**Queues**

# Queues as Abstract Data Types (ADT): *focus on what, not how*

Queues are referred to as abstract data types (ADTs) for several key reasons:

1. **Focus on behavior, not implementation**: An ADT defines what a data structure does without specifying how it's done. This allows for flexibility in implementing the queue in different ways (e.g., using arrays, linked lists) while still adhering to the same core behavior. This separation of concern simplifies reasoning about the program and makes code more reusable.
2. **Abstraction hides complexity**: An ADT hides the implementation details of the queue, like memory management and pointer manipulation. This makes the code easier to understand and write, as programmers can focus on the logical operations of the queue (adding, removing, checking size) without worrying about the underlying mechanics.
3. **Data independence**: Queues are generic data structures, meaning they can store any type of data (integers, strings, objects) without needing to be modified. This generality is captured by the abstract data type concept, which focuses on the behavior independent of the specific data elements.
4. **Promotes code reuse and interoperability**: ADTs like queues can be implemented in different languages and libraries while still adhering to the same core behavior. This allows programmers to easily reuse queue functionality across different parts of a program or even different programs altogether.
5. **Encourages clear thinking about data structures**: By focusing on the essential operations and properties of a queue, ADTs encourage programmers to think more deeply about the concepts behind data structures and how they can be effectively used in different situations.

# ADTs As Blueprints of a Data Structure

ADT is like a blueprint for a data structure, specifying what it can do and how it should behave, without getting bogged down in the specific details of how it's built. It's like focusing on the recipe for a delicious cake (what ingredients, what steps) rather than the specific tools or oven you use to bake it.

**Here's an analogy to illustrate the point:**

Imagine you're building a house. You wouldn't start by focusing on the exact type of bricks, the brand of nails, or the specific tools the construction workers will use. Instead, you'd first define what the house should look like - the number of rooms, the layout, the basic functionalities. This is similar to an ADT, which defines the core functionalities of a data structure like a queue (adding, removing, checking size) without specifying the underlying implementation.

The actual construction (implementation) can be done using different materials and techniques (arrays, linked lists), just like the same cake recipe can be baked in different ovens with variations in tools. The focus is on the desired outcome (a functioning house/queue) rather than the specific methods used to achieve it.

# Queues: The FIFO Champions

Now, let's apply this to queues. A queue ADT specifies operations like adding items (enqueuing), removing them (dequeuing), checking the size, and determining if it's empty. It doesn't tell you how the queue keeps track of its elements or how new items are inserted. This is the beauty of the ADT – it captures the essence of a first-in-first-out (FIFO) data structure without getting bogged down in the implementation details.

By focusing on the queue ADT, you can reason about your program's logic without worrying about the underlying mechanics. You can easily swap between different queue implementations (array-based, linked list-based) depending on your needs, and the core behavior remains consistent. This abstraction makes queue operations clear, concise, and reusable.

# The Power of Abstraction

Think of ADTs like a bridge between the conceptual world of data structures and the practical world of implementation. They provide a common language for understanding and working with different data structures like queues, stacks, trees, and more. This abstraction empowers programmers to build robust and efficient software, focusing on the "what" while leaving the "how" to the chosen implementation.

# Basic Operations:

* **Enqueue**: Add an element to the rear (back) of the queue. This is the primary operation for adding data to the queue.
* **Dequeue**: Remove the element from the front (head) of the queue. This retrieves the oldest element that has been waiting the longest.
* **Peek or first**: Access the element at the front of the queue without removing it. This allows checking the next element to be dequeued without modifying the queue itself.
* **IsEmpty**: Check if the queue contains any elements. This helps determine if the queue is empty and avoid errors when attempting to dequeue.
* **IsFull**: Check if the queue has reached its maximum capacity (if applicable). This is relevant for some implementations with fixed sizes.
* **Size**: Get the current number of elements in the queue. This provides information about the queue's fullness and can be used for various purposes.

## Additional Operations:

* **Clear**: Remove all elements from the queue, resetting it to an empty state.
* **Search**: Find a specific element within the queue without modifying the order.
* **Rotate**: Shift the elements in the queue by a specified number of positions.
* **Merge**: Combine two or more queues into a single queue, maintaining the FIFO order.

Variations

* **Priority Queues**: Allow elements to be prioritized based on a certain value, affecting the dequeue order.
* **Double-ended Queues (Deques):** Support adding and removing elements from both ends of the queue, offering more flexibility.

A screenshot of a table

Description automatically generated

# Dequeue Operation: O(1) vs O(n) time

Python's built-in **pop(0)** method offers convenient queue access, but its hidden cost is performance. Every dequeue triggers a costly "element shift," where remaining elements move left, taking O(n) time as the queue grows. This bottleneck slows down your code, especially with large queues.

