K.R. MANGALAM UNIVERSITY

THE COMPLETE WORLD OF EDUCATION

Data Structure

ENCS205

School of Engineering & Technology (SOET)
K.R. MANGALAM University

UNIT-2

Session: 19 DOUBLY LINKED LIST(DLL)

Recap

Definition: Singly linked list is a linear data structure with nodes containing data and pointers.

Components: Nodes consist of data and a next pointer; head pointer points to the first node; tail pointer is optional.

Representation: Nodes linked via pointers; dynamic memory allocation; access via head pointer, traversal for manipulation.

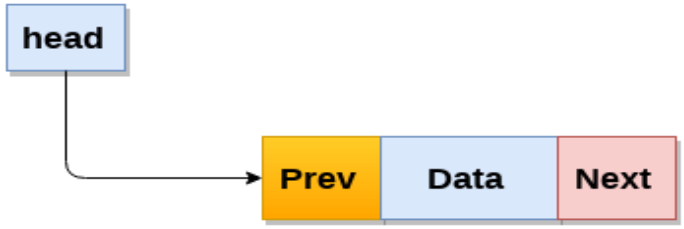
Advantages: Efficient insertion/deletion; dynamic size, flexibility.

Disadvantages: No direct access, traversal needed; higher memory overhead; inefficient reverse traversal.

Sessions 19 Outlook

- Doubly linked list
- Inserting and Deleting a Node
- Memory Representation of a doubly linked list
- Practice Questions

Doubly linked list

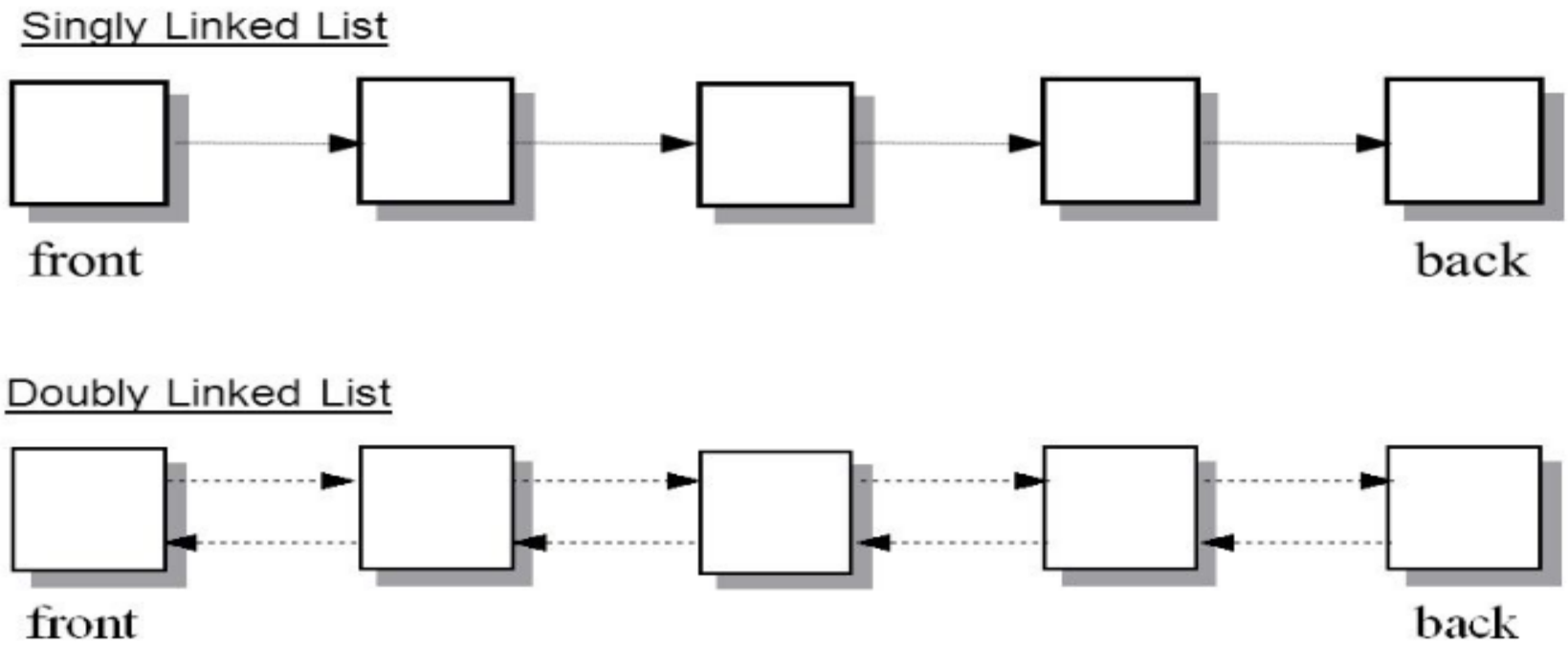


Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.

