CSE 203: Binary Tree

Dr. Mohammed Eunus Ali

Professor

CSE, BUET

Slides are from Douglas Wilhelm Harder, dwharder@alumni.uwaterloo.ca

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
- Phylogenetic trees
- Lossless encoding algorithms

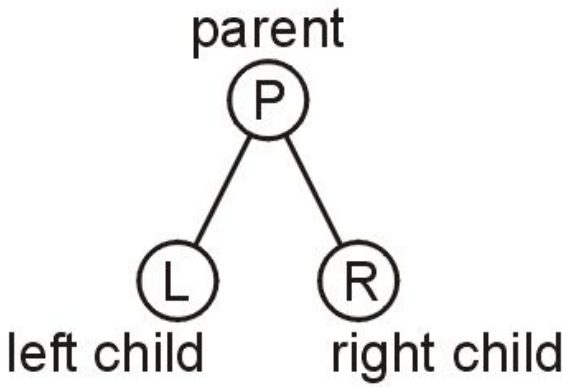
There are also issues with general trees:

- There is no natural order between a node and its children

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

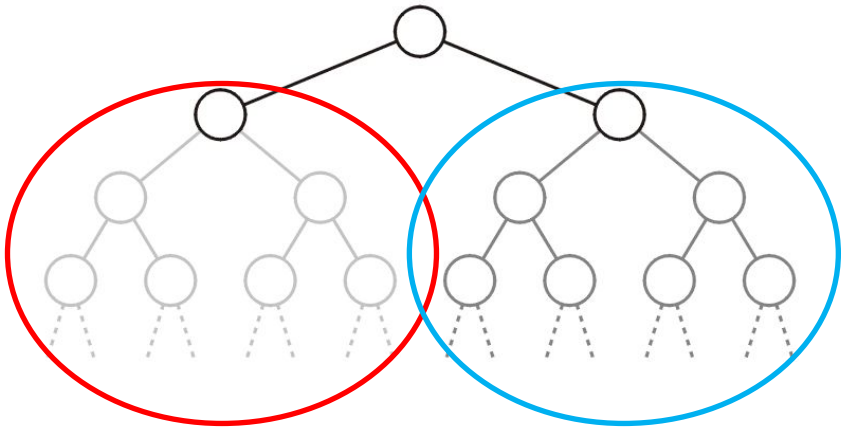


At this point, recall that $\lg(n) = \Theta(\log_b(n))$ for any b

Definition

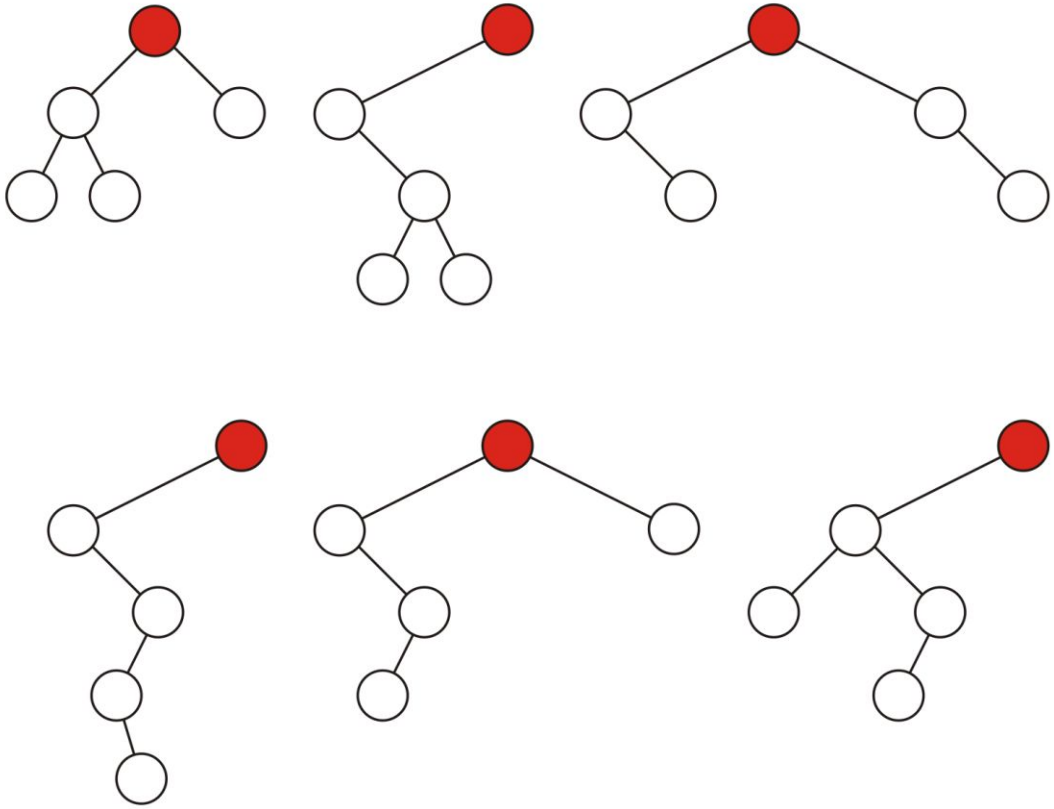
We will also refer to the two sub-trees as

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



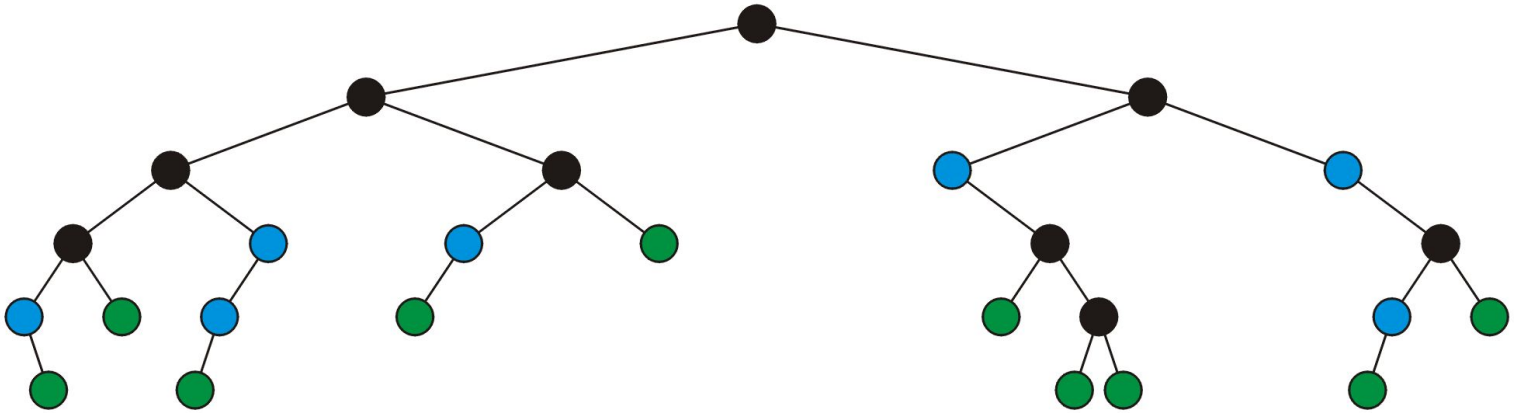
Definition

Sample variations on binary trees with five nodes:



Definition

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

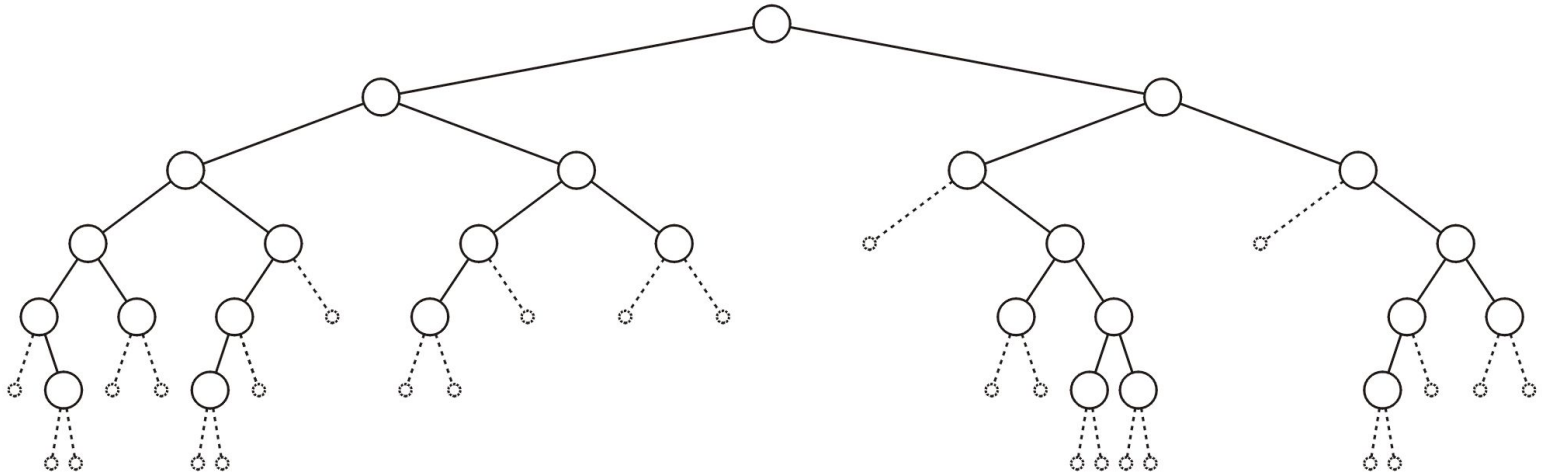


Legend:

full nodes neither leaf nodes

Definition

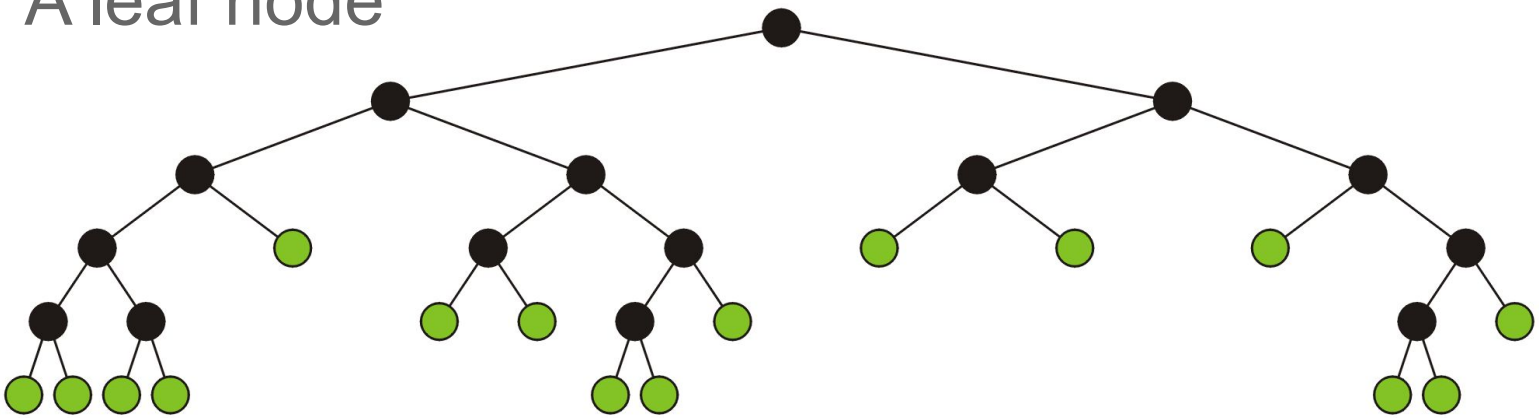
An *empty node* or a *null sub-tree* is any location where a new leaf node could be appended



Definition

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

- A full node, or
- A leaf node



These have applications in

- Expression trees
- Huffman encoding

Binary Node Class

The binary node class is similar to the single node class:

```
#include <algorithm>

template <typename Type>
class Binary_node {
    protected:
        Type node_value;
        Binary_node *p_left_tree;
        Binary_node *p_right_tree;

    public:
        Binary_node( Type const & );

        Type value() const;
        Binary_node *left() const;
        Binary_node *right() const;

        bool is_leaf() const;
        int size() const;
        int height() const;
        void clear();
}
```

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

We will see that:

- The worst is clearly $\Theta(n)$
- Under average conditions, the height is $\Theta\left(\sqrt{n}\right)$
- The best case is $\Theta(\ln(n))$

Run Times

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

Logarithmic time is not significantly worse than constant time:

$$\lg(\ 1000\ ) \approx 10 \qquad \text{kB}$$

$$\lg(\ 1\ 000\ 000\ ) \approx 20 \qquad \text{MB}$$

$$\lg(\ 1\ 000\ 000\ 000\ ) \approx 30 \qquad \text{GB}$$

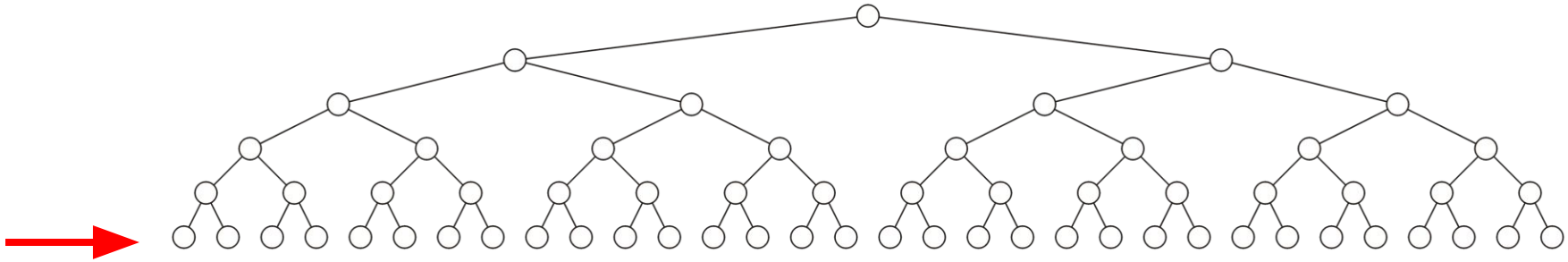
$$\lg(\ 1\ 000\ 000\ 000\ 000\ ) \approx 40 \qquad \text{TB}$$

