Data Structures and Algorithms

XMUT-COMP 103 - 2026 T1

Decision trees

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

Announcement

The result of Test 1 is out. Please check your result. If you have any questions about it, please ask.

The test's solution will be discussed either next week (Week 10) or the week after (Week 11).

- Assignment 4 is out.
- Due date is the 23rd of May 2026

Tree

A tree is an abstract Data Type (ADT) that stores elements in a hierarchical structure.

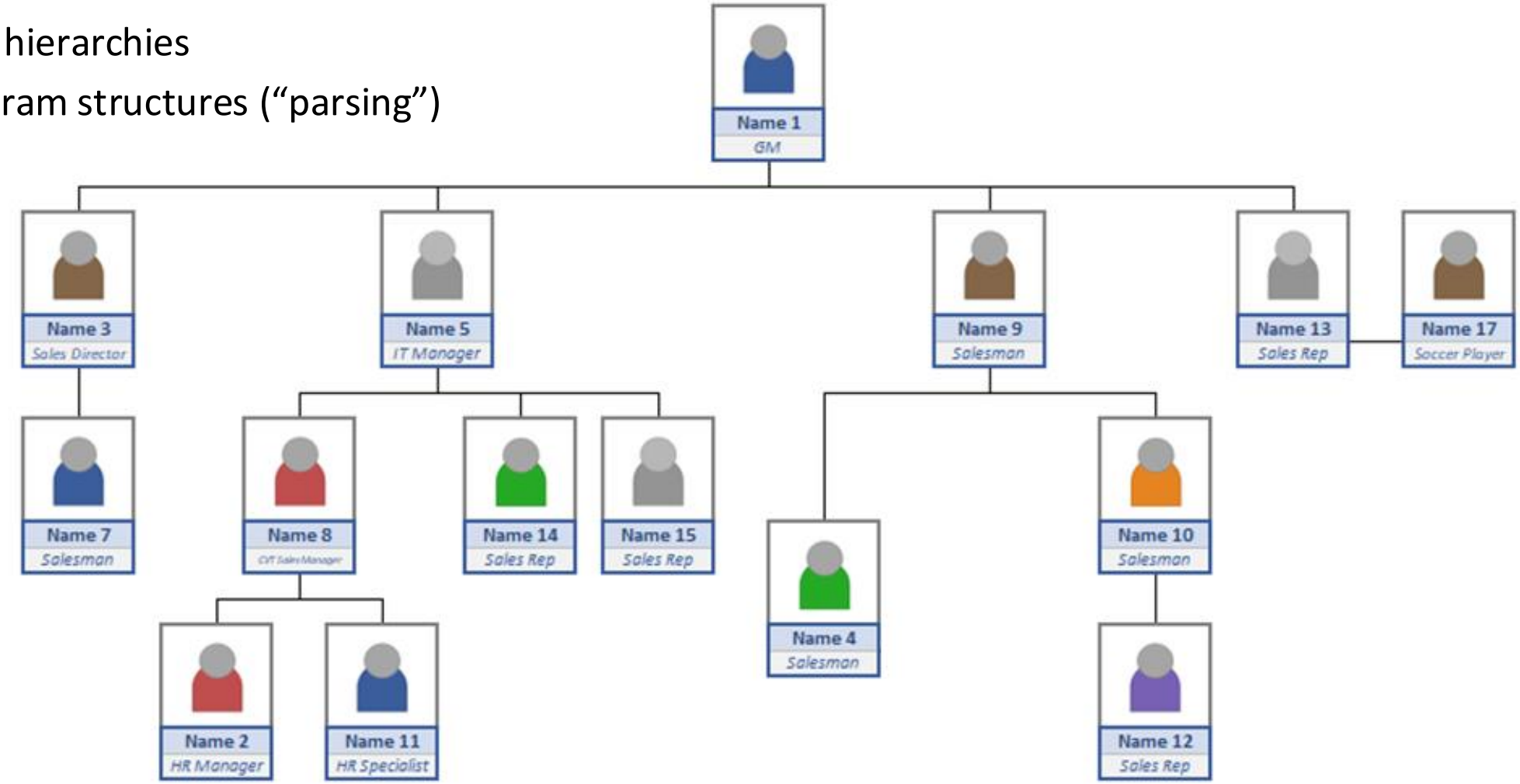
Each element in a tree has a *parent, and zero or more child* elements, except for the top element.

The tree is visualised by placing elements inside ovals or rectangles, and by drawing the connections between parents and children with straight lines.

The top element is usually referred to as the *root* element

Tree Structured Data

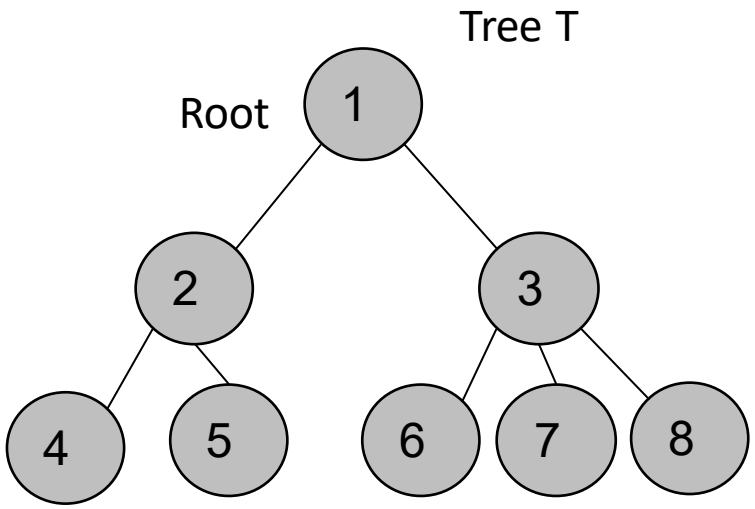
- Examples:
  - genealogy
  - organisational hierarchies
  - language/program structures ("parsing")
  - decision trees



Tree: Formal Definition

tree T is a set of *nodes* storing elements, such that the nodes have a parent-child relationship that satisfies the following:

- If *T* is nonempty, it has a special *node*, called the *root* of *T*, that has no parent.
- Each node *v* of *T* different from the root has a unique parent node *w*; every node with parent *w* is a child of *w*.



