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 (from text): collecting characters on a line of text
  - Example: doing math with operator precedence (more on this later)
  - Example: matching braces
- Both 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: function call return stack
- What methods does a stack need?
What methods are needed for a stack?

- 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
  - \{ (x) \} is OK, but \{ (x) \} isn’t
- Rule: \{ ok string \} is OK
- 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)
Stack example: postfix notation

- HP calculators use postfix notation (as do some human languages)
  - 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 $+ \text{ first!}$
- See example at right

5 4 3 + *

$$\begin{array}{c}
7 \\
* \\
5 \\
\hline
35 \\
\end{array}$$
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: \((8*4) + 5\)
- 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 two stacks**
  - One for operators being reordered
  - One for the actual postfix operations

- **Rules are**
  - Operands always pushed onto the postfix stack
  - “(“ pushed onto reorder stack
  - For each operator
    - Pop off reorder stack and push onto postfix stack until empty or “(“ or lower precedence operator
    - Push operator onto postfix stack
  - On “)”, pop off reorder stack until “(“ is found
    - Delete “(“: postfix needs no parentheses
  - At end of string, pop all off reorder stack and onto postfix stack
Example reordering: a-(b+c*d)/e

- Operands always pushed onto the postfix stack
- “(“ pushed onto reorder stack
- For each operator
  - Pop off reorder stack and push onto postfix stack until empty or “(“ or lower precedence operator
  - Push operator onto postfix stack
- On “)”, pop off reorder stack until “(“ is found
  - Delete “(“: postfix needs no parentheses
- At end of string, pop all off reorder stack and onto postfix stack
- Here, do operations rather than push operators onto postfix stack

<table>
<thead>
<tr>
<th>Reorder stack</th>
<th>Postfix stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>d</td>
</tr>
<tr>
<td>+</td>
<td>e</td>
</tr>
<tr>
<td>/</td>
<td>(b+c*d)/e</td>
</tr>
<tr>
<td>-</td>
<td>a-(b+c*d)/e</td>
</tr>
</tbody>
</table>
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
public interface StackADT {
    public int length ();
    public void popAll ();
    public void push (Object o);
    public Object pop ()
        throws StackException;
    public Object peek ()
        throws StackException;
}

public class StackException
    extends RunTimeException {
    ...
}

public class StackArray
    implements StackADT {
    final int MAX_STACK = 50;
    private Object items[];
    private int top;
    public StackArray () {
        // constructor
    }
    public int length () {
        return (top+1);
    }
    ...
}
OK, so stacks are useful

- Stacks have many uses
  - Arithmetic
  - Language parsing
  - Keeping track of recursion (more in this in a week or so)

- How can stacks be implemented?
  - Using a generic List class
    - Works fine, easy to do
    - May not be as efficient
  - Using an array directly
  - Using a linked list

- Tradeoff between generic and tailored implementations
  - Generic implementation: simple, quick
  - Tailored implementation: often more efficient
Review: methods needed for stacks

- Stacks need six methods
  - Create: make a new stack
  - Push: add an element to the top of the stack
  - Pop: remove an element from the top of the stack
  - Peek: examine the element on the top of the stack
  - PopAll: remove all the elements from the stack
  - isEmpty: return true if the stack has no elements

- Implement these methods using
  - Methods existing for a list
  - Operations on an array
  - Linked list operations directly
Stack using a (generic) list

```java
class StackList {
    private List l;
    int size;
    public StackList() {
        l = new List();
        size = 0;
    }
    public void push (Object item) {
        l.insert (item, 0);
        size++;
    }
    public Object pop () {
        Object item = l.index (0);
        l.delete (0);
        size--;
        return (item);
    }
    public Object peek () {
        return (l.index(0));
    }
    public boolean isEmpty () {
        return (size == 0);
    }
    public void popAll () {
        while (!isEmpty()) {
            pop();
        }
    }
}
```
Issue: what about empty lists?

- All this works well if we call pop() with things on the stack
- What if we call pop() on an empty stack?
  - This has no reasonable result!
  - Need to indicate an error somehow
- Solution #1: return a special value
  - Return null if there’s an error
  - Problem: always checking for null!
  - This approach usually taken in C
- Solution #2: generate an exception
What’s an exception?

