STACK & QUEUE - SET 2 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://practice.geeksforgeeks.org/problems/reverse-first-k-elements-of-queue/1/?category[]=Stack&category[]=Stack&page=1&query=category[]Stackpage1category[]Stack
- **Difficulty level**: Easy

Explanation: we can achieve the required result by using a temporary stack:

- Make a temporary stack empty
- Deque first k elements from the queue one by one and as and when dequeued push them into the empty stack.
- Then pop the stack elements one by one and enqueue them at the back of the queue
- Dequeue (size of queue - k) elements from the front and enque them one by one to the same queue.

Solution:

```
#include <bits/stdc++.h>
using namespace std;

void modifyQueue(int k, queue<int>& Queue)
{
    if (Queue.empty() == true || k > Queue.size())
        return;
    if (k <= 0)
```

```
        return;

    stack<int> Stack;
    for (int i = 0; i < k; i++)
    {
        Stack.push(Queue.front());
        Queue.pop();
    }

    while (!Stack.empty())
    {
        Queue.push(Stack.top());
        Stack.pop();
    }

    for (int i = 0; i < Queue.size() - k; i++)
    {
        Queue.push(Queue.front());
        Queue.pop();
    }
}

void Print(queue<int>& Queue)
{
    while (!Queue.empty())
    {
        cout << Queue.front() << " ";
        Queue.pop();
    }
}
```

Complexity:

- Time: O(N)
- Space: O(N)

Question 2:

- **Problem link**: https://practice.geeksforgeeks.org/problems/get-min-at-pop/1/?category[]=Stack&category[]=Stack&page=1&query=category[]Stackpage1category[]Stack
- **Difficulty level**: Easy

Explanation: start by pushing elements from the array to the stack, and in the process of doing that find the minimum element in each pass of the for loop by checking if it is lesser than the previous minimum, if it is lesser, then make that element as the new minimum and push that element onto the stack. And finally push the minimum element to the stack too. This way the topmost element is always the smallest in the stack. But since there can be duplicate elements in the stack because we are pushing twice to the stack(the minimum and element being pushed can be the same), that's why we pop twice before the next top element is the smallest in the stack at that point in time.

Solution:

```
#include <bits/stdc++.h>
using namespace std;

stack<int> _push(int arr[],int n);

void _getMinAtPop(stack<int>s);
stack<int> _push(int arr[],int n)
{ stack<int> s;
   s.push(arr[0]);
   int min=arr[0];
   s.push(min);
   for(int i=1;i<n;i++)
   {
       if(arr[i]<=min)
         min=arr[i];
```

```
        s.push(arr[i]);
       s.push(min);
    }
    return s;
}
void _getMinAtPop(stack<int>s)
{
    while(!s.empty())
    {
        cout<<s.top()<<" ";
        s.pop();
        s.pop();
    }
}
```

Complexity:

- Time: O(N)
- Space: O(N)

Question 3:

- **Problem link**: https://www.hackerearth.com/practice/data-structures/queues/basics-of-queues/practice-problems/algorithm/disk-tower-b7cc7a50/
- **Difficulty level**: Easy

Explanation: we use priority queues to solve this problem. As and when we take inputs for each size of disk, we push that size to a queue. Initially the max available size is N. Then after pushing the size to the queue, we check if that size is the same as the max available space now. If they both are equal then we simply print the top of the queue and then reduce the size of the max available space by 1 and then pop the top element. We do this step until max available space is not equal to the current top element. This entire process is repeated n times to achieve the desired result.

Solution:

```
#include <bits/stdc++.h>
using namespace std ;
int main ()
{
    int n;
    cin>>n;
    int max = n;
    priority_queue <int> b;
    int a[n] ;

    for (int i=0; i<n ;i++)
       {
        cin >> a[i];
        b.push(a[i]);
        while (b.top() == max){
            cout << b.top() << " " ;
            max--;
            b.pop();
               }

        cout << endl ;
        }

return 0;}
```

Complexity:

❖ Time: O(N)
❖ Space: O(N)

Question 4:

- **Problem link**: https://leetcode.com/problems/next-greater-element-i/
- **Difficulty level**: Easy

Explanation:

Maintain a map in which the next greater element for each element of the array will be stored. Maintain a stack to store the indices of the elements whose next greater element has not been found so far. Iterate over the 'nums' array once. For each element, pop all indices on the top of the stack whose corresponding element is less than the current element and for each of these popped indices, the next greater element will be the current element. Put this in the map. Now push the current index to the stack. When this is done for all the elements of the array, the map will contain the next greater element for each of the elements of the array. If the map doesn't contain an element, then, there exists no greater element for that element.