Doubly Linked List

Singly Linked List vs Doubly Linked List



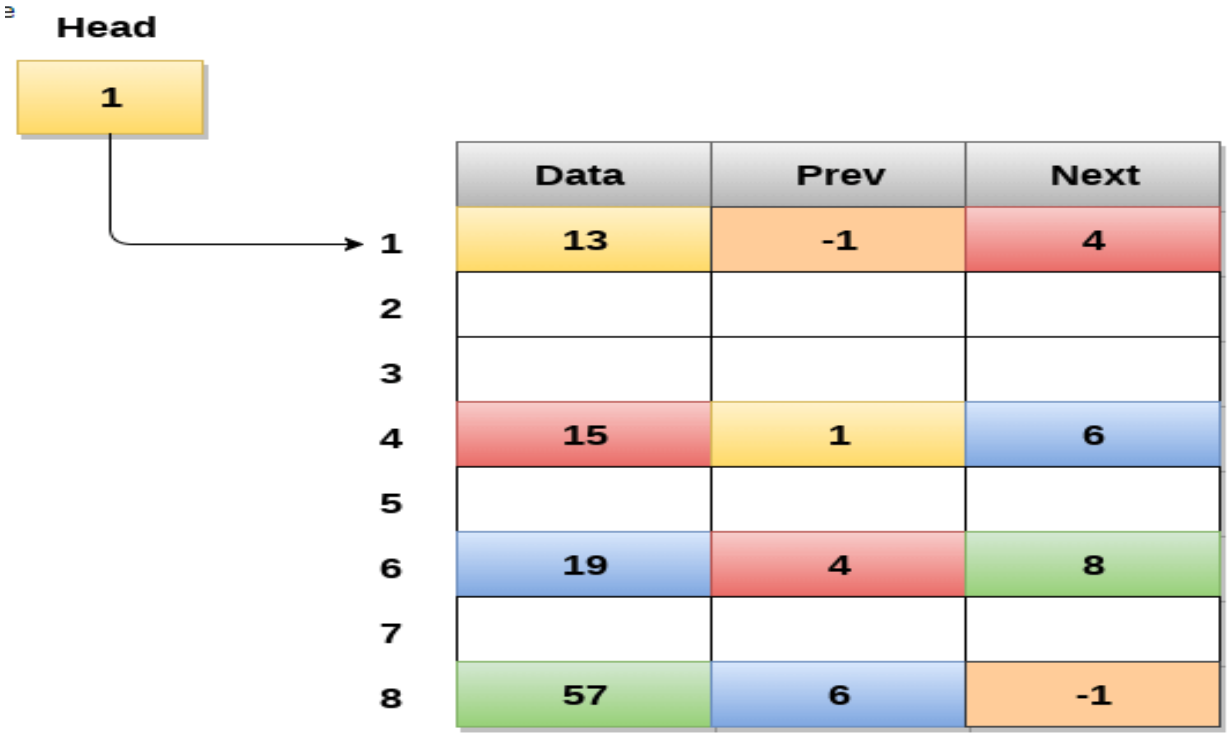
Singly Linked List vs Doubly Linked List

Aspect	Singly Linked List	Doubly Linked List
Node Structure	Data, Next Pointer	Data, Next Pointer, Previous Pointer
Traversal Direction	Forward Only	Forward and Backward
Memory Efficiency	Less memory per node (single pointer)	More memory per node (two pointers)
Insertion/Deletion	Efficient at the beginning; less efficient elsewhere	Efficient at any position (constant time)
Tail Pointer	Often lacks tail pointer	Typically includes tail pointer for efficient insertion/deletion at both ends

