Linked Lists Cheatsheet

Linked lists are linear data structures where elements (nodes) are not stored in contiguous memory locations. Instead, each node contains data and a pointer (or reference) to the next node in the sequence. This non-contiguous storage allows for efficient insertions and deletions compared to arrays.

Singly Linked List

A singly linked list is a linear data structure where each element (node) points to the next element in the sequence. Each node typically has two components: a `data` part to store the value and a `next` pointer part to store the address of the next node.

Node Structure for Singly Linked List

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

    // Default constructor
    Node() {
        data = 0;
        next = NULL;
    }

    // Parameterized Constructor
    Node(int data) {
        this->data = data;
        this->next = NULL;
    }
};
```

Basic Operations (Singly Linked List)

1. Insertion at Head:

```
void insertAtHead(Node* &head, int data) {
    Node* newNode = new Node(data);
    newNode->next = head;
    head = newNode;
}
```

2. Traversal/Printing:

```
void printList(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
```

Doubly Linked List

A Doubly Linked List (DLL) is a two-way list in which each node has two pointers: `next` and `prev`. The `next` pointer refers to the next node, and the `prev` pointer refers to the previous node. This allows traversal in both forward and backward directions, unlike singly linked lists.

Node Structure for Doubly Linked List

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

    // Default constructor
    Node() {
        data = 0;
        next = NULL;
        prev = NULL;
    }

    // Parameterized Constructor
    Node(int data) {
        this->data = data;
        this->next = NULL;
        this->prev = NULL;
    }
};
```

Basic Operations (Doubly Linked List)

1. Insertion at Beginning:

```
void insertAtBeginning(Node* &head, int data) {
    Node* newNode = new Node(data);
    if (head == NULL) {
        head = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
}
```

2. Insertion at End:

```
void insertAtEnd(Node* &head, int data) {
    Node* newNode = new Node(data);
    if (head == NULL) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}
```

3. Deletion from Beginning:

```
void deleteFromBeginning(Node* &head) {
    if (head == NULL) {
        cout << "List is empty." << endl;
        return;
    }
    Node* temp = head;
    head = head->next;
    if (head != NULL) {
        head->prev = NULL;
    }
    delete temp;
}
```

4. Traversal (Forward):

```
void printListForward(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
```

5. Traversal (Backward):

```
void printListBackward(Node* head) {
    Node* temp = head;
    if (temp == NULL) return; // Handle empty list
    while (temp->next != NULL) {
        temp = temp->next;
    }
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->prev;
    }
    cout << endl;
}
```

