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/introalgsds/43bst

Elementary implementations: summary

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

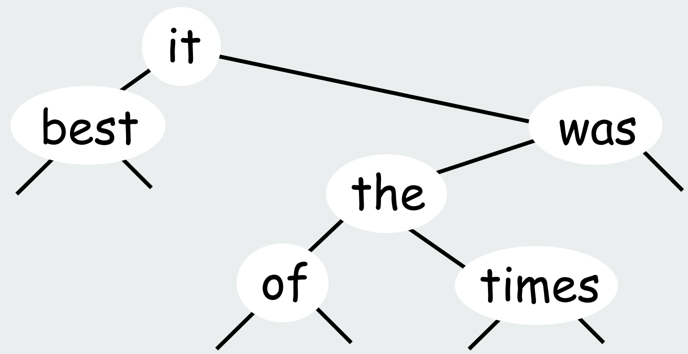
Challenge:

Efficient implementations of `get()` and `put()` and ordered iteration.

- basic implementations
- randomized BSTs
- deletion in BSTs

Binary Search Trees (BSTs)

Def. A BINARY SEARCH TREE is a binary tree in symmetric order.



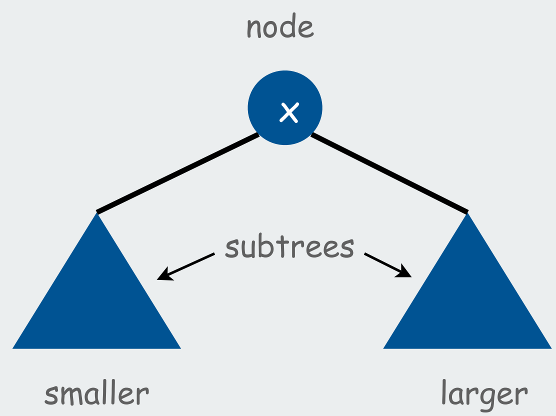
A binary tree is either:

- empty
- a key-value pair and two binary trees
  [neither of which contain that key]

equal keys ruled out to facilitate
associative array implementations

Symmetric order means that:

- every node has a key
- 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 **Node**.

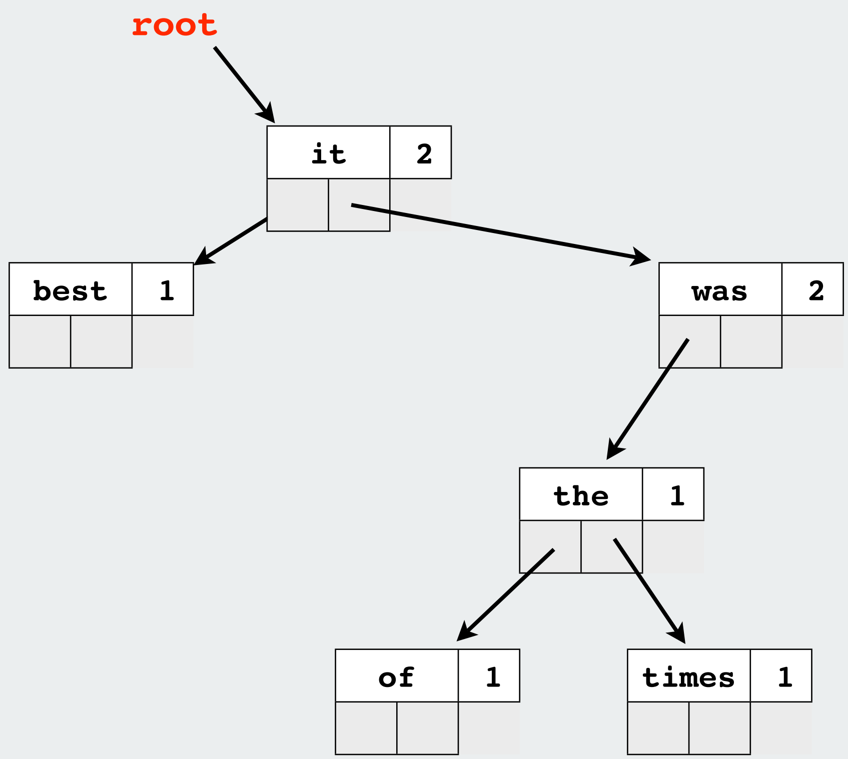
A **Node** is comprised of four fields:

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

smaller keys larger keys

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

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

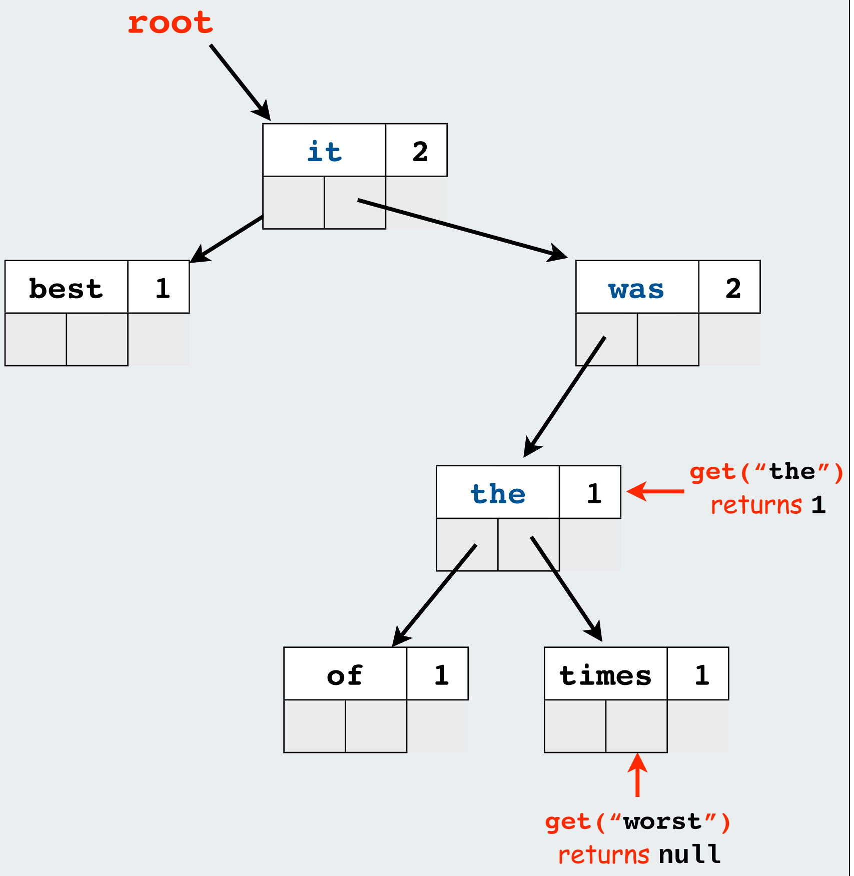


BST implementation (skeleton)

```
public class BST<Key extends Comparable<Key>, Value>
             implements Iterable<Key>
{
    private Node root;                          ← instance variable

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

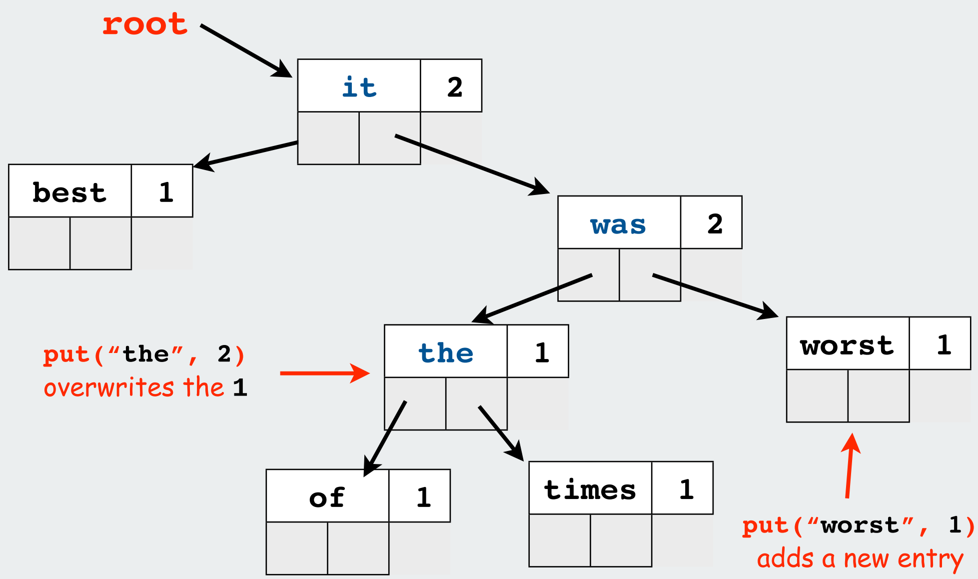
BST implementation (search)

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



BST implementation (insert)

