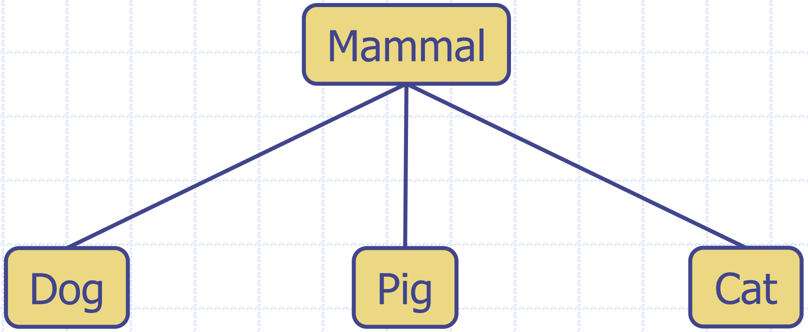
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Trees



Trees

- General Trees
- Binary Trees
- Implementing Trees
- Tree Traversal Algorithms

Lists and Iterators - Review

- Lists represent a linear sequence of elements, with **more general support for adding or removing elements at arbitrary positions**.
- Java defines a general interface, java.util.**List**, that includes also index-based methods
- An array list can be implemented using arrays
  - **Add** and **remove** operations run in **O(n)** time

Dynamic Array-based Array List

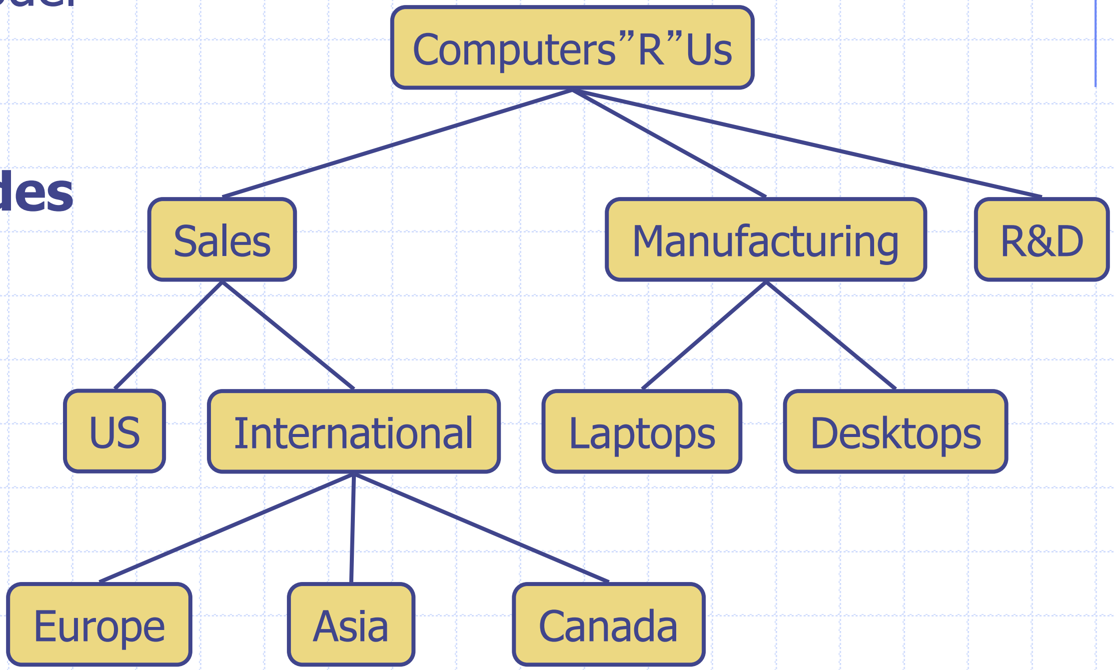
- A **dynamic array list** can be also implemented using arrays
- When the array is full, we replace the array with a larger one
  - Incremental strategy: increase the size by a constant c
    - The amortized time of a push operation is $O(n)$
  - Doubling strategy: double the size
    - The amortized time of a push operation is $O(1)$

Positional Lists

- **Positional list** ADT uses a **position element that acts as a marker** or token within the broader positional list.
- A position **p** is **unaffected by changes elsewhere in a list.**
- A position instance *p* is a simple object, supporting only ***getElement()*** method returning the element stored at position *p*.
- Use a **doubly linked list to implement a positional list**

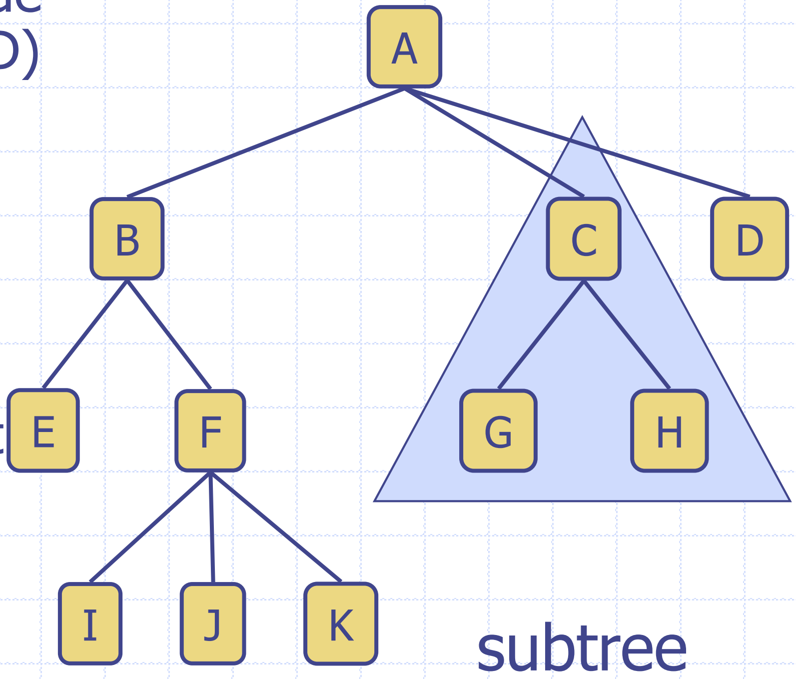
What is a Tree

- 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



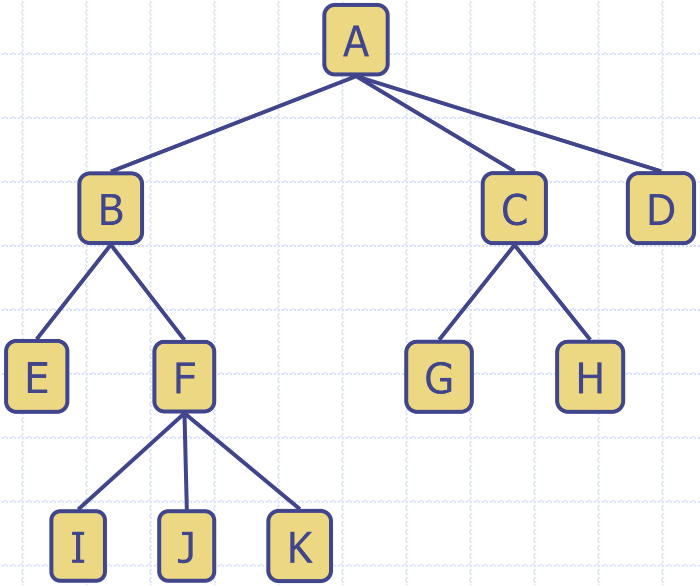
Tree Terminology

- **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



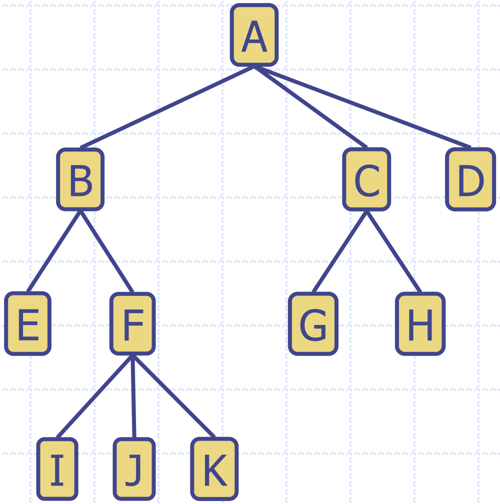
Tree Terminology

- **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. Example: (C,G)
  - A ***path*** of *T* is a sequence of nodes such that any two consecutive nodes in the sequence form an edge. Example (B, F, J).