$$\lg(\ 1000^{n}\ ) \approx 10\ n$$

Perfect Binary Tree

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



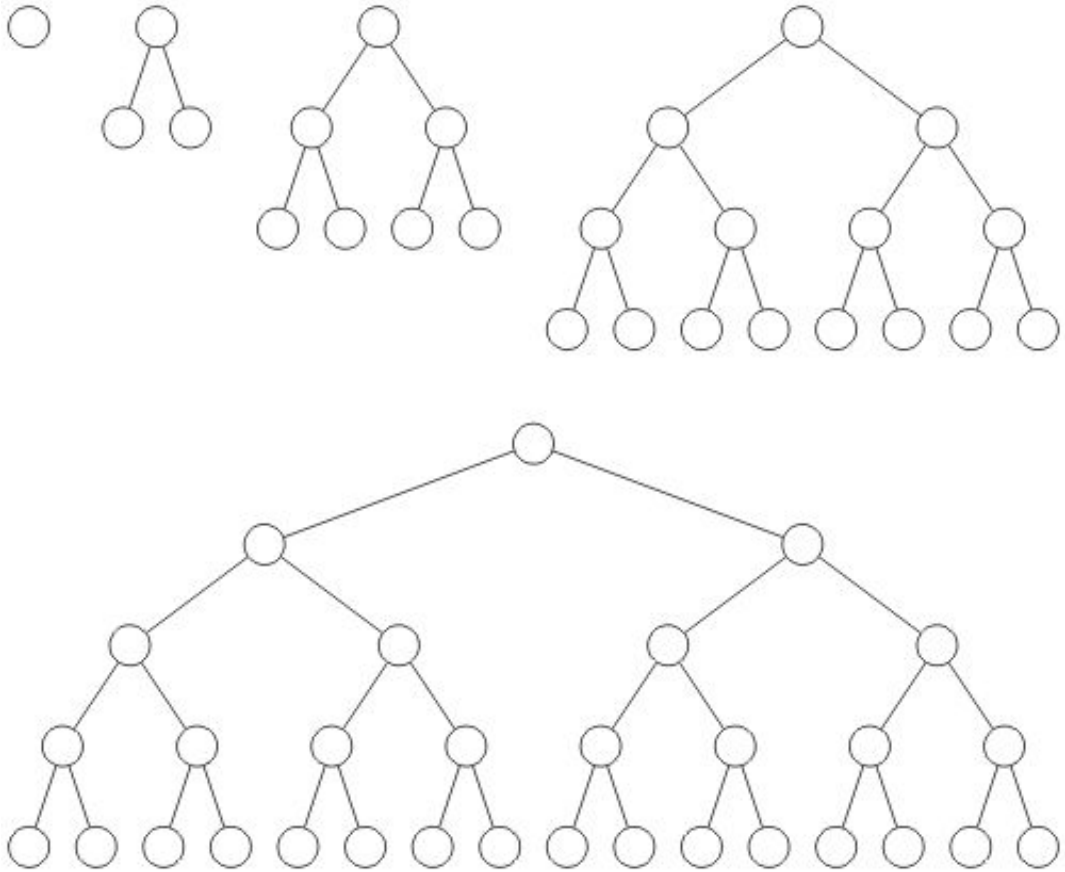
Definition

Recursive definition:

- A binary tree of height $h = 0$ is perfect
- A binary tree with height $h > 0$ is a perfect 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:

- A perfect tree has $2^{h+1} - 1$ nodes
- The height is $\Theta(\ln(n))$
- There are 2^h leaf nodes
- 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
3. Show that the truth for h implies the truth for $h + 1$

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

The base case:

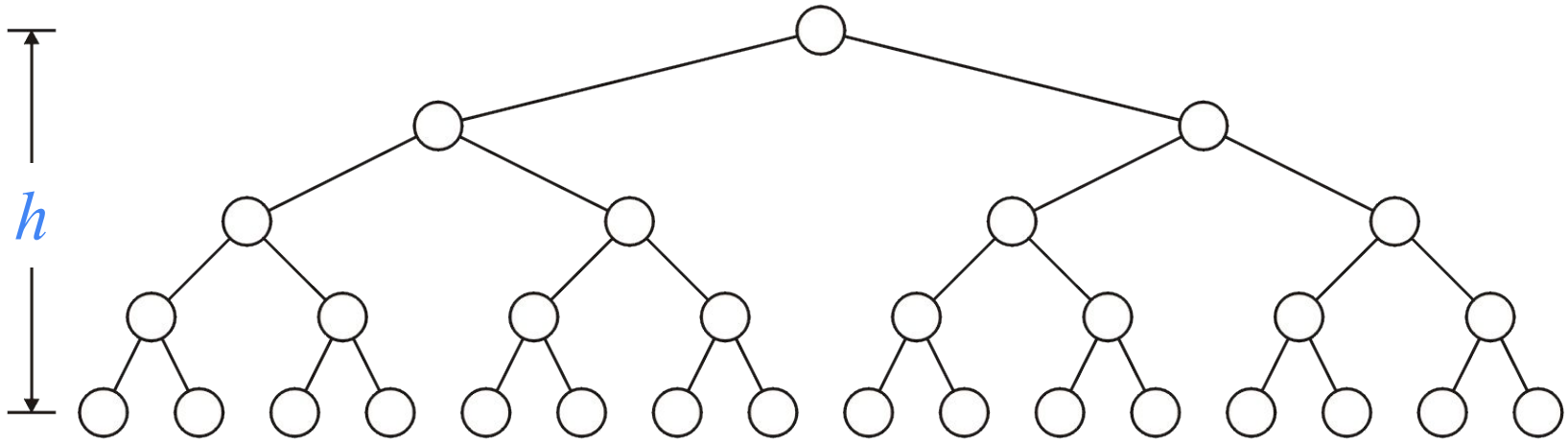
- When $h = 0$ we have a single node $n = 1$
- The formula is correct: $2^{0+1} - 1 = 1$

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

The inductive step:

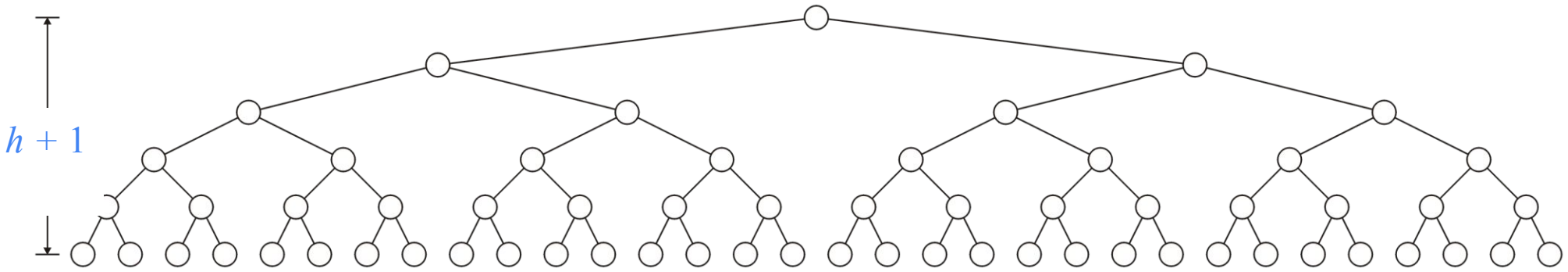
- If the height of the tree is h, then assume that the number of nodes is

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



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

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

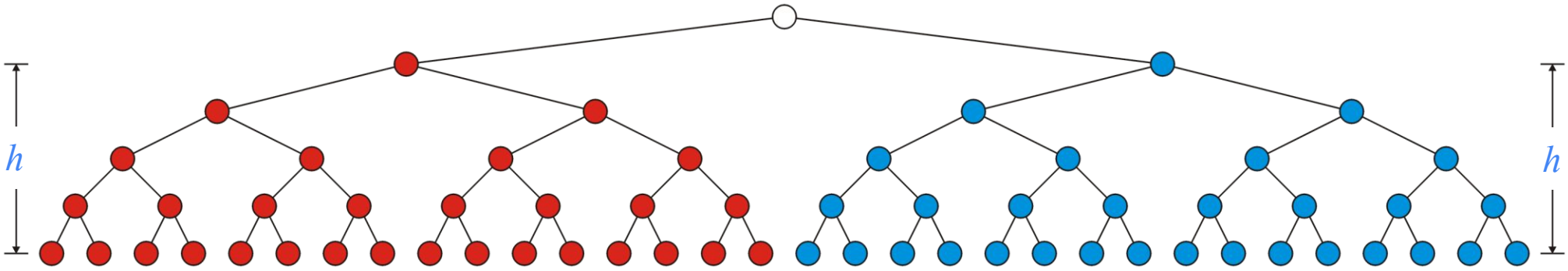


$2^{h+1} - 1$ Nodes

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 $\lg(n + 1) - 1$

Proof

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

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

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

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

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.

