Data Structures

**Stack:**

**Methods:**

**top():** returns the value on top of the stack

**push(arg):** takes only one argument and add it on the top of the stack

**pop():** remove and returns value on top of the stack

**reverse(arg):** It takes only one argument of type string and then reverse it

**is\_empty():** returns true is stack is empty else false

**is\_balance(arg):** it takes only one argument(in-order expression) and returns true if the expression is balanced otherwise false.

**Queue:**

**Methods:**

**enqueue(arg):** takes only one argument and add it to the end of the queue

**dequeue():** removes the elements from the front of queue. It does not require any parameters and returns the deleted item

**display():** returns the elements of the queue.

**Is\_empty():** returns the boolean value true if queue is empty otherwise false.

**front():** returns the front value of the queue

**rear ():** returns the rear value of the queue

**Circular Queue:**

**Attributes:** self.size, self.front, self.rear, self.queue

**Methods:**

**enqueue(arg):** takes only one argument and add it to the rear pointer of the queue.

**dequeue():** removes the elements from the front pointer of queue. It does not require any parameters and returns the deleted item

**display():** returns the elements of the queue.

**Linked\_list:**

**Methods:**

**display():**prints and returns the elements in the linked list

**insert\_last (arg):** takes one argument of non iterable type and add it to the end of the linked list.

**Insert\_first(arg):** takes one argument of non iterable type and add it to the front of the linked list.

**Insert\_position(index, data):** it takes two arguments of non iterable type. Fist is index at which the data is to be inserted and the second is data which is to be inserted.

**delete\_position (index):** it takes one argument of non iterable type and delete the node at the given index.

**delete\_data (data):** it takes one argument and delete the node at the given data.

**delete\_all ():** deletes the whole linked list.

**delete\_last ():** deletes the last node of the linked list

**delete\_first ():** deletes the first node of the linked list

**traverse ():** it traverse the linked list.

**count() :** returns the total numbers of elements in the linked list.

**find(data):** finds the node according to given data in the linked list. It takes only one argument of non iterable type.

**merge():**it merges the two linked list.

**divide\_odd ():** prints the odd data from the linked list.

**divide\_even ():** prints the even data from the linked list.

**Double Linked List:**

**Methods:**

**display():**prints and returns the elements in the linked list

**insert\_last (arg):** takes one argument of non iterable type and add it to the end of the linked list.

**Insert\_first(arg):** takes one argument of non iterable type and add it to the front of the linked list.

**Insert\_position(index, data):** it takes two arguments of non iterable type. Fist is index at which the data is to be inserted and the second is data which is to be inserted.

**delete\_position (index):** it takes one argument of non iterable type and delete the node at the given index.

**delete\_all ():** deletes the whole linked list.

**delete\_last ():** deletes the last node of the linked list

**delete\_first ():** deletes the first node of the linked list

**traverse ():** it traverse the linked list.

**count() :** returns the total numbers of elements in the linked list.

**find(data):** finds the node according to given data in the Double linked list. It takes only one argument of non iterable type.

**divide\_odd ():** prints the odd data from the linked list.

**divide\_even ():** prints the even data from the linked list.

**merge():**it merges the two linked list

**Binary Search Tree:**

**Methods:**

**insert(data):** insert the data into the tree. It takes only one numeric argument

**display():** display the tree through pre-order traversal

**find(data):** find the data from tree and then print the path of it. It takes only one argument

**level(data):** finds the level of the data and prints it. It takes only one argument.

**tree\_depth():** it returns the height of the tree

Sorting Algorithms

**Bubble Sorting:**

**Function: Bubble\_Sorting(arr, order = 0)**

It takes two arguments, the first argument is an array which is to be sorted and the second argument is order according to which array is to be sorted. 0 for ascending order and 1 is for descending order. The second argument is optional. By default, it will sort in ascending order.

**Selection Sorting:**

**Function: Selection\_Sorting(arr, order = 0)**

It takes two arguments, the first argument is an array which is to be sorted and the second argument is order according to which array is to be sorted. 0 for ascending order and 1 is for descending order. The second argument is optional. By default, it will sort in ascending order

**Insertion Sorting:**

**Function: Insertion\_Sorting(arr, order = 0)**

It takes two arguments; the first argument is an array which is to be sorted and the second argument is order according to which array is to be sorted. 0 for ascending order and 1 is for descending order. The second argument is optional. By default, it will sort in ascending order.

**Merge Sorting:**

**Function: Merge\_Sorting(arr, order = 0)**

It takes two arguments; the first argument is an array which is to be sorted and the second argument is order according to which array is to be sorted. 0 for ascending order and 1 is for descending order. The second argument is optional. By default, it will sort in ascending order.

Path Finding Algorithms

**Kruskal’s Algorithm:**

When we create object, it takes two arguments. One is connection of nodes with weights and the other argument is number of the nodes. The first argument should be in the following form:

[[source,destination,weight], [source,destination,weight], [source,destination,weight]]

The second argument is of type int.

**Prims Algorithm:**

It takes two arguments when object is created. The first is the source which is of type int and the second one is edges from the nodes. The second argument is dictionary. The format of the dictionary is as follow:

edges\_from\_node = {source:[(destination,weight), (destination,weight)]

source:[(destination,weight), (destination,weight)]}

**Dijkstra’s Algorithm:**

**Methods:** dijkstra(self,G, start, goal):

It takes two arguments. First is graph, the second is start node and the third one is end node. The graph should be of the following format:

{source:{(destination, weight), (destination, weight),

source:{(destination, weight), (destination, weight)}

**make\_path(self,parent, goal):** it takes two arguments. The first is parent(it is the object of the dijkstra() method) and the second argument is the goal (it the node to which the path is to be found)



