Data Structures and Algorithm Analysis

Edition 3.2 (C++ Version)

Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

March 28, 2013
Update 3.2.0.10
For a list of changes, see
http://people.cs.vt.edu/~shaffer/Book/errata.html

Copyright © 2009-2012 by Clifford A. Shaffer.
This document is made freely available in PDF form for educational and other non-commercial use. You may make copies of this file and redistribute in electronic form without charge. You may extract portions of this document provided that the front page, including the title, author, and this notice are included. Any commercial use of this document requires the written consent of the author. The author can be reached at shaffer@cs.vt.edu.
If you wish to have a printed version of this document, print copies are published by Dover Publications
(see http://store.doverpublications.com/048648582x.html).
Further information about this text is available at
http://people.cs.vt.edu/~shaffer/Book/.

be the index for the top element in the stack, rather than the first free position. If this had been done, the empty list would initialize **top** as -1.) Methods **push** and **pop** simply place an element into, or remove an element from, the array position indicated by **top**. Because **top** is assumed to be at the first free position, **push** first inserts its value into the top position and then increments **top**, while **pop** first decrements **top** and then removes the top element.

4.2.2 Linked Stacks

The linked stack implementation is quite simple. The freelist of Section 4.1.2 is an example of a linked stack. Elements are inserted and removed only from the head of the list. A header node is not used because no special-case code is required for lists of zero or one elements. Figure 4.19 shows the complete linked stack implementation. The only data member is **top**, a pointer to the first (top) link node of the stack. Method **push** first modifies the **next** field of the newly created link node to point to the top of the stack and then sets **top** to point to the new link node. Method **pop** is also quite simple. Variable **temp** stores the top nodes' value, while **ltemp** links to the top node as it is removed from the stack. The stack is updated by setting **top** to point to the next link in the stack. The old top node is then returned to free store (or the freelist), and the element value is returned.

4.2.3 Comparison of Array-Based and Linked Stacks

All operations for the array-based and linked stack implementations take constant time, so from a time efficiency perspective, neither has a significant advantage. Another basis for comparison is the total space required. The analysis is similar to that done for list implementations. The array-based stack must declare a fixed-size array initially, and some of that space is wasted whenever the stack is not full. The linked stack can shrink and grow but requires the overhead of a link field for every element.

When multiple stacks are to be implemented, it is possible to take advantage of the one-way growth of the array-based stack. This can be done by using a single array to store two stacks. One stack grows inward from each end as illustrated by Figure 4.20, hopefully leading to less wasted space. However, this only works well when the space requirements of the two stacks are inversely correlated. In other words, ideally when one stack grows, the other will shrink. This is particularly effective when elements are taken from one stack and given to the other. If instead both stacks grow at the same time, then the free space in the middle of the array will be exhausted quickly.

```
// Linked stack implementation
template <typename E> class LStack: public Stack<E> {
private:
  Link<E>* top;            // Pointer to first element
  int size;                   // Number of elements

public:
  LStack(int sz =defaultSize) // Constructor
    { top = NULL; size = 0; }

  ~LStack() { clear(); }          // Destructor

  void clear() {                  // Reinitialize
    while (top != NULL) {         // Delete link nodes
      Link<E>* temp = top;
      top = top->next;
      delete temp;
    }
    size = 0;
  }

  void push(const E& it) { // Put "it" on stack
    top = new Link<E>(it, top);
    size++;
  }

  E pop() {                // Remove "it" from stack
    Assert(top != NULL, "Stack is empty");
    E it = top->element;
    Link<E>* ltemp = top->next;
    delete top;
    top = ltemp;
    size--;
    return it;
  }

  const E& topValue() const { // Return top value
    Assert(top != 0, "Stack is empty");
    return top->element;
  }

  int length() const { return size; } // Return length
};
```

Figure 4.19 Linked stack class implementation.

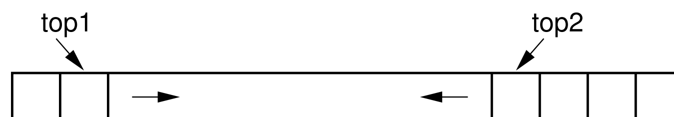

Figure 4.20 Two stacks implemented within in a single array, both growing toward the middle.

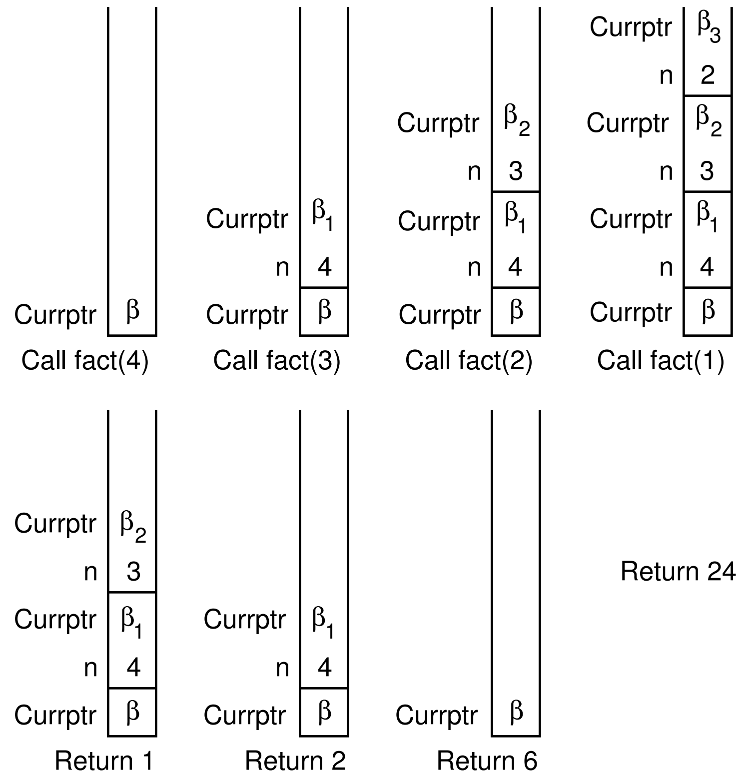

Figure 4.21 Implementing recursion with a stack. β values indicate the address of the program instruction to return to after completing the current function call. On each recursive function call to **`fact`** (as implemented in Section 2.5), both the return address and the current value of **`n`** must be saved. Each return from **`fact`** pops the top activation record off the stack.

4.2.4 Implementing Recursion

Perhaps the most common computer application that uses stacks is not even visible to its users. This is the implementation of subroutine calls in most programming language runtime environments. A subroutine call is normally implemented by placing necessary information about the subroutine (including the return address, parameters, and local variables) onto a stack. This information is called an **activation record**. Further subroutine calls add to the stack. Each return from a subroutine pops the top activation record off the stack. Figure 4.21 illustrates the implementation of the recursive factorial function of Section 2.5 from the runtime environment's point of view.

Consider what happens when we call **`fact`** with the value 4. We use β to indicate the address of the program instruction where the call to **`fact`** is made. Thus, the stack must first store the address β, and the value 4 is passed to **`fact`**.