# DS UNIT 4

[ 4-marks ]

**1. Define linked list. List its advantages over arrays.**

Definition

A linked list is a linear data structure in which elements are stored in the form of nodes.
Each node contains data and a link (pointer) to the next node in the list.
The nodes are not stored in continuous memory locations.

Advantages of Linked List over Array

- Linked lists use dynamic memory allocation, so memory is used efficiently.
- The size of a linked list can grow or shrink at runtime.
- Insertion and deletion operations are easy and fast.
- No memory wastage due to unused positions.
- Reallocation is not required when size changes.
- Elements need not be stored in contiguous memory

**2. Explain dynamic memory allocation in linked lists.**

Dynamic memory allocation means allocating memory at runtime instead of compile time. In linked lists, memory is allocated whenever a new node is created.

Each node in a linked list is created dynamically using functions like malloc() in C or new in C++. This allows flexible use of memory.

Memory allocation happens only when data is inserted into the list. This avoids unnecessary memory usage.

When a node is deleted, the allocated memory is released back to the system. This improves memory management.

Dynamic memory allocation allows linked lists to grow and shrink easily. There is no need to define the size of the list beforehand.

This method helps in handling large data efficiently. It also avoids memory wastage that occurs in static allocation.

Dynamic allocation makes linked lists suitable for applications where the amount of data is unpredictable.

3. **What is a node? Draw the structure of a singly linked list node. What is a Node?**

In computer science, specifically in data structures, a node is a fundamental building block used to construct complex structures like linked lists, trees, and graphs.

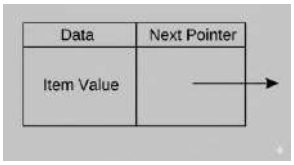
A node typically acts as a container that holds two primary types of information:

1. Data: The actual value or information you want to store (e.g., an integer, a string, a custom object).
2. Link(s) or Pointer(s): References to other nodes in the structure, which connect them together to form a sequence or hierarchy.

Structure of a Singly Linked List Node

A singly linked list is the simplest form of a linked structure. Each node in a singly linked list contains:

- Data Field: Stores the information.
- Next Pointer**:** Stores the address of the *next* node in the sequence. The very last node's pointer is set to NULL to indicate the end of the list.




4. **List different types of linked lists.**

**Singly Linked List** consists of nodes having one data field and one link field. Traversal is possible only in the forward direction.

**Doubly Linked List** contains nodes with two link fields: previous and next. This allows traversal in both forward and backward directions.

**Circular Singly Linked List** is a variation where the last node points back to the first node. This forms a circular structure with no NULL pointer.

**Circular Doubly Linked List** combines features of doubly and circular linked lists. Both previous and next pointers form a loop.

Singly linked lists use less memory compared to doubly linked lists. However, doubly linked lists provide more flexibility.

Circular linked lists are useful in applications like CPU scheduling. They allow continuous traversal of elements.

**5. Explain Traversal Operation in a Linked List**

Traversal is the process of visiting each node of a linked list one by one. It is used to access or display the data stored in the list.

Traversal always starts from the head node of the linked list. The head stores the address of the first node.

A temporary pointer variable is used to move through the list. This pointer initially points to the head node.

During traversal, the data part of the current node is accessed. After that, the pointer moves to the next node using the link field.

This process continues until the pointer becomes NULL. A NULL pointer indicates that the end of the linked list has been reached.

Traversal is necessary for operations like searching, displaying elements, and counting nodes. Without traversal, data inside the linked list cannot be accessed.

The time complexity of traversal is O(n). This is because each node is visited exactly once.

**6. Define Doubly Linked List. Mention its Applications**

**Definition**

A doubly linked list is a type of linked list in which each node contains three parts. These parts are data, a pointer to the previous node, and a pointer to the next node.

Unlike a singly linked list, traversal is possible in both forward and backward directions. This makes operations more flexible.

**Applications of Doubly Linked List**

Doubly linked lists are used in navigation systems, such as backward and forward buttons in web browsers. The previous and next pointers help move between pages.

They are used in undo and redo operations in text editors. Each action can be moved backward or forward easily.

Doubly linked lists are used in music players for previous and next song navigation. This allows smooth movement between tracks.

They are useful in implementing deques (double-ended queues). In deques, insertion and deletion can be done at both ends.

