https://algs4.cs.princeton.edu

3.2 BINARY SEARCH TREES

- *BSTs*
- *iteration*
- *ordered operations*
- *deletion (see book or videos)*

https://algs4.cs.princeton.edu

3.2 Binary Search Trees

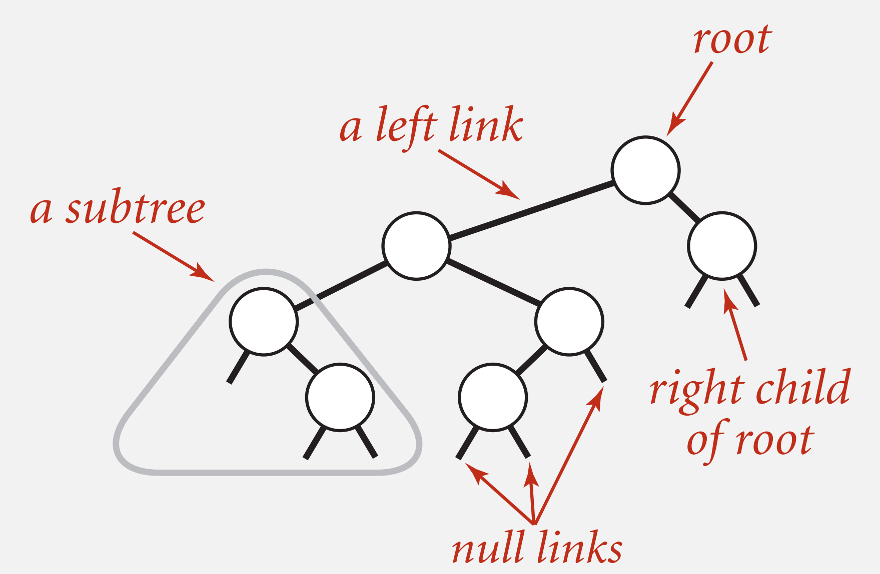
- *BSTs*
- *iteration*
- *ordered operations*
- *deletion (see book or videos)*

Binary search trees

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

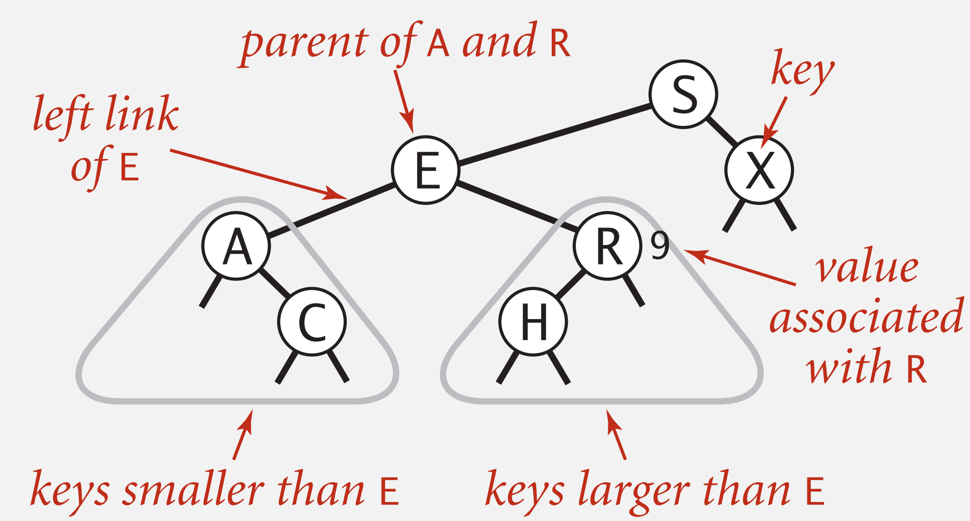
A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right).



Symmetric order. Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



Differences between heaps and binary search trees

	Heap	BST
Supported operations	Insert, delete-max	insert, **search**, delete, ordered operations
What is inserted	Keys	Key-value pairs
Underlying data structure	Resizing array	Linked nodes
Tree shape	Fixed shape given *n* (complete binary tree)	Varies; depends on data
Ordering of keys	Somewhat ordered parent > child for max heap	Totally ordered left child < parent < right child
Duplicate keys allowed?	Yes	No

Binary search trees: quiz 1

Which of the following properties hold?

- **A.** If a binary tree is heap ordered, then it is symmetrically ordered.
- **B.** If a binary tree is symmetrically ordered, then it is heap ordered.
- **C.** Both A and B.
- **D.** Neither A nor B.

BST representation in Java

A BST contains a reference to a root `Node`.

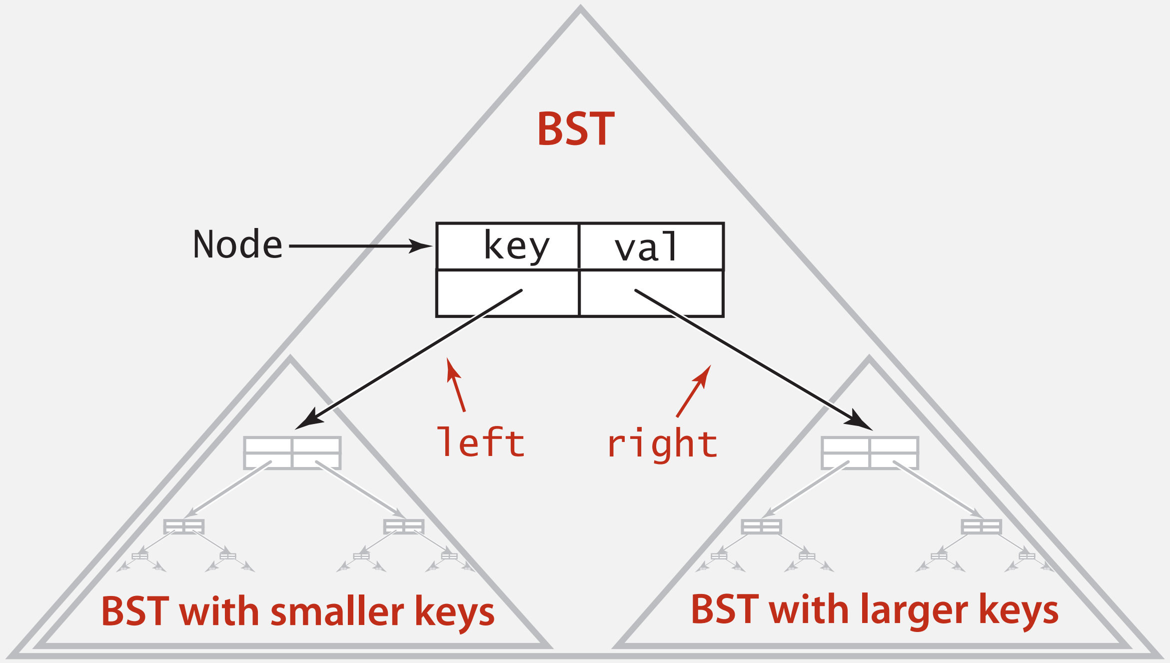
A `Node` is composed of four fields:

- 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, right;

