

Linked List

Introduction

- List of elements which are connected in sequence to each other by a set of pointers.
- Commonly used linear data structure.
- Each element is known as a node.
- A node consists of two parts
 - Data (value or values to be stored in a node).
 - Pointer (link to other nodes in a list).
- Types
 - Singly, Doubly, and Circular.

Contd...

- Advantages
 - Dynamic in nature, i.e. allocates memory when required.
 - Insertion and deletion operations can be executed easily.
 - Stacks and queues can be implemented easily.
 - Expands easily in real time.
 - Efficient memory utilization, i.e. no need to pre-allocate memory.
- Disadvantages
 - Wastage of memory as pointers require extra memory space.
 - No random access; everything sequential.
 - Reverse traversal is difficult if it's singly.
 - Memory space restriction as new node can only be created if space is available in heap.

Operations

- Traversal (Searching, Displaying)
- Insertion
 - At the beginning.
 - At the end.
 - At a specific location.
- Deletion
 - At the beginning.
 - At the end.
 - At a specific location.

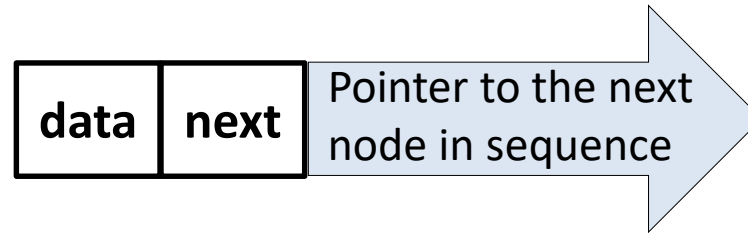
Singly Linked List

Introduction

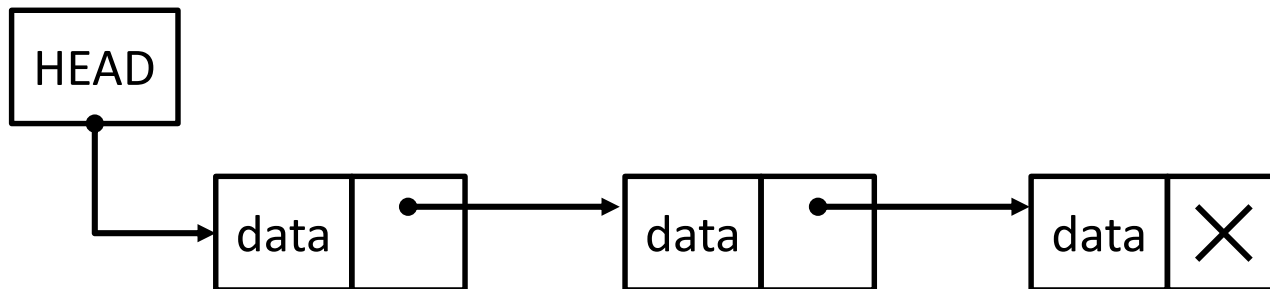
- The most basic type of linked list.
- Two successive nodes are linked together as each node contains address of the next node to be followed, i.e. successor.
- A node may have multiple data fields but only single link for the next node.
- Only forward sequential access is possible (or unidirectional).
- Address of the first node is always stored in a reference node known as **front** or **head**.
- The last node does not have any successor and has reference to **NULL**.

Contd...

- Pictorial representation of a node



- Pictorial representation of a singly linked list



Creation

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

- Define node structure.
- Declare a NULL initialized head node pointer to create an empty list.
- Dynamically allocate memory for a node and initialize all members of a node.

```
struct node *head = NULL;
```

```
struct node *temp =
    (struct node *) malloc (sizeof(struct node));
int num;
scanf("%d",&num);
temp -> data = num;
temp -> next = NULL;
```


Contd...

```
head = temp;
```

- Link the new node temp in the existing empty list.
- Again dynamically allocate memory for a node and initialize all members of a node.

```
*temp = (struct node *) malloc (sizeof(struct node));  
scanf("%d",&num);  
temp -> data = num;  
temp -> next = NULL;
```

- Link the new node temp in the existing list at head.

```
temp -> next = head;  
head = temp;
```

- This process is repeated for all the nodes. A node can be inserted anywhere in the list.

