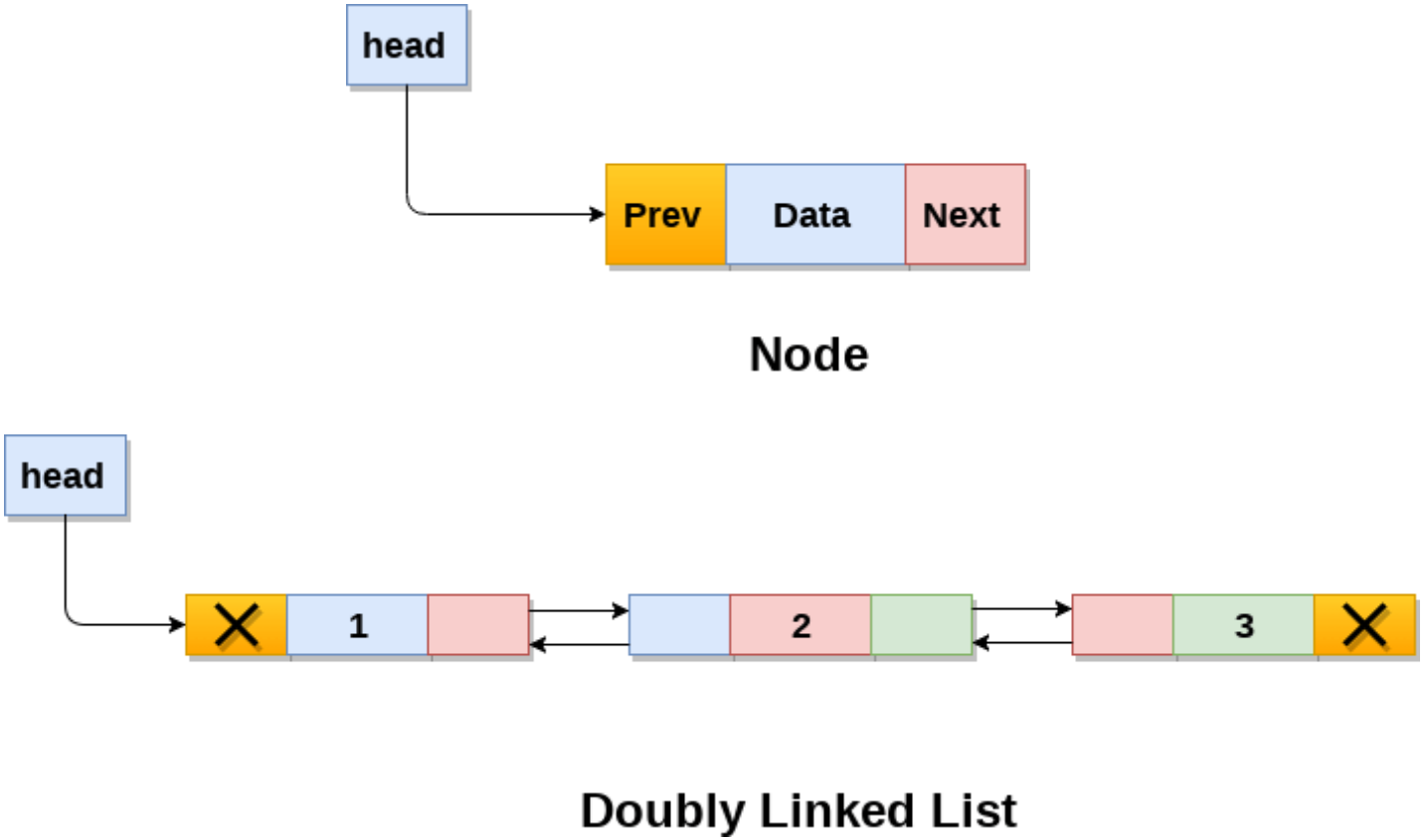
Doubly Linked List

Doubly Linked List

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly linked list, a node consists of three parts:

1. node data,
2. pointer to the next node in sequence (next pointer) ,
3. pointer to the previous node (previous pointer).

