Lesson 06

TREES

Wholeness Statement

A binary search tree is an important data structure that provides a highly flexible perspective on a set of comparable objects.

The whole range of space and time is open to an individual with fully developed awareness.

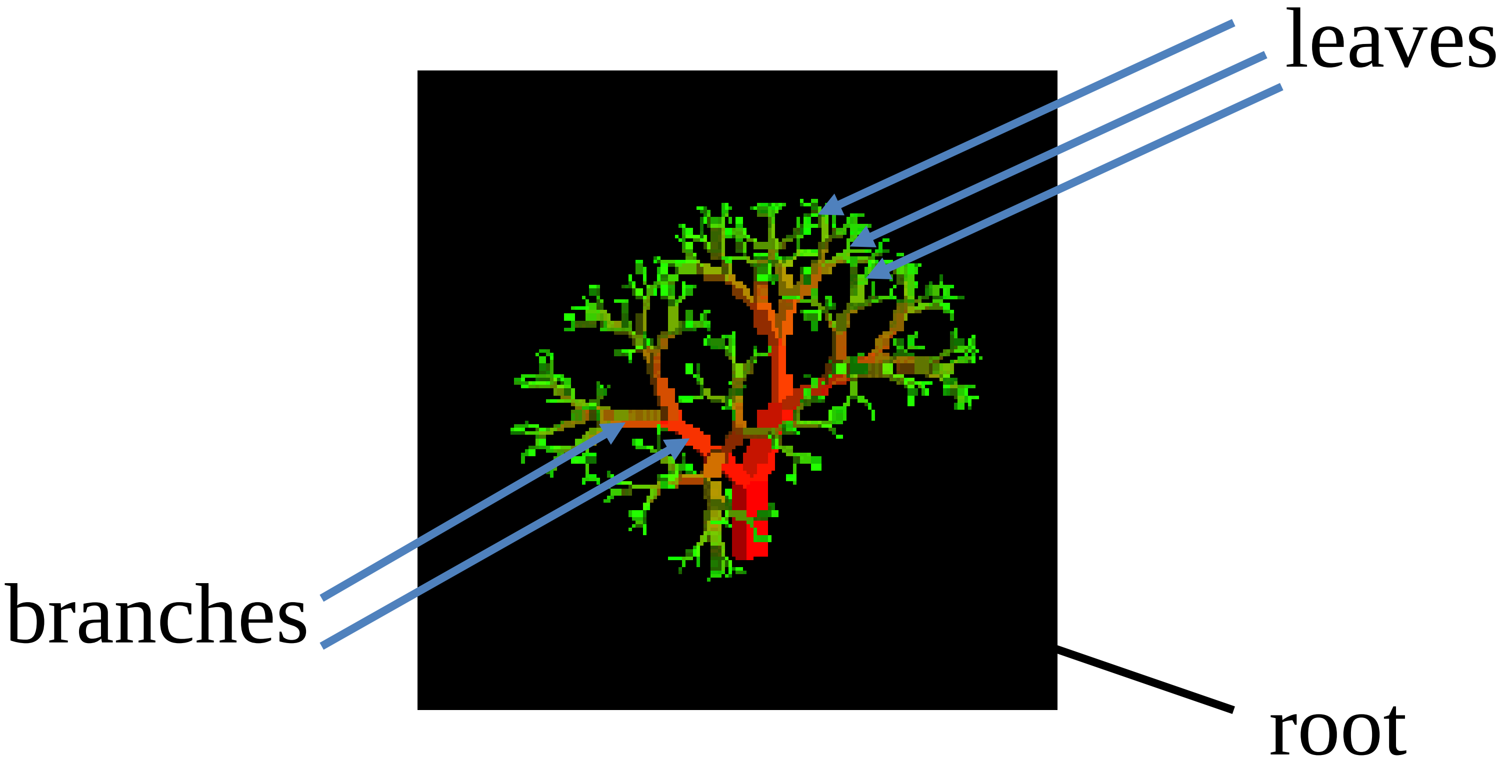
Chapter Objectives

- To learn how to use a tree to represent a hierarchical organization of information
- To learn how to use recursion to process trees
- To understand the different ways of traversing a tree
- To understand the differences between binary trees, binary search trees, and heaps
- To learn how to implement binary trees, binary search trees, and heaps using linked data structures and arrays

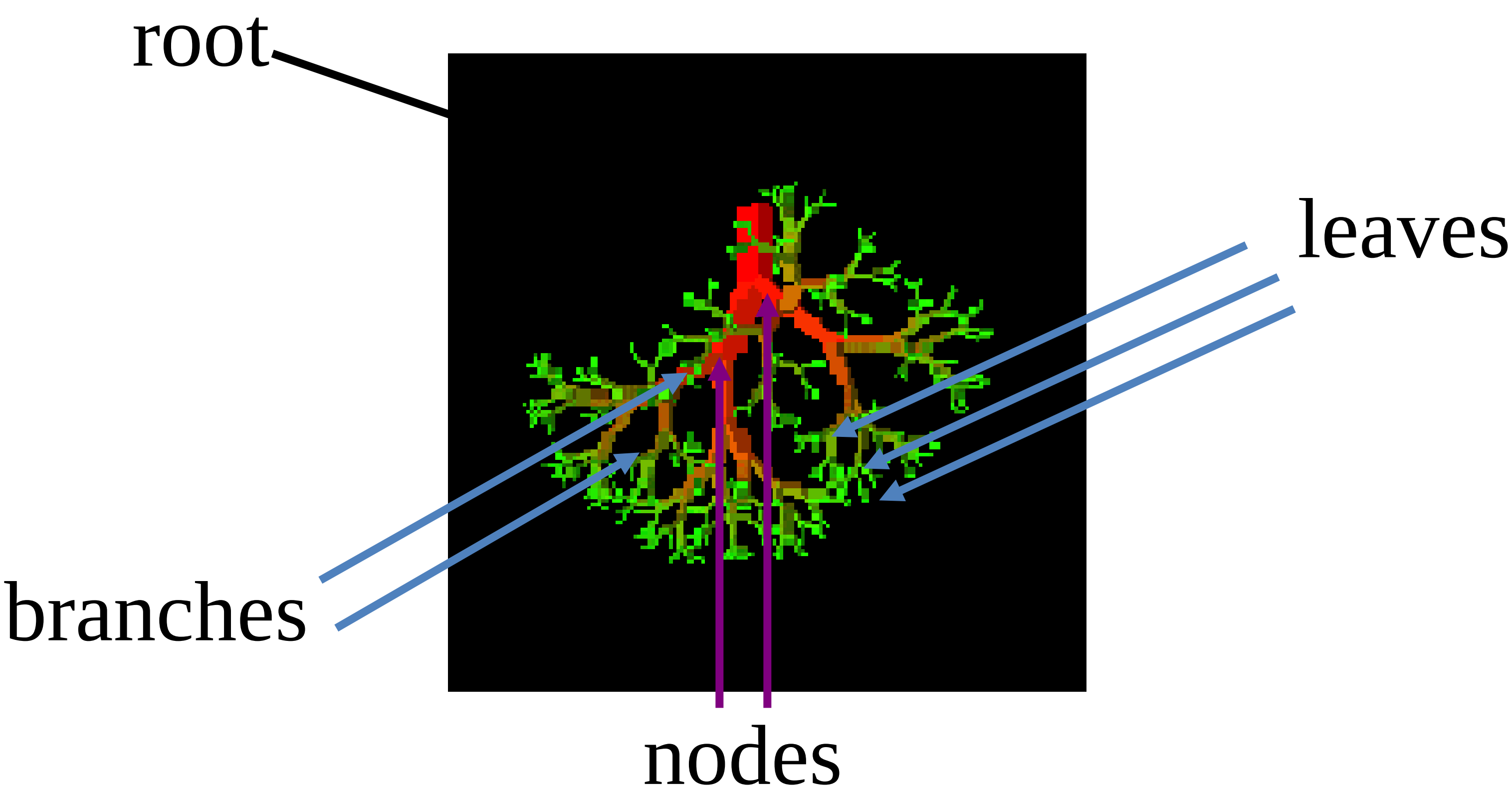
What is the Tree ADT

Section 6.1

Nature View of a Tree



Computer Scientist's View



Trees - Introduction

- Trees can represent hierarchical organizations of information:
  - class hierarchy
  - disk directory and subdirectories
  - family tree
- Trees are recursive data structures because they can be defined recursively
- Many methods to process trees are written recursively

What is a Tree ADT

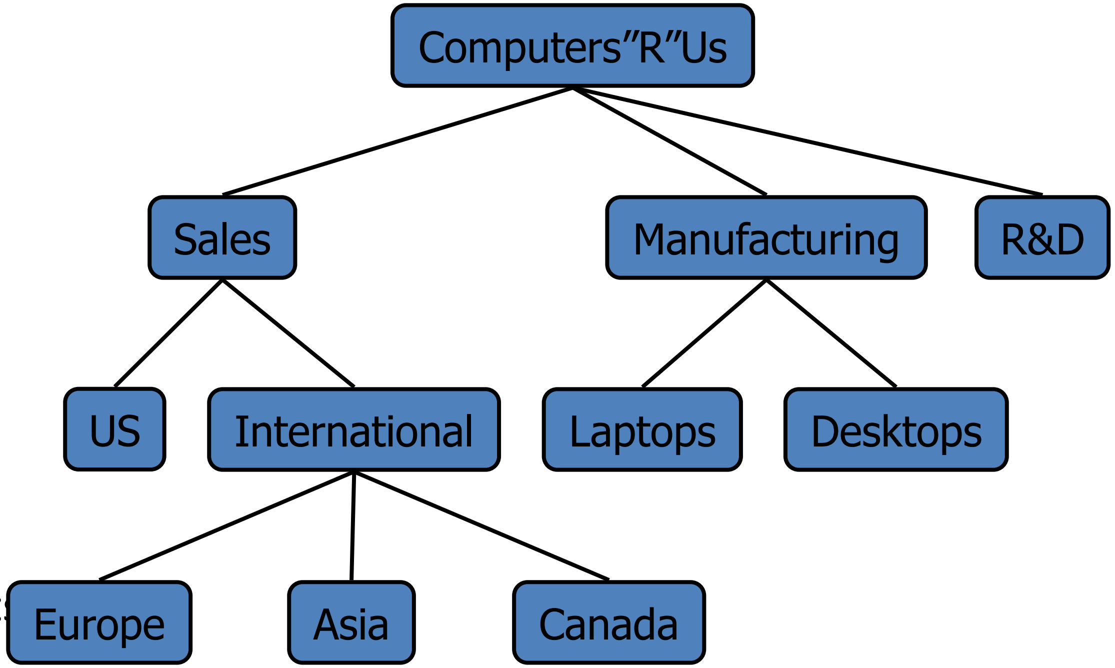
A tree is a finite nonempty set of elements.

It is an abstract model of a hierarchical structure. (nonlinear)

consists of nodes with a parent-child relation.

Applications:

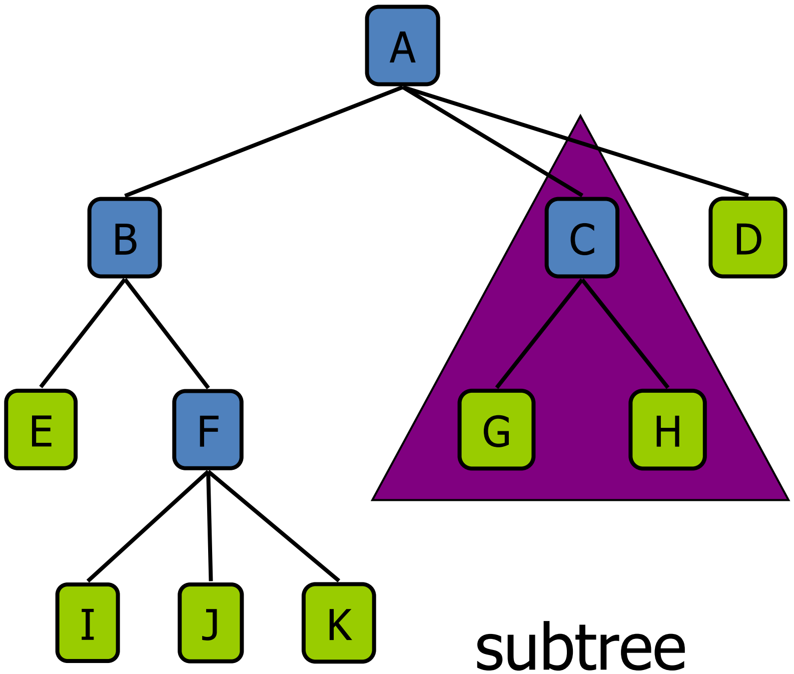
- Organization charts
- File systems
- Programming environments



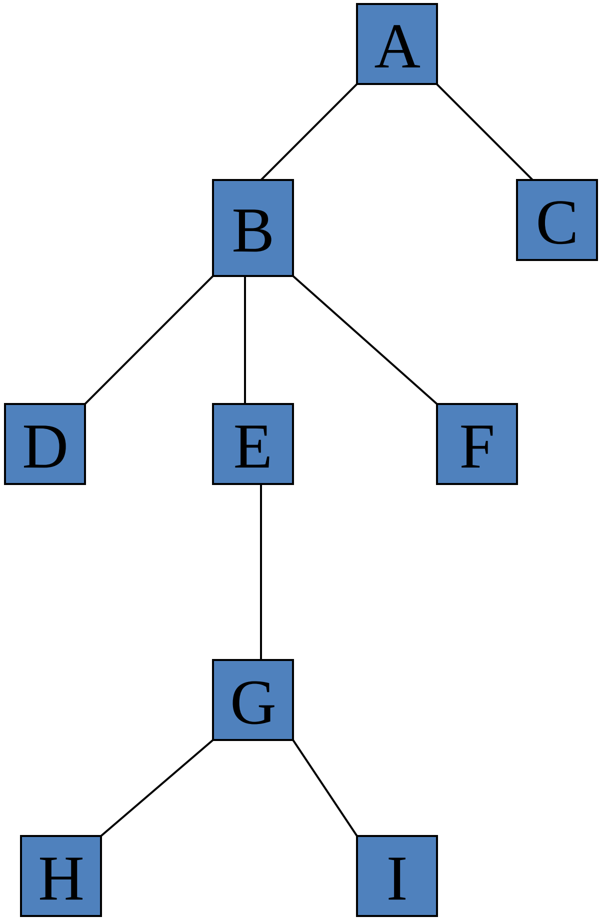
Tree Terminology

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
- **Degree** of a node: the number of its children
- **Degree** of a tree: the maximum number of its node.