- An *exception* is an abnormal condition
  - Null reference dereferenced
  - File not found
  - Stack is empty when pop() called

- Exceptions can be dealt with in two ways
  - Handle exception locally
  - Pass it to the calling method

- Pass to calling method
  - Must declare that method can cause an exception:
    ```java
    public Object pop() throws StackException {...}
    ```
  - Calling method must deal with it now!
How can an exception be “caught”?

- Often useful to “catch” an exception
  - Deal with the problem
  - Try an alternate way of doing things
- Exceptions can be caught with a “try…catch” block
  - Different exceptions can be caught separately
  - Not all exceptions need be caught
- Exceptions are objects
  - May have methods
  - May carry information about the error condition

```java
try {
    mystack.pop();
} catch (StackException e) {
    println("Empty stack!");
}
```

```java
while (true) {
    try {
        f = new FileReader(name);
    } catch (IOException e) {
        print ("Enter a new name:");
        // get another name
    }
}
```
public class StackList {
    private List l;
    int size;
    public StackList () {
        l = new List();
        size = 0;
    }
    public Object peek () {
        if (isEmpty()) {
            throw new StackException
                ("Stack empty");
        }
        return (l.index(0));
    }
    public void popAll () {
        while (!isEmpty()) {
            pop();
        }
    }
    public void push (Object item) {
        l.insert (item, 0);
        size++;
    }
    public Object pop () throws StackException {
        if (isEmpty()) {
            throw new StackException
                ("Stack empty");
        }
        Object item = l.index (0);
        l.delete (0);
        size--;
        return (item);
    }
    public boolean isEmpty () {
        return (size == 0);
    }
}
Implementing stacks with arrays

```java
public class StackArray {
    private Object arr[];
    int size;
    private final int max = 20;
    public StackList () {
        arr = new Object[max];
        size = 0;
    }
    public Object peek () {
        if (isEmpty()) {
            throw new StackException
                ("Stack empty");
        }
        return (arr[size-1]);
    }
    public void push (Object item) throws StackException {
        if (size >= max) {
            throw new StackException
                ("Stack full");
        }
        arr[size++] = item;
    }
    public Object pop () throws StackException {
        if (isEmpty()) {
            throw new StackException
                ("Stack empty");
        }
        return (arr[--size]);
    }
    public void popAll ();
    public boolean isEmpty ();
}
```

Issues with arrays for stacks

- Arrays are good for stacks because
  - Pop and push are easy to implement
    - Unlike general lists, only need to insert/delete at end
  - Very space efficient
    - Only require space for object references
    - No need for extra links
  - Fast
    - Some CPUs can do these operations in a single instruction
- Downside of using arrays
  - Stack has a limited size: hard to grow beyond that
  - Entire stack must be allocated even if it’s never used
    - May be inefficient if maximum size is 1000, but stack never exceeds 10 elements
- Arrays for stacks are very common
Implementing stacks with linked lists

```java
public class StackArray {
    private StackArrayNode head;
    int size;
    public StackList () {
        head = null;
        size = 0;
    }
    public Object peek () {
        if (isEmpty()) {
            throw new StackException
                ("Stack empty");
        }
        return (head.val);
    }
    public void push (Object x) {
        head = new StackArrayNode
            (x, head);
        size++;
    }
    public Object pop ()
        throws StackException {
        if (isEmpty()) {
            throw new StackException
                ("Stack empty");
        }
        Object obj = head.val;
        head = head.next;
        return (obj);
    }
    private class StackArrayNode {
        public Object val;
        public StackArrayNode next;
        public StackArrayNode
            (Object x, StackArrayNode n) {
            val = x;
            next = n;
        }
    }
```
Issues with using linked lists as stacks

- Easier to do specific implementation rather than using generic linked lists
  - Only need to insert / delete at head
  - No need to move through the list
- Implementation is efficient, but not as efficient as arrays
  - More space per object (next reference)
  - Slower operations
- No preset limit on stack size
Example

- Let’s implement a stack