Lecture outline: introduction to trees

Trees come in two flavours:

- From a more theoretical point of view, these are a special kind of graphs, those that are connected and acyclic (*i.e.* have no cycle).
- From an application point of view, one considers **rooted** trees, where a special vertex is distinguished (the root); they appear in many applications.

We are going to explore quickly these two aspects of trees, in order to achieve two goals:

- Introduce some definitions, concepts and basic results we will reuse a lot in the rest of this course.
- Provide convincing examples of the wide range of applications of rooted trees.

Trees as graphs

We first take a purely theoretical point of view: trees are special graphs.

Definition: trees and forests. A **tree** T on the vertex set V is a connected and acyclic (so loop-free) graph with vertex set V.
A **forest** is a collection of trees, or equivalently an acyclic graph.

Examples.

Terminology. A vertex v that is incident to a single edge in a tree is called a **pendant vertex** or a **leaf**.

We now introduce basic but very important properties on trees.

Theorem. Let T be a tree. If a and b are distinct vertices of T, there is a **unique path** that connects a and b.

We are not going to provide formal proof of this result now, but will convince ourself that it is true. This is important as when we will write proofs (soon), it is always important to have a general intuition of why a result is true before writing a formal proof.

Theorem. Let T be a tree such that $|V(T)| \geq 2$ (T has at least two vertices). Then T has **at least two pendant vertices**.

Again, let's convince ourselves that this is true and in the process of this, gather ideas for writing a proof.

We now introduce a fundamental definition: spanning trees.

Definition: spanning tree. Let G be a connected graph. A **spanning tree** of G is a connected and acyclic spanning subgraph of G.

Remark. Another way to read this definition is that a spanning tree of $G = (V, E)$ is a tree $T = (V, E')$ such that $E' \subseteq E$.

Remark. If G is not connected, it can not have a spanning graph, but it has a spanning forest, composed of spanning trees of all its connected components.

Examples.

Application: network robustness. We now review a natural application of spanning trees.

Context. Assume you manage a network, for example computers connected into a network, that can be represented by a graph (vertices are computers, and edges as communication cables between computers).

You want to make sure that, even if a few cables stop to work, any pair of computers can still communicate. To do so, you can use some cables that can not fail. But these cables are expensive and you can not use them for all your network. So you want to minimize the cost of this operation.

To model the cost of using the non-failing cables, we assume that for every edge $e = (a, b)$ of your network, there is a cost c_e associated to linking computer a and b by a cable that can not fail.

Question. Which edges should you pick that represent the pairs of computers that will be linked by cables that can not fail ?

Solution. You want to select a set of edges that forms a **spanning tree** of G and whose sums of the costs associated to the edges is minimum among all such spanning trees: this is called a **minimum spanning tree** of G.

Building roads. How is that different?

We conclude this quick overview of trees with a useful theorem that gives equivalent definitions of a tree.

Theorem. Let $G = (V, E)$ be a loop-free graph. The following statements are **equivalent**:

a) G is a tree.

b) G is connected and removing any edge from E disconnects G.

c) G contains no cycle and $|E| = |V| - 1$.

d) G is connected and $|E| = |V| - 1$.

e) G contains no cycle and, for every $\{a, b\} \notin E$, adding the edge $\{a, b\}$ creates a cycle.

To prove this we should do something such as proving 5 one-way implications: $a) \Rightarrow b)$, then $b) \Rightarrow c)$, then $c) \Rightarrow d)$, then $d) \Rightarrow e)$, then $e) \Rightarrow a)$. Again, we will only look at ideas on how to prove this set of equivalent statements.