Doubly linked lists are also used in LRU cache implementation. They allow fast deletion and insertion from both ends.

**7. Explain Merge Sort on Linked List with Algorithm and Example**

**Explanation**

Merge sort is a divide and conquer sorting algorithm. It works efficiently on linked lists because random access is not required.

The linked list is divided into two halves repeatedly until each sublist contains only one node. A single node is always considered sorted.

After division, the sublists are merged in a sorted manner. This merging is done by comparing node values.

Merge sort is preferred for linked lists because it does not require extra shifting of elements. Only pointer adjustments are needed.

The time complexity of merge sort on a linked list is O(n log n). It also provides stable sorting.

**Algorithm (Simple Steps)**

- Step 1: If the list has zero or one node, return it as sorted.
- Step 2: Find the middle of the linked list using slow and fast pointers.
- Step 3: Split the list into two halves.
- Step 4: Recursively apply merge sort on both halves.
- Step 5: Merge the two sorted halves into one sorted list.

**Example**

- Consider the linked list: 8 → 3 → 5 → 2
- First, divide it into two lists: 8 → 3 and 5 → 2
- Further divide them into 8, 3, 5, and 2
- Merge sorted pairs: 3 → 8 and 2 → 5
- Final merge gives the sorted list: 2 → 3 → 5 → 8

**[ 5- marks ]**

1. **Explain representation of singly linked list with diagram.**

   A singly linked list is a linear data structure in which elements are stored in the form of nodes. Each node is connected to the next node using a link.

   Every node in a singly linked list has two parts:

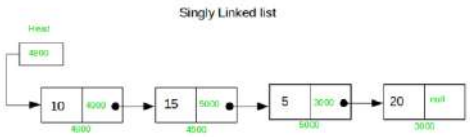
   - Data part, which stores the actual information.
   - Link (or next) part, which stores the address of the next node in the list.

   The list is accessed using a pointer called head, which stores the address of the first node. If the head is NULL, the list is empty.

   The last node of the singly linked list contains the data and its link part stores NULL, indicating the end of the list.

   Nodes are not stored in continuous memory locations. Instead, memory is allocated dynamically, which makes insertion and deletion easier.

   This structure is widely used in implementing stacks, queues, and dynamic memory management systems.




2. **Explain insertion operations in a singly linked list.**

**Insertion in a singly linked list** means adding a new node at a specific position in the list. The position can be at the beginning, at the end, or at a given location.

Before insertion, memory is dynamically allocated for the new node. The data is stored in the data part of the node and the link part is set later.

**Insertion at the beginning** is done by making the new node point to the current head node. After that, the head pointer is updated to the new node.

This type of insertion is fast because no traversal is required. It takes constant time, which makes it efficient.

**Insertion at the end** requires traversal of the list to reach the last node. The link of the last node is then updated to point to the new node.

In this case, the new node's link is set to NULL. This operation takes more time because the entire list may need to be traversed.

**Insertion at a specific position** involves traversing the list until the required position is found. The new node is inserted by adjusting the links of neighboring nodes.

Proper link adjustment is very important to avoid breaking the list. After insertion, the linked list remains connected in correct order.

**3. Explain deletion of a node from a singly linked list.**

**Deletion in a singly linked** list means removing a node from the list and freeing its memory. After deletion, the remaining nodes must stay properly linked.

Before deletion, the linked list should be checked for emptiness. If the head is NULL, deletion is not possible because the list is empty.

**Deletion at the beginning** is performed by moving the head pointer to the next node. The original first node is then deleted from memory.

This operation is simple and fast because no traversal is required. It takes constant time to complete.

**Deletion at the end** requires traversal of the list to reach the second-last node. The link of this node is set to NULL to remove the last node.

After updating the link, the last node's memory is freed. This operation takes more time because the list must be fully traversed.

**Deletion of a specific node** requires searching for the node that needs to be removed. The link of the previous node is adjusted to skip the deleted node.

Proper pointer handling is very important during deletion. Incorrect linking can break the linked list.

