ESO 207A: Mid Semester Exam

Full time for this examination is 120 mts
Mention your name and roll number in each page of the paper in the space provided.
Answer the questions in the spaces provided on the question sheets.

Name: ______________________________

Roll No: ______________________________

(Feb 21, 2018)

1. (7 points) Propose implementation of a data structure D which supports stack operations `push` and `pop` and a third operation `findMIN` in O(1) time. Give algorithms (pusedo code) for each operation.

Solution: It can be implemented using two stacks. We will call these S and M. S supports usual stack operations of `push` and `pop` whereas the second stack M keeps track of current minimum.

```
S.push(x) {
    if (!isFull(S)) {
        top++;
        S.val[top] = x;
        if (x < M.top())
            M.push(x);
    } else
        print "stack overflow";
    return;
}

S.pop() {
    x = Nil; // a sentinel representing no element.
    if (!isEmpty(S)) {
        x = S.val[top--];
        if (x = M.top())
            M.pop();
    }
    return x;
}

S.findMIN() {
    return M.top();
}
```

Grading Policy:

7 marks (full) if all three operations in O(1) time.

4 marks if one of Push/Pop take O(n) time.

2. (4 points) Define the min-heap property. Can the min-heap property be used to print out the keys of an n-node heap in a sorted order in O(n) time? Explain how, or why not?

Solution: In a min-heap the key in a node is ≤ keys in both of its children. The keys in heap are not in sorted order, because it does not tell which all keys need to be printed before the key in the current node or which all keys need to be printed after.

A heap can be built by O(n) time as we have proved in the class. So, if we could have an O(n) time algorithm for printing elements in sorted order from heap, then we could have sorting O(n) time sorting algorithm. But we know lower bound of sorting is O($n \log n$). So, it is impossible to have an algorithm for printing keys in sorted order from a heap.

Grading Policy:

1 mark for defining min-heap property.

1 mark for quoting lower bound of sorting.

2 marks for correct solution/justification.

3. (4 points) Recall that in quiz 2, there was a problem concerning a d-min-heap. This is a generalization of binary min-heap, it consists of d children instead of 2 children per internal node. How would you represent a d-min-heap in an array? Your answer should also explain how d-min-heap is equivalent to a binary mean-heap if $d = 2$.

The scheme for representing a d-heap in an array will be as follows:

The root is at $A[1]$. The d-children of the root will be filling up positions from 2 through $d + 1$: $A[2], A[3], \ldots, A[d + 1]$. Next from children of $A[2]$ will be in positions $A[d + 2]$ through $A[2d + 1]$ and so on. So, the grand children of the roots will be placed between $A[d + 2]$ through $A[d^2 + d + 1]$ and so on.

For extracting parent of node at position $A[i]$ we access $A[(i - 2)/d + 1]$. For extrcating $1 \leq j \leq d$th child of node $A[i]$ we access $A[d * (i - 1) + j + 1]$.

Grading Policy: Correct optimal scheme [0 or 1] along with a solution for $d = 2$ case gets full mark. -1 for the omission of $d = 2$ case. Sub-optimal but correct scheme will receive 2 marks. No mark for incorrect solution.

4. (6 points) Is it possible to recreate a BST only from its inorder traversal list?
   (a) (3 points) If yes, given the inorder list: 12, 30, 35, 45, 55, 70, 81, uniquely recreate the BST. If not, why not, explain your answer through a smallest nontrivial example consisting of at least 4 elements.
   (b) (3 points) Apply minimum modifications to usual recursive inorder traversal method to obtain the reverse sorted listing of the keys of a BST.

Solution: (a) It is not possible to uniquely recreate a BST from its inorder list. Consider the inorder list of a BST having 4 elements. Let the list be: a_1, a_2, a_3, a_4. Since inorder list is a sorted list, we have $a_1 \leq a_2 \leq a_3 \leq a_4$. If for creating BST, the order of insertion is in sorted order a right skewed BST will result. However if the order of insertion is: a_2, a_1, a_4, a_3 then a different tree is obtained.

```
a_1                 a_2
 \                 /   \
 a-2             a_1     a_4
   \                     /
   a_3                 a_3
     \
     a_4
```

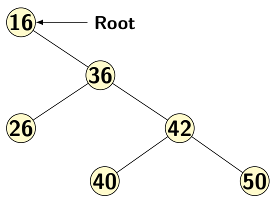
Grading Policy: If example is of four nodes, then 3 marks. If five or more then 2 marks.

(b) This algorithm is simple, it just hav to change the order of visit to: right subtree, root and left subtree. So the modified algorithm is as follows:

```
define a new array A of appropriate size;
define pos=0; // Current array position
if tree is not empty {
    1. Copy all keys of right subtree of the root in consecutive positions
       of the array using the algorithm recursively;
    2. Copy the root;
    3. Copy all keys of left subtree of the root in consecutive positions
       of the array using the algorithm recursively;
}
```

Grading Policy: Only correct answer gets full mark, 0 otherwise.

5. (10 points) This is linked question on Binary Search Trees (BST). The every part of the question is dependent of the answer of the preceding part. Consider the following BST:



(a) (4 points) What are the possible order of insertions of the keys that could produce the BST tree given in the figure? Explain your answer.

(b) (3 points) Draw the BST after insertion of keys: 6, 4, 5, 13, 1 in that order into the BST shown above.

(c) (2 points) Delete 6, 36 from the resulting BST of part (b).

(d) (1 point) Now specify the minimum number of deletions needed (from the BST resulting from deletions as in part (c)) so the resulting BST is is perfectly balanced.

Solution: (a) The relative order of insertion of 16, 36 and 42 would remain same irrespective of the order in which 26 is inserted. Furthermore, 40 and 50 can be inserted only after 42 has been inserted. 26 cannot be inserted before 16 and 36 due structural property of the given BST. However, there four possible ways in which 26 can be inserted: (1) after 36, (2) after 42 , (3) after 40, and (4) after 50.

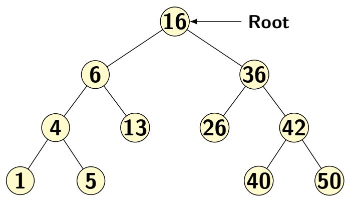
Each of the first three possibility of insertion generate 2 distinct permutations while the last possibility give one permutation:

Insertion	Order of Insertions
(1)	16, 36, 26, 42, 40, 50
	16, 36, 26, 42, 50, 40
(2)	16, 36, 42, 26, 40, 50
	16, 36, 42, 26, 50, 40
(3)	16, 36, 42, 40, 26, 50
	16, 36, 42, 50, 40, 26
(4)	16, 36, 42, 40, 50, 26

Hence, there are 7 possible order of insertions which generate the given BST.

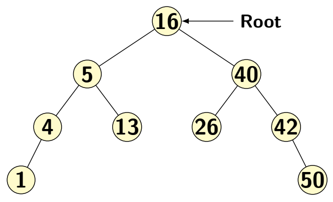
Grading Policy: 4 marks if all insertion orders are written. Partial marks if only some were mentioned.

(b) The tree after insertions is as follows:



Grading Policy: Only correct answers get full mark, 0 otherwise.

(c) The tree resulting after deletion of 6, 26 is:

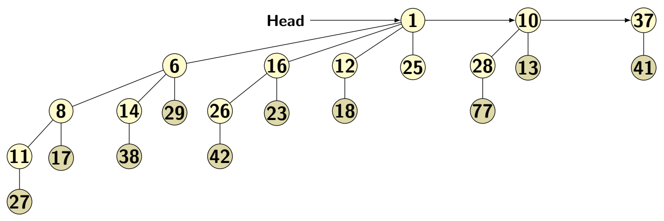


Grading Policy: Only correct answers get full mark. Answers may either consistently use successor or consistently use predecessor. No mark for mixing both.

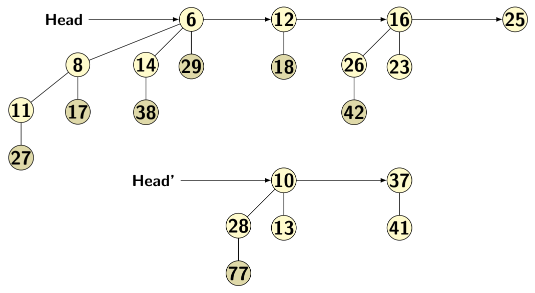
(d) Just two deletions: 1, 50 will produce a perfectly balanced BST.

Grading Policy: Only correct answers get full mark, 0 otherwise.

