# Fundamental Data Structures and Algorithms 04 - Array, Linked List, Stack and Queue


**Fundamental Data Structures and Algorithms 04 - Array, Linked List, Stack and Queue**

- Unit 3: Basic Data Structures
  - 3.1 Data Types and Data Structures
  - 3.2 Array
    - 3.2.1 Array with Python Lists
    - 3.2.2 Array with NumPy Library
  - 3.3 Linked List
    - 3.3.1 Linked List Implementation
    - 3.3.2 Arrays, Lists & Linked Lists
  - 3.4 Stack
    - 3.4.1 Stack Implementation using List
    - 3.4.2 Stack Implementation using Linked List
  - 3.5 Queue
    - 3.5.1 Queue Implementation using Circular Array

# Unit 3: Basic Data Structures

For many problems, the ability to formulate an efficient algorithm depends on being able to organize the data in an appropriate manner. A **data structure** can be defined to be the way data is stored and organized in a computer. These notes will look at numerous data structures ranging from familiar *arrays* and *lists* to more complex structures such as *trees* and *graphs*, and we will see how their choice affects the efficiency of the algorithms based upon them.

To fully understand data structures and algorithms you will almost certainly need to complement the introductory material in these notes with textbooks or other sources of information. The lectures associated with these notes are designed to help you understand them and fill in some of the gaps they contain, but that is unlikely to be enough because often you will need to see more than one explanation of something before it can be fully understood.

There is no single best textbook that will suit everyone. The subject of these notes is a classical topic, so there is no need to use a textbook published recently. Books published 10 or 20 years ago are still good, and new good books continue to be published every year. The reason is that these notes cover important fundamental material that is taught in all university degrees in computer science. These days there is also a lot of very useful information to be found on the internet, including complete freely-downloadable books. It is a good idea to go to your library and browse the shelves of books on data structures and algorithms. If you like any of them, download, borrow or buy a copy for yourself, but make sure that most of the topics in the above contents list are covered. Wikipedia is generally a good source of fairly reliable information on all the relevant topics, but you hopefully should not need reminding that not everything you read on the internet is necessarily true. It is also worth pointing out that there are often many different equally-good ways to solve the same task, different equally-sensible names used for the same thing, and different equally-valid conventions used by different people, so don't expect all the sources of information you find to be an exact match with each other or with what you find in these notes.

We will begin by differentiating data structures from data types before moving on to some common data structures in detail.

## 3.1 Data Types and Data Structures

In the early chapters of many programming languages, many of us will be introduced to the basic data types that are defined in that particular language and eventually some basic data structures. In fact, you should have already seen some of these data types and data structures. Some examples of data types include integers, floats and strings and some examples of data structures include lists, tuples, sets and dictionaries. Data structures should not be confused with data types. While they sound similar, there is a difference between the two terms. So what are the differences? Below, we categorically identify the properties that determine whether an object is a data type or data structure.

| Property | Data Type | Data Structure |
|---|---|---|
| Definition | Data type is the representation of nature and type of data that has been going to be used in programming or in other words data type describes all that data which share a common property. For example an integer data type describes every integer that the computers can handle. | On other hand Data structure is the collection that holds data which can be manipulated and used in programming so that operations and algorithms can be more easily applied. For example tree type data structures often allow for efficient searching algorithms. |
| Implementation | Data type in programming are implemented in abstract implementation whose definition is provided by different languages in different ways. | On other hand Data structure in programming are implemented in concrete implementation as their definition is already defined by the language that what type of data they going to store and deal with. |
| Storage | In case of data type the value of data is not stored as it only represents the type of data that can be get stored. | On other hand data structure holds the data along with its value that actually acquires the space in main memory of the computer. Also data structure can hold different kind and types of data within one single object |
| Assignment | As data type already represents the type of value that can be stored so values can directly be assigned to the data type variables. | On other hand in case of data structure the data is assigned to using some set of algorithms and operations like push, pop and so on. |
| Performance | If case of data type only type and nature of data is concern so there in no issue of time complexity. | On other hand time complexity comes in case of data structure as it mainly deals with manipulation and execution of logic over data that it stored. |

## 3.2 Array

You should have seen how Python provides some quite useful and flexible general data structures. In particular, *list* objects can be considered a real workhorse with many convenient characteristics and application areas. However, scientific and financial applications generally have a need for high-performing operations on special data structures. One of the most important data structures in this regard is the *array*. Arrays generally structure other (fundamental) objects in rows and columns. In this topic, we will revisit NumPy arrays in the context of data structures.

Assume for the moment that we work with numbers only, although the concept generalizes to other types of data as well. In the simplest case, a one-dimensional array then represents, mathematically speaking, a *vector* of, in general, real numbers, internally represented by *float* objects. It then consists of a *single* row or column of elements only. In a more common case, an array represents an $(i \times j)$ matrix of elements. This concept generalizes to $(i \times j \times k)$ cubes of elements in three dimensions as well as to general *n*-dimensional arrays of shape $(i \times j \times k \times l \times \ldots)$ .

Mathematical disciplines like linear algebra and vector space theory illustrate that such mathematical structures are of high importance in a number of disciplines and fields. It can therefore prove fruitful to have available a specialized class of data structures explicitly designed to handle arrays conveniently and efficiently. This is where the Python library *NumPy* comes into play, with its `ndarray` class.

### 3.2.1 Array with Python Lists

Before we turn to `NumPy`, let us first construct arrays with the built-in data structures presented in the previous section. *list* objects are particularly suited to accomplishing this task. A simple *list* can already be considered a one-dimensional array:

```
v = [0.5, 0.75, 1.0, 1.5, 2.0]  # vector of numbers
```

Since *list* objects can contain arbitrary other objects, they can also contain other *list* objects. In that way, two- and higher-dimensional arrays are easily constructed by nested *list* objects:

```
m = [v, v, v]
m
```

This will output:

```
[[0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0]]
```

We can also easily select rows via simple indexing or single elements via double indexing (whole columns, however, are not so easy to select).

