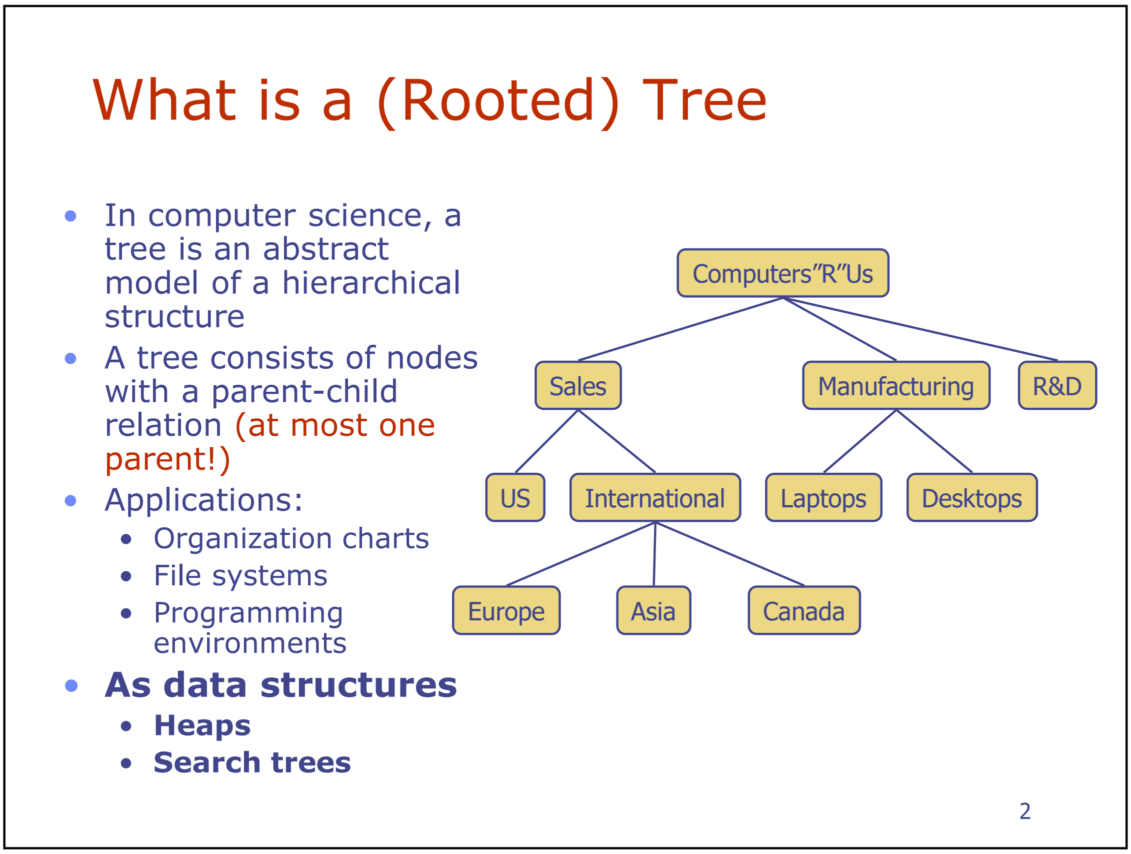
COMP2054-ADE
Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

Trees : Terminology, Traversals, Representations, and Properties

So far we have looked at arrays and lists which we might think of as “linear” or “non-branching” but of course trees are also well-known and vital.
So will look at terminology, how to traverse them, various representations and properties.
Many parts of the lecture are probably already familiar, but some parts are probably new to you.

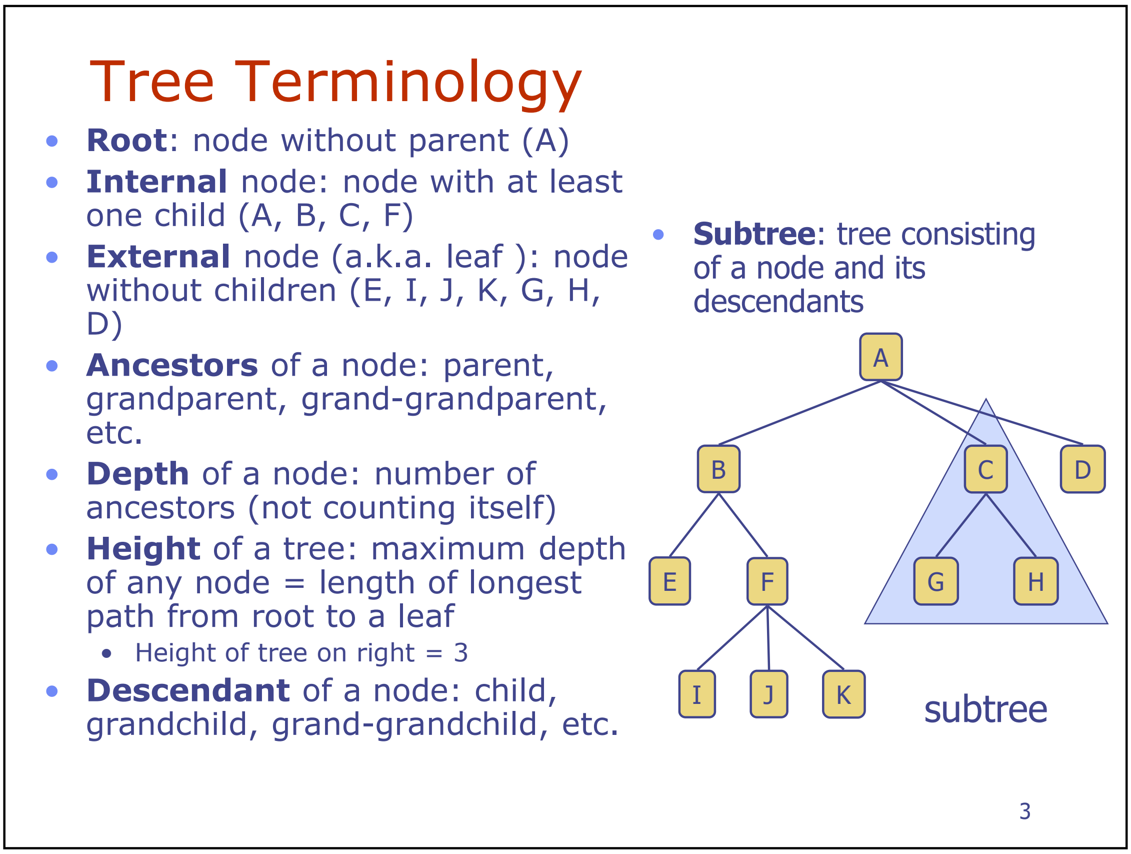


What is a (Rooted) Tree

- 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
- **As data structures**
  - **Heaps**
  - **Search trees**

Computers"R"Us
Sales
Manufacturing
R&D
US
International
Laptops
Desktops
Europe
Asia
Canada

Firstly, we will deal with “Rooted” trees – and so an ordering of parent and child.
A tree might well be non-rooted and so have no special node, but these are less used – and may well just be treated as special cases of graphs.
There are many obvious applications.
The main usage in this module is that the form the underlying structure used in the data structures: “heaps” and “search trees”



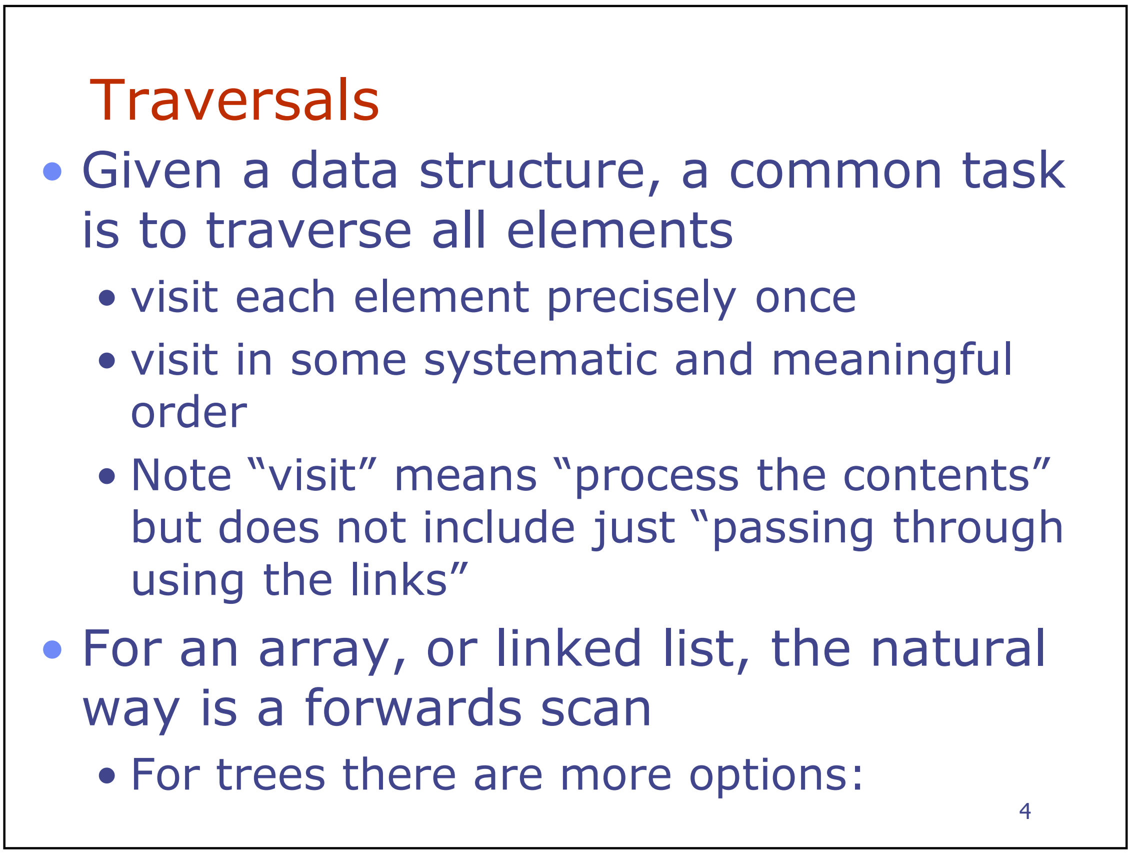
Tree Terminology

- **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
  - Height of tree on right = 3
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.

- **Subtree**: tree consisting of a node and its descendants

3

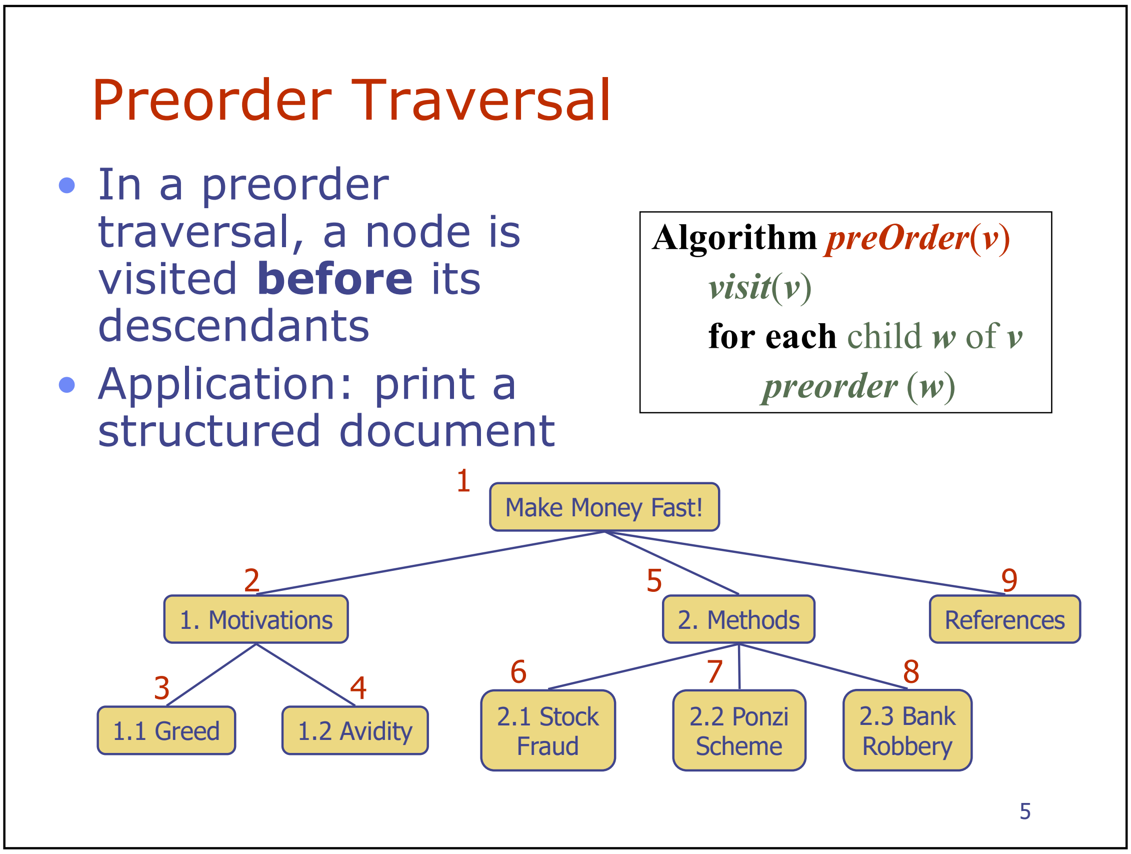
Some terminology.
Many parts are obvious, but do need to be understood.
Some key properties are that a subtree formed from a node and all its descendants is itself a tree. This allows for the obvious inductive definition of a tree being a node and child trees.
Perhaps the most confusing issue is depth and height and avoiding “off by one errors”.
The tree has height 3 as a longest path is A-B-F-K which traverses 3 edges.



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
  - Note “visit” means “process the contents” but does not include just “passing through using the links”
- For an array, or linked list, the natural way is a forwards scan
  - For trees there are more options:

Of course, having defined a data structure, a natural question is how to traverse it. For arrays it is so natural, that it is probably done without thinking; but for a tree is more complex.



Preorder Traversal

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

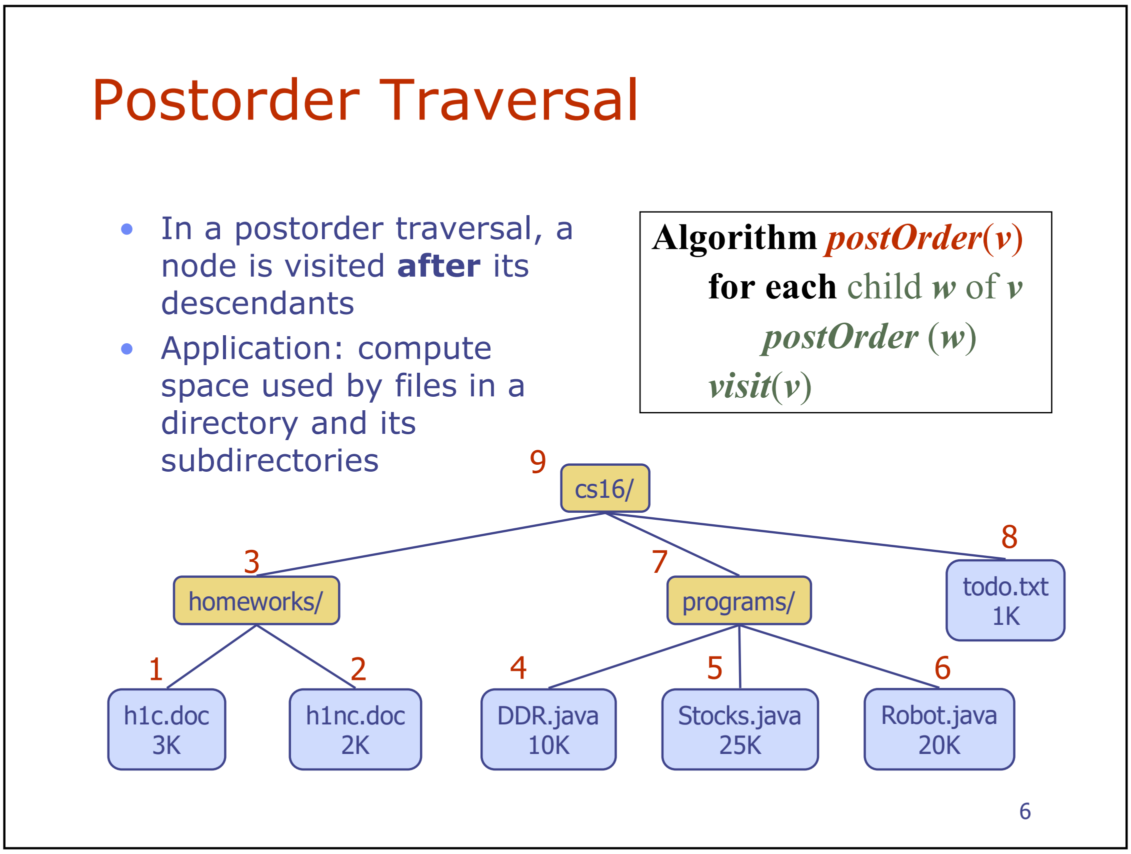
```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preorder (w)
```

5

In discussing traversals we have to distinguish between just accessing location of a node, and properly visiting it, i.e. processing its contents.
We might just use a node to access the children, without looking at what it actually stores.
The most important decision is whether to properly visit a parent node before its children, or to do children first.
If we want to process the children together then there are just two choices.
The case of doing parent first is called pre-order.
The code is simple when expressed using recursion – though in practice may well be done as iteration to same the overhead of function calls, and also to avoid that the stack is exceeded.

The pattern of access is easy to work out, and is given by the extra numbers.
As you can see it is basically depth-first, shooting down to a leaf processing nodes on the way, and then working across other children.
Note that the order of processing of the children is not specified – and may well be implementation dependent. There can be many different orders of traversal.

One usage is when the tree represents a document structure, and we want to print a table of contents.



Postorder Traversal

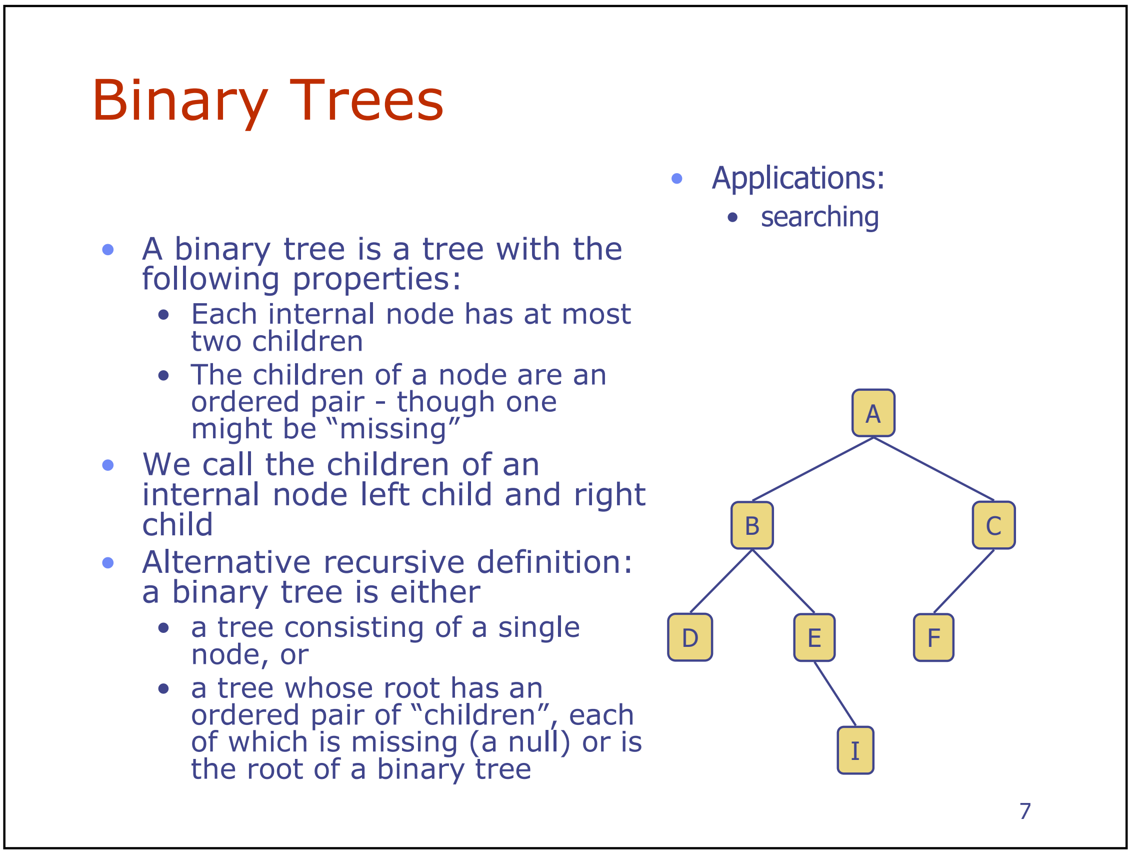
- In a postorder traversal, a node is visited **after** its descendants
- Application: compute space used by files in a directory and its subdirectories

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

9
cs16/
3
homeworks/
7
programs/
8
todo.txt
1K
1
h1c.doc
3K
2
h1nc.doc
2K
4
DDR.java
10K
5
Stocks.java
25K
6
Robot.java
20K

6

Postorder is when the parent is not processing until after all the children.
It gives the order as shown.
It is useful when the parent cannot be properly processed until all the results of children are computed.
For example if the tree is a structure of folders/directories and we want to compute total storage, then we need to process all sub-folders before we can return the result for a folder.



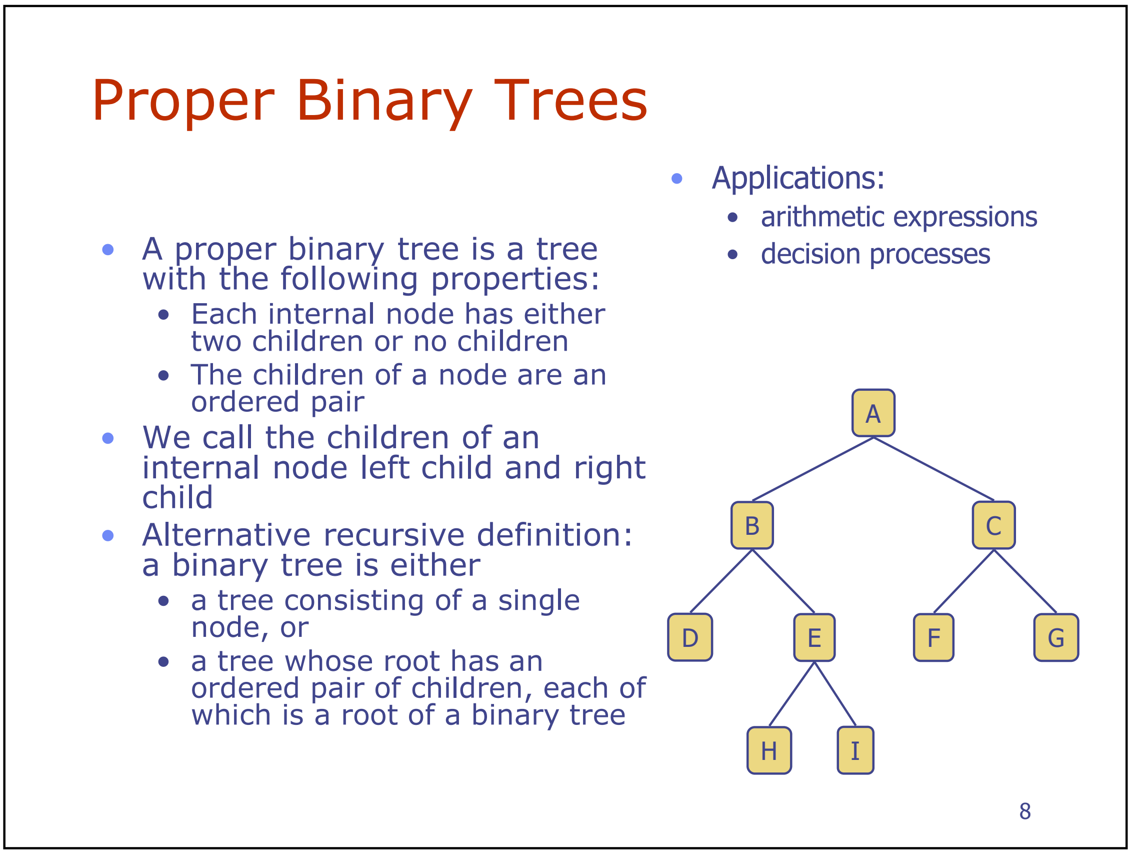
Binary Trees

- 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

- Applications:
  - searching

We now specialise to "binary trees" which just means that we have at most to children, and then it is usual to think of these are left and right.

In the case of just one child then it can be left or right and the two cases are different.



Proper Binary Trees

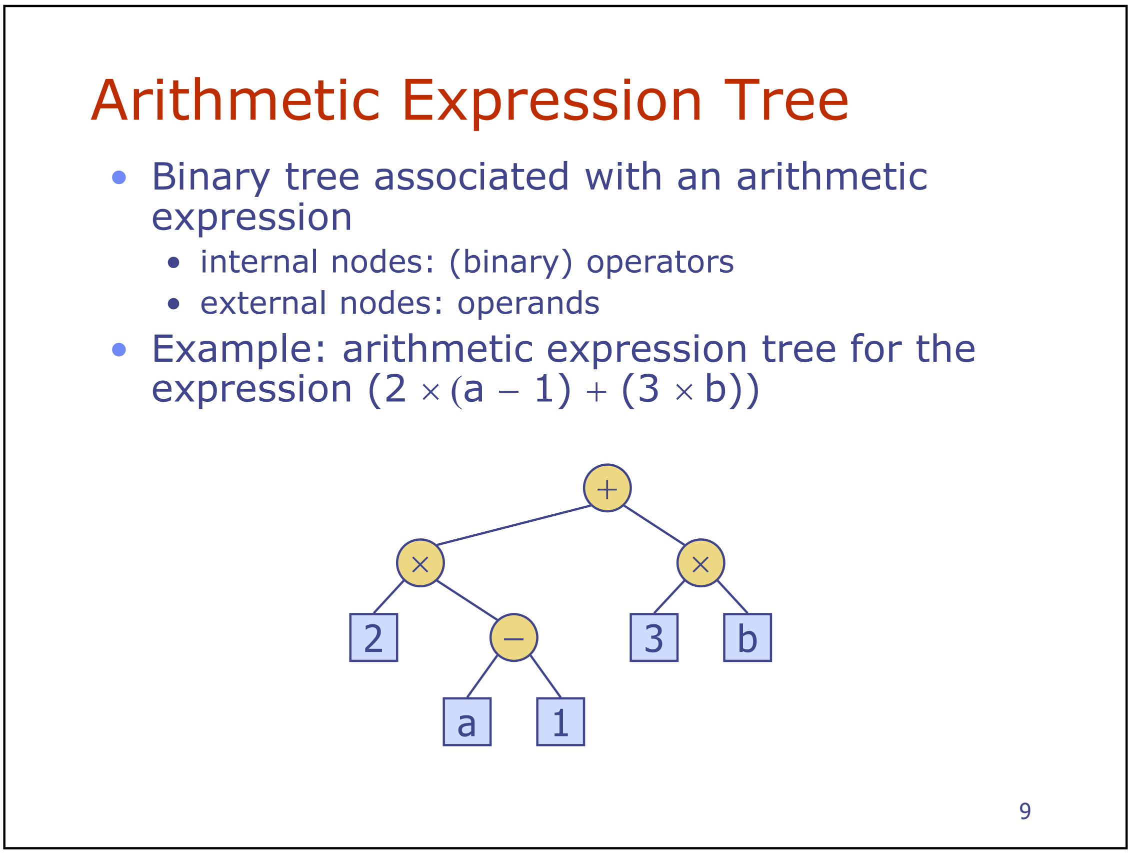
- A proper binary tree is a tree with the following properties:
  - Each internal node has either two children or no children
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either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

8

If the number of children is either zero or two, then it is said to be proper.

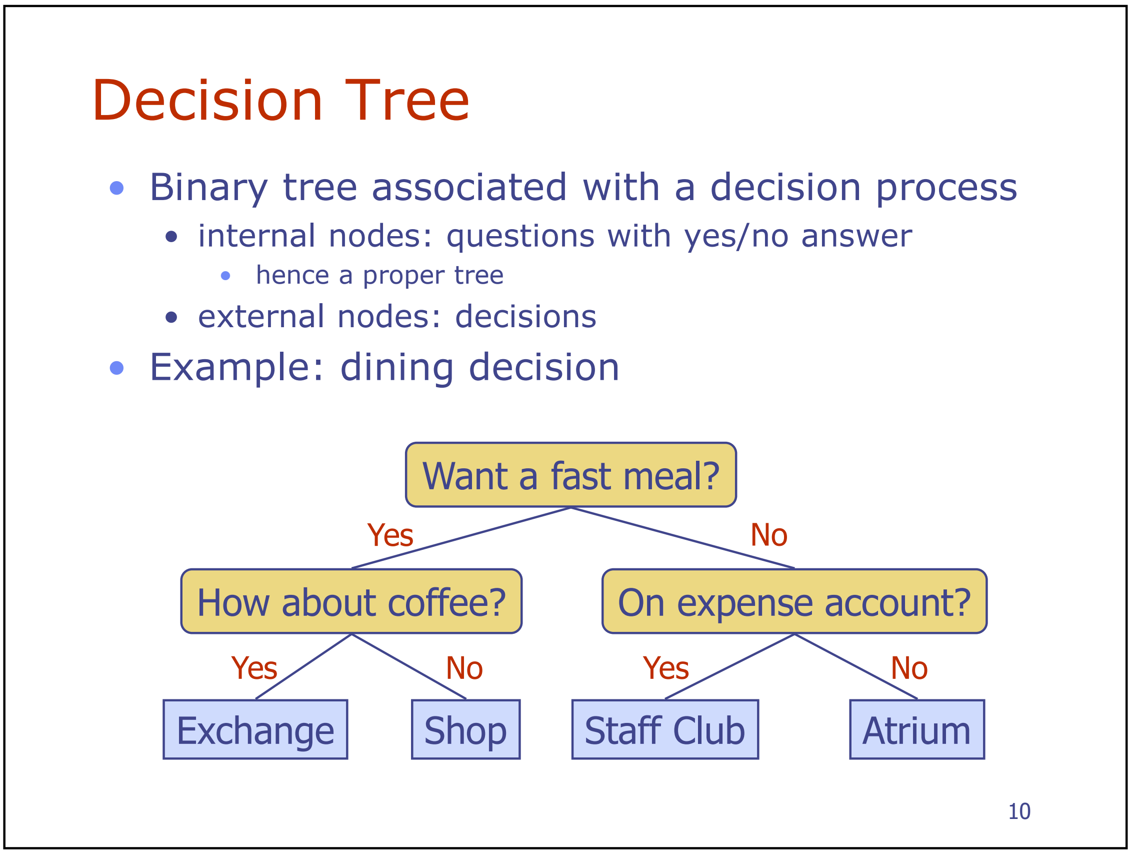
Note that the subtree from any node is also a proper tree.



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 × (a − 1) + (3 × b))

A usage of binary trees is to represent arithmetic expressions.



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

Want a fast meal?

Yes

No

How about coffee?

On expense account?