- **Subtree**: tree consisting of a node and its descendants



Tree Properties



Property	Value
Number of nodes	
Height	
Root Node	
Leaves	
Interior nodes	
Ancestors of H	
Descendants of B	
Siblings of E	
Right subtree of A	
Degree of this tree	

Intuitive Representation of Tree Node

- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  - The root comes first, followed by a list of links to sub-trees

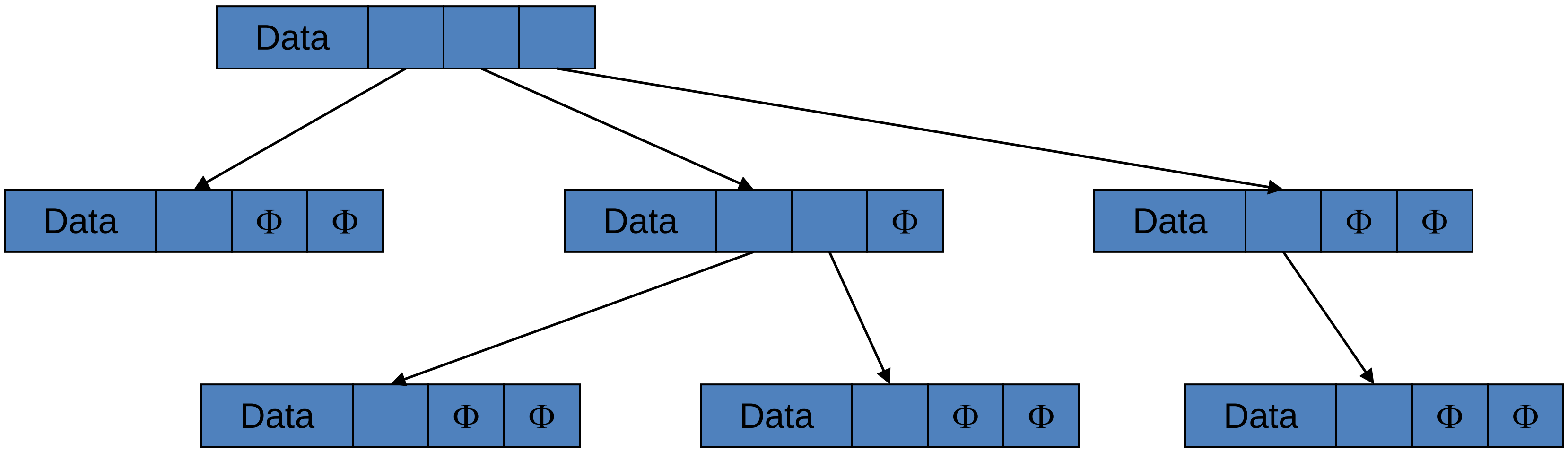
How many link fields are needed in such a representation?

Data	Link 1	Link 2	…	Link n

Trees

Every tree node:

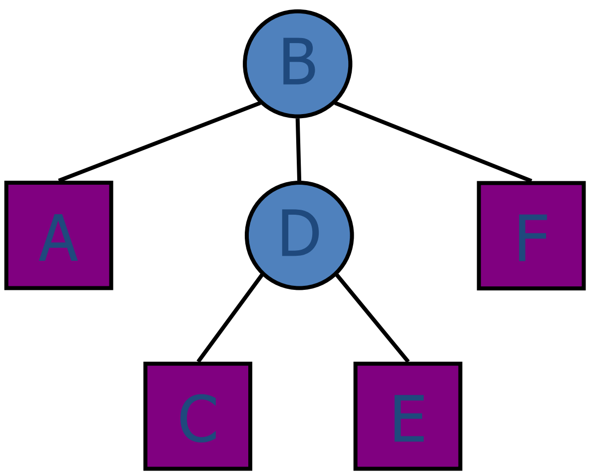
- object – useful information
- children – pointers to its children

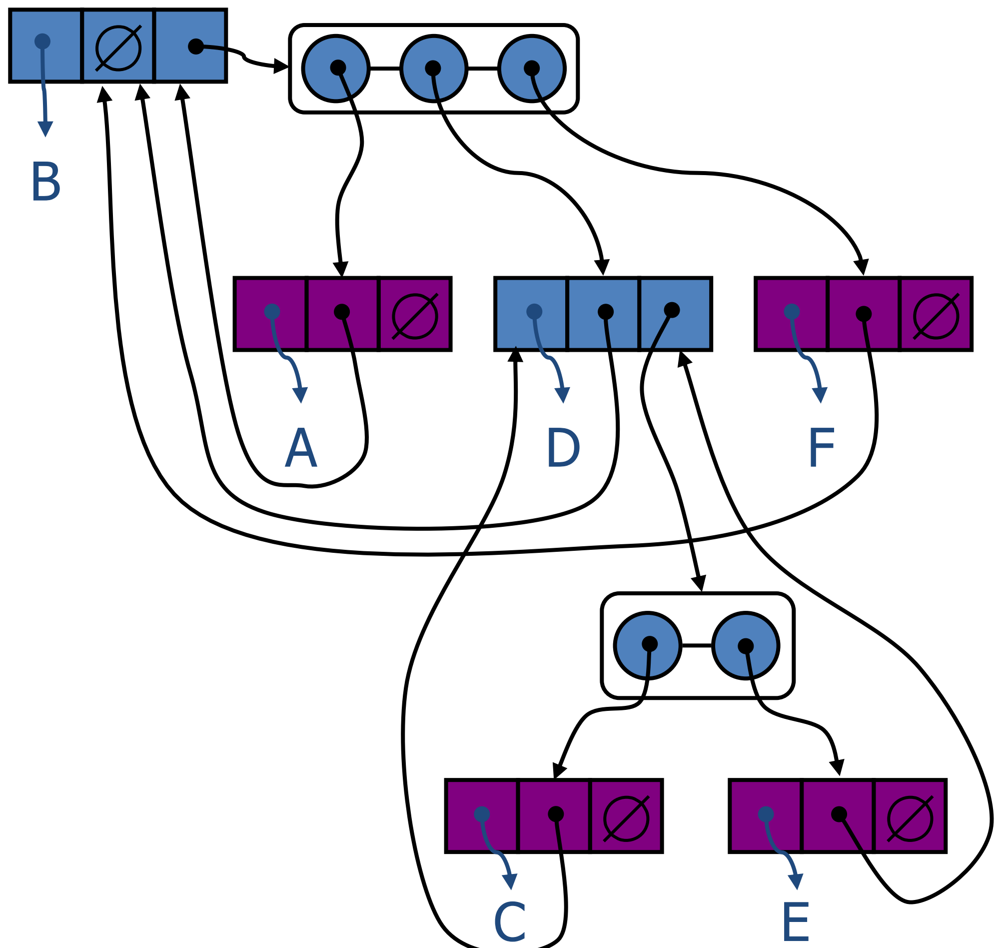


A Tree Representation

A node is represented by an object storing

- Element
- Parent node
- Sequence of children nodes

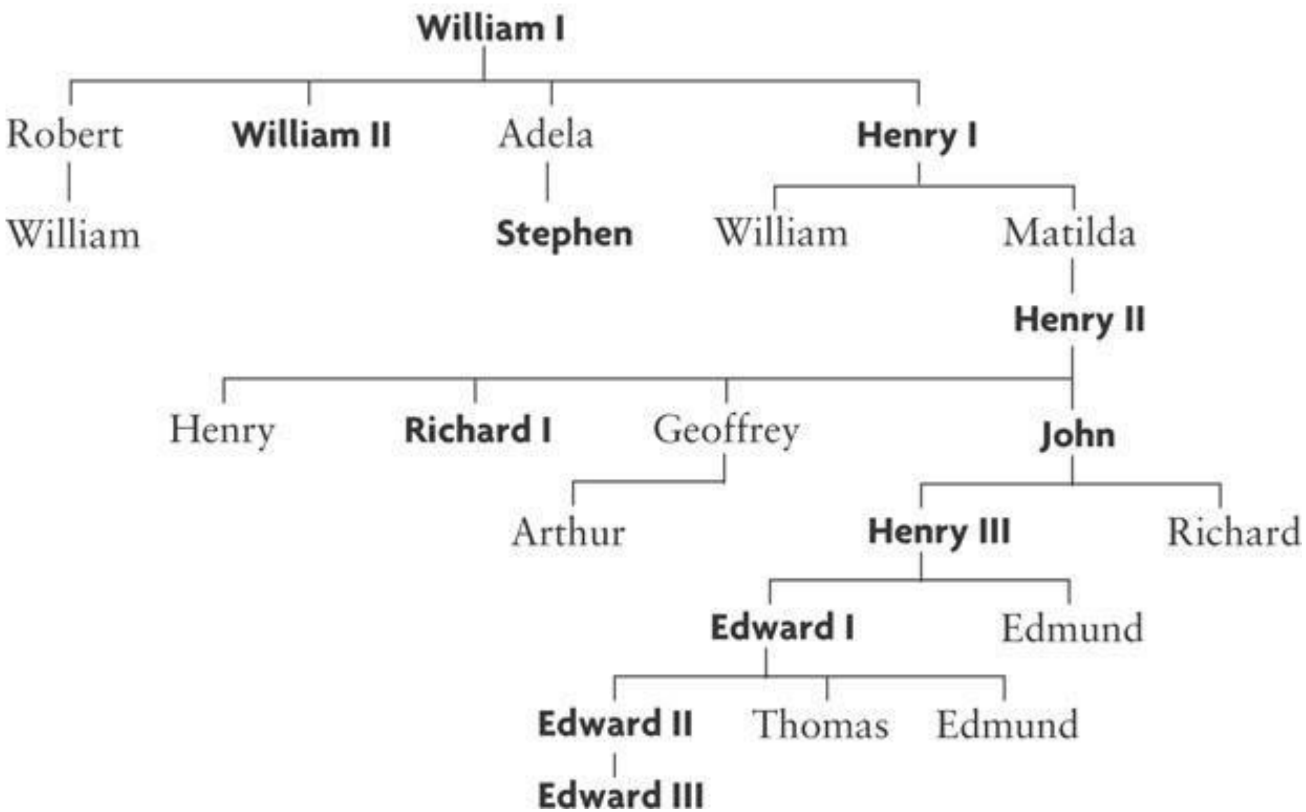




Check basic implementation `w316.GeneralLinkedTree`

General Trees

- We do not discuss general trees implementation in this chapter, but nodes of a general tree can have any number of subtrees



Tree Traversal

Traversal methods:

- Preorder
- Postorder
- Inorder

Recursive definition

Preorder: <root><left><right>

- visit the root
- traverse in preorder the children (subtrees)

Postorder: <left><right><root>

- traverse in postorder the children (subtrees)
- visit the root

Inorder: <left><root><right>

- A node is visited after its left subtree and before its right subtree

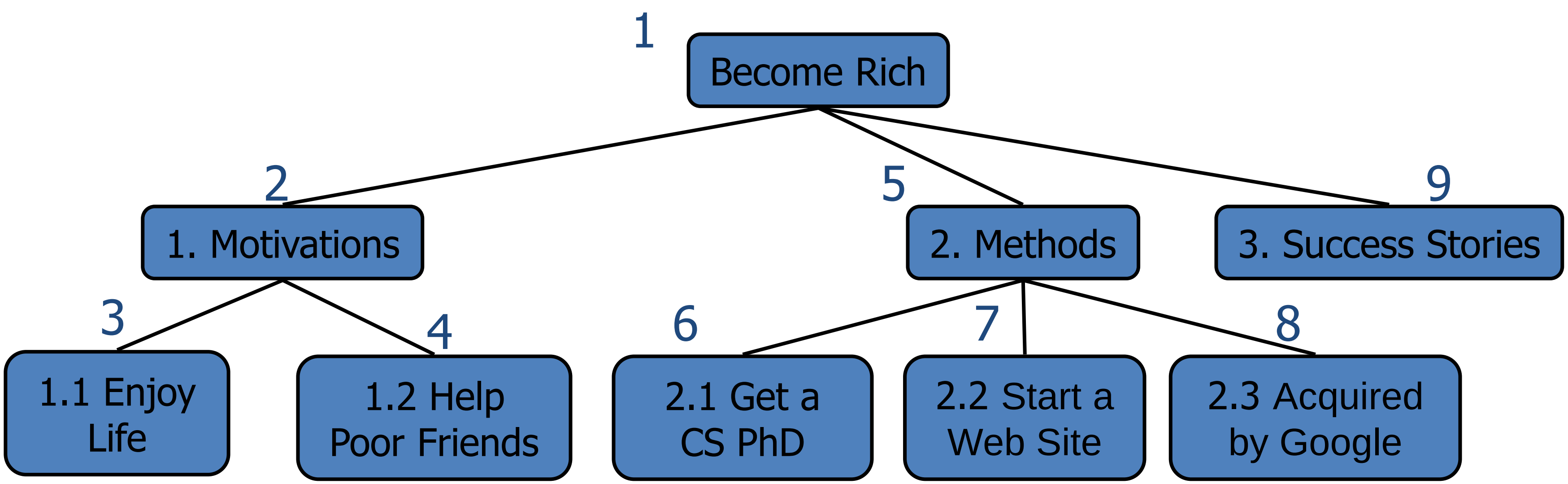
Preorder Traversal

A traversal visits the nodes of a tree in a systematic manner

In a preorder traversal, a node is visited before its descendants

Application: print a structured document

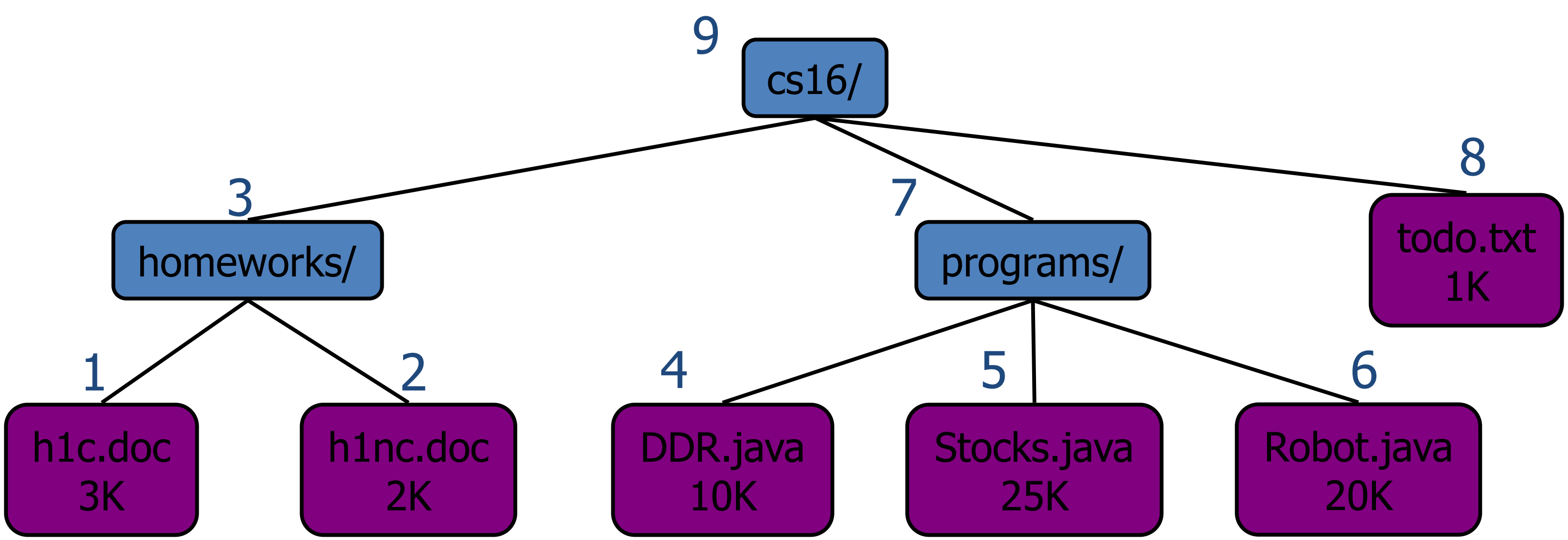
Algorithm ***preOrder***(v)
    visit(v)
    for each child w of v
        preorder (w)



