Lesson:

Stacks in JAVA

Pre Requisites:

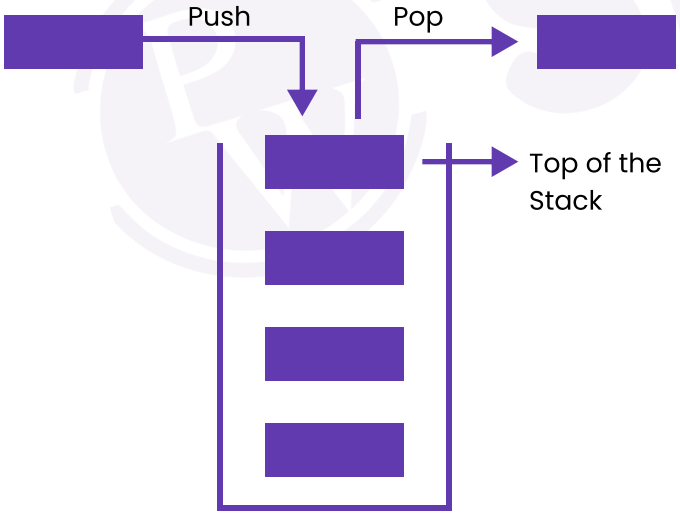
- Basic java syntax

List of concepts involved :

- Introduction to stack
- Operations in stack
- Stack class in java
- Implementation of various functions of stack class
- Valid parentheses
- Largest rectangle in histogram

Introduction to stack

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first.
There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out). Another example could be a stack of books, there also it can be seen that the book placed last is at the top and will be removed first.



Different operations/terminologies associated with a stack are as follows :

- **Push :** Adding an element to a stack is known as a push operation. Push operation is not possible if the stack is full.

- **Pop:** Removing an element from a stack is known as pop operation and is not possible if the stack is empty.

- **Peek :** Viewing the top most element present in a stack is known as peek. Its functionality is similar to pop, the only difference is that in pop operation we remove the top most element but in peek we just return the top most element's value and we do not remove it. This operation is also not feasible if the stack is empty.

Stack class in JAVA :

Java Collection framework provides a Stack class that models and implements a Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search, and peek.

Syntax of defining a stack :
In order to create a stack, we must import the "java.util.Stack" package and use the Stack() constructor of this class. The below example creates an empty Stack.
Stack<datatype> stack_name = new Stack<>();

Q1. write a program to implement various functionalities of a stack, including push, pop and peek.

Solution :

Code : LP_Code1.java

Output :

```
The peek element of the stack is : 1
The peek element of the stack is : 5
The peek element of the stack is : 3
The stack elements are as follows : [1, 3]
```

Approach :
The stack looks as follows after first two the push operations have been performed:

| 2 |
|_1_|

Here the peek element is 2.
Then we popped the element.
Hence stack looks like :

|_1_|

Now we inserted the elements 3, 5
| 5 |
| 3 |
|_1_|
Again we popped the element.
Hence stack looks like :
| |
| |
| 3 |
|_1_|

Now the peek element is 3.

At last we have simply printed the elements present within the stack and they are 1 and 3.

Q2. Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

Open brackets must be closed by the same type of brackets.
Open brackets must be closed in the correct order.
Every close bracket has a corresponding open bracket of the same type.

Example 1:
Input: s = "()"
Output: true

Example 2:
Input: s = "()[]{}"
Output: true

Example 3:
Input: s = "(]"
Output: false

Example 4:
Input: s = "({})"
Output: true

Solution :

Code : LP_Code2.java

Output :

```
Enter the string: [[{}]()
The given string is valid ? false
```

Approach :

- We have to keep some data structure which will tell us, whether its opening term comes or not.
- Means suppose we met parentheses '}', but parentheses do not come before already '{', then it can never be valid.
- And the data structure we are going to use is stack.
- And why STACK? Because it will count as the last element that comes in.
- Stack works on the principle of LIFO i.e Last In First Out.
- Let's dry run it in the above example.
- Suppose our string given to us as, s = "(({}))"

