Self-reading notes

. What is a Linked List?

A **Linked List** is a linear data structure where elements (called **nodes**) are not stored at continuous memory locations. Each node stores:

- **Data** (actual value)
- **Link / Pointer** (address of the next node)

Unlike arrays, linked lists are **dynamic** (size can grow or shrink at runtime).

Real-Life Example

Think of a **treasure hunt**:

- Each clue tells you **where the next clue is**.
- You cannot jump directly to clue 5 without following clues 1 → 2 → 3 → 4.

Corporate Example

- **Customer support ticket chain**: each ticket points to the next follow-up.
- **Browser history**: each page stores the link to the next visited page.

->Now the question is What is **dynamic ?? And runtime here?**

What is Memory Allocation?

Memory allocation means reserving space in RAM so a program can store data while it runs.

There are two main types:

1. **Static Memory Allocation**
2. **Dynamic Memory Allocation**

🔹 **STATIC MEMORY ALLOCATION**

Definition

In static memory allocation, memory size is fixed at compile time (before the program runs).

->Memory is allocated automatically
-> Size cannot change during execution

Where is static memory stored?

- **Stack memory**
- **Data segment (for global/static variables)**

Characteristics

- **Size is known in advance**
- **Faster than dynamic allocation**
- **Cannot resize**
- **Memory is freed automatically**
- **Risk of wastage if size is larger than needed**

Examples of Static Allocation

- **Example 1: Simple variable**
  ```
  int a = 10;
  ```

  Memory for a is fixed
  Allocated at compile time

Example 2: Static array

```
int arr[5];
```

-> Always reserves memory for 5 integers
->Even if you use only 2 elements

Example 3: Function local variable (Stack)

```
void func() {
  int x = 20;
 }
```

-> x is created when function starts
-> Destroyed when function ends

DYNAMIC MEMORY ALLOCATION

Definition

In dynamic memory allocation, memory is allocated at runtime (while program is running).

-> Size can be decided by user input
-> Memory is allocated manually

Where is dynamic memory stored?

- **Heap memory**

Characteristics

- **Size decided at runtime**
- **Flexible (can grow/shrink)**
- **Slightly slower**
- **Must be freed manually**
- **Risk of memory leak if not freed**

-> How dynamic allocation works (C++)

- new → allocate memory
- delete → free memory

Examples of Dynamic Allocation

Example 1: Dynamic variable

```
int* ptr = new int;

*ptr = 10;

delete ptr;
```

Memory allocated in heap
-> ptr stores address
-> delete frees memory

STACK (Static)

----

int a

int arr[5]

local variables

HEAP (Dynamic)

----

new int

new int[10]

malloc()

Static Memory:

Fixed chairs in classroom
Even if students are less, chairs stay

Dynamic Memory:

Chairs arranged as students arrive
More students → more chairs
No students → remove chairs

When to Use What?

Use Static when:

- Size is fixed
- Small programs
- Fast execution needed

Use Dynamic when:

- Size unknown
- Large data (DSA, linked list, trees)
- User input decides size

Now The question is what is compile time and run time ?

What is Compile Time?

Definition

Compile time is the phase when your **source code is converted into machine code** by the compiler **before the program runs**.

Errors found here are called **compile-time errors**.

What happens at Compile Time?

- Syntax checking
- Type checking
- Memory allocation for **static variables**
- Code optimization
- Creation of .exe / object file

What is Runtime?

Definition

Runtime is the phase when the **program is actually executing** after compilation.

Errors found here are called **runtime errors**.

What happens at Runtime?

- User input is taken
- Dynamic memory allocation (new, malloc)
- Function calls executed
- Logic is processed
- Output is produced

How many minutes?

Compile time:

Takes **milliseconds to seconds** (depends on code size)

Runtime:
Can take **seconds, minutes, hours, or forever**
(depends on program logic)

Singly Linked List

2.1 Node Structure

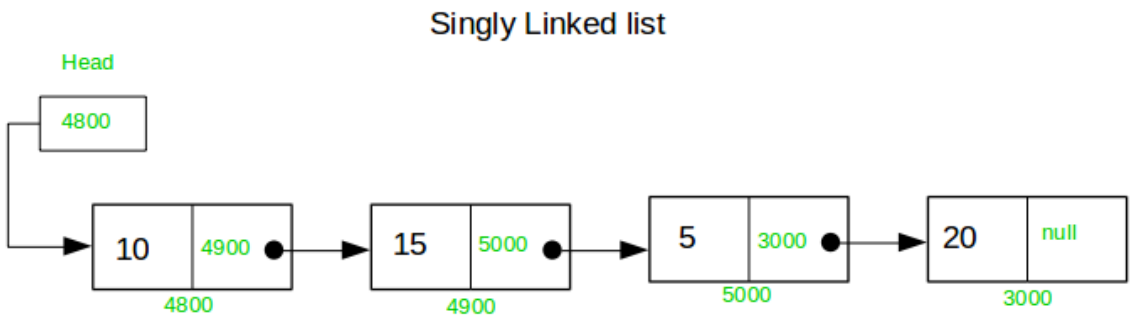
Each node contains:

- data
- next (pointer to the next node)

[Data | Next] → [Data | Next] → [Data | NULL]

Head and Tail

- **Head** → First node of the list
- **Tail** → Last node (points to NULL)



Head and Tail

- **Head** → First node of the list
- **Tail** → Last node (points to NULL)

Real-Life Example 🚶

A **line of people**:

- Each person knows who is standing **next** to them.
- Last person has no one after them.