Example, `m[1]` gives us `[0.5, 0.75, 1.0, 1.5, 2.0]`,

while `m[1][0]` gives us `0.5`.

Nesting can be pushed further for even more general structures:

```
v1 = [0.5, 1.5]
v2 = [1, 2]
m = [v1, v2]
c = [m, m]  # cube of numbers
c
```

Output: `[[[0.5, 1.5], [1, 2]], [[0.5, 1.5], [1, 2]]]`

Similarly, selecting nested index, we can use `c[1][1][0]` to get an output of `1`

Note that combining objects in the way just presented generally works with reference pointers to the original objects. What does that mean in practice? Let us have a look at the following operations:

```
v = [0.5, 0.75, 1.0, 1.5, 2.0]
m = [v, v, v]
m
```

Output:

```
[[0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0]]
```

Now, change the value of the first element of the `v` object and see what happens to the `m` object:

```
v[0] = 'Python'
m
```

Output:

```
[['Python', 0.75, 1.0, 1.5, 2.0],
 ['Python', 0.75, 1.0, 1.5, 2.0],
 ['Python', 0.75, 1.0, 1.5, 2.0]]
```

This can be avoided by using the `deepcopy` function of the `copy` module:

```
from copy import deepcopy
v = [0.5, 0.75, 1.0, 1.5, 2.0]
m = 3 * [deepcopy(v), ]
m
```

Output:

```
[[0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0]]
```

Now, let's see the difference:

```
v[0] = 'Python'
m
```

Output is unchanged:

```
[[0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0]]
```

### 3.2.2 Array with NumPy Library

Obviously, composing array structures with *list* objects works, somewhat. But it is not really convenient, and the *list* class has not been built with this specific goal in mind. It has rather been built with a much broader and more general scope. From this point of view, some kind of specialized class could therefore be really beneficial to handle array-type structures.

Such a specialized class is `numpy.ndarray`, which has been built with the specific goal of handling *n*-dimensional arrays both conveniently and efficiently — i.e., in a highly performing manner. The basic handling of instances of this class is again best illustrated by examples:

```
import numpy as np
a = np.array([0, 0.5, 1.0, 1.5, 2.0])
```

We can determine the type by using `type(a)`. This gives us: `numpy.ndarray`,

and using `a[:2]` give all elements from the zeroth index to first index i.e. `array([0. , 0.5])`

A major feature of the `numpy.ndarray` class is the multitude of built-in methods. For instance, you can invoke `a.sum()` and `a.std()`, to calculate the sum and standard deviations of the array.

Invoking `a.sum()` gives us `5.0`, and invoking `a.std()` gives us `0.7071067811865476`.

Another major feature is the (vectorized) mathematical operations defined on `ndarray` objects:

`a*2` is a scalar multiplication of the array by a factor of 2 which gives us :

```
array([ 0.,  1.,  2.,  3.,  4.])
```

while `a**2` is the raising each element to a factor of 2 to give us :

```
array([ 0.  ,  0.25,  1.  ,  2.25,  4.  ])
```

We can also find the square root of each element by using `np.sqrt(a)` equating to:

```
array([ 0.        ,  0.70710678,  1.        ,  1.22474487,  1.41421356])
```

The transition to more than one dimension is seamless, and all features presented so far carry over to the more general cases. In particular, the indexing system is made consistent across all dimensions:

```
b = np.array([a, a * 2])
b
```

Output:

```
array([[ 0. ,  0.5,  1. ,  1.5,  2. ],
       [ 0. ,  1. ,  2. ,  3. ,  4. ]])
```

We can call the first row using: `b[0]` which gives us an output :

```
array([ 0. ,  0.5,  1. ,  1.5,  2. ])
```

and we can call the third element of the first row using: `b[0, 2]` which is simply `1.0`

In contrast to our *list* object-based approach to constructing arrays, the `numpy.ndarray` class knows axes explicitly. Selecting either rows or columns from a matrix is essentially the same:

To sum the the elements across rows we use, `b.sum(axis=0)` to give us :

```
array([ 0. ,  1.5,  3. ,  4.5,  6. ])
```

and across columns `b.sum(axis=1)` to give us :

```
array([  5.,  10.])
```

There are a number of ways to initialize (instantiate) a `numpy.ndarray` object. One is as presented before, via `np.array`. However, this assumes that all elements of the array are already available. In contrast, one would maybe like to have the `numpy.ndarray` objects instantiated first to populate them later with results generated during the execution of code. To this end, we can use the following functions:

```
c = np.zeros((2, 3, 4), dtype='i', order='C')  # also: np.ones()
c
```

Output:

```
array([[[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],

       [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]]], dtype=int32)
```

With all these functions we provide the following information:

- `shape`

  Either an *int*, a sequence of *int* , or a reference to another `numpy.ndarray`

- `dtype` **(optional)**

  A `numpy.dtype` — these are *NumPy*-specific data types for `numpy.ndarray` objects

Here, it becomes obvious how *NumPy* specializes the construction of arrays with the `numpy.ndarray` class, in comparison to the *list*-based approach:

- The shape/length/size of the array is *homogenous* across any given dimension.
- It only allows for a *single* data type (`numpy.dtype`) for the whole array.

