COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY

SCHOOL OF COMPUTING
DEPARTMENT OF NETWORKING AND COMMUNICATIONS

21SCS201J DATA STRUCTURES AND ALGORITHMS
PRESENTATION ON REAL WORLD APPLICATIONS

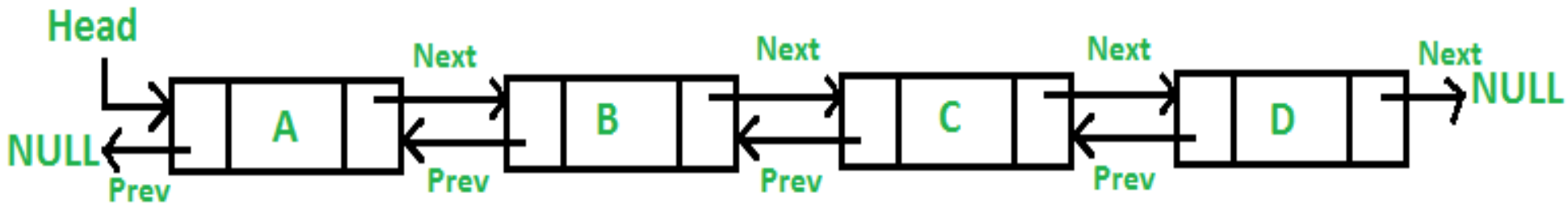
Doubly linked list

RA221108010030-TIKAM
RA2211028010026-UJJWAL
RA221108010063-LAMIH

Doubly linked list

Introduction

A doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields references to the previous and to the next node in the sequence of nodes and one data field. The beginning and ending nodes' previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.


Head
NULL
Prev
A
Next
Prev
B
Next
Prev
C
Next
Prev
D
Next
NULL

Key Characteristics

- Bidirectional Traversal: Doubly linked lists allow for traversal in both forward and backward directions. This feature is especially useful for operations that require accessing elements in reverse order or for efficient navigation in both directions.

- Efficient Insertion and Deletion: Doubly linked lists provide efficient methods for inserting and deleting elements at various positions within the list. This makes them suitable for scenarios where data is frequently modified.

- Doubly Linked Nodes: Each node in a doubly linked list contains data and two pointers: one pointing to the previous node (prev) and one pointing to the next node (next). This bidirectional structure enables easy navigation and manipulation.

- Circular or Linear: Doubly linked lists can be implemented as circular (where the last node points to the first node) or linear (with a distinct beginning and end). The choice of implementation depends on the specific application requirements.

- Memory Overhead: Unlike singly linked lists, doubly linked lists require more memory due to the additional prev pointers. This should be considered when choosing a data structure, especially for large datasets.

- Versatility: Doubly linked lists are versatile and can be used in various scenarios, such as managing a history of actions, implementing undo/redo functionality, and more.

- Real-World Applications: Doubly linked lists are found in various software applications, including text editors (for undo/redo), browsers (for navigation history), and more. Understanding their characteristics is crucial for building efficient systems.

- Complexity Analysis: Understanding the time complexity of common operations like insertion, deletion, and traversal is vital for optimizing the use of doubly linked lists in algorithms and data processing.

Creating a Doubly Linked List

```
#include <stdio.h>
#include <stdlib.h>

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

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
```

```
void insertEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
    }
}
void displayList(struct Node* head) {
    printf("Doubly Linked List: ");
    while (head != NULL) {
        printf("%d <-> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
```

```
int main() {
    struct Node* head = NULL;
    int data, numNodes;

    printf("Enter the number of nodes: ");
    scanf("%d", &numNodes);

    for (int i = 0; i < numNodes; i++) {
        printf("Enter data for node %d: ", i + 1);
        scanf("%d", &data);
        insertEnd(&head, data);
    }
    displayList(head);

    while (head != NULL) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
    }
    return 0;
}
```

Output

main.c

Run

```
40          head = head->next;
41      }
42      printf("NULL\n");
43  }
44
45  int main() {
46      struct Node* head = NULL;
47      int data, numNodes;
48
49      printf("Enter the number of nodes: ");
50      scanf("%d", &numNodes);
51
52      for (int i = 0; i < numNodes; i++) {
53          printf("Enter data for node %d: ", i + 1);
54          scanf("%d", &data);
55          insertEnd(&head, data);
56      }
57
58      displayList(head);
59
60      int newData;
61      printf("Enter data for the new node to insert at the end: ");
62      scanf("%d", &newData);
63
64      insertEnd(&head, newData);
65      displayList(head);
66
67      // Free allocated memory
68      while (head != NULL) {
69          struct Node* temp = head;
70          head = head->next;
71          free(temp);
72      }
73
74      return 0;
75  }
76
```

Output

```
/tmp/E7rpQeMefb.o
Enter the number of nodes: 4
Enter data for node 1: 22
Enter data for node 2: 21
Enter data for node 3: 23
Enter data for node 4: 24
Doubly Linked List: 22 <-> 21 <-> 23 <-> 24 <-> NULL
```

Adding and Removing Elements

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

struct Node* head = NULL;

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}
```

```
// Function to insert a new node at the end of the list
void insertAtEnd(int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* current = head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
    }
```

```
// Function to remove a node with a given value
void deleteNode(int data) {
    struct Node* current = head;
    while (current != NULL) {
        if (current->data == data) {
            if (current->prev != NULL) {
                current->prev->next = current->next;
            } else {
                head = current->next;
            }
            if (current->next != NULL) {
                current->next->prev = current->prev;
            }
            free(current);
            return;
        }
        current = current->next;
    }
}
```

