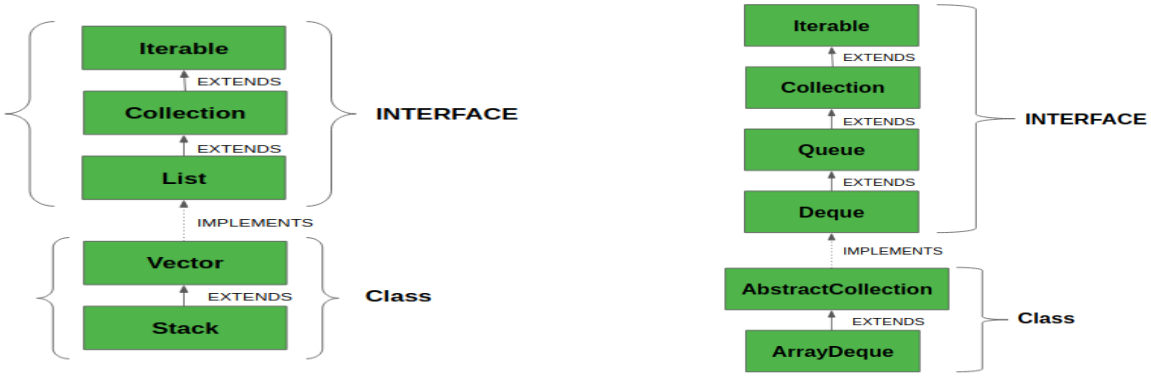
Stack in Java

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. The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions. The class can also be referred to as the subclass of Vector.



NOTE: Please note that the Stack class in Java is a legacy class and inherits from Vector in Java. It is a thread-safe class and hence involves overhead when we do not need thread safety. It is recommended to use **ArrayDeque** for stack implementation as it **is more efficient in a single-threaded environment**.

Example 1:

```
class GFG {
    public static void main (String[] args) {
        // Stack<Integer> stack = new Stack<>();
        ArrayDeque<Integer> stack = new ArrayDeque<>();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        System.out.println(stack.peek());
        System.out.println(stack.pop());
        System.out.println(stack.peek());
    }
}
```

Output:

```
30
30
20
```

```
/* A Java Program to show implementation of Stack
using ArrayDeque */
    public static void main (String[] args) {
        //     Stack<Integer> stack
        //  = new Stack<>();
        ArrayDeque<Integer> stack = new ArrayDeque<>();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        System.out.println(stack.size());
        System.out.println(stack.isEmpty());
    }
}
```

Output:

```
3
False
```

Stack Functions in Java

Method	Description	Time Complexity
push(Object element)	Pushes an element on the top of the stack.	O(1)

pop()	Removes and returns the top element of the stack. An 'EmptyStackException' exception is thrown if we call pop() when the invoking stack is empty.	O(1)
peek()	Returns the element on the top of the stack, but does not remove it.	O(1)
isEmpty()	It returns true if nothing is on the top of the stack. Else, returns false.	O(1)
size()	This method is used to get the size of the Stack or the number of elements present in the Stack.	O(1)

Applications, Advantages and Disadvantages of Stack

Stack is a simple linear data structure used for storing data. Stack follows the LIFO(Last In First Out) strategy that states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first. It can be implemented through an array or linked lists. Some of its main operations are: **push(), pop(), top(), isEmpty(), size(), etc.** In order to make manipulations in a stack, there are certain operations provided to us. When we want to insert an element into the stack the operation is known as the push operation whereas when we want to remove an element from the stack the operation is known as the pop operation. If we try to pop from an empty stack then it is known as underflow and if we try to push an element in a stack that is already full, then it is known as overflow.

Primary Stack Operations:

- void push(int data): When this operation is performed, an element is inserted into the stack.
- int pop(): When this operation is performed, an element is removed from the top of the stack and is returned.

Auxiliary Stack Operations:

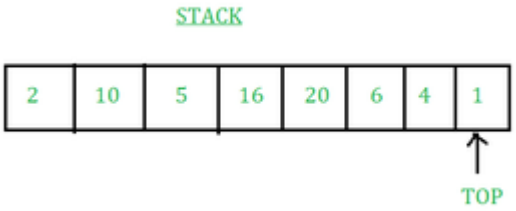
- **int top():** This operation will return the last inserted element that is at the top without removing it.
- **int size():** This operation will return the size of the stack i.e. the total number of elements present in the stack.
- **int isEmpty():** This operation indicates whether the stack is empty or not.
- **int isFull():** This operation indicates whether the stack is full or not.

Types of Stacks:

- Register Stack: This type of stack is also a memory element present in the memory unit and can handle a small amount of data only. The height of the register stack is always limited as the size of the register stack is very small compared to the memory.
- Memory Stack: This type of stack can handle a large amount of memory data. The height of the memory stack is flexible as it occupies a large amount of memory data.

What is meant by Top of the Stack?

