INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

BINARY SEARCH TREES

- *Symbol Table*
- *BSTs*
- *Ordered operations*
- *Iteration*
- *Deletion*



http://ds.cs.Rutgers.edu

Slides adopted and modified from Sedgewick/Wayne

Binary Search Trees

- *Symbol Table*
- *BSTs*
- *Ordered operations*
- *Iteration*
- *Deletion*

Motivation – Symbol Table — to associate a value with a key — ability to efficiently search through the table

Telephone books are obsolete



key = name
value = phone number

Unsupported Operations

- Add a new name and associated number
- Remove a given name and associated number
- Change the number associated with a given name

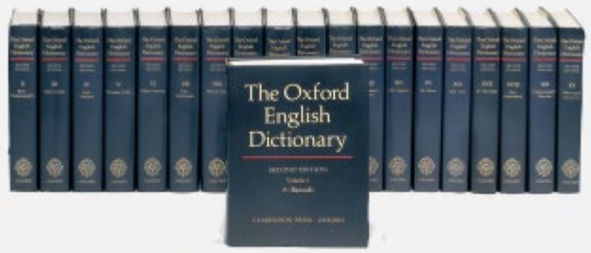
Idea. Store data as (key, value) pairs



key = math function and input
value = function output



key = term
value = article



key = word
value = definition

The client can insert key-value pairs into the symbol table and later being able to search for the value associated with a given key, from among all of the key-value pairs in the table

A Symbol Table API

two generic type parameters

```
public class ST<Key extends Comparable<Key>, Value>
```

-key is unique (no duplicate)
-key cannot be NULL

	`ST()`	*create an empty symbol table*
//insert, new val replace the old val if key exist		
`void`	`put(Key key, Value val)`	*insert key–value pair* ← `a[key] = val;`
`Value`	`get(Key key)` //search	*value paired with key* ← `a[key]`
`boolean`	`contains(Key key)`	*is there a value paired with key?*
`Iterable<Key>`	`keys()`	*all the keys in the symbol table*
`void`	`delete(Key key)`	*remove key (and associated value)*
`boolean`	`isEmpty()`	*is the symbol table empty?*
`int`	`size()`	*number of key–value pairs*

An **API** (Application Programming Interface) defines what kind of operations are available to client programs.

Goal – Design a Symbol Table

Design a symbol table that can support ordered operations



Ordered Symbol Table API

public class ST<Key extends Comparable<Key>, Value>	
⋮	
Key min()	*smallest key*
Key max()	*largest key*
Key floor(Key key)	*largest key less than or equal to key*
Key ceiling(Key key)	*smallest key greater than or equal to key*
int rank(Key key)	*number of keys less than key*
Key select(int k)	*key of rank k*
⋮	

Goal. Implement ordered symbol using appropriate data structure that can support ordered operations, **efficiently**

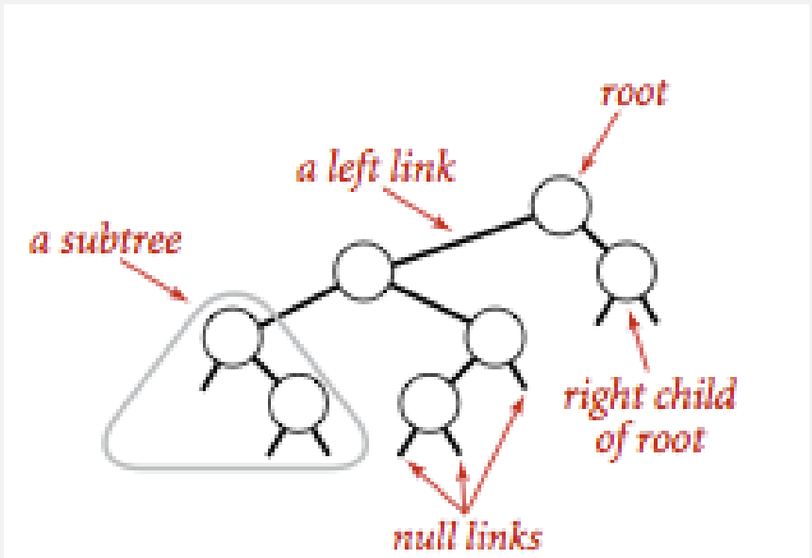
Ordered Symbol Table API – running time analysis

**To study performance (running time) for the larger inputs, two measures are of interest:

- **The total number of keys being searched** (get())
- **The total number of distinct keys in the table**; i.e., the size of the table (get() and put())

**Typical characteristics shared by symbol-table clients:

- Search and insert operations are intermixed.
- The number of distinct keys is not small.
- Substantially more searches than inserts are likely.
- Search and insert patterns, though unpredictable, are not random.



Anatomy of a binary tree

BINARY SEARCH TREES

- *Symbol Table*
- *BSTs* one of the most fundamental algorithms in computer science
- *Ordered operations*
- *Iteration*
- *Deletion*

Binary Search Tree (BST) Motivation (Optional)

Data Structure		Insert		Delete		Search	
		Best	Worst	Best	Worst	Best	Worst
Unsorted	Array	O(1)	O(1)	O(1)	O(n)	O(1)	O(n)
	Linked List	O(1)	O(1)	O(1)	O(n)	O(1)	O(n)
Sorted	Array	O(n)	O(n)	O(1)	O(n)	O(1)	O(log n)
	Linked List	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)

Only if array is kept sparse (with empty cells)

doesn't shift up to remove the deleted spot

BST is a dynamic structure that combines the *flexibility of insertion in a linked structure* with the *efficiency of search in an ordered array*.

We don't know
- How many data items will be inserted
- The insertion sequence

Binary search tree (BST)

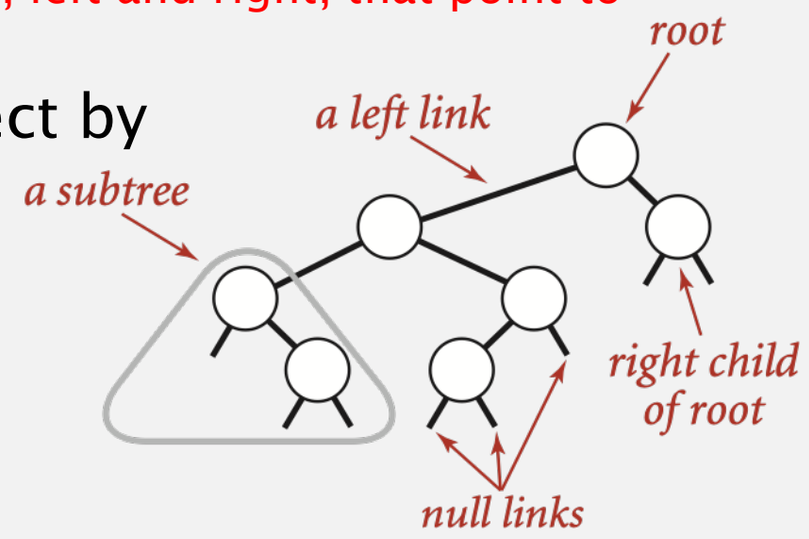
Definition. A BST is a binary tree in symmetric order.

A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right) connect by a root node.

Binary tree —>

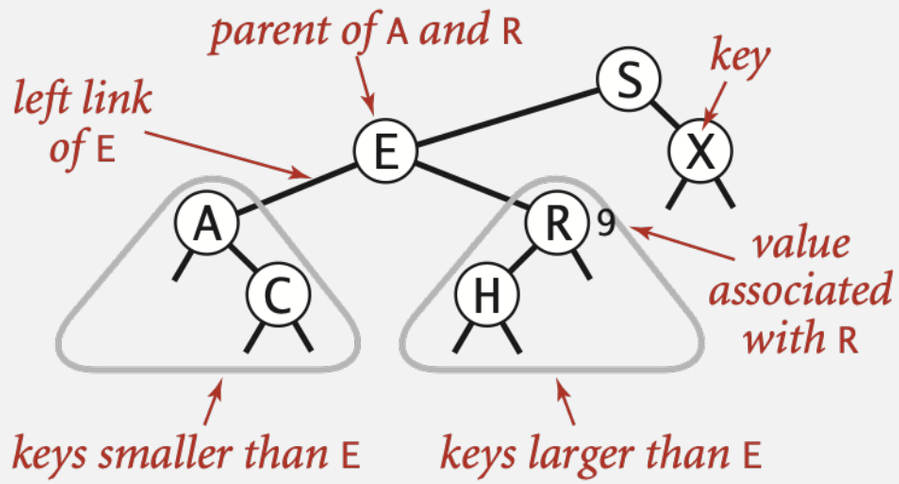
- Every node is pointed to by just one other node (parent), except the root node
- Each node has exactly two links, left and right, that point to its left child and right child



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

no duplicate keys

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

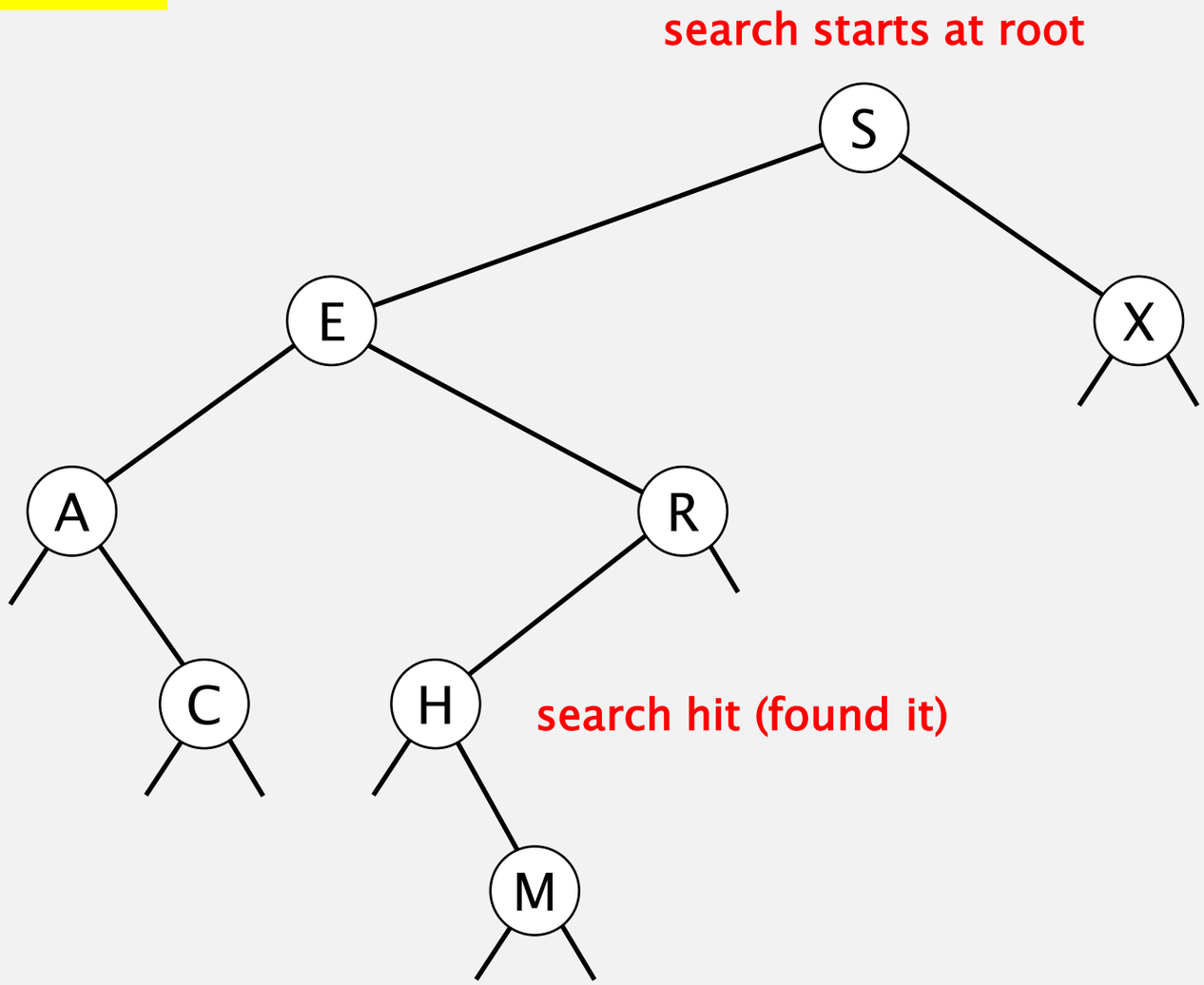


Binary search tree demo

Search.

Start at the root; If less, go left; if greater, go right; if equal, search hit.

successful search for H



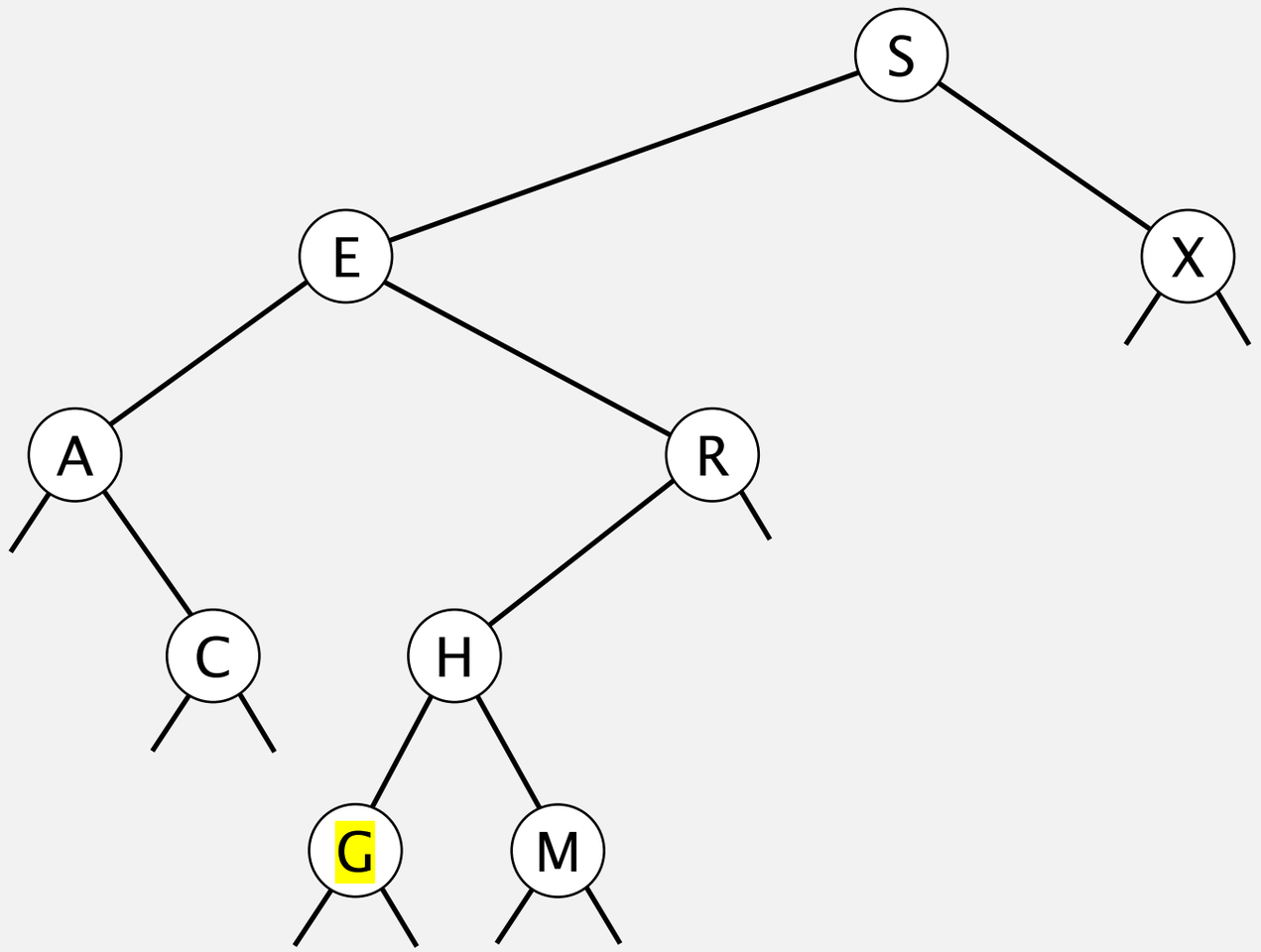
Binary search tree demo

Insert.

