Doubly Linked List Tutorial - GeeksforGeeks

Source: https://www.geeksforgeeks.org/doubly-linked-list/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Doubly Linked List Tutorial Last Updated : 19 Sep, 2025 A doubly linked list is a more complex data structure than a singly linked list, but it offers several advantages. The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous node and a pointer to the next node. This allows for quick and easy insertion and deletion of nodes from the list, as well as efficient traversal of the list in both directions. Representation of Doubly Linked List in Data Structure In a data structure, a doubly linked list is represented using nodes that have three fields: Data A pointer to the next node (next) A pointer to the previous node (prev) Node Definition Here is how a node in a Doubly Linked List is typically represented: Try it on GfG Practice C++ #include <iostream> using namespace std ; class Node { public : // To store the Value or data int data ; // Pointer to point the Previous Element Node * prev ; // Pointer to point the Next Element Node * next ; // Constructor Node (int d) { data = d ; prev = nullptr ; next = nullptr ; } }; Java class Node { // To store the Value or data. int data ; // Reference to the Previous Node Node prev ; // Reference to the next Node Node next ; // Constructor Node (int d) { data = d ; prev = next = null ; } }; Python class Node : def __init__ (self , data): # To store the value or data. self . data = data # Reference to the previous node self . prev = None # Reference to the next node self . next = None C# class Node { // To store the value or data public int Data ; // Pointer to the next node public Node Next ; // Pointer to the previous node public Node Prev ; // Constructor public Node (int d){ Data = d ; Prev = Next = null ; } } JavaScript class Node { constructor (data){ // To store the value or data. this . data = data ; // Reference to the previous node this . prev = null ; // Reference to the next node this . next = null ; } } Each node in a Doubly Linked List contains the data it holds, a pointer to the next node in the list, and a pointer to the previous node in the list. By linking these nodes together through the next and prev pointers, we can traverse the list in both directions (forward and backward), which is a key feature of a Doubly Linked List. Creating a Doubly Linked List with 4 Nodes Create the head node. Allocate a node and set head to it. Its prev and next should be null/None. Create the next node and link it to head. head.next = new Node(value2) head.next.prev = head Create further nodes the same way. For the third node: => head.next.next = new Node(value3) => head.next.next.prev = head.next Repeat until you have the required nodes. Ensure the tail's next is null. The last node you created must have next == null Set / keep track of head (and optionally tail). Use head to access the list from the front. Keeping a tail pointer simplifies appends. C++ #include <iostream> using namespace std ; class Node { public : int data ; Node * prev ; Node * next ; Node (int value) { data = value ; prev = nullptr ; next = nullptr ; } }; int main () { // Create the first node (head of the list) Node * head = new Node (10); // Create and link the second node head -> next = new Node (20); head -> next -> prev = head ; // Create and link the third node head -> next -> next = new Node (30); head -> next -> next -> prev = head -> next ; // Create and link the fourth node head -> next -> next -> next = new Node (40); head -> next -> next -> next -> prev = head -> next -> next ; // Traverse the list forward and print elements Node * temp = head ; while (temp != nullptr) { cout << temp -> data ; if (temp -> next != nullptr) { cout << " <-> " ; } temp = temp -> next ; } return 0 ; } Java class Node { int data ; Node prev ; Node next ; Node (int value) { data = value ; prev = null ; next = null ; } } class GfG { public static void main (String [] args) { // Create the first node (head of the list) Node head = new Node (10); // Create and link the second node head . next = new Node (20); head . next . prev = head ; // Create and link the third node head . next . next = new Node (30); head . next . next . prev = head . next ; // Create and link the fourth node head . next . next . next = new Node (40); head . next . next . next . prev = head . next . next ; // Traverse the list forward and print elements Node temp = head ; while (temp != null) { System . out . print (temp . data); if (temp . next != null) { System . out . print (" <-> "); } temp = temp . next ; } } } Python class Node : def __init__ (self , value): self . data = value self . prev = None self . next = None if __name__ == "__main__" : # Create the first node (head of the list) head = Node (10) # Create and link the second node head . next = Node (20) head . next . prev = head # Create and link the third node head . next . next = Node (30) head . next . next . prev = head . next # Create and link the fourth node head . next . next . next = Node (40) head . next . next . next . prev = head . next . next # Traverse the list forward and print elements temp = head while temp is not None : print (temp . data , end = "") if temp . next is not None : print (" <-> " , end = "") temp = temp . next C# using System ; class Node { public int data ; public Node prev ; public Node next ; public Node (int value) { data = value ; prev = null ; next = null ; } } class GfG { static void Main (string [] args) { // Create the first node (head of the list) Node head = new Node (10); // Create and link the second node head . next = new Node (20); head . next . prev = head ; // Create and link the third node head . next . next = new Node (30); head . next . next . prev = head . next ; // Create and link the fourth node head . next . next . next = new Node (40); head . next . next . next . prev = head . next . next ; // Traverse the list forward and print elements Node temp = head ; while (temp != null) { Console . Write (temp . data); if (temp . next != null) { Console . Write (" <-> "); } temp = temp . next ; } } } JavaScript class Node { constructor (value) { this . data = value ; this . prev = null ; this . next = null ; } } // Driver Code // Create the first node (head of the list) let head = new Node (10); // Create and link the second node head . next = new Node (20); head . next . prev = head ; // Create and link the third node head . next . next = new Node (30); head . next . next . prev = head . next ; // Create and link the fourth node head . next . next . next = new Node (40); head . next . next . next . prev = head . next . next ; // Traverse the list forward and print elements let temp = head ; let output = "" ; while (temp !== null) { output += temp . data ; if (temp . next !== null) { output += " <-> " ; } temp = temp . next ; } console . log (output); Output 10 <-> 20 <-> 30 <-> 40 Common Operation in Doubly Linked List Traversal : Display Linked List Elements Insertion : At the Beginning , At the End and At the specific position Deletion : From the Beginning , From End and From a Specific Position Application of Doubly Linked List Advantages of Doubly Linked List Bidirectional Traversal - You can traverse forward (using next) as well as backward (using prev). Efficient Deletion - Given a pointer to a node, you can delete it in O(1) time (no need to traverse from the head), since you can update both prev and next. Insertion at Both Ends - Insertion at head or tail is efficient because you can update both directions easily. Easy to Implement Deque / Navigation Features - Useful for undo/redo, browser history, and music playlist navigation, where both forward and backward movement is needed. Disadvantages of Doubly Linked List Extra Memory Per Node - Each node requires an additional pointer (prev), making DLL more memory-consuming than singly linked list. More Complex Implementation - Both prev and next must be handled carefully during insertion and deletion, which increases chances of errors (broken links, null pointer issues) Slower Operations Due to Overhead - Extra pointer manipulations during insertion/deletion cause slightly more overhead compared to singly linked list. Not Cache-Friendly - Like singly linked list, nodes are scattered in memory, so traversals may be slower compared to arrays due to poor locality of reference. Comment Article Tags: Article Tags: DSA doubly linked list DSA Tutorials