Birla Institute of Technology & Science, Pilani

SSZG519, Data Structure Algorithm & Design

MID- Semester- Solution, EC-2 (Make-UP)

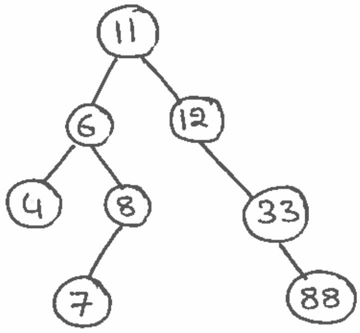
S1 18-19

1. Pre-order: A, B, D, H, I, E, J, C, F, G, K. — 1M

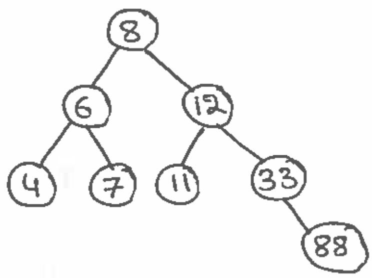
Post-order: H, I, D, J, E, B, F, K, G, C, A. — 1M

In-order: H, D, I, B, E, J, A, F, C, G, K. — 1M

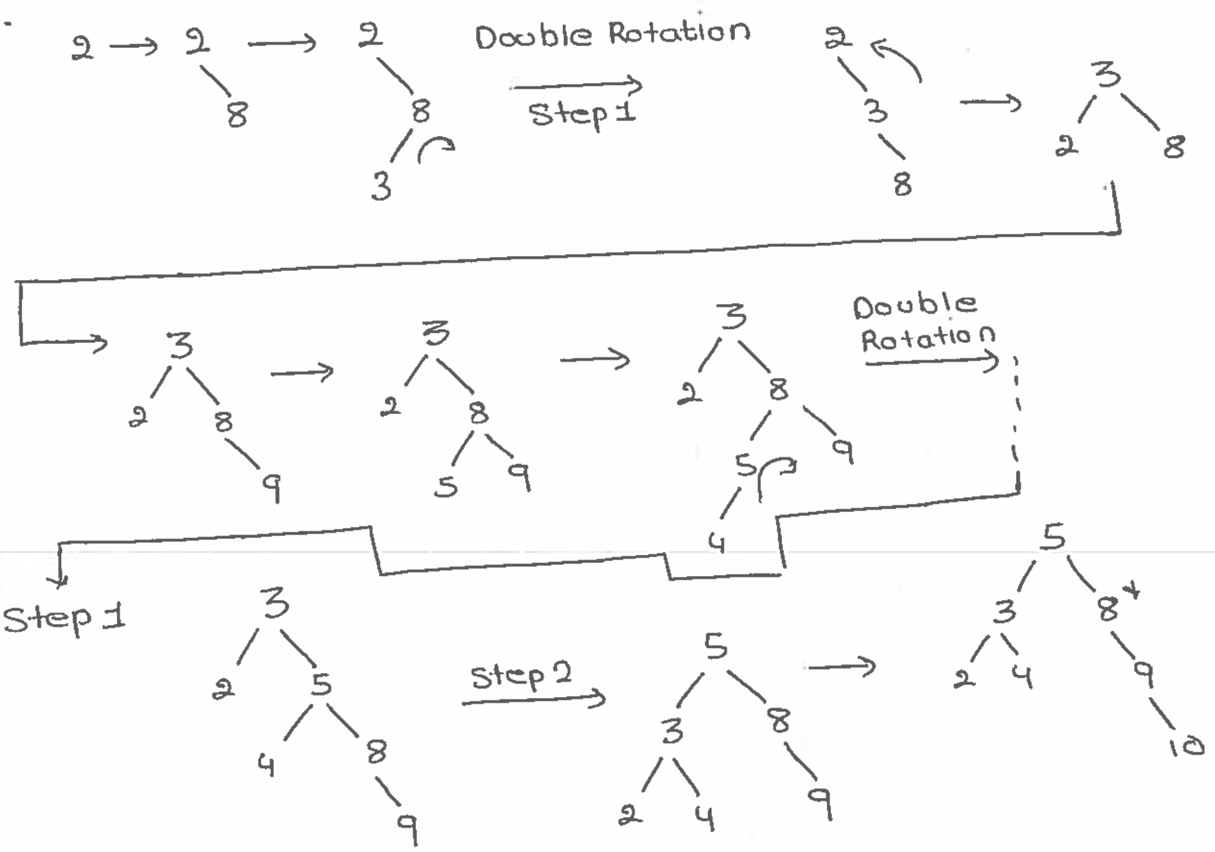
2. Replace with sucessor.

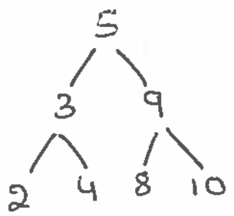


Replace with predecessor



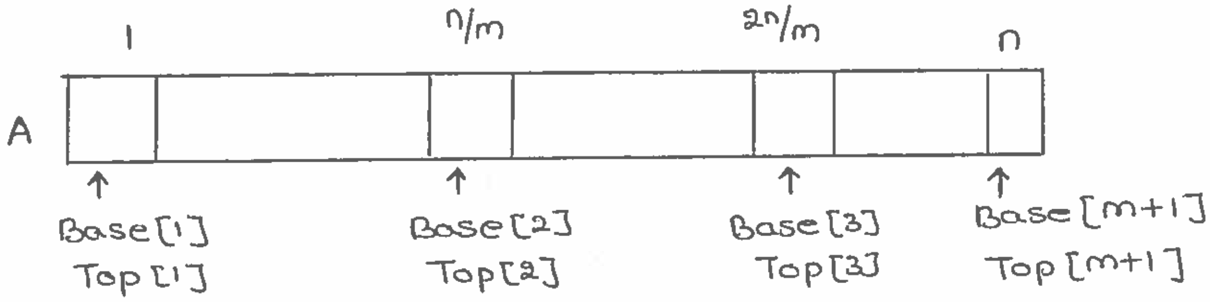
3.





finial answer.