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

Key and Value are generic types; Key is Comparable



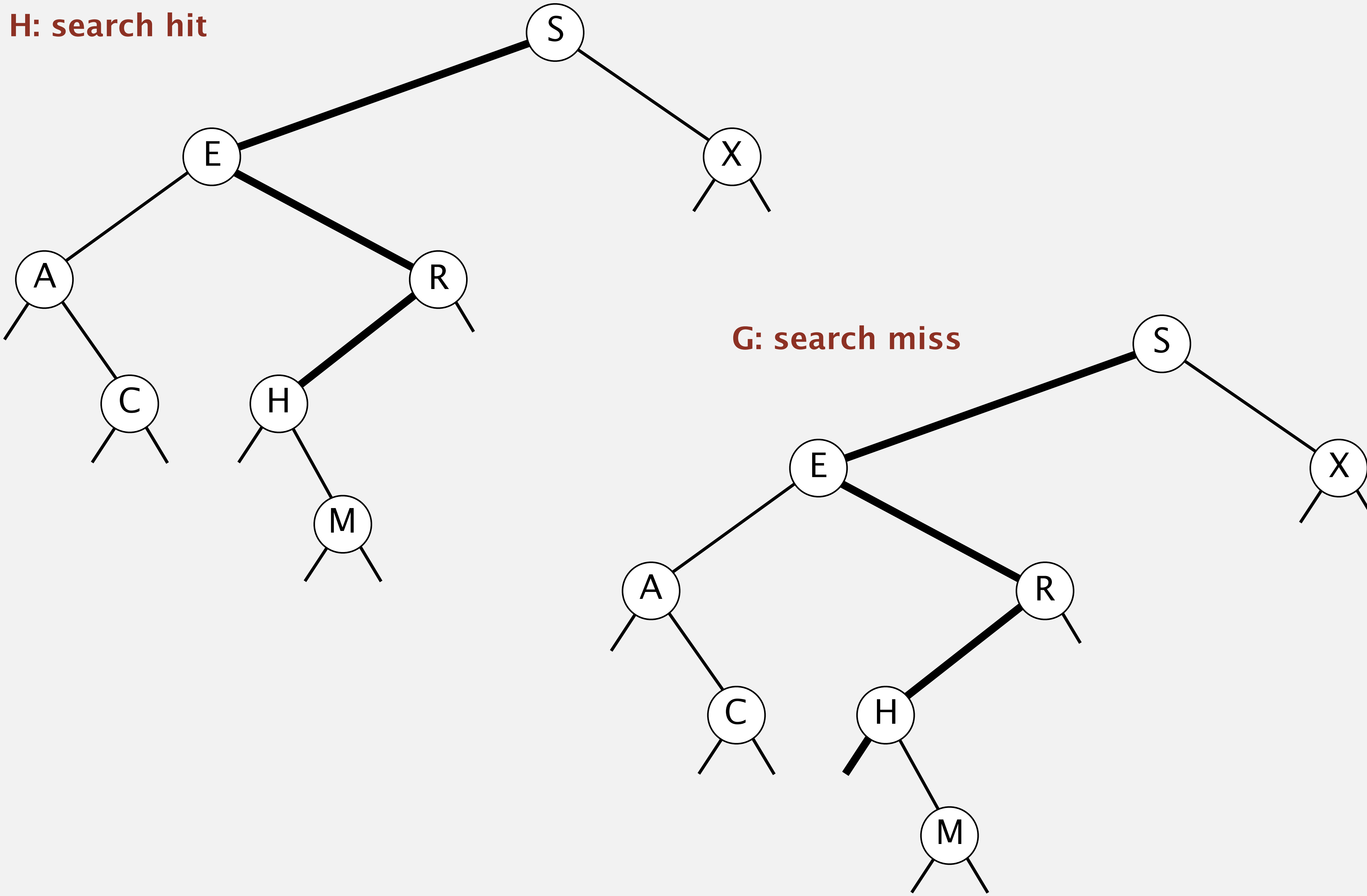
Binary search tree

BST implementation (skeleton)

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

   private class Node
   {  /* see previous slide */  }

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

   public Value get(Key key)
   {  /* see next slide */  }

   public Iterable<Key> keys()
   {  /* see slides in next section */  }

   public void delete(Key key)
   {  /* see textbook */  }

}
```

BST search (get)

If less, go left; if greater, go right; if equal, search hit; if null node, search miss.



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;
}
```

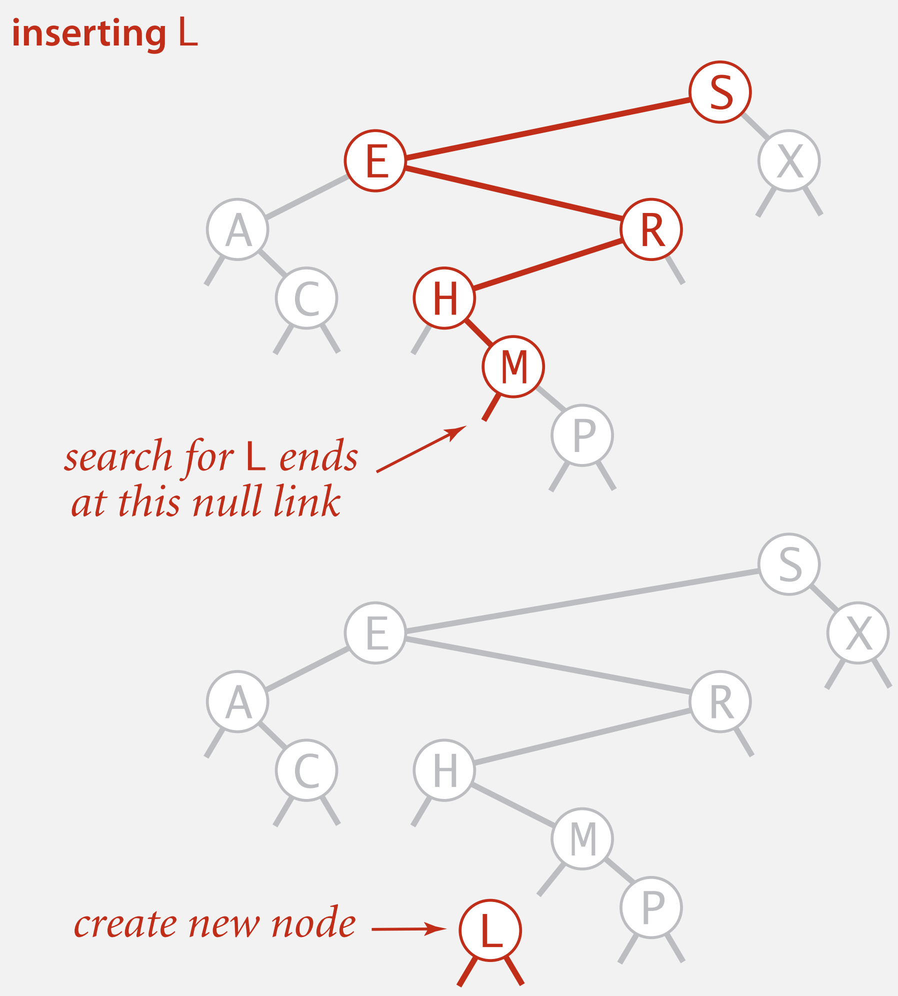
Cost. Number of compares = 1 + depth of node.

BST insert (put)

Associate value with key.

Search for key, then two cases:

- Key in tree $\Rightarrow$ reset value.
- Key not in tree $\Rightarrow$ add new node.

