Data Structure

Data Structure Overview

Data structures are fundamental concepts of computer science which helps is 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.

In this chapter we are going to study a short overview of some frequently used data structures in general and how they are related to some specific python data types. There are also some data structures specific to python which is listed as another category.

General Data Structures

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.
* **Linked List** − Each data element contains a link to another element along with the data present in it.
* **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).
* **Matrix** − It is two dimensional data structure in which the data element is referred by a pair of indices.

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.

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.

# Python - Arrays

[Previous](https://www.tutorialspoint.com/python_data_structure/python_data_structure_environment.htm)

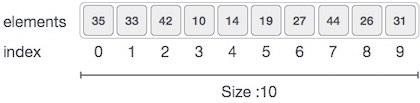
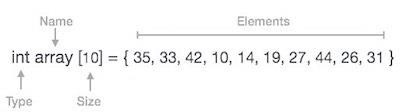
[Next](https://www.tutorialspoint.com/python_data_structure/python_lists_data_structure.htm)

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array are as follows −

* **Element** − Each item stored in an array is called an element.
* **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

## Array Representation

Arrays can be declared in various ways in different languages. Below is an illustration.



As per the above illustration, following are the important points to be considered −

* Index starts with 0.
* Array length is 10, which means it can store 10 elements.
* Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

## Basic Operations

The basic operations supported by an array are as stated below −

* **Traverse** − print all the array elements one by one.
* **Insertion** − Adds an element at the given index.
* **Deletion** − Deletes an element at the given index.
* **Search** − Searches an element using the given index or by the value.
* **Update** − Updates an element at the given index.

ypecode are the codes that are used to define the type of value the array will hold. Some common typecodes used are as follows −

|  |  |
| --- | --- |
| **Typecode** | **Value** |
| b | Represents signed integer of size 1 byte |
| B | Represents unsigned integer of size 1 byte |
| c | Represents character of size 1 byte |
| i | Represents signed integer of size 2 bytes |
| I | Represents unsigned integer of size 2 bytes |
| f | Represents floating point of size 4 bytes |
| d | Represents floating point of size 8 bytes |

Before looking at various array operations lets create and print an array using python.

#Example1 Create Array

from array import\*

array1=array('i',[10,20,30,40,50])

for i in array1:

    print(i)

#Example 2 Accessing Array element

print("\nAccessing arrya Items:",array1[0])

print(array1[1])

print(array1[2])

print(array1[3])

print(array1[4])

#Example 3 Insetion

"""

Insert operation is to insert one or more data elements into an array.

Based on the requirement,

a new element can be added at the beginning, end, or any given index of array.

    """

print("\n Insert 60 into specifice index(1):")

array1.insert(1,60)

for i in array1:

    print(i)

#Example 4 Deletion

"""

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

    """

print("\n Item 60 remove from array: ")

array1.remove(60)

for i in array1:

    print(i)

#Example 5 Search Operation

"""

You can perform a search for an array element based on its value or its index.

Here, we search a data element using the python in-built index() method.

"""

print("\nIndex of 20 is:",array1.index(20))

#Example 6 Update Operation

"""

Update operation refers to updating an existing element from the array at a given index.

Here, we simply reassign a new value to the desired index we want to update."""

array1[4]=45

print("Update last item in array:")

for i in array1:

    print(i)

# Python - Lists

[Previous](https://www.tutorialspoint.com/python_data_structure/python_arrays.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_tuples_data_structure.htm)

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets.

### For example

list1 = ['physics', 'chemistry', 1997, 2000]

list2 = [1, 2, 3, 4, 5 ]

list3 = ["a", "b", "c", "d"]

#Example 1 Accessing Values

"""

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index.

"""

list1=[1,2,3,4,5]

list2=["A","B","C",5,6,7]

print('List1 first item:',list1[0])

print('List2 first item:',list2[0])

#Example 2 Updating Lists

"""

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method."""

print("\nAfter update list1:",list1)

list1[4]=55

print("Update last item:",list1)

#Example 3 Delete List Elements

"""

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know."""

del list1[4]

print("\nAfter delete last1 item:",list1)

#Example 4 Basic List Operations

"""

Lists respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string."""

print("\nLength of list1:",len(list1))

print("\nConcatenation of two lists:",list1+list2)

print("\nRepetition of string:",("Shary")\*3)

# Python - Tuples

[Previous](https://www.tutorialspoint.com/python_data_structure/python_lists_data_structure.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_dictionary_data_structure.htm)

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also.

### For example

tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5 );

tup3 = "a", "b", "c", "d";

#Example1 Empty tuple

tuple1=()

tuple1=(2,)

print(tuple1)

#Example2 Accessing Tuple items

tuple2=(1,2,3,4,4,5,"Shary","Khan","Hamad")

print(tuple2[2])

#Example3 Updating Tuple

"""Tuples are immutable which means you cannot update or change the values of tuple elements."""

# tuple2[2]=100 Show error means not changing

tuple3=(10,20,30)

t3=tuple2+tuple3# by this technique can update

print(t3)

#Example 4 delete tuple item

del t3

# print(t3)

# Python - Dictionary

[Previous](https://www.tutorialspoint.com/python_data_structure/python_tuples_data_structure.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_2darray.htm)

In Dictionary each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this − {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

#Example dict

d1={"Name":"Shary","Age":29,'Class':'BSSE'}

#Example accessing dict item

print("d1['Name'] :",d1['Name'])

#Example3 Updating

d1['Section']='CA'

print(d1)

#Example delete dect element

del d1['Section']

print(d1)

# Python - 2-D Array

[Previous](https://www.tutorialspoint.com/python_data_structure/python_dictionary_data_structure.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_matrix.htm)

Two dimensional array is an array within an array. It is an array of arrays. In this type of array the position of an data element is referred by two indices instead of one. So it represents a table with rows an dcolumns of data.

In the below example of a two dimensional array, observer that each array element itself is also an array.

#Example 1 create 2d array

array=[[1,2,3,4],[5,6,7,8],[9,10,11,12]]

print(array)

#Example2 Accessing element

print(array[0])#[1,2,3,4]

print(array[1][1])#6

#Example3 all element access of the array

for i in array:

    for j in i:

        print(j,end=" ")

    print()

#Example4 Insert the another row

array.insert(1,[1.1,2.2,3.3,4.4])

for i in array:

    for j in i:

        print(j,end=" ")

    print()

#Example5 Update

array[1]=[1.1,1.2,1.3,1.4]

for i in array:

    for j in i:

        print(j,end=" ")

    print()

#Example6 Delete items

del array[1]

for i in array:

    for j in i:

        print(j,end=" ")

    print()

# Python - Matrix

[Previous](https://www.tutorialspoint.com/python_data_structure/python_2darray.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_sets.htm)

Matrix is a special case of two dimensional array where each data element is of strictly same size. So every matrix is also a two dimensional array but not vice versa.

Matrices are very important data structures for many mathematical and scientific calculations. As we have already discussed two dimnsional array data structure in the previous chapter we will be focusing on data structure operations specific to matrices in this chapter.

#EXample 1 Creating metrix

import numpy as np

a=np.array([['Mon',5,10,15,20],['Tue',5,10,15,20],['Wed',5,10,15,20]])

print(a)

a1=np.reshape(a,(3,5))

#Example2 Accessing the items

print(a[0])

print(a[0][1])

#Example 3 adding row

a2=np.append(a,[["Tu",5,10,15,20]],0)

print(a2)

#Example4 adding a column

a3=np.insert(a2,[5],[[25],[25],[25],[25]],1)

print(a3)

#Example 5 delete a row

"""We can delete a row from a matrix using the delete() method. We have to specify the index of the row and also the axis value which is 0 for a row and 1 for a column."""

d=np.delete(a3,[3],0)

print('\n',d)

#Example 6 delete a column

a4=np.delete(a3,[5],1)

print(a4)

#Example 7 Update a row

"""

To update the values in the row of a matrix we simply re-assign the values at the index of the row. In the below example all the values for thrusday's data is marked as zero. The index for this row is 3."""

a3[3]=['TUe',0,0,0,0,0]

print(a3)

# Python - Sets

[Previous](https://www.tutorialspoint.com/python_data_structure/python_matrix.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_maps.htm)

Mathematically a set is a collection of items not in any particular order. A Python set is similar to this mathematical definition with below additional conditions.

