Binary Search Trees

- basic implementations
- randomized BSTs
- deletion in BSTs

References:
Algorithms in Java, Chapter 12
Intro to Programming, Section 4.4
`http://www.cs.princeton.edu/algs4/42bst`

Elementary implementations: summary

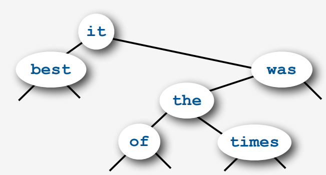
implementation	worst case		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
unordered array	N	N	N/2	N	no	`equals()`
unordered list	N	N	N/2	N	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
ordered list	N	N	N/2	N/2	yes	`compareTo()`

Challenge. Efficient implementations of search and insert.

- binary search tree
- randomized BSTs
- deletion in BSTs

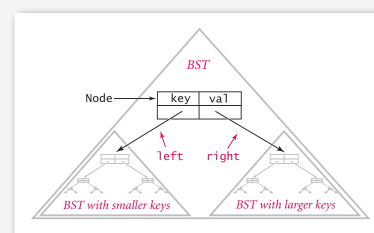
Binary search trees

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



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

Symmetric order. Every node's key is:
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



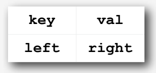
BST representation

A BST is a reference to a root node.

A `Node` is comprised of four fields:

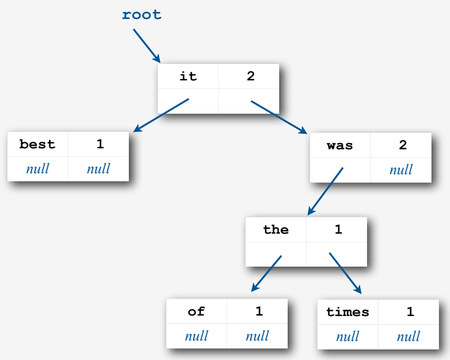
- A `Key` and a `Value`.
- A reference to the left and right subtree.

smaller keys　　larger keys




```
private class Node
{
   private Key key;
   private Value val;
   private Node left;
   private Node right;
}
```

`Key` and `Value` are generic types; `Key` is `Comparable`



BST implementation (skeleton)

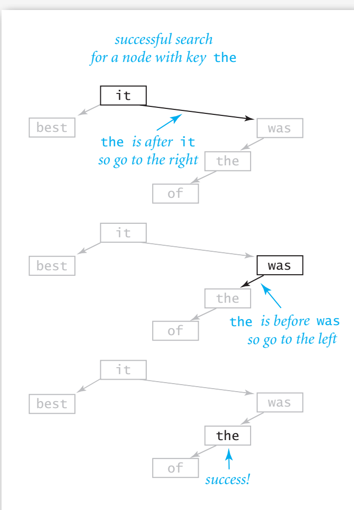
```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root;

   private class Node
   {
      private Key key;
      private Value val;
      private Node left, right;
      public Node(Key key, Value val)
      {
         this.key = key;
         this.val = val;
      }
   }

   public void put(Key key, Value val)
   {  /* see next slides */  }

   public Val get(Key key)
   {  /* see next slides */  }
}
```

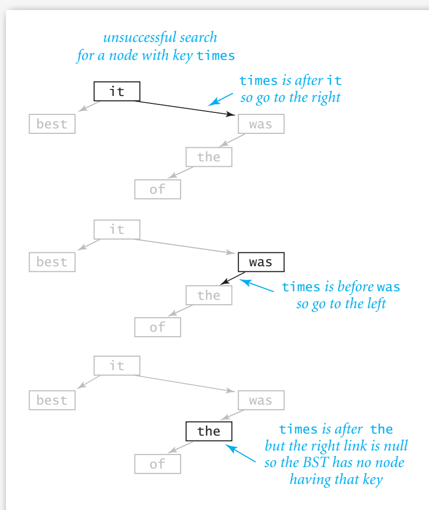
instance variable

instance variable

BST search

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





BST search: Java implementation

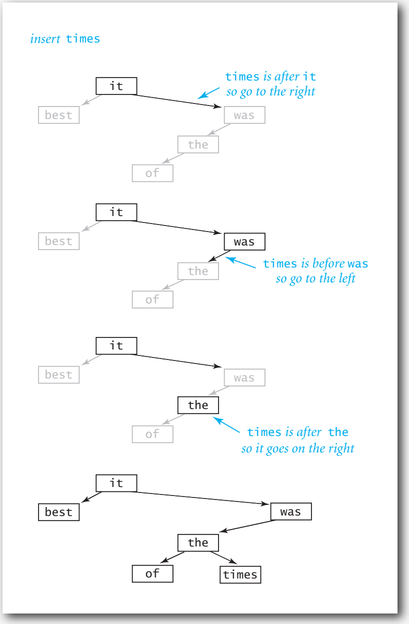
Get. Return value corresponding to given key, or `null` if no such key.

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

Running time. Proportional to depth of node.

BST insert

Put. Associate value with key.



BST insert: Java implementation

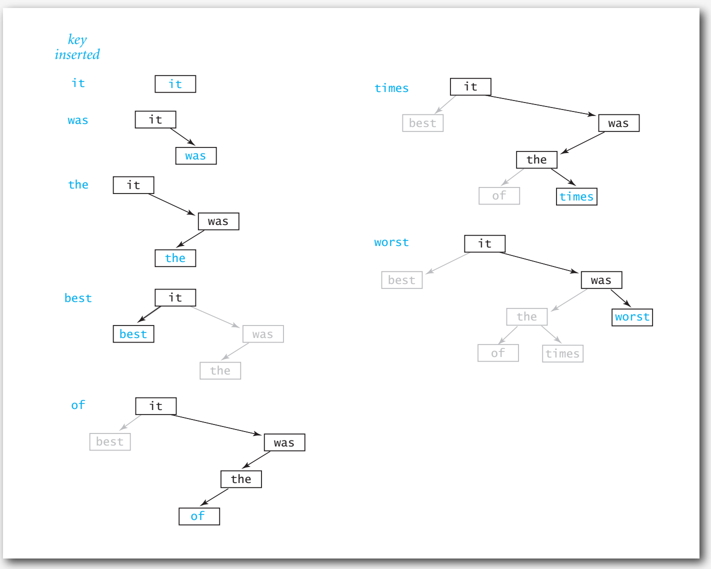
Put. Associate value with key.

