Todays Content:

a) Double linked list basics

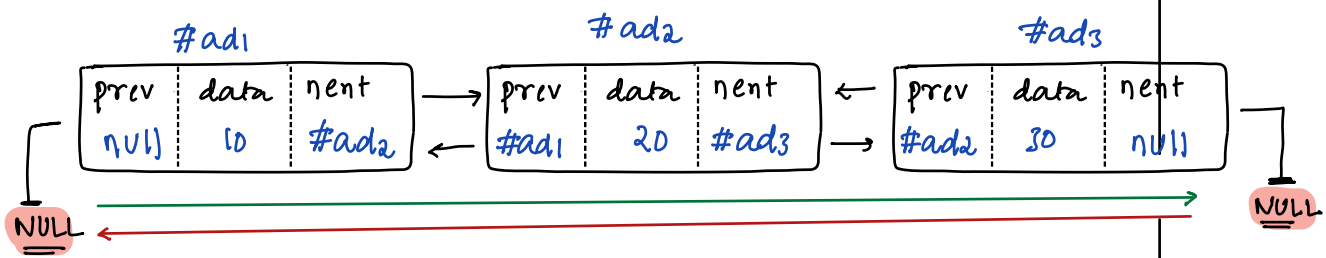
b) LRU Cache

c) Clone linked list

Double linked list:


```
class Node{
    int data
    Node nent
    Node prev
    Node(int n){
        data = n
        nent = null
        prev = null
    }
}
```

// obj references can hold address of node objects



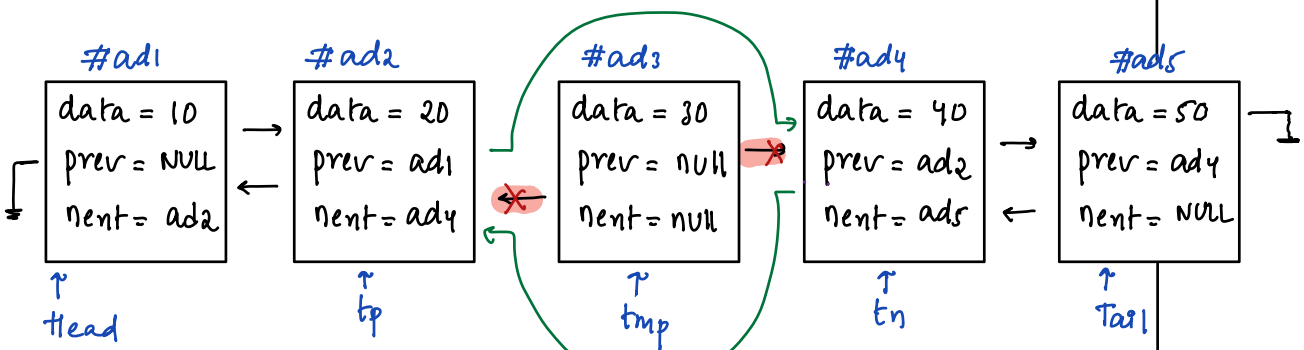
Obs: Traversing in both directions is possible

Q) Delete a given node from DLL

Note: Node reference is given

Note2: Given Node is not a head/tail node

Ex1: Delete #ad3 // address of node



Note: linked list is not a null

```
void DeleteNode(Node temp){   TC: O(1)
    Node tp = tmp.prev        // It is not head/not tail
    Node tn = tmp.next
    tp.next = tn
    tn.prev = tp
    tmp.prev = null
    tmp.next = null
    fru(tmp) // de-allocate memory assigned to tmp node
             // no need to worry in java/python
}
```

