Todays Content:

a) Double linked list basics

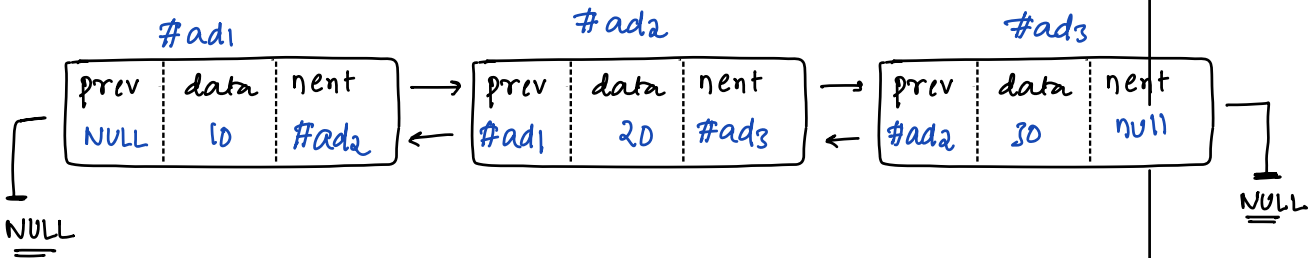
b) LRU Cache

c) Clone linked list

{ Revin Stacks }

Double linked list:


```
class Node{
    int data
    Node nent   // obj references can
    Node prev   hold address of node objects
    Node(int n){
        data = n
        nent = null
        prev = null
    }
}
```

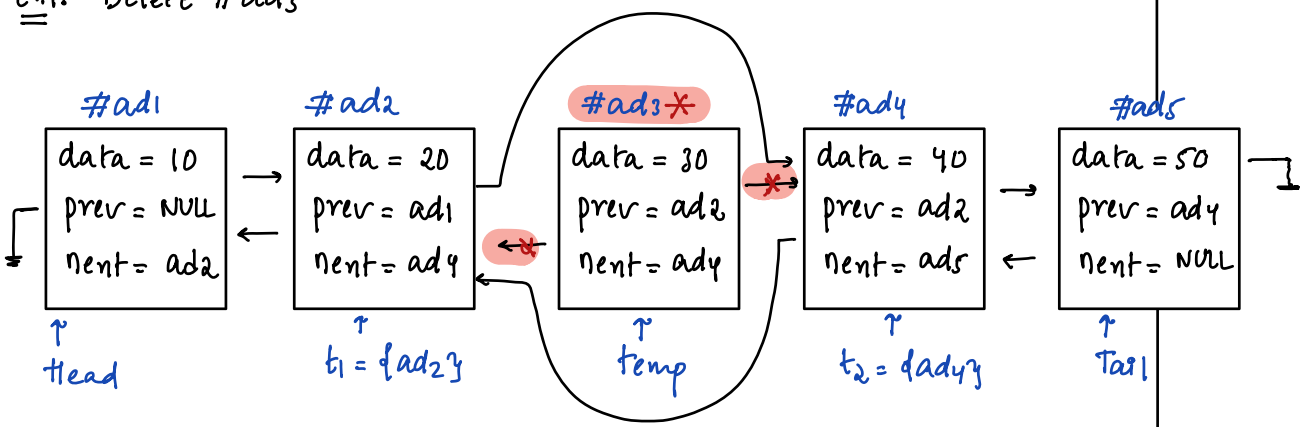


Q) Delete a given node from DLL

Note: Node reference/add is given

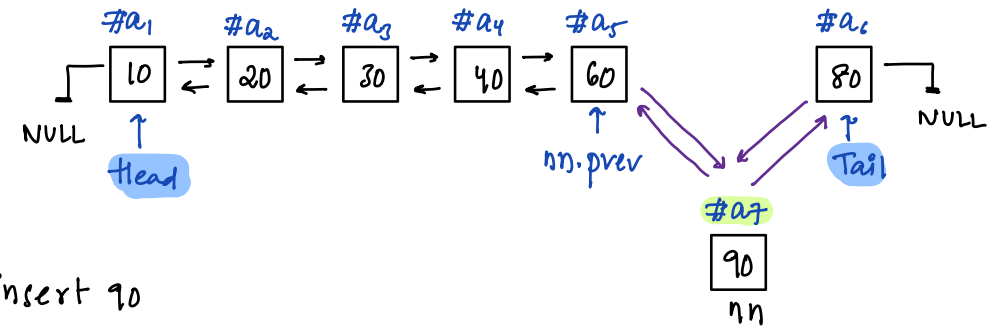
Note2: Given Node is not a head/tail node

Ex1: Delete #ad3




```
void DeleteNode(Node temp){   TC: O(1) SC: O(1)
                              ↳ Neither head / nor tail
    Node t1 = temp.prev
    Node t2 = temp.nent
    t1.nent = t2
    t2.prev = t1
    temp.prev = null
    temp.nent = null
    free(temp) // de-allocate memory
}
```

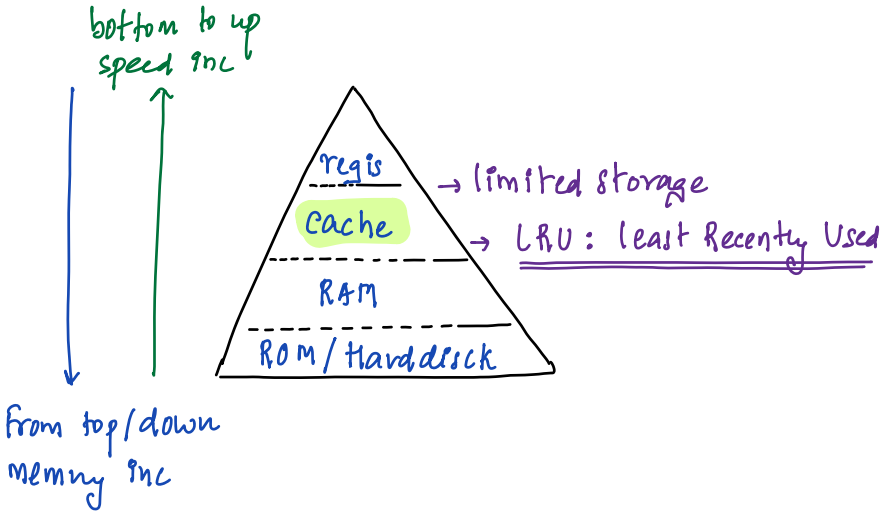
2Q) Insert a newnode Just before tail of a Double linked list



Eg: insert 90

```
void Insertback (Node nn, Node tail){ TC:O(1) SC:O(1)
                                    // Insert node nn before tail
    Node nn = new Node(x) // skip it
    nn.next = Tail
    nn.prev = Tail.prev
    Tail.pre = nn
    nn.prev.next = nn
}
```

Memory hirarchy:



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

7x 5x 2x 10x 9x 6x 14x

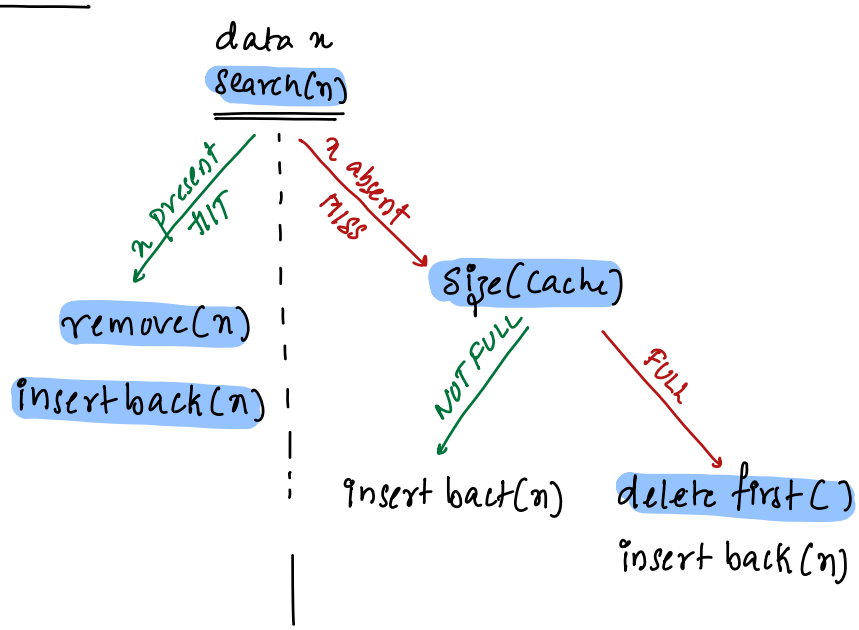
10✓ 14✓ 2✓ 10✓ 15✓ 8✓ 14✓

Cache:5

: go from left → right : most recently used

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

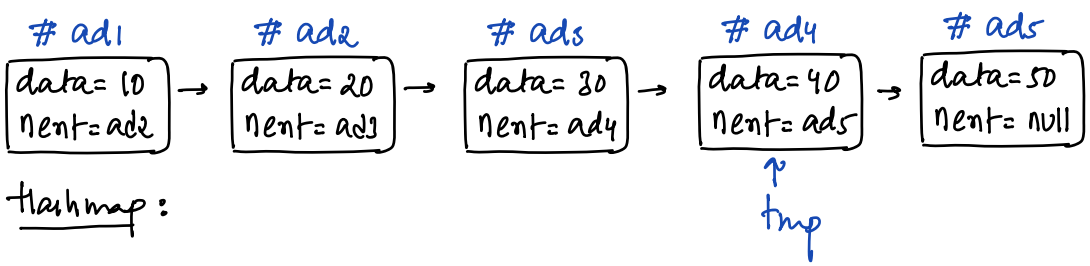
Flowchart:



operations	List	SLL	SLL + hashset<int>
search(x)	O(N)	O(N)	O(1)
remove(x)	O(N)	O(N)	O(N) ? iterate from start & delete node which contains x
insert back(x)	O(1)	O(1): tail is given	O(1)
delete front()	O(N)	O(1)	O(1)

operations	SLL + hashmap<int, Node> : value & addres of node	value & prev node addres
search(x)	O(1)	O(1) Implementation is slightly convulated
remove(x)	O(N)	O(1) better implementation
insert back(x)	O(1)	O(1)
delete front()	O(1)	O(1)

Eg:



hashmap:

10, ad1
20, ad2
30 ad3
40 ad4
50 ad5

operations:

: search(40): ✓ O(1)

: delete(40): → #ad4

↳ iterate & get prev node addres

↳ TC: O(N)

DLL + hashmap< int, Node > :
value & addres of node

operations	
search(n)	O(1)
remove(n)	O(1)
insert back(n)	O(1)
delete front()	O(1)

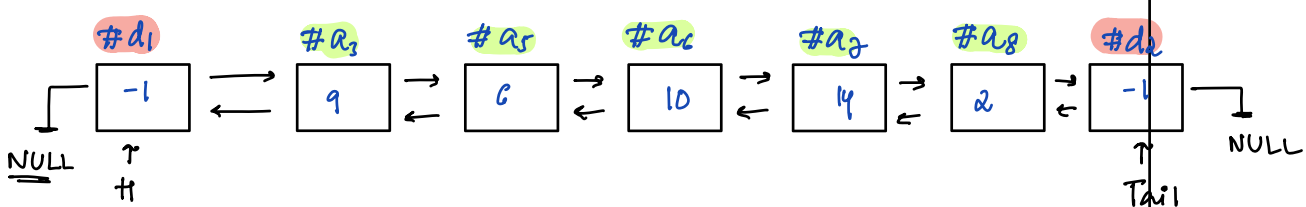
Note: Create a dummy head & tail nodes, to avoid edge cases for insert back / delete front operations

Data: 7 3 9 2 6 10 14 2 10 15 8 14
(7 3 9 2 6 ✓; 10 * #a_1; 14 * #a_2; 2 #a_4; 10 15 8 14 TODO)

Cache size = 5

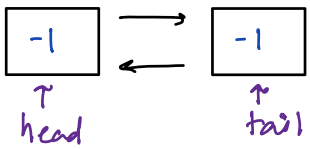
Hashmap< int, Node > :

