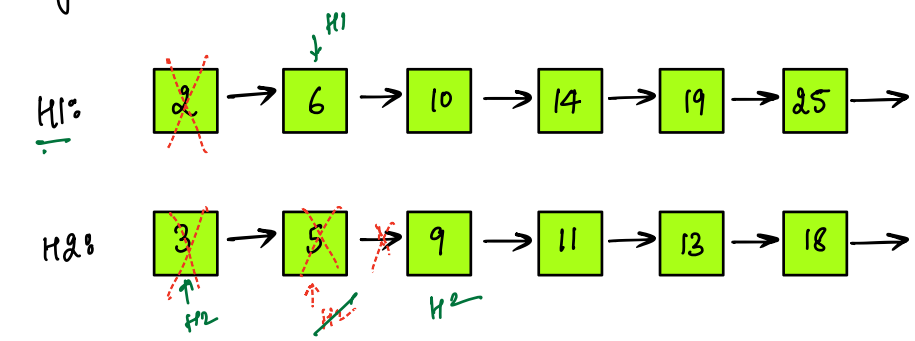
- Merge 2 sorted Linked Lists & return the head of merged LL.




```
tail.next = h2
tail = tail.next
h2 = h2.next
tail.next = NULL
```

```
tail.next = h1
tail = tail.next
h1 = h1.next
tail.next = null
```

```
Node merge ( Node h1 , Node h2)
{
    Node h3 = null, tail = null;
    Node h3 = new Node(-1); tail = h3;   // dummy node

    if( h1 == null) return h2;
    if( h2 == null) return h1;

    if( h1.data < h2.data) { h3 = h1; tail = h1; h1 = h1.next }
    else { h3 = h2; tail = h2; h2 = h2.next; }
```

```
while ( h1 != null && h2 != null )
{
        if ( h1.data < h2.data { // ____ }
        else { // _ }
}
if ( h1 != null ) { tail.next = h1; }
if ( h2 != null ) { tail.next = h2; }

return h3;        return h3.next;
}                   ↳ in case of dummy node.
```

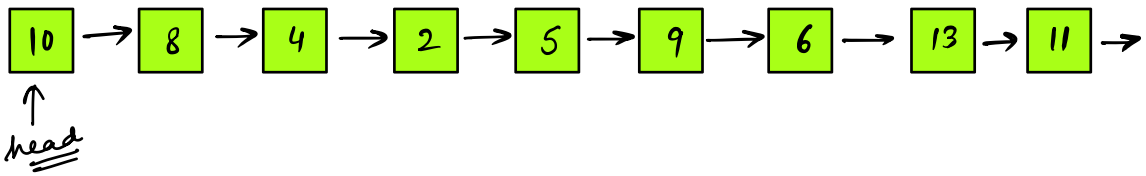
Merge sort

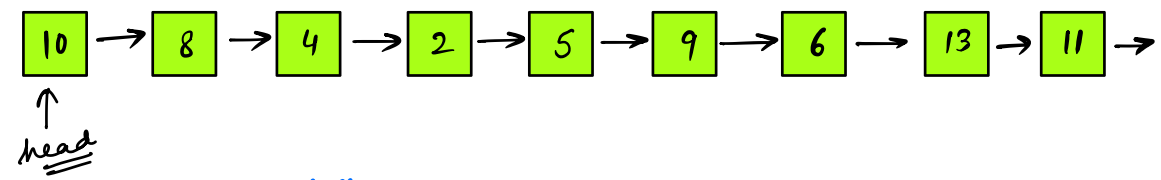
```
Node mergeSort ( Node head )
{
    if( head == NULL || head.next == NULL) return head;

    Node m = middle ( head);
Node head2 = m.next;
    m.next = NULL;

    head = mergesort ( head );
    head2 = mergesort ( head2);

    return merge(head, head2);

}
```

