

**CSC 402, Data Structures II, Dr. Dale Parson, Fall 2010, Assignment 4,
External Sorting Using Radix Sort, due by 11:59 PM November 19.**

```
cp ~parson/AdvDataStructures/extradixsort.assn4.zip ~/AdvDataStructures
cd ~/AdvDataStructures
unzip extradixsort.assn4.zip
cd ./ extradixsort
gmake clean build
```

Initially you will get lots of syntax errors because the 4 source files that implement queues are for queues of string objects. You need to convert the following four files to store queues of ints.

interface_queueOfInts.h interface_queueOfInts.cxx file_queueOfInts.h and file_queueOfInts.cxx have the correct names, but internally they implement queues of strings. Change the class names to match their source file names, and change the values that they store and retrieve from “string” to “int”. Be careful, because some string fields such as a file name do not change to ints. The values being stored and retrieved in these files become ints. Please make no other changes to these files.

Once you have accomplished this migration of the above source files, **gmake clean test** should work successfully. At this point the test driver in radixsort.cxx is using mergesort to sort its output file as a sequence of ints. Note that the .out and .ref files store the ints as integer strings, one per line. In a production application we would store our ints as binary data in binary data files in order to reduce storage space and input-output conversion times for string-to-int and int-to-string conversions. In this case we are storing the ints as text in text files as a debugging aid. If you have a crash or bug, you can look at these files using a text editor.

After you have gmake test working without errors, you must replace mergesort inside of radixsort.cxx with a radixsort function that you will write. Check STUDENT comments in radixsort.cxx for details. There is a backup copy in file radixsort.cxx.original in case you want to look at the queue construction for mergesort after you start making changes. I have given this function declaration inside radixsort.cxx:

```
void radixsort(interface_queueOfInts *queueToSort, int numbits,
              interface_queueOfInts * temporaryQueues[]);
```

Here are the comments for the parameters:

```
// STUDENT: Write the code for radixsort to use radixsort
// 1. queueToSort is a queue constructed over the array or sequential file
// to be sorted.
// 2. numbits is a value 1 to 4, where 2^numbits gives the number of
// queues in temporary Queues.
```

```
// 3. temporaryQueues is an array of queues, each of which is an empty queue
// at the time radixsort is called. Radixsort uses these queues as
// temporary storage of ints that have been partitioned across
// 'buckets' that have not yet been merged together. See the assignment
// handout for examples. WHEN YOUR TEST RUN TERMINATES THERE SHOULD BE
// NO tmp... FILES LEFT SITTING IN THE TEST DIRECTORY
```

The illustrations and example below should help with understanding radix sort. Suppose that numbits = 4, giving us an array of 16 initially-empty queues in temporaryQueues. Suppose further that we want to sort these 8 values:

```
0xdead1234 0xdead4321 0xfeed1234 0xfeed4321
0x1234dead 0x1234feed 0x4321feed 0x4321dead
```

Each 4-bit “nibble” selects a bucket (a.k.a. bin) in temporaryQueues. In the first pass the split phase of radixsort would partition the above values in this order, starting with the rightmost, least significant nibble selecting the bucket. The front of each of the following queues is to the left in this illustration.

Partition on nibble 0:

```
temporaryQueues[0]
temporaryQueues[1] 0xdead4321 0xfeed4321
temporaryQueues[2]
temporaryQueues[3]
temporaryQueues[4] 0xdead1234 0xfeed1234
temporaryQueues[5]
temporaryQueues[6]
temporaryQueues[7]
temporaryQueues[8]
temporaryQueues[9]
temporaryQueues[10]
temporaryQueues[11]
temporaryQueues[12]
temporaryQueues[13] 0x1234dead 0x1234feed 0x4321feed 0x4321dead
temporaryQueues[14]
temporaryQueues[15]
```

In the merge phase radix sort reads the queues from 0 through 15 in that order, dequeuing these temporary queues and re-populating the original array in this order:

```
0xdead4321 0xfeed4321 0xdead1234 0xfeed1234
0x1234dead 0x1234feed 0x4321feed 0x4321dead
```

Note that the entries are sorted on the rightmost hex digit. At this point the temporary queues are empty. The following illustrations show the effect of working

up through the nibbles 1 through 7, first partitioning into buckets based on the next nibble (working towards the most significant, leftmost nibble 7), and the merging. The illustrations do not show empty buckets (temporaryQueues).