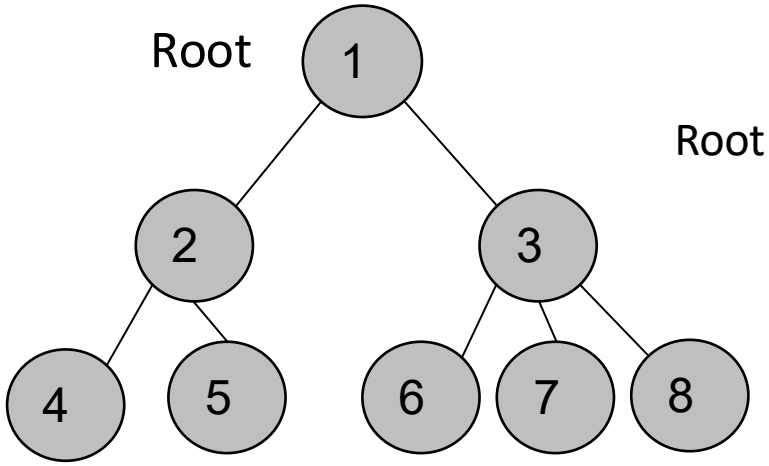
Root has two children (v): [2,4]

Node 2 has a parent (w): root

Tree: Other Relationship

Two nodes that are children of the same parent are *siblings*

Tree T



Root has two children (v): [2,4]

Node 2 and node 3 are *siblings*, that has a parent (w): root

Tree: Other Relationship

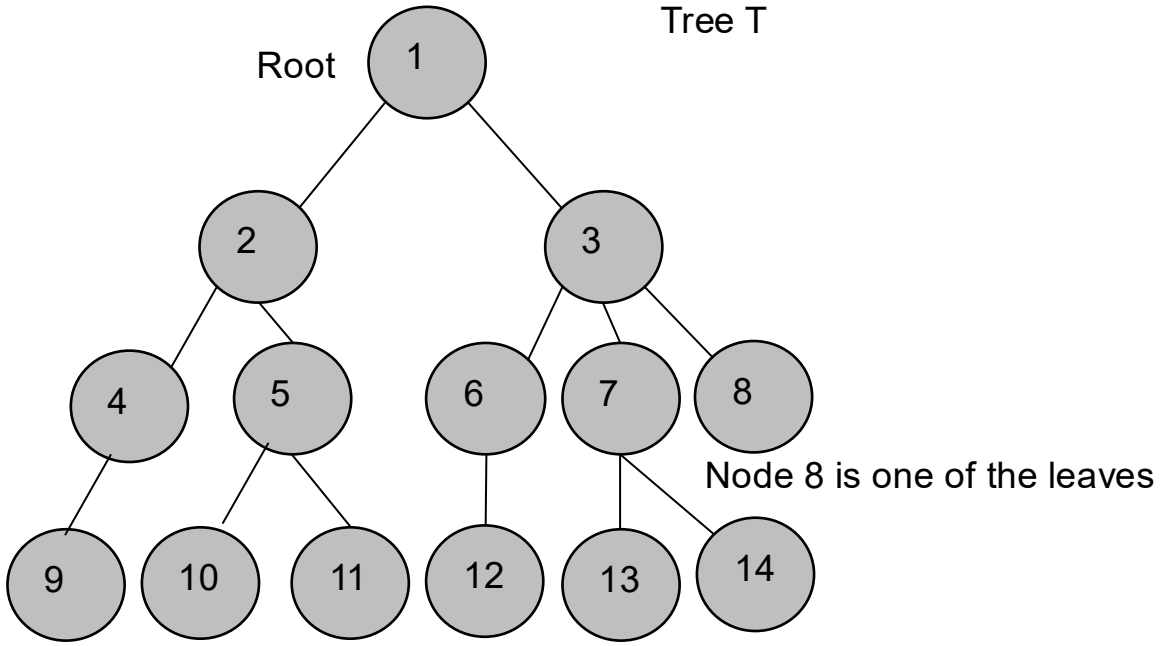
A node u is an ancestor of a node v,

Node v is the descendant of node u

Internal node is a node *with one or more children*

External node is a node *without children*

External nodes are also called Leaves



Node 12 is the descendant of Node 3

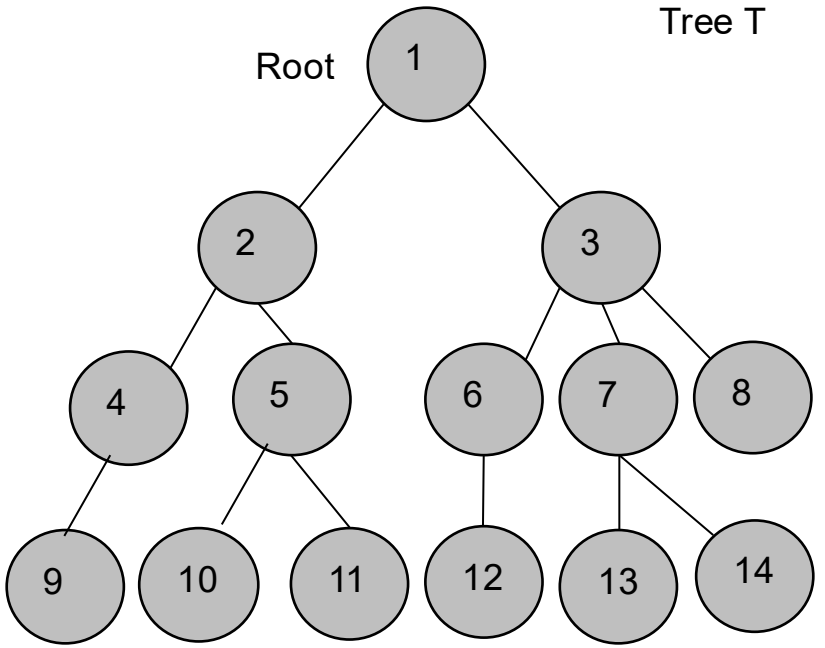
Tree: Depth and Height

Let p be a position within tree T. The depth of p is the number of ancestors of p, other than p itself.

