CS61B Note 3: Values and Containers

Kevin Moy

Abstract

In this note, we analyze the foundations of data structures in this class: values and containers. We will also examine our first data structure, the linked list, as well as various ways to implement it in code.

1 The Fundamentals

We are going to build our knowledge of data structures here with the absolute basic building blocks: Values, Containers, and Pointers.

1.1 Values

First, we begin with the **value**. The value is like an element- it's an entity that cannot be further reduced, not made from anything else. It's just a thing that exists, and **can never change**.

We can easily think of some examples.

- The integer 3.
- The character 'c'.
- The boolean true.
- A pointer (that points to something in memory).

As for CS61B, the only values we really need to know are **numbers, booleans, and pointers**.

1.2 Containers

Containers, as the name suggests, contain stuff.

Simple containers contain values. Examples are:

- Variables.
- Array elements (individual).
- Fields (or attributes) of a class.

- Parameters of functions

Structured containers contain other containers, or nothing (an empty object can be created!). More examples:

- Probably the easiest to think of is the array object, which contains a number of other individual elements (simple containers).
- A class object that you generate yourself: More than likely, an object will have multiple fields, which as stated above are simple containers.
- A LinkedList node. As we'll learn later, a single linked list node (object!) consists of a head and tail field.

So just think of simple containers as boxes containing a single item, and structured containers as the UPS truck than can carry an arbitrary number of these boxes.

1.3 Pointers

Finally, we hit pointers. Pointers are **references** to containers - they "point to" containers. In reality, they are actually *references to memory* of the container they are pointing to. What does this mean?

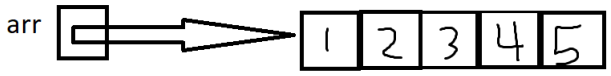
To demonstrate, let's create a quick integer array.

```
int[] arr = new int[5];
for (int i = 0; i < 5; i++) {
   arr[i] = i;
```

NOTE: If you need to quickly make an array and are too lazy to write a 2 line for loop, you can simply write the array's elements out by hand with curly brackets:

```
int[] arr = {1, 2, 3, 4, 5};
```

The variable "arr" above actually refers to the **pointer** to the array, not the array itself. We can see this in the box and pointer diagram visualization.



KEY: Remember that "arr" is actually a simple container (with a pointer), while the array being pointed at is a structured container! Hence the box around the pointer.

Essentially every programming language uses pointers. Why? The short version: when you create an object, it gets created and then has to be placed in a computer's memory somewhere. In order to use that object to do stuff, we need

a reference to the memory address that holds that object. This is what our pointer ("arr") is. [1]

In fact, in Java, every single thing uses a pointer, whether object or value! However, as you also learned in 61A, a pointer to an integer most definitely does not have the same properties as a pointer to an array. We can generalize this and say different things happen with pointers to simple and structured containers. The difference has to do with *what's copied and what's not.*

Before I explain, let's look at a little example code.

```
int a = 4;
int b = a;
a = 3;

int[] arr1 = {1,2,3}
int[] arr2 = arr1;
arr1[3] = 4;
```

Check out Philip Guo's wonderful Java visualizer on the code above. Is there a difference in pointers for ints and int arrays?

Absolutely. Notice that pointers `a` and `b` effectively point to copies of 4, while pointers `arr1` and `arr2` point to the same object. We checked this by editing the pointed-at object and observing what the pointers pointed to.

Thus, **pointers only point to structured containers**. We don't need pointers to point to simple containers: copies are automatically made, so we don't really need the pointers to "keep track" of the container state.

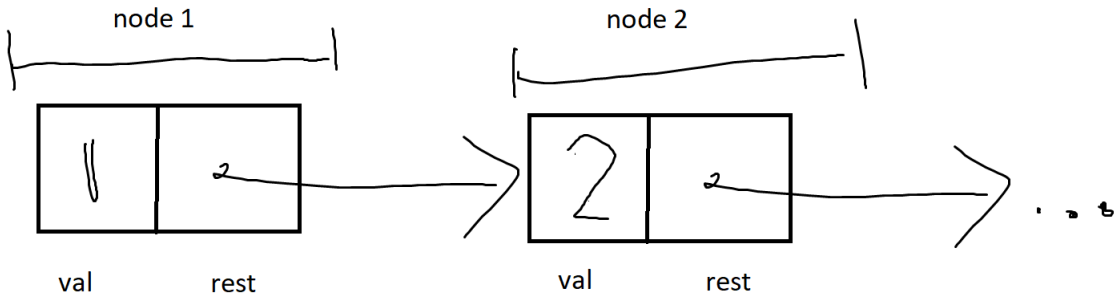
In Java, **assignment (`b = a`) always copies values into simple containers**- whether it be something like an integer or a pointer to an object. Remember that pointers themselves (NOT what they point to!) are values! This is a rule that applies universally.

2 Making Our own Types

In order to make our own special kinds of objects, with whatever attributes and methods, we need to write a class for it.

We'll demonstrate this by creating a linked list of integers, called an **IntList**. It's important to remember that a linked list is actually just a single **node**, consisting of a value and a pointer to the rest of the linked list (the next node in the linked list).

