Introduction to Algorithms & Data Structures 2

A solid foundation for the real world of machine learning and data analytics

Bolakale Aremu

Ojula Technology Innovations

Introduction to Algorithms & Data Structures 2

First Edition

ISBN: 9791222095417

Printed in the United States of America

Print Number: 01

Print Year: April 2023

I am indebted to my mother for her love, understanding and support throughout the time of writing this textbook.

Bolakale Aremu

Table of Contents

About The Author

My name is Bolakale Aremu. My educational background is in software development. I have a few colleagues who are software developers and system engineers. I spent over 17 years as a software developer, and I’ve done a bunch of other things too. I've been involved in SDLC/process, data science, operating system security and architecture, and many more. My most recent project is serverless computing where I simplify the building and running of distributed systems. I always use a practical approach in my projects and courses.

Bolakale Aremu
CEO, Ojula Technology Innovations
Web developer and Software Engineer
Ojulaweb.com

0. What You Will Learn in Volume 2

This book is the second volume of a four-part series titled *Introduction to Algorithms & Data Structures*. The design of an efficient algorithm for the solution of the problem calls for the inclusion of appropriate data structures. In the field of computer science, data structures are used to store and organize data in a way that is easy to understand and use. They are used to organize and represent data in a way that will make it easier for computers to retrieve and analyze it. These are the fundamental building blocks that any programmer must know how to use correctly in order to build their own programs.

0.1. Benefits of learning about algorithms and data structures

First, they will help you become a better programmer. Another benefit is that they will make you think more logically. Furthermore, they can help you design better systems for storing and processing data. They also serve as a tool for optimization and problem-solving.

As a result, the concepts of algorithms and data structures are very valuable in any field. For example, you can use them when building a web app or writing software for other devices. You can apply them to machine learning and data analytics, which are two hot areas right now. If you are a hacker, algorithms and data structures in Python are also important for you everywhere.

Now, whatever your preferred learning style, I've got you covered. If you're a visual learner, you'll love my clear diagrams and illustrations throughout this book. If you're a practical learner, you'll love my hands-on lessons so that you can get practical with algorithms and data structures and learn in a hands-on way.

0.2. Course Structure

There are three volumes in this course. This is volume two. In volume one, I took a deep dive into the world of algorithms. I covered what **algorithms** are, how they work, and where they can be found (real life applications).

In this part of the series (volume two), we'll work through an introduction to data structures. You're going to learn about two introductory data structures - **arrays** and **linked lists**. You'll look at common operations and how the runtimes of these operations affect our everyday code.

In the third volume, you're going to bring your knowledge of algorithms and data structures together to solve the problem of sorting data using the Merge Sort algorithm. We will look at algorithms in two categories: **sorting** and **searching**. You'll implement well-known sorting algorithms like Selection Sort, Quicksort, and Merge Sort. You'll also learn basic search algorithms like Sequential Search and Binary Search.

At the end of many sections of this course, short practice exercises are provided to test your understanding of the topic discussed. Answers are also provided so you can check how well you have performed in each section. At the end of the course, you will find a **link to download more helpful resources, such as codes and screenshots used in this book, and more practice exercises**. You can use them for quick references and revision as well. My **support link is also provided** so you to contact me any time if you have questions or need further help.

By the end of this course, you will understand what algorithms and data structures are, how they are measured and evaluated, and how they are used to solve real-life problems. So, everything you need is right here in this book. I really hope you'll enjoy it. Are you ready? Let's dive in!

2. Introduction to Data Structures

In this part of the course, I will introduce data structures. First, we're going to answer one fundamental question: **why do we need more data structures than a programming language provides?** This question will be answered in section 2.3.1. For now, let’s do some housekeeping.

In this course, we're going to rely on the concepts we learned in volume one (*Introduction to Algorithms & Data Structures 1*), namely big O notation, space and time complexity, and recursion. If you're unfamiliar with those concepts or just need a refresher, check out volume one. In addition, this course does assume that you have some programming experience.

We're going to use data structures that come built into nearly all programming languages as our point of reference. While we will explain the basics of how these structures work, we won't be going over how to use them in practice. If you're looking to learn how to program in Python before digging into this content, check out the following book that completely simplifies Python programming language for beginners:

https://www.scribd.com/book/513773394/Python-Programming-from-Beginner-to-Paid-Professional-Part-1-Learn-Python-for-Automation-IT-with-Video-Tutorials (or use this BIT.LY short link: http://bit.ly/3ZKREhB).

Even if you know nothing about Python, as long as you understand the fundamentals of programming, you should be able to follow along pretty well. If you're good to go, then awesome.

2.1. Course Overview

Let's start with an overview of this course. The first thing we're going to do is to explore a data structure we are somewhat already familiar with: arrays. If you've written code before, there's a high chance you have used an array. In this course, we're going to spend some time understanding how arrays work, what are the common operations on an array, and what are the run times associated with those operations.

Once we've done that, we're going to build a data type of our own called a **linked list**. In doing so, we're going to learn that there's more than one way to store data. In fact, there's way more than just one way. We're also going to explore what motivates us to build specific kinds of structures and look at the pros and cons of these structures. We'll do that by exploring four common operations: accessing a value, searching for a value, inserting a value, and deleting a value.

After that, we're going to circle back to algorithms and implement a new one: a sorting algorithm. In the introduction to algorithms course (volume one), we implemented a binary search algorithm. A precondition to binary search was that the list needed to be sorted. We're going to try our hand at sorting a list and open the door to an entirely new category of algorithms. We're going to implement our sorting algorithm on two different data structures and explore how the implementation of one algorithm can defer based on the data structure being used.

We'll also look at how the choice of data structure potentially influences the runtime of the algorithm. In learning about sorting, we're also going to encounter another general concept of algorithmic thinking called divide and conquer. Along with recursion, divide and conquer will be a fundamental tool that we will use to solve complex problems, all in due time. In the next section let's talk about arrays.

2.2. Exploring Arrays

2.2.1. Array Basics

A common data structure built into nearly every programming language is the array. Arrays are a fundamental data structure and can be used to represent a collection of values. But it is much more than that. Arrays are also used as building blocks to create even more custom data types and structures. In fact, in most programming languages, text is represented using the *string* type and, under the hood, strings are just a bunch of characters stored in a particular order in an array.

What is a Data Structure?

Before we go further and dig into arrays, what exactly is a data structure? A data structure is a way of storing data when programming. It's not just the collection of values and the format they're stored in, but the relationship between the values in the collection as well as the operations applied on the data stored in the structure.

An array is one of very many data structures in general. An array is a data structure that stores a collection of values where each value is referenced using an **index** or a **key**. An example is shown in Figure 2.2.1.

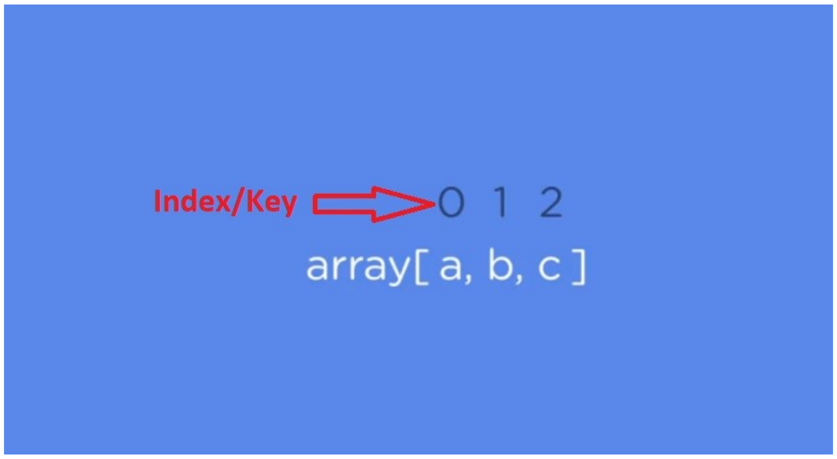


Figure 2.2.1: An example of an array with its values (a, b, c) and keys (0, 1, 2)

A common analogy for thinking about arrays is as a set of train cars. Each car has a number, and these cars are ordered sequentially. Inside each car (the array in this analogy) some data are stored. While this is the general representation of an array, it can differ slightly from one language to another, but for the most part, all these fundamentals remain the same. In a language like Swift or Java, arrays are **homogenous structures**, which means they can only contain values of the same type. If you use an array to store integers in Java, it can only store integers.

In other languages like Python, arrays are **heterogeneous structures** because they can store any kind of value. So, in Python for example, you can mix numbers and text with no issues. Now, regardless of this nuance, the fundamental concept of an array is the index. This index value is used for every operation on the array, from accessing values to inserting, updating and deleting.

In Python, the language we're going to be using for this course, is a little bit confusing. The type that we generally refer to as an array in most languages is best represented by the **list** type in python. Python does have a type called array as well, but it's something different, so we're not going to use it. While Python calls it a list, when we use a list in this course, we'll be talking about concepts that apply to arrays as well in other languages, so definitely don't skip any of this.

There's one more thing. In computer science, a list is actually a different data structure than an array, and in fact, we're going to build a list later on in this course. Generally, though, this structure is called a linked list as opposed to just list, so hopefully the terminology isn't too confusing. To properly understand how arrays work, let's take a peek at how arrays are stored under the hood.