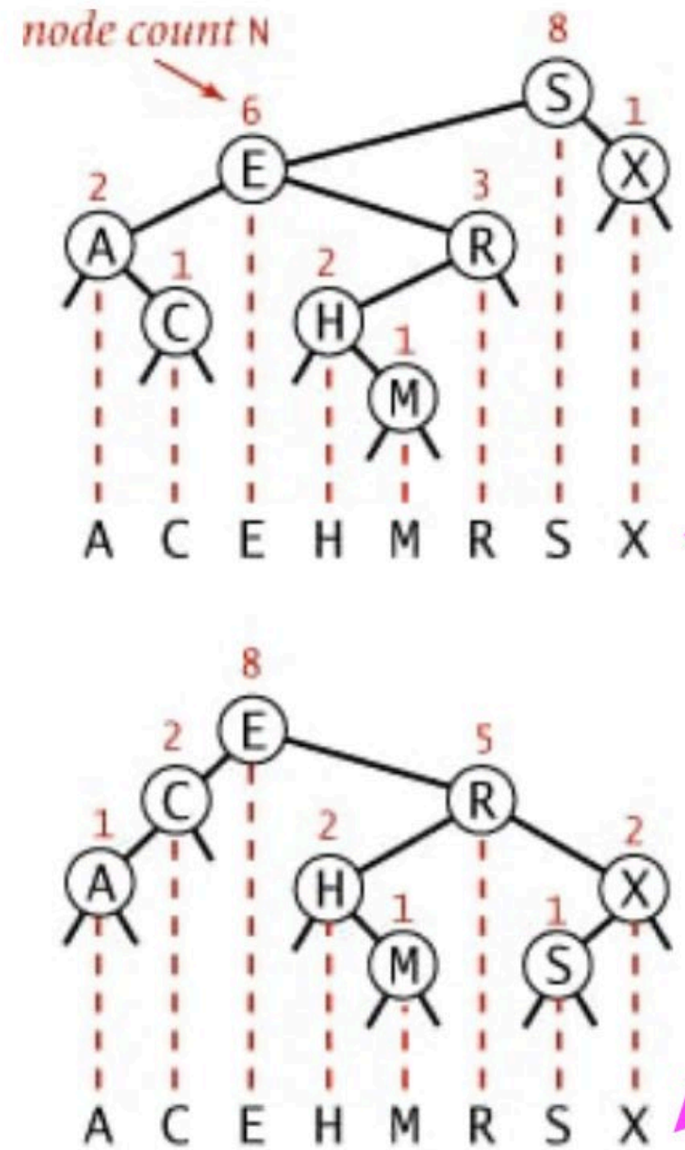
Start at the root; If less, go left; if greater, go right; if null, insert.

Intuition: search until it fails, insert at failure point.

insert G



Tree shapes could be different with the same sequence of keys



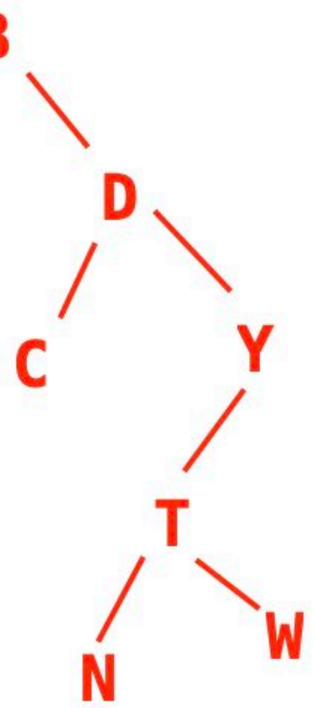
same order
different tree shape

Two BSTs that represent the same set of keys

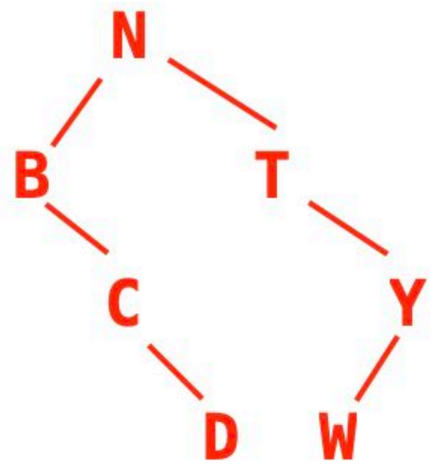
–depending on the order of insertions (building the BST)

Another Example

Insertion sequence
B, D, Y, T, W, C, N



Insertion sequence
N, B, C, T, Y, W, D



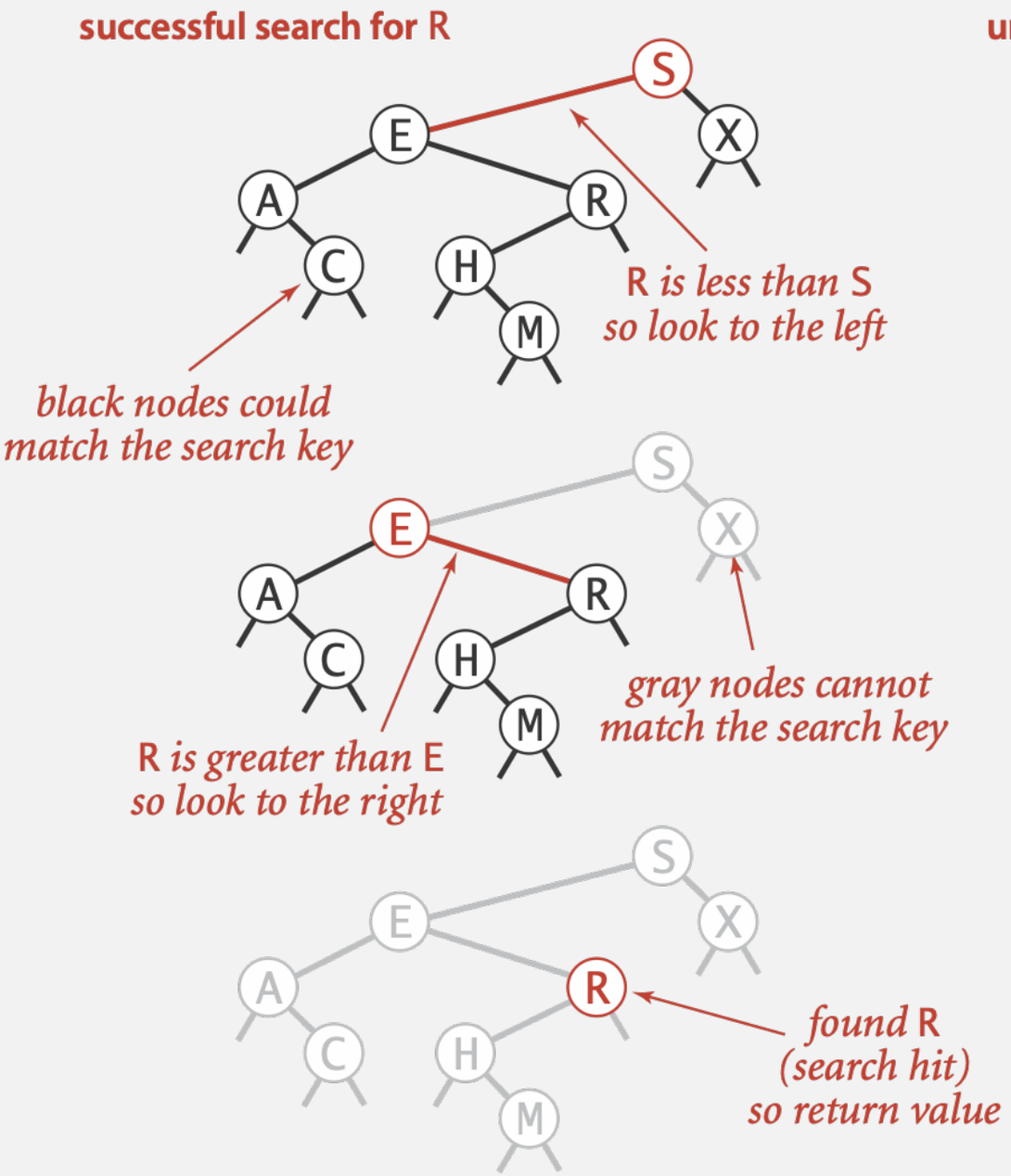
B C D N T W Y

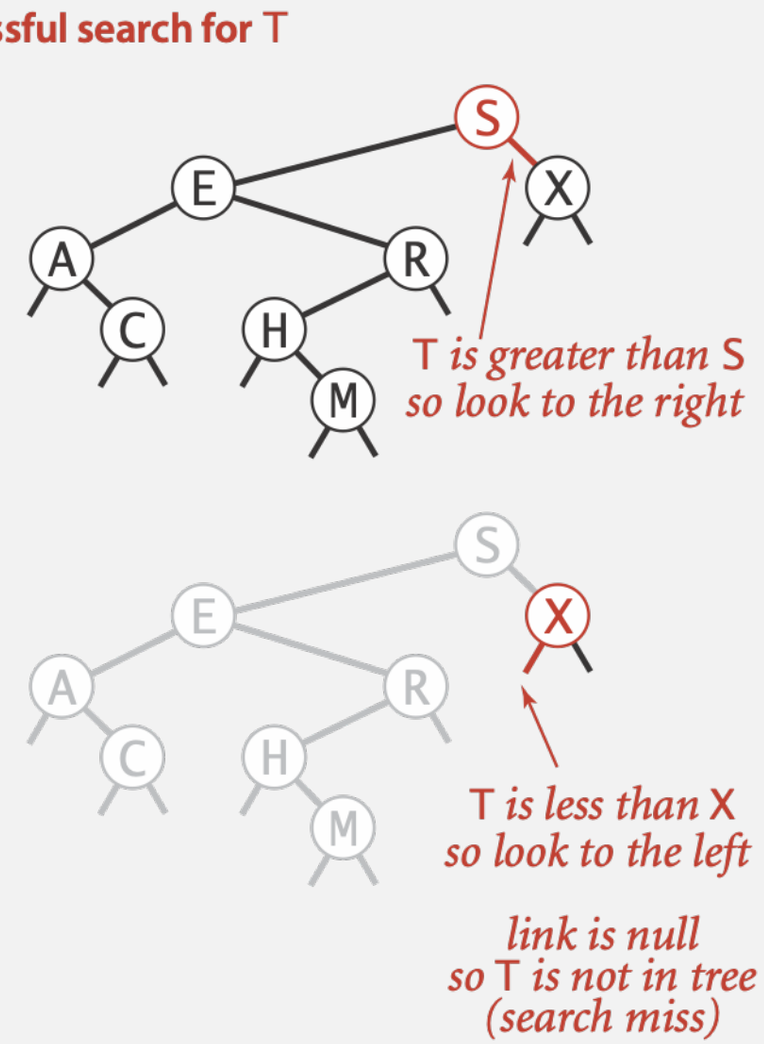
B C D N T W Y

same set of keys however different tree shapes

BST search

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





BST representation in Java

Java definition. A BST is a reference to a root Node.

(like the "first" pointer in a singly linked list)

A Node is composed of four fields:

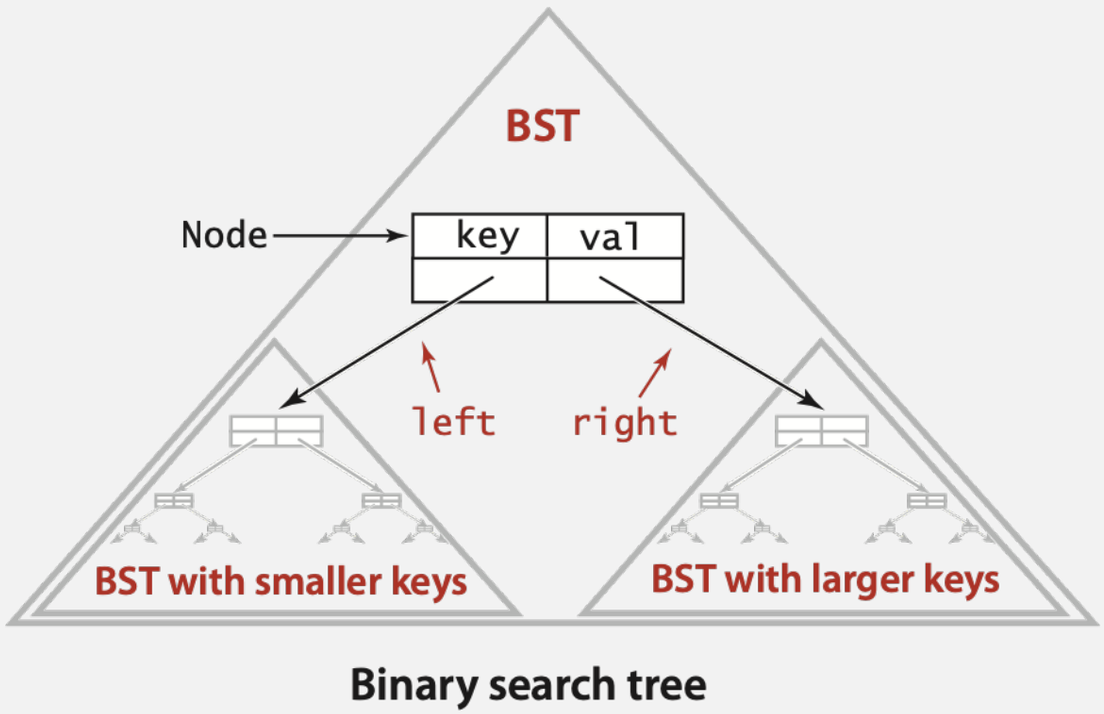
- A Key and a Value.
- A reference to the left and right subtree.

