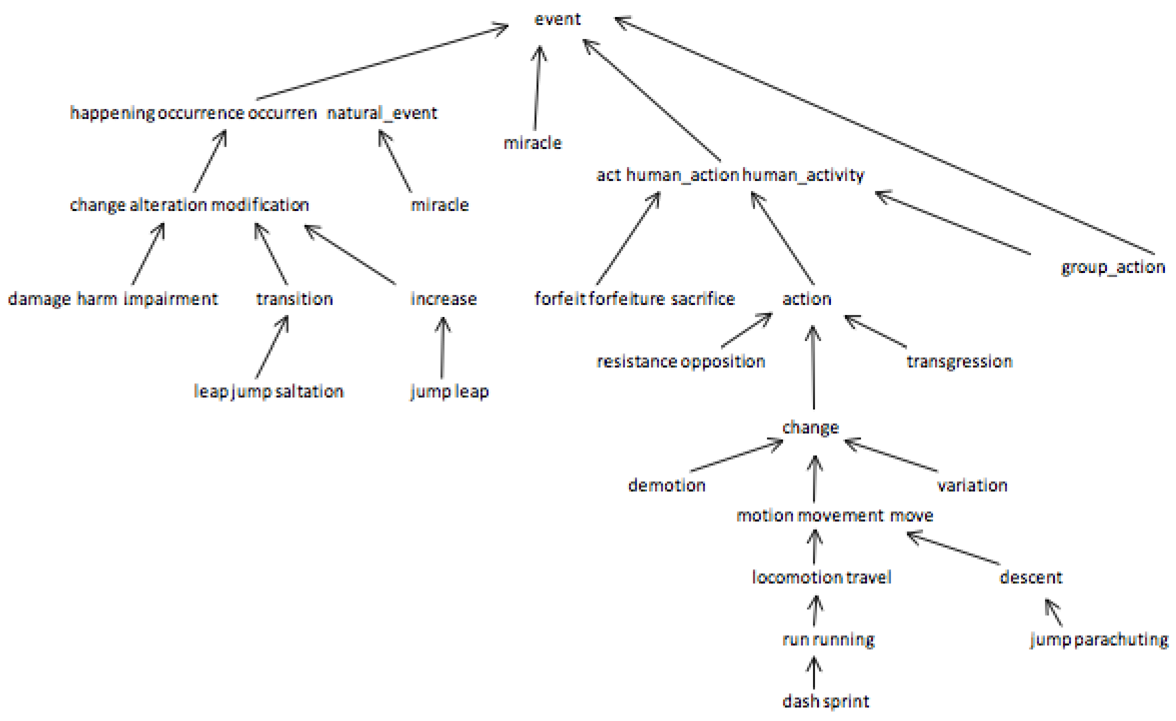
Programming Assignment 1: WordNet

WordNet is a semantic lexicon for the English language that is used extensively by computational linguists and cognitive scientists; for example, it was a key component in IBM's Watson. WordNet groups words into sets of synonyms called *synsets* and describes semantic relationships between them. One such relationship is the *is-a* relationship, which connects a *hyponym* (more specific synset) to a *hypernym* (more general synset). For example, a *plant organ* is a hypernym of *carrot* and *plant organ* is a hypernym of *plant root*.

The WordNet digraph. Your first task is to build the wordnet digraph: each vertex *v* is an integer that represents a synset, and each directed edge *v*→*w* represents that *w* is a hypernym of *v*. The wordnet digraph is a *rooted DAG*: it is acyclic and has one vertex—the root—that is an ancestor of every other vertex. However, it is not necessarily a tree because a synset can have more than one hypernym. A small subgraph of the wordnet digraph is illustrated below.



The WordNet input file formats. We now describe the two data files that you will use to create the wordnet digraph. The files are in *CSV format*: each line contains a sequence of fields, separated by commas.

- *List of noun synsets.* The file `synsets.txt` lists all the (noun) synsets in WordNet. The first field is the *synset id* (an integer), the second field is the synonym set (or *synset*), and the third field is its dictionary definition (or *gloss*). For example, the line
  - `36,AND_circuit AND_gate,a circuit in a computer that fires only when all of its inputs fire`

  means that the synset { `AND_circuit`, `AND_gate` } has an id number of 36 and it's gloss is `a circuit in a computer that fires only when all of its inputs fire`. The individual nouns that comprise a synset are separated by spaces (and a synset element is not permitted to contain a space). The S synset ids are numbered 0 through $S - 1$; the id numbers will appear consecutively in the synset file.