An array is a contiguous data structure. This means that the array is stored in blocks of memory that are right beside each other with no gaps. The advantage of doing this is that retrieving values is very easy. In a non-contiguous data structure, and we're going to build one soon, the structure stores a value *as well as* a **reference** to where the next value is.

To retrieve that next value, the language has to follow that reference, also called a **pointer**, to the next block of memory. This adds some overhead, which, as you'll see, increases the runtime of common operations. A moment ago, I mentioned that depending on the language, arrays can either be homogenous (containing the same type of value) or heterogeneous where any kind of value can be mixed. This choice also affects the memory layouts of the array.

For example, in a language like C, swift or Java, where arrays or homogenous, when an array is created, since the kind of value is known to the language **compiler**, and you can think of the compiler as the brains behind the language, it can choose a contiguous block of memory that fits the array size and values created.

If the values were integers, assuming an integer took up space represented by one block, then for a five-item array, the compiler can allocate five blocks of equally sized memory. Figure 2.2.2.

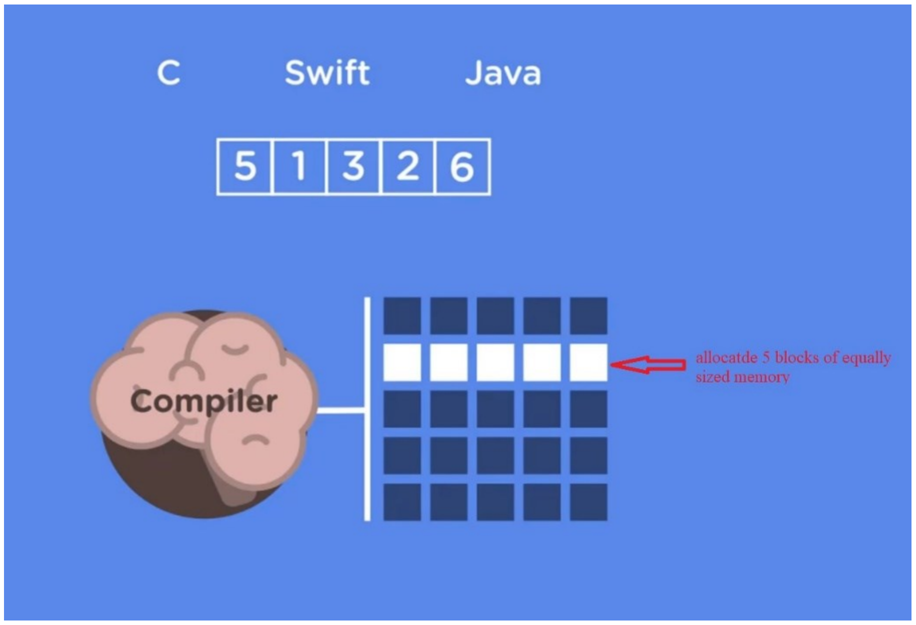


Figure 2.2.2: Illustration of how a compiler allocates 5 blocks of equally sized memory

In Python, however, this is not the case. We can put any value in a Python list. There's no restriction. The way this works is a combination of contiguous memory and the pointers or references I mentioned earlier. When we create a list in Python, there is no information about what will go into that array, which makes it hard to allocate contiguous memory of the same size.

There are several advantages to having contiguous memory. Since the values are stored beside each other, **accessing the values happens in almost constant time**. This is a characteristic we want to preserve. The way Python gets around this is by allocating contiguous memory and storing in it, not the value we want to store, but a reference or a pointer to the value that's stored somewhere else in memory. Figure 2.2.3.

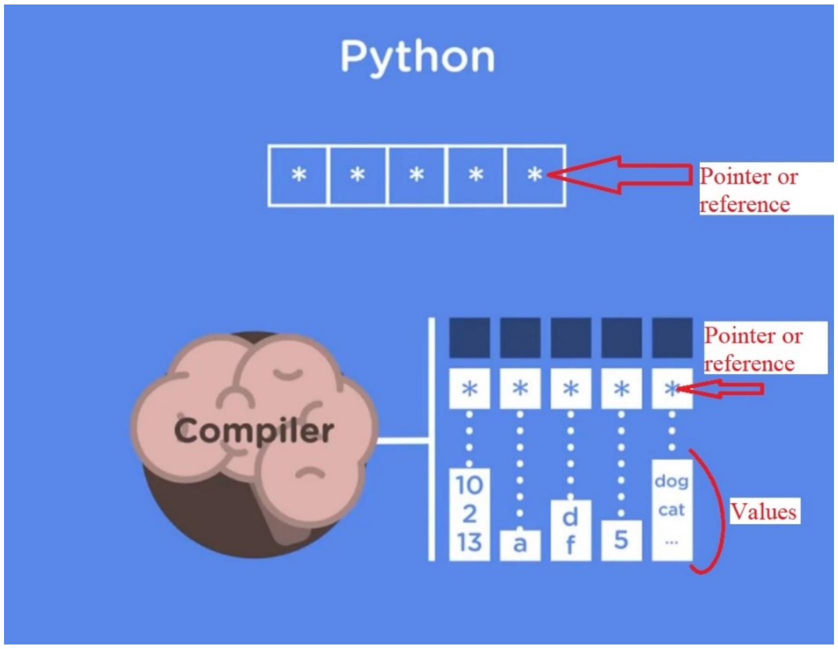


Figure 2.2.3: Illustration of how a Python compiler allocates 5 blocks of equally-sized memory using pointers or references

By doing this, it can allocate equally sized contiguous memory since regardless of the value, size, the size of the pointer to that value is always going to be equal. This incurs an additional cost in that when a value is accessed, we need to follow the pointer to the block of memory where the value is actually stored. But Python has ways of dealing with these costs that are outside of the scope of this course.

Now that we know how an array stores its values, let's look at common operations that we execute on an array.

2.2.2. Accessing a Value in an Array

Regardless of the kind of data structure you work with, all data structures are expected to carry out four kinds of operations.

Operations on Data Structure

At minimum,

1. we need to be able to **access and read** values stored in the structure.
2. we need to be able to **search** for an arbitrary value.
3. we also need to be able to **insert** a value at any point into the structure, and finally,
4. we need to be able to **delete** structures.

Let's look at how these operations are implemented on the array structure in some detail, starting with **access**. Elements in an array are identified using a value known as an index. We use this index to access and read the value. Most programming languages follow a zero-based numbering system when it comes to arrays. All this means is that the first index value is equal to **0**, not 1. Figure 2.2.4.

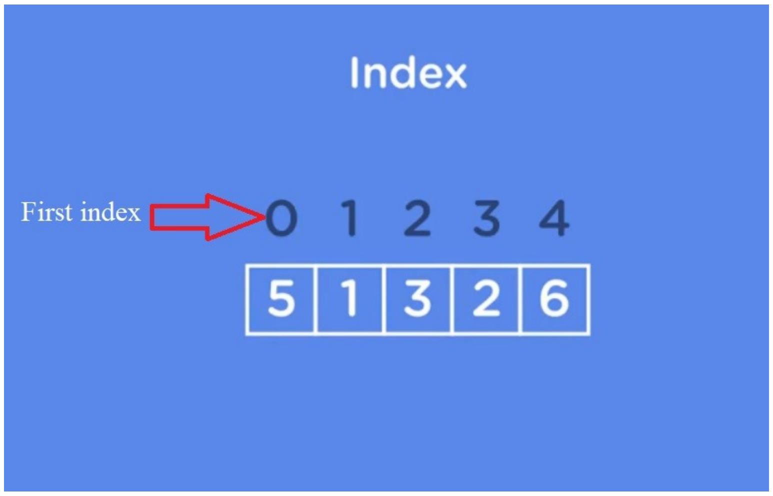


Figure 2.2.4: Illustration of a zero-based numbering system in an array

Generally speaking, when an array is declared a base amount of contiguous memory is allocated as the array storage. Computers refer to memory through the use of an address, but instead of keeping a reference to all the memory allocated for an array, the array only has to store the address of the first location because the memory is contiguous. Using the base address, the array can calculate the address of any value by using the index position of that value as an offset. If you want to be more specific, think of it this way.

Let's say we want to create an array of integers, and then each integer takes up a certain amount of space in memory that we'll call **M**. Let's also assume that we know how many elements we're going to create, so the size of the array is some number of elements we'll call **N**. So, the total amount of space that we need to allocate is **N** times the space per item M: **Space = N * M**. See Figure 2.1.5.

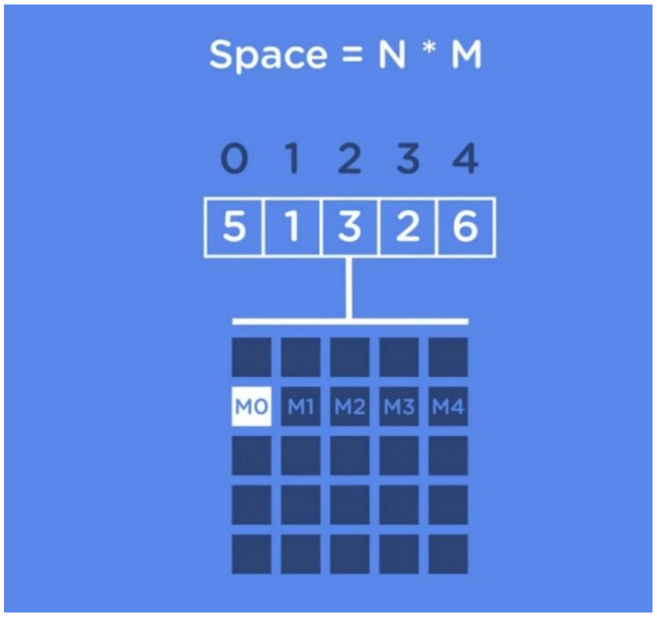


Figure 2.2.5: Illustration of how memory is allocated in an array using address system

If the array keeps track of the location in memory where the first value is held (let's label that **M0**), then it has all the information it needs to find any other element in the list. When accessing a value in an array, we use the index. So, to get the first element in the list, we use the 0th index. To get the second, we use the index value 1, and so on.