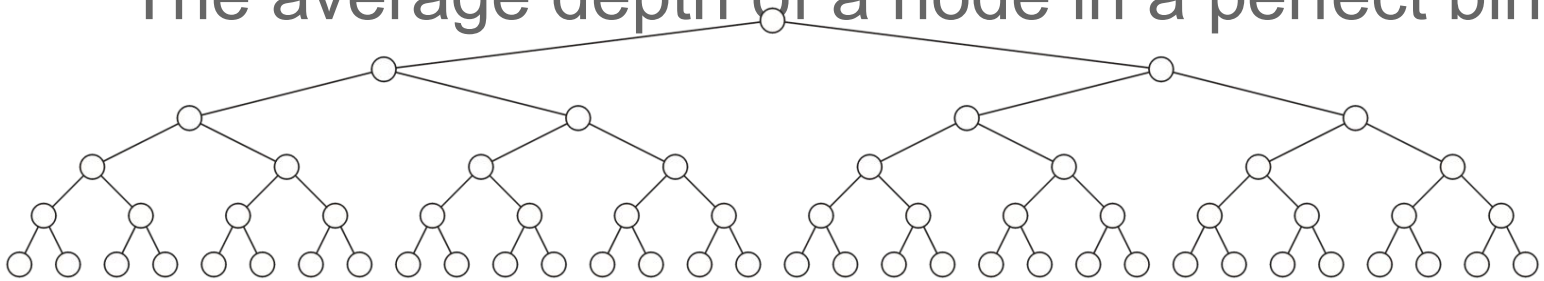
Assume that a perfect binary tree of height h has 2^h leaf nodes and observe that both sub-trees of a perfect binary tree of height $h + 1$ have 2^h leaf nodes.

The Average Depth of a Node

Consequence:

- 50-50 chance that a randomly selected node is a leaf node

The average depth of a node in a perfect binary tree is



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

Sum of the depths

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

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

Number of nodes

Limitations

A perfect binary tree has ideal properties but restricted in the number of nodes: $n = 2^{h+1} - 1$ for $h = 0, 1, \ldots$

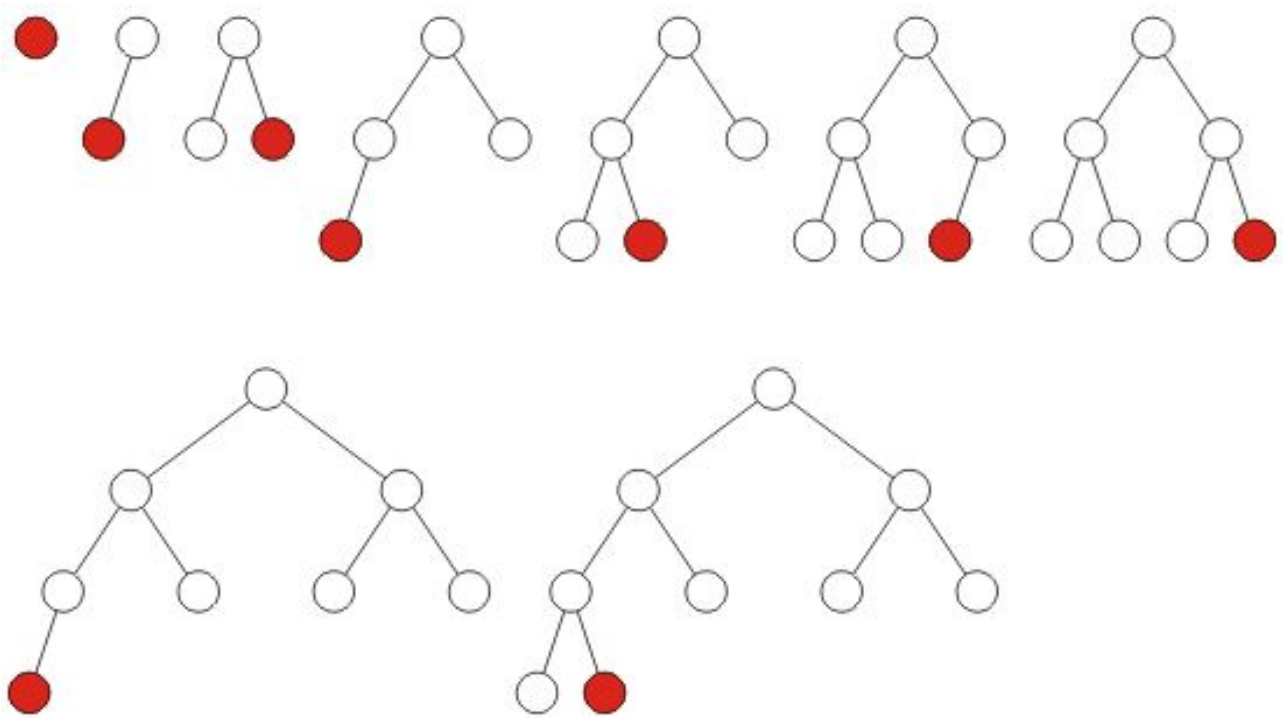
$$1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, ...$$

