Splay Trees and B-Trees

CSE 373
Data Structures
Lecture 9

Self adjustment for better living

- Ordinary binary search trees have no balance conditions
  - what you get from insertion order is it
- Balanced trees like AVL trees enforce a balance condition when nodes change
  - tree is always balanced after an insert or delete
- Self-adjusting trees get reorganized over time as nodes are accessed

Splay Trees

- Splay trees are tree structures that:
  - Are not perfectly balanced all the time
  - Data most recently accessed is near the root.
- The procedure:
  - After node X is accessed, perform “splaying” operations to bring X to the root of the tree.
  - Do this in a way that leaves the tree more balanced as a whole

Splay Tree Terminology

- Let X be a non-root node with ≥ 2 ancestors.
  - P is its parent node.
  - G is its grandparent node.

Zig-Zig and Zig-Zag

- Parent and grandparent in different directions.

Splay Tree Operations

1. Helpful if nodes contain a parent pointer.

2. When X is accessed, apply one of six rotation routines.
   - Single Rotations (X has a P (the root) but no G)
     ZigFromLeft, ZigFromRight
   - Double Rotations (X has both a P and a G)
     ZigZigFromLeft, ZigZigFromRight
     ZigZagFromLeft, ZigZagFromRight

Zig at depth 1

- "Zig" is just a single rotation, as in an AVL tree
- Let R be the node that was accessed (e.g. using Find)

- ZigFromLeft moves R to the top → faster access next time

Zig at depth 1

- Suppose Q is now accessed using Find

- ZigFromRight moves Q back to the top

Zig-Zig operation

- "Zig-Zig" consists of two single rotations of the same direction (R is the node that was accessed)

Zig-Zag operation

- "Zig-Zag" consists of two rotations of the opposite direction (assume R is the node that was accessed)

Decreasing depth - "autobalance"
Splay Tree Insert and Delete

- Insert x
  - Insert x as normal then splay x to root.
- Delete
  - Splay x to root and remove it. Two trees remain, right subtree an left subtree.
  - Splay the max in the left subtree to the root
  - Attach the right subtree to the new root of the left subtree.

Example Insert

- Inserting in order 1,2,3,…,8
- Without self-adjustment
  - O(n^2) time

With Self-Adjustment

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

With Self-Adjustment

<table>
<thead>
<tr>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

O(n) time!!

Example Deletion

Analysis of Splay Trees

- Splay trees tend to be balanced
  - M operations takes time O(M log N) for M ≥ N operations on N items.
  - Amortized O(log n) time.
- Splay trees have good "locality" properties
  - Recently accessed items are near the root of the tree.
  - Items near an accessed are pulled toward the root.
Beyond Binary Search Trees: Multi-Way Trees

- B-tree of order 3 has 2 or 3 children per node

\[
\begin{array}{c}
\text{13} \\
\text{6} \quad \text{11} \\
\text{3} \quad \text{4} \\
\text{6} \quad \text{7} \quad \text{8} \\
\text{11} \quad 	ext{12} \\
\text{13} \quad \text{14} \\
\text{17} \quad \text{18}
\end{array}
\]

- Search for 8

B-Trees

B-Trees are multi-way search trees commonly used in database systems or other applications where data is stored externally on disks and keeping the tree shallow is important.

A B-Tree of order \(M\) has the following properties:
1. The root is either a leaf or has between 2 and \(M\) children.
2. All non-leaf nodes (except the root) have between \([M/2]\) and \(M\) children.
3. All leaves are at the same depth.

All data records are stored at the leaves. Leaves store between \([M/2]\) and \(M\) data records.

B-Tree Details

Each (non-leaf) internal node of a B-tree has:
- Between \([M/2]\) and \(M\) children.
- up to \(M-1\) keys \(k_1 < k_2 < \ldots < k_{M-1}\)

Keys are ordered so that:
\(k_1 < k_2 < \ldots < k_{M-1}\)

Properties of B-Trees

Children of each internal node are "between" the items in that node. Suppose subtree \(T_i\) is the \(i\)th child of the node:
- All keys in first subtree \(T_i < k_i\)
- All keys in last subtree \(T_M > k_{M-1}\)

Example: Searching in B-trees

- B-tree of order 3: also known as 2-3 tree (2 to 3 children)

\[
\begin{array}{c}
\text{13} \\
\text{6} \quad \text{11} \\
\text{3} \quad \text{4} \\
\text{6} \quad \text{7} \quad \text{8} \\
\text{11} \quad \text{12} \\
\text{13} \quad \text{14} \\
\text{17} \quad \text{18}
\end{array}
\]

- Examples: Search for 9, 14, 12
- Note: If leaf nodes are connected as a Linked List, B-tree is called a B+ tree — Allows sorted list to be accessed easily

Inserting into B-Trees

- Insert X: Do a Find on X and find appropriate leaf node
  - If leaf node is not full, fill in empty slot with X
  - E.g. Insert 5
  - If leaf node is full, split leaf node and adjust parents up to root node
    - E.g. Insert 9
Deleting From B-Trees

- Delete X: Do a find and remove from leaf
  - Leaf underflows – borrow from a neighbor
  - E.g. 11
  - Leaf underflows and can’t borrow – merge nodes, delete parent
    - E.g. 17

Run Time Analysis of B-Tree Operations

- For a B-Tree of order M
  - Each internal node has up to M-1 keys to search
  - Each internal node has between \[ \lceil M/2 \rceil \] and M children
  - Depth of B-Tree storing N items is \( O(\log_{M/2} N) \)
- Find: Run time is:
  - \( O(\log M) \) to binary search which branch to take at each node
  - Total time to find an item is \( O(\text{depth} \times \log M) = O(\log N) \)

Summary of Search Trees

- Problem with Search Trees: Must keep tree balanced to allow
  - fast access to stored items
- AVL trees: Insert/Delete operations keep tree balanced
- Splay trees: Repeated Find operations produce balanced trees
- Multi-way search trees (e.g. B-Trees): More than two children
  - per node allows shallow trees; all leaves are at the same depth
  - keeping tree balanced at all times