Postorder Traversal

In a postorder traversal, a node is visited after its descendants
Application: compute space used by files in a directory and its subdirectories

Algorithm ***postOrder***(*v*)
  for each child *w* of *v*
    postOrder (*w*)
  visit(*v*)



Inorder Traversal (Binary)

In an inorder traversal a node is visited after its left subtree and before its right subtree

Algorithm ***inOrder*(*v*)**

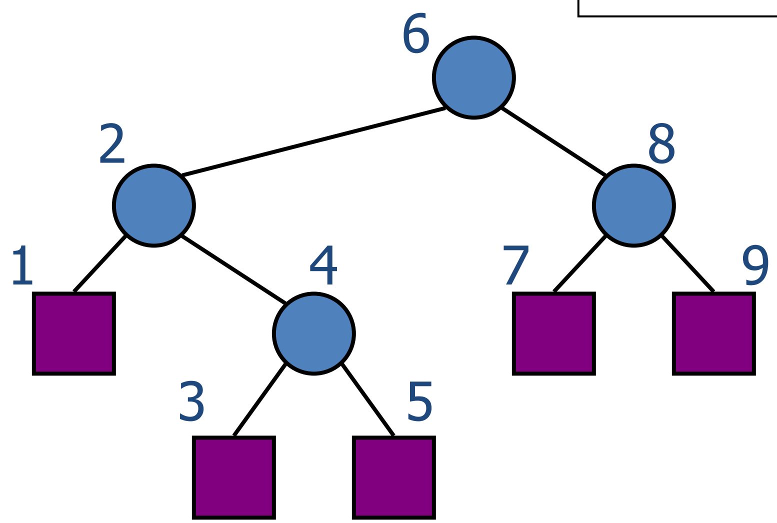
    if ***isInternal*** **(*v*)**

        inOrder **(*leftChild*** **(*v*))**

    ***visit*(*v*)**

    if ***isInternal*** **(*v*)**

        inOrder **(*rightChild*** **(*v*))**



Binary Tree ADT

Section 6.2

Binary Tree ADT

The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT

Additional methods:

- position leftChild(p)
- position rightChild(p)
- position sibling(p)

Update methods may be defined by data structures implementing the BinaryTree ADT

Binary Tree

A binary tree is a tree with the following properties:

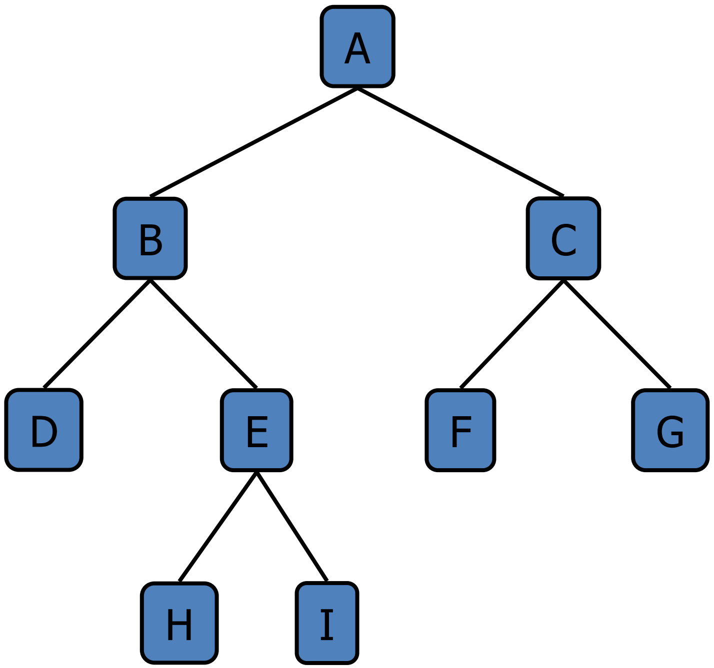
- Each internal node has at most two children (degree of two)
- The children of a node are an ordered pair

We call the children of an internal node left child and right child

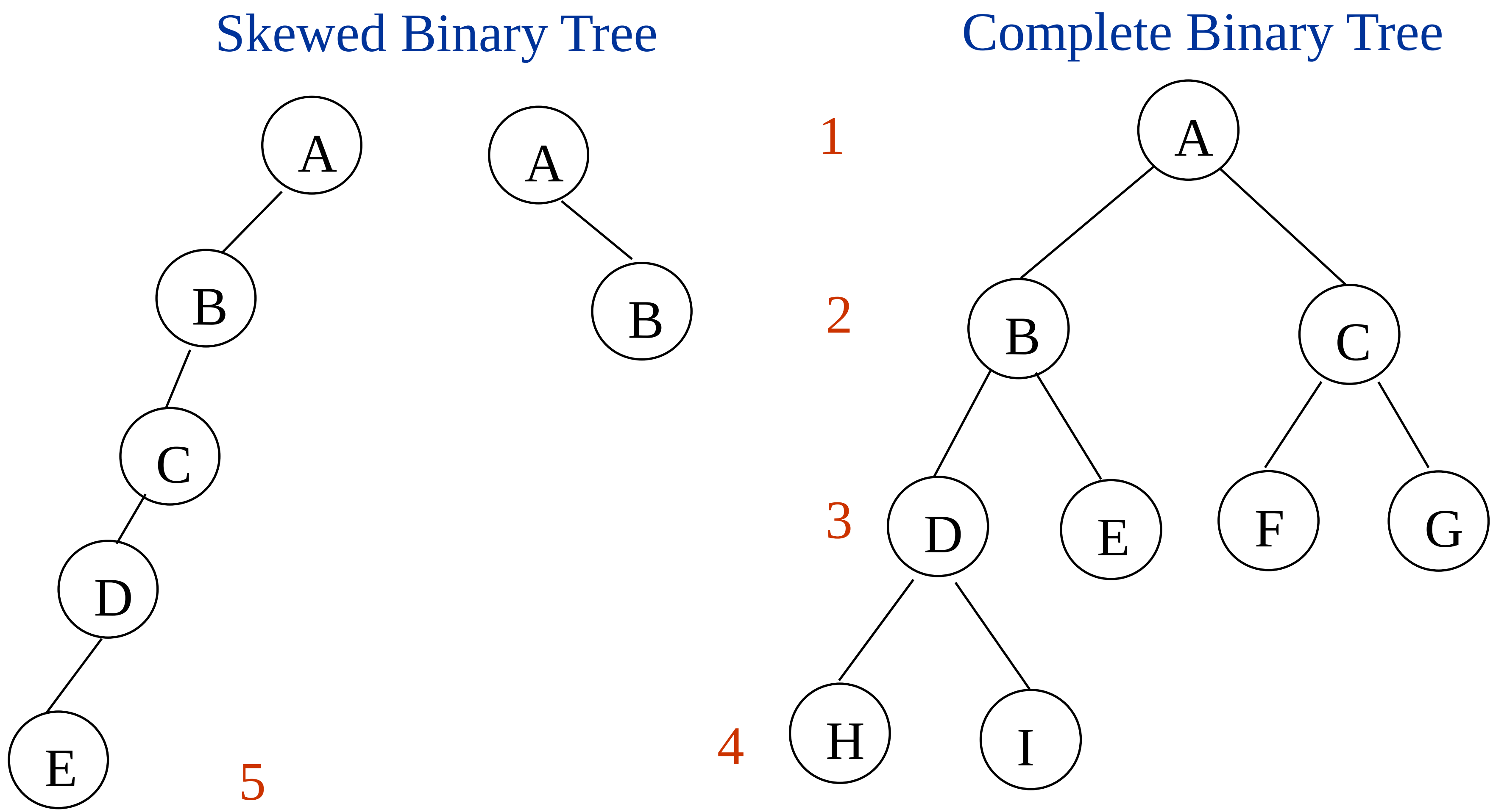
Alternative recursive definition: a binary tree is either

- a tree consisting of a single node, OR
- a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



Examples of the Binary Tree



Differences Between A Tree and A Binary Tree

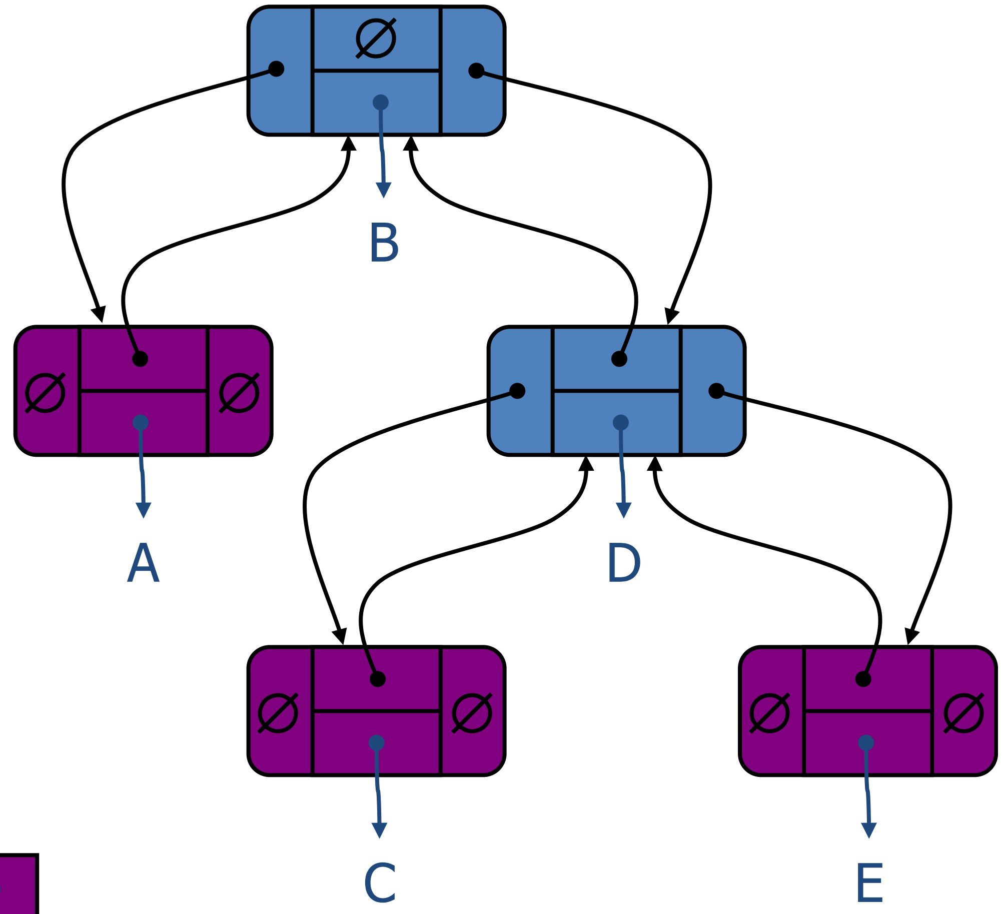
The subtrees of a binary tree are ordered; those of a tree are not ordered.

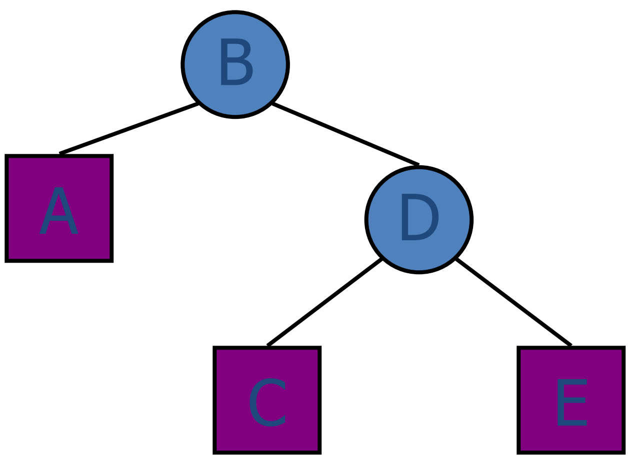
- Are different when viewed as binary trees.
- Are the same when viewed as trees.

Data Structure for Binary Trees

A node is represented by an object storing

- Element
- Parent node
- Left child node
- Right child node





Arithmetic Expression Tree

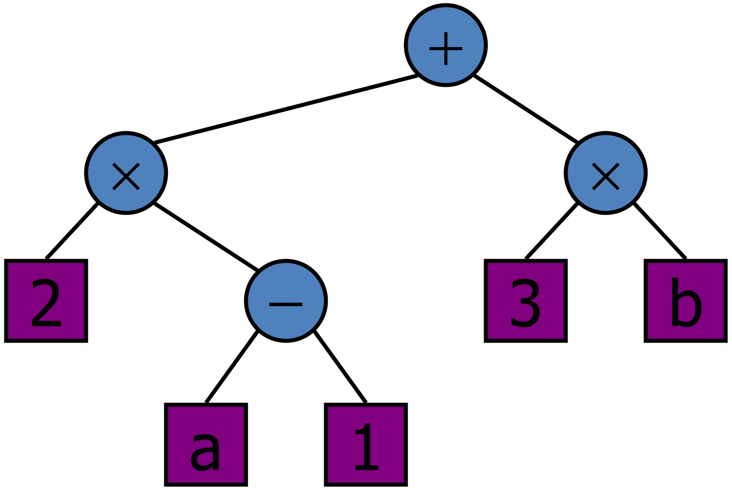
Binary tree associated with an arithmetic expression

