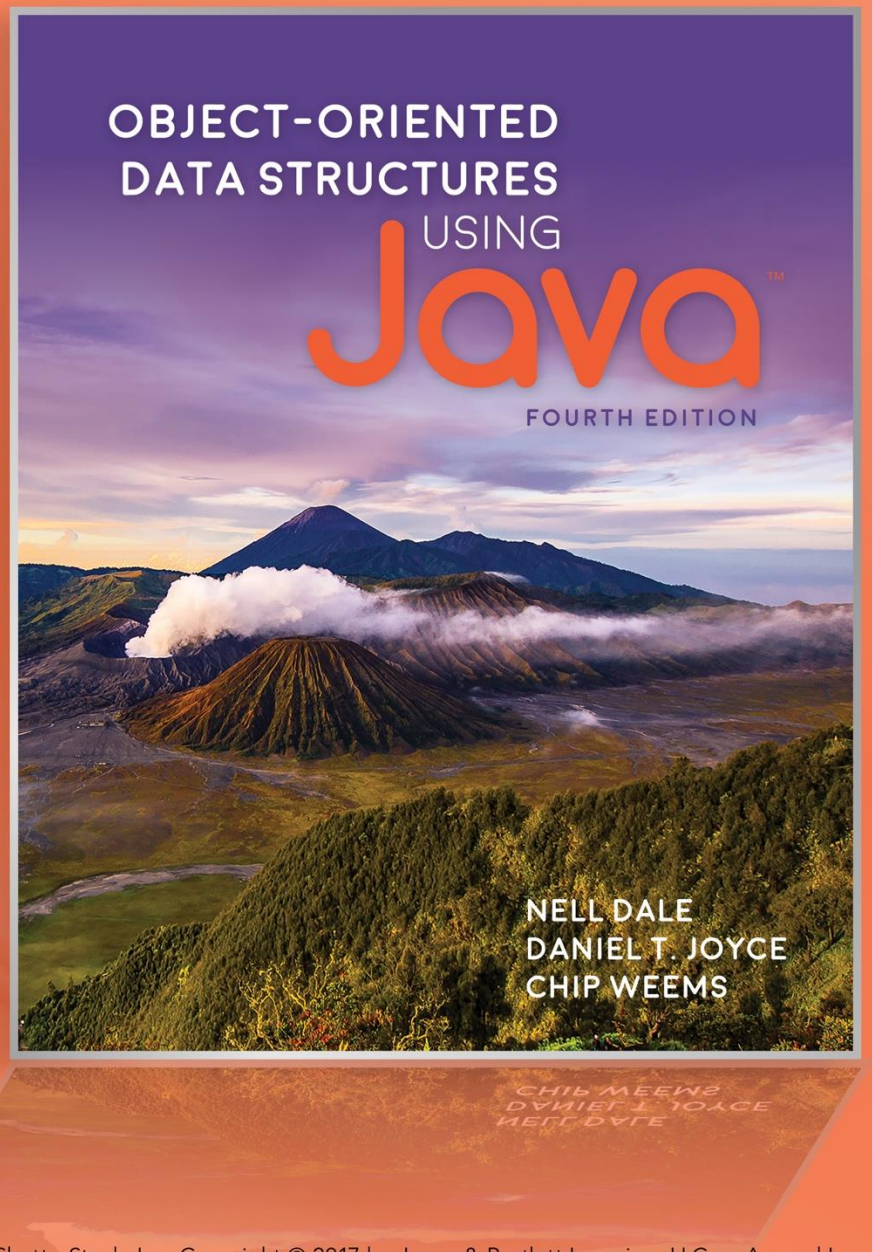
Chapter 7

The Binary Search Tree ADT

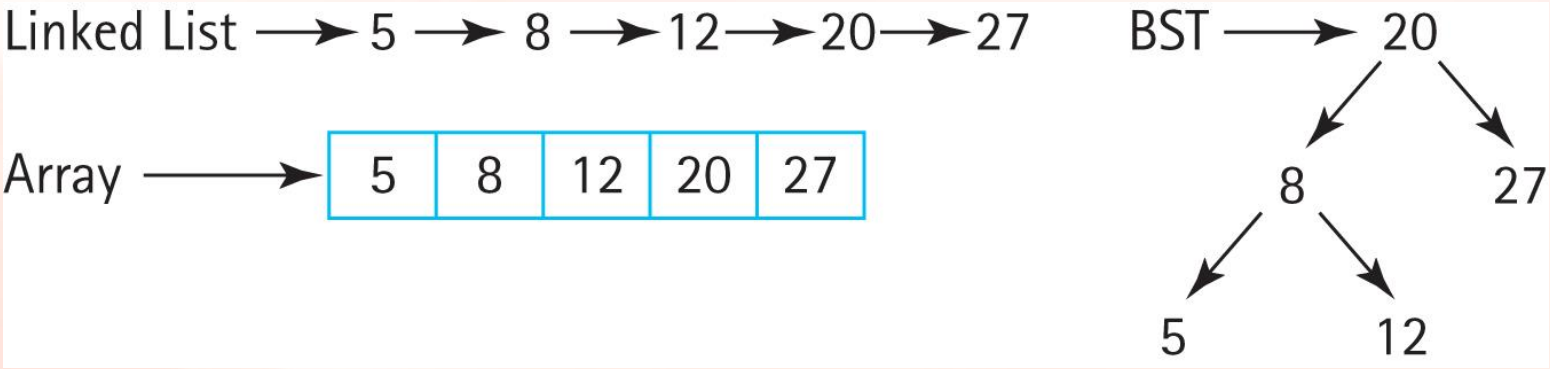


Chapter 7: The Binary Search Tree ADT

7.1 – Trees
7.2 – Binary Search Trees
7.3 – The Binary Search Tree Interface
7.4 – The Implementation Level: Basics
7.5 – Iterative Versus Recursive Method Implementations
7.6 – The Implementation Level: Remaining Observers
7.7 – The Implementation Level: Transformers
7.8 – Binary Search Tree Performance
7.9 – Application: Word Frequency Counter
7.10 – Tree Variations

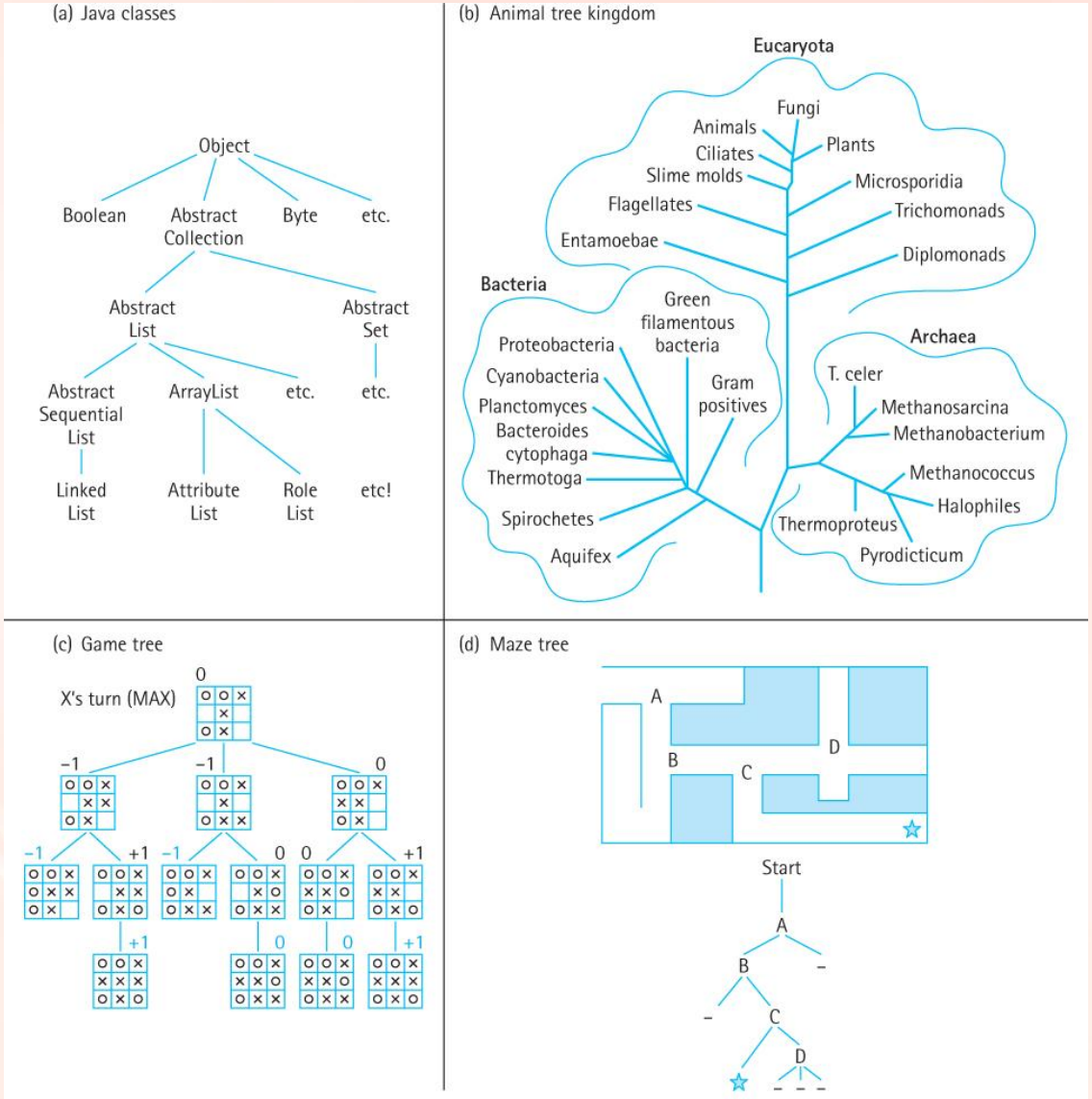
Binary Search Tree

- When maintaining a sorted list
  - Linked List requires O(N) find location, O(1) insertion
  - Array requires $O(\log_2 N)$ find location, O(N) insertion
  - Binary Search Tree, in general allows $O(\log_2 N)$ find location , O(1) insertion



7.1 Trees

- A **tree** is a nonlinear structure in which each node is capable of having many successor nodes, called *children*.
- Trees are useful for representing lots of varied relationships among data items.

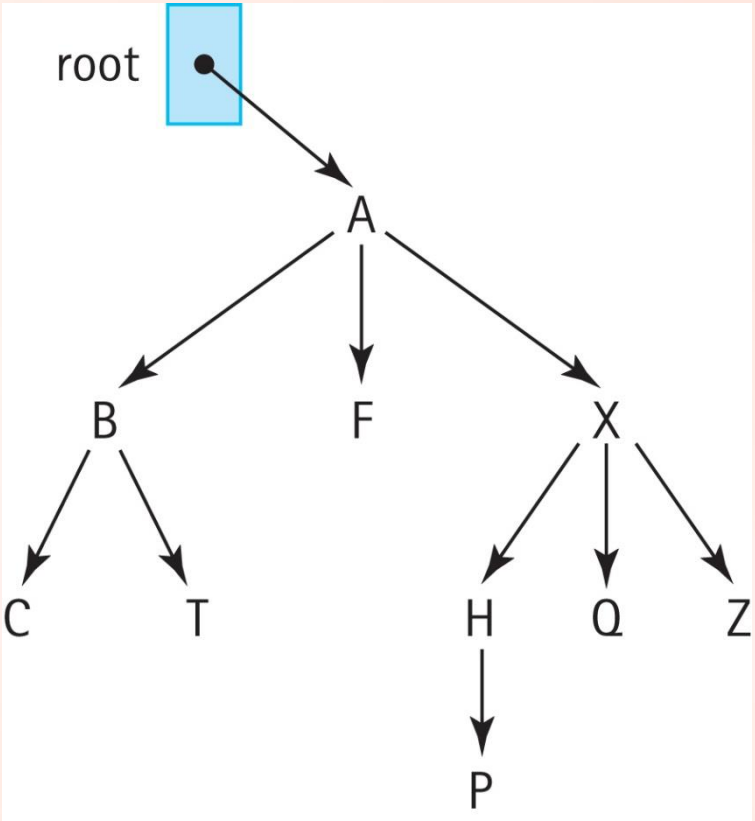


Definitions

- **Tree** A structure with a unique starting node (the root), in which each node is capable of having multiple successor nodes (its children), and in which a unique path exists from the root to every other node.
- **Root** The top node of a tree structure; a node with no parent
- **Parent node** The predecessor node of a node is its parent
- **Subtree** A node and all of its descendants form a subtree rooted at the node

Examples

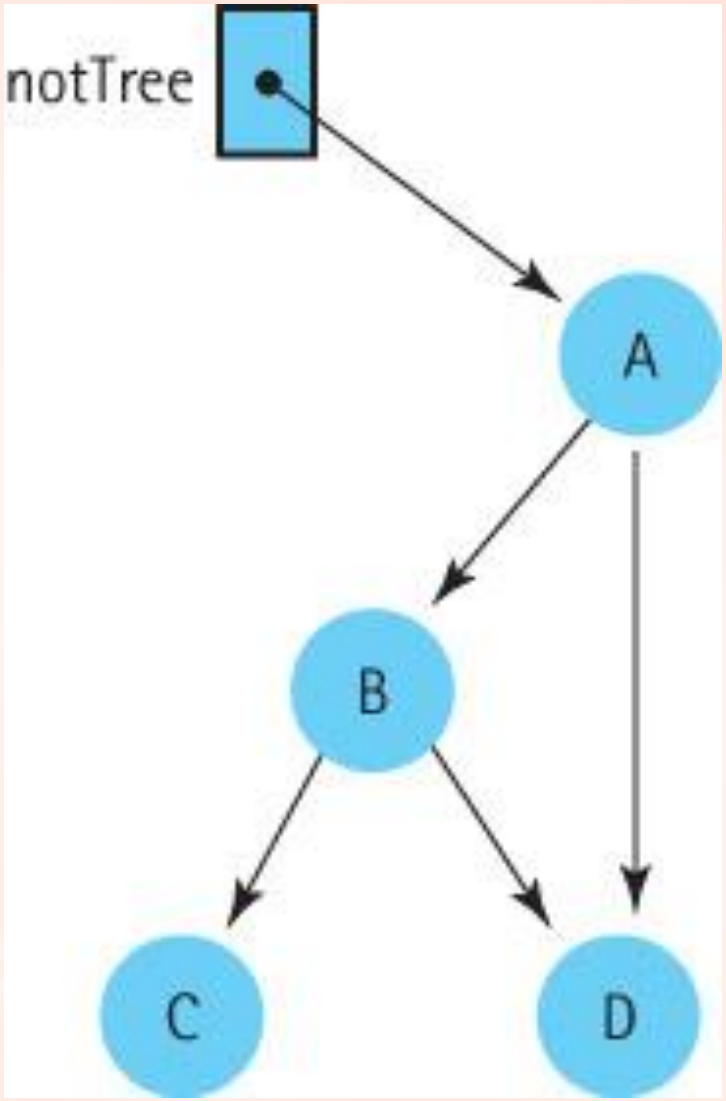
- The root is A
- The children of A are B, F, and X
- The parent of Q is X
- The leftmost subtree of X contains H and P



