BLM267

Chapter 6: Linked Lists

Data Structures Using C, Second Edition
Reema Thareja

- Introduction
- Singly Linked Lists
- Circular Linked Lists
- Doubly Linked Lists
- Circular Doubly Linked Lists
- Header Linked Lists
- Multi-linked Lists
- Application of Linked Lists

Introduction

- 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.
- For example, if we declare an array as int marks(10), then the array can store a maximum of 10 data elements but not more than that.
- But what if we are not sure of the number of elements in advance?
- 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.

Introduction

- Linked list is a data structure that is free from those restrictions.
- A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.
- However, unlike 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.

Introduction

- Basic terminologies
- A linked list, in simple terms, is a linear collection of data elements.
- These data elements are called nodes.
- Linked list is a data structure which in turn can be used to implement other data structures.
- Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations.
- A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

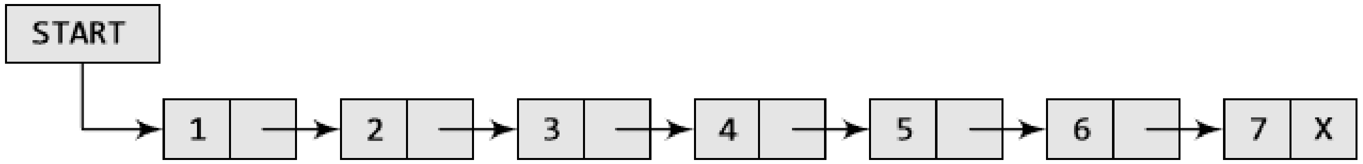


Figure 6.1 Simple linked list

Introduction

- Basic terminologies
- In Fig. 6.1, we can see a linked list in 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 Fig. 6.1, the NULL pointer is represented by X. While programming, we usually define NULL as –1.
- Hence, a NULL pointer denotes the end of the list.
- Since 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.

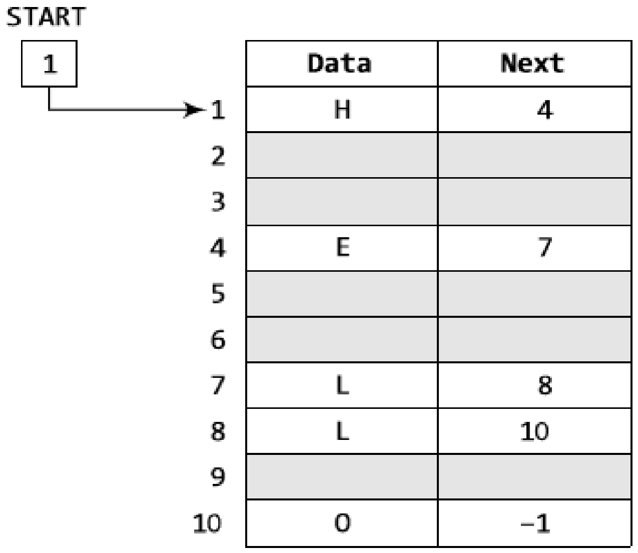
Introduction

- Basic terminologies
- 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 the linked list is empty and contains no nodes.
- In C, we can implement a linked list using the following code:

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

Introduction

- Basic terminologies
- Let us see how a linked list is maintained in the memory.
- 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. Consider Fig. 6.2.



	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

Introduction

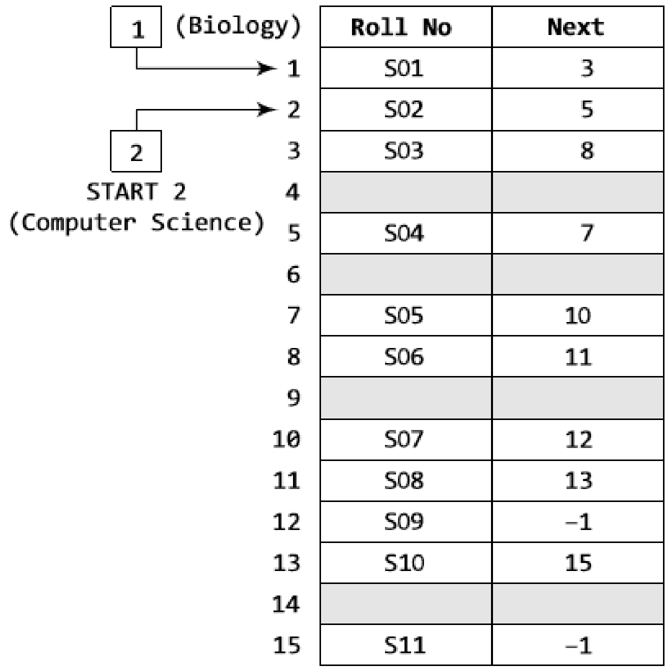
- Basic terminologies
- In the figure, we can see that the variable START is used to store the address of the first node.
- Here, in this example, START = 1, so the first data is stored at address 1, which is H.
- The corresponding NEXT stores the address of the next node, which is 4.
- So, we will look at address 4 to fetch the next data item. 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.

Introduction

- Basic terminologies
- Note that Fig. 6.2 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.
- Let us take another example to see how two linked lists are maintained together in the computer's memory.
- For example, the students of Class XI of Science group are asked to choose between Biology and Computer Science.
- Now, we will maintain two linked lists, one for each subject.
- That is, the first linked list will contain the roll numbers of all the students who have opted for Biology and the second list will contain the roll numbers of students who have chosen Computer Science.

Introduction

- Basic terminologies
- Now, look at Fig. 6.3, 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



