L - 03

Data Structure and Operations

Data structures:

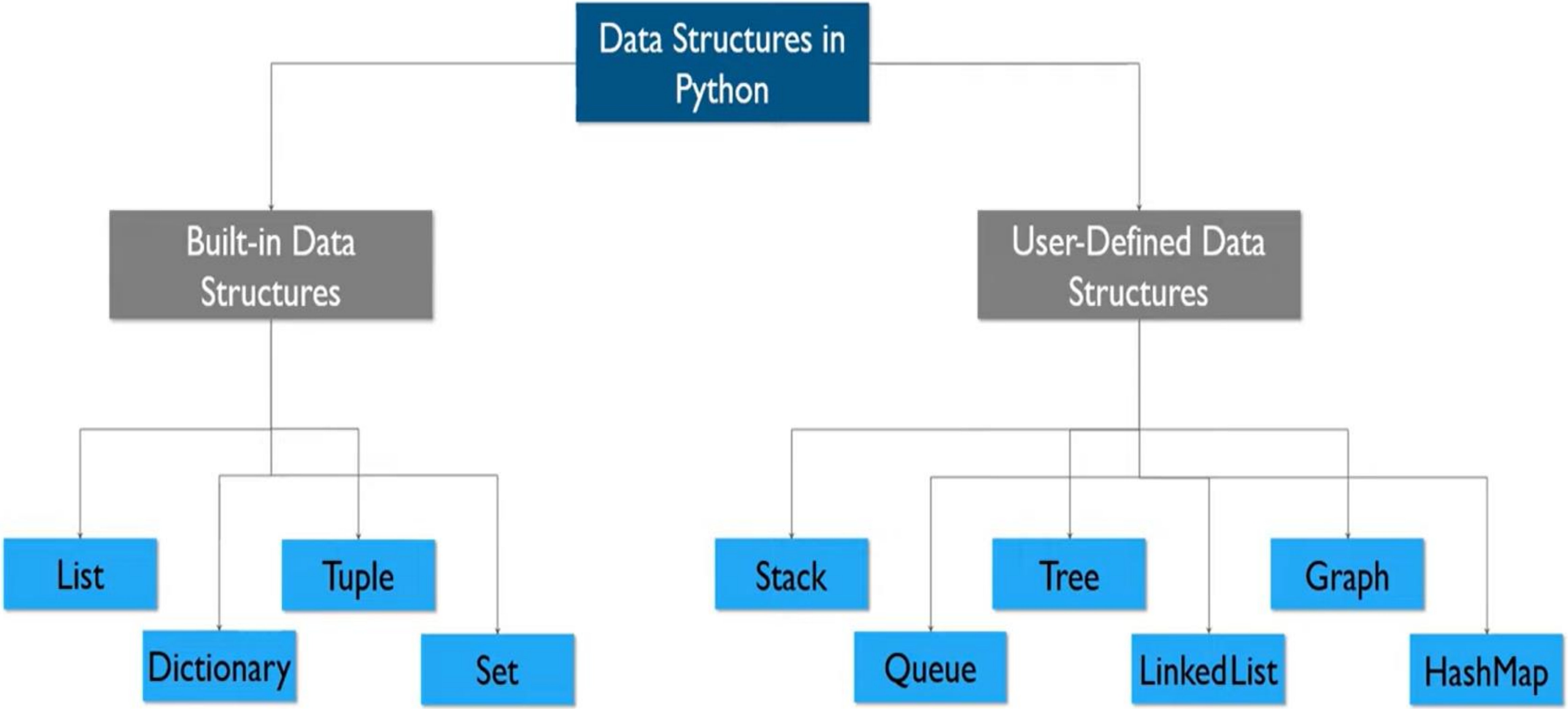
- Use of data structures in application development, e.g. stack, array, multi-dimensional array, set, queue, list and linked list.
- Apply tree types, including active, passive and recursive.

Operations:

- Use of operations in application development, e.g. hash functions and pointers.
- Utilise sorts, e.g. insertion, quick, merge and heap.
- Utilise searches, e.g. linear, binary tree and recursive

What is Data Structure

- A data structure is a way of organizing and storing data in a computer so that it can be accessed and used efficiently. It refers to the logical or mathematical representation of data, as well as its implementation in a computer program
- It allows us to manage and manipulate data effectively, enabling faster access, insertion, and deletion operations and provide a means of handling information, rendering the data for easy use.


Data Structures in Python
Built-in Data Structures
User-Defined Data Structures
List
Dictionary
Tuple
Set
Stack
Queue
Tree
LinkedList
Graph
HashMap

List

- Lists are used to store multiple items in a single variable.
- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0], the second item has index [1] etc.
- Eg.

```
mylist = ["apple", "banana", "cherry"]
```

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Dictionary

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values.
- Eg.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Tuple

- Tuples are used to store multiple items in a single variable.
- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.
- A tuple is a collection which is ordered and unchangeable.
- Tuples are written with round brackets.
- Eg.

```
mytuple = ("apple", "banana", "cherry")
```

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Sets

- Sets are used to store multiple items in a single variable.
- A set is a collection which is unordered, unchangeable*, and unindexed.
- Sets are written with curly brackets.
- Eg.

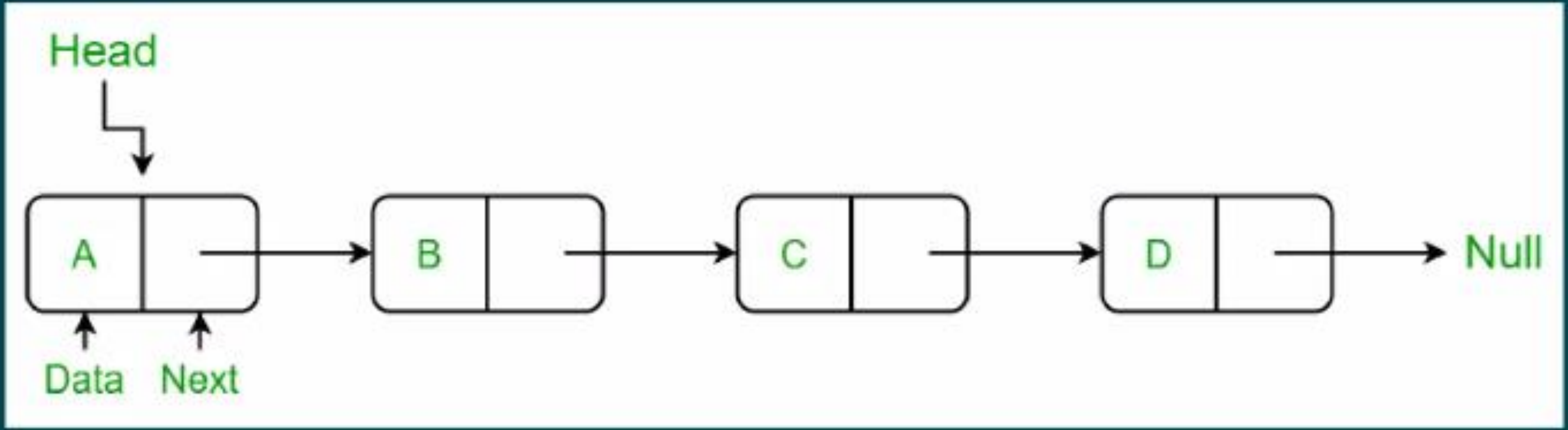
```
myset = {"apple", "banana", "cherry"}
```

