DATA STRUCTURE(3130702)

UNIT -2
LINKED LIST

Array

- We have studied that an array is a linear collection of data elements in which the elements are stored in consecutive memory locations.
- While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store.
- Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations.
- So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs.

Linked List

- Linked list is a data structure that is free from the restrictions mentioned in the previous slide.
- A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.
- However, like an array, a linked list does not allow random access of data.
- Elements in a linked list can be accessed only in a sequential manner.
- But like an array, insertions and deletions can be done at any point in the list in a constant time.

Basic Terminologies

- It is a linear collection of data elements.
- These data elements are called *nodes.*
- Each node contains one or more data fields and a pointer to the next node.
- It can be used to implement other data structures.

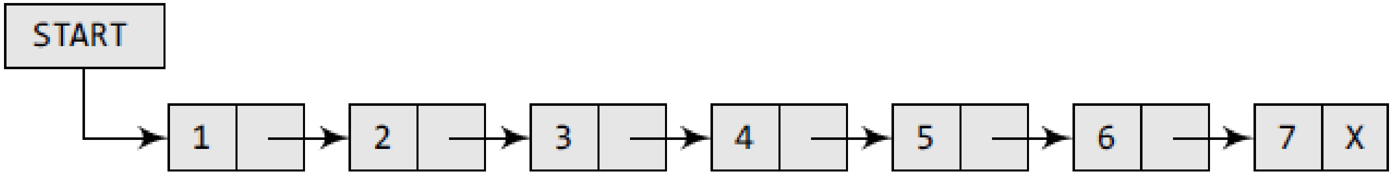


Figure 6.1 Simple linked list

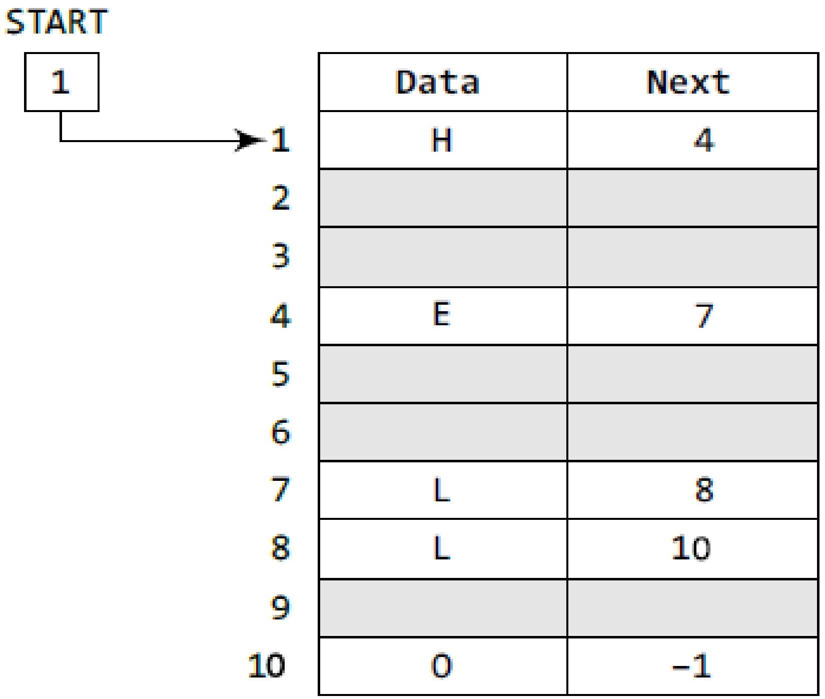
Linked List

- In given linked list, which every node contains two parts, an integer and a pointer to the next node.
- The left part of the node which contains data may include a simple data type, an array, or a structure.
- The right part of the node contains a pointer to the next node (or address of the next node in sequence).
- The last node will have no next node connected to it, so it will store a special value called NULL.
- In given linked list , the NULL pointer is represented by X.
- While in programming, we usually define NULL as –1.
- Hence, a NULL pointer denotes the end of the list.

Linked List

- In a linked list, every node contains a pointer to another node which is of the same type, it is also called a *self-referential data type.*
- Linked lists contain a pointer variable START that stores the address of the first node in the list.
- We can traverse the entire list using START which contains the address of the first node;
- the next part of the first node in turn stores the address of its succeeding node.
- Using this technique, the individual nodes of the list will form a chain of nodes.
- If START = NULL, then linked list is empty & contains no nodes.

Linked List in memory

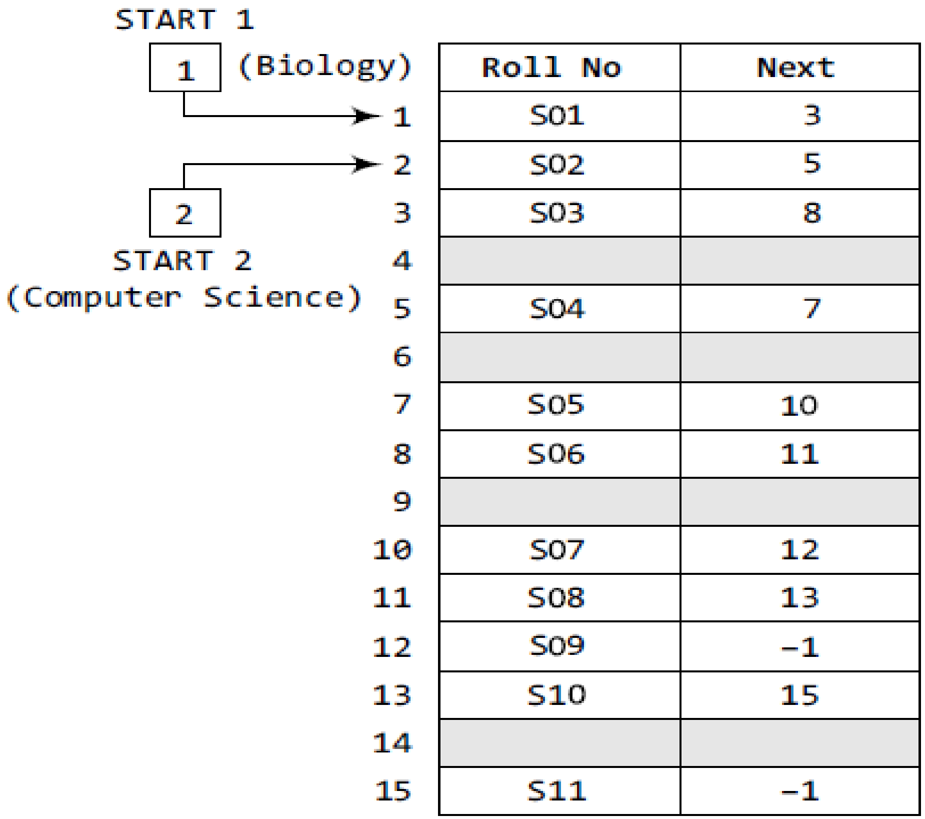


START

1

	Data	Next
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	-1

Figure 6.2 START pointing to the first element of the linked list in the memory



START 1

1 (Biology)

2

START 2

(Computer Science)

	Roll No	Next
1	S01	3
2	S02	5
3	S03	8
4		
5	S04	7
6		
7	S05	10
8	S06	11
9		
10	S07	12
11	S08	13
12	S09	-1
13	S10	15
14		
15	S11	-1

Figure 6.3 Two linked lists which are simultaneously maintained in the memory

Linked List

- In order to form a linked list, we need a structure called *node which has two fields, DATA and NEXT.*
- *DATA will store the* information part and
- NEXT will store the address of the next node in sequence.
- variable START is used to store the address of the first node.
- Here, in this example, START = 1, so the first data is stored at address 1, which is H.
- The corresponding NEXT stores the address of the next node, which is 4.
- So, we will look at address 4 to fetch the next data item.

Linked List

- The second data element obtained from address 4 is E.
- Again, we see the corresponding NEXT to go to the next node.
- From the entry in the NEXT, we get the next address, that is 7,
- And fetch L as the data.
- We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the end of the linked list.
- When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form the word HELLO.

Linked List

- Note that Figure shows a chunk of memory locations which range from 1 to 10.
- The shaded portion contains data for other applications.
- Remember that the nodes of a linked list need not be in consecutive memory locations.
- In our example, the nodes for the linked list are stored at addresses 1, 4, 7, 8, and 10.

Linked List

- Now, look at Figure, two different linked lists are simultaneously maintained in the memory.
- There is no ambiguity in traversing through the list because each list maintains a separate Start pointer, which gives the address of the first node of their respective linked lists.
- The rest of the nodes are reached by looking at the value stored in the NEXT.
- By looking at the figure, we can conclude that roll numbers of the students who have opted for Biology are S01, S03, S06, S08, S10, and S11.
- Similarly, roll numbers of the students who chose Computer Science are S02, S04, S05, S07, and S09.

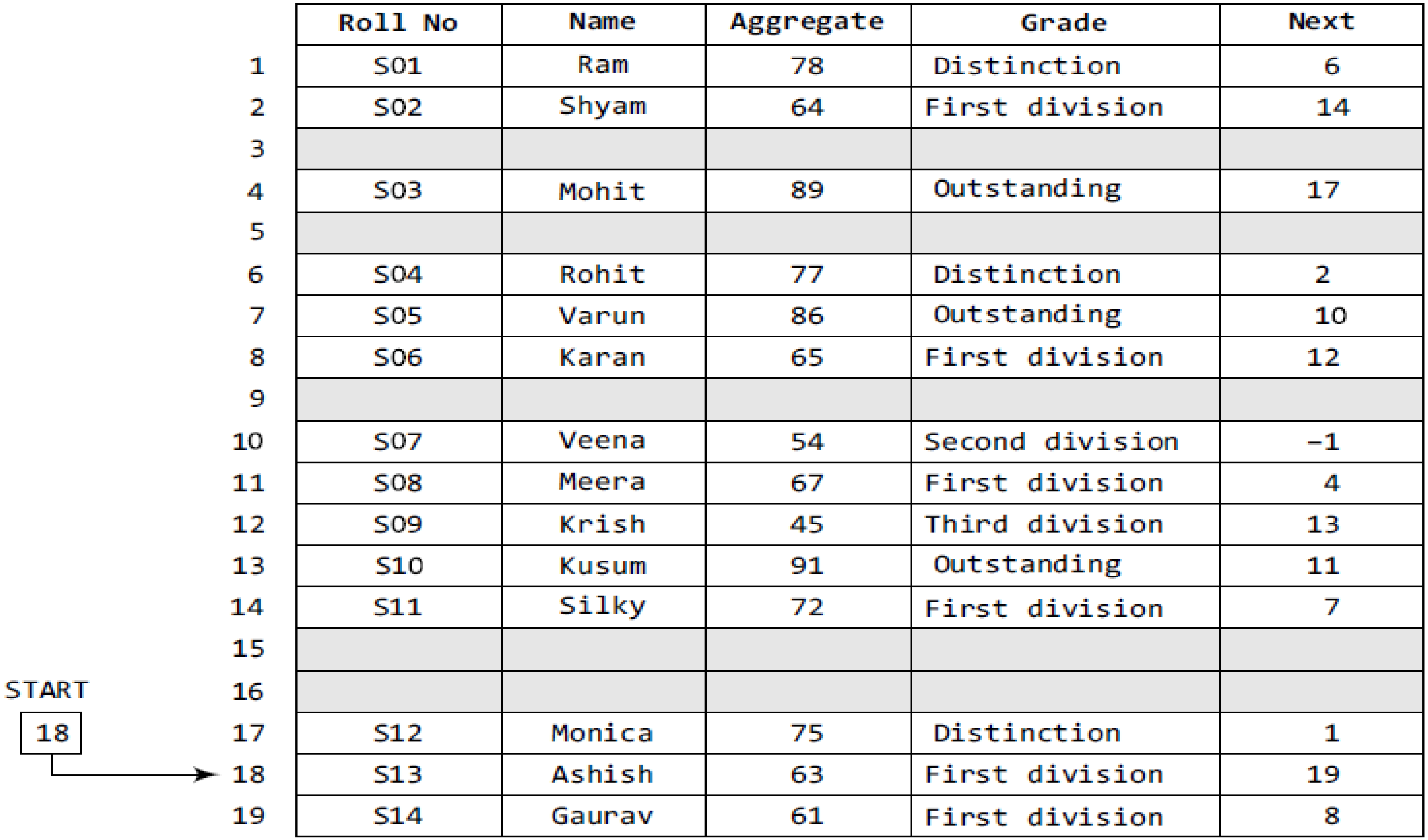
Linked List

- We have already said that the DATA part of a node may contain just a single data item, an array, or a structure.
- Let us take an example to see how a structure is maintained in a linked list that is stored in the memory.
- Consider a scenario in which the roll number, name, aggregate, and grade of students are stored using linked lists.
- Now, we will see how the NEXT pointer is used to store the data alphabetically. This is shown in Fig. 6.4.

Linked Lists versus Arrays

- Linked list does not store its nodes in consecutive memory locations.
- It does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner.
- we can add any number of elements in the list.
- But like an array, insertions and deletions can be done at any point in the list in a constant time.
- Thus, linked lists provide an efficient way of storing related data and performing basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing the address of next nodes.

Students' Linked List



START

18

	Roll No	Name	Aggregate	Grade	Next
1	S01	Ram	78	Distinction	6
2	S02	Shyam	64	First division	14
3					
4	S03	Mohit	89	Outstanding	17
5					
6	S04	Rohit	77	Distinction	2
7	S05	Varun	86	Outstanding	10
8	S06	Karan	65	First division	12
9					
10	S07	Veena	54	Second division	–1
11	S08	Meera	67	First division	4
12	S09	Krish	45	Third division	13
13	S10	Kusum	91	Outstanding	11
14	S11	Silky	72	First division	7
15					
16					
17	S12	Monica	75	Distinction	1
18	S13	Ashish	63	First division	19
19	S14	Gaurav	61	First division	8

Figure 6.4 Students' linked list

Memory Allocation and De-allocation for a Linked List

- If we want to add a node to an already existing linked list in the memory, we first find free space in the memory and then use it to store the information.

Insert

- If a new student joins the class and is asked to appear for the same test that the other students had taken, then the new student's marks should also be recorded in the linked list.
- For this purpose, we find a free space and store the information there.
- In Fig. 6.5 the grey shaded portion shows free space, and thus we have 4 memory locations available.
- We can use any one of them to store our data. This is illustrated in Figs 6.5(a) and (b).

Insert

- The computer maintains a list of all free memory cells.
- This list of available space is called the *free pool.*
- We have seen that every linked list has a pointer variable START which stores the address of the first node of the list.
- Likewise, for the free pool (which is a linked list of all free memory cells), we have a pointer variable AVAIL which stores the address of the first free space.
- Let us revisit the memory representation of the linked list storing all the students' marks in Biology.

Insert

- Now, when a new student’s record has to be added, the memory address pointed by AVAIL will be taken and used to store the desired information.
- After the insertion, the next available free space’s

  address will be stored in AVAIL.
- For example, in Fig. 6.6, when the first free memory space is utilized for inserting the new node, AVAIL will be set to contain address 6.

Delete

- When we delete a particular node from an existing linked list or delete the entire linked list, the space occupied by it must be given back to the free pool so that the memory can be reused by some other program that needs memory space.
- The operating system does this task of adding the freed memory to the free pool.
- The operating system will perform this operation whenever it finds the CPU idle or whenever the programs are falling short of memory space.

Delete

- The operating system scans through all the memory cells and marks those cells that are being used by some program.
- Then it collects all the cells which are not being used and adds their address to the free pool, so that these cells can be reused by other programs.
- This process is called *garbage collection.*

Types of Linked List

- Singly linked list
- Circular linked list
- Doubly linked list
- Circular doubly linked list
- Header linked list
- Multi-linked list

Singly linked list

- It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.
- By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence.
- A singly linked list allows traversal of data only in one way.

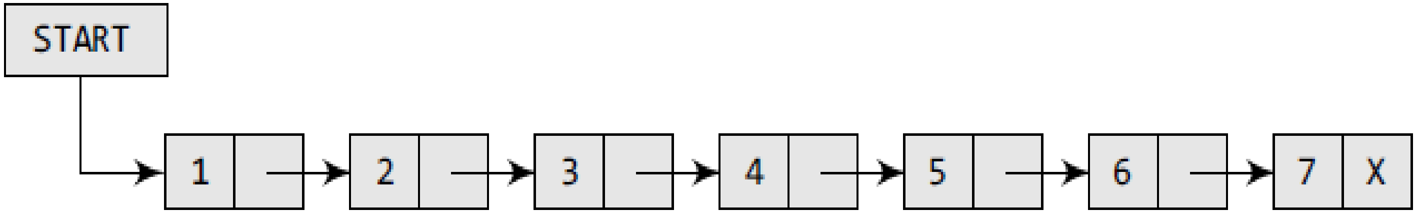
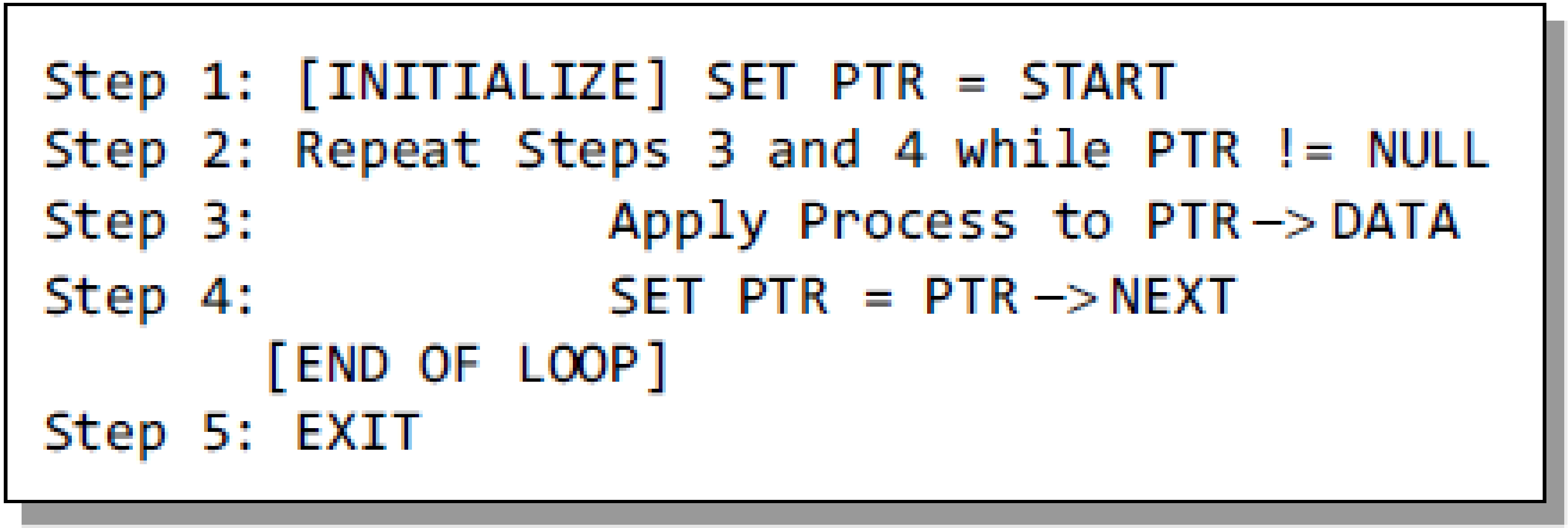


Figure 6.7 Singly linked list

Traversing a Linked List

- Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.
- A linked list always contains a pointer variable START which stores the address of the first node of the list.
- End of the list is marked by storing NULL or –1 in the NEXT field of the last node.
- For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed.

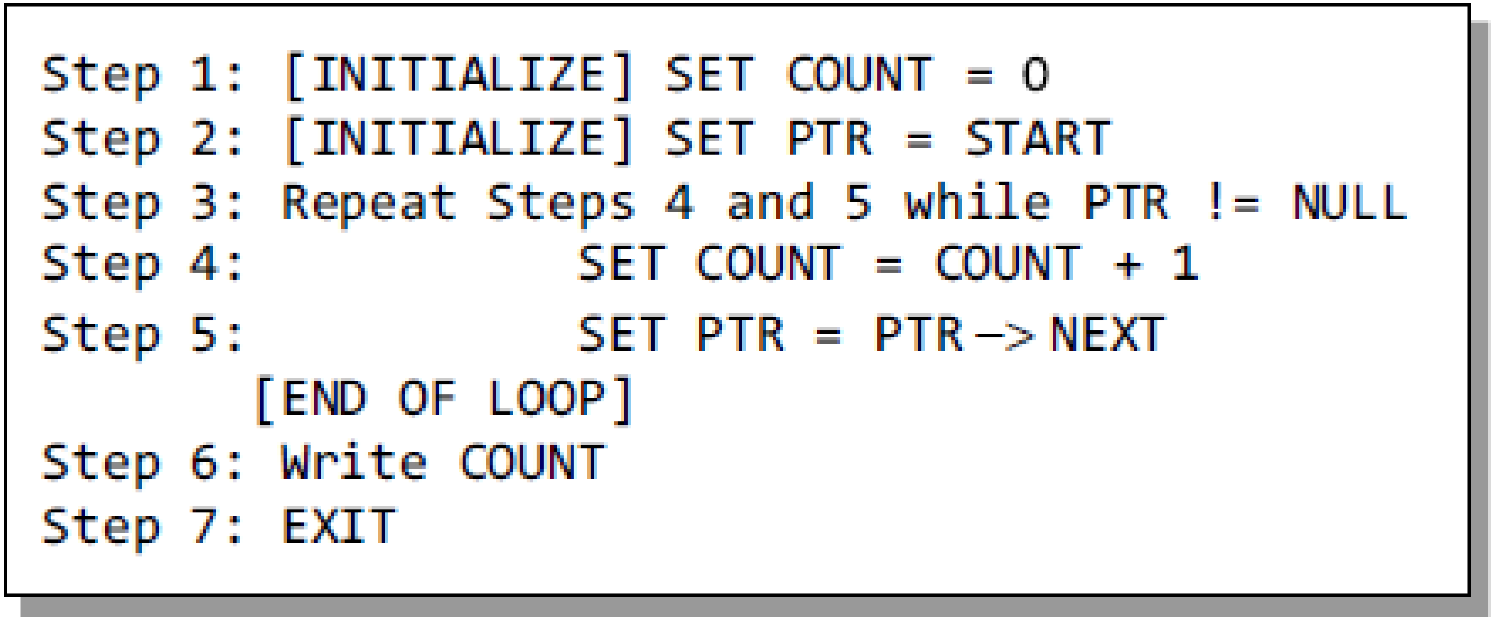
Traversing a Linked List



```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:            Apply Process to PTR -> DATA
Step 4:            SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 5: EXIT
```

Figure 6.8 Algorithm for traversing a linked list

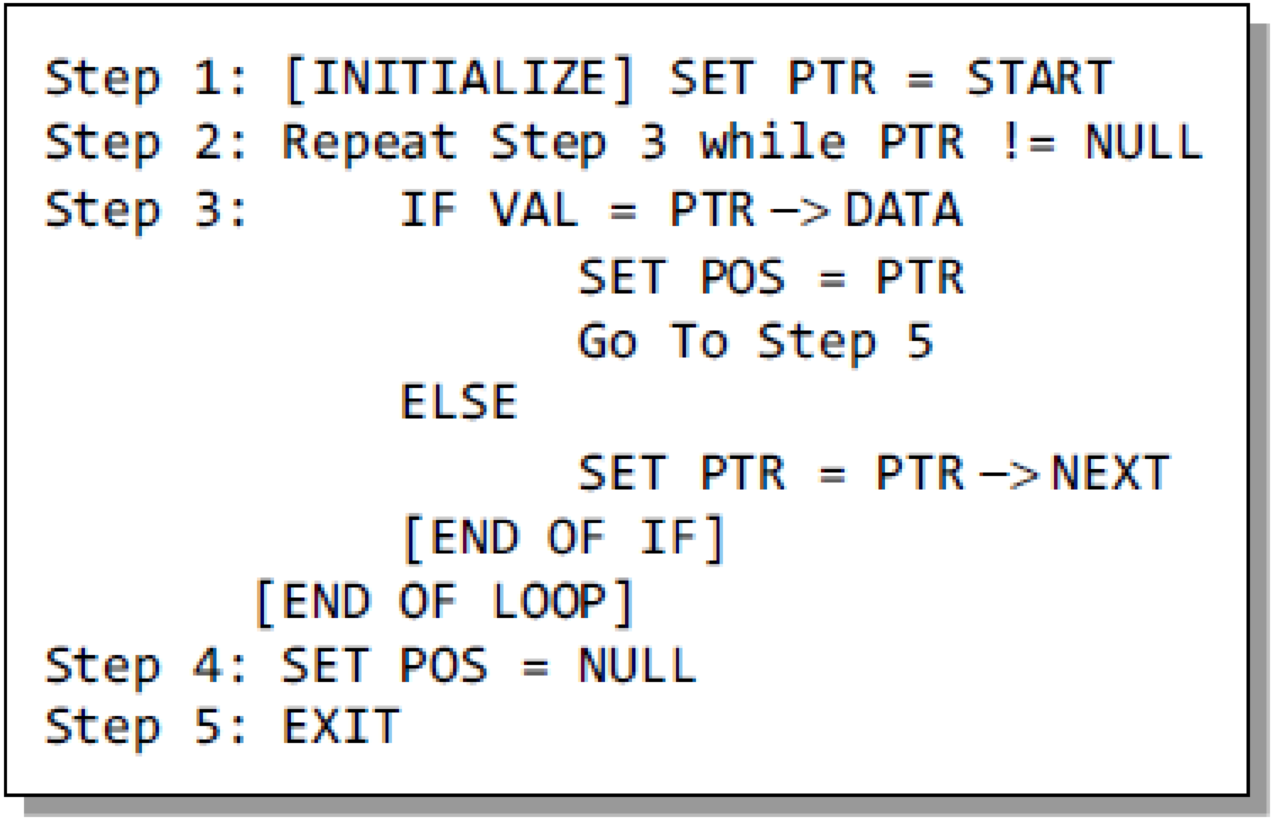
Count nodes in a Linked List



```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:             SET COUNT = COUNT + 1
Step 5:             SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

Figure 6.9 Algorithm to print the number of nodes in a linked list

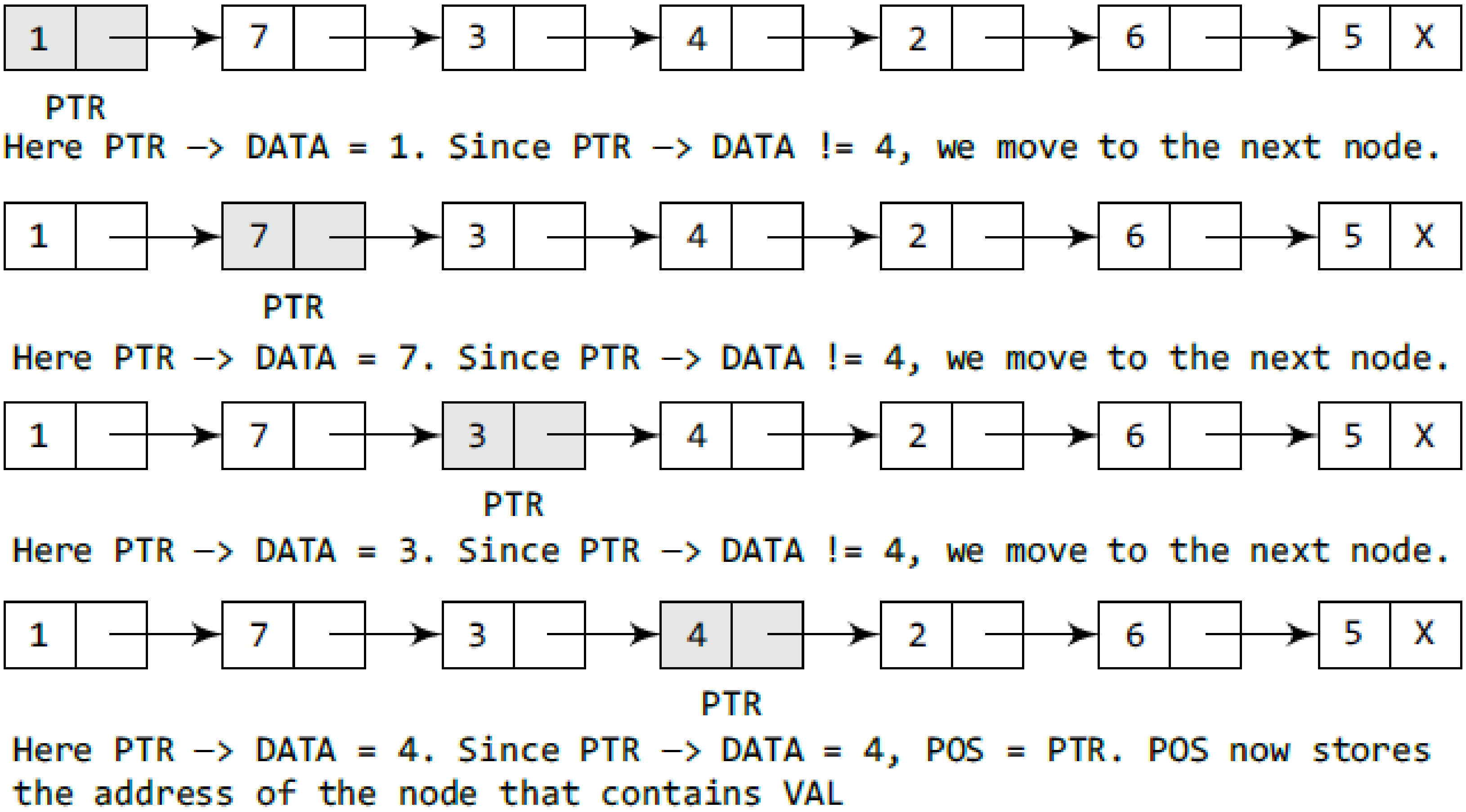
Searching for a Value in a Linked List



