COMP251: DATA STRUCTURES & ALGORITHMS

* Some slides from “Algorithms and Data Structures”
by Douglas Wilhelm Harder

Binary Trees

Outline

This topic discusses the concept of a binary tree:

- Definitions
- Properties
- Applications

Definition

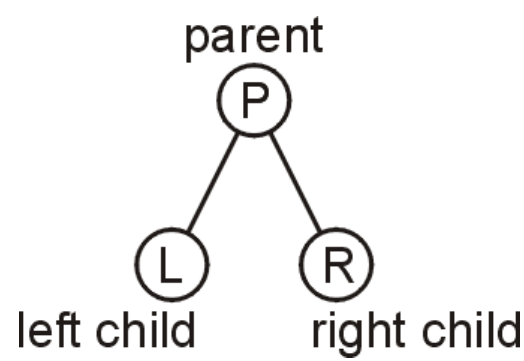
The arbitrary number of children in general trees is often *unnecessary* — many real-life trees are restricted to two branches

- Expression trees using binary operators
- An ancestral tree of an individual, parents, grandparents, *etc*.
- Phylogenetic trees
- Lossless encoding algorithms

Definition

A binary tree is a restriction where each node has exactly two children:

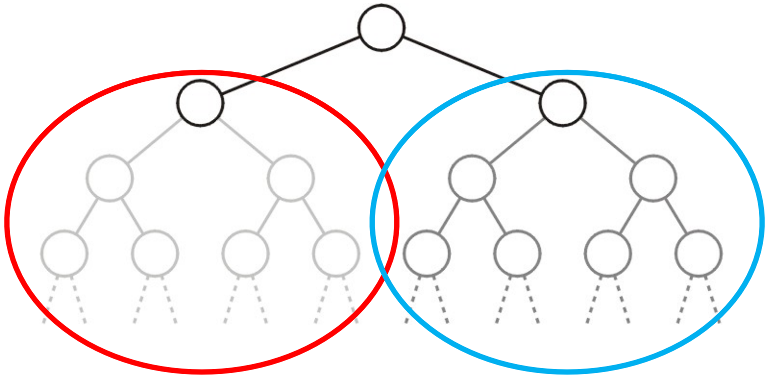
- Each child is either empty or another binary tree
- This restriction allows us to label the children as *left* and *right* subtrees



Definition

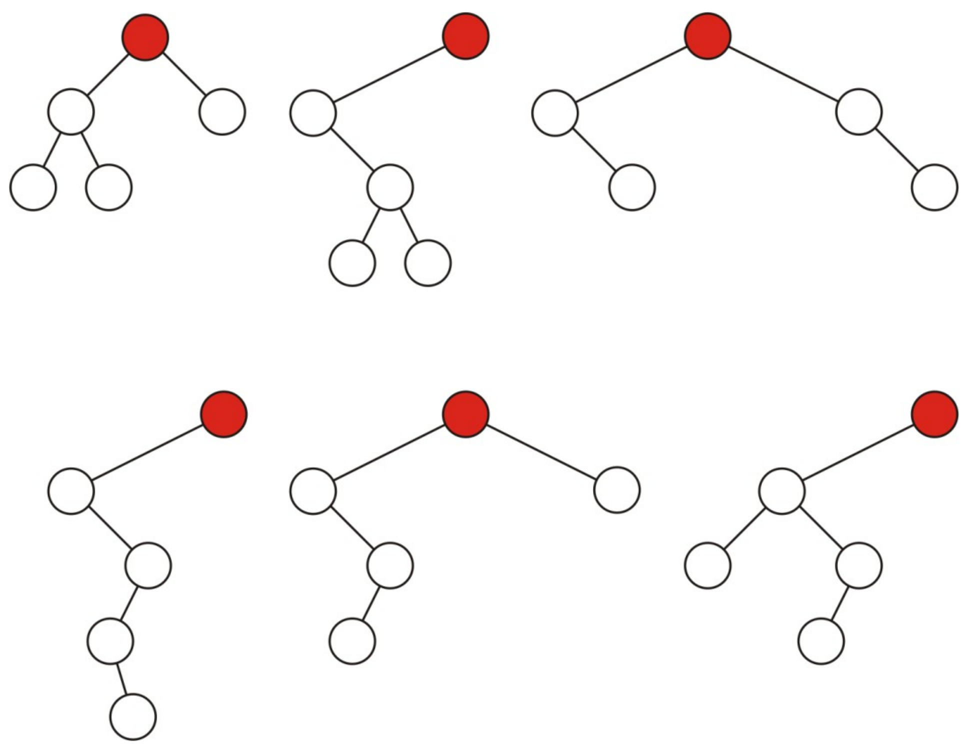
We will also refer to the two sub-trees as

- The left-hand sub-tree, and
- The right-hand sub-tree



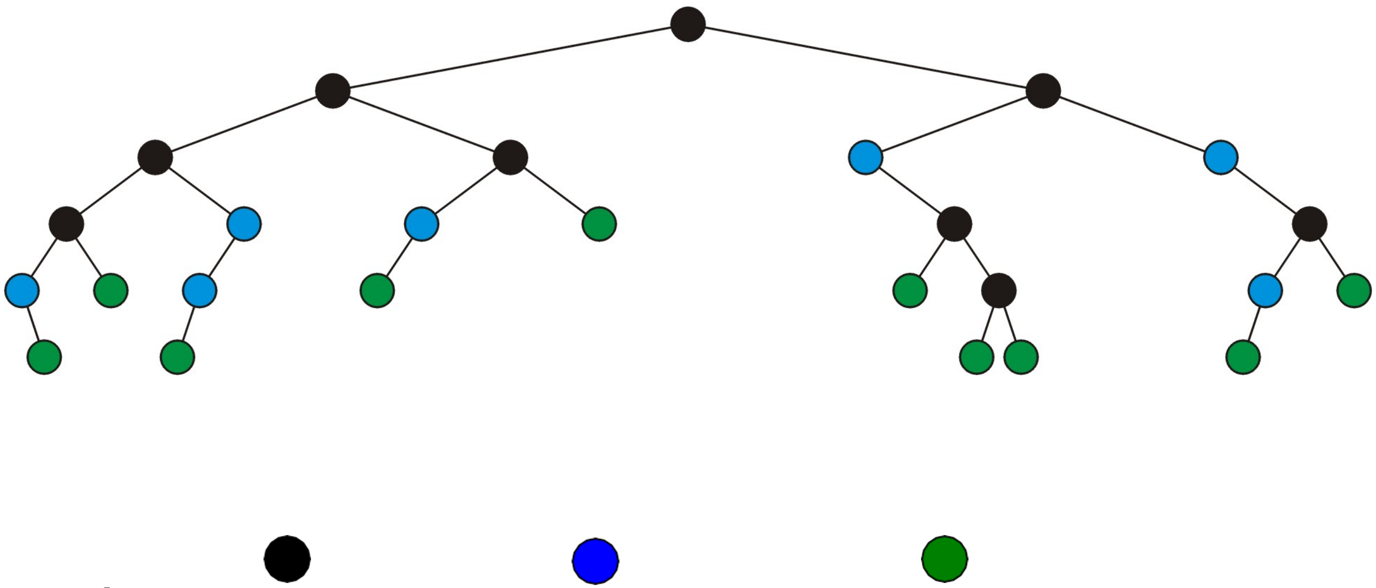
Definition

Sample variations on binary trees with five nodes: (root is shown in red)



Definition

A *full* node is a node where both the left and right sub-trees are non-empty trees

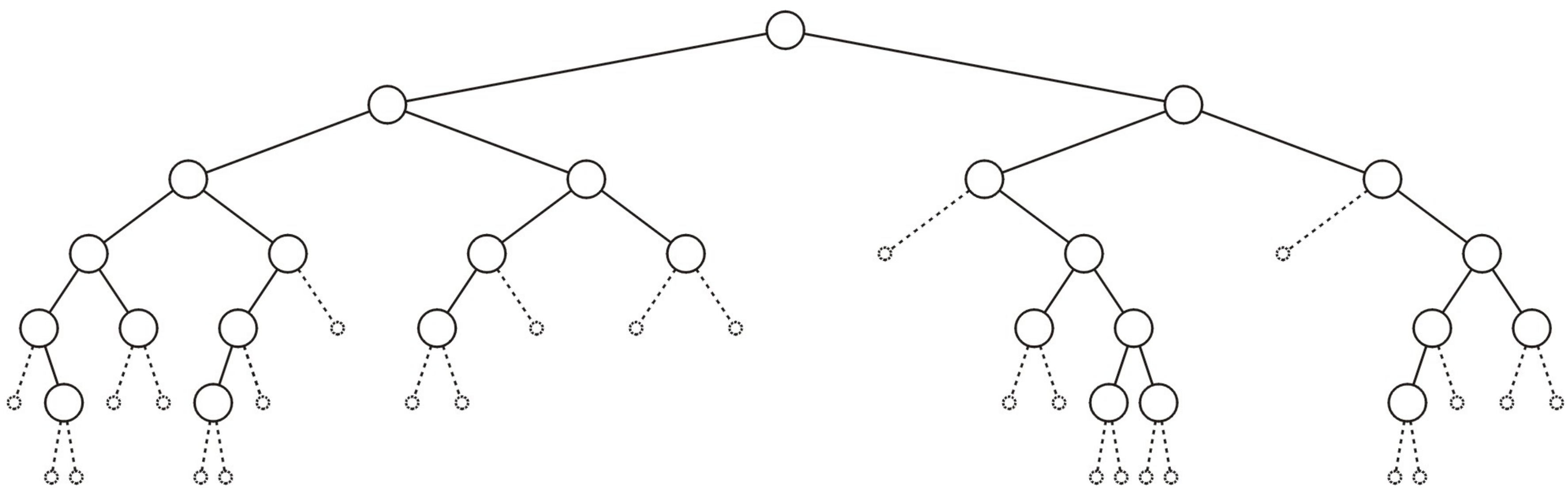


Legend:

full nodes neither leaf nodes

Definition

An *empty node* or a *null sub-tree* is any location where a new leaf node could be appended



Definition

Recursive definition:

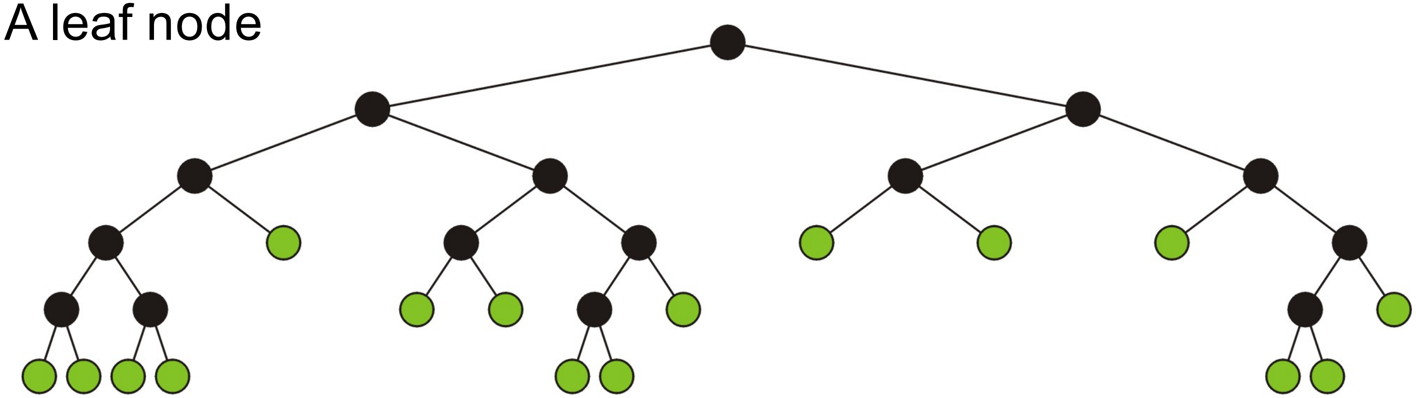
– A tree of height $h = 0$ (a leaf node) is a binary tree
– A tree with height $h > 0$ is a binary tree if it has *at most two subtrees* (children) which *are binary trees*

- To make the definition simpler to use we also add empty binary trees to the definition:
  - An ***empty tree*** is a binary tree.
    - It is empty (not even root!)
    - As the height of leaf node is 0, we define the height of empty trees to be -1.