We can optimize this time complexity by using a variable to keep track of the first non None index

1. **Replace elements with None**: Instead of shifting, it simply marks the dequeued element at index 0 as empty with a single, fast None assignment. This takes only O(1) time, independent of queue size.
2. **Track the next index**: It keeps track of the next element's position using a variable, avoiding costly searches and shifts. Accessing the next element becomes another O(1) operation, blazingly fast regardless of queue size.

## Drawbacks:

**Memory Usage:** The use of **None** to mark dequeued elements can lead to memory wastage, especially if the queue experiences a high turnover of enqueues and dequeues. **Turnover** here means that we repeatedly enqueue a new element and then dequeue another (allowing the front to drift rightward).Over time, the list may accumulate a significant number of **None** values.

Over time, the size of the underlying list would grow to *O*(*m*) where *m* is the ***total*** number of enqueue operations since the creation of the queue, rather than the current number of elements in the queue. This design would have detrimental consequences in applications in which queues have relatively modest size (queue is not excessively large), but which are used for long periods of time. For example, the waitlist for a restaurant might never have more than 30 entries at one time, but over the course of a day (or a week), the overall number of entries would be significantly larger.

## Circular Queue: for relatively modest sized queues

If you are implementing a circular queue and once the queue is full, you reset the index back to 0 when a dequeue operation takes place, it indeed mitigates the issue mentioned in the statement.

In a circular queue implementation, resetting the index allows you to reuse the space efficiently, preventing the underlying list from growing indefinitely. This approach helps maintain a relatively constant memory footprint, even in scenarios where the queue experiences a high turnover of elements.

The strategy of resetting the index when a dequeue operation reaches the end of the underlying list ensures that the space is reused, and the size of the underlying list remains bounded, avoiding the issue of growing to O(m), where m is the total number of enqueue operations. This is a valid technique to manage the memory efficiently in scenarios where the queue size is relatively modest but experiences a high turnover.

The expression **f = (f + 1) % N** is commonly used in the context of a circular buffer, which is an efficient way of implementing a queue using a fixed-size array.

Here's what each part of the expression means:

* **f**: This is typically the index of the front of the queue.
* **f + 1**: This moves the index **f** forward by one position.
* **N**: This is the size of the array or the maximum number of elements the queue can hold.
* **%**: This is the modulo operator, which gives the remainder of the division of **f + 1** by **N**.

The whole expression, **f = (f + 1) % N**, ensures that **f** wraps around to the beginning of the array when it reaches the end. This is what makes the buffer "**circular**"—instead of going out of bounds when you reach the end of the array, you loop back to the beginning.

## In the Context of Enqueueing Operations

When enqueueing (adding) an element to the queue:

1. You place the new element at the "rear" index of the queue.
2. You increment the "rear" index by one, using a similar expression: **rear = (rear + 1) % N**.

This makes sure that if the rear index reaches the end of the array, it loops back to the start, utilizing the array space that gets freed as items are dequeued from the front.

## In the Context of Dequeueing Operations

When dequeueing (removing) an element from the queue:

1. You retrieve the element at the "front" index.
2. You increment the "front" index using **f = (f + 1) % N**.

By doing this, you effectively move the front of the queue to the next element. If you have reached the end of the array, the front will loop back to the beginning.

**Example**

If **N** is 5 and **f** is currently 3, after the operation **f = (f + 1) % N**, **f** would be 4. The next time the operation is performed, **f** would become **(4 + 1) % 5**, which is 0, effectively wrapping around the array.

This circular buffer technique is especially useful in situations where you have a fixed amount of memory and want to make efficient use of space without needing to resize the underlying array or shift elements.

# **Double Ended Queue (Deque)**

The **deque** (short for "double-ended queue") class in Python is a highly optimized container that provides fast, efficient operations at its endpoints. However, its performance characteristics change when dealing with elements near or at the middle.

For example, we described a restaurant using a queue to maintain a waitlist. Occasionally, the first person might be removed from the queue only to find that a table was not available; typically, the restaurant will re-insert the person at the *first* position in the queue. It may also be that a customer at the end of the queue may grow impatient and leave the restaurant

## **Fast Operations at Either End (O(1) Time Complexity)**

Operations like **append**, **appendleft**, **pop**, and **popleft** are O(1) operations in a **deque**. This is because a **deque** is implemented using a doubly linked list, which allows it to add or remove elements at either end in constant time, regardless of the size of the deque.

**Slow Operations with Index Notation Near the Middle (O(n) Time Complexity)**

However, if you try to access or modify an element using index notation (like **dq[2]**), especially near the middle of the deque, it becomes an O(n) operation. This is because the deque needs to traverse from one end to reach the indexed position.