Comparison of Singly and Doubly Linked Lists

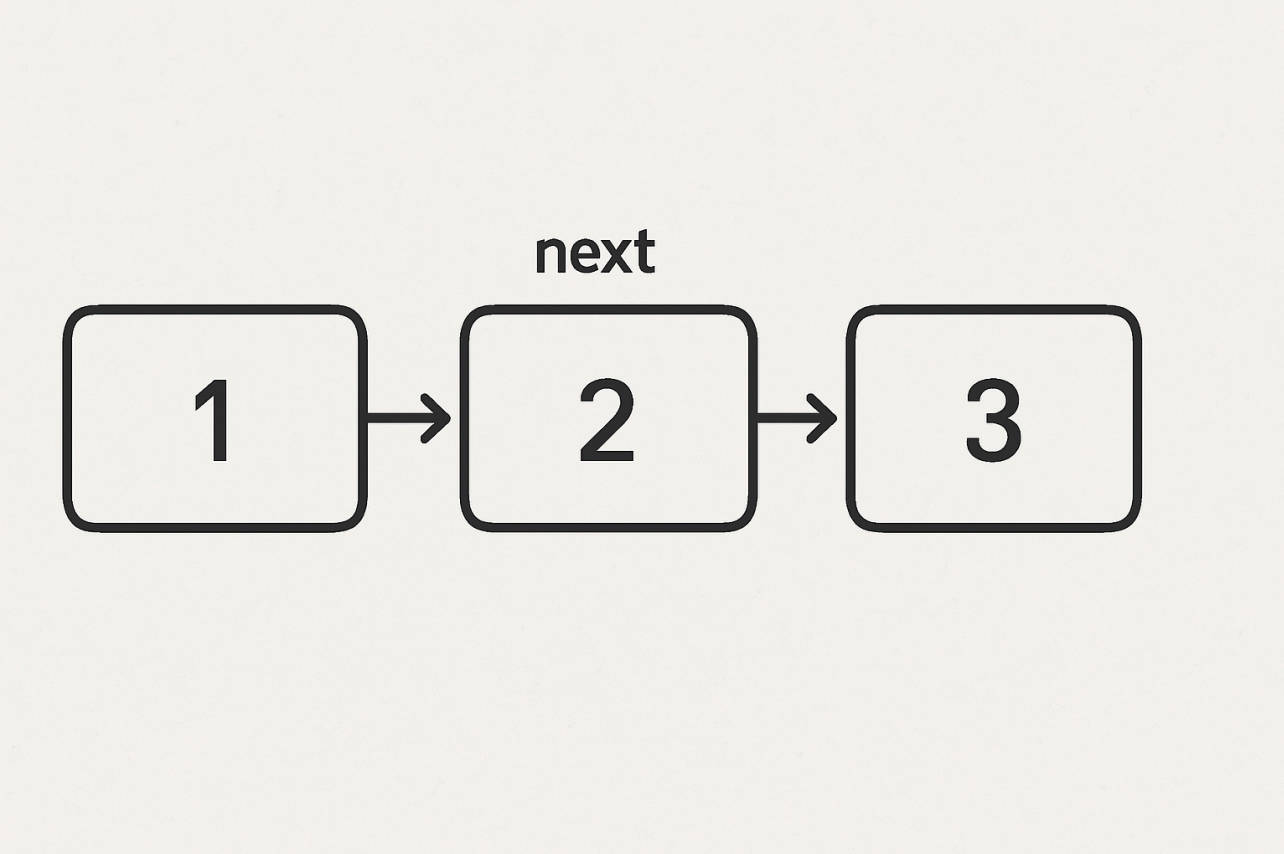
Feature	Singly Linked List	Doubly Linked List
Traversal	Unidirectional (forward only)	Bidirectional (forward and backward)
Memory	Less memory per node (one pointer)	More memory per node (two pointers)
Insertion/Deletion	Efficient at ends, requires traversal for middle	More efficient for middle (no full traversal needed)
Complexity	Simpler to implement	More complex to implement

When to Use Which?

- **Singly Linked List:** Ideal when memory is a critical concern and traversal is primarily in one direction (e.g., implementing a stack or queue).

- **Doubly Linked List:** Preferred when frequent insertions/deletions are needed at arbitrary positions, or when bidirectional traversal is required (e.g., implementing an LRU cache or a browser history).

This cheatsheet provides a fundamental understanding of singly and doubly linked lists with C++ examples. For more advanced operations and optimizations, refer to specialized data structure resources.



Singly Linked List: Detailed Operations and Considerations

Node Structure Revisited

Each node in a singly linked list is a self-referential structure. It contains the actual data (`data`) and a pointer (`next`) that points to the memory address of the subsequent node in the sequence. The last node's `next` pointer typically points to `NULL` or `nullptr` to signify the end of the list.

```cpp
class Node {
public:
    int data;  // Data stored in the node
    Node* next; // Pointer to the next node

    // Constructor to initialize a new node
    Node(int val) : data(val), next(nullptr) {}
};
```

Advanced Operations (Singly Linked List)

Beyond basic insertion at the head and traversal, here are other common operations:

1. Insertion at Tail:

Adding a new node at the end of the list requires traversing the list to find the last node. If the list is empty, the new node becomes the head.

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

2. Insertion at a Specific Position:

Inserting a node at a given position (e.g., after `n` nodes) involves traversing to the `(n-1)`th node and then adjusting pointers. Edge cases include inserting at the beginning (position 0) or beyond the end of the list.

```
void insertAtPosition(Node* &head, int data, int position) {
    if (position < 0) {
        std::cout << "Invalid position.\n";
        return;
    }
    if (position == 0) {
        insertAtHead(head, data);
        return;
    }

    Node* newNode = new Node(data);
    Node* temp = head;
    for (int i = 0; temp != nullptr && i < position - 1; ++i) {
        temp = temp->next;
    }

    if (temp == nullptr) {
        std::cout << "Position out of bounds.\n";
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}
```