- *List of hypernyms.* The file `hypernyms.txt` contains the hypernym relationships: The first field is a synset id; subsequent fields are the id numbers of the synset's hypernyms. For example, the following line
  - `164,21012,56099`

  means that the the synset `164` (`"Actifed"`) has two hypernyms: `21012` (`"antihistamine"`) and `56099` (`"nasal_decongestant"`), representing that Actifed is both an antihistamine and a nasal decongestant. The synsets are obtained from the corresponding lines in the file `synsets.txt`.

  ```
  164,Actifed,trade name for a drug containing an
  antihistamine and a decongestant...
  21012,antihistamine,a medicine used to treat allergies...
  56099,nasal_decongestant,a decongestant that provides
  temporary relief of nasal...
  ```

WordNet data type. Implement an immutable data type `WordNet` with the following API:

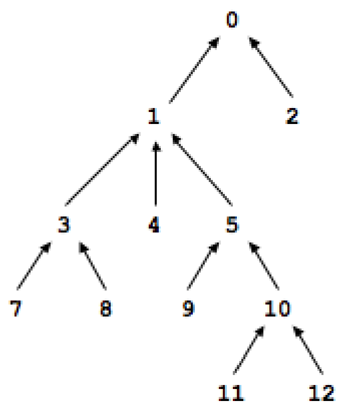
```
public class WordNet {
```

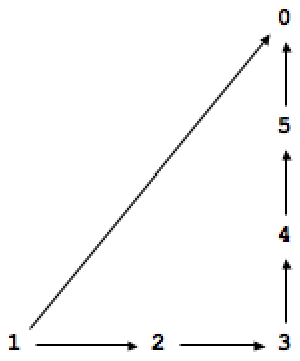
```
   // constructor takes the name of the two input files
   public WordNet(String synsets, String hypernyms)

   // returns all WordNet nouns
   public Iterable<String> nouns()

   // is the word a WordNet noun?
   public boolean isNoun(String word)

   // distance between nounA and nounB (defined below)
   public int distance(String nounA, String nounB)

   // a synset (second field of synsets.txt) that is the common ancestor
of nounA and nounB
   // in a shortest ancestral path (defined below)
   public String sap(String nounA, String nounB)

   // do unit testing of this class
   public static void main(String[] args)
}
```

Corner cases. All methods and the constructor should throw a `java.lang.NullPointerException` if any argument is null. The constructor should throw a `java.lang.IllegalArgumentException` if the input does not correspond to a rooted DAG. The `distance()` and `sap()` methods should throw a `java.lang.IllegalArgumentException` unless both of the noun arguments are WordNet nouns.

Performance requirements. Your data type should use space linear in the input size (size of synsets and hypernyms files). The constructor should take time linearithmic (or better) in the input size. The method `isNoun()` should run in time logarithmic (or better) in the number of nouns. The methods `distance()` and `sap()` should run in time linear in the size of the WordNet digraph. For the analysis, assume that the number of nouns per synset is bounded by a constant.

Shortest ancestral path. An *ancestral path* between two vertices *v* and *w* in a digraph is a directed path from *v* to a common ancestor *x*, together with a directed path from *w* to the same ancestor *x*.A *shortest ancestral path* is an ancestral path of minimum total length. For example, in the digraph at left (digraph1.txt), the shortest ancestral path between 3 and 11 has length 4 (with common ancestor 1). In the digraph at right (digraph2.txt), one ancestral path between 1 and 5 has length 4 (with common ancestor 5), but the shortest ancestral path has length 2 (with common ancestor 0).





SAP data type. Implement an immutable data type SAP with the following API:

```
public class SAP {

   // constructor takes a digraph (not necessarily a DAG)
   public SAP(Digraph G)

   // length of shortest ancestral path between v and w; -1 if no such
path
   public int length(int v, int w)

   // a common ancestor of v and w that participates in a shortest
ancestral path; -1 if no such path
   public int ancestor(int v, int w)

   // length of shortest ancestral path between any vertex in v and any
vertex in w; -1 if no such path
   public int length(Iterable<Integer> v, Iterable<Integer> w)

   // a common ancestor that participates in shortest ancestral path; -1
if no such path
   public int ancestor(Iterable<Integer> v, Iterable<Integer> w)
   // do unit testing of this class
   public static void main(String[] args)
}
```

Corner cases. All methods should throw

a `java.lang.NullPointerException` if any argument is null. All methods

should throw a `java.lang.IndexOutOfBoundsException` if any argument

vertex is invalid—not between`0` and `G.V() - 1`.

