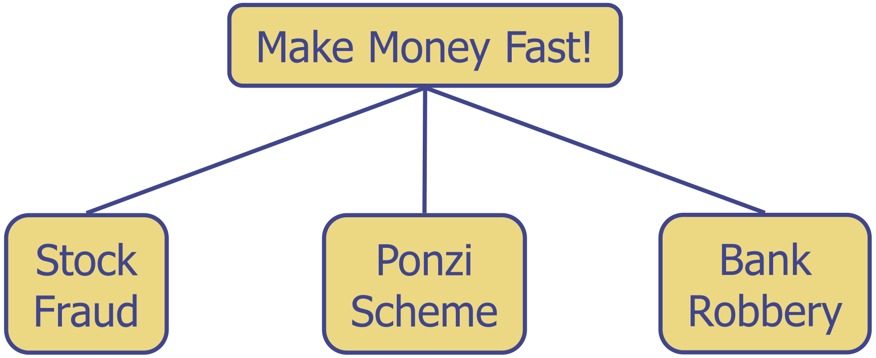
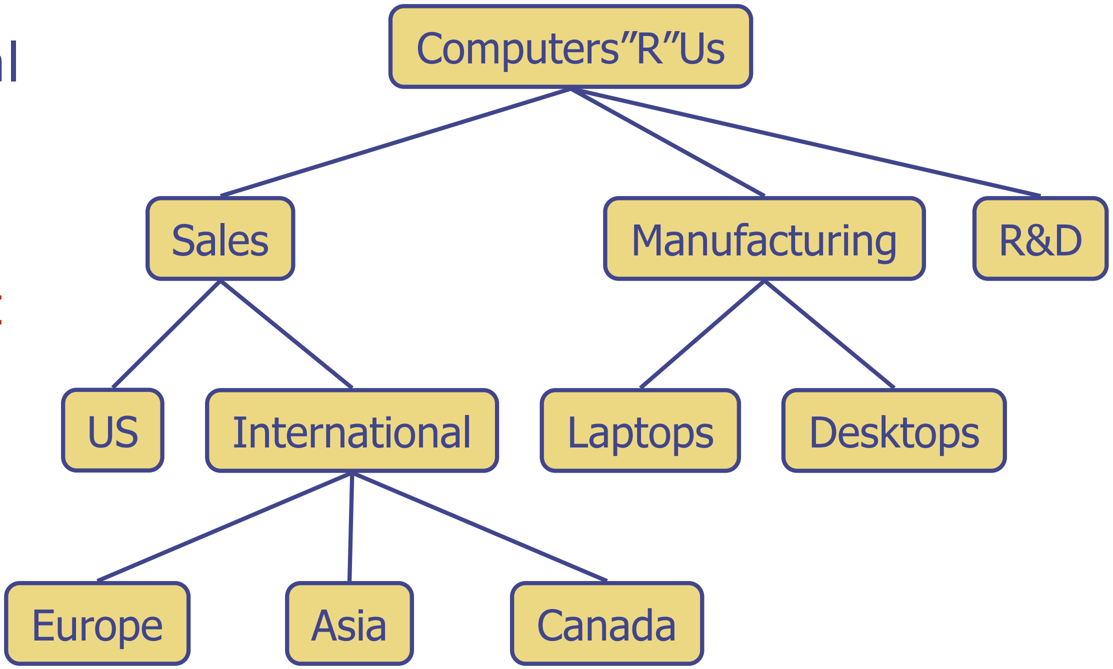
G52ACE 2017-18
Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

Trees



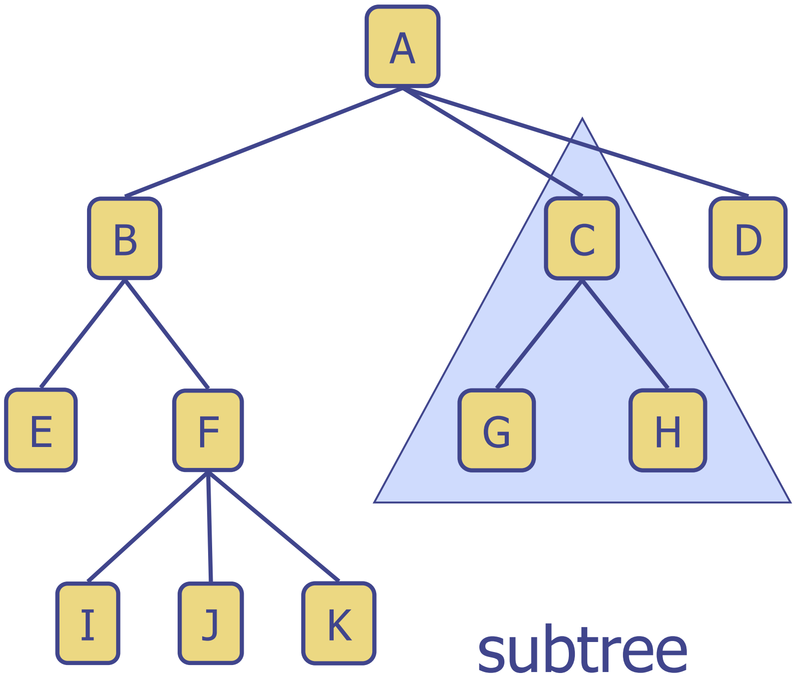
What is a (Rooted) Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation (at most one parent!)
- 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 (not counting itself)
- **Height** of a tree: maximum depth of any node = length of longest path from root to a leaf (3 for tree on right)
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Subtree**: tree consisting of a node and its descendants



Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterator positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update method:
  - object replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

