Linear Data Structures-List

- List as an ADT, Array-based implementation, Linked List implementation, singly linked lists, circularly linked lists, doubly-linked lists, All operations (Insertion, Deletion, Merge, Traversal, etc.) and their analysis, Applications of lists.

Why there is a need for a linked list.

- If we want to store the value in a memory, we need a memory manager that manages the memory for every variable. For example, if we want to create a variable of integer type like:

- int x;

- In the above example, we have created a variable 'x' of type integer. As we know that integer variable occupies 2 bytes, so 'x' variable will occupy 2 bytes to store the value.
- Suppose we want to create an array of integer type like:

-int x[3];

- In the above example, we have declared an array of size 3. As we know, that all the values of an array are stored in a continuous manner, so all the three values of an array are stored in a sequential fashion. The total memory space occupied by the array would be **3*2 = 6 bytes**.

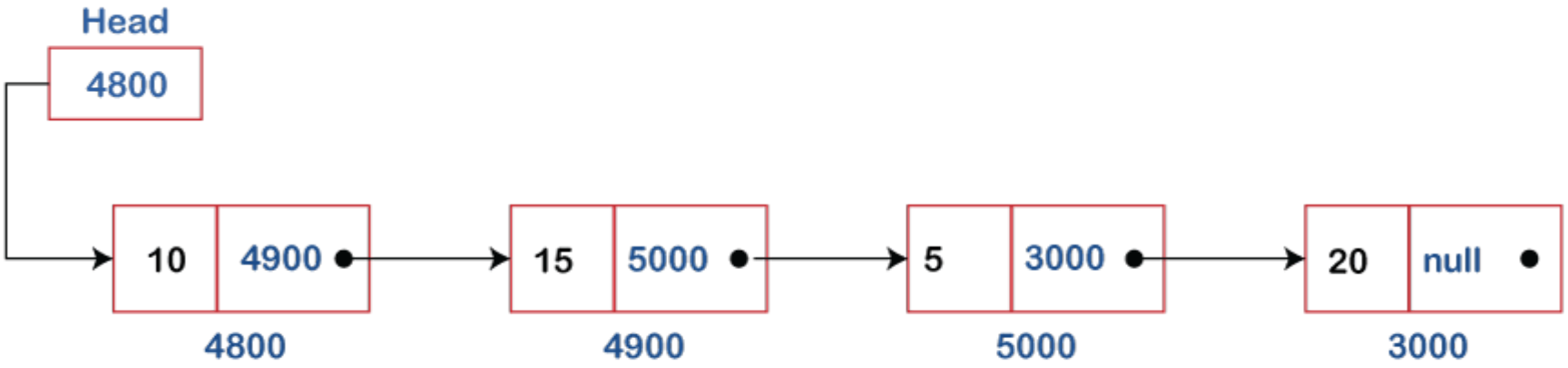
There are two major drawbacks of using array

- We cannot insert more than 3 elements in the above example because only 3 spaces are allocated for 3 elements.
- In the case of an array, lots of wastage of memory can occur. For example, if we declare an array of 50 size but we insert only 10 elements in an array. So, in this case, the memory space for other 40 elements will get wasted and cannot be used by another variable as this whole space is occupied by an array.
- In array, we are providing the fixed-size at the compile-time, due to which wastage of memory occurs. The solution to this problem is to use the **linked list**.

What is Linked List?

- A linked list is also a collection of elements, but the elements are not stored in a consecutive location.
- Suppose a programmer made a request for storing the integer value then size of 2-byte memory block is assigned to the integer value. The programmer made another request for storing 3 more integer elements; then, three different memory blocks are assigned to these three elements but the memory blocks are available in a random location. So, how are the elements connected?.
- These elements are linked to each other by providing one additional information along with an element, i.e., the address of the next element. The variable that stores the address of the next element is known as a pointer. Therefore, we conclude that the linked list contains two parts, i.e., the first one is **the data element**, and the other is the **pointer**. The pointer variable will occupy 2 bytes which is pointing to the next element.

- ***A linked list can also be defined as the collection of the nodes in which one node is connected to another node, and node consists of two parts, i.e., one is the data part and the second one is the address part, as shown in the below figure:***



How can we declare the Linked list?

- The declaration of an array is very simple as it is of single type.
- But the linked list contains two parts, which are of two different types, i.e., one is a simple variable, and the second one is a pointer variable.
- We can declare the linked list by using the user-defined data type known as structure.

The structure of a linked list can be defined as:

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

In the above declaration, we have defined a structure named as ***a node*** consisting of two variables: an integer variable (data), and the other one is the pointer (next), which contains the address of the next node.

Advantages of using a Linked list over Array

- **Dynamic data structure:**
  The size of the linked list is not fixed as it can vary according to our requirements.
- **Insertion and Deletion:**
  Insertion and deletion in linked list are easier than array as the elements in an array are stored in a consecutive location. In contrast, in the case of a linked list, the elements are stored in a random location. The complexity for insertion and deletion of elements from the beginning is O(1) in the linked list, while in the case of an array, the complexity would be O(n). If we want to insert or delete the element in an array, then we need to shift the elements for creating the space. On the other hand, in the linked list, we do not have to shift the elements. In the linked list, we just need to update the address of the pointer in the node.
- **Memory efficient**
  Its memory consumption is efficient as the size of the linked list can grow or shrink according to our requirements.
- **Implementation**
  Both the stacks and queues can be implemented using a linked list.

Disadvantages of Linked list

- **Memory usage**
  The node in a linked list occupies more memory than array as each node occupies two types of variables, i.e., one is a simple variable, and another is a pointer variable that occupies 4 bytes in the memory.
- **Traversal**
  In a linked list, the traversal is not easy. If we want to access the element in a linked list, we cannot access the element randomly, but in the case of an array, we can randomly access the element by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- **Reverse traversing**
  In a linked list, backtracking or reverse traversing is difficult. In a doubly linked list, it is easier but requires more memory to store the back pointer.

Applications of Linked List

- The various operations like student's details, employee's details or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Stack, Queue, tree and various other data structures can be implemented using a linked list.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- To implement hashing, we require hash tables. The hash table contains entries that are implemented using linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Types of Linked list

1. Singly Linked list
2. Doubly Linked list
3. Circular Linked list
4. Doubly Circular Linked list

1. Singly Linked list

- Singly linked list can be defined as the collection of ordered set of elements.
- The number of elements may vary according to need of the program.
- A node in the singly linked list consist of two parts: data part and link part.
- Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.
- One way chain or singly linked list can be traversed only in one direction.
- In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

-Node Creation

```
struct node
{
    int data;
    struct node *next;
};
Struct node *head=NULL
```

Operations on Singly Linked List

-Node Insertion

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

Operations on Singly Linked List

```
void insertAtBeginning(int value)
{ struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
if(head == NULL)
{
newNode->next = NULL;
head = newNode;
}
 else
{ newNode->next = head;
head = newNode; }
```

Insertion

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

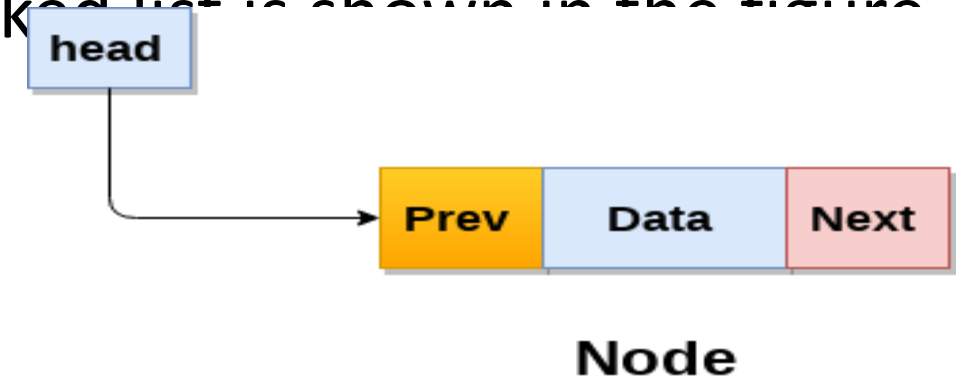
SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned

Head

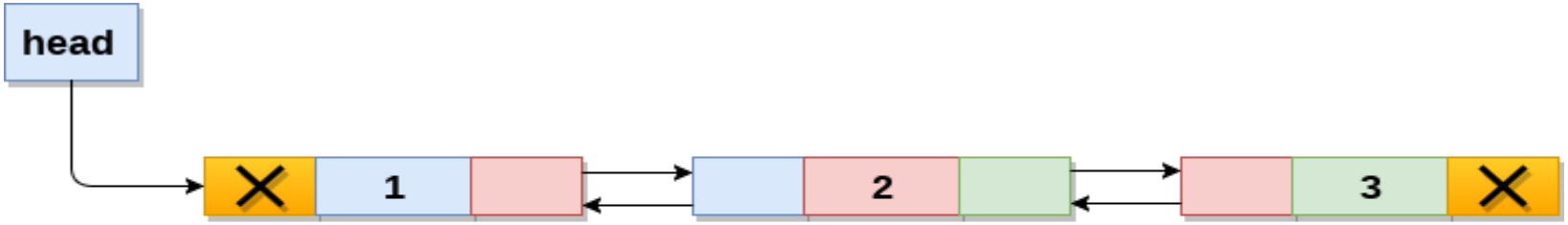
The first node is called the head; it points to the first node of the list and helps us access every other element in the list. The last node, also sometimes called the tail, points to NULL which helps us in determining when the list ends.

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: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

```
1.  struct node
2.  {
3.      struct node
    *prev;
4.      int data;
5.      struct node
    *next;
```

The prev part of the first node and the next part of the last node will always contain null indicating end in each direction.