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element

Stack, Queue, Tree, Linked list, Graph, HashMap

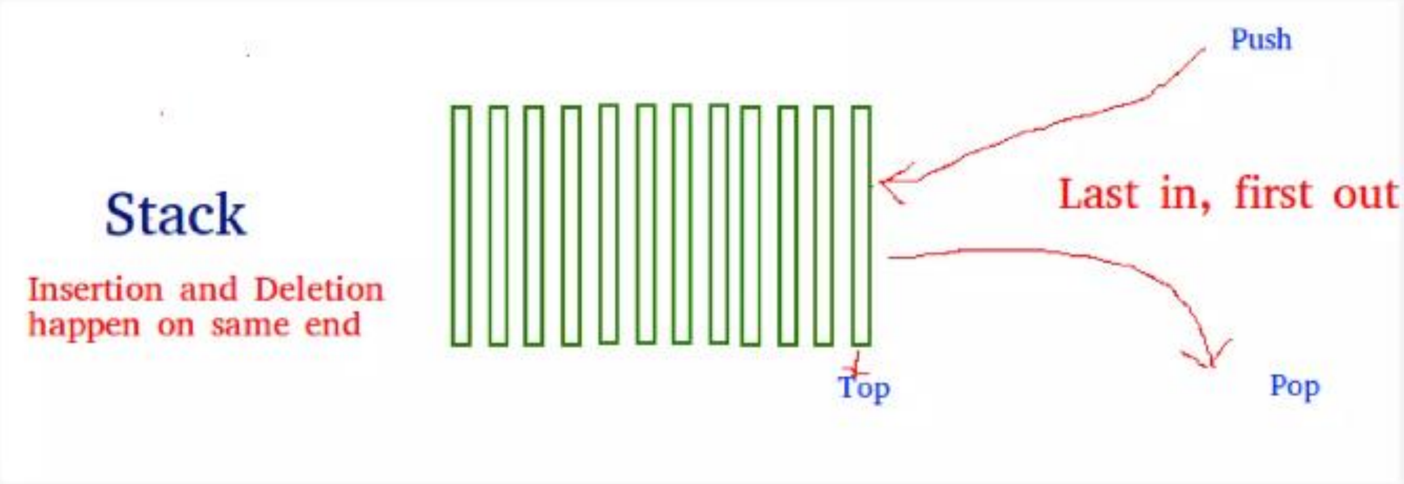
Linked List

Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers. They include a series of connected nodes. Here, each node stores the data and the address of the next node.



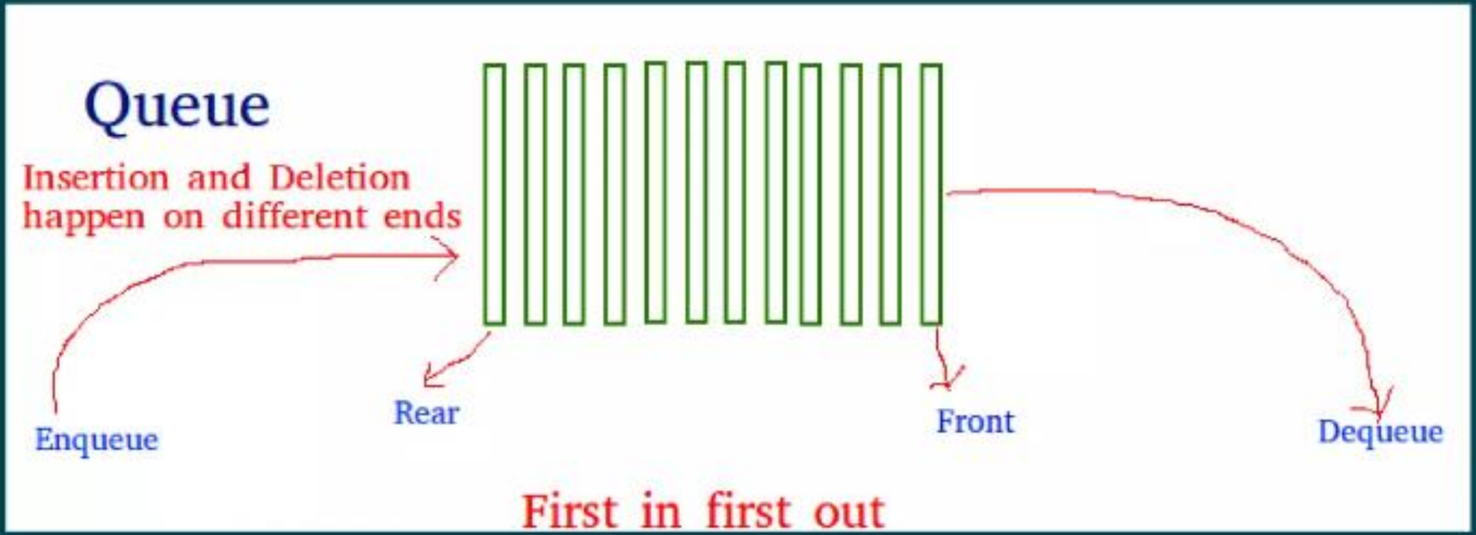
Stack

A stack is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.