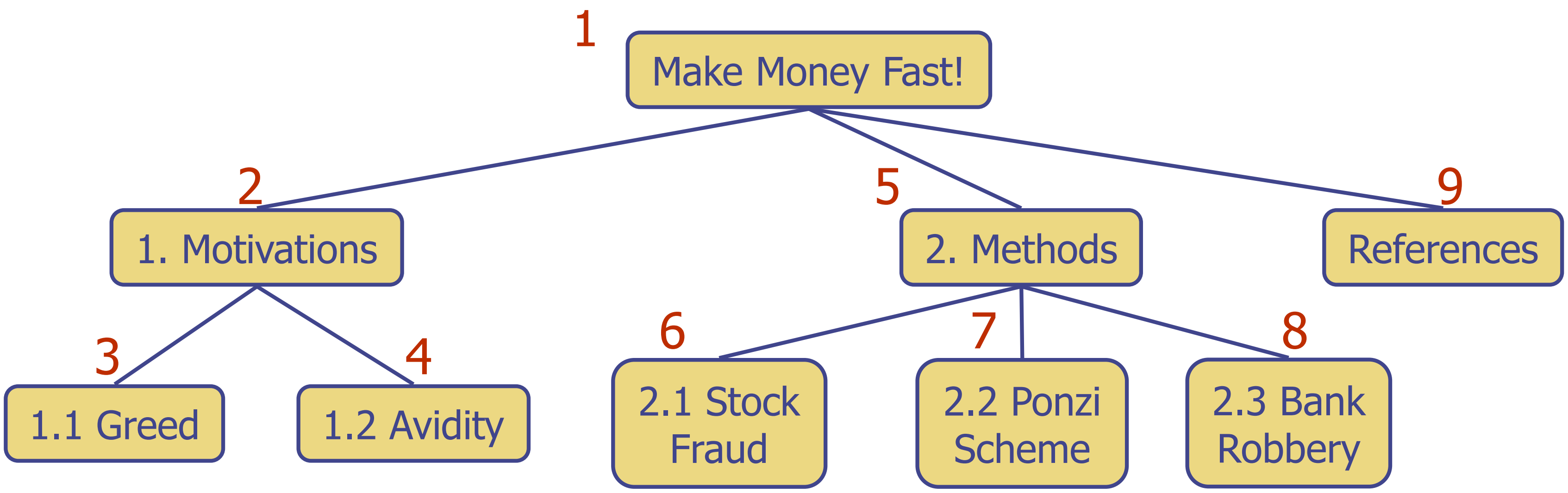
Traversals

- Given a data structure, a common task is to traverse all elements
  - visit each element precisely once
  - visit in some systematic and meaningful order

Preorder Traversal

- In a preorder traversal, a node is visited **before** its descendants
- Application: print a structured document

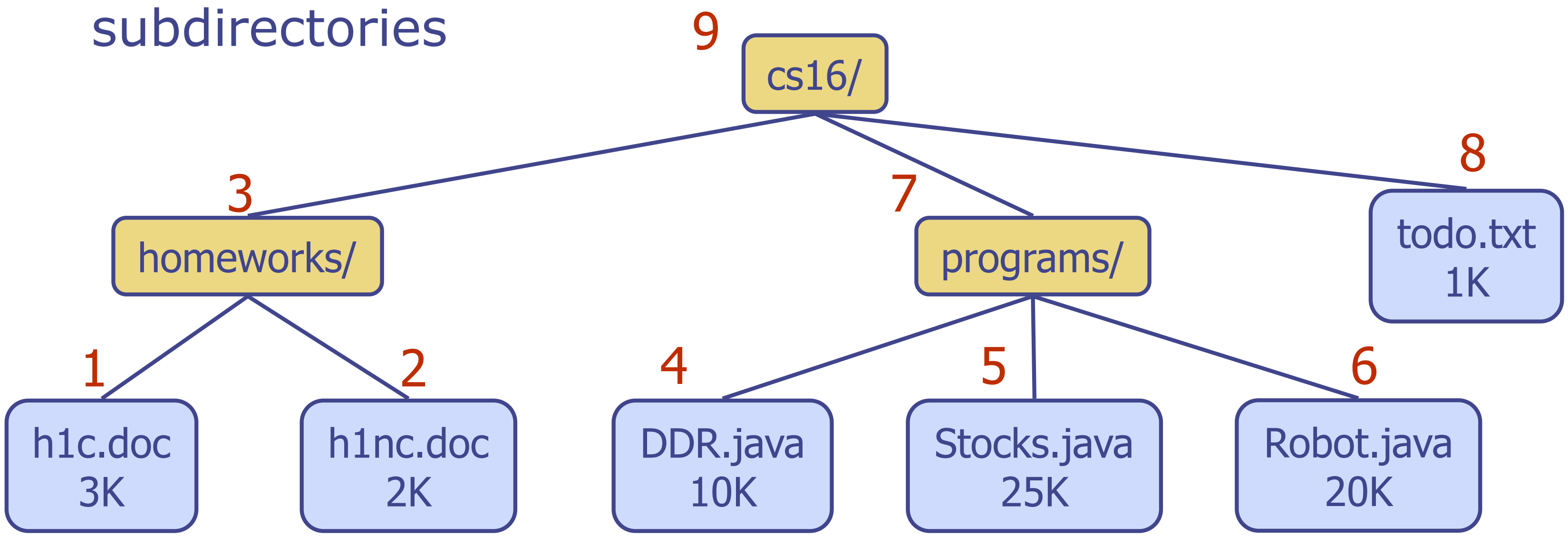
```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preorder (w)
```



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

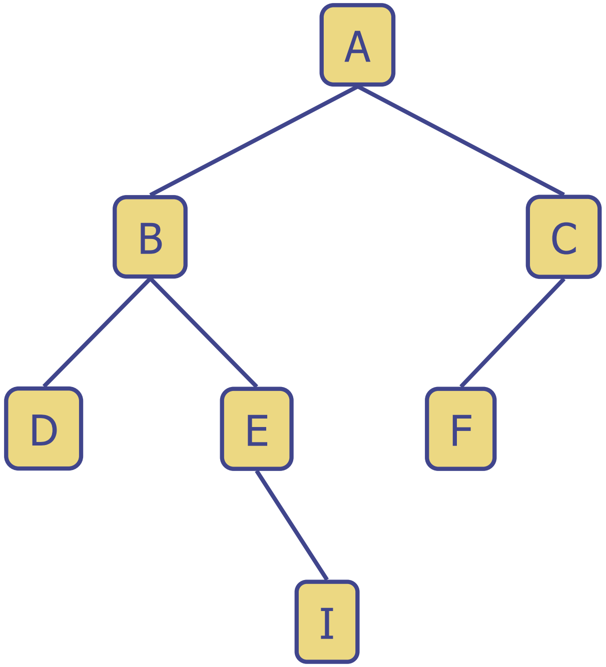
Algorithm *postOrder*(v)
    for each child w of v
        postOrder (w)
    visit(v)



