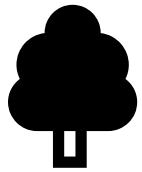
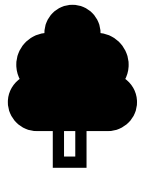




# Trees


Ivan Yahir Gómez Mancilla

# Contents

- Introduction
- Terminology
- Binary trees
- Binary search trees
- Balanced binary search trees
- Some examples

# Introduction

- Linked lists usually provide greater flexibility than arrays, but they are linear structures and it is difficult to use them to organize hierarchical representation of objects. to overcome this limitation, we create a new data type called a tree that consists of nodes and arcs.

# Introduction 2

- Unlike natural trees, these trees are depicted upside down with the root at the top and the leaves at the bottom.
- The root is a node that has no parent, it can have only child nodes.
- Leaves have no children, or rather, their children are empty structures.

# Recursive definition of a tree

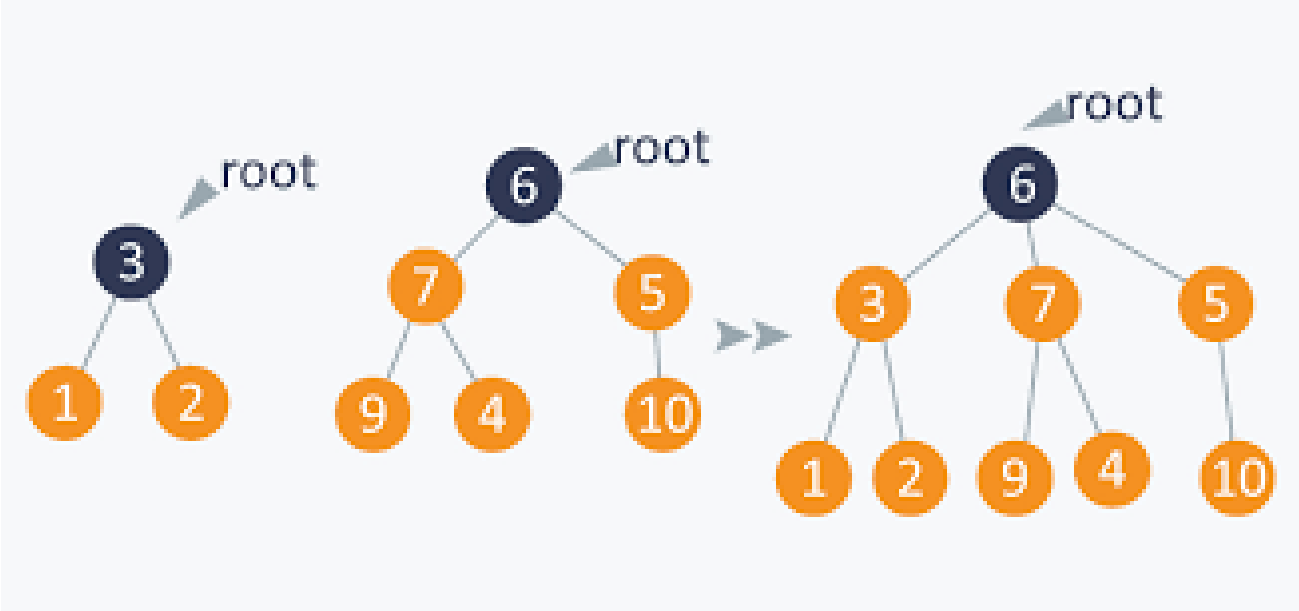
- An empty structure is an empty tree
- if t1,…,tk are disjoint tres, the the structure whose root has as its children the roots of t1,…,tk is also a tree
- Only structures generated by rules 1 and 2 are trees

