Foundations of Computer Science (Semester 2) – 2015

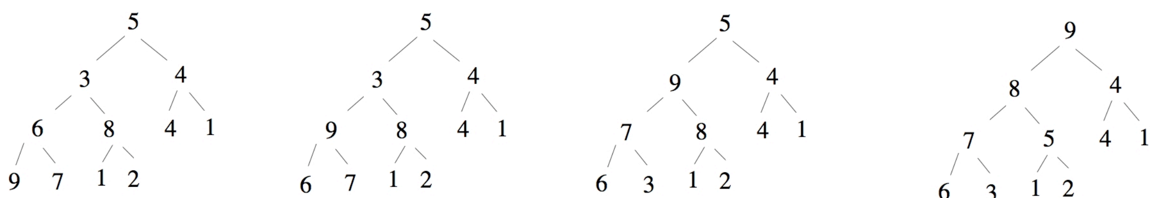
Assessed Exercise Sheet 7 – 10% of Continuous Assessment Mark

Deadline : 11pm Sunday 8th March, via Canvas

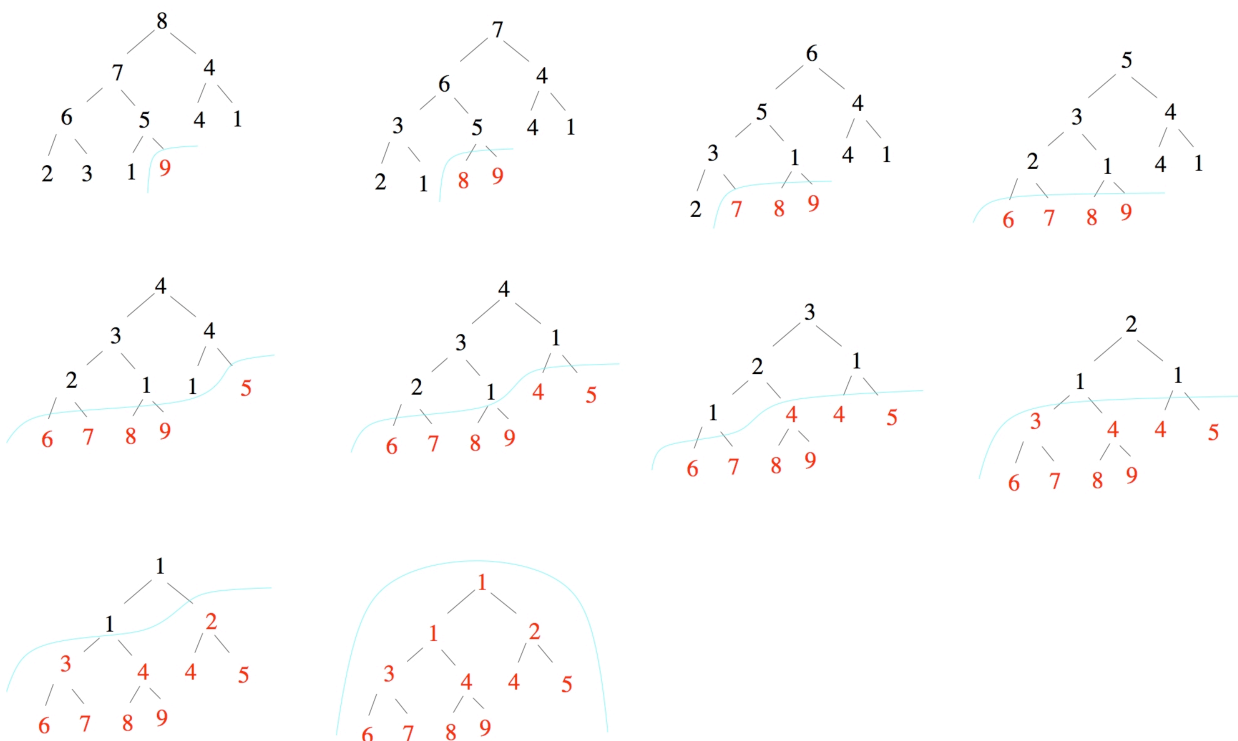
Question 1 (18 marks)

Show how the array [5, 3, 4, 6, 8, 4, 1, 9, 7, 1, 2] would be sorted if the Heapsort algorithm was used. Describe the various processes involved and show the array as a Heap Tree at each stage.

First build a binary heap tree using `heapify`, i.e. load the array items in order into a binary tree (left to right in each level from top to bottom) and bubble down each non-leaf item starting from the bottom:



Then successively swap each top item into the last filled position, with bubbling down of each new top item:



The final heap tree in array form (i.e. read left to right in each level from top to bottom) is the sorted version of the original array.

Question 2 (20 marks)

Outline, in no more than 100 words, the general Quicksort procedure for sorting an array. [Hint: This is another example of the kind of "bookwork" question you can expect in the exam, though you will not normally be given a word limit. The answers are easily found in the lecture notes, but it is worth practicing writing clear and concise answers to questions like this about the key topics in the module, without referring to the lecture notes.]

The general procedure is to repeatedly split/partition each array at each stage in such a way that all the items in the first sub-array are smaller or equal to a chosen "pivot item" and all the items in the second sub-array are greater or equal to that "pivot item", and then concatenate all the final single-item sub-arrays to give the sorted full array.

Sort the array [5, 3, 4, 6, 8, 4, 1, 9, 7, 1, 2] using Quicksort with the pivot chosen to be the middle (rounded down) element of the array at each stage, and a partitioning algorithm that leads to a stable sort. Say how your partitioning algorithm works, and show the state of the array at each stage, i.e. its order and partitioning for each recursive call.