Insertion at beginning of the list

- **Algorithm** insertBeg(head, num)
 - **Input:** Pointer to the first node (**head**) and a new value to insert (**num**).
 - **Output:** Node with value **num** gets inserted at the first position.
1. Create a node pointer (**temp**).
 2. **temp[data] = num.**
 3. **temp[next] = head.**
 4. **head = temp.**

Display elements in the list

- **Algorithm** display(head)
 - **Input:** Pointer to the first node (**head**).
 - **Output:** Display all the elements present in the list.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Return.
 4. Initialize a node pointer (**temp**) with **head**.
 5. while (**temp** is not **NULL**)
 6. Print [**temp[data]**].
 7. **temp = temp[next]**.

Search an element in the list

- **Algorithm** search(head, num)
 - **Input:** Pointer to the first node (**head**) and a value to search (**num**).
 - **Output:** Appropriate message will be displayed.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Return.
 4. Initialize a node pointer (**temp**) with **head**.
 5. while (**temp** is not **NULL** AND **temp[data]** is not equal to **value**)
 6. **temp = temp[next]**
 7. if (**temp** is **NULL**)
 8. Print [**Element not found**].
 9. Else
 10. Print [**Element found**].

Insertion at end of the list

- **Algorithm** insertEnd(head, num)
 - **Input:** Pointer to the first node (**head**) and a new value to insert (**num**).
 - **Output:** Node with value **num** gets inserted at the last position.
1. Create a node pointer (**temp**).
 2. **temp[data] = num**
 3. **temp[next] = NULL**
 4. **If (head == NULL)**
 5. **head = temp**
 6. **Else**
 7. Initialize a node pointer (**temp1**) with **head**.
 8. **while (temp1[next] is not equal to NULL)**
 9. **temp1 = temp1[next]**
 10. **temp1[next] = temp**

Insertion after a specific value in the list

- **Algorithm** insert(head, num, value)
 - **Input:** Pointer to the first node (**head**) and a new value to insert (**num**) after an existing **value**.
 - **Output:** Node with value **num** gets inserted after node with **value**.
1. Create a node pointer (**temp**).
 2. **temp[data] = num**
 3. **temp[next] = NULL**
 4. Initialize a node pointer (**temp1**) with **head**.
 5. while (**temp1 != NULL AND temp1[data] != value**)
 6. **temp1 = temp1[next]**

Contd...

7. if (**temp1 == NULL**)
8. print [**Node is not present in the list**]
9. else
10. **temp[next] = temp1[next]**
11. **temp1[next] = temp**
12. end if (line 7).

Delete from beginning of the list

- **Algorithm** deleteBeg(head)
 - **Input:** Pointer to the first node (**head**).
 - **Output:** The first node gets deleted.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Else
 4. initialize a node pointer (**temp**) with **head**.
 5. **head = head[next]**
 6. Release the memory location pointed by **temp**.
 7. end if

Delete from end of the list

- **Algorithm** deleteEnd(head)
 - **Input:** Pointer to the first node (**head**).
 - **Output:** The last node gets deleted.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Else
 4. initialize a node pointer (**temp**) with **head**.
 5. while (**temp[next]** is not **NULL**)
 6. initialize a node pointer (**pre**) with **temp**.
 7. **temp = temp[next]**
 8. if (**temp == head**)
 9. **head = NULL**
 10. else
 11. **pre[next] = NULL**
 12. Release the memory location pointed by **temp**.
 13. end if

Delete a specific node from the list

- **Algorithm** deleteSpecific(head,num)
 - **Input:** Pointer to the first node (**head**) and a value **num** to be deleted.
 - **Output:** The node with value **num** gets deleted.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Else
 4. initialize a node pointer (**temp**) with **head**.
 5. while (**temp** is not **NULL** AND **temp[data]** is not equal to **value**)
 6. initialize a node pointer (**pre**) with **temp**.
 7. **temp = temp[next]**
 8. if (**temp** is **NULL**)
 9. Print [**Element not found**].
 10. Return.