Rooted trees

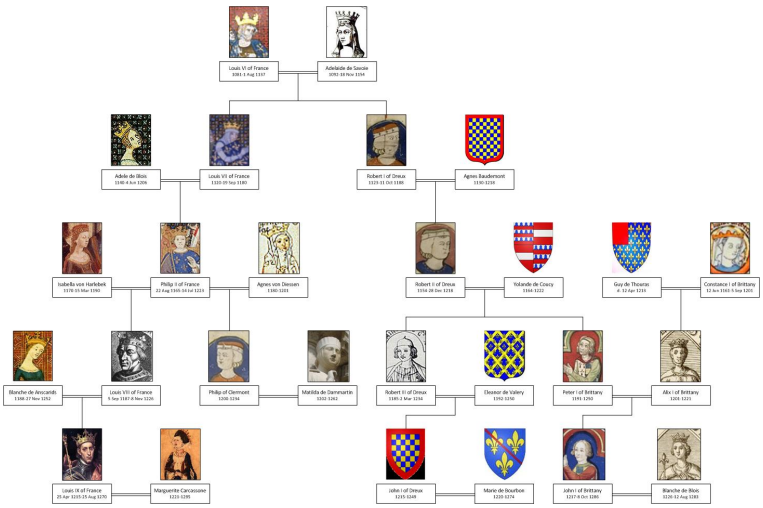
We now turn our attention to rooted trees, that have many applications. The textbook presentation relies on the notions in-degree and out-degree. We follow a simpler and more intuitive way to define a rooted tree.

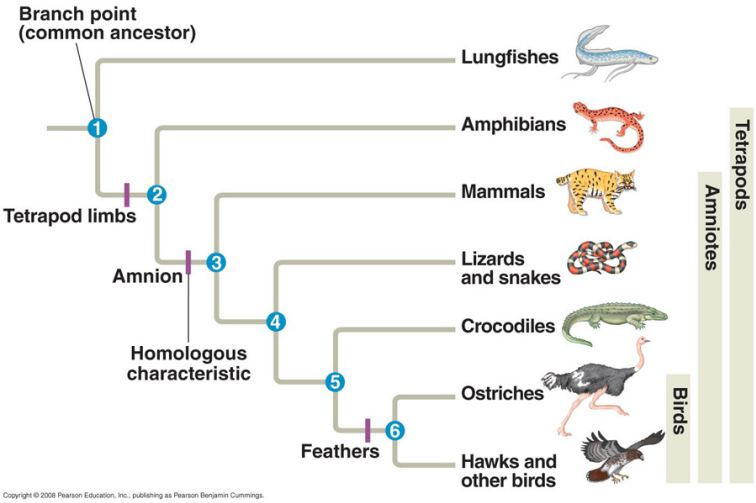
Definition: rooted tree. A **rooted tree** is a pair (T, r), where T is a tree and $r \in V(T)$ is one of its vertices, which is then called the **root**.

Remark. Often, the tree is represented by "pulling" on the root vertex, in order that visually, all vertices are going away from the root. It is then common to consider that the graph is directed, with all arcs going away from the root and toward the pending vertices.

Examples.

The terminology related to trees takes its origin from both the vocabulary of botany, but also from the vocabulary of genealogy, as trees were used earlier as structures to represent family trees, and are still used a lot to represent evolution:





The important terms to know are the following: **root, leaf, internal vertex, parent, child, ancestor, descendant, subtree, level, height**.

1. The root is the only vertex with no parent

2. A vertex a is an **ancestor** of a vertex b (and b is said to be a **descendant** of a) if a appears before b on the unique directed path from the root to b.

3. If (a, b) is an arc of a tree, a is the **parent** of b and b a **child** of a.

4. Leaves are the vertices with no children. Non-leaf vertices are called **internal vertices**.

5. The **subtree** at a vertex v is the subgraph induced by v and all its descendants.

6. The i^{th} **level** in a rooted tree is the set of all vertices that are i edges away from the root (the path from the root contains i edges).

7. A tree has **height** h if it has h levels.

In most applications of trees, the order of children of a given vertex does not matter. But in several applications, especially in computer science, this order does matter: changing the order of the children of a vertex results in a tree representing a different information.

Definition: ordered trees. A **rooted ordered tree** is a rooted tree in which the order of the children of a vertex matters.

Remark. To understand the need to introduce the notion of order of children, the most natural example is the encoding of arithmetic expressions by trees, which is used in many computer systems.

