**Stack & Queues**

The list of the time complexities of various operations on ArrayList and LinkedList is as follows:

1. Finding an element:
   * ArrayList: O(1)
   * LinkedList: O(N)
2. Insertion/Deletion at the beginning:
   * ArrayList: O(N)
   * LinkedList: O(1)
3. Insertion/Deletion at an arbitrary position:
   * ArrayList: O(N)
   * LinkedList: O(N)

#### Q1: ArrayList

Which of the following statements is correct?

Ans: a. The time complexity of finding an element in an arbitrary position in an ArrayList is O(1).

**✓ Correct**

**Feedback:**

If an ArrayList index (position) is given, then you can find the element at that position using arraylist.get(index), which takes constant time. So, the time complexity is O(1).

b. The time complexity of deleting the first element of an ArrayList is O(n).

**✓ Correct**

**Feedback:**

If we delete the first element of an ArrayList, then all the ‘n’ elements after it should be moved to the previous position, which takes O(n) time complexity.

#### Q2. LinkedList

Which of the following statements is correct?

Ans: The time complexity of inserting an element at any arbitrary position in a linked list is O(n) if you are not using an iterator.

**✓ Correct**

**Feedback:**

The time complexity of inserting an element at any arbitrary position in a linked list is O(n) because you first need to traverse the linked list and then add the element. Note that this is different from inserting an element at the beginning/end.



The time complexity of finding an element in a linked list is O(n).

**✓ Correct**

**Feedback:**

The time complexity of finding an element in a linked list is O(n) because you need to traverse through each element of the linked list and check whether or not it matches the required element. In the worst case, you may have to traverse until the end of the linked list.

#### Q3. Data Structures

If you are required to create a program that looks into a database and tries to find an element, then how do you think your choice of ArrayList/LinkedList would impact the performance of the program?

Ans: Elements can be accessed faster with an ArrayList. Therefore, using an ArrayList will make your program faster.

#### Q4. Data Structures

What is the difference between the terms 'data structure' and 'abstract data type'?

 Ans: An abstract data type is an interface that defines the functions that must be implemented by concrete data structure classes that implement it. This ensures flexibility when you code. So, you can use different data structures interchangeably as long as they are implementations of the same abstract data type.

Example: A list is an abstract data type.

Data structures are implementations of abstract data types. For example, ArrayList and LinkedList are implementations of the list abstract data type, and you can use them interchangeably in your code, without much modification.

#### Q5. Last In, First Out

Which of the following options represents ‘last in, first out’?

Ans: A pile of clothes, where one piece of clothing is kept on top of another

**✓ Correct**

**Feedback:**

When clothes are kept in a pile, the piece of clothing that goes in last sits at the top of the pile. Therefore, when you want to get a cloth from the pile, you pick whatever is at the top. This was the last item that you added to the pile. Therefore, a pile of clothes is an example of 'last in, first out'.



The Undo option in text editors (MS Word/Google Docs)

**✓ Correct**

**Feedback:**

When you use the Undo option in a text editor, the last change that you made is the first to be reverted. Therefore, such options are examples of 'last in, first out'.

**Program Stack**

The program stack is also referred to as a call stack, a run-time stack or an execution stack.

There is an extremely common error called ‘stack overflow’, which occurs when you use up more memory for a stack than your program is supposed to. For example, when you frame a recursive logic erroneously and give an infinite recursive call, your compiler will throw a stack overflow error when the size of the stack grows to exceed its maximum allowed size.

#### Q6. Stacks

You learnt that a program stack is an implementation of stacks. Now, why do you think that a stack is a better data structure than a simple array or a linked list to use here?

Ans: The ‘last in, first out’ property of the stack data structure makes it more run-time efficient than a simple array or a linked list in finding the last function called.

**Brief Note About Parsing**

When you were writing programs, you would have encountered errors when you did not use the complete syntax – for instance, a missing bracket or a missing semicolon. You knew that these were syntax errors. Such rules are an extremely significant part of the constructs (grammatical rules) followed while using a programming language. If the input program does not follow any of these language constructs, then the compiler throws an error.

One of the parts of parsing is ‘matching parentheses’. You would have noticed that in Java, you have to enclose all the method arguments within a pair of opening and closing parentheses. Matching parentheses is one of the important tasks that is performed by a parser. You will learn more about it in later segments.

For more information on parsing, you can visit [this](http://www.techopedia.com/definition/3854/parser) link.

# Stacks

## **Stack Class in Java**

There is a Stack class in Java, which implements the stack data structure. The class provides the following functions:

* push(object element): It inserts an element to the top of a stack.
* pop(): It removes an element from the top of a stack.
* isEmpty(): It returns true if a stack is empty; otherwise, it returns false.
* peek(): It returns the element at the top of a stack.
* search(object element): It searches for an element in a stack and returns its location. If the element is not present in the stack, then it returns ‘-1’.

For more details, you can visit the following links:

* [Stack class in Java](https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html)
* [Stack class in Java with an example](https://www.geeksforgeeks.org/stack-class-in-java/)

In this video, you learnt about the position of the Stack class under the Collection interface in Java. You also learnt how to declare the object of the Stack class in Java. Also, note that the Stack class can be implemented using arrays and linked lists, which will be explained in the upcoming segments.