Yes

No

Yes

No

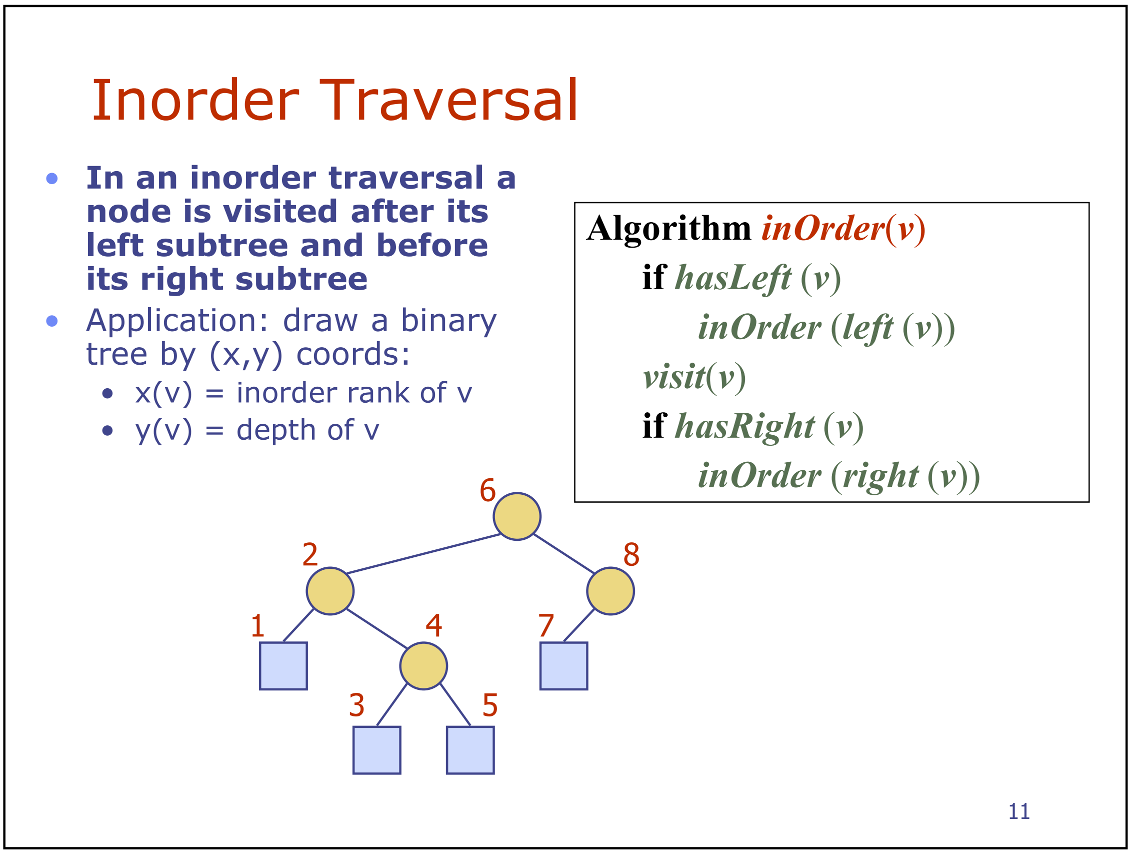
Exchange

Shop

Staff Club

Atrium

Another usage is to represent decision trees. These are naturally proper trees, as if there are any children there will be two children for yes and no.



Inorder Traversal

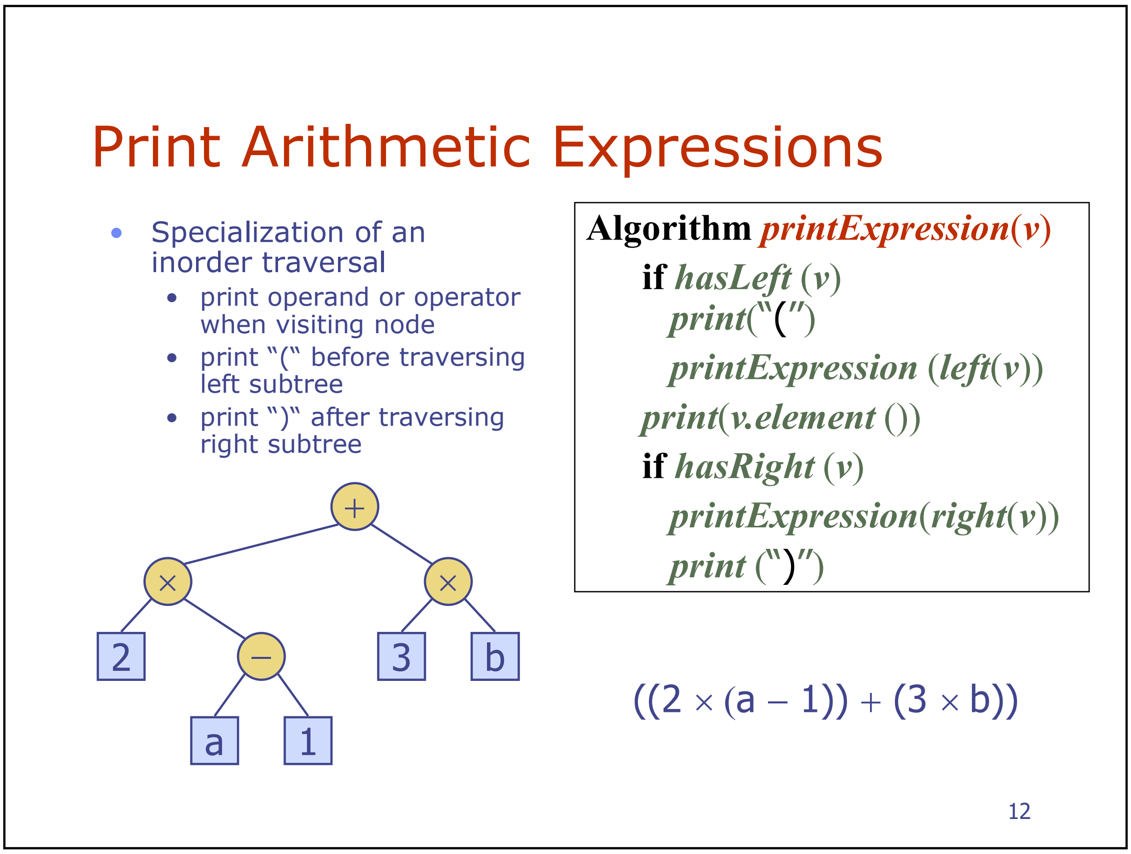
- **In an inorder traversal a node is visited after its left subtree and before its right subtree**
- Application: draw a binary tree by (x,y) coords:
  - x(v) = inorder rank of v
  - y(v) = depth of v

```
Algorithm inOrder(v)
    if hasLeft (v)
        inOrder (left (v))
    visit(v)
    if hasRight (v)
        inOrder (right (v))
```

When we have a binary tree, then we have another traversal option in that we can process the parent in between the two children.
In this case we have an option called ‘in order” in which we work left to right – left-child, parent, right-child.

The order of traversal is given, and as usual you should make sure that you can evaluate it.
It is useful in that it does work across the tree, and so for example, can be used to assign an x coordinate to each node, that allows it to be printed out in a reasonable fashion.

However, it has very important uses that we will see later.



Print Arithmetic Expressions

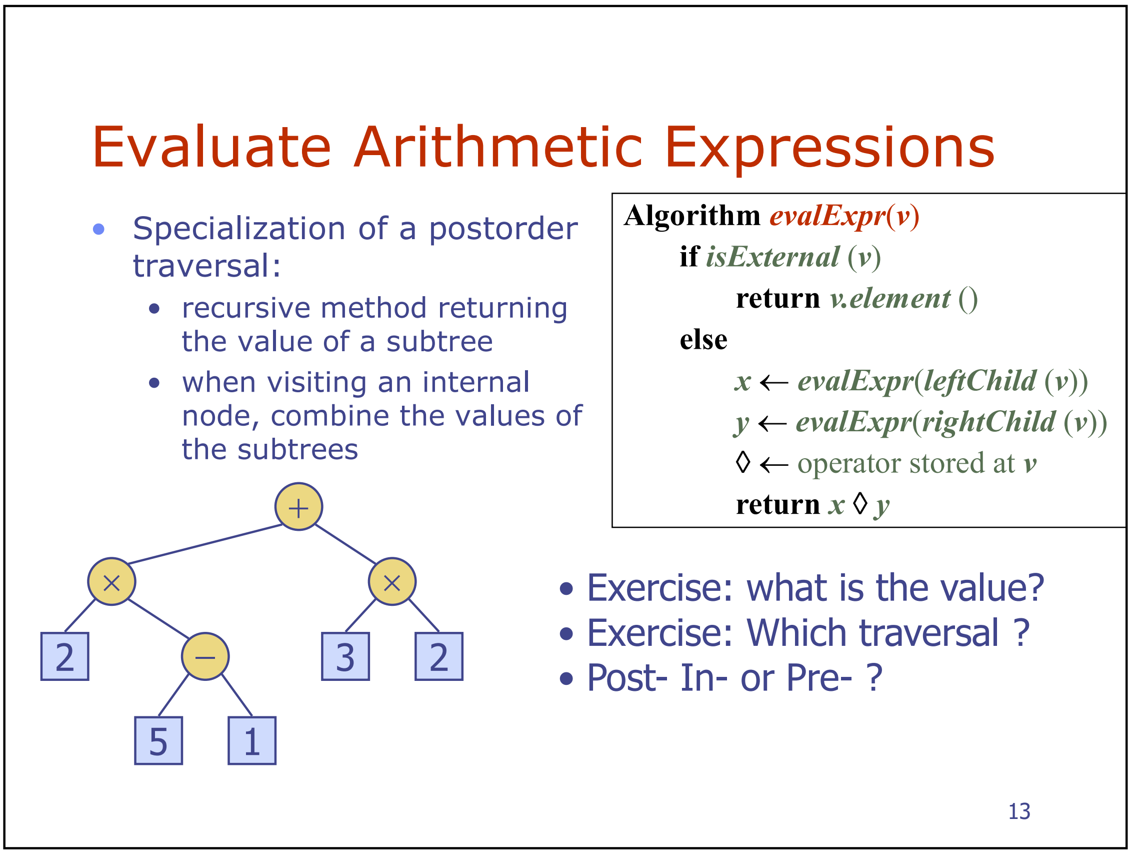
- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

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

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

Another usage of in-order traversal is to print out an arithmetic expression in a standard form.
We want to print the entire left, then the operator, then the right.

Again, work through the process to ensure you understand the traversal.



Evaluate Arithmetic Expressions

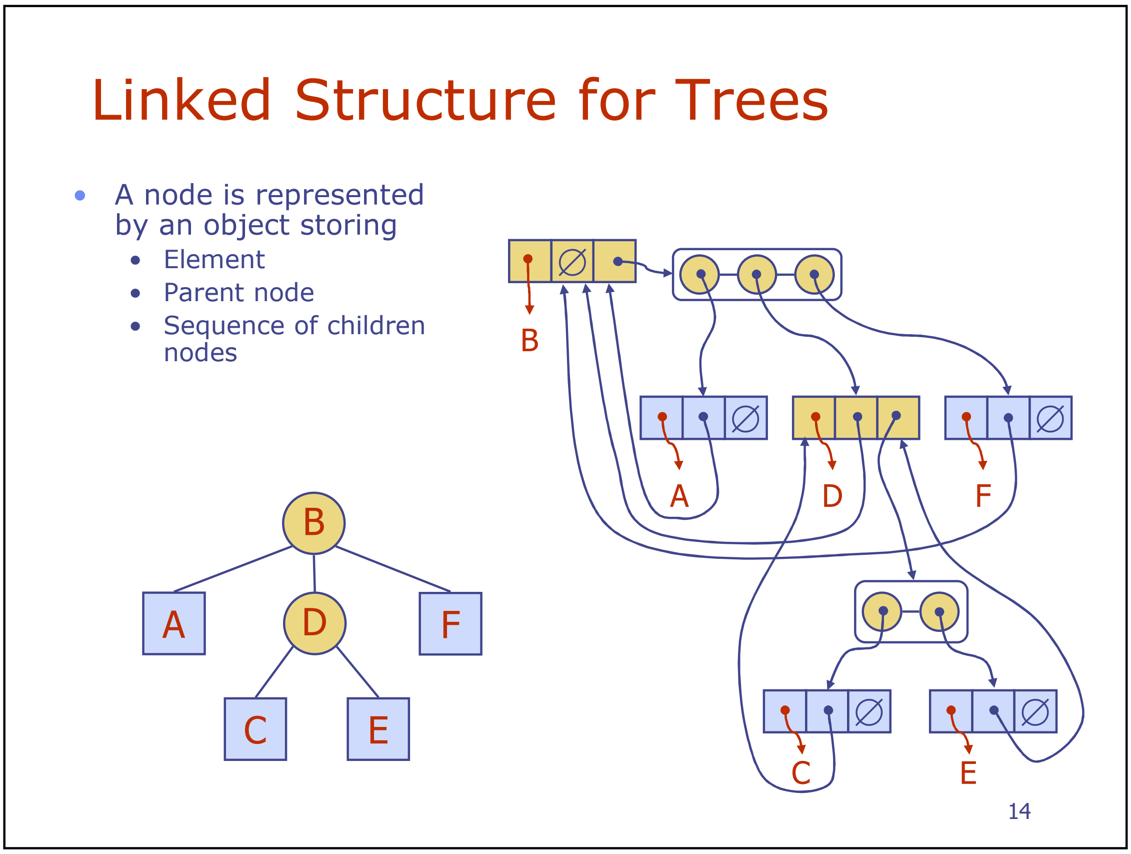
- Specialization of a postorder traversal:
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

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

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

13

But to evaluate an expression we must do evaluate both children before we can evaluate the result from the parent, and so we need a post-order traversal.

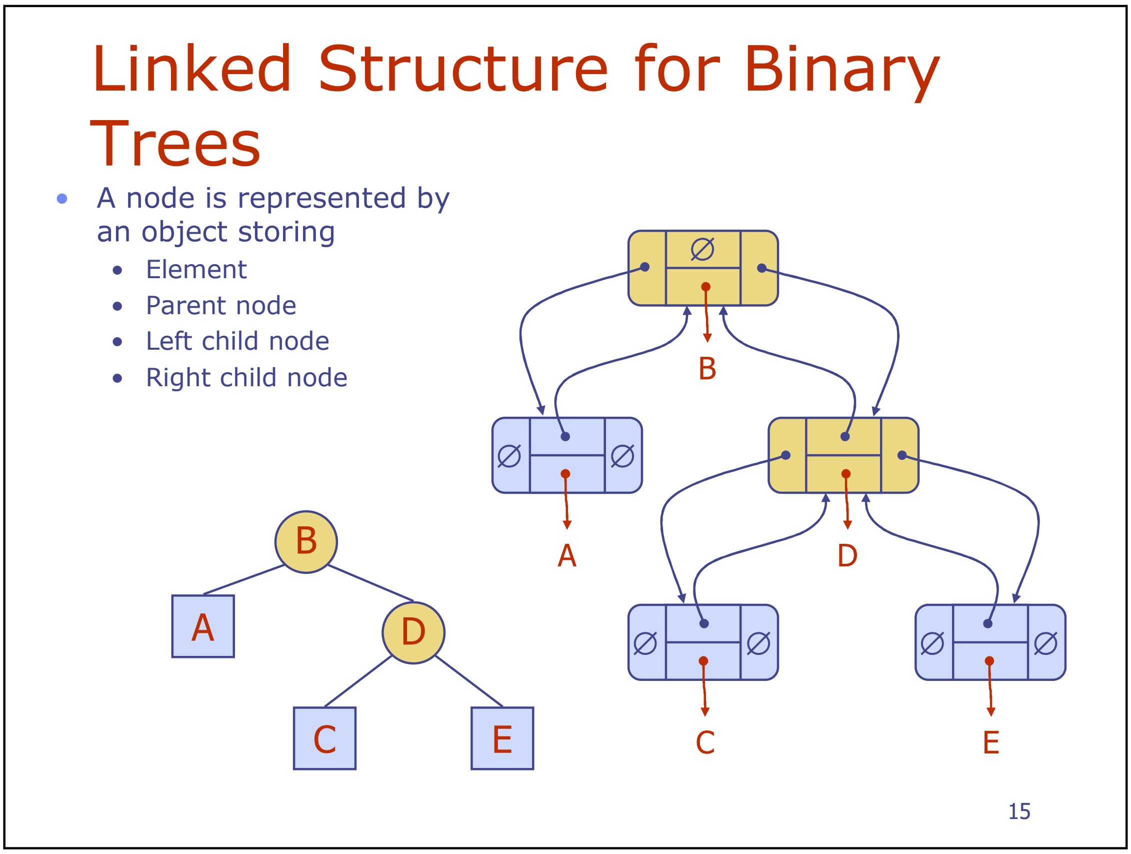


Linked Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes

How do we actually store all the tree.
The system is like a linked list, with nodes and links between them, but more complex.
When the number of children in arbitrary, then we want to store pointers to them in a little array.
The result looks like horrible spaghetti mess, and does mean that such code can be horrible to debug, it is easy to get a few pointers wrong.
However, the general structure is just to have pointers to be able to reach the children, and also it may be useful to have pointers from each node back to the parent.

