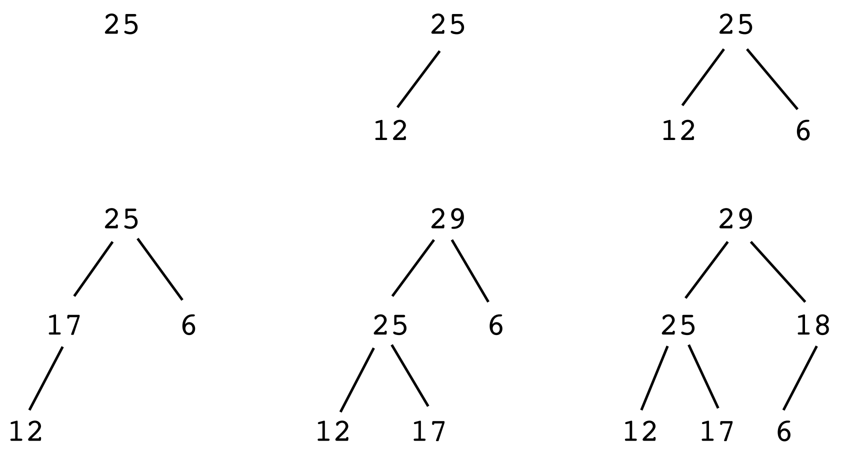
Data Structures and Algorithms – 2018

Assignment 3 – 25% of Continuous Assessment Mark

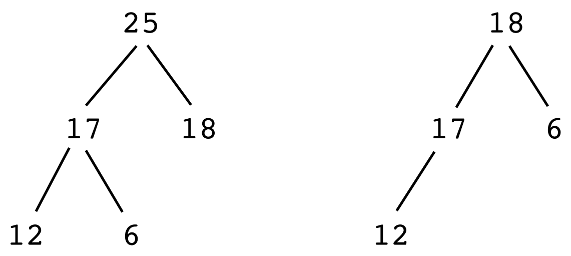
Deadline : 5pm Monday 26th February, via Canvas

Question 1 (12 marks)

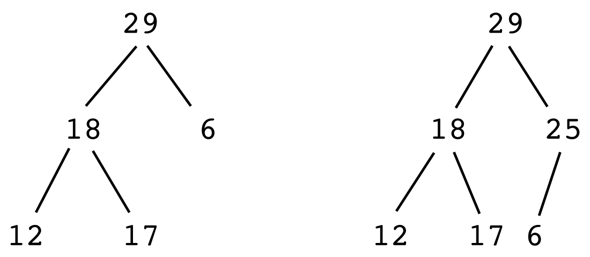
The numbers [25, 12, 6, 17, 29, 18] need to be inserted one at a time, in that order, into an initially empty Binary Heap Tree. Draw the state of the Heap Tree after each number has been inserted.



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.



To what extent does the order of the initial list of items affect the heap tree that is produced and the order in which the highest priority items are removed?

Different initial list orders will produce different heap trees, but the items will always be removed in priority order whatever the order of the initial list. However, if more than one item has the same priority, the initial list order may affect which one gets removed first.

Question 2 (10 marks)

Write an efficient recursion-based procedure `isHeap(t)` that returns `true` if the binary tree `t` is a heap tree, and `false` if it is not. You can call any of the standard primitive binary tree procedures `isEmpty(t)`, `root(t)`, `left(t)` and `right(t)`, and also a procedure `complete(t)` that returns `true` if `t` is complete, and `false` if it is not.

```
isHeap(t) {
   if( not complete(t) )
      return false
   return isHeap2(t)
}

isHeap2(t) {
   if( isEmpty(t) )
      return true
   if( not isEmpty(left(t)) )
      if( root(t) < root(left(t)) )
         return false
   if( not isEmpty(right(t)) )
      if( root(t) < root(right(t)) )
         return false
   return ( isHeap2(left(t)) and isHeap2(right(t)) )
}
```

What can be said about the overall complexity of your algorithm?

Since we are not told the complexity of the procedure `complete(t)`, it is impossible to say what the overall complexity of the algorithm is.

