List is a data structure similar to array

Here are some key points related to list: **Adding an element:**

```
Appending an element to the end: O(1) (amortized)
Inserting an element at a specific position: O(n) - It involves
shifting elements to make space for the new element.
```

Deleting an element:

```
Removing an element by value: O(n) - It may require shifting
elements to fill the gap left by the removed element.
Removing an element by index: O(n) - Similar to removing by value,
it involves shifting elements after the removed index.
Pop from the end of the list: O(1) - Constant time complexity.
Pop from a specific position: O(n) - Similar to removing by index,
it involves shifting elements after the popped index.
```

Sorting:

```
Python's built-in list.sort() method: O(n log n) - Typically uses
Timsort, which is a hybrid sorting algorithm derived from merge
sort and insertion sort. It's efficient for many real-world data
sets.
Sorted() function: O(n log n) - Similar to list.sort(), it returns
a new sorted list.
```

Searching:

```
Linear search (using index() method or iterating over the list):
O(n) - In the worst case, it may need to traverse the entire list.
Binary search (for sorted lists): O(log n) - Requires the list to
be sorted first, but then it can find elements much faster than
linear search.
```

In [ ]:
```
data = []
print(type(data))
```

```
<class 'list'>
```

In [ ]:
```
data = list()
print(type(data))
```

```
<class 'list'>
```

The list declaration '[]' is faster than 'list()' as list() calls a class contributing an extra step

Operations in list

In [ ]:
```
#adding an item
data.append('1')
data.append('2')
```

In [ ]:
```
print(data)
```

```
['1', '2']
```

In [ ]:
```
#empty the list
data.clear()
data
```

Out[ ]:
```
[]
```

In [ ]:
```
#occurance of a number
data = [1,2,3,4,5,7,7,4,7]
data.count(7)
```

Out[ ]:
```
3
```

In [ ]:
```
#concat two lists
num=[5,7,8,99,]
data.extend(num)
data
```

Out[ ]:
```
[1, 2, 3, 4, 5, 7, 7, 4, 7, 5, 7, 8, 99]
```

In [ ]:
```
#sorting
data.sort()
data
```

Out[ ]:
```
[1, 2, 3, 4, 4, 5, 5, 7, 7, 7, 7, 8, 99]
```

In [ ]:
```
list_of_players = ['virat kholi','rohit sharma','ravindra jadeja']
```

In [ ]:
```
# slicing
for player in list_of_players:
    print(player.split())
    split_data = player.split()
    print(split_data[0])
```

```
['virat', 'kholi']
virat
['rohit', 'sharma']
rohit
['ravindra', 'jadeja']
ravindra
```

In [ ]:
```
#list comprehension
items = []

for i in range(1,11):
    items.append(i)
```

```
items
```

Out[ ]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [ ]:
```
#same thing can be done through
list_new=[i for i in range(1,11)]
list_new
```

Out[ ]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

break down of the above code is that the ":" the collen in the for loop indicates indentation pointing below but when the collen is removed the value is place before the for loop -> i for i in range(1,11) which in the above example is "i" which is added to the list_new

In [ ]:
```
#extention with the logic
#adding odd numbers

list_logic = [i for i in range(1,11,2)]
list_logic
```

Out[ ]: [1, 3, 5, 7, 9]

In [ ]:
```
#another example for list comprehension
# consider 3 for loops
x=y=z=1
n=2
final_list = []
for a in range(x+1):
    for b in range(y+1):
        for c in range(z+1):
            if a+b+c !=n:
                final_list.append([a,b,c])

print("final_list", final_list)

#list comprehension
final_list_comp = [[a,b,c] for a in range(x+1) for b in range(y+1) for c in range(z

print("final_list_comp", final_list_comp)
```

```
final_list [[0, 0, 0], [0, 0, 1], [0, 1, 0], [1, 0, 0], [1, 1, 1]]
final_list_comp [[0, 0, 0], [0, 0, 1], [0, 1, 0], [1, 0, 0], [1, 1, 1]]
```

Dictionary a key value pair interaction where key always remains unique

Adding an element:

```
Inserting a key-value pair: O(1) on average case. In some cases, it
may lead to rehashing and take O(n) time, but this is rare.
```

Deleting an element:

Removing a key-value pair: O(1) on average case.

