**Note: All the Code is in JavaScript Programming Language**

**Question 1**

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.If there are **even** nodes, then there would be **two middle** nodes, we need to delete the second middle element. For example, if given linked list is 1->2->3->4->5->6 then it should be modified to 1->2->3->5->6.If the input linked list is NULL or has 1 node, then it should return NULL

**Example 1:**

Input:

LinkedList: 1->2->3->4->5

Output:1 2 4 5

**Example 2:**

Input:

LinkedList: 2->4->6->7->5->1

Output:2 4 6 5 1

**Solution:** The approach is to find the middle node using the while loop from the given linked list. For this, we will need 2 pointers approach where the pointers are traversing the list. The variable will track the node from the list by keeping one variable ahead of the other variable to find the middle node through the function.

| const deleteMiddle = (head) => {  let slow = fast = head;  let prevNode = null;  while (fast && fast.next) {  prevNode = slow;  slow = slow.next;  fast = fast.next.next;  }   if (!prevNode) return null;  prevNode.next = slow.next;    return head; } |
| --- |

**Question 2**

Given a linked list of **N** nodes. The task is to check if the linked list has a loop. Linked list can contain self loop.

**Example 1:**

Input:

N = 3

value[] = {1,3,4}

x(position at which tail is connected) = 2

Output:True

Explanation: In the above test case N = 3. The linked list with nodes N = 3 is given. Then value of x=2 is given which means the last node is connected with xth node of linked list. Therefore, there exists a loop.

**Example 2:**

Input:

N = 4

value[] = {1,8,3,4}

x = 0

Output:False

Explanation:For N = 4 ,x = 0 means then lastNode->next = NULL, then the Linked list does not contains any loop.

**Solution:**

The strategy for this question lies behind the two-pointer approach. We need to create a slow pointer and a fast pointer and iterate over and over again. If the fast pointer catches the slow pointer, then we can say that it is a circular linked list. Otherwise, if the fast pointer gets to the end, then it’s not a circular linked list.

| const hasCycle = (head) => {  let fast = head;  while (fast && fast.next) {  head = head.next;  fast = fast.next.next;  if (head === fast) return true;  }  return false; }; |
| --- |

**Question 3**

Given a linked list consisting of **L** nodes and given a number **N**. The task is to find the **N**th node from the end of the linked list.

**Example 1:**

Input:

N = 2

LinkedList: 1->2->3->4->5->6->7->8->9

Output:8

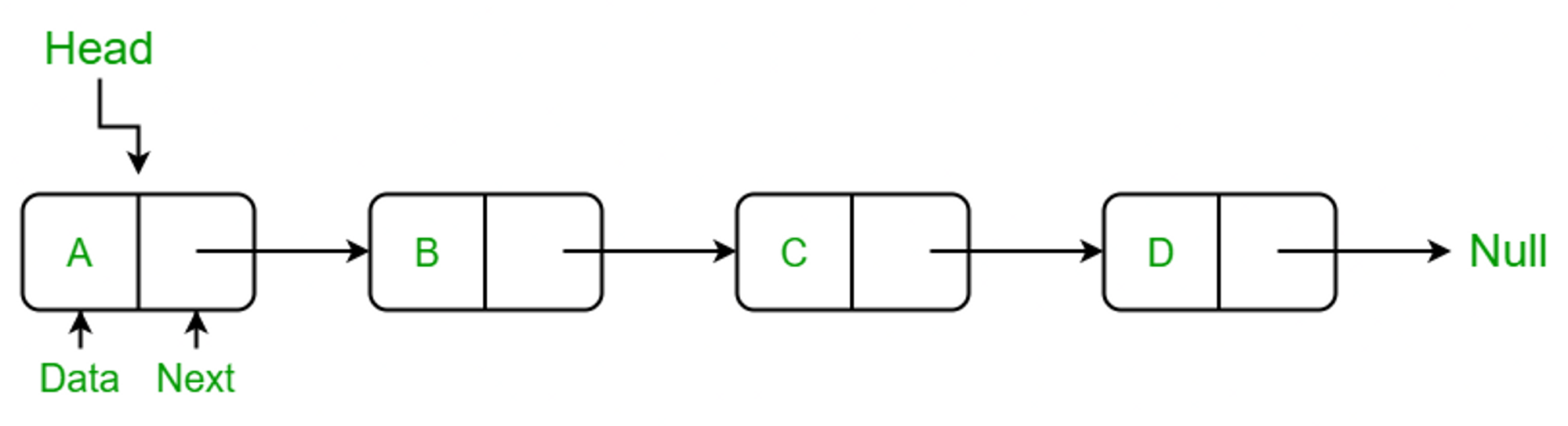
Explanation:In the first example, there are 9 nodes in linked list and we need to find 2nd node from end. 2nd node from end is 8.

