4.3 Binary Search Trees

Binary search trees
Randomized BSTs

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

Symbol Table Challenges

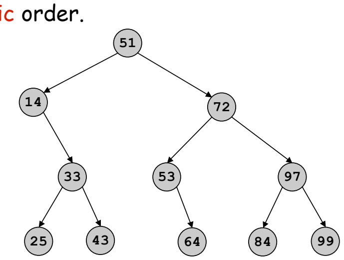
Symbol table: key-value pair abstraction.
- Insert a value with specified key.
- Search for value given key.
- Delete value with given key.

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

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

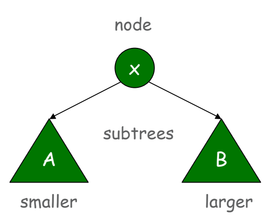
Binary Search Trees

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.



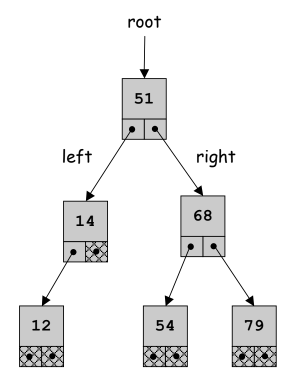
Binary Search Trees in Java

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.

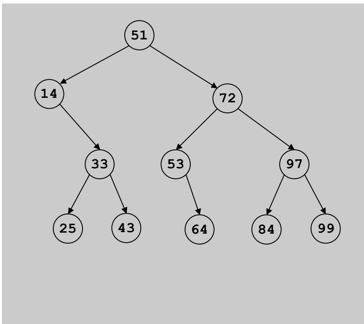
$\underbrace{\text{left}}_{\text{smaller}}$ $\underbrace{\text{right}}_{\text{larger}}$

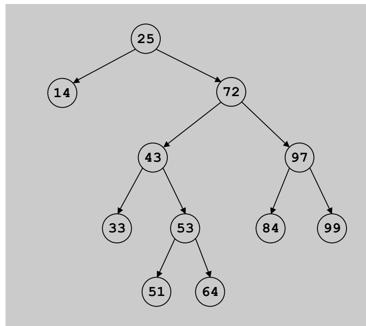
```
private class Node {
   Key   key;     ← key is a generic Comparable object
   Value val;
   Node  l, r;
}
```



BST: Tree Shape

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





BST: Skeleton

```
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.

```
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

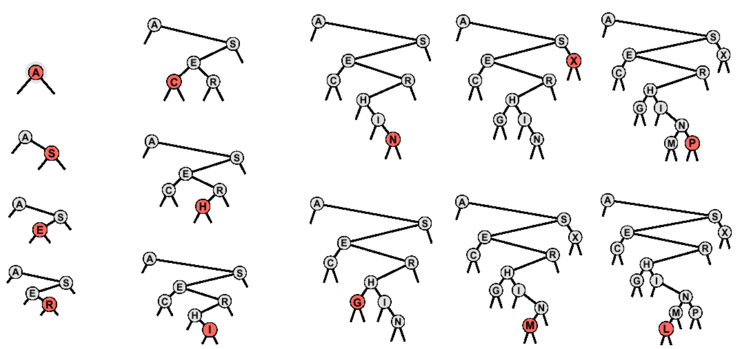
Put: associate value with key.
- Search, then insert.
- Concise (but tricky) recursive code.

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

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: **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.

depth of node corresponds to
depth of function call 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, put and get take $O(\log N)$ time.

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

nodes inserted in ascending or descending order

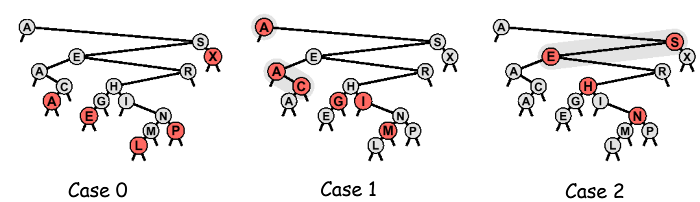
Symbol Table: Implementations Cost Summary

	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	???

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

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.



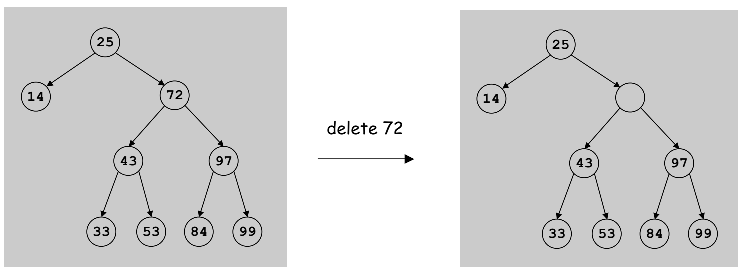
Problem. Strategy clumsy, not symmetric.
Consequence. Trees not random (!!) $\Rightarrow$ sqrt(N) per op.

