Lesson Plan: Linked List-1

Today's Checklist:

1. Limitations of Arrays
2. Introduction to Linked List
3. Implementing Linked List
4. Displaying
5. Insert in Linked List
6. Limitation
7. Delete in Linked List

Limitations of Arrays

1. **Fixed Size:** Most arrays have a fixed size, meaning you need to know the number of elements in advance. This can be a limitation when the size of the data is dynamic and unknown beforehand.

2. **Contiguous Memory Allocation:** Elements in an array are stored in contiguous memory locations. This can lead to fragmentation and might make it difficult to find a large enough block of memory for the array.

3. **Inefficient Insertions and Deletions:** Inserting or deleting elements in the middle of an array requires shifting all subsequent elements, which can be inefficient. The time complexity for these operations is O(n), where n is the number of elements.

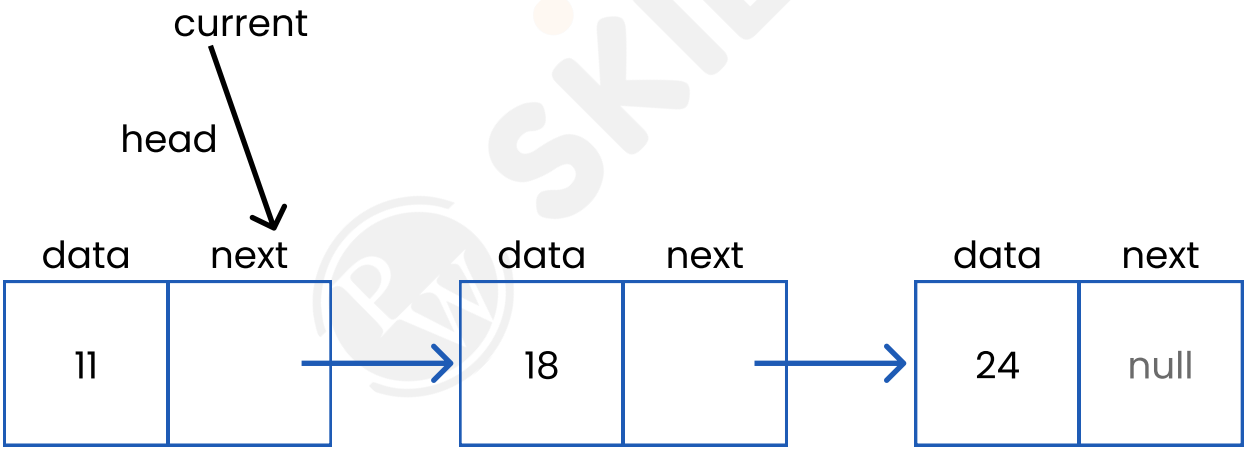
4. **Wastage of Memory:** If you allocate more space than needed for an array, you may end up wasting memory. This is particularly problematic when the array size is predetermined to accommodate the worst-case scenario.

5. **Homogeneous Data Types:** Arrays typically store elements of the same data type. This can be limiting when you need to store elements of different types.

6. **Memory Fragmentation:** The contiguous memory allocation can lead to memory fragmentation, making it challenging to allocate a large contiguous block of memory for a new array.

Introduction to Linked List

It is basically chains of nodes, each node contains information such as data and a pointer to the next node in the chain. In the linked list there is a head pointer, which points to the first element of the linked list, and if the list is empty then it simply points to null or nothing.



Why linked list data structure needed?

- Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.

- Ease of Insertion/Deletion: The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.

- Efficient Memory Utilization: As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.

- Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Implementing Linked List

A linked list can be implemented in various ways, but the basic structure involves nodes, where each node contains data and a reference (or link) to the next node in the sequence.

A step-by-step explanation of how to implement a simple singly linked list:

1. **Node Class:** Create a class for the linked list node with data and a pointer to the next node.

```
class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};
```

2. **LinkedList Class:** Create a class to represent the linked list. Include a pointer to the head of the list.

```
class LinkedList {
private:
    Node* head;

public:
    LinkedList() : head(nullptr) {}
};
```

3. **Add Nodes:**
   - Implement a method to add nodes to the linked list.
   - If the list is empty, create a new node and set it as the head.
   - Otherwise, traverse to the end of the list and add a new node.

```
void addNode(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* current = head;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newNode;
    }
}
```

Displaying

Once, we have created the linked list, we would like to see the elements inside it. Displaying a linked list involves iterating through its nodes and printing their data. This can also be done recursively as shown in the code below.

Code:

```
#include <iostream>
using namespace std;

class Node {
public:
```

```
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

void displayLinkedListIterative(Node* head) {
    while (head ≠ nullptr) {
        cout << head→data << " ";
        head = head→next;
    }
    cout << endl;
}

void displayLinkedListRecursive(Node* head) {
    if (head == nullptr)
        return;

    cout << head→data << " ";
    displayLinkedListRecursive(head→next);
}

int main() {
    Node* head = new Node(1);
    head→next = new Node(2);
    head→next→next = new Node(3);
    head→next→next→next = new Node(4);

    cout << "Iterative Display: ";
    displayLinkedListIterative(head);

    cout << "Recursive Display: ";
    displayLinkedListRecursive(head);
    cout << endl;

    return 0;
}
```

Explanation:

Iterative Display:

1. Initialize a pointer to the head of the linked list.
2. Use a while loop to traverse the list.
3. Print the data of each node and move the pointer to the next node.
4. Repeat until the end of the list is reached.

Recursive Display:

1. Base case: If the current node is null, return.
2. Print the data of the current node.
3. Make a recursive call with the next node.
4. The recursion unwinds, printing nodes in sequential order.
5. Base case ensures termination when the end of the list is reached.

Iterative Display:

Time Complexity: O(n) - where 'n' is the number of nodes in the linked list. The algorithm iterates through each node once.

Space Complexity: O(1) - uses a constant amount of extra space, regardless of the size of the linked list.

Recursive Display:

Time Complexity: O(n) - where 'n' is the number of nodes in the linked list. Similar to the iterative approach, each node is visited once, but the recursive call stack contributes to the time complexity.

Space Complexity: O(n) - due to the recursive call stack. The maximum depth of the recursion is 'n', corresponding to the length of the linked list.

MCQ: What will this function do?

```
void display(Node head) {
if(head == null)
return;
display(head→next);
cout<<head.val<" ";
}
```

a) Print all the elements of the linked list.
b) Print all the elements except last one.
c) Print alternate nodes of linked list
d) Print all the nodes in reverse order

Ans: Print all the nodes in reverse order.

Explanation:

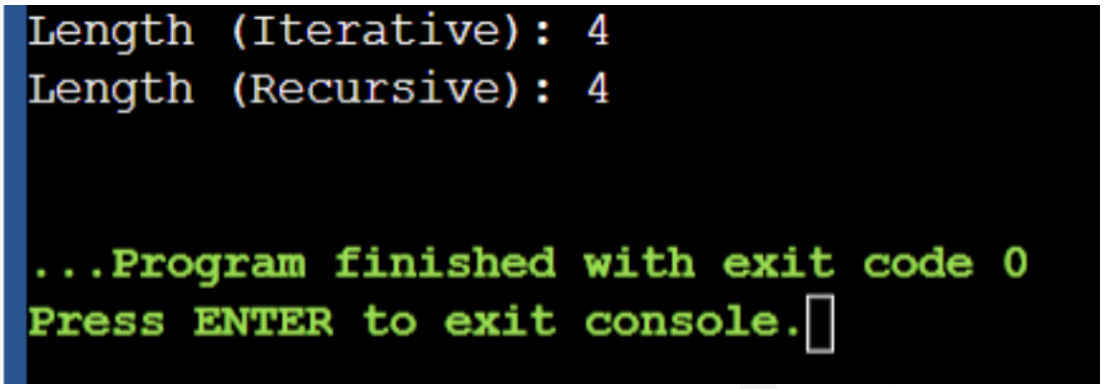
1. The given function is a recursive function that traverses the linked list using a recursive call (display(head->next)) before printing the value of the current node (cout<<head.val<<" ").
2. The base case (if(head == null) return;) ensures that the recursion stops when the end of the linked list is reached.
3. As the recursion unfolds, the values of the nodes are printed in reverse order, starting from the last node and moving towards the head of the linked list.

Length of Linked List

Code:

```
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

int findLengthIterative(Node* head) {
    int length = 0;
    while (head ≠ nullptr) {
        length++;
        head = head→next;
    }
    return length;
}

int findLengthRecursive(Node* head) {
    if (head == nullptr)
        return 0;
    return 1 + findLengthRecursive(head→next);
}

int main() {
    Node* head = new Node(1);
    head→next = new Node(2);
    head→next→next = new Node(3);
    head→next→next→next = new Node(4);
```

```
    std::cout << "Length (Iterative): " <<
findLengthIterative(head) << std::endl;
    std::cout << "Length (Recursive): " <<
findLengthRecursive(head) << std::endl;

    return 0;
}
```



Iterative Method:
Time Complexity: O(n) - where 'n' is the number of nodes in the linked list. In the worst case, it needs to traverse all nodes once.

Space Complexity: O(1) - uses a constant amount of extra space.

Recursive Method:
Time Complexity: O(n) - where 'n' is the number of nodes in the linked list. Similar to the iterative approach, it needs to visit each node once.

Space Complexity: O(n) - due to the recursive call stack. The maximum depth of the recursion is 'n', corresponding to the length of the linked list.

