CSc 130 Midterm Exam

Spring 2021

(Questions 1-18, 5 points each; Question 19, 30 points)

1. If f(n) = Ω(g(n), then
   (a) f(n) is less complex than g(n)
   (b) f(n) grows at least as fast as g(n)
   (c) The complexity of f(n) is no greater than that of g(n)
   (d) The complexity of f(n) is greater than that of g(n)

2. If f(n) = O(g(n), then
   (a) f(n) is more complex than g(n)
   (b) the growth rate of f(n) is no greater than that of g(n)
   (c) f(n) is the upper bound of g(n) in terms of complexity
   (d) g(n) is the lower bound of f(n) in terms of complexity

3. According to the big O notation, which of the following is false?
   (a) $n^{1/2} = O(n/ \log n)$
   (b) $(\log n)^{1000} = O(n^{1/1000})$
   (c) $(\log n)^{n} = O(2^{n})$
   (d) $2^{1,000,000} = O((\log n)^{1/1000})$

4. What is the maximum possible height of a binary search tree with 19 nodes?
   (a) 17
   (b) 18
   (c) 19
   (d) 20

5. Which statement is always true for a binary search tree of size n?
   (a) The value of the parent node is greater than that of its children
   (b) The height of the tree is n/2
   (c) For any node in the tree, the value of its left child is less than that of its right child
   (d) None of above

6. What is the minimum possible height of a binary search tree with 45 nodes?
   (a) 3
   (b) 4
   (c) 5
   (d) 6

7. The average case time complexity of inserting a node into a red-black tree of size n is
   (a) O(1)
   (b) O(log n)
   (c) O(n)
   (d) O(n log n)

8. The worst case time complexity of inserting a node into a red-black tree of size n is
   (a) O(1)
   (b) O(log n)
   (c) O(n)
   (d) O(n log n)

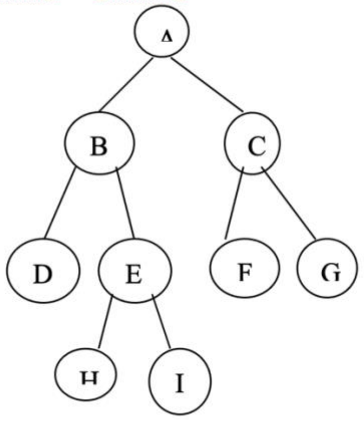
9. The best case time complexity of inserting a node into a red-black tree of size n is
   [✓ (a) O(1)]
   (b) O(log n)
   (c) O(n)
   (d) O(n log n)

10. The time complexity of performing a double rotation in an AVL tree is
   [✓ (a) O(1)]
   (b) O(log n)
   (c) O(n)
   (d) O(n^2)

11. The time complexity of deleting a node from an AVL tree of size n is
   (a) O(1)
   [✓ (b) O(log n)]
   (c) O(n)
   (d) O(n^2)

12. The time complexity of performing a *preorder traversal* against a binary tree of size n is
   (a) O(1)
   (b) O(log n)
   [✓ (c) O(n)]
   (d) O(n^2)

Perform preorder, inorder and postorder traversal for the following tree (



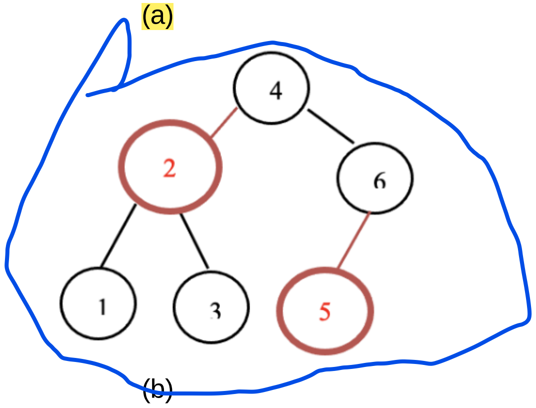
13. The result of postorder traversal is
   (a) A, B, C, D, E, F, G, H, I
   (b) A, B, D, E, H, I, C, F, G
   (c) D, B, H, E, I, A, F, C, G
   [✓ (d) D, H, I, E, B, F, G, C, A]

14. The result of preorder traversal is
   (a) A, B, C, D, E, F, G, H, I
   [✓ (b) A, B, D, E, H, I, C, F, G]
   (c) D, B, H, E, I, A, F, C, G
   (d) D, H, I, E, B, F, G, C, A

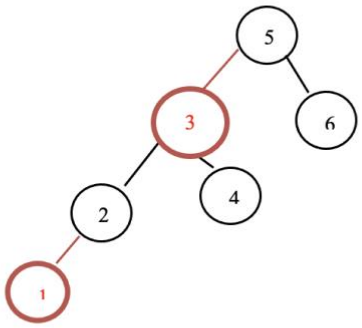
15. The result of inorder traversal is
    - (a) A, B, C, D, E, F, G, H, I
    - (b) A, B, D, E, H, I, C, F, G
    - [✓ (c) D, B, H, E, I, A, F, C, G]
    - (d) D, H, I, E, B, F, G, C, A

16. With an initially empty tree, the resulting red-black tree, after sequentially inserting 1, 2, 3, 4, 5, 6 looks like

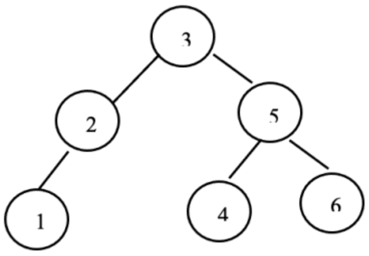
[✓ (a)]



(b)

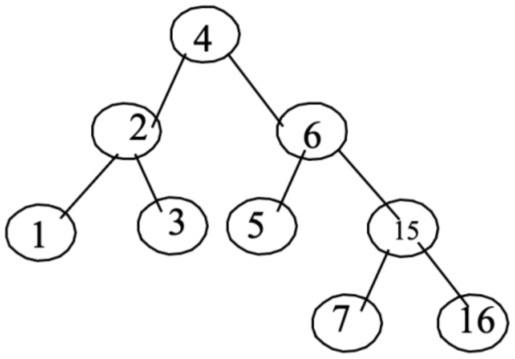


(c)



(d) None of above

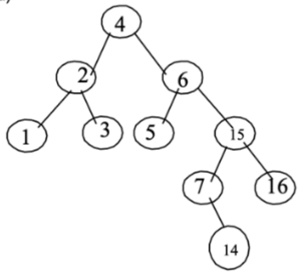
Given the following AVL tree, answer questions



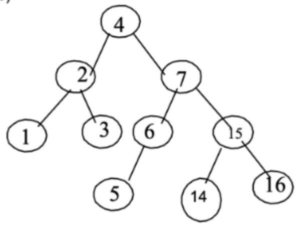
17. After placing 14 just according to BST property, which of the following is true?
    (a) It is still an AVL tree
    (b) It is not an AVL tree, and we need to rotate node 6 to convert it back to an AVL tree
    [✓ (c) It is not an AVL tree, and we need to rotate node 4 to convert it back to an AVL tree]
    (d) It is not an AVL tree, and we need to rotate both nodes 4 and 6 to convert it back to an AVL tree

18. The resulting AVL tree after inserting 14 looks like

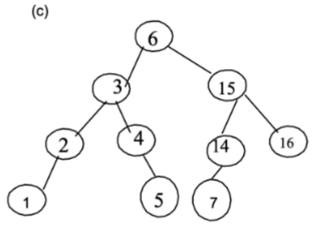
(a)



(b)



(c)



[✓ (d) None of above]

19. AVL Tree Programming (30 points)
    a. There is a 'height' data field in the AvlNode class definition. To get the height of an AVL node, which of the following two methods should you use? Why? (5 points)

```
// Method 1
```

```
private static int height(AvlNode t) {
   if (t == null) return -1;
   return t.height;
}
// Method 2
private static int height(AvlNode t) {
   if (t == null) return -1;
   return 1+ max(height(t.left), height(t.right));
}
```

b. Identify the errors in the code below for the single and double rotation to the right; then write the code accordingly for the single and double rotation to the left. (10 points)

```
private static AvlNode rotateToRight(AvlNode k2) {
   AvlNode k1 = k2.left;
   k2.left = k1.right;
   k1.right = k2;
   k1.height = max(height(k1.left), height(k2.height)) + 1;
   k2.height = max(height(k2.left), height(k2.right)) + 1;
   return k2;
}
private static AvlNode doubleRotateToRight(AvlNode k3) {
   k3.left = rotateToRight(k3.left);
   k3 = rotateToRight(k3)
   k3.height = max(height(k3.left), height(k3.right)) + 1;
   return k3;
}
```

c. Identify the errors and/or missing code in the “insert” and “remove” methods for the AvlNode class; then rewrite the code for the two methods. (15 points)

```
private AvlNode insert(AvlNode t, int x) {
    if (t == null) return new AvlNode(x, null, null, 0);
    if (x < t.val) {
            t.left = insert(t.left, x);
            if (height(t.left) - height (t.right) > 1)
                    if (x > t.left.val)
                            t = doubleRotateToRight(t.left);
                    else
                            t = rotateToRight(t.left);
    } else if (x > t.val) {
            t.right = insert(t.right), x);
            if (height(t.right) – height(t.left) == 2)
                    if (x > t.right.val)
                            t.right = doubleRotateToLeft(t);
                    else
                            t.right = rotateToLeft(t);
    } else ; // duplicate; do nothing

    Return t;
}
private AvlNode remove(AvlNode t, int x) {
    if (t == null) return null;
    if (x < t.val) {
            t.left = remove(t.left, x);
            if (height(t.right) - height(t.left) == 2) {
                    if (height(t.right.right) < height(t.right.left)
                            t = rotateToLeft(t.right);
                    else
                            t = doubleRotateToLeft(t.right);
            }
    } else if (x > t.val) {
            t.right = remove(t.right, x);
            if (height(t.left) - height(t.right) == 2) {
                    if (height(t.left.left) >= height(t.left.right)
                            t.left = rotateToRight(t);
                    else
                            t.left = doubleRotateToRight(t);
            }
    } else if (t.left != null && t.right != null) {
            t.val = findMin(t.right).val;
            t.right = remove(t.right, t.val);
    } else {
            If (t.left != null)
                    t = t.left;
            else
                    t = t.right;
     }

     return t;
}
```