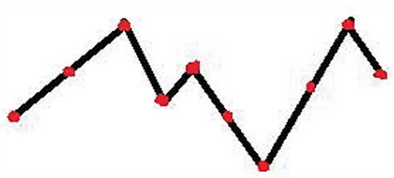
Stack

Approaches

1. Next greater/smaller element to right/left
2. Creating a graph of elements by their value and analyzing based on the top of the mountain value



```
//We are removing top of every mountain created by the trend
while(!s.empty() && s.top() > num[i] && k > 0){
    s.pop();
    k--;
}
```

```
while(!s.empty() && s.top() < nums[i]){
    thirdElement = s.top();
    s.pop();
}
```

Stack using arr

top = -1

empty -> if top = -1

full -> if top = size-1

Queue using arr

f = r = -1

empty -> if f == r

full -> if (rear+1) % size == f

(In circular queue)

-> if rear == size-1

(In normal queue)

3. Problems which need both NGETR and NGETL (Like Trapping) -> Where NGETL of s.top is 2nd element in stack and NGETR of s.top is current element
and NSETL and NSETR (Like Histogram area) -> Where NSETL of s.top is 2nd element in stack and NSETR of s.top is current element

```
int ans = 0;
stack<int> s;
for(int i = 0; i < height.size(); i++){
    while(!s.empty() && height[s.top()] < height[i]){
        int currHt = height[s.top()];
        s.pop();

        if(s.empty()) break;
        int lmax = s.top();
        int rmax = i;

        int width = rmax-lmax-1;
        int ht = min(height[lmax], height[rmax])-currHt;
        ans += ht*width;
    }
    s.push(i);
}
return ans;
```

```
int ans = 0;
for(int i = 0; i <= n; i++){
    int val = (i == n ? 0 : heights[i]);
    while(!s.empty() && heights[s.top()] >= val){
        int height = heights[s.top()];
        s.pop();
        int rb = i; //NSETR of s.top is current idx
        int lb = (!s.empty() ? s.top() : -1); //NSETL of s.top is next element in the stack

        ans = max(ans, height * (rb-lb-1));
    }
    s.push(i);
}
return ans;
```

```cpp
int* NGETR(int *arr, int n){
    stack<int> s;
    int *ans = new int[n];
    ans[n-1] = -1;
    s.push(arr[n-1]);
    for(int i=n-2; i>=0; i--){
        while(!s.empty() && arr[i] >= s.top()){   //* Pop all smaller elements to right(which is stored in stack)
            s.pop();
        }
        if(s.empty())                             //* IF stack is empty, it means there is no element greater than current element
            ans[i] = -1;
        else                                      //* If not empty, it means there is an element greater than current element
            ans[i] = s.top();
        s.push(arr[i]);
    }
    return ans;
}
```

NGETR

```cpp
//Find NGETR of all elements
int ngetr[n];
stack<int> s;
s.push(n-1);  //* In NGETR problem, we used default value as -1, but here we use ARR LENGTH as default value
ngetr[n-1] = n;
for(int i=n-2; i>=0; i--){
    while(!s.empty() && arr[s.top()] <= arr[i]){
        s.pop();
    }
    if(s.empty())
        ngetr[i] = n;
    else
        ngetr[i] = s.top();
    s.push(i);
}
//*Since we are storing the index of NGETR in ngter[] array, we check if the index of NGETR lies in the window(k) then jump
to NGETR and check the same for it, else the current index in the ngetr[] is the max of current window
for(int i=0; i<=n-k; i++){
    int j = i;
    while(ngetr[j] < i+k){
        j = ngetr[j];
    }
    cout<<arr[j]<<endl;
}
```

Sliding Window

Largest area of rectangle in histogram

```cpp
int largestRectangleArea(vector<int>& heights) {
    vector<int> rb = NSETR(heights);
    vector<int> lb = NSETL(heights);

    int ans = 0;
    for(int i = 0; i < heights.size(); i++){
        int width = rb[i] - lb[i] -1;
        int area = width * heights[i];
        ans = max(ans, area);
    }
    return ans;
}
```