**Solution:** For this problem, we will be finding the total length of the linked list and then display the nth node from the last. The function below takes the head of the linked list and value of the input. 2 pointers(fast and slow) will traverse the list. The fast pointers will move n steps ahead and suppose the list is shorter than n nodes means the pointer will become null and therefore it returns null. Once the fast pointer is n steps ahead, both pointers are moved forwards till the fast pointer reaches the end. This entire process ensures that the slow pointer will be at the Nth node from the end. The time complexity is O(n) where n is the length of the linked list and space complexity is O(1) as no extra space has been utilized.

| const getNthFromEnd = (head, n) => {  let slow = fast = head;    // Move the fast pointer n steps ahead  for (let i = 0; i < n; i++) {  if (fast === null) {  // If the linked list is shorter than n nodes, return null  return null;  }  fast = fast.next;  }    // Move both pointers until the fast pointer reaches the end  while (fast !== null) {  slow = slow.next;  fast = fast.next;  }    // The slow pointer will now be at the Nth node from the end  // Return its value  return slow.value; }; |
| --- |

**Question 4**

Given a singly linked list of characters, write a function that returns true if the given list is a palindrome, else false.



**Examples:**

Input: R->A->D->A->R->NULL

**Output:** Yes

**Input:** C->O->D->E->NULL

**Output:** No

**Solution:**

If there is only one node or an empty node then it is considered as a palindrome and hence returns ‘true’. Otherwise, 2 pointers ‘slow’ and ‘fast’ will traverse onto the list. A stack is used to store the characters from the first half of the list. The job of the fast pointer is to move as twice as speedy as the slow pointer and push the characters into the stack until it reaches the end of the list. If the list has an odd number of nodes, the slow pointer moves to the next node to skip the middle node. The slow pointer continues traversing the remaining half of the list while comparing the characters with the characters popped from the stack. If at any point the characters don't match, the function returns false since it's not a palindrome. If all characters match, the function returns true since it's a palindrome.

The time complexity and space complexity would be O(n), where n is the size of the linked list.

| **const isLinkedListPalindrome = (head) => {  if (head === null || head.next === null) {  // An empty list or a single-node list is considered a palindrome  return true;  }   let slow = head;  let fast = head;  let stack = [];   // Move the fast pointer to the end of the list while pushing the characters onto the stack  while (fast !== null && fast.next !== null) {  stack.push(slow.value);  slow = slow.next;  fast = fast.next.next;  }   // If the list has an odd number of nodes, skip the middle node  if (fast !== null) {  slow = slow.next;  }   // Compare the remaining nodes with the characters popped from the stack  while (slow !== null) {  const top = stack.pop();   if (top !== slow.value) {  // The characters don't match, so it's not a palindrome  return false;  }   slow = slow.next;  }   // All characters matched, so it's a palindrome  return true; };** |
| --- |

**Question 5**

Given a linked list of N nodes such that it may contain a loop. A loop here means that the last node of the link list is connected to the node at position X(1-based index). If the link list does not have any loop, X=0.

Remove the loop from the linked list, if it is present, i.e. unlink the last node which is forming the loop.

**Example 1:**

Input:

N = 3

value[] = {1,3,4}

X = 2

Output:1

Explanation:The link list looks like

1 -> 3 -> 4

^ |

|\_\_\_\_|

A loop is present. If you remove it successfully, the answer will be 1.