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.

under random input assumption



Symbol Table: Implementations Cost Summary

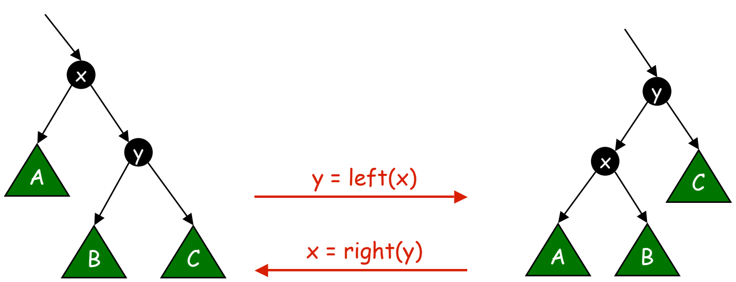
	Worst Case			Average Case		
Implementation	Search	Insert	Delete	Search	Insert	Delete
Sorted array	log N	N	N	log N	N / 2	N / 2
Unsorted list	N	N	N	N / 2	N	N
Hashing	N	1	N	1*	1*	1*
BST	N	N	N	log N †	log N †	log N †

* 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?

Right Rotate, Left Rotate

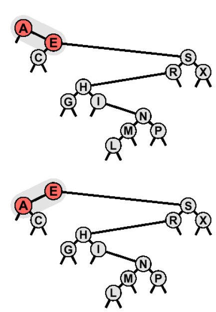
Fundamental operation to rearrange nodes in a tree.
- Maintains BST order.
- Local transformations, change just 3 pointers.



Right Rotate, Left Rotate

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

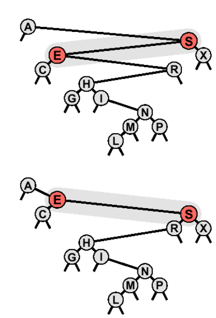
left rotate A



```
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



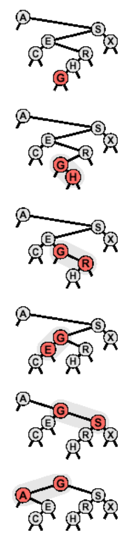
Recursive BST Root Insertion

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.

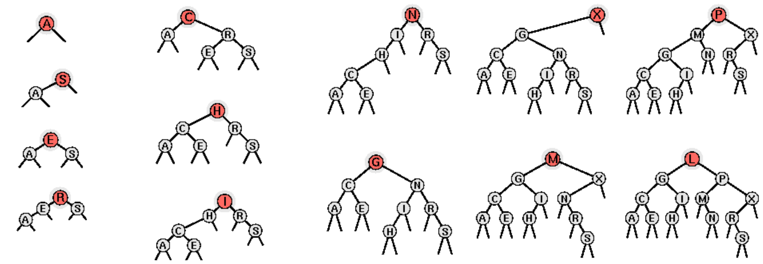
```
private Node rootInsert(Node h, Key key, Value val) {
   if (h == null) return new Node(key, val);
   if (less(key, h.key)) {
      h.l = rootInsert(h.l, key, val);
      h = rotR(h);
   }
   else {
      h.r = rootInsert(h.r, key, val);
      h = rotL(h);
   }
   return h;
}
```

insert G



BST Construction: Root Insertion

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



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.

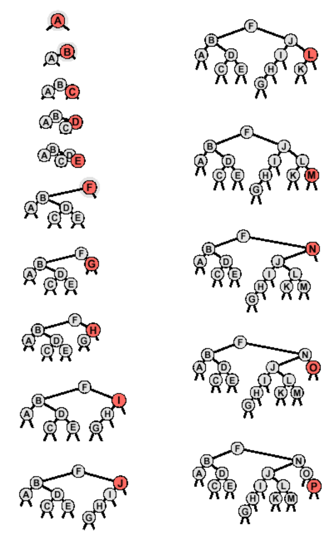
```
private Node insert(Node h, Key key, Value val) {
   if (h == null) return new Node(key, val);
   if (Math.random()*(h.N + 1) < 1)
      return rootInsert(h, key, val);
   if (less(key, h.key))  h.l = insert(h.l, key, val);
   else                   h.r = insert(h.r, key, val);
   h.N++;
   return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

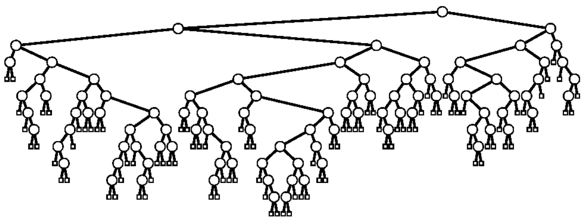
Randomized BST Example

Ex: Insert keys in ascending order.



Randomized BST

Property. Always "looks like" random binary tree.



- Θ(log N) average case.
- Implementation: maintain subtree size in each node.
- 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. Delete node containing given key; join two broken subtrees.

Theorem. Tree still random after delete.

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!
Ahead: 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.

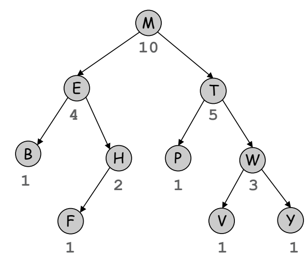
Range search. Find all items whose keys are between k_1 and k_2.

Find k^{th} largest/smallest. 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;
}
```

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