**Q1. Delete middle node of a Linked List**

Given a singly linked list, delete middle of the linked list.

For example, if given linked list is 1->2->3->4->5 then linked list should be modified to 1->2->4->5

**Logic -**

1. Use Slow and Fast Pointer to find the middle node.

**Code -**

public class Solution {

public ListNode solve(ListNode A) {

if(A == null || A.next == null) return null;

ListNode slow = A;

ListNode fast = A;

ListNode prev = null;

while(fast!=null && fast.next!=null) {

prev = slow;

slow = slow.next;

fast = fast.next.next;

}

prev.next = slow.next;

return A;

}

}

**Q2. K reverse linked list**

Given a singly linked list **A** and an integer **B**, reverse the nodes of the list **B** at a time and return the modified linked list.

**Logic -**

1. Use 3 variables (curr, prev, temp) for reversing B nodes then recursively call for next batch and connect tail of current batch to head of next batch.

**Code -**

public class Solution {

public ListNode reverseList(ListNode A, int B) {

if(A==null) return null;

ListNode curr = A;

ListNode prev = null;

ListNode head = A;

int k = B;

while(curr!=null && k>0) {

ListNode temp = curr.next;

curr.next = prev;

prev = curr;

curr = temp;

k-=1;

}

head.next = reverseList(curr, B);

return prev;

}

}

**Q3. Design a LinkedList**

Design and implement a Linked List data structure.  
A node in a linked list should have the following attributes - an integer **value** and a **pointer** to the next node. It should support the following operations:

**Logic -**

1. Insert - Check if the given position is between1 and length +1. If yes, check if position == 1, if yes insert / delete / delete at head else insert in between using prev and curr.

**Code -**

public static class ListNode {

public int val;

public ListNode next;

ListNode(int x) { val = x; next = null; }

}

public static ListNode Head = null;

public static int length = 0;

public static void insert\_node(int position, int value) {

// @params position, integer

// @params value, integer

if (position >= 1 && position <= length+1) {

// Insert at head

if(position == 1) {

ListNode newNode = new ListNode(value);

newNode.next = Head;

Head = newNode;

length += 1;

return;

}

// Other cases

int k = position;

ListNode temp = Head;

ListNode prev = null;

ListNode newNode = new ListNode(value);

while(k>1) {

prev = temp;

temp = temp.next;

k-=1;

}

prev.next = newNode;

newNode.next = temp;

length += 1;

return;

}

}

public static void delete\_node(int position) {

// @params position, integer

if (Head == null) {

return;

}

if (position >= 1 && position <= length) {

if (position == 1) {

Head = Head.next;

length -= 1;

return;

}

int k = position;

ListNode temp = Head;

ListNode prev = null;

while(k>1) {

prev = temp;

temp = temp.next;

k-=1;

}

prev.next = temp.next;

length -= 1;

return;

}

}

public static void print\_ll() {

// Output each element followed by a space

ListNode temp = Head;

int flag = 0;

while(temp != null) {

if (flag==0) {

flag = 1;

System.out.print(temp.val);

}

else {

System.out.print(" " + temp.val);

}

temp = temp.next;

}

return;

}

**Q 4 - Remove Duplicates from Sorted List**

Given a **sorted** linked list, delete all duplicates such that each element appears only once.

**Logic -**

1. Maintain 2 pointers and check their values. If the value of prev and curr is same then connect prev.next to curr.next and move the curr pointer to the next node, else move both pointers to the next respective nodes.

**Code -**

public class Solution {

public ListNode deleteDuplicates(ListNode A) {

if(A==null || A.next==null) {

return A;

}

ListNode prev = A;

ListNode temp = A.next;

while(temp != null) {

if (temp.val == prev.val) {

prev.next = temp.next;

} else {

prev = temp;

}

temp = temp.next;

}

return A;

}

}

**Q5 List Cycle**

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Logic -

1. Use slow and fast pointers and find a point where slow == fast.
2. Start a pointer from the start and from the meeting point with the speed of 1 node/time.
3. The meeting of these pointers is the answer.