```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:       IF VAL = 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
```

Figure 6.10 Algorithm to search a linked list

Searching for a Value in a Linked List



Here PTR -> DATA = 1. Since PTR -> DATA != 4, we move to the next node.

Here PTR -> DATA = 7. Since PTR -> DATA != 4, we move to the next node.

Here PTR -> DATA = 3. Since PTR -> DATA != 4, we move to the next node.

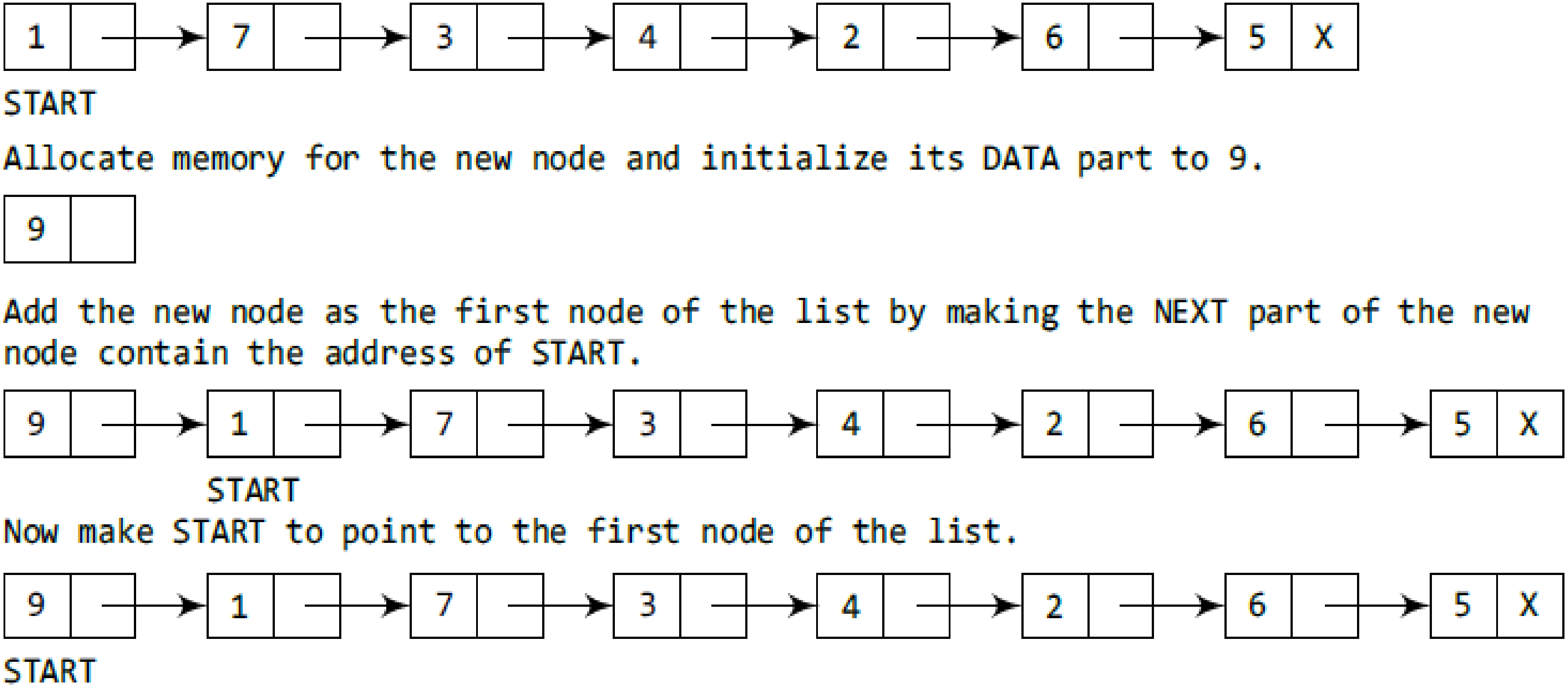
Here PTR -> DATA = 4. Since PTR -> DATA = 4, POS = PTR. POS now stores the address of the node that contains VAL

Figure 6.11 Searching a linked list

Inserting a New Node in a Linked List

- Case 1: The new node is inserted at the beginning.
- Case 2: The new node is inserted at the end.
- Case 3: The new node is inserted after a given node.
- Case 4: The new node is inserted before a given node.
- Overflow is a condition that occurs when AVAIL = NULL or no
- free memory cell is present in the system.
- When this condition occurs, the program must give an appropriate message.

Inserting a Node at the Beginning



1 → 7 → 3 → 4 → 2 → 6 → 5 X
START

Allocate memory for the new node and initialize its DATA part to 9.

Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.

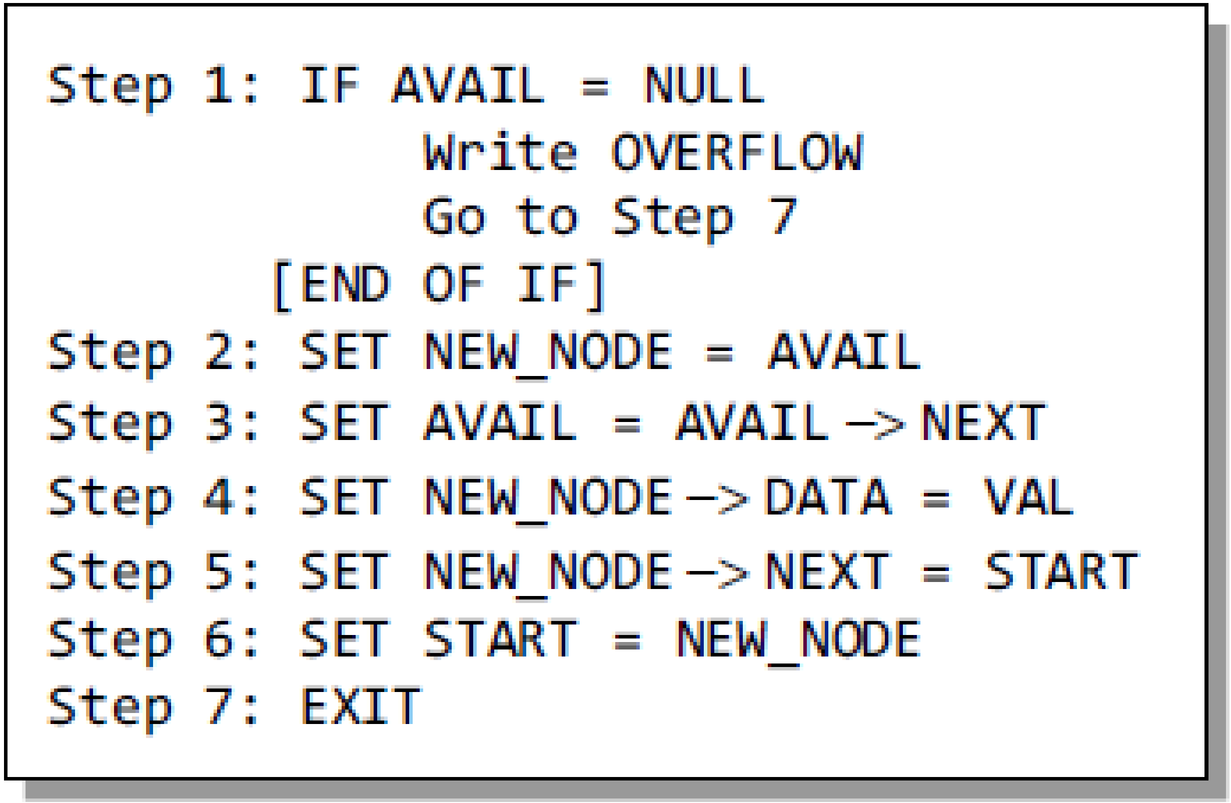
9 → 1 → 7 → 3 → 4 → 2 → 6 → 5 X
START

Now make START to point to the first node of the list.

9 → 1 → 7 → 3 → 4 → 2 → 6 → 5 X
START

Figure 6.12 Inserting an element at the beginning of a linked list

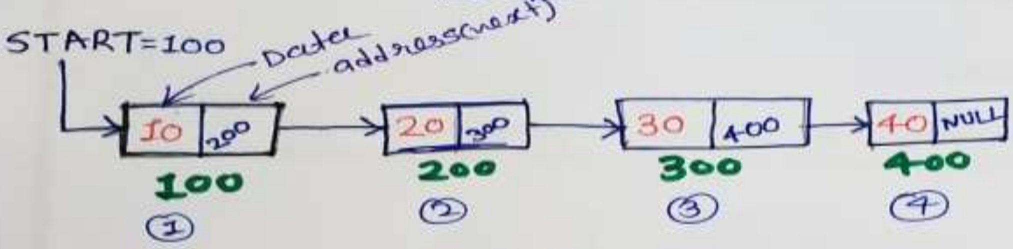
Inserting a Node at the Beginning



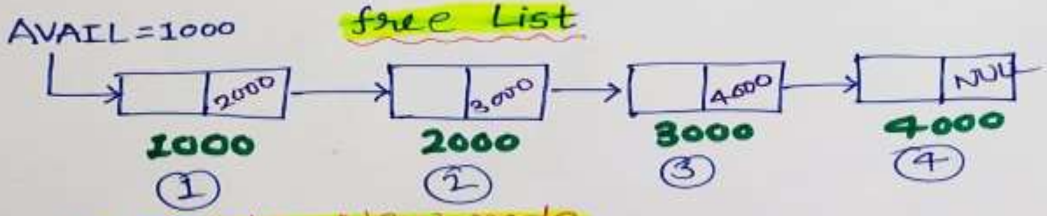
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

Figure 6.13 Algorithm to insert a new node at the beginning

Existing Linked List



free List

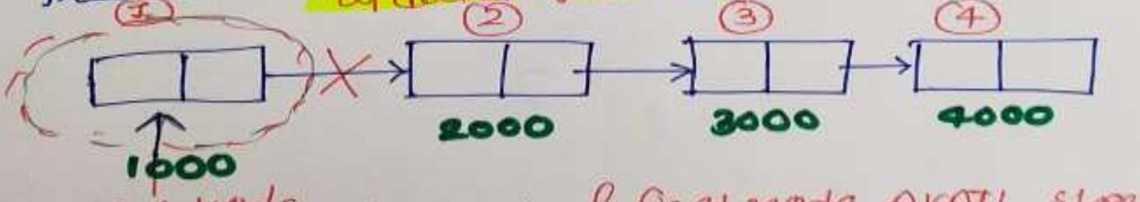


⇒ Create New_node

$\boxed{\text{New\_node} = \text{AVAIL}(1000)}$

so, now AVAIL & New_node both pointing to first node of free list but AVAIL should point first node and now its become new_node so remove from the free List.

updated free list



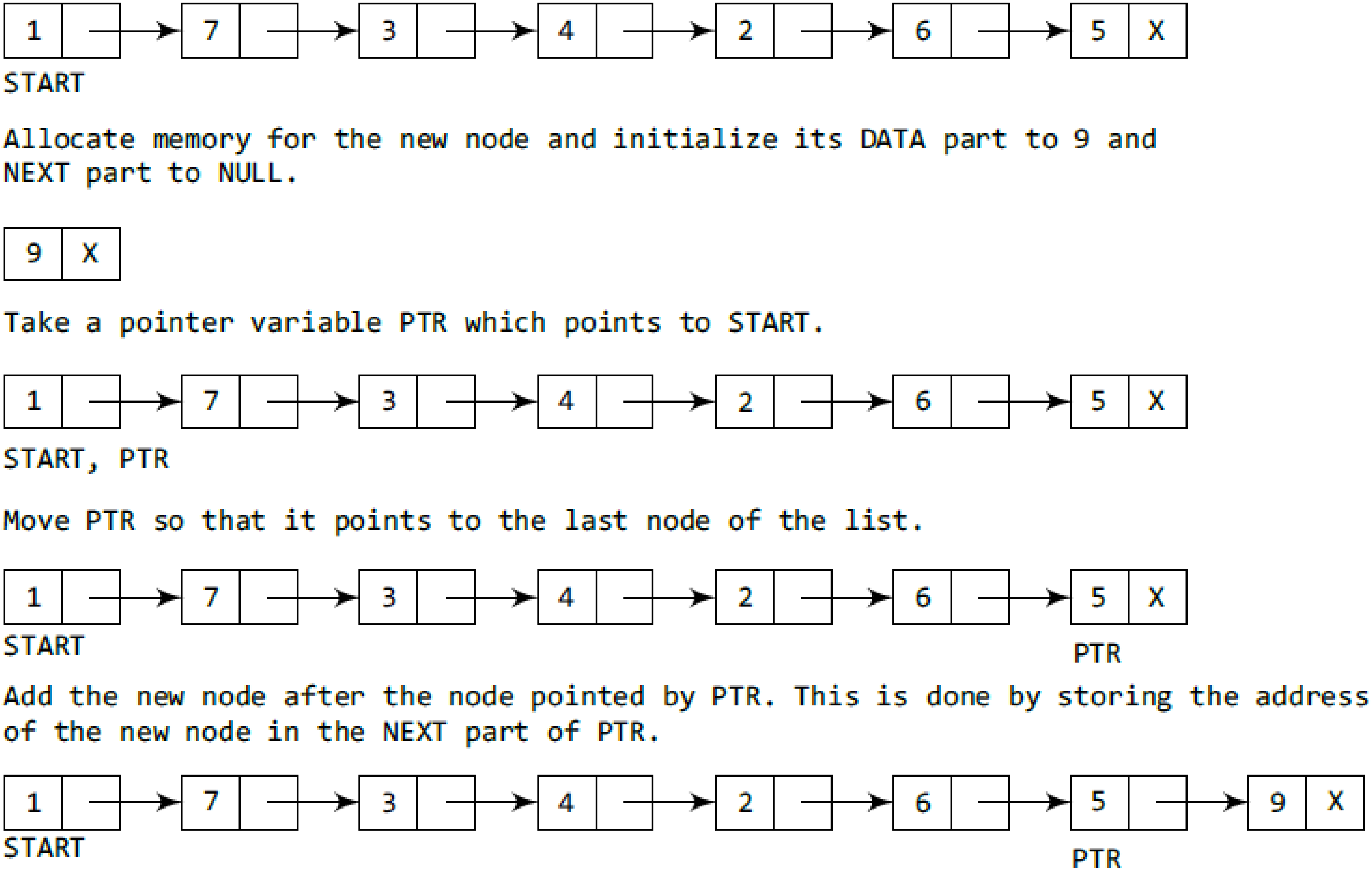
new_node →instead of first node AVAIL should point to the 2nd node as now it is first node

SO,

$\boxed{\text{AVAIL} = \text{AVAIL} \rightarrow \text{Next}}$

{insertion at the beginning}
new_node → DATA = VAL
new node → next = START
START = new_node.

Inserting a Node at the End



Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.

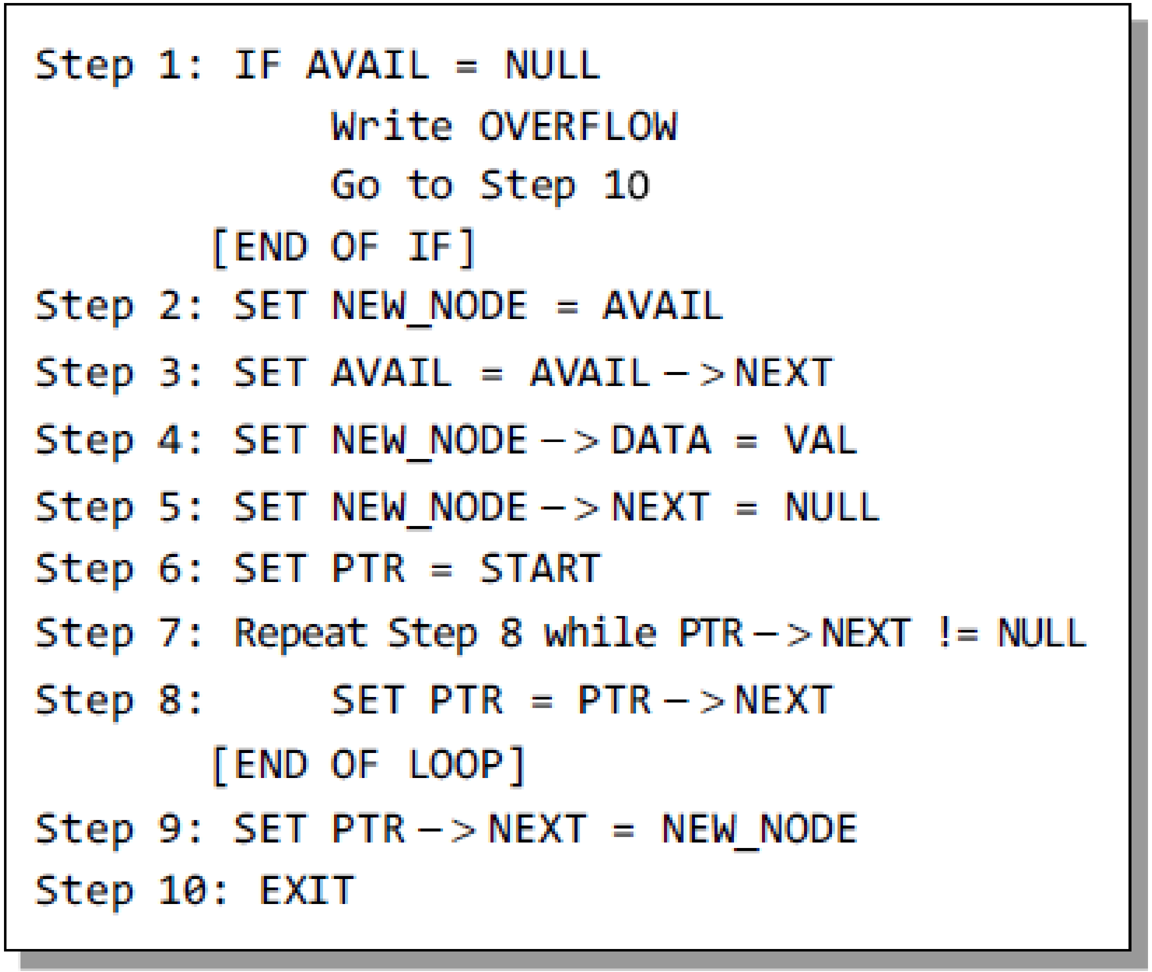
Take a pointer variable PTR which points to START.

Move PTR so that it points to the last node of the list.

Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the NEXT part of PTR.

Figure 6.14 Inserting an element at the end of a linked list

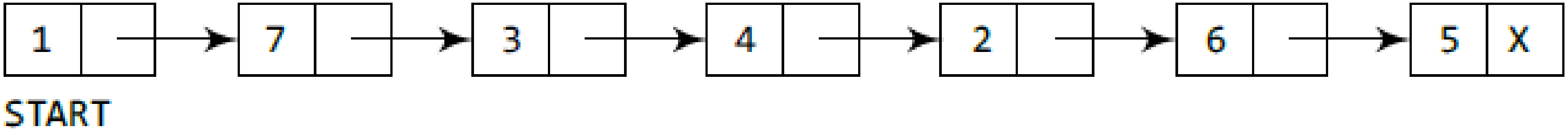
Inserting a Node at the End



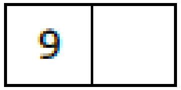
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET NEW_NODE->NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR->NEXT != NULL
Step 8:     SET PTR = PTR->NEXT
       [END OF LOOP]
Step 9: SET PTR->NEXT = NEW_NODE
Step 10: EXIT
```

Figure 6.15 Algorithm to insert a new node at the end

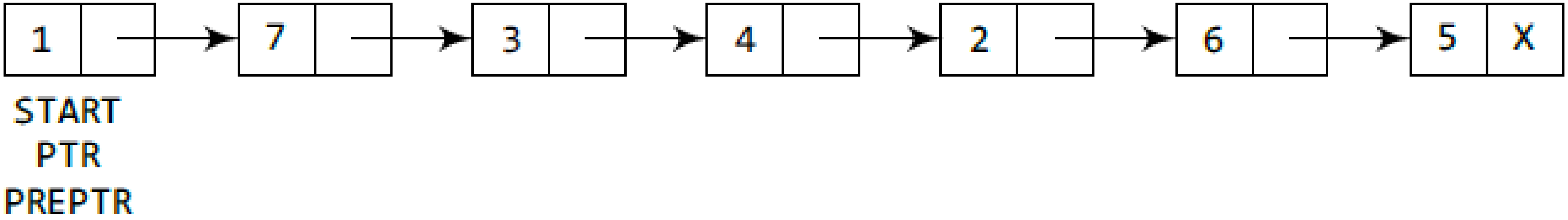
Inserting a Node After a Given Node



Allocate memory for the new node and initialize its DATA part to 9.

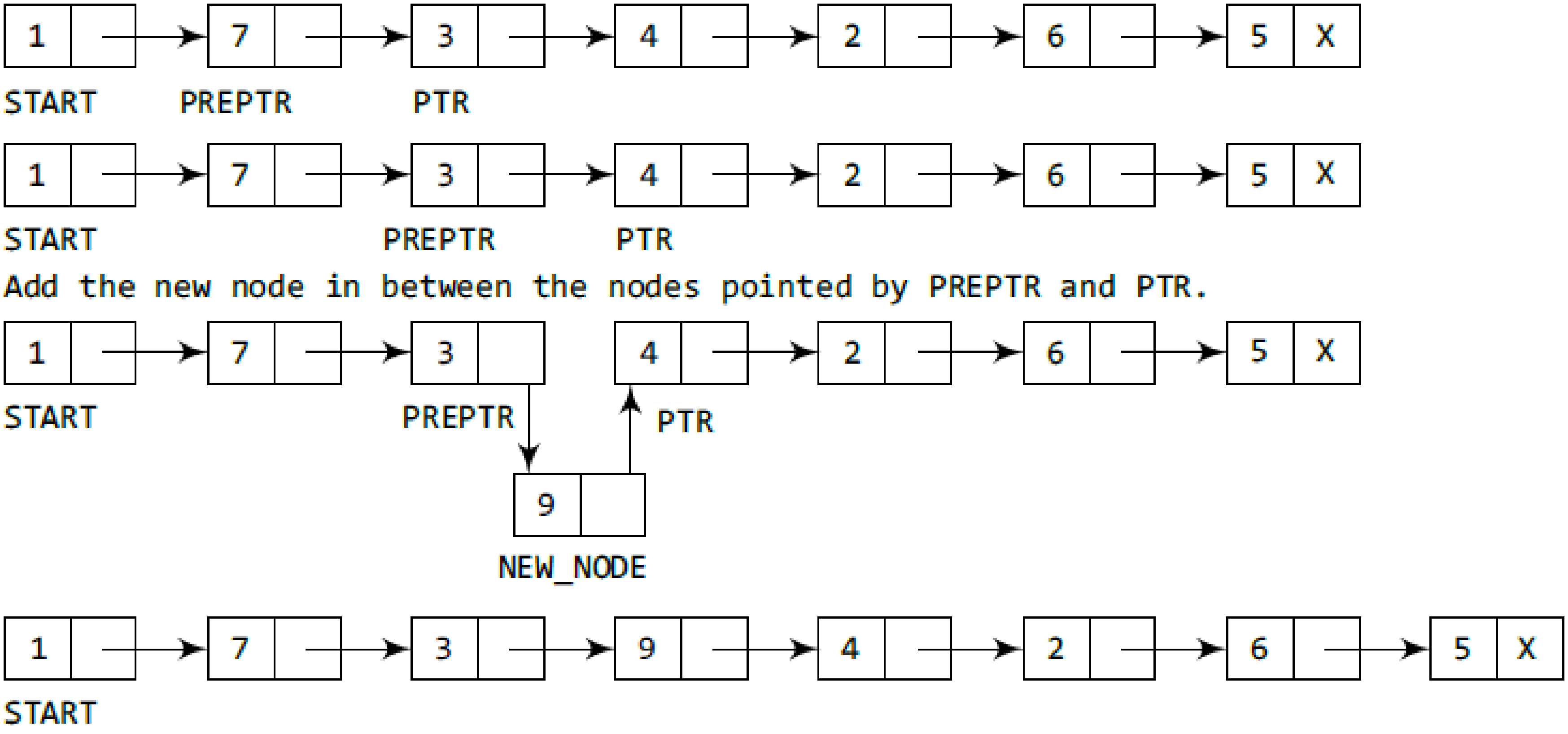


Take two pointer variables PTR and PREPTR and initialize them with START so that START, PTR, and PREPTR point to the first node of the list.



Move PTR and PREPTR until the DATA part of PREPTR = value of the node after which insertion has to be done. PREPTR will always point to the node just before PTR.

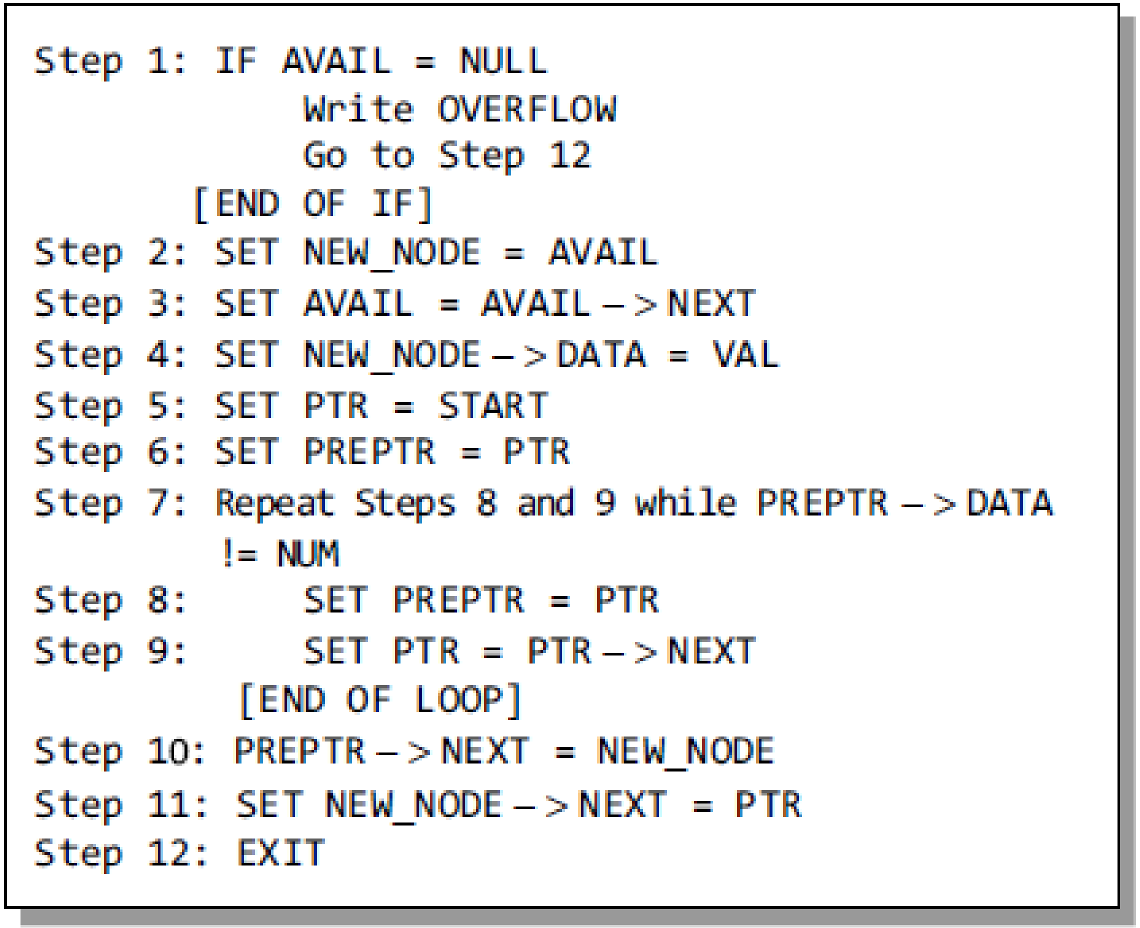
Inserting a Node After a Given Node



Add the new node in between the nodes pointed by PREPTR and PTR.