{ {9, a_3} {6, a_5} {10, a_6} {14, a_7} {2, a_8} }



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

```
Node head = new Node(-1)
Node tail = new Node(-1)
head.nent = tail
tail.prev = head
```




```
HashMap<int, Node> hm
```

```
------ LRU(int x, int limit) { Single operation O(1)
    if(hm.search(x) == true){ // hit
        Node tmp = hm[x] // curr addres of x
        deleteNode(tmp) // deleting node x
        tmp = new Node(x);
        insertNode(tmp, tail) // Create a node with val=x
                              & insert before tail
        hm[x] = tmp // updating node for x
    }
    else{ // miss
        if(hm.size() == limit){ // deleting 1st node
                                as parameter
            Node tmp = head.nent
            hm.remove(tmp.data, tmp) // remove from map
            deletenode(tmp) // Delete node from linked list
        }
        Node tmp = new Node(x)
        insertNode(tmp, tail)
        hm[x] = tmp
    }
}
```

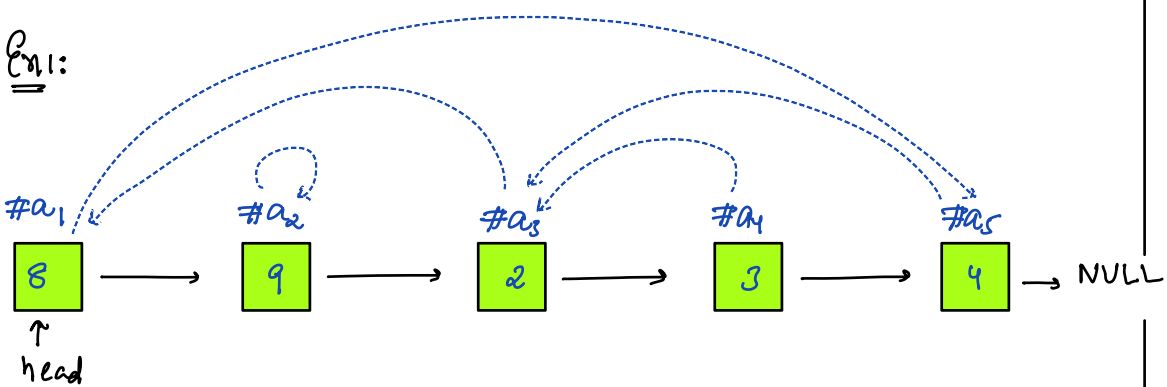
```
void DeleteNode(Node temp){   TC: O(1)  SC: O(1)
                                  ↳ Neither head / nor tail
    Node t1 = temp.prev
    Node t2 = temp.nent
    t1.nent = t2
    t2.prev = t1
    temp.prev = null
    temp.nent = null
    free(temp) // de-allocate memory
}

void Insert back (Node nn, Node tail){   TC: O(1)  SC: O(1)
                                  // Insert node nn before tail
    nn.nent = Tail
    nn.prev = Tail.prev
    Tail.pre = nn
    nn.prev.nent = nn
}
```

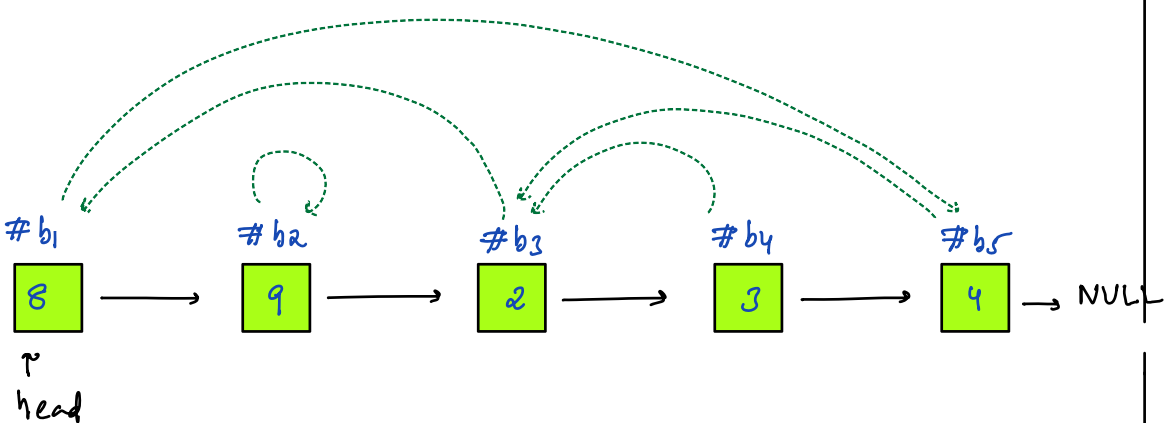
Clone linked list:

```
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
}
```