* The elements in the set cannot be duplicates.
* The elements in the set are immutable(cannot be modified) but the set as a whole is mutable.
* There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.

## Set Operations

The sets in python are typically used for mathematical operations like union, intersection, difference and complement etc. We can create a set, access it’s elements and carry out these mathematical operations as shown below.

#Example 1 creating set

set1=set([1,2,3,3,"Shary","Khan"])

set2={1,2,3,'umair','hamad'}

print(set1)

print(set2)

#Example 2 Accessing Values in a Set

"""

We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set."""

#print(set1[2]) not working index

for i in set1:

    print(i)

#Example 3 Adding Items to a Set

"""

We can add elements to a set by using add() method. Again as discussed there is no specific index attached to the newly added element."""

set1.add('Ali')

print(set1)

#Example4 Removing Item from a Set

"""

We can remove elements from a set by using discard() method. Again as discussed there is no specific index attached to the newly added element"""

set1.discard('Ali')

print(set1)

#Example 5 Union of Sets

"""

The union operation on two sets produces a new set containing all the distinct elements from both the sets. """

newSet=set1|set2

print(newSet)

#Example 6 Intersection of Sets

"""

The intersection operation on two sets produces a new set containing only the common elements from both the sets."""

result=set1&set2

print(result)

#Example 7 Difference of Sets

"""

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set."""

result2=set1-set2

print(result2)

#Example Compare Sets

"""

We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

"""

set3={1,2,'Khan'}

supset=set1>=set3

print(supset)

# Python - Maps

[Previous](https://www.tutorialspoint.com/python_data_structure/python_sets.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_linked_lists.htm)

Python Maps also called ChainMap is a type of data structure to manage multiple dictionaries together as one unit. The combined dictionary contains the key and value pairs in a specific sequence eliminating any duplicate keys. The best use of ChainMap is to search through multiple dictionaries at a time and get the proper key-value pair mapping. We also see that these ChainMaps behave as stack data structure.

## Creating a ChainMap

We create two dictionaries and club them using the ChainMap method from the collections library. Then we print the keys and values of the result of the combination of the dictionaries. If there are duplicate keys, then only the value from the first key is preserved.

import collections

d1={'Day1':'Mon','Day2':'Tuesday','Day3':'Wed'}

d2={'Day4':'Thrusday','Day1':'Friday'}

res=collections.ChainMap(d1,d2)

#Creating a single dict

print(res.maps,'\n')

print("keys={}".format(list(res.keys())))

print("Values={}".format(list(res.values())))

#Print all elements from the results

print("Elements: ")

for key, value in res.items():

    print('{}={}'.format(key,value))

#find a specific value in the results

print("Day1 in res: {}".format('Day1' in res))

#Example 2 Map Reordering

"""

If we change the order the dictionaries while clubbing them in the above example we see that the position of the elements get interchanged as if they are in a continuous chain. This again shows the behavior of Maps as stacks."""

res1=collections.ChainMap(d1,d2)

print(res1.maps,'\n')

res2=collections.ChainMap(d2,d1)

print(res2.maps,'\n')

#Example 3 Updating Map

"""

When the element of the dictionary is updated, the result is instantly updated in the result of the ChainMap. In the below example we see that the new updated value reflects in the result without explicitly applying the ChainMap method again."""

d2['Day4']='Sunday'

print(res2.maps,'\n')

# Python - Stack

[Previous](https://www.tutorialspoint.com/python_data_structure/python_linked_lists.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_queue.htm)

In the english dictionary the word stack means arranging objects on over another. It is the same way memory is allocated in this data structure. It stores the data elements in a similar fashion as a bunch of plates are stored one above another in the kitchen. So stack data strcuture allows operations at one end wich can be called top of the stack.We can add elements or remove elements only form this en dof the stack.

In a stack the element insreted last in sequence will come out first as we can remove only from the top of the stack. Such feature is known as Last in First Out(LIFO) feature. The operations of adding and removing the elements is known as **PUSH** and **POP**. In the following program we implement it as **add** and and **remove** functions. We declare an empty list and use the append() and pop() methods to add and remove the data elements.

#Example 1  PUSH into a Stack

class Stack:

    def \_\_init\_\_(self):

        self.stack=[]

    def add(self,dataval):

        #use list append method to add element

        if dataval not in self.stack:

            self.stack.append(dataval)

            return True

        else:

            return False

    #Use peek to lock at the top

    def peek(self):

        return self.stack[-1]

obj=Stack()

obj.add("Monday")

obj.add("Tuesday")

#obj.peek() or

print(obj.peek())

obj.add("Wed")

obj.add("Friday")

print(obj.peek())

#Example POP from a Stack

"""

As we know we can remove only the top most data element from the stack, we implement a python program which does that. The remove function in the following program returns the top most element. we check the top element by calculating the size of the stack first and then use the in-built pop() method to find out the top most element."""

class Stack:

    def \_\_init\_\_(self):

        self.stack = []  # initialize an empty list

    def add(self, dataval):

        # use list append method to add element

        if dataval not in self.stack:  # check if the element is already present in the stack

            self.stack.append(dataval)  # add the element to the stack

            return True

        else:

            return False

    def peek(self):

        return self.stack[-1]  # return the top element of the stack

    def remove(self):

        if len(self.stack) <= 0:  # check if the stack is empty

            return "No element in the Stack"

        else:

            return self.stack.pop()  # remove and return the top element of the stack

    def printitem(self):

        if len(self.stack) == 0:  # check if the stack is empty

            return "Stack is empty"

        s = list()

        for i in self.stack:

            s.append(i)

        return s  # return the elements of the stack as a list

obj=Stack()

obj.add(1)

obj.add(2)

print("Top of Stack:",obj.peek())

obj.add(3)

obj.add(4)

obj.add(5)

print("Top of Stack:",obj.peek())

print("Orignal Stack:",obj.printitem())

print("Poped Item:",obj.remove())

print("Poped item:",obj.remove())

print("Stack after Poped items:"+str(obj.printitem()))

# Python - Queue

[Previous](https://www.tutorialspoint.com/python_data_structure/python_stack.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_dequeue.htm)

We are familiar with queue in our day to day life as we wait for a service. The queue data structure aslo means the same where the data elements are arranged in a queue. The uniqueness of queue lies in the way items are added and removed. The items are allowed at on end but removed form the other end. So it is a First-in-First out method.

A queue can be implemented using python list where we can use the insert() and pop() methods to add and remove elements. Their is no insertion as data elements are always added at the end of the queue.

## Adding Elements

In the below example we create a queue class where we implement the First-in-First-Out method. We use the in-built insert method for adding data elements.

#Example 1 Creating Queue

class Queue:

    def \_\_init\_\_(self):

        self.queue = []  # initialize an empty list

    def addelement(self, dataval):

        # use list insert method to add element

        if dataval not in self.queue:  # check if the element is already present in the queue

            self.queue.insert(0, dataval)  # add the element to the queue

            return True

        return False

    def size(self):

        return len(self.queue)  # return the number of elements in the queue

    def printqueue(self):

        result = list()

        if self.queue == 0:  # check if the queue is empty

            return "Queue is Empty"

        else:

            for i in self.queue:

                result.append(i)

            return result  # return the elements of the queue as a list

    def remove(self):

        if self.queue==0:

            return ("Queue is empty ")

        else:

            self.queue.pop()

#Create object of queue

obj=Queue()

obj.addelement(1)

obj.addelement(2)

obj.addelement(3)

obj.addelement(4)

print("Size of Queue:",obj.size())

print("Orignal Queue: ",obj.printqueue())

obj.remove()

obj.remove()

print("After Poped element from Queue: ",obj.printqueue())

# Python - Dequeue

[Previous](https://www.tutorialspoint.com/python_data_structure/python_queue.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_advanced_linked_list.htm)

A double-ended queue, or deque, supports adding and removing elements from either end. The more commonly used stacks and queues are degenerate forms of deques, where the inputs and outputs are restricted to a single end.

import collections

doubleEnded=collections.deque([4,5,6,7])

doubleEnded.append(8)

print("8 Appended at Right :",doubleEnded)

doubleEnded.appendleft(3)

print("3 is Appeded at left:",doubleEnded)

doubleEnded.pop()

print("Item Deleted from right: ",doubleEnded)

doubleEnded.popleft()

print("Item deleting from left: ",doubleEnded)

# Python - Linked Lists

[Previous](https://www.tutorialspoint.com/python_data_structure/python_maps.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_stack.htm)

A linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in form of a pointer. Python does not have linked lists in its standard library. We implement the concept of linked lists using the concept of nodes as discussed in the previous chapter.

