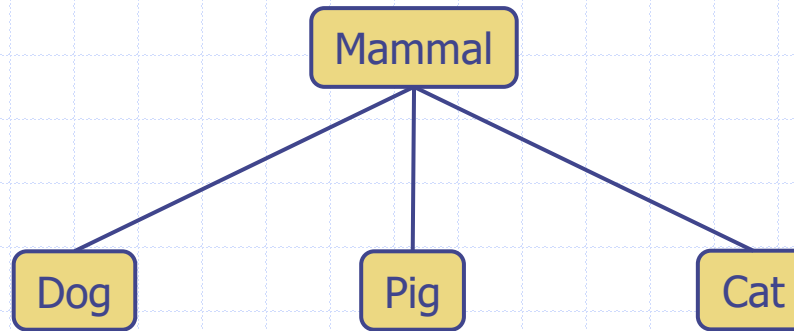


Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Trees



General Trees

- In this chapter, we will discuss one of the most important nonlinear data structures in computing — trees.
- Tree structures are a breakthrough in data organization, for they allow us to implement a host of algorithms much faster than when using linear data structures, such as arrays or linked lists.
- Trees also provide a natural organization for data, and consequently have become ubiquitous structures in file systems, graphical user interfaces, databases, websites, and many other computer systems.

General Trees

- When we say that trees are “nonlinear,” we are referring to an organizational relationship that is richer than the simple “before” and “after” relationships between objects in sequences.
- The relationships in a tree are hierarchical, with some objects being “above” and some “below” others.

General Trees

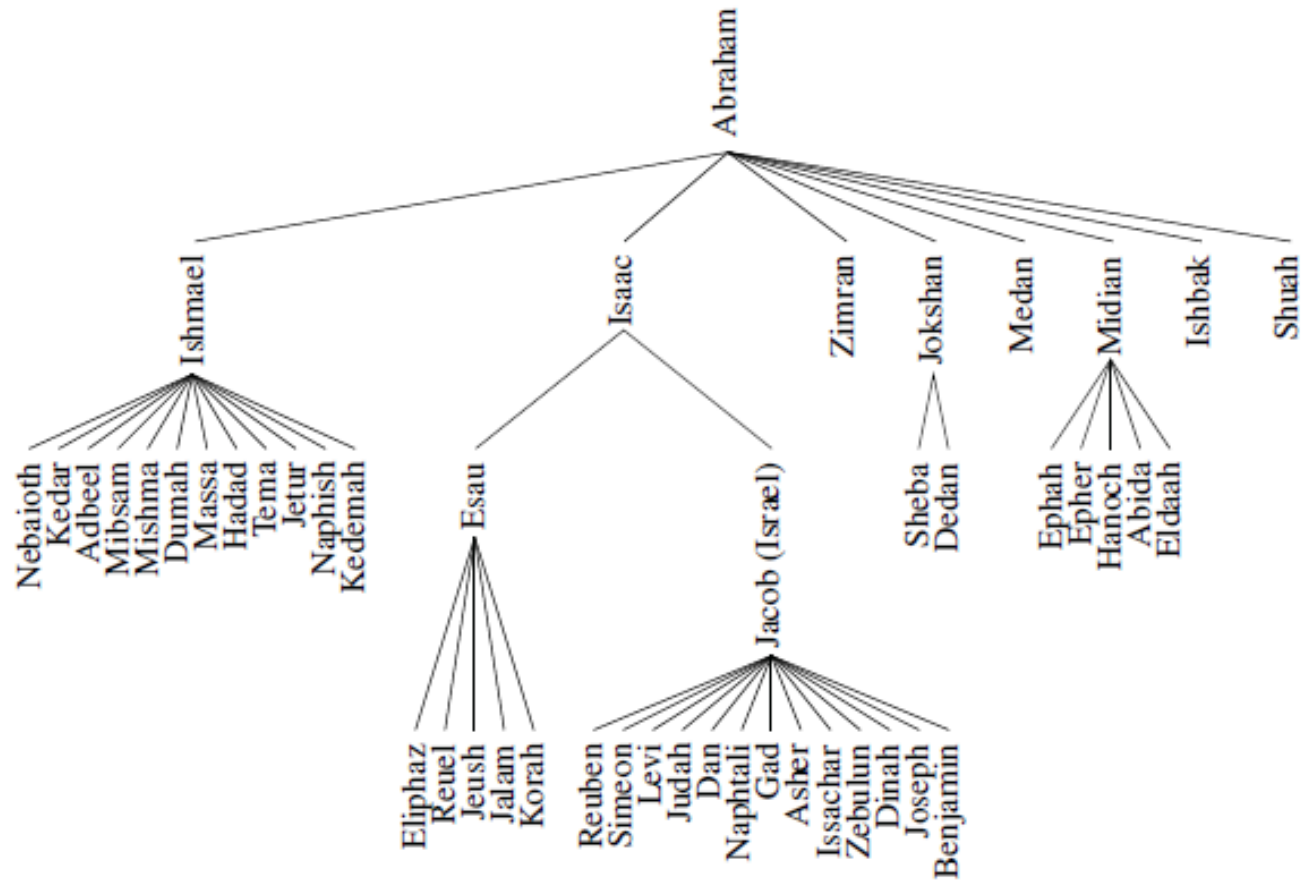
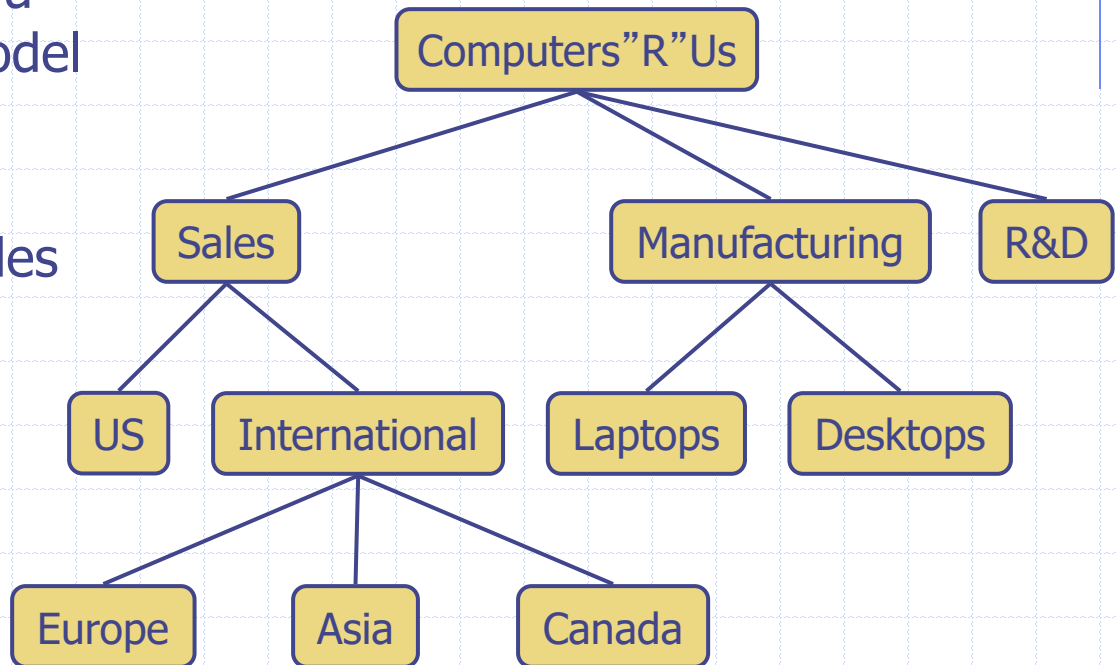


Figure 8.1: A family tree showing some descendants of Abraham, as recorded in Genesis, chapters 25–36.

What is a Tree

- ❑ In computer science, a tree is an abstract model of a hierarchical structure
- ❑ A tree consists of nodes with a parent-child relation
- ❑ Applications:
 - Organization charts
 - File systems
 - Programming environments



Recursive Definition of Tree

- According to our definition, a tree can be empty, meaning that it does not have any nodes.
- This convention also allows us to define a tree recursively such that a tree T is either empty or consists of a node r , called the root of T , and a (possibly empty) set of sub-trees whose roots are the children of r .

Using Trees to Represent a Portion of a File System

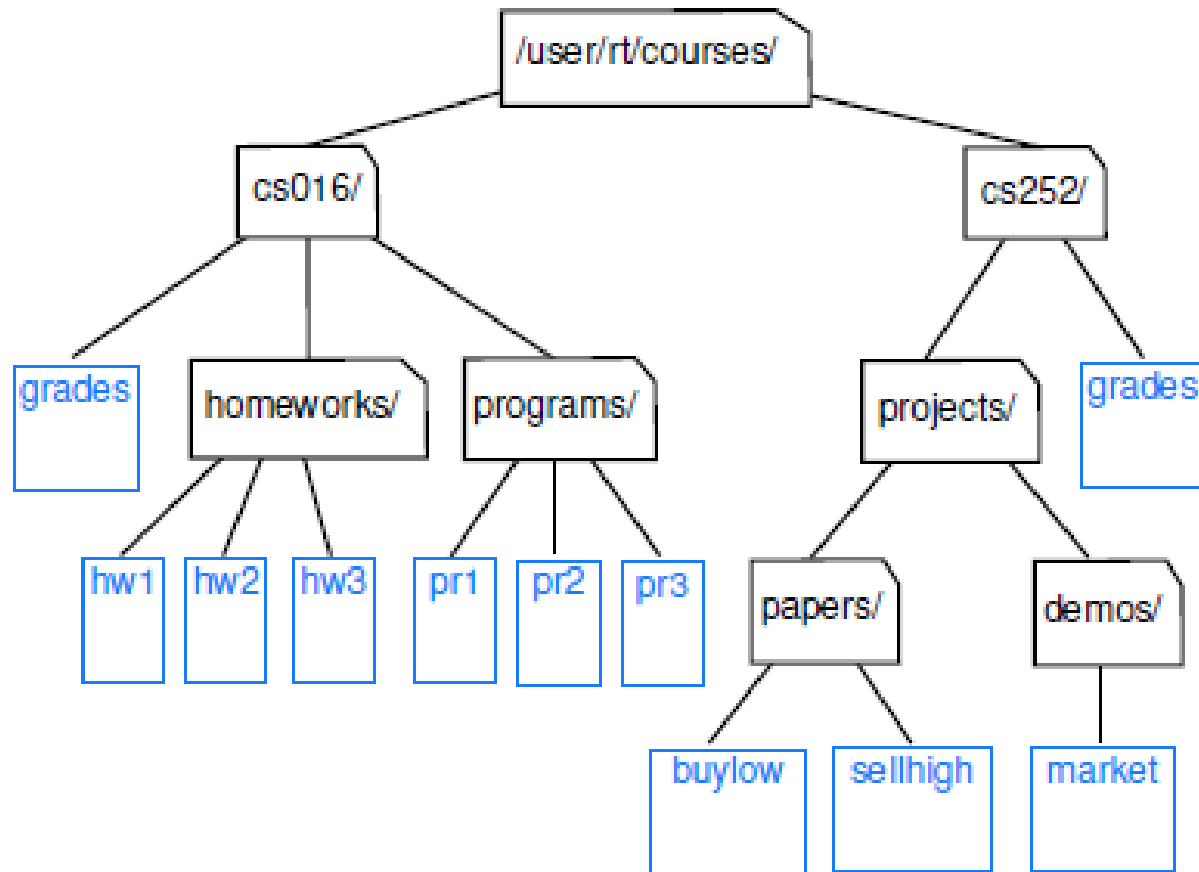
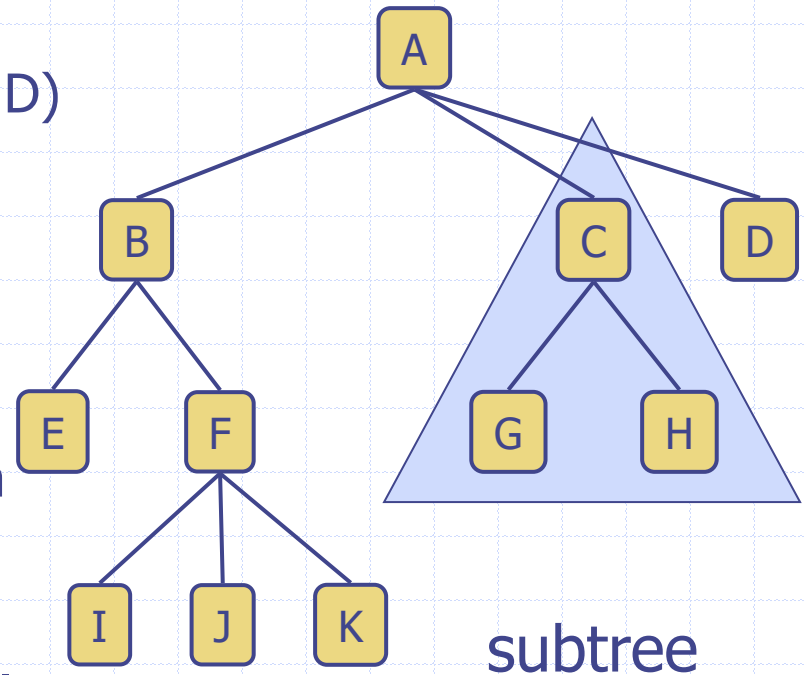


Figure 8.3: Tree representing a portion of a file system.

Tree Terminology

- ❑ Root: node without parent (A)
- ❑ Internal node: node with at least one child (A, B, C, F)
- ❑ External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- ❑ Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- ❑ Depth of a node: number of ancestors
- ❑ Height of a tree: maximum depth of any node (3)
- ❑ Descendant of a node: child, grandchild, grand-grandchild, etc.
- ❑ Subtree: tree consisting of a node and its descendants



Edges and Paths in Trees

- An edge of tree T is a pair of nodes (u,v) such that u is the parent of v , or vice versa.
- A path of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge.