Performance requirements. All methods (and the constructor) should take time at most proportional to *E* + *V* in the worst case, where *E* and *V* are the number of edges and vertices in the digraph, respectively. Your data type should use space proportional to *E* + *V*.

Test client. The following test client takes the name of a digraph input file as as a command-line argument, constructs the digraph, reads in vertex pairs from standard input, and prints out the length of the shortest ancestral path between the two vertices and a common ancestor that participates in that path:

```
public static void main(String[] args) {
    In in = new In(args[0]);
    Digraph G = new Digraph(in);
    SAP sap = new SAP(G);
    while (!StdIn.isEmpty()) {
        int v = StdIn.readInt();
        int w = StdIn.readInt();
        int length   = sap.length(v, w);
        int ancestor = sap.ancestor(v, w);
        StdOut.printf("length = %d, ancestor = %d\n", length, ancestor);
    }
}
```

Here is a sample execution:

```
% more digraph1.txt                % java SAP digraph1.txt
13                                 3 11
11                                 length = 4, ancestor = 1
 7  3
 8  3                              9 12
 3  1                              length = 3, ancestor = 5
 4  1
 5  1
 9  5
10  5
11 10
12 10
 1  0
 2  0
```

```
7 2
length = 4, ancestor = 0

1 6
length = -1, ancestor = -1
```

Measuring the semantic relatedness of two nouns. Semantic relatedness refers to the degree to which two concepts are related. Measuring semantic relatedness is a challenging problem. For example, most of us agree that *George Bush* and *John Kennedy* (two U.S. presidents) are more related than are *George Bush* and *chimpanzee* (two primates). However, not most of us agree that *George Bush* and *Eric Arthur Blair* are related concepts. But if one is aware that *George Bush* and *Eric Arthur Blair* (aka George Orwell) are both communicators, then it becomes clear that the two concepts might be related.

We define the semantic relatedness of two wordnet nouns *A* and *B* as follows:

- *distance(A, B)* = distance is the minimum length of any ancestral path between any synset *v* of *A* and any synset *w* of *B*.

This is the notion of distance that you will use to implement

the `distance()` and `sap()` methods in the `WordNet` data type.

Outcast detection. Given a list of wordnet nouns $A_1, A_2, ..., A_n$, which noun is the least related to the others? To identify *an outcast*, compute the sum of the distances between each noun and every other one:

$d_i = \text{dist}(A_i, A_1) + \text{dist}(A_i, A_2) + ... + \text{dist}(A_i, A_n)$
and return a noun A_t for which d_t is maximum.

Implement an immutable data type `Outcast` with the following API:

```
public class Outcast {
   public Outcast(WordNet wordnet)         // constructor takes a
WordNet object
   public String outcast(String[] nouns)   // given an array of WordNet
nouns, return an outcast
   public static void main(String[] args)  // see test client below
}
```

Assume that argument to `outcast()` contains only valid wordnet nouns (and that it contains at least two such nouns).

The following test client takes from the command line the name of a synset file, the name of a hypernym file, followed by the names of outcast files, and prints out an outcast in each file:

```
public static void main(String[] args) {
    WordNet wordnet = new WordNet(args[0], args[1]);
    Outcast outcast = new Outcast(wordnet);
    for (int t = 2; t < args.length; t++) {
        In in = new In(args[t]);
        String[] nouns = in.readAllStrings();
        StdOut.println(args[t] + ": " + outcast.outcast(nouns));
    }
}
```

Here is a sample execution:

```
% more outcast5.txt
horse zebra cat bear table

% more outcast8.txt
water soda bed orange_juice milk apple_juice tea coffee

% more outcast11.txt
apple pear peach banana lime lemon blueberry strawberry mango watermelon
potato


% java Outcast synsets.txt hypernyms.txt outcast5.txt outcast8.txt
outcast11.txt
outcast5.txt: table
outcast8.txt: bed
outcast11.txt: potato
```

Analysis of running time (optional). Analyze the effectiveness of your approach to this problem by giving estimates of its time requirements.

- Give the order of growth of the *worst-case* running time of the `length()` and `ancestor()` methods in `SAP` as a function of the number of vertices *V* and the number of edges *E* in the digraph.
- Give the order of growth of the *best-case* running time of the same methods.

Deliverables. Submit `WordNet.java`, `SAP.java`, and `Outcast.java`. Also submit any other supporting files (excluding those in `stdlib.jar` and `algs4.jar`). You may not call any library functions other than those in `java.lang`, `java.util`, `stdlib.jar`, and `algs4.jar`.
