Data Structures

Dilip Kumar Maripuri
Computer Applications

Data Structures

Session : Linked Lists - Doubly Linked Lists.

Dilip Kumar Maripuri
Computer Applications

- Doubly linked lists are a sophisticated form of linked lists in which each node contains three components
  - Data
  - A reference to the next node in the list
  - A reference to the previous node.
- This structure allows traversal of the list in both directions, offering greater flexibility compared to singly linked lists.

- **Node Composition**
  - Data: Each node contains a data field to store the value.
  - Next Pointer: A reference or link to the next node in the list.
  - Previous Pointer: A reference or link to the previous node in the list.
- **End Nodes**
  - The previous pointer of the first node and the next pointer of the last node point to null, signaling the end of the list.

Doubly Linked Lists - Characteristics

- **Two-Way Navigation**
  - Unlike singly linked lists, doubly linked lists allow traversal in both forward and backward directions.
  - This is enabled by the presence of two pointers in each node, pointing to the next and the previous nodes.
- **Dynamic Size**
  - The size of the list can increase or decrease dynamically, as nodes can be added or removed from any part of the list.

Doubly Linked Lists - Characteristics

- **No Need for Backtracking**
  - When traversing a singly linked list, sometimes there's a need to keep track of the previous node (for operations like deletion).
  - In doubly linked lists, this is not necessary as each node inherently has a reference to its predecessor.
- **Memory Storage**
  - In memory, the nodes of a doubly linked list are not stored in contiguous memory locations.
  - Each node can exist independently in memory, and the pointers are used to navigate through the list.

Doubly Linked Lists - Characteristics

- **Memory Overhead**
  - Each node in a doubly linked list requires extra memory for one additional pointer compared to singly linked lists.
  - This means they are more memory-intensive.
- **Implementation Details**
  - Implementing a doubly linked list typically involves careful handling of the pointers during insertion and deletion to avoid memory leaks or dangling pointers.
  - Edge cases, such as inserting or deleting at the beginning or end of the list, require special attention to correctly manage the head and tail pointers.

Pros of Using DLL

- **Bidirectional Traversal**: Allows moving forward and backward through the list, increasing flexibility.
- **Ease of Deletion**: Easier to delete a node, as it's straightforward to access the previous node.
- **Dynamic**: Can grow and shrink in size dynamically, using memory efficiently.
- **Insertion and Deletion:** More efficient than singly linked lists in certain scenarios, particularly near the ends of the list.

Cons of Using DLL

- **Memory Overhead**: Each node requires extra memory for an additional pointer (compared to singly linked lists).
- **Complex Implementation**: More complex to implement than singly linked lists due to the handling of two pointers.
- **Increased Processing Time**: Additional processing is required to update two pointers during insertions and deletions.
- **Risk of Memory Leak**: Incorrect handling of pointers during insertion or deletion can lead to memory leaks or dangling pointers.

Head only Vs Head,Tail

Aspect	Single Pointer (Head Only)	Two Pointers (Head and Tail)
Insertion at Tail	$O(n)$ – Must traverse entire list to find tail	$O(1)$ – Tail pointer allows direct access to the end
Deletion at Tail	$O(n)$ – Traversal needed to reach the end	$O(n)$ – Still requires traversal to find previous node
Insertion at Head	$O(1)$ – Directly add node after Head	$O(1)$ – Same as Head only
Space Complexity	Lower space overhead as only one pointer is stored	Slightly higher overhead due to extra Tail pointer
Use Case	Simpler to implement, good for operations at the head	Efficient for operations that involve both ends of the list

Table: Comparison of Single Pointer (Head Only) vs. Two Pointers (Head and Tail)

Aspect	Single Pointer (Head Only)	Two Pointers (Head and Tail)
Insertion at Head	$O(1)$ – Directly add node after Head	$O(1)$ – Same as Head only
Insertion at Tail	$O(n)$ – Must traverse entire list	$O(1)$ – Tail pointer allows direct access to the end
Deletion at Head	$O(1)$ – Directly remove node	$O(1)$ – Same as Head only
Deletion at Tail	$O(n)$ – Traversal needed to reach the last node	$O(1)$ – Direct access to the end with Tail

Table: Pros and Cons of Using Two Pointers (Head and Tail) in Doubly Linked Lists

Aspect	Single Pointer (Head Only)	Two Pointers (Head and Tail)
Bidirectional Traversal	$O(n)$ in forward direction, not possible backward	$O(n)$ in both directions, enabled by previous pointers
Space Complexity	Lower space as each node only stores one pointer	Higher space overhead for storing two pointers per node
Use Case	Good for applications that mostly access the head or need one-way traversal	Useful for cases needing frequent access or modification at both ends

Table: Pros and Cons of Using Two Pointers (Head and Tail) in Doubly Linked Lists

Do We Really Need Two Pointers in DLL

- **Time Complexity Optimization**:
  - For applications where the list is traversed in both directions, doubly linked lists with Head and Tail pointers offer optimal performance, allowing $O(1)$ access to both ends and efficient bidirectional traversal.