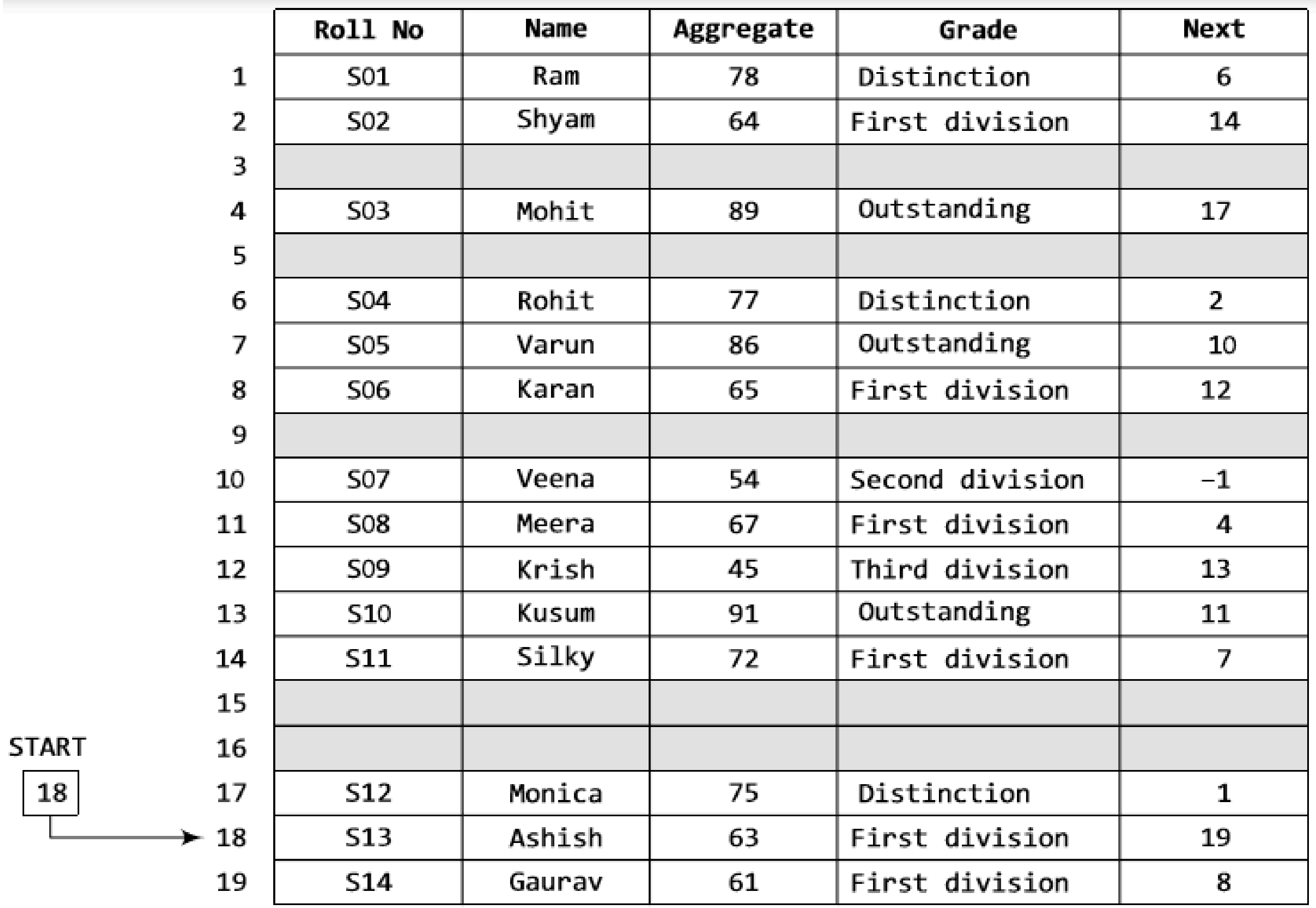
	Roll No	Next
1	S01	3
2	S02	5
3	S03	8
4		
5	S04	7
6		
7	S05	10
8	S06	11
9		
10	S07	12
11	S08	13
12	S09	–1
13	S10	15
14		
15	S11	–1

Figure 6.3 Two linked lists which are simultaneously

Introduction

- Basic terminologies
- 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.

Introduction



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

Introduction

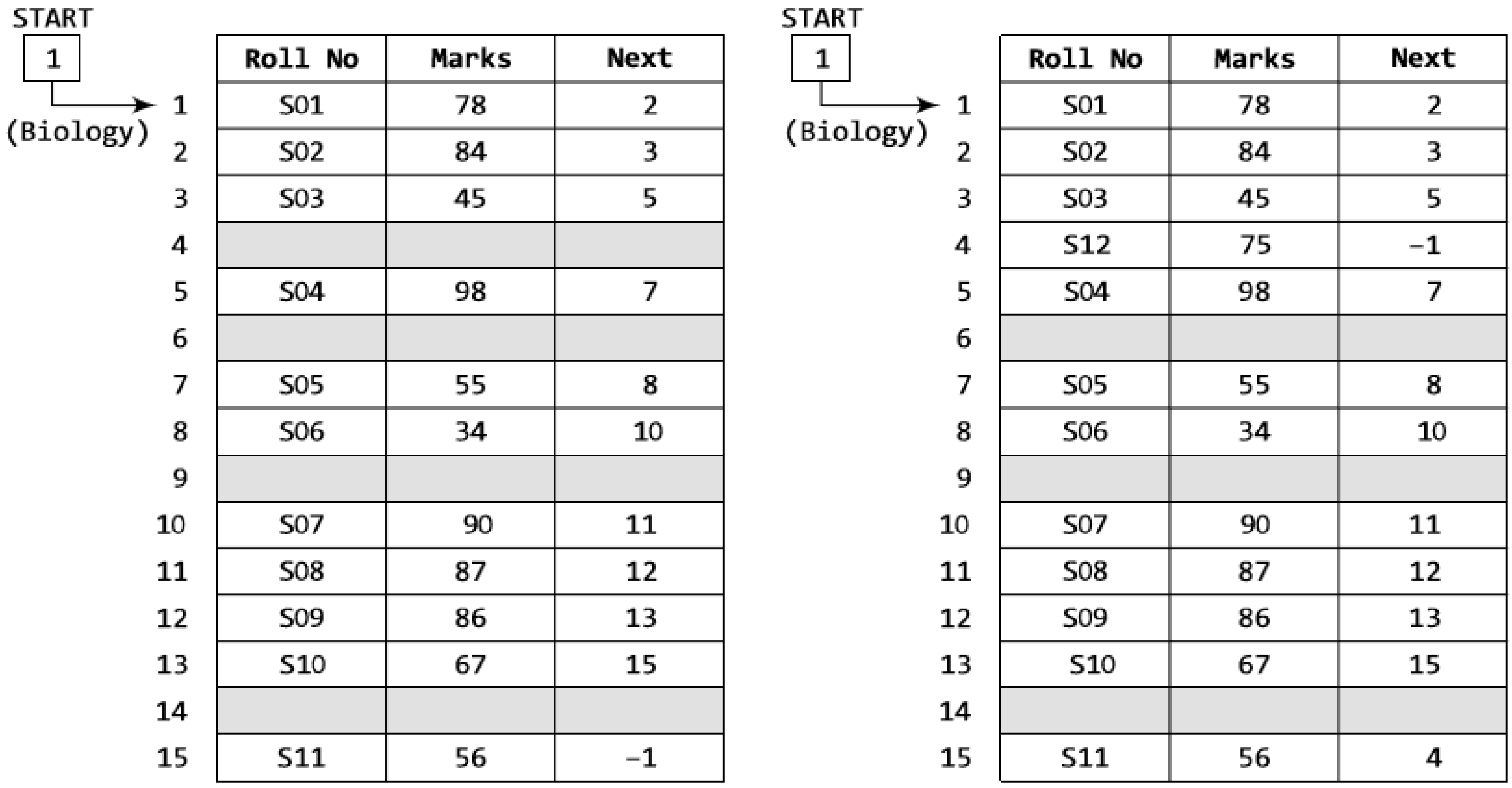
- Linked Lists versus Arrays
- Both arrays and linked lists are a linear collection of data elements.
- But unlike an array, a linked list does not store its nodes in consecutive memory locations.
- Another point of difference between an array and a linked list is that a linked list does not allow random access of data.
- Nodes 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.
- Another advantage of a linked list over an array is that we can add any number of elements in the list.
- This is not possible in case of an array.
- For example, if we declare an array as int marks(20), then the array can store a maximum of 20 data elements only. There is no such restriction in case of a linked list.
- Thus, linked lists provide an efficient way of storing related data and performing basic operations such as insertion, deletion, and updating of information at the cost of extra space required for storing the address of next nodes.

Introduction

- Memory allocation and de-allocation for a Linked List
- We have seen how a linked list is represented in the memory.
- 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.
- For example, consider the linked list shown in Fig. 6.5.
- The linked list contains the roll number of students, marks obtained by them in Biology, and finally a NEXT field which stores the address of the next node in sequence.
- Now, 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).