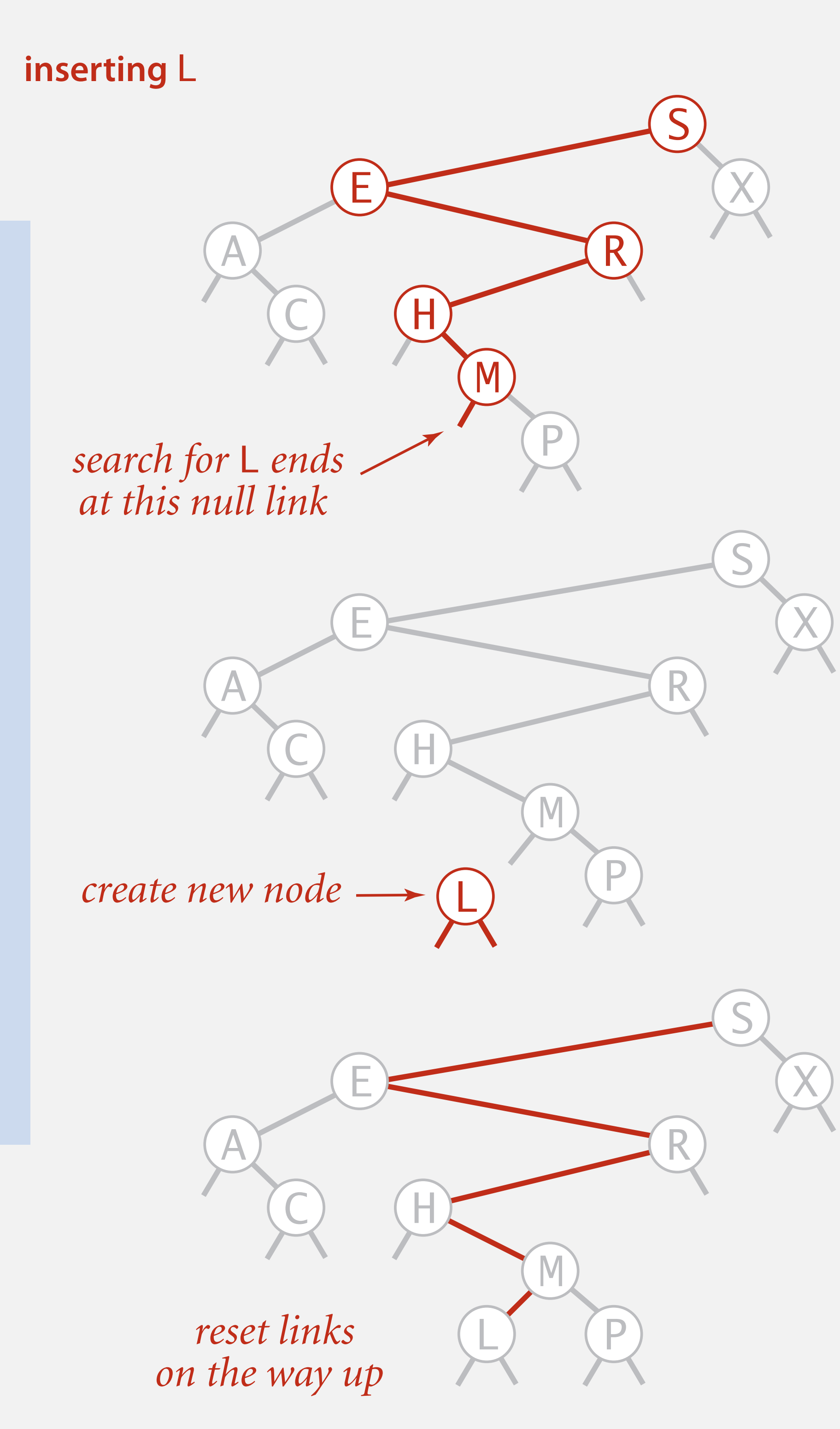


BST insert: Java implementation

Put. Associate value with key.

Tricky!

```
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;
}
```

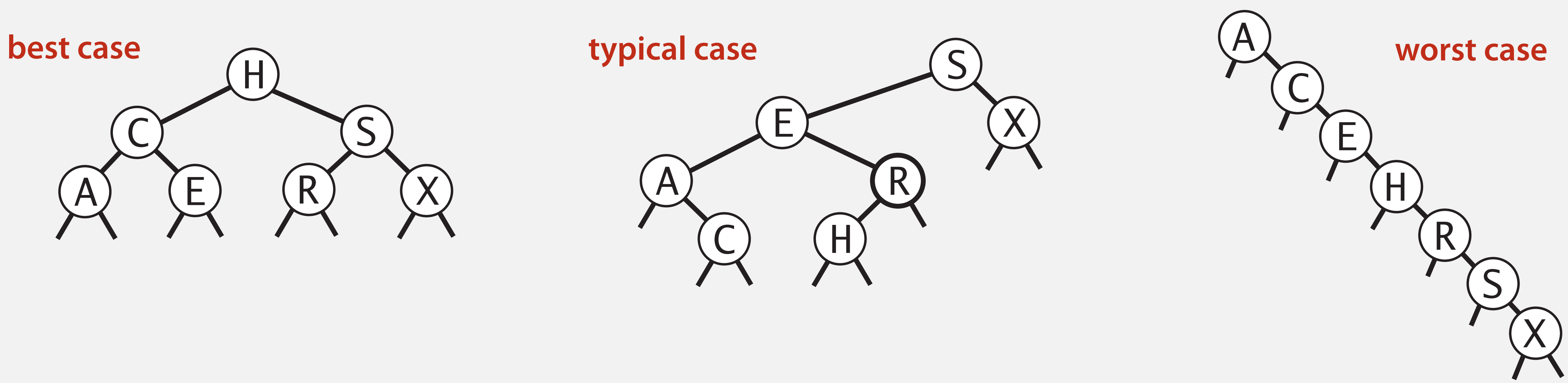


Insertion into a BST

Cost. Number of compares = 1 + depth of node.

Tree shape

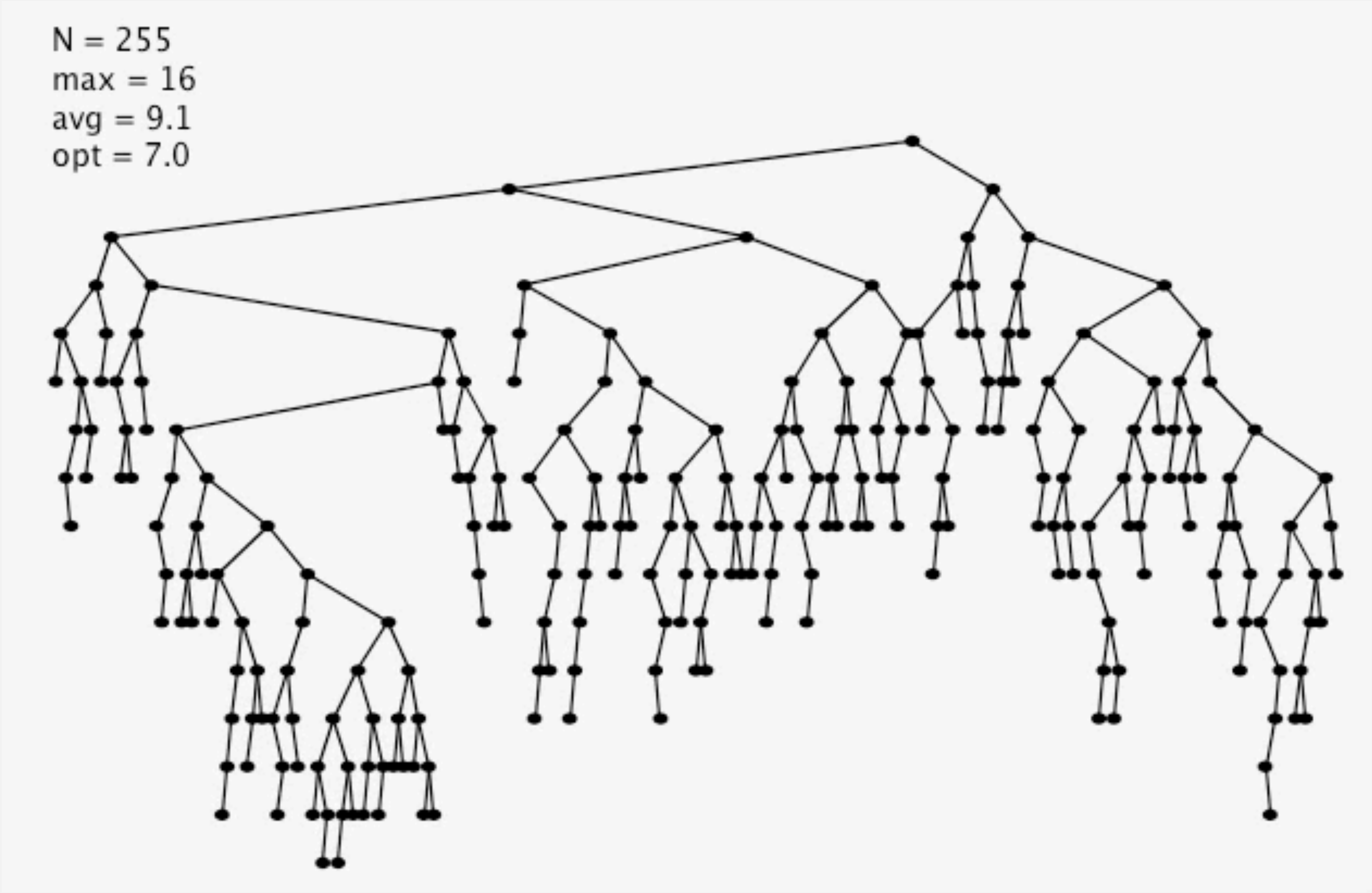
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert = 1 + depth of node.