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

private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val;
   return x;
}
```

concise, but tricky, recursive code; read carefully!

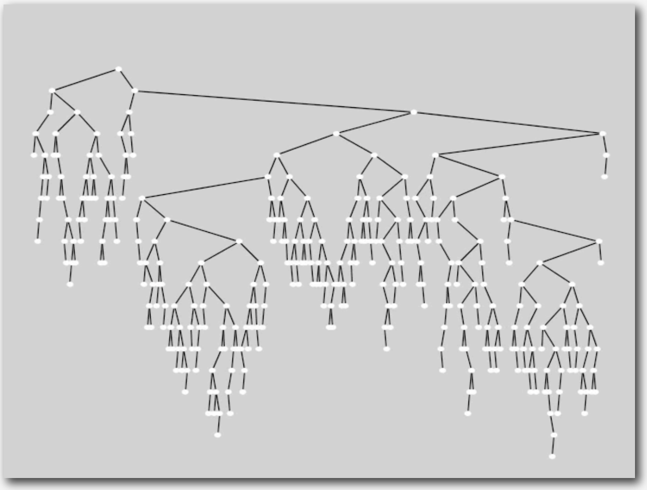
Running time. Proportional to depth of node.

BST construction example

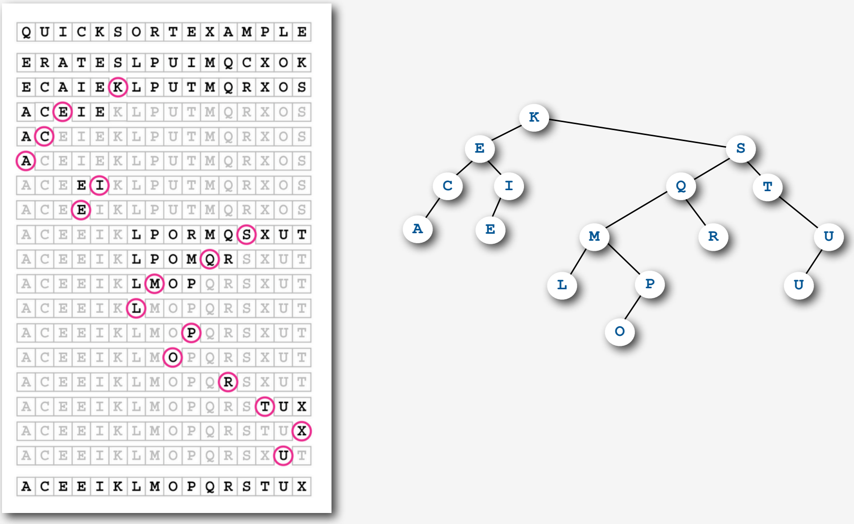


BST insertion: visualization

Ex. Insert keys in random order.



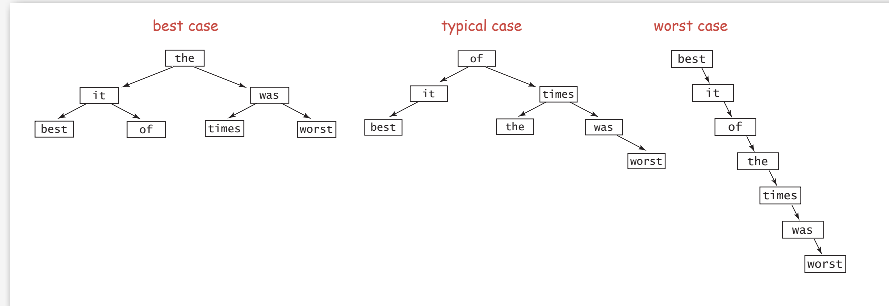
Correspondence between BSTs and quicksort partitioning



Remark. Correspondence is 1-1 if no duplicate keys.

Tree shape

- Many BSTs correspond to same input data.
- Cost of search/insert is proportional to depth of node.



Remark. Tree shape depends on order of insertion.

BSTs: mathematical analysis

Proposition. If keys are inserted in random order, the expected number of compares for a search/insert is ~ 2 ln N.

Pf. 1-1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If keys are inserted in random order, expected height of tree is ~ 4.311 ln N – 1.953 ln ln N.

But... Worst-case for search/insert/height is N (but occurs with exponentially small chance when keys are inserted in random order).

ST implementations: summary

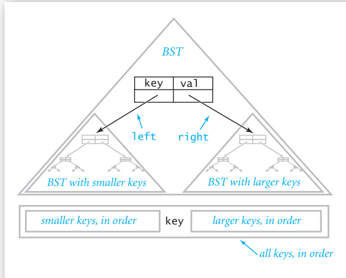
implementation	guarantee		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
unordered array	N	N	N/2	N	no	`equals()`
unordered list	N	N	N/2	N	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
ordered list	N	N	N/2	N/2	yes	`compareTo()`
BST	N	N	1.38 lg N	1.38 lg N	?	`compareTo()`

Next challenge. Ordered iteration.

Inorder traversal

Traversing the tree inorder yields keys in ascending order.

```
public void show()
{  return show(root);  }

private void show(Node x)
{
   if (x == null) return;
   show(x.left);
   StdOut.println(x.key + " " + x.val);
   show(x.right);
}
```

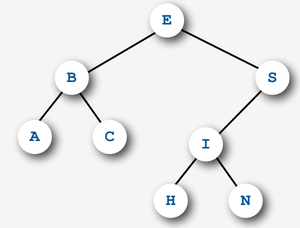


To implement an iterator: need a non-recursive version.

Non-recursive inorder traversal

To process a node:
- Follow left links until empty (pushing onto stack).
- Pop and process.
- Follow right link (push onto stack).

recursive calls	output	stack contents
visit(E)		E
visit(B)		E B
visit(A)		E B A
print A	A	E B
print B	B	E
visit(C)		E C
print C	C	E
print E	E	-
visit(S)		S
visit(I)		S I
visit(H)		S I H
print H	H	S I
print I	I	S
visit(N)		S N
print N	N	S
print S	S	-



Inorder iteartor: Java implementation