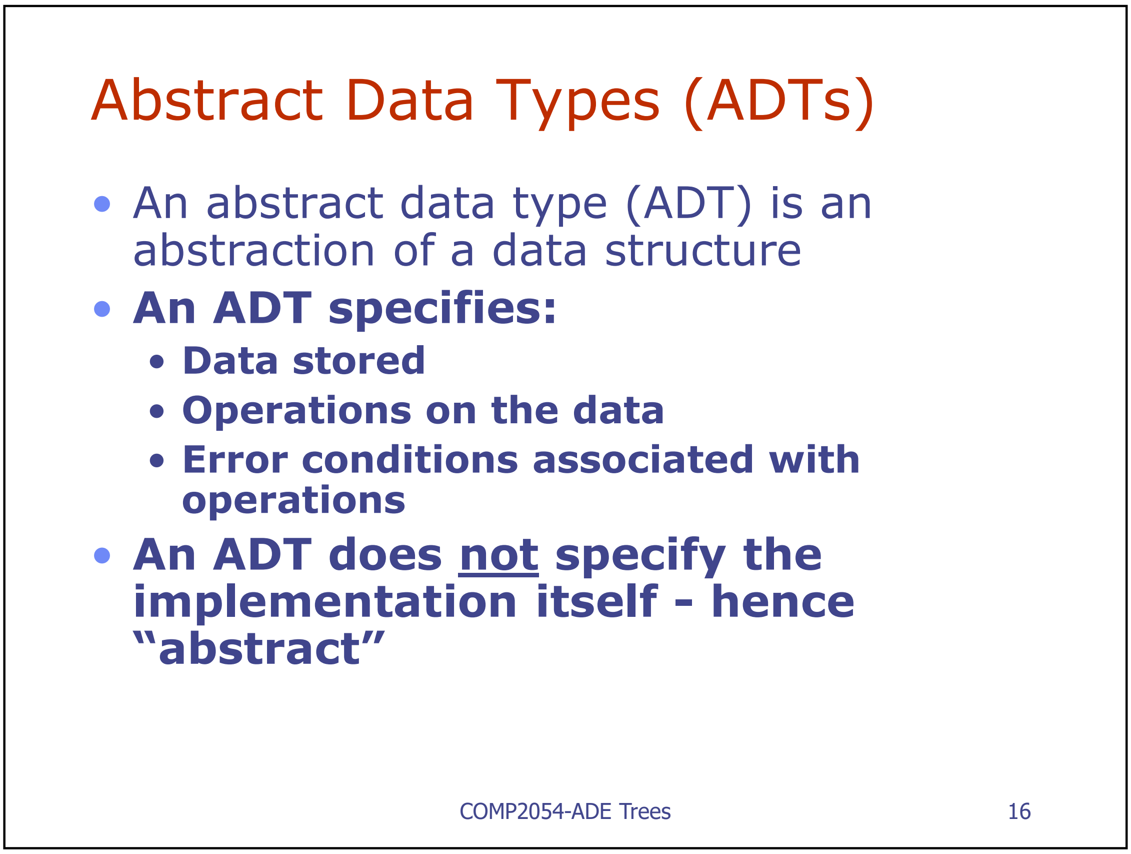
Note that the contents of nodes are also not stored in the node itself here but as a pointer.



Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node

In a binary tree it becomes more useful to put left and right together.

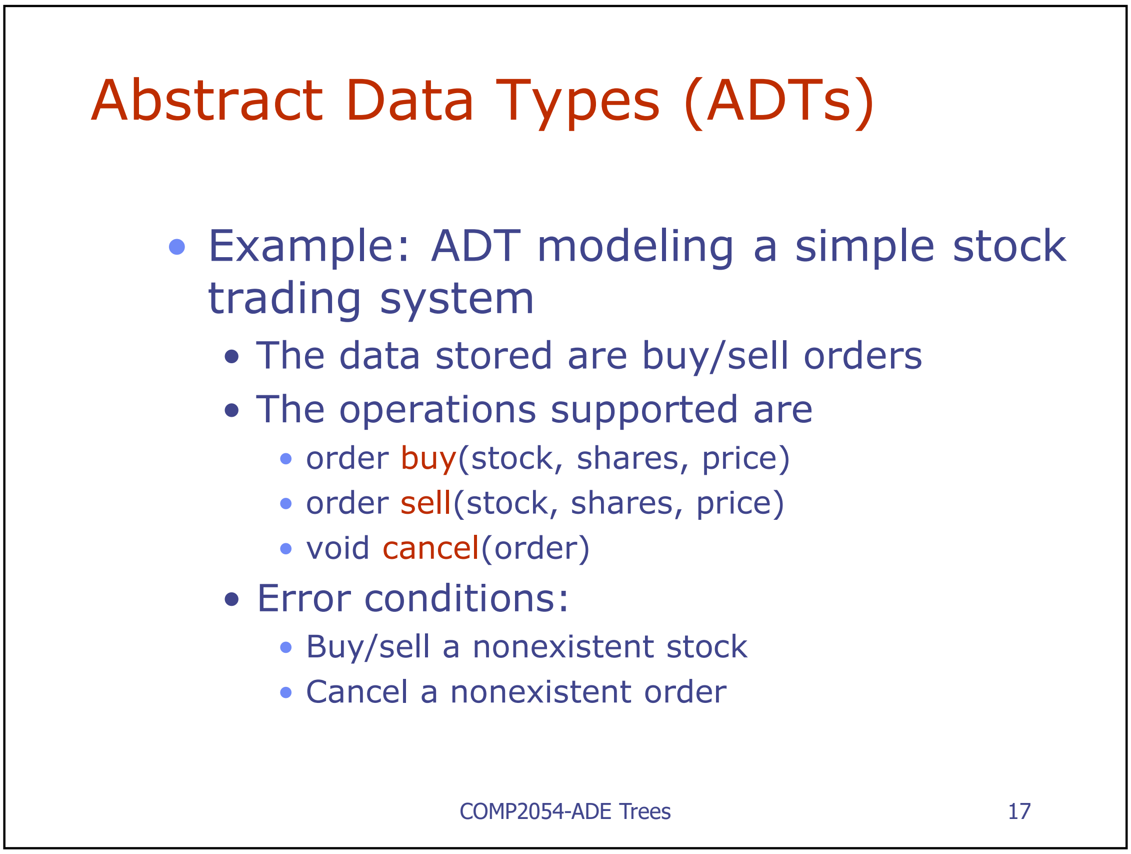


Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- **An ADT specifies:**
  - **Data stored**
  - **Operations on the data**
  - **Error conditions associated with operations**
- **An ADT does not specify the implementation itself - hence "abstract"**

As a quick aside we just mention that there is a concept of an "Abstract data type". The idea is familiar from object oriented programming and just enables to separate the external behaviour from the internal representation.

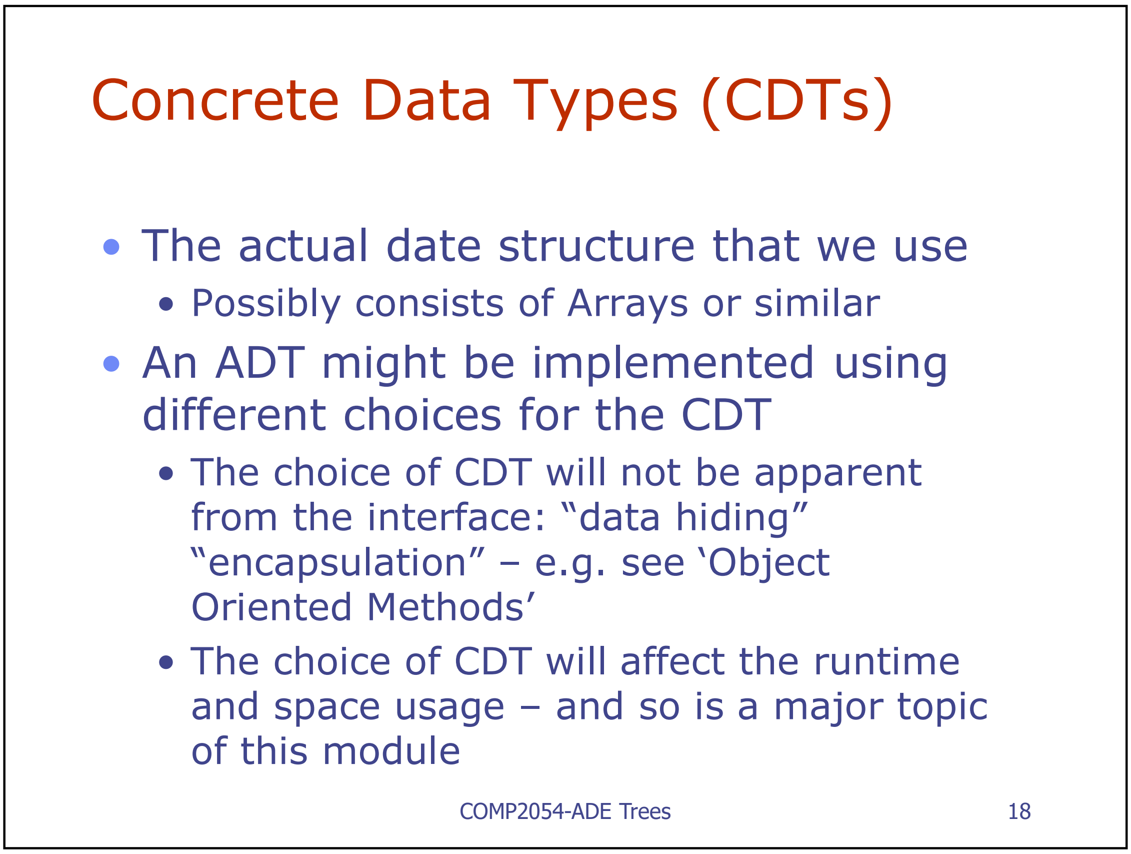
Think of it as the public interface of a class.



Abstract Data Types (ADTs)

- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

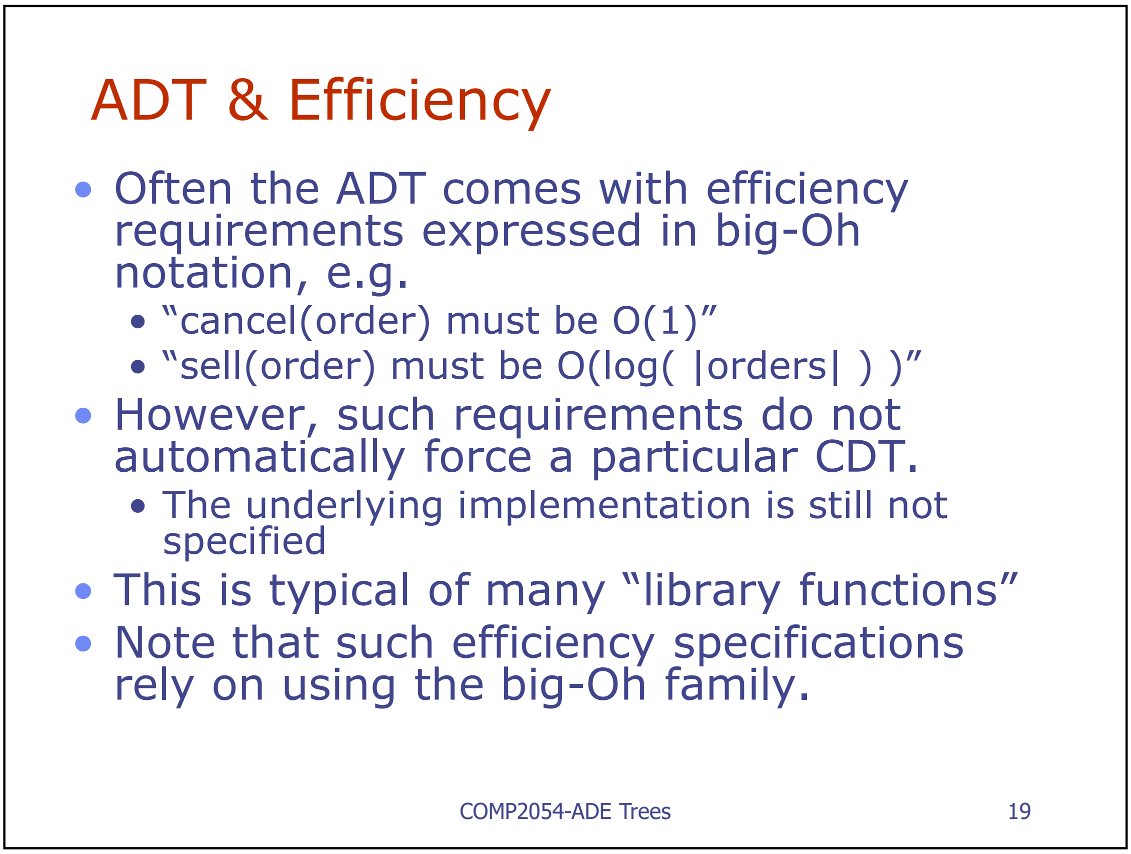
For example in stocks the main interface might be very simple buy sell cancel.



Concrete Data Types (CDTs)

- The actual date structure that we use
  - Possibly consists of Arrays or similar
- An ADT might be implemented using different choices for the CDT
  - The choice of CDT will not be apparent from the interface: “data hiding” “encapsulation” – e.g. see ‘Object Oriented Methods’
  - The choice of CDT will affect the runtime and space usage – and so is a major topic of this module

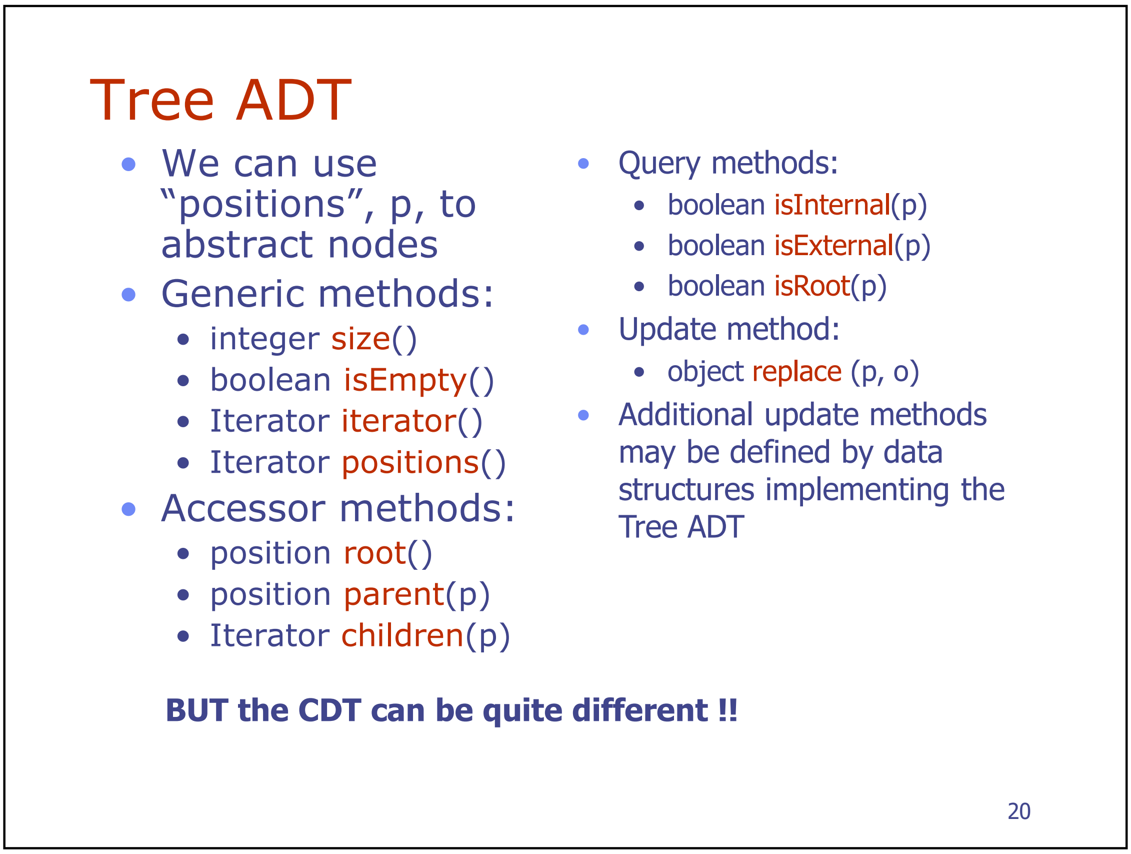
The actual implementation is called a “concrete data type” or CDT.
The main point is that the CDT might well be quite different from the “natural choice” that the ADT would suggest.



