Data Structures and Algorithms – 2018

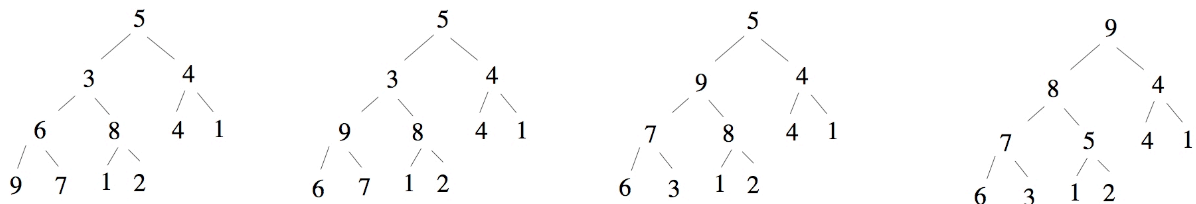
Assignment 4 – 25% of Continuous Assessment Mark

Deadline : 5pm Monday 12th March, via Canvas

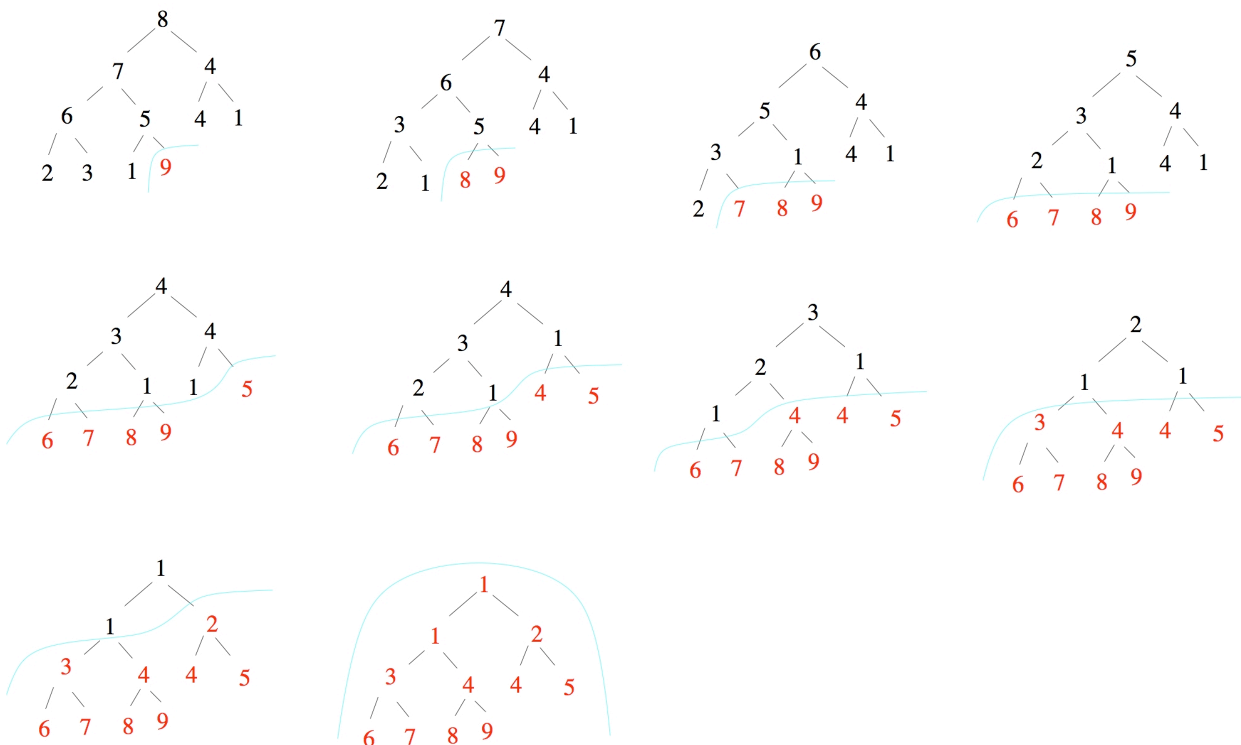
Question 1 (12 marks)

Sort the array [5, 3, 4, 6, 8, 4, 1, 9, 7, 1, 2] using the Heapsort algorithm. Describe what you are doing and show the array as a Binary Tree at each stage.

