CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #11

John Ridgway

March 5, 2015

1 More on Recursion

Recursion and Explicit Stacks

- Recall the recursive `markBlob` method from last lecture. To mark all the land squares reachable from (x, y), we mark (x, y) itself and then call `markBlob` on each of its neighbors.
- We could have used our own stack instead. To explore a node, we put it on the stack and then explore its unseen land neighbors. When we have done all its neighbors, we pop it off.
- You will see such DFS algorithms in CMPSCI 250 and CMPSCI 311.

Removing Recursion

- Some of the recursions we have seen so far are examples of *tail recursion.*
- A tail-recursive method makes exactly one call to itself in the general case, and it occurs at the end of the code.
- Thus the first thing that happens is the sequence of recursive calls is made until there is a call to a base case. Then each call finishes, returning a value to the method that called it.

Tail Recursion Example

```
public void clear(StackInterface s) {
   if (s.isEmpty()) return;
   s.pop();
   clear(s); }

public void loopClear(StackInterface s) {
   while (!s.isEmpty()) {
      s.pop(); }
```

If the parameter for the recursive call is different from the parameter for the original call, we can represent this by variables declared outside the loop.

Tail Recursion

```
public int factorial(int n) {
   if (n == 0) return 1;
   return n * factorial(n-1); }

public int tailfact(int n, int acc) {
  // returns n! * acc
   if (n == 0) return acc;
   return tailfact(n - 1, n * acc); }
```

Strictly speaking, a tail recursion can have no computation after the final recursive call, so the top method is not a tail recursion. But the bottom one is, and we may call `tailfact(n, 1)`.

Removing Recursion

- With a bit more work we can replace the original factorial method by one with a non-recursive loop, shown below.
- In this case, we have a value `retValue` which we set to `1` at the beginning, so that we will correctly return `1` if `n` is `0` and there is no recursion.

```
public int factorial(int n) {
  int retValue = 1;
  while (n > 0) {
    retValue *= n; n -= 1; }
  return retValue; }
```

Removing Recursion (continued)

- We then change `retValue` to reflect the action of each version of the method, also keeping the value of `n` so that we will know when we reach the base case.
- (Note that DJWs code reverses the order of the multiplications of the recursive version.)

Stacking for Non-Tail Recursion

- In the recursive `revPrint` method we saw earlier, each run of the method had at most one recursive call to `revPrint`, but we did not have a tail recursion because there was code to run after the recursive call.
- To simulate this without recursion, we note that the print statement will execute on the elements successively, operating on the element that would be top element of the calling list at that time.

Stacking for Non-Tail Recursion (continued)

We need to record, on the way down the list, the information that the last statement will need. An explicit stack does this for us.

```
public void printReversed() {
     USI<T> stack = new LinkedStack<T>();
     LLNode<T> listNode = top;
     while (listNode != null) {
       stack.push(listNode.getInfo());
       listNode = listNode.getLink(); }
     while (!stack.isEmpty()) {
       System.out.println(
                   " " + stack.top( ));
       stack.pop(); } }
```

Clicker Question #1

```
private int fib(int n) {
  if (n <= 0) return n;
  return fib(n-1) + fib(n-2); }
```

Suppose we tried to use this stacking technique to eliminate the recursion in the recursive Fibonacci method from Discussion #5. Would this work?

A. No, because the method makes two separate recursive calls.

B. Yes.

C. No, because the method is private.

D. No, because ints cannot go on a stack.

Whether to Use Recursion

- Using recursion incurs costs, as do any of the techniques we could use to replace the recursion. Whether to use recursion for a given problem depends on the tradeoffs between these costs.
- A recursive method uses space on the call stack, which may be a limited resource. It uses some additional time to create and dispose of activation records, and these actions may use more memory than is needed by the iterative version.

Whether to Use Recursion (continued)

- The iterative version, on the other hand, may require more lines of code for the same job, and may be harder to debug because the algorithm is further removed from the definition of the problem. (Our Three Questions are enormously powerful for verifying completion and correctness.)
- Well also see next that recursive versions can sometimes be far slower.
- The ideal situation would be to write and debug a recursive version and have it automatically converted to an iterative version by a compiler.