**4. Differentiate between singly linked list and doubly linked list.**

| Basis | Singly Linked List | Doubly Linked List |
|---|---|---|
| Structure of Node | Each node contains two parts: a data field and a link to the next node. It does not store the address of the previous node. | Each node contains three parts: a data field, a link to the previous node, and a link to the next node. This allows more connectivity. |
| Traversal Direction | Traversal is possible only in one direction, that is forward. Backward traversal is not possible because previous node information is not stored. | Traversal is possible in both forward and backward directions. This is because each node stores the address of both neighboring nodes. |
| Memory Requirement | It requires less memory because only one link pointer is stored per node. This makes it more memory efficient. | It requires more memory because two link pointers are stored per node. Extra memory is used for the previous pointer. |
| Ease of Deletion | Deletion is more complex because the previous node must be tracked during traversal. Without it, links cannot be adjusted easily. | Deletion is easier because each node has direct access to its previous node. Link adjustment becomes simple. |
| Implementation Complexity | It is easier to implement and understand due to simple structure. It is commonly used for basic applications. | It is more complex to implement due to extra pointers. Careful handling of links is required. |

5. **Explain searching and sorting operations in a linked list.**

Explain Searching and Sorting Operations in a Linked List

- **Searching** in a linked list means finding a particular element by checking each node one by one. Since nodes are not stored in order, searching starts from the head node.
- A temporary pointer is used to traverse the list and compare the given value with node data. If the value is found, the search stops.
- If the pointer reaches NULL, it means the element is not present in the list. Searching follows the linear search method.
- The time complexity of searching is O(n) because nodes are accessed sequentially.
- **Sorting** means arranging nodes in ascending or descending order. In linked lists, sorting is done by changing links, not by shifting elements.
- Common sorting methods include bubble sort and merge sort. Merge sort is efficient and commonly used for linked lists.
- Sorting improves data organization and makes searching easier.

6. **Explain merging of two linked lists. Describe an algorithm to create nodes before given node and after given node in a singly Linked List.**

Merging of two linked lists means combining two separate linked lists into a single linked list. The lists are usually merged in a sorted manner.

In merging, nodes are compared one by one and linked in proper order. No new data shifting is required; only pointers are adjusted.

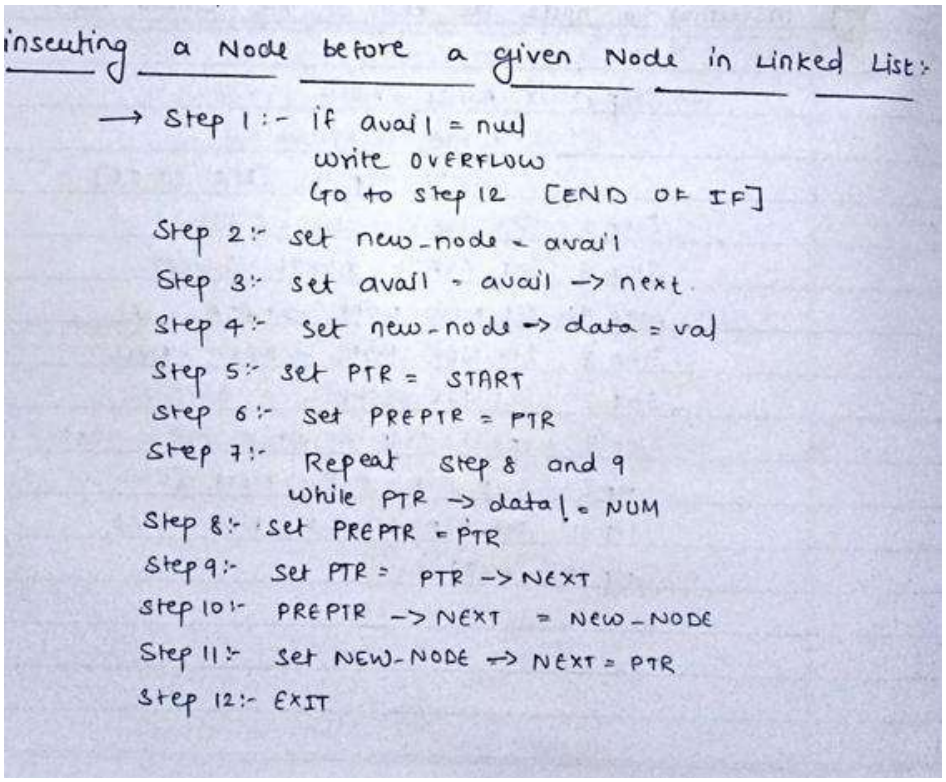
The process starts by comparing the first nodes of both lists. The smaller node becomes part of the merged list.