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

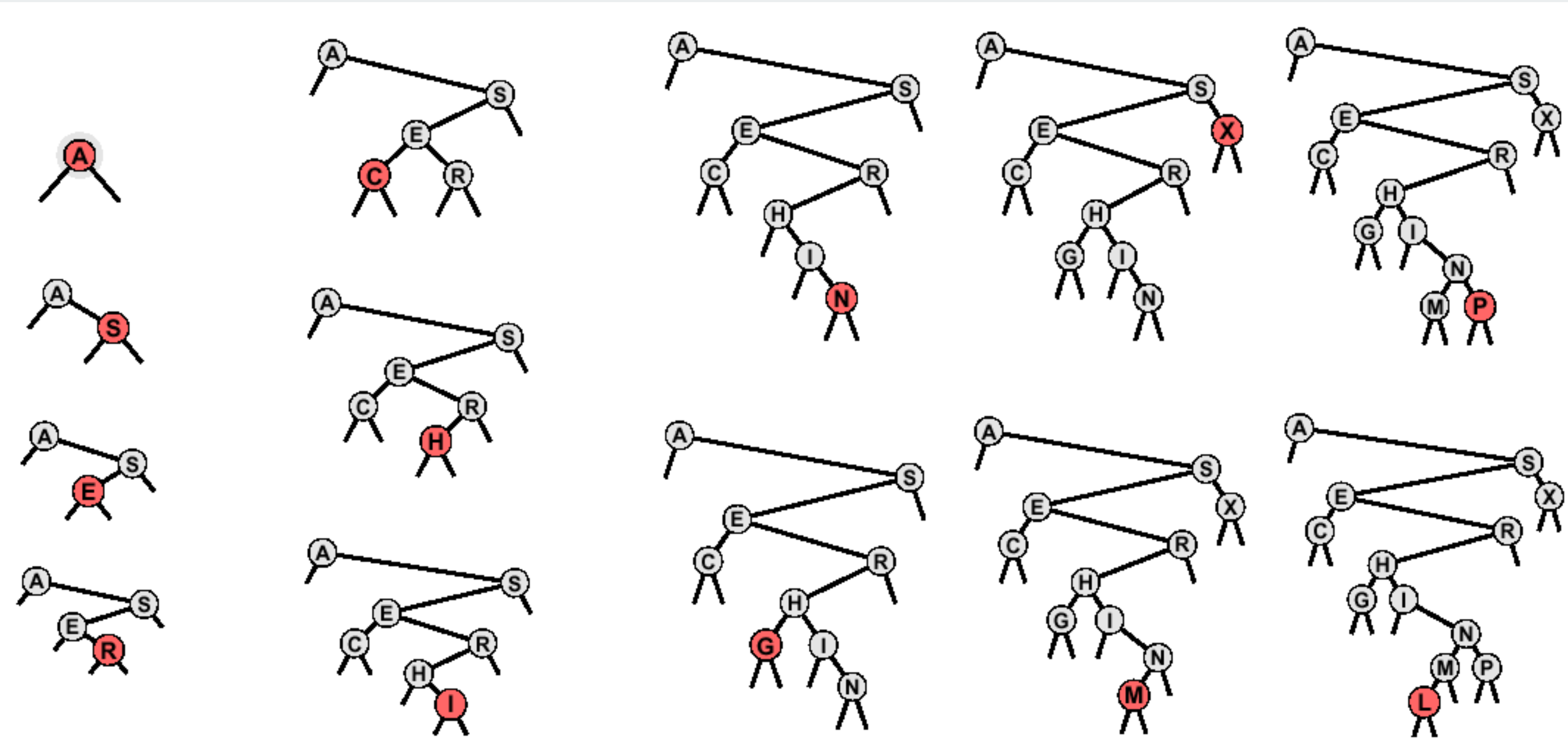


Caution: tricky recursive code.
Read carefully!

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

BST: Construction

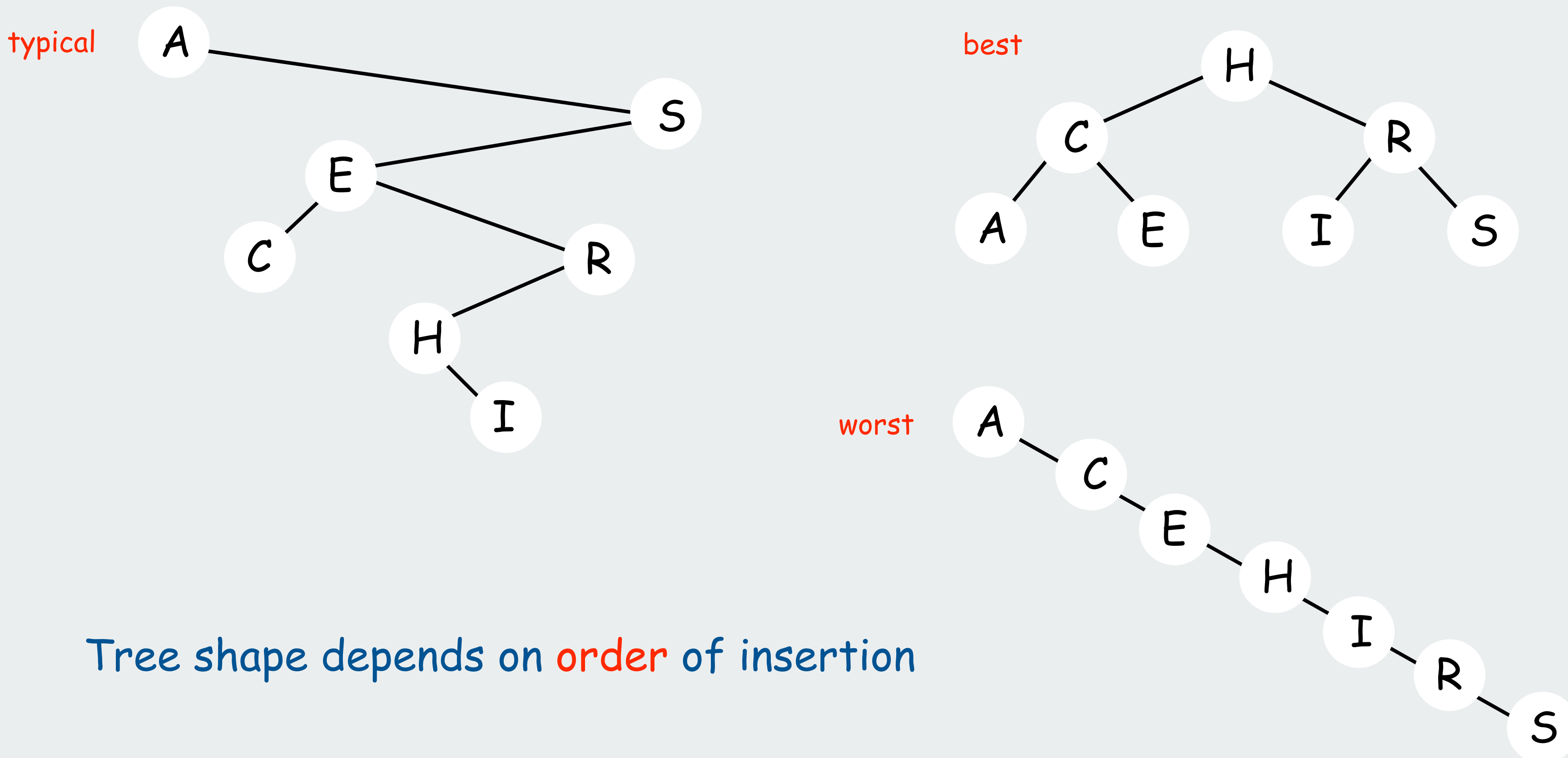
Insert the following keys into BST. **A S E R C H I N G X M P L**



Tree Shape

Tree shape.

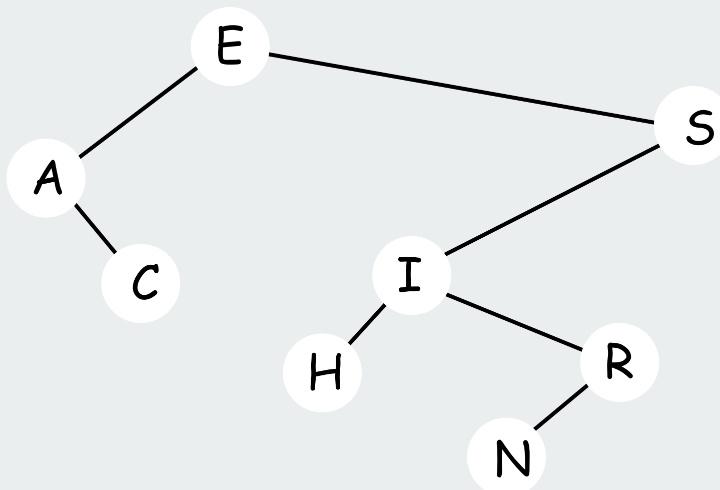
- Many BSTs correspond to same input data.
- Cost of search/insert is proportional to depth of node.



Tree shape depends on order of insertion

BST implementation: iterator?

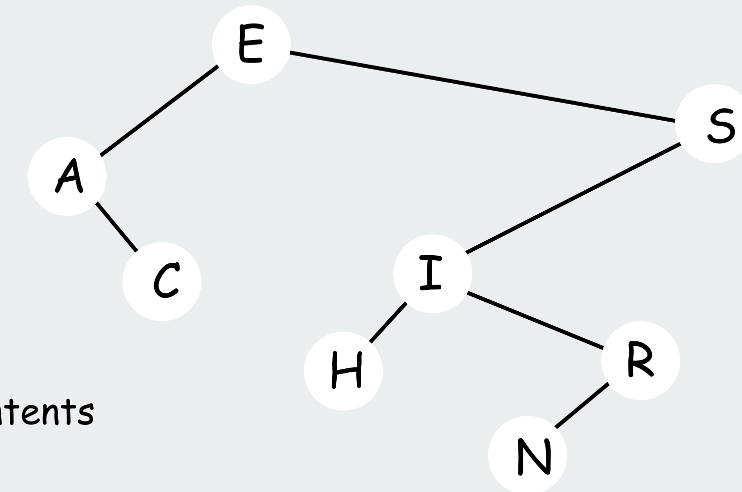
```
public Iterator<Key> iterator()
{ return new BSTIterator(); }