Code -

public class Solution {

public ListNode detectCycle(ListNode A) {

if (A==null || A.next==null) return A;

ListNode slow = A;

ListNode fast = A;

int flag = 0;

while(fast.next != null && fast.next.next!=null) {

slow = slow.next;

fast = fast.next.next;

if (slow == fast) {

// A cycle exists

flag = 1;

break;

}

}

if (flag == 0) {

return null;

}

slow = A;

while(slow != fast) {

slow = slow.next;

fast = fast.next;

}

return fast;

}

}

**Q6. Reorder List**

Given a singly linked list **A**

A: A0 → A1 → … → An-1 → An

reorder it to:

A0 → An → A1 → An-1 → A2 → An-2 → …

**Logic -**

1. Find the mid-point using slow-fast pointers.
2. Reverse the next half of the Linked List.
3. Merge the 2 Linked Lists.

**Code -**

public class Solution {

public ListNode reverseList(ListNode A) {

if (A==null || A.next==null) return A;

ListNode curr = A;

ListNode prev = null;

ListNode temp = null;

while(curr != null) {

temp = curr.next;

curr.next = prev;

prev = curr;

curr = temp;

}

return prev;

}

public ListNode reorderList(ListNode A) {

if (A==null || A.next==null || A.next.next==null) return A;

ListNode slow = A;

ListNode fast = A;

// Find the mid point of linked list

while(fast.next != null && fast.next.next != null) {

slow = slow.next;

fast = fast.next.next;

}

ListNode h2 = slow.next;

slow.next = null;

ListNode p1 = A;

ListNode p2 = reverseList(h2);

ListNode t2 = null;

// Merge the reversed second half List and current list

while(p2 != null) {

t2 = p2.next;

p2.next = p1.next;

p1.next = p2;

p1 = p1.next.next;

p2 = t2;

}

return A;

}

}

**Q7. Sort List**

Sort a linked list, **A** in **O(n log n)** time using constant space complexity.

**Logic** – Use the same logic as merge sort

**Code -**

public class Solution {

public ListNode midNode(ListNode A) {

// Returns mid node

if (A==null || A.next==null) return A;

ListNode slow = A;

ListNode fast = A;

while(fast.next != null && fast.next.next != null) {

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

public ListNode mergeList(ListNode A, ListNode B) {

// Merges 2 sorted lists to form a sorted linked list

ListNode ans = new ListNode(-1);

ListNode p1 = ans;

ListNode currA = A;

ListNode currB = B;

while(currA != null && currB != null) {

if (currA.val < currB.val) {

p1.next = currA;

currA = currA.next;

p1 = p1.next;

} else {

p1.next = currB;

currB = currB.next;

p1 = p1.next;

}

}

if (currA == null) {

p1.next = currB;

}

if (currB == null) {

p1.next = currA;

}

return ans.next;

}

public ListNode sortList(ListNode A) {

if (A == null || A.next == null) return A;

ListNode temp = midNode(A);

ListNode h2 = temp.next;

temp.next = null;

// A and h2

ListNode temp1 = sortList(A);

ListNode temp2 = sortList(h2);

return mergeList(temp1, temp2);

}

}

**Q8 Palindrome List**

Given a singly linked list **A**, determine if it's a palindrome. Return **1** or **0,** denoting if it's a palindrome or not, respectively.

**Logic -**

1. Compare the first half of the linked list with the reverse of the second half.

**Code -**

public class Solution {

public ListNode reverseList(ListNode A) {

ListNode prev = null;

ListNode curr = A;

ListNode temp = null;

while(curr != null) {

temp = curr.next;

curr.next = prev;

prev = curr;

curr = temp;

}

return prev;

}

public ListNode findMid(ListNode A) {

ListNode slow = A;

ListNode fast = A;

while(fast.next != null && fast.next.next != null) {

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

public int lPalin(ListNode Head) {

ListNode mid = findMid(Head);

ListNode p1 = mid.next;

ListNode A = Head;

p1 = reverseList(p1);

while(p1 != null && A != null) {

if(A.val != p1.val) {

return 0;

}

p1 = p1.next;

A = A.next;

}

return 1;

}

}

**Q9. Longest Palindromic List**  
Given a linked list of integers. Find and return the length of the longest palindrome list that exists in that linked list.

A palindrome list is a list that reads the same backward and forward.