The pointer then moves forward in the list from which the node was taken. This process continues until one list becomes empty.

After one list ends, the remaining nodes of the other list are attached directly. The final list contains all nodes in sorted order.

Merging is efficient in linked lists because only links are changed. It is commonly used in merge sort.

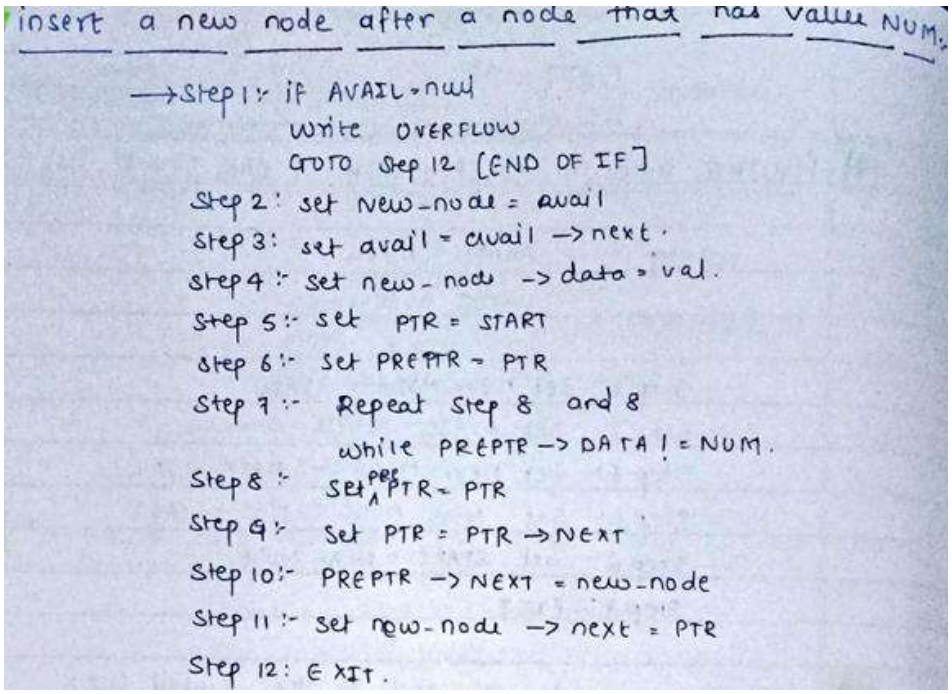
**Inserting Node before Give node :**



Inserting a Node before a given Node in Linked List:-

→ Step 1 :- if avail = null
write OVERFLOW
Go to step 12 [END OF IF]
Step 2 :- set new_node = avail
Step 3 :- set avail = avail -> next
Step 4 :- set new_node -> data = val
Step 5 :- set PTR = START
Step 6 :- Set PREPTR = PTR
Step 7 :- Repeat step 8 and 9
while PTR -> data != NUM
Step 8 :- set PREPTR = PTR
Step 9 :- Set PTR = PTR -> NEXT
Step 10 :- PREPTR -> NEXT = NEW_NODE
Step 11 :- Set NEW_NODE -> NEXT = PTR
Step 12 :- EXIT

**Inserting Node after Give node :**



Insert a new node after a node that has value NUM.

→ Step 1 :- if AVAIL = null
write OVERFLOW
GOTO Step 12 [END OF IF]
Step 2 : set NEW_node = avail
Step 3 : set avail = avail -> next
Step 4 : set new_node -> data = val.
Step 5 :- set PTR = START
Step 6 :- Set PREPTR = PTR
Step 7 :- Repeat Step 8 and 8
while PREPTR -> DATA != NUM.
Step 8 :- Set PREPTR = PTR
Step 9 :- Set PTR = PTR -> NEXT
Step 10 :- PREPTR -> NEXT = new_node
Step 11 :- Set new_node -> next = PTR
Step 12 : EXIT.

**[ 10-marks ]**

1. **Explain linked list data structure. Describe its representation and various operations with algorithms.**

**Linked List Data Structure**

A linked list is a linear data structure in which elements are stored in the form of nodes. Each node is connected to the next node using a pointer.

Unlike arrays, linked list elements are not stored in continuous memory locations. This allows efficient use of memory.