Definition

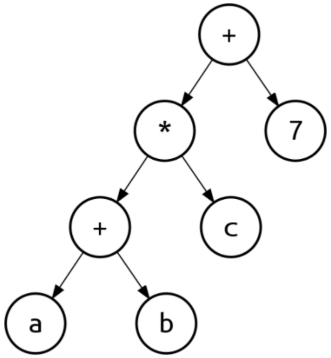
A *full binary tree* is where each node is:

- A full node, or
- A leaf node



It has applications in

- Expression trees
- Huffman encoding



Expression tree of the expression (a+b)*c+7

Implementation

Binary Node Class

We define a node class:

```
class BinaryNode<AnyType>
{
    private AnyType     element;
    private BinaryNode<AnyType> left;
    private BinaryNode<AnyType> right;



    public BinaryNode( AnyType theElement, BinaryNode<AnyType> lt,
      BinaryNode<AnyType> rt) {
        element = theElement;
        left    = lt;
        right   = rt;
    }
  // *** methods ***
   …
}
```

Binary Node Class

We define a node class:

Java Generic classes enable programmers to specify, with a single class declaration, a set of related types, respectively.

```
class BinaryNode<AnyType>
{
    private AnyType      element;
    private BinaryNode<AnyType> left;
    private BinaryNode<AnyType> right;



    public BinaryNode( AnyType theElement, BinaryNode<AnyType> lt,
      BinaryNode<AnyType> rt) {
        element = theElement;
        left    = lt;
        right   = rt;
    }
  // *** methods ***
    …
}
```

Binary Node Class

Accessor and Mutator methods for binary node class:

```
class BinaryNode<AnyType> {

  // access to fields
  public AnyType getElement( ) { return element; }
  public BinaryNode<AnyType> getLeft( ) { return left; }
  public BinaryNode<AnyType> getRight( ) { return right; }

  // change fields
  public void setElement( AnyType x ) { element = x; }
  public void setLeft( BinaryNode<AnyType> t ) { left = t; }
  public void setRight( BinaryNode<AnyType> t ) { right = t; }
}
```

Binary Tree Class

Binary tree class which use node class:

- two constructors

```
public class BinaryTree<AnyType> {
    // *** Fields ***
   private BinaryNode<AnyType> root;

   public BinaryTree() {
      root=null;
   }
   public BinaryTree( AnyType rootItem ){
      root = new BinaryNode<AnyType>( rootItem, null, null);
   }

    // *** methods ***
      …
}
```

creates an empty binary tree

creates a binary tree with
a single node (root)

Binary Tree Class

We can create larger binary trees using the recursive definition.

- We have two constructors to create empty tree and single node tree
- We need a method that takes two trees, merges them and creates a larger one
    - It should create a root node and assign the smaller trees as left and right subtrees of the root.

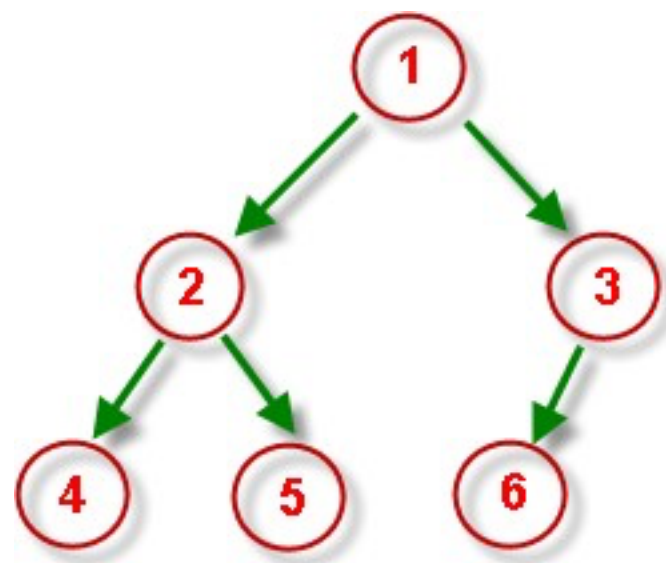
Binary Tree Class

The method merge

```
public void merge(AnyType rootItem, BinaryTree<AnyType> t1, BinaryTree<AnyType> t2){

     if( t1.root == t2.root && t1.root != null ) {
          System.err.println( "leftTree==rightTree; merge aborted" );
          return;
     }

      // Allocate new node as root and assigning t1 and t2 as left and
      // right subtrees
     root = new BinaryNode<AnyType>( rootItem, t1.root, t2.root );

      // Ensure that every node is in just one tree!
     if( this != t1 )
          t1.root = null;
     if( this != t2 )
          t2.root = null;
}
```

Exercise

Use the method merge (and contractors) to create a tree like this:



Exercise

```
BinaryTree<Integer> t4 = new BinaryTree<Integer>( 4 );
BinaryTree<Integer> t5 = new BinaryTree<Integer>( 5 );
BinaryTree<Integer> t6 = new BinaryTree<Integer>( 6 );
BinaryTree<Integer> t1 = new BinaryTree<Integer>( );
BinaryTree<Integer> t2 = new BinaryTree<Integer>( );
BinaryTree<Integer> t3 = new BinaryTree<Integer>( );
BinaryTree<Integer> temp = new BinaryTree<Integer>( );

t2.merge( 2, t4, t5 );
t3.merge( 3, t6, temp );
t1.merge( 1, t2, t3 );
```

Binary Tree Class

We can add more basic methods like:

```
public void clear() {
    root = null;
}

public boolean isEmpty() {
    return root == null;
}

public BinaryNode<AnyType> getRoot() {
    return root;
}
```

Size

- Returns the number of nodes in the tree.
- Uses a recursive helper which is defined in BinaryNode class
  - It recurs down the tree and counts the nodes.

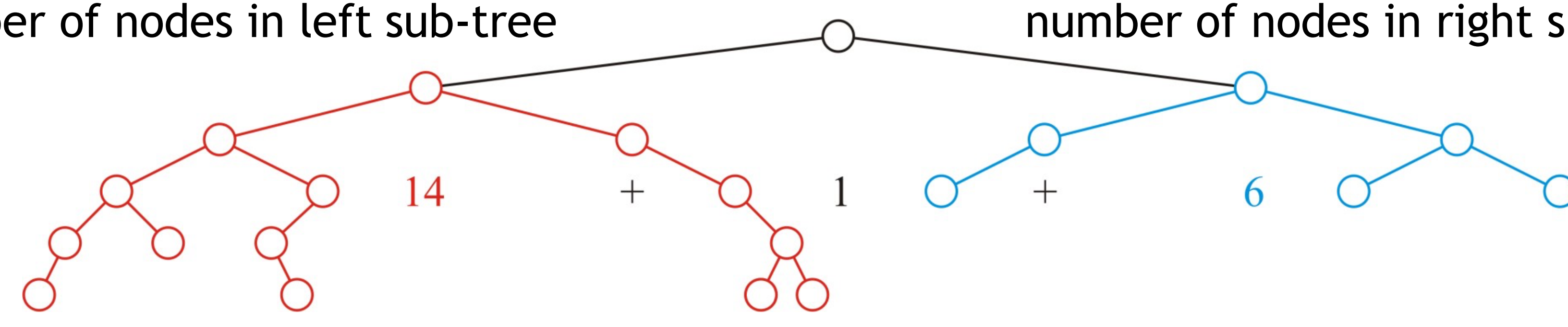
```
// A method defined in BinaryTree class
public int size() {
      return BinaryNode.size( root );
}
```

Size

number of nodes:

number of nodes in left sub-tree

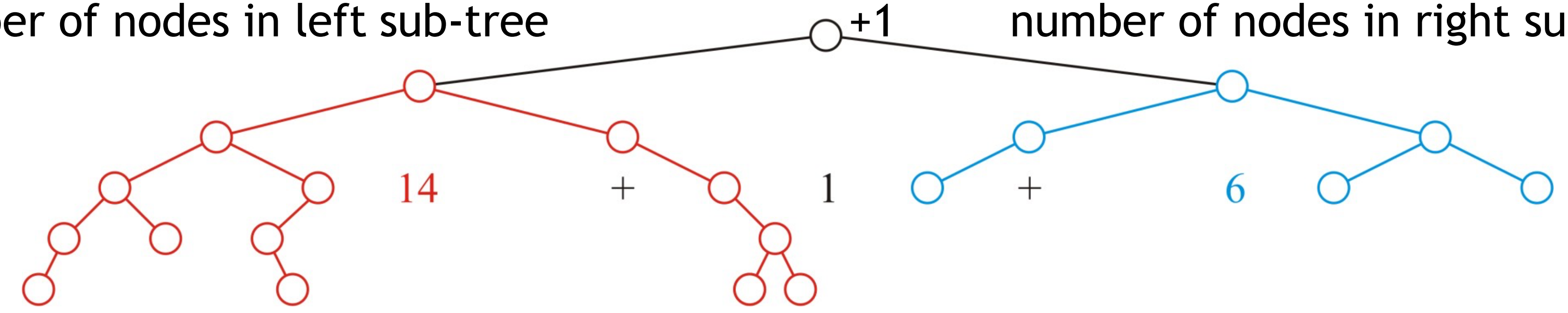
number of nodes in right sub-tree



Size

number of nodes:

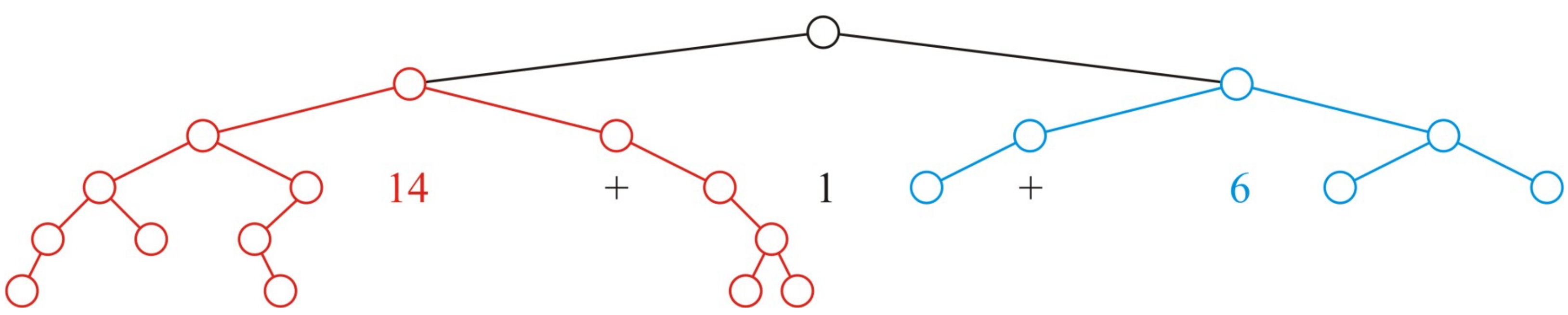
number of nodes in left sub-tree +1 number of nodes in right sub-tree



Size

The recursive size function runs in $\Theta(n)$ time and $\Theta(h)$ memory

```
// A method defined in BinaryNode class
public static <AnyType> int size(BinaryNode<AnyType> t){
    if (t == null)
        return 0;
    return 1 + size( t.left ) + size( t.right );
}
```



Height

- Returns the height of tree (maximum root to leaf depth)
- Uses a recursive helper which is defined in `BinaryNode` class
  - It recurs down the tree to find the max depth.

