NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
WRITTEN QUIZ 1 (14%)
AY2017/18 Semester 1

CS2010 – Data Structures and Algorithms II

22 September 2017 Time allowed: 90 minutes

INSTRUCTIONS TO CANDIDATES

1. Do **NOT** open the question paper until you are told to do so.
2. This question paper contains THREE (3) sections with sub-questions. Each section has a different length and different number of sub-questions. It comprises TWELVE (12) printed pages, including this page.
3. Answer all questions in this paper itself. You can use either pen or pencil. Write **legibly**!
4. This is an **Open Book Quiz**. You can check the lecture notes, tutorial files, problem set files, CP3 book, or any other books that you think will be useful. But remember that the more time that you spend flipping through your files implies that you have less time to actually answer the questions.
5. When this Quiz starts, **please immediately write your Matriculation Number** and **Tutorial Group**.
6. The total marks for this paper is **50**.

TUTORIAL GROUP

STUDENT NUMBER: A | | | | | | | | |

For examiners' use only		
Question	***Max***	***Marks***
Q1-4	12	
Q5	8	
Q6	10	
Q7	10	
Q8	10	
Total	**50**	

Section A – Analysis (12 Marks)

Prove (the statement is correct) or disprove (the statement is wrong) the following statements below. If you want to prove it, provide the proof or at least a convincing argument. If you want to disprove it, provide at least one counter example. 3 marks per each statement below (1 mark for saying correct/wrong, 2 marks for explanation):

1. Instead of using a 1-based compact array to store a binary heap, we can use a 0-based compact array if we modify the navigation operations as follows
   - `parent(i) = floor(i/2) if i > 0`
   - `left(i) = (i*2)+1 if i < heap size`
   - `right(i) = (i*2)+2 if i < heap size`

   False.

   In a 0-based compact array, index 0 should be the parent of both index 1 and 2, but par(1) == 0 while par(2) == 1

2. The smallest AVL tree where deleting any vertex will cause 2 re-balancing operations (the 4 cases given in lecture) has 13 vertices.

   False.

   Example in slide 40 of the lecture 04 notes is an AVL tree with 12 vertices that requires 2 re-balancing operations.

   Common Mistake: Student misunderstand the question, and give an example of a 13 vertex AVL tree where deletion does not need any re-balancing to show that this statement is false.

3. To remove an item i from a disjoint set that has ≥ 2 items in a UFDS and make it the only item of a new set, we can simply set P[i] = i.

   False.

   If i is not a leaf in the disjoint set, then the entire subtree it anchors will be removed from the disjoint set and becomes another disjoint set.

4. Assume that ShiftDown(i) will always fix max heap property of the subtree rooted at i iff i is the only node that might violate max heap property in that subtree. Fast heap creation for a max heap of size N will then still work if we modify it as follows

```
For i = floor(logN) down to 1
  For j = 2^i to min(heapsize,2^(i+1)-1)
    ShiftDown(j)
```

True.

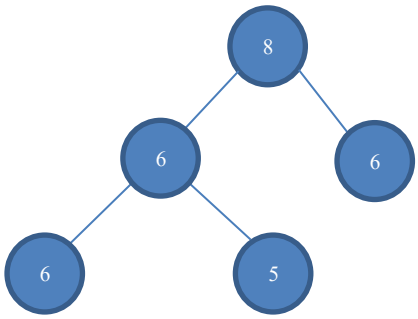
1. i will start from the last level and move upwards to the root. This is the same as usual fast heap creation algo. For each level, j moves from leftmost node 2^i to rightmost node min(heapsize,2^{i+1}-1) the opposite of the usual fast heap creation algo which goes from rightmost to leftmost.
2. Thus the algorithm ensures that all vertices are processed by ShiftDown.
3. If there is only 1 level the algo is obviously true (only 1 item).
4. If there are 2 or more level, the last level and possibly 2nd last level can contain leaves.
    a. Calling ShiftDown on the last level will not anything (all leaves). At the 2nd last level, moving from left to right, we will encounter internal vertices first before possibly some leaves. For j which are internal vertices, left(j) and right(j) are leaves, thus only j is potentially out of place so ShiftDown(j) will work accordingly. For the leaves again nothing needs to be done.
    b. For subsequent levels (if any), for each j, tree rooted at left(j) and right(j) are max heaps (processed in previous level), so ShiftDown(j) will work.
    c. At root level, ShiftDown(1) will heapify the entire max heap.

Section B – Extensions (18 Marks)

Write in pseudo-code. However, any algorithm/functions not taught in CS2010 must be described, there must be no black boxes. Partial marks will be awarded for correct answers not meeting the time complexity required.

5. New max heap operation [8 marks]

For a max heap containing possibly repeated integer keys, write an algorithm for a new operation **int Frequency(i)** which returns the number of items in the heap with the same key as the item at index i. **Frequency(i)** must run in time $O(1)$. Insert and ExtractMax operation must be maintained at $O(logN)$ respectively. For example, given the max heap below



If Frequency(2) is called, then it should return 3.

