

Data Structures and Algorithms

Chapter 8

General Trees

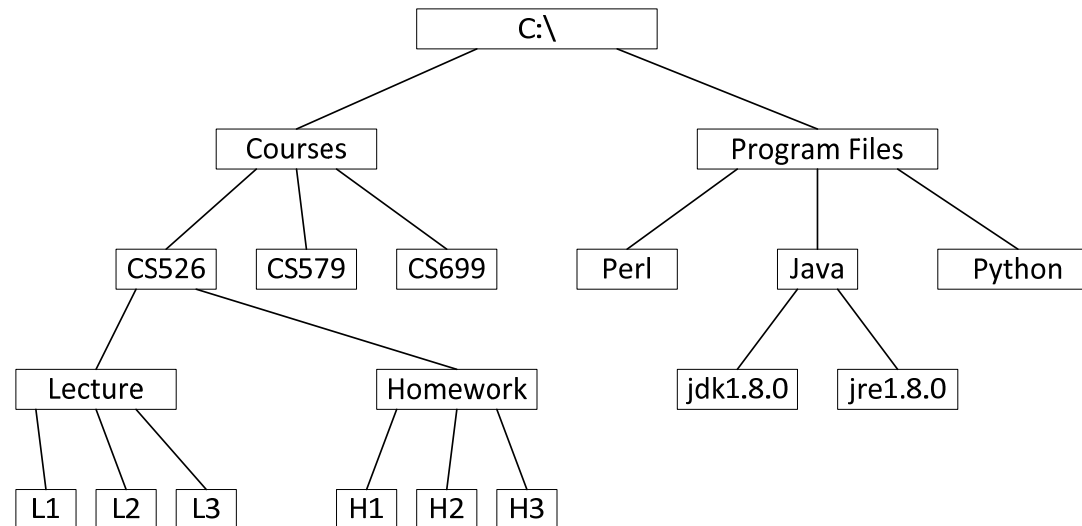
Basics

- A *graph* is a set of nodes and a set of edges.
- Formally, a graph $G = (V, E)$, where V is a set of nodes (or vertices) and E is a set of edges.
- Each edge connects two nodes, and is represented as (u, v) , where u and v are nodes.
- A ***tree*** is a connected, acyclic, undirected graph with a distinguished node called *root*.
- *Connected*: There is a path from every node to every other node.
- *Acyclic*: There is no cycle
- *Undirected*: Edges have no direction

General Trees

Basics

- Example

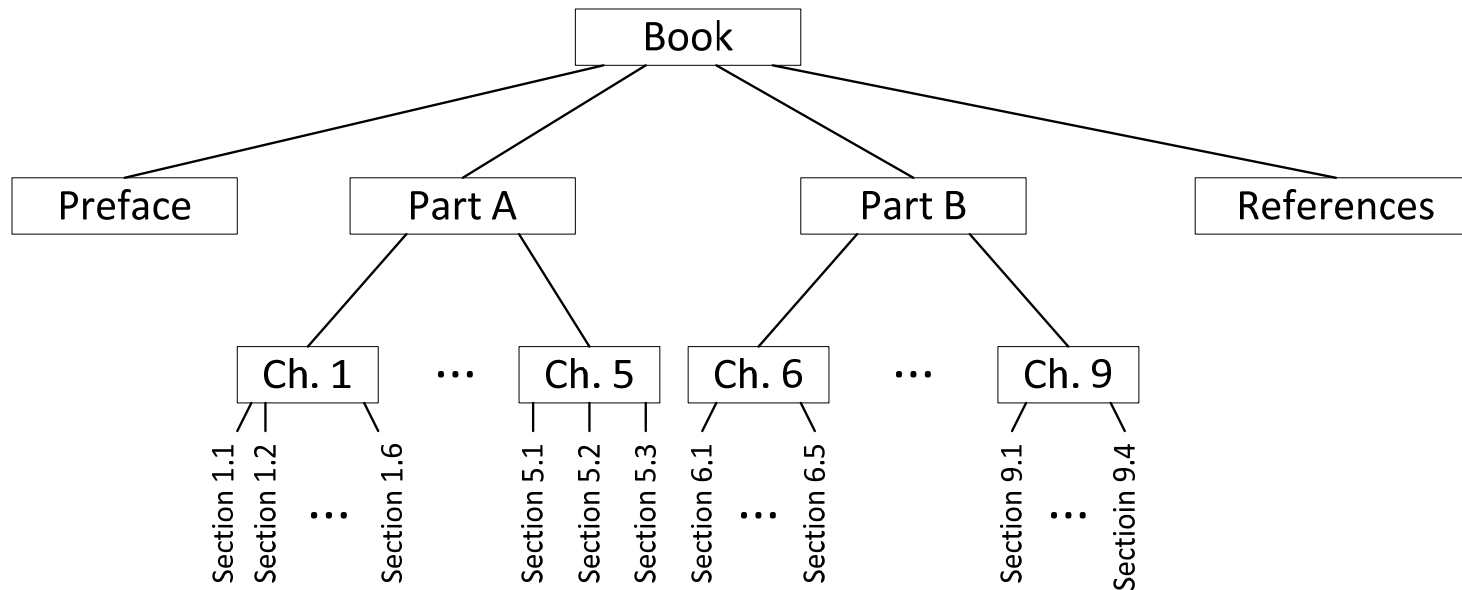


- Root, parent, child, siblings
- Internal node, external node (or leaf node)
- Ancestor, descendant
- Path

General Trees

Basics

- Ordered tree: There is meaningful ordering among siblings:



General Trees

Basics

- In the subsequent slides, “position” is used in many examples. You can consider it as a “node” in the data structure.
- The source programs that come with our textbook uses “position.”
- If you are required to write a program that uses our textbooks source code that uses “position,” I will give you a substitute code that does not use “position.”