Insert in Linked List

The insertion operation can be performed in three ways. They are as follows...

1. Inserting At the Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Code:

```
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

// Insert at the Beginning of the list
Node* insertAtBeginning(Node* head, int value) {
    Node* newNode = new Node(value);
    newNode->next = head;
    return newNode;
}

// Insert at End of the list
Node* insertAtEnd(Node* head, int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        return newNode;
    }
    Node* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = newNode;
    return head;
}

// Insert at Specific location in the list
Node* insertAtLocation(Node* head, int value, int position)
{
    Node* newNode = new Node(value);
    if (position == 1) {
        newNode->next = head;
        return newNode;
    }
    Node* current = head;
    for (int i = 1; i < position - 1 && current != nullptr;
++i) {
        current = current->next;
    }
```

```
    if (current == nullptr) {
        cout << "Invalid position." << endl;
        return head;
    }
    newNode->next = current->next;
    current->next = newNode;
    return head;
}

// Display the linked list
void displayLinkedList(Node* head) {
    while (head != nullptr) {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

int main() {
    Node* head = nullptr;

    // Insert at the Beginning
    head = insertAtBeginning(head, 1);
    displayLinkedList(head);

    // Insert at the End
    head = insertAtEnd(head, 3);
    displayLinkedList(head);

    // Insert at Specific location
    head = insertAtLocation(head, 2, 2);
    displayLinkedList(head);

    return 0;
}
```

Insert at the Beginning:

1. Create a New Node: Allocate memory for a new node and set its data.
2. Link to Current Head: Set the next pointer of the new node to the current head.
3. Update Head: Set the new node as the new head of the linked list.

Insert at the End:

1. Create a New Node: Allocate memory for a new node and set its data.
2. Traverse to the Last Node: Iterate through the linked list until the last node is reached.
3. Link to Last Node: Set the next pointer of the last node to the new node.

Insert at Specific Location:

1. Create a New Node: Allocate memory for a new node and set its data.
2. Handle Special Case (Insert at the Beginning): If the position is 1, link the new node to the current head and update the head.
3. Traverse to the Previous Node: Iterate through the list to the node preceding the desired position.
4. Link the New Node: Set the next pointer of the new node to the next node of the previous node, and set the next pointer of the previous node to the new node.

Time and Space Complexity:

1. Insert at the Beginning:

   Time Complexity: O(1)
   Space Complexity: O(1)

2. Insert at the End:

   Time Complexity: O(n) - in the worst case
   Space Complexity: O(1)

3. Insert at Specific Location:

   Time Complexity: O(position) - in the worst case
   Space Complexity: O(1)

Delete in Linked List

The deletion operation can be performed in three ways. They are as follows:

1. Deleting from the Beginning of the list
2. Deleting from the End of the list
3. Deleting a Specific Node

Code:

```
#include <iostream>
using namespace std;
```

```
class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

// Delete from the Beginning of the list
Node* deleteFromBeginning(Node* head) {
    if (head == nullptr) {
        cout << "List is empty. Cannot delete." << endl;
        return nullptr;
    }
    Node* newHead = head->next;
    delete head;
    return newHead;
}

// Delete from the End of the list
Node* deleteFromEnd(Node* head) {
    if (head == nullptr) {
        cout << "List is empty. Cannot delete." << endl;
        return nullptr;
    }
    if (head->next == nullptr) {
        delete head;
        return nullptr;
    }
    Node* current = head;
    while (current->next->next != nullptr) {
        current = current->next;
    }
    delete current->next;
    current->next = nullptr;
    return head;
}

// Delete a Specific Node
Node* deleteNode(Node* head, int value) {
    if (head == nullptr) {
        cout << "List is empty. Cannot delete." << endl;
        return nullptr;
    }
    if (head->data == value) {
        Node* newHead = head->next;
        delete head;
        return newHead;
```

```
    }
    Node* current = head;
    while (current->next != nullptr && current->next->data
!= value) {
        current = current->next;
    }
    if (current->next == nullptr) {
        cout << "Node with value " << value << " not found."
 << endl;
        return head;
    }
    Node* temp = current->next;
    current->next = current->next->next;
    delete temp;
    return head;
}

// Display the linked list
void displayLinkedList(Node* head) {
    while (head != nullptr) {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}
int main() {
    Node* head = nullptr;

    // Delete from the Beginning
    head = deleteFromBeginning(head);
    displayLinkedList(head);

    // Delete from the End
    head = deleteFromEnd(head);
    displayLinkedList(head);

    // Delete a Specific Node
    head = deleteNode(head, 2);
    displayLinkedList(head);

    return 0;
}


Certainly! Below is a C++ code snippet that demonstrates
deletion operations in a linked list:
```

```
cpp
Copy code
#include <iostream>

using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

// Delete from the Beginning of the list
Node* deleteFromBeginning(Node* head) {
    if (head == nullptr) {
        cout << "List is empty. Cannot delete." << endl;
        return nullptr;
    }
    Node* newHead = head->next;
    delete head;
    return newHead;
}

// Delete from the End of the list
Node* deleteFromEnd(Node* head) {
    if (head == nullptr) {
        cout << "List is empty. Cannot delete." << endl;
        return nullptr;
    }
    if (head->next == nullptr) {
        delete head;
        return nullptr;
    }
    Node* current = head;
    while (current->next->next != nullptr) {
        current = current->next;
    }
    delete current->next;
    current->next = nullptr;
    return head;
}

// Delete a Specific Node
Node* deleteNode(Node* head, int value) {
```

```
    if (head == nullptr) {
        cout << "List is empty. Cannot delete." << endl;
        return nullptr;
    }
    if (head->data == value) {
        Node* newHead = head->next;
        delete head;
        return newHead;
    }
    Node* current = head;
    while (current->next != nullptr && current->next->data
!= value) {
        current = current->next;
    }
    if (current->next == nullptr) {
        cout << "Node with value " << value << " not found."
 << endl;
        return head;
    }
    Node* temp = current->next;
    current->next = current->next->next;
    delete temp;
    return head;
}

// Display the linked list
void displayLinkedList(Node* head) {
    while (head != nullptr) {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

int main() {
    Node* head = nullptr;

    // Delete from the Beginning
    head = deleteFromBeginning(head);
    displayLinkedList(head);

    // Delete from the End
    head = deleteFromEnd(head);
    displayLinkedList(head);
```

```
    // Delete a Specific Node
    head = deleteNode(head, 2);
    displayLinkedList(head);

    return 0;
}
```

Explanation:

Delete from the Beginning:

Check if the list is empty. If not, delete the current head and set the next node as the new head.

Delete from the End:

Check if the list is empty or has only one node. If not, traverse to the second-to-last node, delete the last node, and set the next pointer of the second-to-last node to null.

Delete a Specific Node:

Check if the list is empty. If not, traverse the list to find the node with the specified value. Delete the node by adjusting pointers.

Time and Space Complexity:

1. Delete from the Beginning:

   Time Complexity: O(1)
   Space Complexity: O(1)

2. Delete from the End:

   Time Complexity: O(n) - in the worst case
   Space Complexity: O(1)

3. Delete a Specific Node:

   Time Complexity: O(n) - in the worst case
   Space Complexity: O(1)

Lesson Plan: Linked List-2

Today's Checklist:

1. Delete Node in a Linked List (Leetcode-237)
2. Middle of Linked List (Leetcode-876)
3. Remove Nth Node from End of List (Leetcode-19)
4. Intersection of two Linked Lists (Leetcode-160)
5. Linked List Cycle (Leetcode-141)
6. Linked List Cycle-II (Leetcode-142)

Delete Node in a Linked List (Leetcode-237)

There is a singly-linked list head and we want to delete a node node in it.
You are given the node to be deleted node. You will not be given access to the first node of head.
All the values of the linked list are unique, and it is guaranteed that the given node node is not the last node in the linked list.

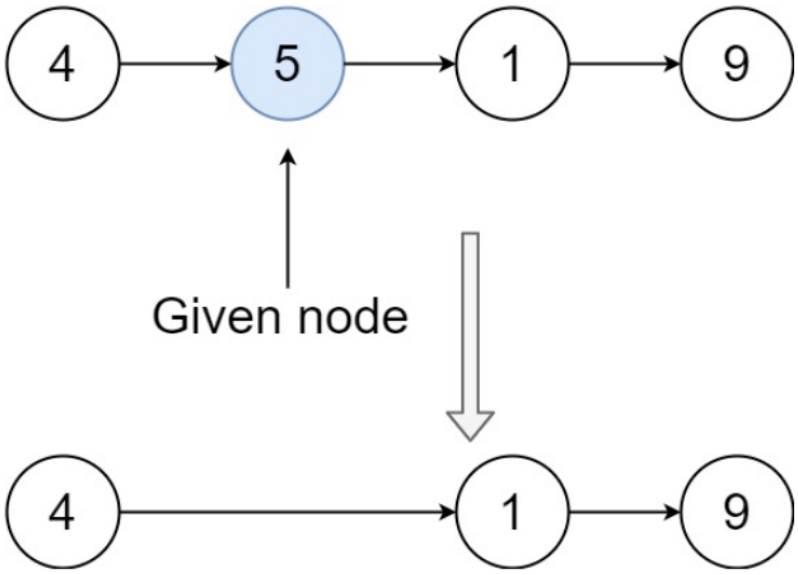
Delete the given node. Note that by deleting the node, we do not mean removing it from memory. We mean:

- The value of the given node should not exist in the linked list.
- The number of nodes in the linked list should decrease by one.
- All the values before node should be in the same order.
- All the values after node should be in the same order.

