Data Structures

Week 14 – Trees: concepts, binary trees, representation, traversals

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- So far **linear** data structures: array, list, stack, queue – every element has at most **one** successor
- **Recursion** (week 12): reducing the problem to smaller instances
- **Stack** and **queue**: the tools of the traversals

Questions for today

- How do we store **hierarchical** data?
- What is a binary tree, and how can it be represented in memory?
- How do we visit every node of a tree?

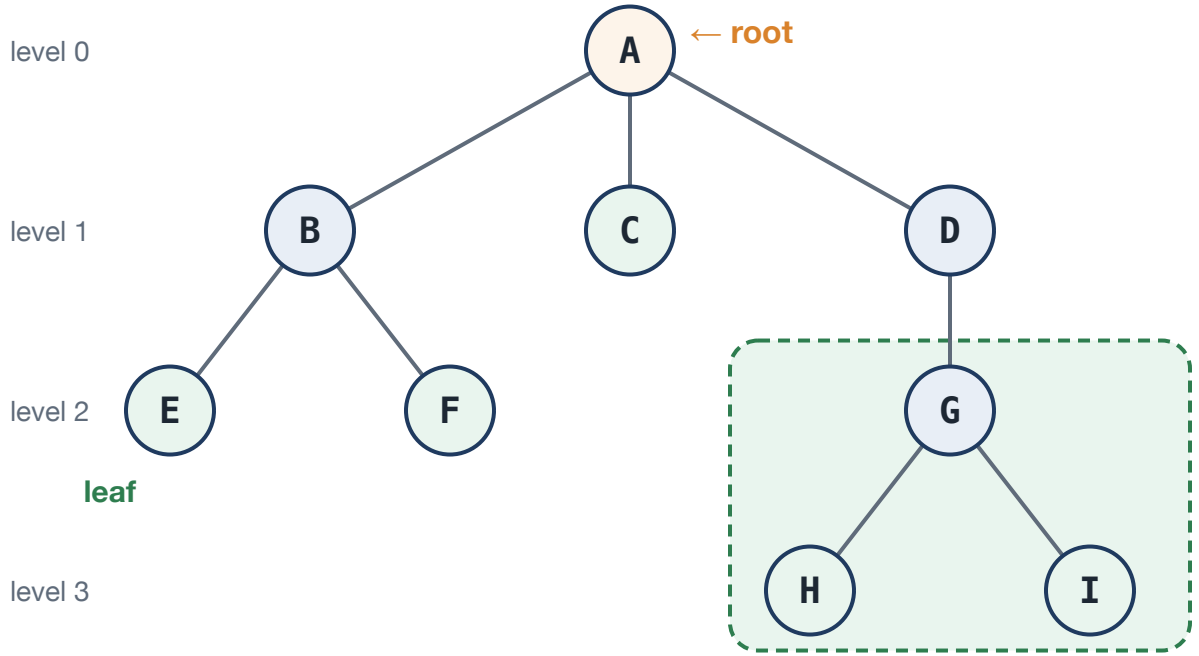
Hierarchical data

A lot of data is **tree-shaped** by nature:

- **File system:** folders, subfolders, files
- **Organization chart:** university → faculty → institute → department
- **Family tree**
- **Web page** (HTML DOM): `<html>` → `<body>` → `<div>` → ...
- **Expressions:** `(3 + 4) * (5 - 2)`
- **Decisions:** the move tree of a chess program, decision trees in machine learning

> A tree is a **hierarchical** data structure: every element has **at most one predecessor** (parent), but it can have **several successors** (children) (week 1: classification).

Concepts



parent of B: A · children of A: B, C, D

B, C, D are siblings

ancestors of H: G, D, A

degree of A: 3 · degree of G: 2

depth of H: 3

height of the tree: 3

leaves: C, E, F, H, I

Concepts – summary

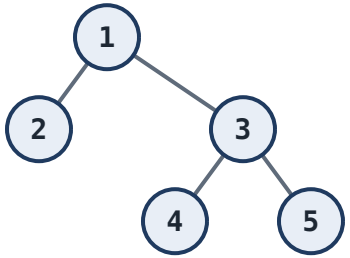
Concept	Meaning
node, **edge**	the elements, and the parent–child connections
root	the only node that has no parent
leaf	a node without children
parent, child, sibling	direct connection · children of the same parent
ancestor, descendant	nodes on the path towards the root, or downwards
subtree	a node and all of its descendants
depth / level	the length of the path from the root (the root's is 0)
height	the length of the longest root–leaf path
degree	the number of children · **forest:** a set of trees

Recursive definition: a tree is either **empty**, or consists of a **root** and its **subtrees**.