private class BSTIterator
              implements Iterator<Key>
{
   BSTIterator()
   {   }

   public boolean hasNext()
   {   }

   public Key next()
   {   }
}
```



BST implementation: iterator?

Approach: mimic recursive inorder traversal

```
public void visit(Node x)
{
   if (x == null) return;
   visit(x.left)
   StdOut.println(x.key);
   visit(x.right);
}
```



Stack contents

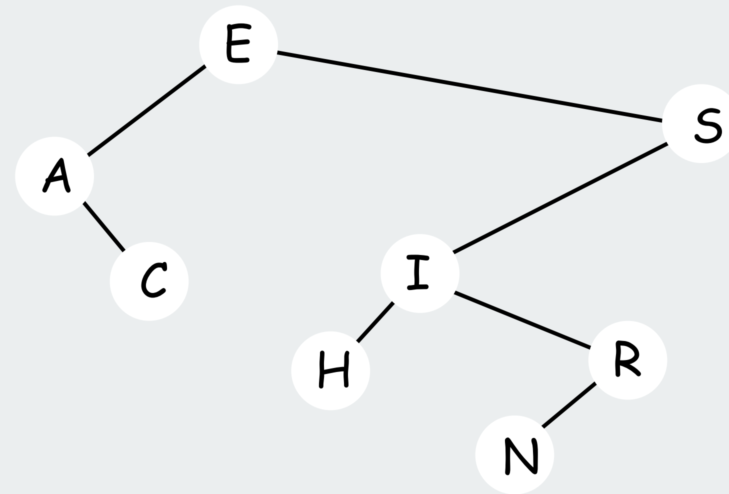
```
visit(E)                      E
  visit(A)                    A  E
    print A           A       E
    visit(C)                  C  E
      print C         C       E
  print E             E
  visit(S)                    S
    visit(I)                  I  S
      visit(H)        H       H  I  S
        print H               I  S
      print I         I       S
      visit(R)                R  S
        visit(N)              N  R  S
          print N     N       R  S
        print R       R       S
    print S           S
```

To process a node

- follow left links until empty (pushing onto stack)
- pop and process
- process node at right link

BST implementation: iterator

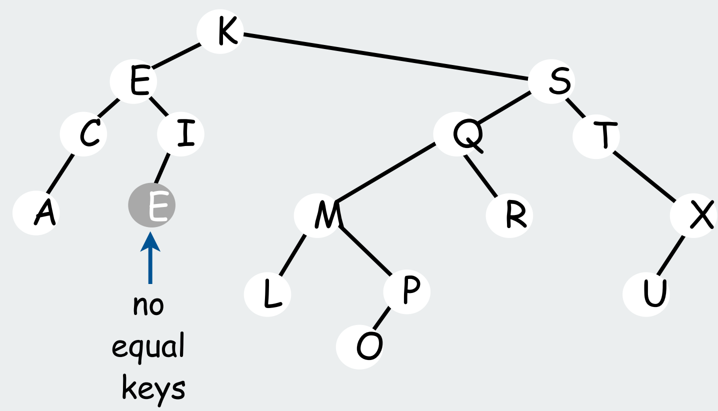
```
public Iterator<Key> iterator()
{ return new BSTIterator(); }