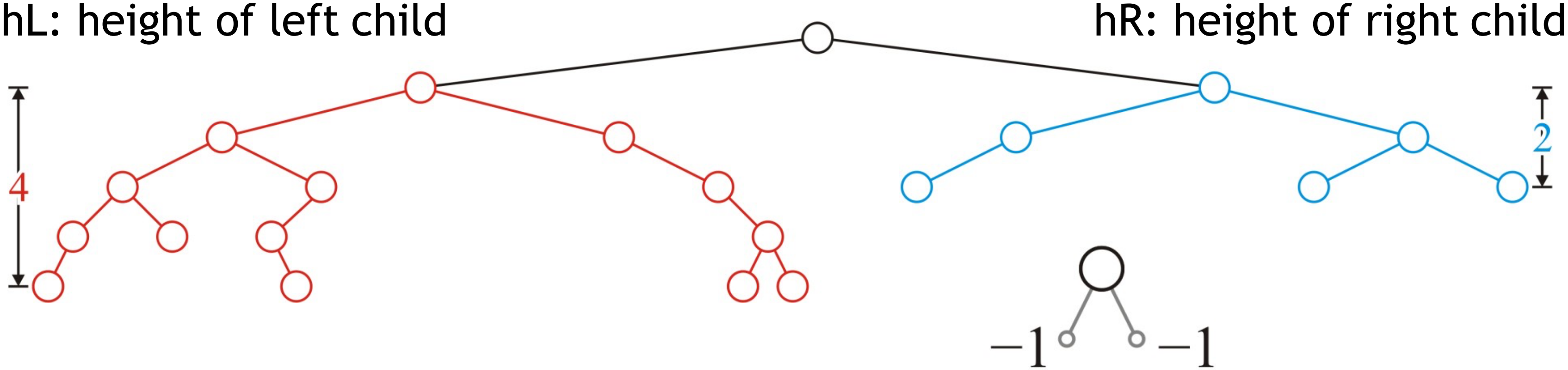
```
// A method defined in BinaryTree class
public int height() {
    return BinaryNode.height( root );
}
```

Height

hight = max(hL, hR) +1

hL: height of left child

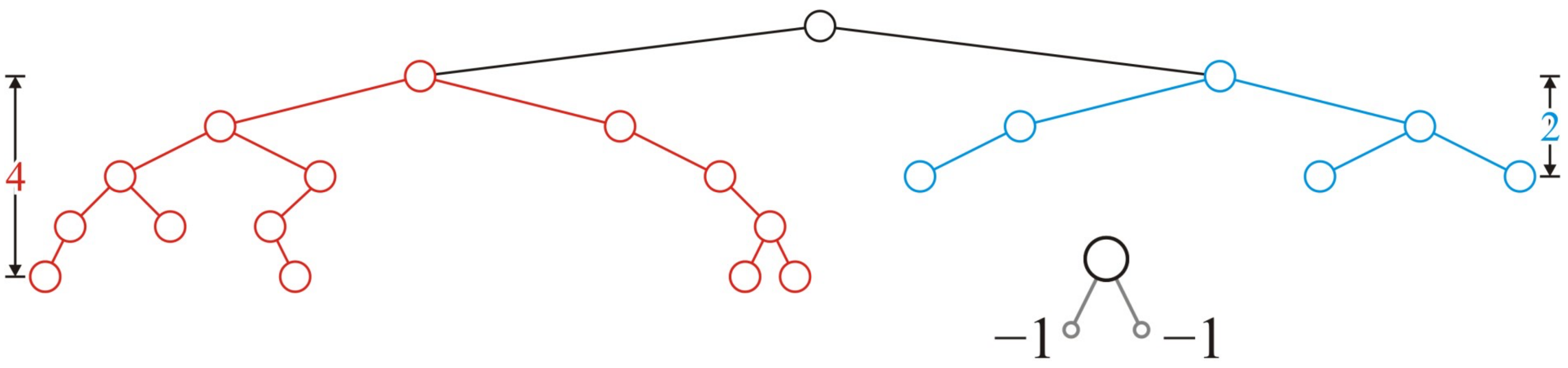
hR: height of right child



Height

- The recursive height function also runs in $\Theta(n)$ time and $\Theta(h)$ memory
  - Later we will implement this in Θ(1) time

```java
// A method defined in BinaryNode class
public static <AnyType> int height(BinaryNode<AnyType> t){
    if( t == null ) return -1;
    return 1 + Math.max(height(t.left),height(t.right));
}
```

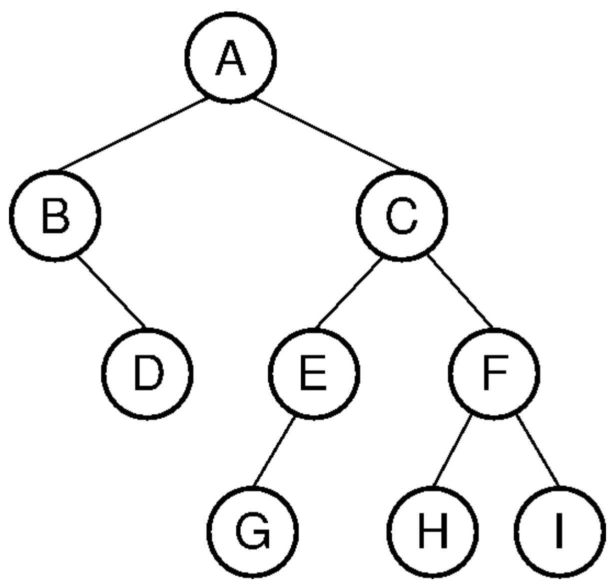


Binary Tree Traversals

- Like tree traversal, we have some traversal orders:
  - *preorder*: first visit the node, then left child, then right child
  - *postorder*: first visit left child, then right child, then the node itself
  - *in-order*: first visit left child, then the node itself, then right child

Binary Tree Traversals

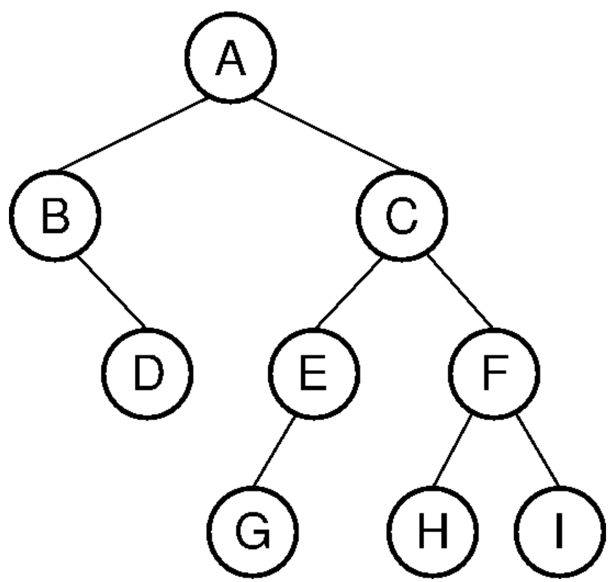
- *in-order*: first visit left child, then the node itself, then right child



Binary Tree Traversals

- *in-order*: first visit left child, then the node itself, then right child

B, D, A, G, E, C, H, F, I



Run Times

- Recall that with linked lists and arrays, some operations would run in $\Theta(n)$ time
- The run times of operations on binary trees, we will see, depends on the height of the tree $\Theta(h)$ which is:
  – Worst case?
  – Best case?
  – Average case?

Run Times

- Recall that with linked lists and arrays, some operations would run in $\Theta(n)$ time
- The run times of operations on binary trees, we will see, depends on the height of the tree $\Theta(h)$ which is:
  - Worst case? $\Theta(n)$
  - Best case? $\Theta(\log(n))$
  - Average case? $\Theta(\sqrt{n})$

Run Times

If we can achieve and maintain a height $\Theta(\log(n))$, we will see that many operations can run in $\Theta(\log(n))$.

Logarithmic time is not significantly worse than constant time:

$\log(1000) \approx 10$ kB

$\log(1\,000\,000) \approx 20$ MB

$\log(1\,000\,000\,000) \approx 30$ GB

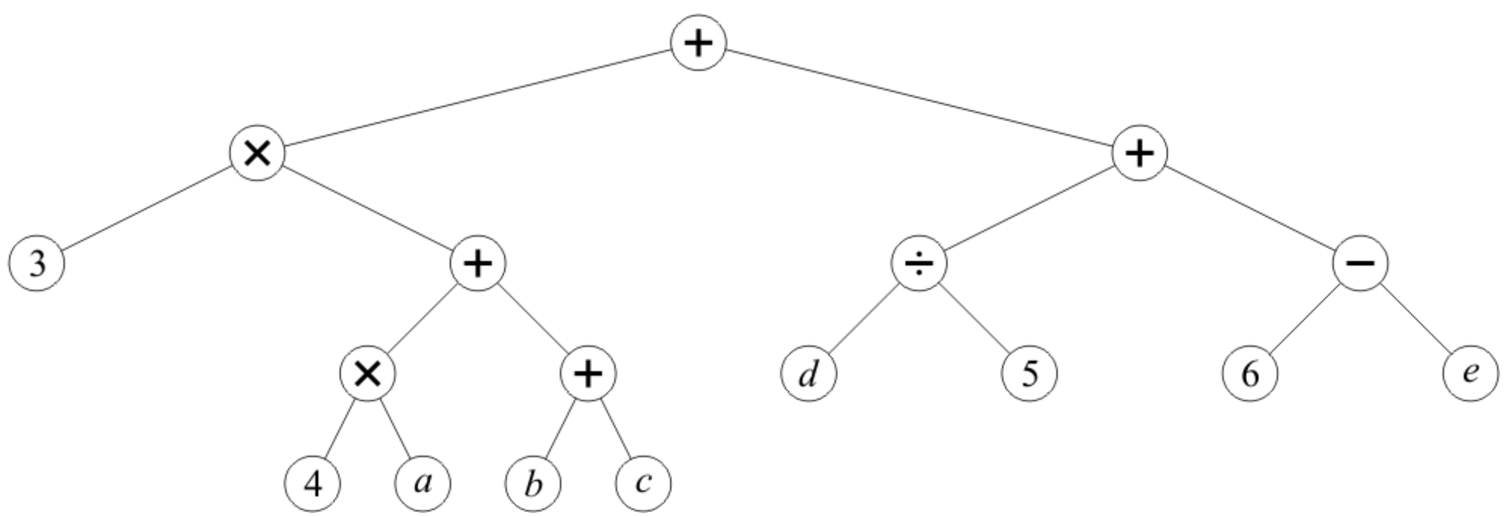
$\log(1\,000\,000\,000\,000) \approx 40$ TB

$\log(1000^n) \approx 10\,n$

Application: Expression Trees

Any basic mathematical expression containing binary operators may be represented using a binary tree

