Recursion in Java

Marius Nita
CS Tutors – Portland State University

May 3, 2005

1 Introduction

Lower-division programming courses, such as CS161-CS163 and CS202, use the C++ programming language to introduce students to I/O, data structures, and programming in general. An unfortunate result of the choice for C++ is that students become unduly dependent on C++ references (a feature unavailable in most other languages) and pointers to encode basic computation patterns such as recursion.

A basic example of using references in recursion is performing modifications to trees or other recursive data structures. Consider the following tree data structure:

```
struct Tree {
    int data;
    struct Tree *left;
    struct Tree *right;
};
```

A C++ function which removes a node from a tree of this kind could be encoded as follows:

```
void remove(struct Tree *&t, int d) {
    if (t == NULL)
        return;
    else if (t->data == d) {
        if (t->left == NULL)
            t = t->right;
        else {
            // attach t->right to the rightmost leaf in t->left
            patch_tree(t);
            t = t->left;
        }
    } else if (t->data > d)
        remove(t->left,d);
    else
        remove(t->right,d);
}
```

This pattern might be completely unfamiliar to a functional, or even Java programmer, since it relies on using function arguments as a vehicle for modifying variables in the caller's space, in a completely obscure manner.

One could encode the same problem in C, by passing an address to the tree pointer:

```
void remove(struct Tree **t, int d) {
    Tree *tree = *t;
    if (tree == NULL)
        return;
    else if (tree->data == d) {
        if (tree->left == NULL)
            *t = tree->right;
        else {
            // attach tree->right to the rightmost leaf in tree->left
            patch_tree(tree);
            *t = tree->left;
        }
    } else if (tree->data > d)
        remove(&tree->left,d);
    else
        remove(&tree->right,d);
}
```

The pure C code is a bit more readable, since the tree nodes are passed explicitly by pointer; the call is giving a syntactic indicator for the possibility that the function might modify what the argument points to.

1.1 Coding C++ in Java is problematic

When lower-division students move to Java programming and are faced with solving this problem, they invariably try to duplicate the reference or double-pointer pattern that they became used to in their C++ classes. The "problem" with doing this in Java is that Java lacks references and pointer arithmetic. Consequently, one is unable to (a) change the caller's space by assigning into a function argument, or (b) pass the *address* of an object pointer into a function. For example, writing

```
void remove(Tree t, int d) {
    ... t = null; ...
}
```

will not affect the caller's argument to the function `remove`. In effect, `t` becomes a local variable, initialized to the value of the passed-in argument.

A popular (but unfortunate) hack is to use an explicit mechanism to model the recursion, or "fake" the double indirection in the C encoding by using "double-boxing." This involves using a class which simply wraps the `Tree` class,

```
class TreePtr {
    Tree t;
    TreePtr(Tree t) { this.t = t; }
}
```

and coding `remove` to deal with this extra-level of boxing:

```
void remove(TreePtr tp, int d) {
    Tree tree = tp.t;
    if (tree == null)
        return;
    else if (tree.data == d) {
        if (tree.left == null)
            tp.t = tree.right;  // <- set the caller
        else {
            // attach tree.right to the rightmost leaf in tree.left
            patch_tree(tree);
            tp.t = tree.left;   // <- set the caller
        }
    } else if (tree.data > d) {
        TreePtr nl = new TreePtr(tree.left);
        remove(nl,d);
        tree.left = nl.t;
    } else {
        TreePtr nr = new TreePtr(tree.right);
        remove(nr,d);
        tree.right = nr.t;
    }
}
```

Java's `TreePtr` is intended to mimic `Tree**` in C/C++; `tp.t = e` is effectively equivalent to `*t = e`. To set up the call for this function, we do

```
TreePtr tp = new TreePtr(t); // wrap the tree
remove(tp,3);                // run remove
t = tp.t;                    // "unwrap" the tree
```

where `t` is a `Tree`. Not only is this unduly complicated, but it is also wasteful memory- and time-wise. There is a much more intuitive approach to all this.