Binary trees

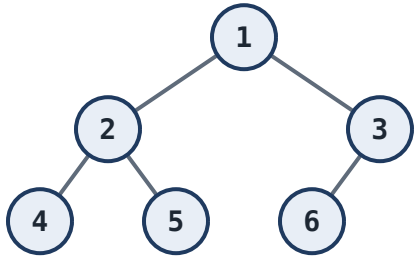
Binary tree

Binary tree: every node has **at most two** children: a **left** and a **right** child.



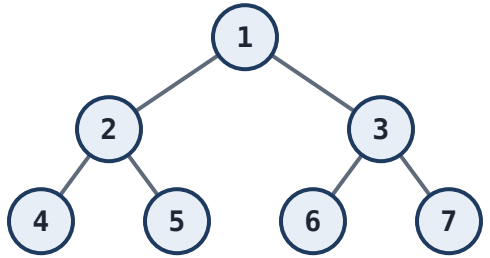
full (strict)

0 or 2 children



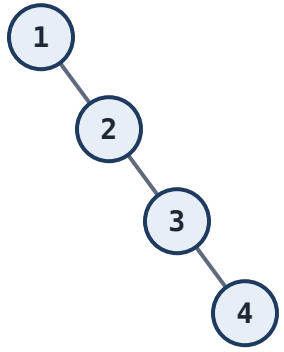
complete

filled level by level, left to right



perfect

every level full



degenerate

actually a list

Properties of the binary tree

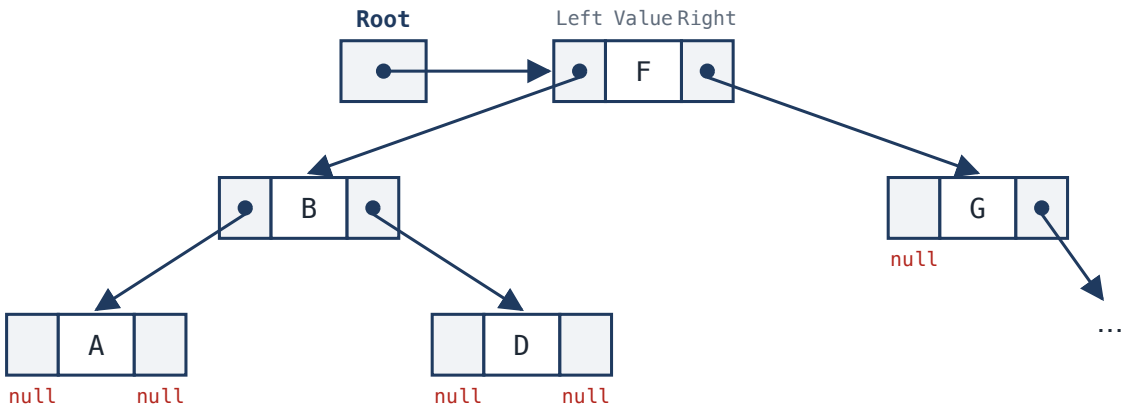
- Level **i** has at most 2^i nodes
- A binary tree of height **h** has at most $2^{h+1} - 1$ nodes (perfect tree)
- The height of a binary tree with **n** nodes is at least $\lfloor \log_2 n \rfloor$, at most $n - 1$ (degenerate)

Property	Binary tree
Elements	homogeneous
Relationship	**hierarchical**
Size	dynamic
Storage	scattered (linked) or contiguous (array)
Access	**sequential** (starting from the root, through pointers)

The height determines the **number of steps** of the tree operations – that is why it matters that the tree is **balanced** (week 15).

Representing a binary tree

Linked representation



Java

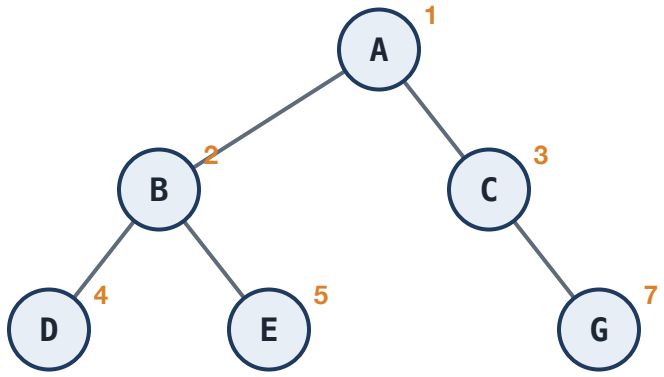
```
class Node {
    char value;
    Node left, right;
}
Node root;   // null: empty tree
```

Go

```
type Node struct {
    Value       rune
    Left, Right *Node
}
```

Pseudocode: `Elem[P].Value`, `Elem[P].Left`, `Elem[P].Right`, the **Root** pointer points to the root.

