Today's content

(i) Doubly linked list basics

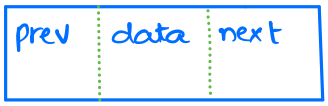
(ii) LRU cache

(iii) Clone linked list.

Doubly linked lists.

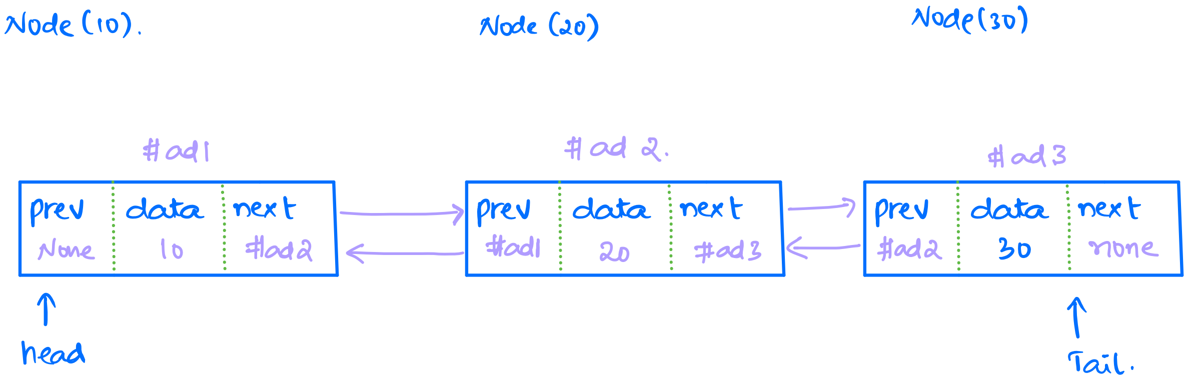
SLL.

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



DLL.

```
class  Node :
        def __init__(self, data):
                self.data = data
                self.next = None.
                Self. prev = None.
```

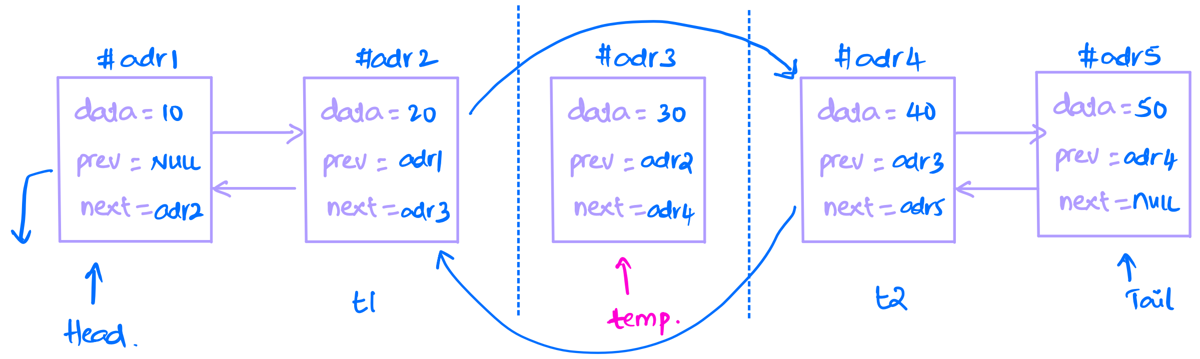


1Q. Delete a given node from DLL.

Note: 1) Node reference is given

2) Given node is not a head/tail node.

Ex1: Delete #adr3.