The table below provides an overview of `numpy.dtype` objects (i.e., the basic data types *NumPy* allows).

| dtype | Description | Example |
|---|---|---|
| `t` | Bit field | `t4` (4 bits) |
| `b` | Boolean | `b` (true or false) |
| `i` | Integer | `i8` (64 bit) |
| `u` | Unsigned integer | `u8` (64 bit) |
| `f` | Floating point | `f8` (64 bit) |
| `c` | Complex floating point | `c16` (128 bit) |
| `O` | Object | `0` (pointer to object) |
| `S`, `a` | String | `S24` (24 characters) |
| `U` | Unicode | `U24` (24 Unicode characters) |
| `V` | Other | `V12` (12-byte data block) |

*NumPy* provides a generalization of regular arrays that loosens at least the `dtype` restriction, but let us stick with regular arrays for a moment and see what the specialization brings in terms of performance.

As a simple exercise, suppose we want to generate a matrix/array of shape 5,000 × 5,000 elements, populated with (pseudo)random, standard normally distributed numbers. We then want to calculate the sum of all elements. First, the pure Python approach, where we make use of list comprehensions and functional programming methods as well as lambda functions. Note that you are not required to understand how list comprehensions and lambda functions work at this point, as the objective is to demonstrate the performance only:

```
1
2 import random, functools
3 I = 2000
```

```
# pure Python method
# generate I x I matrix mat
print("time taken to generate square matrix of size I:")
%time mat = [[random.gauss(0, 1) for j in range(I)] for i in range(I)]
print("\ntime taken to sum all elements in the matrix:")
# sum of all elements in matrix mat
%time functools.reduce(lambda x, y: x + y,      \
               [functools.reduce(lambda x, y: x + y, row) \
                       for row in mat])
```

Output:

```
time taken to generate square matrix of size I:
Wall time: 2.15 s

time taken to sum all elements in the matrix:
Wall time: 262 ms
```

Let us now turn to `NumPy` and see how the same problem is solved there. For convenience, the `NumPy` sub-library `random` offers a multitude of functions to initialize a `numpy.ndarray` object and populate it at the same time with (pseudo)random numbers:

```
# NumPy array method
# generate I x I matrix mat
print("time taken to generate square matrix of size I:")
%time mat = np.random.standard_normal((I, I))
# sum of all elements in matrix mat
print("\ntime taken to sum all elements in the matrix:")
%time mat.sum()
```

Output:

```
time taken to generate square matrix of size I:
Wall time: 145 ms

time taken to sum all elements in the matrix:
Wall time: 4.59 ms
```

We observe the following:

- **Syntax**

  Although we use several approaches to compact the pure `Python` code, the *NumPy* version is even more compact and readable.

- **Performance**

  The generation of the `numpy.ndarray` object is roughly 15 times faster and the calculation of the sum is roughly 60 times faster than the respective operations in pure *Python* (values might be different depending on system hardware).

We see that the use of NumPy *arrays* generally result in compact, easily readable code and significant performance boost over pure Python *list*.

## 3.3 Linked List

A **linked list** is a data structure in which the objects are arranged in a linear order.

We should all be familiar with the train and its structure. It is made of a single row of cabins, each connected to adjacent cabins. A linked list has a very similar structure to that of the train in that it is made up of a series of *nodes* (cabins) connected linearly. In order to reach the end of the linked list *none*, you have to start from the *head* and pass through each node. Each node consists of two data fields:

1. `data` - containing the data to be stored in the node
2. `next` - containing the reference to the next node in the list. This is also represented by using arrows when visualizing linked list.

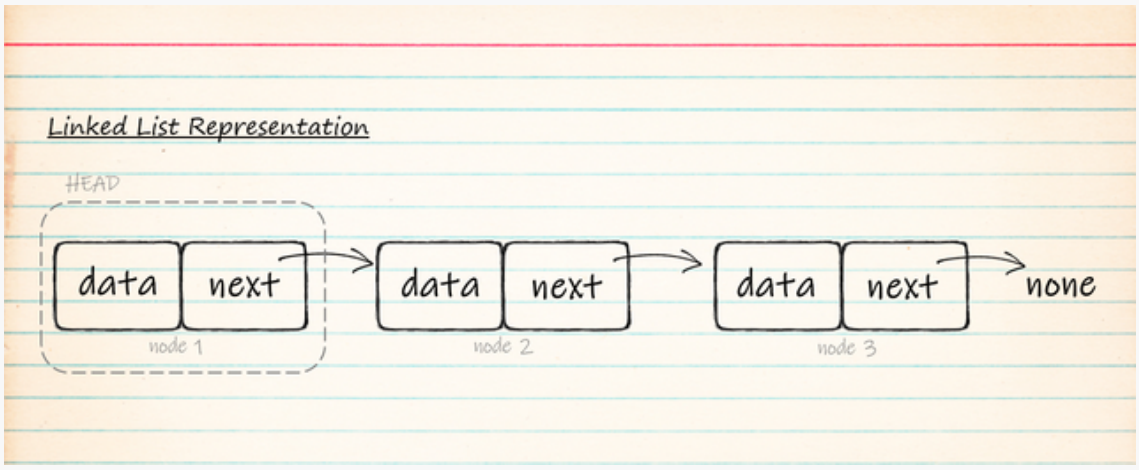


Fig 3.1. Visual representation of linked list

### 3.3.1 Linked List Implementation

We will know implement linked list in Python. First, we have to define the `node` class.

```
class Node:
    # constructor
    def __init__(self, data):
        self.data = data
        self.next = None
```

Once we have the Node class, we can implement any *linked list* as follows:

```
node1 = Node("This is the first node and also the head node.")
node2 = Node("This is the second node. I can basically put any data I want.")
node3 = Node(43)
node4 = Node(1.618)
node5 = Node(Node("this is a node inside a node"))
```

Then, we can link it up using the following:

```
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5
```

We can now display the entire linked list, by passing the head node as the argument:

```
def printLinkedList(node):
    while node is not None:
        print(node.data)
        node = node.next
    print()

printLinkedList(node1)
```

The results should be similar to the following:

```
this is the first node and also the head node.
this is the second node. I can basically put any data I want.
43
1.618
<__main__.Node object at 0x00000164E7589A30>
```

You would have noticed that `node5` prints out a weird output as shown in the last line (the numbers may look different for different systems) but do not fret as this is to be expected. If we try to print `node5` in particular, `print(node5.data)` , the output will still be the same as above.