[1] CS61C will teach you a lot more about this.



That first node above is actually a Linked List in itself, and it points to another linked list, the second node. We can tell that this implementation will force an IntList to have yet another IntList as one of its attributes. Because of this inherent repetition, a Linked List (IntList) is a **recursive data structure**.

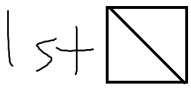
Let's create the IntList class! According to the diagram above, an IntList is represented by a node containing a value and a pointer to another IntList. We can easily represent this in a Java class (taken from official lecture slides):

```
public class IntList {
   // Constructor
   /** List cell containing (HEAD, TAIL). */
   public IntList(int head, IntList tail) {
      this.head = head;
      this.tail = tail;
   }
   // Names of simple containers (fields/attributes)
   // WARNING: public instance variables usually bad style!
   public int head;
   public IntList tail;
}
```

Notice that the attributes or fields specify what an IntList (node) consists of, and the constructor allows its creation. Now let's see how we'd use this class to create an IntList in real time (main method).

```
IntList lst;
```

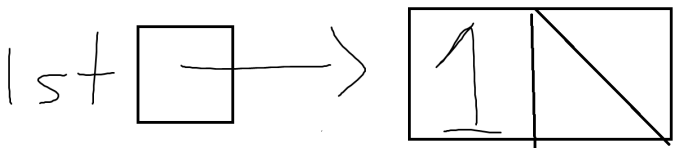
Whenever we just create an uninstantiated variable all we do is make a pointer that doesn't point to anything. So we'd visually have this below:



Now in order to actually create our IntList, we have to use the `new` keyword, which calls upon our `IntList` constructor and creates our object with our spec-

ified parameters.

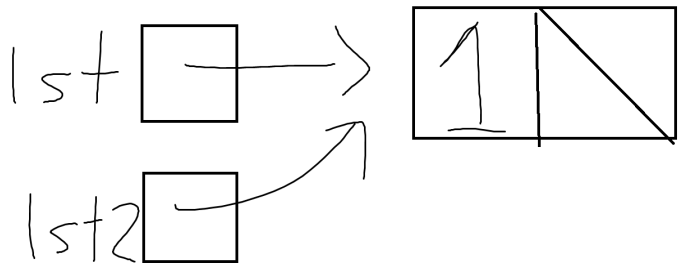
```
lst = new IntList(1,null);
```



Now we have something to work with.

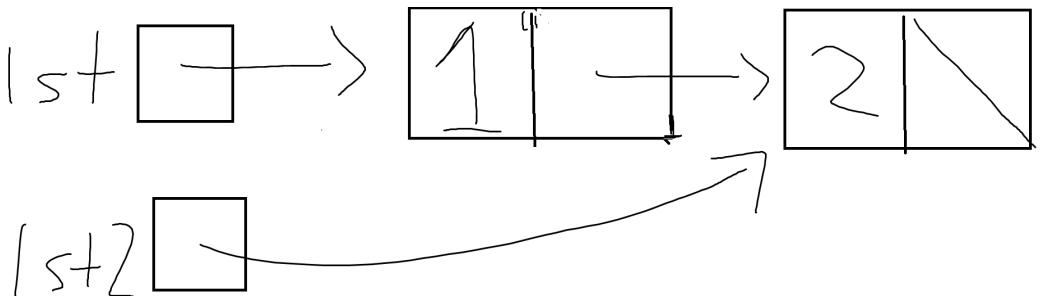
```
IntList lst2 = lst;
```

Remember that `lst` itself is a simple container containing a pointer. Thus, when we assign `lst` to another IntList variable, our pointer to the node above is just copied.



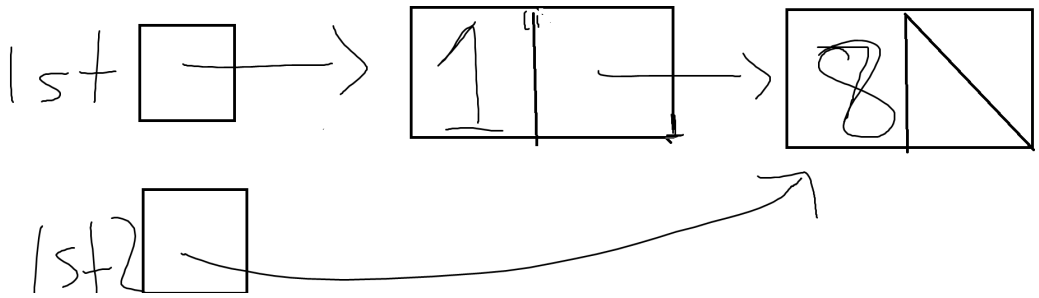
Now how would we make `lst` longer? Right now, the `tail` field of our IntList points to null- we simply create another IntList and point `tail` to it.

```
lst2 = new IntList(2, null);
lst.tail = lst2;
//Alternatively we could simply write lst.tail = new IntList(2, null);.
    However, we wouldn't have a pointer pointing directly to our newly
    created node, and would have to traverse to it to do stuff with it,
    which might be annoying.
```



Finally, note that we can directly change the values of any node we want in this linked list, since they are all objects with public fields (Note that 61B Style requires getter/setter methods and private fields though).

```
lst.tail.head += 7;
```



Many students have some trouble understanding what points where. Just remember this: pointers by themselves are completely useless. **It's what they point to that matters.** For example, `lst.tail` isn't the arrow in the first node, but the second node, now with value 8 and a null tail, that it points to. Thus, we view lst.tail as an IntList object in itself, from which we edit its `head` field by adding 7 to it.