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;

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

  public void put(Key key, Value val)  //search miss -> insert at leaf
  {  /* see next slide */  }                    //search hit -> replace old val with new val

  public Value get(Key key) //search, return null if search miss
  {  /* see next slide */  }

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

  public void delete(Key key) //remove the node identified by the key
  {  /* see textbook */  }

}
```

root of BST

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; //search hit
   }
   return null;  //search miss
}
```

key in x is too big
(so look in left subtree)

key in x is too small
(so look in right subtree)

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

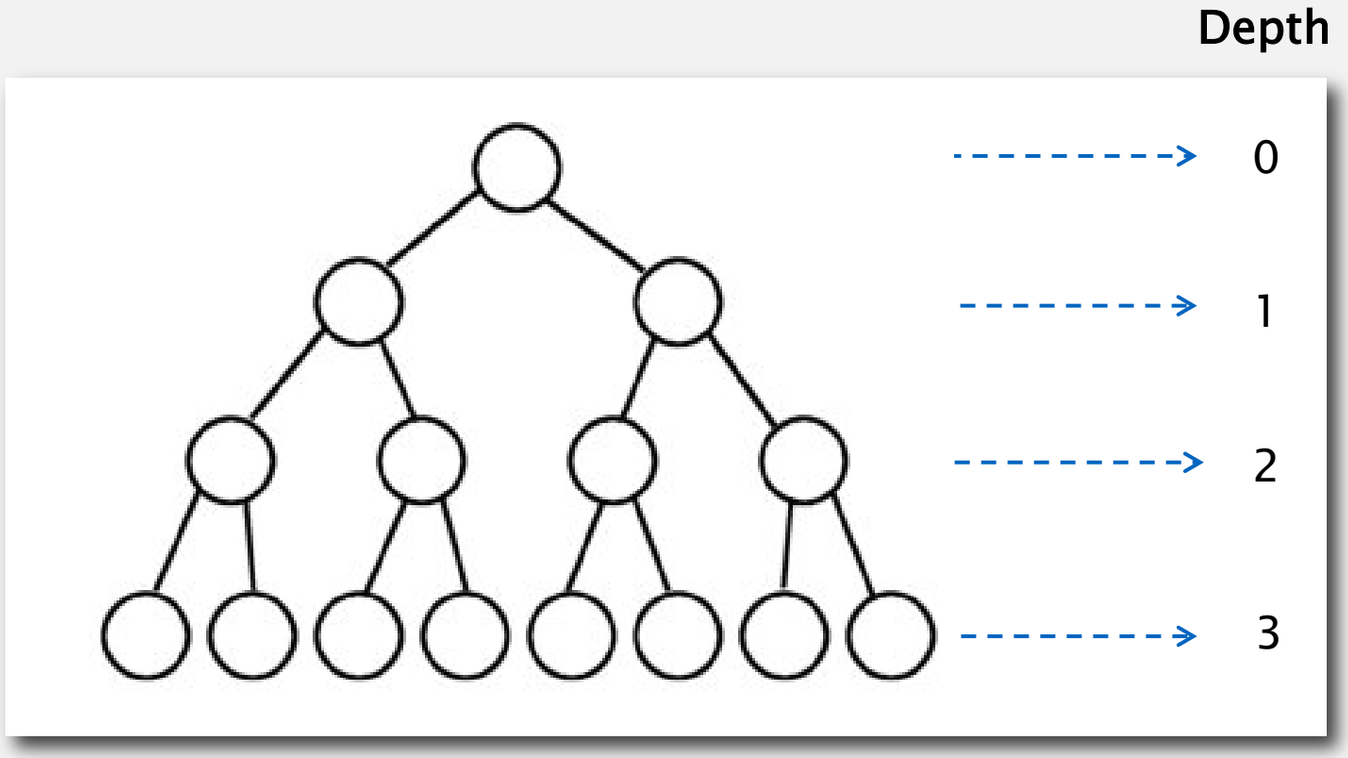
start at root node

# of links to root

Depth of a node in a tree and the height of a tree

The *depth of a node* is the number of links on the path from it to the root.

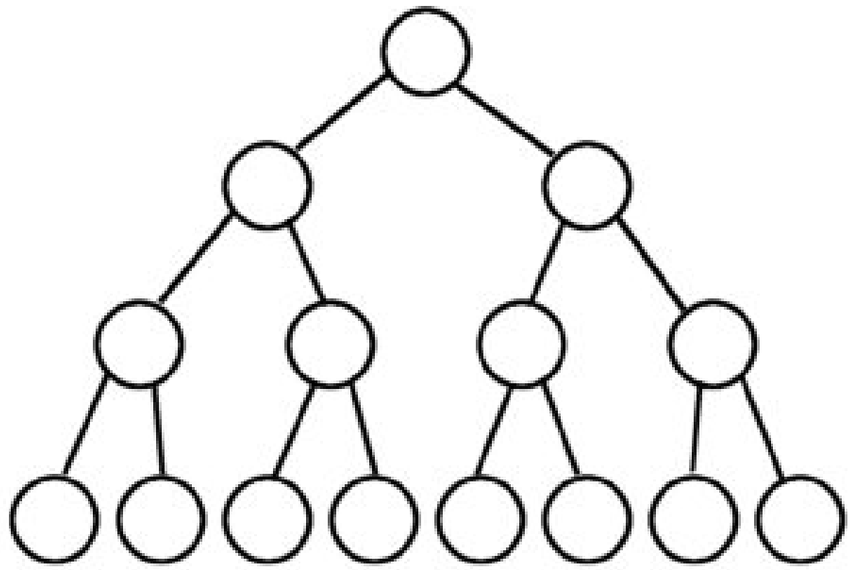
Root's depth is 0



The *height of a tree* is the maximum depth among its nodes.

Height of a binary tree

The *height (h) of a tree* is the maximum depth among its nodes.



Depth	Max Number of nodes at depth
0	2^0
1	2^1
2	2^2
3	2^3

Use the maximum number of nodes in the tree to derive its height.

The last level, deepest depth, may have up to 2^h nodes.

Let N_{max} be the maximum number of nodes in a tree of height h

$N_{max} = 2^0 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$

$N_{max} + 1 = 2^{h+1}$

$h = \log (N_{max} + 1) - 1$

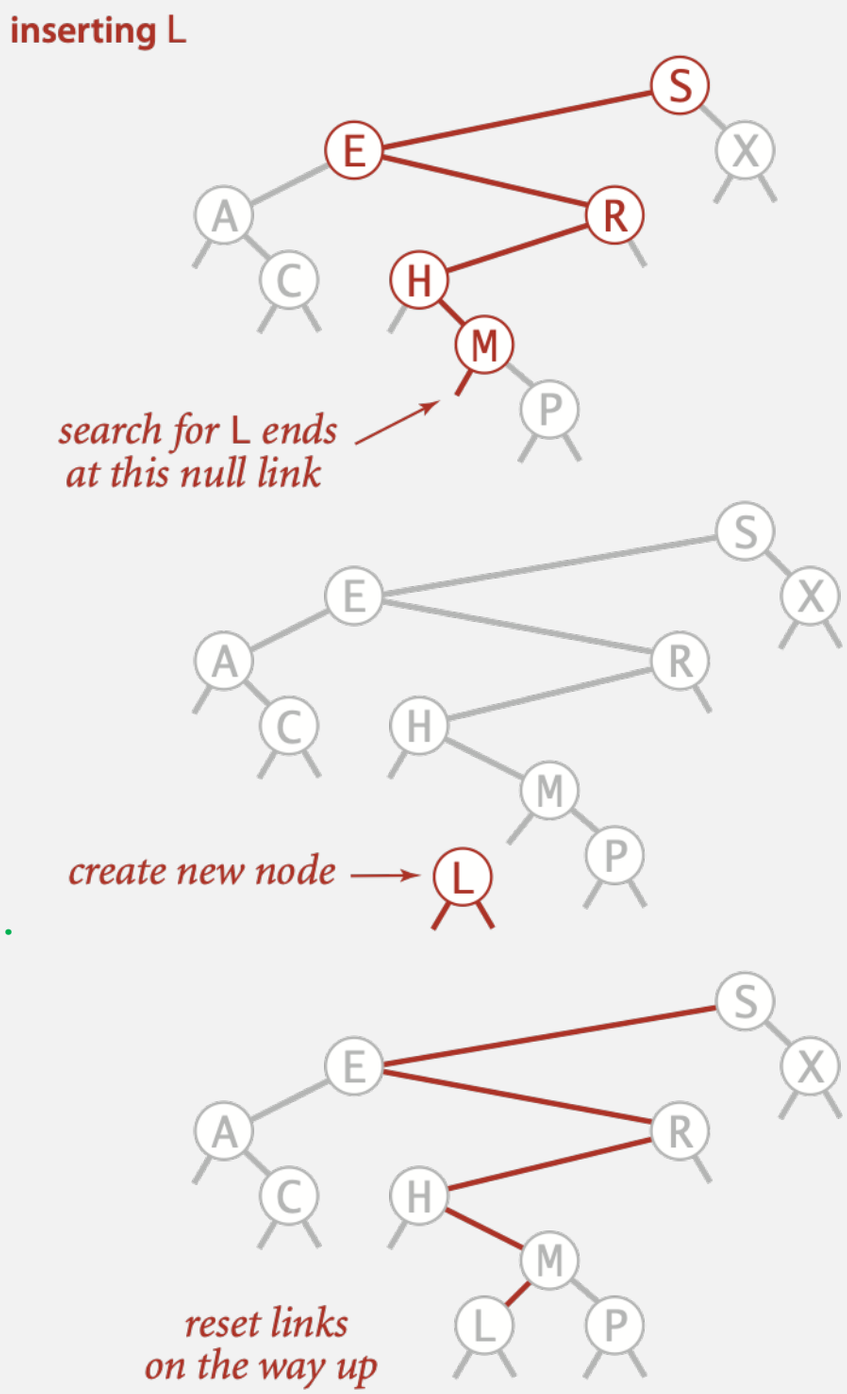
BST insert

Put. Associate value with key.

Search for key, then two cases:

- Key in tree ⇒ reset value.
- Key not in tree ⇒ add new node.

```
private Node put(Node x, Key key, Value val)
{
   // Change value to val if key in subtree rooted at x.
   // Otherwise, add new node to subtree.
   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 x.val = val;
      return x;
}
```



Insertion into a BST

BST insert: Java implementation

Put. Associate value with key.

```
public void put (Key key, Value val)
{  root = put (root, key, val);  }  //call the helper method put()

private Node put (Node x, Key key, Value val)  //recursively called
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);  //key is smaller
   else if (cmp  > 0) x.right = put(x.right, key, val);  //key is larger
   else if (cmp == 0) x.val   = val;  //key in BST, replace old val with new val
   return x;
}
```



Warning: concise but tricky code; read carefully!

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

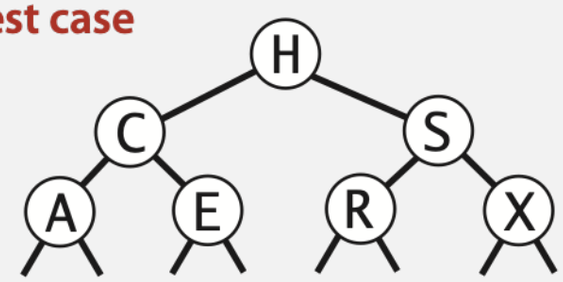
Tree shape

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

tree height + 1

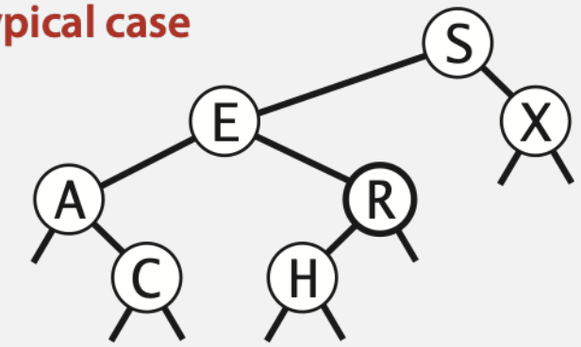
- The worst-case number of compares for search/insert depends on the tree shape:

best case

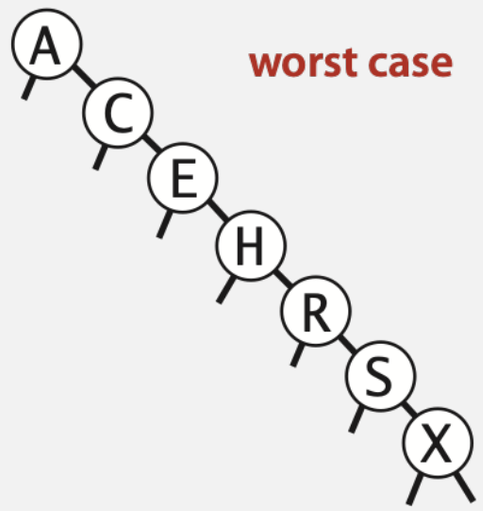


O(log n)

typical case



worst case



O(n)

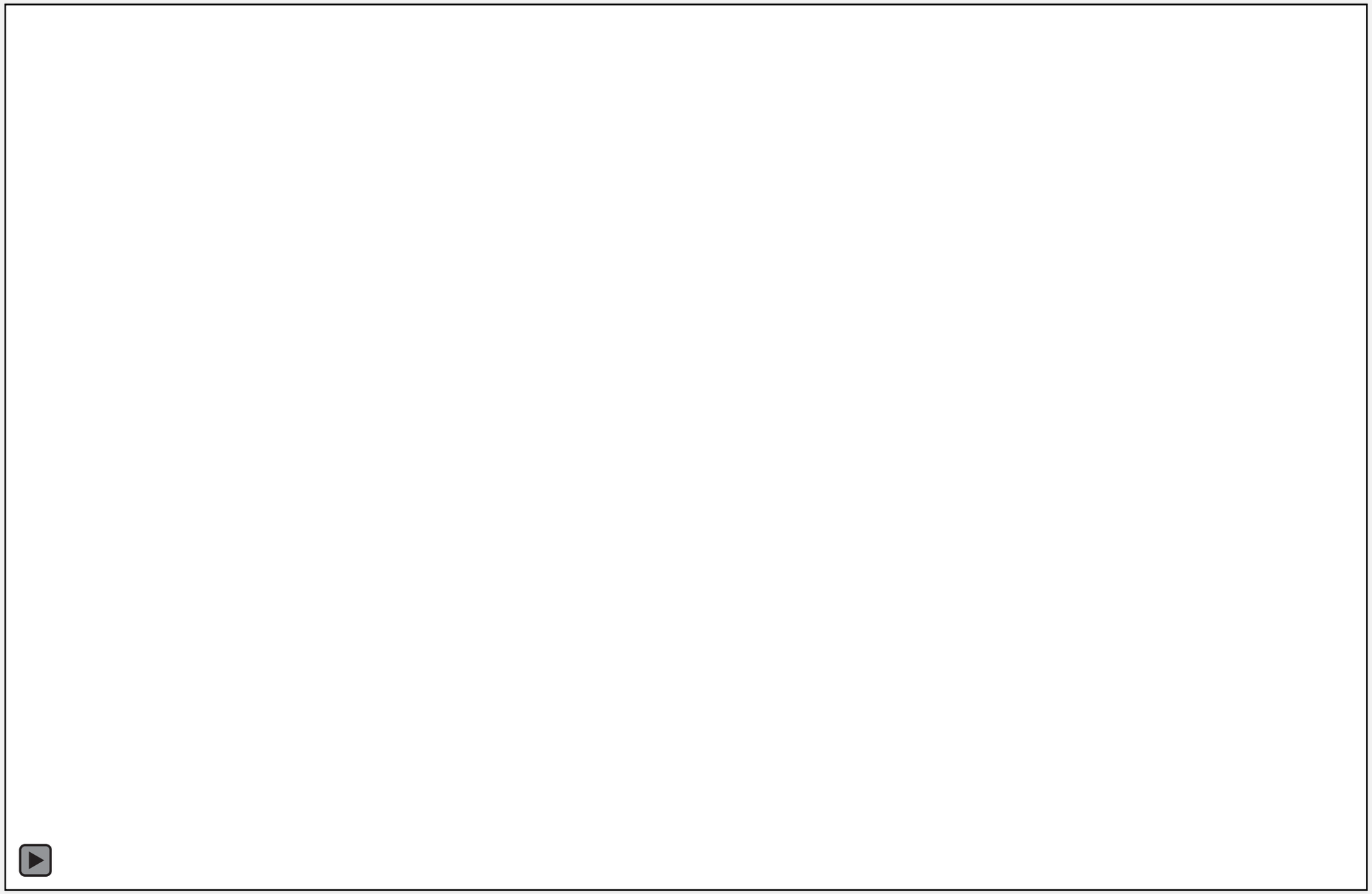
skewed to right or left

Bottom line. Tree shape depends on order of insertion.

