

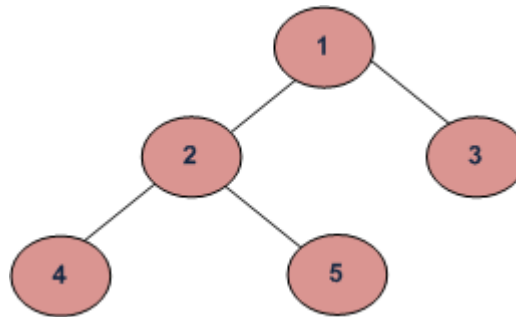
CS101 Data Structures

Binary Trees

Textbook Ch B.5.3, 10.4

Orders of DFS traversal

- Inorder (Left, Root, Right) : 4 2 5 1 3
- Preorder (Root, Left, Right) : 1 2 4 5 3
- Postorder (Left, Right, Root) : 4 5 2 3 1
- Breadth-First or Level Order Traversal: 1 2 3 4 5



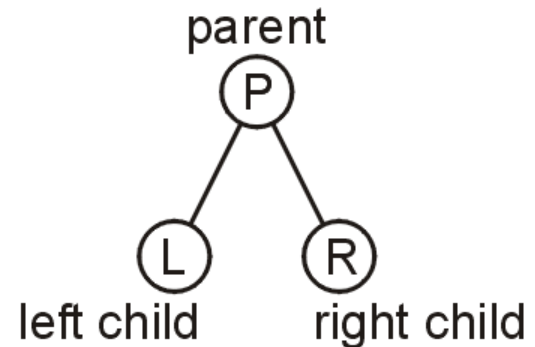
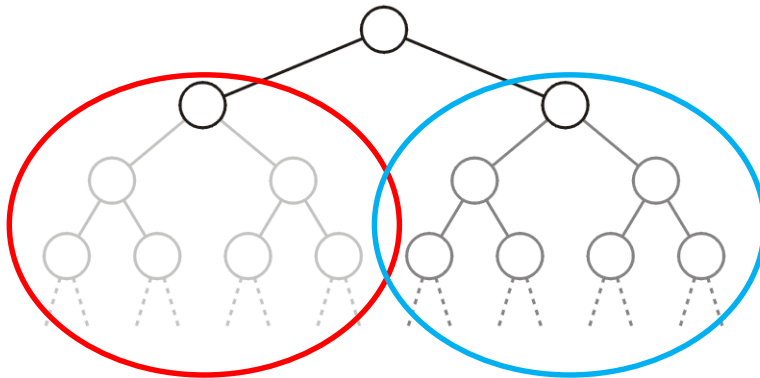
Outline

- Binary tree
- Perfect binary tree
- Complete binary tree
- Left-child right-sibling binary tree

Definition

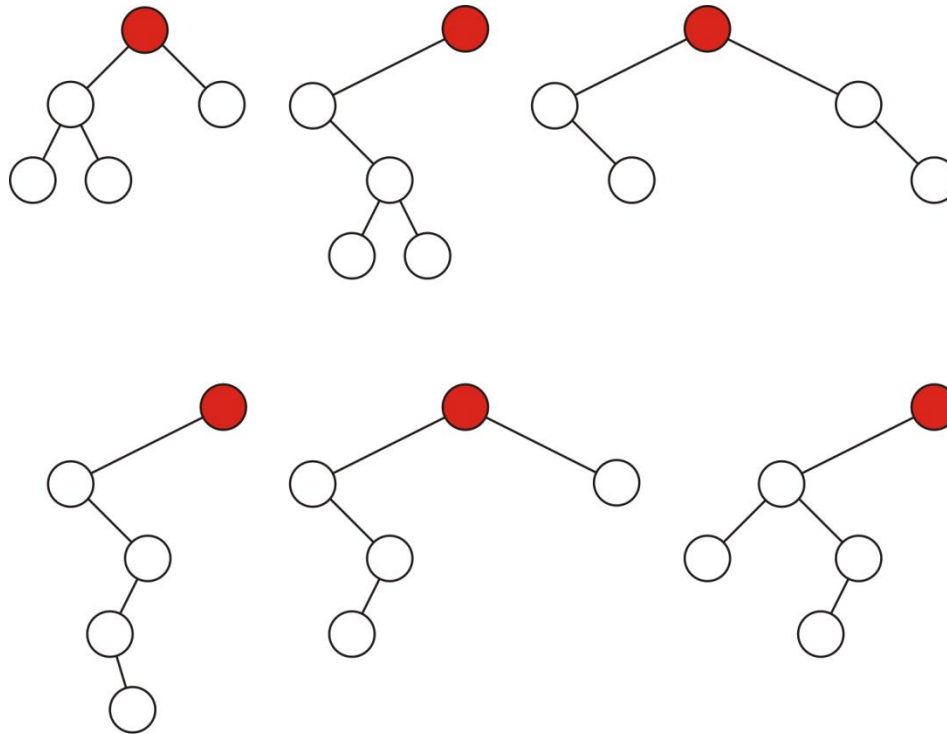
A binary tree is a restriction where each node has exactly two children:

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



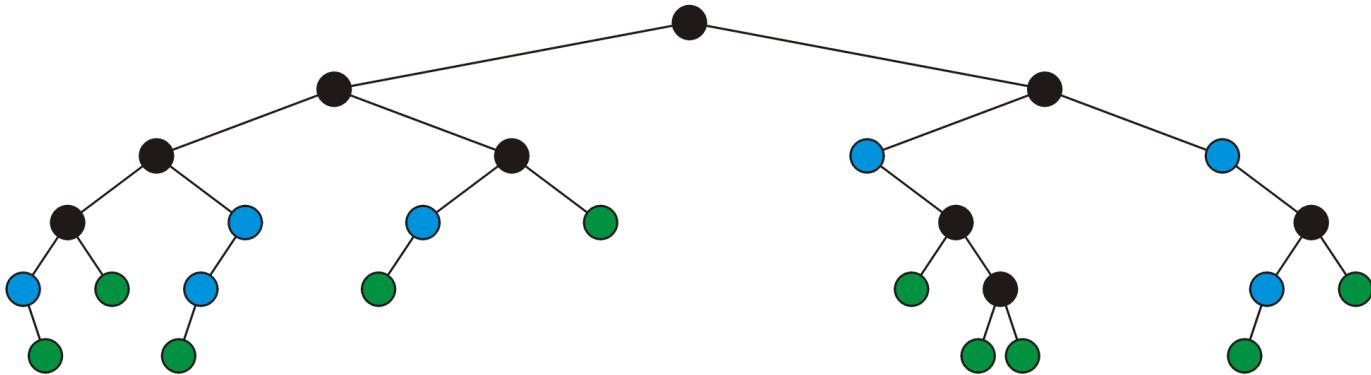
Definition

Some 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:

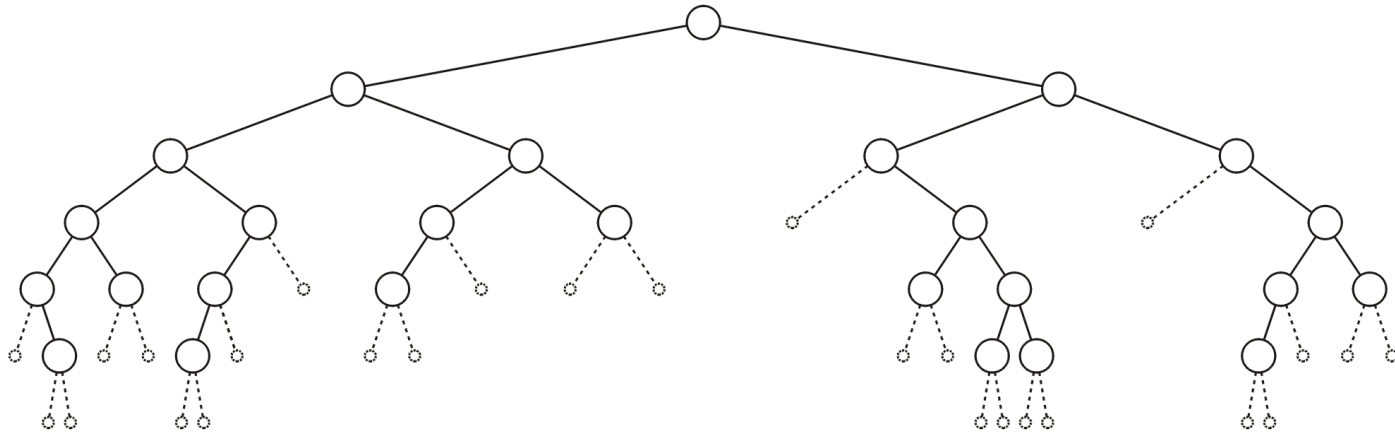
full nodes ●

neither ●

leaf nodes ●

Definition

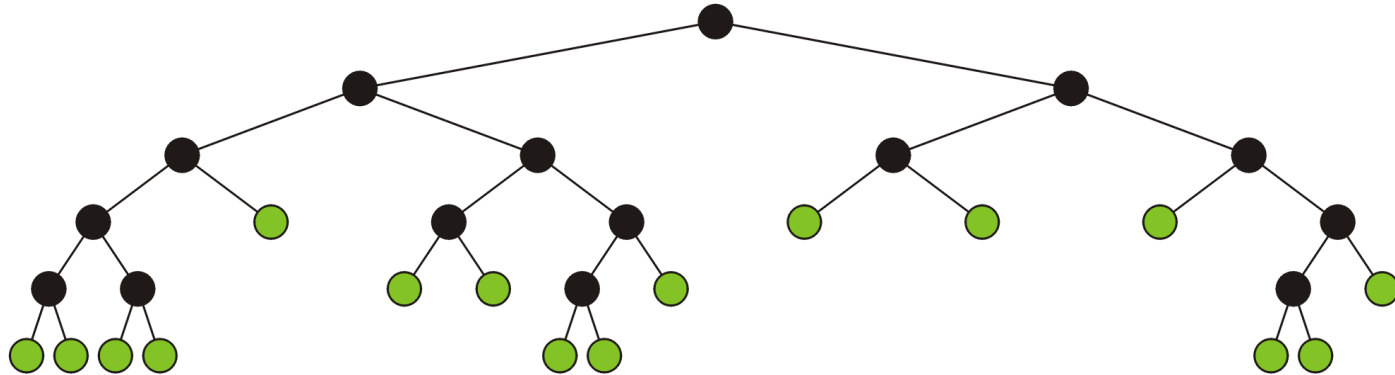
An *empty node* or a *null sub-tree* is any location where a new leaf node could be appended



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:

```
template <typename Type>
class Binary_node {
    protected:
        Type element;
        Binary_node *left_tree;
        Binary_node *right_tree;

    public:
        Binary_node( Type const & );

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

        bool is_leaf() const;
        int size() const;
}
```

Binary Node Class

We will usually only construct new leaf nodes

```
template <typename Type>
Binary_node<Type>::Binary_node( Type const &obj ):
    element( obj ),
    left_tree( nullptr ),
    right_tree( nullptr ) {
    // Empty constructor
}
```

Binary Node Class

The accessors are similar to that of `Single_list`

```
template <typename Type>
Type Binary_node<Type>::retrieve() const {
    return element;
}
```

```
template <typename Type>
Binary_node<Type> *Binary_node<Type>::left() const {
    return left_tree;
}
```

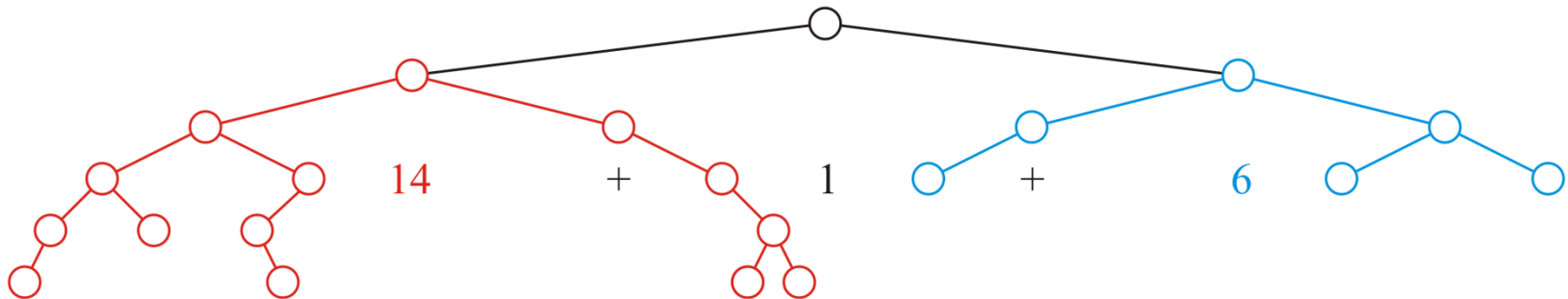
```
template <typename Type>
Binary_node<Type> *Binary_node<Type>::right() const {
    return right_tree;
}
```

Binary Node Class

```
template <typename Type>
bool Binary_node<Type>::is_leaf() const {
    return left() == nullptr && right() == nullptr;
}
```

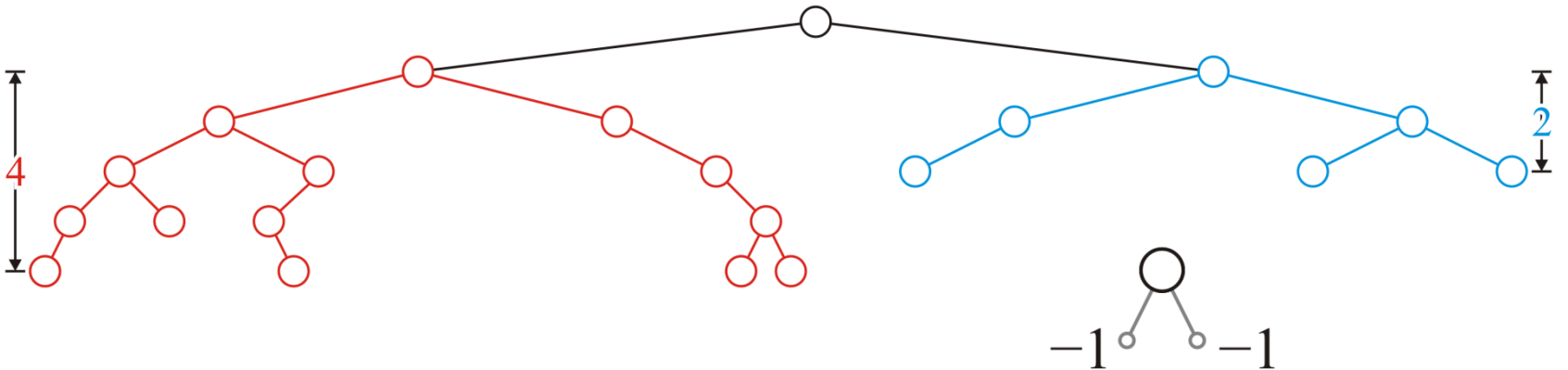
Size

```
template <typename Type>
int Binary_node<Type>::size() const {
    return 1
        + left() == nullptr? 0 : left()->size()
        + right() == nullptr? 0 : right()->size();
}
```



Height

```
template <typename Type>
int Binary_node<Type>::height() const {
    return empty() ? -1 :
        1 + std::max( left()->height(), right()->height() );
}
```