Input: head = [4,5,1,9], node = 5

Output: [4,1,9]

Explanation: You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.



Code:

```
class Solution {
public:
    void deleteNode(ListNode* node) {
        if (node ≠ NULL && node→next ≠ NULL) {
            node→val = node→next→val;
            node→next = node→next→next;
        }
    }
};
```

Time complexity: O(1) - Constant time complexity. The deletion operation involves updating the value of the given node with the value of its next node and then updating the next pointer to skip the next node. These operations are constant time.

Space complexity: O(1) - Constant space complexity. The algorithm uses only a constant amount of extra space, regardless of the size of the input linked list.

Middle of Linked List (Leetcode-876)

Given the head of a singly linked list, return the middle node of the linked list.
If there are two middle nodes, return the second middle node.

Input: head = [1,2,3,4,5]

Output: [3,4,5]

Explanation: The middle node of the list is node 3.

Code:

```
class Solution {
public:
    void deleteNode(ListNode* node) {
        if (node ≠ NULL && node→next ≠ NULL) {
            node→val = node→next→val;
            node→next = node→next→next;
        }
    }
};
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through the linked list with two pointers (slow and fast), and in each iteration, the fast pointer moves two steps while the slow pointer moves one step.

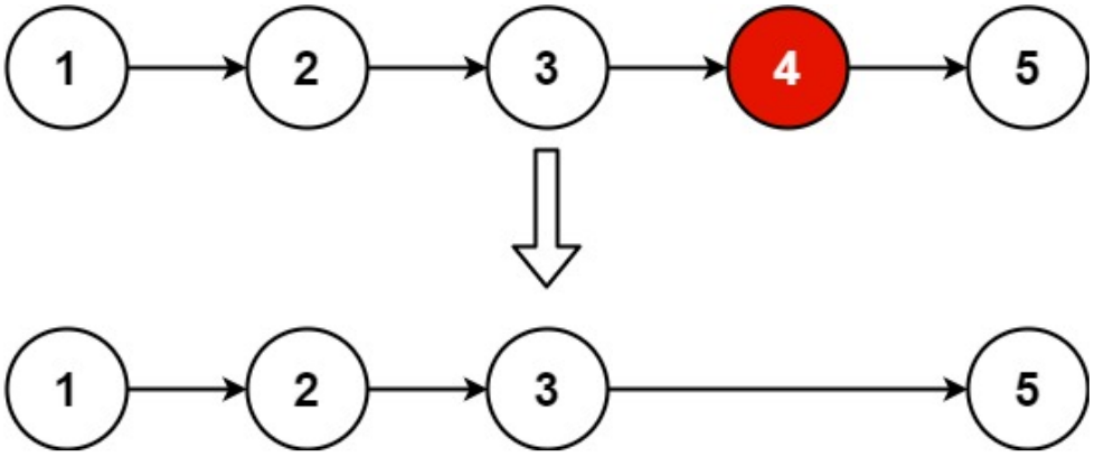
Space complexity: O(1). The algorithm uses only a constant amount of extra space for the two pointers, regardless of the size of the input linked list.

Remove Nth Node from End of List (Leetcode-19)

Given the head of a linked list, remove the nth node from the end of the list and return its head.

Input: head = [1,2,3,4,5], n = 2

Output: [1,2,3,5]



Code:

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode *fast = head, *slow = head;
        for (int i = 0; i < n; i++) fast = fast->next;
        if (!fast) return head->next;
        while (fast->next) fast = fast->next, slow = slow
->next;
        slow->next = slow->next->next;
        return head;
    }
};
```

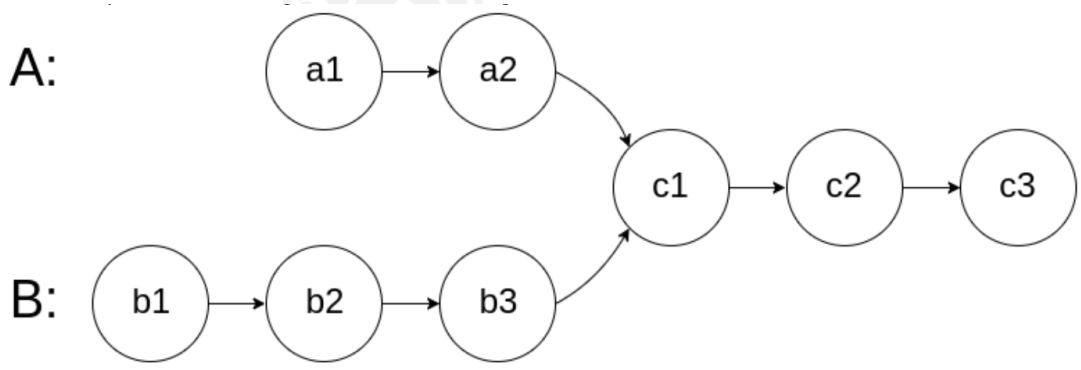
Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm uses two pointers to traverse the linked list. The first loop advances the fast pointer by n nodes, and then the second loop advances both pointers until the fast pointer reaches the end.

Space complexity: O(1). The algorithm uses only a constant amount of extra space for the two pointers, regardless of the size of the input linked list.

Intersection of two Linked Lists (Leetcode-160)

Given the heads of two singly linked-lists headA and headB, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return null.

For example, the following two linked lists begin to intersect at node c1:

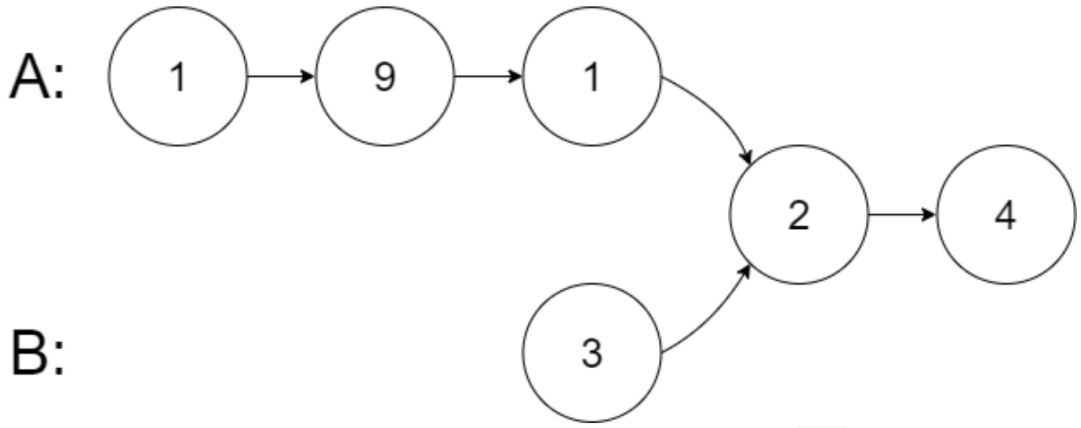


Input: intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1

Output: Intersected at '2'

Explanation: The intersected node's value is 2 (note that this must not be 0 if the two lists intersect).

From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4]. There are 3 nodes before the intersected node in A; There are 1 node before the intersected node in B.




```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *head1, ListNode
*head2) {
        if(head1 == NULL || head2 == NULL)return NULL;

        ListNode *a = head1, *b = head2;

        while(a != b){
            a = (a == NULL)? head2 : a->next;
            b = (b == NULL)? head1 : b->next;
        }
        return a;
    }
};
```

Time complexity: O(m + n), where m and n are the lengths of the two linked lists. The pointers traverse the linked lists once, and the loop continues until either the intersection is found or both pointers reach the end.

Space complexity: O(1). The algorithm uses only a constant amount of extra space for the two pointers (a and b), regardless of the size of the linked lists.

Linked List Cycle (Leetcode-141)

Given head, the head of a linked list, determine if the linked list has a cycle in it.

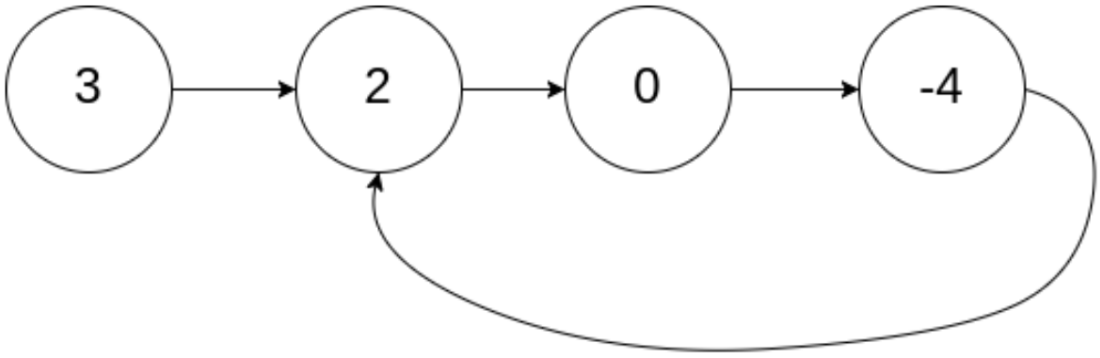
There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Return true if there is a cycle in the linked list. Otherwise, return false.