However, if we used the following syntax, `print((node5.data).data)`, we will get the correct output:

```
this is a node inside a node
```

The reason being is that the *data* that is stored in `node5` is actually the memory address of the nested node.

To delete a node, for example `node3`, we can do a simple redirection of the `next` values:

```
# changing the memory address that is stored 'next' field in node2 and node3
node2.next = node4
node3.next = None
```

There are various ways of implementing *linked list* in Python, the above example is a simplified version to demonstrate how *linked list* works. You may find many other resources that implements the *linked list* class and various *methods* such as node insertion, node search, node deletion, etc. You may also find a *doubly linked list*, *circular linked list* or even *recursive linked list*. You are highly encouraged to visit these different topics on your own.

### 3.3.2 Arrays, Lists & Linked Lists

*Linked lists* differ from *lists* and *arrays* (NumPy arrays) in the way that they store elements in memory. For comparison, take a look at Fig 1.6 for how a typical *array* is represented and compare it with the visual representation of a *linked list* in Fig 3.1. While *lists* and *arrays*, use a contiguous memory block to store references to their data, *linked lists* store references as part of their own elements. This has several implications:

1. It is slower to access a particular node within a *linked list* than it is to access an element within a *list* or *array*. This is due to the fact that in a *linked list*, we need to visit every node between the head to reach a given node while in a *list* or an *array*, we just require an index. As such, the worst-case time complexity to access a node in a *linked list* is $O(n)$.

2. Modifying a *linked list* such as inserting or deleting a node is more efficient than it is to create add an element within an *array*. For example, an *array* is implemented in such a way that in order to insert an element, we need to create a new *array* that is 1 size bigger than the previous, and then copying the values over to the new array including the new element to be inserted before deleting the old array. This entire process has a lot of memory overhead especially if the size of the array is large. In contrast, a linked list just requires a redirection of the *next* value of the node that is position before the inserted node. *List* in Python on the other hand falls in between an *array* and a *linked list*. It has very a similar structure to that of an *array* in that memory is stored contiguously but with the flexibility of *linked list* to expand and reduce the size of the *linked list* with much less memory overheads.

## 3.4 Stack

A **stack** is a data structure that we should be familiar with. We can think of a stack like a deck of cards or a stack of papers, where pieces are accessed from the top. It is a linear data structure that implements a First-In-Last-Out (FILO), or Last-In-First-Out (LIFO), strategy. This means that nodes are added or removed only at the end, or more commonly referred to as the *top of the stack*. On an abstract level, it is equivalent to linked lists.

Two of the most commonly used methods in a *stack* data structure is the `push` and `pop`:

1. `push(data)` - adding a node containing the *data* to the top of the stack
2. `pop()` - removing and returning the top (if any) node of the stack. The next item becomes then becomes the new top.

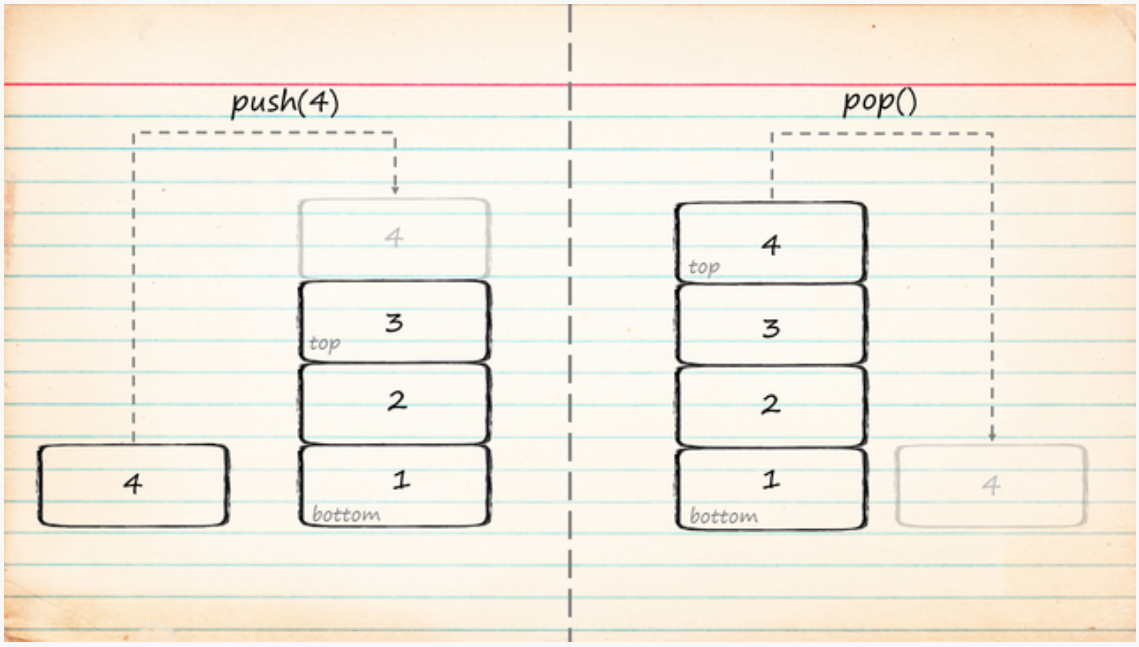


Fig 3.2. Push and Pop on a Stack data structure

*Stack* can be implemented in several ways. The two most common of Python implementations include using *list* and a *linked list*. The choice depends on the application required.