Queue

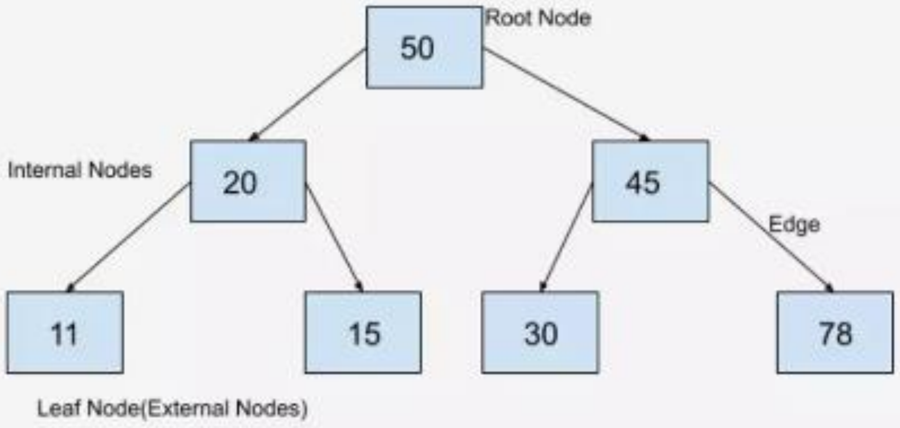
Like stack, queue is a linear data structure that stores items in First In First Out (FIFO) manner. With a queue the least recently added item is removed first. A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.



Tree

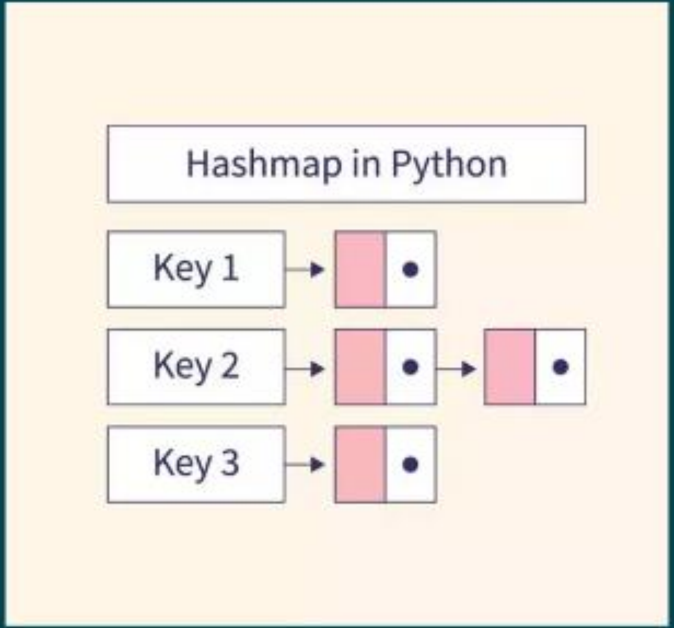
Tree represents the nodes connected by edges. It is a non-linear data structure. It has the following properties –

- One node is marked as Root node.
- Every node other than the root is associated with one parent node.
- Each node can have an arbiatry number of chid node.



Hash Map

Hash maps are indexed data structures. A hash map makes use of a hash function to compute an index with a key into an array of buckets or slots. Its value is mapped to the bucket with the corresponding index. The key is unique and immutable. Think of a hash map as a cabinet having drawers with labels for the things stored in them. For example, storing user information- consider email as the key, and we can map values corresponding to that user such as the first name, last name etc to a bucket.



Algorithms

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.
- From the data structure point of view, following are some important categories of algorithms –
- Search – Algorithm to search an item in a data structure.
- Sort – Algorithm to sort items in a certain order.
- Insert – Algorithm to insert item in a data structure.
- Update – Algorithm to update an existing item in a data structure.
- Delete – Algorithm to delete an existing item from a data structure.

What is Sorting in Python?

Sorting refers to arranging available data in a particular format

Example:
Arranging or sorting a list L1 in descending order;

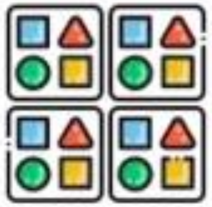
L1 = [2,4,6,7,3,1,5,9,8]

L1 = [9,8,7,6,5,4,3,2,1]

Searches for an item in a list is made faster and easier with sorting

Sorting is also used to represent data in more readable formats

Sorting helps locate various patterns in the data



Duplicate values can be found in a list very quickly when the list is sorted

Sorting Algorithms

Python's Built-In Sorting method - sorted()

By default, the **sorted()** method sorts the list in ascending order

L1 = [2,4,6,7,3,1,5,9,8]