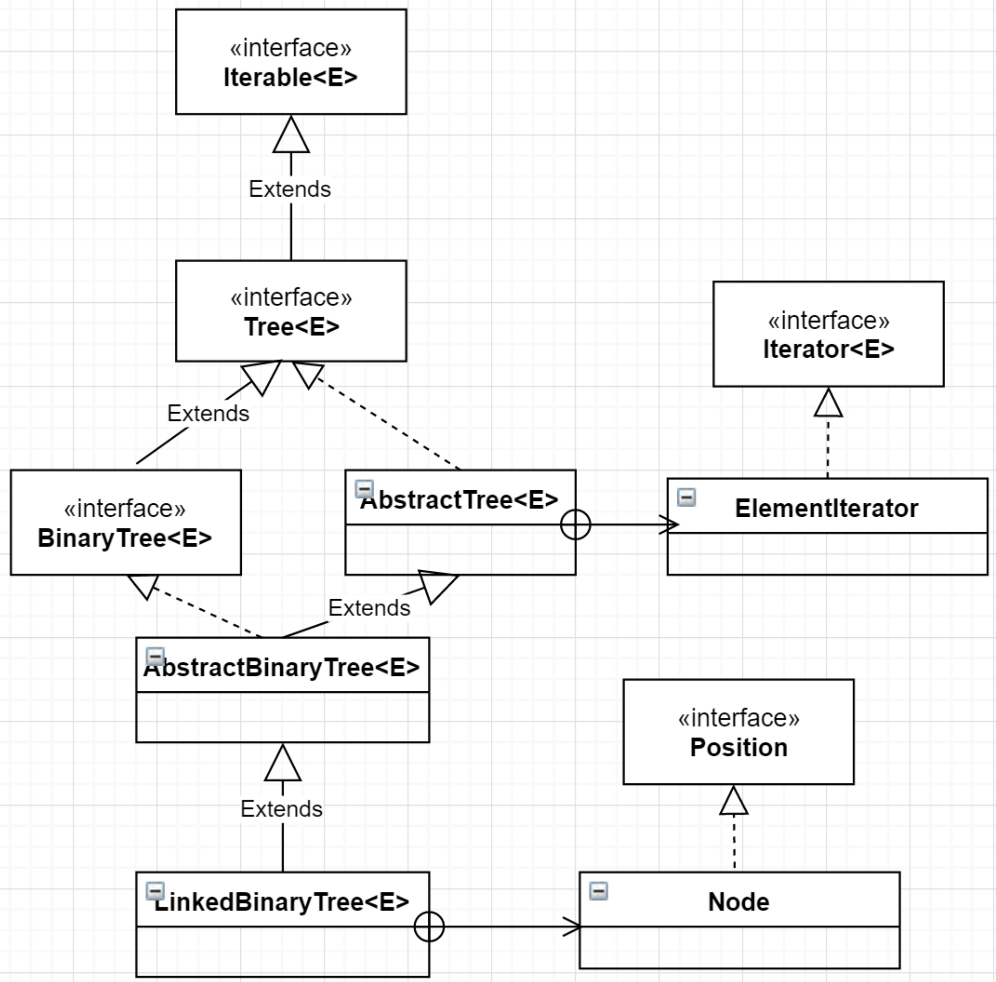


Tree Terminology

- **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.



Tree ADT



«interface»
Iterable<E>
Extends
«interface»
Tree<E>
«interface»
Iterator<E>
Extends
«interface»
BinaryTree<E>
AbstractTree<E>
ElementIterator
Extends
AbstractBinaryTree<E>
«interface»
Position
Extends
LinkedBinaryTree<E>
Node

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:

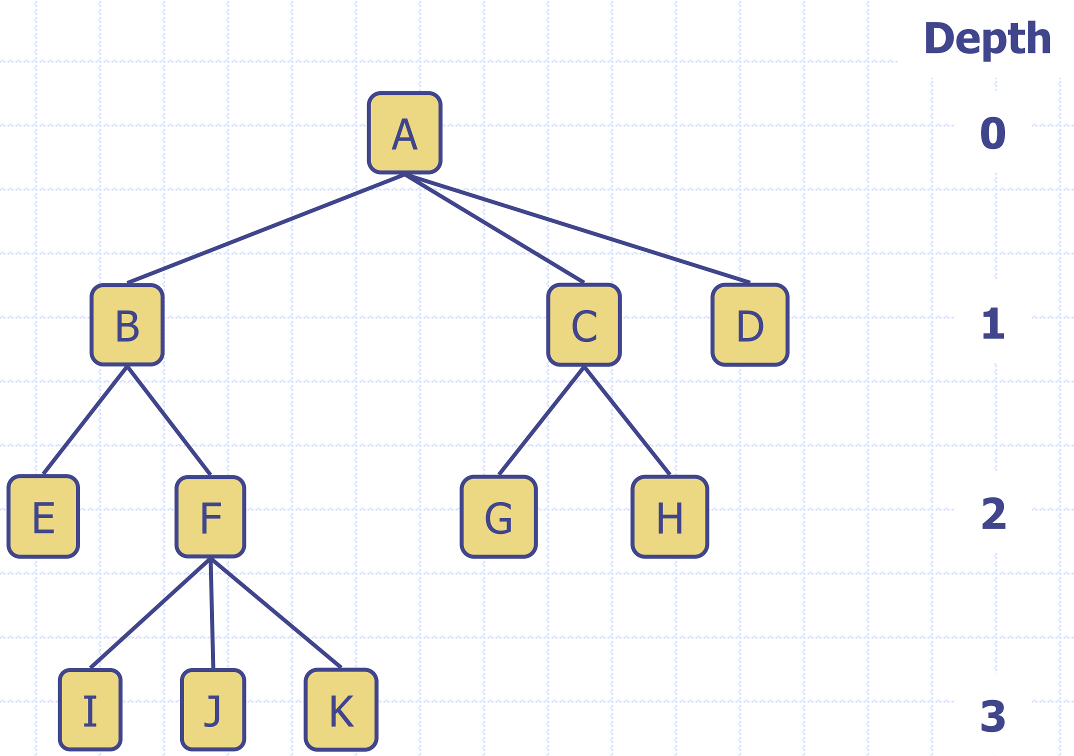
```
/** An interface for a tree where nodes can have an arbitrary number of children. */
public interface Tree<E> extends Iterable<E> {
  Position<E> root( );
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p)
                                      throws IllegalArgumentException;
  int numChildren(Position<E> p) throws IllegalArgumentException;
  boolean isInternal(Position<E> p) throws IllegalArgumentException;
  boolean isExternal(Position<E> p) throws IllegalArgumentException;
  boolean isRoot(Position<E> p) throws IllegalArgumentException;
  int size( );
  boolean isEmpty( );
  Iterator<E> iterator( );
  Iterable<Position<E>> positions( );
}
```

An AbstractTree Base Class in Java

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

Computing depth

- Let *p* be a position within tree *T*. The ***depth*** of *p* is the number of ancestors of *p*, other than *p* itself.



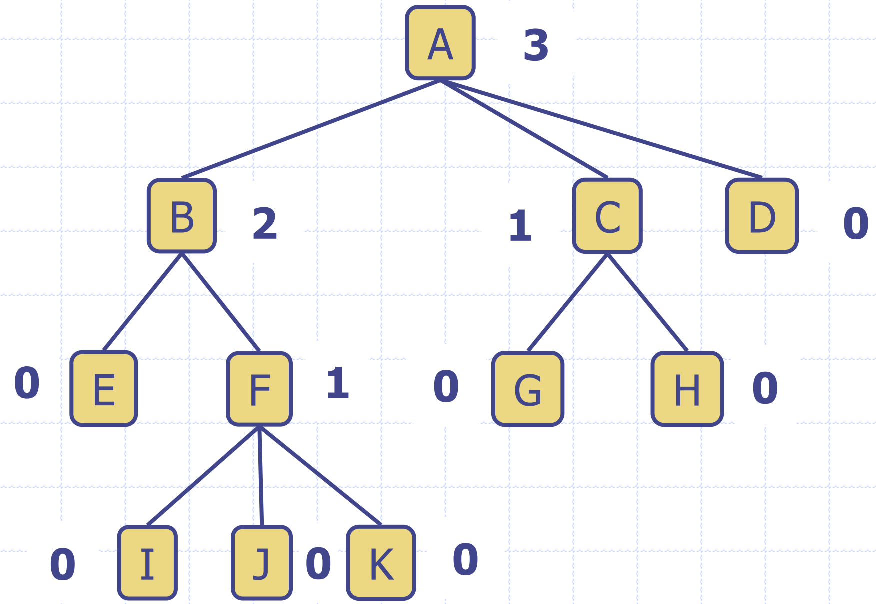
Computing depth

- 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*.

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

Computing height

- 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**.



