LIFO

st. push(10)

st.push(15)

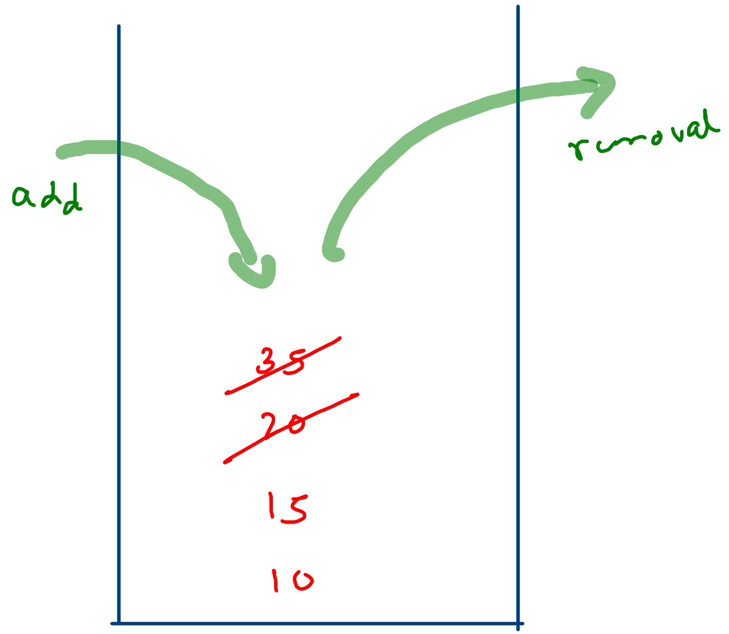
st. pus (20)

st. peek( ) -> 20

st.push(35)