General Trees

Tree ADT

- Accessor methods
 - `root()`: Returns the position of the root of the tree, or null if the tree is empty.
 - `parent(p)`: Returns the position of the parent of position p , or null if p is the root.
 - `children(p)`: Returns the children of position p , if any. If the tree is an ordered tree, children are ordered in the result.
 - `numChildren(p)`: Returns the number of children of position p .

General Trees

Tree ADT

- Query methods
 - `isInternal(p)`: Returns true if position p is an internal node.
 - `isExternal(p)`: Returns true if position p is an external node (or a leaf node).
 - `isRoot(p)`: Returns true if position p is the root of the tree.

General Trees

Tree ADT

- Other general methods
 - size(): Returns the number of positions (or the elements) in the tree.
 - isEmpty(): Returns true if the tree does not have any position (or element).
 - iterator(): Returns an iterator for all elements in the tree. So, the tree is *Iterable*.
 - positions(): Returns an iterable collection of all positions of the tree.

General Trees

Tree ADT

- Tree interface

```
1  public interface Tree<E> extends Iterable<E> {  
2      Position<E> root();  
3      Position<E> parent(Position<E> p) throws IllegalArgumentException;  
4      Iterable<Position<E>> children(Position<E> p)  
5          throws IllegalArgumentException;  
6      int numChildren(Position<E> p) throws IllegalArgumentException;  
7      boolean isInternal(Position<E> p) throws IllegalArgumentException;  
8      boolean isRoot(Position<E> p) throws IllegalArgumentException;  
9      int size();  
10     boolean isEmpty();  
11     Iterator<E> iterator();  
12     Iterable<Position<E>> positions();  
13 }
```

General Trees

Tree ADT

- AbstractTree abstract class

```
1  public abstract class AbstractTree<E> implements Tree<E> {  
  
2  public boolean isInternal(Position<E> p)  
    { return numChildren(p) > 0; }  
3  public boolean isExternal(Position<E> p)  
    { return numChildren(p) == 0; }  
4  public boolean isRoot(Position<E> p) { return p == root(); }  
5  public boolean isEmpty() { return size() == 0; }  
6  ...  
7  }
```

General Trees

Depth and Height

- Depth
 - If p is the root, the depth of p is 0.
 - Otherwise, the depth of p is one plus the depth of its parent.

```
1 public int depth(Position<E> p) throws IllegalArgumentException
{
2     if (isRoot(p))
3         return 0;
4     else
5         return 1 + depth(parent(p));
6 }
```

Running time = $O(d_p + 1)$ d_p is the depth of p
--

General Trees

Depth and Height

- The *height* of a tree is the length of the longest path from the root downward to an external node.
- Recursive definition:
 - 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.

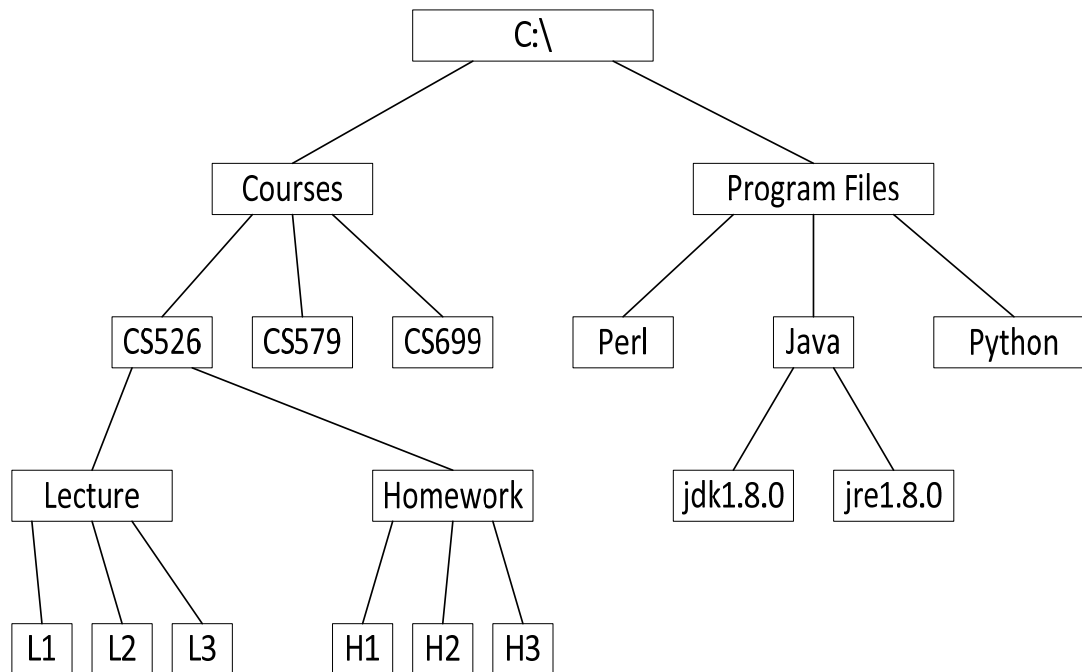
```
1 public int height(Position<E> p) throws IllegalArgumentException {  
2     int h = 0;                // base case if p is external  
3     for (Position<E> c : children(p))  
4         h = Math.max(h, 1 + height(c));  
5     return h;  
6 }
```

Running time = $O(n)$ n is the number of positions

General Trees

Depth and Height

- Example



- c:\
 - depth 0
 - height 4
- CS526
 - depth 2
 - height 2
- Program Files
 - depth 1
 - height 2