Computing height

- The following method computes the height of a subtree rooted at position *p* using the recursive definition.
- The base case is would be when p is an external position, hence the height is 0:

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

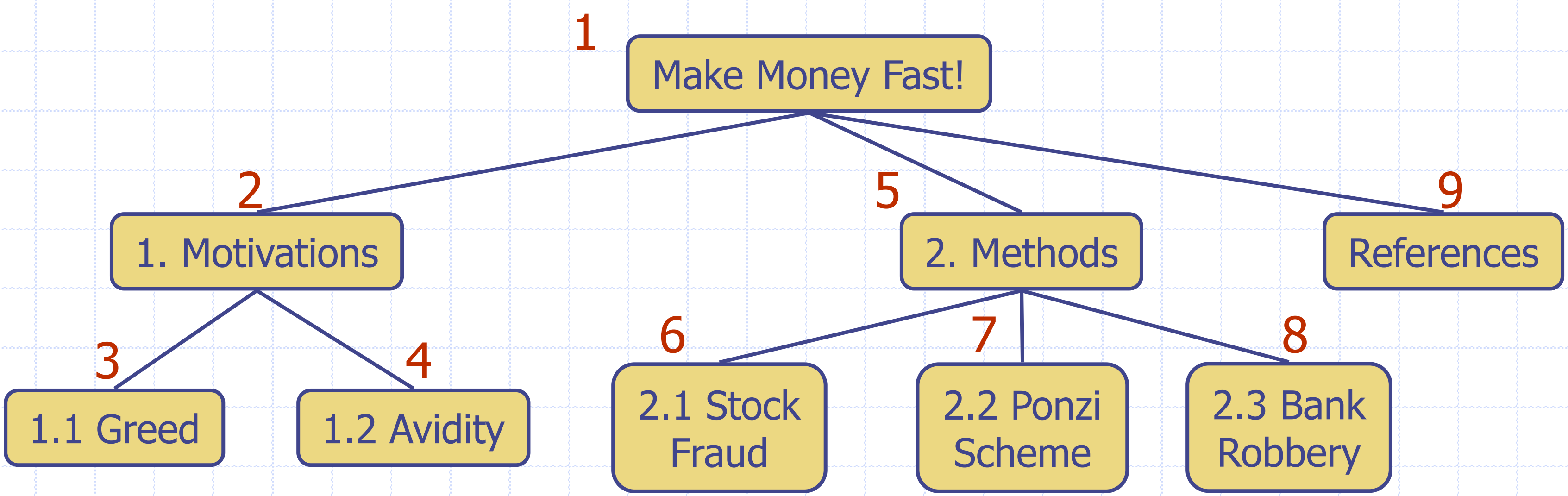
Tree Traversal

- A ***traversal*** of a tree *T* is a systematic way of accessing, or "visiting," all the positions of *T*
- 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.
- In **postorder traversal** of a tree it recursively **traverses the subtrees rooted at the children of the root first**, and **then visits the root**.
  - In some sense, this algorithm can be viewed as the opposite of the preorder traversal(hence, the name "postorder")

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

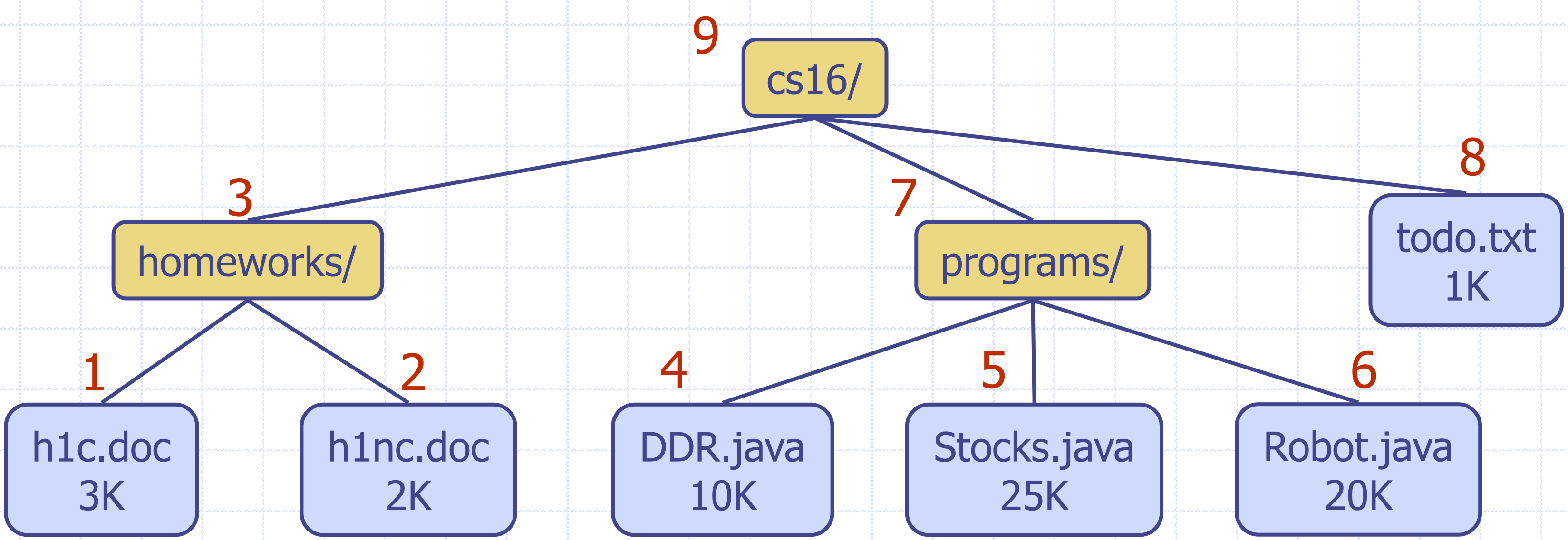
Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)



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

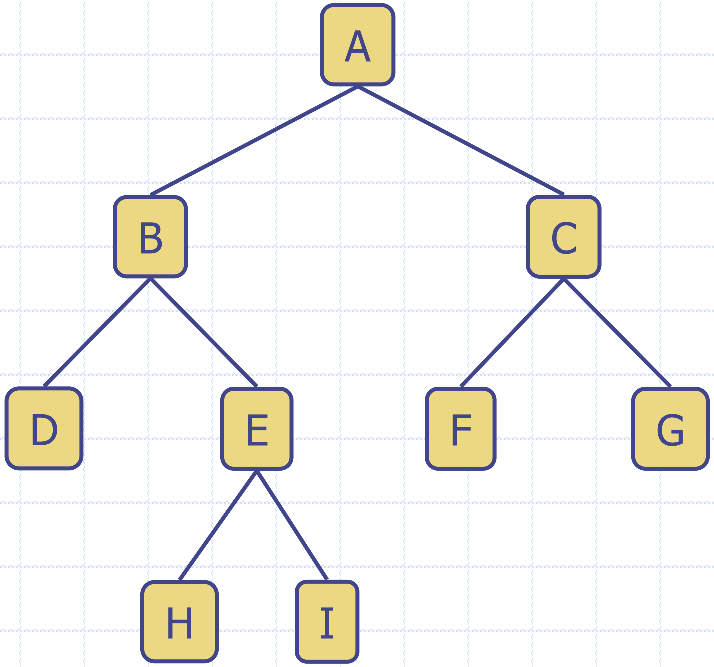
Algorithm *postOrder*(*v*)
    for each child *w* of *v*
        postOrder (*w*)
    visit(*v*)



Binary Trees

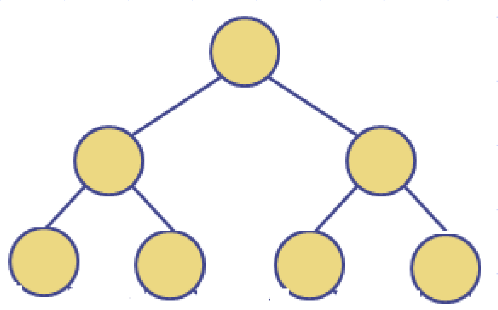
- A **binary tree** is an ordered tree with the following properties:
  - Each internal node has **at most two children** (exactly two for proper binary trees).
  - Each child node is labeled as being either a **left child** or a **right child**.
  - A **left child precedes a right child** in the order of children of a node.
- The subtree rooted at a left or right child of an internal node *v* is called a **left subtree** or **right subtree**, respectively, of *v*.

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



Binary Trees

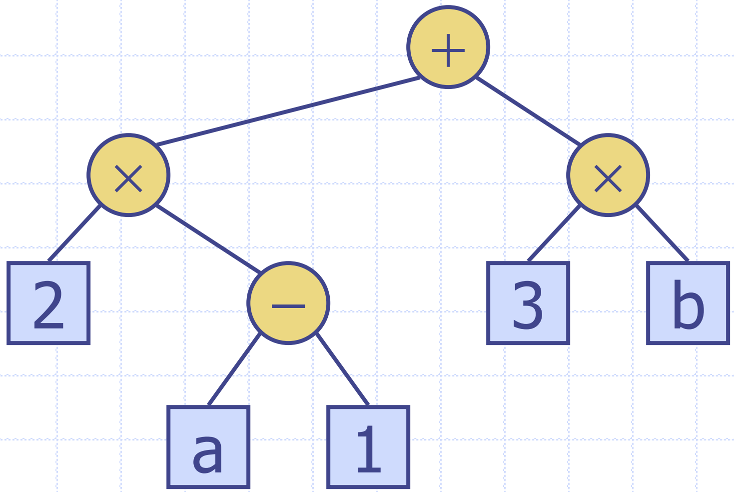
- Alternative **recursive definition**: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree
- A binary tree is ***proper*** if each node has either zero or two children.
  - Some people also refer to such trees as being ***full*** binary trees.
  - In a proper binary tree, every internal node has exactly two children.



- A binary tree that is not proper is ***improper***.

