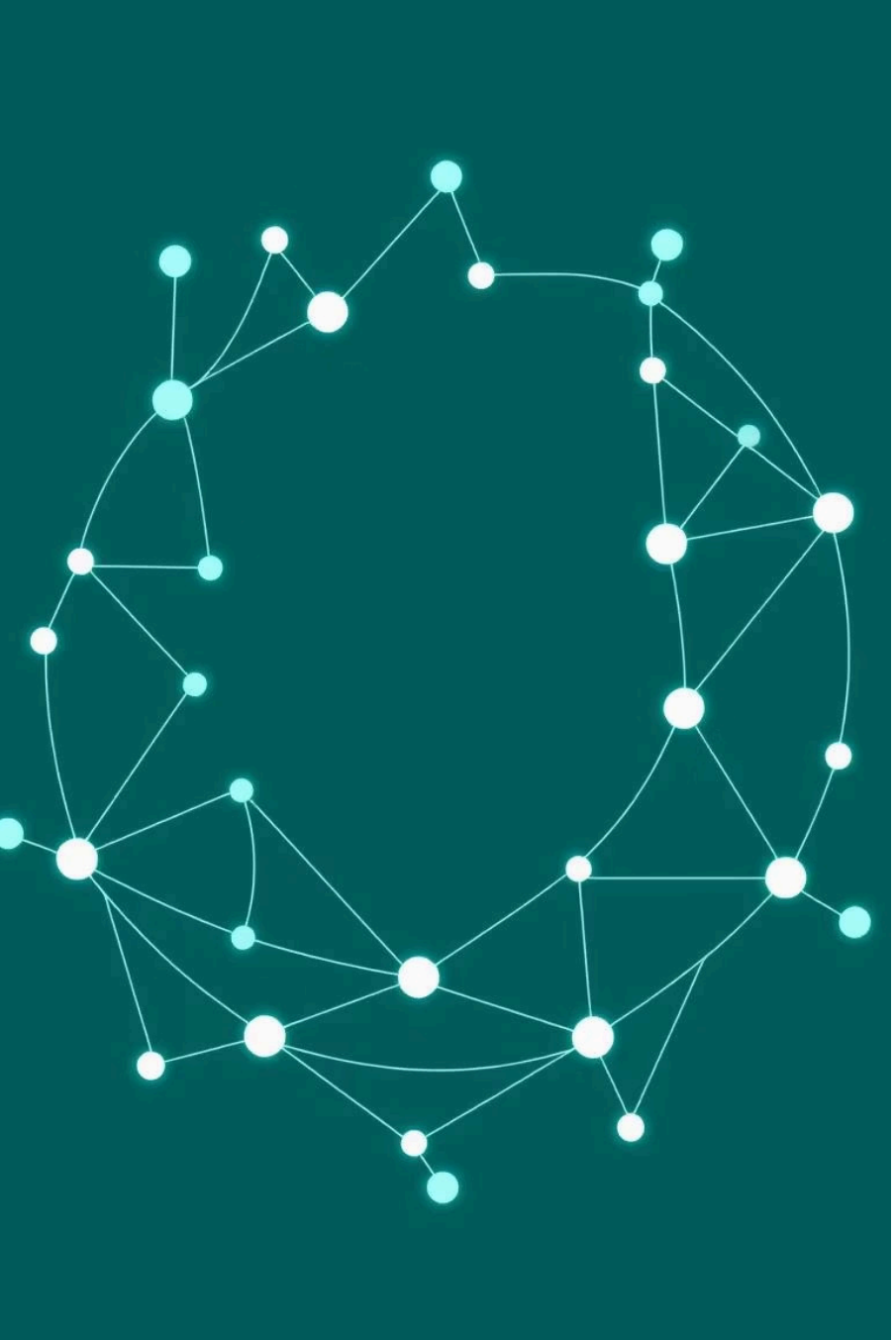


DSA Session 3: Circular & Doubly Linked Lists

Exploring advanced linked list structures and their applications.

