Today's Content

(a) Doubly Linked List basics

(b) LRU Cache

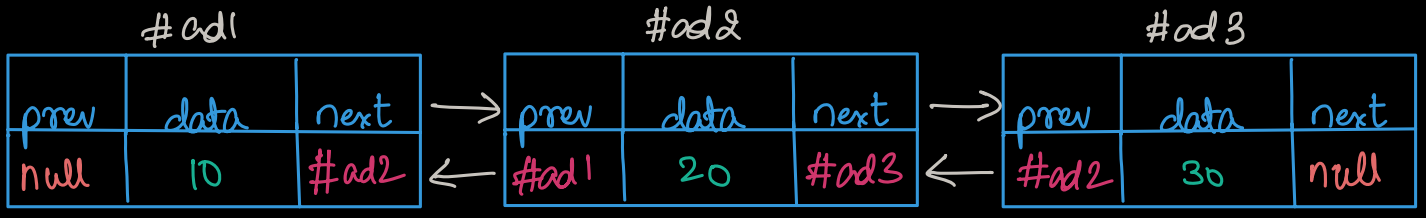
(c) Clone Linked List

Double Linked List

prev	data	next

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

// obj references can hold address of Node object.



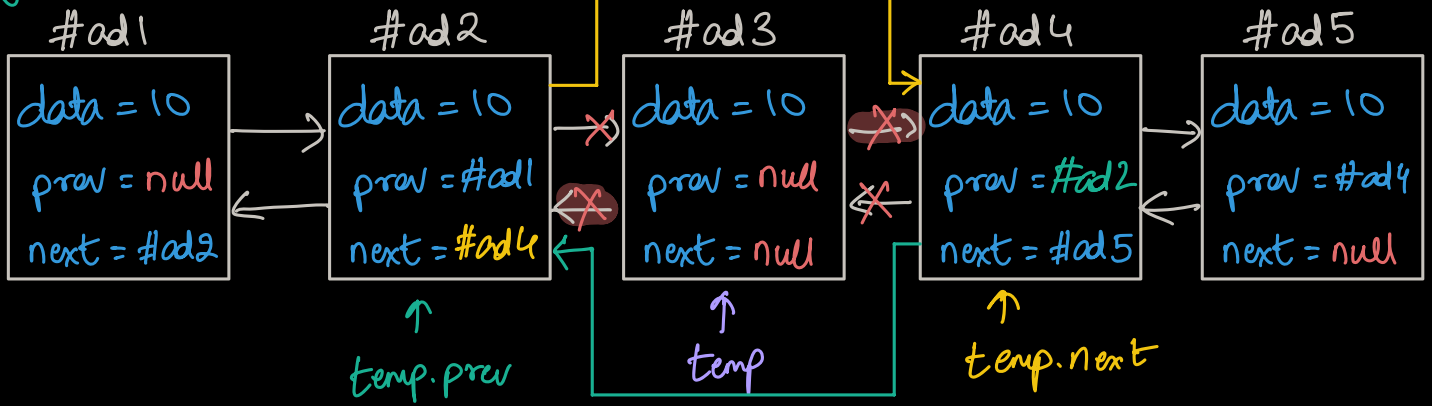
Bidirectional travel is possible.

Q1: Delete a node in DLL

Note: Node reference / address is given in DLL

Note 2: Given node is NOT a head / tail node.