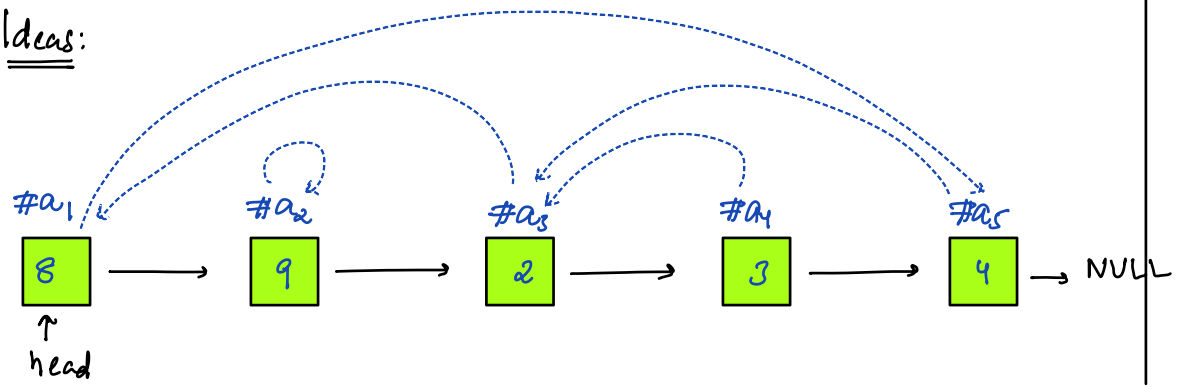
Given a linked list, Create & return clone of it, Expected SC: O(1)

Ex1:





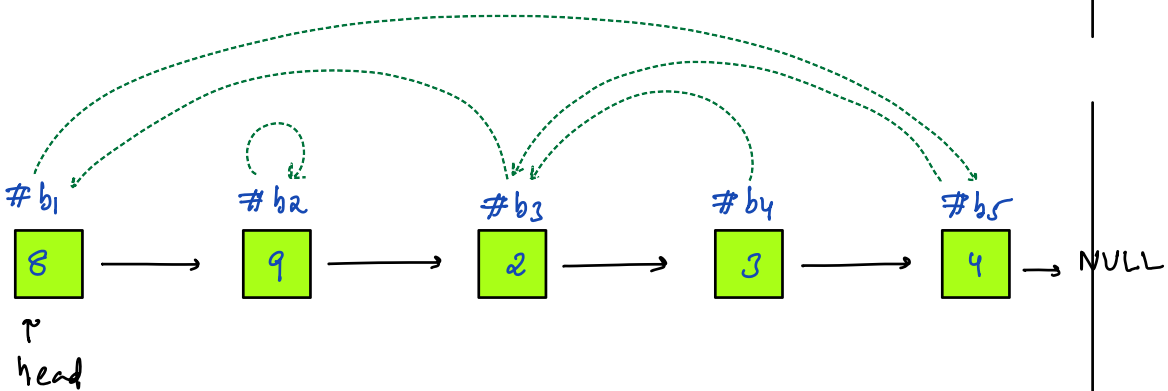
Ideas:



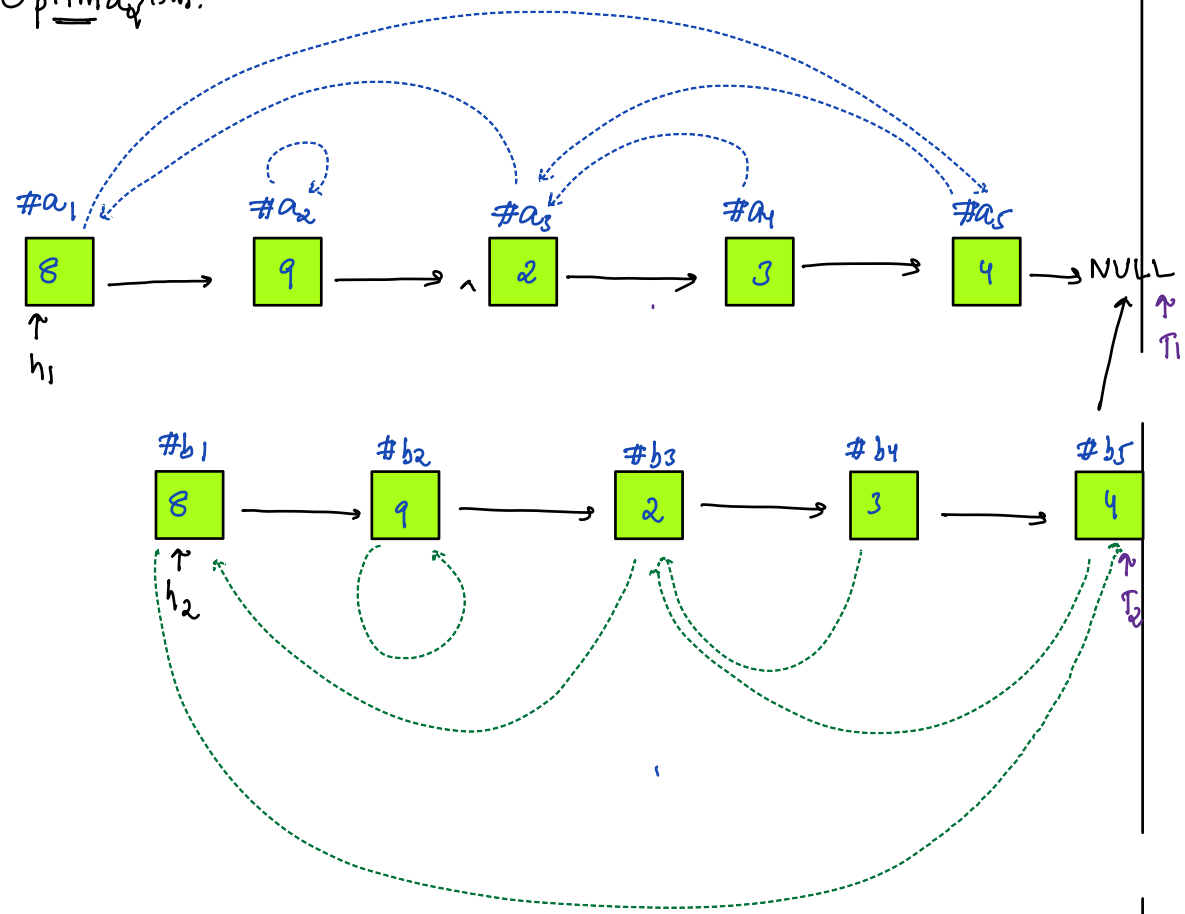
Idea: 1) Create clone linked list only using next pointer TC: O(N)

2) For every node in given LL, get its random node distance from start, & travel same distance from start in clone linked list & mark random for a node

TC: O(N²) SC: O(1)



Optimization:



```
Node Clone(Node h1) {                                   TC: O(N)  SC: O(1)

    Node t = h1
    while(t != NULL) {                                  // Step1: Create clone nodes
        Node nn = new Node(t.data)                      between original
        nn.next = t.next                                linked list
        t.next = nn
        t = nn.next
    }

    Node h2 = h1.next                                   // Step2: Fill rand value
    Node t1 = h1, t2 = h2                               for all nodes in clone
    while(t1 != NULL) {                                 linked list
        T2.rand = T1.rand.next
        T1 = T2.next // After last node T1 = null
        if(T1 == null) {break}
        T2 = T1.next
    }

    T1 = h1  T2 = h2                                    // Step:3 fill the next value for
    while(T1 != NULL) {                                 all nodes in clone linked list
        T1.next = T2.next
        T1 = T1.next // after last node T1 = null
        if(T1 == null) {break}
        T2.next = T1.next
        T2 = T2.next
    }

    return h2
}
```