Input: head = [3,2,0,-4], pos = 1

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).



Code:

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *slow_pointer = head, *fast_pointer = head;
        while (fast_pointer ≠ nullptr && fast_pointer→next
≠ nullptr) {
            slow_pointer = slow_pointer→next;
            fast_pointer = fast_pointer→next→next;
            if (slow_pointer == fast_pointer) {
                return true;
            }
        }
        return false;
    }
};
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm has at most two pointers traversing the linked list, and the loop will continue until the fast pointer reaches the end or the two pointers meet in a cycle.

Space complexity: O(1). The algorithm uses only a constant amount of extra space for the two pointers (slow_pointer and fast_pointer), regardless of the size of the linked list.

Linked List Cycle-II (Leetcode-142)

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

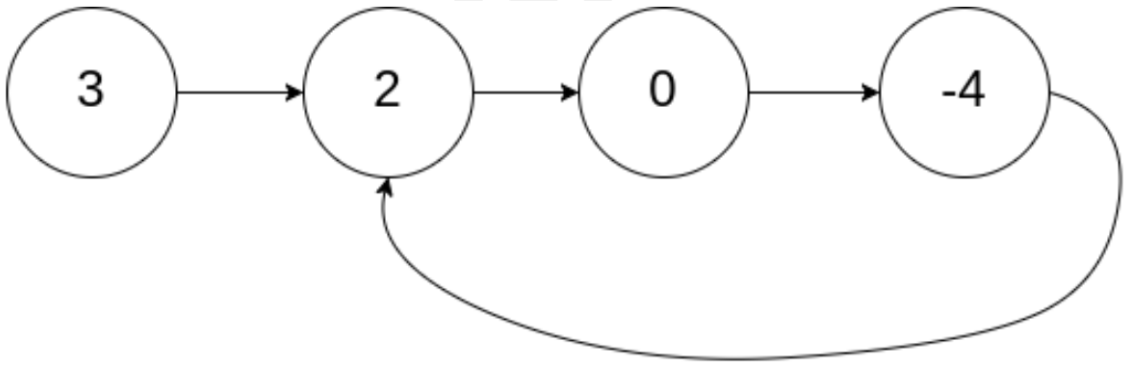
There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

Do not modify the linked list.

Input: head = [3,2,0,-4], pos = 1

Output: tail connects to node index 1

Explanation: There is a cycle in the linked list, where tail connects to the second node.



Code:

```
class Solution {
 public:
  ListNode* detectCycle(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;
```

```
    while (fast && fast->next) {
      slow = slow->next;
      fast = fast->next->next;
      if (slow == fast) {
              slow = head;
        while (slow != fast) {
          slow = slow->next;
          fast = fast->next;
        }
        return slow;
      }
    }
    return nullptr;
  }
};
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm uses two pointers, slow and fast, to traverse the linked list. In the worst case, it needs to iterate through the entire linked list once.

Space complexity: O(1). The algorithm uses only a constant amount of extra space for the two pointers (slow and fast), regardless of the size of the linked list. It doesn't use any additional data structures that scale with the input size.

Lesson Plan: Linked List-3

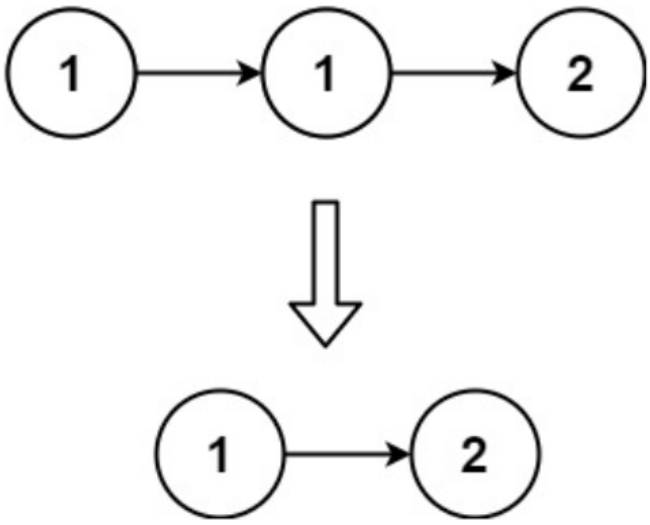
Today's Checklist:

1. Remove Duplicates from the Sorted List (Leetcode-83)
2. Rotate List (Leetcode-61)
3. Spiral Matrix IV (Leetcode-2326)
4. Merge 2 sorted lists (Leetcode-21)
5. Merge k sorted lists (Leetcode-23)
6. Sort List (Leetcode-148)
7. Partition List (Leetcode-86)
8. Reverse Linked List (Leetcode-206)
9. Palindrome Linked List (Leetcode-234)
10. Reverse Linked List II (Leetcode-92)
11. Reorder List (Leetcode-143)

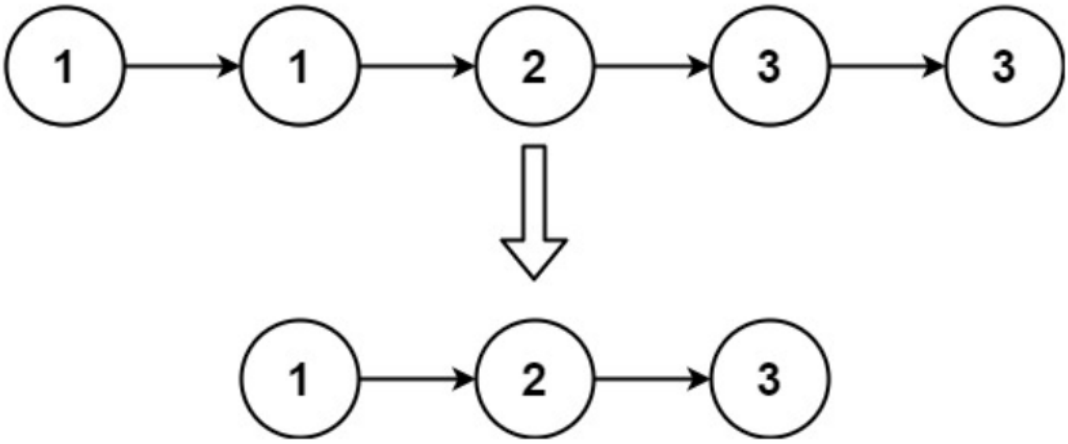
Remove Duplicates from the Sorted List (Leetcode-83)

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

Input: head = [1,1,2]
Output: [1,2]



Input: head = [1,1,2,3,3]
Output: [1,2,3]



Code:

```
ListNode* deleteDuplicates(ListNode* head) {

 if(!head) return head;
 ListNode* tmp = head;

 while(tmp && tmp -> next)
 {
  if(tmp -> val == tmp -> next -> val)
   tmp -> next = tmp -> next -> next;
  else
   tmp = tmp -> next;
 }

 return head;
}
```

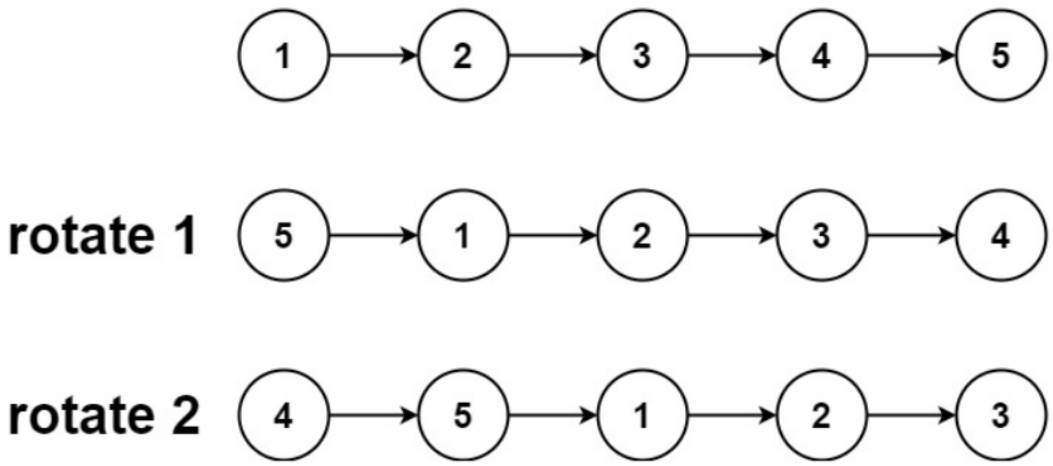
Time complexity: O(n) - where n is the number of nodes in the linked list.

Space complexity: O(1) - constant space is used, as no additional data structures are employed in the algorithm.

Rotate List (Leetcode-61)

Given the head of a linked list, rotate the list to the right by k places.

Input: head = [1,2,3,4,5], k = 2
Output: [4,5,1,2,3]



Code:

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {

        if(head==NULL || head->next==NULL || k==0)return
head;

        ListNode *temp=head;
        int size=0;
        while(temp!=NULL)
        {
            temp = temp->next;
            size++;
        }
        k = k%size;
        if(k==0)return head;

        while(k--){
        ListNode *prev=head,*curr=head->next;
        while(curr->next!=NULL){
            curr=curr->next;
            prev=prev->next;
        }
        curr->next=head;
        prev->next=NULL;
        head=curr;
    }
        return head;
    }
};
```

Time complexity: O(n) - where n is the number of nodes in the linked list. The algorithm iterates through the list twice: once to calculate the length and once to find the new head.