```
public Iterator<Key> iterator()
{  return new Inorder();  }

private class Inorder implements Iterator<Key>
{
   private Stack<Node> stack = new Stack<Node>();

   private void pushLeft(Node x)
   {
      while (x != null)
      {  stack.push(x); x = x.left;  }
   }

   BSTIterator()
   {  pushLeft(root);  }

   public boolean hasNext()
   {  return !stack.isEmpty();  }

   public Key next()
   {
      Node x = stack.pop();
      pushLeft(x.right);
      return x.key;
   }
}
```

go down left spine and push all keys onto stack

ST implementations: summary

implementation	guarantee		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
unordered array	N	N	N/2	N	no	`equals()`
unordered list	N	N	N/2	N	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
ordered list	N	N	N/2	N/2	yes	`compareTo()`
BST	N	N	1.38 lg N	1.38 lg N	yes	`compareTo()`

Next challenge. Guaranteed efficiency for search and insert.

Searching challenge 3 (revisited):

Problem. Frequency counts in "Tale of Two Cities"

Assumptions. Book has 135,000+ words; about 10,000 distinct words.

Which searching method to use?

1) Unordered array.
2) Unordered linked list
3) Ordered array with binary search.
4) Need better method, all too slow.
5) Doesn't matter much, all fast enough.
6) BSTs.

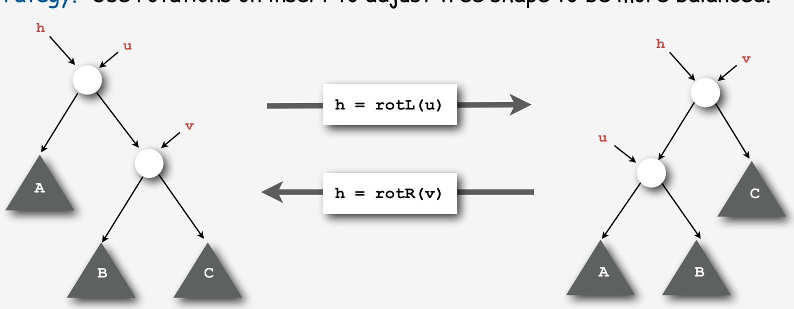
insertion cost < 10000 * 1.38 * lg 10000 < .2 million
lookup cost < 135000 * 1.38 * lg 10000 < 2.5 million

Rotation in BSTs

Two fundamental operations to rearrange nodes in a tree.

- Maintain symmetric order.
- Local transformations (change just 3 pointers).
- Basis for advanced BST algorithms.

Strategy. Use rotations on insert to adjust tree shape to be more balanced.



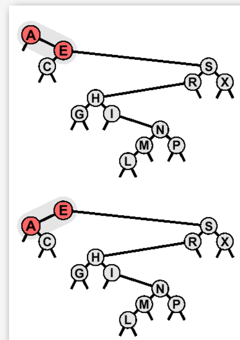
Key point. No change to BST search code (!)

Rotation in BSTs

Two fundamental operations to rearrange nodes in a tree.

- Easier done than said.
- Raise some nodes, lowers some others.

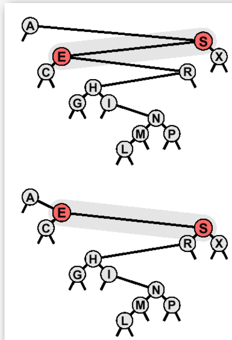
`root = rotL(A)`




```
private Node rotL(Node h)
{
   Node v = h.right;
   h.right = v.left;
   v.left = h;
   return v;
}
```

```
private Node rotR(Node h)
{
   Node u = h.left;
   h.left = u.right;
   u.right = h;
   return u;
}
```

`a.right = rotR(S)`



BST root insertion

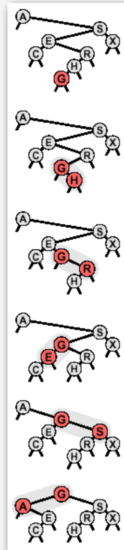
Insert a node and make it the new root.

- Insert node at bottom, as in standard BST.
- Rotate inserted node to the root.
- Compact recursive implementation.