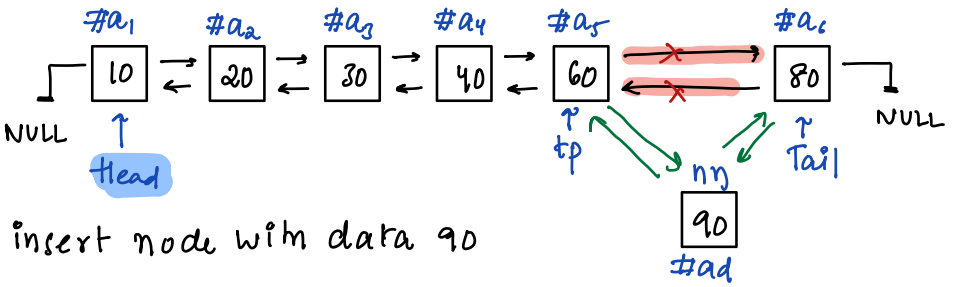
Obs: In a double linked list

To Delete a node current node address is sufficient

2Q) Insert a newnode Just before tail of a Double linked list

Note: Tail ref is given in input

Note: no: of nodes >= 2



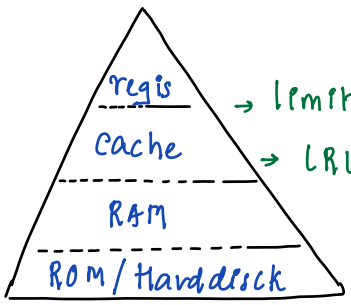
Ex: insert node with data 90

```
void insertback (Node nn, Node tail){   TC: O(1) SC: O(1)
    Node tp = tail.prev
    tp.next = nn
    nn.prev = tp
    nn.next = tail
    tail.prev = nn
}
```

Memory hirarchy:

bot-up acces speed increases

funcodeschool.com → 1st time: 2sec
→ 2nd time: 0.2sec



→ limited memory storage

→ LRU ⇒ least recently used

Data is stored inside cache, based on recent usage.

Top-down memory capacity inc

Data: 7 3 9 2 6 10 14 2 10 15 8 14

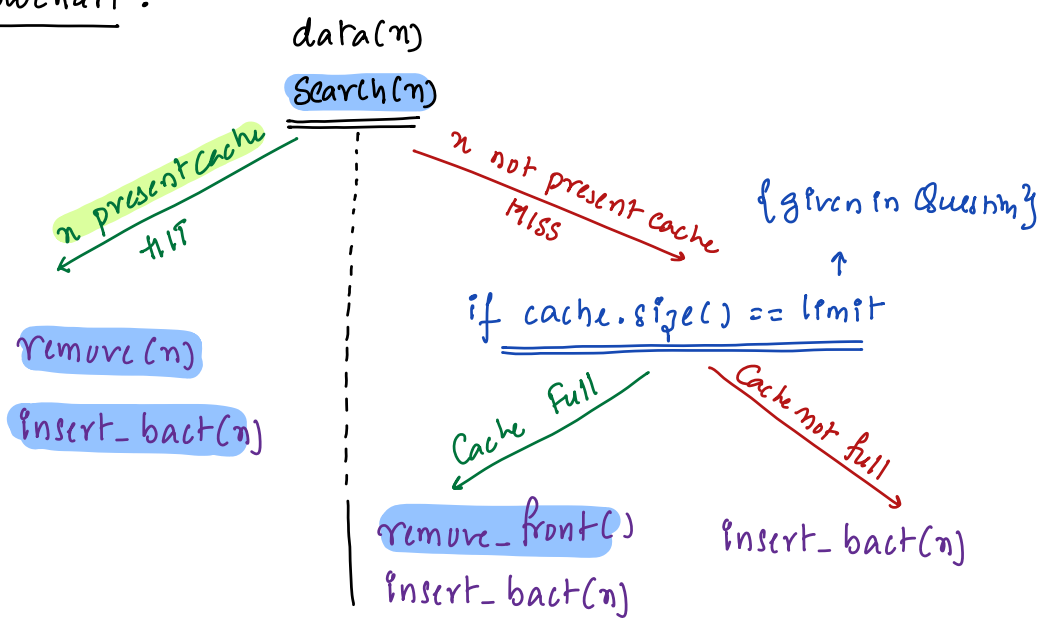
7x 3x 9x 6x

10✓ 14✓ 15✓ 8✓

front → back

Cache:5 ~~7~~ ~~3~~ ~~9~~ ~~2~~ ~~6~~ ~~10~~ ~~14~~ 2 10 15 8 14

Flowchart:



operations	Queue:	Dynamic array	Single linked list
search(n)		O(N)	O(N)
remove(n)	not possib? In Queue we can only delet ele at front	O(N)	O(N)
insert back(n)		O(1)	O(1) : if we store tail
delete front()		O(N)	O(1):

