Ruchika PSA Class 4

Ruchika Shashidhara

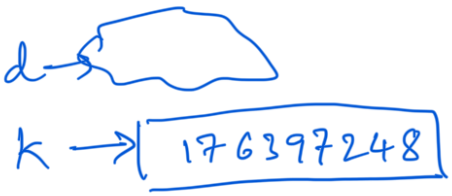
NU 002245068

How Dict works on basic type

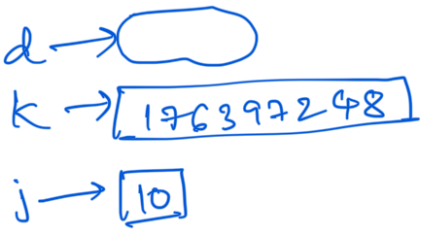
Dictionary

d = { } or d = dict()

k = 1763997248

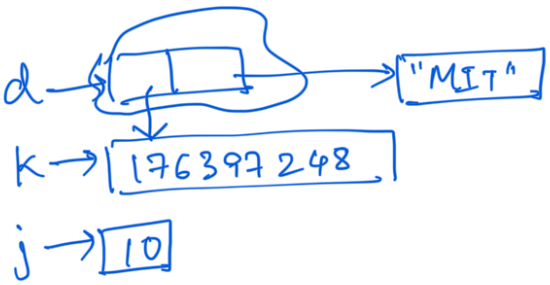


j = 10

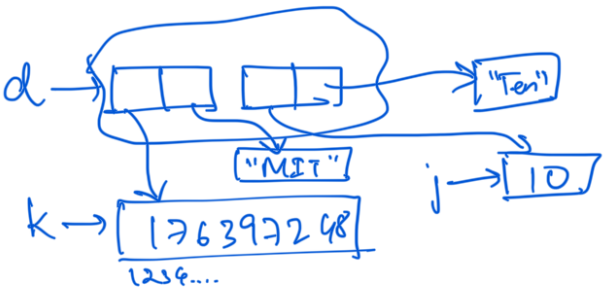


d[k] = "MIT"

// Insertion occurs in O(1) Time



d[j] = "Ten"



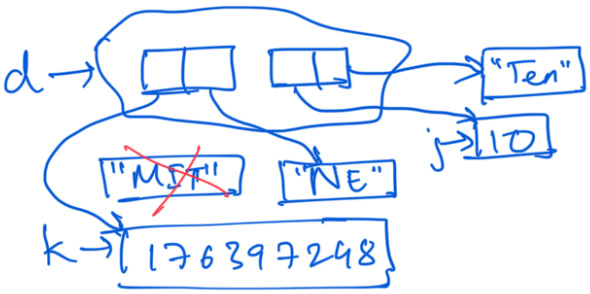
print (k, id(k))

1763997248 , 1234...

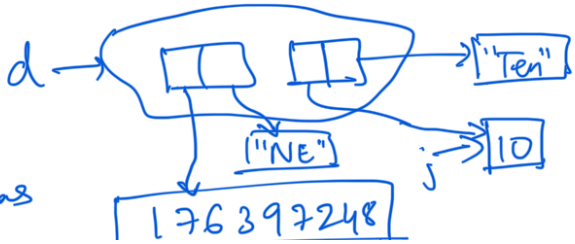
print (d)

d: "dict" : { 1763997248 : "MIT", 10 : "Ten" }

d[k] = "NE"



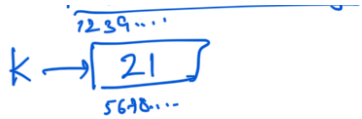
k = 21



print (k, id(k)) // Id of k was

- Dictionary & Set in Python are heterogenous containers

21 , 5678... chandra



print(d)

d : "dict" : { 1763997248 : "NE" , 10 : "Ten" }

print (d[10]) // Get occurs in O(1) Time

"Ten"

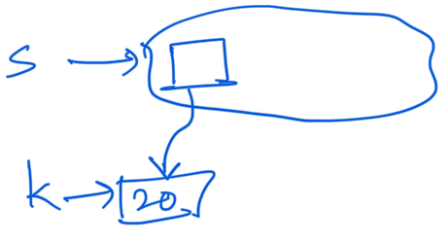
How Set works on basic type

Set

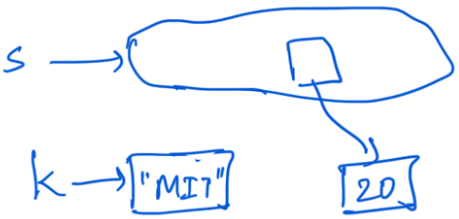
s = set ()

k : 20

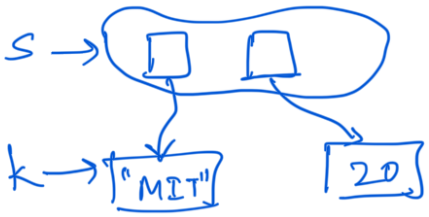
s.add(k) // Insertion occurs in O(1) time



k = "MIT"



s.add(k)



print (k in S) // Find occurs in O(1) time

True

print (20 in S)