Figure 6.17 Inserting an element after a given node in a linked list

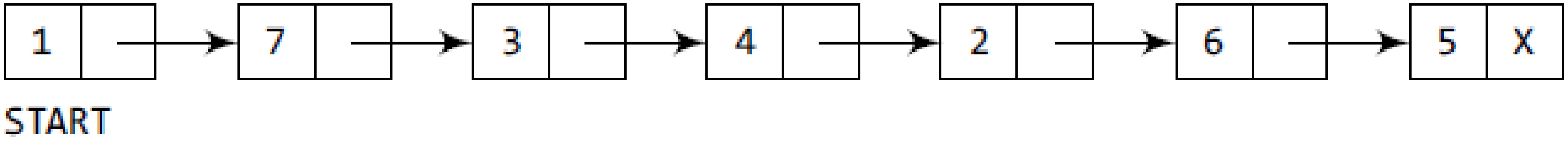
Inserting a Node After a Given Node



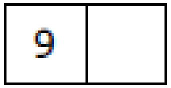
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
         != 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
```

Figure 6.16 Algorithm to insert a new node after a node that has value NUM

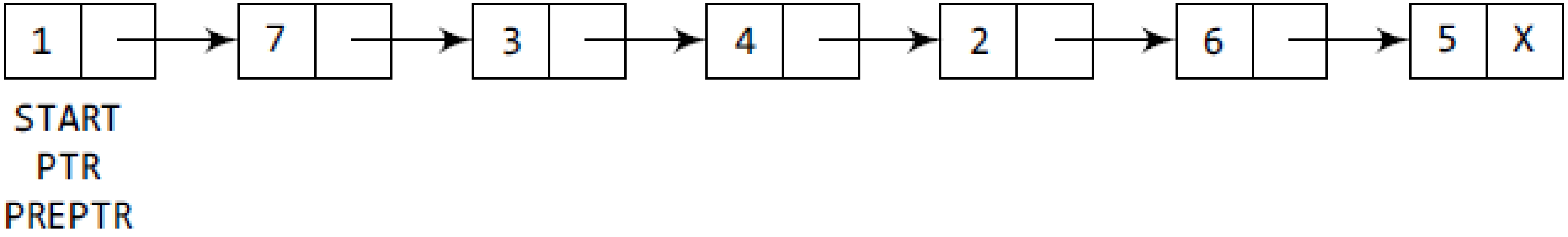
Inserting a Node Before a Given Node



Allocate memory for the new node and initialize its DATA part to 9.

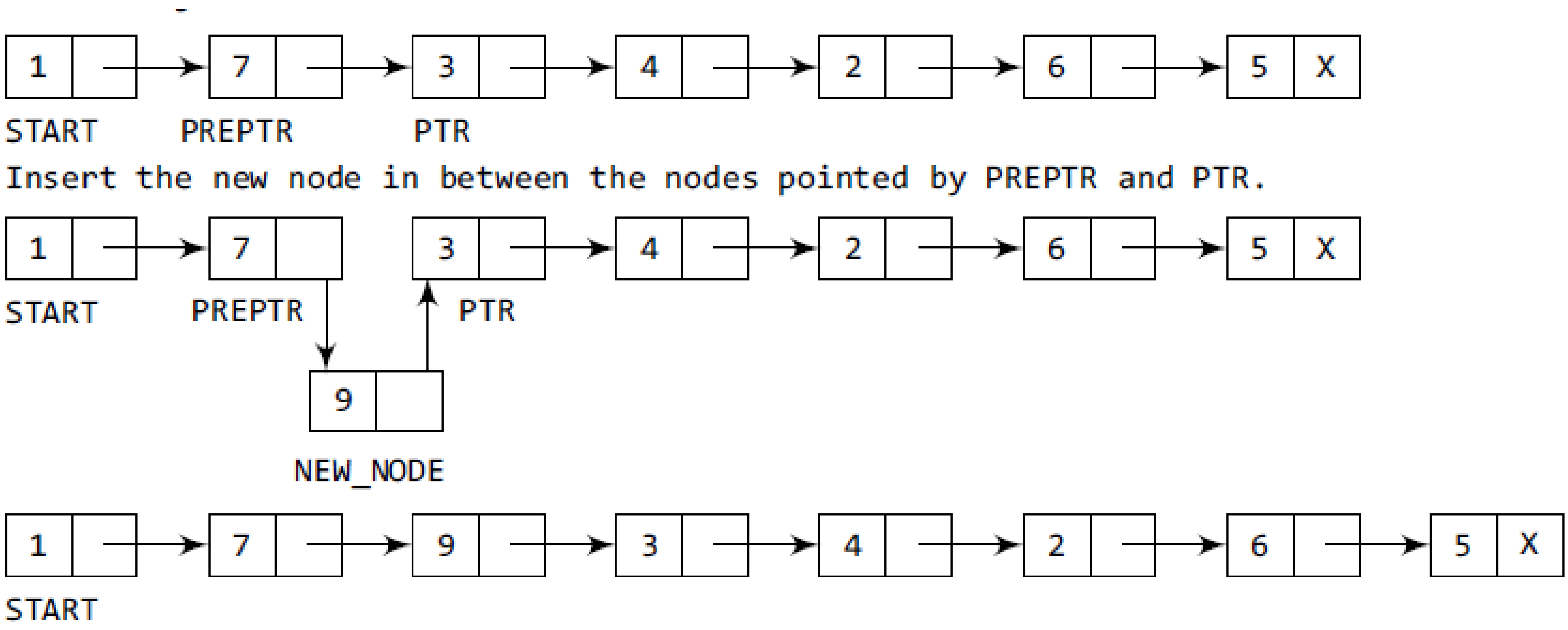


Initialize PREPTR and PTR to the START node.



Move PTR and PREPTR until the DATA part of PTR = value of the node before which insertion has to be done. PREPTR will always point to the node just before PTR.

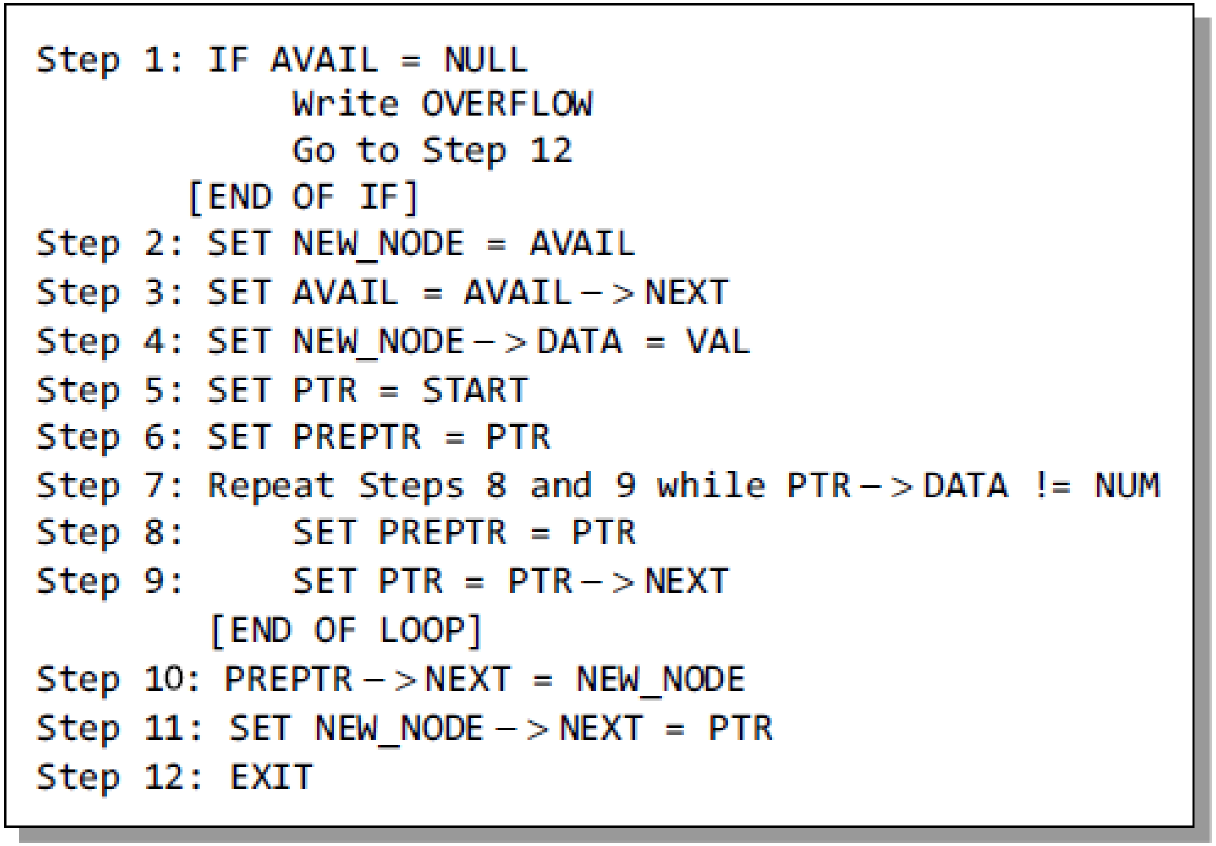
Inserting a Node Before a Given Node



Insert the new node in between the nodes pointed by PREPTR and PTR.

Figure 6.19 Inserting an element before a given node in a linked list

Inserting a Node Before a Given Node



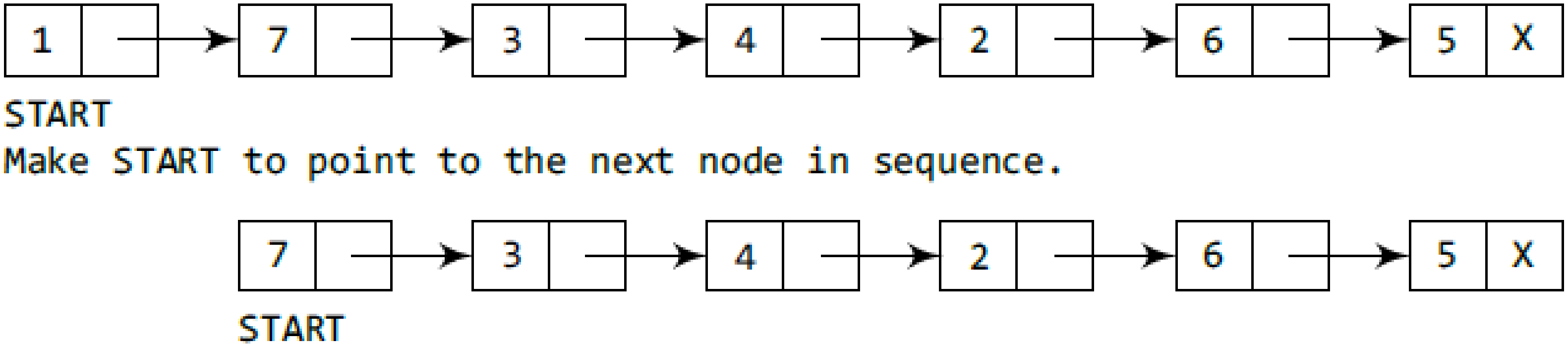
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR -> DATA != NUM
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
```

Figure 6.18 Algorithm to insert a new node before a node that has value NUM

Deleting a Node from a Linked List

- Case 1: The first node is deleted.
- Case 2: The last node is deleted.
- Case 3: The node after a given node is deleted.
- Underflow is a condition that occurs when we try to delete a node from a linked list that is empty.
- This happens when START = NULL or when there are no more nodes to delete.
- when we delete a node from a linked list, we actually have to free the memory occupied by that node.
- The memory is returned to the free pool so that it can be used to store other programs and data.

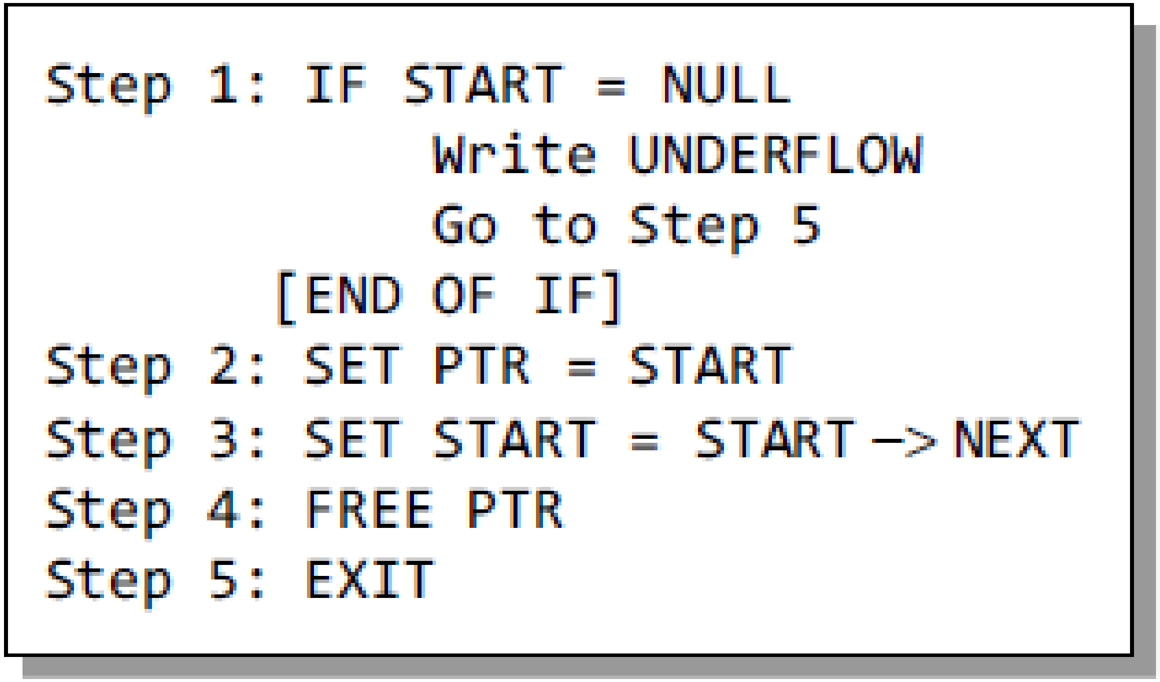
Deleting the First Node



Make START to point to the next node in sequence.

Figure 6.20 Deleting the first node of a linked list

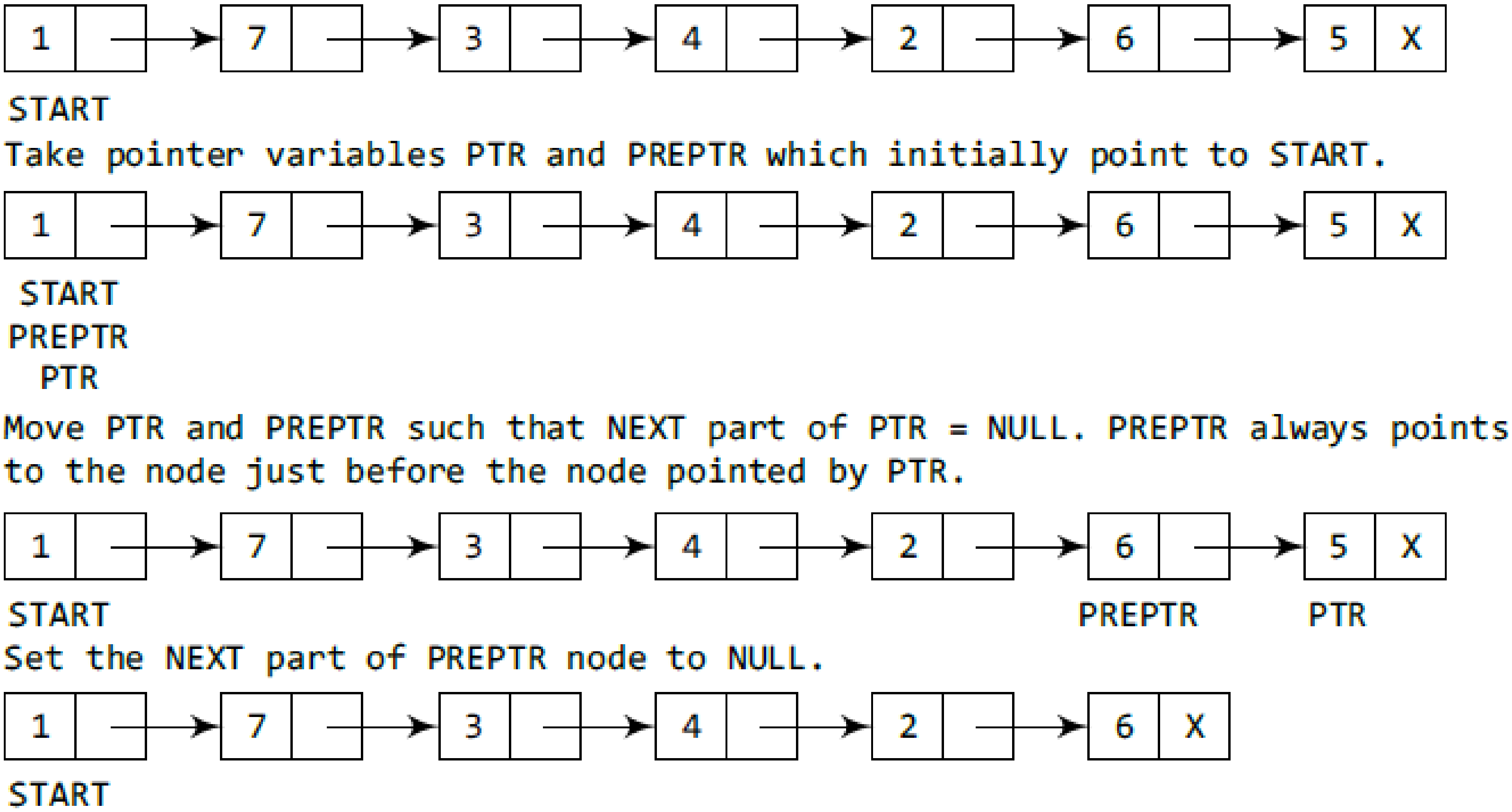
Deleting the First Node



```
Step 1: IF START = NULL
            Write UNDERFLOW
            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
```

Figure 6.21 Algorithm to delete the first node

Deleting the Last Node



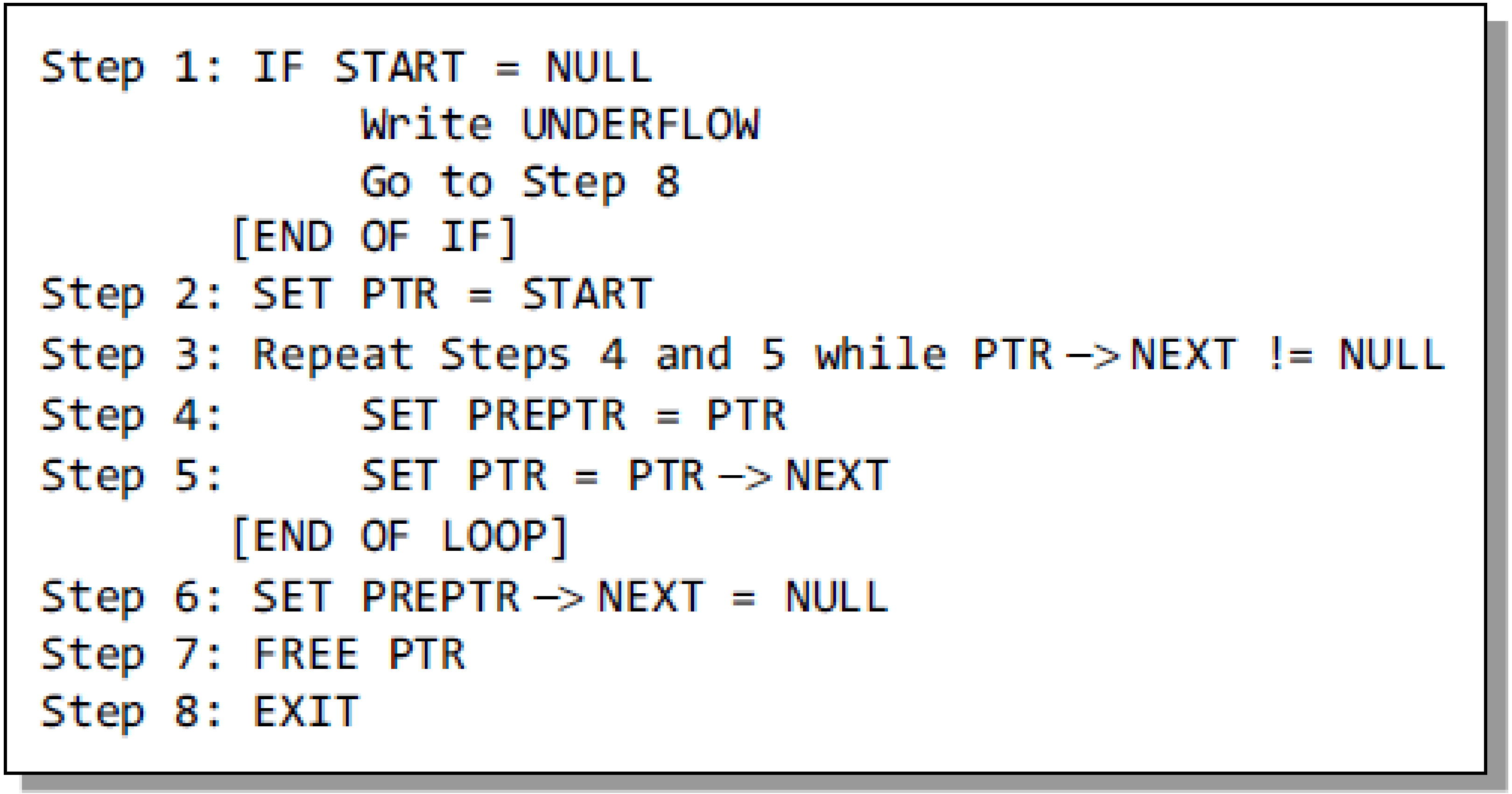
Take pointer variables PTR and PREPTR which initially point to START.

Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.

Set the NEXT part of PREPTR node to NULL.

Figure 6.22 Deleting the last node of a linked list

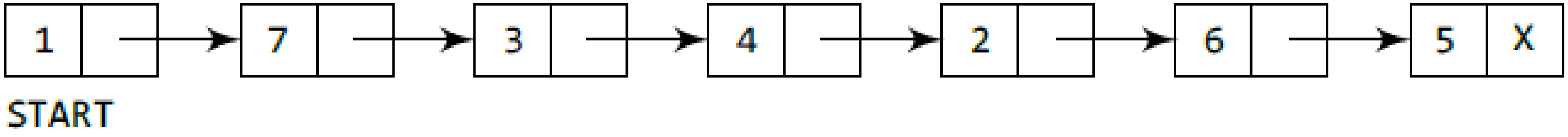
Deleting the Last Node



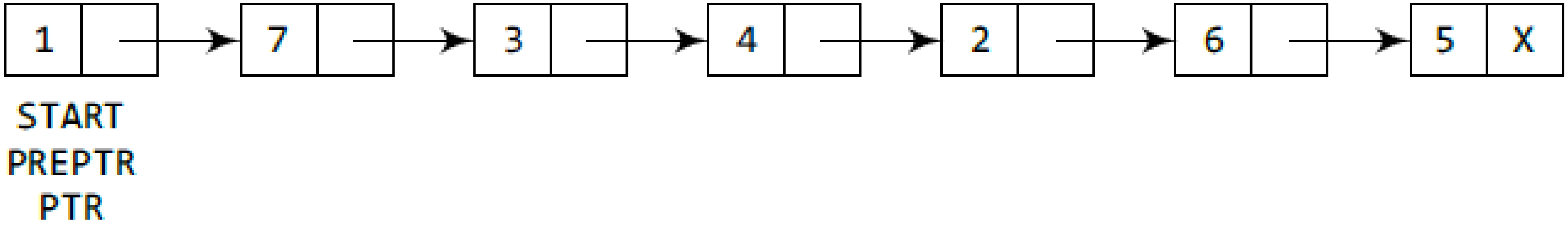
```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:       SET PREPTR = PTR
Step 5:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT =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

Figure 6.23 Algorithm to delete the last node

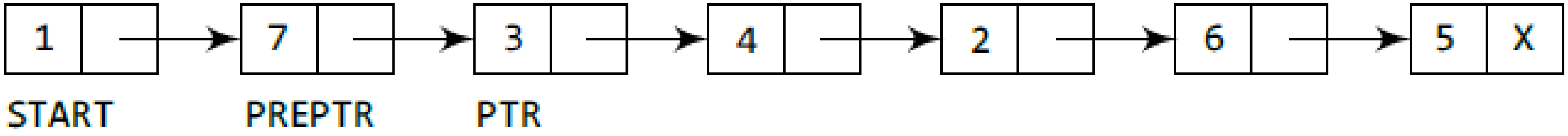
Deleting the Node After a Given Node



Take pointer variables PTR and PREPTR which initially point to START.



Move PREPTR and PTR such that PREPTR points to the node containing VAL and PTR points to the succeeding node.



Deleting the Node After a Given Node

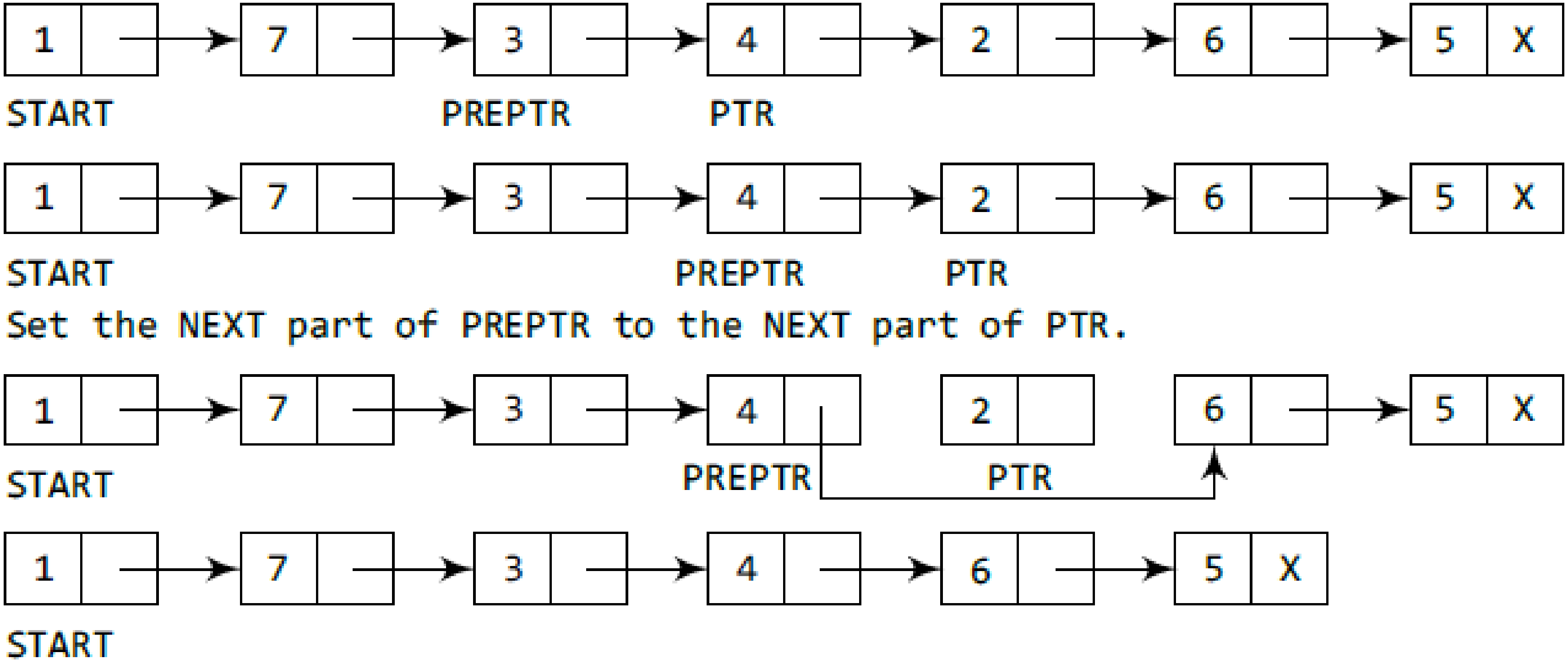
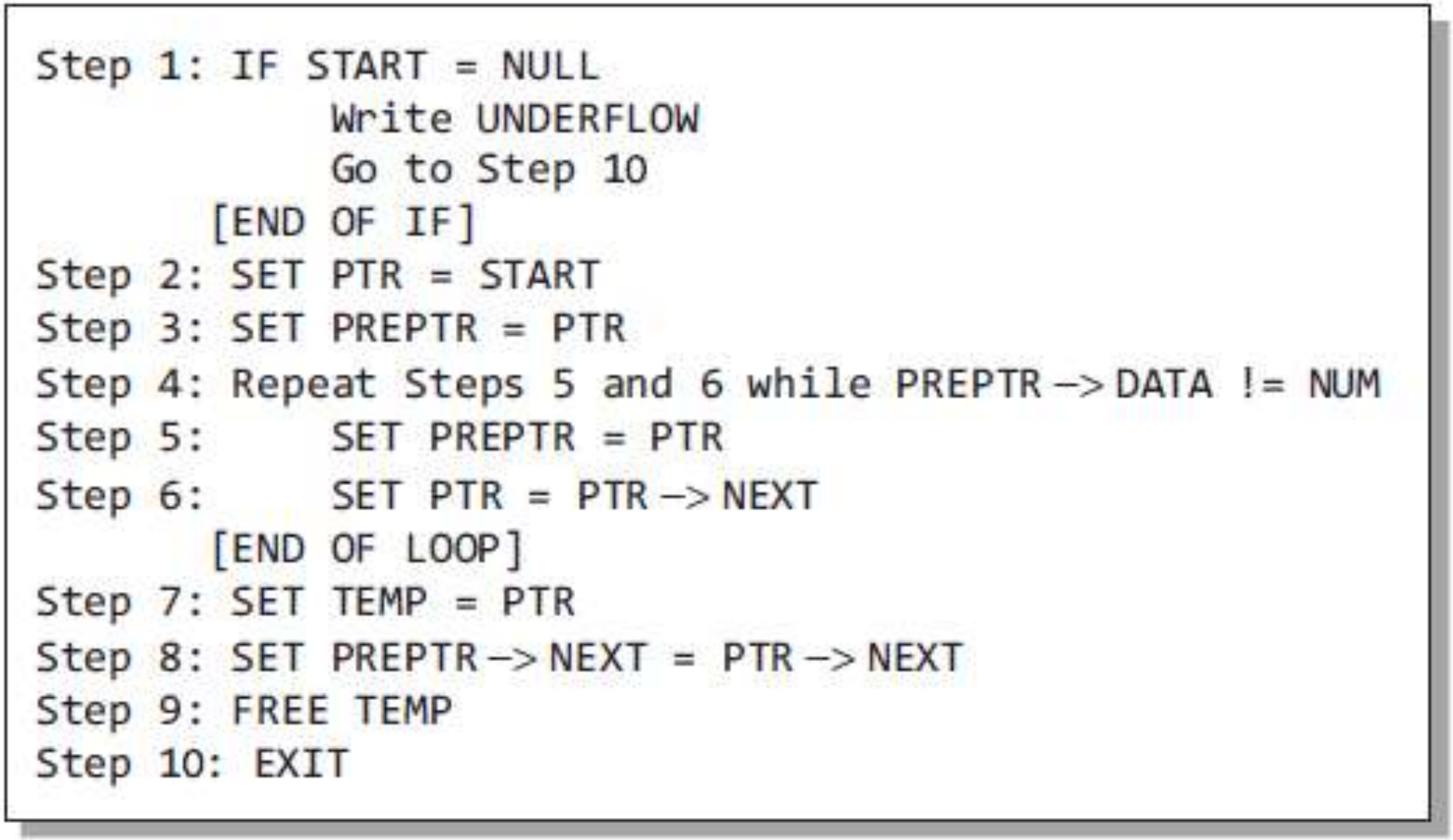


Figure 6.24 Deleting the node after a given node in a linked list

Deleting the Node After a Given Node