internal nodes: operators

external nodes: operands

Example: arithmetic expression tree for the expression

$(2 \times (a - 1) + (3 \times b))$

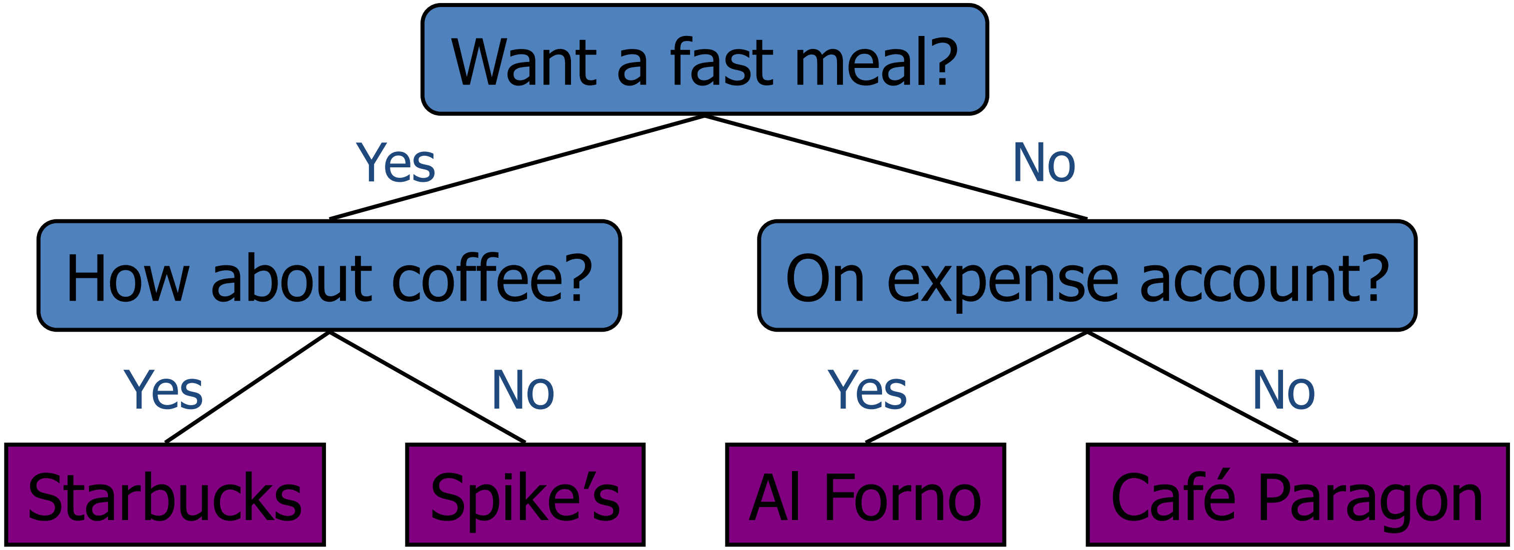


Decision Tree

Binary tree associated with a decision process

- internal nodes: questions with yes/no answer
- external nodes: decisions

Example: dining decision



Maximum Number of Nodes

Maximum number of nodes in a Binary Tree

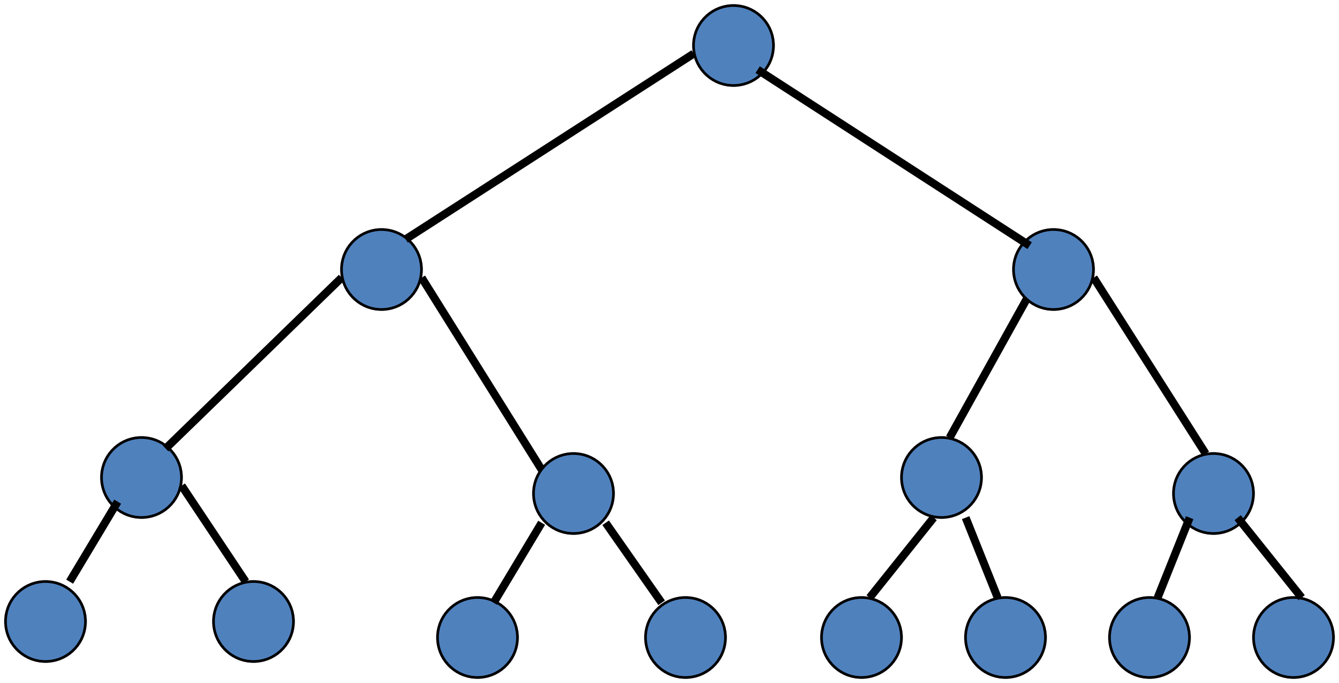
- The maximum number of nodes on depth i of a binary tree is 2i, i>=0.

- The maximum number of nodes in a binary tree of height k is 2k+1-1, k>=0.

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

Full Binary Tree

A full binary tree of a given height k has $2^{k+1}-1$ nodes.



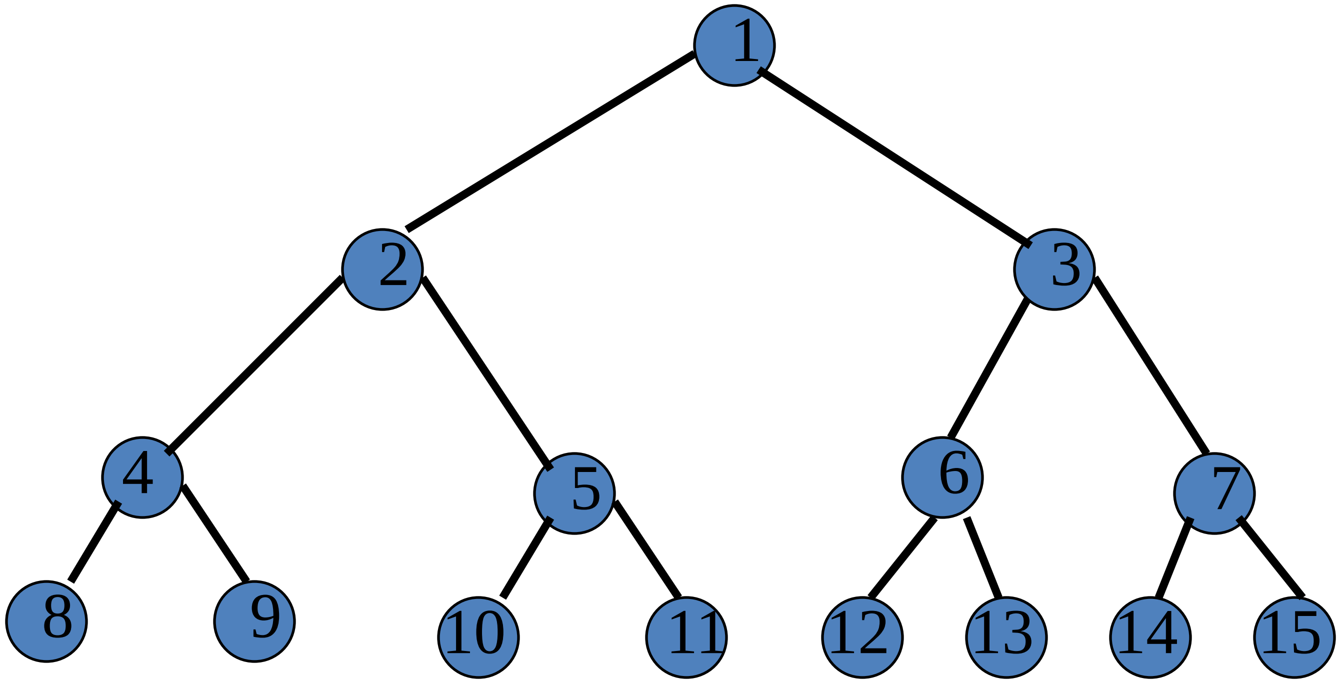
Height 3 full binary tree.

Labeling Nodes In A Full Binary Tree

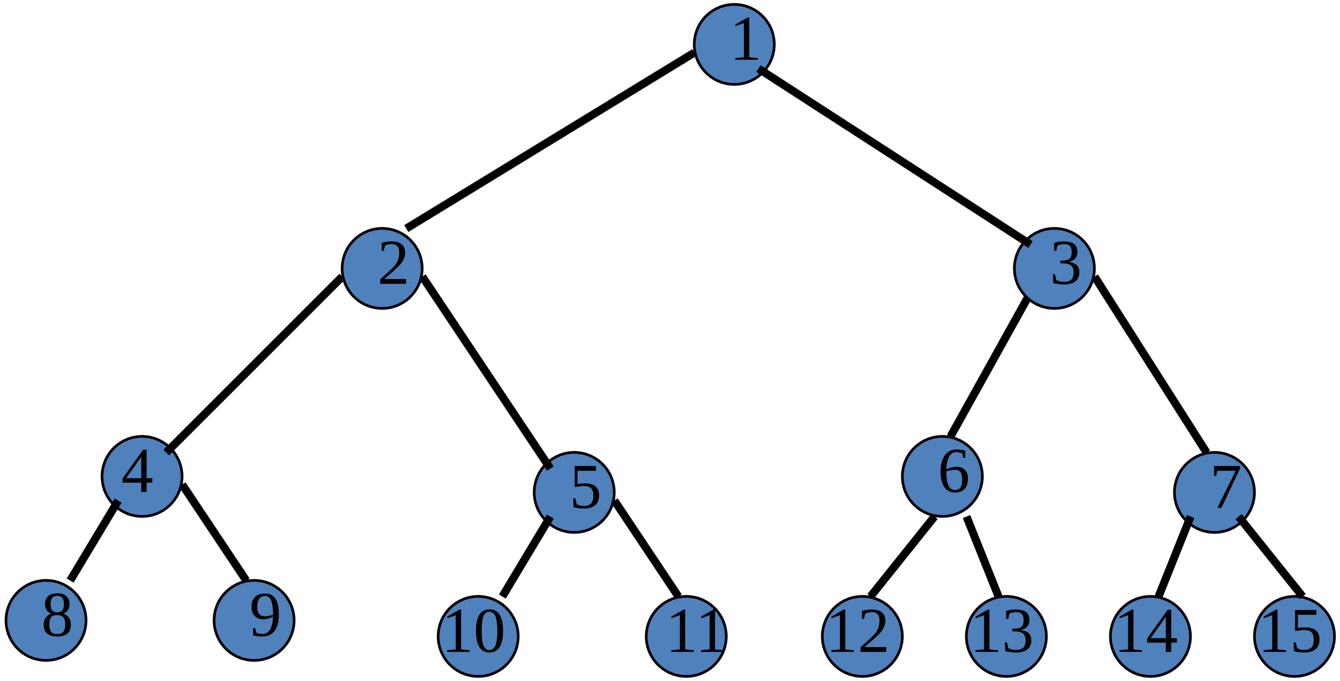
Label the nodes 1 through $2^{k+1} – 1$.

Label by levels from top to bottom.

Within a level, label from left to right.



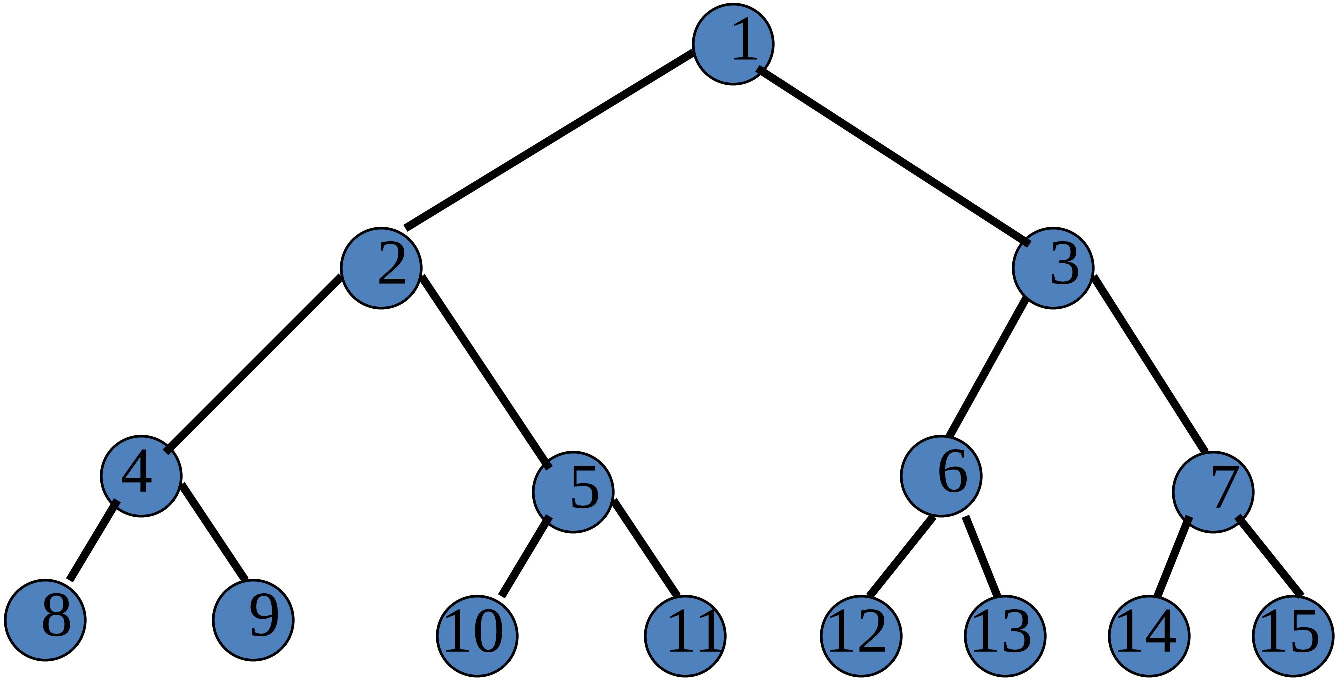
Node Number Properties



Parent of node i is node i / 2, unless i = 1.

Node 1 is the root and has no parent.