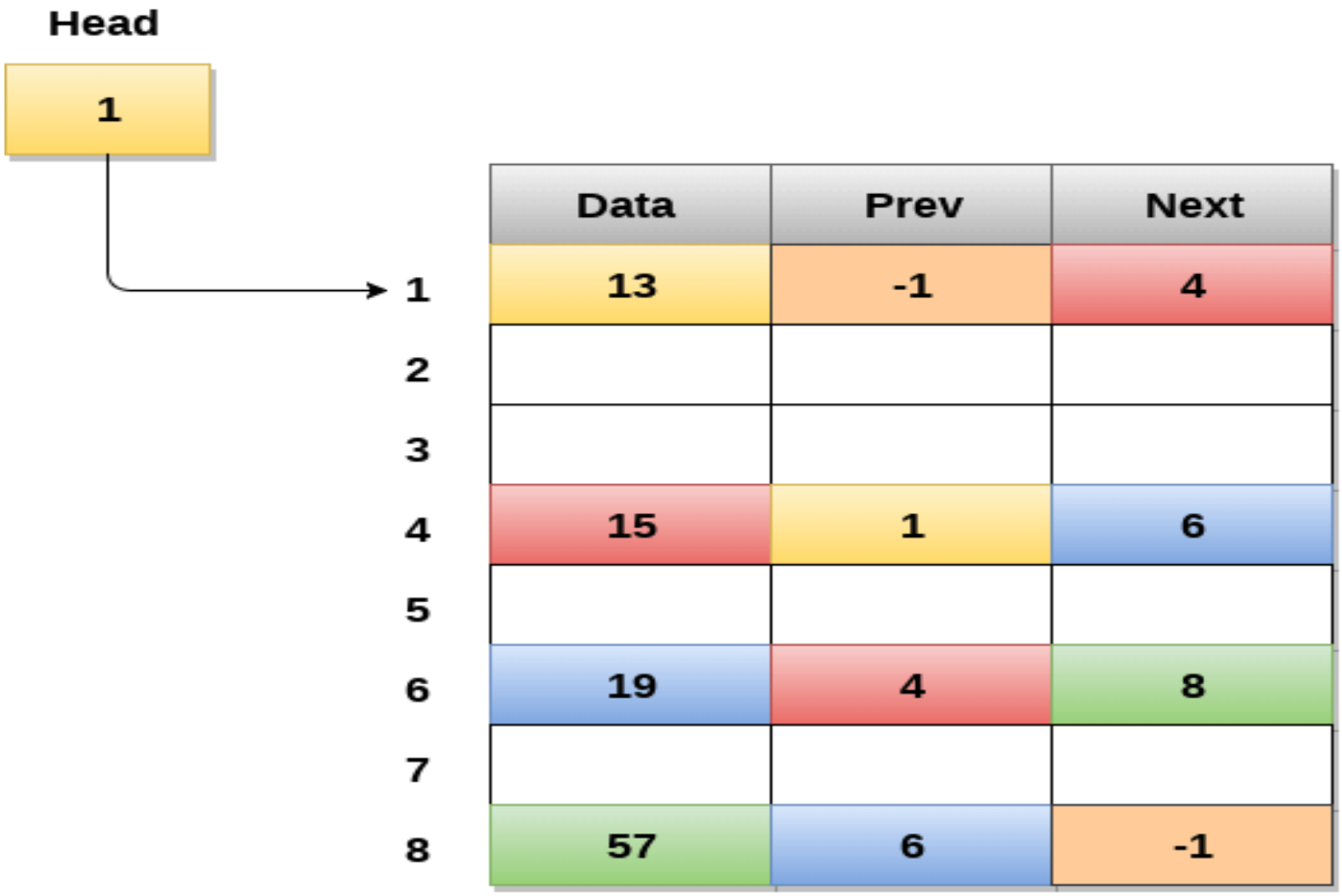
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev**of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

Operations on doubly linked list

Node Creation

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

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some

Insertion in doubly linked list at beginning

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

Allocate the space for the new node in the memory. This will be done by using the following statement.

*ptr = (struct node *)malloc(sizeof(struct node));*

Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

```
ptr->next = NULL;
        ptr->prev=NULL;
        ptr->data=item;
        head=ptr;
```

In the second scenario, the condition head == NULL become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.

This will be done by using the following statements.

```
ptr->next = head;
  head → prev=ptr;
```

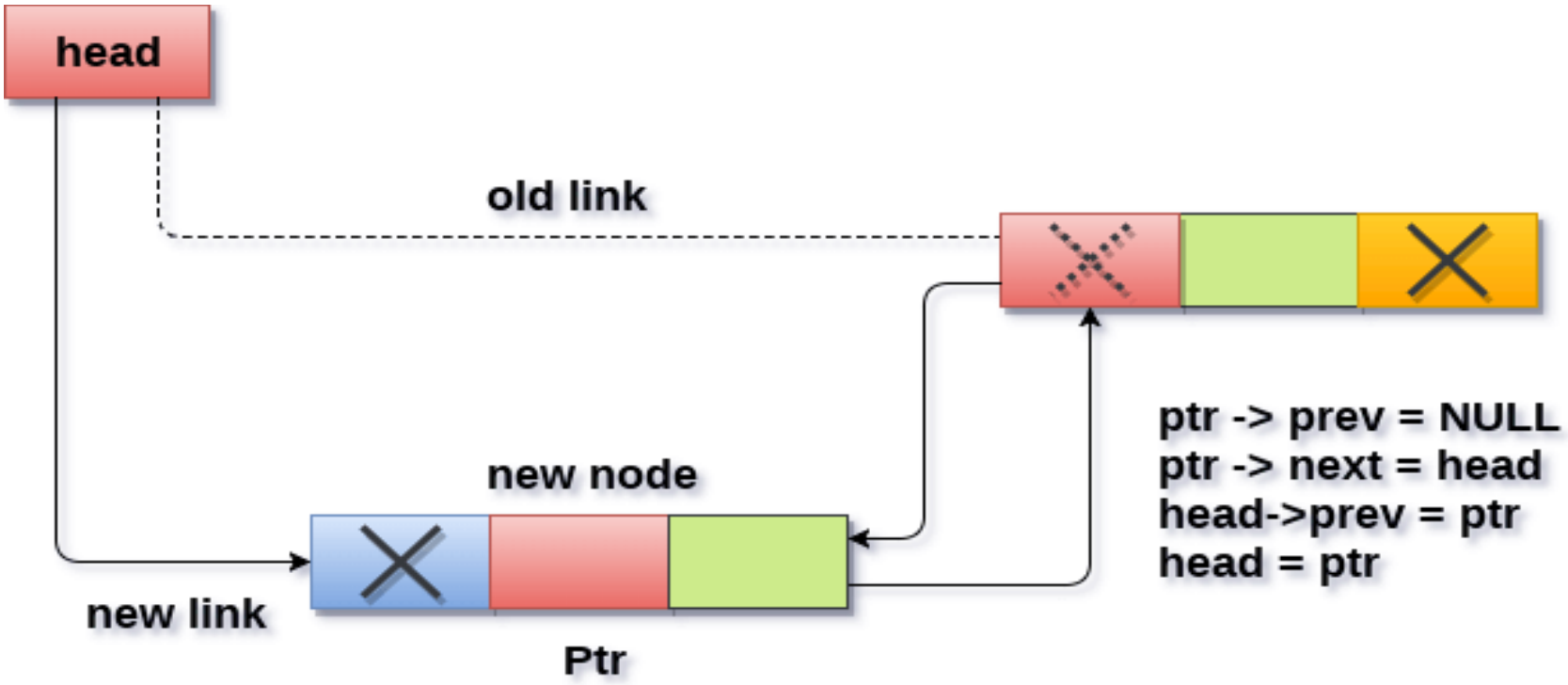
Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

ptr → prev =NULL

head = ptr

Algorithm :

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



Insertion into doubly linked list at beginning

Insertion in doubly linked list after Specified node

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.

Use the following steps for this purpose.

Allocate the memory for the new node. Use the following statements for this.

*ptr = (struct node *)malloc(sizeof(struct node));*

Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.

```
temp=head;
   for(i=0;i<loc;i++)
  {
      temp = temp->next;
      if(temp == NULL) // the temp will be //null if the list doesn't last
long //up to mentioned location
      {
           return;
      }
  }
```

The temp would point to the specified node at the end of the for loop. The new node needs to be inserted after this node therefore we need to make a fer pointer adjustments here. Make the next pointer of ptr point to the next node of temp.

ptr → next = temp → next;

make the prev of the new node ptr point to temp.

ptr → prev = temp;

make the next pointer of temp point to the new node ptr.

temp → next = ptr;

make the previous pointer of the next node of temp point to the new node.

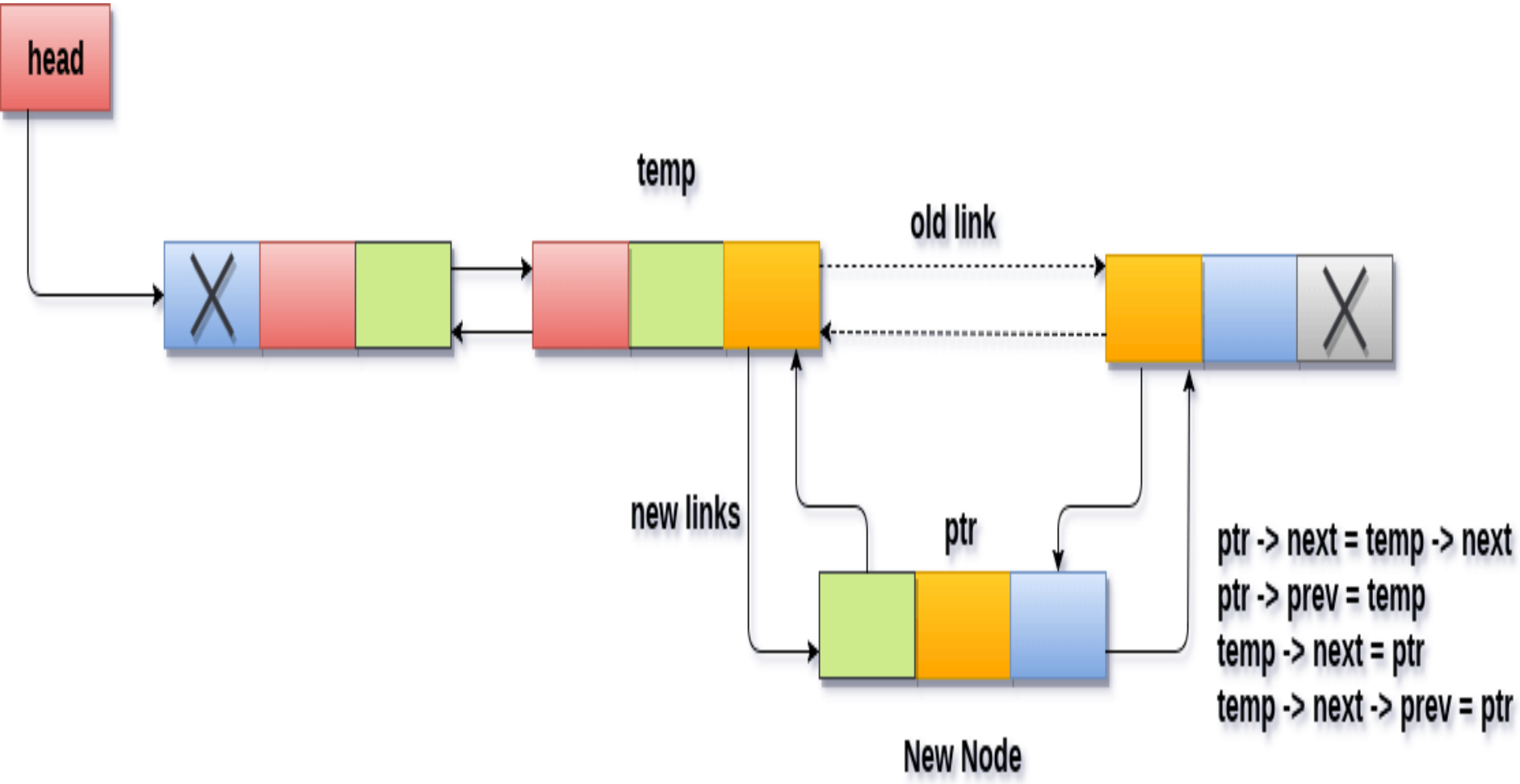
temp → next → prev = ptr;

