**Design a stack with operations on middle element**

How to implement a stack which will support following operations in O(1) time complexity?

1) push() which adds an element to the top of stack.

2) pop() which removes an element from top of stack.

3) findMiddle() which will return middle element of the stack.

4) deleteMiddle() which will delete the middle element.

**We will use DLL for stack implementation** (Array will take O(n) for deleting middle element and it is not possible to delete middle element as pointer in both the direction is not there.) We can move mid pointer in both directions using previous and next pointers in DLL.

If there are even elements in stack, findMiddle() returns the first middle element. For example, if stack contains {1, 2, 3, 4}, then findMiddle() would return 2.

**Update the mid pointer when the count is odd after pushing and even after popping.**

# Implement Stack using Queues

A stack can be implemented using two queues. Let stack to be implemented be ‘s’ and queues used to implement be ‘q1′ and ‘q2′. Stack ‘s’ can be implemented in two ways:

**Method 1 (By making push operation costly)**  
This method makes sure that newly entered element is always at the front of ‘q1′, so that pop operation just dequeues from ‘q1′. ‘q2′ is used to put every new element at front of ‘q1′.

push(s, x) // x is the element to be pushed and s is stack

1) Enqueue x to q2

2) One by one dequeue everything from q1 and enqueue to q2.

3) Swap the names of q1 and q2

// Swapping of names is done to avoid one more movement of all elements

// from q2 to q1.

pop(s)

1) Dequeue an item from q1 and return it.

**Method 2 (By making pop operation costly)**  
In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

push(s, x)

1) Enqueue x to q1 (assuming size of q1 is unlimited).

pop(s)

1) One by one dequeue everything except the last element from q1 and enqueue to q2.

2) Dequeue the last item of q1, the dequeued item is result, store it.

3) Swap the names of q1 and q2

4) Return the item stored in step 2.

// Swapping of names is done to avoid one more movement of all elements

// from q2 to q1.

# Merge Overlapping Intervals

1. Sort all intervals in increasing order of start time.
2. Traverse sorted intervals starting from first interval,

Do following for every interval. Check the end time of current with start time of next.

If(start time(next)<=end time(current))

Include next interval in this

Else

Make next as new interval and calculate for it accordingly.

TC : O(nlogn)

# Implement GetMin() in O(1) in stack

Use two stacks: one to store actual stack elements and other as an auxiliary stack to store minimum values. The idea is to do push() and pop() operations in such a way that the top of auxiliary stack is always the minimum.

**Push(int x) // inserts an element x to Special Stack**  
1) push x to the first stack (the stack with actual elements)  
2) compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.  
…..a) If x is smaller than y then push x to the auxiliary stack.

.

**int Pop() // removes an element from Special Stack and return the removed element**

Pop the element from stack1. If it is same as stack2 then pop the element from stack2 also.  
We can also manage index while in the element structure in case of multiple instances of same value.

**int getMin() // returns the minimum element from Special Stack**  
Return the top element of auxiliary stack.

We can see that **all above operations are O(1)**.

# Implement two stacks in an array

# ****(A space efficient implementation)****

# It doesn’t cause an overflow if there is space available in arr[].

# The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1).

# Both stacks grow (or shrink) in opposite direction.

# To check for overflow, all we need to check is for space between top elements of both stacks.

# Implement a stack using single queue

// x is the element to be pushed and s is stack

**push(s, x)**

1) Let size of q be s.

1) Enqueue x to q

2) One by one Dequeue s items from queue and enqueue them.

// Removes an item from stack

**pop(s)**

1) Dequeue an item from q

# Sort a stack using recursion

// Recursive function to insert an item x in sorted way

void sortedInsert(struct stack \*\*s, int x)

{

    // Base case: Either stack is empty or newly inserted

    // item is greater than top (more than all existing)

    if (isEmpty(\*s) || x > top(\*s))

    {

        push(s, x);

        return;

    }

    // If top is greater, remove the top item and recur

    int temp = pop(s);

    sortedInsert(s, x);

    // Put back the top item removed earlier

    push(s, temp);

}

// Function to sort stack

void sortStack(struct stack \*\*s)

{

    // If stack is not empty

    if (!isEmpty(\*s))

    {

        // Remove the top item

        int x = pop(s);

        // Sort remaining stack

        sortStack(s);

        // Push the top item back in sorted stack

        sortedInsert(s, x);

    }

}

TC : O(n^2)

# Reverse a stack using recursion

// Below is a recursive function that inserts an element

// at the bottom of a stack.

void insertAtBottom(struct sNode\*\* top\_ref, int item)

{

    if (isEmpty(\*top\_ref))

        push(top\_ref, item);

    else

    {

        /\* Hold all items in Function Call Stack until we

           reach end of the stack. When the stack becomes

           empty, the isEmpty(\*top\_ref)becomes true, the

           above if part is executed and the item is inserted

           at the bottom \*/

        int temp = pop(top\_ref);

        insertAtBottom(top\_ref, item);

        /\* Once the item is inserted at the bottom, push all

           the items held in Function Call Stack \*/

        push(top\_ref, temp);

    }

}

// Below is the function that reverses the given stack using

// insertAtBottom()

void reverse(struct sNode\*\* top\_ref)

{

    if (!isEmpty(\*top\_ref))

    {

        /\* Hold all items in Function Call Stack until we

           reach end of the stack \*/

        int temp = pop(top\_ref);

        reverse(top\_ref);

        /\* Insert all the items (held in Function Call Stack)

           one by one from the bottom to top. Every item is

           inserted at the bottom \*/

        insertAtBottom(top\_ref, temp);

    }

}

TC : O(n^2)