First convert the given array into a 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 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 (16 marks)

Suppose you have an unbalanced binary search tree `t` containing `n` integers. Write a procedure `rebalanceBST(t,n)` that rebalances it by first outputting its contents to give a sorted array using the `fillArray(t,a,j)` algorithm from the Lecture Notes (Section 9.9), and then creating a perfectly balanced binary search tree by repeated calls of the primitive binary tree procedure `makeBT(v,l,r)` that returns a binary tree with root value `v`, left binary sub-tree `l` and right binary sub-tree `r`. [Hint: Consider writing and using a recursive procedure `balancedBST(a,i,j)` that deals with the sub-array of a sorted array `a` that starts at index `i` and ends at index `j`.]

```
rebalanceBST(t,n) {
   int a[n]
   fillArray(t,a,0)
   return balancedBST(a,0,n-1)
}

balancedBST(a,i,j) {
   if( j == i )
       return makeBT(a[i],emptyTree,emptyTree)
   if( j == i+1 )
       return makeBT(a[i],emptyTree,
                               makeBT(a[j],emptyTree,emptyTree))
   int mid = (i+j)/2
   return makeBT(a[mid],balancedBST(a,i,mid-1),
                                         balancedBST(a,mid+1,j))
}
```

Assuming `fillArray(t,a,j)` and `makeBT(v,l,r)` have both been implemented as efficiently as possible, what are the overall average-case and worst-case time complexities of your algorithm? Explain how you arrived at your answer,

The average-case and worst-case time complexity of an efficient `fillArray(t,a,j)` will be O(n). The total number of calls of `balancedBT(a,i,j)` will always be O(n), and efficiently making the tree at each call will be O(1), so the total complexity of all the calls of `balancedBT(a,i,j)` will always be O(n). Thus the overall average-case and worst-case time complexities are both O(n).

Discuss whether this would be the most efficient approach for putting a whole new array `a` of `n` integers into a perfectly balanced binary search tree. If there is a better approach, explain what it is.

Inserting the whole array into a binary search tree using the standard `insert` procedure would have time complexity O(n log n) in the average case and O(n^2) in the worst case. Following that with the above `rebalanceBST(t,n)` procedure would have O(n) complexity, leaving the overall time complexity of creating a perfectly balanced binary search tree at O(n log n) in the average case and O(n^2) in the worst case. In general it would be more efficient to use a better sorting algorithm to produce the sorted array `a` for passing to `balancedBST(a,0,n-1)`, for example `mergesort(a,0,n-1)` would have the optimal time complexity O(n log n) in both the average and worst cases. Applying appropriate tree rotation operations to keep the tree perfectly balanced while inserting the

whole array into a binary search tree using the standard `insert` procedure would also have time complexity O(n log n) in both the average and worst cases, and be more efficient in practice, but the algorithm for doing that would be much more complicated.

Question 3 (12 marks)

Outline, in no more than 100 words, the general Quicksort procedure for sorting an array. [Hint: This is 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 all the key topics in the module, without referring to any notes.]

The general procedure is to repeatedly/recursively split/partition the array 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 sub-arrays to give the sorted full array.

Sort the array [3, 7, 2, 4, 9, 6, 8, 7, 5, 1, 6] 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 pivots and state of the array at each stage, i.e. the ordering 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. To maintain stability. 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	[3, 7, 2, 4, 9, 6, 8, 7, 5, 1, 6]
Partition with pivot 6	[3, 2, 4, 5, 1] 6 [7, 9, 8, 7, 6]
Partition with pivots 4,8	[3 2 1] 4 [5] 6 [7, 7, 6] 8 [9]
Partition with pivots 2,7	[1] 2 [3] 4 [5] 6 [7 6] 7 [] 8 [9]
Partition with pivot 7	[1] 2 [3] 4 [5] 6 [6] 7 [] 7 [] 8 [9]
Finished	[1, 2, 3, 4, 5, 6, 6, 7, 7, 8, 9]

Question 4 (14 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. Begin by stating 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: [09 Ce, 09 Fa, 16 Mo, 16 Fa, 07 Ce, 13 Fa, 09 Mo, 07 Ba, 13 Ca]. 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:

```
Ba : 07 Ba
Ca : 13 Ca
Ce : 09 Ce, 07 Ce
Fa : 09 Fa, 16 Fa, 13 Fa
Mo : 16 Mo, 09 Mo
```

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

```
07 Ba, 13 Ca, 09 Ce, 07 Ce, 09 Fa, 16 Fa, 13 Fa, 16 Mo, 09 Mo
```

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

```
07 : 07 Ba, 07 Ce
09 : 09 Ce, 09 Fa, 09 Mo
13 : 13 Ca, 13 Fa
16 : 16 Fa, 16 Mo
```

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

```
07 Ba, 07 Ce, 09 Ce, 09 Fa, 09 Mo, 13 Ca, 13 Fa, 16 Fa, 16 Mo
```

Question 5 (18 marks)

Suppose you have a hash table of size 19, the keys are words, and the hash map is defined as follows: Each letter is assigned a number according to its position in the alphabet, i.e.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

and the primary hash function is "x modulo 19", where x is the number corresponding to the first letter of the word. Why is this hash function not ideal?