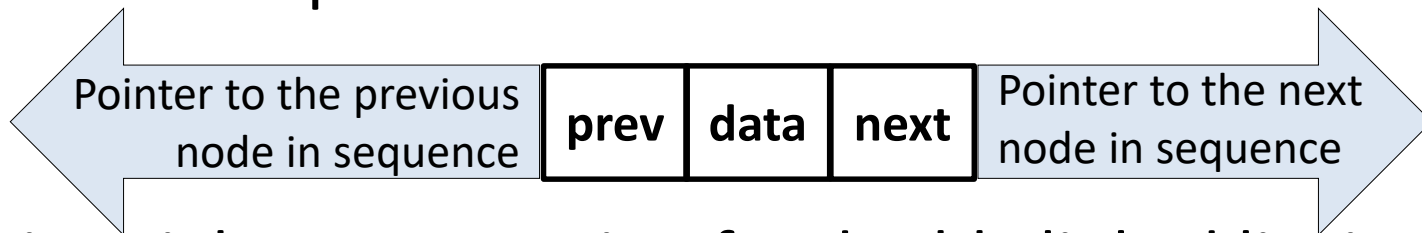
Contd...

```
12.     else if (temp == head)
13.         head = head[next]
14.     else if (temp[next] == NULL)
15.         pre[next] = NULL
16.     else
17.         pre[next] = temp[next]
18.     Release the memory location pointed by temp.
19. end if (line 8).
20. end if (line 1).
```

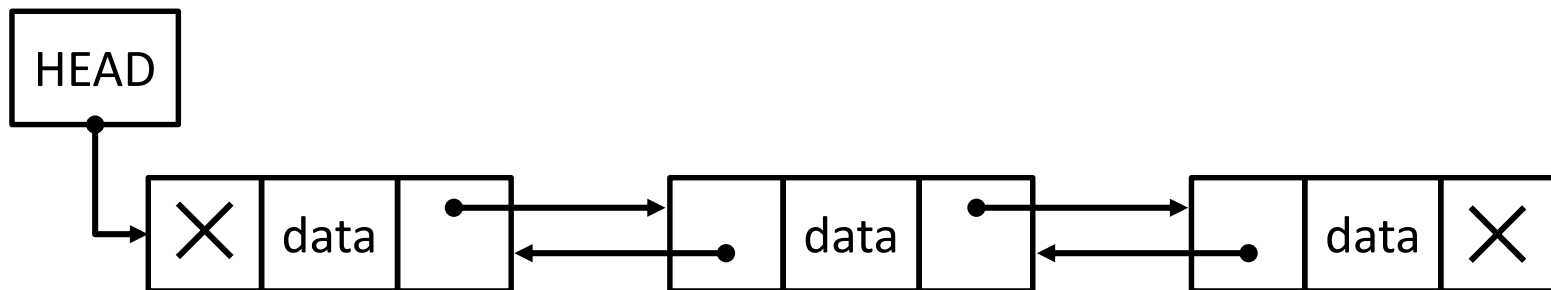
Doubly Linked List

Introduction

- Sequence of elements in which every element has links to its previous element and next element in the sequence.
- Each node contains three fields: data, link to the next node, and link to the previous node.



- The pictorial representation for doubly linked list is as shown below:



Contd...

- Advantages:
 - Can be traversed in either direction.
 - Some operations, such as deletion and insertion before a node, become easier.
- Disadvantages:
 - Requires more space.
 - List manipulations are slower.
 - Greater chances of having bugs.

Creation

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

- Define node structure.
- Declare a NULL initialized head node pointer to create an empty list.
- Dynamically allocate memory for a node and initialize all members of a node.

```
struct node *head = NULL;
```

```
struct node *temp =
    (struct node *) malloc (sizeof(struct node));
int num;
scanf("%d",&num);
temp -> data = num;
temp -> prev = temp -> next = NULL;
```

Contd...

```
head = temp;
```

- Link the new node temp in the existing empty list.
- Again dynamically allocate memory for a node and initialize all members of a node.

```
*temp = (struct node *) malloc (sizeof(struct node));  
scanf("%d",&num);  
temp -> data = num;  
temp -> prev = temp -> next = NULL;
```

- Link the new node temp in the existing list at head.

```
temp -> next = head; head -> prev = temp;  
head = temp;
```

- This process is repeated for all the nodes. A node can be inserted anywhere in the list.