In your answer, describe any auxiliary data structure you require and also any modification to Insert and ExtractMax (and argue they still maintain O(logN) runtime).

Use a hashmap M where key is item key in the max heap, and value is the count of how many items have that key in the max heap.

1. Insert(k) – if M.get(k) is not null, increment its associated value by 1 and put it back into M, else insert a new entry with value = 1. Then perform insert in the heap as per normal. → O(logN)
2. ExtractMax - decrement value associated with key heap[1] in M by 1, then perform ExtractMax as per normal → O(logN)
3. Frequency(i) – return M.get(heap[i]) → O(1)

6. More BST/bBST operations [10 marks]

i.) Given T the root of a BST containing unique integer values (no repeated values) of size N. Write function **boolean isAVL(*T*)** which will return true if T is also an AVL tree in $O(N)$ time. You cannot assume the height attribute is already computed for each vertex. **[5 marks]**

```
boolean isAVL(T)
{
 if (T != null)
   if (isAVL(T.left) && isAVL(T.right))
     if (-2 < T.left - T.right < 2)
       T.height = max(T.left.height, T.right.height)+1;
       return true
     else
       return false
 else
    T.height = -1 // empty tree
    return true
}
```

This is basically post-order processing of the vertices to correctly calculate height and bf.

ii.) Given roots of two **BST** T and T' of size N and M respectively, containing unique integers, write the algorithm for a merge operation which will merge the two BST into an **AVL tree** (and return the root of the new tree) in O(N+M) time. **[*5 marks]**

1.) In order traversal of T and T' and output the sequence into arrays A and A' respectively. A and A' contains the elements of T and T' in ascending order. → O(N+M)

2.) Use merging function of merge sort to merge A and A' into an array B. → O(N+M)

3.) Now convert B into AVL tree as follows

```
Vertex ConvertToAVL(B,left,right)
{
  if (left <= right)
     Create Vertex V for B[mid]
     V.left = ConvertToAVL(B,left,mid-1) & link left child to V
     V.right = ConvertToAVL(B,mid+1,right) & link right child to V
     return V
   else
     return null
}
```

//This is pre-order creation of AVL tree.

Call T'' = ConvertToAVL(B,0,N+M-1) to perform the conversion

This takes time O(N+M) since each node is processed only once.

4.) After step 3, the tree generated must be balanced. To compute the height information for each node, simply perform post-order traversal in a similar way to the answer for 7i. → O(N+M)

Thus total time taken is O(N+M)

Section C – Application (20 Marks)

Write in pseudo-code. However, any algorithm/functions not taught in CS2010 must be described, there must be no black boxes. Partial marks will be awarded for correct answers not meeting the time complexity required.

7. **A range of values [10 marks]**

Given an array A of N unsorted and non-repeated integers, output the M smaller and the M larger values than the median of the N integers, including the median. They must be output in ascending order. Assume that $M \geq 1$, N is odd and $N \leq 1{,}000{,}000{,}000$,

and $2 \times M$ is much smaller than N ($2M \ll N$). Your algorithm must run in expected time O($N \log M$) or better.

For example, given [132,13,151,15,1,41,6] the median is 15. If M=2, then 6,13,15,41,132 will be output.

Solve the problem in O($N \log M$) time or better using any data structure you have learned in CS1020 & CS2010. **[10 marks]**

Best solution (courtesy of fellow student):

1.) Quick select M = median-m → O(N) time

2.) Quick select M' = median+m →O(N) time

3.) Scan through the input again putting all numbers between M and M' into another list. → O(N)

4.) Sort that list → O(MlogM) time

Total time is O(Max(MlogM,N)).

O(NlogM) solution:

Use two AVL trees T and T'. T to store the M numbers smaller than the median and T' to store the M numbers larger than the median.

1. Use quick select algorithm to find the median (the (N/2)th integer) in expected O(N) time.
2. Go through the A from index i = 0 to N-1
    a. If A[i] < K
        i. If |T| < M: insert A[i] into T ← O(logM)
        ii. If |T| >= M && A[i] > findMin(T): delete min and insert A[i] ←O(logM) since size of T is maintained at |M|
    b. If A[i] > K
        i. If |T'| < M: insert A[i] into T ← O(logM)
        ii. If |T'| >= M && A[i] < findMax(T): delete max and insert A[i] ←O(logM) since size of T' is maintained at |M|

    Total time taken is O(NlogM)
3. perform inorder traversal on T and T' and output the values including K. ← O(M)

Total time taken is expected O(N+NlogM+M) = O(NlogM) since M<<N.

Another Solution:

Use a min heap H and a max heap H'. H' will store the M values larger than the median.

H will store the M values smaller than the median.

1. Use quick select algorithm on A to find the (N/2)th smallest element (the median). Let K store the median. → expected O(N) time
2. Now go through A from i = 0 to N-1
    a. If A[i] < K
        i. if |H| < M: Insert A[i] into H →O(logM)
        ii. if |H| >= M & if min value in H < A[i]: extractMin(H) and insert A[i] into H → O(logM) since |H| is maintained at M.
    b. If A[i] > K
        i. if |H'| < M: Insert A[i] into H' → O(lgM)
        ii. if |H'| >= M & if max value in H' > A[i]: extractMax(H') and insert A[i] into H' → O(logM) since |H'| is maintained at M.