3 Destructive vs. Non-Destructive Functions

Functions can act on data structures that are passed in as parameters. They can act destructively or non-destructively:

- **Destructive** functions modify the data structures passed in. Usually don't return anything (void).
- **Non-destructive** functions DO NOT edit the passed-in structures and create and return a copy of the structure instead. Such methods are generally "safer" to use because the original copy is saved in case of a fuck-up.

The example IntList function given in lecture was the `incrList` method, where

we pass in an IntList and an integer and increment every element of the IntList by that integer.

3.1 A Non-Destructive Approach

```
static IntList incrList(IntList P, int n) {
   return /*( P, with each element incremented by n )*/
}
```

Notice that since this function returns an IntList, this is probably non-destructive.

It is through the implementation of such functions that you see the power of understanding how to program recursively. Compare the recursive and iterative implementations (respectively) below.

```
/**Recursive implementation. */
static IntList incrList(IntList P, int n) {
   if (P == null) {
      return null;
   } else return new IntList(P.head+n, incrList(P.tail, n));
}

/**Iterative implementation. */
static IntList incrList(IntList P, int n) {
   if (P == null) {
      return null;
   }
   IntList result, last;
   result = new IntList(P.head+n, null);
   last = result;
   while (P.tail != null) {
      P = P.tail;
      last.tail = new IntList(P.head+n, null);
      last = last.tail;
   }
   return result;
}
```

Notice that the recursive implementation simply implements the incrementation for a single node and makes its recursive call on the **tail** of the node, effectively incrementing the rest of the IntList.

The iterative implementation, on the other hand, is far longer. It requires two pointers- one to point to jump from one node to the next in the IntList and increment their respective values (`last`) and another to point to the entire IntList (`result`) so that the edited list can be returned. Although arguably more straightforward, iterative implementations leaves room for a lot of errors, **especially** `NullPointerExceptions`.

In general, there is a trade-off between complexity and code length in terms of function implementation. While on tests you will most likely be forced to do one or the other, use the implementation type that you can understand best.

3.2 A Destructive Approach

Destructive functions are actually nearly always implemented as **methods**![2] This is because methods have direct access to the calling object's attributes, and can thus edit them directly.

Let's add this destructive incrementation as a method to the IntList class.

```
public class IntList {
   // Constructor
   /** List cell containing (HEAD, TAIL). */
   public IntList(int head, IntList tail) {
      this.head = head;
      this.tail = tail;
   }

   /** Destructive incrementation of all nodes by n. */
   public void incrList(int n) {
      IntList scout = this;
      while (scout != null) {
         scout.head += n;
         scout = scout.tail;
      }
   }
   // Names of simple containers (fields/attributes)
   public int head;
   public IntList tail;
}
```

So simply creating any `IntList` and calling the `incrList` method would destructively edit its node values.

```
IntList L = new IntList(1, IntList(2, IntList(3, null))); //L = [1,2,3]
L.incrList(6); //L = [7,8,9]
```

Let's end our note here. Next week, we'll dive deeper into pointer manipulation, and recursive/iterative/destructive/non-destructive functions. This is an integral part of building knowledge on how to not only understand but **manipulate** data structures.

[2]Static destructive methods can be implemented, but remember that you MUST preserve the original pointer to the data structure in question, or it will be lost forever. So you'd just need to make a separate traversal pointer to modify the structure. Methods basically do the same thing with less steps.

For practice, trace through a visual representation of a sample call to `incrList`, or view the official lecture slides for an example trace.

Here's another challenging practice problem: implement the non-destructive `incrSqrList` below, where we increment the ith-indexed node of an IntList, where i is a **square number** (node index starts from 0!!). Here's a hint: recursion probably won't do it for you here.

```
/** Increment non-destructively nodes with square number indices.
Thus if L = [1,2,3,4,5,6,7,8,9], incrSqrList(L, 5) would return a
    pointer to [6,7,3,4,10,6,7,8,14]
*/
static IntList incrSqrList(IntList P, int n) {
   //Your code here
}
```

4 Summary

In this note, we took our first look into data structures. We began with the basics: values and containers. Simple containers contained values, while structured containers contained multiple other containers. The two types of containers work in tandem as a data structure: a structured container held the data structure itself, while the simple container contained a pointer to the data structure/structured container.

Our first official data structure comes as an integer-valued linked list, or an IntList. Through implementing the creation of an arbitrary IntList we gained a better grasp on pointer function and manipulation.

Finally, one of the most common skills not only in this class but in computer science is the ability to manipulate data structures in various ways with various constraints. We went over a sample function `incrList`, which we implemented both destructively (as a void method) and non-destructively (as a static function that returned an IntList).