The depth of p, recursively :

- If p is the root, then the depth of p =0
- Otherwise, the depth of p is one + the depth of the parent of p

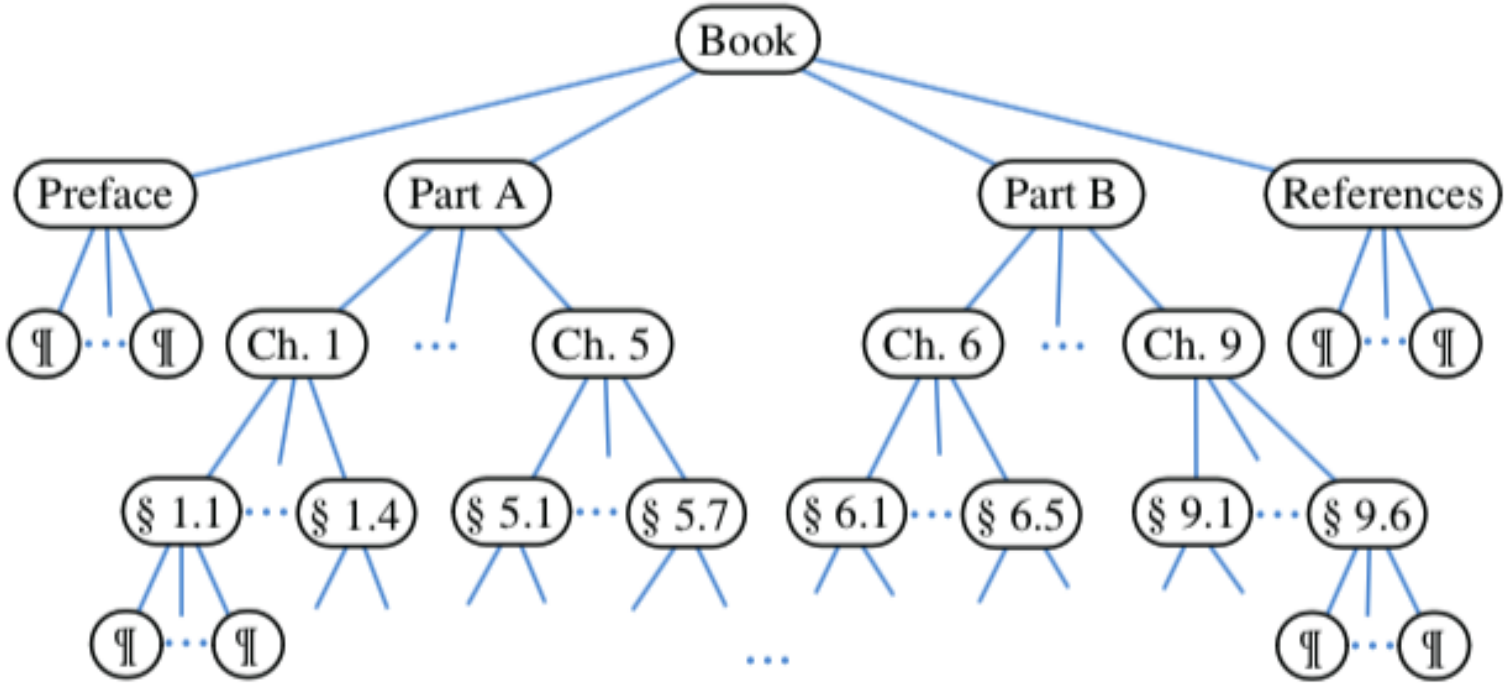
The depth of Node 5 is 2 (1 + the depth of the parent)



Ordered Trees

A tree is ordered :

- If there is a meaningful linear order among the children of each node,
- purposely identify the children of a node as being the first, second, third, and so on.
- The order is from left to right



Data structure and algorithm in Java 6th edition

Binary Trees

A binary tree is an ordered tree with:

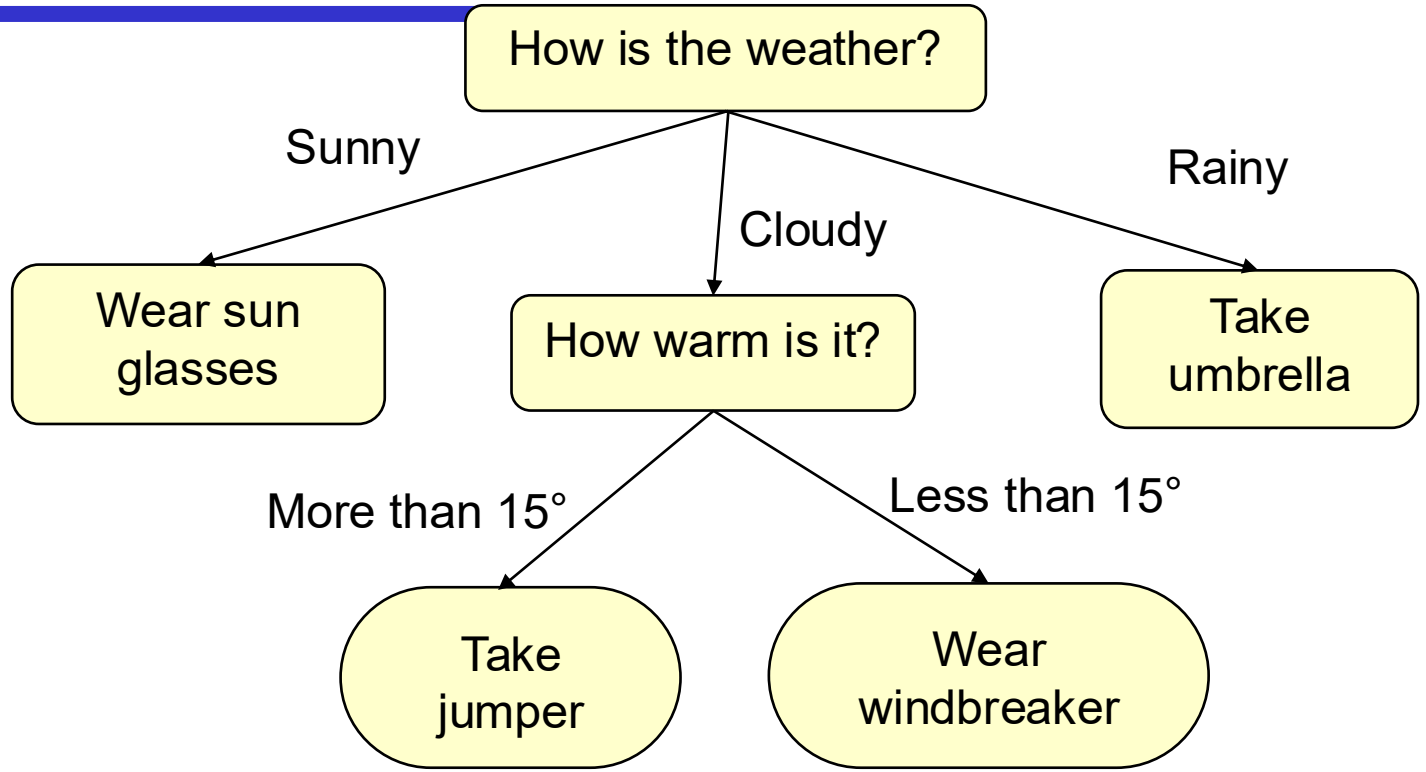
- Every node has at most 2 children
- Each child node is labelled as being either a left child or a right child
- A left child precedes a right child in the order of children of a node

