# Lab 3: Linked Lists and the Josephus Problem

**Due Thursday, Feb 17, 2011 by 11:59PM EDT**

---

## Concepts Covered

The following concepts are covered in this lab

- Implementing a singly linked list library
- Solving the josephus problem with LL's
- Debugging with Testers

---

## PART I : Implementing a Linked Lists Library (60 points)

The part I of this assignment is to implement a comprehensive library of linked list manipulation functions. We define a linked list node as:

```
typedef struct node{
  int data;
  struct node* next;
} node;
```

Each node contains a data field (int) and a pointer to the next node (struct node*). The function prototypes to be implemented in part I are given in circularll.h. Here is a list of the function prototypes.

```
/* function prototypes */
void freeAll(node*);
int size(node*);
void append(node**,int);
node* prepend(node**,int);
void insertAt(node**,int, int );
char* toString(node*);
int contains(node*, int );
int isEmpty(node*);
int removeAt(node**, int );
node* rotate(node**, int );
int elementAt(node*,int);
void doCircular(node*);
void undoCircular(node*);
node* sort(node*);
int isCircular(node* );
```

A short description of the function, pre conditions and post conditions of the functions are given in the implementation file circularll.c.

As usual, you must consider all edge cases, when implementing these functions. Functions will be tested with assertions using the file clltester.c.

We will continue to modify this file to make sure we cover all possible cases. As you develop your code, get the latest clltester.c from the downloads and test. You can always compile and run your code as:

- gcc –ansi –Wall –pedantic  -std=c99 circularll.c /afs/andrew/course/15/123/downloads/lab3/clltester.c
- ./a.out data.txt

This will make sure you are always testing with the latest tester. Do one last test just before you submit your files.  You must download all files from

**/afs/andrew/course/15/123/downloads/lab3**

## PART II – Josephus Problem (30 Points)

This problem goes back to Josephus Flavius, a 1st century Roman historian. The story is slightly bloody.

> 41 rebels are trapped in a cave, surrounded by enemy troops. They decide to commit suicide: they line up in a circle and systematically kill every other one, going around and around, until only one rebel is left -- who supposedly kills himself (fat chance).
>
> Who is the lone survivor?

Clearly, we can represent the rebels by a list of numbers {1,2,...,41} and we can simulate the demise of the corresponding rebel by manipulating this list. For example, we could just mark the dead rebels by switching item i to -i (a standard hack, and a true abuse of negative numbers). Actually, it will be more interesting to generalize slightly: let us deal with the problem for an arbitrary number n of rebels. So, our original list is {1,2,...,n} , and we have to make a n-1 deletions in the right place to find the survivor.

A good implementation will sometimes differ from a verbatim translation of the original problem. For example, marking the deceased rebels in the list without actually removing them turns out to be a bad idea: we will have to skip over the dead bodies. It is easier to actually remove items from the list. However, it is still quite complicated to do the necessary bookkeeping (we have to skip over the next element, and we have to be careful when we reach the end of the list). If we use an array to simulate this problem, the things can be quite complicated. We need to remove every other element in the array to remove dead bodies. Besides there is a huge cost to filling the holes in the array since we have to shift elements to fill the holes. Here is a simple idea that dissolves all these difficulties, once and for all. Let’s think about a linked list to do this problem. Not just a linked list, but a circular linked list. One step in our simulation will be to

***Rotate the list to the left by one position, and then delete the first element***.

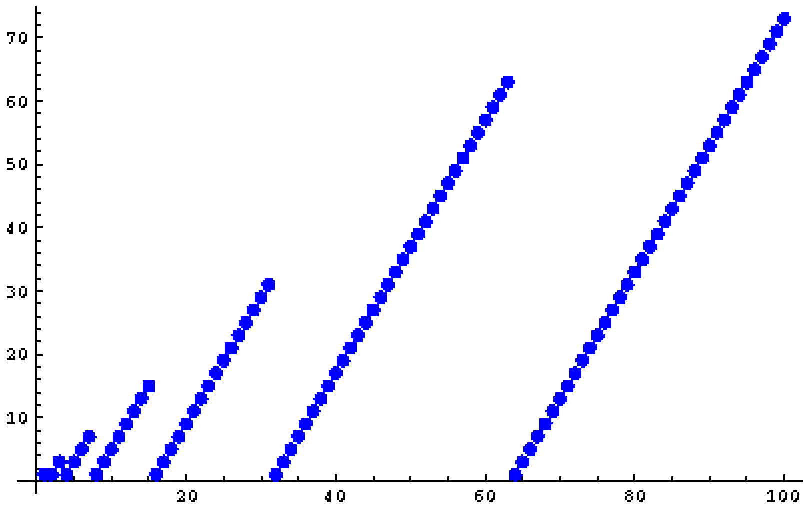
This does implement the every-other-guy protocol of the rebels. Make sure you understand why. Also note that the description of the problem is really a bit ambiguous: we might also delete the first element, and then rotate. As it turns out, the results are slightly easier to describe in the other version.