# Disjoint tree

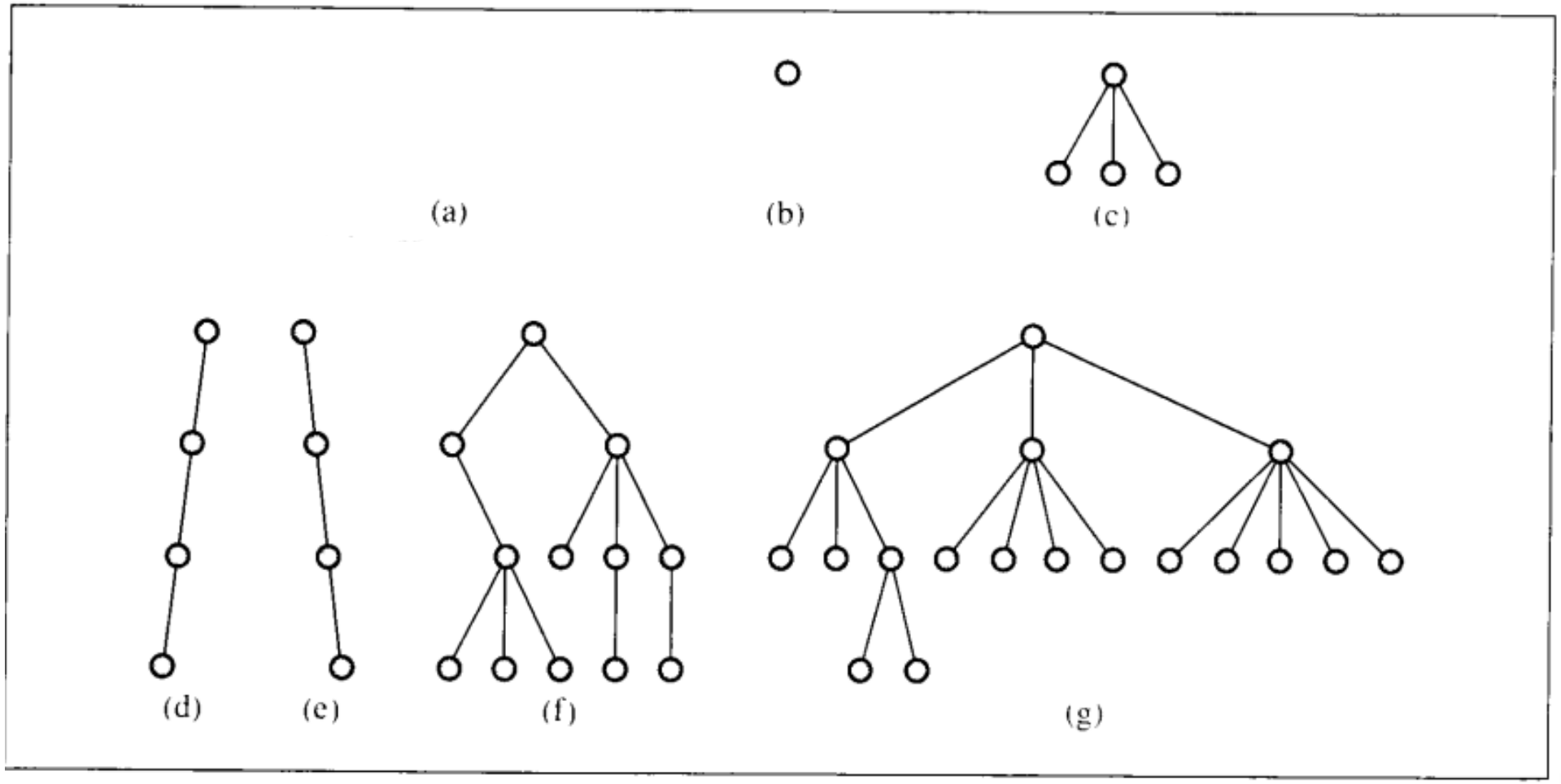
- data structure used to represent a collection of disjoint sets. Each set is represented by a tree, where each node in the tree represents an element in the set. The root of each tree represents the set itself.

# Disjoint set

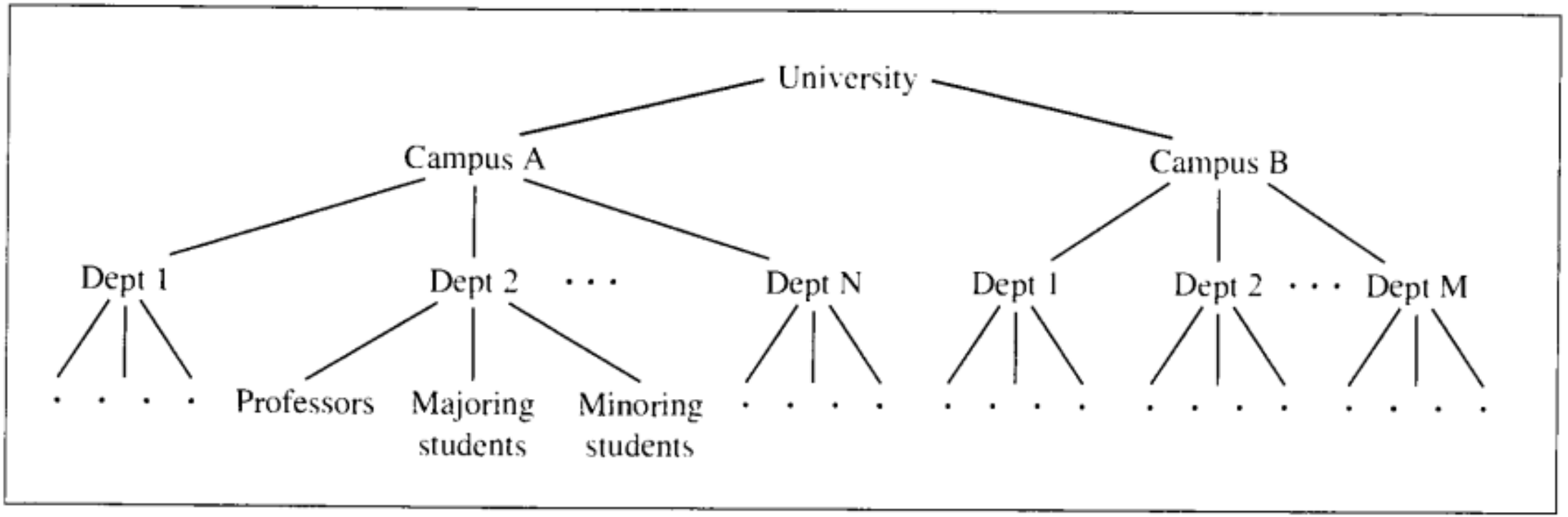
- Collection of sets where each elemt belongs to exactly one set.
- No element can belong to more tan one set in a disjoint set.



