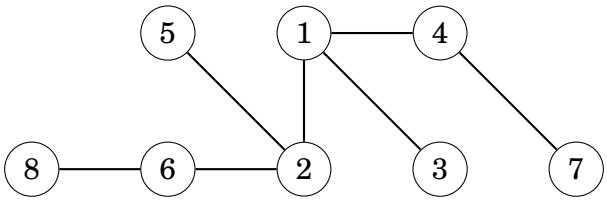
Chapter 14

Tree algorithms

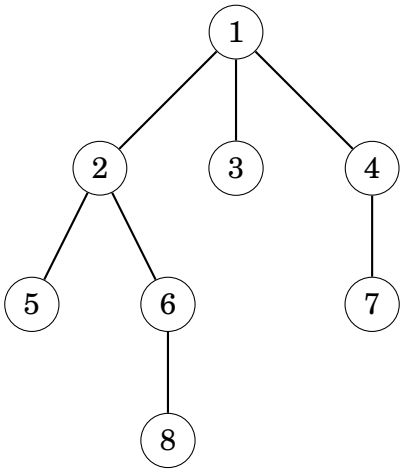
A **tree** is a connected, acyclic graph that consists of n nodes and $n-1$ edges. Removing any edge from a tree divides it into two components, and adding any edge to a tree creates a cycle. Moreover, there is always a unique path between any two nodes of a tree.

For example, the following tree consists of 8 nodes and 7 edges:



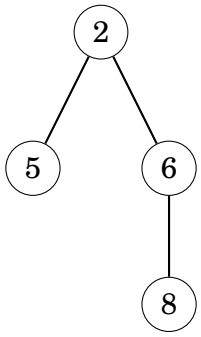
The **leaves** of a tree are the nodes with degree 1, i.e., with only one neighbor. For example, the leaves of the above tree are nodes 3, 5, 7 and 8.

In a **rooted** tree, one of the nodes is appointed the **root** of the tree, and all other nodes are placed underneath the root. For example, in the following tree, node 1 is the root node.



In a rooted tree, the **children** of a node are its lower neighbors, and the **parent** of a node is its upper neighbor. Each node has exactly one parent, except for the root that does not have a parent. For example, in the above tree, the children of node 2 are nodes 5 and 6, and its parent is node 1.

The structure of a rooted tree is *recursive*: each node of the tree acts as the root of a **subtree** that contains the node itself and all nodes that are in the subtrees of its children. For example, in the above tree, the subtree of node 2 consists of nodes 2, 5, 6 and 8:



Tree traversal

General graph traversal algorithms can be used to traverse the nodes of a tree. However, the traversal of a tree is easier to implement than that of a general graph, because there are no cycles in the tree and it is not possible to reach a node from multiple directions.

The typical way to traverse a tree is to start a depth-first search at an arbitrary node. The following recursive function can be used:

```
void dfs(int s, int e) {
    // process node s
    for (auto u : adj[s]) {
        if (u != e) dfs(u, s);
    }
}
```

The function is given two parameters: the current node s and the previous node e. The purpose of the parameter e is to make sure that the search only moves to nodes that have not been visited yet.

The following function call starts the search at node x:

```
dfs(x, 0);
```

In the first call $e = 0$, because there is no previous node, and it is allowed to proceed to any direction in the tree.

Dynamic programming

Dynamic programming can be used to calculate some information during a tree traversal. Using dynamic programming, we can, for example, calculate in $O(n)$ time for each node of a rooted tree the number of nodes in its subtree or the length of the longest path from the node to a leaf.