C-1.24 Suppose that each row of an n × n array A consists of 1's and 0's such that, in any row of A, all the 1's come before any 0's in that row. Assuming A is already in memory, describe a method running in O(n) time (not O(n^2) time) for finding the row of A that contains the most 1's. (4 Marks)

```
j = 0

max_row = -1

for i from n-1 to 0 step -1 do {

       while (j < n)and(A[i,j] == 1) do {

              j = j+1

              max_row = i

      }

}

return max_row
```

C-1.29 Consider an extendable table that supports both add and remove methods, as defined in the previous exercise. Moreover, suppose we grow the underlying array implementing the table by doubling its capacity any time we need to increase the size of this array, and we shrink the underlying array by half any time the number of (actual) elements in the table dips below N/4, where N is the current capacity of the array. Show that a sequence of n add and remove methods, starting from an array with capacity N = 1, takes O(n) time. (4 Marks)

Sequence of add operations = 1, 2, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, …

Let c_i be the cost of the i-th insertion:

c_i = i if i−1 is a power of 2
1 otherwise

Let's consider the size of the table s_i and the cost c_i for the first few insertions in a sequence:

i ⇒	1	2	3	4	5	6	7	8	9	10
s_i ⇒	1	2	4	4	8	8	8	8	16	16
c_i ⇒	1	2	3	1	5	1	1	1	9	1

Alternatively we can see that $c_i=1+d_i$ where d_i is the cost of doubling the table size. That is

$d_i = i-1$ if $i-1$ is a power of 2

0 otherwise

Then summing over the entire sequence, all the 1's sum to O(n), and all the d_i also sum to O(n). That is,

$\Sigma_{1\le i\le n}\, c_i \;\le\; n + \Sigma_{0\le j\le m}\, 2^{j-1}$

where $m = \log(n - 1)$. Both terms on the right hand side of the inequality are O(n), so the total running time of n insertions is O(n).

Similarly, in a series of remove operations

R-2.7 Let T be a binary tree such that all the external nodes have the same depth. Let De be the sum of the depths of all the external nodes of T, and let Di be the sum of the depths of all the internal nodes of T. Find constants a and b such that

$De + 1 = aDi + bn,$

where n is the number of nodes of T. (4 Marks)

Consider n = 3.

Two external nodes both have depth 1 since each external node has one ancestor, namely the root. Thus, $D_e = 2$. The root is the only internal node and has no ancestors, thus $D_i = 0$.

Thus, in this case, equation is

$3 = 0 \cdot a + 3 \cdot b$

which implies that b = 1.

To determine another equation, consider the binary tree T with seven nodes (n = 7) in which the four external nodes are at the same depth, 2. So De = 8 in this case. Two of the internal nodes have depth 1 and the third (the root) has depth zero, thus Di = 2.

Equation (1) becomes

$9 = 2a + 7b$

but since b = 1 this equation implies that a = 1. So answer is a = b = 1

R-2.8 Let T be a binary tree with n nodes, and let p be the level numbering of the nodes of T, so that the root, r, is numbered as p(r) = 1, and a node v has left child numbered 2p(v) and right child numbered 2p(v) + 1, if they exist. (assume each internal node has two children)

a. Show that, for every node v of T, $p(v) \le 2^{(n+1)/2} - 1$.
b. Show an example of a binary tree with at least five nodes that attains the above upper bound on the maximum value of p(v) for some node v. (5 Marks)