Eg 1: Delete #ad3 ← address {say}



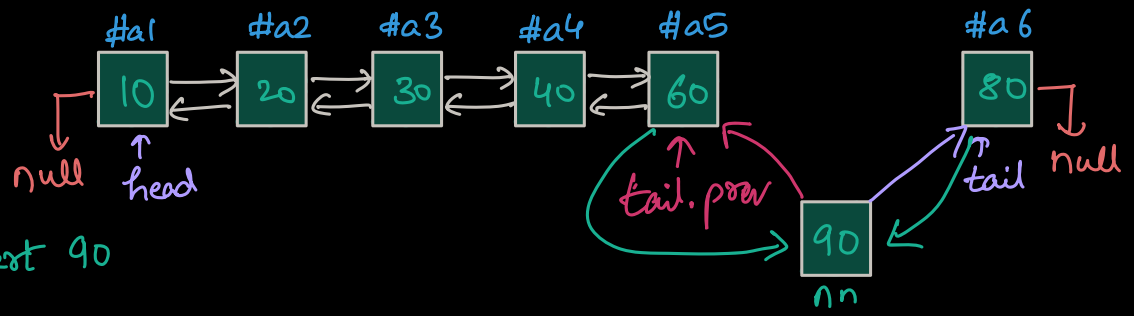

```
void DeleteNode (Node temp) {
    temp.prev.next = temp.next
    temp.next.prev = temp.prev
    temp.next = null
    temp.prev = null
    free (temp)   // deallocates the memory. or temp = null.
}
```

TC: O(1)

SC: O(1)

Obs: To delete a node in DLL, the current node address is sufficient.

Q2: Insert a new node just before tail node of a DLL.



Eg: insert 90

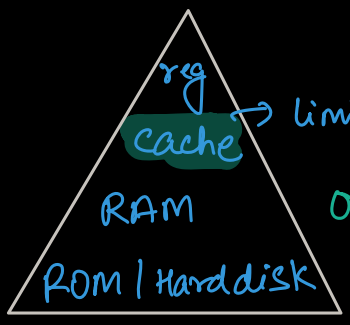
```
void insert back (Node nn, Node tail ){
    nn.next = tail
    nn.prev = tail.prev
    tail.prev.next = nn
    tail.prev = nn
}
```

TC: O(1)

SC: O(1)

Memory Hierarchy

Access speed inc.



→ limited memory size

Ordered in some fashion.

size of memory (↑)

LRU: Least Recently Used.