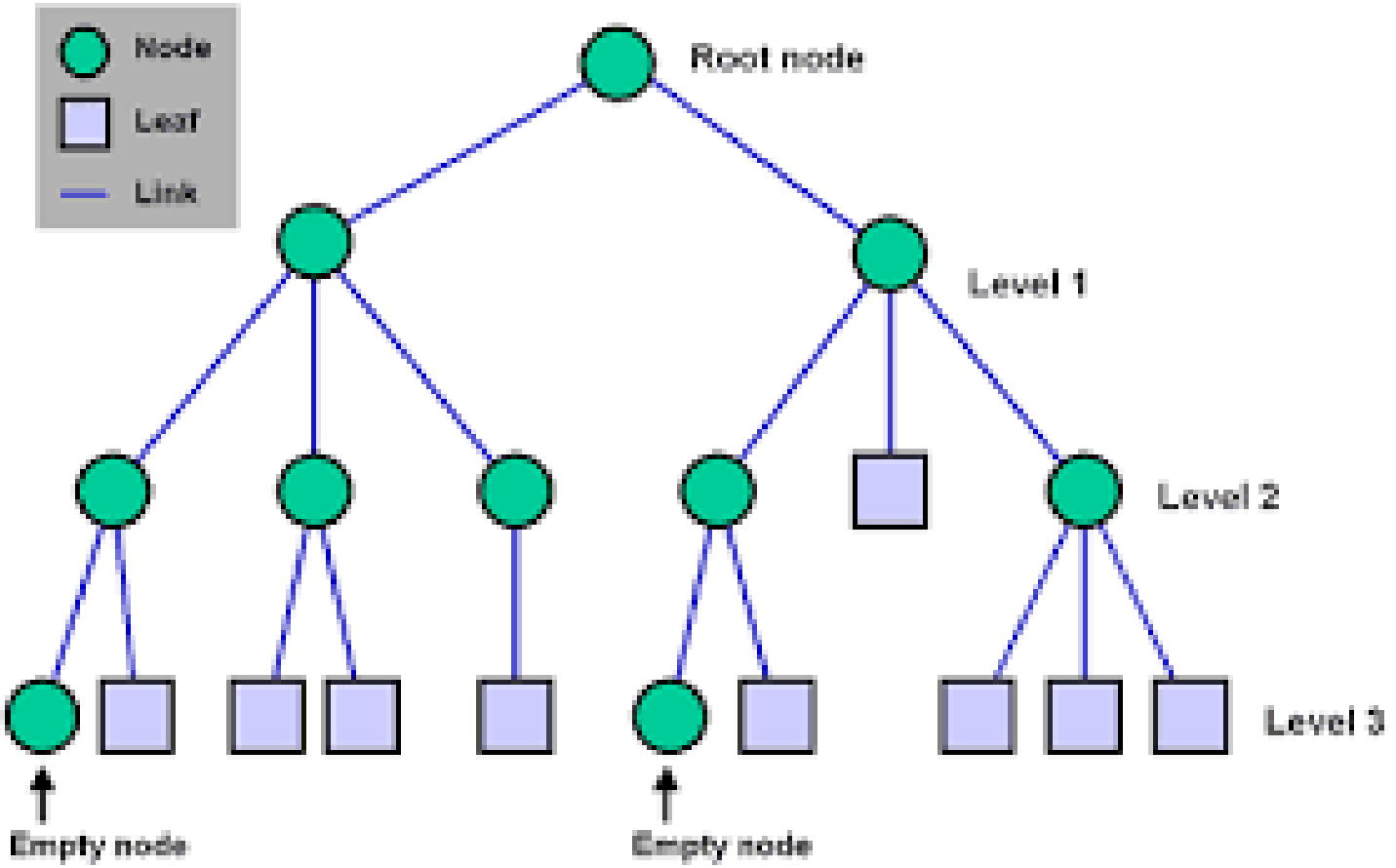
# Which ones are trees?