ADT & Efficiency

- Often the ADT comes with efficiency requirements expressed in big-Oh notation, e.g.
  - “cancel(order) must be O(1)”
  - “sell(order) must be O(log(|orders|))”
- However, such requirements do not automatically force a particular CDT.
  - The underlying implementation is still not specified
- This is typical of many “library functions”
- Note that such efficiency specifications rely on using the big-Oh family.

ADT usually comes with requirements of speed, and naturally these are expressed in terms of the Big-Oh family.



Tree ADT

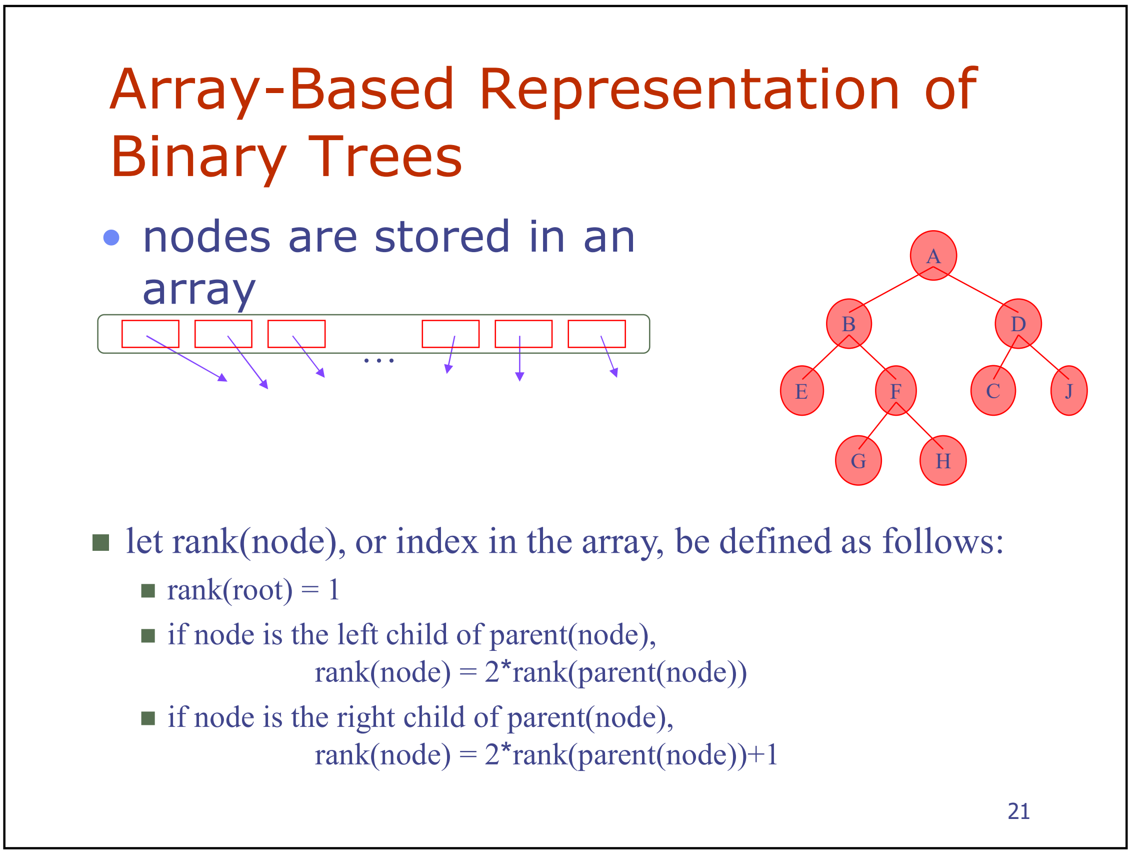
- We can use “positions”, p, to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterator positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)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

BUT the CDT can be quite different !!

20

As an example for a tree the ADT might just use some notion of a position, but without actually saying how it is represented internally.
Then the ADT interface ahs to be able to scan the tree in a reasonable fashion.
For example being able to scan the children or recognise whether a node is a leaf, etc.

But the point we are getting to is that the CDT might look quite different.



Array-Based Representation of Binary Trees

- nodes are stored in an array

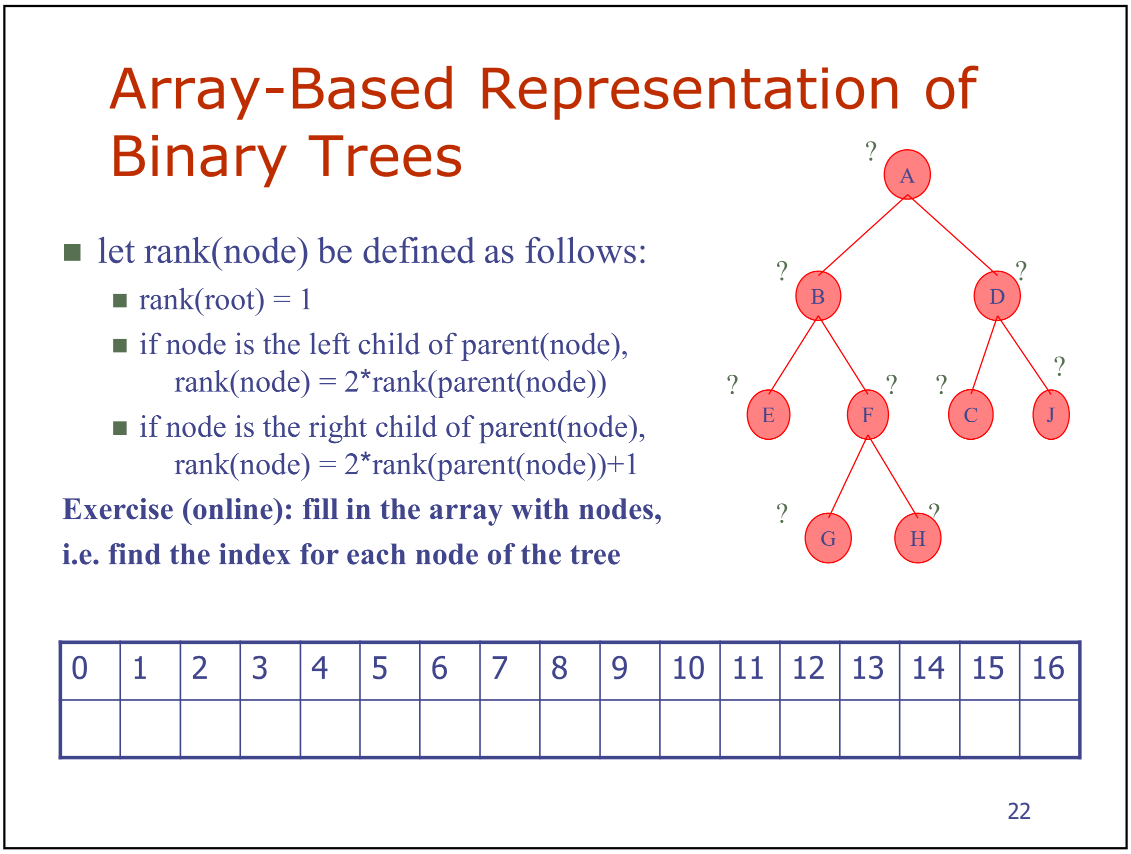
- let rank(node), or index in the array, be defined as follows:
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2*rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2*rank(parent(node))+1

21

So a particularly useful case is when a BINARY tree is stored by using an array as the CDT.
This can be quite confusing, because even when stored as an array, the access to the data should be in the fashion of a tree.

A standard way to do this is given here

We need to be able to map a tree to the array in such a way that the full structure of the tree is retained.
We decide to put the root as index 1 – it is not essential but makes the formulas easier.
Then the left child is at double the parent index, and the right child is one further along.



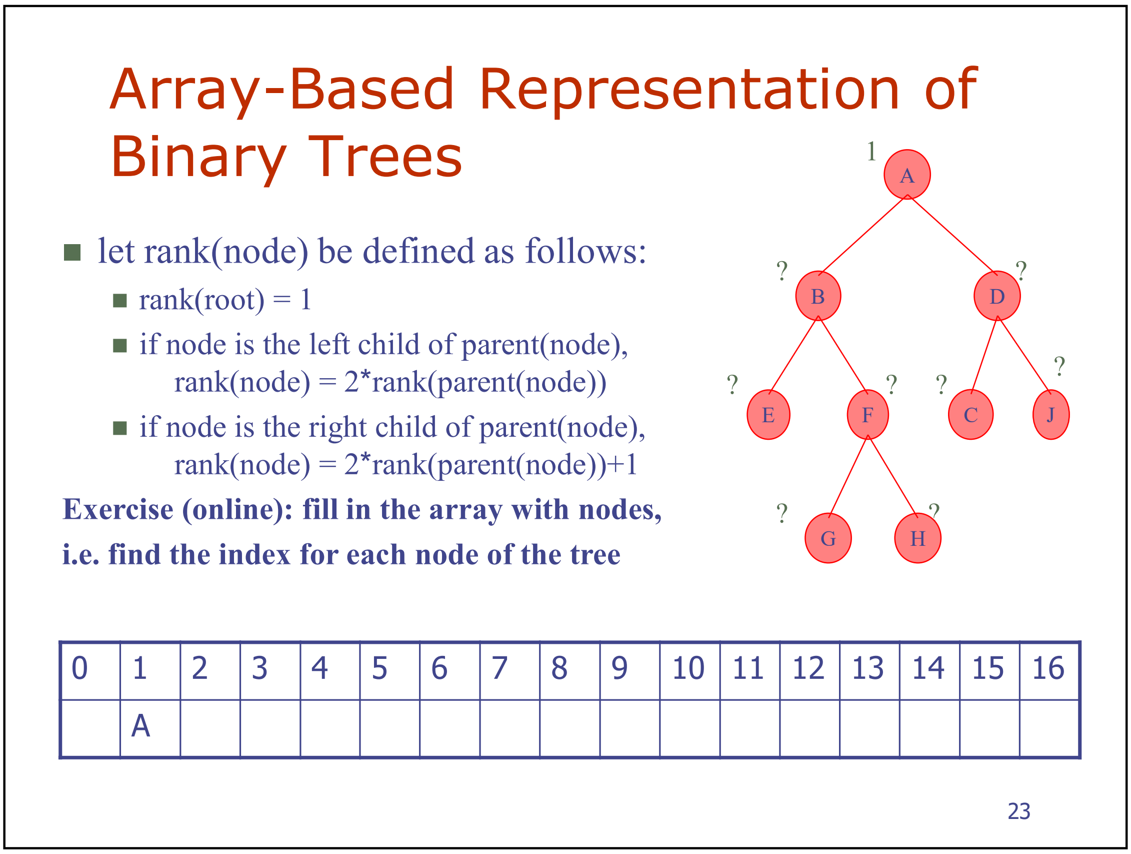
Array-Based Representation of Binary Trees

- 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

This is best done via an example and so will give you a brief pause to try to fill in some of the entries.



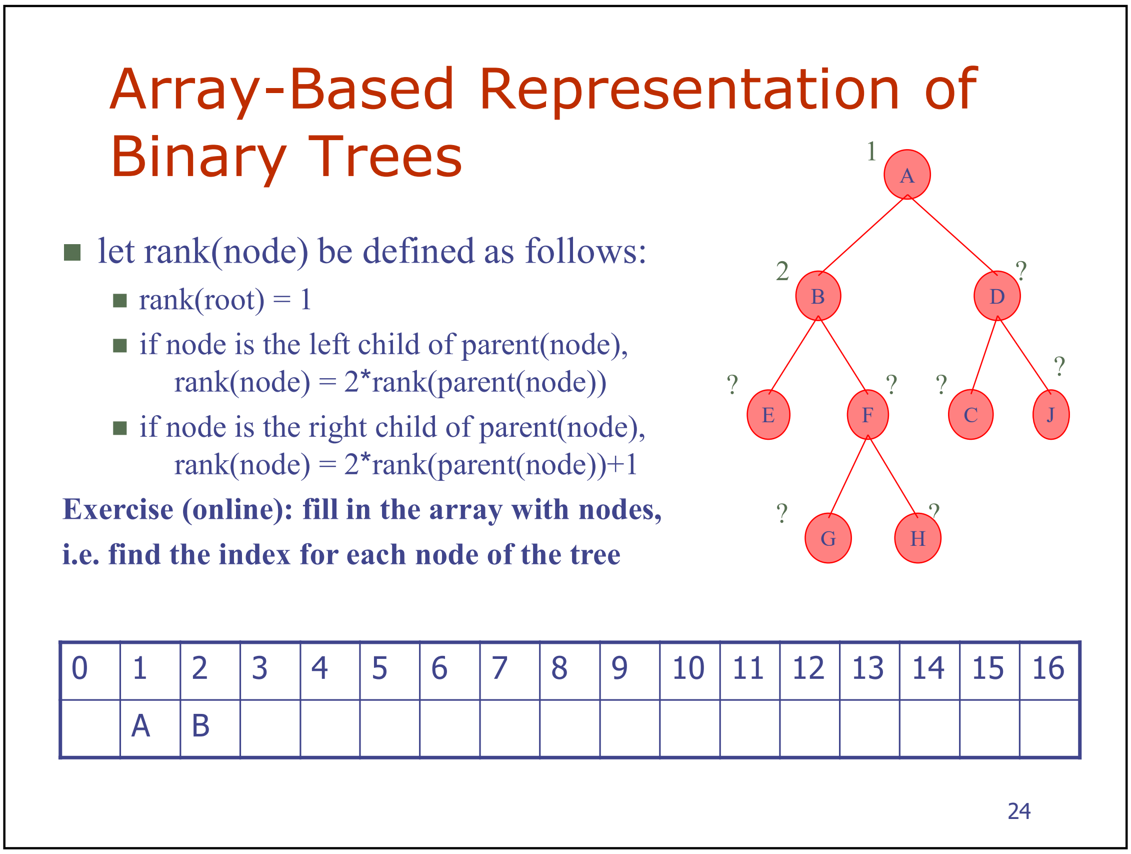
Array-Based Representation of Binary Trees

