Stacks

Lists are great, but…
- Lists are simply collections of items
  - Useful, but nice to have some meaning to attach to them
  - Restrict operations to create useful data structures
- We want to have ADTs that actually do something useful
  - Example: collecting characters on a line of text
  - Example: doing math with operator precedence (more on this later)
  - Example: matching braces
- All of these applications can use a stack
  - A stack is also an ADT!
  - Stacks can be based on (abstract) lists!

What is a stack?
- A stack is a data structure that keeps objects in Last-In-First-Out (LIFO) order
- Objects are added to the top of the stack
- Only the top of the stack can be accessed
- Visualize this like a stack of paper (or plates)
- Example: track characters entered on a command line
- What methods does a stack need?
  - Create a stack
  - Determine whether a stack is empty (or how many items are on it)
  - Add an object to the top of the stack (push)
  - Remove an object from the top of the stack (pop)
    - Does this return the object removed?
  - Remove all of the objects from the stack
    - Can be done by repeatedly calling pop until the stack is empty
  - Retrieve the object from the top of the stack (peek)

Stack example: matching braces and parens
- Goal: make sure left and right braces and parentheses match
  - This can’t be solved with simple counting
  - Rule: { ok string } is OK, but { (x) } isn’t
  - Rule: ( ok string ) is OK
- Use a stack
  - Place left braces and parentheses on stack
  - When a right brace / paren is read, pop the left off stack
  - If none there, report an error (no match)
  - If stack is not empty when finished examining string, report an error

Stack example: postfix notation
- HP calculators use postfix notation
  - Operations are done by specifying operands, then the operator
    - Example: 2 3 4 + * results in 14
      - Calculates 2 * (3 + 4)
  - We can implement this with a stack
    - When we see a operand (number), push it on the stack
    - When we see an operator
      - Pop the appropriate number of operands off the stack
      - Do the calculation
      - Push the result back onto the stack
    - At the end, the stack should have the (one) result of the calculation
More on postfix notation

- Calculate $5 \times (4 + 3)$
- Numbers ordered 5 4 3
- Operands ordered $\times$ +
- Note reverse order!
- Must compute $\times$ first!
- See example at right
- What about $5 \times 4 + 3$?
- $5 \times 4 + 3$

Postfix is nice, but infix is more common

- Postfix works if you’re used to HP calculators
- Most people are more used to infix
  - Example: $(8\times 4) + 5$
- Can we convert infix to postfix?
  - Yes!
  - Use a stack to do this…
- Observations
  - Operands stay in the same order from infix to postfix
  - Operator $x$ moves “to the right” to ensure that $x$ precedes any operands that it should

How is this done?

- Use a stack and a String
  - Stack for operators being reordered
  - String to build postfix expression
- Rules are
  - Operands always appended to the String
  - “)” pushed onto reorder stack
  - For each operator
    - Pop off reorder stack and append to String until reorder stack is empty or top element is “)” or lower precedence operator
    - Push operator onto reorder stack
  - On “),” pop off reorder stack until “)“ is found
  - Delete “)” postfix needs no parentheses
  - At end of string, pop all off reorder stack and append to String

Example reordering: $a-(b+c*d)/e$

- Operands always appended to postfix String
  - “)” pushed onto reorder stack
  - For each operator
    - Pop operators off reorder stack and append to postfix String until reorder stack is empty or top is “)” or lower precedence operator
    - Push operator onto reorder stack
  - On “),” pop off reorder stack until “)“ is found
  - Delete “)” postfix needs no parentheses
  - At end of string, pop all off reorder stack and append to postfix String

Reorder and Evaluate at same time?

- Use two Stacks
  - A reorder stack
  - A postfix evaluation stack
- Rules are
  - Operands always pushed on the postfix stack
  - “)” pushed onto reorder stack
  - For each operator
    - Pop off reorder stack and perform operation on postfix stack. Repeat until reorder stack is empty or top element is “)” or lower precedence operator
    - Push operator onto reorder stack
  - On “),” pop off reorder stack until “)“ is found
  - Delete “)” postfix needs no parentheses
  - At end of string, pop all off reorder stack and perform operations on postfix stack
Familiar Stacks: Web Browser

- Web Browsers use stacks. Where?
  - To validate HTML tags to make sure that they match.
  - `<b>`-bold text-`</b>`-plain text-`<i>`-italics-`</i>`
  - `<b>`-bold-`</b>`-italics with messed up `<i>`-tag-`</i>`-tags
  - Similar to matching braces and parentheses
  - Stack of pages visited
  - Used by Back and Forward

Using interfaces to declare a stack

- Java has good support for abstract data types
  - An interface is a Java class without any methods
  - Classes may implement interfaces
  - Example: StackInterface
  - May be implemented by array, linked list, etc.
  - We’ll go over implementation on Friday
  - For now, useful to see how to declare functions using interfaces

Interfaces and ADTs

```java
public interface StackADT {
    public int length();
    public void push(Object o);
    public Object pop();
    public Object peek();
    public void popAll();
}
```

```java
public class StackArray {
    final int MAX_STACK = 50;
    private Object items[];
    private int top;
    public StackArray() {} // constructor
    public int length() { return top+1; }
    public void push(Object o) { ... }
    public Object pop() { ... }
    public Object peek() { ... }
    public void popAll() { ... }
}
```

Assignment #2

- Goal: count words in files using linked lists
  - Run wordcount.pl for sample output
  - You may need to do `perl wordcount.pl file1 file2`
  - Break the assignment down into small pieces
    - Do this before writing lots of pseudocode
    - Pieces may include
      - Doubly linked lists (keep the list sorted)
      - Breaking a line up into words
      - Printing the results
      - Parsing the command line
    - You may not use code from elsewhere for
      - Doubly linked lists
      - Breaking a line up into words

Hints for Assignment #2

- Start early!
  - This program is (considerably) more difficult than the first
  - You should have a good design by the end of this week
  - Break the program into manageable pieces
    - It looks really big unless you turn it into lots of short methods
    - It’s relatively easy to write a 10–15 line method
    - It’s hard to write a 100–150 line program
  - Get doubly-linked lists working first
    - Try things out with just one list, rather than one per letter of the alphabet
  - Get word breakup working early
  - Use short files for testing
    - Any files with words, punctuation will work
    - Try using Java source files