    Total time taken is O(NlogM)
3. perform heap sort on H and H' and output the values including K. ← O(MlogM)

Total time taken is O(N+NlogM+MlogM) = O(NlogM) since M<<N.

8. **Rebuild the code! [10 marks]**

Jane the manager of a bank has the code for the safe on a piece of paper that is locked in her office drawer. One day, she discovers her office ransacked and the paper missing! However, Jane does not panic because she realizes it is not easy for the thief to use the code to open the bank safe.

This is because the bank safe code is very special. It consists of N non-repeated integers ($2 \leq N \leq 1{,}000{,}000$) and have values in the range 0 to 1000,000,000. This is not the special thing about the code. The special thing is that they are actually arranged in a BST and this BST is the one that is to be entered into the security system in order to open the safe (Imagine there is some way to do it …).

What was written on the piece of paper is simply a sequence of the N integers obtained by either the **in-order**, **pre-order** or **post-order** traversal of the BST.

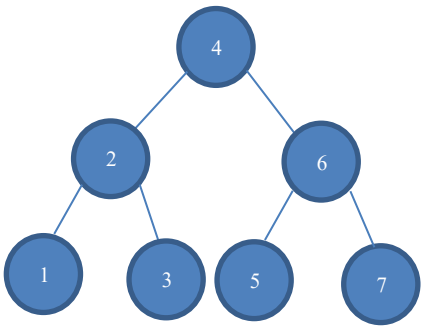
But wait! Jane remembers that she has put an asterisk beside the integer which is the root of the BST. This could provide vital information for the thief to rebuild the BST!

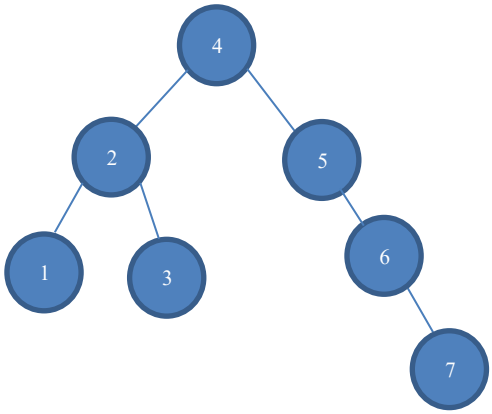
You are the thief who stole the piece of paper, and you have entered the sequence of N integers into an array A. Answer the following questions in order to reconstruct the BST.

i.) You realize that in general it is not possible for the sequence to be generated using in-order traversal. Argue why this is so (Do this like an analysis question). **[3 marks]**

An inorder sequence can be generated from multiple BSTs even if the root is known. Thus it is not possible to re-construct the BST from the sequence in general.

For example the inorder 1234567 where 4 is the root can be from the following BSTs (there are more possibilities).





Other arguments:

1. **1 mark argument**: Some students argue that it is easy to spot inorder sequence as a reason. However it is just as easy if the root index is given to spot post and pre-order sequence.

ii.) Now given the sequence A and the index of the root r in the sequence, how do you tell whether it is a pre-order or post-order sequence? **[2 marks]**

if r == 0 then it is preorder since the root is always output first
else it must be postorder

iii.) Now assuming that the sequence is generated using post-order traversal, write the algorithm for $\boldsymbol{T}$ **RebuildTreePost(**$\boldsymbol{A}$**)** which will rebuild the BST T in $O(N)$ time given A the array containing the sequence and return T the root of the rebuilt tree. You can include any additional helper functions. **[*5 marks]**

T RebuildTreePost(A)

0. Let cur = |A|-1, cur is a global variable
1. V = create vertex for A[cur]
2. Let max = 1000,000,000, min = 0, r = A[cur]
3. cur -= 1
4. V.right = recursiveBuild(A,r+1,max) & link right child to V
5. V.left = recursiveBuild(A,min,r-1) & link left child to V
6. return V

T recursiveBuild(A,i,min,max) // [min,max] is the allowable range for A[cur]

1. if (cur < 0) return null // processed all values in the sequence
2. r = A[cur]
3. if (min <= r <= max) // create the vertex only if A[cur] is within range
    a. V = create vertex for A[cur]
    b. cur -= 1

c. V.right = recursiveBuild(A,r+1,max) & link right child to V
d. V.left = recursiveBuild(A,min,r-1) & link left child to V

else
return null

call T = RebuildTreePost(A) to rebuild the BST.

The algorithm guarantees that all the required vertices are created, since each item in A is considered (cur goes from |A|-1 to 0), thus there will be O(N) calls to recursiveBuild to create all he vertices (minus the root).
For each created vertex, recursiveBuild is called on both its left and right child. In the worst case, both left and right child are null, thus O(2N) calls are made.
In total recursiveBuild is called O(3N) = O(N) time.
Each call to recursiveBuild takes O(1) time (constant number of statements executed).
Thus total time is O(N).

== END OF PAPER ==