Doubly Linked List



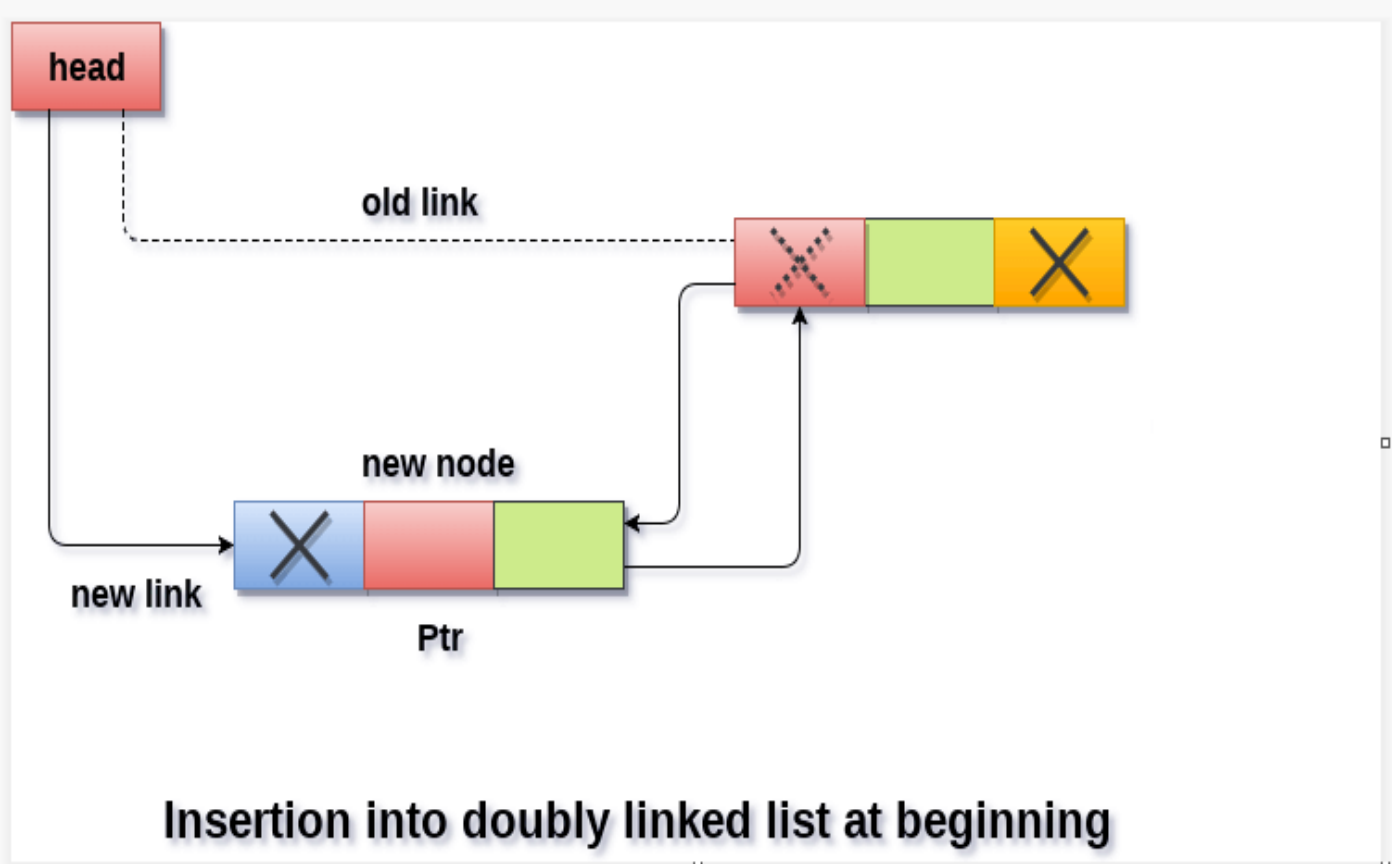
Operations on Doubly Linked List

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

Insertion in doubly linked list at beginning

```
Step 1: IF NEW_NODE  = NULL
            Write OVERFLOW
               Go to Step 7
            [END OF IF]
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: SET NEW_NODE -> PREV = NULL
Step 4: SET NEW_NODE -> NEXT = START
Step 5: SET head -> PREV = NEW_NODE
Step 6: SET head = NEW_NODE
Step 7: EXIT
```

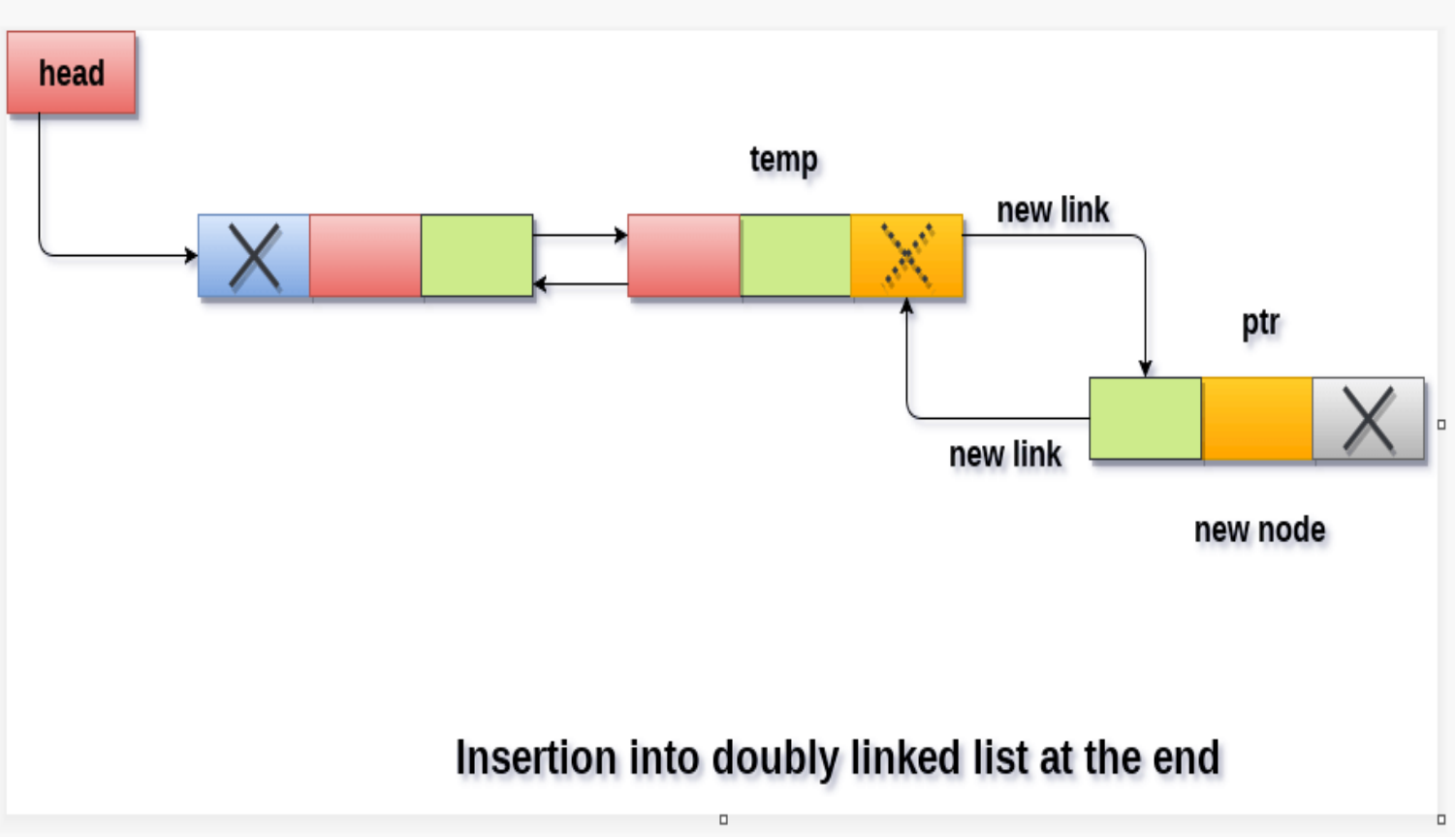
Insertion in doubly linked list at beginning



Insertion in doubly linked list at the end

Step 1: IF NEW_NODE = NULL
Write OVERFLOW
Go to Step 11
[END OF IF]
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: SET NEW_NODE -> NEXT = NULL
Step 4: SET TEMP = START
Step 5: Repeat Step 6 while TEMP -> NEXT != NULL
Step 6: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 7: SET TEMP -> NEXT = NEW_NODE
Step 8: SET NEW_NODE -> PREV = TEMP
Step 9: EXIT