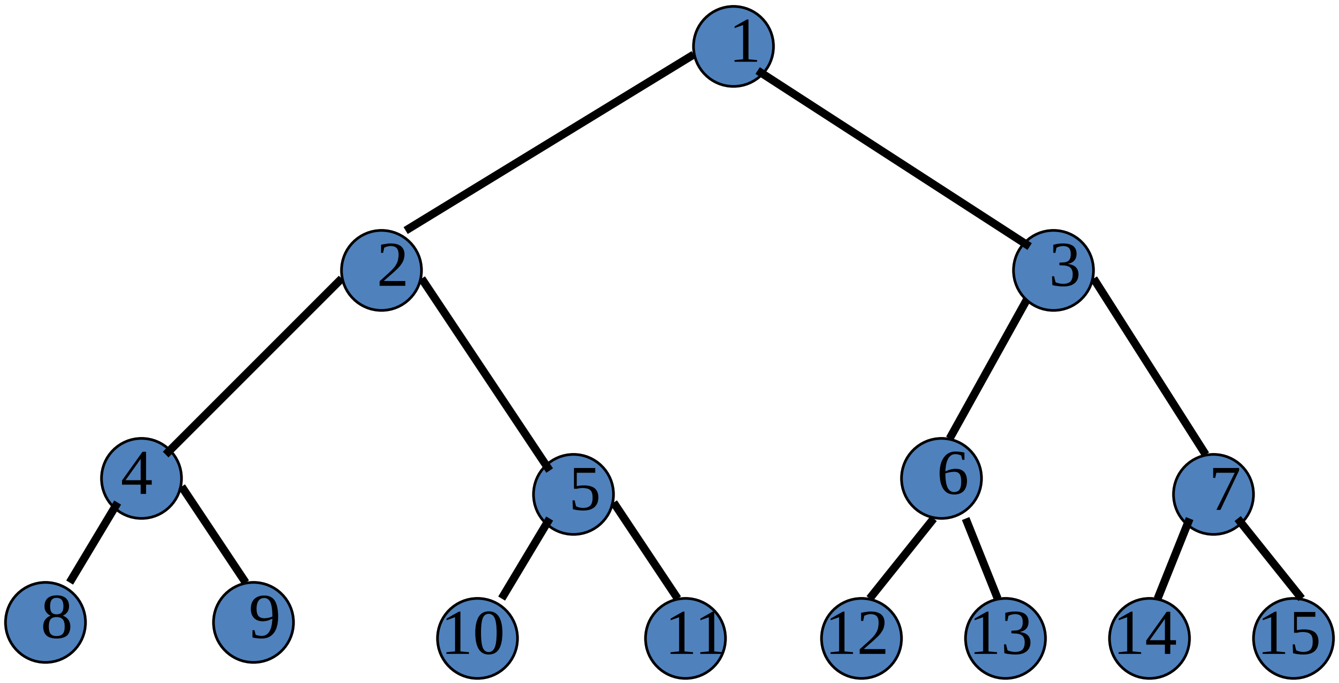
Node Number Properties



Left child of node i is node 2i, unless 2i > n, where n is the number of nodes.

If 2i > n, node i has no left child.

Node Number Properties

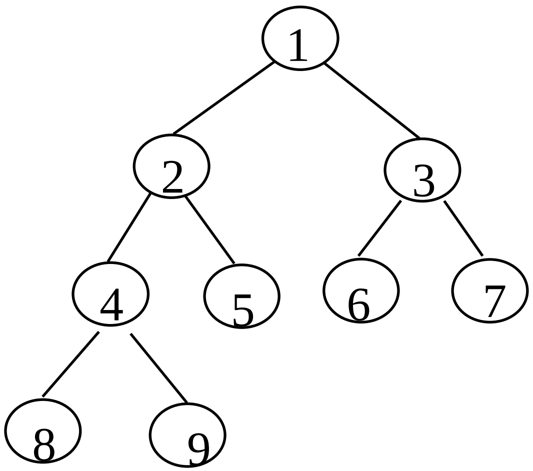


Right child of node i is node 2i+1, unless 2i+1 > n, where n is the number of nodes.

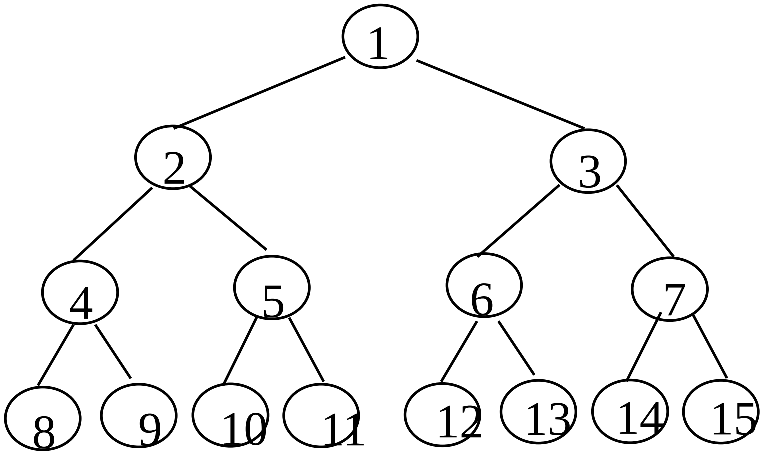
If 2i+1 > n, node i has no right child.

Complete Binary Trees

- A labeled binary tree containing the labels 1 to n with root 1, branches leading to nodes labeled 2 and 3, branches from these leading to 4, 5 and 6, 7, respectively, and so on.
- A binary tree with n nodes and level k is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of level k.



Complete binary tree



Full binary tree of depth 3

Binary Tree Traversals

- Let l, R, and r stand for moving left, visiting the node, and moving right.

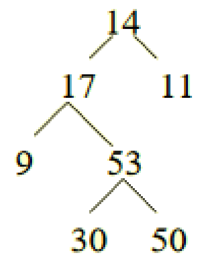
- There are six possible combinations of traversal
lRr, lrR, Rlr, Rrl, rRl, rlR

- Adopt convention that we traverse left before right, only 3 traversals remain
- lRr, lrR, Rlr
- **inorder, postorder, preorder**

Binary Tree Traversals - Example

Examples:

Display the nodes of the tree:

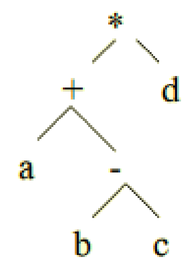


Pre-order: 14, 17, 9, 53, 30, 50, 11;

In-order: 9, 17, 30, 53, 50, 14, 11;

Post-order: 9, 30, 50, 53, 17, 11, 14;

Tree representation of the algebraic expression: (a+(b-c))*d.
Display the nodes of the tree:



Pre-order: * + a – b c d .
This gives the prefix notation of expressions;

In-order: a + b – c * d
This requires bracketing for representing sub-expressions.
It is called infix notation of expressions;

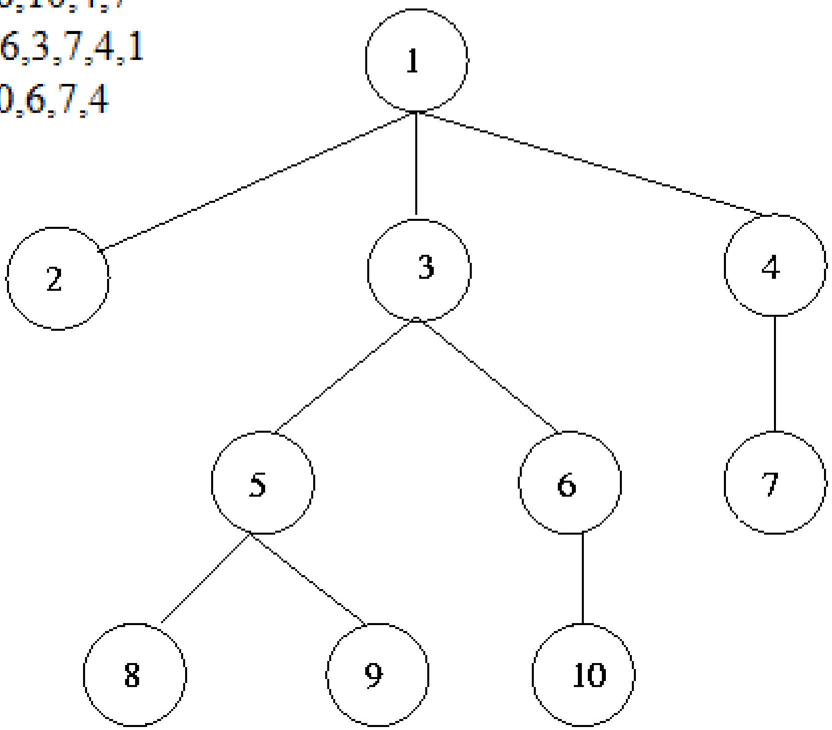
Post-order: a b c - + d *
This gives the postfix notation of expressions.

Exercise

What is the order of the following for the given tree

i. **Pre-order**

Preorder 1,2,3,5,8,9,6,10,4,7
Postorder 2,8,9,5,10,6,3,7,4,1
Inorder 2,1,8,5,9,3,10,6,7,4

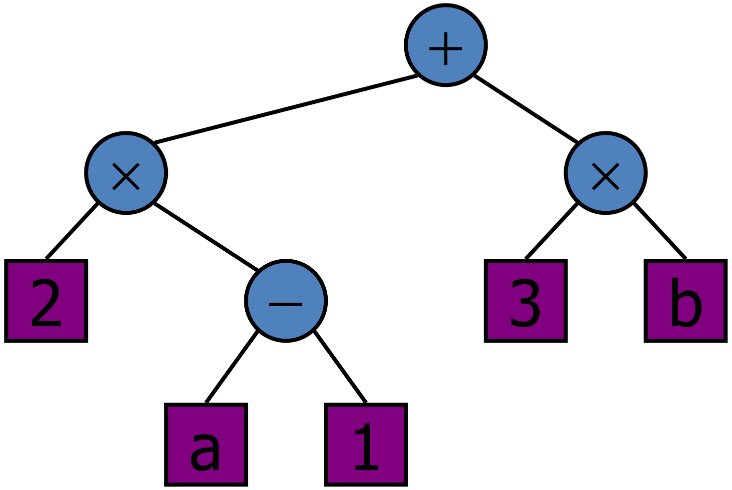


Print Arithmetic Expressions

Specialization of an inorder traversal

- print operand or operator when visiting node
- print “(“ before traversing left subtree
- print “)“ after traversing right subtree

Algorithm *inOrder* (*v*)
 if *isInternal* (*v*){
  print(“(’’)
  inOrder (*leftChild* (*v*))}
 print(*v.element* ())
 if *isInternal* (*v*){
  inOrder (*rightChild* (*v*))
  print (“)’’)}



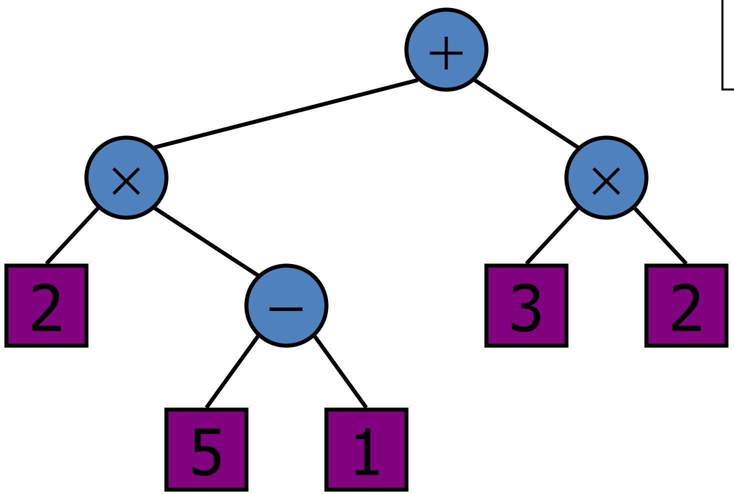
$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expressions

recursive method returning the value of a subtree

when visiting an internal node, combine the values of the subtrees

Algorithm *evalExpr*(v)
  if *isExternal* (v)
    return *v.element* ()
  else
    $x \leftarrow$ *evalExpr*(*leftChild* (v))
    $y \leftarrow$ *evalExpr*(*rightChild* (v))
    $\Diamond \leftarrow$ operator stored at v
    return $x \Diamond y$

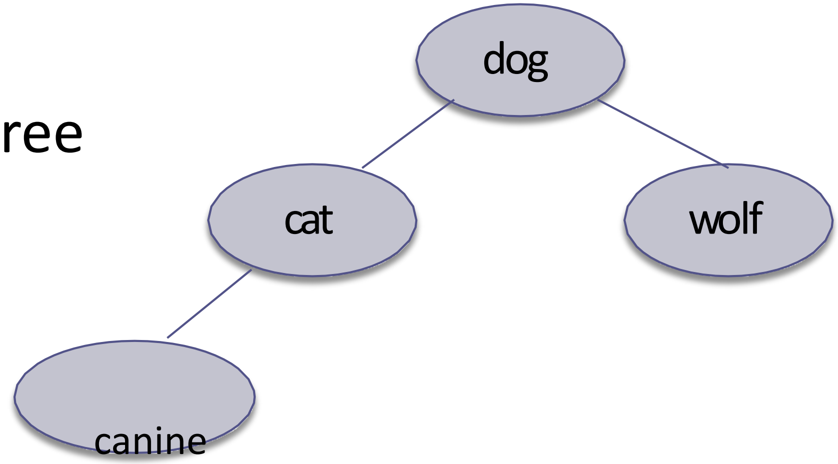


Binary Search Trees

Section 6.3

Binary Search Tree

- Binary search trees
  - All elements in the left subtree precede those in the right subtree
- A formal definition:



A set of nodes T is a binary search tree if either of the following is true:

- T is empty
- If T is not empty, its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary search trees and the value in the root node of T is greater than all values in T_L and is less than all values in T_R

Binary Search Tree (cont.)

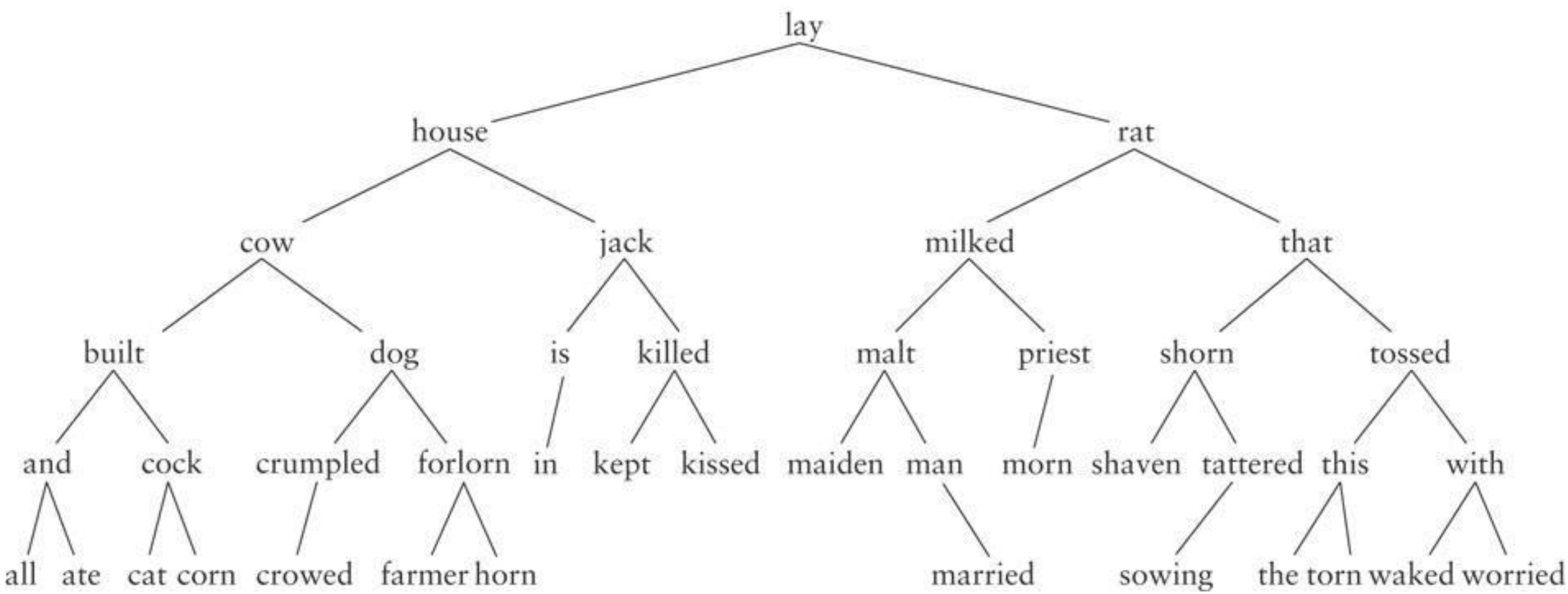
- A binary search tree never has to be sorted because its elements always satisfy the required order relationships
- When new elements are inserted (or removed) properly, the binary search tree maintains its order
- In contrast, a sorted array must be expanded whenever new elements are added, and compacted whenever elements are removed—expanding and contracting are both O(*n*)

Binary Search Tree (cont.)