scaler.com → 1st → 5 seconds.
→ 2nd → faster than 5 secs

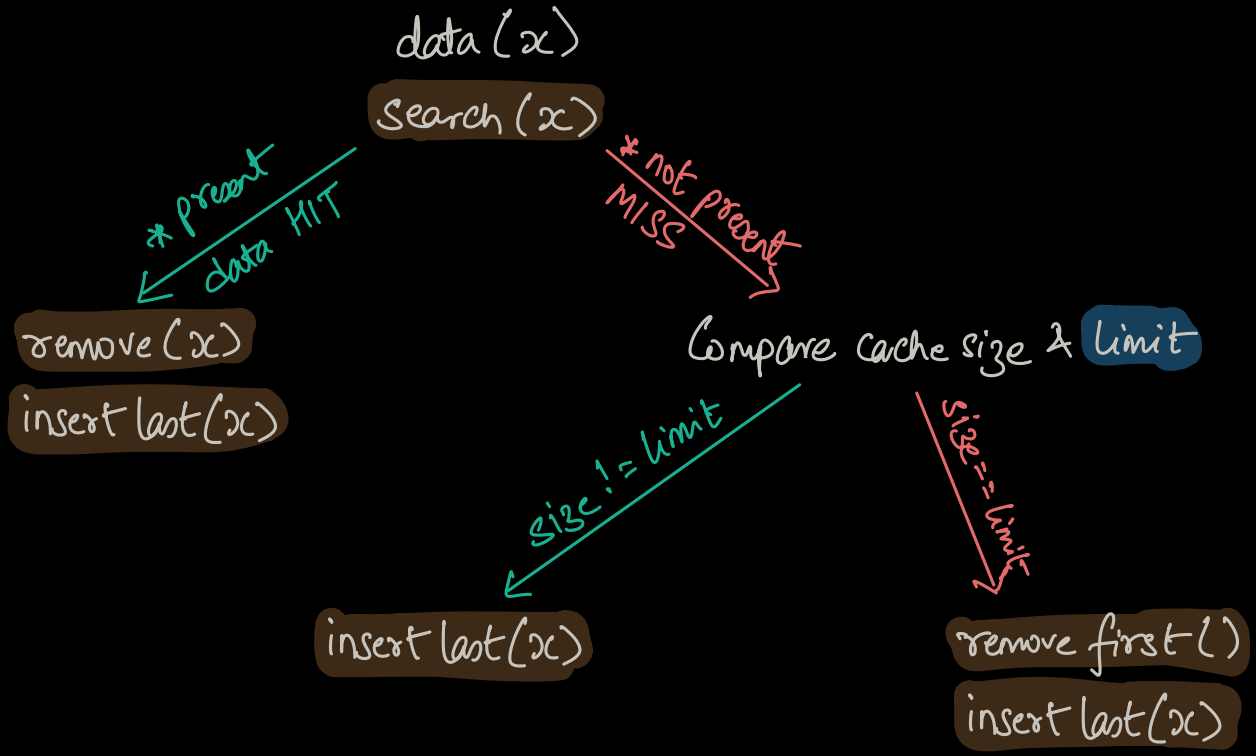
Data : 7 3 9 2 6 10 14 2 10 15 8 14

Cache size : 5

7x 3x 9x 6x

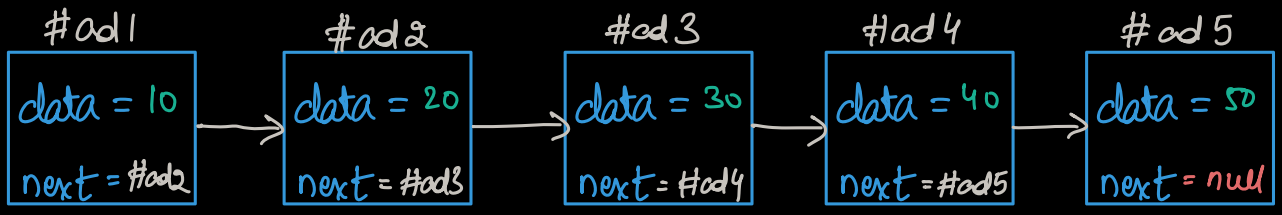
~~7~~	~~3~~	~~9~~	~~2~~	~~6~~	~~10~~	~~14~~	2	10	15	8	14

Flowchart :



operations	Dynamic array	Single LL	Single LL + Hashset<int>
Search(x)	O(n)	O(n)	O(1)
remove(x)	O(n)	O(n)	O(n)
insert back(x)	O(1)	O(1) : given tail node	O(1) : given tail node
delete first( )	O(n)	O(1)	O(1)

operations	Single LL + Hashmap<int, Node> (Node ↓ addr of data)
Search(x)	O(1)
remove(x)	O(n)
insert back(x)	O(1)
delete first(x)	O(1)



Single LL + Hashmap <data, node addr>

prev

temp

{ 10, #ad1
20, #ad2
30, #ad3
40, #ad4
50, #ad5 }

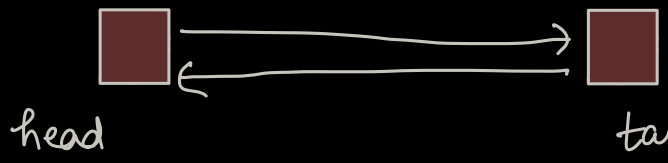
Operations

Search(40) : present! ⇒ O(1)

delete(40) : get address for 40 ⇒ #ad4 → temp.
Iterate LL & get prev of temp

operations	DLL + Hashmap <int, Node> value & addr. of node
search(x)	O(1)
remove(x)	O(1)
insert back(x)	O(1)
delete first(x)	O(1)

Note: To avoid a few edge cases we are using 2 dummy nodes.

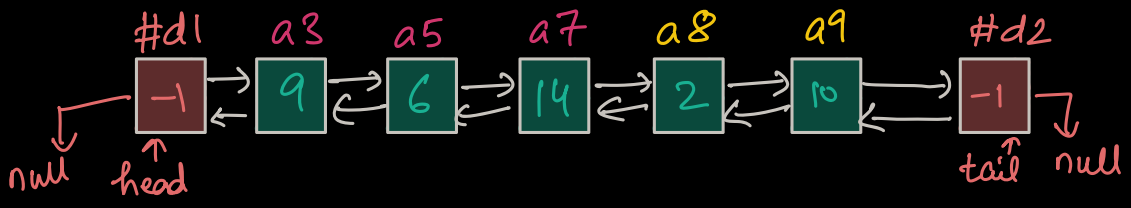


Data : 7 3 9 2 6 10 14 2 10 15 8 14 → Todo

Cache size : 5

Hashmap <int, Node> :

