FALL 2018 CISC 311

OBJECT & STRUCTURE & ALGORITHM I

– Chapter 7: Binary Trees

Outline

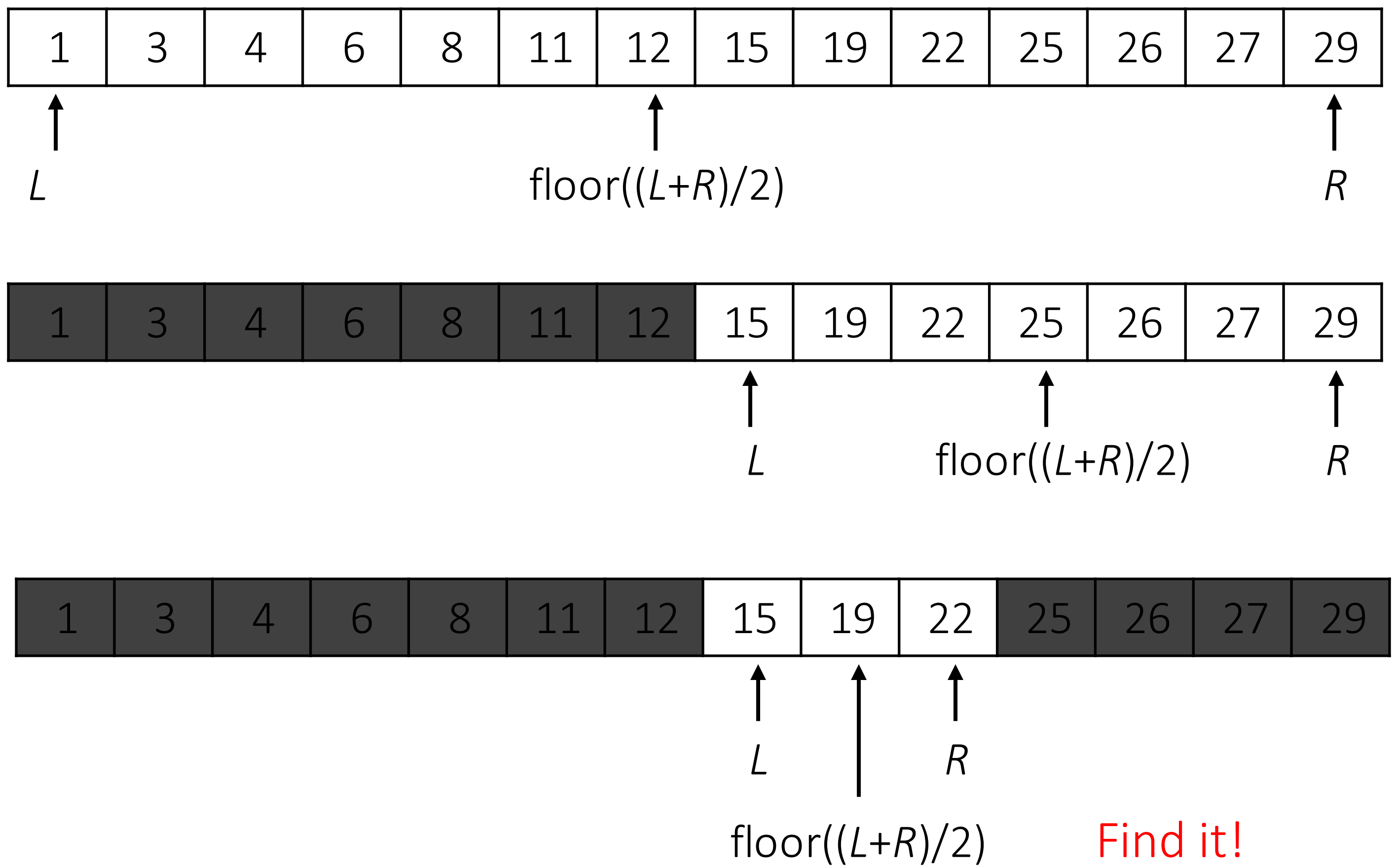
- The tree ADT
- The binary tree ADT
- Implementations
- Tree traversal
  - Preorder traversal
  - Postorder traversal
  - Inorder traversal
  - Breadth-first search

Exercise

- Given a sorted sequence and a search key, find it in the sequence
  - For example: 1, 3, 4, 6, 8, 11, 12, 15, 19, 22, 25, 26, 27, 29
  - Search key: 19
  - What's the complexity of the solution? $O(n)$
  - Can we come up with another solution? $O(\log n)$

`BruteForceSearch.java`

Binary Search



`BinarySearch.java`

Binary Search (cont'd.)

```
Algorithm binarySearch(A, key)
    Input A array A, key search key
    Output m
    L ← 0
    R ← arr.length - 1
    if L > R then return -1
    while L <= R do
        m ← floor((L + R) / 2)
        if arr[m] = key then return m
        else if arr[m] < key then
            L ← m + 1
        else if arr[m] > key then
            R ← m – 1
    return –1
```

`BinarySearch.java`

Second Exercise

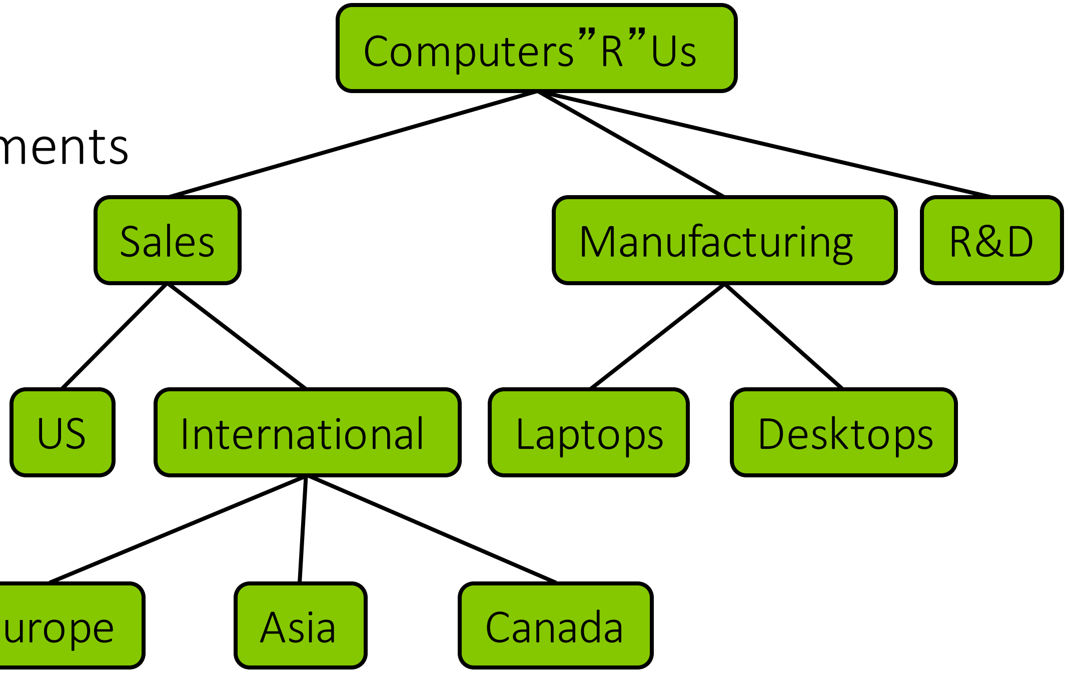
- Given a sorted sequence, insert a new element into the sequence
  - For example: 1, 3, 4, 6, 8, 11, 12, 15, 19, 22, 25, 26, 27, 29
  - Insert 20 into the sequence
  - What's the complexity of the solution? $O(n)$
- Slow insertion in an ordered array
  - These multiple moves are time-consuming, requiring, on the average, moving half the items ($n/2$ moves)
  - Deletion involves the same multiple move operation and is thus equally slow
  - If you're going to be doing a lot of insertions and deletions, an ordered array is a bad choice

Why We Need Trees

- Slow searching in a linked list
  - Insertions and deletions are quick to perform on a linked list
  - These operations require $O(1)$ time
  - You will need to visit an average of $n/2$ objects, comparing each one's value with the key. This process is slow, requiring $O(n)$ time
- Tree to the rescue!
  - With the quick insertion and deletion of a linked list
  - And also with the quick searching of an ordered array

