Doubly Linked List

Course: Introduction to Programming and Data Structures

Dr. Laltu Sardar

Institute for Advancing Intelligence (IAI),
TCG Centres for Research and Education in Science and Technology (TCG Crest)

Doubly Linked List

Introduction to Doubly Linked Lists

Definition:

- A Doubly Linked List is a type of linked list where each node contains a data field and two pointers.
- One pointer points to the next node, and the other points to the previous node.

Example:

$$\text{head} \rightarrow \boxed{\text{Node1}} \leftrightarrow \boxed{\text{Node2}} \leftrightarrow \boxed{\text{Node3}} \rightarrow \text{NULL}$$

Node Structure

A node in a doubly linked list typically contains:

- **Data**: The value or data stored in the node.
- **Next Pointer:** A reference to the next node in the list.
- **Prev Pointer:** A reference to the previous node in the list.

Structure:

$$\text{Node} = \boxed{\text{Prev} \mid\mid \text{Data} \mid\mid \text{Next}}$$

Advantages of Doubly Linked Lists

- **Bi-directional Traversal:** Can be traversed in both forward and backward directions.
- **Efficient Deletion:** Allows quick deletion of a node when you have a reference to it.
- **Previous Node Access:** Can easily access the previous node without needing to traverse from the head.

Basic Operations

The common operations on a doubly linked list are:

- **Insertion:** Insert a new node at the beginning, end, or a specific position.
- **Deletion:** Remove a node from the list.
- **Traversal:** Traverse through the list from the head to the tail or from the tail to the head.

Insertion in a Doubly Linked List

Insertion at the beginning:

- Create a new node.
- Set the new node's next pointer to the current head.
- Set the current head's previous pointer to the new node.
- Set the new node as the new head.

Example: Initial List:

$$\text{head} \rightarrow \boxed{\text{Node1}} \leftrightarrow \boxed{\text{Node2}} \leftrightarrow \boxed{\text{Node3}} \rightarrow \text{NULL}$$

Inserting Node0 at the start:

$$\text{head} \rightarrow \boxed{\texttt{Node0}} \leftrightarrow \boxed{\text{Node1}} \leftrightarrow \boxed{\text{Node2}} \leftrightarrow \boxed{\text{Node3}} \rightarrow \text{NULL}$$

Example Code

```c
// Function to insert a node at the beginning of the doubly linked list
void insertAtBeginning(struct Node** head_ptr, int data) {
    // Create the new node
    struct Node* newNode = create_node(data);

    // Make the next of new node as the head and previous as NULL
    newNode->next = *head_ptr;
    newNode->prev = NULL;

    // Change the previous of the head node to the new node (if head exists)
    if (*head_ptr != NULL) {
        (*head_ptr)->prev = newNode;
    }

    // Move the head to point to the new node
    *head_ptr = newNode;
}
```

Deletion in a Doubly Linked List

Deleting a node:

- Adjust the previous node's next pointer to point to the current node's next.
- Adjust the next node's previous pointer to point to the current node's previous.

Example: Example: Initial List:

$$\text{head} \rightarrow \boxed{\text{Node1}} \leftrightarrow \boxed{\text{Node2}} \leftrightarrow \boxed{\text{Node3}} \rightarrow \text{NULL}$$

Deleting Node2 from the list

$$\text{head} \rightarrow \boxed{\texttt{Node0}} \leftrightarrow \boxed{\text{Node1}} \leftrightarrow \boxed{\text{Node3}} \rightarrow \text{NULL}$$

Example Code I

```c
// Function to delete a node with a given value
void deleteNode(struct Node** head, int key) {
    struct Node* temp = *head;

    // If the list is empty
    if (*head == NULL) {
        printf("List is empty, no node to delete.\n");
        return;
    }

    // Traverse the list to find the node with the given key
    while (temp != NULL && temp->data != key) {
        temp = temp->next;
    }

    // If the node with the given key is not found
    if (temp == NULL) {
        printf("Node with value %d not found.\n", key);
        return;
    }
```

Example Code II

```c

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

    // If the node to be deleted is not the last node
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }

    // If the node to be deleted is not the first node
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    // Free the memory of the node to be deleted
    free(temp);
    printf("Node with value %d deleted successfully.\n", key);
}
```

Traversal in a Doubly Linked List

Forward Traversal:

- Start at the head and move through the list using the next pointer.

Backward Traversal:

- Start at the tail and move through the list using the previous pointer.

Example:

- Forward: head → Node1 → Node2 → Node3
- Backward: Node3 → Node2 → Node1

Applications of Doubly Linked Lists

- **Navigation Systems:** Back and forward operations in browsers, text editors, etc.
- **Undo-Redo Functionality:** Implement undo-redo features in applications.
- **Music/Video Playlist:** Can navigate to previous or next media file easily.

Comparison with Singly Linked Lists

- **Singly Linked List:** Only allows traversal in one direction (forward).
- **Doubly Linked List:** Allows traversal in both directions (forward and backward).
- **Memory Overhead:** Doubly linked lists require more memory to store two pointers (next and previous).

Summary

Doubly Linked Lists:

- Efficient for bi-directional traversal.
- More flexible for insertion and deletion operations.
- Suitable for applications that require frequent back-and-forth navigation.

Important Operations on Doubly Linked Lists I

- **Insertion**:
  - **At the Beginning**: Inserting a new node at the start of the list.
  - **At the End**: Inserting a new node at the end of the list.
  - **At a Specific Position**: Inserting a new node before or after a given node.
- **Deletion**:
  - **From the Beginning**: Removing the first node of the list.
  - **From the End**: Removing the last node of the list.
  - **From a Specific Position**: Removing a node located at a specific position in the list.
- **Traversal**:
  - **Forward Traversal**: Accessing each node of the list from the head to the tail.

Important Operations on Doubly Linked Lists II

  - **Backward Traversal**: Accessing each node of the list from the tail to the head.
- **Search**:
  - **Search by Value**: Finding the first node containing a specific value.
  - **Search by Position**: Accessing the node at a particular index in the list.
- **Updating**:
  - Modifying the data stored in a specific node without altering the structure of the list.
- **List Reversal**:
  - Reversing the order of nodes in the list so that the first node becomes the last and vice versa.

Important Operations on Doubly Linked Lists III

- **Splitting**:
  - Dividing the list into two smaller lists at a given position.
- **Concatenation**:
  - Merging two doubly linked lists into a single list.
- **Length Calculation**:
  - Counting the number of nodes present in the list.

tcg crest
Inventing Harmonious Future

Dr. Laltu Sardar
laltu.sardar@tcgcrest.org
https://laltu-sardar.github.io.