PW SKILLS

Lesson Plan

Linked List – 1

PW SKILLS

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
8. Delete Node in a Linked List (Leetcode-237)
9. Middle of Linked List (Leetcode-876)
10. Remove Nth Node from End of List (Leetcode-19)
11. Intersection of two Linked Lists (Leetcode-160)
12. Linked List Cycle (Leetcode-141)
13. Linked List Cycle-II (Leetcode-142)

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.

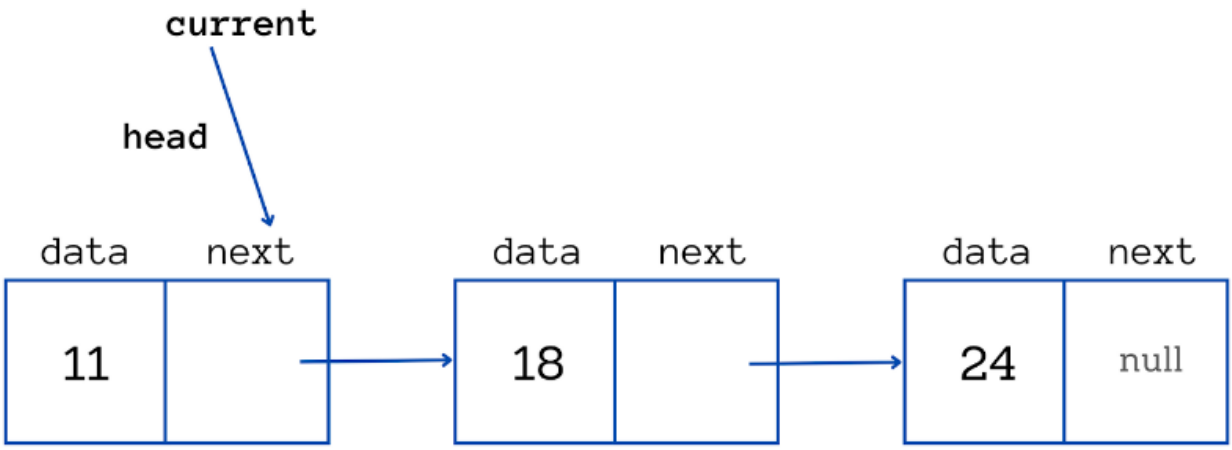
4. **Wastage of Memory:** If you allocate more space than needed for an array, you may end up wasting memory. This is particularly problematic when the array size is predetermined to accommodate the worst-case scenario.

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;
}
```

```
Length (Iterative): 4
Length (Recursive): 4


...Program finished with exit code 0
Press ENTER to exit console.
```

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)

Delete Node in a Linked List (Leetcode-237)

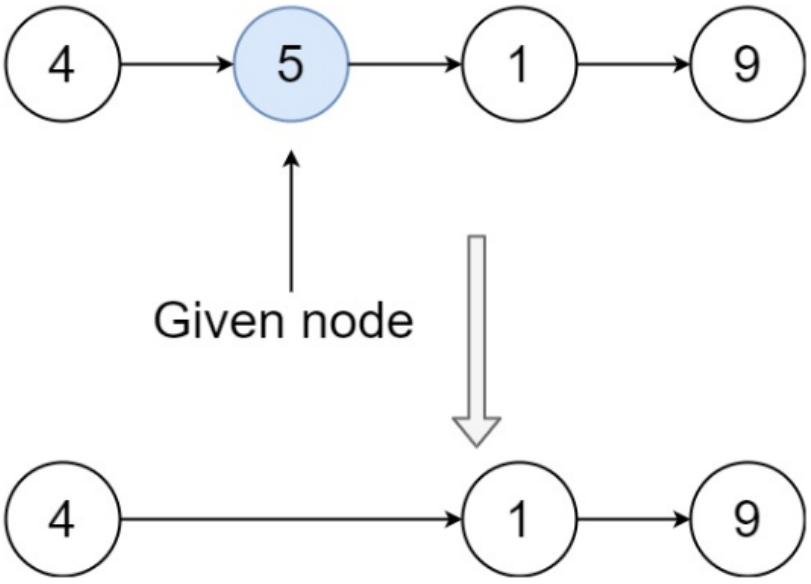
There is a singly-linked list head and we want to delete a node node in it.
You are given the node to be deleted node. You will not be given access to the first node of head.
All the values of the linked list are unique, and it is guaranteed that the given node node is not the last node in the linked 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:
    ListNode* middleNode(ListNode* head) {
        ListNode *slow = head, *fast = head;
        while (fast && fast→next) {
            slow = slow→next;
            fast = fast→next→next;
        }
        return slow;
    }
};
```