Introduction

- **Memory allocation and de-allocation for a Linked List**



	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			
5	S04	98	7
6			
7	S05	55	8
8	S06	34	10
9			
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			
15	S11	56	–1

(a)

	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4	S12	75	–1
5	S04	98	7
6			
7	S05	55	8
8	S06	34	10
9			
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			
15	S11	56	4

(b)

Figure 6.5 (a) Students' linked list and (b) linked list after the insertion of new student's record

Introduction

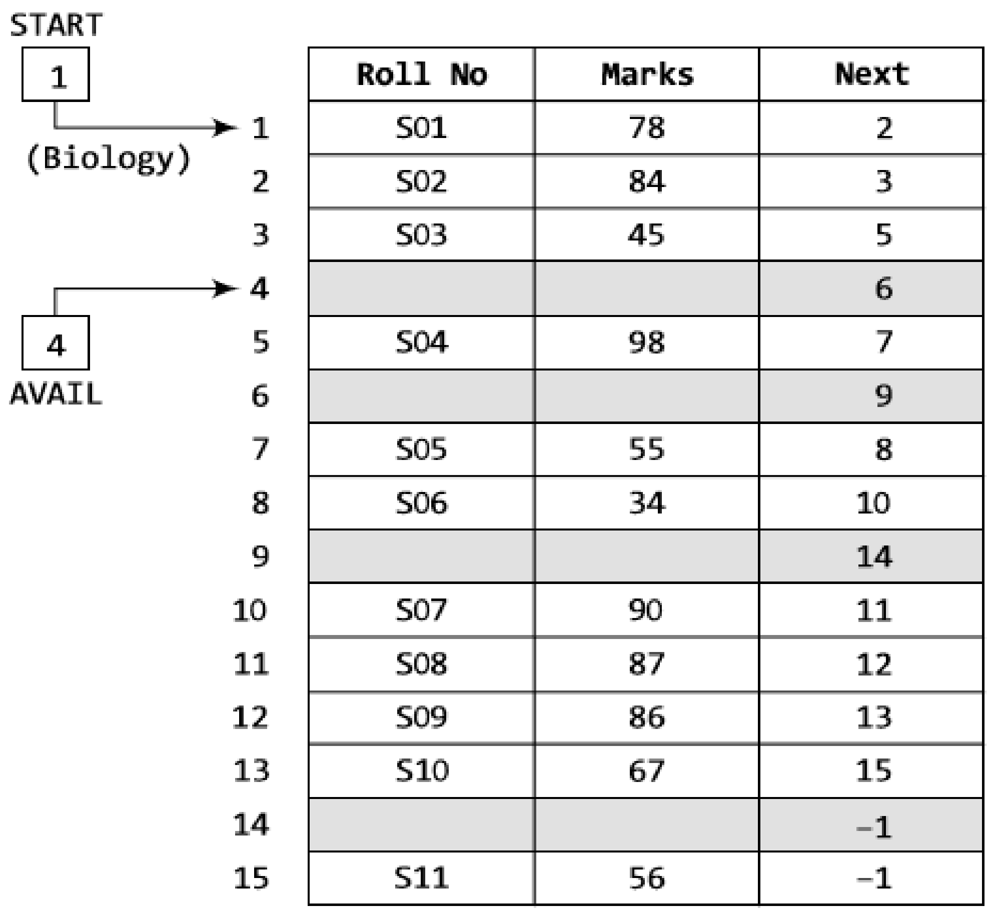
- Memory allocation and de-allocation for a Linked List
- Now, the question is which part of the memory is available and which part is occupied?
- When we delete a node from a linked list, then who changes the status of the memory occupied by it from occupied to available?
- The answer is the operating system.
- Discussing the mechanism of how the operating system does all this is out of the scope of this book.
- So, in simple language, we can say that the computer does it on its own without any intervention from the user or the programmer.
- As a programmer, you just have to take care of the code to perform insertions and deletions in the list.
- However, let us briefly discuss the basic concept behind it.
- The computer maintains a list of all free memory cells. This list of available space is called the free pool.

Introduction

- Memory allocation and de-allocation for a Linked List
- 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.
- 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.

Introduction

- Memory allocation and de-allocation for a Linked List



	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			6
5	S04	98	7
6			9
7	S05	55	8
8	S06	34	10
9			14
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			–1
15	S11	56	–1

Figure 6.6 Linked list with AVAIL and START pointers

Introduction

- Memory allocation and de-allocation for a Linked List
- This was all about inserting a new node in an already existing linked list.
- Now, we will discuss deleting a node or the entire linked list.
- 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.
- 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.

Singly Linked Lists

- A singly linked list 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 shows a singly linked list.

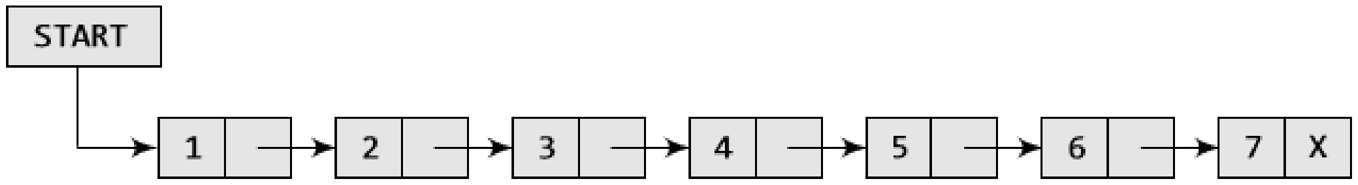


Figure 6.7 Singly linked list

Singly Linked Lists

- Traversing a Linked List
- Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.
- Remember 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.
- The algorithm to traverse a linked list is shown in Fig. 6.8

Singly Linked Lists

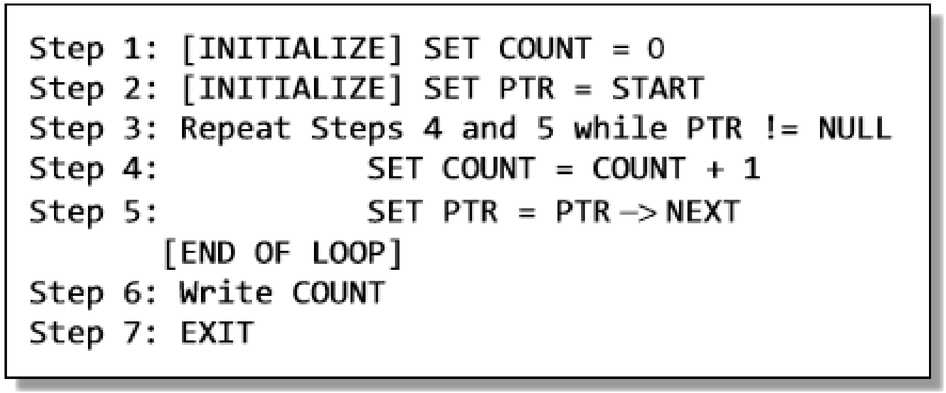
- Traversing a Linked List
- In this algorithm, we first initialize PTR with the address of START.
- So now, PTR points to the first node of the linked list.
- Then in Step 2, a while loop is executed which is repeated till PTR processes the last node, that is until it encounters NULL.
- In Step 3, we apply the process (e.g., print) to the current node, that is, the node pointed by PTR.
- In Step 4, we move to the next node by making the PTR variable point to the node whose address is stored in the NEXT field.

```
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

Singly Linked Lists

- Traversing a Linked List
- Let us now write an algorithm to count the number of nodes in a linked list.
- To do this, we will traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1.
- Once we reach NULL, that is, when all the nodes of the linked list have been traversed, the final value of the counter will be displayed.
- Figure 6.9 shows the algorithm to print the number of 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
```