operations	Single linked list + Hashset<int>	Single linked list + Hashmap<int, addres of node which contains data>
search(n)	O(1)	O(1)
remove(n)	iterate find ele & del : O(N)	iterate even though we know addres O(N)
insert back(n)	O(1)	O(1)
delete front()	O(1)	O(1)

e

Obs: Given node, address we can direcly in a DLL

operations	Double linked list + Hashmap<int, addres of node which contains data>
search(n)	O(1)
remove(n)	O(1)
insert back(n)	O(1) : if we know tail we can do this
delete front()	O(1)

// Just to reduce a lot of base conditions we are taking 2 dummy node head & tail, & then wont be changed.

Data: 7 3 9 2 6 10 14 2 10 15 8 14

7✗ 3✗

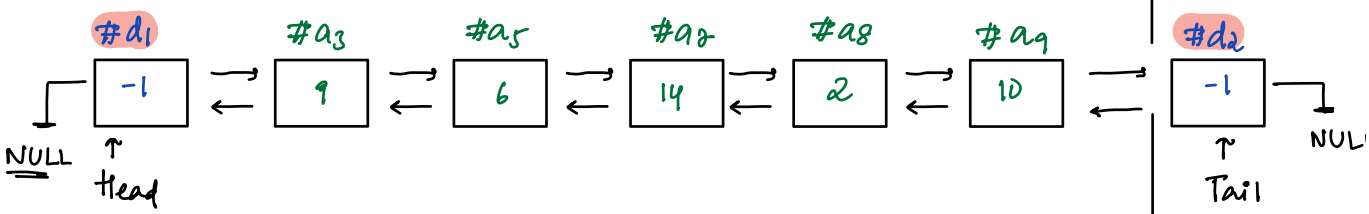
10✓ 14✓

TODO

Cache size = 5

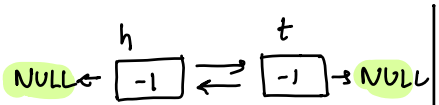
Hashmap<int, Node>:

{ <9, a_3> <2, a_8> <6, a_5> <10, a_9> <14, a_7> }



```
class Node{
    int data
    Node nent
    Node prev
    Node(int x){
        data = x
        nent = null
        prev = null
    }
}
```

Initializations:




```
Node h = new Node(-1)
Node t = new Node(-1)
h.nent = t
t.prev = h
hashmap<int, Node> hm;
```

```
------ LRU(int x, int limit){ 10:45 → 10:55pm
```

↳ Cache size limit, given in input

```
    if(hm.search(x) == True){ // x is present in cache
        Node t = hm[x] // getting address of x
        DeleteNode(t)
        Node nn = new Node(x) // creating new node
        Insert back(nn, tail) // insert just befre tail
        hm[x] = nn // updating addrs of x in hm
    }
    else{ // x is not present in cache
        if(hm.size() == limit){ // Cache reached limit
            Node t = h.nent
            hm.delete(t.data, t) // delete from hashmap
            DeleteNode(t)
        }
        Node nn = new Node(x) // creating new node
        Insert back(nn, tail) // insert just befre tail
        hm.insert(x, nn) // inserting in hashmap
    }
}
```