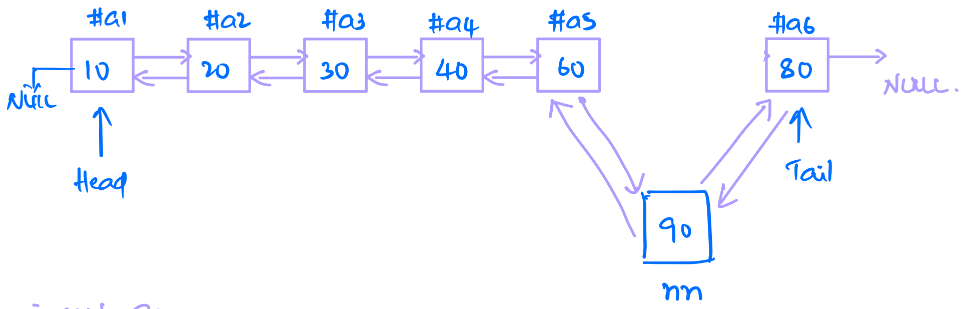


```
def deleteNode(temp)
        t1 = temp.prev
        t2 = temp.next

        t1.next = t2
        t2.prev = t1

        temp.next = None
        temp.prev = None.
```

TC : O(1).

SC : O(1).

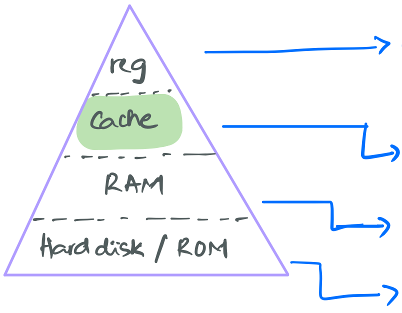
29. Insert a new node just before the tail of a DLL.



Ex: insert 90.

```
def   insert Back ( nn, tail )
{
        nn.next= tail.
        nn. prev = tail.prev.
        tail.prev= nn
         nn.prev.next= nn.
 }
```

Memory hierarchy.



→ executing instructions in CPU currently.

↳ located as a separate chip on motherboard

→ To store entire apps while running.

→ OS | movies .

Cache operations.

(i) insert
(ii) delete
(iii) update

→ To implement these operations effectively,

LRU → least Recently Used cache.

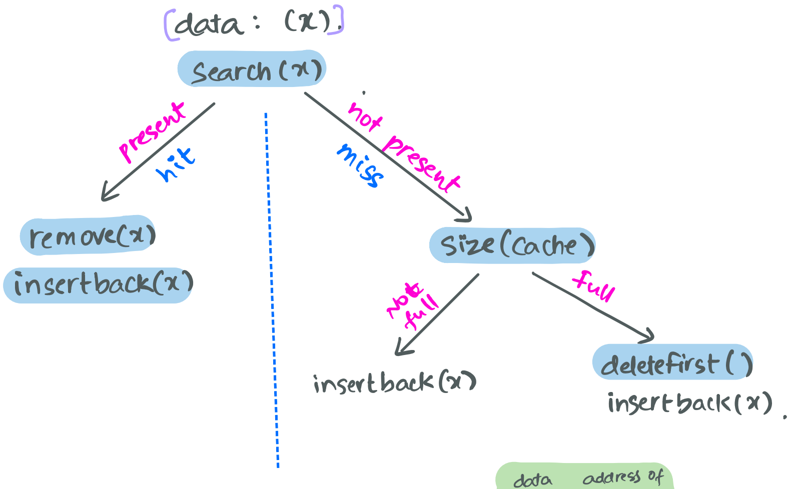
Numbers.

						✓ 10	14	update	update	15✓	8✓	
	✓	✓	✓	✓	✓	7✗	3✗	↑	↑	9✗	6✗	
Data:	7	3	9	2	6	10	14	2	10	15	8	14

limit for cache size : 5.

~~7~~	~~3~~	~~9~~	~~2~~	~~6~~	~~10~~	~~14~~	2	10	15	8	14

Flowchart : Working of cache.



data ↑ (key, value) ↑ address of node.