```
// Function to display the linked list
void display() {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d <-> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    int choice, data;
    printf("\nDoubly Linked List Menu:\n");
        printf("1. Add an element\n");
        printf("2. Remove an element\n");
        printf("3. Display the list\n");
        printf("4. Exit\n");
```

```
while (1) {
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the element to add: ");
                scanf("%d", &data);
                insertAtEnd(data);
                break;
            case 2:
                printf("Enter the element to remove: ");
                scanf("%d", &data);
                deleteNode(data);
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
    return 0;
}
```

main.c

Run

Output

```c
#include <stdio.h>
#include <stdlib.h>

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

struct Node* head = NULL;

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}

// Function to insert a new node at the end of the list
void insertAtEnd(int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* current = head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
    }
}

// Function to remove a node with a given value
void deleteNode(int data) {
```

```
/tmp/E7rpQeMefb.o
Doubly Linked List Menu:
1. Add an element
2. Remove an element
3. Display the list
4. Exit
Enter your choice: 1
Enter the element to add: 22
Enter your choice: 2
Enter the element to remove: 22
Enter your choice: 3
NULL
Enter your choice: 4
```

Traversing the List Forward and Backward

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
struct Node* head = NULL;

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}
```

```
// Function to insert a new node at the end of the list
void insertAtEnd(int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* current = head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
    }
```

```
// Function to display the list forward
void displayForward() {
    struct Node* current = head;
    printf("Forward Traversal: ");
    while (current != NULL) {
        printf("%d <-> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
// Function to display the list backward
void displayBackward() {
    struct Node* current = head;
    if (current == NULL) {
        printf("Backward Traversal: NULL\n");
        return;
    }
    while (current->next != NULL) {
        current = current->next;
    }
```

```
printf("Backward Traversal: ");
    while (current != NULL) {
        printf("%d <-> ", current->data);
        current = current->prev;
    }
    printf("NULL\n");
}
int main() {
    int choice, data;
    printf("\nDoubly Linked List Menu:\n");
        printf("1. Add an element\n");
        printf("2. Display forward\n");
        printf("3. Display backward\n");
        printf("4. Exit\n");

    while (1) {
```

```
printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element to add: ");
                scanf("%d", &data);
                insertAtEnd(data);
                break;
            case 2:
                displayForward();
                break;
            case 3:
                displayBackward();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

main.c

Run

Output

```
60      while (current != NULL) {
61          printf("%d <-> ", current->data);
62          current = current->prev;
63      }
64      printf("NULL\n");
65  }
66
67  int main() {
68      int choice, data;
69      printf("\nDoubly Linked List Menu:\n");
70          printf("1. Add an element\n");
71          printf("2. Display forward\n");
72          printf("3. Display backward\n");
73          printf("4. Exit\n");
74
75      while (1) {
76
77          printf("Enter your choice: ");
78          scanf("%d", &choice);
79
80          switch (choice) {
81              case 1:
82                  printf("Enter the element to add: ");
83                  scanf("%d", &data);
84                  insertAtEnd(data);
85                  break;
86              case 2:
87                  displayForward();
88                  break;
89              case 3:
90                  displayBackward();
91                  break;
92              case 4:
93                  exit(0);
94              default:
95                  printf("Invalid choice. Please try again.\n");
96          }
```

```
/tmp/E7rpQeMefb.o
Doubly Linked List Menu:
1. Add an element
2. Display forward
3. Display backward
4. Exit
Enter your choice: 1
Enter the element to add: 22
Enter your choice: 1
Enter the element to add: 39
Enter your choice: 2
Forward Traversal: 22 <-> 39 <-> NULL
Enter your choice: 3
Backward Traversal: 39 <-> 22 <-> NULL
Enter your choice: 4
```

Advantages of Doubly Linked Lists

- Bidirectional Traversal: Unlike singly linked lists, which only allow traversal in one direction (from head to tail), doubly linked lists allow for easy traversal in both directions. This bidirectional traversal is beneficial in various situations.

- Insertion and Deletion: Insertions and deletions are more efficient compared to singly linked lists for some scenarios. In a singly linked list, when you want to delete a node, you need to traverse from the head to find the node and its previous node. In a doubly linked list, you can directly access the previous node and perform the deletion, making the operation more efficient.

- Reversing: Reversing a singly linked list requires additional memory and is a relatively complex operation. In a doubly linked list, reversing the list is straightforward, as you can swap the next and prev pointers for each node.

- Dynamic Memory Allocation: Doubly linked lists are well-suited for managing dynamic memory. They can be efficiently used to implement memory allocation and deallocation mechanisms in programming languages or memory management systems.

- Tail Operations: In a singly linked list, appending to the end of the list requires traversing the entire list. In a doubly linked list, you can directly access the last node, which makes append operations more efficient.

Disadvantages of Doubly Linked Lists

- Memory Overhead: Doubly linked lists require more memory compared to singly linked lists because they store both next and prev pointers for each node. This additional memory overhead can be a concern when working with large data sets.

- Complexity: Managing doubly linked lists can be more complex than singly linked lists. This complexity arises from the need to maintain and update both next and prev pointers when inserting or deleting nodes. This makes the code more error-prone and harder to maintain.

- Slower Insertions and Deletions: Although doubly linked lists can offer advantages in some situations, such as efficient deletion of a node when you have a reference to it, insertions and deletions can be slower than singly linked lists in cases where you don't need the extra pointers. This is due to the extra operations required to maintain both next and prev pointers.

- Increased Code Complexity: Managing doubly linked lists typically involves more complex code compared to singly linked lists. This can lead to more opportunities for bugs and maintenance challenges.

- Additional Storage Requirements: In some memory-constrained applications, the extra storage required for the prev pointers in doubly linked lists can be a significant drawback.

Thank you