True

print ("MIT" in s)

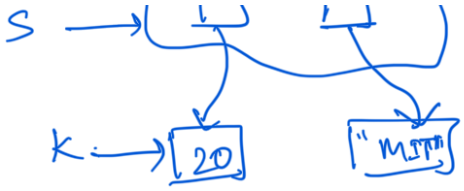
True

print (21 in s)

True

k = 20

```
print("MIT" in s)
True
print(k in s)
True
```



Creating our own class:

`__init__`	How to initialize the object?
`__str__`	How to print/represent the object?
`__len__`	What is the size of the object?
`__lt__`	How do we compare objects of same class type?
`__hash__`	How do we use objects uniquely in set & dictionaries?
`__add__`	How do we add objects?

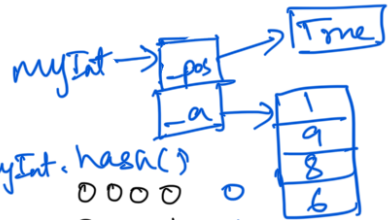
What do we want to acheive by writing `__hash__`?

To ensure hash of each object is different & probability of two hashes being the same is as minimal as possible. (If 2 objects clash : collision)

Our big number `Int()` class, we can write:

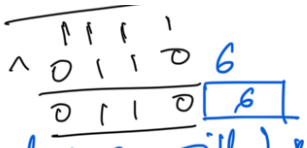
```
def __hash__(self)->"int":
    l = self.__get_key()
    h = 0
    for e in l:
        h^= hash(e)  # hash for basic python ints is given by Python itself
    return h
```

```
myInt = Int(1968)
```




```
myInt.hash()
0000  0
^0001  1
-----
 1110
^ 1001  9
-----
 1000
^ 1000  8
-----
```

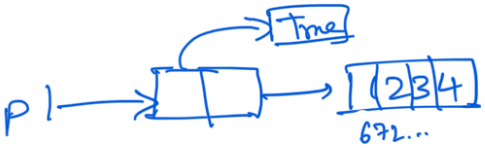
if def __hash__(self) → "int":
return 42 // Collisions will always occur → Containers will behave like a python list, operations like insert: O(n) not O(1)



For smaller int nums ($< 10^{12}$):
hash(n) = n, usually addresses change in Python, so python has not implemented hash using address

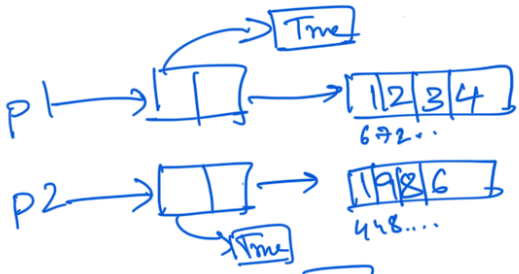
How Dict works on UDT

p1 = Int(1234)



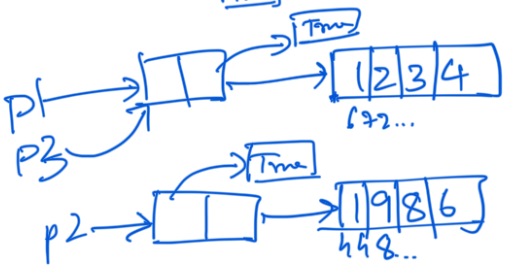
hash(p1) = 4
id(p1): 672...

p2 = Int(1986)



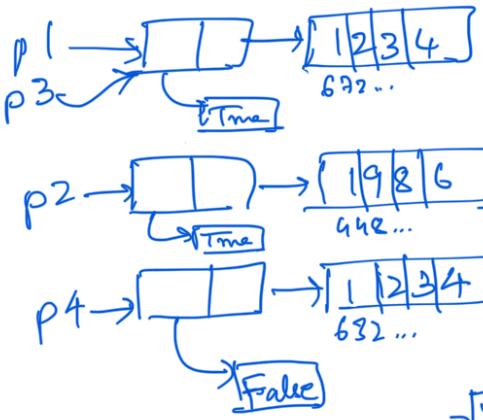
hash(p2) = 6
id(p2) = 448...

p3 = Int(1234)



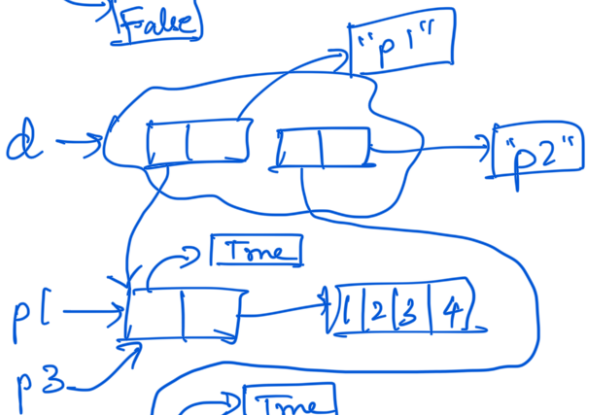
hash(p3) = 4
id(p3) = 672...

p4 = Int(-1234)



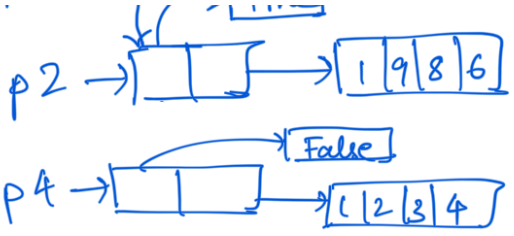
hash(p4) = 4
id(p4) = 632...