Taking just one letter as the basis of the hash function means that words with the same first letter will be mapped to the same position, and since the letters in words are far from equally distributed, that will result in many collisions and a clustered table. This will be made worse because several consecutive positions correspond to two letters, and the rest only to one.

Suppose instead you have a hash table of size 13 and the primary hash function is "x modulo 13", where x is the sum of the numbers corresponding to all the letters in the key word. Insert the following list of words into an initially empty hash table using linear probing:
`[computer, science, in, birmingham, dates, back, to, the, sixties]`

0	1	2	3	4	5	6	7	8	9	10	11	12
back			sixties	the	dates	birmingham	to	in			science	computer

What is the load factor of the resulting table, and how many collisions occurred?

The load factor is 9/13 ~ 0.69 and 5 collisions occurred.

What is the effort (i.e. number of comparisons) involved in checking whether each of the following words are in the hash table: `teaching, research, admin`?

Checking for `teaching` takes 6 comparisons, `research` takes 3 comparisons, and `admin` takes 1 comparison.

Show what the resulting hash table would look like if direct chaining had been used rather than linear probing.

0	1	2	3	4	5	6	7	8	9	10	11	12
↓				↓	↓	↓	↓	↓				↓
back				the	dates	birmingham	to	in				computer
							↓					↓
							sixties					science

Now what is the effort (i.e. number of comparisons, not including the processing of NULL pointers) involved in checking whether each of the following words are in the hash table: `teaching, research, admin`?

Checking for `teaching` takes 2 comparisons, `research` takes 1 comparison, and `admin` takes 0 comparisons.

Question 6 (12 marks)

A call centre uses a hash table with open addressing to store customer orders taken during its working hours of 7am to 11pm every day of the year. Orders are kept in the hash table for 5 years and then deleted. It currently has about 5 million entries, and handles about 1 million transactions per year (i.e. between 2 and 3 a minute).

What size hash table would you suggest?

About two or three times the number of entries would be appropriate, i.e. 10 to 15 million.

The performance of the hash table has degraded. Why might this have happened?

Since markers need to be left after an entry is deleted, it can become very costly to search for keys after a number of deletions. In this case, the rate of deletions will be similar to the number of transactions, and there could be as many deletion markers as real entries in the table, and that will have a large negative effect on the performance when searching for entries. In the worst case, the whole table may have to be checked to find if an entry exists. This is particularly likely if the table was set up to be smaller than is optimal now because the number of entries was much smaller then.

Suppose someone suggested copying the entries in the hash table into a new hash table, and it takes 2 milliseconds to copy one item. Would that be a sensible thing to do?

It would take at most 5 or 6 hours to create the whole new hash table (since the old table cannot contain more entries than its size, and it is unlikely to take more time to process a deletion entry than copying a real data entry), so it could easily be completed overnight while the call centre is closed. The current build up of deletions would be removed, and the size of the table could be increased. So, it would probably be a very sensible thing to do.

Question 7 (16 marks)

Given the efficiency of hash tables and their techniques for dealing with hash collisions, one might expect them to provide a more efficient method of detecting duplicate items than the algorithms in Questions 5 and 7 of Assignment 3. Outline in words how a hash table using direct chaining could be used to detect duplicates in a list of items.

Depending on the nature and number the items, a suitable hash table and hash function would be created. Then the items would be inserted one at a time into an initially empty table. A collision might indicate a duplicate item, in which case the chain of items at that table location would be checked to see if it really does contains a duplicate. The insertions would continue until a duplicate is found, and return `true`, otherwise it would continue until all the items had been inserted without finding a duplicate, and return `false`.

Explain the average-case time complexity of this approach, and compare it with those of the three duplicate detection algorithms in Assignment 3. You may assume that the hash table has a reasonably low load factor and a sensible hash function, and you can use any results from the Lecture Notes without repeating the computations. [Hint: As before, consider the extreme cases of duplicates being very rare and duplicates being very common.]

If duplicates are very rare, inserting items and detecting a duplicate will both have O(1) time complexity on average, and if there are n items there will be O(n) insertions, so the overall average-case time complexity is O(n). This is better than all the approaches considered previously.

If there are very many duplicates, only a few will need to be inserted with O(1) complexity before finding a duplicate, with overall time complexity of O(1). This is no worse than any of all approaches considered previously.

So the hash-map approach is best when average-case time complexity is important.

Comment on the worst-case time complexity and how that compares to the algorithms in Assignment 3.

The worst case would correspond to no or very few duplicates and a poor or unlucky choice of hash function that maps all the items to the same table location. In this case there could be O(n) insertions with O(n) sized chains needing checking for duplicates, leading to $O(n^2)$ time complexity overall. This is worse than the sorting approach in Assignment 3, but in practice it would never happen with a reasonably large table and carefully chosen hash function.