Binary Search Trees

Binary search trees
Randomized BSTs


Guaranteeing Performance

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

Challenge 1: guarantee symbol table performance.
- Make average case independent of input distribution.
- Extend average case guarantee to worst-case.
- Remove assumption on having a good hash function.
- Remove expensive (but infrequent) re-doubling operations.

Challenge 2: expand interface when keys are ordered.
- Find the \( i \)th largest key.
- Range searching.

Binary Search Tree

Binary search tree: binary tree in symmetric order.

Binary tree is either:
- Empty.
- A key-value pair and two binary trees.

Symmetric order:
- Keys in nodes.
- No smaller than left subtree.
- No larger than right subtree.

A BST is a reference to a node.

A Node is comprised of four fields:
- A key and a value.
- A reference to the left and right subtree.

private static class Node {
    Comparable key; // key can be any Comparable object
    Object value;
    Node left;
    Node right;
}

helper class

Binary Search Tree in Java

root

left left

root

left

left

12 54 79

43 84 99

97 64 25

33 53 97

68 14 43

12 54 79
Tree Shape

- Tree shape.
  - Many BSTs correspond to same input data.
  - Have different tree shapes.
  - Performance depends on shape.

BST Skeleton

```java
public class SymbolTable {
    private Node root;

    private static class Node {
        Comparable key;
        Object value;
        Node left, right;
        Node(Comparable key, Object value) {
            this.key = key;
            this.value = value;
        }
    }

    private static boolean less(Comparable k1, Comparable k2) { }
    private static boolean equals(Comparable k1, Comparable k2) { }
    public void put(Comparable key, Object value) { }
    public Object get(Comparable key) { }
}
```

BST Search

Search for specified key and return corresponding value or null.
- Code follows from BST definition.
- Use helper function to search for key in subtree rooted at h.

```java
public Object get(Comparable key) {
    return search(root, key);
}

private Object search(Node h, Comparable key) {
    if (h == null) return null;
    if (equals(key, h.key)) return h.value;
    if (less(key, h.key)) return search(h.left, key);
    else return search(h.right, key);
}
```

BST Insert

Insert key-value pair.
- Code follows from BST definition.
- Search, then insert.
- Simple (but tricky) recursive code.
- Duplicates allowed.

```java
public void put(Comparable key, Object value) {
    root = insert(root, key, value);
}

private Node insert(Node h, Comparable key, Object value) {
    if (h == null) return new Node(key, value);
    if (less(key, h.key)) h.left = insert(h.left, key, value);
    else h.right = insert(h.right, key, value);
    return h;
}
```
BST Construction

Insert the following keys into BST: A S E R C H I N G X M P L

BST Analysis

Cost of search and insert BST:
- Proportional to depth of node.
- 1-1 correspondence between BST and quicksort partitioning.
- Height of node corresponds to number of function calls on stack when node is partitioned.

Theorem. If keys are inserted in random order, then height of tree is $\Theta(\log N)$, except with exponentially small probability. Thus, search and insert take $O(\log N)$ time.

Problem. Worst-case search and insert are proportional to $N$.
- If nodes in order, tree degenerates to linked list.

Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td>Insert</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$\log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>$N$</td>
<td>$1$</td>
</tr>
<tr>
<td>Hashing</td>
<td>$N$</td>
<td>$1$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

BST: log $N$ insert and search if keys arrive in random order.  
Ahead: Can we make all ops log $N$ if keys arrive in arbitrary order?

Symbol Table: Delete

To delete a node:
- Case 1 (zero children): just remove it.
- Case 2 (one child): pass the child up.
- Case 3 (two children): find the next largest node using right-left* or left-right*, swap with next largest, remove as in Case 1 or 2.

Problem: strategy clumsy, not symmetric.  
Serious problem: trees not random (!!)
Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td>Insert</td>
</tr>
<tr>
<td>Sorted array</td>
<td>log N</td>
<td>N</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>Hashing</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

* assumes our hash function can generate random values for all keys
† if delete allowed, insert/search become sqrt(N) too

Ahead: Can we achieve log N delete?
Ahead: Can we achieve log N worst-case?