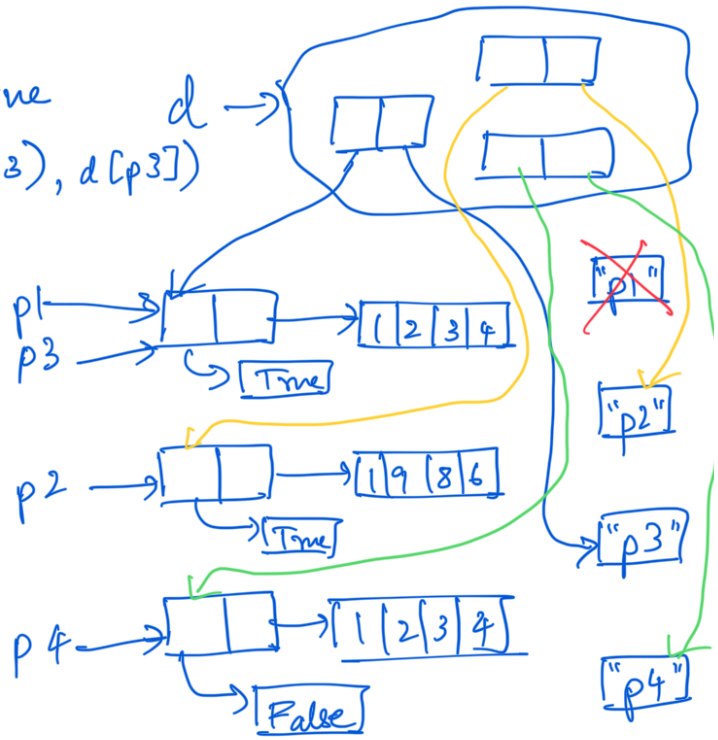
d = { }
d[p1] = "p1"
d[p2] = "p2"



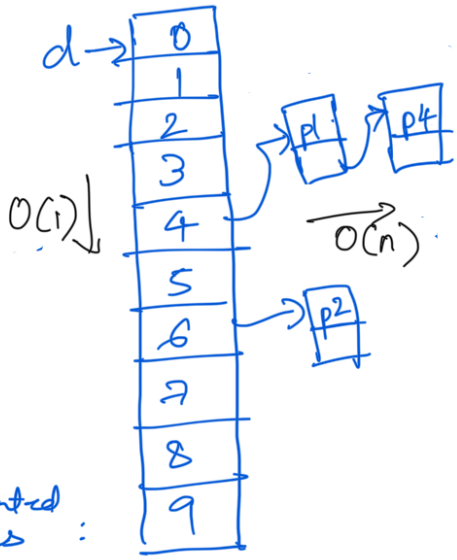
k = d.keys()

```
v = d.values()

for ki in k:
    print(ki, id(ki), hash(ki), d[ki])
// +1234, 672..., 4, "p1"
// +1986, 448..., 6, "p2"

print(p3 in d)    // True
print(p3, id(p3), hash(p3), d[p3])
    // +1234, 672..., 4, "p1"

d[p3] = "p3"

d[p4] = "p4"

k = d.keys()
v = d.values()

for ki in k:
    print(ki, id(ki), hash(ki), d[ki])
    // +1234, 672..., 4, "p3"
    // +1986, 448..., 6, "p2"
    // -1234, 632..., 4, "p4"
```





- +1234 & -1234 have the same hash, though they are 2 diff objects
- collision has occurred here
- In the dict, p1 & p4 will be represented as:



- So d has 3 diff objects: p1, p2, p4 now

```
p1[0] = 8        // Correct the data in dict
```

p1[0] = 8 // Wrong key ...

print(d[p1]) // Exception is thrown since we cannot find the object in the dictionary, since hash should have had changed, but it is not the case in Python

Why key cannot be changed

Hence we need private & public accessors, we shouldn't be allowing p1.__setitem__. In Java & Python,

Key should be immutable in Python Dict

How Set works on UDT

p1 = Int(1234)

p2 = Int(1986)

p3 = Int(1234)

p4 = Int(-1234)