Space complexity: O(1) - constant space is used, as only a constant number of pointers are used regardless of the size of the input linked list.

Spiral Matrix IV (Leetcode-2326)

You are given two integers m and n, which represent the dimensions of a matrix.

You are also given the head of a linked list of integers.

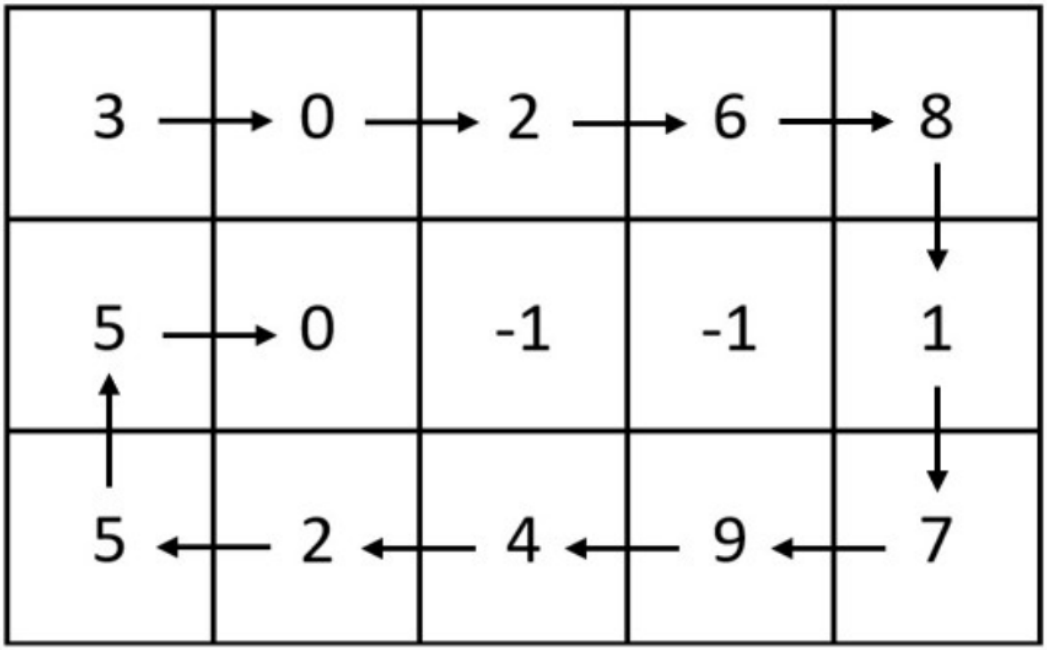
Generate an m x n matrix that contains the integers in the linked list presented in spiral order (clockwise), starting from the top-left of the matrix. If there are remaining empty spaces, fill them with -1.

Return the generated matrix.

Input: m = 3, n = 5, head = [3,0,2,6,8,1,7,9,4,2,5,5,0]

Output: [[3,0,2,6,8],[5,0,-1,-1,1],[5,2,4,9,7]]

Explanation: The diagram above shows how the values are printed in the matrix.
Note that the remaining spaces in the matrix are filled with -1.



Code:

```
class Solution {
 public:
  vector<vector<int>> spiralMatrix(int m, int n, ListNode*
head) {
   vector<vector<int>> mat(m,vector<int>(n,-1));
   ListNode* curr=head;
   int minr=0;
   int minc=0;
   int maxr=m-1;
   int maxc=n-1;
   while(curr≠NULL){
    for(int j=minc;j⩽maxc && curr≠NULL;j++){
     int i=minr;
     mat[i][j]=curr→val;
     curr=curr→next;
    }
    minr++;
    for(int i=minr;i⩽maxr && curr≠NULL;i++){
     int j=maxc;
     mat[i][j]=curr→val;
     curr=curr→next;
    }
    maxc--;
    for(int j=maxc;j⩾minc && curr≠NULL;j--){
     int i=maxr;
     mat[i][j]=curr→val;
     curr=curr→next;
    }
    maxr--;
    for(int i=maxr;i⩾minr && curr≠NULL;i--){
     int j=minc;
     mat[i][j]=curr→val;
     curr=curr→next;
    }
    minc++;
   }
   return mat;
  }
 };
```

Time complexity: O(m * n) - where m is the number of rows and n is the number of columns in the resulting matrix. The algorithm iterates through each cell in the matrix to fill it with values from the linked list.

Space complexity: O(m * n) - the space used by the result matrix. The space complexity is determined by the size of the output matrix, which is m x n.

Merge 2 sorted lists (Leetcode-21)

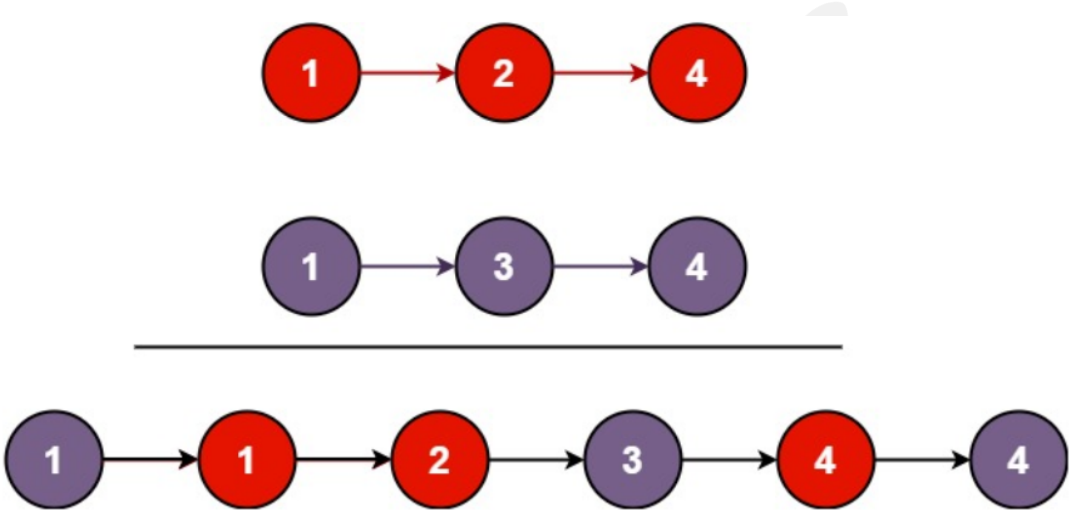
You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]



Code:

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if(l1 == NULL) return l2;
        if(l2 == NULL) return l1;
        if(l1→val⩾l2→val) l2→next = mergeTwoLists(l1,
l2→ next);
        else{
            l1→next = mergeTwoLists(l1→next, l2);
            l2 = l1;
        }return l2;
    }
```

Time complexity: O(m + n) where m and n are the lengths of the two linked lists, l1 and l2. The function recursively traverses through each node in the linked lists.

Space complexity: O(m + n) due to the recursive stack. In the worst case, the maximum depth of the recursion would be the length of the longer linked list between l1 and l2.

Merge k sorted lists (Leetcode-23)

You are given an array of k linked-lists lists, each linked list is sorted in ascending order. Merge all the linked lists into one sorted linked list and return it.

Input: lists = [[1,4,5],[1,3,4],[2,6]]

Output: [1,1,2,3,4,4,5,6]

Explanation: The linked lists are:
[
 1->4->5,
 1->3->4,
 2->6
]
merging them into one sorted list:
1->1->2->3->4->4->5->6

Code:

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) {
            return nullptr;
        }
        return mergeKListsHelper(lists, 0, lists.size() -
1);
    }

    ListNode* mergeKListsHelper(vector<ListNode*>& lists,
int start, int end) {
        if (start == end) {
            return lists[start];
```

```
        }
        if (start + 1 == end) {
            return merge(lists[start], lists[end]);
        }
        int mid = start + (end - start) / 2;
        ListNode* left = mergeKListsHelper(lists, start,
mid);
        ListNode* right = mergeKListsHelper(lists, mid + 1,
end);
        return merge(left, right);
    }

    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0);
        ListNode* curr = dummy;

        while (l1 && l2) {
            if (l1->val < l2->val) {
                curr->next = l1;
                l1 = l1->next;
            } else {
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
        }

        curr->next = l1 ? l1 : l2;

        return dummy->next;
    }
};
```

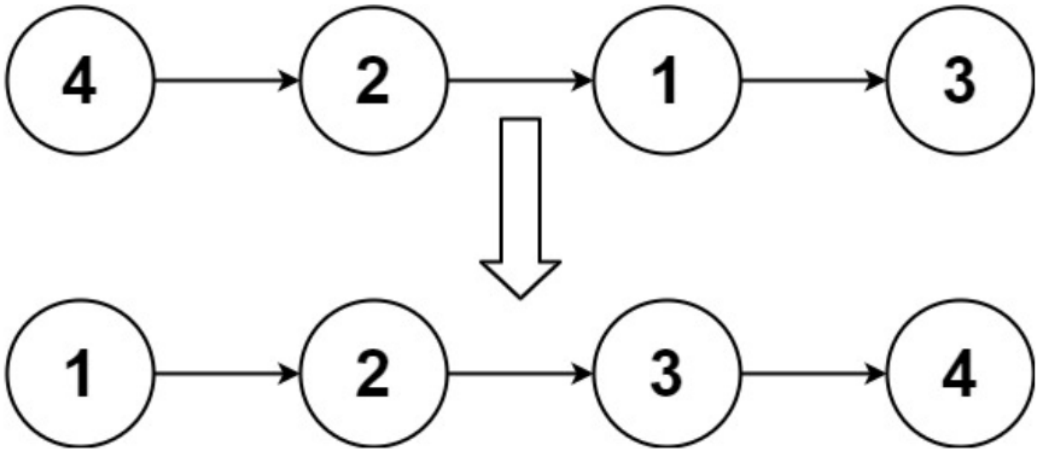
Time complexity: O(N log k), where N is the total number of nodes in all linked lists, and k is the number of linked lists. The mergeKLists function uses a divide-and-conquer strategy to merge the lists, and at each level of the recursion, it performs a linear-time merge operation.