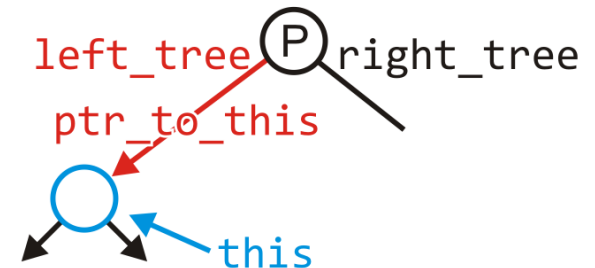
Clear

Removing all the nodes in a tree is similarly recursive:

```
template <typename Type>
void Binary_node<Type>::clear( Binary_node *&ptr_to_this ) {
    if ( empty() ) {
        return;
    }

    left()->clear( left_node );
    right()->clear( right_node );

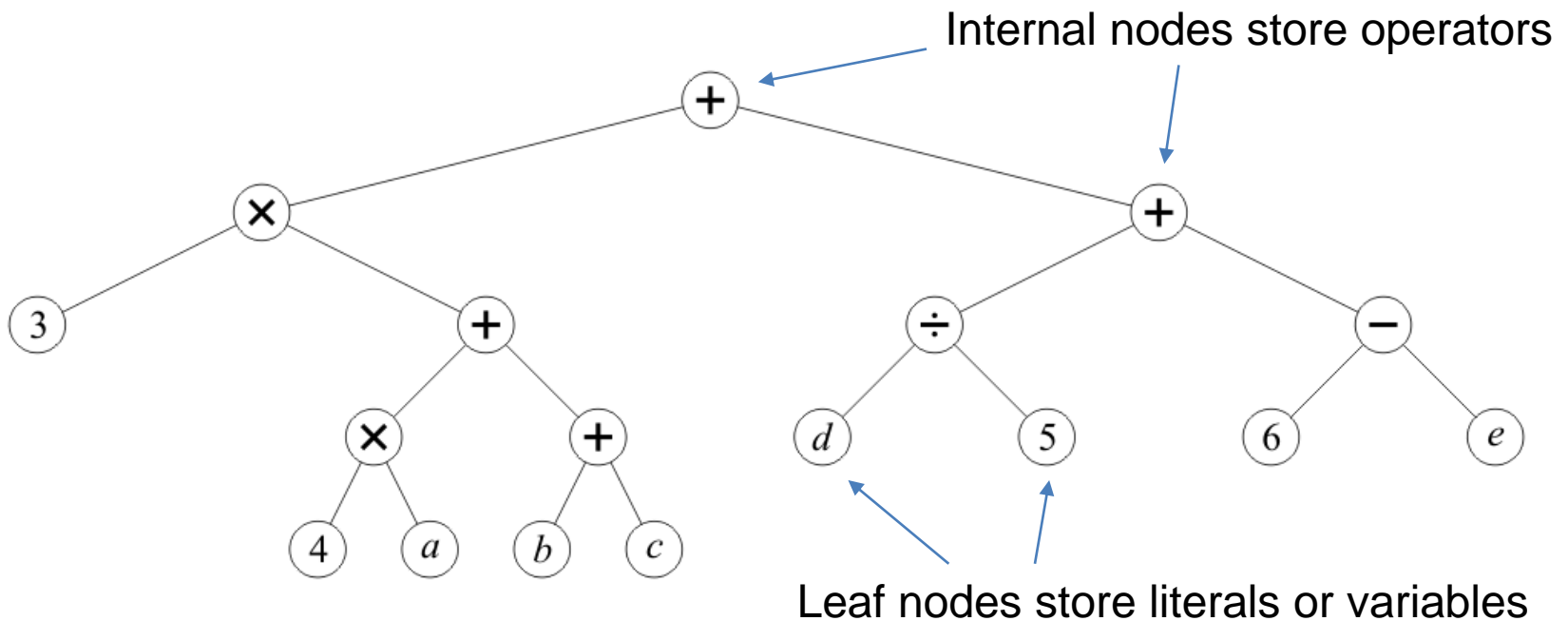
    delete this;
    ptr_to_this = nullptr;
}
```



Application: Expression Trees

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

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



Application: Expression Trees

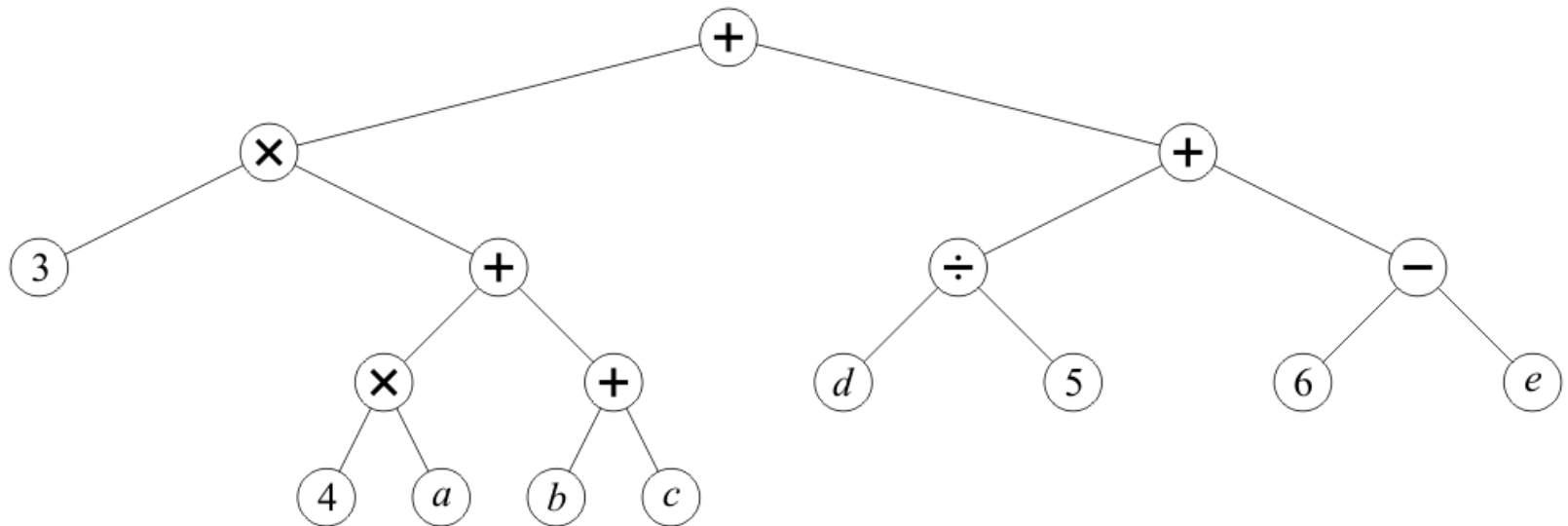
Observations:

- 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:

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

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

Computers think in post-order:

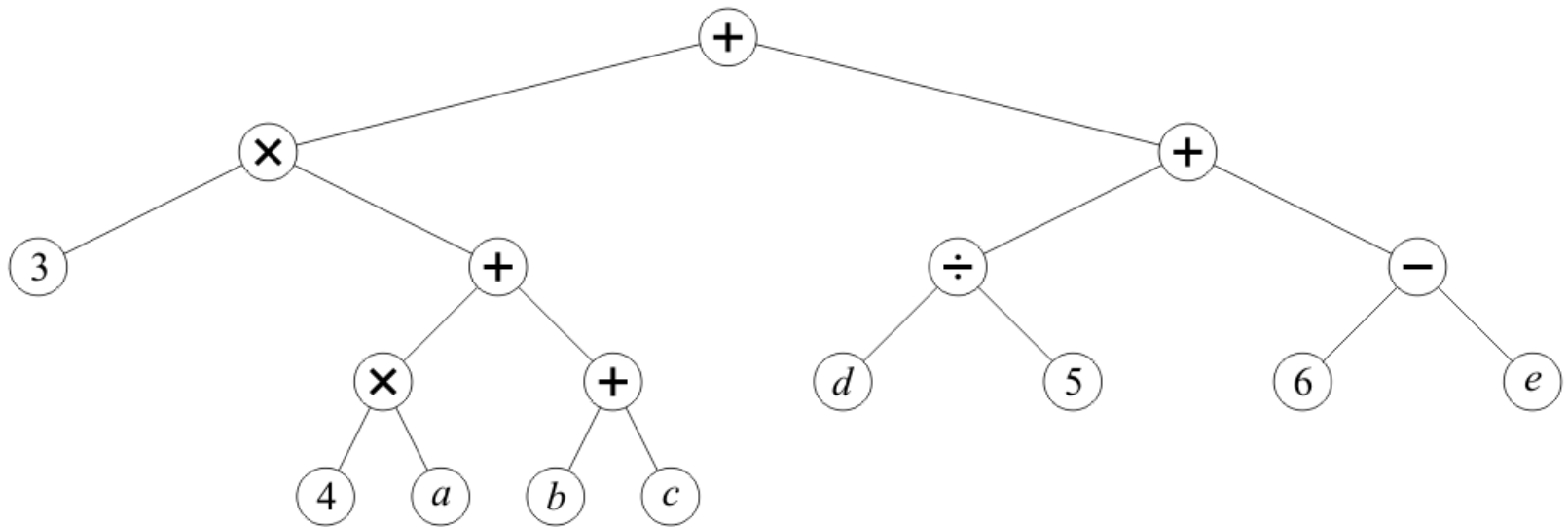
- Both operands must be loaded into registers
- The operation is then called on those registers

Humans think in in-order:

- First, the left sub-tree is traversed
- Then, the current node is visited
- Finally, the right-sub-tree is traversed