# Trees in real life



# Terminology



# Relations between nodes

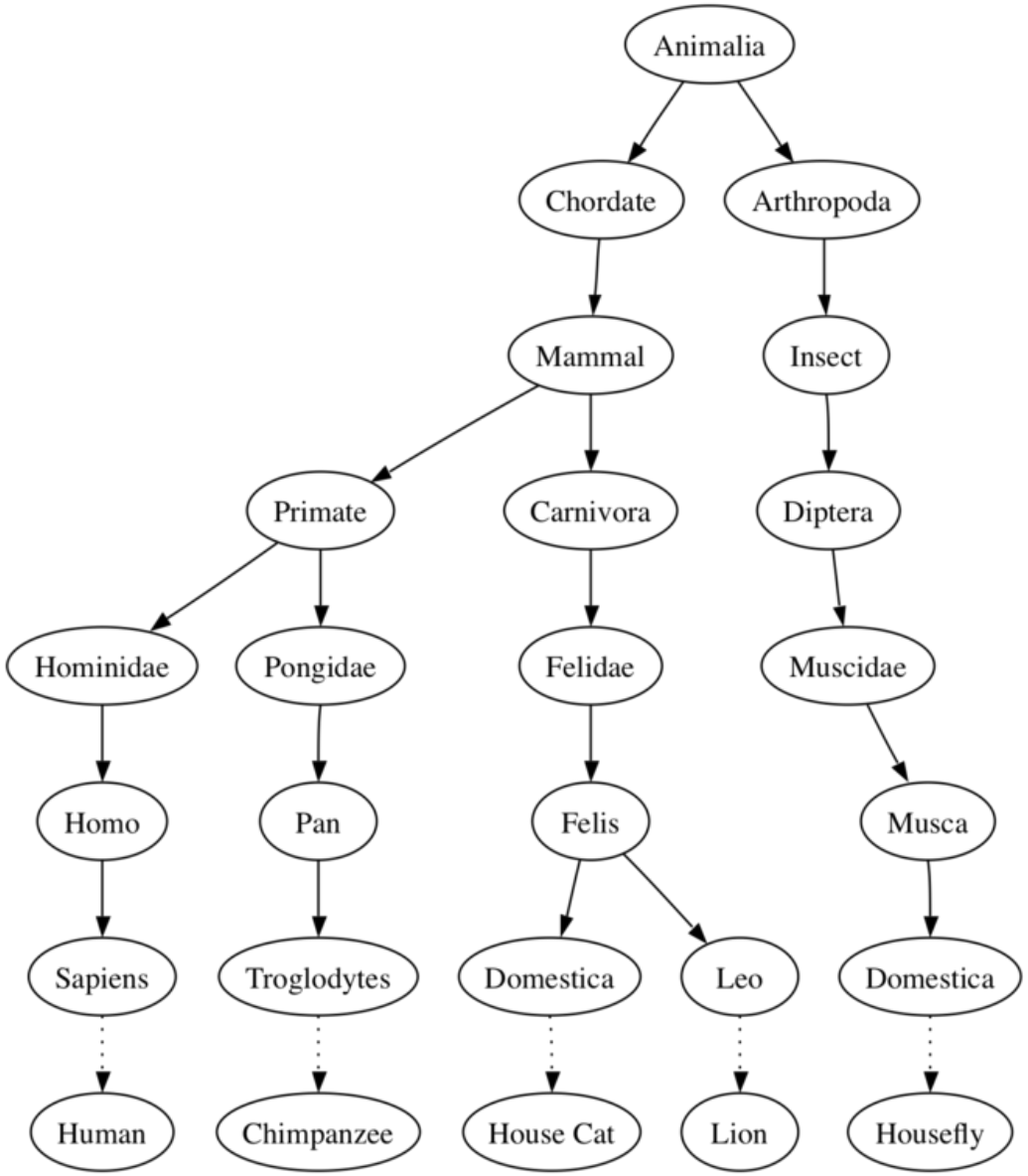
- Parent: A node in a tree that has one or more child nodes.
- Child: A node in a tree that has a parent node.
- Sibling: Nodes that share the same parent node

# Distances and subtrees

- Branch: Path of nodes between the root and a leaf node
- Depth: number of edges from the root node to any node.
- Height: The height of a node in a tree is the number of edges from any node to the deepest leaf node in its subtree
- Subtree: A subtree is a portion of a tree that consists of a node and all of its descendants.