Space complexity: O(log k), where k is the number of linked lists. This is the space used by the recursive call stack. In the worst case, the maximum depth of the recursion would be log k.

Sort List (Leetcode-148)

Given the head of a linked list, return the list after sorting it in ascending order.

Input: head = [4,2,1,3]
Output: [1,2,3,4]



Code:

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        // Base case: if the list is empty or has only one
element, it is already sorted
        if (head == nullptr || head->next == nullptr)
            return head;

        // Use slow and fast pointers to find the middle of
the list
        ListNode* slow = head;
        ListNode* fast = head->next;

        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
        }

        // Divide the list into two parts
        fast = slow->next;
        slow->next = nullptr;

        // Recursively sort each half and then merge them
        return merge(sortList(head), sortList(fast));
    }
```

```
    // Merge two sorted lists
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode dump(0);
        ListNode* cur = &dump;

        // Traverse both lists and merge them
        while (l1 ≠ nullptr && l2 ≠ nullptr) {
            if (l1→val < l2→val) {
                cur→next = l1;
                l1 = l1→next;
            } else {
                cur→next = l2;
                l2 = l2→next;
            }
            cur = cur→next;
        }

        // Attach the remaining elements of the non-empty
list
        if (l1 ≠ nullptr)
            cur→next = l1;
        else
            cur→next = l2;

        return dump.next;
    }
};
```

Time complexity: O(n log n), where n is the number of nodes in the linked list. This is because the sortList function recursively divides the list into halves, and the merge function performs a linear-time merge operation at each level of the recursion.

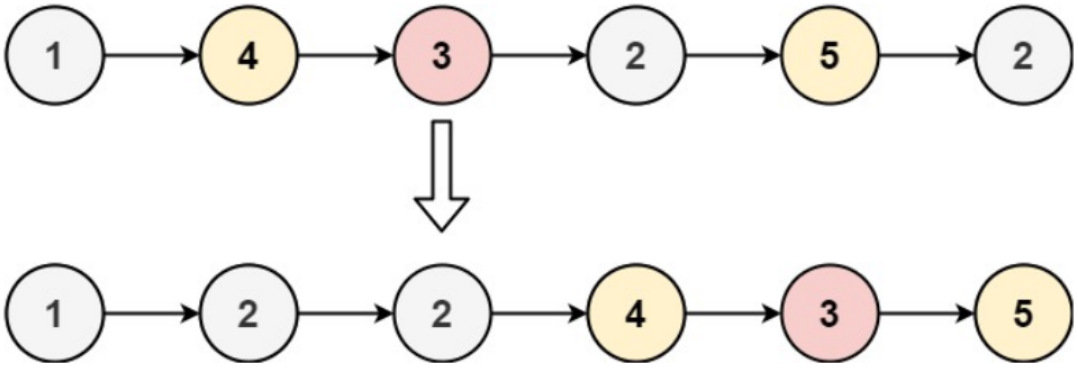
Space complexity: O(log n), where n is the number of nodes in the linked list. This is the space used by the recursive call stack. In the worst case, the maximum depth of the recursion would be log n.

Partition List (Leetcode-86)

Given the head of a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

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

Input: head = [1,4,3,2,5,2], x = 3
Output: [1,2,2,4,3,5]



Code:

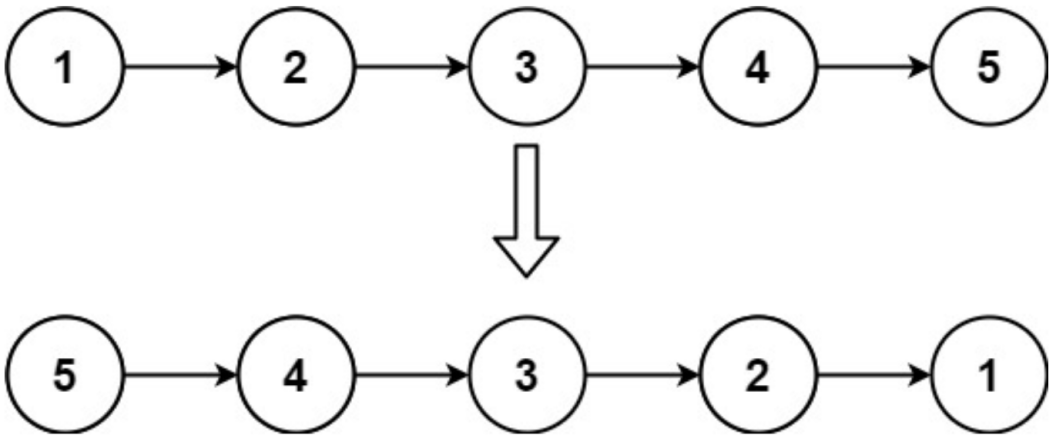
```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode *left = new ListNode(0);
        ListNode *right = new ListNode(0);

        ListNode *leftTail = left;
        ListNode *rightTail = right;

        while(head != NULL){
            if(head->val < x){
                leftTail->next = head;
                leftTail = leftTail->next;
            }
            else{
                rightTail->next = head;
                rightTail = rightTail->next;
            }
            head = head->next;
        }

        leftTail->next = right->next;
        rightTail->next = NULL;

        return left->next;
    }
};
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through each node once and performs constant-time operations.

Space complexity: O(1). The algorithm uses a constant amount of extra space to store the left and right partitions, regardless of the size of the input linked list.

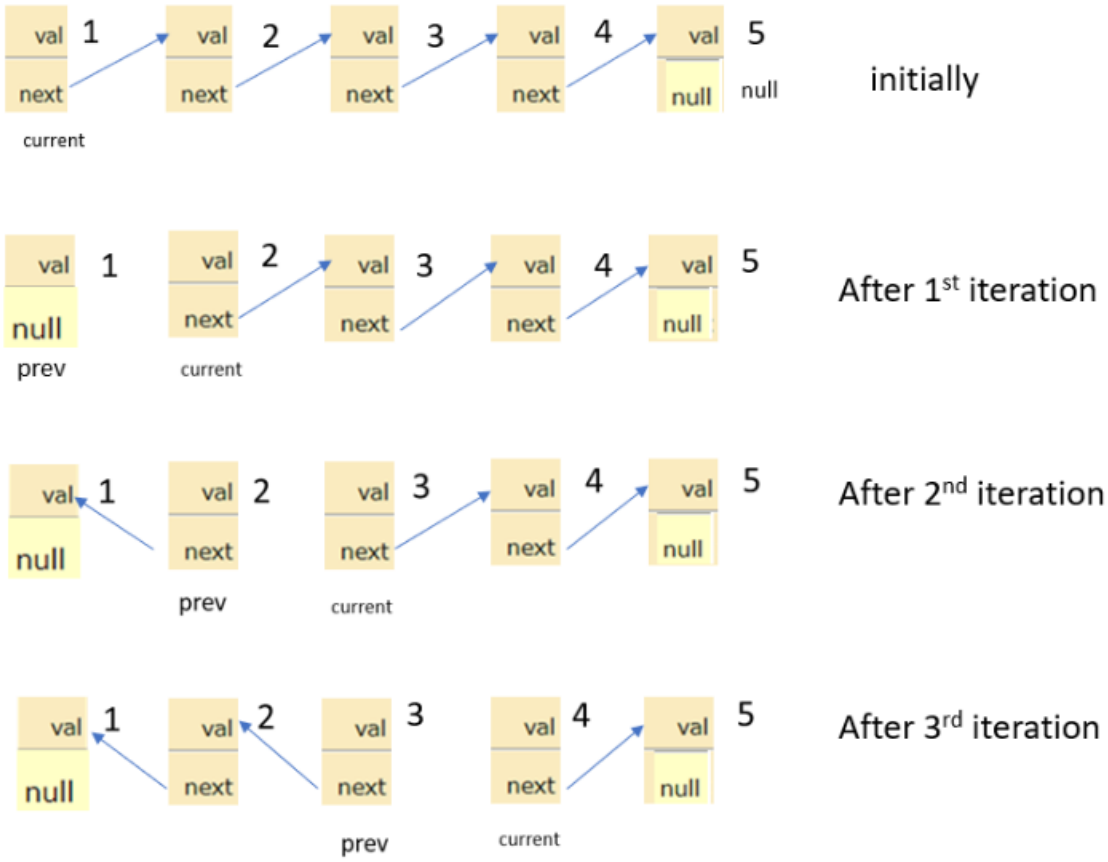
Reverse Linked List (Leetcode-206)

Given the head of a singly linked list, reverse the list, and return the reversed list.



