Stacks and Queues

- Introduction to abstract data types (ADTs)
- Stack ADT
  - applications
  - interface for Java **`Stack`**
  - ex use: reverse a sequence of values
- Queue ADT
  - applications
  - interface for Java **`Queue`**
- Time permitting: additional **`Comparator`** example

Announcements

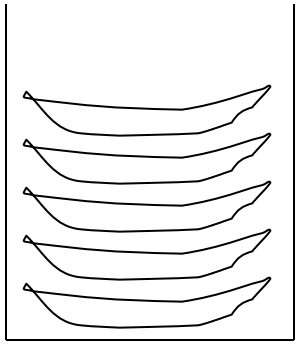
- Next week is Spring Break
- PA3 due after we return
- Lab this week about Exceptions

Abstract Data Types (ADTs)

- An abstract idea of a data structure
- Usually is implemented with a class
- ADT operations = class methods
- Some ADT examples:
  Stack, Queue, Set, Map
- Some concrete data structure examples:
  array, linked list, hash table.
- The ADT is implemented in terms of a concrete data structure
- The ADT usually has more than one possible implementation

Stacks

- a collection of things (like an array is)
- but with restricted access
- the only item you can look at or remove is the *last* item you inserted. Last In First Out (LIFO).
- E.g. stack of dishes
  - push a plate on the top of the stack
  - pop a plate from the top of the stack
  - examine the plate at the top of the stack
  - ask if the stack is empty



Stacks for method call/return

- a second example of a stack is the *system stack* (a.k.a., run-time stack, or call stack)
- element iscalled a *stack frame* (aka, *activation record*):
  - all the data associated with that call: e.g., locals, params, return addr.
- method call/return follows LIFO order.
- last method called will return before any ones that called it.
- Let's look at an example:

```
void main() {
  B();
  D();
  B();
}

void B() {
  C();
  D();
}

void C() {
  D();
}

void D() {
  return;
}
```

Example of method call/return

```
void main() {
  B();
  D();
  B();
}
```

```
void B() {
  C();
  D();
}
```

```
void C() {
  D();
}
```

```
void D() {
  return;
}
```

Java **`Stack`** class interface

```
import java.util.Stack;

Stack<Integer> s = new Stack<Integer>();
                        // creates an empty stack

s.push(3);              // add an element to the top of
                        // the stack

int n = s.peek();       // returns top element in the
                        // stack (does not modify stack)

int top = s.pop();      // pops top element off of
                        // stack and returns it

s.empty();              // tells whether stack is empty
```

Using **`Stack`** operations

```
Stack<Character> s = new Stack<Character>();
         // creates an empty stack of characters

s.push('a');
s.push('b');
s.push('c');
System.out.println(s.peek());

s.pop();
s.push('d');
System.out.println(s.peek());

s.pop();
s.pop();
System.out.println(s.empty());
```

Ex: Reversing a sequence

- Stacks are good for reversing things
  - The last item you put on is the first one you get out
  - The second to last item you put on will be the second item you get out, etc.
- Example problem: read in a bunch of integer values and then print them out in reverse order.

Stack representations

- How to represent?
- Where should top be?
- What is big-O of each operation?

Queue

- A container of elements that can be accessed/inserted/removed like standing in line.
- Enter queue at the end (enqueue)
- Exit queue at the front (dequeue)
- Can access front element
- First-in First-out (FIFO)

- Example of Comparator interface

Applications of Queues

- Major application is to represent lines (queues) in software.

- E.g., big in operating systems
  - print queue – print jobs waiting to print. They are processed in FIFO order.
  - process queue – processes waiting for their turn to execute.

Queue applications (cont.)

- also in simulations
  - queue of events waiting to be processed
  - simulating queues from the world we are simulating (e.g., Bank line, airplanes waiting to take off)
- and Java GUI system
  - queue of user input events waiting to be processed
  - (e.g., mouse clicks, keyboard strokes)

Java **`Queue`**

- **`Queue`** is an **`interface`** rather than a class.
- **`LinkedList`** implements this interface.
- To create one:

```
Queue<MyType> q = new LinkedList<MyType>();
```

- means we intend to only use the LL ops specified by **`Queue`**.

Java **Queue** interface

```
import java.util.Queue;
import java.util.LinkedList;
Queue<Integer> q = new LinkedList<Integer>();
                // creates an empty queue of integers

q.add(3);          // add an element to the end

int n = q.peek();  // returns first element in
                // the queue (does not modify queue)

int front = q.remove();
                    // removes first element from queue
                    // and returns it

q.isEmpty();       // tells whether queue is empty
```

Using **queue** operations

```
Queue<Character> q = new LinkedList<Character>();

q.add('a');
q.add('b');
q.add('c');
System.out.println(q.peek());

q.remove();
q.add('d');
System.out.println(q.peek());

q.remove();
q.remove();
System.out.println(q.isEmpty());
```

Queue representations

- How to represent?
- Where is front and end?
- What is big-O of each operation?

Related data structures

- Why is **`Queue`** an interface?
  - some other queue implementations are used in multi-thread Java applications
  - one queue implementation is a *priority queue*
    - not FIFO, but…
    - an element's priority determines how soon it gets to the front of the line.

Additional `Comparator` example

- Problem: sort an array of `Rectangle`'s in increasing order by area.
- Do not implement your own sort method!

```
public static void sortIncrByArea(
                              Rectangle[] rects) {

    Arrays.sort(
```