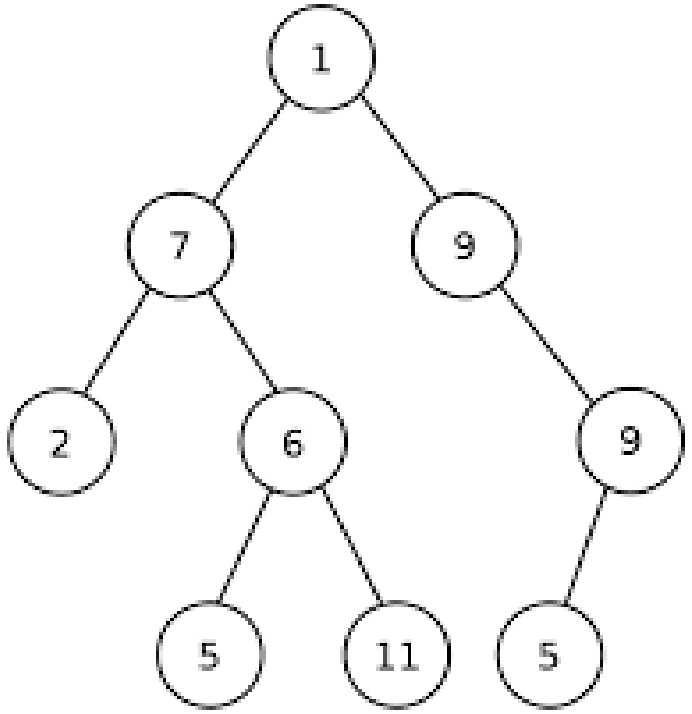
# Exercise

- How many parent nodes does the tree has?
- How many childs does “Mammal” node has?
- Which is the largest branch in the tree?
- Which depth does “Pan” node has?
- Between “Mammal” and “Insect” nodes, which one has the biggest height?
- How many nodes does the subtree starting at “Felidae” node has?



# Binary trees

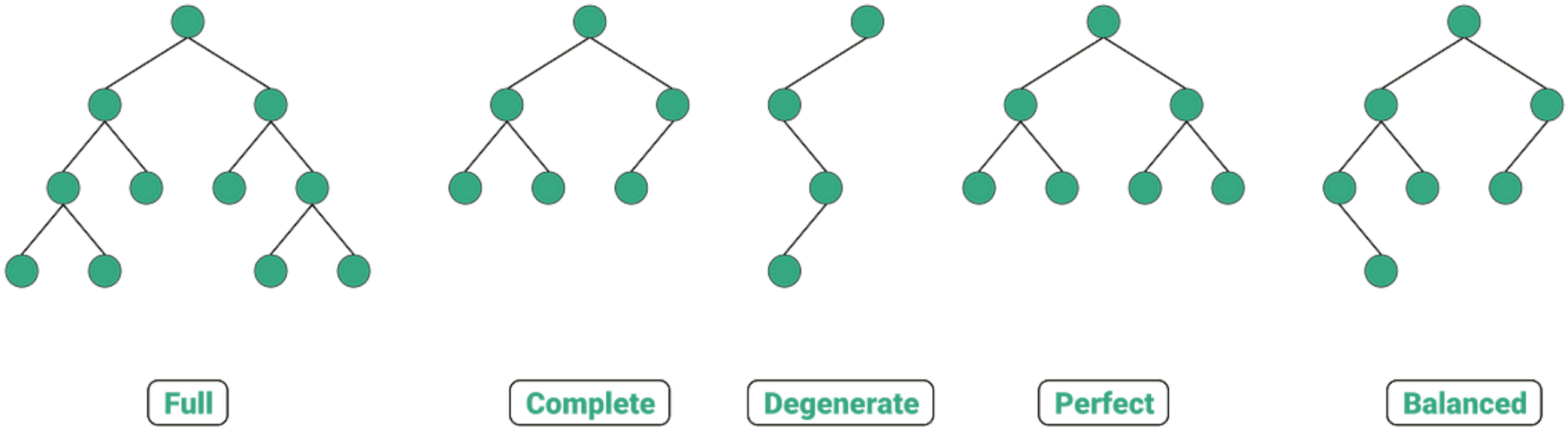
- A tree whose nodes have two children, and each child is designated as either a left child or a right child.