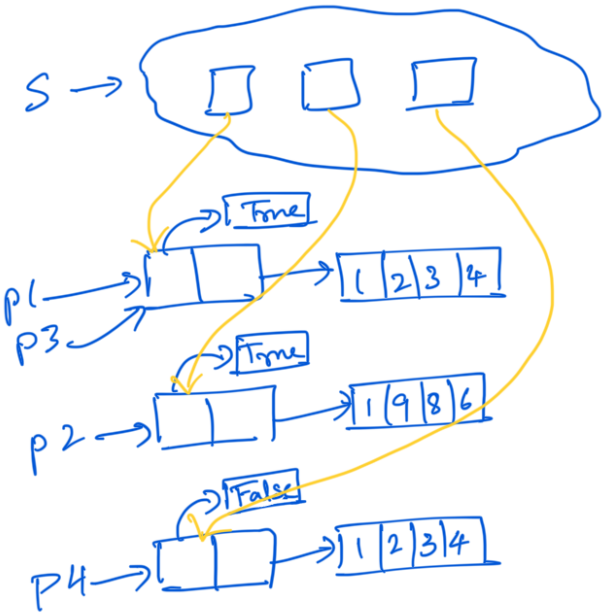
s = set()

s.add(p1)

s.add(p2)

s.add(p3)

s.add(p4)



for si in s:

print(si, id(si), hash(si))

// +1234, 672..., 4

// +1986, 448..., 6

// -1234, 632..., 4

Again, if we try:

p1[0] = 8

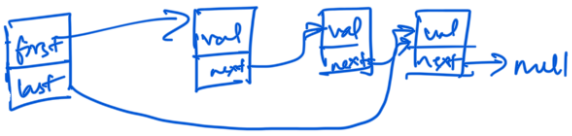
print(p1 in s) // Throws exception

Python Set

Object should be immutable in Python

Singly Linked List

SList : Singly linked list



Ex: Train, Chain

	Time complexity
Deletion/Insertion at head	$\Theta(1)$
Deletion/Insertion at tail	$\Theta(1)$
Deletion/Insertion at any position	$O(n)$

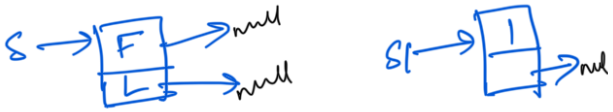
We can't access de list slist[k]
We can find any element in the SList in $O(n)$ Time

All operations here are $\Theta(1)$ Space

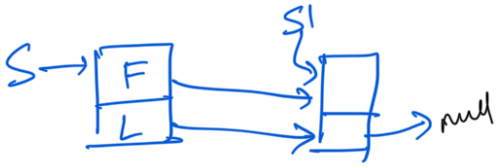
SList is a fragmented memory data store,
∴ we can't access slist[1], it is not continuous memory to find slist.address+1, we must traverse through the slist

Initialize
S = slist()

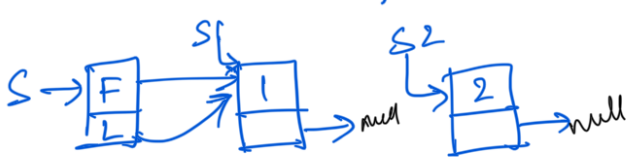
SList Append
S1 = snode(1)
Insert at tail
S.append(s1)



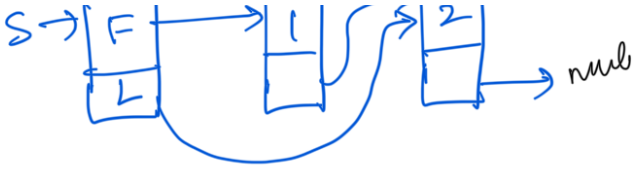
S2 = snode(2)



S.append(s2)



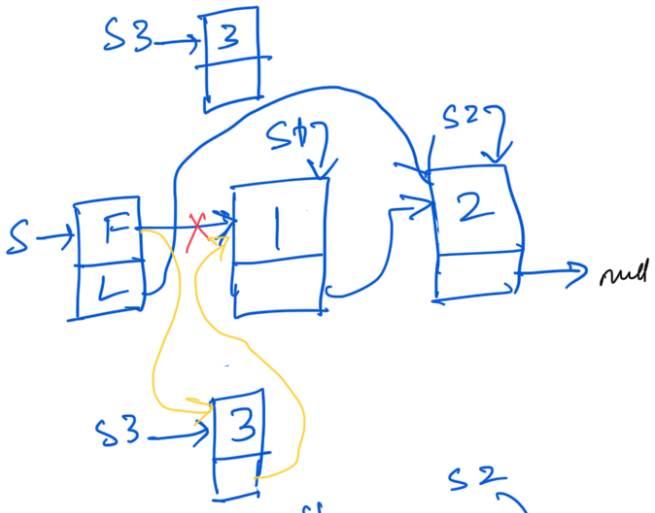
s3 = snode (3)



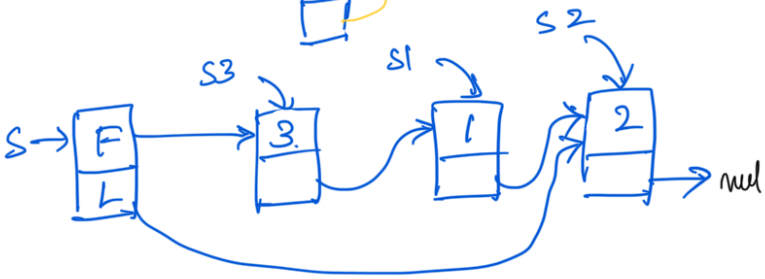
Prepend SList

Insert at head

s.prepend(s3)