### 3.4.1 Stack Implementation using List

We can demonstrate the *stack* implementation using *list* by using a simple example.

To initialize the *stack*, we simple declare an empty list to a variable, `s`.

```
1 s = []
```

Also, note that the built-in *list* data structure uses `append()` instead of `push()`. Therefore, we can add items to the stack by using the following:

```
1 s.append('this enters the stack first')
2 s.append('I am next!')
3 s.append('I am last.')
```

To verify, a simple `print(s)` will suffice. The output will be:

```
['this enters the stack first', 'I am next!', 'I am last.']`
```

Now, if we *pop* the *stack* using `s.pop()`. The method will return the last element:

```
'I am last.'
```

`print(s)` will now be :

```
['this enters the stack first', 'I am next!']
```

### 3.4.2 Stack Implementation using Linked List

To implement a *stack* using linked list, we first need to define a `StackNode` class similar to the *node* class as shown earlier in the *linked list* topic:

```
class _StackNode:
    def __init__(self, data, next):
        self.data = data
        self.next = next
```

We then define the `Stack` class and various methods including the `push()` and `pop()` methods, and various other methods:

```
class Stack:
    # initializes an empty stack
    def __init__(self):
        self._top = None
        self._size = 0

    # returns True if the stack is empty
    def isEmpty(self):
        return self._top is None

    # returns the number of elements in the stack
    def __len__(self):
        return self._size

    # return the top data
    def peek(self):
        if self.isEmpty():
            return None
        return self._top.data

    # adds an element to the top of the stack
    def push(self, data):
        self._top = _StackNode(data, self._top)
        self._size += 1
```

```
    # removes and returns the element from the top of the stack
    def pop(self):
        if self.isEmpty():
            print ("Stack is empty")
            return
        node = self._top
        self._top = self._top.next
        self._size -= 1
        return node.data
```

Each *stack* instance maintains two variables namely, the reference to the top element of the stack given by `self._top` and the current number of elements in the stack using the `self._size` variable.

We can create a new *stack* object *s* by declaring :

```
s = Stack()
```

We can then *push* elements onto the *stack*. For example,

```
s.push('this enters the stack first')
s.push('I am next!')
s.push('I am last.')
```

And *pop* the top element by using:

```
pop()
```

To verify, we can print the top element or implement a `peek()` method:

```
# option 1: simple print function
print(s._top.data)

# option 2: peek method
s.peek()
```

We can also verify the size of the stack by

```
len(s)
```

## 3.5 Queue

A **queue** is another data structure that we are familiar with. It is commonly used to define a line of people waiting to be served at many business establishments. Each person is served based on the order in which the entered the *queue*. Similarly, a *queue* data structure used to model a First-In-First-Out (FIFO) or a Last-In-Last-Out (LILO) strategy. Conceptually, we add to the end of a queue and take away elements from its front. A *queue* can be graphically represented in a similar way to a list or stack. The differences are that references are defined for both the first element and last element in the *queue*, and instead of push and pop, we use *enqueue* and *dequeue* respectively.

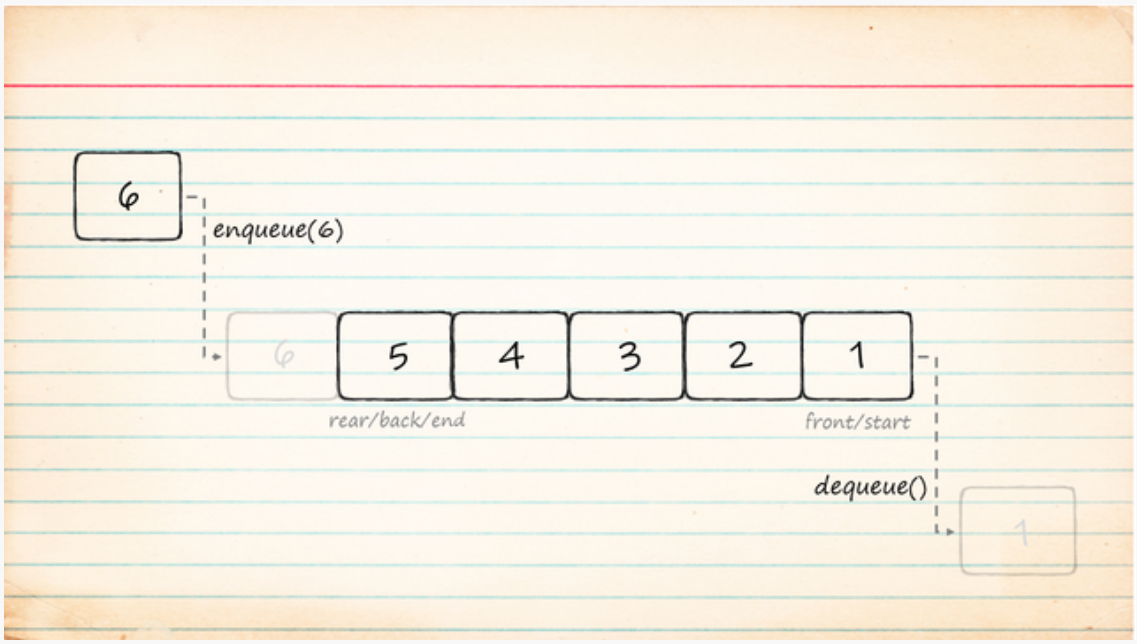


Fig 3.3. Enqueue and Dequeue on a Queue data structure

### 3.5.1 Queue Implementation using Circular Array

The *queue* data structure can be implemented similar to that of the *stack* with some minor variations. However, it requires linear time for the *enqueue* and *dequeue* operations. In order to improve this, we will implement the queue in a circular array. A circular array is simply an array viewed as a circle instead of a line.