Given that the array knows how much storage is needed for each element, it can get the address of any element by starting off with the address for the first element, and adding to that the index value times the amount of storage per element. For example, to access the second value, we can start with M0 and to that add M times the index value 1 (Space = 1 * M), giving us M1 as the location and memory for the second address.

This is a very simplified model, but that's more or less how it works. This is only possible because we know that array memory is contiguous with no gaps. Let's switch over to some code. As I mentioned earlier, and also in volume one of the course, we're going to be using Python and Pycharm IDE (Integrated development environment) for writing our code. Feel free to use any other IDE you like, such as a notepad or even the Python interpreter you installed. While the code will be in Python, the concepts are universal, and more importantly, simple enough that you should have no issue following along in aby of your favorite programming languages.

Now, open Pycharm and create a new file. Name this file *arrays.py*. Figure 2.2.6.

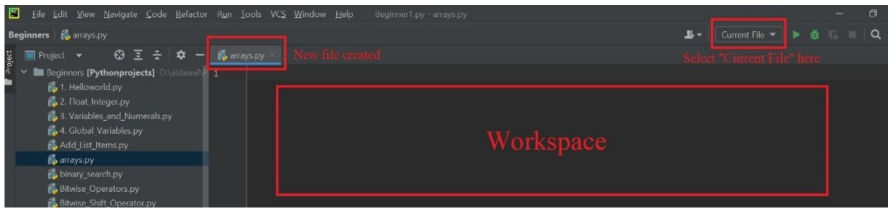


Figure 2.2.6: Screenshot of a new file created in Pycharm. The workspace is where will write our Python code

Creating a list in Python is quite simple. Here's an example:

```
new_list = [1, 2, 3]
```

This allocates a base amount of memory for the array to use. When I say array, know that in Python, I mean a list. Since this is Python, the values aren't stored directly in memory. Instead, the values 1, 2, and 3 are stored elsewhere in memory, and the array stores references to each of those objects. To access a value, we use a subscript along with an index value. So, to get the first value, we use the index 0. If we were to assign this to another variable, we would say

```
result = new_list[0]
```

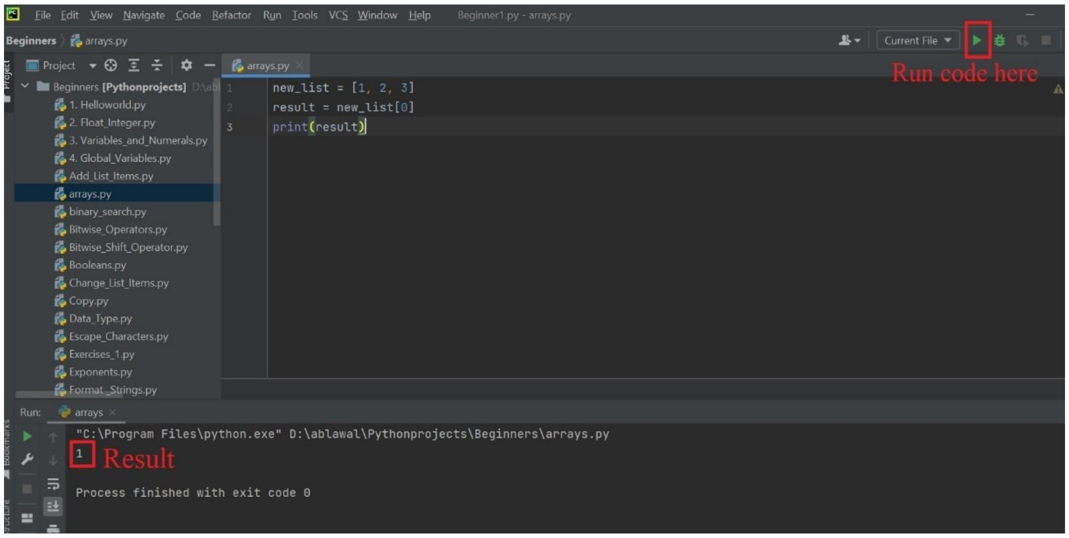
We write out *new_list* since this is the array that we're accessing the value from, and then a subscript notation, which is two square brackets, and then the index value 0. As you saw earlier, since the array has a reference to the base location in memory, the position of any element can be determined pretty easily. We don't have to iterate over the entire list. All we need to do is a simple calculation of an offset from the base memory since we're guaranteed that the memory is contiguous. For this reason, access is a constant time operation on an array or a Python list. This is also why an array crashes if you try to access a value using an index that is out of bounds of what the array stores, such as *new_list[3]* or *new_list[10].*

If you've used an array before, you've undoubtedly run into an error or a crash where you try to access a value using an index that was larger than the number of elements in the array. Since the array calculates the memory address on the fly, when you access a value with an **out-of-bounds** index, as it's called, the memory address returned is not one that's part of the array structure and therefore cannot be read by the array.

Now in Python, this is represented by an index error, and we can make this happen by using an index we know our array won't contain. Now if we add

```
print (result)
```

and run our code in Pycharm, you'll see that we get the value 1, which is the value stored at the 0th index. Figure 2.2.7.



```
new_list = [1, 2, 3]
result = new_list[0]
print(result)
```

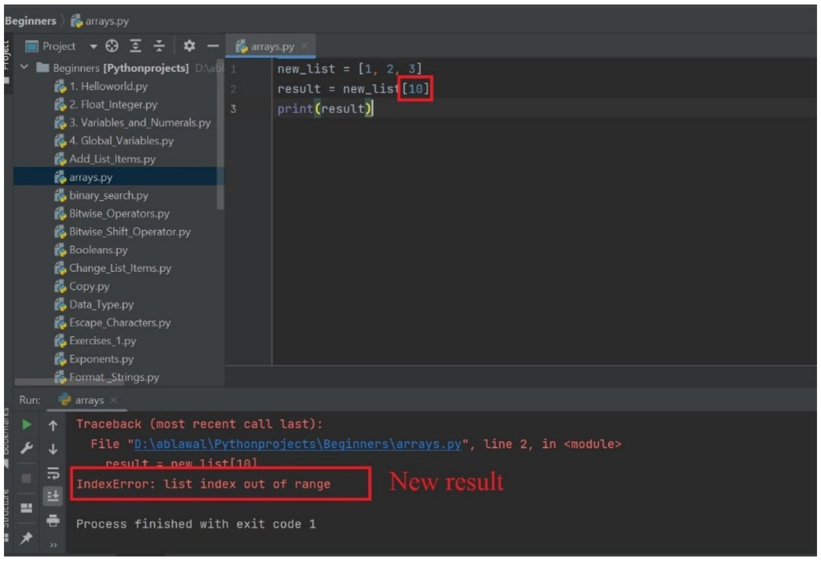
```
"C:\Program Files\python.exe" D:\ablawal\Pythonprojects\Beginners\arrays.py
1

Process finished with exit code 0
```

Figure 2.2.7: Screenshot showing the result of our Python code

Now to highlight that index error, we can do new list and, inside the square brackets, we can provide an index that we know our array doesn't contain. So, change the second line of your code to

```
result = new_list[10]
```

If you save and then run the updated code, you'll get the error shown in Figure 2.2.8.



```
new_list = [1, 2, 3]
result = new_list[10]
print(result)
```

```
Traceback (most recent call last):
  File "D:\ablawal\Pythonprojects\Beginners\arrays.py", line 2, in <module>
    result = new_list[10]
IndexError: list index out of range

Process finished with exit code 1
```

Figure 2.2.8: Screenshot showing a different result when we used an out-of-bounds index value

Now Python says *IndexError: list index out of range*.

Those are the basics of how we create and read values from an array. In the next section, we'll take a look at searching but first here's some practice exercise.

2.2.3. Practice Exercise 1

Q1. Blocks of memory allocated for an array are of equal size. True or False?

A. True
B. False

Q2. What advantage does an array gain by using a contiguous block of memory?

A. Access to elements is faster since the memory is located in one area
B. All index values do not have to be stored and can be calculated at runtime using an offset
C. As the number of elements approaches the maximum memory allocated, the array needs to be resized
D. All of the above
E. A and B

Q3. How does an array store elements that are of different sizes?

A. By storing a reference to the element's location, which is elsewhere in memory
B. By allocating non-contiguous memory to store the element
C. By reallocating memory for individual elements

Q4. In all programming languages, arrays are homogenous data structures and can only contain one type of value once defined.

A. True
B. False

Q5. Please fill in the correct answer in the blank.

An array is a data structure where each element in the collection is identified by means of an ___.