Partition on nibble 1:

temporaryQueues[2] 0xdead4321 0xfeed4321
temporaryQueues[3] 0xdead1234 0xfeed1234
temporaryQueues[10] 0x1234dead 0x4321dead
temporaryQueues[14] 0x1234feed 0x4321feed

0xdead4321 0xfeed4321 0xdead1234 0xfeed1234
0x1234dead 0x4321dead 0x1234feed 0x4321feed

Partition on nibble 2:

temporaryQueues[2] 0xdead1234 0xfeed1234
temporaryQueues[3] 0xdead4321 0xfeed4321
temporaryQueues[14] 0x1234dead 0x4321dead 0x1234feed 0x4321feed

0xdead1234 0xfeed1234 0xdead4321 0xfeed4321
0x1234dead 0x4321dead 0x1234feed 0x4321feed

Partition on nibble 3:

temporaryQueues[1] 0xdead1234 0xfeed1234
temporaryQueues[4] 0xdead4321 0xfeed4321
temporaryQueues[13] 0x1234dead 0x4321dead
temporaryQueues[15] 0x1234feed 0x4321feed

0xdead1234 0xfeed1234 0xdead4321 0xfeed4321
0x1234dead 0x4321dead 0x1234feed 0x4321feed

Partition on nibble 4:

temporaryQueues[1] 0x4321dead 0x4321feed
temporaryQueues[4] 0x1234dead 0x1234feed
temporaryQueues[13] 0xdead1234 0xfeed1234 0xdead4321 0xfeed4321

0x4321dead 0x4321feed 0x1234dead 0x1234feed
0xdead1234 0xfeed1234 0xdead4321 0xfeed4321

Partition on nibble 5:

temporaryQueues[2] 0x4321dead 0x4321feed
temporaryQueues[3] 0x1234dead 0x1234feed
temporaryQueues[10] 0xdead1234 0xdead4321
temporaryQueues[14] 0xfeed1234 0xfeed4321

0x4321dead 0x4321feed 0x1234dead 0x1234feed
0xdead1234 0xdead4321 0xfeed1234 0xfeed4321

Partition on nibble 6:

temporaryQueues[2] 0x1234dead 0x1234feed
temporaryQueues[3] 0x4321dead 0x4321feed
temporaryQueues[14] 0xdead1234 0xdead4321 0xfeed1234 0xfeed4321

0x1234dead 0x1234feed 0x4321dead 0x4321feed
0xdead1234 0xdead4321 0xfeed1234 0xfeed4321

Partition on nibble 7:

temporaryQueues[1] 0x1234dead 0x1234feed
temporaryQueues[4] 0x4321dead 0x4321feed
temporaryQueues[13] 0xdead1234 0xdead4321
temporaryQueues[15] 0xfeed1234 0xfeed4321

0x1234dead 0x1234feed 0x4321dead 0x4321feed
0xdead1234 0xdead4321 0xfeed1234 0xfeed4321

After exhausting all of the bit positions in the integers to be sorted, radixsort leaves queueToSort with the final, sorted sequence. Each of the temporaryQueues is empty at that point.

Consider the parameters passed to radixsort.

```
void radixsort(interface_queueOfInts *queueToSort, int numbits,  
              interface_queueOfInts * temporaryQueues[]);
```

There are some values you can derive, stored in local variables, that are functions of numbits.

sizeof(int) gives the number of bytes in an int.

8 * sizeof(int) gives the number of bytes in an int.

(1 << numbits) is the same as 2^{numbits} . It gives the number of bins (buckets), i.e., the number of queues in temporaryQueues[].

~(~0 << N) produces N 1's in the least significant bit positions. For example, if N is 5, **~(~0 << 5)** gives the 32 bit number 000000000000000000000000000011111₂.

You can use this latter expression to construct a **bit mask**.

You can use the >> operator to get numbits bits of an integer into the least significant position of a temporary variable.

You can then use & between this shifted number and the bit mask to extract a digit in the radix within which you are sorting (radix=16 for 4 bits, radix=8 for 3 bits, etc.).

You can use that number to figure out into which bin (bucket) the integer goes during the split phase of radix sort.

The merge phase simply drains the temporaryQueues, starting at 0 and ending with the final queue, enqueueing values back into queueToSort in FIFO order.