2 Side effect-free recursion

The sole purpose of updating the argument, whether via references or double indirection, is to communicate "something" to the caller—typically a new value for the tree's root. Not surprisingly, Java, as well as any other imperative language, offers a perfectly good built-in mechanism for doing this: function return! Very simply put, instead of updating the argument in-place with a new tree, we simply return the new tree. The `remove` function's signature would then be

```
Tree remove(Tree t,int d);
```

and can be read as follows: the function `remove` takes a `Tree` and an `int`, and returns a new `Tree`, which is in every way like the one that was passed in, except that it is missing the node with label `d`. Though this explanation is philosophically sound, it is baffling to students who are faced with lack of double indirection and references. Using this approach, the Java implementation for `remove` would look like

```
Tree remove(Tree t, int n) {
    if (t == null)
        return null;          // <- set caller
    else if (t.data == n) {
        if (t.left == null)
            return t.right;   // <- set caller
        else {
            patch_tree(t);
            return t.left;    // <- set caller
        }
    }
    else if (t.data > n)
        t.left = remove(t.left,n);
    else
        t.right = remove(t.right,n);
    return t;                 // <- set caller
}
```

Very simply, this approach uses assignment and function return to build a new tree which is missing the given node. The caller would invoke this function with a command such as `t = remove(t,5);` . Note that simply saying `remove(t,5);` is problematic. It will work in every case, except the case where the root itself is removed. When the root is removed, a *new* node will be returned; but since the result is not recorded anywhere, `t` will keep pointing to the old root, which is no longer a legitimate node in the tree.

To help you gain an intuition for how this function proceeds, Figure 1 outlines the computation steps taken to remove the node with label `3` from the given binary search tree (BST).

If our pointer to the tree is held in variable `t`, part (a) illustrates the state at the point of the call `remove(t,3)`. A square box is used to represent the tree pointer `t`. Part (b) shows the state of the call after it is determined that the data will be found in the left branch of the BST. This corresponds to the line `t.left = remove(t.left,n)` in the `remove` function, where `t` still points to the root, and `n` is `3`. The arrow pointing from the root, representing `t.left`, is now being reassigned to whatever `remove(t.left,n)` will return. Part (c) shows the state of the call after the inner `remove(t.left.n)` has finished, returning to the caller the pointer with label `4`. This pointer is assigned into `t.left`, indicated by the dotted line. Part (d) shows the resulting tree, where now nothing points to the `3` node anymore. Since Java is a garbage collected language, the `3` node will be automatically reclaimed by the collector, resulting in the state illustrated by (e).

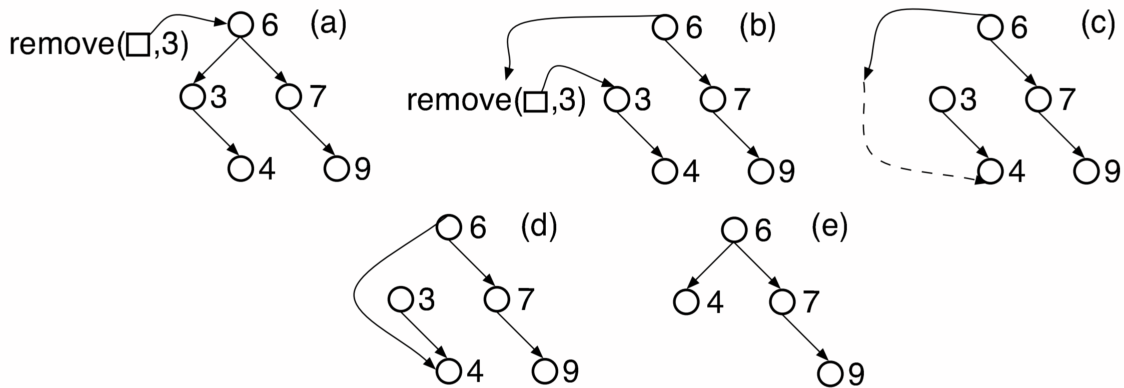

Figure 1: State progression for `remove(t,3);`

2.1 Error return

One questions that students immediately ask when presented with this recursion pattern is: since the return value is used to return a tree, what happens if you want to return an error value? There are several ways to handle this issue, the cleanest being exceptions. Instead of threading error values through return, which is a clumsy and outdated way of doing it anyway, throw a custom exception. The caller will then trap it in a typical `try/catch` block:

```
try {
    t = remove(t,3);
} catch (NodeNotFound e) {
    ...
}
```

3 Conclusion

We have presented a clean way to model recursion in Java, not nearly as clumsy and problematic as recursion patterns which try to replicate functionality available in C and C++. It promotes a high-level mindset which allows more effective reasoning about what the functions do, and which provides a good stepping stone for learning to program in modern functional programming languages, such as OCaml and Standard ML.

Using references and double-indirection to gather recursion results is bad style, simply because it depends on archaic features not necessarily available in all languages.