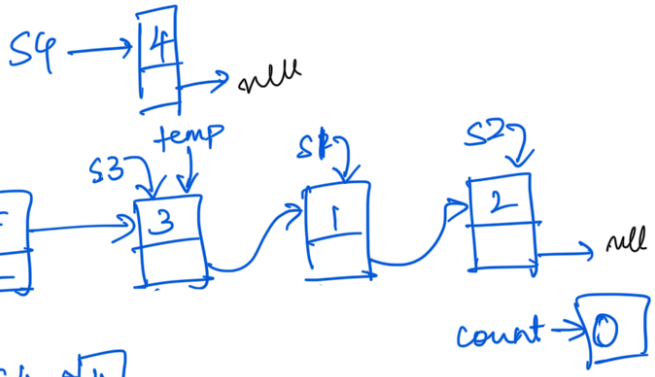


s4 = snode(4)

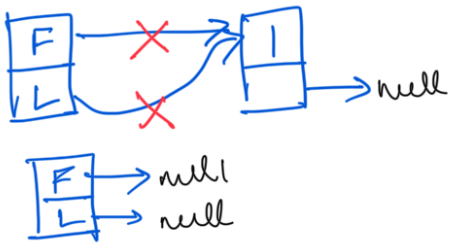


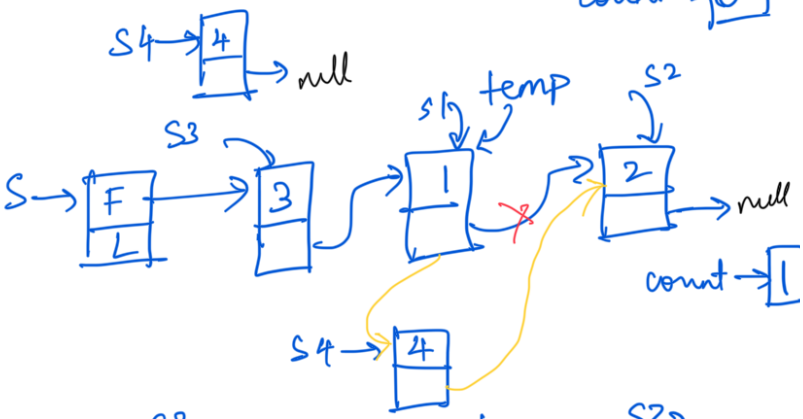
Insert at any position

s.insertAt(1,4)



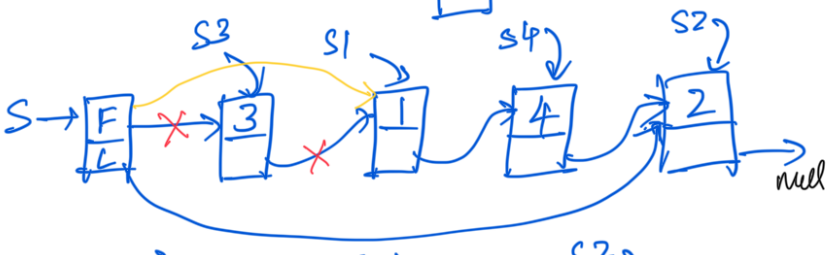
Delete w/ 1 element





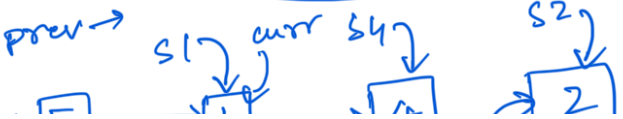
Delete at head

s.deleteHead()