Initially, our stack is empty, and looks like, t = [],

s = "(({ }))"
↑
we met an opening bracket, we will say there is a probability of meeting same closing bracket in future, so push into stack,
now stack looks like, t = ['(']
↑ t.top()

s = "(({ }))"
↑
we met an opening bracket, we will say there is a probability of meeting same closing bracket in future, so push into stack,
now stack looks like, t = ['(', '(']
↑ t.top()

s = "(({ }))"
↑
we met an opening bracket, we will say there is a probability of meeting same closing bracket in future, so push into stack,
now stack looks like, t = ['(', '(', '{']
↑ t.top()

s = "(({ }))"
↑
Now, we met an closing bracket, so we say our parentheses will valid only if it encounter the last same opening parentheses of this type, and how we encounter our last opening parentheses, for that we are using stack
t = ['(', '(', '{']
↑ t.top()

so, yes our current bracket is '}' and top of stack is '{', so they are valid,
therefore move forward and pop the peek/top element from stack,
t = ['(', '(',]
↑ t.top()
so, yes our current bracket is ')' and top of stack is '(', so they are valid,
therefore move forward and pop the top from stack,
t = ['(']

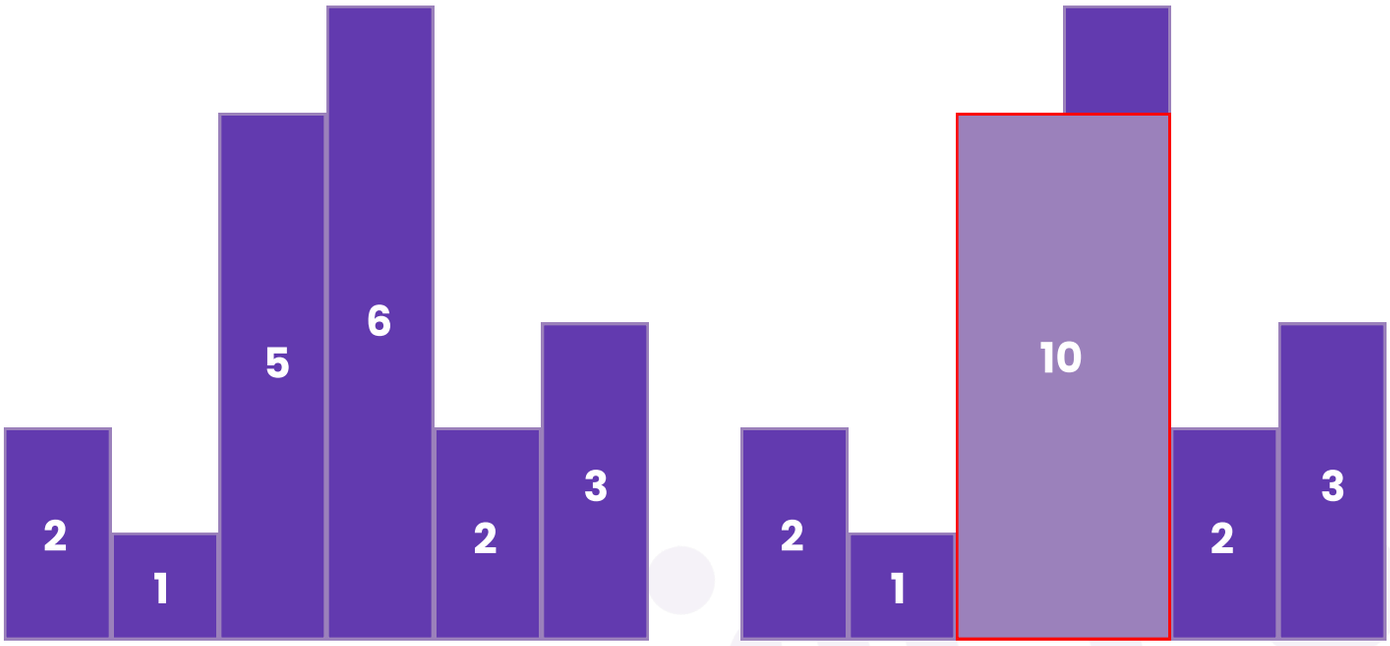
s = "(({ }))"
↑
Now, we met an closing bracket, so we say our parentheses will valid only if it encounter the last same opening parentheses of this type, and how we encounter our last opening parentheses, for that we are using stack
our current stack looks like, t = ['(']
↑ t.top()
so, yes our current bracket is ')' and top of stack is '(', so they are valid,
therefore move forward and pop the top from stack,
t = [ ]

Since, we are on the last index and our stack is empty,
therefore it is valid.