Required

- A tree's subtrees must be disjoint
- There is a unique path from the root of a tree to any other node of the tree

<u>Not</u> a Tree →

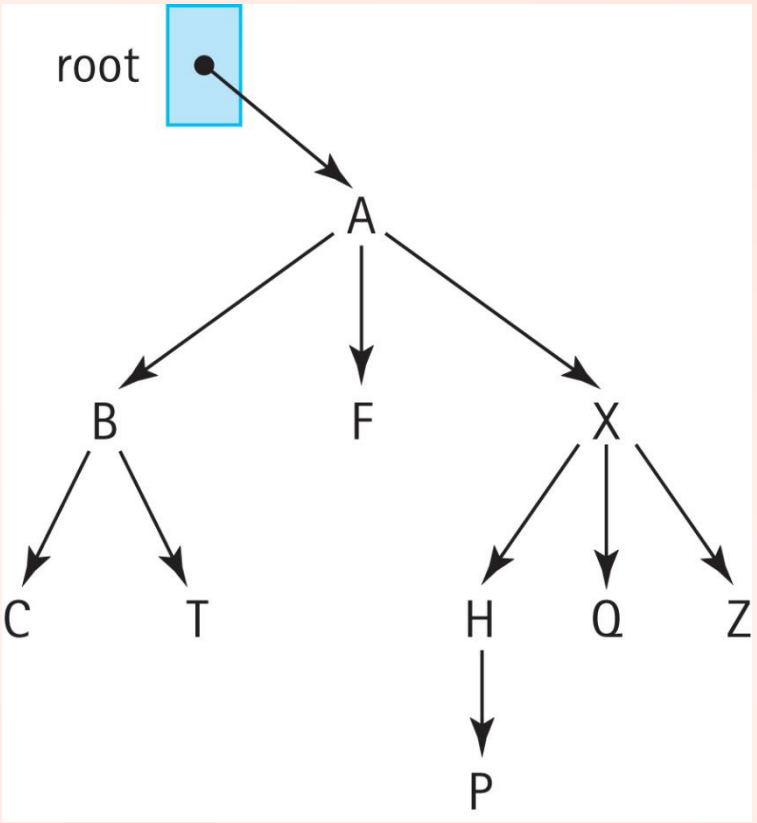


Definitions

- **Ancestor** A parent of a node, or a parent of an ancestor
- **Descendant** A child of a node, or a child of a descendant
- **Leaf** A node that has no children
- **Interior node** A node that is not a leaf
- **Siblings** Nodes with the same parent

Examples

- The ancestors of P are H, X, and A
- The descendants of X are H, Q, Z, and P
- The leaf nodes are C, T, F, P, Q, and Z
- The interior nodes are A, B, X
- The siblings of B are F and X

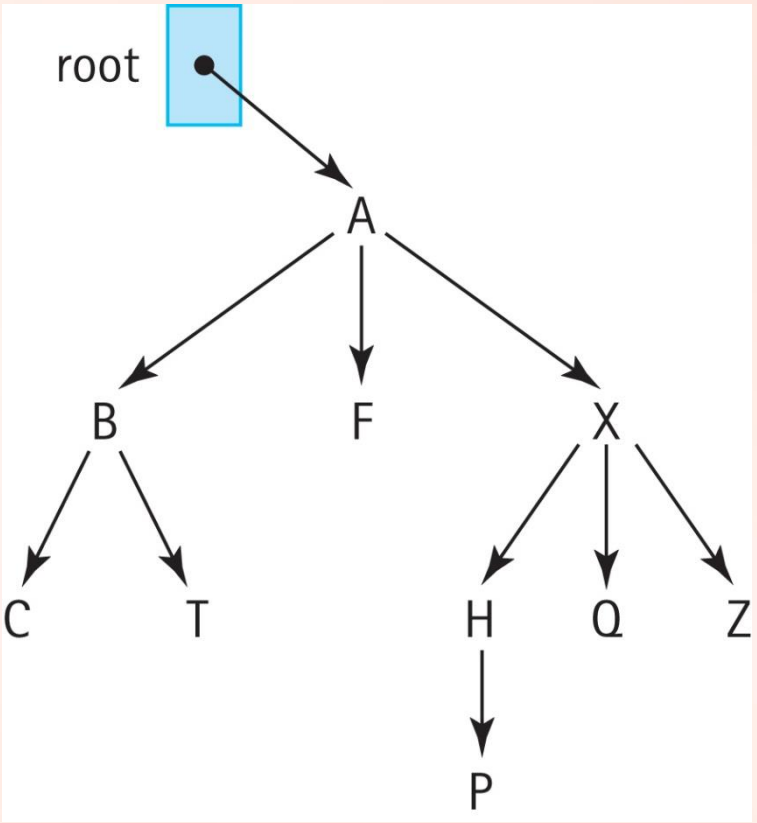


Definitions

- **Level** The level of a node is its distance from the root (the number of connections between itself and the root)
- **Height** The maximum level of the tree

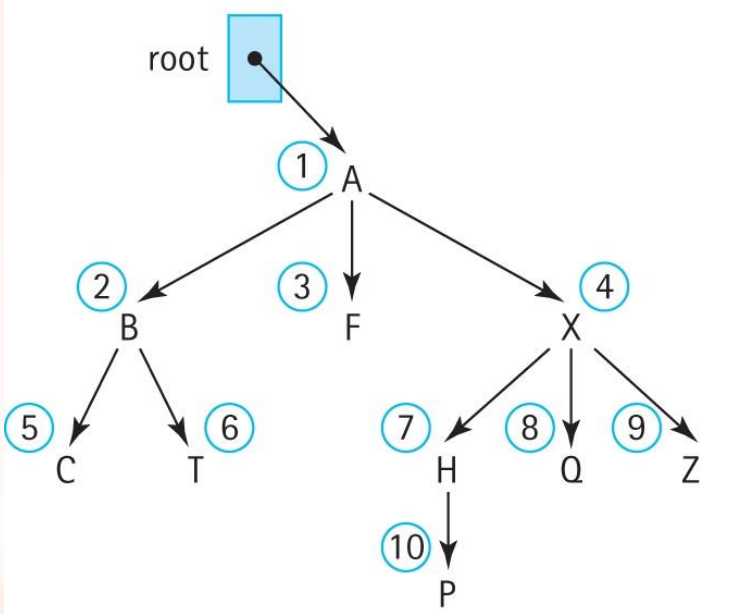
Examples

- The level of A is 0
- The level of P is 3
- The height of the tree is 3



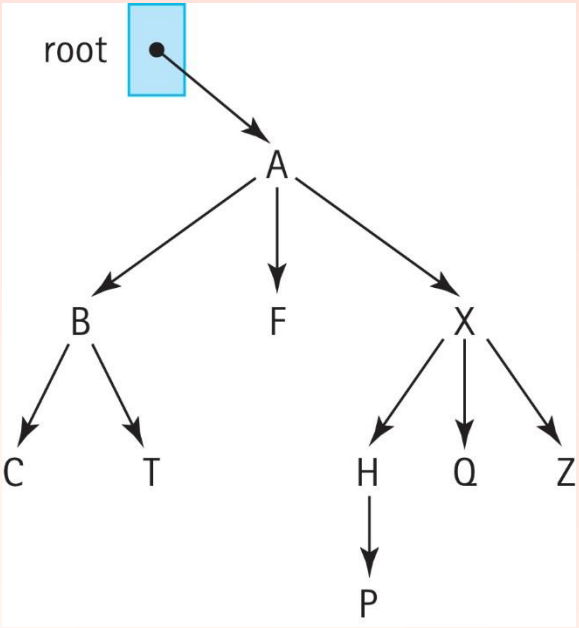
Breadth-First Traversal

- Also called “level-order traversal”
- A B F X C T H Q Z P



Breadth-First Traversal(root)

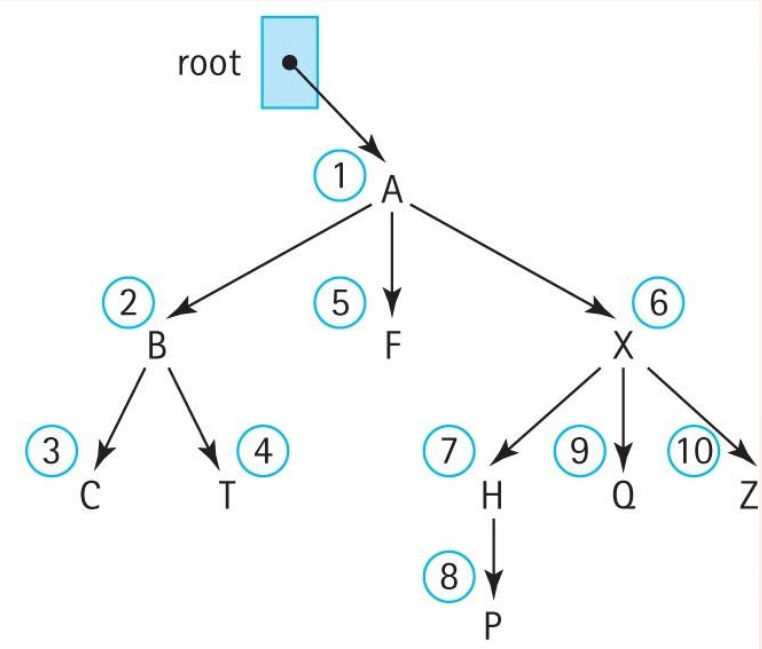
```
Instantiate a queue of nodes
if (root is not null)
{
   queue.enqueue(root)
   while (!queue.isEmpty())
   {
      node = queue.dequeue()
      Visit node
      Enqueue the children of node
         (from left to right) into queue
   }
}
```



Try It!

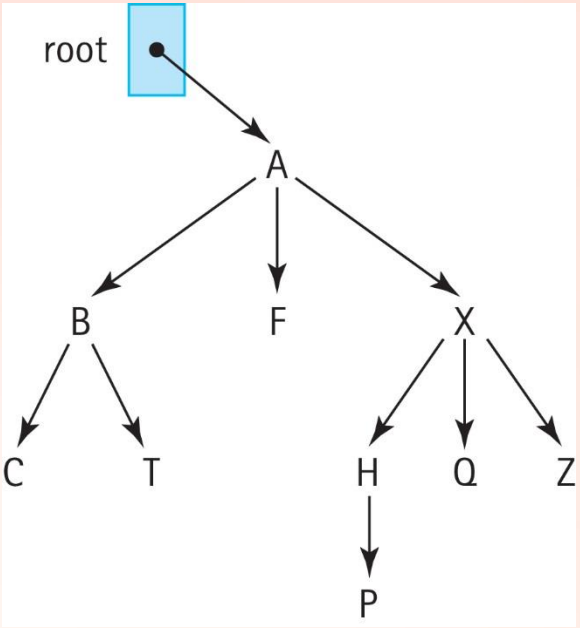
Depth-First Traversal

- A B C T F X H P Q Z



Depth-First Traversal(root)

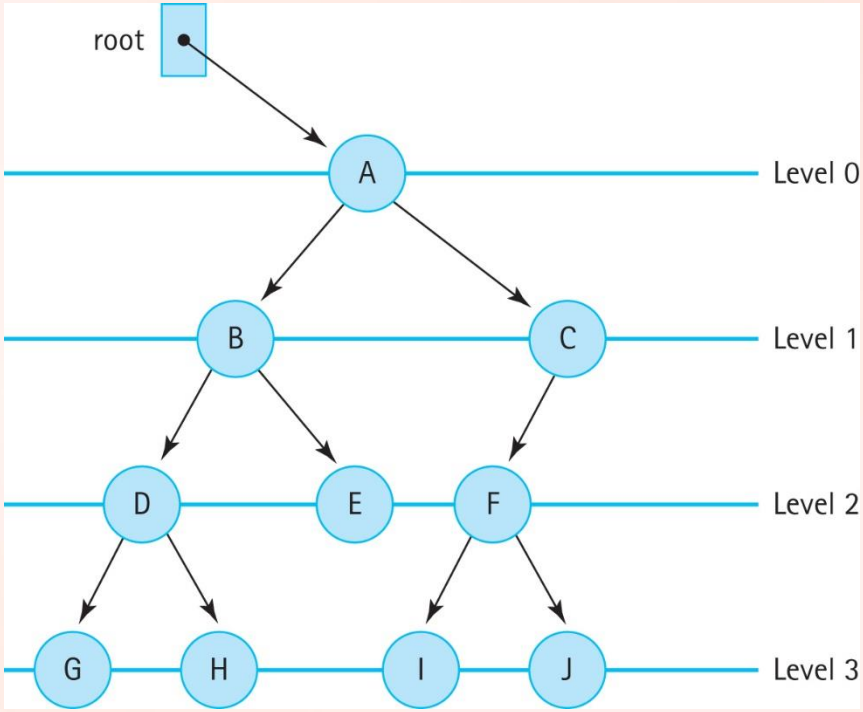
```
Instantiate a stack of nodes
if (root is not null)
{
   stack.push(root)
   while (!stack.isEmpty())
   {
      node = stack.top()
      stack.pop()
      Visit node
      Push the children of node
         (from right to left) onto queue
   }
}
```



Try It!

