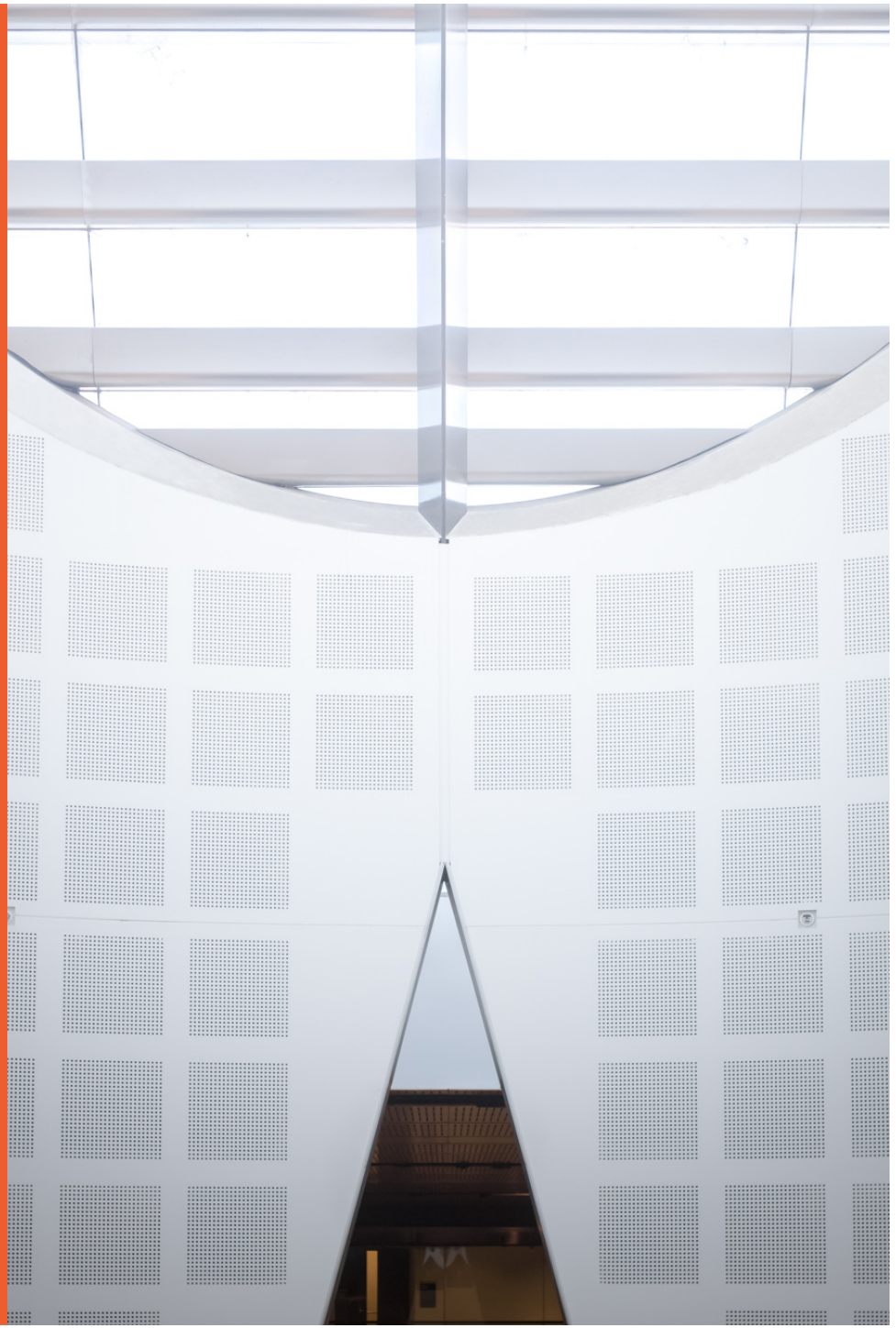
INFO1105/1905
Data Structures
Week 4: Trees
see textbook chapter 8

Professor Alan Fekete
Dr John Stavrakakis
School of Information Technologies

using material from the textbook
and A/Prof Kalina Yacef



THE UNIVERSITY OF SYDNEY

Copyright warning

- These slides contain material from the textbook (Goodrich, Tamassia & Goldwasser)
  - Data structures and algorithms in Java (5^{th} & 6^{th} edition)
- With modifications and many additions from the University of Sydney

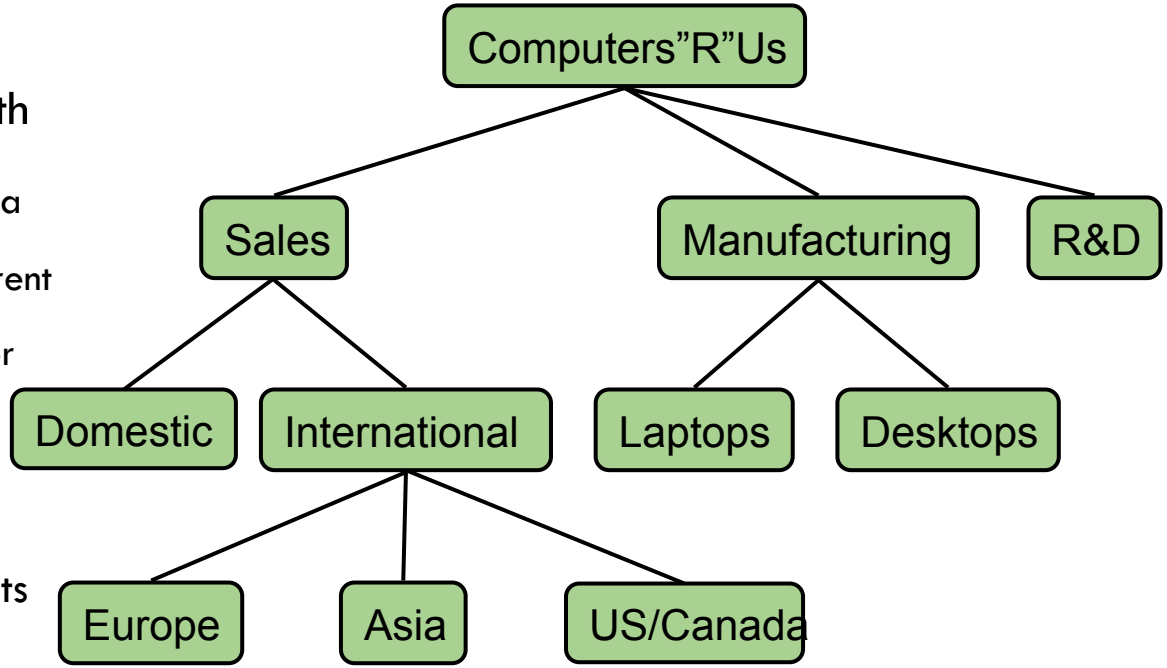
- The slides are a guide or overview of some big ideas
  - Students are responsible for knowing what is in the referenced sections of the textbook, not just what is in the slides

Outline

- Trees
  - Terminology
  - Tree ADT
  - Tree traversal algorithms
- Binary Trees
  - Terminology, properties
  - BinaryTree ADT
  - More tree traversals
- Implementing trees
- Recursive code on trees

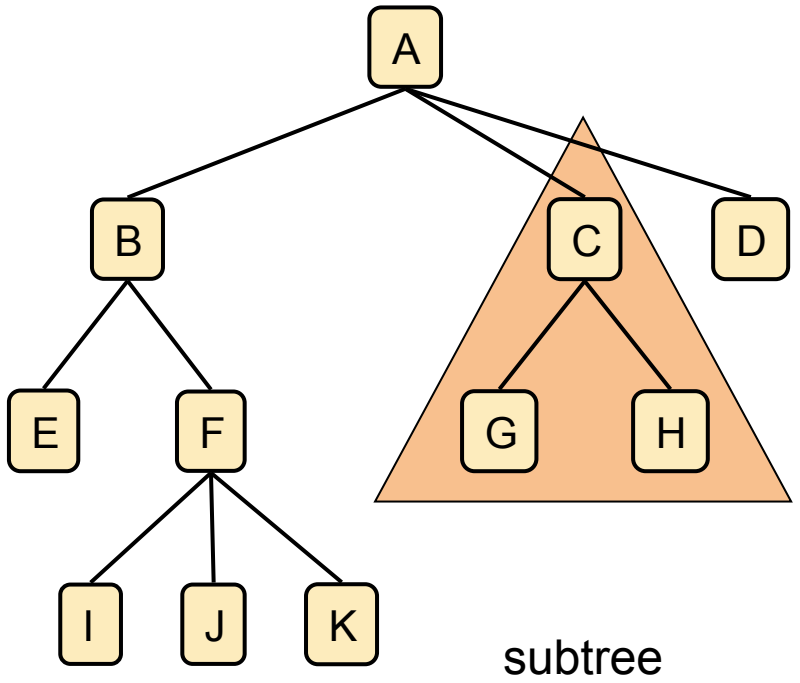
What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
  - if n is parent of m, then m is a child of n
  - a node has at most ONE parent in a tree
  - a node can have zero, one or more children
- Applications:
  - Organisation charts
  - File systems
  - Programming environments



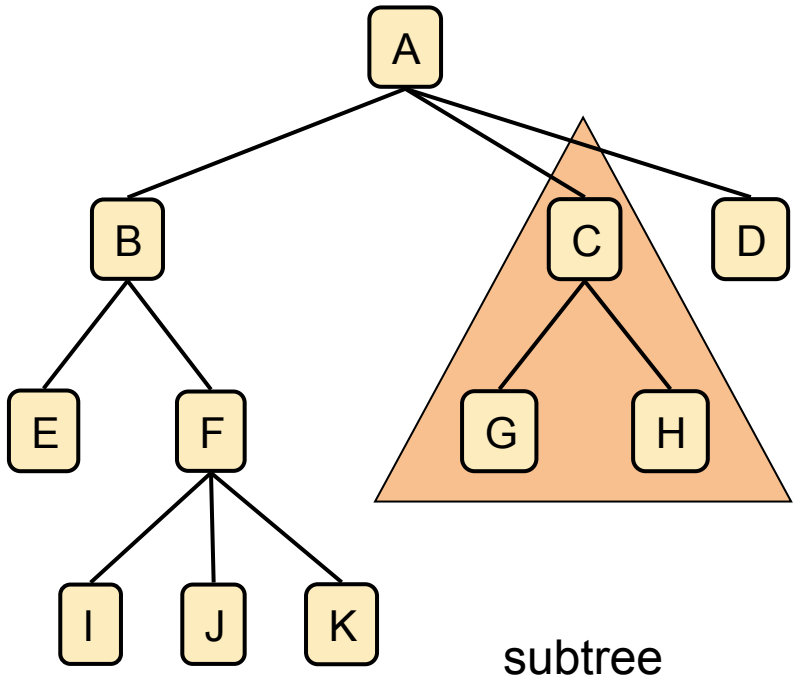
Tree Terminology

- 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 any children (E, I, J, K, G, H, D)
- Ancestors of a node: itself, parent, grandparent, grand-grandparent, etc.
  - eg ancestors(F)={F,B,A}
- Descendant of a node: itself, any child, any grandchild, grand-grandchild, etc.
- Two nodes with the same parent are siblings (eg B and D)
- **Subtree**: tree consisting of a node and its descendants (eg {C,G,H})