```
private Node putRoot(Node x, Key key, Val val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if (cmp < 0)
   {
      x.left = putRoot(x.left,  key, val);
      x = rotR(x);
   }
   else if (cmp > 0)
   {
      x.right = putRoot(x.right, key, val);
      x = rotL(x);
   }
   else if (cmp == 0) x.val = val;
   return x;
}
```

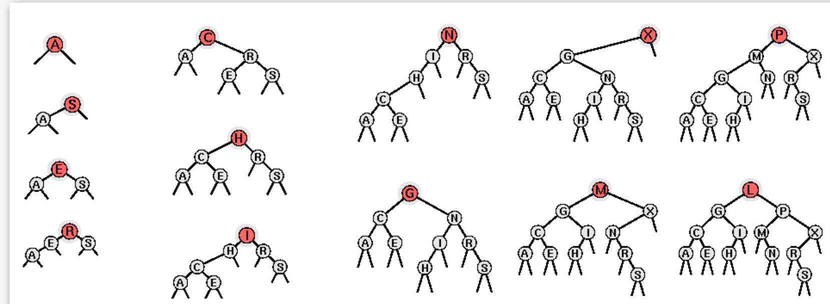
tricky recursive code; read very carefully!

insert G



BST root insertion: construction

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



Why bother?

- Recently inserted keys are near the top (better for some clients).
- Basis for randomized BST.

Randomized BSTs (Roura, 1996)

Intuition. If tree is random, height is logarithmic.
Fact. Each node in a random tree is equally likely to be the root.

Idea. Since new node should be the root with probability 1/(N+1), make it the root (via root insertion) with probability 1/(N+1).

and apply idea recursively

```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);

   if (cmp == 0) { x.val = val; return x; }

   if (StdRandom.bernoulli(1.0 / (x.N + 1.0))
      return putRoot(h, key, val);

   if      (cmp < 0) x.left  = put(x.left,  key, val);
   else if (cmp > 0) x.right = put(x.right, key, val);

