The Sack ADT

Our first project was to implement the *list abstract data type* (ADT) using an array. We approached this as a matter of looking at the predefined Java class `ArrayList` and deciding what methods we wanted to have, but another way of looking at this is that we designed the list ADT to have the methods `append` (add on the end), `insert` (add in a specified position), `remove`, `get`, `replace` (named `set`), and `size`.

> Just the method headers for some idea of how to store and access a collection of items, without any notion of how the class is implemented, is known as an ADT.

Note that we had an implicit idea of a list, namely that the items in the list are listed in order: first, second, third, and so on.

Now we want to look at a related but completely different tool for working with a collection of items known as the *sack ADT*.

Some situations require a data structure that maintains a collection of items that are in a strict sequential arrangement, where the order of the items matters for some reason. For example, a text editor really must keep the items—the lines of the document, or the individual symbols in the document—in a strict order (sorting a list of lines or symbols is a bad idea, by the way!).

But, for many situations, we simply need a collection of items, with no concept of them being arranged in some sequential order. We will refer to this type of collection as a "sack" (like a sack of groceries—we just toss items in and later take them out, and we don't think of them as being organized any more than that).

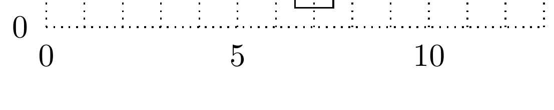
$\Leftarrow$ Consider the physical sack that I brought to class. Explore what operations make sense for such a collection of objects (we'll refer to the particular objects in the sack as "blocks").

Motivating Example for Designing the Sack ADT

In order to continue designing the sack ADT, we will look at a simple situation where we naturally have a collection of items that do not need to be sequentially arranged.

Suppose we want to implement a tile-based 2D computer game (or similar application) where all of 2D space as partitioned into a uniform grid of size 1, and each game object is viewed as sitting at the intersection of two grid lines.

Here is a picture of such a world with 8 items in it:



In this example, every item in the sack has a *key*, namely the x and y coordinates of its center point. And, we want to only allow at most one item, known as a "block" (like a child's building block), at each location, to simulate reasonable physics where objects can't share the same space.

In the following discussion, each paragraph is labeled by the name of the method in the Sack ADT that is suggested by the scenario.

find: Typically in this sort of game a player or artificial intelligence will want to move a block from its current location to some other location. In order to do this without violating the law of physics, we have to ask the sack "is there a block at this particular location?" For example, the block `A`, with key $(5, 3)$ might want to move one tile to its right, so it would ask the sack if there is a block at $(6, 3)$. Upon being told that there is no block there, `A` could safely and cheerfully move there.

get: If `A` keeps moving to its right, eventually it will be told that there is a block with key $(11, 3)$. Actually, we want to have a reference to the block with that key so we can ask if it is something good to eat.

remove: If block `B` is good to eat, then `A` might decide to eat that block, so then we want to be able to ask the sack to remove the block with key $(11, 3)$.

add: After eating `B`, `A` might decide to build some new block at that location, so we want to be able to add an item to the sack.

> Actually, for `A` to move, we have to remove the block with key $(5, 3)$ and add a new block (really just `A`, but with a different key, namely $(6, 3)$).

construct: Of course, at the beginning of the game, we will probably want to read a data file that tells us where all the blocks of various kinds are located, so we'll need to begin by constructing an empty sack, so that we can then proceed to create new blocks and add them to the sack.

size: When we want to save the game to a data file (so the user can come back to it later), we would need to know how many blocks the sack contains.

traverse: In order to draw all the blocks in the sack on screen (even if they are out of sight), we want to be able to have the sack give us a reference to each block, in some arbitrary order. Note that this is different from finding a block with a certain key—the world has vastly more keys than it does blocks.

$\Leftarrow$ Go back to the physical sack and interpret all these Sack ADT operations in that context.

The Sack ADT

Now we want to finish the design of the sack ADT by writing Java method headers that will specify very precisely the details of the methods we've identified. Note that we will not write the method bodies at this point, nor will we decide what instance variables to use in the class—that will come shortly.

⇐ Create a class named `BlockSack` that has just the method headers. Assume that the type of items stored in the sack is `Block`, and assume that the `Block` class has a method `getKey()` that returns the `Key` for the block, which is its (x, y) center coordinates. In fact, we should go ahead and start implementing the `Block` and `Key` classes.

> Maybe we should be using type parameters to make the kind of objects stored in the sack be more generic, but we won't confuse ourselves with that—yet.

Three Array-Based Approaches to Implementing the Sack ADT

Now we want to implement our specific example of the Sack ADT, namely the `BlockSack`, in two different ways, both of which use an array to hold the items in the sack.