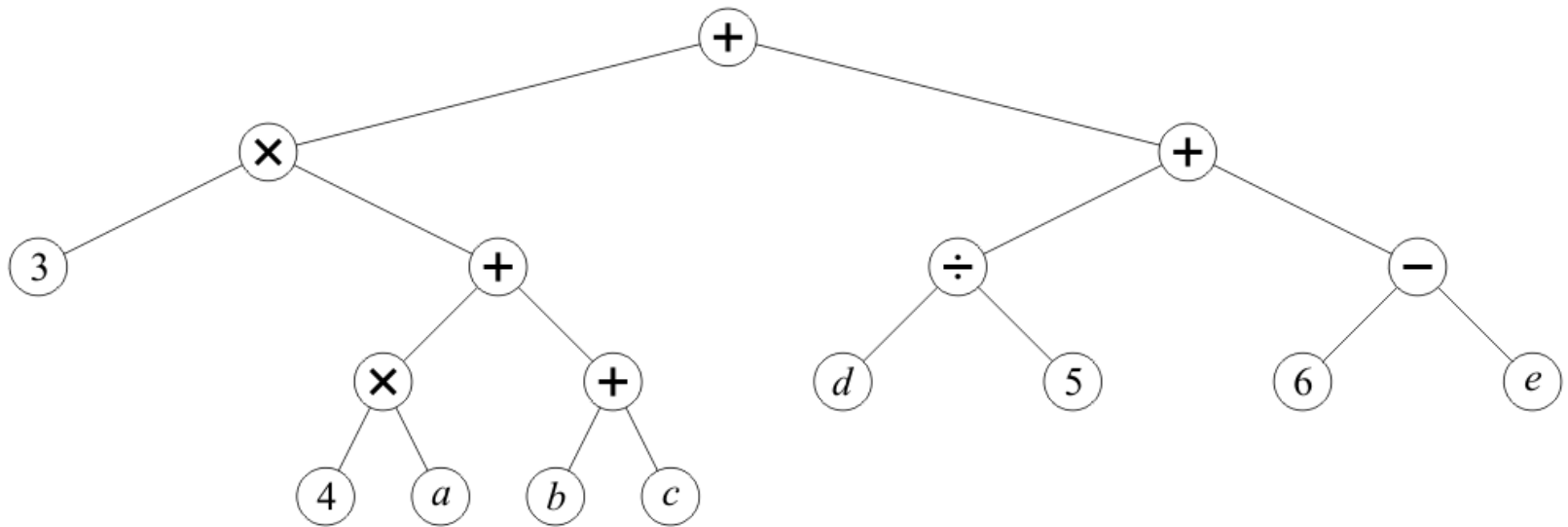
This is called an *in-order traversal*

In-order Traversal



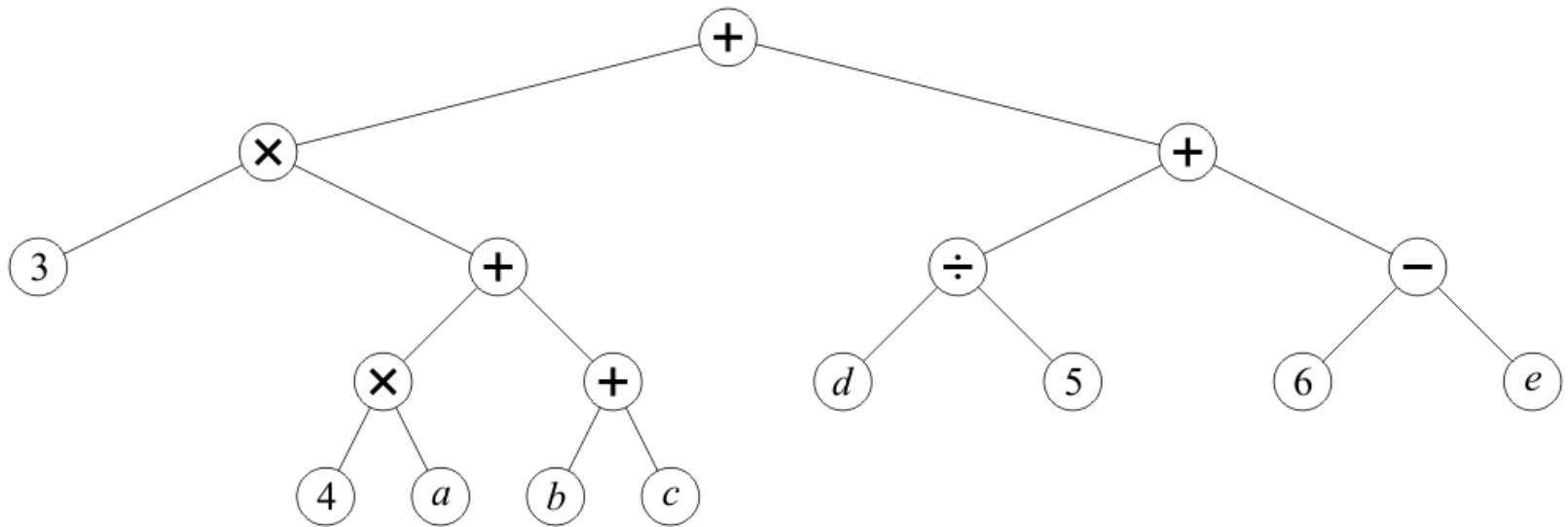
3

In-order Traversal



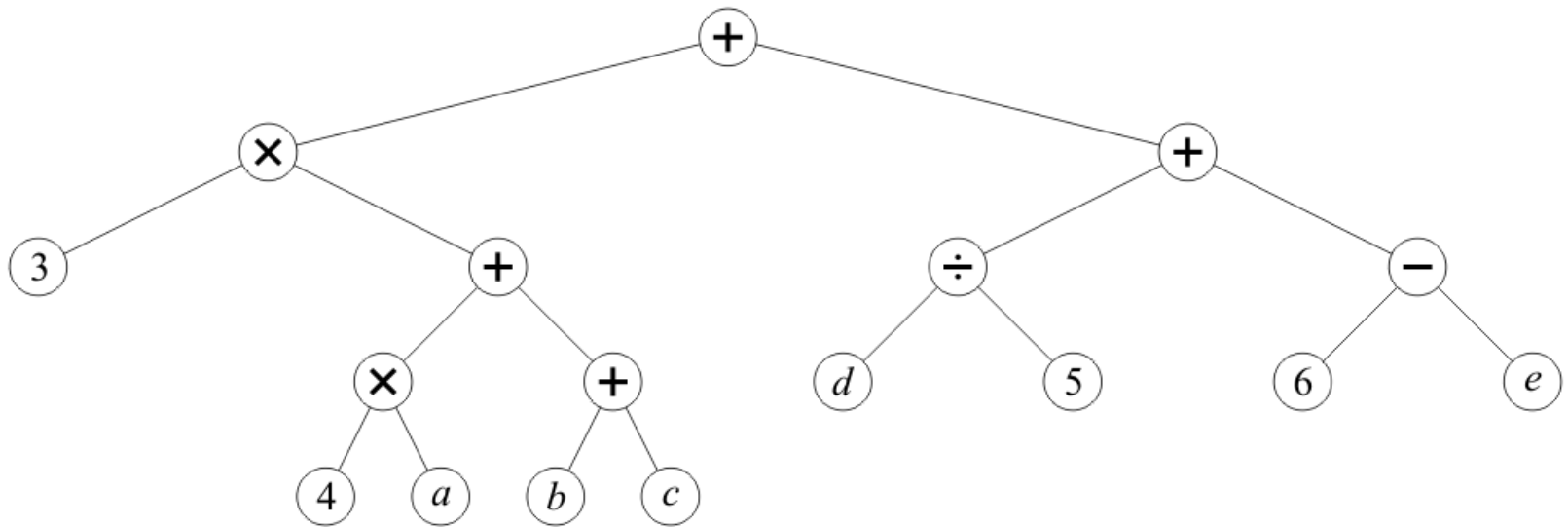
$3 \times$

In-order Traversal



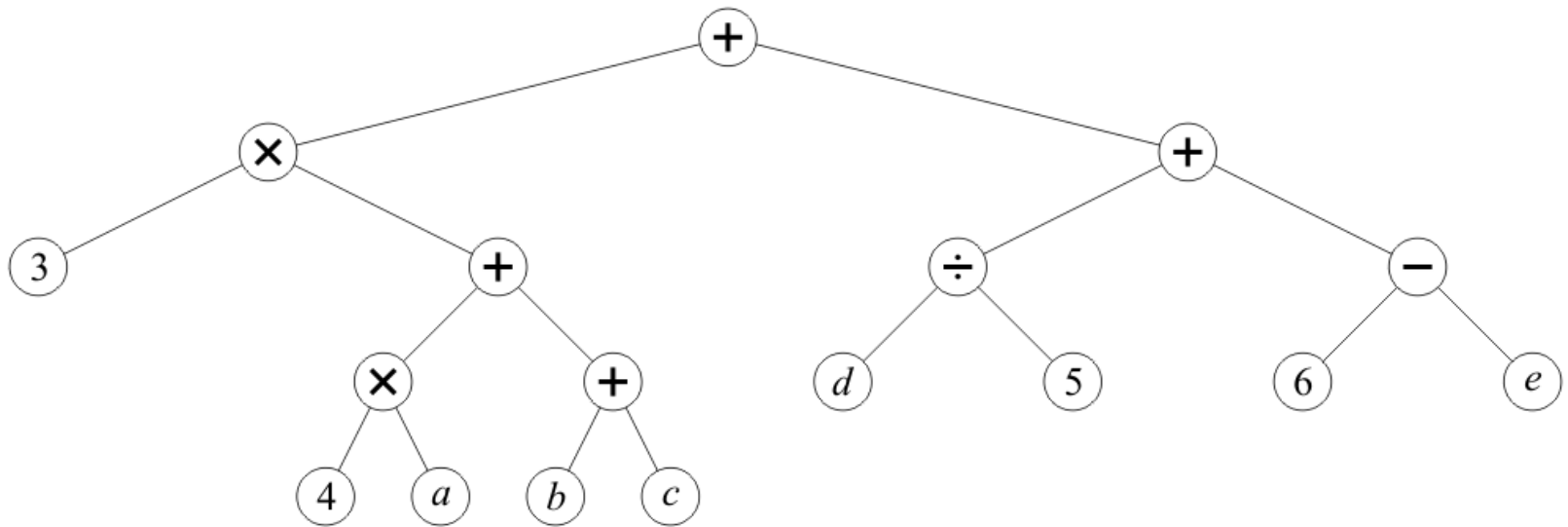
$$3 \times 4$$

In-order Traversal



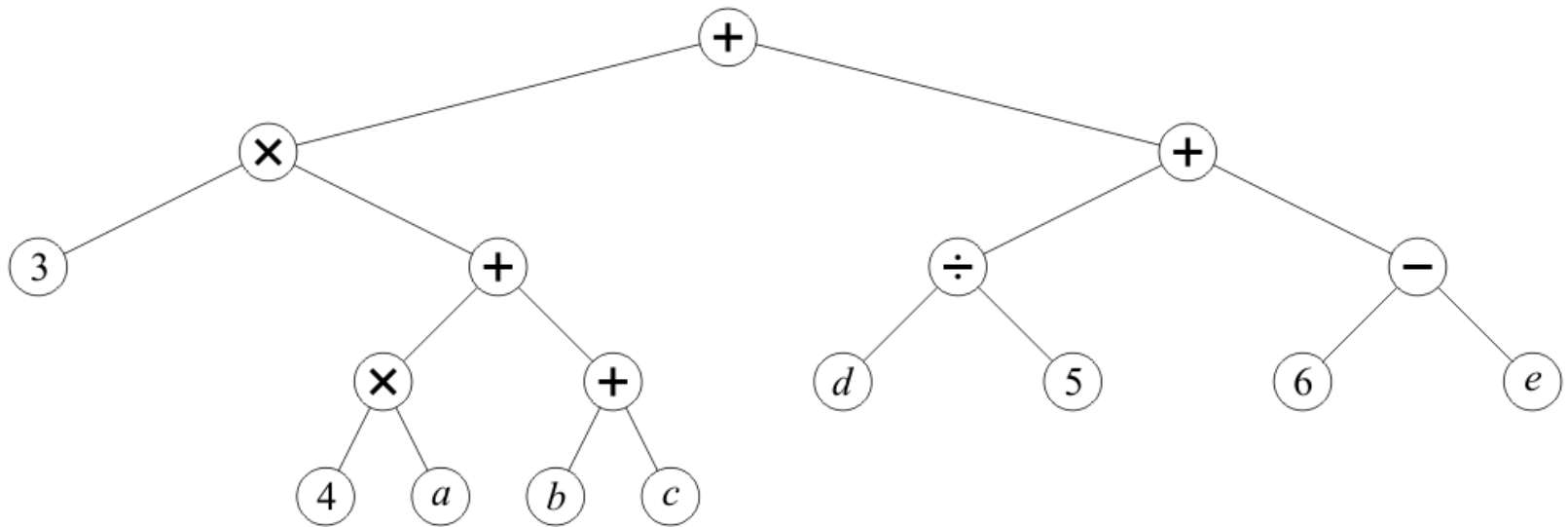
$3 \times 4 \times$

In-order Traversal



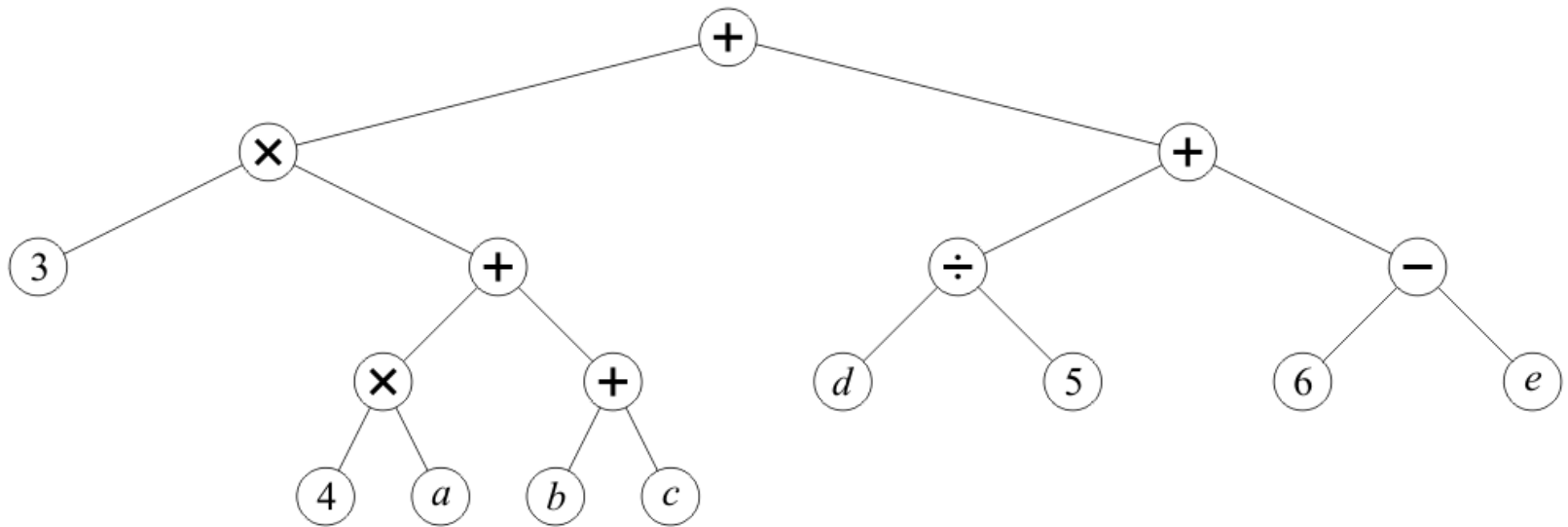
$$3 \times 4 \times a$$

In-order Traversal



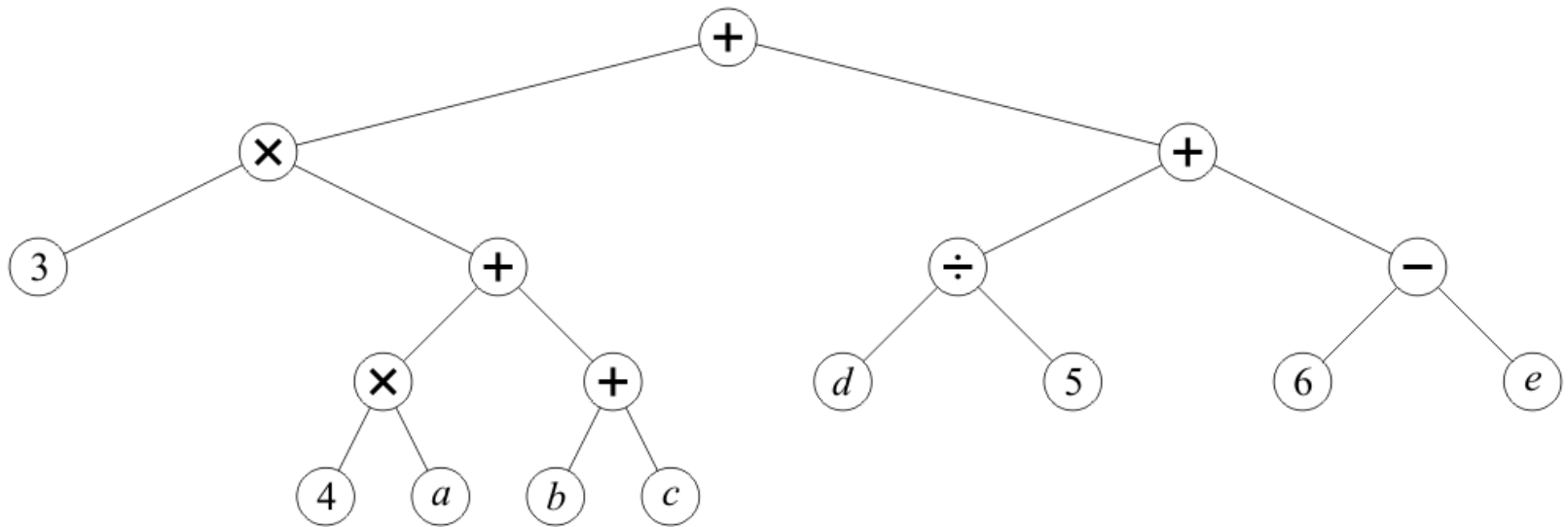
$3 \times 4 \times a +$

In-order Traversal



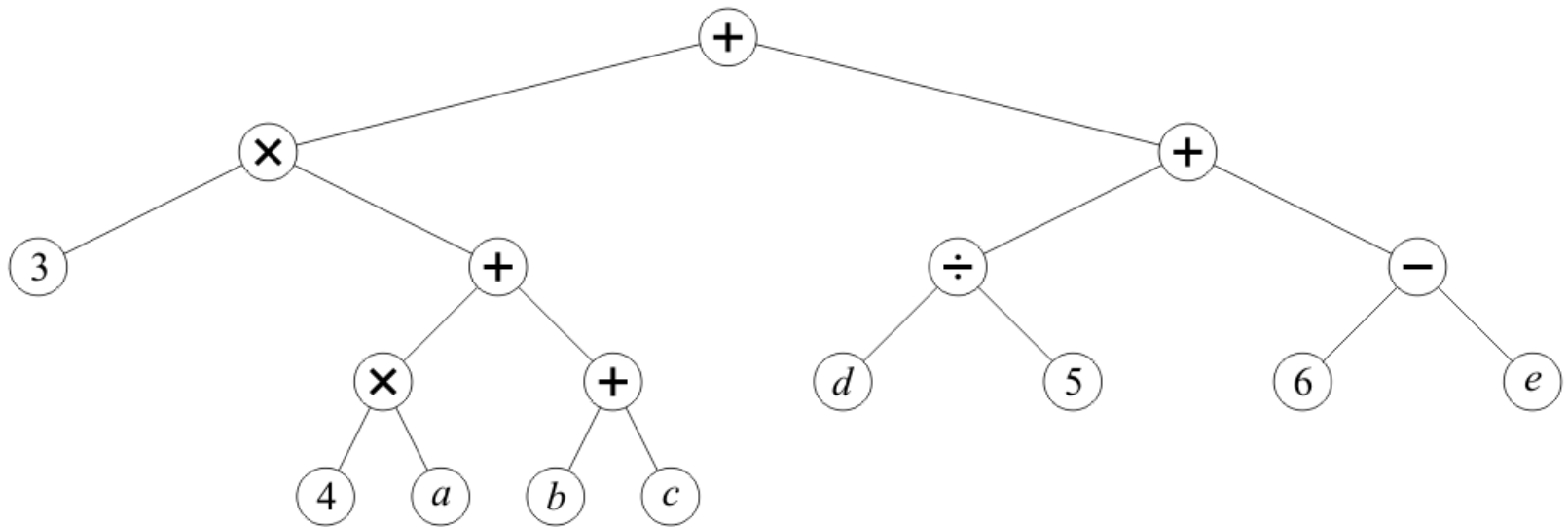
$$3 \times 4 \times a + b$$

In-order Traversal



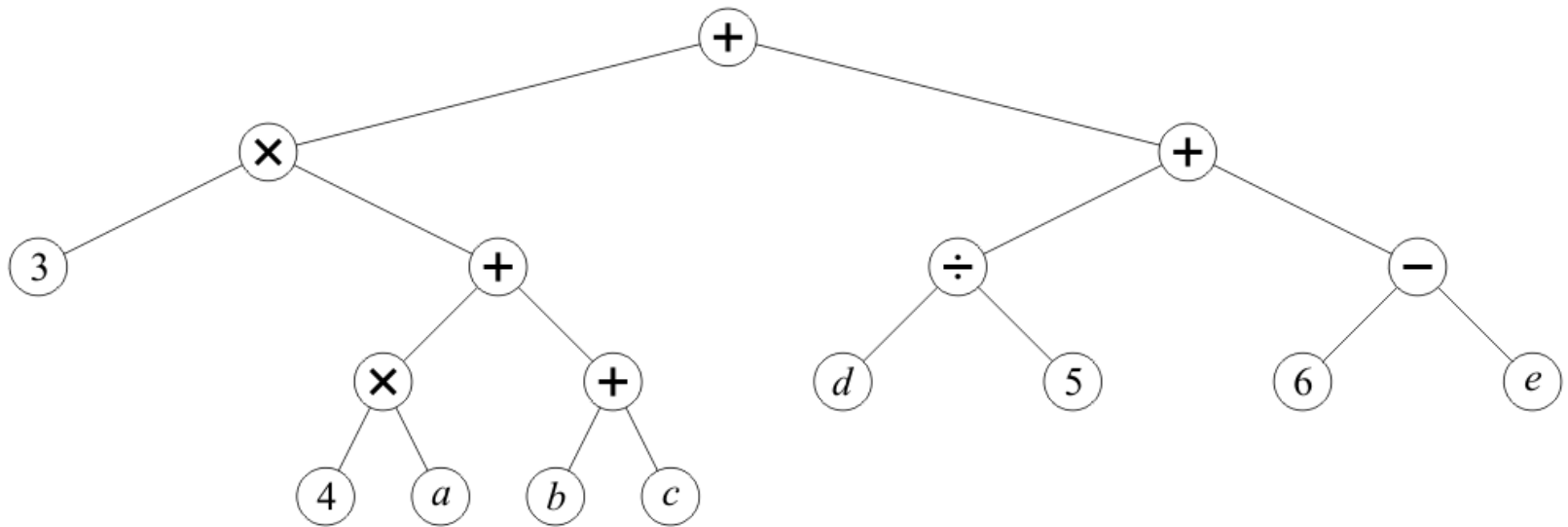
$3 \times 4 \times a + b +$

In-order Traversal



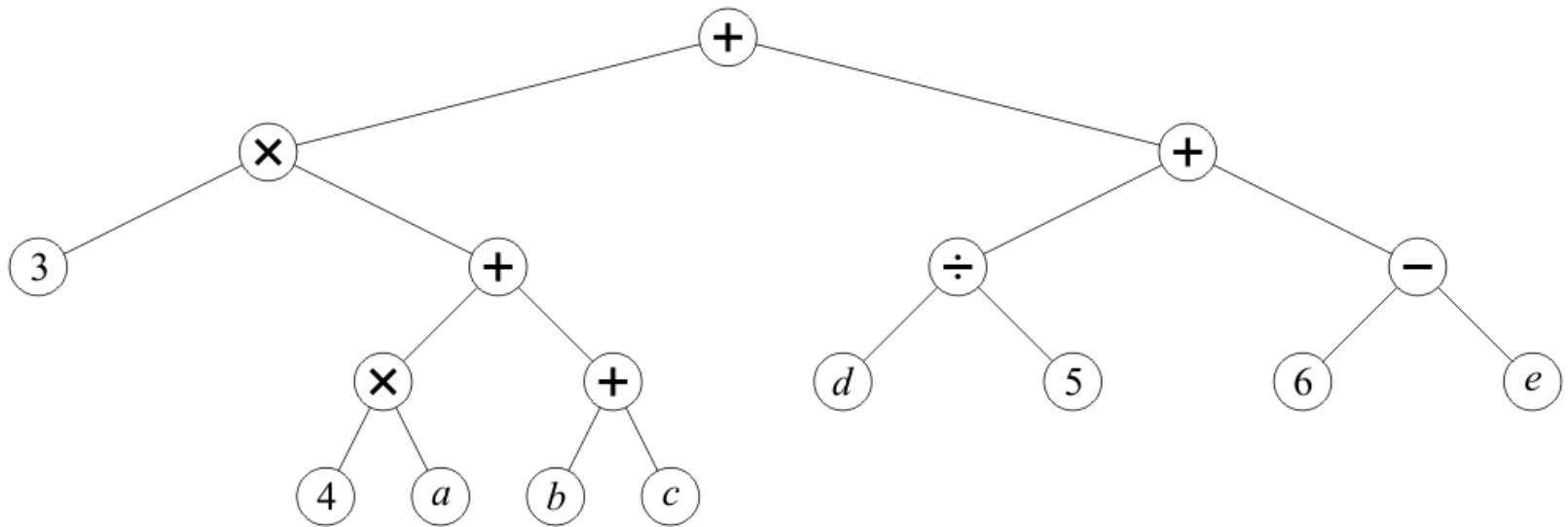
$$3 \times 4 \times a + b + c$$

In-order Traversal