7.2 Binary Search Trees

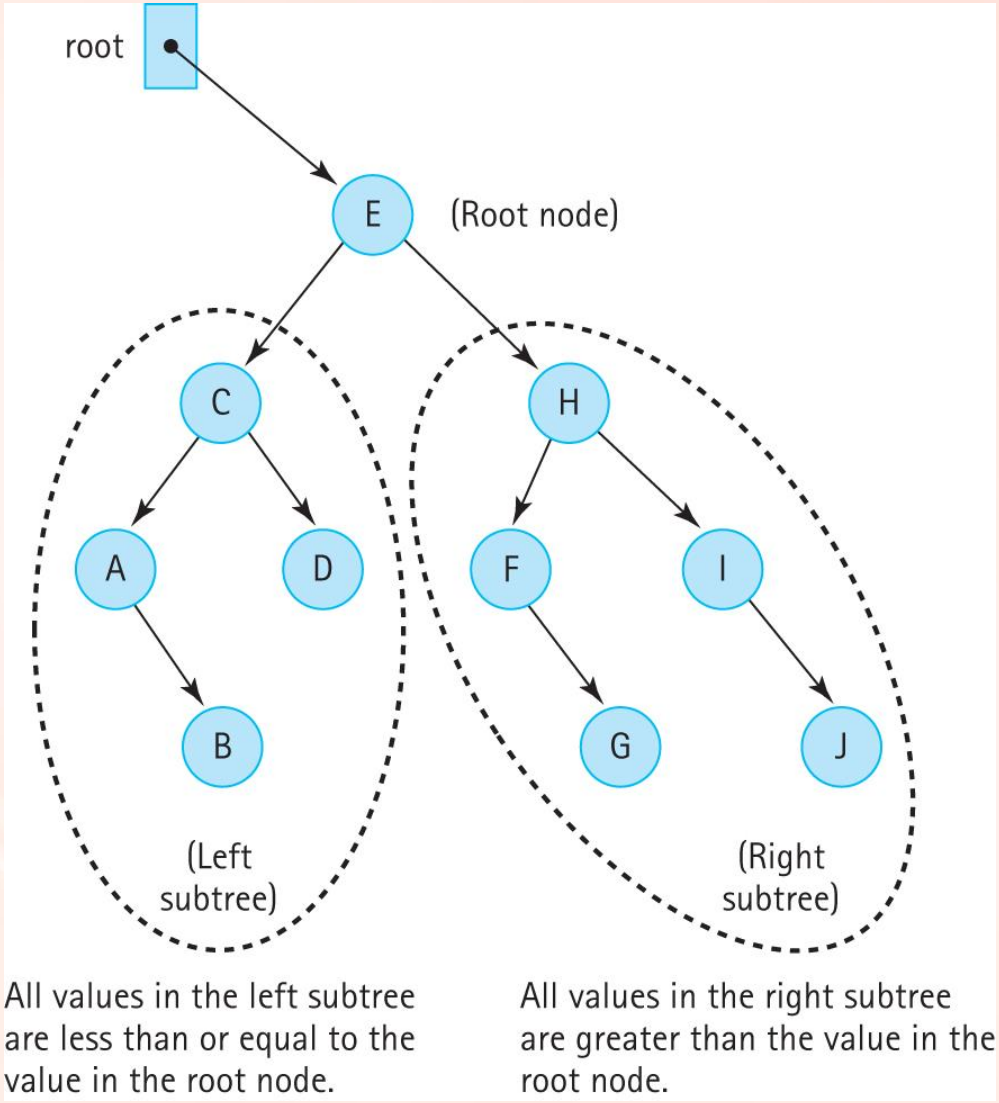
- **Binary tree** A tree in which each node is capable of having two child nodes, a left child node and a right child node



A Binary Tree

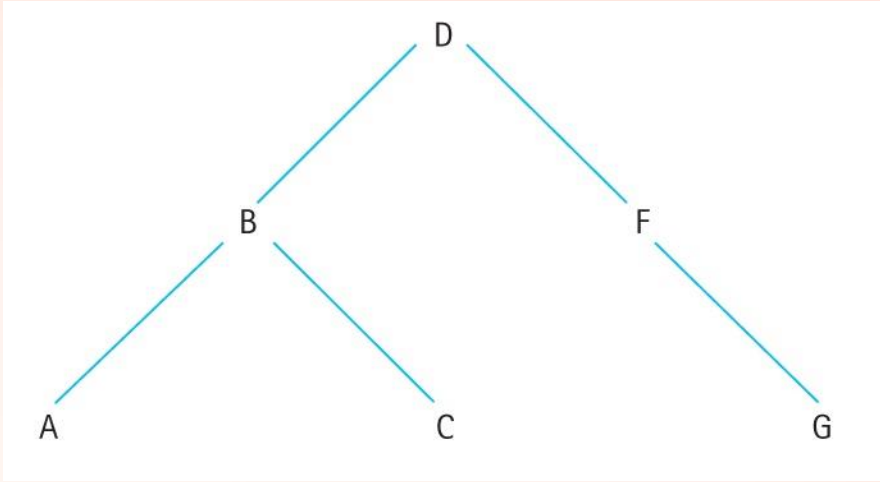
A Binary Search Tree

These trees facilitate searching for an element.



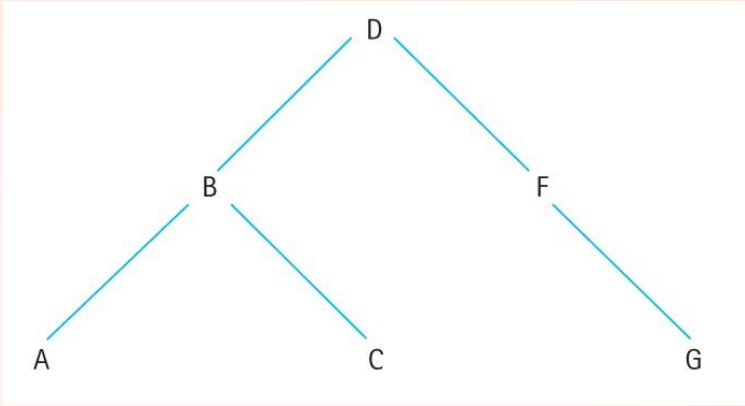
Binary search tree

- A binary tree in which the key value in any node
  - is greater than or equal to the key value in its left child and any of its descendants (the nodes in the left subtree)
  - is less than the key value in its right child and any of its descendants (the nodes in the right subtree)
- We call this the “binary search tree” property

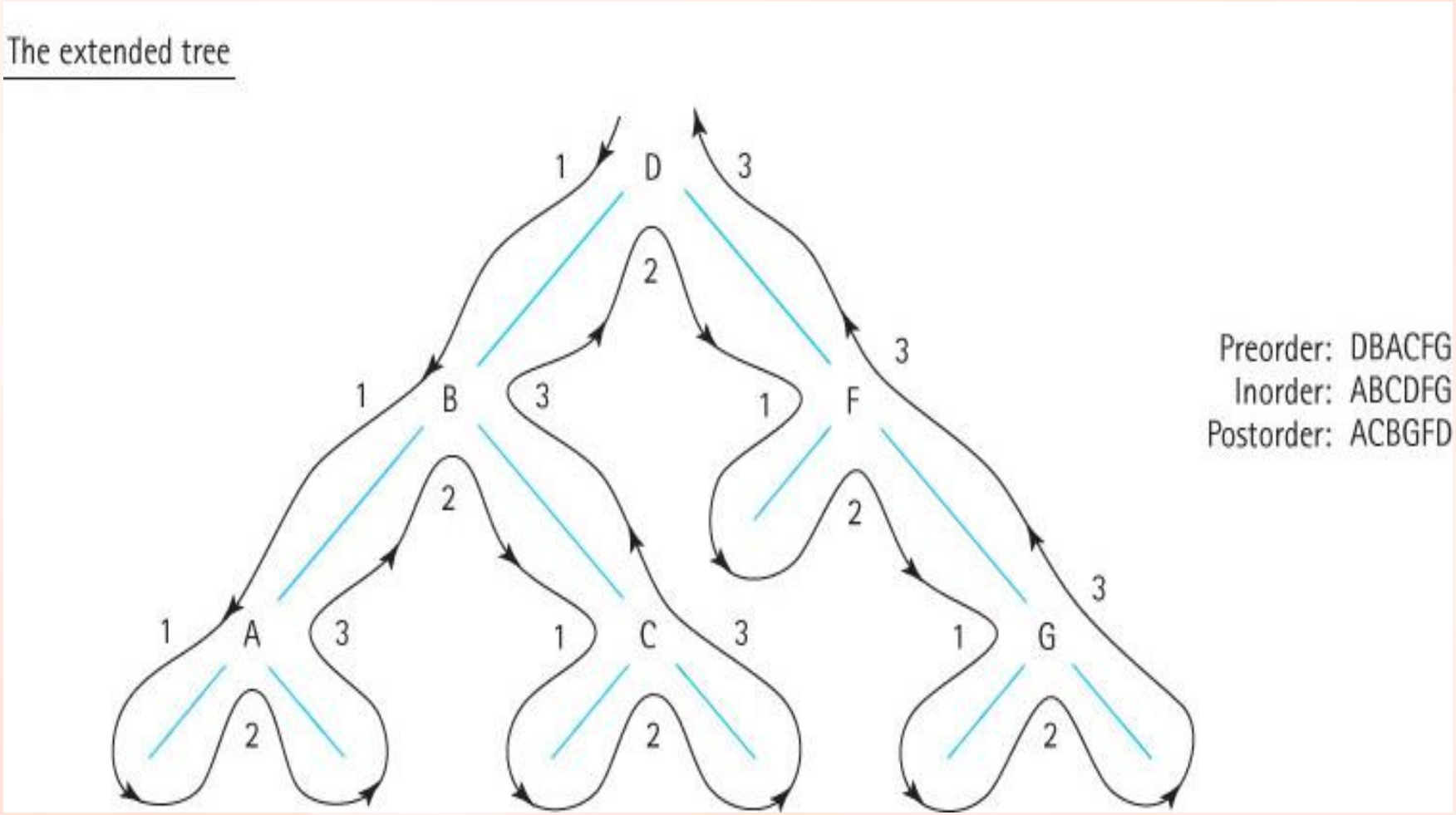


Binary Tree Traversals

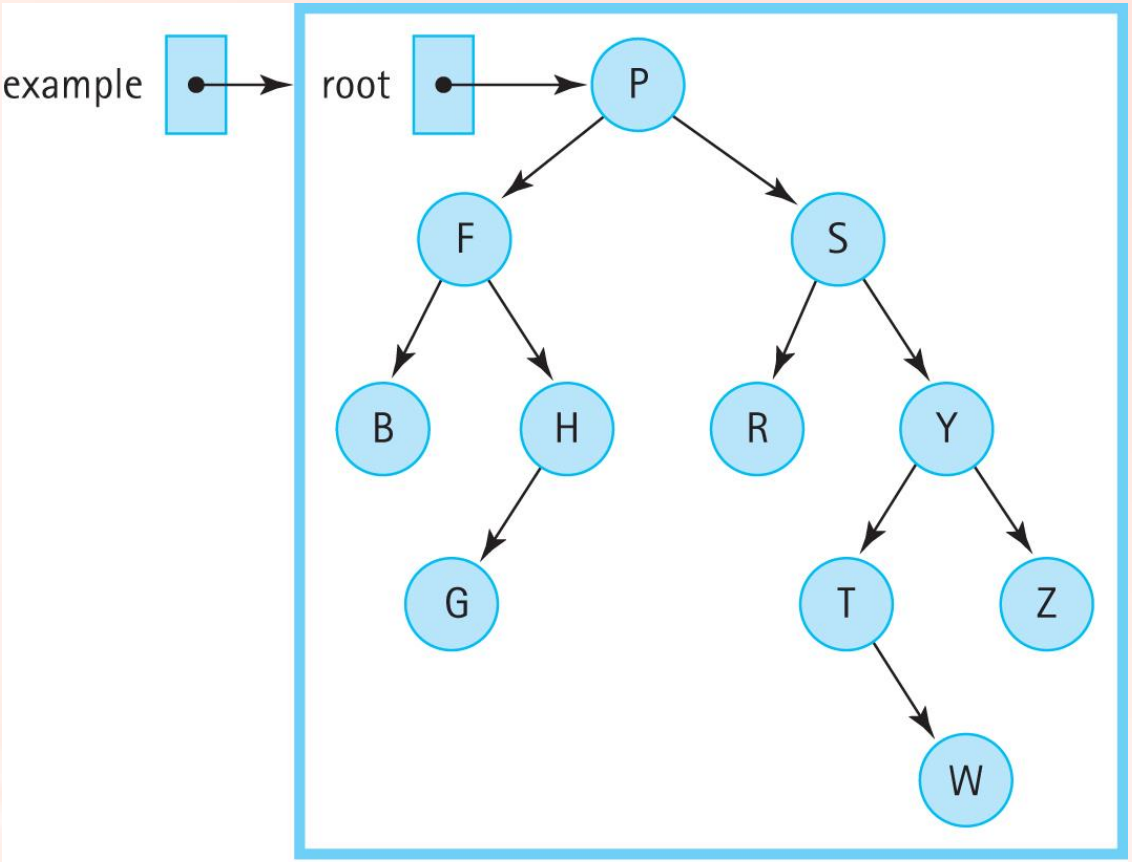
- **Preorder traversal**: Visit the root, visit the left subtree, visit the right subtree:
  - D B A C F G
- **Inorder traversal**: Visit the left subtree, visit the root, visit the right subtree:
  - A B C D F G
- **Postorder traversal**: Visit the left subtree, visit the right subtree, visit the root:
  - A C B G F D



Visualizing Binary Tree Traversals



Three Binary Tree Traversals



Inorder: B F G H P R S T W Y Z
Preorder: P F B H G S R Y T W Z
Postorder: B G H F R W T Z Y S P

7.3 – The Binary Search Tree Interface

- Our binary search trees
  - are similar to the sorted lists of Chapter 6
  - implement this text's `CollectionInterface`
  - implement the Java Library's `Iterable` interface
  - are unbounded, allow duplicate elements, and disallow `null` elements
  - support max and min operations
  - support preorder, inorder, and postorder traversals

```
package ch07.trees;

import ch05.collections.CollectionInterface;
import java.util.Iterator;

public interface BSTInterface<T> extends CollectionInterface<T>,
                                         Iterable<T>
{
  // Used to specify traversal order.
  public enum Traversal {Inorder, Preorder, Postorder};

  T min();
  // If this BST is empty, returns null;
  // otherwise returns the smallest element of the tree.

  T max();
  // If this BST is empty, returns null;
  // otherwise returns the largest element of the tree.

  public Iterator<T> getIterator(Traversal orderType);
  // Creates and returns an Iterator providing a traversal of a "snapshot"
  // of the current tree in the order indicated by the argument.
}
```

Iteration

- In addition to the getIterator method, a class that implements the `BSTInterface` must provide a separate `iterator` method, because `BSTInterface` extends `Iterable`.
- This method should return an `Iterator` that provides iteration in the “natural” order of the tree elements.
- For most applications this would be an inorder traversal, and we make that assumption in our implementation.