3. Deletion of a Node (by Value):

To delete a node with a specific value, you must find the node and its preceding node to correctly adjust pointers. Special handling is needed if the head node is to be deleted.

```
void deleteNodeByValue(Node* &head, int value) {
    if (head == nullptr) {
        std::cout << "List is empty, nothing to delete.\n";
        return;
    }

    // If head node itself holds the value to be deleted
    if (head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
        std::cout << "Node with value " << value << " deleted.\n";
        return;
    }

    Node* current = head;
    Node* prev = nullptr;
    while (current != nullptr && current->data != value) {
        prev = current;
        current = current->next;
    }

    // If value was not present in list
    if (current == nullptr) {
        std::cout << "Value " << value << " not found in the list.\n";
        return;
    }

    // Unlink the node from linked list
    prev->next = current->next;
    delete current; // Free memory
    std::cout << "Node with value " << value << " deleted.\n";
}
```

4. Deletion at a Specific Position:

Similar to insertion, deleting at a position requires finding the node before the target node.

```
void deleteAtPosition(Node* &head, int position) {
    if (head == nullptr) {
        std::cout << "List is empty, nothing to delete.\n";
        return;
    }
    if (position < 0) {
        std::cout << "Invalid position.\n";
        return;
    }

    Node* temp = head;

    // If head needs to be removed
    if (position == 0) {
        head = temp->next;
        delete temp;
        std::cout << "Node at position " << position << " deleted.\n";
        return;
    }

    // Find previous node of the node to be deleted
    for (int i = 0; temp != nullptr && i < position - 1; ++i) {
        temp = temp->next;
    }

    // If position is more than number of nodes
    if (temp == nullptr || temp->next == nullptr) {
        std::cout << "Position out of bounds.\n";
        return;
    }

    // Node temp->next is the node to be deleted
    // Store pointer to the next of node to be deleted
    Node* nextNode = temp->next->next;
    delete temp->next; // Free memory
    temp->next = nextNode;
    std::cout << "Node at position " << position << " deleted.\n";
}
```

5. Searching for an Element:

Traverse the list from the head until the element is found or the end of the list is reached.

```
bool search(Node* head, int key) {
    Node* current = head;
    while (current != nullptr) {
        if (current->data == key) {
            return true;
        }
        current = current->next;
    }
    return false;
}
```

Time Complexities (Singly Linked List)

Operation	Time Complexity	Notes
Insertion at Head	O(1)	Only pointer updates required
Insertion at Tail	O(N)	Requires traversal to the last node
Insertion at Position	O(N)	Requires traversal to (position-1)th node
Deletion at Head	O(1)	Only pointer updates required
Deletion at Tail	O(N)	Requires traversal to the second-to-last node
Deletion at Position	O(N)	Requires traversal to (position-1)th node
Search	O(N)	May need to traverse the entire list
Access by Index	O(N)	Requires traversal from the head

Doubly Linked List: Detailed Operations and Considerations

Node Structure Revisited

Each node in a doubly linked list has three parts: `data`, a `next` pointer to the subsequent node, and a `prev` pointer to the preceding node. This bidirectional linking allows for more flexible traversal and manipulation.

```
 class Node {
public:
    int data;   // Data stored in the node
    Node* next; // Pointer to the next node
    Node* prev; // Pointer to the previous node

    // Constructor to initialize a new node
    Node(int val) : data(val), next(nullptr), prev(nullptr) {}
};
```

Advanced Operations (Doubly Linked List)

Doubly linked lists offer more efficient operations for insertions and deletions in the middle of the list compared to singly linked lists.

1. Insertion at a Specific Position:

Inserting a node at a given position is more efficient than in a singly linked list if you have a pointer to the node *before* or *after* the insertion point, as you can traverse in both directions.

```
void insertAtPositionDLL(Node* &head, int data, int position) {
    if (position < 0) {
        std::cout << "Invalid position.\n";
        return;
    }
    if (position == 0) {
        insertAtBeginning(head, data);
        return;
    }

    Node* newNode = new Node(data);
    Node* current = head;
    for (int i = 0; current != nullptr && i < position - 1; ++i) {
        current = current->next;
    }

    if (current == nullptr) {
        std::cout << "Position out of bounds.\n";
        return;
    }

    newNode->next = current->next;
    newNode->prev = current;
    if (current->next != nullptr) {
        current->next->prev = newNode;
    }
    current->next = newNode;
}
```