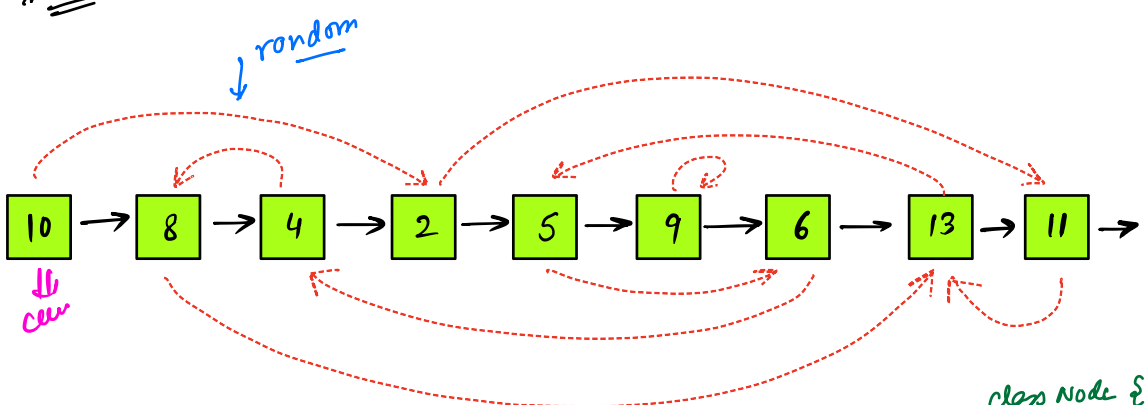
T.C: nlogn

S.C: O(1) + recursion space

Q Clone LL





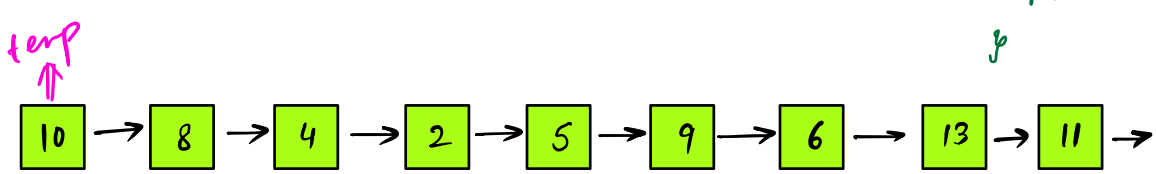


x = HM[curr.random]

temp.random = x;

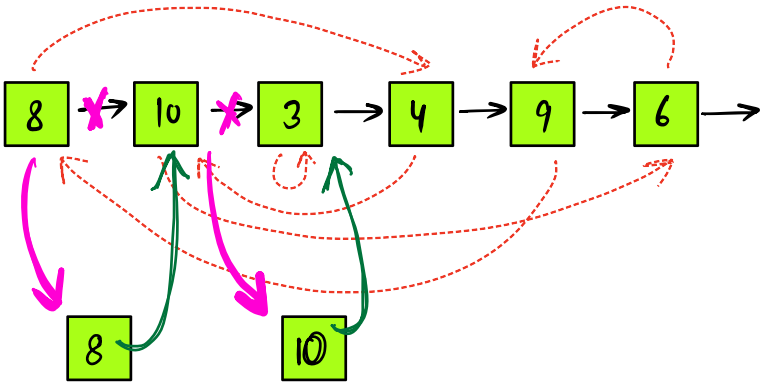
→ T.C : O(N)
S.C : O(N)