BST insertion: random order visualization

Ex. Insert keys in random order.

- Because each node has two links, the tree tends to grow out, rather than just down
- The number of keys examined becomes a smaller and smaller fraction of the number of keys in the tree as the tree size



Binary search trees: quiz 1

Suppose that you insert n keys in random order into a BST.

What is the expected height of the resulting BST?

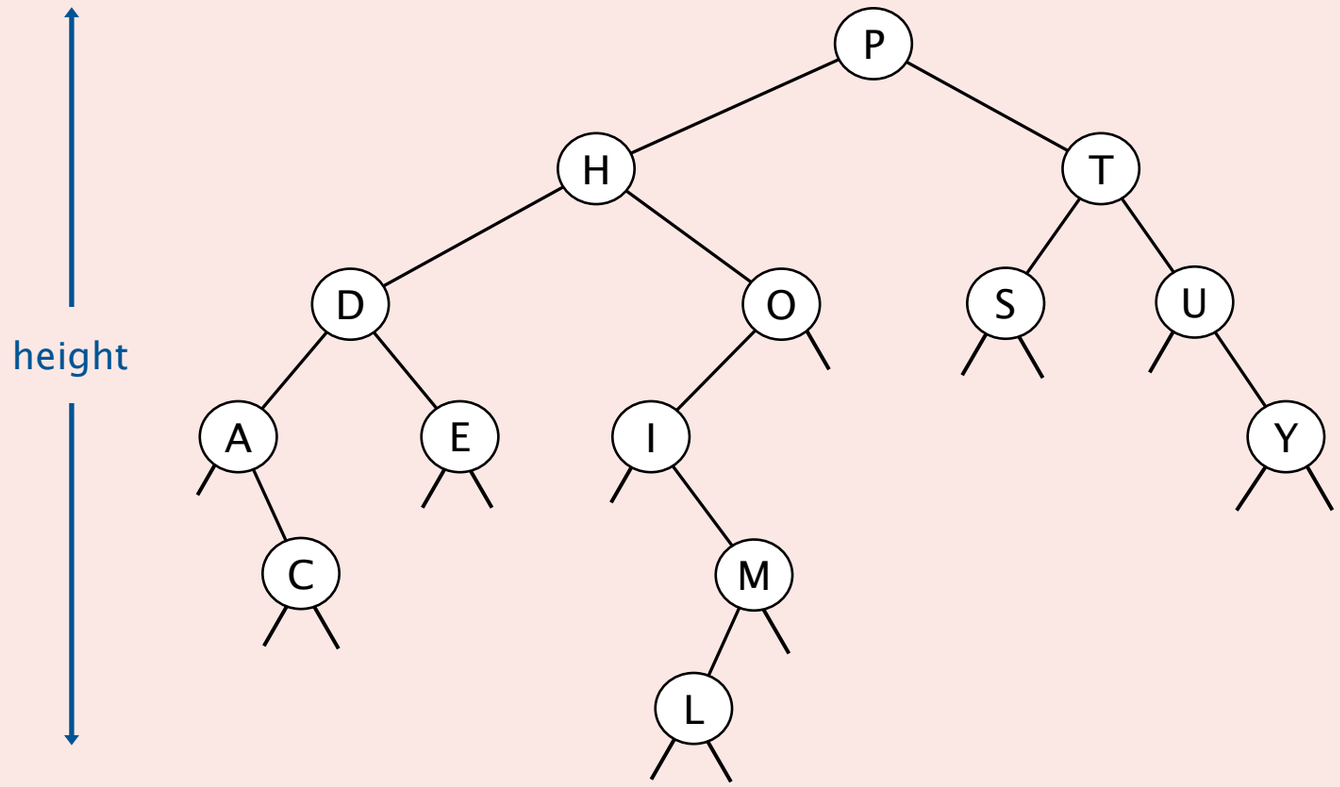
A. $\sim \lg n$ → $log_2 n$

B. $\sim \ln n$ → $log_e n$, $e = 2.718$

C. $\sim 2 \lg n$

D. $\sim 2 \ln n$ → *#compares*

E. $\sim 4.31107 \ln n$



→ *expected height not guaranteed, as data are randomly inserted*

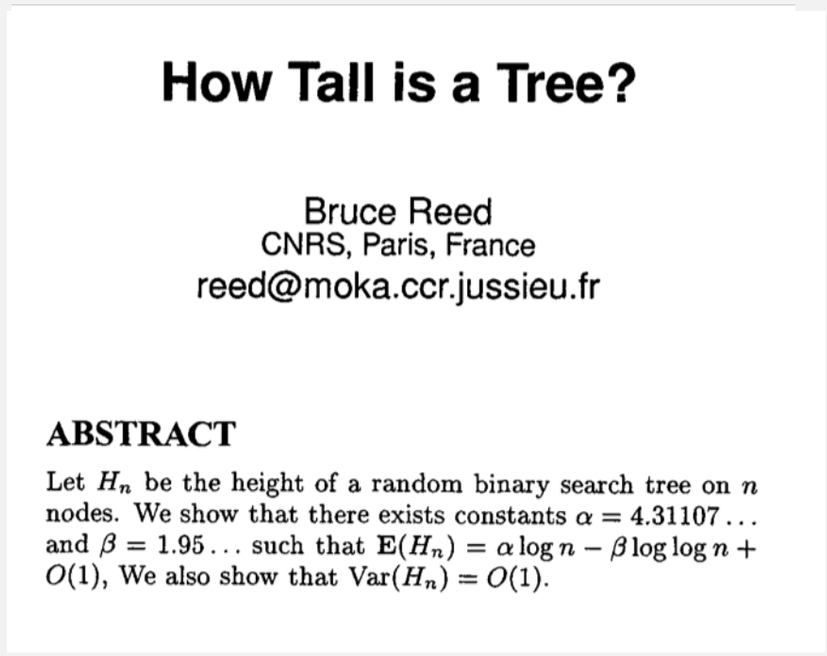
BSTs: mathematical analysis

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 $\sim 4.311 \ln n$.

expected depth of function-call stack in quicksort



How Tall is a Tree?

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

ABSTRACT

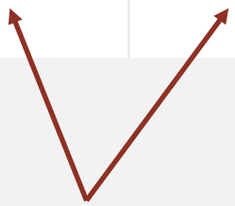
Let H_n be the height of a random binary search tree on n nodes. We show that there exists constants $\alpha = 4.31107\ldots$ and $\beta = 1.95\ldots$ such that $\mathbf{E}(H_n) = \alpha \log n - \beta \log \log n + O(1)$, We also show that $\mathrm{Var}(H_n) = O(1)$.

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

[exponentially small chance when keys are inserted in random order]

ST implementations: summary

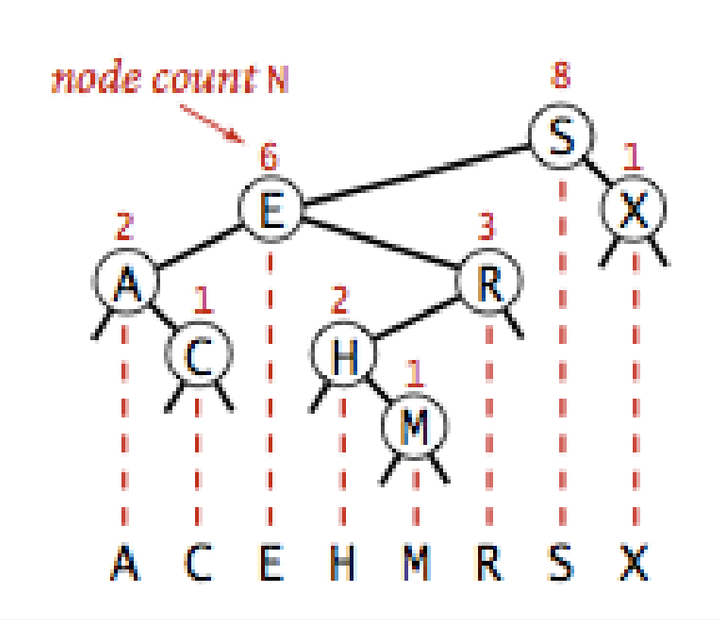
implementation	no duplicates allowed				operations on keys
	guarantee		average case		
	search	insert	search hit	insert	
sequential search (unordered list)	n	n	n	n	equals()
binary search (ordered array)	$\log n$	n	$\log n$	n	compareTo()
BST	n	n	$\log n$	$\log n$	compareTo()



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

1) Cannot assume all data is available up front

2) ~4.311 ln n is not guaranteed because there is an assumption about the input distribution



Binary Search Trees

- *Symbol Table*
- *BSTs*
- *Iteration*
- *Ordered operations*
- *Deletion*

Binary search trees: quiz 2

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

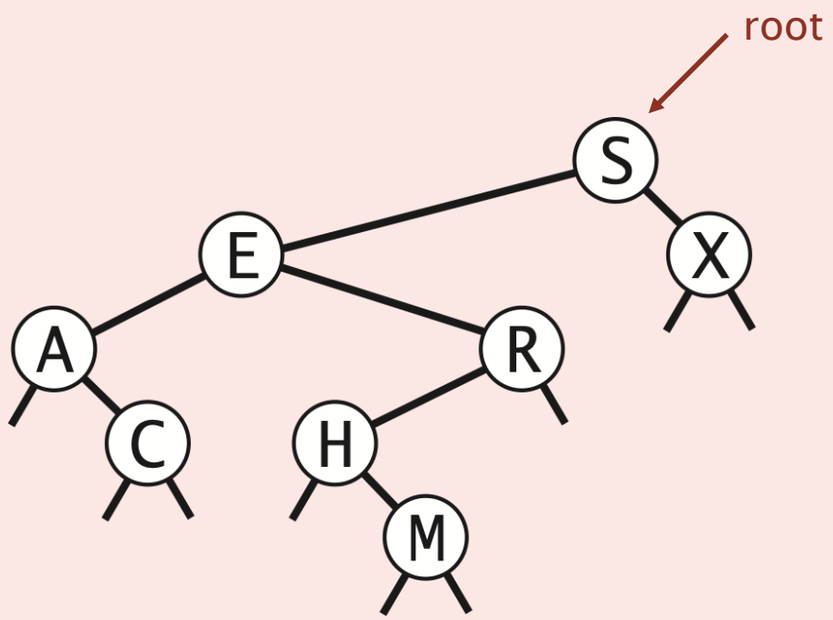
```
private void traverse(Node x)
{
   if (x == null) return;  //reach the leaf node
   traverse(x.left);  //recursively called with left subtree
   StdOut.println(x.key);
   traverse(x.right);  //recursively called with right subtree
}
```

A. A C E H M R S X in-order

B. S E A C R H M X pre-order

C. C A M H R E X S post-order

D. S E X A R C H M level-order

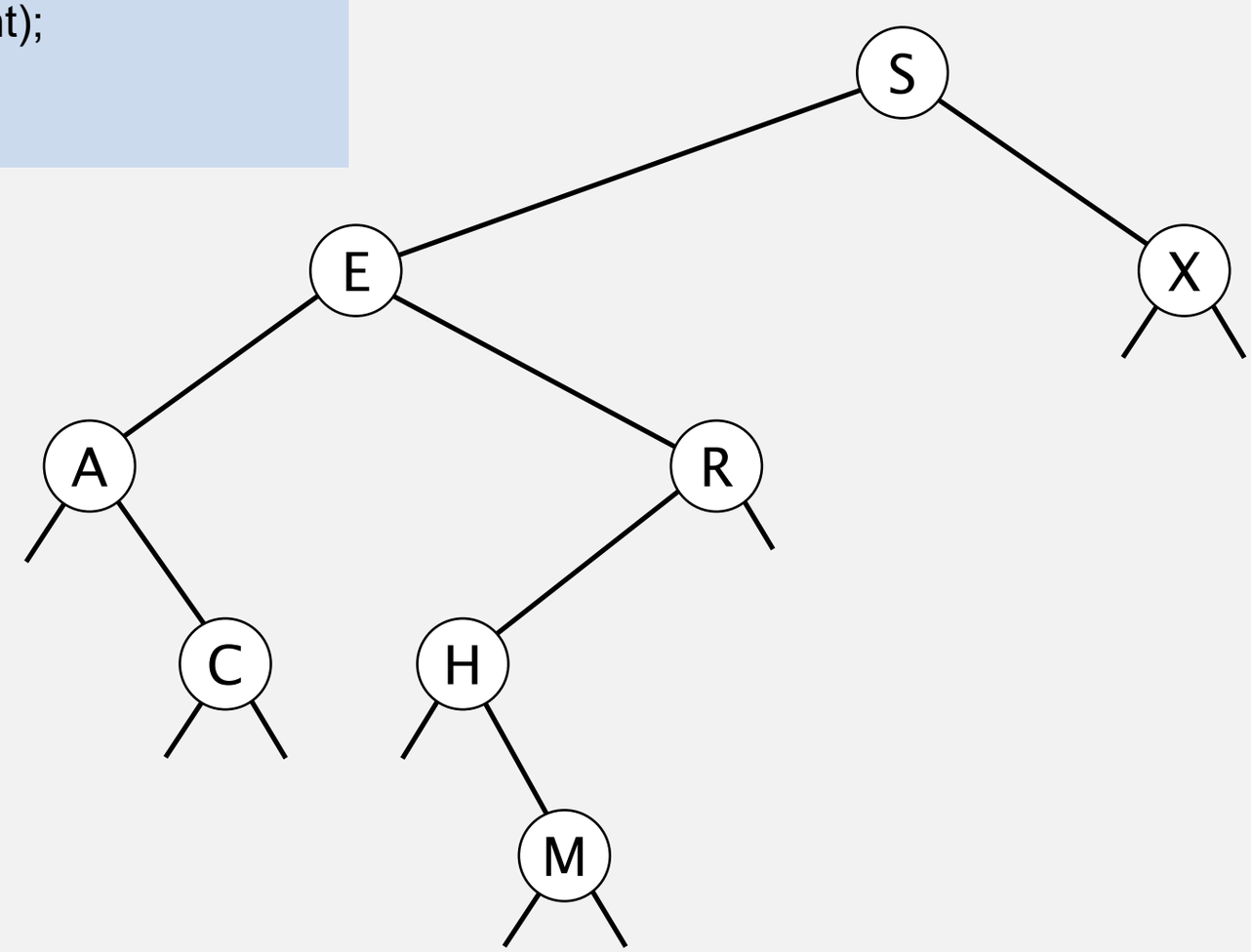


Inorder traversal


```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
        print H
        inorder(M)
          print M
          done M
        done H
      print R
      done R
  done E
print S
inorder(X)
  print X
  done X
done S
```