```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 10
       [END OF IF]
Step 2: SET PTR = START
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
```

Figure 6.25 Algorithm to delete the node after a given node

CIRCULAR LINKED LISTs

- In a circular linked list, the last node contains a pointer to the first node of the list.
- We can have a circular singly linked list as well as a circular doubly linked list. While traversing a circular doubly linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started.
- Thus, a circular linked list has no beginning and no ending. Figure 6.26 shows a circular linked list.

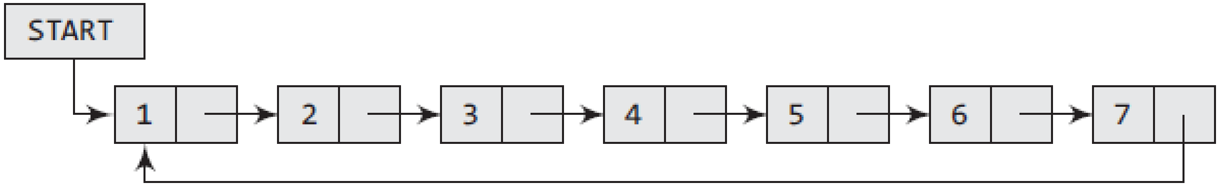


Figure 6.26 Circular linked list

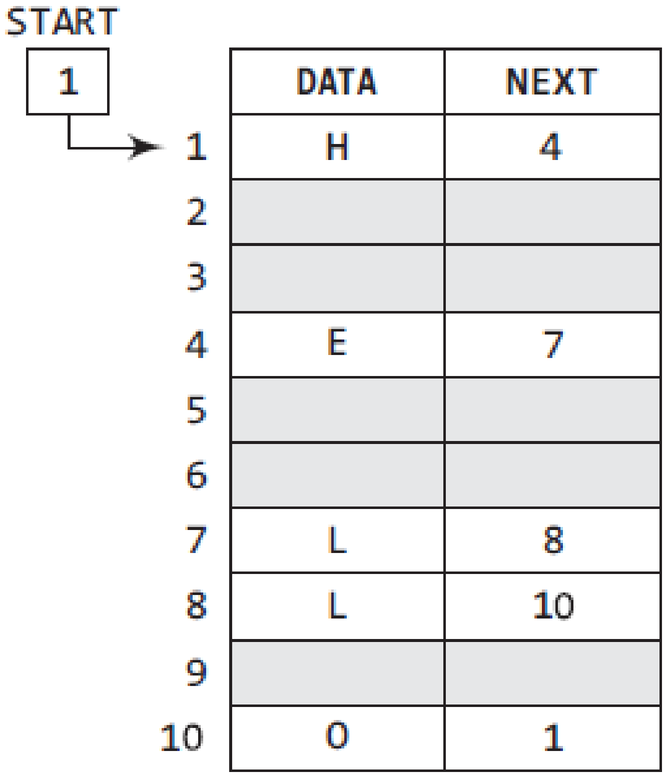
CIRCULAR LINKED LIST

- The only downside of a circular linked list is the complexity of iteration. Note that there are no NULL values in the NEXT part of any of the nodes of list.
- We can traverse the list until we find the NEXT entry that contains the address of the first node of the list. This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list. When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in Fig. 6.27 stores characters that when put together form the word HELLO.

CIRCULAR LINKED LIST

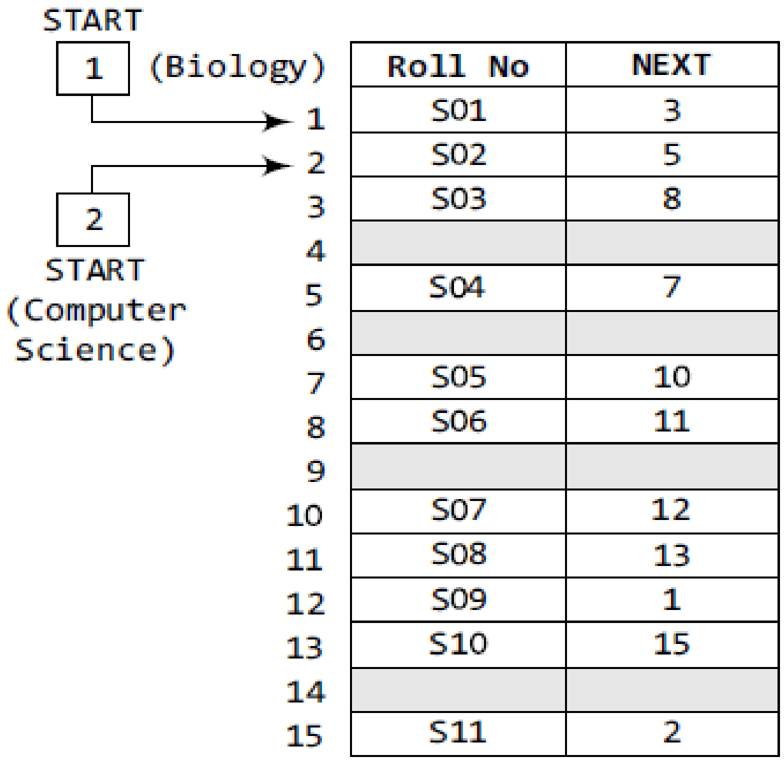
- Now, look at Fig. 6.28. Two different linked lists are simultaneously maintained in the memory. There is no ambiguity in traversing through the list because each list maintains a separate START pointer which gives the address of the first node of the respective linked list. The remaining nodes are reached by looking at the value stored in NEXT.
- By looking at the figure, we can conclude that the roll numbers of the students who have opted for Biology are S01, S03, S06, S08, S10, and S11. Similarly, the roll numbers of the students who chose Computer Science are S02, S04, S05, S07, and S09.

CIRCULAR LINKED LIST



	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	1

Figure 6.27 Memory representation of a circular linked list



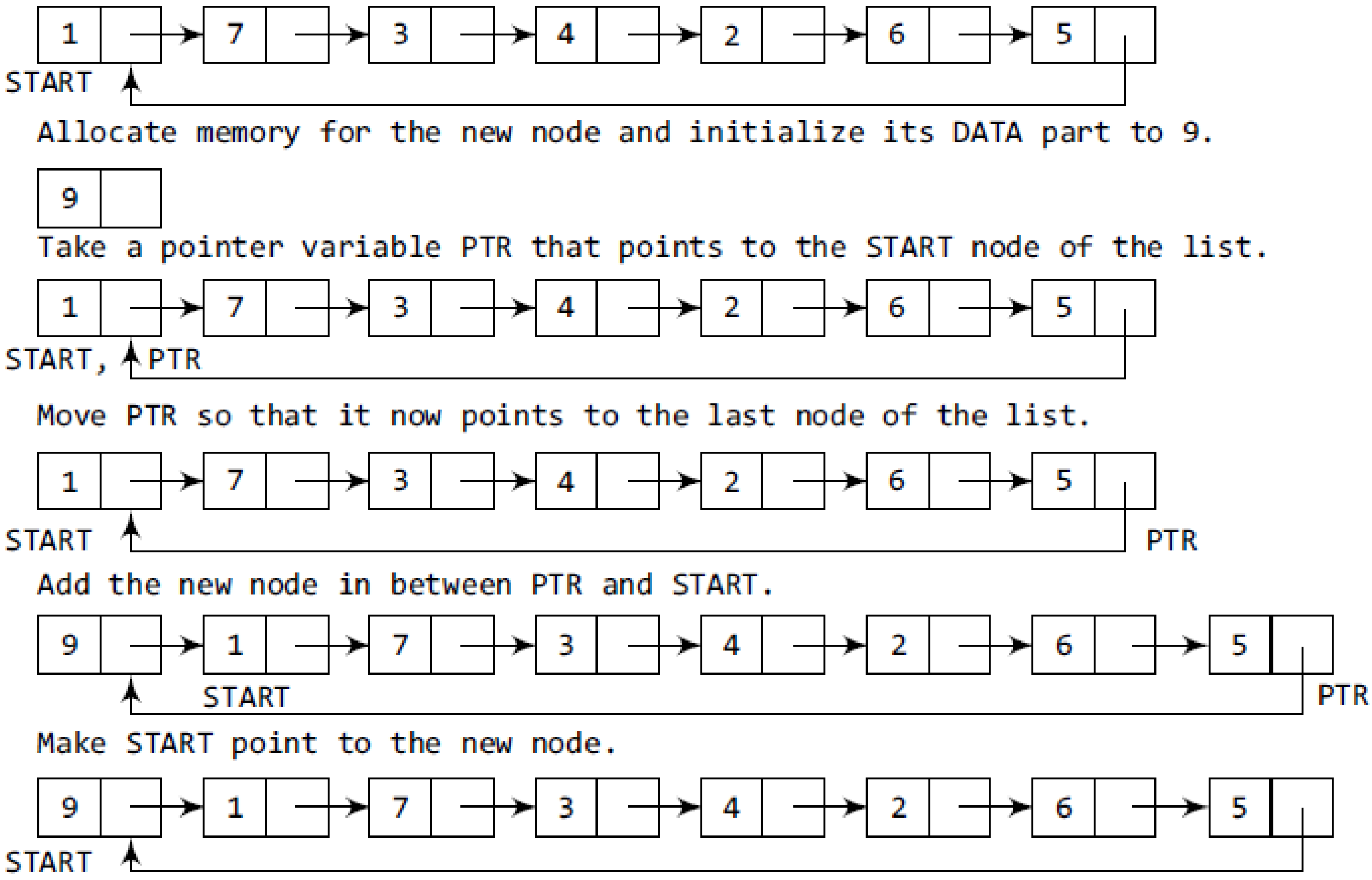
	Roll No	NEXT
1	S01	3
2	S02	5
3	S03	8
4		
5	S04	7
6		
7	S05	10
8	S06	11
9		
10	S07	12
11	S08	13
12	S09	1
13	S10	15
14		
15	S11	2

Figure 6.28 Memory representation of two circular linked lists stored in the memory

Inserting a New Node in a Circular Linked List

- We will take two cases and then see how insertion is done in each case.
- Case 1: The new node is inserted at the beginning of the circular linked list.
- Case 2: The new node is inserted at the end of the circular linked list.

Inserting a Node at the Beginning of a Circular Linked List



Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable PTR that points to the START node of the list.

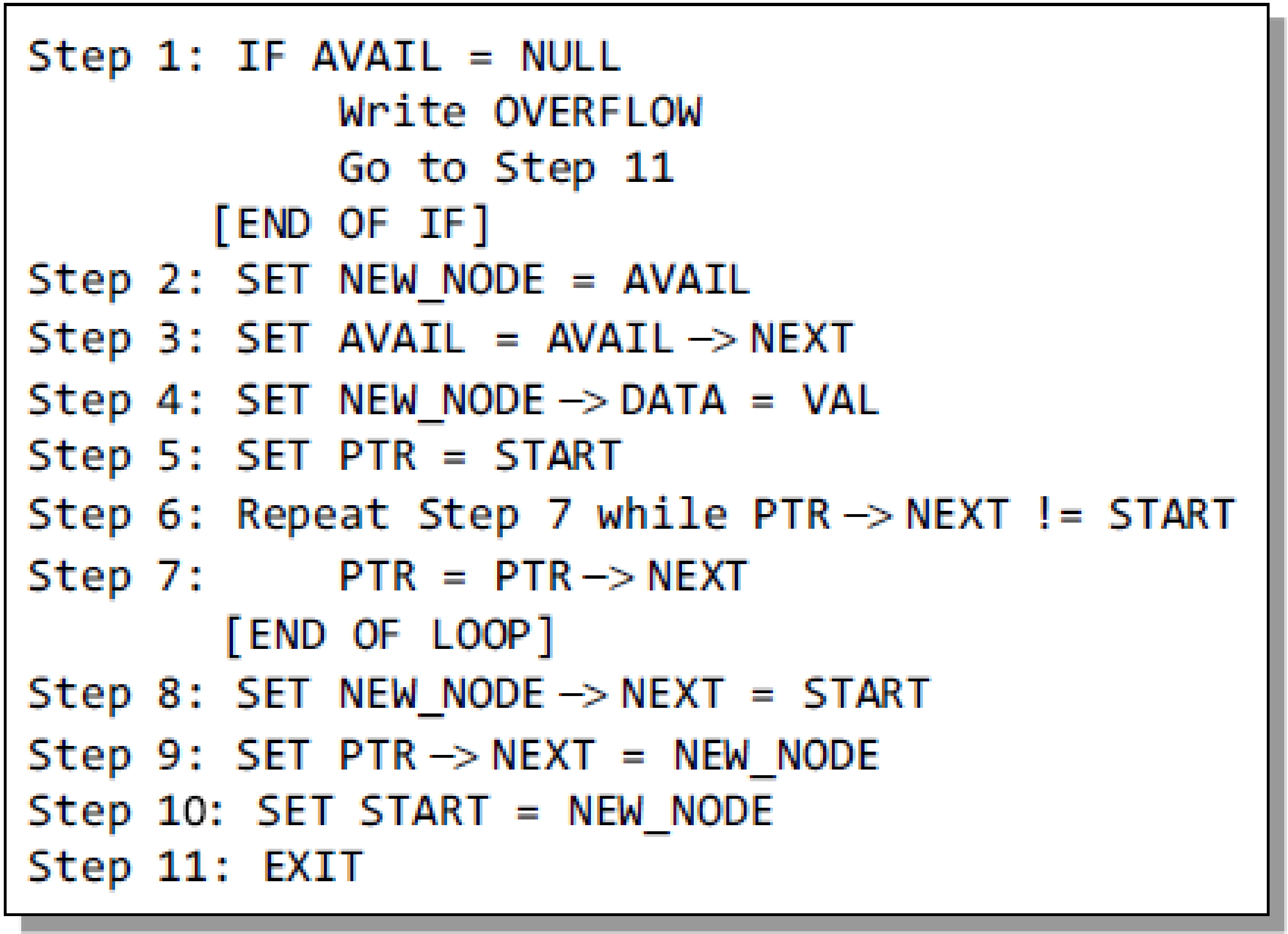
Move PTR so that it now points to the last node of the list.

Add the new node in between PTR and START.

Make START point to the new node.

Figure 6.29 Inserting a new node at the beginning of a circular linked list

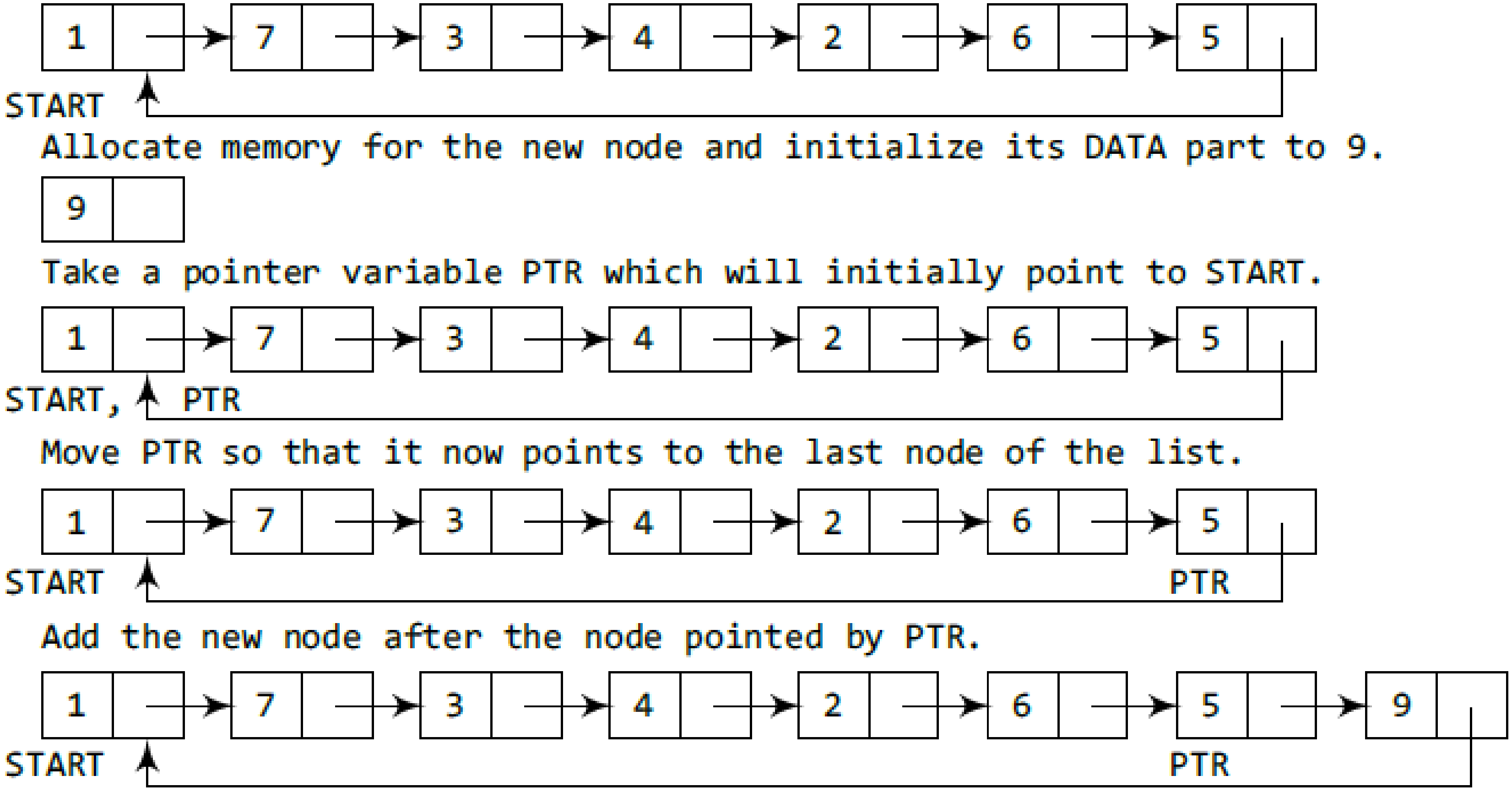
Inserting a Node at the Beginning of a Circular Linked List



```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 11
        [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: Repeat Step 7 while PTR -> NEXT != START
Step 7:       PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Figure 6.30 Algorithm to insert a new node at the beginning

Inserting a Node at the End of a Circular Linked List



Allocate memory for the new node and initialize its DATA part to 9.

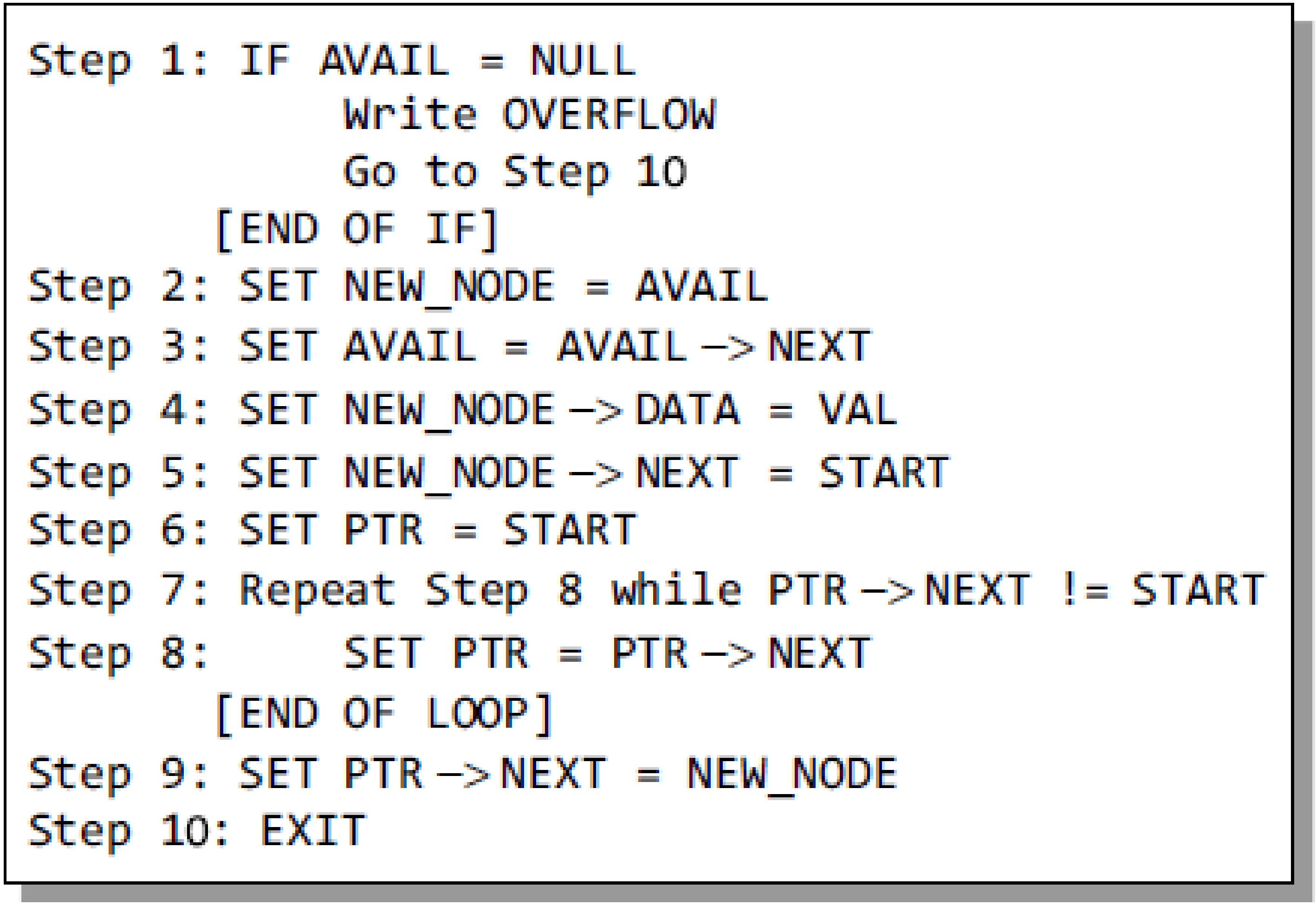
Take a pointer variable PTR which will initially point to START.

Move PTR so that it now points to the last node of the list.

Add the new node after the node pointed by PTR.

Figure 6.31 Inserting a new node at the end of a circular linked list

Inserting a Node at the End of a Circular Linked List



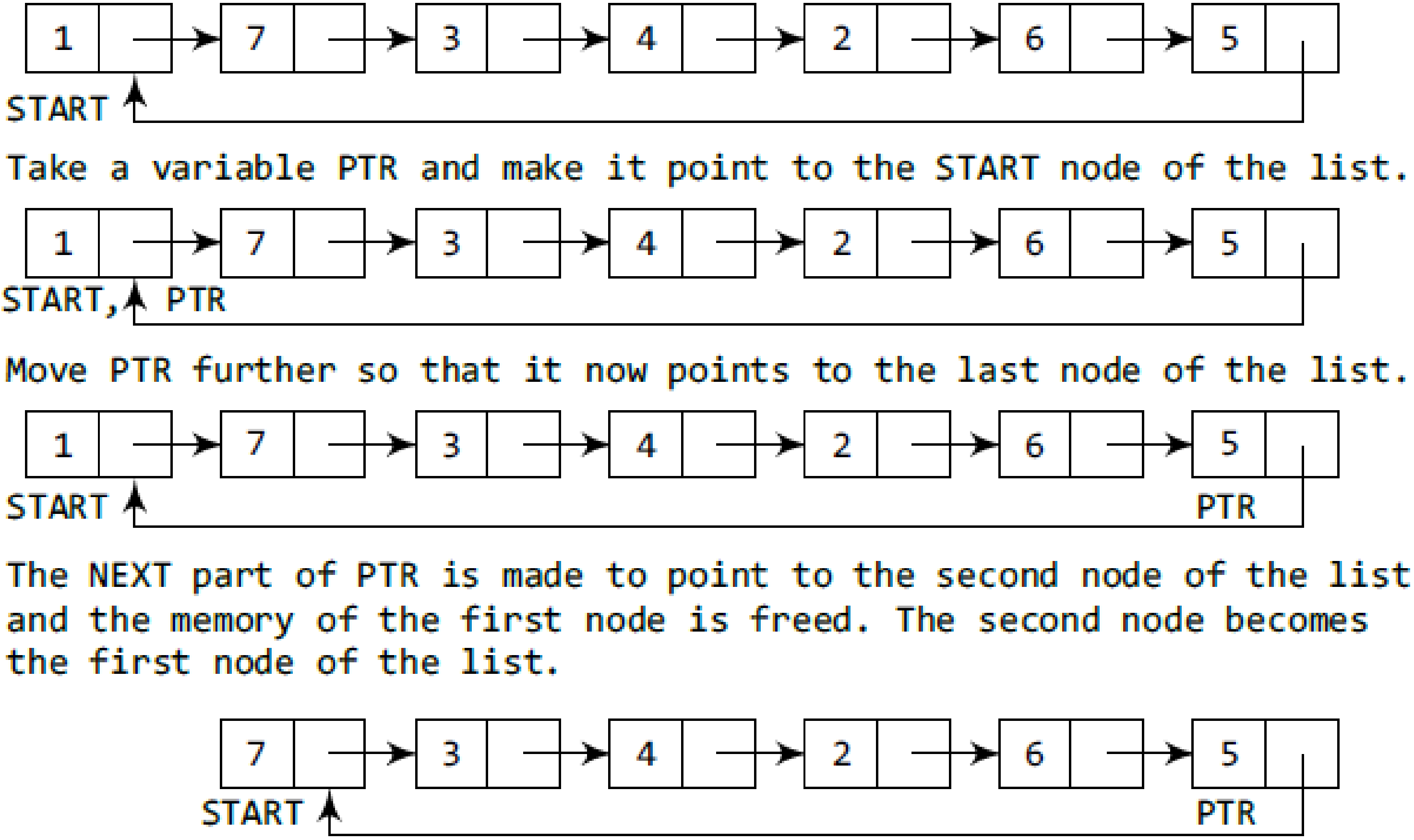
```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Figure 6.32 Algorithm to insert a new node at the end

Deleting a Node from a Circular Linked List

- We will take two cases and then see how deletion is done in each case. Rest of the cases of deletion are same as that given for singly linked lists.
- Case 1: The first node is deleted.
- Case 2: The last node is deleted.

Deleting the First Node from a Circular Linked List



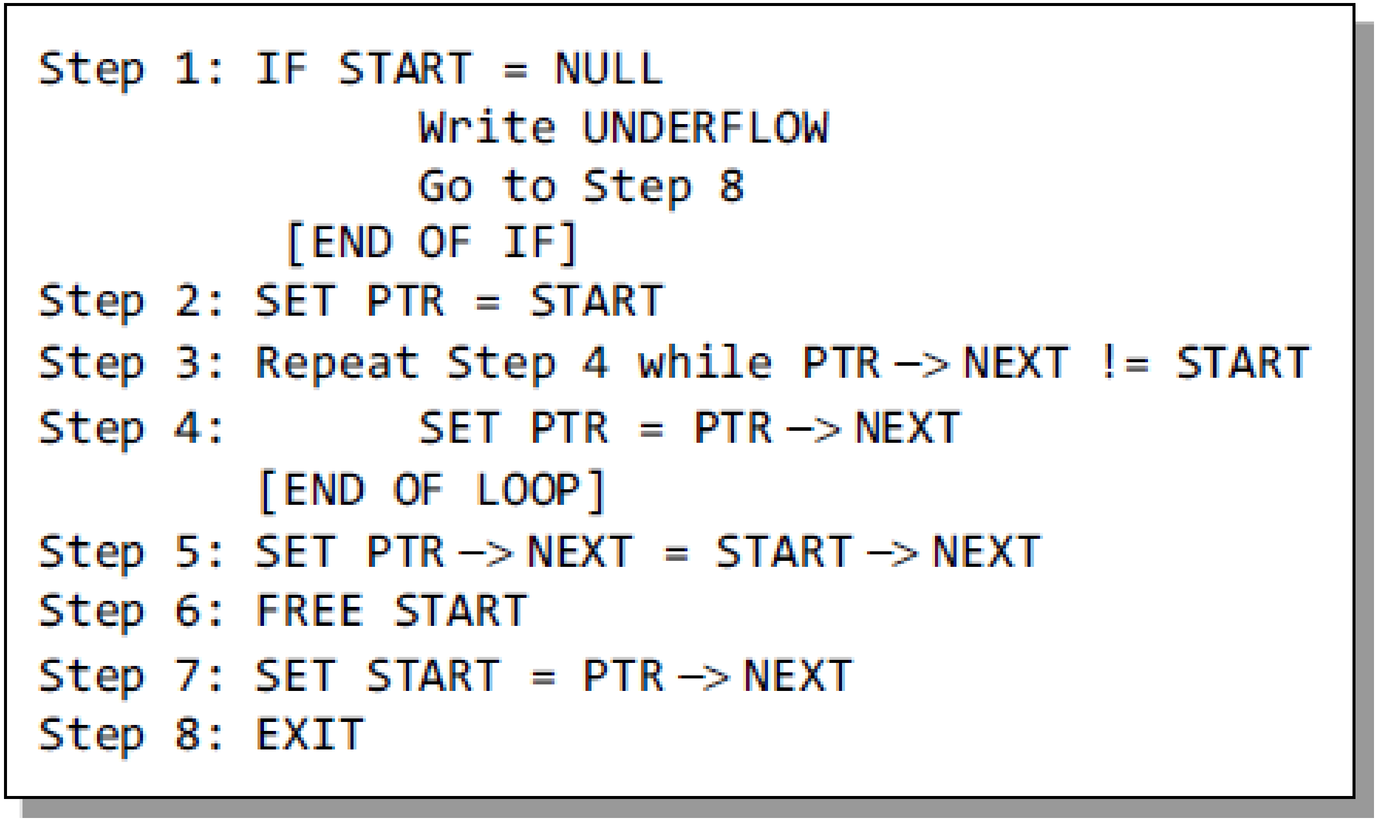
Take a variable PTR and make it point to the START node of the list.

Move PTR further so that it now points to the last node of the list.

The NEXT part of PTR is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.

Figure 6.33 Deleting the first node from a circular linked list

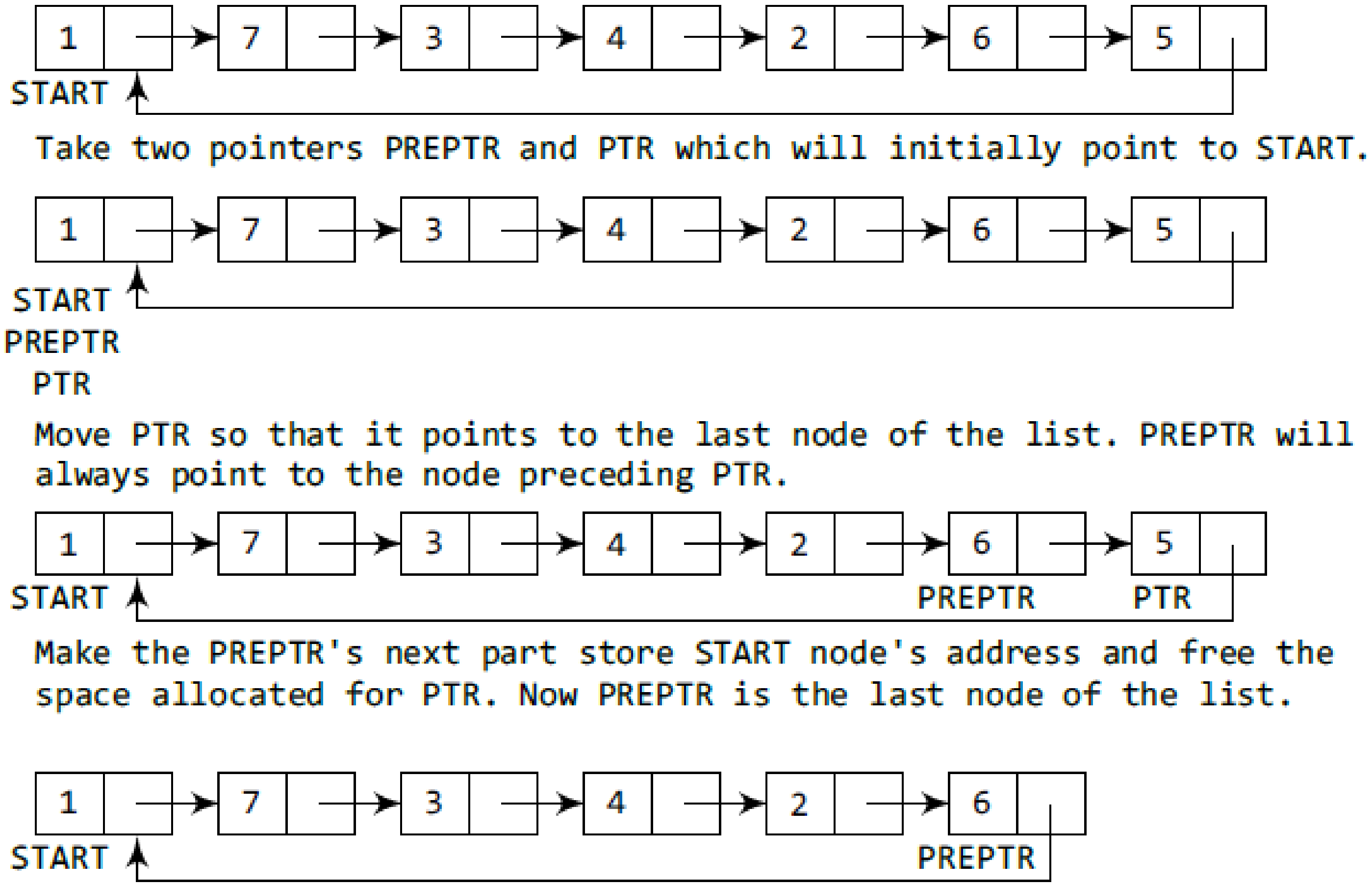
Deleting the First Node from a Circular Linked List



