PW SKILLS

Lesson Plan

Stacks - 2

PW SKILLS

Today's checklist:

- Check for Balanced Brackets in an expression (well-formedness)
- Given a string S, The task is to remove all the consecutive duplicate characters of the string and return the resultant string.
- Given an array, print the Next Greater Element (NGE) for every element.
- Given an array of distinct elements, find previous greater element for every element. If previous greater element does not exist, print -1.
- Stock span problem
- Area of histogram [Leetcode 84]
- Number of Visible People in a Queue (leetcode 1944)
- Sliding Window Maximum (leetcode 239)
- Min Stack (leetcode 155)

Q. Check for Balanced Brackets in an expression (well-formedness)

Input: exp = "[()]{}{[()()]()}"
Output: Balanced

Explanation: all the brackets are well-formed

Input: exp = "[(])"
Output: Not Balanced

Explanation: 1 and 4 brackets are not balanced because there is a closing ']' before the closing '('

Code:

```
bool areBracketsBalanced(string expr)
{
    // Declare a stack to hold the previous brackets.
    stack<char> temp;
    for (int i = 0; i < expr.length(); i++) {
        if (temp.empty()) {

            // If the stack is empty
            // just push the current bracket
            temp.push(expr[i]);
        }
        else if ((temp.top() == '(' && expr[i] == ')')
                 || (temp.top() == '{' && expr[i] == '}')
                 || (temp.top() == '[' && expr[i] == ']')) {

            // If we found any complete pair of bracket
            // then pop
            temp.pop();
        }
        else {
```

```
            temp.push(expr[i]);
        }
    }
    if (temp.empty()) {

        // If stack is empty return true
        return true;
    }
    return false;
}
```

Q. Given a string S, The task is to remove all the consecutive duplicate characters of the string and return the resultant string.

Input: S= "aaaaabbbbbb"
Output: ab

Code:

```
string deleteConsecutiveStrings(string s)
{
    // Initialize start and stop pointers
    int i = 0;
    int j = 0;

    // Initialize an empty string for new elements
    string newElements = "";

    // Iterate string using j pointer
    while (j < s.length()) {
        // If both elements are same then skip
        if (s[i] == s[j]) {
            j++;
        }
        // If both elements are not same then append new element
        else if (s[j] != s[i] || j == s.length() - 1) {
            newElements += s[i];

            // After appending, slide over the window
            i = j;
            j++;
        }
    }

    // Append the last string
    newElements += s[j - 1];
    return newElements;
}
```

Q. Given an array, print the Next Greater Element (NGE) for every element.

Input: arr[] = [4 , 5 , 2 , 25]
Output: 4 –> 5
5 –> 25
2 –> 25
25 –> -1

Explanation: Except 25 every element has an element greater than them present on the right side

Code:

```
void printNGE(int arr[], int n)
{
    stack<int> s;

    /* push the first element to stack */
    s.push(arr[0]);

    // iterate for rest of the elements
    for (int i = 1; i < n; i++) {

        if (s.empty()) {
            s.push(arr[i]);
            continue;
        }

        /* if stack is not empty, then
           pop an element from stack.
           If the popped element is smaller
           than next, then
        a) print the pair
        b) keep popping while elements are
        smaller and stack is not empty */
        while (s.empty() == false && s.top() < arr[i]) {
            cout << s.top() << " ⟶ " << arr[i] << endl;
            s.pop();
        }

        /* push next to stack so that we can find
        next greater for it */
        s.push(arr[i]);
    }


    while (s.empty() == false) {
        cout << s.top() << " ⟶ " << -1 << endl;
        s.pop();
    }
}
```

Q. Given an array of distinct elements, find previous greater element for every element. If previous greater element does not exist, print -1.

Input : arr[] = {10, 4, 2, 20, 40, 12, 30}
Output : -1, 10, 4, -1, -1, 40, 40

Code:

```
void prevGreater(int arr[], int n)
{
    // Create a stack and push index of first element
    // to it
    stack<int> s;
    s.push(arr[0]);

    // Previous greater for first element is always -1.
    cout << "-1, ";

    // Traverse remaining elements
    for (int i = 1; i < n; i++) {

        // Pop elements from stack while stack is not empty
        // and top of stack is smaller than arr[i]. We
        // always have elements in decreasing order in a
        // stack.
        while (s.empty() == false && s.top() < arr[i])
            s.pop();

        // If stack becomes empty, then no element is greater
        // on left side. Else top of stack is previous
        // greater.
        s.empty() ? cout << "-1, " : cout << s.top() << ", ";

        s.push(arr[i]);
    }
}
```

Q. The stock span problem is a financial problem where we have a series of N daily price quotes for a stock and we need to calculate the span of the stock's price for all N days. The span Si of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

Input: N = 7, price[] = [100 80 60 70 60 75 85]
Output: 1 1 1 2 1 4 6

Explanation: Traversing the given input span for 100 will be 1, 80 is smaller than 100 so the span is 1, 60 is smaller than 80 so the span is 1, 70 is greater than 60 so the span is 2 and so on. Hence the output will be 1 1 1 2 1 4 6.