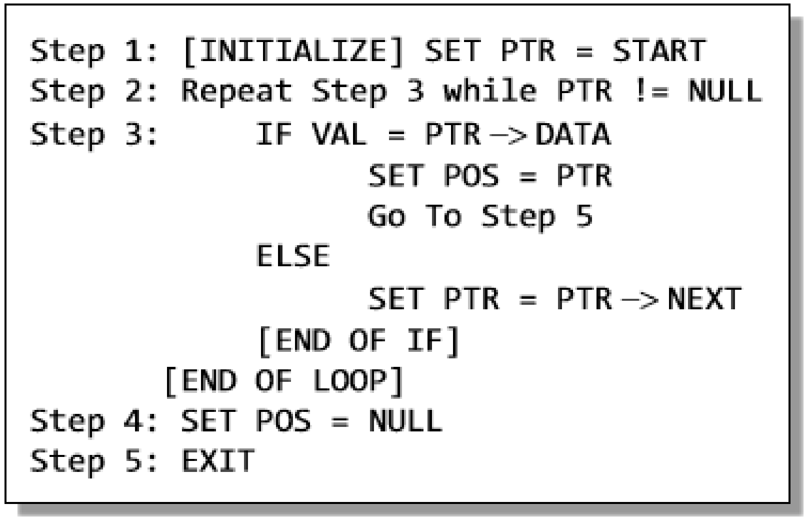
Figure 6.9 Algorithm to print the number of nodes in a linked list

Singly Linked Lists

- Searching for a value in a Linked List
- Searching a linked list means to find a particular element in the linked list.
- As already discussed, a linked list consists of nodes which are divided into two parts, the information part and the next part.
- So searching means finding whether a given value is present in the information part of the node or not.
- If it is present, the algorithm returns the address of the node that contains the value.

Singly Linked Lists

- Searching for a value in a Linked List
- Figure 6.10 shows the algorithm to search a linked list.
- In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node.
- In Step 2, a while loop is executed which will compare every node's DATA with VAL for which the search is being made.
- If the search is successful, that is, VAL has been found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm.
- However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

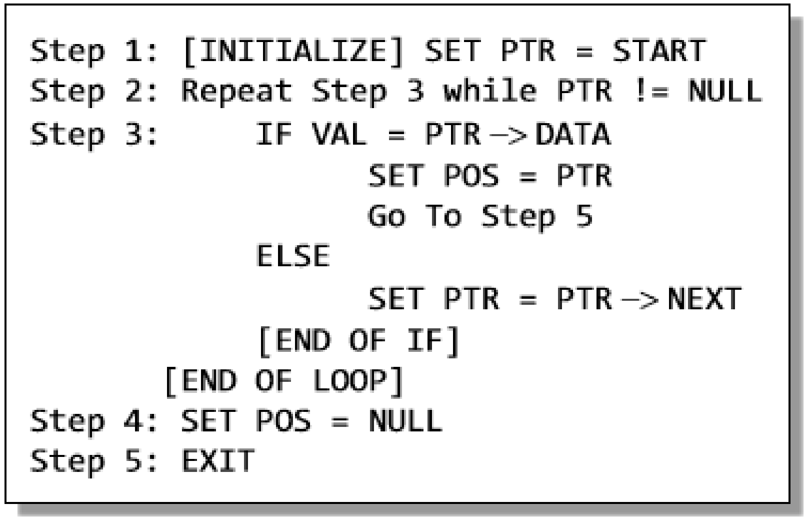


```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:      IF VAL = PTR -> DATA
                   SET POS = PTR
                   Go To Step 5
             ELSE
                   SET PTR = PTR -> NEXT
             [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Figure 6.10 Algorithm to search a linked list

Singly Linked Lists

- Searching for a value in a Linked List
- Figure 6.10 shows the algorithm to search a linked list.
- In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node.
- In Step 2, a while loop is executed which will compare every node's DATA with VAL for which the search is being made.
- If the search is successful, that is, VAL has been found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm.
- However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

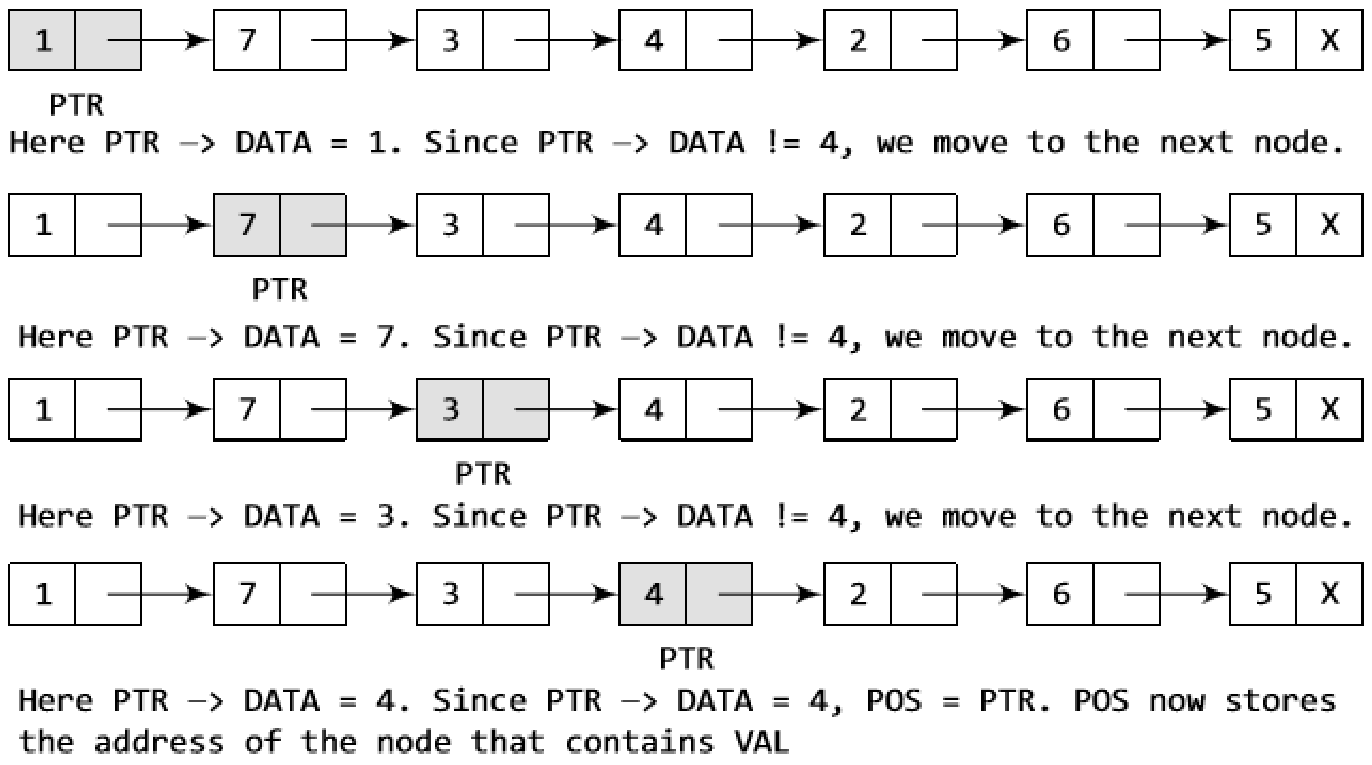


```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:       IF VAL = PTR -> DATA
                    SET POS = PTR
                    Go To Step 5
              ELSE
                    SET PTR = PTR -> NEXT
              [END OF IF]
         [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Figure 6.10 Algorithm to search a linked list

Singly Linked Lists

- Searching for a value in a Linked List
- Consider the linked list shown in Fig. 6.11. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.



