Name & UNI: _Xintong (Amy) Qi xq2224_________ Date: _Nov 7_____________

Point values are assigned for each question. Points earned: ____ / 100, = ____ %

1. Give a recursive algorithm in pseudocode to compute the diameter of a binary tree. Recall that diameter is defined as the number of nodes on the longest path between any two nodes in the tree. Nodes have `left` and `right` references, but nothing else. You must use the height function, defined as follows, in your solution. Your solution will return the diameter of the tree as an integer.

```
function height(Node n)
1. if n = null
2.     return -1
3. return 1 + max(height(n.left), height(n.right))
```

Write your solution below. (8 points)

```
function diameter(Node n)
1. if n = null
2.     return 0 // base case
3. int left_dia = diameter(n.left); // in left subtree
4. int right_dia = diameter(n.right); // in right subtree
5. int dia = height(n.left) + height(n.right) + 1; // across middle
6. return max(dia, max(left_dia,right_dia));
```

2. Give an iterative algorithm in pseudocode to visit all the nodes in a binary tree. Nodes have `left` and `right` references as well as a `data` field. Each time a Node is visited, your solution will print its data field. You must use a stack in your solution.

Write your solution below. (12 points)

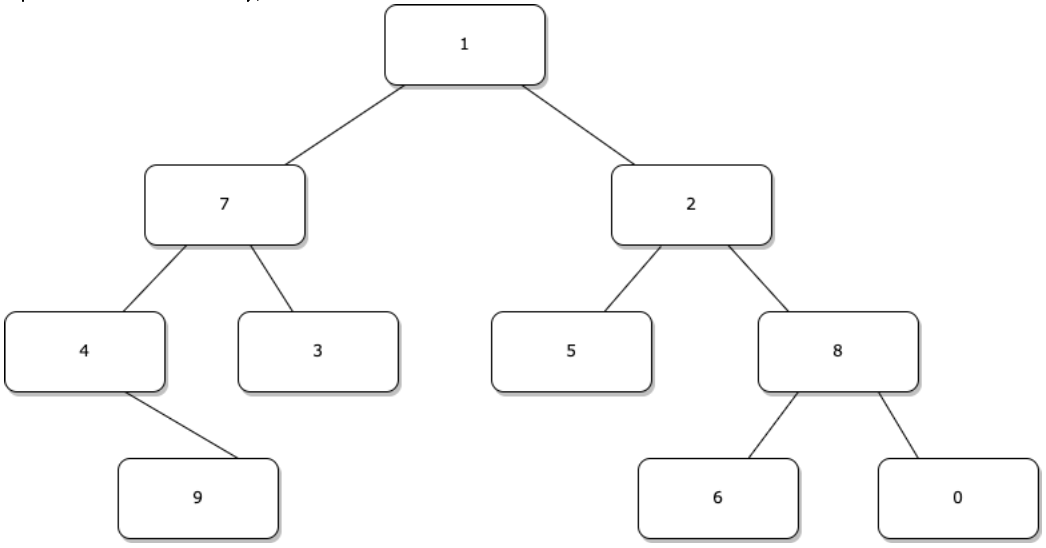
```
function iterative_inorder(Node root)
   // inorder: left root right
   // idea is to keep going left until you can't by pushing to stack

   1. if root = null
   2.     return // does nothing to an empty tree
   3. Node cur = root; // pointer to traverse the tree

   4. Stack<Node> stack = new Stack<Node>();
   5. while(cur != null || !stack.isEmpty())
   6.     while(cur != null)
   7.         stack.push(cur);
   8.         cur = cur.left; // pushing until the leftmost
   9.     cur = stack.pop(); // leftmost top of stack
   10.    System.out.print(cur.data); // print data
   11.    cur = cur.right; // print right if any
```

3. You are given the preorder and inorder traversals of a binary tree of integers that contains no duplicates. Using this information, reconstruct the binary tree and draw it. (10 points)
   Inorder: [4, 9, 7, 3, 1, 5, 2, 6, 8, 0]
   Preorder: [1, 7, 4, 9, 3, 2, 5, 8, 6, 0]

   Idea behind:
   Inorder: left root right
   Preorder: root left right
   We can find the root and divide the sequence into left and right correspondingly. Carry out this operation recursively, we can reconstruct the tree. The tree is reconstructed as below.



4. Write a recursive algorithm in pseudocode that finds the lowest common ancestor (LCA) of two given nodes in a binary tree T. The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself). If either p or q is null, the LCA is null. For this problem, Nodes have `left`, `right`, and `parent` references as well as a field called `level` which stores the level of the node in the tree. In the sample tree below, node 5 is on level 0, while nodes 4 and 6 are on level 1.

```
  5   <- level 0
 / \
4   6 <- level 1
```

Write your solution here. (10 points)

```
function lowest_common_ancestor(Node p, Node q)
   if (p.compareTo(q)==0):
       return p;
   if (p.level > q.level):
       return lowest_common_ancestor(p.parent, q);
   else if (p.level < q.level):
       return lowest_common_ancestor(p, q.parent);
   else:
       return lowest_common_ancestor(p.parent, q.parent);
```

5. Using structural induction, prove that the number of leaves in a non-empty full binary tree is one more than the number of internal nodes. (10 points)

Base case:

We'll show the statement is true for the smallest non-empty full binary tree, which is a single node

With a single node, the number of internal nodes is 0, and the number of leaves is 1. So the number of leaves is one more than the number of internal nodes, and the statement is true for base case.

Inductive step:

Assume the statement is true for a full binary tree with k nodes, i.e. the number of leaves in a full binary tree with k nodes is one more than the number of internal nodes. We'll show that the statement is also true for a full binary tree with k+2 nodes.

In a full binary tree with k+2 nodes, there must be at least one internal node t with two leaves being its children.

Removing the two children yields a full binary tree with k nodes, in which the number of leaves is one more than the number of internal nodes. (According to assumption)

Suppose originally the number of internal nodes in a tree with k nodes is x, and the number of leaves in the same tree is x+1.

Now consider the tree with k+2 nodes.

The number of internal nodes is x+1 because one of the leaves in the original tree now becomes an internal node; and the number of leaves is (x+1)-1+2, because one of the leaves becomes an internal node, and two new leaves are added compared with the original tree.

So the number of leaves – the number of internal nodes = (x+1)-1+2 – (x+1) = 1

And the statement holds for a full binary tree with k+2 nodes as long as it holds for a full binary tree with k nodes. The inductive step is completed.

Combine the inductive step and the base case, we know the statement is true for all non-empty full binary trees, i.e. the number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

6. Insert a Node with key 51 into the following AVL tree and draw the tree after performing all rotations. (5 points)

```
    50
   /  \
  /    \
30      55
       /  \
     52    60
```

Step 1: insert 51 to its place without any rotations

```
    50
   /  \
  /    \
 /      \
30      55
       /  \
      /    \
     52    60
    /
   51
```

Step 2: find the node that is unbalanced, which is 50. Notice that a double rotation is needed because we are inserting into the left subtree of the right child of the unbalanced node.

Step 3: perform the double rotation, and we obtain a balanced binary search tree as below.

```
       52
      /  \
     /    \
    /      \
   50       55
  /  \        \
 /    \        \
30    51        60
```

7. Draw an AVL tree of height 3 with the smallest possible number of nodes. (2 points)
   Then draw an AVL tree of height 4 with the smallest possible number of nodes. (3 points)

Note that the numbers marked on the nodes simply represent a node. They are NOT the keys and the AVL tree is NOT constructed accordingly to their numeric values.The real keys can be something else, and we assume that the tree is indeed a valid binary search tree with those keys.

AVL tree of height 3 with the smallest possible number of nodes:

```
         1
        / \
       /   \
      /     \
     2       3
    / \       \
   /   \       \
  4     5       6
 /
7
```

AVL tree of height 4 with the smallest possible number of nodes:

```
              1
             / \
            /   \
           /     \
          /       \
         2         3
        / \       / \
       /   \     /   \
      /     \   /     \
     4       5 6       7
    / \       \         \
   /   \       \         \
  8     9      10        11
 /
12
```

In pseudocode, write a function that returns the number of nodes in the smallest possible AVL tree of height h. You may assume that only valid heights from -1 up to 25 will be supplied.

Write your solution below. (5 points)

```
function minAvlSize(int height)
    if height == -1:
        return 0; // base case 1: an empty tree
    if height == 0:
        return 1; // base case 2: root itself
    return minAvlSize(height-1) + minAvlSize(height-2) + 1;
    // a minAvl has one subtree as minAvl with height-1, and
    // its other subtree as minAvl with height-2
    // plus the root level, which add 1 to height
```

8. Insert the following keys one-by-one into an initially empty hash table of size 7. Use the hash function h(x) = x mod 7

{10, 1, 18, 15, 26, 11, 19}

Show the result for

(a) a separate chaining hash table (do not rehash if the load factor becomes too large). (3 points)

The blocks below are indexed from 0 to 6. Within each is a linked list of values.
Insert 10:

			10		**26, 19**	

Insert 1:

	1		**10**		**26, 19**	

Insert 18:

	1, 15		**10**	**18, 11**	**26, 19**	

Insert 15: collision

	1, 15		**10**	**18, 11**	**26, 19**	

Insert 26:

	1, 15		**10**	**18, 11**	**26, 19**	

Insert 11: collision

	1, 15		**10**	**18, 11**	**26, 19**	

Insert 19: collision

	1, 15		**10**	**18, 11**	**26, 19**	

(b) a table with linear probing, i.e. using the probing function f(i) = i. (4 points)

The blocks below are indexed from 0 to 6. Within each is a single value.
Insert 10:

			10			

Insert 1:

	1		10			

Insert 18:

	1		10	18		

Insert 15: collision
Try to insert at (15+1) mod 7 = 2, ok

	1	15	10	18		

Insert 26:

	1	15	10	18	26	

Insert 11: collision
Try to insert at (11+1) mod 7 = 5, collision
Try to insert at (11+2) mod 7 = 6, ok

	1	15	10	18	26	11

Insert 19: collision
Try to insert at (19+1) mod 7 = 6, collision
Try to insert at (19+2) mod 7 = 0, ok

19	1	15	10	18	26	11

For parts (c) and (d), it is possible that you will not be able to insert some of the keys into the table. If you find such a key, rehash the table into a new table of size 11. You need to modify the hash function h(x) accordingly to adjust for the new table size. You do not have to change g(x) for part (d). Do not rehash earlier for any of the tables (even if the load factor becomes too large).