Searching:

Retrieving the value for a given key: O(1) on average case. This is because dictionaries use hash tables for efficient key lookup.

In [ ]:
```
my_dict={
    "name":"tejas",
    "role":"developer"
}
```

In [ ]:
```
#getting keys
print(my_dict['name'])
print(my_dict.get('role'))
print("items",my_dict.items())
print("keys",my_dict.keys())
print("values",my_dict.values())
```

```
tejas
developer
items dict_items([('name', 'tejas'), ('role', 'developer')])
keys dict_keys(['name', 'role'])
values dict_values(['tejas', 'developer'])
```

In [ ]:
```
my_dict.update({"hobby":"teaching"})
my_dict
```

Out[ ]:
```
{'name': 'tejas', 'role': 'developer', 'hobby': 'teaching'}
```

In [ ]:
```
for key in my_dict.keys():
    print(key)
```

```
name
role
hobby
```

In [ ]:
```
for value in my_dict.values():
    print(key)
```

```
hobby
hobby
hobby
```

In [ ]:
```
for key,value in my_dict.items():
    print(key,value)
```

```
name tejas
role developer
hobby teaching
```

A tuple in Python is a collection data type that is similar to a list, but with some key differences. Here are the characteristics of tuples:

1. **Immutable**: Tuples are immutable, meaning once they are created, their elements cannot be changed, added, or removed. However, if an element of a tuple is mutable (like a list), that mutable element can be changed.

2. **Ordered**: Tuples maintain the order of elements as they are defined. This means that the order in which elements are inserted into a tuple is preserved when accessing them.

3. **Heterogeneous**: Tuples can contain elements of different data types. You can have integers, floats, strings, lists, dictionaries, or even other tuples as elements within a tuple.

4. **Indexing and Slicing**: Like lists, tuples support indexing and slicing operations. You can access individual elements of a tuple using square brackets and the index of the element you want to access.

Now, let's provide the time complexities for common operations on Python tuples:

1. **Adding an element**: Since tuples are immutable, you cannot add elements to an existing tuple. You would have to create a new tuple with the additional element, which takes O(n) time complexity, where n is the number of elements in the original tuple.

2. **Deleting an element**: Similarly, you cannot delete elements from a tuple due to its immutable nature. If you want to remove an element, you would need to create a new tuple without that element, which also takes O(n) time complexity.

3. **Searching**: Searching for an element in a tuple is similar to searching in a list. It has a time complexity of O(n) because it may need to traverse the entire tuple to find the desired element.

4. **Sorting**: Tuples are immutable, so you cannot sort them in place like lists. However, you can use the `sorted()` function to create a new sorted list from the elements of the tuple. The time complexity of sorting a list created from a tuple is O(n log n), where n is the number of elements in the tuple.

Overall, while tuples and lists share some similarities, their immutability makes their operations differ significantly in terms of time complexity.

Tuples are faster than lists ,as memory allocation is faster

In [ ]:
```
t=1,  #this is a tuple... cant belive?
print(type(t))
```

```
<class 'tuple'>
```

In [ ]:
```
tup = (1,2,3,4)
print(tup[2])
```

```
3
```

In [ ]:
```
#tuple are immutable
tup[2] =300
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[131], line 2
      1 #tuple are immutable
----> 2 tup[2] =300

TypeError: 'tuple' object does not support item assignment
```

In [ ]:
```
import timeit

print(timeit.timeit('x=(1,2,3,4)', number=1000000))
print(timeit.timeit('x=[1,2,3,4]', number=1000000))
print(timeit.timeit('x={1:2,2:4,3:6,4:8}', number=1000000))
```

```
0.022843700004159473
0.0802246999955969
0.2236955999978818
```