We have already seen how we create a node class and how to traverse the elements of a node.In this chapter we are going to study the types of linked lists known as singly linked lists. In this type of data structure there is only one link between any two data elements. We create such a list and create additional methods to insert, update and remove elements from the list.

## Creation of Linked list

A linked list is created by using the node class we studied in the last chapter. We create a Node object and create another class to use this ode object. We pass the appropriate values through the node object to point the to the next data elements. The below program creates the linked list with three data elements. In the next section we will see how to traverse the linked list

# Python - Advanced Linked list

[Previous](https://www.tutorialspoint.com/python_data_structure/python_dequeue.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_hash_table.htm)

We have already seen Linked List in earlier chapter in which it is possible only to travel forward. In this chapter we see another type of linked list in which it is possible to travel both forward and backward. Such a linked list is called Doubly Linked List. Following is the features of doubly linked list.

* Doubly Linked List contains a link element called first and last.
* Each link carries a data field(s) and two link fields called next and prev.
* Each link is linked with its next link using its next link.
* Each link is linked with its previous link using its previous link.
* The last link carries a link as null to mark the end of the list.

## Creating Doubly linked list

We create a Doubly Linked list by using the Node class. Now we use the same approach as used in the Singly Linked List but the head and next pointers will be used for proper assignation to create two links in each of the nodes in addition to the data present in the node.

# Python - Hash Table

[Previous](https://www.tutorialspoint.com/python_data_structure/python_advanced_linked_list.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_binary_tree.htm)

Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function. That makes accessing the data faster as the index value behaves as a key for the data value. In other words Hash table stores key-value pairs but the key is generated through a hashing function.

So the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.

In Python, the Dictionary data types represent the implementation of hash tables. The Keys in the dictionary satisfy the following requirements.

* The keys of the dictionary are hashable i.e. the are generated by hashing function which generates unique result for each unique value supplied to the hash function.
* The order of data elements in a dictionary is not fixed.

So we see the implementation of hash table by using the dictionary data types as below.

#Example 1 Accessing Values in Dictionary

"""

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value"""

#decalre dict

d1={"name":"Shary","Age":26,"Class":"BSSE"}

#Access dict with keys

print("d1[Name]:",d1["name"])

print("d1[Age]:",d1["Age"])

print("D1:\n",d1)

#Example 2 Updating Dictionary

"""

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example −"""

#change the age

d1["Age"]=20

#Add new value with key

d1["Marks"]=400

print("d1[Marks]:",d1["Marks"])

print("d1[Age]:",d1["Age"])

print("\nAfter Update:",d1)

#Example 3 Delete Dictionary Elements

"""

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.To explicitly remove an entire dictionary, just use the del statement."""

#Remove entry key with value

del d1["Class"]

print("\nAfter delete item:",d1)

#Remove all entries

d1.clear()

print("\n After clear all elements: ",d1)

#Delete entire dict

del d1

# Python - Binary Tree

[Previous](https://www.tutorialspoint.com/python_data_structure/python_hash_table.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_binary_search_tree.htm)

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.

We create a tree data structure in python by using the concept os node discussed earlier. We designate one node as root node and then add more nodes as child nodes. Below is program to create the root node.

Note:

1. In linear data structure like stack ,queue,array etc. have only one traversing each node
2. Where the tree has various way to traversing each node
3. On the basis of root node the tree traversing can be classified three way
4. Pre-order: Parent-Left-right
5. In-Order: L-P-R
6. Post-Order: L-R-P

#EXample 1 Create Root

"""

We just create a Node class and add assign a value to the node. This becomes tree with only a root node"""

class Node:

    def \_\_init\_\_(self,data):

        self.data=data

        self.left=None

        self.right=None

    def printTree(self):

        return self.data

#Create obj

obj=Node(10)

print("Root Node: ",obj.printTree())

#EXample 2 Inserting into a Tree

"""

To insert into a tree we use the same node class created above and add a insert class to it. The insert class compares the value of the node to the parent node and decides to add it as a left node or a right node. Finally the PrintTree class is used to print the tree."""

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Insert Into Tree \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

class Node:

    def \_\_init\_\_(self,data):

        self.data=data

        self.left=None

        self.right=None

    #Insert into Tree

    def insert(self,newdata):

        if self.data:

            if newdata<self.data:

                if self.left is None:

                    self.left=Node(newdata)

                else:

                    self.left.insert(newdata)

            elif newdata>self.data:

                if self.right is None:

                    self.right=Node(newdata)

                else:

                    self.right.insert(newdata)

        else:

            self.data=newdata

    #Print the tree

    def printTree(self):

         if self.left:

             self.left.printTree()

         print(self.data)

         if self.right:

            self.right.printTree()

#Use the insert method to add nodes

obj=Node(12)

obj.insert(6)

obj.insert(14)

obj.insert(3)

obj.insert(11)

obj.insert(13)

obj.printTree()

## Traversing a Tree

The tree can be traversed by deciding on a sequence to visit each node. As we can clearly see we can start at a node then visit the left sub-tree first and right sub-tree next. Or we can also visit the right sub-tree first and left sub-tree next. Accordingly there are different names for these tree traversal methods.

## Tree Traversal Algorithms

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree.

* In-order Traversal
* Pre-order Traversal
* Post-order Traversal

### In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then, we create an insert function to add data to the tree. Finally, the In-order traversal logic is implemented by creating an empty list and adding the left node first followed by the root or parent node.

At last the left node is added to complete the In-order traversal. Please note that this process is repeated for each sub-tree until all the nodes are traversed.

### Example

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* In-Order Traversing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

# Define a class Node

class Node:

    def \_\_init\_\_(self, data):

        self.data = data  # Initialize the data attribute of the node with the value of data

        self.left = None  # Initialize the left child of the node as None

        self.right = None  # Initialize the right child of the node as None

    # Insert into Tree

    def insert(self, newdata):

        if self.data:

            if newdata < self.data:

                if self.left is None:

                    self.left = Node(newdata)  # Create a new node with newdata and set it as the left child of the current node

                else:

                    self.left.insert(newdata)  # Recursively call insert method on the left child of the current node

            elif newdata > self.data:

                if self.right is None:

                    self.right = Node(newdata)  # Create a new node with newdata and set it as the right child of the current node

                else:

                    self.right.insert(newdata)  # Recursively call insert method on the right child of the current node

        else:

            self.data = newdata  # If the current node is empty, set its data to newdata

    # Print the tree

    def printTree(self):

        if self.left:

            self.left.printTree()  # Recursively call printTree method on the left child of the current node

        print(self.data)  # Print the data of the current node

        if self.right:

            self.right.printTree()  # Recursively call printTree method on the right child of the current node

    # In order traversal

    # Left -> Parent -> Right

    def In\_Order(self, root):

        res = []

        if root:

            res = self.In\_Order(root.left)  # Recursively call In\_Order method on the left child of the current node

            res.append(root.data)  # Append the data of the current node to the result list

            res = res + self.In\_Order(root.right)  # Recursively call In\_Order method on the right child of the current node

        return res

# Use the insert method to add nodes

obj = Node(27)

obj.insert(14)

obj.insert(35)

obj.insert(10)

obj.insert(19)

obj.insert(31)

obj.insert(42)

obj.printTree()  # Print the tree

print(obj.In\_Order(obj))  # Print the tree in order

"""

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* In-Order Traversing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

10

14

19

27

31

35

42

[10, 14, 19, 27, 31, 35, 42]

"""

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then, we create an insert function to add data to the tree. Finally, the Pre-order traversal logic is implemented by creating an empty list and adding the root node first followed by the left node.