   x.N++;
   return x;
}
```

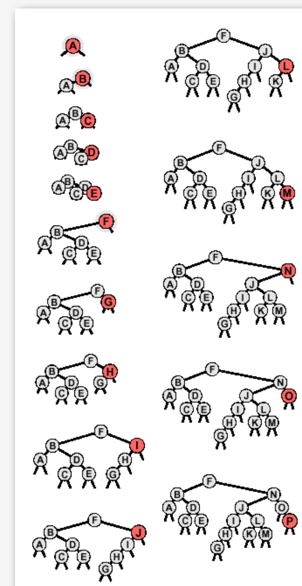
no rotations if key in table

root insert with the right probability

maintain count of nodes in subtree rooted at x

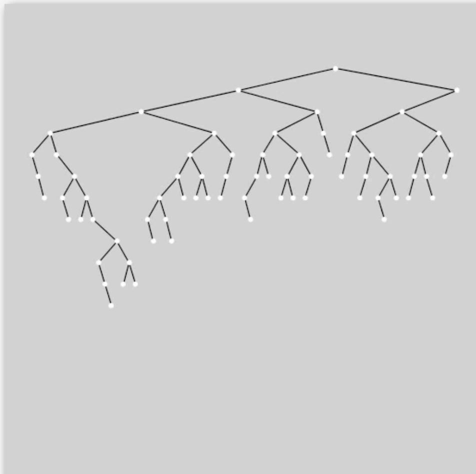
Randomized BST: construction

Ex. Insert 15 keys in ascending order.



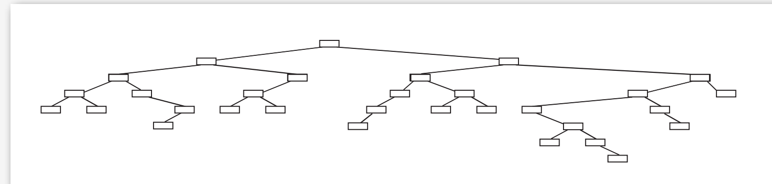
Randomized BST construction: visualization

Ex. Insert 500 keys in random order.



Randomized BST: analysis

Proposition. Randomized BSTs have the same distribution as BSTs under random insertion order, no matter in what order keys are inserted.



- Expected height is ~ 4.31107 ln N.
- Average search cost is ~ 2 ln N.
- Exponentially small chance of bad balance.

Implementation cost. Need to maintain subtree size in each node.

ST implementations: summary

implementation	guarantee		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
unordered array	N	N	N/2	N	no	`equals()`
unordered list	N	N	N/2	N	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
ordered list	N	N	N/2	N/2	yes	`compareTo()`
BST	N	N	1.38 lg N	1.38 lg N	yes	`compareTo()`
randomized BST	3 lg N	3 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`

Bottom line. Randomized BSTs provide the desired guarantee.

probabilistic, with exponentially small chance of linear time

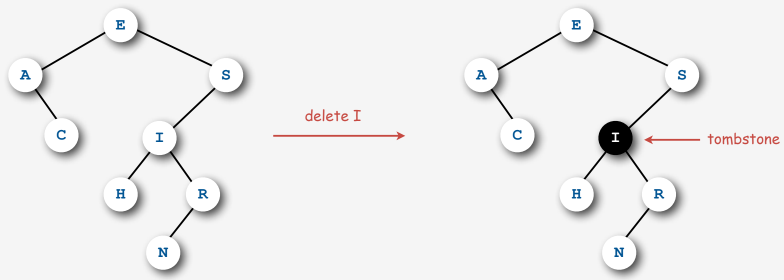
Bonus. Randomized BSTs also support delete (!)

- basic implementations
- randomized BSTs
- **deletion in BSTs**

BST deletion: lazy approach

To remove a node with a given key:

- Set its value to `null`.
- Leave key in tree to guide searches (but don't consider it equal to search key).



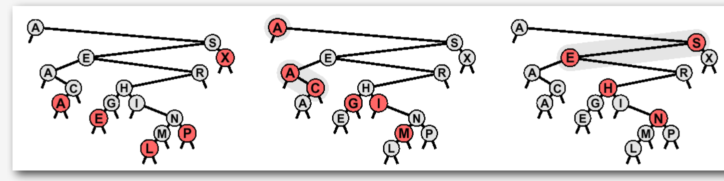
Cost. O(log N') per insert, search, and delete, where N' is the number of elements ever inserted in the BST.

Unsatisfactory solution. Tombstone overload.

BST deletion: Hibbard deletion

To remove a node from a BST:

- Zero children: just remove it.
- One child: pass the child up.
- Two children: find the next largest node using right-left*, swap with next largest, remove as above.

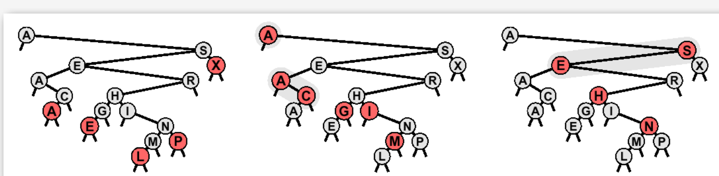


zero children one child two children

BST deletion: Hibbard deletion

To remove a node from a BST:

- Zero children: just remove it.
- One child: pass the child up.
- Two children: find the next largest node using right-left*, swap with next largest, remove as above.



zero children one child two children

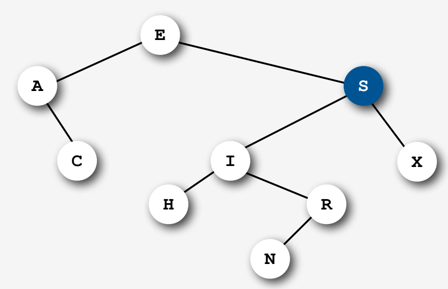
Unsatisfactory solution. Not symmetric, code is clumsy.
Surprising consequence. Trees not random (!) $\Rightarrow$ sqrt(N) per op.
