Eric Roberts
CS 106B

Handout #36
February 13, 2015

Answers to Midterm Exam

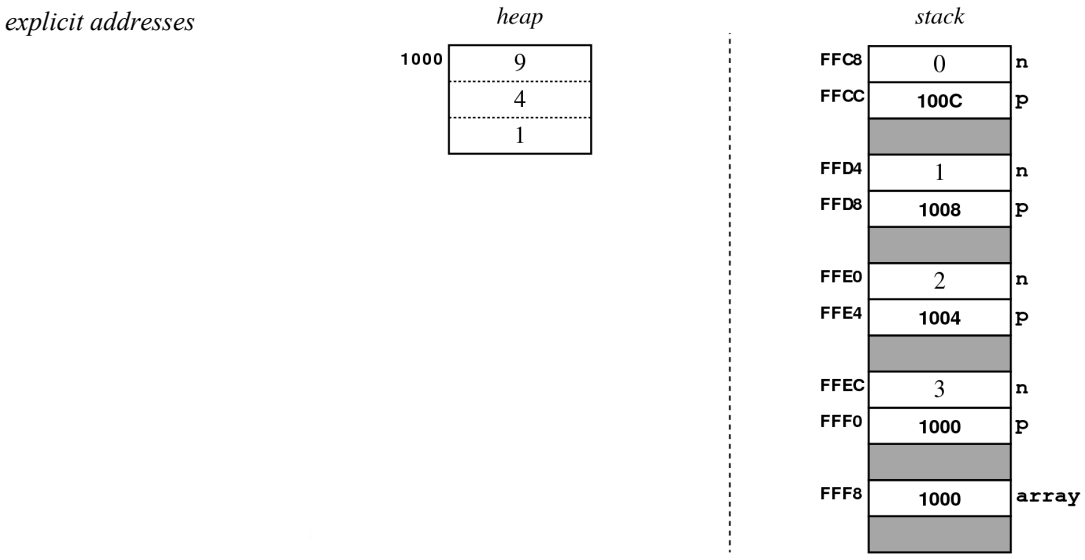
Most of you did well on this exam. There were 12 perfect scores, and the median was 51 out of 60 (85%). Following my usual practice when the class does well, I have pegged the median as the highest B+. The grade assignments for the scores look like this:

Grade	Range	N
A+	59–60	26
A	55–58	97
A–	52–54	72
B+	49–51	54
B	45–48	61
B–	42–44	30
C+	39–41	27
C	33–38	17
C–	30–32	12
D	22–29	17
NP	00–21	11

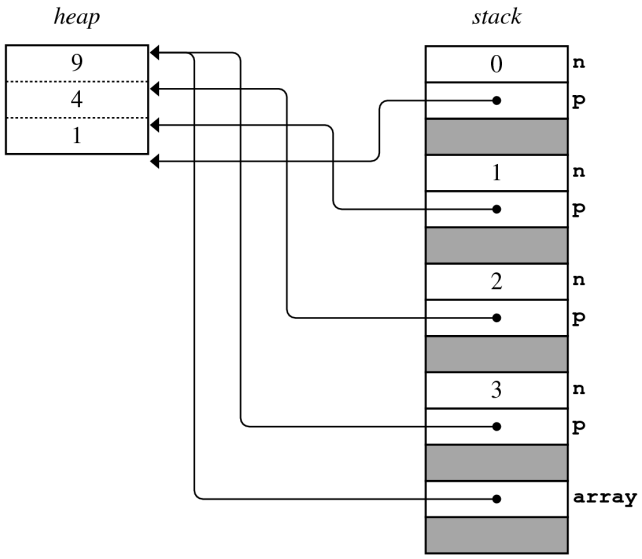
I am, however, concerned that there are quite a few very low scores. If you got fewer than half the points on this exam, you should consider that outcome to be a wake-up call that indicates that you have some catching up to do. Please come and talk with me or Kevin during office hours, and we'll see if we can find a plan to get you back on track.

Problem 1: Function tracing and/or memory diagramming (10 points)

Afternoon exam:

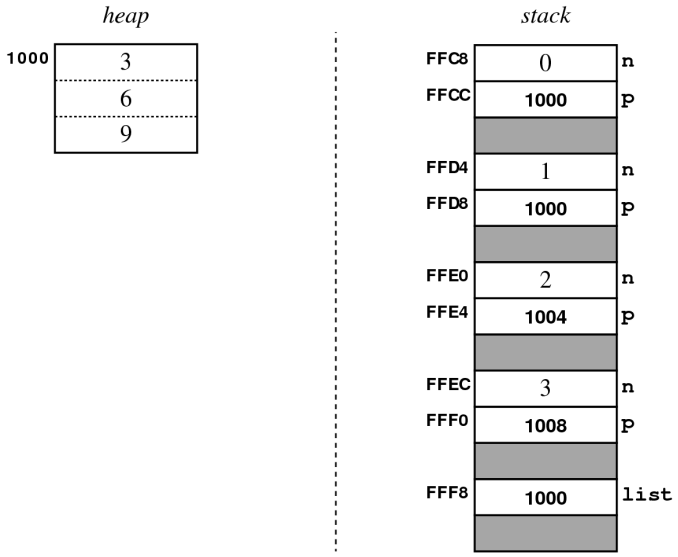


arrow representation

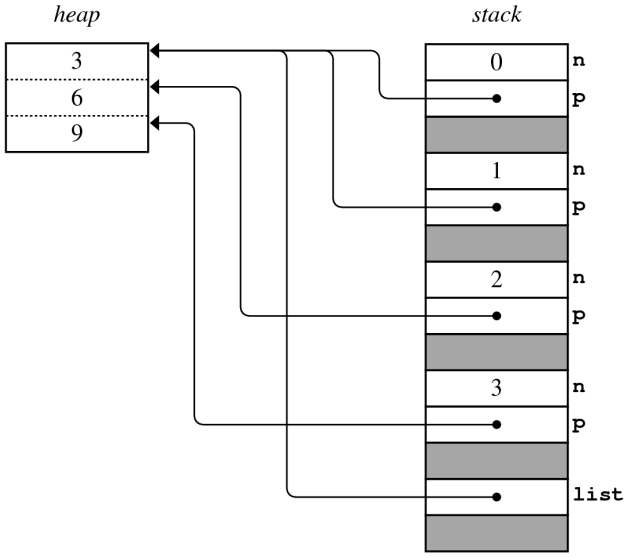


Evening exam:

explicit addresses



arrow representation



Problem 2: Vectors, grids, stacks, and queues (10 points)

```
/*
 * Function: splitQueue
 * Usage: splitQueue(oldQueue, newQueue, max);
 * -------------------------------------------
 * Removes all customers from oldQueue who have no more than max items
 * in their shopping cart and transfers them, preserving the same order,
 * into newQueue.  Any customer with more than max items remains in
 * oldQueue when splitQueue returns.
 */

void splitQueue(Queue<Customer> & oldQueue,
                Queue<Customer> & newQueue, int max) {
   Queue<Customer> copy = oldQueue;
   oldQueue.clear();
   while (!copy.isEmpty()) {
      Customer c = copy.dequeue();
      if (c.numberOfItems > max) {
         oldQueue.enqueue(c);
      } else {
         newQueue.enqueue(c);
      }
   }
}
```

Problem 3: Lexicons, maps, and iterators (15 points)

```
/*
 * Function: createRhymingDictionary
 * Usage: createRhymingDictionary(lex, dict, minSuffix, maxSuffix);
 * ----------------------------------------------------------------
 * Creates a rhyming dictionary for all suffixes whose lengths are in
 * the range given by minSuffix and maxSuffix. The map stored in the
 * reference parameter dict associates suffixes with a set of words that
 * "rhyme" with that suffix, in the sense that the spelling is the same.
 */

void createRhymingDictionary(Lexicon & lex,
                             Map<string,Set<string>> & dict,
                             int minSuffix, int maxSuffix) {
   for (string word : lex) {
      int p1 = word.length() - maxSuffix;
      int p2 = word.length() - minSuffix;
      if (p1 < 0) p1 = 0;
      for (int i = p1; i <= p2; i++) {
         dict[word.substr(i)].add(word);
      }
   }
}
```

On the evening midterm, the second argument was called **`rhymes`**.