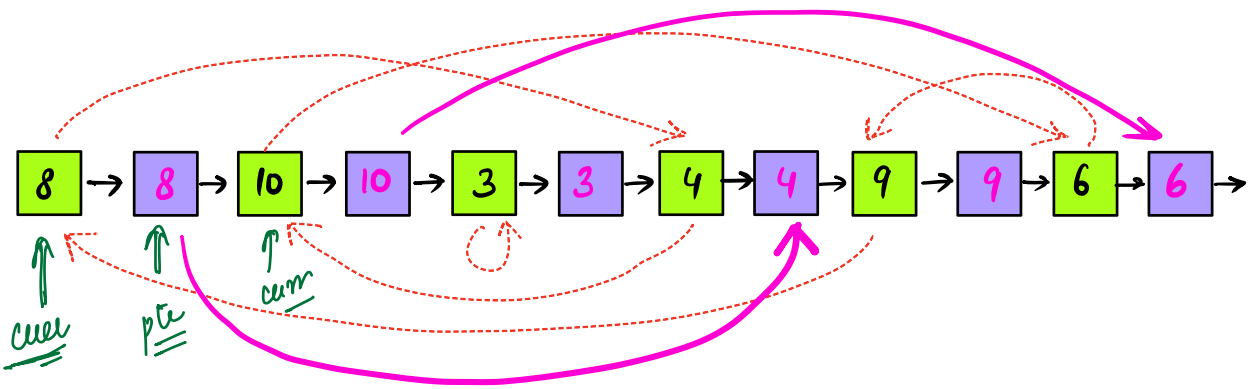
```
class Node {
    int data;
    Node next;
    Node random;
}
```



distance approach - $O(N^2)$

Hashmap < Node , Node > → while creating the new LL, we can build this HM





curr.next.random = curr.random.next

curr = curr.next.next;

curr.next = curr.next.next;

ptr.next = ptr.next.next;

if(head==NULL) return NULL;

Step 1 : combine the nodes

```
curr = head;
while( curr!=NULL)
{   Node x = new Node(curr.data);
    x.next = curr.next;
    curr.next = x;
    curr = curr.next.next;
}
```

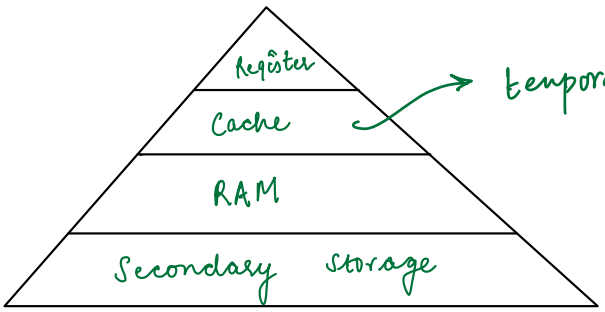
Step2:- update random pointers for new nodes

```
curr = head;
while ( curr != NULL)
{       if( curr.random != null)
            curr.next.random = curr.random.next;
        curr = curr.next.next;
}
```

Step3:

```
curr = head;  head2 = curr.next;  ptr = curr.next;
while ( curr != null)
{       curr.next = curr.next.next;
        curr = curr.next;
        if( curr != null)
    { ptr.next = ptr.next.next;
        ptr = ptr.next; }
}
return head2;
```