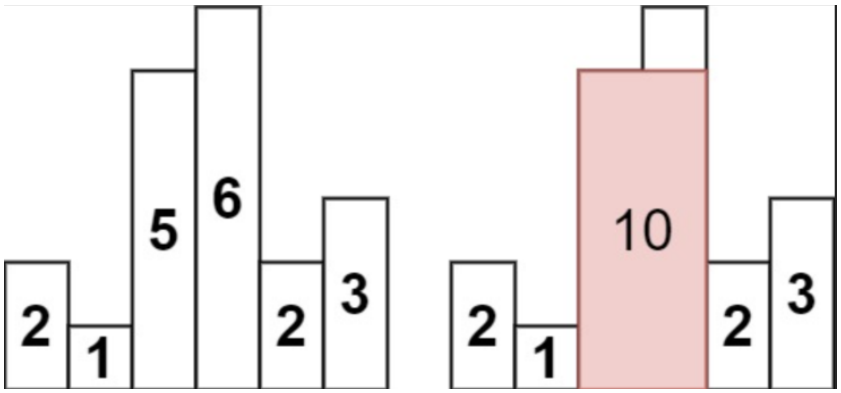
Code:

```
void calculateSpan(int price[], int n, int S[])
{
    // Create a stack and push index of first
    // element to it
    stack<int> st;
    st.push(0);

    // Span value of first element is always 1
    S[0] = 1;

    // Calculate span values for rest of the elements
    for (int i = 1; i < n; i++) {
        // Pop elements from stack while stack is not
        // empty and top of stack is smaller than
        // price[i]
        while (!st.empty() && price[st.top()] ⩽ price[i])
            st.pop();

        // If stack becomes empty, then price[i] is
        // greater than all elements on left of it,
        // i.e., price[0], price[1], ..price[i-1].  Else
        // price[i] is greater than elements after
        // top of stack
        S[i] = (st.empty()) ? (i + 1) : (i - st.top());

        // Push this element to stack
        st.push(i);
    }
}
```

Q. Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.(leetcode 84)

Input: heights = [2,1,5,6,2,3]
Output: 10

Explanation: The above is a histogram where width of each bar is 1.
The largest rectangle is shown in the red area, which has an area = 10 units.



Code:

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        //Method 2 Optimized

        //store the distance of next smaller element in left
        stack<int> s;
        int n=heights.size();
        int left[n];
        for(int i=0;i<n;i++){
            while(!s.empty() && heights[s.top()]>=heights[i])
s.pop();
            if(s.empty()) left[i]=i;
            else left[i]=i-s.top()-1;
            s.push(i);
        }

        //similarly store the distance of next smaller element in
right
        int right[n];
        while(!s.empty()) s.pop();
        for(int i=n-1;i>=0;i--){
            while(!s.empty() && heights[s.top()]>=heights[i])
s.pop();
            if(s.empty()) right[i]=n-1-i;
            else right[i]=s.top()-i-1;
            s.push(i);
        }

        //now we have next and right smaller element
        int ans=-1;
        for(int i=0;i<n;i++){
            int area=(left[i]+right[i]+1)*heights[i];
            ans=max(ans,area);
        }
        return ans;

