Loop Detection in Linked List

The goal of loop detection in a linked list is to determine whether the linked list contains a cycle, i.e., if any node in the list points back to a previous node, causing an infinite loop. This problem ensures that operations on a linked list don't run indefinitely.

The intuition behind detecting a loop is based on the observation that in a looped linked list, the fast-moving pointer will eventually meet the slow-moving pointer, as they will "lap" each other due to the loop's cyclic nature.

- Time Complexity: **O(n)** because each pointer only traverses the list once.
- Space Complexity: **O(1)** as only two pointers are used, regardless of the list size.

Applications

- **Memory Management**: Detect and prevent infinite loops caused by circular references in data structures.
- **Network and Graph Theory**: Identify cycles in network paths or dependency graphs.
- **Compiler Design**: Check for circular dependencies in variable declarations.
- **Games and Simulation**: Detect infinite loops in game states or simulations.
- **Linked List Problems**: Prevent infinite traversals in data handling algorithms.

  Input : 1->2->3->4->2
  Output : Yes CYcle Exists

Segregate Even & Nodes in a Linked List

The task is to rearrange the nodes of a linked list so that all nodes with even values come before the nodes with odd values while maintaining the relative order of nodes within the even and odd groups. The intuition behind the solution is to treat the linked list as two separate lists—one for even nodes and one for odd nodes—and then combine them. By traversing the original list once, we can partition the nodes into two sub-lists (even and odd) and finally merge them to achieve the desired result.

- **Even List:** Stores all nodes with even values.
- **Odd List:** Stores all nodes with odd values.
- Time Complexity: **O(n)** because the list is traversed only once.
- Space Complexity: **O(1)** because the reordering is done in place without using extra space for new nodes.

Applications:

- **Preprocessing for Algorithms:** Simplify operations or analyses that treat even and odd values differently.
- **Load Balancing:** In distributed systems, segregating tasks based on even or odd IDs.
- **Partitioning for Simplicity:** Useful for problems where even and odd nodes need to be processed separately.
- **Sorting or Filtering Data Streams:** Quick filtering of data based on parity while maintaining order.
- **Linked List Problems:** This technique serves as a foundational step for more complex linked list transformations.

Input : 1 6 3 2 0 4 5

Output : 6 2 0 4 1 3 5

Sort Bitonic in DLL

A bitonic sequence is a sequence that is first monotonically increasing and then monotonically decreasing. The idea for sorting such a sequence efficiently is to split the sequence into two parts. The increasing part (already sorted in ascending order).The decreasing part (sorted in descending order). Once split, the problem reduces to merging two sorted sequences: The increasing part stays as it is. The decreasing part is reversed to convert it into ascending order. After reversing, the two sorted parts can be merged efficiently using a two-pointer technique (similar to merging two sorted lists).

- A bitonic sequence has two parts: an increasing part followed by a decreasing part.
- Identifying these parts helps split the problem into manageable sub-problems.
- **Split the List - > Reverse the Decreasing Part - > Merge the Two Sorted Lists**
- **Time Complexity: O(nlogn)**, where n is the number of nodes in the list. Each operation (splitting, reversing, merging) takes linear time.
- **Space Complexity: O(1)**, as operations are done in place without requiring additional memory.

Applications:

- **Sorting Specialized Data:** Bitonic sequences often arise in real-world problems like signal processing and time-series data.
- **Parallel Computing:** Bitonic sorting is widely used in parallel computing because it can be efficiently implemented using divide-and-conquer techniques.
- **Data Analysis and Compression:** Bitonic sequences can appear in datasets where values first increase and then decrease (e.g., sales trends, stock market data).
- **Networking (Bitonic Sort in Sorting Networks):** Bitonic sort is an essential component of sorting networks, which are used in hardware-based sorting algorithms.
- **Optimization Problems:** Useful in problems where splitting and merging data provides a more optimal solution compared to general-purpose sorting.

Input : 1 2 3 4 5 9 8 7 6

Output 1 2 3 4 5 6 7 8 9

Merge Sort in DLL

The Merge Sort algorithm divides the linked list into two halves, recursively sorts each half, and then merges the two sorted halves to produce a fully sorted list. For a doubly linked list (DLL), this approach leverages the bidirectional nature of the list to efficiently split and merge nodes without requiring extra memory for arrays or additional data structures

- **Time Complexity: O(n log n)** for the entire algorithm.
- **Space Complexity: O(log n)** for the recursion stack.