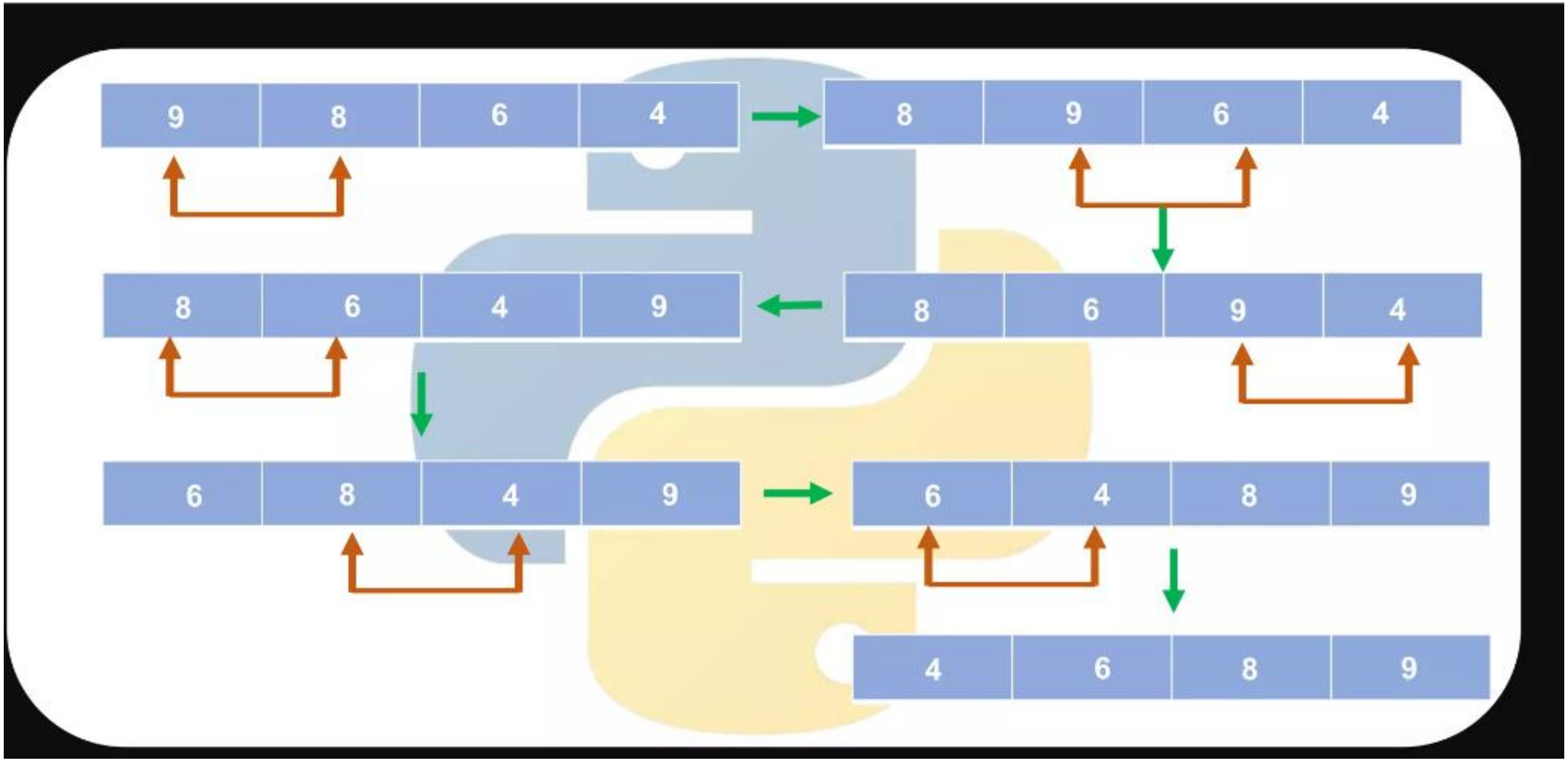
↓

L1 = [1,2,3,4,5,6,7,8,9]

Bubble Sort

- In Bubble Sort Algorithm, the first element is compared with the adjacent element.
- If the adjacent element is smaller than the first element, the elements are swapped.
- The algorithm then compares the second element with its adjacent element and the process continues to bubble the largest element to the right side of the list and finally sorts the list in ascending order.

Bubble Sort

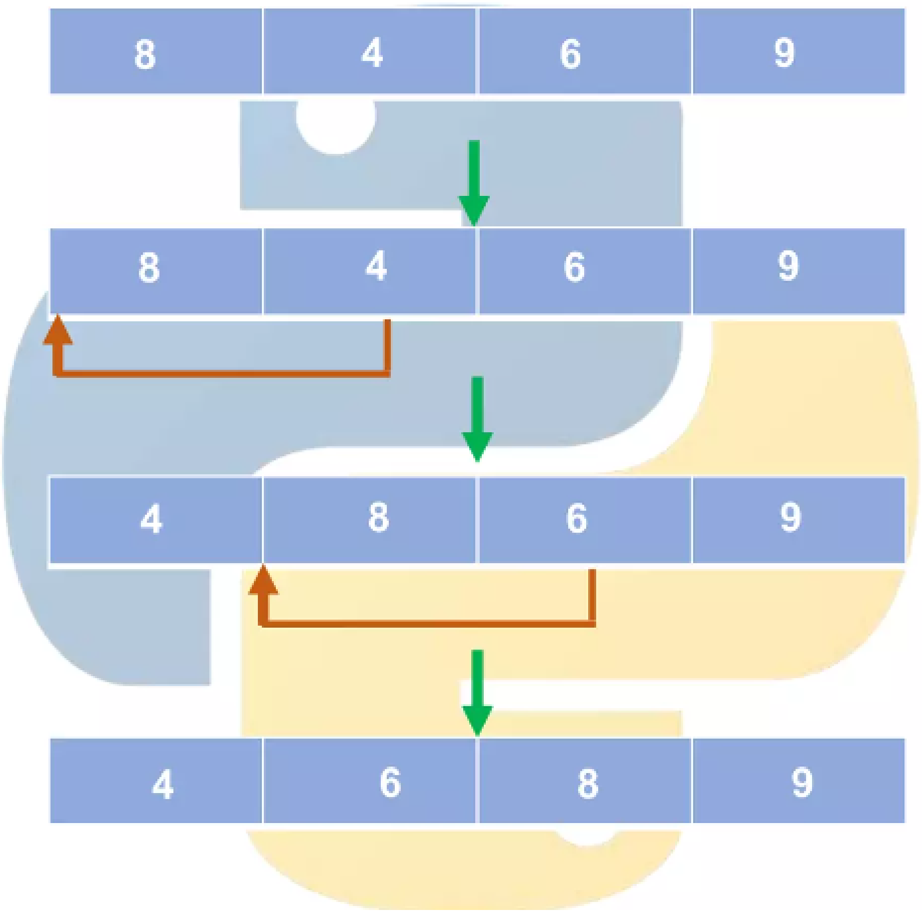


Insertion Sort

Insertion sort places a given element at the right position in a sorted list. So, in the beginning, we compare the first two elements and sort them by comparing them

Then we pick the third element and find its proper position among the last two sorted elements. This process continues till all elements land in their proper positions