Solution:

```
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>&
nums2) {

        int n1 = nums1.size();
        int n2 = nums2.size();

        unordered_map<int, int> nextGreater;
        stack<int> st;

        for(int i=0; i<n2; i++)
        {
            while(!st.empty() && nums2[st.top()] < nums2[i])
            {
                nextGreater[nums2[st.top()]] = nums2[i];
                st.pop();
            }

            st.push(i);
        }

        vector<int> output;

        for(int i=0; i<n1; i++)
        {
            if(nextGreater.find(nums1[i]) != nextGreater.end())
            {
                output.push_back(nextGreater[nums1[i]]);
            }
            else
            {
                output.push_back(-1);
            }
        }

        return output;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n)

Question 5:

- **Problem link**: https://leetcode.com/problems/next-greater-element-ii/
- **Difficulty level**: Medium

Explanation:
This problem is an extension of Q4 and can be done in 2 passes. Start traversing ‘nums’ from left to right and only push onto the stack if the element is less than the element at the index at the top of the stack. If the element is greater than the element at the index at the top of the stack, then, this happens to be the next greatest element. So, pop all the indices from the stack whose corresponding element is smaller and put the current element as the next greater element for each of the popped indices. Repeat this a second time as to account for the circular-ness of the array.

Solution:
```
class Solution {
public:
    vector<int> nextGreaterElements(vector<int>& nums) {

        int n = nums.size();
        stack<int> st;
        vector<int> output(n, -1);

        for(int i=0; i<n; i++)
        {
            while(!st.empty() && nums[st.top()]<nums[i])
            {
                output[st.top()] = nums[i];
                st.pop();
            }

            st.push(i);
        }

        for(int i=0; i<n; i++)
        {
            while(!st.empty() && nums[st.top()]<nums[i])
            {
                output[st.top()] = nums[i];
```

```
                    st.pop();
                }

                st.push(i);
            }

            return output;
        }
};
```

Complexity:

- Time: O(n)
- Space: O(n)

Question 6:

- **Problem link**: https://leetcode.com/problems/remove-k-digits/
- **Difficulty level**: Medium

Explanation:

In stack push an element if it is bigger than top of the stack
For Example:- num = "1211" and k = 2
insert first element stack=1
1211 here 1(stack top) < 2 so take it stack=12 k=2
1211 here 2(stack top) > 1 so remove all elements greater than 1 alongside reducing k ,then take 1 stack=11 k = 1
1211 here 1(stack top) =1 so take it stack=111 k=1
since k!=0 take out k elements from stack stack=11 k=0
Result:- 11

Solution:

```
string removeKdigits(string num, int k) {
        if(num.length()==k)return "0";
        // get all eligible elements in a stack
      stack<char> s;
        for(char c:num){
            while(k && !s.empty() &&
int(s.top())>int(c)){s.pop();k--;}
            s.push(c);
        }
        //if still no change in k then remove k elements
        if(k) while(k--){s.pop();}
        // remove leading 0s ,if present
        stack<char> t;
        while(!s.empty()){t.push(s.top());s.pop();}
        while(t.top()=='0'&&t.size()!=1)t.pop();
        //return the answer
        num="";
        while(!t.empty()){num.push_back(t.top());t.pop();}
        return num;
    }
```

Complexity:

- Time: O(n)
- Space: O(n)

Question 7:

- **Problem link**: https://practice.geeksforgeeks.org/problems/count-the-reversals0401/1/?category[]=Queue&category[]=Queue&page=1&query=category[]Queuepage1category[]Queue
- **Difficulty level**: Medium

Explanation: we first start by removing all balanced parentheses. For eg. }{{}}{{{" to "}{{{. After this step we are left with an expression that contains 0 or more numbers of closing brackets followed by 0 or more numbers of opening brackets.
Let m be the total number of closing brackets and n be the number of opening brackets. We need ⌈m/2⌉ + ⌈n/2⌉ reversals. For example }}}}{{ requires 2+1 reversals.

Solution:

```
#include<bits/stdc++.h>
using namespace std;

int countMinReversals(string expr)
{
    int len = expr.length();
    if (len%2)
       return -1;
    stack<char> s;
    for (int i=0; i<len; i++)
    {
        if (expr[i]=='}' && !s.empty())
        {
            if (s.top()=='{')
                s.pop();
            else
                s.push(expr[i]);
        }
        else
            s.push(expr[i]);
    }

    int red_len = s.size();
    int n = 0;
    while (!s.empty() && s.top() == '{')
    {
        s.pop();
        n++;
    }

    return (red_len/2 + n%2);
}
```

Complexity:

- Time: O(|S|)
- Space: O(1)

Question 8:

- **Problem link**: https://practice.geeksforgeeks.org/problems/longest-valid-parentheses5657/1/?category[]=Stack&category[]=Stack&page=2&query=category[]Stackpage2category[]Stack
- **Difficulty level**: Hard

Explanation: Create an empty stack and push -1 to it. The first element of the stack is used to check if the next string is valid. Initially the resulting longest valid substring is 0. If the character is '(' i.e. str[i] == '(', push index 'i' to the stack. Else (if the character is ')'), then pop an item from the stack (Most of the time an opening bracket). Else if the stack is not empty, then find the length of current valid substring by taking the difference between the current index and top of the stack. If current length is more than the result, then update the result. Else If the stack is empty, push the current index as a base for the next valid substring. Then finally print the result string.

Solution:

```cpp
#include <bits/stdc++.h>
using namespace std;

int maxLength(string S)
{
    int n = S.length();
    stack<int> stk;
    stk.push(-1);
    int result = 0;
    for (int i = 0; i < n; i++)
    {
        if (S[i] == '(')
            stk.push(i);
        else
        {
            if (!stk.empty())
                stk.pop();

            if (!stk.empty())
                result = max(result, i - stk.top());
            else
                stk.push(i);
        }
    }

    return result;
}
```

Complexity:

- Time: O(|S|)
- Space: O(|S|)