papergrid
Date: 10 / Aug / 2024

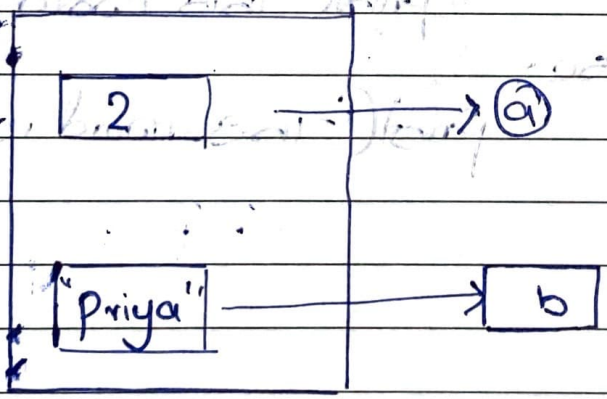
Day 11

Internal Working of Python

Heap	% Free Memory Dynamic
Stack	% Function calls and Local Variable
Static/Global	% Global Variable
Code	% Instruction

Application Memory

Memory Heap



a = 2
b = "priya"

① Python → Dynamic Memory Allocation

Dynamic list

lst = | 24 | 47 | 76 | 64 | 42 | 65 | 67 |

0 1 2 3 4 5 6 n=6

Linear Data structure

lst.append(67)

lst

0	24	-10
1	47	-9
2	76	-8
3	64	-7
4	42	-6
5	65	-5
6	67	-4
7	74	-3
8		-2
9		-1

(mapping {key : value})
index num user

lst(2) = 76 or lst(-5) = 76 } Random Access O(1)

lst.append(100)

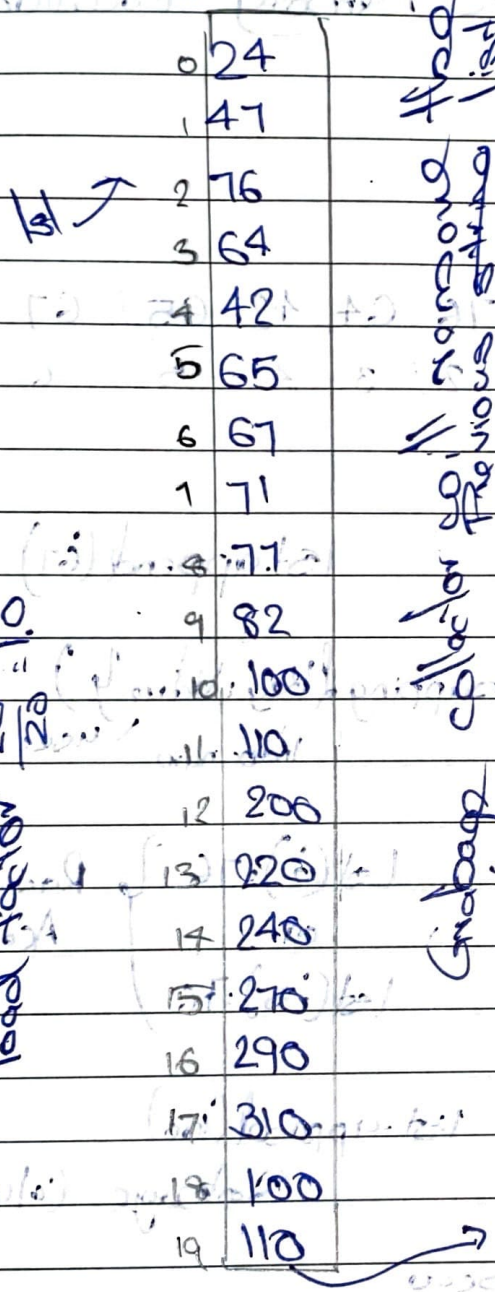
Garbage Collector

new list

n*2 ⇒ 10*2 = 20

copy the existing list of element

Load factor → 8/10 < 0.8



list →

Dynamic list

Expensive task & too much time

list.append(120)

above automatically delete the old string and create new with bigger string

same process repeat

⑤ Data Structures

→ Way to store the data in an efficient manner

Linear	Non-Linear
- Sequential manner	- Hierarchial manner
- Array / List, Stack, Queue, Linked List ↓ Website	- Tree, Graphs, Tries ↓ Networking