Insertion Sort

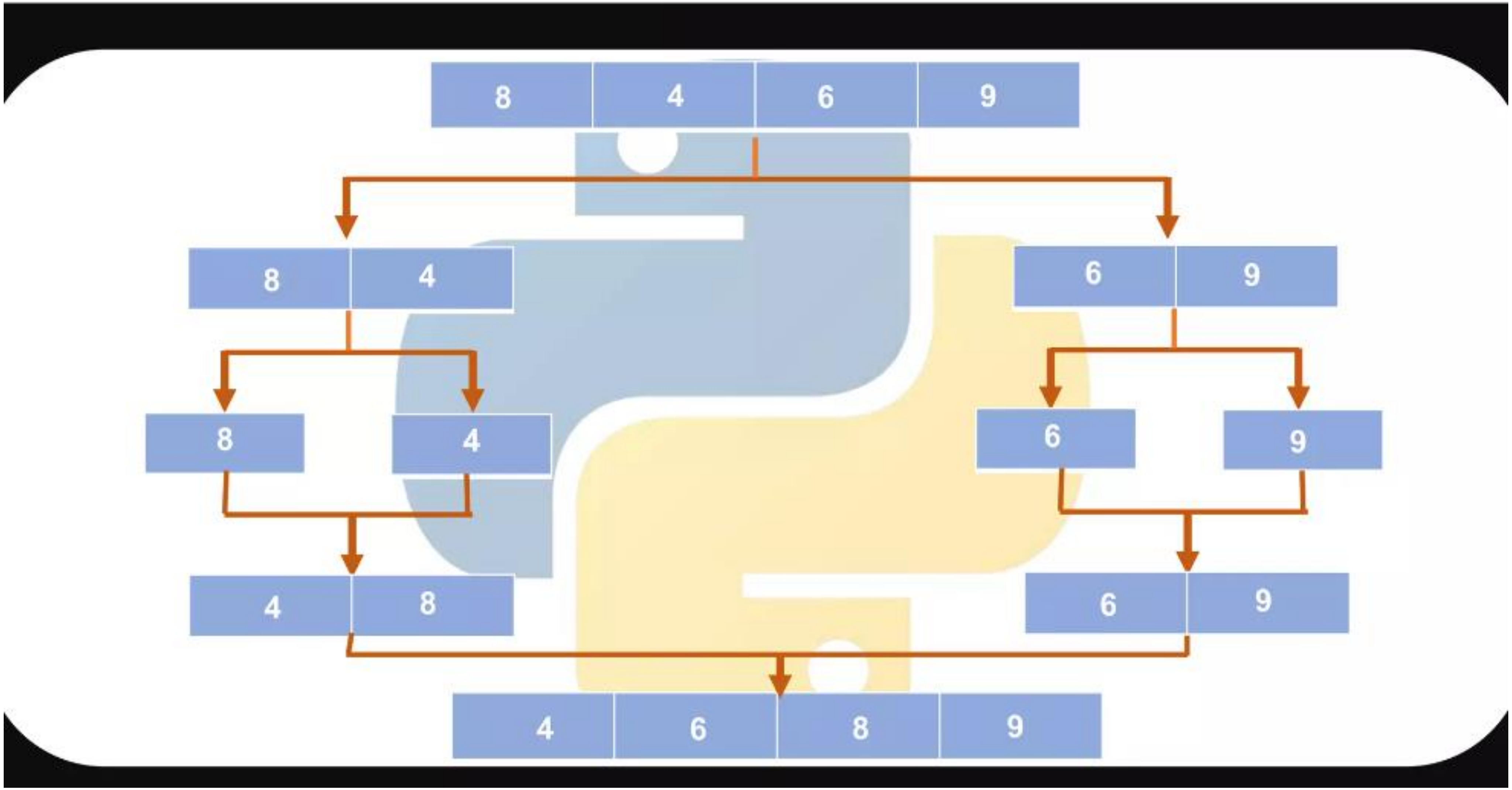


Merge Sort

Merge sort is a Divide and Conquer algorithm

Merge sort first divides the list into equal parts and then sorts the parts and unites them as a sorted list