Code:

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = NULL;

        while(head) {
            ListNode* nxt = head→next;
            head→next = prev;
            prev = head;
            head = nxt;
        }
        return prev;
    }
};
```



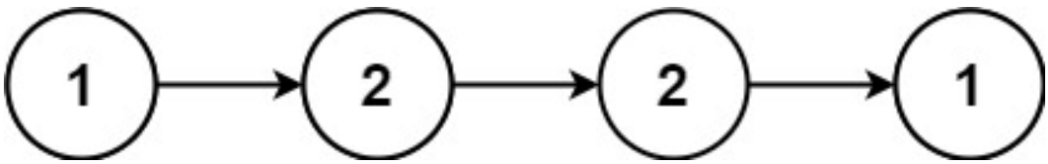
Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through each node once, performing constant-time operations in each iteration.

Space complexity: O(1). The algorithm uses a constant amount of extra space for the three-pointers (prev, current, next), regardless of the size of the input linked list.

Palindrome Linked List (Leetcode-234)

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

Input: head = [1,2,2,1]
Output: true



Code:

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        ListNode *slow = head, *fast = head, *prev, *temp;
        while (fast && fast→next)
            slow = slow→next, fast = fast→next→next;
        prev = slow, slow = slow→next, prev→next = NULL;
        while (slow)
            temp = slow→next, slow→next = prev, prev =
slow, slow = temp;
        fast = head, slow = prev;
        while (slow)
            if (fast→val ≠ slow→val) return false;
            else fast = fast→next, slow = slow→next;
        return true;
    }
};
```

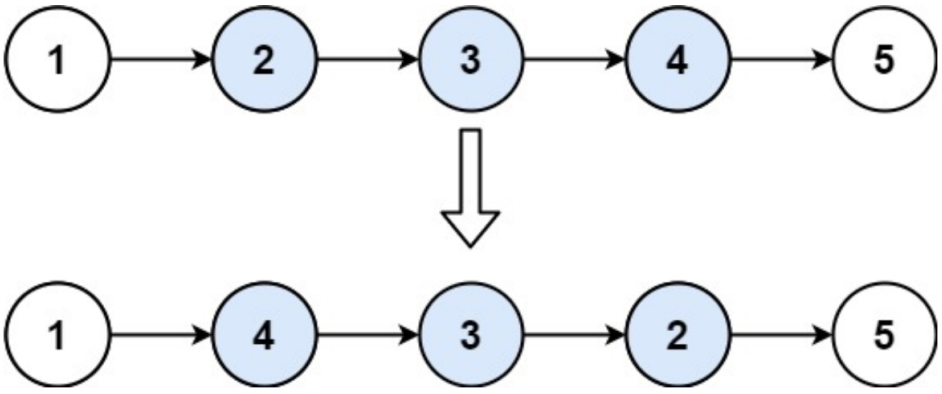
Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through the linked list twice—once to find the middle and reverse the second half, and once to compare the reversed second half with the first half.

Space complexity: O(1). The algorithm uses a constant amount of extra space for pointers and temporary variables, regardless of the size of the input linked list.

Reverse Linked List II (Leetcode-92)

Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return the reversed list.

Input: head = [1,2,3,4,5], left = 2, right = 4
Output: [1,4,3,2,5]



Code:

```
class Solution {
public:
   ListNode* reverseBetween(ListNode* head, int m, int n) {
       ListNode *dummy = new ListNode(0), *pre = dummy,
*cur;
       dummy -> next = head;
       for (int i = 0; i < m - 1; i++) {
           pre = pre -> next;
       }
       cur = pre -> next;
       for (int i = 0; i < n - m; i++) {
           ListNode* temp = pre -> next;
           pre -> next = cur -> next;
           cur -> next = cur -> next -> next;
           pre -> next -> next = temp;
       }
       return dummy -> next;
   }
};
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through the linked list once, reversing the specified portion of the list.

Space complexity: O(1). The algorithm uses a constant amount of extra space for pointers and temporary variables, regardless of the size of the input linked list.

Reorder List (Leetcode-143)

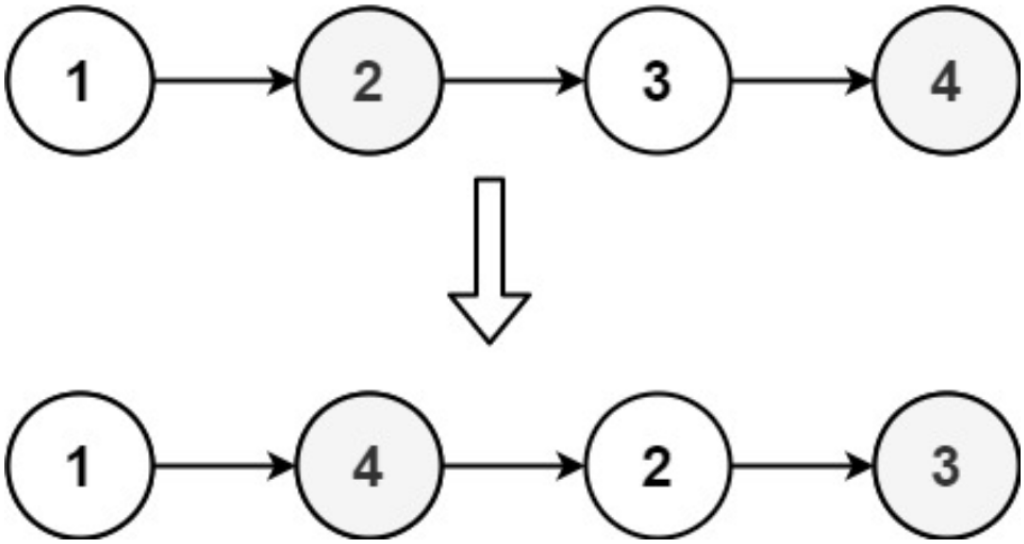
You are given the head of a singly linked-list. The list can be represented as:

L0 → L1 → … → Ln - 1 → Ln
Reorder the list to be on the following form:

L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …

You may not modify the values in the list's nodes. Only nodes themselves may be changed.

Input: head = [1,2,3,4]
Output: [1,4,2,3]



Code:

```
class Solution {
public:
    void reorderList(ListNode* head) {
        //base case i.e if the linked list has zero,one or
two elments just return it
        if(!head || !head->next || !head->next->next)
return;

        //Find the penultimate node i.e second last node of
the linkedlist
        ListNode* penultimate = head;
        while (penultimate->next->next) penultimate =
penultimate->next;

        // Link the penultimate with the second element
        penultimate->next->next = head->next;
        head->next = penultimate->next;

        //Again set the penultimate to the the last
        penultimate->next = NULL;

        // Do the above steps rcursive
        reorderList(head->next->next);
    }
};
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm has three main steps: cutting the list into two halves, reversing the second half, and merging the two halves. Each step involves iterating through a portion of the linked list, resulting in linear time complexity.

Space complexity: O(1). The algorithm uses a constant amount of extra space for pointers and temporary variables, regardless of the size of the input linked list.

Lesson Plan: Linked List-4

Doubly Linked List in C++ is very similar to a linked list, except that each node also contains a pointer to the node previous to the current one. This means that in a doubly linked list in C++ we can travel not only in the forward direction, but also in the backward direction, which was not possible with a palin linked list in C++.

Inserting a new node in a doubly linked list is very similar to inserting new node in linked list. There is a little extra work required to maintain the link of the previous node. A node can be inserted in a Doubly Linked List in four ways:

- At the front of the DLL.
- In between two nodes
  - After a given node.
  - Before a given node.
- At the end of the DLL.

Add a node at the front in a Doubly Linked List:

The new node is always added before the head of the given Linked List. The task can be performed by using the following 5 steps:

- Firstly, allocate a new node (say new_node).
- Now put the required data in the new node.
- Make the next of new_node point to the current head of the doubly linked list.
- Make the previous of the current head point to new_node.
- Lastly, point head to new_node.

```
void push(Node** head_ref, int new_data)
{
    // 1. allocate node
    Node* new_node = new Node();

    // 2. put in the data
    new_node->data = new_data;
```

```
    // 3. Make next of new node as head
    // and previous as NULL
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    // 4. change prev of head node to new node
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    // 5. move the head to point to the new node
    (*head_ref) = new_node;
}
```

Time Complexity: O(1)
Auxiliary Space: O(1)

Add a node in between two nodes:

It is further classified into the following two parts:

We are given a pointer to a node as prev_node, and the new node is inserted after the given node. This can be done using the following 6 steps:

- Firstly create a new node (say new_node).
- Now insert the data in the new node.
- Point the next of new_node to the next of prev_node.
- Point the next of prev_node to new_node.
- Point the previous of new_node to prev_node.
- Change the pointer of the new node's previous pointer to new_node.

```
void insertAfter(Node* prev_node, int new_data)
{
    // Check if the given prev_node is NULL
    if (prev_node == NULL) {
        cout << "the given previous node cannot be NULL";
        return;
    }

    // 1. allocate new node
    Node* new_node = new Node();

    // 2. put in the data
    new_node->data = new_data;
```

```
    // 3. Make next of new node as next of prev_node
    new_node->next = prev_node->next;

    // 4. Make the next of prev_node as new_node
    prev_node->next = new_node;

    // 5. Make prev_node as previous of new_node
    new_node->prev = prev_node;

    // 6. Change previous of new_node's next node
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
}
```

Time Complexity: O(1)
Auxiliary Space: O(1)

Add a node before a given node in a Doubly Linked List:

Let the pointer to this given node be next_node. This can be done using the following 6 steps.