Each element of a linked list is called a node. A node consists of two parts: a data field and a link field.

The data field stores the actual value, while the link field stores the address of the next node.

The first node of the linked list is called the head. The head contains the address of the first node in the list.

The last node has its link pointing to NULL, which marks the end of the linked list.

**Representation of Linked List**

A linked list is represented using a series of nodes connected through pointers. Each node is dynamically created in memory.

The structure of a node in a singly linked list is represented as [ Data | Link ].

The link field connects one node to the next node in sequence. This forms a chain-like structure in memory.

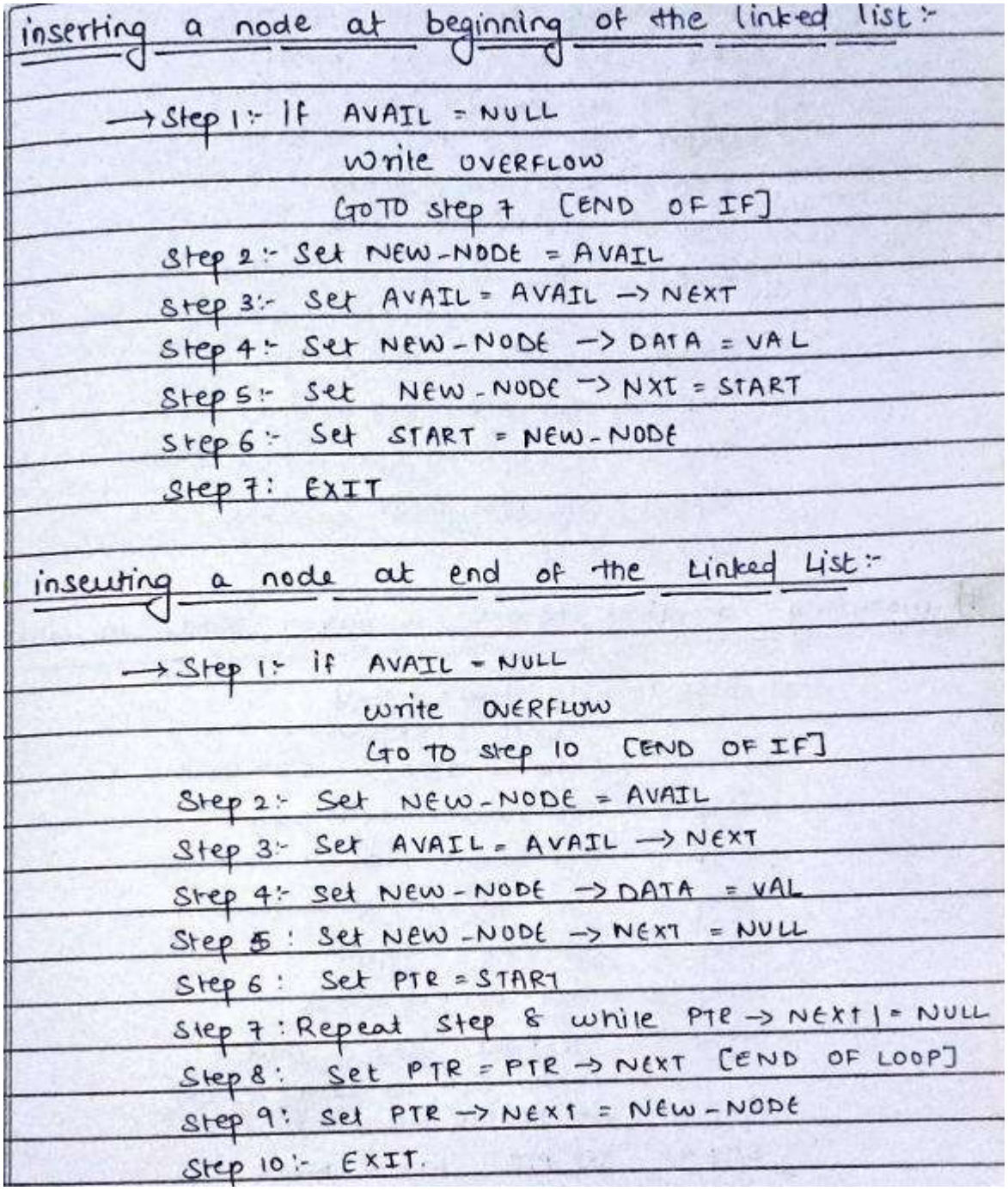
Access to nodes is sequential, meaning elements are accessed one after another starting from the head.