DSA Session 3

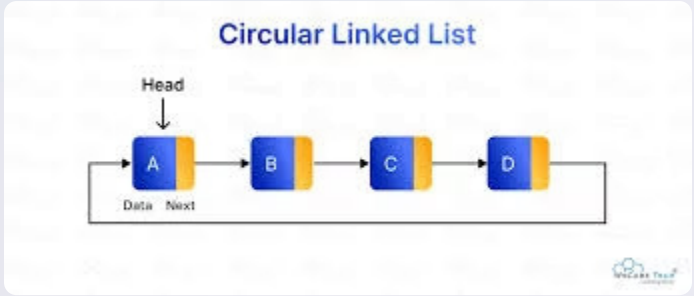
Prerequisites & Objectives

Prerequisites

- Singly linked lists
- Pointers & node structures
- Traversal, insertion/deletion logic

Learning Objectives

- Understand Circular & Doubly Linked Lists
- Implement basic operations
- Analyze time/space complexities
- Know when to prefer these structures



What is a Circular Linked List (CLL)?

A Circular Linked List (CLL) is a variation of the singly linked list where the last node points back to the head, forming a circle.

```
struct Node{
  int data;
  Node* next;
};
```

CLL vs. Singly Linked List: Key Functional Changes

Traversal

CLL stops when reaching the head again (do-while loop), unlike SLL's `nullptr` check.

Insertion at End

In CLL, new node's next points to head; last node's next points to new node.

Deletion

Deleting head in CLL requires updating the last node's pointer. Special attention for circular structure to avoid infinite loops.

Edge Cases

Single-node circular lists (head->next == head) need special handling.

CLL Operations: Traversal & Insertion

1. Traversal

```
void printCLL(Node* head) {
  if (!head) return;
  Node* temp = head;
  do {
    cout << temp->data << " -> ";
    temp = temp->next;
  } while (temp != head);
  cout << "(back to head)\n";
}
```

2. Insertion at Beginning

```
void insertAtHeadCLL(Node*& head, int val) {
  Node* newNode = new Node{val, nullptr};
  if (!head){
     head = newNode;
     newNode->next = newNode;
     return;
   }
  Node* temp = head;
  while(temp->next != head){
     temp = temp->next;
  }
  newNode->next = head; temp->next = newNode;
  head = newNode;
}
```

3. Insertion at End

```
void insertAtEndCLL(Node*& head, int val) {
 Node* newNode = new Node{val, nullptr};
 if(!head){
    head = newNode; newNode->next = head;
    return;
 }
 Node* temp = head;
 while(temp->next != head){
    temp = temp->next;
 }
 temp->next = newNode;
 newNode->next = head;
}
```

Applications of Circular Linked Lists

CLLs are useful when data needs to be processed in a looping or circular fashion.

Games

Multiplayer turn-based games for rotating player turns.

Playlists

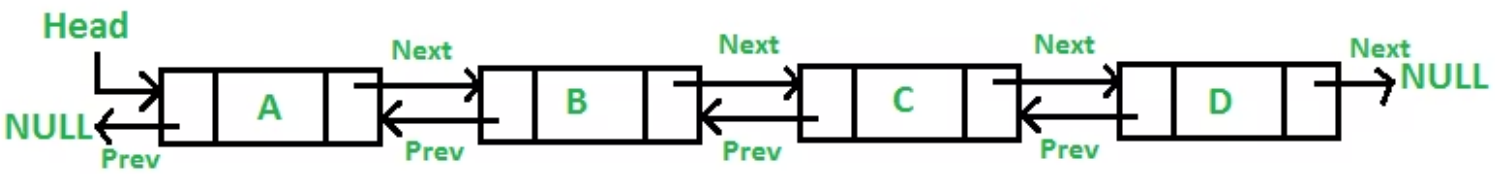
Music/video playlists that loop from the start.

Token Passing

Network ring topologies where a token circulates.

Undo/Redo

Advanced undo/redo functionality with bounded history.



What is a Doubly Linked List (DLL)?

A Doubly Linked List (DLL) is a linear data structure where each node contains three components: data, a pointer to the previous node, and a pointer to the next node.

```
struct Node {
  int data;
  Node* prev; // Pointer to the previous node
  Node* next; // Pointer to the next node
};
```

DLLs allow both forward and backward traversal, unlike singly linked lists.

DLL Operations: Traversal & Insertion

1. Traversal (Backward)

One of the key advantages of a DLL is the ability to traverse backward from any point, including the tail. Here's how to print the list in reverse:

```
void printBackward(Node* tail) {
  Node* temp = tail;
  while (temp) {
    cout << temp->data << " <-> ";
    temp = temp->prev;
  }
  cout << "NULL\n";
}
```

This function starts from the `tail` and follows the `prev` pointers until it reaches the beginning (`nullptr`).