Ordered Trees

- A tree is ordered if there is a meaningful linear order among the children of each node; that is, we purposefully identify the children of a node as being the first, second, third, and so on.
- Such an order is usually visualized by arranging siblings left to right, according to their order

Ordered Trees

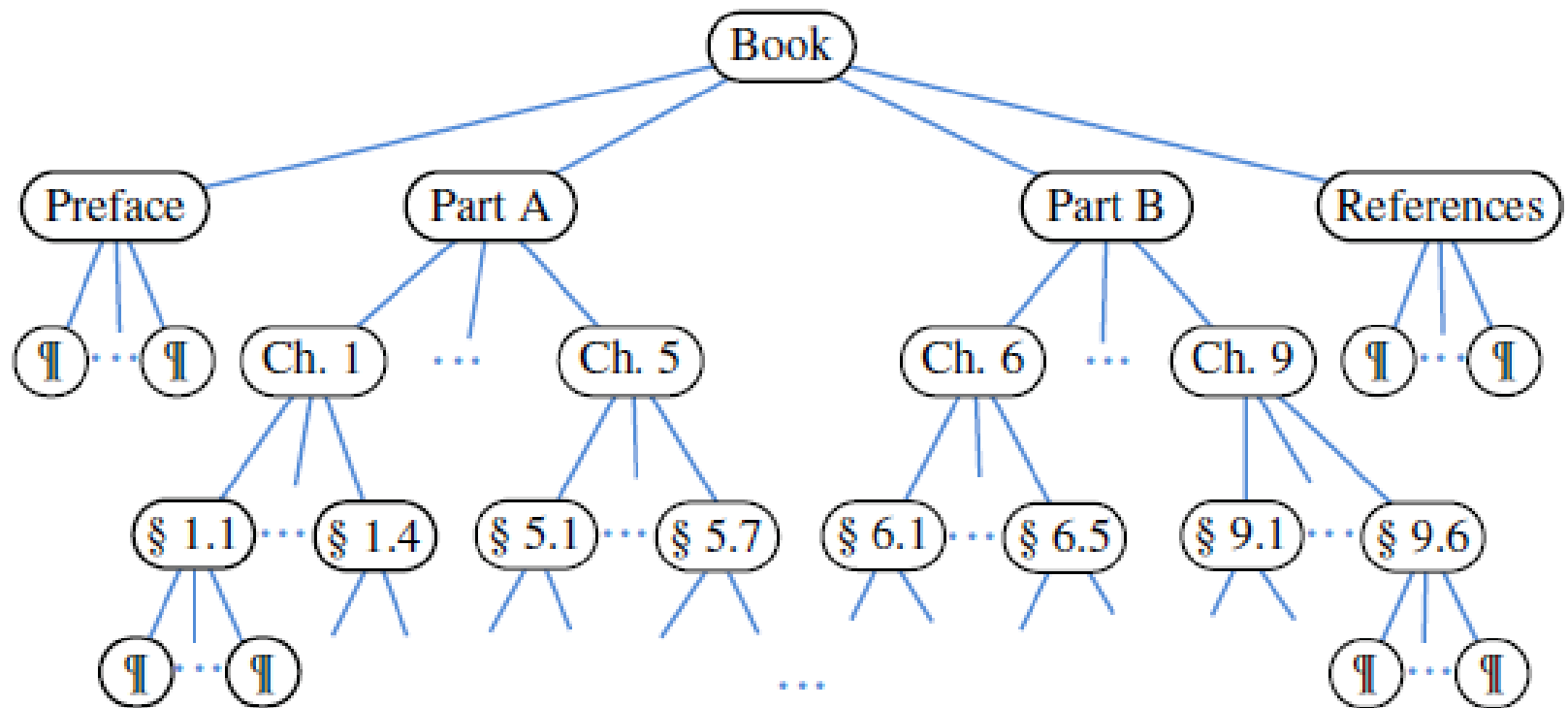


Figure 8.4: An ordered tree associated with a book.

Tree ADT

- We use positions to abstract nodes
- Generic methods:
 - integer `size()`
 - boolean `isEmpty()`
 - Iterator `iterator()`
 - Iterable `positions()`
- Accessor methods:
 - position `root()`
 - position `parent(p)`
 - Iterable `children(p)`
 - Integer `numChildren(p)`

◆ Query methods:

- boolean `isInternal(p)`
- boolean `isExternal(p)`
- boolean `isRoot(p)`

◆ Additional update methods may be defined by data structures implementing the Tree ADT

Java Interface

Methods for a Tree interface:

```
1  /** An interface for a tree where nodes can have an arbitrary number of children. */
2  public interface Tree<E> extends Iterable<E> {
3      Position<E> root();
4      Position<E> parent(Position<E> p) throws IllegalArgumentException;
5      Iterable<Position<E>> children(Position<E> p)
6                               throws IllegalArgumentException;
7      int numChildren(Position<E> p) throws IllegalArgumentException;
8      boolean isInternal(Position<E> p) throws IllegalArgumentException;
9      boolean isExternal(Position<E> p) throws IllegalArgumentException;
10     boolean isRoot(Position<E> p) throws IllegalArgumentException;
11     int size();
12     boolean isEmpty();
13     Iterator<E> iterator();
14     Iterable<Position<E>> positions();
15 }
```

An AbstractTree Base Class in Java

- ❑ An abstract class may define concrete implementations for some of its methods, while leaving other abstract methods without definition.
- ❑ An abstract class is designed to serve as a base class, through inheritance, for one or more concrete implementations of an interface.
- ❑ When some of the functionality of an interface is implemented in an abstract class, less work remains to complete a concrete implementation.
- ❑ Code Fragment 8.2 presents an initial implementation of an AbstractTree base class that provides the most trivial methods of the Tree interface.

An AbstractTree Base Class in Java

```
1  /** An abstract base class providing some functionality of the Tree interface. */
2  public abstract class AbstractTree<E> implements Tree<E> {
3      public boolean isInternal(Position<E> p) { return numChildren(p) > 0; }
4      public boolean isExternal(Position<E> p) { return numChildren(p) == 0; }
5      public boolean isRoot(Position<E> p) { return p == root(); }
6      public boolean isEmpty() { return size() == 0; }
7  }
```

Code Fragment 8.2: An initial implementation of the AbstractTree base class. (We add additional functionality to this class as the chapter continues.)

Computing Depth and Height

- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node
- The depth of p can also be recursively defined as follows:
 - If p is the root, then the depth of p is 0.
 - Otherwise, the depth of p is one plus the depth of the parent of p .

Computing Depth and Height

- ❑ Code Fragment 8.3, for computing the depth of a position p in tree T .
- ❑ This method calls itself recursively on the parent of p , and adds 1 to the value returned.
- ❑ The running time of $\text{depth}(p)$ for position p is $O(d_p + 1)$, where d_p denotes the depth of p in the tree, because the algorithm performs a constant-time recursive step for each ancestor of p .
- ❑ Thus, algorithm $\text{depth}(p)$ runs in $O(n)$ worst-case time, where n is the total number of positions of T , because a position of T may have depth $n-1$ if all nodes form a single branch.

Computing Depth and Height

```
1  /** Returns the number of levels separating Position p from the root. */
2  public int depth(Position<E> p) {
3      if (isRoot(p))
4          return 0;
5      else
6          return 1 + depth(parent(p));
7  }
```

Code Fragment 8.3: Method `depth`, as implemented within the `AbstractTree` class.

Computing Depth and Height

- ❑ We next define the height of a tree to be equal to the maximum of the depths of its positions (or zero, if the tree is empty).
- ❑ In Code Fragment 8.4, we present a method that computes the height of a tree based on this definition.
- ❑ Unfortunately, such an approach is not very efficient, and so name the algorithm `heightBad` and declare it as a private method of the `AbstractTree` class (so that it cannot be used by others).

Computing Depth and Height

```
1  /** Returns the height of the tree. */
2  private int heightBad() {                // works, but quadratic worst-case time
3      int h = 0;
4      for (Position<E> p : positions())
5          if (isExternal(p))              // only consider leaf positions
6              h = Math.max(h, depth(p));
7      return h;
8  }
```

Code Fragment 8.4: Method `heightBad` of the `AbstractTree` class. Note that this method calls the `depth` method from Code Fragment 8.3.

Computing Depth and Height

Time Complexity of heightBad:

- heightBad calls algorithm depth(p) on each leaf of T

its running time is $O(n + \sum_{p \in L} (d_p + 1))$,

where L is the set of leaf positions of T .

In the worst case, the sum $\sum_{p \in L} (d_p + 1)$ is proportional to n^2 .

- Thus, algorithm heightBad runs in $O(n^2)$ worst-case time.

Computing Depth and Height

- ❑ We can compute the height of a tree more efficiently, in $O(n)$ worst-case time, by considering a recursive definition.
- ❑ To do this, we will parameterize a function based on a position within the tree, and calculate the height of the sub-tree rooted at that position.
- ❑ Formally, we define the height of a position p in a tree T as follows:
 - ❑ If p is a leaf, then the height of p is 0.
 - ❑ Otherwise, the height of p is one more than the maximum of the heights of p 's children.
- ❑ An implementation of a recursive algorithm to compute the height of a sub-tree rooted at a given position p is presented in Code Fragment 8.5.

Computing Depth and Height

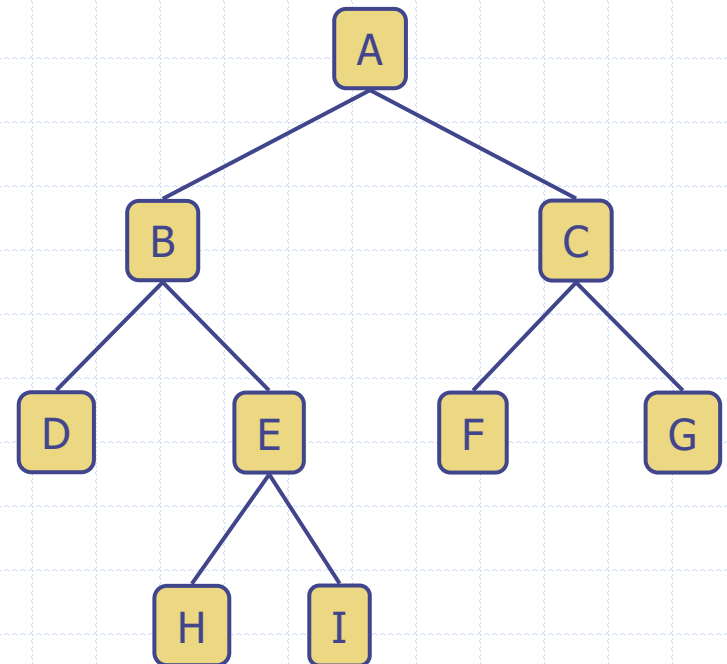
```
1  /** Returns the height of the subtree rooted at Position p. */
2  public int height(Position<E> p) {
3      int h = 0;                                // base case if p is external
4      for (Position<E> c : children(p))
5          h = Math.max(h, 1 + height(c));
6      return h;
7  }
```

Code Fragment 8.5: Method `height` for computing the height of a subtree rooted at a position p of an `AbstractTree`.

Binary Trees

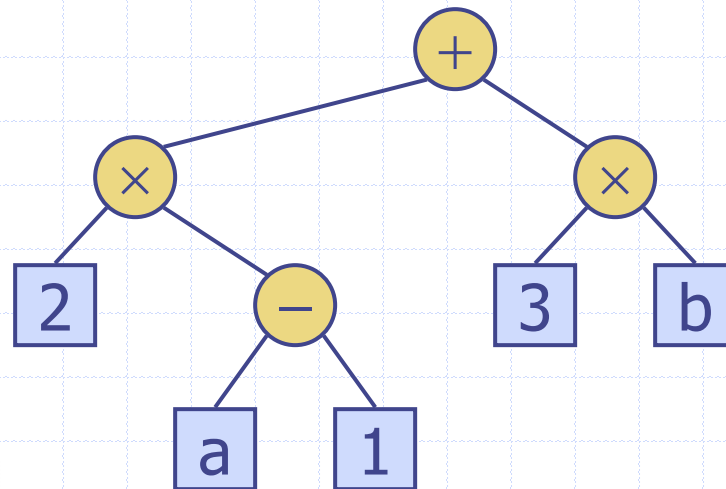
- A binary tree is a tree with the following properties:
 - Each internal node has at most two children (exactly two for **proper** binary trees)
 - The children of a node are an ordered pair
- We call the children of an internal node **left child** and **right child**
- Alternative recursive definition: 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



