete}(x) )</td>
<td>( \Theta(\lg n) )</td>
<td>( \Theta(\lg n) )</td>
</tr>
<tr>
<td>( y = S.\text{predecessor}(x) \rightarrow \text{next-smaller} )</td>
<td>( \Theta(n) )</td>
<td>( \Theta(\lg n) )</td>
</tr>
<tr>
<td>( y = S.\text{successor}(x) \rightarrow \text{next-larger} )</td>
<td>( \Theta(n) )</td>
<td>( \Theta(\lg n) )</td>
</tr>
</tbody>
</table>