Insertion at beginning of the list

- **Algorithm** insertBeg(head, num)
 - **Input:** Pointer to the first node (**head**) and a new value to insert (**num**).
 - **Output:** Node with value **num** gets inserted at the first position.
1. Create a node pointer (**temp**).
 2. **temp[data] = num.**
 3. **temp[prev] = NULL.**
 4. **temp[next] = head.**
 5. **if (head == NULL)**
 6. **head = temp.**
 7. **else**
 8. **head[prev] = temp.**
 9. **head = temp.**

Search an element in the list

- **Algorithm** search(head, num)
 - **Input:** Pointer to the first node (**head**) and a value to search (**num**).
 - **Output:** Appropriate message will be displayed.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Return.
 4. Initialize a node pointer (**temp**) with **head**.
 5. while (**temp** is not **NULL** AND **temp[data]** is not equal to **value**)
 6. **temp = temp[next]**
 7. if (**temp** is **NULL**)
 8. Print [**Element not found**].
 9. Else
 10. Print [**Element found**].

Display elements in the list

- **Algorithm** display(head)
 - **Input:** Pointer to the first node (**head**).
 - **Output:** Display all the elements present in the list.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Return.
 4. Initialize a node pointer (**temp**) with **head**.
 5. while (**temp** is not **NULL**)
 6. Print [**temp[data]**].
 7. **temp = temp[next]**.

Insertion at end of the list

- **Algorithm** insertEnd(head, num)
 - **Input:** Pointer to the first node (**head**) and a new value to insert (**num**).
 - **Output:** Node with value **num** gets inserted at the last position.
1. Create a node pointer (**temp**).
 2. **temp[data] = num**
 3. **temp[prev] = temp[next] = NULL**
 4. **If (head == NULL)**
 5. **head = temp**
 6. **Else**
 7. Initialize a node pointer (**temp1**) with **head**.
 8. **while (temp1[next] is not equal to NULL)**
 9. **temp1 = temp1[next]**
 10. **temp1[next] = temp**
 11. **temp[prev] = temp1**

Insertion after a specific value in the list

- **Algorithm** insert(head, num, value)
 - **Input:** Pointer to the first node (**head**) and a new value to insert (**num**) after an existing **value**.
 - **Output:** Node with value **num** gets inserted after node with **value**.
1. Create a node pointer (**temp**).
 2. **temp[data] = num**
 3. **temp[prev] = temp[next] = NULL**
 4. If (**head == NULL**)
 5. **head = temp**
 6. Else
 7. Initialize a node pointer (**temp1**) with **head**.
 8. while (**temp1** is not **NULL** AND **temp1[data]** is not equal to **value**)
 9. **temp1 = temp1[next]**

Contd...

```
10.    if (temp1 == NULL)
11.        print [Node is not present in the list]
12.    else if (temp1[next] is NULL)
13.        temp1[next] = temp
14.        temp[prev] = temp1
15.    else
16.        temp[prev] = temp1
17.        temp[next] = temp1[next]
18.        temp1[next] = temp
19.        (temp[next])[prev] = temp
20.    end if (line 10).
21. End if (line 4).
```

Delete from beginning of the list

- **Algorithm** deleteBeg(head)
 - **Input:** Pointer to the first node (**head**).
 - **Output:** The first node gets deleted.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Else
 4. initialize a node pointer (**temp**) with **head**.
 5. **head = head[next]**
 6. if (**head != NULL**)
 7. **head[prev] = NULL**
 8. Release the memory location pointed by **temp**.

Delete from end of the list

- **Algorithm** deleteEnd(head)
 - **Input:** Pointer to the first node (**head**).
 - **Output:** The last node gets deleted.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Else
 4. initialize a node pointer (**temp**) with **head**.
 5. while (**temp[next]** is not **NULL**)
 6. **temp = temp[next]**
 7. if (**temp == head**)
 8. **head = NULL**
 9. else
 10. (**temp[prev]**)[**next**] = **NULL**
 11. Release the memory location pointed by **temp**.
 12. end if

Delete a specific node from the list

- **Algorithm** deleteSpecific(head,num)
 - **Input:** Pointer to the first node (**head**) and a value **num** to be deleted.
 - **Output:** The node with value **num** gets deleted.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Else
 4. initialize a node pointer (**temp**) with **head**.
 5. while (**temp != NULL AND temp[data] != value**)
 6. **temp = temp[next]**
 7. if (**temp is NULL**)
 8. Print [**Element not found**].
 9. Return.

