Data Structure Overview

Data structures are fundamental concepts of computer science which helps in writing efficient programs in any language. Python is a high-level, interpreted, interactive and object-oriented scripting language using which we can study the fundamentals of data structure in a simpler way as compared to other programming languages.

What are Data Structures?

Data structures are a way of storing and organizing data efficiently. This will allow you to easily access and perform operations on the data.

There is no one-size-fits-all kind of model when it comes to data structures. You will want to store data in different ways to cater to the need of the hour. Maybe you want to store all types of data together, or you want something for faster searching of data, or maybe something that stores only distinct data items.

Luckily, Python has a host of in-built data structures that help us to easily organize our data. Therefore, it becomes imperative to get acquainted with these first so that when we are dealing with data, we know exactly which data structure will solve our purpose effectively.

The various data structures in computer science are divided broadly into two categories shown below. We will discuss about each of the below data structures in detail in subsequent chapters.

Liner Data Structures

These are the data structures which store the data elements in a sequential manner.

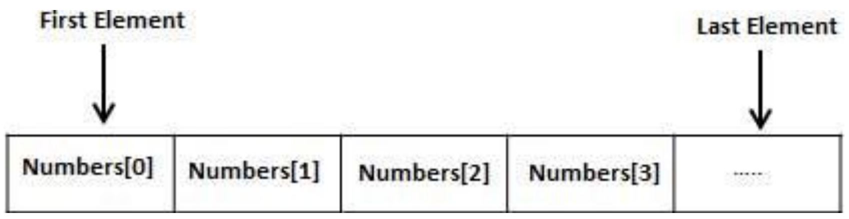
- **Array:** It is a sequential arrangement of data elements paired with the index of the data element.
- **Stack:** It is a data structure which follows only to specific order of operation. LIFO(last in First Out) or FILO(First in Last Out).
- **Queue:** It is similar to Stack but the order of operation is only FIFO(First In First Out).
- **Linked List:** Each data element contains a link to another element along with the data present in it.

Array

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Stack

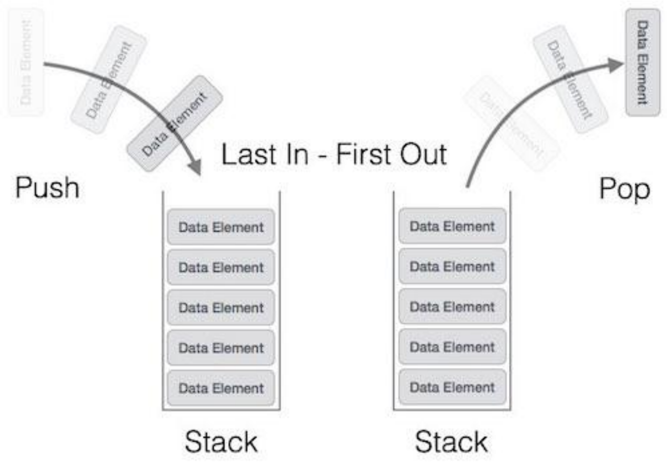
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

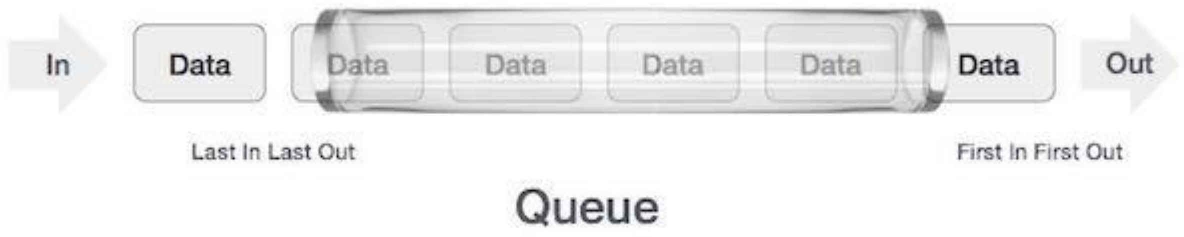
Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

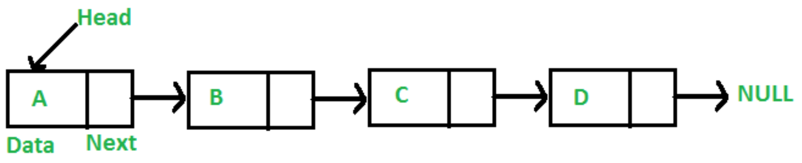
Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Linked List

Like arrays, 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.