We require binary trees which are

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

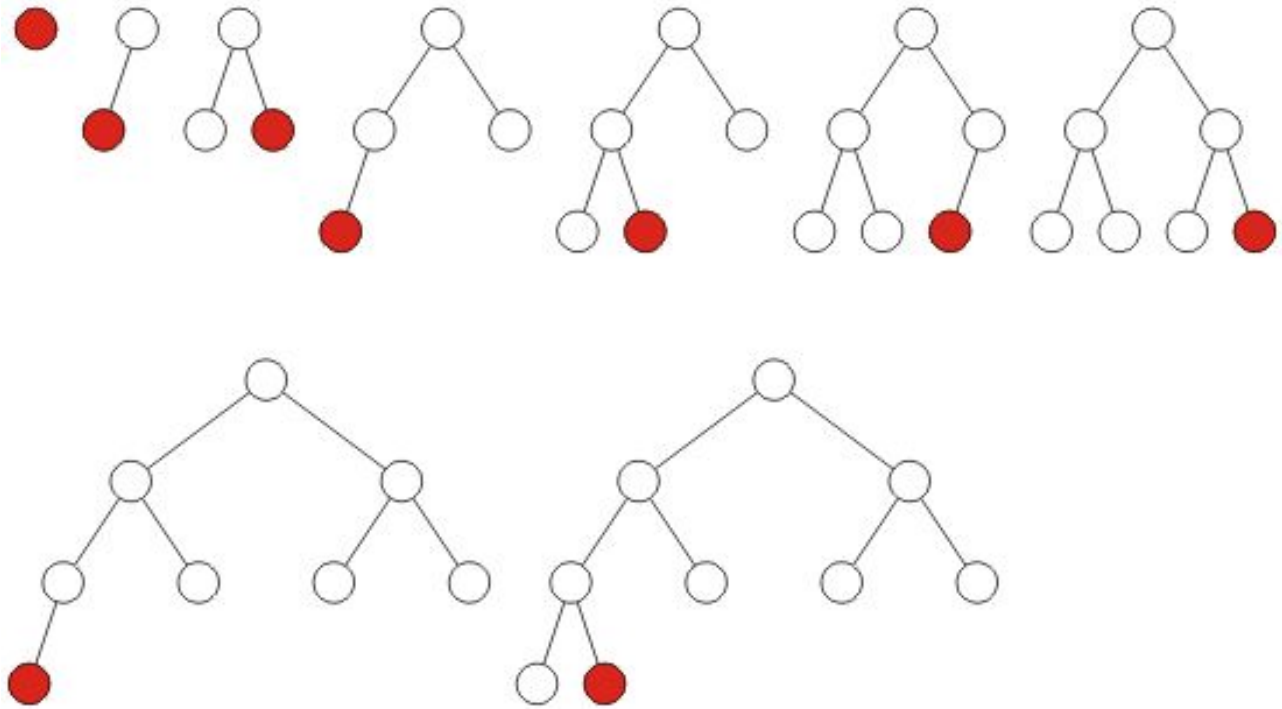
Complete Binary Tree

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



Definition

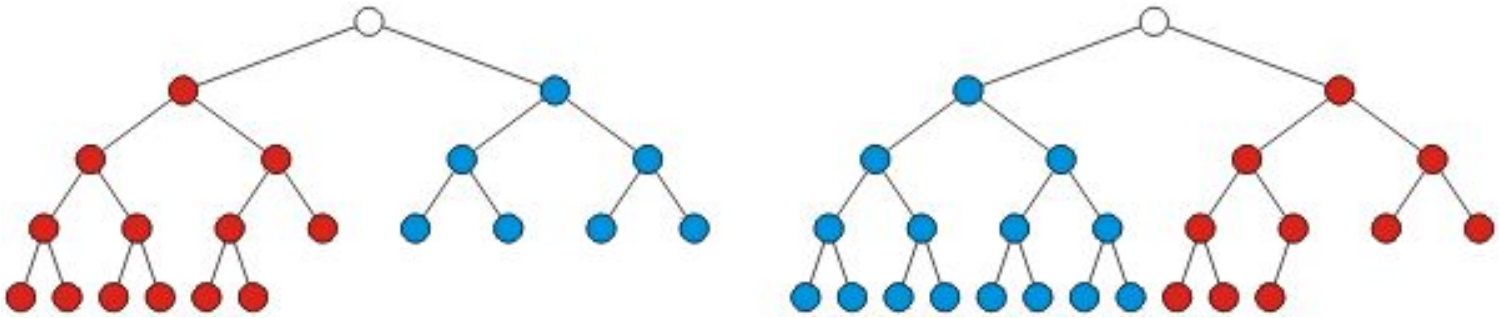
The order is identical to that of a breadth-first traversal



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:

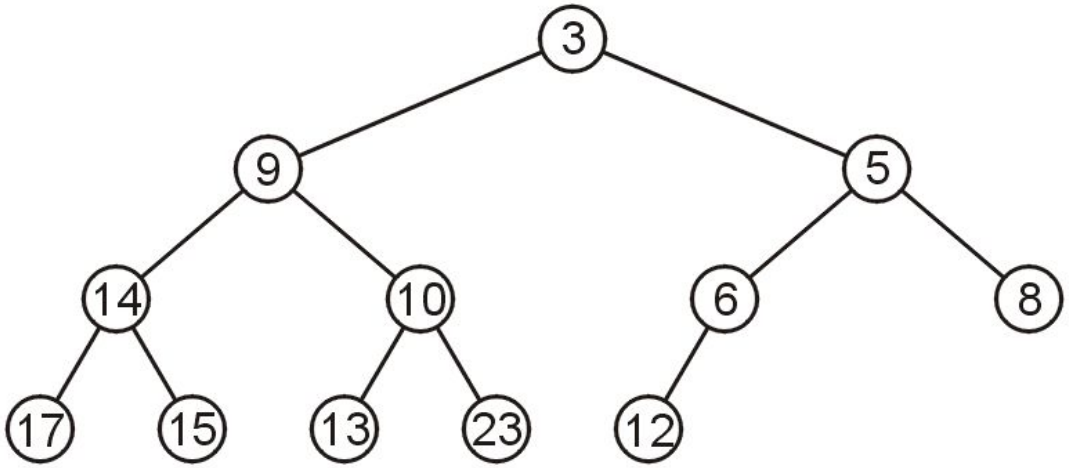
- The left sub-tree is a **complete tree** of height $h - 1$ and the right sub-tree is a **perfect tree** of height $h - 2$, or
- The left sub-tree is **perfect tree** with height $h - 1$ and the right sub-tree is **complete tree** with height $h - 1$



Array storage

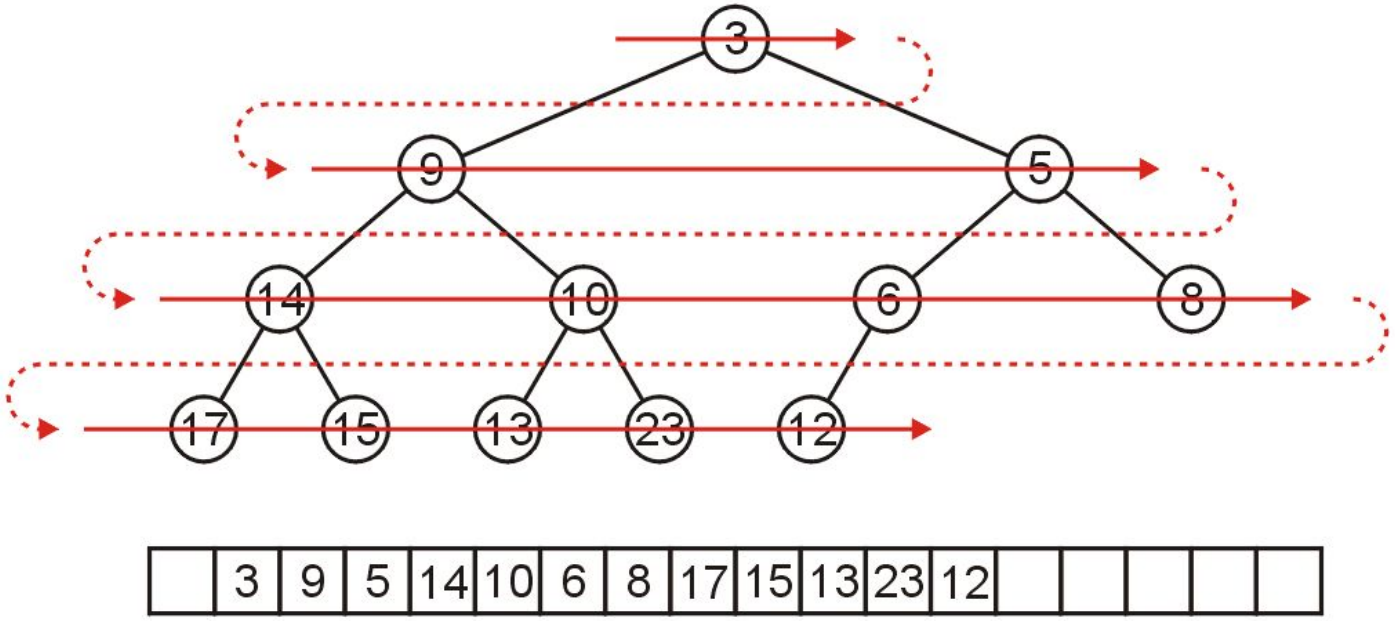
We are able to store a complete tree as an array