2.2.4. Answers to Practice Exercise 1

1. A
2. E
3. A
4. B
5. index

2.2.5. Array Search, Insert and Delete

Array Searching

In the previous section, we learned what happens under the hood when we create an array and read a value using an index. In this section, we're going to look at how the remaining data structure operations work on arrays. If you took the introduction to algorithms course (volume one of this book), we spent time learning about two search algorithms: linear search, and binary search. While arrays are really fast at accessing values, they're pretty bad at searching. Taking an array as is, the best we can do is use linear search for a worst-case linear runtime.

Linear search works by accessing and reading each value in the list until the element in concern is found. If the element we're looking for is at the end of the list, then every single element in the list will have been accessed and compared. Even though accessing and comparing are constant time operations, having to do this for every element results in an overall linear time. Now, let's look at how search works in code.

In Python, we can search for an item in an array in one of two ways. We can use the *in* operator to check whether a list contains an item. For example, we can add the following line to our previous code:

```
if 1 in new_list: print(True)
```

The *in* operator actually calls a *contains* method that is defined on the *list* type, which runs a linear search operation. In addition to this, we can also use a *for* loop to iterate over the list manually and perform a comparison operation. So, I can say

```
for n in new_list:
  if n==1:
    print(True)
    break
```

After printing the result then after that break out of the loop. This is more or less the implementation of linear search. If the array were sorted, however, we could use binary search. But because sort operations incur a cost of their own, languages usually stay away from sorting the list and running binary search since for smaller arrays linear search on its own may be faster.

So, if you run the following code, you'll see that Python prints *True*.

```
new_list = [1, 2, 3]
if 1 in new_list: print(True)
```

Now because we've already learned about linear and binary search in a previous course, there's nothing new going on here. What's more interesting to look at in my opinion is inserting and deleting values in an array. Let's start with inserting.

Array Inserting

In general, most array implementations support three types of insert operations. The first type is a **true insert** using an index value where we can insert an element *anywhere* in a list. This operation has a **linear runtime**. Imagine you wanted to insert an item at the start of the list.

Figure 2.2.9.

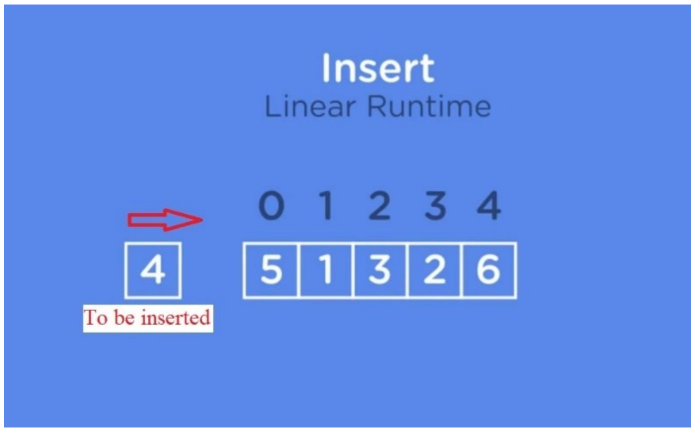


Figure 2.2.9: The element 4 is to be inserted into the first position with index value 0

When we insert into the first position (index position 0), what happens to the item that is currently in that spot? Well, it has to move to the next spot at index position 1. What happens to the second item at index position 1? That one moves to the next spot at index position 2. This keeps happening until all elements have been shifted forward by one index position. Figure 2.2.10.

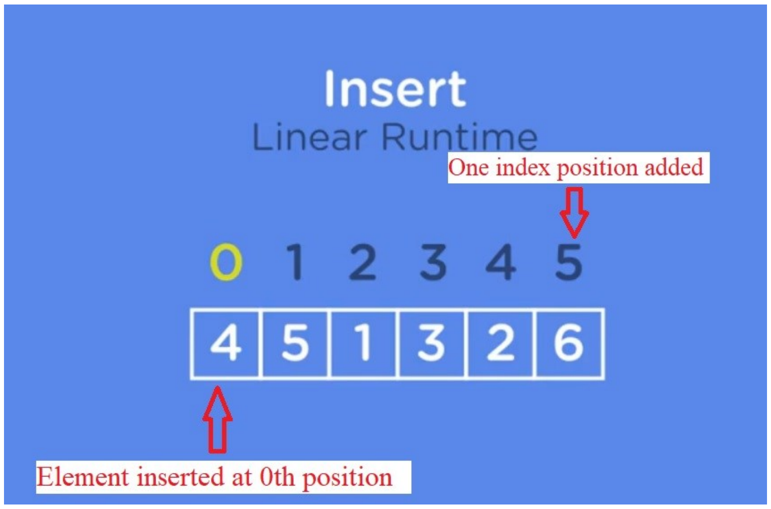


Figure 2.2.10: The element 4 is inserted into the first position and one index value is added to the array

So, in the worst-case scenario, inserting at the 0th position of an array, every single item in the array has to be shifted forward and we know that any operation that involves iterating through every single value means a linear runtime.

Now the second type of insert is called **appending**. Although appending is technically an insert operation in that it inserts an item into an existing array, it doesn't incur the same runtime cost because it simply adds the item to the **end of the list**. We can simplify and say that this is constant time operation, but it depends on the language implementation of the array.

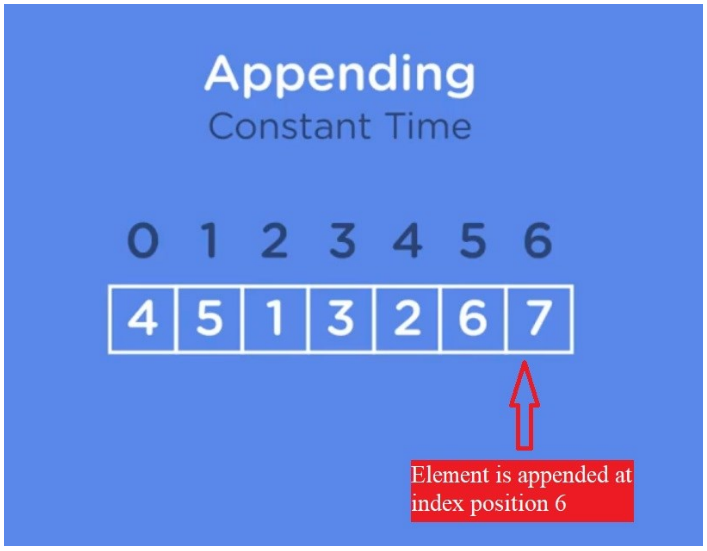


Figure 2.2.11: Element 7 is appended to the array at the last index position and one index value is added

To highlight why that matters, let's consider how lists in Python work. In Python when we create a list, the list doesn't know anything about the size of the list and how many elements we're going to store.

Creating a new list like

```
numbers = []
```

creates a list and allocates a space of size **n + 1**. Since n here is zero, there are no elements in this array. So, in this list, space is allocated for a one element list to start off. Because the space **allocated for the list** and the space **used by the list** are not the same, what do you think happens when we use the following code to ask Python for the length of this list?

```
print(len(numbers))
```

We correctly get 0 back.

```
0
```

This means that the list doesn't use the memory allocation as an indicator of its size. This is because, as I mentioned earlier, it has allocated space for a one element list, but it returns zero. So numbers in this list currently has space for one element. Now, let's use the append method defined on the type to insert a number at the end of the list.

```
numbers.append(2)
```

Now, the memory allocation and the size of the list are the same since the list contains one element. What if I were to do something like this?

```
numbers.append(200)
```

Now, since the list only has an allocation for one item at this point, before it can add the new element to the list, it needs to increase the memory allocation and thereby the size of the list. It does this by calling a **list-resize** operation. List resizing is quite interesting because it shows the ingenuity in solving problems like this.

Python doesn't resize the list to accommodate just the element, say **5**, that we want to add. Instead, in this case, to add one element 5, it would allocate four blocks of memory to increase the size to a total of four contiguous blocks of memory. It does this so that it doesn't have to resize the list every single time we add another element like **1**, but at very specific points. The growth pattern of the list type in Python is 0, 4, 8, 16, 25, 35, 46 and so on. Figure 2.2.12. This means that as the list size approaches these specific values, **list-resize** operation is called again.

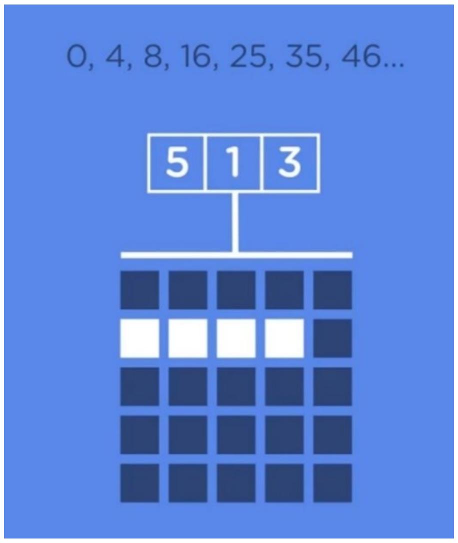


Figure 2.2.12: Python allocates 4 blocks of memory even though only three *elements have been added*

If you look at when the size of the list is four, this means that when appending four more values until the size of eight, each of those append operations do not increase the amount of space taken at specific points. However, when resizing is triggered, space required increases as memory allocation increases. Figure 2.2.13.