2. Insertion at Head

Inserting a new node at the head of a DLL requires updating pointers for both the new node and the existing head:

```
void insertAtHead(Node*& head, int val) {
  Node* newNode = new Node{val, nullptr, head};
  if (head)
    head->prev = newNode;
  head = newNode;
}
```

The `newNode`'s `next` pointer is set to the current `head`. If the list is not empty, the current `head`'s `prev` pointer is updated to point to the `newNode`, and finally, `head` is updated to be the `newNode`.

DLL Operations: Insertion & Deletion

3. Insertion at End

Adding a new node to the end of a DLL involves finding the current last node and linking it to the new node. This means updating the `next` pointer of the original last node and the `prev` pointer of the new node.

```
void insertAtTail(Node*& head, int val) {
  Node* newNode = new Node{val, nullptr, nullptr};
  if (!head) {
    head = newNode;
    return;
  }
  Node* temp = head;
  while (temp->next)
    temp = temp->next;
  temp->next = newNode;
  newNode->prev = temp;
}
```

Special handling is required if the list is initially empty.

4. Deletion by Value

To delete a node by its value, we first traverse the list to find the target node. Once found, we need to update the `next` pointer of its previous node and the `prev` pointer of its next node to bypass the deleted node. Edge cases include deleting the head or a value not found.

```
void deleteByValue(Node*& head, int val) {
  Node* temp = head;
  while (temp && temp->data != val)
    temp = temp->next;
  if (!temp) return; // value not found

  if (temp->prev)
    temp->prev->next = temp->next;
  else
    head = temp->next; // deleting the head

  if (temp->next)
    temp->next->prev = temp->prev;
  delete temp;
}
```

Key Differences: SLL vs. DLL

Navigation	One-way (only next)	Two-way (prev and next)
Memory per node	Less (one pointer)	More (two pointers)
Reverse traversal	Not possible	Easy and direct
Deletion efficiency	Less efficient (need prev)	More efficient (direct access)

Applications of Doubly Linked Lists

Browser History

Forward & backward navigation.

Undo/Redo

Functionality in editors.

Music Playlists

Move to next/previous songs.

LRU Cache

Efficient eviction operations.

Further Learning & Practice

Resources

- GeeksforGeeks – Circular Linked List
- GeeksforGeeks – Doubly Linked List
- VisualAlgo – Linked List Explorer

Recommended Problems

- 141. Linked List Cycle
- 142. Linked List Cycle II
- 61. Rotate List
- 1472. Design Browser History

Introduction to C++ Sets (STL)

A **set** is an STL (Standard Template Library) container that:

- Stores **unique elements** (no duplicates).
- Stores them in **sorted order** (by default ascending).

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

int main() {

    set<int> s;

    s.insert(5);
    s.insert(1);
    s.insert(3);
    s.insert(3); // Duplicate element, ignored

    cout << "Elements in set";
    for(int x : s) cout << x << " "; // Output: 1 3 5


    return 0;
}
```

Problems:

Anton and Letters (CF 443A)

Boy or Girl

`std::unordered_set`

Similar to `std::set`, an `std::unordered_set` also stores unique elements, but elements are stored in **no particular order**.

Introduction to C++ Maps (STL)

A **map** is an STL container that:

- Stores elements as **key-value pairs**.
- Ensures each **key is unique**.
- Keeps elements **sorted by key** in ascending order.

```
#include <iostream>
#include <map>
#include <string> // Required for std::string
using namespace std;

int main() {
    map<int, string> m;

    m[1] = "One"; // Insert using array-like syntax
    m[2] = "Two";
    m[5] = "Five";

    // Insert using the insert method
    m.insert({3, "Three"});
    m.insert({1, "Uno"}); // Key 1 already exists, this will be ignored

    cout << "Elements in map:" << endl;
    for(auto const& [key, val] : m) {
        cout << key << ": " << val << endl;
    }
    // Expected output (sorted by key):
    // 1: One
    // 2: Two
    // 3: Three
    // 5: Five

    return 0;
}
```

Maps provide efficient lookup, insertion, and deletion of elements based on their keys, making them ideal for associative arrays.

Problems:

[Registration system]

[Petya and Strings]

Unordered Map:

Similar to `map`, but stores **key → value pairs** in **no order**.