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





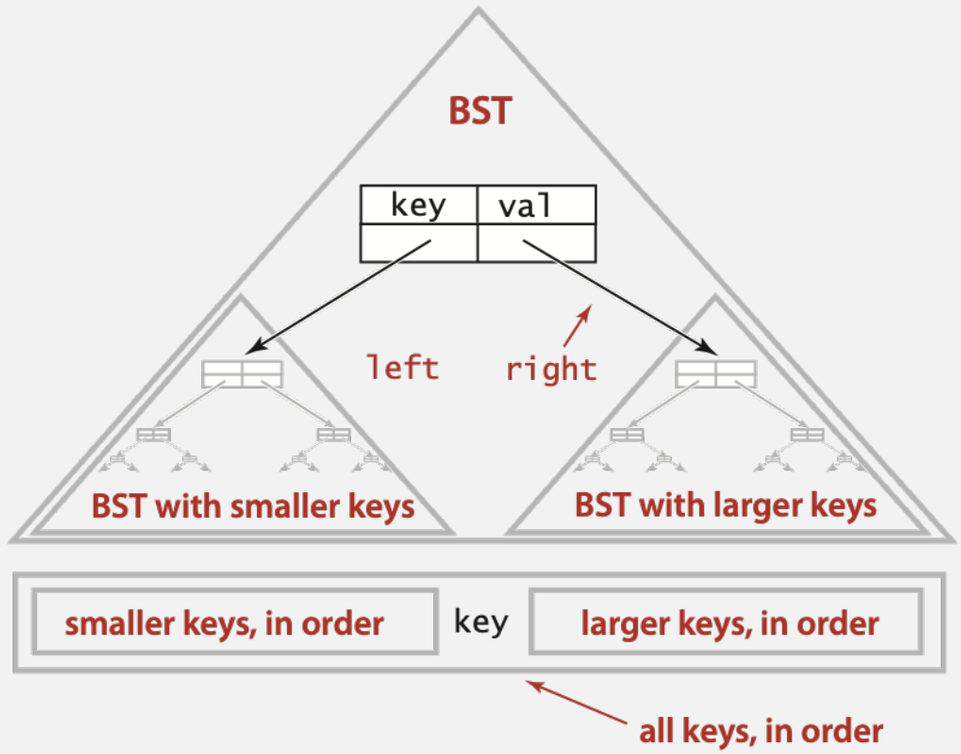
output: **A C E H M R S X**

Inorder traversal

1. Traverse left subtree.
2. Enqueue key.
3. 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) //recursively called
{
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```



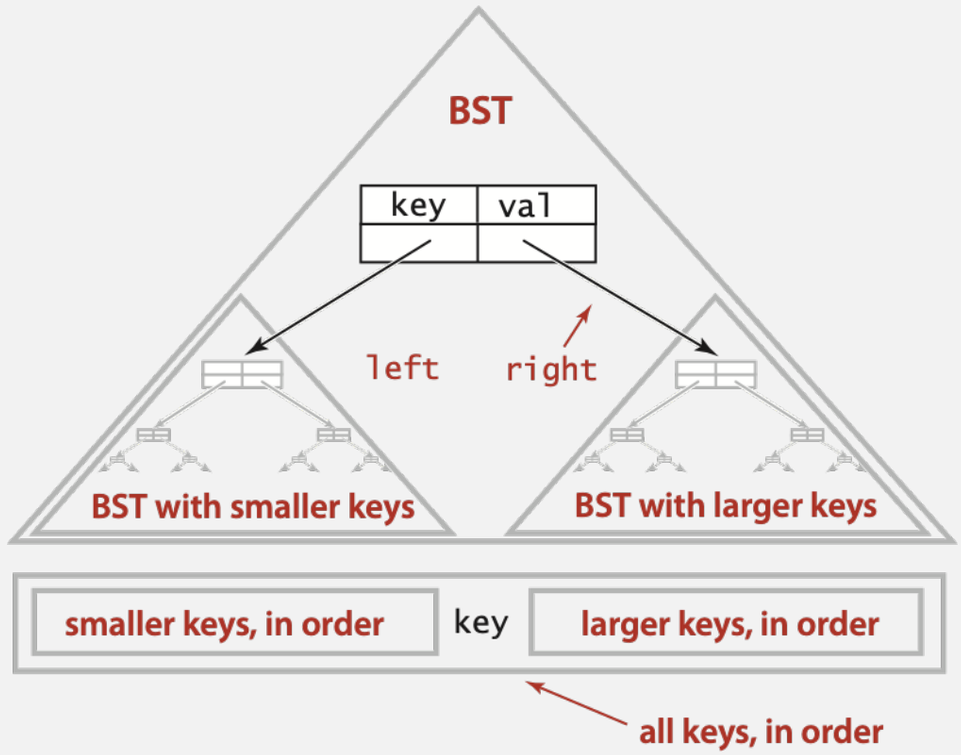
Property. Inorder traversal of a BST yields keys in ascending order.

Preorder traversal

1. Enqueue key.
2. Traverse left subtree.
3. Traverse right subtree.

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

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

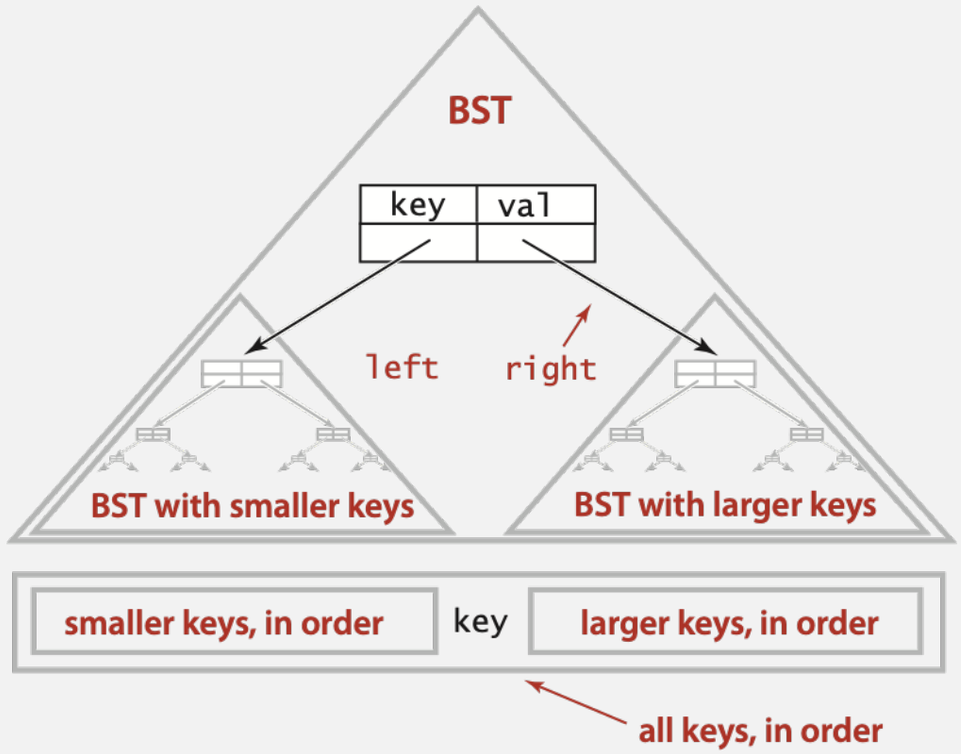


Postorder traversal

1. Traverse left subtree.
2. Traverse right subtree.
3. Enqueue key.

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

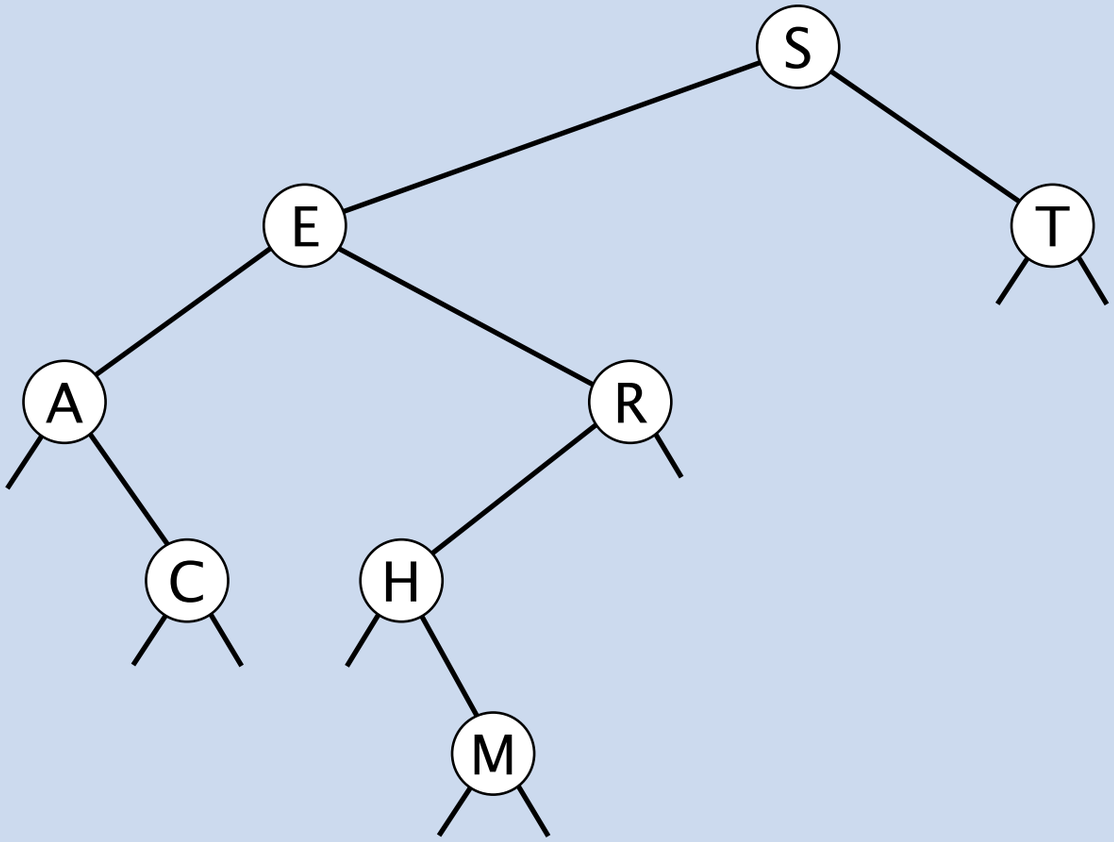
```
private void postorder(Node x, Queue<Key> q)
{
   if (x == null) return;
   postorder(x.left, q);  //left child
   postorder(x.right, q);   //right child
   q.enqueue(x.key);  //parent node
}
```



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**

Binary Search Trees

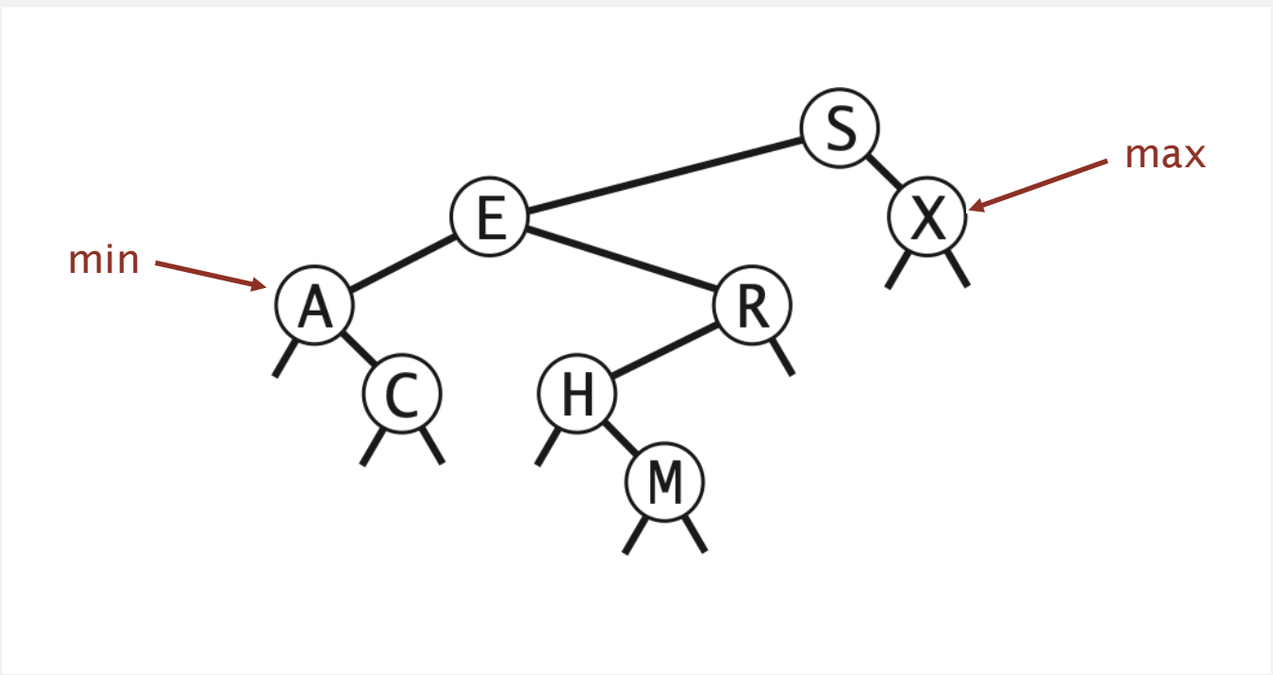
- *Symbol Table*
- *BSTs*
- *Iteration*
- *Ordered operations*
- *Deletion*

Minimum and maximum

Minimum. Smallest key in BST.

Maximum. Largest key in BST.