- 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															

Root is A and goes at 1



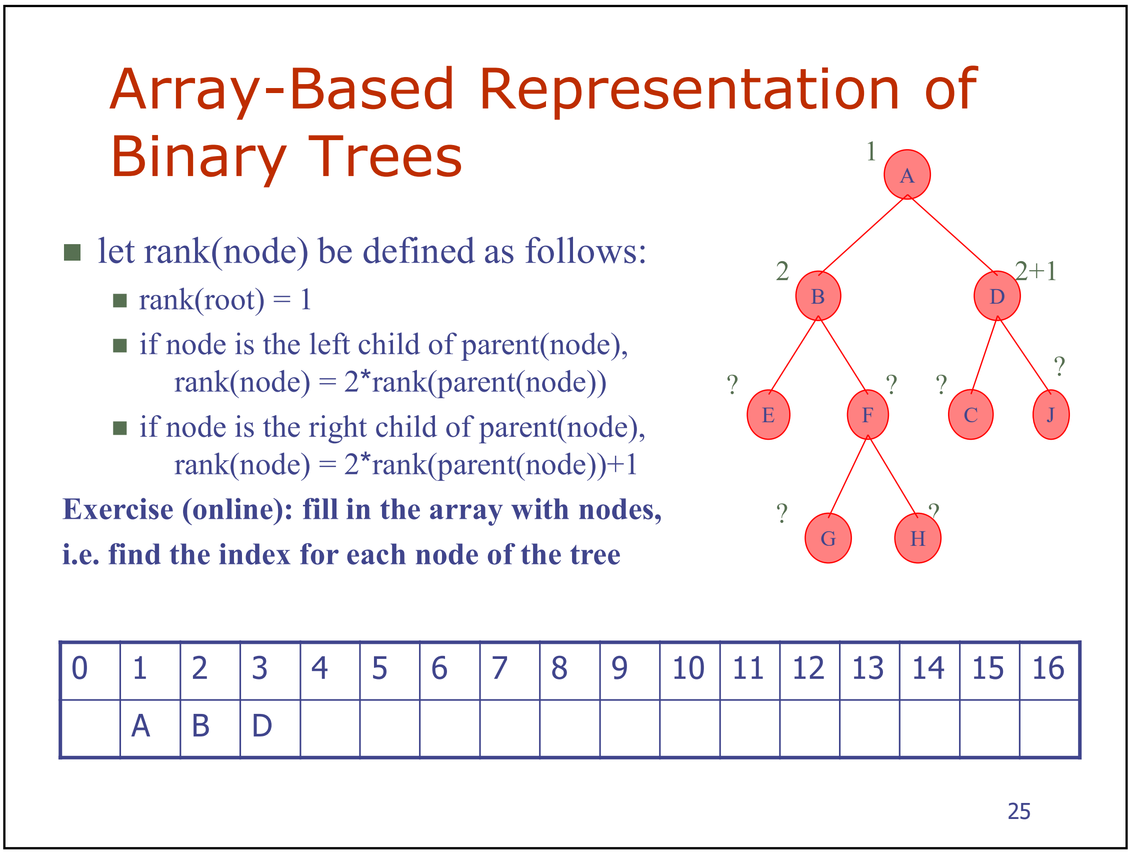
Array-Based Representation of Binary Trees

- 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
	A	B														

B is a left child and so goes at 2*iindex(A) = 2



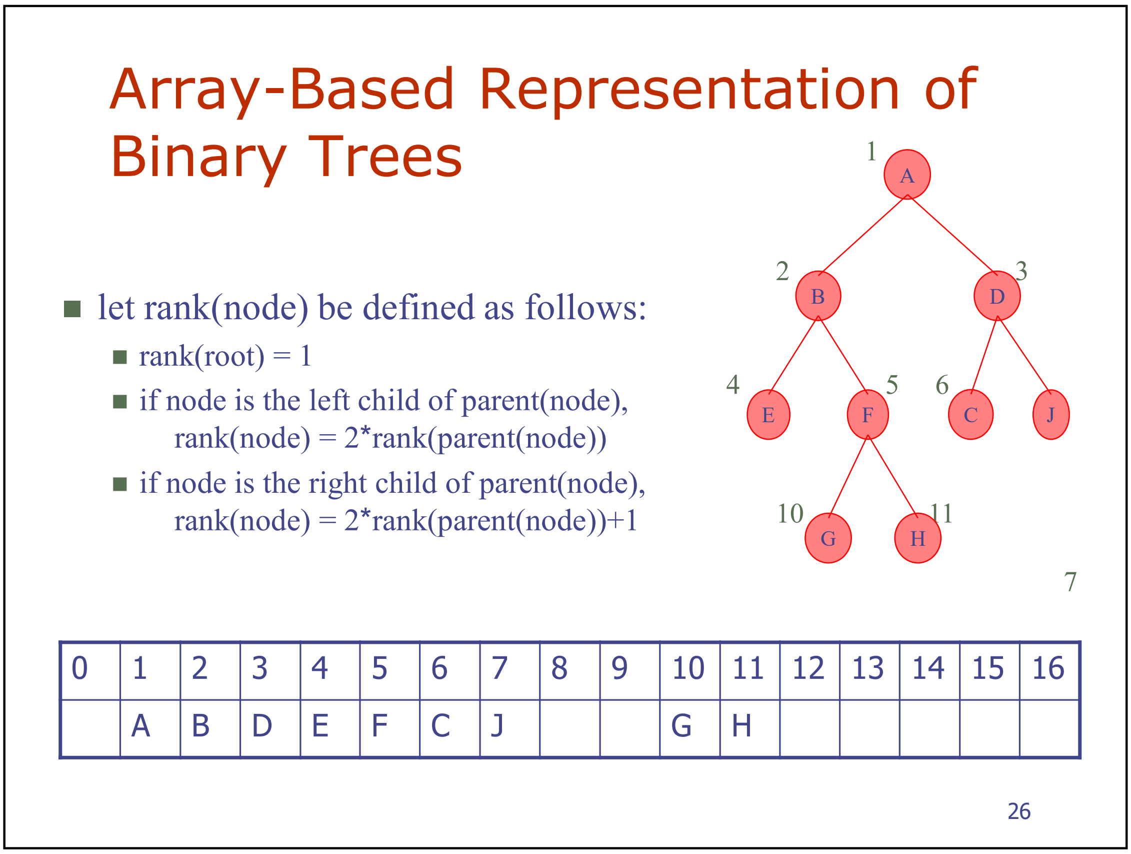
Array-Based Representation of Binary Trees

- 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													

Right child goes at 2*1+1



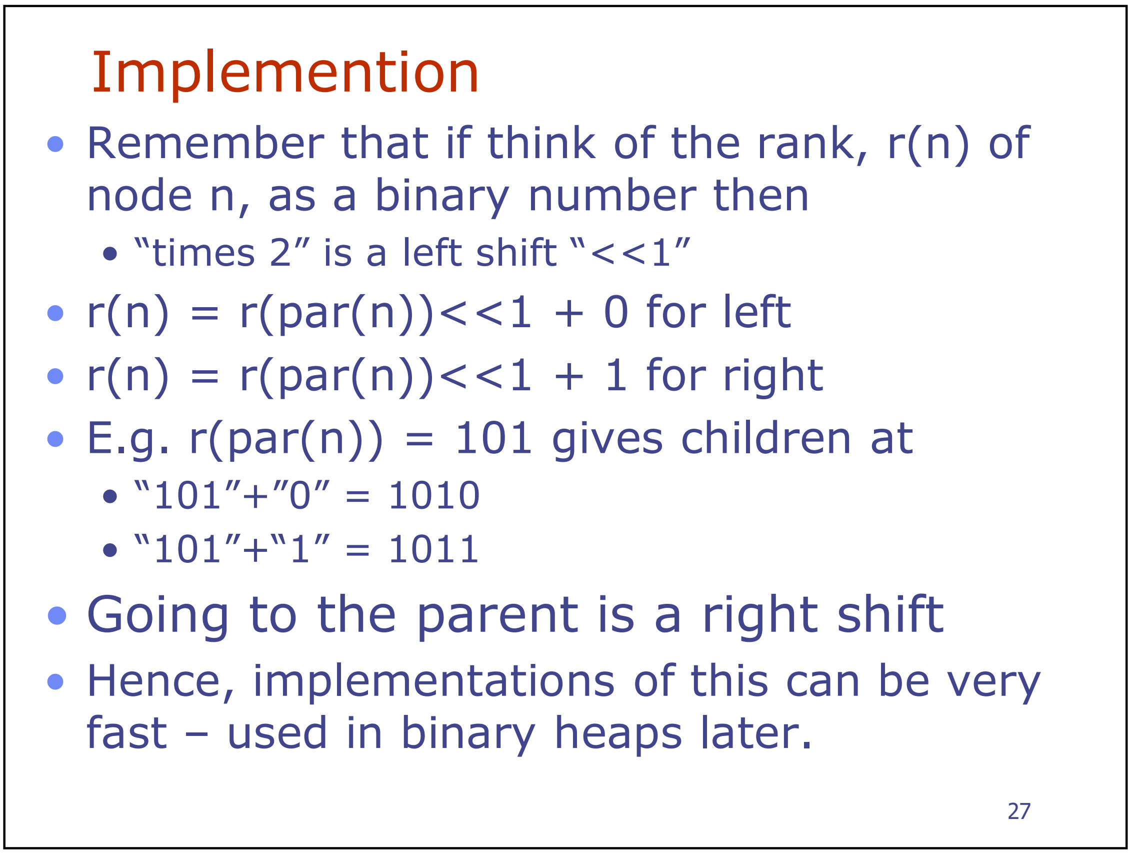
Array-Based Representation of Binary Trees

- 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

1
A
2
B
3
D
4
E
5
F
6
C
J
10
G
11
H
7

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					

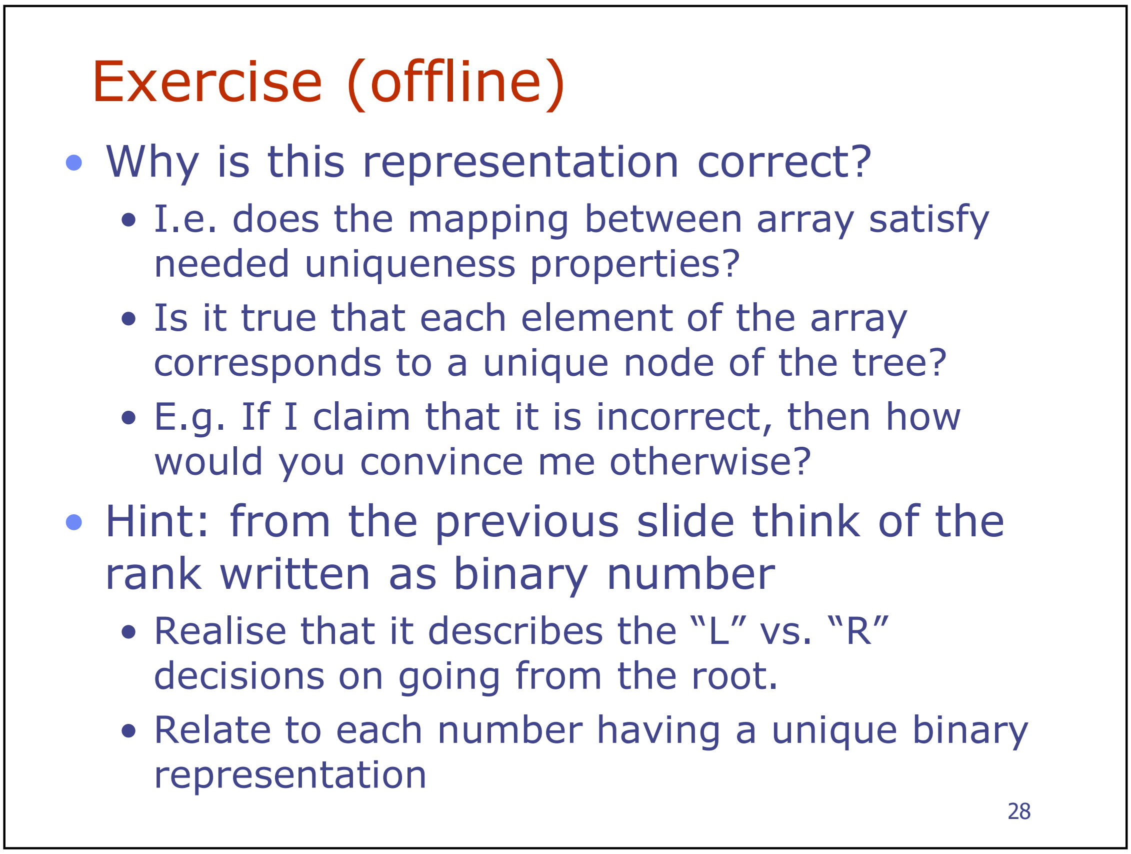
We can continue to get the result.
Notice that there can be gaps, and so some entries of the array are ‘empty’ – maybe marked with a null pointer, or similar indicator of “empty”.



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 later.

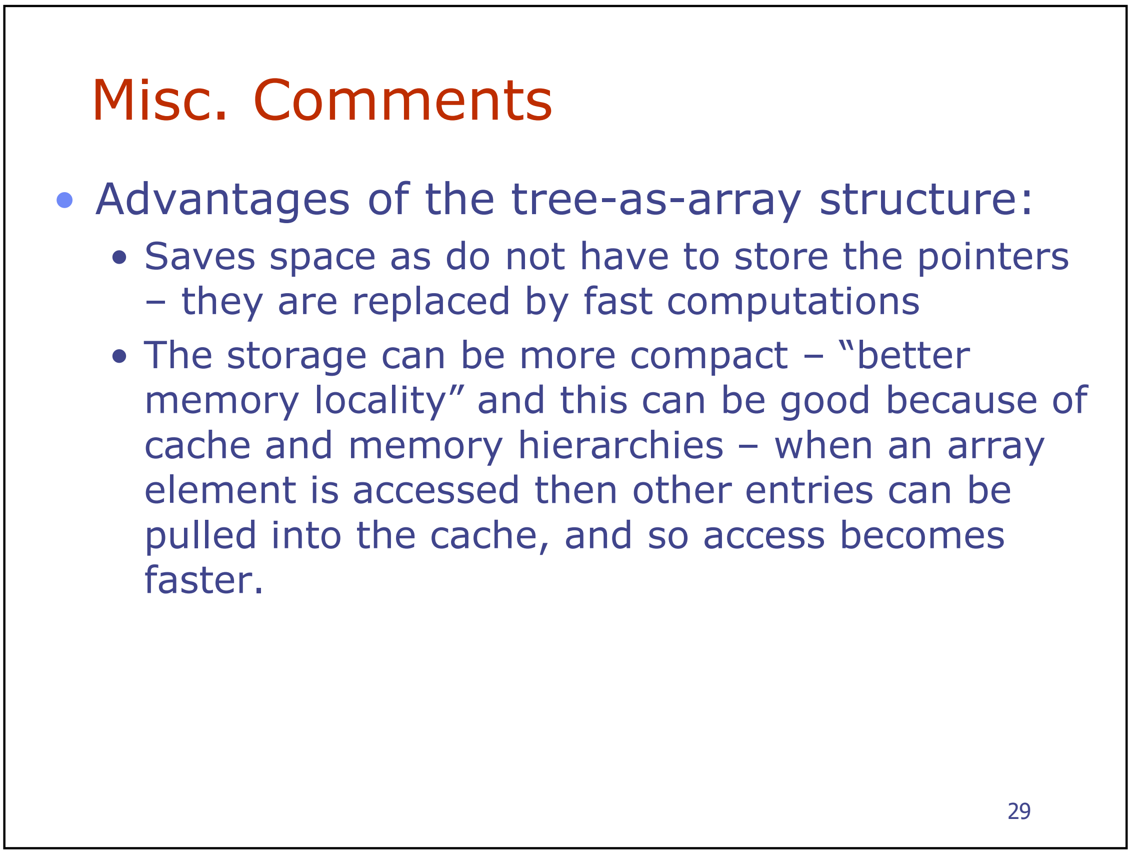
The system is very good because the computation can be very fast using shifts and simple increments.
Note that going to the parent of a node is just a “integer divide by two” and so is just a right shift.