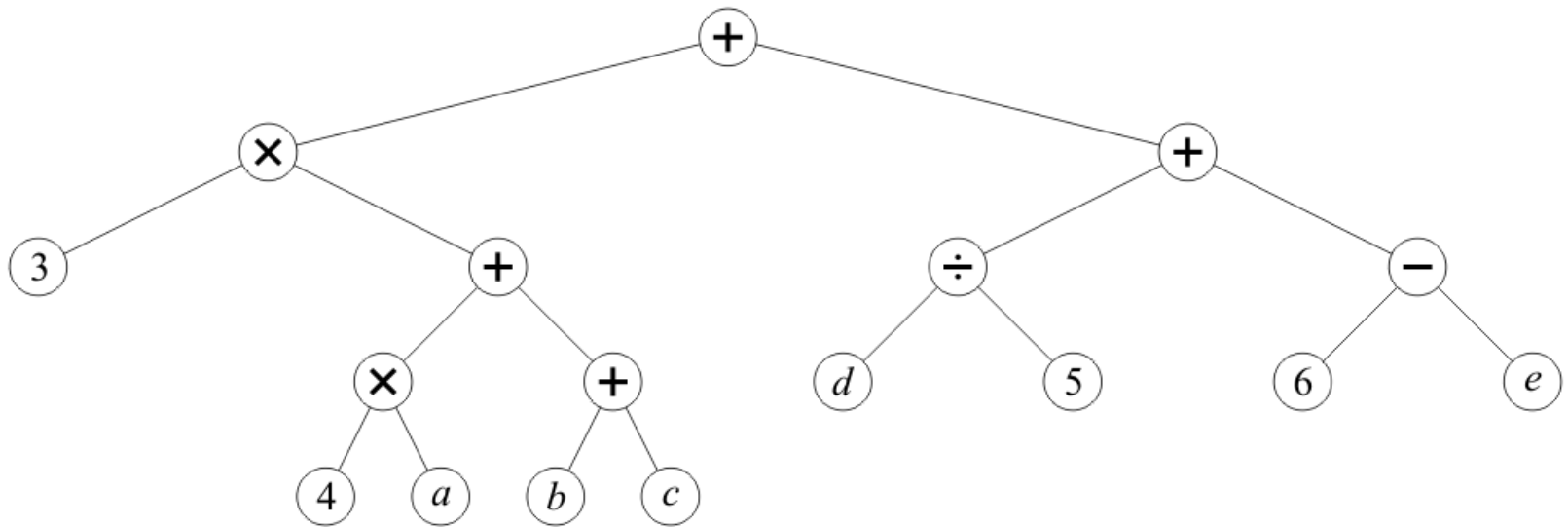
$3 \times 4 \times a + b + c +$

In-order Traversal



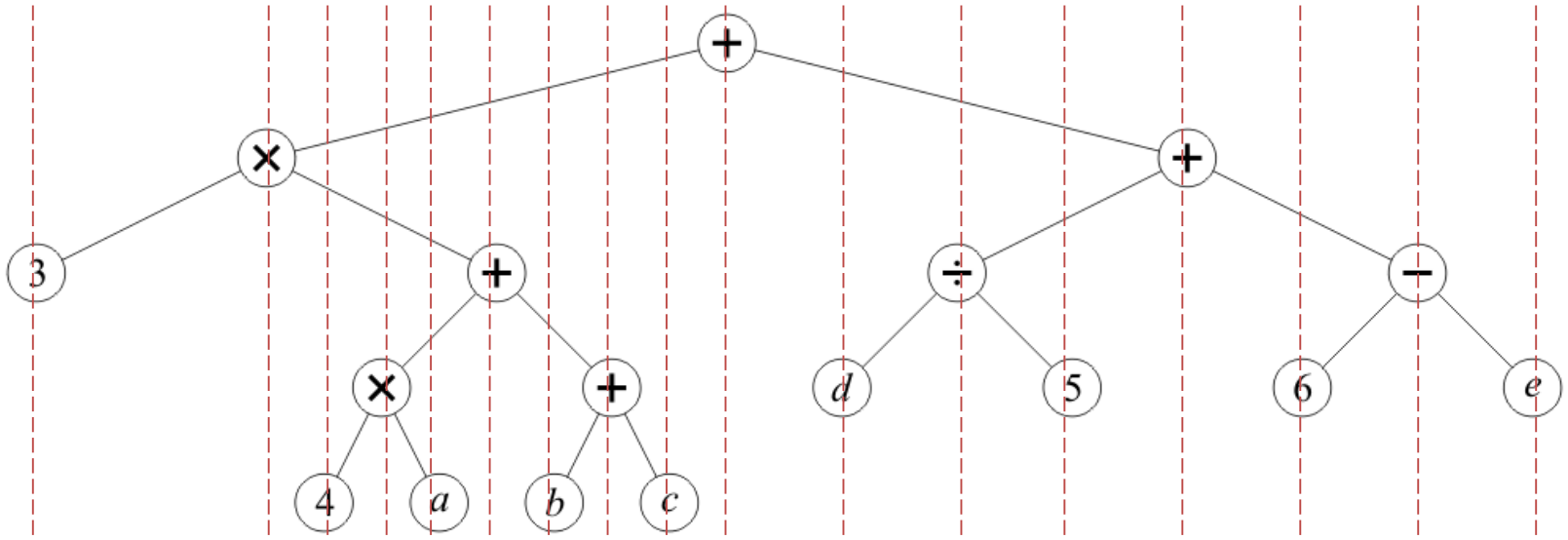
$$3 \times 4 \times a + b + c + d \div 5 + 6 - e$$

In-order Traversal



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

In-order Traversal



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

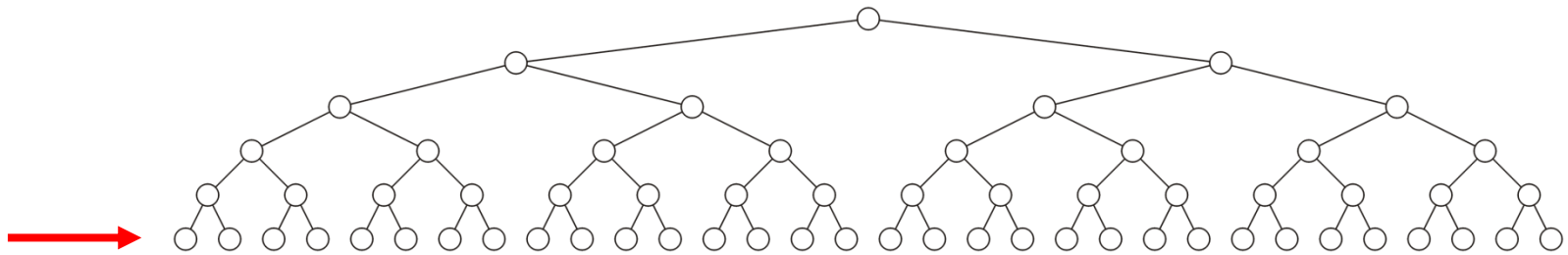
Outline

- Binary tree
- Perfect binary tree
- Complete binary tree
- Left-child right-sibling binary tree

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



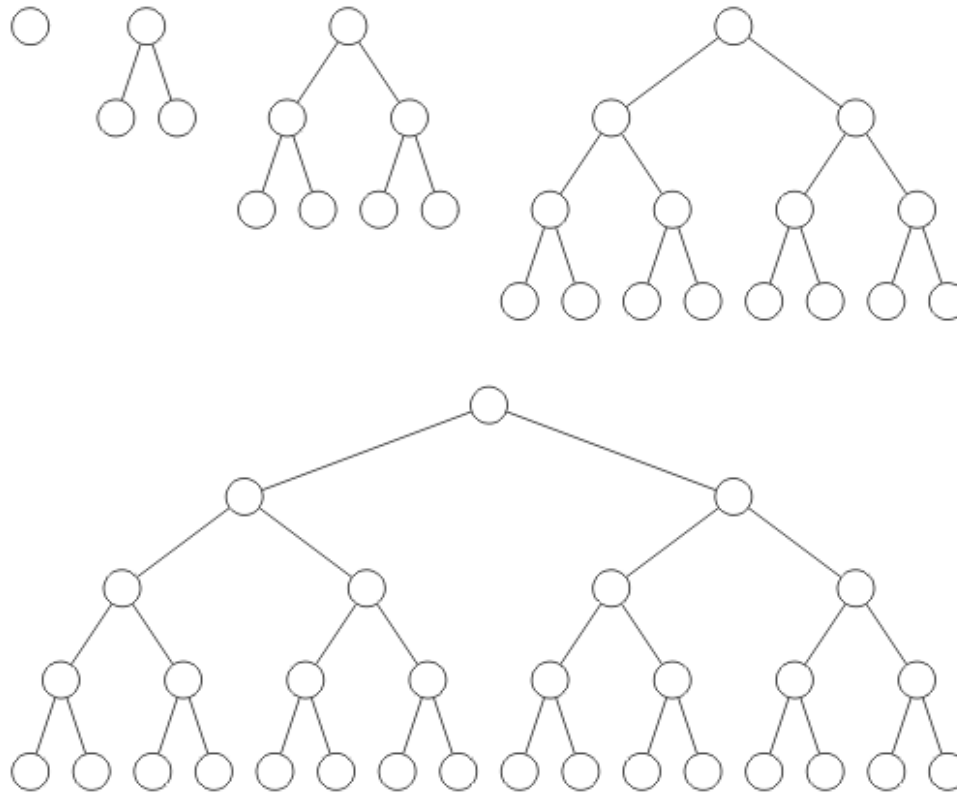
Definition

Recursive definition:

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

Examples

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



Examples

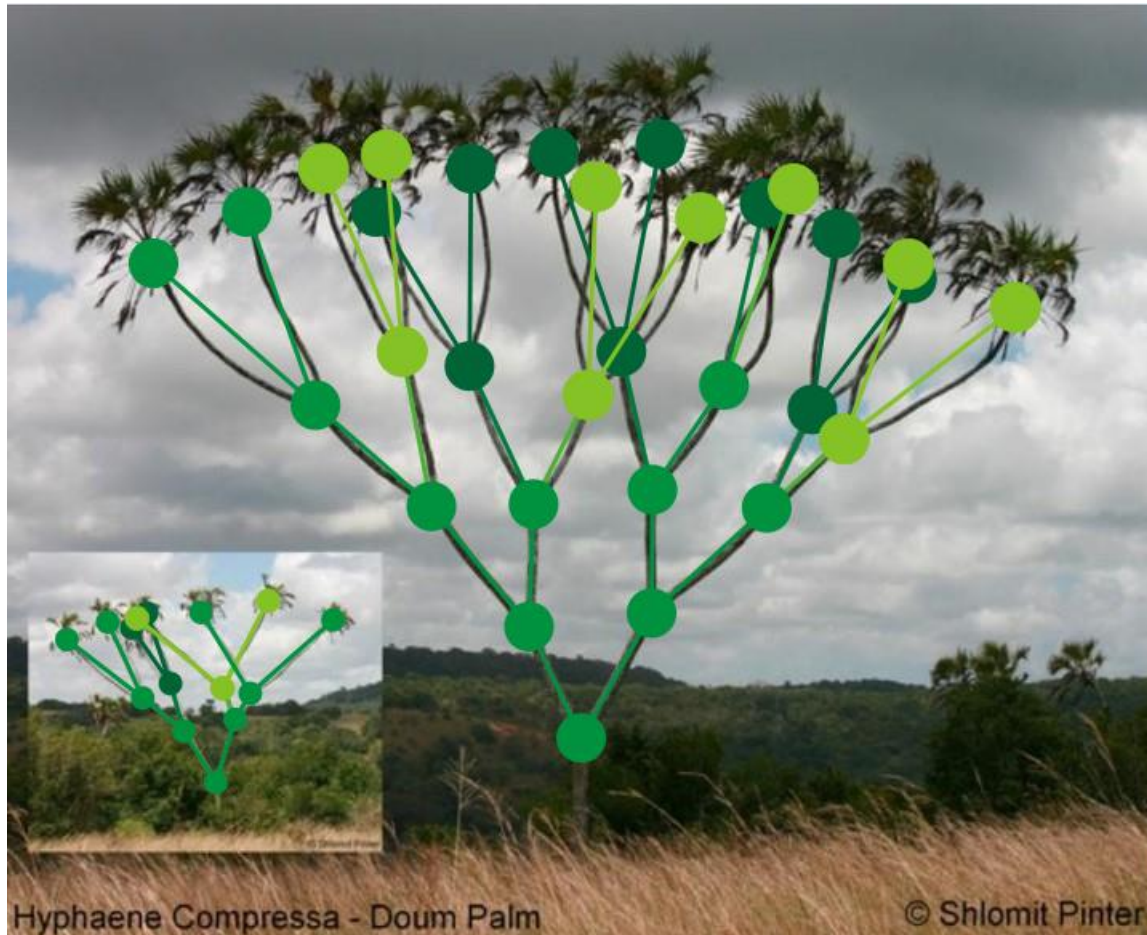
Perfect binary trees of height $h = 3$ and $h = 4$