Contd...

```
10.     else if (temp == head)
11.         deleteBeg(head)
12.     else if (temp[next] == NULL)
13.         deleteEnd(head)
14.     else
15.         (temp[prev][next] = temp[next])
16.         (temp[next][prev] = temp[prev])
17.         Release the memory location pointed by temp.
18.     end if (line 7).
19. end if (line 1).
```

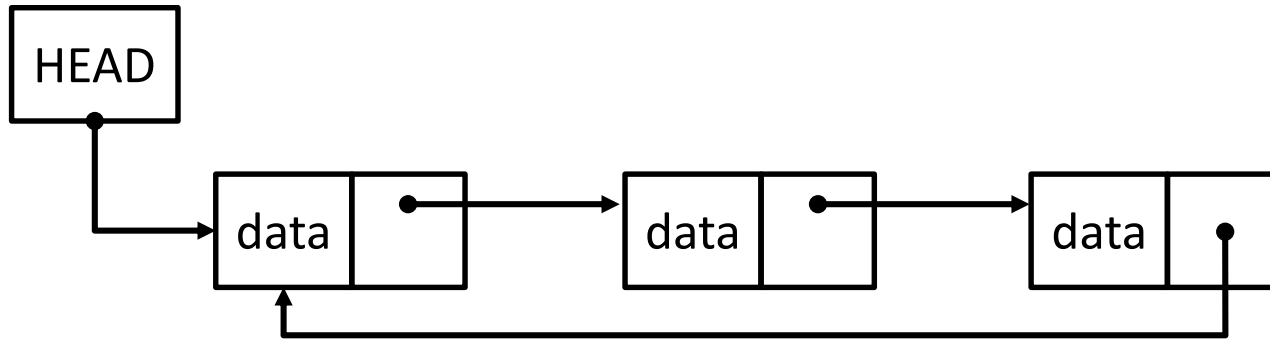
Circular Linked List

Introduction

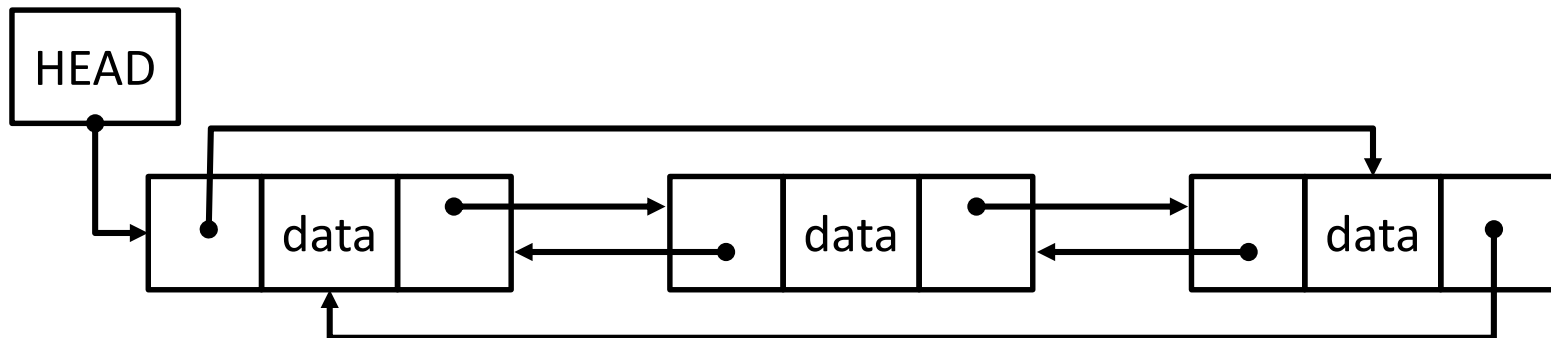
- The first element points to the last element and the last element points to the first element.
- There is no NULL node.
- While traversal, get back to a node from where you have started.
- Pointer to any node can serve as a handle to the complete list.
- Both singly and doubly linked lists can be circular.

Contd...

