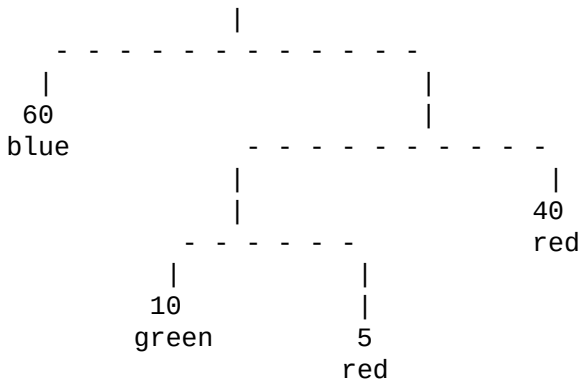
4 Methods for Self-Referential Data; Abstracting over Data Definitions

4.1 Methods for Self-Referential Data.

4.1.1 Problem: Mobiles

This problem continues the work on mobiles we have started during one of the earlier lectures. The file **MobileMethods.java** contains the data definitions, examples of data, and the method `countWeights`.

A. Make an additional example of mobile data that represents the following mobile (The number of dashes in the struts and lines represents their length):



B. Design the method `totalWeight` that computes the total weight of a mobile. The weight of the lines and struts is given by their lengths (a strut of length n has weight n).

C. Design the method `height` that computes the height of the mobile. We would like to hang the mobile in a room and want to make sure it will fit in.

Make sure you keep updating the *TEMPLATE* as you go along. (We have already started you on your way.)

4.1.2 Problem: Strings

For this problem start with the file **Strings.java** that defines a list fo `String`s.

Note: The following method defined for the class `String` may be useful:

```
// does this String come before the given String lexicographically?
// produce value < 0   --- if this String comes before that String
// produce value zero  --- if this String and that String are the same
// produce value > 0   --- if this String comes after that String
int compareTo(String that)
```

A. Design the method `isSorted` that determines whether the list is sorted in alphabetical order.

   Hint: You may need a helper method. You may want remember to the accumulator style functions we have seen in Scheme.

B. Design the method `merge` that consumes two sorted lists of `String`s and produces a sorted list of `String`s that contains all items in both original lists (including duplicates).

Again, make sure you keep updating the *TEMPLATE* as you go on.

4.2 Abstracting over Data Definitions.

Review of Designing Methods for Unions of Classes.

A file in a computer can contain either a text, or an image, or an audio recording. Every file has a name and the owner of the file. There is additional information for each kind of file as shown in the program **Files.java**.

Download the file and work out the following problems:

A. Make one more example of data for each of the three classes and add the tests for the method `size` that is already defined.

   Now design the methods that will deal with the files:

B. Design the method `downloadTime` that determines how many seconds does it take to download the file at the given download rate.

   The rate is given in bytes per second.

C. Design the method `sameOwner` that determines whether the owner of this file is the same as the owner of the given file.

Save the work you have done. Copy the file and continue.

Abstracting over Data Definitions: Lifting Fields

Save your work. Possibly start a new project and import the file into it. Alternatively, save the a copy of the file you have been working on in another folder.

Look at the code and identify all places where the code repeats — the opportunity for abstraction.

Lift the common fields to an abstract class `AFile`. Make sure you include a constructor in the abstract class, and change the constructors in the derived classes accordingly. Run the program and make sure all test cases work as before.

Abstracting over Data Definitions: Lifting Methods

For each method that is defined in all three classes decide to which category it belongs:

A. The method bodies in the different classes are all different, and so the method has to be declared as `abstract` in the `abstract` class.

B. The method bodies are the same in all classes and it can be implemented concretely in the `abstract` class.

C. The method bodies are the same for two of the classes, but are different in one class — therefore we can define the common body in the `abstract` class and override it in only one derived class.

Now, lift the methods that can be lifted and run all tests again.

Note: You can lift the method `sameOwner` only if you change its contract. Do so — make sure you adjust the test cases accordingly.