Delete at tail

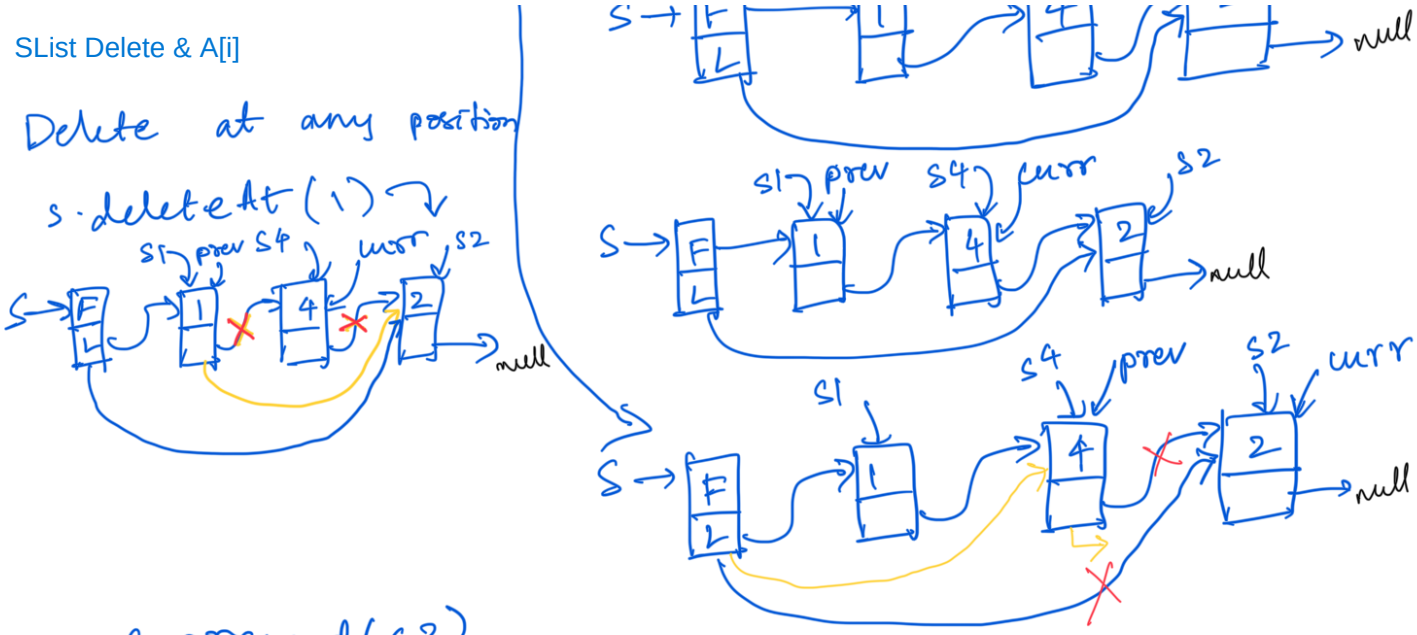
s.deleteTail()



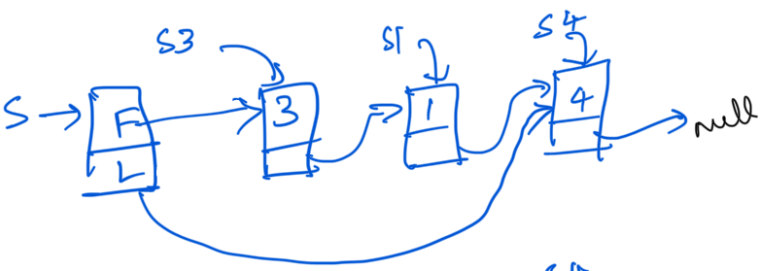
SList Delete & A[i]

Delete at any position

s.deleteAt(1)



s.prepend(s3)