The subtree rooted at a left or right child of a node v is called a left subtree or right subtree, respectively, of node v.

A binary tree is proper (full binary) if each node has either zero or two children, otherwise the tree is improper.

Decision Trees

- A *decision tree* is a tree whose nodes represent decision points, and whose children represent the options available



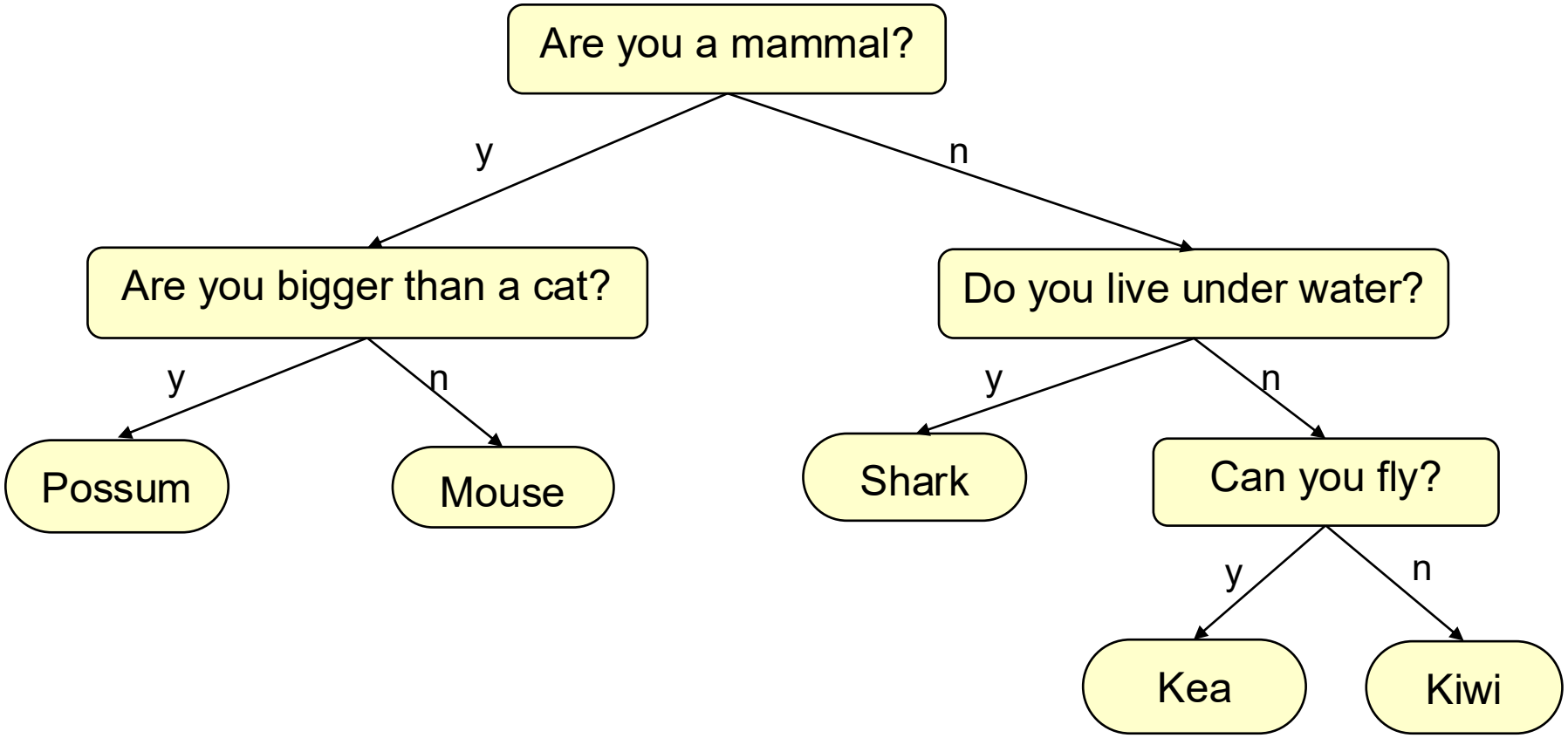
- The leaf nodes of a decision tree represent possible conclusions that might be drawn

- Decision trees are useful in diagnostic situations (medical, car repair, etc.)