BST is a binary tree in symmetric order



Q. How to find the min / max?

A. Follow leftmost/rightmost spine

Implementing min() and max()


```
public Key max() {
    if (isEmpty()) throw new NoSuchElementException("calls max() with empty symbol table");
    return max(root).key;
}

private Node max(Node x) {
    if (x.right == null) return x;
    else                 return max(x.right);
}
```

recursive call to right subtree

```
public Key min() {
    if (isEmpty()) throw new NoSuchElementException("calls min() with empty symbol table");
    return min(root).key;
}

private Node min(Node x) {
    if (x.left == null) return x;
    else                return min(x.left);
}
```

recursive call to left subtree

Floor and ceiling

LO 5.9

Floor. Largest key in BST $\leq$ query key.

Ceiling. Smallest key in BST $\geq$ query key.

Q. How do we find the floor and ceiling?

Computing the floor

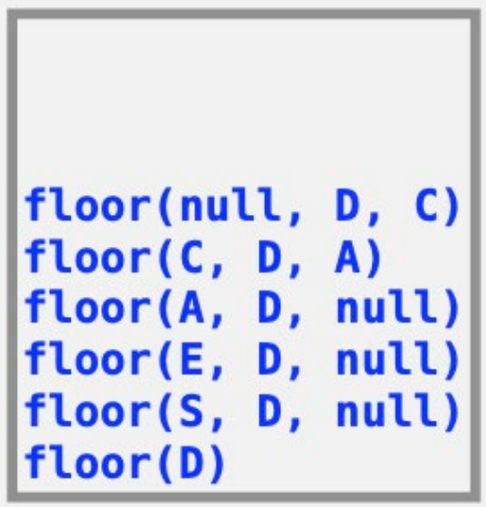
Floor. Largest key in BST $\leq k$?

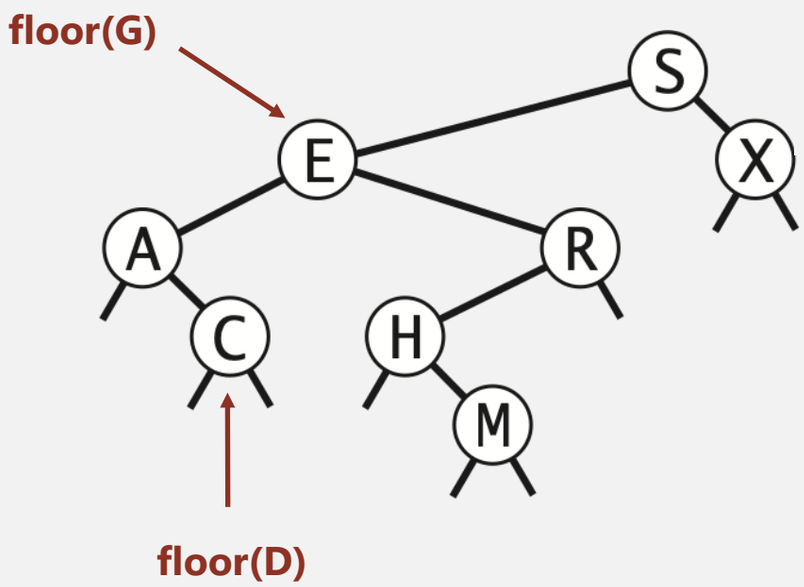
Key idea.

- To compute `floor(key)`, search for `key`.
- On search path, must encounter `floor(key)` and `ceiling(key)`. Why?

Interchanging right and left gives ceiling().

- if key is less than key in node, floor(key) must be in left subtree
- if key is larger than key in node, floor(key) may be in left subtree of the right subtree





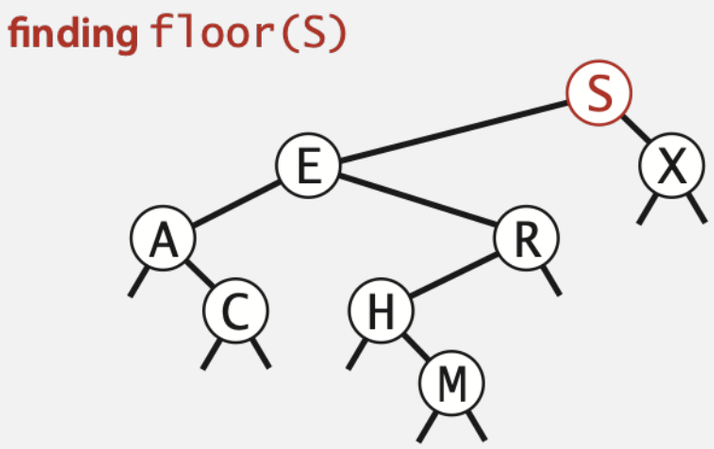


Computing the floor

Floor. Largest key in BST $\leq k$?

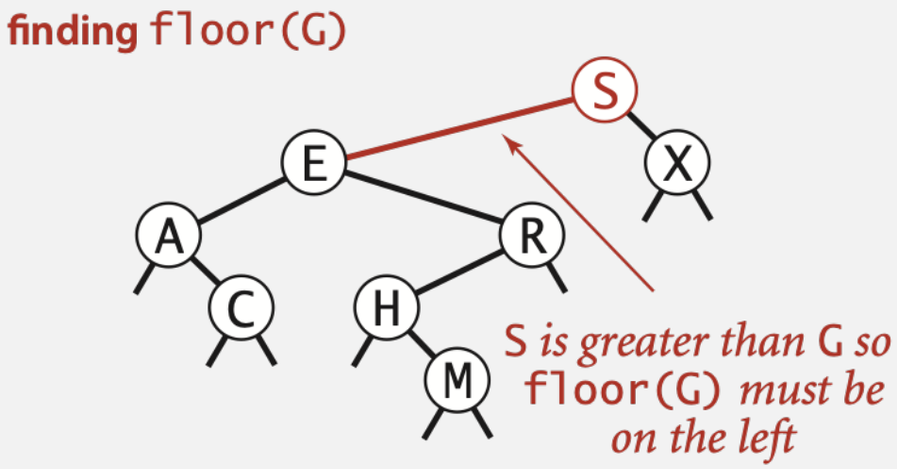
Case 1. [key in node $x = k$]

The floor of k is k.



Case 2. [key in node $x > k$]

The floor of k is in the left subtree of x.

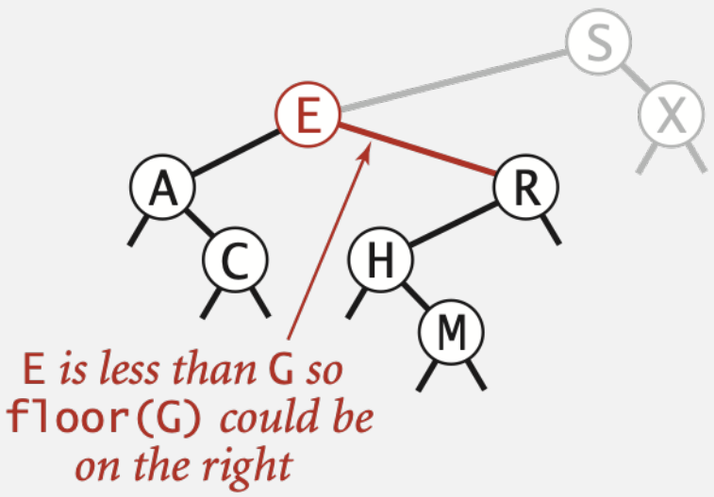


Case 3. [key in node $x < k$]

The floor of k can't be in left subtree of x:

it is either in the right subtree of x or

it is the key in node x.



Computing the floor


```
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;
}
```

initially null

key in node is too big

(so look in left subtree)

smaller than all keys
in right subtree

key in node is best candidate for floor

(but maybe better one in right subtree)

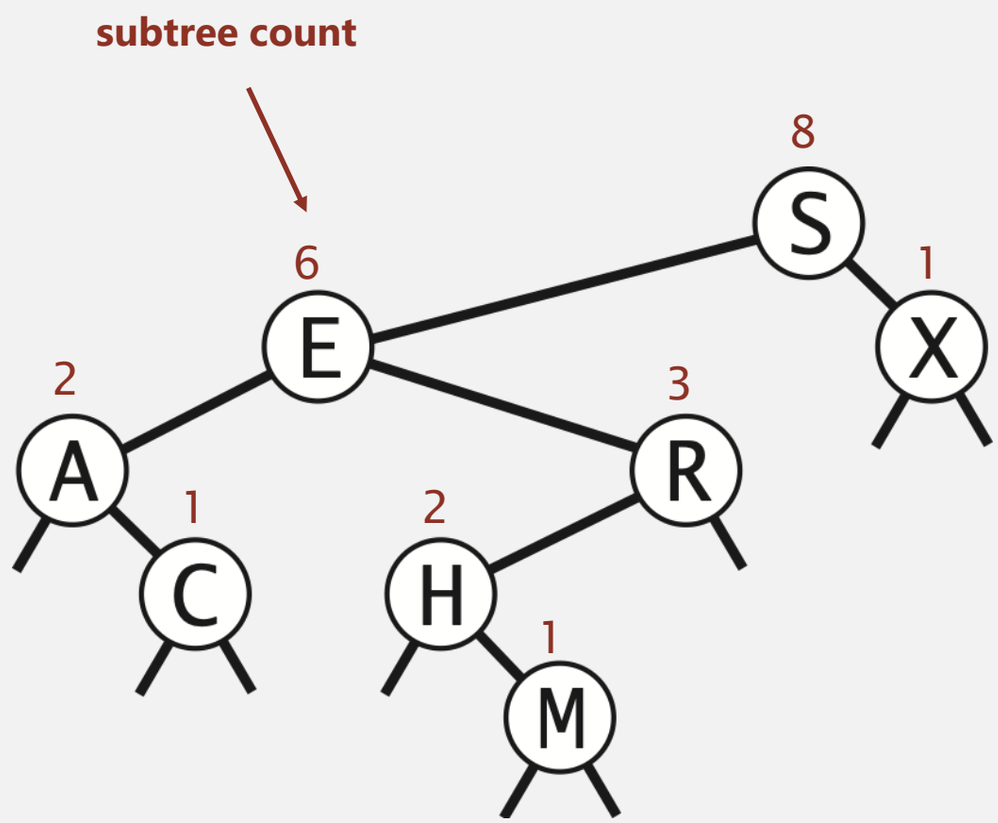
Rank and select

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. //including itself



BST implementation: subtree counts

LO 5.9

```
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

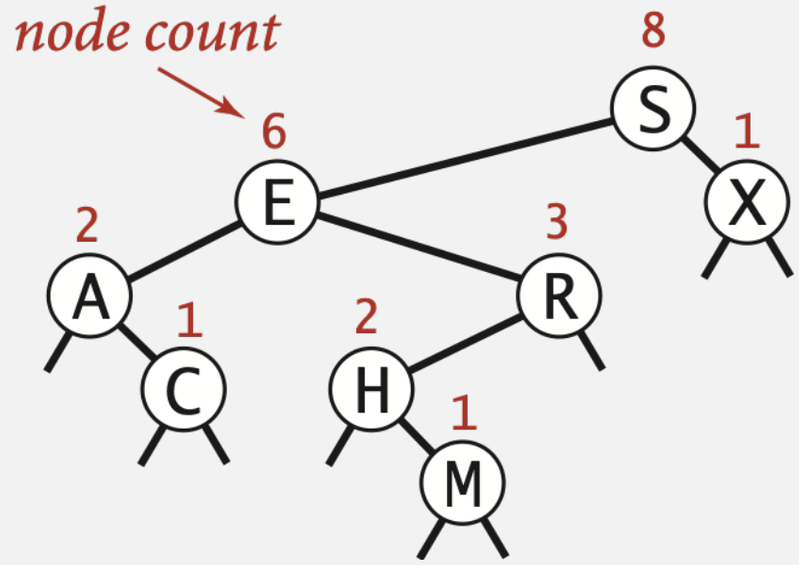
```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val, 1);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val; //key exists, replace it with the new value
   x.count = 1 + size(x.left) + size(x.right);  //update the size the way back

   return x;
}
```

initialize subtree count to 1

Computing the rank

Rank. How many keys $< key$?



Case 1. [$key <$ key in node]

- Keys in left subtree? *count*
- Key in node? 0
- Keys in right subtree? 0

Case 2. [$key >$ key in node]

- Keys in left subtree? *all*
- Key in node. 1
- Keys in right subtree? *count*

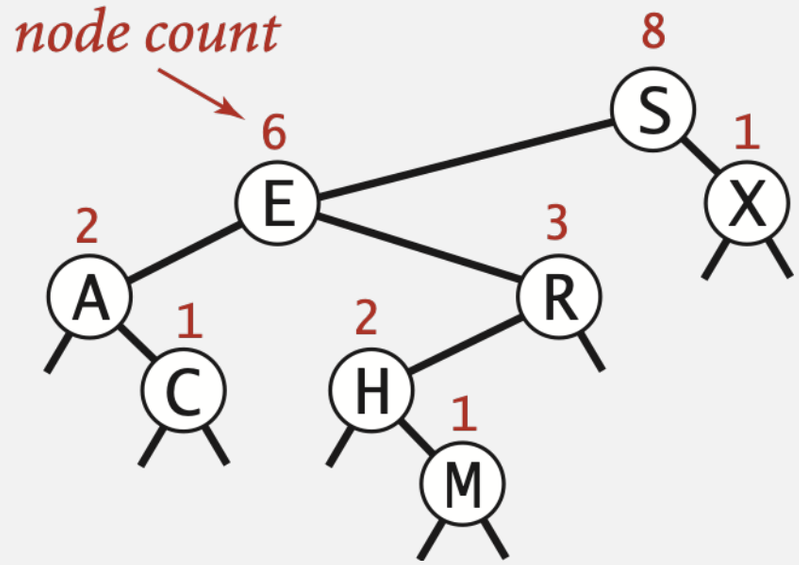
Case 3. [$key =$ key in node]

- Keys in left subtree? *count*
- Key in node. 0
- Keys in right subtree? 0

Rank

Rank. How many keys $< key$?

Easy recursive algorithm (3 cases!)