Merge Sort



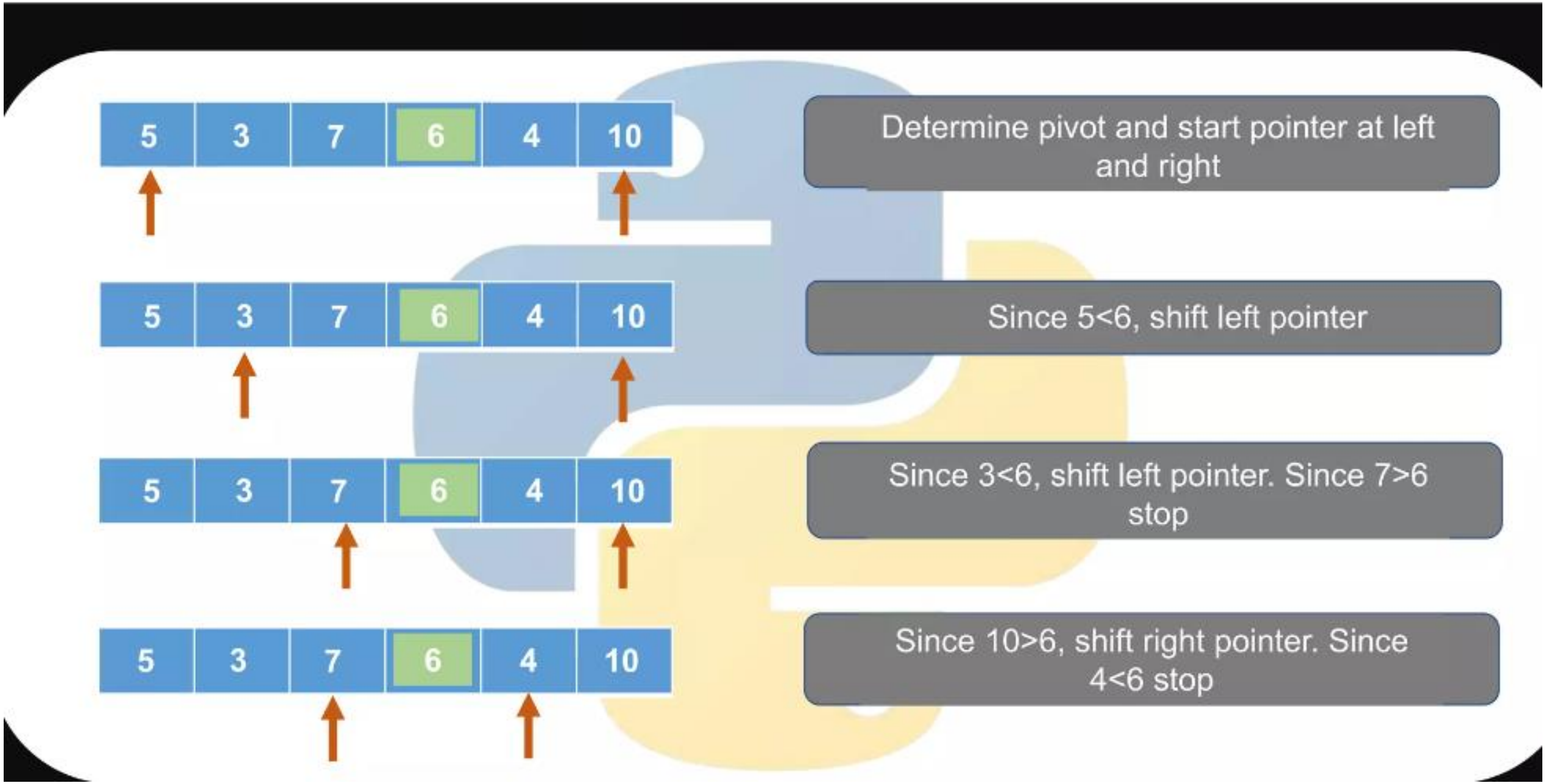
Quick Sort

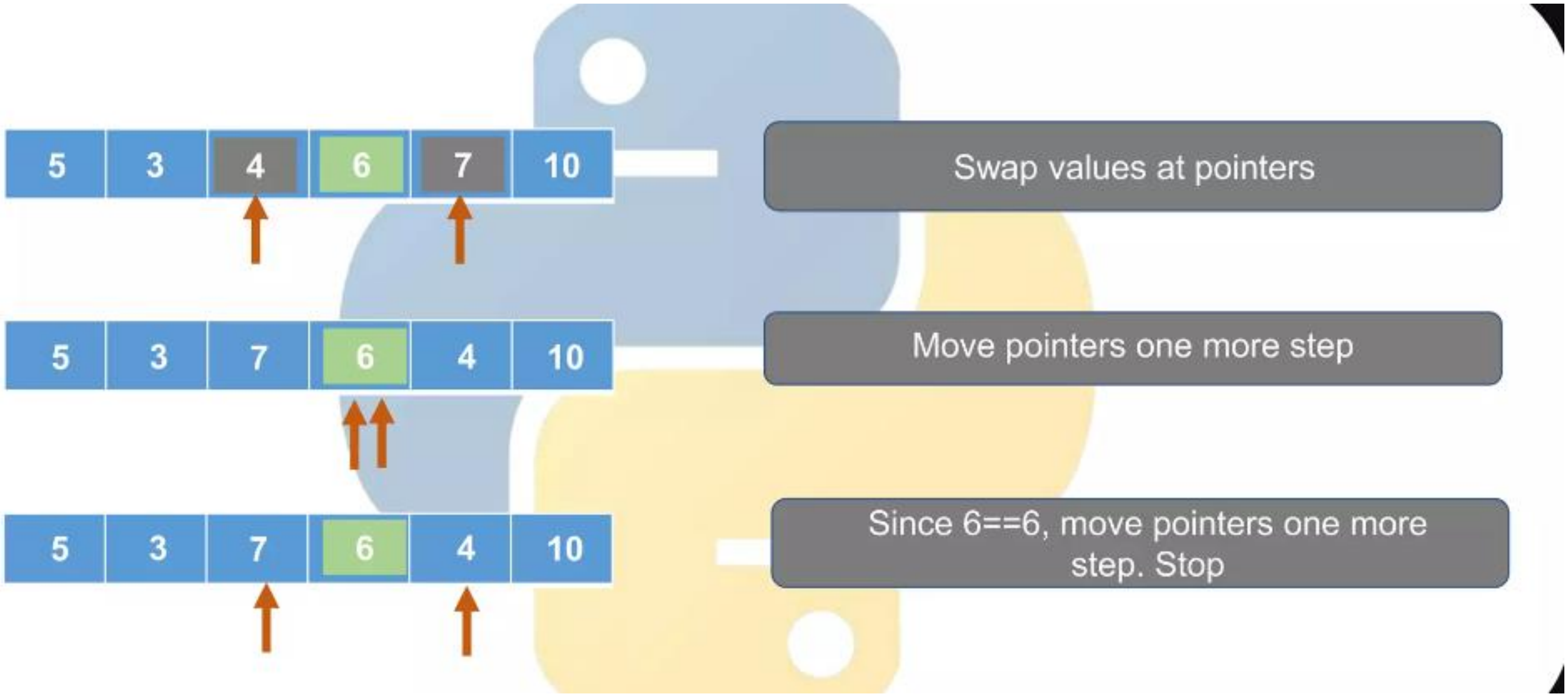
Quicksort is also a Divide and Conquer algorithm

First, you choose a pivot element in the array. Then stores elements less than pivot in left subarray and elements greater than pivot in right subarray and all this is done in linear time

After that, we will call quicksort on the left subarray and similarly, we will call quicksort on the right subarray

Sorting Algorithms




5 3 4 6 7 10
Swap values at pointers
5 3 7 6 4 10
Move pointers one more step
5 3 7 6 4 10
Since 6==6, move pointers one more step. Stop

What is searching?

- In computer science, searching is the process of finding an item with specified properties from a collection of items.
- The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs, or maybe be elements in other search place.

- The **definition** of a **search** is the process of looking for something or someone

Why do we need searching?

✓Searching is one of the core computer science algorithms.

✓We know that today's computers store a lot of information.

✓To retrieve this information proficiently we need very efficient searching algorithms.

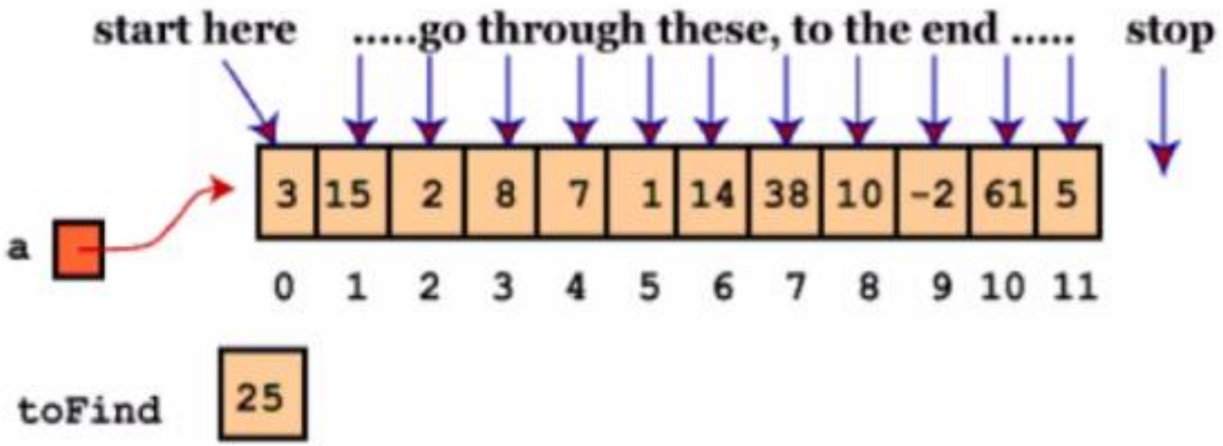
Types of Searching

- Linear search
- Binary search

Linear Search (LS)

Linear Search involves checking all the elements of the array (or any other structure) one by one and in sequence until the desired result is found.