The pointer through which the elements are accessed, inserted, and deleted in the stack is called the **top of the stack**. It is the pointer to the topmost element of the stack.



STACK

Application of Stack Data Structure:

- Stack is used for evaluating expression with operands and operations.
- Matching tags in HTML and XML
- Undo function in any text editor.
- Infix to Postfix conversion.
- Stacks are used for backtracking and parenthesis matching.
- Stacks are used for conversion of one arithmetic notation to another arithmetic notation.
- Stacks are useful for function calls, storing the activation records and deleting them after returning from the function. It is very useful in processing the function calls.
- Stacks help in reversing any set of data or strings.

Application of Stack in real life:

- CD/DVD stand.
- Stack of books in a book shop.
- Undo and Redo mechanism in text editors.
- The history of a web browser is stored in the form of a stack.
- Call logs, E-mails, and Google photos in any gallery are also stored in form of a stack.
- YouTube downloads and Notifications are also shown in LIFO format(the latest appears first).

Advantages of Stack:

- Stack helps in managing data that follows the LIFO technique.
- Stacks are be used for systematic Memory Management.
- It is used in many virtual machines like JVM.

- When a function is called, the local variables and other function parameters are stored in the stack and automatically destroyed once returned from the function. Hence, efficient function management.
- Stacks are more secure and reliable as they do not get corrupted easily.
- Stack allows control over memory allocation and deallocation.
- Stack cleans up the objects automatically.

Disadvantages of Stack:

- Stack memory is of limited size.
- The total of size of the stack must be defined before.
- If too many objects are created then it can lead to stack overflow.
- Random accessing is not possible in stack.
- If the stack falls outside the memory it can lead to abnormal termination.

Sample Problem: Reverse order of items in Java

Problem: Given a list of integers in Java, the task is to reverse the given list of integers.
Example:

```
Input: list[] = {10, 20, 30}
Output: 30, 20, 10
Input: list[] = {3, 8, 10, 12, 15}
Output: 15, 12, 10, 8, 3
```

Solution: We can solve this problem in many ways, like using an auxiliary list to first copy elements and then again copy elements in reverse order in the original list.
One more way to solve this problem is using two pointers, one pointing to the first element and one pointing to the last element initially, and then keep on swapping the elements at both pointers and simultaneously increment the first pointer and decrement the last pointer.
But here, we will use Stack class of collections to solve this problem. To solve this problem using stack:

- First, traverse the List and push all elements of it one by one on to the Stack.
- Then traverse the List again and update the element at the current index by the element present at top of stack and pop the stack.

```
// Java Program to reverse a List using Stack
    static void myReverse(List<Integer> list){
        Stack<Integer> s = new Stack<Integer>();
        // Push all the elements in the list into the stack. The last element of
        // the list will be at the top and 1st element at the bottom of the stack.
        for(Integer x: list)
            s.push(x);
        // Pop the stack and insert back into the list.
        for(int i=0; i<list.size(); i++)
        {
            list.set(i, s.pop());
        }
```

Output:

```
[30, 20, 10]
```

Time Complexity: The time complexity of this solution is O(N) if there are N elements in the List.
Auxiliary Space: The auxiliary space is also seen to be O(N) as we need to store all N elements onto a Stack to reverse the List.
Note: This approach is useful when we don't have random access for a particular container. Arrays generally allow us random access, so we can simply use the two pointer approach for arrays as described in the article above. This approach helps when we don't have random access, like for Linked Lists, Deque etc.

Sample Problem: Stock Span in Java

Problem: Given prices of a Stock for **N** consecutive days, the task is to find the span for every day. The span of a stock for a day ***i*** is defined as the number of consecutive days before it with price less than the price on i-th day including the i-th day.
Example:

```
Input: prices[] = {13, 15, 12, 14, 16, 8, 6, 4, 10, 30}
Output: 1, 2, 1, 2, 5, 1, 1, 1, 4, 10
Explanation:
The span for day 1 is always 1 as it is the first day.
The span for day 2 is 2 because price of the day just before it is lesser.
The span for day 3 is 1 because the price of the day before is it greater
The span for day 4 is 2 because there is just 1 consecutive day
    before it with lesser price. Please note here that price of day 1
    is lesser than the price of day 4, but it is not included in the span.
The span of day 5 is 5 because there are 4 consecutive days immediately
    before it. So including the 5th day itself, span is 5.
So on for the rest of the days. ..
Input: prices[] = {30, 20, 25, 28, 27, 29}
Output: 1, 1, 2, 3, 1, 5
```

Solution: If we observe carefully we can see that, span for a day is given by:

- Difference between index of current element and closest greater element on left, if there exists a greater element on left.
- Otherwise, if there doesn't exist any greater element on left, span is equal to "index of current element + 1".

So, the task is to just find the position of the closest element on left of every element which is greater than it. This can be easily done using two nested loops. The idea is:

- The outer loop is used to traverse the array from first to last element.
- The inner loop will be used to find the closest element on left of every element greater than it.