- Singly linked list as circular



- Doubly linked list as circular



Creation

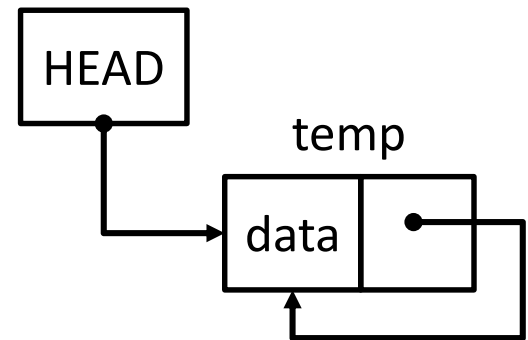
- Define node structure.
- Declare a NULL initialized head node pointer to create an empty list.
- Dynamically allocate memory for a node and initialize all members of a node.
- Link the new node temp in the existing empty list.
- Again dynamically allocate memory for a node and initialize all members of a node.
- Link the new node temp in the existing list at head.
- This process is repeated for all the nodes. A node can be inserted anywhere in the list.

Contd...

- Link the new node temp in the existing empty list.

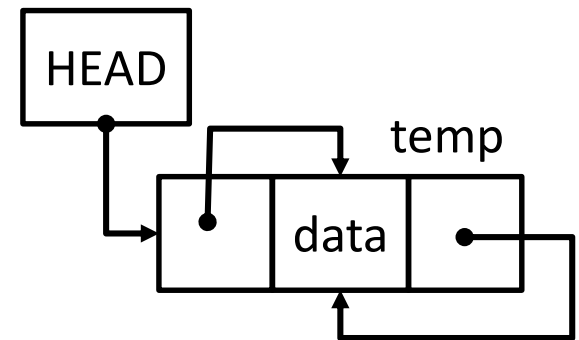
- Singly

```
head = temp;  
temp -> next = head;
```



- Doubly

```
head = temp;  
temp -> prev = head;  
temp -> next = head;
```



Contd...

- Link the new node temp in the existing list at head.
- Singly: temp1 is a node pointer pointing to the last node in a linked list.

```
temp -> next = head;  
temp1 -> next = temp;  
head = temp;
```

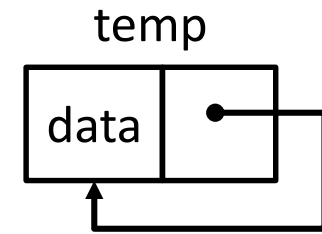
- Doubly

```
temp -> next = head;  
temp -> prev = head -> prev;  
head -> prev = temp;  
temp -> prev -> next = temp;  
head = temp;
```


Insertion at beginning of the list (singly)

- **Algorithm** insertBeg(head, num)
- **Input:** Pointer to the first node (**head**) and a new value to insert (**num**).
- **Output:** Node with value **num** gets inserted at the first position.

1. Create a node pointer (**temp**).
2. **temp[data] = num.**
3. **if (head == NULL)**
4. **temp[next] = temp.**



Contd...

5. else
6. **temp[next] = head.**
7. Initialize a node pointer (**temp1**) with **head**.
8. while (**temp1[next]** is not equal to **head**)
9. **temp1 = temp1[next]**
10. **temp1[next] = temp.**
11. end if (line 3).
12. **head = temp.**

Search an element in the list

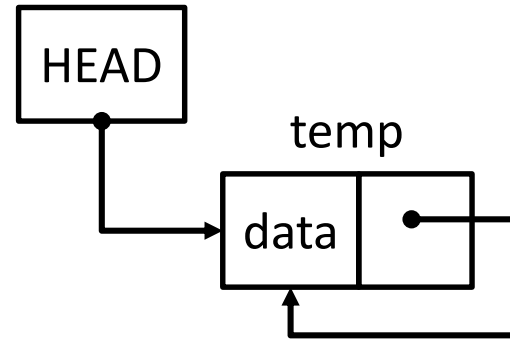
- **Algorithm** search(head, num)
 - **Input:** Pointer to the first node (**head**) and a value to search (**num**).
 - **Output:** Appropriate message will be displayed.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Return.
 4. Initialize a node pointer (**temp**) with **head**.
 5. while (**temp[next] != head AND temp[data] != value**)
 6. **temp = temp[next]**
 7. if (**temp[data] == value**)
 8. Print [**Element found**].
 9. Else
 10. Print [**Element not found**].

Display elements in the list

- **Algorithm** display(head)
 - **Input:** Pointer to the first node (**head**).
 - **Output:** Display all the elements present in the list.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Return.
 4. Initialize a node pointer (**temp**) with **head**.
 5. while (**temp[next]** is not **head**)
 6. Print [**temp[data]**].
 7. **temp = temp[next]**.
 8. Print [**temp[data]**].

