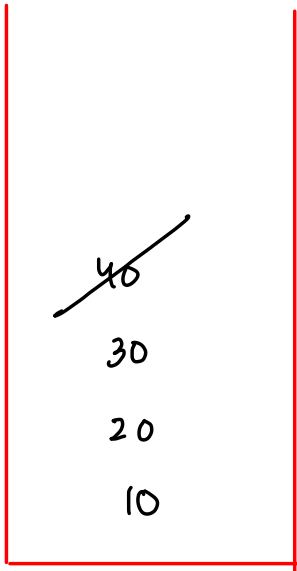


Stack

FIFO: First in last out

$\begin{bmatrix} O(1) \\ \text{access} \end{bmatrix} \rightarrow \text{top most element}$



st. push(10)

st. push(20)

st. push(30)

st. push(40)

st. peek() \rightarrow 40

st. pop() \rightarrow 40

st. peek() \rightarrow 30

push

pop

peek

Duplicate Brackets

The diagram shows the expression $((a+b) + (c+d))$. Red lines connect the opening and closing parentheses of $(a+b)$ to the text $a+b$ below. Similarly, purple lines connect the opening and closing parentheses of $(c+d)$ to the text $c+d$ below. A large black bracket underneath the entire expression points to a plus sign at the bottom, indicating the final simplified form.

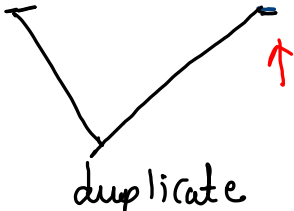
no duplicate
brackets

The diagram shows the expression $((a+b) + ((c+d)))$. Red lines connect the opening and closing parentheses of $(a+b)$ to the text $a+b$ below. Similarly, red lines connect the opening and closing parentheses of $((c+d))$ to the text $c+d$ below. A black line connects the two opening parentheses at the top, with the word "redundant" written above it, indicating that the outer parentheses are unnecessary.

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

i

~~e~~
~~+~~
~~d~~
~~(~~
~~*~~
~~e~~
~~(~~
~~+~~
~~b~~
~~+~~
~~a~~
~~(~~
~~(~~

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

 duplicate
 $c \geq 0$

~~d~~
~~-~~
~~c~~
~~(~~
 (
 *
 b
 +
 a
 (

if (ch == ')') {
 // settle
 }
 else if (ch != ' '){
 {
 st.push(ch);
 }
 (, operators,
 operands

$$((a+b) * ((c-d) + y))$$

```

for(int i=0; i < exp.length();i++) {
    char ch = exp.charAt(i);
    if(ch == ')') {
        //settle
        int count = 0;

        while(st.peek() != '(') {
            count++;
            st.pop();
        }

        //this pair of bracket is redundant
        if(count == 0) {
            return true;
        }

        st.pop(); //for '(' bracket
    }
    else if(ch != ' ') {
        //'(', operand, operators
        st.push(ch);
    }
}

```

$c = 0$

i

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

$O(n)$

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

duplicate

$$c \neq 0$$

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

    if(ch == ')') {
        //settle
        int count = 0;

        while(st.peek() != '(') {
            count++;
            st.pop();
        }

        //this pair of bracket is redundant
        if(count == 0) {
            return true;
        }

        st.pop(); //for '(' bracket
    }
    else if(ch != ' ') {
        //'(', operand, operators
        st.push(ch);
    }
}
```

~~d~~
~~e~~
~~f~~
(
*
~~b~~
*
~~a~~
)

Balanced Brackets

$\{ (a+b) + [c+d] \}$

(i) mismatch $(a+b\}$

(ii) no. of op == no. of closing bracket $((a+b)$
or
 $(a+b))$

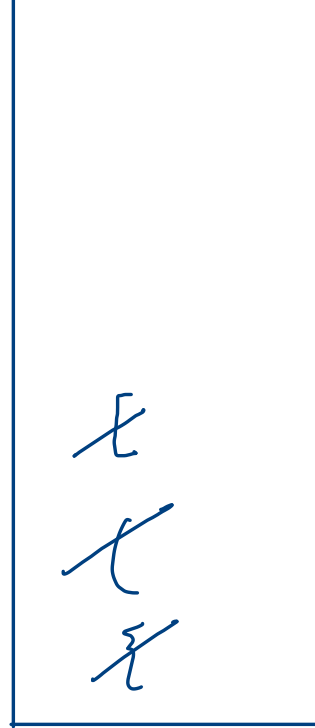
(iii) pairing (closing bracket should have

a same type opening and vice-versa
should be also true).

$\{ (a+b) + [c+d] \};$

```
for(int i=0; i < exp.length(); i++) {
    char ch = exp.charAt(i);
    if(ch == '(' || ch == '[' || ch == '{') {
        st.push(ch);
    }
    else if(ch == ')' || ch == ']' || ch == '}') {
        //validate
        if(ch == ')') {
            if(st.size() == 0 || st.peek() != '(') {
                return false;
            }
        }
        else if(ch == ']') {
            if(st.size() == 0 || st.peek() != '[') {
                return false;
            }
        }
        else if(ch == '}') {
            if(st.size() == 0 || st.peek() != '{') {
                return false;
            }
        }

        st.pop(); //to pop opening bracket
    }
}
```



opening \rightarrow st.push

closing \rightarrow validate

$((a+b] + [c+d])$

mismatch

↑

ch =]

```
for(int i=0; i < exp.length();i++ ) {
    char ch = exp.charAt(i);
    |
    if(ch == '(' || ch == '[' || ch == '{') {
        st.push(ch);
    }
    else if(ch == ')' || ch == ']' || ch == '}') {
        //validate
        if(ch == ')') {
            if(st.size() == 0 || st.peek() != '(') {
                return false;
            }
        }
        else if(ch == ']') {
            if(st.size() == 0 || st.peek() != '[') {
                return false;
            }
        }
        else if(ch == '}') {
            if(st.size() == 0 || st.peek() != '{') {
                return false;
            }
        }

        st.pop(); //to pop opening bracket
    }
}
```

(

(

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

```
for(int i=0; i < exp.length();i++ ) {
    char ch = exp.charAt(i);
    if(ch == '(' || ch == '[' || ch == '{') {
        st.push(ch);
    }
    else if(ch == ')' || ch == ']' || ch == '}') {
        //validate
        if(ch == ')') {
            if(st.size() == 0 || st.peek() != '(') {
                return false;
            }
        }
        else if(ch == ']') {
            if(st.size() == 0 || st.peek() != '[') {
                return false;
            }
        }
        else if(ch == '}') {
            if(st.size() == 0 || st.peek() != '{') {
                return false;
            }
        }

        st.pop(); //to pop opening bracket
    }
}
```

~~(~~
~~(~~
(

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

↑

```
for(int i=0; i < exp.length();i++ ) {
    char ch = exp.charAt(i);
    if(ch == '(' || ch == '[' || ch == '{') {
        st.push(ch);
    }
    else if(ch == ')' || ch == ']' || ch == '}') {
        //validate
        if(ch == ')') {
            if(st.size() == 0 || st.peek() != '(') {
                return false;
            }
        }
        else if(ch == ']') {
            if(st.size() == 0 || st.peek() != '[') {
                return false;
            }
        }
        else if(ch == '}') {
            if(st.size() == 0 || st.peek() != '{') {
                return false;
            }
        }

        st.pop(); //to pop opening bracket
    }
}
```

~~/~~

~~/~~

$[(a+b) + \{ (c+d) * (e+j) \}])$

↑

```
for(int i=0; i < exp.length(); i++) {  
    char ch = exp.charAt(i);  
    if(ch == '(' || ch == '[' || ch == '{') {  
        st.push(ch);  
    }  
    else if(ch == ')' || ch == ']' || ch == '}') {  
        //validate  
        if(ch == ')') {  
            if(st.size() == 0 || st.peek() != '(') {  
                return false;  
            }  
        }  
        else if(ch == ']') {  
            if(st.size() == 0 || st.peek() != '[') {  
                return false;  
            }  
        }  
        else if(ch == '}') {  
            if(st.size() == 0 || st.peek() != '{') {  
                return false;  
            }  
        }  
        st.pop(); //to pop opening bracket  
    }  
}
```

~~(~~
~~)~~
~~{~~
~~}~~
~~[~~
~~]~~

extra closing
brackets.

Next Greater Element To The Right

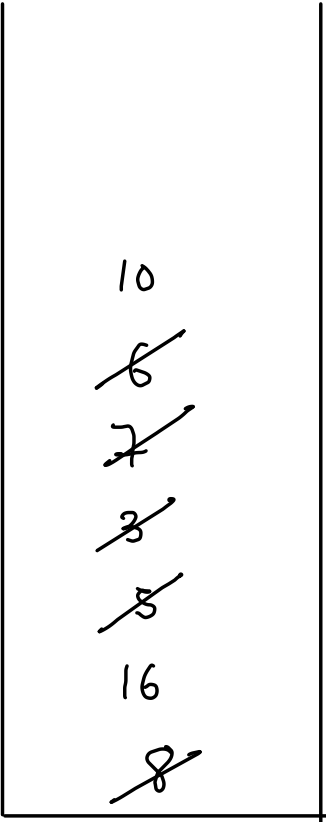
$O(n)$



10	6	7	3	5	16	8
0	1	2	3	4	5	6

nge

16 7 16 5 16 -1 -1



↓

10	6	9	17	5	4	11
0	1	2	3	4	5	6

nge

17 9 17 -1 11 11 -1

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

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

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

10
~~6~~
~~9~~
 17
~~5~~
~~4~~
~~11~~