- A simple decision tree, with yes/no questions, can be modelled by a **binary tree**

Decision Tree - Example

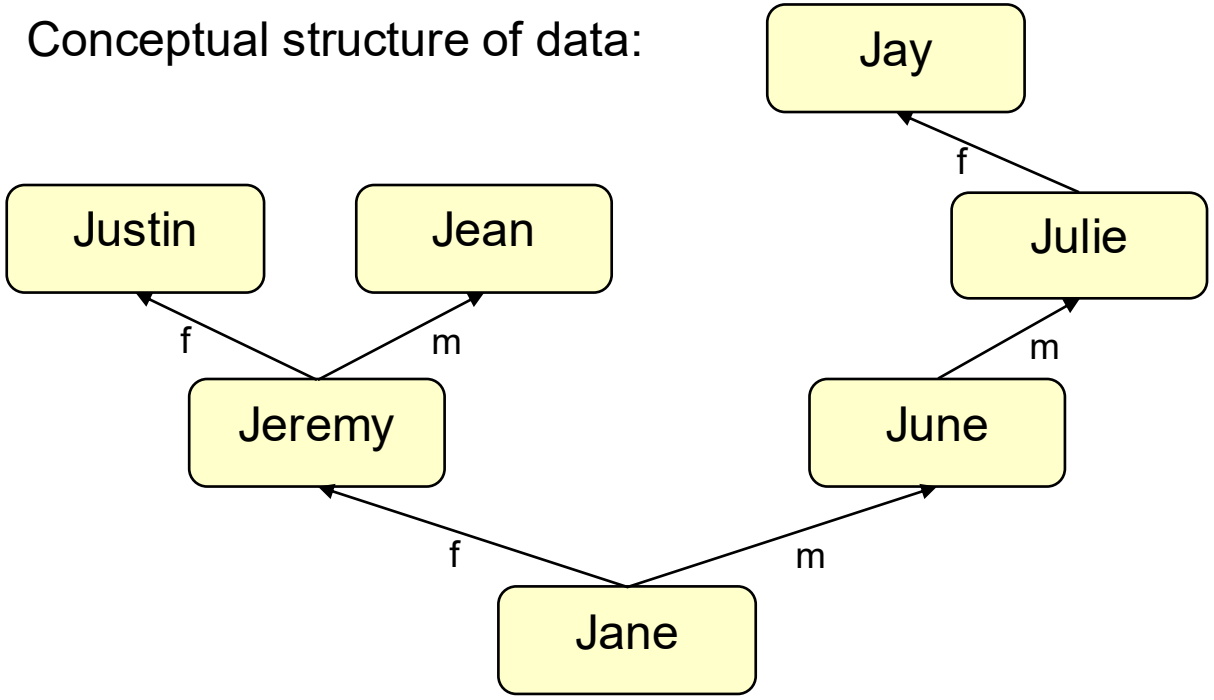
- Binary tree for a yes-no-decision process (Who are you?)



Trees

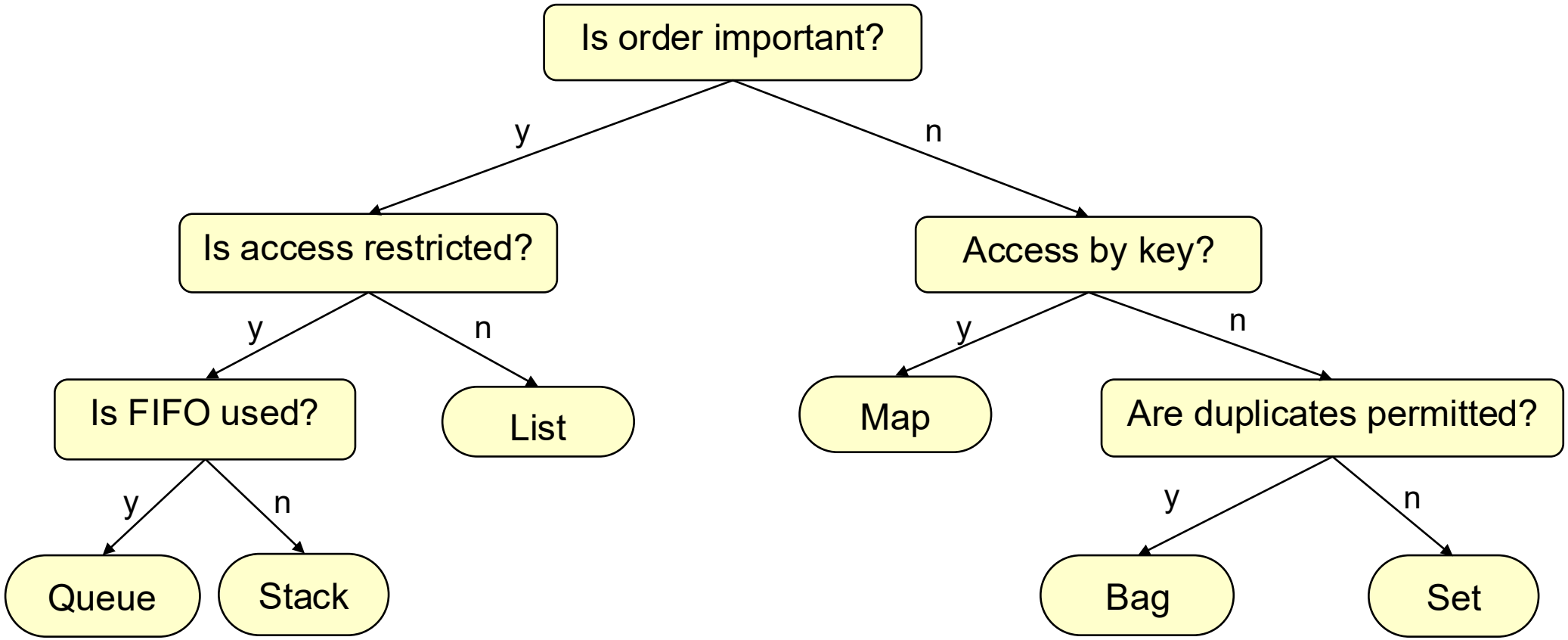
- Maps, Sets, Bags: collections with no structure
- Lists, Queues, Stacks, Deques: collections with linear structure (in order)