```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
         [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START
Step 7: SET START = PTR -> NEXT
Step 8: EXIT
```

Figure 6.34 Algorithm to delete the first node

Deleting the Last Node from a Circular Linked List



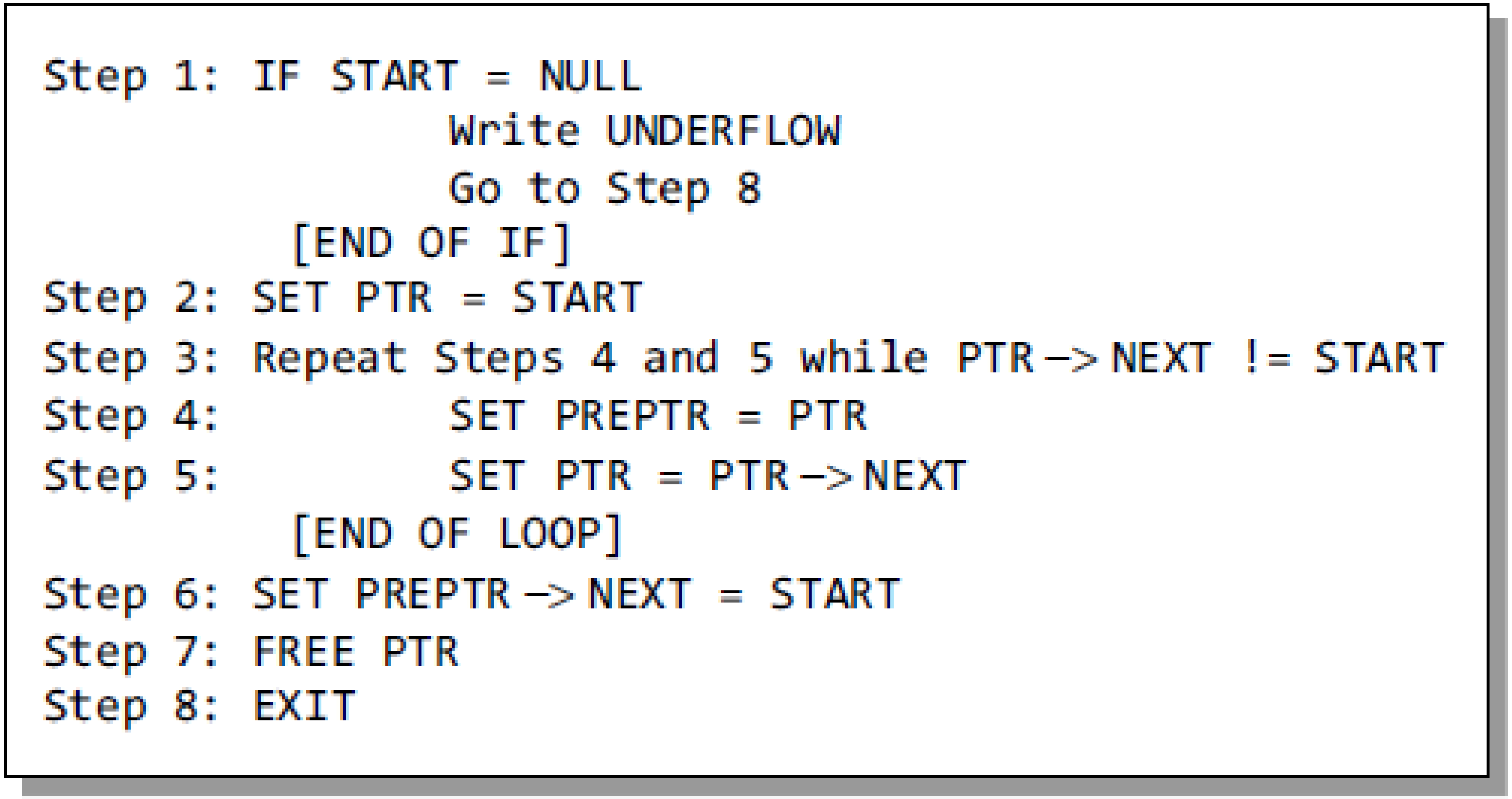
Take two pointers PREPTR and PTR which will initially point to START.

Move PTR so that it points to the last node of the list. PREPTR will always point to the node preceding PTR.

Make the PREPTR's next part store START node's address and free the space allocated for PTR. Now PREPTR is the last node of the list.

Figure 6.35 Deleting the last node from a circular linked list

Deleting the Last Node from a Circular Linked List



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

Figure 6.36 Algorithm to delete the last node

DOUBLY LINKED LISTS

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node as shown in Fig. 6.37.

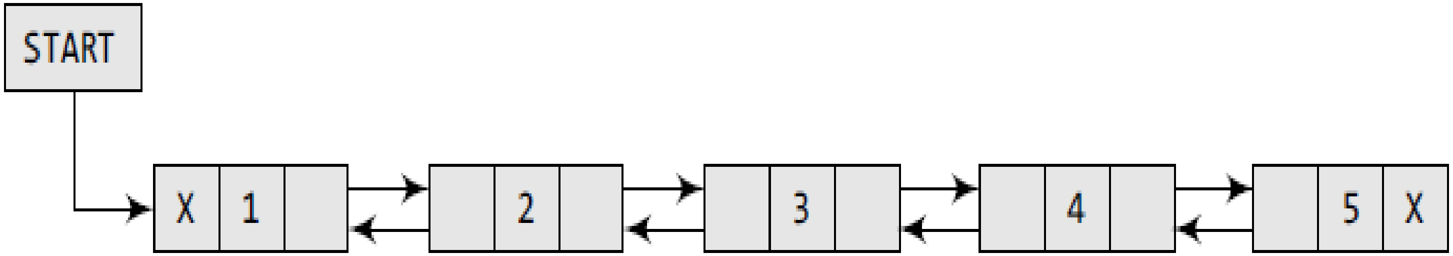
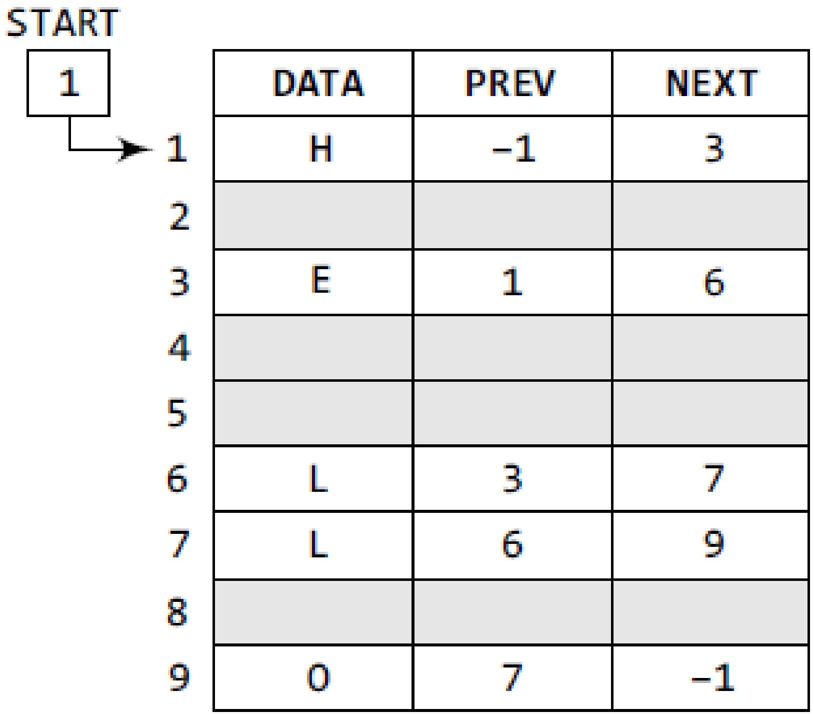


Figure 6.37 Doubly linked list

DOUBLY LINKED LISTS

- Doubly linked list calls for more space per node and more expensive basic operations. However, it provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward & backward).
- Its main advantage is that it makes searching twice as efficient.
- Let us view how a doubly linked list is maintained in the memory. Consider Fig. 6.38. In the figure, a variable START is used to store the address of the first node.
- In this example, START = 1, so the first data is stored at address 1, which is H. Since this is the first node, it has no previous node and hence stores NULL or –1 in the PREV field.
- We will traverse the list until we reach a position where the NEXT entry contains –1 or NULL. This denotes the end of the linked list.
- When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in the above example stores characters that when put together form the word HELLO.

DOUBLY LINKED LISTS



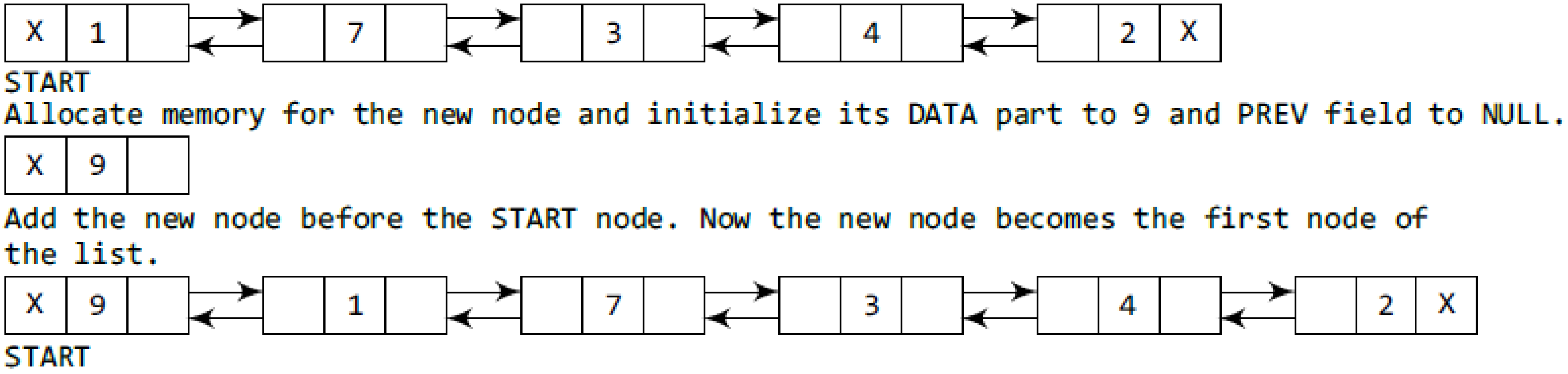
	DATA	PREV	NEXT
1	H	–1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	–1

Figure 6.38 Memory representation of a doubly linked list

Inserting a New Node in a Doubly Linked List

- We will take four cases and then see how insertion is done in each case.
- Case 1: The new node is inserted at the beginning.
- Case 2: The new node is inserted at the end.
- Case 3: The new node is inserted after a given node.
- Case 4: The new node is inserted before a given node.

Inserting a Node at the Beginning of a Doubly Linked List

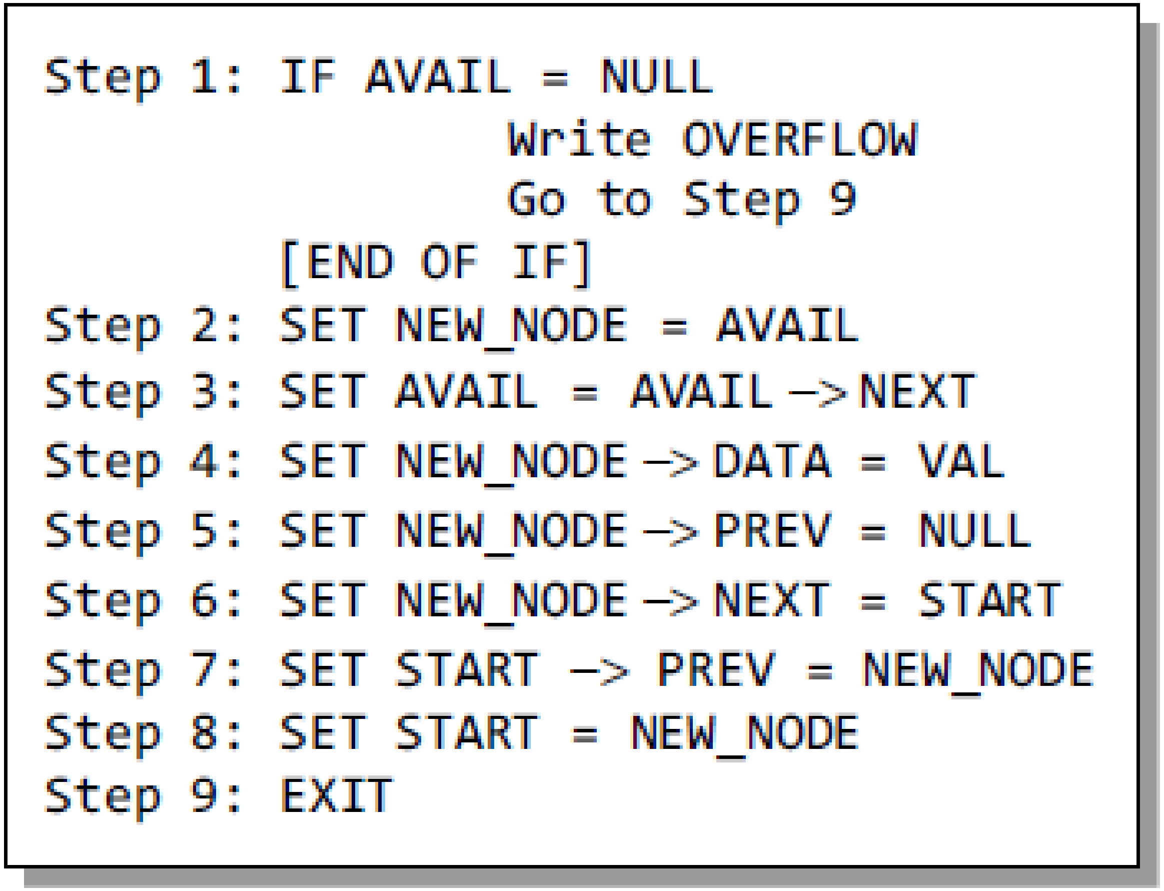


Allocate memory for the new node and initialize its DATA part to 9 and PREV field to NULL.

Add the new node before the START node. Now the new node becomes the first node of the list.

Figure 6.39 Inserting a new node at the beginning of a doubly linked list

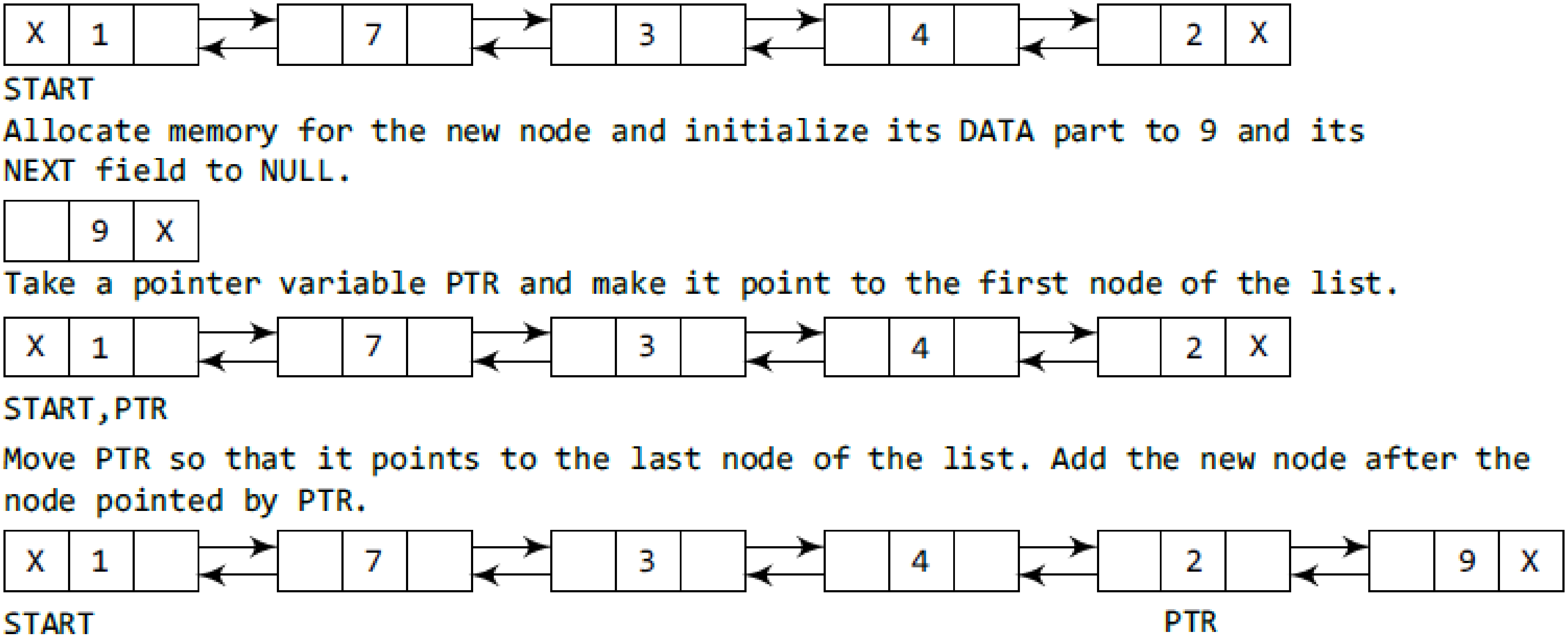
Inserting a Node at the Beginning of a Doubly Linked List



```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [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 -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Figure 6.40 Algorithm to insert a new node at the beginning

Inserting a Node at the End end of a Doubly Linked List



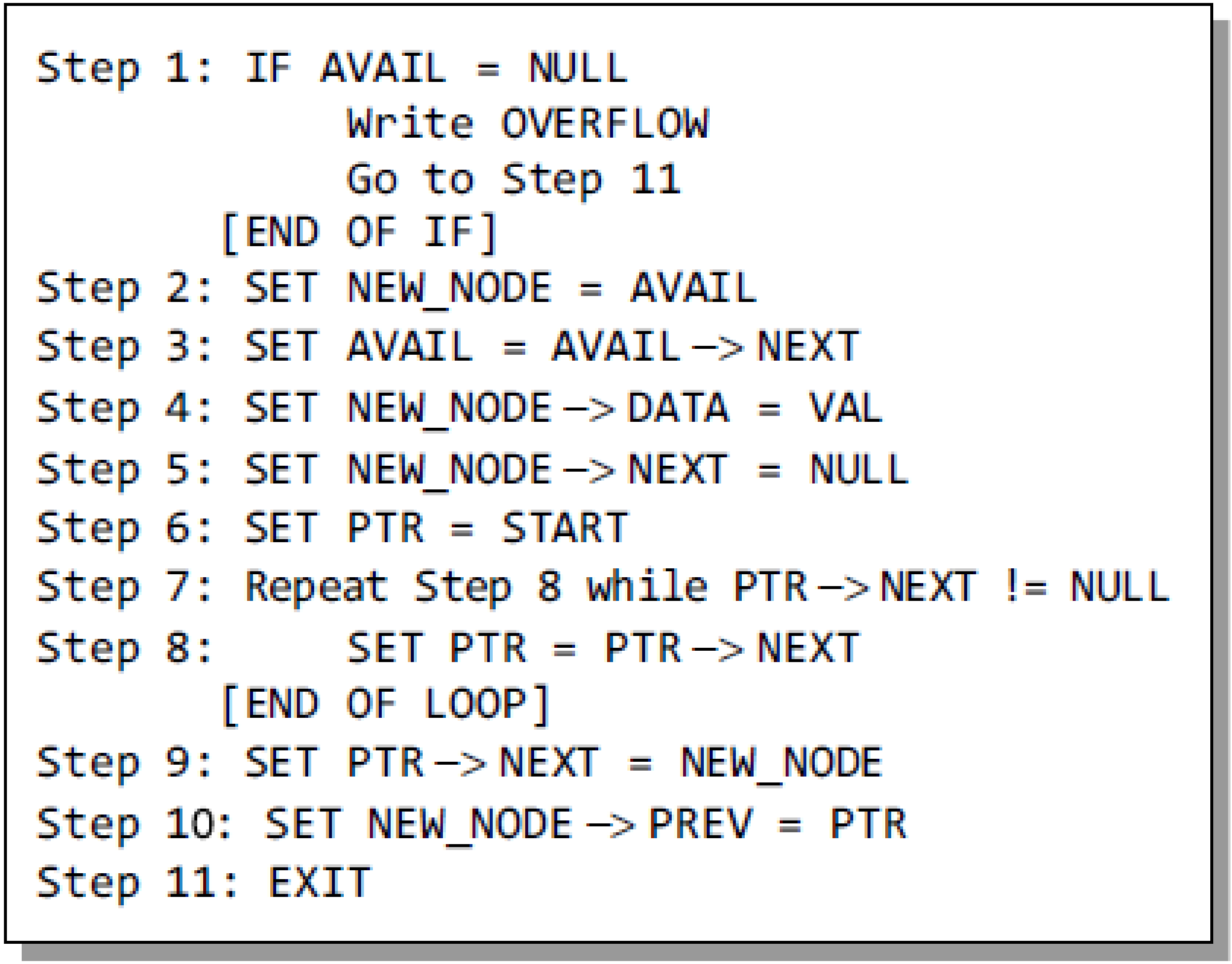
Allocate memory for the new node and initialize its DATA part to 9 and its NEXT field to NULL.

Take a pointer variable PTR and make it point to the first node of the list.

Move PTR so that it points to the last node of the list. Add the new node after the node pointed by PTR.

Figure 6.41 Inserting a new node at the end of a doubly linked list

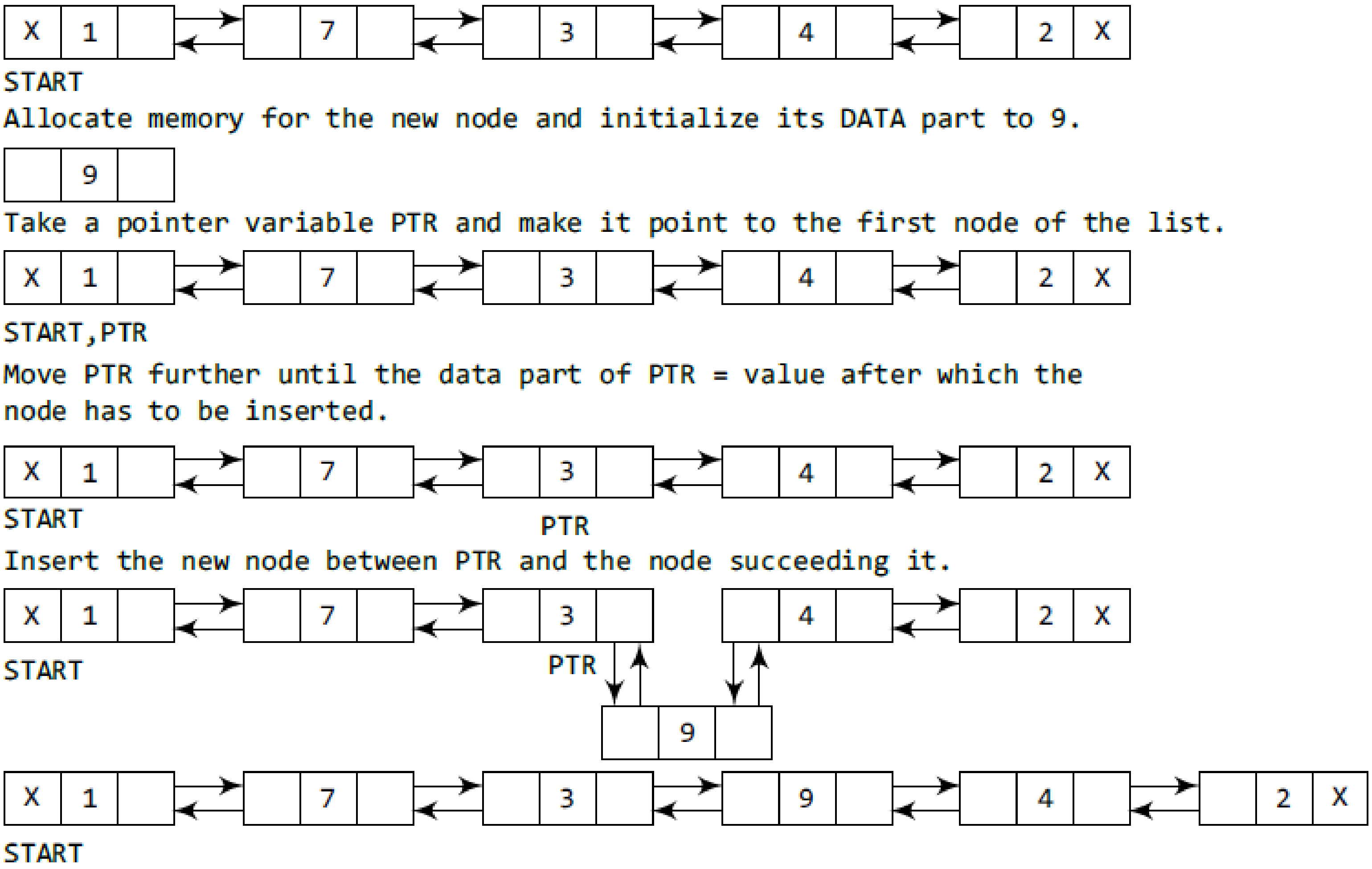
Inserting a Node at the End end of a Doubly Linked List



```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
       [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 = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

Figure 6.42 Algorithm to insert a new node at the end

Inserting a Node After a Given Node in a Doubly Linked List



Allocate memory for the new node and initialize its DATA part to 9.

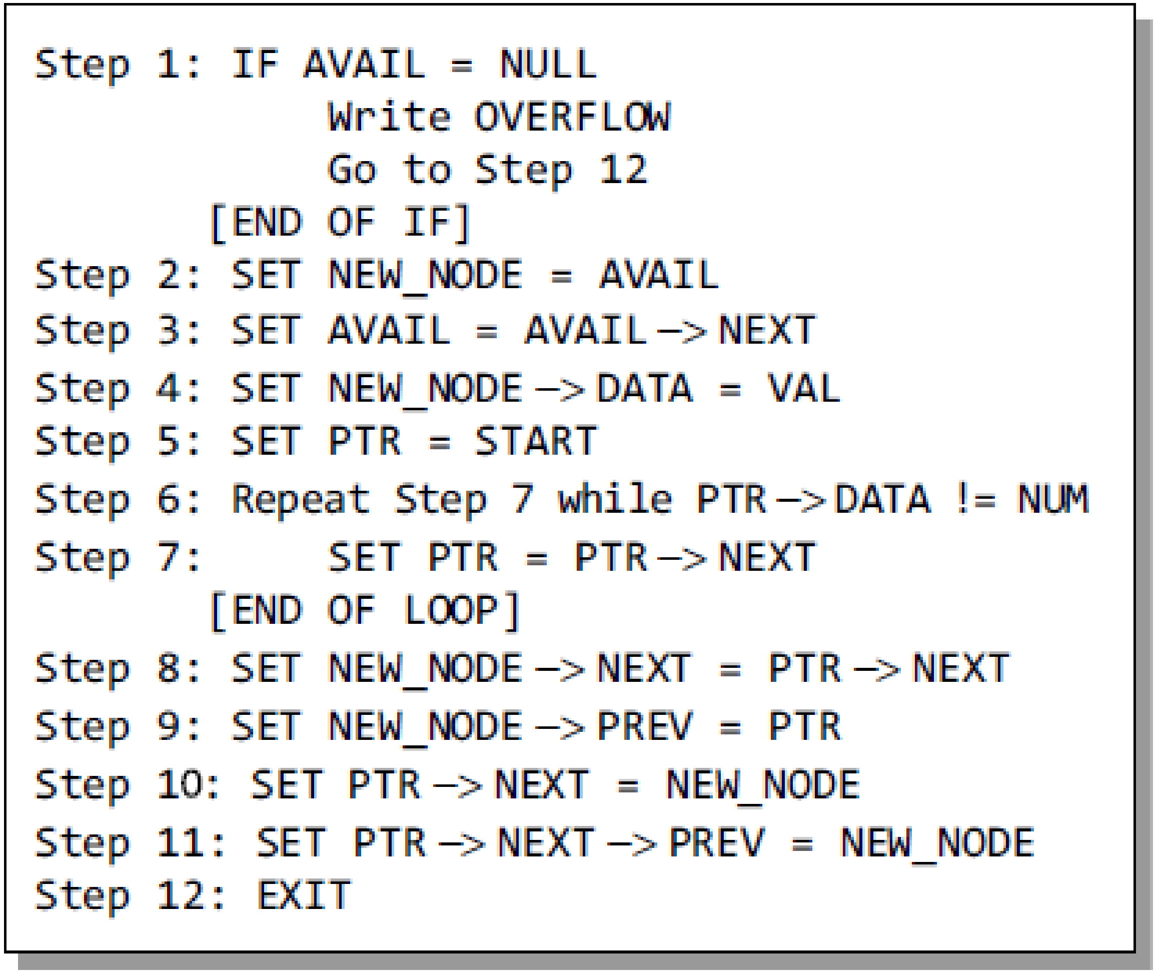
Take a pointer variable PTR and make it point to the first node of the list.

Move PTR further until the data part of PTR = value after which the node has to be inserted.

Insert the new node between PTR and the node succeeding it.

Figure 6.44 Inserting a new node after a given node in a doubly linked list

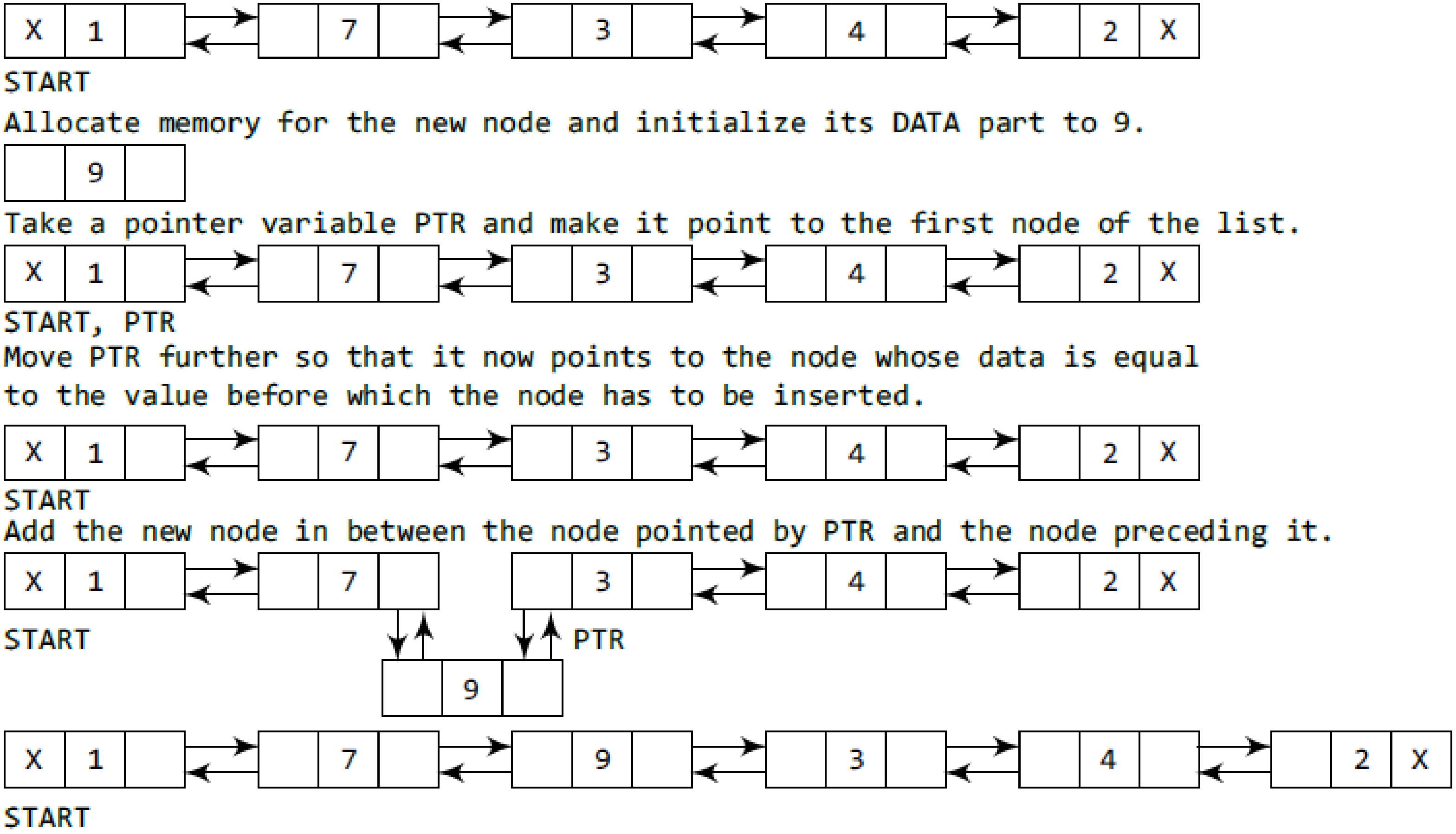
Inserting a Node After a Given Node in a Doubly Linked List