## **Comprehension**

You could encounter the following two types of exceptions in stacks:

* Underflow: Trying to pop/peek an element from an empty stack
* Overflow: Trying to push an element into a stack that is already at its maximum capacity

You can try to avoid these exceptions by writing checks in your code or by handling exceptions. For underflow, you saw that we framed a condition wherein a pop is not allowed from an empty stack. On the other hand, you learnt that stack overflow occurs when you have used up more memory for a stack than your program was supposed to. An instance of this occurs when there are infinite recursive calls, and the compiler shows stack overflow due to the overflow in the program stack.

#### Q7. Stacks

Can you push an element to the top of a stack if it is already full?

Ans: No

**✓ Correct**

**Feedback:**

You cannot push an element to the top of a stack if it is already full. This would throw a stack overflow exception.

#### Q8: Stacks

Can you pop an element from the top of the stack if it is empty?

Ans: No

**✓ Correct**

**Feedback:**

You cannot pop an element from the top of a stack if it is empty. This would throw a stack underflow exception.

#### Q9: Stacks

Which of the following is the correct instruction to create a stack using the Stack class in Java, which takes an integer and where the name of the stack is 'st'?

Ans: Stack<Integer> st = new Stack<Integer>();

**✓ Correct**

**Feedback:**

Correct prototype to create Stack in java.

Q10: What would be the output of the following code?

**import** **java.util.Stack**;

**public** **class** **Source**{

**public** **static** **void** main(String arg[]) {

Stack<Integer> s = **new** Stack<Integer>();

System.out.println(s.empty());

s.push(11);

s.push(22);

s.push(33);

s.push(44);

s.push(55);

System.out.println(s);

System.out.println(s.search(22));

System.out.println(s.search(6));

System.out.println(s.pop());

System.out.println(s);

}

}

Ans: **Output**

true  
[11, 22, 33, 44, 55]  
4  
-1  
55  
[11, 22, 33, 44]

**Let's go through the main functions in the program given above:**

* “Stack<Integer> s = new Stack<Integer>();” creates a stack of name ‘s’.
* “System.out.println(s.empty());” prints 'true' if the stack is empty and 'false' if the stack is not empty. Here, since the stack is empty, it will print 'true'.
* “s.push(element);” pushes the element to the top of the stack. Hence, 11, 22, 33, 44 and 55 will be pushed to the stack.
* “System.out.println(s);” prints out all the elements in the stack.
* In “System.out.println(s.search(22));”, “s.search(22)” returns the position of an element from the top of the stack if it is present; otherwise, it returns “-1”.
* “System.out.println(s.search(6));” prints “-1” since the element is not present in the stack.

To visualise stacks, you can visit the links given below:

[Array implementation](https://www.cs.usfca.edu/~galles/visualization/SimpleStack.html)

[LinkedList implementation](https://www.cs.usfca.edu/~galles/visualization/StackLL.html)

# Matching Multiple Types of Parentheses

**Additional Information**

Suppose you are performing the following operation

**5 + 6 \* (⅔) - 4**

The expression given above is called an infix expression.

Now, how does the compiler perform this operation? How will it recognise which operation to perform first? As you know how parentheses are balanced, you would be able to understand how the operation given above is performed.

(**Hint:** This operation is performed using stacks.)

The compiler evaluates the postfix expression faster than the infix expression. So, it is better to convert an infix expression to a postfix expression.

Conversion of an infix expression to a postfix expression and evaluation of a postfix expression are covered in later sessions.

For additional references, you can visit the links given below:

* [Link 1](http://condor.depaul.edu/ichu/csc415/notes/notes9/Infix.htm) - Convert Infix Expression to Post-Fix Expression.
* [Link 2](https://www.codeproject.com/Articles/752516/Math-Equation-Parsing-using-Call-Stacks) - Equation Parsing Using Call Stack.

# Queues in Detail

**In Java, you can use an in-built Queue implemented using a Linked List.**  
Here, the convention changes a bit: Java queues do not have enqueue and dequeue methods; these operations are carried out using the following methods:

* Enqueuing
  + add(x) – It throws an exception if it fails to insert the object, i.e., when the queue is full.
* Dequeuing
  + remove()– It throws an exception if the queue is empty.
* peek()– It returns null if the queue is empty.
* isEmpty()– It returns true if the queue is empty.

#### Q10: Queues

What would be the output of the following code?

**import** **java.util.LinkedList**;

**import** **java.util.Queue**;

**class** **Source**{

**public** **static** **void** main(String arg[]) {

Queue<Character> q=**new** LinkedList<>();

System.out.println(q.isEmpty());

q.add(*'a'*);

q.add(*'b'*);

q.add(*'c'*);

q.add(*'d'*);

System.out.println(q);

System.out.println(q.size());

System.out.println(q.peek());

System.out.println(q.remove());

System.out.println(q);

System.out.println(q.isEmpty());