Algorithm

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 15
[END OF IF]

Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET TEMP = START
Step 6: SET I = 0
Step 7: REPEAT 8 to 10 until I
Step 8: SET TEMP = TEMP -> NEXT
STEP 9: IF TEMP = NULL
STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
GOTO STEP 15
[END OF IF]
[END OF LOOP]

Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT
Step 12: SET NEW_NODE -> PREV = TEMP
Step 13 : SET TEMP -> NEXT = NEW_NODE
Step 14: SET TEMP -> NEXT -> PREV = NEW_NODE
Step 15: EXIT



Insertion into doubly linked list after specified node

Deletion at beginning

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

Ptr = head;

head = head → next;

now make the prev of this new head node point to NULL. This will be done by using the following statements.

head → prev = NULL

Now free the pointer ptr by using the free function.

f*ree(ptr)*

Algorithm

STEP 1: IF HEAD = NULL

WRITE UNDERFLOW

GOTO STEP 6

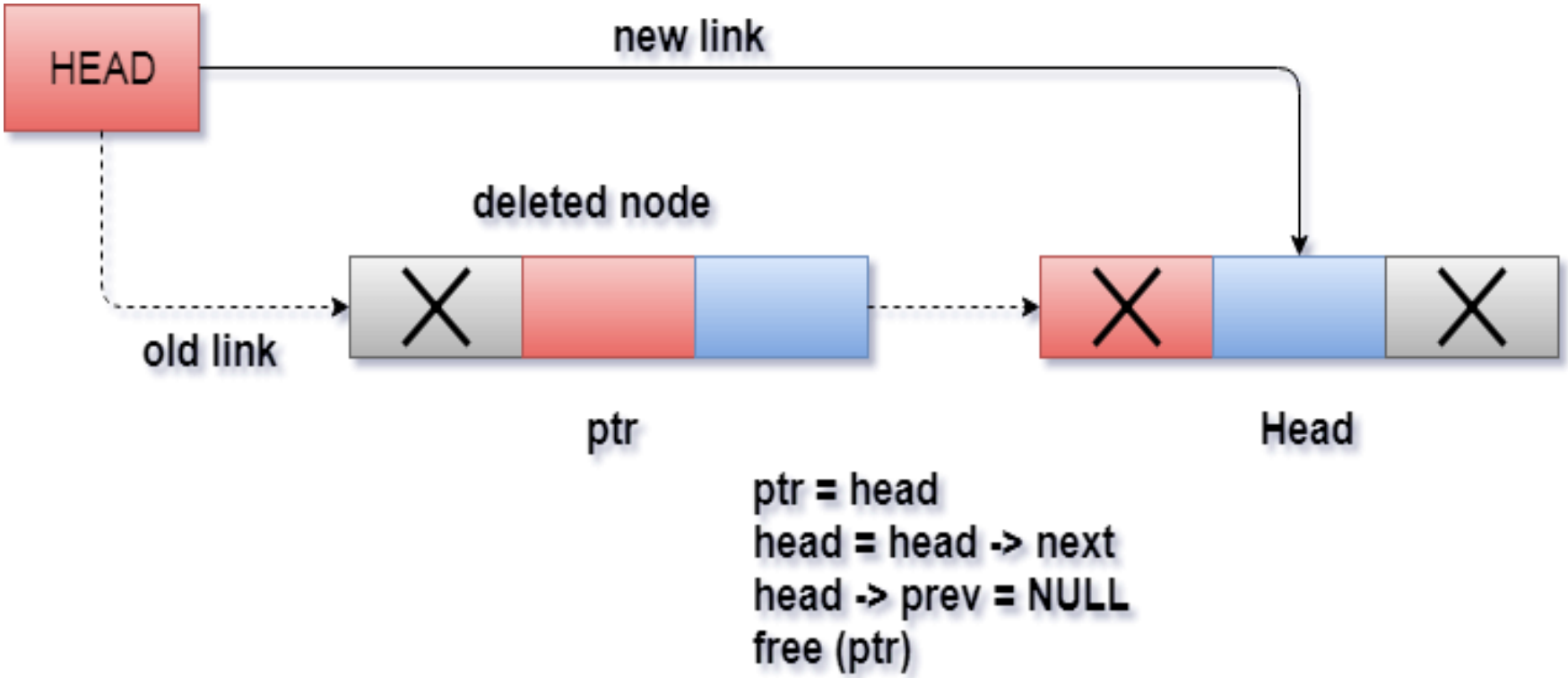
STEP 2: SET PTR = HEAD

STEP 3: SET HEAD = HEAD → NEXT

STEP 4: SET HEAD → PREV = NULL

STEP 5: FREE PTR

STEP 6: EXIT



Deletion in doubly linked list from beginning

Deletion in doubly linked list at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.

If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.

Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
        if(ptr->next != NULL)
        {
            ptr = ptr -> next;
        }
```

The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of ptr to NULL.

ptr → prev → next = NULL

free the pointer as this the node which is to be deleted.

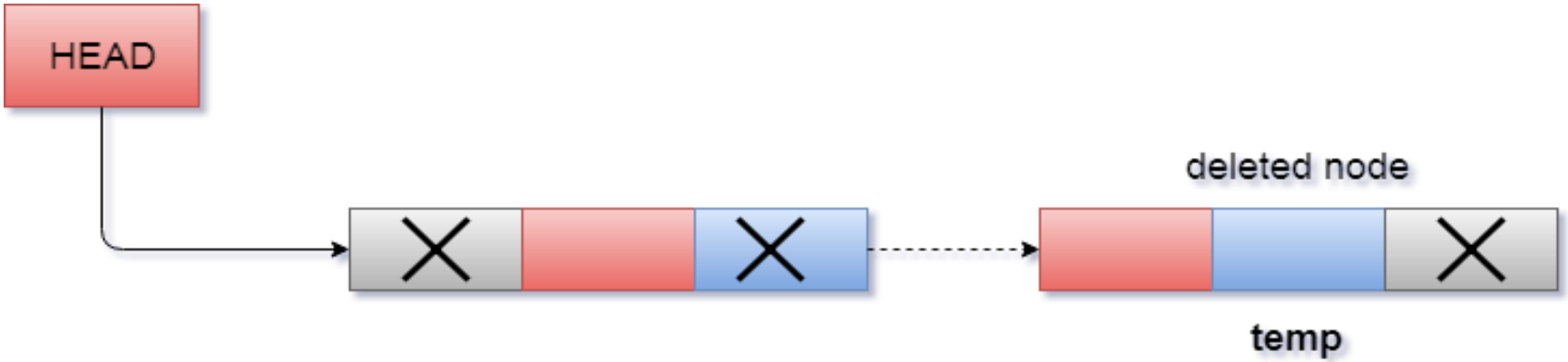
free(ptr)

Algorithm

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



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

Deletion in doubly linked list at the end

Deletion in doubly linked list after the specified node

In order to delete the node after the specified data, we need to perform the following steps.

Copy the head pointer into a temporary pointer temp.

temp = head

Traverse the list until we find the desired data value.

while(temp -> data != val)

temp = temp -> next;

Check if this is the last node of the list. If it is so then we can't perform deletion.

```
if(temp -> next == NULL)
   {
return;
   }
```

Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.

```
if(temp -> next -> next == NULL)
   {
      temp ->next = NULL;
   }
```

Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

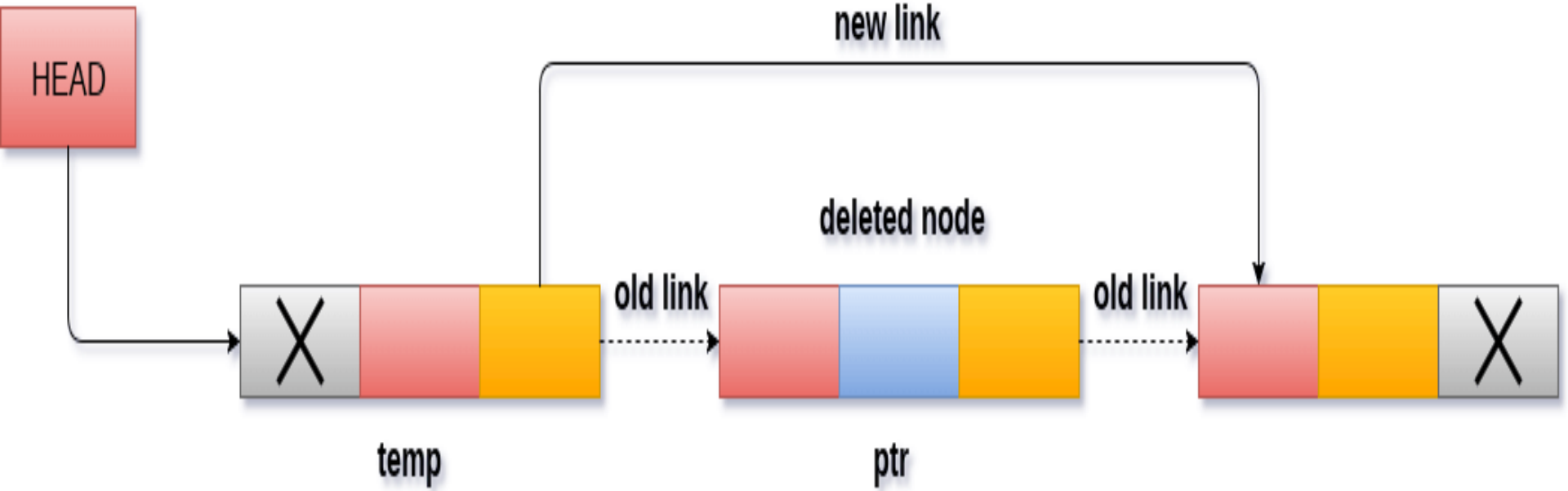
```
ptr = temp -> next;
      temp -> next = ptr -> next;
      ptr -> next -> prev = temp;
      free(ptr);
```

Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 9
[END OF IF]

Step 2: SET TEMP = HEAD
Step 3: Repeat Step 4 while TEMP -> DATA != ITEM
Step 4: SET TEMP = TEMP -> NEXT
[END OF LOOP]

Step 5: SET PTR = TEMP -> NEXT
Step 6: SET TEMP -> NEXT = PTR -> NEXT
Step 7: SET PTR -> NEXT -> PREV = TEMP
Step 8: FREE PTR
Step 9: EXIT




