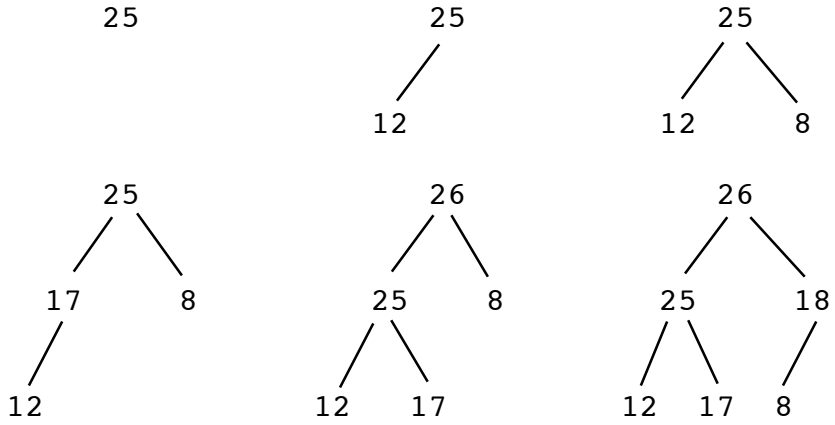
Foundations of Computer Science (Semester 2) – 2015

Assessed Exercise Sheet 5 – 10% of Continuous Assessment Mark

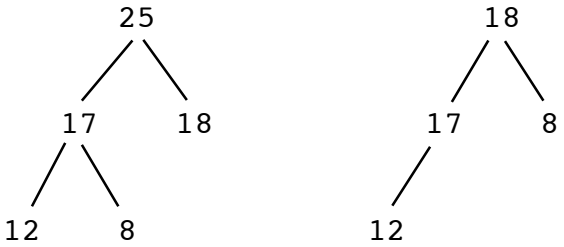
Deadline : 11pm Sunday 22nd February, via Canvas

Question 1 (20 marks)

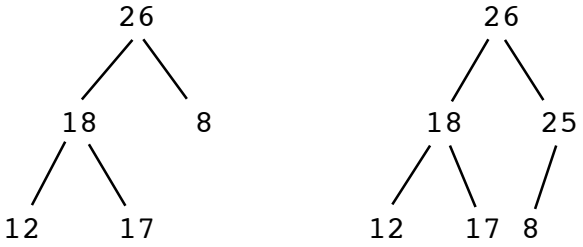
The numbers [25, 12, 8, 17, 26, 18] need to be inserted in that order into an initially empty Binary Heap Tree. Draw the state of the Heap Tree after each number has been inserted.



Now draw the Heap Tree after each of the first two highest priority items have been removed from the resulting Heap Tree.



Finally, draw the Heap Tree after each of the two removed items have been added back to the Heap Tree in the order they were removed.



Question 2 (20 marks)

There are three obvious ways of merging two similarly sized binary heap trees `s` and `t` into a single heap tree:

(i) Move the items one at a time from the smaller heap tree into the larger heap tree using the `insert` algorithm studied in the lecture notes (section 7.4).

(ii) Repeatedly move the last items from one heap tree to the other, with bubbling up, until the new binary tree `makeTree(0,t,s)` is complete. Then move the last item of the new tree to replace the dummy root "0", and bubble down that new root.

(iii) Concatenate the array forms of `s` and `t` and use the `heapify` algorithm from the lecture notes (section 7.6) to convert it into a new heap tree.

Comment on the average-case time complexities of each approach, and thus suggest which approach would be the best in practice to use in general. Are there any special cases for which a different choice might be appropriate?

Since the two heap trees have a similar number of items n, the three approaches will have time complexities as follows:

(i) O(n) items will need to be moved and bubbled at cost O(log$n$), giving an overall time complexity of O($n$log$n$).

(ii) Typically, half the items in the last level of one tree will need moving and bubbling, so that is again O(n) moves at cost O(log$n$), again giving an overall time complexity of O($n$log$n$). However, the actual number of operations will on average be many less than approach (i), by something like a factor of four, so this approach would be better, even though the algorithm is more complex.

(iii) The `heapify` algorithm has time complexity O(n), and the concatenation need be no more than that, so this approach is O(n), making it in the best general approach of all three.

If the two heap trees are such that very few moves are required for approach (ii), then that may look like a better choice of approach than the O(n) approach (iii), but `makeTree` will itself be an O(n) procedure if the tree is in array-based rather than pointer-based form, which it usually is for heap trees.