- Not all collections fit into those two structures.
- eg, genealogy data



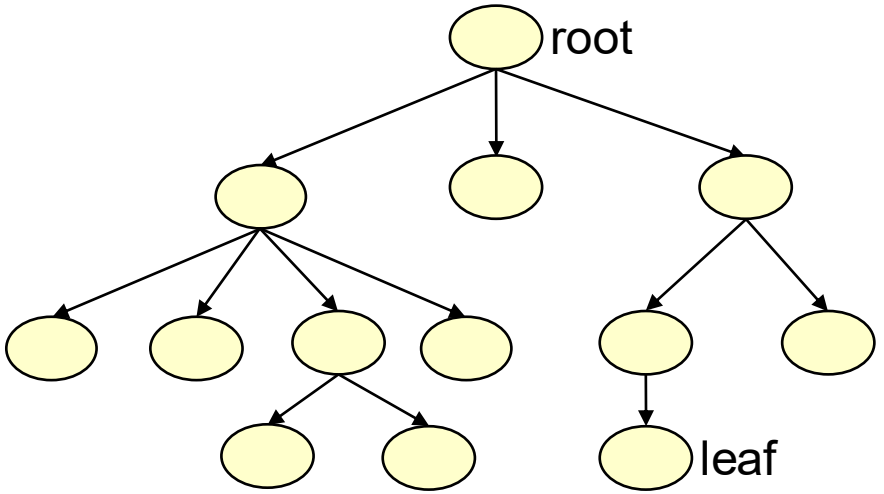
Decision Tree - Example

- Binary tree for a yes-no-decision process (finding a data structure)



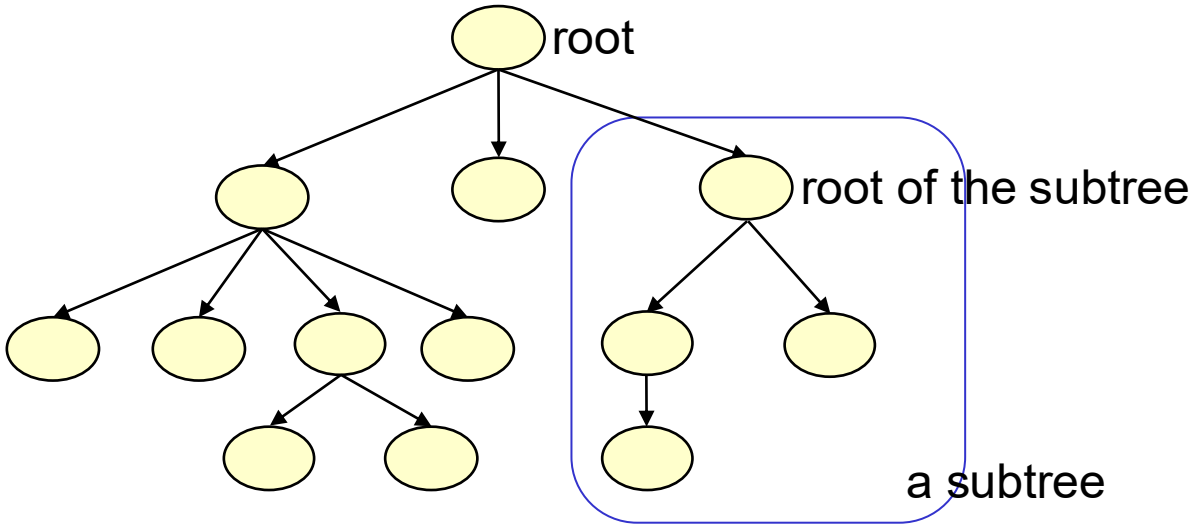
Tree Notation:

- Tree made of nodes with links
- Nodes linked to child nodes
  - might have a limit on number of children, or no limit
  - each node has one parent
- Root node is the base of the tree
  - root node has no parent
  - we typically draw it at the top!!
- Leaf nodes are nodes with no children
  - we typically draw them at the bottom!

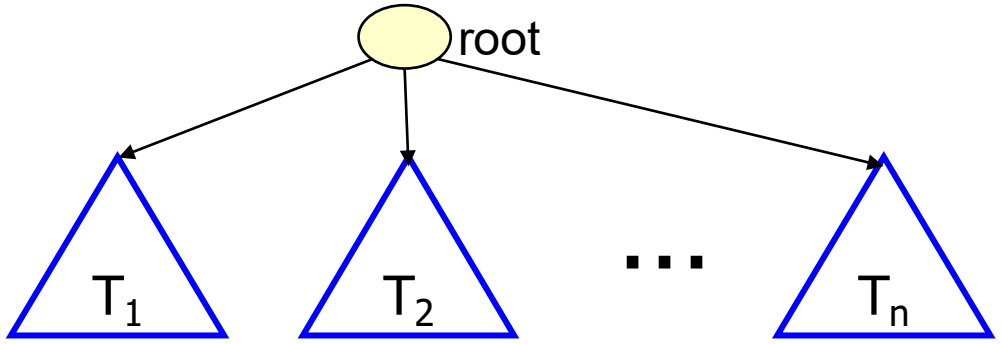


Subtrees of a Tree

- A *subtree* is a tree structure that makes up part of another tree



- A tree T consists of a root and a sequence of subtrees $T_1, T_2, \ldots, T_n$
  - One subtree for each of the children of the root