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

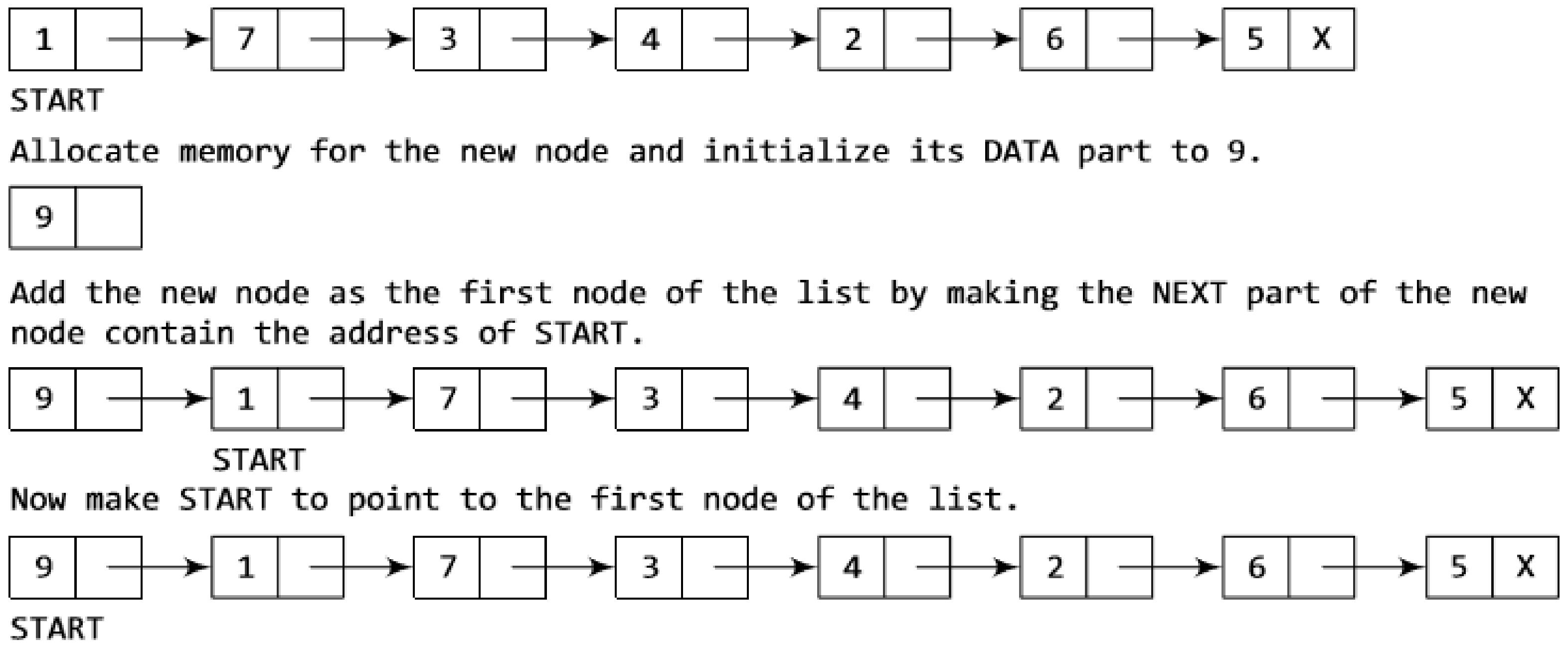
Figure 6.11 Searching a linked list

Singly Linked Lists

- Inserting a new node in a Linked List
- In this section, we will see how a new node is added into an already existing 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.
- Before we describe the algorithms to perform insertions in all these four cases, let us first discuss an important term called OVERFLOW.
- 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.

Singly Linked Lists

- Inserting a Node at the Beginning of a Linked List
- Consider the linked list shown in Fig. 6.12.
- Suppose we want to add a new node with data 9 and add it as the first node of the list.
- Then the following changes will be done in the linked list.


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

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

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

Singly Linked Lists

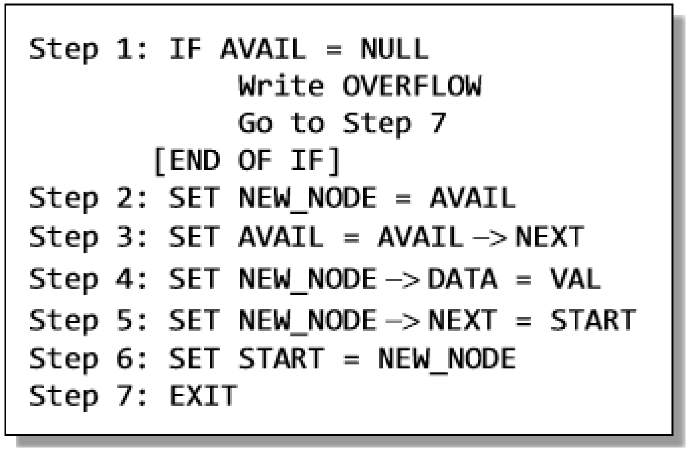
- Inserting a Node at the Beginning of a Linked List
- Figure 6.13 shows the algorithm to insert a new node at the beginning of a linked list.
- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if a free memory cell is available, then we allocate space for the new node.

```
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

Singly Linked Lists

- Inserting a Node at the Beginning of a Linked List
- Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE.
- Note the following two steps:
- Step 2: SET NEW_NODE = AVAIL
- Step 3: SET AVAIL = AVAIL -> NEXT
- These steps allocate memory for the new node.
- In C, there are functions like malloc(), alloc, and calloc() which automatically do the memory allocation on behalf of the user.



```
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

Singly Linked Lists

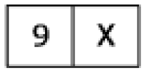
- Inserting a Node at the End of a Linked List
- Consider the linked list shown in Fig. 6.14.
- Suppose we want to add a new node with data 9 as the last node of the list.
- Then the following changes will be done in the linked list.
- Figure 6.15 shows the algorithm to insert a new node at the end of a linked list.
- In Step 6, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node.
- Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

Singly Linked Lists

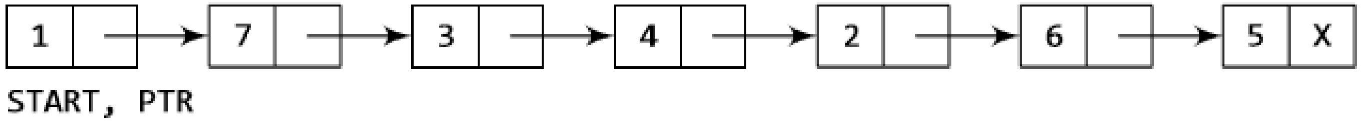
Inserting a Node at the End of a Linked List

START

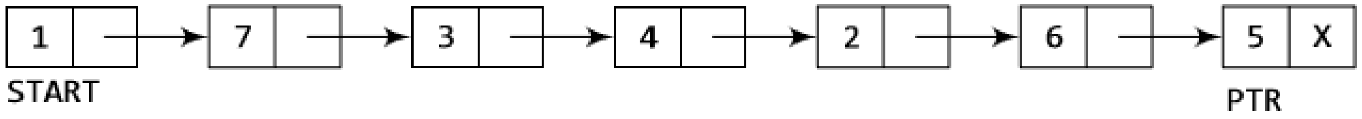
Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.