Very Bad Recursive Algorithms

- Discussion #6 featured both a recursive and a non-recursive method to compute Fibonacci numbers.
- The non-recursive method computed $F(n)$ in $O(n)$ time, as it did a constant amount of work for each i from 1 to n.
- The time for the recursive method turned out to be proportional to $F(n)$ itself, which is an exponential function of n (about 1.61^n, as you will probably see in CMPSCI 250).

Very Bad Recursive Algorithms (continued)

- Lets look at how this discrepancy happens, using another example.
- I can define the function $f(x) = 2^x$ recursively: $f(0) = 1$ and $f(x) = f(x-1) + f(x-1)$. If I use this definition to write a recursive algorithm, it will make 2^x different calls to $f(0)$ and add their results.

```
public static int twoToThe(int n) {
  if (n == 0) return 1;
  else
    return twoToThe (n-1) + twoToThe (n-1); }
```

- But of course a simple loop, that sets `y` to `1` and then multiplies it by two x times, is $O(x)$.

Clicker Question #2

```
public static int method1(int n) {
     if (n == 0) return 1;
     else return
        method1(n-1) + method1(n-1); }
public static int method2(int n) {
     if (n == 0) return 1;
     else return
        2 * method2(n-1); }
```

Consider these two methods that are supposed to input n and output 2^n. Which is true?

A. the first is incorrect

B. the second is much faster

C. the first is much faster

D. both are incorrect

Very Bad Recursive Algorithms (continued)

DJW give a third example, foreshadowing a problem you will see in CMPSCI 240: If we have a group of n members, our problem is to count the number of subsets of the group that have exactly k members. Here is a correct but horribly slow method:

```
public int combinations(int group, int members) {
  if (members == 1) return group;
  else if (members == group) return 1;
  else return (combinations(group - 1, members - 1) +
               combinations(group - 1, members)); }
```

Very Bad Recursive Algorithms (continued)

- $C(g, 1) = g$ as there are g one-member subsets.
- $C(g, g) = 1$ as there is one g-member subset.
- In CMPSCI 240 youll talk about why $C(g, m) = C(g-1, m-1) + C(g-1, m)$. This is the identity behind Pascals Triangle.

Pascals Triangle

- Of course this is called Yang Huis Triangle in China, and Yang Hui beat Pascal by several centuries.

Very Bad Recursive Algorithms

- If we try to find $c(5, 3)$ with this method, it asks for $c(4, 2)+c(4, 3)$, which then becomes $c(3, 1) + c(3, 2) + c(3, 2) + c(3, 3)$, and then $3 + c(2, 1) + c(2, 2) + c(2, 1) + c(2, 2) + 1 = 3 + 2 + 1 + 2 + 1 + 1 = 10$.
- Lots of repeated calls, more as input grows.

2 Queues in the Abstract

The Queue Operations

- A **queue** is a data structure holding a collection of objects.
- Like a stack, it allows insertion of new elements and removal of existing elements.
- Unlike a stack, it allows removal of only the *earliest-inserted* element, rather than the most recently inserted one.
- This is called **first-in-first-out** or **FIFO**.

The Queue Operations

- In DJWs vocabulary, we add an element by **enqueueing** it, whereupon it is at the **rear** of the queue. We can remove the element at the **front** of the queue by **dequeueing** it.
- Queue is another word for waiting line, particularly in British English, and our queues operate like those at a supermarket — you enter at the rear and wait until you reach the front and are served.

The Queue Operations

- The queue classes in `java.util` have a different vocabulary which well review later.
- DJWs queue classes do not have a way to look at the item at the front of the queue without removing it, and this time they do not appear to care that `dequeue` is both an observer and a transformer.