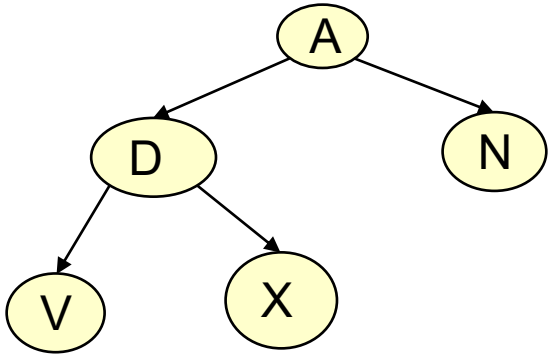
Tree Structures

We will discuss how to create, use and update a tree structure

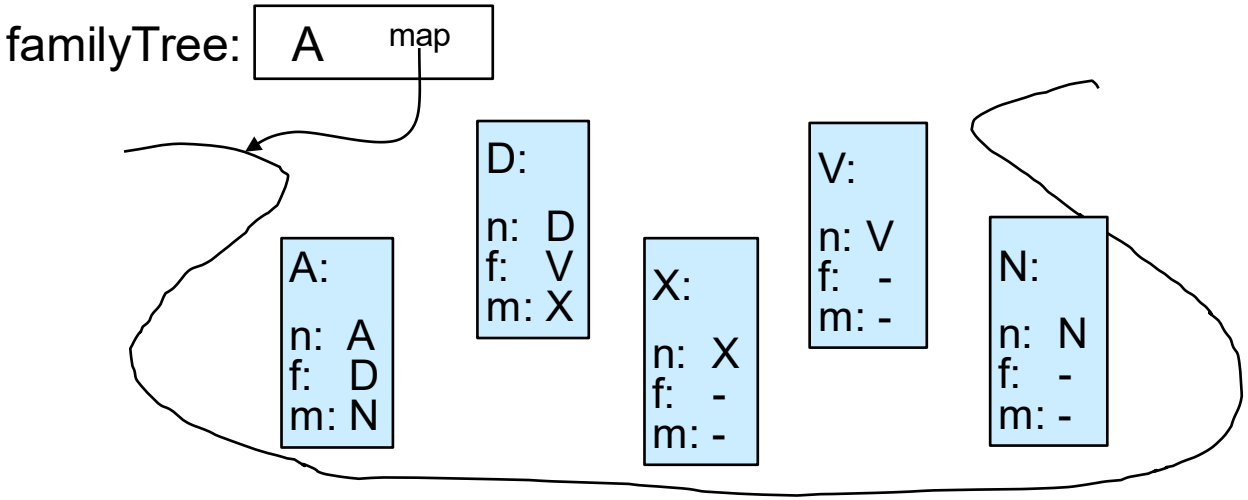
- What Data Structures support tree-structured data?
- How to insert nodes into a tree structure?
- How to retrieve data from a tree structure?
  - One data item
  - Data items along a path from the root
  - All data items in a tree -> Tree Traversal

Data Structures for Tree Structured data

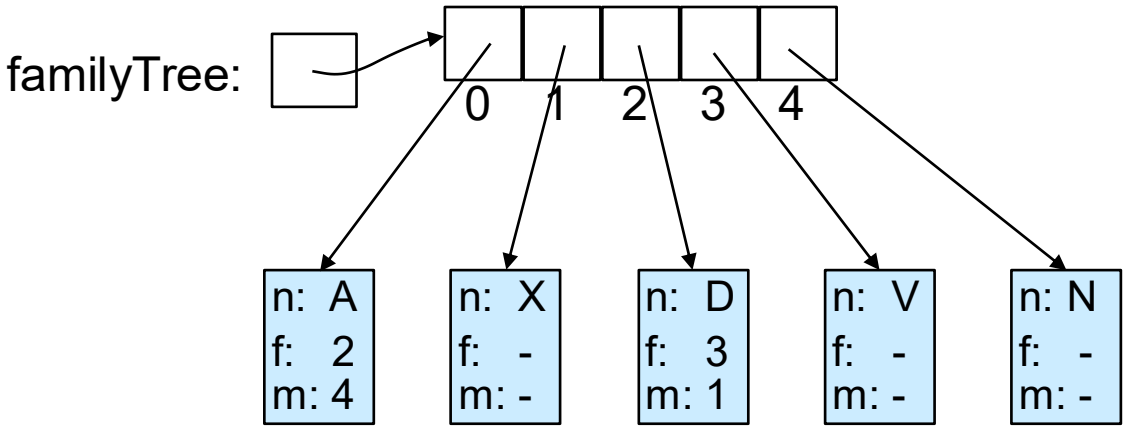
- Nothing new - you already have all the bits!



- Map:
  - key = name of item,
  - item contains data plus names of child nodes
  - need name of root node.



- List:
  - item contains data plus the index of child nodes
  - root at index 0



Data Structures Recap: Map

The “Map” (Key = Name, Value = Child Node Data)

Purpose: This structure is designed to represent a direct relationship between a node's name and its associated data *plus* a list of its child nodes.

Think of it like a dictionary where you look up a name to find the related information.

- Key: `name` (String) – This is the unique identifier for a node in the hierarchy.
- Value: A complex object (likely another tree structure or a list) containing:
- data: The data associated with the node.
- children: A list of child nodes, where each child node is also represented by its `name`.

Data Structures Recap: Map

```
Map<String, NodeData> nodeMap = new HashMap<>();
    NodeData root = new NodeData("Root", /* root data */);
    nodeMap.put("Root", root);

    NodeData child1 = new NodeData("Child1", /* child1 data */);
    nodeMap.put("Child1", child1);

    NodeData child2 = new NodeData("Child2", /* child2 data */);
    nodeMap.put("Child2", child2);

    NodeData child1Data = nodeMap.get("Child1"); // Accessing data for "Child1"
```

Strengths: Simple to understand and implement. Provides direct access to node data by name.

Weaknesses: Can become inefficient for very deep hierarchies because you have to traverse the map to find a node. Doesn't inherently represent the hierarchical structure in a way that's easy to navigate.

Data Structures Recap: List

“List” (Index-Based)

Purpose: This structure represents the hierarchy as a linear list, where the index of each element in the list corresponds to its position in the tree.

- Structure: A simple `List<NodeData>` where each `NodeData` object contains:
  - `data`: The data associated with the node.
  - `childIndices`: An *integer array* representing the indices of the node's child nodes within the list.