```
temp -> next = ptr -> next
ptr ->next -> prev = temp
free(ptr)
```

Deletion of a specified node in doubly linked list

Searching for a specific node in Doubly Linked List

We just need traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

Copy head pointer into a temporary pointer variable ptr.

ptr = head

declare a local variable I and assign it to 0.

i=0

Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.

Compare each element of the list with the item which is to be searched.

If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm

Step 1: IF HEAD == NULL
  WRITE "UNDERFLOW"
 GOTO STEP 8
 [END OF IF]

Step 2: Set PTR = HEAD
Step 3: Set i = 0
Step 4: Repeat step 5 to 7 while PTR != NULL
Step 5: IF PTR → data = item
return i
[END OF IF]

Step 6: i = i + 1
Step 7: PTR = PTR → next
Step 8: Exit

Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

Ptr = head

then, traverse through the list by using while loop. Keep shifting value of pointer variable ptr until we find the last node. The last node contains null in its next part.

```
while(ptr != NULL)
    {
       printf("%d\n",ptr->data);
       ptr=ptr->next;
    }
```

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm

Step 1: IF HEAD == NULL
  WRITE "UNDERFLOW"
 GOTO STEP 6
 [END OF IF]

Step 2: Set PTR = HEAD
Step 3: Repeat step 4 and 5 while PTR != NULL
Step 4: Write PTR → data
Step 5: PTR = PTR → next
Step 6: Exit

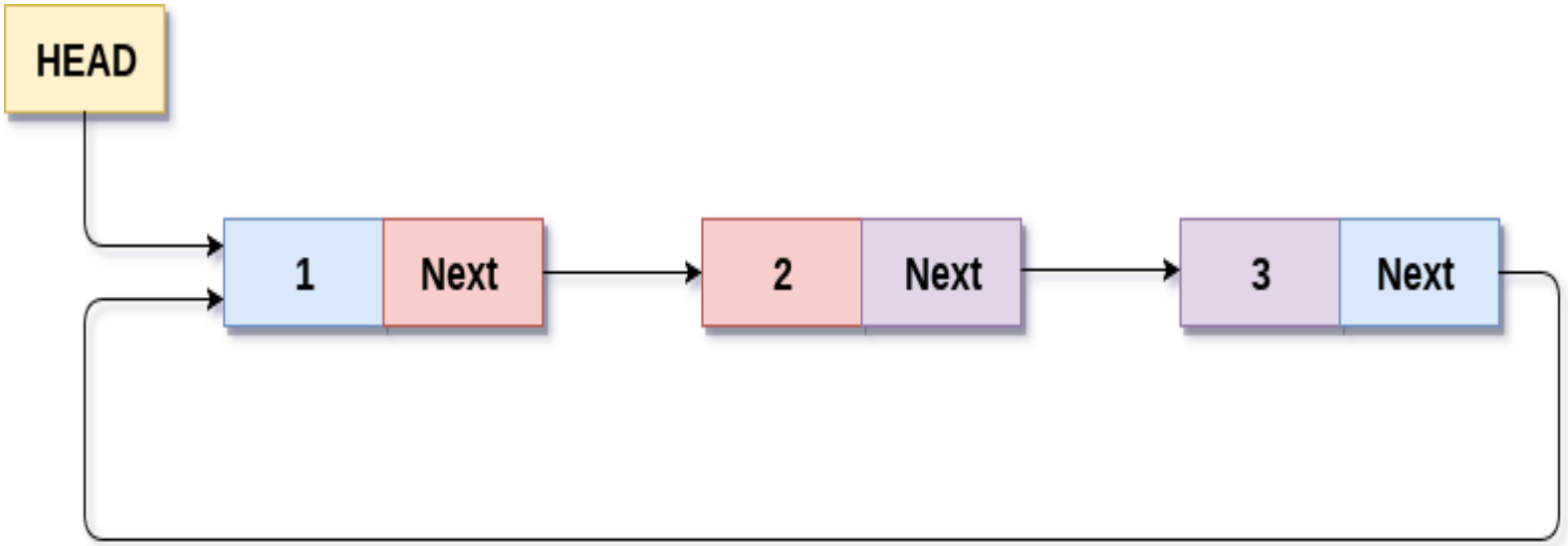
Circular Singly Linked List

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

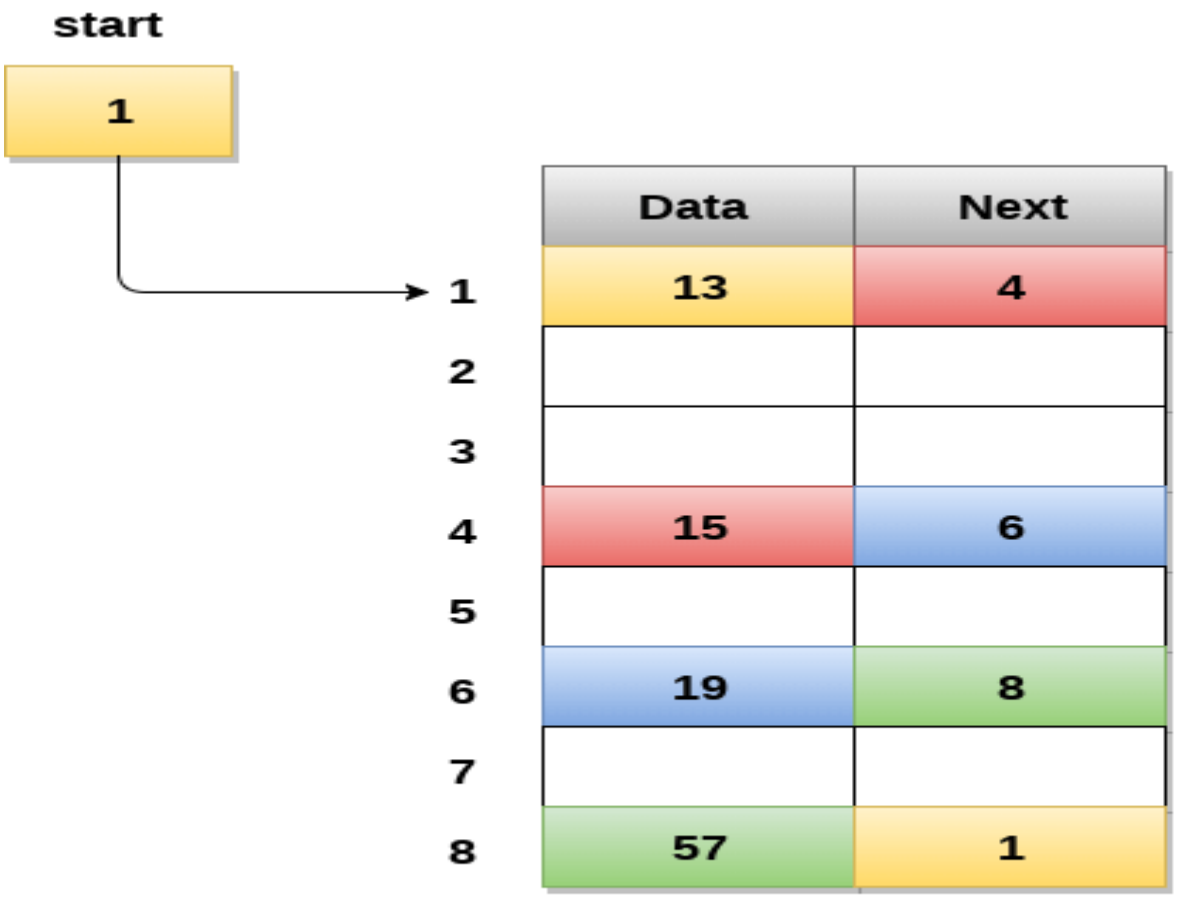


Circular Singly Linked List

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



Memory Representation of a circular linked list

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Insertion into circular singly linked list at beginning

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

Firstly, allocate the memory space for the new node by using the malloc method of C language.

*struct node *ptr = (struct node *)malloc(sizeof(struct node));*

In the first scenario, the condition head == NULL will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

```
if(head == NULL)
    {
       head = ptr;
       ptr -> next = head;
    }
```

In the second scenario, the condition head == NULL will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