}

}

Ans: **Output**

true  
[a, b, c, d]  
4  
a  
a  
[b, c, d]  
false

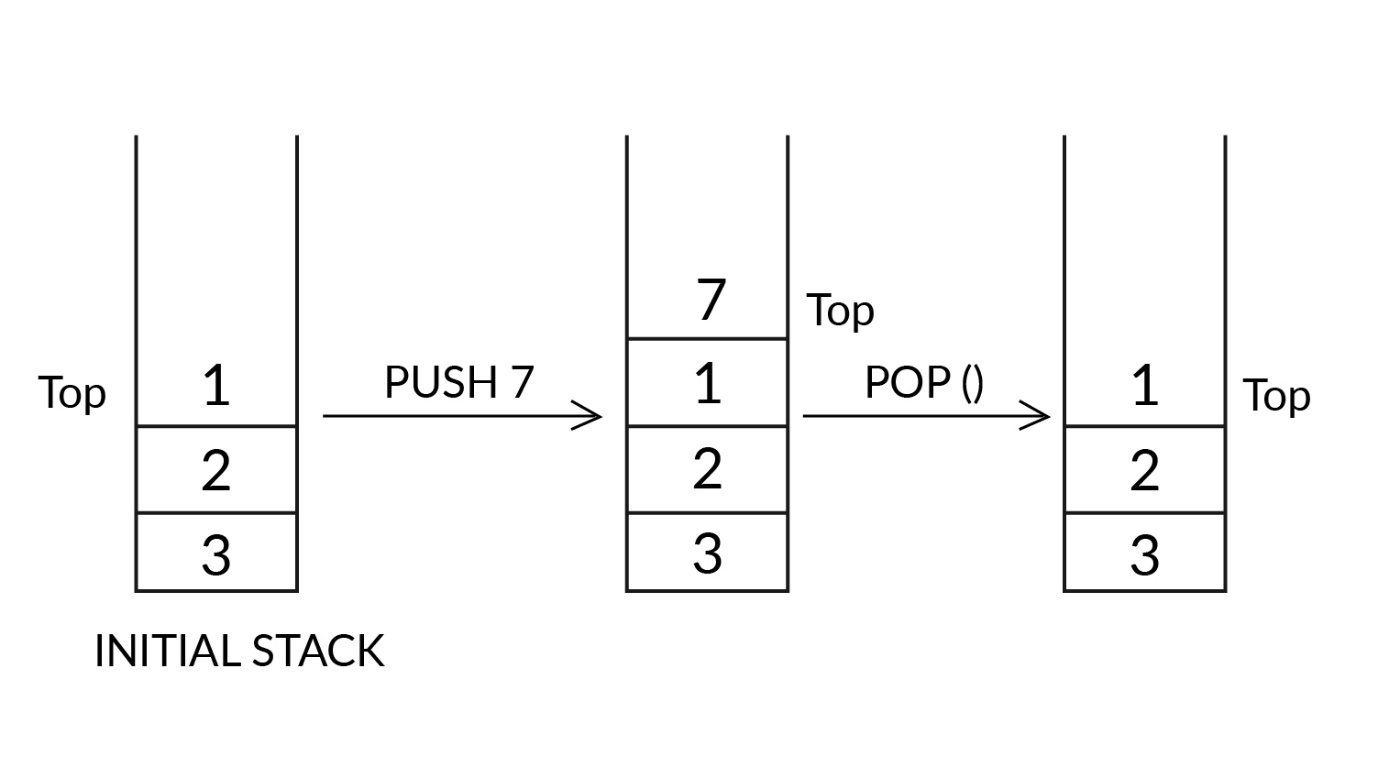
**Now, let us go through the main function in the program above.**  
“Queue<Character> q=new LinkedList<>();” creates queue with name q.  
“q.isEmpty()” returns true if the ‘q’ is empty; otherwise, it returns false.  
“System.out.println(q.isEmpty());” prints true, since the ‘q’ is empty now.  
“q.add('a');” adds the character ‘a’ to the end of the queue, because the queue follows the FIFO order. Similarly b, c and d are added to the queue.  
“System.out.println(q);” prints all the elements in the queue from the beginning to the end.  
“System.out.println(q.size());” prints the size of the queue.  
“System.out.println(q.peek());” prints the peek element, i.e., the starting element of the queue.  
“q.remove()” deletes the peek element of the queue and returns that value.  
“System.out.println(q);” prints the elements in the queue.

# Implement a Stack Using Two Queues

Let us revise some concepts of stacks and queues that we have covered already and get started with trying to implement a stack using two queues.