Binary Trees

- Applications:
  - searching

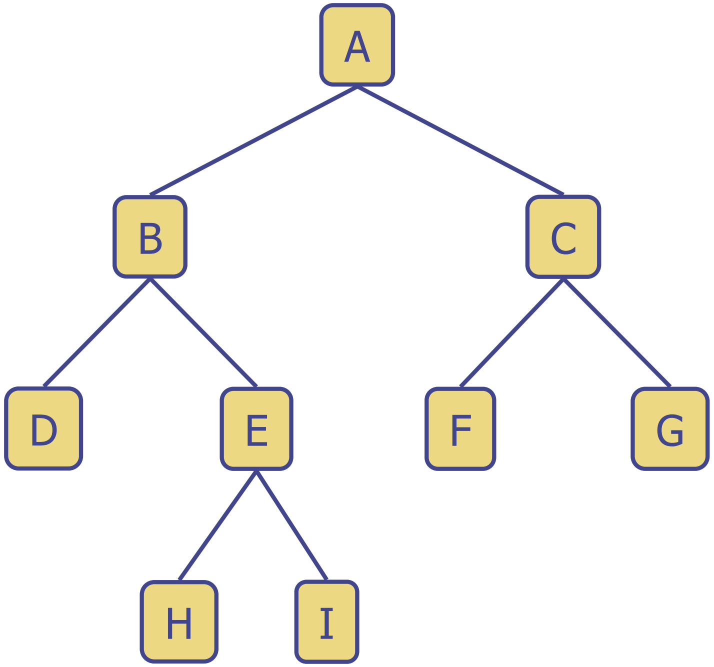
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children
  - The children of a node are an ordered pair - though one might be "missing"
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of "children", each of which is missing (a null) or is the root of a binary tree



Proper Binary Trees

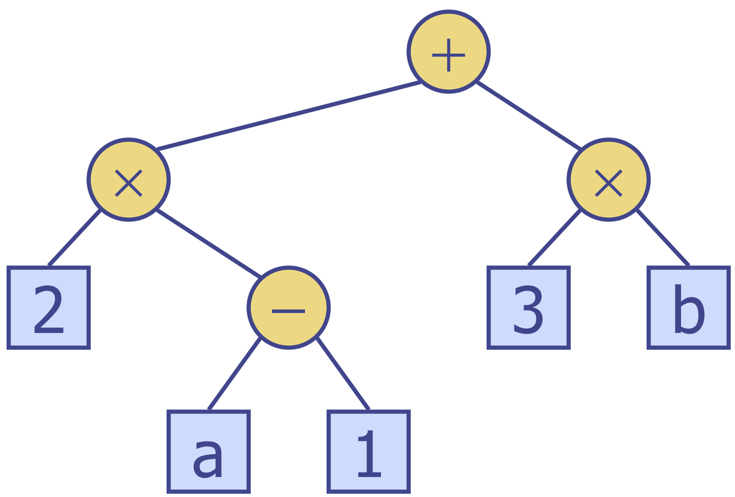
- A proper binary tree is a tree with the following properties:
  - Each internal node has either two children or no children
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a root of a binary tree

- Applications:
  - arithmetic expressions
  - decision processes



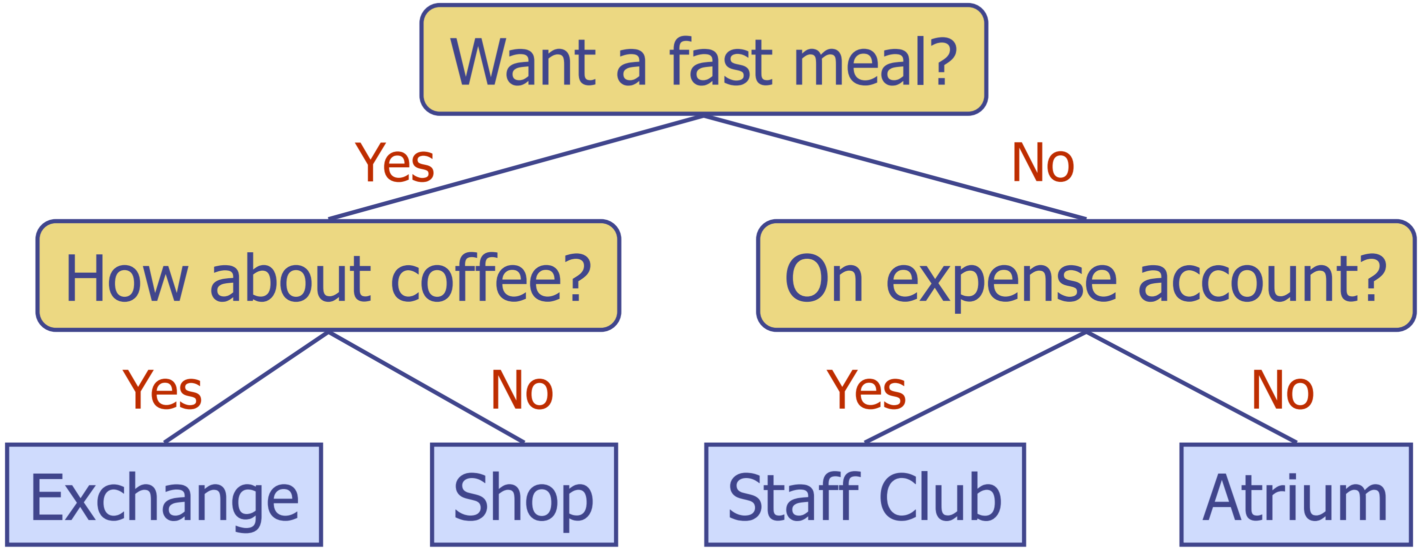
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: (binary) 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
    - hence a proper tree
  - external nodes: decisions
- Example: dining decision