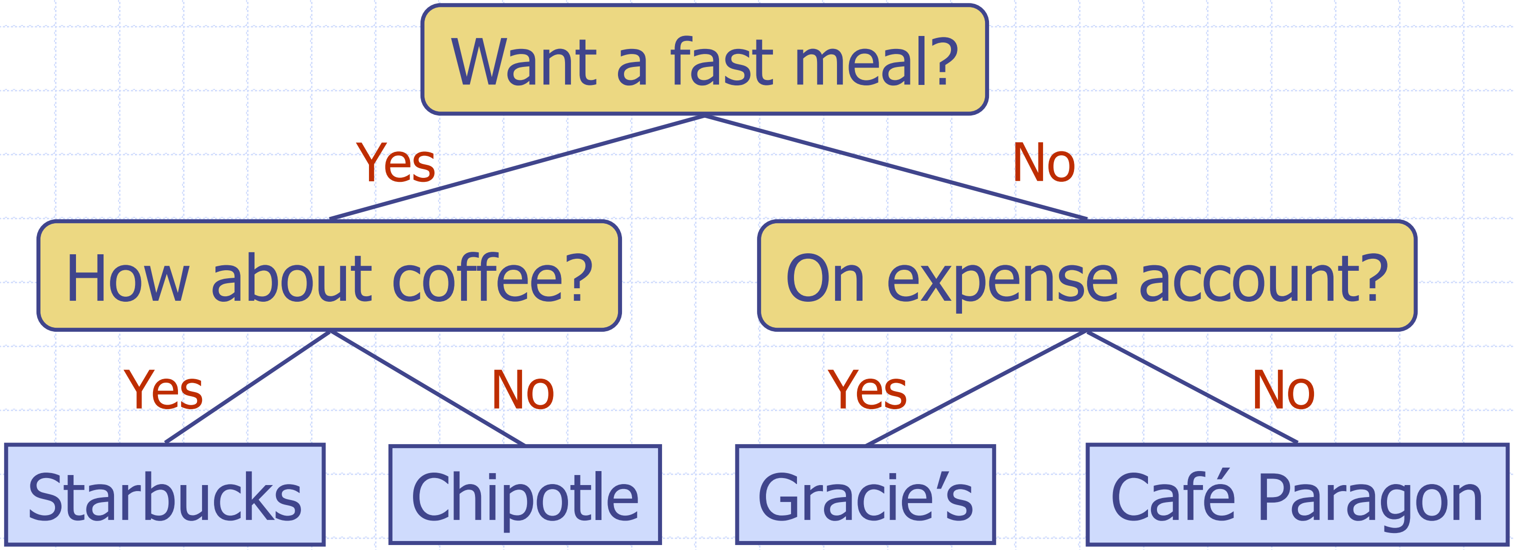
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



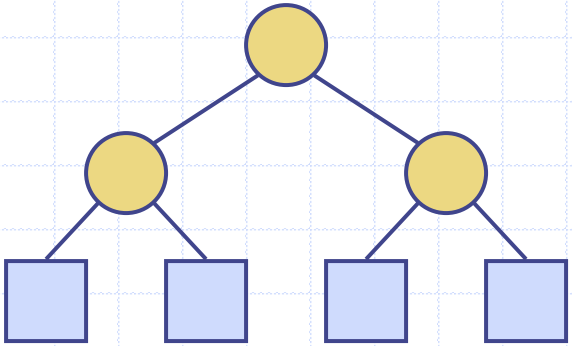
Decision Tree

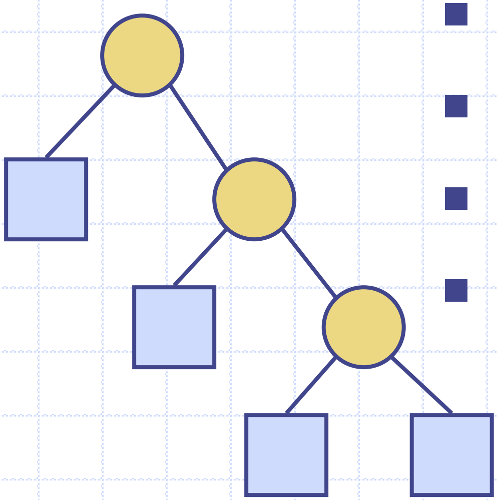
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Properties of Proper Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height





- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

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

Defining a BinaryTree Interface

- **Extends the Tree interface** to **add the three new behaviors.**
  - In this way, a binary tree is expected to support all the functionality that was defined for general trees

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

Defining an AbstractBinaryTree Base Class

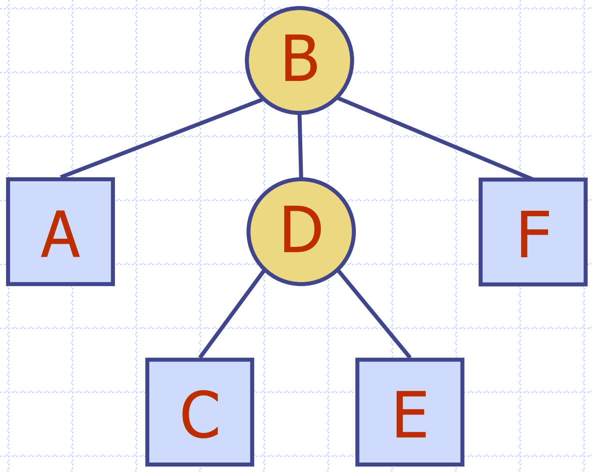
```
/** An abstract base class providing some functionality of the BinaryTree interface.*/
public abstract class AbstractBinaryTree<E> extends AbstractTree<E>
                                            implements BinaryTree<E> {
  /** Returns the Position of p's sibling (or null if no sibling exists). */
  public Position<E> sibling(Position<E> p) {
    Position<E> parent = parent(p);
    if (parent == null) return null;                  // p must be the root
    if (p == left(parent))                            // p is a left child
      return right(parent);                           // (right child might be null)
    else                                              // p is a right child
      return left(parent);                            // (left child might be null)
  }
  /** Returns the number of children of Position p. */
  public int numChildren(Position<E> p) {
    int count=0;
    if (left(p) != null)
      count++;
    if (right(p) != null)
      count++;
    return count;
  }
  /** Returns an iterable collection of the Positions representing p's children. */
  public Iterable<Position<E>> children(Position<E> p) {
    List<Position<E>> snapshot = new ArrayList<>(2);    // max capacity of 2
    if (left(p) != null)
      snapshot.add(left(p));
    if (right(p) != null)
      snapshot.add(right(p));
    return snapshot;
  }
}
```

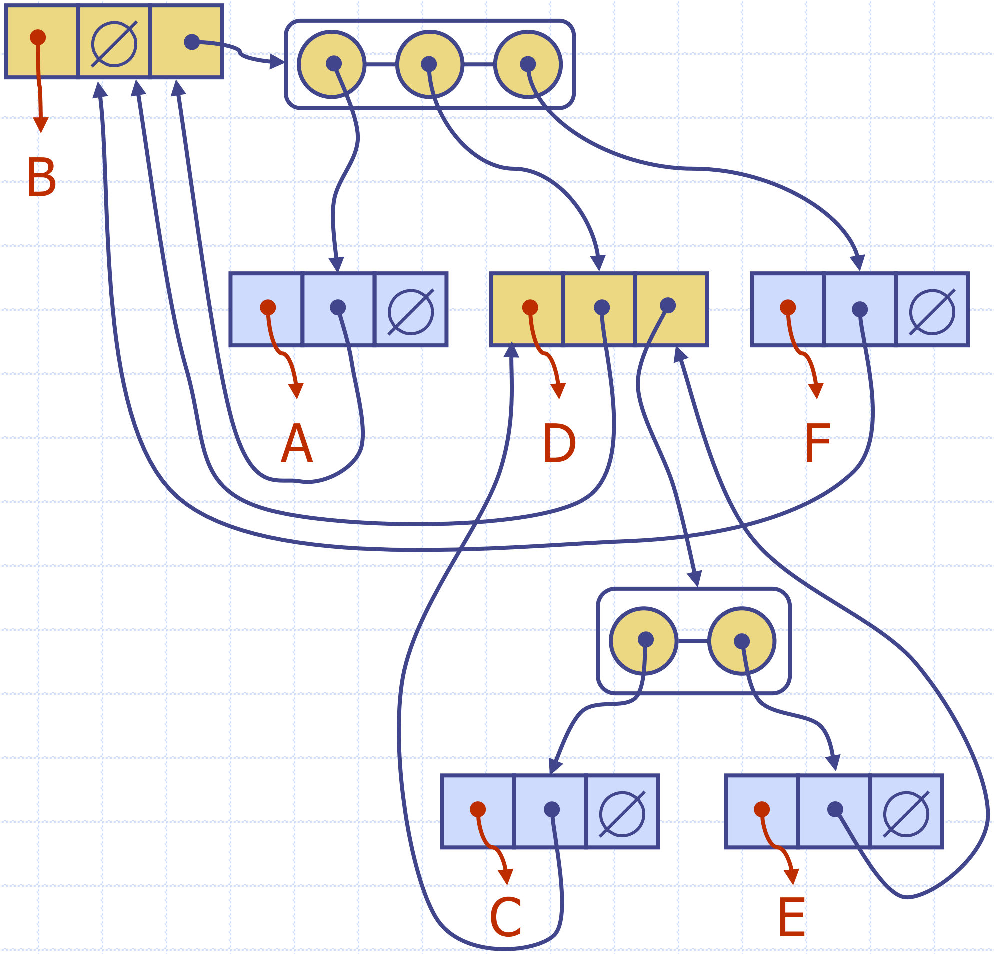
Implementing Trees

- A natural way to realize a **binary tree** ***T*** is to use a ***linked structure***, with a node that maintains references to the element stored at a position *p* and to the nodes associated with the children and parent of *p*.
  - If p is the root of T, then the parent field of p is null.
  - Likewise, if p does not have a left child (respectively, right child), the associated field is null.
- The tree itself maintains an instance variable, called *root*, storing a reference to the root node (if any), and a variable, called *size*, that represents the overall number of nodes of T.