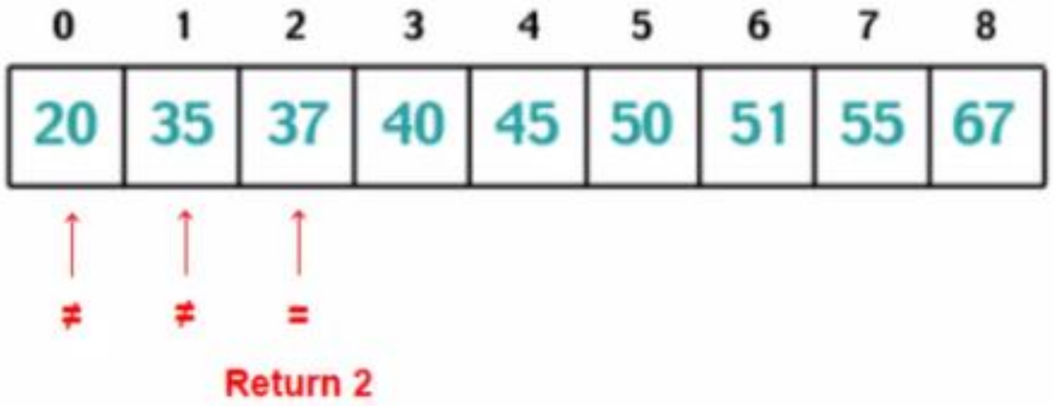
Graphical Illustration of LS



Every item is checked but no match is found till the end of the data collection

Graphical Illustration of LS

- Find 37?



Found a match at index 2

Linear Search Algorithm

- Linear Search (Array A, Value x)

- Step 1: Set i to 1
- Step 2: if i > n then go to step 7
- Step 3: if A[i] = x then go to step 6
- Step 4: Set i to i + 1
- Step 5: Go to Step 2
- Step 6: Print Element x Found at index i and go to step 8
- Step 7: Print element not found
- Step 8: Exit

Adv. & Disadv. Of LS

- Advantages
  - Easiest to understand and implement
  - No sorting required
  - Suitable for small list sizes
  - Works fine for small number of elements

- Disadvantages
  - Time inefficient as compared to other algorithms
  - Not suitable for large-sized lists
  - Search time increases with number of elements

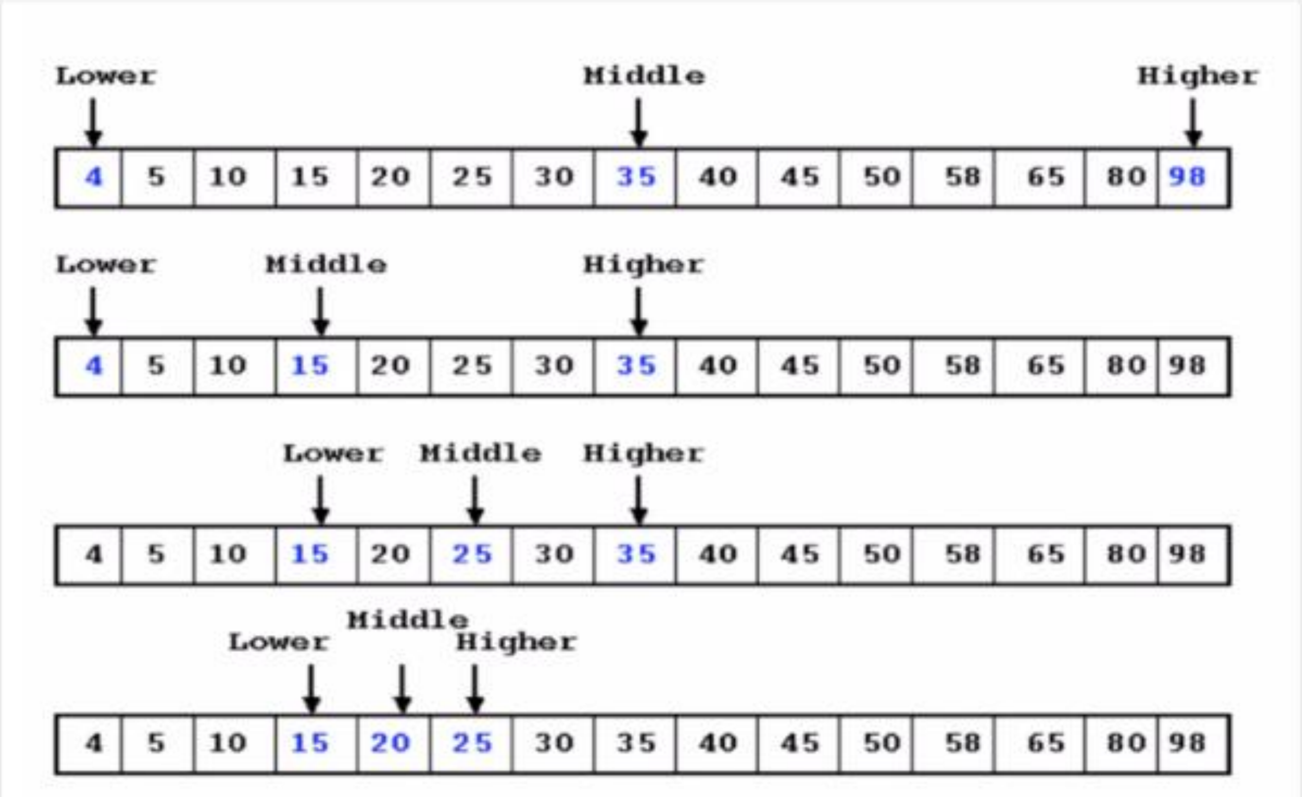
Binary Search (BS)

- Binary Search is a Divide and Conquer algorithm
- Binary search algorithm finds the position of a target value within a sorted array
- A more efficient approach than Linear Search because Binary Search basically reduces the search space to half at each step

Binary Search

- The algorithm begins by comparing the target value to the value of the middle element of the sorted array
- If they are equal the middle position is returned and the search is finished
- If the target value is less than the middle element's value, then the search continues on the lower half of the array;
- If the target value is *greater* than the middle element's value, then the search continues on the upper half of the array
- This process continues, eliminating half of the elements until the value is found

Graphical Illustration of BS



Graphical Illustration of BS

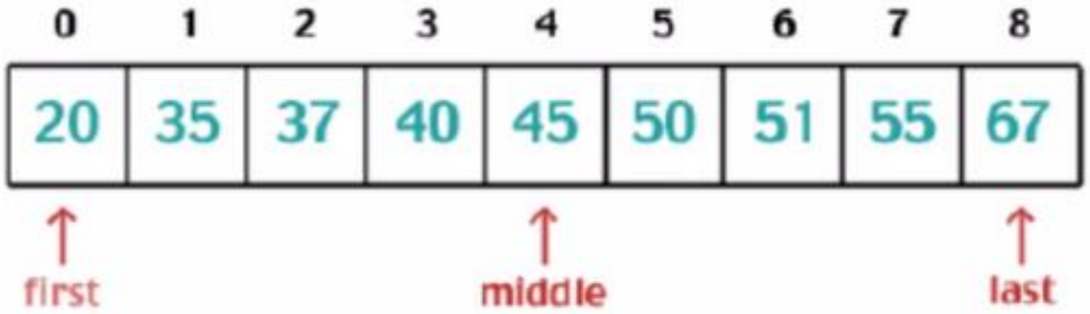
- Find 37?
  1. Sort Array.