```
temp = head;
        while(temp->next != head)
            temp = temp->next;
```

At the end of the loop, the pointer temp would point to the last node of the list. Since, in a circular singly linked list, the last node of the list contains a pointer to the first node of the list. Therefore, we need to make the next pointer of the last node point to the head node of the list and the new node which is being inserted into the list will be the new head node of the list therefore the next pointer of temp will point to the new node ptr.

This will be done by using the following statements.

temp -> next = ptr;

the next pointer of temp will point to the existing head node of the list.

ptr->next = head;

Now, make the new node ptr, the new head node of the circular singly linked list.

head = ptr;

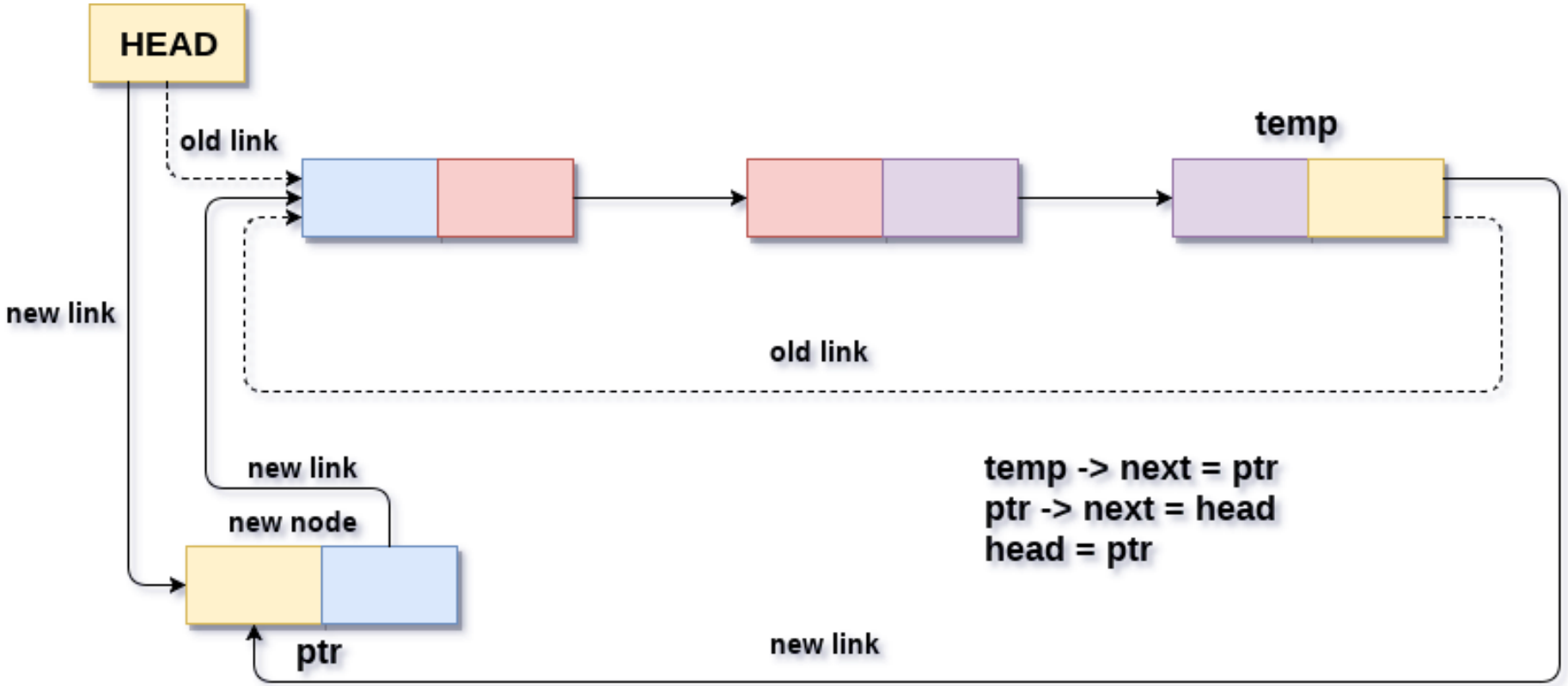
in this way, the node ptr has been inserted into the circular singly linked list at beginning.

Algorithm

Step 1: IF PTR = NULL
  Write OVERFLOW
 Go to Step 11
 [END OF IF]

Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET TEMP = HEAD
Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD
Step 7: SET TEMP = TEMP -> NEXT
[END OF LOOP]

Step 8: SET NEW_NODE -> NEXT = HEAD
Step 9: SET TEMP → NEXT = NEW_NODE
Step 10: SET HEAD = NEW_NODE
Step 11: EXIT



Insertion into circular singly linked list at beginning

Insertion into circular singly linked list at the end

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

Firstly, allocate the memory space for the new node by using the malloc method of C language.

*struct node *ptr = (struct node *)malloc(sizeof(struct node));*

In the first scenario, the condition head == NULL will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

```
if(head == NULL)
    {
      head = ptr;
      ptr -> next = head;
    }
```

In the second scenario, the condition head == NULL will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

```
temp = head;
        while(temp->next != head)
            temp = temp->next;
```

At the end of the loop, the pointer temp would point to the last node of the list. Since, the new node which is being inserted into the list will be the new last node of the list. Therefore the existing last node i.e. temp must point to the new node ptr. This is done by using the following statement.

```
temp -> next = ptr;
```

The new last node of the list i.e. ptr will point to the head node of the list.

```
ptr -> next = head;
```

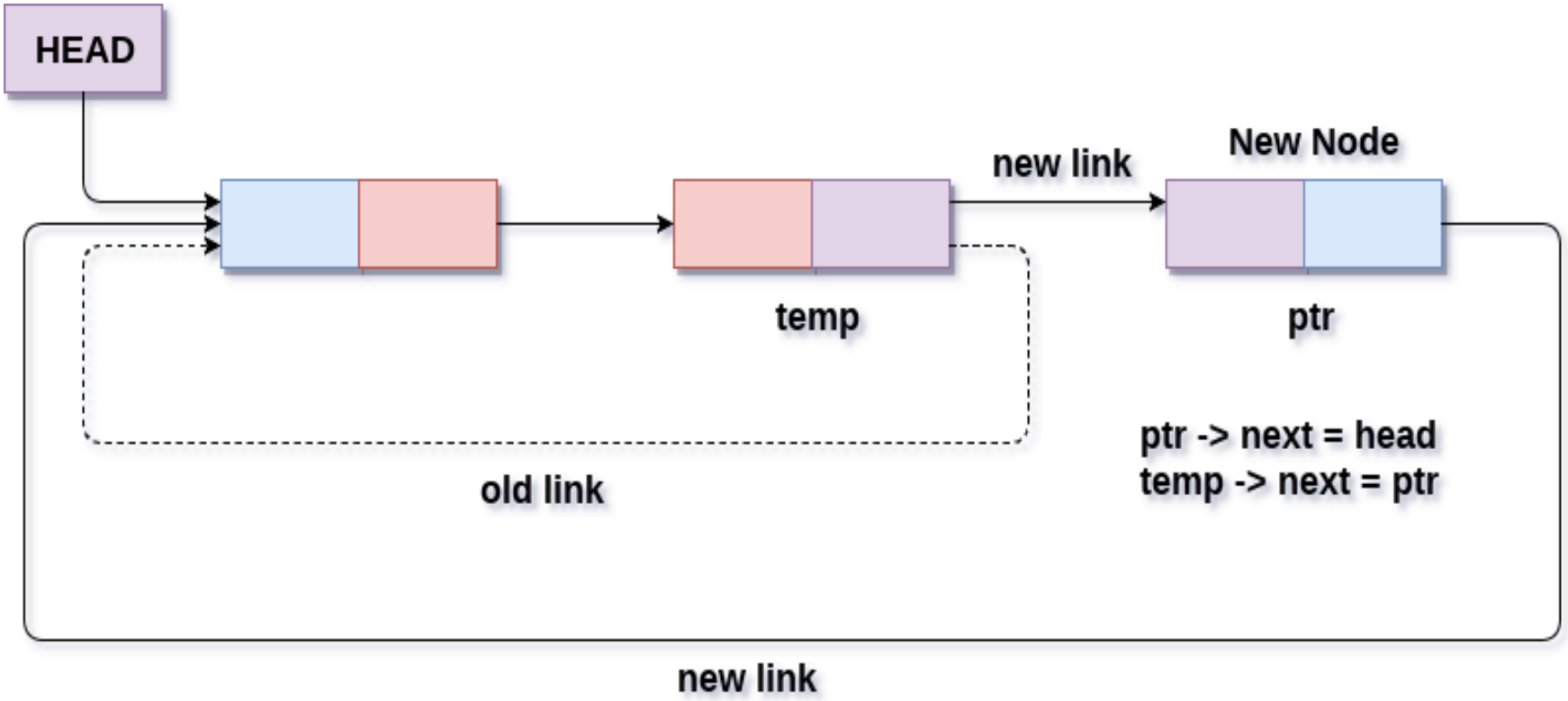
In this way, a new node will be inserted in a circular singly linked list at the beginning.

Algorithm

Step 1: IF PTR = NULL
  Write OVERFLOW
   Go to Step 1
  [END OF IF]

Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = HEAD
Step 6: SET TEMP = HEAD
Step 7: Repeat Step 8 while TEMP -> NEXT != HEAD
Step 8: SET TEMP = TEMP -> NEXT
[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE
Step 10: EXIT



Insertion into circular singly linked list at end

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Deletion in circular singly linked list at beginning

In order to delete a node in circular singly linked list, we need to make a few pointer adjustments.

There are three scenarios of deleting a node from circular singly linked list at beginning.

Scenario 1: (The list is Empty)

If the list is empty then the condition head == NULL will become true, in this case, we just need to print underflow on the screen and make exit.

```
if(head == NULL)
   {
      printf("\nUNDERFLOW");
      return;
   }
```

Scenario 2: (The list contains single node)

If the list contains single node then, the condition head → next == head will become true. In this case, we need to delete the entire list and make the head pointer free. This will be done by using the following statements.

```
if(head->next == head)
{
   head = NULL;
   free(head);
}
```

Scenario 3: (The list contains more than one node)

If the list contains more than one node then, in that case, we need to traverse the list by using the pointer ptr to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
        while(ptr -> next != head)
           ptr = ptr -> next;
```

At the end of the loop, the pointer ptr point to the last node of the list. Since, the last node of the list points to the head node of the list. Therefore this will be changed as now, the last node of the list will point to the next of the head node.

```
ptr->next = head->next;
```

Now, free the head pointer by using the free() method in C language.

```
free(head);
```

Make the node pointed by the next of the last node, the new head of the list.

```
head = ptr->next;
```

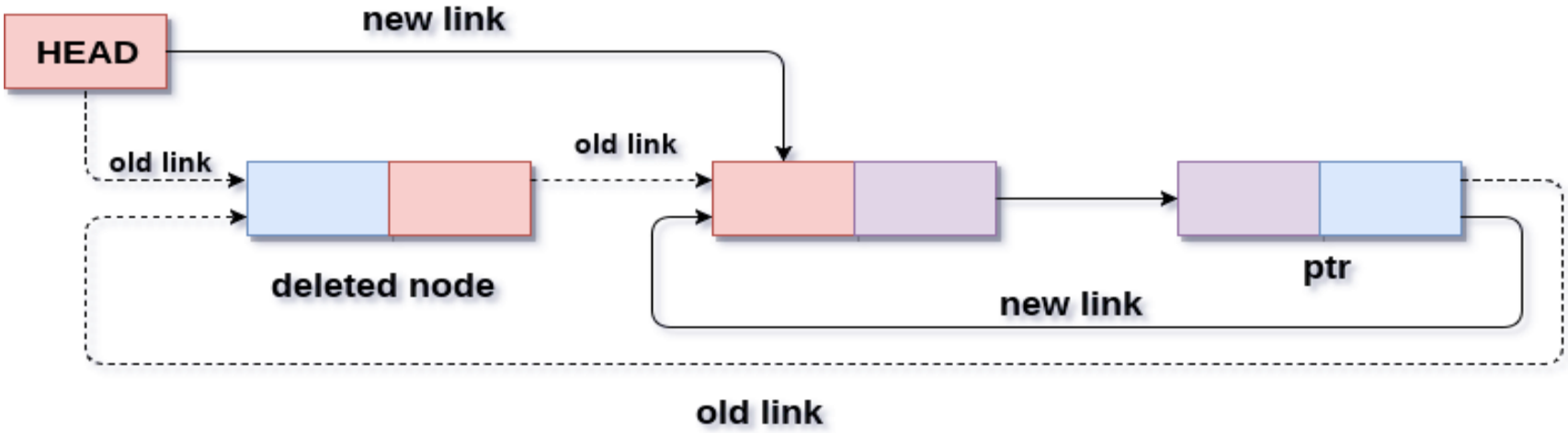
In this way, the node will be deleted from the circular singly linked list from the beginning.

Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
  Go to Step 8
 [END OF IF]

Step 2: SET PTR = HEAD
Step 3: Repeat Step 4 while PTR → NEXT != HEAD
Step 4: SET PTR = PTR → next
[END OF LOOP]

Step 5: SET PTR → NEXT = HEAD → NEXT
Step 6: FREE HEAD
Step 7: SET HEAD = PTR → NEXT
Step 8: EXIT



ptr -> next = head -> next
free head
head = ptr -> next

Deletion in circular singly linked list at beginning

Deletion in Circular singly linked list at the end

There are three scenarios of deleting a node in circular singly linked list at the end.

Scenario 1 (the list is empty)

If the list is empty then the condition head == NULL will become true, in this case, we just need to print underflow on the screen and make exit.

```
if(head == NULL)
   {
      printf("\nUNDERFLOW");
      return;
   }
```

Scenario 2(the list contains single element)

If the list contains single node then, the condition head → next == head will become true. In this case, we need to delete the entire list and make the head pointer free. This will be done by using the following statements.

```
if(head->next == head)
{
   head = NULL;
   free(head);
}
```

Scenario 3(the list contains more than one element)

If the list contains more than one element, then in order to delete the last element, we need to reach the last node. We also need to keep track of the second last node of the list. For this purpose, the two pointers ptr and preptr are defined. The following sequence of code is used for this purpose.

```
ptr = head;
      while(ptr ->next != head)
      {
         preptr=ptr;
         ptr = ptr->next;
      }
```

now, we need to make just one more pointer adjustment. We need to make the next pointer of preptr point to the next of ptr (i.e. head) and then make pointer ptr free.

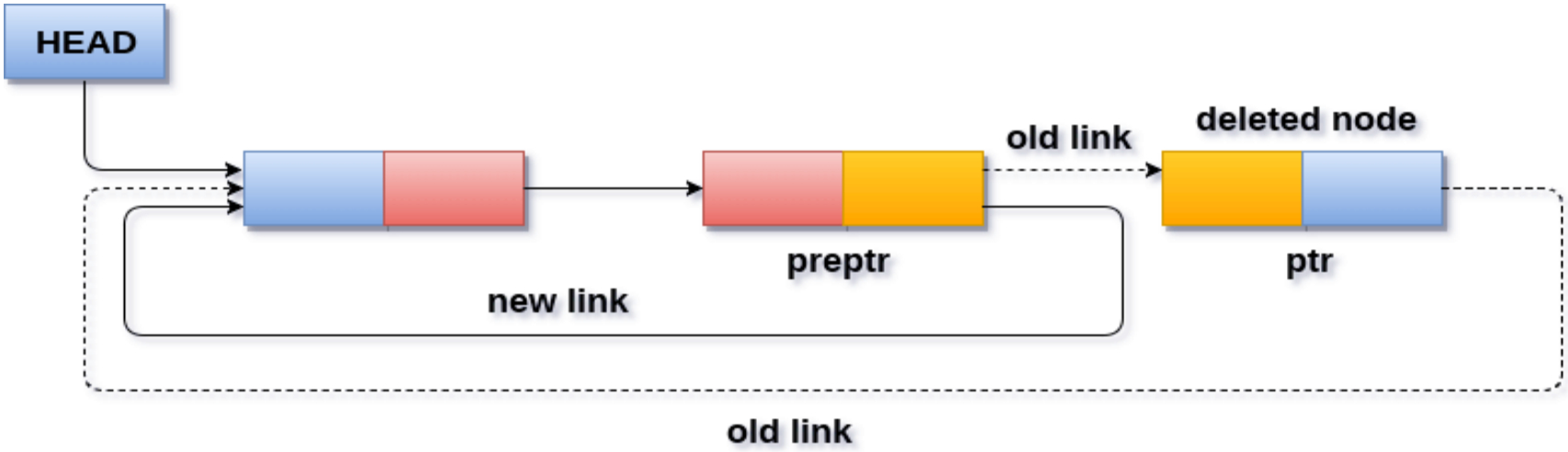
```
preptr->next = ptr -> next;
      free(ptr);
```

Algorithm

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 != HEAD
Step 4: SET PREPTR = PTR
Step 5: SET PTR = PTR -> NEXT
[END OF LOOP]

Step 6: SET PREPTR -> NEXT = HEAD
Step 7: FREE PTR
Step 8: EXIT



preptr -> next = head
free ptr

Deletion in circular singly linked list at end

Searching in circular singly linked list

Searching in circular singly linked list needs traversing across the list. The item which is to be searched in the list is matched with each node data of the list once and if the match found then the location of that item is returned otherwise -1 is returned.

Algorithm

Step 1: SET PTR = HEAD
Step 2: Set I = 0
STEP 3: IF PTR = NULL
WRITE "EMPTY LIST"
GOTO STEP 8
END OF IF

STEP 4: IF HEAD → DATA = ITEM
WRITE i+1 RETURN [END OF IF]
STEP 5: REPEAT STEP 5 TO 7 UNTIL PTR->next != head
STEP 6: if ptr → data = item
write i+1
RETURN
End of IF
STEP 7: I = I + 1
STEP 8: PTR = PTR → NEXT
[END OF LOOP]

STEP 9: EXIT

Traversing in Circular Singly linked list

Traversing in circular singly linked list can be done through a loop. Initialize the temporary pointer variable temp to head pointer and run the while loop until the next pointer of temp becomes head.

Algorithm

STEP 1: SET PTR = HEAD
STEP 2: IF PTR = NULL
WRITE "EMPTY LIST"
GOTO STEP 8
END OF IF

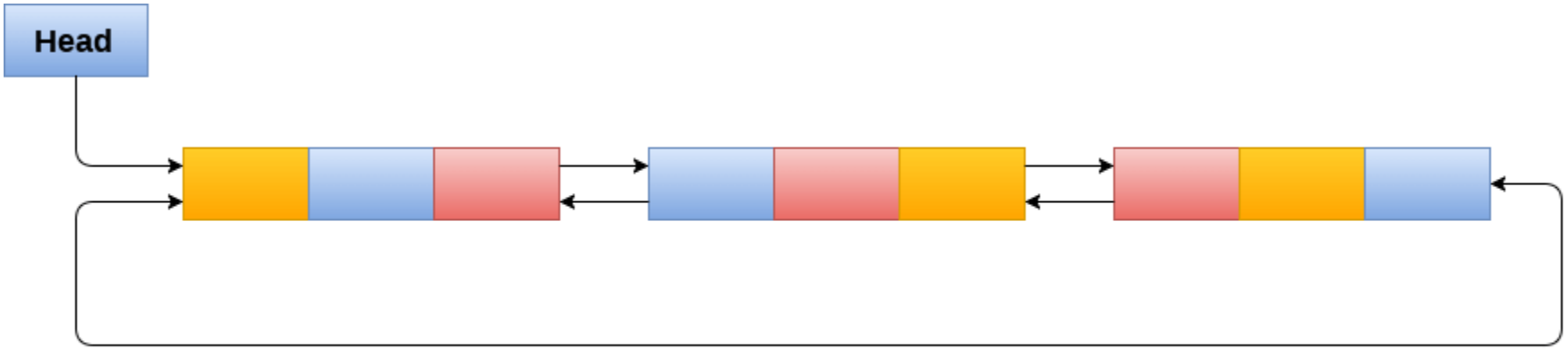
STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR → NEXT != HEAD
STEP 5: PRINT PTR → DATA
STEP 6: PTR = PTR → NEXT
[END OF LOOP]

STEP 7: PRINT PTR → DATA
STEP 8: EXIT

Circular Doubly Linked List

Circular Doubly Linked List

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.