To implement a queue as a circular array, we must maintain a *count* field and two markers, i.e., the *front* and the *back*. The count field is necessary to keep track of how many items are currently in the queue since only a portion of the array may actually contain queue items. Consider the following circular array:

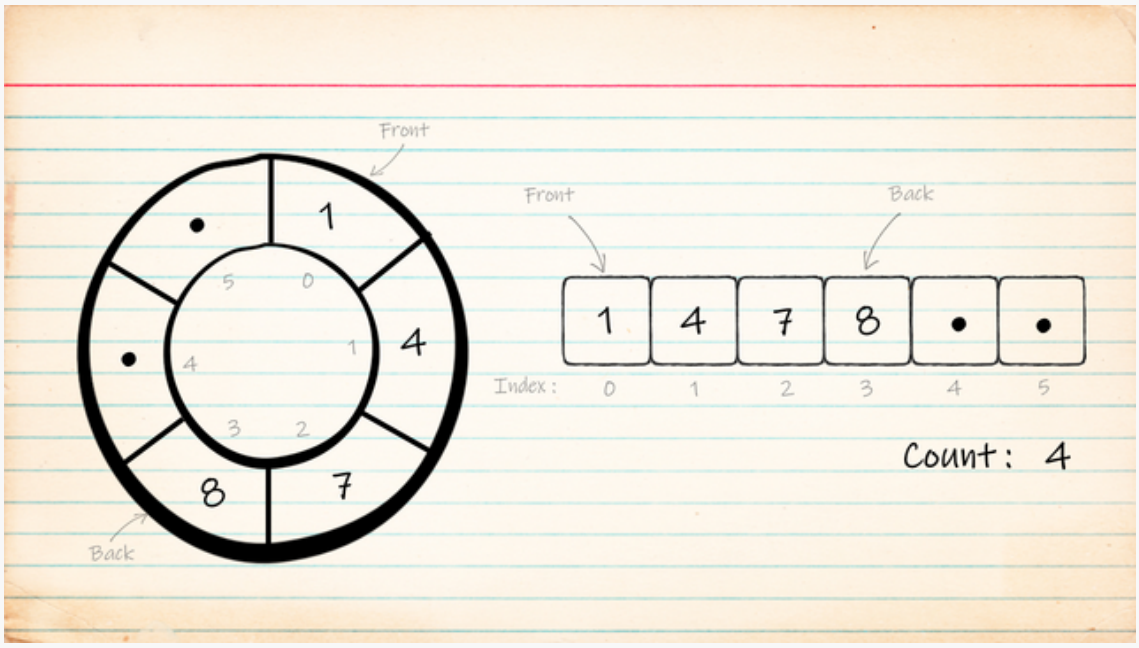


Fig 3.4. Example of an abstract view of circular array (left) and the physical view (right)

New items are added to the queue by inserting them in the position immediately at the *back*. The marker is then advanced one position and the counter is incremented to reflect the addition of the new item. For example,
suppose we `enqueue(6)` into the queue. The back marker is advanced to index $4$ and value $6$ is inserted:

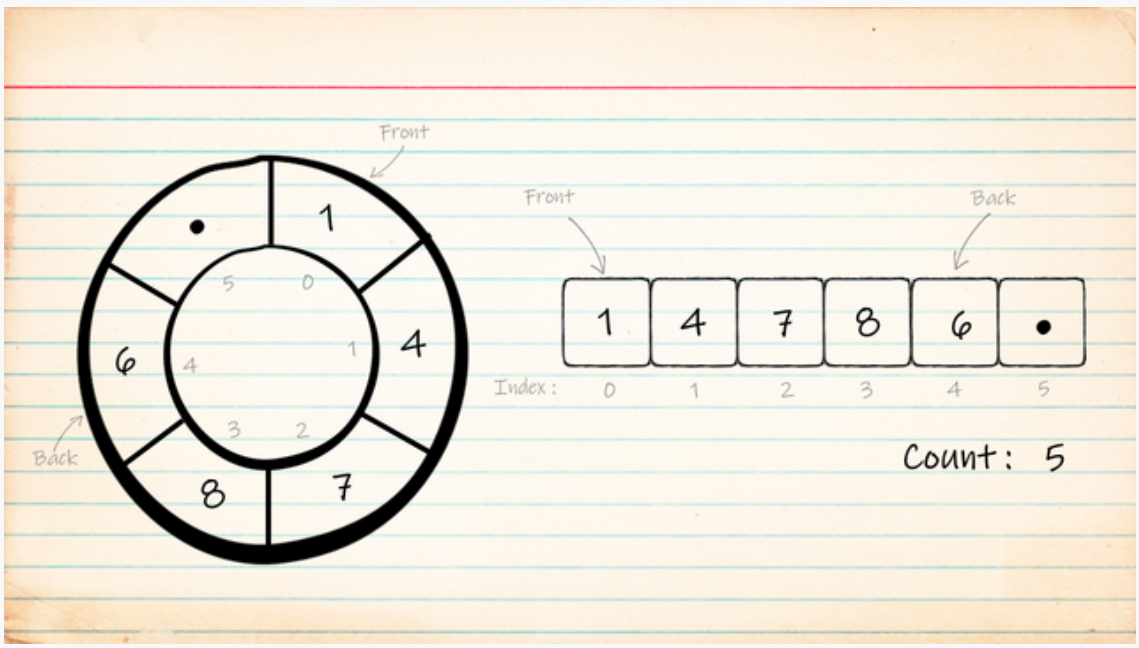


Fig 3.5. After 'enqueue(6)' to the back of the queue

To dequeue an item, the value in the element marked by front will be returned and the marker is advanced one position:

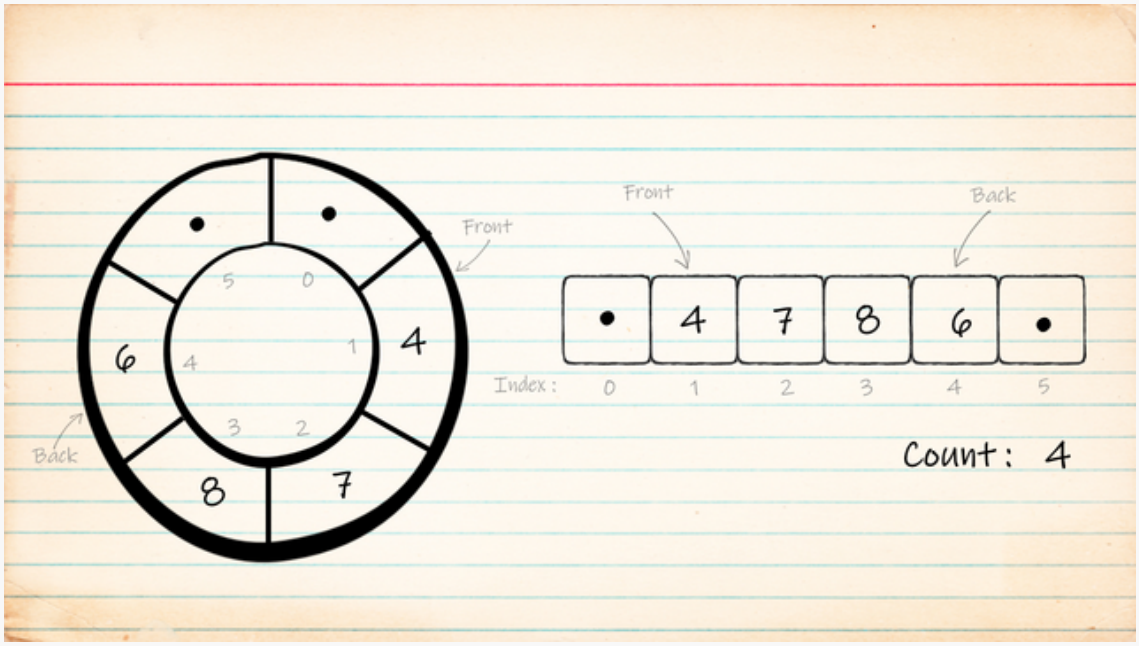


Fig 3.6. After a 'dequeue()'

Notice the remaining items in the queue are not shifted. Instead, only the front marker is moved. Now, suppose we add value 3 to the queue. This value is added in the position following the back marker:

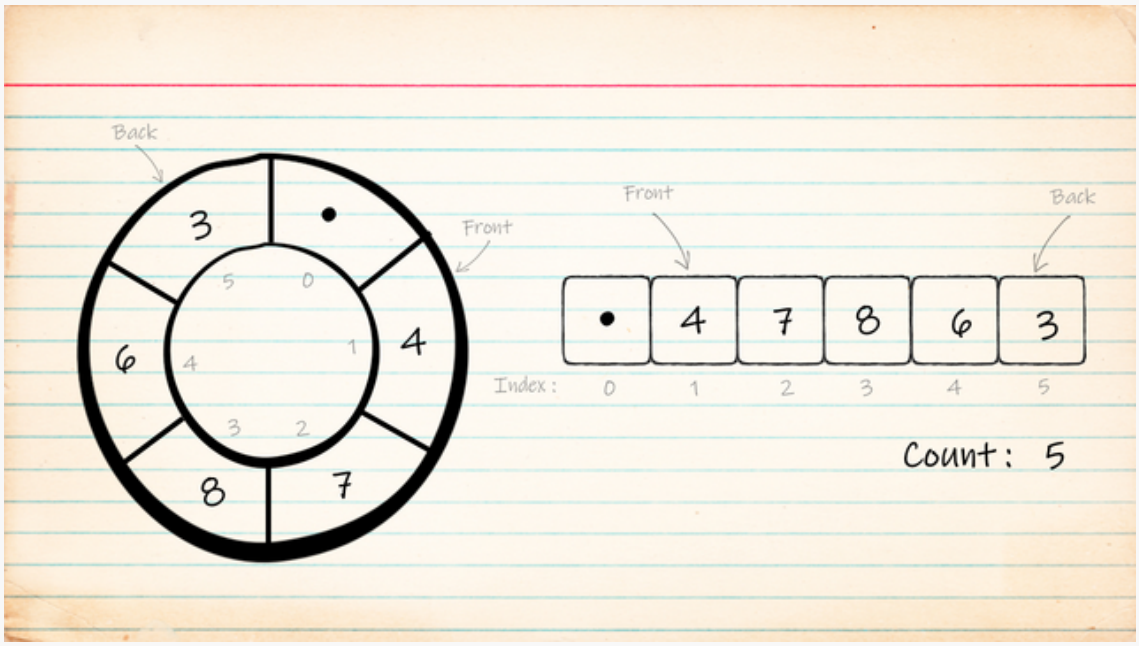


Fig 3.7. After 'enqueue(3)' to the back of the queue'

The queue now contains five items in index 1 to 5 with one empty slot.

