CS 171: Introduction to Computer Science II

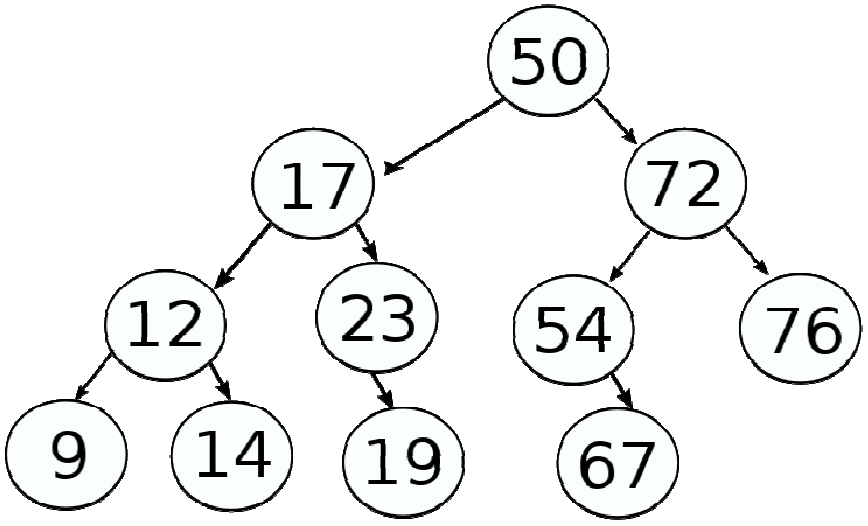
Binary Search Trees

Binary Search Trees

- Generalized from Linked List.
- Advantages:
  - Fast to search
  - Fast to insert and delete
- Recall the search and insertion costs of
  1. Ordered Array
  2. Linked List
- Binary tree combines the advantages of both.

Trees

- What is a tree?
  - **Nodes:** store data and links
  - **Edges:** links, typically directional
- The tree has a top node: **root node**
- The structure looks like reversed from real trees.

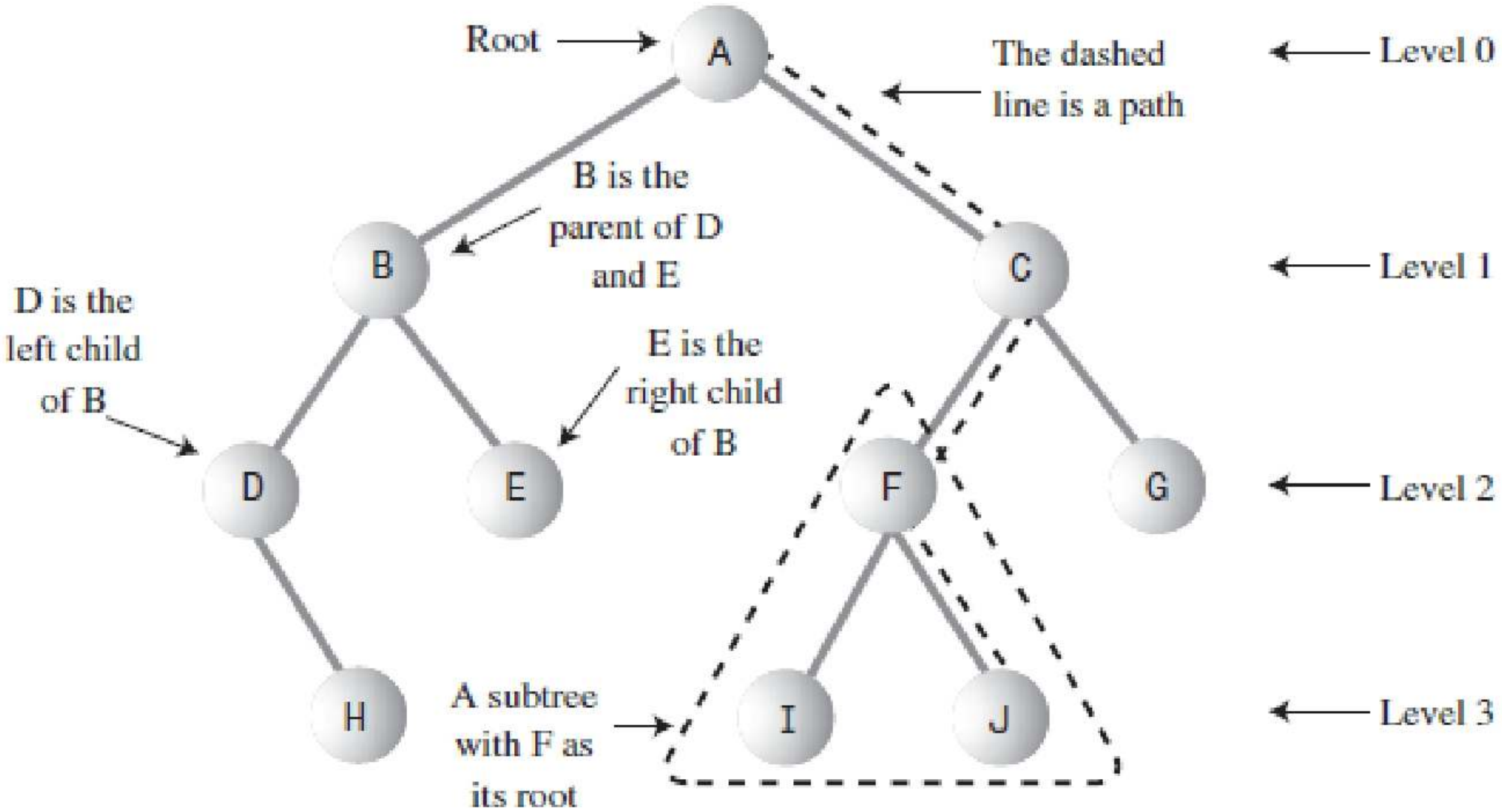


Binary Trees

- Strictly speaking: trees are connected acyclic graphs (i.e. no loops).
- Some other examples of abstract tree structure:
  - Think about the way computer files are organized.

- There are many different kinds of trees.
- Here we will focus on **binary trees**
  - Each node has at most 2 children
  - In other words, at most 2 branches at each node

Terminology



Terminology

- **Root**
  - The node at the top of the tree.
  - There is only one root.
- **Path**
  - The sequence of nodes traversed by traveling from the root to a particular node.
  - Each path is **unique**, why?

Terminology

- **Parent**
  - The node that points to the current node.
  - Any node, except the root, has 1 and only 1 parent.
- **Child**
  - Nodes that are pointed to by the current node.
- **Leaf**
  - A node that has no children is called a leaf.
  - There can be many leaves in a tree.

Terminology

- **Interior node**
  - An interior node has at least one child.
- **Subtree**
  - Any node can be considered the root of a subtree.
  - It consists of all descendants of the current node.
- **Visit**
  - Checking the node value, display node value etc.
- **Traverse**
  - Visit all nodes in some specific order.
  - For example: visit all nodes in ascending key value.

Terminology

- **Levels**
  - The path length from root to the current node.
  - Recall that each path is unique.
  - Root is at level 0.
- **Height**
  - The maximum level in a tree.
  - O(logN) for a reasonably balanced tree.
- **Keys**
  - Each node stores a key value and associated data.

Terminology

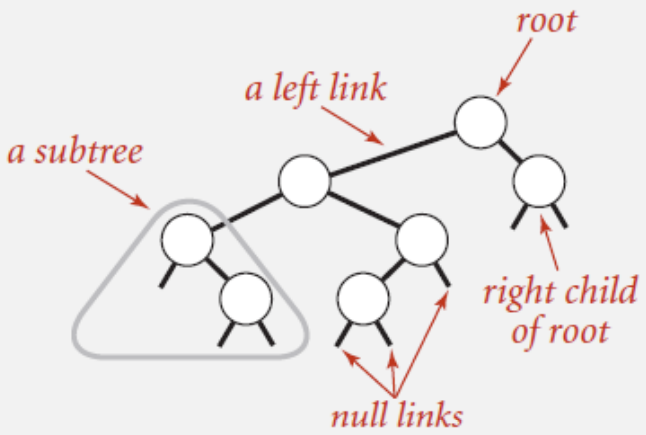
- **Left child / Right child**
  - These are specific to binary trees.
  - Some nodes may have only 1 child.
  - Leaf nodes have no child.
- **Binary Search Tree (BST)**
  - For any node A, its **entire left subtree** must have values less than A, and the **entire right subtree** must have values larger than or equal to A.

Binary search trees

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

A binary tree is either:

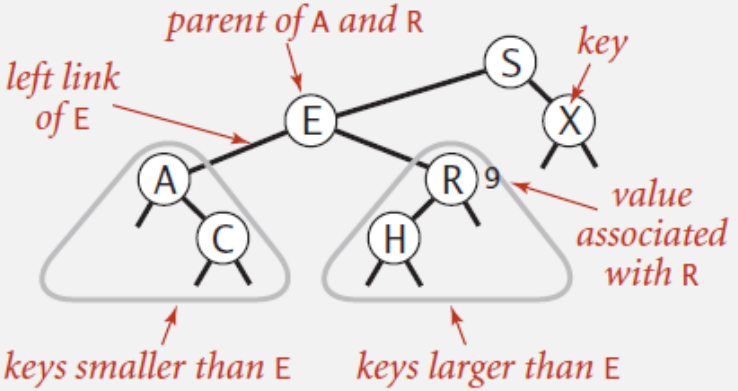
- Empty.
- Two disjoint binary trees (left and right).



Anatomy of a binary tree

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.



Anatomy of a binary search tree

BST representation in Java

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

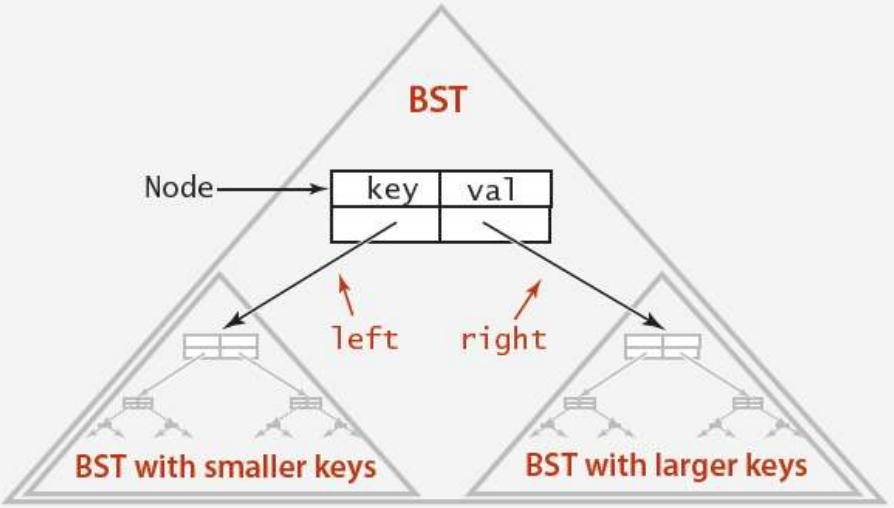
A `Node` is comprised of four fields:

- 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, 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)
   {  /* see next slides */  }

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

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

   public Iterable<Key> iterator()
   {  /* see next slides */  }

}
```

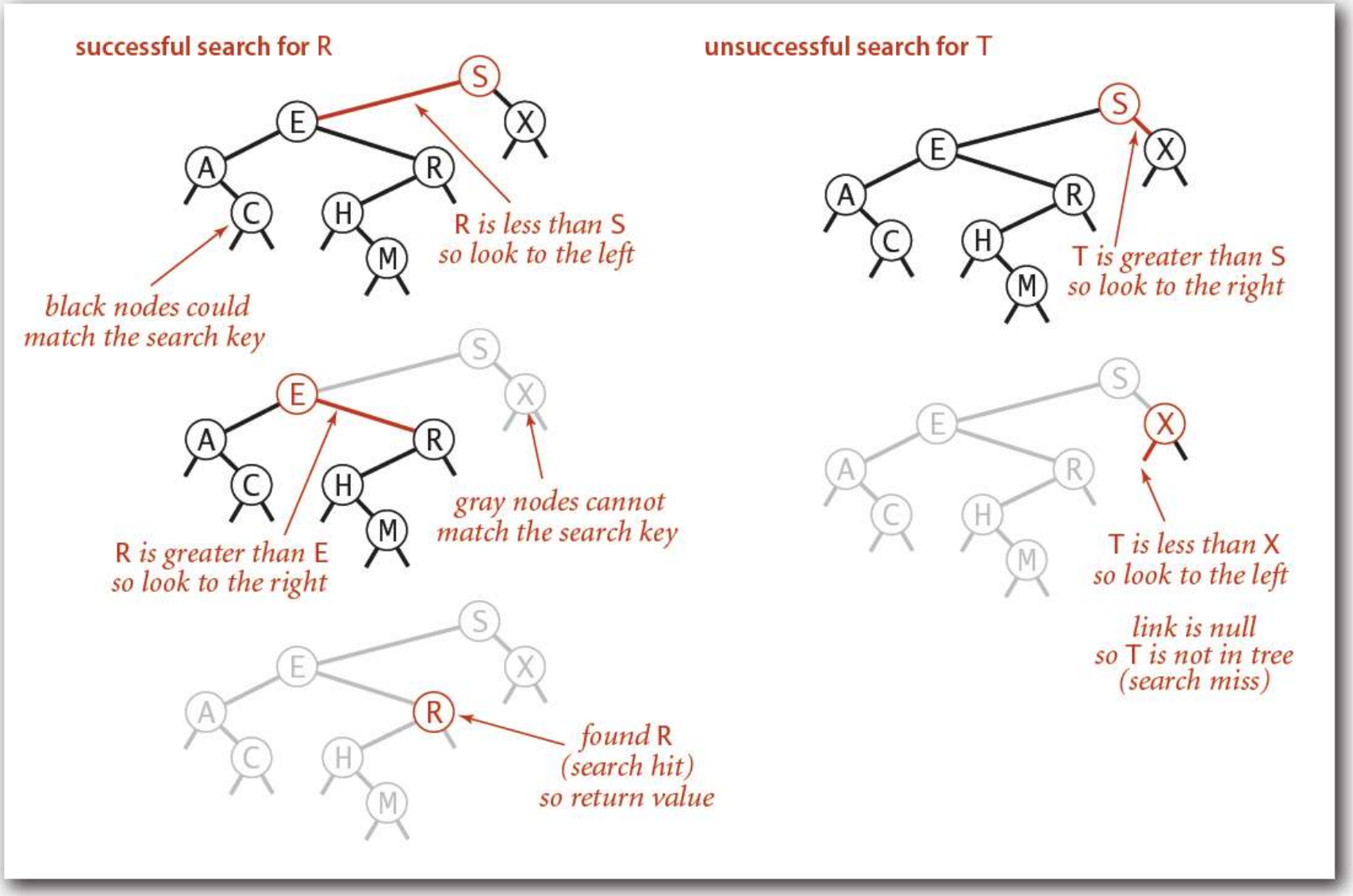
root of BST

BST Demo

- http://algs4.cs.princeton.edu/lectures/32DemoBinarySearchTree.mov

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;
}
```