4.
let us assume that array indexes are from 1 to n. Similar to the problem to implement 3 stacks, to implement m stacks in one array, we divide the array into m parts (as shown below). The size of each part is $\frac{n}{m}$.



from, the above representation we can see that, first stack is starting at index 1 (starting index is stored in Base[1]), second stack is starting at index $\frac{n}{m}$ (starting index is stored in Base[2]), third stack is starting at index $2n/m$ (starting index is stored in Base[3]), and so on. Similar to Base array, let us assume that top array stores the top indexes for each of the stack.

Consider the following terminology for the discussion.

↳ Top[i], for $1 \le i \le m$ will point to the topmost element of the stack i

↳ If Base [i] == Top [i], then we can say the stack i is Empty

↳ If Top [i] == Base [i+1], then we can say the stack i is full.

Initially Base [i] = Top [i] = $\frac{n}{m}(i-1)$, for $1 \le i \le m$.

↳ The i^{th} stack grows from Base [i]+1 to Base [i+1].

Pushing on to i^{th} stack:

1) for pushing on to the i^{th} stack, we check whether the top of i^{th} stack is pointing to Base [i+1] (this case defines that i^{th} stack is full). That means, we need to see if adding a new element causes it to bump into the $i+1^{th}$ stack. If so, try to shift the stacks from $i+1^{th}$ stack to m^{th} stack toward the right. Insert the new element at [Base [i]+ Top[i]).

2) If right shifting is not possible then try shifting the stacks from 1 to $i-1^{th}$ stack toward the left.

3) If both of them are not possible then we can say that all stacks are full.

```
void push(int stack2D ,int data) {
    if (Top[i]== Base [i+1])
      print ith stack is full and does the necessary action
                                              (shifting);

       Top [i] = Top[i]+1;
       A[Top[i]]= data;
    }
```

Time complexity: O(n), since we may need to adjust the stacks. Space complexity: O(1).

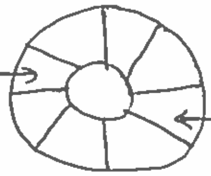
Popping from i^{th} stack:

for popping, we don't need to shift, just decrement the size of the appropriate stack. The only case to check is stack Empty case.

```
int Pop (int Stack2D) {
    if (Top[i]== Base[i])
        Print ith stack is Empty;
    return A[Top[i]--];
  }.
```

Time complexity: O(1), space complexity: O(1)

5) Consider the following figure to get a clear idea of the queue.



↳ Rear of the queue is ~~sometim~~ somewhere clockwise from the front.

↳ To enqueue an element, we move rear one position clockwise and write the element in that position.

↳ To dequeue, we simply move front one position clockwise.

↳ Queue migrates in a clockwise direction as we enqueue and dequeue.

↳ Emptiness and fullness to be checked carefully.

↳ Analyze the possible situations (make some drawings to see where front and rear are when the queue is empty, and partially and totally filled). we will get this

$$\text{Number of Element} = \begin{cases} rear - front + 1 & \text{if } rear == front \\ rear - front + n & \text{otherwise.} \end{cases}$$

6)
a) $O(n^2 \log n)$
b) $O(n^5)$

7)
a) True

$T(n) = T(n-1) + n$
$T(n-1) = T(n-2) + n - 1$
...
$+ T(2) = T(1) + 2.$

$T(n) = 1 + 2 + \cdots + n = n(n+1)/2.$

Then we get $T(n) = O(n^2) = O(n^3)$.

b) false

c) True

Detail explanation of question

6a)
```
void function (int n)
{
 int count = 0;
 //outer loop executes n/2 times
 for (int i=n/2; i<=n; i++)
 // middle loop executes n/2 times
   for(int j=1; j+n/2<=n; j=j++)
  // inner loop executes logn times
   for(int k=1; k<=n; k=k*2)
     count++;
 }
```

Time complexity of the above function $O(n^2 \log n)$

6b)
```
void function (int n)
{ int count = 0;
 // Executes n times
   for (int i=0; i<n; i++);
 // Executes O(n*n) times
   for (int j=1; j<i*i; j++)
     if (j%i==0)
 {
   // Executes j times = O(n*n) times
   for (int k=0; k<j; k++)
      printf("*");
  }
 }
```

Time complexity of the above function $O(n^5)$.

Q7

a) True

$T(n) = T(n-1) + n$

$T(n-1) = T(n-2)$

$\vdots$

$+T(2) = T(1) + 2$

$T(n) = 1 + 2 + \cdots + n = n(n+1)/2.$

Then we get $T(n) = O(n^2) = O(n^3)$.

b) false

There are many ways to show this is false, here is one. consider the problem, where given n-numbers as input, the algorithm has to output all the permutations of the n-numbers since there are n! permutations that need to be output every algorithm for this problem runs in exponential time.

c) True,

Do a linear scan & remember the ~~two~~ three smallest numbers seen so far. whenever you encounter a new number, one can figure out in constant time, if it should displace any of the correct three minimum guys. At the end of the linear scan, output the third smallest number.

[the running time is $O(n)$ as the algorithm spends only $O(1)$ time per element).