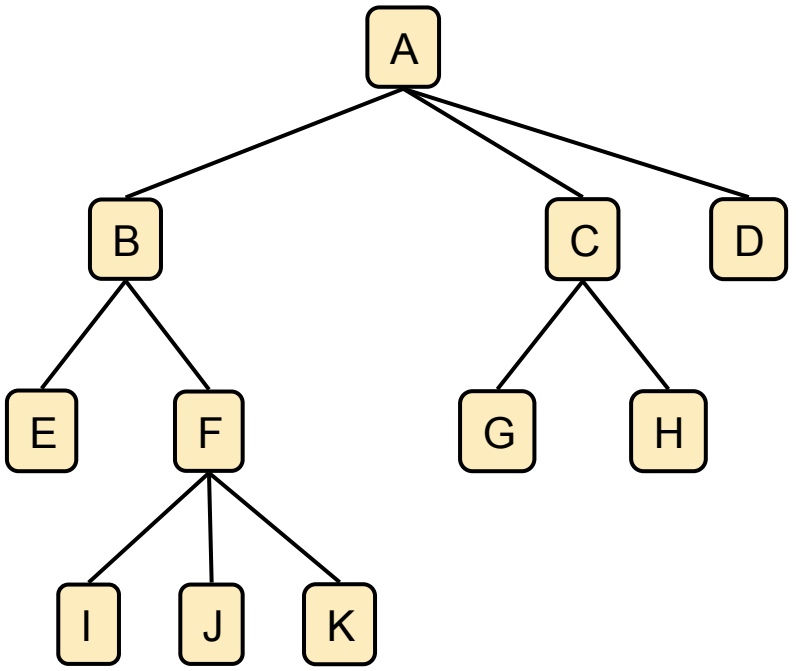
Tree Terminology

- Depth of a node: number of ancestors not including itself
  - eg depth(F) = 2
- Level: the set of nodes with a given depth
  - eg {E,F,G,H} are level 2
  - usually drawn in a horizontal arrangement
- Height of a tree: maximum depth of any node (3)



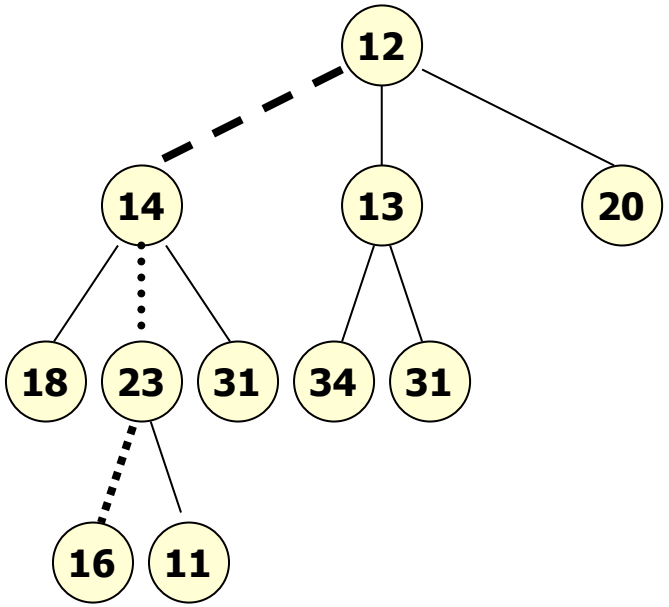
Tree Terminology (cont)

- Edge: pair of nodes (u,v) such that one is the parent of the other
  - Eg (B,F)
- Path: sequence of nodes such that 2 consecutive nodes in the sequence have an edge
  - Eg: A,B, F, K
  - Eg B,F,J



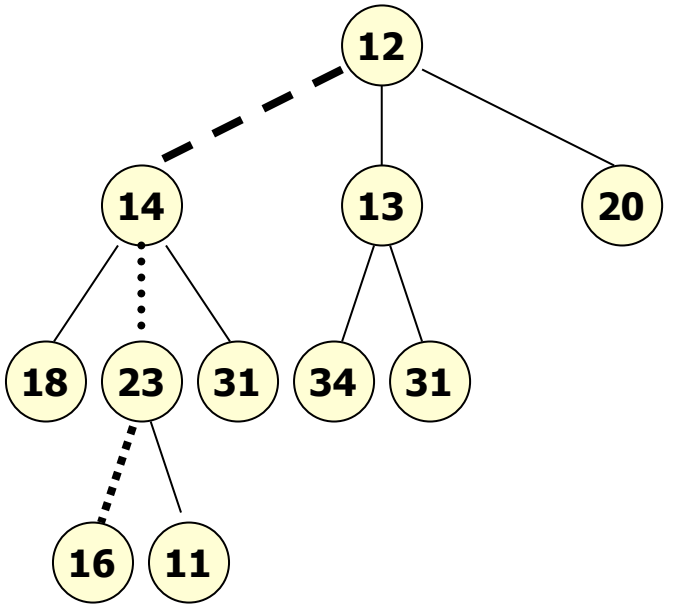
Examples

- Node14 has depth …
- The tree has height …
- Subtree rooted at node 14 has height …
- Any subtree from a leaf has height …
- The root has depth …

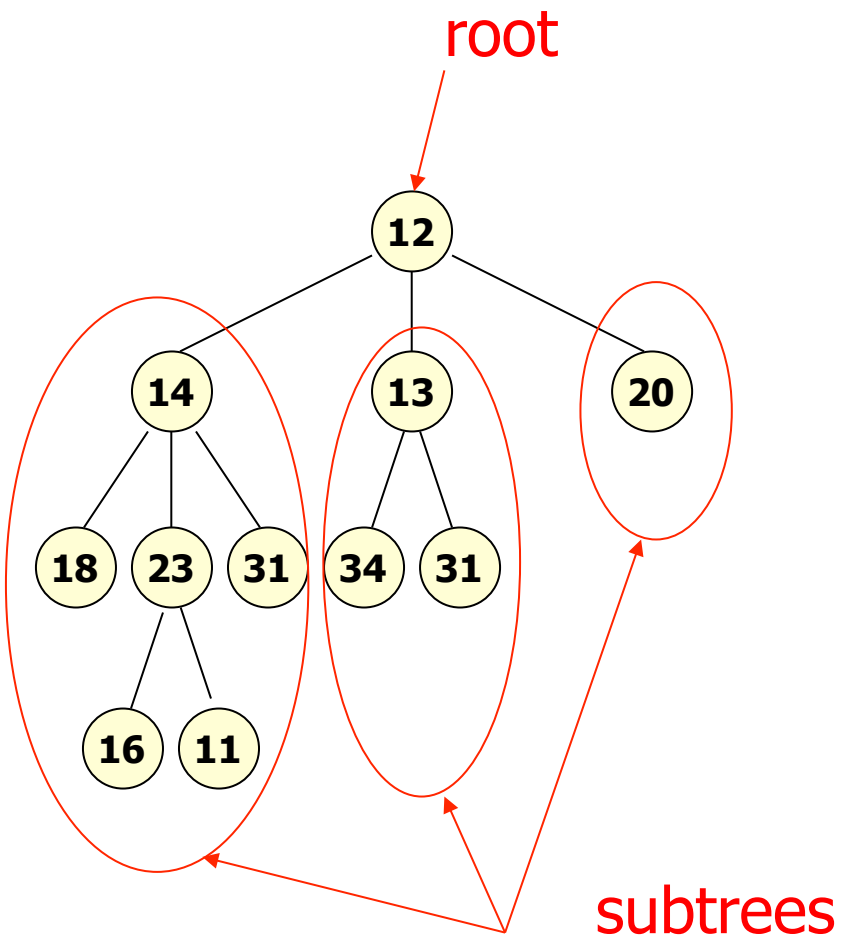


Examples

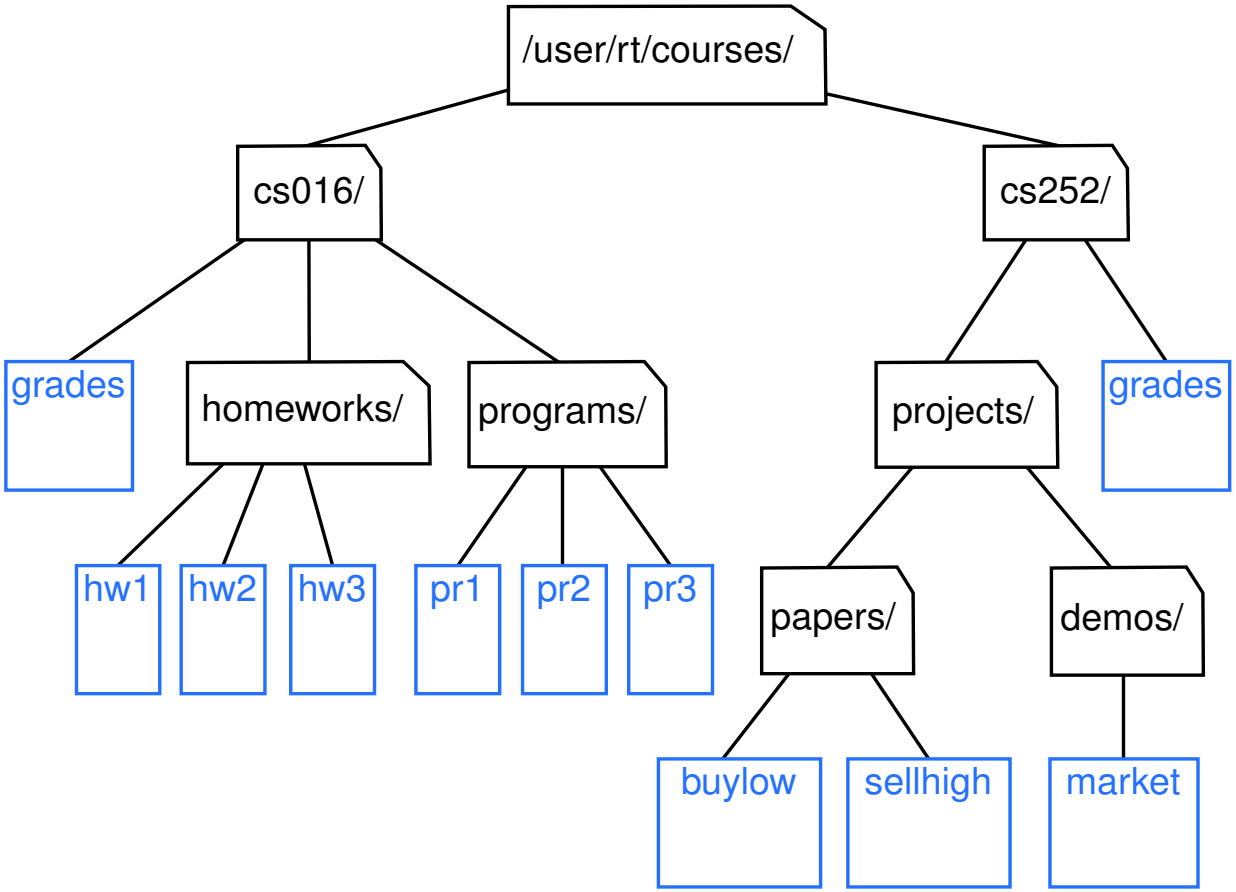
- Node14 has depth 1
- The tree has height 3
- Subtree rooted at node 14 has height 2
- Any subtree from a leaf has height 0
- The root has depth 0



- Different subtrees can have different shapes
- Each node can store a value or an object