Right Rotate, Left Rotate

Fundamental operation to rearrange nodes in a tree.
- Maintains BST order.
- Local transformations, change just 3 pointers.

Recursive BST Root Insertion

Root insertion: insert a node and make it the new root.
- Insert the node using standard BST.
- Use rotations to bring it up to the root.

Why bother?
- Faster if searches are for recently inserted keys.
- Basis for advanced algorithms.
BST Construction: Root Insertion

**Observation.** If keys are inserted in random order then BST is balanced with high probability.

**Idea.** When inserting a new node, make it the root (via root insertion) with probability \(1/(N+1)\) and do it recursively.

**Fact.** Tree shape distribution is identical to tree shape of inserting keys in random order.
- No assumptions made on the input distribution!

```java
private Node insert(Node h, Comparable key, Object value) {
    if (h == null) return new Node(key, value);
    if (Math.random()*(h.N+1) < 1) return insertT(h, key, value);
    if (less(key, h.key))  h.left  = insert(h.left,  key, value); else h.right = insert(h.right, key, value);
    h.N++;
    return h;
}
```

Randomized BST Example

Insert keys in order.
- Tree shape still random.

Always "looks like" random binary tree.
- Implementation: maintain subtree size in each node.
- Supports all symbol table ops.
- \(\log N\) average case.
- Exponentially small chance of bad balance.
**Randomized BST: Delete**

Join. Merge two disjoint symbol tables A (of size M) and B (of size N), assuming all keys in A are less than all keys in B.
- Use A as root with probability $M / (M + N)$, and recursively join right subtree of A with B
- Use B as root with probability $N / (M + N)$, and recursively join left subtree of B with A

Delete. Given a key $k$, delete and return a node with key $k$.
- Delete the node.
- Join two broken subtrees as above.

Theorem. Tree still random after delete.

---

**Symbol Table: Implementations Cost Summary**

<table>
<thead>
<tr>
<th></th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td>Search</td>
<td>Insert</td>
</tr>
<tr>
<td>Sorted array</td>
<td>log N</td>
<td>N</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>Hashing</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Randomized BST</td>
<td>log N †</td>
<td>log N †</td>
</tr>
</tbody>
</table>

* assumes our hash function can generate random values for all keys
† if delete allowed, insert/search become sqrt(N)
‡ assumes system can generate random numbers

Randomized BST: guaranteed log N performance!
Next time: Can we achieve deterministic guarantee?

---

**BST: Other Operations**

Sort. Traverse tree in ascending order.
- Inorder traversal.
- Same comparisons as quicksort, but pay space for extra links.

Range search. Find all items whose keys are between $k_1$ and $k_2$.

Find $k^{th}$ largest. Generalized PQ that finds $k^{th}$ smallest.
- Special case: find min, find max.
- Add subtree size to each node.
- Takes time proportional to height of tree.

---

**Randomized BST: Other Operations**

Ceiling. Given key $k$, return smallest element that is at least as big as $k$.

Best-fit bin packing heuristic. Insert the item in the bin with the least remaining space among those that can store the item.

Theorem. Best-fit decreasing is guaranteed use no more than $11B/9 + 1$ bins, where B is the best possible.
- within 22% of best possible.
- original proof of this result was over 70 pages of analysis!
Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Find k&lt;sup&gt;th&lt;/sup&gt;</th>
<th>Sort</th>
<th>Join</th>
<th>Ceiling</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>
<td>log N</td>
</tr>
<tr>
<td>Unsorted list</td>
<td>N</td>
<td>1</td>
<td>1</td>
<td>N log N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Hashing</td>
<td>1*</td>
<td>1*</td>
<td>1*</td>
<td>N</td>
<td>N log N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>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>
<td>log N†</td>
</tr>
</tbody>
</table>

* assumes our hash function can generate random values for all keys
† assumes system can generate random numbers

makes no assumption on input distribution

**Randomized BST**: $O(\log N)$ average case for ALL ops!

**Next time**: Can we achieve deterministic guarantee?