Iteration

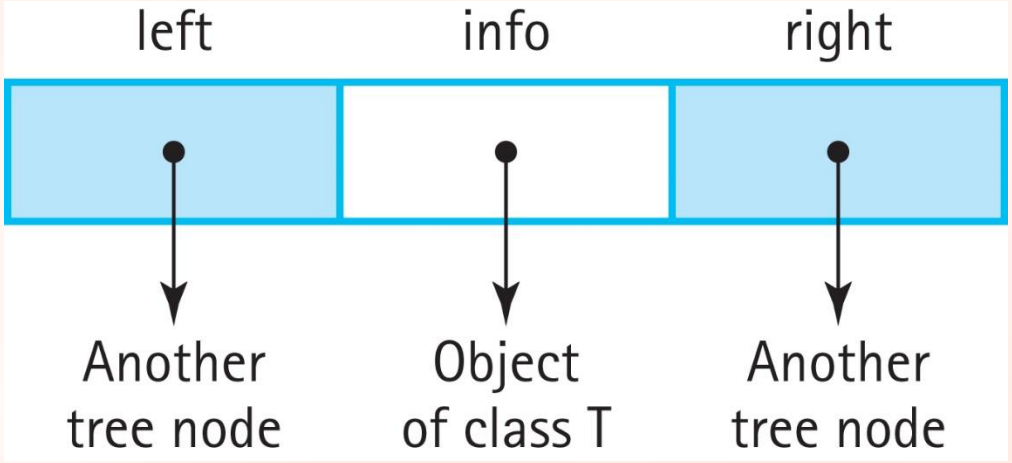
- We intend the iterators created and returned by `getIterator` and `iterator` to provide a **snapshot** of the tree as it exists at the time the iterator is requested.
- They represent the state of the tree at that time and subsequent changes to the tree should not affect the results returned by the iterator's `hasNext` and `next` methods.
- Our iterators will throw an `UnsupportedOperationException` if `remove` is invoked.

Example Application

- Instructors can review and demonstrate the application `BSTExample` found in the `ch07.apps` package
- This application demonstrates the use of iterators with our binary search tree

7.4 The Implementation Level: Basics

- We define `BSTNode.java` in our `support` package to provide nodes for our binary search trees
- Visually, a `BSTNode` object is:



BSTNode.java

instance variables:

```
private T info;                 // The info in a BST node
private BSTNode<T> left;        // A link to the left child node
private BSTNode<T> right;       // A link to the right child node
```

Constructor:

```
public BSTNode(T info)
{
  this.info = info;
  left = null;
  right = null;
}
```

Plus it includes the standard setters and getters

Beginning of BinarySearchTree.java

```
package ch07.trees;

import java.util.*;   // Iterator, Comparator

import ch04.queues.*;
import ch02.stacks.*;
import support.BSTNode;

public class BinarySearchTree<T> implements BSTInterface<T>
{
  protected BSTNode<T> root;      // reference to the root of this BST
  protected Comparator<T> comp;   // used for all comparisons

  protected boolean found;   // used by remove
. . .
```

The Constructors

```
public BinarySearchTree()
// Precondition: T implements Comparable
// Creates an empty BST object - uses the natural order of elements.
{
   root = null;
   comp = new Comparator<T>()
   {
       public int compare(T element1, T element2)
       {
          return ((Comparable)element1).compareTo(element2);
       }
   };
}

public BinarySearchTree(Comparator<T> comp)
// Creates an empty BST object - uses Comparator comp for order of elements.
{
   root = null;
   this.comp = comp;
}
. . .
```

Example Observer methods

```
public boolean isFull()
// Returns false; this link-based BST is never full.
{
  return false;
}

public boolean isEmpty()
// Returns true if this BST is empty; otherwise, returns false.
{
  return (root == null);
}

public T min()
// If this BST is empty, returns null;
// otherwise returns the smallest element of the tree.
{
  if (isEmpty())
     return null;
  else
  {
     BSTNode<T> node = root;
     while (node.getLeft() != null)
       node = node.getLeft();
     return node.getInfo();
  }
}
```

7.5 Iterative versus Recursive Method Invocations

- Trees are inherently recursive; a tree consists of subtrees
- In this section we look at recursive and iterative approaches to the `size` method
- We then discuss the benefits of recursion versus iteration for this problem

Recursive Approach

- We create a public method, `size`, that calls a private recursive method, `recSize` and passes it a reference to the root of the tree.

```
public int size()
// Returns the number of elements in this BST.
{
  return recSize(root);
}
```

- We design the `recSize` method to return the number of nodes in the subtree referenced by the argument passed to it.

- Note that the number of nodes in a tree is:
  1 + number of nodes in left subtree + number of nodes in right subtree

`recSize` Algorithm Version 1

recSize(node): returns int

```
if (node.getLeft( ) is null) AND (node.getRight( ) is null)
    return 1
else
    return 1 + recSize(node.getLeft( )) + recSize(node.getRight( ))
```

- The corresponding method would crash when we try to access `node.getLeft` when `node` is `null`.

`recSize` Algorithm Version 2

```
recSize(node): returns int    Version 2
if (node is null)
    return 0
else
if (node.getLeft() is null) AND (node.getRight() is null)
    return 1
else
    return 1 + recSize(node.getLeft()) + recSize(node.getRight())
```

- Works, but can be simplified. There is no need to make the leaf node a special case – it would also be handled properly by the final else clause

`recSize` Algorithm Version 3

```
recSize(node): returns int    Version 3
if node is null
    return 0
else
    return 1 + recSize(node.getLeft( )) + recSize(node.getRight( ))
```

- Works and is “simple”.
- This example illustrates two important points about recursion with trees:
  - always check for the empty tree first
  - leaf nodes do not need to be treated as separate cases.

The `recSize` Code

```
private int recSize(BSTNode<T> node)
// Returns the number of elements in subtree rooted at node.
{
  if (node == null)
    return 0;
  else
    return 1 + recSize(node.getLeft()) + recSize(node.getRight());
}
```

Iterative Version

- We use a stack to hold nodes we have encountered but not yet processed
- We must be careful that we process each node in the tree exactly once. We follow these rules:
  - Process a node immediately after removing it from the stack.
  - Do not process nodes at any other time.
  - Once a node is removed from the stack, do not push it back onto the stack.

Code for the iterative approach

```
public int size()
// Returns the number of elements in this BST.
{
  int count = 0;
  if (root != null)
  {
    LinkedStack<BSTNode<T>> nodeStack = new LinkedStack<BSTNode<T>> ();
    BSTNode<T> currNode;
    nodeStack.push(root);
    while (!nodeStack.isEmpty())
    {
      currNode = nodeStack.top();
      nodeStack.pop();
      count++;
      if (currNode.getLeft() != null)
        nodeStack.push(currNode.getLeft());
      if (currNode.getRight() != null)
        nodeStack.push(currNode.getRight());
    }
  }
  return count;
}
```

Recursion or Iteration?

- *Is the depth of recursion relatively shallow?* **Yes**
- *Is the recursive solution shorter or clearer than the nonrecursive version?* **Yes**
- *Is the recursive version much less efficient than the nonrecursive version?* **No**

- This is a good use of recursion.

7.6 The Implementation Level: Remaining Operations

- In this section, we use recursion to implement the remaining Binary Search Tree operations
  - `contains`
  - `get`
  - `add`
  - `remove`
  - `iterator`
  - `getIterator`

The `contains` operation

- We implement `contains` using a private recursive method called `recContains` which
  - is passed the `target` we are searching for and a `node` representing the root of the subtree in which to search
  - has two base cases
    - if `node` is `null` returns `false`
    - if `node` contains `target` returns `true`
  - has two recursive cases
    - one searches the left subtree of `node`
    - one searches the right subtree of `node`

The `contains` method

```
public boolean contains (T target)
// Returns true if this BST contains a node with info i such that
// comp.compare(target, i) == 0; otherwise, returns false.
{
  return recContains(target, root);
}

private boolean recContains(T target, BSTNode<T> node)
// Returns true if the subtree rooted at node contains info i such that
// comp.compare(target, i) == 0; otherwise, returns false.
{
  if (node == null)
    return false;       // target is not found
  else if (comp.compare(target, node.getInfo()) < 0)
    return recContains(target, node.getLeft());   // Search left subtree
  else if (comp.compare(target, node.getInfo()) > 0)
    return recContains(target, node.getRight());  // Search right subtree
  else
    return true;        // target is found
}
```

The `get` method is similar

```
public T get(T target)
// Returns info i from node of this BST where comp.compare(target, i) == 0;
// if no such node exists, returns null.
{
  return recGet(target, root);
}

private T recGet(T target, BSTNode<T> node)
// Returns info i from the subtree rooted at node such that
// comp.compare(target, i) == 0; if no such info exists, returns null.
{
  if (node == null)
    return null;              // target is not found
  else if (comp.compare(target, node.getInfo()) < 0)
    return recGet(target, node.getLeft());          // get from left subtree
  else
  if (comp.compare(target, node.getInfo()) > 0)
    return recGet(target, node.getRight());         // get from right subtree
  else
    return node.getInfo();  // target is found
}
```

Iteration: Review of Traversal Definitions

- **Preorder traversal:** Visit the root, visit the left subtree, visit the right subtree
- **Inorder traversal:** Visit the left subtree, visit the root, visit the right subtree
- **Postorder traversal:** Visit the left subtree, visit the right subtree, visit the root

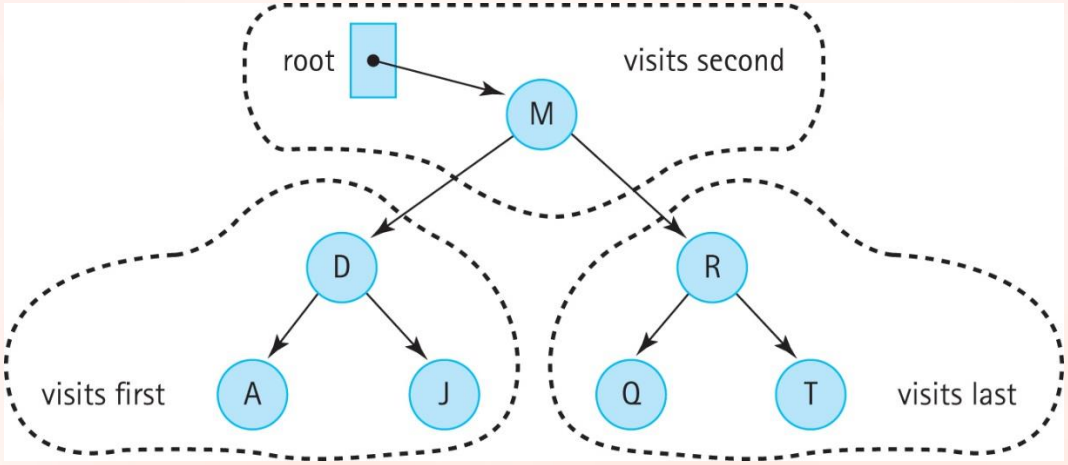
the `getIterator` method

- client passes `getIterator` an argument indicating one of the three traversal orders
- `getIterator` creates the appropriate `iterator` and returns it by traversing the tree in the desired order and building a queue of T
- It then creates an `iterator` using the anonymous inner class approach
- returned `iterator` represents a snapshot of the tree at the time `getIterator` is invoked and does not support `remove`

```
public Iterator<T> getIterator(BSTInterface.Traversal orderType)
{
  final LinkedQueue<T> infoQueue = new LinkedQueue<T>();
  if (orderType == BSTInterface.Traversal.Preorder)
    preOrder(root, infoQueue);
  else  if (orderType == BSTInterface.Traversal.Inorder)
    inOrder(root, infoQueue);
  else  if (orderType == BSTInterface.Traversal.Postorder)
    postOrder(root, infoQueue);

  return new Iterator<T>()
  {
    public boolean hasNext()
    {
      return !infoQueue.isEmpty();
    }

    public T next()
    {
       if (!hasNext())
        throw new IndexOutOfBoundsException("illegal invocation of next …
      return infoQueue.dequeue();
    }

    public void remove()
   {
      throw new UnsupportedOperationException("Unsupported remove attempted …
   }
  };
}
```

InOrder traversal

```
private void inOrder(BSTNode<T> node, LinkedQueue<T> q)
// Enqueues the elements from the subtree rooted at node into q in inOrder.
{  if (node != null)
  {
    inOrder(node.getLeft(), q);
    q.enqueue(node.getInfo());
    inOrder(node.getRight(), q);

} }
```