- While doubly linked lists inherently require more memory per node due to the two pointers, adding a Tail pointer at the list level provides significant performance benefits in terms of time complexity for operations involving the end of the list.

- This additional pointer is generally worth the minor increase in space complexity, particularly in scenarios with frequent modifications at both ends of the list.

```c
typedef struct node {
    int data;
    struct node *lptr, *rptr; } *NODE;

NODE createNode(int value) {
    NODE temp = (NODE) malloc (sizeof(NODE));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        return NULL;
    }
    newNode->data = value;
    newNode->lptr = newNode->rptr = NULL;
    return newNode;
}
```

- **Insert a Node at the Front in a Doubly Linked List**

Algorithm Insert_Front(Head, data):

Create new_node $\leftarrow$ createNode(data)
if new_node $\neq$ NULL **then**
    Set new_node$\rightarrow$rptr $\leftarrow$ Head
    Set new_node$\rightarrow$lptr $\leftarrow$ NULL
    if Head $\neq$ NULL **then**
        Set Head$\rightarrow$lptr $\leftarrow$ new_node
    end if
    Set Head $\leftarrow$ new_node
end if
Return Head

End Algorithm

Doubly Linked Lists

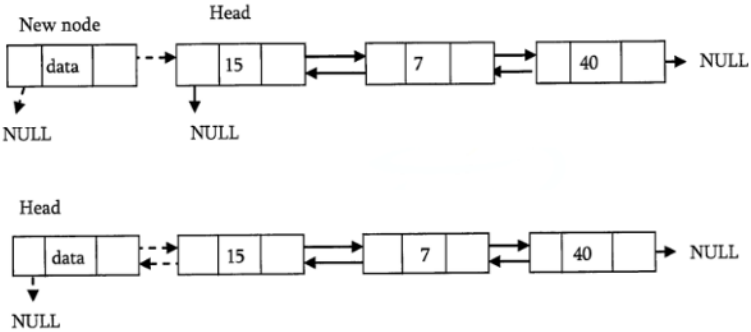


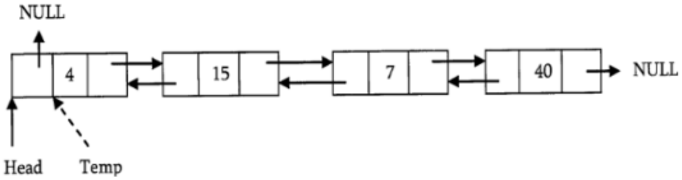
Figure: Insert a Node at Beginning

- **Delete the First Node in a Doubly Linked List**

Algorithm Delete_First(Head):

```
if Head = NULL then
    Print "List is empty. No node to delete."
    Return NULL
end if
Set temp ← Head
Set Head ← Head→ rptr
if Head ≠ NULL then
    Set Head→ lptr ← NULL
end if
Free temp
Return Head
```

End Algorithm



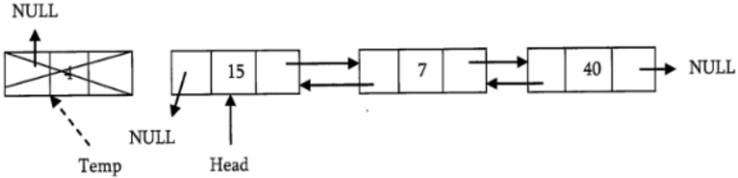


Figure: Delete a Node at Beginning

- **Insert a Node at the End in a Doubly Linked List**

Algorithm Insert_End(Head, data):

```
Create new_node ← createNode(data)
if new_node = NULL then
    Print "Memory allocation failed."; Return Head
end if
if Head = NULL then
    Return new_node
end if
Set temp ← Head
while temp→rptr ≠ NULL do
    Set temp ← temp→rptr
end while
Set temp→rptr ← new_node
Set new_node→lptr ← temp
Return Head
```