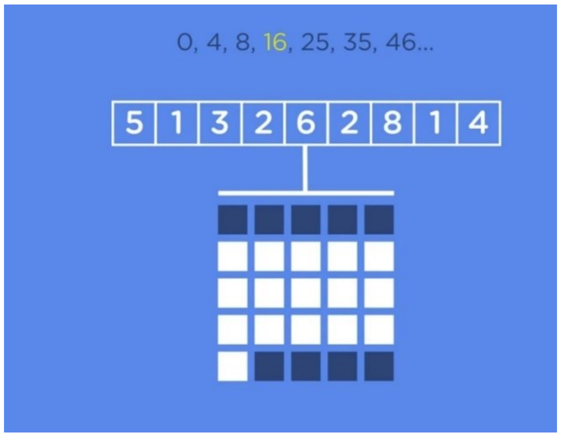


Figure 2.2.13: Allocation of 16 blocks of memory is triggered when the 9th element 4 is added

This might signify that the append method has a non-constant space complexity, but it turns out that, because some operations don't increase space and others do, when you average all of them out, append operations take constant space. We say that it has an **Amortized Constant Space Complexity**.

This also happens with insert operations. If we had four-element array, we would have four elements and a memory allocation of four. An insert operation at that point, it doesn't matter where it happens on the list, it would trigger a resize. Inserting is still more expensive though because after the resize, every element needs to be shifted over one.

The last (third) insert operation that is supported in most languages is the ability to **add one list to another**. In Python, this is called an **extend**. Now, let’s clear your workspace and start writing a new code. We will begin with an empty list.

```
numbers = []
```

```
numbers.extend([4,5,6])
```

In the second line, as an argument, I just passed in (added) a new list (4, 5, 6) entirely. Now, to print out *numbers*, we write

```
print(numbers)
```

If you run this code, you'll see that our list now contains the values 4, 5, and 6.

```
[4, 5, 6]
```

So, *extend* takes another list to add. Extend effectively makes a series of append calls on each of the elements in the new list until all of them have been appended to the original list. This operation has a runtime of Big O(k) where k represents the number of elements in the list that we're adding to our existing list.

Array Deleting

The last type of operation we need to consider are delete operations. Deletes are similar to inserts in that when a delete operation occurs, the list needs to maintain correct index values. So, where an insert shifts every element to the right, a delete operation shifts every element to the left. Just like an insert as well, if we delete the first element in the list, every single element in the list needs to be shifted to the left. Delete operations have an upper bound of Big O(n), also known as a **linear runtime**.

Now that we've seen how common operations work on a data structure that we're quite familiar with, let's switch tracks and build our own data structure.

2.2.6. Practice Exercise 2

Q1. Adding one array to another in a series of append operations has a runtime of O(k). What does k represent here?

A. The size of the array being added
B. The size of the original array
C. The runtime of an append operation

Q2. Why do array inserts and deletes have a linear runtime in the worst case? Choose the correct answer below:

A. With insert operations, the array needs to read every value to ensure it doesn't insert the same value twice
B. With deletes the array needs to search for the element before it can delete it and this can take an n number of read operations
C. Every element in the array has to be shifted to either the left (deletes) or right (inserts)

Q3. What do we use to access values in an array? Choose the correct answer below:

A. Node
B. Index
C. Key

2.2.7. Answers to Practice Exercise 2

1. A

2. C.

3. B

2.3. Building a Linked List

Despite having powerful collection types built in to most languages, we often have a need for custom data structures that store data in different ways. In this section we'll take a look at what a linked list is.

2.3.1. What Is a Linked List?

Over the next few sections, we're going to build a data structure that you may have worked with before: a linked list. Before we get into what a linked list is, let's talk about why we build data structures instead of just using the ones that come built into our languages.

Each data structure solves a particular problem. We just went over the basics of the array data structure and looked at the cost of common operations that we carry out on arrays. We found that arrays were particularly good at accessing. Reading values happens in constant time. But arrays are pretty bad at inserting and deleting, both of which run in linear time.

Linked lists on the other hand, are somewhat better at this, although there are some caveats. If we're trying to solve a problem that involves far more inserts and deletes than accessing, a linked list can be a better tool than an array.

So, what is a linked list? A linked list is a linear data structure where each element in the list is contained in a separate object called a **node**. A node models two pieces of information: an individual item of the data we want to store and a reference to the next node in the list. Figure 2.3.1.

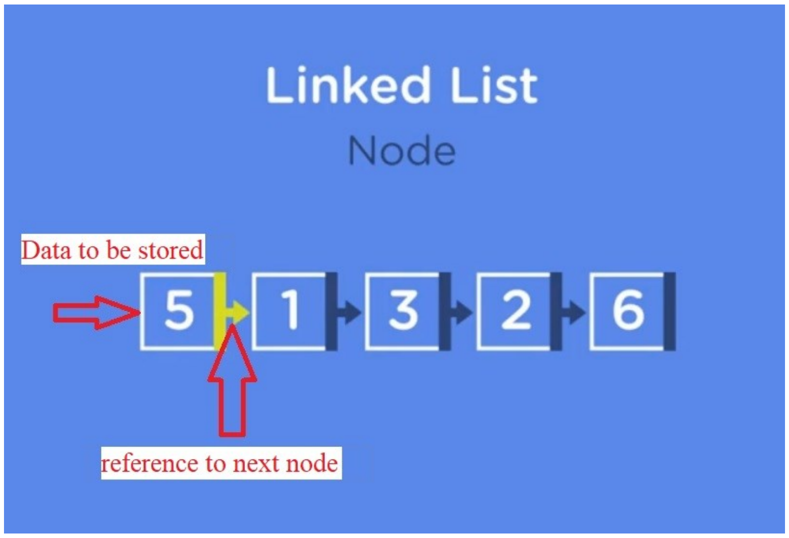


Figure 2.3.1: Identifying a node in a linked list

The first node in the linked list (**5** in the screenshot) is called the **head** of the list. While the last node (**6** in the screenshot) is called the **tale**. The head and the tail nodes are special. The list only maintains a reference to the head, although in some implementations it keeps a reference to the tail as well. This aspect of linked list is very important and, as you'll see, most of the operations on the list need to be implemented quite differently compared to an array. The opposite of the head, the tail, denotes the end of the list. Every node other than the tail points to the next node in the list, but tail doesn't point to anything. This is basically how we know it's the end of the list.

Nodes are what are called **self referential objects**. Figure 2.3.2.

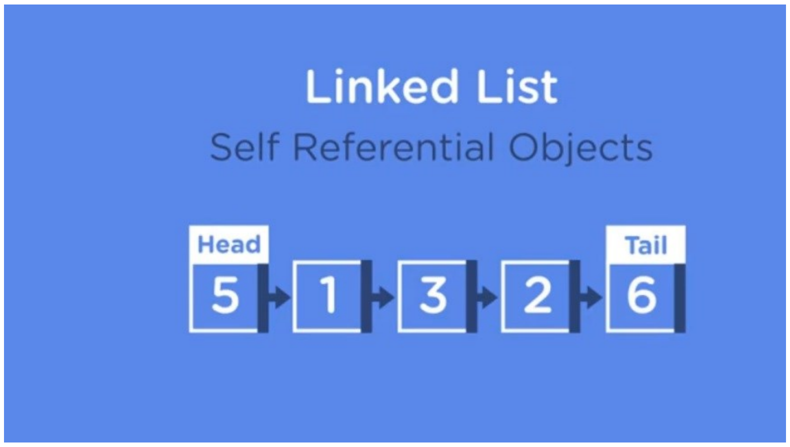


Figure 2.3.2: Nodes in a linked list have heads and a tails

The definition of a node includes a link to another node and self-referential here means the definition of node includes the node itself. Linked lists often come in two forms: a **singly linked list** where each node stores a reference to the next node in the list, or a **doubly linked list** where each node stores a reference to both the node before and after.

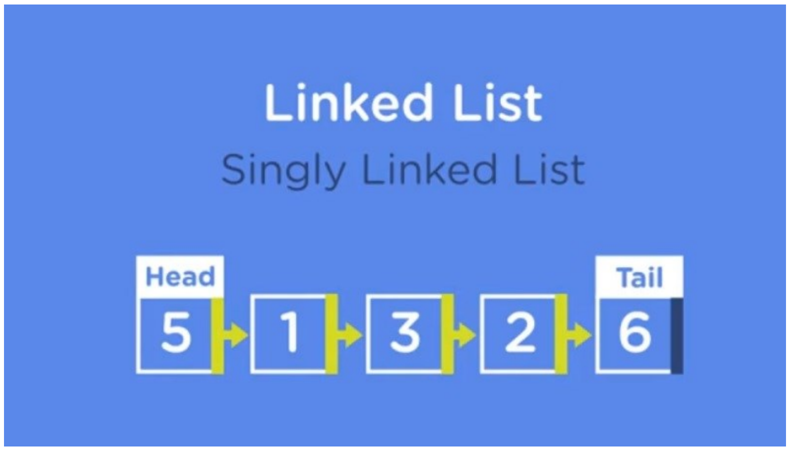


Figure 2.3.3: A singly linked list

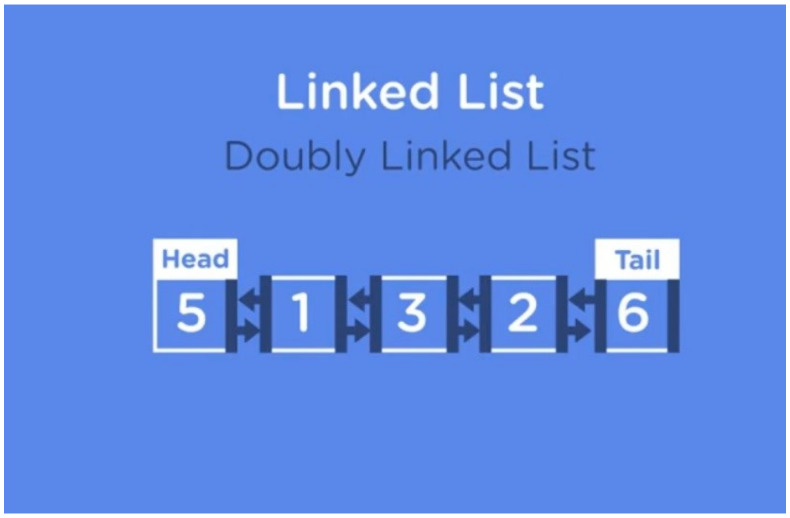


Figure 2.3.4: A doubly linked list

If an array is a train with a bunch of cars in order, then a linked list is like a treasure hunt. When you start the hunt, you have a piece of paper with a location of the first treasure. You go to that location and you find an item along with a location to the next item of treasure. When you finally find an item that doesn't also include a location, you know that the hunt has ended. Now that we have a high-level view of what a linked list is, let's jump into code and build one together.

