Week 8, Friday Class — October 19, 2001

Stacks: ADT, Implementation and Applications

**HW, Program and Test Dates**

Here is a revised schedule of dates for the course. From now on, the homeworks and programming assignments will be separate, with the homeworks being *relatively short* and only requiring hand-written solutions.

<table>
<thead>
<tr>
<th>What</th>
<th>When Given</th>
<th>When Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW 6</td>
<td>Tue, Oct 23</td>
<td>Tue, Oct 30</td>
</tr>
<tr>
<td>Test 2</td>
<td></td>
<td>Tue, Nov 6</td>
</tr>
<tr>
<td>Prog 2</td>
<td>Tue, Oct 20</td>
<td>Fri, Nov 16</td>
</tr>
<tr>
<td>HW 7</td>
<td>Tue, Nov 13</td>
<td>Tue, Nov 20</td>
</tr>
<tr>
<td>Prog 3</td>
<td>Fri, Nov 16</td>
<td>Tue, Dec 4</td>
</tr>
<tr>
<td>HW 8</td>
<td>Tue, Nov 27</td>
<td>Fri, Dec 7</td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td>Wed, Dec 12</td>
</tr>
</tbody>
</table>

**Review from Tuesday’s Class**

- Recursive definition of a linked list. Many other important data structures are defined recursively as well.

- Recursive linked list algorithms follow from the recursive definition of a linked list.

- Algorithm design strategy based on dividing up a problem. In the examples, we divided an array interval in half and recursively worked (or didn’t work) on each half.

- Binary search, based on this idea, is a fast search technique for sorted arrays.

- We examined merging two sorted arrays as a precursor to merge sort which is also based on this idea.

**Reading:** Carrano and Prichard 273-296, 298-303
Today’s Class

• Finish discussion of merging and merge sort.
• Stacks and the stack ADT.
• Stack implementation using arrays and linked-lists.
• Stack-based evaluation of postfix expressions.
• Queue ADT (a start only).
• Comparing sets, lists, stacks and queues.

Stacks: Fundamental Operations

• A stack is a restricted data structures that allows only the following operations:
  – “Pushing” an item on the “top” of the stack.
  – “Popping” an item off the “top” of the stack.
  – Accessing the “top” element of the stack.
  – Checking to see if the stack is empty.

• These should correlate with your intuitive notion of a stack in the physical world.

• Note that there is no means of direct access to items that aren’t on top of the stack.

Stacks: Uses

Surprisingly, this simple data structure has a large number of uses:

• Activation records for function calls are stored on a stack.
• Eliminating recursion to make an algorithm faster can be achieved by using a stack.
• Stacks can be used in expression evaluation
• Stacks are used by compilers during parsing.
• Search algorithms often exploit stack structures.
Stack Example: Detecting Unbalanced Parentheses

• Automatically checking the syntax of expressions is a major job of a compiler (the parser in the compiler).
• As a simple example we can use a stack to check if the parentheses in an expression are balanced.
• We push each '(' onto the stack.
• Each ')' causes a stack pop.
• Errors are detected when the expression is ended with at least one '(' still on the stack or when a ')' is seen but the stack is empty.

Aside: I will describe an even simpler method for solving this problem that I use in my head.

The Stack ADT

• Here’s the ADT for the stack, written as a C++ header file. (The book uses UML notation.)

```
// Assumes a generic data type, T, to be stored on the stack.

class Stack {
    public:
        Stack(); // create an empty stack
        Stack( const& Stack); // copy an existing stack
        ~Stack(); // destroy a stack
        void push( const& T item ); // add item to top
        void pop(); // remove top
        void pop( T& item ); // remove top and place in item
        void getTop( T& item ) const; // return top in item
        bool isEmpty( ) const; // is the stack empty?
    };
```

• In implementation, the type T must be filled in.
  – In a week or so we will see how to make this “generic” using templates.
Stack Implementation: Linked Version

- One implementation uses a singly-linked list, with the “head” pointer replaced by pointer to the “top” node.
  - Not surprisingly, this pointer is called `topPtr`.
- Each node stores an “item” and a pointer to the next node on the stack.
- In this particular case, the item type T is a `char`.
- The code is attached.
- The example application is checking balanced parentheses.

Stack Implementation: Array Version

- A second implementation uses arrays.
- “top” is the largest array index with a stored stack value.
  - Hence an empty stack has `top == -1`.
- The current implementation uses a fixed-size array. In general, this is a VERY bad idea. We will fix it in lab.
- The stack items in this example are `doubles`. We will see an application soon.

Postfix Expressions

- A postfix expression is one where the operators follow the operands instead of being inbetween the operands.
- Here are several examples with their *infix* equivalents:

<table>
<thead>
<tr>
<th>Postfix</th>
<th>Infix</th>
</tr>
</thead>
<tbody>
<tr>
<td>567 * +</td>
<td>5 + 6 * 7</td>
</tr>
<tr>
<td>567 + *</td>
<td>5 * (6 + 7)</td>
</tr>
<tr>
<td>56 + 7 * 89 -</td>
<td>((5 + 6) * 7) * (8 - 9)</td>
</tr>
</tbody>
</table>

In the last case, I have added extra parentheses to the *infix* expression for clarity.
An important aspect of postfix is that parenthesis are not needed.

HP calculators are based on this idea. (At least they used to be.)

Postfix expressions can be evaluated using a stack. Infix can be converted to postfix with a stack.

We will only consider evaluation here. See the Carrano and Prichard text for infix-to-postfix conversion.

Using a Stack to Evaluate Postfix Expressions

- The idea is very simple:
  - Whenever an operand (a number) is seen, push it onto the stack.
  - Whenever an operator ('+', '-', '/', '*') is seen, pop the top two numbers off the stack, apply the operation and push the result. Note that the first number popped is the right-hand side of the operator.
  - In addition, we will add the '=' operator to output the top of the stack (without popping it).

- From the coding standpoint, the toughest part is in fact taking the input and breaking it up into “tokens”, where each token is an operand or an operator. A particular difficulty occurs when there is no blank space between tokens.

- The code is attached and will be discussed in class.

Queue ADT (briefly)

- A queue is a data structure that only allows items to be inserted in one end (the “back”) and removed from the other (the “front”).

- The fundamental operations are called “enqueue” (insert in the back) and “dequeue” (remove from the front).

- Queues are FIFO (“first-in-first-out”) structures, whereas stacks are LIFO (“last-in-first-out”) structures.
• Queues are used frequently in
  – operating systems (printing and other services),
  – networks, and
  – simulations.

• We will revisit queues on Tuesday.

Sets, Lists, Stacks and Queues

• We have seen 4 abstract data types (ADTs). Here is a brief summary of their properties:

  Sets: Each item is unique. Order is irrelevant. Operations are standard mathematical set operations.

  Lists: Items are not unique. Order is relevant. Operations include insert, remove, find. Positions can be specified.

  Stacks: Insert and remove only at one end. No direct access to other items.

  Queues: Insert at one end and remove at the other. No direct access to other items.

• Of these, lists are the most general.

• Stacks and queues are the simplest and can be implemented most efficiently.

• Choices among ADTS are based on mostly on simplicity. Choices of data structures implementing ADTs are based mostly on efficiency.

• In November we will develop tools for analyzing the efficiency of data structures and their associated algorithms.