Question 3 (14 marks)

Insert the integers [5, 4, 9, 1, 3, 8] in that order into an initially empty binomial heap. Show the heap after each item is inserted.

```
  5               5              9  +  5
                  |                    |
                  4                    4

  9           3  +   9           8  +    9
 /|                 /|           |      /|
5 1                5 1           3     5 1
|                  |                   |
4                  4                   4
```

Now delete, one at a time, the two highest priority items. Show the binomial heap that is left after each item has been deleted.

The highest priority item is the 9. Deleting that produces two binomial heaps:

```
8     and     5   +   1
|             |
3             4
```

which need to be merged, leaving:

```
 1    +     8
           /|
          5 3
          |
          4
```

The next highest priority item is the 8. Deleting that produces two binomial heaps:

```
1     and     5   +   3
              |
              4
```

which need to be merged, leaving:

```
      5
     /|
    3 4
    |
    1
```

Question 4 (12 marks)

The following segment of C/Java code will sort an array `a` of finite size `n`:

```
for( i = 1 ; i != n ; i++ ) {
    j = i;
    t = a[j];
    while( j > 0 && t < a[j-1] ) a[j] = a[--j];
    a[j] = t;
}
```

Which sorting algorithm discussed in the lectures is this an implementation of?

Insertion Sort.

Work through the sorting of the array [6, 4, 8, 5, 2, 7] using this algorithm, writing down the values of `i, j, t` and the array `a` at the end of each iteration of the `for` loop.

```
i   j   t     a
1   0   4  :  4, 6, 8, 5, 2, 7
2   2   8  :  4, 6, 8, 5, 2, 7
3   1   5  :  4, 5, 6, 8, 2, 7
```

```
4    0    2   :   2, 4, 5, 6, 8, 7
5    4    7   :   2, 4, 5, 6, 7, 8
```

State an appropriate *loop invariant* for this algorithm, and use that to argue why the algorithm is guaranteed to terminate with the array `a` sorted.

An appropriate loop invariant is "array a[0]…a[i-1] is sorted". The for-loop begins with i = 1, and a[0] on its own is trivially sorted, so the process starts with the loop invariant true. Then each of the finite n-1 executions of the for-loop brings the next item into the correct position, using at most n-1 steps of the while-loop, maintaining the loop invariant. The for-loop thus terminates in a finite total number of steps with i = n and the whole array a[0]…a[n-1] sorted.

Question 5 (22 marks)