End Algorithm

Doubly Linked Lists

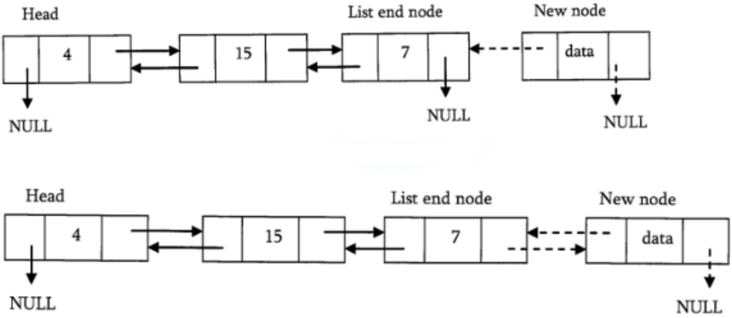


Figure: Insert a Node at the end

- **Delete the Last Node in a Doubly Linked List**

Algorithm Delete_Last(Head):

```
if Head = NULL then
    Print "List is empty. No node to delete.";        Return NULL
end if
if Head→rptr = NULL then
    Free Head;        Return NULL
end if
Set temp ← Head
while temp→rptr→rptr ≠ NULL do
    Set temp ← temp→rptr
end while
Set last ← temp→rptr
Set temp→rptr ← NULL
Free last
Return Head
```

End Algorithm

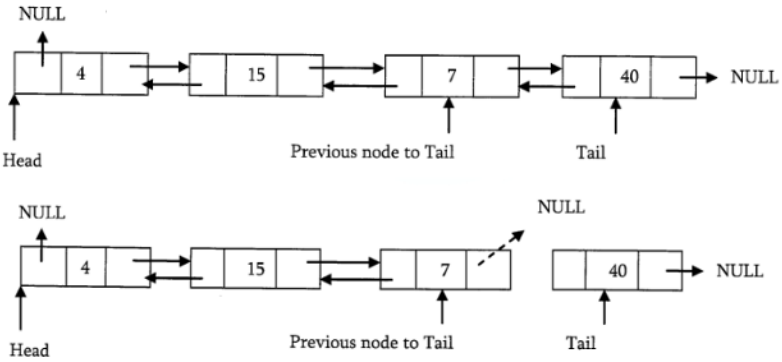


Figure: Delete a Node at End

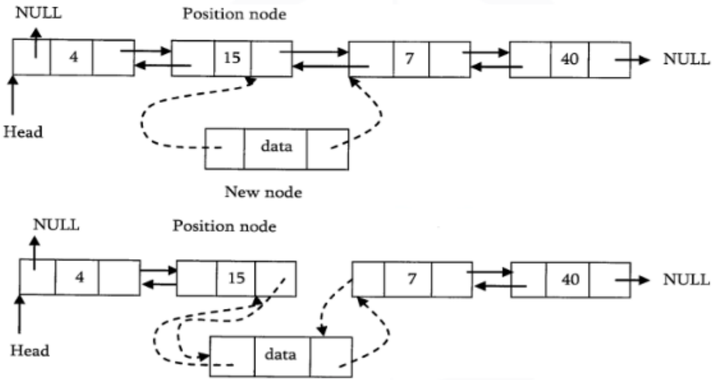


Figure: Insert a Node in between Two Nodes

- **Insert a Node after a Given Node in a Doubly Linked List**

Algorithm Insert_After(Curr, Temp):

```
Set Temp→lptr ← Curr
Set Temp→rptr ← Curr→rptr
if Curr→rptr ≠ NULL then
    Set Curr→rptr→lptr ← Temp
end if
Set Curr→rptr ← Temp
```

End Algorithm

- **Insert a Node at a Specified Position in a Doubly Linked List**

Algorithm Insert_At_Position(Head, Temp, position):

```
Initialize Curr ← Head
Initialize i ← 1
while Curr ≠ NULL and i < position do
    Set Curr ← Curr→ rptr
    Increment i
end while
if Curr = NULL then
    Print "Position out of range. Insertion not possible."
    Return Head
end if
Call Insert_After(Curr, Temp)
Return Head
```

End Algorithm

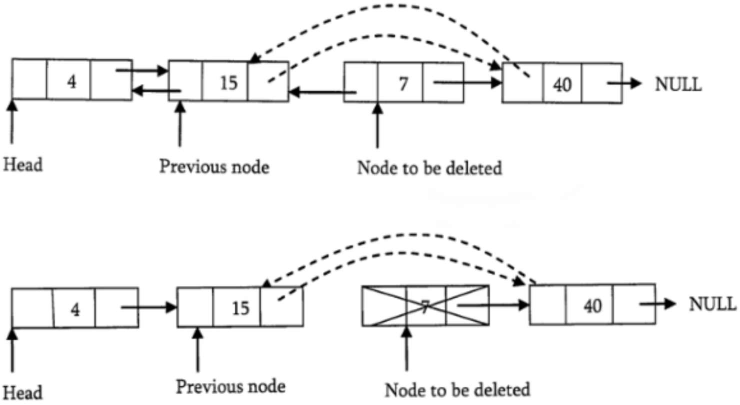


Figure: Delete a Node in between Two Nodes

- **Delete a Node at a Specified Position in a Doubly Linked List**

Algorithm Delete_At_Position(Head, position):

```
Initialize Curr ← Head;          Initialize i ← 1
if Head = NULL then
    Print "List is empty. Deletion not possible.";          Return NULL
end if
while Curr ≠ NULL and i < position do
    Set Curr ← Curr→ rptr
    Increment i
end while
if Curr = NULL then
    Print "Position out of range. Deletion not possible."
    Return Head
end if
```

