CMPSCI 187: Programming With Data Structures

Lecture #18: Applications of Lists
21 October 2011

Applications of Lists

- Pairing Teams in a Tournament
- The Team Class for the Tournament
- The TournamentMaker Class
- The Josephus Problem
- An Ordered List as a Priority Queue
- Project #5: Finding the Best Path

Pairing Teams In a Tournament

- Sports leagues typically play a ``regular season" of multiple games followed by a tournament, where some of the teams compete for a championship. The results of the season are used to determine which team plays which in the tournament. We will consider **single-elimination** tournaments where any team that loses a game is eliminated.

- In Section 6.2 L&C use an ordered list to compute which team in a bowling league ought to play which other in the first round of the tournament. Their algorithm requires that the number of teams is a power of two, and checks this.

- The basic idea is simple -- add the teams to the list one by one, maintaining its order according to number of regular-season wins. Then repeatedly remove the first and last elements of the list, pairing them with each other.

The `Team` Class for the Tournament

- We need this class to implement the `Comparable<Team>` interface so that we can put Team objects in an ordered list. That means we need to write the `CompareTo` method as specified in the definition of `Comparable`.

```
public class Team implements Comparable<Team> {
   public String teamName;
   private int wins;
   public Team (String name, int numWins) {
      teamName = name; wins = numWins;}
   public String getName ( ) {return teamName;}
   public int compareTo (Team other) {
      if (this.wins < other.wins) return -1;
      if (this.wins == other.wins) return 0;
      return 1;}
   public String toString ( ) {return teamName;}
```

The `TournamentMaker` Class

- Suppressing I/O details, this method takes the team names and stats and prints out the schedule for the first round.

```
public class TournamentMaker {
   public void make ( ) throws IOException {
      OrderedListADT<Team> tournament =
         new ArrayOrderedList<Team>( );
      String team1, team2, teamName;
      int numWins, numTeams = 0;
      // I/O stuff to get user input
      // get number of teams, check it is power of 2, > 2
      for (int count = 1; count <= numTeams; count++)
         // get team name and numWins from user
         tournament.add (new Team (teamName, numWins);
      for (int count = 1; count <= numTeams/2; count++) {
         team1 = tournament.removeFirst( ).getName( );
         team2 = tournament.removeLast( ).getName( );
         // announce "Game #(count) is team1 vs. team 2"
         }}}
```

The Josephus Problem

- In 67 C.E., the historian Flavius Josephus was (by his account) one of 41 Jewish rebels about to be killed or captured at the end of the siege of Yodfat. They all resolved to die rather than surrender, and so agreed to form a circle and kill every third person in the circle until none were left. Through luck, cleverness, or divine providence, Josephus was one of the last two, and lacking a third person to kill they agreed to surrender to the Romans and Josephus lived to tell the story.

- In Section 6.3 L&C define a class that simulates the sequence of executions, this time going to the last person, for any given initial number and any given gap size. They use an indexed list, so that they can remove the person at a known position and then update that position to reflect the gap.

- *"Suicide is forbidden by Roman law -- the penalty is death!"* (film version of *A Funny Thing Happened on the Way to the Forum*)

Code for the Josephus Simulation

- The `ArrayList` class lets us remove or look at the entry at a particular index. This code is slightly adjusted from that in L&C. Note that the indices are numbered 0 through n-1, even though we have named the entries from 1 to n. Why do we add `gap - 1` rather than `gap`?

```
public void josephus (int startNum, int gap) {
   ArrayList<Integer> list = new ArrayList<Integer>( );
   for (int i = 1; i <= startNum; i++)
      list.add (new Integer(i));
   circleSize = startNum;
   counter = gap - 1;
   while (!list.size( ) > 1) {
      int victim = list.remove (counter);
      System.out.println ("Next to die is # " + victim);
      circleSize--;
      counter = (counter + gap - 1) % circleSize;}
   System.out.println
      ("The lucky survivor is # " + list.get(0));}
```

An Ordered List as a Priority Queue

- A **priority queue** is a fundamental data structure where we may put items in and get out the *best* element on request, according to some ordering on the elements. For example, stacks and queues are priority queues where "best" means "newest" for the stack and "oldest" for the queue.

- With the `add` and `removeFirst` methods of an ordered list, we get the functionality of a priority queue, assuming that we define "best" to be "first in the natural order". The elements of the list are lined up in order from lowest to highest. We can remove and return the lowest, or add a new element, which the list will put where it belongs in the order.

- We don't expect O(1) performance for priority queue operations. Using our implementations of ordered lists will give us O(n). Java's `PriorityQueue` class gets O(log n) performance by using a heap, as we will see later.

Project #5: Finding the Best Path

- Our next programming project is yet another version of finding paths in a maze. This time there will be costs for entering each open cell, and our goal is to find a path from the source to the destination that minimizes the total cost along it.

- Our framing story, taken from the 2003 ACM programming contest, says that a merchant has a stock of silver spoons at the source, and can sell them at the destination. But on the way are villages and towns (in our version, every cell will contain either a town or a village). To enter a village costs one spoon, and to enter a town costs five percent of the spoons carried in, rounded up. Thus which path is best might depend on how many spoons you start with.

- Our main idea for finding the best path is to replace the queue of Project #4 with a priority queue, where cells will be ordered by how many spoons we can bring there by the best route.

Searching With a Priority Queue

- As before, when we take a cell x off the queue, we look for its open unseen neighbors and put them on to the queue.

- But this time, we mark each neighbor y as to how many spoons we can bring there, by following the best path to x and then going directly to y. We no longer mark y as "seen" until it comes off the queue, because we may not yet have the best path. Again we save a "parent" pointer to x when we put y on.

- The first time a node comes off the queue, it is labeled with the best path to it. (We will prove this fact formally in CMPSCI 250.) If it comes off the queue again, we can ignore it because we have already found and processed a better path.

- When the destination comes off the queue, we reconstruct the path (using the parent pointers as for the ordinary queue) and output it.