**Example 2:**

Input:

N = 4

value[] = {1,8,3,4}

X = 0

Output:1

Explanation:The Linked list does not contains any loop.

**Example 3:**

Input:

N = 4

value[] = {1,2,3,4}

X = 1

Output:1

Explanation:The link list looks like

1 -> 2 -> 3 -> 4

^ |

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

A loop is present. If you remove it successfully, the answer will be 1.

**Solution:** Here we are going to use the Floyd Cycle Detection algorithm, where we have to detect and remove the loop. In Floyd’s algo, the slow and fast pointers meet a loop node which can be used to remove the cycle. In this approach, we use Floyd’s cycle detection and get the pointer to a loop node. Then we count the number of nodes in the loop. We fix one pointer to the head and another to the kth node from the head. Then we move both pointers at the same pace until they meet at the loop starting node. At last, the pointer will point to the last node of the loop and make it next to it NULL.

| // Javascript program to detect and // remove loop in linked list var head;   class Node {  constructor(val)  {  this.data = val;  this.next = null;  } }   // Function that detects loop in the list function detectAndRemoveLoop(node) {  var slow = node, fast = node;  while (slow != null &&  fast != null &&  fast.next != null)  {  slow = slow.next;  fast = fast.next.next;    // If slow and fast meet at same  // point then loop is present  if (slow == fast)  {  removeLoop(slow, node);  return 1;  }  }  return 0; }   // Function to remove loop function removeLoop(loop, head) {  var ptr1 = loop;  var ptr2 = loop;    // Count the number of nodes in loop  var k = 1, i;    while (ptr1.next != ptr2)  {  ptr1 = ptr1.next;  k++;  }    // Fix one pointer to head  ptr1 = head;    // And the other pointer to  // k nodes after head  ptr2 = head;  for(i = 0; i < k; i++)  {  ptr2 = ptr2.next;  }    /\* Move both pointers at the same pace,  they will meet at loop starting node \*/  while (ptr2 != ptr1)  {  ptr1 = ptr1.next;  ptr2 = ptr2.next;  }    // Get pointer to the last node  while (ptr2.next != ptr1)  {  ptr2 = ptr2.next;  }    /\* Set the next node of the loop ending node  to fix the loop \*/  ptr2.next = null; }   // Function to print the linked list function printList(node) {  while (node != null)  {  document.write(node.data + " ");  node = node.next;  } } |
| --- |

The time complexity would be O(n) and the space complexity is O(1), where n is the number of nodes in the tree.

**Question 6**

Given a linked list and two integers M and N. Traverse the linked list such that you retain M nodes then delete next N nodes, continue the same till end of the linked list.

Difficulty Level: Rookie

**Examples**:

Input:

M = 2, N = 2

Linked List: 1->2->3->4->5->6->7->8

Output:

Linked List: 1->2->5->6

Input:

M = 3, N = 2

Linked List: 1->2->3->4->5->6->7->8->9->10

Output:

Linked List: 1->2->3->6->7->8

Input:

M = 1, N = 1

Linked List: 1->2->3->4->5->6->7->8->9->10

Output:

Linked List: 1->3->5->7->9

**Solution:**

Every node has 2 properties in it, first where it stores the data and next where it stores the reference of the next node. The ‘push’ function in the given node takes the head reference and creates a new node and new data to it which also links it to the old list and moves the head reference to the new node. At last, it returns the updated head reference. The function of the main function is to skip M nodes and then delete N nodes of the linked list which takes M and N as the parameters.

The time complexity of the below code is O(n) where n is the length of the linked list. The space complexity would be O(1) as it uses a constant amount of space regardless of the size of the linked list.