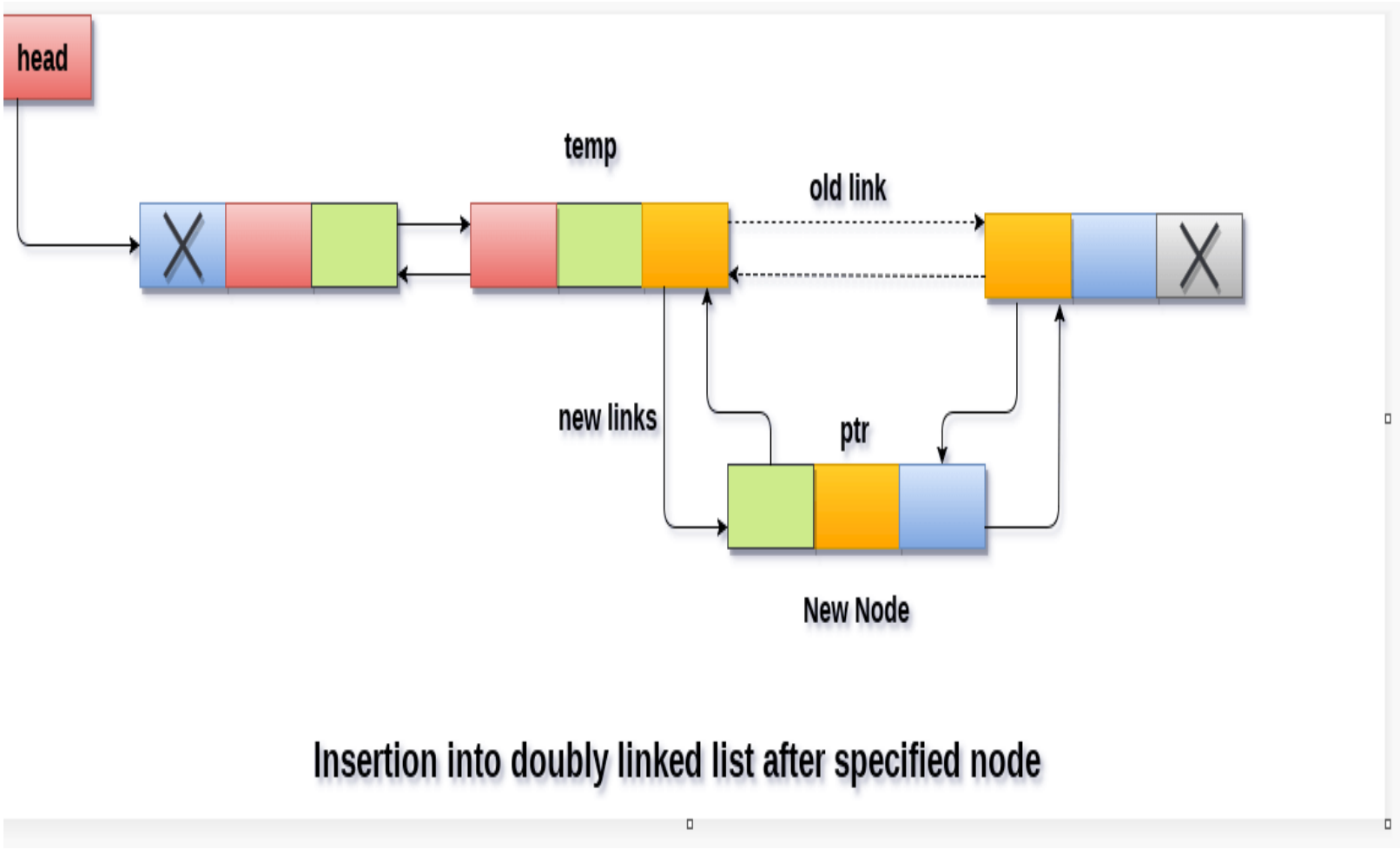
Insertion in doubly linked list at the end



Insertion in doubly linked list after specified Node(Location)

```
Step 1: IF PTR = NULL
          Write OVERFLOW
           Go to Step 13
          [END OF IF]
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: SET TEMP = START
Step 4: SET I = 0
Step 5: REPEAT 6 to 8 until I
Step 6: SET TEMP = TEMP -> NEXT
Step 7: IF TEMP = NULL
Step 8: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
          GOTO STEP 13
          [END OF IF]
          [END OF LOOP]
Step 9: SET NEW_NODE -> NEXT = TEMP -> NEXT
Step 10: SET NEW_NODE -> PREV = TEMP
Step 11 : SET TEMP -> NEXT = NEW_NODE
Step 12: SET TEMP -> NEXT -> PREV = NEW_NODE
Step 13: EXIT
```

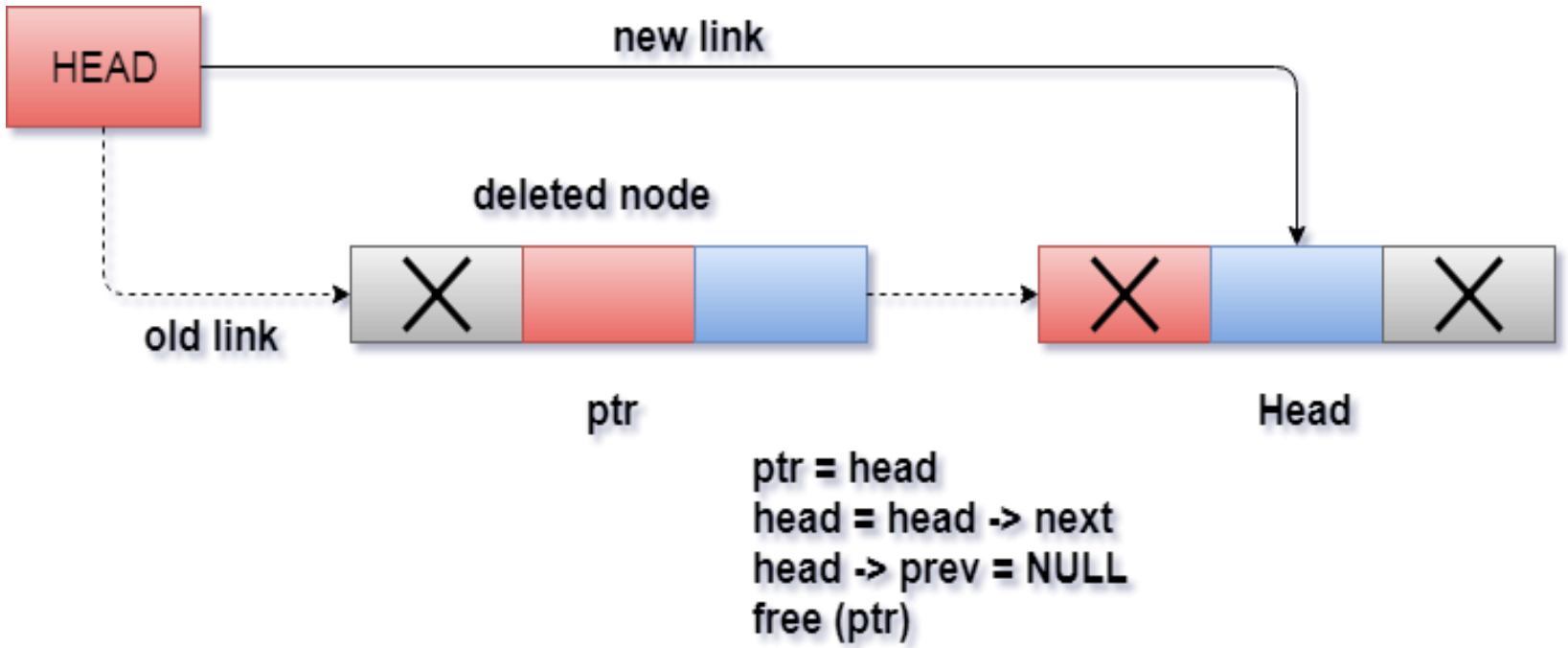
Insertion in doubly linked list after specified Node



Deletion in doubly linked list at beginning

STEP 1: IF HEAD = NULL
WRITE UNDERFLOW
GOTO STEP 6
STEP 2: SET PTR = HEAD
STEP 3: SET HEAD = HEAD → NEXT
STEP 4: SET HEAD → PREV = NULL
STEP 5: FREE PTR
STEP 6: EXIT

Deletion in doublylinked list at beginning



Deletion in doubly linked list from beginning

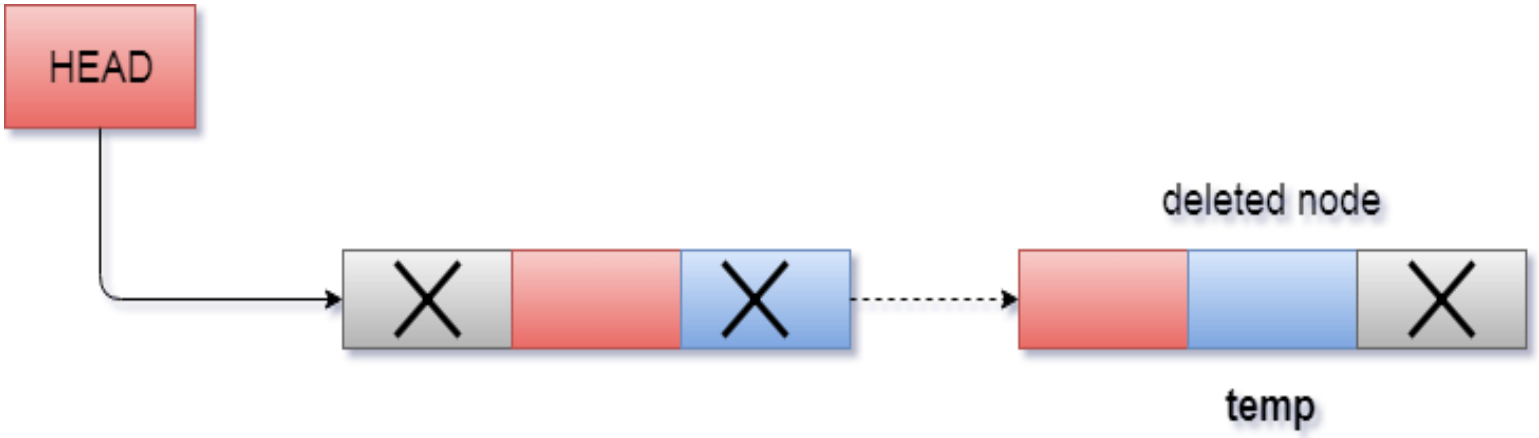
Deletion in doubly linked list at the end

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 7
[END OF IF]

Step 2: SET TEMP = HEAD
Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL
Step 4: SET TEMP = TEMP->NEXT
[END OF LOOP]

Step 5: SET TEMP ->PREV-> NEXT = NULL
Step 6: FREE TEMP
Step 7: EXIT

Deletion in doubly linked list at the end



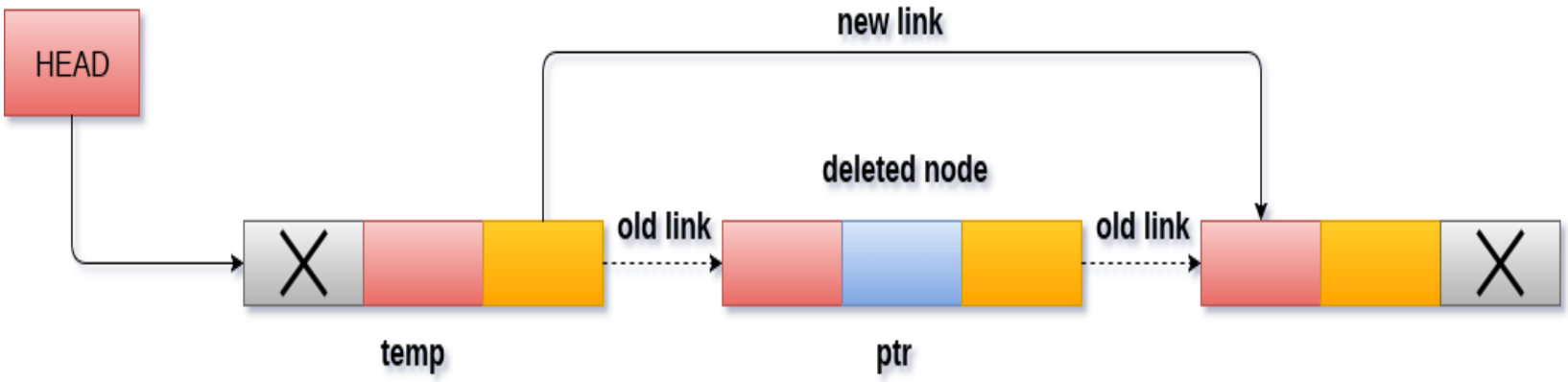
temp->prev->next = NULL
free(temp)

Deletion in doubly linked list at the end

Deletion in doubly linked list after the specified node(Location)

```
Step 1: IF HEAD = NULL
          Write UNDERFLOW
           Go to Step 13
           [END OF IF]
Step 2: SET TEMP = HEAD
Step 3: SET I = 0
Step 5: REPEAT 6 to 8 until I
Step 6: SET TEMP = TEMP -> NEXT
Step 7: IF TEMP = NULL
Step 8: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
          GOTO STEP 13
          [END OF IF]
          [END OF LOOP]
Step 9: SET PTR = TEMP -> NEXT
Step 10: SET TEMP -> NEXT = PTR -> NEXT
Step 11: SET PTR -> NEXT -> PREV = TEMP
Step 12: FREE PTR
Step 13: EXIT
```

Deletion in doubly linked list after the specified node(Location)



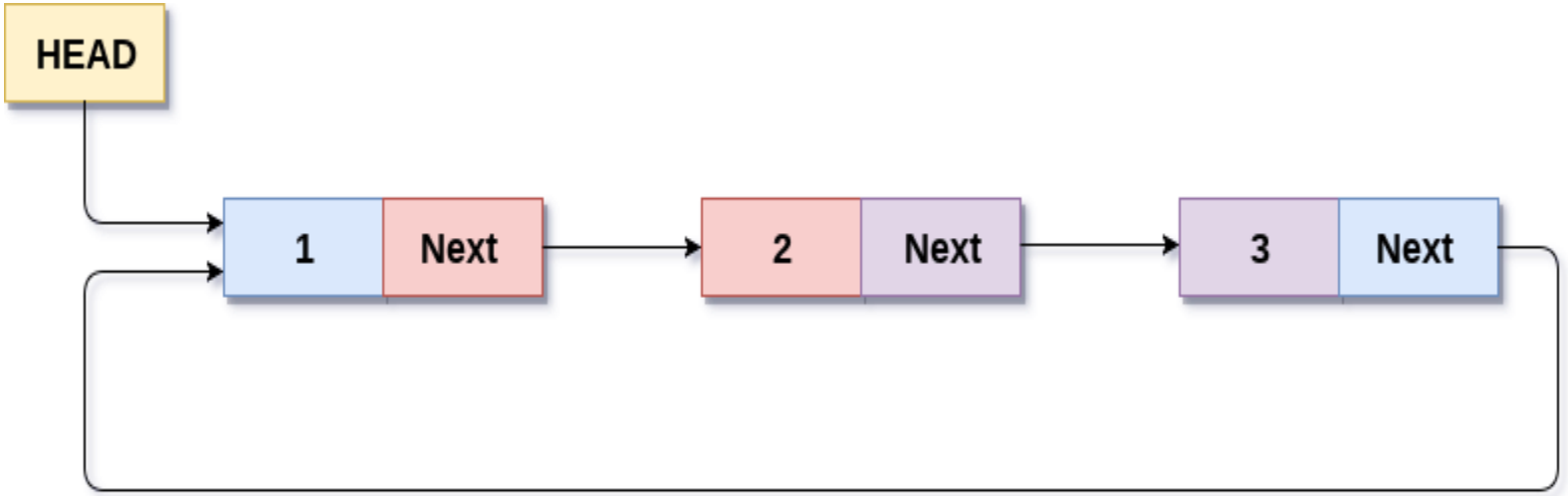
Deletion of a specified node in doubly linked list

Circular Linked List

- There are two types:

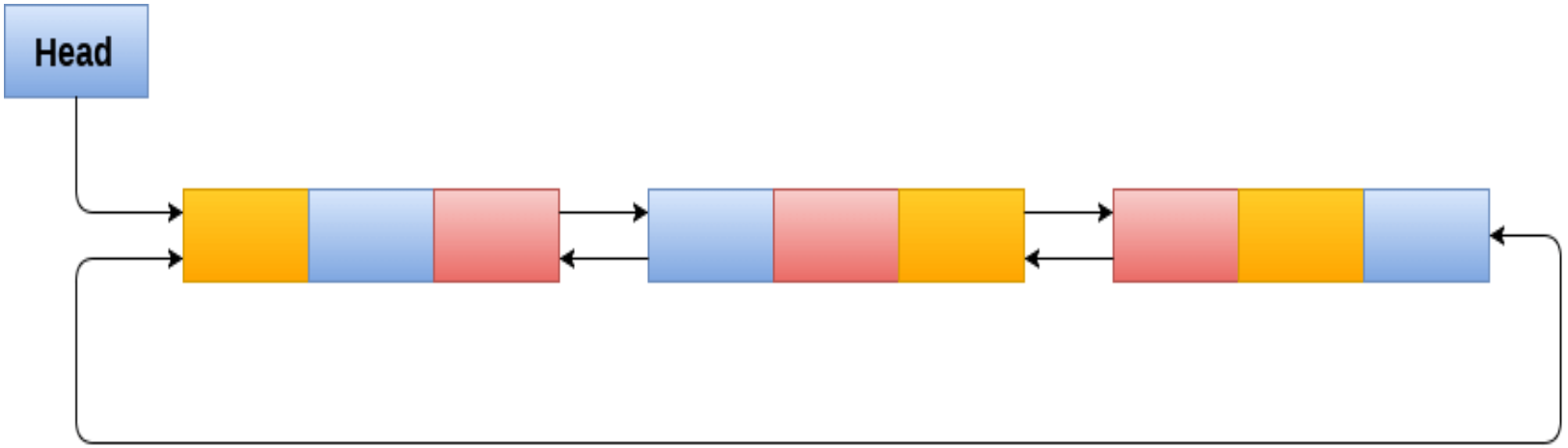
1. Circular Singly Linked List
2. Circular Doubly Linked List

Circular Singly Linked List



Circular Singly Linked List

Circular Doubly Linked List



Circular Doubly Linked List

Applications of Linked List

- Polynomial Manipulation representation
- Addition of long positive integers
- Representation of sparse matrices
- Symbol table creation
- Mailing list
- Memory management
- Linked allocation of files
- Multiple precision arithmetic etc

Polynomial Manipulation representation

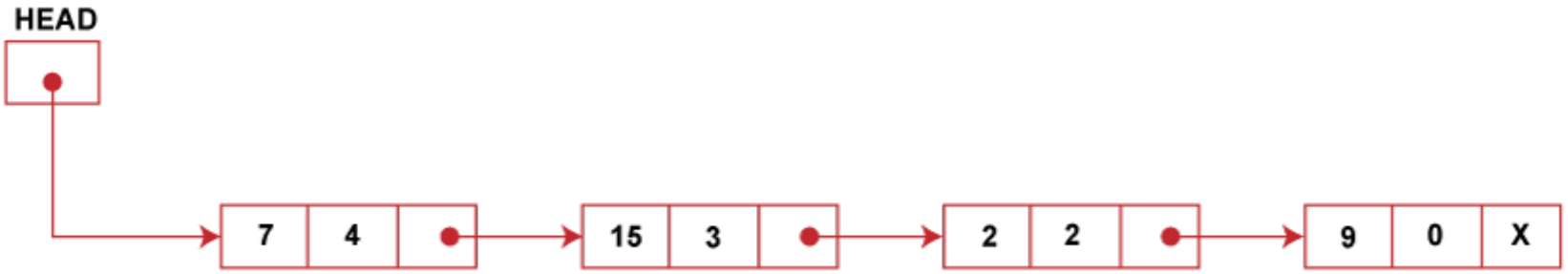
- Each node of a linked list representing polynomial constitute three parts:

1. The first part contains the value of the coefficient of the term.
2. The second part contains the value of the exponent.
3. The third part, LINK points to the next term (next node).