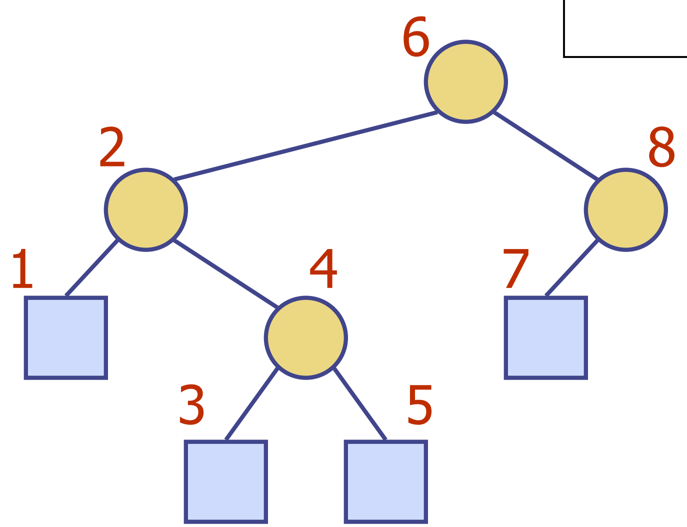
BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position left(p)
  - position right(p)
  - boolean hasLeft(p)
  - boolean hasRight(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

Inorder Traversal

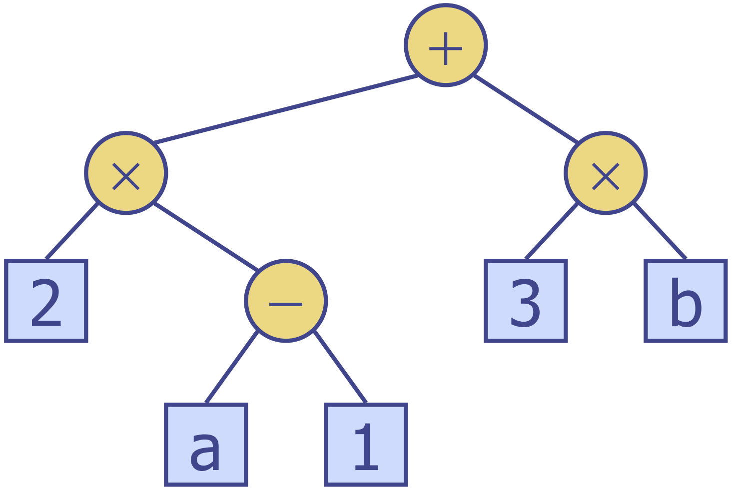
- **In an inorder traversal a node is visited after its left subtree and before its right subtree**
- Application: draw a binary tree by (x,y) coords:
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder*(*v*)
    if *hasLeft* (*v*)
        inOrder (*left* (*v*))
    visit(*v*)
    if *hasRight* (*v*)
        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



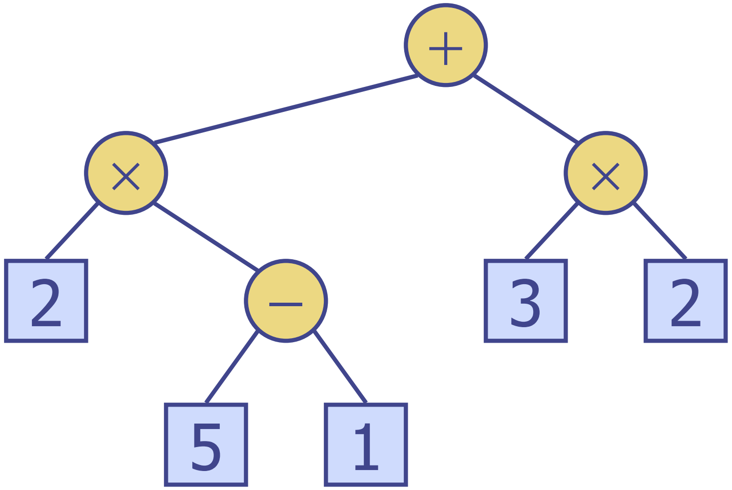
Algorithm *printExpression*(*v*)
  if *hasLeft* (*v*)
    print("(")
    printExpression (*left*(*v*))
  print(*v.element* ())
  if *hasRight* (*v*)
    printExpression(*right*(*v*))
    print (")")

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

Evaluate Arithmetic Expressions

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

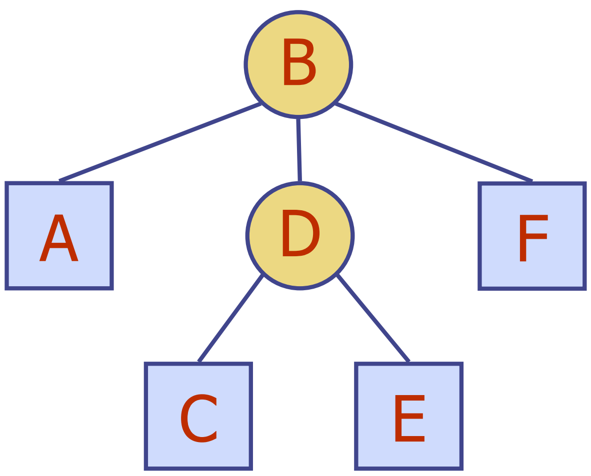
Algorithm *evalExpr*(v)
    if *isExternal* (v)
        return *v.element* ()
    else
        $x \leftarrow$ ***evalExpr***(***leftChild*** (v))
        $y \leftarrow$ ***evalExpr***(***rightChild*** (v))
        $\Diamond \leftarrow$ operator stored at v
        return $x \Diamond y$