preOrder and postorder traversals

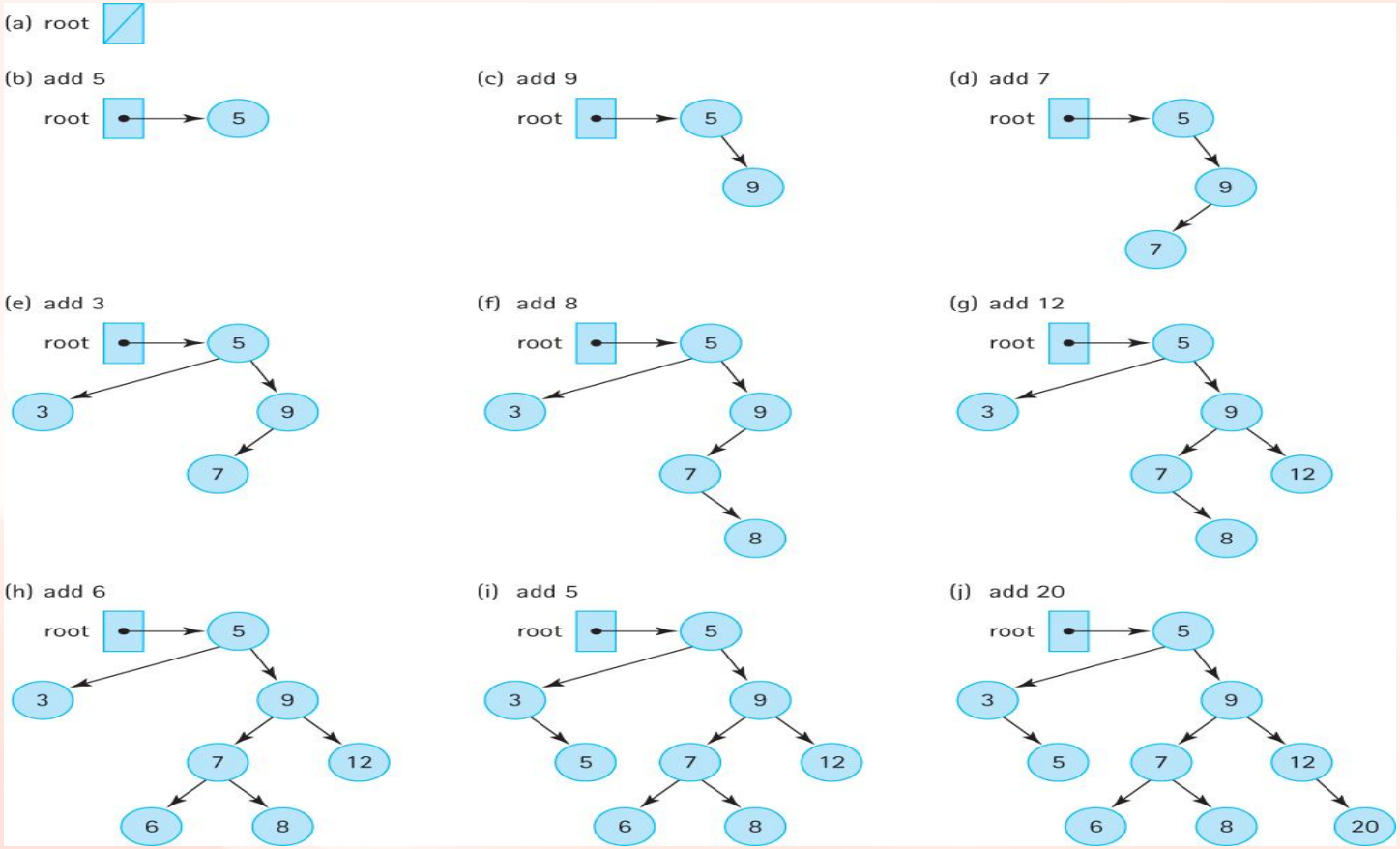
```
private void preOrder(BSTNode<T> node, LinkedQueue<T> q)
{
  if (node != null)
  {
    q.enqueue(node.getInfo());
    preOrder(node.getLeft(), q);
    preOrder(node.getRight(), q);
  }
}

private void postOrder(BSTNode<T> node, LinkedQueue<T> q)
{
  if (node != null)
  {
    postOrder(node.getLeft(), q);
    postOrder(node.getRight(), q);
    q.enqueue(node.getInfo());
  }
}
```

7.7 The Implementation Level: Transformers

- To complete our implementation of the Binary Search Tree ADT we need to create the transformer methods add and remove.
- These are the most complex operations.
- We use a similar approach as used in the subsection “Transforming a Linked List” in Section 3.4, “Recursive Processing of Linked Lists”

The `add` operation

- A new node is always inserted into its appropriate position in the tree as a leaf



The `add` operation

- The `add` method invokes the recursive method, `recAdd`, and passes it the element to be added plus a reference to the root of the tree.

```
public boolean add (T element)
// Adds element to this BST. The tree retains its BST property.
{
  root = recAdd(element, root);
  return true;
}
```

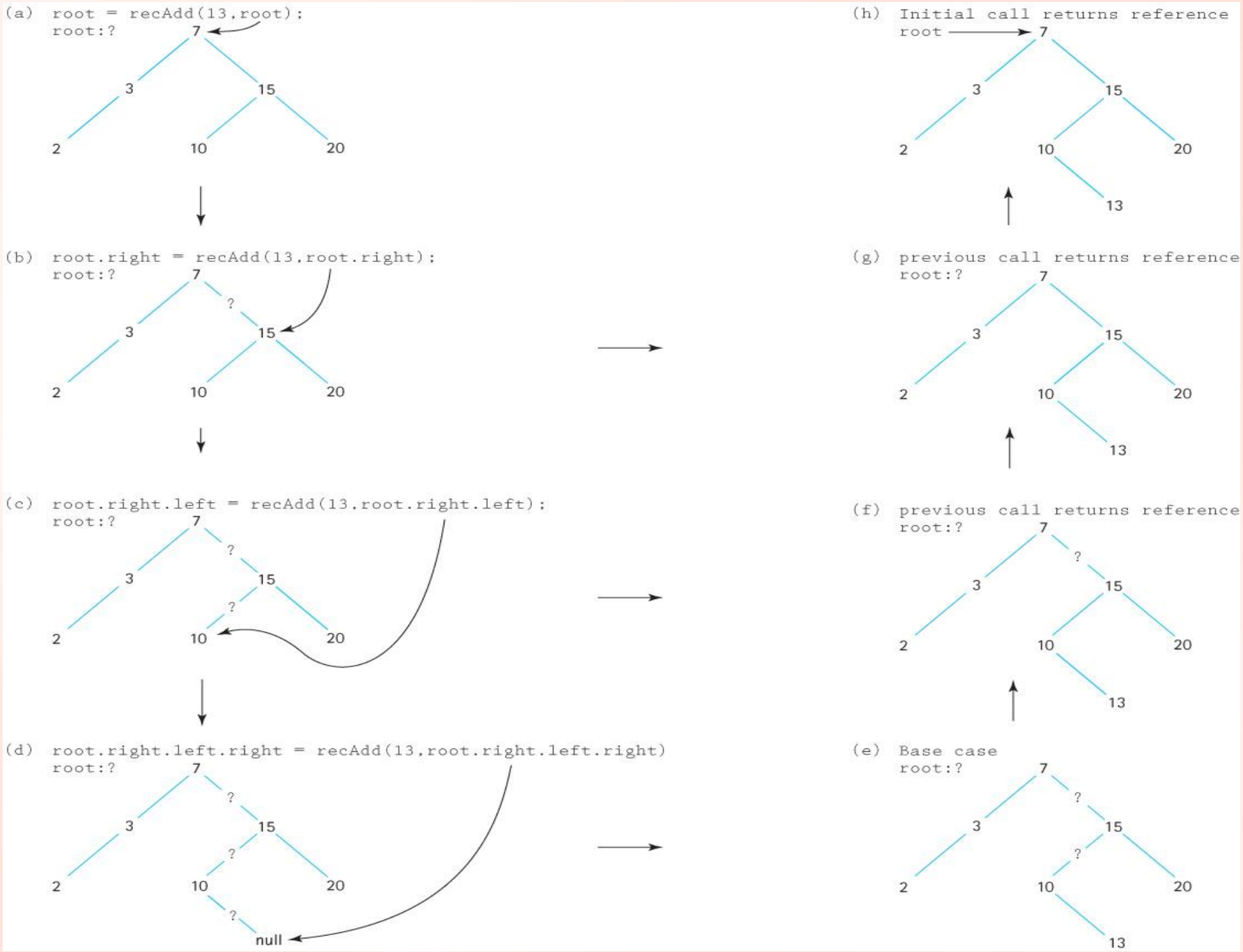
- The call to `recAdd` returns a `BSTNode`. It returns a reference to the new tree, that is, to the tree that includes element. The statement

```
root = recAdd(element, root);
```

can be interpreted as “Set the reference of the root of this tree to the root of the tree that is generated when element is added to this tree.”

The `add` method

```
private BSTNode<T> recAdd(T element, BSTNode<T> node)
// Adds element to tree rooted at node; tree retains its BST property.
{
  if (node == null)
    // Addition place found
    node = new BSTNode<T>(element);
  else if (element.compareTo(node.getInfo()) <= 0)
    node.setLeft(recAdd(element, node.getLeft()));     // Add in left subtree
  else
    node.setRight(recAdd(element, node.getRight()));   // Add in right subtree
  return tree;
}

public boolean add (T element)
// Adds element to this BST. The tree retains its BST property.
{
  root = recAdd(element, root);
  return true;
}
```


(a) root = recAdd(13,root);
root:?
7
3
15
2
10
20
(b) root.right = recAdd(13,root.right);
root:?
7
?
3
15
2
10
20
(c) root.right.left = recAdd(13,root.right.left);
root:?
7
?
3
15
?
2
10
20
(d) root.right.left.right = recAdd(13,root.right.left.right)
root:?
7
?
3
15
?
2
10
20
?
null
(e) Base case
root:?
7
?
3
15
?
2
10
20
13
(f) previous call returns reference
root:?
7
?
3
15
2
10
20
13
(g) previous call returns reference
root:?
7
3
15
2
10
20
13
(h) Initial call returns reference
root
7
3
15
2
10
20
13

The `remove` Operation

- The most complicated of the binary search tree operations.
- We must ensure when we remove an element we maintain the binary search tree property.

The code for `remove`:

- The set up for the remove operation is the same as that for the add operation.
- The private `recRemove` method is invoked from the public `remove` method with arguments equal to the target element to be removed and the root of the tree to remove it from.
- The recursive method returns a reference to the revised tree
- The `remove` method returns the boolean value stored in `found`, indicating the result of the remove operation.

```
public boolean remove (T target)
// Removes a node with info i from tree such that
// comp.compare(target,i) == 0 and returns true;
// if no such node exists, returns false.
{
  root = recRemove(target, root);
  return found;
}
```

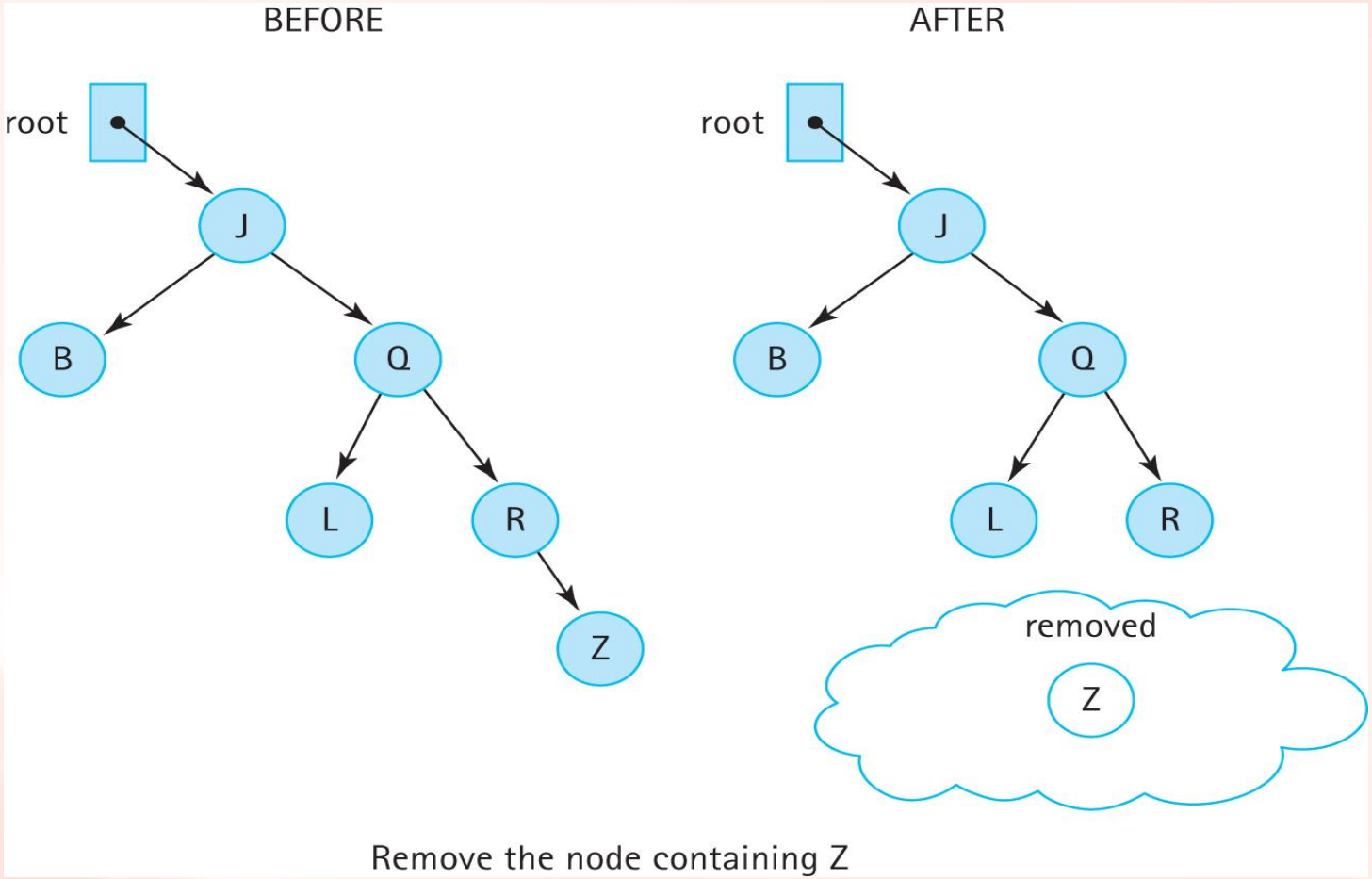
The `recRemove` method

```
private BSTNode<T> recRemove(T target, BSTNode<T> node)
// Removes element with info i from tree rooted at node such that
// comp.compare(target, i) == 0 and returns true;
// if no such node exists, returns false.
{
   if (node == null)
      found = false;
   else if (comp.compare(target, node.getInfo()) < 0)
      node.setLeft(recRemove(target, node.getLeft()));
   else if (comp.compare(target, node.getInfo()) > 0)
      node.setRight(recRemove(target, node.getRight()));
   else
   {
      node = removeNode(node);
      found = true;
   }
   return node;
}
```