Why Linked List?
Arrays can be used to store linear data of similar types, but arrays have the following limitations.
1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.
For example, in a system, if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion

Drawbacks:
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
2) Extra memory space for a pointer is required with each element of the list.
3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:
A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.
Each node in a list consists of at least two parts:
1) data
2) Pointer (Or Reference) to the next node
In C, we can represent a node using structures. Below is an example of a linked list node with integer data.
In Java or C#, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

Non-Liner Data Structures

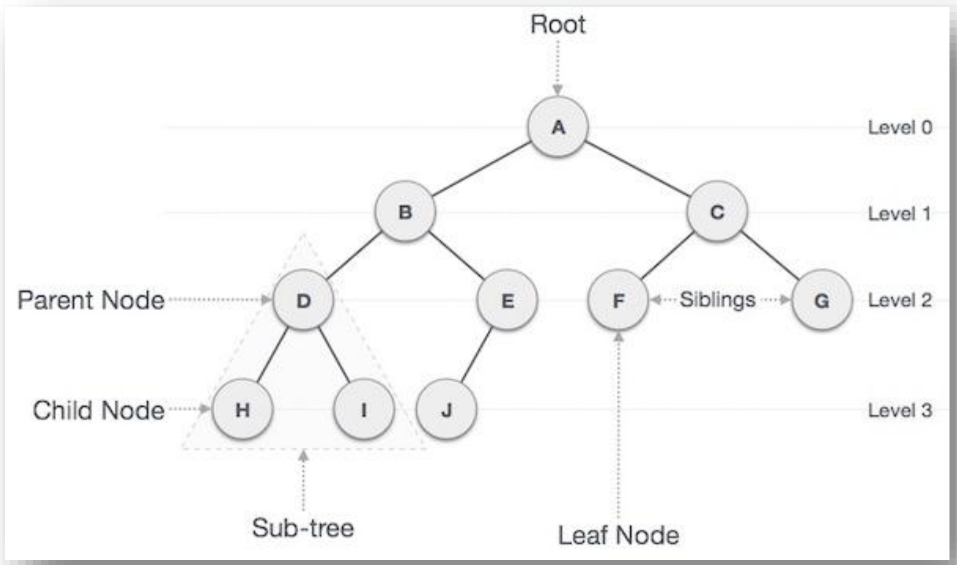
These are the data structures in which there is no sequential linking of data elements. Any pair or group of data elements can be linked to each other and can be accessed without a strict sequence.

- **Binary Tree:** It is a data structure where each data element can be connected to maximum two other data elements and it starts with a root node.
- **Heap:** It is a special case of Tree data structure where the data in the parent node is either strictly greater than/ equal to the child nodes or strictly less than it’s child nodes.
- **Hash Table:** It is a data structure which is made of arrays associated with each other using a hash function. It retrieves values using keys rather than index from a data element.
- **Graph: .**It is an arrangement of vertices and nodes where some of the nodes are connected to each other through links.

Tree

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Python Specific Data Structures

These data structures are specific to python language and they give greater flexibility in storing different types of data and faster processing in python environment.

- **List:** It is similar to array with the exception that the data elements can be of different data types. You can have both numeric and string data in a python list.

- **Tuple:** Tuples are similar to lists but they are immutable which means the values in a tuple cannot be modified they can only be read.
- **Dictionary:** The dictionary contains Key-value pairs as its data elements.

In the next chapters we are going to learn the details of how each of these data structures can be implemented using Python.

Operations on Data Structures

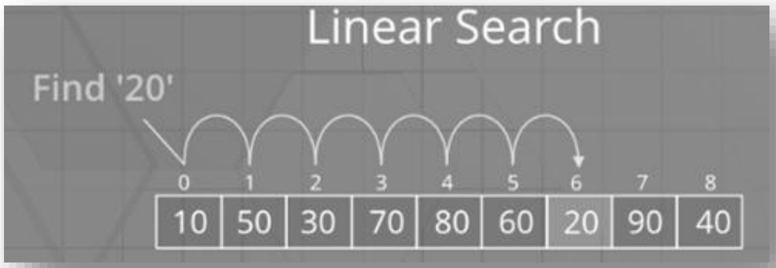
- **Insertion:** adding new data elements in a Data Structure
- **Deletion:** removal of data elements from Data Structure
- **Searching:** finding specific data elements in the Data Structure
- **Traversal:** processing all the data elements of Data Structure one by one
- **Sorting:** Arranging all the data elements of Data Structure in specific order
- **Merging**: combining elements of two similar Data Structure to form a new data structure of same type.