How to implement the Linkelist in programming

```
#include <iostream>
using namespace std;
struct node
{
  int data;
  node* next;
};
int main()
{
  node *newnode, *head = NULL, *temp;
  int choice = 1;
  while(choice)
  {
    newnode = new node();
    cout << "Enter data: ";
    cin >> newnode->data;
    newnode->next = NULL;
    if(head == NULL)
    {
      head = temp = newnode;
    }
    else
    {
      temp->next = newnode;
      temp = newnode;
    }
    cout << "Do you want to continue 0/1?: ";
```

```
    cin >> choice;
  }
  cout << "Linked List: ";
  temp = head;
  while(temp != NULL)
  {
    cout << temp->data << " -> ";
    temp = temp->next;
  }
  cout << "NULL" << endl;
  return 0;
}
```

Create & Display a Singly Linked List (Dynamic Memory)

---

1. Header Files

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

- Includes input/output functions
- cin, cout become directly usable

---

2. Structure Definition

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

Meaning:

Each **node** contains:

- data → stores value
- next → stores address of next node

This is the **building block of a linked list**.

---

3. Main Function

int main()

{

---

4. Pointer Declarations

node *newnode, *head = NULL, *temp;

Explanation:

- newnode → stores address of newly created node
- head → points to first node (initially NULL)
- temp → used for traversal and linking

---

5. Choice Variable

int choice = 1;

- Controls the while loop
- 1 → continue
- 0 → stop

---

6. While Loop (Create Nodes)

while(choice)

- Loop runs **until user enters 0**
- Creates nodes dynamically

---

7. Dynamic Memory Allocation

newnode = new node();

- Memory allocated in **heap**
- Node is created at runtime

🔹 8. Input Data

```
cout << "Enter data: ";
cin >> newnode->data;
```

- Stores user input in node's data

🔹 9. Set Next Pointer

```
newnode->next = NULL;
```

- New node is last node initially

🔹 10. Check for First Node

```
if(head == NULL)
{
  head = temp = newnode;
}
```

Explanation:

- If list is empty:
    - head points to first node
    - temp also points to first node

🔹 11. Insert at End

```
else
{
  temp->next = newnode;
  temp = newnode;
}
```

Explanation:

- Link previous node to new node
- Move temp to last node

---

12. Continue or Stop

```
cout << "Do you want to continue 0/1?: ";
cin >> choice;
```

- User decides whether to add more nodes

---

13. Display Linked List

```
cout << "Linked List: ";
temp = head;
```

- Start traversal from head

---

14. Traversal Loop

```
while(temp != NULL)
{
   cout << temp->data << " -> ";
   temp = temp->next;
}
```

Explanation:

- Print data
- Move to next node
- Stop when temp becomes NULL

---

15. End of List

```
cout << "NULL" << endl;
```

- Indicates end of linked list

DRY RUN

Suppose user input:

Enter data: 10

Do you want to continue 0/1?: 1

Enter data: 20

Do you want to continue 0/1?: 1

Enter data: 30

Do you want to continue 0/1?: 0

First Iteration

- newnode → 10
- head = temp = newnode

head → [10 | NULL]

Second Iteration

- newnode → 20
- temp->next = newnode
- temp = newnode

head → [10 | *] → [20 | NULL]

Third Iteration

- newnode → 30
- Linked to end

head → [10 | *] → [20 | *] → [30 | NULL]

Traversal Output

10 -> 20 -> 30 -> NULL

Memory Used

- **Heap** → Nodes (new node)
- **Stack** → Pointers (head, temp, newnode)

3. Insertion in Singly Linked List

3.1 Insertion at Head (Beginning)

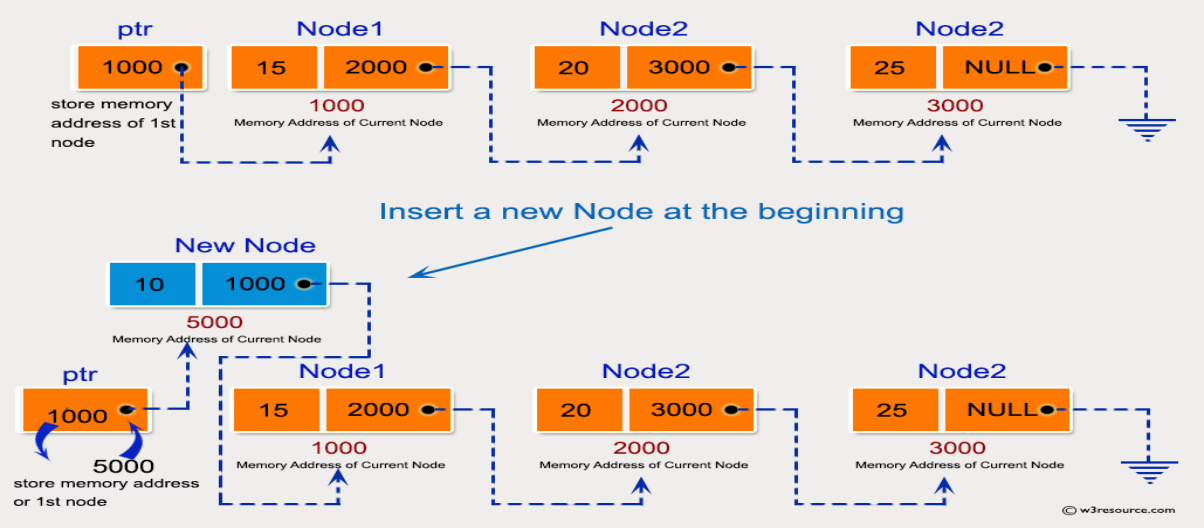
Steps:

1. Create a new node
2. New node's next → current head
3. Update head to new node

Use Case

- **Undo operation** in editors (latest action first)
- **New notifications** shown at top

```
// Function to insert node at the beginning
node* insertAtBeginning(node* head, int value) {
    node* newnode = new node();
    newnode->data = value;
    newnode->next = head;
    head = newnode;
    return head;
}
```



Insertion at Tail (End)

Steps:

1. Create new node
2. Traverse till last node
3. Last node's next → new node

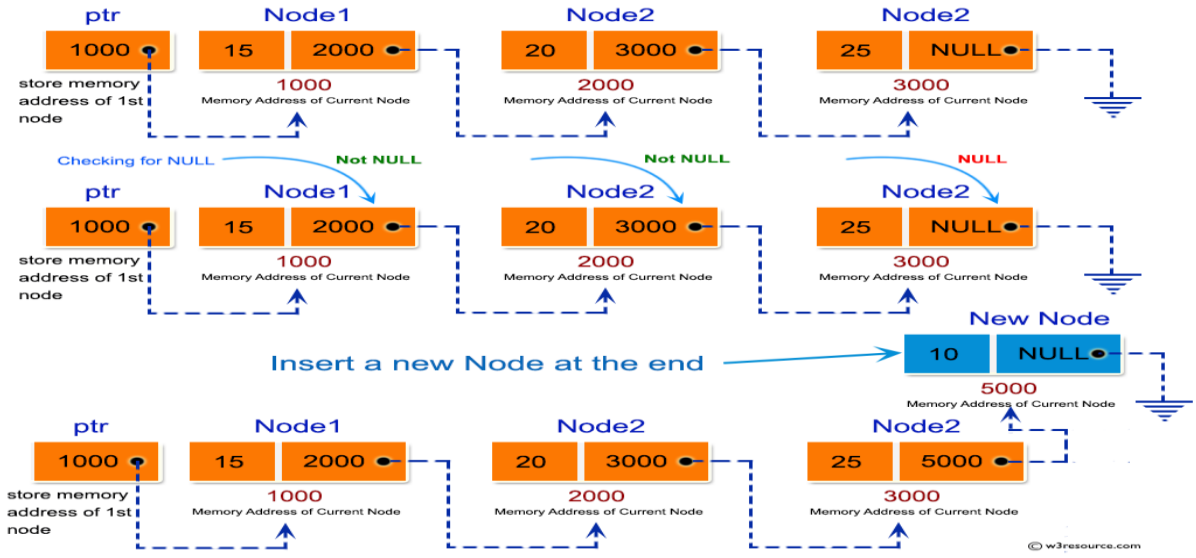
Use Case

- **Music playlist** (songs added at the end)
- **Job queue** in companies

```
// Function to insert node at the end
node* insertAtEnd(node* head, int value) {
    node* newnode = new node();
    newnode->data = value;
    newnode->next = NULL;

    if(head == NULL) {
        head = newnode;
    } else {
        node* temp = head;
        while(temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newnode;
    }

    return head;
}
```



Insertion in the Middle (At Position)

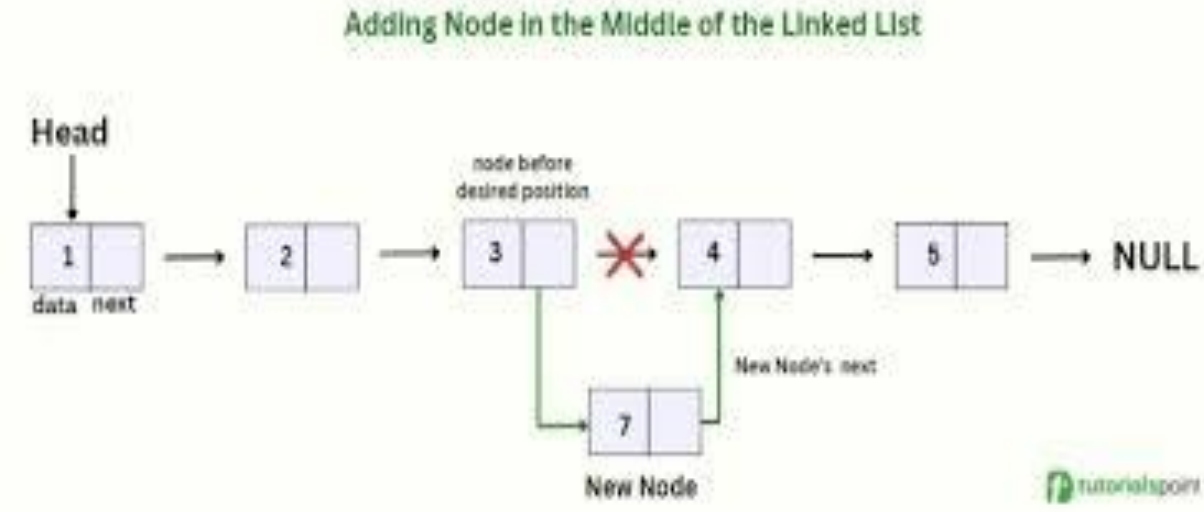
Steps:

1. Traverse till position − 1
2. New node's next → current node's next
3. Current node's next → new node

Use Case

- Insert a **new task** in the middle of a project timeline

```
/ Function: Insert at Position
node* insertAtPos(node* head, int pos, int val) {
    newnode = new node();
    newnode->data = val;
    // Insert at beginning
    if (pos == 1) {
        newnode->next = head;
        head = newnode;
        return head;
    }
    temp = head;
    for (int i = 1; i < pos - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        cout << "Invalid Position. Insertion not possible.\n";
        return head;
    }
    newnode->next = temp->next;
    temp->next = newnode;
    return head;
}
```



Deletion in Singly Linked List

4.1 Deletion by Value

Steps:

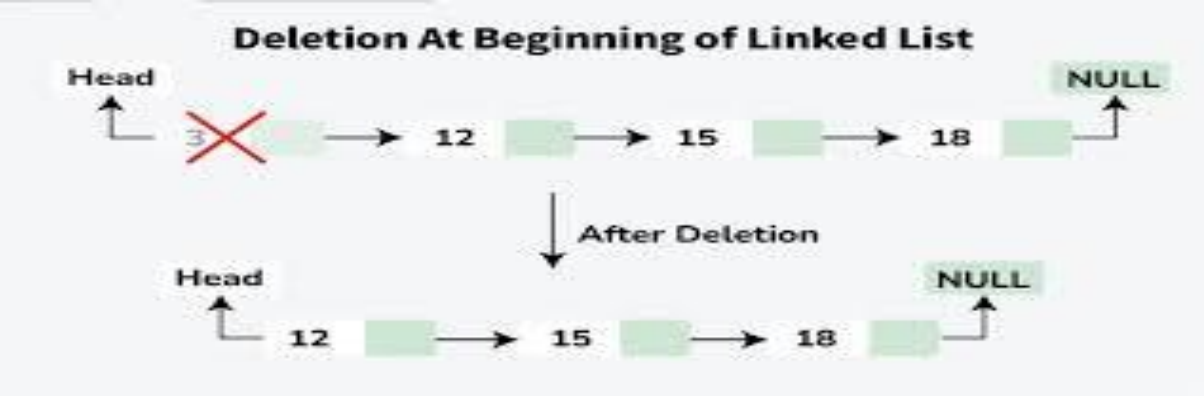
1. Search the value
2. Store previous node
3. Previous node's next → current node's next

Corporate Example

- Removing an **inactive user** from a linked user list

```
Node* deleteFromBeginning(Node* head) {
    if (head == NULL) {
        cout << "List is empty, nothing to delete." << endl;
        return head;
    }

    Node* temp = head;
    head = head->next;   // Move head to next node
    delete temp;         // Free old head
    return head;
}
```



Deletion by Position

Steps:

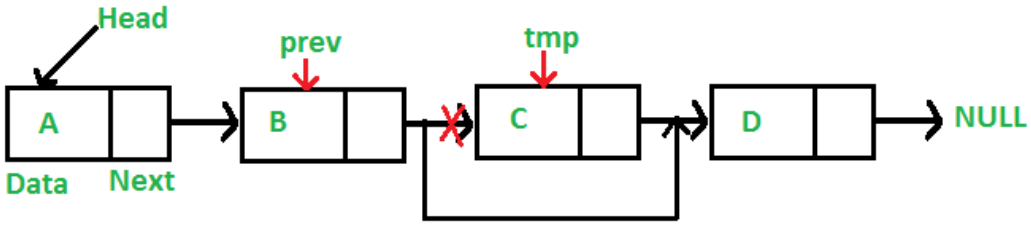
1. Traverse to position − 1
2. Skip the target node

Use Case

- Removing a task from a **priority list**

```
// Function to delete node from a specified position
Node* deleteFromPos(Node* head)
{
  Node* temp;
  Node* nextnode;
  int pos, i = 1;
  if (head == nullptr)
  {
    cout << "List is empty, nothing to delete." << endl;
    return head;
  }
  cout << "Enter position to delete: ";
  cin >> pos;
  // Deleting first node
  if (pos == 1)
  {
    temp = head;
    head = head->next;
    delete temp;
    return head;
  }
  temp = head;
  while (i < pos - 1 && temp->next != nullptr)
  {
    temp = temp->next;
    i++;
  }
```

```
    if (temp->next == nullptr)
    {
        cout << "Invalid position." << endl;
        return head;
    }
    nextnode = temp->next;
    temp->next = nextnode->next;
    delete nextnode;
    return head;
}
```



Length Calculation

Count number of nodes while traversing.