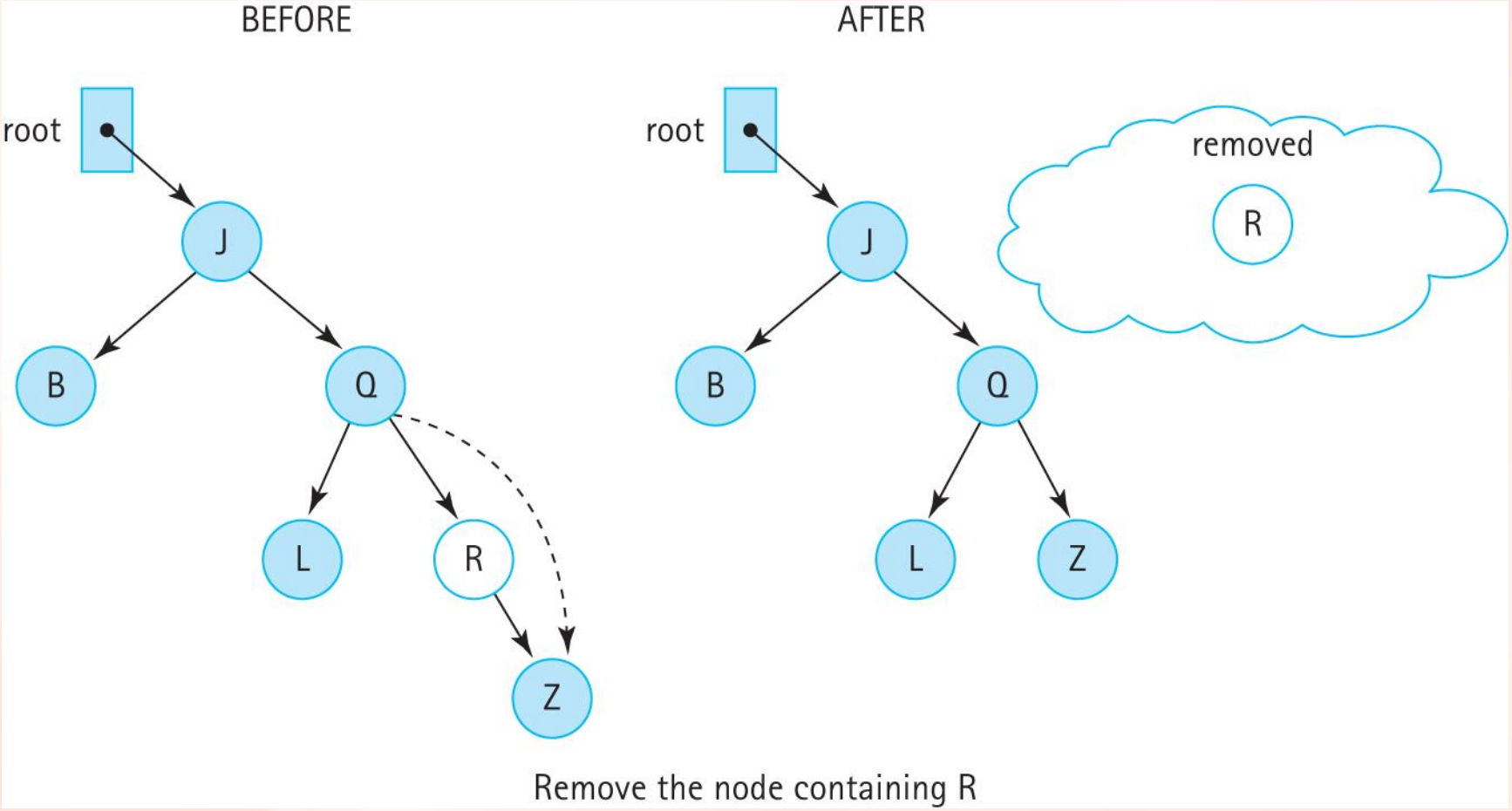
Three cases for the `removeNode` operation

- Removing a leaf (no children): removing a leaf is simply a matter of setting the appropriate link of its parent to null.
- Removing a node with only one child: make the reference from the parent skip over the removed node and point instead to the child of the node we intend to remove
- Removing a node with two children: replaces the node's info with the info from another node in the tree so that the search property is retained - then remove this other node

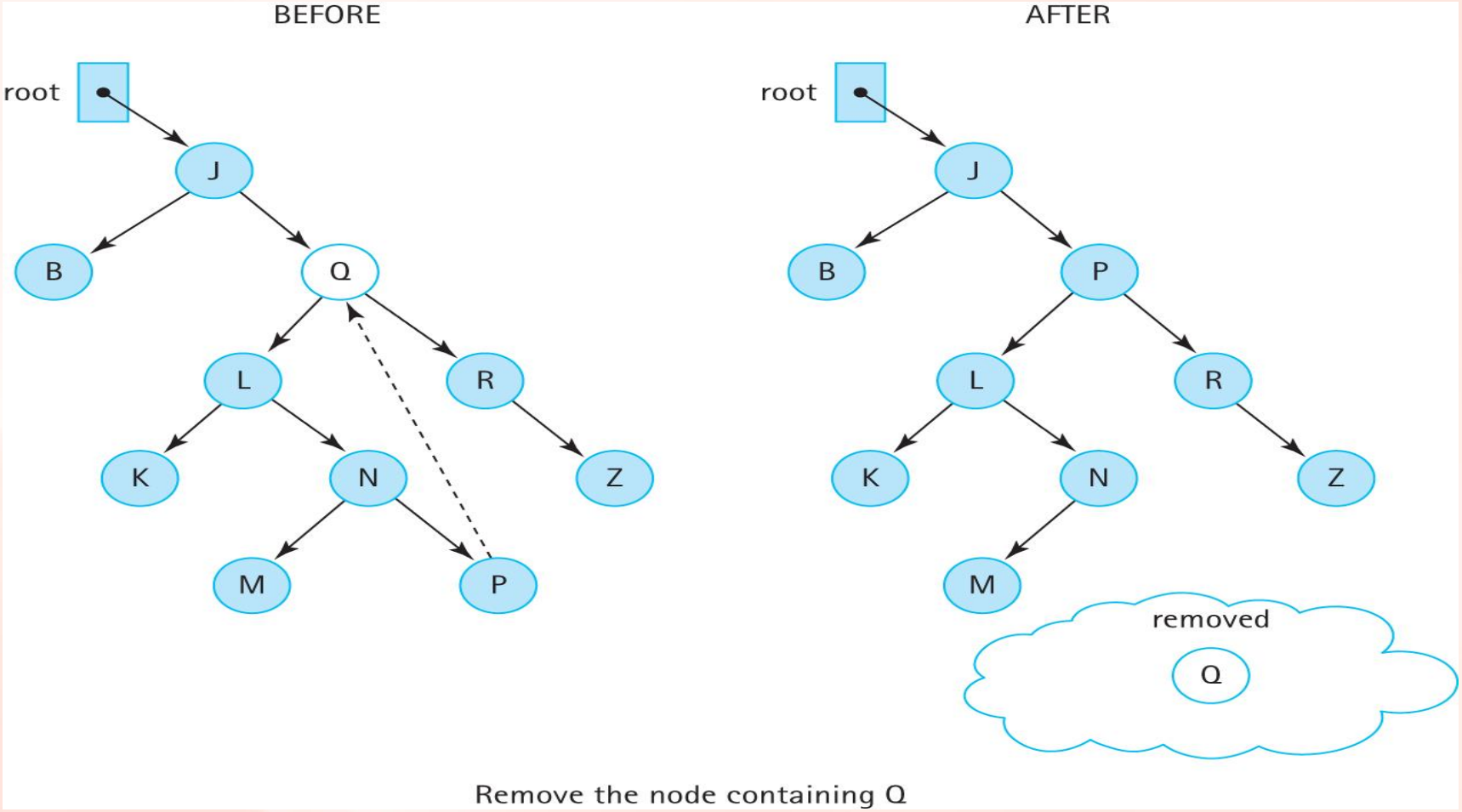
Removing a Leaf Node



Removing a node with one child



Removing a Node With Two Children



The Remove Node Algorithm

removeNode (node): returns BSTNode

```
if (node.getLeft( ) is null) AND (node.getRight( ) is null)
    return null
else if node.getLeft( ) is null
    return node.getRight( )
else if node.getRight( ) is null
    return node.getLeft( )
else
    Find predecessor
    node.setInfo(predecessor.getInfo( ))
    node.setLeft(recRemove(predecessor.getInfo( ), node.getLeft( )))
return node
```

Note: We can remove one of the tests if we notice that the action taken when the left child reference is null also takes care of the case in which both child references are null. When the left child reference is null, the right child reference is returned. If the right child reference is also null, then null is returned, which is what we want if they are both null.

The removeNode method

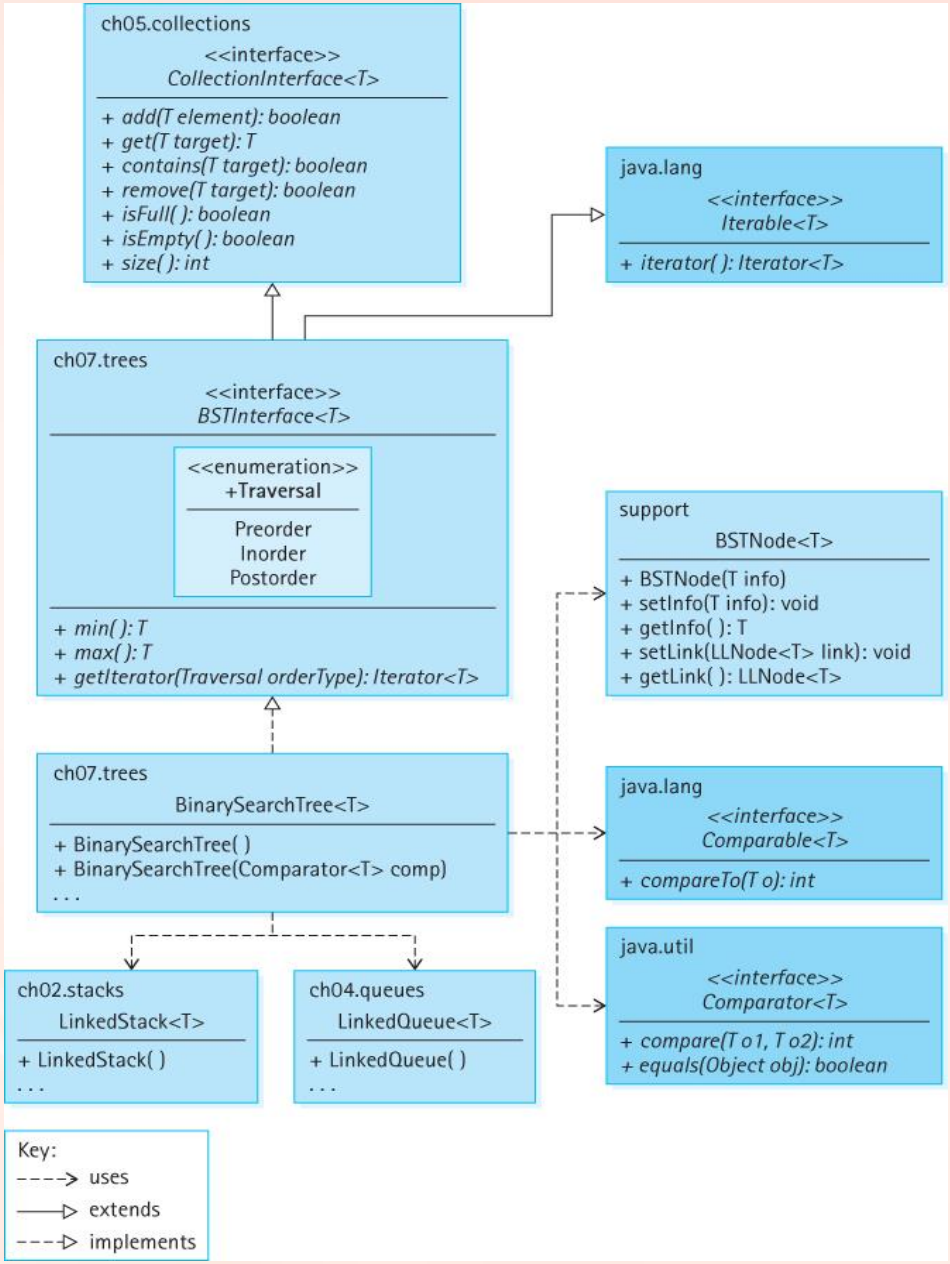
```
private BSTNode<T> removeNode(BSTNode<T> node)
// Removes the information at node from the tree.
{
  T data;
  if (node.getLeft() == null)
    return node.getRight();
  else if (node.getRight() == null)
    return node.getLeft();
  else
  {
    data = getPredecessor(node.getLeft());
    node.setInfo(data);
    node.setLeft(recRemove(data, node.getLeft()));
    return node;
  }
}
```

The `getPredecessor` method

- The logical predecessor is the maximum value in node's left subtree.
- The maximum value in a binary search tree is in its rightmost node.
- Therefore, given node's left subtree, we just keep moving right until the right child is null.
- When this occurs, we return the info reference of the node.

```
private T getPredecessor(BSTNode<T> subtree)
// Returns the information held in the rightmost node of subtree
{
  BSTNode temp = subtree;
  while (temp.getRight() != null)
    temp = temp.getRight();
  return temp.getInfo();
}
```

Our Binary Search Tree Architecture



7.8 Binary Search Tree Performance

- A binary search tree is an appropriate structure for many of the same applications discussed previously in conjunction with sorted lists.
- Similar to a sorted array-based list, it can be searched quickly, using a binary search.
- Similar to a linked list, it allows insertions and removals without having to move large amounts of data.
- There is a space cost - the binary search tree, with its extra reference in each node, takes up more memory space than a singly linked list.