Example

$$2 + 3a + (a - 1)(b + 1) + (a(b + 1) - b(a - 1))$$

Definition: m-ary trees. A rooted tree is m-ary if every vertex has up to m children. If $m = 2$, the tree is said to be **binary**.
An m-ary tree is **complete** if every vertex has 0 or m children.
An m-ary tree of height h is **balanced** if every leaf is at level h or $h - 1$.

Remark. For a complete (ordered) binary tree, one says that every internal vertex has a **left child** and a **right child**.

Theorem. Let T be an m-ary tree of height h with ℓ leaves.
Then $\ell \leq m^h$ and $h \geq \log_m(\ell)$. If T is a complete and balanced m-ary tree, then $h = \lceil \log_m(\ell) \rceil$.

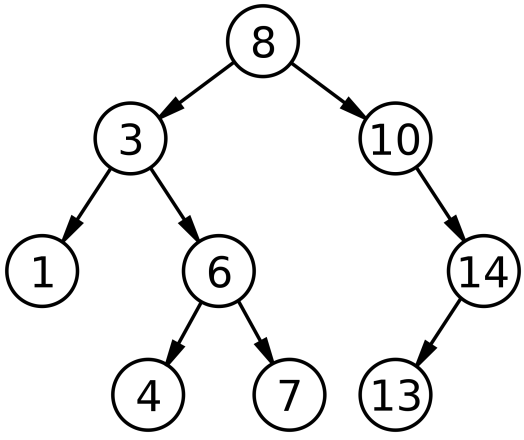
Example.

Application: balanced binary search trees. We want to use trees as an efficient data structure to store and retrieve information from a set of distinct integers.

Context. Assume we have a computer programs manipulating sets of integers, and requesting often to decide if a given number x is in a given set S.

A naive way to perform this task is to have the numbers of S arranged into an array and to scan this array for every request. In the worst case, it can take up to $|S|$ comparisons between x and the numbers in S to answer this request.

Solution: Balanced Binary Search Trees. We can do better with a balanced binary search tree (BST): this is a complete binary tree of height $\lceil \log_2(|S|) \rceil$, where every vertex is labeled by an integer from S, such that, for a vertex x, all vertices in the subtree at the left (respectively right) child of x are smaller (respectively larger) than x.



Claim. If S is encoded into a balanced BST, every request to decide if a given integer belongs to S or not can be answered in time $O(\log_2(|S|))$.

Suffix trees.

We conclude by an example of using trees in bioinformatics that was instrumental to help computational biologists to cope with the dramatic size of genomes such as the human genome (a text of roughly **three billions** letters over $\{A, C, G, T\}$). It has also applications in many other information retrieval contexts.

The problem. We are given a very large string (a human genome for example), denoted by T (the text), and a small string, called the pattern and denoted by P. Assume that T has size n and P size m.

We want to answer the following question: what are the positions i in T such that $T[i, i + m - 1] = P$? In other words where are the occurrences of P in T, if any ?

This is a very common question in bioinformatics: assuming you know the biological function of the molecule encoded by the string P, then in a newly sequenced genome you want to find if this molecule appears, which will give you some insight about the biology of this new genome.

The naive solution. We can design a very simple algorithm to solve this problem.

```
For i from 1 to n Do
    j := 1
    While j ≤ m and T[i + j − 1] = P[j] Do j := j + 1
    If j = m + 1 Then Print "Pattern found at position i"
```

What are the problems with this algorithm, if we consider its complexity in terms of the number of comparisons between symbols of T and symbols of P ?