2. Deletion of a Node (by Value):

Deletion by value is also more straightforward as you don't need a separate `prev` pointer during traversal; the node itself contains a pointer to its predecessor.

```cpp
void deleteNodeByValueDLL(Node* &head, int value) {
    if (head == nullptr) {
        std::cout << "List is empty, nothing to delete.\n";
        return;
    }

    Node* current = head;

    // Traverse to find the node to delete
    while (current != nullptr && current->data != value) {
        current = current->next;
    }

    if (current == nullptr) {
        std::cout << "Value " << value << " not found in the list.\n";
        return;
    }

    // If node to be deleted is the head node
    if (current == head) {
        head = current->next;
    }

    // If not the last node, change next of previous node
    if (current->prev != nullptr) {
        current->prev->next = current->next;
    }

    // If not the first node, change prev of next node
    if (current->next != nullptr) {
        current->next->prev = current->prev;
    }

    delete current; // Free memory
    std::cout << "Node with value " << value << " deleted.\n";
}
```

3. Deletion at a Specific Position:

```cpp
void deleteAtPositionDLL(Node* &head, int position) {
    if (head == nullptr) {
        std::cout << "List is empty, nothing to delete.\n";
        return;
    }
    if (position < 0) {
        std::cout << "Invalid position.\n";
        return;
    }

    Node* current = head;

    // Traverse to the node at the given position
    for (int i = 0; current != nullptr && i < position; ++i) {
        current = current->next;
    }

    if (current == nullptr) {
        std::cout << "Position out of bounds.\n";
        return;
    }

    // If node to be deleted is the head node
    if (current == head) {
        head = current->next;
    }

    // If not the last node, change next of previous node
    if (current->prev != nullptr) {
        current->prev->next = current->next;
    }

    // If not the first node, change prev of next node
    if (current->next != nullptr) {
        current->next->prev = current->prev;
    }

    delete current; // Free memory
    std::cout << "Node at position " << position << " deleted.\n";
}
```

Time Complexities (Doubly Linked List)

Operation	Time Complexity	Notes
Insertion at Head	O(1)	Only pointer updates required
Insertion at Tail	O(N)	Requires traversal to the last node
Insertion at Position	O(N)	Requires traversal to the position
Deletion at Head	O(1)	Only pointer updates required
Deletion at Tail	O(N)	Requires traversal to the last node
Deletion at Position	O(N)	Requires traversal to the position
Search	O(N)	May need to traverse the entire list
Access by Index	O(N)	Requires traversal from either end

Note on O(N) for Insertion/Deletion at Position: While the *actual pointer manipulation* is O(1) once the position is found, the *process of finding the position* still takes O(N) time in the worst case (e.g., if the position is near the end of a long list). However, if you already have a pointer to the node *before* or *at* the insertion/deletion point, the operation becomes O(1).

Advanced Comparison and Use Cases

Aspect	Singly Linked List	Doubly Linked List
Memory Footprint	Smaller (one pointer per node)	Larger (two pointers per node)
Traversal Direction	Unidirectional (forward only)	Bidirectional (forward and backward)
Ease of Implementation	Simpler to implement	More complex due to managing two pointers
Deletion Efficiency	Requires a pointer to the previous node for efficient deletion (O(N) to find previous)	More efficient deletion (O(1) if node pointer is known, O(N) to find)
Reverse Traversal	Not directly supported without external stack/recursion	Directly supported
Common Applications	Stacks, Queues, hash tables, simple lists	LRU caches, browser history, undo/redo functionality, advanced data structures (e.g., skip lists)

Choosing the Right Linked List

- **Singly Linked List:** Choose when memory is a primary concern, and you mostly need to add/remove elements from one end (like a stack or queue), or when forward-only traversal is sufficient. They are simpler to implement and debug.
- **Doubly Linked List:** Opt for a doubly linked list when you need to frequently insert or delete elements from arbitrary positions within the list, or when bidirectional traversal is a requirement. The added memory overhead for the `prev` pointer is a trade-off for increased flexibility and efficiency in certain operations.

Both types of linked lists are fundamental data structures, and understanding their nuances is key to selecting the most appropriate one for a given problem or application. While `std::list` in C++ STL is typically implemented as a doubly linked list, understanding the underlying principles allows for custom implementations and optimization when needed.