STACK & QUEUE - SET 1 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://leetcode.com/problems/valid-parentheses/
- **Difficulty level**: Easy

Explanation:

1. Initialize a stack S.
2. Process each bracket of the expression one at a time.
3. If we encounter an opening bracket, we simply push it onto the stack. This means we will process it later, let us simply move onto the sub-expression ahead.
4. If we encounter a closing bracket, then we check the element on top of the stack. If the element at the top of the stack is an opening bracket of the same type, then we pop it off the stack and continue processing. Else, this implies an invalid expression.
5. In the end, if we are left with a stack still having elements, then this implies an invalid expression.

Solution:

```
class Solution {
public:
    bool isValid(string s) {
        stack<char>stack;
        int n=s.length();

        for(int i=0;i<n;i++)
        {
            if(s[i]=='}' or s[i]==')' or s[i]==']')
            {
                if(stack.empty())
                {
                    return false;
                }
                char top=stack.top();
                if(s[i]==']' and top!='[' or s[i]=='}' and top!='{'
or s[i]==')' and top!='(')
                {
                    return false;
                }
                stack.pop();
            }
            else
            {
                stack.push(s[i]);
            }
        }
        if(!stack.empty())
        {
            return false;
        }
        return true;
    }
};
```

Complexity:

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

Question 2:

- **Problem link**: https://leetcode.com/problems/remove-all-adjacent-duplicates-in-string/
- **Difficulty level**: Easy

Solution:

```
class Solution {
public:
string removeDuplicates(string s) {
   stack<char> temp;

    for(int i=0 ; i<s.size() ; i++)
    {
        if(temp.empty()) temp.push(s[i]);
        else
        {
            if(s[i] == temp.top()) temp.pop();
            else temp.push(s[i]);
        }
    }

    s.resize(temp.size());
    for(int i = s.size()-1 ; i>=0 ; i--)
    {
        s[i] = temp.top();
        temp.pop();
    }

    return s;
}
};
```

Complexity:

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

Question 3:

❖ **Problem link**: https://leetcode.com/problems/implement-stack-using-queues/
❖ **Difficulty level**: Easy

Explanation:
This method implements a stack using queues by making the push operation costly. This method makes sure that the newly entered element is always at the front of queue ‘q1’, so that pop operation just dequeues from ‘q1’. Queue ‘q2’ is used to put every new element in front of ‘q1’.

- push(s, x) operation’s step are described below:
  - Enqueue x to q2
  - One by one dequeue everything from q1 and enqueue to q2.
  - Swap the names of q1 and q2
- pop(s) operation’s function are described below:
  - Dequeue an item from q1 and return it.

Solution:
```
class MyStack {
public:
    queue<int> q1, q2;

    /** Initialize your data structure here. */
    MyStack() {

    }

    /** Push element x onto stack. */
    void push(int x) {
        q2.push(x);

        while(!q1.empty())
        {
            q2.push(q1.front());
            q1.pop();
        }

        queue<int> temp = q1;
        q1 = q2;
        q2 = temp;
    }

    /** Removes the element on top of the stack and returns that
element. */
    int pop() {
        if(!q1.empty())
        {
            int element = q1.front();
            q1.pop();
            return element;
        }

        return -1;
    }

    /** Get the top element. */
    int top() {
        if(!q1.empty())
            return q1.front();

        return -1;
    }

    /** Returns whether the stack is empty. */
    bool empty() {
        if(q1.empty() && q2.empty())
            return true;
        return false;
    }
};
```

Complexity:

- Time: ‘Push’ takes O(n) time. ‘Pop’ takes O(1) time
- Space: O(n)

Question 4:

- **Problem link**: https://leetcode.com/problems/implement-queue-using-stacks/
- **Difficulty level**: Easy

Explanation:

In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

- enQueue(q, x)
    - Push x to stack1 (assuming size of stacks is unlimited).

  Here time complexity will be O(1)

- deQueue(q)
    - If both stacks are empty then error.
    - If stack2 is empty
        - While stack1 is not empty, push everything from stack1 to stack2.
    - Pop the element from stack2 and return it.

  Here time complexity will be O(n)

Solution:

```
class MyQueue {
public:
    stack<int> stack1, stack2;

    /** Initialize your data structure here. */
    MyQueue() {

    }

    /** Push element x to the back of queue. */
    void push(int x) {
        stack1.push(x);
    }

    /** Removes the element from in front of queue and returns that
element. */
    int pop() {
        if(stack1.empty() && stack2.empty())
            return -1;

        if(stack2.empty())
        {
            while(!stack1.empty())
            {
                stack2.push(stack1.top());
                stack1.pop();
            }
        }

        int element = stack2.top();
        stack2.pop();
        return element;
    }
    /** Get the front element. */
    int peek() {
        if(stack1.empty() && stack2.empty())
            return -1;

        if(stack2.empty())
        {
            while(!stack1.empty())
            {
                stack2.push(stack1.top());
                stack1.pop();
            }
        }

        return stack2.top();
    }

    /** Returns whether the queue is empty. */
    bool empty() {
        if(stack1.empty() && stack2.empty())
            return true;
        return false;
    }
};
```

Complexity:

❖ Time: ‘Enqueue’ operation takes O(1) time and ‘Dequeue’ operation takes O(n) time

❖ Space: O(n)

Question 5:

- **Problem link**: https://leetcode.com/problems/design-circular-queue/
- **Difficulty level**: Medium

Explanation:
Use a vector of fixed size to implement a circular queue. Maintain two pointers 'front' and 'rear' which point the beginning and end of the current sequence of elements in the queue. Maintain a 'count' variable to track the number of elements inside the queue. This will help us to check if the queue is full or empty. For 'enqueue' operation, if the queue is not full, increment the 'rear' pointer and add the element at this position. For 'dequeue' operation, if the queue is not empty, set the element at 'front' to -1 and increment the 'front' pointer.