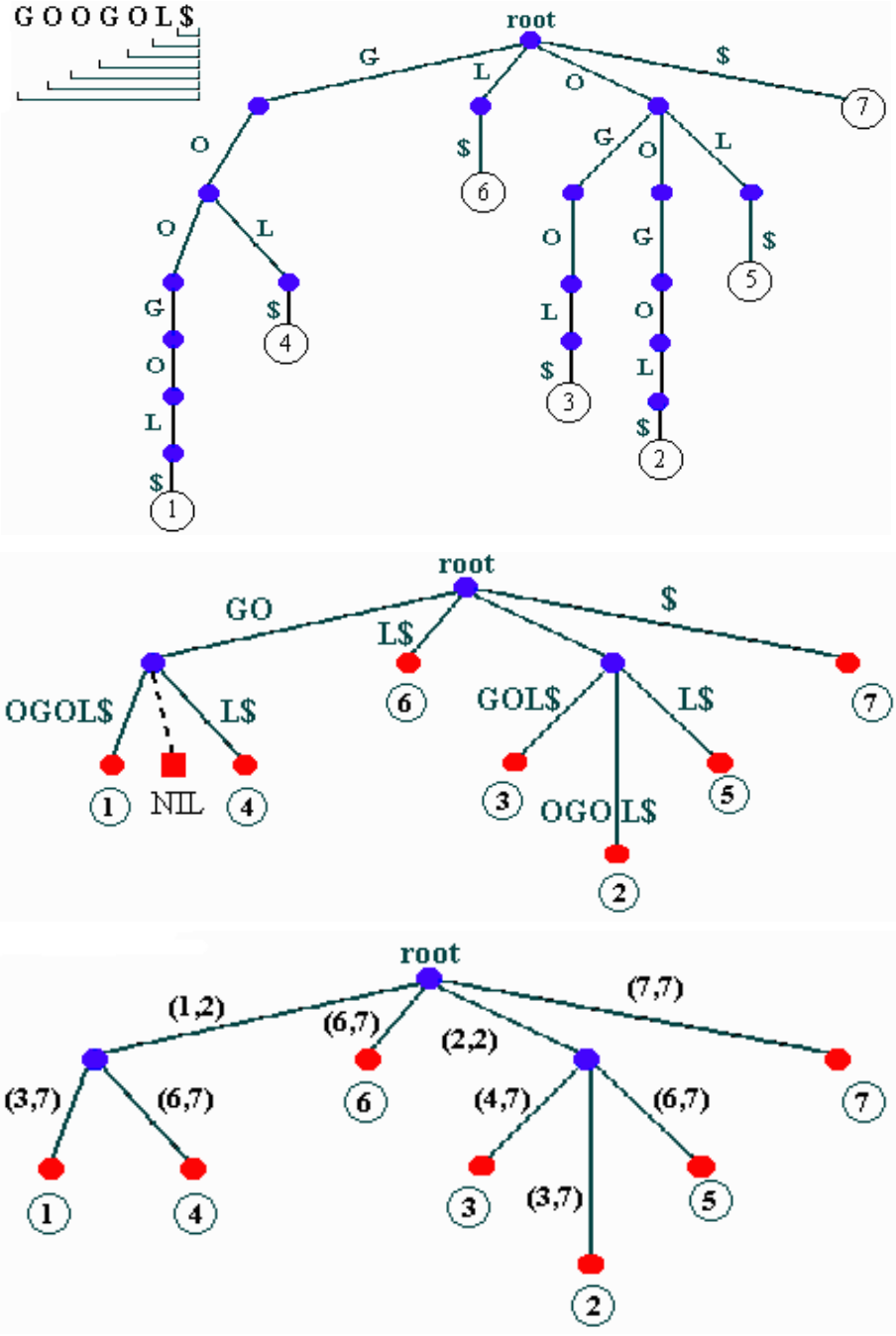
The suffix tree. Then comes the following crucial observation.

Claim. If $P = T[i, i+m]$ for some i (*i.e* there is an occurrence of P in position i of T), then P is a prefix of the suffix $S[i, n]$.

We can leverage this property to try to reduce the time needed to find the positions of all occurrences of P in T ?

Idea. We will organize all suffixes of T into a rooted tree structure in order that:

- The space occupied in memory by this tree is of the same order than n (*i.e.* in $O(n)$);
- Detecting if P appears in T can be done using at most m comparisons between symbols of T and symbols of P.



Remark. The key observation to be convinced that the space occupied by the tree is in $O(n)$ is that every internal vertex has at least two children, which implies that the number of internal vertices is not larger than the number of leaves, which is the size of T.