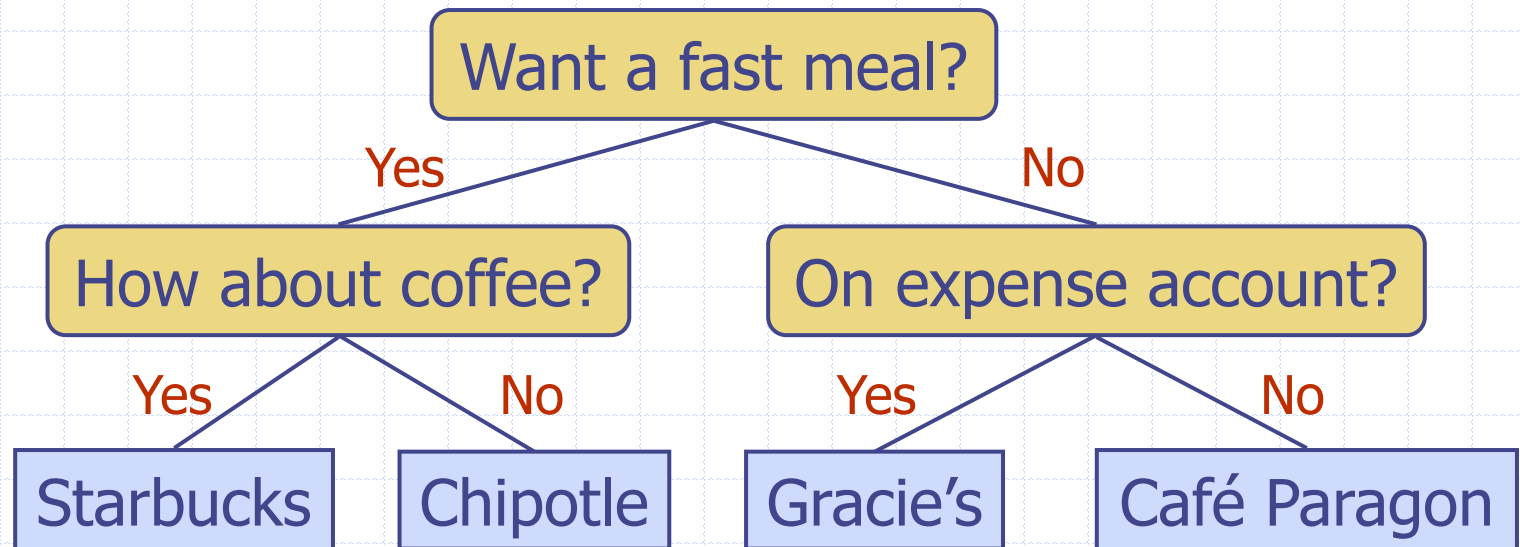
Arithmetic Expression Tree

- 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



BinaryTree ADT

- The **BinaryTree** ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
 - position **left**(p)
 - position **right**(p)
 - position **sibling**(p)
- The above methods return **null** when there is no left, right, or sibling of p, respectively
- Update methods may be defined by data structures implementing the BinaryTree ADT

Java Code for BinaryTree ADT

```
1  /** An interface for a binary tree, in which each node has at most two children. */
2  public interface BinaryTree<E> extends Tree<E> {
3      /** Returns the Position of p's left child (or null if no child exists). */
4      Position<E> left(Position<E> p) throws IllegalArgumentException;
5      /** Returns the Position of p's right child (or null if no child exists). */
6      Position<E> right(Position<E> p) throws IllegalArgumentException;
7      /** Returns the Position of p's sibling (or null if no sibling exists). */
8      Position<E> sibling(Position<E> p) throws IllegalArgumentException;
9  }
```

Code Fragment 8.6: A `BinaryTree` interface that extends the `Tree` interface from Code Fragment 8.1.

Defining an AbstractBinaryTree Base Class

- The AbstractBinaryTree class, presented in Code Fragment 8.7, inherits from the AbstractTree class.
- It provides additional concrete methods that can be derived from the newly declared left and right methods (which remain abstract).

Defining an AbstractBinary Tree Base Class

```
1  /** An abstract base class providing some functionality of the BinaryTree interface.*/
2  public abstract class AbstractBinaryTree<E> extends AbstractTree<E>
3                                     implements BinaryTree<E> {
4      /** Returns the Position of p's sibling (or null if no sibling exists). */
5      public Position<E> sibling(Position<E> p) {
6          Position<E> parent = parent(p);
7          if (parent == null) return null;           // p must be the root
8          if (p == left(parent))                    // p is a left child
9              return right(parent);                // (right child might be null)
10         else                                       // p is a right child
11             return left(parent);                  // (left child might be null)
12     }
13     /** Returns the number of children of Position p. */
14     public int numChildren(Position<E> p) {
15         int count=0;
16         if (left(p) != null)
17             count++;
18         if (right(p) != null)
19             count++;
20         return count;
21     }
22     /** Returns an iterable collection of the Positions representing p's children. */
23     public Iterable<Position<E>> children(Position<E> p) {
24         List<Position<E>> snapshot = new ArrayList<>(2); // max capacity of 2
25         if (left(p) != null)
26             snapshot.add(left(p));
27         if (right(p) != null)
28             snapshot.add(right(p));
29         return snapshot;
30     }
31 }
```

Code Fragment 8.7: An AbstractBinaryTree class that extends the AbstractTree class of Code Fragment 8.2 and implements the BinaryTree interface of Code Fragment 8.6.

Properties of Proper Binary Trees

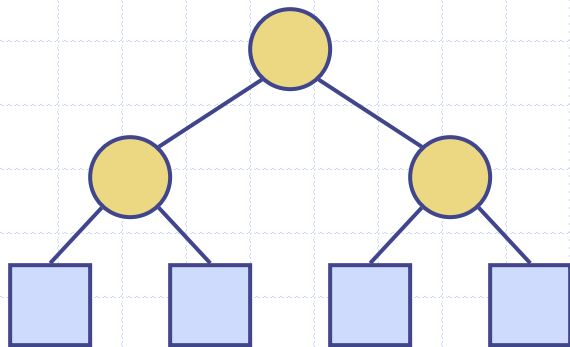
□ Notation

n number of nodes

e number of
external nodes

i number of internal
nodes

h height



◆ Properties:

■ $e = i + 1$

■ $n = 2e - 1$

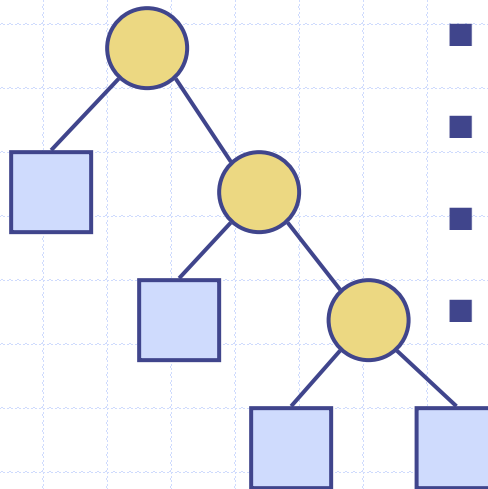
■ $h \leq i$

■ $h \leq (n - 1)/2$

■ $e \leq 2^h$

■ $h \geq \log_2 e$

■ $h \geq \log_2 (n + 1) - 1$



Implementing Trees

- We have not yet defined key implementation details for how a tree will be represented internally, and how we can effectively navigate between parents and children.
- There are several choices for the internal representation of trees.
- We describe the most common representations in this section.

Linked Structure for Binary Trees

- **A natural way to realize a binary tree T is to use a *linked structure*, with a node (see Figure next slide)**
- **It maintains references to the element stored at a position p and to the nodes associated with the children and parent of p .**

Linked Structure for Binary Trees

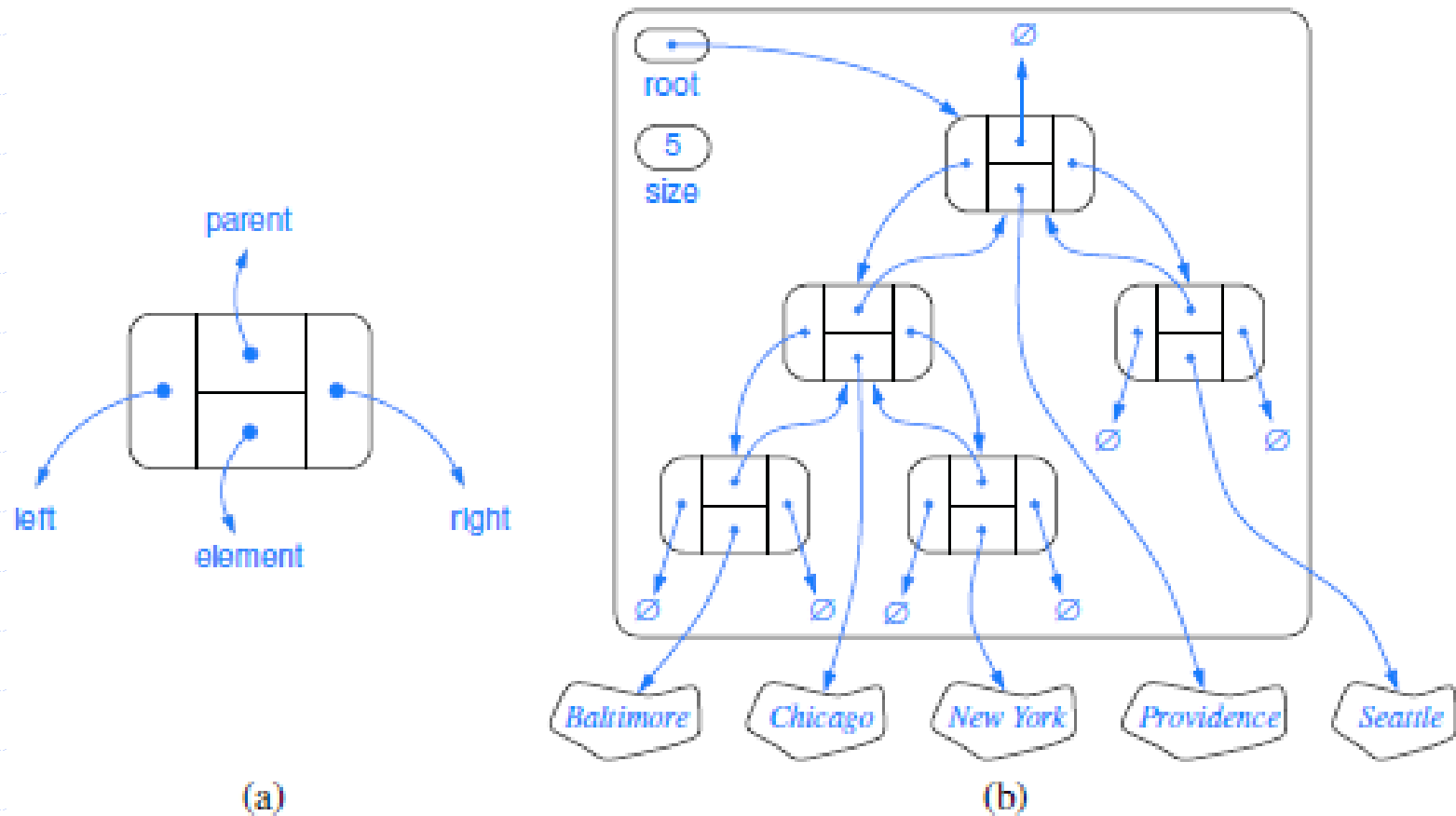
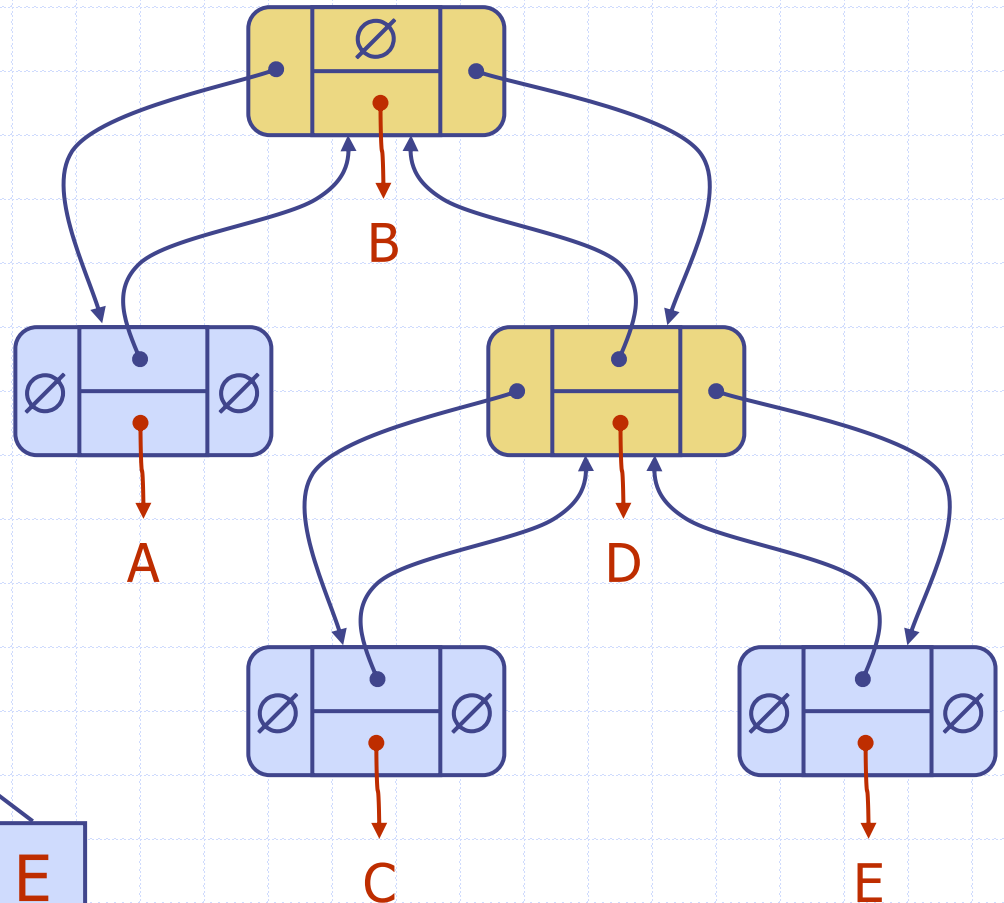
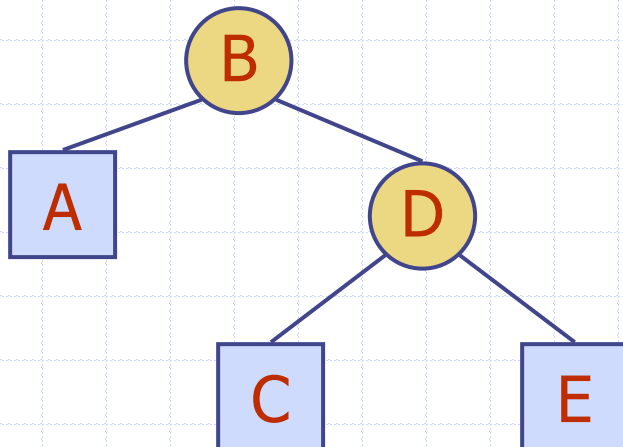


Figure 8.9: A linked structure for representing: (a) a single node; (b) a binary tree.