Bottom line. Tree shape depends on order of insertion.

BST insertion: random order visualization

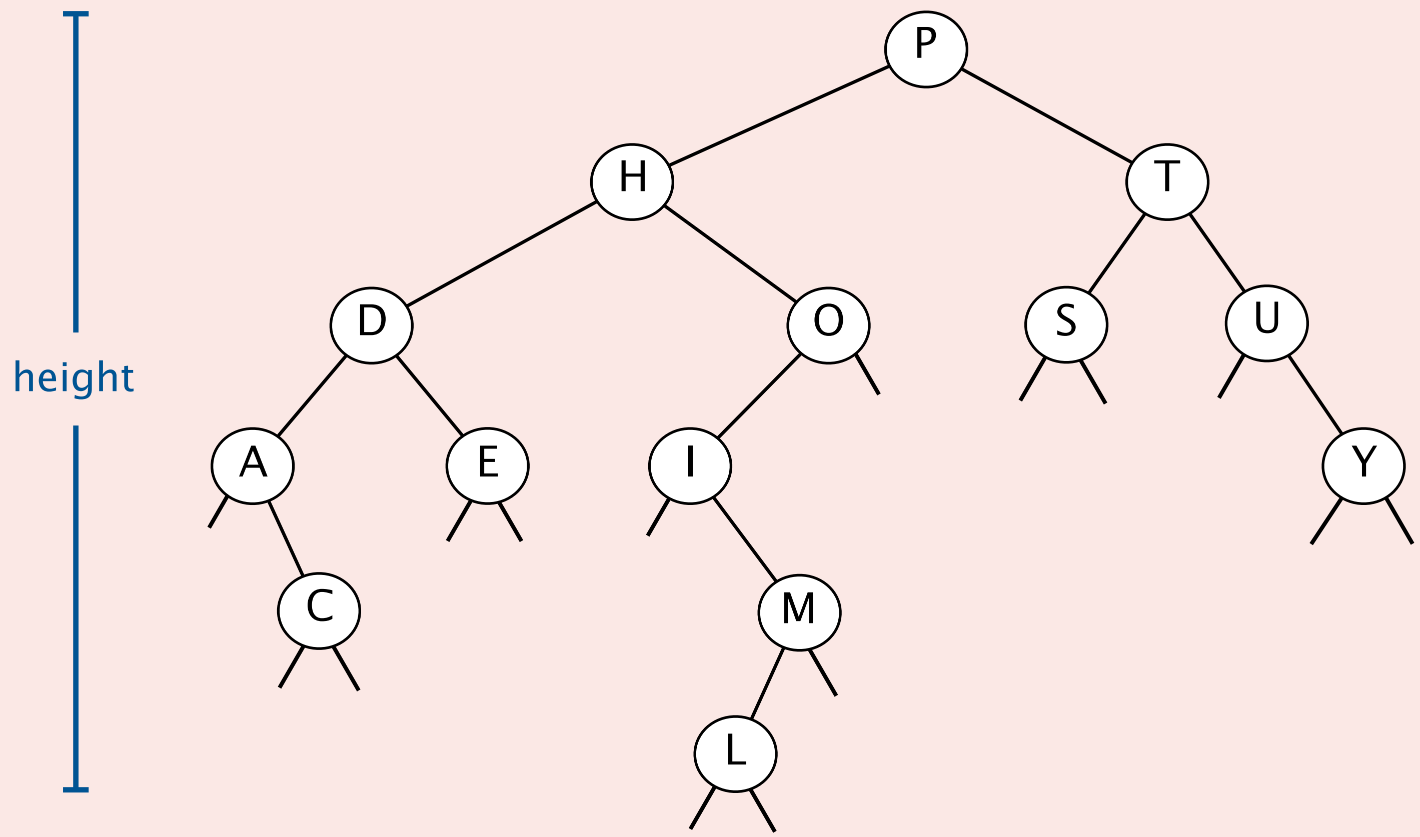
Ex. Insert keys in random order.



Binary search trees: quiz 2

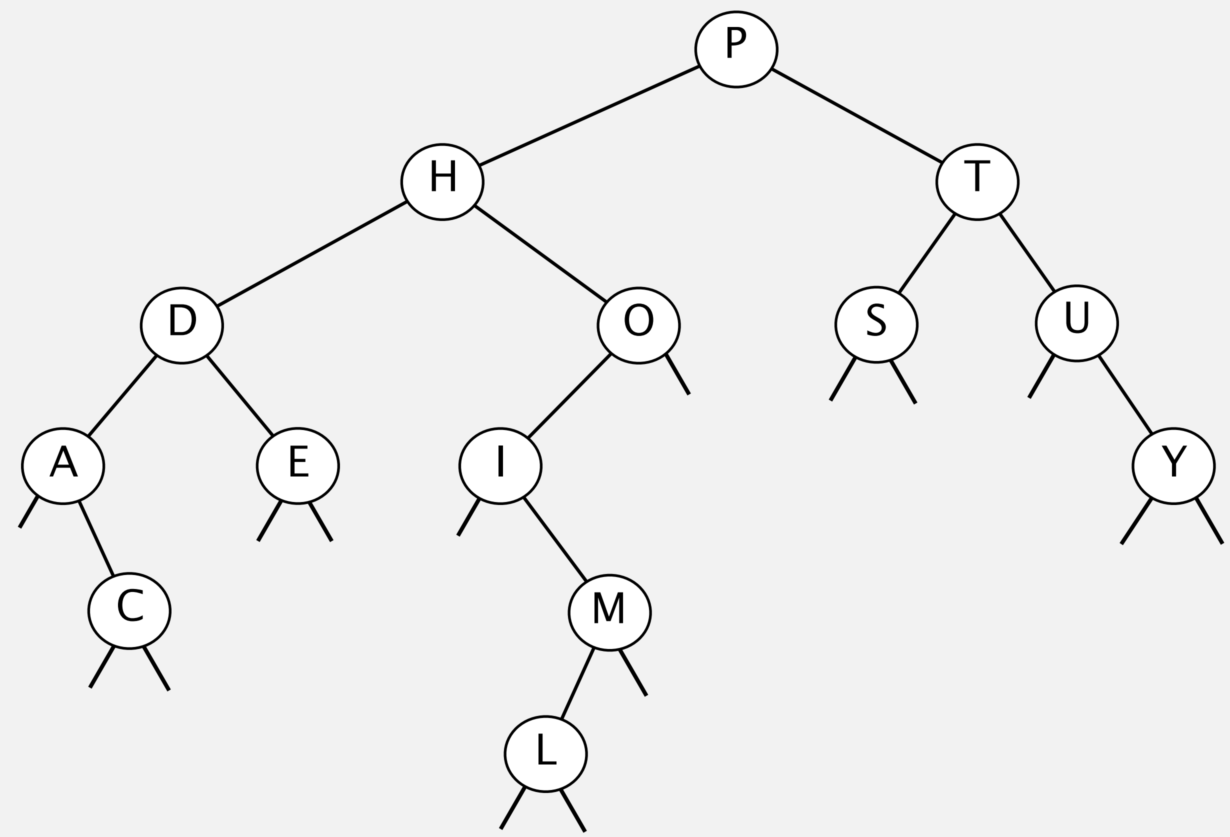
Suppose that you insert n keys in random order into a BST. What is the expected height of the resulting BST?