# Types of binary trees

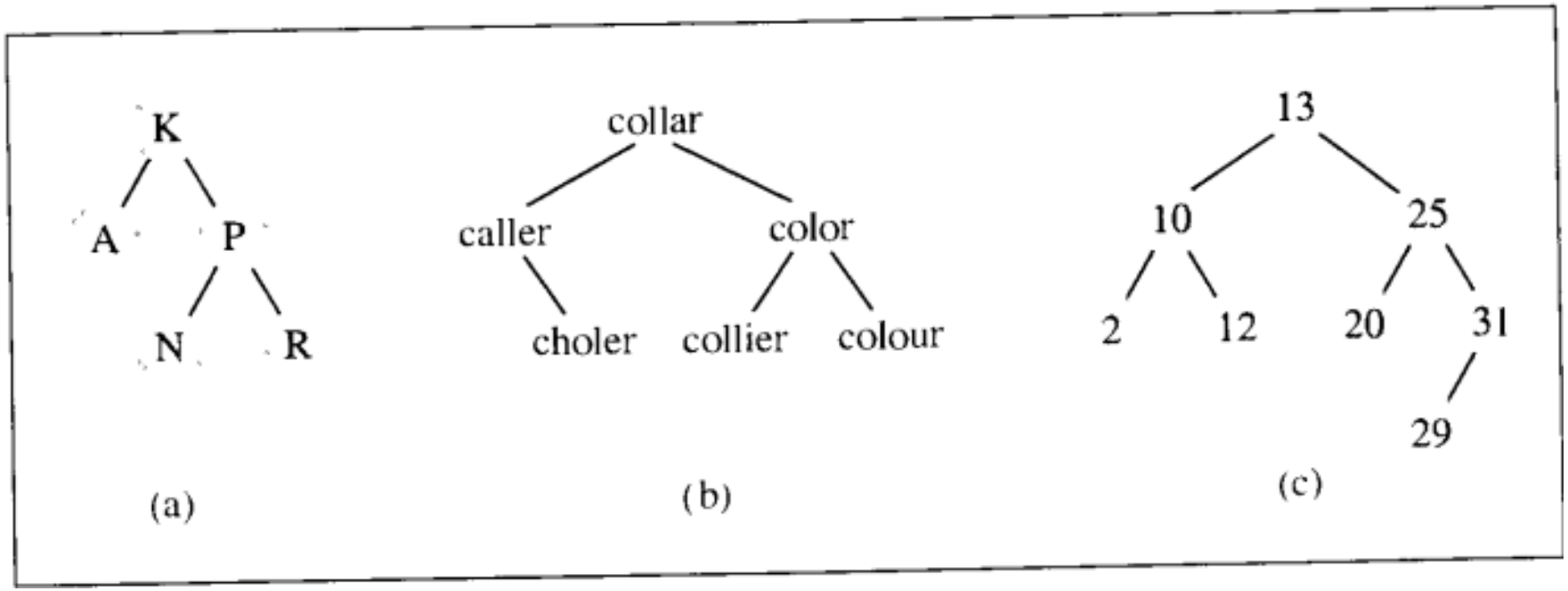
- Full binary tree: every node has either 0 or 2 children.
- Perfect binary tree: all interior nodes have two children and all leaves have the same depth.
- Complete binary tree: every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- Balanced binary tree: the left and right subtrees of every node differ in height by no more than 1
- Degenerate tree: each parent node has only one associated child node.

# Types visualization



# Binary search trees

- For each node n of the tree, all values stored in its left subtree are less than value v stored in n, and all values stored in the right subtree are greater than v.