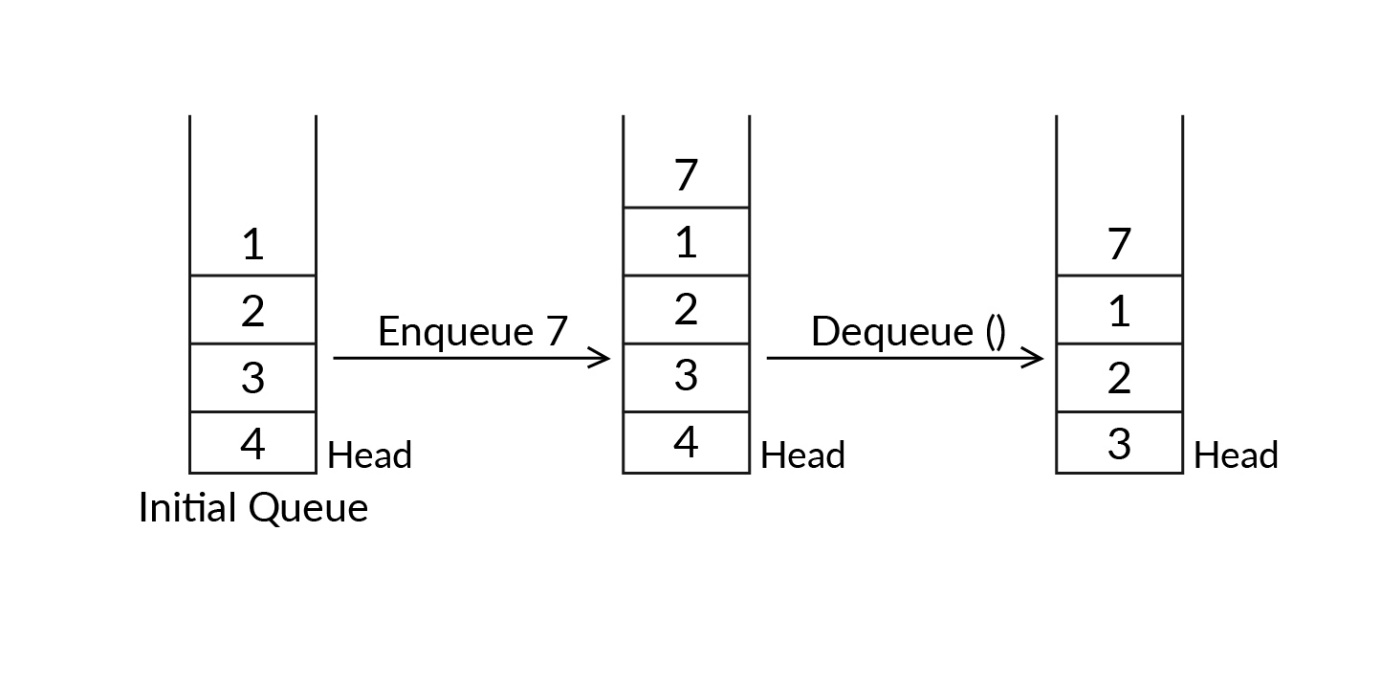
Stack is the data structure that follows the LIFO property. LIFO stands for Last In, First Out, which means the most recently added element would be removed/accessed first. However, Stack has a restriction: Insertion and deletion can be done only at one end, which is called the **'TOP'** of the stack.

The image below illustrates the **'push'** and **'pop'** operations on a stack.



Push and Pop

In contrast, Queue is the data structure that follows the FIFO property. FIFO stands for First In, First Out, which implies the element that is inserted first would be removed/accessed first. In queues, you can insert (enqueue) elements only at one end, which is called ‘TAIL’, and deletion (dequeue) of elements in the queue can take place at the other end, which is called ‘HEAD’.    
  
The image below illustrates the '**enqueue'**and **'dequeue'** operations on a queue.



Enqueue and Dequeue

So, in this segment, you will try to implement stacks using single and multiple queues. Let’s begin by answering some fundamental questions.

#### Q11: Stack Using two Queues

In the previous approach, mainly the push operation was made costly while implementing a stack using queues.

Now, you need to use the pop operation to implement a stack using two queues.

(**Hint:** You have to maintain the order of the elements in one queue as that in a stack while adding a new element.)

Since a new element will be added at the rear of the queue, to pop it, you have to first bring the new element to the front of the queue by dequeuing every other element to another queue.

Ans: We will be using two queues, Q1 and Q2, to implement the stack.

Pseudocode:

push(x)

1. Enqueue x to Q1

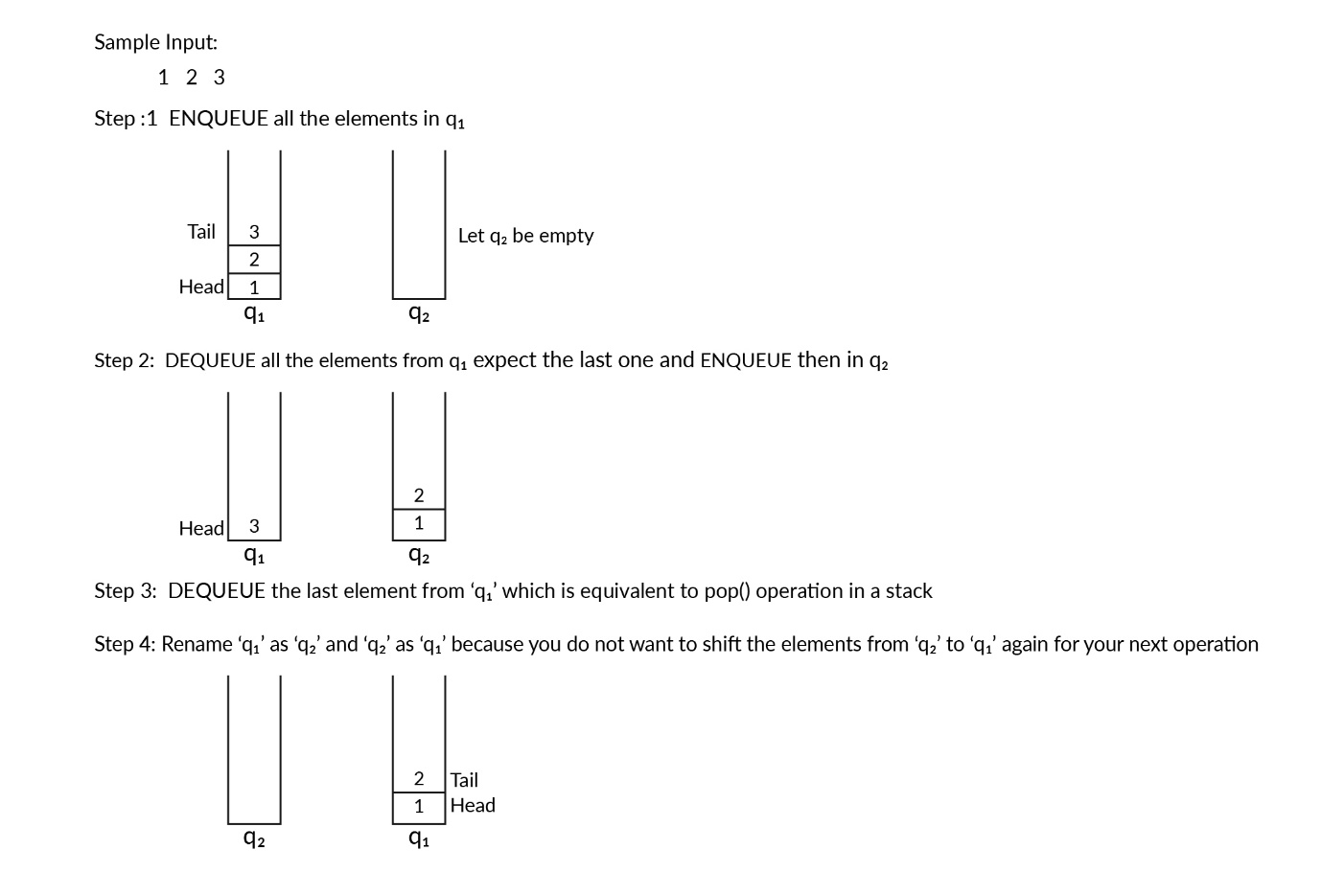
pop()

1. Dequeue each element, except the last one, from Q1 and insert into Q2 one by one
2. Dequeue the last element from Q1 – which is the result – and store it
3. Swap the names of Q1 and Q2
4. Return the element stored in Step 2

/\* This method is based on making the **‘pop’ operation costly**. \*/

Note: The method above is known as “Implementing stacks using two queues by making the pop operation costly”.

Now, let us revise how the pop() operation can be implemented using two queues.



Stack using two queues

#### Q12: Time Complexity

Predict the time and space complexity of the pseudocode designed in the previous video given that the queue initially has ‘n-1’ elements.

Ans: O(n) and O(n), respectively

**✓ Correct**

**Feedback:**

The time complexity of the pseudocode would be the sum of the time taken by the following subtasks:

1. Pushing a new element into queue ‘q1’ would take O(1) time.
2. N-1 elements will be dequeued from ‘q1’ and added to ‘q2’, which would take O(n-1) time.
3. Finally, the last element would be dequeued in O(1) time.

Therefore, total time complexity would be O(1) + O(n-1) + O(1) = O(n).

The method requires an extra queue of the same size as the number of elements in the first queue. Hence, the space complexity is also O(n).

#### Q13: Stacks Using a Single Queue

Write a basic algorithm to implement the push and pop operations of stacks using a single queue.

Ans: **push(x)**

1. Calculate the size (s) of the queue (Q)
2. Enqueue x to the queue (Q)
3. One by one, dequeue s items from Q and enqueue them to Q

/\* pop(x) removes an item from the stack \*/

**pop(x)**

1) Dequeue an item from the queue (Q)

Watch the following video to understand the pseudocode better.

#### Q14: Identify a Palindrome Using a Stack

Write an algorithm to determine whether or not a given string is a palindrome, using stacks.

Ans: 1. Use a ‘**for**’ loop to push each character of the string into an array stack  
2. Initialise an empty string, say, “reversedString”  
3. pop() the characters from the stack one by one and add them to the reversedString  
4. Compare the input string and the reversed string. If they are equal, then print  'The input String is a palindrome'  
5. If they are not equal, then print  'The input String is not a palindrome'

#### Q15: Identify a Palindrome Using a Stack

Now that you have framed an algorithm to identify a palindrome using stacks, given below is an incomplete piece of code for the same. From the options below, choose the correct values of X, Y and Z that can complete this code:

(**Hint:** Look at the suggested answer to the previous question if you were not able to create an algorithm)

**for** (...X...) {

stack.push(inputString.charAt(i));

}

String reverseString = "";

**while** (...Y...) {

reverseString = reverseString+stack.pop();

}

**if** (...Z...)

System.out.println("The input String is a palindrome.");

**else**

System.out.println("The input String is not a palindrome.");

}



Ans:   
X = int i = 0; i < inputString.length(); i++,   
Y = !stack.isEmpty()  
Z = inputString.equals(reverseString)

**✓ Correct**

**Feedback:**

**X:**According to this condition of the 'for' loop, the characters of the string will be inserted in the order in which they are originally present; hence, the option is correct.

**Y:**The given 'while' loop will pop elements from the stack and add them to reversedString, which is required for comparing with the original string. Therefore, the while loop should execute until the stack is not empty, and according to “!stack.isEmpty”, the 'while' loop will execute until the stack is not empty.

**Z:** The ‘if’ condition prints 'The input String is a palindrome', which must be printed only when the input string matches the reversed string. Therefore, this option is correct.

