CS61B Note 4: Pointer Manipulation

Kevin Moy

Abstract

In this note, we gain a firmer understanding of the many different ways to use pointers to manipulate data structures. It is important to understand that the "mobility" and flexibility of pointers allow us to do a myriad of wonderful things.

1 Quick Note: The NULL POINTER

I completely forgot to address in the last note one of the most common pointers of all: the **null** pointer. A null pointer is a pointer that points to nothing. It is what the last node in a linked list (IntList) has as its `tail` field. It is the "stopping point" of nearly all data- structures, algorithms, and processing events. It is important to note that unlike Python, **null cannot be interchanged with false**, and therefore null cannot be compared with any types.

2 Quick Note 2: The Final Keyword

A variable given the `final` keyword means that its value cannot be changed, ever. Whenever we need to reference something that will never ever change, it is often a good idea to make it final so you don't edit it by accident and subsequently lose the data.

For example: `final int x = 3` would create a variable x that locks to 3. If x is reassigned, this would raise an exception and the code would not compile.

However, we run into a problem with pointers.

```
final IntList L = new IntList(1, IntList(2, null));
L.head = 2;
```

This code **would** compile. The `final` keyword only made the **pointer** `L` unchangeable- however, the contents of the object that it points to, which is a structured container, is changeable. So while we wouldn't be able to reassign `L` to anything else, we could still modify the object it points to. Thus, it is probably wise to only use final with numbers and booleans, not objects.

3 Destructive Functions Continued

In our last note, we briefly covered how to implement a destructive method, where a data structure passed in as an argument was edited directly in order to save time or memory.

3.1 Destructive Methods: A Review

We looked at `incrList`, which destructively added an integer N to all of an IntList's nodes:

```
static IntList incrList(IntList L, int n) {
   for (IntList p = L; p != null; p = p.tail) {
      p.head += n;
   }
   return L;
}
```

NOTE: The code above does the *exact* same thing as the code in the previous note, except smartly utilizes the for loop to save coding space. Also, note that this is a static function, so we'll have to actually return the edited structure for it to have use.

4 Non-Destructive Deletion [Recursive]

Let's implement the `removeAll` function from lecture, which takes in an IntList `L` and an integer `x` and removes **all** instances of `x` from `L`. Let's write up our code skeleton first. The code below resembles skeleton code you might see on a midterm one day, so try to write some good code yourself first!

```
/** Remove all instances of x from L. For example, if L = [1,2,2,3,3,3],
    removeAll(L,2) returns [1,3,3,3]. */
static IntList removeAll(IntList L, int x) {
   if (L == null) {
      return ____
   } else if (L.head == x) {
      return _____
   } else {
      return ____
   }
}
```

Gave it a shot? Good.

The first thing we should consider is our base case, which checks if the IntList is `null`. If so, we effectively are calling to remove elements from an empty list, which will always just result in an empty list!. So we know to return null for our base case.

```
/** Remove all instances of x from L. For example, if L = [1,2,2,3,3,3],
    removeAll(L,2) returns [1,3,3,3]. */
static IntList removeAll(IntList L, int x) {
   if (L == null) {
      return null;
   } else if (L.head == x) {
      return _____
   } else {
      return ____
   }
}
```

Now the middle case seems kind of weird and a byproduct of recursion. It's saying to check if the first node is something to delete. What do we do here? Doesn't seem too obvious at first.

We'll come back to that. Let's move on to our recursive `else` case. **Remember that our function is non-destructive and thus we need to return an IntList!** Given that we are trying to one-line our solution and have no room for pointers, we probably need to return `new IntList` directly. However, what do we put for our head and tail arguments?

Well, we already know that the our head passed the delete test (second base case), so we can keep head as `L.head`.

Now let's think about what `removeAll` does. It returns an IntList from L, except with all x's removed. Hey, this can be our tail!

Think about how this makes sense conceptually: If we trust `removeAll` works, then calling `removeAll` on L is the same as keeping L's head (assuming it isn't == x) and calling `removeAll` on the rest of the IntList. Another example of the importance of thinking how to simplify the problem with each recursive call.

