A Hash Table Implementation of the Sack ADT

We will now present another approach to implementing the sack ADT, using the *hash table* idea.

Suppose we wanted to store a bunch of student objects, each with a 9 digit student number. In our previous approaches to this, we used an array and stored students in consecutive positions, starting from 0, and found different efficiencies for the different sack methods with our two different approaches (keep the students sorted by their student numbers, or don't). Now we consider a radically different approach: create an array with 9 billion spots in it, and store `null` in a position to indicate that the student with that student number is not in the sack, and store a reference to the student object in a position if the student with that student number is in the sack. In other words, the position in the array corresponds exactly with the student number of the one and only student that can be stored in that spot in the array.

⇐ Perform efficiency analysis for the sack methods if this approach is used, where the representative operation is an array store.

⇐ Other than the horrible inefficiency (imagine storing a million students—rather a large school—with this scheme) of one of the sack methods, what else is very bad about this approach?

We can now appreciate the *hash table* idea: store items in a large array with the position in the array where an item must be stored corresponding directly to the item's key. But, instead of doing a completely direct correspondence like in the previous approach, we use a function, known as a "hash function," that takes any key and computes, in a randomizing sort of way, the position in the array where the item with that key belongs.

For example, we might use an array with 11 spots and use as the hash function (where mod is the common mathematical name for the remainder function—% in Java)

$$h(k) = (7 \cdot k + 13) \text{mod } 11,$$

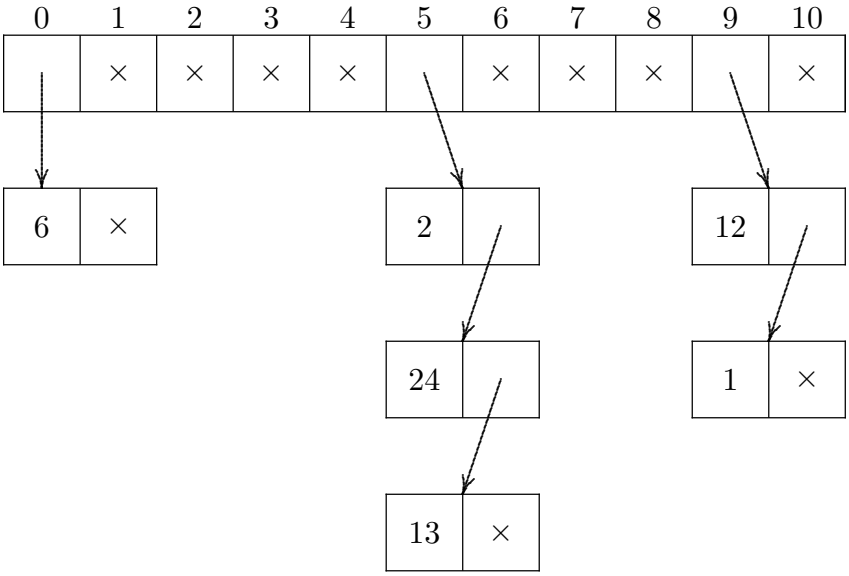
where k is the key, viewed as simply an integer here.

This approach is appealing because there is no apparent $O(n)$ time required to locate an item—we simply compute the hash function of its key and look in that position in the array, which takes $O(1)$.

⇐ Compute the hash function of a bunch of different keys and observe the obvious flaw with this approach.

Our final idea for this approach to the sack is to keep, in each slot in the array, a simple linked-list containing all the items whose keys map to that position in the array.

Here is how this data structure will look, after adding items 2, 12, 6, 1, 24, and 13 (blurring the distinction between an item and its key) in that order:



⇐ Start with an empty array and add the given keys to understand this picture. Add some more random keys. Then think about doing `find`, `get`, and `remove` operations on this data structure. Note the obvious problem of two different blocks wanting to be in the same position.

To handle collisions, we need each position in the area to be able to hold a collection of blocks, namely all the blocks that hash to that position. We could use, say, an `ArrayList` in each spot, but it's more efficient to use the linked-list idea, which is great for automatically adjusting to have lists of different sizes.

⇐ Note why the hash table approach is appealing by analyzing the time to add, find, remove, and get with this approach, making the idealistic assumption that blocks don't collide. Also consider the time efficiency for traversing the sack with this approach. Then consider the time efficiency in the worst case, namely that all the blocks we add happen to hash to the same cell.

⇐ Suppose we have n items stored in a hash table with n cells—that is, suppose we have an array that is exactly the size it needs to be to hold the most items in the sack. Assuming that the hash function does a good job of randomly assigning keys to positions in the array, what would you expect to be the average number of items stored in a position of the array?

⇐ Create a full implementation of the sack based on a hash table and test it with applications that we used to test our earlier sack implementations.