```
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);   //key is less than key at node
   else if (cmp  > 0) return 1 + size(x.left) + rank(key, x.right);  //key is greater than key at node
   else if (cmp == 0) return size(x.left);  //key is equal to key at node

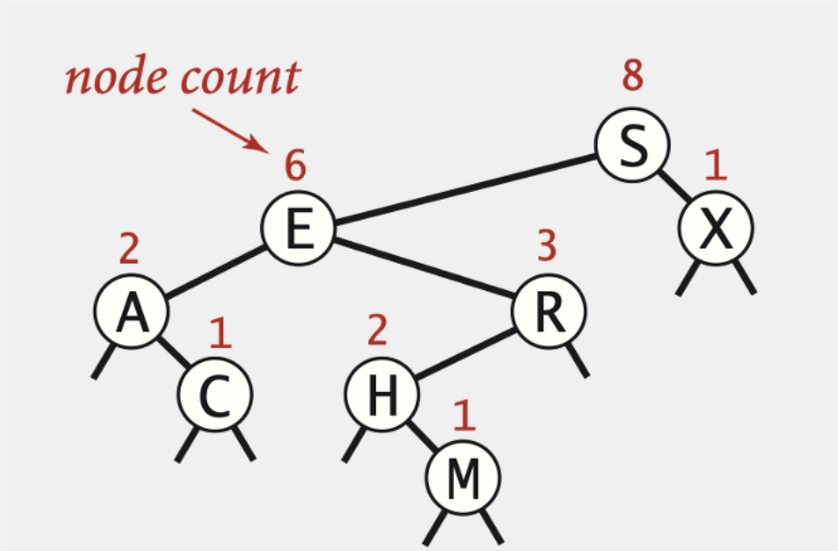
}
```

Examples - Rank

rank(B) = 1

rank(B, null)	retun 0
rank(B, C)	
rank(B, A)	1 + size(A.left) + rank(B, C)
rank(B, E)	
rank(B, S)	

Call stack



rank(F) = 3

rank(F, null)	retun 0
rank(F, H)	
rank(F, R)	
rank(F, E)	1 + size(E.left) + rank(F, R)
rank(F, S)	

Call stack

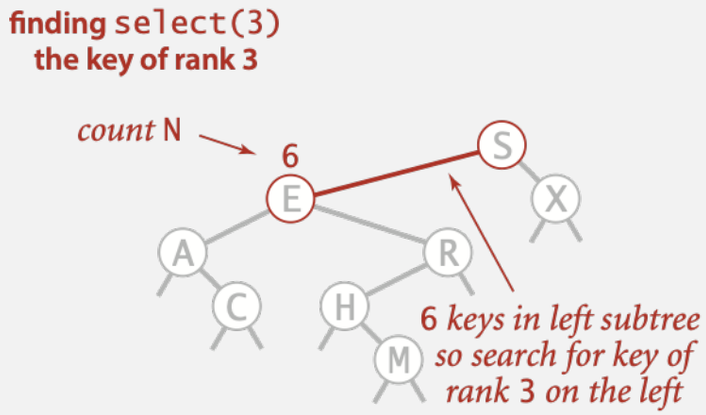
rank(I) = 4

Rank(I, null)	return 0
rank(I, M)	
rank(I, H)	1 + size(H.left) + rank(I, M)
rank(I, R)	
rank(I, E)	1 + size(E.left) + rank(I, R)
rank(I, S)	

Call stack

Select. Key in BST of rank.

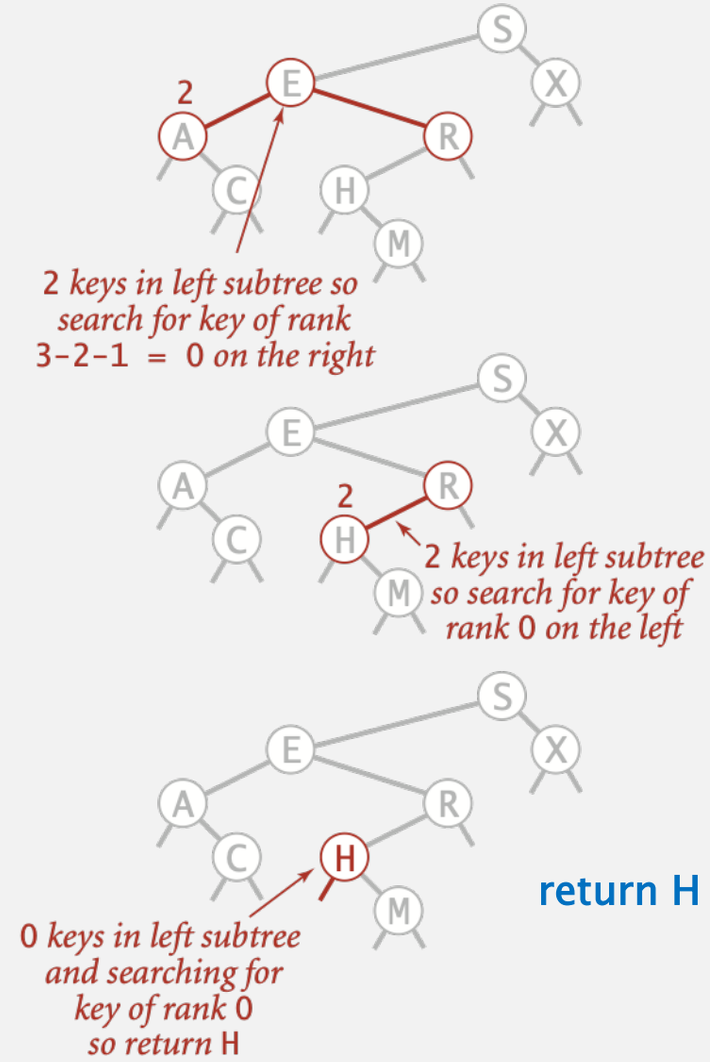





```
public Key select(int k)
{  return select(root, k); }

private Key select(Node x, int k)
{
   if (x == null) return null;
   int t = size(x.left);
   if      (t  > k) return select(x.left,  k); // # of keys in left is > k
   else if (t  < k) return select(x.right, k-t-1);  //look for keys in the right
   else if (t == k) return x.key;
}
```

call stack	
select(H, 0)	t = 0 (t = k, return H)
select(R, 0)	t = 2 (> 0, left subtree next)
select(E, 3)	t = 2 (< 3, right subtree next)
select(S, 3)	t = 6 (> 3, left subtree next)



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.

Binary Search Trees

- *Symbol Table*
- *BSTs*
- *Iteration*
- *Ordered operations*
- *Deletion*

ST implementations: summary

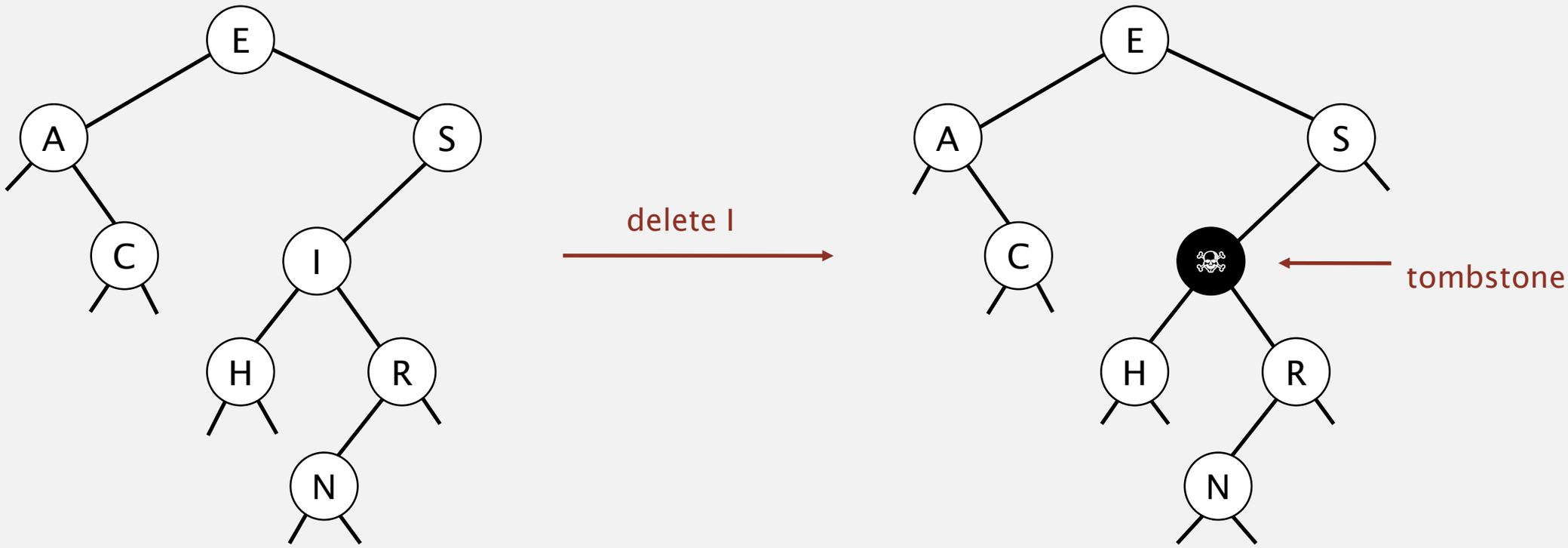
implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search(unordered list)	n	n	n	n	n	n		equals()
binary search(ordered array)	$\log n$	n	n	$\log n$	n	n	✓	compareTo()
BST	n	n	n	$\log n$	$\log n$	?	✓	compareTo()

Next. 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 search (but don't consider it equal in search).



Cost. $\sim 2 \ln n'$ per insert, search, and delete (if keys in random order), where n' is the number of key-value pairs ever inserted in the BST.

Unsatisfactory solution. Tombstone (memory) overload.

Deleting the minimum

To delete the minimum key:

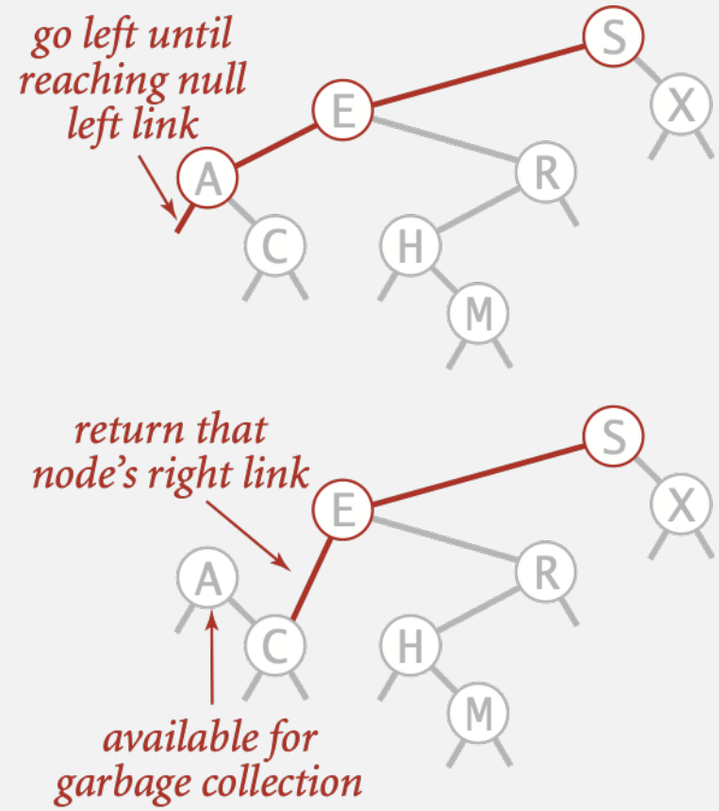
- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

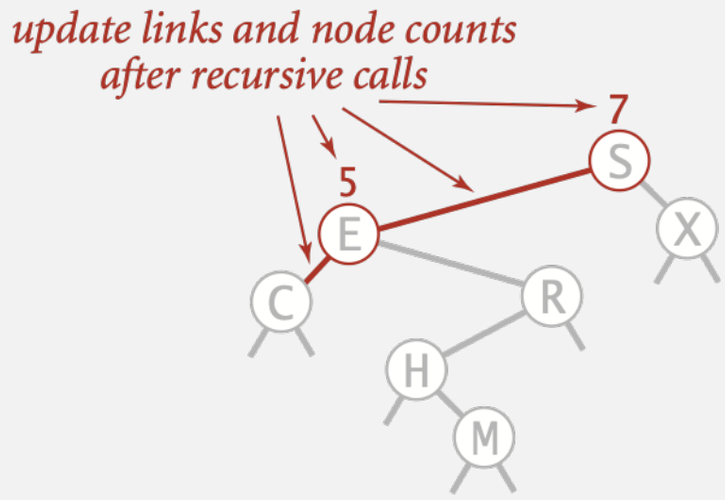
deleteMin(A)	left is null, return C, update size
deleteMin(E)	E.left = deleteMin(A) = C, update size
deleteMin(S)	S.left = deletMin(E) = E, update size

Call stack