Exercise (offline)

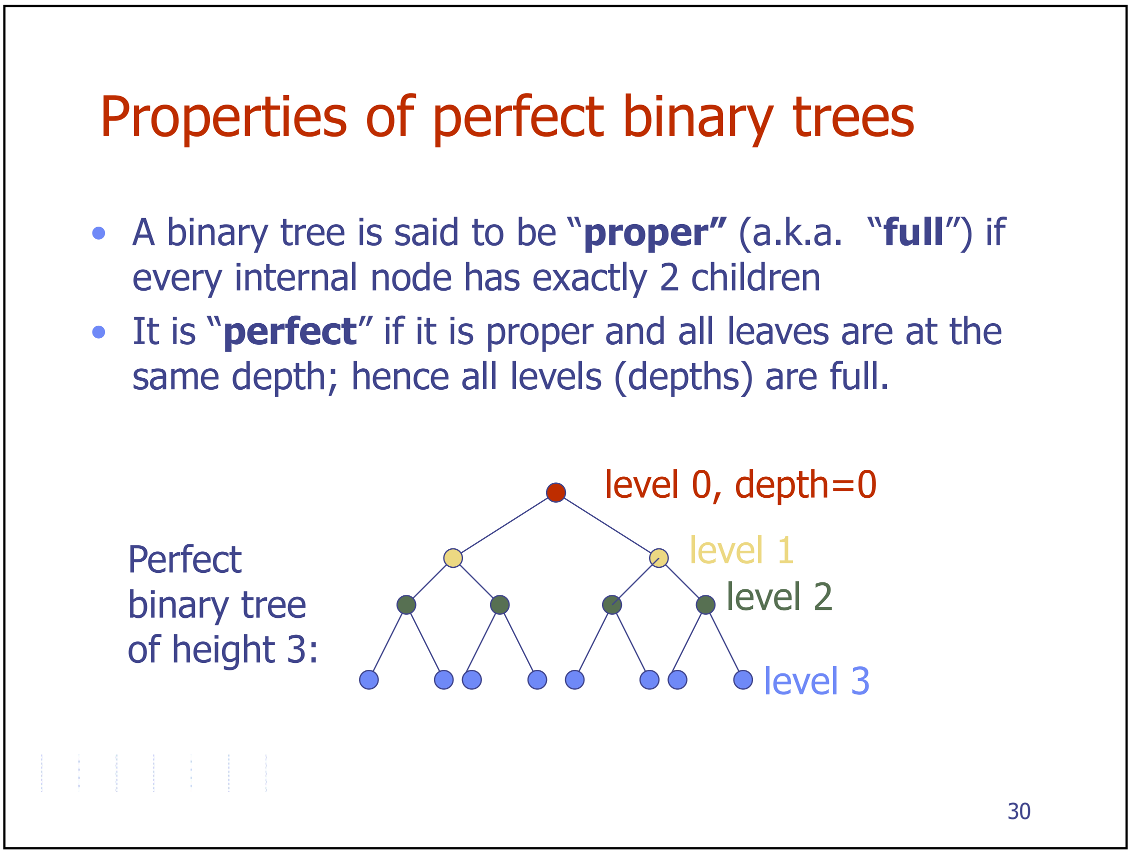
- Why is this representation correct?
  - I.e. does the mapping between array satisfy needed uniqueness properties?
  - Is it true that each element of the array corresponds to a unique node of the tree?
  - E.g. If I claim that it is incorrect, then how would you convince me otherwise?
- Hint: from 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

Something to think about is why this representation actually does work properly.
In particular it is important that different nodes of any tree always map to different indices.
This is not quite as trivial as it might seem.



Misc. Comments

- Advantages of the tree-as-array structure:
  - Saves space as do not have to store the pointers – they are replaced by fast computations
  - The storage can be more compact – “better memory locality” and this can be good because of cache and memory hierarchies – when an array element is accessed then other entries can be pulled into the cache, and so access becomes faster.



Properties of perfect binary trees

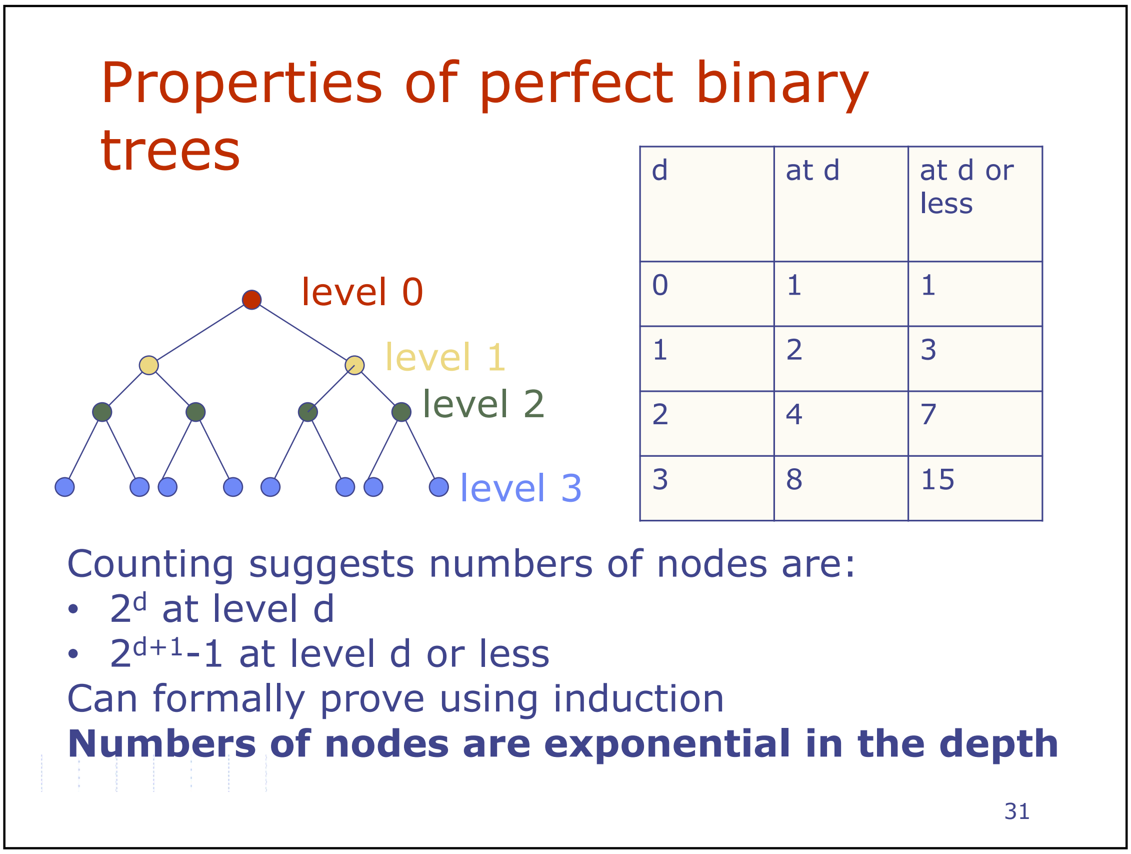
- A binary tree is said to be "**proper"** (a.k.a. "**full**") if every internal node has exactly 2 children
- It is "**perfect**" if it is proper and all leaves are at the same depth; hence all levels (depths) are full.

Perfect binary tree of height 3:

level 0, depth=0
level 1
level 2
level 3

30

Now move to a special case of trees which are called "perfect" – this simply means that we are doing a tree that is totally full down to some depth.
For a given height, it has the most nodes possible.



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

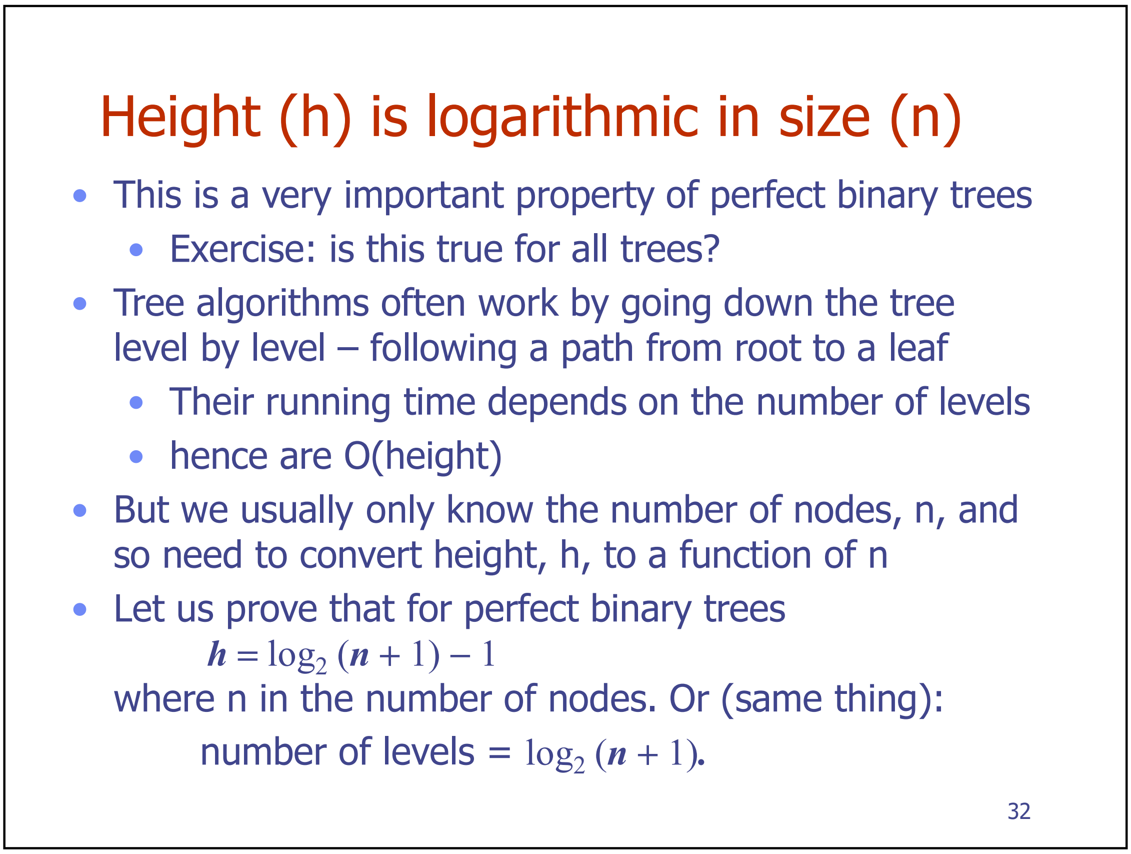
Can formally prove using induction

Numbers of nodes are exponential in the depth

What we want to know is how the numbers of nodes varies with the depth.
We distinguish between nodes with precisely some depth, and nodes with at most some depth.
As a usual technique, it is a good idea to just do some numbers from simple examples.
This is just simple counting, and gives the numbers in the table.
Now looking at the numbers we can try to match them to a pattern, and come up with a hypothesis for a general formula.

Looking at the numbers the "at d" is clearly powers of 2 – ie. 2^d.
As usual remember that 2^0 = 1.
The "at d or less" is then a sum of powers, and remembering the geometric series we can expect it to be "the next term, minus one"

This is strong evidence. For a proof we need 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
  - Their running time depends on the number of levels
  - hence are O(height)
- But we usually only know the number of nodes, n, and so need to convert height, h, to a function of n
- Let us prove that for perfect binary trees

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

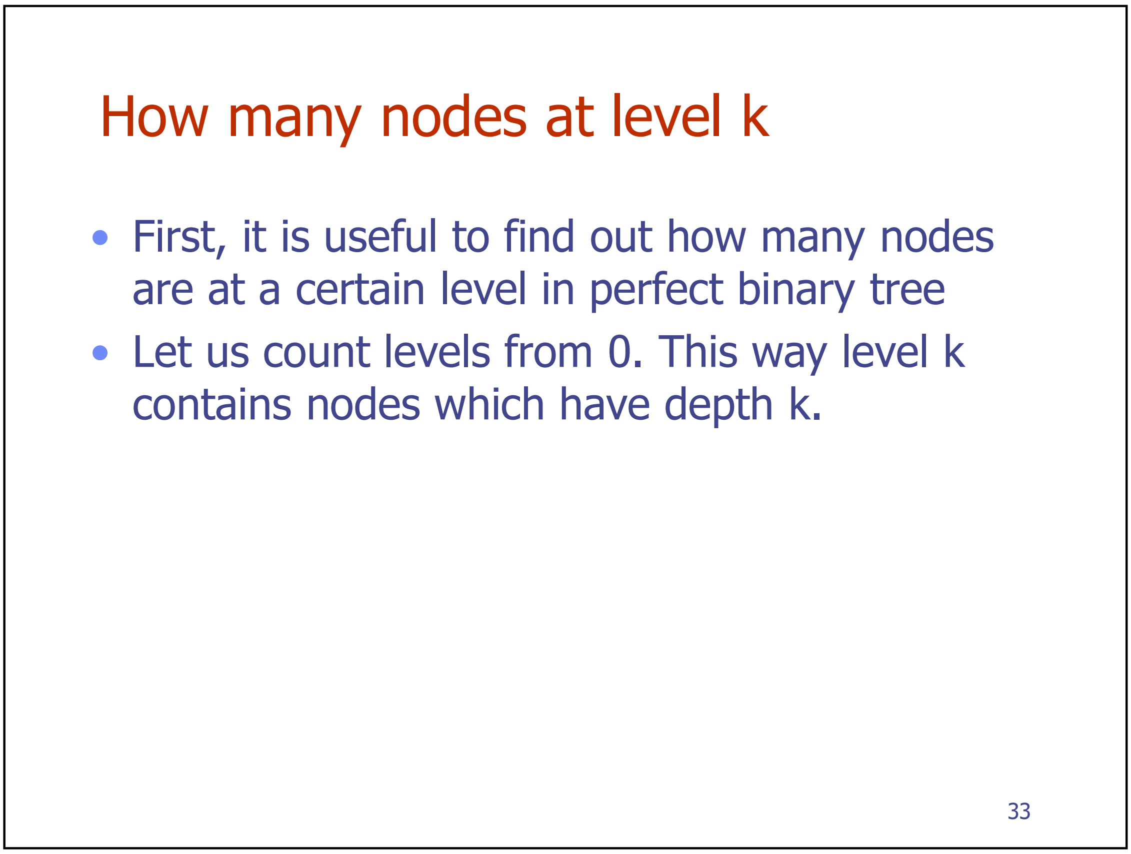
  where n in the number of nodes. Or (same thing):

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

32

Firstly, assuming the results are correct, what is the impact, and why do we care.

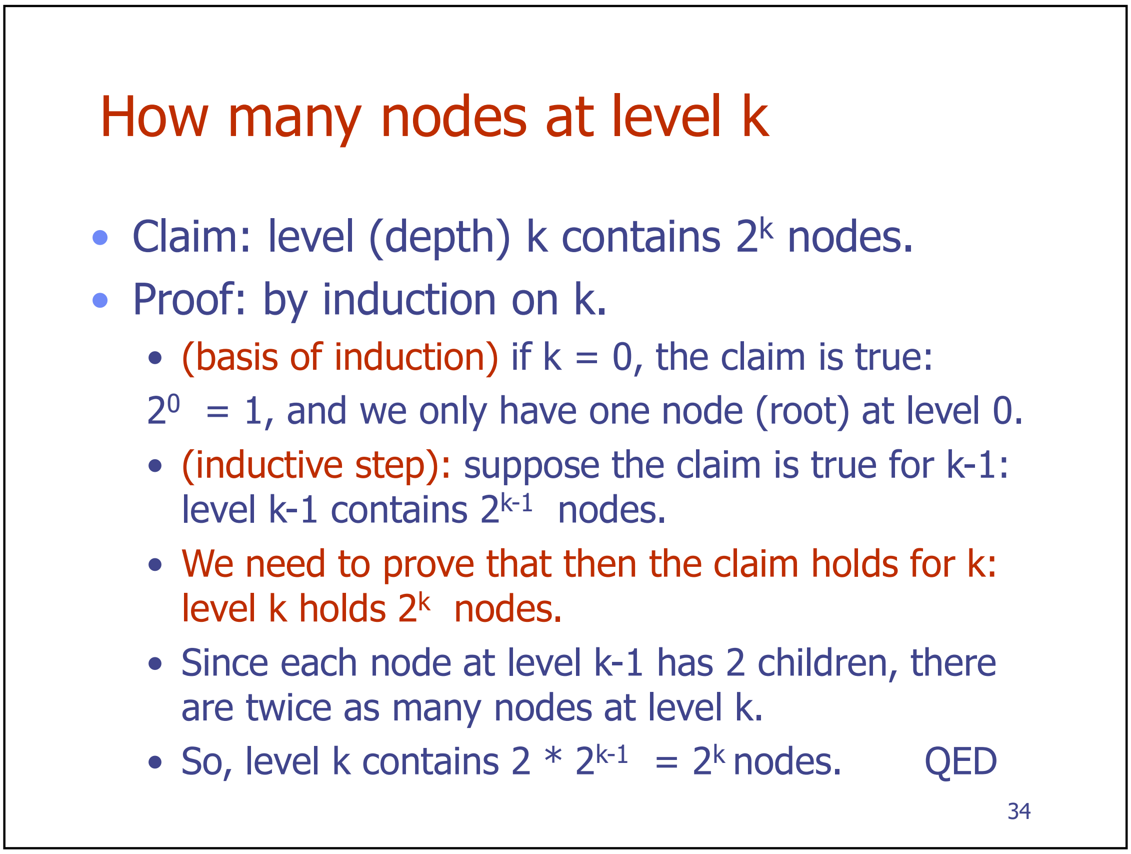
Since the nodes are exponential in the height then the height is logarithmic in the number of nodes.
We care because many algorithms work by following paths from a leaf to the root or vice versa.
And so are naturally big-Oh of the height h.
But then we want to express height in terms of the number of nodes.
For this we will show the formula given.