private class BSTIterator
              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;
   }
}
```



	A	E	
A	C	E	
C	E		
E	H	I	S
H	I	S	
I	N	R	S
N	R	S	
R	S		
S			

1-1 correspondence between BSTs and Quicksort partitioning

Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
A	C	E	E	I	K	L	P	O	R	M	Q	S	X	U	T
A	C	E	E	I	K	L	P	O	M	Q	R	S	X	U	T
A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X



BSTs: analysis

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

≈ 1.38 lg N, variance = O(1)

Proof: 1-1 correspondence with quicksort partitioning

Theorem. If keys are inserted in random order, height of tree is proportional to lg N, except with exponentially small probability.

mean ≈ 6.22 lg N, variance = O(1)

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

e.g., keys inserted in ascending order

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

Elementary implementations: summary

implementation	guarantee		average case		ordered iteration?	operations on keys
	search	insert	search	insert		
unordered array	N	N	N/2	N/2	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
unordered list	N	N	N/2	N	no	`equals()`
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 **`get()`** and **`put()`** and ordered iteration.

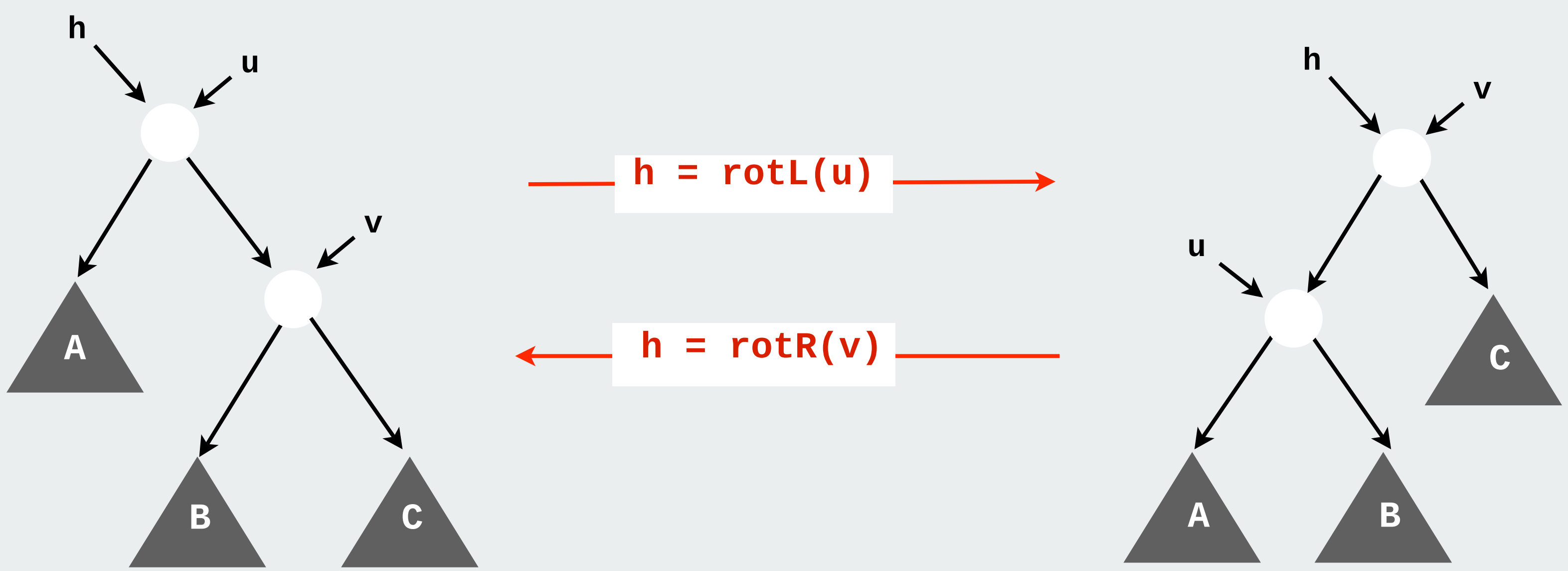
- basic implementations
- **randomized BSTs**
- deletion in BSTs

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



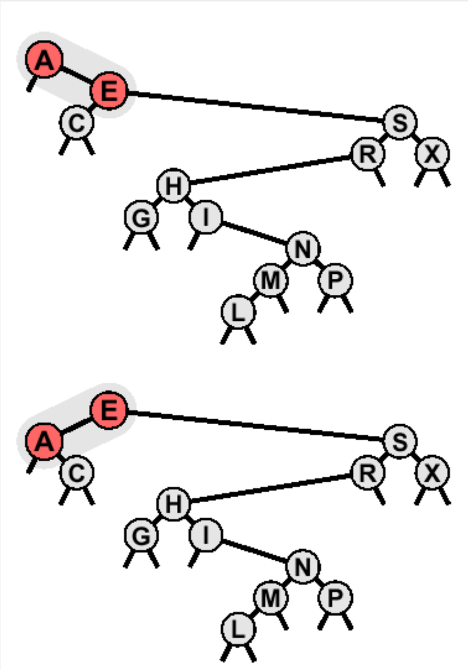
Key point: no change in search code (!)

Rotation

Fundamental operation to rearrange nodes in a tree.

- easier done than said
- raise some nodes, lowers some others

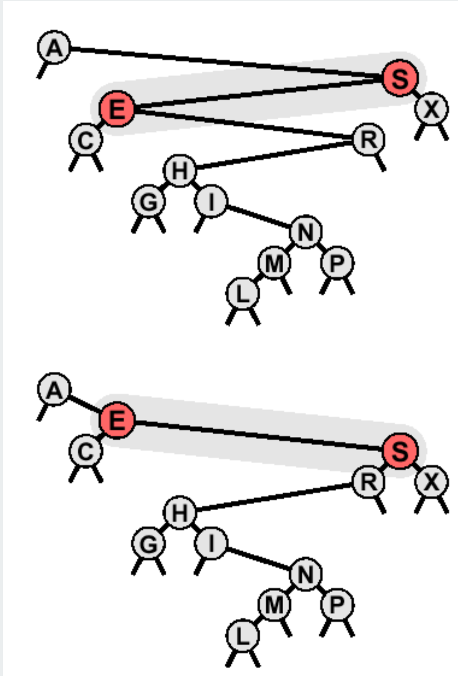
root = rotL(A)




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

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

A.left = rotR(S)

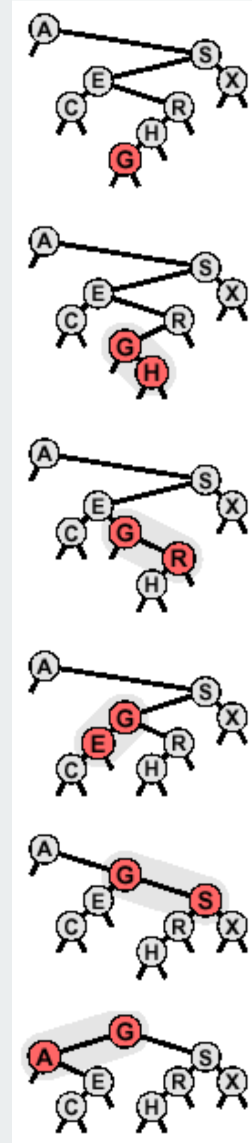


Recursive BST Root Insertion

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

- Insert as in standard BST.
- Rotate inserted node to the root.
- Easy recursive implementation

insert G

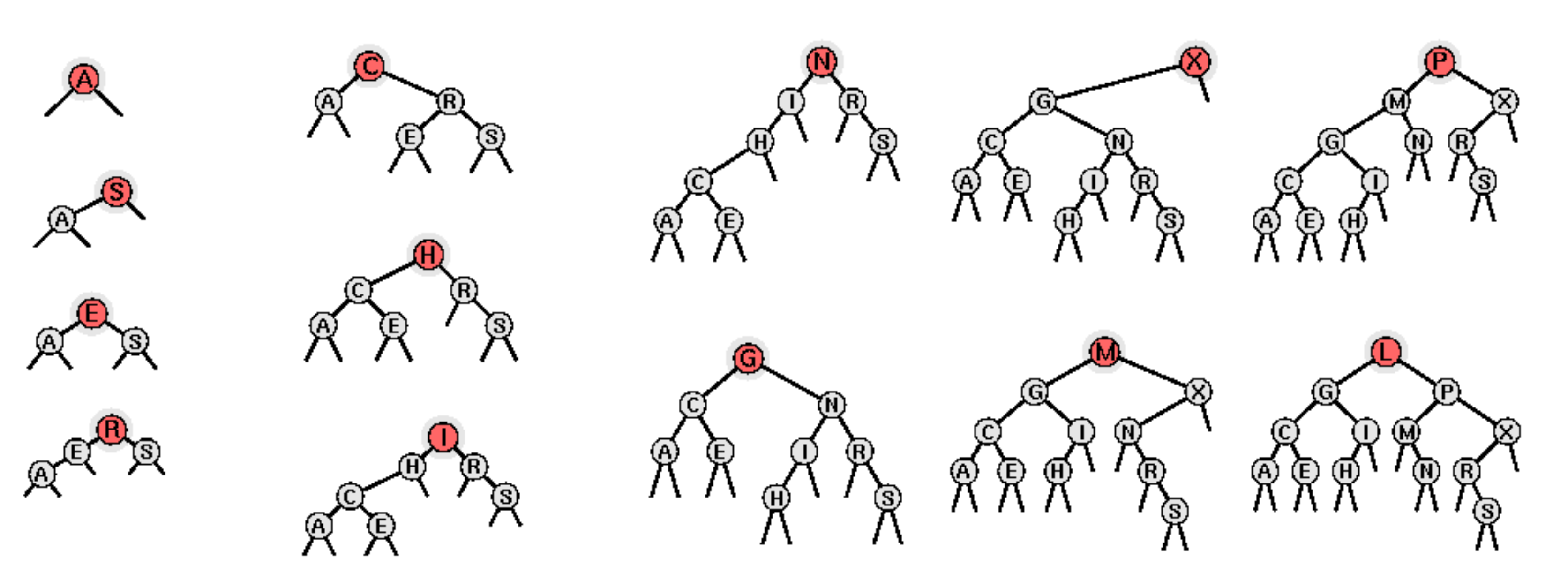


Caution: very tricky recursive code.
Read very carefully!