st.pop( ) -> 35

st.pop( ) -> 20

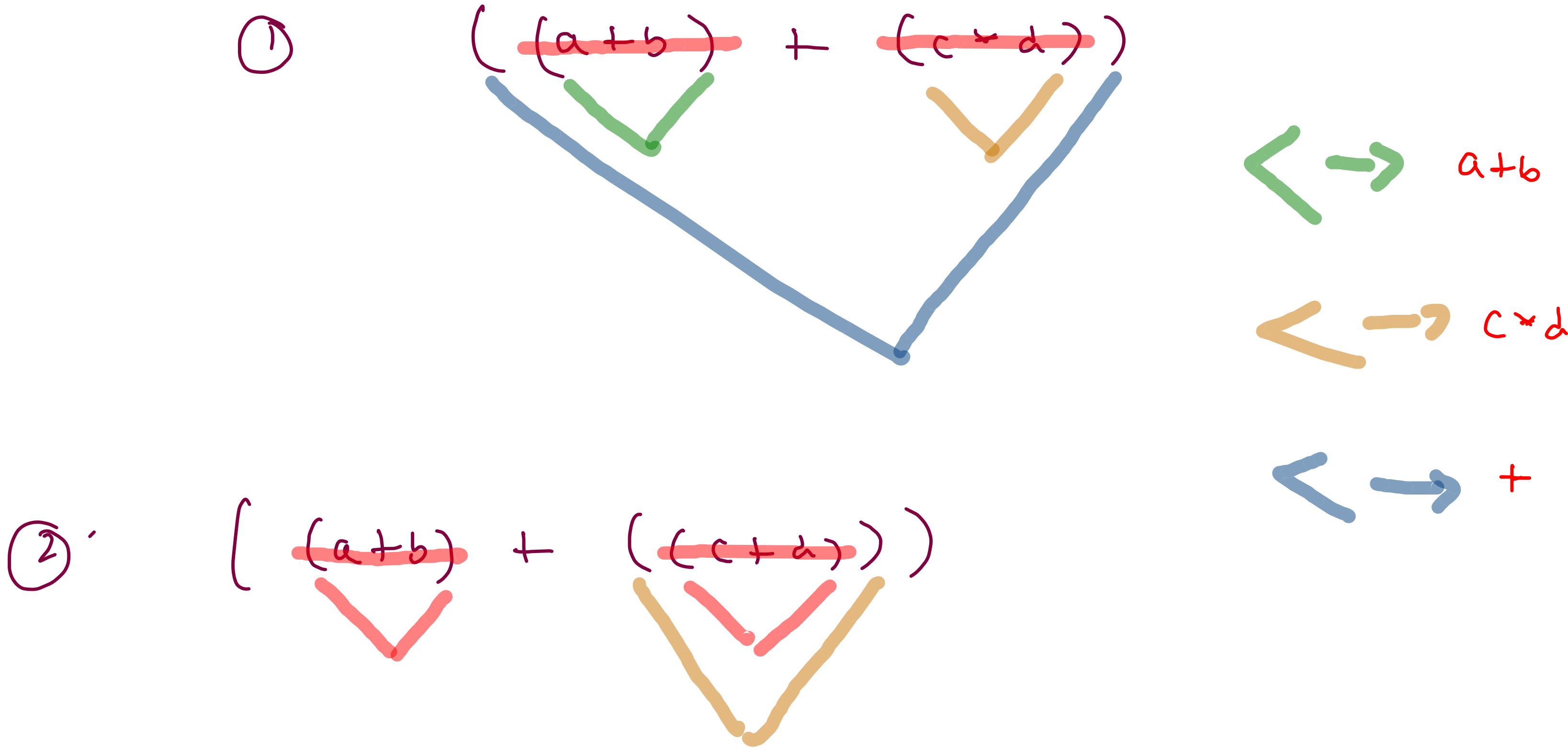


add -> push

remove -> pop

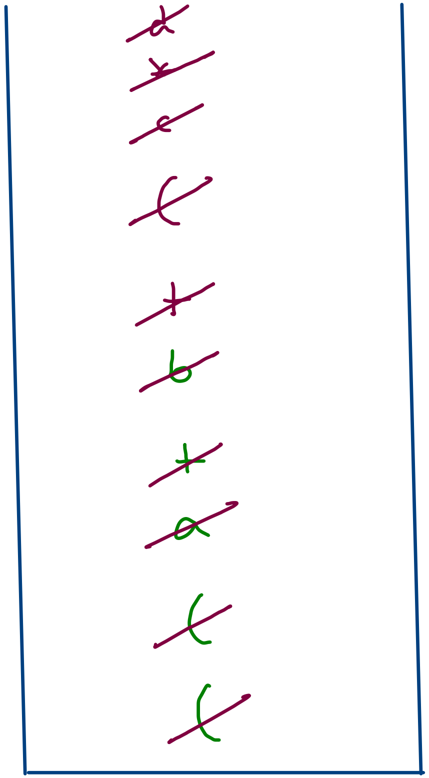
get -> peek

Duplicate brackets / redundant brackets



$((a+b) \; + (c*d))$

pc = 1

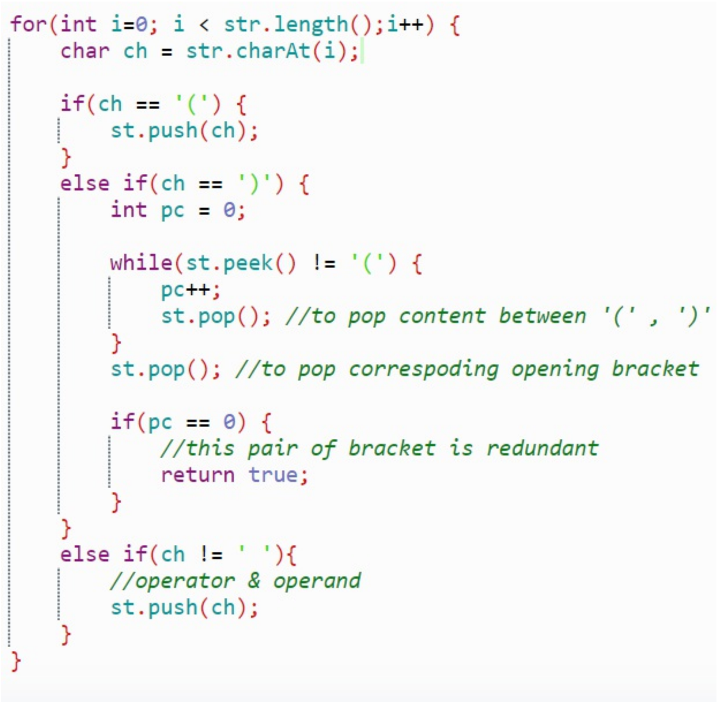


'(' || content

-> st.push.