- Data structure in Python

1. List → Reverse
2. Tuple
3. Set
4. Dictionary

Inplace → not using any extra data structure to return the result

1. List : Mutable (can be changed)
2. Tuple : Immutable
3. Set : Remove duplicate value.

Dictionary

↳ Data Structures in Python

↳ List, Tuple (Difference)

↳ Set (Remove Duplicate entries)

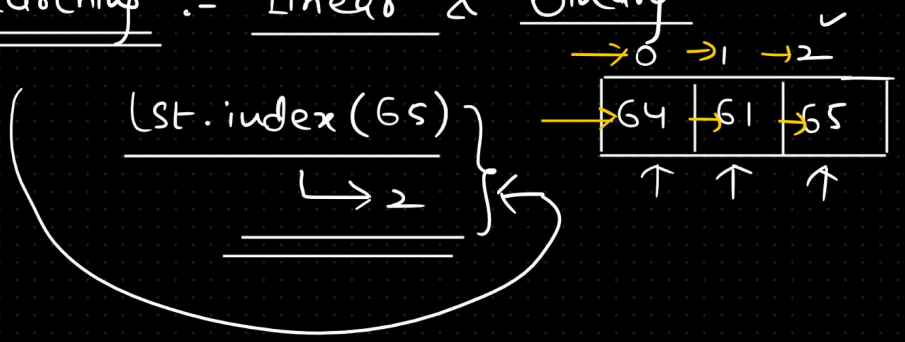
Dynamic

List → Load factor

Hashing Data Structure

↳ Searching :- Linear & Binary

O(n) O(log n)



Employee Data — Unique entity

dict → EID

key	value (Mapping)
1401	[Priya, Bhatia, 1 Lakh, 27, Bengaluru]

dict [1401] = value

List ⇒ [ ]

Tuple ⇒ ( )

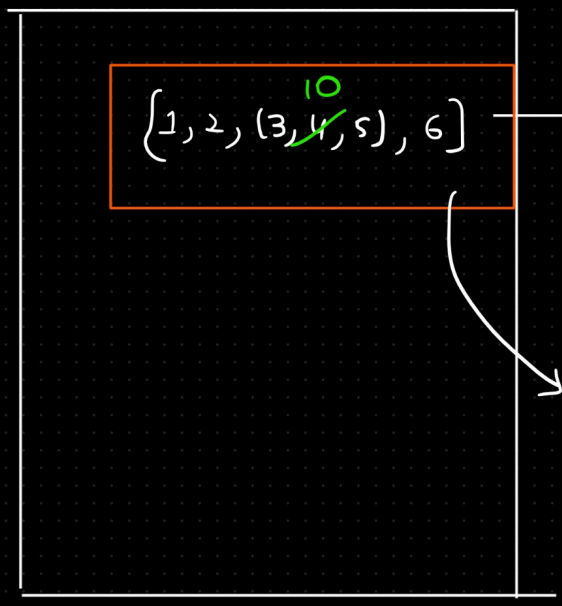
↳ Set ⇒ { }

↳ Dict ⇒ { key : value }

my_dict ⇒

```
{'name': ['priya', 'priyanka', 'ram', 'shyam', 'debashish'],
 'age': [27, 34, 46, 57, 25, 32],
 'salary': [100000, 150000, 50000, 200000, 300000, 115000]}
```

key → name, age, salary




```
import copy
original_list = [1, 2, [3, 4, 5], 6]
## access the element '4'
## original_list[2][1]

## shallow copy
shallow_copy = copy.copy(original_list)
## print(shallow_copy)

## update the original_list
original_list[2][1] = 10
print('Original list', original_list)
print('Shallow copy', shallow_copy)
```

Original_list

Shallow_copy

(1, 2, (3, 10, 5), 6)

```
import copy
original_list = [1, 2, [3, 4, 5], 6]
## access the element '4'
## original_list[2][1]

## deep copy
deep_copy = copy.deepcopy(original_list)
## print(shallow_copy)

## update the original_list
original_list[2][1] = 10

print('Original list', original_list)
print('Deep copy', deep_copy)
```