Operations	Arrays	Linked list	SLL + hashset	SLL + hashmap	DLL + Hashmap
Search (x)	O(N)	O(N)	O(1)	O(1)	O(1)
remove (x)	O(N)	O(N)	O(N)	O(N)	O(1)
insert back(x)	O(1)	O(1)	O(1)	O(1)	O(1)
deleteFirst(x)	O(N)	O(1)	O(1)	O(1)	O(1)

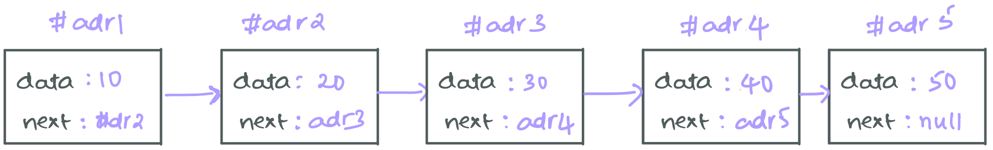
data: 10 20 30 40 50 40 - - .
size : 5.

Ex:

Hash map.

(10, adr1)
(20, adr2)
(30, adr3)
(40, adr4)
(50, adrs5)

Linked list.



Even with SLL + hashmap, we're not able to remove an ele in O(1) time => lead us to DLL.

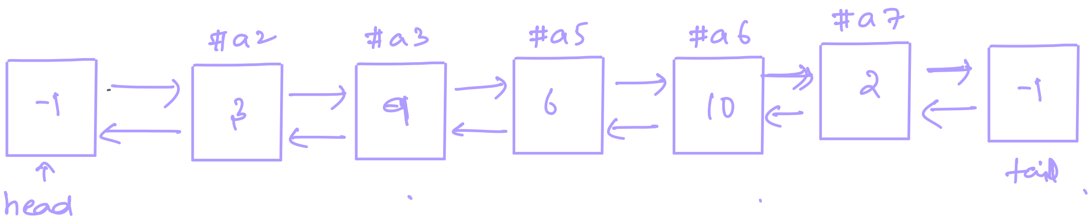
[Note : You can store prev node adr instead of current node in hashmap. (impl. is tricky).]

Data: 7 3 9 2 6 10 2 10 15 - - -

Cache size : 5

Hashmap : [(10,a6),(3, a2), (9,a3), (2,a7), (6,a5).].

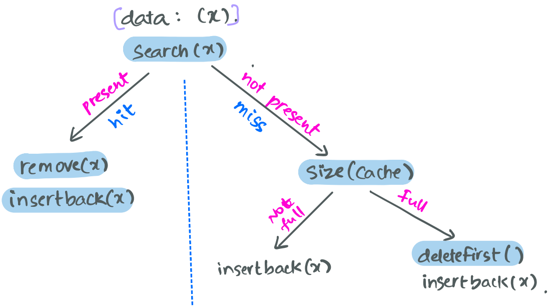
DLL:



Code of LRU. // (hm & head are available).

```
def LRU( x, limit)
{
    if(x in hm)
        delete Node( hm[x] ) // 1st question
        temp = Node(x)
        insertNode(temp ,tail)   // 2nd question
        hm[x] = temp // update address for x .

    else
        if (hm.size() == limit)
            temp = head.next
            hm.remove( temp.data)
            deleteNode (temp) // 1st q.
        temp = Node (x)
        insertNode (temp ,tail) // 2nd q .
        hm[x] = temp .
}
```

```
class Node :
    def __init__(self, data):
        self.data = data
        self.next = None .
        self. prev = None.
```