- **Delete a Node at a Specified Position in a Doubly Linked List**

Algorithm Delete_At_Position(Head, position):

if Curr$\rightarrow$lptr $\neq$ NULL **then**
    Set Curr$\rightarrow$lptr$\rightarrow$rptr $\leftarrow$ Curr$\rightarrow$rptr
else
    Set Head $\leftarrow$ Curr$\rightarrow$rptr
end if
if Curr$\rightarrow$rptr $\neq$ NULL **then**
    Set Curr$\rightarrow$rptr$\rightarrow$lptr $\leftarrow$ Curr$\rightarrow$lptr
end if
Free Curr
Return Head

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

// Structure of the Node
typedef struct node
{
        int data;
        struct node *next, *prev;
} *NODE;
```

Figure: DLL Node Structure

```
NODE create_node(int data)
{
        NODE NewNode = ( NODE ) malloc (sizeof(struct node));
        if(NewNode == NULL)
                printf("\nOut of Memory");
        else
        {
                NewNode->data = data;
                NewNode->prev = NULL;
                NewNode->next = NULL;
        }
        return NewNode;
}
```

Figure: Create a Node

```
void display(NODE HEADER)
{
        NODE curr;
        if(HEADER == NULL)
        {
        printf("Empty List\n");
                return;
        }
        printf("\n\nHEADER-->");
        for(curr = HEADER; curr != NULL; curr = curr->next)
        printf("[%d]->",curr->data);
        printf("NULL\n\n");
}
```

Figure: Display Nodes of a DLL

```
NODE Insert_Front(int data,NODE HEADER)
{
        NODE NewNode = create_node(data);
        if (HEADER != NULL) {
                NewNode->next = HEADER;
                HEADER->prev = NewNode;
        }
        return NewNode;
}
```

Figure: Insert at the Beginning of DLL

```
NODE Insert_Pos(int data,int req_pos,NODE HEADER)
{
        NODE new,curr;
        int curr_pos;
        new = create_node(data);
        if((req_pos <= 1)||(HEADER == NULL))
        {       new->next = HEADER;
                HEADER->prev=new;
                HEADER = new;
                return HEADER;          }
```

Figure: Insert a node at a position

```
        curr = HEADER;
        curr_pos = 1;
        while(curr->next != NULL && curr_pos < (req_pos-1))
        {       curr = curr->next;
                curr_pos++;             }

        new->next = curr->next;
        new->prev=curr;
        curr->next = new;
        curr=new->next;
        curr->prev=new;

        return HEADER;
}
```

Figure: Insert a Node at a position

```
NODE Delete_First(NODE HEADER)
{
        NODE temp;
        if(HEADER == NULL)
        {
                printf("\n Empty List!!");
                return HEADER;
        }
        if(HEADER->next== NULL)
        {       printf("Deleted Node is : [%d]",HEADER->data);
                free(HEADER);
                return NULL;
        }
```

Figure: Delete the first node of a DLL

```
    temp = HEADER;
    HEADER = HEADER->next;
    HEADER->prev=NULL;
    printf("Deleted Node is:[%d]",temp->data);
    free(temp);
    return HEADER;
}
```

Figure: Delete the first node of a DLL

```
NODE Delete_Last(NODE HEADER)
{
        NODE temp;
        if(HEADER == NULL)
        {       printf("\n Empty List!!!");
                return HEADER;
        }
        if(HEADER->next== NULL)
        {       printf("Deleted Node is : [%d]",HEADER->data);
                free(HEADER);
                return NULL;
        }
```

Figure: Delete last node of DLL

```
    temp = HEADER;
    while(temp->next != NULL)
            temp=temp->next;
    printf("Deleted Node is : [%d]",temp->data);
    (temp->prev)->next=NULL;
    free(temp);
    return HEADER;
}
```

Figure: Delete last node of DLL

- **Additional Exercises**
  - Insert a Node in Order in a DLL
  - Delete a Node at a particular Position in a DLL
  - Reverse a Doubly Linked List
  - Count the number of nodes in a Doubly Linked List
  - Sort a Doubly Linked List
- **Note:** These Operations have been discussed in the context of SLL.
- Students are advised to build the logic and implement the same.

Thank You

Dilip Kumar Maripuri
Associate Professor
Department of Computer Applications
dilip.maripuri@pes.edu
8073212026