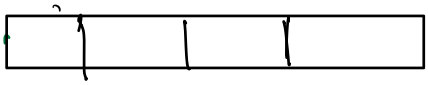
Q

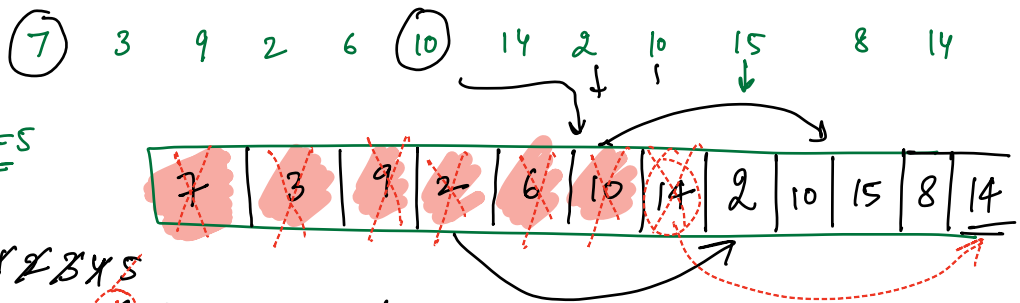


→ temporary storage

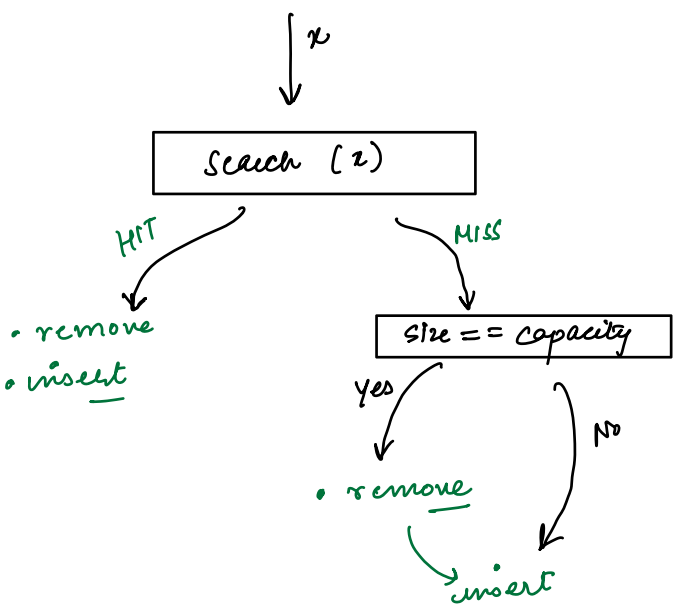
- limited capacity
(replacement

LRU - Least Recently used

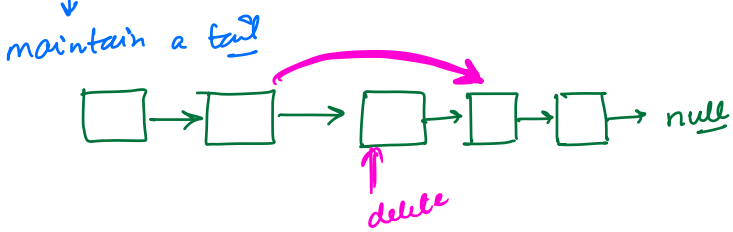


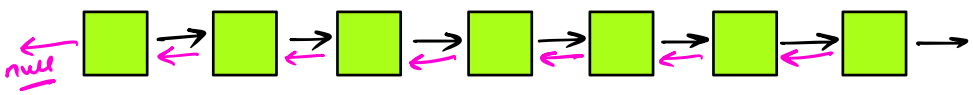


- search
- delete
- insert



	arrays	SLL	SLL + Hashmap	DLL + Hashmap
Search	O(n)	O(n)	O(1)	
insert at end	O(1)	O(1)	O(1)	
delete	O(n)	O(1)	O(1)	

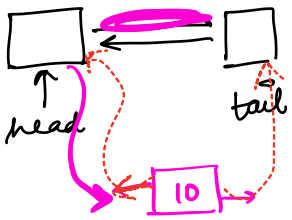


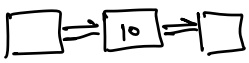



```
class Node {
    int data;
    Node next;
    Node prev;
}
```