**Logic -**

1. Reverse and count the size of pallidrome by considering the 2 nodes as a center.

**Code -**

public class Solution {

public int count(ListNode A, ListNode B) {

int count = 0;

ListNode temp1 = A;

ListNode temp2 = B;

while(temp1 != null && temp2 != null && temp1.val == temp2.val) {

temp1 = temp1.next;

temp2 = temp2.next;

count += 1;

}

return count;

}

public int solve(ListNode A) {

int count = 0;

ListNode curr = A;

ListNode prev = null;

ListNode temp = null;

while(curr != null) {

temp = curr.next;

curr.next = prev;

// Count the pallidrome length

int x = 2 \* count(curr, temp); // Even pallidrome

int y = 2 \* count(prev, temp) + 1; // Odd pallidrome

count = Math.max(count, x);

count = Math.max(count, y);

prev = curr;

curr = temp;

}

return count;

}

}

**Q10 Flatten a linked list**

Given a linked list where every node represents a linked list and contains two pointers of its type:

1. Pointer to next node in the main list (**right pointer**)
2. Pointer to a linked list where this node is head (**down pointer**). All linked lists are sorted.

You are asked to flatten the linked list into a single list. Use **down** pointer to link nodes of the flattened list. **The flattened linked list should also be sorted.**

**Logic -**

1. We merge 1 linked list to the next flattened linked list.

**Code -**

ListNode merge (ListNode a, ListNode b) {

ListNode ans = new ListNode(-1);

ListNode p1 = ans;

ListNode h1 = a;

ListNode h2 = b;

while(h1 != null && h2 != null) {

if(h1.val < h2.val) {

p1.down = h1;

p1 = p1.down;

h1 = h1.down;

}

else {

p1.down = h2;

p1 = p1.down;

h2 = h2.down;

}

}

if (h1 != null) p1.down = h1;

if (h2 != null) p1.down = h2;

return ans.down;

}

ListNode flatten(ListNode root) {

if (root == null || root.right == null) {

return root;

}

return merge(root, flatten(root.right));

}

**Q11 Partition List**

Given a linked list **A** and a value **B**, partition it such that all nodes less than **B** come before nodes greater than or equal to **B**.

You should preserve the original relative order of the nodes in each of the two partitions.

**Logic -**

1. Maintain 2 nodes for nodes with value >= B and value < B.

**Code -**

public class Solution {

public ListNode partition(ListNode A, int B) {

ListNode p1 = new ListNode(-1);

ListNode p2 = new ListNode(-1);

ListNode temp1 = p1;

ListNode temp2 = p2;

ListNode head = A;

while(head != null) {

if(head.val < B) {

temp1.next = head;

temp1 = temp1.next;

} else {

temp2.next = head;

temp2 = temp2.next;

}

head = head.next;

}

temp1.next = p2.next;

temp2.next = null;

return p1.next;

}

}

**Q12. LRU Cache**

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

* get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
* set(key, value) - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least recently used item before inserting the new item.

The LRUCache will be initialized with an integer corresponding to its capacity. Capacity indicates the maximum number of unique keys it can hold at a time.

**Logic –** Maintain a Hashmap and doubly linked list.

1. Get -> If the key is present in HM, then remove the it from doubly LL and insert again. If not present in HM return –1.
2. Set -> If the key is present in HM, Remove the node and insert the node with new value and update in HM. If not present in HM, check if capacity == size. If cap == size then delete the first node in doubly LL and then insert the new Node in doubly LL and HM. If cap > size then simply insert the node in HM and doubly LL.

**Code -**

public class Solution {

int capacity;

int size;

public class Node {

int key;

int value;

Node prev;

Node next;

public Node(int k, int v) {

prev = null;

next = null;

key = k;

value = v;

}

}

Node head = new Node(-1, -1);

Node tail = new Node(-1, -1);

HashMap<Integer, Node> map = new HashMap<Integer, Node>(); // key -> Node

public Solution(int capacity) {

this.capacity = capacity;

this.size = 0;

head.next = tail;

tail.prev = head;

}

public void insert(Node node) {

node.next = tail;

node.prev = tail.prev;

tail.prev.next = node;

tail.prev = node;

}

public void remove(Node node) {

node.prev.next = node.next;

node.next.prev = node.prev;

}

public int get(int key) {

if(map.containsKey(key)) {

Node x = map.get(key);

remove(x);

insert(x);

return x.value;

}

return -1;

}

public void set(int key, int value) {

if(map.containsKey(key)) {

Node x = map.get(key);

Node newNode = new Node(key, value);

map.remove(key);

remove(x);

insert(newNode);

map.put(key, newNode);

return;

}

if(size == capacity) {

map.remove(head.next.key);

remove(head.next);

size-=1;

}

Node newNode = new Node(key, value);

insert(newNode);

map.put(key, newNode);

size +=1;

}

}