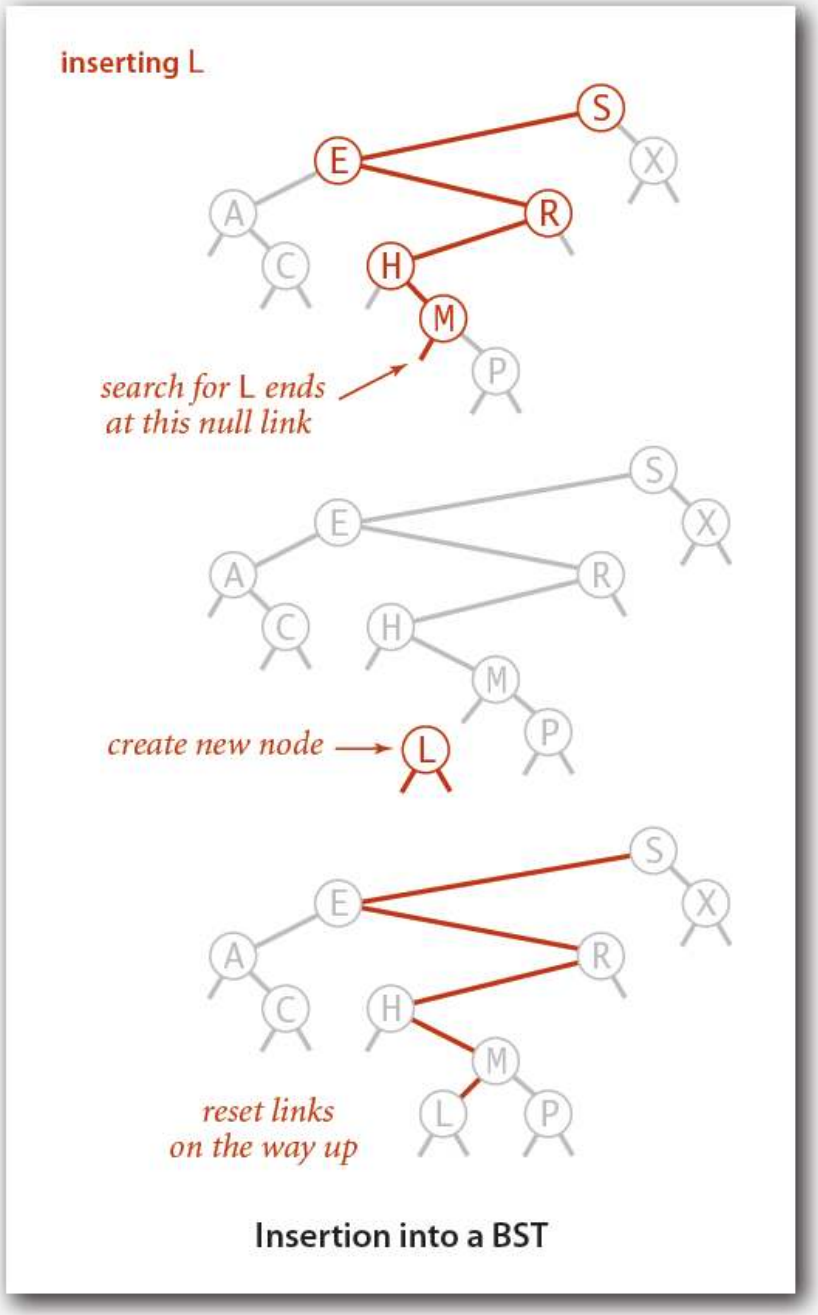
Cost. Number of compares is equal to 1 + depth of node.