- **A.** $\sim \lg n$
- **B.** $\sim \ln n$
- **C.** $\sim 2 \lg n$
- **D.** $\sim 2 \ln n$
- **E.** $\sim 4.31107 \ln n$



Correspondence between BSTs and quicksort partitioning

0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
H	L	E	A	D	O	M	C	I	P	T	Y	U	S
D	C	E	A	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y



Remark. Correspondence is 1–1 if array has no duplicate keys.

BSTs: mathematical analysis

Proposition. If n distinct keys are inserted into a BST in random order, the expected number of compares for a search/insert is $\sim 2 \ln n$.

Pf. 1–1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If n distinct keys are inserted into a BST in random order, the expected height is $\sim 4.31107 \ln n$.

expected depth of function-call stack in quicksort

How Tall is a Tree?

Bruce Reed
CNRS, Paris, France
reed@moka.ccr.jussieu.fr

ABSTRACT

Let H_n be the height of a random binary search tree on n nodes. We show that there exists constants $\alpha = 4.31107\ldots$ and $\beta = 1.95\ldots$ such that $\mathrm{E}(H_n) = \alpha \log n - \beta \log \log n + O(1)$, We also show that $\mathrm{Var}(H_n) = O(1)$.

But... Worst-case height is $n - 1$.

Unlike quicksort, worst case matters — client may not insert in random order.

ST implementations: summary

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	n	n	n	n	`equals()`
binary search (ordered array)	$\log n$	n	$\log n$	n	`compareTo()`
BST	n	n	$\log n$	$\log n$	`compareTo()`

Why not shuffle to ensure a (probabilistic) guarantee of $\log n$?

https://algs4.cs.princeton.edu

3.2 Binary Search Trees

- BSTs
- *iteration*
- ordered operations
- deletion

Binary search trees: quiz 3

In which order does traverse(root) print the keys in the BST?

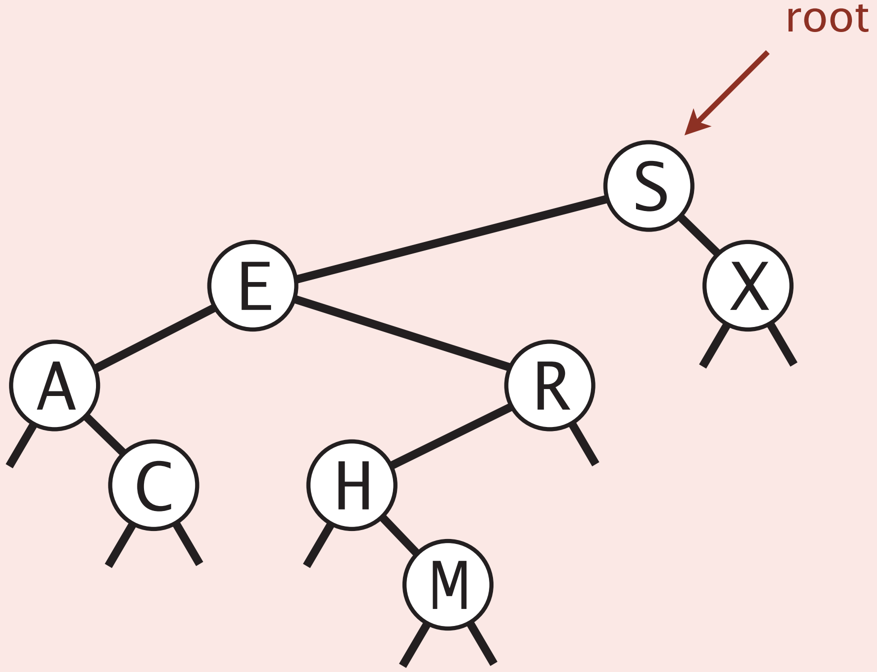
```
private void traverse(Node x)
{
   if (x == null) return;
   traverse(x.left);
   StdOut.println(x.key);
   traverse(x.right);
}
```

A. A C E H M R S X

B. S E A C R H M X

C. C A M H R E X S

D. S E X A R C H M

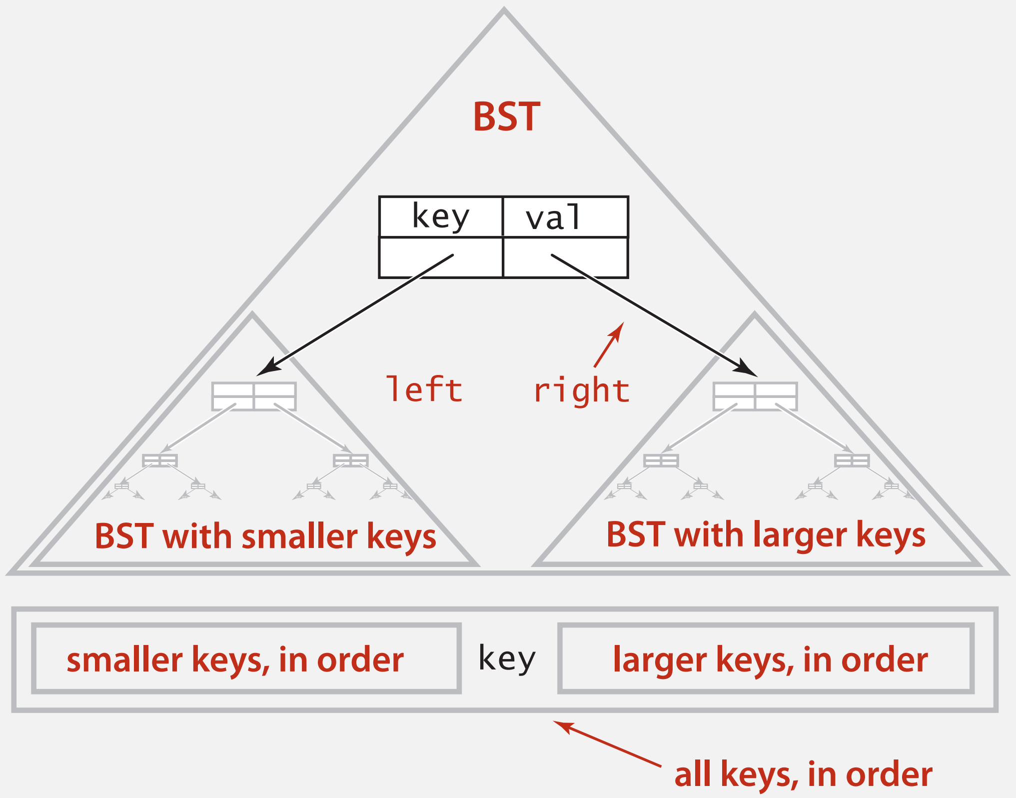


Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
public Iterable<Key> keys()
{
   Queue<Key> q = new Queue<Key>();
   inorder(root, q);
   return q;
}
```

```
private void inorder(Node x, Queue<Key> q)
{
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```



Property. Inorder traversal of a BST yields keys in ascending order.