Binary Trees

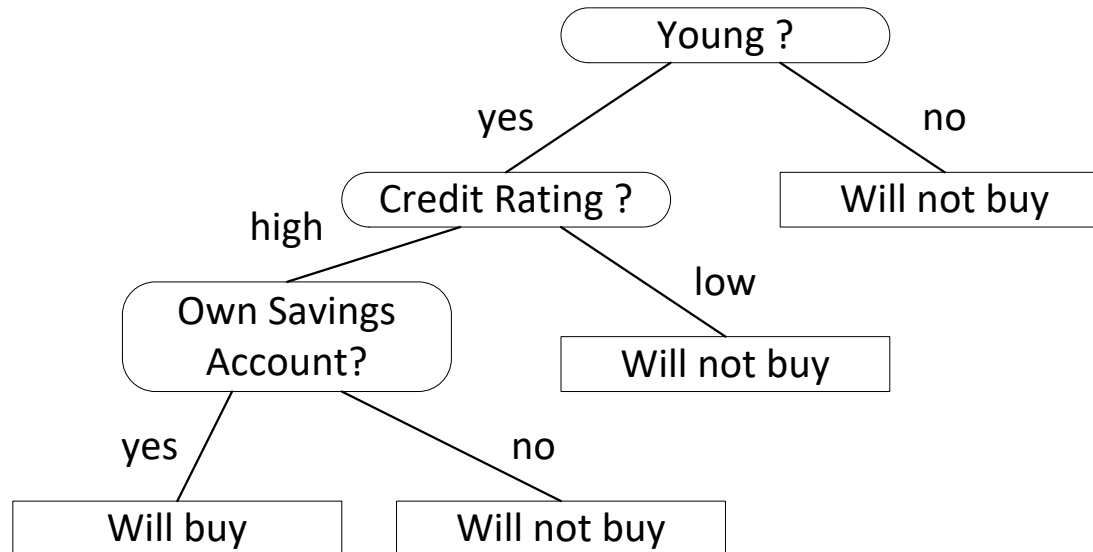
- A binary tree is an ordered tree with the following properties:
 - Every node has at most two children.
 - Each child node is labeled as being a *left child* or a *right child*.
 - A left child precedes a right child in the order of children of a node

Binary Trees

- The subtree rooted at the left or right child of an internal node v is called the *left subtree* or the *right subtree*, respectively, of v .
- A binary tree is *proper* if each node has either zero or two children. (also referred to as *full binary tree*).
- So, in a proper binary tree, every internal node has exactly two children.
- A binary tree that is not proper is *improper*.

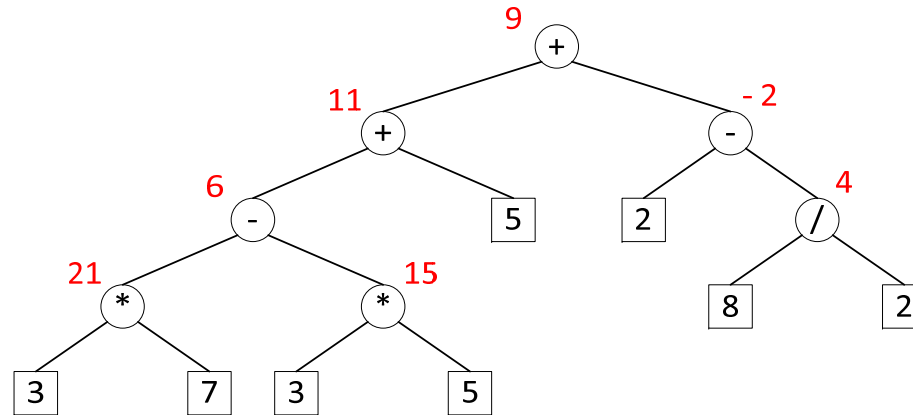
Binary Trees

- Example (a decision tree)



Binary Trees

- Example (arithmetic expression tree)



- $((((3 * 7) - (3 * 5)) + 5) + (2 - (8 / 2)))$

Binary Trees

- A binary tree can be recursively defined as follows:
 - A binary tree is either
 - An empty tree, or
 - A nonempty tree with a root node r and two binary trees that are the left subtree and the right subtree of r . One or both of these subtrees can be empty, by definition.

Binary Trees

ADT

- The binary tree ADT is a specialization of the *Tree ADT*.
- Following additional methods are defined:
 - $\text{left}(p)$: Returns the position of the left child of p .
Returns null if p has no left child.
 - $\text{right}(p)$: Returns the position of the right child of p .
Returns null if p has no right child.
 - $\text{sibling}(p)$: Returns the position of the sibling of p .
Returns null if p has no sibling.

Binary Trees

ADT

- BinaryTree interface

```
1  public interface BinaryTree<E> extends Tree<E> {  
2      Position<E> left(Position<E> p) throws  
           IllegalArgumentException;  
3      Position<E> right(Position<E> p) throws  
           IllegalArgumentException;  
4      Position<E> sibling(Position<E> p) throws  
           IllegalArgumentException;  
5  }
```