**Operations on Linked List**

1. Insertion Operation

Insertion is the process of adding a new node to the linked list. It can be done at the beginning, at the end, or at a specific position.

**Algorithm (Insertion at Beginning) & Algorithm (Insertion at End):**




2. Deletion Operation

- Deletion is the process of removing a node from the linked list. Proper link adjustment is required to maintain list structure.
- Algorithm (Deletion at Beginning) & Algorithm (Deletion at End):

## Deleting the 1st node of the linked list :-

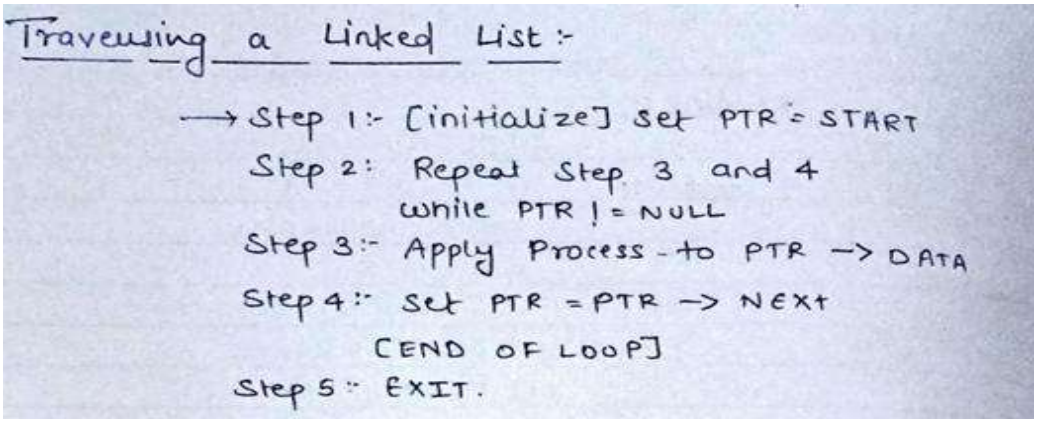
→ Step 1 :- if START = NULL
Write OVERFLOW
Go to Step 5
[END OF IF]
Step 2 :- Set PTR = START
Step 3 :- Set START = START -> NEXT
Step 4 :- FREE PTR
Step 5 :- EXIT

## deleting the last node from the Linked List :-

→ Step 1 :- if START = NULL.
Write UNDERFLOW
Go to Step 8 [END OF IF]
Step 2 :- Set PTR = START
Step 3 :- Repeat step 4 and 5
while PTR -> next != NULL
Step 4 :- set PREPTR = PTR
Step 5 :- Set PTR = PTR -> NEXT
[END OF LOOP]
Step 6 :- set PREPTR -> NEXT = NULL
Step 7 :- FREE PTR
Step 8 : EXIT.