At last, the right node is added to complete the Pre-order traversal. Please note that, this process is repeated for each sub-tree until all the nodes are traversed.

print('\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Post-Order Traversing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

# Define a class Node

class Node:

    def \_\_init\_\_(self, data):

        self.data = data  # Initialize the data attribute of the node with the value of data

        self.left = None  # Initialize the left child of the node as None

        self.right = None  # Initialize the right child of the node as None

    # Insert into Tree

    def insert(self, newdata):

        if self.data:

            if newdata < self.data:

                if self.left is None:

                    self.left = Node(newdata)  # Create a new node with newdata and set it as the left child of the current node

                else:

                    self.left.insert(newdata)  # Recursively call insert method on the left child of the current node

            elif newdata > self.data:

                if self.right is None:

                    self.right = Node(newdata)  # Create a new node with newdata and set it as the right child of the current node

                else:

                    self.right.insert(newdata)  # Recursively call insert method on the right child of the current node

        else:

            self.data = newdata  # If the current node is empty, set its data to newdata

    # Print the tree

    def printTree(self):

        if self.left:

            self.left.printTree()  # Recursively call printTree method on the left child of the current node

        print(self.data)  # Print the data of the current node

        if self.right:

            self.right.printTree()  # Recursively call printTree method on the right child of the current node

    def Post\_Order(self,root):

        #L--R-P

        result=[]

        if root:

            result=self.Post\_Order(root.left)

            result=result + self.Post\_Order(root.right)

            result.append(root.data)

        return result

obj=Node(27)

obj.insert(14)

obj.insert(35)

obj.insert(10)

obj.insert(19)

obj.insert(31)

obj.insert(42)

#called printTree()

obj.printTree()

#called pre\_Order()

print(obj.Post\_Order(obj))

"""

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Post-Order Traversing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

10

14

19

27

31

35

42

[10, 19, 14, 31, 42, 35, 27]"""

# Python -Binary Search Tree

[Previous](https://www.tutorialspoint.com/python_data_structure/python_binary_tree.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_heaps.htm)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties.The left sub-tree of a node has a key less than or equal to its parent node's key.The right sub-tree of a node has a key greater than to its parent node's key.Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree

left\_subtree (keys) ≤ node (key) ≤ right\_subtree (keys)

### Search for a value in a B-tree

Searching for a value in a tree involves comparing the incoming value with the value exiting nodes. Here also we traverse the nodes from left to right and then finally with the parent. If the searched for value does not match any of the exiting value, then we return not found message, or else the found message is returned.

class Node:

    def \_\_init\_\_(self, data):

        # Initialize a node with data and set left and right child nodes to None

        self.data = data

        self.left = None

        self.right = None

    def inser(self, newdata):

        # Insert new data into the binary search tree

        if self.data:

            # If the current node has data

            if newdata < self.data:

                # If the new data is less than the current node's data

                if self.left is None:

                    # If there's no left child node, create a new one with the new data

                    self.left = Node(newdata)

                else:

                    # Otherwise, recursively call insert on the left child node

                    self.left.inser(newdata)

            elif newdata > self.data:

                # If the new data is greater than the current node's data

                if self.right is None:

                    # If there's no right child node, create a new one with the new data

                    self.right = Node(newdata)

                else:

                    # Otherwise, recursively call insert on the right child node

                    self.right.inser(newdata)

        else:

            # If the current node has no data, set the data to the new data

            self.data = newdata

    def Find\_Value(self, key, index=0):

        # Search for a value (key) in the binary search tree

        if key < self.data:

            # If the key is less than the current node's data

            if self.left is None:

                # If there's no left child node, the key is not found

                return str(key) + " Is Not Found", None

            else:

                # Recursively search for the key in the left subtree

                return self.left.Find\_Value(key, index)

        elif key > self.data:

            # If the key is greater than the current node's data

            if self.right is None:

                # If there's no right child node, the key is not found

                return str(key) + " Is Not Found", None

            else:

                # Recursively search for the key in the right subtree

                return self.right.Find\_Value(key, index)

        else:

            # If the key matches the current node's data, return the data and index

            return self.data, index

    def printTree(self):

        # Print the binary search tree in sorted order

        if self.left:

            # If there's a left child node, recursively print its subtree

            self.left.printTree()

        print(self.data)  # Print the current node's data

        if self.right:

            # If there's a right child node, recursively print its subtree

            self.right.printTree()

obj = Node(12)

obj.inser(6)

obj.inser(14)

obj.inser(5)

obj.inser(3)

obj.inser(13)

# Print the binary search tree

obj.printTree()

# Search for a value in the binary search tree

n = int(input("Enter number: "))

# Call the Find\_Value method to search for the value

find, index = obj.Find\_Value(n)

# Check if the result is a string (indicating the value was not found)

if isinstance(find, str):

    print(find)  # Print a message indicating the value was not found

else:

    # Print a message indicating the value was found along with its index

    print(f"The value {find} is found at index {index}")

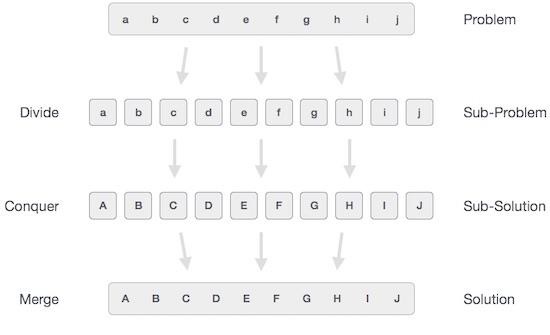
## Python - Divide and Conquer

## (Binary Search implementation)

[Previous](https://www.tutorialspoint.com/python_data_structure/python_algorithm_design.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_recursion.htm)

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



Broadly, we can understand **divide-and-conquer** approach in a three-step process.

## Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

## Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

## Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer &s; merge steps works so close that they appear as one.

### Examples

The following program is an example of **divide-and-conquer** programming approach where the binary search is implemented using python.

## Binary Search implementation

In binary search we take a sorted list of elements and start looking for an element at the middle of the list. If the search value matches with the middle value in the list we complete the search. Otherwise we eleminate half of the list of elements by choosing whether to procees with the right or left half of the list depending on the value of the item searched.

This is possible as the list is sorted and it is much quicker than linear search.Here we divide the given list and conquer by choosing the proper half of the list. We repeat this approcah till we find the element or conclude about it's absence in the list.

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Binary Search \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

class BSearch:

    def \_\_init\_\_(self, list1):

        # Initialize the Binary Search object with a sorted list

        self.list = list1  # Assign the provided list to the instance variable

        self.left = 0  # Initialize the left pointer to the start index of the list

        self.right = len(self.list) - 1  # Initialize the right pointer to the end index of the list

    def bSearch(self, key):

        # Perform binary search to find the index of the key in the list

        while self.left <= self.right:

            midValue = (self.left + self.right) // 2  # Calculate the middle index

            if self.list[midValue] == key:

                # If the key is found at the middle index, return the index

                return midValue

            elif key > self.list[midValue]:

                # If the key is greater than the value at the middle index, update the left pointer

                self.left = midValue + 1

            else:

                # If the key is less than the value at the middle index, update the right pointer

                self.right = midValue - 1

        return None  # Return None if the key is not found in the list after the loop ends

list1 = [2, 7, 19, 34, 53, 72]

obj = BSearch(list1)  # Create an object of the BSearch class with the list

n = int(input("Enter the key: "))  # Prompt the user to enter the key to search for

# Call the bSearch method to search for the key in the list

result = obj.bSearch(n)

# Check if the key is found in the list and print the result accordingly

if n in list1:

    print(f"{n} Found at index:", result)

else:

    print(f"{n} Not Found!")

# Python – Recursion

[Previous](https://www.tutorialspoint.com/python_data_structure/python_divide_and_conquer.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_backtracking.htm)

Recursion allows a function to call itself. Fixed steps of code get executed again and again for new values. We also have to set criteria for deciding when the recursive call ends. In the below example we see a recursive approach to the binary search. We take a sorted list and give its index range as input to the recursive function.