#### Q16: Identify a Palindrome Using a Queue

Predict the time and space complexity, based on the code that you designed in the previous question, to check whether a given string (with ‘n’ characters) is a palindrome or not, using a queue.

(**Hint:** If you are not able to solve the question you can have a look a the sample solution of the previous question and then attempt.)

Ans: Time complexity is O(n) and space complexity is O(n2).

**✓ Correct**

**Feedback:**

The code will contain two separate loops. The ‘for’ loop would push all the characters of the string in the queue, and the ‘while’ loop would compose a reversed string out of the elements dequeued from the queue. Since the loops are not nested, the time complexity of both the loops is O(n). In the program, you need to have an extra queue as well as a string variable and both of them need space. The size of the queue and the variable string should be large enough to store all the ‘n’ characters of the input string. The queue would take O(n) space and the variable string would take O(n2) space, because they are immutable in Java. The space complexity of both the data structures adds up to give O(n) + O(n2) = O(n2).

#### Q17: Stacks and Queues

Which data structure is better for identifying a palindromic string?

(**Note:** Better in terms of time and space complexity)

Ans: Both perform equally.

**✓ Correct**

**Feedback:**

 The number of steps, as well as the number of loops, is the same for both the data structures; therefore, the comparison does not yield any result, and it can be concluded that both perform equally. The time complexity is O(n) and the space complexity is O(n2) for both the data structures.

#### Q18: Identify a Palindrome

Suggest a method with which you can reduce the space complexity of the solution suggested in the questions above from O(n2) to O(n).

Ans: The space complexity of the solution code is O(n2), due to the variable string used. Strings are immutable in Java and, hence, a new string of length ‘n’ will be created ‘n’ times. To avoid creating a new string everytime you add a character, you can use the StringBuilder class in Java. Strings created using the StringBuilder class are modifiable and do not need any repetitive creation. Therefore, space complexity is reduced to O(n). Below, you can find an incomplete piece of code that can be used to identify a palindromic string.

Scanner in = **new** Scanner(System.in)

StringBuilder inputString = **new** StringBuilder( );

inputString=in.nextLine();

StringBuilder reverseString = **new** StringBuilder( );

reverseString=sb.reverse();

**if**(reverseString==inputString) System.out.println( "The input string is palindrome");

**else** System.out.println( "The input string is not a palindrome");

#### Q18: Detect Balanced Parentheses

Write a pseudocode or a Java code to detect balanced parentheses in an input expression. For example:

**SAMPLE INPUT:**  
(a+b)

**SAMPLE OUTPUT:**  
true

**SAMPLE INPUT:**  
(a()

**SAMPLE OUTPUT:**  
false

[**Note:** You need to consider balancing only parentheses, i.e., ( ).]

Ans: The given problem can be solved using a stack, as shown below:

**public** **static** **boolean** **checkBalanced**(String inputString){

Stack < Integer > inputStack = **new** Stack < Integer > ();

Stack<Character> newStack=**new** Stack<>();

**for**(**int** i=**0**;i<inputString.length();i++) {

**if** (inputString.charAt(i) == '(') {

newStack.push(inputString.charAt(i));

}

**if** (inputString.charAt(i) == ')') {

**if** (newStack.isEmpty()) {

**return** **false**;

} **else** {

newStack.pop();

}

}

}

**if**(newStack.isEmpty())

**return** **true**;

**else**

**return** **false**;

}



The time complexity of the algorithm above is O(n), and its space complexity is O(n) as well.

# Reverse a Stack

Stacks can be implemented using various data structure, such as Arrays, ArrayLists and LinkedLists, in accordance with the different purposes for which a stack is to be used. Generally, a stack is represented by an array. One of the applications of a stack is to reverse an input sequence. To reverse a sequence, you can insert its characters one by one and, later, pop them out to get the reversed output. But what if we want the input sequence to be reversed within the stack itself?

For example:  
If the input stack contains the elements { a b c d e }, where 'e' is the element at the top of the stack, then the reverse stack would contain the elements { e d c b a }, where 'a' is the element at the top of the stack.

#### Q19: Reverse a Stack

Which of the following data structures is used by recursion to make the function calls?

(**Note:** Recall the concepts of recursion and then answer)

Ans: Stacks

**✓ Correct**

**Feedback:**

The critical function of any recursive algorithm is to break a given problem into a large number of similar but simpler problems, which can be solved individually and combined later to get the final result. The solutions of those individual problems have to be stored in an order where they can be backtracked after the base condition returns a value. The LIFO property of stack helps with backtracking and, therefore, it can be easily used for implementing a recursive algorithm. The stack that is used for recursion is known as a ‘Program Stack’. For a better understanding of how a stack is used in recursion, read the text [here](http://mooreccac.com/kcppdoc/Recursion.htm).

#### Q20: Reverse a Stack

Write a pseudocode to reverse a sequence stored in a stack using recursion.

(**Note:**You are not allowed to use any while loop for your algorithm.)

Ans: //Pseudocode to reverse the stack

function **reverse**()

A. IF stack is not empty

//Hold all items in function call stack until we reach the end of the stack

**1.** LET i contains top element from stack.

**2.** POP the top element from stack.

**3.** reverse()

// Insert all the items in function call stack one by one from the bottom to top. Each item is inserted at the bottom of the stack

**4.** insertAtEnd(i)

// The following function inserts an element at the bottom of a stack

function **insertAtEnd**(i)

A. IF stack is empty

**1.** PUSH i to the top of the stack

B. ELSE

/\* All items are held in function call stack until we reach end of the stack.

