EE 2331 Data Structures and Algorithms, Semester B, 2009/10

Tutorial 13: Sorting

Week 13 (22nd April, 2010)

The tasks of tutorial exercises are divided into three levels. Level 1 is the basic tasks. You should have enough knowledge to complete them after attending the lecture. Level 2 is the advanced tasks. You should able to tackle them after revision. Level 3 is the challenge tasks which may be out of the syllabus and is optional to answer. I expect you to complete task A and B in the tutorial.

Outcomes of this tutorial

1. Able to apply insertion sort to sort integer data
2. Able to apply radix sort to sort integer data

Sorting is a common task in manipulation of set of data. There are a number of sorting algorithms of which each has its advantages and disadvantages in accordance with the characteristics and the representation of data.

In this tutorial, we will perform sorting on integer data using stacks and queues.

The integer data is stored in a text file with the following format:

Row	Content	Remark
1st	$I_1\ I_2\ I_3 \ldots I_n$	The integer data going to be sorted I_k is the k^{th} element (int. type) of the input sequence

The given program framework will read in the input from the text file and store the input data into an integer queue (***Queue q***).

Task A (Level 2): Insertion sort using stacks

Design the function **void insertionSort(Queue *_q_)** to sort the data by insertion sort with the help of **two stacks**. The function accepts an integer queue (*q*). It will print the content of the data sequence after each pass. And finally the sorted integer will be stored in *q* in ascending order.

A supporting function **top(Stack *_s_)** has been given to you for your easier implementation. The top function will return the topmost element of the stack while leaving the stack remains unchanged. The precondition of this function is the given stack cannot be empty.

Please refer to lecture notes 10 page 143 – 145 for the detailed description of the algorithm.

Expected Output:

```
Enter the file name for testing: test1.txt
Enter your action ( 1) insertion sort, 2) radix sort ): 1

The input sequence is: 8 5 9 6 3
The sequence after 1 pass: 5 8 9 6 3
The sequence after 2 pass: 5 8 9 6 3
The sequence after 3 pass: 5 6 8 9 3
The sequence after 4 pass: 3 5 6 8 9
The sorted sequence is: 3 5 6 8 9
```

```
Enter the file name for testing: test3.txt
Enter your action ( 1) insertion sort, 2) radix sort ): 1

The input sequence is: 101 90 43 27 5 1 66 321 2 16 17 21 40 70 6
The sequence after 1 pass: 90 101 43 27 5 1 66 321 2 16 17 21 40 70 6
The sequence after 2 pass: 43 90 101 27 5 1 66 321 2 16 17 21 40 70 6
The sequence after 3 pass: 27 43 90 101 5 1 66 321 2 16 17 21 40 70 6
The sequence after 4 pass: 5 27 43 90 101 1 66 321 2 16 17 21 40 70 6
The sequence after 5 pass: 1 5 27 43 90 101 66 321 2 16 17 21 40 70 6
The sequence after 6 pass: 1 5 27 43 66 90 101 321 2 16 17 21 40 70 6
The sequence after 7 pass: 1 5 27 43 66 90 101 321 2 16 17 21 40 70 6
The sequence after 8 pass: 1 2 5 27 43 66 90 101 321 16 17 21 40 70 6
The sequence after 9 pass: 1 2 5 16 27 43 66 90 101 321 17 21 40 70 6
The sequence after 10 pass: 1 2 5 16 17 27 43 66 90 101 321 21 40 70 6
The sequence after 11 pass: 1 2 5 16 17 21 27 43 66 90 101 321 40 70 6
The sequence after 12 pass: 1 2 5 16 17 21 27 40 43 66 90 101 321 70 6
The sequence after 13 pass: 1 2 5 16 17 21 27 40 43 66 70 90 101 321 6
The sequence after 14 pass: 1 2 5 6 16 17 21 27 40 43 66 70 90 101 321
The sorted sequence is 1 2 5 6 16 17 21 27 40 43 66 70 90 101 321
```

Task B (Level 2): Radix sort using queues

Design the function **void radixSort(Queue *q)** to sort the data by radix sort with the help of **ten queues**. The function accepts an integer queue (*q*). It will print the content of the queue sequence after each pass. And finally the sorted integer will be stored in *q* in ascending order.

Two supporting functions **power(int base, int exponent)** and **Queue_max(Queue *q)** have been given to you for your easier implementation. The power function will return the value of $base^{exponent}$. To simplify your program, you can assume the integer overflow problem will not be happened. The Queue_max function will return the maximum element in the queue. You can use this information to determine the correct number of passes in your radix sort.

Expected Output:

```
Enter the file name for testing: test2.txt
Enter your action ( 1) insertion sort, 2) radix sort ): 2

The input sequence is: 32 53 41 42 2 92 81 33
The sequence after 1 pass: 41 81 32 42 2 92 53 33
The sequence after 2 pass: 2 32 33 41 42 53 81 92
The sorted sequence is: 2 32 33 41 42 53 81 92
```

```
Enter the file name for testing: test3.txt
Enter your action ( 1) insertion sort, 2) radix sort ): 2

The input sequence is: 101 90 43 27 5 1 66 321 2 16 17 21 40 70 6
The sequence after 1 pass: 90 40 70 101 1 321 21 2 43 5 66 16 6 27 17
The sequence after 2 pass: 101 1 2 5 6 16 17 321 21 27 40 43 66 70 90
The sequence after 3 pass: 1 2 5 6 16 17 21 27 40 43 66 70 90 101 321
The sorted sequence is 1 2 5 6 16 17 21 27 40 43 66 70 90 101 321
```

Discussion: Which sorting algorithm is faster? Why? Please use the last test case which has 3,000 integers to test.