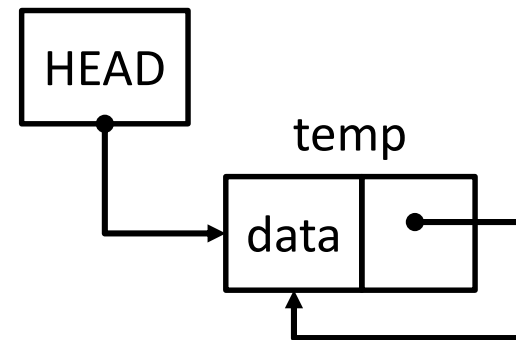
Insertion at end of the list (singly)

- **Algorithm** insertEnd(head, num)
 - **Input:** Pointer to the first node (**head**) and a new value to insert (**num**).
 - **Output:** Node with value **num** gets inserted at the last position.
1. Create a node pointer (**temp**).
 2. **temp[data] = num**
 3. If (**head == NULL**)
 4. **temp[next] = temp**
 5. **head = temp**
 6. Else
 7. Initialize a node pointer (**temp1**) with **head**.
 8. while (**temp1[next]** is not equal to **head**)
 9. **temp1 = temp1[next]**
 10. **temp1[next] = temp**
 11. **temp[next] = head**



Insertion after a specific value in the list (singly)

- **Algorithm** insert(head, num, value)
 - **Input:** Pointer to the first node (**head**) and a new value to insert (**num**) after an existing **value**.
 - **Output:** Node with value **num** gets inserted after node with **value**.
1. Create a node pointer (**temp**).
 2. **temp[data] = num**
 3. If (**head == NULL**)
 4. **temp[next] = temp**
 5. **head = temp**



Contd...

6. else
7. Initialize a node pointer (**temp1**) with **head**.
8. while (**temp1[next] != head AND temp1[data] != value**)
9. **temp1 = temp1[next]**
10. if (**temp1[data] != value**)
11. print [**Node is not present in the list**]
12. else
13. **temp[next] = temp1[next]**
14. **temp1[next] = temp**
15. end if (line 10).
16. End if (line 3).

Delete from beginning of the list (singly)

- **Algorithm** deleteBeg(head)
 - **Input:** Pointer to the first node (**head**).
 - **Output:** The first node gets deleted.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Else
 4. initialize node pointers (**temp** and **temp1**) with **head**.
 5. while (**temp1[next]** is not equal to **head**)
 6. **temp1 = temp1[next]**
 7. if (**temp1 == head**)
 8. **head == NULL**
 9. else
 10. **temp1[next] = head[next]**.
 11. **head = head[next]**
 12. Release the memory location pointed by **temp**.

Delete from end of the list (singly)

- **Algorithm** deleteEnd(head)
 - **Input:** Pointer to the first node (**head**).
 - **Output:** The last node gets deleted.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Else
 4. initialize a node pointer (**temp**) with **head**.
 5. while (**temp[next]** is not **head**)
 6. initialize a node pointer (**pre**) with **temp**.
 7. **temp = temp[next]**
 8. if (**temp == head**)
 9. **head = NULL**
 10. else
 11. **pre[next] = head**
 12. Release the memory location pointed by **temp**.

Delete a specific node from the list (singly)

- **Algorithm** deleteSpecific(head,num)
 - **Input:** Pointer to the first node (**head**) and a value **num** to be deleted.
 - **Output:** The node with value **num** gets deleted.
1. If (**head == NULL**)
 2. Print [**List is Empty**].
 3. Else
 4. initialize a node pointer (**temp**) with **head**.
 5. while (**temp[next] != head AND temp[data] != value**)
 6. initialize a node pointer (**pre**) with **temp**.
 7. **temp = temp[next]**
 8. if (**temp[data] != value**)
 9. Print [**Element not found**].
 10. Return.

Contd...

```
11.    else if (temp == head)
12.        deleteBeg(head)
13.    else if (temp[next] == head)
14.        deleteEnd(head)
15.    else
16.        pre[next] = temp[next]
17.        Release the memory location pointed by temp.
18.    end if (line 8).
19. end if (line 1).
```

Thank You