In [ ]:
```
import timeit

# Adding an element to list
list_add_time = timeit.timeit('my_list.append(10)', setup='my_list = []', number=10

# Deleting an element from list
list_delete_time = timeit.timeit('my_list.pop()', setup='my_list = list(range(1000)

# Searching in list
list_search_time = timeit.timeit('100 in my_list', setup='my_list = list(range(1000

# Sorting a list
list_sort_time = timeit.timeit('sorted(my_list)', setup='import random; my_list = [

# Adding an element to tuple
tuple_add_time = timeit.timeit('my_tuple += (10,)', setup='my_tuple = ()', number=1

# Deleting an element from tuple
tuple_delete_time = timeit.timeit('my_tuple = my_tuple[:-1]', setup='my_tuple = tup

# Searching in tuple
tuple_search_time = timeit.timeit('100 in my_tuple', setup='my_tuple = tuple(range(

# Sorting a list created from tuple
tuple_sort_time = timeit.timeit('sorted(my_tuple)', setup='import random; my_tuple

# Adding an element to dictionary
dict_add_time = timeit.timeit('my_dict[10] = 10', setup='my_dict = {}', number=1000

# Deleting an element from dictionary
dict_delete_time = timeit.timeit('my_dict.pop(999, None)', setup='my_dict = {i: i f

# Searching in dictionary
dict_search_time = timeit.timeit('999 in my_dict', setup='my_dict = {i: i for i in
```

```
print("List:")
print("Adding:", list_add_time)
print("Deleting:", list_delete_time)
print("Searching:", list_search_time)
print("Sorting:", list_sort_time)

print("\nTuple:")
print("Adding:", tuple_add_time)
print("Deleting:", tuple_delete_time)
print("Searching:", tuple_search_time)
print("Sorting (list from tuple):", tuple_sort_time)

print("\nDictionary:")
print("Adding:", dict_add_time)
print("Deleting:", dict_delete_time)
print("Searching:", dict_search_time)
```

```
List:
Adding: 5.9600002714432776e-05
Deleting: 3.759999526664615e-05
Searching: 7.559999357908964e-05
Sorting: 0.0002423999976599589

Tuple:
Adding: 0.002518299996154383
Deleting: 0.0010990000009769574
Searching: 7.46000005165115e-05
Sorting (list from tuple): 0.00021490000654011965

Dictionary:
Adding: 2.439999661874026e-05
Deleting: 4.339999577496201e-05
Searching: 3.300010575912893e-06
```

Stack is a data structure which follows LIFO ie Last in first out

Inserting (Pushing): O(1)
Deleting (Popping): O(1)
Searching (Peeking): O(1)
Sorting: O(n^2) or O(n log n) depending on the sorting algorithm used and whether additional data structures are allowed

In [ ]:
```
# Class based approch of Stack
import array
class Stack:
    def __init__(self):
        self.my_stack = array.array('i',[])
        self.top = -1
    def get_top(self):
        return len(self.my_stack)-1
    def is_empty(self):
        if len(self.my_stack):
            return False
        else:
            True
```

```
    def push(self,element):
        self.my_stack.append(element)

    def pop(self):
        if self.is_empty():
            print("stack is empty cant pop")
        else:
            self.my_stack.pop()
```

In [ ]:
```
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
stack.push(4)
print("top element",stack.get_top())

stack.my_stack
```

```
top element 3
```

Out[ ]:
```
array('i', [1, 2, 3, 4])
```

In [ ]:
```
stack.pop()
stack.pop()
stack.pop()
stack.pop()

stack.my_stack
```

Out[ ]:
```
array('i')
```

In [ ]:
```
stack.get_top()
```

Out[ ]:
```
-1
```

In [ ]:
```
stack.pop()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[71], line 1
----> 1 stack.pop()

Cell In[65], line 20, in Stack.pop(self)
     18     print("stack is empty cant pop")
     19 else:
---> 20     self.my_stack.pop()

IndexError: pop from empty array
```

Queue: Follows FIFO first in first out

```
    **Enqueuing** (Adding): O(1)
    **Dequeuing** (Removing): O(1)
```

Searching (Peeking): O(1) **Sorting**: O(n log n) if using efficient sorting algorithms, but the process may involve dequeuing and enqueuing elements, leading to additional time complexity considerations.