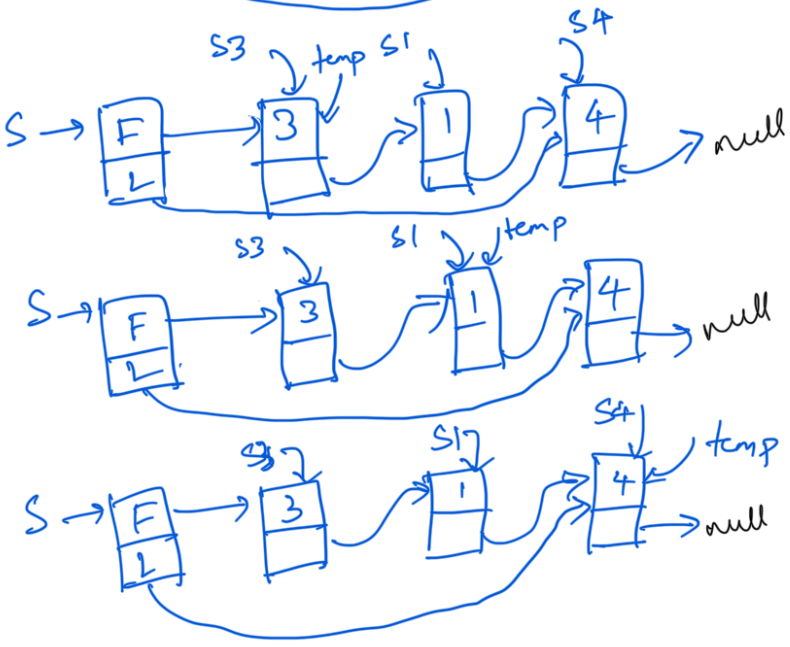


SList find

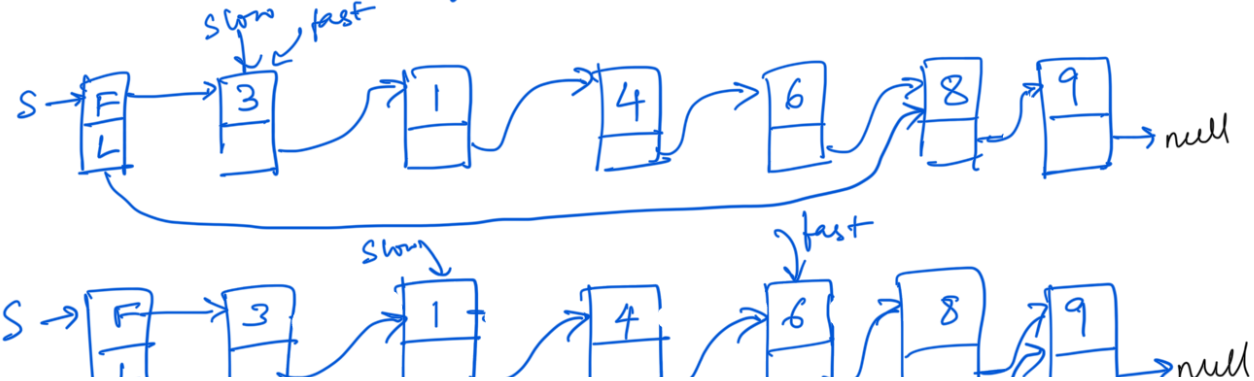
Find an element

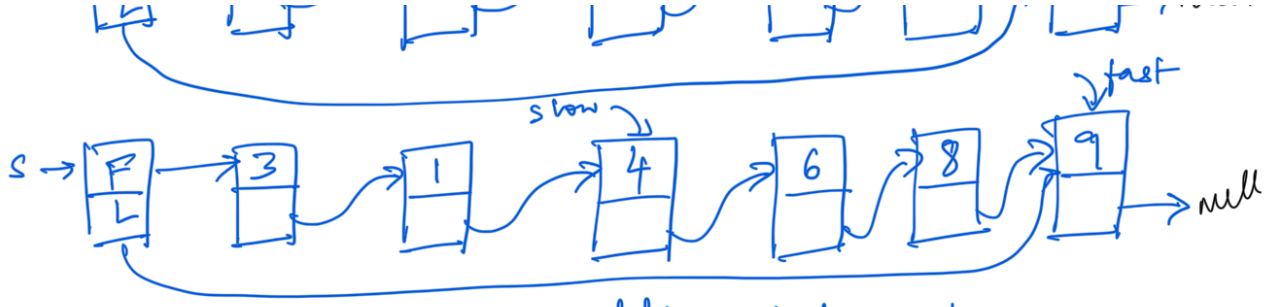
s.find(4)