If the stack is not found empty then we have concluded that for some opening bracket, there does not exist a valid closing parentheses so the given string is invalid.

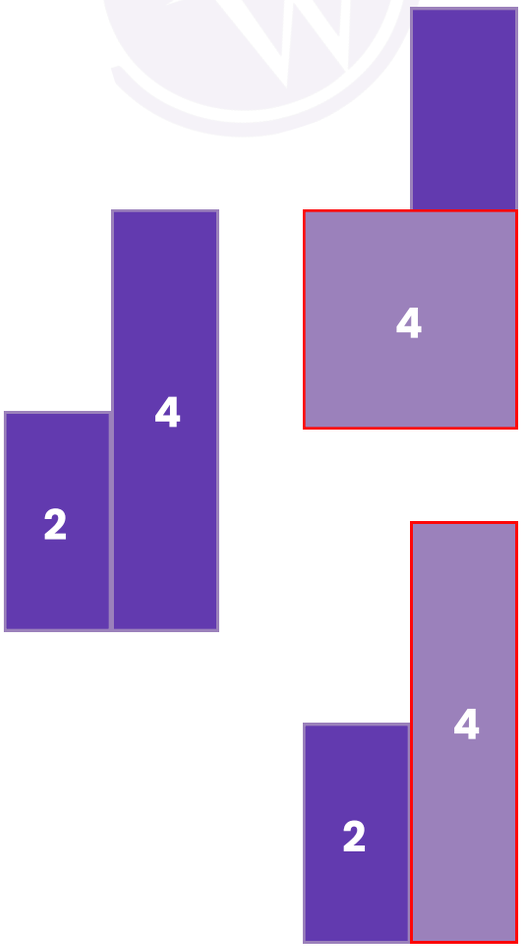
Q3. Given an array of integer heights representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.

Example 1:



Input: heights = [2,1,5,6,2,3]
Output: 10
Explanation: The above is a histogram where the width of each bar is 1.
The largest rectangle is shown in the red area, which has an area = 10 units.

Example 2:



Input: heights = [2,4]
Output: 4

Solution :

Code : LP_Code3.java

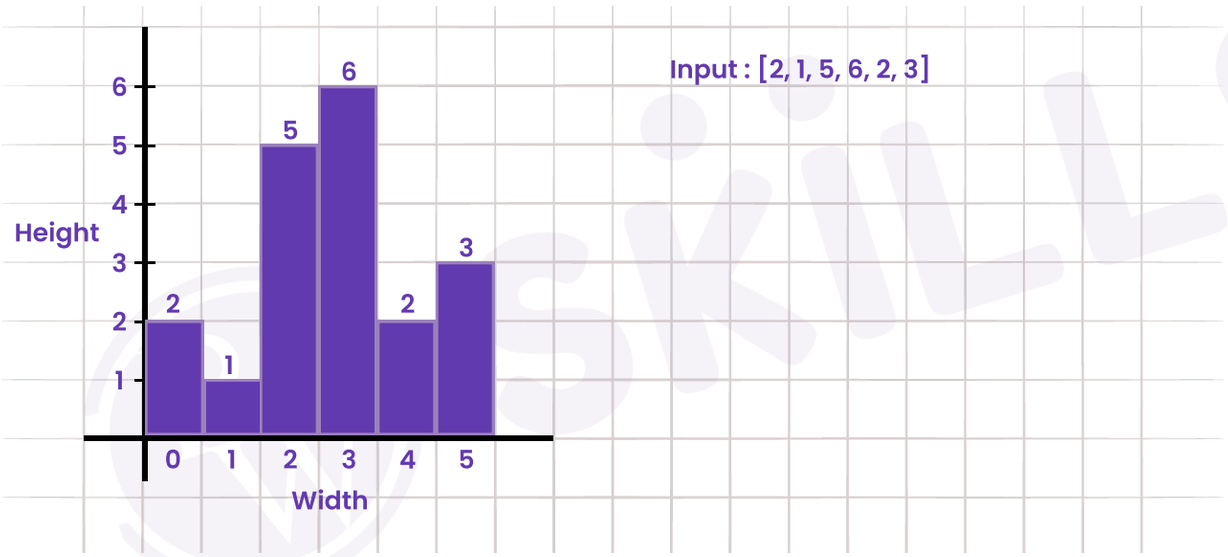
Output :

```
The maximum area is : 10
```

Approach :

The main concept here is that we need to find the minimum element to the right and to the left for any particular element so that we can use this difference as the width of the rectangle.