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.



Insertion into a BST

BST insert: Java implementation

Put. Associate value with key.

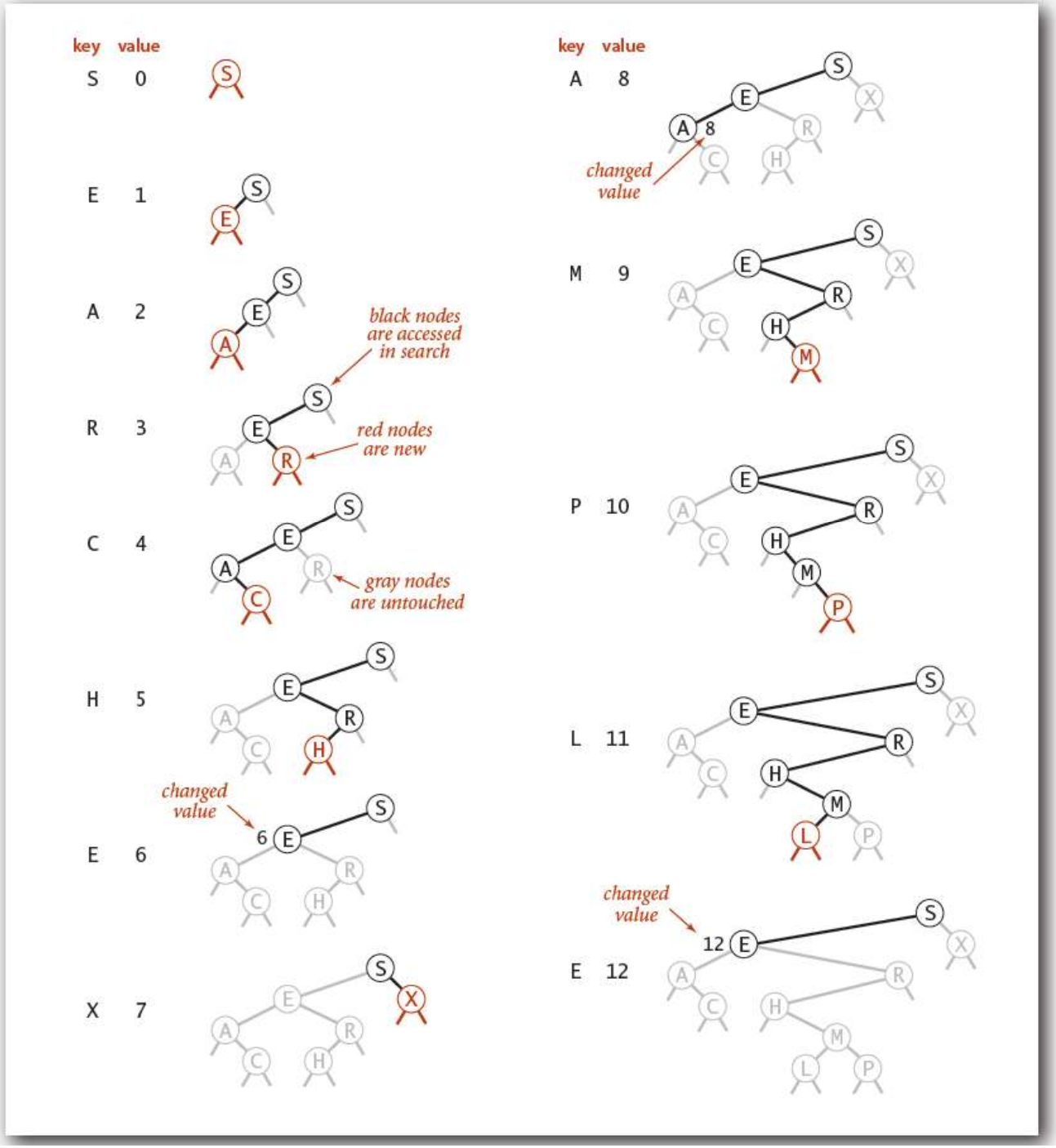
concise, but tricky, recursive code; read carefully!

```
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;
}
```

Cost. Number of compares is equal to 1 + depth of node.