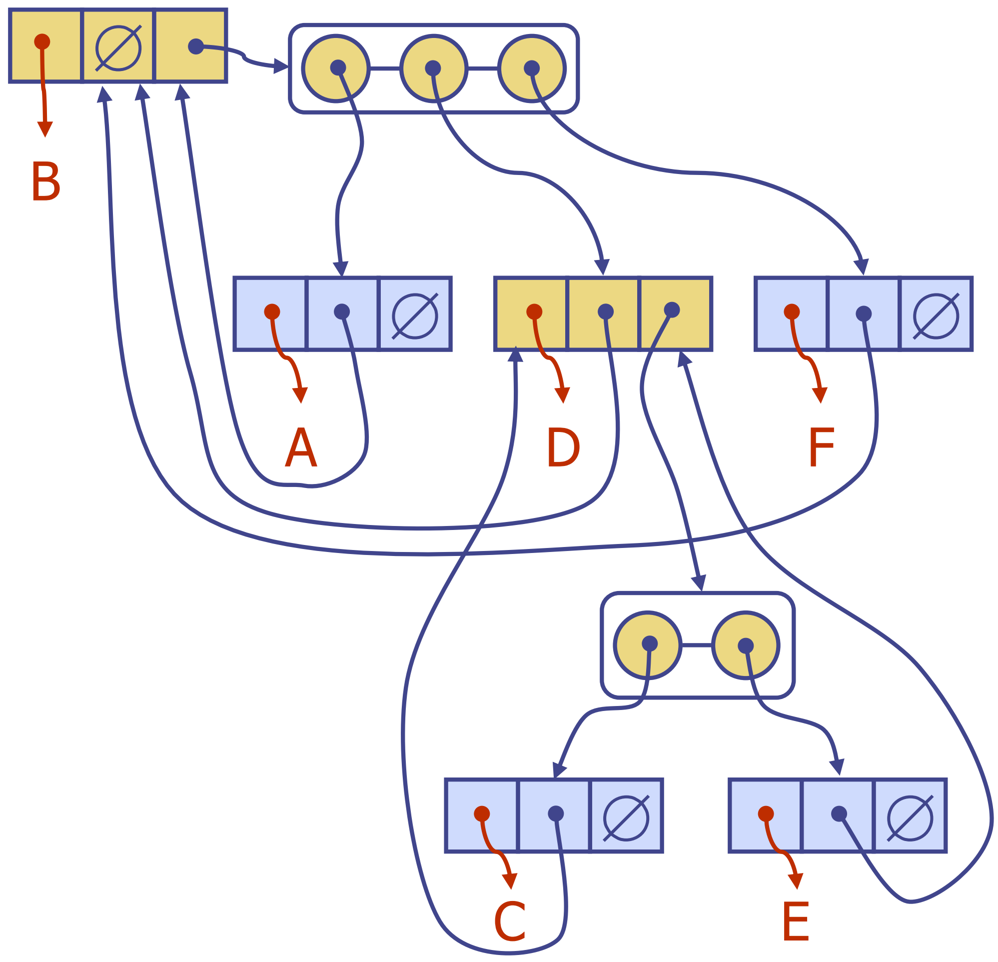


- Exercise: what is the value?
- Exercise: Which traversal ?
- Post- In- or Pre- ?

Linked Structure for Trees

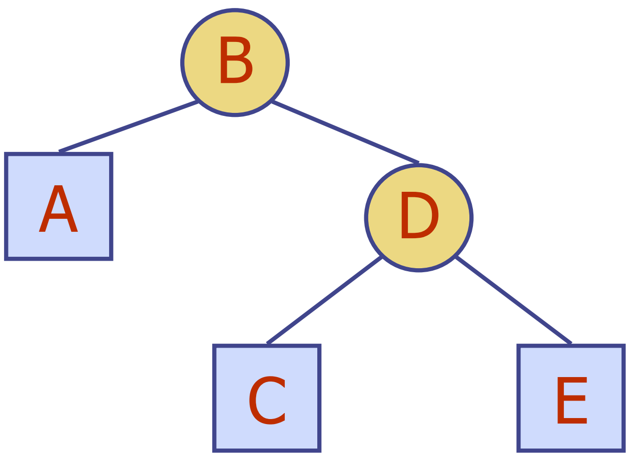
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT

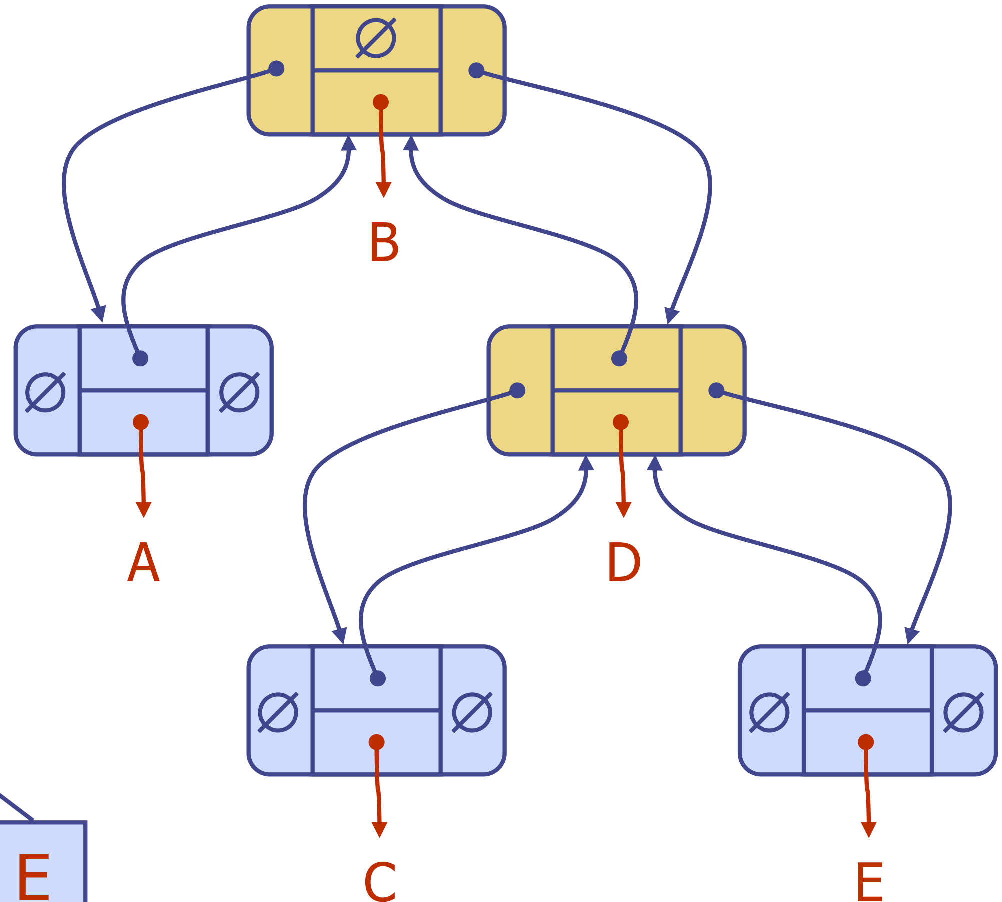




Linked Structure for Binary Trees

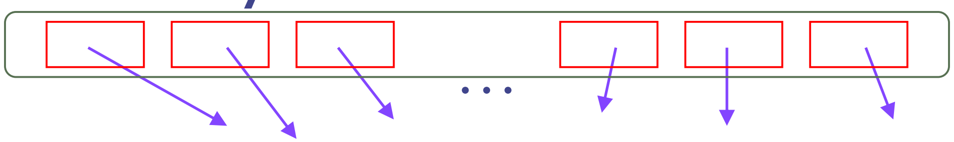
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT

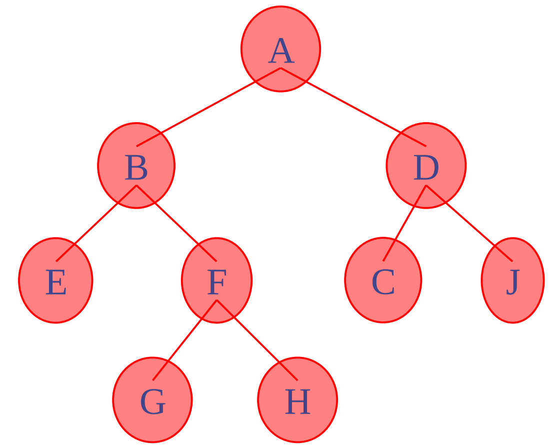




Array-Based Representation of Binary Trees

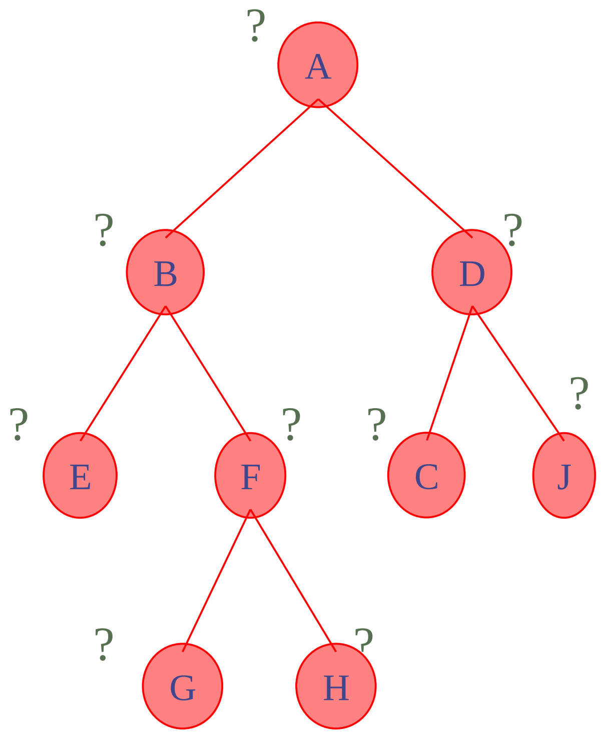
- nodes are stored in an array





- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node),
    rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node),
    rank(node) = 2*rank(parent(node))+1

Array-Based Representation of Binary Trees



- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node),
    rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node),
    rank(node) = 2*rank(parent(node))+1

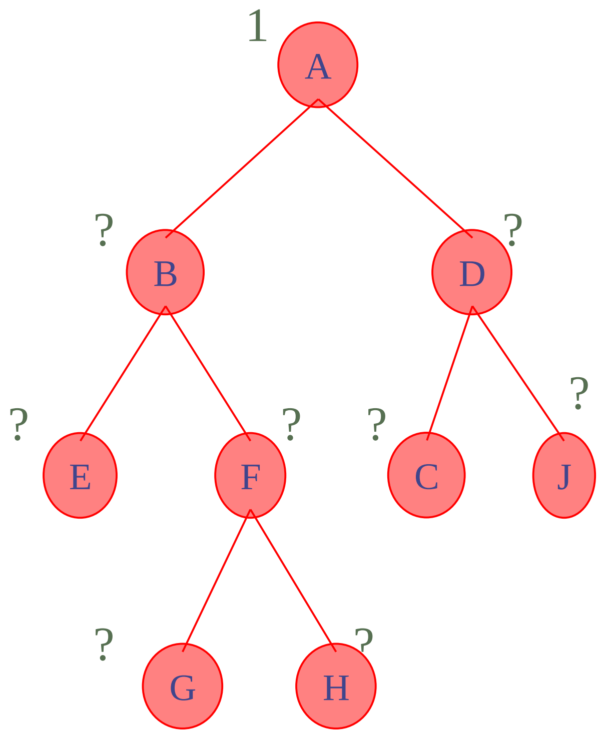
Exercise (online): fill in the array with nodes,

i.e. find the index for each node of the tree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Array-Based Representation of Binary Trees