- When searching a BST, each probe has the potential to eliminate half the elements in the tree, so searching can be O(log *n*)
- In the worst case, searching is O(log *n*)

- Practice BST animated site : https://visualgo.net/en/bst

Overview of a Binary Search Tree (cont.)



Recursive Algorithm for Searching a Binary Search Tree

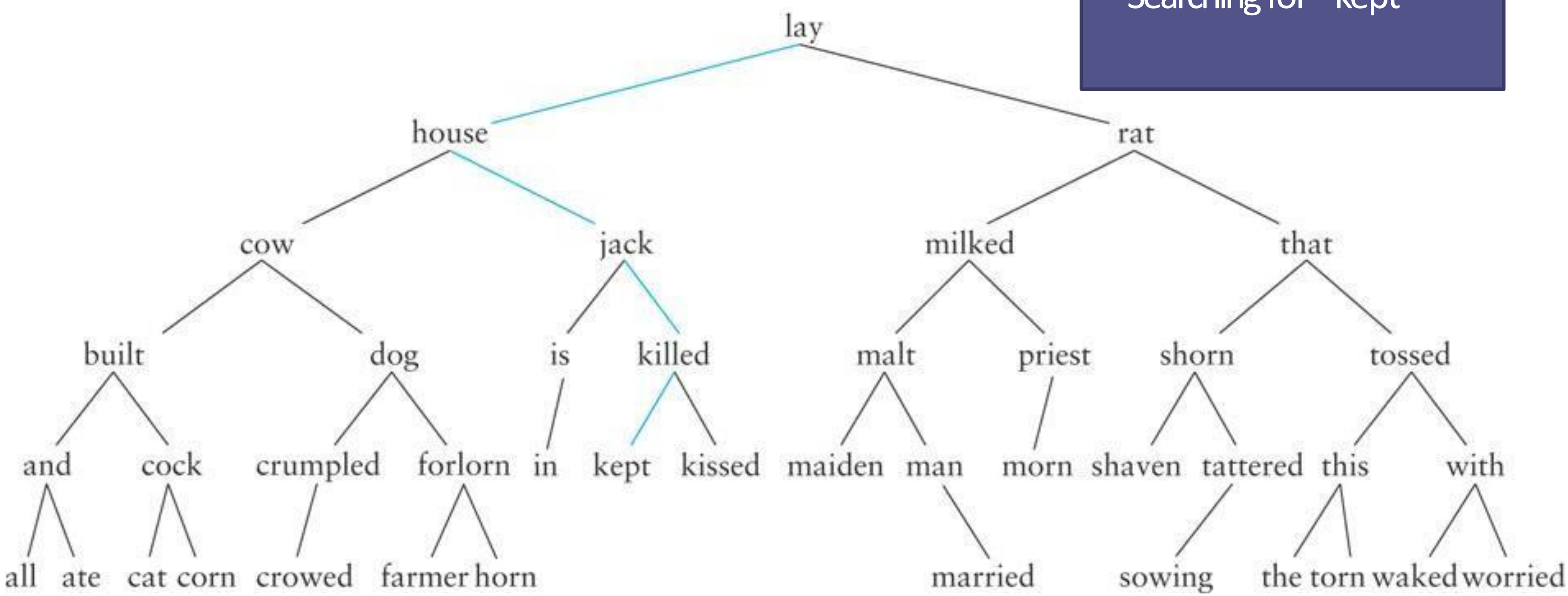
1. **`if`** the root is **`null`**
2. the item is not in the tree; return **`null`**
3. Compare the value of **`target`** with **`root.data`**
4. **`if`** they are equal
5. the target has been found; return the data at the root

   `else if` the target is less than **`root.data`**
6. return the result of searching the left subtree

   `else`
7. return the result of searching the right subtree

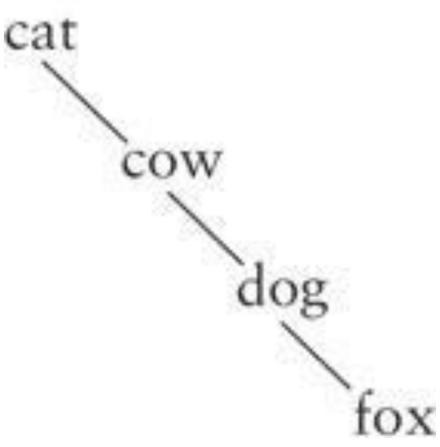
Searching a Binary Tree

Searching for "kept"



Performance

- Search a tree is generally O(log *n*)
- If a tree is not very full, performance will be worse
- Searching a tree with only right subtrees, for example, is O(*n*)



Using BSTs For Sorting

- The following is an algorithm for sorting a list of Integers:
  - Insert them into a BST
  - Do an inorder traversal of the BST to get the sorted list.)

BST Implementation

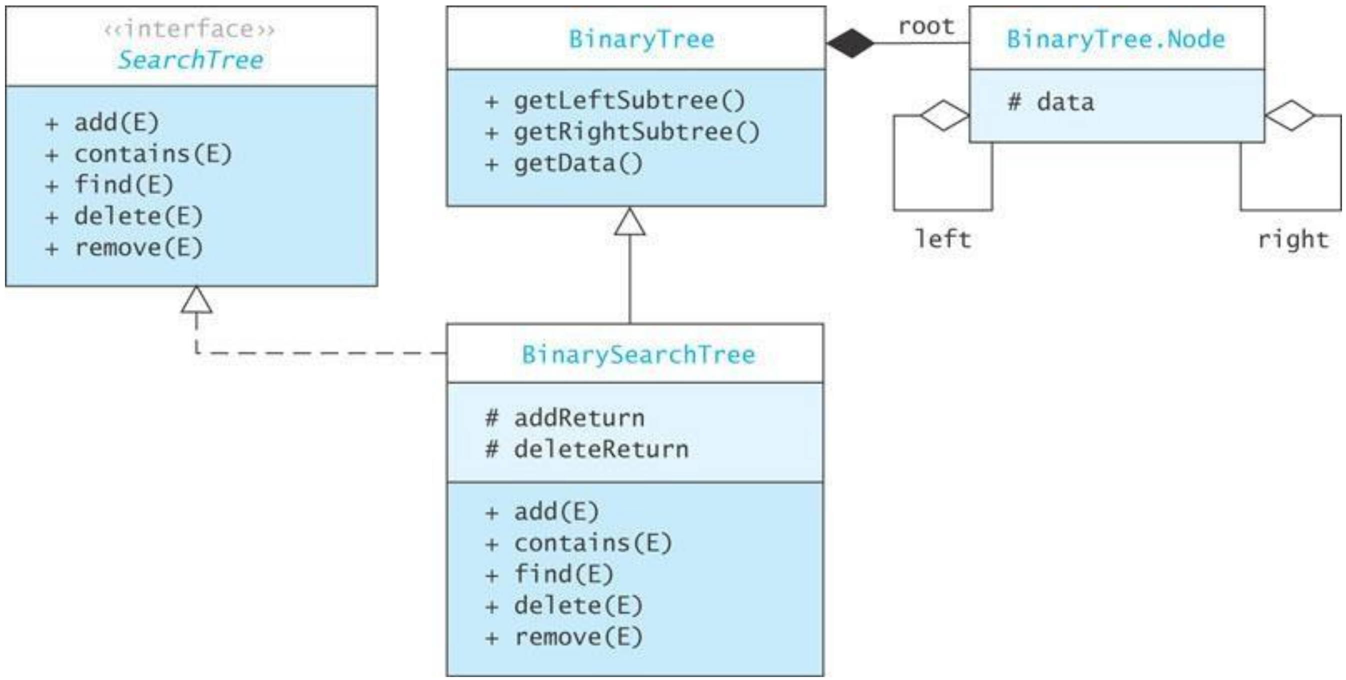
Section 6.4

Interface SearchTree<E>

Method	Behavior
`boolean add(E item)`	Inserts `item` where it belongs in the tree. Returns **`true`** if item is inserted; **`false`** if it isn't (already in tree).
`boolean contains(E target)`	Returns **`true`** if `target` is found in the tree.
`E find(E target)`	Returns a reference to the data in the node that is equal to `target`. If no such node is found, returns **`null`**.
`E delete(E target)`	Removes `target` (if found) from tree and returns it; otherwise, returns **`null`**.
`boolean remove(E target)`	Removes `target` (if found) from tree and returns **`true`**; otherwise, returns **`false`**.

BinarySearchTree<E> Class

Data Field	Attribute
`protected boolean addReturn`	Stores a second return value from the recursive `add` method that indicates whether the item has been inserted.
`protected E deleteReturn`	Stores a second return value from the recursive `delete` method that references the item that was stored in the tree.



Implementing `find` Methods

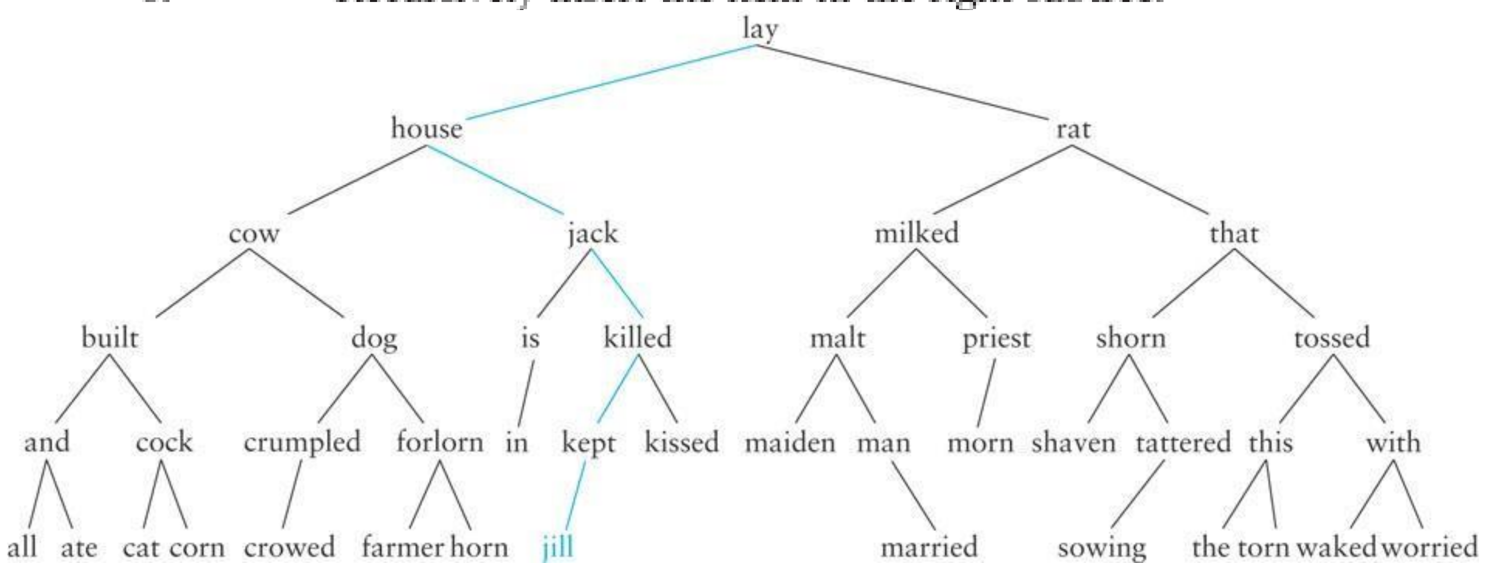
BinarySearchTree `find` Method

```
/** Starter method find.
    pre: The target object must implement
         the Comparable interface.
    @param target The Comparable object being sought
    @return The object, if found, otherwise null
*/
public E find(E target) {
    return find(root, target);
}

/** Recursive find method.
    @param localRoot The local subtree's root
    @param target The object being sought
    @return The object, if found, otherwise null
*/
private E find(Node<E> localRoot, E target) {
    if (localRoot == null)
        return null;

    // Compare the target with the data field at the root.
    int compResult = target.compareTo(localRoot.data);
    if (compResult == 0)
        return localRoot.data;
    else if (compResult < 0)
        return find(localRoot.left, target);
    else
        return find(localRoot.right, target);
}
```

Insertion into a Binary Search Tree

Recursive Algorithm for Insertion in a Binary Search Tree

1. `if` the root is `null`
2. Replace empty tree with a new tree with the item at the root and return `true`.
3. `else if` the item is equal to `root.data`
4. The item is already in the tree; return `false`.
5. `else if` the item is less than `root.data`
6. Recursively insert the item in the left subtree.
7. `else`
8. Recursively insert the item in the right subtree.



Implementing the `add` Methods

```
/** Starter method add.
    pre: The object to insert must implement the
         Comparable interface.
    @param item The object being inserted
    @return true if the object is inserted, false
            if the object already exists in the tree
*/
public boolean add(E item) {
    root = add(root, item);
    return addReturn;
}
```

Implementing the `add` Methods (cont.)