BST trace: standard indexing client



Exercise

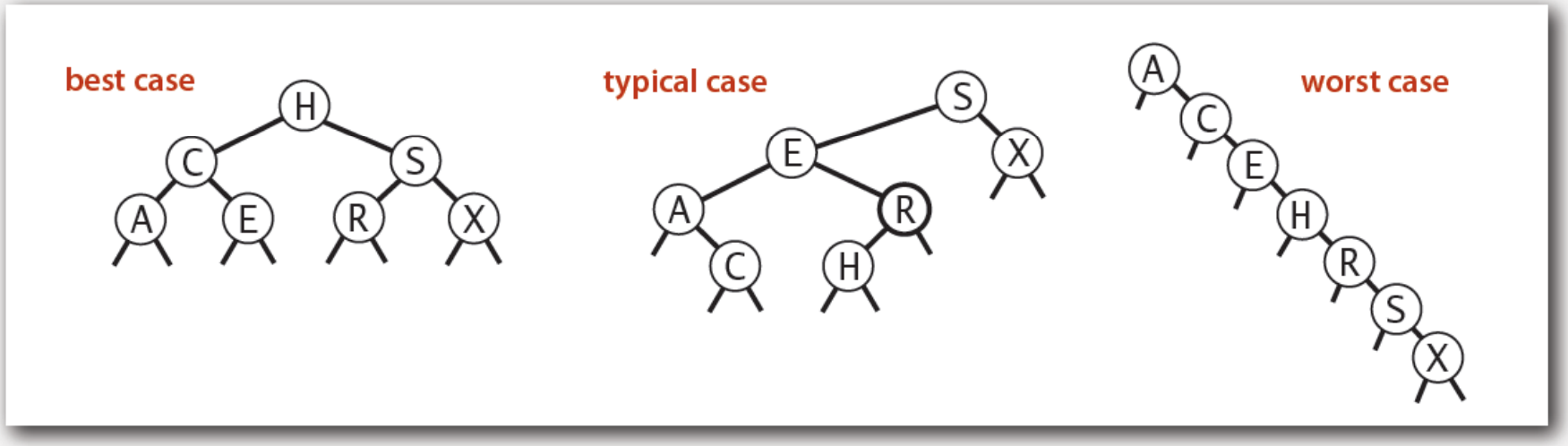
- Insert the following keys (in the order) into an empty BST tree
- Case 1

  H, A, E, R, C, X, S
- Case 2

  A, C, E, H, R, S, X

Tree shape

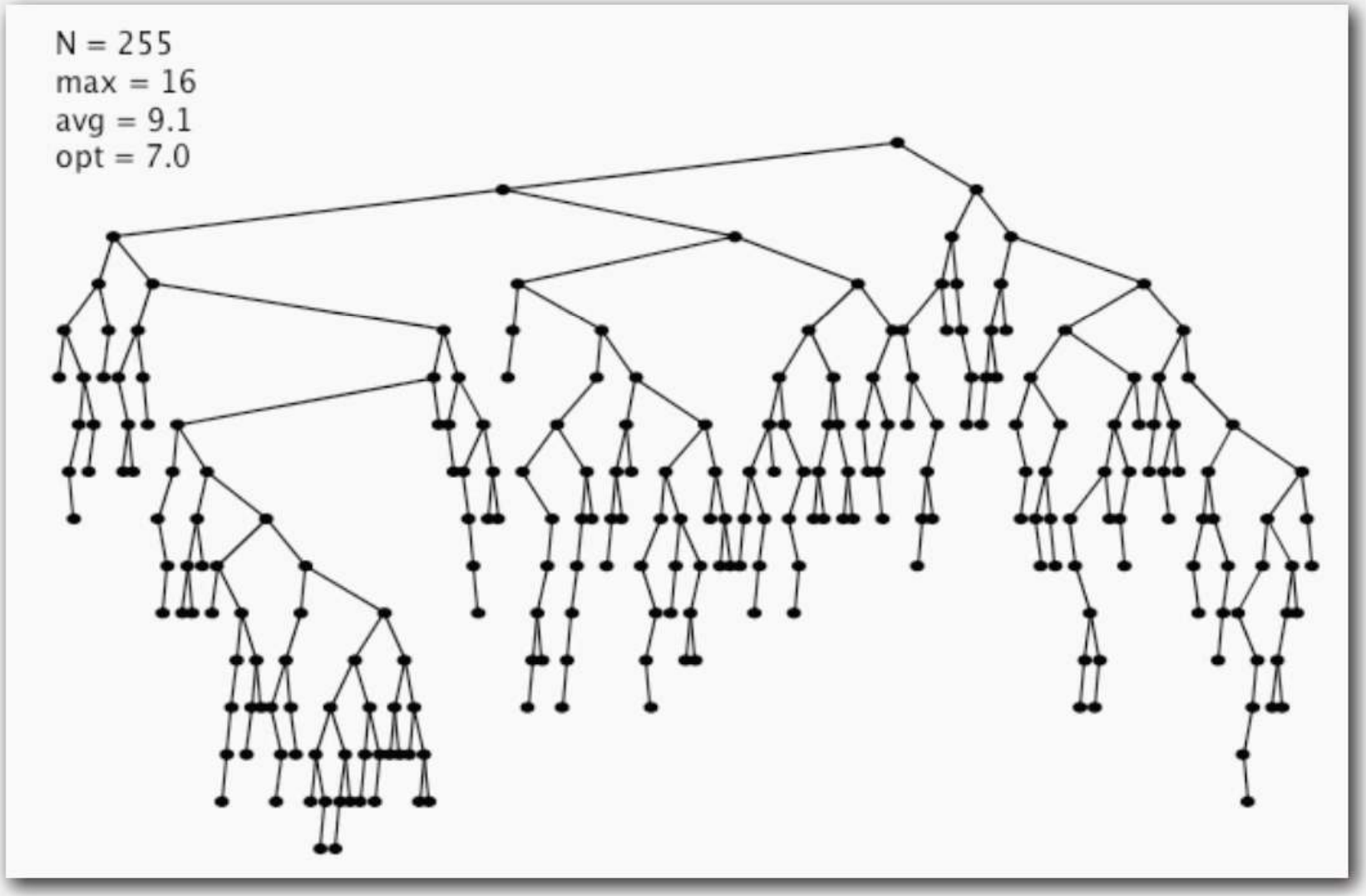
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to 1 + depth of node.