Linked Structure for Binary Trees

- A node is represented by an object storing
 - Element
 - Parent node
 - Left child node
 - Right child node
- Node objects implement the Position ADT



Operations for Updating a Linked Binary Tree

- `addRoot( $e$ )`: Creates a root for an empty tree, storing e as the element, and returns the position of that root; an error occurs if the tree is not empty.
- `addLeft( $p, e$ )`: Creates a left child of position p , storing element e , and returns the position of the new node; an error occurs if p already has a left child.
- `addRight( $p, e$ )`: Creates a right child of position p , storing element e , and returns the position of the new node; an error occurs if p already has a right child.
- `set( $p, e$ )`: Replaces the element stored at position p with element e , and returns the previously stored element.
- `attach( $p, T_1, T_2$ )`: Attaches the internal structure of trees T_1 and T_2 as the respective left and right subtrees of leaf position p and resets T_1 and T_2 to empty trees; an error condition occurs if p is not a leaf.
- `remove( $p$ )`: Removes the node at position p , replacing it with its child (if any), and returns the element that had been stored at p ; an error occurs if p has two children.

Java Implementation of a Linked Binary Tree Structure

- The entire `LinkedBinaryTree` class is presented in Code Fragments 8.8–8.11.
- The `remove` method, given at the end of Code Fragment 8.11:
 - intentionally sets the `parent` field of a deleted node to refer to itself.
 - This is our conventional representation of a defunct node (as detected within the `validate` method).
 - It resets all other fields to null, to aid in garbage collection.

Java Implementation of a Linked Binary Tree Structure

```
1  /** Concrete implementation of a binary tree using a node-based, linked structure. */
2  public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {
3
4      //----- nested Node class -----
5      protected static class Node<E> implements Position<E> {
6          private E element;                // an element stored at this node
7          private Node<E> parent;            // a reference to the parent node (if any)
8          private Node<E> left;              // a reference to the left child (if any)
9          private Node<E> right;             // a reference to the right child (if any)
10         /** Constructs a node with the given element and neighbors. */
11         public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild) {
12             element = e;
13             parent = above;
14             left = leftChild;
15             right = rightChild;
16         }
17         // accessor methods
18         public E getElement() { return element; }
19         public Node<E> getParent() { return parent; }
20         public Node<E> getLeft() { return left; }
21         public Node<E> getRight() { return right; }
22         // update methods
23         public void setElement(E e) { element = e; }
24         public void setParent(Node<E> parentNode) { parent = parentNode; }
25         public void setLeft(Node<E> leftChild) { left = leftChild; }
26         public void setRight(Node<E> rightChild) { right = rightChild; }
27     } //----- end of nested Node class -----
28 }
```

Code Fragment 8.8: An implementation of the LinkedBinaryTree class.
(Continues in Code Fragments 8.9–8.11.)

Java Implementation of a Linked Binary Tree Structure

```
29  /** Factory function to create a new node storing element e. */
30  protected Node<E> createNode(E e, Node<E> parent,
31                               Node<E> left, Node<E> right) {
32      return new Node<E>(e, parent, left, right);
33  }
34
35  // LinkedBinaryTree instance variables
36  protected Node<E> root = null;           // root of the tree
37  private int size = 0;                     // number of nodes in the tree
38
39  // constructor
40  public LinkedBinaryTree() { }             // constructs an empty binary tree
```

Code Fragment 8.8: An implementation of the `LinkedBinaryTree` class.
(Continues in Code Fragments 8.9–8.11.)

Java Implementation of a Linked Binary Tree Structure

```
41 // nonpublic utility
42 /** Validates the position and returns it as a node. */
43 protected Node<E> validate(Position<E> p) throws IllegalArgumentException {
44     if (!(p instanceof Node))
45         throw new IllegalArgumentException("Not valid position type");
46     Node<E> node = (Node<E>) p;           // safe cast
47     if (node.getParent() == node)        // our convention for defunct node
48         throw new IllegalArgumentException("p is no longer in the tree");
49     return node;
50 }
51
52 // accessor methods (not already implemented in AbstractBinaryTree)
53 /** Returns the number of nodes in the tree. */
54 public int size() {
55     return size;
56 }
57
58 /** Returns the root Position of the tree (or null if tree is empty). */
59 public Position<E> root() {
60     return root;
61 }
62
```

Code Fragment 8.9: An implementation of the `LinkedBinaryTree` class.

(Continued from Code Fragment 8.8; continues in Code Fragments 8.10 and 8.11.)

Java Implementation of a Linked Binary Tree Structure

```
63  /** Returns the Position of p's parent (or null if p is root). */
64  public Position<E> parent(Position<E> p) throws IllegalArgumentException {
65      Node<E> node = validate(p);
66      return node.getParent();
67  }
68
69  /** Returns the Position of p's left child (or null if no child exists). */
70  public Position<E> left(Position<E> p) throws IllegalArgumentException {
71      Node<E> node = validate(p);
72      return node.getLeft();
73  }
74
75  /** Returns the Position of p's right child (or null if no child exists). */
76  public Position<E> right(Position<E> p) throws IllegalArgumentException {
77      Node<E> node = validate(p);
78      return node.getRight();
79  }
```

Code Fragment 8.9: An implementation of the `LinkedBinaryTree` class.

(Continued from Code Fragment 8.8; continues in Code Fragments 8.10 and 8.11.)

Java Implementation of a Linked Binary Tree Structure

```
80 // update methods supported by this class
81 /** Places element e at the root of an empty tree and returns its new Position. */
82 public Position<E> addRoot(E e) throws IllegalStateException {
83     if (!isEmpty()) throw new IllegalStateException("Tree is not empty");
84     root = createNode(e, null, null, null);
85     size = 1;
86     return root;
87 }
88
89 /** Creates a new left child of Position p storing element e; returns its Position. */
90 public Position<E> addLeft(Position<E> p, E e)
91     throws IllegalArgumentException {
92     Node<E> parent = validate(p);
93     if (parent.getLeft() != null)
94         throw new IllegalArgumentException("p already has a left child");
95     Node<E> child = createNode(e, parent, null, null);
96     parent.setLeft(child);
97     size++;
98     return child;
99 }
100
```

Code Fragment 8.10: An implementation of the `LinkedBinaryTree` class.
(Continued from Code Fragments 8.8 and 8.9; continues in Code Fragment 8.11.)

Java Implementation of a Linked Binary Tree Structure

```
101  /** Creates a new right child of Position p storing element e; returns its Position. */
102  public Position<E> addRight(Position<E> p, E e)
103      throws IllegalArgumentException {
104      Node<E> parent = validate(p);
105      if (parent.getRight() != null)
106          throw new IllegalArgumentException("p already has a right child");
107      Node<E> child = createNode(e, parent, null, null);
108      parent.setRight(child);
109      size++;
110      return child;
111  }
112
113  /** Replaces the element at Position p with e and returns the replaced element. */
114  public E set(Position<E> p, E e) throws IllegalArgumentException {
115      Node<E> node = validate(p);
116      E temp = node.getElement();
117      node.setElement(e);
118      return temp;
119  }
```

Code Fragment 8.10: An implementation of the `LinkedBinaryTree` class.

(Continued from Code Fragments 8.8 and 8.9; continues in Code Fragment 8.11.)

Java Implementation of a Linked Binary Tree Structure

```
120  /** Attaches trees t1 and t2 as left and right subtrees of external p. */
121  public void attach(Position<E> p, LinkedBinaryTree<E> t1,
122                    LinkedBinaryTree<E> t2) throws IllegalArgumentException {
123      Node<E> node = validate(p);
124      if (isInternal(p)) throw new IllegalArgumentException("p must be a leaf");
125      size += t1.size() + t2.size();
126      if (!t1.isEmpty()) { // attach t1 as left subtree of node
127          t1.root.setParent(node);
128          node.setLeft(t1.root);
129          t1.root = null;
130          t1.size = 0;
131      }
132      if (!t2.isEmpty()) { // attach t2 as right subtree of node
133          t2.root.setParent(node);
134          node.setRight(t2.root);
135          t2.root = null;
136          t2.size = 0;
137      }
138  }
```

Code Fragment 8.11: An implementation of the `LinkedBinaryTree` class.
(Continued from Code Fragments 8.8–8.10.)

Java Implementation of a Linked Binary Tree Structure

```
139  /** Removes the node at Position p and replaces it with its child, if any. */
140  public E remove(Position<E> p) throws IllegalArgumentException {
141      Node<E> node = validate(p);
142      if (numChildren(p) == 2)
143          throw new IllegalArgumentException("p has two children");
144      Node<E> child = (node.getLeft() != null ? node.getLeft() : node.getRight());
145      if (child != null)
146          child.setParent(node.getParent()); // child's grandparent becomes its parent
147      if (node == root)
148          root = child; // child becomes root
149      else {
150          Node<E> parent = node.getParent();
151          if (node == parent.getLeft())
152              parent.setLeft(child);
153          else
154              parent.setRight(child);
155      }
156      size--;
157      E temp = node.getElement();
158      node.setElement(null); // help garbage collection
159      node.setLeft(null);
160      node.setRight(null);
161      node.setParent(null); // our convention for defunct node
162      return temp;
163  }
164  } //----- end of LinkedBinaryTree class -----
```

Code Fragment 8.11: An implementation of the `LinkedBinaryTree` class.
(Continued from Code Fragments 8.8–8.10.)

Performance of the Linked Binary Tree Implementation

Method	Running Time
size, isEmpty	$O(1)$
root, parent, left, right, sibling, children, numChildren	$O(1)$
isInternal, isExternal, isRoot	$O(1)$
addRoot, addLeft, addRight, set, attach, remove	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

Table 8.1: Running times for the methods of an n -node binary tree implemented with a linked structure. The space usage is $O(n)$.

Array-Based Representation of Binary Trees

- An alternative representation of a binary tree T is based on a way of numbering the positions of T .
- For every position p of T , let $f(p)$ be the integer defined as follows.
 - If p is the root of T , then $f(p) = 0$.
 - If p is the left child of position q , then $f(p) = 2 * f(q) + 1$.
 - If p is the right child of position q , then $f(p) = 2 * f(q) + 2$.
- The numbering function f is known as a level numbering of the positions in a binary tree T , for it numbers the positions on each level of T in increasing order from left to right. (See Figure 8.10.)

Array-Based Representation of Binary Trees

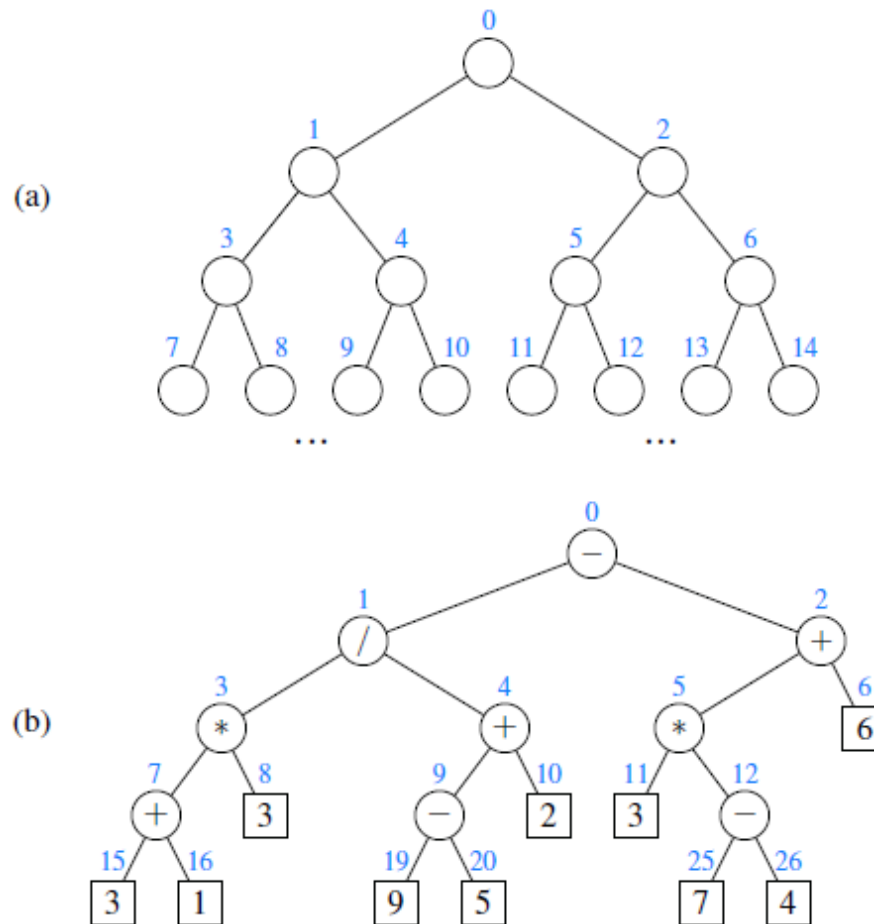
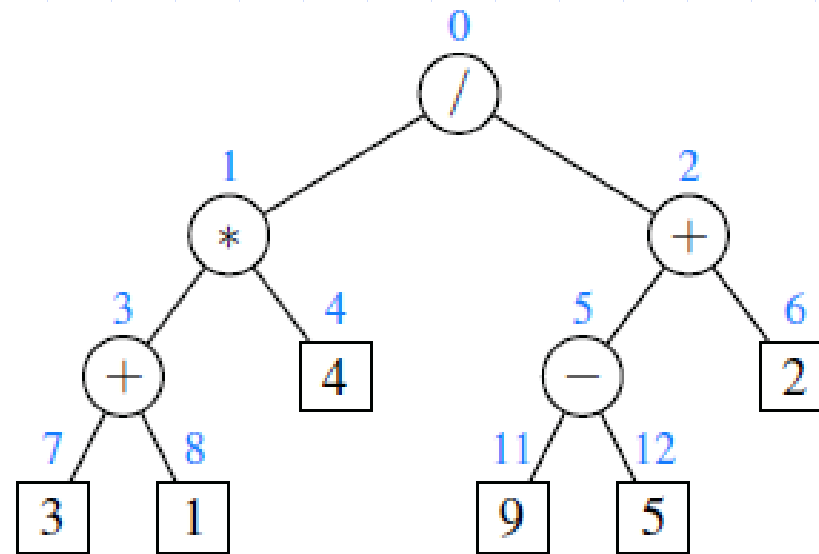


Figure 8.10: Binary tree level numbering: (a) general scheme; (b) an example.