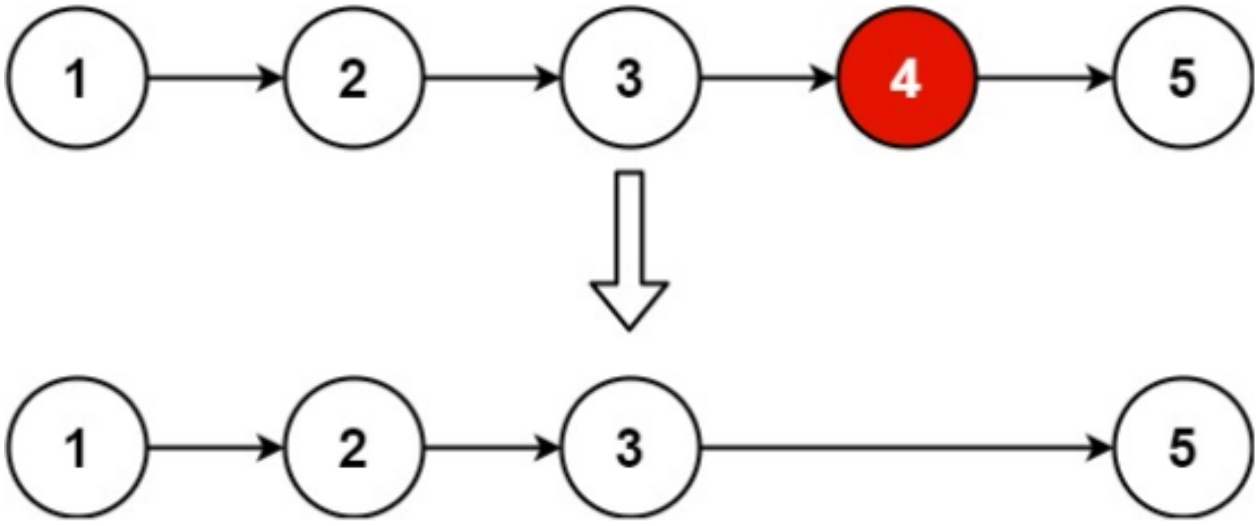
Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through the linked list with two pointers (slow and fast), and in each iteration, the fast pointer moves two steps while the slow pointer moves one step.

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;
    }
};
```

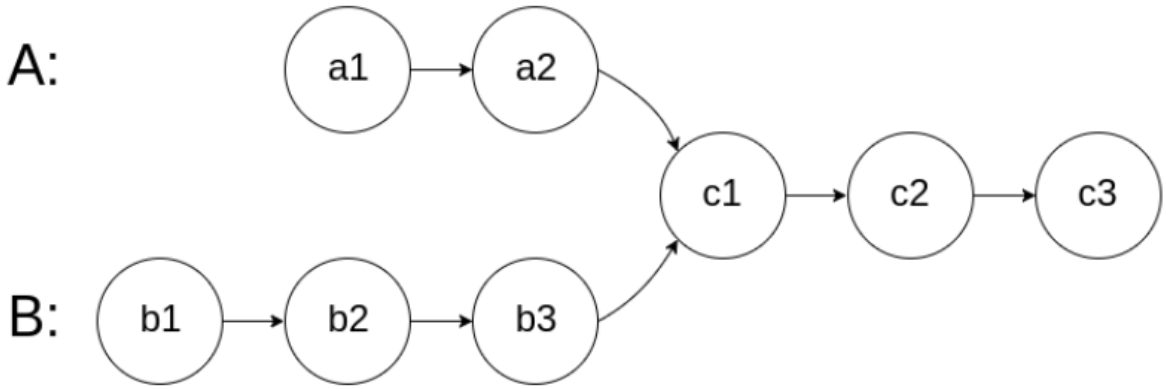
Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm uses two pointers to traverse the linked list. The first loop advances the fast pointer by n nodes, and then the second loop advances both pointers until the fast pointer reaches the end.

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)

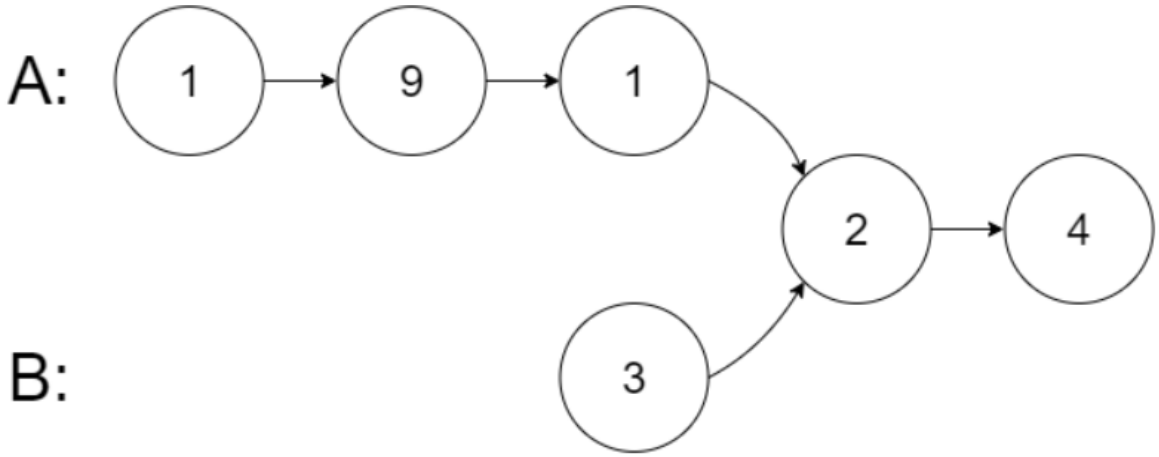
Given the heads of two singly linked-lists headA and headB, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return null.

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.



Code:

```
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)

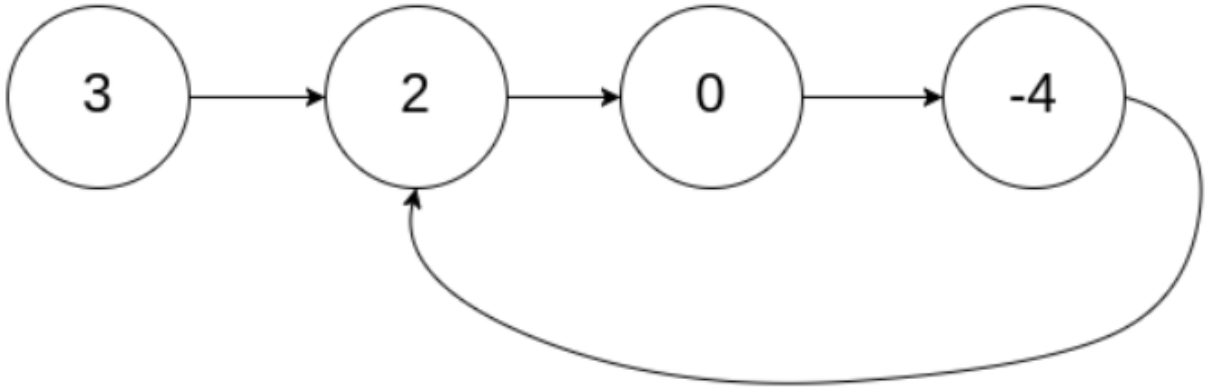
Given head, the head of a linked list, determine if the linked list has a cycle in it.

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)

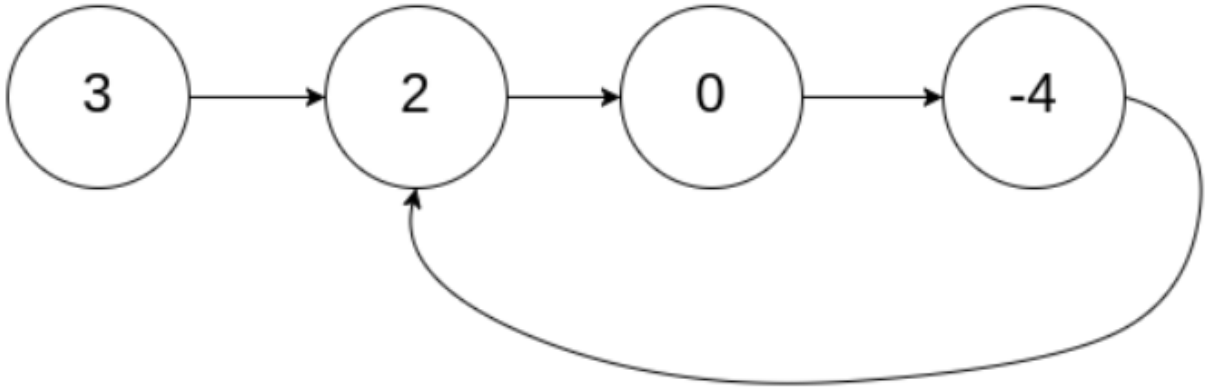
Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return null.

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.

PW SKILLS

THANK YOU !

PW SKILLS