Hyphaene Compressa - Doum Palm

Examples

Perfect binary trees of height $h = 3$ and $h = 4$



Theorems

Four theorems of perfect binary trees:

- A perfect binary tree of height h 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 $\Theta(\ln(n))$

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

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

Theorem

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

Proof:

We will use mathematical induction

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

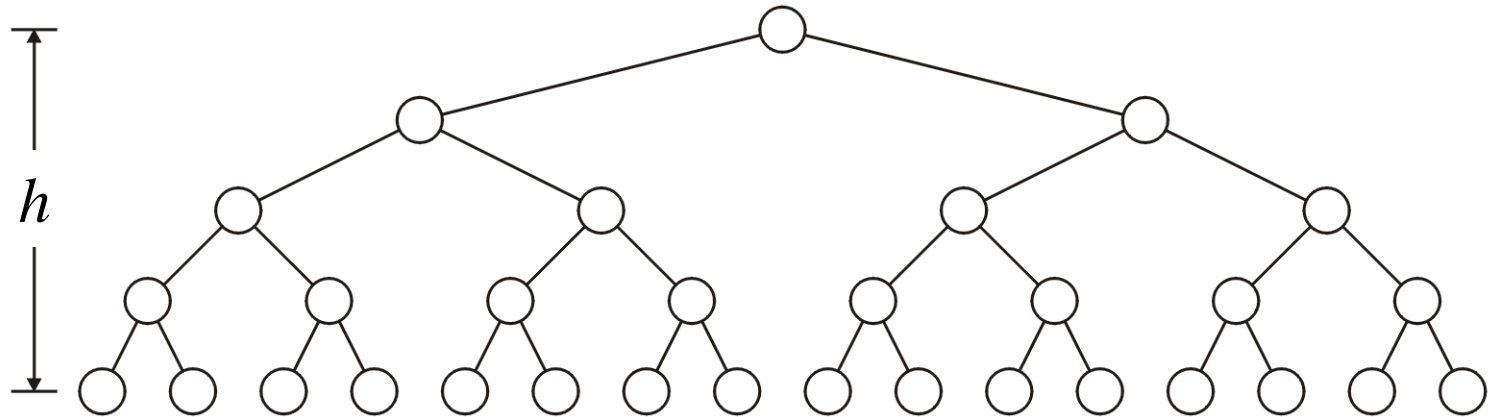
The base case:

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

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

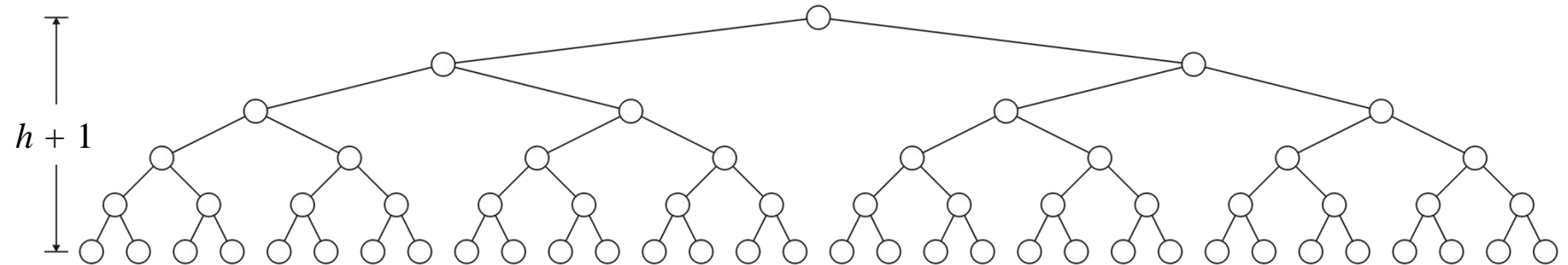
The inductive step:

- Assume that a tree of height h has $n = 2^{h+1} - 1$ nodes



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

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

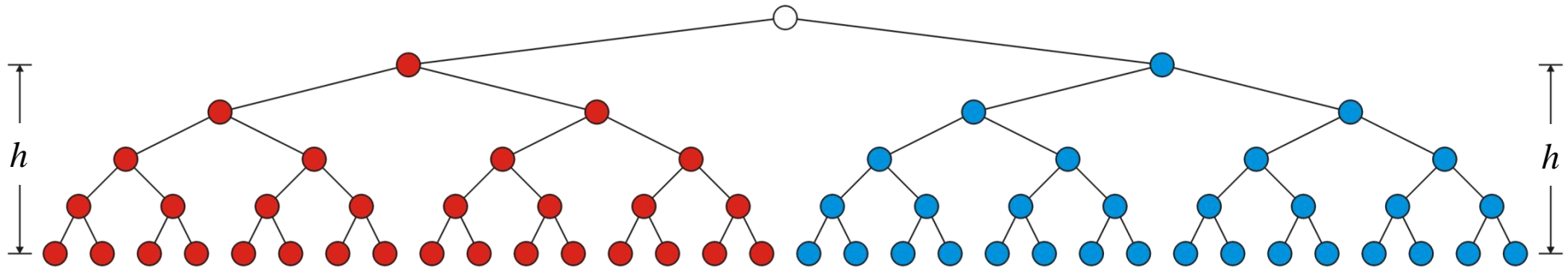


$2^{h+1} - 1$ Nodes

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

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



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

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.

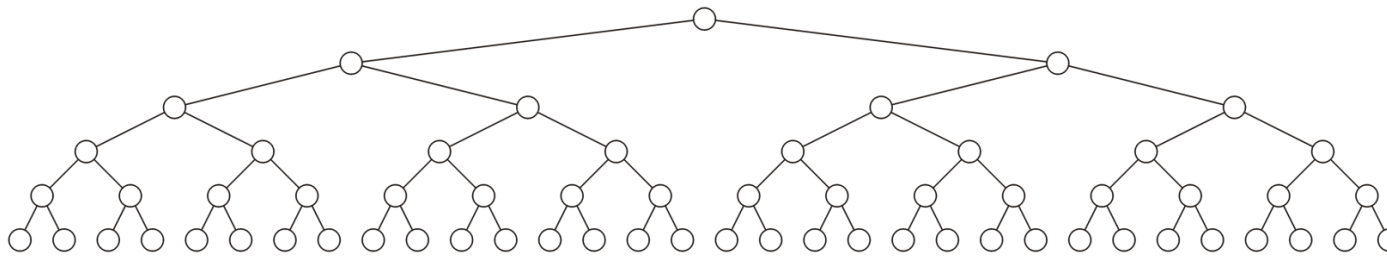
Assume that a perfect binary tree of height h has 2^h leaf nodes and observe that both sub-trees of a perfect binary tree of height $h + 1$ have 2^h leaf nodes.

Consequence: Over half of the nodes are leaf nodes:

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

The Average Depth of a 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

$$\sum_{k=0}^h k 2^k$$

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

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

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

Number of nodes

Outline

- Binary tree
- Perfect binary tree
- Complete binary tree
- Left-child right-sibling binary tree

Background

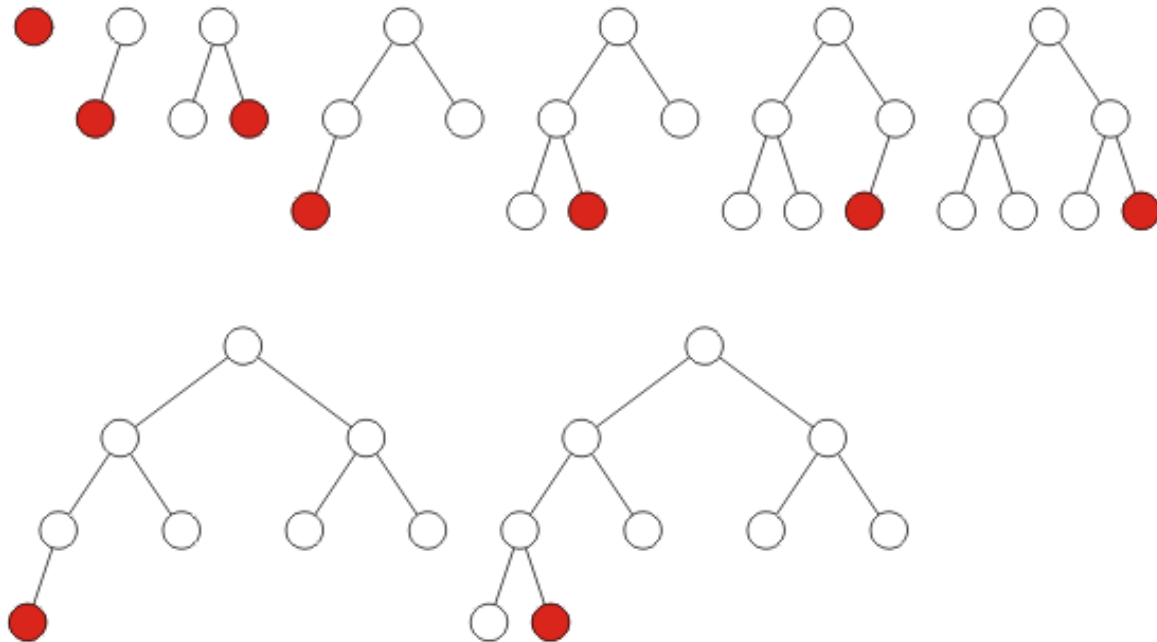
We require binary trees which are

- Similar to perfect binary trees, but
- Defined for any number of nodes

Definition

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

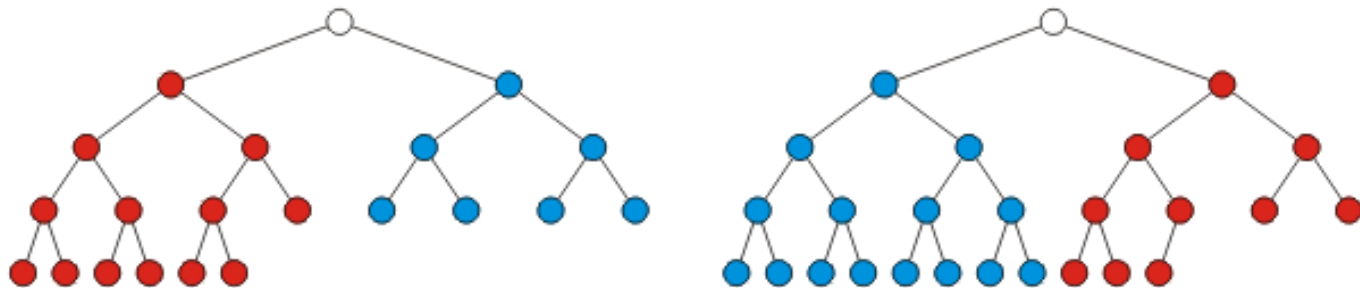
- Identical order to that of a breadth-first traversal



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 $h - 1$



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 with n nodes is perfect):

- If it is a perfect tree then
 - It had $n = 2^{h+1} - 1$ nodes
 - Adding one more node must increase the height
- So the tree with $n+1$ nodes has height $h+1$ and we have:

$$\lfloor \lg(n+1) \rfloor = \lfloor \lg(2^{h+1} - 1 + 1) \rfloor = \lfloor \lg(2^{h+1}) \rfloor = h+1$$

Height

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

- If it is not a perfect tree then

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

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

$$h < \lg(2^h + 1) \leq \lg(n + 1) < \lg(2^{h+1}) = h + 1$$

$$h \leq \lfloor \lg(2^h + 1) \rfloor \leq \lfloor \lg(n + 1) \rfloor < h + 1$$

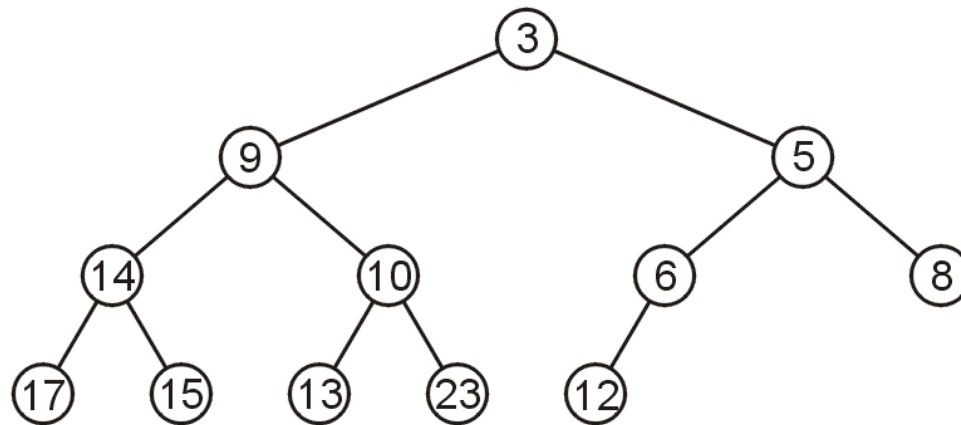
- So the tree with $n+1$ nodes has height h and we have $\lfloor \lg(n + 1) \rfloor = h$

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

Array storage

We are able to store a complete tree as an array

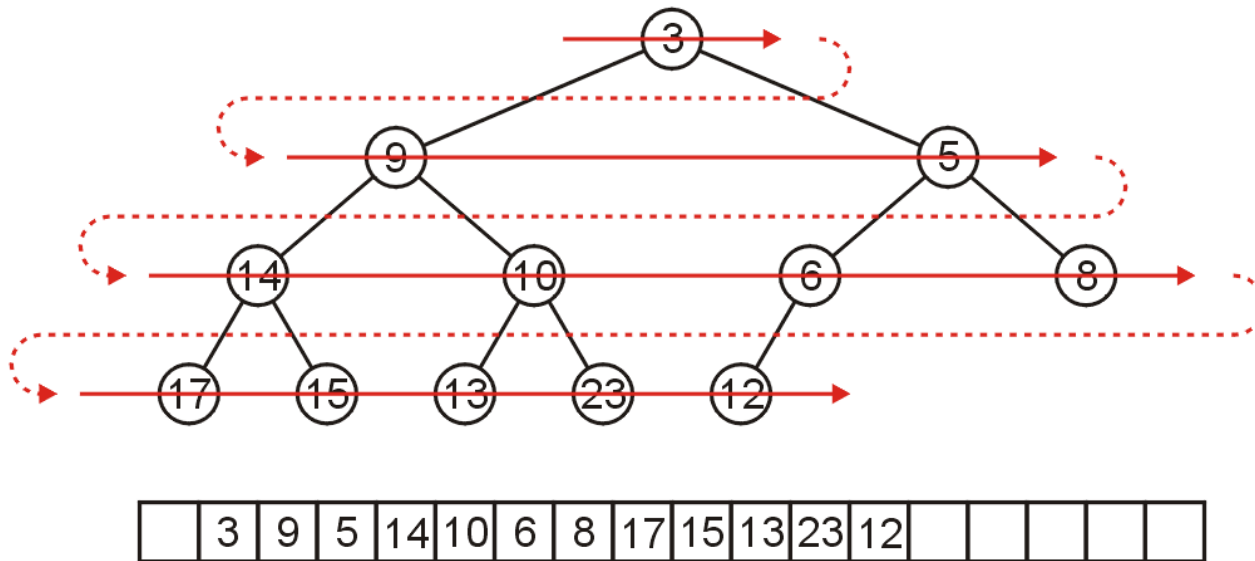
- Traverse the tree in breadth-first order, placing the entries into the array