Applications:

- **Sorting Large Linked Data Structures:** Efficiently sort doubly linked lists in scenarios where in-place sorting is required.
- **Database Management Systems:** Used for indexing and sorting rows in doubly linked structures.
- **Optimal Sorting for Sparse Memory:** No additional memory allocation makes it suitable for low memory environments.
- **Backend Operations:** Efficiently sort logs, tasks, or records stored in doubly linked lists.
- **Foundational Algorithm:** Merge Sort for DLLs is a foundational step for understanding advanced linked list manipulations and sorting algorithms.

Input - > Some unsorted data
Output -> Sorted in ascending / descending order

Minimum Stack

The goal of a Minimum Stack (Min Stack) is to design a stack that, in addition to the standard stack operations (push, pop, top), can retrieve the minimum element in constant time O(1). The intuition behind the efficient implementation is to use an auxiliary stack to track the minimum values. This auxiliary stack stores the minimum value observed so far whenever a new value is pushed onto the main stack. Thus, at any point, the top of the auxiliary stack represents the current minimum value.

- **Time Complexity:** O(1)
- **Space Complexity:** O(n), as the auxiliary stack stores an extra value for each element in the main stack.

Applications:

- **Efficient Data Retrieval:** Retrieve minimum values quickly in stack-based algorithms.
- **Histogram Problems:** Useful for solving problems involving rectangles in histograms or areas bounded by stacks.
- **Dynamic Minimum Tracking:** Real-time systems that require constant-time retrieval of minimum values (e.g., price monitoring).
- **Algorithm Optimization:** Used in dynamic programming or sliding window problems to optimize minimum value calculations.
- **Game Development:** Track minimum values during gameplay logic involving stack-like undo/redo features.

Input : Push(10)
Push(15)
Push(12)

…….. 12 -> 15 -> 10
getMin() ->10
getMin() -> 12
getMin() -> 15

Celebrity Problem

The core idea is to use a stack to eliminate non-celebrities efficiently. By leveraging the properties of a celebrity:

- A celebrity does not know anyone.
- Everyone knows the celebrity.

We can systematically eliminate potential candidates by comparing pairs of people and only keeping those who may satisfy the celebrity criteria.

- **Time Complexity: O(n)**.
- **Space Complexity: O(n)** for the stack.

Applications:

- **Social Network Analysis:** Identify influencers or key nodes in a network.
- **Game Theory:** Solve problems where hierarchy or isolation criteria are involved.
- **Database Systems:** Resolve relationships in datasets with directed graphs.
- **Event Planning:** Identify VIPs at an event based on who is recognized by others.
- **Graph Theory Problems:** Applicable in problems involving dominance relationships in directed graphs

Input -> mat[][]={

0 1 1 0 ->0
0 0 0 0 -> 1
0 1 0 0 -> 2
1 1 0 0 -> 3

} 4x4 {n=4}

Celebrity -> 1 or 2 or 3 or 4
> celebrity knows no one and everyone knows him
> mat[i][j] = 1 => i knows j => j is celebrity maybe
> mat[j][i] = 1 => j knows i => i ‘’

Output : Celebrity is 2

> min -> 0
> max -> 1

Tower of Hanoi

The Tower of Hanoi is a classic problem where the goal is to move n disks from a source rod to a target rod, following these rules:

- Only one disk can be moved at a time.
- A larger disk cannot be placed on top of a smaller disk.
- There is a third rod (auxiliary) that can be used as a temporary holding area.

The key intuition is to divide the problem into smaller subproblems:

- To move n disks, first move the top n-1 disks to the auxiliary rod.
- Move the largest disk (nth disk) directly to the target rod.
- Finally, move the n-1 disks from the auxiliary rod to the target rod.

This recursive strategy systematically reduces the problem size until it reaches the base case of moving a single disk.

- **Base Case:** When n = 1, simply move the disk from the source rod to the target rod.
- **Time Complexity:** O(2^n).
- **Space Complexity:** O(n) due to the recursive function call stack space.
- n=2 -> No of movements => 2^n -1 =>4-1=3
  n=3 => 8-1=7

Applications:

- **Algorithmic Thinking:** Teaches recursive problem-solving and divide-and-conquer techniques.
- **Data Structure Manipulation:** Useful in scenarios requiring stepwise movement or transfer of data while maintaining order.
- **Real-Life Scheduling:** Models tasks that require intermediate steps, such as assembly lines or multitiered workflows.
- **Disk Balancing in Distributed Systems:** Simulates transferring data between servers while ensuring order is maintained.
- **Educational Tools:** Demonstrates recursion, exponential growth, and problem decomposition in computer science courses.