Array (contiguous) representation



`A[1..7]`

A	B	C	D	E		G
1	2	3	4	5	6	7

```
left child of i: 2·i
right child of i: 2·i + 1
parent of i: i div 2
```

empty slot if the tree is not complete (index 6)

- Without pointers: the indexes of the children and of the parent **can be computed**
- Perfect for a **complete** tree (no empty slots) – this is what the **heap** uses (week 15)
- A lot of empty space for a degenerate tree: a chain of height **h** needs an array of $2^{h+1} - 1$ elements!

Representing a general tree with a binary tree

If a node can have **any** number of children: **left child – right sibling** representation

- **Left** pointer → the **first child**
- **Right** pointer → the **next sibling**



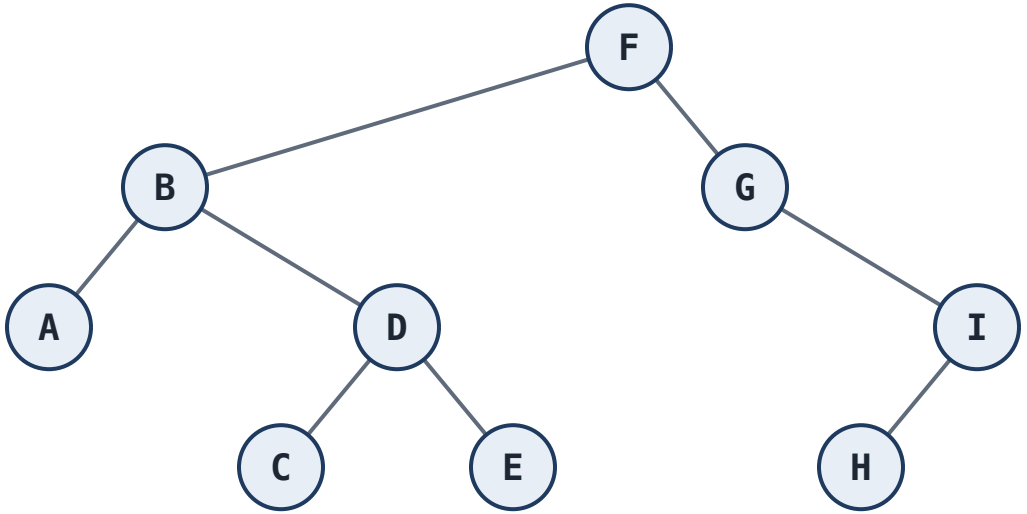


This way every general tree (and forest) can be stored as a **binary tree** – for example the folder structure of a file system.

Traversals

Traversing a tree

Traversal: processing every node **exactly once**, in a given order.



Preorder (root, left, right):

F B A D C E G I H

Inorder (left, root, right):

A B C D E F G H I

Postorder (left, right, root):

A C E D B H I G F

Level order (breadth-first):

F B G A D I C E H

Depth-first traversals – recursively

Pseudocode

```
Procedure Preorder(P)
  If P ≠ Nil then
    Process(Elem[P].Value)
    Preorder(Elem[P].Left)
    Preorder(Elem[P].Right)
  End if
End procedure

Procedure Inorder(P)
  If P ≠ Nil then
    Inorder(Elem[P].Left)
    Process(Elem[P].Value)
    Inorder(Elem[P].Right)
  End if
End procedure

Procedure Postorder(P)
  If P ≠ Nil then
    Postorder(Elem[P].Left)
    Postorder(Elem[P].Right)
    Process(Elem[P].Value)
  End if
End procedure
```

Java

```java
static void inorder(Node p) {
    if (p == null) return;
    inorder(p.left);
    System.out.print(p.value + " ");
    inorder(p.right);
}
```

Go

```go
func inorder(p *Node) {
    if p == nil {
        return
    }
    inorder(p.Left)
    fmt.Printf("%c ", p.Value)
    inorder(p.Right)
}
```

The three traversals differ **only** in **when** the root is processed. Steps: **O(n)**, space: O(h).