Array storage

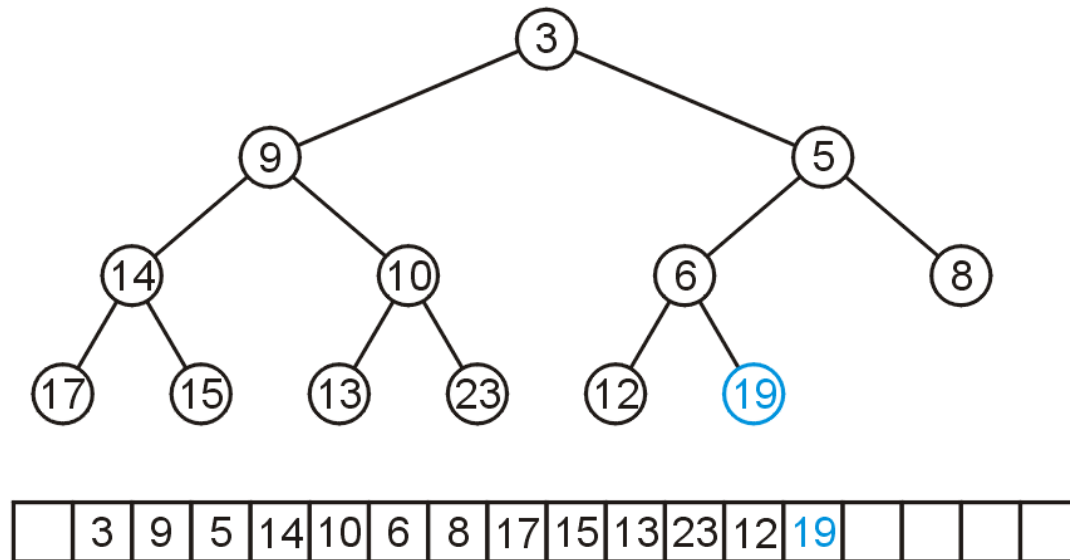
We are able to store a complete tree as an array

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



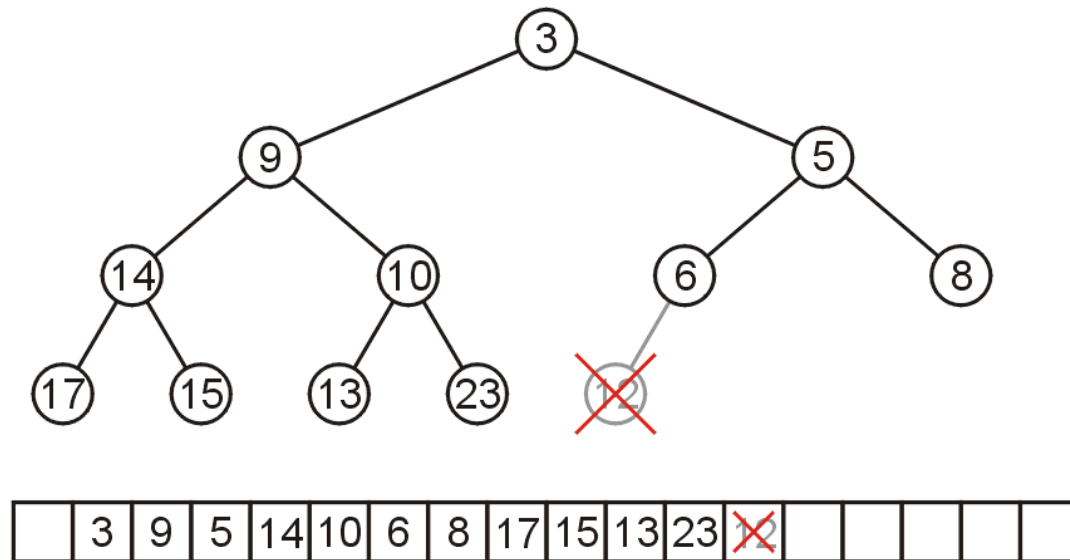
Array storage

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



Array storage

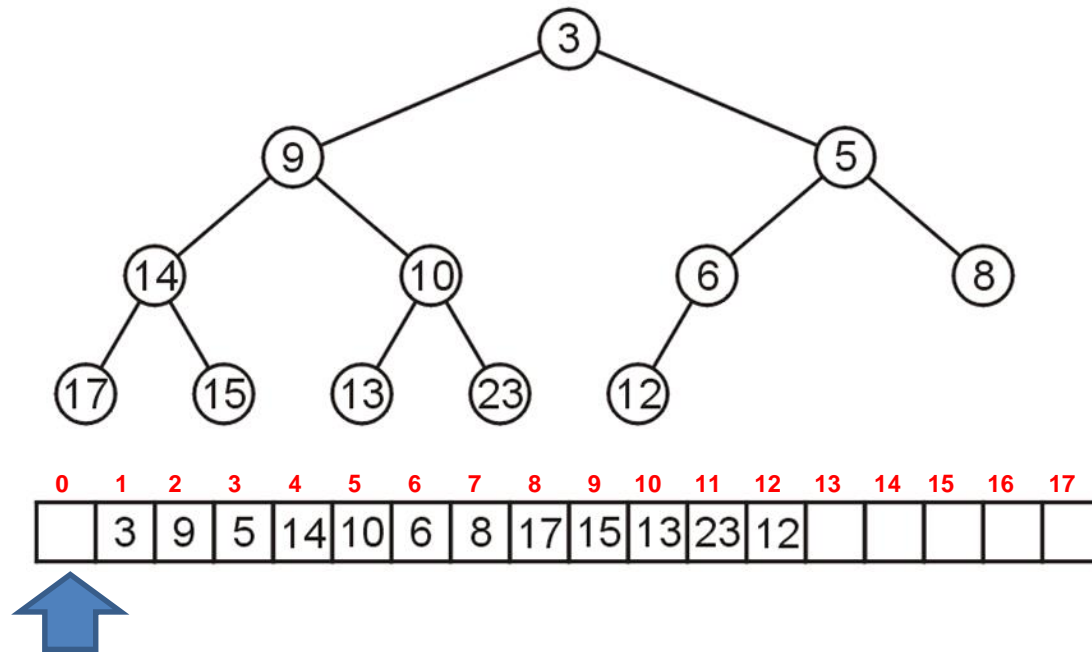
To remove a node while keeping the complete-tree structure, we must remove the last element in the array



Array storage

Leaving the first entry blank yields a bonus:

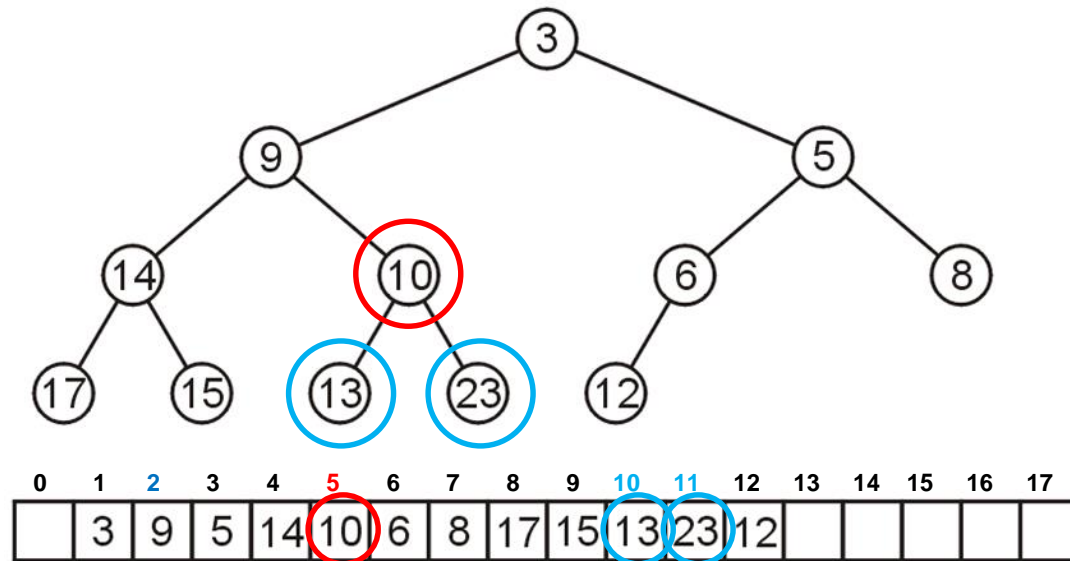
- The children of the node with index k are in $2k$ and $2k + 1$
- The parent of node with index k is in $k \div 2$



Array storage

For example, node 10 has index **5**:

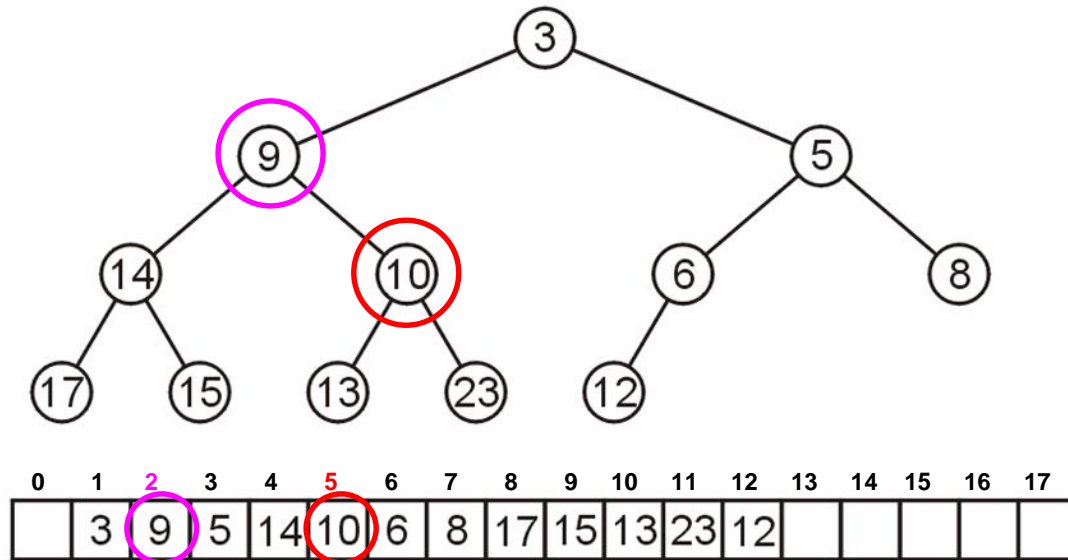
- Its children 13 and 23 have indices **10** and **11**, respectively



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 $5/2 = 2$



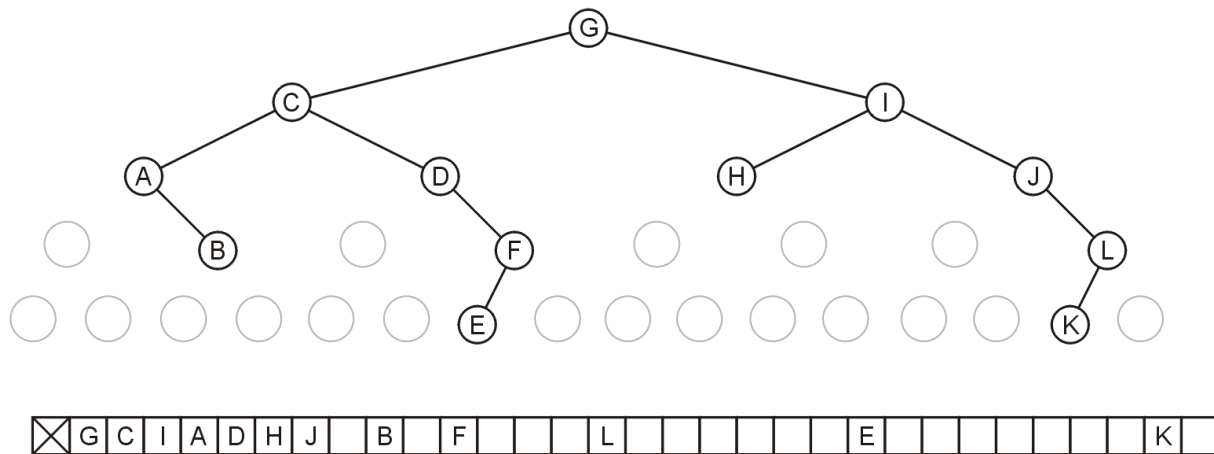
Array storage

Question: why not store any binary tree as an array in this way?

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

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

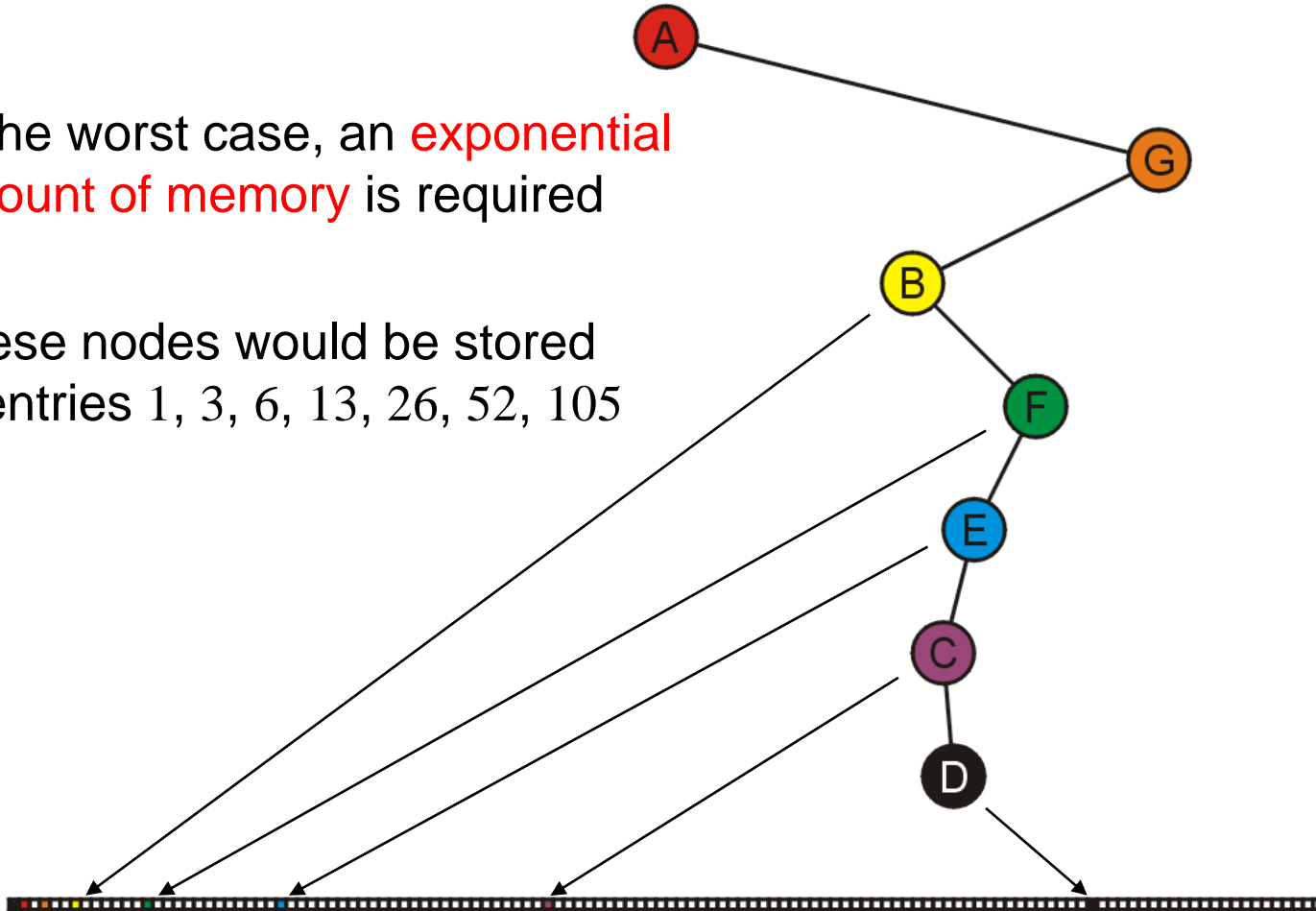
- Adding a child to node K doubles the required memory



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



Outline

- Binary tree
- Perfect binary tree
- Complete binary tree
- Left-child right-sibling binary tree

Background

Our simple tree data structure is node-based where children are stored as a linked list

- Is it possible to store a general tree as a binary tree?