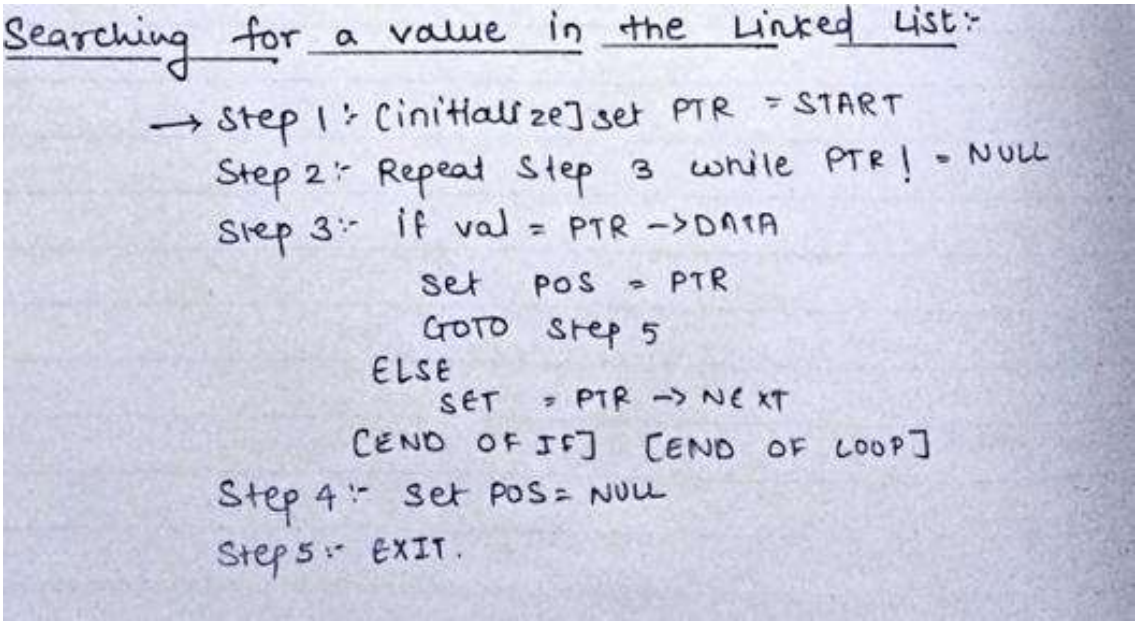
3. Traversal Operation

- Traversal is used to visit each node of the linked list. It is required to display or process data.
- Algorithm:




**4. Searching Operation**

- **Searching is used to find whether a particular element exists in the linked list. It uses sequential access.**
- **Algorithm:**



2. **Explain insertion, deletion, searching, sorting, and merging operations on singly linked list with algorithms.**

   ( for insertion deletion and searching refer above question , and sorting and merging algo for LinkedList are not given , below is from ChatGPT )

Sorting Operation

- Sorting arranges nodes in ascending or descending order. In linked lists, sorting is done by changing links instead of shifting data.
- Algorithm (Bubble Sort): Compare adjacent nodes and swap their links if they are in the wrong order. Repeat until the list is sorted.
- Algorithm (Merge Sort): Divide the list into two halves, sort each half recursively, and merge them in sorted order.

Merging Operation

- Merging combines two singly linked lists into one list, usually in sorted order. It is commonly used in merge sort.
- Algorithm: Compare the first nodes of both lists and select the smaller one. Continue comparing nodes and linking them until one list ends.
- Attach the remaining nodes of the other list to complete the merged list.

**3. Explain doubly linked list with node structure, insertion and deletion operations.**

**Doubly Linked List :**

- A doubly linked list is a linear data structure in which each node is connected to both its previous and next nodes. This allows traversal in both forward and backward directions.
- Unlike a singly linked list, each node stores two links. One link points to the previous node and the other points to the next node.
- The first node's previous link points to NULL. The last node's next link also points to NULL.
- Doubly linked lists require more memory than singly linked lists. This is because they store an extra pointer in each node.

**Node Structure of Doubly Linked List**

- Each node in a doubly linked list consists of three parts. These parts are previous link, data field, and next link.
- The previous link stores the address of the previous node. The next link stores the address of the next node.
- The data field stores the actual information. Proper linking maintains the list structure.
- Conceptually, the structure is represented as [ Prev | Data | Next ]. This structure supports two-way traversal.

**Insertion Operations**

Insertion at the Beginning

- In this operation, a new node is added before the first node. The new node becomes the head of the list.

Insertion at the End

- This operation adds a new node at the end of the list. Traversal is required to reach the last node.

**Deletion Operations**

Deletion at the Beginning

- Deletion at the beginning removes the first node of the list. The head pointer is updated.