For example, $3(4a + b + c) + d/5 + (6 - e)$



Application: Expression Trees

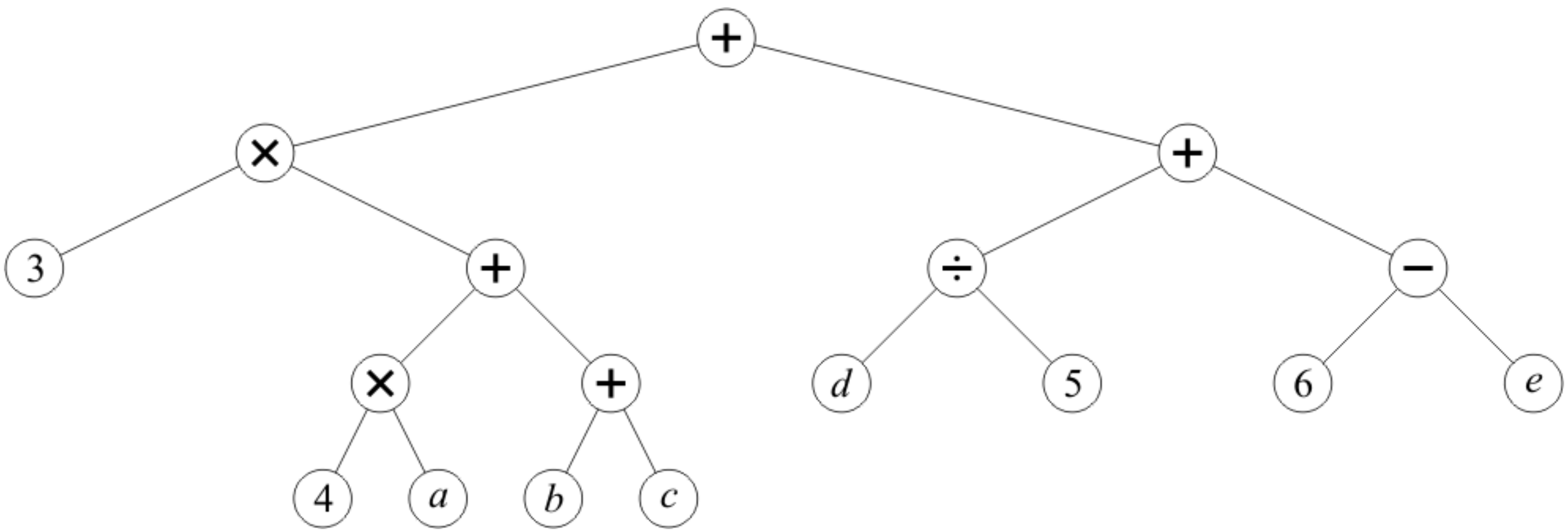
Observations:

- Internal nodes store operators
- Leaf nodes store literals or variables
- No nodes have just one sub tree
- The order is not relevant for
  - Addition and multiplication (commutative)
- Order is relevant for
  - Subtraction and division (non-commutative)
- It is possible to replace non-commutative operators using the unary negation and inversion:

$(a/b) = a\ b^{-1} \qquad (a - b) = a + (-b)$

Application: Expression Trees

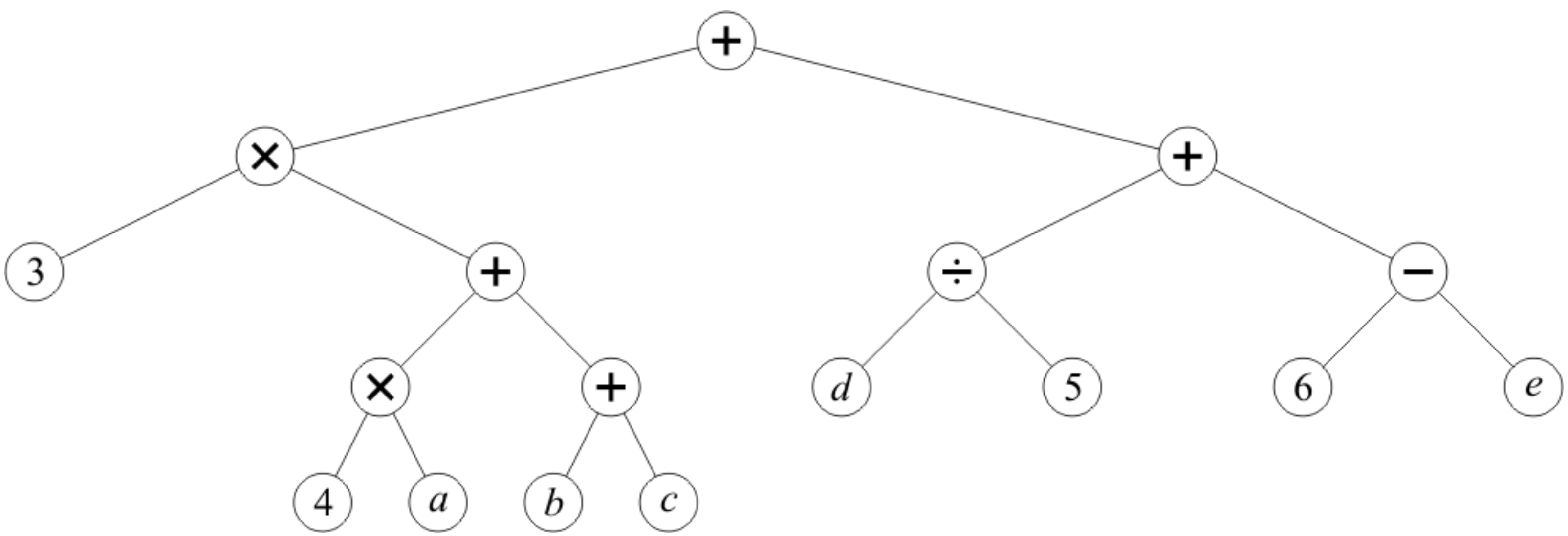
A post-order depth-first traversal converts such a tree to the reverse-Polish format



$$3\ 4\ a\ \times\ b\ c\ +\ +\ \times\ d\ 5\ \div\ 6\ e\ -\ +\ +$$

Application: Expression Trees

Write a program to compute the mathematical expressions, using expression trees!



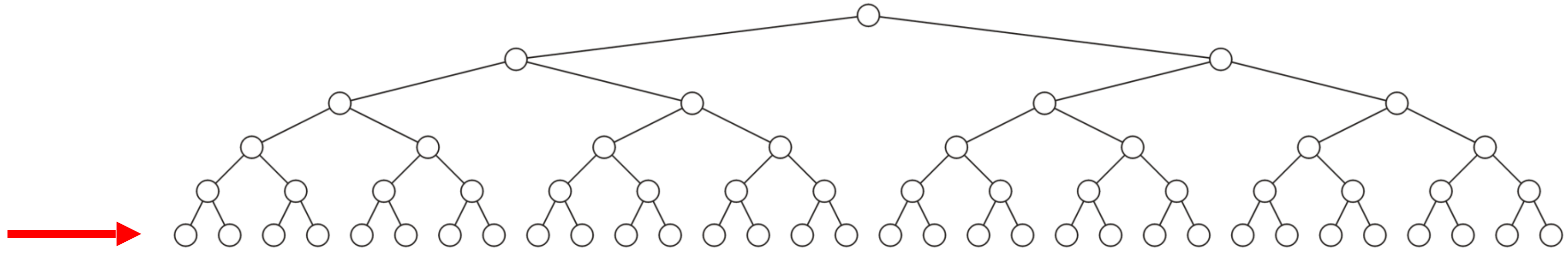
Perfect Binary Trees

Definition

Standard definition:

– A perfect binary tree of height h is a binary tree where

- All leaf nodes have the same depth h
- All other nodes are full



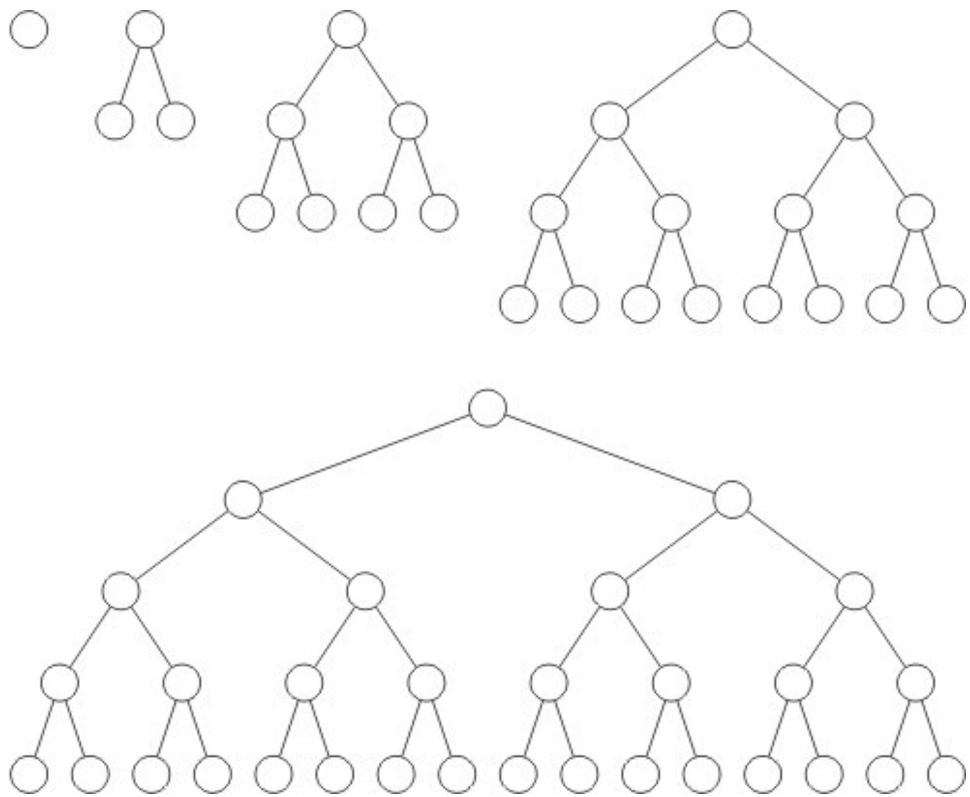
Definition

Recursive definition:

- A binary tree of height $h = 0$ (a single node) is perfect
- A binary tree with height $h > 0$ is a perfect binary tree if both sub-trees are prefect binary trees of height h-1

Examples

Perfect binary trees of height $h = 0, 1, 2, 3$ and 4



Theorems

We will now look at four theorems that describe the properties of perfect binary trees:

1. *A perfect binary tree has* $2^{h+1} - 1$ *nodes*
2. *The height is* $\Theta(\ln(n))$
3. *There are* 2^h *leaf nodes*
4. *The average depth of a node is* $\Theta(\ln(n))$

The results of these theorems will allow us to determine the optimal run-time properties of operations on binary trees

$2^{h+1} - 1$ Nodes

Theorem

A perfect binary tree of height h has $2^{h+1} - 1$ nodes

Proof:

We will use mathematical induction:

1. Show that it is true for $h = 0$
2. Assume it is true for an arbitrary h
   - Show that the truth for h implies the truth for $h + 1$