- Traverse the tree in breadth-first order, placing the entries



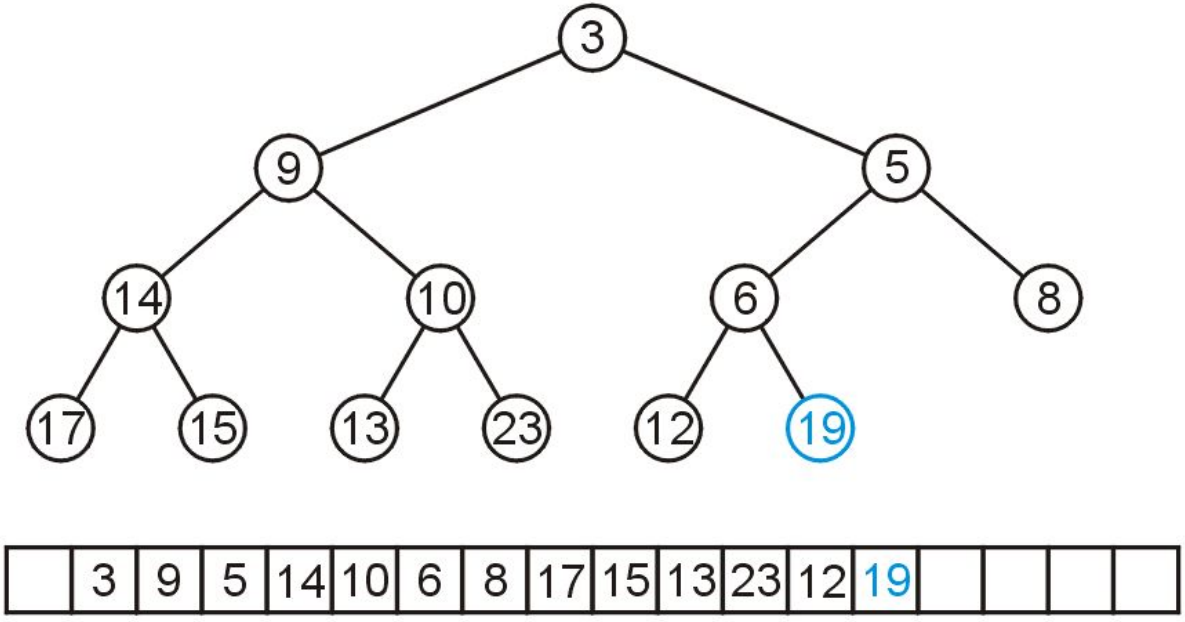
Array storage

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



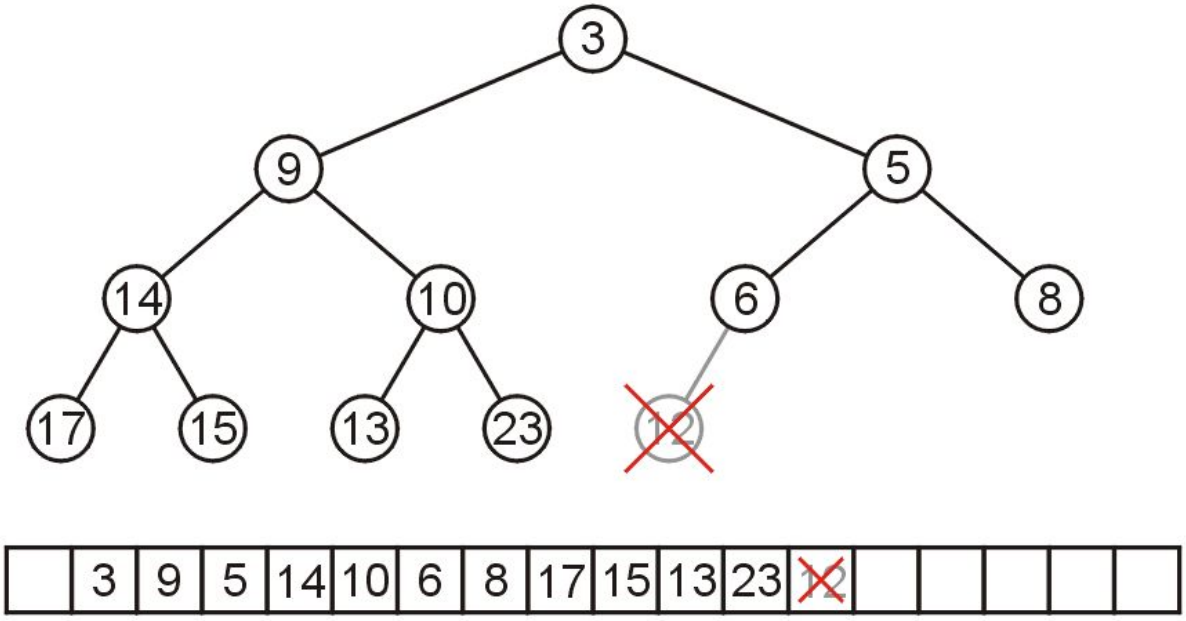
Array storage

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



Array storage

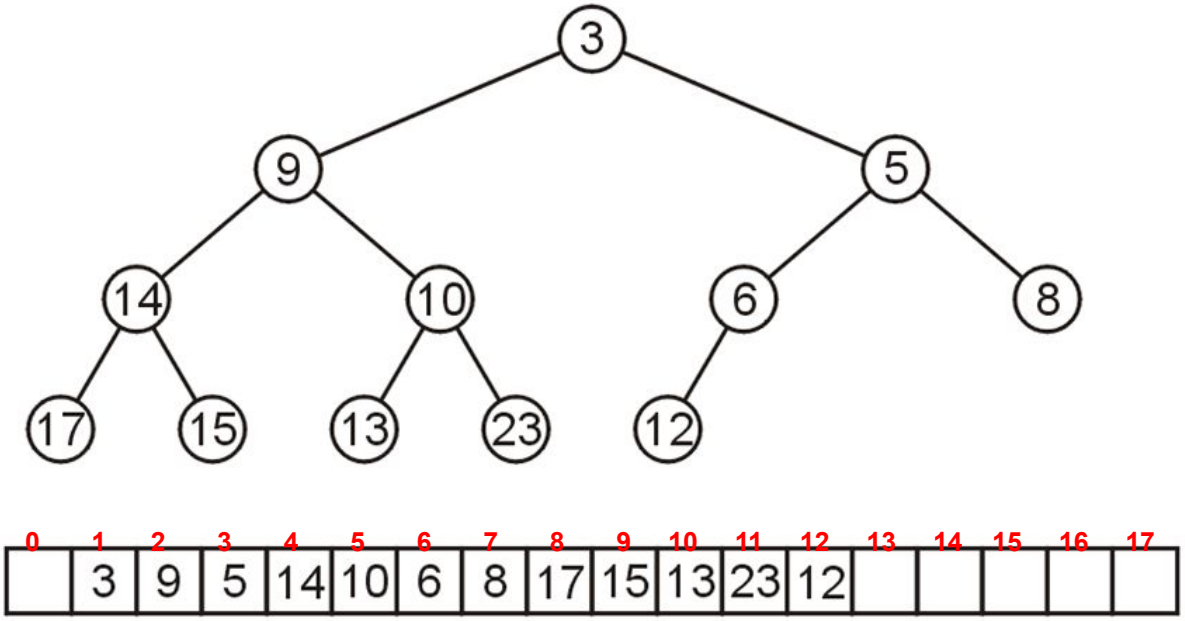
To remove a node while keeping the complete-tree structure, we must remove the last element in the array



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 $k \div 2$

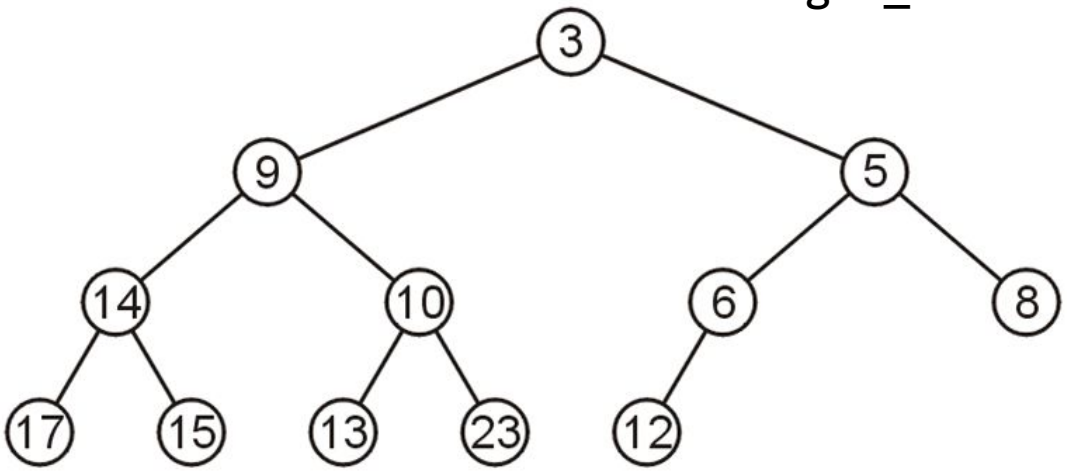


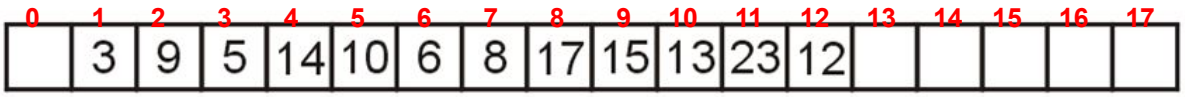
Array storage

Leaving the first entry blank yields a bonus:

– this simplifies the calculations:

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

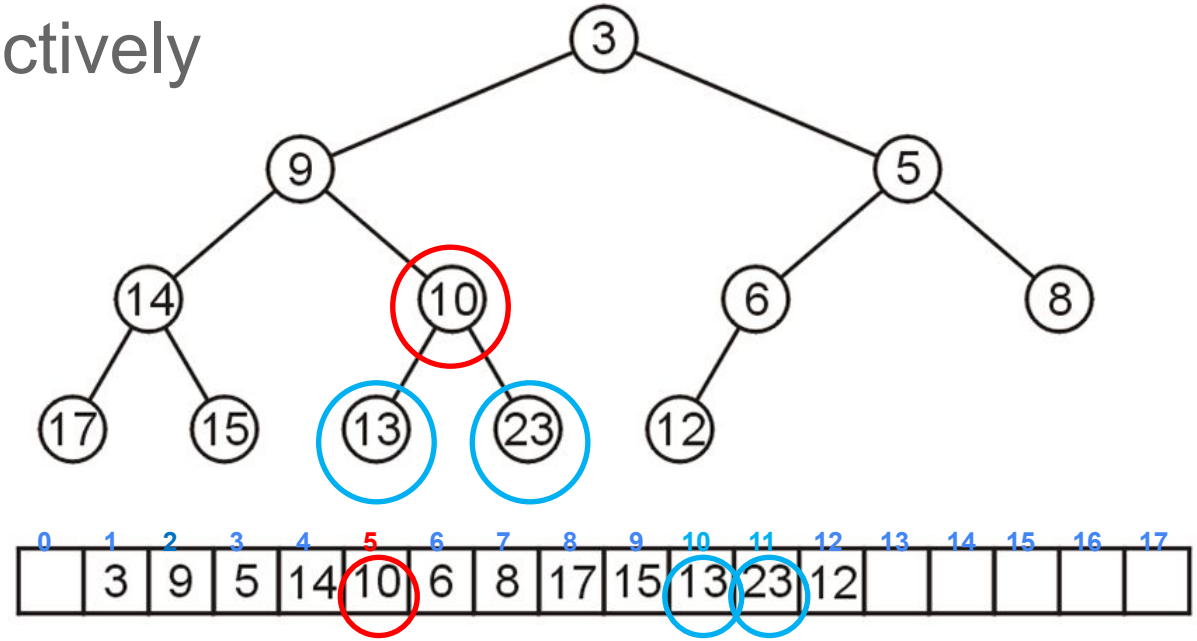




Array storage

For example, node 10 has index 5:

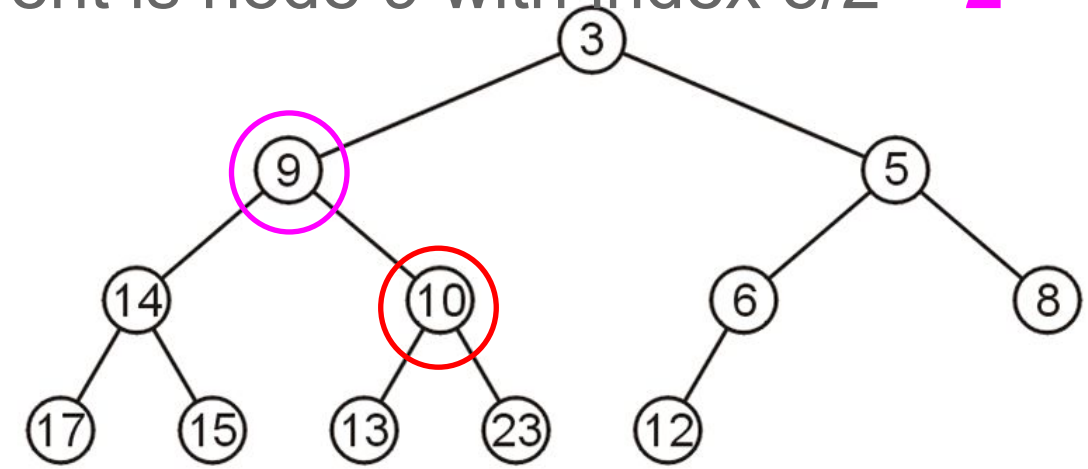
- Its children 13 and 23 have indices 10 and 11, respectively