Clicker Question #3

Assuming all the variables are of type `Dog` and have non-`null` values, which dogs information will be printed by this code?

```
Queue<Dog> q = new Queue<Dog>();
q.enqueue(cardie); q.enqueue(duncan);
Dog x = q.dequeue();
q.enqueue(ebony); q.dequeue();
Dog y = q.dequeue(); q.enqueue(x);
System.out.println(q.dequeue());
```

A. Duncan

B. Ebony

C. none, an exception will be thrown

D. Cardie

Using Queues: Repeating Strings

- DJW have a sample program that first allows the user to input exactly three lines of text, then outputs them in the same order they were input.
- Earlier we had a very similar program where the strings were put into a stack instead of a queue. Of course in that case they came out of the stack in reverse order.

Using Queues: Repeating Strings

In the code, BQI abbreviates BoundedQueueInterface.

```
public class RepeatStrings {
  public static void main(String[] args) {
    Scanner conIn = new Scanner(System.in);
    BQI<String> queue =
      new ArrayBndQueue<String>(3);
    String line;
    for (int i = 1; i <= 3; i += 1) {
      System.out.println
        ("Enter a line of text > ");
      line = conIn.nextLine();
      queue.enqueue(line); }
    System.out.println("Order is:\n");
    while (!queue.isempty()) {
      System.out.println(queue.dequeue());}}}
```

The Queue Interfaces

As they did for stacks, DJW define three separate interfaces for the Queue abstract data type. One is for bounded queues that could conceivably become full, and the other is for unbounded queues that can always find more memory.

```
public interface QueueInterface<T> {
  T dequeue() throws QueueUnderflowException;
  boolean isEmpty(); }
public interface BQI<T> extends QI<T> {
  void enqueue(T element)
    throws QueueOverflowException;
  boolean isFull(); }
public interface UQI<T> extends QI<T> {
  void enqueue(T element); }
```

The Queue Interfaces

Remember that since the queue exceptions are unchecked (they extend `RuntimeException`), the `throws` clauses for them are advisory rather than required.

```
public interface QueueInterface<T> {
  T dequeue() throws QueueUnderflowException;
  boolean isEmpty(); }
public interface BQI<T> extends QI<T> {
  void enqueue(T element)
    throws QueueOverflowException;
  boolean isFull(); }
public interface UQI<T> extends QI<T> {
  void enqueue(T element); }
```

The Queue Interfaces

Also note that the central meaning of a queue, that dequeuing returns the element that has been in the queue the longest, is not guaranteed by the interface — the interface only defines the possible interactions with the queue.

```
public interface QueueInterface<T> {
  T dequeue() throws QueueUnderflowException;
  boolean isEmpty(); }
public interface BQI<T> extends QI<T> {
  void enqueue(T element)
    throws QueueOverflowException;
  boolean isFull(); }
public interface UQI<T> extends QI<T> {
  void enqueue(T element); }
```

Clicker Question #4

Next lecture well see DJWs generic classes `ArrayQueue<T>`, which implements `BQI<T>`, and `LinkedQueue<T>`, which implements `UQI<T>`. Which of these statements is false?

A. both classes implement `QI<T>`

B. any class extending `LQ<T>` implements `UQI<T>`

C. `LinkedQueue<T>` implements `BQI<T>`

D. `AQ<T>` implements any interface extended by `BQI<T>`

Queues in `java.util`

- In `java.util`, `Queue` is an interface that is part of the Collections Framework. It is implemented by various classes such as `ArrayQueue`.
- It usually acts as a FIFO queue, but can be defined to work in other ways.
- For example, well later study another data structure called a **priority queue**. In Java, the `PriorityQueue` class implements the Queue interface, although priority queues are not FIFO.

Enqueuing in `java.util`

- Enqueuing can be done with either of two methods: `add`, which throws an exception if the queue is full, and `offer` which returns `false` if it is full.
- Both methods return a boolean which is true if the enqueuing succeeds.

Dequeuing and Peeking

- Dequeuing can be done with either of two methods as well: `remove`, which throws an exception if the queue is empty, and `poll`, which returns `null` (not false, as DJW say on page 305) if it is empty.
- There are also two methods to return the front element without removing it. If the queue is empty, `element` throws an exception but `peek` returns null.