Running time

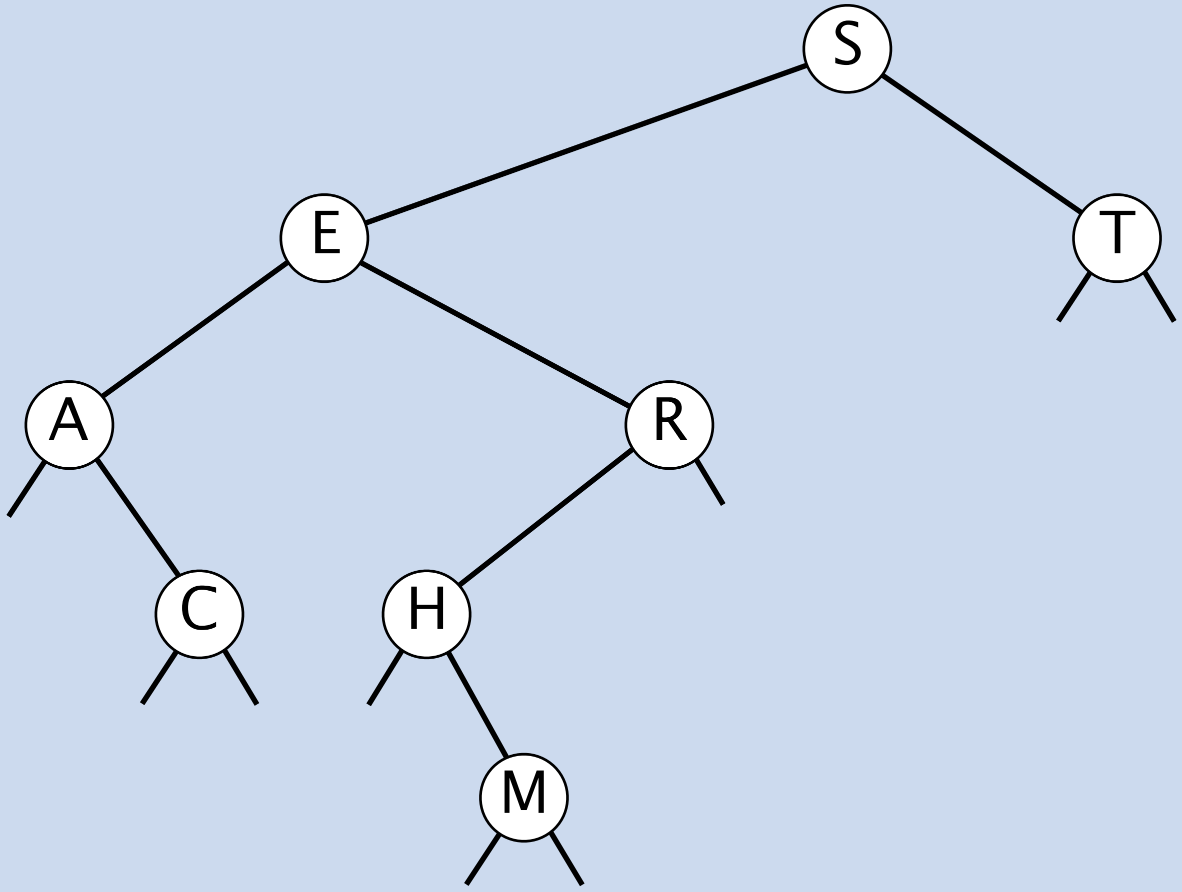
Property. Inorder traversal of a BST takes linear time.

Silicon Valley

Level-Order Traversal

Level-order traversal of a binary tree.

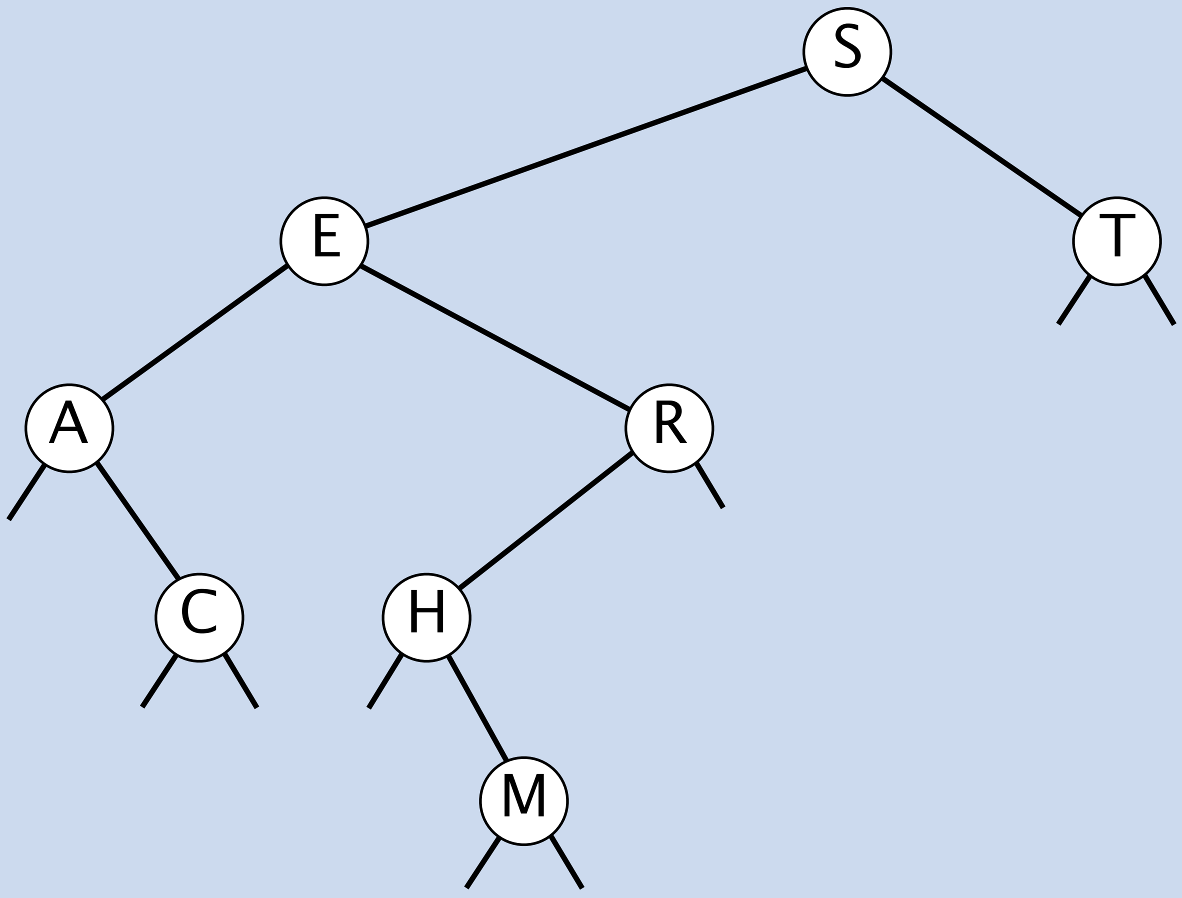
- Process root.
- Process children of root, from left to right.
- Process grandchildren of root, from left to right.
- ...