When the stack becomes empty the item is inserted at the bottom.\*/

**1.** In varaible j we will store the value which is at the top of stack.

**2.** POP the top element from the stack

**3.** insertAtEnd(i)

//Push all the items which are held in function call stack once the item is inserted at the bottom

**4.** PUSH j to the top of the stack

#### Q21: Reverse a Stack

Choose the correct time and space complexity for reversing a sequence in a stack using recursion. 

(**Note:** The input sequence has ‘n’ number of elements.)

**Hint:**If you are unable to answer the question, then look at the suggested answer to the previous question.

Ans: Time complexity is O(n) and space complexity is O(n).

**✓ Correct**

**Feedback:**

According to the pseudocode above, there are two functions that will determine time and space complexity. The reverse() method traverses each element of the stack recursively, stores it in reverse order and, later, calls the other function, insertAtBottom, for each element. Time complexity is O(n) since the reverse() method contains some executable statements for each of the ‘n’ elements of the stack.

The ‘insertAtBottom’ method would also traverse the list recursively and make sure the new element is inserted at the bottom-most position of the existing stack; hence, time complexity is O(n). The time complexity of both the methods is O(n). Space complexity is O(n) because along with the input stack, another stack, called the program stack, is being used by the recursive algorithm.

#### Q22: Reverse a Stack

What is the time and space complexity of reversing a stack of the code that you have created above?  
Assume the stack contains ‘n’ elements.

(**Hint:** If you are unable to solve this question, then look at the sample solution to the previous question.)

Ans: Time complexity is O(n) and space complexity is O(1).

**✓ Correct**

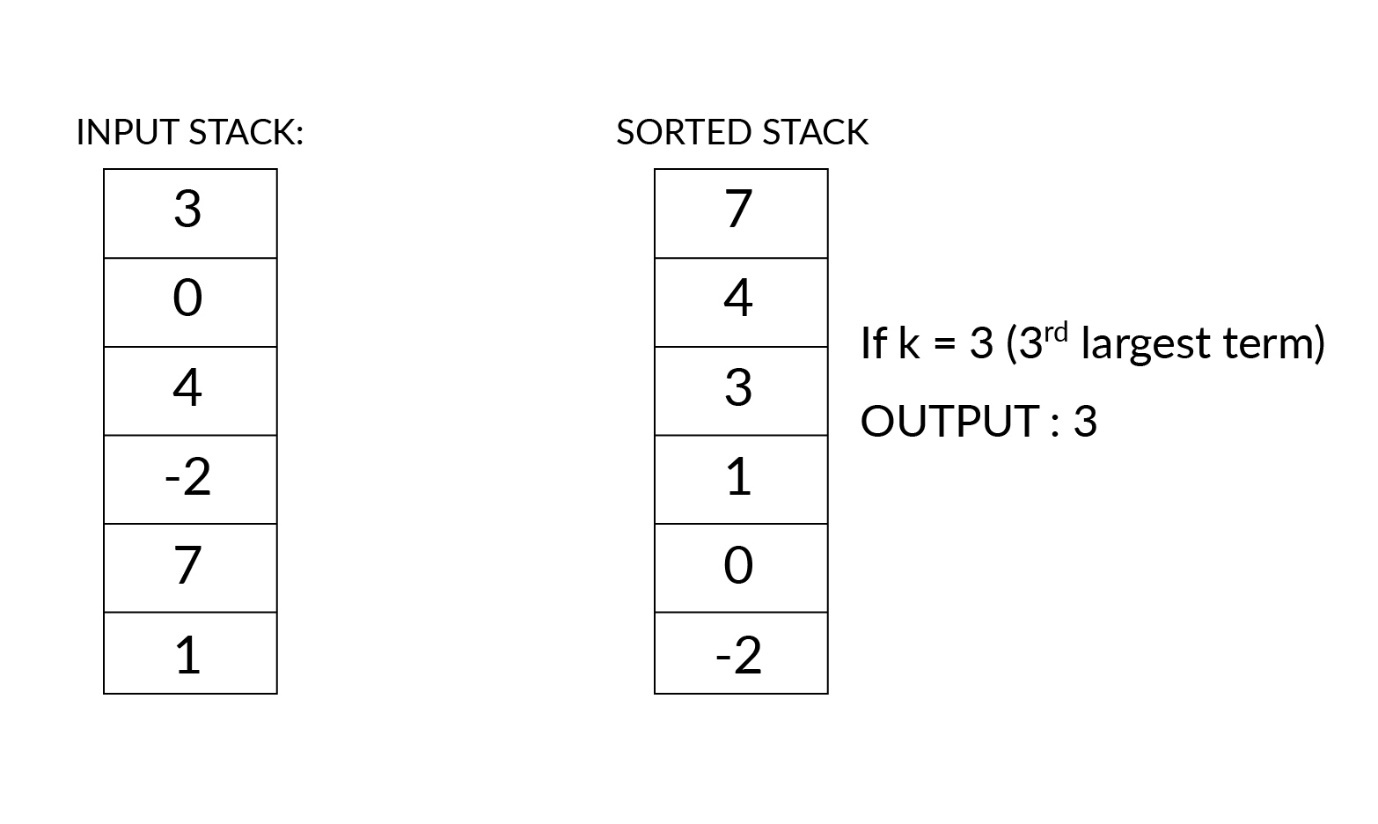
**Feedback:**

In the code above, the ‘while’ loop iterates through each and every, node/element of the stack and reverses it. Therefore, time complexity is O(n). Also, no auxiliary memory is used; hence, space complexity is constant, i.e., O(1).