## Binary Search using Recursion

We implement the algorithm of binary search using python as shown below. We use an ordered list of items and design a recursive function to take in the list along with starting and ending index as input. Then, the binary search function calls itself till find the searched item or concludes about its absence in the list.

class BS:

    def \_\_init\_\_(self, left, right, list1):

        # Initialize the Binary Search object with left and right pointers and the list to search

        self.list = list1  # Assign the provided list to the instance variable

        self.left = left  # Initialize the left pointer

        self.right = right  # Initialize the right pointer

    def BSearch(self, key):

        # Perform binary search to find the index of the key in the list

        if self.left <= self.right:  # Check if left pointer is less than or equal to right pointer

            midValue = (self.left + self.right) // 2  # Calculate the middle index

            if key == self.list[midValue]:  # If key is found at middle index

                return midValue  # Return the index of the key

            elif key < self.list[midValue]:  # If key is less than value at middle index

                # Recursively search the left half of the list

                return BS(self.left, midValue - 1, self.list).BSearch(key)

            else:  # If key is greater than value at middle index

                # Recursively search the right half of the list

                return BS(midValue + 1, self.right, self.list).BSearch(key)

        else:

            return None  # Return None if key is not found after the search

list1 = [8, 11, 24, 56, 88, 131]  # Define the sorted list

obj = BS(0, 5, list1)  # Create an object of the BS class with left pointer, right pointer, and the list

n = int(input("Enter the key value: "))  # Prompt the user to enter the key to search for

result = obj.BSearch(n)  # Call the BSearch method to search for the key in the list

if result is not None:  # If key is found in the list

    print(f"{n} found at index: ", result)  # Print the index where key is found

else:  # If key is not found in the list

    print(f"{n} not found!")  # Print a message indicating key is not found

# Python - Graphs

[Previous](https://www.tutorialspoint.com/python_data_structure/python_heaps.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_algorithm_design.htm)

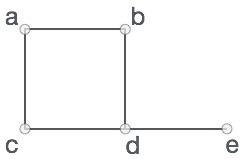
A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges. The various terms and functionalities associated with a graph is described in great detail in our tutorial here.

In this chapter we are going to see how to create a graph and add various data elements to it using a python program. Following are the basic operations we perform on graphs.

* Display graph vertices
* Display graph edges
* Add a vertex
* Add an edge
* Creating a graph

A graph can be easily presented using the python dictionary data types. We represent the vertices as the keys of the dictionary and the connection between the vertices also called edges as the values in the dictionary.

Take a look at the following graph −



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

#Example 1 We can present this graph in a python program as below

#Create the dictionary with graph element

graph={

    "a":["b","c"],

    "b":["a","d"],

    "c":["a","d"],

    "d":["e"],

    "e":["d"]

}

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Create Graph from dict \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

#printthe graph

print(graph)

"""

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Display graph vertices \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['a', 'b', 'c', 'd', 'e']

"""

#Example 2 Display graph vertices

"""

To display the graph vertices we simple find the keys of the graph dictionary. We use the keys() method."""

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Display graph vertices \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

class graph:

    def \_\_init\_\_(self,gdict=None):

        if gdict is None:

            gdict=[]

        self.gdict=gdict

    #Get the keys of the dict

    def getVerticks(self):

        return list(self.gdict.keys())

#Create the dict with graph elements

graph\_Dict= {

    "a":["b","c"],

    "b":["a","d"],

    "c":["a","d"],

    "d":["e"],

    "e":["d"]

}

#Create object and pass this dict

obj=graph(graph\_Dict)

print(obj.getVerticks())

"""

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Display graph vertices \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['a', 'b', 'c', 'd', 'e']

"""

#Example 3 Display graph edges

"""

Finding the graph edges is little tricker than the vertices as we have to find each of the pairs of vertices which have an edge in between them. So we create an empty list of edges then iterate through the edge values associated with each of the vertices. A list is formed containing the distinct group of edges found from the vertices."""

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Display graph edges \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ")

class graph:

    def \_\_init\_\_(self,gdict=None):

        if gdict is None:

            gdict=[]

        self.gdict=gdict

    def edges(self):

        return self.findedges()

    #Find the distinct list of edges

    def findedges(self):

        edgename=[]

        for vrtx in self.gdict:

            for nextvrtx in self.gdict[vrtx]:

                if {nextvrtx,vrtx} not in edgename:

                    edgename.append({vrtx,nextvrtx})

        return edgename

#Create the dict with graph elements

graph\_Dict= {

    "a":["b","c"],

    "b":["a","d"],

    "c":["a","d"],

    "d":["e"],

    "e":["d"]

}

#Create object and pass this dict

obj=graph(graph\_Dict)

print(obj.edges())

#Example 4  Adding a vertex

"""

Adding a vertex is straight forward where we add another additional key to the graph dictionary."""

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Adding a vertex \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

class graph:

    def \_\_init\_\_(self,gdict=None):

        if gdict is None:

            gdict=[]

        self.gdict=gdict

    #Get the keys of the dict

    def getVerticks(self):

        return list(self.gdict.keys())

    #Add vertix

    def addVertex(self,vrtx):

        if vrtx not in self.gdict:

            self.gdict[vrtx]=[]

#Create the dict with graph elements

graph\_Dict= {

    "a":["b","c"],

    "b":["a","d"],

    "c":["a","d"],

    "d":["e"],

    "e":["d"]

}

#Create object and pass this dict

obj=graph(graph\_Dict)

obj.addVertex('f')

print(obj.getVerticks())

#Example 5 Adding an edge

"""

Adding an edge to an existing graph involves treating the new vertex as a tuple and validating if the edge is already present.

If not then the edge is added."""

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Add graph edges \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ")

class graph:

    def \_\_init\_\_(self,gdict=None):

        if gdict is None:

            gdict=[]

        self.gdict=gdict

    def edges(self):

        return self.findedges()

    def addEdges(self,edge):

        edge=set(edge)

        (vrtx1,vrtx2)=tuple(edge)

        if vrtx1 in self.gdict:

            self.gdict[vrtx1].append(vrtx2)

        else:

            self.gdict[vrtx1]=[vrtx2]

    #Find the distinct list of edges

    def findedges(self):

        edgename=[]

        for vrtx in self.gdict:

            for nextvrtx in self.gdict[vrtx]:

                if {nextvrtx,vrtx} not in edgename:

                    edgename.append({vrtx,nextvrtx})

        return edgename

#Create the dict with graph elements

graph\_Dict= {

    "a":["b","c"],

    "b":["a","d"],

    "c":["a","d"],

    "d":["e"],

    "e":["d"]

}

#Create object and pass this dict

obj=graph(graph\_Dict)

obj.addEdges({'a','e'})

obj.addEdges({'a','c'})

print(obj.edges())

# Python - Graph Algorithms

[Previous](https://www.tutorialspoint.com/python_data_structure/python_searching_algorithms.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_algorithm_analysis.htm)

Graphs are very useful data structures in solving many important mathematical challenges. For example computer network topology or analysing molecular structures of chemical compounds. They are also used in city traffic or route planning and even in human languages and their grammar. All these applications have a common challenge of traversing the graph using their edges and ensuring that all nodes of the graphs are visited. There are two common established methods to do this traversal which is described below.

## Depth First Traversal

Also called depth first search (DFS),this algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration. We implement DFS for a graph in python using the set data types as they provide the required functionalities to keep track of visited and unvisited nodes.

#EXample 1 DFT (Depth First Traversal)

# Define a class named 'graph'

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* DFS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

class graph:

    # Define the initialization method for the class

    def \_\_init\_\_(self, gdict=None):

        # Check if a graph dictionary is provided, otherwise initialize an empty dictionary

        if gdict is None:

            gdict = {}

        # Assign the provided or initialized dictionary to the 'gdict' attribute of the class instance

        self.gdict = gdict

    # Define the Depth First Search (DFS) method

    def DFS(self, start, visited=None):

        # Initialize the set of visited nodes if not provided

        if visited is None:

            visited = set()

        # Mark the current node as visited

        visited.add(start)

        # Print the current node

        print(start)

        # Iterate through the adjacent nodes of the current node

        for next\_node in self.gdict[start] - visited:

            # Recursively call DFS for unvisited adjacent nodes

            self.DFS(next\_node, visited)

        # Return the set of visited nodes

        return visited

# Define the graph as a dictionary

gdict = {

   "a": set(["b", "c"]),

   "b": set(["a", "d"]),

   "c": set(["a", "d"]),

   "d": set(["e"]),

   "e": set(["a"])

}

# Create an instance of the graph class

obj = graph(gdict)

# Perform DFS traversal starting from node 'a'

obj.DFS("a")

## Breadth First Traversal

Also called breadth first search (BFS),this algorithm traverses a graph breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration. Please visit this link in our website to understand the details of BFS steps for a graph.

