Assignment 6 - More on Lists

Q1.
For implementing multiple stacks and queues within a single array, I used a 2-d array which stores the data at the first col, and the pointer index info at the second col.
The initializer of Hybrid creates a 2 d int array, and each 2nd column is -1, representing no data stored for that row:

```
class Hybrid {
   public int[][] arr;

   Hybrid(int size) {
       this.arr = new int[size][2];
       for (int i = 0; i < size; i++) {
           this.arr[i][1] = -1;
       }
   }
}
```

The queue will have a head and tail index pointer besides the hybrid array:

```
class HybridQueue {
   public int head;
   public int tail;
   public Hybrid hyb;

   HybridQueue(Hybrid hyb) {
       this.hyb = hyb;
       this.head = -1;
       this.tail = -1;
   }
```

Add method searches unallocated space (rows having 2nd col as -1) and sets the new value to it and the tail pointer points itself. Then refresh the previous tail's pointer to the current slot, finally setting the current tail's pointer as -2, which means other queues or stacks cannot use this slot.

```
   public void add(int item) {
       if (isFull()) return;
       for (int i = 0; i < this.hyb.arr.length; i++) {
           if (this.hyb.arr[i][1] == -1) {
               if (this.head == -1) {
```

```
                this.head = i;
            } else {
                this.hyb.arr[tail][1] = i;
            }
            this.tail = i;
            this.hyb.arr[tail][0] = item;
            this.hyb.arr[tail][1] = -2;
            break;
        }
    }
}
```

Remove method is getting a value of head pointer points, and set the head pointer to the next location, and reset the previous head's val and next pointer.

```
public int remove() {
    if (!isEmpty()) {
        int tmp = this.hyb.arr[this.head][0];
        int nextHead = this.hyb.arr[this.head][1];
        this.hyb.arr[this.head][0] = 0;
        this.hyb.arr[this.head][1] = -1;
        this.head = nextHead;
        return tmp;
    } else {
        return 0;
    }
}
```

Peek() is using the head pointer to access the oldest allocated spot, and get the value from it:

```
public int peek(){
    return this.hyb.arr[this.head][0];
}
```

isEmpty method checks the header's pointer, if it's undefined or not:

```
public boolean isEmpty() {
    return this.head == -1;
}
```

isFull scanning through the hybrid array, and check if all spots are allocated or not:

```
public boolean isFull(){
    for (int i =0 ;i < this.hyb.arr.length; i++){
```

```
            if (this.hyb.arr[i][1]== -1){
                return false;
            }
        }
        return true;
    }
}
```

For stacks, the similar approach can be taken. Since it follows the LIFO characteristic, I defined the bottom element's next pointer as -2, and other elements have the pointer indicating the index of the element one below.

The HybridStack class has a pointer value of the top of the stack, and the hybrid array space:

```
class HybridStack {
   public int top;
   public Hybrid hyb;

   HybridStack(Hybrid hyb) {
        this.hyb = hyb;
        this.top = -1;
   }
```

Push operation can be done by searching for a new space and changing that space's value as given, also setting its next pointer as the previous index. Then update the top pointer value.

```
public void push(int item) {
        if (isFull()) return;
        for (int i = 0; i < this.hyb.arr.length; i++) {
            if (this.hyb.arr[i][1] == -1) {
                if (this.top == -1) {
                    this.hyb.arr[i][1] = -2;
                } else {
                    this.hyb.arr[i][1] = this.top;
                }
                this.hyb.arr[i][0] = item;
                this.top = i;
                break;
            }
        }
    }
```

Pop operation can be accomplished by accessing the top element of the stack and shift the top pointer to the one below it:

```
    public int pop() {
        if (!isEmpty()) {
            int tmp = this.hyb.arr[this.top][0];
            this.hyb.arr[this.top][0] = 0;
            this.hyb.arr[this.top][1] = -1;
            this.top = this.hyb.arr[this.top][1];
            return tmp;
        } else {
            return 0;
        }
    }
```

Peek, isEmpty and isFull methods are similar to ones for HybridQueue:

```
    public int peek(){
        return this.hyb.arr[this.top][0];
    }

    public boolean isEmpty() {
        return this.top == -1;
    }

    public boolean isFull(){
        for (int i =0 ;i < this.hyb.arr.length; i++){
            if (this.hyb.arr[i][1]== -1){
                return false;
            }
        }
        return true;
    }
}
```

Q2:
a node at level n in a binary tree has n ancestors:
If n = 0, the node itself is a root, so no ancestor.
If n = 1, the node's parent is a root node, and no ancestor for the root. So it has 1 ancestor
If n = 2, the node's parent is a node at level 1, and this parent node has 1 ancestor, then the child has 2 ancestors.

Hence for any n, a node at that level has n ancestors.

Q3.
Given a regular binary tree with n leaves, the bottom level has n nodes.
Since it's a binary tree, 2 children are connected to 1 parent. So for each level from the bottom of the tree, there are n/2 nodes, and n/4, n/8, …, n/(2^i) nodes, and for the root there's only one node.

So the total number of the nodes would be: $n + \frac{n}{2} + \frac{n}{4} + \ldots + 2 + 1 = 2n - 1$

Q4.

If there's only one node in the m-ary tree (n=1), the number of null child pointer fields would be m.

If n=2, there are 2*m pointer fields in total, and since two nodes are connected so the one pointer field of the parent is filled with a non-null node. Therefore 2*m-1 fields are empty.

For any number n of the nodes, there are n-1 parent-child relationships. And since each node has m fields, the total pointer fields in the tree is n*m.
Therefore the number of null child pointer fields is:

$$n \cdot m - (n - 1)$$
$$= nm - n + 1$$
$$= n(m - 1) + 1$$