Remark. Tree shape depends on order of insertion.

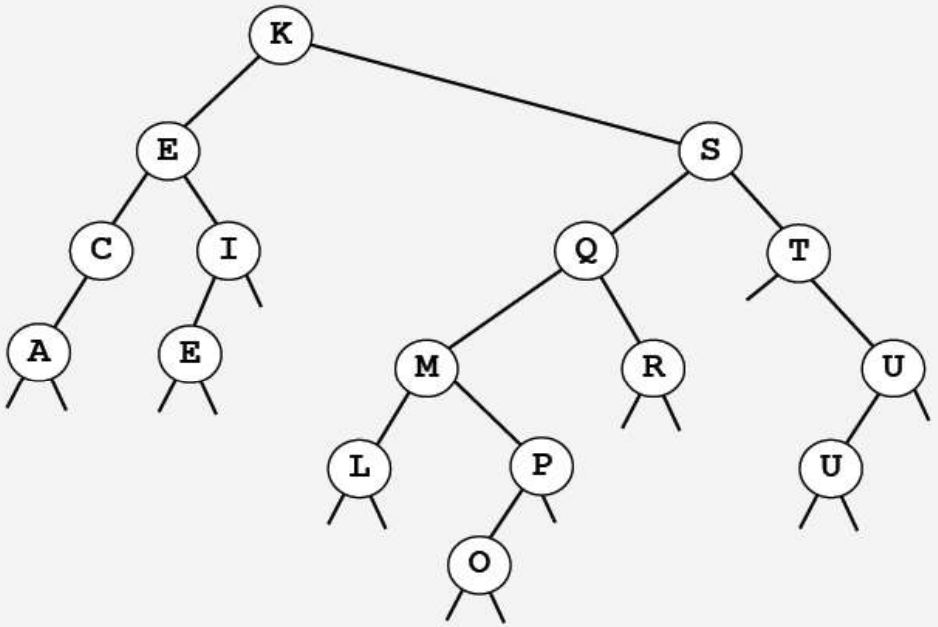
BST insertion: random order

Observation. If keys inserted in random order, tree stays relatively flat.



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	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



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

BSTs: mathematical analysis

Proposition. If keys are inserted 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 keys are inserted in random order, expected height of tree is $\sim 4.311 \ln N$.

But... Worst-case height is N.
(exponentially small chance when keys are inserted in random order)

ST implementations: summary

implementation	guarantee		average case		ordered ops?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N/2	N	no	`equals()`
binary search (ordered array)	lg N	N	lg N	N/2	yes	`compareTo()`
BST	N	N	1.39 lg N	1.39 lg N	?	`compareTo()`

Traversing the Tree

- Traversing – visiting all nodes in a specific order.
  - This is obvious for Array and Linked List.
  - For a tree, there are three different ways.
- **In-order** traversal
- **Pre-order** traversal
- **Post-order** traversal

- All of these use recursion.

In-Order Traversal

- At any node, follows this recursion:
  1. Traverse the **left** subtree.
  2. Visit (e.g. print out) the **current** node.
  3. Traverse the **right** subtree.

- Step 2 is why it's called in-order traversal.

In-Order Traversal

- For a BST, in-order traversal will visit all nodes in **ascending order**.