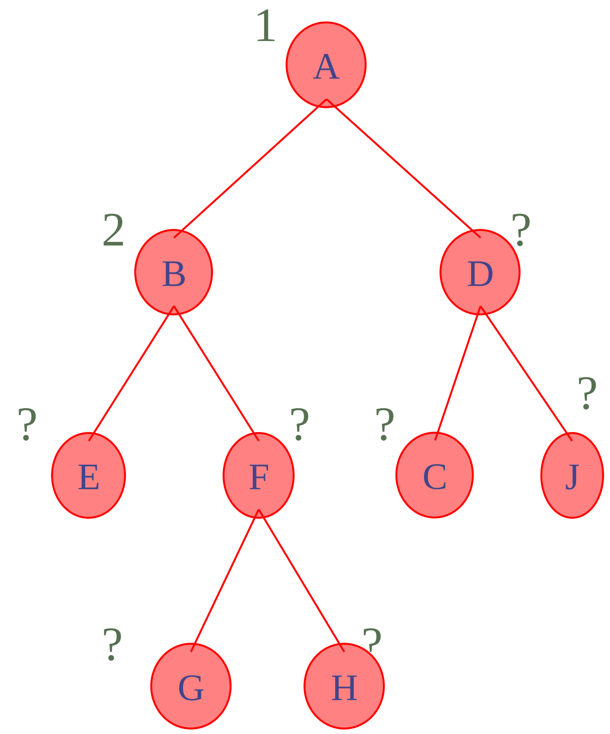
- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2*rank(parent(node))+1



Exercise (online): fill in the array with nodes, i.e. find the index for each node of the tree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A															

Array-Based Representation of Binary Trees

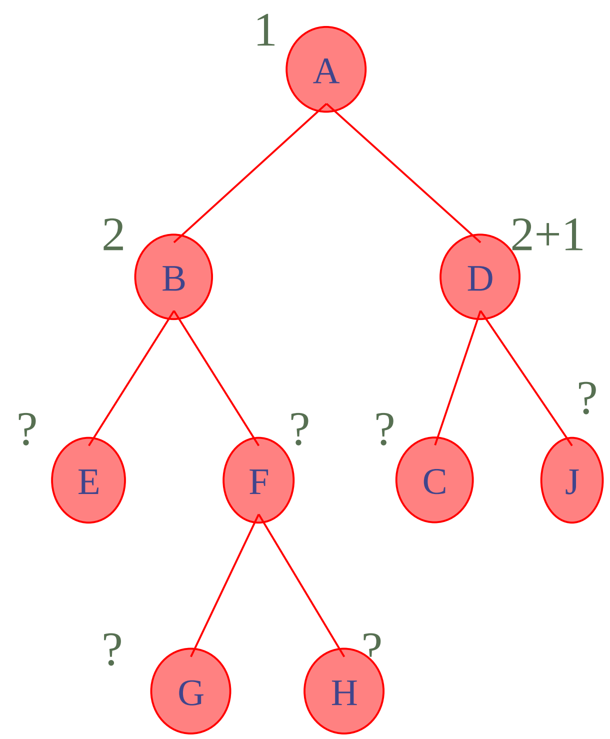


- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node),
    rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node),
    rank(node) = 2*rank(parent(node))+1

Exercise (online): fill in the array with nodes, i.e. find the index for each node of the tree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A	B														

Array-Based Representation of Binary Trees

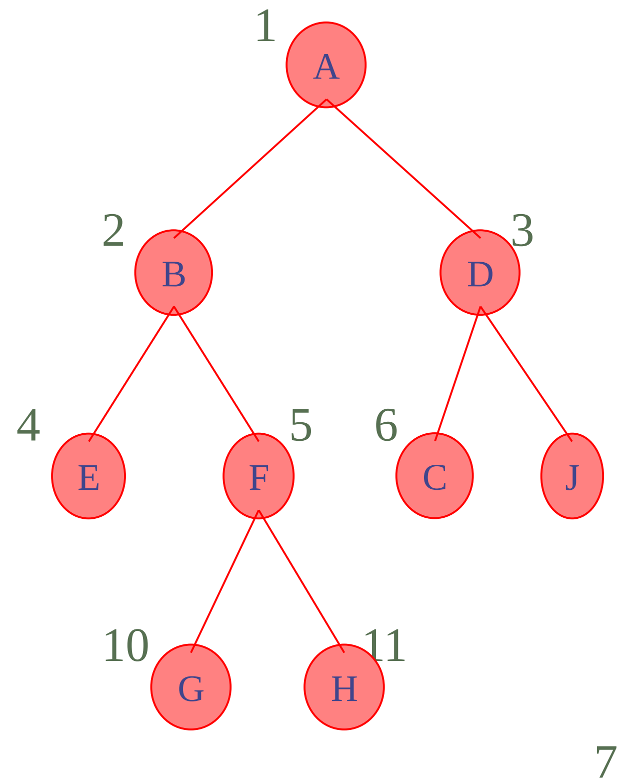


- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2*rank(parent(node))+1

Exercise (online): fill in the array with nodes, i.e. find the index for each node of the tree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A	B	D													

Array-Based Representation of Binary Trees



- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2*rank(parent(node))+1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A	B	D	E	F	C	J			G	H					

Implemention

- Remember that if think of the rank, r(n) of node n, as a binary number then
  - "times 2" is a left shift "<<1"
- r(n) = r(par(n))<<1 + 0 for left
- r(n) = r(par(n))<<1 + 1 for right
- E.g. r(par(n)) = 101 gives children at
  - "101"+"0" = 1010
  - "101"+"1" = 1011
- Going to the parent is a right shift
- Hence, implementations of this can be very fast – used in binary heaps next semester.