$2^{h+1} - 1$ Nodes

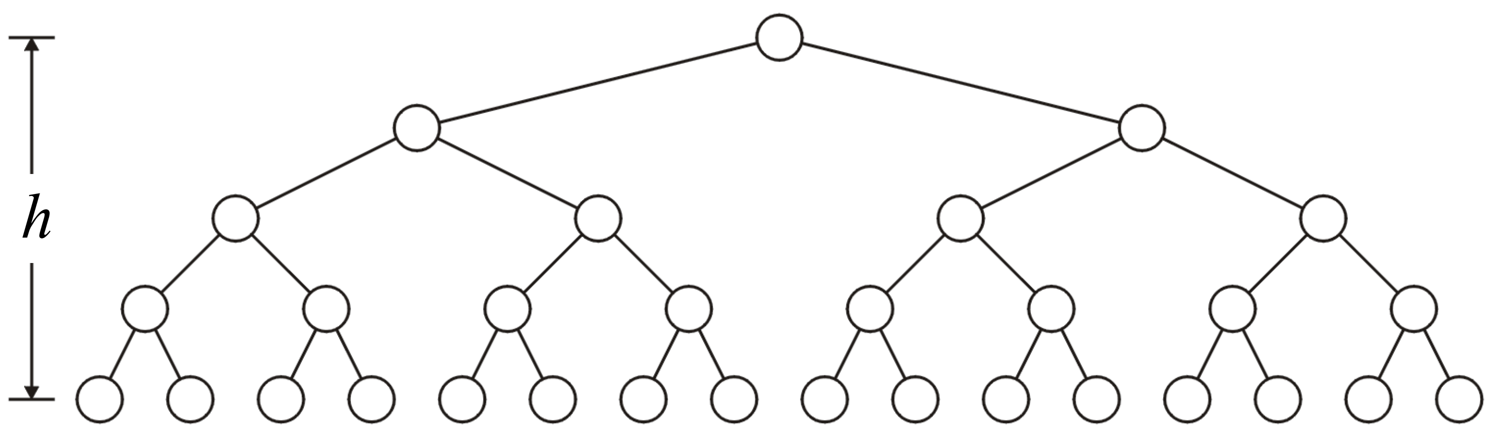
The base case:

- When $h = 0$ we have a single node $n = 1$
- The formula is correct: $2^{0+1} - 1 = 1$

$2^{h+1} - 1$ Nodes

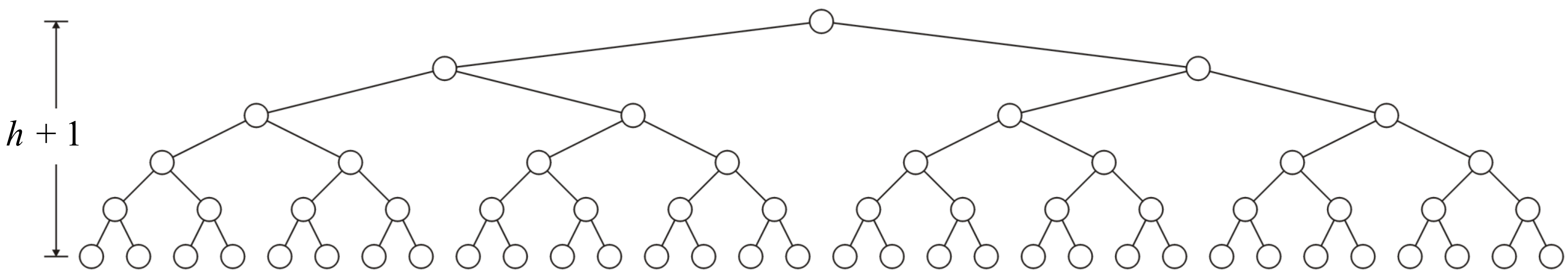
The inductive step:

- If the height of the tree is h, then we assume the theorem is correct then the number of nodes: $2^{h+1}-1$



$2^{h+1} - 1$ Nodes

We must show that a tree of height $h + 1$ has $n = 2^{(h+1)+1} - 1 = 2^{h+2} - 1$ nodes

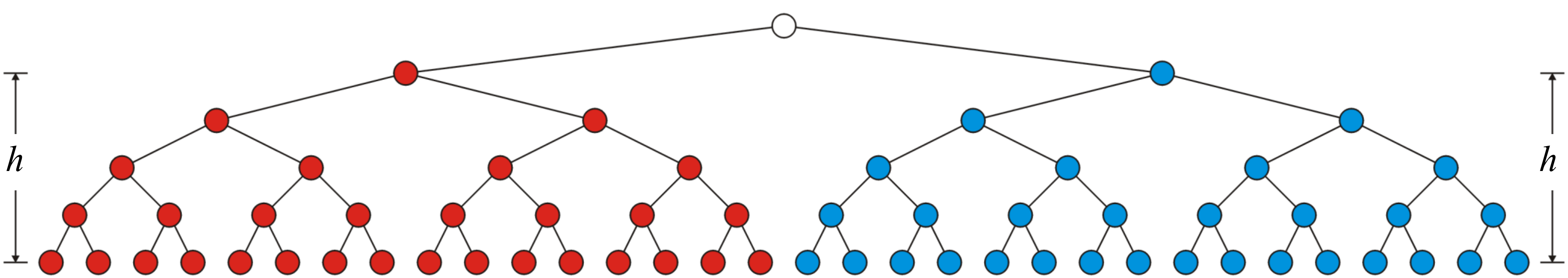


$2^{h+1} - 1$ Nodes

Using the recursive definition, both sub-trees are perfect trees of height h

- By assumption, each sub-tree has $2^{h+1} - 1$ nodes
- Therefore the total number of nodes is

$$(2^{h+1} - 1) + 1 + (2^{h+1} - 1) = 2^{h+2} - 1$$



$2^{h+1} - 1$ Nodes

Consequently

- The statement is true for $h = 0$ and the truth of the statement for an arbitrary h implies the truth of the statement for $h + 1$.

- Therefore, by the process of mathematical induction, the statement is true for all $h \geq 0$

Logarithmic Height

Theorem

A perfect binary tree with n nodes has height:

$\log(n + 1) - 1$

Proof

Solving $n = 2^{h+1} - 1$ for h:

$n + 1 = 2^{h+1}$

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

$h = \log(n + 1) - 1$

Logarithmic Height

Important

$$h = \log(n + 1) - 1 = \Theta(\log(n))$$

2^h Leaf Nodes

Theorem

A perfect binary tree with height h has 2^h leaf nodes

Proof (by induction):

When $h = 0$, there is $2^0 = 1$ leaf node.

Assume that a perfect binary tree of height h has 2^h leaf nodes and observe that both sub-trees of a perfect binary tree of height $h + 1$ have 2^{h+1} leaf nodes.

2^h Leaf Nodes

Theorem

A perfect binary tree with height h has 2^h leaf nodes

Proof (by induction):

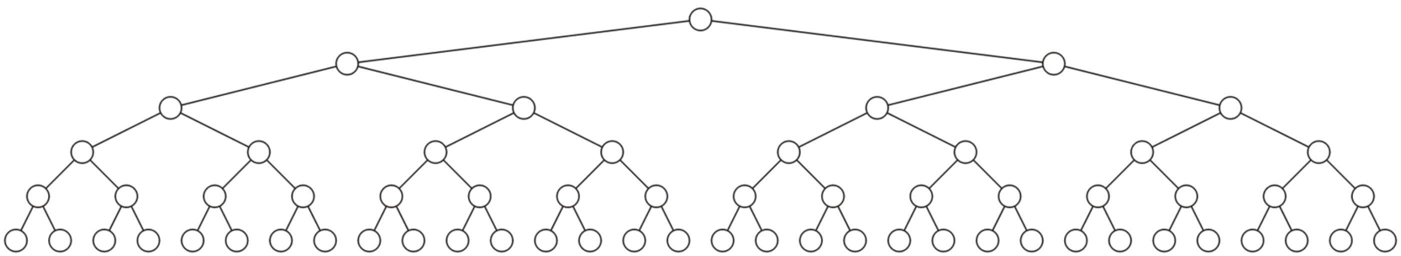
When $h = 0$, there is $2^0 = 1$ leaf node.

Assume that a perfect binary tree of height h has 2^h leaf nodes and observe that both sub-trees of a perfect binary tree of height $h + 1$ have 2^{h+1} leaf nodes.

Consequence: Over half all nodes are leaf nodes:

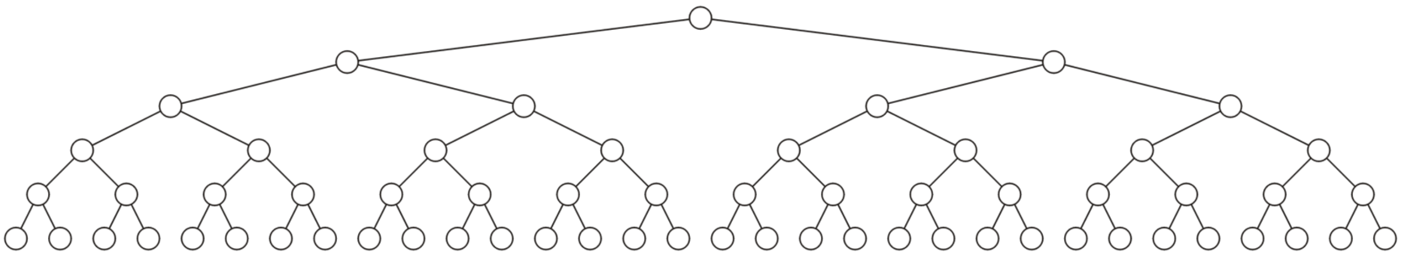
$$\frac{2^h}{2^{h+1}-1} > \frac{1}{2}$$

The Average Depth of a Node



Theorem: The average depth of a node in a perfect binary tree is: $\Theta(\ln(n))$

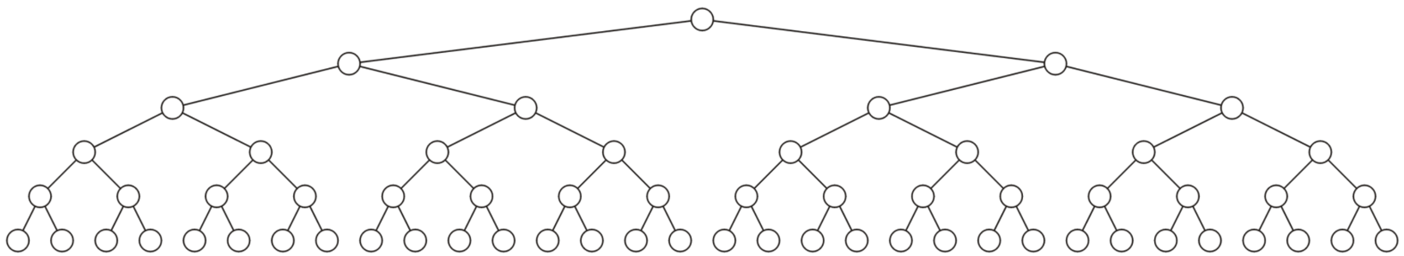
The Average Depth of a Node



Depth	Count
0	1
1	2
2	4
3	8
4	16
5	32

Theorem: The average depth of a node in a perfect binary tree is: $\Theta(\ln(n))$