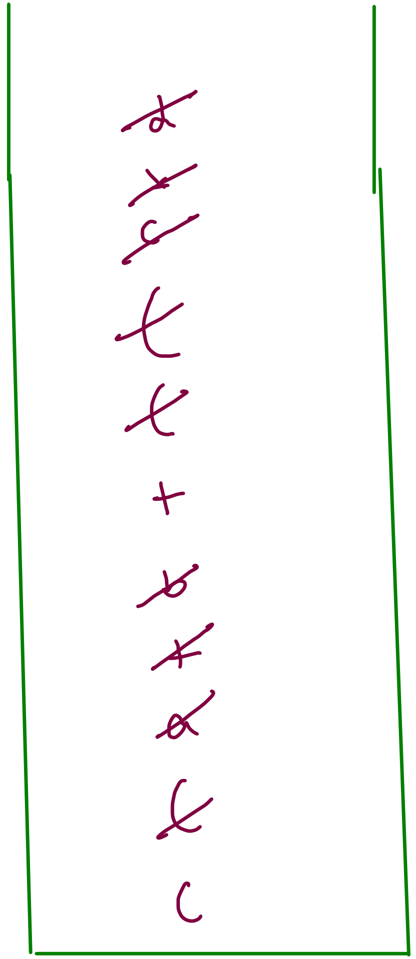
')' -> settle out

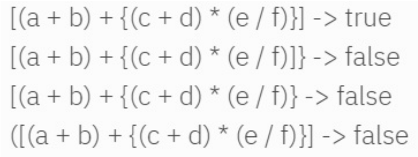
((a+b) + ((c* d)))



```
for(int i=0; i < str.length();i++) {
    char ch = str.charAt(i);

    if(ch == '(') {
        st.push(ch);
    }
    else if(ch == ')') {
        int pc = 0;

        while(st.peek() != '(') {
            pc++;
            st.pop(); //to pop content between '(' , ')'
        }
        st.pop(); //to pop correspoding opening bracket

        if(pc == 0) {
            //this pair of bracket is redundant
            return true;
        }
    }
    else if(ch != ' '){
        //operator & operand
        st.push(ch);
    }
}
```

pc = 0





[(a + b) + {(c + d) * (e / f)}] -> true
[(a + b) + {(c + d) * (e / f)]} -> false
[(a + b) + {(c + d) * (e / f)} -> false
([(a + b) + {(c + d) * (e / f)}] -> false

( ), { }, [ ]

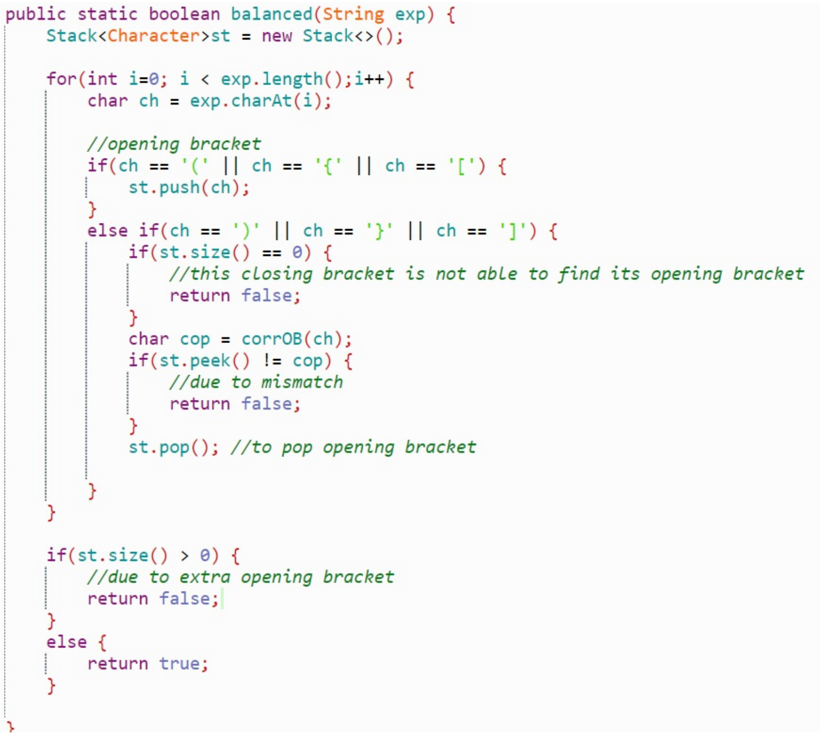
① (a+b} -> X (different type)