Stock Span

The Stock Span Problem involves finding the span of stock prices for each day. The span for a given day is defined as the maximum number of consecutive days (including the current day) the stock price has been less than or equal to the stock price of the current day.
The stack-based solution relies on maintaining a monotonic decreasing stack of indices. For each day's stock price, we check previous days' prices efficiently by "jumping" over days that have lower prices.

- **Time Complexity: O(n)**.
- **Space Complexity: O(n)** for the stack.

Applications:

- **Stock Market Analysis:** Calculate stock spans to understand trends in stock prices.
- **Financial Data Analysis:** Useful in scenarios where consecutive patterns of higher prices need to be identified.
- **Histogram Problems:** Variations of this technique are applied in solving histogram-based problems like finding the largest rectangle.
- **Weather Trends:** Analyze temperature or weather data for consecutive days with similar or lower temperatures.
- **Competitive Programming:** Frequently used in coding competitions to demonstrate efficient stack-based solutions.

Input : 7 2 1 3 3 1 8
Output : 1 1 1 3 4 1 7 -> span[]

Ngl |
Ngr |
Nsl |
Nsr ……

Priority Queue Using DLL

A priority queue is a data structure that processes elements based on their priority rather than their insertion order. The goal of implementing a priority queue using a doubly linked list (DLL) is to leverage the DLL's bidirectional traversal to efficiently insert and retrieve elements while maintaining their priority order.
The main idea is to:

1. Maintain a sorted doubly linked list where the elements are arranged in descending or ascending order of priority.
2. Insertion is performed at the correct position based on the priority, ensuring the list remains sorted.
3. Removal (either the highest or lowest priority element) is performed in constant time O(1), as the list's head or tail can be accessed directly.

- Insert: **O(n)** (due to traversal to find the correct position).
- Delete: **O(1)**.
- Peek: **O(1)**.

Applications:

- **Scheduling Tasks:** Use in operating systems to schedule tasks based on priority.
- **Dijkstra's Algorithm:** Manage the processing of nodes in shortest path algorithms.
- **Data Stream Processing:** Handle dynamic elements with priorities (e.g., streaming logs, alerts).
- **Event Handling:** Simulate or manage real-time events based on priority.
- **Job Scheduling:** Assign tasks to resources in priority order.
- **Real-Time Systems:** Ensure high-priority tasks are processed before lower-priority tasks.

> PQ by default JCF -> ascending p
> PriorityQueue<Integer> pq=new PriorityQueue<>();
> Queue<Integer> q=new LinkedList<>();

Sort Without Extra Space

The problem involves sorting elements while using a queue (or queues) as the primary data structure and avoiding extra space for storage or auxiliary arrays. Queues inherently operate in a FIFO (First-In-First-Out) manner, so sorting requires careful manipulation of their structure. The idea is to repeatedly extract the smallest (or largest) element, reposition it at the end, and ensure that all elements are sorted within the queue itself.

- **Time Complexity:**
    - For n elements, this results in a total complexity of **$O(n^2)$**.
- **Space Complexity:**
    - No extra space beyond the queue itself, so **O(1)** auxiliary space.

Applications:

- **Memory-Constrained Systems:** Sort data in environments where additional memory allocation is restricted.
- **Queue-Based Data Processing:** Situations where input and output must remain in queue form, such as in network or task scheduling systems.
- **Streaming Data:** Process and sort elements in queues for streaming applications.
- **Embedded Systems:** Use in low-memory devices where efficient in-place sorting is necessary.
- **Event Management:** Prioritize events in real-time systems without using additional memory.

Stack Permutations

The Stack Permutation Problem asks if a given sequence can be the result of a stack operation starting from an increasing sequence of numbers (typically from 1 to n). The goal is to determine if it is possible to push elements onto a stack and pop them to obtain a particular permutation.

To solve this:

• We simulate the process of pushing and popping elements into/from the stack.

• We start by pushing numbers from 1 to n onto the stack.

• For each number in the desired permutation, check if it is at the top of the stack. If it is, pop it. Otherwise, continue pushing numbers onto the stack until the top of the stack matches the current element in the desired permutation.