Linked Structure for General Trees

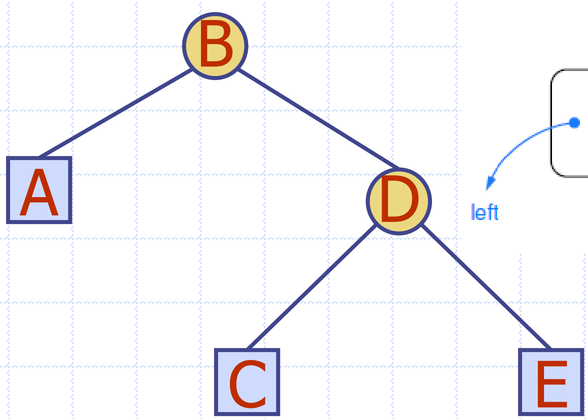
- A node is represented by an object storing
  - **Element**
  - **Parent** node
  - **Sequence of children** nodes
- Node objects implement the Position ADT

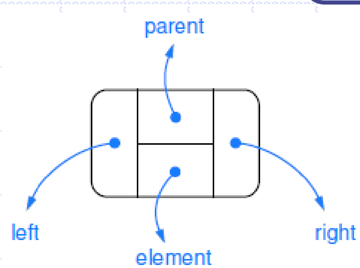


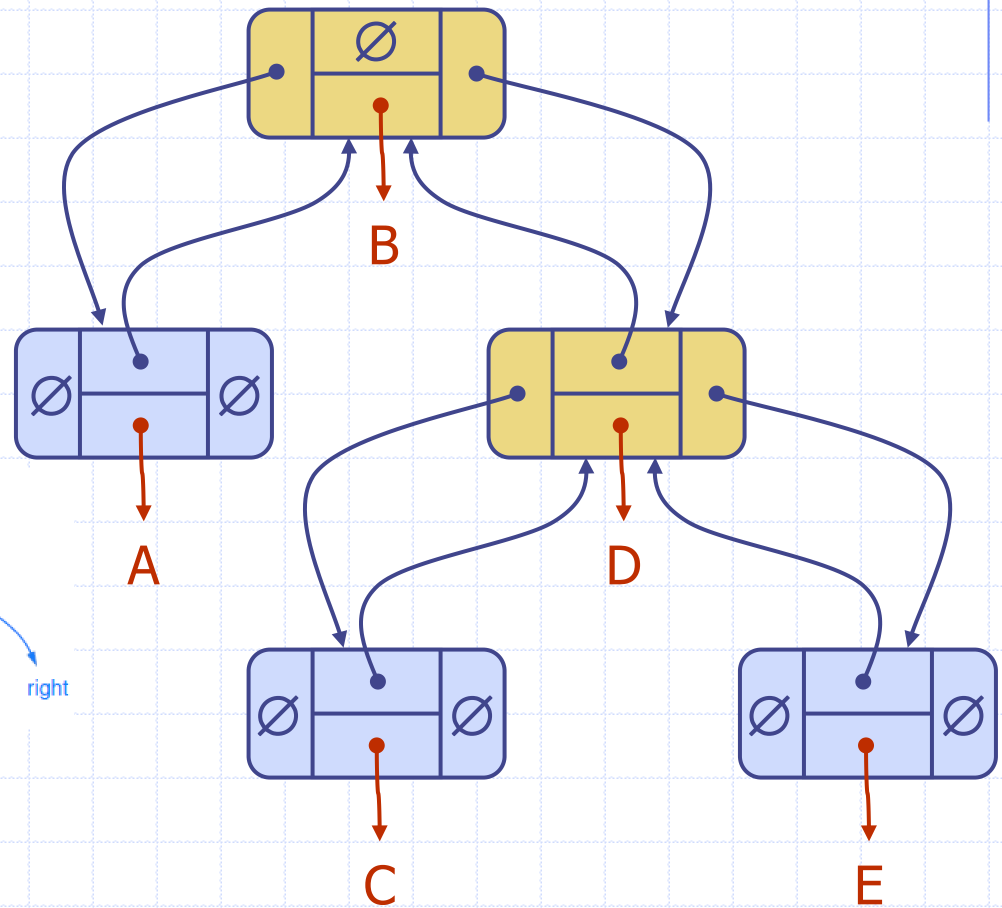


Linked Structure for Binary Trees

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







Java Implementation

- **LinkedBinaryTree** class:
  - Inner **Node** class, which implements the **Position** interface.
    - Instance variables: **element**, ***parent***, ***left***, ***right***
    - **createNode** method that returns a new node instance (factory pattern).
  - Two instance variables: ***root*** and ***size***
  - The **validate(p)** method, followed by the accessors **size**, **root**, **left**, and **right**.
  - six update methods for a linked binary tree:,
    - **addRoot**, **addLeft**, **addRight, set, attach, remove**

Java Implementation

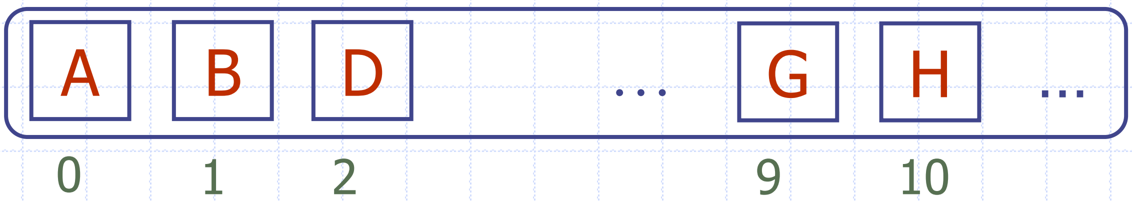
- Running times of the LinkedBinaryTree methods, including derived methods that are inherited from the AbstractTree and AbstractBinaryTree classes:

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 $O(n)$.

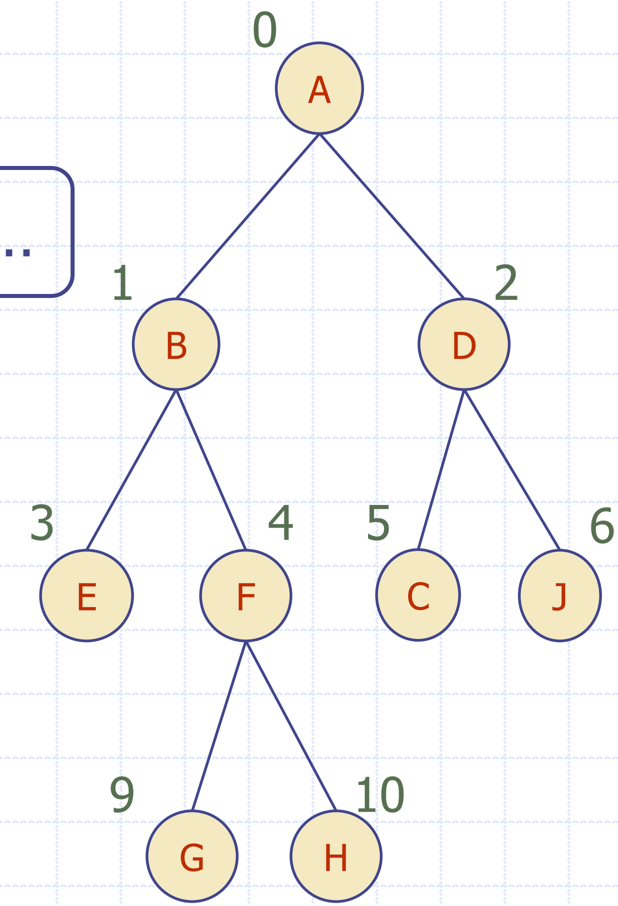
Array-Based Representation of Binary Trees

- Nodes are stored in an array A



Node v is stored at A[rank(v)], where:

- rank(root) = 0
- if node is the left child of parent(node),
  rank(node) = 2 · rank(parent(node)) + 1
- if node is the right child of parent(node),
  rank(node) = 2· rank(parent(node)) + 2



Preorder Tree Traversal Algorithm

- A traversal of a tree T is a systematic way of accessing, or “visiting,” all the positions of T.
- In the tree below the **preorder** algorithm:
  - Visits the **root**
  - Recursively traverses the **subtree for each child of root**