The Idea

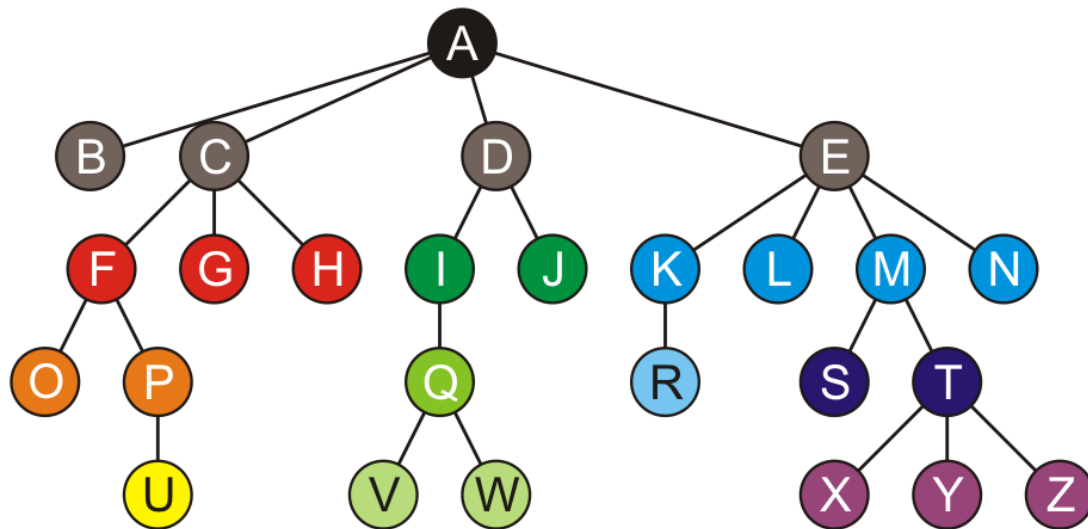
Consider the following:

- The first child of each node is its left sub-tree
- The next sibling of each node is in its right sub-tree

This is called a left-child—right-sibling binary tree

Example

Consider this general tree

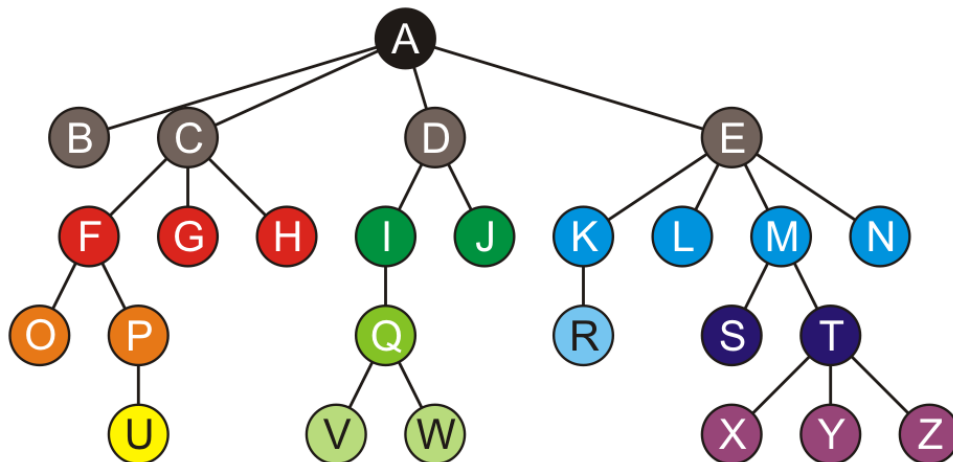
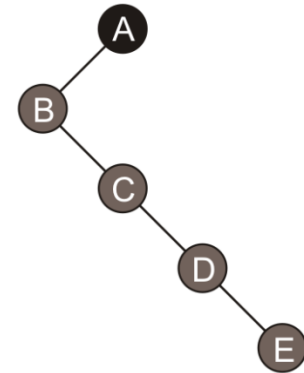


Example

B, the first child of A, is the left child of A

For the three siblings C, D, E:

- C is the right sub-tree of B
- D is the right sub-tree of C
- E is the right sub-tree of D



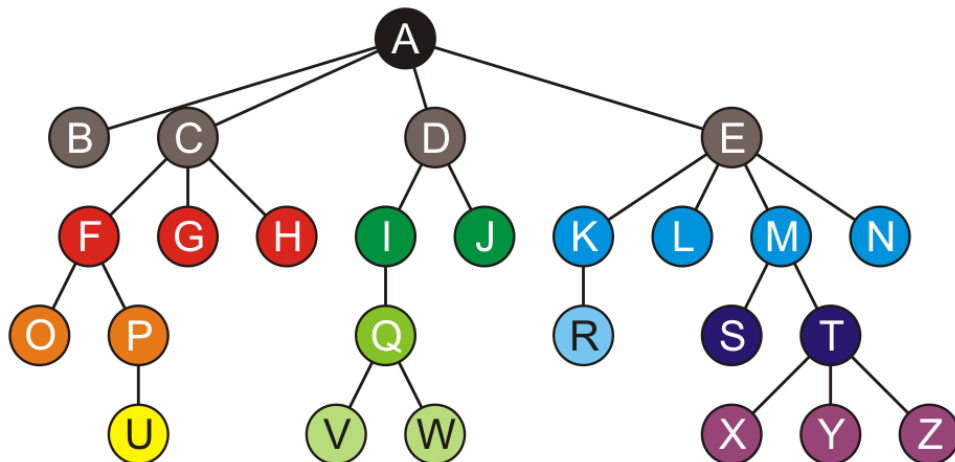
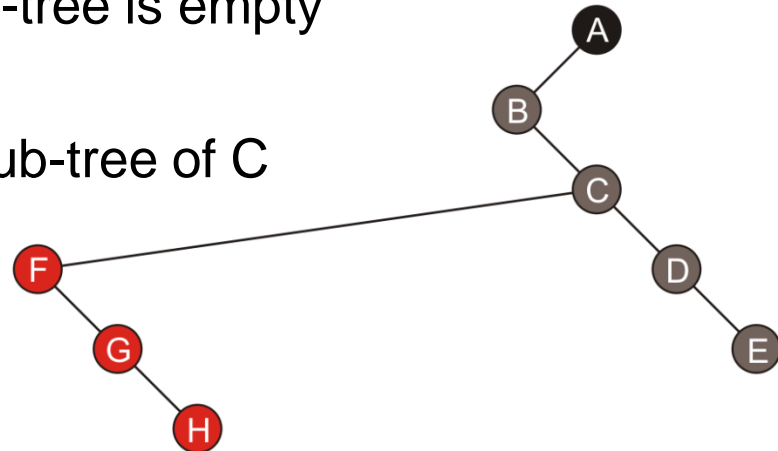
Example

B has no children, so its left sub-tree is empty

F, the first child of C, is the left sub-tree of C

For the next two siblings:

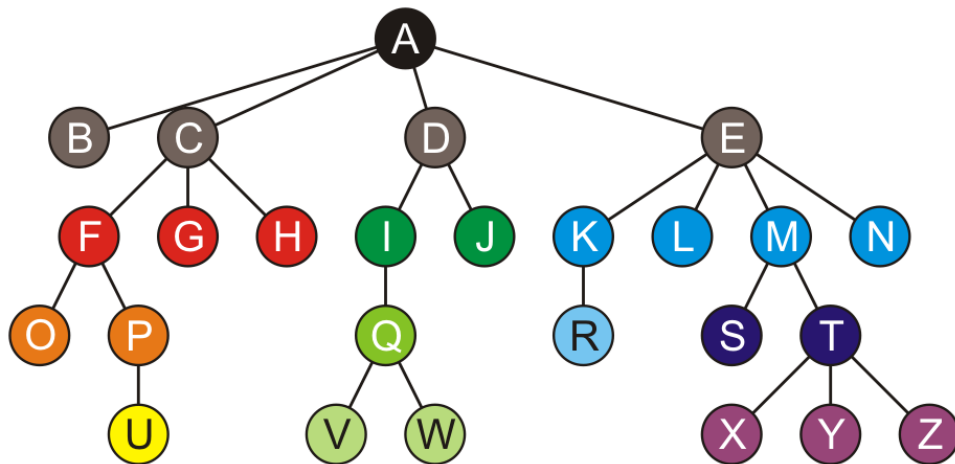
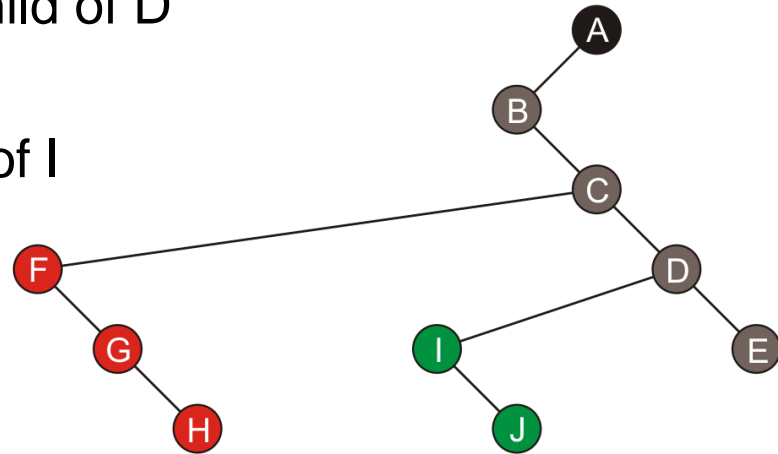
- G is the right sub-tree of F
- H is the right sub-tree of G



Example

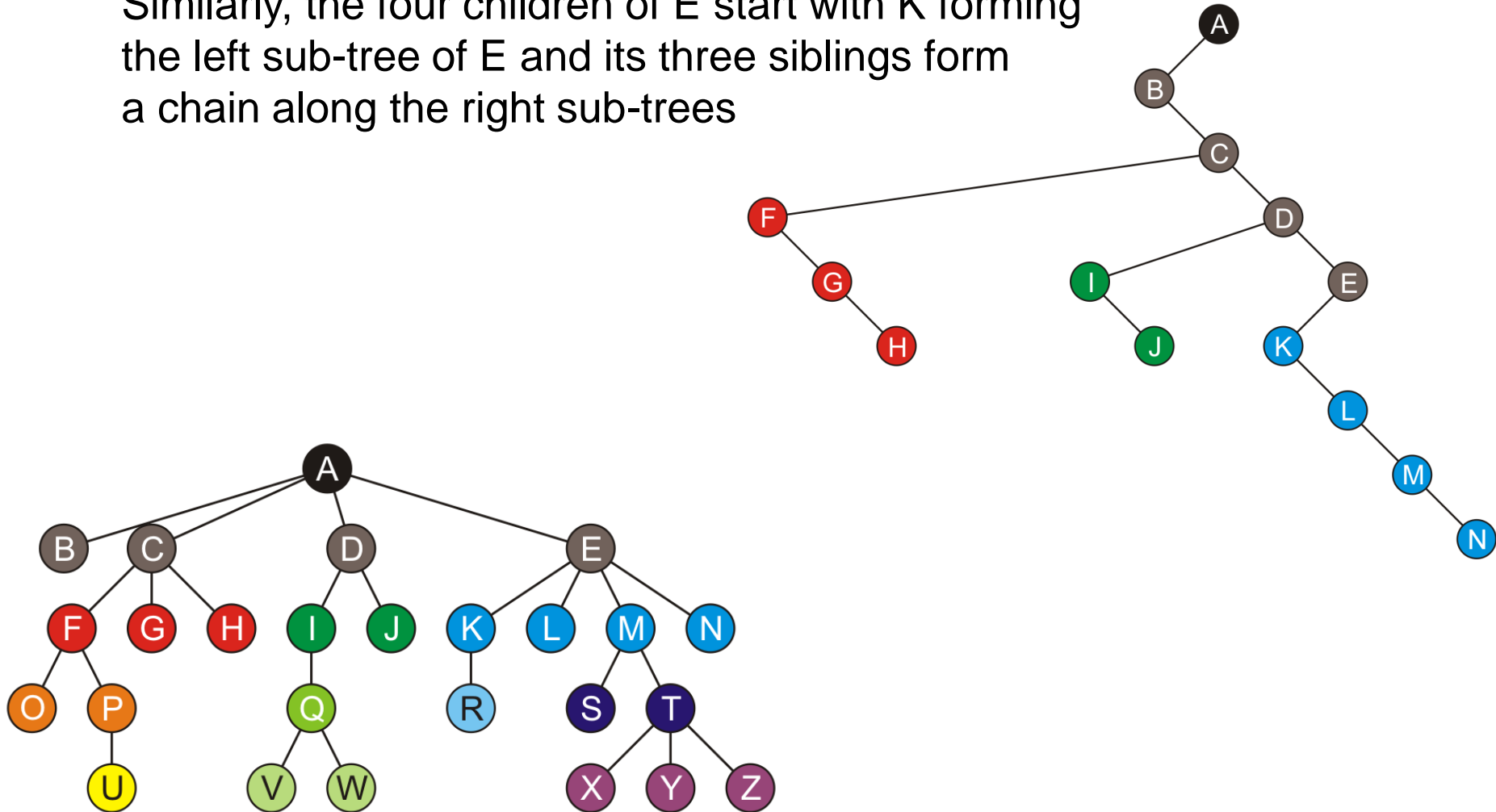
I, the first child of D, is the left child of D

Its sibling J is the right sub-tree of I

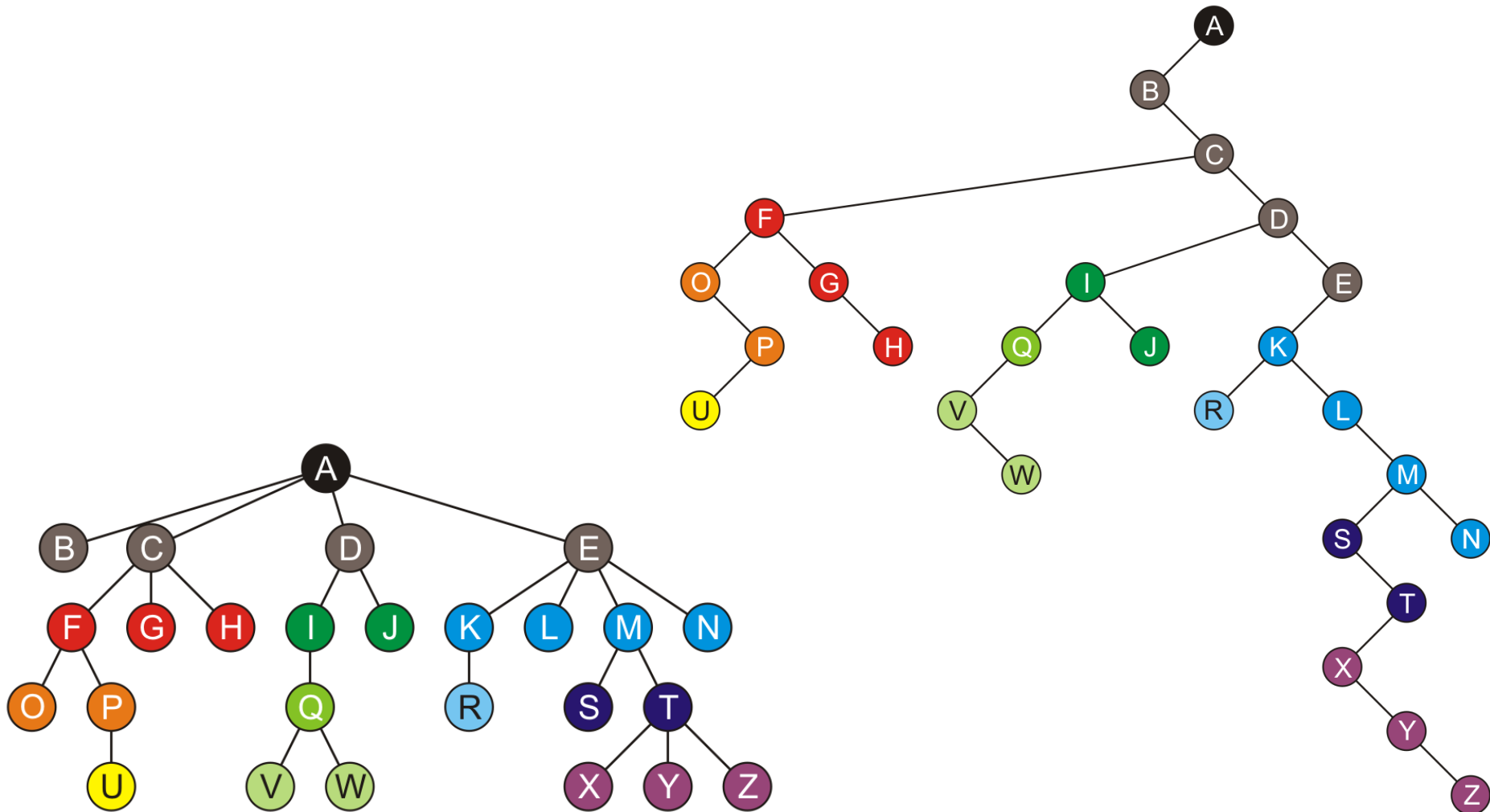


Example

Similarly, the four children of E start with K forming the left sub-tree of E and its three siblings form a chain along the right sub-trees