10 15 19 20 15 18 23 20 19 17 17

capacity = 5
size = 0
head
tail





HM

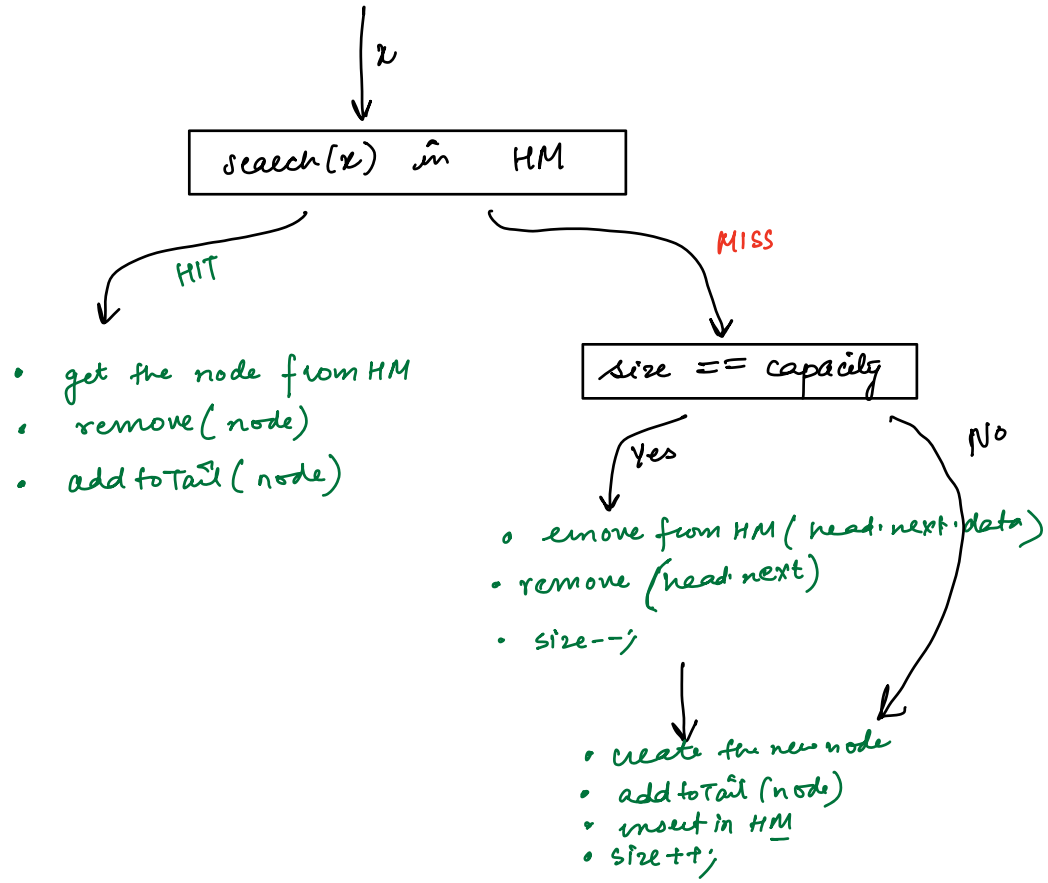
<10, idx of 10>
<15, —>
<19 —>
<20 —>
<18 —>

```
addToTail ( Node x)
{   x.next = tail
    x.prev = tail.prev
    tail.prev.next = x
    tail.prev = x
}
```

0 ⇌ 10 ⇌ 15 ⇌ 19 ⇌ 20 ⇌ 15 ⇌ 18

temp = HM[15]

```
remove( temp )
{  temp.prev.next = temp.next
   temp.next.prev = temp.p
}
```


x
search(x) in HM
HIT
MISS
get the node from HM
remove(node)
addToTail(node)
size == capacity
Yes
No
remove from HM(head.next.data)
remove(head.next)
size--;
create the new node
addToTail(node)
insert in HM
size++;

```
class LRUcache {
    class Node {
        int data;
        Node next;
        Node prev;
        Node() { }
    }
    HM<int, Node> m;
    Node head;
    Node tail;
    int size;
    int capacity;

    LRUcache(int cap) {
        head = new Node(-1);
        tail = new Node(-1);
        size = 0;
        capacity = cap;
        // initialise your HM
    }

    void put(int data)
    {
        if( data is present in HM)
        {   Node temp = HM[data];

            remove(temp);
            addToTail(temp);
        }
```

get / put

HIT

```
else
{
    if (size == capacity)
    {
        HM.remove(head.next.data);
        remove(head.next);
        size--;
    }

    Node temp = new Node(data);
        hm.insert({data, temp});
        addToTail(temp);
        size++;
}
}
```

get