4.3 Binary Search Trees

Binary search trees
Randomized BSTs

Reference: Chapter 12, Algorithms in Java, 3rd Edition, Robert Sedgewick.

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.

Challenge 1. Guarantee symbol table performance.
↖ hashing analysis depends on input distribution

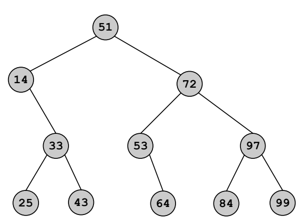
Challenge 2. Expand API when keys are ordered.
↖ find the kth largest

Binary Search Trees

Def. A binary search tree is a binary tree in symmetric order.

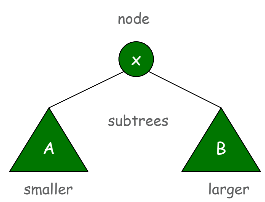
Binary tree is either:

- 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.

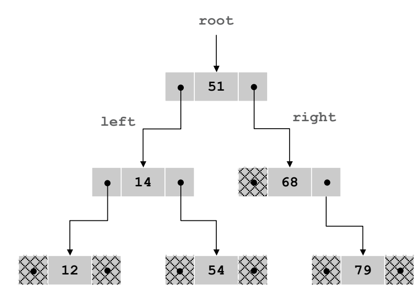
A Node is comprised of four fields:

- A key and a value.
- A reference to the left and right subtree.

(left → smaller, right → larger)

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

Key and Val are generic types;
Key is Comparable



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) { … }
}
```


Search

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

```
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

Put. Associate `val` with `key`.

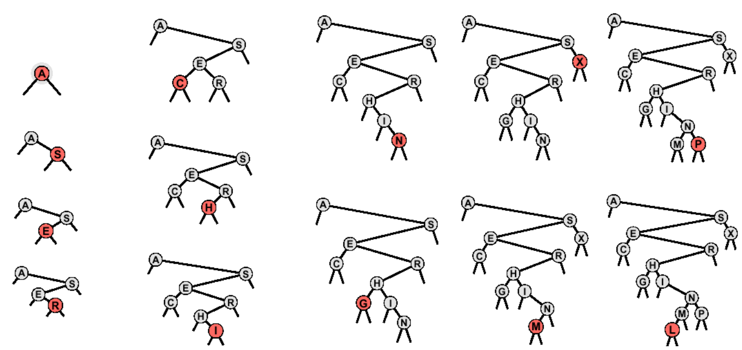
- Search, then insert.
- Concise (but tricky) recursive code.

```
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;
}
```


BST: Construction

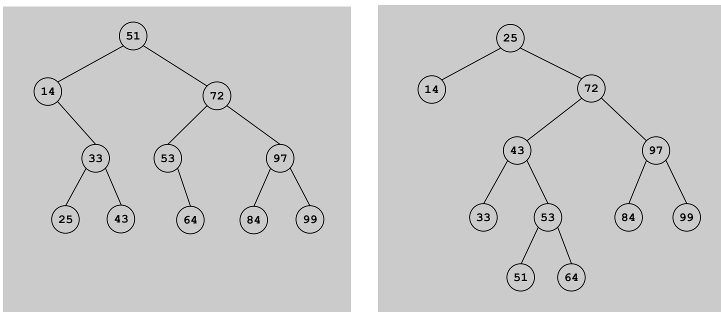
Insert the following keys into BST. **A S E R C H I N G X M P L**



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.

depth of node corresponds to
depth of function call stack when node is partitioned



BST: Analysis

Theorem. If keys are inserted in random order, height of tree is $\Theta(\log N)$, 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

Symbol Table: Implementations Cost Summary

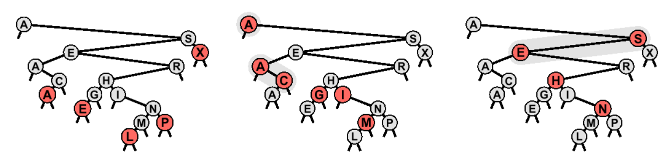
	Worst Case			Average Case		
Implementation	Get	Put	Remove	Get	Put	Remove
Sorted array	log N	N	N	log N	N/2	N/2
Unsorted list	N	N	N	N/2	N	N
Hashing	N	1	N	1*	1*	1*
BST	N	N	N	log N	log N	???

* assumes hash function is random

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



zero children   one child   two children

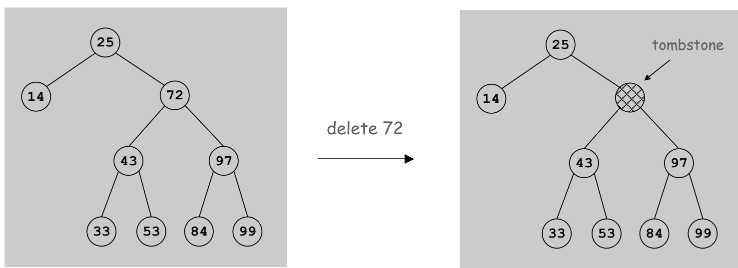
Problem. Eager deletion strategy clumsy, not symmetric.
Consequence. Trees not random (!) ⇒ sqrt(N) per op.