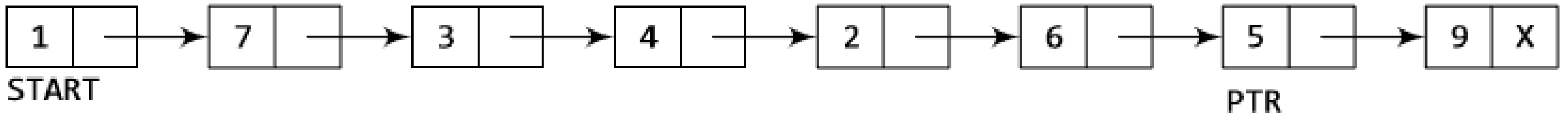
Take a pointer variable PTR which points to START.



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.



Singly Linked Lists

Inserting a Node at the End of a 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 = 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

Singly Linked Lists

- Inserting a Node After a Given Node in a Linked List
- Consider the linked list shown in Fig. 6.17.
- Suppose we want to add a new node with value 9 after the node containing data 3.
- Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 6.16.

```
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
```

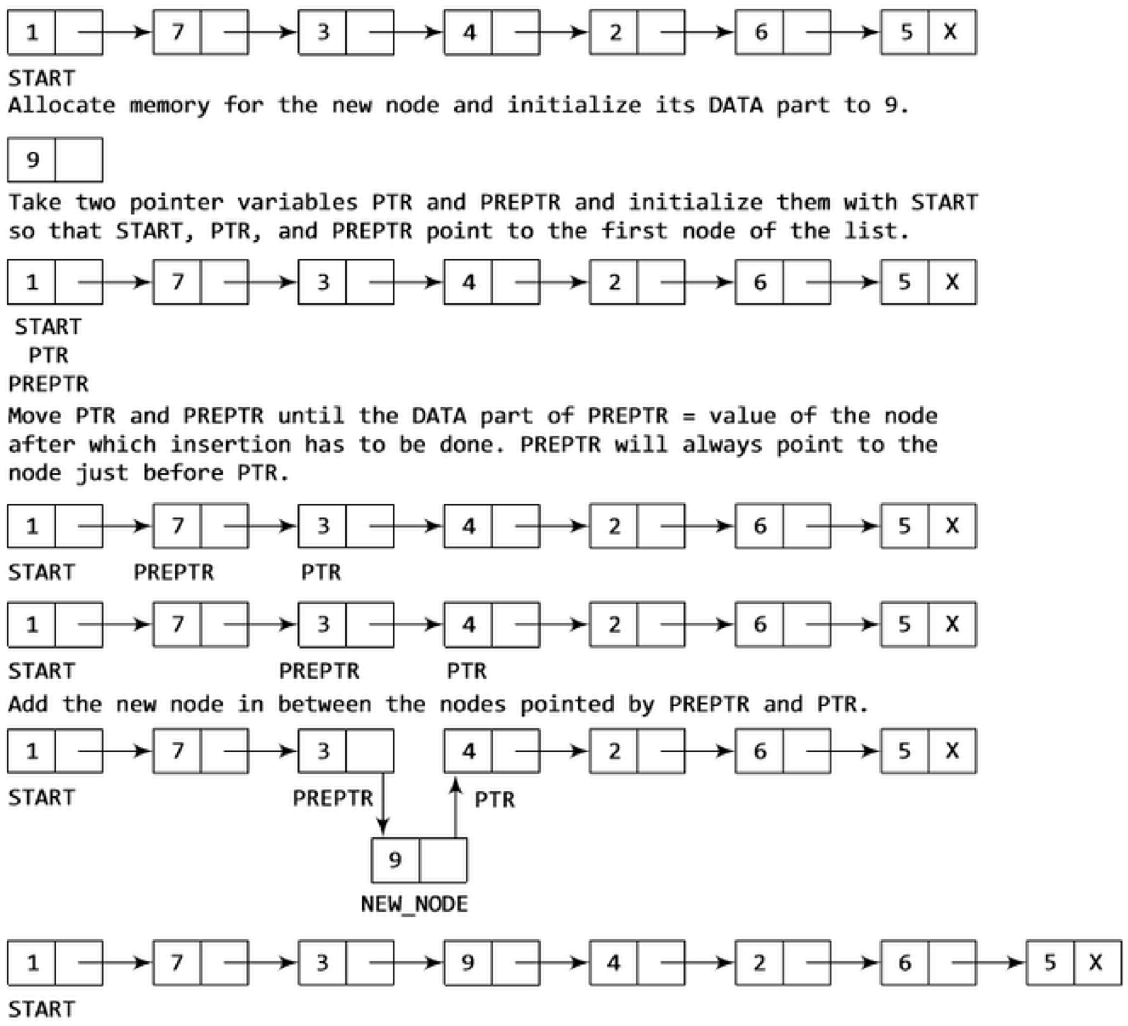
Figure 6.16 Algorithm to insert a new node after a node that has value NUM

Singly Linked Lists

- Inserting a Node After a Given Node in a Linked List
- In Step 5, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR.
- Initially, PREPTR is initialized to PTR.
- So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted after this node.
- Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.

Singly Linked Lists

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



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.

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

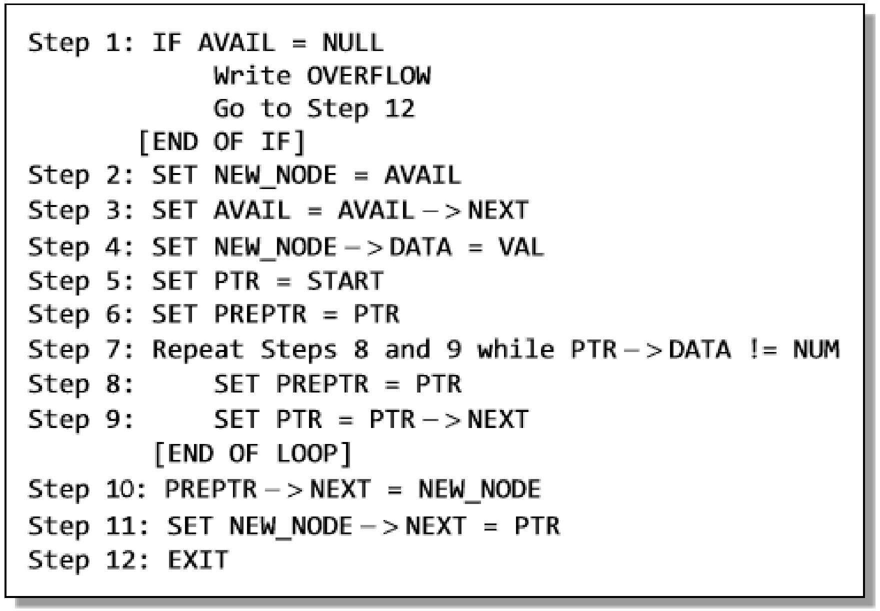
Figure 6.17 Inserting an element after a given node in a linked list

Singly Linked Lists

- Inserting a Node Before a Given Node in a Linked List
- Consider the linked list shown in Fig. 6.19.
- Suppose we want to add a new node with value 9 before the node containing 3.
- Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 6.18.
- In Step 5, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- Then, we take another pointer variable PREPTR and initialize it with PTR.
- So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.