Corporate Use

- Count number of **active sessions** on a server

```
// Function to count number of nodes
int getLength() {
    int count = 0;
    Node* temp = head;
    while (temp != nullptr) {
        count++;
        temp = temp->next;
    }
    return count;
}
```

Doubly Linked List

Structure

Each node contains:

- prev
- data
- next

Advantages

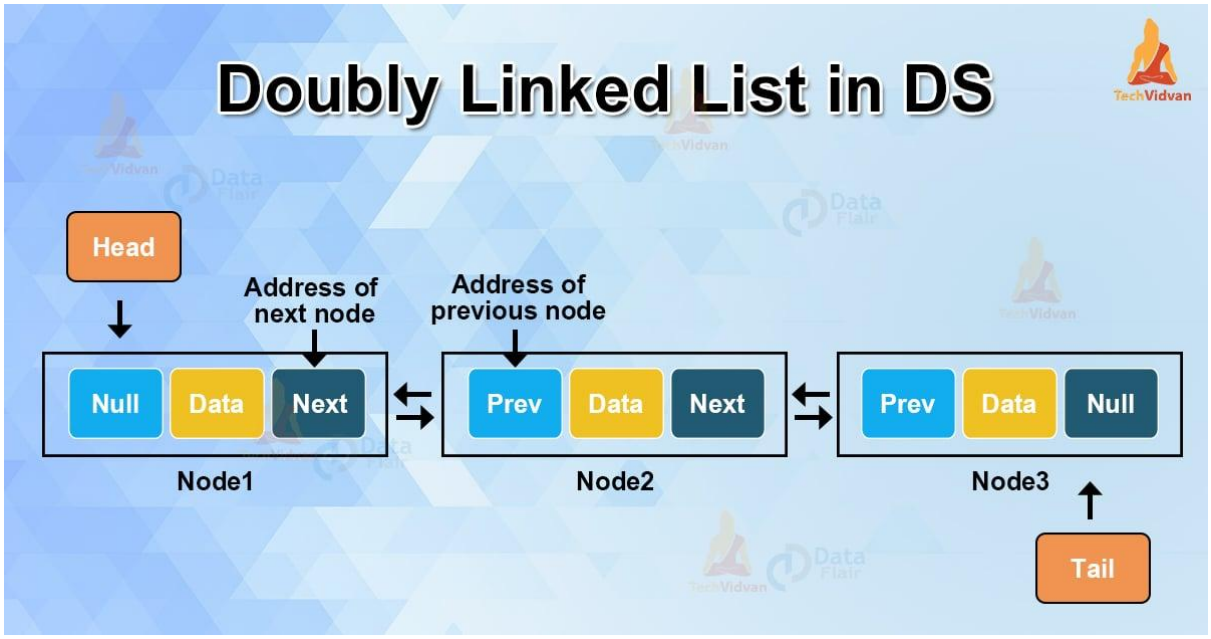
- Traversal in **both directions**
- Easy deletion

Real-Life Example

- **Browser back & forward buttons**
- **Undo / Redo** functionality

Corporate Example

- Navigation between pages in enterprise dashboards



Implementation of Doubly linkelist

```
#include <iostream>
using namespace std;
//sturcture of a node
struct node
{
   int data;
   node *next;
   node *prev;
};
node *head = nullptr, *newnode = nullptr;
void create()
{
   head = nullptr;
   node *temp = nullptr;
   int choice = 1;

   while (choice)
   {
      newnode = new node();
      cout << "Enter data = ";
      cin >> newnode->data;

      newnode->next = nullptr;
      newnode->prev = nullptr;

      if (head == nullptr)  // if the list is empty
```

```
    {
        head = temp = newnode;
    }
    else
    {
        temp->next = newnode;
        newnode->prev = temp;
        temp = newnode;
    }

    cout << "Do you want to continue (1/0): ";
    cin >> choice;
  }
}
void display() //print the list
{
  node *temp = head; //

  cout << "Doubly Linked List elements are:\n";
  while (temp != nullptr)
  {
    cout << temp->data << " ";  //printing data by using temp ptr
    temp = temp->next;  //move the temp to next node
  }
}

int main()
{
```

```
    create();
    display();
    return 0;
}
```

Real-Life & Corporate Example

Real-Life Example

Browser Navigation:

- Back button → uses prev
- Forward button → uses next

Corporate Example

Employee Records System:

- HR can move to previous or next employee profile easily

Traversal in Doubly Linked List

Forward Traversal

Move from head to tail using next pointer.

Example:

10 ⇄ 20 ⇄ 30

Output: 10 20 30

Backward Traversal

Move from tail to head using prev pointer.

Output: 30 20 10

Use Case

- Undo / Redo operations
- Media playlist navigation

Insertion Functions in Doubly Linked List

Insertion at Beginning (Head)

Steps:

1. Create new node
2. New node next → current head
3. Current head prev → new node
4. Update head

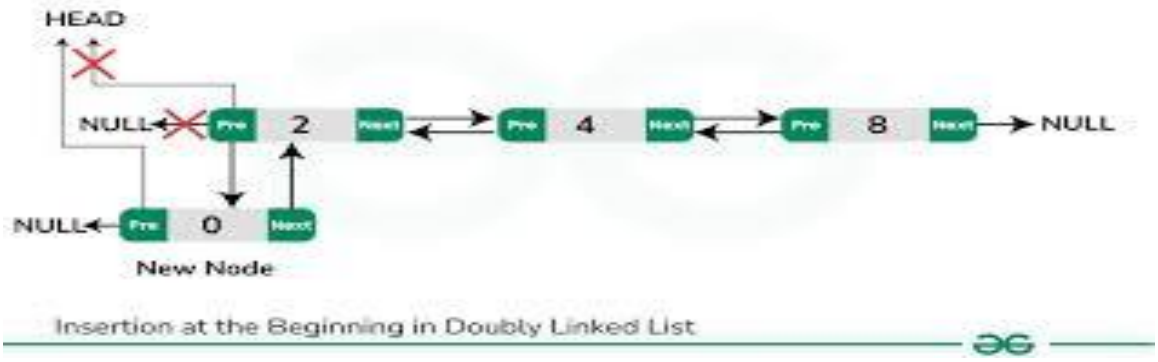
Before: 10 ⇄ 20 ⇄ 30

After : 5 ⇄ 10 ⇄ 20 ⇄ 30

Real-Life Example

- Adding a **new recent file** at the top

```
// Insert at beginning
void insertAtBeg() {
    Node* newnode = new Node();
    cout << "\nEnter data to insert at beginning: ";
    cin >> newnode->data;
    newnode->prev = nullptr;
    newnode->next = nullptr;
    if (head == nullptr) {
        head = newnode;
    } else {
        head->prev = newnode;
        newnode->next = head;
        head = newnode;
    }
}
```