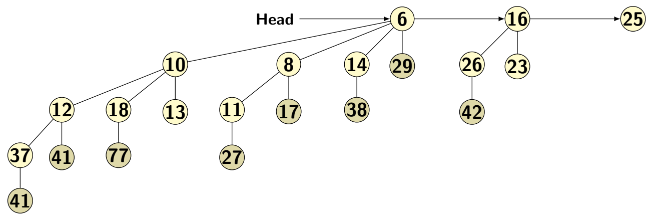
6. (6 points) Given the following binomial heap perform one deleteMIN operation and determine the resulting heap.



Solution: Heaps H and H' generated after deletion of 1.



After merging of binomial trees in H and H'.



Grading Policy: Deletion part without merging - 1 mark (first part in solution)

There are three mergings,

if 1 merging then 2 marks

if 2 mergings then 3 marks

if all mergings then full mark

7. (6 points) Given a binary tree T with preorder, postorder and inorder ranks of its nodes, create a simple set of inequalities that determine if a node x is an ancestor or descendant of another node y. Give a proof of correctness of your answer.

Solution: Let $pre(n)$ and $post(n)$ respectively denote the preorder and the postorder ranks of a node n in T. Notice that a node x and y are related by ancestor-descendant relation if and only if there is a tree path from x to y. A tree path can exist between x and y provided:

$$pre(x) \leq pre(y) \text{ and } post(x) \geq post(y)$$

The existence of tree path is proved as a follows when above inequalities hold.

If $pre(x) \leq pre(y$, it implies x was visited earlier than y in pre order traversal. However, if $post(x) \leq post(y)$ also holds then it implies that y must have been visited during exploration of the subtree of x.

Grading Policy: Correct inequalities with formal/informal proof gets full marks. No marks for writing incorrect/incomplete inequalities. -2 for if no explanation given but inequalities are correct.

8. (6 points) Find the connection between Fibonacci number and the maximum height of an AVL tree. Also find the minimum number of nodes in an AVL tree of height 15.

Solutions: To answer the question determine an expression for the minimum number of nodes in an AVL tree of a given height h. Let T_h be an AVL tree of height h having minimum number of nodes. $|T_0| = 1$, $|T_1| = 2$, $|T_2| = 4$. In general,

$$|T_h| = |T_{h-1}| + |T_{h-2}| + 1$$

Compare with Fibonacci numbers: $f_n = f_{n-1} + f_{n-2}$, $f_0 = f_1 = 1$

h	1	2	3	4	5	6	7	8
f_h	1	2	3	5	8	13	21	34
$\|T_h\|$	2	4	7	12	20	33	54	88

Add 1 to both sides above equation:

$$|T_h| + 1 = (|T_{h-1}| + 1) + (|T_{h-2}| + 1)$$

Thus $|T_h| + 1$ essentially represents a Fibonacci number f_{h+2}.

So, AVL tree of height 15 with minimum number of node will have $f_{17} - 1 = 2583$ number of nodes.

Grading Policy: If only relationship mentioned (without explanation) correctly - 1 mark,

Relationship with explanation - 3 marks.

For AVL tree expression or value is correct full marks - 3 marks

9. (6 points) Suppose we implement the following algorithm for inorder traversal of a subtree of a height balanced BST, assuming $preOrder(x) + k \leq n$

```
1.  x = find(T,x);
2.  print x;
3.  repeat the following for k times {
        x = inorderSuccessor(T, x)
        print x;
    }
```

What will be time complexity of the above algorithm?

Solution: After find(T, x) s executed in time of O(h), i.e., O($\log n$) as T is height balanced. Now consider the execution of finding of inorder successor. Since, it is making call to inorder successor of the previous node, the execution will never go to left subtree. So, each edge of the tree will be traversed at most twice. Hence, for $k-1$ calls to successor the time will O(k). Therefore, the the inorder traversals is implemented in O($k + \log n$) time. Since, k can be of the order of n the overall running time is O(n).

Grading Policy: Full marks to O($k + \log n$). 0 marks to O($k \log n$).

10. (4 points) Prove that in a red-black tree the longest simple path from a node x to a descendant leaf has a length at most twice of the shortest simple path from x to descendant leaf.