Linear Search Operation

Let's see a basic linear search operation on Python list and tuples.

A simple approach is to do **linear search**, i.e

- Start from the leftmost element of list and one by one compare x with each element of the list.
- If x matches with an element, return True.
- If x doesn't match with any of elements, return False.



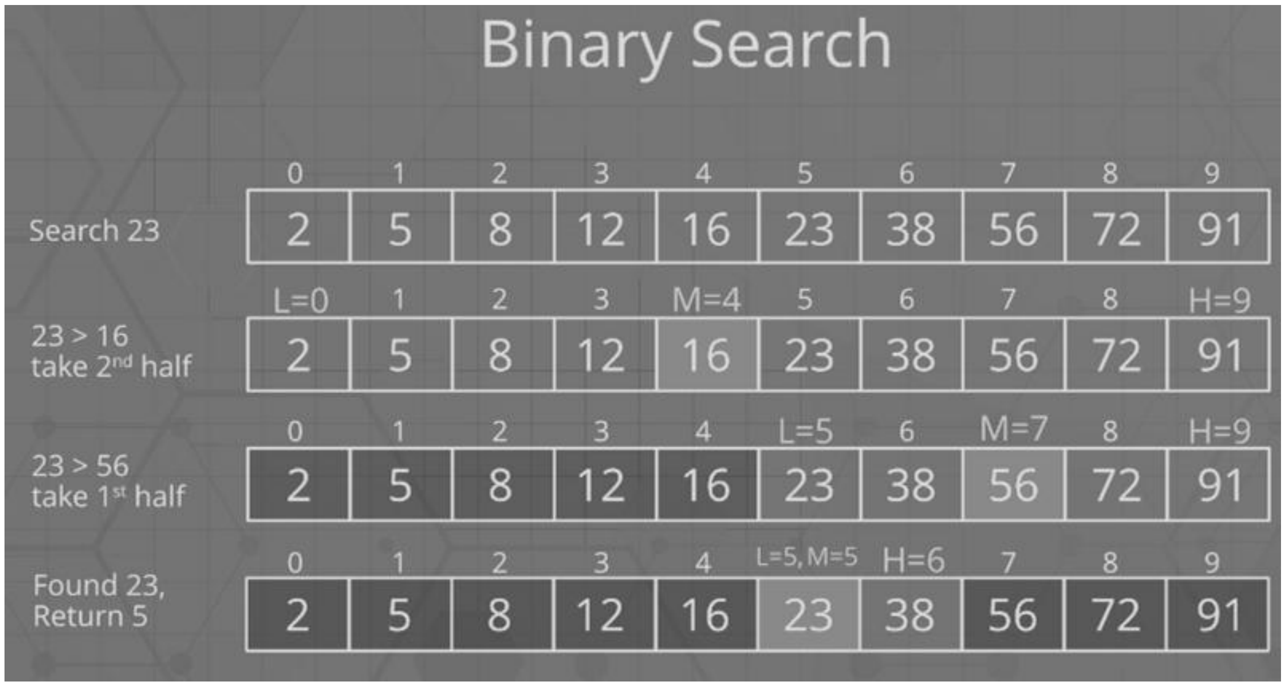
```
# Search function with parameter list name and the value to be searched
def search(list,n):
   for i in range(len(list)):
      if list[i] == n:
         return True
   return False
# list which contains numbers.
list = [33,55,687,89,45,89,22,345]
n = 45
if search(list, n):
         print("Found")
else:
         print("Not Found")
```

Binary Search Operation

Binary Search: Binary Search is a search technique to perform search operation on a sorted array or list by repeatedly dividing the search interval in half. Begin with an interval covering the whole array/list. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

In a nutshell, this search algorithm takes advantage of a collection of elements that is already sorted by ignoring half of the elements after just one comparison.

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else if x is greater than the mid element, then x can only lie in the right (greater) half subarray after the mid element. Then we apply the algorithm again for the right half.
4. Else if x is smaller, the target x must lie in the left (lower) half. So we apply the algorithm for the left half.