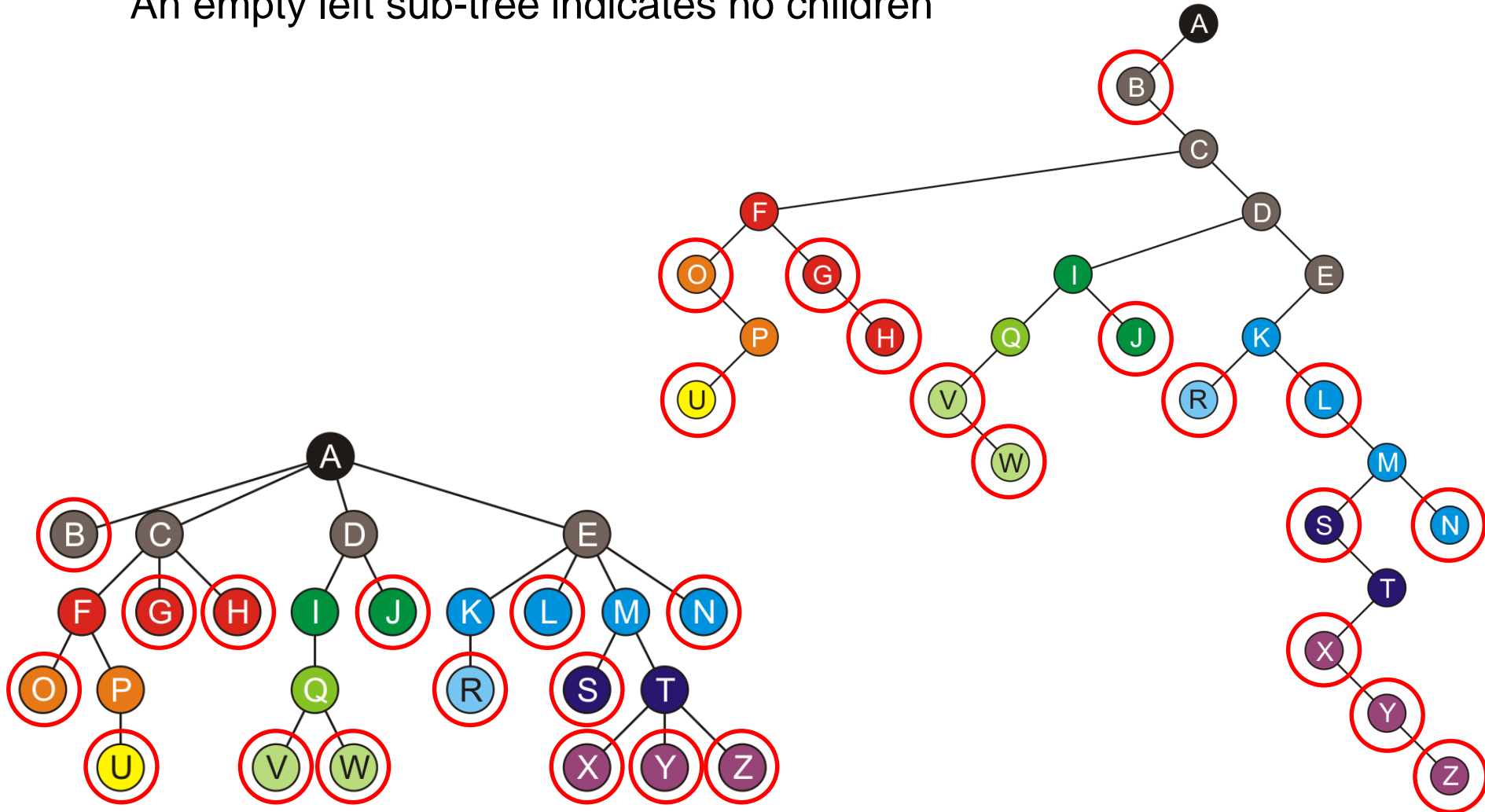


Example



Example

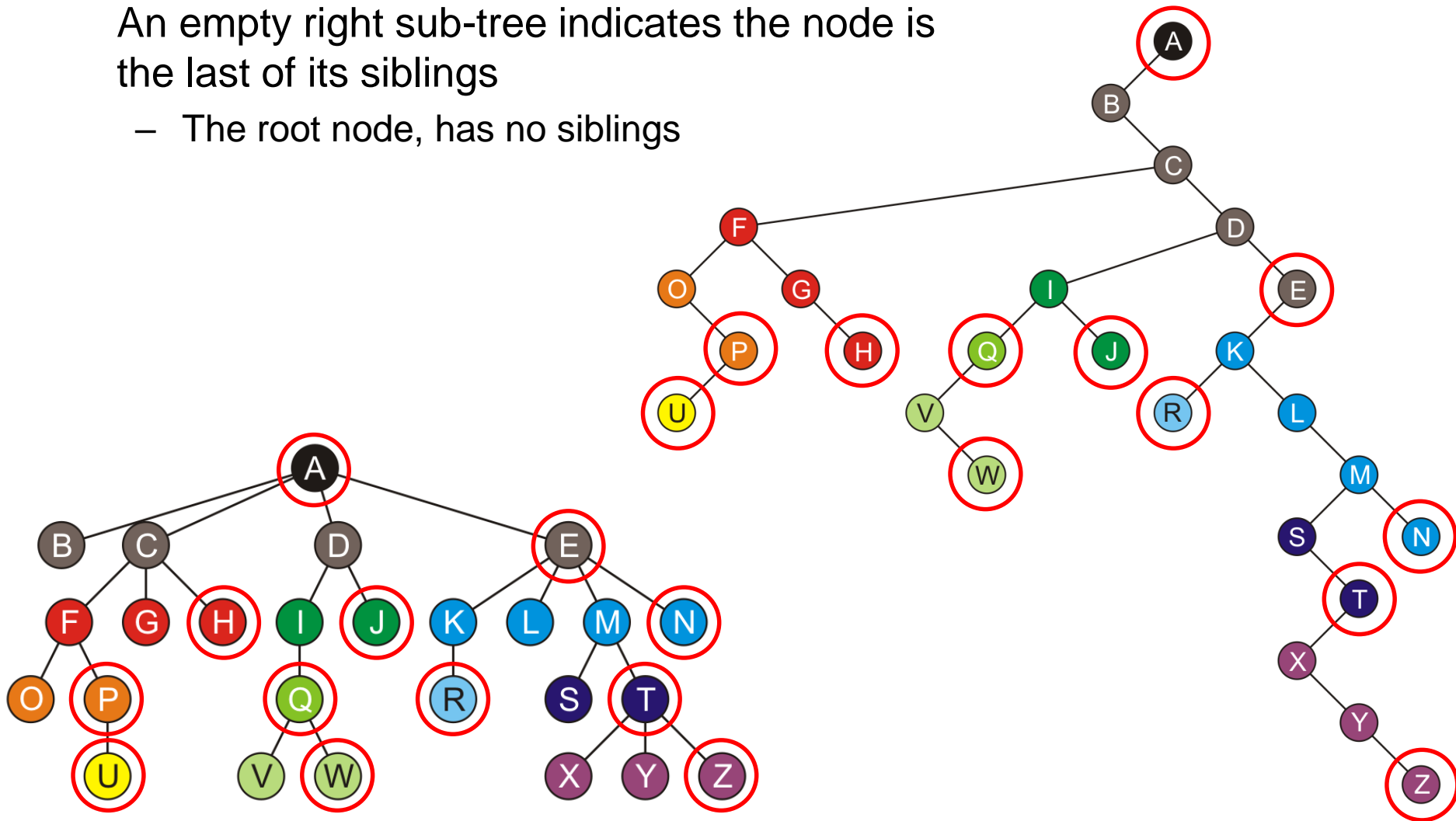
An empty left sub-tree indicates no children



Example

An empty right sub-tree indicates the node is the last of its siblings

- The root node, has no siblings



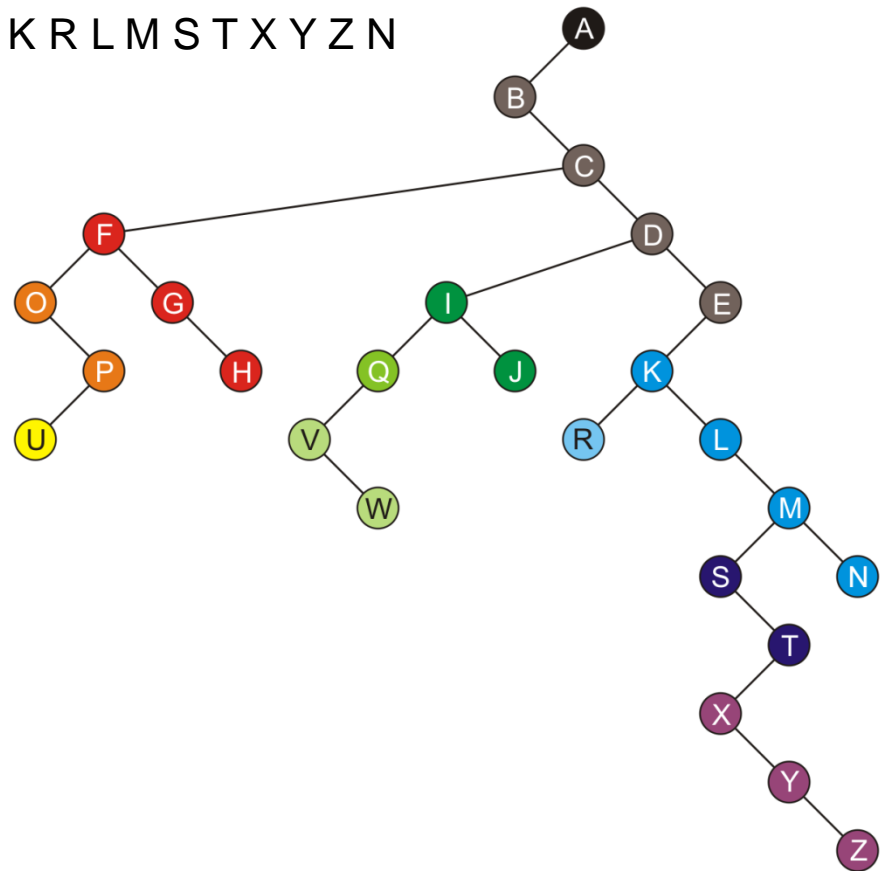
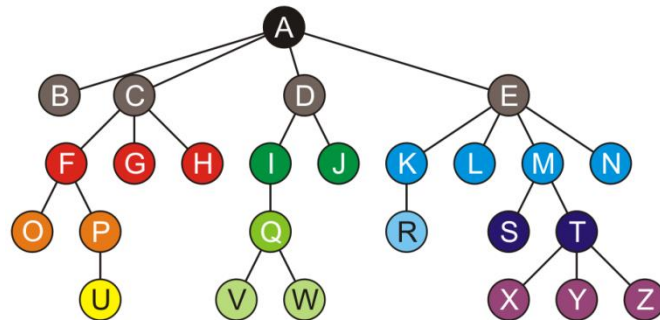
Transformation

The transformation of a general tree into a left-child right-sibling binary tree has been called the Knuth transform

Traversals

A **pre-order** traversal of the original tree is identical to the **pre-order** traversal of the Knuth transform

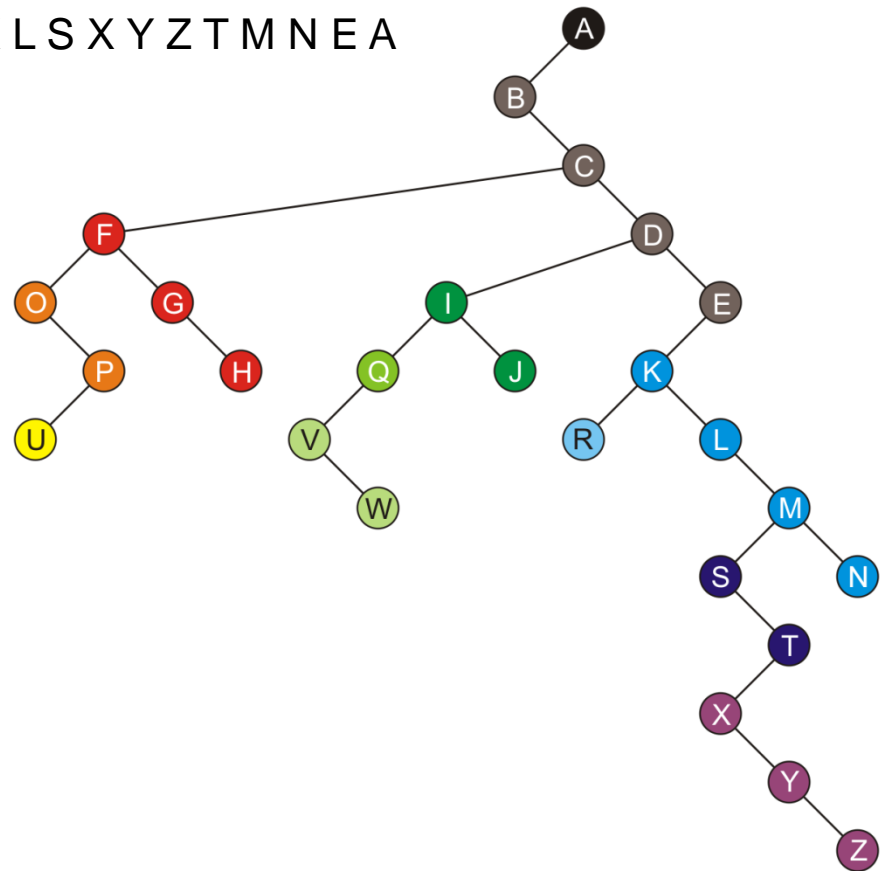
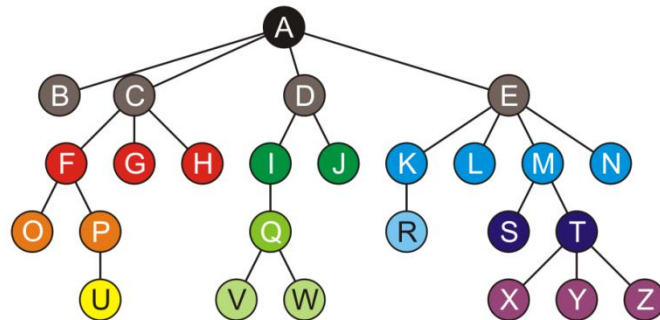
A B C F O P U G H D I Q V W J E K R L M S T X Y Z N



Traversals

A **post-order** traversal of the original tree is identical to the **in-order** traversal of the Knuth transform

BOUPFGHCVWQIJDRKLSXYZTMNEA



Summary

- Binary tree
 - Each node has two children
 - In-order traversal
- Perfect binary tree
 - Number of nodes, height, number of leaf nodes, average depth
- Complete binary tree
 - Height, array storage
- Left-child right-sibling binary tree