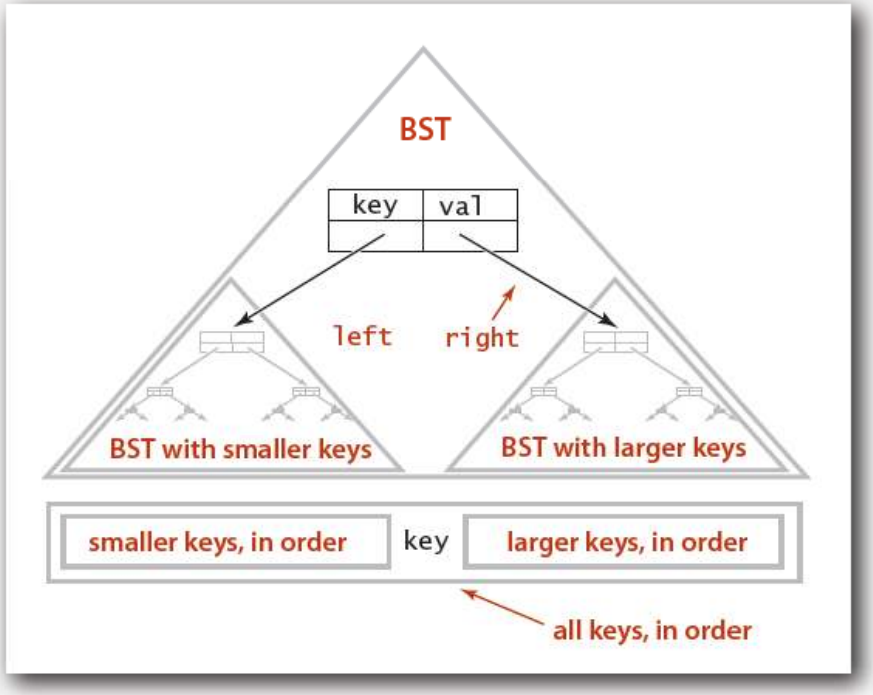
- For other types of trees, in-order traversal still works, but it won't guarantee ascending order.

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);
}
```

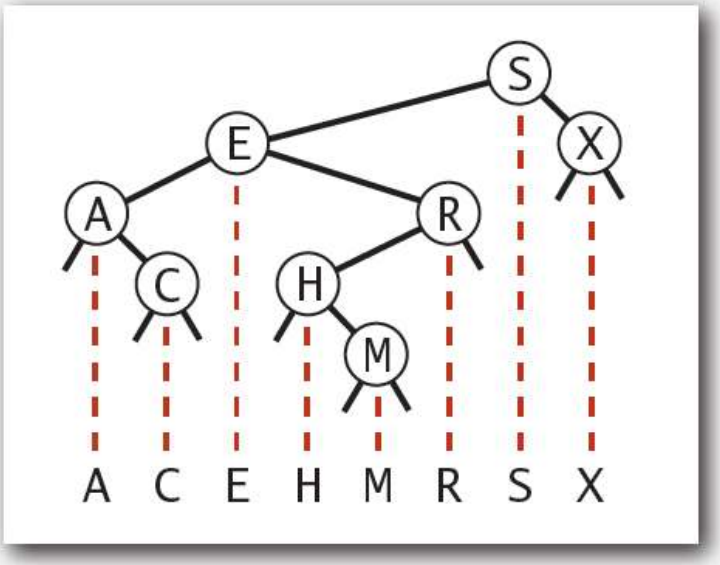


Property. Inorder traversal of a BST yields keys in ascending order.

Inorder traversal

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

recursive calls	queue	function call stack
`inorder(S)`		S
`  inorder(E)`		S E
`    inorder(A)`		S E A
`      enqueue A`	A	
`      inorder(C)`		S E A C
`        enqueue C`	C	
`    enqueue E`	E	
`    inorder(R)`		S E R
`      inorder(H)`		S E R H
`        enqueue H`	H	
`        inorder(M)`		S E R H M
`          enqueue M`	M	
`      enqueue R`	R	
`  enqueue S`	S	
`  inorder(X)`		S X
`    enqueue X`	X	



Pre-Order Traversal

- At any node, follows this recursion:
  1. Visit (e.g. print out) the **current** node.
  2. Traverse the **left** subtree.
  3. Traverse the **right** subtree.
- Step 1 is why it's called pre-order traversal.

- What's the result of this for BST?

Post-Order Traversal

- At any node, follows this recursion:
  1. Traverse the **left** subtree.
  2. Traverse the **right** subtree.
  3. Visit (e.g. print out) the **current** node.
- Step 3 is why it's called post-order traversal.

- What's the result of this for BST?