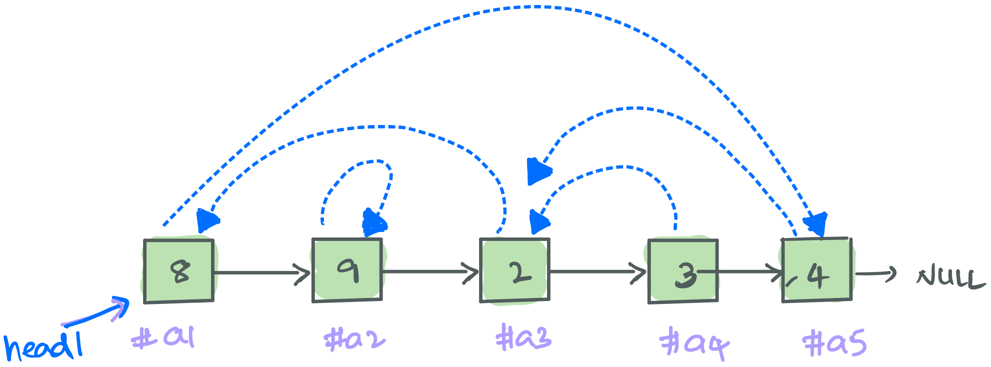


38. Given a linked list, where the structure of each node is given below.

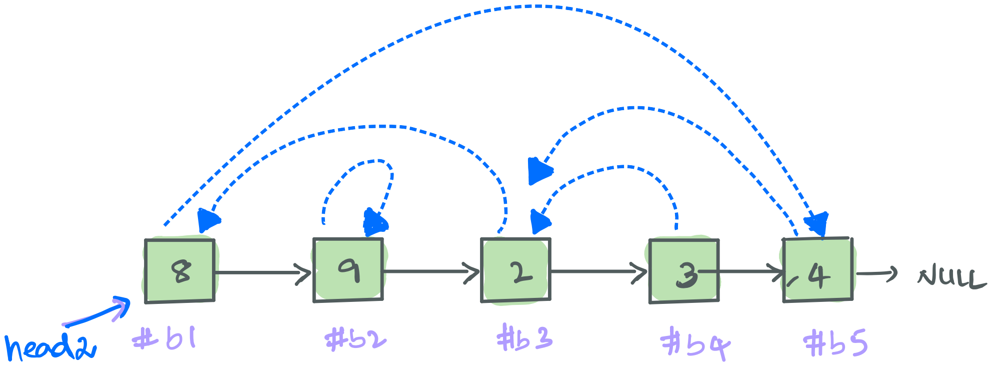
Create and return a clone of it, SC : (1).

```
class Node :
    def __init__(self, data):
        self.data = data
        next = None      // pointing to next node
        rand = None      // pointing to any node in LL.
```

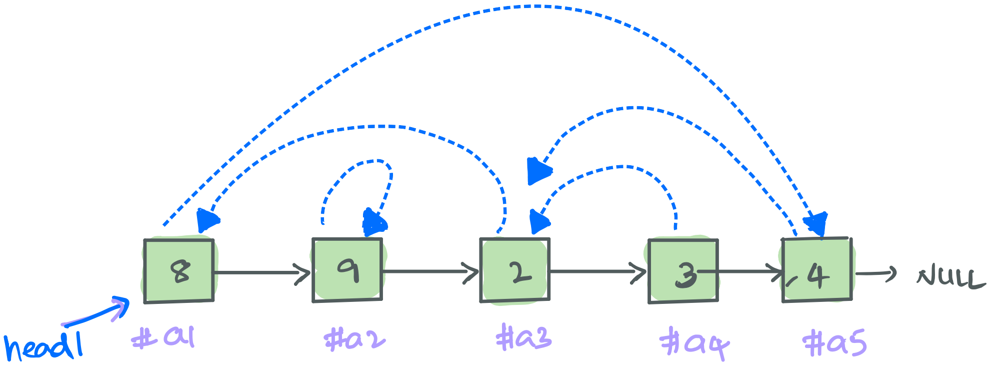
Exs:

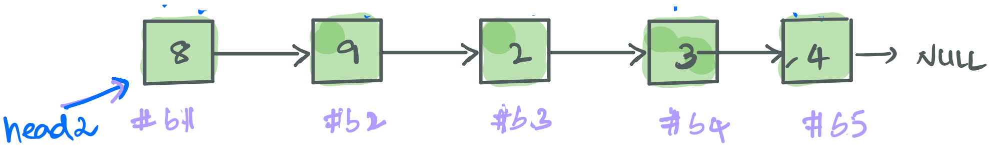


Node
    data
    next
    random..



Idea:





Try to implement.