Root at Index 0: The root node is always at index 0.

Data Structures Recap: List

```
List<NodeData> nodeList = new ArrayList<>();
    NodeData root = new NodeData("Root", /* root data */);
    nodeList.add(root); // Root at index 0

    NodeData child1 = new NodeData("Child1", /* child1 data */);
    nodeList.add(child1); // Child1 at index 1

    NodeData child2 = new NodeData("Child2", /* child2 data */);
    nodeList.add(child2); // Child2 at index 2

    NodeData child1Data = nodeList.get(1); // Accessing Child1:
```

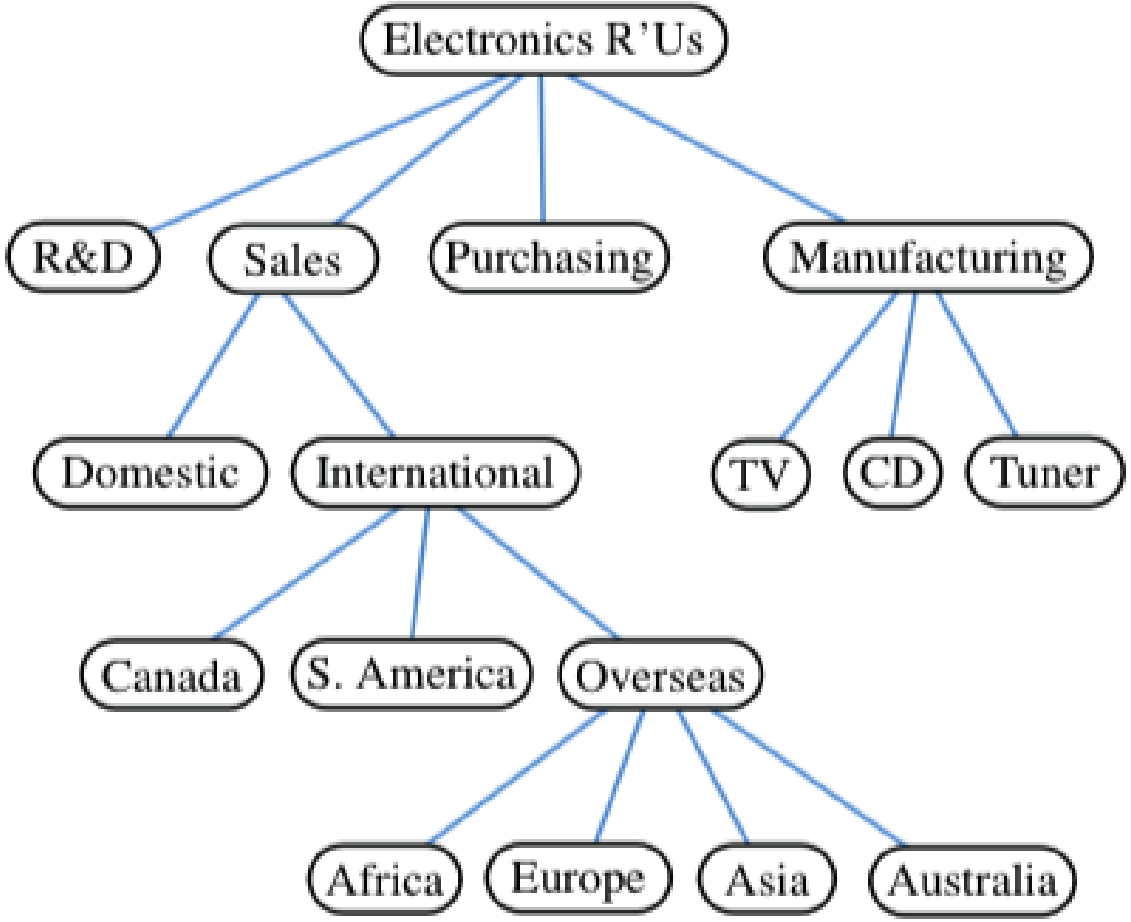
Strengths: Efficient for traversing the tree in a depth-first or breadth-first manner. Simple to implement.

Weaknesses: Requires you to know the index of a node to access it. Less intuitive for representing the hierarchical relationships directly. Can be less efficient if you need to frequently access nodes by name.

Review

The following questions refer to the tree T.

- Which node is the root?
- What are the internal nodes?
- How many descendants does node International have?
- How many ancestors does Node Canada have?
- What are the siblings of the Purchasing node?



- What are the minimum and maximum number of internal and external nodes in an improper binary tree with n nodes?

Review

The following questions refer to the tree T.

1. Which node is the root?
2. What are the internal nodes?
3. How many descendants does node International have?
4. How many ancestors does Node Canada have?
5. What are the siblings of the Purchasing node?

1. Root = Electronic R us
2. Internal nodes: Manufacturing, Sales, International, Overseas
3. International: 4 descendants
4. Ancestor of Canada = 2
5. Siblings: Manufacturing, Sales, R&D

Review

What are the minimum and maximum number of internal and external nodes in an improper binary tree with n nodes?

Let **i** be the number of internal nodes, and **e** be the number of external (leaf) nodes.

$$i + e = n$$

Minimum **Internal** Nodes (i = 1)
The absolute minimum number of internal nodes is one. This is because a tree **must** have a root node. The root node is, by definition, an internal node – it has at least one child.
Without a root, you don't have a tree

Maximum **Internal** Nodes (i = n/2)
To maximise the number of internal nodes, we want to pack as many nodes as possible into the internal.

Review

What are the minimum and maximum number of internal and external nodes in an improper binary tree with n nodes?

Let **i** be the number of internal nodes, and **e** be the number of external (leaf) nodes.

$$i + e = n$$

Minimum **External** Nodes (e = n - 1)
If you have only one internal node (the root), then all the remaining n - 1 nodes must be leaves (external nodes).

Maximum **External** Nodes (e = n/2)
To maximise the number of external nodes, we want each node to have only one child. If we have n/2 external nodes, and each has one child, we've used 1 * (n/2) = n/2 nodes.