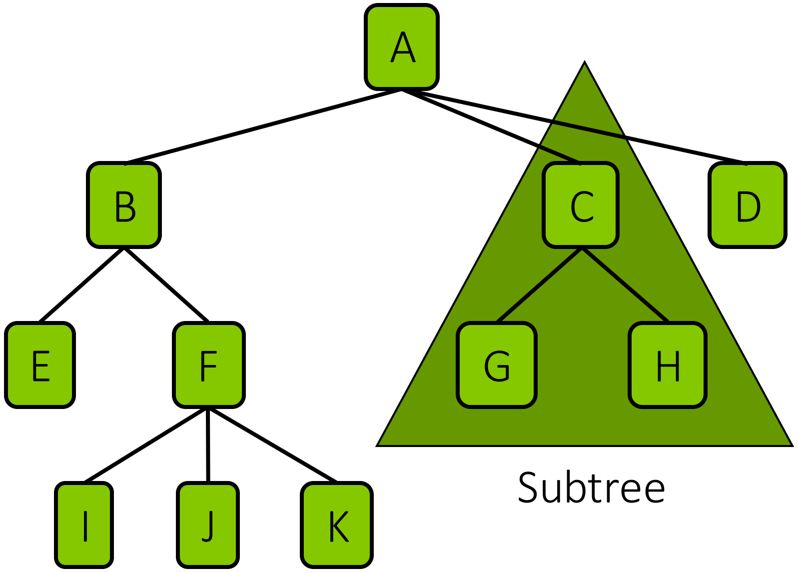
What is a Tree

- 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



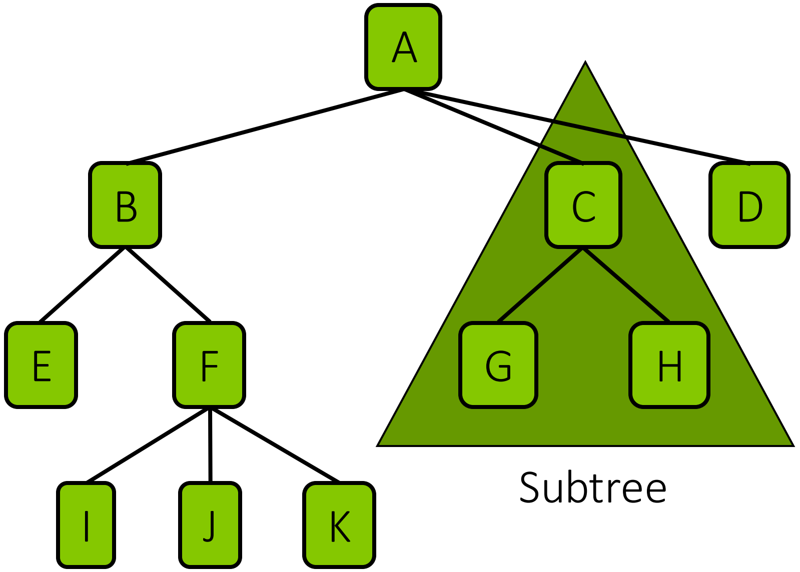
Tree Terminology

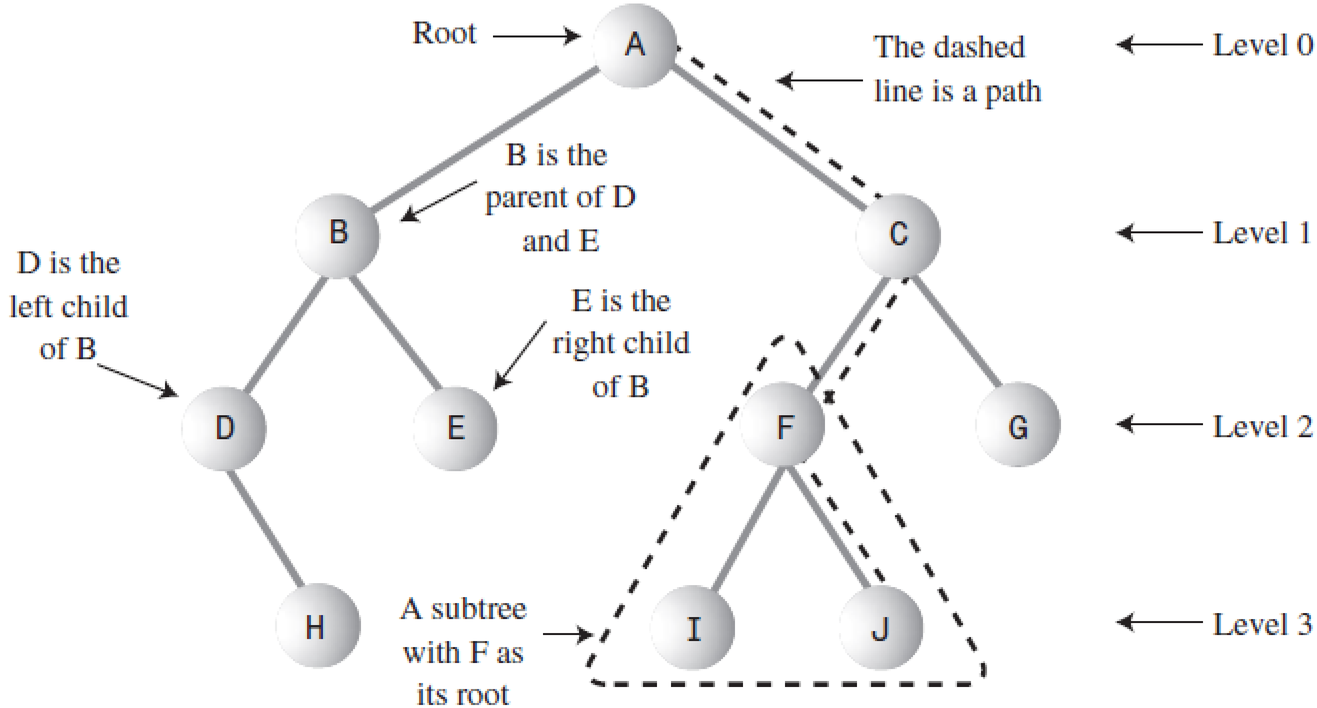
- Path: edge connected nodes
- Root: node without parent (A)
  - One and only one path from root to any nodes
- Ancestors of a node: parent, grandparent, great-grandparent, etc.
- Descendant of a node: child, grandchild, great-grandchild, etc.
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)

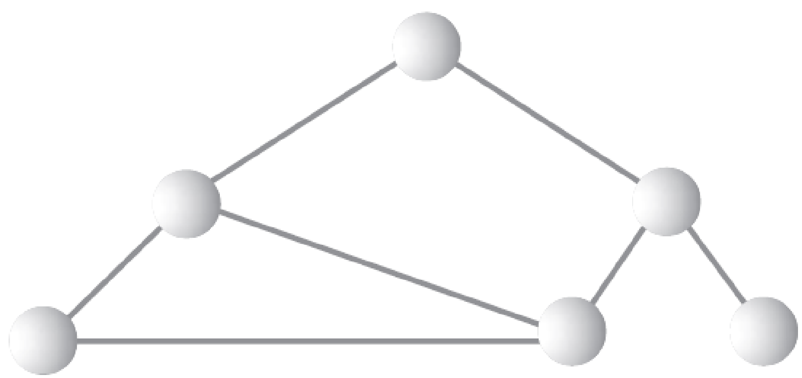


Tree Terminology (cont'd.)

- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Subtree: tree consisting of a node and it descendants
- Visiting: program control arrives at the node
- Traversing: visit all the nodes in some specified order




Root
A
The dashed
line is a path
Level 0
B is the
parent of D
and E
B
C
Level 1
D is the
left child
of B
E is the
right child
of B
D
E
F
G
Level 2
H
A subtree
with F as
its root
I
J
Level 3



Not a tree

Tree ADT

- A tree node supports `getElement()`: returns the element stored at the tree node
- Generic methods:
  - `Integer size()`
  - `boolean isEmpty()`
- Accessor methods:
  - `Node root()`
  - `Node parent(`*n*`)`
  - `Iterable children(`*n*`)`
  - `Integer numChildren(`*n*`)`
- Query methods:
  - `boolean isInternal(`*n*`)`
  - `boolean isExternal(`*n*`)`
  - `boolean isRoot(`*n*`)`
  - `Iterable nodes()`

`Tree.java`

Method	Description
`size()`	Returns the number of nodes and hence elements that are contained in a tree
`isEmpty()`	Returns `true` if the tree does not contain any nodes and thus no elements
`root()`	Returns the node of the root of the tree or `null` if empty
`parent(`*n*`)`	Returns the node of the parent of the node *n* or `null` if empty
`children(`*n*`)`	Returns an iterable collection containing the children of node *n* if any
`numChildren(`*n*`)`	Returns the number of children of node *n*

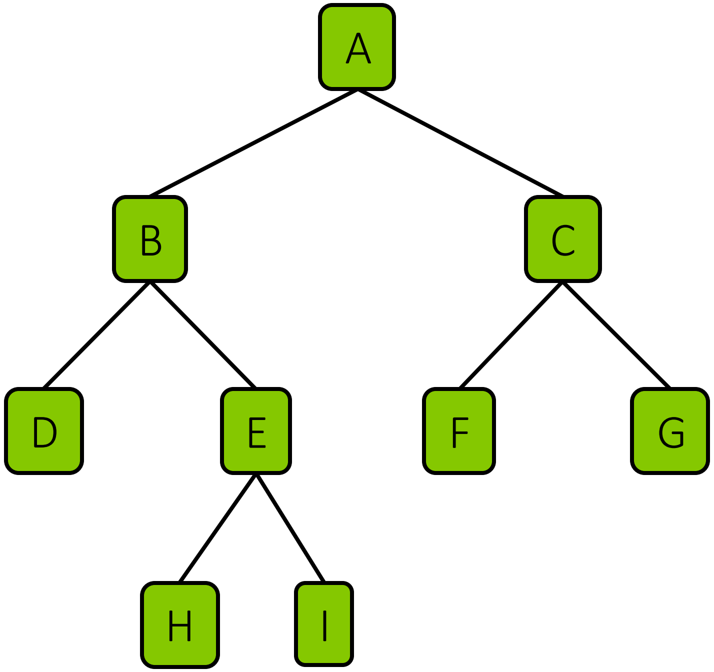
`Tree.java`

Method	Description
`isInternal(`*n*`)`	Returns `true` if node *n* has at least one child
`isExternal(`*n*`)`	Returns `true` if node *n* does not have any children
`isRoot(`*n*`)`	Returns `true` if node *n* is the root of the tree
`nodes()`	Returns an iterable collection of all nodes of the tree