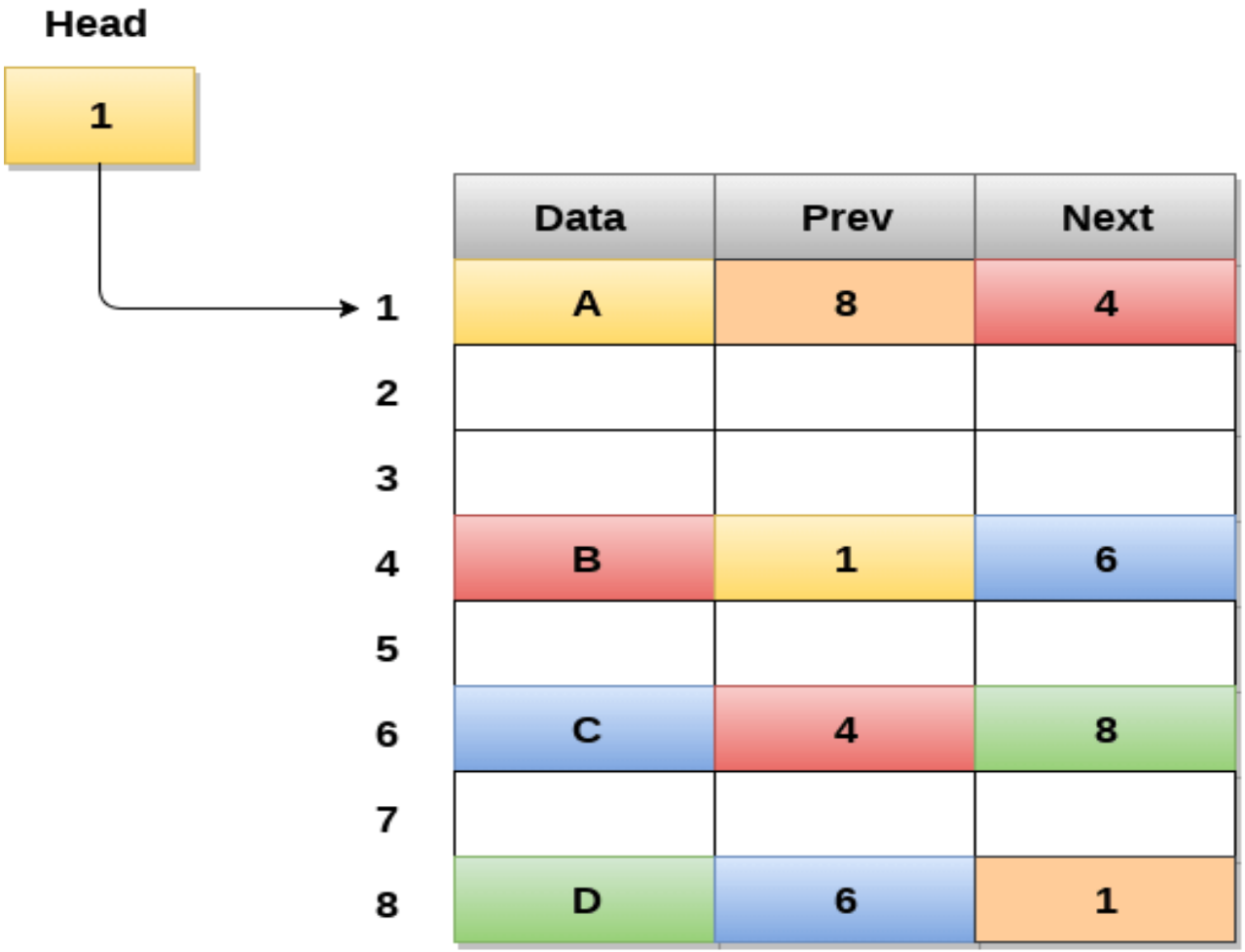


Circular Doubly Linked List

Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

Memory Management of Circular Doubly linked list

The following figure shows the way in which the memory is allocated for a circular doubly linked list. The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1. Since, each node of the list is supposed to have three parts therefore, the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4. The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image i.e. 1. In circular doubly linked list, the last node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.



Memory Representation of a Circular Doubly linked list

Operations on circular doubly linked list :

SN	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.

Insertion in circular doubly linked list at beginning

There are two scenario of inserting a node in circular doubly linked list at beginning. Either the list is empty or it contains more than one element in the list.

Allocate the memory space for the new node ptr by using the following statement.

*ptr = (struct node *)malloc(sizeof(struct node));*

In the first case, the condition head == NULL becomes true therefore, the node will be added as the first node in the list. The next and the previous pointer of this newly added node will point to itself only. This can be done by using the following statement.

head = ptr;

ptr -> next = head;

ptr -> prev = head;

In the second scenario, the condition head == NULL becomes false. In this case, we need to make a few pointer adjustments at the end of the list. For this purpose, we need to reach the last node of the list through traversing the list. Traversing the list can be done by using the following statements.

```
temp = head;
while(temp -> next != head)
{
   temp = temp -> next;
}
```

At the end of loop, the pointer temp would point to the last node of the list. Since the node which is to be inserted will be the first node of the list therefore, temp must contain the address of the new node ptr into its next part.

All the pointer adjustments can be done by using the following statements.

temp -> next = ptr;

ptr -> prev = temp;

head -> prev = ptr;

ptr -> next = head;

head = ptr;

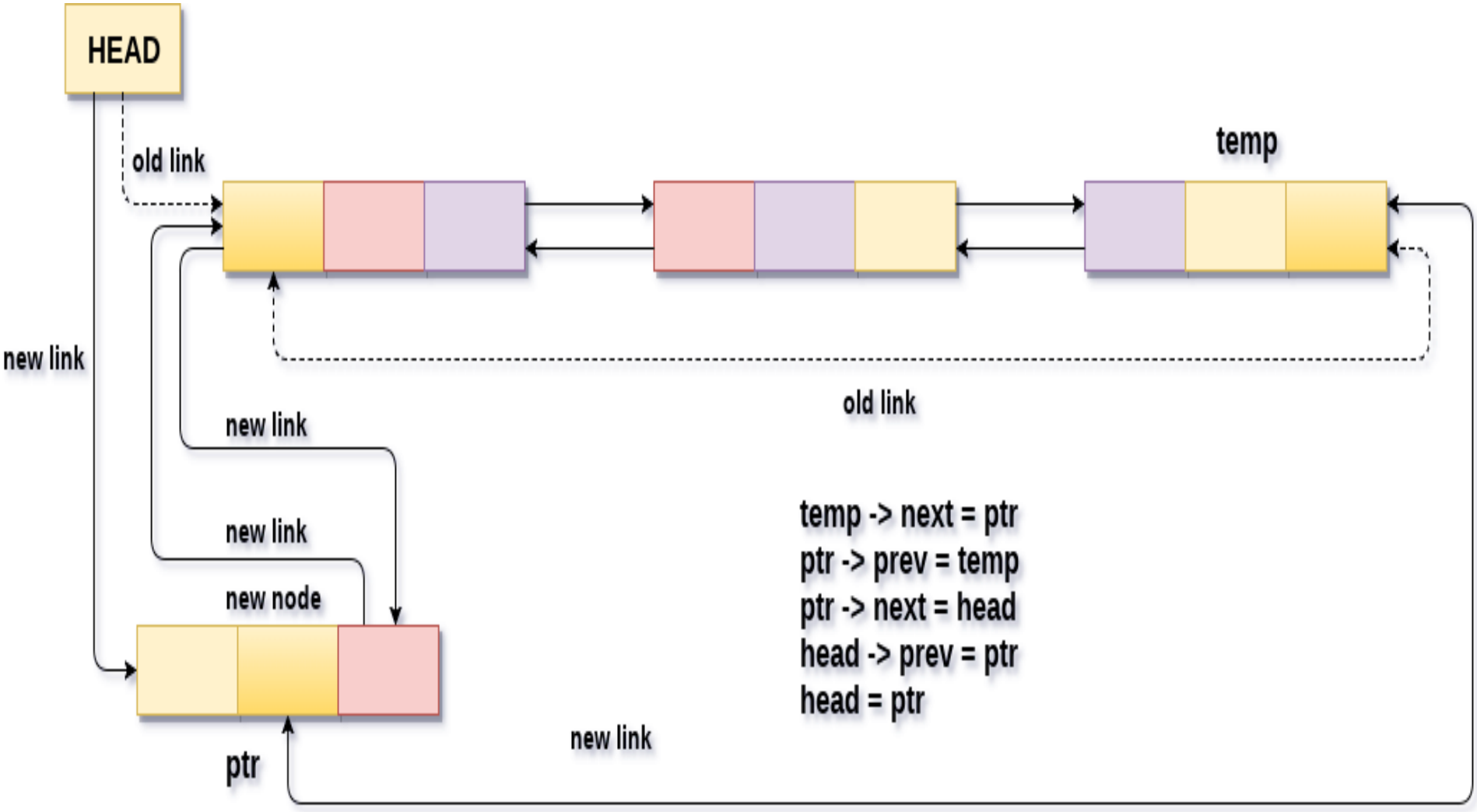
In this way, the new node is inserted into the list at the beginning.

Algorithm

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 13
[END OF IF]

Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET TEMP = HEAD
Step 6: Repeat Step 7 while TEMP -> NEXT != HEAD
Step 7: SET TEMP = TEMP -> NEXT
[END OF LOOP]

Step 8: SET TEMP -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = TEMP
Step 1 : SET NEW_NODE -> NEXT = HEAD
Step 11: SET HEAD -> PREV = NEW_NODE
Step 12: SET HEAD = NEW_NODE
Step 13: EXIT



Insertion into circular doubly linked list at beginning

Insertion in circular doubly linked list at end

There are two scenario of inserting a node in circular doubly linked list at the end. Either the list is empty or it contains more than one element in the list.

Allocate the memory space for the new node ptr by using the following statement.

*ptr = (struct node *)malloc(sizeof(struct node));*

In the first case, the condition head == NULL becomes true therefore, the node will be added as the first node in the list. The next and the previous pointer of this newly added node will point to itself only. This can be done by using the following statement.

head = ptr;

ptr -> next = head;

ptr -> prev = head;

In the second scenario, the condition head == NULL become false, therefore node will be added as the last node in the list. For this purpose, we need to make a few pointer adjustments in the list at the end. Since, the new node will contain the address of the first node of the list therefore we need to make the next pointer of the last node point to the head node of the list. Similarly, the previous pointer of the head node will also point to the new last node of the list.

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

Now, we also need to make the next pointer of the existing last node of the list (temp) point to the new last node of the list, similarly, the new last node will also contain the previous pointer to the temp. this will be done by using the following statements.

temp->next = ptr;

ptr ->prev=temp;

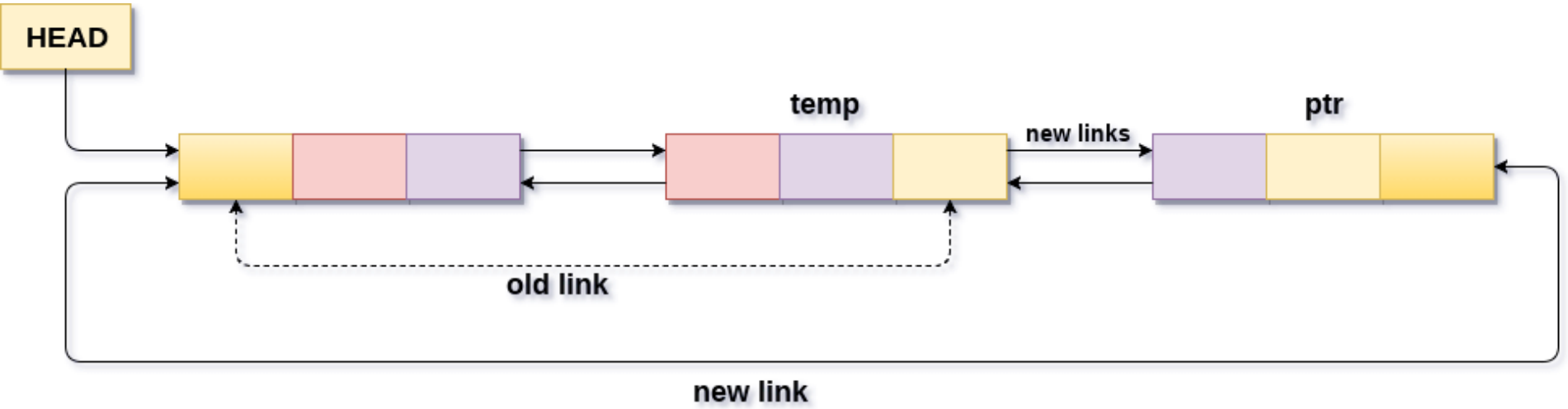
In this way, the new node ptr has been inserted as the last node of the list.

