Symbol Table Review

Symbol table: key-value pair abstraction.
- Insert a value with specified key.
- Search for value given key.
- Delete value with given key.

Randomized BST.
- $O(\log N)$ time per op. [unless you get ridiculously unlucky]
- Store subtree count in each node.
- Generate random numbers for each insert/delete op.

This lecture. 2-3-4 trees, red-black trees, B-trees.

2-3-4 Trees

2-3-4 tree. Generalize node to allow multiple keys; keep tree balanced.

Perfect balance. Every path from root to leaf has same length.

Allow 1, 2, or 3 keys per node.
- 2-node: one key, two children.
- 3-node: two keys, three children.
- 4-node: three keys, four children.

2-3-4 Tree
2-3-4 Tree: Search

Search.
- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

2-3-4 Tree: Insert

Insert.
- Search to bottom for key.
- 2-node at bottom: convert to 3-node.
- 3-node at bottom: convert to 4-node.
- 4-node at bottom: ??

2-3-4 Tree: Splitting Four Nodes

Transform tree on the way down.
- Ensures last node is not a 4-node.
- Local transformation to split 4-nodes:

Invariant. Current node is not a 4-node.
Consequence. Insertion at bottom is easy since it’s not a 4-node.

Ex. To split a four node, move middle key up.
2-3-4 Tree

Tree grows up from the bottom.

Direct implementation. Complicated because of:
- Maintaining multiple node types.
- Implementation of getChild().
- Large number of cases for split().

```java
private void insert(Key key, Val val) {
    Node x = root;
    while (x.getChild(key) != null) {
        x = x.getChild(key);
        if (x.is4Node()) x.split();
    }
    if (x.is2Node()) x.make3Node(key, val);
    else if (x.is3Node()) x.make4Node(key, val);
}
```

2-3-4 Tree: Implementation?

Tree height.
- Worst case: $\log N$ [all 2-nodes]
- Best case: $\log_4 N = \frac{1}{2} \log N$ [all 4-nodes]
- Between 10 and 20 for a million nodes.
- Between 15 and 30 for a billion nodes.

Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>$N$</td>
<td>$1$</td>
<td>$1$</td>
<td>$N$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>Hashing</td>
<td>$N$</td>
<td>$1$</td>
<td>$1$</td>
<td>$N$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Randomized BST</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>Splay</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>2-3-4</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
</tbody>
</table>

Note. Comparison within nodes not accounted for.
Red-Black Trees

Represent 2-3-4 tree as a BST.
- Use “internal” edges for 3- and 4- nodes.

not 1-1 because 3-nodes swing either way.
- Correspondence between 2-3-4 trees and red-black trees.

Two easy cases.
Switch colors.

Two hard cases.
Use rotations.
- do single rotation
- do double rotation

Red-Black Tree: Splitting Nodes

Two easy cases.
Switch colors.

Red-Black Tree: Splitting Nodes

Two easy cases.
Switch colors.

**Red-Black Tree: Balance**

**Property A.** Every path from root to leaf has same number of black links.

**Property B.** At most one red link in-a-row.

**Property C.** Height of tree is less than $2 \log N + 2$.

**Symbol Table: Implementations Cost Summary**

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array</td>
<td>$\log N$</td>
<td>N</td>
<td>N</td>
<td>$\log N$</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>N</td>
<td>1</td>
<td>N</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Hashing</td>
<td>N</td>
<td>1</td>
<td>$1^{*}$</td>
<td>$1^{*}$</td>
<td>$1^{*}$</td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>$\log N^{+}$</td>
<td>$\log N^{+}$</td>
<td>$\log N^{+}$</td>
</tr>
<tr>
<td>Randomized BST</td>
<td>$\log N^{†}$</td>
<td>$\log N^{†}$</td>
<td>$\log N^{†}$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>Splay</td>
<td>$\log N^{‡}$</td>
<td>$\log N^{‡}$</td>
<td>$\log N^{‡}$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>Red-Black</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
</tbody>
</table>

* assumes hash map is random for all keys
† $N$ is the number of nodes ever inserted
§ probabilistic guarantee
§ amortized guarantee

**Note.** Comparison within nodes are accounted for.
Red-Black Trees: Practice

Red-black trees vs. splay trees.
- Fewer rotations than splay trees.
- One extra bit per node for color.