| // A linked list node class Node {  constructor() {  this.data = 0;  this.next = null;  } }    /\* Function to insert a node at the beginning \*/  function push(head\_ref , new\_data) {  /\* allocate node \*/ var new\_node = new Node();    /\* put in the data \*/  new\_node.data = new\_data;    /\* link the old list of the new node \*/  new\_node.next = (head\_ref);    /\* move the head to point to the new node \*/  (head\_ref) = new\_node;    return head\_ref;  }    /\* Function to print linked list \*/  function printList(head) { var temp = head;  while (temp != null) {  document.write(temp.data+" ");  temp = temp.next;  }  document.write("<br/>");  }    // Function to skip M nodes and then  // delete N nodes of the linked list.  function skipMdeleteN(head , M , N) { var curr = head, t;  var count;    // The main loop that traverses  // through the whole list  while (curr != null) {  // Skip M nodes  for (count = 1; count < M && curr != null;   count++)  curr = curr.next;    // If we reached end of list, then return  if (curr == null)  return;    // Start from next node and delete N nodes  t = curr.next;  for (count = 1; count <= N && t != null;   count++)  {  var temp = t;  t = t.next;  }    // Link the previous list with remaining nodes  curr.next = t;    // Set current pointer for next iteration  curr = t;  }  } |
| --- |

**Question 7**

Given two linked lists, insert nodes of second list into first list at alternate positions of first list. For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, the first list should become 5->12->7->10->17->2->13->4->11->6 and second list should become empty. The nodes of second list should only be inserted when there are positions available. For example, if the first list is 1->2->3 and second list is 4->5->6->7->8, then first list should become 1->4->2->5->3->6 and second list to 7->8.

Use of extra space is not allowed (Not allowed to create additional nodes), i.e., insertion must be done in-place. Expected time complexity is O(n) where n is number of nodes in first list.

**Solution:**

The approach would be to iterate over the 2 linked lists simultaneously and keep track of the present and next nodes in both lists. One function will push all the new node and creates a link to the next node. In this way, all the nodes will be joined together

The space complexity would be O(1) as no extra space is required while the time complexity would be O(n) where n is the number of nodes.

| **// A nexted list node  class Node  {   constructor()  {  this.data = 0;  this.next = null;  } };    /\* Function to insert a node at the beginning \*/ function push(head\_ref, new\_data)  {   var new\_node = new Node();  new\_node.data = new\_data;   new\_node.next = (head\_ref);   (head\_ref) = new\_node;  return head\_ref;   }    /\* Utility function to print a singly linked list \*/ function printList(head)  {   var temp = head;   while (temp != null)   {   document.write( temp.data + " ");   temp = temp.next;   }   document.write("<br>"); }    // Main function that inserts nodes of linked list q into p at  // alternate positions. Since head of first list never changes  // and head of second list may change, we need single pointer  // for first list and double pointer for second list.  function merge(p, q)  {   var p\_curr = p, q\_curr = q;   var p\_next, q\_next;     // While there are available positions in p   while (p\_curr != null && q\_curr != null)   {   // Save next pointers   p\_next = p\_curr.next;   q\_next = q\_curr.next;     // Make q\_curr as next of p\_curr   q\_curr.next = p\_next; // Change next pointer of q\_curr   p\_curr.next = q\_curr; // Change next pointer of p\_curr     // Update current pointers for next iteration   p\_curr = p\_next;   q\_curr = q\_next;   }     q = q\_curr; // Update head pointer of second list   return q; }** |
| --- |

**Question 8**

Given a singly linked list, find if the linked list is [circular](https://www.geeksforgeeks.org/circular-linked-list/amp/) or not.



**Solution:** In general empty linked list is also considered circular. The approach is to store the head of the linked list and traverse it. While iterating if it reaches null, the list is not circular else if it reaches the head again, we can call it circular.

We will be declaring a node pointer and initializing to the head’s next. Then we move the node pointer to the next node until we found the null or the head of the loop. After coming out we check whether the node is equal to the head and returns true and false accordingly.

| // structure of linked list node class Node{  constructor(data){  this.data = data;  this.next = null;  } } let head = null; function addToFront(data){  let newNode = new Node(data);  newNode.next = head;  head = newNode; }   function isCircular(){  if(head == null) return false;  let slow = head;  let fast = head.next;  while(fast != null && fast.next != null){  if(slow == fast) return true;  slow = slow.next;  fast = fast.next.next;  }  return false; } |
| --- |

The time complexity would be O(n) where n is the number of nodes and space complexity is O(1) as no extra space is required.