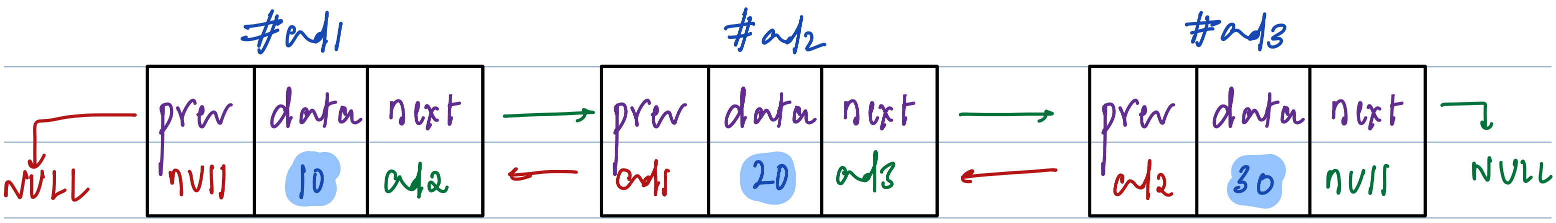
Todays Content

1. Double Linked List
2. LRU Cache

Double linked list

```
class Node{
    int data;
    Node *next, *prev;
    Node(int x){
        data = x;
        next = nullptr;
        prev = nullptr;
    }
}
```

Ex:



#obs: It's bidirectional.

We can travel from L → R & R → L.

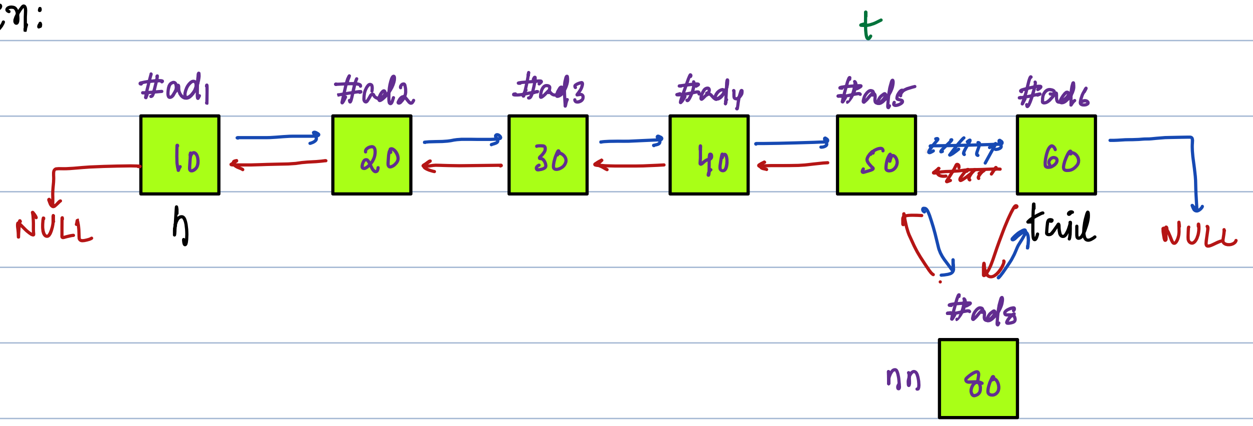
2Q Insert a new node, Just before tail of a Double Linked List

#Note1: Tail ref is given in Input

#Note2: No: of nodes >= 2

#Note3: Newnode is given, directly add before tail.

Ex:




```
void InsertBeforeTail(Node *nn, Node *tail){   TC: O(1)  SC: O(1)
    Node *t = tail->prev;
    tail->prev = nn;
    nn->prev = t;
    t->next = nn;
    nn->next = tail;
}
```