a) In a binary tree with each internal node having 2 children,

$$p(v) <= 2^{h+1} - 1$$

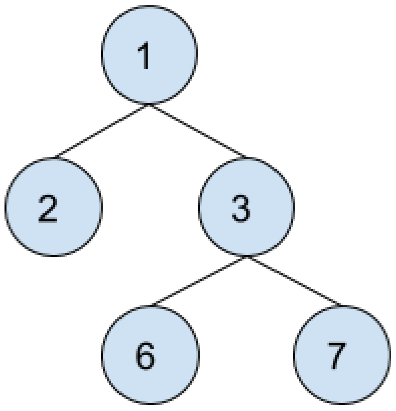
Also in general for binary trees

$$2h + 1 <= n <= 2^{h+1} - 1$$
$$\Rightarrow n >= 2h + 1$$
$$\Rightarrow n + 1 >= 2h + 1 + 1 \quad \text{(Adding 1 on both sides)}$$
$$\Rightarrow n + 1 >= 2(h + 1)$$
$$\Rightarrow (n + 1) / 2 >= h + 1$$
$$\Rightarrow 2^{(n+1)/2} >= 2^{h+1} \quad \text{(Taking power of 2 on both sides)}$$

Since $p(v) <= 2^{h+1}$ and $2^{h+1} <= 2^{(n+1)/2}$

Therefore **$p(v) \le 2^{(n+1)/2} - 1$**

b)



$P_{max}(v) = 7 = 2^{(5+1)/2} - 1$ n = 5

C-2.4 Describe how to implement a queue using two stacks, so that the amortized running time for dequeue and enqueue is O(1), assuming that the stacks support constant-time push, pop, and size methods. What is the worst-case running time of the enqueue() and dequeue() methods in this case? (4 Marks)

<u>enQueue(x)</u>
Push x to stack1 (assuming size of stacks is unlimited).

Time complexity O(1)

<u>deQueue(q)</u>

```
If both stacks are empty
    return null

 If stack2 is empty
    While stack1 is not empty do{
            e = stack1.pop()
            stack2.push(e)
    }
    return stack2.pop()
```

Here worst case time complexity will be O(n)

C-2.6 Describe a recursive algorithm for enumerating all permutations of the numbers {1, 2, . . . , n}. What is the running time of your method? (using O(n) space) (4 Marks)

```
def permute(string, start, end):
    if l==r:
      print(string)
  else:
      while i ⇐ start to end
         string[start], string[i] = string[i], string[start]        // Swap
         permute(string, start + 1, end)                            // Permutate
         string[start], string[i] = string[i], string[start]        // Swap back
```

total running time = n X n!

C-2.11 Design algorithms for the following operations for a node v in a binary tree T:

- **preorderNext(v): return the node visited after v in a preorder traversal of T**
- **inorderNext(v): return the node visited after v in an inorder traversal of T**
- **postorderNext(v): return the node visited after v in a postorder traversal of T.** **(5 Marks)**

```
preorderNext(v){
    temp = v->parent

    If(v->left!=NULL){
        return v->left
    }

    If(v->right!=NULL){
        Return v->right
    }
    Else{
        While(temp!=NULL && !temp->right){
            temp=temp->parent
        }

        If(temp!=NULL){
            temp=temp->right
        }
        Return temp
    }
}

inorderNext(v)
{
    If(v->right != NULL){
        temp = v->right;

        While (temp!=NULL && temp-> left != NULL){
            temp = temp->left;
        }
        Return temp
    }

    If (v->right == NULL){
        temp = v-> parent
```

```
                While (!temp-> right && temp!=NULL){
                    temp=temp->parent
                }

            If(temp!=NULL){
                temp=temp->right
            }

            While(temp!=NULL&&temp->left!=NULL){
                temp=temp->left
            }
            Return temp;
        }
}
```

postorderNext(v){

```
        temp =v->parent;
        if(v==temp->right){
            Return temp
        }
        Else if(temp->right!=NULL){
            temp = temp->right;
            While (temp!=NULL && temp-> left != NULL){
                temp = temp->left;
            }
            Return temp
        }
        Else{
            While(temp!=NULL && !temp->right){
                temp=temp->parent
            }
            If(temp!=NULL){
                temp = temp-> right

                While(temp!=NULL && temp-> left != NULL){
                    temp = temp->left
                }
            }
            Return temp;
        }
```