cache size

remove front ele

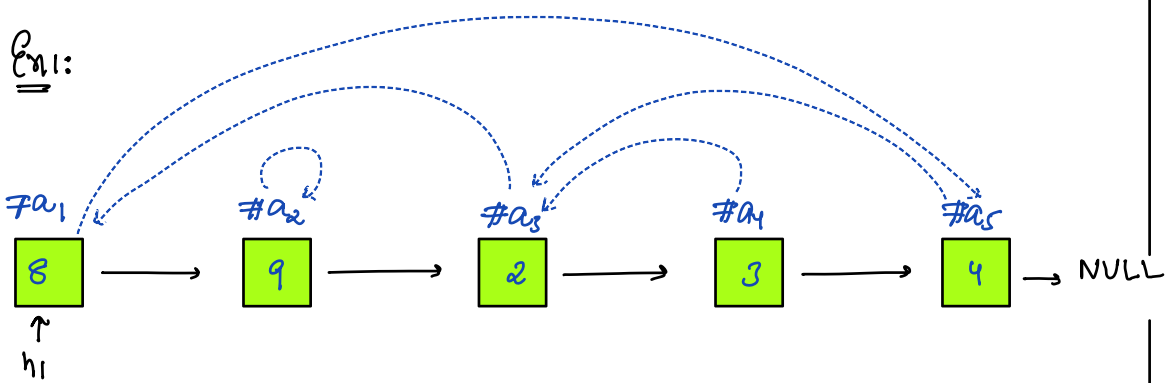
```
void DeleteNode(Node temp){   TC: O(1) SC: O(1)
                              ↳ Neither head / nor tail
    Node tp = tmp.prev
    Node tn = tmp.next
    tp.next = tn
    tn.prev = tp
    tmp.prev = null
    tmp.next = null
    free(tmp) // de-allocate memory assigned to tmp node
              ↳ no need to worry in java/python
}

void InsertBack(Node nn, Node tail){   TC: O(1) SC: O(1)
    Node tp = tail.prev      // Insert node nn before tail
    tp.next = nn
    nn.prev = tp
    nn.next = tail
    tail.prev = nn
}
```

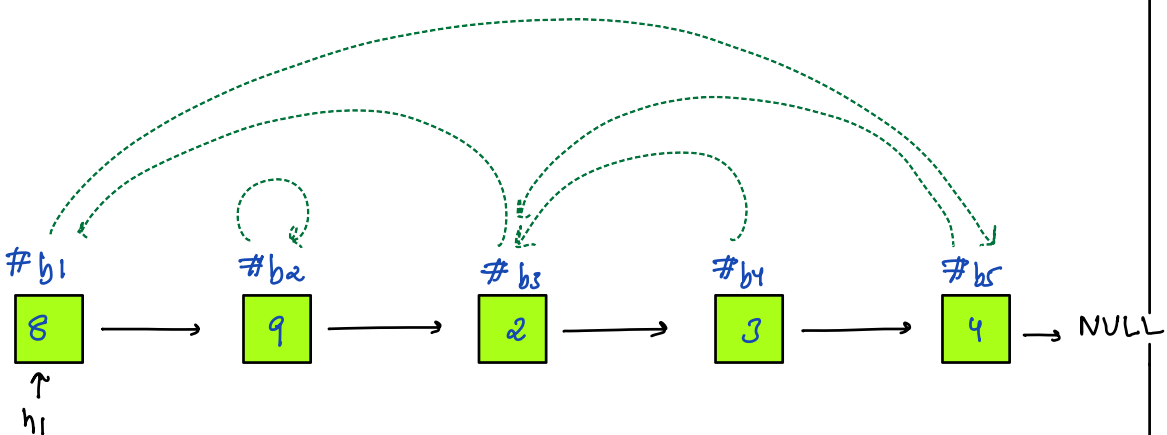
Clone linked list:

```
class Node {
    int data;
    Node next; // pointing to next node
    Node rand; // pointing to any one node in linked list
               // Note: rand is not null
}
```

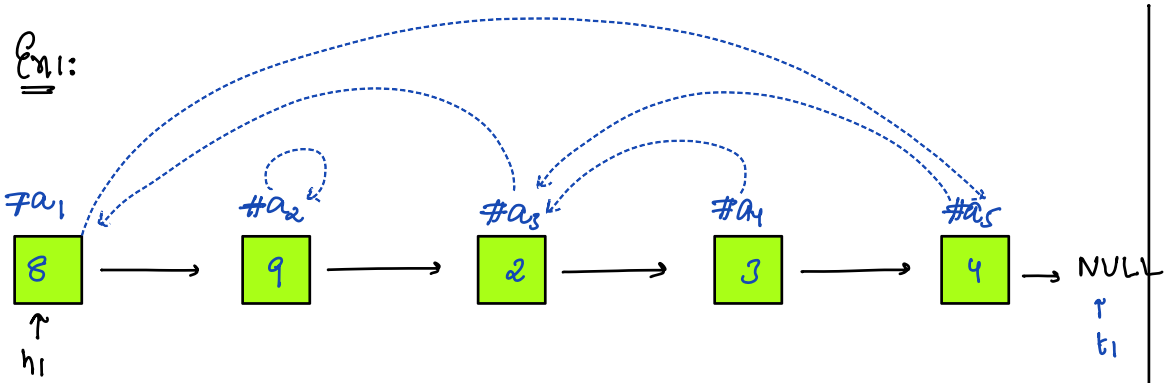
Given a linked list, Create & return head node of **Cloned linked list?**