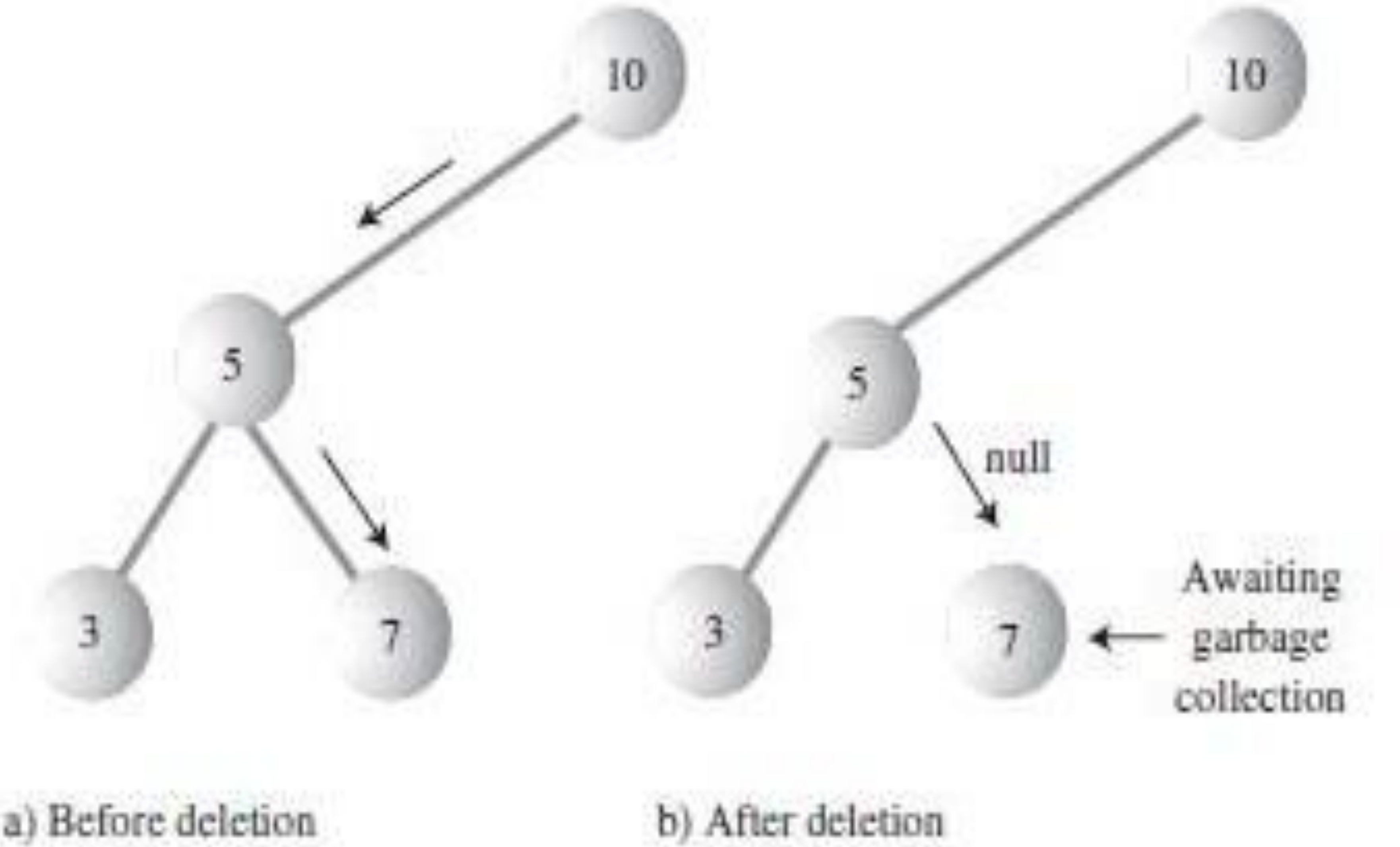
```
/** Recursive add method.
    post: The data field addReturn is set true if the item is added to
          the tree, false if the item is already in the tree.
    @param localRoot The local root of the subtree
    @param item The object to be inserted
    @return The new local root that now contains the
          inserted item
*/
private Node<E> add(Node<E> localRoot, E item) {
    if (localRoot == null) {
          // item is not in the tree — insert it.
          addReturn = true;
          return new Node<E>(item);
    } else if (item.compareTo(localRoot.data) == 0) {
          // item is equal to localRoot.data
          addReturn = false;
          return localRoot;
    } else if (item.compareTo(localRoot.data) < 0) {
          // item is less than localRoot.data
          localRoot.left = add(localRoot.left, item);
          return localRoot;
    } else {
          // item is greater than localRoot.data
          localRoot.right = add(localRoot.right, item);
          return localRoot;
    }
}
```

Deleting a Node

- Start by finding the node you want to delete.
- Then there are three cases to consider:
  1. The node to be deleted is a leaf
  2. The node to be deleted has one child
  3. The node to be deleted has two children

Deletion cases: Leaf Node(Case-1)

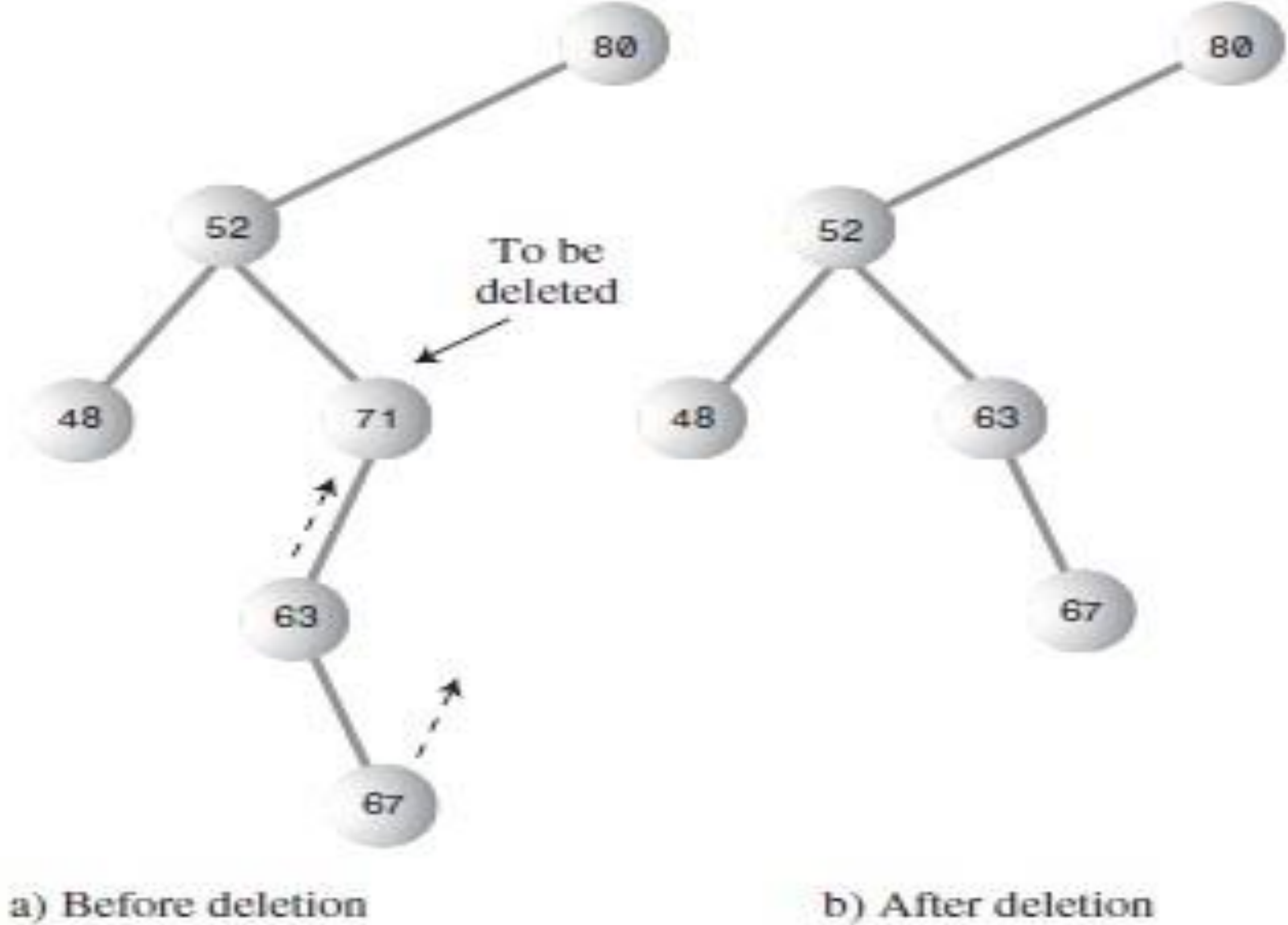
- To delete a leaf node, simply change the appropriate child field in the node's parent to point to *null*, instead of to the node.
- The node still exists, but is no longer a part of the tree.
- Because of Java's garbage collection feature, the node need not be deleted explicitly.



Deleting a node with no children.

Deletion: One Child(Case-2)

- The node to be deleted in this case has only two connections: to its parent and to its only child.
- Connect the child of the node to the node's parent, thus cutting off the connection between the node and its child, and between the node and its parent.

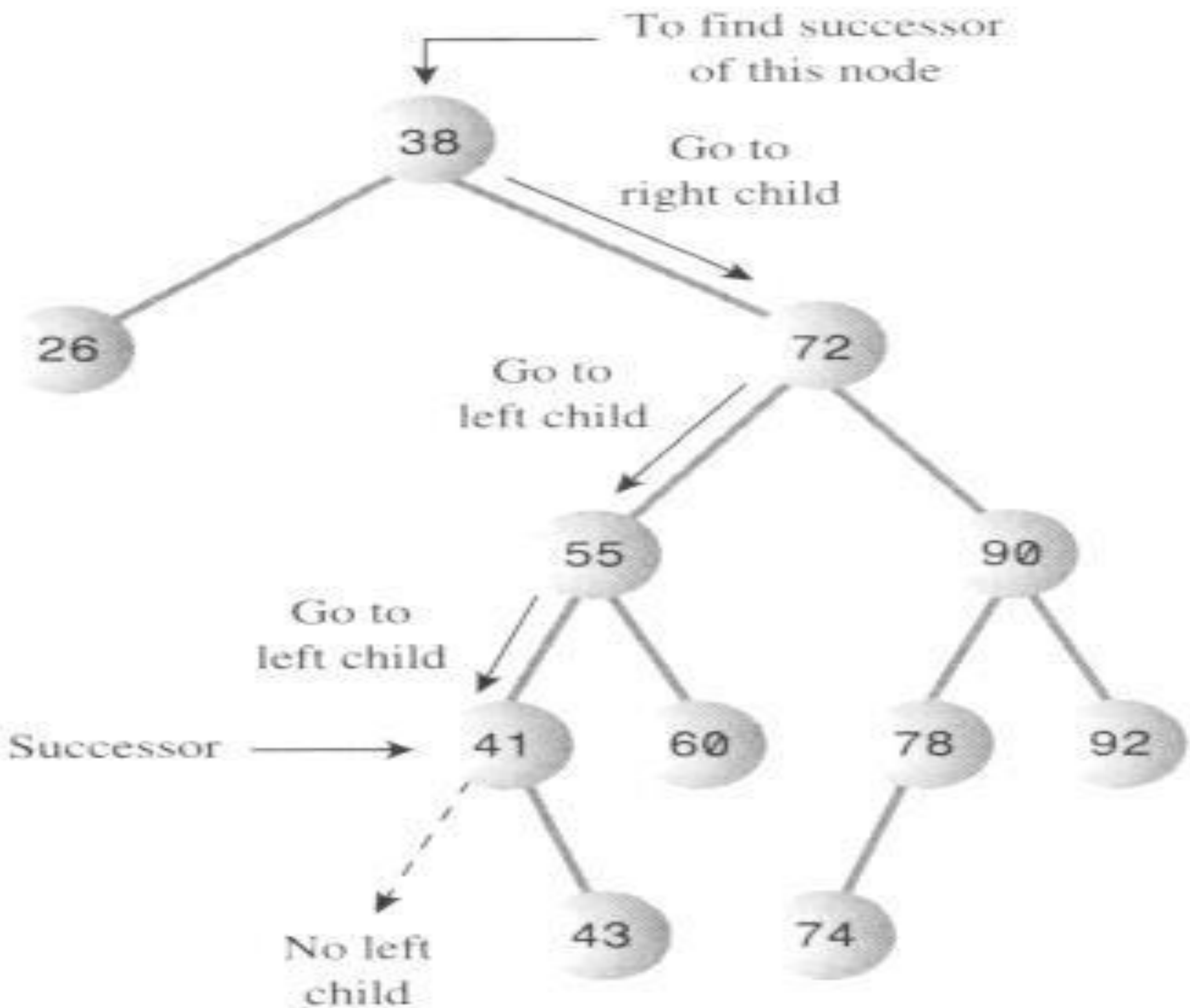


Deleting a node with one child.

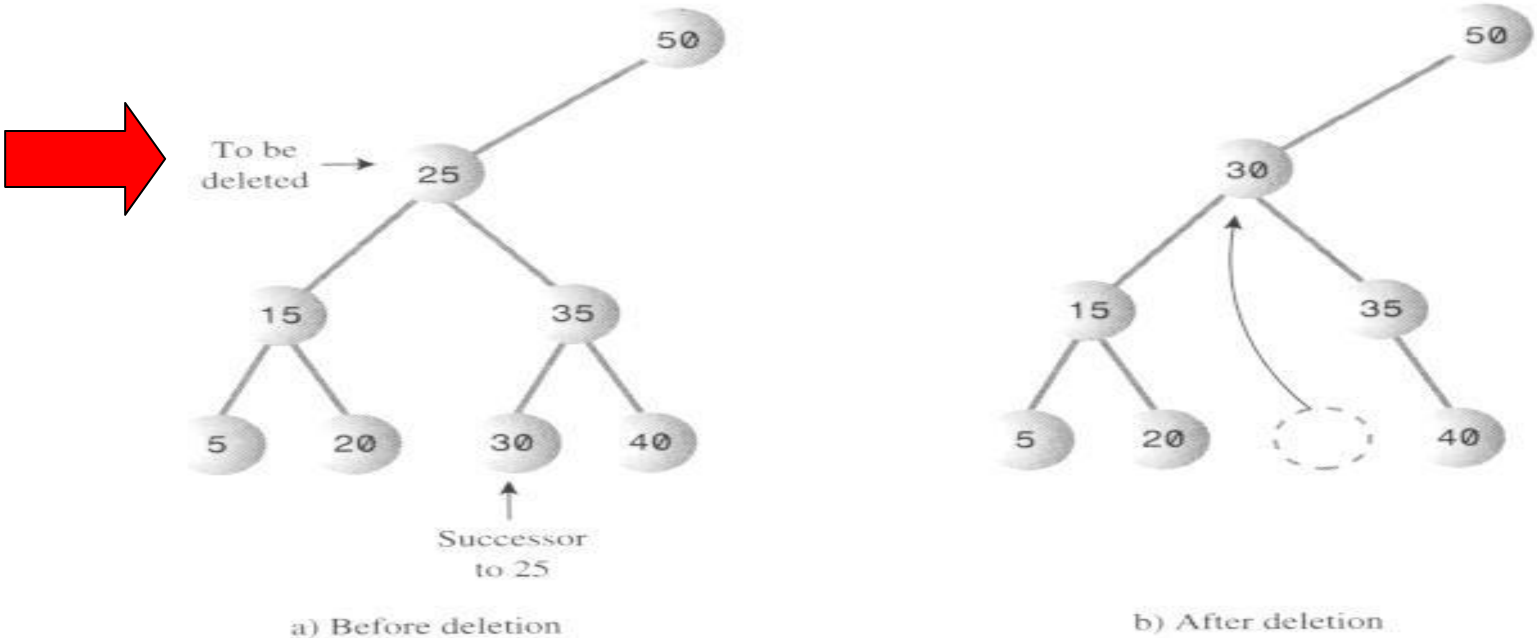
Deletion: Two Children(Case-3)

- To delete a node with two children, replace the node with its
  inorder successor or inorder predessor.
- For each node, the node with the next-highest key (to the deleted node) in the subtree is called its inorder successor.
- To find the successor,
  - start with the original (deleted) node’s right child.
  - Then go to this node’s left child and then to its left child and so on, following down the path of left children.
  - The last left child in this path is the successor of the original node.