`Tree.java`

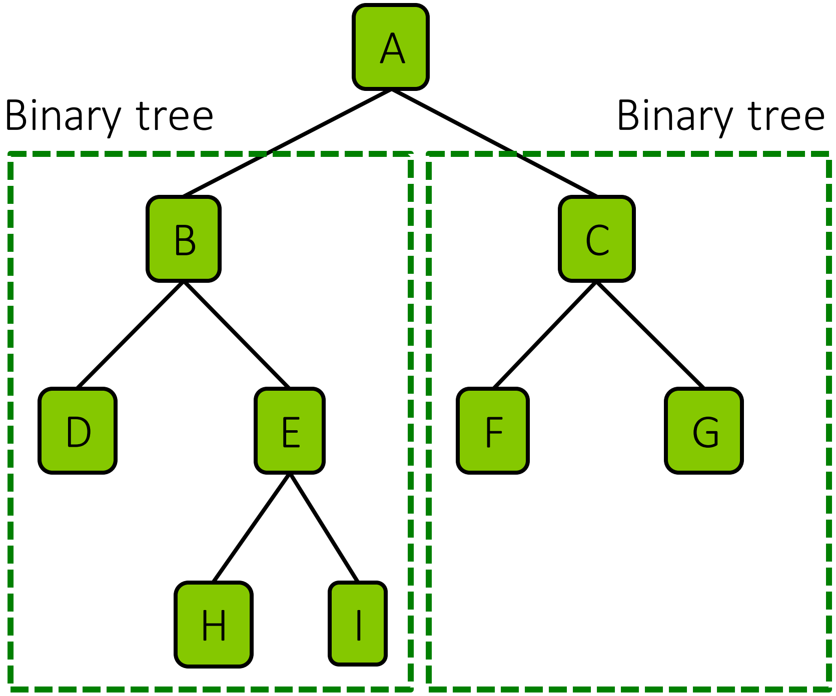
Binary Trees

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



Binary Trees (cont'd.)

- 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



Binary Tree ADT

- The Binary Tree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
  - Additional methods:
    - `Node left(`*n*`)`
    - `Node right(`*n*`)`
    - `Node sibling(`*n*`)`
  - Sibling of a Node *n* as the other child of *n*'s parent
  - *n* does not have a sibling if it is the root, or if it is the only child of its parent

`BinaryTree.java`

Additional Methods of Binary Trees

Method	Description
`left(`n`)`	Returns the node of the left child of n or null if n has no left child
`right(`n`)`	Returns the node of the right child of n or `null` if n has no left child
`sibling(`n`)`	Returns the node of the sibling of n or null if n has no siblings

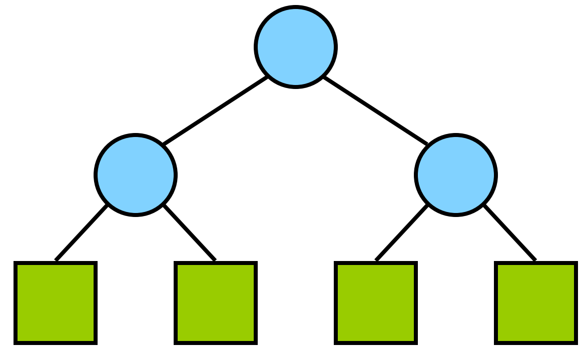
`BinaryTree.java`

Types of Binary Trees

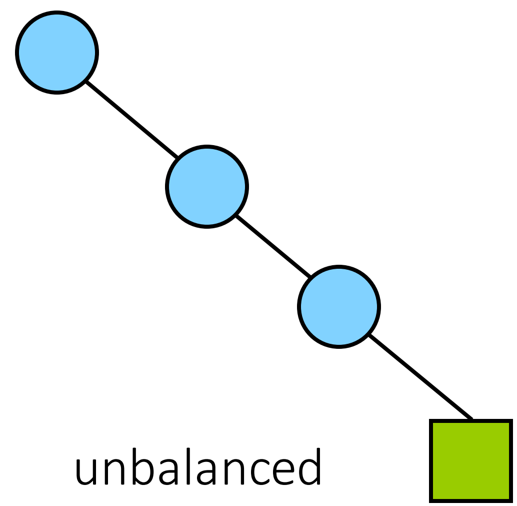
- A proper binary tree is a binary tree in which every node has either zero or two children
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible
- A perfect binary tree is a binary tree in which all internal nodes have two children and all leaves have the same depth or same level
- A balanced binary tree is a binary tree in which the left and right subtrees of every node differ in height by no more than 1

Properties of Binary Trees

- Notation
  - T : nonempty tree
  - n : number of nodes
  - n_E : number of external nodes
  - n_I : number of internal nodes
  - h : height
- Properties:
  - $h + 1 \le n \le 2^{h+1} - 1$
  - $1 \le n_E \le 2^h$
  - $h \le n_I \le 2^h - 1$
  - $\log_2 (n + 1) - 1 \le h \le n - 1$

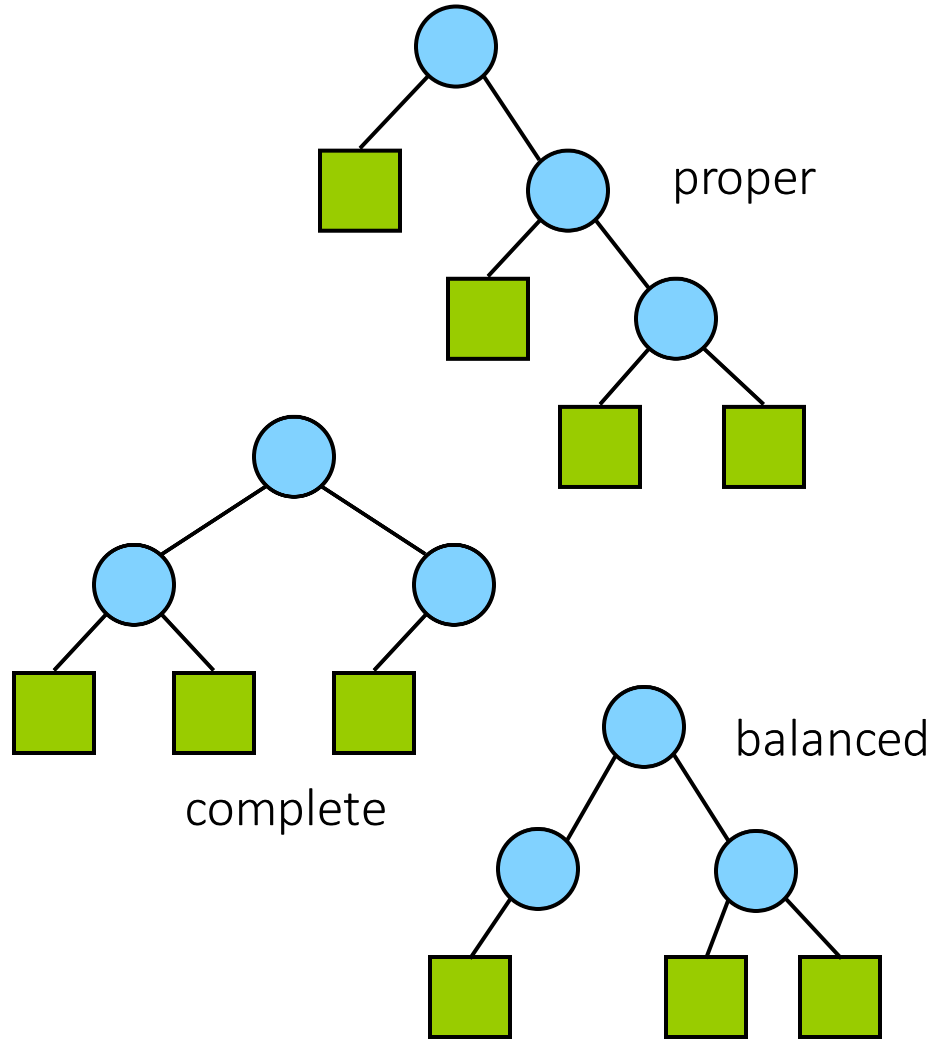


balanced, complete, perfect and proper

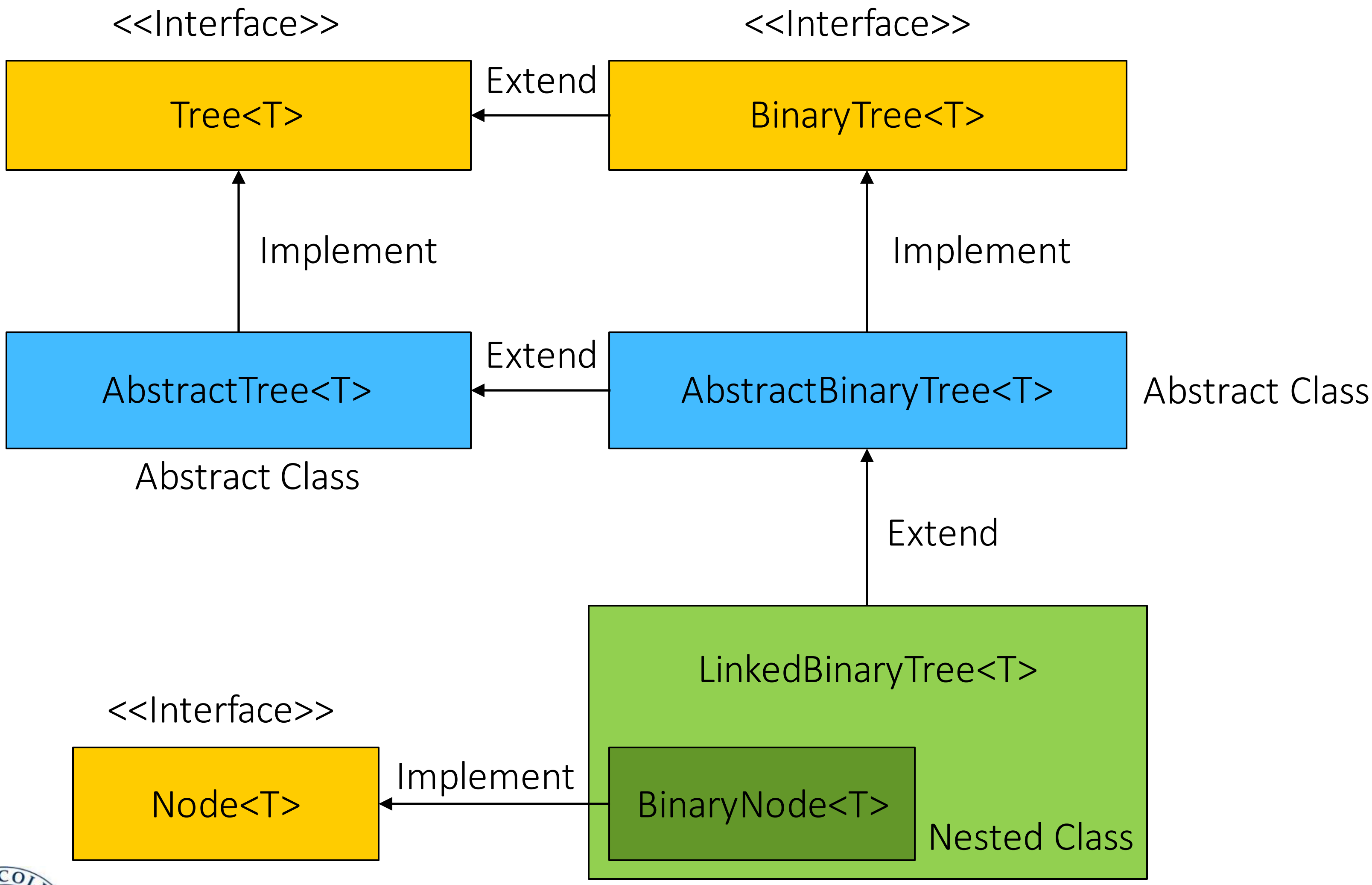


Properties of Binary Trees (cont'd.)

- Notation
  - T : nonempty proper tree
  - n : number of nodes
  - n_E : number of external nodes
  - n_I : number of internal nodes
  - h : height
- Properties:
  - $2h + 1 \leq n \leq 2^{h+1} - 1$
  - $h + 1 \leq n_E \leq 2^h$
  - $h \leq n_I \leq 2^h - 1$
  - $\log_2 (n + 1) - 1 \leq h \leq (n - 1)/2$
  - $n_E = n_I + 1$

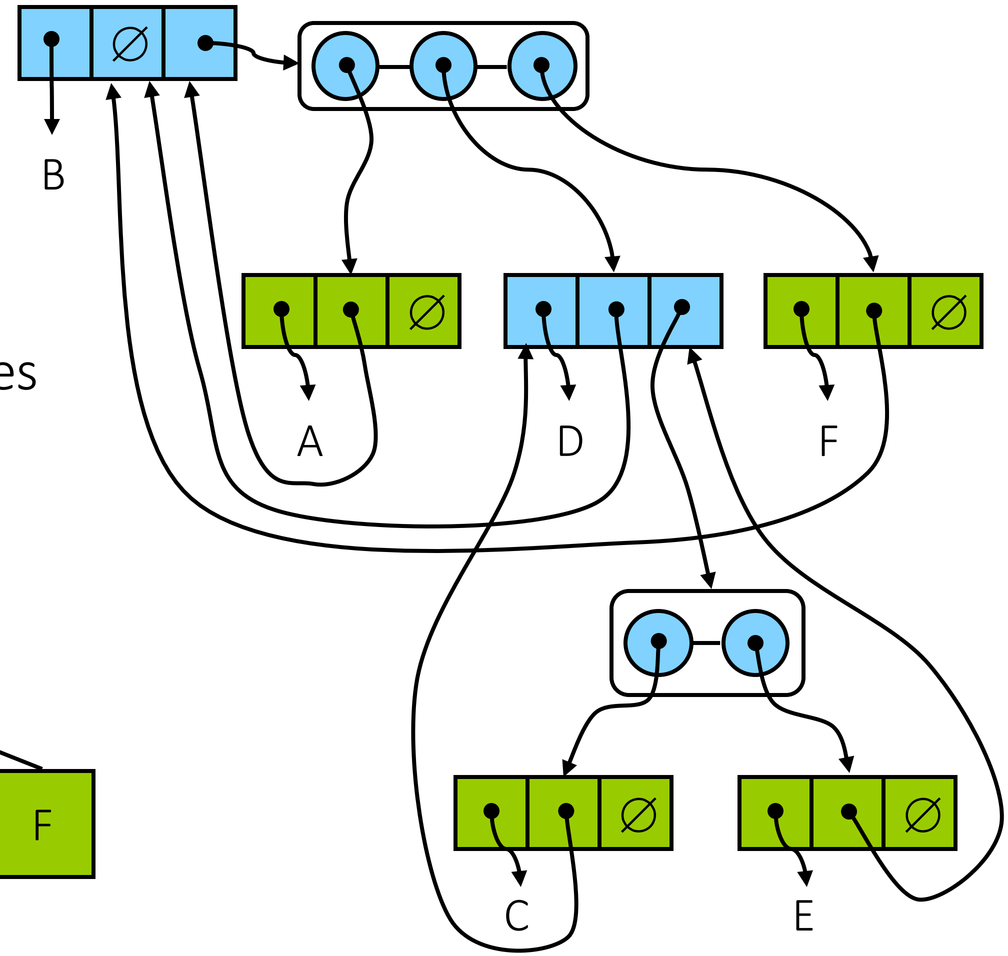


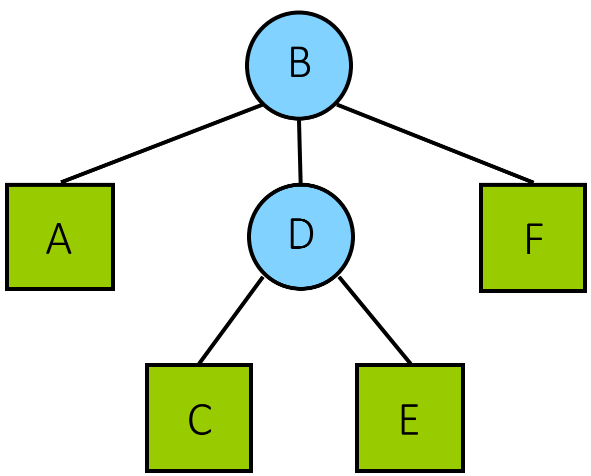
Implementation



Linked Structure for Trees

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

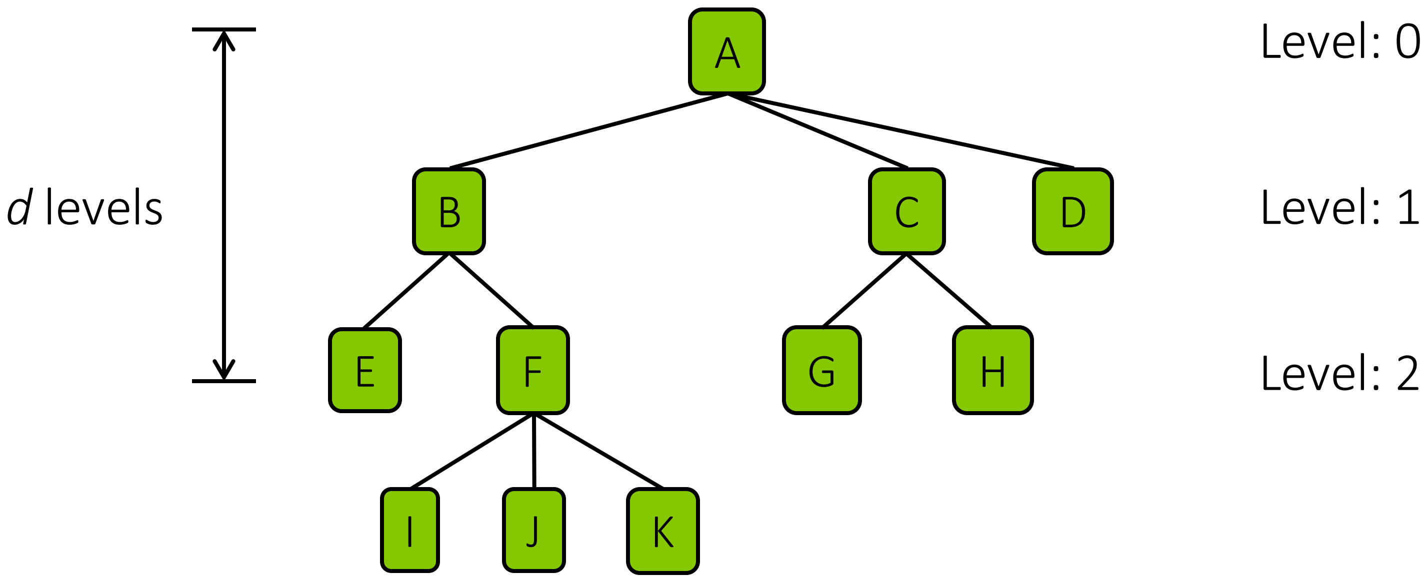




`AbstractTree.java`

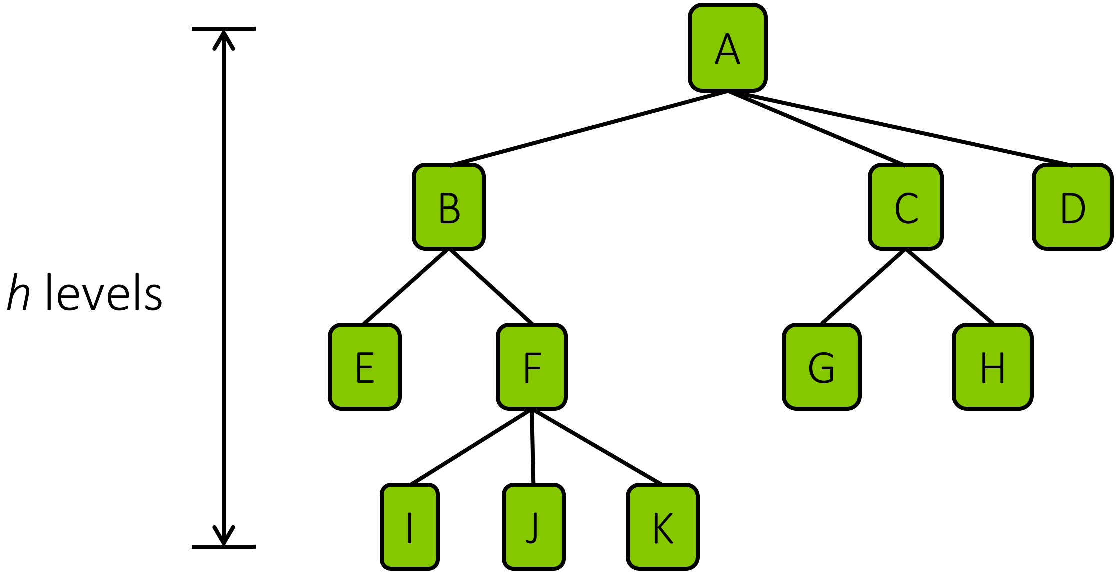
Depth of a Node

- Method: `depth(`n`)`
  - If n is the root, then the depth of the tree is 0
  - Otherwise, the depth of n is one plus the depth of the parent of n



Height of a Tree

- Method: `height()`
  - Height of a tree = maximum of the depths of its nodes

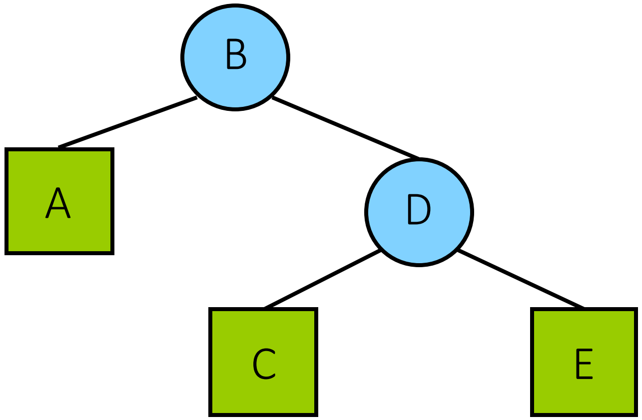


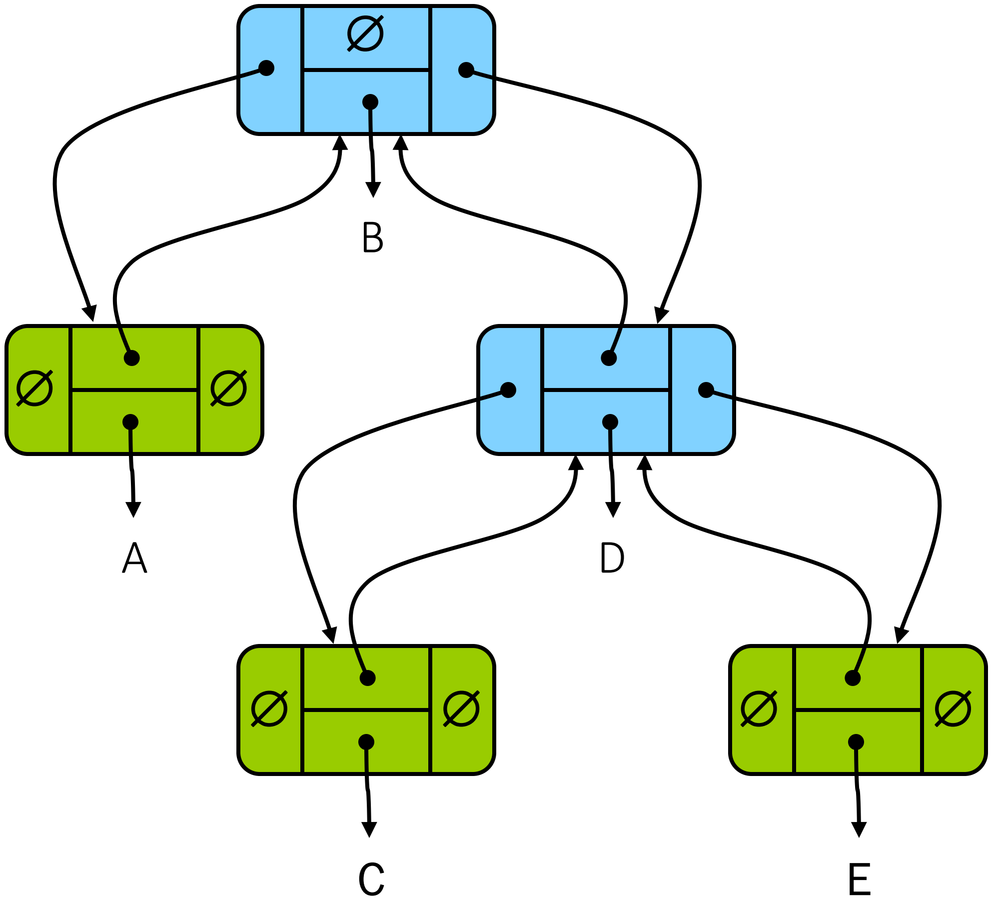
Two Additional Methods

Method	Description
`depth(`n`)`	Returns the number of levels separating node n from the root
`height(`n`)`	Returns the height of the 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 node ADT





`AbstractBinaryTree.java`

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

LinkedBinaryTree.java

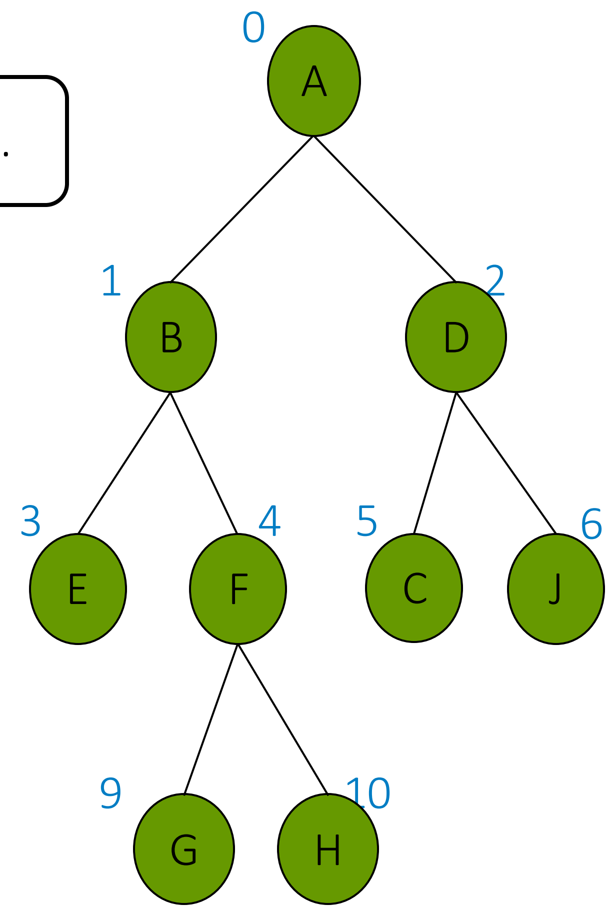
Array-Based Representation of Binary Trees