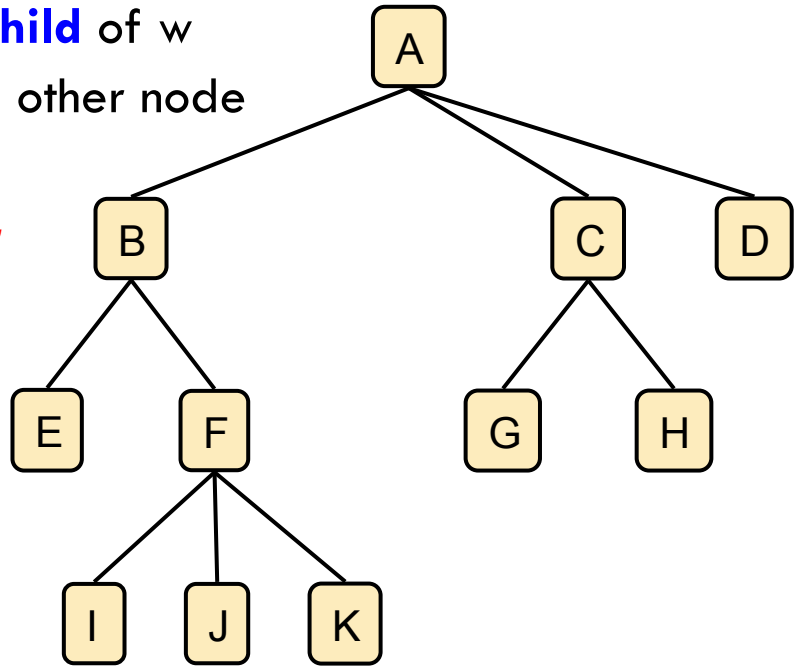
Application: file structure on O/S



A formal tree definition

- Tree T: set of **nodes** storing elements such that nodes have **parent-child** relationship with following properties:
  - If T is non empty, it has a special node, the **root** of T, that has no parent
  - Each node v of T different from the root has a unique **parent**
    - Every node with parent w is a **child** of w
  - There is a path from the root to any other node

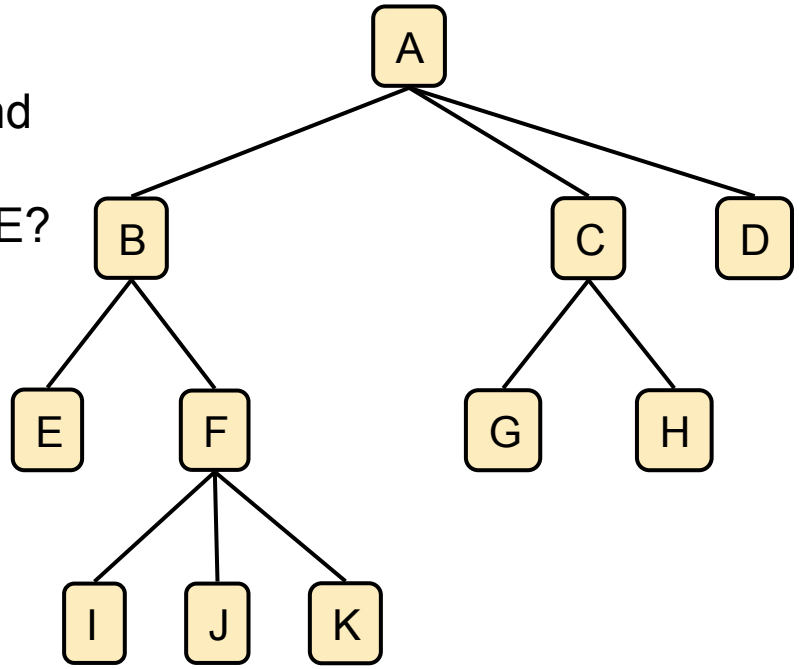
Warning: this last condition is important, and it is missing in the textbook!



Some facts

- If node X is an ancestor of node Y, then Y is a descendant of X.
- Every node is a descendant of the root
- There are nodes where neither is an ancestor of the other
- For every pair of nodes, there is at least one common ancestor (a node that is ancestor of both)

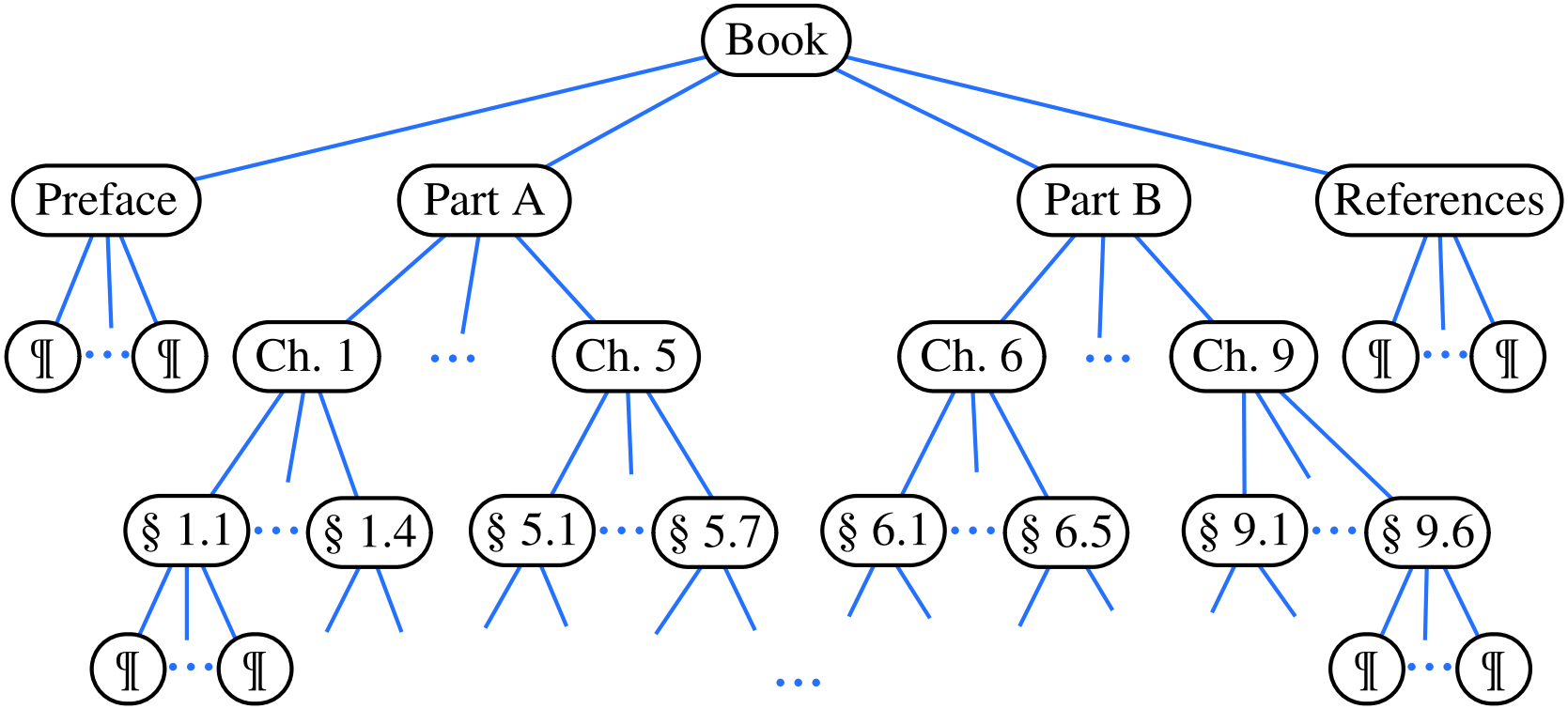
Note: the set of common ancestors of n and m forms a path starting from the root
What are the common ancestors of J and E?



Ordered Trees

- Sometimes order in the siblings matter
- A tree is ordered if there is a linear ordering for the children of each node
  - In diagram, siblings usually arranged left to right
- Example: components of a book

Example of an ordered tree



What about the organisation chart? The portion of file system?
A family genealogy tree?

Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()
- Access 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 for Tree ADT

```
/** 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();
}
```

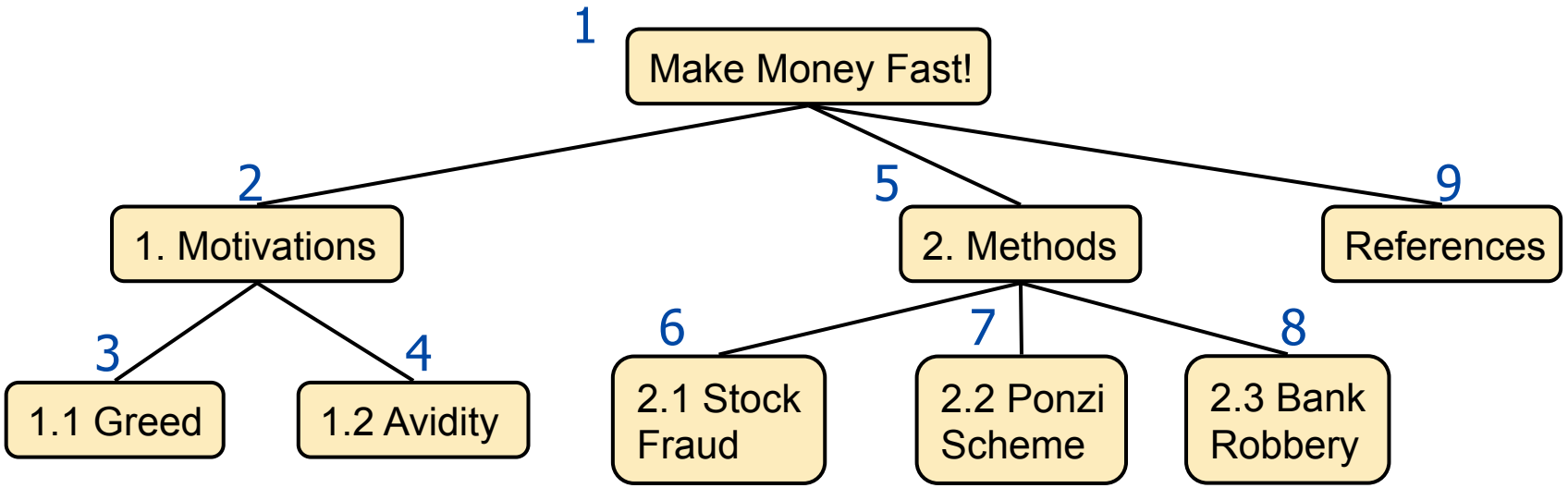
Traversing trees

- A traversal visits the nodes of a tree in a systematic manner
- We will see 2 traversals for now:
  - Preorder
  - Postorder

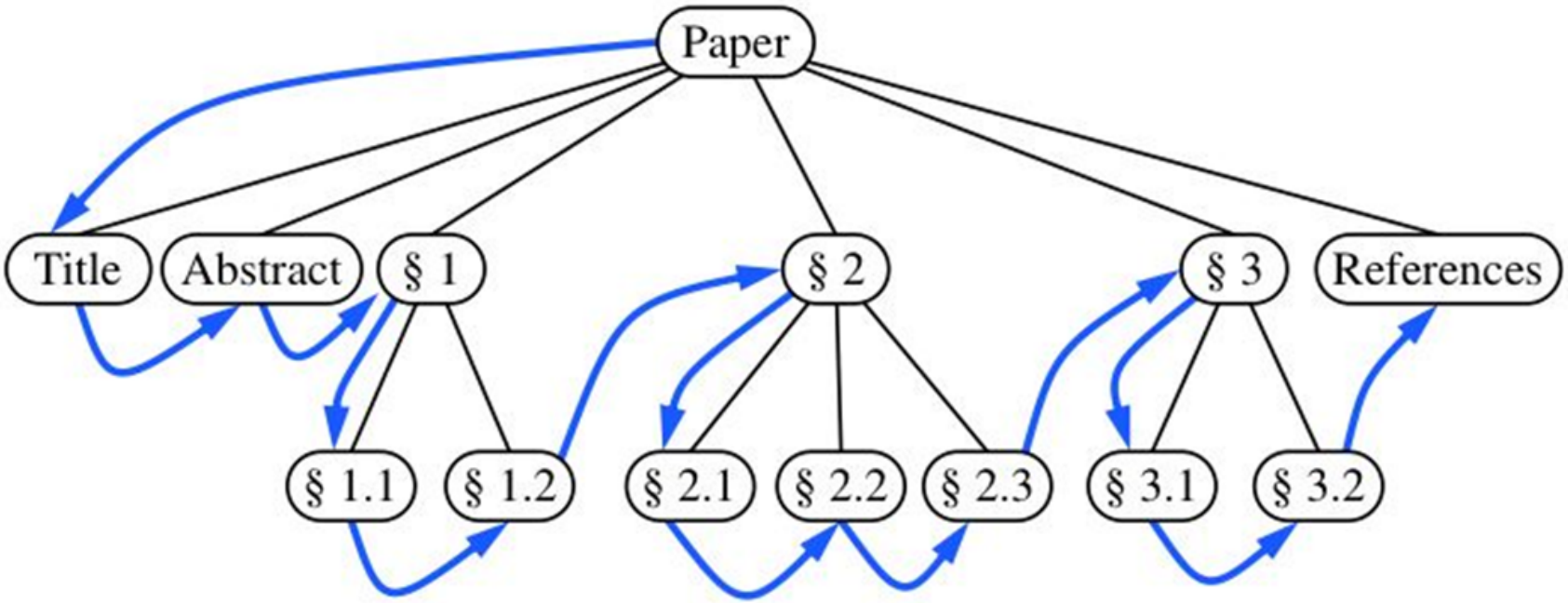
Preorder Traversal

- 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)



