Arrays vs. lists

Last lecture we used an array data structure to keep track of a set of objects of some class, in particular, to keep them in some order. We saw how to insert and remove an element from an array, and we noticed that this involved repeatedly shifting the elements of the array either to make room (add) or to fill the hole (remove). This shifting can involve a lot of work if the array has many elements. This is one disadvantage of using an array.

A second disadvantage of arrays is that when you create the array you need to specify how many elements it has (see p. 98 of text GT). Often, however, you don't know in advance how many elements you will have. You can always declare the array size to be a huge number, just to be safe, but this will waste computer memory if you only need a small array.

Today we will look at another Data Structure for keeping track of a set of objects of some class, called a *linked list*. Before we look at how linked lists work, though, let's make sure we understand arrays as best we can. Suppose we wish to define a set of objects that belong to some class. If we construct these objects using an array, for example,

```
GameEntry[]  entries = new GameEntry[maxEntries];
```

then `entries` is an array of reference variables `entries[0]`, ..., `entries[maxEntries-1]` which each point to an object. Reference variable `entries[i]` is initialized so it contains the address of the `i`-th `GameEntry` object.

The reference variables `entries[i]` sit next to each other in memory.[1] The newly constructed objects themselves also sit next to each other in memory (though possibly far away from the `entries` array of reference variables). Each object has an address that is a *constant* offset from the previous one. The offset is the constant number of bytes of memory needed by each object. When the array is constructed, the `entries[i]` variable contains an address, which is a *base* address (`entries[0]`) + `i` times the constant size of an object, i.e.

$$address = base\_address + i * object\_size.$$

For example, if `entries[0]` contains 3240 (the address of the first object) and each object is 76 bytes, then `entries[1]` will contain 3240 + 76 = 3316, and `entries[2]` will contain 3292, etc.

The above just describes what happens when the array is constructed. *Subsequent statements* may modify what an `entries[i]` variable refers to. For example, if we wish to remove an entry, then we would adjust the references by shifting them down (just as last lecture, we removed a score by shifting the scores down). We could also create a new `GameEntry` object, and reference it, e.g.

```
GameEntry  entries[2] = new GameEntry();
```

This new object might be at some faraway address in memory, i.e. not near the original objects that were created (next to each other).

[1] The size of each reference is the number of bytes needed to to specify the address of an object. Typically this has been 4 bytes i.e 32 bits, although there are many 64 bit machines these days.

Singly linked lists

A linked list works very differently. Here, all the objects are created one-by-one and could be anywhere in memory. We do not know in advance how many objects we will have, so we do not know how many reference variables we will need.

Suppose that (like with an array) we would still like to order the objects, possibly by ranking them according to some value of one of the fields, or possibly by the order in which they were created, or some other criteria. To order objects in a list, the minimal thing we need to do is to specify which object is first and then, for any object, we need to specify which comes next.

Consider a class `SNode` (where S is for 'single' – you will see why shortly) which has some set of fields and methods, including a field `next` which is a reference to the next object. For simplicity, in the class diagram below I have only included one field `x` which is of type `float`. In general, `SNode` contains all the fields and methods that you would like for your class (the same fields and methods that you would use if you were storing your objects in an array), plus an extra field `next` and its getter and setter methods.

```
----  SNode -------------
|      :                |
|  float  x             |
|  SNode  next          |
|                       |
|  SNode()              |
|  float getX()         |
|  void setX( float)    |
|  SNode getNext()      |
|  void setNext( SNode) |
-------------------------
```

We also define the class `SLinkedList`. This class keeps track of the first and last element of the list, by maintaining reference variables `head` and `tail`. (It might have been simpler to name them `first` and `last`, but I will go with what is in the GT textbook.)

```
-------------------------
|  SLinkedList          |
|                       |
|  SNode head           |
|  SNode tail           |
|                       |
|  SLinkedList()        |
|  addFirst(SNode)      |
|  addLast(SNode)       |
|  SNode removeFirst()  |
-------------------------
```

How do these methods work? Let's first look at `addFirst`, which adds a node to the front of the list. This method has one parameter which is of type `SNode`, namely a reference to the node to

be added. Here's what the method does:[2]

```
 void addFirst( node ){
    node.setNext(head);
    head = node;
 }
```

The method for adding a node to the end of the list is similar.

```
 void addLast( node ){
    node.setNext(null);   // It probably has this value already, but
                          // we make sure.
    tail.setNext(node);
    tail = node;
 }
```

And the method for removing the first node is:

```
SNode removeFirst(){
//    if (head == null)  then throw Exception, else...
    SNode tmp = head;
    head = head.getNext();
    tmp.setNext(null);
    return tmp;
 }
```

[2]In the GT textbook (p. 117), they also keep track of the number of nodes in the list using a size variable. I will ignore this here.