Longstanding open problem. Simple and efficient delete for BSTs.

Randomized BST deletion

To delete a node containing a given key:

- Find the node containing the key.
- Remove the node.
- Join its two subtrees to make a tree.

Ex. Delete S.



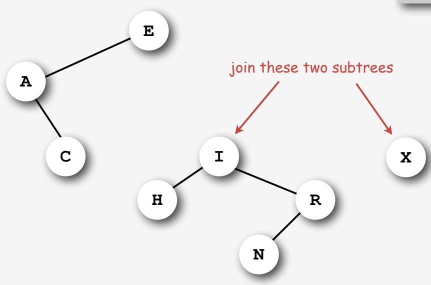
Randomized BST deletion

To delete a node containing a given key:

- Find the node containing the key.
- Remove the node.
- Join its two subtrees to make a tree.

Ex. Delete S.

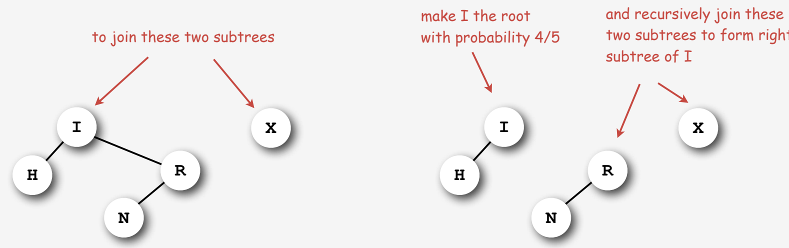
```
private Node remove(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if (cmp < 0)
      x.left  = remove(x.left,  key);
   else if (cmp > 0)
      x.right = remove(x.right, key);
   else if (cmp == 0)
      return join(x.left, x.right);
   return x;
}
```



Randomized BST join

To join two subtrees with all keys in one less than all keys in the other:

- Maintain counts of nodes in subtrees a and b.
- With probability |a|/(|a|+|b|):
  - root = root of a
  - left subtree = left subtree of a
  - right subtree = join b and right subtree of a
- With probability |a|/(|a|+|b|) do the symmetric operations.

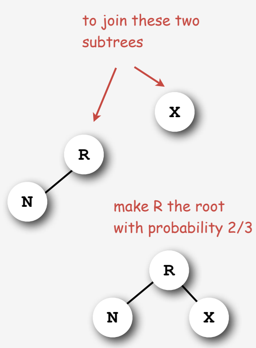


Randomized BST join

To join two subtrees with all keys in one less than all keys in the other:

- Maintain counts of nodes in subtrees a and b.
- With probability |a|/(|a|+|b|):
  - root = root of a
  - left subtree = left subtree of a
  - right subtree = join b and right subtree of a
- With probability |a|/(|a|+|b|) do the symmetric operations.

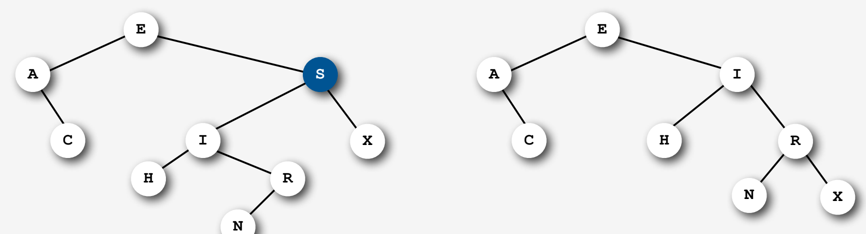
```
private Node join(Node a, Node b)
{
   if (a == null) return b;
   if (b == null) return a;
   if (StdRandom.bernoulli((double) a.N / (a.N + b.N))
   {  a.right = join(a.right, b); return a;  }
   else
   {  b.left  = join(a, b.left ); return b;  }
}
```



Randomized BST deletion

To delete a node containing a given key:

- Find the node containing the key.
- Remove the node.
- Join its two subtrees to make a tree.



Proposition. Tree still random after delete (!).

Bottom line. Logarithmic guarantee for search/insert/delete.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
unordered array	N	N	N	N/2	N	N/2	no	`equals()`
unordered list	N	N	N	N/2	N	N/2	no	`equals()`
ordered array	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
ordered list	N	N	N	N/2	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
randomized BST	3 lg N	3 lg N	3 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`

Bottom line. Randomized BSTs provide the desired guarantee.

Next lecture. Can we do better?

probabilistic, with exponentially small chance of linear time