The Average Depth of a Node



Depth	Count
0	1
1	2
2	4
3	8
4	16
5	32

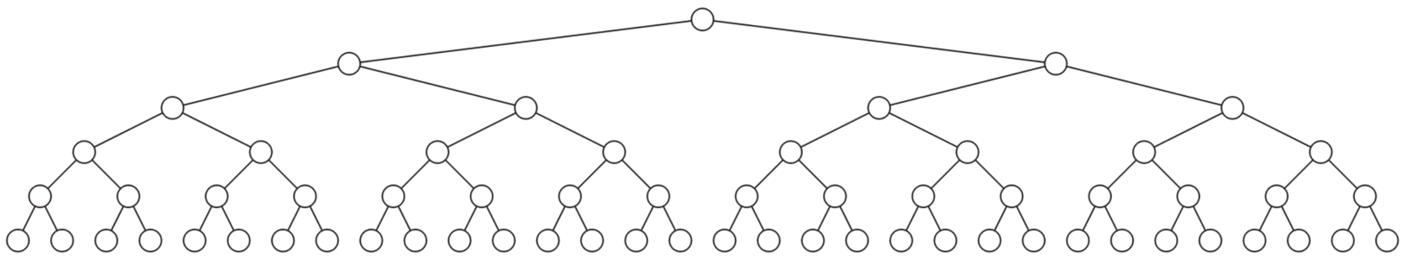
Theorem: The average depth of a node in a perfect binary tree is: $\Theta(\ln(n))$

Sum of the depths

$$\frac{\sum_{k=0}^{h} k2^k}{2^{h+1}-1}$$

Number of nodes

The Average Depth of a Node



Depth	Count
0	1
1	2
2	4
3	8
4	16
5	32

Theorem: The average depth of a node in a perfect binary tree is: $\Theta(\ln(n))$

Sum of the depths

Number of nodes

$$\frac{\sum_{k=0}^{h} k2^k}{2^{h+1}-1} = \frac{h2^{h+1} - 2^{h+1} + 2}{2^{h+1}-1} = \frac{h(2^{h+1}-1) - (2^{h+1}-1) + 1 + h}{2^{h+1}-1}$$

$$= h - 1 + \frac{h+1}{2^{h+1}-1} \approx h - 1 = \Theta\left(\ln(n)\right)$$

Applications

Perfect binary trees are considered to be the *ideal* case

- The height and average depth are both $\Theta(\ln(n))$

Recall that, the run times of operations on binary trees depends on the height of the tree $\Theta(h)$

- In the worst case $\Theta(n)$

We will attempt to find trees which are as close as possible to perfect binary trees

Complete Binary Trees

A perfect binary tree has ideal properties but restricted in the number of nodes:

$n = 2^h - 1$

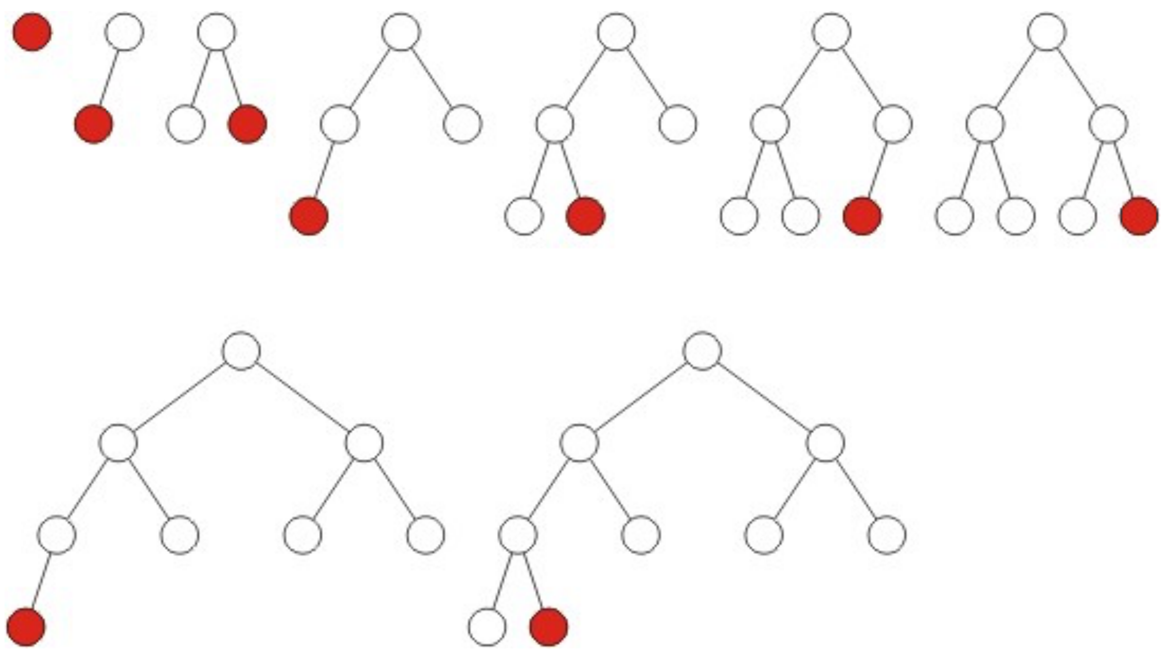
1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, ...

We require binary trees which are

- –Similar to perfect binary trees, but
- –Defined for all n

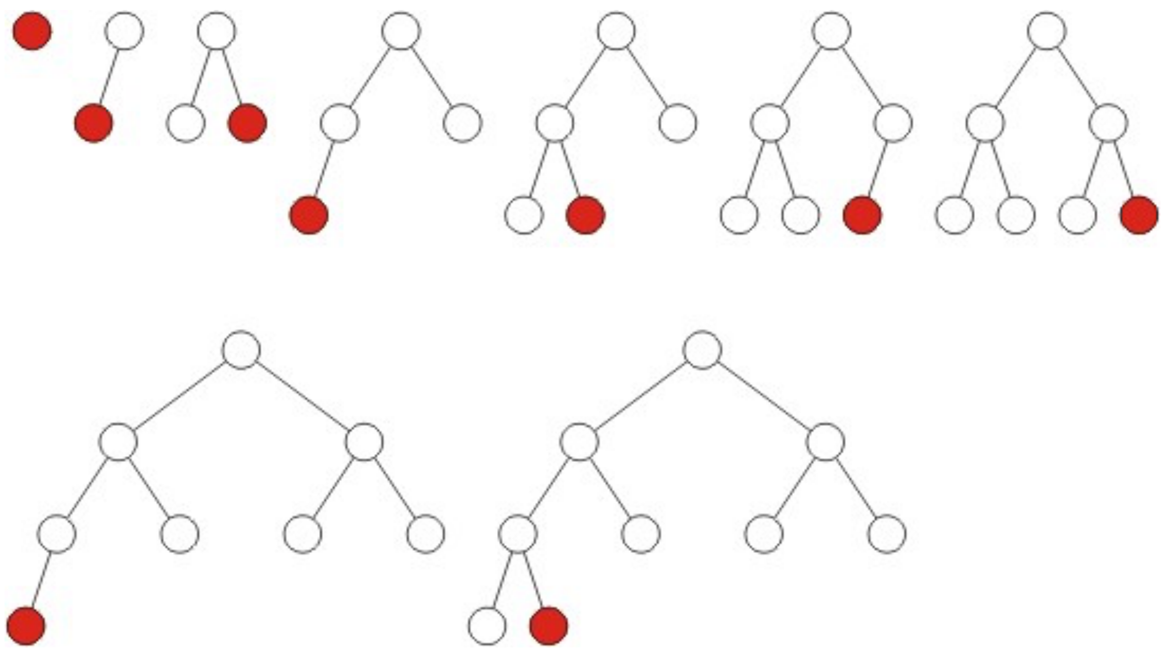
Complete Binary Trees

Definition: A complete binary tree filled at each depth from left to right:



Complete Binary Trees

Like a perfect binary tree which is missing some leaf nodes (from right side)!



Complete Binary Trees

Theorem

The height of a complete binary tree with n nodes is

$$h = \lfloor \log(n) \rfloor$$

Proof:

- Using mathematical induction
- In extra slides

Complete Binary Trees

Consequence:

- In Complete binary trees, the height and average depth are both $\Theta(\log(n))$

Extra Slides

Complete Binary Trees

Background

A perfect binary tree has ideal properties but restricted in the number of nodes: $n = 2^h - 1$

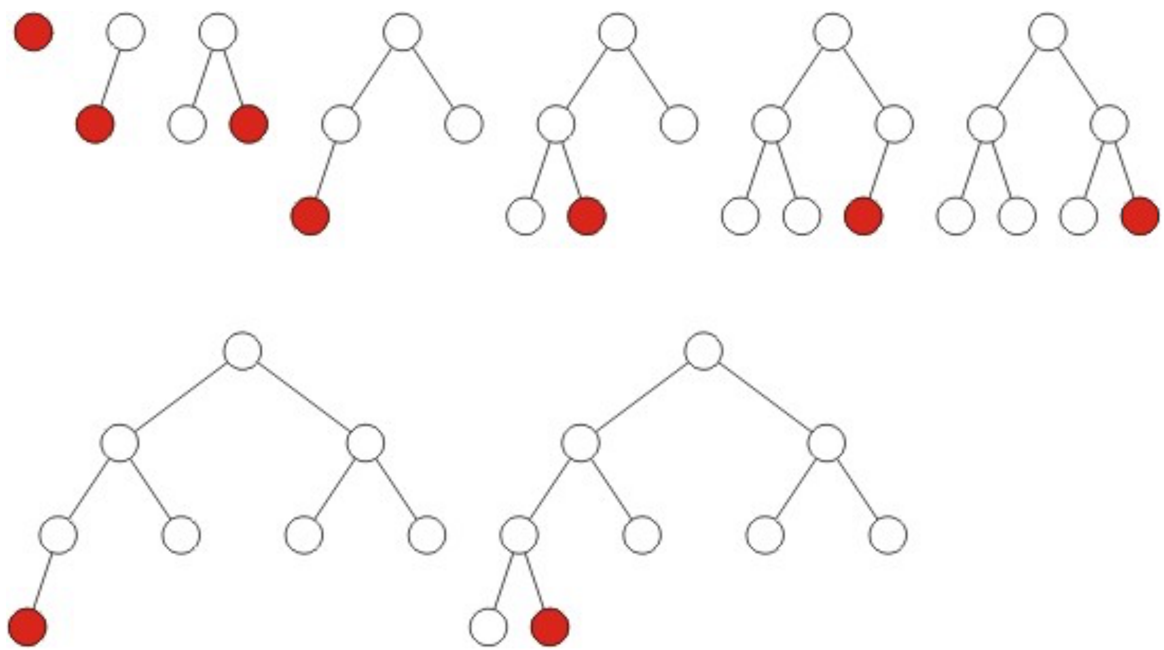
1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, ...