Calculating NSETL and NSETR separately

```cpp
//* The apprpach is same but here we are calculating NGETR while calculating NGETL
void NSETL(vector<int>& heights, vector<int>& rb, vector<int>& lb){
    int n = heights.size();
    stack<int> s;

    s.push(0);
    for(int i = 1; i <n; i++){
        while(!s.empty() && heights[s.top()] >= heights[i]){
            rb[s.top()] = i; //* NSETR of s.top is current element
            s.pop();
        }
        if(s.empty()) lb[i] = -1;
        else lb[i] = s.top();

        s.push(i);
    }
}

int largestRectangleArea(vector<int>& heights) {
    int n = heights.size();
    vector<int> rb(n, n) , lb(n, -1);
    NSETL(heights, rb, lb);

    int ans = 0;
    for(int i = 0; i < heights.size(); i++){
        int width = rb[i] - lb[i] -1;
        int area = width * heights[i];
        ans = max(ans, area);
    }
    return ans;
}
```

```cpp
//*In this approach we are finding area of rectangle formed by top element of the stack (Not each element of array as before)
int largestRectangleArea(vector<int>& heights) {
    int n = heights.size();
    stack<int> s;

    int ans = 0;
    for(int i = 0; i <= n; i++){
        int val = (i == n ? 0 : heights[i]);
        while(!s.empty() && heights[s.top()] >= val){
            int height = heights[s.top()];
            s.pop();
            int rb = i; //* NSETR of s.top is current idx
            int lb = (!s.empty() ? s.top() : -1); //* NSETL of s.top is next element in the stack

            ans = max(ans, height * (rb-lb-1));
        }
        s.push(i);
    }
    return ans;
}
```

The point we need to note here is

At any point of time the NSETR of s.top is current elemrnt and NSETL of s.top is 2nd element in stack

```cpp
int minEle;
void pop(stack<int> &S){
    if(S.top() < minEle)
        minEle = 2*minEle - S.top();

    S.pop();
}

void push(stack<int> &S, int x){
    if(S.empty()){
        S.push(x);
        minEle = x;
    }
    else if(x < minEle){
        S.push(2*x - minEle);
        minEle = x;
    }
    else{
        S.push(x);
    }
}

int getMin(stack<int> &S){
    return minEle;
}
int getMin(stack<int> &S){
    return minEle;
}

int main(){
    stack<int> S;
    int choice;
    do{
        int x;
        cin>>x;

        push(S,x);
        cin>>choice;
    }while(choice);

    cout<<getMin(S)<<endl;
    pop(S);
    pop(S);
    cout<<getMin(S)<<endl;
    return 0;
}
```

getMin

Push(x) : Inserts x at the top of stack.

If stack is empty, insert x into the stack and make minEle equal to x.

If stack is not empty, compare x with minEle. Two cases arise:

1. If x is greater than or equal to minEle, simply insert x.
2. If x is less than minEle, insert (2*x – minEle) into the stack and make minEle equal to x.
   For example, let previous minEle was 3. Now we want to insert 2. We update minEle as 2 and insert 2*2 – 3 = 1 into the stack.

Pop() : Removes an element from top of stack.

Remove element from top. Let the removed element be y. Two cases arise:

1. If y is greater than or equal to minEle, the minimum element in the stack is still minEle.
2. If y is less than minEle, the minimum element now becomes (2*minEle – y), so update (minEle = 2*minEle – y).

This is where we retrieve previous minimum from current minimum and its value in stack.

For example, let the element to be removed be 1 and minEle be 2. We remove 1 and update minEle as 2*2 – 1 = 3.

How it works

In push : IN here we check if the new value is lesser then minELe, if so, we modify the value and push to stack

x < minEle
x-minEle < 0
2x-minEle < 0 + x
2x-minEle < x

ie modified val < newMinValue --> Here the modified value is lesser than the min value. **Therefore modified value is the only value that is less than minVal (which is the identity of modified value)**

In pop: In here we check if the top is a modified value, if so we get the prev min by (top < minEle)

we assign minEle = 2*minEle-top
2*minELe-mdified
2*minEle-2x-prevMinEle
2*minEle-2*minEle-prevMinEle
prevMinEle

Therefore we get the prev min value