```
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: Repeat Step 7 while PTR -> DATA != NUM
Step 7:       SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

Figure 6.43 Algorithm to insert a new node after a given node

Inserting a Node Before a Given Node in a Doubly Linked List



Allocate memory for the new node and initialize its DATA part to 9.

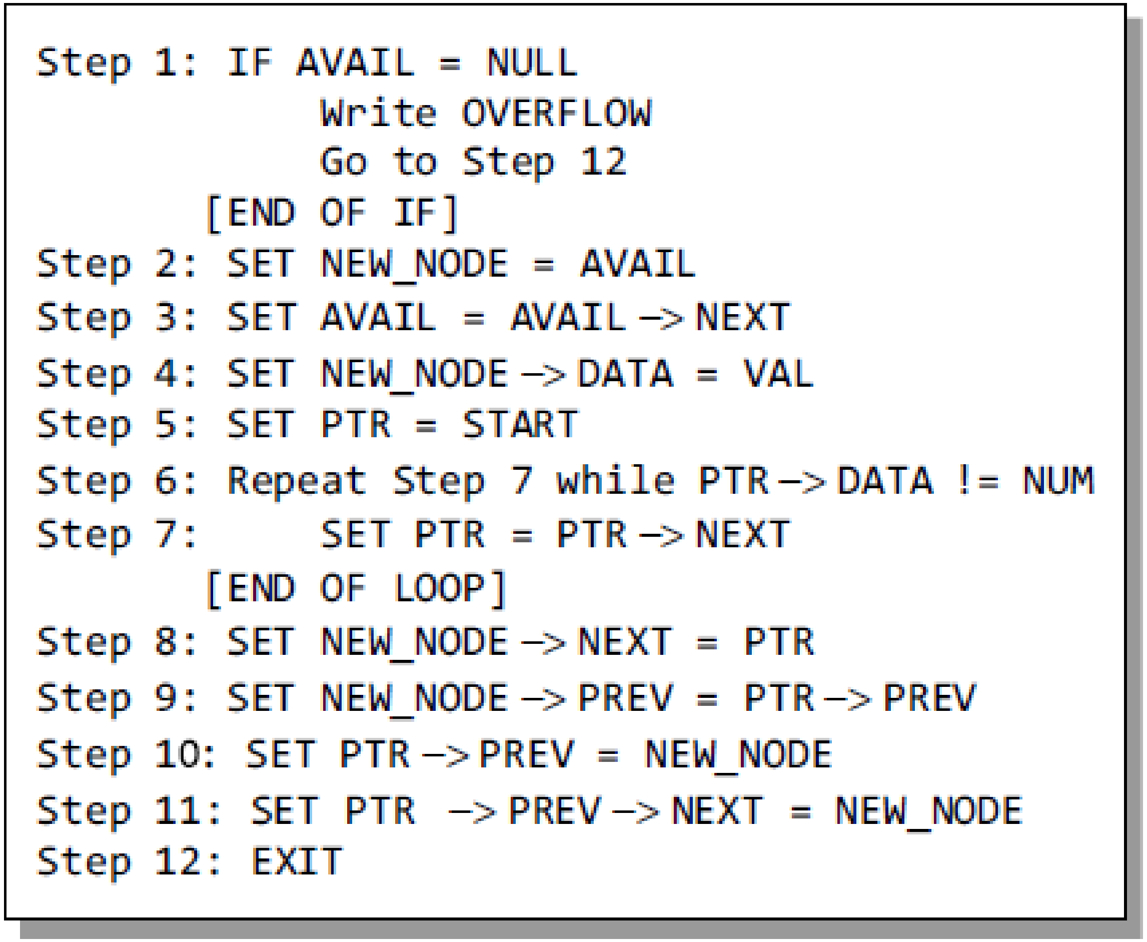
Take a pointer variable PTR and make it point to the first node of the list.

Move PTR further so that it now points to the node whose data is equal to the value before which the node has to be inserted.

Add the new node in between the node pointed by PTR and the node preceding it.

Figure 6.46 Inserting a new node before a given node in a doubly linked list

Inserting a Node Before a Given Node in a Doubly Linked List



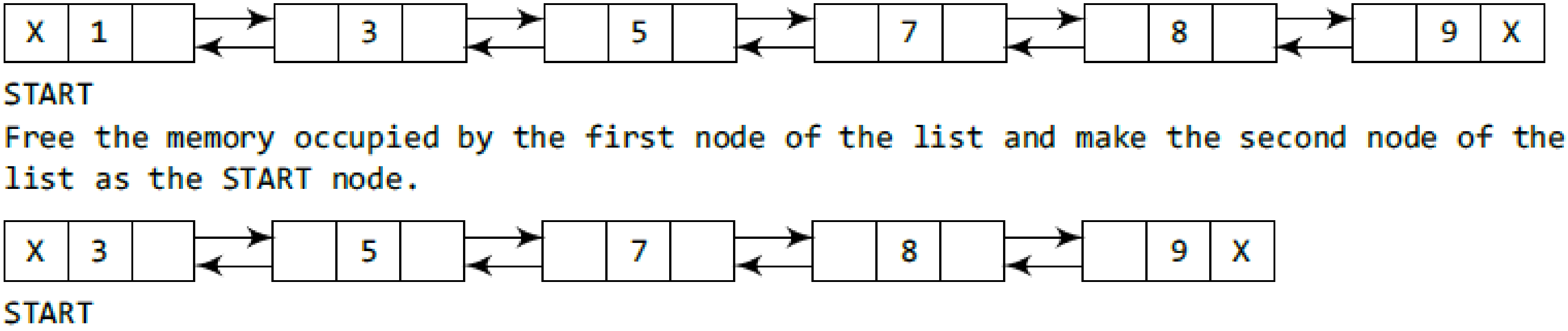
```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 12
         [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR
Step 9: SET NEW_NODE -> PREV = PTR -> PREV
Step 10: SET PTR -> PREV = NEW_NODE
Step 11: SET PTR  -> PREV -> NEXT = NEW_NODE
Step 12: EXIT
```

Figure 6.45 Algorithm to insert a new node before a given node

Deleting a Node from a Doubly Linked List

- We will take four cases and then see how deletion is done in each case.
- Case 1: The first node is deleted.
- Case 2: The last node is deleted.
- Case 3: The node after a given node is deleted.
- Case 4: The node before a given node is deleted.

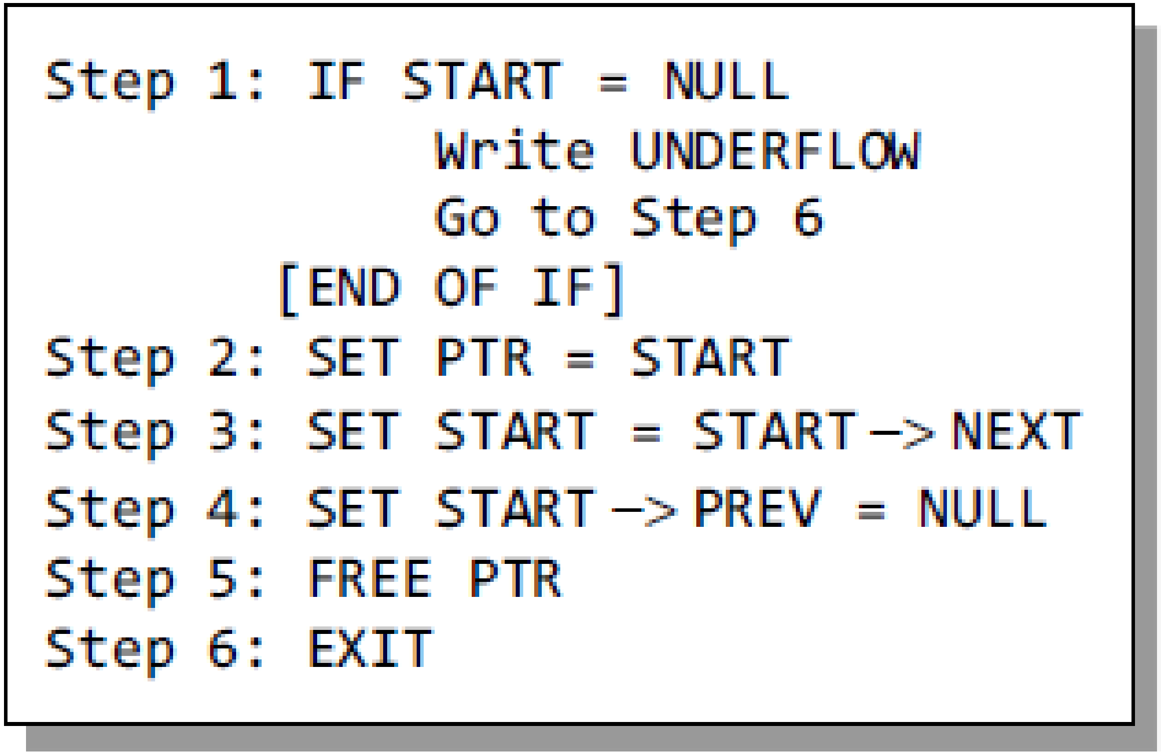
Deleting the First Node from a Doubly Linked List



Free the memory occupied by the first node of the list and make the second node of the list as the START node.

Figure 6.47 Deleting the first node from a doubly linked list

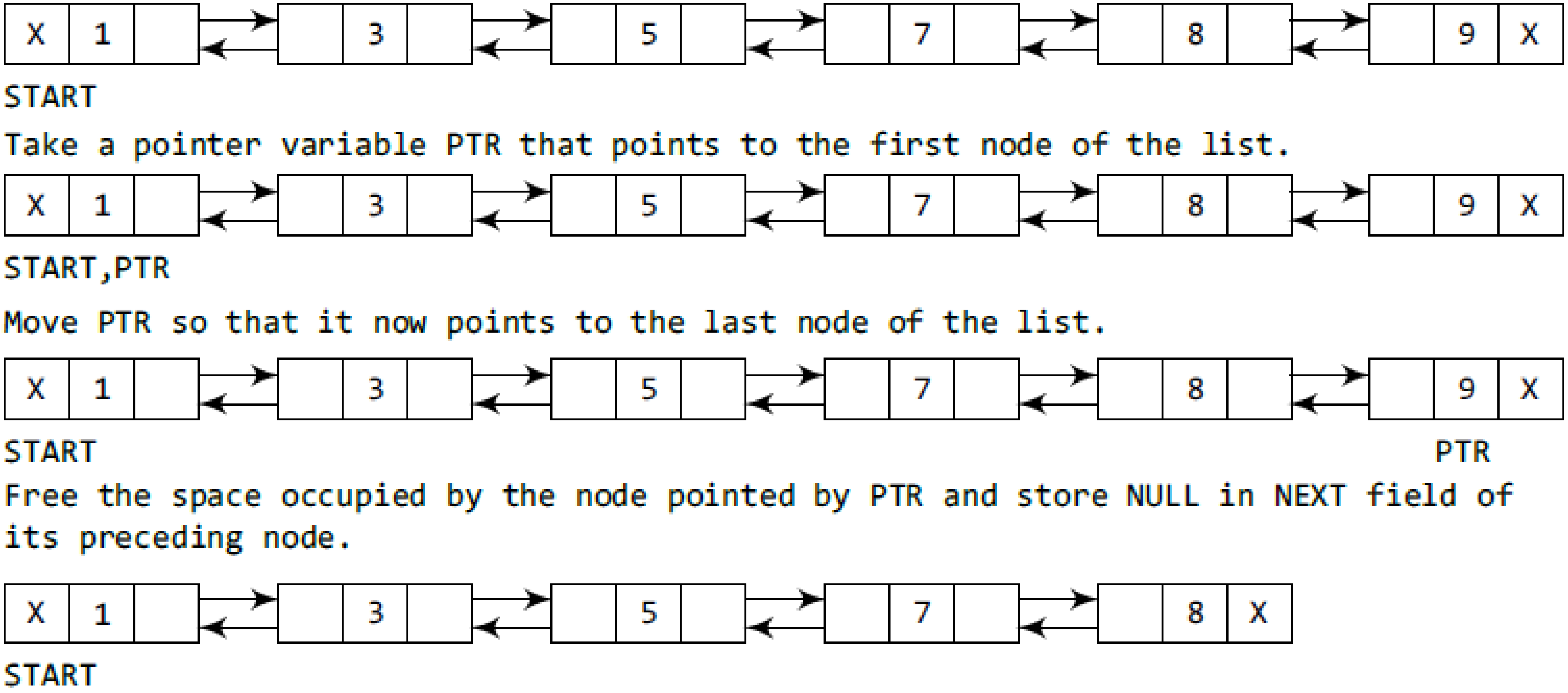
Deleting the First Node from a Doubly Linked List



```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 6
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

Figure 6.48 Algorithm to delete the first node

Deleting the Last Node from a Doubly Linked List



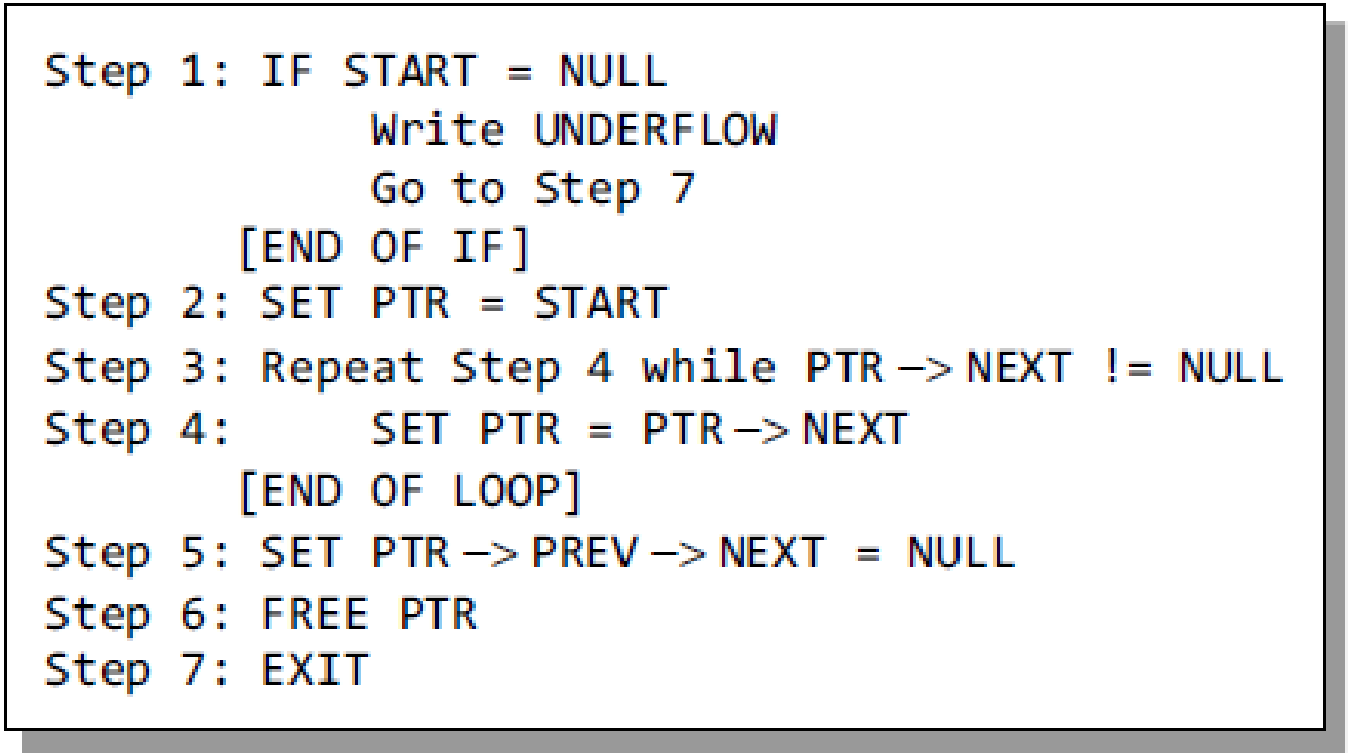
Take a pointer variable PTR that points to the first node of the list.

Move PTR so that it now points to the last node of the list.

Free the space occupied by the node pointed by PTR and store NULL in NEXT field of its preceding node.

Figure 6.49 Deleting the last node from a doubly linked list

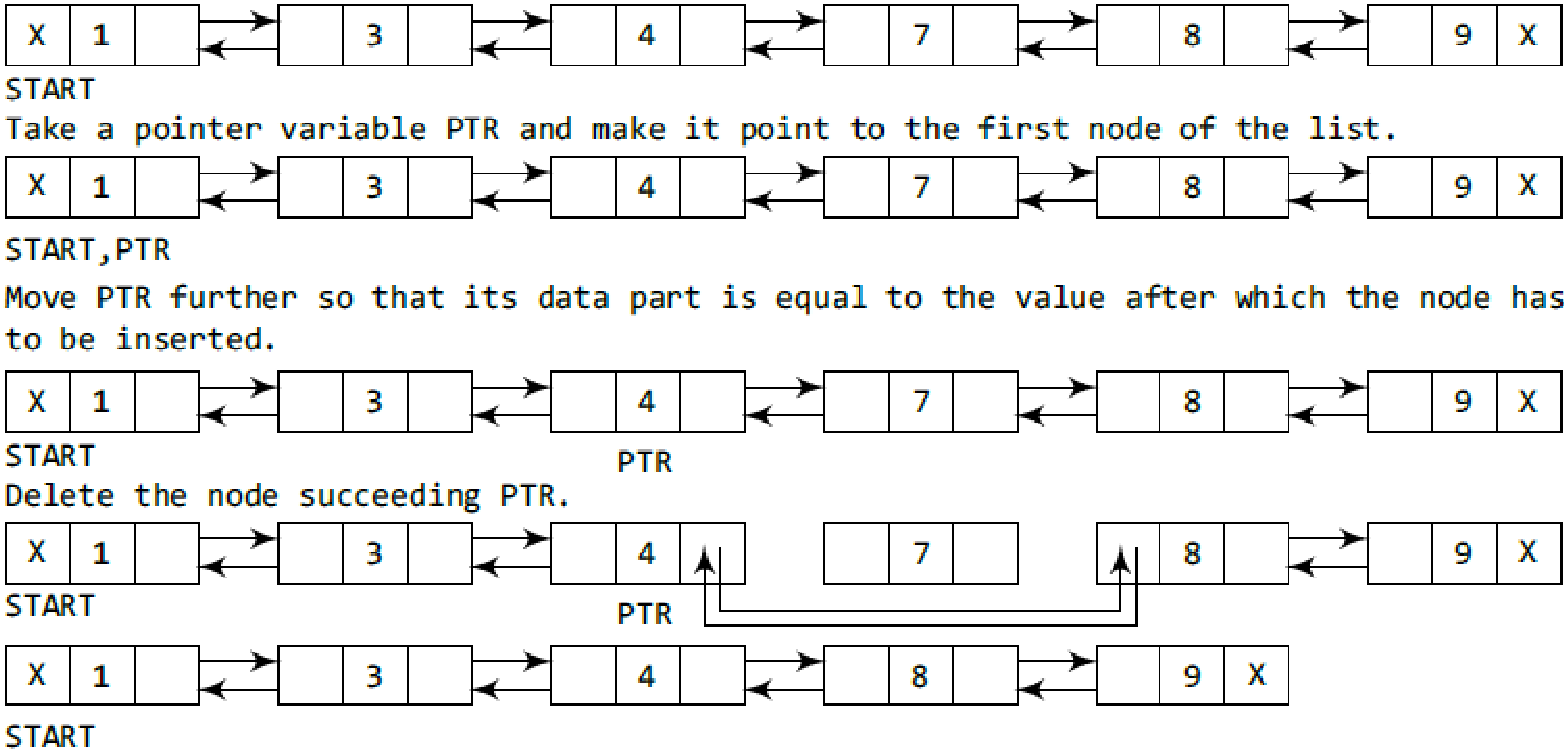
Deleting the Last Node from a Doubly Linked List



```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 7
       [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != NULL
Step 4:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

Figure 6.50 Algorithm to delete the last node

Deleting the Node After a Given Node in a Doubly Linked List



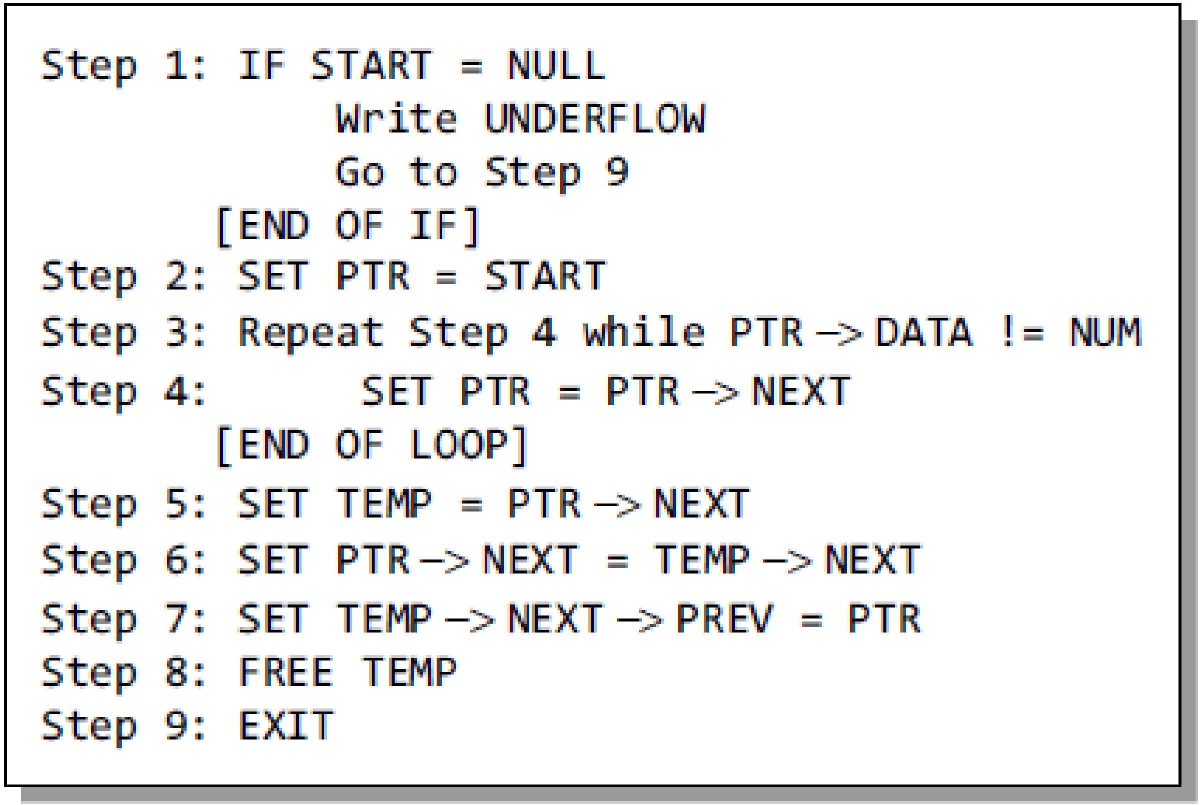
Take a pointer variable PTR and make it point to the first node of the list.

Move PTR further so that its data part is equal to the value after which the node has to be inserted.

Delete the node succeeding PTR.