level-order traversal: S E T A R C H M

Level-Order Traversal

Q. Given binary tree, how to compute level-order traversal?



level-order traversal: S E T A R C H M

https://algs4.cs.princeton.edu

3.2 Binary Search Trees

- *BSTs*
- *iteration*
- *ordered operations*
- *deletion*

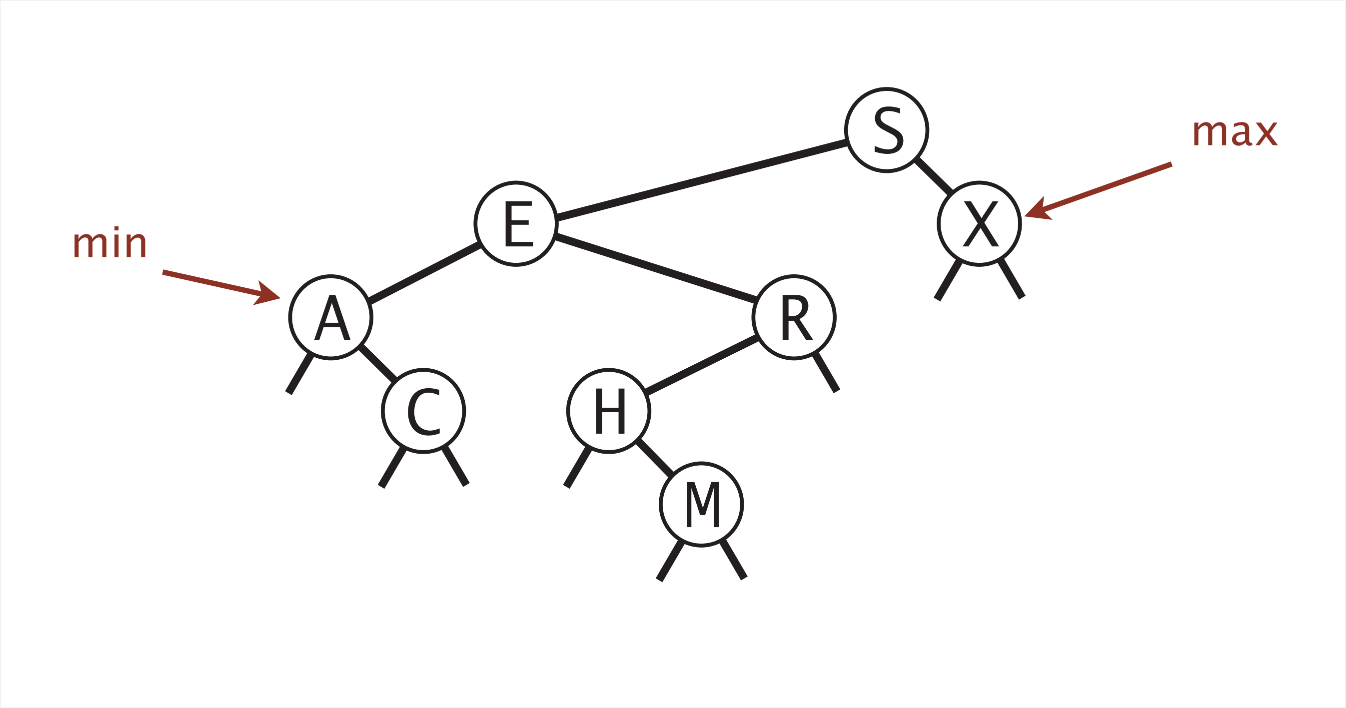
Omitted for midterm.

Only *Rank* discussed in lecture.
See book/videos for the rest.

Minimum and maximum

Minimum. Smallest key in BST.

Maximum. Largest key in BST.

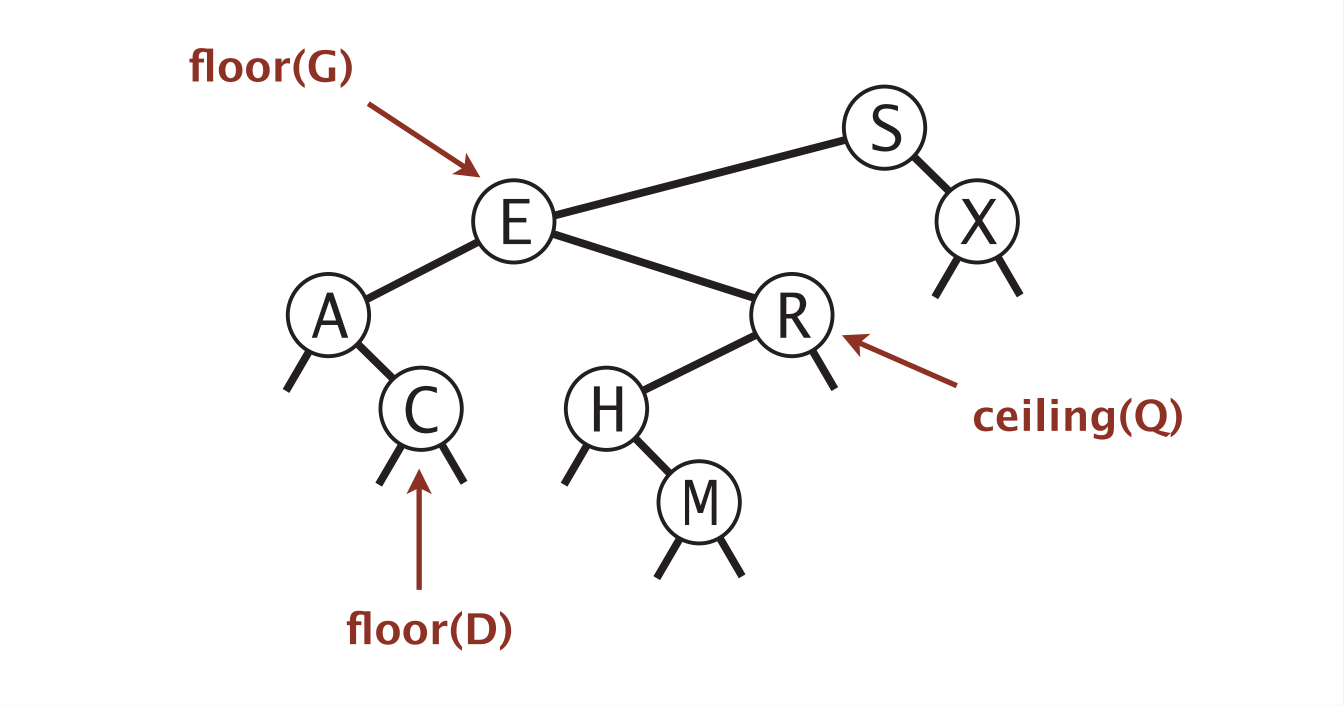


Q. How to find the min / max?

Floor and ceiling

Floor. Largest key in BST ≤ query key.

Ceiling. Smallest key in BST ≥ query key.



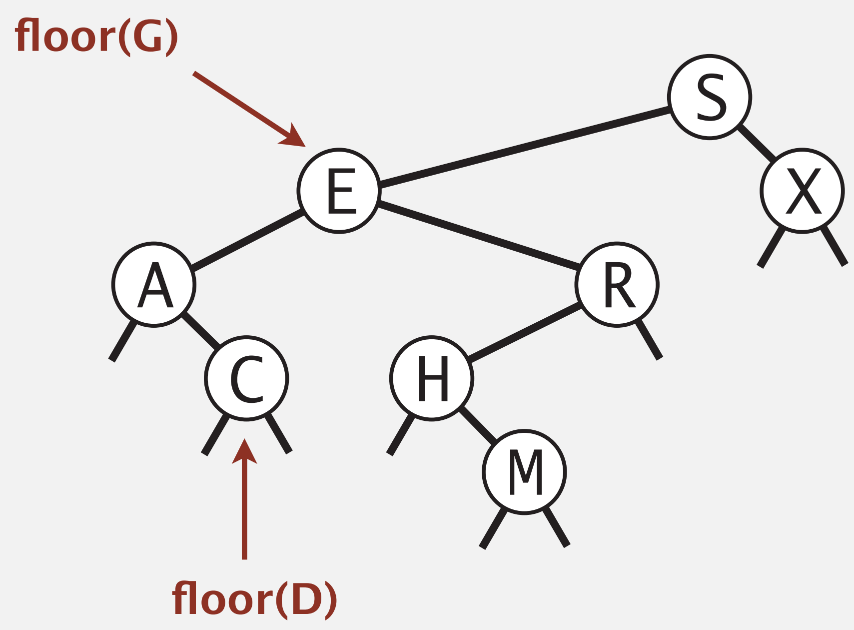
Q. How to find the floor / ceiling?