(c) a table with quadratic probing, i.e. using the probing function $f(i) = i^2$. (4 points)

The blocks below are indexed from 0 to 6. Within each is a single value.
Insert 10:

			10			

Insert 1:

	1		10			

Insert 18:

	1		10	18		

Insert 15: collision
Try to insert at (15+1^2) mod 7 = 2, ok

	1	15	10	18		

Insert 26:

	1	15	10	18	26	

Insert 11: collision
Try to insert at (11+1^2) mod 7 = 5, collision
Try to insert at (11+2^2) mod 7 = 1, collision
Try to insert at (11+3^2) mod 7 = 6, ok

	1	15	10	18	26	11

Insert 19: collision
Try to insert at (19+1^2) mod 7 = 6, collision
try to insert at (19+2^2) mod 7 = 2 collision
Try to insert at (19+3^2) mod 7 = 0, ok

19	1	15	10	18	26	11

(d) a table that uses double hashing with a secondary hash function, i.e. f(i) = i * g(x), where the secondary hash function is g(x) = 5 - (x mod 5). (4 points)

The blocks below are indexed from 0 to 6. Within each is a single value.
Insert 10:

			10			

Insert 1:

	1		10			

Insert 18:

	1		10	18		

Insert 15: collision
Try to insert at (15+1*(5-(15 mod 5))) mod 7= 6, ok

	1		10	18		15

Insert 26:

	1		10	18	26	15

Insert 11: collision
Try to insert at (11+1*(5-(11 mod 5))) mod 7= 1, collision
Try to insert at (11+2*(5-(11 mod 5))) mod 7= 5, collision
Try to insert at (11+3*(5-(11 mod 5))) mod 7= 2, ok

	1	11	10	18	26	15

Insert 19: collision
Try to insert at (19+1*(5-(19 mod 5))) mod 7= 3, collision

Try to insert at (19+2*(5-(19 mod 5))) mod 7= 0, ok

19	**1**	**11**	**10**	**18**	**26**	**15**

9.

(a) Insert the values {8, 12, 14, 11, 19, 16, 10, 7, 6} into an initially empty binary min heap. Show the heap after each insertion as an array, inserting the first node at index 1. You do not need to show each individual percolation step. (9 points)

Insert 8

	8								

Insert 12

	8	12							

Insert 14

	8	12	14						

Insert 11

	8	11	14	12					

Insert 19

	8	11	14	12	19				

Insert 16

	8	11	14	12	19	16			

Insert 10

	8	11	10	12	19	16	14		

Insert 7

	7	8	10	11	19	16	14	12	

Insert 6

	6	7	10	8	19	16	14	12	11

(b) Show the **final result** of using the linear-time buildHeap algorithm on the same input {8, 12, 14, 11, 19, 16, 10, 7, 6} with the root in index 1. You do not have to show individual steps. (3 points)

	6	7	10	8	19	16	14	12	11

(c) Perform one deleteMin operation on the heap below. Show the resulting heap. (3 points)

	1	2	3	4	5	6	7	8	9

Delete 1

Replace it with the last node

	9	2	3	4	5	6	7	8	

Keep bubbling down (intermediate steps omitted)

	2	4	3	8	5	6	7	9	

10. Instead of a binary heap, we could implement a d-ary heap, which uses d-ary tree. In such a tree, each node has between 0 and d children. As for the binary heap, we assume that a d-ary heap is a complete d-ary tree and can be stored in an array. Assuming the root is stored in index 1, if a node in the d-ary tree is at array position i, where is the parent of this node (2 points) and where are the children (3 points)?

Its parent is indexed at floor[(i+d-2)/d], where floor means the largest integer that is less than or equal to (i+d-2)/d. e.g. floor(1.2)=1, floor (1)=1, floor(1.9)=1

Its children are indexed at a range of values: id-d+2, id-d+3, … , id+1

The idea is to find the children first. The formula for the children can simply be derived through observation. As for the parent, we can work out its index backwards. Note how the term id contains i, and that the rest of the formula is a term ranging from -d+2 to 1. To eliminate its effect, we add d-2 to all of these terms, and we get id, id+1, … , id+d-1. Now divide all these terms by d, we get i, i+ 1/d, … , i+1-1/d, where i is incremented with a term ranging from 0 to 1-1/d, which is always less than or equal to 1 depending on the value of d. So we can simply take the floor of this value to eliminate its effect. That's how we get the index for the parent.