In [ ]:
```
#class based approch for Queue
class Queue:
    def __init__(self):
        self.my_queue = array.array('i',[])

    def get_rear(self):
            if len(self.my_queue) == 0:
                return None
            else:
                return self.my_queue[-1]

    def get_front(self):
        if len(self.my_queue) == 0:
            return None
        else:
            return self.my_queue[0]



    def is_empty(self):
        if len(self.my_queue):
            return False
        else:
            True
    #enqueue [][][]<--
    def enqueue(self, element):
        self.my_queue.append(element)
    #dequeue -->[][][]
    def dequeue(self):
        if self.is_empty():
            print("queueu is empty cant deque")
        else:
            self.my_queue.pop(0)
```

In [ ]:
```
queue  = Queue()
print(queue.my_queue)
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.my_queue)
print("front",queue.get_front())
print("rear",queue.get_rear())
```

```
array('i')
array('i', [1, 2, 3])
front 1
rear 3
```

In [ ]:
```
queue.dequeue()
queue.dequeue()
```

```
queue.dequeue()
```

In [ ]:
```
print(queue.my_queue)
```

```
array('i')
```

In [ ]:
```
queue.dequeue()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[100], line 1
----> 1 queue.dequeue()

Cell In[95], line 33, in Queue.dequeue(self)
     31     print("queueu is empty cant deque")
     32 else:
---> 33     self.my_queue.pop(0)

IndexError: pop from empty array
```

For a singly linked list implemented in Python, here are the time complexities for various operations:

1. **Insertion at the Head (Prepend):**
   - Time Complexity: O(1)
   - Explanation: Inserting a new node at the head of a singly linked list involves creating a new node and updating pointers to make the new node the head. Since these operations can be performed in constant time, the time complexity is O(1).
2. **Insertion at the Tail (Append):**
   - Time Complexity: O(n)
   - Explanation: Inserting a new node at the tail of a singly linked list requires traversing the entire list to reach the tail node, so the time complexity is proportional to the number of nodes in the list, i.e., O(n).
3. **Insertion at a Specific Position:**
   - Time Complexity: O(n)
   - Explanation: Inserting a new node at a specific position in the linked list also requires traversing the list to reach the desired position. If the position is close to the head, it may still take constant time, but in the worst case, it takes O(n) time if the position is at the end of the list.
4. **Deletion from the Head:**
   - Time Complexity: O(1)
   - Explanation: Deleting a node from the head of a singly linked list involves updating the head pointer to point to the next node, which can be done in constant time.
5. **Deletion from the Tail:**
   - Time Complexity: O(n)

- Explanation: Deleting a node from the tail of a singly linked list also requires traversing the entire list to reach the node before the tail node, so the time complexity is O(n).

6. **Deletion from a Specific Position:**

- Time Complexity: O(n)
- Explanation: Similar to insertion, deleting a node from a specific position requires traversing the list to reach the node before the one to be deleted. In the worst case, it takes O(n) time.

7. **Search (Traversal):**

- Time Complexity: O(n)
- Explanation: Searching for a specific element in a singly linked list requires traversing the entire list until the element is found or the end of the list is reached. Therefore, the time complexity is O(n).

8. **Accessing an Element by Index:**

- Time Complexity: O(n)
- Explanation: Unlike arrays, accessing an element in a singly linked list by index requires traversing the list from the head until the desired index is reached. Therefore, the time complexity is O(n).

In [ ]:
```
#class based approch for inkedLIst


class Node:
    def __init__(self,data):
        self.data = data
        self.nextval = None

class LinkedList:
    def __init__(self):
        self.head = None

    def traverse(self):
        current = self.head
        while  current is not None:
            print(current.data)
            current = current.nextval


n1 = Node(1)
n2 = Node(2)
n3 = Node(3)

# print(n1.data)

linkedlist = LinkedList()
linkedlist.head = n1
linkedlist.head.nextval = n2
```

```
n2.nextval = n3
n3.nextval = None

linkedlist.traverse()
```

```
1
2
3
```

Binary Tree: For a binary tree implemented in Python, here are the time complexities for various operations:

1. **Insertion:**
   - Time Complexity: O(log n) on average for balanced trees, O(n) in the worst case for unbalanced trees.
   - Explanation: In a balanced binary tree (e.g., AVL tree, Red-Black tree), insertion involves traversing the tree from the root to find the appropriate position for the new node. Since at each level, the number of nodes is roughly halved, the time complexity is logarithmic, O(log n). However, in the worst case, if the tree becomes unbalanced, insertion can take linear time, O(n).
2. **Deletion:**
   - Time Complexity: O(log n) on average for balanced trees, O(n) in the worst case for unbalanced trees.
   - Explanation: Similar to insertion, deletion in a balanced binary tree typically has a time complexity of O(log n) since it also involves traversing the tree to find the node to delete. However, in the worst case, deletion can take linear time, O(n), if the tree becomes unbalanced.
3. **Searching:**
   - Time Complexity: O(log n) on average for balanced trees, O(n) in the worst case for unbalanced trees.
   - Explanation: Searching in a balanced binary tree has a time complexity of O(log n) since it involves traversing the tree from the root downward, eliminating half of the nodes at each level. However, in the worst case, if the tree is unbalanced, searching can take linear time, O(n).
4. **Traversals (In-order, Pre-order, Post-order):**
   - Time Complexity: O(n)
   - Explanation: Traversing a binary tree requires visiting every node exactly once. Therefore, the time complexity of any traversal algorithm is linear, O(n), where n is the number of nodes in the tree.
5. **Accessing an Element by Key:**
   - Time Complexity: O(log n) on average for balanced trees, O(n) in the worst case for unbalanced trees.

- Explanation: Accessing an element in a binary search tree by key involves searching for the key in the tree, which has a time complexity of O(log n) on average for balanced trees and O(n) in the worst case for unbalanced trees.

It's important to note that the time complexities mentioned above are based on the assumption of a well-balanced binary tree. In real-world scenarios, the actual time complexity may vary depending on factors such as the structure of the tree and the specific operations being performed. Additionally, self-balancing binary trees like AVL trees and Red-Black trees are designed to maintain balance and provide efficient operations even in the worst-case scenarios.

In [ ]:
```python
#class implementation of graph
class Node:
    def __init__(self,data)-> None:
        self.left = None
        self.right = None
        self.data = data
    def show(self):
        if self.left:
            self.left.show()
        if self.right:
            self.right.show()
        print(self.data)

root = Node(100)
r_left = Node(99)
r_right = Node(101)
root.left = r_left
root.right = r_right
root.show()
```

```
99
101
100
```

In [ ]:
```python
import pprint

class Vertex:
    def __init__(self, name):
        self.name = name
        self.connections = []

    def add_edge(self,obj):
        self.connections.append(obj)

class Edge:
    def __init__(self):
        self.connections = []

    def add_edge(self,from_ver,to_ver):
        self.connections.append(from_ver.name)
        self.connections.append(to_ver.name)
```

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertices(self,obj):
        self.graph.update({obj.name:obj.connections})

v1 = Vertex("1")
v2 = Vertex("2")
v3 = Vertex("3")
v4 = Vertex("4")

e1 = Edge()
e1.add_edge(v1,v2)

e2 = Edge()
e2.add_edge(v1,v3)

e3 = Edge()
e3.add_edge(v2,v3)

e4 = Edge()
e4.add_edge(v3,v4)

e5 = Edge()
e5.add_edge(v4,v1)

#v1
v1.add_edge(e1.connections)
v1.add_edge(e2.connections)

#v2
v2.add_edge(e3.connections)

#v3
v3.add_edge(e4.connections)

#v4
v4.add_edge(e5.connections)

g1 = Graph()

g1.add_vertices(v1)
g1.add_vertices(v2)
g1.add_vertices(v3)
g1.add_vertices(v4)


pprint.pprint(g1.graph)
```

```
{'1': [['1', '2'], ['1', '3']],
 '2': [['2', '3']],
 '3': [['3', '4']],
 '4': [['4', '1']]}
```

In [ ]:

In [ ]:

In [ ]: