K.R. MANGALAM UNIVERSITY

THE COMPLETE WORLD OF EDUCATION

Data Structure

ENCS205

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

UNIT-2

Session 16: Introduction to Linked List

Sessions 16 Outlook

- Linked List: Definition
- Components of linked list
- Representation
- Advantages and Disadvantages

Objective

- Students should be able to lean about Dynamic data Structure.
- Implementing algorithms for creating, inserting, deleting, searching, and displaying elements in various types of linked lists.
- Analyzing the complexities of algorithms for linked list operations

Pretest

1. Which of these best describes an array?

a: A data structure that shows a hierarchical behaviour

b: Container of objects of similar types

c: Container of objects of mixed types

d: All of the mentioned

Option:b

2. What is the time complexity of insertion at any point in an array?

a: O(N)

b: O(N^2)

c: O(NLogN)

d: None of these

Option:a

Pretest

3. Pick incorrect options from the following

a: If structure contains 3 types of data then one pointer is sufficient for that structure

b: One variable can be bound to many structures

c: Contents of a structure are stored in contiguous memory

d: None of these

Option:b

4. The memory occupied by a pointer _____

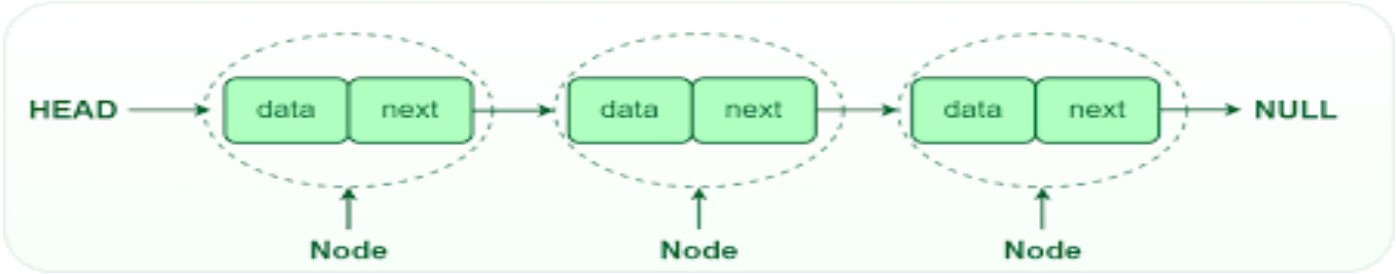
a: Is dependent on which data type the pointer points to.
b: Is independent of data type.
c: Depends on the program we are writing,
d: None of these

Option:b

Link List

Link List

- Linked list is a series of connected nodes.
- A node in the linked list contains two parts
  - ✓ first is the data part
  - ✓ second is the address part.
- The last node of the list contains a pointer to the null



https://www.geeksforgeeks.org/what-is-linked-list/

Why Linked List?

- Arrays can be used to store linear data of similar types, but arrays have the following limitations.

1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

For example

- In a system, if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

- And if we want to insert a new ID 1005, then to maintain the sorted order,
- we have to move all the elements after 1000 (excluding 1000).Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Representation

A linked list is represented by a pointer to the first node of the linked list.

- The first node is called the head. If the linked list is empty, then the value of the head is NULL.

Each node in a list consists of at least two parts:

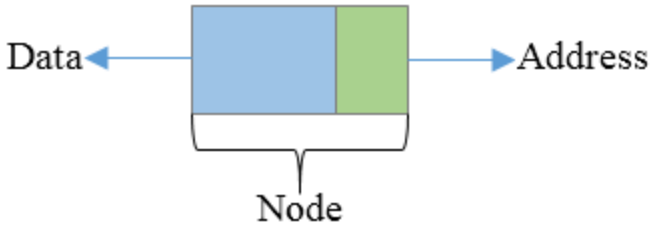
1) data

2) Pointer (Or Reference) to the next node

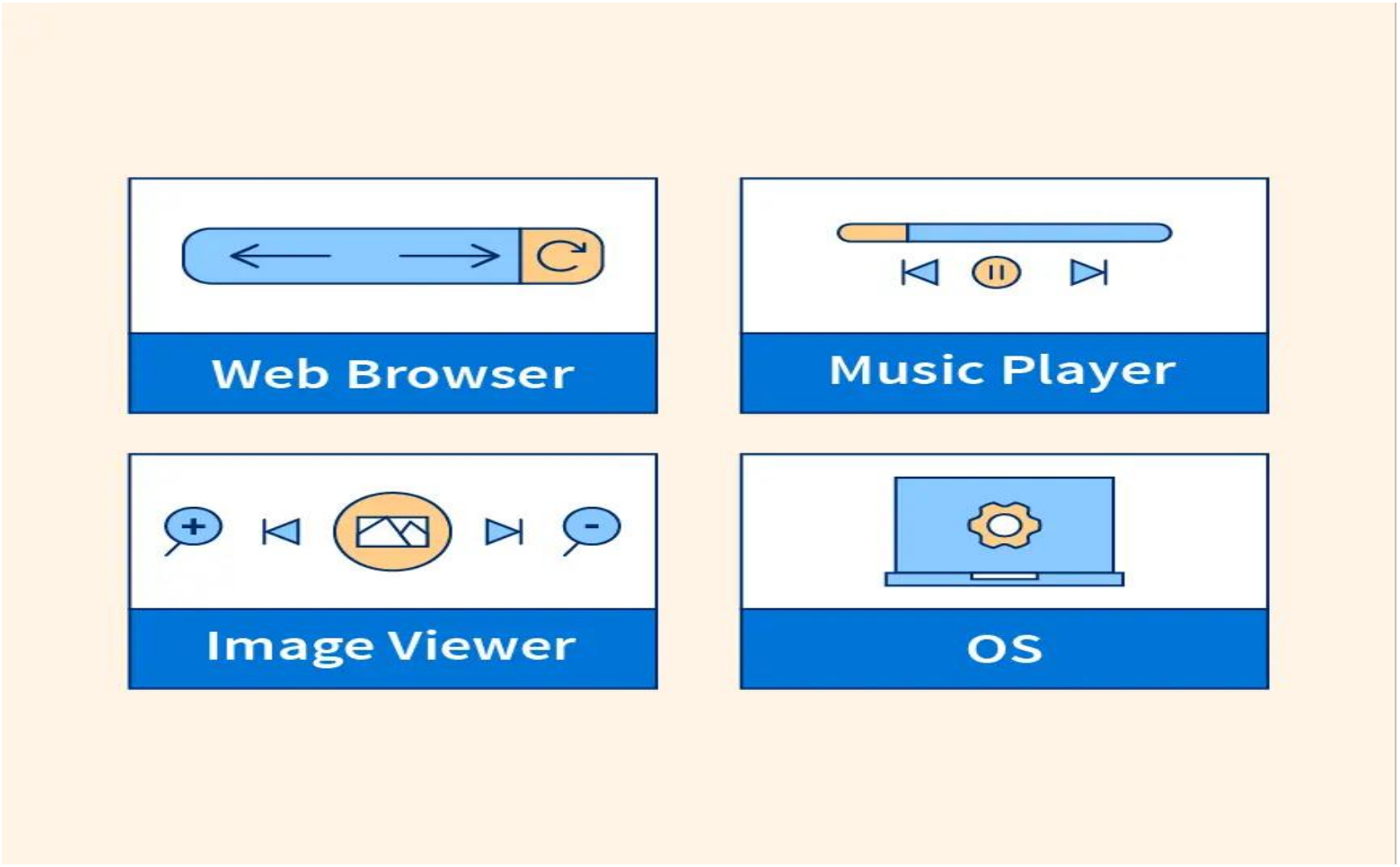
Creation of Node and Declaration of Linked Lists

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

n=(struct node*)malloc(sizeof(struct node*));
```



Applications



Applications

- Image viewer
- Previous and next page in a web browser
- Music Player
- GPS navigation systems
- Task Scheduling

Advantages And Disadvantages

Advantages

- Dynamic data structure
- Implementation
- Insertion and deletion
- Versatility
- Persistence

Disadvantages

- *Memory usage*
- *Accessing an element*
- *More complex implementation*
- *Lack of cache locality*

Importance of Linked List

- **Dynamic Data structure**
- **Ease of Insertion/Deletion.**
- **Efficient Memory Utilization**
- **Implementation**

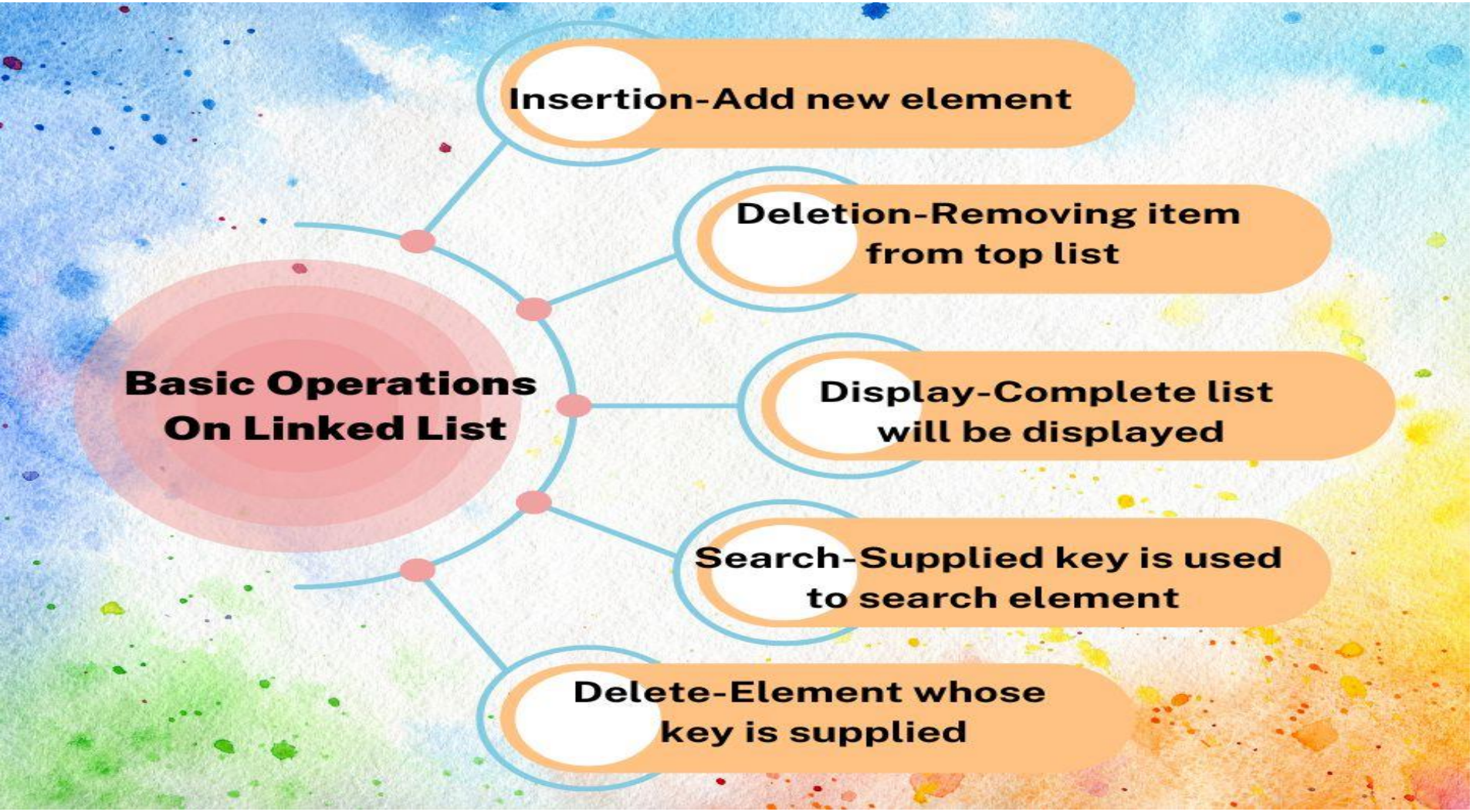
Linked List vs Array

Array	Linked List
Arrays are stored in contiguous location.	Linked Lists are not stored in contiguous location.
Fixed size.	Dynamic Size
Memory is allocated at compile time.	Memory is allocated at run time.
Uses less memory than Linked Lists.	Uses more memory than Arrays as it stores both data and address of next node.
Elements can be accessed easily in O(1) time.	Elements can be access by traversing through all the nodes till we reach the required node.
Insertion and deletion operation is slower than Linked List.	Insertion and deletion operation is faster than Linked List.

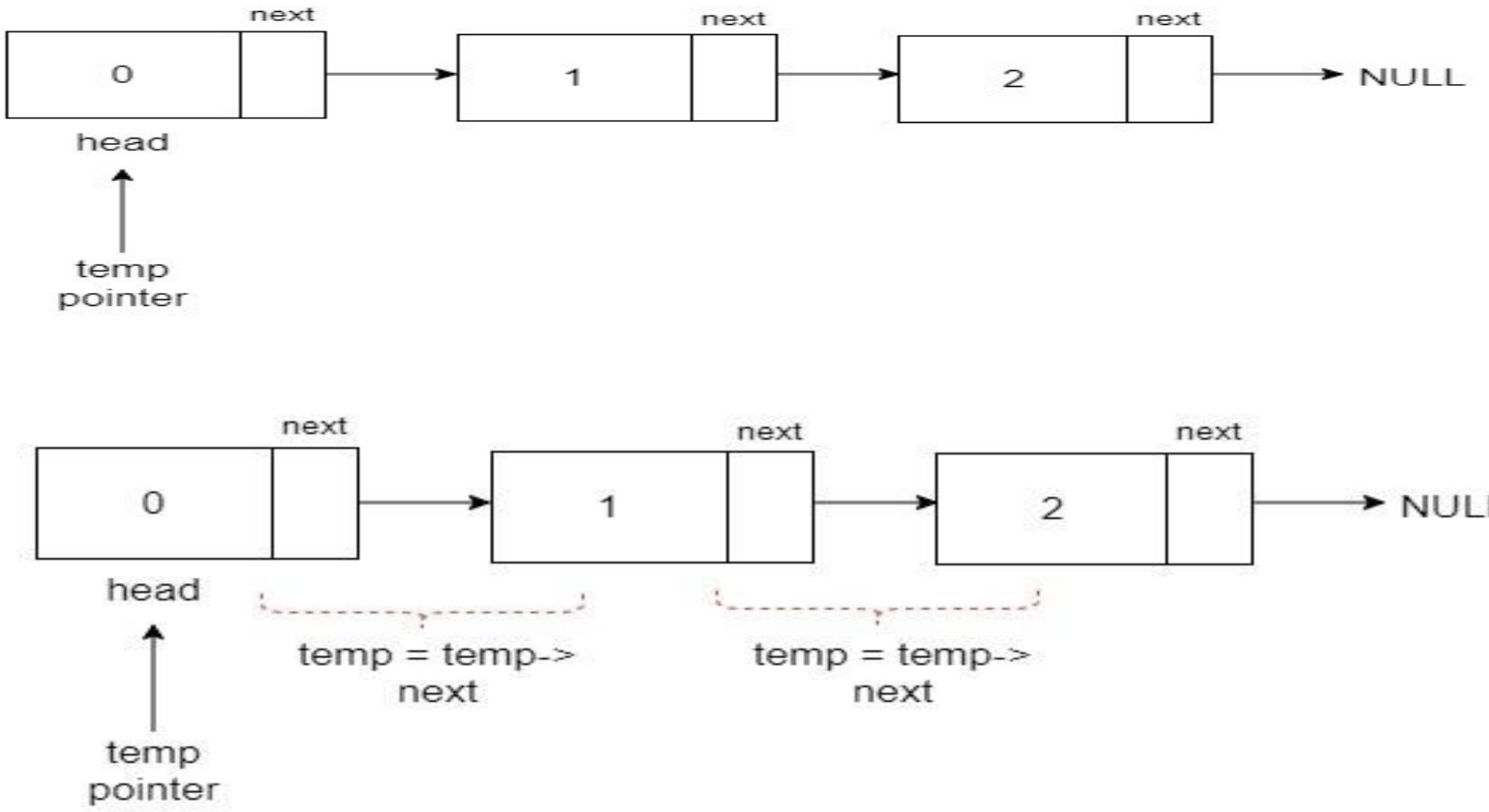
Time Complexity Analysis of Linked List and Array

Operation	Linked list	Array
Random Access	O(N)	O(1)
Insertion and deletion at beginning	O(1)	O(N)
Insertion and deletion at end	O(N)	O(1)
Insertion and deletion at a random position	O(N)	O(N)

Operations


Basic Operations On Linked List
Insertion-Add new element
Deletion-Removing item from top list
Display-Complete list will be displayed
Search-Supplied key is used to search element
Delete-Element whose key is supplied

Traversal Operations



Algorithm

STEP 1: SET PTR = HEAD

STEP 2: IF PTR = NULL

WRITE "EMPTY LIST"

GOTO STEP 7

END OF IF

STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL

STEP 5: PRINT PTR→ DATA

STEP 6: PTR = PTR → NEXT

[END OF LOOP]

STEP 7: EXIT

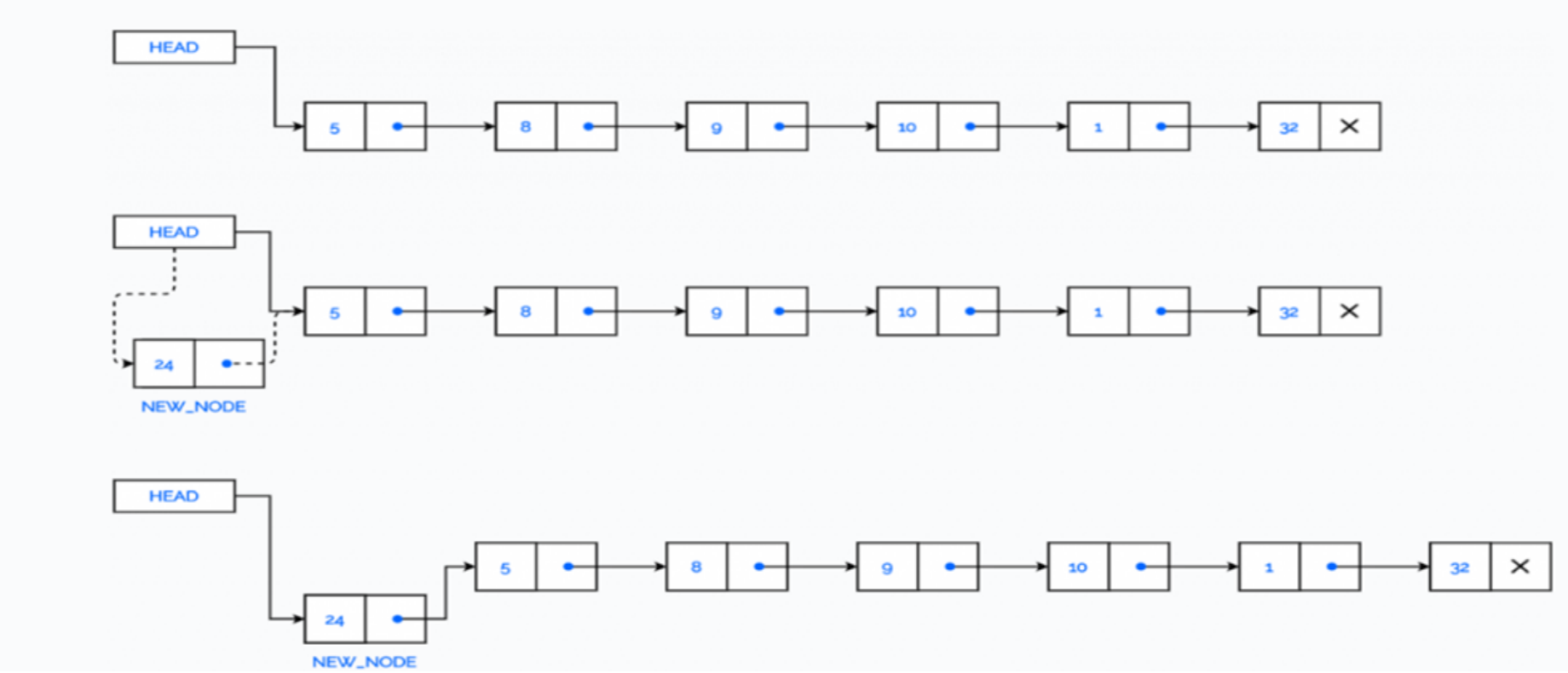
Code for traversing :

```
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL) {
printf("%d --->",temp->data);
temp = temp->next; }
```

Complete code here.

Insertion Operations At the beginning.

- At the beginning.
- At the end.
- At Location node



Algorithm: Insert At Beginning

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 7 [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 NEW_NODE -> NEXT = HEAD

Step 6: SET HEAD = NEW_NODE

Step 7: EXIT

Code for node insertion

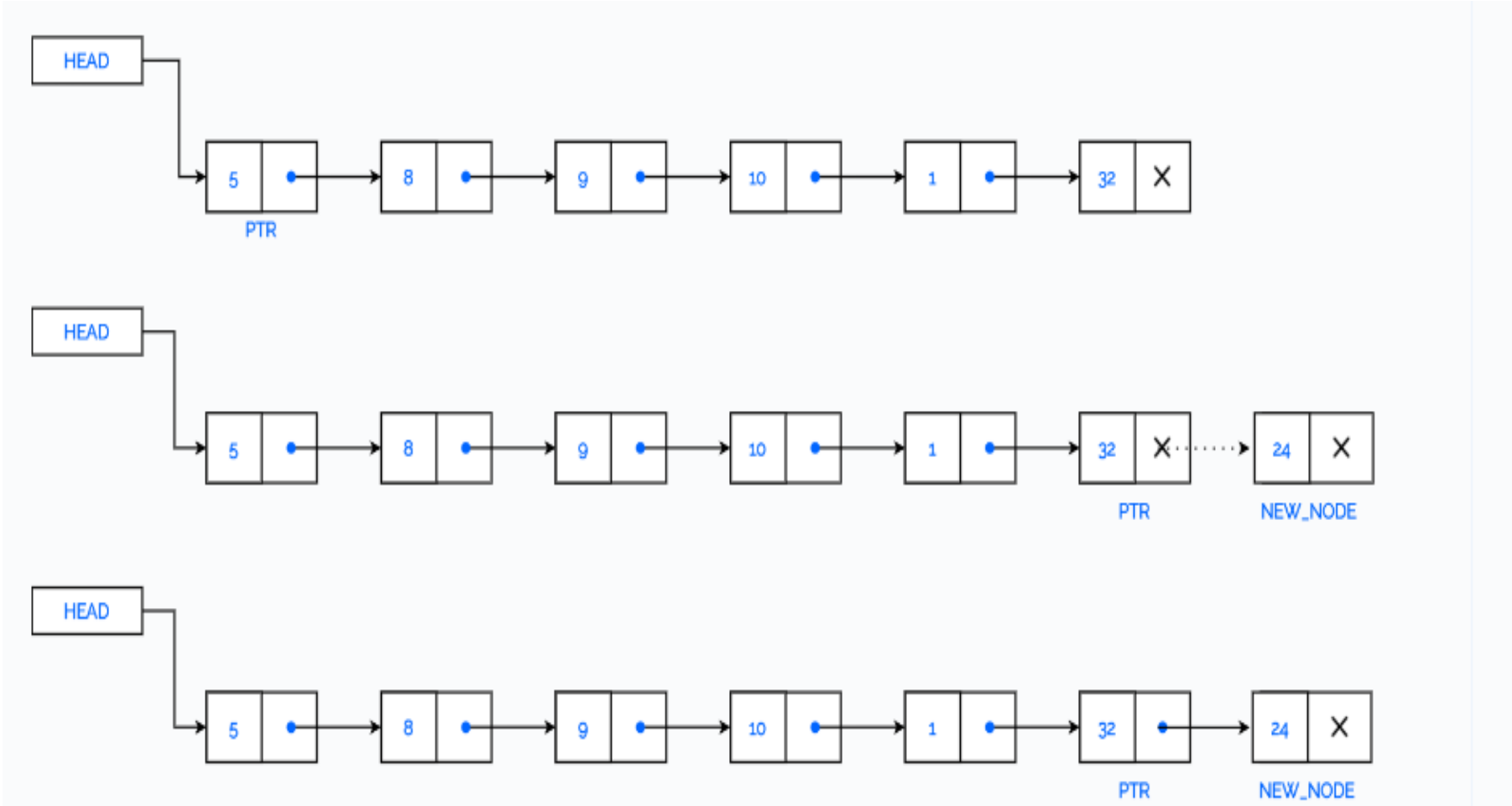
```
void LinkedList::insertFront(int data){

    Node* new_node = new Node();

    // assign data value
    new_node->data = data;
    // change the next node of this new_node
    // to current head of Linked List
    new_node->next = head;
```

Complete code here.

Inserting at the End



Algorithm: Insert At End

- Step 1: IF AVAIL = NULL
  Write OVERFLOW
  Go to Step 10 [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 NEW_NODE -> NEXT = NULL
- Step 6: SET PTR = HEAD
- Step 7: Repeat Step 8 while PTR -> NEXT != NULL Step
- Step 8: SET PTR = PTR -> NEXT [END OF LOOP]
- Step 9: SET PTR -> NEXT = NEW_NODE
- Step 10: EXIT

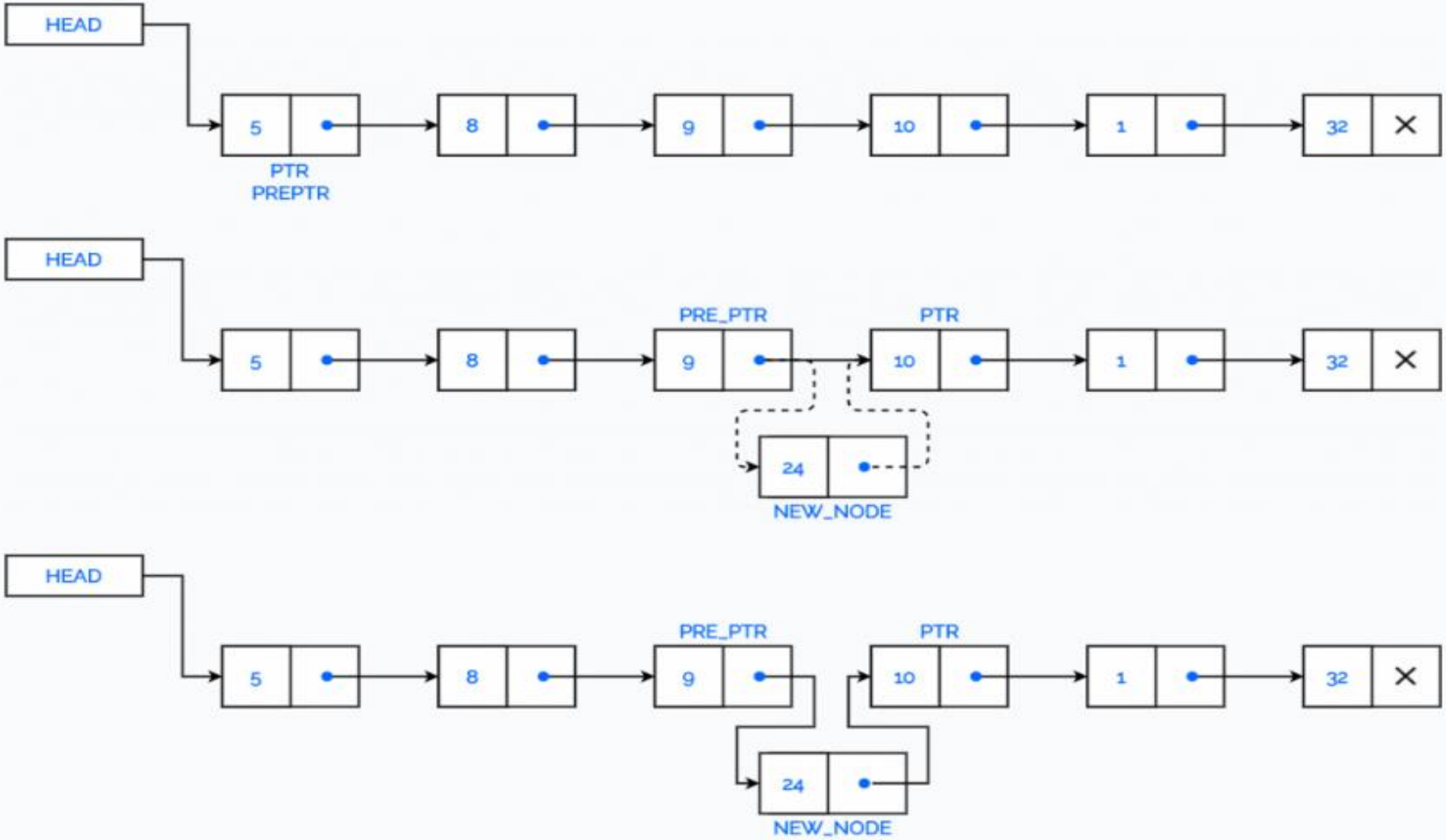
Code

```
void LinkedList::insertLast(int data)
{
   Node* freshNode = new Node();
   freshNode->data = data;
   // since this will be the last node so it will point to NULL
   freshNode->next = NULL;

   //need this if there is no node present in linked list at all
   if(head==NULL)
   {
      head = freshNode;
      cout << freshNode->data << " inserted" << endl;
      return;
   }
   struct Node* temp = head;
   // traverse to the last node of Linked List
   while(temp->next!=NULL)
      temp = temp->next;
   // assign last node's next to this freshNode
   temp->next = freshNode;
   cout << freshNode->data << " inserted" << endl;
}
```

Complete Code here

Insert a Node after a given Node in a Linked list

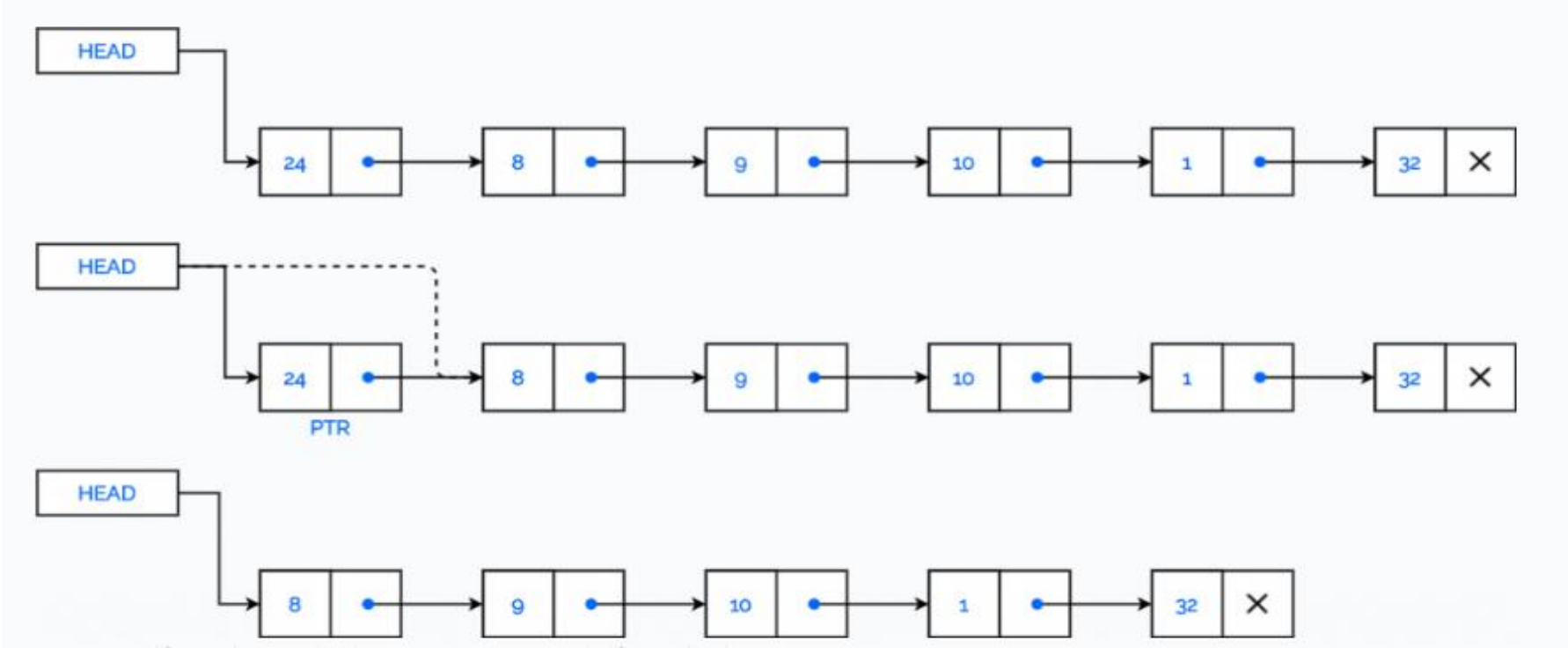


Algorithm: Insert After An Element

- 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 = HEAD
- Step 6: SET PREPTR = PTR
- Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA != NUM
- Step 8: SET PREPTR = PTR
- Step 9: SET PTR = PTR -> NEXT [END OF LOOP]
- Step 10 : PREPTR -> NEXT = NEW_NODE
- Step 11: SET NEW_NODE -> NEXT = PTR
- Step 12: EXIT

Deleting First Elements from a Linked List

- The first node is deleted.
- The last node is deleted.
- The node after a given node is deleted



Algorithm: Delete From Beginning

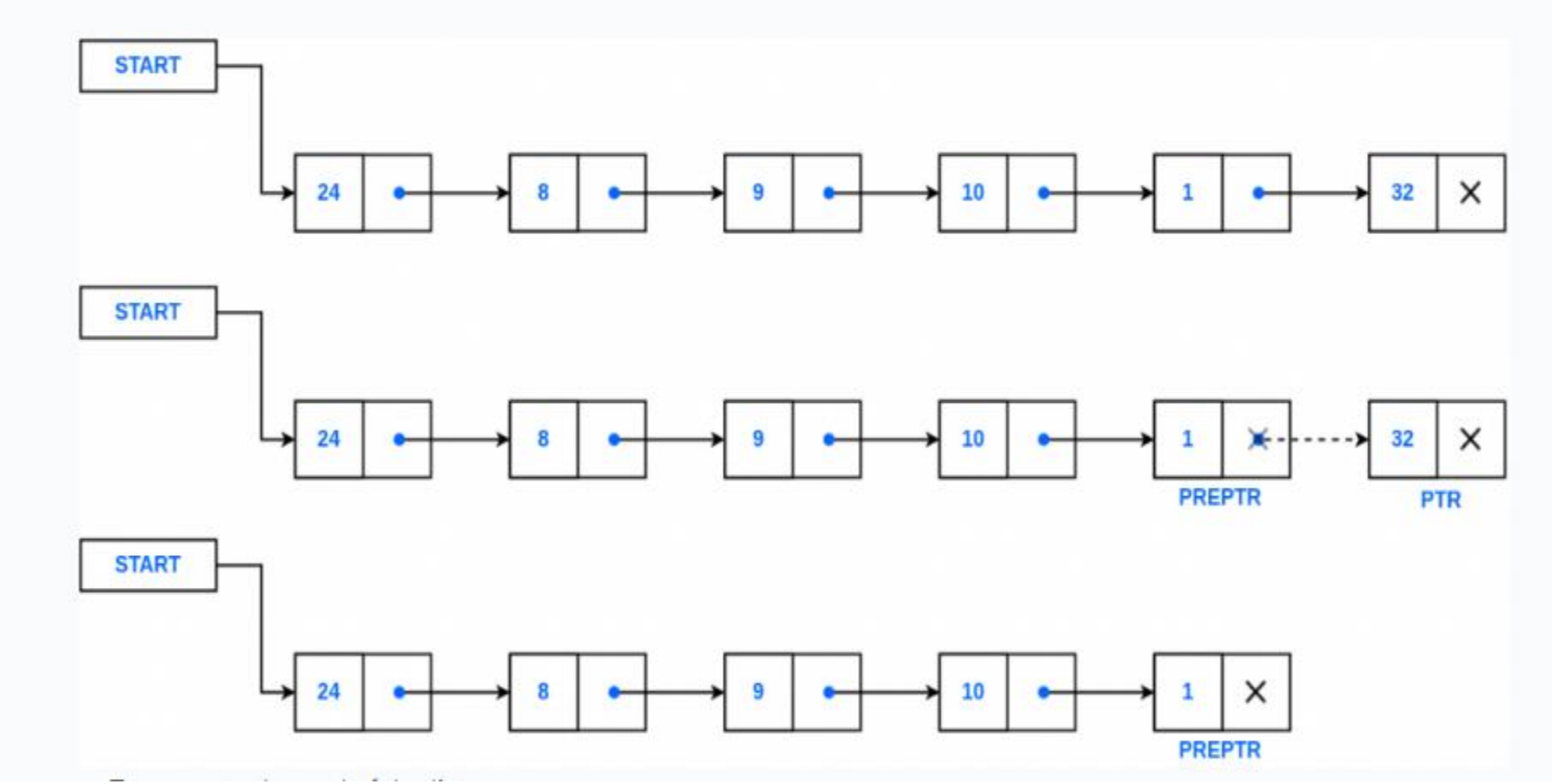
- Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 5 [END OF IF]

- Step 2: SET PTR = HEAD
- Step 3: SET HEAD = HEAD -> NEXT
- Step 4: FREE PTR
- Step 5: EXIT

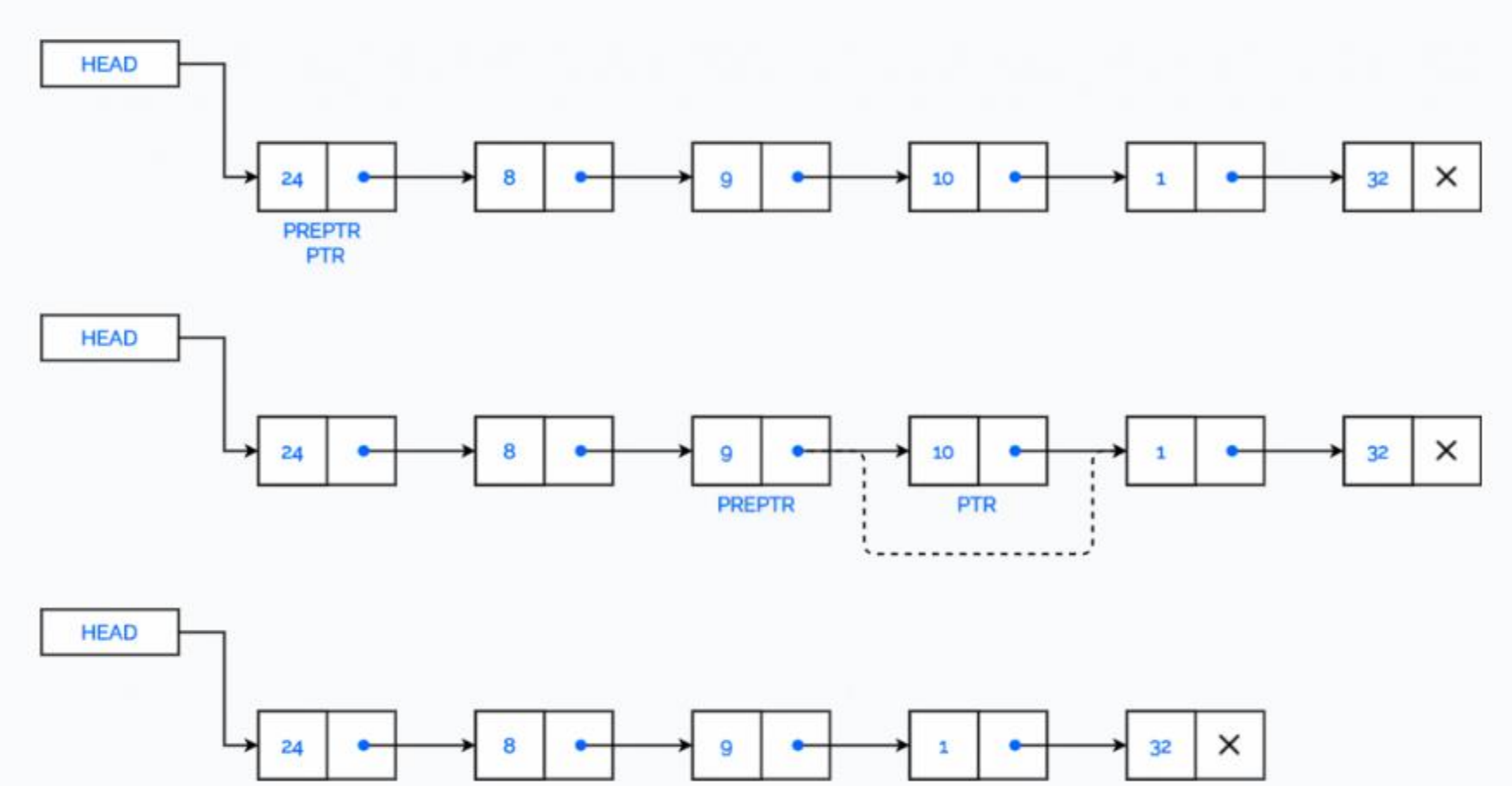
Delete last Node from a Linked list



Algorithm: Delete From End

- Step 1: IF HEAD = NULL Write UNDERFLOW Go to Step 8 [END OF IF]
- Step 2: SET PTR = HEAD
- Step 3: Repeat Steps 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

Delete the Node after a given Node



Algorithm: Delete After A Node

- Step 1: IF HEAD = NULL Write UNDERFLOW Go to Step 10 [END OF IF]
- Step 2: SET PTR = HEAD
- Step 3: SET PREPTR = PTR
- Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
- Step 5: SET PREPTR = PTR
- Step 6: SET PTR = PTR -> NEXT [END OF LOOP]
- Step 7: SET TEMP = PTR
- Step 8: SET PREPTR -> NEXT = PTR -> NEXT
- Step 9: FREE TEMP
- Step 10 : EXIT

Code

```
void del(node *before_del)
{
if (before_del == NULL || before_del->next == NULL)
    return;

node* temp = before_del->next;
before_del->next = temp->next;
delete temp;
}
```

Complete code here

Search

Algorithm:

Step 1: [INITIALIZE] SET PTR = HEAD

Step 2: Repeat Steps 3 and 4 while PTR != NULL

Step 3: If ITEM = PTR -> DATA SET POS = PTR

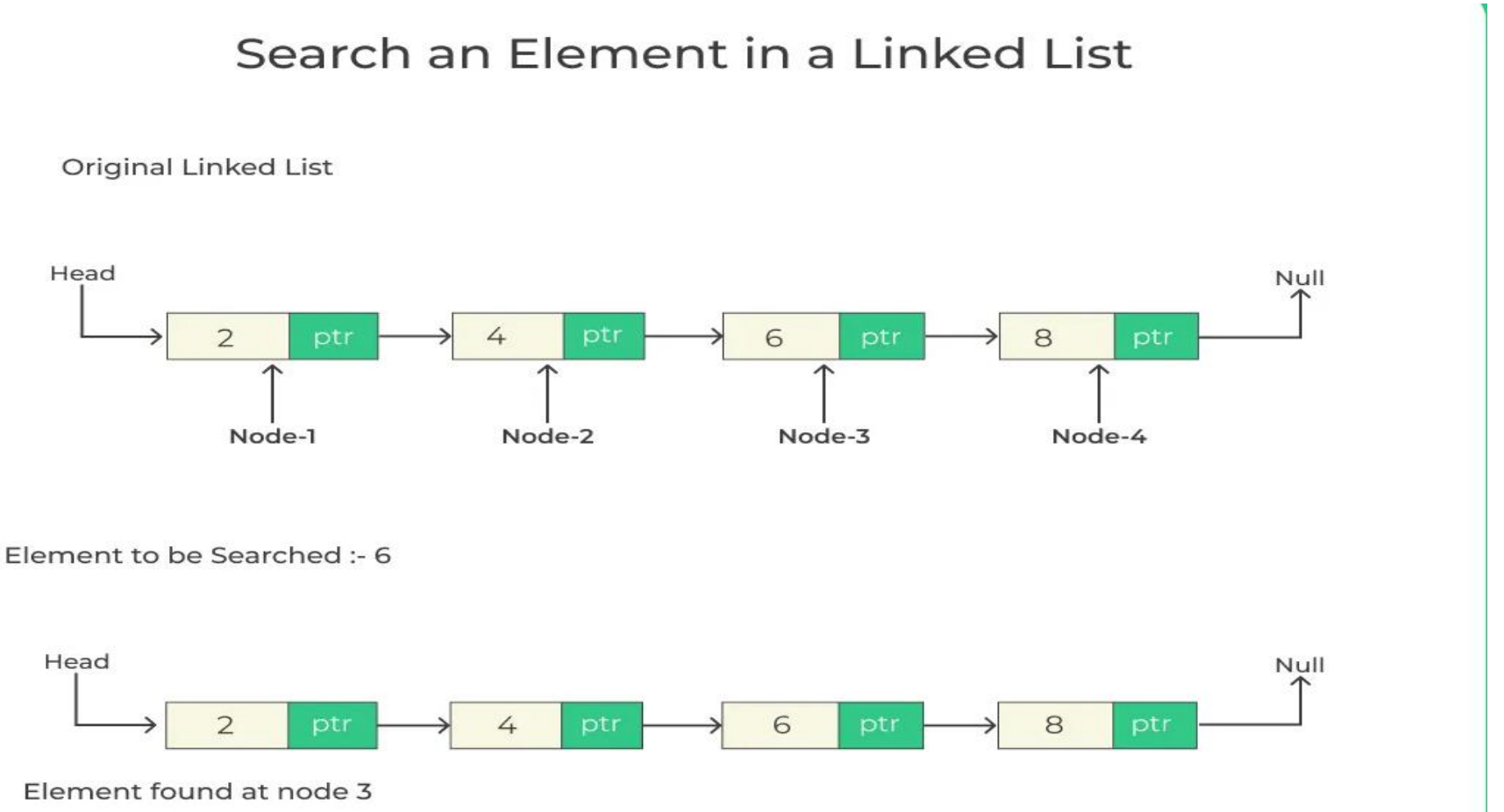
Go To Step 5 ELSE SET PTR = PTR -> NEXT [END

OF IF] [END OF LOOP]

Step 4: SET POS = NULL

Step 5: EXIT

Search


Search an Element in a Linked List
Original Linked List
Head
2
ptr
4
ptr
6
ptr
8
ptr
Null
Node-1
Node-2
Node-3
Node-4
Element to be Searched :- 6
Head
2
ptr
4
ptr
6
ptr
8
ptr
Null
Element found at node 3

Review

Linked List: Definition

Linear data structure composed of nodes.

Each node contains data and a pointer/reference to the next node.

Components of linked list

Nodes: Elements containing data and pointers.

Head pointer: Points to the first node.

Tail pointer (optional): Points to the last node.

Review

Representation

Nodes linked via pointers.

Dynamic memory allocation for nodes.

Head pointer for access.

Advantages and Disadvantages

Dynamic memory allocation.

Efficient insertion/deletion.

Disadvantages:

Higher memory overhead.

No direct access to elements.

Potentially slower access than arrays.


THANK
YOU