Wrapping that entire rant into a single line of code:

```
/** Remove all instances of x from L. For example, if L = [1,2,2,3,3,3],
    removeAll(L,2) returns [1,3,3,3]. */
static IntList removeAll(IntList L, int x) {
   if (L == null) {
      return null;
   } else if (L.head == x) {
      return _____
   } else {
      return new IntList(L.head, removeAll(L.tail, x));
   }
}
```

Finally, the second base case might make a little more sense now, or it might not. What happens if our "head" node has the x value we want to remove?

Remember `removeAll` returns an IntList!! Calling removeAll on L's tail effectively "moves on" from L's head node! This is a nifty way to "skip" nodes for processing.

Finally, our completed method:

```java
/** Nondestructively, and recursively, remove all instances of x from L.
    For example, if L = [1,2,2,3,3,3], removeAll(L,2) returns
    [1,3,3,3]. */
static IntList removeAll(IntList L, int x) {
   if (L == null) {
      return null;
   } else if (L.head == x) {
      return removeAll(L.tail, x);
   } else {
      return new IntList(L.head, removeAll(L.tail, x));
   }
}
```

5 Non-Destructive Deletion [Iterative]

How would we implement this iteratively?

Well, all we really have to do is traverse the original IntList and build up a new one from nodes that **are not** equal to x.

```java
/** Nondestructively, and iteratively, remove all instances of x from L.
    */
static IntList removeAll(IntList L, int x) {
   IntList result, last;
   result = last = null;
   for (; L != null; L = L.tail) {
      if (x == L.head) {
         continue;
      } else if (last == null) {
         result = last = new IntList(L.head, null);
      } else {
         last = last.tail = new IntList(L.head, null);
      }
   }
   return result;
}
```

What you see above is our for loop looking through each node of L sequentially. `last` serves as our IntList builder: if the node being looked at is equal to x, it is skipped with the `continue` keyword, which just moves on to the next iteration of the for loop without doing anything else in the loop body. If the node being looked at is **not** equal to x, then it becomes part of the list being built up in the `else` case. We see our second base case is, again, our stopping point where we return our built-up IntList as our final product.

For brevity, I will not walk through an example as it is far more straightforward than the recursive implementation. Look at slides 15-22 on the official lecture slides for a walkthrough.

6 Destructive Deletion [Recursive]

Now what if we could mess with the original list passed in? How would the destructive and non-destructive recursive methods differ?

Remember that the key point of destructive methods is that we **don't have to use the `new` keyword** to copy stuff. We just edit what's passed in and return that.

So we still want to return the pointer to the IntList we passed in!

Knowing this, we can come up with two key differences.

1. We don't want to return `null` for our base case- we just do nothing and return instead, since we aren't building any lists up.
2. Instead of returning a *new* IntList, we can just call `removeAll` on our IntList's tail directly.

In code:

```
/** Destructively, and recursively, remove all instances of x from L. */
static IntList dremoveAll(IntList L, int x) {
   if (L == null) {
      return
   } else if (L.head == x) {
      return dremoveAll(L.tail, x);
   } else {
      L.tail = dremoveAll(L.tail, x);
      return L;
   }
}
```

7 Destructive Deletion [Iterative]

This is probably the trickiest because if we "delete" a node outright, we risk losing our data. So we have to know how to **safely** delete nodes to implement this.

Turns out, all we need is another pointer to point to the **next node** after our current one, in case the current one needs to be skipped! This way, we won't lose access to all the nodes that follow the skipped node. We'll rightfully call this pointer `next`.

```java
/** Destructively and iteratively remove all instances of x from L. */
static IntList removeAll(IntList L, int x) {
   IntList result, last;
   result = last = null;
   while (L != null) {
      IntList next = L.tail;
      if (x != L.head) {
         if (last == null) {
            result = last = L;
         } else {
            last = last.tail = L;
         }
         L.tail = null;
      }
      L = next;
   }
   return result;
}
```

Analyze and trace the above code thoroughly. This may be a good function to highlight on your endless cheat sheets.

8 Summary

In this note, we thoroughly analyzed 4 different versions of the `removeAll` function, a function that removed all nodes with a specific value from an IntList. Such 4 implementations were all combinations of recursive/iterative and destructive/nondestructive versions of the function. Through this, we realized that pointers not only have importance in modification, but also in keeping track of select parts of a data structure for access when needed.

When we cover arrays next week, which share similarities but also have significant differences with linked lists, we will realize that being able to work with pointers will help us tremendously once again.