Array-Based Representation of Binary Trees

- The level numbering function f suggests a representation of a binary tree T by means of an array-based structure A .
- The element at position p of T stored at index $f(p)$ of the array.
- We show an example of an array-based representation of a binary tree in Figure 8.11.

Array-Based Representation of Binary Trees

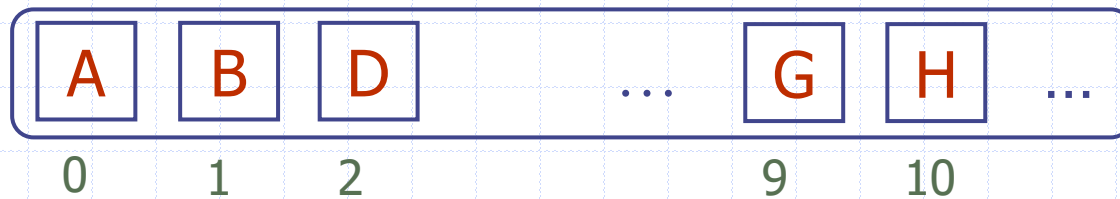


/	*	+	+	4	-	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Figure 8.11: Representation of a binary tree by means of an array.

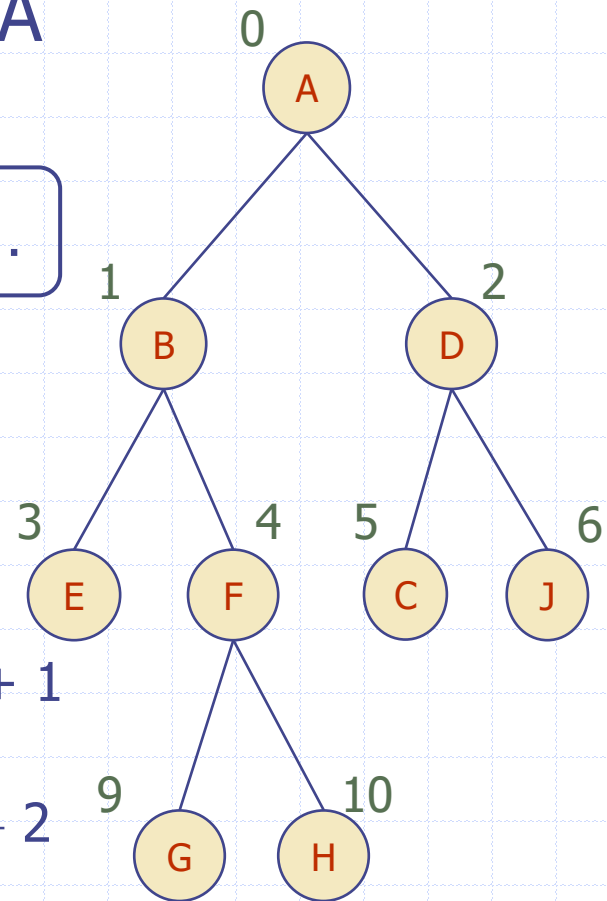
Array-Based Representation of Binary Trees

- Nodes are stored in an array A



Node v is stored at $A[\text{rank}(v)]$

- $\text{rank}(\text{root}) = 0$
- if node is the left child of $\text{parent}(\text{node})$,
 $\text{rank}(\text{node}) = 2 \cdot \text{rank}(\text{parent}(\text{node})) + 1$
- if node is the right child of $\text{parent}(\text{node})$,
 $\text{rank}(\text{node}) = 2 \cdot \text{rank}(\text{parent}(\text{node})) + 2$



Adv. Of Array-Based Representation of Binary Trees

- One advantage of an array-based representation of a binary tree is that a position p can be represented by the single integer $f(p)$.
- Position-based methods such as root, parent, left, and right can be implemented using simple arithmetic operations on the number $f(p)$.
- Based on our formula for the level numbering,
 - Left child of p has index $2f(p)+1$,
 - Right child of p has index $2f(p)+2$, and
 - Parent of p has index $\lfloor (f(p)-1)/2 \rfloor$.

Disadv. Of Array-Based Representation of Binary Trees

1. The space usage of an array-based representation depends greatly on the shape of the tree. Let n be the number of nodes of T , and let fM be the maximum value of $f(p)$ over all the nodes of T . The array A requires length $N = 1 + fM$, since elements range from $A[0]$ to $A[fM]$. Note that A may have a number of empty cells that do not refer to existing positions of T . In fact, in the worst case, $N = 2n - 1$.
2. Another drawback of an array representation is that many update operations for trees cannot be efficiently supported. For example, removing a node and promoting its child takes $O(n)$ time because it is not just the child that moves locations within the array, but all descendants of that child.

Linked Structure for General Trees

- A natural way to realize a general tree T as a linked structure is to have each node store a single *container* of references to its children.
- For example, a children field of a node can be an array or list of references to the children of the node (if any).

Linked Structure for General Trees

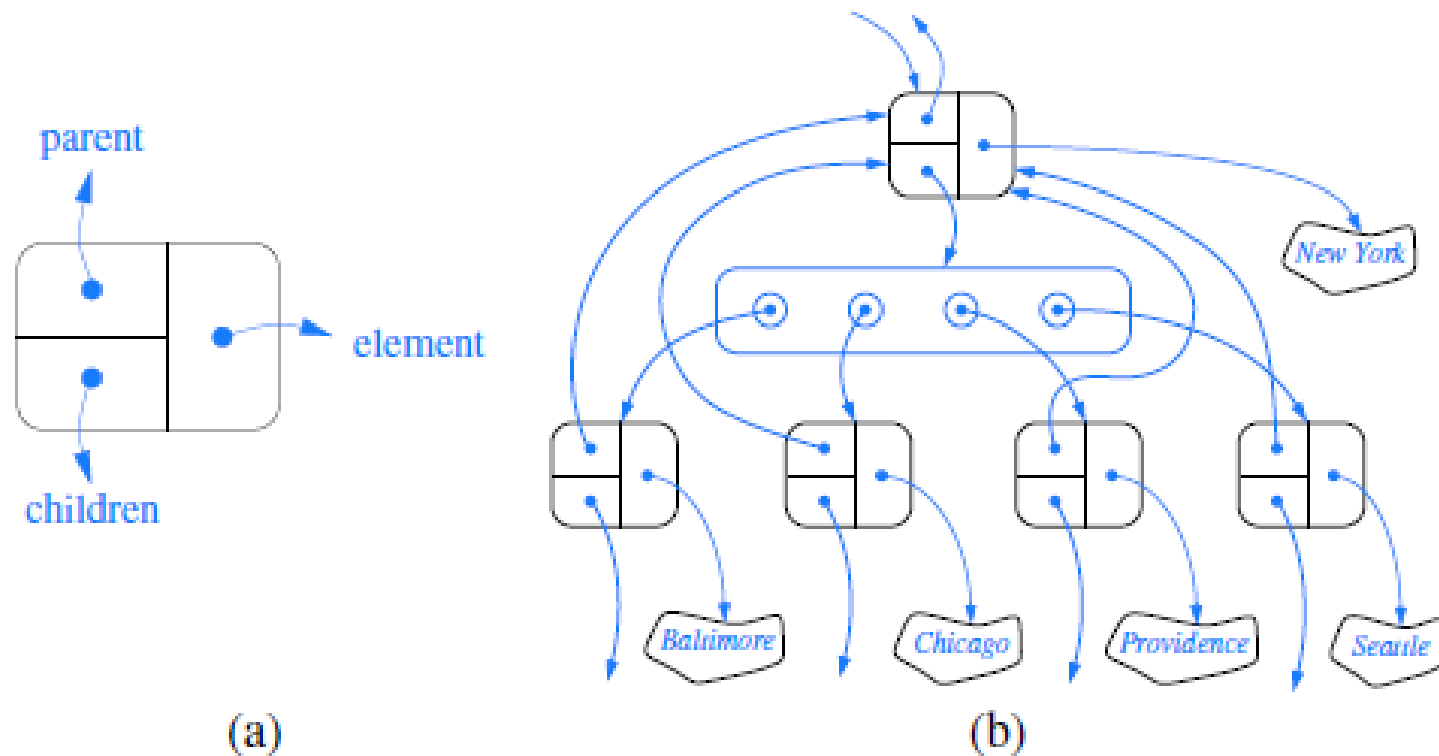
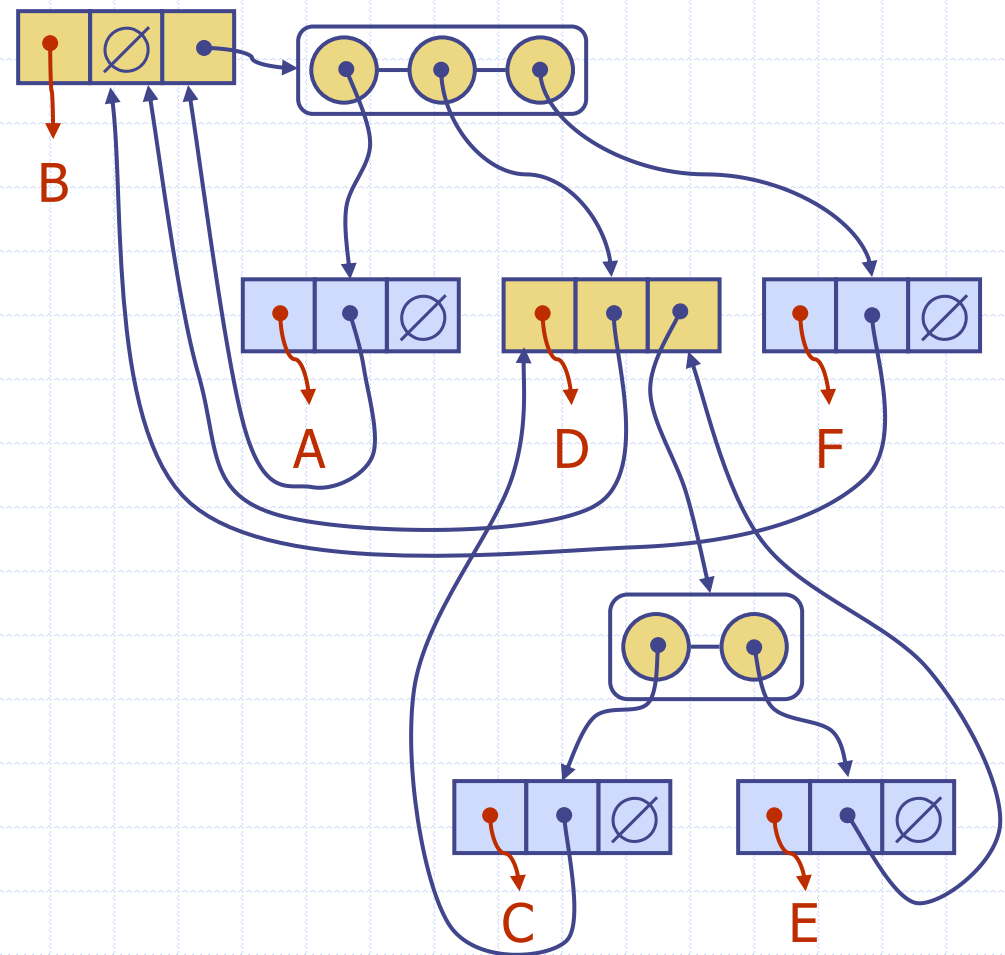
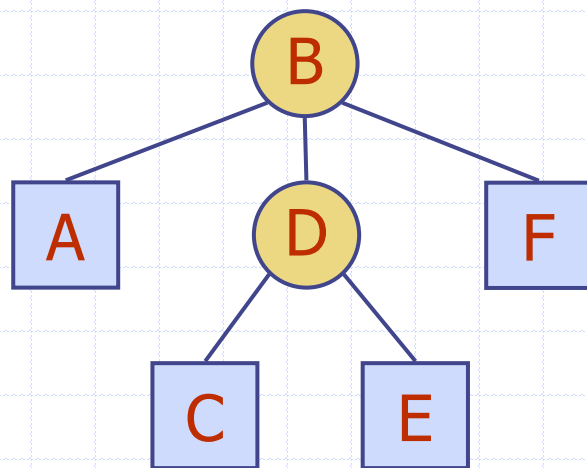


Figure 8.12: The linked structure for a general tree: (a) the structure of a node; (b) a larger portion of the data structure associated with a node and its children.

Linked Structure for Trees

- A node is represented by an object storing
 - Element
 - Parent node
 - Sequence of children nodes
- Node objects implement the Position ADT



Performance of Implementation of General Trees Using a Linked Structure

Method	Running Time
size, isEmpty	$O(1)$
root, parent, isRoot, isInternal, isExternal	$O(1)$
numChildren(p)	$O(1)$
children(p)	$O(c_p + 1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

Table 8.2: Running times of the accessor methods of an n -node general tree implemented with a linked structure. We let c_p denote the number of children of a position p , and d_p its depth. The space usage is $O(n)$.