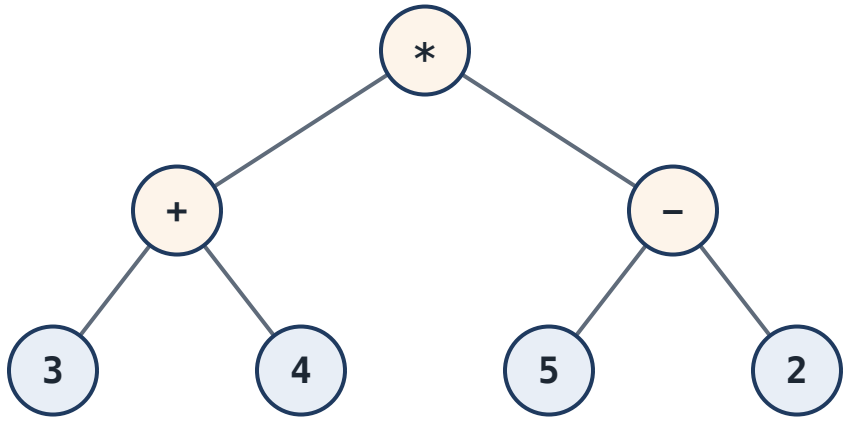
Level-order (breadth-first) traversal

Starting from the root, **level by level**, from left to right – with a **queue** (week 8):

```
Procedure LevelOrder(Root)
  If Root = Nil then exit
  Enqueue(Root)
  While not IsEmpty
    P ← Dequeue
    Process(Elem[P].Value)
    If Elem[P].Left ≠ Nil then Enqueue(Elem[P].Left)
    If Elem[P].Right ≠ Nil then Enqueue(Elem[P].Right)
  End while
End procedure
```

Example: **F B G A D I C E H** · If we replace the queue with a **stack** (and push the right child first), we get the **preorder** traversal!

Application: expression tree



inorder (with parentheses):
`(3 + 4) * (5 - 2)`

preorder (prefix):
`* + 3 4 - 5 2`

postorder (postfix, week 7):
`3 4 + 5 2 - *`

value: 7 · 3 = 21

- The internal nodes are **operators**, the leaves are **operands**
- The **postorder** traversal gives exactly the **postfix** form – the tree can be built from the postfix form with a stack
- **Evaluation:** `Value(P) = Value(left) op Value(right)` – recursively

Recursive operations on trees

Pattern: **“handle the root, and call yourself for the two subtrees”**

```
Function Count(P)
  If P = Nil then Return 0
  Return 1 + Count(Elem[P].Left)
           + Count(Elem[P].Right)
End function

Function Height(P)
  If P = Nil then Return -1
  Return 1 + max(Height(Elem[P].Left),
                 Height(Elem[P].Right))
End function
```

```
Function Leaves(P)
  If P = Nil then Return 0
  If Elem[P].Left = Nil and Elem[P].Right = Nil
    then Return 1
  Return Leaves(Elem[P].Left)
       + Leaves(Elem[P].Right)
End function
```

Further examples: sum, maximum, search (O(n)), mirroring, copying, comparing two trees

For the example tree: 9 nodes, 4 leaves (A, C, E, H), height 3.

Summary

- **Tree:** hierarchical data structure; root, leaf, parent, child, subtree, depth, height
- **Binary tree:** at most two children; full, complete, perfect, degenerate
- **Linked representation:** Left and Right pointers · **array:** children 2i and 2i + 1, parent i div 2
- **Traversals:** preorder (N-L-R), inorder (L-N-R), postorder (L-R-N) – recursively, O(n); level order – with a queue
- Operations on trees are **naturally recursive**

Next week: binary search tree – search, insertion, deletion; balance; heap.

Review questions

1. True or false: the binary tree is a hierarchical data structure.
2. Give the traversal orders: root 1; the left child of 1 is 2, its right child is 3; the children of 2 are 4 and 5; the right child of 3 is 6.
3. The elements of a binary tree are stored in an array A (A[1] is the root, the children of the **i**-th element are A[2i] and A[2i + 1]). A = [K, F, R, B, H, –, T]. Draw the tree and give the inorder traversal!
4. At most how many nodes can a binary tree of height 4 have?
5. Which traversal gives the postfix form of an expression tree?
6. What is the value of `Height(P)` for a tree consisting of a single node?

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Binary tree	`BinaryTree.java`	`w14/binarytree`	four traversals, count, leaves, height, sideways printing
Array tree	`ArrayTree.java`	`w14/arraytree`	index formulas, traversal with indexes
Expression tree	`ExpressionTree.java`	`w14/expressiontree`	building from postfix, evaluation, infix/prefix
Traversal with a stack	`IterativeTraversal.java`	`w14/iterative`	preorder and inorder without recursion

✎ Exercises

1. **Sum and maximum:** compute the sum and the largest element of a binary tree of integers!
2. **Nodes with one child:** how many nodes have exactly one child?
3. **Full tree:** decide whether every node has 0 or 2 children!
4. **Level k:** print the elements on level **k**!
5. **Paths:** print all root–leaf paths!
6. **Identical trees:** decide whether two binary trees are identical both in structure and in values!

Solutions: `Exercises14.java`, `w14/exercises`

✎ Homework

1. Write a function that converts a binary tree with linked representation into the **array** representation (and vice versa)! How large an array is needed?
2. Build the binary tree from its **preorder** and **inorder** traversals! (E.g. preorder: F B A D C E G I H, inorder: A B C D E F G H I)
3. Print the tree **level by level, one line per level** (every level on a new line)!
4. Write a program that prints the content of a folder as a **tree** (like the `tree` command)! (Java: `File.listFiles()`, Go: `os.ReadDir`)