Problem 4: Recursive functions (10 points)

```
/*
 * Function: maxHailstoneNumber
 * Usage: int max = maxHailstoneNumber(limit);
 * -------------------------------------------
 * Returns the largest hailstone number (i.e., the largest number of steps)
 * for the integers between 1 and limit, inclusive.
 */

int maxHailstoneNumber(int limit) {
   if (limit <= 1) {
      return 0;
   } else {
      int largest = maxHailstoneNumber(limit - 1);
      int current = hailstoneNumber(limit);
      if (current > largest) largest = current;
      return largest;
   }
}

/*
 * Function: hailstoneNumber
 * Usage: nSteps = hailstoneNumber(n);
 * -----------------------------------
 * Returns the hailstone number of n, which is the number of steps
 * required to reach 1 by executing the following steps repeatedly:
 *
 * - If n is 1, stop.
 * - If n is even, replace n by n / 2.
 * - If n is odd, replace n by 3 * n + 1.
 *
 * The recursive formulation of this function appeared on the answers
 * to the second practice exam.
 */

int hailstoneNumber(int n) {
   if (n == 1) {
      return 0;
   } else if (n % 2 == 0) {
      return 1 + hailstoneNumber(n / 2);
   } else {
      return 1 + hailstoneNumber(3 * n + 1);
   }
}
```

You can simplify the coding of the recursive case by using the **max** function that we wrote in class. If you do so, the recursive case of **maxHailstoneNumber** is simply

```
return max(hailstoneNumber(limit), maxHailstoneNumber(limit - 1));
```

On the evening exam, the argument name **limit** was changed to **max**.

Problem 5: Recursive procedures (15 points)

I was pleased when one of the students at the evening exam came up to me at the end and said that she had done the Jammed Tractor Maze at Blenheim Palace. They have several other mazes there, and it's worth a visit if you're ever in Oxford:

```
/*
 * Function: solveUsingLeftTurns
 * Usage: if (solveUsingLeftTurns(maze, pt, dir)) . . .
 * ----------------------------------------------------
 * This function attempts to generate a solution to the current maze
 * from point pt after having just moved here in direction dir.  This
 * implementation supports going straight or turning left, but not
 * turning right or backing up.
 */

bool solveUsingLeftTurns(Maze maze, Point start, Direction dir) {
   HashMap<Point,Set<Direction>> directionsTried;
   return recursiveSolver(maze, start, dir, directionsTried);
}

/*
 * Function: recursiveSolver
 * Usage: if (recursiveSolver(maze, pt, dir, directionsTried)) . . .
 * -----------------------------------------------------------------
 * This function does the real work for solveUsingLeftTurns.  The
 * additional directionsTried argument is a map between points in the
 * maze and the set of directions that have been tried at that point.
 */

bool recursiveSolver(Maze & maze, Point pt, Direction dir,
                     HashMap<Point,Set<Direction>> & directionsTried) {
   if (maze.isOutside(pt)) return true;
   if (directionsTried[pt].contains(dir)) return false;
   directionsTried[pt].add(dir);
   for (int i = 0; i < 2; i++) {
      if (!maze.wallExists(pt, dir)) {
         if (recursiveSolver(maze, adjacentPoint(pt, dir), dir,
                             directionsTried)) {
            return true;
         }
      }
      directionsTried[pt].remove(dir);
      dir = leftFrom(dir);
   }
   return false;
}

/* The adjacentPoint function is defined in the text. */

Point adjacentPoint(Point start, Direction dir) {
   switch (dir) {
      case NORTH: return Point(start.getX(), start.getY() - 1);
      case EAST:  return Point(start.getX() + 1, start.getY());
      case SOUTH: return Point(start.getX(), start.getY() + 1);
      case WEST:  return Point(start.getX() - 1, start.getY());
   }
   return start;
}
```