Doubly linked list

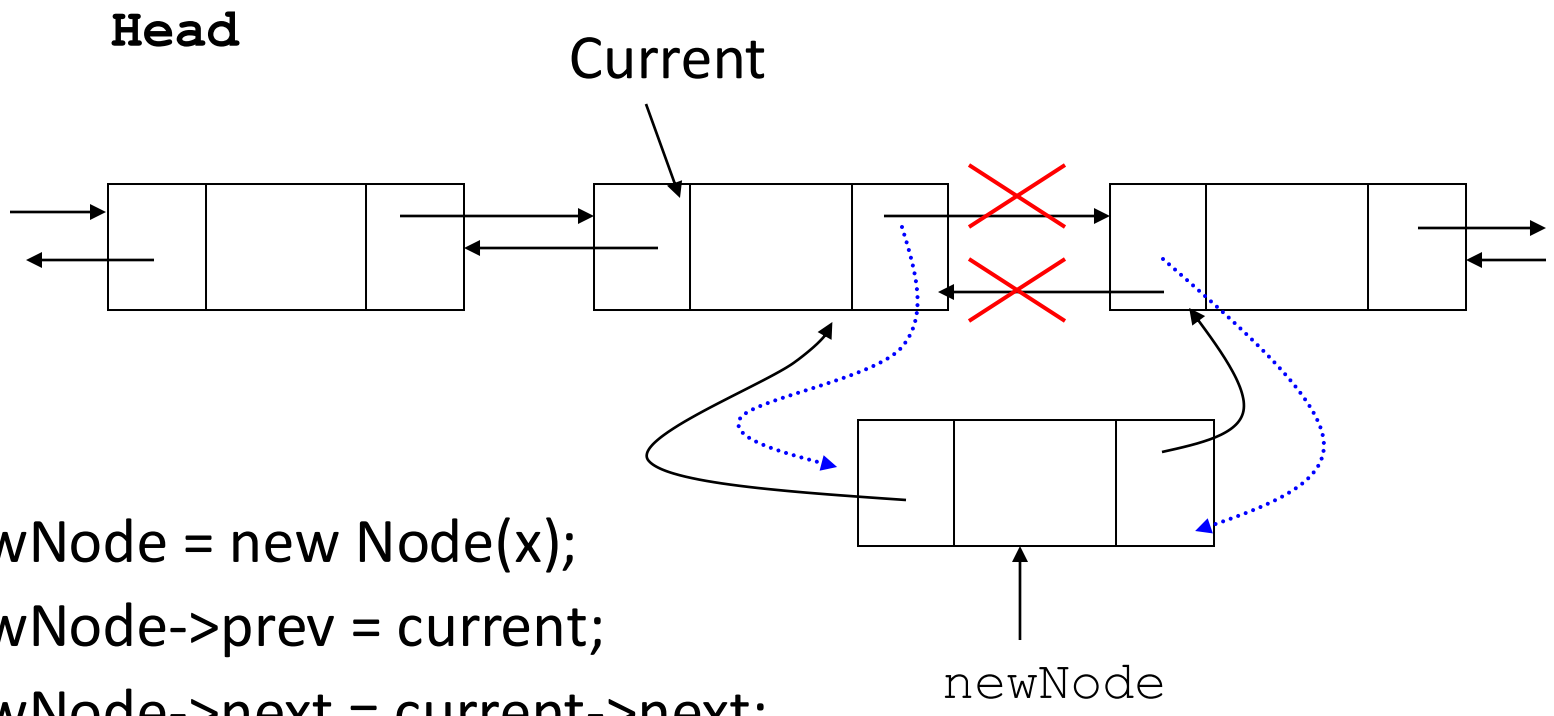
```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
}
```

Memory Representation of a doubly linked list



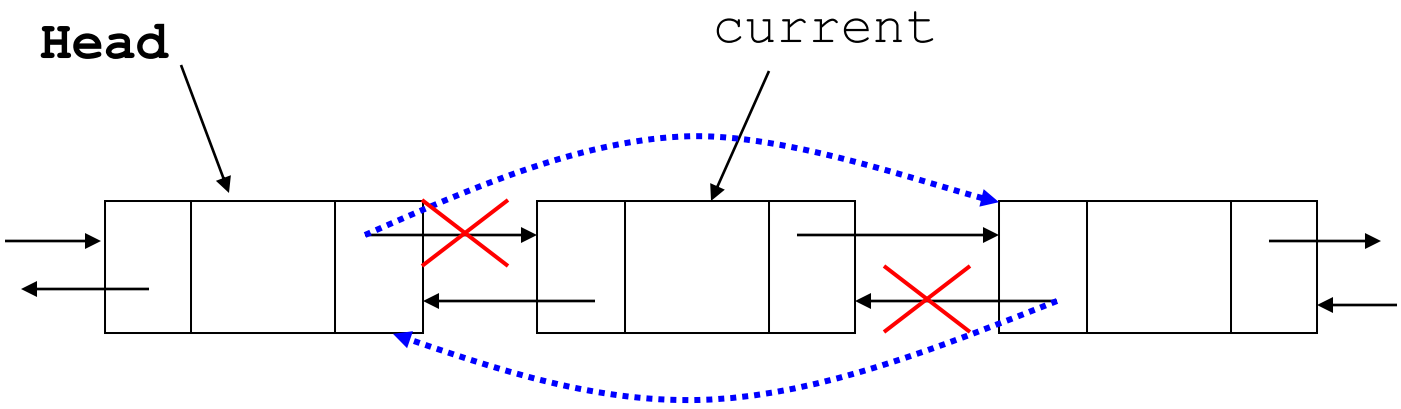
Memory Representation of a Doubly linked list