Your solution to this assignment must use `queueToSort` and `temporaryQueues` to sort the integers. As with our discussion of mergesort in class, radixsort can access values to be sorted strictly using FIFO enqueue, peek, and dequeue operations. For mergesort there must be at least 2 and possibly a few more values in memory at a time; the remainder can reside in files during an external sort. With radixsort, only one value needs to reside in memory at a time! It is the element being binned during the splitphase, or being returned to the original queue during the mergephase. Radixsort does not compare values to each other. Instead, it uses their bit fields as indices for binning.

Unlike mergesort and quicksort, which are $O(N \log(N))$ where N is the growth rate of elements, radixsort is $O(N)$. If you consider the above steps, you will see that doubling N simply doubles the number of times each split and merge step must be performed. Unfortunately, radixsort suffers from very high copy overhead and potentially from high storage overhead, depending on its implementation. It does not pay for this fixed overhead until N grows to a very large value. But, in plotting growth curves, radixsort has the potential of running faster than $O(N \log(N))$ sorts for very large sequences to be sorted. The danger in sorting large sequences in memory, even if they fit, is that the operating system will begin paging portions of large arrays to disk. The danger in sorting large sequences in files using external sorting is that the large fixed overhead of file I/O operations makes external mergesort preferable to external radixsort unless the number of elements to sort at one time is extremely large. There are cases where radixsort is more effective than mergesort.

Your radixsort function must **not** hard code limitations on the number of bits. It can assume that the `numbits` is ≥ 1 , that there is enough memory to house the queues, and that your test driver can open the required number of `file_queueOfInts` data files. In addition to writing the radixsort function itself you must replace the code that sets up data for mergesort with code that sets up data for radixsort, for example by constructing the queues. You can reuse most of that code, and your `.out` file must match my `.ref` file. Also, after a successful run of **gmake clean test**, there should be no "tmp..." files left sitting in your directory. These are created by the constructor for an anonymous `file_queueOfInts` file, and should be cleaned up by its constructor before your program exits. When a test run is clean and there are no diffs, please run **gmake turnitin** as usual.

Below is a simple in-memory, 1 bit radixsort from the `timesort.cxx` handout. While it is limited to 1 bit and uses an array instead of a sequential file for storage, it helps to illustrate the algorithm.

```

static void radixsort(int iarray[], int left, int right) {
    static int bitsinbyte = 8 ;
    static int numbits = sizeof(int) * bitsinbyte ;
    int size = right - left + 1 ;
    int *binarray = new int [ 2 * size ];
    for (int i = 0, mask = 1 ; i < numbits ; i++, mask = mask << 1) {
        int count[2] ;
        count[0] = count[1] = 0 ; // SPLIT PHASE:
        for (int j = left ; j <= right ; j++) {
            int bin = (iarray[j] & mask) >> i ;
            binarray[bin * size + count[bin]] = iarray[j] ; // ENQUEUE into tmp array
            count[bin] = count[bin] + 1 ;
        }
        int restoreix = 0 ; // MERGE PHASE
        for (int bin = 0 ; bin < 2 ; bin++) {
            for (int j = 0 ; j < count[bin] ; j++) {
                iarray[restoreix++] = binarray[bin * size + j]; // ENQUEUE into main array
            }
        }
        delete [] binarray ;
    }
}

```

Here is the code that you must modify to construct appropriate queues for radixsort.

```

bool isvalid = false ;
file_queueOfInts merger(argv[4], isvalid);
if (! isvalid) {
    cerr << "Error: Cannot open file-to-sort " << argv[4] <<
        " as a FIFO." << endl ;
    exit(DATAERR);
} // YOU WILL NEED TO DETERMINE NUMBER OF TEMPORARY QUEUES,
file_queueOfInts q0(isvalid); // CONSTRUCT THEM USING "new" AND
if (! isvalid) { // POPULATE THEM IN A LOOP, BECAUSE THEIR NUMBER
    cerr << "Error: Cannot open first anonymous temporary file as a FIFO."
        << endl ; // IS NOT FIXED!!!
    exit(DATAERR);
}
file_queueOfInts q1(isvalid);
if (! isvalid) {
    cerr << "Error: Cannot open second anonymous temporary file as a FIFO."
        << endl ;
    exit(DATAERR);
}
interface_queueOfInts *splitter[] = {
    &q0, &q1
};
mergesort(&merger, splitter);

```