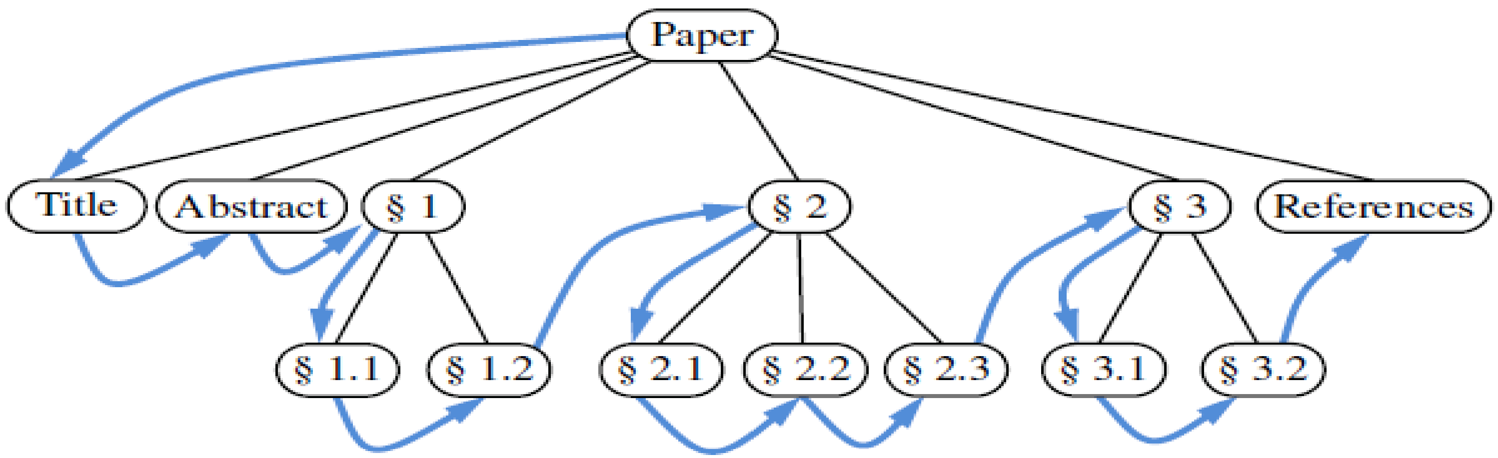


Figure 8.13: Preorder traversal of an ordered tree, where the children of each position are ordered from left to right.

Preorder Tree Traversal Algorithm

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 }

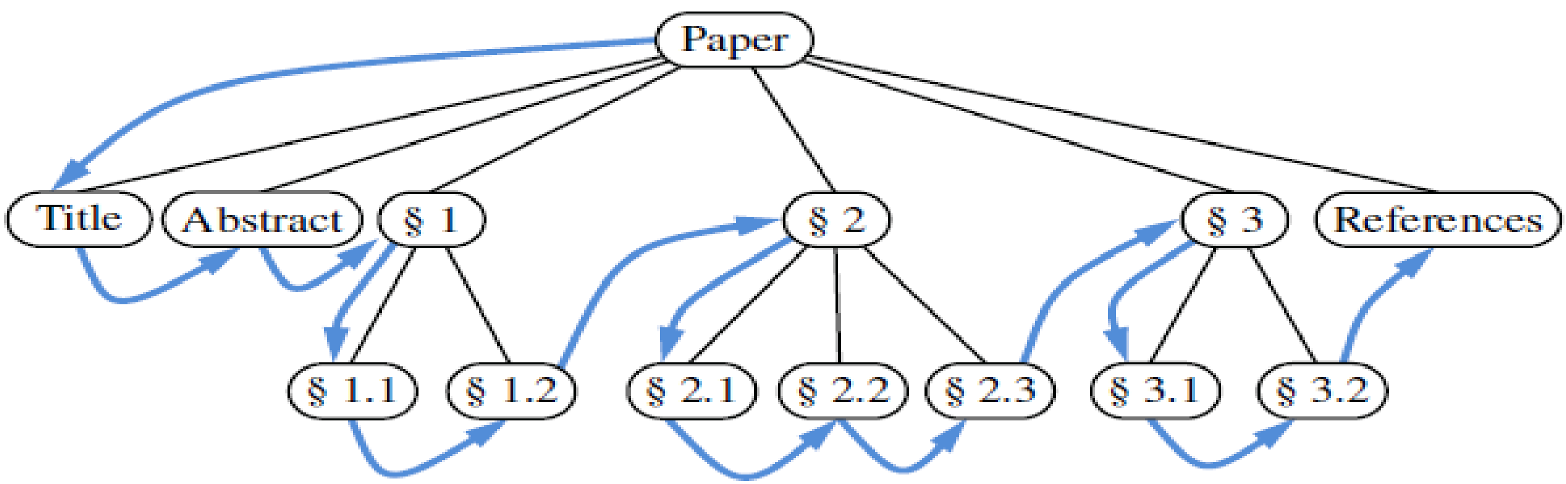


Figure 8.13: Preorder traversal of an ordered tree, where the children of each position are ordered from left to right.