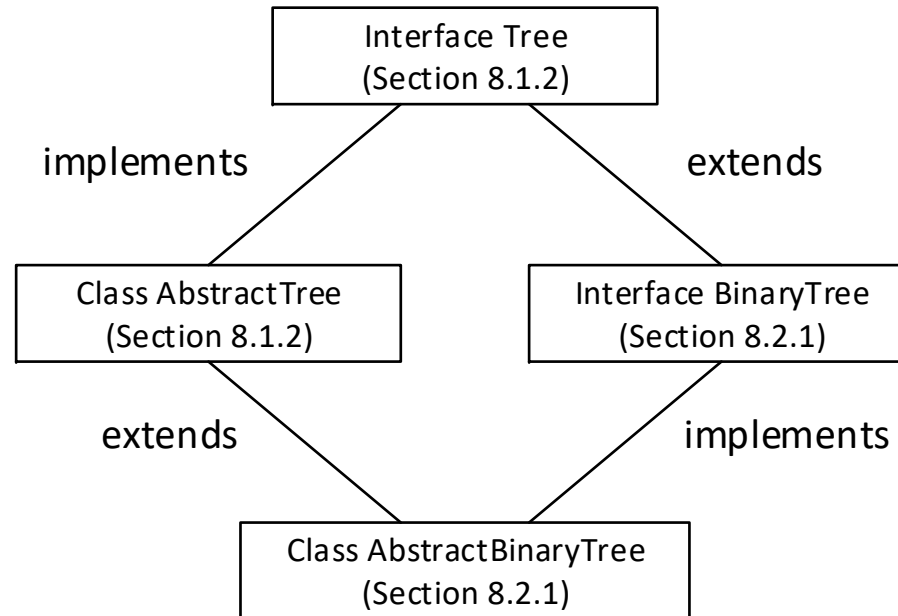
Binary Trees

ADT

- AbstractBinaryTree: extends AbstractTree and implements BinaryTree

- Additional methods:

- sibling
- numChildren
- children

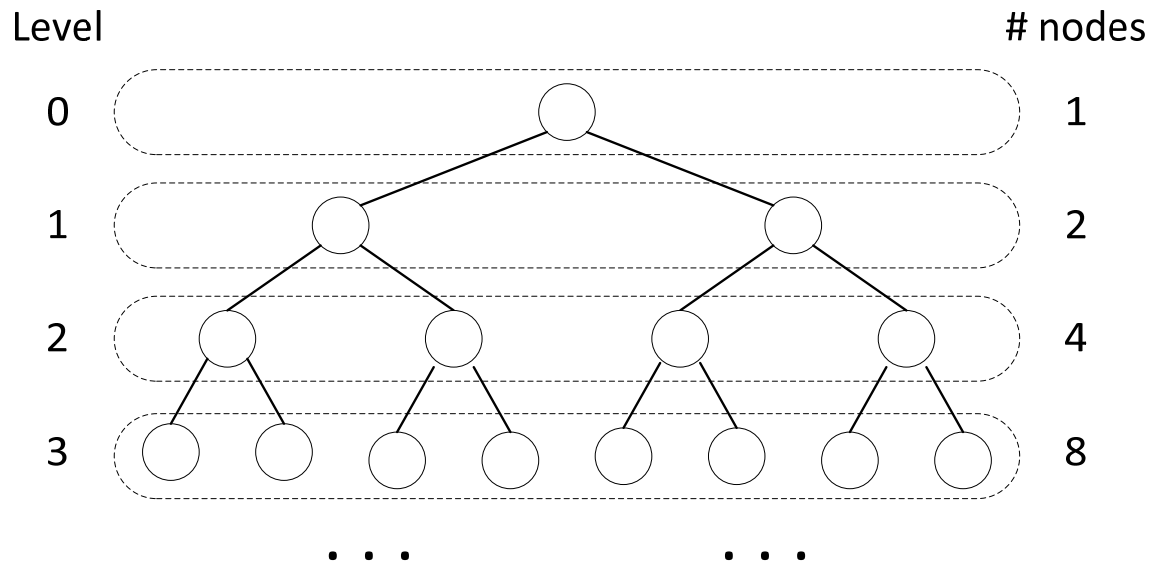


- AbstractBinaryTree.java

Binary Trees

Binary Tree Properties

- Let *level* d of a binary tree T be the set of nodes at depth d of T .



- The maximum number of nodes at level d is 2^d .

Binary Trees

Binary Tree Properties

- n : the number of nodes in T
 - n_E : the number of external nodes in T
 - n_I : the number of internal nodes in T
 - h : the height of T
-
- $h + 1 \leq n \leq 2^{h+1} - 1$
 - $1 \leq n_E \leq 2^h$
 - $h \leq n_I \leq 2^h - 1$
 - $\log(n + 1) - 1 \leq h \leq n - 1$