Text Analysis Experiment Revisited

Table 7.2 Results of Vocabulary Density Experiment

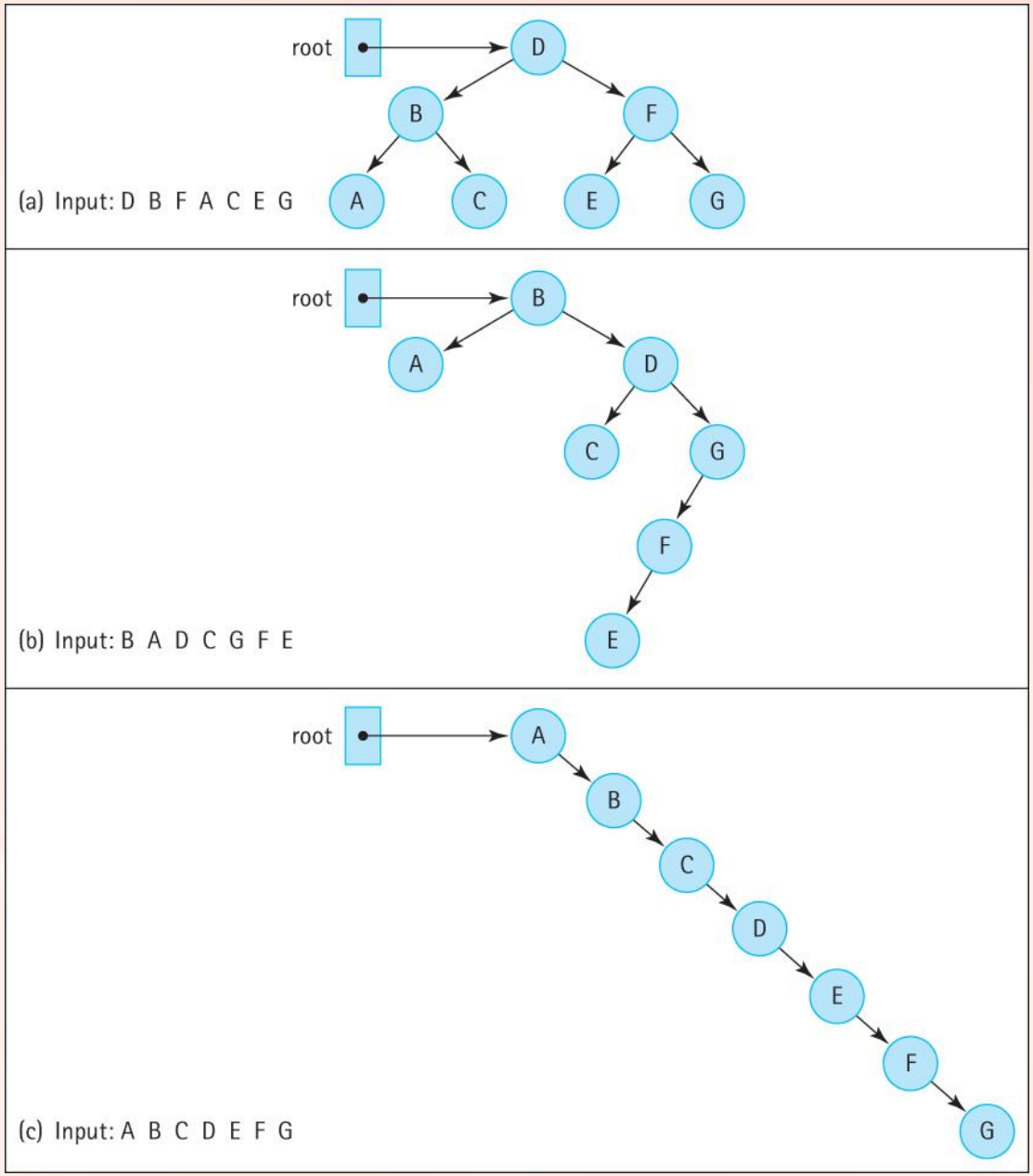
Source	File Size	Array	Sorted Array	Binary Search Tree
Shakespeare's18th Sonnet	1 KB	20 msecs	23 msecs	22 msecs
Shakespeare's *Hamlet*	177 KB	236 msecs	128 msecs	127 msecs
Linux Word File	400 KB	9,100 msecs	182 msecs	Stack overflow error, 33,760 msecs with revision
Melville's *Moby-Dick*	1,227 KB	2,278 msecs or 2.3 seconds	382 msecs	334 msecs
Complete Works of William Shakespeare	5,542 KB	9.7 seconds	1.2 seconds	0.9 seconds
Webster's Unabridged Dictionary	28,278 KB	4.7 minutes	9.5 seconds	4.2 seconds
11th Edition of the *Encyclopaedia Britannica*	291,644 KB	56.4 minutes	2.5 minutes	41.8 seconds
Mashup	608,274 KB	10 hours	7.2 minutes	1.7 minutes

Text Analysis Experiment Revisited

- Due to operation complexity, performance gains from the binary search tree are only clearly evident as the size of the file increases.
- The table also reveals a serious issue when the binary search tree structure is used for the Linux Word file. The application bombs—it stops executing and reports a “Stack overflow error.”
- The problem is that the underlying tree is completely skewed.

Insertion Order and Tree Shape

Linux Word File is like this →



Balancing a Binary Search Tree

- A beneficial addition to our Binary Search Tree ADT operations is a `balance` operation
- The specification of the operation is:

```
public balance();
// Restructures this BST to be optimally balanced
```

- It is up to the client program to use the `balance` method appropriately

Our Approach

- Basic algorithm:
  - Save the tree information in an array
  - Insert the information from the array back into the tree
- The structure of the new tree depends on the order that we save the information into the array, or the order in which we insert the information back into the tree, or both
- We save the information “in order” so that the array is sorted

To Ensure a Balanced Tree

- Even out as much as possible, the number of descendants in each node's left and right subtrees
- First insert the "middle" item of the inOrder array
  - Then insert the left half of the array using the same approach
  - Then insert the right half of the array using the same approach

Our Balance Tree Algorithm

Balance

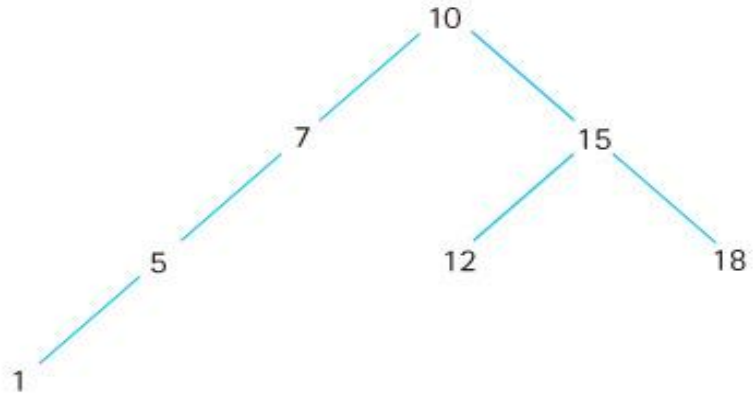
```
Iterator iter = tree.getIterator(Inorder )
int index = 0
while (iter.hasnext())
     array[index] = iter.next( )
     index++
tree = new BinarySearchTree()
tree.InsertTree(0, index - 1)
```

InsertTree(low, high)

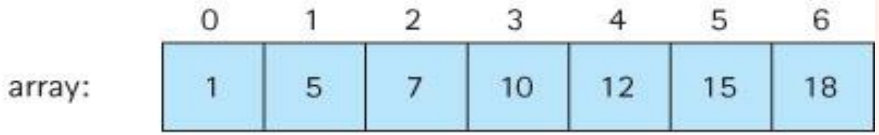
```
if (low == high)                  // Base case 1
    tree.add(array[low])
else if ((low + 1) == high)    // Base case 2
    tree.add(array[low])
    tree.add(array[high])
else
    mid = (low + high) / 2
    tree.add(array[mid])
    tree.InsertTree(low, mid – 1)
    tree.InsertTree(mid + 1, high)
```

Using recursive `insertTree`

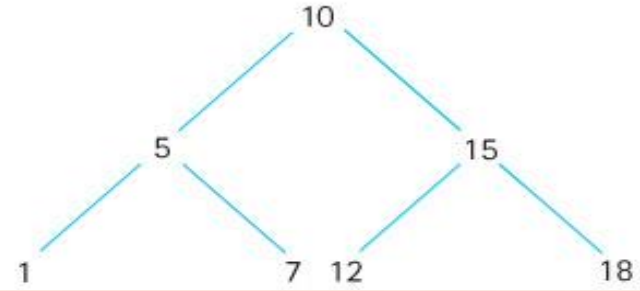
(a) The original tree



(b) The inorder traversal



(c) The resultant tree if InsertTree (0,6) is used



7.9 Application: Word Frequency Counter

- In this section we develop a word frequency counter that uses our binary search tree
- Read a text file and generate an alphabetical listing of the unique words, along with a count of how many times each word occurs
- We allow users to specify a minimum word size and a minimum frequency count
- Present a few summary statistics

The WordFreq Class

```
package support;
import java.text.DecimalFormat;

public class WordFreq implements Comparable<WordFreq>
{
  private String word;  private int freq;
  DecimalFormat fmt = new DecimalFormat("00000");

  public WordFreq(String newWord)
  {
    word = newWord;    freq = 0;
  }

  public String getWordIs(){return word;}

  public int getFreq(){return freq;}

  public void inc()  {    freq++;  }

  public int compareTo(WordFreq other)
  {
    return this.word.compareTo(other.word);
  }

  public String toString()
  {
    return(fmt.format(freq) + " " + word);
  }
}
```

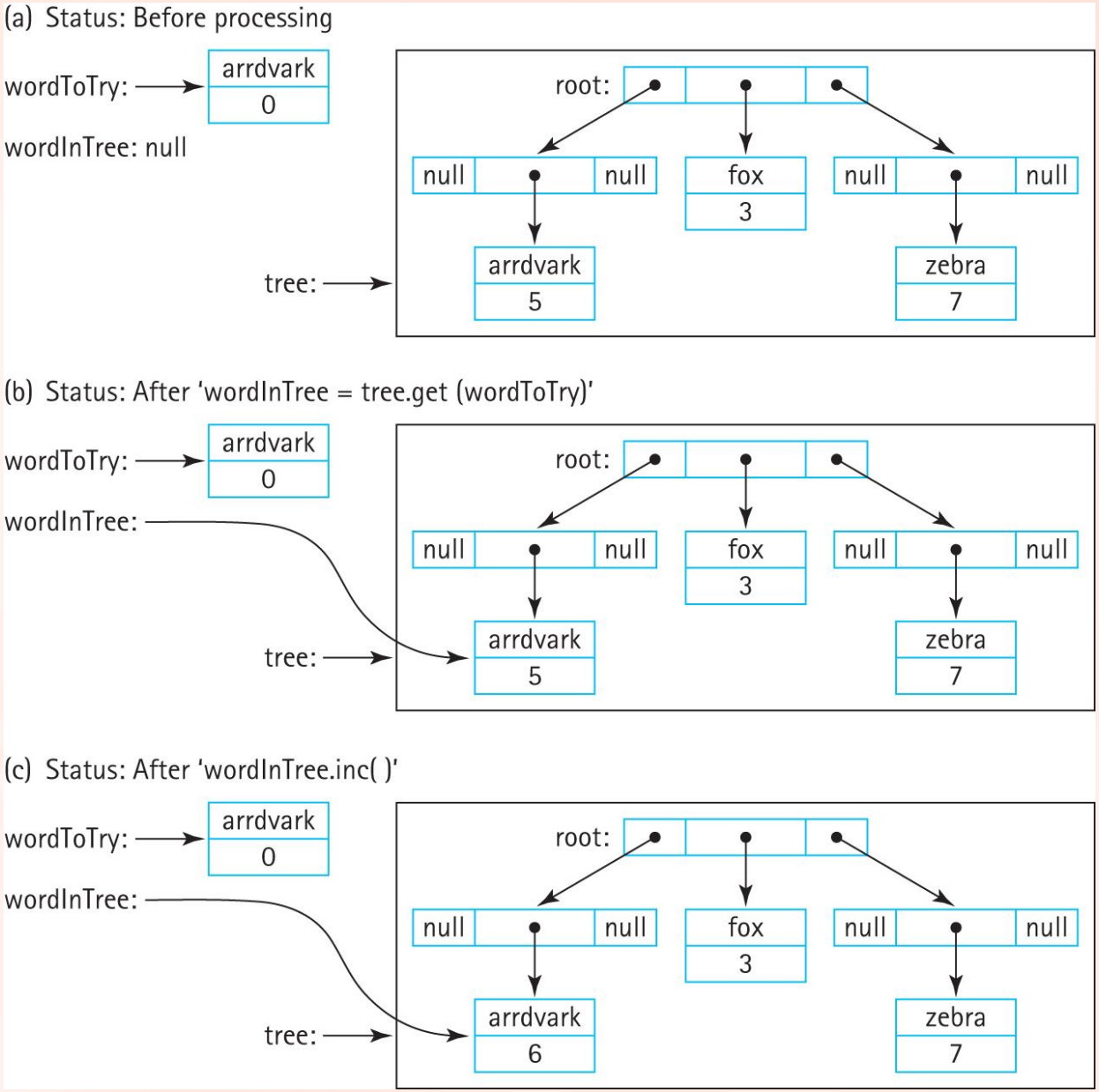
The Application

- Scans the input file for words, and after reading a word, it checks to see if a match is already in the tree, and if not it inserts a WordFreq object that holds the word and a frequency count of 1
- If the word is already in the tree, increment the frequency associated with the word

Algorithm to process a word

```
wordToTry = new WordFreq(word)
wordInTree = tree.get(wordToTry)
if (wordInTree != null)
   wordInTree.inc()
else
   wordToTry.inc()  // to set its frequency to 1
   tree.add(wordToTry);
```

Processing a word



The Application

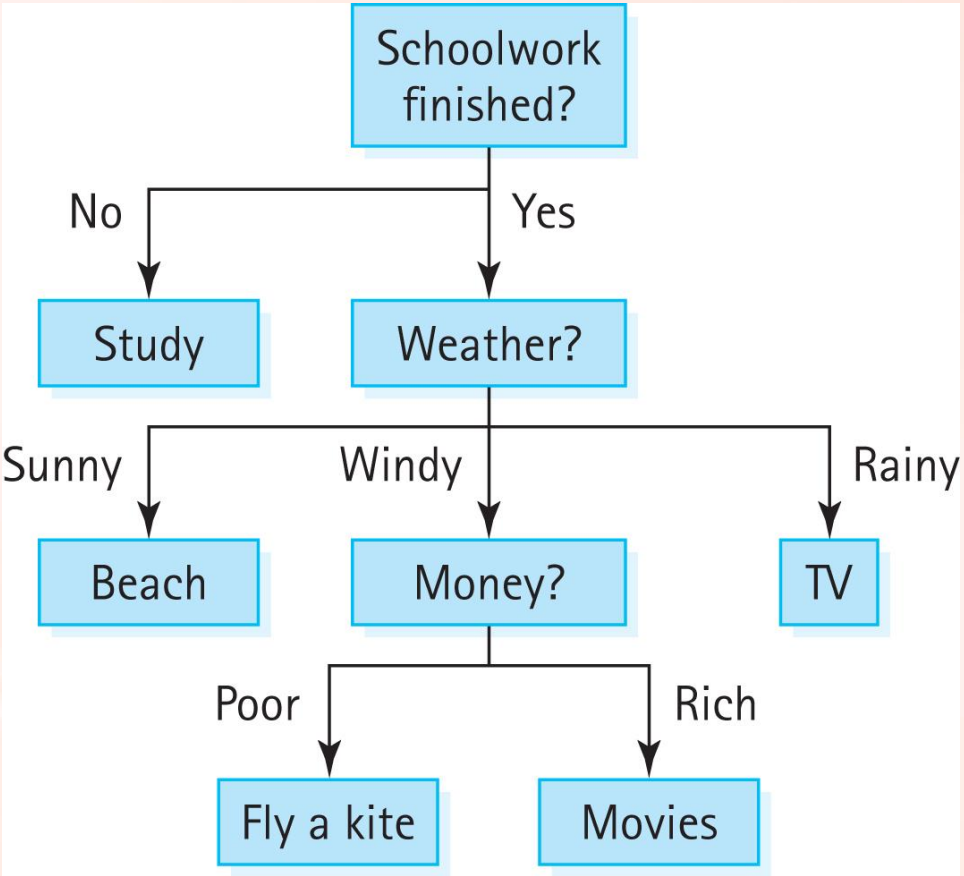
- Instructors can now review the code and demonstrate/discuss the application.