② (a+b)) -> X (extra closing bracket)

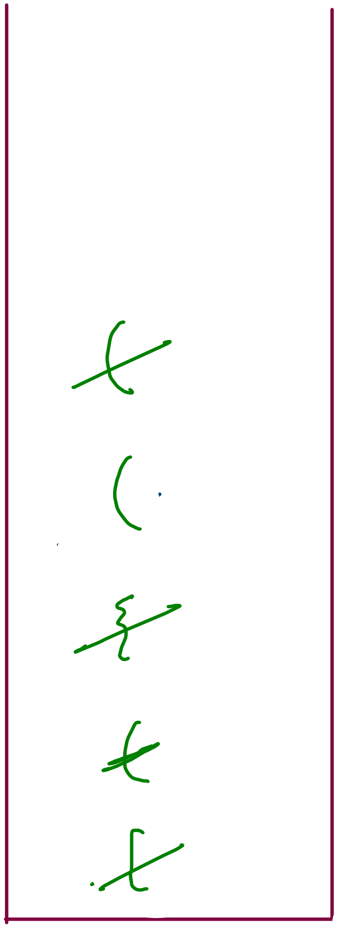
③ ((a+b) -> X (extra opening bracket)

④)(a+b)(

[(a+b) * {c - d}] + ((x + y).

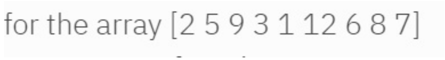


```
public static boolean balanced(String exp) {
    Stack<Character>st = new Stack<>();

    for(int i=0; i < exp.length();i++) {
        char ch = exp.charAt(i);

        //opening bracket
        if(ch == '(' || ch == '{' || ch == '[') {
            st.push(ch);
        }
        else if(ch == ')' || ch == '}' || ch == ']') {
            if(st.size() == 0) {
                //this closing bracket is not able to find its opening bracket
                return false;
            }
            char cop = corrOB(ch);
            if(st.peek() != cop) {
                //due to mismatch
                return false;
            }
            st.pop(); //to pop opening bracket

        }
    }

    if(st.size() > 0) {
        //due to extra opening bracket
        return false;
    }
    else {
        return true;
    }

}
```



(, [, { -> St.push

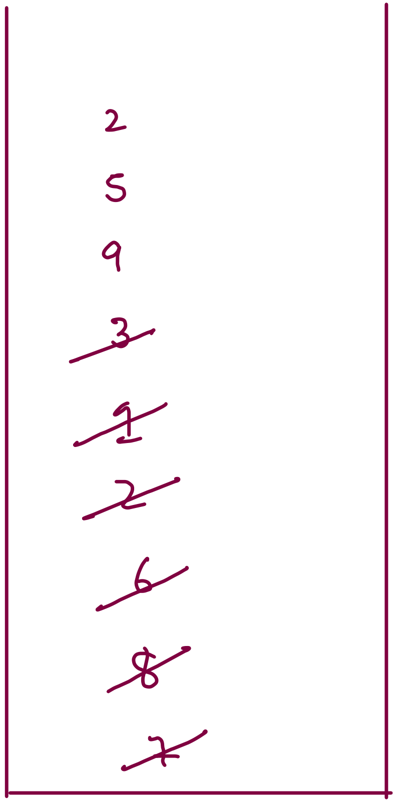
),], } -> validate



for the array [2 5 9 3 1 12 6 8 7]