Figure 6.51 Deleting the node after a given node in a doubly linked list

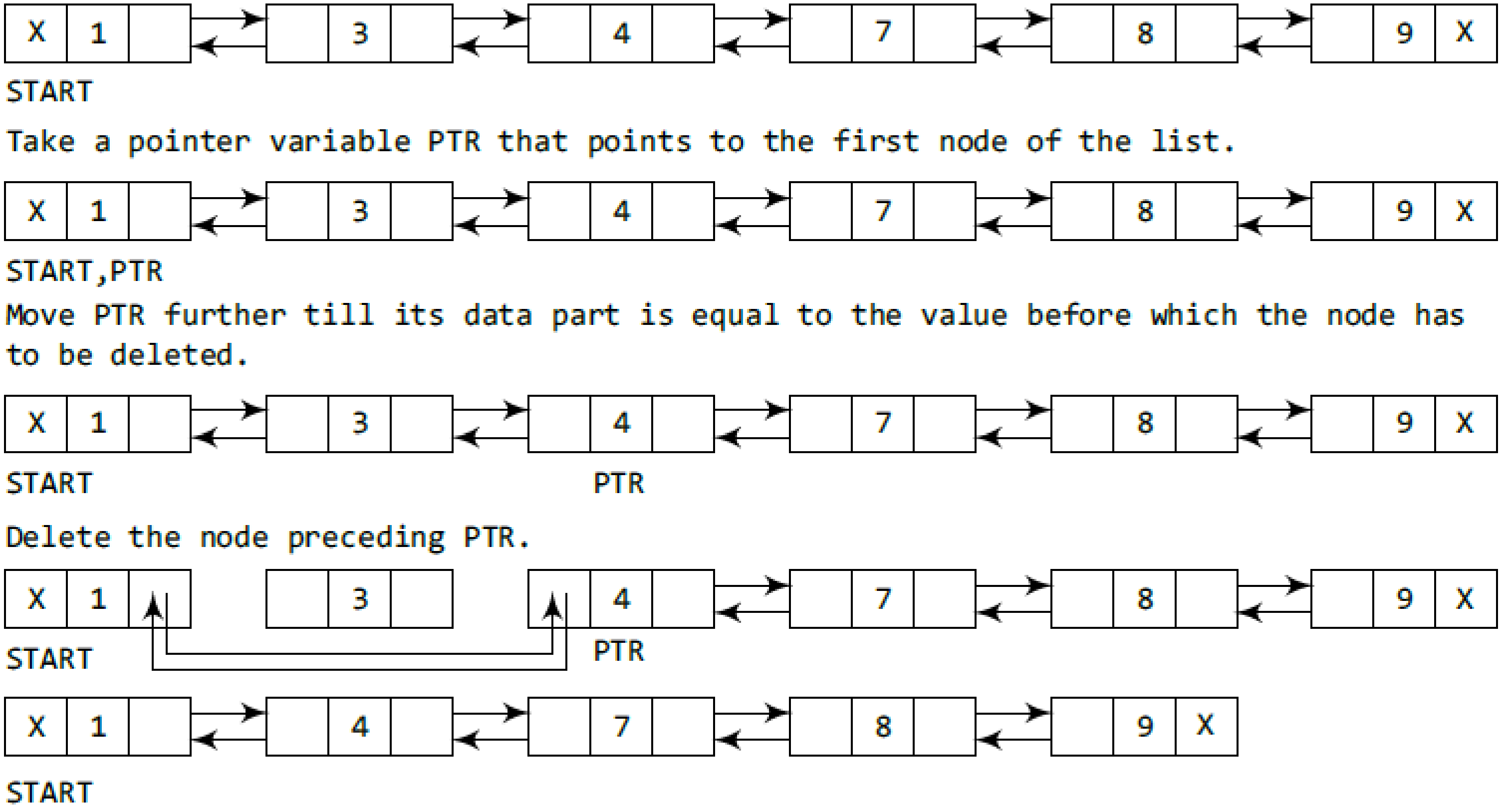
Deleting the Node After a Given Node in a Doubly Linked List



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

Figure 6.52 Algorithm to delete a node after a given node

Deleting the Node Before a Given Node in a Doubly Linked List



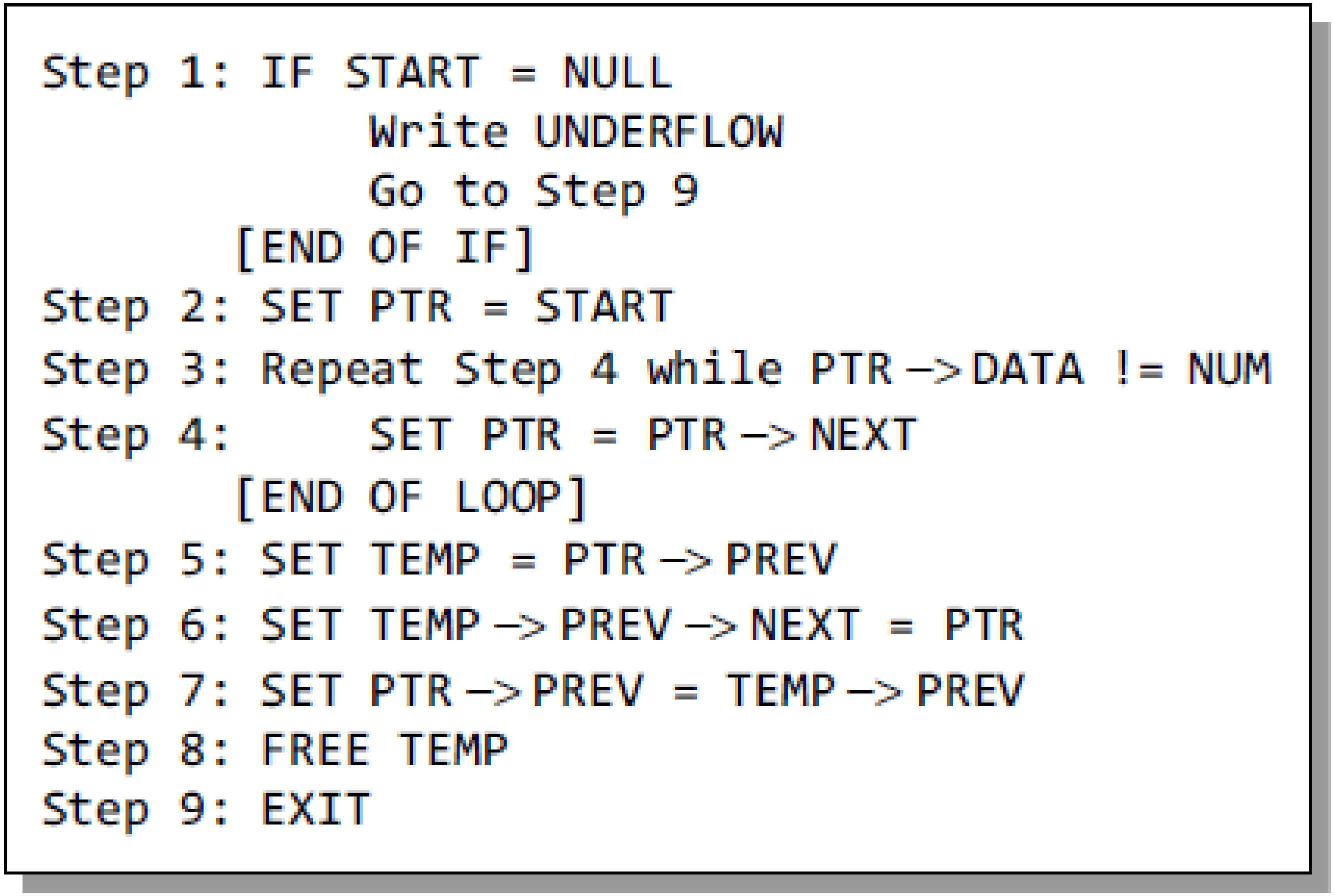
Take a pointer variable PTR that points to the first node of the list.

Move PTR further till its data part is equal to the value before which the node has to be deleted.

Delete the node preceding PTR.

Figure 6.53 Deleting a node before a given node in a doubly linked list

Deleting the Node Before a Given Node in a Doubly Linked List



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

Figure 6.54 Algorithm to delete a node before a given node

CIRCULAR DOUBLY LINKED LISTs

- A circular doubly linked list or a circular two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.
- The difference between a doubly linked and a circular doubly linked list is same as that exists between a singly linked list and a circular linked list.
- The circular doubly linked list does not contain NULL in the previous field of the first node and the next field of the last node. Rather, the next field of the last node stores the address of the first node of the list, i.e., START.
- Similarly, the previous field of the first field stores the address of the last node. A circular doubly linked list is shown in Fig. 6.55.

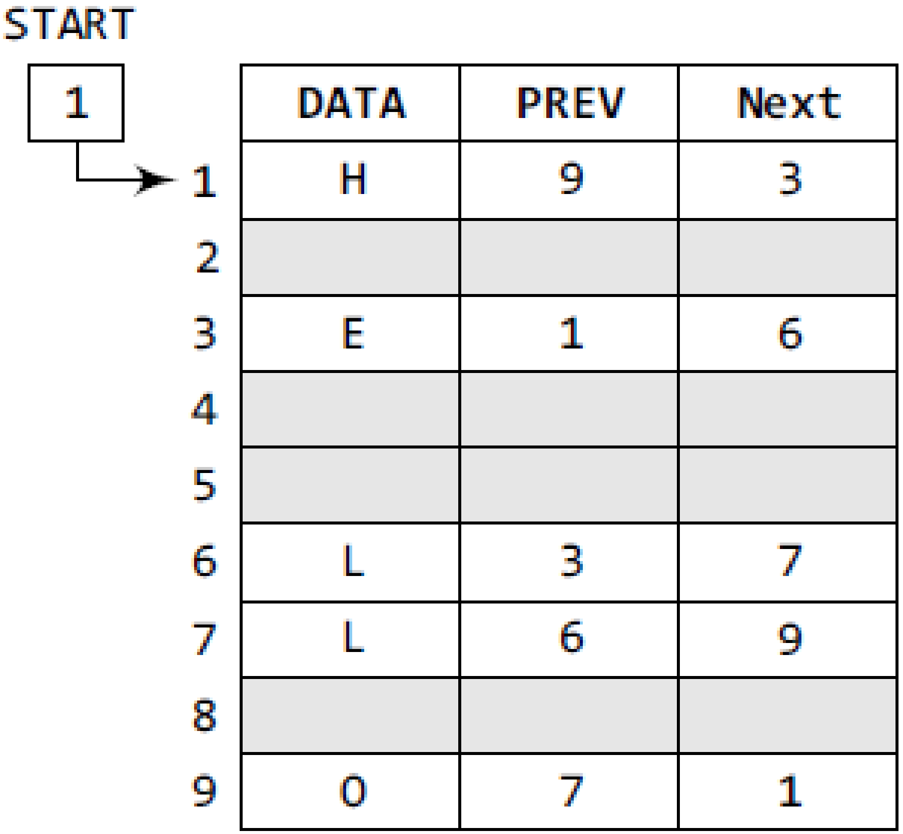
CIRCULAR DOUBLY LINKED LIST

- Since a circular doubly linked list contains three parts in its structure, it calls for more space per node and more expensive basic operations. However, a circular doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward). The main advantage of using a circular doubly linked list is that it makes search operation twice as efficient.

CIRCULAR DOUBLY LINKED LIST

- Let us view how a circular doubly linked list is maintained in the memory.
- Consider Fig. 6.56. In the figure, we see that a variable START is used to store the address of the first node. Here in this example, START = 1, so the first data is stored at address 1, which is H. Since this is the first node, it stores the address of the last node of the list in its previous field. The corresponding NEXT stores the address of the next node, which is 3. So, we will look at address 3 to fetch the next data item. The previous field will contain the address of the first node. The second data element obtained from address 3 is E.
- We repeat this procedure until we reach a position where the NEXT entry stores the address of the first element of the list. This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list.

CIRCULAR DOUBLY LINKED LIST



START 1

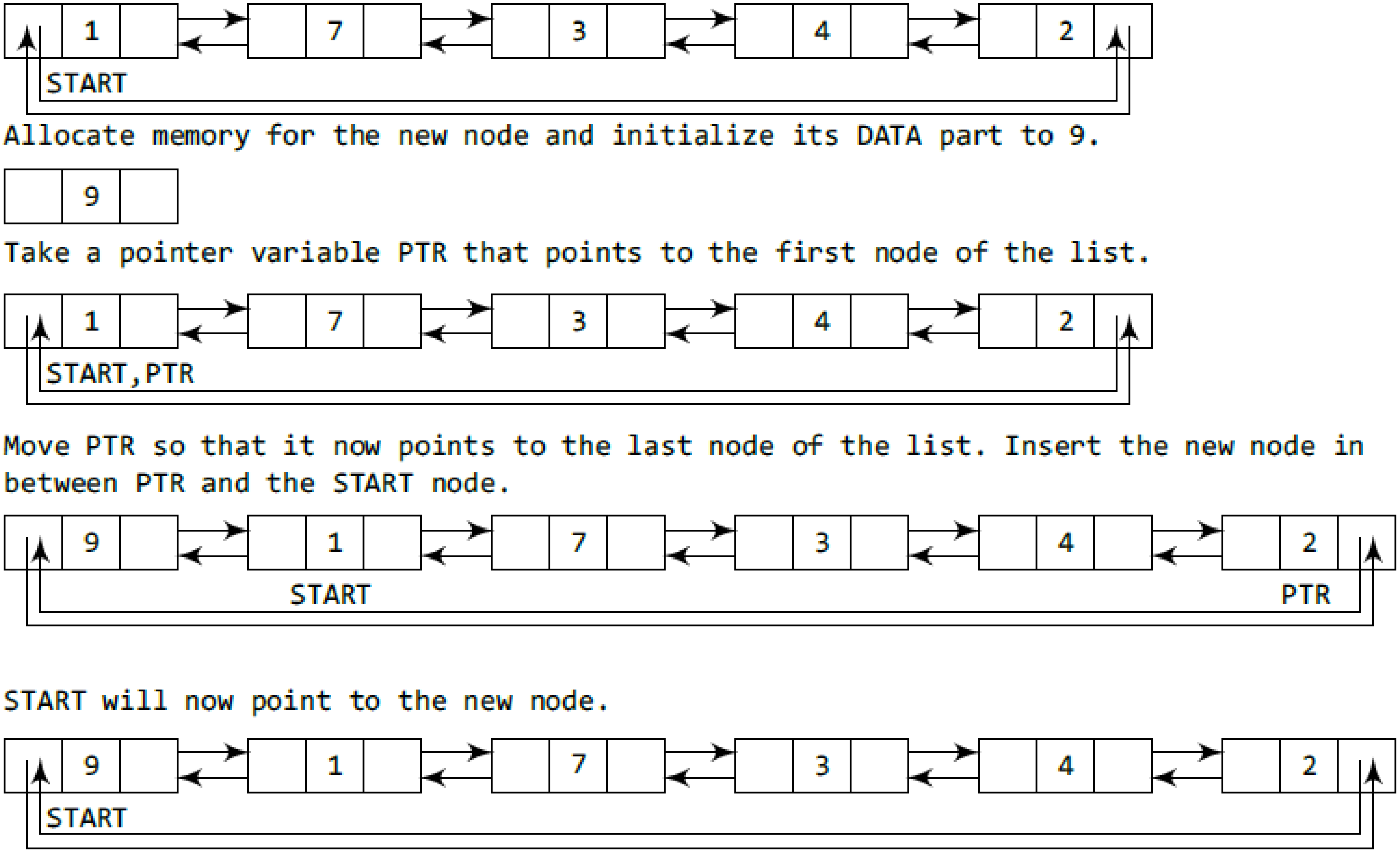
	DATA	PREV	Next
1	H	9	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	1

Figure 6.56 Memory representation of a circular doubly linked list

Inserting a New Node in a Circular Doubly Linked List

- We will take two cases and then see how insertion is done in each case. Rest of the cases are similar to that given for doubly linked lists.
- Case 1: The new node is inserted at the beginning.
- Case 2: The new node is inserted at the end.

Inserting a Node at the Beginning of a Circular Doubly Linked List



Allocate memory for the new node and initialize its DATA part to 9.

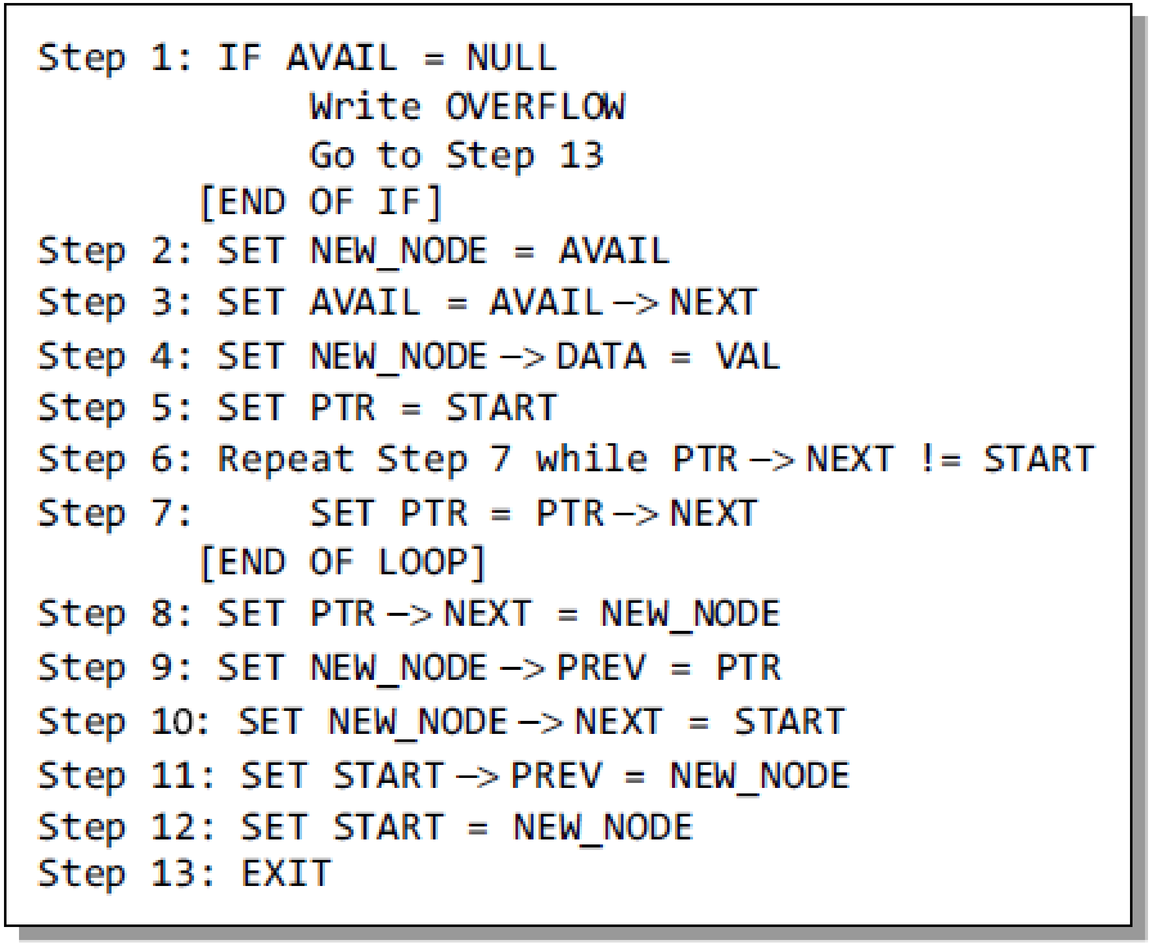
Take a pointer variable PTR that points to the first node of the list.

Move PTR so that it now points to the last node of the list. Insert the new node in between PTR and the START node.

START will now point to the new node.

Figure 6.57 Inserting a new node at the beginning of a circular doubly linked list

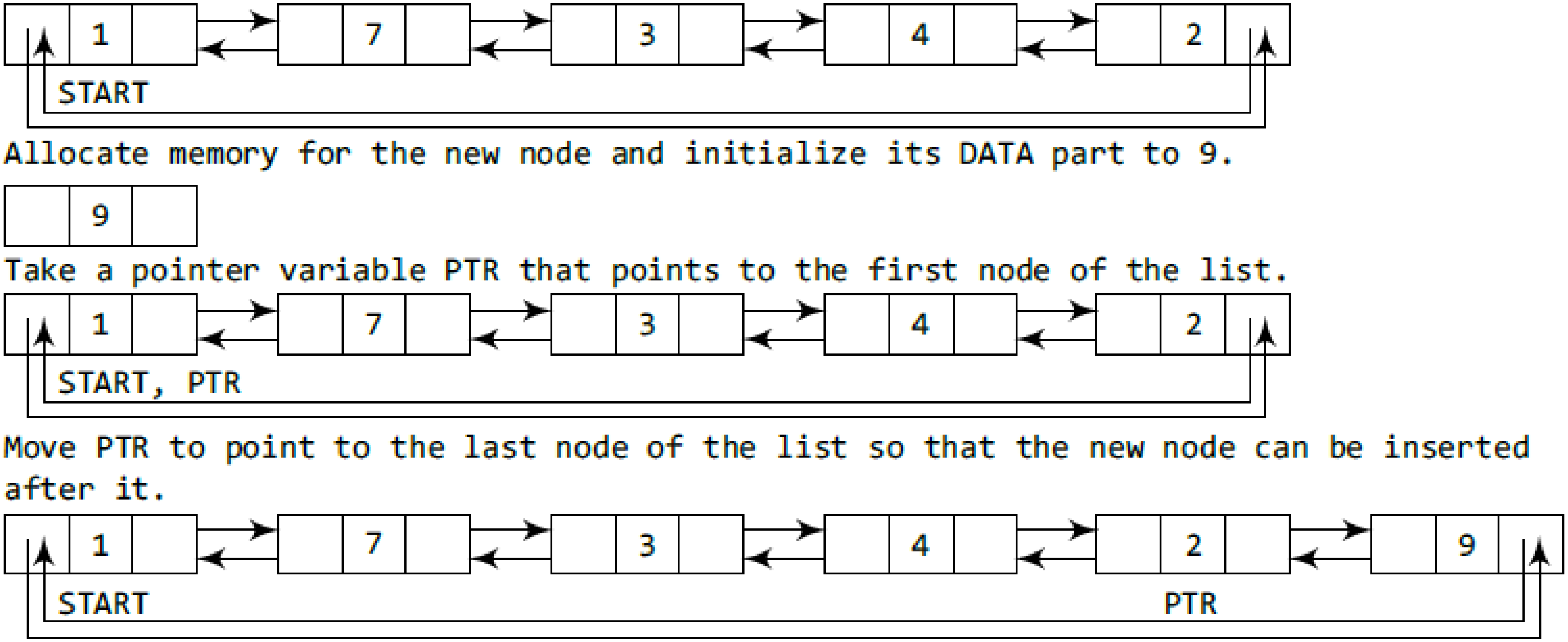
Inserting a Node at the Beginning of a Circular Doubly Linked List



```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 13
       [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: Repeat Step 7 while PTR -> NEXT != START
Step 7:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 8: SET PTR -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET NEW_NODE -> NEXT = START
Step 11: SET START -> PREV = NEW_NODE
Step 12: SET START = NEW_NODE
Step 13: EXIT
```

Figure 6.58 Algorithm to insert a new node at the beginning

Inserting a Node at the End of a Circular Doubly Linked List



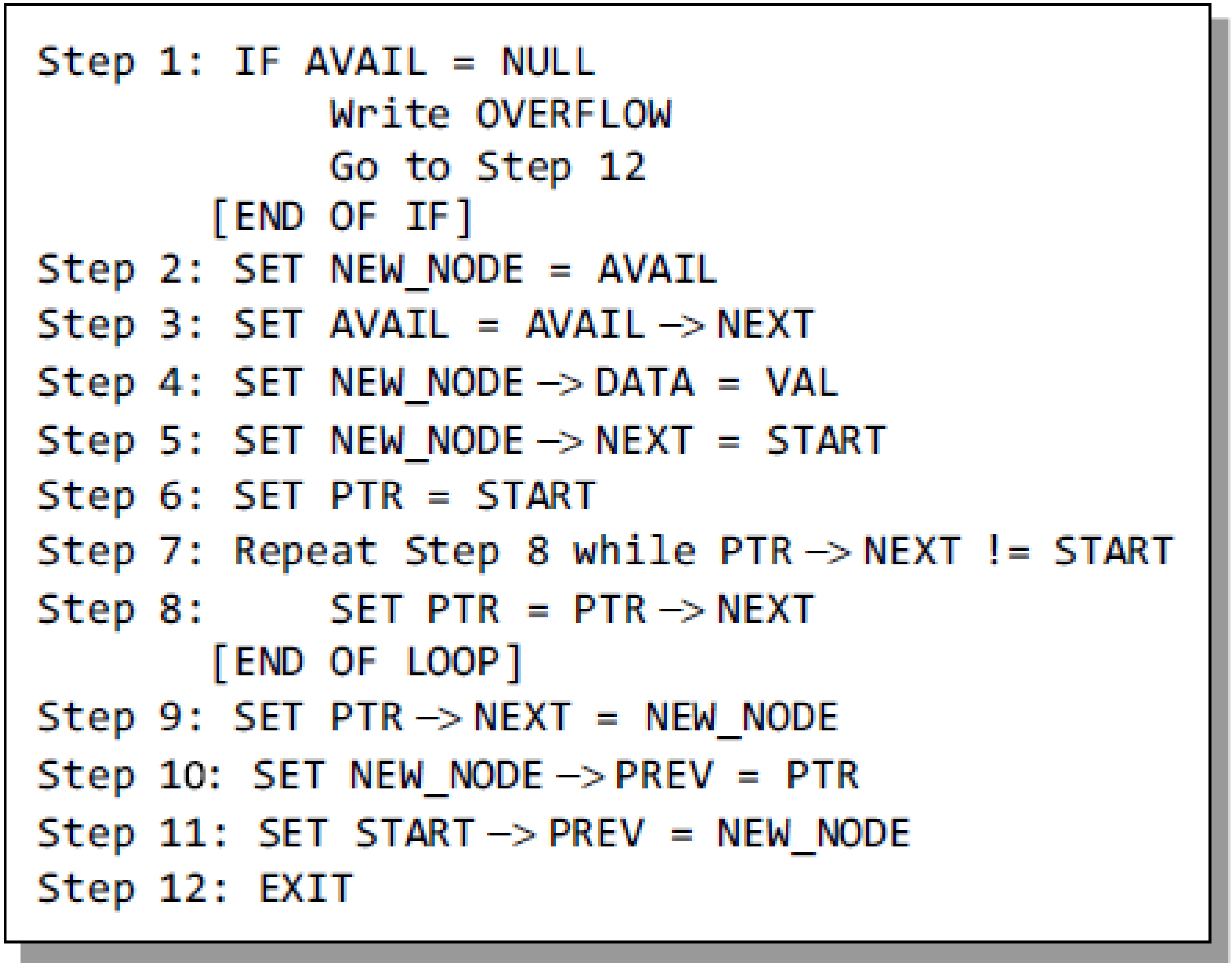
Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable PTR that points to the first node of the list.

Move PTR to point to the last node of the list so that the new node can be inserted after it.

