

Implementing Quicksort with a Stack

CSE 332: Data Abstractions
May 31, 2011

The subsection of Section 3.6 of the textbook called “Method Calls” discusses how a compiler implements recursion by use of a stack. Here is a concrete example. Consider the recursive version of quicksort, an algorithm to sort an array of keys:

```
// sort list between positions 0 and size-1, inclusive
procedure quickSort(int[] list, int size) {
    quickSort(list, 0, size-1);
}

// sort the part of list between positions low and high, inclusive, using recursion
procedure quickSort(int[] list, int low, int high) {
    if (low <= high) {
        int pivotLocation = partition(list, low, high);
        quickSort(list, low, pivotLocation-1);
        quickSort(list, pivotLocation+1, high);
    }
}
```

The details of the function `partition` aren’t important for our purposes. All you need to know is that it rearranges the keys and returns an index i with the property that all the keys at positions `low`, \dots , $i - 1$ are less than or equal to the key at position i , and all the keys at positions $i + 1$, \dots , `high` are greater than or equal to the key at position i . (If you are curious about the details, see Section 7.7.2 of the textbook.) See Figure 7.11 of the textbook for an illustrative run of `quickSort`.

Here is a simple version of this procedure using a stack `s` to implement the recursion. The entries on the stack will be pairs of integers. (This is conceptual only. The simplest implementation would use a stack of integers, and you would push and pop two at a time.)

```
// sort list between positions 0 and size-1, inclusive, using a stack
procedure quickSort(int[] list, int size) {
    int low = 0;
    int high = size-1;
    int pivotLocation;
    Stack<IntPair> s = new Stack<IntPair>();
    s.push(<0,-1>); // a marker to identify the bottom of the stack
    while (!s.isEmpty()) {
        while (low <= high) {
            pivotLocation = partition(list, low, high);
            s.push(<pivotLocation+1,high>); // record info for second recursive call
            high = pivotLocation-1; // execute first recursive call
        }
        <low,high> = s.pop(); // fetch next recursive call to execute
    }
}
```