# Find the kth Largest Element

Sorting refers to arranging a sequence of data systematically based on specific criteria. One of the benefits of sorting is that it makes selection easier. For example, if you need to find the kth largest element in a list of numbers, then you can reach that element in constant time if the list is arranged in either descending or ascending order. Stacks and queues are two data structures that are used frequently for storing data. Hence, you will encounter situations where you need to sort elements using these data structures. In this segment, your task is to first sort and then find the kth largest element in the given input sequence.

For example:



Kth largest

#### Q23: Find the kth Largest Element

Write an algorithm to sort a sequence of numbers stored in an input stack using another temporary stack. The sequence should be arranged in a descending order starting from the top of the stack.

Ans: 1. Create a temporary stack, say, ‘temp’

2. When the input stack is NOT empty:

1. pop() an element and store it in the temporary integer variable ‘value’
2. When the ‘temp’ stack is NOT empty AND ‘value’ <top value in ‘temp’ stack:
   1. Pop the top value from ‘temp’ stack and push it into ‘input’ stack
3. Push ‘value’ into ‘temp’ stack

3. Return ‘temp’ stack, which contains all the elements in the input stack in descending sorted order

#### Q24: Find the kth Largest Element

Predict the time and space complexity of the algorithm used in the previous question to sort a given sequence in a stack using another stack. Note that the input stack contains ‘n’ elements initially.

(**Note:** Look at the suggested answer to the previous question if you are unable to answer this question.)

Ans: O(n2) and O(n), respectively

**✓ Correct**

**Feedback:**

The algorithm created in the previous question contains two nested ‘while’ loops. The first 'while' would pop all the elements from the input stack one by one, and each of the elements would be compared to the elements in the temporary stack. If a number popped from input stack is less than the element at the top of the temporary stack, then it would replace the element at the top, and the replaced element would be pushed back into the input stack. Both the 'while' loops would operate for all the elements, and, hence, time complexity is O(n\*n) =O(n2). The extra space used here should be of the same size as the input stack, because the final sorted list would be stored in the temporary stack only. Therefore, space complexity is O(n).

#### Q25: Pseudocode

You are given a company logo that is circular in shape:



Your computer-vision algorithm starts from the character at the top and proceeds in a clockwise manner. The algorithm marks the characters in the lower half with a ‘\*’. Therefore, for the logo above, it wolud generate the following output:

[B,U,C,K,S,E\*,E\*,F\*,F\*,O\*,C\*,S,T,A,R]

Which of the following pseudocode snippets would process the above text successfully to produce “STARBUCKSCOFFEE”?

A.

queue\_text = Queue()

stack\_text = Stack()

output = Queue()

// This will go through each element present in the input

**for** (element : input){

**if** (element contains "\*"){

stack\_text.push(element - "\*") //This will remove "\*" from element

}

**else** {

queue\_text.enqueue(element)

}

}

**while** !(queue\_text.isEmpty()){output.enqueue(queue\_text.dequeue())}

**while** !(stack\_text.isEmpty()){output.enqueue(stack\_text.pop())}



B.

queue\_text = Queue()

stack\_text = Stack()

output = Queue()

// This will go through each element present in the input

**for** (element : input){

**if** (element contains "\*"){

queue\_text.enqueue(element - "\*")//This will remove "\*" from element

}

**else** {

stack\_text.push(element)

}

}

**while** !(queue\_text.isEmpty()){output.enqueue(queue\_text.dequeue())}

**while** !(stack\_text.isEmpty()){output.enqueue(stack\_text.pop())}



C.

queue\_text = Queue()

stack\_text = Stack()

output = Queue()

top\_end = **false**

// This will go through each element present in the input

**for** (element : input){

**if** (element contains "\*"){

top\_end = **true**

stack\_text.push(element - "\*") //This will remove "\*" from element

}

**else** {

**if** (top\_end){stack\_text.push(element)}

**else** {queue\_text.enqueue(element)}

}

}

**while** !(stack\_text.isEmpty()){output.enqueue(stack\_text.pop())}

**while** !(queue\_text.isEmpty()){output.enqueue(queue\_text.dequeue())}



D.

queue\_text = Queue()

stack\_text = Stack()

top\_end = **false**

output = Queue()

// This will go through each element present in the input

**for** (element : input){

**if** (element contains "\*"){

stack\_text.push(element - "\*")//This will remove "\*" from element

top\_end = **true**

}

**else** {

**if** (top\_end){output.enqueue(element)}

**else** {queue\_text.enqueue(element)}

}

}

**while** !(queue\_text.isEmpty()){output.enqueue(queue\_text.dequeue())}

**while** !(stack\_text.isEmpty()){output.enqueue(stack\_text.pop())}

Ans: D

**✓ Correct**

**Feedback:**

At the end of the iteration, the code would output STAR. Then BUCKS would be dequeued from the queue and COFFEE from the stack.