Ex1:





Eg1:



HashMap<Node, Node> hm : // mapping address between original & clone

$a_1 : b_1$
$a_2 : b_2$
$a_3 : b_3$
$a_4 : b_4$
$a_5 : b_5$

Eg1:

a_3 $a_3.rand = a_1$

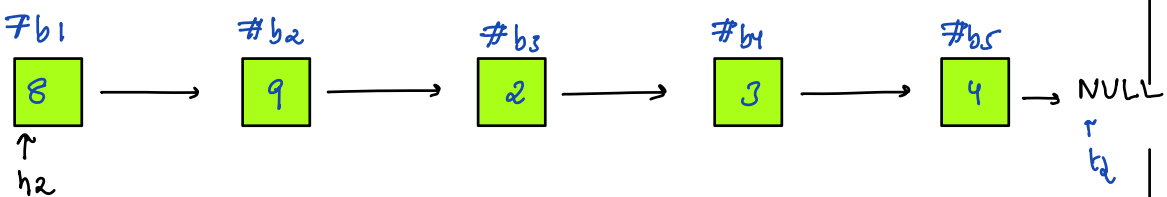
$hm[a_3].rand = hm[a_3.rand]$

$b_3.rand = b_1$

a_1 $a_1.rand = a_5$

$hm[a_1].rand = hm[a_1.rand]$

$b_1.rand$ b_5



Node Clone(Node h1) { TC: O(N) O(N) → because of hashmap

Step1: Create Clone linked lists only using next pointer & say head node for that h_2 : TODO

Step2: Mapping nodes between original & clone

HashMap<Node, Node> hm

Node $t_1 = h_1$, $t_2 = h_2$

while (t_1 != null) {

    hm.insert(t_1, t_2)

    $t_1 = t_1$.next

    $t_2 = t_2$.next

}

// Step: 3 // Using hashmap fill random links

Node $t_1 = h_1$

while (t_1 != NULL) {

    hm[t_1].rand = hm[t_1.rand]

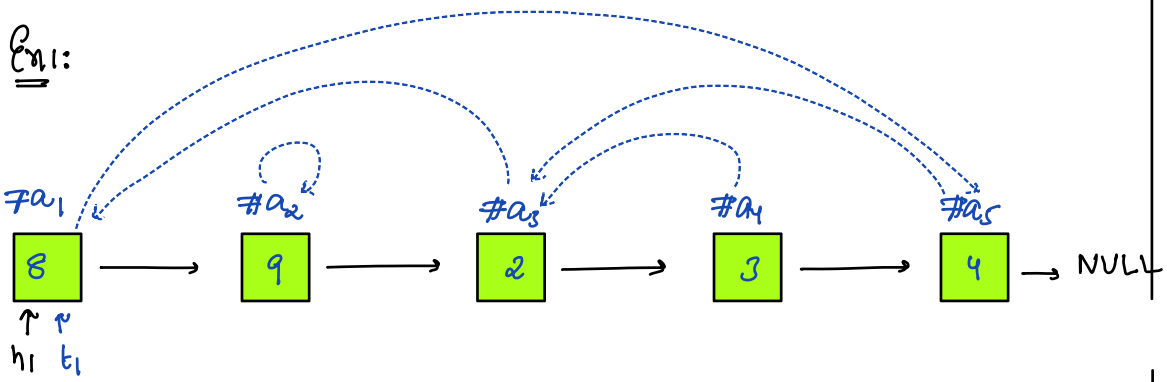
} $t_1 = t_1$.next

return h_2;

}

Note: No Extra Space : {45min Tuesday}

Ex1:



- Tuesday 1hr: 9pm → 10pm
- Wednesday: off / Solve linkedlist
- Thursday: Stacks : 1