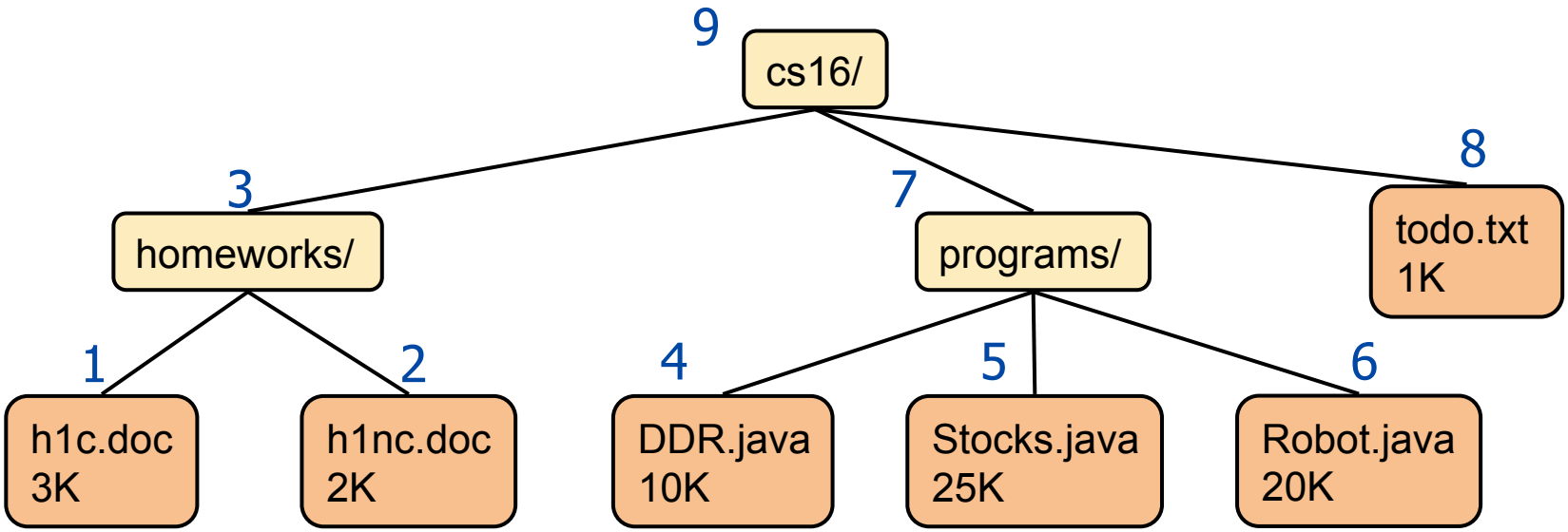
Preorder traversal



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

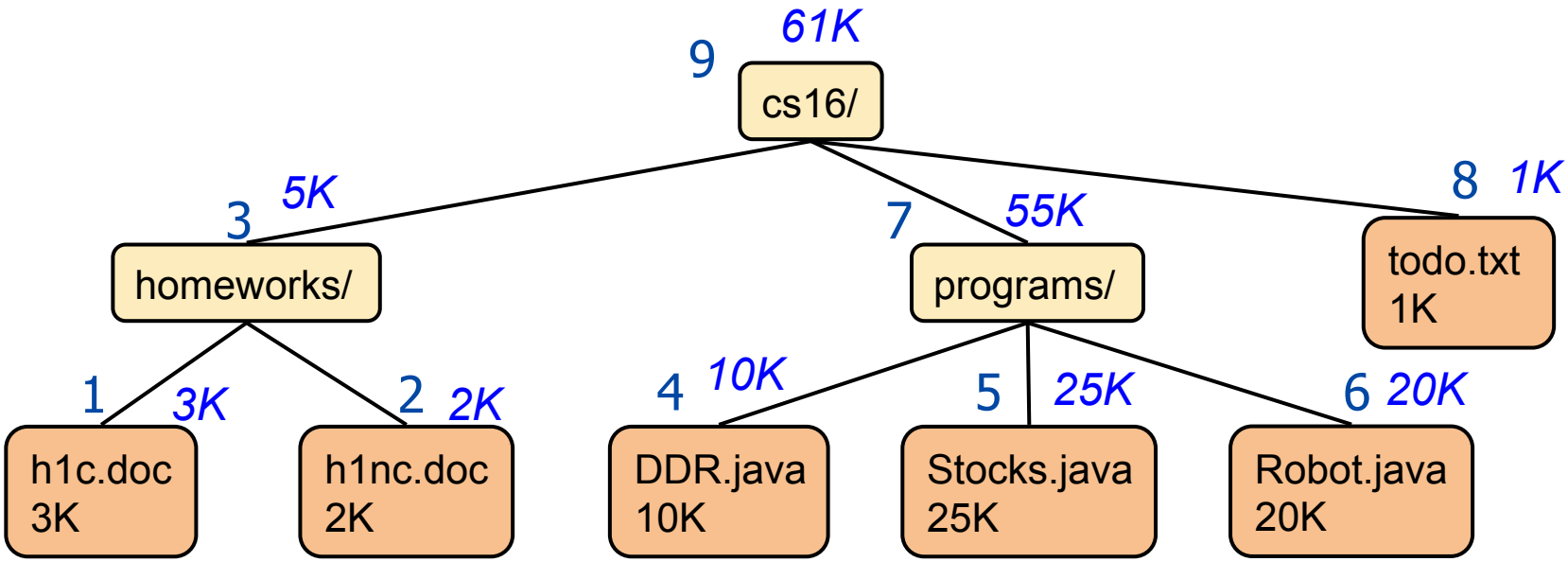
Algorithm ***postOrder(v)***
  for each child ***w*** of ***v***
    postOrder ***(w)***
  visit(v)



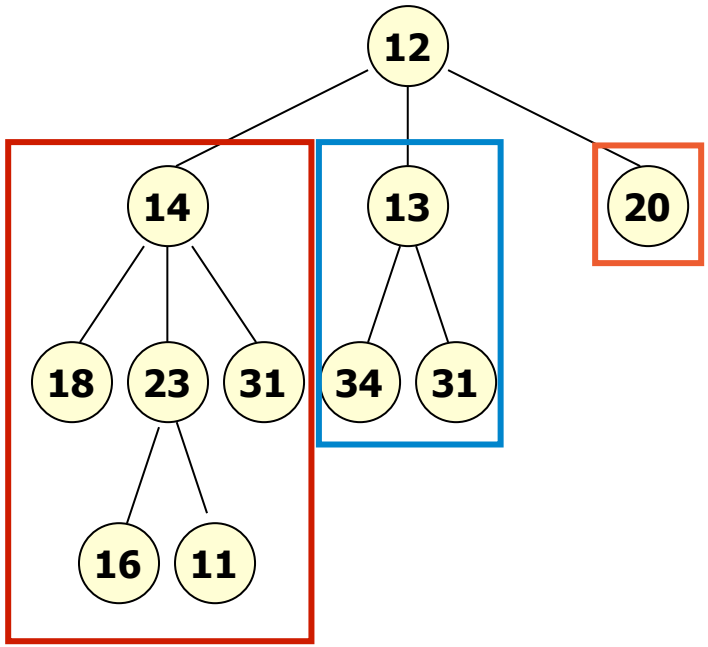
Postorder Traversal

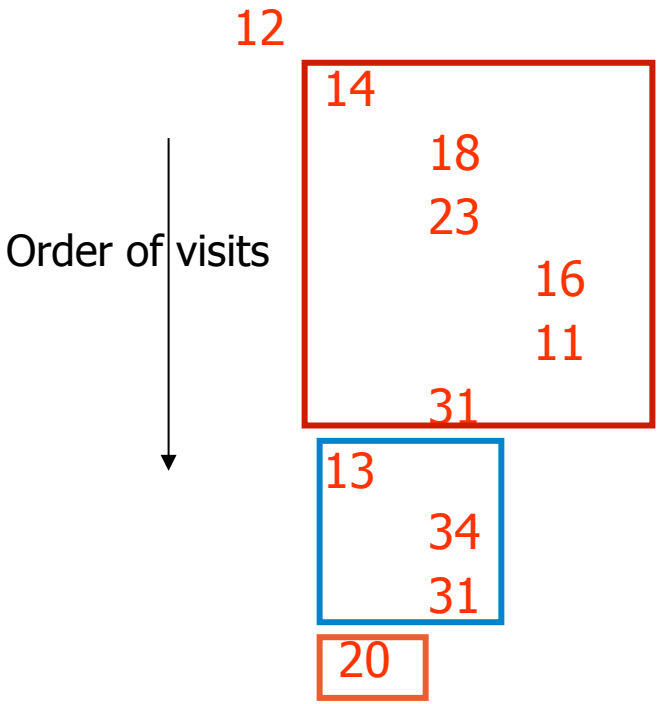
- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

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



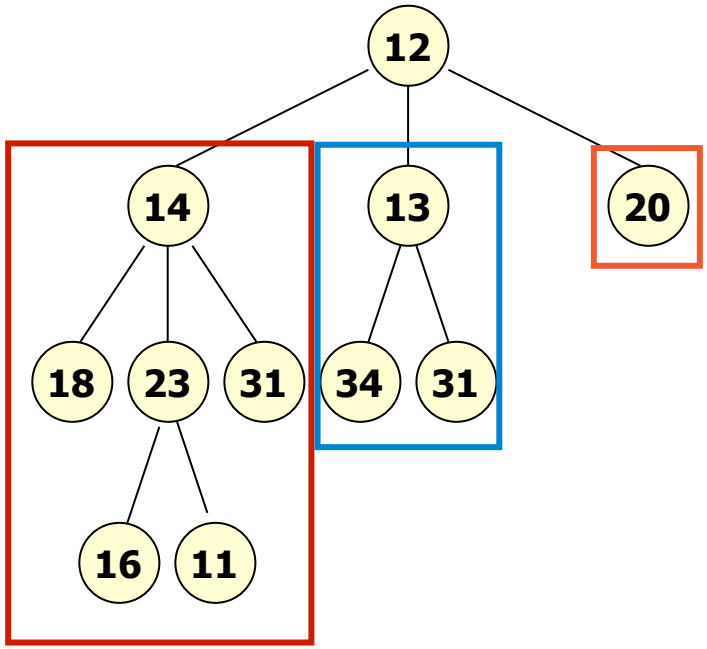
Traversing in preorder

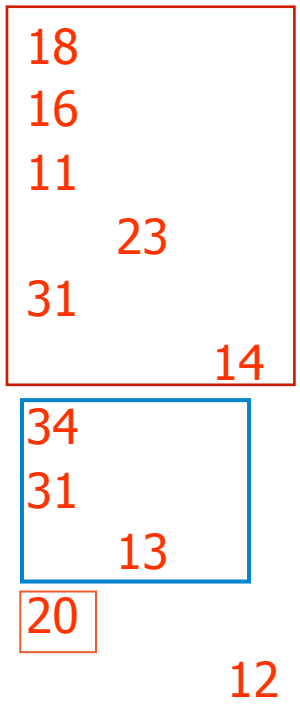




Preorder traversal of subtree

Traversing in postorder





Warning: postorder traversal is not the same as reversal of preorder traversal! Even though the root is first in preorder, and last in postorder.

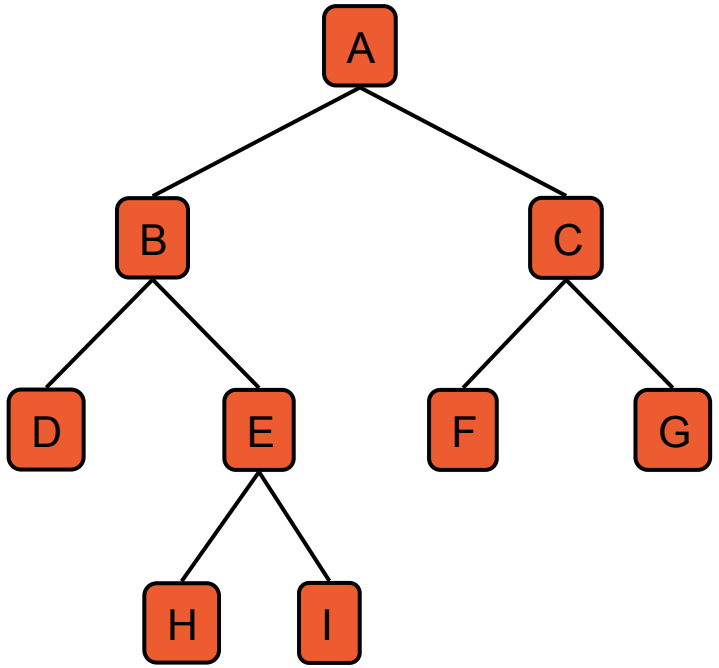
Outline

- Trees
  - Terminology
  - Tree ADT
  - Tree traversal algorithms
- Binary Trees
  - Terminology, properties
  - BinaryTree ADT
  - More tree traversals
- Implementing trees
- Recursive code on trees

Binary Trees

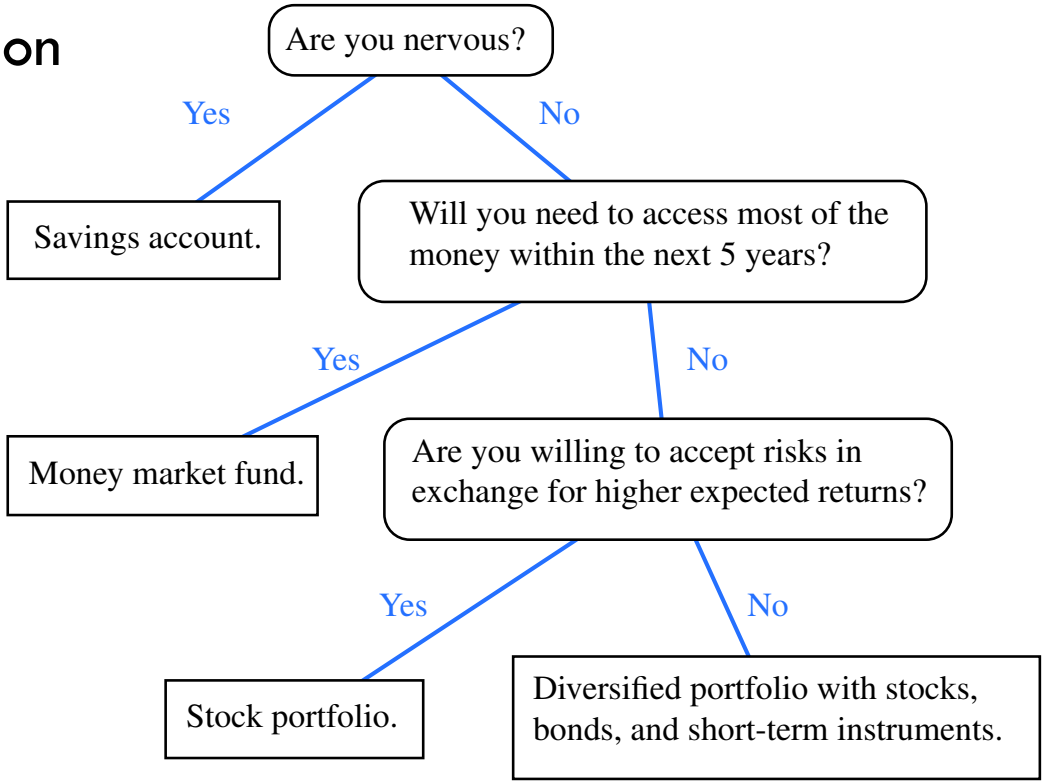
- A binary tree is an ordered tree with the following properties:
  - Each internal node has at most two children
  - Each child node is labeled as a left child or a right child
    - draw left child on lower level but further to left than its parent
    - draw right child on lower level but further to right than its parent
  - The children of a node are ordered with left before right (when there are two children)

- Applications
  - Arithmetic expressions
  - Decision processes
  - Searching (see next week)



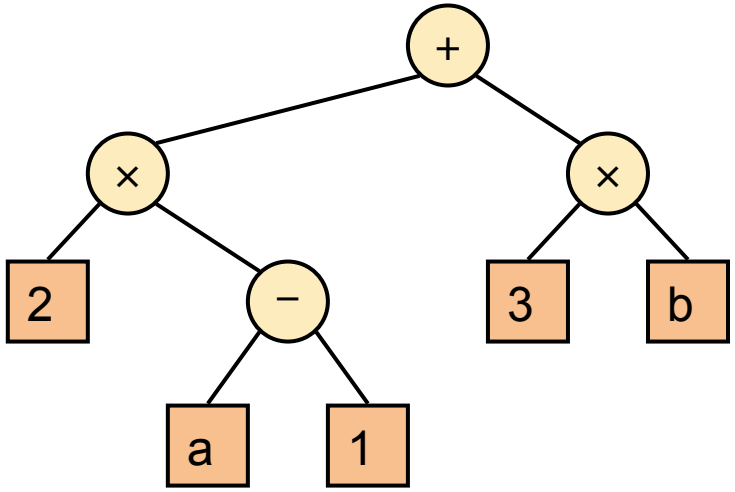
Decision Tree: an example of a binary tree

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



Arithmetic Expression Tree

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



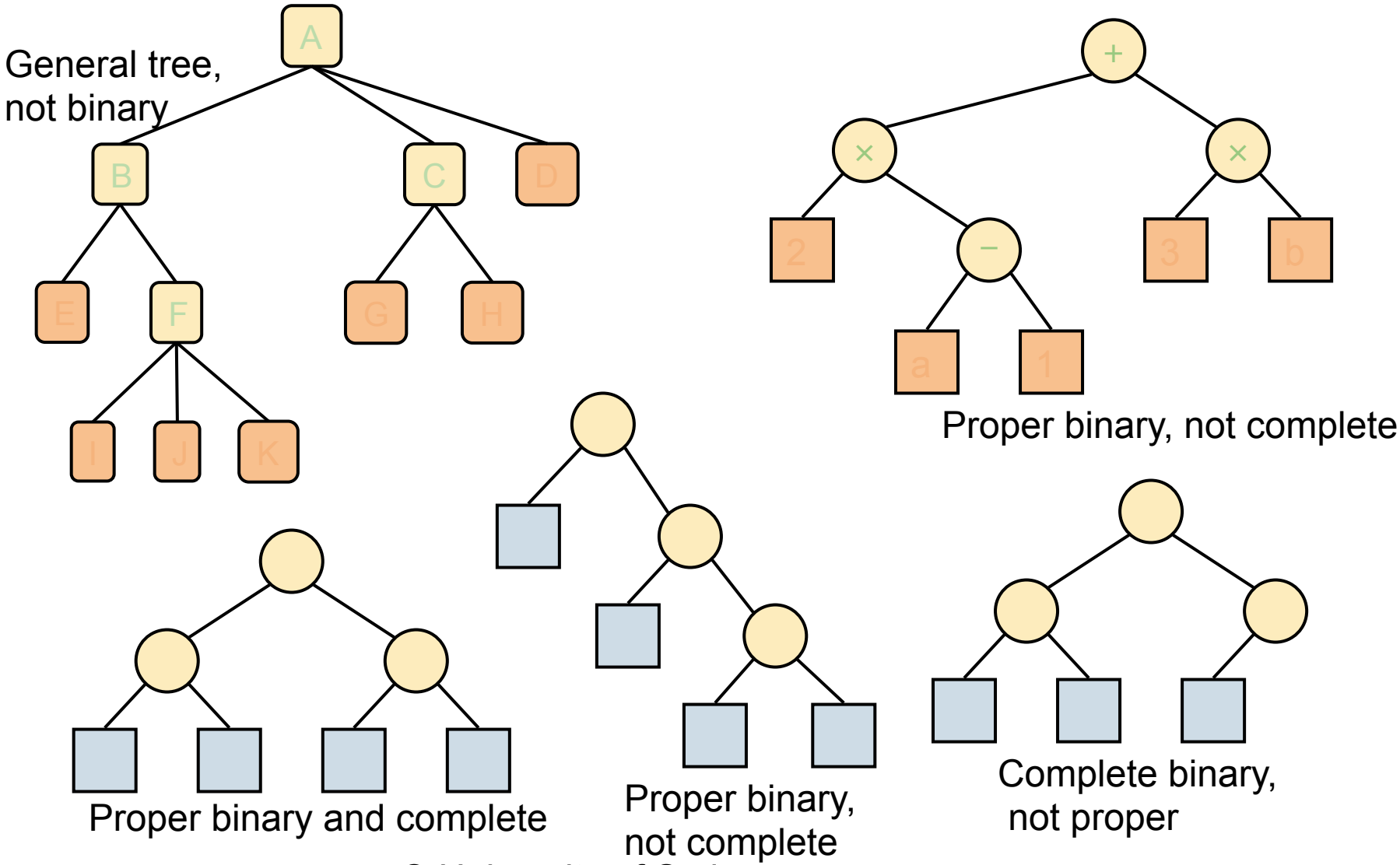
Alternative recursive definition of binary trees

- recursive definition of a binary tree
  - A binary tree is a set T of nodes such that either
    - T is empty, or
    - T is partitioned into three disjoint subsets:
      - A single node r, the root
      - The left subtree (this is possibly empty)
      - The right subtree (this is possibly empty)

More terminology

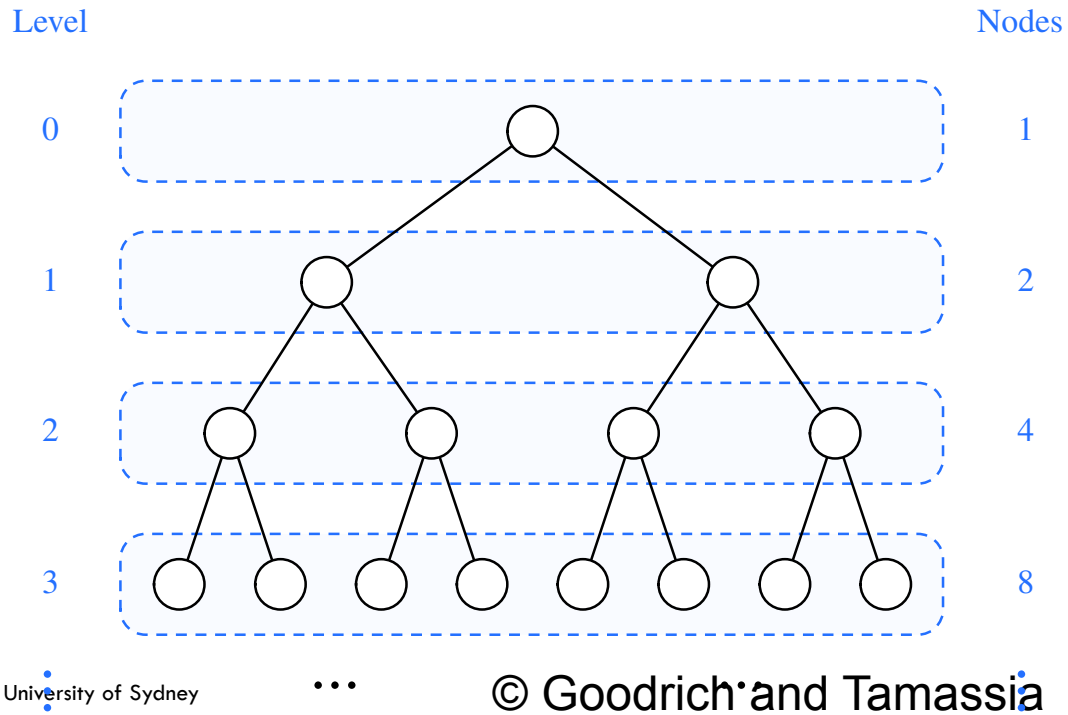
- Proper (or Full) binary trees : each node has exactly 0 or 2 children
  - Recursive definition of a full binary tree:
    - A leaf node is a proper binary tree
    - A tree is a proper binary tree if both subtrees rooted at its children are proper binary trees
- Complete binary trees: each level is completely filled, except possibly the last level, which must be filled from left to right
  - A binary tree T of height h is complete if
    - All nodes at level h – 2 and above have two children each, and
    - When a node at level h – 1 has children, all nodes to its left at the same level have two children each, and
    - When a node at level h – 1 has one child, it is a left child
- Balanced binary trees *(will be important later)*
  - A binary tree is balanced if the height of any node's right subtree differs from the height of the node's left subtree by no more than 1

Are these trees binary? proper? complete?



Properties of Binary Trees

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



Properties:

At level k, maximum 2^k nodes

Total nodes, max $\Sigma 2^k = 2^{h+1} - 1$

Therefore:

$$h+1 \leq n \leq 2^{h+1} - 1$$

$$1 \leq e \leq 2^h$$

$$h \leq i \leq 2^h - 1$$

$$h \leq (n-1)$$

$$h \geq \log_2 (n+1) - 1$$

For many algorithms, the time or space cost is best considered to scale with h rather than with n

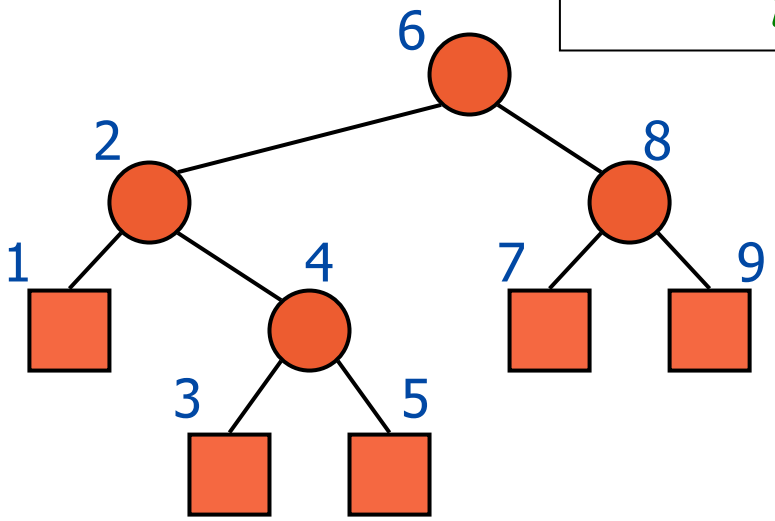
BinaryTree ADT

- 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)
- These 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

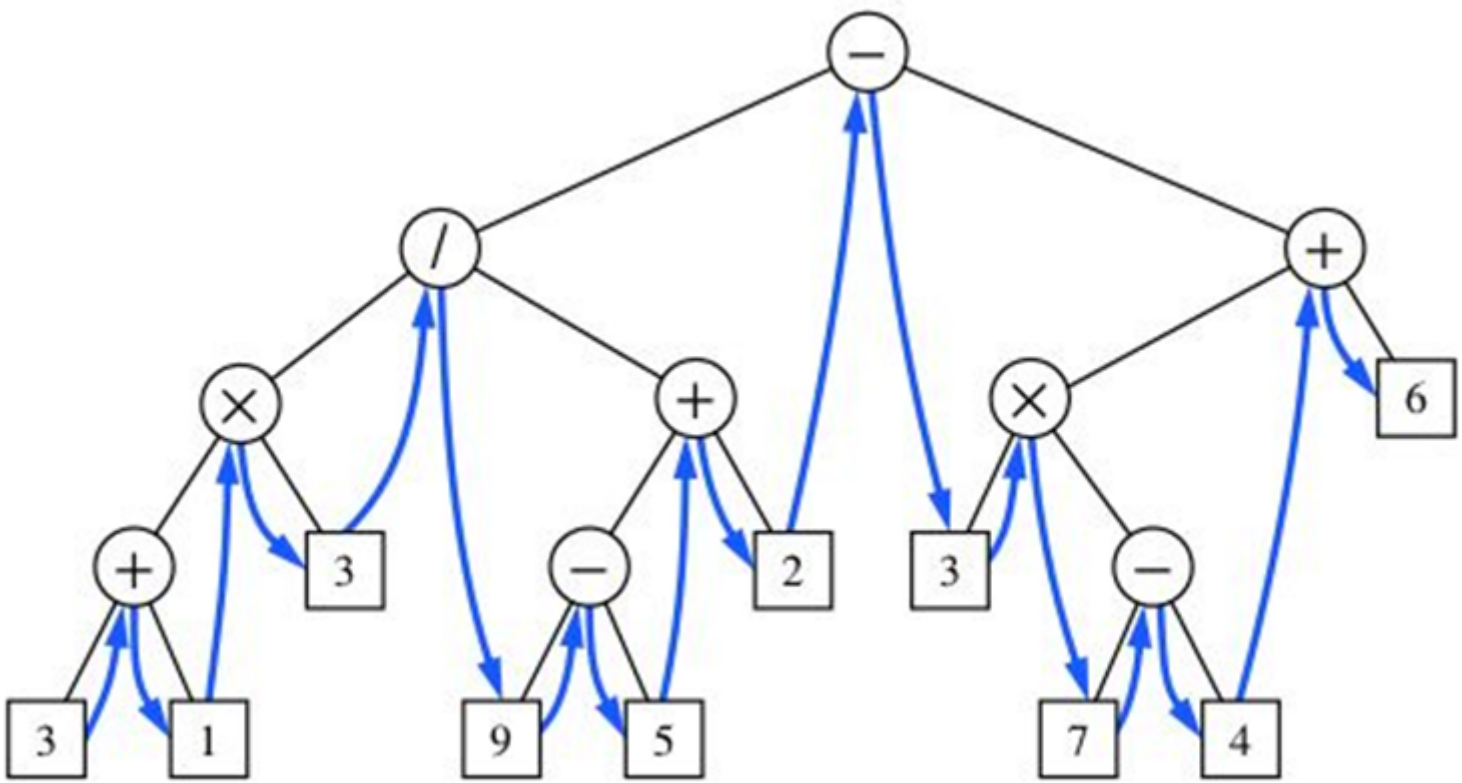
Another traversal for *Binary* Trees: Inorder

- Binary trees also have another traversal: inorder
- 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))

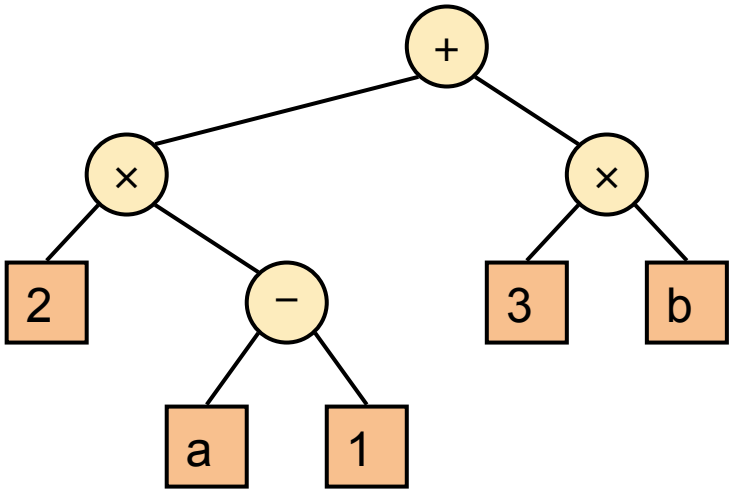


Inorder traversal on a binary tree



Inorder traversal for printing Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree



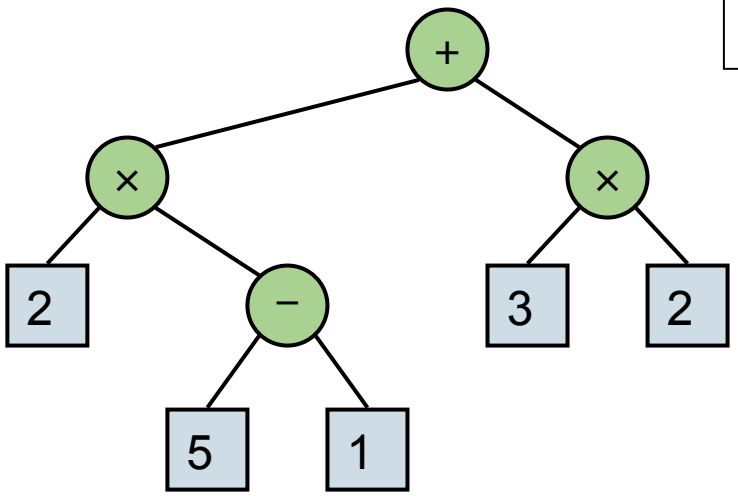
Algorithm *printExpression*(*v*)
  if *left*(*v*) ≠ **null**
    print("(")
    printExpression(*left*(*v*))
  print(*v.element* ())
  if *right*(*v*) ≠ **null**
    printExpression(*right*(*v*))
    print (")")

Output:
((2 × (a − 1)) + (3 × b))

Postorder traversal for evaluating Arithmetic Expressions

- Specialisation of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

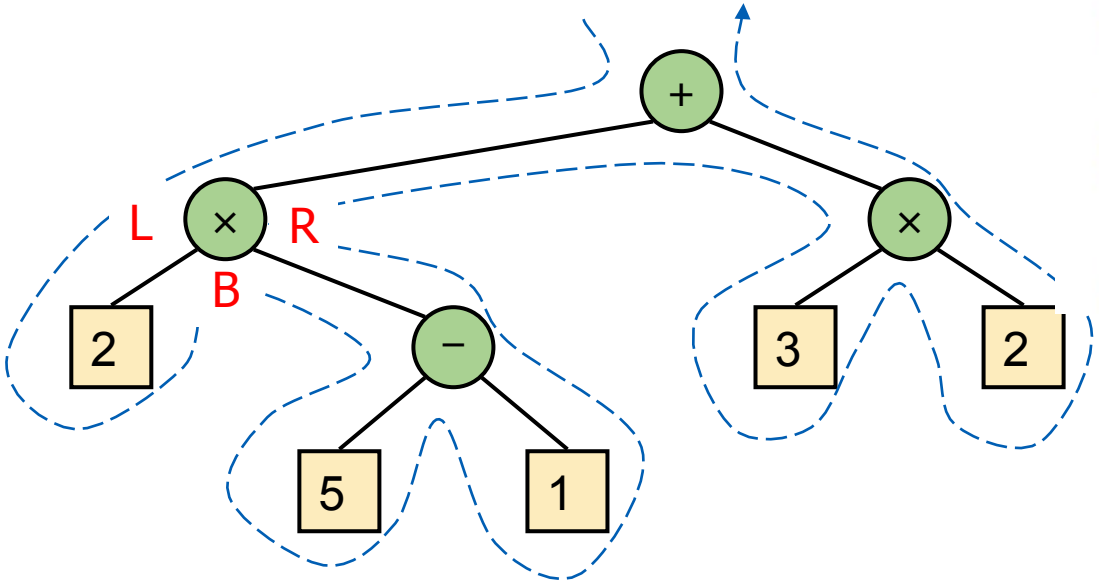
```
Algorithm evalExpr(v)
    if isExternal (v)
        return v.element ()
    else
        x ← evalExpr(leftChild (v))
        y ← evalExpr(rightChild (v))
        ◊ ← operator stored at v
        return x ◊ y
```



