Stacks and Queues

Two more containers: Stack and Queue.

Both are lists but with restricted usages.

Stack: add to front, remove from front

Queue: add to back, remove from front.

No other add or remove allowed.

Stack

Process data in the opposite order received:
Last-In, first-out (LIFO).

LIFO is common in real life:
- layoffs
- plates at the cafeteria
- people in elevators
- online content
- vomit

Stack interface

An API for a stack is easy:

- push(x): put x on the top of the stack.
- pop(): remove and return what is top of the stack.
- peek(): return what is top of the stack, but don't remove it.
- isEmpty(): return `true` if and only if the stack is empty.
- size(): return number of emelments.

In Java:

```
Stack<String> s = new Stack<String>();
```

Stack equivalents

```
public class Stack<E> {
    // fields
    private List<E> list;
    // methods
    public          Stack()       { list = new LinkedList<E>(); }
    public void     push(E item)  { list.add(0,item);           }
    public E        pop()         { return list.remove(0);     }
    public E        peek()        { return list.get(0);        }
    public boolean  isEmpty()     { return list.isEmpty();     }
    public int      size()        { return list.size();        }
}
```

Stack example: Eval postfix notation

```
while tokens remain
    x = get next token
    if x is number
       push x on stack
    else                    // x is an operator
       a = pop stack
       b = pop stack
       y = a b x            // combine a and b with operator x
       push y on stack
pop result from stack
```

Queue

Process data in the order received:
First-In, first-out (FIFO).

FIFO is common in real life:
- waiting in line for anything
- voicemail
- stock at stores

Queue interface

An API for a queue is easy:

```
- add(x): put x at the back of the queue.
- remove(): remove and return what is at the front of the queue.
- peek(): return what is at the front of the queue, but don't remove it.
- isEmpty(): return `true` if and only if the queue is empty.
- size(): return number of emelments.
```

In Java:

```
Queue<String> q = new LinkedList<String>();  // Queue is an interface
```

Queue equivalents

```
public class Queue<E> {
    // fields
    private List<E> list;
    // methods
    public          Queue()       { list = new LinkedList<E>(); }
    public void     add(E item)   { list.add(item);              }
    public E        remove()      { return list.remove(0);      }
    public E        peek()        { return list.get(0);         }
    public boolean  isEmpty()     { return list.isEmpty();      }
    public int      size()        { return list.size();         }
}
```

Deque

Practice-It (next lab) requires use of `Stack` and `Queue`.

Java recommends using a `Deque` (double-ended queue) which supports all these methods.

```
Deque<String> stack = new LinkedList<String>();
stack.push("Hello");
String result = stack.pop();

Deque<String> q = new LinkedList<String>();
q.add("Hello");
String result = q.remove();
```

Practice-It Exercise 14.20: interleave.

Write a method interleave that accepts a queue of integers as a parameter and rearranges the elements by alternating the elements from the first half of the queue with those from the second half of the queue. You may use one auxiliary Stack or Queue.

```java
public static void interleave(Queue<Integer> q) {
    int n = q.size();
    if (n%2==1)
        throw new IllegalArgumentException();
    Queue<Integer> aux = new LinkedList<Integer>();
    for (int i=0; i<n/2; i++) {
        aux.add(q.remove());
    }
    for (int i=0; i<n/2; i++) {
        q.add(aux.remove());
        q.add(q.remove());
    }
}
```