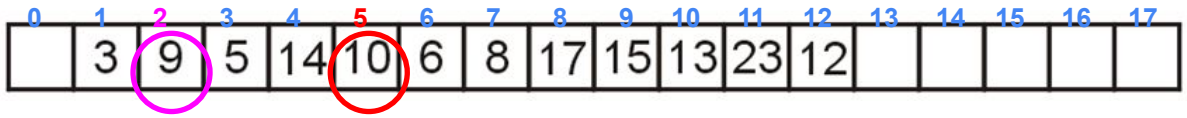


Array storage

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**





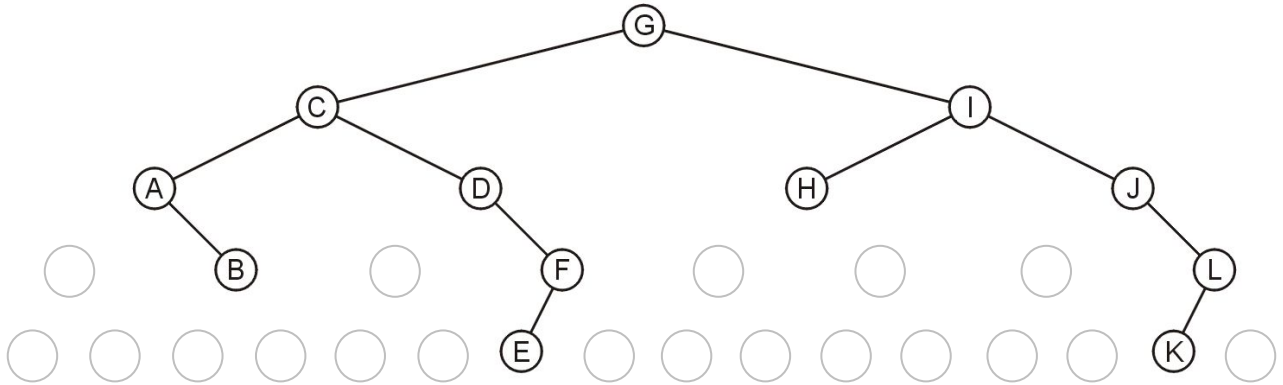
Array storage

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

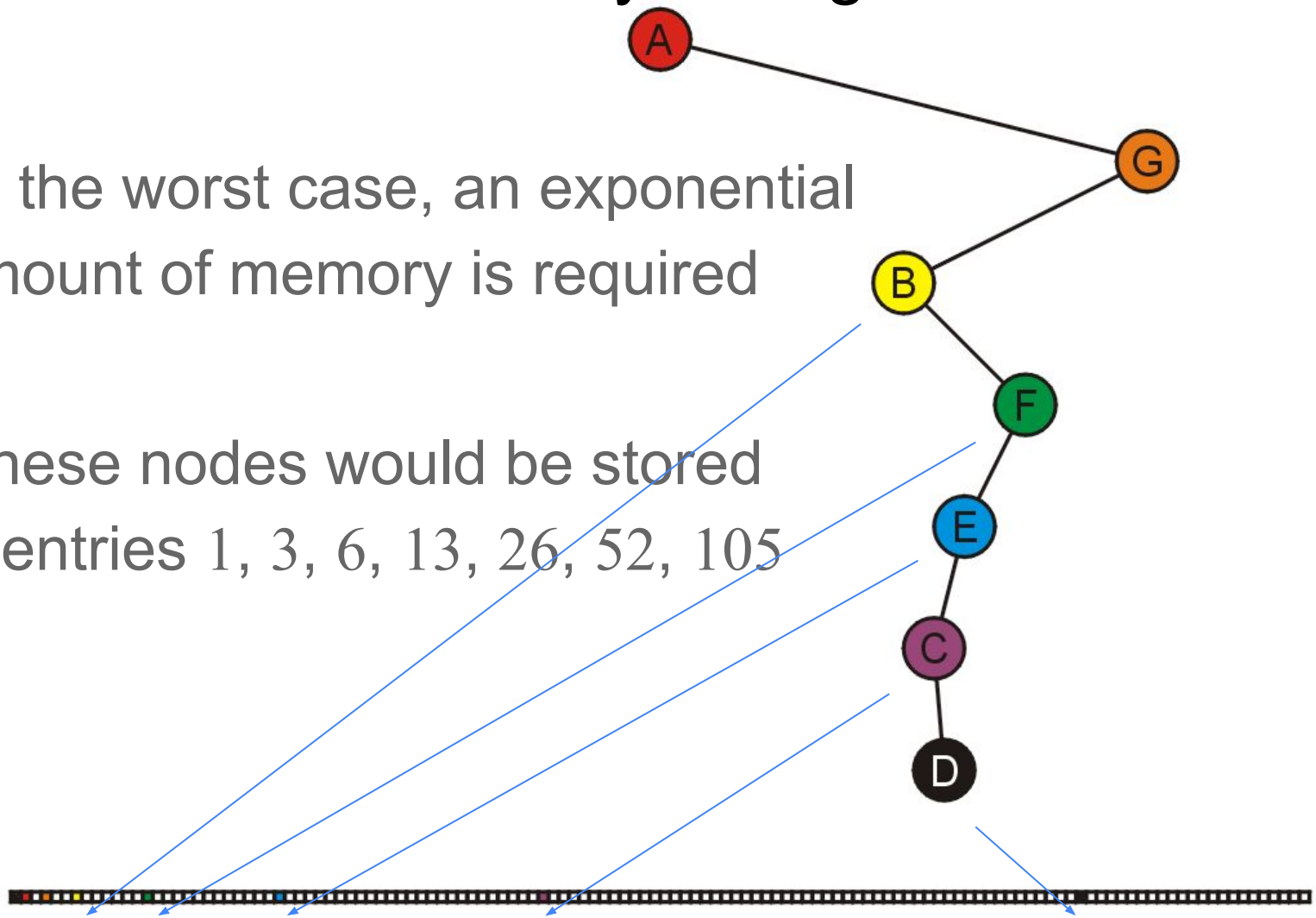
- Adding a child to node K doubles the required memory



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



Background

Run times depend on the height of the trees

As was noted in the previous section:

- The best case height is $\Theta(\ln(n))$
- The worst case height is $\Theta(n)$

The average height of a randomly generated binary search tree is actually $\Theta(\ln(n))$

- However, following random insertions and erases, the average height

  tends to increase to $\Theta\left(\sqrt{n}\right)$

Requirement for Balance

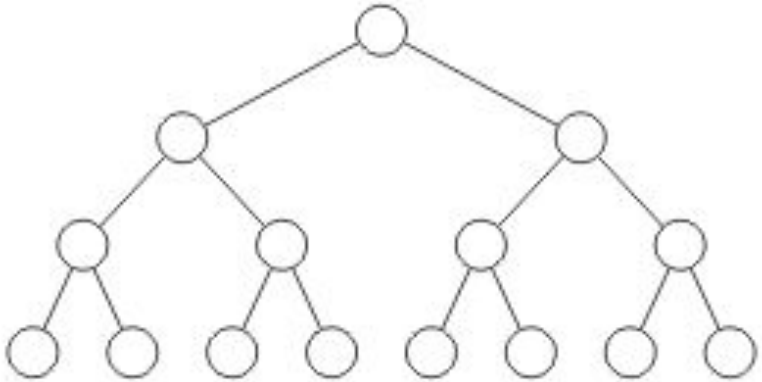
Requirement:

- We must maintain a height which is $\Theta(\ln(n))$

To do this, we will define an idea of balance

Examples

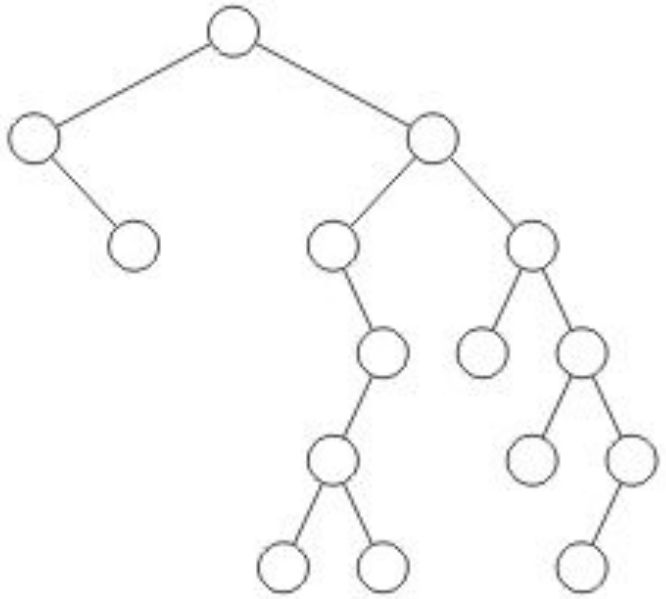
For a perfect tree, all nodes have the same number of descendants on each side



Perfect binary trees are balanced while linked lists are not

Examples

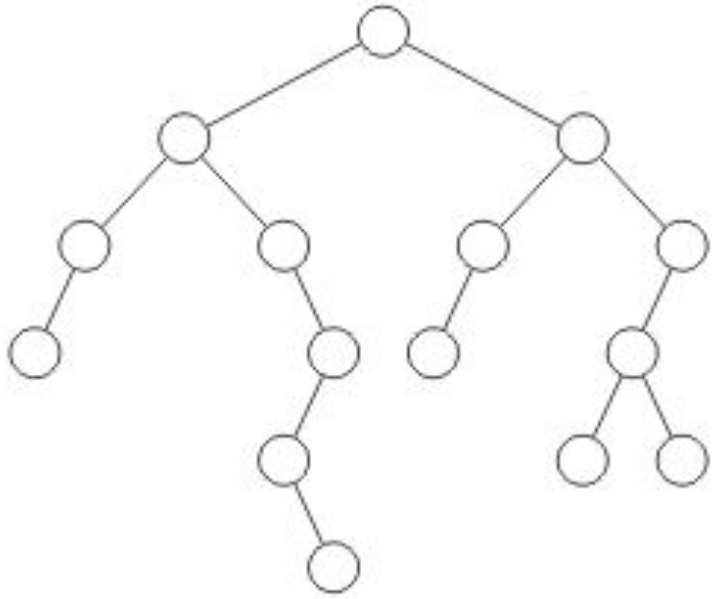
This binary tree would also probably not be considered to be “balanced” at the root node



Examples

How about this example?

- The root seems balanced, but what about the left sub-tree?



Definition for Balance

We must develop a quantitative definition of *balance* which can be applied

Balanced may be defined by:

- *Height balancing*: comparing the heights of the two sub trees
- *Null-path-length balancing*: comparing the null-path-length of each of the two sub-trees (the length to the closest null sub-tree/empty node)
- *Weight balancing*: comparing the number of null sub-trees in each of the two sub trees

If a tree satisfies the definition of balance, its height is $\Theta(\ln(n))$

Definition for Balance

We will see one definition of height balancing:

- AVL trees

We will also look at B+-trees

- Balanced trees, but not binary trees