Postorder Tree Traversal Algorithm

- In a **postorder traversal**, a node is visited after its descendants:

```
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.

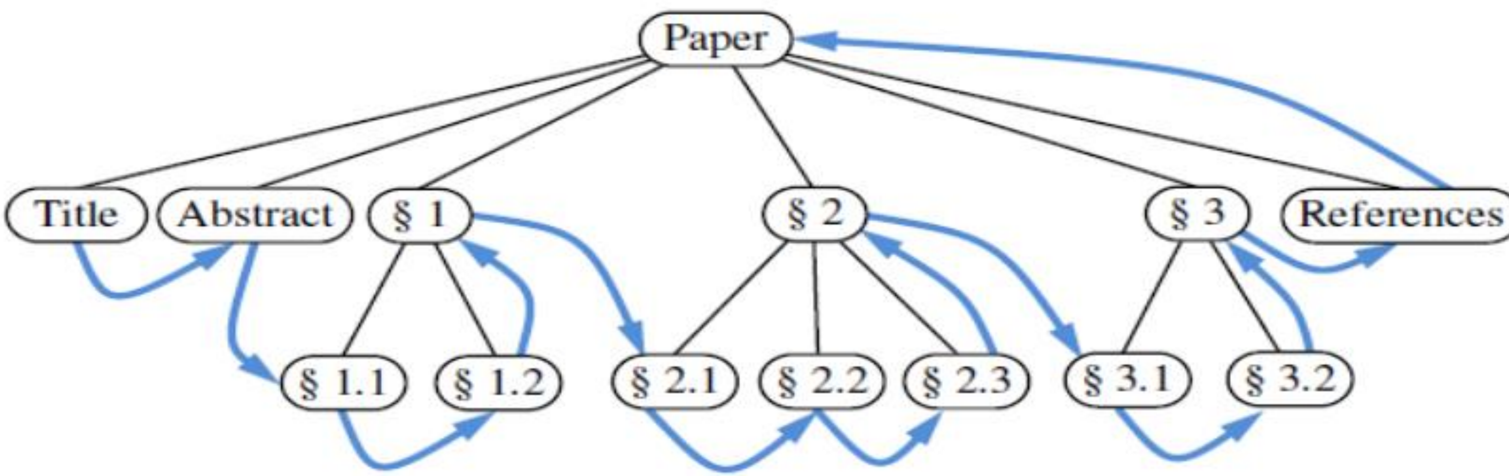
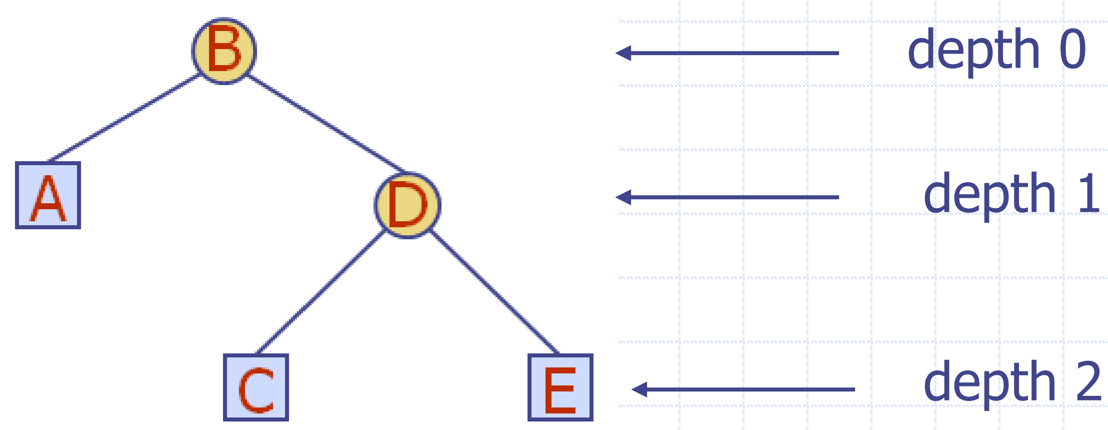


Figure 8.14: Postorder traversal of the ordered tree of Figure 8.13.

Breadth-First Tree Traversal Algorithm

- A **breadth-first** traversal is a common approach used in software for playing games.
  - traverse a tree so that we visit all the positions at depth **d** before we visit the positions at depth **d+1**.



Breadth-First Tree Traversal Algorithm

- 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.

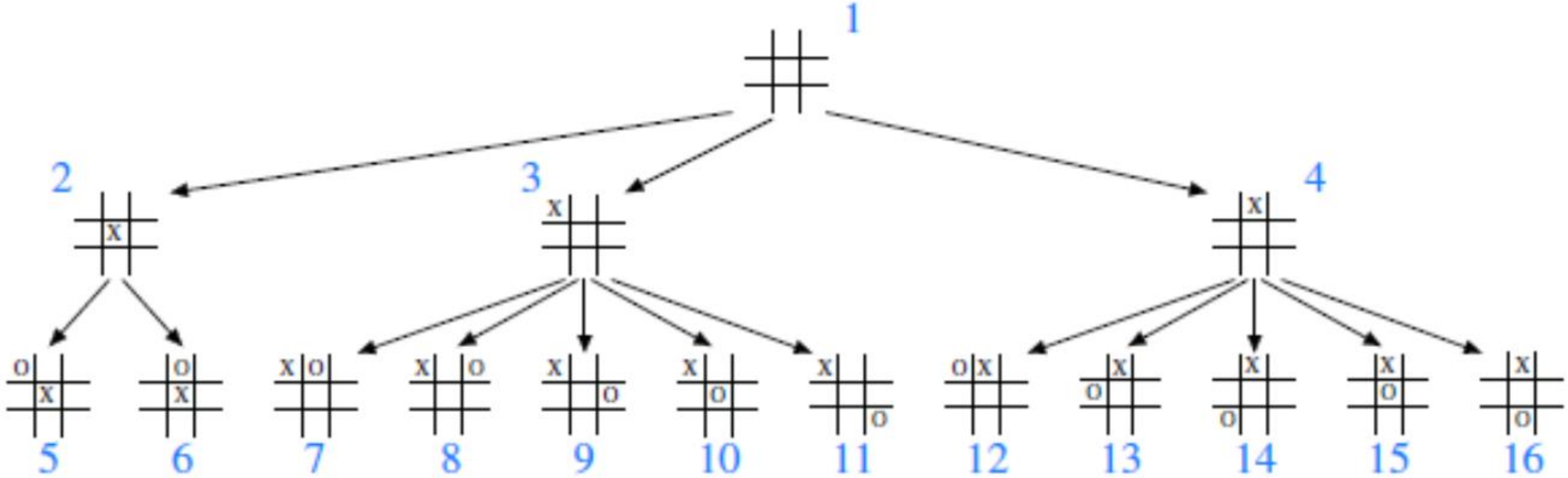
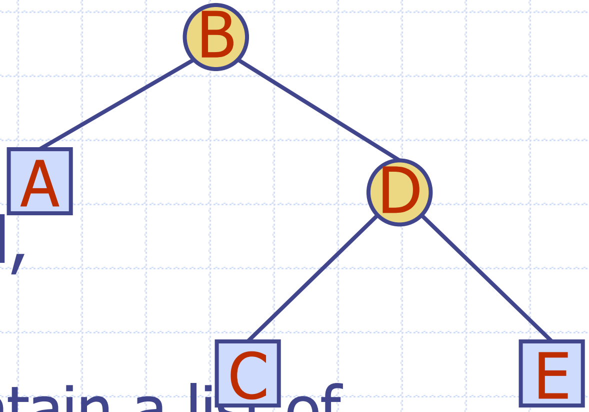


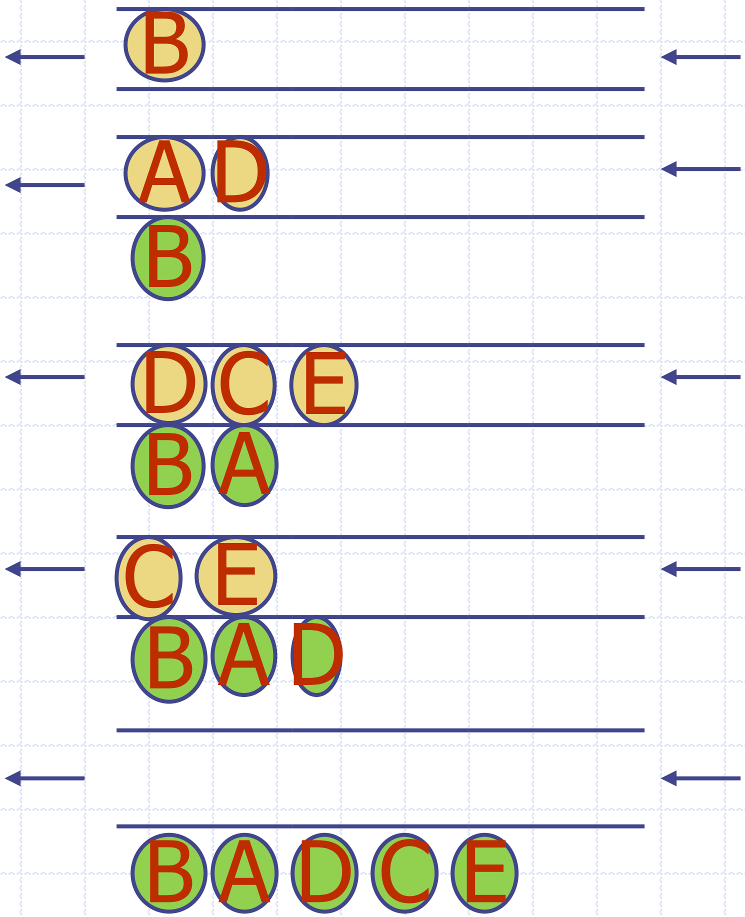
Figure 8.15: Partial game tree for Tic-Tac-Toe when ignoring symmetries; annotations denote the order in which positions are visited in a breadth-first tree traversal.

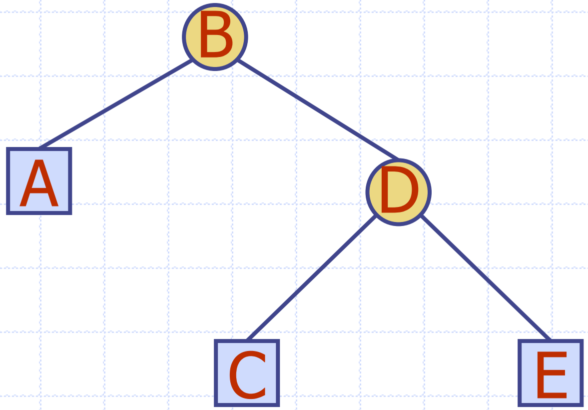
Breadth-First Tree Traversal Algorithm



- Visit the root (perform an action)
- Visit the left child
- Next, we need to visit the right child,
  But there is no link from A to D!
- So, for each node, we need to maintain a list of references to children nodes, known as discovered nodes, not visited yet.
- We can remove the node, then process its children
- A **queue** is a FIFO data structure and can do that.
  - Just initialize the queue to contain the root.

Breadth-First Tree Traversal Algorithm





1. visit the node
2. add references to its children (enqueue)
3. remove the node (dequeue)

Breadth-First Tree Traversal Algorithm

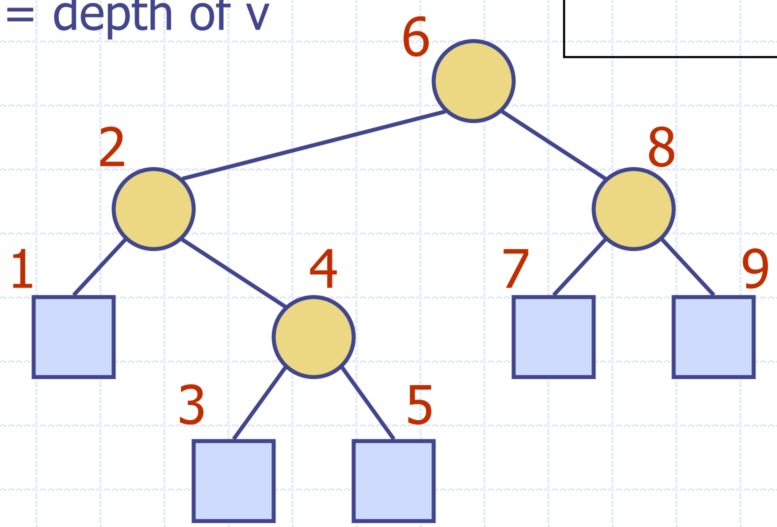
- We use a queue to produce a FIFO (i.e., first-in first-out) semantics for the order in which we visit nodes.
- The overall running time is O(n), due to the **n** calls to *enqueue* and **n** calls to *dequeue*.

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

- 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))

Inorder Traversal of a Binary Tree

- During an **inorder** traversal, we visit a position between the recursive traversals of its left and right subtrees.

Algorithm inorder(p):
  if p has a left child lc **then**
    inorder(lc) { recursively traverse the left subtree of p }
  perform the "visit" action for position p
  if p has a right child rc **then**
    inorder(rc) { recursively traverse the right subtree of p }

Code Fragment 8.15: Algorithm inorder for performing an inorder traversal of a subtree rooted at position p of a binary tree.

Inorder Traversal of a Binary Tree

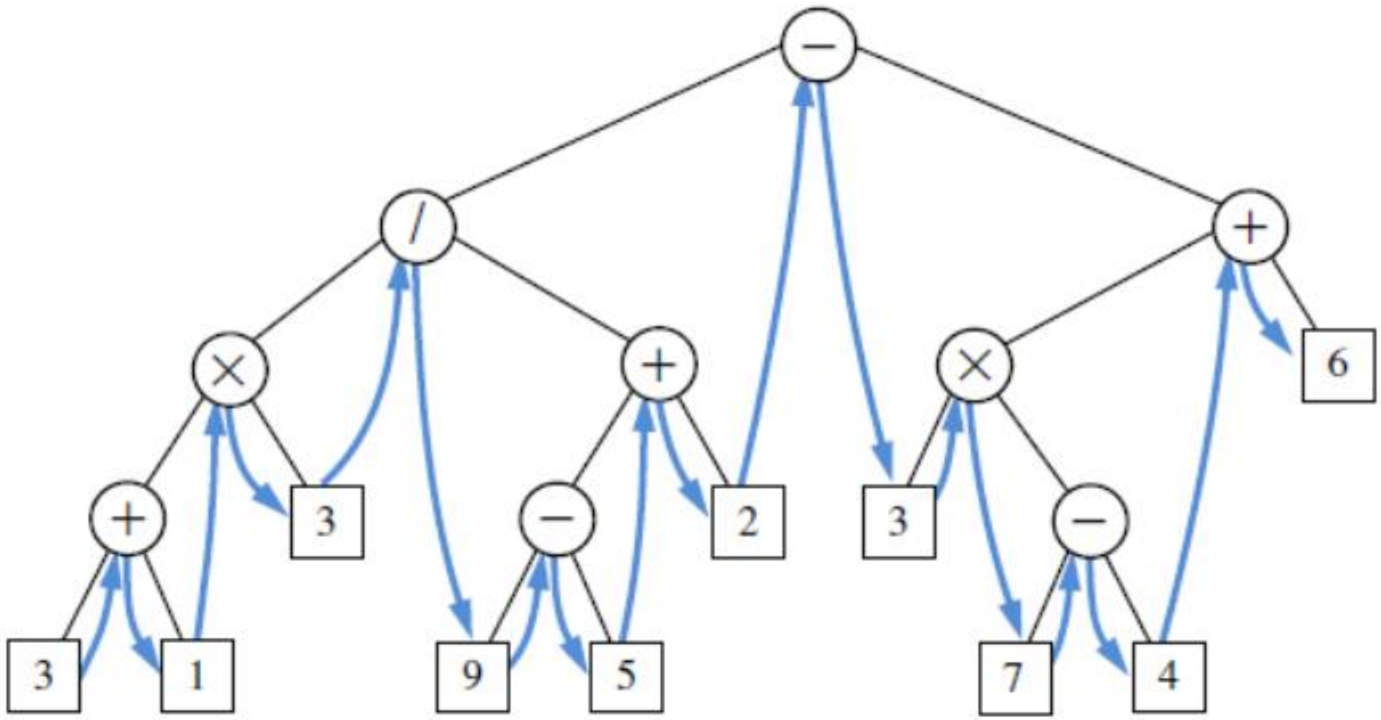


Figure 8.16: Inorder traversal of a binary tree.

Using a binary tree to represent an arithmetic expression:
3+1×3/9 −5 + 2 - 3x7 - 4 + 6.

Implementing Tree Traversals in Java - *preorder*

- iterator( ): Returns an iterator for all elements in the tree.
- positions( ): Returns an iterable collection of all positions of the tree.

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

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

Implementing Tree Traversals in Java - *postorder*

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

Implementing Tree Traversals in Java - *breadthfirst*