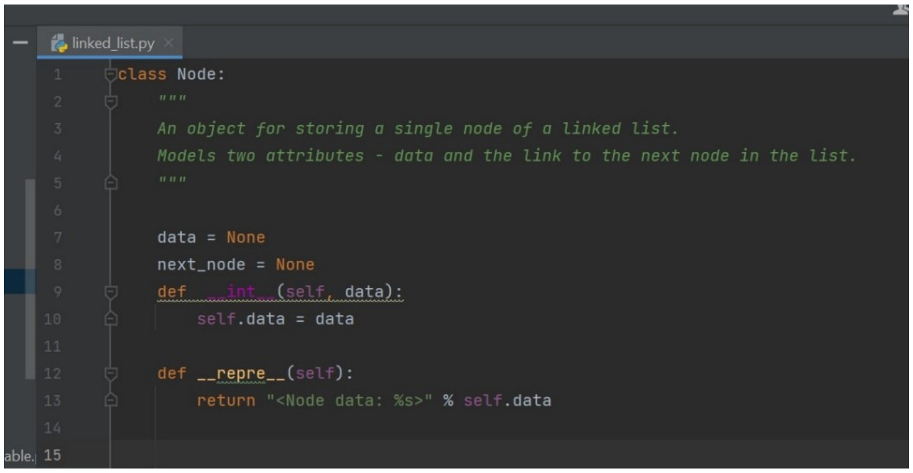
We'll focus on building a singly linked list for this course. There are advantages to having a doubly linked list, but we don't want to get ahead of ourselves. Let's start by creating a new file where we're going to put all our code for our linked list. We'll name this file *linked_list.py*.

First, we're going to create a class to represent a node.

```
class Node:
  data = None
  next_node = None
```

Now, *node* is a simple object in that it won't model much. So first, we'll add a data variable. It's an instance variable here called *data* and we'll assign the value *None* initially. Then we'll add one more and call this *next_node* and to this we'll assign *None* as well. So, we've created two instance variables: *data* to hold onto the data that we're storing and *next_node* to point to the next node in the list.

Now we need to add a constructor to make this class easy to create. So, we'll add *def _ _ini_ _(self,data)* **.** Figure 2.3.5 shows the updated code from my workspace. This will be explained shortly.



```python
class Node:
    """
    An object for storing a single node of a linked list.
    Models two attributes - data and the link to the next node in the list.
    """

    data = None
    next_node = None
    def __int__(self, data):
        self.data = data

    def __repre__(self):
        return "<Node data: %s>" % self.data
```

Figure 2.3.5: Code for a singly linked list

An *init* method here that takes *self* and some data to start off. And all we're going to do is assign data to that instance variable we created. That's all we need to model *Node*. Before we do anything else though, let's document our code. So, right after the class definition, let's create a docstring.

```
"""
An object for storing a single node of a linked list.
Models two attributes - data and the link to the next node in the list.
"""
```

Using the *Node* class is fairly straightforward so we can create a new instance of node with some data to store.

Now, just so you know, every time you run this code you start off from scratch. So, let's go ahead and create it:

```
N1 = Node(10)
print (N1)
```

When you run the code now, we have the following result.

```
<Node data: 10>
```

You can see that we have a node and it contains the data 10. We can now create another node by adding this:

```
 N2 = Node(20)
N1.next_node = N2
```

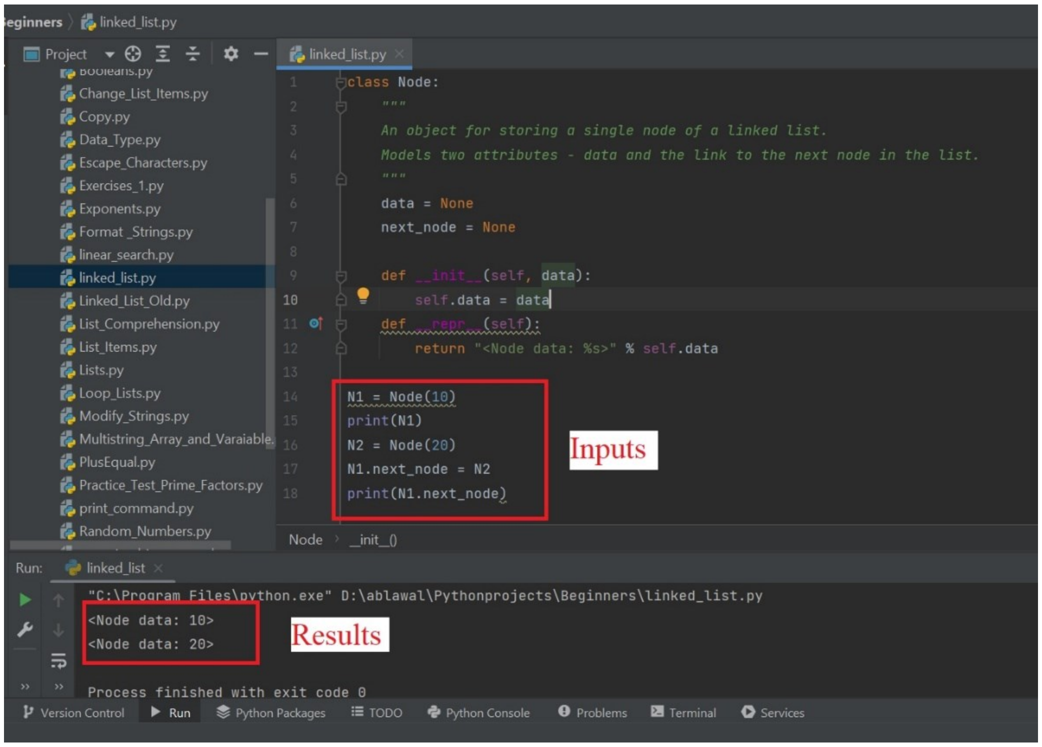
So, N1 now points to N2. If we say

```
print(N1.next_node)
```

you'll see from the following result that it points to that node (the node containing 20).

```
<Node data: 20>
```

Figure 2.3.6 shows the updated code and results inside Pycharm.



```python
class Node:
    """
    An object for storing a single node of a linked list.
    Models two attributes - data and the link to the next node in the list.
    """
    data = None
    next_node = None

    def __init__(self, data):
        self.data = data
    def __repr__(self):
        return "<Node data: %s>" % self.data

N1 = Node(10)
print(N1)
N2 = Node(20)
N1.next_node = N2
print(N1.next_node)
```

```
"C:\Program Files\python.exe" D:\ablawal\Pythonprojects\Beginners\linked_list.py
<Node data: 10>
<Node data: 20>

Process finished with exit code 0
```

Figure 2.3.6: Updated code and results for examples of some nodes

Nodes are the building blocks for a list and now that we have a node object, we can use it to create a singly linked list.

class LinkedList:

def _ _init_ _(self):
self.head = None

We have also created a constructor (def _ _init_ _(self)). This head attribute models the only node that the list will have a reference to. Since every node points to the next node, to find a particular node, we can go from one node to the next in a process called **list traversal**. Since the class constructor here will have set the default value of *head* to *None* so that new lists created are always empty.

Again, you'll notice in the above code that I didn't explicitly declare the *head* attribute at the top of the *class* definition. Don't worry, that's not an oversight. The *self.head* in the initializer means that it's still created. That's all there is to modeling a linked list.

Now we can add methods that make it easier to use this data structure. Also, let's add a really simple docstring below the *class* definition to provide some information.

```
"""
Singly linked list
"""
```

A common operation carried out on data structures is checking whether it contains any data or whether it's empty. At the moment, to check if a list is empty, we would need to query these instance variables head and so on every time. Ideally, we would like to not expose the inner workings of our data structure to code that uses it. Instead, let's make this operation more explicit by defining a method.

```
def is_empty(self):
    return self.head == None
```

All we're doing in the second line is checking to see if head is *None*. If it is, this condition evaluates to true which indicates the list is empty. Now before we end this section, let's add one more convenience method to calculate the size of our list.