Polynomial Manipulation representation

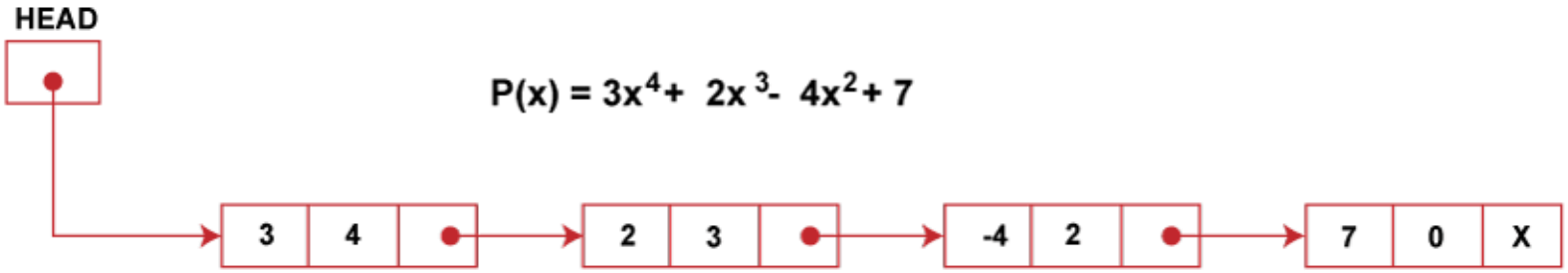
- The structure of a node of a linked list that represents a polynomial is shown below:

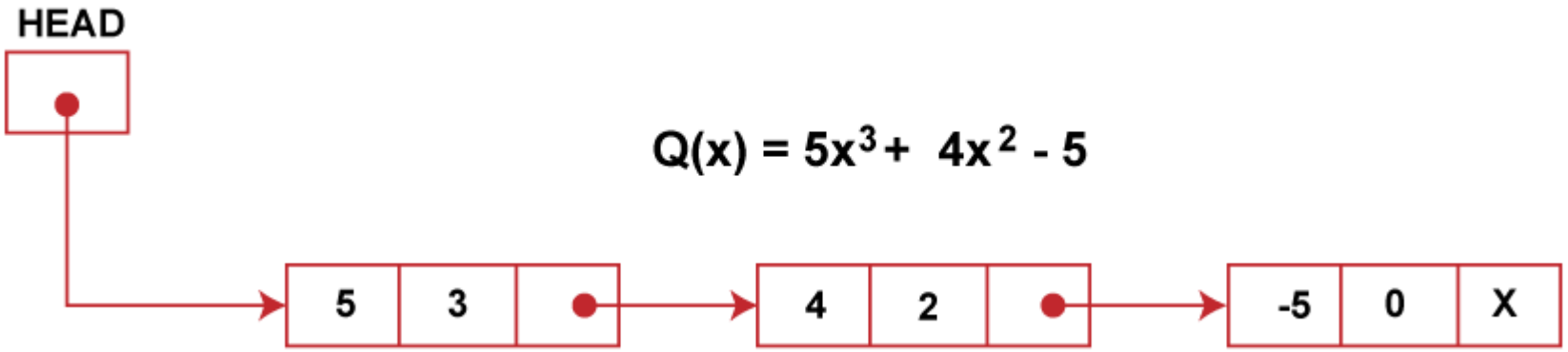
Node representing a term of a polynomial



Linked representation of polynomial P(x)

Polynomial Manipulation : Addition

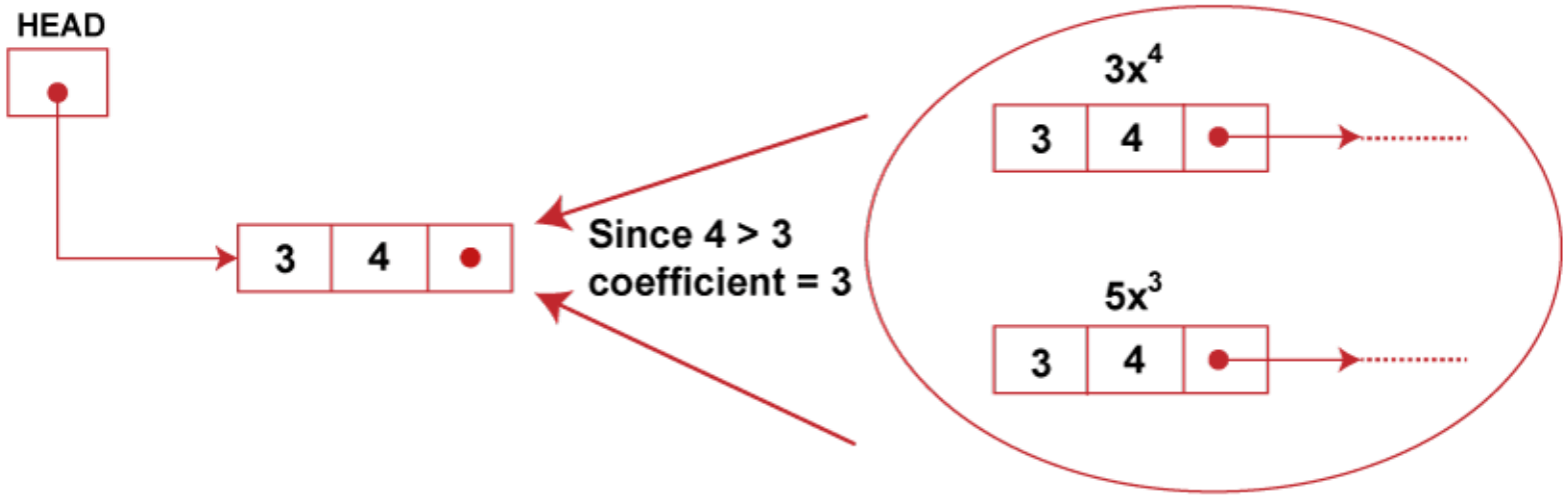




Steps for Polynomial Addition

1. Traverse the two lists P and Q and examine all the nodes.
2. We compare the exponents of the corresponding terms of two polynomials. The first term of polynomials P and Q contain exponents 4 and 3, respectively. Since the exponent of the first term of the polynomial P is greater than the other polynomial Q, the term having a larger exponent is inserted into the new list. The new list initially looks as shown below:

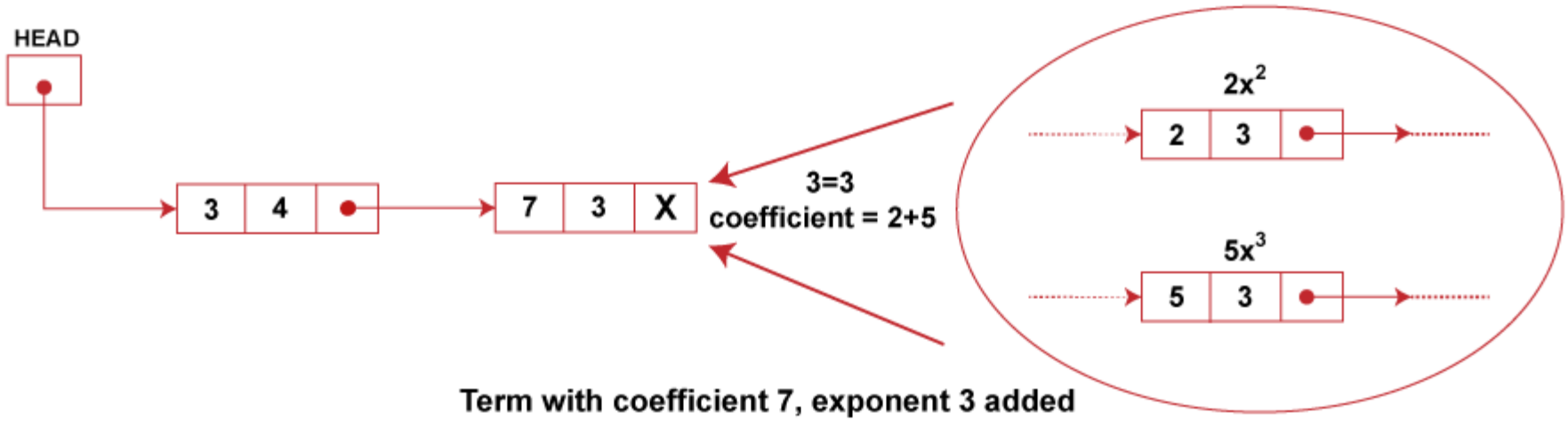
Steps for Polynomial Addition



Term with coefficient 3, exponent 4 added

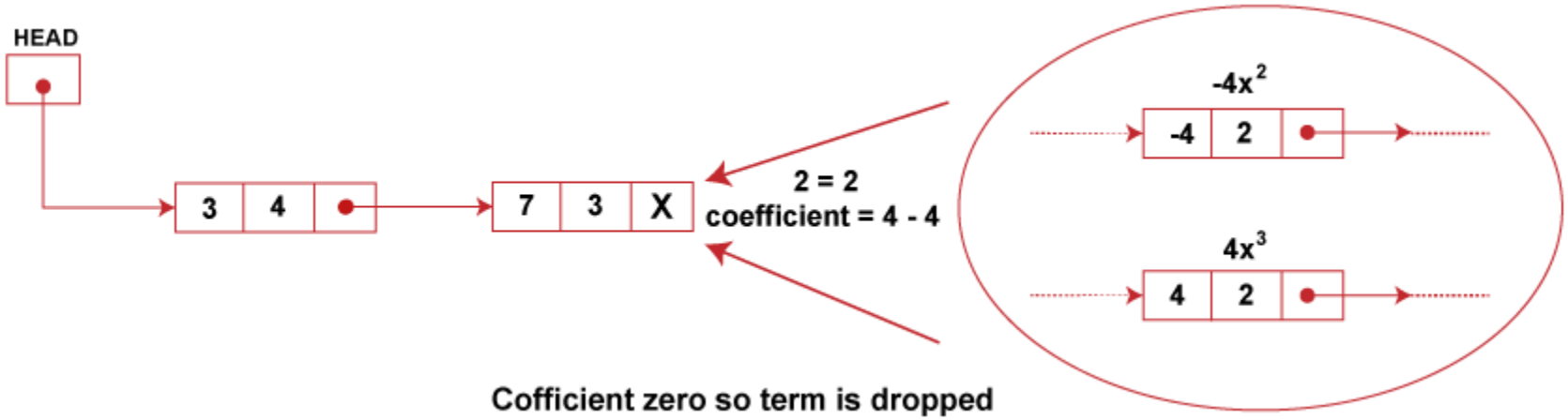
Steps for Polynomial Addition

3. We then compare the exponent of the next term of the list P with the exponents of the present term of list Q. Since the two exponents are equal, so their coefficients are added and appended to the new list as follows:



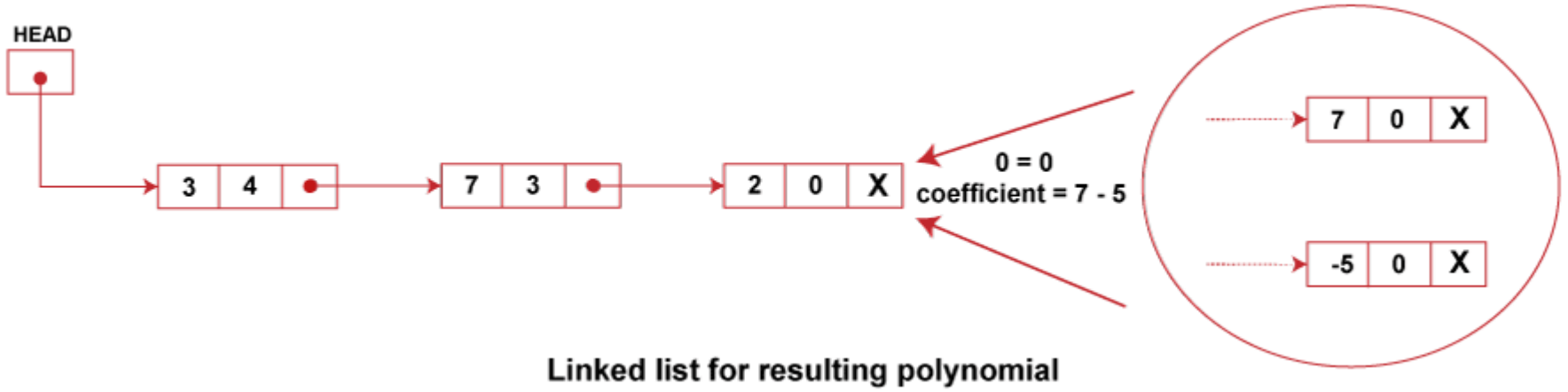
Steps for Polynomial Addition

4. Then we move to the next term of P and Q lists and compare their exponents. Since exponents of both these terms are equal and after addition of their coefficients, we get 0, so the term is dropped, and no node is appended to the new list after this,



Steps for Polynomial Addition

5. Moving to the next term of the two lists, P and Q, we find that the corresponding terms have the same exponents equal to 0. We add their coefficients and append them to the new list for the resulting polynomial as shown below:



Linked list for resulting polynomial

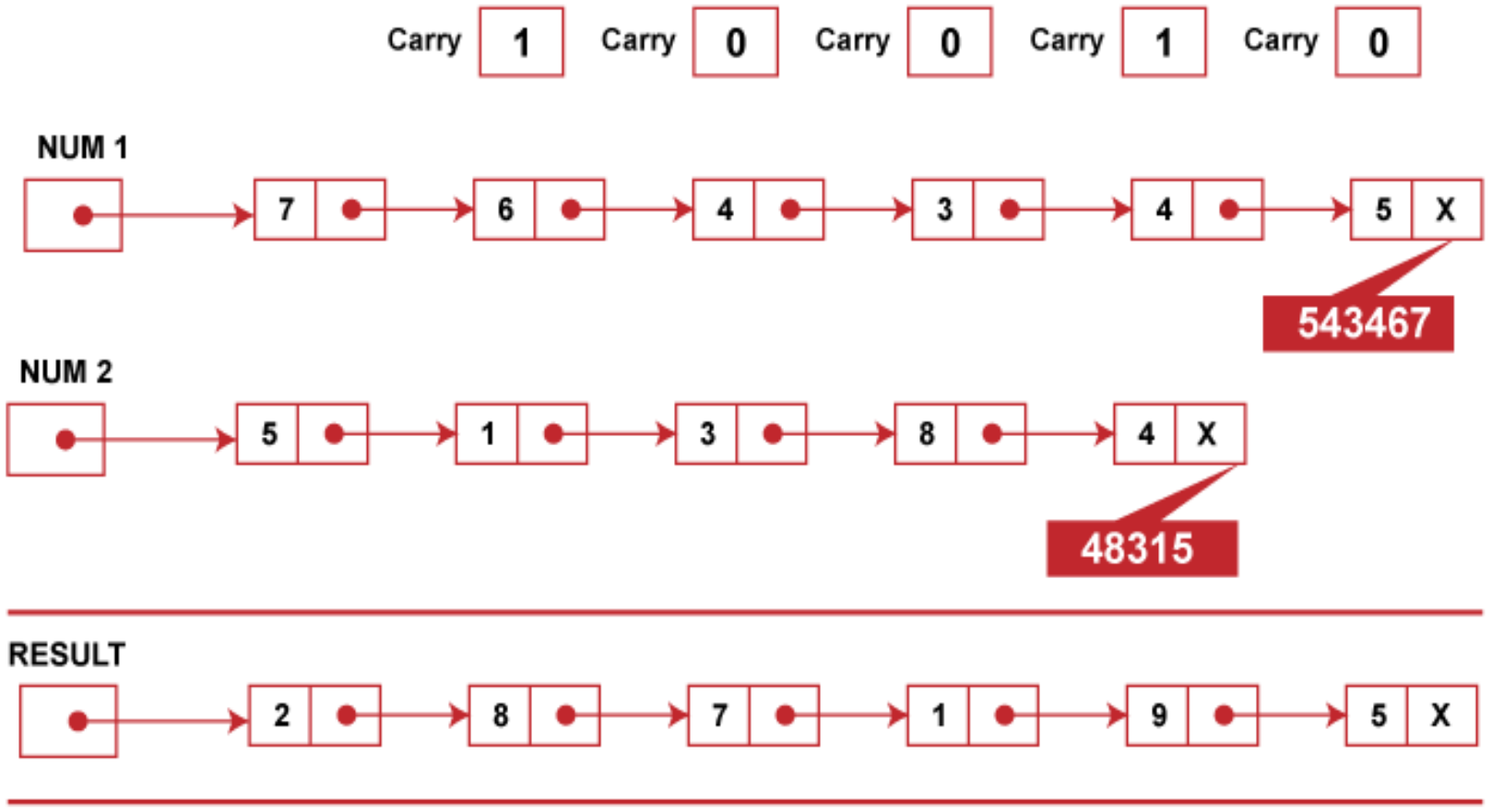
Addition of long positive integer using linked list

- Most programming languages allow restrictions on the maximum value of integers stored. The maximum value of the largest integers is 32767, and the largest is 2147483647.
- Sometimes, applications such as security algorithms and cryptography require storing and manipulating integers of unlimited size. So in such a situation, it is desirable to use a linked list for storing and manipulating integers of arbitrary length.

Addition of long positive integer

1. Traverse the two linked lists in parallel from left to right.
2. During traversal, corresponding digits and a carry from prior digits sum are added, then stored in the new node of the resultant linked list.

Addition of long positive integer



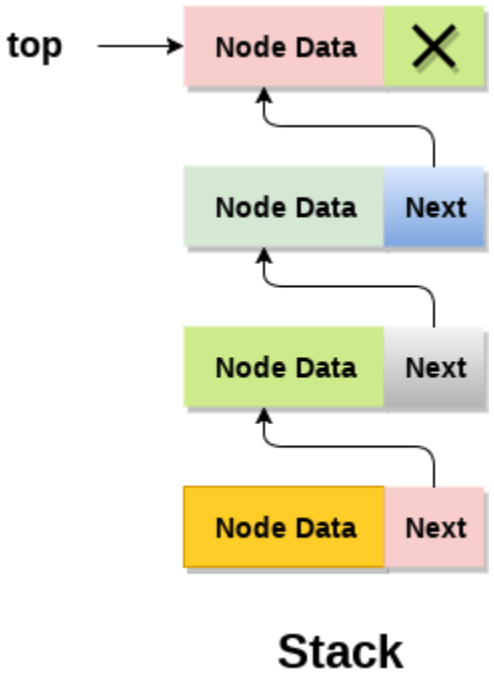
Other Applications of Linked List

- **Memory Management:** Memory management is one of the operating system's key features. It decides how to allocate and reclaim storage for processes running on the system. We can use a linked list to keep track of portions of memory available for allocation.
- **Mailing List:** Linked lists have their use in email applications. Since it is difficult to predict multiple lists, maybe a mailer builds a linked list of addresses before sending a message.

Other Applications of Linked List

- **Linked allocation of files:** A file of large size may not be stored in one place on a disk. So there must be some mechanism to link all the scattered parts of the file together. The use of a linked list allows an efficient file allocation method in which each block of a file contains a pointer to the file's text block. But this method is good only for sequential access.
- **Virtual Memory:** An interesting application of linked lists is found in the way systems support virtual memory.
- **Support for other data structures:** Some other data structures like stacks, queues, hash tables, graphs can be implemented using a linked list.

Stack using linked list



THANK YOU