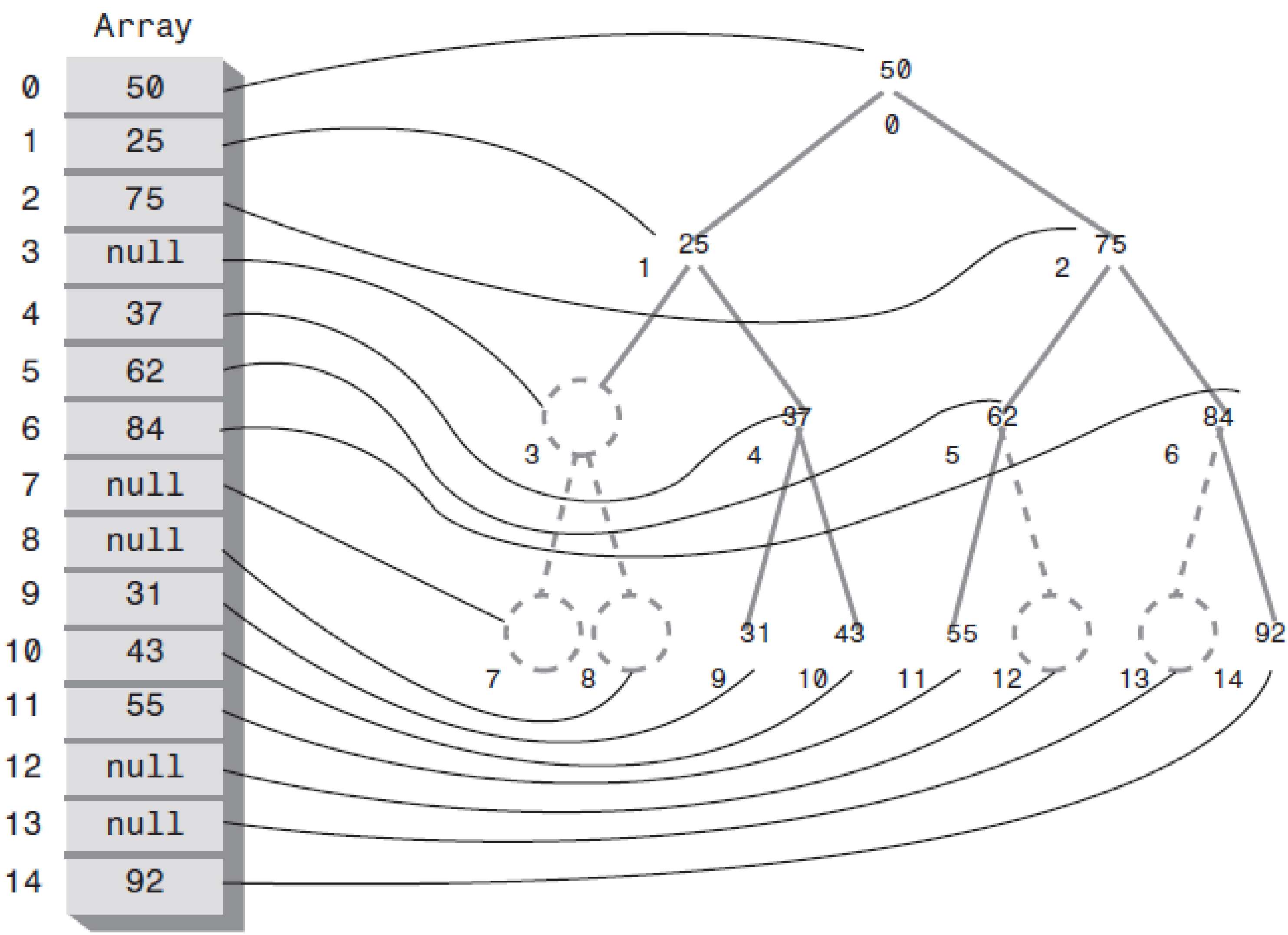
- Based on a way of numbering the node of T
  - If p is the root of T, then $f(p) = 0$
  - If p is the left child of node q, then $f(p) = 2f(q) + 1$
  - If p is the right child of node q, then $f(p) = 2f(q) + 2$
  - If p is the parent of node q, then $f(p) = (f(q) – 1)/2$
  - $f()$: Level numbering of the nodes in a binary tree T

Array-Based Representation of Binary Trees (cont'd.)

- Nodes are stored in an array A

- Node v is stored at $A[rank(v)]$
  - $rank(root) = 0$
  - If node is the left child of $parent(node)$,
    $rank(node) = 2 \cdot rank(parent(node)) + 1$
  - If node is the right child of $parent(node)$,
    $rank(node) = 2 \cdot\ rank(parent(node)) + 2$





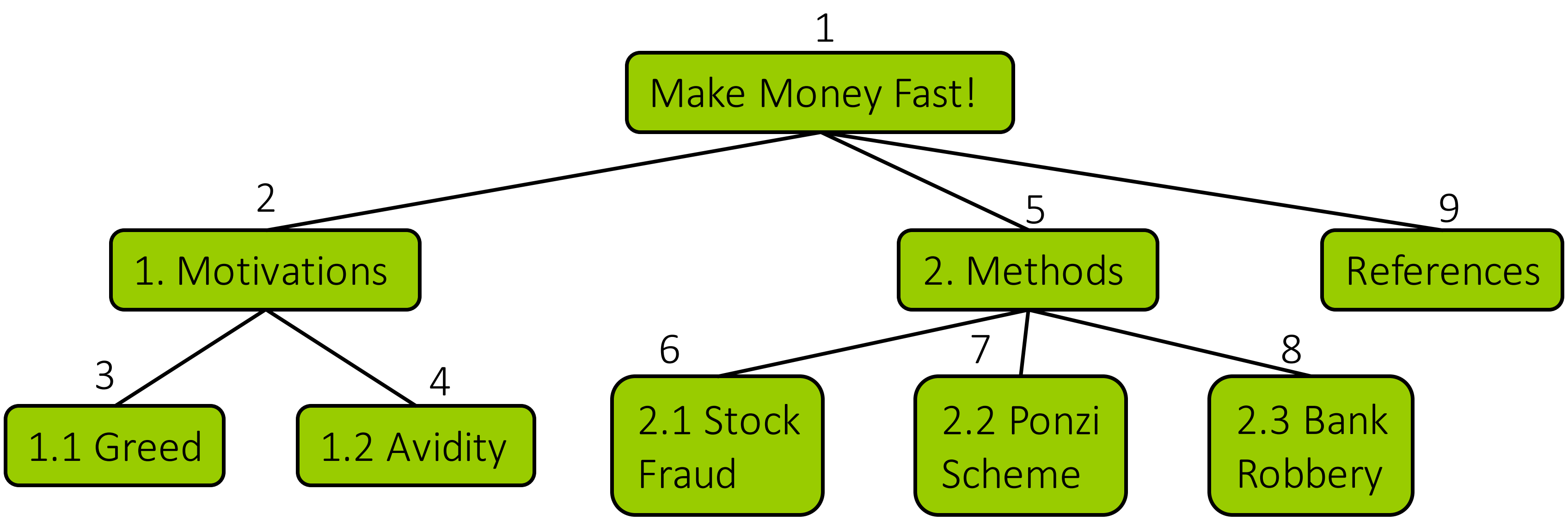
Array
0
50
1
25
2
75
3
null
4
37
5
62
6
84
7
null
8
null
9
31
10
43
11
55
12
null
13
null
14
92
50
0
25
1
75
2
3
37
4
62
5
84
6
7
8
31
9
43
10
55
11
12
13
92
14

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

Preorder Traversal

Algorithm preOrder(v)
    visit(v)
    for each child c of v **do**
        preOrder(c)

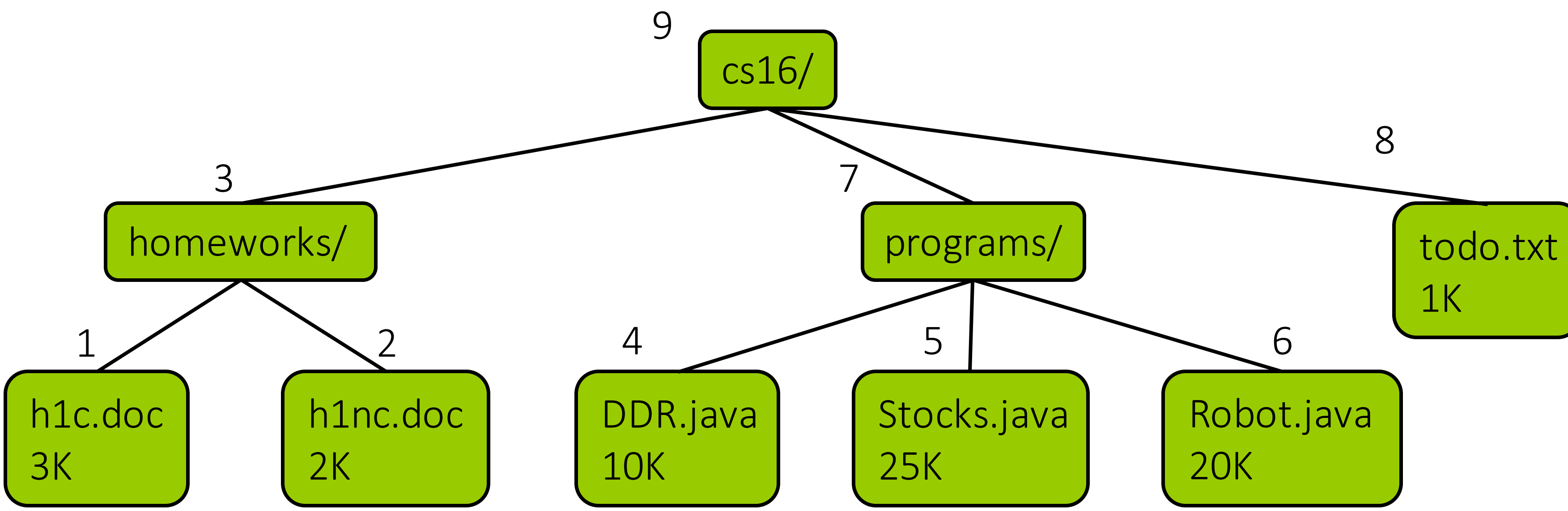


Postorder Traversal

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

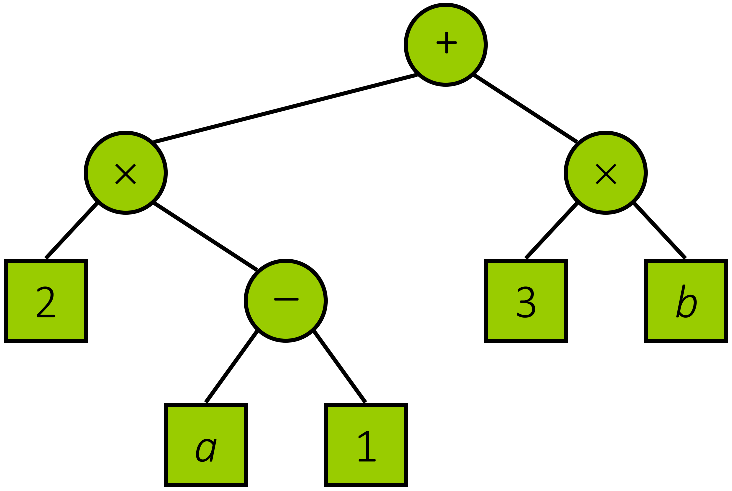
Postorder Traversal

```
Algorithm postOrder(v)
    for each child c of v do
        postOrder(c)
    visit(v)
```