```

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

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

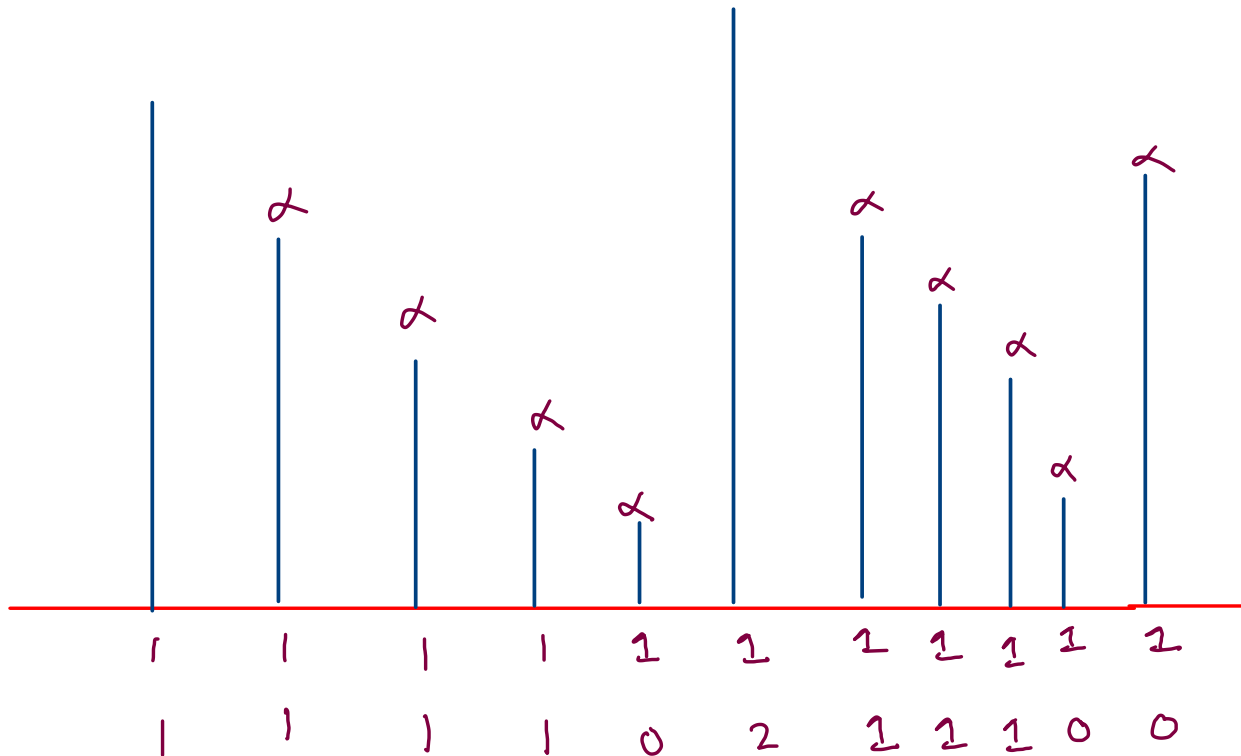
    st.push(arr[i]);
}

```

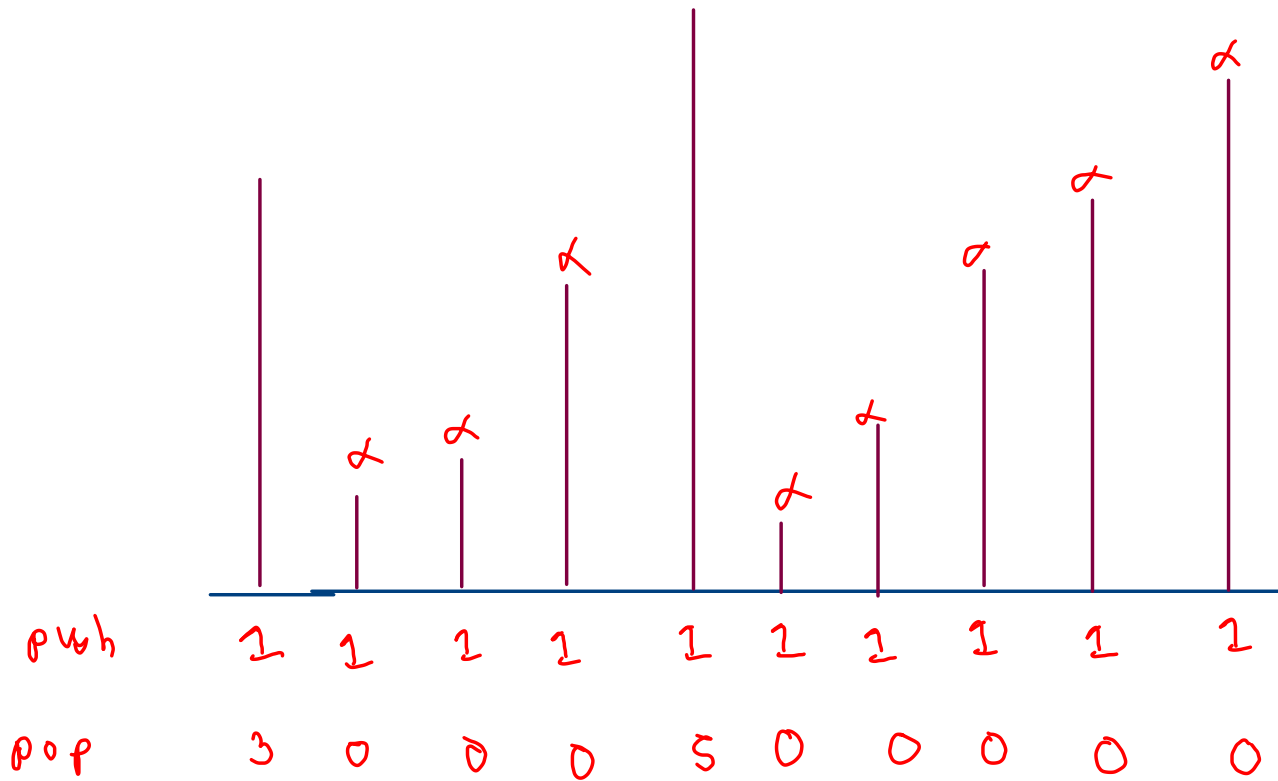
$O(n)$

push

pop



$n \text{ push} + n \text{ pop}$



n pop
n pwh

ngeor

right to left

ngeol

left to right

ngeor

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

nseor