Tree Traversal Algorithms

- **Traversal** of a tree T is a systematic way of accessing, or “visiting,” all the positions of T .
- The specific action associated with the “visit” of a position p depends on the application of this traversal, and could involve anything from:
 - incrementing a counter to
 - performing some complex computation for p .
- In this section, we describe several common traversal schemes for trees, implement them in the context of our various tree classes, and discuss several common applications of tree traversals.

Preorder Traversals of General Trees

- In a ***preorder traversal*** of a tree T , the root of T is visited first and then the subtrees rooted at its children are traversed recursively.
- If the tree is ordered, then the subtrees are traversed according to the order of the children.
- The pseudocode for the preorder traversal of the subtree rooted at a position p is shown in Code Fragment 8.12.

Preorder Traversals of General Trees

Algorithm `preorder( $p$ )`:

```
perform the “visit” action for position  $p$     { this happens before any recursion }  
for each child  $c$  in children( $p$ ) do  
    preorder( $c$ )                                { recursively traverse the subtree rooted at  $c$  }
```

Code Fragment 8.12: Algorithm `preorder` for performing the preorder traversal of a subtree rooted at position p of a tree.

Preorder Traversals of General Trees

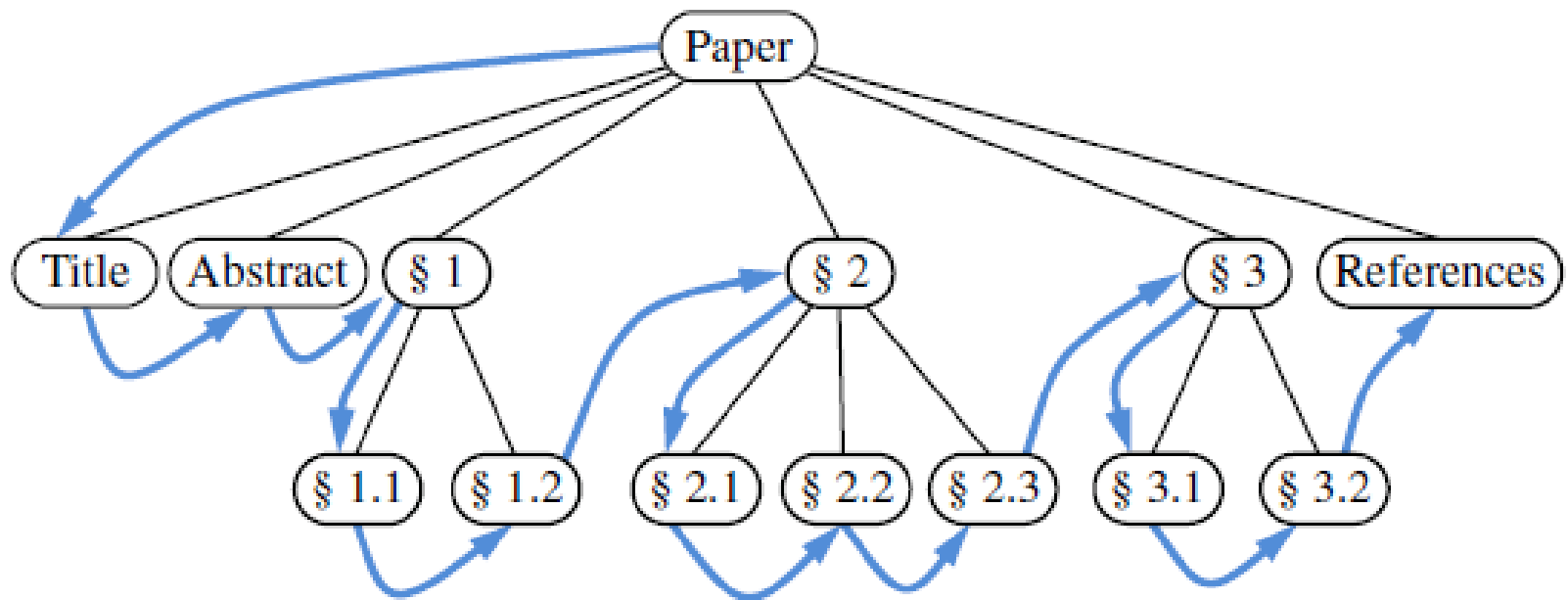
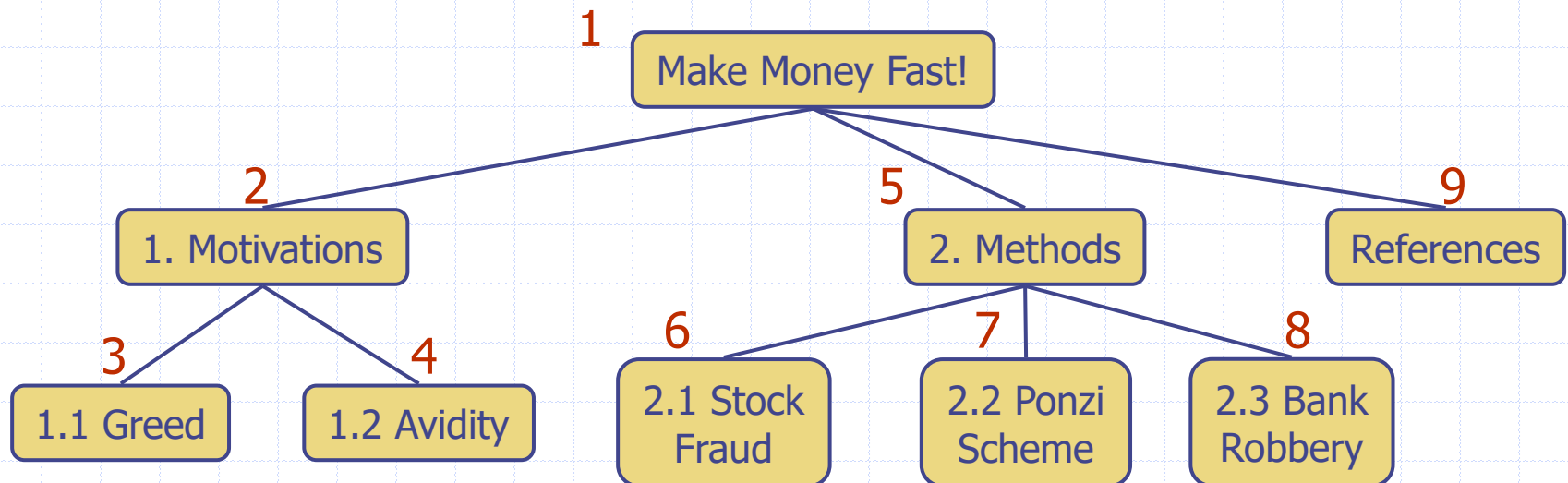


Figure 8.13: Preorder traversal of an ordered tree, where the children of each position are ordered from left to right.

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 of General Trees

- In some sense, *postorder traversal* can be viewed as the opposite of the preorder traversal, because it recursively traverses the subtrees rooted at the children of the root first, and then visits the root (hence, the name “postorder”).
- Pseudocode for the postorder traversal is given in Code Fragment 8.13, and an example of a postorder traversal is portrayed in Figure 8.14.

Postorder Traversal of General Trees

Algorithm postorder(p):

 for each child c in children(p) do

 postorder(c) { recursively traverse the subtree rooted at c }

 perform the “visit” action for position p { this happens after any recursion }

Code Fragment 8.13: Algorithm postorder for performing the postorder traversal of a subtree rooted at position p of a tree.

Postorder Traversal of General Trees

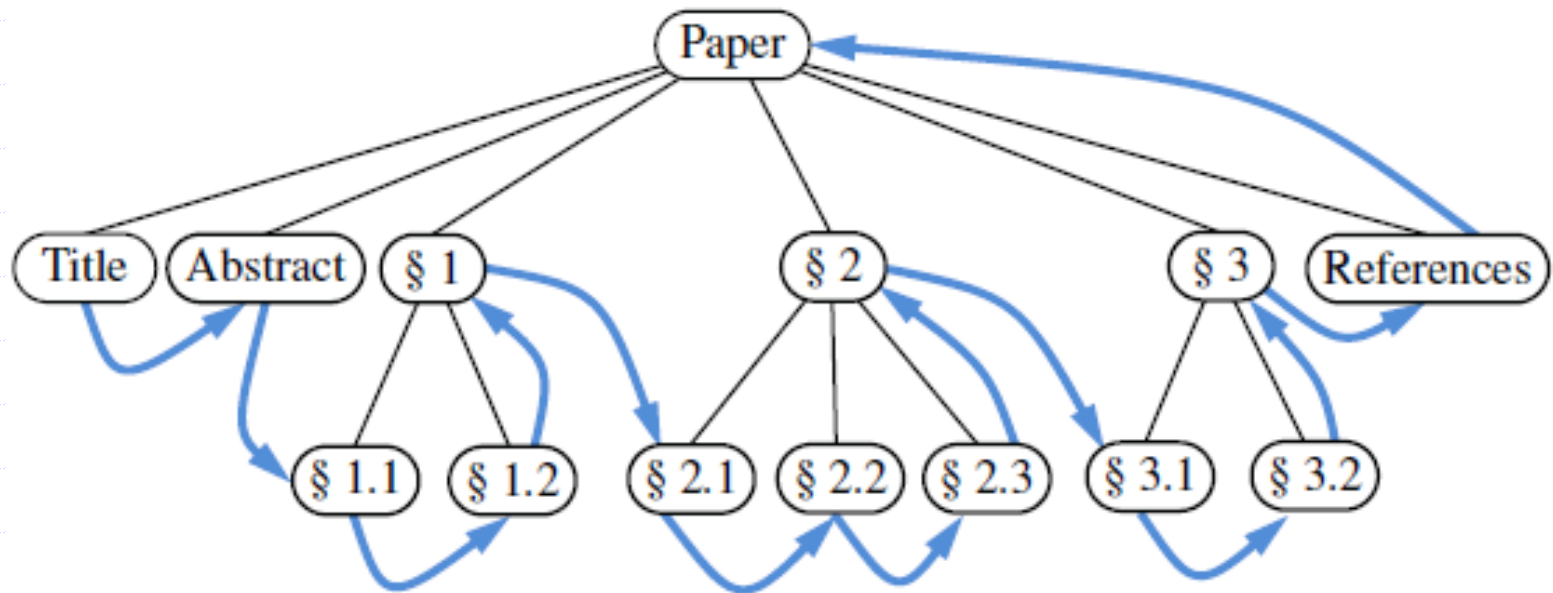
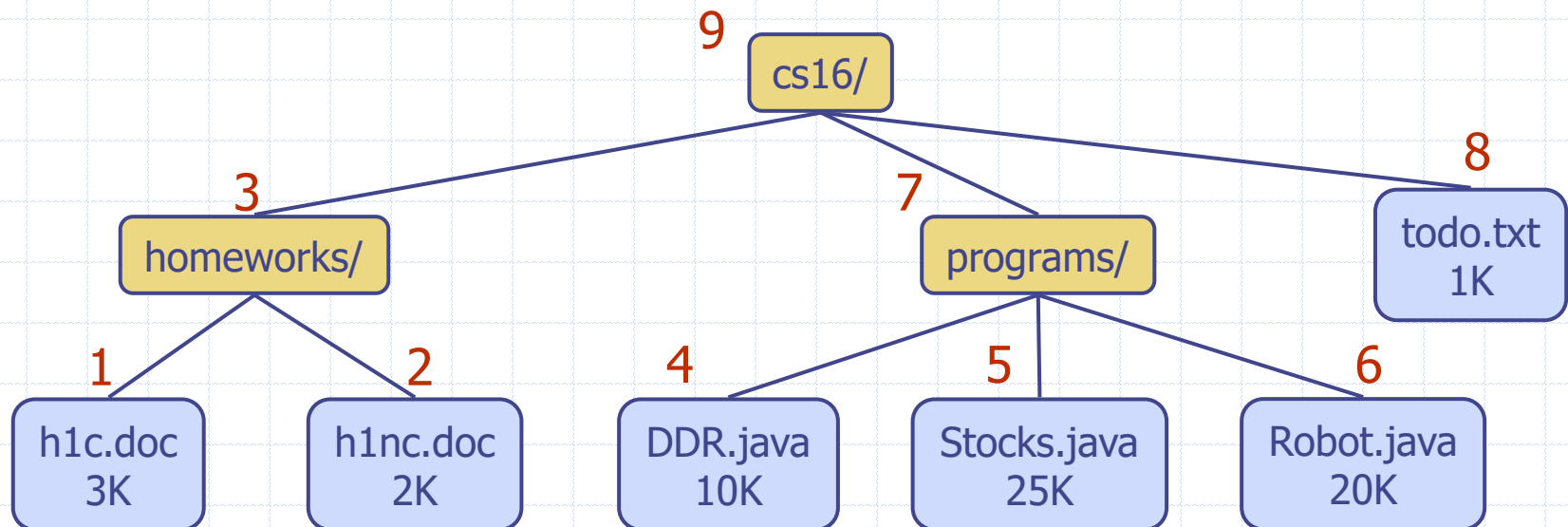


Figure 8.14: Postorder traversal of the ordered tree of Figure 8.13.

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)



Running-Time Analysis

- Both preorder and postorder traversal algorithms are efficient ways to access all the positions of a tree.
- The analysis of either of these traversal algorithms is similar to that of algorithm height, given in Code Fragment 8.5 of Section 8.1.3.
- At each position p , the nonrecursive part of the traversal algorithm requires time $O(cp+1)$, where cp is the number of children of p , under the assumption that the “visit” itself takes $O(1)$ time.
- By Proposition 8.4, the overall running time for the traversal of tree T is $O(n)$, where n is the number of positions in the tree.
- This running time is asymptotically optimal since the traversal must visit all n positions of the tree.