Solution: From properties of a red black tree we know that every simple path from a node x consist of same number of black nodes. Let P_L and P_S respectively denote the longest and the shortest paths to a leaf descendant from x. Let there be k black nodes on each of the two paths. We consider two cases separately: (i) x is red, (ii) x is black.

Case (i): The leaf nodes are black. So, there can be exactly k red nodes on P_L. This means P_L consists of $2k$ nodes. So, $|P_L| = 2k - 1$. On the other hand, P_S consists of just $k + 1$ nodes including the leaf node. Therefore, $|P_L| = k$.

Case (ii): x is black, so $|P_S| = k - 1$. Now let us consider P_L. Since it begins with a black node and ends at a black nodes, there can be 1 more red node than $k - 2$ other black nodes on the path. Hence total number of nodes on P_L is equal to $k + k - 1 = 2k - 1$ nodes. This implies $|P_L| = 2(k - 1) = 2|P_L|$

Grading Policy: P_L description ok, P_S description ok - 2 marks

No marks for just mention red black tree properties.

Correct proof - full marks

11. (5 points) This question is for refreshing your memory on sorting algorithms taught in ESC101A and also in this course.
    (a) (3 points) Given an efficient algorithm for even-odd partitioning of an array of integers. For example, if input is $A = \langle 7, 17, 74, 21, 7, 9, 26, 10 \rangle$ after partitioning might be $A = \langle 74, 10, 26, 17, 7, 21, 9, 7 \rangle$. Your partitioning method must be an in-place algorithm, i.e., you may use only constant memory space in addition to arry A for storing the input. It also means that you cannot make use of another temporary array. What is the running time of your algorithm?
    (b) (2 points) What is the running time of Quick sort algorithm when all elements are same in an array of size n.

Solution: (a) The algorithm is like partitioning method of quick sort.

```
set left=0; right=n-1;

keep moving from left to right incrementing left pointer
until hitting an odd number.

keep moving from right to left decrementing right pointer
until hitting and even number.

if (left <= right) then swap(A[left], A[right])
```

Grading Policy: Correct algorithm 2 marks, Time complexity: 1 mark.

(b) It is O(n^2) as one of the partition is always empty.

Grading Policy: Zero for incorrect answer, and full marks for correct answer.

12. (8 points) To generate a random permutation of first n natural numbers we may use following two algorithms:

```
algorithm-1() {
    // Assume existence of a function rand(i, j) which generates
    // a random number between integers i to j with equal probability
    // and takes 1 unit of time.
    // Fill the array A[1..N]  as follows:
    i = 1;
    while (i < n+1) {
        r = rand(1, n);
        if (none of the entries A[1..i-1] equals r) {
           A[i++] = r;
        }
    }
}

algorithm-2() {
    // Keep an array Used[] such that Use[r] = 1 if r has been used.
    // Other assumptions are as in algorithm-1
    i = 1;
    while (i < n+1) {
        r = rand(1, n);
        if (Used[r] == 0) {
            A[i++] = r;
        }
    }
}
```

(a) (2 points) Prove that the two algorithms generate only legal permutations and all the permuations are equally likely.
(b) (4 points) Give a big Oh analysis of the *expected* running time of each algorithm.

(c) (2 points) What is the worst case running time of each algorithm?

Solution: (a) Both algorithms have test for duplicates. So no number will ever be repeated in filling positions of the array $A[1..n]$.

Grading Policy: Zero for incorrect answer, and full marks for correct answer. (b) The first algorithm requires O(i) time to check for duplicates. The expected number probes needed for duplicate check is $n/(n-i)$. The time bound is thus:

$$\begin{aligned}\sum_{0}^{n-1} \frac{ni}{n-i} &< \sum_{0}^{n-1} \frac{n^2}{n-i} \\ &< n^2 \sum_{0}^{n-1} \frac{1}{n-i} \\ &< n^2 \sum_{1}^{n} \frac{1}{i} = O(n^2 \log n)\end{aligned}$$

In the second algorithm Use array allows us to save comparison by factor of i for each of the generated numbers. Thus time will be reduced to bound O($n \log n$).

Grading Policy: Probes for duplicate check correctly specified for both algorithms - 1 mark

Formulation of expression is correct - 2 marks

(c) Worst case running time cannot be bounded, because there is a finite probability that algorithm will not terminate by finite time T.

Grading Policy: Zero for incorrect answer, and full marks for correct answer.