- Note the different traversal techniques for different needs !
  - Inorder for printing
  - Postorder to evaluate

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)



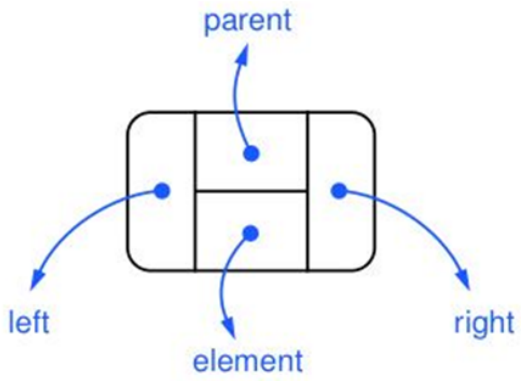
Algorithm eulerTourBinary(T, p):
perform the "pre visit" action for position p
if p has a left child lc **then**
eulerTourBinary(T, lc)
perform the "in visit" action for position p
if p has a right child rc **then**
eulerTourBinary(T, rc)
perform the "post visit" action for position p

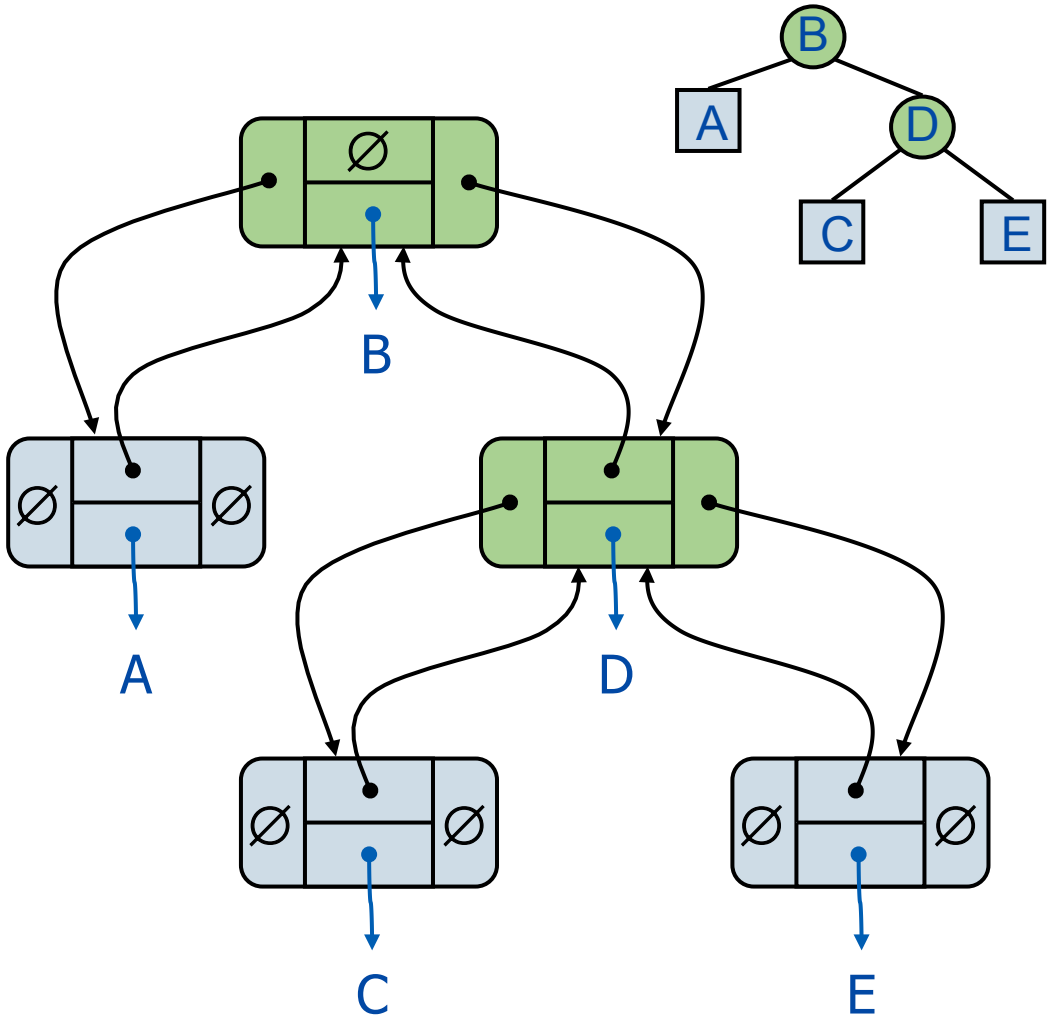
Outline

- Trees
  - Terminology
  - Tree ADT
  - Tree traversal algorithms
- Binary Trees
  - Terminology, properties
  - BinaryTree ADT
  - More tree traversals
- Implementing trees
  - Linked-lists
  - Array-based
- Recursive code on trees

Linked Structure for Binary Trees

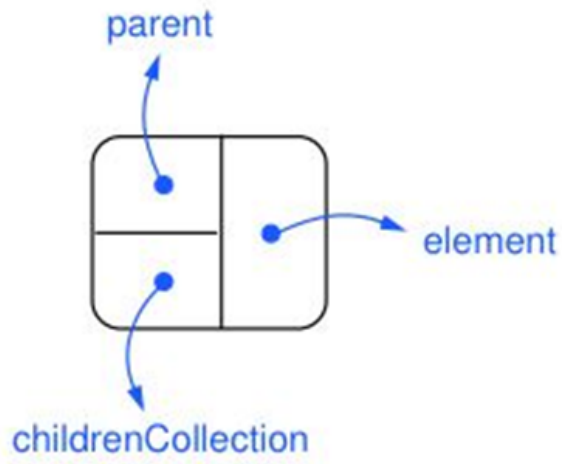
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT

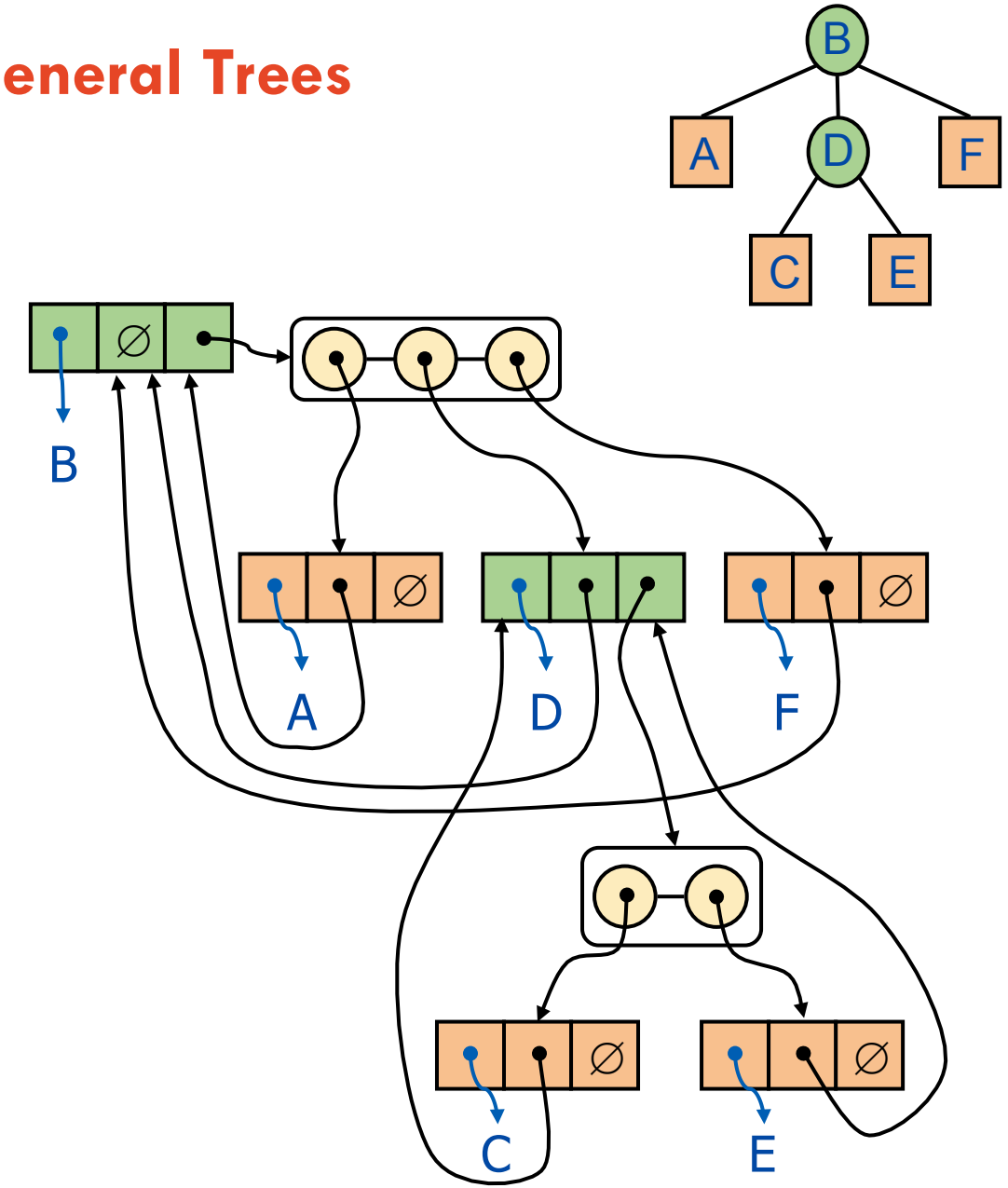




Linked Structure for general Trees

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





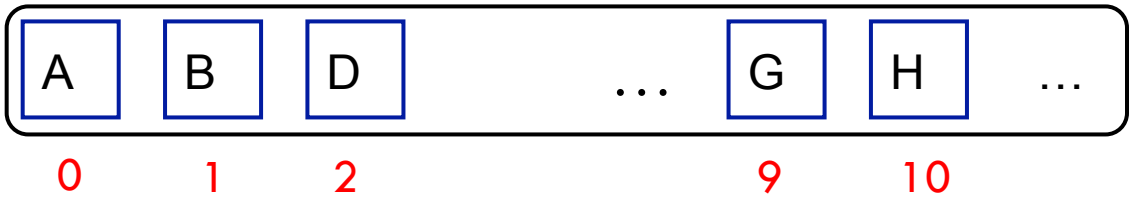
Space requirements and running times for LinkedList-based representation