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

Graphical Illustration of BS

2. Calculate middle = (low + high) / 2.

 = (0 + 8) / 2 = 4.



If 37 == array[middle] → return middle

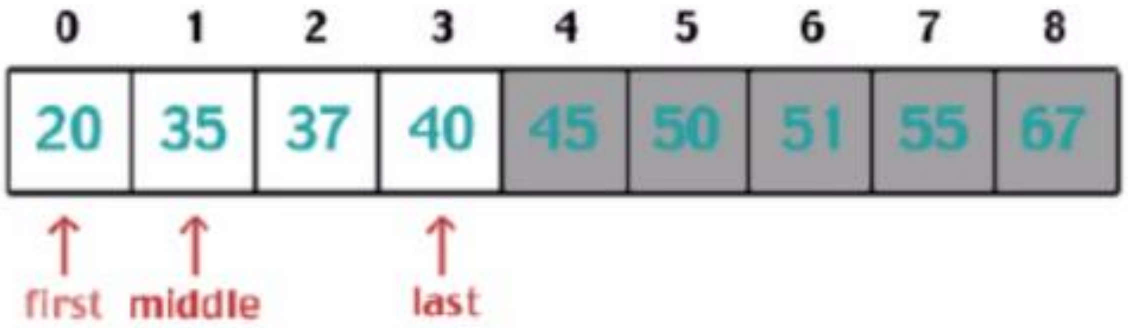
Else if 37 < array[middle] → high = middle -1

Else if 37 > array[middle] → low = middle +1

Graphical Illustration of BS

Repeat 2. Calculate middle = (low + high) / 2.

= (0 + 3) / 2 = 1.



If 37 == array[middle] ➔ return middle

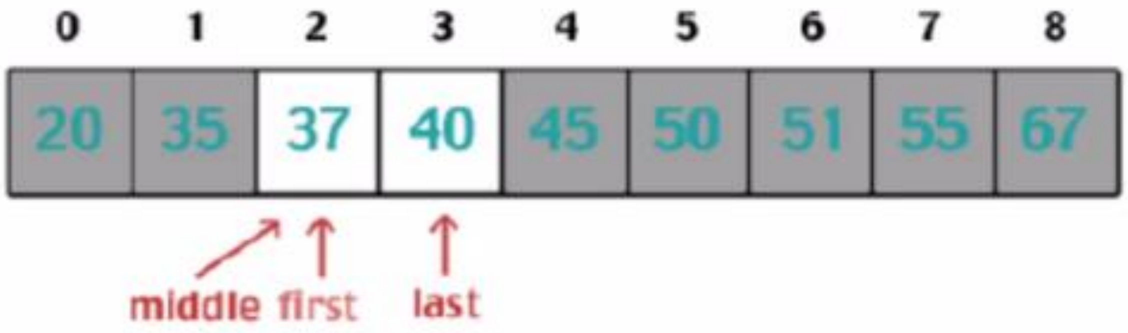
Else if 37 < array[middle] ➔ high = middle -1

Else if 37 > array[middle] ➔ low = middle +1

Graphical Illustration of BS

Repeat 2. Calculate middle = (low + high) / 2.

= (2 + 3) / 2 = 2.

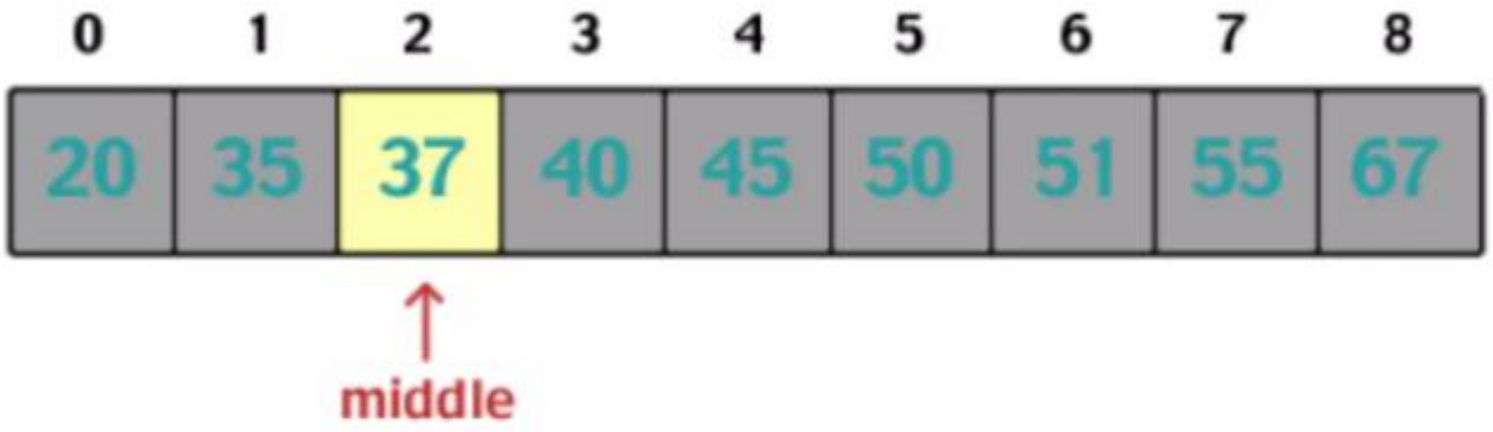


If 37 == array[middle] ➔ return middle

Else if 37 < array[middle] ➔ high = middle -1

Else if 37 > array[middle] ➔ low = middle +1

Graphical Illustration of BS



Binary Search

- With each test that fails to and a match, the search is continued with one or other of the two sub-intervals, each at most half the size
- If the original number of items is N then after the first iteration there will be at most N/2 items remaining, then at most N/4 items, and so on
- In the worst case, when the value is not in the list, the algorithm must continue iterating until the list is empty