Arrays (aka lists) organise items sequentially, stored in contiguous memory. As they have the least amount of rules and the smallest footprint in terms of memory, Arrays are the simplest of DS.

Example:

In [12]:
```
strings = [5,8,2,9,17,43,25,10] # 4*4 = 16 bytes of storage
```

- Lookup/Access

  Any item of an array can be accessed by its index. We just need to ask for the particular index of that item, and we will get it in constant time.

In [13]:
```
strings[1]
```

Out[13]:
```
8
```

This returned the 2nd item of the array "strings" - in this case, 8 - in O(1) time.

In [14]:
```
strings[5]
```

Out[14]:
```
43
```

Accessed the 6th item of the array, 43. Again, in O(1) time.

- Push/Pop

Push corresponds to pushing or adding an item at the end of the array. In Python, the operation Push is known as Append! Similarly, Pop corresponds to removing the item at the end of the array.

Since the index of the end of the array is known, finding it and pushing or popping an item will only require O(1) time.

In [15]:
```
strings.append(68)
print(strings)
```

```
[5, 8, 2, 9, 17, 43, 25, 10, 68]
```

Important note: In some situations, the append(push) operation may take more time, because Python has dynamic arrays, as mentioned earlier. When an item is appended and the array is filled, the entire array has to be copied to a new location, this time with more space allocated(generally double the space), to append more items. Therefore, some individual operations may require O(n) time or more, but when averaged over a large number of operations, the time complexity can be safely considered as O(1).

In [16]:
```
strings.pop() #Pop removes the item positioned at the end of the array in (
print(strings)
```

```
[5, 8, 2, 9, 17, 43, 25, 10]
```

- Insert (i, x)

Insert operation inserts an item at a given position. The first argument is the index of the item before which to insert (a.insert(0, x) inserts at the front of the array, and a.insert(len(a), x) is equivalent to a.append(x) which adds the item at the end of the array).

The time complexity of this operation is O(n), because after inserting the item at the desired location, the items to the right of the array have to be updated with the correct index, as they all have shifted by one place.

This requires looping through the array, which leads to O(n).

In [17]:
```
strings.insert(0,50)
# Inserts 50 at the beginning of the array and shifts all other elements o

strings.insert(4,0)
# Inserts '0' at index '4', thus shifting all elements starting from index
```

In [18]:
```
print(strings)
```

```
[50, 5, 8, 2, 0, 9, 17, 43, 25, 10]
```

- Delete/Remove/Del statement

Similar to insert, it deletes an item from a specified location in O(n) time. The items to the right of the deleted item have to shift one space to the left, which requires looping over the entire array, leading to a O(n) time complexity.

There is a way to remove an item from a list given its index instead of its value: the del statement. This differs from the .pop() method, which returns a value. The del statement can also be used to remove slices from a list or clear the entire list.

In [20]:
```
strings.pop(0)
# This pops the 1st item of the array, shifting the remaining elements of 
print(strings)

strings.remove(17)
# This command removes the first occurence of the item '17' in the array, 
print(strings)

del strings[2:4]
# This command deletes items from position 2 to position 4. -> O(n)
print(strings)
```

```
[8, 2, 0, 9, 17, 43, 25, 10]
[8, 2, 0, 9, 43, 25, 10]
[8, 2, 43, 25, 10]
```

Sources:

- https://docs.python.org/3/tutorial/datastructures.html
- https://docs.python.org/3/tutorial/datastructures.html#the-del-statement
- https://github.com/VicodinAbuser/ZTM-DS-and-Algo-Python/blob/master/venv/Scripts/Data%20Structures/Arrays/Introduction.py

In [ ]: