Govind Pillai
gopillai@ucsc.edu
4/22/2021

CSE 13s Spring 2021
Assignment 3: Sorting: Putting your affairs in order
Design Document

PURPOSE:

The purpose of this lab is to learn about the implementation and efficiency of three different sorting algorithms (Bubble sort, Shell sort, and Quick sort). In addition to this, Stacks and Queues are also introduced and used in both Quick sort implementations. A test bench is also implemented that displays the amount moves and comparisons used for each sort in order to properly compare the efficiency of each sorting algorithm.

Pre-lab Questions:

Part 1:

1. Original:

8	22	7	9	31	5	13

First Pass:

8	7	9	22	5	13	31

Second Pass:

7	8	9	5	13	22	31

Third Pass:

7	8	5	9	13	22	31

Fourth Pass:

7	5	8	9	13	22	31

Fifth Pass:

5	7	8	9	13	22	31

Sixth Pass (no swap):

5	7	8	9	13	22	31

Answer: 5 rounds of swapping

2. Worst case for bubble sort is sorting an array that is the complete reverse of the sorted array.

Original:

31	22	13	9	8	7	5

First Pass:

22	13	9	8	7	5	31

Second Pass:

13	9	8	7	5	22	31

Third Pass:

9	8	7	5	13	22	31

Fourth Pass:

8	7	5	9	13	22	31

Fifth Pass:

7	5	8	9	13	22	31

Sixth Pass:

5	7	8	9	13	22	31

Seventh Pass (no swap):

5	7	8	9	13	22	31

Answer: There were 7 elements in the arrays, and 6 comparisons in the first pass. There were 5 comparisons in the second pass and so forth. Therefore, the number of comparisons was 6+5+4+3+2+1 = 21 comparisons. To put this in general terms, the amount of comparisons in the worst case scenario for bubble sort is (n-1) + (n-2)...+ 0.

Part 2:

1. The worst case scenario depends on the gap size for shell sort because it is basically insertion sort. So a very small gap like 1, is really just running insertion sort on the array. Insertion sort will run O(N^2) which is not that efficient. Creating a bigger gap size will increase the efficiency of the sorting algorithm.

Part 3:

1. The worst case scenario for quicksort happens when the pivot is always the smallest or largest element in the array. I believe this is because an extreme pivot would require a larger amount of "split-ups" inside the area. The more divides that happen in the area, more comparisons and moves are required. The reason why quicksort is not "doomed" is because a programmer can make sure these extreme cases do not happen in order to obtain efficiency.

   Citations:
   This video helped me understand quicksort: https://www.youtube.com/watch?v=SLauY6PpjW4

Part 4:

1. In order to keep track of the moves and comparisons, I would create helper functions that would be used in the sorting algorithms. These helper functions would increment steps and moves and could return those values. Then I could include these functions in the test bench and have access to the values.

Stacks:

Stack_empty:

Return true if top is 0 (top is always on the next available space on the stack. If it is at 0, this means there are no elements on the stack)

Stack_full:

Return true if top equals capacity (capacity is the size of the stack which is equal to one plus the last index of the stack. This is the same value top will be holding if the stack is full)

Stack_size:

Return top (because top is at the first available space on the stack, it is equal to one plus the last occupied index on the stack. This is equal to the current size of the stack)

Stack_push:

Check that stack is not full //use stack_full
If it is not full add item to the array (stack)
Increment top

Stack_pop:

Check that stack is not empty //use stack_empty

If it is not empty remove the item from the array (stack)
Set pointer provided equal to the value popped
Decrement top

Stack_print:
Iterate through the stack and print each element

Queues:

Queue_create:
Allocate space and create q object
If q was created:
Set head equal to 0
Set tail equal to 0
Set size equal to 0
Set capacity to capacity provided
Allocate space for items array
If q was not created:
Free q and set it equal to NULL

Return q

Queue_delete:
Free queue pointer
Set queue pointer equal to NULL

Queue_empty:
Return true if the head of the queue is 0

Queue_full:
Return true if the head of the queue equals capacity

Queue_size:

Return head minus tail

Enqueue:

Check that queue is not full //use queue_full
If queue is not full add item to the array (queue)
Increment tail

Dequeue:

Check that queue is not empty //use queue_empty
If queue is not empty add item to the array (queue)
Increment head

Queue_print:

Increment and print each element in the queue