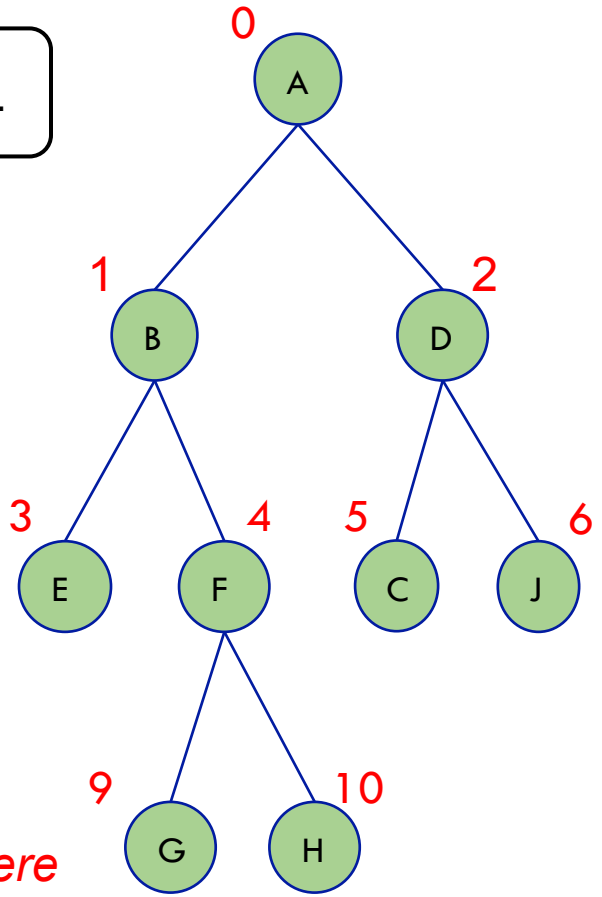
Space requirement : O(n)

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	$O(n)$

Array-Based Representation of Binary Trees

- Nodes are stored in an array A

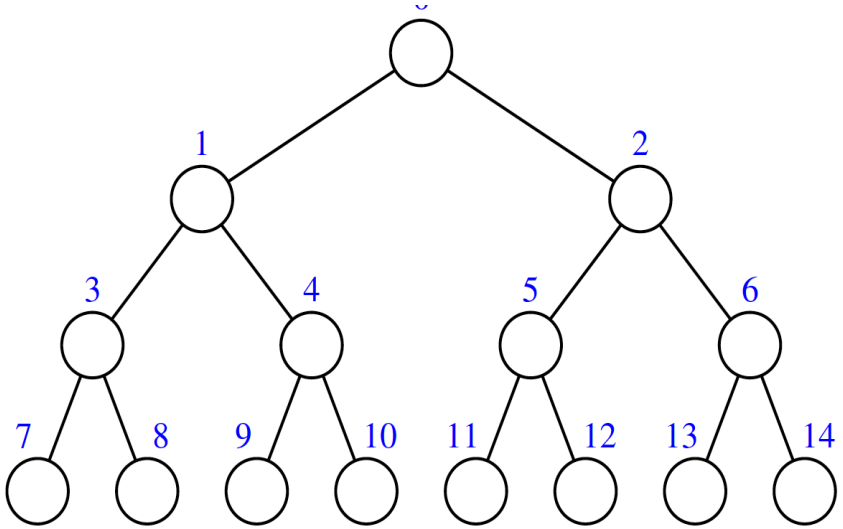


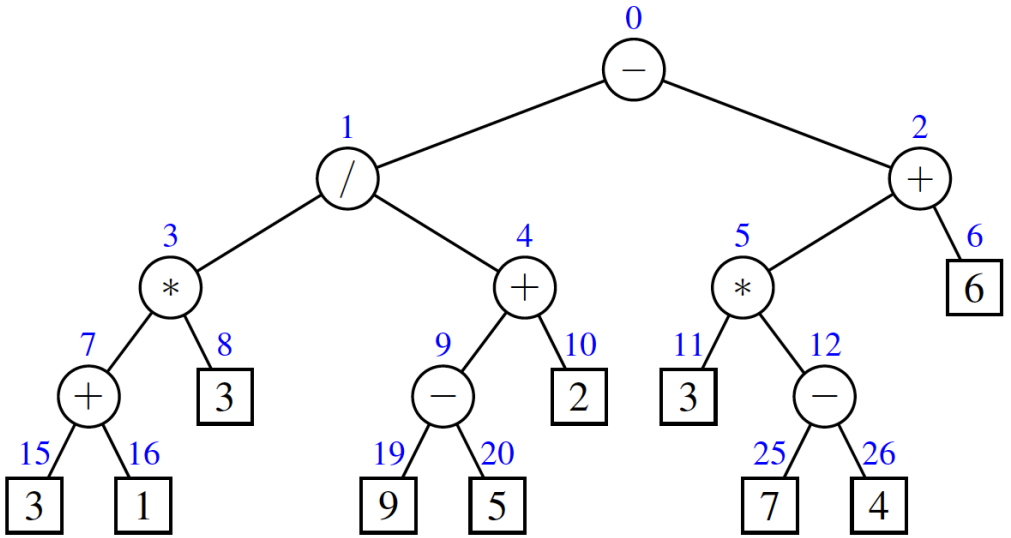


- Node v is stored at A[rank(v)]
  - rank(root) = 0
  - if node is the left child of parent(node),
    rank(node) = 2 x rank(parent(node)) +1
  - if node is the right child of parent(node),
    rank(node) = 2 x rank(parent(node)) + 2

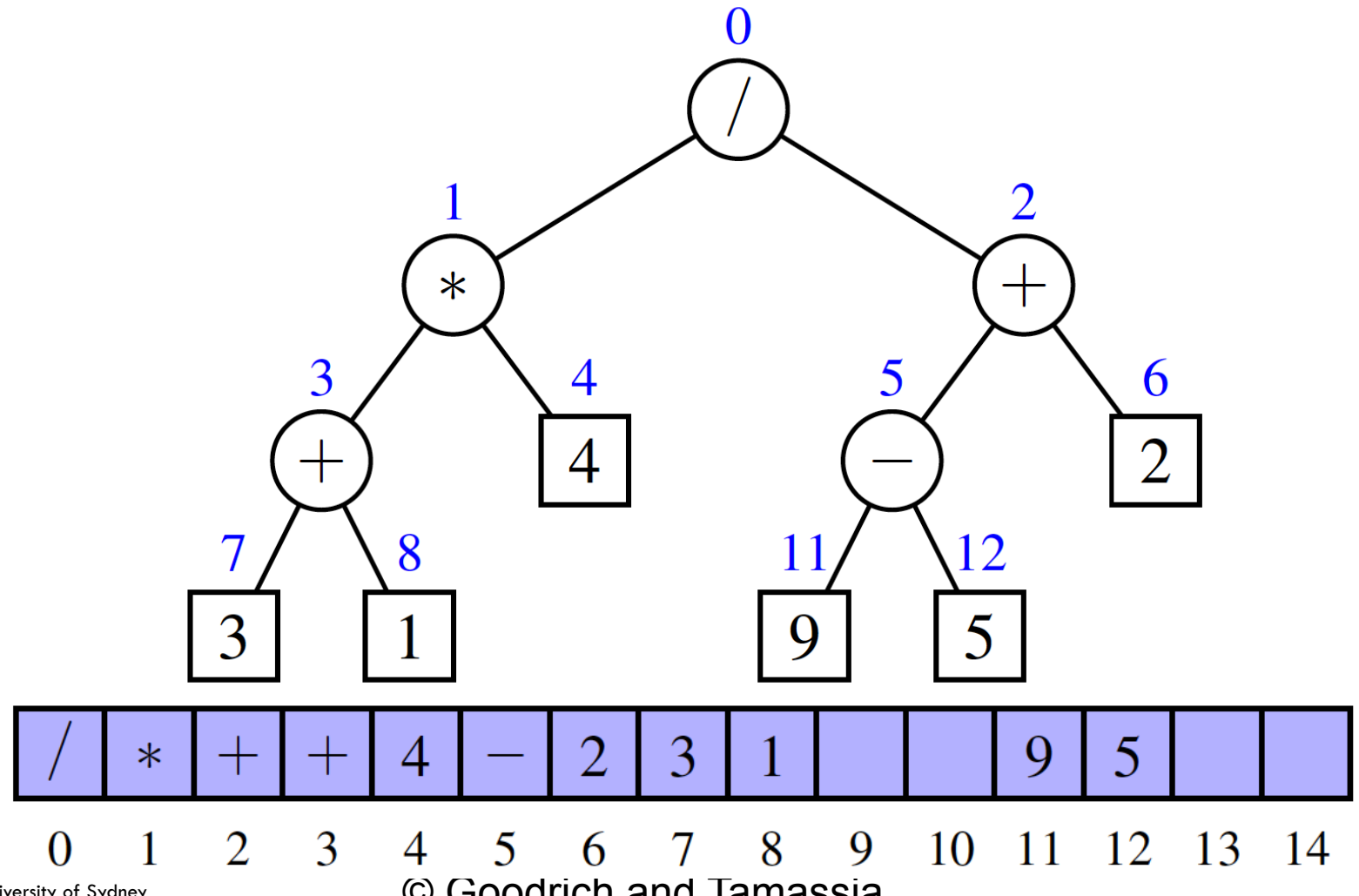
Warning: there can be gaps in the array, where no node is stored.

Level numbering for binary trees





Array representation of a binary tree



Space requirements for array representation

- Good for balanced trees
- May require too much space for a general binary tree, due to many gaps
  - What is the worst case scenario?
    - what type of Binary tree will lead to great waste of memory?
    - For a tree of n nodes, how large an array might be needed?
      - Answer: array size is 2^h+1
      - h is between $\log_2(n+1)$ -1 and n-1
      - So worst case array size is $O(2^n)$

Outline

- Trees
  - Terminology
  - Tree ADT
  - Tree traversal algorithms
- Binary Trees
  - Terminology, properties
  - BinaryTree ADT
  - More tree traversals
- Implementing trees
  - Linked-lists
  - Array-based
- Recursive code on trees

Examples of recursive code on trees

- Calculating depth
- Calculating height
- Java implementations of a recursive height( ) method

Depth of a node

Algorithm depth(T, v):
  if v is the root of T **then**
    return 0
  else
    return $1 + \text{depth}(T, w)$, where w is the parent of v in T

Worst case running time: O(h) where h is height of T
Scaling with n, worst case is O(n) for a tree that is arranged as a long chain

Method depth in the java AbstractTree class:

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

Height of a tree (naïve method)

```
Algorithm heightBad( T):
h ← 0
for each node x in T do
      if x is external then
            h ← max(h, depth (T, x))
return h
```

direct expression of the definition:
height is maximum depth
of any node in the tree

Works correctly but is very inefficient:
loop over n nodes x
for each x, call depth, which costs O(n)
so total cost is $O(n^2)$

A better height method

Recursive method
Let's first give a formula for the height of a node:

if n is a leaf, height(n) = 0
otherwise height(n) = 1+
max of heights of n's children

```
Algorithm height( T, n):
if n is external then
    return 0
h ← 0
for each child w of n in T do
    h ← max(h, height2(T, w))
return 1+h
```

Method height in the java AbstractTree class:

```
/* Returns the height of the subtree rooted at Position p */
public int height(<Position<E> p){
    int h = 0;
    for (Position<E> c : children(p))
        h = Math.max(h, 1+height2(c));
    }
    return h;
}
```

Running time for improved height

- The recursive method will be called once for every node
- For every node, it will take ($A*c_p+B$) where c_p is the number of children of node p, A is time to get each child, B is time for other work associated with the node
- Total running time:
  - $\sum_p A*c_p + B$
  - $= A(\sum_p c_p) + Bn$
    - Each node, except for the root, is the child of another node, thus will appear once in $\sum_p c_p$
      Thus $\sum_p c_p = n-1$
  - so running time is $A(n-1)+Bn$, that is $O(n)$

Cost of recursive code on trees

- Sometimes, the method may call itself on all children
  - In worst case, do a call on every node
  - If the work done, *excluding the recursion*, is constant per call, then the total cost is linear in *the number of nodes*

- Sometimes, the method calls itself on at most one child
  - In worst case, do one call at each level of the tree
  - If the work done, *excluding the recursion*, is constant per call, then the total cost is linear in *the height of the tree*
- If the work done, excluding the recursion, is more than constant per call, see later in the term!

Summary

- Read chapter 8 of the textbook
- Trees
  - Terminology
  - Tree ADT
  - Tree traversal algorithms
- Binary Trees
  - Terminology, properties
  - BinaryTree ADT
  - More tree traversals
- Implementing trees
- Recursive code on trees