```
while (st.size() > 0 &&  
    st.peek() >= arr[i]) {  
    st.pop();  
}
```

ngeor (index-based)

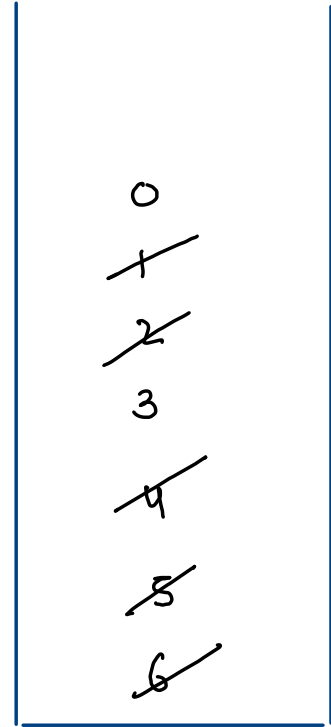


10	6	9	17	5	4	11
0	1	2	3	4	5	6

ngeori 3 2 3 -1 6 6 -1

dist 3 1 1 -1 2 1 -1

arr [st.peek()]



Stack \rightarrow index



10	6	9	17	5	4	11
0	1	2	3	4	5	6

nge

3 1 1 -1 2 1 -1

→ gap between me (ith ele) and nge on right for ith ele.

0
~~1~~
~~2~~
3
~~4~~
~~5~~
~~6~~

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

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

    st.push(i);
}
```