Binary Trees

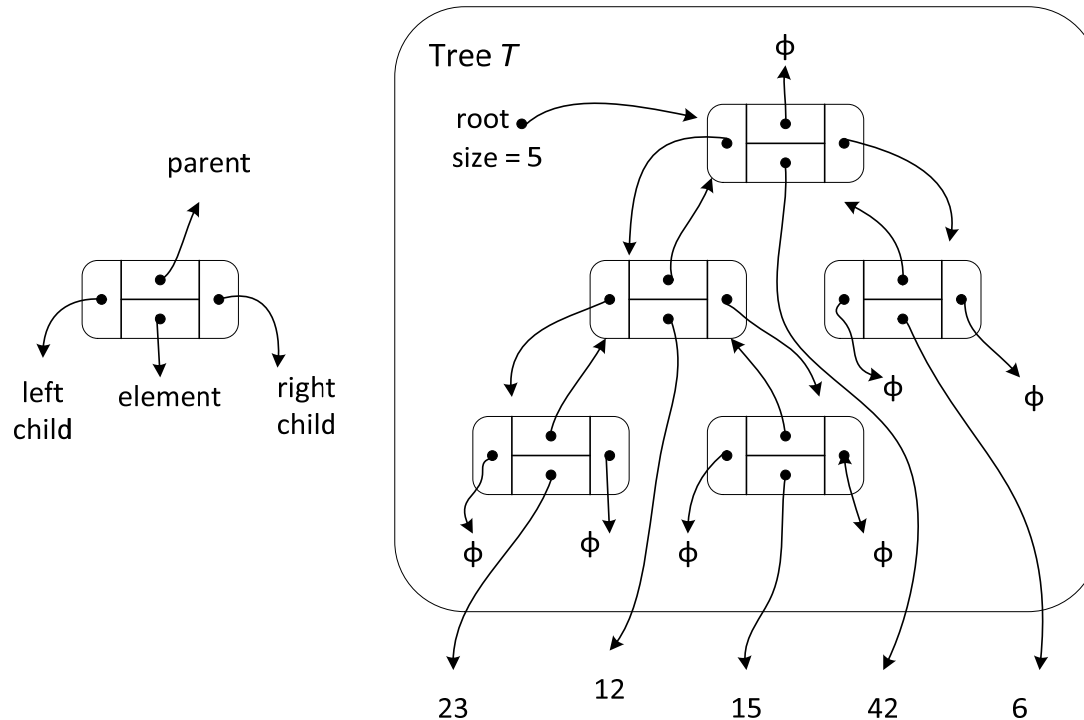
Binary Tree Properties

- If T is a proper binary tree:
- $2h + 1 \leq n \leq 2^{h+1} - 1$
- $h + 1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $\log(n + 1) - 1 \leq h \leq (n - 1)/2$
- $n_E = n_I + 1$

Binary Trees

Implementation Using Linked Structure

- A node has the following linked structure.



Binary Trees

Implementation Using Linked Structure

- `LinkedBinaryTree` extends `AbstractBinaryTree` abstract class with the following update methods:
 - `addRoot(e)`: Creates a new node with element *e* and make it the root of an empty tree. Returns the position of the root. An error occurs if the tree is not empty.
 - `addLeft(p, e)`: Creates a new node with element *e* and make it a left child of position *p*. Returns the position of the new node (left child). An error occurs if *p* already has a left child.
 - `addRight(p, e)`: Creates a new node with element *e* and make it a right child of position *p*. Returns the position of the new node (right child). An error occurs if *p* already has a right child.

Binary Trees

Implementation Using Linked Structure

- Update methods (continued):
 - $\text{set}(p, e)$: Replaces the element of p with element e . Returns the previously stored element.
 - $\text{attach}(p, T_1, T_2)$: Attaches internal structure of T_1 and T_2 as the left subtree and the right subtree, respectively, of a leaf node position p and resets T_1 and T_2 to empty trees. If p is not a leaf node, an error occurs.
 - $\text{remove}(p)$: Removes the node at position p , replacing it with its child (if any). Returns the element that had been stored at p . An error occurs if p has two children.

Binary Trees

Implementation Using Linked Structure

- A node is a *position* (instance variables shown below)

```
1  protected static class Node<E> implements Position<E> {  
2    private E element;      // an element stored at this node  
3    private Node<E> parent;  // a reference to the parent node (if any)  
4    private Node<E> left;    // a reference to the left child (if any)  
5    private Node<E> right;   // a reference to the right child (if any)
```

- LinkedBinaryTree has two instance variables

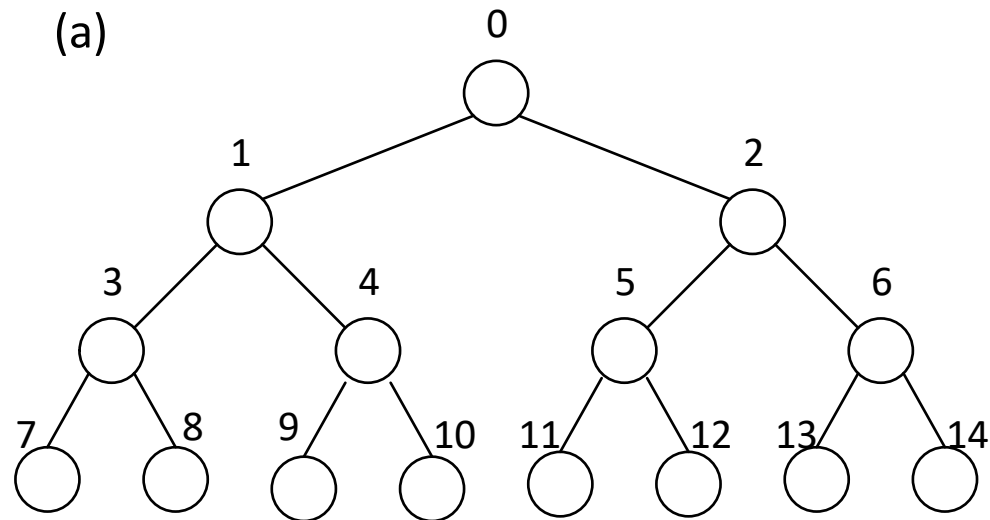
```
protected Node<E> root = null;  
private int size = 0;
```

- LinkedBinaryTree.java

Binary Trees

Implementation Using Array

- Nodes are stored in an array.
- *Level numbering* scheme is used.