{ <7, a1> <3, a2> <9, a3> <2, a8> <6, a5> <10, a9> <14, a7> }



For existing node :

Delete & Add at Back

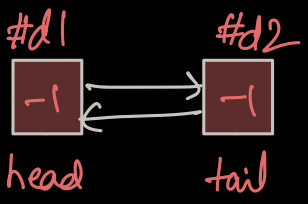
For a new node :

if space ⇒ Add at back

no space ⇒ Delete front
Add at back

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

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



Hashmap<int, Node> hm

Cache size limit is given

```
LRU(int x, int limit) {
    if(hm.search(x) == true) { // HIT
        Node temp = hm[x] // Current addr of x
        delete Node(temp)  // delete from DLL
        Node nn = new Node(x) // Create new node of x
        insert back(nn, tail)  // Insert at back of DLL
        hm[x] = nn  // Update its address in hm.
    }                  → hm.put(x, nn)
    else { // MISS
        if(hm.sizeOf() == limit) { // size is full
            Node temp = head.next
            hm.remove(temp.val)
            delete Node(temp)
        }
        Node nn = new Node(x)
        insert back(nn, tail)
        hm.insert({x, nn})
    }
}
```

A single operation will take O(1) time.

```
void DeleteNode (Node temp) {
    temp.prev.next = temp.next
    temp.next.prev = temp.prev
    temp.prev = null
    temp.next = null
    free (temp)      // deallocating the memory.
}
```

isolating the node

TC: O(1)
SC: O(1)

```
void insertBack (Node nn, Node tail) {
    nn.next = tail
    nn.prev = tail.prev
    tail.prev.next = nn
    tail.prev = nn
}
```

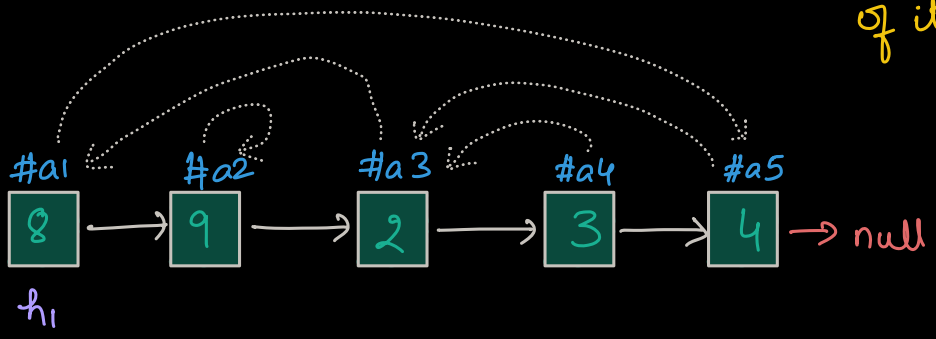
TC: O(1)
SC: O(1)

* Can be done with LinkedHashMap ← To Do

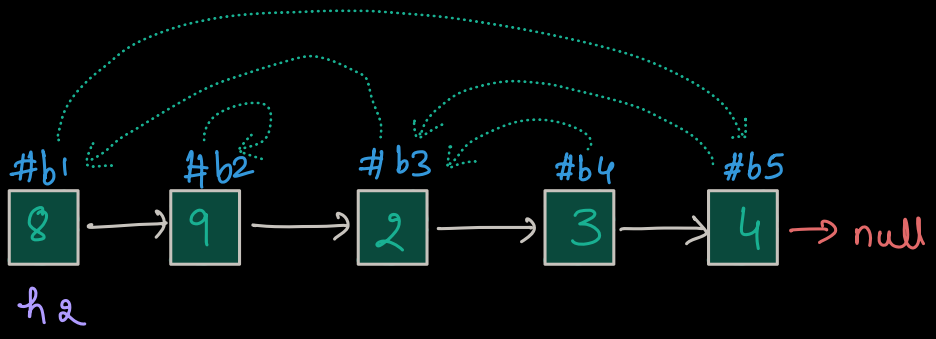
```
class Node {
    int data
    Node next
    Node random
}
```

Qn: Given a LL, which also has a random pointer. Create & return clone of it.

