Announcements

- Final exam review and practice problems are being distributed today.
- Next Monday, April 29th: no class.
- Next Friday, May 3rd, 2pm: I will offer a question-and-answer review session in Amos Eaton 214.
- Final exam: Monday May 6th, 3-6pm in Amory Mids.

Review from Monday

- Solutions to the extra credit homework
- Limitations of singly-linked lists
- Doubly-linked lists:
  - Structure
  - Insert
  - Remove
- Our own version of the list<T> class, cs2list<T>
- cs2list<T>::iterator
- cs2list<T>::push_front and cs2list<T>::erase

Today’s Class

- Stacks and stack implementations
- Stack-based evaluation of postfix expressions.
- Queues and queue implementations
- Course summary

Note: since we will not have a lot of homework on stacks and queues, the review problems on stacks and queues are particularly important.
Stacks: Fundamental Operations

- A stack is a restricted data structure 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 Class

- Here's the stack “abstract data type”, defined in terms of a templated C++ class declaration:

```cpp
template <class T>
class stack {
public:
    stack(); // create an empty stack
    stack( const stack<T>& ); // a stack that is a copy of an existing one
    ~stack(); // destroy a stack
    stack<T>& operator=( const stack<T>& ); // assignment operator
    void push( const T& item ); // add item to top
    void pop( ); // remove top
    const T& top() const; // const access to top item
    bool empty( ) const; // is the stack empty?
    int size() const; // return the number of items stored
};
```
Stack Implementations

- We could use a `std::vector<T>` or a `std::list<T>` as the underlying implementation. Both are trivial to implement.
- We could also use a specially-tailored linked list implementation, involving slightly less overhead.
  - There are no major advantages to doing this, but it is a good exercise.
- The standard library has a stack implementation.

Postfix Expressions

- A postfix expression is one where the operators follow the operands instead of being between 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 feature of postfix is that parentheses 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 Weiss text for infix-to-postfix conversion algorithm. You are not responsible for this for the final.
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 is the ‘-’ symbol, which could
  be the start of a negative number or could indicate subtraction.

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

Queues

- 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.
The Queue Class

template <class T>
class queue {
public:
    queue(); // create an empty queue
    queue( const& queue<T>); // copy an existing queue
    ~queue(); // destroy a queue
    queue<T>&
        operator=( const queue<T>& ); // assignment operator
    void enqueue( const T& item ); // add item to the back queue
    void dequeue( ); // remove front of queue
    const T& front( ) const; // return front of queue in item
    bool empty( ) const; // is the queue empty?
    int size() const; // return the number of items stored
};

Queue Implementations

• std::list<T> or std::deque<T>
  
    – We haven’t discussed the latter, but it acts (sort of) like a double-ended array.

• A simplified, singly-linked list can also be used.

• The standard library has a queue implementation.

Stacks vs. Queues

• Queues are FIFO; stacks are LIFO

• Stacks reverse order; queues maintain order

• An interesting example involves using a stack and a queue to determine if a character sequence is a palindrome.

Stacks and Queues: Why Bother?

• $O(1)$ for all operations.

• In many applications, the simplified view of the world crystalizes the solution to a problem.
Course Summary

- The standard library offers container classes and algorithms that simplify the programming process and raise your conceptual level of thinking in designing solutions to programming program.
  
  - While many of you felt the Week 11 homework was hard, it would have been extremely difficult without generic container classes.

- When choosing between algorithms and between container classes (data structures) you should consider
  
  - efficiency,
  - naturalness of use, and
  - ease of programming.

All three are important

- Use classes with well-designed public member functions to encapsulate sections of code.

- Writing your own container classes and data structures usually requires building linked structures and managing memory through the big three:
  
  - copy constructor,
  - assignment operator, and
  - destructor.

Work hard to get these correct.

- When testing and debugging:
  
  - Figure out what your program actually does, not what you wanted it to do.
  - Always find the first mistake in the flow of your program and fix it before considering other apparent mistakes
  - Use small examples and boundary conditions when testing.

- Above all, remember the excitement and satisfaction — developing a deep, computational understanding of a problem and turning it into a program that realizes your understanding flawlessly.