```
/** Returns an iterable collection of positions of the tree in breadth-first order. */
public Iterable<Position<E>> breadthfirst( ) {
  List<Position<E>> snapshot = new ArrayList<>( );
  if (!isEmpty( )) {
    Queue<Position<E>> fringe = new LinkedQueue<>( );
    fringe.enqueue(root( ));                          // start with the root
    while (!fringe.isEmpty( )) {
      Position<E> p = fringe.dequeue( );              // remove from front of the queue
      snapshot.add(p);                                // report this position
      for (Position<E> c : children(p))
        fringe.enqueue(c);                            // add children to back of queue
    }
  }
  return snapshot;
}
```

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.

Binary Search Trees

- A binary search tree 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 $e(p)$.

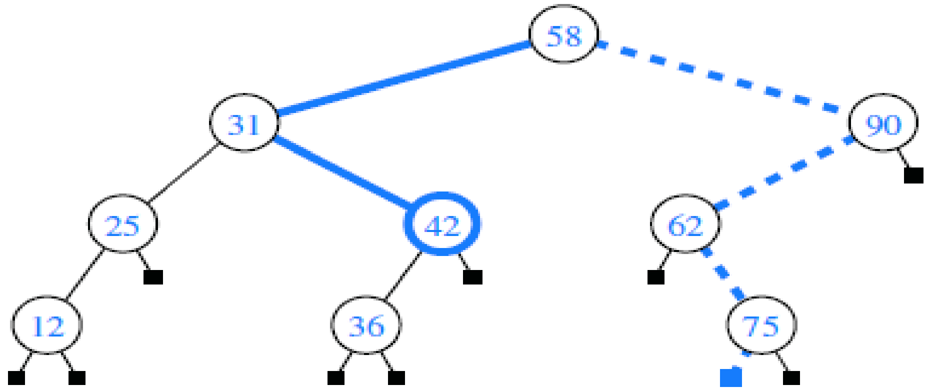
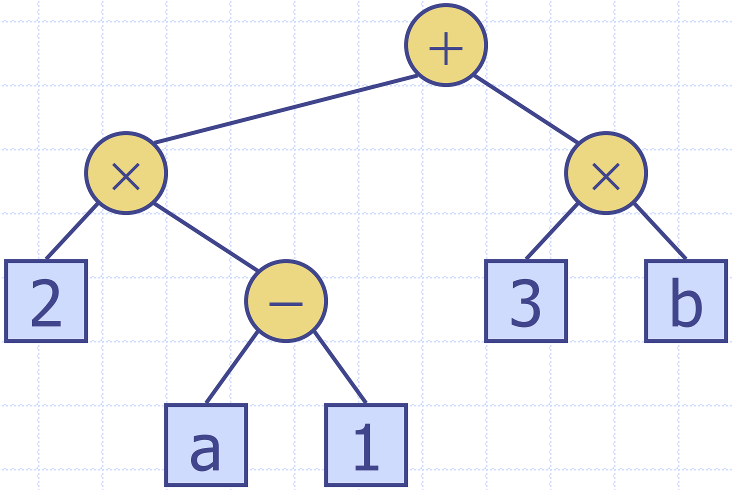


Figure 8.17: A binary search tree storing integers. The solid path is traversed when searching (successfully) for 42. The dashed path is traversed when searching (unsuccessfully) for 70.

Applications of Tree Traversals - 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

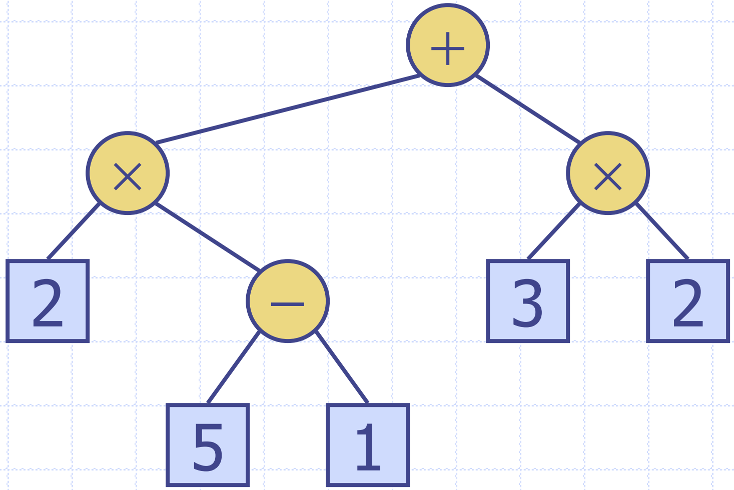


Algorithm *printExpression*(*v*)
  if *left* (*v*) ≠ **null**
    print("(")
    inOrder (*left*(*v*))
  print(*v.element* ())
  if *right*(*v*) ≠ **null**
    inOrder (*right*(*v*))
    print (")")

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

Applications of Tree Traversals - 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
        $x \leftarrow$ *evalExpr*(*left*(*v*))
        $y \leftarrow$ *evalExpr*(*right*(*v*))
        $\Diamond \leftarrow$ operator stored at ***v***
        return $x \Diamond y$

Applications of Tree Traversals – Euler Tour Traversal

- **Generic traversal** of a binary tree
- Includes 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**)