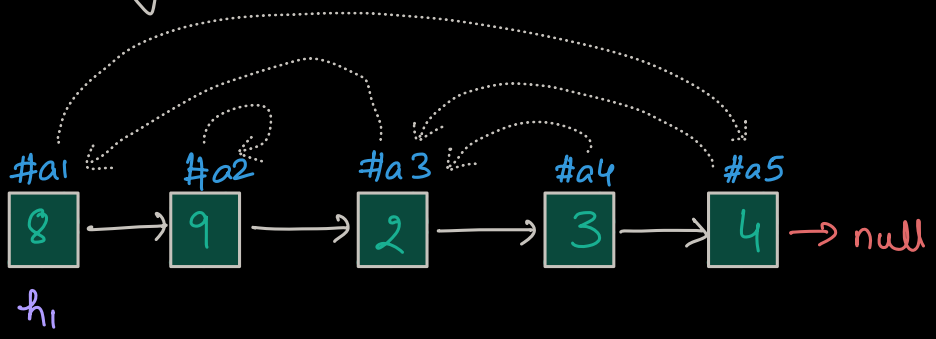
Eg 1:



cloned data ⇒



Ideas: Map every node in original to respective node in clone.



Ideas: Hashmap <Node, Node> hm

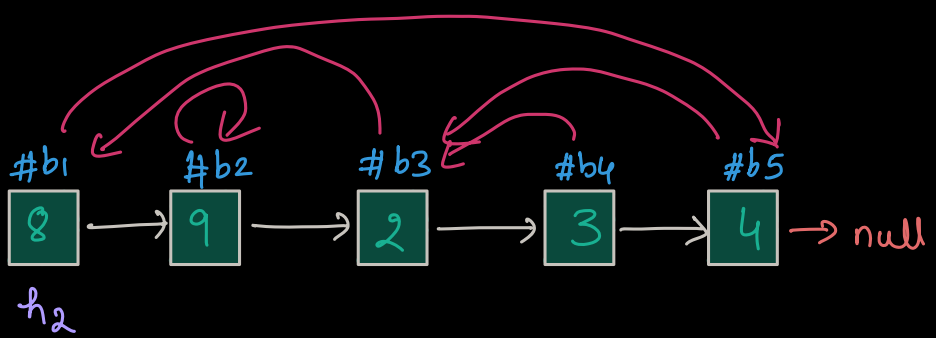
$\langle a_1, b_1 \rangle$
$\langle a_2, b_2 \rangle$
$\langle a_3, b_3 \rangle$
$\langle a_4, b_4 \rangle$
$\langle a_5, b_5 \rangle$

$b_1.\text{random} = hm[a_1.\text{random}] = hm[a_5] = b_5$

$b_2.\text{random} = hm[a_2.\text{random}] = hm[a_2] = b_2$

$\vdots$

$b_5.\text{random} = hm[a_5.\text{random}] = hm[a_3] = b_3$



Pseudocode

```
Node cloneLL(Node h1) {
```

Step 1: Create a clone of h_1 & only fill the next nodes
Say head of clone list = h_2

Step 2: For a node in original, map it to its respective clone

```
Node temp1 = h1
Node temp2 = h2
Hashmap<Node, Node> hm
while (temp1 != null) {
    hm.add({temp1, temp2})
    temp1 = temp1.next
    temp2 = temp2.next
}
```

TC : O(n)
SC : O(n)

Step 3: Fill the random values for all nodes in hm

```
temp1 = h1, temp2 = h2
while(temp1 != null) {
    temp2.random = hm[temp1.random]
    temp1 = temp1.next
    temp2 = temp2.next
}
```

Step 4: Return head of cloned list

```
return h2
}
```