Insertion in the Middle (At Position)

Steps:

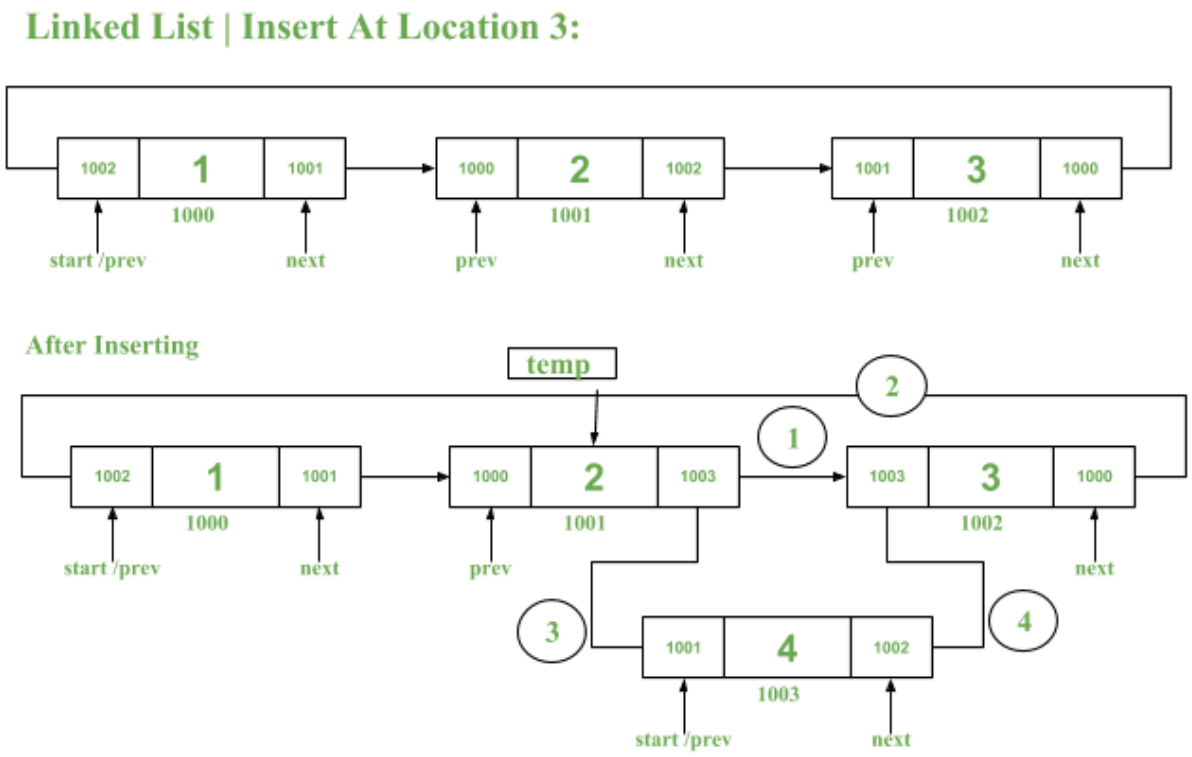
1. **Traverse to (position – 1)**
2. **Set links carefully:**
    - **New node next → current next**
    - **New node prev → current**
    - **Adjust surrounding nodes**

```
// Function to delete node from a given position
void deleteFromPos() {
  int pos, i = 1;
  Node* temp = head;
  if (head == nullptr) {
    cout << "\nList is empty\n";
    return;
  }
  cout << "\nEnter position to delete: ";
  cin >> pos;
```

```
    // Delete first node
    if (pos == 1) {
        head = head->next;
        if (head != nullptr)
            head->prev = nullptr;
        delete temp;
        cout << "\nNode deleted from position " << pos << endl;
        return;
    }
    // Traverse to the given position
    while (i < pos && temp != nullptr) {
        temp = temp->next;
        i++;
    }
    // Invalid position
    if (temp == nullptr) {
        cout << "\nInvalid position\n";
        return;
    }
    // Delete last node
    if (temp->next == nullptr) {
        temp->prev->next = nullptr;
        endNode = temp->prev;
    }
    // Delete middle node
    else {
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
```

```
}

  delete temp;
  cout << "\nNode deleted from position " << pos << endl;
}
```



Insertion at End (Tail)

Steps:

1. Create new node
2. Traverse to last node
3. Last node next → new node
4. New node prev → last node

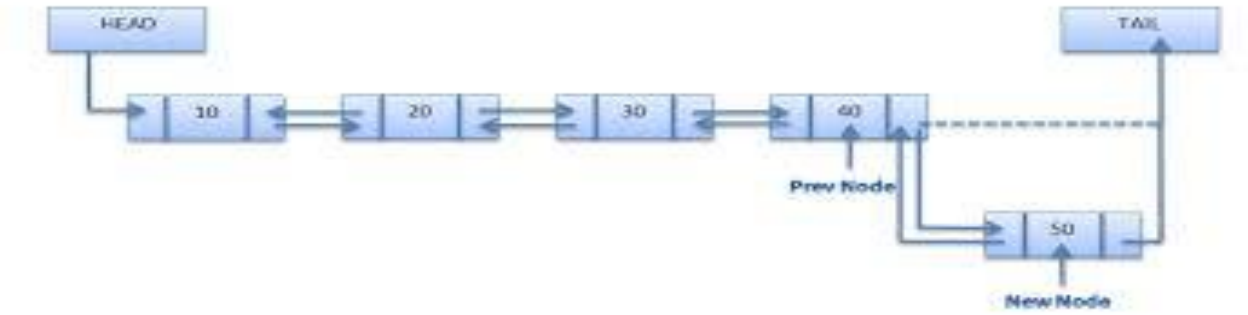
Before: 10 ⇄ 20 ⇄ 30

After : 10 ⇄ 20 ⇄ 30 ⇄ 40

Corporate Example

- Adding a **new employee** at the end of records

```
void insertAtEnd() {
    Node* newnode = new Node();
    cout << "\nEnter data to insert at end: ";
    cin >> newnode->data;
    newnode->next = nullptr;
    newnode->prev = nullptr;
    if (head == nullptr) {
        head = endNode = newnode;
    } else {
        endNode->next = newnode;
        newnode->prev = endNode;
        endNode = newnode;
    }
}
```



Deletion Functions in Doubly Linked List

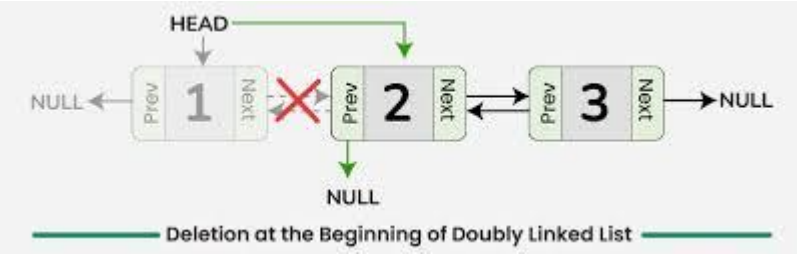
Deletion from Beginning

Steps:

1. Move head to second node
2. Set new head prev → NULL

Before: 10 ⇄ 20 ⇄ 30

After : 20 ⇄ 30



Real-Life Example

- Removing **oldest notification**

```
// Function to delete node from beginning
Node* deleteFromBeginning(Node* head)
{
    if (head == nullptr)
    {
        cout << "List is empty, nothing to delete." << endl;
        return head;
    }
    Node* temp = head;
    head = head->next;   // Move head to next node
    delete temp;         // Free old head
    return head;
}
```

Deletion from End

Steps:

1. Traverse to last node
2. Update second-last node next → NULL

Before: 10 ⇄ 20 ⇄ 30

After : 10 ⇄ 20

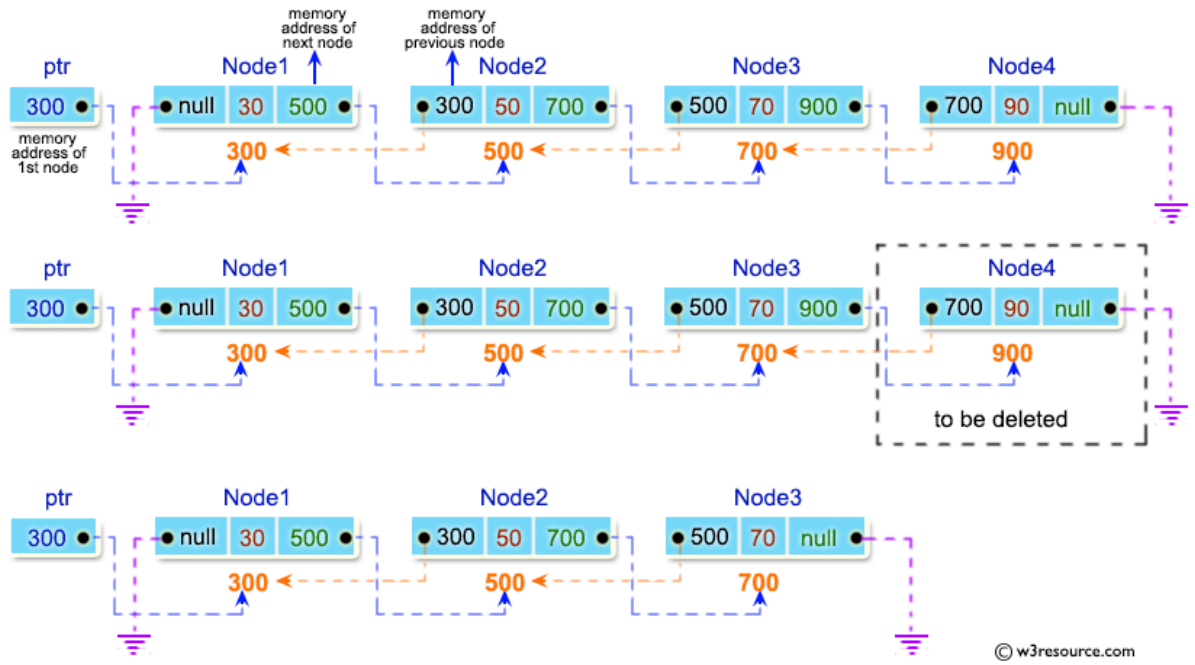
Corporate Example

- Removing resigned employee from database

```
// Function to delete node from end
Node* deleteFromEnd(Node* head)
{
    Node* prevnode = nullptr;
    Node* temp = head;

    if (head == nullptr)
    {
        cout << "List is empty, nothing to delete." << endl;
        return head;
    }
    while (temp->next != nullptr)
    {
        prevnode = temp;
        temp = temp->next;
    }
    // Only one node in the list
    if (temp == head)
    {
        head = nullptr;
        delete temp;
```

```
        }
        else
        {
            prevnode->next = nullptr;
            delete temp;
        }
        return head;
    }
```



Deletion by Value / Position

Steps:

1. Search the node
2. Update both prev and next links

Delete 20

10 ⇄ 20 ⇄ 30 → 10 ⇄ 30

Important Advantage

No need to track previous node separately (unlike singly list)

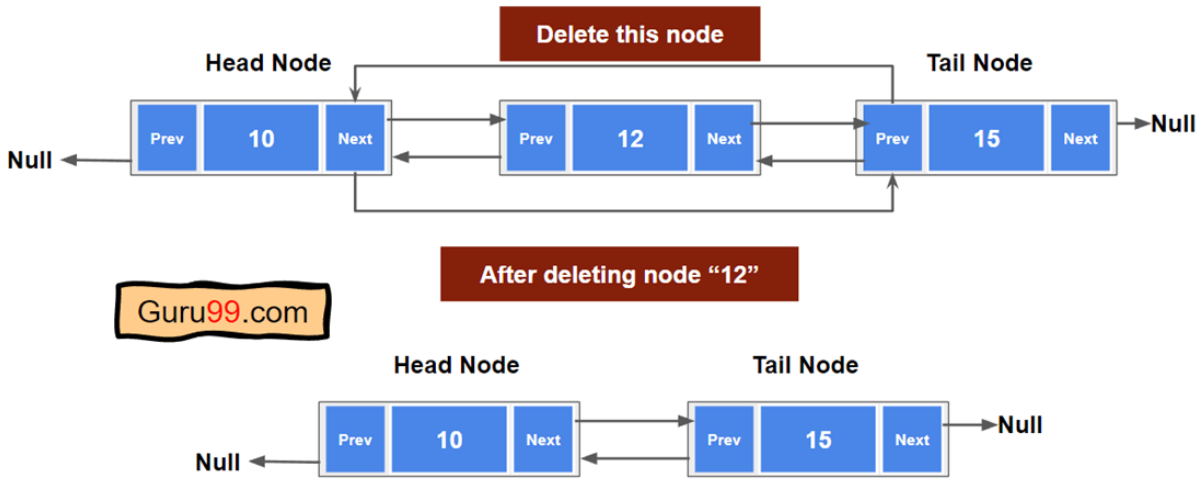
// Function to delete node from a given position

```
void deleteFromPos() {
  int pos, i = 1;
  Node* temp = head;
```

```
    if (head == nullptr) {
        cout << "\nList is empty\n";
        return;
    }
    cout << "\nEnter position to delete: ";
    cin >> pos;
    // Delete first node
    if (pos == 1) {
        head = head->next;
        if (head != nullptr)
            head->prev = nullptr;
        delete temp;
        cout << "\nNode deleted from position " << pos << endl;
        return;
    }
    // Traverse to the given position
    while (i < pos && temp != nullptr) {
        temp = temp->next;
        i++;
    }
    // Invalid position
    if (temp == nullptr) {
        cout << "\nInvalid position\n";
        return;
    }
    // Delete last node
    if (temp->next == nullptr) {
        temp->prev->next = nullptr;
```

```
    endNode = temp->prev;
  }
  // Delete middle node
  else {
    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;
  }

  delete temp;
  cout << "\nNode deleted from position " << pos << endl;
}
```



Interview questions

Q1. Why does a browser need a Doubly Linked List?

Answer:

A browser needs a doubly linked list because it supports **two-way navigation**.

- prev pointer → **Back** button
- next pointer → **Forward** button

Each webpage knows:

- which page came **before it**
- which page comes **after it**

If only a singly linked list was used, the browser could go **forward**, but going **back** would be difficult and slow because the previous page address is not directly available.

Real-life example:
Like flipping pages in a book where you can go **back and forth easily**.

Corporate example:
Enterprise dashboards where users move between previous and next reports.

Hence, browsers use Doubly Linked List.

Q2. Why is a Singly Linked List not enough for Undo/Redo operations?

Answer:
Undo/Redo requires movement in **both directions**:

- **Undo** → go to previous action
- **Redo** → go to next action

A singly linked list only stores the **next** address, not the previous one.

So:

- Undo is possible only if we **traverse again from head** (inefficient)
- Redo becomes very difficult

A **doubly linked list** solves this because:

- prev → Undo
- next → Redo

Real-life example:
Typing in MS Word:

- Undo → go back to previous edit
- Redo → move forward to next edit

Corporate example:
Version control systems where changes move backward and forward.