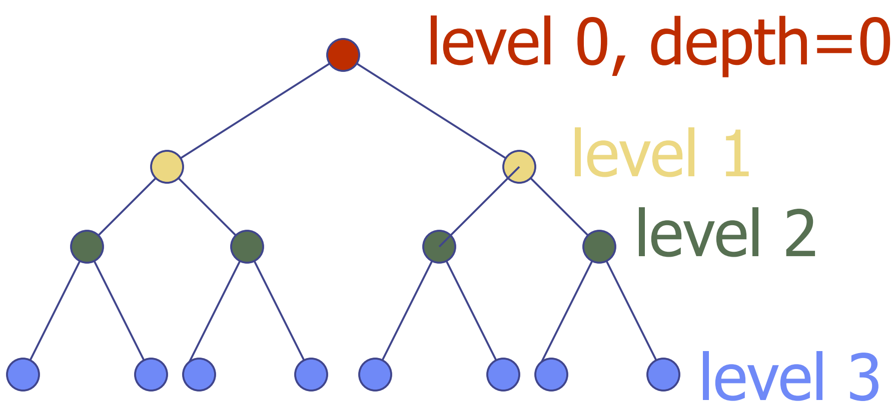
Exercise (offline)

- Why is this representation correct?
  - I.e. does the mapping between array satisfy needed uniqueness properties?
  - Is it true that each element of the array corresponds to a unique node of the tree?
  - E.g. If I claim that it is incorrect, then how would you convince me otherwise?
- Hint: from the previous slide think of the rank written as binary number
  - Realise that it describes the “L” vs. “R” decisions on going from the root.
  - Relate to each number having a unique binary representation

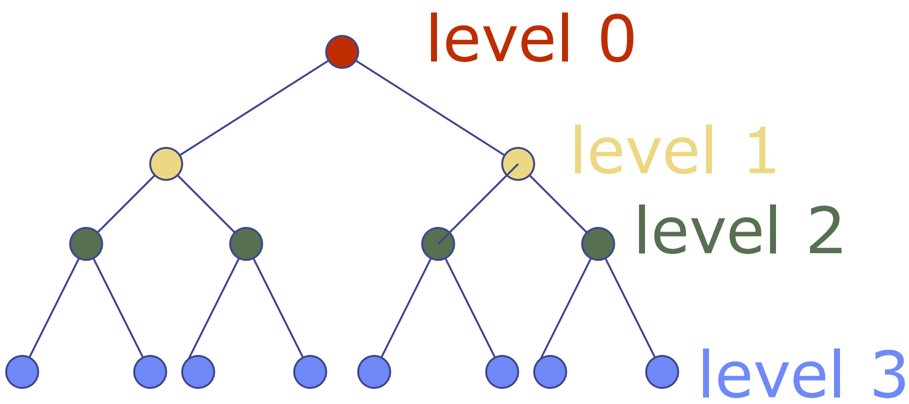
Properties of perfect binary trees

- A binary tree is said to be "**proper**" (a.k.a. "**full**") if every internal node has exactly 2 children
- It is "**perfect**" if it is proper and all leaves are at the same depth; hence all levels (depths) are full.

Perfect binary tree of height 3:



Properties of perfect binary trees



d	at d	at d or less
0	1	1
1	2	3
2	4	7
3	8	15

Counting suggests numbers of nodes are:

- 2^d at level d
- 2^{d+1}-1 at level d or less

Can formally prove using induction

Height (h) is logarithmic in size (n)

- This is a very important property of perfect binary trees
- Exercise: is this true for all trees?
- Tree algorithms often work by going down the tree level by level – following a path from root to a leaf
- So, their running time depends on the number of levels – but we usually only know the number of nodes
- Let us prove that for perfect binary trees

  $$\boldsymbol{h} = \log_2(\boldsymbol{n}+1) - 1$$

  where n in the number of nodes. Or (same thing):

  number of levels = $\log_2(\boldsymbol{n}+1)$.

How many nodes at level k

- First, it is useful to find out how many nodes are at a certain level in perfect binary tree
- Let us count levels from 0. This way level k contains nodes which have depth k.

How many nodes at level k

- Claim: level k contains 2^k nodes.
- Proof: by induction on k.
  - (basis of induction) if $k = 0$, the claim is true:

  $2^0 = 1$, and we only have one node (root) at level 0.
  - (inductive step): suppose the claim is true for k-1: level k-1 contains 2^{k-1} nodes.
  - We need to prove that then the claim holds for k: level k holds 2^k nodes.
  - Since each node at level k-1 has 2 children, there are twice as many nodes at level k.
  - So, level k contains $2 * 2^{k-1} = 2^k$ nodes. QED

How many nodes in a tree of height h?

Theorem: A perfect binary tree of height h contains

$2^{h+1} - 1$ nodes.

Proof: by induction on h

- (basis of induction): $h=0$. The tree contains $2^1 - 1 = 1$ node.
- (inductive step): assume a tree of height $h-1$ contains $2^h - 1$ nodes. A tree of depth h has one more level (h) which contains 2^h nodes. The total number of nodes in the tree of height h is: $2^h - 1 + 2^h = 2 * 2^h - 1 = 2^{h+1} - 1$. QED

What is the height of a (perfect) binary tree of size n (with n nodes)?

We know that $n = 2^{h+1} - 1$.

So, $2^{h+1} = n + 1$.

$h + 1 = \log_2(n+1)$

$h = \log_2(n+1) - 1$.

So, the height of the tree is logarithmic in the size of the tree.

The size of the tree is exponential in the height (number of levels) of the tree.

What is the height of an arbitrary binary tree of size n (with n nodes)?

- If the tree is perfect, then it has height that is logarithmic in the size of the tree: $\Theta(log(n))$
- If it is imperfect then for the same n it must have at least this height: $\Omega(\log(n))$
- However, consider a simple "chain"
  - each node has just one child (or none)
  - It is a special case of a binary tree.
  - It has height n and is (obviously) the maximal height
  - Hence, trees have height O(n).
- Hence, for a general binary tree on n nodes, the height is $\Omega(\log(n))$ and $O(n)$

Minimum Expectations

- Definitions associated with trees
- **Post- Pre- and In-order traversal and their usages**
- Implementation methods
  - nodes
  - array based
- Binary Trees – meaning of proper, perfect
- **Sizes and heights of binary trees**