- Allocate memory for the new node, let it be called new_node.
- Put the data in new_node.
- Set the previous pointer of this new_node as the previous node of the next_node.
- Set the previous pointer of the next_node as the new_node.
- Set the next pointer of this new_node as the next_node.
- Now set the previous pointer of new_node.
- If the previous node of the new_node is not NULL, then set the next pointer of this previous node as new_node.
- Else, if the prev of new_node is NULL, it will be the new head node.

```
void insertBefore(Node* next_node, int new_data)
{
    // Check if the given next_node is NULL
    if (next_node == NULL) {
        printf("the given next node cannot be NULL");
        return;
    }

    // 1. Allocate new node
    Node* new_node = new Node();

    // 2. Put in the data
    new_node->data = new_data;
```

```
    // 3. Make previous of new node as previous of next_node
    new_node->prev = next_node->prev;

    // 4. Make the previous of next_node as new_node
    next_node->prev = new_node;

    // 5. Make next_node as next of new_node
    new_node->next = next_node;

    // 6. Change next of new_node's previous node
    if (new_node->prev != NULL)
        new_node->prev->next = new_node;
    else
        head = new_node;
}
```

Time Complexity: O(1)
Auxiliary Space: O(1)

Add a node at the end in a Doubly Linked List:

The new node is always added after the last node of the given Linked List. This can be done using the following 7 steps:

- Create a new node (say new_node).
- Put the value in the new node.
- Make the next pointer of new_node as null.
- If the list is empty, make new_node as the head.
- Otherwise, travel to the end of the linked list.
- Now make the next pointer of last node point to new_node.
- Change the previous pointer of new_node to the last node of the list.

```
void append(Node** head_ref, int new_data)
{
    // 1. allocate node
    Node* new_node = new Node();

    Node* last = *head_ref; /* used in step 5*/

    // 2. put in the data
    new_node->data = new_data;

    // 3. This new node is going to be the last node, so
    // make next of it as NULL
    new_node->next = NULL;
```

```
    // 4. If the Linked List is empty, then make the new
    // node as head
    if (*head_ref == NULL) {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }

    // 5. Else traverse till the last node
    while (last->next != NULL)
        last = last->next;

    // 6. Change the next of last node
    last->next = new_node;

    // 7. Make last node as previous of new node
    new_node->prev = last;

    return;
}
```

Time Complexity: O(n)
Auxiliary Space: O(1)

The deletion of a node in a doubly-linked list can be divided into three main categories:

- After the deletion of the head node.
- After the deletion of the middle node.
- After the deletion of the last node.

All three mentioned cases can be handled in two steps if the pointer of the node to be deleted and the head pointer is known.

If the node to be deleted is the head node then make the next node as head.
If a node is deleted, connect the next and previous node of the deleted node.

Algorithm:

Let the node to be deleted be del.
If node to be deleted is head node, then change the head pointer to next current head.
if headnode == del then

headnode = del.nextNode

Set prev of next to del, if next to del exists.

if del.nextNode != none

del.nextNode.previousNode = del.previousNode

Set next of previous to del, if previous to del exists.

if del.previousNode != none

del.previousNode.nextNode = del.next

Below is the implementation of the above approach:

```
#include <bits/stdc++.h>
using namespace std;

/* a node of the doubly linked list */
class Node
{
    public:
    int data;
    Node* next;
    Node* prev;
};

/* Function to delete a node in a Doubly Linked List.
head_ref --> pointer to head node pointer.
del --> pointer to node to be deleted. */
void deleteNode(Node** head_ref, Node* del)
{
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head_ref == del)
        *head_ref = del->next;

    /* Change next only if node to be
    deleted is NOT the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;

    /* Change prev only if node to be
    deleted is NOT the first node */
    if (del->prev != NULL)
        del->prev->next = del->next;
```

```
    /* Finally, free the memory occupied by del*/
    free(del);
    return;
}

/* Function to insert a node at the
beginning of the Doubly Linked List */
void push(Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = new Node();

    /* put in the data */
    new_node->data = new_data;

    /* since we are adding at the beginning,
    prev is always NULL */
    new_node->prev = NULL;

    /* link the old list of the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given doubly linked list
This function is same as printList() of singly linked list
*/
void printList(Node* node)
{
    while (node != NULL)
    {
        cout << node->data << " ";
        node = node->next;
    }
}
int main()
{
    /* Start with the empty list */
    Node* head = NULL;
```

```
    /* Let us create the doubly linked list 10⟷8⟷4⟷2
*/
    push(&head, 2);
    push(&head, 4);
    push(&head, 8);
    push(&head, 10);

    cout << "Original Linked list ";
    printList(head);

    /* delete nodes from the doubly linked list */
    deleteNode(&head, head); /*delete first node*/
    deleteNode(&head, head→next); /*delete middle node*/
    deleteNode(&head, head→next); /*delete last node*/

    /* Modified linked list will be NULL←8→NULL */
    cout << "\nModified Linked list ";
    printList(head);

    return 0;
}
```

Output

Original Linked list 10 8 4 2
Modified Linked list 8

Complexity Analysis:

Time Complexity: O(1).
Since traversal of the linked list is not required, the time complexity is constant.
Auxiliary Space: O(1).
As no extra space is required, the space complexity is constant.

Q1. Split linked list in parts [Leetcode 725]

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next)
```

```
{}
 * };
 */
class Solution {
public:
    vector<ListNode*> splitListToParts(ListNode* head, int
k)
    {
        ListNode* curr=head;
        int n=0;
        while(curr){
            curr=curr->next;
            n++;
        }
        auto [q, r]=div(n, k);
        vector<int> iLen(k, q);
        for (int i=0; i<r; i++)
            iLen[i]++;
        vector<ListNode*> ans(k);
        curr=head;
        for(int i=0; i<k; i++){
            ans[i]=curr;
            int j=0;
            ListNode* prev=NULL;//extra pointer is needed
            while(j<iLen[i]){
                prev=curr;
                curr=curr->next;
                j++;
            }
            if (prev)
                prev->next=NULL;//Remove the link
        }
        return ans;
    }
};
```

Q2. Leetcode 2058

```
class Solution {
public:
    vector<int> nodesBetweenCriticalPoints(ListNode* head) {
```

```
        vector<int> ans={-1,-1};
        ListNode* prev=head;
        if(!prev) return ans;
        ListNode* curr=prev->next;
        if(!curr) return ans;
        ListNode* next=curr->next;
        if(!next) return ans;

        int first=-1;
        int last=-1;
        int mind=INT_MAX;
        int i=1;
        while(next){
            bool iscp=((curr->val>prev->val && curr
->val>next->val)||(curr->val<prev->val&&curr->val<next
->val))?true:false;
            if(iscp&&first==-1){
                first=i;
                last=i;
            }
            else if(iscp){
                mind=min(mind,i-last);
                last=i;
            }
            i++;
            prev=curr;
            curr=next;
            next=next->next;
        }
        if(first==last)
         return ans;
         else
         {
             ans[0]=mind;
             ans[1]=last-first;
         }
         return ans;
    }
 };
```

Q3. Leetcode 2074

```
class Solution {
public:
    // Reverse function
    ListNode* reverse(ListNode* head) {
        if(head == NULL)
            return head;

        ListNode* prev = NULL, *forward = NULL;
        while(head != NULL) {
            forward = head->next;
            head->next = prev;
            prev = head;
            head = forward;
        }
        return prev;
    }

    ListNode* reverseEvenLengthGroups(ListNode* head) {
        ListNode* dummy = new ListNode(), *prev = dummy;
        dummy->next = head;

        // Step 1 - determine the length of groups
        for(int len = 1; len < 1e5 && head; len++) {
            ListNode* tail = head, *nextHead;

            // Determining the length of the current group
            // Its maximum length can be equal to len
            int j = 1;
            while(j < len && tail != NULL && tail->next !=
NULL) {
                tail = tail->next;
                j++;
            }

            // Head of the next group
            nextHead = tail->next;
            if((j % 2) == 0) {
                // If group size is even then reverse the
group and set prev and head
                tail->next = NULL;
                prev->next = reverse(head);
                prev = head;
                head->next = nextHead;
                head = nextHead;
```

```
            }
            else {     // If group is odd sized then simply
go to the next group
                prev = tail;
                head = nextHead;
            }
        }

        return dummy->next;
    }
};
```

Q4. Leetcode 138

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = NULL;
        random = NULL;
    }
};
*/

class Solution {
public:
    Node* copyRandomList(Node* head) {

        if (head == nullptr)
        return nullptr;

        unordered_map<Node*, Node*> nodeMap;

        // Create copies of nodes without random pointers
        Node* current = head;
        while (current != nullptr) {
            nodeMap[current] = new Node(current->val);
            current = current->next;
        }
```

```
        // Set the next and random pointers of the copied
nodes
        current = head;
        while (current != nullptr) {
            nodeMap[current]->next = nodeMap[current->next];
            nodeMap[current]->random = nodeMap[current
->random];
            current = current->next;
        }

        return nodeMap[head];
    }
};
```

Q5. Leetcode 430

```
Node* flatten(Node* head) {
 for (Node* h = head; h; h = h->next)
 {
  if (h->child)
  {
   Node* next = h->next;
   h->next = h->child;
   h->next->prev = h;
   h->child = NULL;
   Node* p = h->next;
   while (p->next) p = p->next;
   p->next = next;
   if (next) next->prev = p;
  }
 }
 return head;
}
```