CIS 575. Introduction to Algorithm Analysis
Material for March 27, 2024

Union-Find Data Structures

The topic introduced in this note is covered in *Cormen*'s Chapter 19.

1 Union-Find Data Structures

1.1 Motivation

Given a collection of n elements, they are often divided into **classes**, such that each element belongs to *exactly one* class. It will therefore be useful to design a data structure that enables us to efficiently

- check if two elements are in the same class
- merge two classes into one.

We shall soon see one application: *Kruskal*'s algorithm will for each edge check if its end points are in the same connected component, and if not, merge the components.

1.2 Approach

The basic idea is that from each class, *one* of its members is designated as its **representative**. An implementation must support two operations:

UNION given two elements, merge their classes (if they are already in the same class, nothing happens).

FIND given an element, find the representative of its class. (Two elements will be in the same class iff they have the same representative.)

Accordingly, such a data structure is called a **union-find** structure. (We also need suitable operations for initialization, and adding elements, but we shall ignore those.)

In the rest of this set of notes we shall see how to implement a union-find structure. We shall first present a simple approach and then improve it through various steps.

1.3 Linear Representation

We may have a table that associates each element with its representative. For example, suppose there are 3 classes: the first contains B, D, E, I, J and L; the second contains A and F; the third contains C, G, H and K. If the first class is represented by B, the second class by F, and the third is represented by H, we get the table

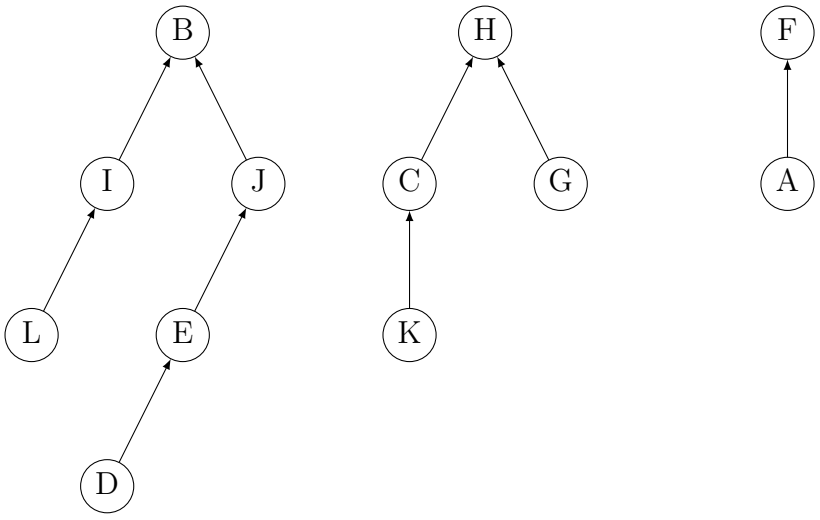
x	A	B	C	D	E	F	G	H	I	J	K	L
representative of x	F	B	H	B	B	F	H	H	B	B	H	B

Obviously, FIND can be computed in **constant time**. But UNION is much more difficult, as can be seen from the above example: if say the first class is merged with the third class then either the entries for B, D, E, I, J and L would need to be changed into H, or the entries for C, G, H and K would need to be changed into B. In general, the number of entries that need to be changed could be up to half the total number of elements, and even if only a few need to be changed, it will still take a linear search to find these entries. We conclude that UNION would have a running time in $\mathbf{\Theta(n)}$ which is hardly satisfactory.

1.4 Tree Representation

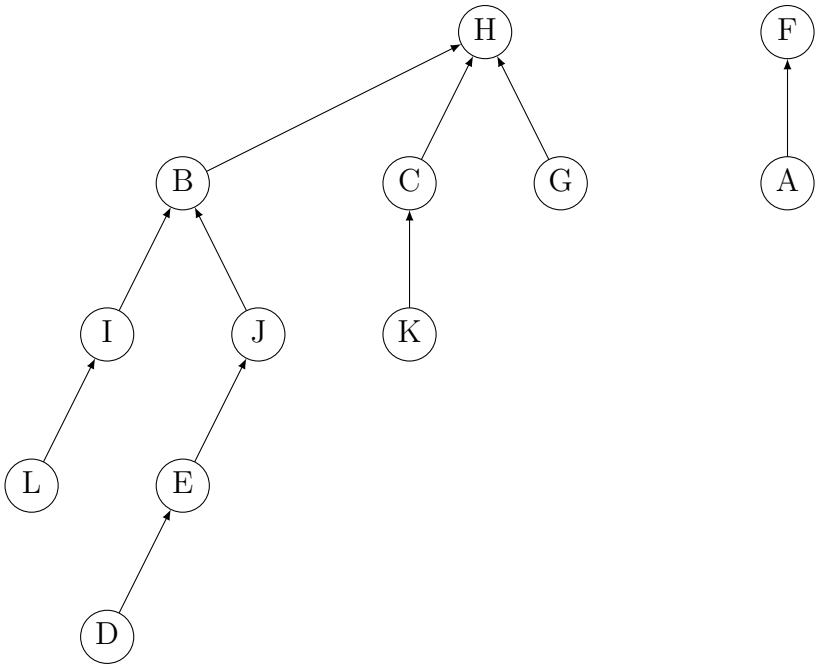
The idea is to create a **rooted tree** for each class (all these trees then form a *forest*), with the **root** being the **representative** of that class. To facilitate finding the root, edges are directed from a child towards its parent.

To illustrate this, let us show one way of representing the previous example:

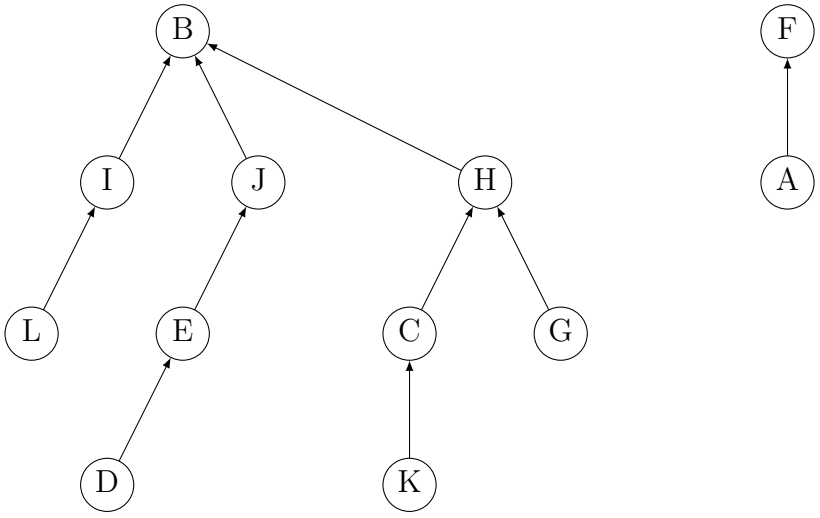


It is easy to see how to implement FIND: just follows the pointers, which takes time in $\mathbf{\Theta(h)}$ where h is the **height** of the tree.

For UNION, we must first find the roots; if the two roots are different we must make one become the child of the other. For example, to take the union of E and G, we may either let B become the child of H:



or let H become the child of B:



Since UNION involves doing FIND twice, followed by a constant time operation (adding one edge), also UNION runs in time $\mathbf{\Theta(h)}$.

All this may seem very good, until we realize that h could be close to n, for example if we keep merging with singleton classes and always make that singleton the new root. In the next notes, we shall see how to ensure $h \in O(\lg(n))$.