```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right; //key at node is the min
   x.left = deleteMin(x.left); //recursively goes down to left subtree
   x.count = 1 + size(x.left) + size(x.right); //update count the way up
   return x;
}
```

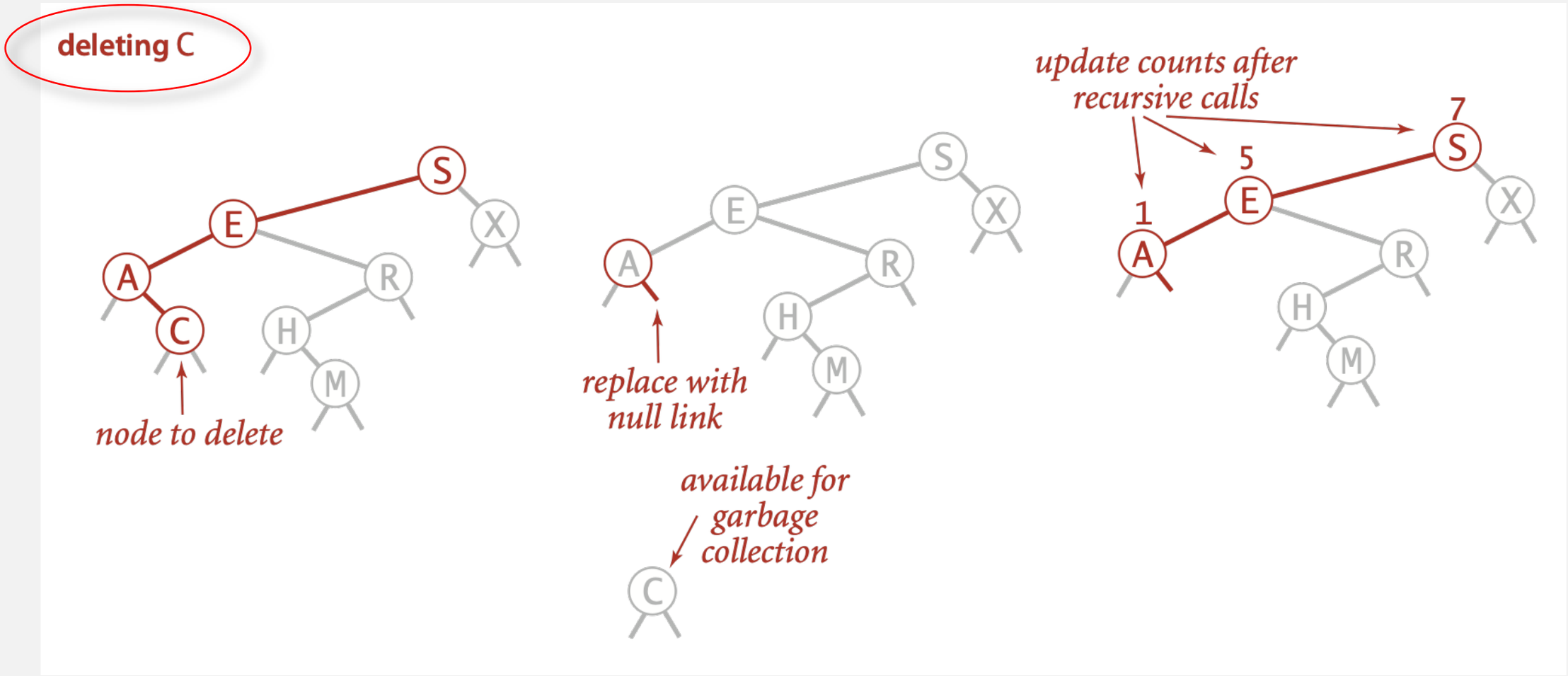




Hibbard deletion

To delete a node with key k: search for node t containing key k.

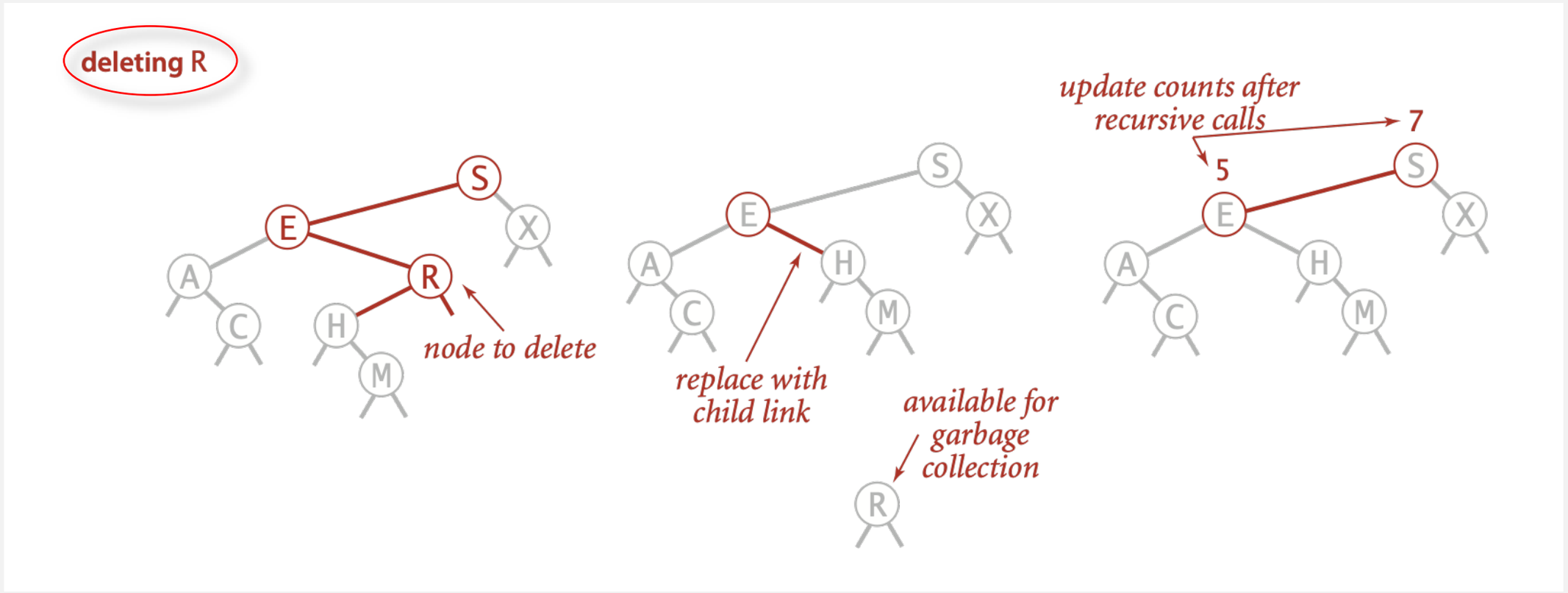
Case 0. [0 children] Delete t by setting parent link to null.



Hibbard deletion

To delete a node with key k: search for node t containing key k.

Case 1. [1 child] Delete t by replacing parent link.

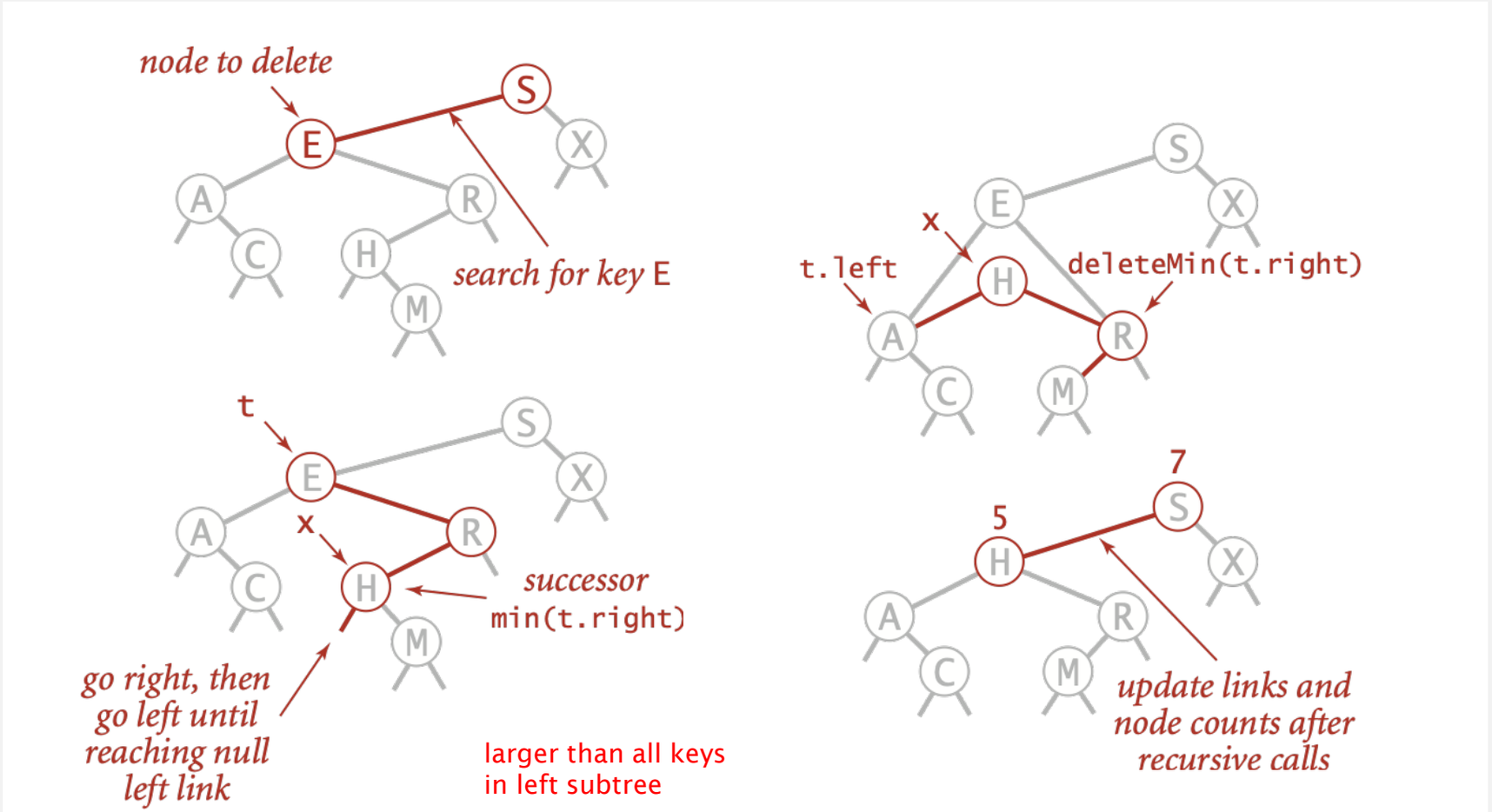


Hibbard deletion

To delete a node with key k: search for node t containing key k.

Case 2. [2 children]

- Find successor x of t. ← x has no left child
- Delete the minimum in t's right subtree. ← but don't garbage collect x
- Put x in t's spot. ← still a BST



Hibbard deletion: Java implementation


```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key) {
   if (x == null) return null; //stop recursion
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {  //found the key to delete
      if (x.right == null) return x.left;
      if (x.left  == null) return x.right;

      Node t = x; //keep the reference to the key to be deleted (E)
      x = min(t.right); //x is the successor (H)
      x.right = deleteMin(t.right); //remove (H)
      x.left = t.left;
   }
   x.count = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

one child

no right child

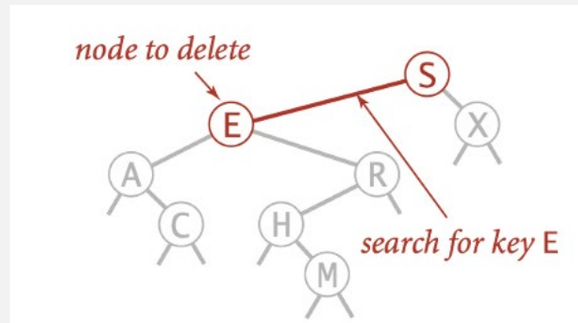
no left child

2 children

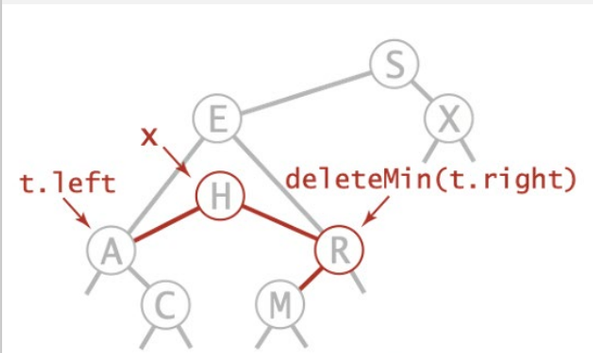
replace with successor

update subtree counts

deleteMin(t.right) ➔ delete H
M becomes left of R



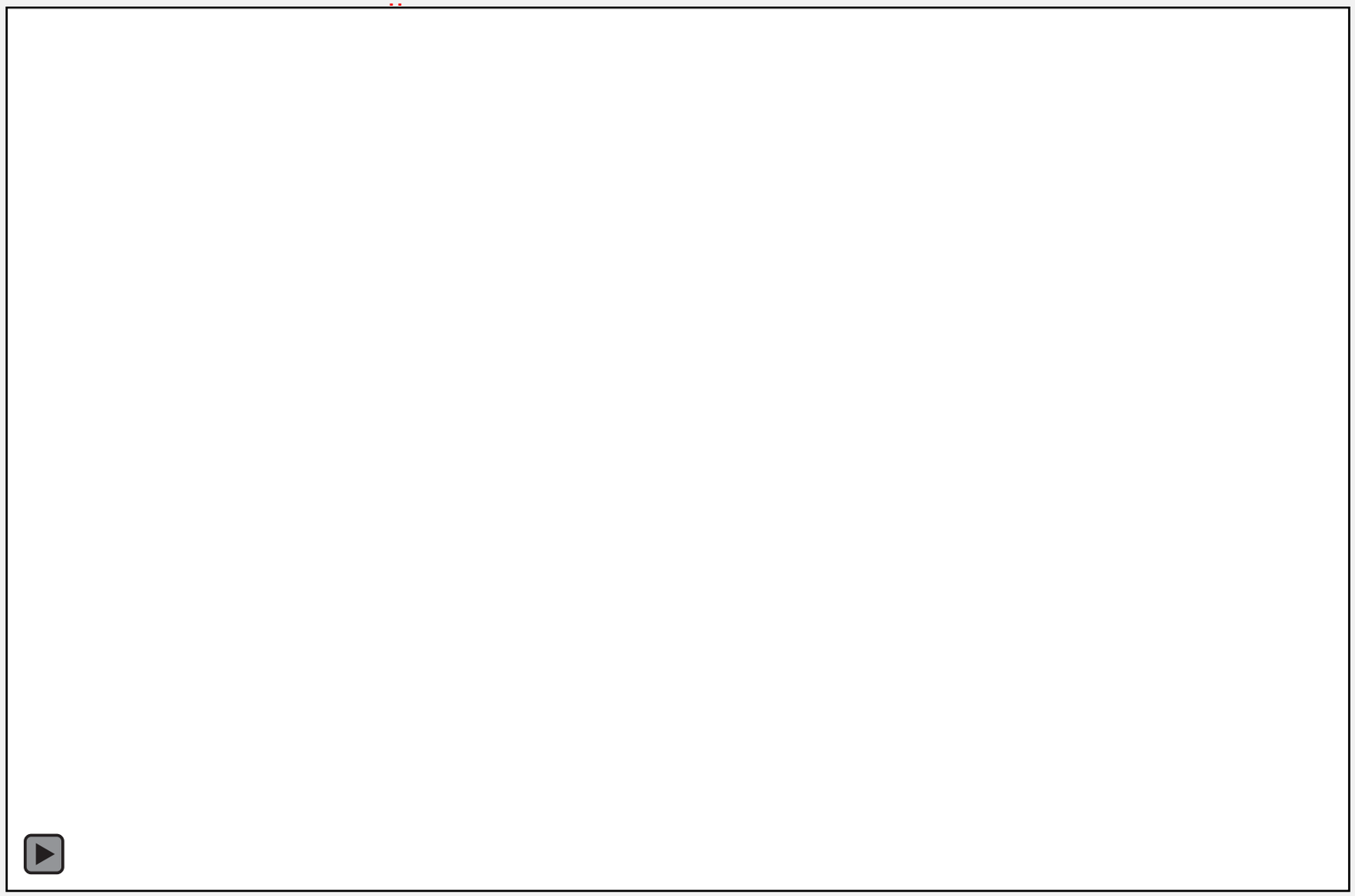
delete E
H to replace E



Hibbard deletion: analysis

Unsatisfactory solution. not symmetric.

- If a delete() always promote the successor, the BST increases its average path length to approximately half of the square root of its size.
- If a delete() makes a random choice to promote the predecessor (max in left subtree) or successor node (min in right subtree, the average path length of the tree remains



Surprising consequence. Trees not random (!) $\Rightarrow$ $\sqrt{n}$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search(unordered list)	n	n	n	n	n	n		equals()
binary search(ordered array)	$\log n$	n	n	$\log n$	n	n	✓	compareTo()
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$	✓	compareTo()

other operations also become √n if deletions allowed

Next lecture. Guarantee logarithmic performance for all operations.

Applications of BST

A well formed Balanced Binary Tree (stay tuned) provides a search in the order of log n

Characteristics of Applications suitable/unsuitable for BST's

1. Applications where data is inserted, updated and deleted dynamically as well as finding max, min, rank etc. For example, maintaining a set of daily bank transactions. (suitable if the tree can be maintained in balanced order)
2. Applications where a large numbers of data records exists are suitable for BST (why?)
3. Applications where small number of records exists, and searching is done rarely are suitable more as a simple array application.

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

BINARY SEARCH TREES

- *Symbol Table*
- *BSTs*
- *Ordered operations*
- *Iteration*
- *Deletion*



http://ds.cs.Rutgers.edu

Slides adopted and modified from Sedgewick/Wayne