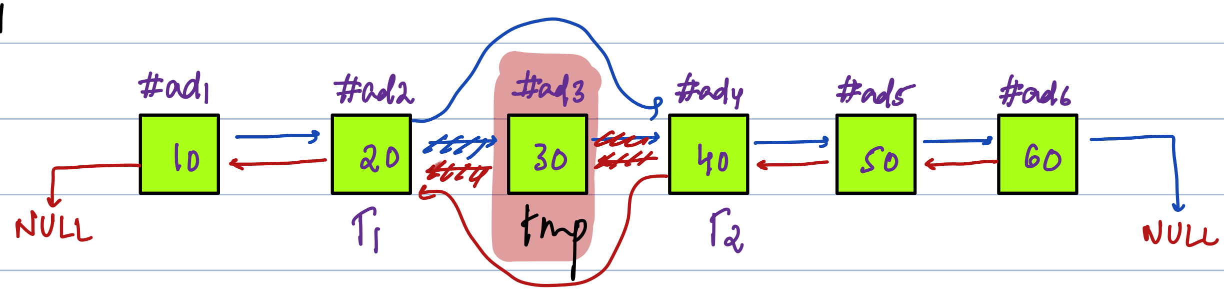
3Q Delete a given node from DLL, delete that node

#Note1: Node reference is given, to delete

#Note2: Given node is not head/tail node.

Note3: #No. of nodes >= 3

Ex1




```
void DeleteNode(Node *tmp){ TC: O(1) SC: O(1)
    Node *T1 = tmp->prev, *T2 = tmp->next;
    T1->next = T2;
    T2->prev = T1;
    tmp->next = nullptr;
    tmp->prev = nullptr;        # Isolating node.
}   delete tmp; # It will actually delete node
```

#obs: If we want to delete data from between

1. Using Double Linked List, if we have node address we can do it in O(1) time.

Memory hirarchy:

Top to down:
Memory limit
Increases.

Bottom to Top:
Retrival speed Increases.

Cache:

limit memory, To manage it, It follows principle

LRU: Last / least recently used.

Ex: limit: 5

Data : 7 3 9 2 3 7 10 14 2 10 3 8 14 15 16

#Cache: ~~7~~ ~~3~~ ~~9~~ ~~2~~ ~~3~~ ~~7~~ ~~10~~ ~~14~~ ~~2~~ ~~10~~ 3 8 14 15 16

old Head Delete. Delete Delete Delete New Tail

#Note: In cache duplicates not allowed ?
If a same data comes, we arrange position to latest

Flowchart:

search(x)

HIT → delete(x), addTail(x)

MISS → if (cache_size == limit) { delete(head) } addTail(x)

# Operations we perform	# Suitable DS	TC:
1. search(x)	Hashset(x) / HashMap < x, Node* >	O(1)
2. delete(x)	Doubly Linked List	O(1)
3. addTail(x)		O(1)
4. Cachesize:	c; # A count variable	O(1)
5. Insertion order	Vector/ Linked List/	✓

Each Page:

Page No : Key

Data : value

A page info given <Key, value>

Design a data structure that follows the constraints of a **Least Recently Used (LRU) cache**.

Implement the `LRUCache` class:

- `LRUCache(int capacity)` Initialize the LRU cache with **positive** size `capacity`.
- `int get(int key)` Return the value of the `key` if the key exists, otherwise return `-1`.
- `void put(int key, int value)` Update the value of the `key` if the `key` exists. Otherwise, add the `key-value` pair to the cache. If the number of keys exceeds the `capacity` from this operation, **evict** the least recently used key.

The functions `get` and `put` must each run in `O(1)` average time complexity.

#Note1: Both get & set will count as accessing item.

#Note2: Key : Page No   Value : Data at page no

Ex1:

Capacity = 3;

put(1, 1) put(2, 4) put(5, 3) put(2, 7) put(1, 8) get(5) → 3 put(7, 10) get(10) → -1

#Cache

#old head

~~(1, 1)~~	~~(2, 7)~~	~~(5, 3)~~	~~(2, 7)~~	(1, 8)	(5, 3)	(7, 10)