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.

As before in the example we split

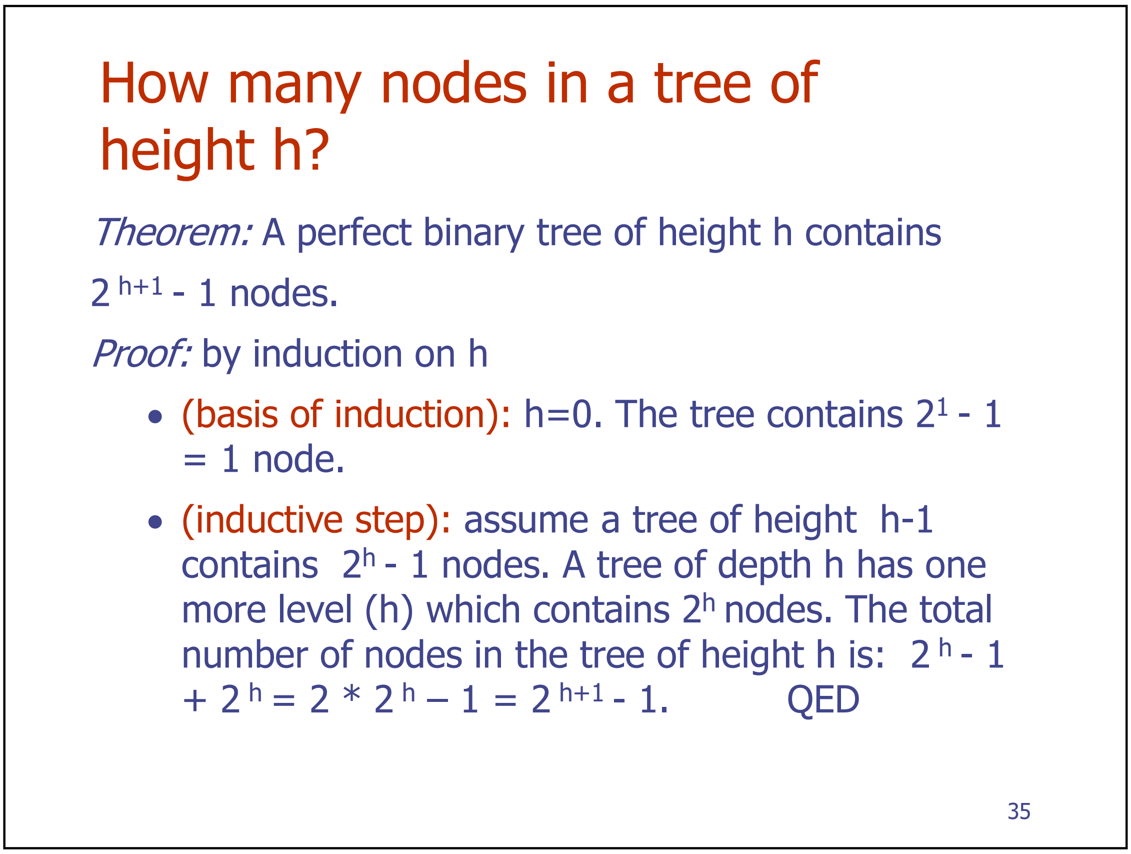


How many nodes at level k

- Claim: level (depth) 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

The claim, as we did before the we have a simple formula.
We need to prove it via induction.
At the base case it is trivially true.
The step case is also straight forward, and follows a standard pattern.
If you have any worries about the general procedure for doing proofs by induction then very carefully study this proof and make sure you understand the structure of the proof.
We will use induction proofs next week in doing recurrence relations.



How many nodes in a tree of height h?

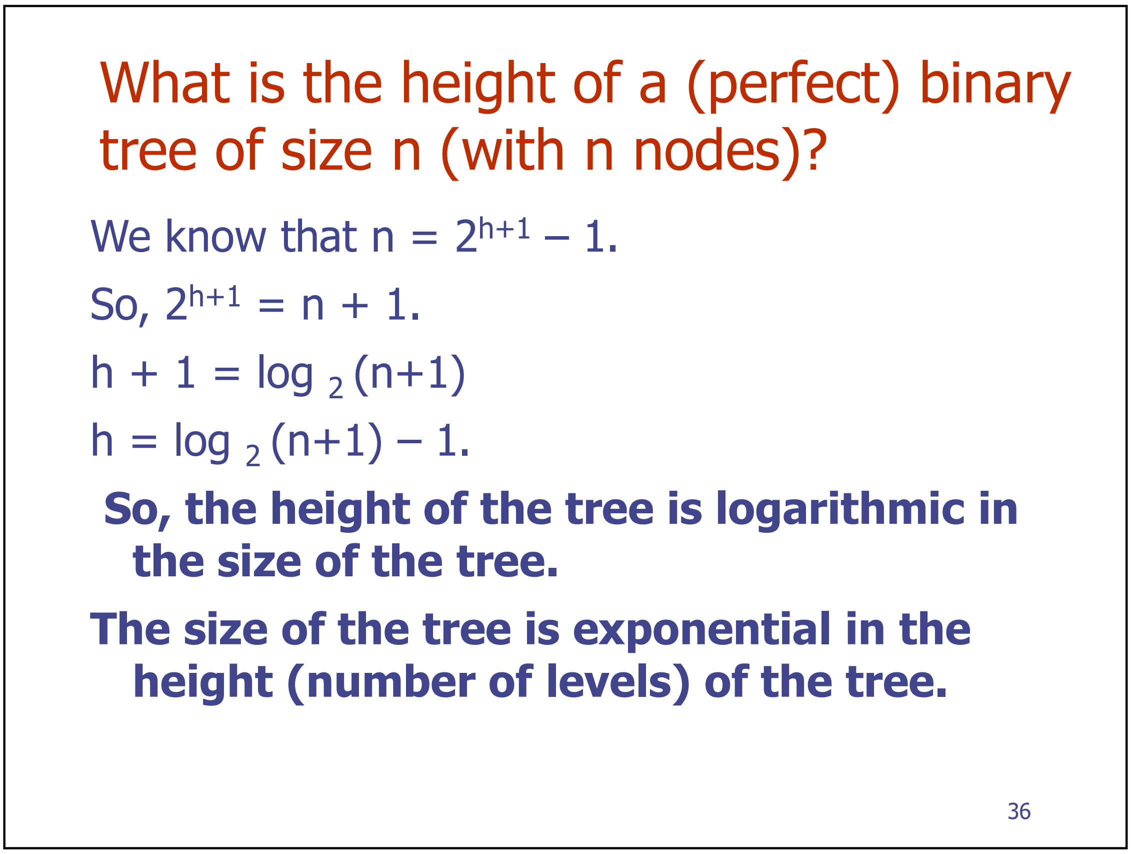
Theorem: A perfect binary tree of height h contains

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

Now lets do the total number of nodes.
Similar structure.
The base case works find.
The step case needs a bit of algebra.
Again I strongly suggest to work through this yourself.
Study it and then see if you can reproduce it the day after, without looking at notes.



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

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

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

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

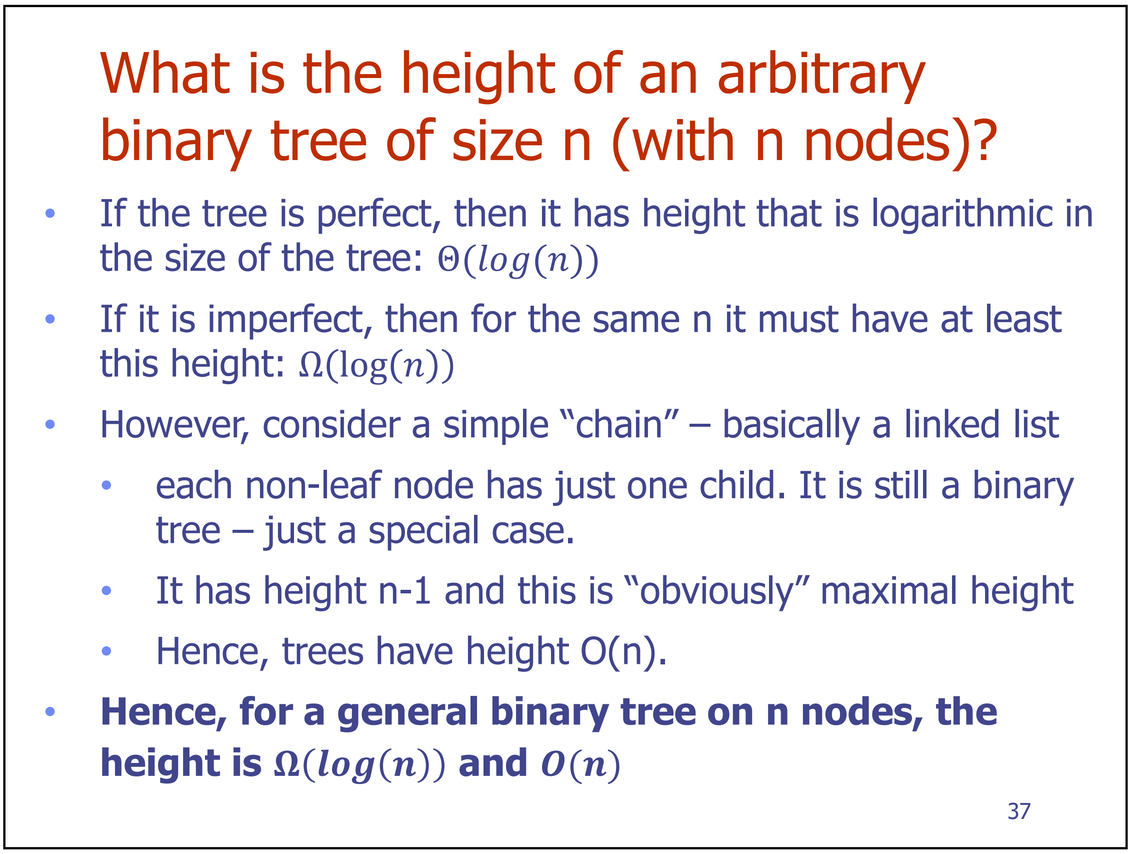
$h = \log_2(n+1) - 1$.

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.

We have to invert the function, which is just easy algebra – assuming you are happy with logs.

The main impact, as written is that the height of a perfect tree scales with the log of the nodes.



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

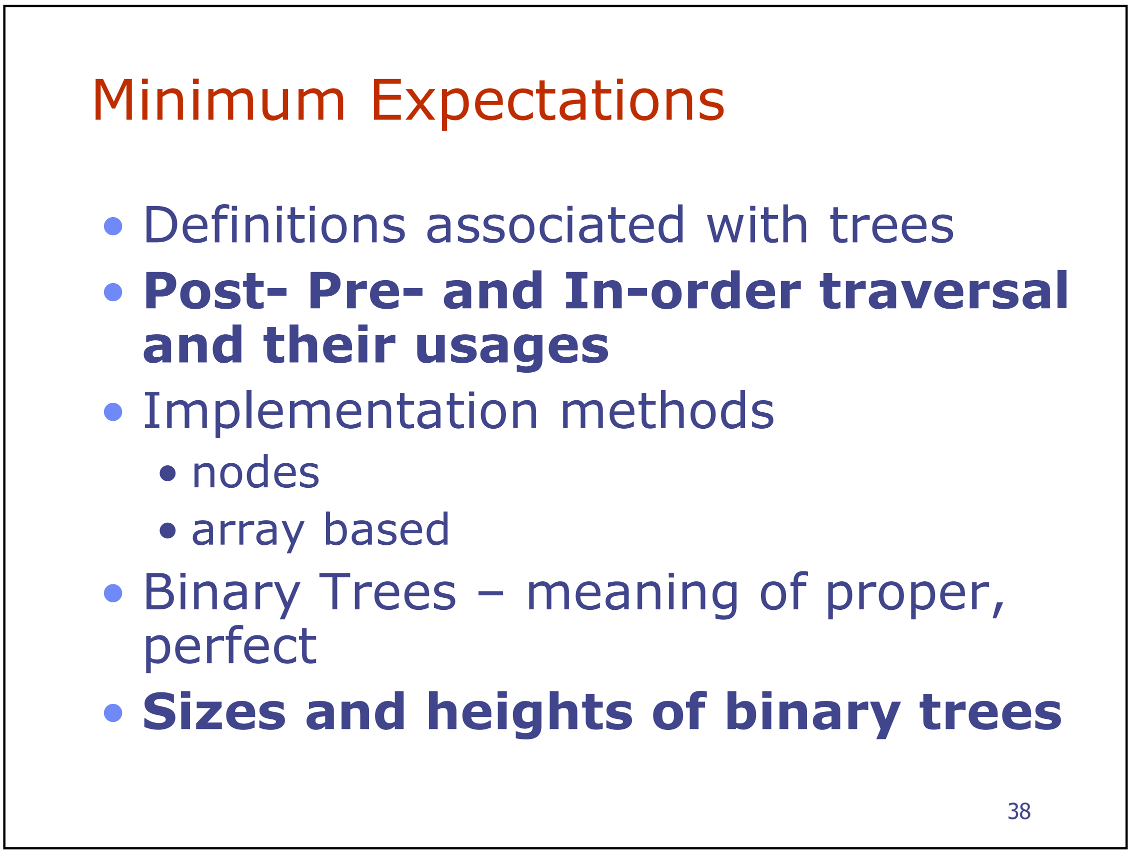
- If the tree is perfect, then it has height that is logarithmic in the size of the tree: $\Theta(log(n))$
- If it is imperfect, then for the same n it must have at least this height: $\Omega(\log(n))$
- However, consider a simple "chain" – basically a linked list
  - each non-leaf node has just one child. It is still a binary tree – just a special case.
  - It has height n-1 and this is "obviously" maximal height
  - Hence, trees have height O(n).
- **Hence, for a general binary tree on n nodes, the height is $\boldsymbol{\Omega(log(n))}$ and $\boldsymbol{O(n)}$**

37

We were doing perfect trees. What about other trees?
Firstly, the perfect tree is the most densely packed – it has the largest possible number of nodes for a given height.
Hence, any other tree with n nodes must have at least the same h.
This is, succinctly, expressed by saying that the height of an arbitrary (binary) tree is OMEGA(n).
Note again how the big-Oh family allows to have freedom to hide details of the exact formulas, and so is a very useful way to express knowledge.

But how bad can the height get?
Worst case is when there is the worst possible packing – every depth has just one node.

Even though it is a chain and looks like a list, it is still a binary tree, just a special degenerate case.
Height is clearly O(n).
This gives the best and worst cases.
Height is Omega(log n) and O(n).
Note how the big-Oh family is used to limit.
Note we can only give a Theta when we know the exact upper bound on the height as well.



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**

So for trees it is vital to know traversals, methods to store them, and the properties in terms of Big-Oh family.