Algorithm

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 12
[END OF IF]

Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = HEAD
Step 6: SET TEMP = HEAD
Step 7: Repeat Step 8 while TEMP -> NEXT != HEAD
Step 8: SET TEMP = TEMP -> NEXT
[END OF LOOP]

Step 9: SET TEMP -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = TEMP
Step 11: SET HEAD -> PREV = NEW_NODE
Step 12: EXIT



temp -> next = ptr
ptr -> prev = temp
head -> prev = ptr
ptr -> next = head

Insertion into circular doubly linked list at end

Deletion in Circular doubly linked list at beginning

There can be two scenario of deleting the first node in a circular doubly linked list.

The node which is to be deleted can be the only node present in the linked list. In this case, the condition head → next == head will become true, therefore the list needs to be completely deleted.

It can be simply done by assigning head pointer of the list to null and free the head pointer.

```
head = NULL;
   free(head);
```

in the second scenario, the list contains more than one element in the list, therefore the condition head → next == head will become false. Now, reach the last node of the list and make a few pointer adjustments there. Run a while loop for this purpose

```
temp = head;
   while(temp -> next != head)
   {
      temp = temp -> next;
   }
```

Now, temp will point to the last node of the list. The first node of the list i.e. pointed by head pointer, will need to be deleted. Therefore the last node must contain the address of the node that is pointed by the next pointer of the existing head node. Use the following statement for this purpose.

temp -> next = head -> next;

The new head node i.e. next of existing head node must also point to the last node of the list through its previous pointer. Use the following statement for this purpose.

head -> next -> prev = temp;

Now, free the head pointer and the make its next pointer, the new head node of the list.

free(head);

head = temp -> next;

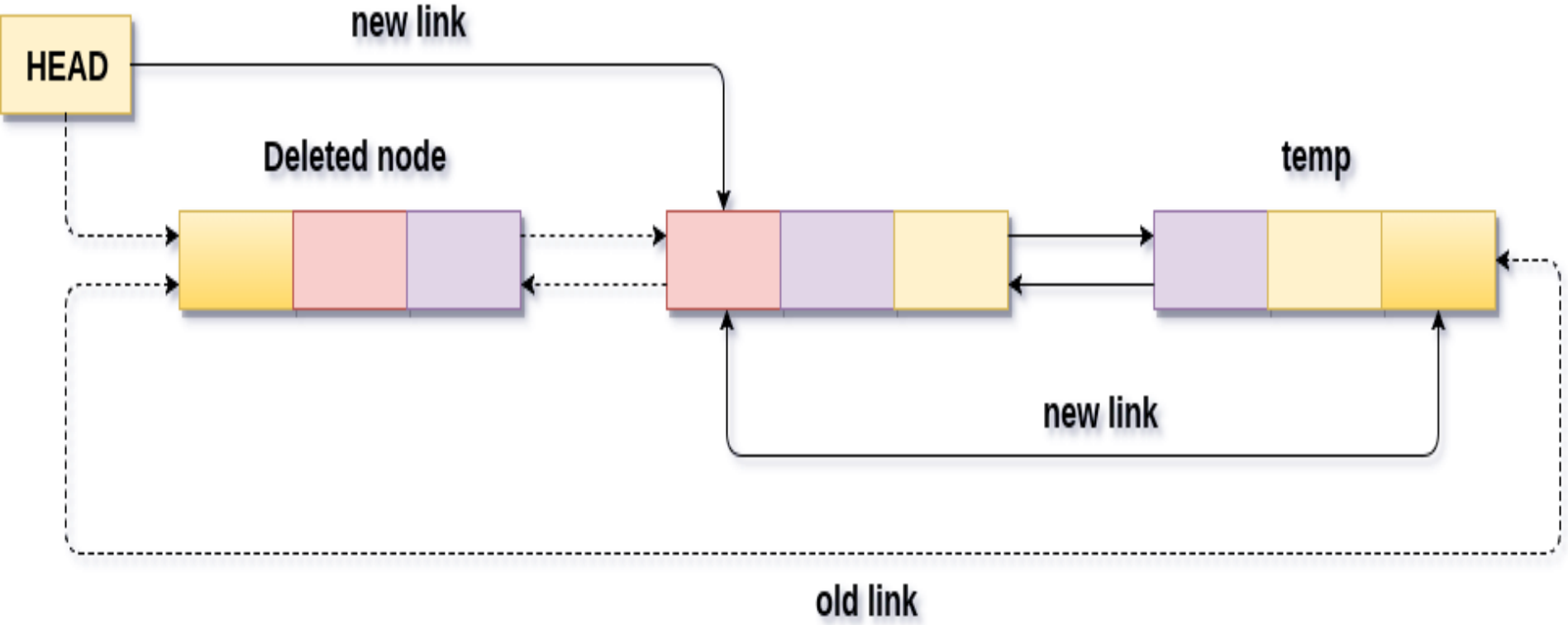
in this way, a node is deleted at the beginning from a circular doubly linked list.

Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]

Step 2: SET TEMP = HEAD
Step 3: Repeat Step 4 while TEMP -> NEXT != HEAD
Step 4: SET TEMP = TEMP -> NEXT
[END OF LOOP]

Step 5: SET TEMP -> NEXT = HEAD -> NEXT
Step 6: SET HEAD -> NEXT -> PREV = TEMP
Step 7: FREE HEAD
Step 8: SET HEAD = TEMP -> NEXT



temp -> next = head -> next
head -> next -> prev = temp
free head
head = temp -> next

Deletion in circular doubly linked list at beginning

Deletion in circular doubly linked list at end

There can be two scenario of deleting the first node in a circular doubly linked list.

The node which is to be deleted can be the only node present in the linked list. In this case, the condition head → next == head will become true, therefore the list needs to be completely deleted.

It can be simply done by assigning head pointer of the list to null and free the head pointer.

```
head = NULL;
   free(head);
```

in the second scenario, the list contains more than one element in the list, therefore the condition head → next == head will become false. Now, reach the last node of the list and make a few pointer adjustments there. Run a while loop for this purpose

```
temp = head;
    while(temp -> next != head)
    {
        temp = temp -> next;
    }
```

Now, temp will point to the node which is to be deleted from the list. Make the next pointer of previous node of temp, point to the head node of the list.

temp -> prev -> next = head;

make the previous pointer of the head node, point to the previous node of temp.

head -> prev = ptr -> prev;

Now, free the temp pointer to free the memory taken by the node.

free(head)

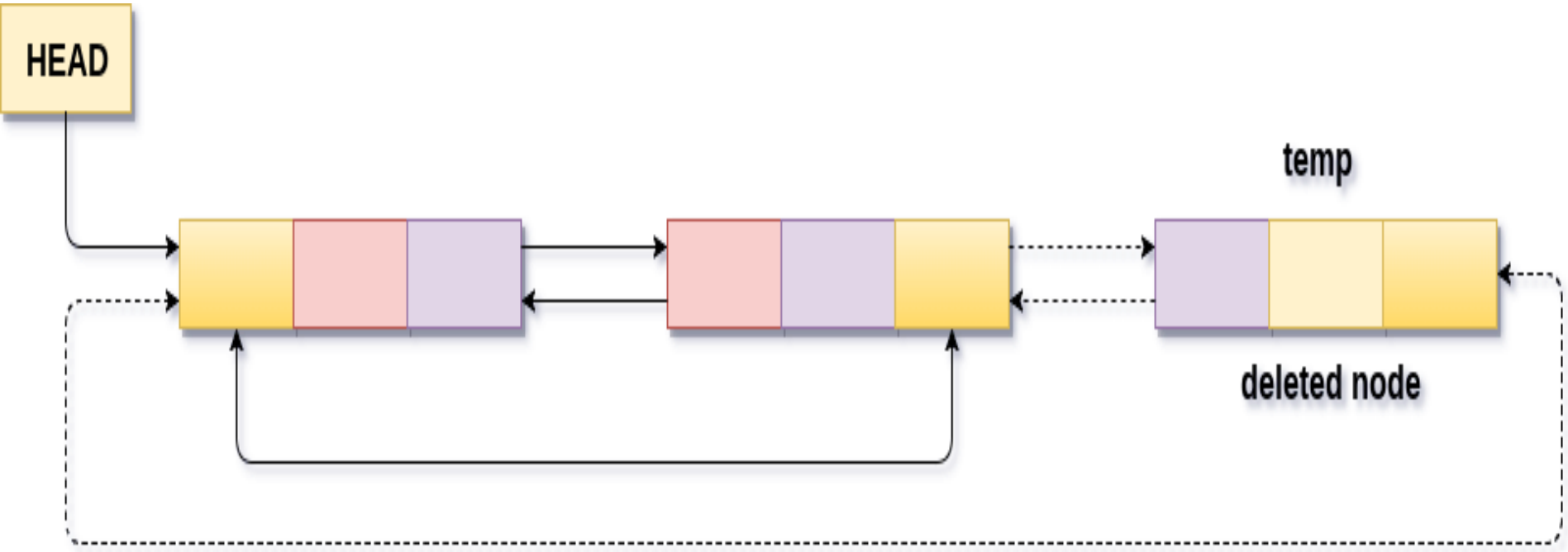
in this way, the last node of the list is deleted.

Algorithm

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]

Step 2: SET TEMP = HEAD
Step 3: Repeat Step 4 while TEMP -> NEXT != HEAD
Step 4: SET TEMP = TEMP -> NEXT
[END OF LOOP]

Step 5: SET TEMP -> PREV -> NEXT = HEAD
Step 6: SET HEAD -> PREV = TEMP -> PREV
Step 7: FREE TEMP
Step 8: EXIT



temp -> prev -> next = HEAD
HEAD -> prev = temp -> prev
free temp

Deletion in circular doubly linked list at beginning