```
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.val = val;
   else 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);  }
   return x;
}
```

Constructing a BST with root insertion

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 advanced algorithms.

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).

```
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;
}
```

need to maintain count of nodes in tree rooted at x

Constructing a randomized BST

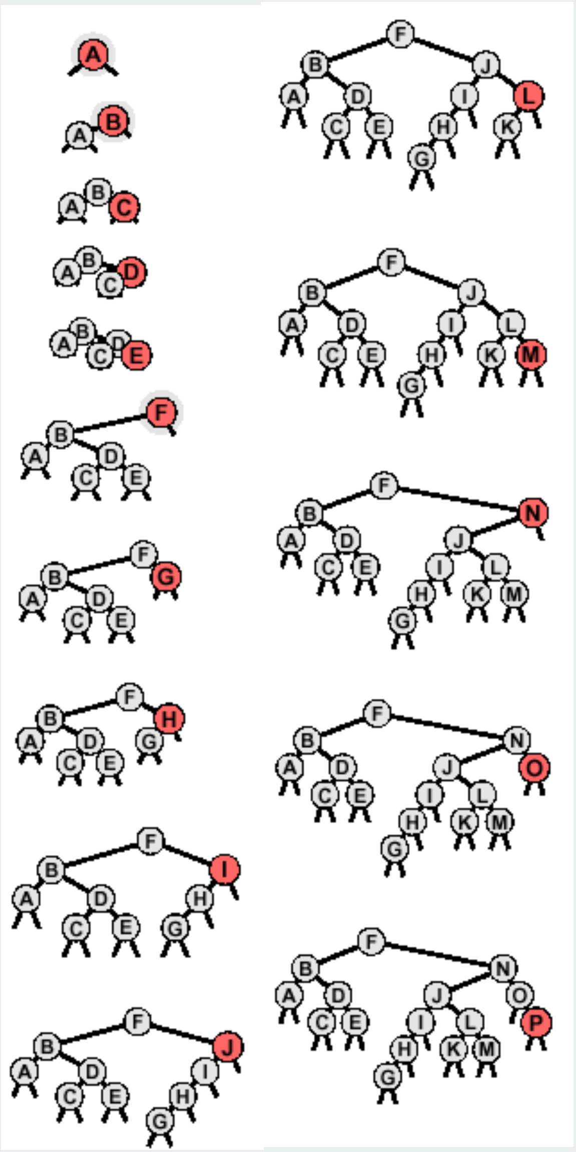
Ex: Insert distinct keys in ascending order.

Surprising fact:

Tree has same shape as if keys were inserted in random order.

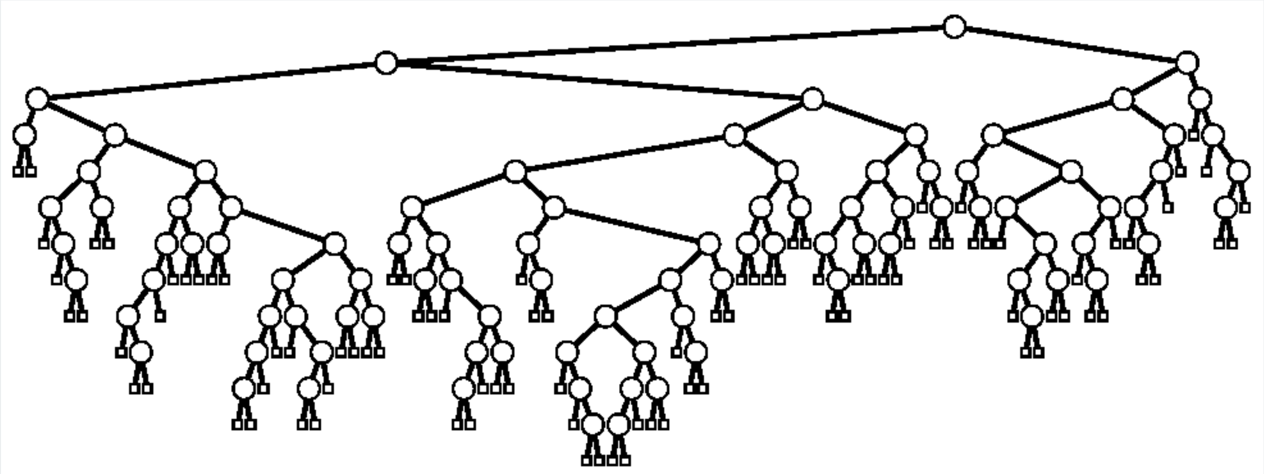
Random trees result from any insert order

Note: to maintain associative array abstraction need to check whether key is in table and replace value without rotations if that is the case.



Randomized BST

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



- Expected height is ~6.22 lg N
- Average search cost is ~1.38 lg N.
- Exponentially small chance of bad balance.

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

Summary of symbol-table implementations

implementation	guarantee		average case		ordered iteration?	operations on keys
	search	insert	search	insert		
unordered array	N	N	N/2	N/2	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
unordered list	N	N	N/2	N	no	`equals()`
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	7 lg N	7 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`