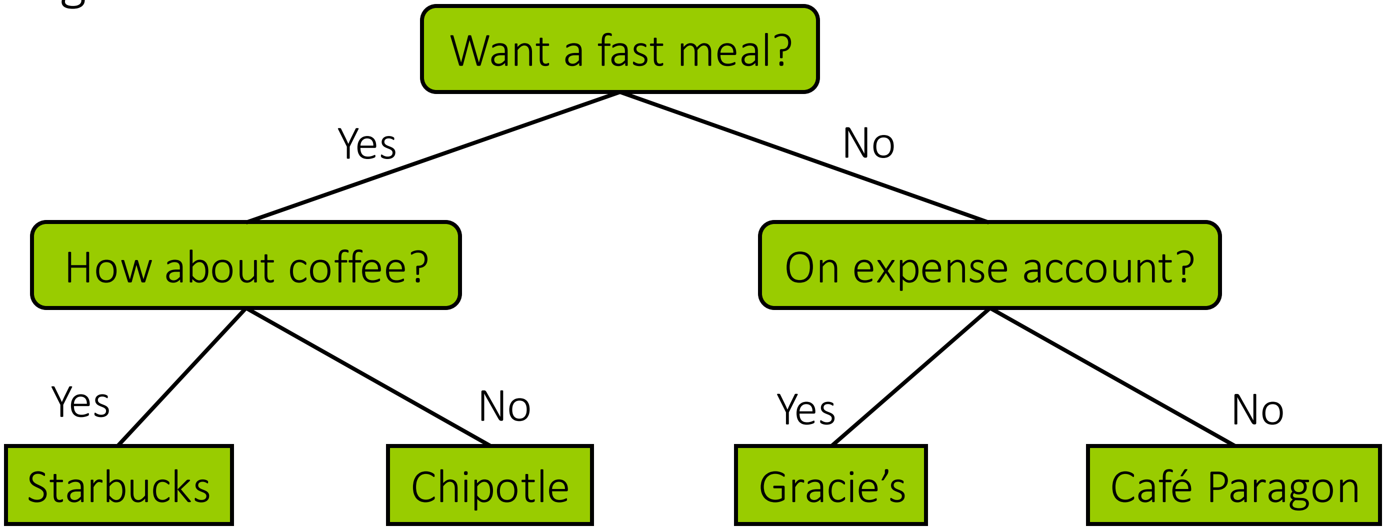
Arithmetic Expression Tree

- 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

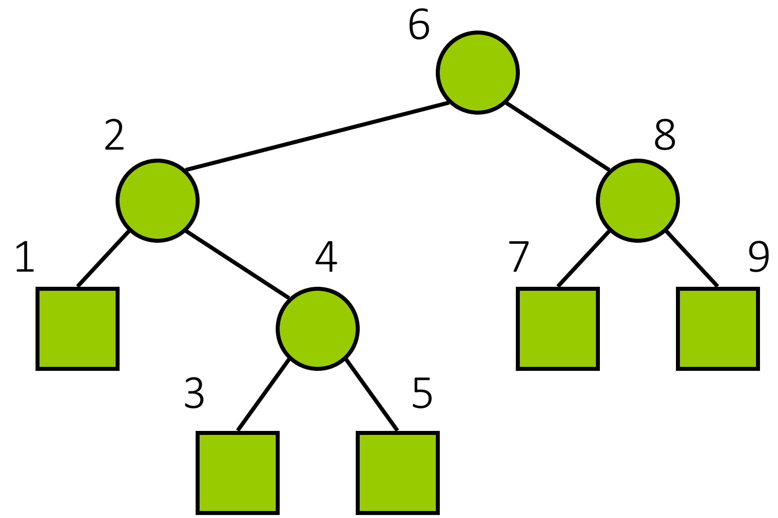
- Binary tree associated with a decision process
  - Internal nodes: questions with yes/no answer
  - External nodes: decisions
- Example:
  - Dining decision



Inorder Traversal

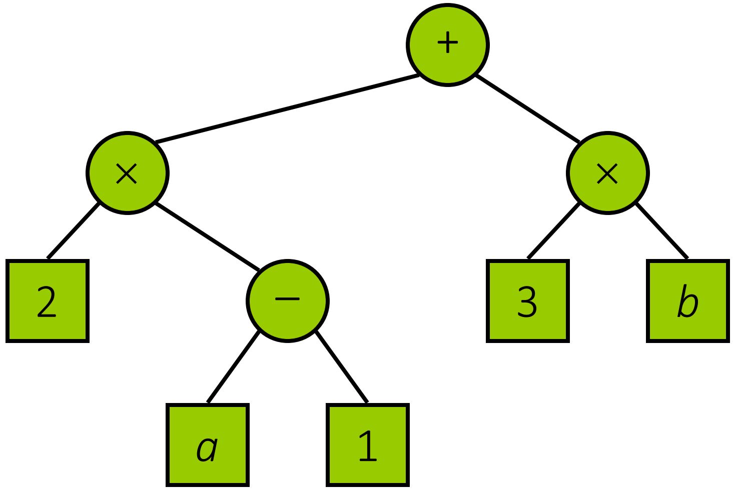
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - $x(v)$ = inorder rank of v
  - $y(v)$ = depth of v

```
Algorithm inOrder(v)
    if left(v) ≠ null then
        inOrder(left(v))
    visit(v)
    if right(v) ≠ null then
        inOrder(right(v))
```



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



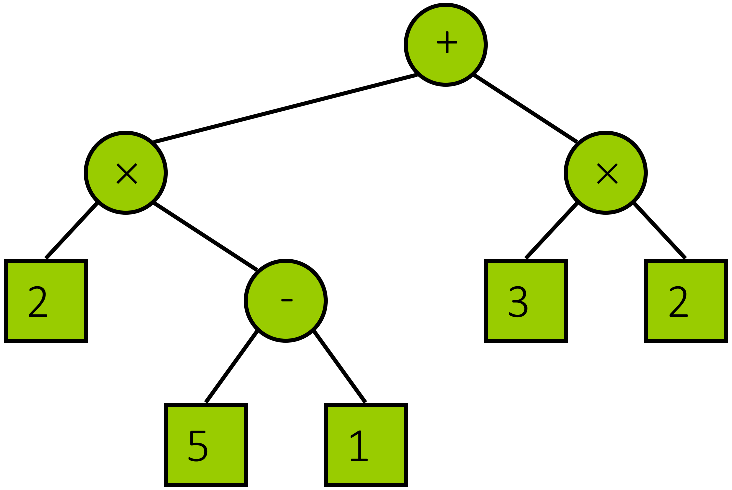
$((2 \times (a - 1)) + (3 \times b))$

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

Evaluate Arithmetic Expressions

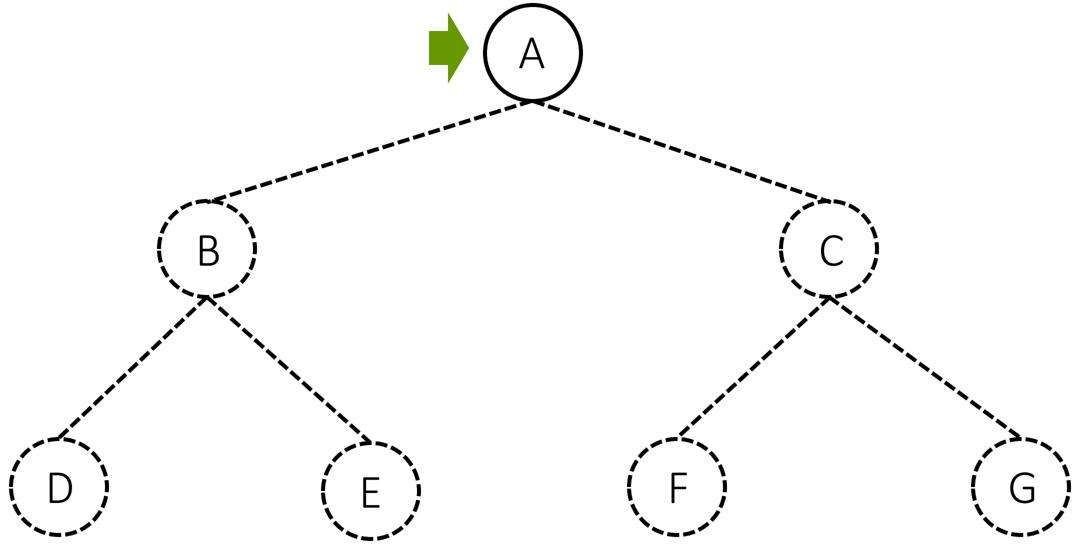
- Specialization of a postorder traversal
  - Recursive method returning the value of a subtree
  - When visiting an internal node, combine the values of the subtrees

```
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
```