We require binary trees which are

- Similar to perfect binary trees, but
- Defined for all n

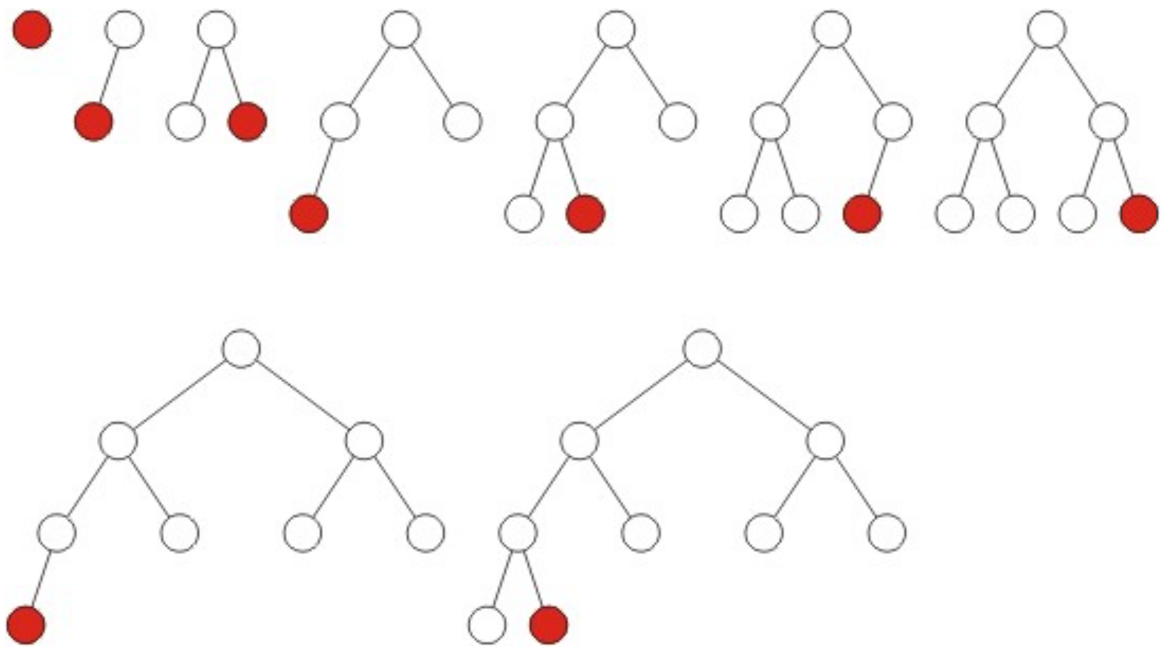
Definition

A complete binary tree filled at each depth from left to right:



Definition

The order is identical to that of a breadth-first traversal



Height

Theorem

The height of a complete binary tree with n nodes is h

$= \lfloor \lg(n) \rfloor$

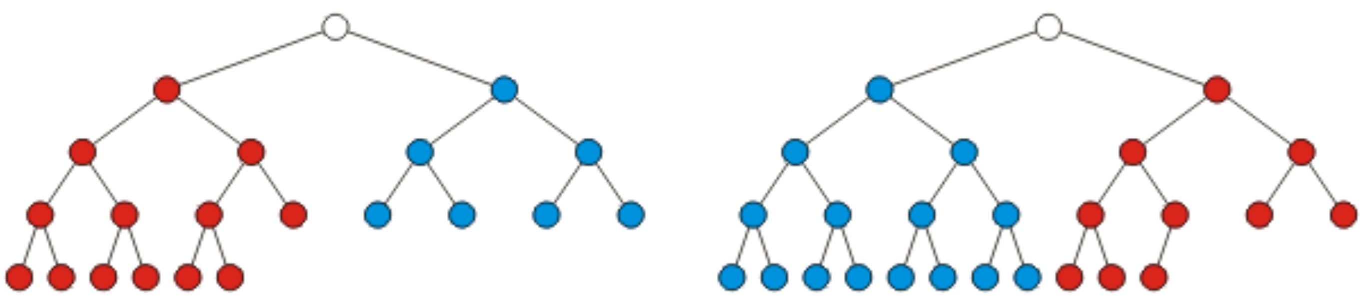
Proof:

- Using mathematical induction
- In extra slides

Recursive Definition

Recursive definition: a binary tree with a single node is a complete binary tree of height $h = 0$ and a complete binary tree of height h is a tree where either:

- –The left sub-tree is a **complete tree** of height $h-1$ and the right sub-tree is a **perfect tree** of height $h-2$, or
- –The left sub-tree is **perfect tree** with height $h-1$ and the right sub-tree is **complete tree** with height $h-1$



Height

Theorem

The height of a complete binary tree with n nodes is $h = \lfloor \lg(n) \rfloor$

Proof:

–Base case:

- When $n = 1$ then $\lfloor \lg(1) \rfloor = 0$ and a tree with one node is a complete tree with height $h = 0$

–Inductive step:

- Assume that a complete tree with n nodes has height $\lfloor \lg(n) \rfloor$
- Must show that $\lfloor \lg(n + 1) \rfloor$ gives the height of a complete tree with $n + 1$ nodes
- Two cases:
  - –If the tree with n nodes is perfect, and
  - –If the tree with n nodes is complete but not perfect

Height

Case 1 (the tree is perfect):

– If it is a perfect tree then

  - Adding one more node must increase the height

– Before the insertion, it had $n = 2^{h+1} - 1$ nodes:

$$2^h < 2^{h+1} - 1 < 2^{h+1}$$

$$h = \lg\left(2^h\right) < \lg\left(2^{h+1} - 1\right) < \lg\left(2^{h+1}\right) = h + 1$$

$$h \le \left\lfloor \lg\left(2^{h+1} - 1\right) \right\rfloor < h + 1$$

– Thus, $\lfloor \lg(n) \rfloor = h$

– However, $\lfloor \lg(n+1) \rfloor = \left\lfloor \lg\left(2^{h+1} - 1 + 1\right) \right\rfloor = \left\lfloor \lg\left(2^{h+1}\right) \right\rfloor = h + 1$

Height

Case 2 (the tree is complete but not perfect):

–If it is not a perfect tree then

$$2^h \le n < 2^{h+1} - 1$$

$$2^h + 1 \le n + 1 < 2^{h+1}$$

$$h < \lg\left(2^h + 1\right) \le \lg\left(n + 1\right) < \lg\left(2^{h+1}\right) = h + 1$$

$$h \le \left\lfloor \lg\left(2^h + 1\right) \right\rfloor \le \left\lfloor \lg\left(n + 1\right) \right\rfloor < h + 1$$

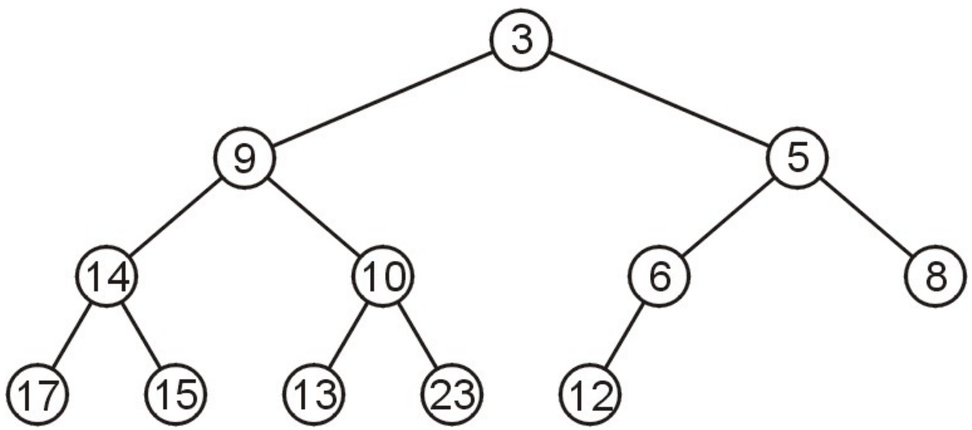
–Consequently, the height is unchanged: $\lfloor \lg(n + 1) \rfloor = h$

By mathematical induction, the statement must be true for all $n \ge 1$

Array storage

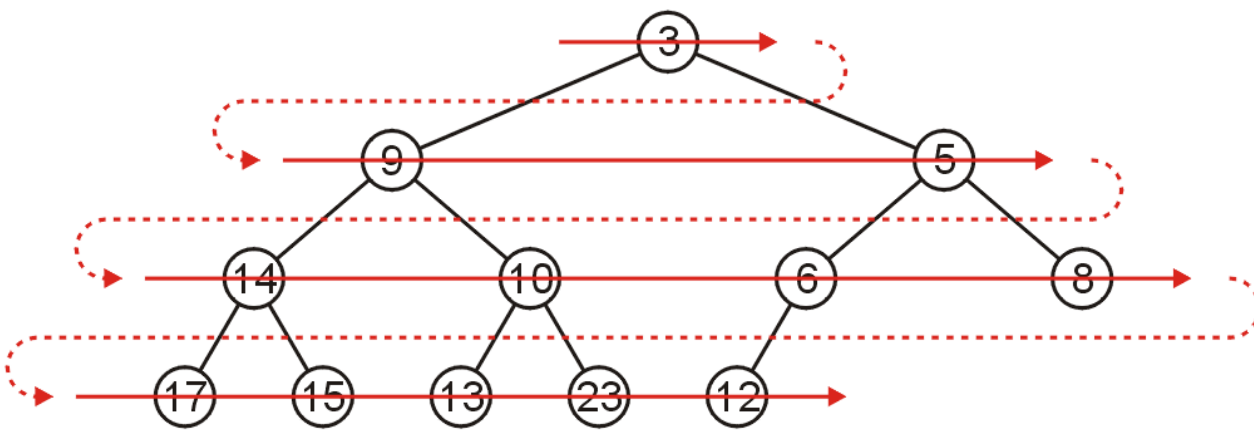
We are able to store a complete tree as an array

– Traverse the tree in breadth-first order, placing the entries into the array



Array storage

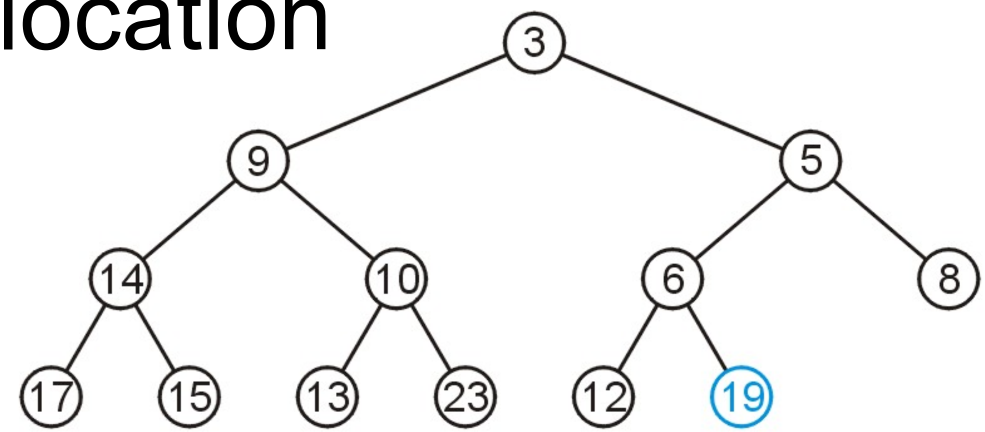
We can store this in an array after a quick traversal:



	3	9	5	14	10	6	8	17	15	13	23	12				

Array storage

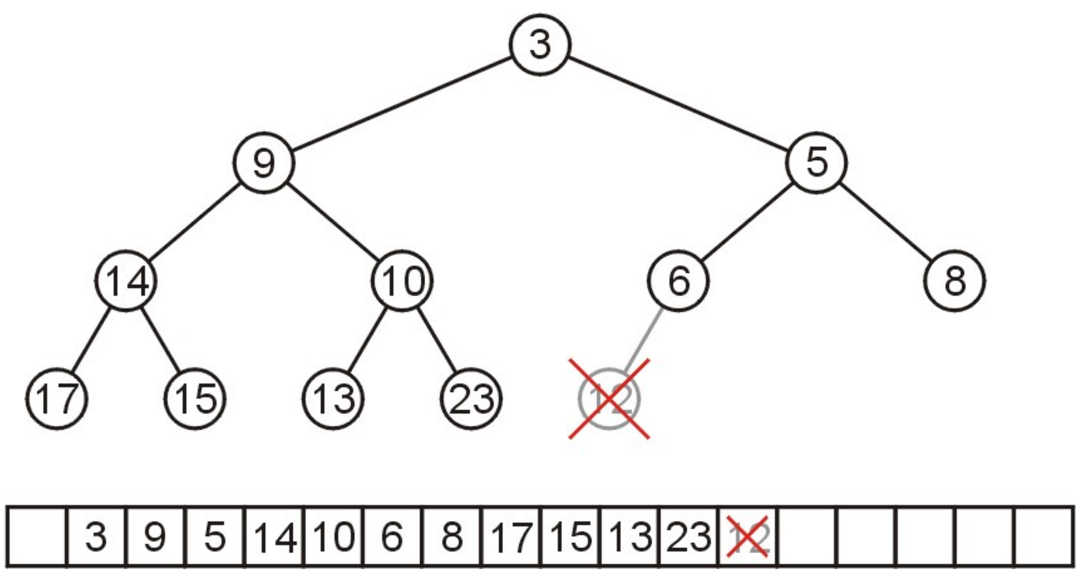
To insert another node while maintaining the complete-binary-tree structure, we must insert into the next array location