We implement BFS for a graph in python using queue data structure discussed earlier. When we keep visiting the adjacent unvisited nodes and keep adding it to the queue. Then we start dequeue only the node which is left with no unvisited nodes. We stop the program when there is no next adjacent node to be visited.

#Example 2 BFS (Breadth first Search)

# Print a header indicating BFS (Breadth First Search) is being performed

print("\n\n \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* BFS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

# Import the 'collections' module for deque data structure

import collections

# Define a class named 'graph'

class graph:

    # Define the initialization method for the class

    def \_\_init\_\_(self, gdict=None):

        # Check if a graph dictionary is provided, otherwise initialize an empty dictionary

        if gdict is None:

            gdict = {}

        # Assign the provided or initialized dictionary to the 'gdict' attribute of the class instance

        self.gdict = gdict

    # Define the BFS (Breadth First Search) method

    def BFS(self, start\_Node):

        # Track the visited and unvisited nodes using a set for 'seen' and a deque for 'dqueue'

        seen, dqueue = set([start\_Node]), collections.deque([start\_Node])

        # Iterate until the deque is empty

        while dqueue:

            # Remove the leftmost node from the deque and assign it to 'vertex'

            vertex = dqueue.popleft()

            # Call the 'marked' method to print the visited node

            self.marked(vertex)

            # Iterate through the adjacent nodes of 'vertex'

            for node in self.gdict[vertex]:

                # If the adjacent node is not visited yet

                if node not in seen:

                    # Mark it as visited and add it to the deque for further traversal

                    seen.add(node)

                    dqueue.append(node)

    # Define the 'marked' method to print the visited node

    def marked(self, n):

        # Print the visited node

        print(n)

# Define the graph as a dictionary

gdict = {

   "a": set(["b", "c"]),

   "b": set(["a", "d"]),

   "c": set(["a", "d"]),

   "d": set(["e"]),

   "e": set(["a"])

}

# Create an instance of the graph class with the provided graph dictionary

obj = graph(gdict)

# Perform BFS traversal starting from node 'a'

obj.BFS("a")

# Python - Backtracking

[Previous](https://www.tutorialspoint.com/python_data_structure/python_recursion.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_sorting_algorithms.htm)

Backtracking is a form of recursion. But it involves choosing only option out of any possibilities. We begin by choosing an option and backtrack from it, if we reach a state where we conclude that this specific option does not give the required solution. We repeat these steps by going across each available option until we get the desired solution.

Below is an example of finding all possible order of arrangements of a given set of letters. When we choose a pair we apply backtracking to verify if that exact pair has already been created or not. If not already created, the pair is added to the answer list else it is ignored.

def permut(a, s):

    # Define a function called permut that takes two arguments: list (an integer) and s (a list of characters)

    if a == 1:

        # Base case: if the value of 'a' is 1, return the list 's'

        return s

    else:

        # Recursive case:

        return [

            y + x  # For each character 'y' in the result of permut(1, s)

            for y in permut(1, s)

            for x in permut(a - 1, s)  # For each character 'x' in the result of permut(list - 1, s)

        ]

# Example 1: permut(1, ["a", "b", "c"])

print(permut(1, ["a", "b", "c"]))

# Example 2: permut(2, ["a", "b", "c"])

print(permut(2, ["a", "b", "c"]))

# Python - Algorithm Design

[Previous](https://www.tutorialspoint.com/python_data_structure/python_graphs.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_divide_and_conquer.htm)

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.

## Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

* **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
* **Input** − An algorithm should have 0 or more well-defined inputs.
* **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
* **Finiteness** − Algorithms must terminate after a finite number of steps.
* **Feasibility** − Should be feasible with the available resources.
* **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

## How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

### Example

Let's try to learn algorithm-writing by using an example.

* **Problem** − Design an algorithm to add two numbers and display the result.

**step 1** − START

**step 2** − declare three integers **a**, **b** & **c**

**step 3** − define values of **a** & **b**

**step 4** − add values of **a** & **b**

**step 5** − store output of step 4 to **c**

**step 6** − print **c**

**step 7** − STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

**step 1** − START ADD

**step 2** − get values of **a** & **b**

**step 3** − c ← a + b

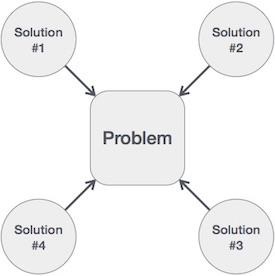
**step 4** − display c

**step 5** − STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

# Python - Sorting Algorithms

[Previous](https://www.tutorialspoint.com/python_data_structure/python_backtracking.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_searching_algorithms.htm)

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Below we see five such implementations of sorting in python.

* Bubble Sort
* Merge Sort
* Insertion Sort
* Shell Sort
* Selection Sort

## Bubble Sort

It is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

**Example** #Example 1 Boble Sort

count=0  # Initialize a counter variable for iterations

def BobleSort(list1):

    global count

    count=0

    # Loop for iteration

    for i in range(len(list1)):

        swapped=False

        # Loop for comparing elements

        for j in range(0, len(list1) - i - 1):

            # Compare adjacent elements

            if list1[j] > list1[j + 1]:

                # Swap if the current element is greater than the next element

                list1[j],list1[j+1]=list1[j+1],list1[j]

                # temp = list1[j]

                # list1[j] = list1[j + 1]

                # list1[j + 1] = temp

                count += 1  # Increment the counter after each iteration of the outer loop

                swapped=True

    # no swapping means the array is already sorted

    # so no need for further comparison

        if not swapped:

            break

list1 = [-2, 45, 0, 11, -9]

BobleSort(list1)  # Call the Bubble Sort function

# Print the sorted list in ascending order

print(f"Sorted list in Ascending order:", list1)

# Print the number of iterations performed during sorting

print(f"Number of iterations:", count)

### Output

## Sorted list in Ascending order: [-9, -2, 0, 11, 45]

## Number of iterations: 6

## Merge Sort

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

### Example

#EXample 2 Merged Sort

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Merge Sort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

def Merge\_Sort(unsorted\_list):

    # Base case: if the length of the list is 1 or less, it's already sorted

    if len(unsorted\_list) <= 1:

        return unsorted\_list

    # Find the middle point and divide the list into two halves

    middle = len(unsorted\_list) // 2

    left\_list = unsorted\_list[:middle]  # Left half of the list

    right\_list = unsorted\_list[middle:]  # Right half of the list

    # Recursively call Merge\_Sort on each half of the list

    left\_list = Merge\_Sort(left\_list)

    right\_list = Merge\_Sort(right\_list)

    # Merge the sorted halves using the merge function

    return list(merge(left\_list, right\_list))

def merge(left\_list, right\_list):

    result = []  # Initialize an empty list to store the merged result

    # Loop until either left\_list or right\_list becomes empty

    while len(left\_list) != 0 and len(right\_list) != 0:

        # Compare the first elements of left\_list and right\_list

        if left\_list[0] < right\_list[0]:

            # If the first element of left\_list is smaller, append it to result

            result.append(left\_list[0])

            # Remove the first element from left\_list

            left\_list.remove(left\_list[0])

        else:

            # If the first element of right\_list is smaller, append it to result

            result.append(right\_list[0])

            # Remove the first element from right\_list

            right\_list.remove(right\_list[0])

    # Add the remaining elements of left\_list and right\_list to result

    if len(left\_list) == 0:

        result += right\_list

    else:

        result += left\_list

    return result  # Return the merged result

unsorted\_list = [64, 34, 25, 12, 11, 90]

# Call Merge\_Sort function to sort the unsorted\_list

print("Sorted list After Merge Sort:", Merge\_Sort(unsorted\_list))

### Output

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Merge Sort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Sorted list After Merge Sort: [11, 12, 25, 34, 64, 90]

## Insertion Sort

Insertion sort involves finding the right place for a given element in a sorted list. So in 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 previous two sorted elements. This way we gradually go on adding more elements to the already sorted list by putting them in their proper position.

