



Ahmad Albarasy · Follow

Last edited by Ahmad Albarasy on Dec 13, 2025

Contributed by



DSaA Lab 5 - Stacks and Queues

The Islamic University of Gaza
Engineering Faculty
Department of Computer Engineering

Prepared by: Eng. Ahmad Albarasy

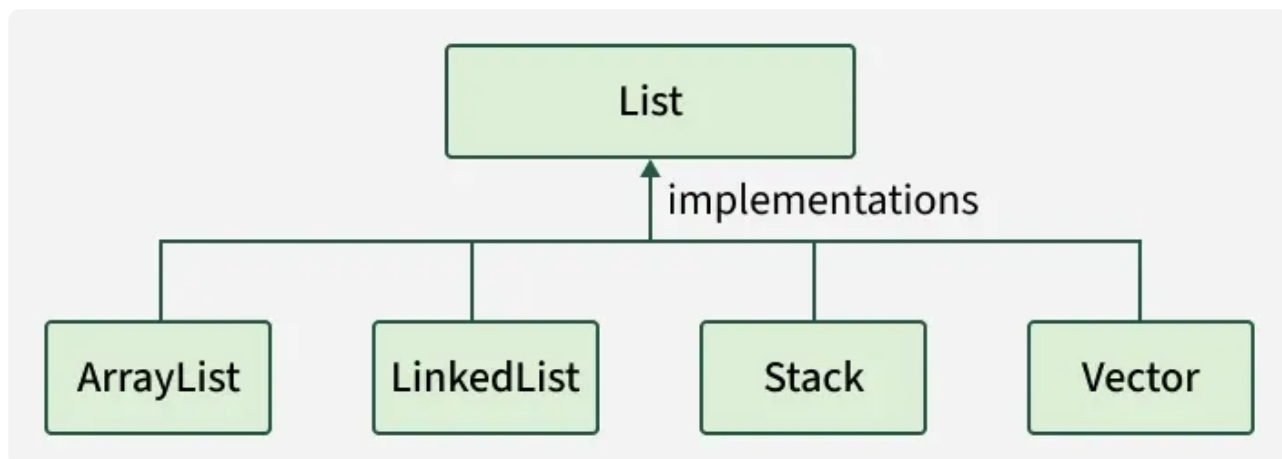
Under the supervision of: Prof. Ahmad Mahdi

In this lab, we are going to explore stacks and queues, which are two fundamental linear data structures used to organize data in two different fashions, leading to each one being suitable for solving a distinct set of problems. We will also learn about and compare the different ways of implementing them.

What is an ADT?

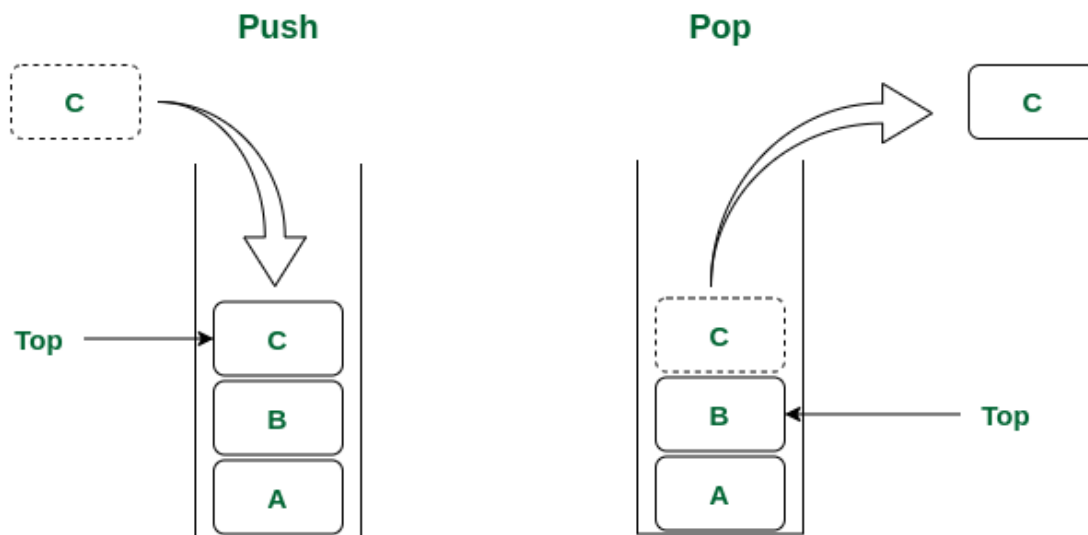
An Abstract Data Type is a mathematical model for a data structure that specifies what operations can be performed on the data and what their expected behavior is, without specifying how the data or operations are implemented.