	3	9	5	14	10	6	8	17	15	13	23	12	19				

Array storage

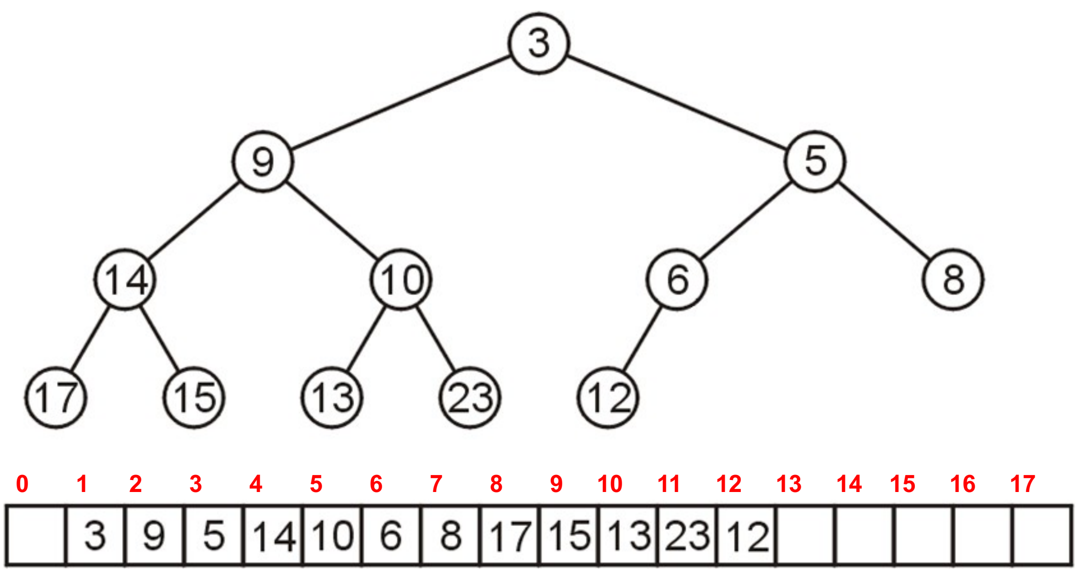
To remove a node while keeping the complete-tree structure, we must remove the last element in the array



Array storage

Leaving the first entry blank yields a bonus:

- The children of the node with index k are in $2k$ and $2k + 1$
- The parent of node with index k is in $k \div 2$

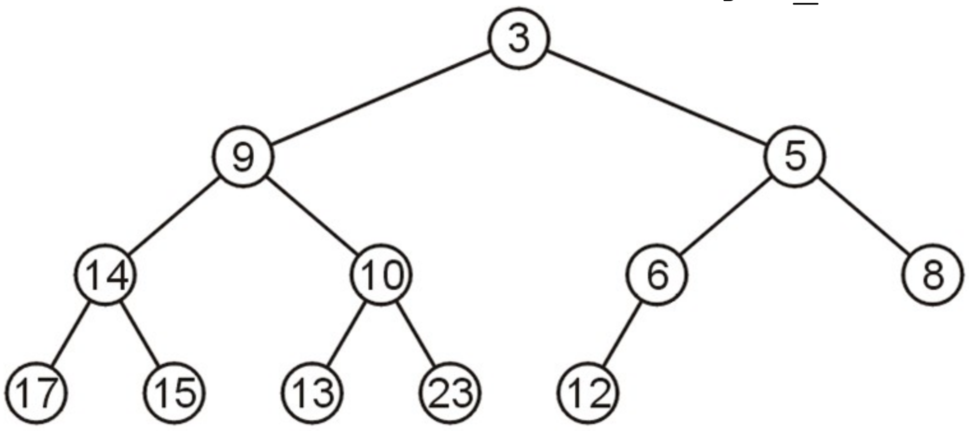


Array storage

Leaving the first entry blank yields a bonus:

– In C++, this simplifies the calculations:

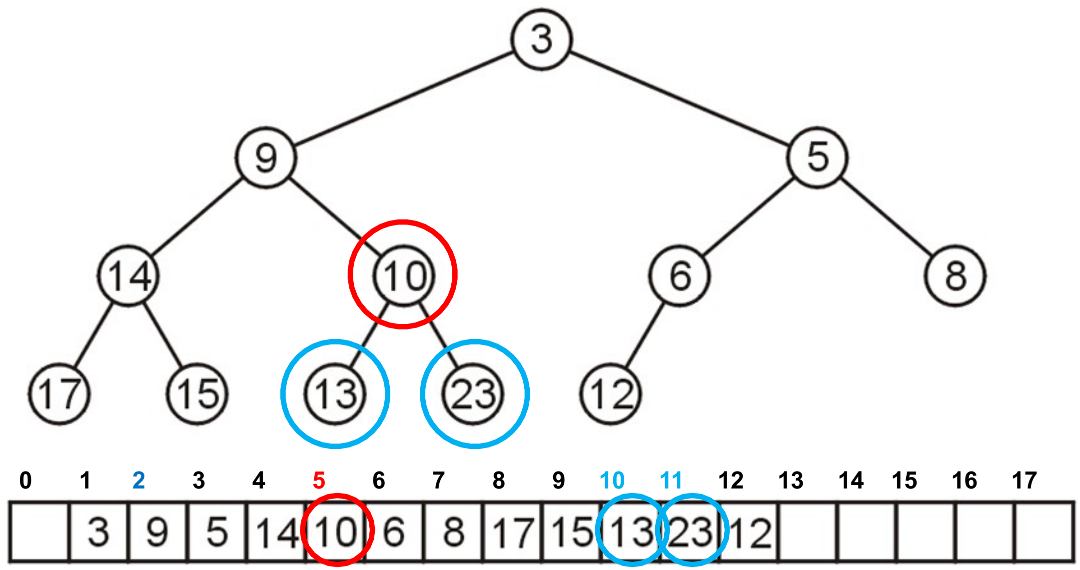
```
parent = k >> 1;
left_child = k << 1;
right_child = left_child | 1;
```



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	3	9	5	14	10	6	8	17	15	13	23	12					

Array storage

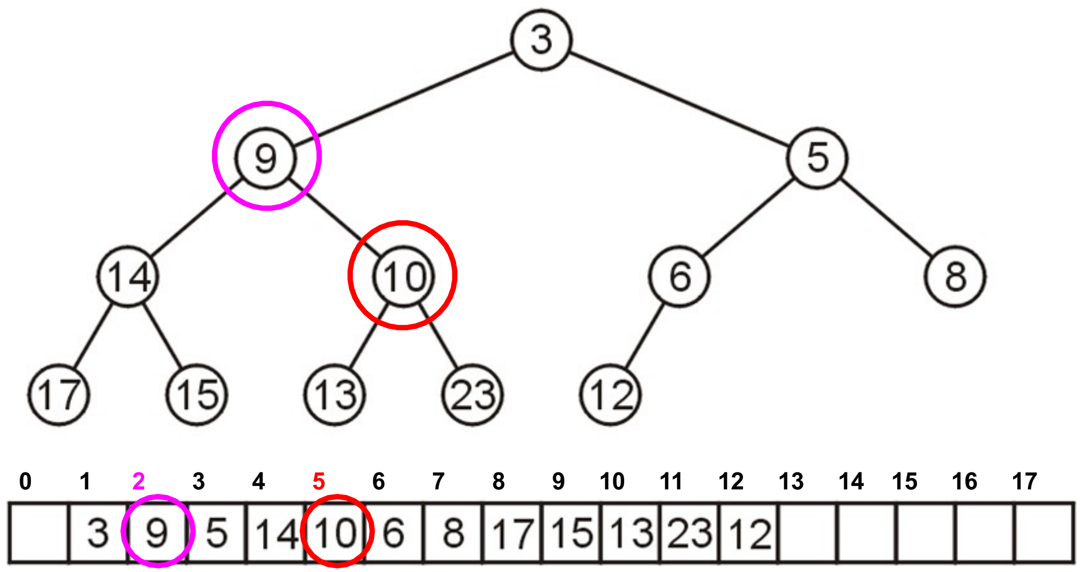
- For example, node 10 has index 5:
  - Its children 13 and 23 have indices 10 and 11, respectively



Array storage

For example, node 10 has index 5:

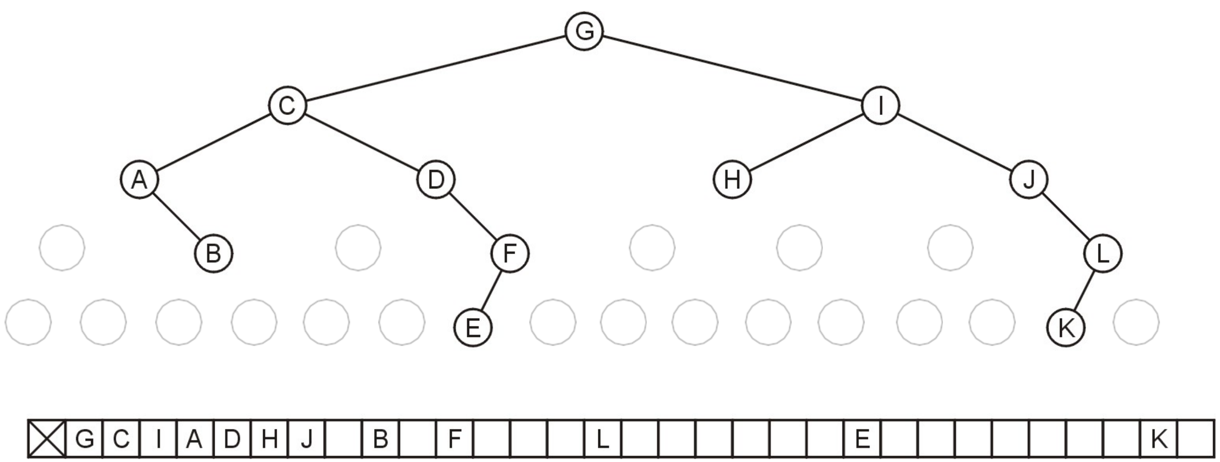
- Its children 13 and 23 have indices 10 and 11, respectively
- Its parent is node 9 with index 5/2 = **2**



Array storage

Question: why not store any tree as an array using breadth-first traversals?

– There is a significant potential for a lot of wasted memory



Consider this tree with 12 nodes would require an array of size 32

– Adding a child to node K doubles the required memory

Array storage

In the worst case, an exponential amount of memory is required

These nodes would be stored in entries 1, 3, 6, 13, 26, 52, 105