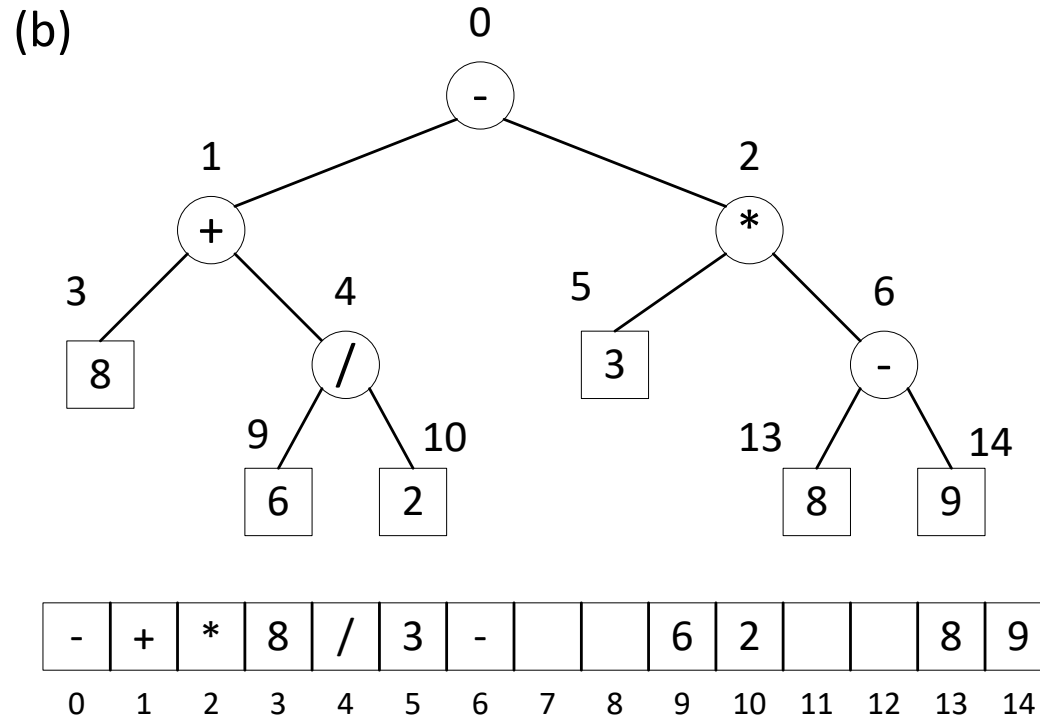


- A number above a node is the index in the array.

Binary Trees

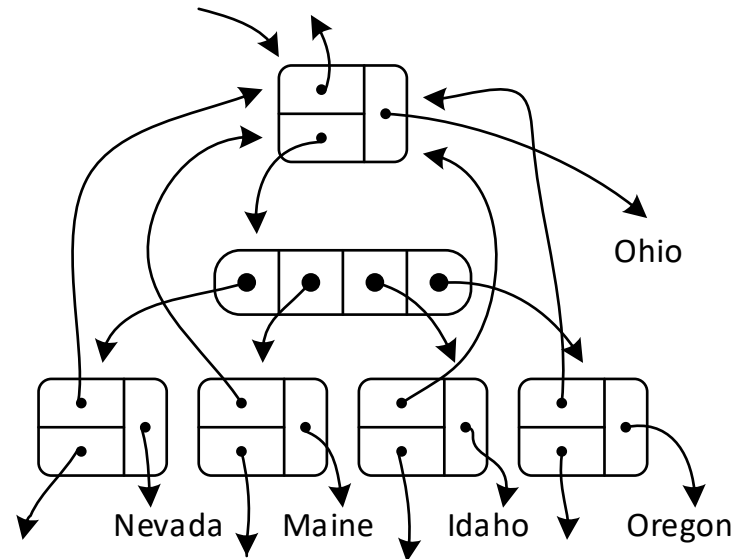
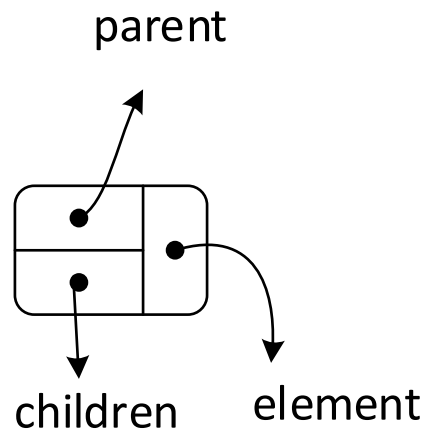
Implementation Using Array

- Example



Binary Trees

Linked Structure for General Trees



Binary Trees

Tree Traversal

- A *traversal* of a tree T is a systematic way of visiting all positions in T .
- Preorder tree traversal:
 - visit the root
 - visit all children

Algorithm $\text{preorder}(p)$

visit p

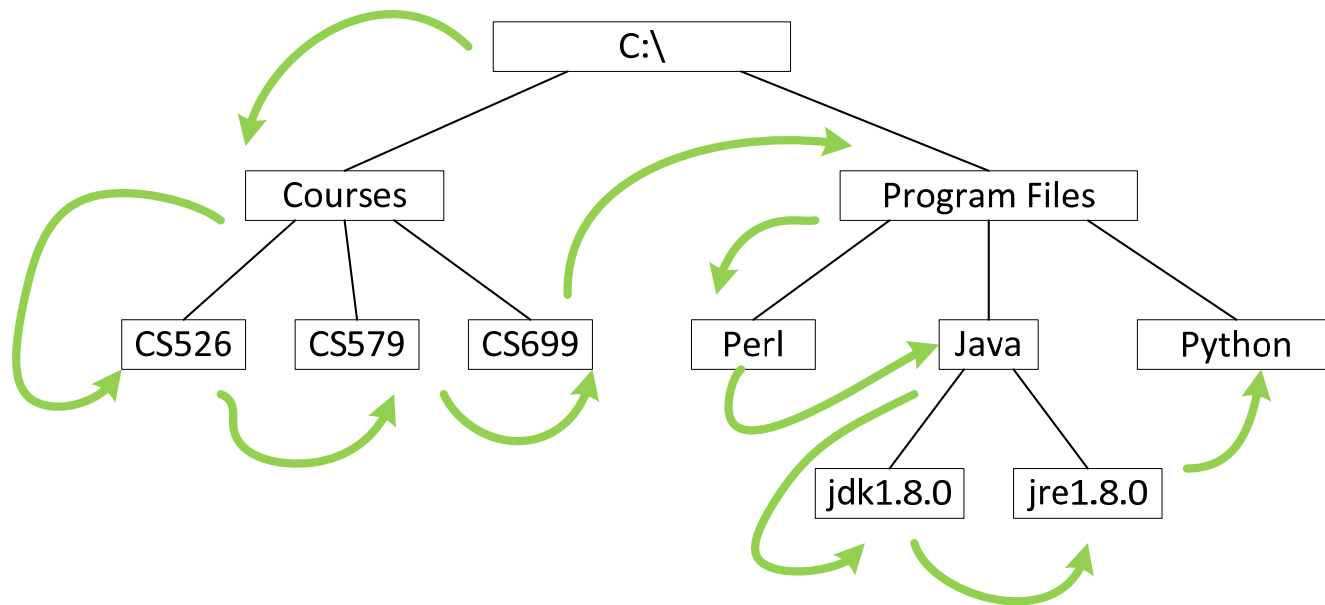
for each child c in $\text{children}(p)$