So what happens when we `enqueue(5)`? Since we are using a circular array, the same procedure is used and the new item will be inserted into the position immediately following the back marker. In this case, that position will be at index 0. Thus, the queue wraps around the circular array as items are added and removed, which eliminates the need to shift items. The resulting queue is shown here:

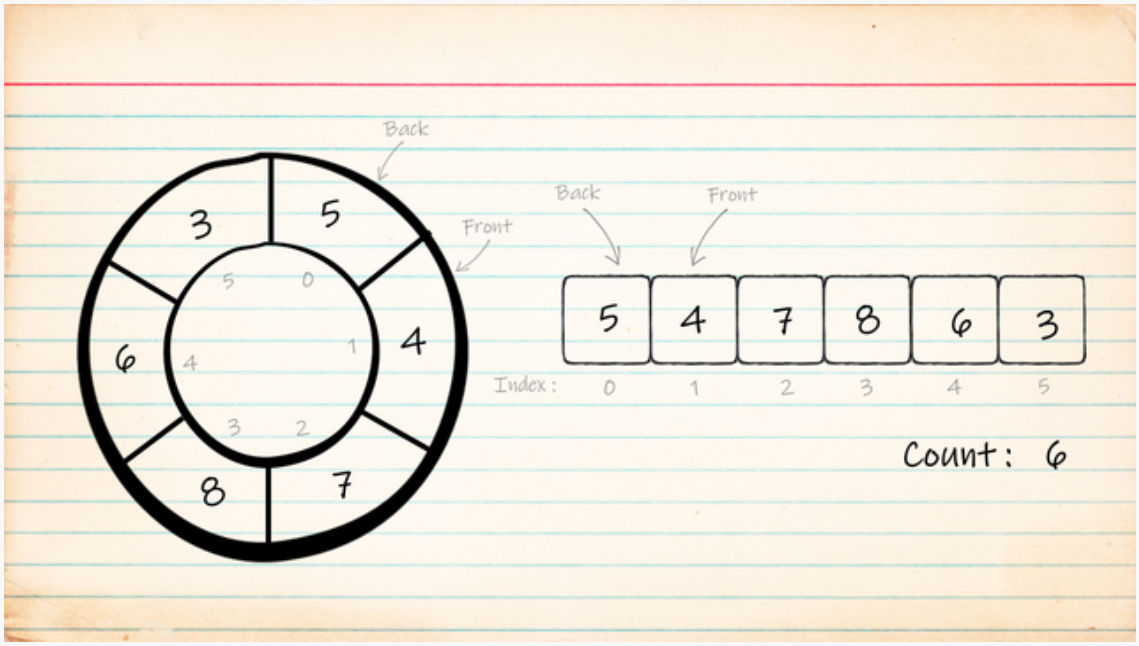


Fig 3.8. After 'enqueue(5)' to the back of the queue'

This also represents a full queue since all slots in the array are filled. No additional items can be added until existing items have been removed. This is a change from the original definition of the *Queue* data structure and requires an additional operation to test for a full queue. The implementation is as shown below:

```
# Implementation of the Queue ADT using a circular array.
from array import Array

class Queue :
    # Creates an empty queue.
    def __init__(self, maxSize) :
        self._count = 0
        self._front = 0
        self._back = maxSize - 1
        self._qArray = Array(maxSize)

    # Returns True if the queue is empty.
    def isEmpty(self) :
        return self._count == 0

    # Returns True if the queue is full.
    def isFull(self) :
        return self._count == len(self._qArray)

    # Returns the number of items in the queue.
    def __len__(self) :
        return self._count
```

```
    # Adds the given item to the queue.
    def enqueue(self, item):
        assert not self.isFull(), "Cannot enqueue to a full queue."
        maxSize = len(self._qArray)
        self._back = (self._back + 1) % maxSize
        self._qArray[self._back] = item
        self._count += 1

    # Removes and returns the first item in the queue.
    def dequeue(self):
        assert not self.isEmpty(), "Cannot dequeue from an empty queue."
        item = self._qArray[self._front]
        maxSize = len(self._qArray)
        self._front = (self._front + 1) % maxSize
        self._count -= 1
        return item
```

For the circular queue, the array is created with `maxSize` elements as specified by the argument to the constructor. The two markers are initialized so the first item will be stored in index 0. This is achieved by setting front to 0 and back to the index of the last element in the array. When the first item is added, back will wrap around to index 0 and the new value will be stored in that position.

The `size()` and `isEmpty()` methods use the value of count to return the appropriate result. As indicated earlier, implementing the *Queue* as a circular array creates the special case of a queue with a maximum capacity, which can result in a full queue. For this implementation of the queue, we must add the `isFull()` method, which can be used to test if the queue is full. Again, the count field is used to determine when the queue becomes full.

To enqueue an item we must first test the precondition and verify the queue is not full. If the condition is met, the new item can be inserted into the position immediately following the back marker. But remember, we are using a circular array and once the marker reaches the last element of the actual linear array, it must wrap around to the first element. This can be done using a condition statement to test if back is referencing the last element and adjusting it appropriately, as shown here:

```
self._back += 1
if self._back == len( self._qArray ) :
    self._back = 0
```

A simpler approach is to use the modulus operator as part of the increment step. This reduces the need for the conditional and automatically wraps the marker to the beginning of the array as follows:

```
self._back = ( self._back + 1 ) % len( self._qArray )
```

The *dequeue()* operation is implemented in a similar fashion as *enqueue()*. The item to be removed is taken from the position marked by front and saved. The marker is then advanced using the modulus operator as was done when enqueueing a new item. The counter is decremented and the saved item is returned.

The circular array implementation provides a more efficient solution than the Python list. Each method now have a worst case time-complexity of $O(1)$ since the array items never have to be shifted. However, the circular array does introduce the drawback of working with a maximum-capacity queue. Nevertheless, this *queue* implementation is well suited for many applications.

In the previous chapters, you have learnt about lists, arrays, linked lists, stacks and queues, each of them having unique properties. However, they are all inherently linear data structures. In the following chapters, we will be looking at nonlinear data structures such as *trees* and *graphs*.