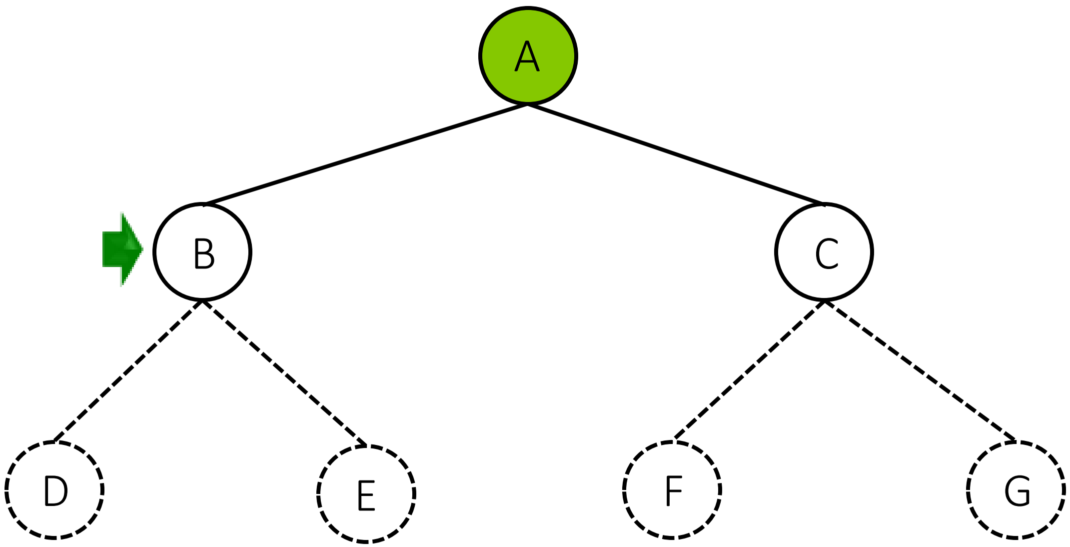
Breadth-first Search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



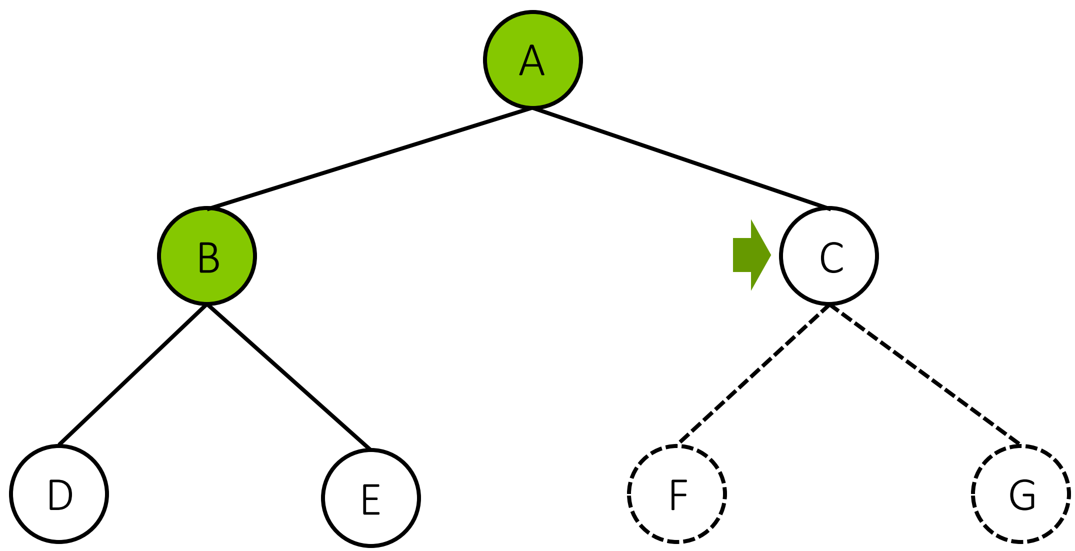
Breadth-first Search (cont'd.)

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



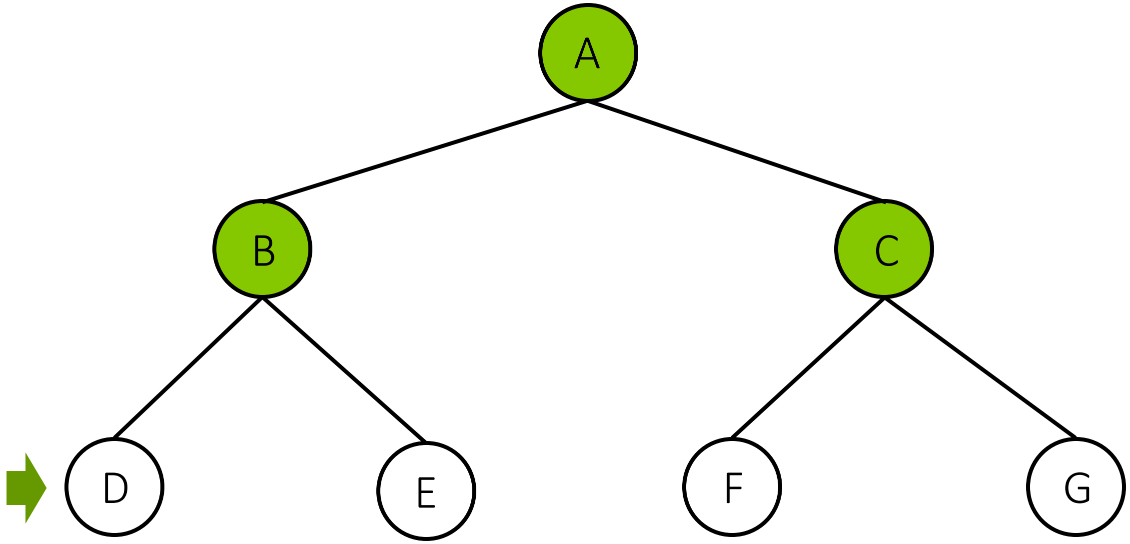
Breadth-first Search (cont'd.)

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

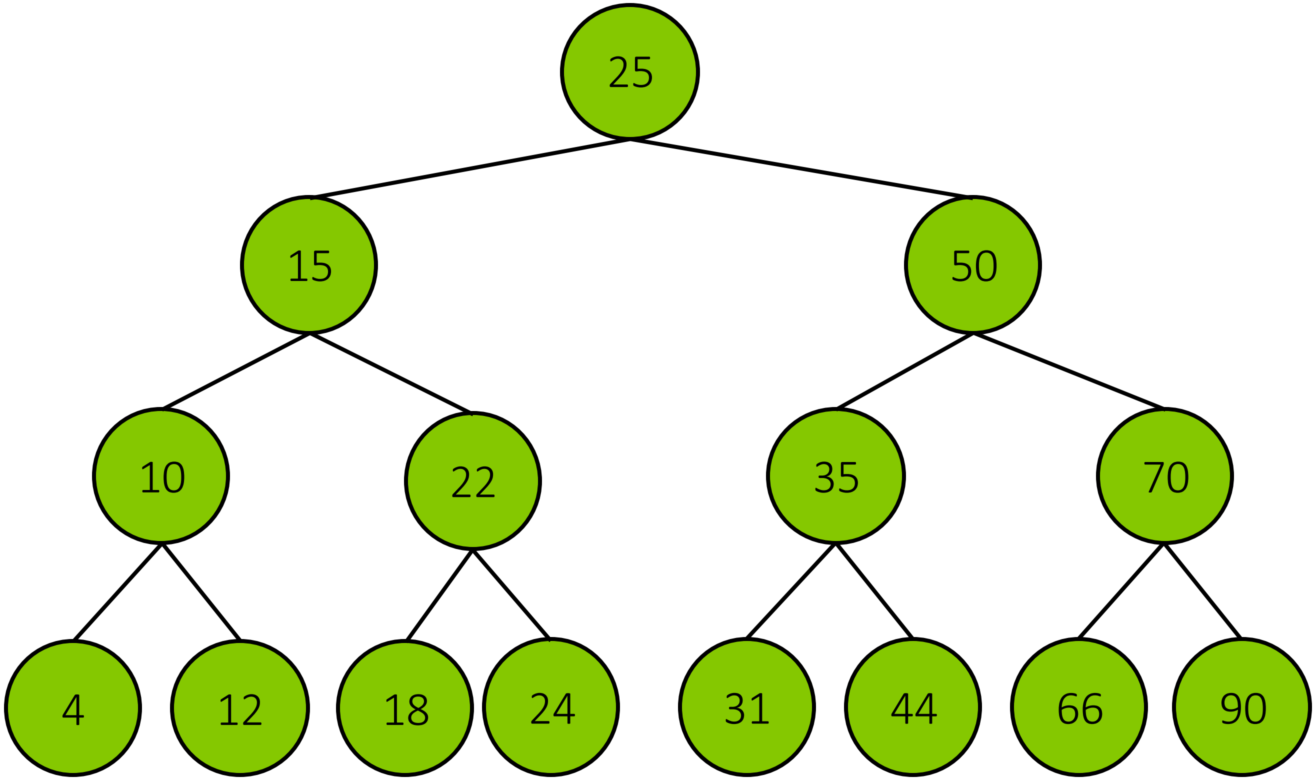


Breadth-first Search (cont'd.)

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Exercise 7.1



Preorder:

Postorder:

Inorder:

Breadth-first:

Summary

- Tree
- Binary tree
- Implementation
  - Linked-based structure
  - Array-based structure
- Tree traversal
  - Preorder traversal
  - Postorder traversal
  - Inorder traversal
  - Breadth-first search