CS 42—Implementing Graphs (in Java)

Thursday, December 6, 2018

Representing graphs

Edge list: store a list of edges (where an edge is stored as a triple of source, destination, weight).

Adjacency list: for each source node, store a list of adjacencies (where an adjacency is stored as a pair of destination, weight).

Adjacency matrix: a table that, for each pair of nodes, stores the cost of an edge between those nodes.

Java generics

Java **generics** is a language feature that allows us parameterize our code with types (just a functions are a language feature that allows us to parameterize our code with values).

As an example, the Java library defines an interface `List<T>`, where `T` is a **type parameter** that describes the type of values stored in the list.

Restrictions on type parameters

- A type parameter must be an `Object`, not a primitive. Fortunately, there are `Object` versions of primitive types, e.g., `Integer`.
- Our code cannot create an instance of a type parameter.

Java `Maps`

A Java `Map` is like a Python dictionary.

The Java library defines an interface `Map<K, V>`, where `K` is a type parameter that describes the type of keys used in the map and `V` describes the type of values stored in the map.

To use Java `Map`s, we need to import them from the `java.util` package.

The Java library defines classes `HashMap<K, V>` (which we often use to store unordered data) and `TreeMap<K, V>` (which we often use to store ordered data).

`equals` and `hashCode`

So that we can store user-defined types in `Map`s, we follow some programming conventions:

- If two objects are `.equals`, their `hashCode` methods should return the same value.
- Therefore, if we define an `equals` method, we should define a `hashCode` method.
- Often, we let Eclipse define both these methods for us.

Next time: review!