⇐ To have a specific application to work with, we will create a simple game, `BlockGame1`, where the application first reads the block information from a data file, and then lets the player move a special block around, with the player collecting points by consuming delicious green blocks, and dying if the player tries to land on a poisonous red block. As an exercise, let's write this application just knowing the sack interface, before we've implemented the `BlockSack` class.

⇐ Now, let's implement the sack ADT using our first approach, in a class named `BlockSack1`, namely by using an `ArrayList<Block>` object to hold the blocks, where added blocks go on the end and we keep the blocks in chronological order. After designing and implementing the instance variable(s), and writing the bodies of the methods, and testing our work in our game, we should do efficiency analysis of all the methods.

⇐ Now create a class `BlockSack2` that uses an array to store the blocks, but now doesn't try to keep the blocks in chronological order. If we are clever, we'll be able to perform all the methods in $O(1)$ time!

The third array-based approach to implementing a sack is to keep the items in sorted order. You will implement that approach in Exercise 2. But before you can do that, we need to develop a slightly clever algorithm for searching a sorted list efficiently.

The Binary Search Algorithm

We saw in our earlier work that searching a sack for an item with a particular key takes $O(n)$ time. In this little section we will study how this can be done more efficiently using the *binary search* algorithm.

The idea of the binary search algorithm is simple: if we have a list of items that are comparable in some way, and those items are sorted into increasing order, then we can find a given key by first looking in the middle of the list. If the middle item has the target key, then we return the position of the middle item. Otherwise, we *recursively* binary search either the lower half of the list or the upper half of the list, depending on whether the target key is smaller than or larger than the key of the middle item.

$\Leftarrow$ Using our favorite number cards, face down on the chalk tray, and sorted in increasing left-to-right order, try out the binary search tree idea.

$\Leftarrow$ What is the worst-case number of comparisons required to find a target in a sorted list, or to realize that it's not in the list, assuming as usual that the list has n items?

$\Leftarrow$ Create a class named `BinSearchTest` that has a static `binarySearch` method and a `main` method that creates a sorted list of blocks (stored in an array) and tests the binary search method on that list interactively, asking the user repeatedly for a key to be searched for. The binary search method should use a cheap trick: if the target key is actually found in the list, then the method should return the position of the item with that key, and if the target key is not in the list, then the method should return the position where the item with that key should be inserted.

Exercise 2 [10 points] (due 2 weeks from when we reach this point in class—see the text file `dates` for all defined due dates)

At the course web site, in the `Sacks` folder, you will find the class `BlockGame1` that we created in class, using the implementation `BlockSack2` of the sack ADT.

Your job on this exercise is to create your own implementation, in the class `BlockSack3`, of the sack ADT following this approach: keep the items sorted (by adding items where they belong) and use binary search to find a target (for both adding and finding).

Note that you will need to take the version of binary search that we did in class and transform it to work in the `add` and `find` methods of your new class—it will need to work with blocks rather than with strings. For everyone's convenience, you must put this method in your `BlockSack3` class (you can chose whether to make it a static or an instance method, with suitable adjustments).

Also, you will need to modify `BlockGame1` in a tiny way so that it uses a `BlockSack3` object to hold its blocks.

In addition to—and probably before—you test your `BlockSack3` class with the block game, you should do *unit testing* of your `BlockSack3` class—write a `main` method in the class that will do simple tests of all the methods of the class. In addition to adding the test driver, you might want to add an instance method that will nicely display the current contents of a sack, for debugging purposes. This method can simply list the keys of the blocks in the sack, in the order they appear in the array.

Note that your sack implementation might work adequately for the block game, even though it has problems, and that I will probably detect those problems with the unit testing that I do when you turn in your work!

When you are done, email me at `shultzj@msudenver.edu` with just your file `BlockSack3.java` attached. Your class must work with the `BlockGame1.java`, `Block.java`, and `Key.java` classes as written during class time (including a `compareTo` method in the `Key` class that isn't needed for the first two approaches to implementing the Sack ADT that we've studied).

Proving Correctness of Binary Search

Our view of the binary search algorithm was a littl emuddled, and our code isn't actually all that elegant, so let's do something different: let's try to *prove* that our binary search algorithm is correct—that it always works.

Of course, since we don't really have an algorithm, this technique—trying to formally prove that an algorithm works—gets merged with trying to invent the algorithm.

When doing recursion, what we'd like to have is an *invariant*, which is a fact about the situation whose truth is preserved by the recursive calls made by the algorithm.

Our first idea would be to use the invariant

$$a[f] \le t \le a[l],$$

where the method call is `binarySearch( a, t, f, l)` with a being the array, t being the target, f being the index of the first cell in the range, and l being the index of the last cell in the range.