Find successor

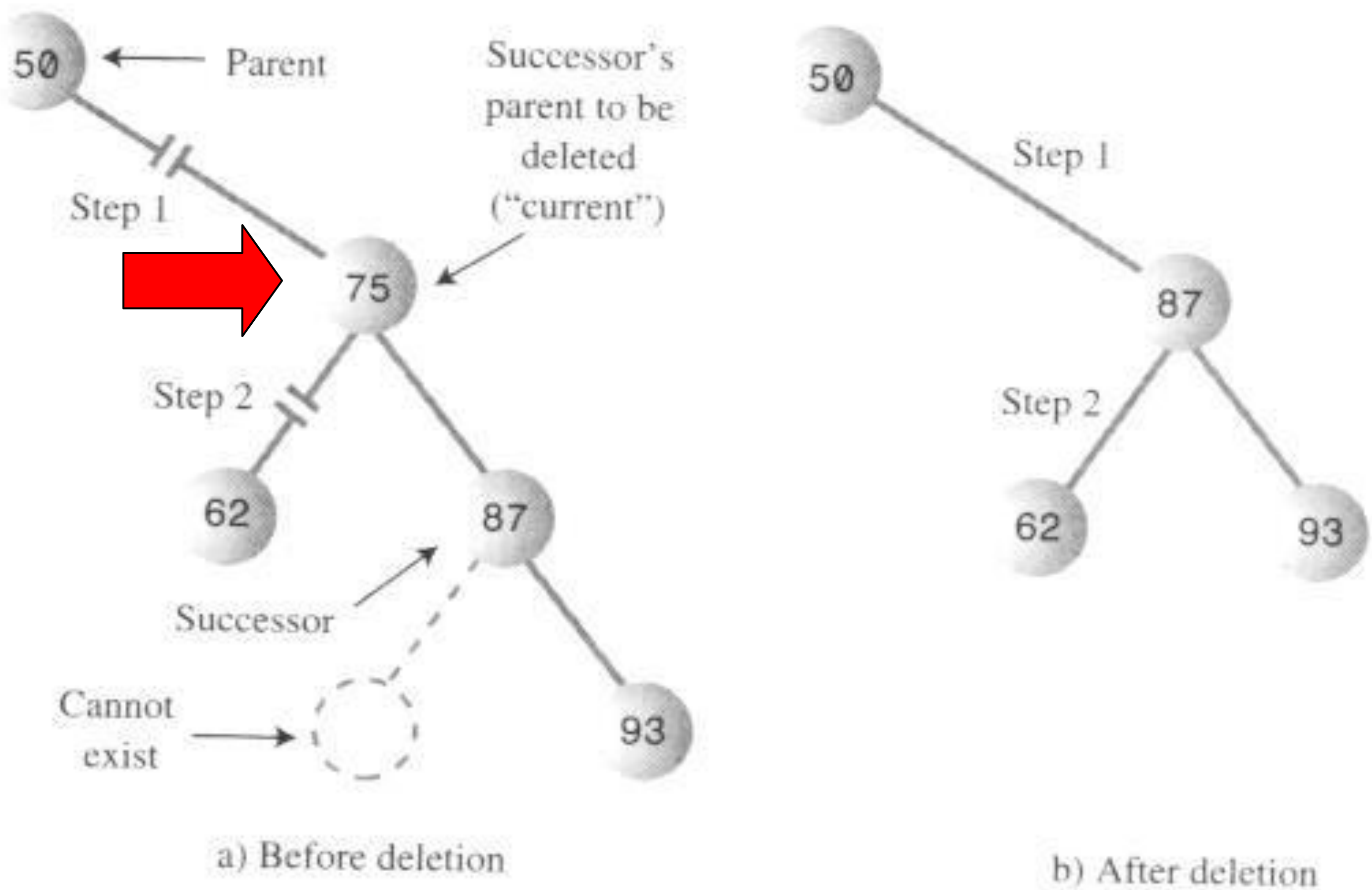


Delete a node with subtree (case 3a) (Successor has no left and right)



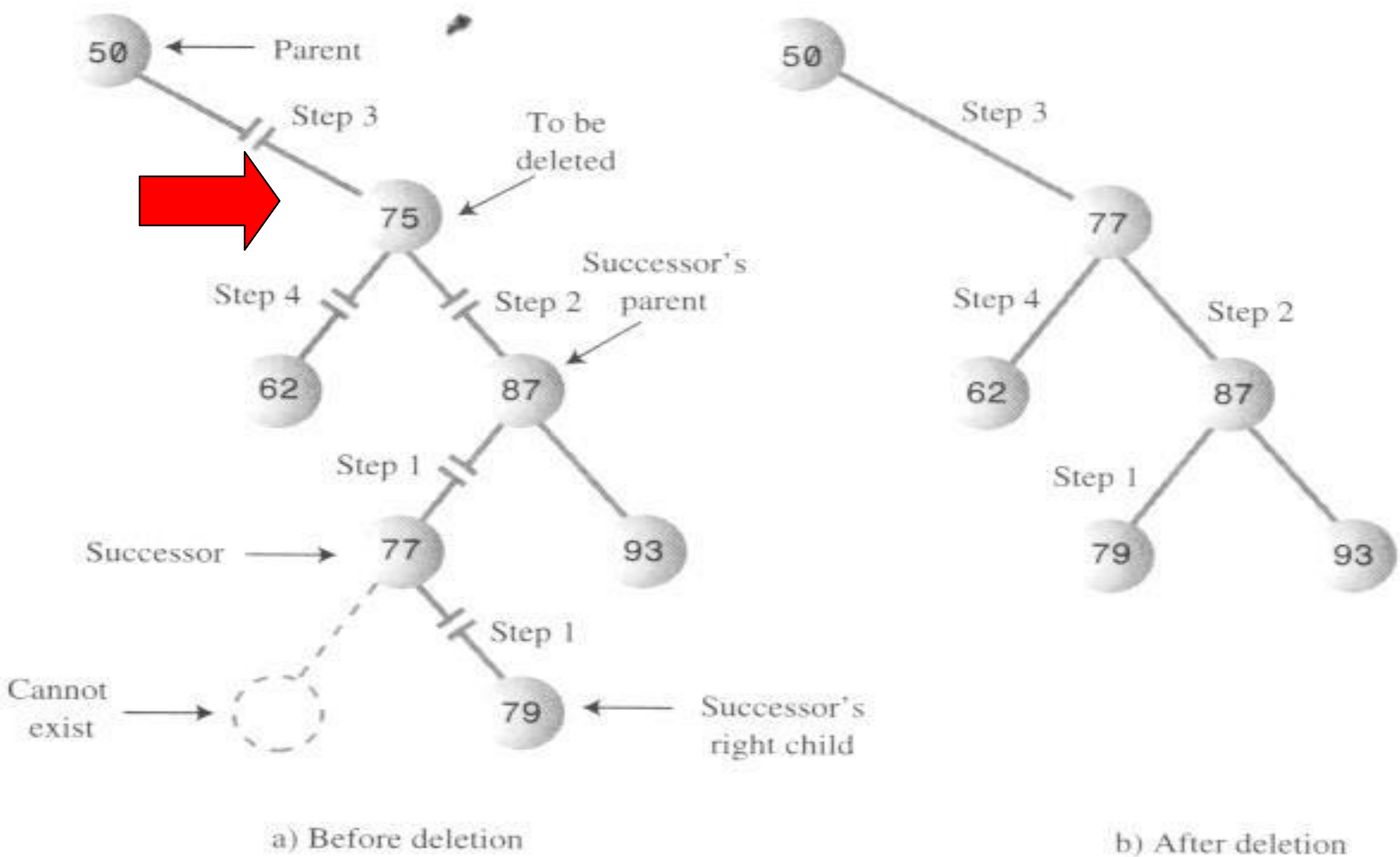

```
if(current == root)
        root = successor;
    else if(isLeftChild)
        parent.leftChild = successor;
    else
        parent.rightChild = successor;
```

Delete a node with subtree (case 3b) Successor is the Right child of delnode



Delete a node with subtree (case 3c)

Successor is left Decendent of right child of delnode



Algorithm for Removing from a Binary Search Tree

Recursive Algorithm for Removal from a Binary Search Tree

```
1.   if the root is null
2.       The item is not in tree – return null.
3.   Compare the item to the data at the local root.
4.   if the item is less than the data at the local root
5.       Return the result of deleting from the left subtree.
6.   else if the item is greater than the local root
7.       Return the result of deleting from the right subtree.
8.   else // The item is in the local root
9.       Store the data in the local root in deletedReturn.
10.      if the local root has no children
11.          Set the parent of the local root to reference null.
12.      else if the local root has one child
13.          Set the parent of the local root to reference that child.
14.      else // Find the inorder predecessor
15.          if the left child has no right child it is the inorder predecessor
16.              Set the parent of the local root to reference the left child.
17.          else
18.              Find the rightmost node in the right subtree of the left
                 child.
19.              Copy its data into the local root's data and remove it by
                 setting its parent to reference its left child.
```

Implementing the `delete` Method

- Listing 6.5 (`BinarySearchTree delete` Methods; pages 325-326)

Method `findLargestChild`

BinarySearchTree findLargestChild Method

```
/** Find the node that is the
    inorder predecessor and replace it
    with its left child (if any).
    post: The inorder predecessor is removed from the tree.
    @param parent The parent of possible inorder
                  predecessor (ip)
    @return The data in the ip
*/
private E findLargestChild(Node<E> parent) {
    // If the right child has no right child, it is
    // the inorder predecessor.
    if (parent.right.right == null) {
        E returnValue = parent.right.data;
        parent.right = parent.right.left;
        return returnValue;
    } else {
        return findLargestChild(parent.right);
    }
}
```

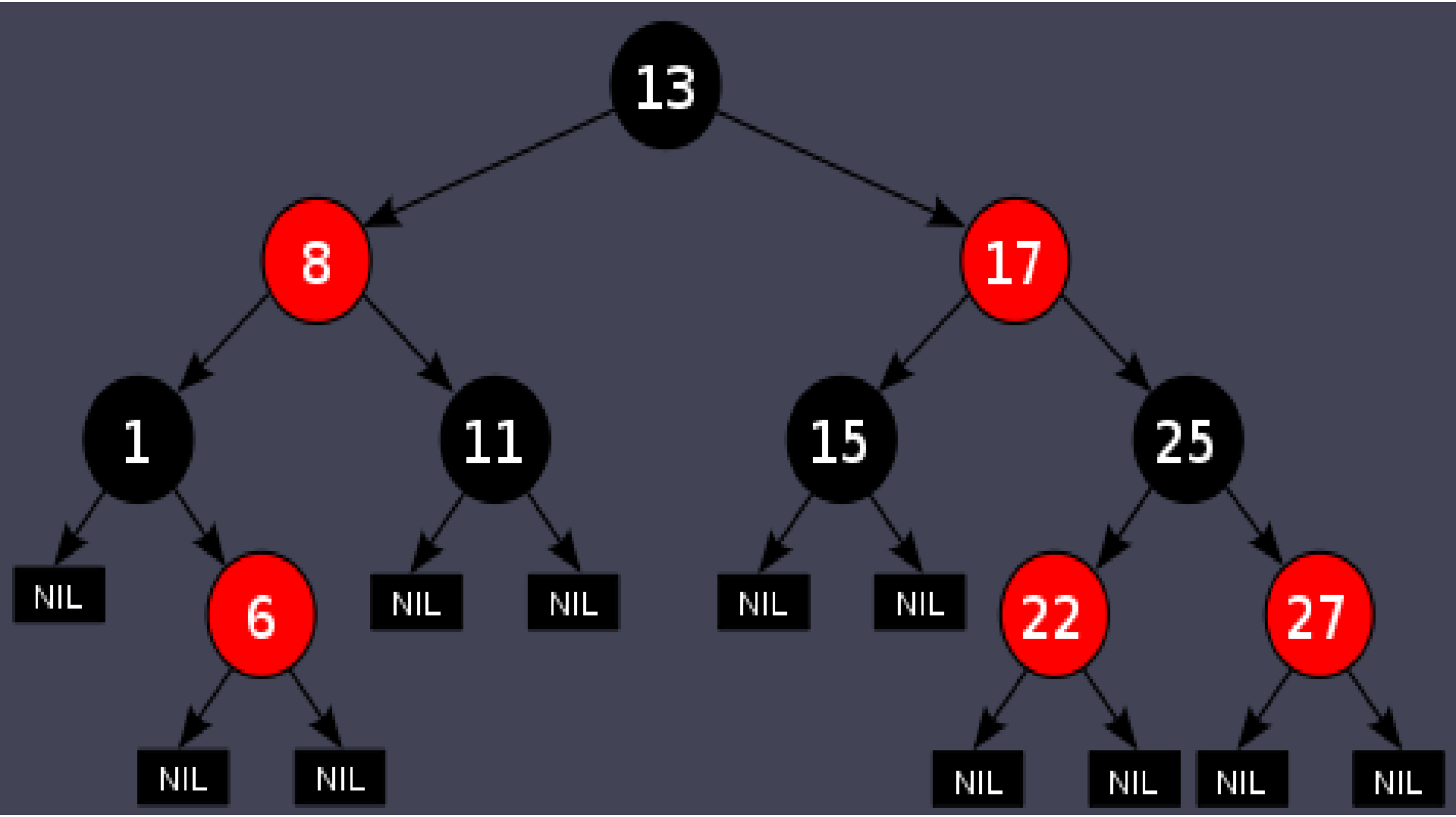
Testing a Binary Search Tree

- To test a binary search tree, verify that an inorder traversal will display the tree contents in ascending order after a series of insertions and deletions are performed

BST from Collection Framework

- If a BST becomes unbalanced, its performance degrades dramatically
- Techniques have been developed to keep a tree from slipping into an unbalanced condition – the most popular such technique uses *red-black trees* (a type of BST, where each node has a color, red or black.)
- Java's *TreeSet* and *TreeMap* classes implement balanced trees using *red-black trees.*

Red Black Tree



TreeSet

- In a **TreeSet**, elements are kept in order
- That means Java must compare elements to decide which is —larger‖ and which is —smaller‖
- Java done this by using the Comparator Interface
- TreeSetTest.java

Tree Map

- *You can create a map using one of its three concrete classes:* **HashMap**, **LinkedHashMap**, *or* **TreeMap**.
- A *map* is a container object that stores a collection of key/value pairs. It enables fast retrieval, deletion, and updating of the pair through the key.
- A map stores the values along with the keys.
- The keys are like indexes. In **List**, the indexes are integers. In **Map**, the keys can be any objects.
- A map cannot contain duplicate keys. Each key maps to one value. A key and its corresponding value form an entry stored in a map.