#new tail

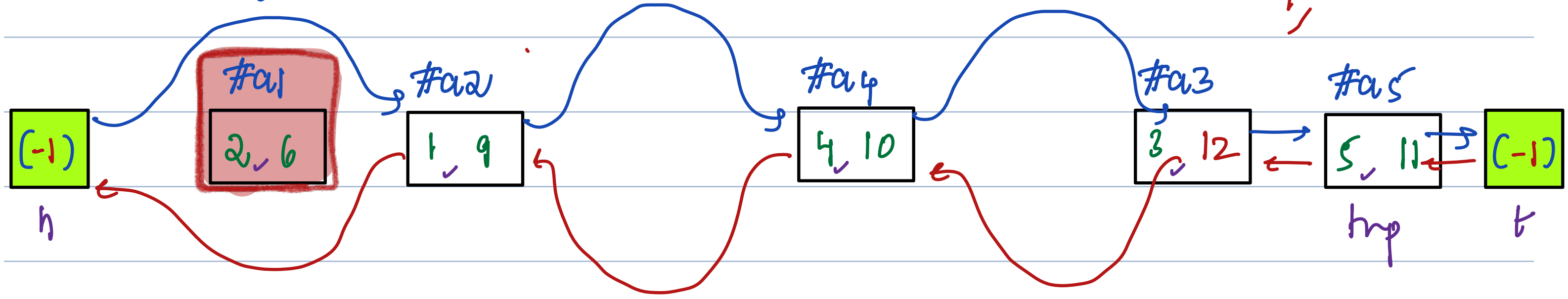
LRU Cache using DLL + HashMap:

Note: Create a double linked list, with h & t.

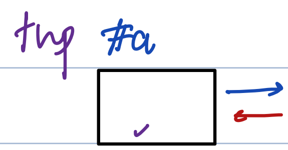
capacity = 4

put(2, 6) put(1, 9) put(3, 10) put(4, 10) put(5, 11) put(3, 12) get(10) get(5)

-1;



HM: { ~~<2 : a1>~~ <1 : a2> <3 : a3> <4 : a4> <5 : a5> }

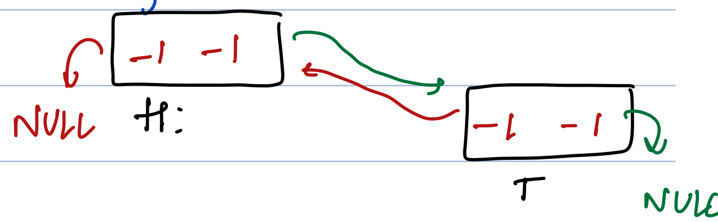



```
class Node {
  public:
    int k, v;   # K = Key of page no   V = Value of page no
    Node *prev, *next;
    Node(int u, int v) {
        k = u, v = v;
        prev = nullptr;
        next = nullptr;
    }
};
```

```
unordered_map<int, Node*> um;
                    |   ↳ Value = Node address in linked list
                    ↳ key = Page no

int c=0;
Node *H, *T;   # h & t pointer of Doubly linked list
class LRUCache{
    public:
        LRUCache(int capacity){
            c=capacity; # c is global variable, can use c across
            H = new Node(-1,-1);
            T = new Node(-1,-1);
            H->next=T
            T->prev = H;
        }

        int get(int key){
            if(um.find(key) ==um.end()){
                return -1;
            }
            else{
                Node * tmp = um[key];
                DeleteNode(tmp); #Only Isolate tmp node
                InsertBeforeTail(tmp, T);
                return tmp->v;
            }
        }
```



```
void put(int key, int value){
    if(um.find(key) ==um.end()){
        if( c== um.size()){
            Node *tmp = H->next;
            DeleteNode(tmp); # Only Isolate tmp node
            int pageNo = tmp->k;
            um.erase(pageNo); # It is removed from hashmap
            delete tmp; # Now delete node.
        }

        Node *nn = new Node(key, Value);
        InsertBefore Tail(nn,T);
    }   um[key]= nn; # Store <pageno, Node*> in hashmap
    else{
        Node * tmp = um[key];
        DeleteNode(tmp); # Only Isolate tmp node
        InsertBefore Tail(tmp, T);
        tmp->V = value; # update value for pageno key.
    }
}
void InsertBeforeTail(Node *nn, Node *tail){
    Node *t = tail->prev;
    tail->prev = nn;
    nn->prev = t;
    t->next = nn;
    nn->next = tail;
}
```

```
void DeleteNode(Node *tmp){   TC: O(1)  SC: O(1)
    Node *T1 = tmp->prev, *T2 = tmp->next;
    T1->next = T2;
    T2->prev = T1;
    tmp->next = nullptr;
    tmp->prev = nullptr;
}
}
```

Isolating node.