Randomized BSTs provide the desired guarantee

probabilistic, with exponentially small chance of quadratic time

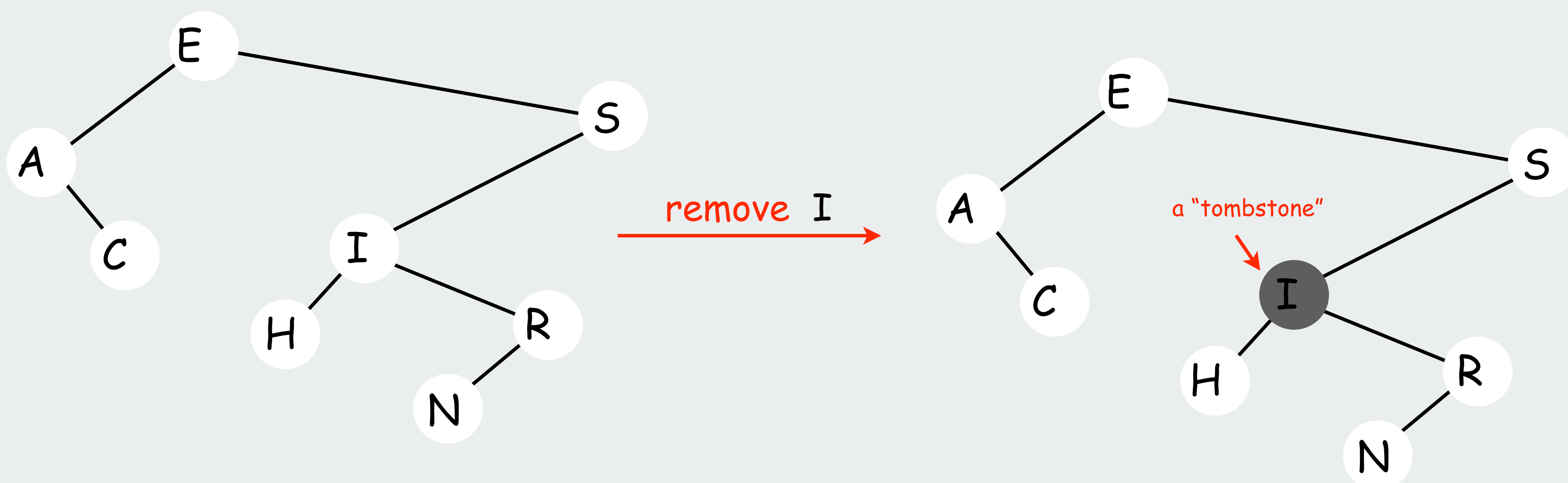
Bonus (next): Randomized BSTs also support delete (!)

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

BST delete: lazy approach

To remove a node with a given key

- set its value to **`null`**
- leave key in tree to guide searches
  [but do not consider it equal to any search key]



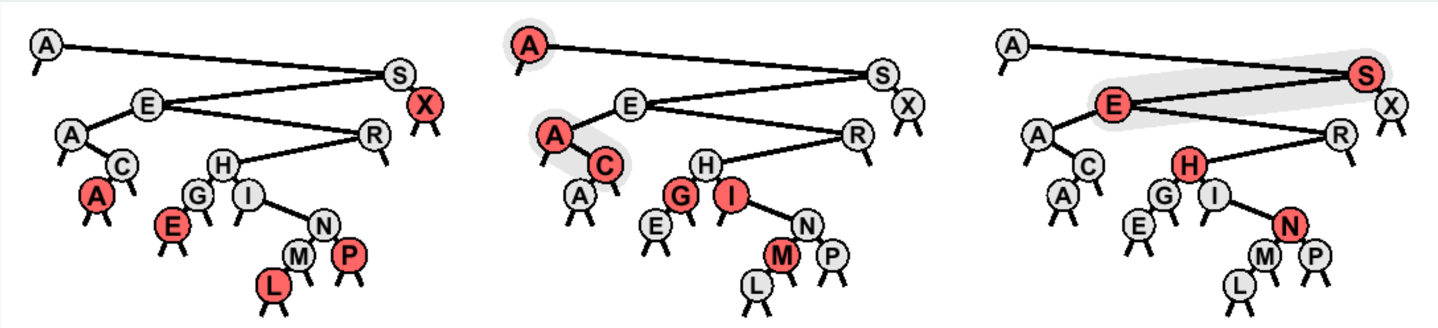
Cost. O(log N') per insert, search, and delete, where N' is the number of elements ever inserted in the BST.

Unsatisfactory solution: Can get overloaded with tombstones.

BST delete: first approach

To remove a node from a BST. [Hibbard, 1960s]

- 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

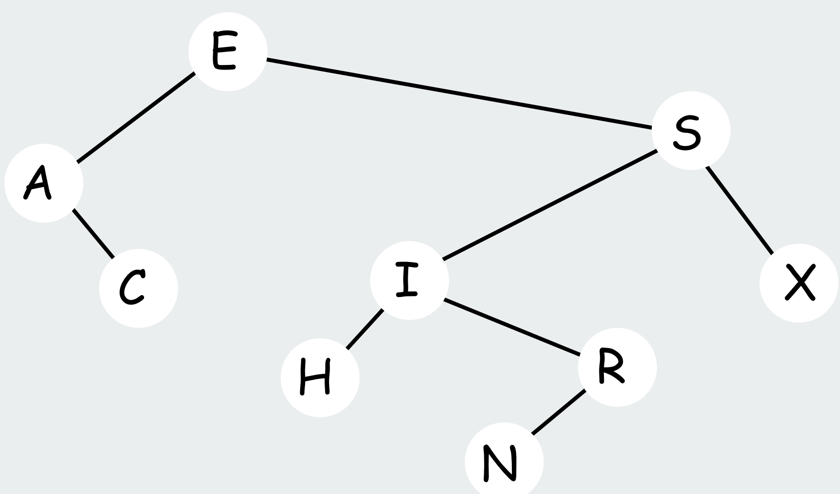
Unsatisfactory solution. Not symmetric, code is clumsy.
Surprising consequence. Trees not random (!) $\Rightarrow$ sqrt(N) per op.

Longstanding open problem: simple and efficient delete for BSTs

To delete a node containing a given key

- remove the node
- join the two remaining subtrees to make a tree

Ex. Delete S in



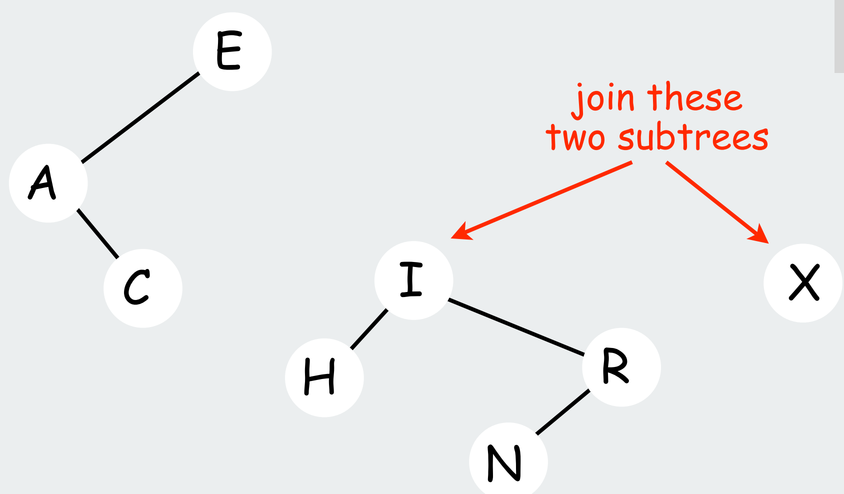
Deletion in randomized BSTs

To delete a node containing a given key

- remove the node
- join its two subtrees

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

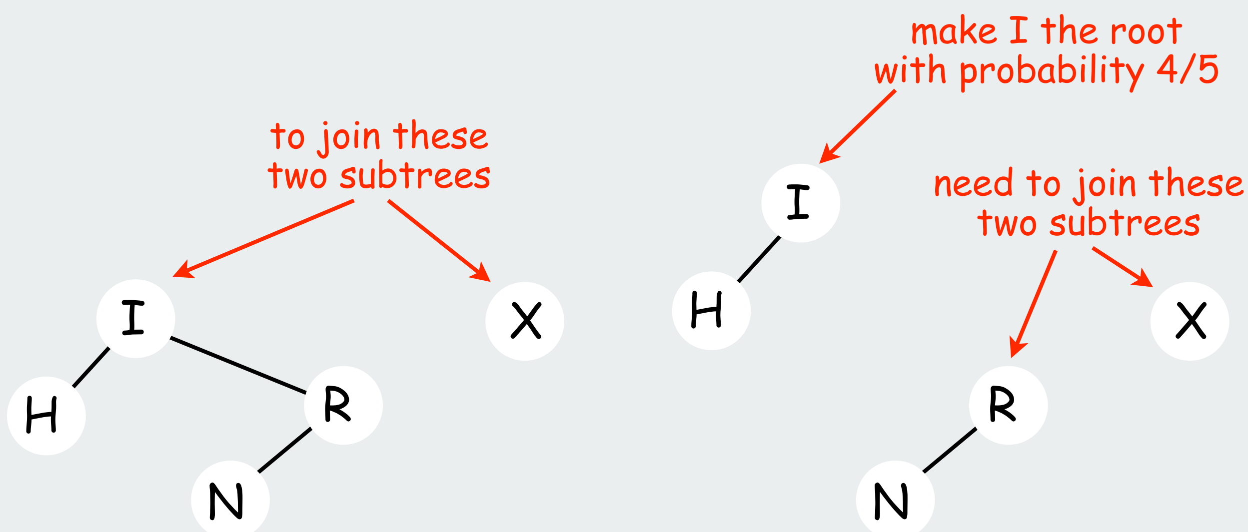
Ex. Delete S in



Join in randomized BSTs

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