Computing the floor

Floor. Largest key in BST ≤ query key.

Key idea.

- To compute `floor(key)`, search for key.
- Both `floor(key)` and `ceiling(key)` must be on search path. Why?



Computing the floor

key in node is too large
(floor can't be in right subtree)

```
public Key floor(Key key)
{  return floor(root, key, null);  }
```

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

key in node is a candidate for floor
(floor can't be in left subtree)

key in node is better candidate than best
(x must be in right subtree of node containing best)

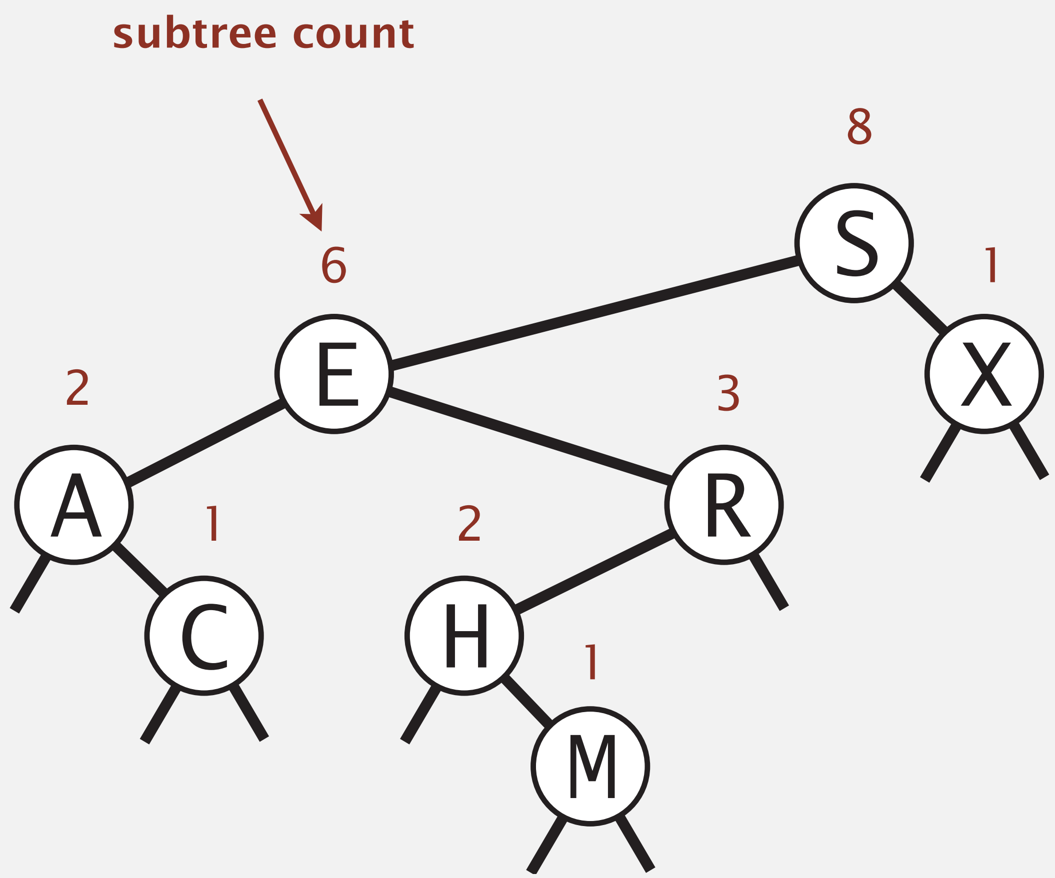
Rank and select

Rank. How many keys < *key* ?

Select. Key of rank *k*.

Q. How to implement `rank()` and `select()` efficiently for BSTs?

A. In each node, store the number of nodes in its subtree.



BST implementation: subtree counts

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

number of nodes in subtree

```
public int size()
{  return size(root);  }

private int size(Node x)
{
   if (x == null) return 0;
   return x.count;
}
```

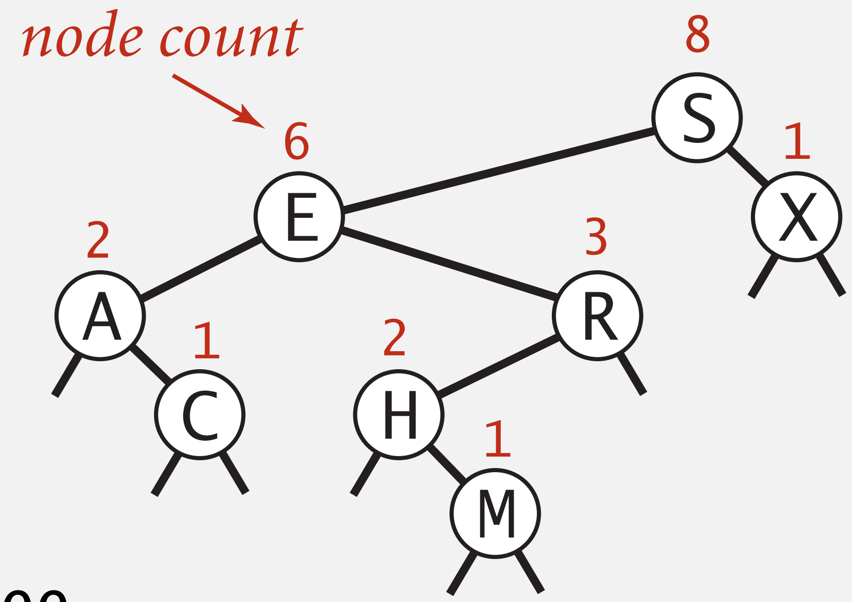
ok to call when x is null

```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val, 1);
   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;

   x.count = 1 + size(x.left) + size(x.right);

   return x;
}
```

initialize subtree count to 1

Rank

Rank. How many keys < *key* ?



key < key in node? Recur on left subtree.

key == key in node? Everything in left subtree.

key > key in node? Everything in left subtree + 1

+ recursive result from right subtree.

```
public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) return rank(key, x.left);
   else if (cmp  > 0) return 1 + size(x.left) + rank(key, x.right);
   else if (cmp == 0) return size(x.left);
}
```

Note: use `size(x.left)` instead of `x.left.count` to avoid null reference.

BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	n	$\log n$	h
insert	n	n	h
min / max	n	1	h
floor / ceiling	n	$\log n$	h
rank	n	$\log n$	h
select	n	1	h
ordered iteration	$n \log n$	n	n

h = height of BST

order of growth of running time of ordered symbol table operations

ST implementations: summary

implementation	guarantee		average case		ordered ops?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	n	n	n	n		`equals()`
binary search (ordered array)	$\log n$	n	$\log n$	n	✔	`compareTo()`
BST	n 🙁	n 🙁	$\log n$	$\log n$	✔	`compareTo()`
red–black BST	$\log n$	$\log n$	$\log n$	$\log n$	✔	`compareTo()`

Next week. Guarantee logarithmic performance for all operations.