Question 3 (18 marks)

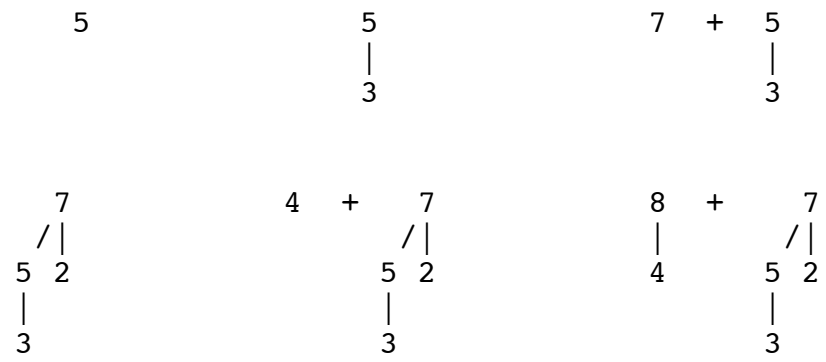
It is stated in the lecture notes (section 7.7) that the structure of a Binomial Heap with n nodes "is unique, and will consist of at most $\log_2 n+1$ trees". Give an intuitive explanation of why that must be true.

By definition, a binomial tree of order k contains exactly 2^k nodes, and a Binomial Heap can only contain zero or one binomial tree of each order. Thus the total number of nodes in a Binomial Heap must be

$$n = \sum_{k=0}^{\infty} b_k 2^k \qquad b_k \in [0,1]$$

where b_k are the number of trees of order k. Thus there is a one-to-one mapping between the Binomial Heap structure and the standard binary representation of n, and since the binary representation is clearly unique, so is the Binomial Heap structure. The maximum number of trees is the maximum number of non-zero binary digits, which is the number of digits when n is written in binary without leading zeros, i.e. $\log_2 n+1$.

Insert the integers [5, 3, 7, 2, 4, 8] in that order into a Binomial Heap. Show the heap after each item is inserted.



Question 4 (20 marks)

Often one needs to determine whether an array contains any duplicate items. Write a procedure `duplicates(a)`, which doesn't involve sorting, that returns `true` if integer array `a` contains duplicates, and `false` otherwise. You may assume that you have access to a procedure `size(a)` that returns the size of an array `a`.

```
duplicates(a) {
   for( int i = 0 ; i <= size(a)-2 ; i++ ) {
      for( int j = i+1 ; j <= size(a)-1 ; j++ ) {
         if( a[i] == a[j] ) return true
      }
   }
   return false
}
```

What is the worst case time complexity of your algorithm?

Let the size of the array be n. The worst case will be to do a full check and return false. In that case, the outer loop must go through n iterations, and on average the inner loop will go through $\sim n/2$ iterations, so the worst case time complexity is $O(n^2)$.

What about the average case time complexity of your algorithm?

The average case time complexity will depend on the nature of the data, i.e. what arrays the average is taken over. If there are usually many duplicates, the algorithm will return true after a few checks, however big the array is, so the average case complexity may be close to O(1). If there are very few duplicates and often none, the average case complexity will be close to the worst case complexity, i.e. $O(n^2)$.

Question 5 (22 marks)

Suppose you have access to a procedure `Xsort(a)` that returns a sorted version of array `a`, that you know has average and worst case time complexity of at least $O(n \log n)$. Write an efficient new procedure `duplicates2(a)` to perform the test for duplicates, that begins by calling `Xsort(a)` to sort the given array. You may again assume that you have access to a

procedure `size(a)` that returns the size of an array `a`.

```
duplicates2(a) {
   a = Xsort(a)
   for( int i = 0 ; i <= size(a)-2 ; i++ ) {
      if( a[i] == a[i+1] ) return true
   }
   return false
}
```

What is the overall worst case time complexity of your algorithm?

Let the size of the array be n. The for loop will go through at most n-1 iterations, so its worst case time complexity is O(n). So the overall worst case time complexity of the algorithm will be equal to that of `Xsort`, because we know that is at least O(n log n).

What about the overall average case time complexity of your algorithm?

The overall average case time complexity of the algorithm will again equal that of `Xsort`, however many duplicates there are in the data.

Thus, comment on when it is worth using `Xsort` in a duplicates testing algorithm.

The worst case time complexity will be improved if the worst case time complexity of `Xsort` is better than O(n^2), for example O(n log n). The advantage for average case time complexity will depend on the nature of the data, i.e. what arrays the average is taken over. If there are usually many duplicates, then using `Xsort` will make the average performance worse. If there are very few duplicates and often none, the average case complexity will be close to the worst case complexity, and thus lead to improvements if the average case time complexity of `Xsort` is better than O(n^2), for example O(n log n).