The name convenience method indicates that what this method is doing is not providing any additional functionality that our data structure can't handle right now, but instead making existing functionality easier to use. We could calculate the size of our linked list by traversing it every time using a loop until we hit a tail node, but doing that every time is a hassle.

```
def size(self):
```

Unlike calling *len* on a Python list, not to be confused with a linked list, which is a constant time operation, our size operation is going to run in linear time. The only way we can count how many items we have is to visit each node and call next until we hit the tail node. So, we'll start by getting a reference to the head. We'll say

```
  current = self.head
  count = 0
```

Let's also define a local variable named *count* with an initial value of 0 that will increment every time we visit a node. Once we hit the tail count will reflect the size of that list.

Next, we'll define a *while* loop that will keep going until there are no more nodes. So, we say

```
  while current:
```

Now inside this loop, we will increment the count value. So,

```
    count+=1
```

Next, we're going to assign the next node in the list to current. So

current = current.next_node

This way, once we get to the tail and call next node, current will equal *None* and the *while* loop terminates. So in the end we can return count.

return count

As you can see, we need to visit every node to determine the size, meaning our algorithm runs in linear time.

Let's take a break here. We can now create lists, check if they're empty and check the size in the next section. Let's start implementing some common operations.

2.3.2. Adding Nodes to a Linked List

At the moment we can create an empty list but nothing else. Let's define a method to add data to our list. Technically speaking, there are three ways we can add data to a list. We can add nodes at the head of the list, which means that the most recent node we created will be the head and the first node we created will be the tail. Or we could flip that around: most recent nodes are the tail of the list and the first node to be added is the head.

I mentioned that one of the advantages of linked lists over arrays is that inserting data into the list is much more efficient than to the array. This is only true if we're inserting at the head or the tail. Technically speaking, this isn't an insert, and you'll often see this method called **prepend** if the data is added to the head, or **append** if it's added to the tail. A true insert is where you can insert the data at any point in the list, which is our third way of adding data. We're going to circle back on that.

If we wanted to insert at the tail, then the list needs a reference to the tail node, otherwise we would have to start at the head and walk down the length of the list, or traverse it to find the tail. Since our list only keeps a reference to the head, we're going to add new items at the head of the list.

2.3.3. Bonus Algorithm Code

Try the following bonus linked list code written in Python in your favorite Python editor. If you use a different programming language, try to translate it to your own language. If you need further help, just shoot me an email.

```
class Node:

    """

    An object for storing a single node in a linked list
    Attributes:

        data: Data stored in node

        next_node: Reference to next node in linked list

    """


    def __init__(self, data, next_node = None):

        self.data = data

        self.next_node = next_node


    def __repr__(self):

        return "<Node data: %s>" % self.data


class SinglyLinkedList:

    """

    Linear data structure that stores values in nodes. The list maintains a
reference to the first node, also called head. Each node points to the next
node in the list


    Attributes:

        head: The head node of the list

    """


    def __init__(self):

        self.head = None

        # Maintaining a count attribute allows for len() to be implemented in

        # constant time

        self.__count = 0
```

```
def is_empty(self):
    """
    Determines if the linked list is empty
    Takes O(1) time
    """

    return self.head is None

def __len__(self):
    """
    Returns the length of the linked list
    Takesn O(1) time
    """

    return self.__count

def add(self, data):
    """
    Adds new Node containing data to head of the list
    Also called prepend
    Takes O(1) time
    """

    new_head = Node(data, next_node=self.head)
    self.head = new_head
    self.__count += 1

def search(self, key):
    """
    Search for the first node containing data that matches the key
    Returns the node or `None` if not found
    Takes O(n) time
    """

    current = self.head

    while current:
        if current.data == key:
            return current
        else:
            current = current.next_node
    return None

def insert(self, data, index):
    """
    Inserts a new Node containing data at index position
    Insertion takes O(1) time but finding node at insertion point takes
    O(n) time.
    Takes overall O(n) time.
    """

    if index >= self.__count:
        raise IndexError('index out of range')

    if index == 0:
        self.add(data)
        return
    if index > 0:
        new = Node(data)
        position = index
        current = self.head

        while position > 1:
            current = current.next_node
            position -= 1

        prev_node = current
        next_node = current.next_node

        prev_node.next_node = new
        new.next_node = next_node

    self.__count += 1

def node_at_index(self, index):
    """
    Returns the Node at specified index
    Takes O(n) time
    """

    if index >= self.__count:
        raise IndexError('index out of range')

    if index == 0:
        return self.head

    current = self.head
    position = 0

    while position < index:
        current = current.next_node
        position += 1

    return current

def remove(self, key):
    """
    Removes Node containing data that matches the key
    Returns the node or `None` if key doesn't exist
    Takes O(n) time
    """

    current = self.head
    previous = None
    found = False

    while current and not found:
        if current.data == key and current is self.head:
            found = True
            self.head = current.next_node
            self.__count -= 1
            return current
        elif current.data == key:
            found = True
            previous.next_node = current.next_node
            self.__count -= 1
            return current
        else:
            previous = current
            current = current.next_node

    return None

def remove_at_index(self, index):
    """
    Removes Node at specified index
    Takes O(n) time
    """

    if index >= self.__count:
        raise IndexError('index out of range')

    current = self.head

    if index == 0:
        self.head = current.next_node
        self.__count -= 1
        return current

    position = index
    while position > 1:
        current = current.next_node
        position -= 1

    prev_node = current
    current = current.next_node
    next_node = current.next_node

    prev_node.next_node = next_node
    self.__count -= 1

    return current


def __iter__(self):
    current = self.head

    while current:
        yield current
        current = current.next_node


def __repr__(self):
    """
    Return a string representation of the list.
    Takes O(n) time.
    """
```

```
nodes = []

current = self.head

while current:

    if current is self.head:

        nodes.append("[Head: %s]" % current.data)

    elif current.next is None:

        nodes.append("[Tail: %s]" % current.data)

    else:

        nodes.append("[%s]" % current.data)

    current = current.next_node

return  '-> '.join(nodes)
```

2.4. Circularly Linked lists

2.4.1. Representation

A circularly linked list, circular linked list or simply a circular list is a type of data structure that consists of a collection of nodes, where each node contains a data element and a pointer to the next node in the list. In a circularly linked list, the last node in the list points back to the first node, forming a circle or loop.

This circular linking structure has a number of advantages over a standard linked list discussed earlier, such as the ability to traverse the list in a circular manner without having to restart at the beginning of the list. It also allows for easy insertion and deletion of nodes at any position in the list.

Circular linked lists can be implemented in a variety of programming languages, and are commonly used in various applications such as music playlists, computer game data structures, and memory management algorithms

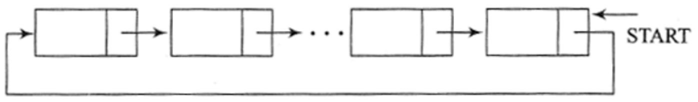


Figure 2.4.1: Illustration of circularly linked list

2.4.2. Advantages of circularly linked lists over singly linked lists

There are several advantages of circularly linked lists over singly linked lists:

1. **Traversing the list**: In a circular linked list, it is easy to traverse the entire list, as the last node is connected to the first node. This means that you can start at any node in the list and move in either direction to traverse the entire list.

2. **Insertion and deletion**: Circular linked lists make it easy to insert or delete nodes anywhere in the list. This is because you only need to update the pointers of the neighboring nodes, whereas in singly linked lists, you need to find the previous node to update its pointer.

3. **Efficient use of memory**: Since the last node in a circular linked list points back to the first node, there is no need to allocate memory for a separate "null" pointer. This can lead to more efficient use of memory.

4. **Applications**: Circular linked lists are useful in various applications, such as music

playlists, computer game data structures, and memory management algorithms.

However, circular linked lists also have some disadvantages, such as increased complexity in implementation and potential for infinite loops if the pointers are not set correctly. Therefore, the choice between a circular linked list and a singly linked list depends on the specific needs of the application.

2.5. Doubly Linked Lists

A doubly linked list is a type of data structure that consists of a collection of nodes, where each node contains a data element and two pointers: one to the previous node in the list and one to the next node in the list.

In a doubly linked list, you can traverse the list in both forward and backward directions, as each node has a pointer to both the previous and the next nodes. This makes it easier to insert or delete nodes at any position in the list, as you can move in either direction to find the node you want to modify.

The first and last nodes in a doubly linked list are called the **head** and **tail**, respectively. They are special nodes that do not have a previous or next node, respectively. A doubly linked list can also be circular, in which case the tail node points back to the head node, forming a loop.

Doubly linked lists are used in various applications, such as music playlists, computer game data structures, and text editors. However, they require more memory than singly linked lists, as each node needs to store two pointers instead of one. Additionally, implementing a doubly linked list can be more complex than implementing a singly linked list, as you need to update two pointers when inserting or deleting a node.

2.5.1. Representation of a doubly linked list

Figure 2.5.1 illustrates the structure of a node in a doubly linked list and the various types of lists.