### Example

#Example 3 Insertion Sort

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Insertion Sort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

def insertion\_Sort(input\_list):

    # Iterate over each element in the input\_list, starting from the second element

    for i in range(1, len(input\_list)):

        next\_item = input\_list[i]  # Store the current element to be compared and inserted

        j = i - 1  # Start comparing with the previous element

        # Compare the current element with the previous ones and shift them if necessary

        while j >= 0 and input\_list[j] > next\_item:

            input\_list[j + 1] = input\_list[j]  # Shift the element to the right

            j -= 1  # Move to the previous element

        input\_list[j + 1] = next\_item  # Insert the current element at its correct position

    return input\_list  # Return the sorted list

# Create a list to be sorted using insertion sort

input\_list = [19, 2, 31, 45, 30, 11, 121, 27]

# Call the insertion\_Sort function to sort the input\_list

sorted\_list = insertion\_Sort(input\_list.copy())  # Use copy to avoid modifying the original list

print("Sorted list After insertion sort:", sorted\_list)  # Print the sorted list

### Output

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Insertion Sort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Sorted list After insertion sort: [2, 11, 19, 27, 30, 31, 45, 121]

## Shell Sort

Shell Sort involves sorting elements which are away from each other. We sort a large sublist of a given list and go on reducing the size of the list until all elements are sorted. The below program finds the gap by equating it to half of the length of the list size and then starts sorting all elements in it. Then we keep resetting the gap until the entire list is sorted.

#Example 4 Shell Sort

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Shell Sort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

def Shell\_Sort(input\_list):

    # Initialize the gap size to half the length of the input list

    gap = len(input\_list) // 2

    # Iterate until the gap becomes 0

    while gap > 0:

        # Perform insertion sort for elements at each gap interval

        for i in range(gap, len(input\_list)):

            temp = input\_list[i]  # Store the current element to be compared and inserted

            j = i  # Start comparing with the previous elements at the specified gap

            # Compare the current element with the elements at a gap distance and shift them if necessary

            while j >= gap and input\_list[j - gap] > temp:

                input\_list[j] = input\_list[j - gap]  # Shift the element to the right by the gap

                j -= gap  # Move to the previous element at the specified gap

            input\_list[j] = temp  # Insert the current element at its correct position

        # Reduce the gap for the next iteration

        gap = gap // 2

# Create a list to be sorted using shell sort

input\_list = [19, 2, 31, 45, 30, 11, 121, 27]

Shell\_Sort(input\_list)  # Call the Shell\_Sort function to sort the input\_list

print("Sorted list after shell sort:", input\_list)  # Print the sorted list

Output

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Shell Sort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sorted list after shell sort: [2, 11, 19, 27, 30, 31, 45, 121]Selection Sort

In selection sort we start by finding the minimum value in a given list and move it to a sorted list. Then we repeat the process for each of the remaining elements in the unsorted list. The next element entering the sorted list is compared with the existing elements and placed at its correct position.So, at the end all the elements from the unsorted list are sorted.

Example

#Example 5 Selection Sort

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Selection Sort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

def Selection\_sort(input\_list):

    # Iterate over each index of the input\_list

    for current\_idx in range(len(input\_list)):

        min\_idx = current\_idx  # Assume the current index as the minimum index

        # Iterate over the unsorted part of the list to find the minimum value

        for jth\_idx in range(current\_idx + 1, len(input\_list)):

            # Check if the value at the min\_idx is greater than the value at jth\_idx

            if input\_list[min\_idx] > input\_list[jth\_idx]:

                min\_idx = jth\_idx  # Update the minimum index to the index of the smaller value

        # Swap the minimum value with the value at the current index

        input\_list[current\_idx], input\_list[min\_idx] = input\_list[min\_idx], input\_list[current\_idx]

# Define a list to be sorted using selection sort

list = [19, 2, 31, 45, 30, 11, 121, 27]

# Call the Selection\_sort function to sort the list

Selection\_sort(list)

# Print the sorted list

print("Sorted list after selection sort:", list)

Output:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Selection Sort \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sorted list after selection sort: [2, 11, 19, 27, 30, 31, 45, 121]

# Python - Searching Algorithms

[Previous](https://www.tutorialspoint.com/python_data_structure/python_sorting_algorithms.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_graph_algorithms.htm)

Searching is a very basic necessity when you store data in different data structures. The simplest approach is to go across every element in the data structure and match it with the value you are searching for.This is known as Linear search. It is inefficient and rarely used, but creating a program for it gives an idea about how we can implement some advanced search algorithms.

## Linear Search

In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data structure.

#Example 1 Linear Search

# Print a header indicating that the following code is an example of Linear Search

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Linear Search \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

# Define a function called Linear\_Search that takes two parameters:

# - list\_Values: a list of values to search through

# - Search\_key: the value to search for in the list

def Linear\_Search(list\_Values, Search\_key):

    # Initialize a variable to keep track of the index being searched

    search\_at = 0

    # Initialize a variable to store the search result, initially set to False

    search\_result = False

    # Start a loop that continues as long as the search index is less than the length of the list

    # and the search result is False

    while search\_at < len(list\_Values) and search\_result is False:

        # Check if the value at the current index matches the search key

        if list\_Values[search\_at] == Search\_key:

            # If a match is found, set the search result to True

            search\_result = True

        else:

            # If no match is found, move to the next index by incrementing search\_at

            search\_at += 1  # increment index

    # Return the final search result (True if the value was found, False otherwise)

    return search\_result

# Create a list of values

list = [64, 34, 25, 12, 22, 11, 90]

# Call the Linear\_Search function to search for the value 12 in the list

# and print the result (True if found, False if not found)

print(Linear\_Search(list, 12))

# Call the Linear\_Search function to search for the value 10 in the list

# and print the result (True if found, False if not found)

print(Linear\_Search(list, 10))

## Interpolation Search

This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.Initially, the probe position is the position of the middle most item of the collection.If a match occurs, then the index of the item is returned.If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the subarray to the left of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.

def Interploation\_Search(list\_Values, key):

    # Initialize start and end indices for the search range

    start\_idx = 0

    end\_idx = len(list\_Values) - 1

    # Perform interpolation search as long as start index is less than or equal to end index

    # and the search key is within the range of values in the list

    while start\_idx <= end\_idx and key >= list\_Values[start\_idx] and key <= list\_Values[end\_idx]:

        # Calculate the mid-point using interpolation formula

        mid = start\_idx + int((float(end\_idx - start\_idx) / (list\_Values[end\_idx] - list\_Values[start\_idx])) \* (key - list\_Values[start\_idx]))

        # Check if the key is found at the mid-point

        if key == list\_Values[mid]:

            return "Searched element " + str(key) + " found at index: " + str(mid)

        # If key is greater than the value at mid-point, update start index

        elif key > list\_Values[mid]:

            start\_idx = mid + 1

        # If key is smaller than the value at mid-point, update end index

        else:

            end\_idx = mid - 1

    # If key is not found in the list, return a message indicating that

    return "Searched element " + str(key) + " not found in the list"

# Create a list of values

list = [2, 6, 11, 19, 27, 31, 45, 121]

# Perform interpolation search for key 50 in the list and print the result

print(Interploation\_Search(list, 50))

# Perform interpolation search for key 31 in the list and print the result

print(Interploation\_Search(list, 31))

# Python - Heaps

[Previous](https://www.tutorialspoint.com/python_data_structure/python_binary_search_tree.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_graphs.htm)

Heap is a special tree structure in which each parent node is less than or equal to its child node. Then it is called a **Min Heap**.

If each parent node is greater than or equal to its child node then it is called a **max heap**.

It is very useful is implementing priority queues where the queue item with higher weightage is given more priority in processing.

A detailed discussion on heaps is available in our website here. Please study it first if you are new to heap data structure. In this chapter we will see the implementation of heap data structure using python.

## Create a Heap

A heap is created by using python’s inbuilt library named heapq. This library has the relevant functions to carry out various operations on heap data structure. Below is a list of these functions.

* **heapify** − This function converts a regular list to a heap. In the resulting heap the smallest element gets pushed to the index position 0. But rest of the data elements are not necessarily sorted.
* **heappush** − This function adds an element to the heap without altering the current heap.
* **heappop** − This function returns the smallest data element from the heap.
* **heapreplace** − This function replaces the smallest data element with a new value supplied in the function.

## Creating a Heap

A heap is created by simply using a list of elements with the heapify function. In the below example we supply a list of elements and the heapify function rearranges the elements bringing the smallest element to the first position.

#Example 1 Creating a Heap

import heapq

l1=[21,1,45,78,3,5]

#Use heapify to rearrange the elements

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Creating heap \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print("Orignal list: ",l1)

heapq.heapify(l1)

print("After Heap list:",l1)

#Example 2 Inserting into heap

"""

Inserting a data element to a heap always adds the element at the last index.

But you can apply heapify function again to bring the newly added element to the first index only if it smallest in value.

In the below example we insert the number 8."""

print("\n\n \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Inserting in Heap \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

#Add element to above heaplist

heapq.heappush(l1,8)

print("\nAfter inserting 8 in Heap: ",l1)

heapq.heappush(l1,0)

print("\nAfter inserting 0 in Heap: ",l1)

#Example 3 Removing from heap

"""

You can remove the element at first index by using this function.

In the below example the function will always remove the element at the index position 1."""

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Removing from heap \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

#Remove element from the heap

heapq.heappop(l1)

print("\nAfter removing  from heap:",l1)

heapq.heappop(l1)

print("\nAfter removing  from heap:",l1)

#Example 4 Replacing in a Heap

"""

The heap replace function always removes the smallest element of the heap

and inserts the new incoming element at some place not fixed by any order."""

print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Replacing in heap\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

#Replace an element

heapq.heapreplace(l1,20)

print("\nAfter replacing (20): ",l1)

heapq.heapreplace(l1,40)

print("\nAfter replacing (40): ",l1)

# Python - Algorithm Analysis

[Previous](https://www.tutorialspoint.com/python_data_structure/python_graph_algorithms.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_big_o_notation.htm)

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

* **A Priori Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
* **A Posterior Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

## Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

* **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
* **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

## Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

* A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
* A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

**Algorithm: SUM(A, B)**

Step 1 − START

Step 2 − C ← A + B + 10

Step 3 − Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

## Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

# Python - Algorithm Types

[Previous](https://www.tutorialspoint.com/python_data_structure/python_algorithm_analysis.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_algorithm_classes.htm)

The efficiency and accuracy of algorithms have to be analysed to compare them and choose a specific algorithm for certain scenarios. The process of making this analysis is called Asymptotic analysis. It refers to computing the running time of any operation in mathematical units of computation.

For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.

Usually, the time required by an algorithm falls under three types −

* **Best Case** − Minimum time required for program execution.
* **Average Case** − Average time required for program execution.
* **Worst Case** − Maximum time required for program execution.

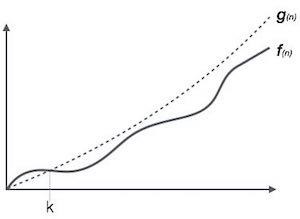
## Asymptotic Notations

The commonly used asymptotic notations to calculate the running time complexity of an algorithm.

* Ο Notation
* Ω Notation
* θ Notation

### Big Oh Notation, Ο

The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

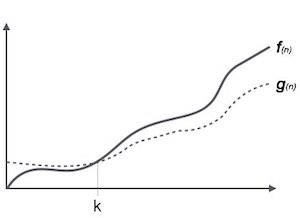


For example, for a function ***f*(n)**

Ο(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *f*(n) ≤ c.*g*(n) for all n > n0. }

### Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

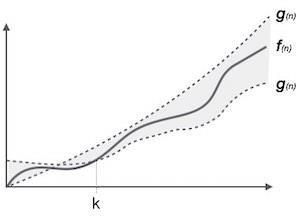


For example, for a function ***f*(n)**

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

### Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



θ(*f*(n)) = { *g*(n) if and only if *g*(n) = Ο(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n0. }

## Common Asymptotic Notations

A list of some common asymptotic notations is mentioned below −

|  |  |  |
| --- | --- | --- |
| constant | − | Ο(1) |
| logarithmic | − | Ο(log n) |
| linear | − | Ο(n) |
| n log n | − | Ο(n log n) |
| quadratic | − | Ο(n2) |
| cubic | − | Ο(n3) |
| polynomial | − | nΟ(1) |
| exponential | − | 2Ο(n) |

# Python - Algorithm Classes

[Previous](https://www.tutorialspoint.com/python_data_structure/python_big_o_notation.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_amortized_analysis.htm)

Algorithms are unambiguous steps which should give us a well-defined output by processing zero or more inputs. This leads to many approaches in designing and writing the algorithms. It has been observed that most of the algorithms can be classified into the following categories.

## Greedy Algorithms

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

So greedy algorithms look for a easy solution at that point in time without considering how it impacts the future steps. It is similar to how humans solve problems without going through the complete details of the inputs provided.

Most networking algorithms use the greedy approach. Here is a list of few of them −

* Travelling Salesman Problem
* Prim's Minimal Spanning Tree Algorithm
* Kruskal's Minimal Spanning Tree Algorithm
* Dijkstra's Minimal Spanning Tree Algorithm

## Divide and Conquer

This class of algorithms involve dividing the given problem into smaller sub-problems and then solving each of the sub-problem independently. When the problem can not be further sub divided, we start merging the solution to each of the sub-problem to arrive at the solution for the bigger problem.

The important examples of divide and conquer algorithms are −

* Merge Sort
* Quick Sort
* Kruskal's Minimal Spanning Tree Algorithm
* Binary Search

## Dynamic Programming

Dynamic programming involves dividing the bigger problem into smaller ones but unlike divide and conquer it does not involve solving each sub-problem independently. Rather the results of smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems.Dynamic algorithms are motivated for an overall optimization of the problem and not the local optimization.

The important examples of Dynamic programming algorithms are −

* Fibonacci number series
* Knapsack problem
* Tower of Hanoi

# Python - Amortized Analysis

[Previous](https://www.tutorialspoint.com/python_data_structure/python_algorithm_classes.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_algorithm_justifications.htm)

Amortized analysis involves estimating the run time for the sequence of operations in a program without taking into consideration the span of the data distribution in the input values. A simple example is finding a value in a sorted list is quicker than in an unsorted list.

If the list is already sorted, it does not matter how distributed the data is. But of course the length of the list has an impact as it decides the number of steps the algorithm has to go through to get the final result.

So we see that if the initial cost of a single step of obtaining a sorted list is high, then the cost of subsequent steps of finding an element becomes considerably low. So Amortized analysis helps us find a bound on the worst-case running time for a sequence of operations. There are three approaches to amortized analysis.

* **Accounting Method** − This involves assigning a cost to each operation performed. If the actual operation finishes quicker than the assigned time then some positive credit is accumulated in the analysis.

In the reverse scenario it will be negative credit. To keep track of these accumulated credits, we use a stack or tree data structure. The operations which are carried out early ( like sorting the list) have high amortized cost but the operations that are late in sequence have lower amortized cost as the accumulated credit is utilized. So the amortized cost is an upper bound of actual cost.

* **Potential Method** − In this method the saved credit is utilized for future operations as mathematical function of the state of the data structure. The evaluation of the mathematical function and the amortized cost should be equal. So when the actual cost is greater than amortized cost there is a decrease in potential and it is used utilized for future operations which are expensive.
* **Aggregate analysis**− In this method we estimate the upper bound on the total cost of n steps. The amortized cost is a simple division of total cost and the number of steps (n)..

# Python - Algorithm Justifications

[Previous](https://www.tutorialspoint.com/python_data_structure/python_amortized_analysis.htm)

[Next](https://www.tutorialspoint.com/python_data_structure/python_data_structure_quick_guide.htm)

In order to make claims about an Algorithm being efficient we need some mathematical tools as proof. These tools help us on providing a mathematically satisfying explanation on the performance and accuracy of the algorithms. Below is a list of some of those mathematical tools which can be used for justifying one algorithm over another.

* **Direct Proof** − It is direct verification of the statement by using the direct calculations. For example sum of two even numbers is always an even number. In this case just add the two numbers you are investigating and verify the result as even.
* **Proof by induction** − Here we start with a specific instance of a truth and then generalize it to all possible values which are part of the truth. The approach is to take a case of verified truth, then prove it is also true for the next case for the same given condition. For example all positive numbers of the form 2n-1 are odd. We prove it for a certain value of n, then prove it for the next value of n. This establishes the statement as generally true by proof of induction.
* **Proof by contraposition** − This proof is based on the condition If Not A implies Not B then A implies B. A simple example is if square of n is even then n must be even. Because if square on n is not even then n is not even.
* **Proof by exhaustion** − This is similar to direct proof but it is established by visiting each case separately and proving each of them. An example of such proof is the four color theorem.