In Java, we use interfaces to define the expected behavior of a data structure without specifying its implementation. For instance, take a look at the [List](#) interface in Java, which is used to represent an ordered collection of elements.



The Stack

A stack is a linear data structure that follows the **LIFO (Last-In, First-Out)** principle, where the last element that's pushed onto the stack is the first element that gets removed. A real world analogy would be a pile of plates placed on top of each other, where you can only place or remove plates from the top.



Stack Data Structure

The Stack ADT

```

1  /*
2   * A collection of objects that are inserted and removed
3   * according to the last-in first-out principle.
4   * Although similar in purpose, this interface differs from
5   * java.util.Stack
6   */
7  public interface Stack<E> {
8      /* returns the number of elements in the stack */
9      int size();
10     /* return true if the stack is empty, false otherwise */
11     boolean isEmpty();
12     /* returns the top element without removing it (or null if empty) */
13     E top();
14     /* pushes an element on top of the stack */
15     void push(E element);
16     /* removes and returns the top element (or null if empty)*/
17     E pop();
18 }

```

Implementation

To implement a stack, we commonly rely on two fundamental data structures: either an array or a linked list. Each implementation comes with its own advantages and

disadvantages, and your choice will heavily rely on the nature of the problem you are trying to solve.

For instance, an array-based implementation would be inefficient in cases where the size of the stack grows over time, as that would be expensive due to the cost of resizing the array (moving all elements to the new larger one). On the contrary, a linked list-based implementation would allow the stack to grow and shrink dynamically with no need for resizing, but it introduces an additional memory overhead due to the storage of pointers.

Array-based Implementation

```
1 public class ArrayStack<E> implements Stack<E> {
2     public static final int CAPACITY = 1000; // default array capacity
3     private E[] data; // generic array used for storage
4     private int t = -1; // index of the top element in stack
5
6     public ArrayStack() {
7         this(CAPACITY); // constructs stack with default capacity
8     }
9
10    public ArrayStack(int capacity) { // constructs stack with given capacity
11        data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
12    }
13
14    public int size() {
15        return t + 1;
16    }
17
18    public boolean isEmpty() {
19        return t == -1;
20    }
21
22    public void push(E e) throws IllegalStateException {
23        if (size() == data.length) throw new IllegalStateException("Stack is full");
24        data[++t] = e; // increment t before storing new item
25    }
26
27    public E top() {
28        if (isEmpty()) return null;
29        return data[t];
30    }
31
32    public E pop() {
33        if (isEmpty()) return null;
34        E answer = data[t];
35        data[t] = null; // dereference to help garbage collection
36        t--;
37        return answer;
38    }
39 }
```

Linked List-based Implementation

```
1 public class LinkedStack<E> implements Stack<E> {
2     private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty list
3
4     public LinkedStack() {
5         // new stack relies on the initially empty list
6     }
7
8     public int size() {
9         return list.size();
10    }
11
12    public boolean isEmpty() {
13        return list.isEmpty();
14    }
15
16    public void push(E element) {
17        list.addFirst(element);
18    }
19
20    public E top() {
21        return list.first();
22    }
23
24    public E pop() {
25        return list.removeFirst();
26    }
27 }
```

Usage

Stacks are widely used due to their simple **FIFO** storage fashion, which makes them ideal for problems where the most recent item needs to be processed first. Some common applications of stacks include:

1. **Undo/Redo operations in editors:** Programs like text editors use stacks to keep track of user actions. When the user performs an action, it is pushed onto the undo stack. If the user presses Undo, the action is popped from the undo stack and pushed onto a redo stack, allowing it to be redone later:

```
1  import java.util.Stack;
2
3  Stack<Action> undoStack = new Stack<>();
4  Stack<Action> redoStack = new Stack<>();
5
6  void perform(Action a) {
7      a.apply();
8      undoStack.push(a);
9      redoStack.clear();
10 }
11
12 void undo() {
13     if (!undoStack.isEmpty()) {
14         Action a = undoStack.pop();
15         a.reverse();
16         redoStack.push(a);
17     }
18 }
19
20 void redo() {
21     if (!redoStack.isEmpty()) {
22         Action a = redoStack.pop();
23         a.apply();
24         undoStack.push(a);
25     }
26 }
```

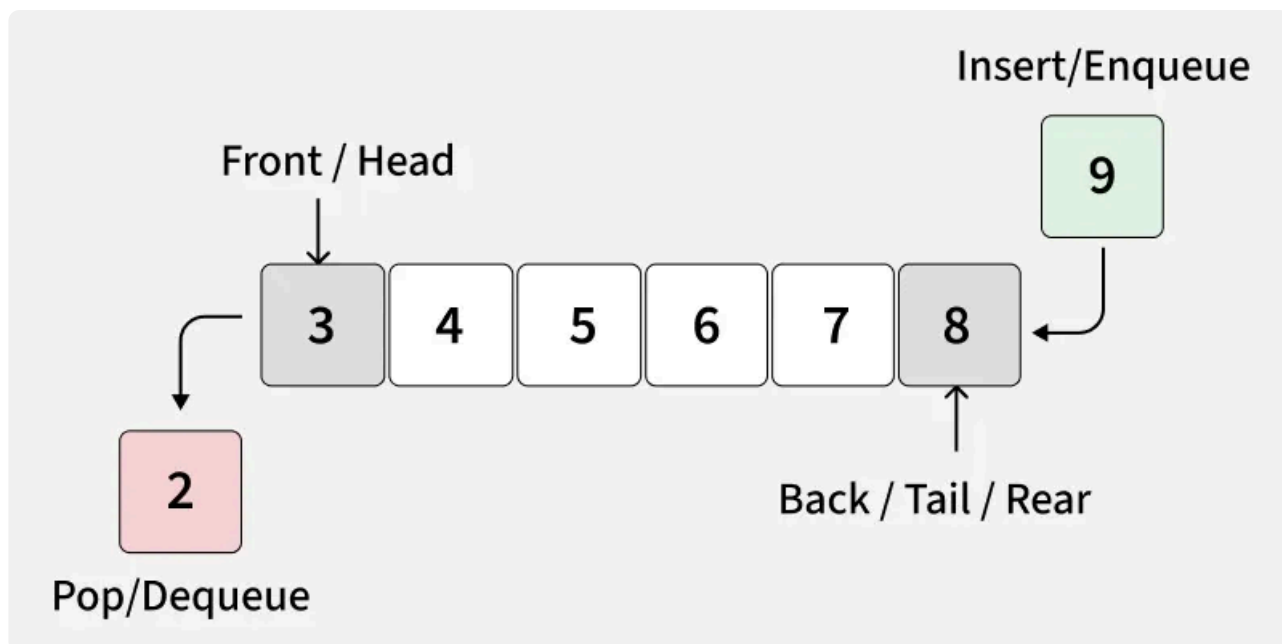
2. **Naive path finding algorithms:** A stack can be used to keep track of the positions to explore when implementing a simple path-finding algorithm. For instance, the following algorithm explores a maze and finds the exit by backtracking whenever it reaches a dead end:


```
1  import java.util.Stack;
2
3  class Position {
4      int row, col;
5      Position(int r, int c) { row = r; col = c; }
6  }
7
8  public class MazeSolver {
9
10     public static boolean simpleMazeSolver(int[][] maze) {
11         int rows = maze.length;
12         int cols = maze[0].length;
13
14         Stack<Position> stack = new Stack<>();
15         stack.push(new Position(0, 0)); // start at top-left
16         boolean[][] visited = new boolean[rows][cols];
17
18         while (!stack.isEmpty()) {
19             Position p = stack.pop();
20             int r = p.row;
21             int c = p.col;
22
23             // Check boundaries, walls, and visited cells
24             if (r < 0 || r >= rows || c < 0 || c >= cols) continue;
25             if (maze[r][c] == 1 || visited[r][c]) continue;
26
27             visited[r][c] = true;
28
29             printMaze(maze, p);
30
31             // Check if we reached the exit (bottom-right)
32             if (r == rows - 1 && c == cols - 1) {
33                 System.out.println("Maze solved!");
34                 return true;
35             }
36
37             // Push adjacent positions (up, down, left, right)
38             stack.push(new Position(r + 1, c));
39             stack.push(new Position(r - 1, c));
40             stack.push(new Position(r, c + 1));
41             stack.push(new Position(r, c - 1));
42         }
43
44         System.out.println("No path found.");
45         return false;
46     }
47
48     public static void printMaze(int [][] maze, Position p){
49         for(int i = 0; i < maze.length; i++){
50             for(int j = 0; j < maze[0].length; j++){
51                 if(i == p.row && j == p.col)
52                     System.out.print("m "); // indicates current position
53                 else
54                     System.out.print(maze[i][j] + " ");
55             }
56             System.out.println(); // end of row
57         }
58         System.out.println(); // end of grid
```

```
59     }
60
61     public static void main(String[] args) {
62         // 0 = path, 1 = wall/block
63         int[][] maze = {
64             {0, 0, 1, 0},
65             {1, 0, 1, 0},
66             {0, 0, 0, 1},
67             {0, 1, 0, 0}
68         };
69
70         simpleMazeSolver(maze);
71     }
72 }
```

The Queue

A queue is a linear data structure that follows the **FIFO (First-In, First-Out) principle**, where the first element that's added to the queue is the first element that gets removed. A real-world analogy would be people standing in line, where the person who arrives first is the first one to be served.



The Queue ADT:

```
1  /*
2   * A collection of objects that are inserted and removed
3   * according to the first-in first-out principle.
4   * Although similar in purpose, this interface
5   * slightly differs from java.util.Queue
6   */
7
8  public interface Queue<E> {
9      /** Returns the number of elements in the queue. */
10     int size();
11
12     /** Tests whether the queue is empty. */
13     boolean isEmpty();
14
15     /** Inserts an element at the rear of the queue. */
16     void enqueue(E e);
17
18     /** Returns, but does not remove,
19      * the first element of the queue (null if empty).
20      */
21     E first();
22
23     /** Removes and returns the first element of the queue (null if empty). */
24     E dequeue();
25 }
```

Implementation

To implement a queue, we also have 2 options: either implementing it using an array of a linked list. The choice depends on the nature of the problem we are trying to solve, as each option comes with its own advantages and disadvantages.

An array-based implementation is relatively simple and offers constant access times, but it may require resizing as our queue grows in size and careful index management are needed. On the other hand, a linked list-based implementation provides more flexibility in size and avoids resizing, though it uses extra memory due to additional pointers.

Array-based Implementation


```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3
4      // instance variables
5      private E[] data;    // generic array used for storage
6      private int f = 0;   // index of the front element
7      private int sz = 0;  // current number of elements
8
9      // constructors
10     public ArrayQueue() {
11         this(CAPACITY);
12     }
13
14     public ArrayQueue(int capacity) {
15         data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
16     }
17
18     // methods
19
20     /** Returns the number of elements in the queue. */
21     public int size() {
22         return sz;
23     }
24
25     /** Tests whether the queue is empty. */
26     public boolean isEmpty() {
27         return (sz == 0);
28     }
29
30     /** Inserts an element at the rear of the queue. */
31     public void enqueue(E e) throws IllegalStateException {
32         if (sz == data.length)
33             throw new IllegalStateException("Queue is full");
34         int avail = (f + sz) % data.length; // use modular arithmetic
35         data[avail] = e;
36         sz++;
37     }
38
39     /** Returns, but does not remove,
40      * the first element of the queue (null if empty).*/
41     public E first() {
42         if (isEmpty())
43             return null;
44         return data[f];
45     }
46
47     /** Removes and returns the first element of the queue (null if empty). */
48     public E dequeue() {
49         if (isEmpty())
50             return null;
51         E answer = data[f];
52         data[f] = null; // dereference to help garbage collection
53         f = (f + 1) % data.length;
54         sz--;
55         return answer;
56     }
57 }
58
```

Linked List-based Implementation

```
1  /** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
2
3  public class LinkedQueue<E> implements Queue<E> {
4
5      private LinkedList<E> list = new SinglyLinkedList<>(); // an empty list
6
7      public LinkedQueue() {
8          // new queue relies on the initially empty list
9      }
10
11     public int size() {
12         return list.size();
13     }
14
15     public boolean isEmpty() {
16         return list.isEmpty();
17     }
18
19     public void enqueue(E element) {
20         list.addLast(element);
21     }
22
23     public E first() {
24         return list.first();
25     }
26
27     public E dequeue() {
28         return list.removeFirst();
29     }
30 }
```

Usage

Queues are used in a variety of problems where data need to be processed in the exact order in which they arrive. This is useful in situations where requests or tasks should be processed fairly, one after another, without skipping earlier ones. Some common applications of queues include:

1. **Buffering arriving packets in a router:** When a router receives network packets faster than it can process or forward them, the packets are temporarily stored in a queue. Packets are then forwarded in the order they arrive, ensuring organized and fair handling of network traffic.
2. **Server joining queues in multiplayer games:** Queues are used in online games multiplayer servers to moderate the number of players currently in the server to prevent overloading it. Players who requested to join the server while full are placed

in a wait queue and are allowed in once someone leaves the server, thus making room for others to join.

Tasks

Task 1 (4 points)

You are given a string `s`.

Your task is to remove all digits by doing this operation repeatedly:

Question link on [LeetCode](#)

- Delete the first digit and the closest non-digit character to its left.
Return the resulting string after removing all digits.

Note that the operation cannot be performed on a digit that does not have any non-digit character to its left.

Example 1:

Input: `s = "abc"`

Output: `"abc"`

Explanation:

There is no digit in the string.

Example 2:

Input: `s = "cb34"`

Output: `""`

Explanation:

First, we apply the operation on `s[2]`, and `s` becomes `"c4"`.

Then we apply the operation on `s[1]`, and `s` becomes `""`.

Constraints:

- $1 \leq s.length \leq 100$
- `s` consists only of lowercase English letters and digits.
- The input is generated such that it is possible to delete all digits.

Task 2 (6 points)

Implement a **last-in-first-out (LIFO)** stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Question link on [LeetCode](#)

Implement the `MyStack` class:

- `void push(int x)` Pushes element `x` to the top of the stack.
- `int pop()` Removes the element on the top of the stack and returns it.
- `int top()` Returns the element on the top of the stack.
- `boolean empty()` Returns `true` if the stack is empty, `false` otherwise.

Notes:

- You must use only standard operations of a queue, which means that only push to back, peek/pop from front, size and is empty operations are valid.
- Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Constraints:

- $1 \leq x \leq 9$
- At most 100 calls will be made to `push`, `pop`, `top`, and `empty`.
- All the calls to `pop` and `top` are valid.

Tasks Submission Guide

You will solve both problems on LeetCode and you have to make sure that your solution passes all test cases and that LeetCode accepts it. Once you are done, its time to submit your solution on GradeScope.

Firstly, make sure to submit **only** two files:

1. TaskOne.java

2. MyStack.java

Secondly, for each question, you have to follow these steps to make sure GradeScope's autograder runs without issues:

For Task 1:

1. Copy your whole solution into a file named `TaskOne.java`
2. Change the class name from `Solution` to `TaskOne`, and make sure that the class is `public`
3. add the following package identifier to the beginning of the file, make sure to copy it and **don't write it manually to avoid typos**:

```
1 | package com.gradescope.labFive;
```

Here's an example solution for reference:

```
1 | package com.gradescope.labFive;
2 |
3 | public class TaskOne {
4 |     public String clearDigits(String s) {
5 |         // your solution here
6 |     }
7 |     /* if any helper methods are needed, make sure to include them */
8 | }
```

For Task 2:

1. Copy your whole solution into a file named `MyStack.java`
2. Make sure that the class is public and named `MyStack`
3. add the following package identifier to the beginning of the file, make sure to copy it and **don't write it manually to avoid typos**:

```
1 | package com.gradescope.labFive;
```

Here's an example solution for reference:

```
1 | package com.gradescope.labFive;
2 |
3 | public class MyStack {
4 |
5 |     public MyStack() {
6 |         // your implementation goes here
7 |     }
8 |
9 |     public void push(int x) {
10 |         // your implementation goes here
11 |     }
12 |
13 |     public int pop() {
14 |         // your implementation goes here
15 |     }
16 |
17 |     public int top() {
18 |         // your implementation goes here
19 |     }
20 |
21 |     public boolean empty() {
22 |         // your implementation goes here
23 |     }
24 | }
```

Happy Coding 😊

Last changed by

Ahmad Albarasy · Follow

0 13

Add a comment

Read more

Data Structures and Algorithms Lab 4 - Recursion

In this lab, we are going to learn about recursion and how can we use it to solve interesting problems. We ar...
Dec 6, 2025

Data Structures and Algorithms Lab 3 - Algorithm Analysis

In this lab, we are going to learn how to analyze our algorithms in terms of time and space complexity, whi...
Dec 3, 2025

Data Structures and Algorithms Lab 2 - Linked Lists

In this lab, we are going to learn about Linked Lists. What they are, how can we implement them, their key...
Dec 3, 2025

Data Structures and Algorithms Lab 1 - Arrays

In this lab, we are going to dive into Arrays, one of the most basic yet fundamental data structures that we us...
Dec 3, 2025

Read more from Ahmad Albarasy

Published on HackMD