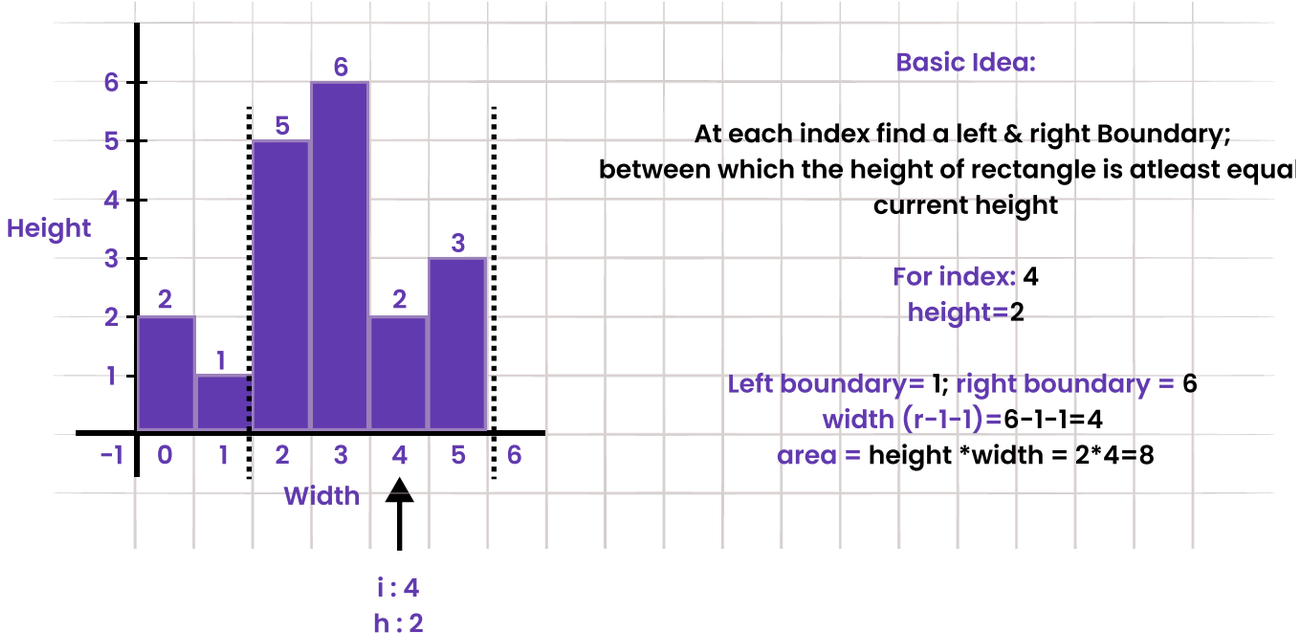
- Let's understand this problem with an example;
- Input: heights = [2,1,5,6,2,3]



- Let's understand how we calculate this, let's take index 4 height is 2. If we go to the left and see where the height is less then 2 and i.e. at index 1. So, that becomes the left Boundary. Similarly if we go to the right and see where the height is less then 2 & i.e. at index 6 [which is end of the array].

- Now we have left & right boundary the area could be find out by height * width. And width will get by (right - left - 1) in case of 4 that will be 6 - 1 - 1 = 4. And we know the height already i.e. 2.

- The Area we get is 2 * 4 i.e. 8



- The main idea here would be how to find the left boundary & right boundary for every index. The brute way is using an ARRAY
- **For left boundary array:**

- Start with index 1, (left[0] = -1)
- For each index -> go to left & find the nearest index where height [index] < height[curr]
- Start with index 1,
- Eg : Index 1 : left[1] = -1

- **For right boundary array:**

- Start with index n - 2 right[n - 1] = n
- For each index -> go to right & find the nearest index where height[index] < height[curr]
- Start with index n - 2
- Eg : Index 2 : right[2] = 4