4.3 Binary Search Trees

Binary search trees
Randomized BSTs

Symbol Table Challenges

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

  - Hashing analysis depends on input distribution

Challenge 2. Expand API when keys are ordered.
  - Find the kth largest


Binary Search Trees

Def. A binary search tree is a 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.

Binary Search Trees in Java

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 class Node {
  Key key;
  Val val;
  Node l, r;
}

Key and Val are generic types; Key is Comparable

Root 51 |
| 54 48 |
| 14 32 78 |
| smaller |
| larger |

A node has two subtrees, A and B, with smaller and larger values.
Java Implementation of BST: Skeleton

```
public class BST<Key extends Comparable, Val> {
    private Node root;

    private class Node {
        private Key key;
        private Val val;
        private Node l, r;

        private Node(Key key, Val val) {
            this.key = key;
            this.val = val;
        }

        private boolean less(Key k1, Key k2) { ... }
        private boolean eq (Key k1, Key k2) { ... }
        public void put(Key key, Val val) { ... }
        public Val get(Key key) { ... }
    }

    public void put(Key key, Val val) {
        root = insert(root, key, val);
    }

    private Node insert(Node x, Key key, Val val) {
        if (x == null) return new Node(key, val);
        else if (eq(key, x.key)) x.val = val;
        else if (less(key, x.key)) x.l = insert(x.l, key, val);
        else x.r = insert(x.r, key, val);
        return x;
    }

    public Val get(Key key) {
        Node x = root;
        while (x != null) {
            if (eq(key, x.key)) return x.val;
            else if (less(key, x.key)) x = x.l;
            else x = x.r;
        }
        return null;
    }
```

BST: Insert

- Associate val with key.
- Search, then insert.
- Concise (but tricky) recursive code.

BST: Construction

Insert the following keys into BST.

Search

Get. Return val corresponding to given key, or null if no such key.
Tree Shape

Tree shape.
- Many BSTs correspond to same input data.
- Cost of search/insert proportional to depth of node.
- 1-1 correspondence between BST and quicksort partitioning.

Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Get</th>
<th>Put</th>
<th>Remove</th>
<th>Get</th>
<th>Put</th>
<th>Remove</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/2</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>???</td>
</tr>
</tbody>
</table>

* assumes hash function is random

BST.  O(log N) insert and search if keys arrive in random order.

BST: Analysis

Theorem. If keys are inserted in random order, height of tree is Θ(log N), except with exponentially small probability.

- mean = 4.311 ln N, variance = O(1)

Property. If keys are inserted in random order, expected number of comparisons for a search/insert is about 2 ln N.

But... Worst-case for search/insert/height is N.

e.g., keys inserted in ascending order

BST: Eager Delete

Delete a node in a BST.  [Hibbard]
- Zero children:  just remove it.
- One child:  pass the child up.
- Two children:  find the next largest node using right-left* or left-right*, swap with next largest, remove as above.

Problem.  Eager deletion strategy clumsy, not symmetric.
Consequence.  Trees not random (!) ⇒ sqrt(N) per op.
**Right Rotate, Left Rotate**

**Two fundamental ops to rearrange nodes in a tree.**
- Maintains symmetric order.
- Local transformations, change just 3 pointers.

```java
private Node rotL(Node h) {   Node x = h.r;   h.r = x.l;   x.l = h;   return x;}
```

```java
private Node rotR(Node h) {   Node x = h.l;   h.l = x.r;   x.r = h;   return x;}
```

**BST: Lazy Delete**

**Lazy delete.** To delete node with a given key, set its value to `null`.

**Cost.** $O(\log N')$ per insert, search, and delete, where $N'$ is the number of elements ever inserted in the BST.

- Under random input assumption

**BST.** $O(\log N)$ insert and search if keys arrive in random order.

**Q.** Can we achieve $O(\log N)$ independent of input distribution?
**Randomized BST**

**Intuition.** If keys are inserted in random order, height is logarithmic.

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

**Ex:** Insert keys in ascending order.

Private Node insert(Node h, Key key, Val val) {
    if (h == null) return new Node(key, val);
    if (less(key, h.key)) {
        // return rootInsert(h, key, val);
        if (Math.random() > (h.N + 1) / h.N) {
            h.l = insert(h.l, key, val); // maintain size of subtree rooted at h
        } else if (less(key, h.key)) {
            h.l = insert(h.l, key, val);
        } else {
            h.r = insert(h.r, key, val);
        }
    }
    return h;
}

**Fact.** Tree shape distribution is identical to tree shape of inserting keys in random order.

**Recall.** Root insertion: insert a node and make it the new root.

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

**Ex:** A S E R C H I N G X M P L
Randomized BST

Property. Always "looks like" random binary tree.

- As before, expected height is $\Theta(\log N)$.
- Exponentially small chance of bad balance.

Implementation. Need to maintain subtree size in each node.

Randomized BST: Delete

Delete. Delete node containing given key; join two broken subtrees.

Randomized BST: Join

Join. Merge $T_1$ (of size $N_1$) and $T_2$ (of size $N_2$) assuming all keys in $T_1$ are less than all keys in $T_2$.
- Use root of $T_1$ as root with probability $N_1 / (N_1 + N_2)$, and recursively join right subtree of $T_1$ with $T_2$.
- Use root of $T_2$ as root with probability $N_2 / (N_1 + N_2)$, and recursively join left subtree of $T_2$ with $T_1$. 

Goal. Join $T_1$ and $T_2$, where all keys in $T_1$ are less than all keys in $T_2$. 

prob = 7/12
Randomized BST: Join

Join. Merge $T_1$ (of size $N_1$) and $T_2$ (of size $N_2$) assuming all keys in $T_1$ are less than all keys in $T_2$.
- Use root of $T_1$ as root with probability $N_1 / (N_1 + N_2)$, and recursively join right subtree of $T_1$ with $T_2$.
- Use root of $T_2$ as root with probability $N_2 / (N_1 + N_2)$, and recursively join left subtree of $T_2$ with $T_1$.

Randomized BST: Delete

Join. Merge $T_1$ (of size $N_1$) and $T_2$ (of size $N_2$) assuming all keys in $T_1$ are less than all keys in $T_2$.

Delete. Delete node containing given key; join two broken subtrees.

Analysis. Running time bounded by height of tree.

Theorem. Tree still random after delete.

Corollary. Expected number of comparisons for a search/insert/delete is $\Theta(\log N)$.

Symbol Table: Implementations Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Worst Case Search</th>
<th>Average Case Search</th>
<th>Delete</th>
<th>Average Case Insert</th>
<th>Worst Case Insert</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>
</tr>
<tr>
<td>Unsorted list</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N/2$</td>
<td>$N$</td>
</tr>
<tr>
<td>Hashing</td>
<td>$N$</td>
<td>$1$</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>
</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>
</tr>
</tbody>
</table>

* Assumes our hash function can generate random values for all keys
† Assumes $m$ is the number of keys ever inserted
‡ Assumes system can generate random numbers, randomized guarantee

Randomized BST. Guaranteed log $N$ performance!

Next lecture. Can we achieve deterministic guarantee?
**BST: Bin Packing Application**

*Ceiling.* Given key $k$, return smallest element that is $\geq k$.

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

**Theorem.** [D. Johnson] Best-fit decreasing is guaranteed use at most $11B/9 + 1$ 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 $k^{th}$</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>N</td>
<td>N</td>
<td>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>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, randomized guarantee