Example

```
 # Python Program for recursive binary search. # Returns index of x in arr if present, else -1
def binarySearch (arr, l, r, x):
        # Check base case
        if r >= l:
                mid = l + (r - l) // 2
                # If element is present at the middle itself
                if arr[mid] == x:
                        return mid
                # If element is smaller than mid, then it      # can only be present in left subarray
                elif arr[mid] > x:
                        return binarySearch(arr, l, mid-1, x)
                # Else the element can only be present        # in right subarray
                else:
                        return binarySearch(arr, mid + 1, r, x)
        else:
                # Element is not present in the array
                return -1

# Driver Code
arr = [ 2, 3, 4, 10, 40 ]
x = 10
# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
        print ("Element is present at index % d" % result)
else:
        print ("Element is not present in array")
```

```
# Python code to implement iterative Binary  Search.
# It returns location of x in given array arr # if present, else returns -1
def binarySearch(arr, l, r, x):
        while l <= r:
                mid = l + (r - l) // 2;
                # Check if x is present at mid
                if arr[mid] == x:
                        return mid
                # If x is greater, ignore left half
                elif arr[mid] < x:
                        l = mid + 1
                # If x is smaller, ignore right half
                else:                   r = mid - 1

        # If we reach here, then the element
        # was not present
        return -1

# Driver Code
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
        print ("Element is present at index % d" % result)
else:
        print ("Element is not present in array")
```

Insertion in Linear List

```
# Python program to insert value in linear list using insert() method

list1 = [ 1, 2, 3, 4, 5, 6, 7 ]
# insert 10 at 4th index
list1.insert(4, 10)
print(list1)
list2 = ['a', 'b', 'c', 'd', 'e']
# insert z at the front of the list
list2.insert(0, 'z')
print(list2)
```

Insertion in Sorted List using Bisect Algorithm Function

The purpose of Bisect algorithm is to find a position in list where an element needs to be inserted to keep the list sorted.

Python in its definition provides the bisect algorithms using the module "**bisect**" which allows to keep the list in sorted order after insertion of each element. This is essential as this reduces overhead time required to sort the list again and again after insertion of each element.

Important Bisection Functions

1. bisect(list, num, beg, end) :- This function returns the **position** in the **sorted** list, where the number passed in argument can be placed so as to **maintain the resultant list in sorted order**. If the element is already present in the list, the **right most position** where element has to be inserted is returned. **This function takes 4 arguments, list which has to be worked with, number to insert, starting position in list to consider, ending position which has to be considered**.

2. bisect_left(list, num, beg, end) :- This function returns the **position** in the **sorted** list, where the number passed in argument can be placed so as to **maintain the resultant list in sorted order**. If the element is already present in the list, the **left most position** where element has to be inserted is returned. **This function takes 4 arguments, list which has to be worked with, number to insert, starting position in list to consider, ending position which has to be considered**.

```
# Python code to demonstrate the working of  bisect(), bisect_left() and bisect_right()
# importing "bisect" for bisection operations
import bisect
# initializing list
li = [1, 3, 4, 4, 4, 6, 7]

# using bisect() to find index to insert new element  returns 5 ( right most possible index )
print ("The rightmost index to insert, so list remains sorted is : ", end="")
print (bisect.bisect(li, 4))

# using bisect_left() to find index to insert new element  returns 2 ( left most possible index )
print ("The leftmost index to insert, so list remains sorted is : ", end="")
print (bisect.bisect_left(li, 4))
# using bisect_right() to find index to insert new element
# returns 4 ( right most possible index )
print ("The rightmost index to insert, so list remains sorted is : ", end="")
print (bisect.bisect_right(li, 4, 0, 4))
```

Deleting Element from List

Deleting elements from list using del() and pop() functions

```
# initializing list

lis = [2, 1, 3, 5, 4, 3, 8]
```

```
  # using del to delete elements from pos. 2 to 5
# deletes 3,5,4
del lis[2 : 5]
  # displaying list after deleting
print ("List elements after deleting are : ",end="")
for i in range(0, len(lis)):
    print(lis[i], end=" ")
 print("\r")
# using pop() to delete element at pos 2
# deletes 3
lis.pop(2)
# displaying list after popping
print ("List elements after popping are : ", end="")
for i in range(0, len(lis)):
    print(lis[i], end=" ")
```

remove() function is used to **delete the first occurrence** of number mentioned in its arguments.

```
# Python code to demonstrate the working of
# insert() and remove()

# initializing list
lis = [2, 1, 3, 5, 3, 8]

# using insert() to insert 4 at 3rd pos
lis.insert(3, 4)

# displaying list after inserting
print("List elements after inserting 4 are : ", end="")
for i in range(0, len(lis)):
    print(lis[i], end=" ")

print("\r")

# using remove() to remove first occurrence of 3
# removes 3 at pos 2
lis.remove(3)

# displaying list after removing
print ("List elements after removing are : ", end="")
for i in range(0, len(lis)):
    print(lis[i], end=" ")
```

****