Complexity Analysis:

- The **time complexity** of above solution is $O(N^2)$ in worst case.
- The auxiliary space used will be O(1) as we are not using any extra space.

So, can we do better?

We can solve this problem in linear time complexity, that is, in O(N) time complexity in worst case. Let's see how.

Intuition: Consider the below sequence of elements. For simplicity let's assume we just need to find the closest greater element on the left of every element and not its position.

```
X0, X1, X2,......, Xi | X(i+1), X(i+2), ....., XN
```

Let's say in the above sequence we have found the previous greater element for every element till ith index. Now, there can be two possibilities:

1. X(i+1) is less than Xi.
2. X(i+1) is greater than Xi.

For the **first possibility,** it can be easily seen that if X(i+1) is less than Xi then previous greater for X(i+1) is Xi. Therefore the span is 1.

For the **second possibility**, if X(i+1) is greater than Xi, then it is clear that Xi is of no use for us. But, what information do we need here now? Let's see what we already have, we already have the information about all of the previous greater elements for elements till index i. So, we know which is the closest element on left of ith index which is greater than the element at ith index and so on.

Now, if X(i+1) is greater than Xi, to calculate the span, we need to find the closest element on the left which is greater than X(i+1). So, what we can do is we can compare X(i+1) with the previous greater of Xi because between this all other elements will be smaller than Xi. Now, if X(i+1) is still greater than the previous greater of Xi, compare it with previous greater of previous greater of Xi and so on and so forth.

So, how can we implement this?

We will use a stack data structure to implement the above approach. As, we saw that we will need the information about the useful elements which are the previous greater elements. So, we can do the following:

Push first element on to the stack.

- Start traversing the array from the second element.
- For every new element, compare it with stack top(previous greater)
- Keep poping element from stack top, until we find an element at stack top which is greater than current element.
- Finally push current element to stack.

Note: Since we need indexes for calculating span, we can store indexes in stack rather than elements.

Implementation:

Java

```
        Deque<Integer> stack = new ArrayDeque<Integer>();

        stack.push(0);
        int span = 1;

        for (int i = 0; i < arr.length; i++) {
            while (stack.isEmpty() == false
                   && arr[stack.peek()] <= arr[i]) {
                stack.pop();
            }

            span = (stack.isEmpty()) ? i + 1
                                     : (i - stack.peek());
            System.out.print(span + " ");

            stack.push(i);
```

output:

1 1 2 3 1 2 6 8 1

Time Complexity: O(N)

Auxiliary Space: O(N)

Sample Problem: Design a Stack that supports getMin() operation in O(1)

Problem: Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), and an additional operation getMin() which should return minimum element from the SpecialStack. All these operations of SpecialStack must be O(1). To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, .. etc.

Example:

```
Input: Sequence of operations:
push(20)
push(10)
getMin()
push(5)
getMin()
pop()
getMin()
pop()
```

```
getMin()
Output: 10, 5, 10, 20
Input: Sequence of operations:
push(5)
push(4)
push(3)
getMin()
pop()
getMin()
push(2)
getMin()
Output: 3, 4, 2
```

Solution: Use two stacks: one to store actual stack elements and other as an auxiliary stack to store minimum values. The idea is to do push() and pop() operations in such a way that the top of the auxiliary stack is always the minimum. Let us see how push() and pop() operations work.

Push(int x) // inserts an element x to Special Stack

1) push x to the first stack (the stack with actual elements)
2) compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.
…..a) If x is smaller than y then push x to the auxiliary stack.
…..b) If x is greater than y then push y to the auxiliary stack.

int Pop() // removes an element from Special Stack and return the removed element

1) pop the top element from the auxiliary stack.
2) pop the top element from the actual stack and return it.

Step 1 is necessary to make sure that the auxiliary stack is also updated for future operations.

int getMin() // returns the minimum element from Special Stack

1) Return the top element of auxiliary stack.

We can see that all the above operations are O(1).

Let us see an example. Let us assume that both stacks are initially empty and 18, 19, 29, 15 and 16 are inserted to the SpecialStack.

```
When we insert 18, both stacks change to following.
Actual Stack
18 <--- top
Auxiliary Stack
18 <---- top
When 19 is inserted, both stacks change to the following.
Actual Stack
19 <--- top
18
Auxiliary Stack
18 <---- top
18
When 29 is inserted, both stacks change to the following.
Actual Stack
29 <--- top
19
18
Auxiliary Stack
18 <---- top
18
18
When 15 is inserted, both stacks change to the following.
Actual Stack
15 <--- top
29
19
18
Auxiliary Stack
15 <---- top
18
18
18
When 16 is inserted, both stacks change to the following.
Actual Stack
16 <--- top
15
29
19
18
Auxiliary Stack
15 <---- top
15
18
18
18
```