7.10 Tree Variations

- Trees can be binary, trinary (up to three children per node), n-ary (up to n children per node), alternating ary, balanced, not balanced, partially balanced, self-adjusting, and store information in their nodes or their edges or both.
- The Java Library does not include a general tree structure and neither do most textbooks.
- Often with trees we let the specific target application dictate the definition, assumptions, rules, and operations associated with a specific implementation.

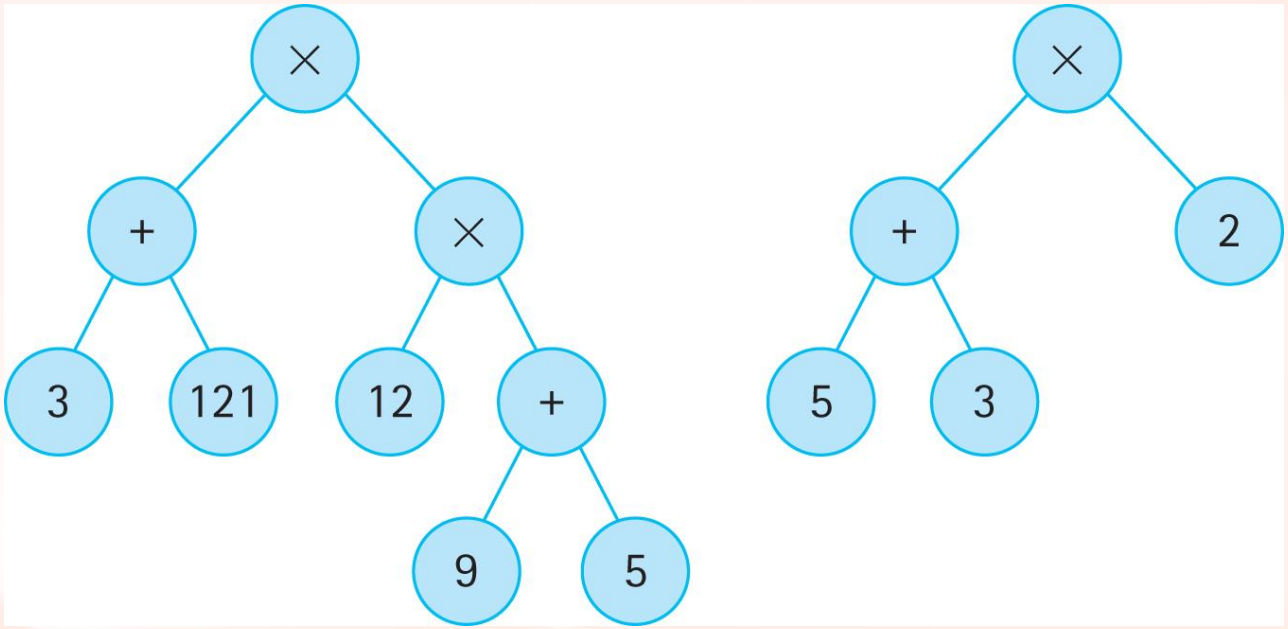
Application-Specific Variations

- Decision Trees:



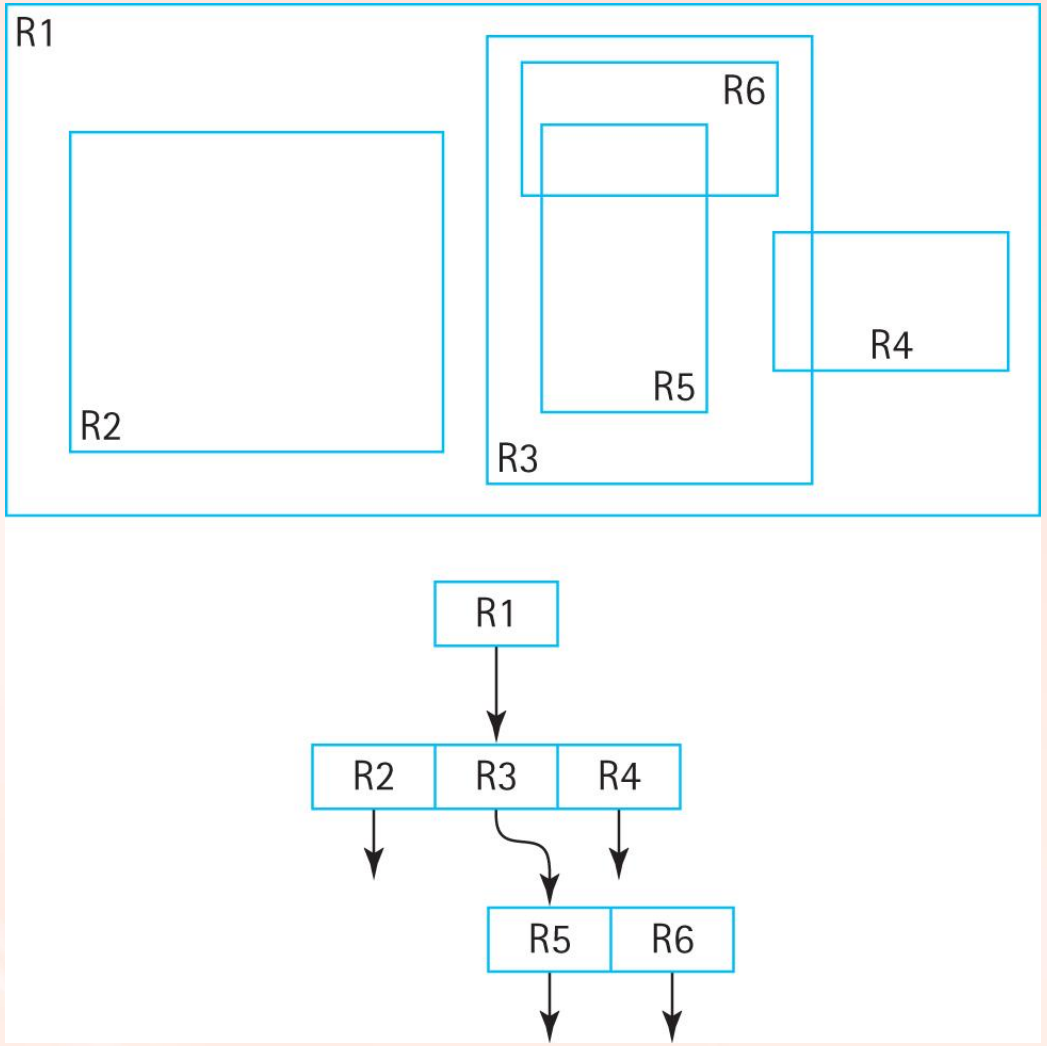
Application-Specific Variations

- Expression/Parse Trees:



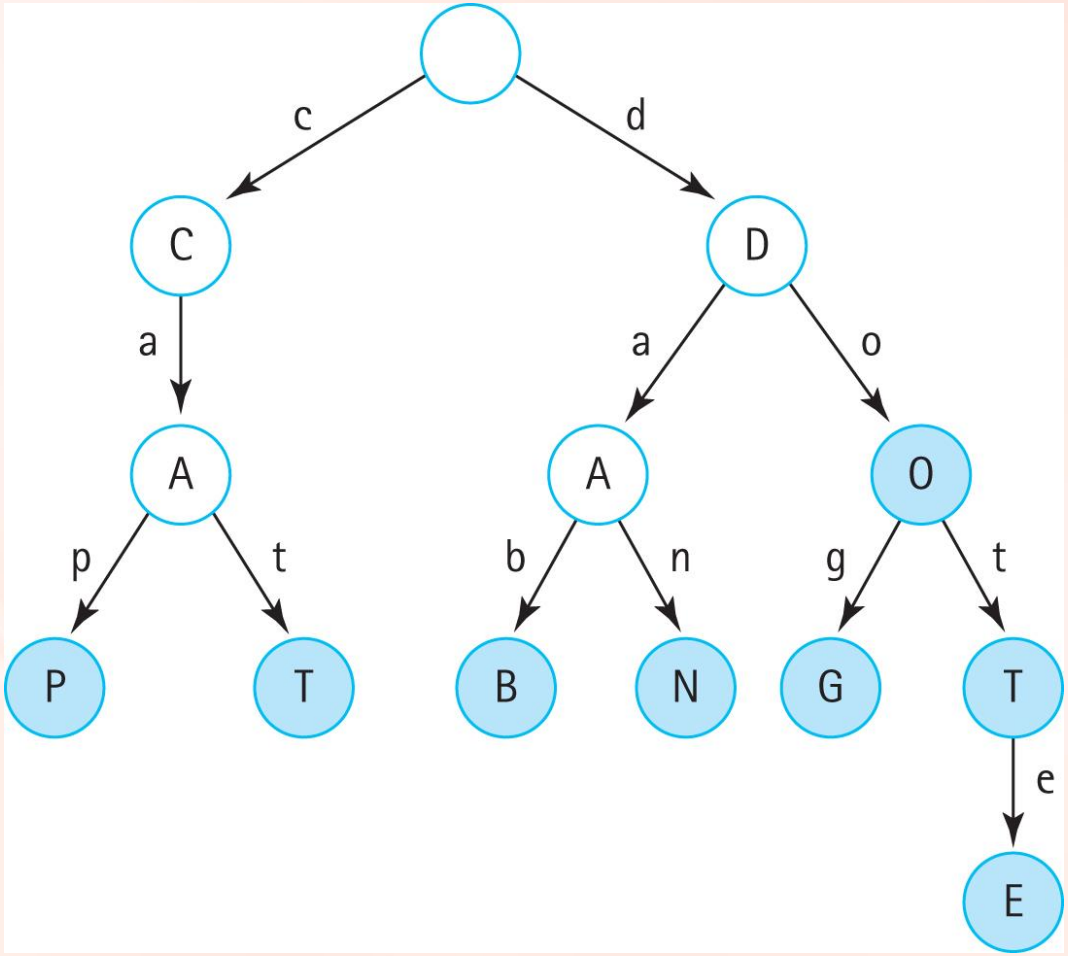
Application-Specific Variations

- R Trees:



Application-Specific Variations

- Tries/Prefix Trees:

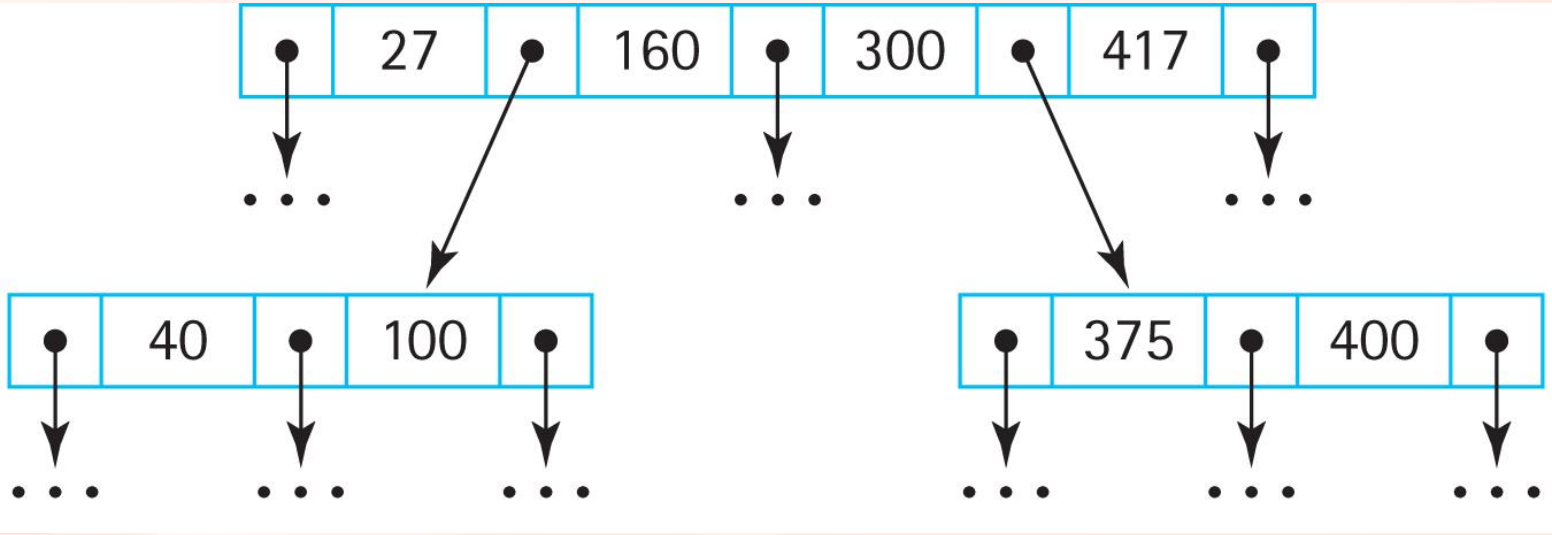


Balanced Search Trees

- The Binary Search Tree ADT presented in this chapter is an excellent collection structure for holding “random” data, but suffers from one major weakness—it can become unbalanced.
- Many search tree variations have been invented to address this weakness, each in turn with their own strengths and weaknesses.

Balanced Search Trees

- B-Trees: a search tree that allows internal nodes to hold multiple values and have multiple children



Balanced Search Trees

- B-Tree Variants: A popular variant of the B-Tree is the 2-3-4 tree, where nodes are constrained to hold 1, 2, or 3 values and therefore have 2, 3, or 4 subtrees (thus the name of the tree)
- Searching, insertion, and removal of information is always $O(\log_2 N)$ and the amount of work required to process the information within any node is constant.

Balanced Search Trees

- A 2-3-4 tree can be implemented using a binary tree structure called a red-black tree. In this structure nodes are “colored” either red or black (each node must hold an extra boolean value that indicates this information). The color of a node indicates how its value fits into the corresponding 2-3-4 tree.
- The Java Library includes two important classes that use tree implementations. The Java TreeMap class supports maps (see Chapter 8) and the Java TreeSet class supports sets (see Section 5.7, “Collection Variations”). In both cases the internal representation is a red-black tree.

Balanced Search Trees

- AVL Trees: the difference in height between a node's two subtrees can be at most 1.
- During addition and removal of nodes extra work is required to ensure the tree remains balanced.