Red-black trees vs. hashing.
- Hashing code is simpler and usually faster:
  - arithmetic to compute hash vs. comparison.
- Hashing performance guarantee is weaker.
- BSTs have more flexibility and can support wider range of ops.

In the wild. Red-black trees are widely used as system symbol tables.
- Java: java.util.TreeMap, java.util.TreeSet.
- C++ STL: map, multimap, multiset.
- Linux kernel: linux/rbtree.h.

B-Trees

B-Tree. Generalizes 2-3-4 trees by allowing up to M links per node.

Main application: file systems.
- Reading a page into memory from disk is expensive.
- Accessing info on a page in memory is free.
- Goal: minimize # page accesses.
- Node size M = page size.

Space-time tradeoff.
- M large ⇒ only a few levels in tree.
- M small ⇒ less wasted space.
- Typical M = 1000, N < 1 trillion.

Bottom line. Number of page accesses is $\log_M N$ per op.

3 or 4 in practice!
B-Trees in the Wild

Variants.
- B trees: Bayer-McCreight. [1972, Boeing]
- B+ trees: all data in external nodes.
- B* trees: keeps pages at least 2/3 full.
- R-trees for spatial searching: GIS, VLSI.

File systems.
- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.

Databases.
- Most common index type in modern databases.
  - ORACLE, DB2, INGRES, SQL, PostgreSQL, ...

Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array</td>
<td>log N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
<td>N / 2</td>
<td>N / 2</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Hashing</td>
<td>N</td>
<td>1</td>
<td>N</td>
<td>1*</td>
<td>1*</td>
<td>1*</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>Randomized BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>Splay</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>Red-Black</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

B-Tree. Number of page accesses is \( \log N \) per op.

Summary

Goal. ST implementation with \( \log N \) guarantee for all ops.
- Probabilistic: randomized BST.
- Amortized: splay tree.
- Worst-case: red-black tree.
- Algorithms are variations on a theme: rotations when inserting.

Abstraction extends to give search algorithms for huge files.
- B-tree.
Splay Trees

Splay trees = self-adjusting BST.
- Tree automatically reorganizes itself after each op.
- After inserting x or searching for x, rotate x up to root using double rotations.
- Tree remains "balanced" without explicitly storing any balance information.

Amortized guarantee: any sequence of N ops, starting from empty splay tree, takes \(O(N \log N)\) time.
- Height of tree can be \(N\).
- Individual op can take linear time.

Splay.
- Check two links above current node.
- ZIG-ZAG: if orientations differ, same as root insertion.
- ZIG-ZIG: if orientations match, do top rotation first.
Splay Trees

Splay.
- Check two links above current node.
- ZIG-ZAG: if orientations differ, same as root insertion.
- ZIG-ZIG: if orientations match, do top rotation first.

![Splay Tree Example](image)

Search for 1.

ZIG-ZIG

Splay Example

Search for 1.

ZIG-ZIG

Splay Example

Root = Splay

Root Insertion

Splay Insertion

Splay Trees
Splay Example

Search for 1.

ZIG-ZIG

ZIG
Splay Example

Search for 2.

Splay Example

Splay Trees

Intuition.
- Splay rotations halve search path.
- Reduces length of path for many other nodes in tree.

Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array</td>
<td>log N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>N</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Hashing</td>
<td>N</td>
<td>1</td>
<td>N</td>
<td>1*</td>
<td>1*</td>
<td>1*</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>Randomized BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>Splay</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

* assumes we know location of node to be deleted
† if delete allowed, insert/search become \sqrt{N}
‡ probabilistic guarantee
§ amortized guarantee

Splay: Sequence of N ops takes linearithmic time.
Ahead: Can we do all ops in log N time guaranteed?