$\text{preorder}(c)$

Binary Trees

Tree Traversal

- Preorder tree traversal illustration:



Binary Trees

Tree Traversal

- Postorder tree traversal:
 - Visit all children (recursively)
 - Visit the root

Algorithm $\text{postorder}(p)$

 for each child c in $\text{children}(p)$

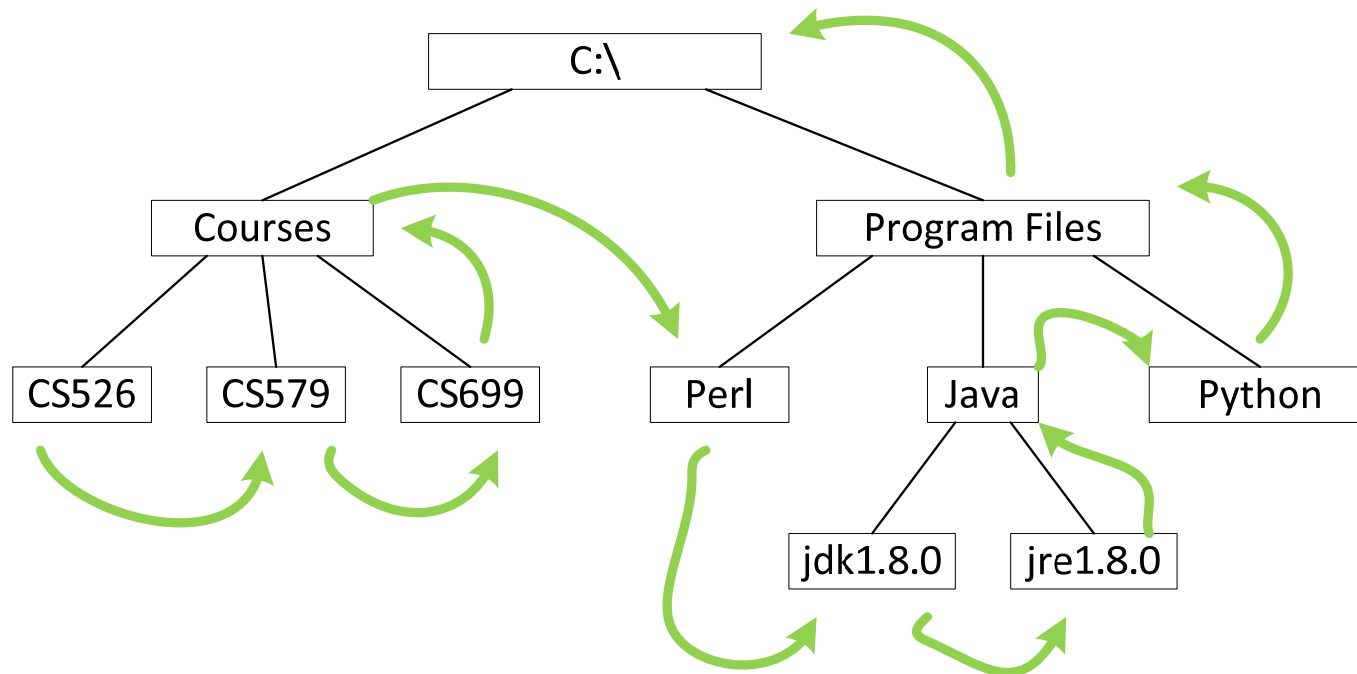
$\text{postorder}(c)$

 visit p

Binary Trees

Tree Traversal

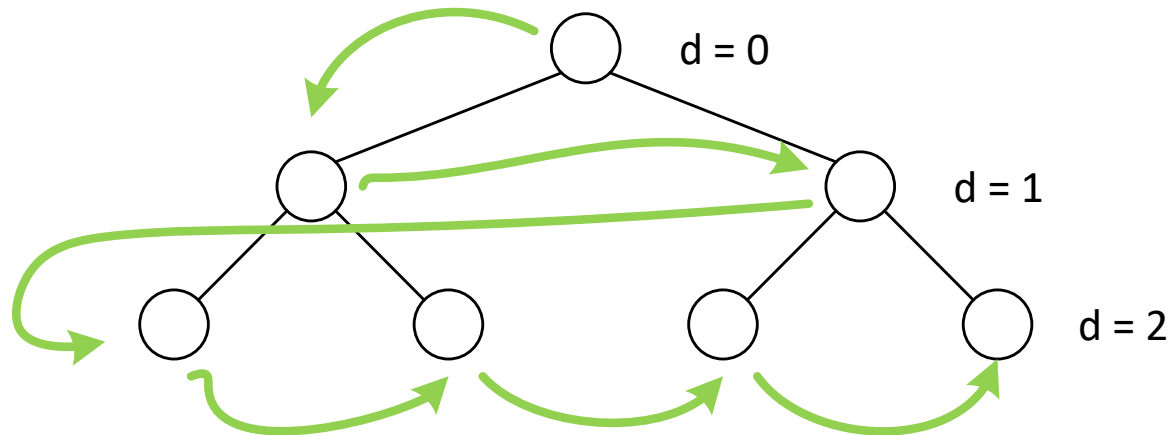
- Postorder tree traversal illustration



Binary Trees

Tree Traversal

- Breadth-first tree traversal
 - Also called *breadth-first search* or *BFS*
 - Visits all positions at depth d before visiting positions at depth $d + 1$.