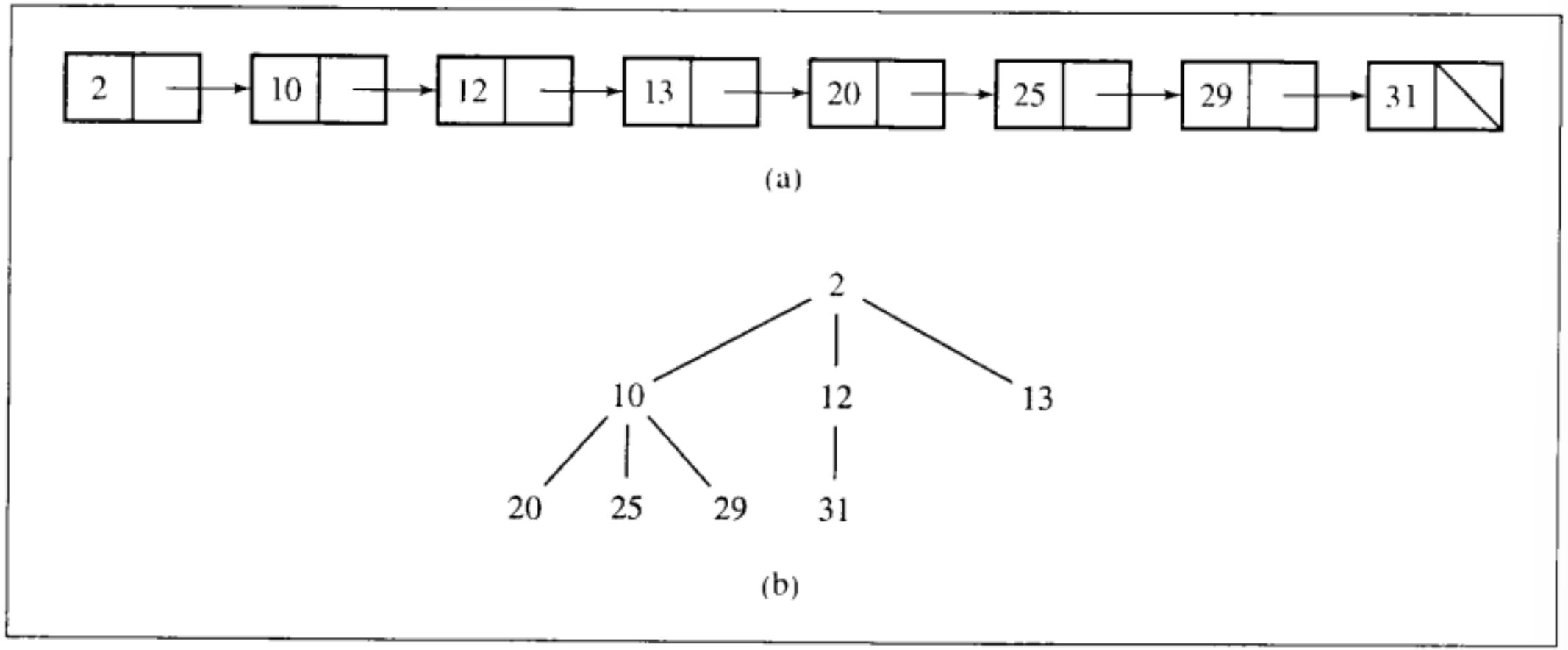


# Implementing a binary tree (as array)

- A node is declared as a structure with an information field and two "pointer" fields. These pointer fields contain the indexes of the array cells in which the left and right children are stored.

| Index | Info | Left | Right |
|---|---|---|---|
| 0 | 13 | 4 | 2 |
| 1 | 31 | 6 | −1 |
| 2 | 25 | 7 | 1 |
| 3 | 12 | −1 | −1 |
| 4 | 10 | 5 | 3 |
| 5 | 2 | −1 | −1 |
| 6 | 29 | −1 | −1 |
| 7 | 20 | −1 | −1 |

# Implementing a binary tree (as a list (by ourselves))



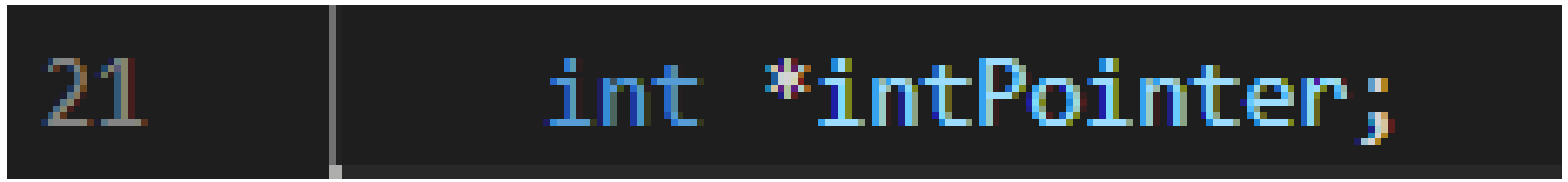
# Node declaration

```
5    typedef struct node
6    {
7        int data;
8        node *left = NULL;
9        node *right = NULL;
10   };
```

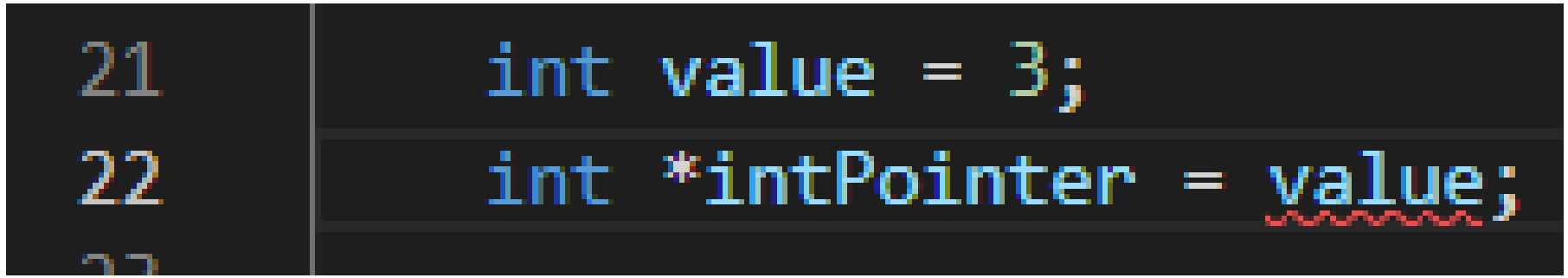
# Pointers clearance

| | |
|---|---|
| _valueType_ *_pointerName_ | Creates a pointer |
| *_pointerName_ | Returns variable being pointed by pointer |
| &_variableName_ | Returns memory adress from a variable |

# Pointers exercises (1)



# Pointers exercises (2)



# Pointers exercises (3)

```
21      int value = 3;
22      int *intPointer = &value;
23      cout << intPointer << endl;
```