The partitioning algorithm processes each given array in left-to-right order, putting items smaller than the pivot in the left sub-array in left-to-right order, items larger than the pivot in the right sub-array in left-to-right order, with the pivot in between them. Items equal to the pivot are put in the left or right sub-array depending on whether they are to the left or right of the pivot in the original array.

Start	[5, 3, 4, 6, 8, 4, 1, 9, 7, 1, 2]
Partition with 4	[3, 4, 1, 1, 2] 4 [5, 6, 8, 9, 7]
Partition with 1,8	[1] 1 [3, 4, 2] 4 [5, 6, 7] 8 [9]
Partition with 4,6	[1] 1 [2, 3] 4 [] 4 [5] 6 [7] 8 [9]
Partition with 2	[1] 1 [] 2 [3] 4 [] 4 [5] 6 [7] 8 [9]
Finished	[1, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9]

Question 3 (20 marks)

Suppose you have an array of integers `a` with no duplicates. Write an efficient procedure `makeBalBST(a)` that creates a perfectly balanced binary search tree from them. You can use procedures `size(a)` that returns the size of an array `a`; `quicksort(a)` that returns the sorted version of array `a`; `makeBT(v,l,r)` that returns a binary tree with root value `v`, left binary sub-tree `l` and right binary sub-tree `r`; and `aPart(a,i1,i2)` that returns an array made up of elements `i1` to `i2` of array `a`. [Hint: Think carefully about how many times, if any, you should call `quicksort(a)` to make your algorithm most efficient. You will probably find it easiest to use recursion, but not necessarily using repeated calls of your main `makeBalBST(a)` procedure.]

```
makeBalBST(a) {
   return balancedBT(quicksort(a))
}

balancedBT(a) {
   int s = size(a) – 1
   if( s == 0 ) return makeBT(a[0],emptyTree,emptyTree)
```

```
        if( s == 1 ) return makeBT(a[0],emptyTree,
                                    makeBT(a[1],emptyTree,emptyTree))
        return makeBT(a[s/2],balancedBT(aPart(a,0,s/2-1)),
                                    balancedBT(aPart(a,s/2+1,s)))
    }
```

What is the overall average time complexity of your algorithm?

If the size of the array is n, the average-case time complexity of quicksort is $O(n \log n)$, and that of `balancedBT(a)` is certainly no worse than that, so the overall time complexity is $O(n \log n)$.

Question 4 (20 marks)

Outline, in no more than 100 words, the general Mergesort procedure for sorting an array.

Starting from the given array, Mergesort repeatedly splits the array at each stage into its first and last halves, retaining the ordering of and within the sub-arrays. This is repeated until each sub-array contains only one item, which is trivially sorted. The sorted sub-arrays are then merged back together again, in the reverse order of their splitting, in a way that preserves the sorting. That merging is easily done by repeatedly taking the "smallest" item from the front of what is left of the two sub-arrays.

Sort the array [4, 7, 8, 2, 3, 1, 2, 3, 6, 5] using Mergesort. Show the state of the array at each stage, i.e. its order and partitioning for each recursive call.

Start	[4, 7, 8, 2, 3, 1, 2, 3, 6, 5]
Split 1	[4, 7, 8, 2, 3] [1, 2, 3, 6, 5]
Split 2	[4, 7] [8, 2, 3] [1, 2] [3, 6, 5]
Split 3	[4] [7] [8] [2, 3] [1] [2] [3] [6, 5]
Split 4	[4] [7] [8] [2] [3] [1] [2] [3] [6] [5]
Merge 4	[4] [7] [8] [2, 3] [1] [2] [3] [5, 6]
Merge 3	[4, 7] [2, 3, 8] [1, 2] [3, 5, 6]
Merge 2	[2, 3, 4, 7, 8] [1, 2, 3, 5, 6]
Merge 1	[1, 2, 2, 3, 3, 4, 5, 6, 7, 8]

Question 5 (22 marks)

Explain, in no more than 150 words, when and why applying two phase Radix Sort to an array is able to produce a sorted array. State any conditions that must be satisfied for it to work well.

Two phase Radix sort works well when there are two sort keys, each with a strictly restricted set of values. The first phase produces a queue of items for each value of the least significant (secondary) key, and those queues are concatenated in the order of the secondary key. Those items are then put into queues for each value of the most significant (primary) key, preserving their existing (secondary key) order. This leaves each queue sorted according to the secondary key. When those queues are concatenated in the order of the primary key, they are in order of the primary key, with repeated values in the order of the secondary key. Thus the

array is sorted as required.

A library has its books organized primarily according to 20 categories represented by the two digit codes 01, 02, 03, … 20, and secondarily according to the first two letters of the first author's surname Aa, Ab, …, Az, Ba, …, Zz. Use Radix Sort to sort the set of books with keys: [07 Ce, 09 Fa, 17 Mo, 09 Ce, 10 Fa, 09 Mo, 07 Aa, 07 Fa]. Show the state of the book list and what is being done at each stage.

First place the books in order into a queue for each secondary key:

```
Aa : 07 Aa
Ce : 07 Ce, 09 Ce
Fa : 09 Fa, 10 Fa, 07 Fa
Mo : 17 Mo, 09 Mo
```

Then concatenate the queues in the order of the secondary key:

```
07 Aa, 07 Ce, 09 Ce, 09 Fa, 10 Fa, 07 Fa, 17 Mo, 09 Mo
```

Next place the books in that order into a queue for each primary key:

```
07 : 07 Aa, 07 Ce, 07 Fa
09 : 09 Ce, 09 Fa, 09 Mo
10 : 10 Fa
17 : 17 Mo
```

Finally concatenate the queues in order of the primary key to give the sorted list:

```
07 Aa, 07 Ce, 07 Fa, 09 Ce, 09 Fa, 09 Mo, 10 Fa, 17 Mo
```