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.

under random input assumption



Symbol Table: Implementations Cost Summary

	Worst Case			Average Case		
Implementation	Get	Put	Remove	Get	Put	Remove
Sorted array	log N	N	N	log N	N/2	N/2
Unsorted list	N	N	N	N/2	N	N
Hashing	N	1	N	1*	1*	1*
BST	N	N	N	log N †	log N †	log N †

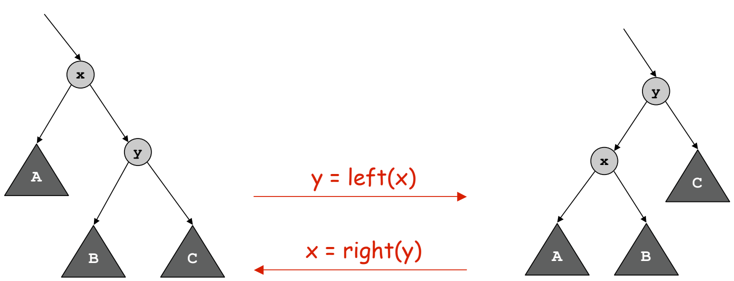
* assumes hash function is random
† assumes N is number of keys ever inserted

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

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

Right Rotate, Left Rotate

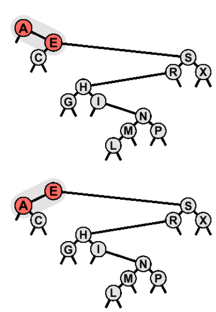
Two fundamental ops to rearrange nodes in a tree.
- Maintains symmetric order.
- Local transformations, change just 3 pointers.



Right Rotate, Left Rotate

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

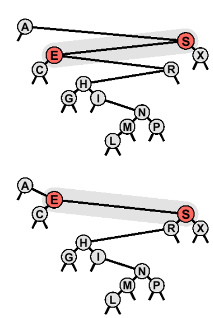
left rotate 'A'



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

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

right rotate 'S'



Recursive BST Root Insertion

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

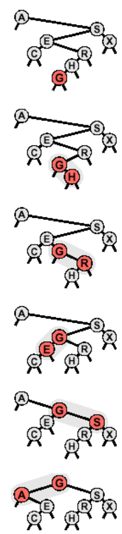
- Insert using standard BST.
- Rotate it up to the root.

Why bother?

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

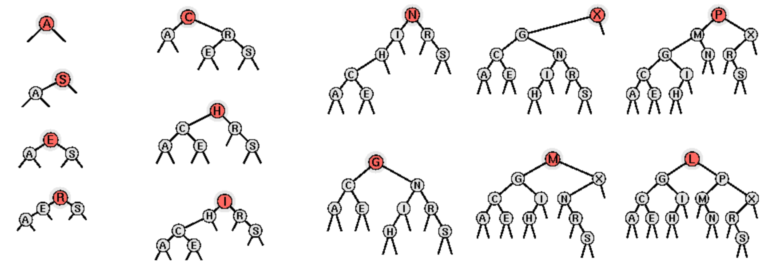
```
private Node rootInsert(Node h, Key key, Val val) {
   if (h == null) return new Node(key, val);
   if (less(key, h.key)) {
      h.l = rootInsert(h.l, key, val);
      h = rotR(h);
   }
   else {
      h.r = rootInsert(h.r, key, val);
      h = rotL(h);
   }
   return h;
}
```

insert G



BST Construction: Root Insertion

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



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/(N+1), and do so recursively.

```
private Node insert(Node h, Key key, Val val) {
   if (h == null) return new Node(key, val);
   if (Math.random()*(h.N + 1) < 1)
      return rootInsert(h, key, val);
   else if (less(key, h.key))   h.l = insert(h.l, key, val);
   else                         h.r = insert(h.r, key, val);
   h.N++;
   return h;
}
```

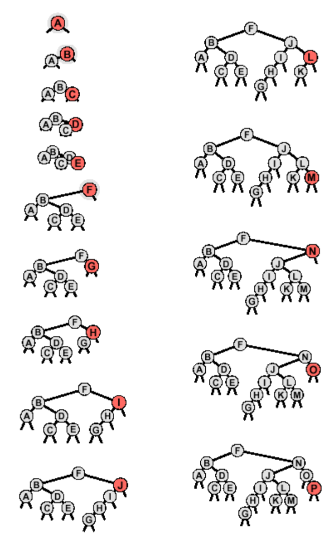
maintain size of subtree rooted at h

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

but now, no assumption made on the input distribution

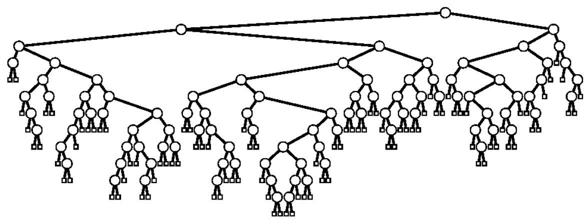
Randomized BST Example

Ex: Insert keys in ascending order.



Randomized BST

Property. Always "looks like" random binary tree.

