CSCI 136
Data Structures & Advanced Programming

Jeannie Albrecht

Lecture 17

March 19, 2014

Administrative Details

- Lab 6 is today
  - Can work with a partner again
- Due Tuesday after break
  - Not sure about TA availability on Sunday and Monday after spring break (I'll try to find out)

Last Time

- Began discussing stacks
- Learned about infix and postfix
- Talked about how stacks can be used to solve mazes

Today's Outline

- Finish up stacks
- Learn about queues

Implementing Maze

- Iteratively: Maze.java
- Recursively: RecMaze.java
  - Recursive methods keep an implicit stack
  - Each recursive call adds another layer to the stack

Maze Solver (iterative)

```
public boolean solve(Maze maze) {
       Stack<Position> path = new StackList<Position>();

       Position current = maze.start();
       maze.visit(current);
       path.push(current);

       while (!path.empty() && !path.peek().equals(maze.finish())) {
           Position next = nextAdjacent(maze, path.peek());
           if (next != null) {
               maze.visit(next);
               path.push(next);
           } else {
               // No adjacent positions left to try.
               // Pop position and pick up path on previous.
               path.pop();
           }
       }
       if (!path.empty()) {
           System.out.println(path);
       }
       return !path.empty();
   }
```

Maze Solver (recursive)

```
public boolean solve(Maze maze, Position current) {


       maze.visit(current);

       if (current.equals(maze.finish())) { return true; }

       else {
          Position next = nextAdjacent(maze, current);
          while (next != null && !solve(maze, next)) {
              next = nextAdjacent(maze, current);
          }
       }
       return next != null;
   }
```

Implementing Maze

- Iteratively: Maze.java
- Recursively: RecMaze.java
  - Recursive methods keep an implicit stack
  - Each recursive call adds another layer to the stack
  - Where should we print our path?
- Question: What is the worst/average Big-O runtime of our maze solver?

Maze Solver (recursive)

```
public boolean solve(Maze maze, Position current) {


       maze.visit(current);

       if (current.equals(maze.finish())) { return true; }

       else {
          Position next = nextAdjacent(maze, current);
          while (next != null && !solve(maze, next)) {
              next = nextAdjacent(maze, current);
          }
          if (next!=null) {
               System.out.print(next+" ");
          }

       }
       return next != null;
   }
```

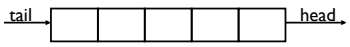
Method Call Stacks

- In JVM, need to keep track of method calls
- JVM maintains stack of method invocations (called frames)
- Stack of frames
  - Receiver object, parameters, local variables
- On method call
  - Push new frame, fill in parameters, run code
- Exceptions print out stack
  - Example: StackEx.java
- Recursive calls recurse too far: StackOverflowException
  - Overflow.java (from last class)
- Recursive call stacks: factorial.java (from last class)

Stacks vs. Queues

- Stacks are LIFO (Last In First Out)
  - Methods: push, pop, peek, empty
  - Used for:
    - Evaluating expressions (postfix)
    - Solving mazes
    - Evaluating postscript
    - JVM method calls
- Queues are FIFO (First In First Out)
  - Another linear data structure (implements Linear interface)
  - Queue interface methods: enqueue (add), dequeue (remove), getFirst (get), peek (get)

Queues



- Examples:
  - Lines at movie theater, grocery store, etc
  - OS event queue (keeps keystrokes in order)
  - Printers
  - Routing network traffic (more on this later)

Queue Interface

```
public interface Queue<E> extends Linear<E> {
  public void enqueue(E item);
  public E dequeue();
  public E getFirst(); //value not removed
  public E peek();  //same as get()
}
```

Implementing Queues

Like Stacks, we have a three options:

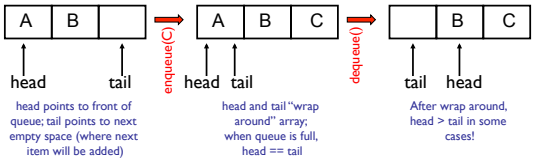
1. QueueVector
```
class QueueVector<E> implements Queue<E> {
    protected Vector<E> data;
 }
```
2. QueueList
```
class QueueList<E> implements Queue<E> {
    protected List<E> data; //uses a CircularList
 }
```
3. QueueArray
```
class QueueArray<E> implements Queue<E> {
    protected Object[] data; //can't declare E[]
    int head;
    int count;
 }
```

Tradeoffs:

- QueueVector:
  - enqueue is O(1) (but O(n) in worst case - ensureCapacity)
  - dequeue is O(n)
- QueueList:
  - enqueue is O(1) (addFirst)
  - dequeue is O(1) (DLL/CLL removeLast)
- QueueArray:
  - enqueue is O(1)
  - dequeue is O(1)
  - Faster operations, but limited size

QueueArray

- Let's look at an example…
- How to implement? (on board)
  - enqueue(item), dequeue(), size()




```
public class queueArray<E> {

    protected Object[] data;
    protected int head;
    protected int count;

   public queueArray(int size) {
         data = new Object[size];
   }

   public void enqueue(E item) {
        Assert.pre(count<data.length,"Queue is full.");
        int tail = (head + count) % data.length;
        data[tail] = item;
        count++;
   }

   public E dequeue() {
         Assert.pre(count>0,"The queue is empty.");
         E value = (E)data[head];
         data[head] = null;
         head = (head + 1) % data.length;
         count--;
         return value;
   }

    public boolean empty() {
         return count>0;
    }
```