Programming Assignment 3 - part I
An Application of Linked Linked Lists: Radix Sort
Due: Sunday March 13 @ 11:59pm

Reading: Chapter 3 of Weiss has a section on Radix Sort

Radix sort is an algorithm for sorting non-negative integers. It is based on sorting the elements “digit” by “digit”. Radix sort **in it’s simplest form** (using base-10 digits) can be described as follows.

[IMPORTANT: the description below is for the specific Radix of 10; your program will be more general and usually use a (much) larger radix]

Given: non-negative (unsigned) integers x_0, x_2, …, x_{n-1}

Goal: sort them!

Procedure:

The algorithm uses an array of 10 lists (actually probably two such arrays):

```
buckets[0..9]
```

Each array entry corresponds to a digit (base 10).

The algorithm proceeds in “passes”.

In the first pass, the values to be sorted are assigned to buckets according to their ***least significant digit.*** Thus, all values ending in 0 are in list buckets[0], all values ending in 1 are in buckets[1], and so on.

In the 2nd pass, the values are visited starting from bucket[0], and *in the same order in which they were inserted into the lists -- i.e., the lists act as queues (first-in-first-out).* During the 2nd pass, the values are assigned to buckets (according to their 2nd least significant digit -- the 10’s place).

Implementation detail: this is where the 2nd bucket array comes in -- think of having a “from” bucket array and a “to” bucket array. Their roles can swap from pass to pass.

We perform additional passes with the 100s place, 1000s place and so on until we are done.

When are we done? Here is a safe criteria:

> We are done after the 10^k digit has been processed where 10^k is larger than the largest value being sorted.

Choosing a Radix

The above describes the algorithm where the “Radix” is 10. In general, the radix can be any integer two or larger.

As the radix increases, the number of “passes” gets smaller. On the other hand, the amount of work done per pass (sometimes) increases.

As it turns out, the best balance (and runtime) is achieved when the radix is set to approximately N (the number of elements being sorted).

Your Program

You will write an implementation of radix sort which behaves as follows:

- It reads a sequence of non-negative integers from standard input. The number of values to be read (N) is not given to you ahead of time.
- The program reads non-negative integers until it reaches EOF or an attempt to read an unsigned int fails (e.g., there is a “token” in the input that cannot be parsed as a non-negative integer). In the latter case, the program itself does not “fail”; it simply sorts the values it ***did*** successfully read.

- **Sorts them using <u>Radix Sort with a radix of N</u> (or near N -- more on this below).**
- Displays the data in sorted order.

~~You will submit a single source file rsort.c containing your implementation.~~

You will submit all necessary files to create your rsort executable in a single archive file. Among the files should be a Makefile supporting

```
$ make rsort
```

Example: If your implementation utilizes the list ADT from the first programming assignment (and so your executable links with llist.o), you must, of course, include the related source files (list.h and llist.c).

Some Things to Think About

Extracting a digit: the least significant base-10 digit of x can be determined with the % operator:

```
x % 10
```

What about the other digits? Intuitively, we want to “shift off” the low-order digits and then perform the mod. For the 10s place:

```
(x/10) % 10
```

Then:

```
(x/100) % 10
```

and so-on.

How will you generalize this concept for an arbitrary radix?

Input: remember that you don’t know ahead of time how many elements you will be sorting. How are you going to store them when you get started? Also, how can you use the library function scanf to facilitate reading of input?

List/Queue Operations: All of your list/queue operations must be constant time (each time you add or remove something). You can utilize the list ADT we have already created. This should result in very clean and readable code. Even if you choose to fold your list code into rsort.c, the list operations should still be nicely encapsulated in functions.

Remember though that we should be able to just type "make rsort" to create your executable even if it is built from multiple files.

Memory: As a good C programmer, you are responsible for deallocating (free-ing) all memory you allocate!

Buckets: you should be able to get by with just two "bucket arrays".

An Enhancement

You won't receive extra credit for this (well, maybe extra credit for the soul…), but you might find it interesting. The algorithm sketch above results in a lot of integer divisions and modulus operations. On most machines, these are among the slower (in terms of number of clock cycles) operations.

You can eliminate these slow operations if you select your radix to be a power of two -- e.g., the power of two closest to N.

Think about it and how you can use (fast) bit-level operations to implement division and modulus.

(You have been exposed to the gprof profiling tool -- why not have a contest with your friends to see who's radix-sort is fastest in practice?)

Submission

You will submit just the a zipped tar file including all necessary source files to create your executable -- including a makefile.

Grading

Your program will be scored on correctness and design/style (documentation, functional decomposition, understandability, etc.)