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 interface when keys are ordered.
- find the kth largest

Binary Search Trees

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 class Node {
    Key k;
    Value v;
    Node l, r;
}

Binary Search Trees in Java
BST: Tree Shape

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

BST: Skeleton

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

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

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

    private static boolean less(Key k1, Key k2) { }
    private static boolean eq (Key k1, Key k2) { }

    public void put(Key key, Value val) { }
    public Value get(Key key) { }
}
```

BST: Search

Get: return value corresponding to given key, or null if no such key.

```java
public Value 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

Put: associate value with key.
- Search, then insert.
- Concise (but tricky) recursive code.

```java
public void put(Key key, Value val) {
    root = insert(root, key, val);
}
```

```java
private Node insert(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val);
    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;
}
```
BST: Construction

Insert the following keys into BST: `ASERCHINGXMPLE`

BST: Analysis

Cost of search and insert BST.
- Proportional to depth of node.
- 1-1 correspondence between BST and quicksort partitioning.

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

Problem. Worst-case put and get is \( \Theta(N) \).

BST: Eager Delete

To delete a node in a BST.
- Case 0 [zero children]: just remove it.
- Case 1 [one child]: pass the child up.
- Case 2 [two children]: find the next largest node using right-left\(^*\) or left-right\(^*\), swap with next largest, remove as in Case 0 or 1.

Problem. Strategy clumsy, not symmetric.

Consequence. Trees not random (!!) \( \Rightarrow \sqrt{N} \) per op.
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.

\[
\begin{array}{l}
\text{Sorted array} & \text{Log N} & N & N \quad \text{Log N} & N/2 & N/2 \\
\text{Unsorted list} & N & N & N/2 & N & N \\
\text{Hashing} & N & 1 & N \quad \text{1*} & 1* & 1* \\
\text{BST} & N & N & N \quad \text{Log N}^* & \text{Log N}^* & \text{Log N}^* \\
\end{array}
\]

* assumes our hash function can generate random values for all keys
\^ assumes \(N\) is number of keys ever inserted

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

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

Symbol Table: Implementations Cost Summary

Right Rotate, Left Rotate

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

Rotation. Fundamental operation to rearrange nodes in a tree.
- Easier done than said.

\[
\begin{array}{l}
\text{private Node rotL(Node h)} \{ \\
\quad \text{Node x = h.r; } \\
\quad \text{h.r = x.l; } \\
\quad \text{x.l = h; } \\
\quad \text{return x; } \\
\} \\
\end{array}
\]

\[
\begin{array}{l}
\text{private Node rotR(Node h)} \{ \\
\quad \text{Node x = h.l; } \\
\quad \text{h.l = x.r; } \\
\quad \text{x.r = h; } \\
\quad \text{return x; } \\
\} \\
\end{array}
\]
Randomized BST

Recall. 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 so recursively.

Fact. Tree shape distribution is identical to tree shape of inserting keys in random order, but now, no assumption made on the input distribution.

Recursive BST Root Insertion

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

Why bother?
- Faster if searches are for recently inserted keys.
- Basis for advanced algorithms.

```java
private Node rootInsert(Node h, Key key, Value val) {
    if (h == null) return new Node(key, val);
    if (less(key, h.key)) {
        h.l = rootInsert(h.l, key, val);
        h = rotR(h);
    } else {
        h.r = rootInsert(h.r, key, val);
        h = rotL(h);
    }
    return h;
}
```

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

Ex: Insert keys in ascending order.

BST Construction: Root Insertion

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

Randomized BST Example

Ex: Insert keys in ascending order.
Randomized BST

**Property.** Always "looks like" random binary tree.

- $\Theta(\log N)$ average case.
- Implementation: maintain subtree size in each node.
- Exponentially small chance of bad balance.

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</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>$\log N$</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>
</tbody>
</table>

*$^*$ assumes our hash function can generate random values for all keys

$^\dagger$ assumes $N$ is the number of keys ever inserted

$^\ddagger$ assumes system can generate random numbers, randomized guarantee

Randomized BST. Guaranteed log N performance!

**Ahead:** Can we achieve deterministic guarantee?

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

**Theorem.** Tree still random after delete.

BST: Advanced Operations

**Sort.** Iterate over keys 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 kth largest/smallest.** Generalized PQ.
- Special case: find min, find max.
- Add subtree size to each node.
- Takes time proportional to height of tree.

```
private class Node {
  Key key;
  Value val;
  Node l, r;
  int N;
}
```
BST: Bin Packing Application

Ceiling. Given key k, return smallest element that is ≥ 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 kth</th>
<th>Sort</th>
<th>Join</th>
<th>Ceil</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 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 log N</td>
<td>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