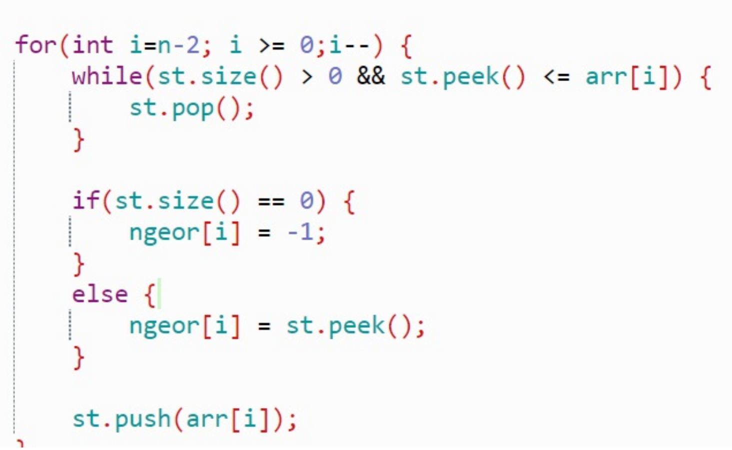
next greater on right

	2	5	9	3	1	2	6	8	7
	0	1	2	3	4	5	6	7	8
ngeor	5	9	-1	6	2	6	8	-1	-1

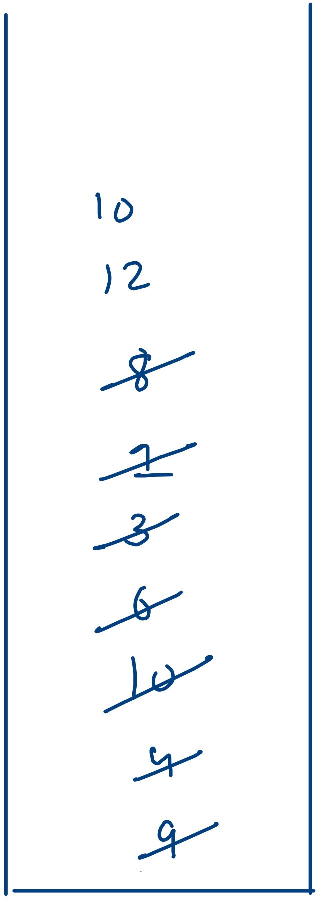


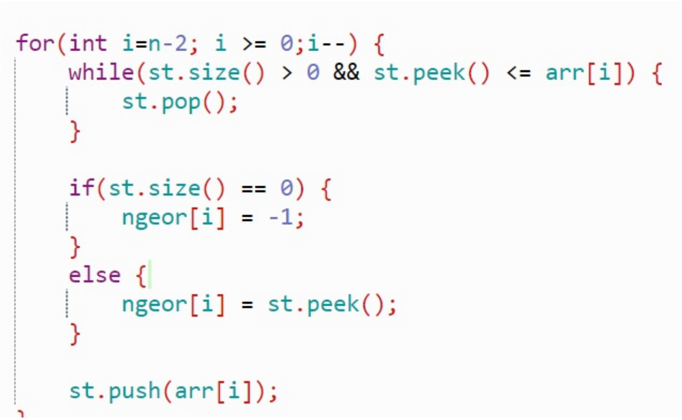
O(n)

	10	12	8	1	3	6	10	4	9
	0	1	2	3	4	5	6	7	8
ngeor	12	-1	10	3	6	10	-1	9	-1

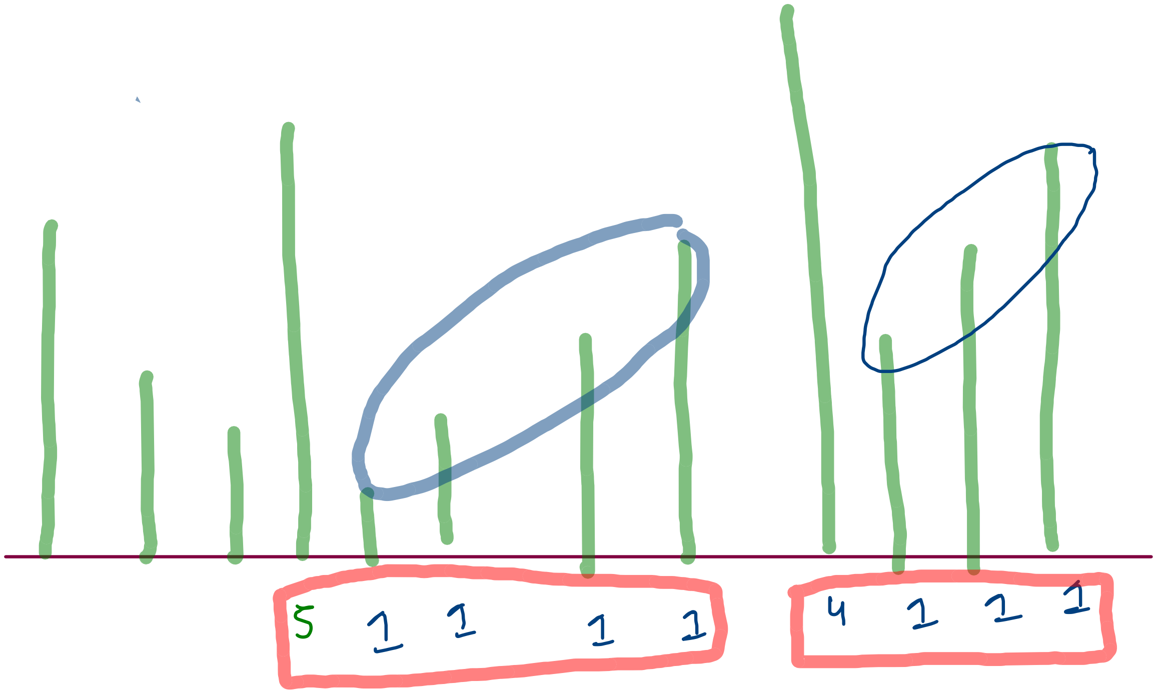


```
for(int i=n-2; i >= 0;i--) {
    while(st.size() > 0 && st.peek() <= arr[i]) {
        st.pop();
    }

    if(st.size() == 0) {
        ngeor[i] = -1;
    }
    else {
        ngeor[i] = st.peek();
    }

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



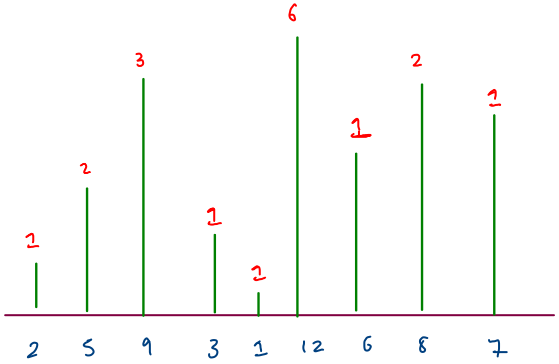


```
for(int i=n-2; i >= 0;i--) {
    while(st.size() > 0 && st.peek() <= arr[i]) {
        st.pop();
    }

    if(st.size() == 0) {
        ngeor[i] = -1;
    }
    else {
        ngeor[i] = st.peek();
    }

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



O(n)

[2 5 9 3 1 12 6 8 7]



	2	5	9	3	1	12	6	8	7
ngeol	-1	-1	-1	9	3	-1	12	12	8

7

8

~~6~~

12

~~1~~

~~3~~

~~9~~

~~5~~

~~2~~

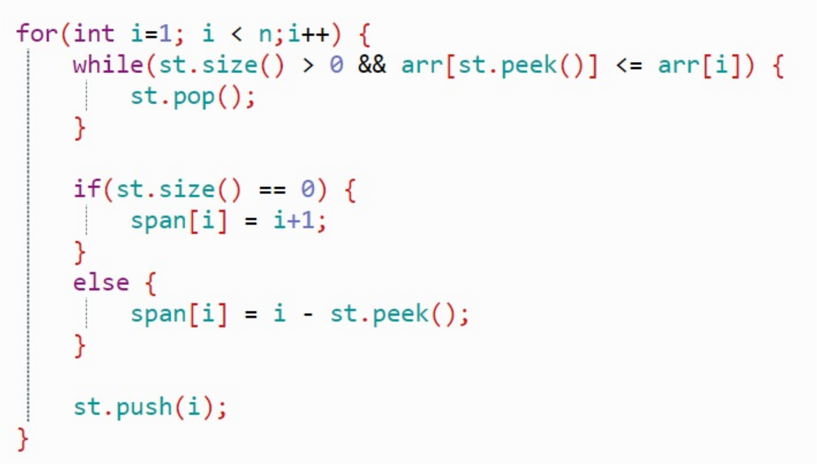
Value

based

ngeol

idx based

	0	1	2	3	4	5	6	7	8
	2	5	9	3	12	1	6	8	7
span	1	2	3	1	5	1	2	3	1



```
for(int i=1; i < n;i++) {
    while(st.size() > 0 && arr[st.peek()] <= arr[i]) {
        st.pop();
    }

    if(st.size() == 0) {
        span[i] = i+1;
    }
    else {
        span[i] = i - st.peek();
    }

    st.push(i);
}
```

8

7

~~6~~

~~5~~

4

~~3~~

~~2~~

~~1~~

~~0~~

idx

1. nger value based
2. nger idx based
3. ngel value based
4. ngel idx based

1. nser value based
2. nser idx based
3. nsel value based
4. nsel idx based

greater	smaller
pop smaller elements	pop larger elements

left	right
left to right	right to left