Figure 6.59 Inserting a new node at the end of a circular doubly linked list

Inserting a Node at the End of a Circular Doubly Linked List



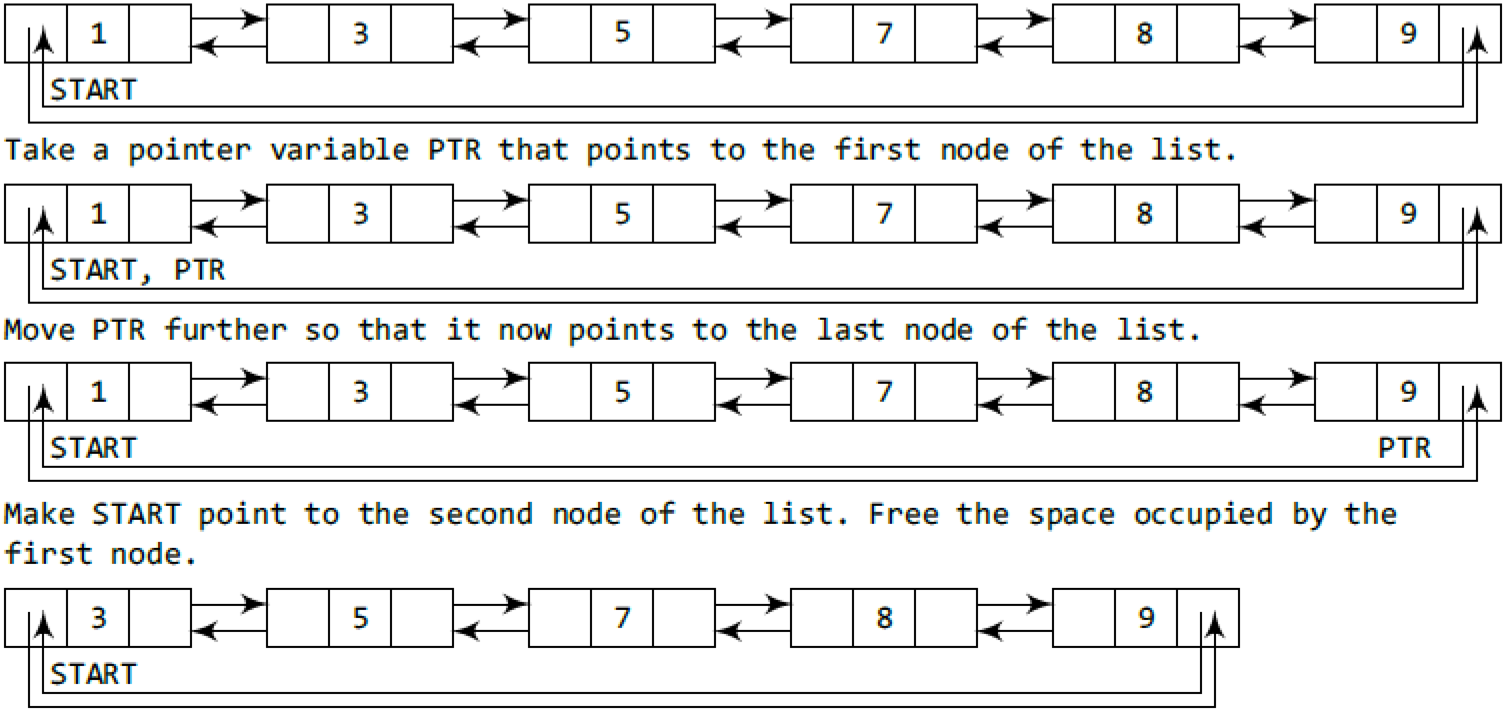
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: SET START -> PREV = NEW_NODE
Step 12: EXIT
```

Figure 6.60 Algorithm to insert a new node at the end

Deleting a Node from a Circular Doubly Linked List

- We will take two cases and then see how deletion is done in each case. Rest of the cases are same as that given for doubly linked lists.
- Case 1: The first node is deleted.
- Case 2: The last node is deleted.

Deleting the First Node from a Circular Doubly Linked List



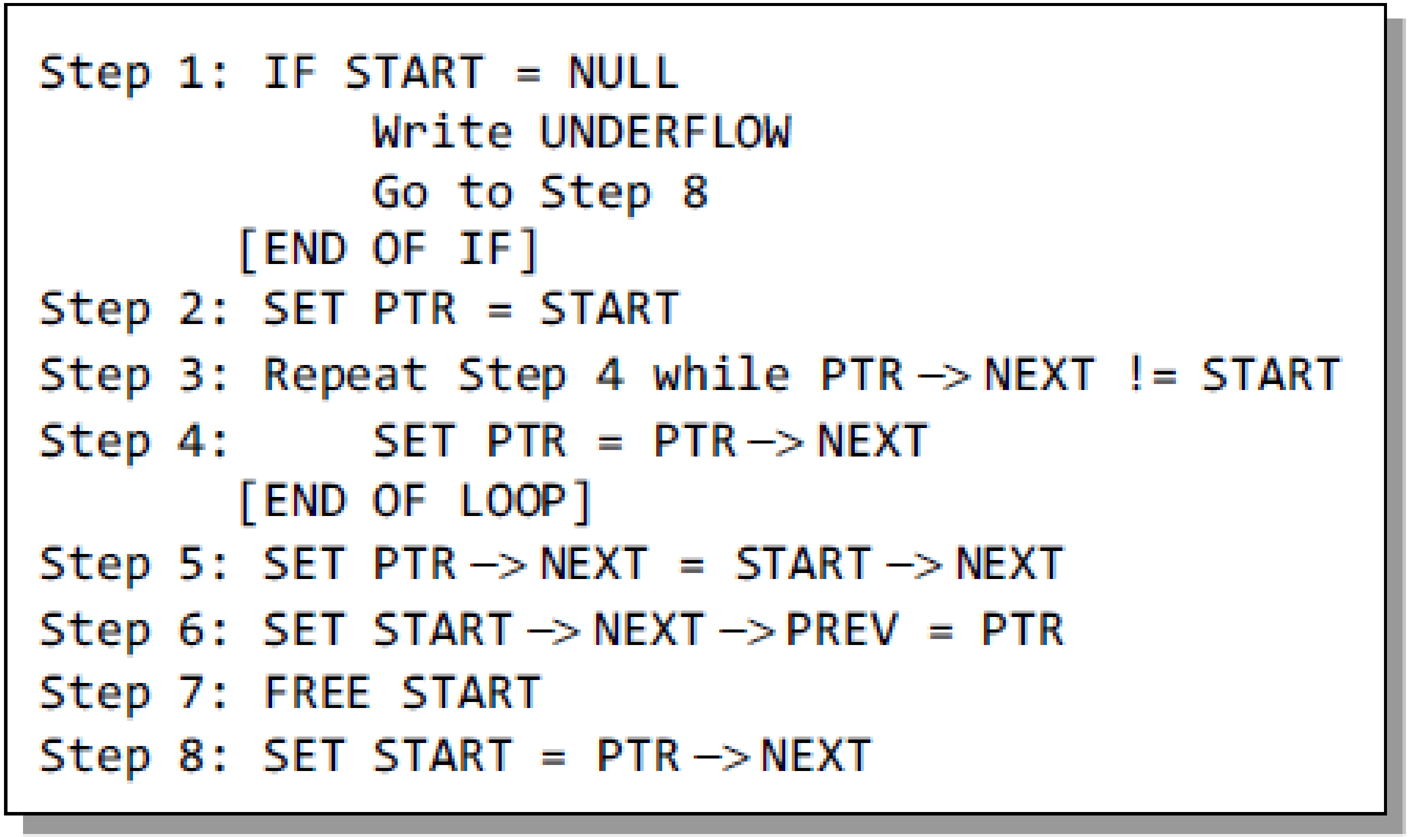
Take a pointer variable PTR that points to the first node of the list.

Move PTR further so that it now points to the last node of the list.

Make START point to the second node of the list. Free the space occupied by the first node.

Figure 6.61 Deleting the first node from a circular doubly linked list

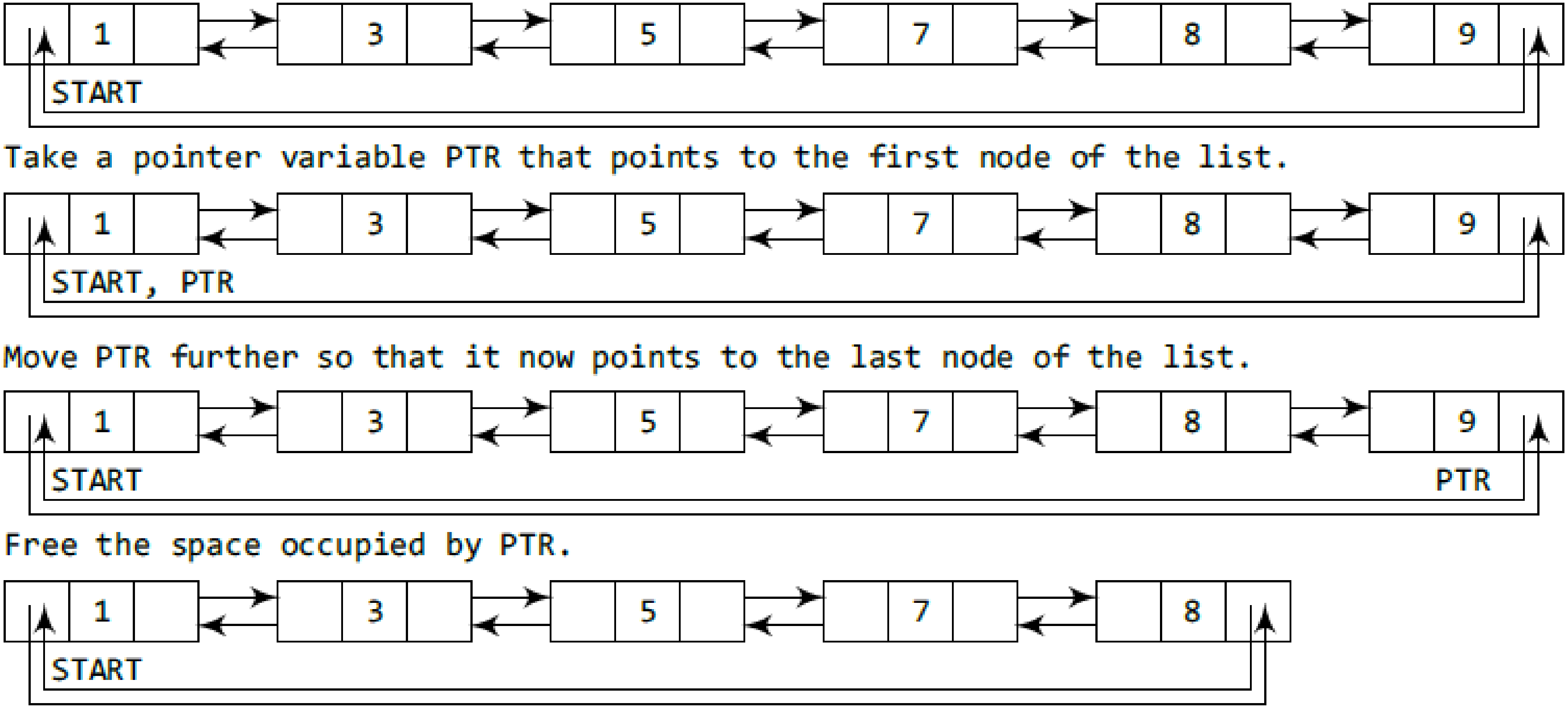
Deleting the First Node from a Circular Doubly Linked List



```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: SET START -> NEXT -> PREV = PTR
Step 7: FREE START
Step 8: SET START = PTR -> NEXT
```

Figure 6.62 Algorithm to delete the first node

Deleting the Last Node from a Circular Doubly Linked List



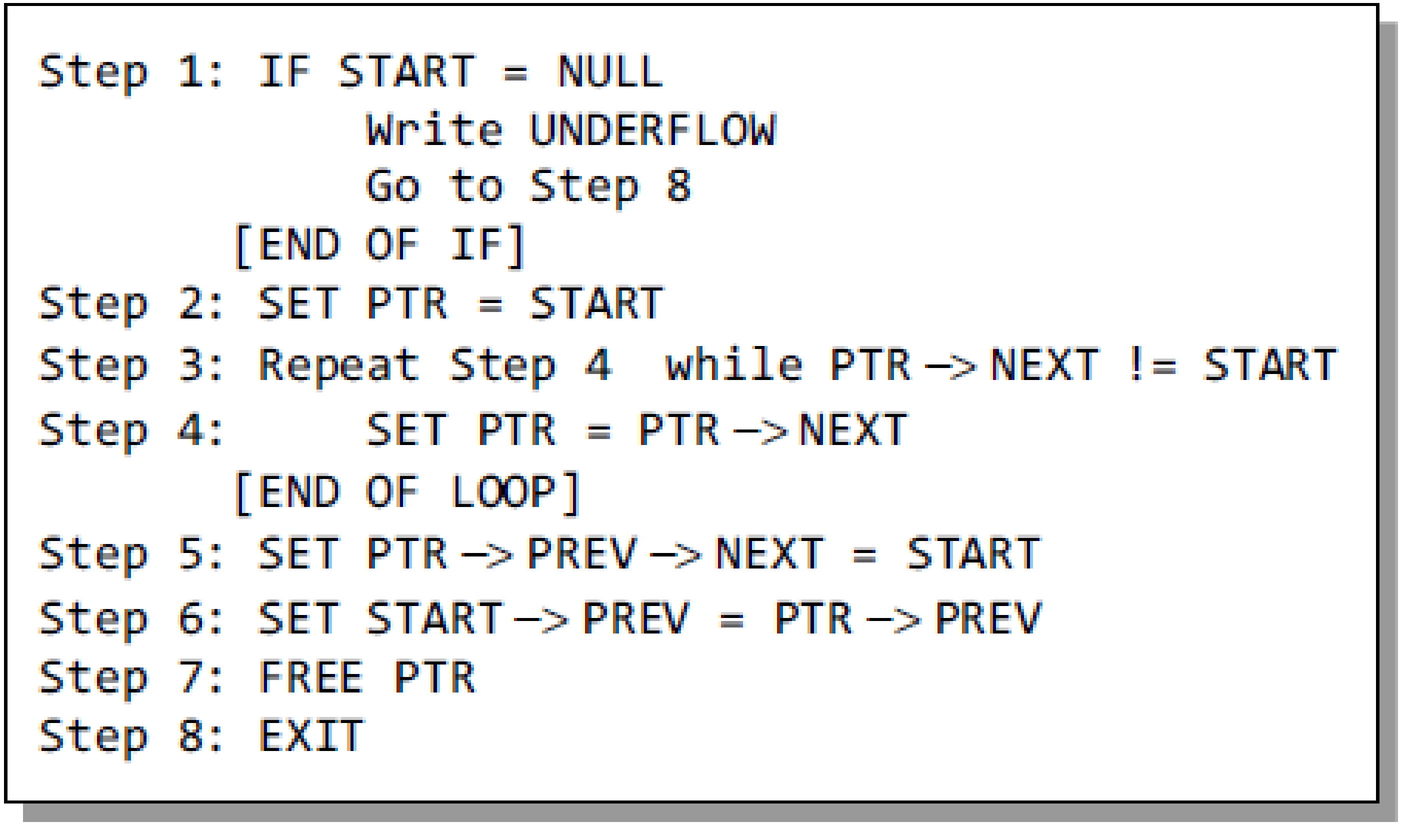
Take a pointer variable PTR that points to the first node of the list.

Move PTR further so that it now points to the last node of the list.

Free the space occupied by PTR.

Figure 6.63 Deleting the last node from a circular doubly linked list

Deleting the Last Node from a Circular Doubly 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  while PTR -> NEXT != START
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = START
Step 6: SET START -> PREV = PTR -> PREV
Step 7: FREE PTR
Step 8: EXIT
```

Figure 6.64 Algorithm to delete the last node

Header Linked Lists

- A header linked list is a special type of linked list which contains a header node at the beginning of the list. So, in a header linked list, START will not point to the first node of the list but START will contain the address of the header node.
- The following are the two variants of a header linked list:
- *Grounded header linked list which stores NULL in the next field of the last node.*
- *Circular header linked list which stores the address of the header node in the next field of* the last node. Here, the header node will denote the end of the list.
- Look at Fig. 6.65 which shows both the types of header linked lists.

Header Linked Lists

- In a grounded header linked list, a node has two fields, DATA and NEXT. The DATA field will store the information part and the NEXT field will store the address of the node in sequence. Consider Fig. 6.66.
- Note that START stores the address of the header node.
- The shaded row denotes a header node. The NEXT field of the header node stores the address of the first node of the list. This node stores H. The corresponding NEXT field stores the address of the next node, which is 3.
- So, we will look at address 3 to fetch the next data item.

Header Linked Lists

- Hence, we see that the first node can be accessed by writing FIRST_NODE = START -> NEXT and not by writing START = FIRST_NODE.
- This is because START points to the header node and the header node points to the first node of the header linked list.
- Let us now see how a circular header linked list is stored in the memory. Look at Fig. 6.67.
- Note that the last node in this case stores the address of the header node (instead of –1).
- Hence, we see that the first node can be accessed by writing FIRST_NODE = START ->NEXT and not writing START = FIRST_NODE.
- This is because START points to the header node and the header node points to the first node of the header linked list.

Header Linked Lists

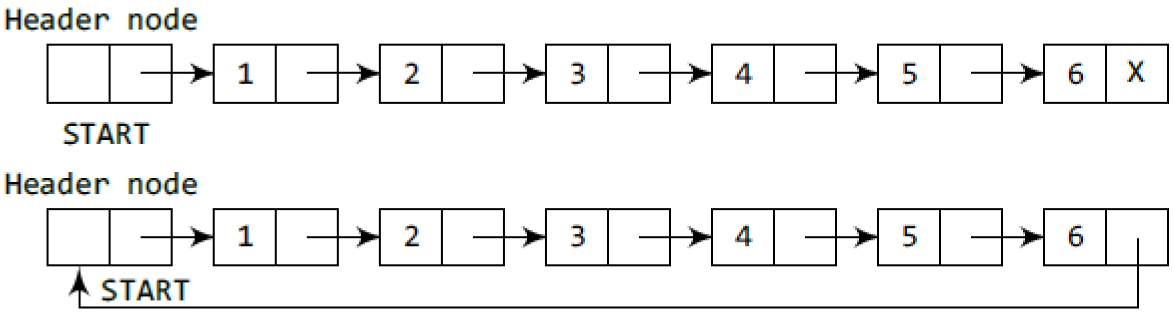


Figure 6.65 Header linked list

START
5

	DATA	NEXT
1	H	3
2		
3	E	6
4		
5	1234	1
6	L	7
7	L	9
8		
9	O	–1

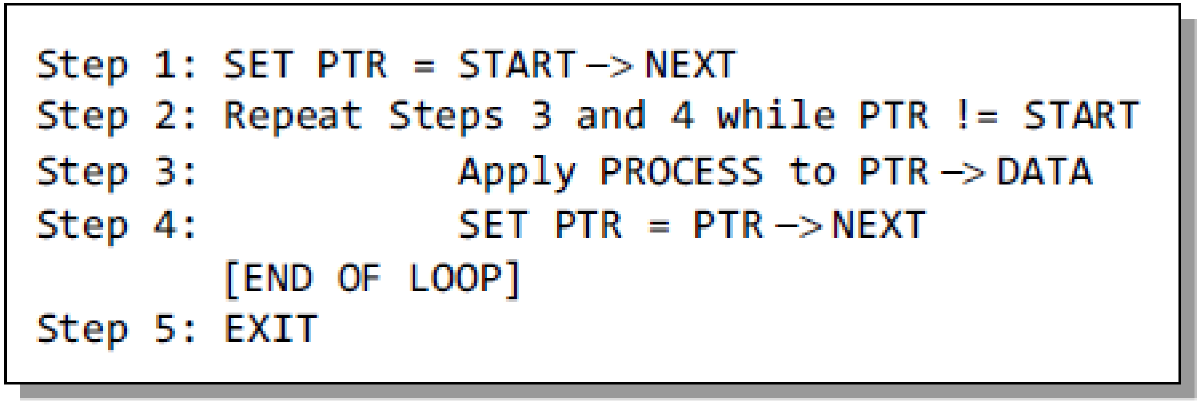
Figure 6.66 Memory representation of a header linked list

START
5

	DATA	NEXT
1	H	3
2		
3	E	6
4		
5	1234	1
6	L	7
7	L	9
8		
9	O	5

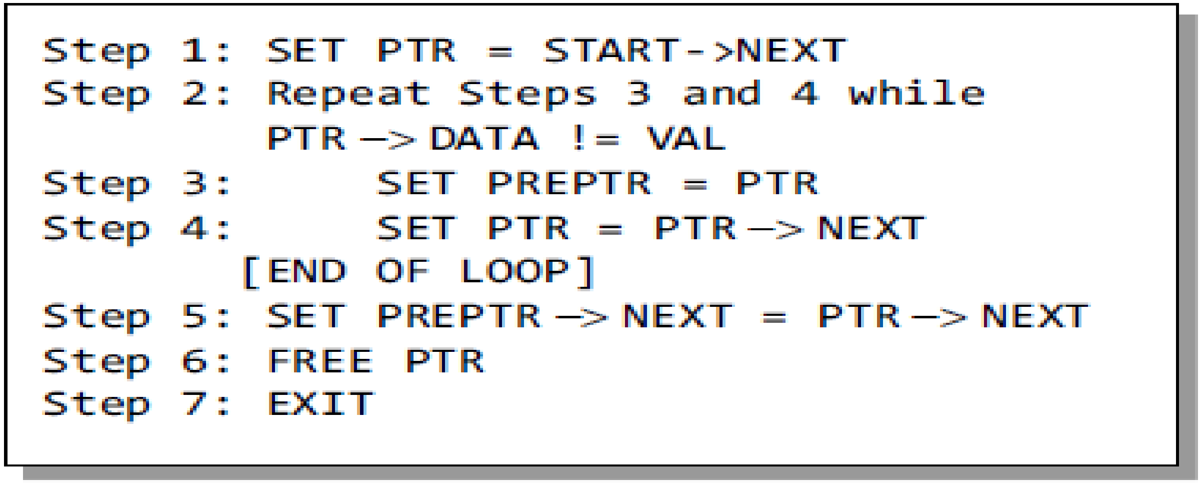
Figure 6.67 Memory representation of a circular header linked list

Header Linked Lists



```
Step 1: SET PTR = START -> NEXT
Step 2: Repeat Steps 3 and 4 while PTR != START
Step 3:             Apply PROCESS to PTR -> DATA
Step 4:             SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: EXIT
```

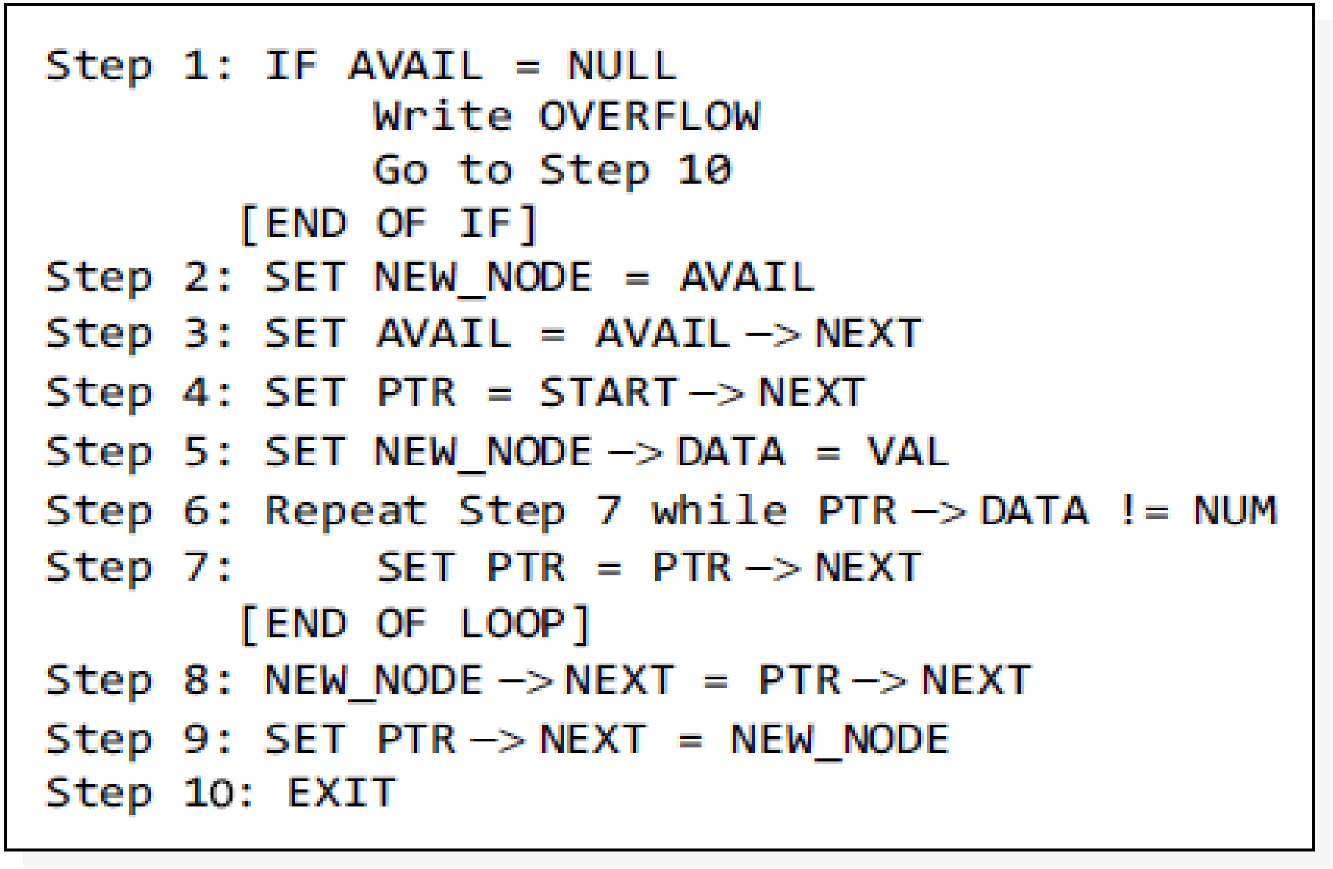
Figure 6.68 Algorithm to traverse a circular header linked list



```
Step 1: SET PTR = START->NEXT
Step 2: Repeat Steps 3 and 4 while
        PTR -> DATA != VAL
Step 3:     SET PREPTR = PTR
Step 4:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 5: SET PREPTR -> NEXT = PTR -> NEXT
Step 6: FREE PTR
Step 7: EXIT
```

Figure 6.70 Algorithm to delete a node from a circular header linked list

Header Linked Lists



```
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 PTR = START -> NEXT
Step 5: SET NEW_NODE -> DATA = VAL
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 8: NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Figure 6.69 Algorithm to insert a new node in a circular header linked list

Multi-Linked Lists

- In a multi-linked list, each node can have n number of pointers to other nodes.
- A doubly linked list is a special case of multi-linked lists. However, unlike doubly linked lists, nodes in a multilinked list may or may not have inverses for each pointer. We can differentiate a doubly linked list from a multi-linked list in two ways:
- (a) A doubly linked list has exactly two pointers. One pointer points to the previous node and the other points to the next node. But a node in the multi-linked list can have any number of pointers.
- (b) In a doubly linked list, pointers are exact inverses of each other, i.e., for every pointer which points to a previous node there is a pointer which points to the next node.
- This is not true for a multi-linked list.

Multi-Linked Lists

- Multi-linked lists are generally used to organize multiple orders of one set of elements.
- For example, if we have a linked list that stores name and marks obtained by students in a class, then we can organize the nodes of the list in two ways:
- (i) Organize the nodes alphabetically (according to the name)
- (ii) Organize the nodes according to decreasing order of marks so that the information of student who got highest marks comes before other students.
- Figure 6.71 shows a multi-linked list in which students' nodes are organized by both the aforementioned ways.
- A new node can be inserted in a multi-linked list in the same way as it is done for a doubly linked list.

Multi-Linked Lists

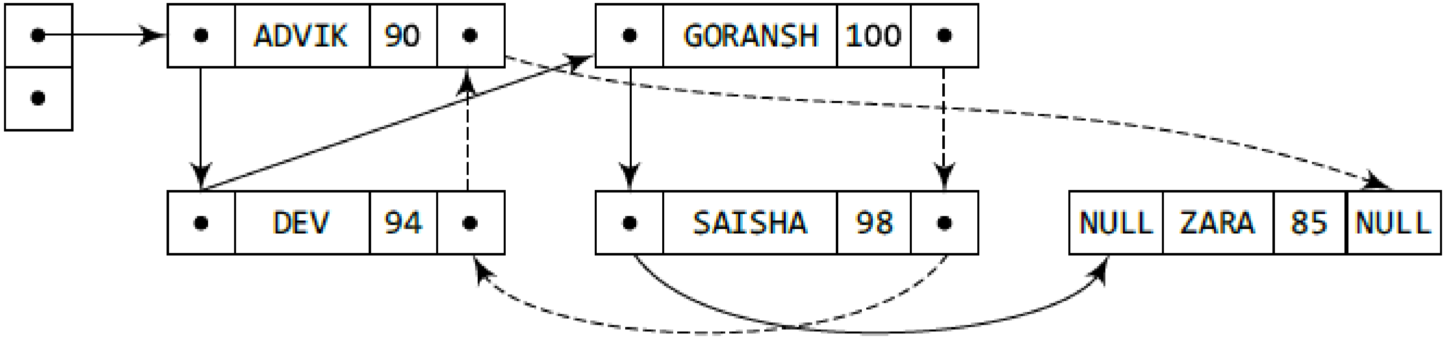


Figure 6.71 Multi-linked list that stores names alphabetically as well as according to decreasing order of marks

Multi-Linked Lists

- Multi-linked lists are also used to store sparse matrices.
- If we use a normal array to store such matrices, we will end up wasting a lot of space.
- Therefore, a better solution is to represent these matrices using multi-linked lists.
- The sparse matrix shown in Fig. 6.72 can be represented using a linked list for every row and column. Since a value is in exactly one row and one column, it will appear in both lists exactly once.
- A node in the multi-linked will have four parts.
- First stores the data, second stores a pointer to the next node in the row, third stores a pointer to the next node in the column, and the fourth stores the coordinates or the row and column number in which the data appears in the matrix.
- However, as in case of doubly linked lists, we can also have a corresponding inverse pointer for every pointer in the multi-linked list representation of a sparse matrix.

Multi-Linked Lists

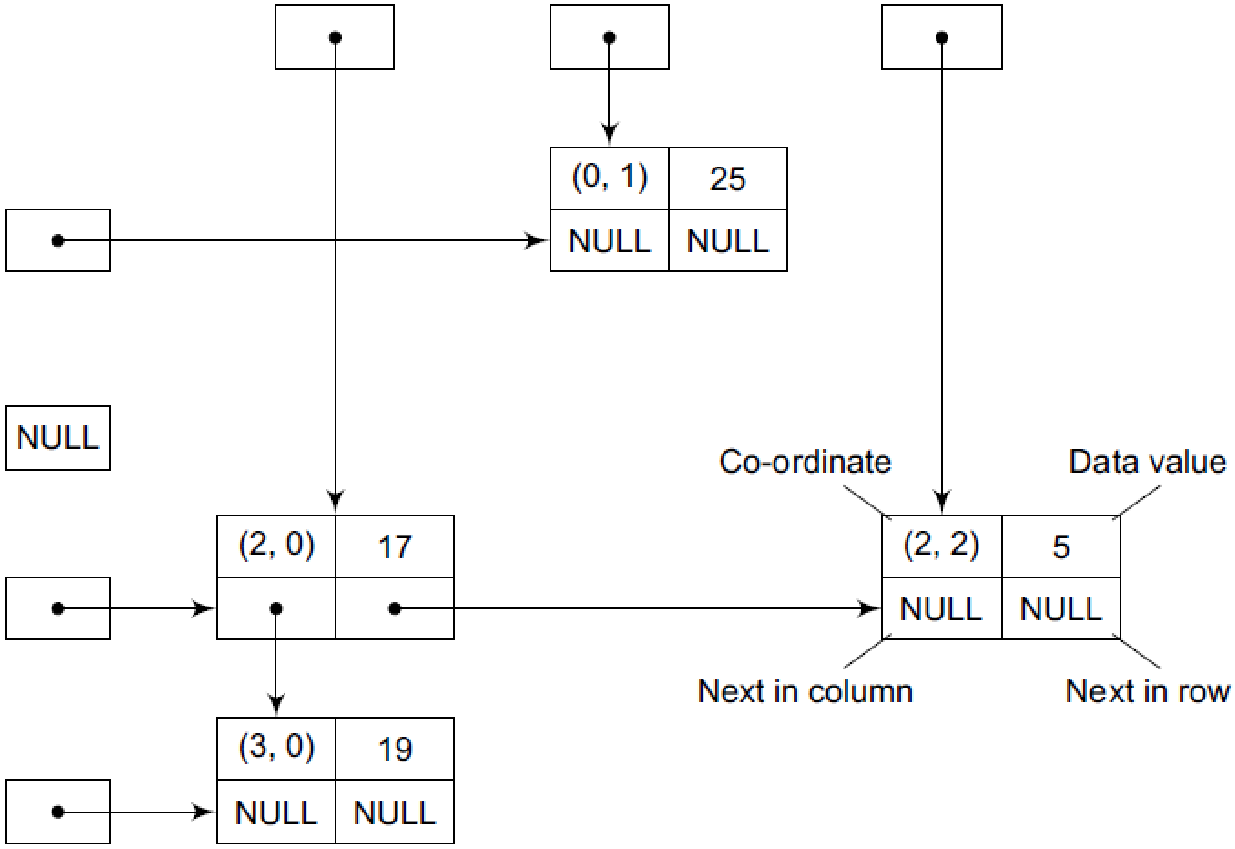


Figure 6.73 Multi-linked representation of sparse matrix shown in Fig. 6.72

APPLICATIONS OF LINKED LISTS

- **6.8.1 Polynomial Representation**
- Consider a polynomial $6x^3 + 9x^2 + 7x + 1$.
- Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.
- Every term of a polynomial can be represented as a node of the linked list.
- Figure 6.74 shows the linked representation of the terms of the above polynomial.

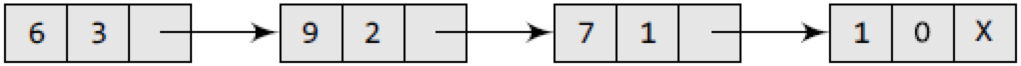


Figure 6.74 Linked representation of a polynomial

References

- Data Structures using C by Reema Thareja