Breadth-First Tree Traversal

- Although the preorder and postorder traversals are common ways of visiting the positions of a tree, another approach is to traverse a tree so that we visit all the positions at depth d before we visit the positions at depth $d+1$.
- Such an algorithm is known as a ***breadth-first traversal***.

Breadth-First Tree Traversal

- A breadth-first traversal is a common approach used in software for playing games.
- A ***game tree*** represents the possible choices of moves that might be made by a player (or computer) during a game, with the root of the tree being the initial configuration for the game.
- For example, Figure 8.15 displays partial game tree for Tic-Tac-Toe.

Breadth-First Tree Traversal

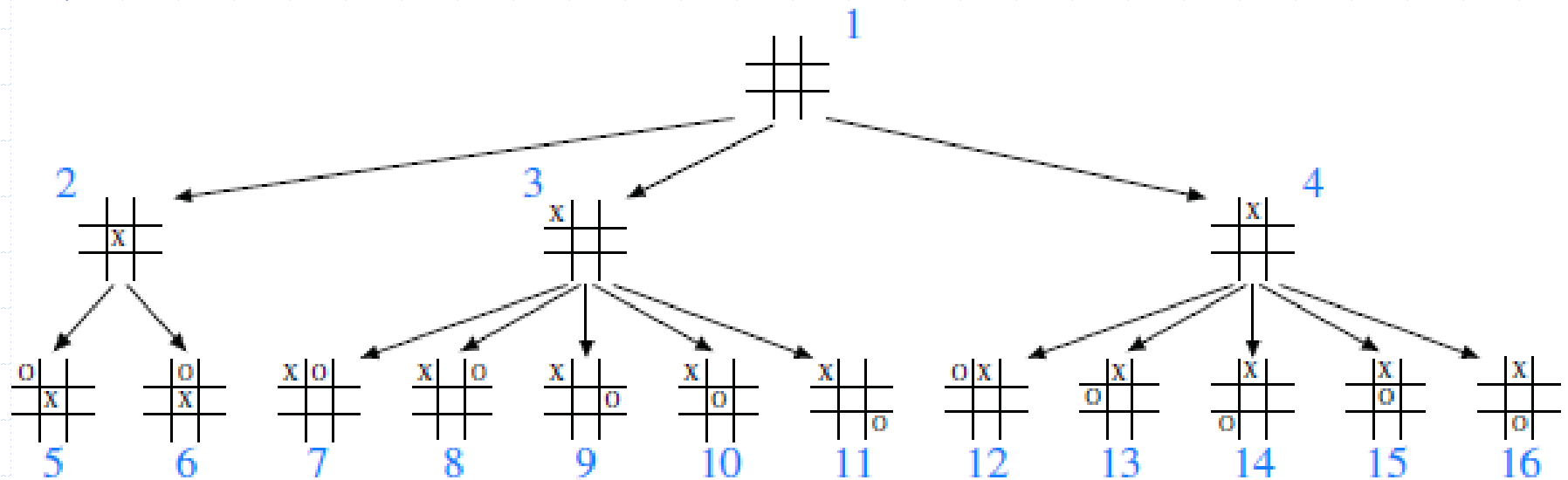


Figure 8.15: Partial game tree for Tic-Tac-Toe when ignoring symmetries; annotations denote the order in which positions are visited in a breadth-first tree traversal.

Breadth-First Tree Traversal

- Pseudocode for a breadth-first traversal is given in Code Fragment 8.14.
- The process is not recursive, since we are not traversing entire subtrees at once.
- We use a queue to produce a FIFO (i.e., first-in first-out) semantics for the order in which we visit nodes.
- The overall running time is $O(n)$, due to the n calls to enqueue and n calls to dequeue.

Breadth-First Tree Traversal

Algorithm breadthfirst():

Initialize queue Q to contain root()

while Q not empty do

$p = Q.dequeue()$

 { p is the oldest entry in the queue }

 perform the “visit” action for position p

 for each child c in children(p) do

$Q.enqueue(c)$ { add p ’s children to the end of the queue for later visits }

Code Fragment 8.14: Algorithm for performing a breadth-first traversal of a tree.

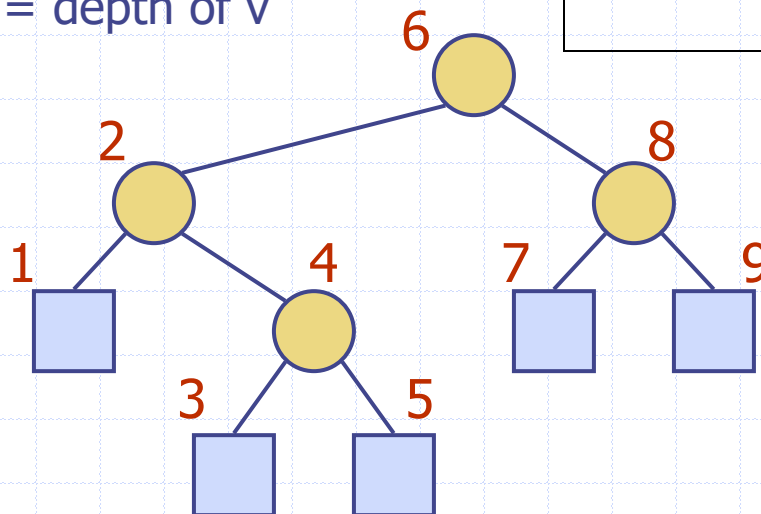
Inorder Traversal of Binary Trees

- The standard preorder, postorder, and breadth-first traversals that were introduced for general trees can be directly applied to binary trees.
- In this section, we will introduce another common traversal algorithm specifically for a binary tree.

Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
 - $x(v)$ = inorder rank of v
 - $y(v)$ = depth of v

```
Algorithm inOrder( $v$ )  
  if  $left(v) \neq null$   
    inOrder ( $left(v)$ )  
  visit( $v$ )  
  if  $right(v) \neq null$   
    inOrder ( $right(v)$ )
```



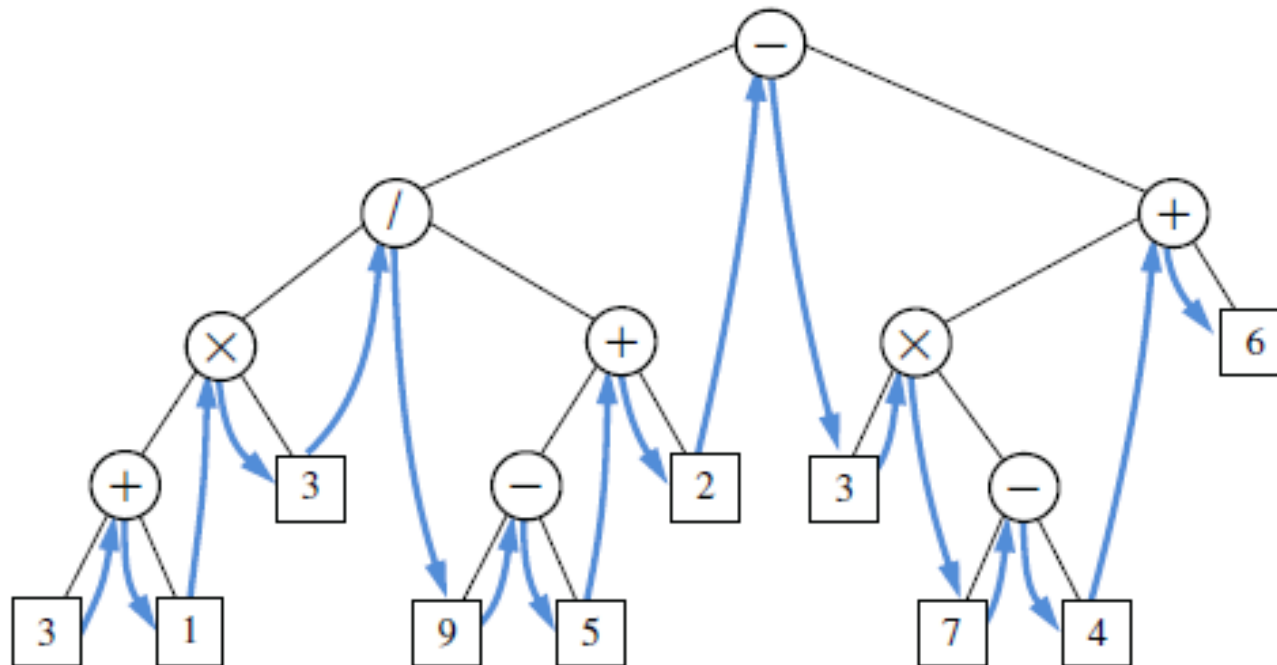
[illegible]

Figure 8.16: Inorder traversal of a binary tree.

Binary Search Trees

- An important application of the inorder traversal algorithm arises when we store an ordered sequence of elements in a binary tree, defining a structure we call a ***binary search tree***.
- Let S be a set whose unique elements have an order relation. For example, S could be a set of integers.
- A binary search tree for S is a proper binary tree T such that, for each internal position p of T :
 - Position p stores an element of S , denoted as $e(p)$.
 - Elements stored in the left subtree of p (if any) are less than $e(p)$.
 - Elements stored in the right subtree of p (if any) are greater than $e(p)$.

Binary Search Trees

- An example of a binary search tree is shown in Figure 8.17.
- The above properties assure that an inorder traversal of a binary search tree T visits the elements in nondecr

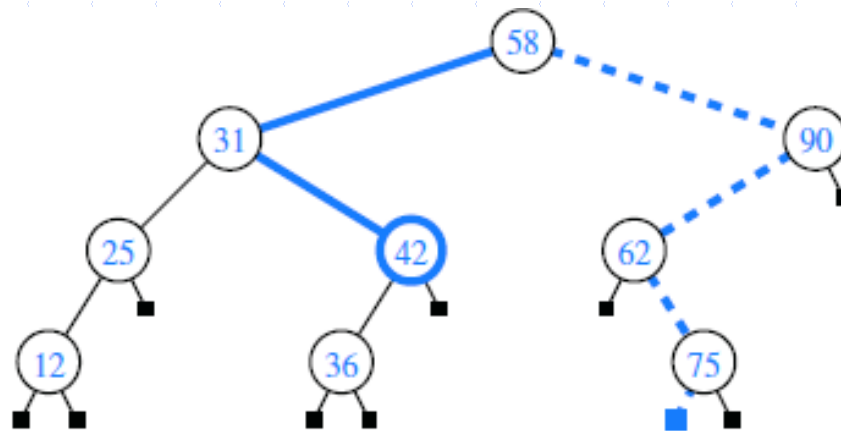


Figure 8.17: A binary search tree storing integers. The solid path is traversed when searching (successfully) for 42. The dashed path is traversed when searching (unsuccessfully) for 70.

Implementing Tree Traversals in Java

- *T* must include the following supporting methods:
 - **iterator()**: Returns an iterator for all elements in the tree.
 - **positions()**: Returns an iterable collection of all positions of the tree.
- We will demonstrate how any of the tree traversal algorithms can be used to produce these iterations as concrete implementations within the `AbstractTree` or `AbstractBinaryTree` base classes.

Implementing Tree Traversals in Java- Implementing iterator()

- First, we note that an iteration of all *elements* of a tree can easily be produced if we have an iteration of all *positions* of that tree.
- Code Fragment 8.16 provides an implementation of the iterator() method by adapting an iteration produced by the positions() method.

Implementing Tree Traversals in Java- Implementing iterator()

```
1  //----- nested ElementIterator class -----
2  /* This class adapts the iteration produced by positions() to return elements. */
3  private class ElementIterator implements Iterator<E> {
4      Iterator<Position<E>> posIterator = positions().iterator();
5      public boolean hasNext() { return posIterator.hasNext(); }
6      public E next() { return posIterator.next().getElement(); } // return element!
7      public void remove() { posIterator.remove(); }
8  }
9
10 /** Returns an iterator of the elements stored in the tree. */
11 public Iterator<E> iterator() { return new ElementIterator(); }
```

Code Fragment 8.16: Iterating all elements of an Abstract Tree instance, based upon an iteration of the positions of the tree.

Implementing Tree Traversals in Java- Implementing positions()

- To implement the positions() method, we have a choice of tree traversal algorithms.
- We provide public implementations of each tree traversal strategy that can be called directly by a user of our class.
- We can then trivially adapt one of those as a default order for the positions method of the AbstractTree class.
- For example, we will define a public method, preorder(), that returns an iteration of the positions of a tree in preorder.
- Code Fragment 8.17 demonstrates how the positions() method can be trivially defined to rely on that order.

```
public Iterable<Position<E>> positions() { return preorder(); }
```

Code Fragment 8.17: Defining preorder as the default traversal algorithm for the public positions method of an abstract tree.

Implementing Tree Traversals in Java- Preorder Traversals

- A public method `preorder()`, in `AbstractTree` class, returns an iterable container of the positions of the tree in preorder.
- We choose to produce a ***snapshot iterator***, returning a list of all positions.
- We begin by defining a private utility method, `preorderSubtree`, which allows us to parameterize the recursive process with a specific position of the tree that serves as the root of a subtree to traverse.

```
1  /** Adds positions of the subtree rooted at Position p to the given snapshot. */
2  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot) {
3      snapshot.add(p);          // for preorder, we add position p before exploring subtrees
4      for (Position<E> c : children(p))
5          preorderSubtree(c, snapshot);
6  }
```

Code Fragment 8.18: A recursive subroutine for performing a preorder traversal of the subtree rooted at position *p* of a tree. This code should be included within the body of the `AbstractTree` class.