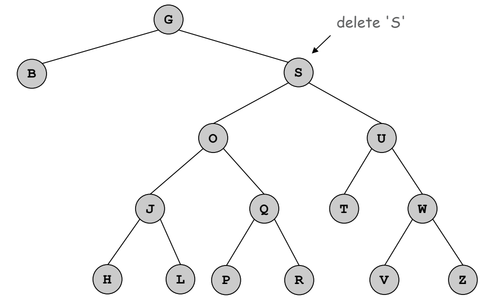


- As before, expected height is $\Theta(\log N)$.
- Exponentially small chance of bad balance.

Implementation. Need to maintain subtree size in each node.

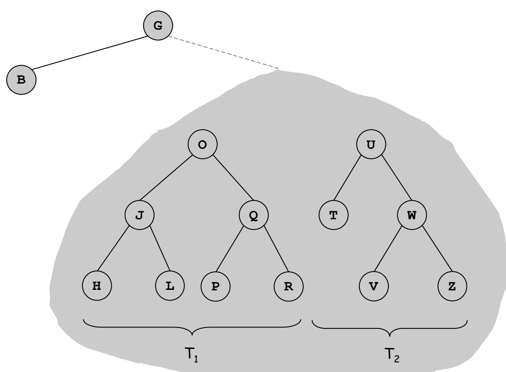
Randomized BST: Delete

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



Randomized BST: Delete

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

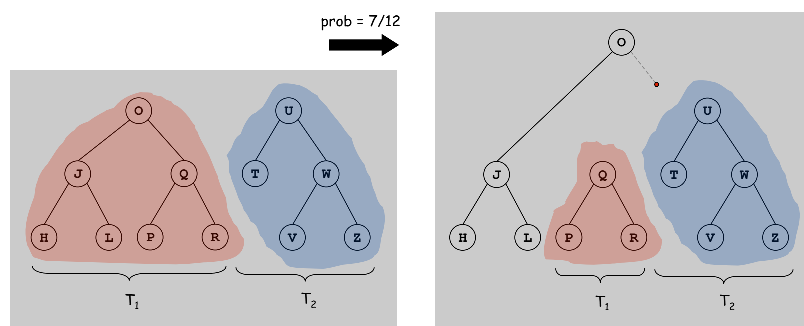


Goal. Join T_1 and T_2, where all keys in T_1 are less than all keys in T_2.

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.

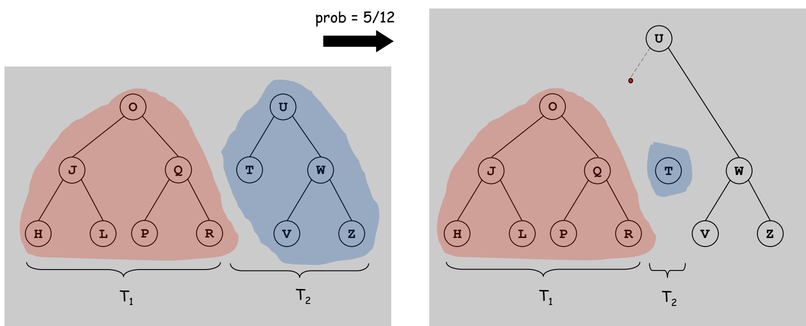
- Use root of T_1 as root with probability $N_1 / (N_1 + N_2)$, and recursively join right subtree of T_1 with T_2.
- Use root of T_2 as root with probability $N_2 / (N_1 + N_2)$, and recursively join left subtree of T_2 with T_1.



Randomized BST: 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

	Worst Case			Average Case		
Implementation	Search	Insert	Delete	Search	Insert	Delete
Sorted array	log N	N	N	log N	N/2	N/2
Unsorted list	N	N	N	N/2	N	N
Hashing	N	1	N	1*	1*	1*
BST	N	N	N	log N †	log N †	log N †
Randomized BST	log N ‡	log N ‡	log N ‡	log N	log N	log N

* assumes our hash function can generate random values for all keys
† assumes N 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: Advanced Operations

Sort. Iterate over keys in ascending order.

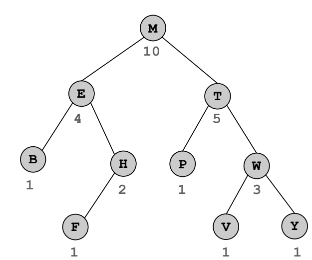
- Inorder traversal.
- Same comparisons as quicksort, but pay space for extra links.

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

Find k^{th} largest/smallest. Generalizes 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;
    Val val;
    Node l, r;
    int N;
}
```
subtree size



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

Asymptotic Cost

Implementation	Search	Insert	Delete	Find k^{th}	Sort	Join	Ceil
Sorted array	log N	N	N	log N	N	N	log N
Unsorted list	N	N	N	N	N log N	N	N
Hashing	1*	1*	1*	N	N log N	N	N
BST	N	N	N	N	N	N	N
Randomized BST	log N ‡	log N ‡	log N ‡	log N ‡	N	log N ‡	log N ‡

* assumes our hash function can generate random values for all keys
‡ assumes system can generate random numbers, randomized guarantee