        //Time -> O(n) , Space -> O(3n)
    }
};
```

Q. Number of Visible People in a Queue (Leetcode 1944)

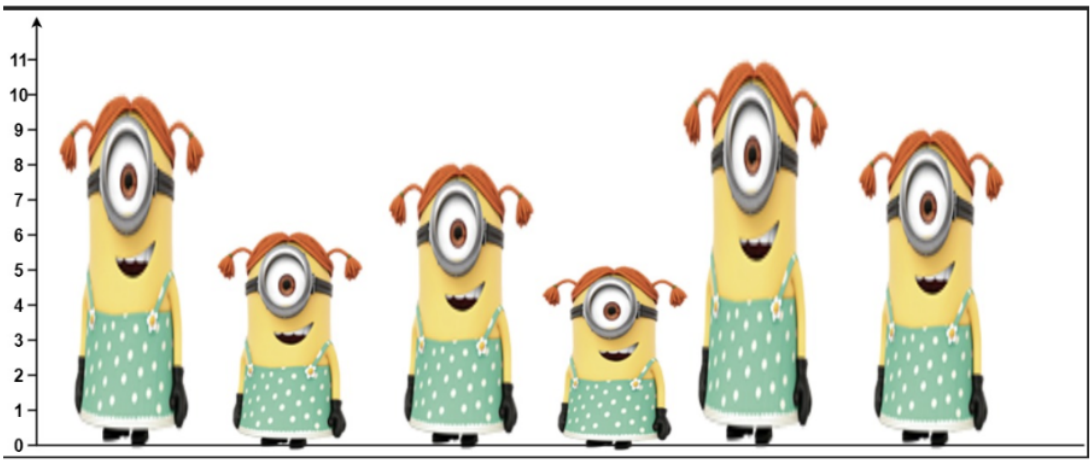
There are n people standing in a queue, and they numbered from 0 to n - 1 in left to right order. You are given an array heights of distinct integers where heights[i] represents the height of the ith person.

A person can see another person to their right in the queue if everybody in between is shorter than both of them. More formally, the ith person can see the jth person if i < j and min(heights[i], heights[j]) > max(heights[i+1], heights[i+2], ..., heights[j-1]).

Return an array answer of length n where answer[i] is the number of people the ith person can see to their right in the queue.

Input: heights = [10,6,8,5,11,9]
Output: [3,1,2,1,1,0]

Explanation:



Person 0 can see person 1, 2, and 4.
Person 1 can see person 2.
Person 2 can see person 3 and 4.
Person 3 can see person 4.
Person 4 can see person 5.
Person 5 can see no one since nobody is to the right of them.

Code:

```
class Solution {
public:
    vector<int> canSeePersonsCount(vector<int>& heights) {
        vector<int> stk, ans;
        for(int i = heights.size() - 1; i ⩾ 0; i--) {
            int cnt = 0;
            while(stk.size() && stk.back() ⩽ heights[i]) {
                ++cnt;
                stk.pop_back();
            }
            ans.push_back(cnt + (stk.size() > 0)); //stk.size() >
0 means the current person can see the first right person who is
higher than him.
            stk.push_back(heights[i]);
        }
        reverse(ans.begin(), ans.end());
        return ans;
    }
};
```

Q. Sliding Window Maximum (Leetcode 239)

You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]

Explanation:

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Example 2:

Input: nums = [1], k = 1
Output: [1]

Code:

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> dq;
        for(int i=0;i<k-1;i++){
            while(!dq.empty() && nums[dq.back()]<=nums[i])
dq.pop_back();
            dq.push_back(i);
        }
        int i=0,j=k-1;
        vector<int> ans(nums.size()+1-k);
        while(j<nums.size()){
            while(!dq.empty() && nums[dq.back()]<=nums[j])
dq.pop_back();
            dq.push_back(j);
            if(dq.front()<i) dq.pop_front();
            ans[i]=nums[dq.front()];
            i++,j++;
        }
        return ans;
    }
};
```

Q. Min Stack (Leetcode 155)

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

MinStack() initializes the stack object.
void push(int val) pushes the element val onto the stack.
void pop() removes the element on the top of the stack.
int top() gets the top element of the stack.
int getMin() retrieves the minimum element in the stack.
You must implement a solution with O(1) time complexity for each function.

Example 1:

Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation:
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top(); // return 0
minStack.getMin(); // return -2

Code:

```
class MinStack {
public:
    stack<long long> st;
    long long mini;
    MinStack() {
        mini=INT_MAX;
    }

    void push(int val) {
        if(st.empty()){
            mini=val;
            st.push(val);
        }
        else{
            if(val<mini){
```

```
                st.push(2*1ll*val-mini);
                // cout<<"pushed: "<<(2*val-mini)<<endl;
                //since val<mini so 2*val<mini+val hence 2*val-
mini<val (val is new mini)
                mini=val;
            }
            else{
                st.push(val);
            }
        }

    }

    void pop() {
        if(st.top()<mini){
            this->mini=2*1ll*mini-st.top();
        }
        st.pop();

    }

    int top() {
        if(st.top()<mini){
            return mini;
        }
        else return st.top();
    }

    int getMin() {
        return mini;
    }
};

/*
 * Your MinStack object will be instantiated and called as such:
 * MinStack* obj = new MinStack();
 * obj->push(val);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->getMin();
 */
```

PW SKILLS

THANK YOU !

PW SKILLS