Binary Trees

Tree Traversal

- Breadth-first tree traversal (continued)

Algorithm breadthfirst()

 initialize Q to contain the root of the tree

 while Q is not empty

$p = Q.dequeue()$ // remove the oldest entry in Q

 visit p

 for each child c in $children(p)$

$Q.enqueue(c)$ // add all children of p to the rear of Q

- Running time
 - Each node is enqueued and dequeued once each.
 - $O(n)$

Binary Trees

Tree Traversal

- Inorder tree traversal of binary tree
 - Visit the left subtree
 - Visit the root
 - Visit the right subtree

Algorithm $\text{inorder}(p)$

if p has a left child lc // visit left subtree

$\text{inorder}(lc)$

visit p

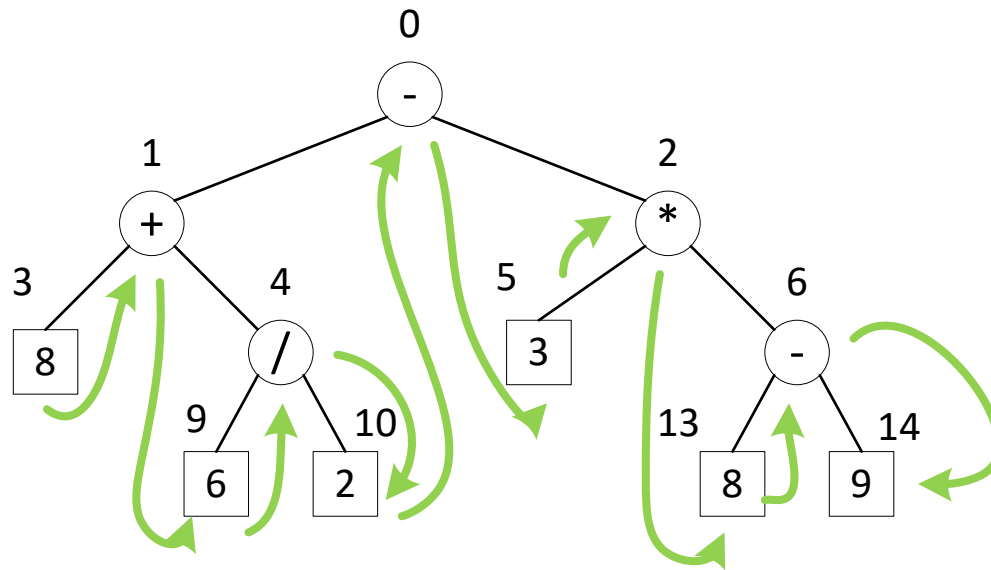
if p has a right child rc // visit right subtree

$\text{inorder}(rc)$

Binary Trees

Tree Traversal

- Inorder tree traversal of binary tree illustration:



- Inorder tree traversal generates: $8 + 6 / 2 - 3 * 8 - 9$
- Correct expression without parentheses

Binary Trees

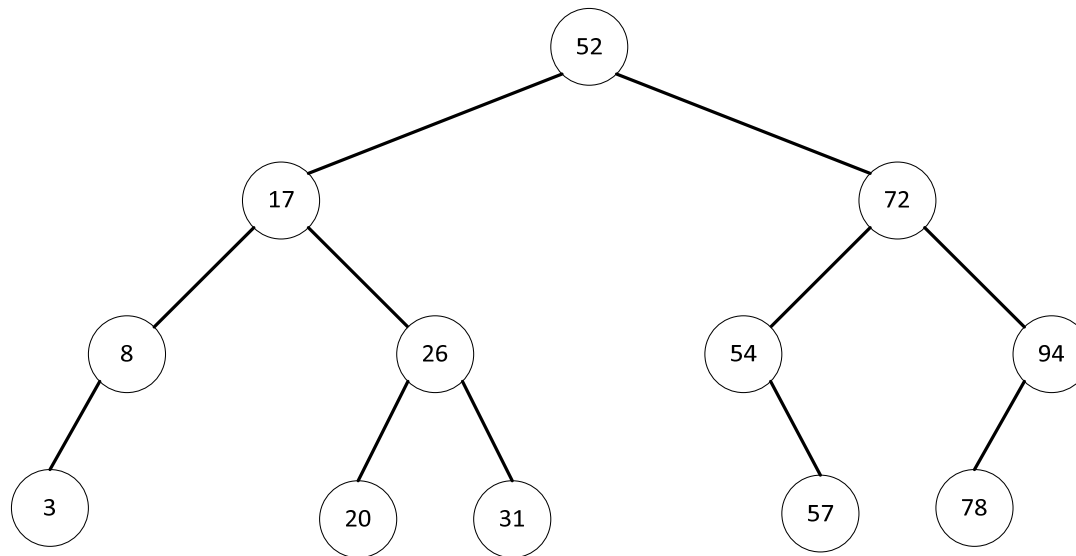
Binary Search Tree

- A binary search tree is a binary tree with additional properties:
 - Each position p stores an element, denoted as $e(p)$.
 - All elements in the left subtree of a position p (if any) are less than $e(p)$.
 - All elements in the right subtree of a position p (if any) are greater than $e(p)$.

Binary Trees

Binary Search Tree

- A binary search tree example:



- Inorder tree traversal generates:
3, 8, 17, 20, 26, 31, 52, 54, 57, 72, 78, 94

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.