

**“AZƏRBAYCAN HAVA YOLLARI” CJSC NATIONAL AVIATION ACADEMY”**

**Individual Work №: 3**

**Topic: Lists. Sets. Stacks. Queues. Time complexity of different operations.**

**Subject: OOP**

**Teacher: Mamed Shahmaliyev**

**Group: 1459i Student: Ilaha Aliyeva**

**Baku 2022**

Data Structures: Stacks and Queues

This post is the fourth in a series on data structures. The topics covered in this series are 6 major data structures that will come up in any kind of software engineering interview:

1. [Hashmaps](https://medium.com/@mariam.jaludi/i-need-to-understand-hashmaps-3ab690c1828e)
2. [Linked Lists](https://medium.com/@mariam.jaludi/data-structures-linked-lists-8f277b61cca)
3. [Trees](https://medium.com/@mariam.jaludi/data-structures-trees-1bafa942cd60)
4. [Stacks and Queues](https://medium.com/@mariam.jaludi/stacks-and-queues-f281aa5462cf)
5. [Heaps](https://medium.com/@mariam.jaludi/data-structures-heaps-b039868a521b)
6. [Graphs](https://medium.com/@mariam.jaludi/data-structures-graphs-50a8a032db03)

**What are Stacks and Queues**

Stacks and Queues are usually explained together because they are similar data structures. Both are linear data structures. The difference between them is how elements are removed.

**Stack**

A Stack is a **LIFO (Last In First Out)**data structure. The last element that is placed in a stack is the first element that can be removed. Elements can be inserted and deleted only from one side of the list, called the **top**.

The insertion of an element into stack is called **pushing.**Deletion of an element from the stack is called **popping**. In stacks, The last element in a list is tracked with a pointer called **top**.

Popping an element from a stack will take O(1) time complexity. Popping the last element in a stack will take O(n).

**Queue**

A Queue is a **FIFO (First In First Out)** data structure. The first element that is placed in a queue is the first one out. Elements can be inserted only from one side of the list called **rear**, and the elements can be deleted only from the other side called the **front**.Think of queues like a queue of people waiting for something. The first person to queue up is the first person served.

The insertion of an element in a queue is called an **enqueue** operation and deleting an element is called a **dequeue** operation. For queues, two pointers are maintained;

1. **Front**pointer: points to the element which was inserted at the first and still present in the list.
2. **Rear** pointer: points to the element inserted last.

Dequeuing the first element takes O(1) time complexity.

**Why are Stacks and Queues Useful?**

Stacks and Queues are commonly used when implementing Breadth-First-Search (BFS) or [Depth-First-Search](https://medium.com/@mariam.jaludi/data-structures-trees-1bafa942cd60) (DFS) for trees and graphs. Queues are commonly used for BFS and Stacks for DFS.

**Stack and Queue Implementation**

Stacks and Queues often have language specific syntax. In Python, lists are usually used to represent stacks. For Queues, there is a collection called deque.

Python’s [deque objects are implemented as doubly-linked lists](https://github.com/python/cpython/blob/947629916a5ecb1f6f6792e9b9234e084c5bf274/Modules/_collectionsmodule.c#L24-L26) which gives them O(1) time complexity for enqueuing and dequeuing elements, but O(n)time complexity for randomly accessing elements in the middle of the queue.

Because deques support adding and removing elements from either end equally well, you can actually use them for both queues and stacks.

Let’s create a stack and queue and see how we operate on them:

from collections import deque  
s1 = [4, 5, 6, 7]  
q1 = deque([2, 3, 4, 5, 6])

We’ve created the stack s1 and queue q1.

lets **push** a value to our stack s1:

s1.append(5)  
print(s1)  
=> [4, 5, 6, 7, 5]

Now let’s **pop** a value from stack s1:

s1.pop()  
=> 5  
print(s1)  
=> [4, 5, 6, 7]

Because 5 was the last value we pushed to our stack, it was the first value popped out, following **LIFO**.

Now let’s look at our queue. To implement a deque object as **FIFO** we will append (**enqueue**) from the left of our queue.

q1.appendleft(1)  
q1  
=> deque([1, 2, 3, 4, 5, 6])

let’s now dequeue a value from our queue:

q1.pop()  
=> 6  
q1  
=> deque([1, 2, 3, 4, 5])

And that is how you can implement a stack and queue in Python.

**Common Algorithms Related to Stacks and Queues**

**Problem 1: Use a stack to check if a string has balanced usage of parentheses**

Ex:

(), (()), ()(), (((()))) → Balanced)), ((), ((((()), ()()) → Not Balanced

To solve this, we are going to loop through the given string. If we encounter an opening parenthesis, we push it onto our stack. If we encounter a closing parenthesis, we pop from our stack. If we encounter a closing parenthesis and there is nothing to pop from our stack, we know that the string is unbalanced. If we have completed looping through our string and our stack is not empty, we know that the string is unbalanced.

def isBalanced(str):  
 s = []  
 is\_balanced = True  
 i = 0  
 while i < len(str) and is\_balanced:  
 paren = str[i]  
 if paren == "(":  
 s.append(paren)  
 elif paren == ")":  
 if s:  
 s.pop()  
 else:  
 is\_balanced = False  
 i += 1  
 if s:  
 is\_balanced = False   
 return is\_balanced

**Problem 2: Breadth First Search using a queue**

Breadth First Search involves searching through a tree one level at a time. If you would like to learn about trees and how they are implement in code, check out the [previous post](https://medium.com/@mariam.jaludi/data-structures-trees-1bafa942cd60).

We traverse through one entire level of children nodes first, before moving on to traverse through the grandchildren nodes. And we traverse through an entire level of grandchildren nodes before going on to traverse through great-grandchildren nodes.

To implement this, we start with our root node in the tree:

1. Our root node is our current node. We initialize a deque object with it.
2. We dequeue a node from our queue print the value of that node.
3. Enqueue the reference to its left child and right child.
4. We move on to the next value in the queue and set that as the current node.
5. Continue until our queue is empty.

# bfs method is part of the Node class for the Tree data structuredef bfs(self):  
 q = deque([self])  
 while q:  
 val = q.pop()  
 print(val.value)  
 if val.left:  
 q.appendleft(val.left)  
 if val.right:  
 q.appendleft(val.right)

It is pretty simple. All we’re doing here is using a while loop to continue to dequeue a node, print it, adding its left child, and adding its right child. We continue iterating through the queue until everything has been removed from it.