Insertion




```
newNode = new Node(x);
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Deletion



oldNode = current;

oldNode->prev->next = oldNode->next;

oldNode->next->prev = oldNode->prev;

delete oldNode;

current = head;

Insertion and deletion using doubly linked list

```
/ Function to insert a new node.
void Linkedlist::insertNode(int data)
{
   // Create the new Node.
   Node* newNode = new Node(data);
   // Assign to head
   if (head == NULL) {
      head = newNode;
      return;
   }

   // Traverse till end of list
   Node* temp = head;
   while (temp->next != NULL) {
```

Complete code here

Operations

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at the beginning.
2	Insertion at end	Adding the node into the linked list at the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from the beginning of the list.
5	Deletion at the end	Removing the node from the end of the list.
6	Deletion of the node with data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and returning its location.
8	Traversing	Visiting each node of the list for operations like searching, sorting, display, etc.

Lets try

1. Which of the following is false about a doubly linked list?

a) We can navigate in both the directions

b) It requires more space than a singly linked list

c) The insertion and deletion of a node take a bit longer

d) Implementing a doubly linked list is easier than singly linked list

Answer: d
Explanation: A doubly linked list has two pointers ‘left’ and ‘right’ which enable it to traverse in either direction. Compared to singly liked list which has only a ‘next’ pointer, doubly linked list requires extra space to store this extra pointer. Every insertion and deletion requires manipulation of two pointers, hence it takes a bit longer time. Implementing doubly linked list involves setting both left and right pointers to correct nodes and takes more time than singly linked list.

2. What is a memory efficient double linked list?

a) Each node has only one pointer to traverse the list back and forth

b) The list has breakpoints for faster traversal

c) An auxiliary singly linked list acts as a helper list to traverse through the doubly linked list

d) A doubly linked list that uses bitwise AND operator for storing addresses

Answer: a

Explanation: Memory efficient doubly linked list has only one pointer to traverse the list back and forth. The implementation is based on pointer difference. It uses bitwise XOR operator to store the front and rear pointer addresses. Instead of storing actual memory address, every node store the XOR address of previous and next nodes.

3. How do you calculate the pointer difference in a memory efficient double linked list?

a) head xor tail

b) pointer to previous node xor pointer to next node

c) pointer to previous node – pointer to next node

d) pointer to next node – pointer to previous node

Answer: b

Explanation: The pointer difference is calculated by taking XOR of pointer to previous node and pointer to the next node.

4. What is the worst case time complexity of inserting a node in a doubly linked list?

a) O(nlogn)

b) O(logn)

c) O(n)

d) O(1)

Answer: c

Explanation: In the worst case, the position to be inserted maybe at the end of the list, hence you have to traverse through the entire list to get to the correct position, hence O(n).

5. What are the advantages of a doubly linked list ?

a: The memory required for a doubly linked list is less than a linked list

b: Doubly linked list allows for traversing both sides.

c: Inserting an element is faster in a doubly linked list.

d: There is no advantage.

6. What is the time complexity for insertion of an element into a doubly linked list?

a: O(1)

b: O(log n)

c: O(n)

d: O(n^2)

Review

- Each node in a doubly linked list contains references to both the next and previous nodes.
- Bidirectional traversal is possible, allowing navigation in both forward and backward directions.
- Nodes consist of data and pointers to the next and previous nodes.
- Efficient deletion operations, especially for nodes not located at the beginning of the list.

Review

- Commonly used in applications requiring bidirectional navigation, such as text editors and browser history management.
- Higher memory consumption compared to singly linked lists due to additional pointers stored in each node.


THANK
YOU