One often needs to determine whether a given collection of items contains any duplicates. Write two simple procedures, `duplicates1(a,n)` and `duplicates2(a,n)`, not involving any form of hash map, that efficiently return `true` if the array `a` of `n` integers contains duplicates, and `false` if it does not. The first procedure cannot use any form of sorting, while the second must start with a procedure `Xsort(a,n)` which returns a sorted version of array `a` with average and worst case time complexity *O(n* log *n)*. You should assume that all array `a` indices run from `0` to `n-1`.

```
duplicates1(a,n) {
   for( int i = 0 ; i <= n-2 ; i++ ) {
      for( int j = i+1 ; j <= n-1 ; j++ ) {
         if( a[i] == a[j] ) return true
      }
   }
   return false
}

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

Give informal arguments why your algorithms are correct.

`duplicates1:` Each iteration of the i loop (for i = 0 to n-2) checks item i against all later items in the array, so it is guaranteed to terminate and return true if a match exists. Otherwise, the i loop will eventually terminate when all pairs have been checked without finding a match, in which case false will be returned.

`duplicates2:` Xsort will leave any duplicates next to each other in the array, so it is sufficient to check each pair i and i+1 for i = 0 to n-2. The i loop checks all such pairs, so it is guaranteed to terminate and return true if a match exists. Otherwise, the i loop will eventually terminate when all such pairs have been checked without finding a match, in which case false will be returned.

Explain the worst-case and average-case time complexities of each of your two algorithms, and hence comment on which approach is best for each case. [Hint: The average-case time complexities will depend on what you are averaging over. Consider the extreme cases of duplicates being very common and duplicates being very rare.]

Without sorting, there are two nested O(n) loops giving overall worst case time complexity of O(n^2). With sorting, there is just one O(n) for loop, so the overall worst case time complexity is equal to that of `Xsort` which is O(n log n). So, the sorting approach is best in the worst case.

The average case time complexities will depend on the nature of the data, i.e. what arrays the average is taken over. If there are usually many duplicates, the without-sorting 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). With sorting, the time complexity will be O(n log n) in all cases, so it will worse than the without-sorting approach if there are usually many duplicates, but better if there are usually very few or no duplicates.

Question 6 (14 marks)

When Binary Search Trees are used for storing items, it makes sense to require that all the search keys inserted into them are unique. In the Lectures Notes (section 7.4), the standard insertion procedure `insert(v,bst)` is presented that inserts a value, or search key, `v` into such a binary search tree `bst`. Summarize, in the form of a list of possible cases that need to be dealt with, how that procedure works.

There are four basic cases `insert(v,bst)` has to deal with:

1. `bst` is empty: in which case a new tree has to be created containing only `v` as root
2. `v < root(bst)`: in which case call `insert(v,left(bst))`
3. `v > root(bst)` : in which case call `insert(v,right(bst))`
4. `v == root(bst)` : meaning `v` is a duplicate, so terminate with an error message

If the binary search trees are to be used as the basis of a sorting algorithm (Treesort), it will generally be necessary to allow duplicate items. Describe the simplest possible ways the standard `insert(v,bst)`procedure could be modified to accommodate duplicate entries.

To accommodate duplicate entries, case 4 as defined above clearly needs changing. The simplest way to proceed is to delete case 4 by combining it with either case 2 (by changing the < to <=) or case 3 (by changing the > to >=).

Comment on how the possible modifications would affect the stability of a Treesort algorithm based on it.

Merging case 4 with case 2 would lead to an unstable Treesort algorithm, because a later duplicate item would always be added to the left of the existing item in the tree, and thus end up before it in the sorted array. Merging case 4 with case 3 would lead to a stable Treesort algorithm, because a later duplicate item would always be added to the right of the existing item in the tree, and thus remain after it in the sorted array.

One often needs to sort a set of items with any duplicate items removed at the same time. State in words how the standard `insert(v,bst)` procedure for Binary Search Trees could easily be modified to create a Treesort algorithm that returns the items sorted in that way.

This is most easily done by simply replacing case 4 above with:

4. `v == root(bst)` : meaning `v` is a duplicate, so do nothing

Question 7 (16 marks)

Suppose, the standard `insert(v,bst)` procedure for Binary Search Trees was modified to return an `EmptyTree` if `v` already exists in `bst`, rather than terminating with an error message. Write an efficient pseudocode procedure `duplicates3(a,n)` using it that returns `true` if integer array `a` of size `n` contains duplicates, and `false` if it does not.

```
duplicates3(a,n) {
   bst = EmptyTree
   for( int i = 0 ; i <= n-1 ; i++ ) {
      bst = insert(a[i],bst)
      if ( isEmpty(bst) ) return true
   }
   return false
}
```

Explain the worst-case and average-case time complexities of your algorithm, and compare them with those you found for your algorithms in Question 5. State which of the three algorithms is best in which circumstances.

In the worst case, the tree will be unbalanced and all n items will be inserted into the tree with $O(n)$ complexity without finding a duplicate, so the worst case complexity overall is $O(n^2)$. This is worse than using the better $O(n \log n)$ sorting approach of Question 5, and the same as the without-sorting approach.

If there are usually many duplicates, the algorithm will on average terminate after a few insertions into a small tree, leading to average case complexity close to $O(1)$. This is better than the sorting approach of Question 5, because it terminates without completing the whole sort, and is the same as the without-sorting approach.

If there are very few duplicates and often none, in the average case $O(n)$ items will be inserted into the tree with $O(\log n)$ complexity, leading to an overall average case time complexity of $O(n \log n)$. This is the same as the sorting approach of Question 5, and better than the without-sorting approach.

If the worst case is most important, the sorting approach of Question 5 is best. If the average case is most important, the tree-based approach of this question is best.