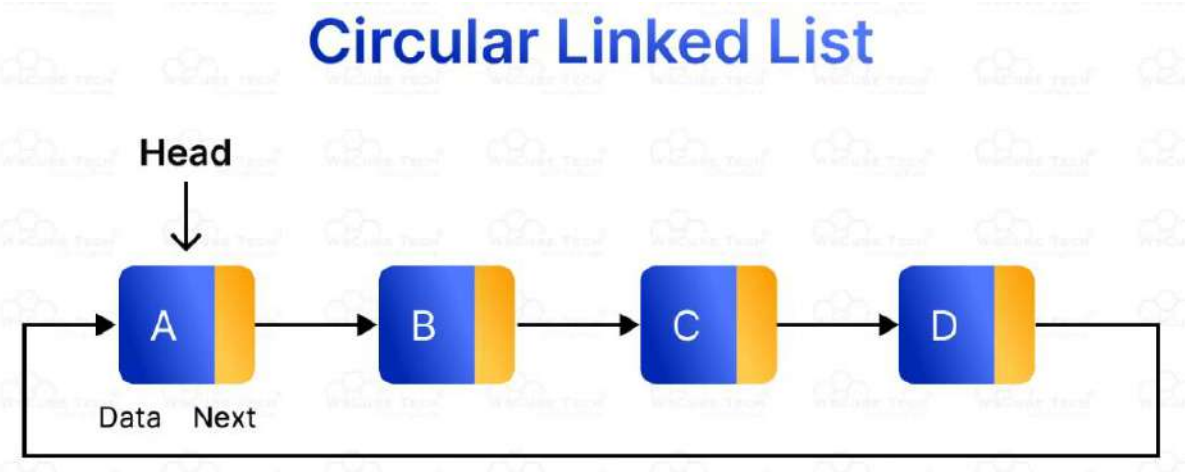
Deletion at the End

- Deletion at the end removes the last node of the list. Backward link helps simplify this operation.

**4. Explain circular linked list in detail with operations and diagrams.**

What is a Circular Linked List?

- A circular linked list is a variation of a linked list in which the last node points back to the first node instead of pointing to NULL.
- This connection forms a circular structure, meaning the list has no clear beginning or end when traversed continuously.
- Like a singly linked list, each node contains two parts: a data part and a link (next) part.
- The list is usually accessed using a pointer called head, which stores the address of the first node.
- Circular linked lists are useful in applications where continuous looping is required.




Operations on Circular Linked List

1. Insertion Operation

- Insertion at Beginning:
    - A new node is created and its link is set to the head node.
    - The last node's link is updated to point to the new node, and head is updated.
- Insertion at End:
    - A new node is created and linked after the last node.
    - The new node's next pointer is set to the head node.
- Insertion at Specific Position:
    - The list is traversed until the desired position is reached.
    - Links are adjusted to insert the new node at that position.

2. Deletion Operation

- Deletion at Beginning:
    - The head pointer is moved to the next node.
    - The last node's link is updated to point to the new head.
- Deletion at End:
    - The list is traversed to the second-last node.
    - Its next pointer is updated to point to the head node.
- Deletion of Specific Node:
    - The list is traversed until the required node is found.
    - Links are adjusted to remove that node from the list.

3. Traversal Operation

- Traversal starts from the head node.
- Each node is visited one by one until the head node is reached again.
- This ensures that all nodes are accessed exactly once.

5. **Compare singly, doubly, and circular linked lists based on structure, memory, and applications**

| Basis | Singly Linked List | Doubly Linked List | Circular Linked List |
|---|---|---|---|
| **Structure** | Each node contains two parts: data and a link to the next node only. Traversal is possible in one direction. | Each node contains three parts: data, link to previous node, and link to next node. Traversal is possible in both directions. | The last node points back to the first node instead of NULL, forming a circular structure. |
| **End of List** | The last node points to **NULL**, which marks the end of the list. | The last node's next pointer is NULL and the first node's previous pointer is NULL. | There is **no NULL pointer**; the list loops back to the head node. |
| **Memory Usage** | Requires less memory because only one link is stored per node. This makes it memory efficient. | Requires more memory because two links (previous and next) are stored in each node. | Uses slightly more memory than singly linked list due to circular reference, but similar node structure. |
| **Traversal** | Traversal is simple but only in forward direction. Backward traversal is not possible. | Traversal can be done both forward and backward, which is more flexible. | Traversal can start from any node and continues until the start node is reached again. |
| **Insertion & Deletion** | Insertion and deletion are easy but sometimes require traversal to the previous node. | Insertion and deletion are easier because previous node information is directly available. | Insertion and deletion are efficient, especially at beginning and end, due to circular linking. |
| **Implementation Complexity** | Simple to implement and easy to understand. | More complex to implement due to extra pointer handling. | Slightly more complex than singly linked list because of circular connections. |
| **Applications** | Used in stacks, queues, polynomial manipulation, and simple dynamic memory allocation. | Used in navigation systems (undo–redo), browser history, and playlist navigation. | Used in CPU scheduling, round-robin algorithms, and continuous looping applications. |