# Pointers exercises (4)

```
21      int value = 3;
22      int *intPointer = &value;
23      cout << *intPointer << endl;
```

# Pointers exercises (5)

```
25      float pi = 3.1415;
26      float *floatPointer = &pi;
```

# Pointers exercises (6)

```cpp
#include <bits/stdc++.h>

using namespace std;

typedef struct node
{
    int data;
    node *left = NULL;
    node *right = NULL;
};

int main(){
    node *traverse;
    node n1;
    n1.data = 1;

    traverse = &n1;

    cout << (*traverse).data << endl;
}
```

# Connecting nodes

```cpp
#include <bits/stdc++.h>

using namespace std;

typedef struct node
{
    int data;
    node *left = NULL;
    node *right = NULL;
};

int main(){
    node *traverse;
    node n1, n2, n3;
    n1.data = 1;
    n2.data = 2;
    n3.data = 3;

    traverse = &n1;

    n1.left = &n2;
    n1.right = &n3;
```

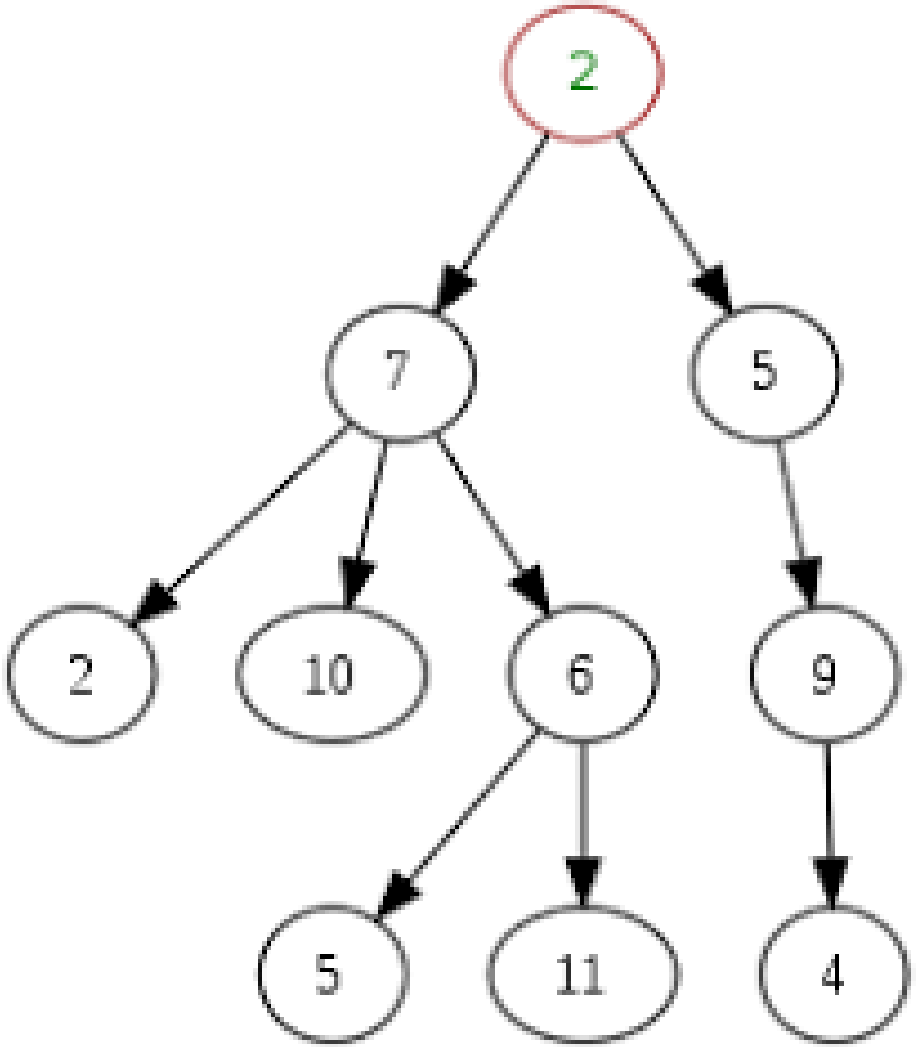
# Connecting nodes exercise

```
24      cout << (*traverse).data << endl;
25      traverse = (*traverse).left;
26      cout << (*traverse).data << endl;
```

# Traversing trees

- Inorder:
  - left subtree
  - root
  - right subtree
- Preorder:
  - root
  - left subtree
  - right subtree
- Postorder
  - left subtree
  - right subtree
  - root

# Tree traversal exercise



# Inorder example

```
12  void printInorder(node *move)
13  {
14      if(move == NULL) return;
15
16      printInorder((*move).left);
17
18      cout << (*move).data << endl;
19
20      printInorder((*move).right);
21  }
```

# Class exercise

- Program preorder
- Program postorder