//True



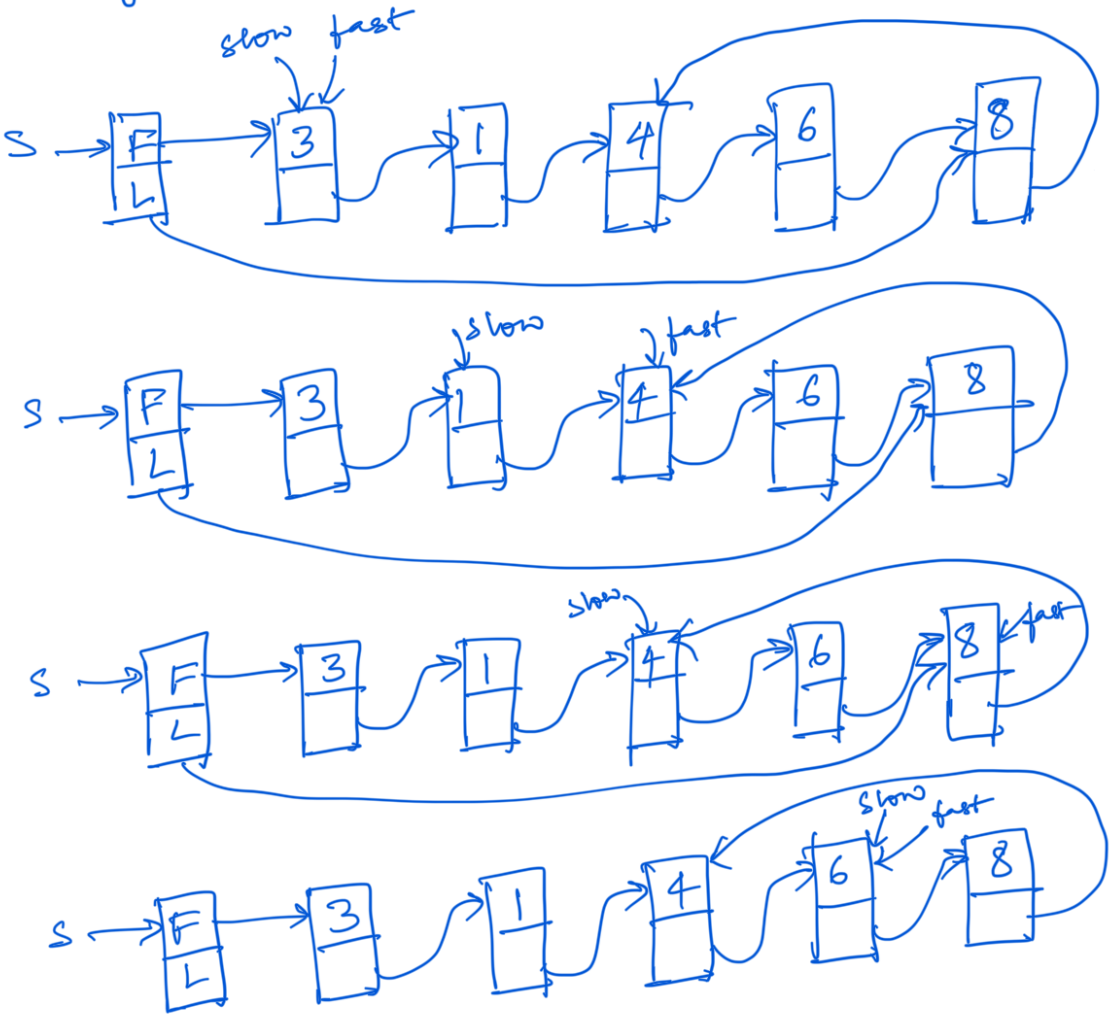
Find the middle of the slist





∴ Middle Node = 4

Finding a loop in a slist



Since slow & fast converge, there is a loop

Hash - Dict

Hash & Dictionary

- If hash or key is between 0-100 & there are ~100 elements use a python list, where key = index

Need for Hash

- Say we need to represent more elements, its not a good idea to use python list since we

a good ... Python list, ... might not need that much quantity or even continuous memory

Need for hash function

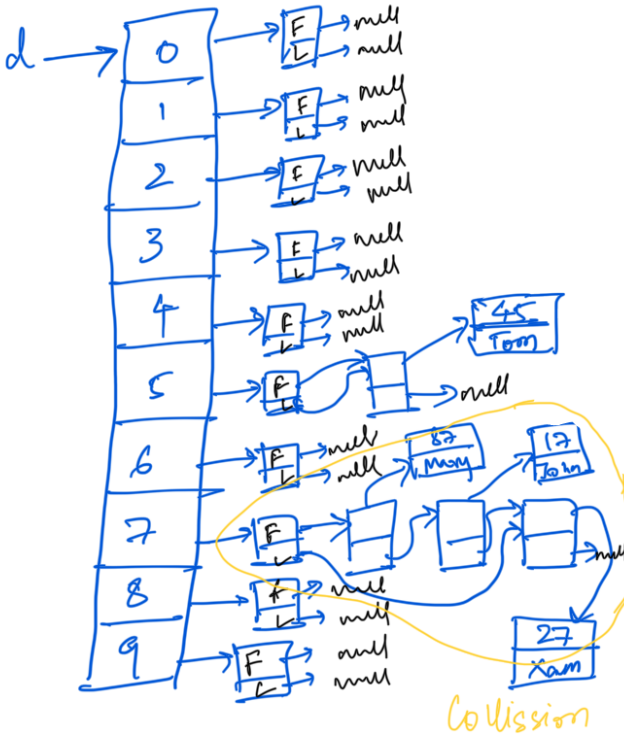
- Hash is a function, where we uniquely want to find an object's key, it should ensure, collision is as small as possible

Ex: Student(Id, Name)

hash(Id) = Id % 10

Insert, find delete O(1)

```
s1 = Student( 27, Mary)
s2 = Student( 45, Tom)
s3 = Student( 17, John)
s4 = Student (27, Xam)

d = MyDict ()
d.add (s1)
d.add (s2)
d.add (s3)
d.add (s4)
d.find (s1) //true
```



Here for 1mi objects, collision ≤ 8, ∴ to find an element in dict, Time complexity = $\Theta(1)$

∴ Hash functions must be:

- simple & fast to compute
- map equal elements to the same index
- map diff. elements to diff. indexes
- have keys distributed evenly among indexes

```
def statistics (self)
    -> "list of min,max":
        (0, ≤ 8)
```

ensure that hash max collisions are ≤ 8

Time Complexity, Space Complexity	Accessing front / top element	Accessing rear element	Accessing center element	Inserting at front / top position	Inserting at rear position	Inserting anything / at any position	Length	Finding any element	Finding min element	Finding max element	Deleting from front / top position	Deleting from rear position	Deleting anything / from any position	Printing DS (__str__)
Python List	Theta (1), Theta (1)	Theta (1), Theta (1)	Theta (1), Theta (1)	Theta (n), Amortized O (1)	Amortized O (1), Amortized O (1)	O (n), Amortized O (1)	Theta (1), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)
Stack using Python List	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Amortized O (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	O (n), O (n)	O (n), O (n)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (n), Theta (n)
Queue	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	Theta (n), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)
Circular Queue using Python List	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	Theta (n), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)
Min Stack using Stack	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (n), Theta (n)
Max Stack using Stack	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	O (n), O (n)	Theta (n), Theta (n)	Theta (1), Theta (1)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (n), Theta (n)
Stack using Circular Queue	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)
Queue using Stack	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	O (n), O (n)	O (n), O (n)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (n), Theta (n)
Slist (first & last pointer)	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)
HashSet	-	-	-	-	-	Theta (1), Theta (1)	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	-	-	Theta (1), Theta (1)	Theta (n), Theta (1)
HashMap (Dict)	-	-	-	-	-	Theta (1), Theta (1)	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	-	-	Theta (1), Theta (1)	Theta (n), Theta (1)