## **Why is it O(n)?**

* A **deque** does not support direct access to the middle elements in constant time.
* To access an element at index **i**, it traverses from the nearest end (either from the start or the end, whichever is closer) to that index.
* In the worst case, if you're accessing an element near the middle, it may need to traverse approximately **n/2** elements (where **n** is the total number of elements in the deque). This traversal is linear in nature, hence the O(n) time complexity.
* When you access an element by index(i.e. not the front or rear but somewhere in the middle) in a **deque** in Python, it traverses from the nearest end (either front or rear) to reach the specified index. This traversal is what results in the O(n) time complexity for such operations.

## **Why Access Elements in the Middle?**

1. **Versatility in Programming**: Sometimes, a programmer might choose a deque for its strengths (efficient front and rear operations) but later find the need to access or modify an element in the middle. This could be due to evolving requirements or unforeseen use cases in the code.
2. **Lack of Awareness**: A programmer might not be fully aware of the performance implications of accessing elements in the middle of a deque. They might use a deque as a general-purpose container and apply list-like operations without considering the efficiency.
3. **Convenience**: In some cases, the ease of using a familiar syntax (like indexing) might lead a developer to treat a deque more like a list, even if it's not the most efficient approach.

## **Why Use Indexing on a Deque?**

1. **Python's Flexibility**: Python allows indexing on a deque, even though it's not the most efficient operation. This is part of Python's philosophy of being easy to use and flexible, allowing programmers to choose how they want to use a data structure, even if it's not the optimal way.
2. **Temporary or Small-Scale Use**: In cases where the deque is small, or the operation is a one-off, the inefficiency of using indexing might not be significant enough to impact overall performance, so a programmer might opt for the simplicity of indexing.
3. **Transitioning from Lists**: Some programmers might transition from using lists to deques for certain functionalities but continue to use familiar list operations out of habit or convenience.

**Time Complexity**

The time complexity of various operations on a deque is one of its most significant aspects:

1. **Appending and Popping**:
   * **append** (add to the right end) and **appendleft** (add to the left end) operations are O(1).
   * **pop** (remove from the right end) and **popleft** (remove from the left end) are also O(1).
2. **Accessing Elements**:
   * Accessing elements at the ends (**[0]**, **[-1]**) is O(1).
   * However, accessing elements in the middle is O(n), as it requires traversal from the nearest end.
3. **Inserting and Removing Elements**:
   * Inserting or removing elements in the middle is also O(n), due to the need to shift other elements to fill or make space.

**Space Complexity**

The space complexity of a **deque** is O(n), where n is the number of elements in the deque. It is designed to provide both efficient memory usage and performance for its intended operations. Unlike a list, a **deque** doesn’t over-allocate memory for future growth, which can make it more memory-efficient, especially when the size fluctuates frequently.

**Operations and ADTs (Abstract Data Types)**

As an ADT, a **deque** supports various operations, making it a flexible and useful data structure:

1. **Double-Ended Operations**: As the name implies, it allows adding and removing elements from both the front and back, supporting stack-like (LIFO) and queue-like (FIFO) behaviors.
2. **Indexing and Slicing**: While not as efficient as in lists, **deque** does support indexing and slicing, which can be useful in certain scenarios.
3. **Length and Count**: You can easily find the number of elements (**len(deque)**) and count the occurrences of a specific element.
4. **Rotation**: Deque supports rotation, which means you can rotate the deque n steps to the right (positive n) or left (negative n).
5. **Reversing**: The **reverse()** method allows you to reverse the elements in the deque.
6. **Max Length**: When initializing a deque, you can specify a maximum length. If the deque reaches the specified maximum length, it will discard elements from the opposite end when new elements are added.
7. **Thread-Safe**: Deque is thread-safe for append and pop operations from opposite ends but not for operations in the middle or iterating through it.

**Use Cases**

Due to its characteristics, a **deque** is ideal for:

* Implementing queues and stacks where elements are frequently added or removed from the ends.
* Applications where memory efficiency and consistent run-time performance are important.
* Scenarios where occasional random access or modification of elements is required, but not the primary use case.

## **Deque ADT Operations**

|  |  |
| --- | --- |
| **Method** | Description |
| dq.add\_first(e) | Add element **e** to the front of deque **dq**. |
| **dq.add\_last(e)** | Add element **e** to the back of deque **dq**. |
| **dq.delete\_first()** | Remove and return the first element from deque **dq**; an error occurs if the deque is empty. |
| **dq.delete\_last()** | Remove and return the last element from deque **dq**; an error occurs if the deque is empty. |
| **dq.first()** | Return (but do not remove) the first element of deque **dq**; an error occurs if the deque is empty. |
| **dq.last()** | Return (but do not remove) the last element of deque **dq**; an error occurs if the deque is empty. |
| **dq.is\_empty()** | Return **True** if deque **dq** does not contain any elements. |
| **len(dq)** | Return the number of elements in deque **dq**; in Python, this is implemented with the special method **\_\_len\_\_**. |