Implementing Tree Traversals in Java- Preorder Traversals

- The public preorder method, has creates an empty list for the snapshot buffer, and invokes the recursive method at the root of the tree.

```
1  /** Returns an iterable collection of positions of the tree, reported in preorder. */
2  public Iterable<Position<E>> preorder() {
3      List<Position<E>> snapshot = new ArrayList<>();
4      if (!isEmpty())
5          preorderSubtree(root(), snapshot);    // fill the snapshot recursively
6      return snapshot;
7  }
```

Code Fragment 8.19: A public method that performs a preorder traversal of an entire tree. This code should be included within the body of the `AbstractTree` class.

Implementing Tree Traversals in Java- Postorder Traversals

We implement a ***postorder traversal*** using a similar design as we used for a preorder traversal. The only difference is that a “visited” position is not added to a postorder snapshot until *after* all of its subtrees have been traversed.

```
1  /** Adds positions of the subtree rooted at Position p to the given snapshot. */
2  private void postorderSubtree(Position<E> p, List<Position<E>> snapshot) {
3      for (Position<E> c : children(p))
4          postorderSubtree(c, snapshot);
5      snapshot.add(p);      // for postorder, we add position p after exploring subtrees
6  }
7  /** Returns an iterable collection of positions of the tree, reported in postorder. */
8  public Iterable<Position<E>> postorder() {
9      List<Position<E>> snapshot = new ArrayList<>();
10     if (!isEmpty())
11         postorderSubtree(root(), snapshot);    // fill the snapshot recursively
12     return snapshot;
13 }
```

Code Fragment 8.20: Support for performing a postorder traversal of a tree. This code should be included within the body of the `AbstractTree` class.

Implementing Tree Traversals in Java-

Breadth-First Traversal

- **Breadth-first traversal algorithm is not recursive.**
- **It relies on a queue of positions to manage the traversal process.**
- **We will use the `LinkedList` class.**

Implementing Tree Traversals in Java- Breadth-First Traversal

```
1  /** Returns an iterable collection of positions of the tree in breadth-first order. */
2  public Iterable<Position<E>> breadthfirst() {
3      List<Position<E>> snapshot = new ArrayList<>();
4      if (!isEmpty()) {
5          Queue<Position<E>> fringe = new LinkedList<>();
6          fringe.enqueue(root());           // start with the root
7          while (!fringe.isEmpty()) {
8              Position<E> p = fringe.dequeue(); // remove from front of the queue
9              snapshot.add(p);                 // report this position
10             for (Position<E> c : children(p))
11                 fringe.enqueue(c);           // add children to back of queue
12         }
13     }
14     return snapshot;
15 }
```

Code Fragment 8.21: An implementation of a breadth-first traversal of a tree. This code should be included within the body of the `AbstractTree` class.

Implementing Tree Traversals in Java-

Inorder Traversal for Binary Trees

- We use a similar design to our preorder and postorder traversals, with a private recursive utility for traversing subtrees. (See Code Fragment 8.22.)
- For many applications of binary trees, inorder traversal is the most natural order.
- Therefore, Code Fragment 8.22 makes it the default for the `AbstractBinaryTree` class by overriding the `positions` method that was inherited from the `AbstractTree` class.
- Because the `iterator( )` method relies on `positions( )`, it will also use inorder when reporting the elements of a binary tree.

Implementing Tree Traversals in Java-

Inorder Traversal for Binary Trees

```
1  /** Adds positions of the subtree rooted at Position p to the given snapshot. */
2  private void inorderSubtree(Position<E> p, List<Position<E>> snapshot) {
3      if (left(p) != null)
4          inorderSubtree(left(p), snapshot);
5      snapshot.add(p);
6      if (right(p) != null)
7          inorderSubtree(right(p), snapshot);
8  }
9  /** Returns an iterable collection of positions of the tree, reported in inorder. */
10 public Iterable<Position<E>> inorder() {
11     List<Position<E>> snapshot = new ArrayList<>();
12     if (!isEmpty())
13         inorderSubtree(root(), snapshot);    // fill the snapshot recursively
14     return snapshot;
15 }
16 /** Overrides positions to make inorder the default order for binary trees. */
17 public Iterable<Position<E>> positions() {
18     return inorder();
19 }
```

Code Fragment 8.22: Support for performing an inorder traversal of a binary tree, and for making that order the default traversal for binary trees. This code should be included within the body of the `AbstractBinaryTree` class.

Applications of Tree Traversals- Table of Contents

- When using a tree to represent the hierarchical structure of a document, a preorder traversal of the tree can be used to produce a table of contents for the document.
- For example, the table of contents associated with the tree from Figure 8.13 is displayed in Figure 8.18.
- Part (a) of that figure gives a simple presentation with one element per line.
- Part (b) shows a more attractive presentation, produced by indenting each element based on its depth within the tree.

Applications of Tree Traversals-Table of Contents

Paper	Paper
Title	Title
Abstract	Abstract
§1	§1
§1.1	§1.1
§1.2	§1.2
§2	§2
§2.1	§2.1
...	...
(a)	(b)

Figure 8.18: Table of contents for a document represented by the tree in Figure 8.13:
(a) without indentation; (b) with indentation based on depth within the tree.

The unindented version of the table of contents can be produced with the following code, given a tree *T* supporting the `preorder( )` method:

```
for (Position<E> p : T.preorder( ))  
    System.out.println(p.getElement( ));
```

Applications of Tree Traversals- Computing Disk Space

- The recursive computation of disk space is emblematic of a *postorder* traversal, as we cannot effectively compute the total space used by a directory until *after* we know the space that is used by its children directories.
- We would like to have a mechanism for children to return information to the parent as part of the traversal process.
- A custom solution to the disk space problem, with each level of recursion providing a return value to the (parent) caller, is provided in Code Fragment 8.25.

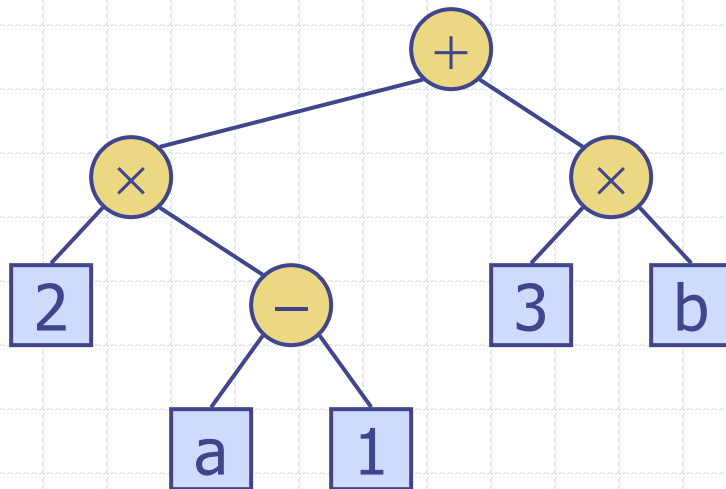
Applications of Tree Traversals- Computing Disk Space

```
1  /** Returns total disk space for subtree of T rooted at p. */
2  public static int diskSpace(Tree<Integer> T, Position<Integer> p) {
3      int subtotal = p.getElement();    // we assume element represents space usage
4      for (Position<Integer> c : T.children(p))
5          subtotal += diskSpace(T, c);
6      return subtotal;
7  }
```

Code Fragment 8.25: Recursive computation of disk space for a tree. We assume that each tree element reports the local space used at that position.

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 *printExpression(v)*

```
if left(v) ≠ null
    print("(")
    inOrder(left(v))
print(v.element())
if right(v) ≠ null
    inOrder(right(v))
    print(")')
```

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

Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
 - 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* ()

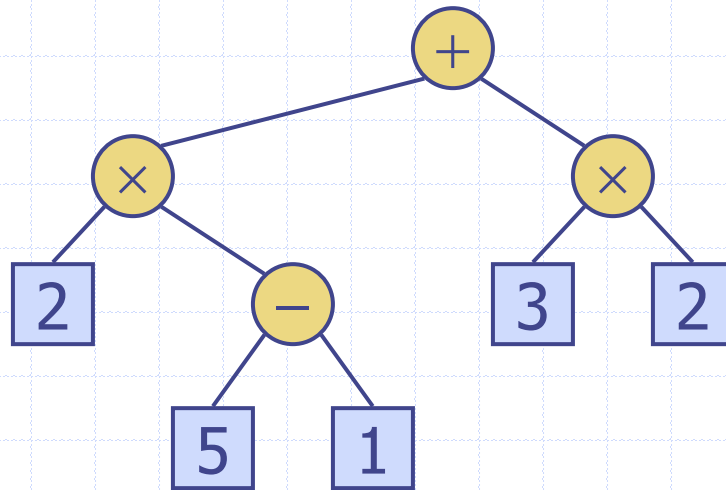
else

x ← *evalExpr*(*left*(v))

y ← *evalExpr*(*right*(v))

◇ ← operator stored at v

return *x* ◇ *y*



Euler Tour Traversal

- Not every application strictly fits the mold of a preorder, postorder, or inorder traversal.
- We can unify the tree-traversal algorithms into a single framework known as an ***Euler tour traversal***.
- The Euler tour traversal of a tree T can be informally defined as a “walk” around T , where we start by going from the root toward its leftmost child, viewing the edges of T as being “walls” that we always keep to our left.

Time Complexity of Euler Tour Traversal

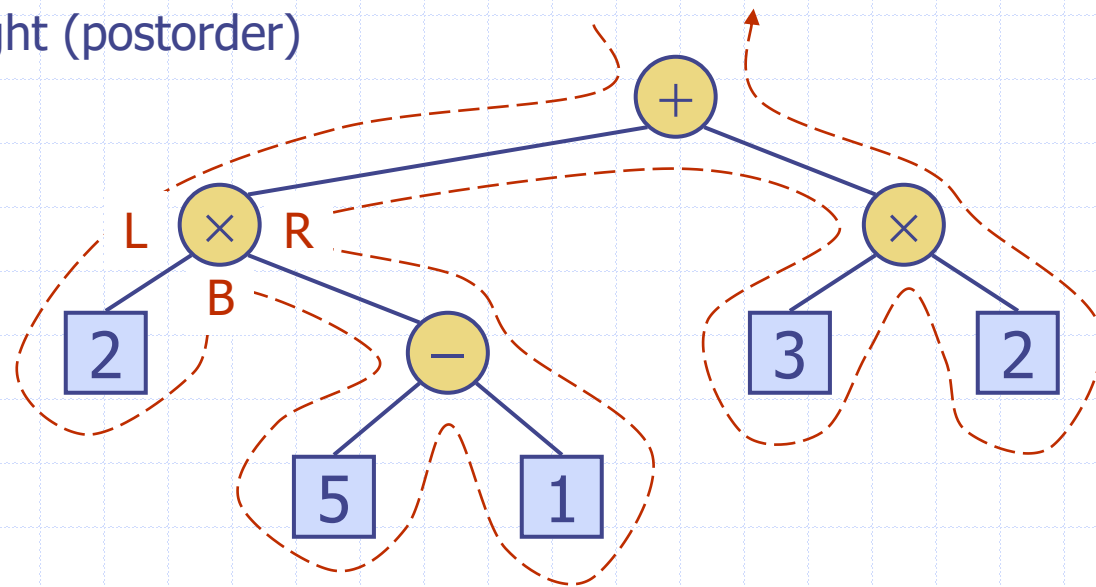
- The complexity of the walk is $O(n)$, for a tree with n nodes, because it progresses exactly two times along each of the $n-1$ edges of the tree—once going downward along the edge, and later going upward along the edge.

Euler Tour Traversal

- The process of an Euler tour can be naturally viewed as recursive.
- In between the “pre visit” and “post visit” of a given position will be a recursive tour of each of its subtrees.
- In the special case of a binary tree, we can designate the time when the walk passes immediately *below* a node as an “in visit” event. This will be just after the tour of its left subtree (if any), but before the tour of its right subtree (if any).

Euler Tour Traversal

- ❑ Generic traversal of a binary tree
- ❑ Includes a special cases the preorder, postorder and inorder traversals
- ❑ Walk around the tree and visit each node three times:
 - on the left (preorder)
 - from below (inorder)
 - on the right (postorder)



- ```

Algorithm eulerTour(T, p):
 perform the “pre visit” action for position p
 for each child c in $T.children(p)$ do
 eulerTour(T, c) { recursively tour the subtree rooted at c }
 perform the “post visit” action for position p

```

# Euler Tour Traversal

- The Euler tour traversal extends the preorder and postorder traversals, but it can also perform other kinds of traversals.
- For example, to compute the number of descendants of each position  $p$  in an  $n$ -node binary tree. We start an Euler tour by initializing a counter to 0, and then increment the counter during the “pre visit” for each position.
- To determine the number of descendants of a position  $p$ , we compute the difference between the values of the counter from when the pre-visit occurs and when the post-visit occurs, and add 1 (for  $p$ ).

# Euler Tour Traversal

- This simple rule gives us the number of descendants of  $p$ , because each node in the subtree rooted at  $p$  is counted between  $p$ 's visit on the left and  $p$ 's visit on the right.
- Therefore, we have an  $O(n)$ -time method for computing the number of descendants of each node.
- For the case of a binary tree, we can customize the algorithm to include an explicit “in visit” action, as shown in Code Fragment 8.29.

# Euler Tour Traversal

**Algorithm** eulerTourBinary( $T, p$ ):

perform the “pre visit” action for position  $p$

if  $p$  has a left child  $lc$  then

    eulerTourBinary( $T, lc$ )                      { recursively tour the left subtree of  $p$  }

perform the “in visit” action for position  $p$

if  $p$  has a right child  $rc$  then

    eulerTourBinary( $T, rc$ )                      { recursively tour the right subtree of  $p$  }

perform the “post visit” action for position  $p$

**Code Fragment 8.29:** Algorithm eulerTourBinary for performing an Euler tour traversal of a subtree rooted at position  $p$  of a binary tree.