LINKED LIST

No.	Problem Statement	Solution	Time complexity	Space complexity
1	**Reverse Linked List**			
	Given the head of a singly linked list, reverse the list, and return the reversed list.	- Maintain 3 pointers: 'prev', 'curr' and 'temp' 1) Initialize 'prev' to NULL 2) While curr is not NULL, do the following: - Store **curr->next** in the '**temp**' pointer. - Update **curr -> next = prev** , effectively reversing the link. - Update **prev = curr** for the next iteration. - Update **curr = temp** for the next iteration.	O(N)	O(1)
2	**Merge Two Sorted Lists**			
	Given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted list.	- Create new node: `ListNode* new_head = new ListNode()` // Dummy node for the merged list - Initialize a `curr = new_head` to keep track of the current node in the merged list. - Iterate through both the lists and compare values and update the `'curr'` pointer - Return `new_head->next`	O(M+N)	O(1)
3	**Linked List Cycle**			
	Given the head of a linked list, determine if the linked list has a cycle in it.	Approach_1: Use **Unordered_set to keep track of visited nodes**	O(N)	O(N)

		Approach_2: Tortoise & Hair Algorithm (slow + fast) pointers
		- 'slow': Moves **one** step at a time
		- 'fast': Moves **two** steps at a time
		- `while(fast!=NULL && fast->next!=NULL)`
		`slow = slow->next;  fast = fast->next->next`
		`if(slow==fast) return true`
			O(N)	O(1)

4	**Reorder List**			
	You are given the head of a singly linked-list. The list can be represented as: L0 → L1 → … → Ln - 1 → Ln Reorder the list to be on the following form: L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …	1) Split the list in half 2) Reverse the second half 3) Merge the two lists	O(N)	O(1)
5	**Remove Nth Node From the End of the List**			
	Given the head of a linked list, remove the nth node from the end of the list and return its head.	1) Find the node to be deleted from the start of the list 2) Delete the node	O(N)	O(1)
6	**Copy List With Random Pointer**			

	A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.. Construct a deep copy of the list. The deep copy should consist of exactly n brand new nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list	Approach_1: **Unordered_map --> {original_node, copy_node}**
		1) Generate copy node for each node in the original list and store it in the map
		`mp[curr] = new Node(curr->val)`
		2) Set the 'next' and 'random' pointers for the copy nodes
			O(N)	O(N)

		Approach_2: **Each original node is followed by it's duplicate node**
		1) Iterate through original list and duplicate each node.
		- Every original node is immediately followed by it's duplicate node
		2) Iterate the new list and assign the random pointer for each duplicate node
		- `curr->next->random = curr->random->next`
		3) Seperate the original list and the dupliacte list
		`Node* curr1 = head | Node* curr2 = head->next;`
		`while(curr1)`
		`curr1->next = curr2->next;`
		`curr1 = curr1->next;`
		`if(curr1) curr2->next = curr1->next;`
		`else curr2->next = NULL;`
		`curr2 = curr2->next;`
			O(N)	O(1)

7	**Add Two Numbers**			

	Given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list. You may assume the two numbers do not contain any leading zero.
	Ex. l1 = [2,4,3] l2 = [5,6,4] -> [7,0,8] (342 + 465 = 807)
		- **Traverse both the lists and sum digit-by-digit**
		// 'dummy_node' that will be used to point at the head of the new list containing 'sum'
		`ListNode* dummy = new ListNode();`
		`ListNode* curr = dummy;`
		`while(curr1 && curr2)`
		`    n1 = curr1->val;      n2 = curr2->val;`
		`    n = n1 + n2 + carry;`
		`    carry = n/10;`
		`    curr->next = new ListNode(n%10);`
		`    curr = curr->next;   curr1 = curr1->next;   curr2 = curr2->next;`
		`if(carry>0)`
		`    curr->next = new ListNode(carry);`
		`    curr = curr->next;`
		curr->next = NULL;
			O(max(N1, N2))	O(max(N1, N2))

8	**Find The Duplicate Number**			

	Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive. There is only one repeated number in nums, return this repeated number.	Idea: **Slow and Fast Pointers**
		- If there's a duplicate there must be a cycle
		- Consider elements of the array as nodes of linked list such as `nums[i]->next = 'element at index = i'`
		- `slow = nums[0], fast[nums[0]]`
		`while(slow!=fast)`
		`    slow = nums[slow];`
		`    fast = nums[nums[fast]];`
		// 1. Take slow pointer back to start
		// 2. Move both pointers **one step** at a time until they meet again.
		// The meeting point is the start of the cycle, which corresponds to the duplicate element in the array
		`slow = 0;`
		`while(slow!=fast)`
		`    slow = nums[slow];`
		`    fast = nums[fast];`
			O(N)	O(1)

9	**LRU Cache**			

	Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.
	- LRUCache(int capacity)
	Initialize the LRU cache with positive size capacity.
	- int get(int key) --> O(1)
	Return the value of the key if the key exists, otherwise return -1.
	- void put(int key, int value) --> O(1)
	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.
		- **Unordered_map + Doubly Linked List**
		- `unordered_map<int, Node*> mp;` // {key, pointer to the node having value for 'key'}
		`Node* left;` // (left->next) points to LRU node
		`Node* right;` // (right->prev) points to MRU node
		`void remove(Node* node){`
		`    node->prev->next = node->next;`
		`    node->next->prev = node->prev;`
		`void insert(Node* node){`
		`    right->prev->next = node;`
		`    node->next = right;`
		`    node->prev = right->prev;`
		`    right->prev = node;`
		- get() : remove(mp[key]); // Remove the node from current location
		insert(mp[key]); // Make the node most recently used
		- put() : Add the {key, value} pair if don't exist
		Remove the node from current location and make the node most recently used
		If the size increases max capacity remove the LRU node
			O(1)	O(capacity)

10	**Merge K Sorted List**			
	Given an array of k linked-lists lists, each is sorted in ascending order. Merge all the linked-lists into one sorted linked-list and return it.	Idea: **Merge 2 lists at a time** - lists[0] = Head of the merged lists lists[0] & lists[n-1] lists[1] = Head of the merged lists lists[1] & lists[n-2] and so on	O(M*logN) *M: total no. of nodes in all the lists* *N: lists.size()*	O(1)
11	**Reverse Nodes in K Groups**			

	Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list.
	Ex. head = [1,2,3,4,5], k = 2 -> [2,1,4,3,5]
		Idea: Maintain
		- Head of current group --> `grp_head`
		- Head of reversed group --> `reverse_head`
		- Tail of the previous group --> `prev_tail`
		`while(cnt<n/k){`
		`ListNode* group_head = curr;`
		`reverse_head = reverse_list(curr, k);`
		`if(prev_tail == NULL) ans = reverse_head;` // Suggests the first group
		`else prev_tail->next = reverse_head;`
		`prev_tail = group_head;`
		`cnt++;`
		`}`
		`if(curr) prev_tail->next = curr;` // For the remaining nodes
			O(N)	O(1)