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

Example 4.1 There is a space-saving technique that can be employed to eliminate the additional space requirement, though it will complicate the implementation and be somewhat slower. Thus, this is an example of a space/time tradeoff. It is based on observing that, if we store the sum of two values, then we can get either value back by subtracting the other. That is, if we store $a + b$ in variable c, then $b = c - a$ and $a = c - b$. Of course, to recover one of the values out of the stored summation, the other value must be supplied. A pointer to the first node in the list, along with the value of one of its two link fields, will allow access to all of the remaining nodes of the list in order. This is because the pointer to the node must be the same as the value of the following node's **`prev`** pointer, as well as the previous node's **`next`** pointer. It is possible to move down the list breaking apart the summed link fields as though you were opening a zipper. Details for implementing this variation are left as an exercise.

The principle behind this technique is worth remembering, as it has many applications. The following code fragment will swap the contents of two variables without using a temporary variable (at the cost of three arithmetic operations).

```
a = a + b;
b = a - b; // Now b contains original value of a
a = a - b; // Now a contains original value of b
```

A similar effect can be had by using the exclusive-or operator. This fact is widely used in computer graphics. A region of the computer screen can be highlighted by XORing the outline of a box around it. XORing the box outline a second time restores the original contents of the screen.

4.2 Stacks

The **stack** is a list-like structure in which elements may be inserted or removed from only one end. While this restriction makes stacks less flexible than lists, it also makes stacks both efficient (for those operations they can do) and easy to implement. Many applications require only the limited form of insert and remove operations that stacks provide. In such cases, it is more efficient to use the simpler stack data structure rather than the generic list. For example, the freelist of Section 4.1.2 is really a stack.

Despite their restrictions, stacks have many uses. Thus, a special vocabulary for stacks has developed. Accountants used stacks long before the invention of the computer. They called the stack a "LIFO" list, which stands for "Last-In, First-

```
// Stack abtract class
template <typename E> class Stack {
private:
  void operator =(const Stack&) {}     // Protect assignment
  Stack(const Stack&) {}         // Protect copy constructor

public:
  Stack() {}                        // Default constructor
  virtual ˜Stack() {}               // Base destructor

  // Reinitialize the stack.  The user is responsible for
  // reclaiming the storage used by the stack elements.
  virtual void clear() = 0;

  // Push an element onto the top of the stack.
  // it: The element being pushed onto the stack.
  virtual void push(const E& it) = 0;

  // Remove the element at the top of the stack.
  // Return: The element at the top of the stack.
  virtual E pop() = 0;

  // Return: A copy of the top element.
  virtual const E& topValue() const = 0;

  // Return: The number of elements in the stack.
  virtual int length() const = 0;
};
```

Figure 4.17 The stack ADT.

Out." Note that one implication of the LIFO policy is that stacks remove elements in reverse order of their arrival.

The accessible element of the stack is called the **top** element. Elements are not said to be inserted, they are **pushed** onto the stack. When removed, an element is said to be **popped** from the stack. Figure 4.17 shows a sample stack ADT.

As with lists, there are many variations on stack implementation. The two approaches presented here are **array-based** and **linked stacks**, which are analogous to array-based and linked lists, respectively.

4.2.1 Array-Based Stacks

Figure 4.18 shows a complete implementation for the array-based stack class. As with the array-based list implementation, `listArray` must be declared of fixed size when the stack is created. In the stack constructor, `size` serves to indicate this size. Method `top` acts somewhat like a current position value (because the "current" position is always at the top of the stack), as well as indicating the number of elements currently in the stack.

```
// Array-based stack implementation
template <typename E> class AStack: public Stack<E> {
private:
  int maxSize;              // Maximum size of stack
  int top;                  // Index for top element
  E *listArray;          // Array holding stack elements

public:
  AStack(int size =defaultSize)   // Constructor
    { maxSize = size; top = 0; listArray = new E[size]; }

  ~AStack() { delete [] listArray; }  // Destructor

  void clear() { top = 0; }           // Reinitialize

  void push(const E& it) {         // Put "it" on stack
    Assert(top != maxSize, "Stack is full");
    listArray[top++] = it;
  }

  E pop() {                // Pop top element
    Assert(top != 0, "Stack is empty");
    return listArray[--top];
  }

  const E& topValue() const {     // Return top element
    Assert(top != 0, "Stack is empty");
    return listArray[top-1];
  }

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

Figure 4.18 Array-based stack class implementation.

The array-based stack implementation is essentially a simplified version of the array-based list. The only important design decision to be made is which end of the array should represent the top of the stack. One choice is to make the top be at position 0 in the array. In terms of list functions, all **insert** and **remove** operations would then be on the element in position 0. This implementation is inefficient, because now every **push** or **pop** operation will require that all elements currently in the stack be shifted one position in the array, for a cost of $\Theta(n)$ if there are n elements. The other choice is have the top element be at position $n - 1$ when there are n elements in the stack. In other words, as elements are pushed onto the stack, they are appended to the tail of the list. Method **pop** removes the tail element. In this case, the cost for each **push** or **pop** operation is only $\Theta(1)$.

For the implementation of Figure 4.18, **top** is defined to be the array index of the first free position in the stack. Thus, an empty stack has **top** set to 0, the first available free position in the array. (Alternatively, **top** could have been defined to 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.

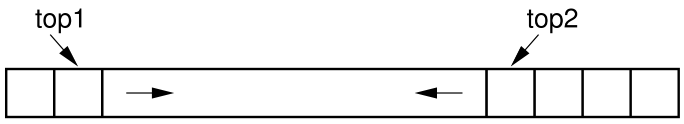

Figure 4.20 Two stacks implemented within in a single array, both growing toward the middle.