The intuition is that a stack follows a Last-In-First-Out (LIFO) order, so elements can only be removed in the reverse order in which they were added.

◦ **Time Complexity:** The process involves pushing and popping each element at most once, so the time complexity is O(n), where n is the number of elements.

◦ **Space Complexity:** The space complexity is O(n) due to the stack.

Applications:

- **Expression Evaluation:** In evaluating expressions (e.g., infix to postfix conversion), understanding stack permutations can help validate if a given sequence of operations is achievable.
- **Compiler Design:** Used in scenarios where operations involve processing sequences based on last-in-first-out constraints.
- **Real-Time Scheduling:** Stack-based operations can be applied in scheduling tasks where the order of execution follows strict last-in-first-out rules.
- **Reversible Operations:** Stack permutations can be used in problems that involve reversible operations or backtracking.
- **Data Structure Simulation:** Helps in simulating how certain data structures (like stacks) behave under different operational constraints.

Input : ip[]={1 2 3}
op[]={ 2 1 3}

Stack ->
If empty then yes we can do it
Output : yes / no / 1 / 0 / true / false
Yes / 1 / true :)

Time and Space Complexities

Problem	Time Complexity	Space Complexity
The Celebrity Problem	**n**	**n**
Stock Span Problem	**n**	**n**
Loop Detection	**n**	**1**
Segregate even & odd nodes in LL	**n**	**1**
Sort the Bitonic DLL	**N log n**	**1**
Merge Sort for DLL	**N log n**	**Log n**
Sort without extra Space	**n^2**	**1**
Minimum Stack	**1**	**n**
Tower of Hanoi	**2^n**	**n**
Stack Permutations	**n**	**n**
Priority Queue using DLL	**Insert -> n** **Delete -> 1** **Peek -> 1**	**n**

LL DLL STACK QUEUE PQ DQ
addFirst for LL DLL -> Node : next data/value
Head / tail
LL & DLL -> LL next | DLL next prev pointers
Stack -> push pop peek (Reversing a string / some data)
LIFO
Queue -> push pop peek
Front and rear -> arrays
Head and tail -> SLL
FIFO
PQ -> add poll peek | ascending priorities
DQ -> front remove and add | rear add and remove

Articles LL STACK QUEUES

Practice

Loop Detection

> Relation between head(start of the LL) and cycleStart(starting point of the cycle)
No relationship exists
>Slow pointer -> moves 1 step | Fast pointer -> moves 2 steps approach
>Floyd's Algo
> Tc Sc

Sort the bitonic DLL

> Most efficient sorting algo -> Merge sort
> Monotonically inc then monotonically dec ->
First we find dec half 1st element -> send dec half to rev fn
-> the dec half becomes into ascending half
-> next we compare one by one and merge
> TC-> for bitonic dll using merge sort -> n
> TC for merge sort -> nlogn

Segregate even & odd nodes in a LL

> Input ? Output

Merge sort for DLL

> TC SC
> getMid() -> slow fast approach to find the middle element / node
> merge fn
> mergesort -> recursion each time we find new mid and call this fn for left half and right half
Sending the newLeftHead & newRightHead
> mege sort and quick sort diff
Quick sort takes sc -> O(1)

Minimum Stack

> pop push peek getmin()
> auxiliary stack or another stack
> to get min element in O(1)

The Celebrity problem

> Input and ask u output
> Who is a celebrity -> Who knows no one and everyone knows him
> Probable output -> maybe a 1 celebrity found or no celebrity exists
> min =0 max=1
> stack ds

Iterative Tower of Hanoi

> TC -> 2^N
> No of movements -> 2^N - 1
N=2
4-1=3

Stock Span problem

> DS stack
> input and ask u the output
> ngl | > ngr nsl nsr

Priority Queue using DLL

> Insert an element ?TC -> O(n)
Peek pop ?
> Sample inputs and some count say 3
U ll have to tick first 3 min elements

Sort without extra Space

>Arrays.sort(arr)
> n^2 tc because we do inplace sorting and we use something selection sort approach

>sc -> 1
> ds used queue or deque

Stack permutations

> When no element is missed or all element is covered
> ip[]=1 2 3
Op[] = 3 2 1
Op[]=2 1 3
Op[]= 3 1 2 -> Invalid case
> DS stack
> Permutation -> push pop ()