Solution:

```
class MyCircularQueue {
public:
    vector<int> queue;
    int size;
    int front;
    int rear;
    int count;

    MyCircularQueue(int k) {
        size = k;
        front = 0;
        rear = size-1;
        count = 0;
        queue = vector<int> (size, -1);
    }

    bool enQueue(int value) {
        if(isFull())
```

```
            return false;

        rear = (rear+1)%size;
        queue[rear] = value;
        count++;
        return true;
    }

    bool deQueue() {
        if(isEmpty())
            return false;

        queue[front] = -1;
        front = (front+1)%size;
        count--;
        return true;
    }

    int Front() {
        if(!isEmpty())
            return queue[front];
        return -1;
    }

    int Rear() {
        if(!isEmpty())
            return queue[rear];
        return -1;
    }

    bool isEmpty() {
        if(count == 0)
            return true;
        return false;
    }
    bool isFull() {
        if(count == size)
            return true;
        return false;
    }
};
```

Complexity:

- Time: Each of the operations takes O(1) time.
- Space: O(k)
  ‘k’ is the size of the circular queue.

Question 6:

- **Problem link**: https://leetcode.com/problems/basic-calculator-ii/
- **Difficulty level**: Medium

Solution:

```
class Solution {
public:
int calculate(string s) {
int len = s.length();
if(len == 0)
return 0;
stack st;
int currentNumber = 0;
char operation = '+';
   // scan the input from left to right 0 - 9 and add it to the
currentNUmber
    // otherwise the current character must be an operation
    // evaluate the expression based on type of operation
    for(int i = 0; i < len; i++){
        char currentChar = s[i];
        if(isdigit(currentChar)){
            currentNumber = (currentNumber * 10) + (currentChar -
'0');
        }
        if(!isdigit(currentChar) && !iswspace(currentChar)  || i ==
len - 1){
            if(operation == '-'){
                st.push(-currentNumber);
            }else if(operation == '+'){
                st.push(currentNumber);
            }else if(operation == '*'){
                int top = st.top();
                st.pop();
                st.push(top * currentNumber);
            }else if(operation == '/'){
                int top = st.top();
                st.pop();
                st.push(top / currentNumber);
            }
            operation = currentChar;
            currentNumber = 0;
        }
    }

    int answer = 0;
    while(st.size() != 0){
        answer += st.top();
        st.pop();
    }
    return answer;
}

};
```

Complexity:

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

Question 7:

- **Problem link**: https://leetcode.com/problems/132-pattern/
- **Difficulty level**: Medium

Solution:

```
bool find132pattern(vector<int>& nums) {
        int n = nums.size();
        if (n<3) return false;
        stack<int> st;vector<int> smallest(n);
        for (int i =0;i<n;i++){
            smallest[i] = i==0? nums[i] : min(smallest[i-1],nums[i]);
        }
        for (int i =0;i<n;i++){
            while(st.size() && nums[st.top()]<=nums[i]) st.pop();
            if (st.size() && st.top()!=0 &&
smallest[st.top()-1]<nums[i]) return true;
            st.push(i);
        }
        return false;
    }
```

Complexity:

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

Question 8:

- **Problem link**: https://leetcode.com/problems/maximum-frequency-stack/
- **Difficulty level**: Hard

Explanation:

Maintain a map to calculate the frequency of each of the elements. Maintain a vector of stacks ‘stackList’ to hold the actual elements. Also keep track of the max- frequency so far. Whenever we have to push an element, first increment the frequency of that element in the map. Update the max frequency variable. If the element is ‘x’, then, push ‘x’ into ‘stackList[frequency[x]]’. You may have to resize the vector of stacks (stackList) if its size is lesser than the max frequency. For ‘pop’ operation, pop the element at the top of ‘stackList[maxFreq]’ and reduce the frequency of this element by 1 in the map. If this stack becomes empty after current operation, decrement the maxFrequency variable by 1 to point to the previous stack.

Solution:

```
class FreqStack {
public:
    unordered_map<int,int> frequency;
    vector<stack<int>> stackList;
    int maxFreq;

    FreqStack() {
        maxFreq = 0;
    }

    void push(int val) {
        frequency[val]++;
        int currFreq = frequency[val];
        maxFreq = max(maxFreq, currFreq);
        if(stackList.size() < currFreq+1)
            stackList.resize(currFreq+1);

        stackList[currFreq].push(val);
    }

    int pop() {
        if(maxFreq == 0)
            return -1;

        int element = stackList[maxFreq].top();
        stackList[maxFreq].pop();
        frequency[element]--;

        if(stackList[maxFreq].empty())
            maxFreq--;

        return element;
    }
};
```

Complexity:

- Time: Each of the operations takes O(1) time.
- Space: O(n)