- maintain counts of nodes in subtrees (L and R)
- with probability L/(L+R)
  - make the root of the left the root
  - make its left subtree the left subtree of the root
  - join its right subtree to R to make the right subtree of the root
- with probability L/(L+R) do the symmetric moves on the right

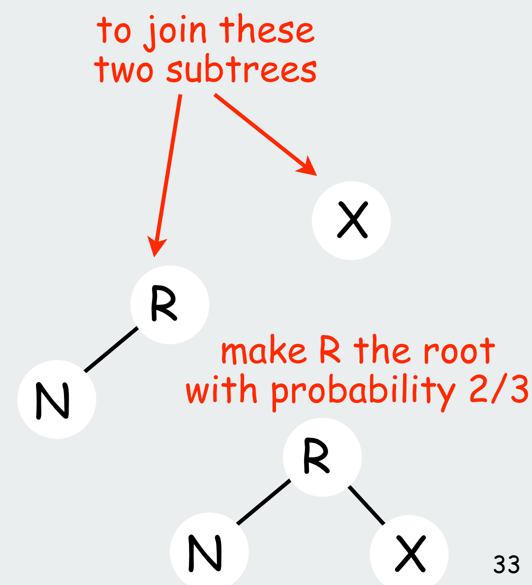


Join in randomized BSTs

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

- maintain counts of nodes in subtrees (L and R)
- with probability L/(L+R)
  - make the root of the left the root
  - make its left subtree the left subtree of the root
  - join its right subtree to R to make the right subtree of the root
- with probability L/(L+R) do the symmetric moves on the right

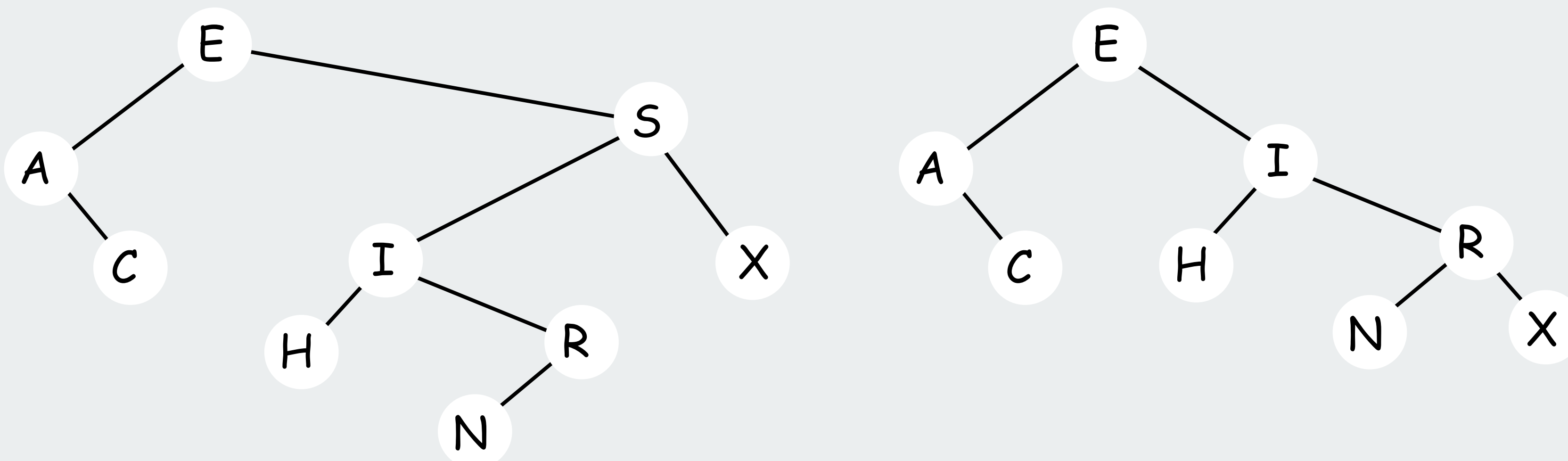
```
private Node join(Node a, Node b)
{
   if (a == null) return a;
   if (b == null) return b;
   int cmp = key.compareTo(x.key);
   if (StdRandom.bernoulli((double)*a.N / (a.N + b.N))
     { a.right = join(a.right, b); return a; }
   else
     { b.left  = join(a, b.left ); return b; }
}
```



Deletion in randomized BSTs

To delete a node containing a given key
- remove the node
- join its two subtrees

Ex. Delete S in



Theorem. Tree still random after delete (!)

Bottom line. Logarithmic guarantee for search/insert/delete

Summary of symbol-table implementations

implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes

Randomized BSTs provide the desired guarantees

probabilistic, with exponentially small chance of error

Next lecture: Can we do better?