This invariant turns out not to work, as follows. Suppose we make the call, with the fact being true, and then we compute m as the index of the middle cell, find out that $a[m] < t$, and therefore want to make the recursive call `binarySearch( a, t, m+1, l )`
Now when we try to prove that the supposedly invariant fact is satisfied, we can't!

$\Leftarrow$ To see this, create a simple example where the fact fails, namely where $a[f] > t$ (noting that the new value of the parameter f is the old value $m + 1$).

Since the first guess at an invariant didn't work, let's try the invariant

$a[f - 1] \le t$ or $f = 0$, and $t \le a[l + 1]$, or $l = n - 1$

(where n is the number of items in the entire array).

So, we assume that this invariant holds for a call `binarySearch(a,t,f,l)`, and try again to prove that the invariant holds on the next call, where we'll first consider the case that $f + 1 < l$ (so that the range has at least three cells).

Okay, if we compute m as usual and if $a[m] = t$, the method returns, saying that the target was found in position m, and since no further call is made, the invariant is good.

If $a[m] < t$, then the recursive call to `binarySearch( a, t, m+1, l )` is made. The invariant for this recursive call, using the current variable meanings, is that $a[m+1-1] \le t$ or $m + 1 = 0$, and $t \le a[l + 1]$ or $l = n - 1$. The second part of the invariant is trivially true because it was for the original call. The first part is true because $m + 1$ can't be 0 and the algorithm made the call it did because $a[m] \le t$ (in fact, $a[m] < t$).

$\Leftarrow$ Similarly, check that the invariant holds in the case that $t < a[m]$.

$\Leftarrow$ Now examine the case that the recursive call is made, with the invariant satisfied, with $f + 1 = l$. Specifically, we need to check that either the method returns with the correct result, or the invariant is satisfied on the next call.

⇐ Finally, examine the case that $f = l$ when the call is made, with the invariant satisfied. Note that the motivating idea for our algorithm has totally collapsed here, and we will only be considering base cases.

⇐ Now we should be able to finish implementing the algorithm, with confidence. Note that the attempt to prove correctness of the algorithm gave us the correct base cases.

Some Practice for Test 1

Now, partly to practice for Test 1 (which will be given on Thursday, September 26), and also to continue with standard material of the course, you will be asked to work in small groups on the following problems.

The Stack ADT

The *stack* ADT is modeled on the idea of a pile, or stack of items, like a pile of shoeboxes, where you can, for safety reasons, only work with the top item in the stack. Here are the official (for us—other people use variations of these) methods for the stack ADT, expressed as the beginnings of a Java class (where for clarity we assume the items are strings):

```
public class StringStack
{
  // construct an empty stack
  public StringStack(){}
  // place the given item on top of the stack
  public void push( String s ){}
  // return a reference to the top item
  public String peek(){}
  // remove the top item from the stack
  public void pop(){}
  // return the number of items in the stack
  public int size(){}
}
```

Small Group Exercises on Stacks:

1. Discuss which list methods correspond most closely to these stack methods.
2. Write—on paper or on a computer if someone has one handy—instance variables and method bodies for the `StringStack` class, assuming that an array named `a` is used to store the items in the stack, with position 0 being the top of the stack, and `n` holding the current number of items in the stack.
3. Perform efficiency analysis of each of the stack methods, using as the representative operation storage into the array.
4. Do the same thing as for problem 2, but this time store the *bottom* item of the stack in position 0.
5. Do the same as for problem 3, but with the implementations from problem 4.

6. Write a static method that takes a stack as its input argument and removes all the items in that stack except the bottom one.
7. Write a static method that takes a stack as its input argument and returns a new stack that contains the input stack in reversed order, and leaves the input stack the way it was at the beginning.

The Queue ADT

The *queue* ADT is modeled on the idea of a line (or, in England, a "queue") of items, like people waiting in line at a grocery store cashier, where you can, for safety reasons, only remove an item from the front of the queue, and can only add an item at the rear of the queue. Here are the official (for us—other people use variations of these) methods for the queue ADT, expressed as the beginnings of a Java class (where for clarity we assume the items are strings):

```
public class StringQueue
{
  // construct an empty queue
  public StringQueue(){}
  // place the given item on the rear of the queue
  public void add( String s ){}
  // return a reference to the front item
  public String peek(){}
  // remove the front item from the queue
  public void remove(){}
  // return the number of items in the stack
  public int size(){}
}
```

Small Group Exercises on Queues:

Figure out two reasonable approaches to using an array to implement the queue ADT, and for both, implement the methods and analyze their efficiency.