Here is a run of the simulation for n = 16 rebels. The sequence of survivors is shown after each death, but rotated a bit. In the end, rebel number 1 is the lucky one.

```
 1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16
 3    4    5    6    7    8    9   10   11   12   13   14   15   16   1
```

```
5    6    7    8    9    10   11   12   13   14   15   16   1    3
7    8    9    10   11   12   13   14   15   16   1    3    5
9    10   11   12   13   14   15   16   1    3    5    7
11   12   13   14   15   16   1    3    5    7    9
13   14   15   16   1    3    5    7    9    11
15   16   1    3    5    7    9    11   13
1    3    5    7    9    11   13   15
5    7    9    11   13   15   1
9    11   13   15   1    5
13   15   1    5    9
1    5    9    13
9    13   1
1    9
1
```

And here are the results of the simulation for all rebel numbers up to 100. The *x*-axis is the number of rebels, and the *y*-axis is the lone survivor.



Clearly, there is a lot of regularity here. Increasing n by 1 increases the number of the survivor by 2 for a while, but then a reset occurs and the survivor number goes back to 1. It is tempting to find a nice, concise description for this effect, but we will not pursue the analysis at this time. Sometime in your life you will learn that there is a ridiculously simpler way to calculate the survivor by manipulating the binary expansion of the number of rebels.

## Files

The project is organized into the following files:

- circularll.h // header file for circular LL of ints
- clltester.c // this is the tester for a LL of Integers
- circularll.c // implementation of circular LL functions
- josephus.c // Driver program for Josephus

`CircularLL.h` is a header file that contains functions necessary to solve the Josephus problem (and much more).

One advice of thought is to test your LL library before using it in the josephus program. This way you know that your LL methods are working properly. Once you test this, solving the Josephus problem is easy.

The input/output conventions are as follows. Program can be managed from a simple command line menu.

1. Solve Josephus problem for some n

```
> ./a.out 41

41      19
```

2. Solve Josephus problem for a range of values.

```
> ./a.out 10 20

10      5
11      7
12      9
13      11
14      13
15      15
16      1
17      3
18      5
19      7
20      9
```

3. Solve the Josephus problem for a single value, but display the state of the line after each shooting.  Here the option "-a" indicates that we need to show all steps in the process.

```
> ./a.out -a 10

 1  2  3  4  5  6  7  8  9 10
 3  4  5  6  7  8  9 10  1
```

```
5  6  7  8  9 10  1  3
7  8  9 10  1  3  5
9 10  1  3  5  7
1  3  5  7  9
5  7  9  1
9  1  5
5  9
5
```

You may assume that these are the only 3 cases to check and input is valid from the command line.

.

---

**Downloading Starter Code**

Starter code can be downloaded from

**/afs/andrew.cmu.edu/course/15/123/downloads/lab3/**

Starter code cannot be changed other than to add your own helper functions and complete the requirements of the assignment. Please DO NOT change the function prototypes, as they will be used to standardize the grading of the assignment. We do know that there may be better prototypes based on design of your code.

---

---

**Handing in your Solution.**

All labs are submitted to afs. The folder to submit your program is

**/afs/andrew.cmu.edu/course/15/123/handin/lab3/yourid**

---

**Grading Criteria**

The following general grading criterion is strictly enforced.

1. A program that does not compile  - 0 points
2. A program that is 24 hours late – max of 50% of the grade
3. A program that is 48 hours late – max is 20% of the grade
4. A program that is more than 48 hours late – 0 points

If you are planning to use a "late day" please send email to your TA. Your TA will keep track of late days used. You can only use one late day per assignment.

See Rubric.txt for more details about how individual parts are graded.

Make sure that your program compiles and runs properly. As usual, take the 10 style points seriously; poorly commented spaghetti code is not acceptable.