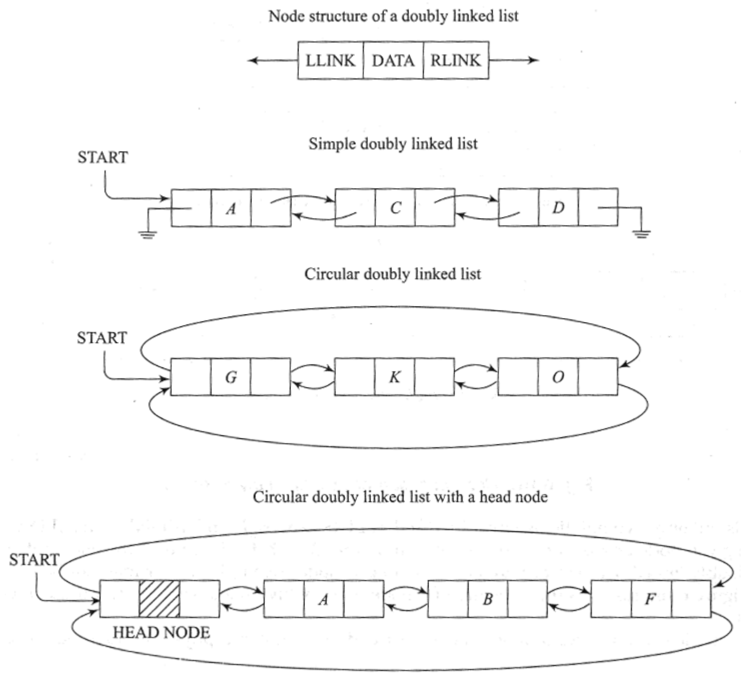

Node structure of a doubly linked list
LLINK
DATA
RLINK
Simple doubly linked list
START
A
C
D
Circular doubly linked list
START
G
K
O
Circular doubly linked list with a head node
START
A
B
F
HEAD NODE

Figure 2.5.1: Illustration of the structure of a node in a doubly linked list and the various types of lists

2.5.2. Advantages of a Doubly Linked List

Doubly linked lists offer several advantages over other types of linked lists:

1. **Bidirectional traversal**: The most significant advantage of a doubly linked list over a singly linked list is that it can be traversed in both forward and backward directions. This is because each node has two pointers, one pointing to the previous node and one to the next node. This feature is particularly useful when you need to search or modify nodes in the middle of the list.
2. **Efficient insertion and deletion**: Doubly linked lists make it easy to insert or delete nodes at any position in the list. In a singly linked list, you need to find the previous node to update its pointer when inserting or deleting a node. In contrast, in a doubly linked list, you can easily update the pointers of the neighboring nodes, which makes the operation more efficient.
3. **Easy to implement stack and queue data structures**: Doubly linked lists can be used to implement stack and queue data structures. Stacks use the "last-in, first-out" (LIFO) approach, while queues use the "first-in, first-out" (FIFO) approach. The doubly linked list is a good fit for implementing these data structures because it allows efficient insertion and deletion operations at both ends of the list.
4. **Reverse traversal**: In certain applications, such as text editors, it is often necessary to traverse a list in reverse order. This is easily achievable in a doubly linked list, as each node has a pointer to the previous node.

Overall, the bidirectional traversal and efficient insertion and deletion operations make doubly linked lists a popular choice for certain types of data structures and algorithms.

2.5.3. Disadvantages of a Doubly Linked List

Doubly linked lists have a few disadvantages compared to other types of linked lists:

1. **Increased memory usage**: Doubly linked lists require more memory than singly linked lists because each node has to store two pointers instead of one.
2. **Increased complexity**: Implementing a doubly linked list can be more complex than implementing a singly linked list. This is because you need to keep track of both the previous and next nodes when inserting or deleting a node.
3. **Additional pointers to maintain**: In a singly linked list, you only need to maintain a pointer to the next node. However, in a doubly linked list, you need to maintain pointers to both the previous and next nodes, which can make the code more complex and error-prone.
4. **Extra care required when modifying pointers**: Because each node has two pointers, it's important to be careful when modifying them to avoid creating circular references or other issues that could cause the list to break or become corrupted.
5. **Slightly slower than singly linked lists**: Due to the extra pointer that needs to be maintained, doubly linked lists may have a slightly slower performance than singly linked lists.

Despite these disadvantages, doubly linked lists are still a popular data structure choice for certain applications that require bidirectional traversal or efficient insertion and deletion operations.

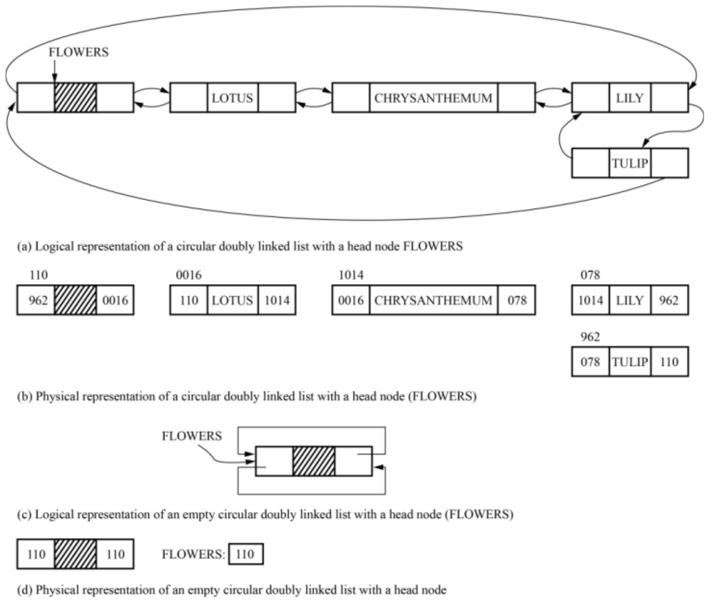

FLOWERS
LOTUS
CHRYSANTHEMUM
LILY
TULIP
(a) Logical representation of a circular doubly linked list with a head node FLOWERS
110
962
0016
0016
110
LOTUS
1014
1014
0016
CHRYSANTHEMUM
078
078
1014
LILY
962
962
078
TULIP
110
(b) Physical representation of a circular doubly linked list with a head node (FLOWERS)
FLOWERS
(c) Logical representation of an empty circular doubly linked list with a head node (FLOWERS)
110
110
FLOWERS: 110
(d) Physical representation of an empty circular doubly linked list with a head node

Figure 2.5.2: The logical and physical representation of a circular doubly linked list with a head node, FLOWERS

2.5.4. Multiply Linked List

A multiply linked list is a type of linked list in which each node has multiple pointers, allowing for more complex data structures to be created.

As explained earlier, in a singly linked list, each node has only one pointer to the next node in the list. In a doubly linked list, each node has two pointers, one to the previous node and one to the next node. But in a multiply linked list, each node can have multiple pointers to other nodes, allowing for more complex relationships between nodes.

A common use case for a multiply linked list is in a graph data structure. Each node in the graph can be represented as a node in the multiply linked list, with pointers to other nodes representing the edges in the graph.

Another use case for a multiply linked list is in a hierarchical data structure such as a tree. Each node in the tree can be represented as a node in the multiply linked list, with pointers to child nodes representing the hierarchy.

In general, multiply linked lists allow for more complex and flexible data structures to be created, but they can be more complex to implement and maintain than simpler linked list structures.

A general node structure of a multiply linked list is shown in Figure 2.6.3. Since each link field connects a group of nodes representing the data elements of a global list L, the multiply linked representation of the list L is a network of nodes that are connected to one another based on some association. The link fields may or may not render their respective lists to be circular, or may or may not possess a head node.

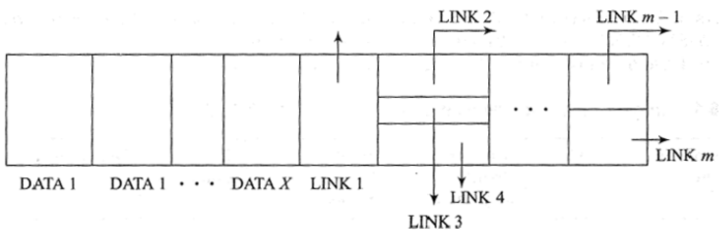


Figure 2.5.3: The node structure of a multiply linked list

2.5.5. Unrolled Linked List

An unrolled linked list is a type of linked list that stores multiple elements, or keys, in each node. In an unrolled linked list, each node consists of a fixed-sized array that can hold a certain number of elements, as well as a pointer to the next node in the list.

The idea behind an unrolled linked list is to reduce the overhead of allocating and deallocating memory for each individual element in the list. By storing multiple elements in each node, the number of allocations and deallocations required is reduced, which can improve performance.

The name "unrolled" comes from the fact that the array in each node is "unrolled" or flattened out, as opposed to a traditional linked list where each node contains only a single element and a pointer to the next node.

Unrolled linked lists are particularly useful for applications where insertions and deletions are frequent, as they can reduce the overhead of memory management. They are also useful in situations where the size of the elements being stored is fixed, as the size of each node in the list can be made to match the size of the elements. However, they may be less efficient for applications that require frequent random access to individual elements in the list.2.5.6. Download Training Resources

2.6. Download Training Resources

Here's the link you can use to download the codes we wrote, screenshots used in this book, more practice exercises and other training resources:

https://ln5.sync.com/dl/64c032e70/n4epdxsk-4xz59by6-k4r5puid-x9wqpmre

Alternatively, you can use this short BIT.LY link:

https://bit.ly/3o9Djhv

These resources are updated regularly to provide further help. If you need to contact me for anything, or if you want to share your codes with me, use my email address below. I'll get back to you very quickly. I promise.

Cheers,
Ojula Technology Innovations
OjulaTech@gmail.com
www.ojulaweb.com