Singly Linked Lists

- Inserting a Node Before a Given Node in a Linked List
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted before this node.
- Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that the new node is in



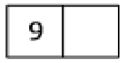
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR -> DATA != NUM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

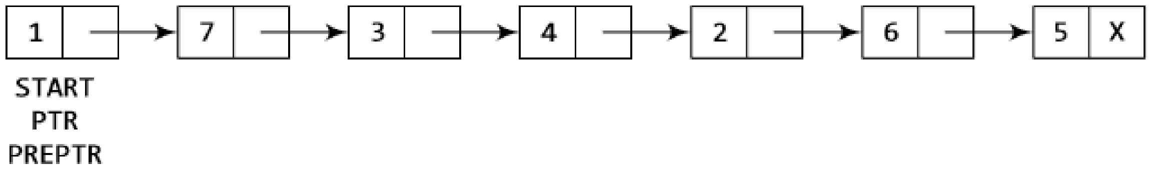
Singly Linked Lists

- **Inserting a Node Before a Given Node in a Linked List**

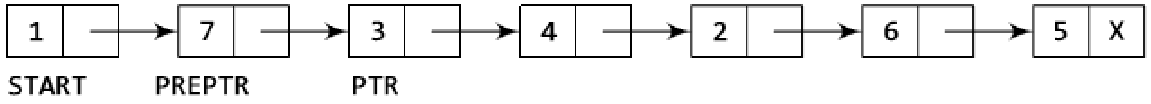
START
Allocate memory for the new node and initialize its DATA part to 9.



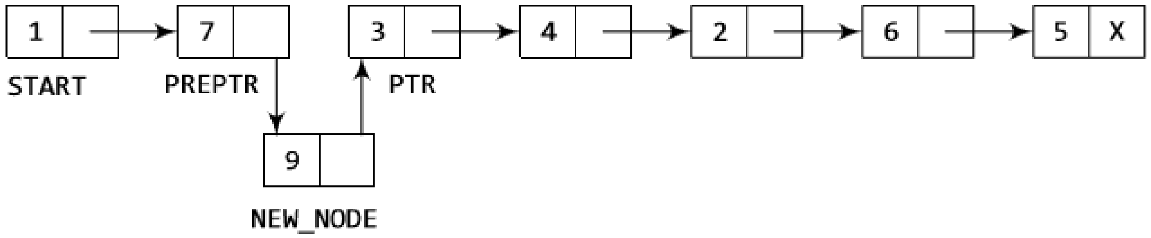
Initialize PREPTR and PTR to the START node.

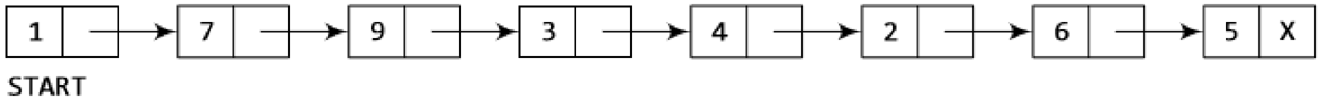


Move PTR and PREPTR until the DATA part of PTR = value of the node before which insertion has to be done. PREPTR will always point to the node just before PTR.



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





Singly Linked Lists

- Deleting a node from a Linked List
- In this section, we will discuss how a node is deleted from an already existing linked list.
- We will consider three 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.
- Before we describe the algorithms in all these three cases, let us first discuss an important term called UNDERFLOW.
- 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.
- Note that 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.
- Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated.

Singly Linked Lists

- Deleting the First Node from a Linked List
- Consider the linked list in Fig. 6.20.
- When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

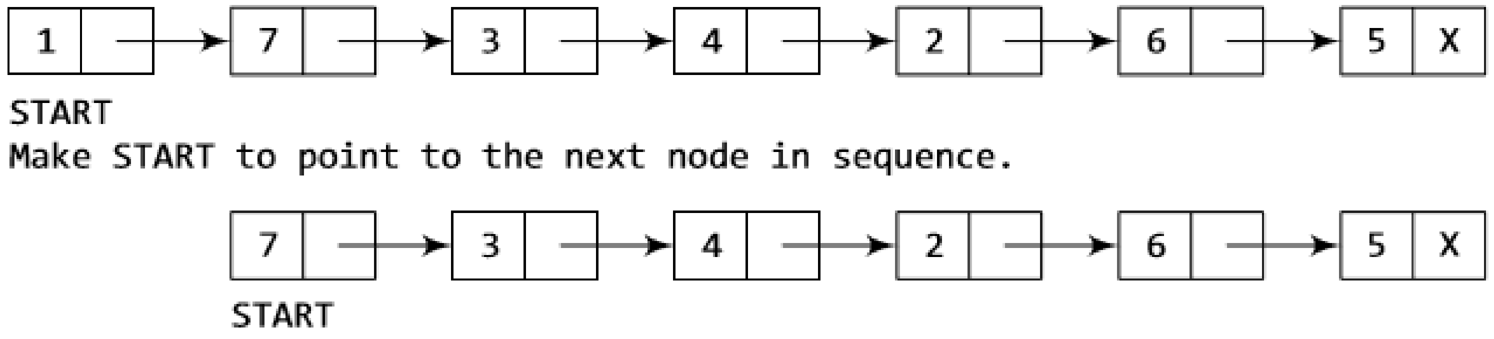
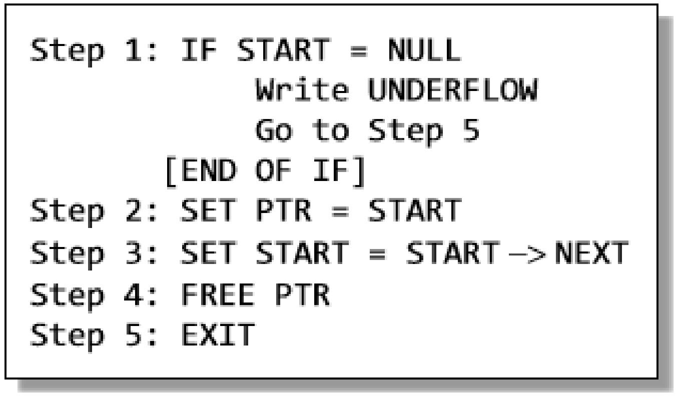


Figure 6.20 Deleting the first node of a linked list

Singly Linked Lists

- Deleting the First Node from a Linked List
- Figure 6.21 shows the algorithm to delete the first node from a linked list.
- In Step 1, we check if the linked list exists or not.
- If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list.
- For this, we initialize PTR with START that stores the address of the first node of the list.
- In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

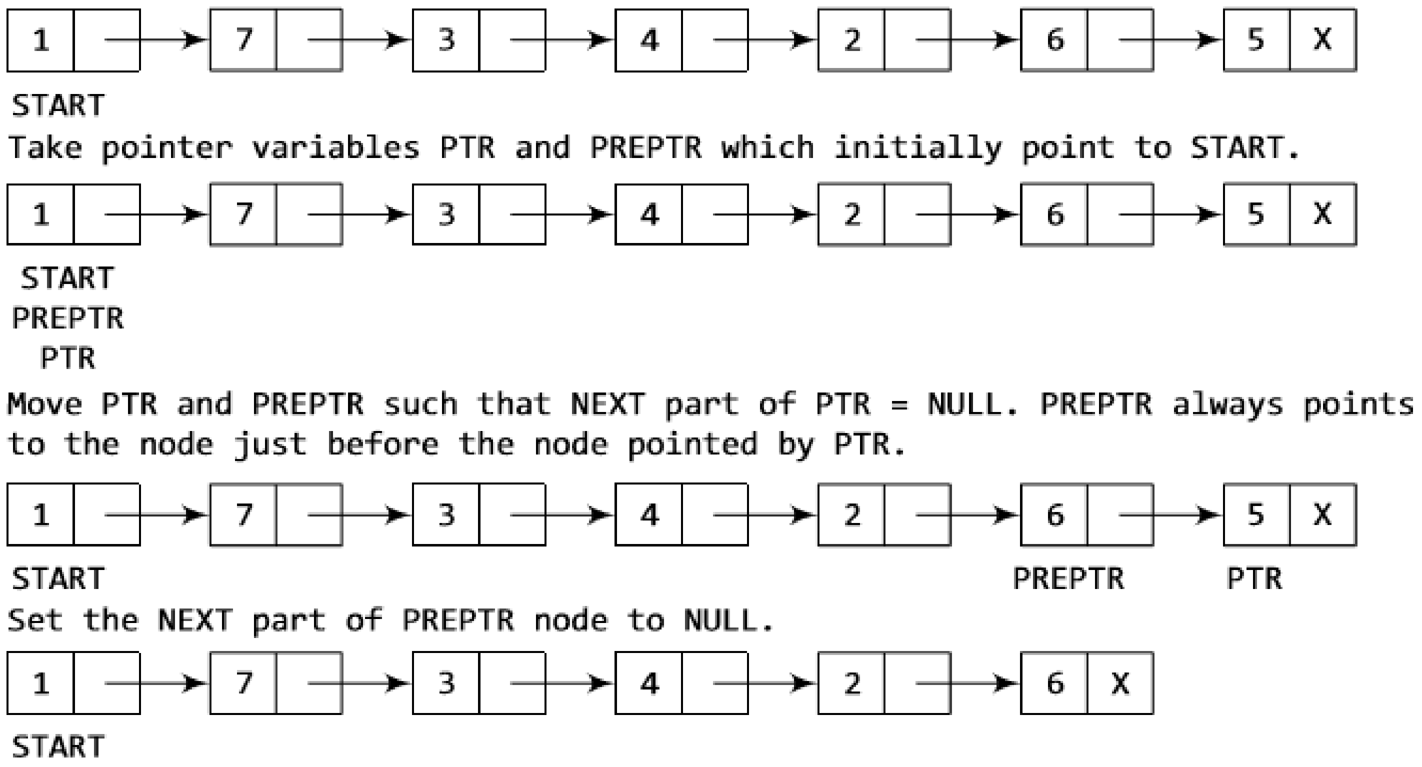


```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
       [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Figure 6.21 Algorithm to delete the first node

Singly Linked Lists

- Deleting the Last Node from a Linked List
- Consider the linked list shown in Fig. 6.22.
- Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.



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

Singly Linked Lists

- Deleting the Last Node from a Linked List
- Figure 6.23 shows the algorithm to delete the last node from a linked list.
- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned back to the free pool.

```
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
```

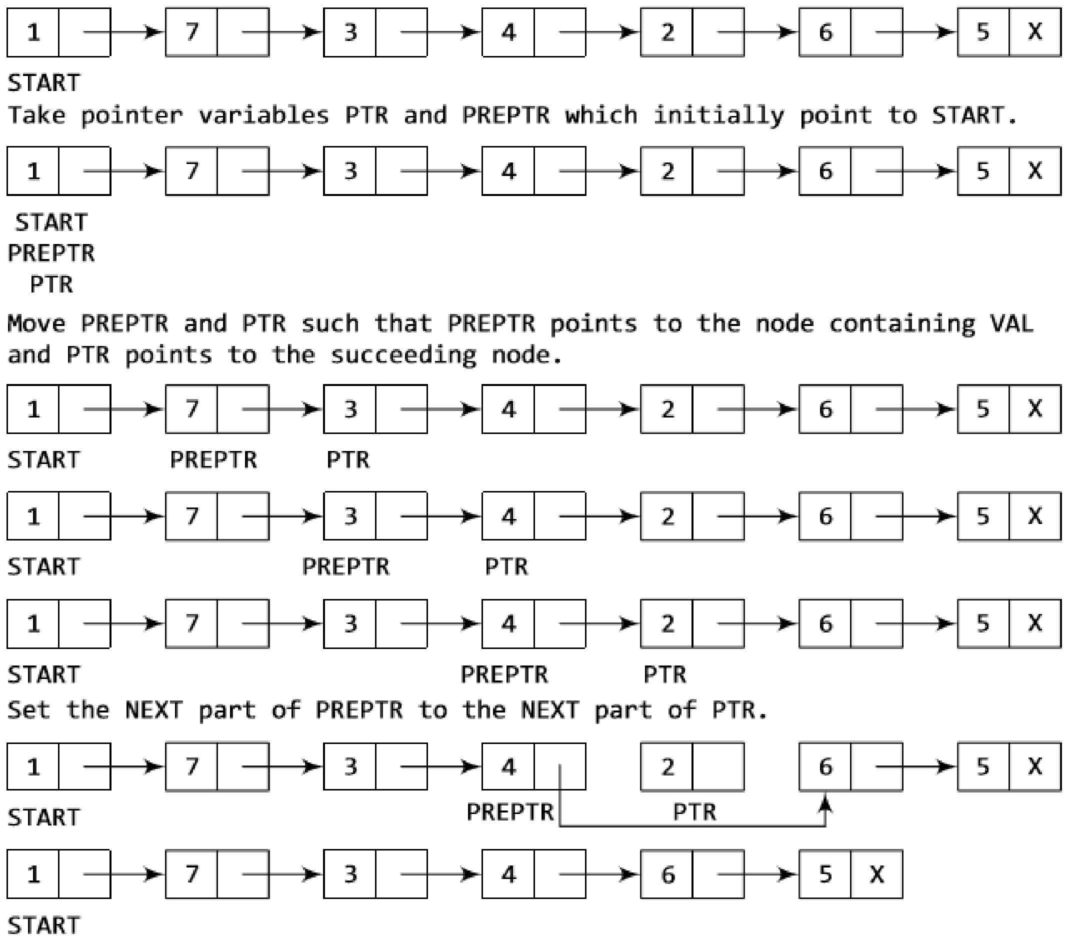
Figure 6.23 Algorithm to delete the last node

Singly Linked Lists

- Deleting the Node After a Given Node in a Linked List
- Consider the linked list shown in Fig. 6.24.
- Suppose we want to delete the node that succeeds the node which contains data value 4.
- Then the following changes will be done in the linked list.
- Figure 6.25 shows the algorithm to delete the node after a given node from a linked list.
- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it.
- The memory of the node succeeding the given node is freed and returned back to the free pool.

Singly Linked Lists

- **Deleting the Node After a Given Node in a Linked List**


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.

Set the NEXT part of PREPTR to the NEXT part of PTR.

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

Singly Linked Lists

- **Deleting the Node After a Given Node in a Linked List**

```
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

Singly Linked Lists

```
struct node *create_ll(struct node *start)
{
        struct node *new_node, *ptr;
        int num;
        printf("\n Enter -1 to end");
        printf("\n Enter the data : ");
        scanf("%d", &num);
        while(num!=-1)
        {
                new_node = (struct node*)malloc(sizeof(struct node));
                new_node -> data=num;
                if(start==NULL)
                {
                        new_node -> next = NULL;
                        start = new_node;
                }
                else
                {
                        ptr=start;

                        while(ptr->next!=NULL)
                        ptr=ptr->next;
                        ptr->next = new_node;
                        new_node->next=NULL;
                }
                printf("\n Enter the data : ");
                scanf("%d", &num);
        }
        return start;
}
```

Singly Linked Lists

```
struct node *display(struct node *start)
{
        struct node *ptr;
        ptr = start;
        while(ptr != NULL)
        {
                printf("\t %d", ptr -> data);
                ptr = ptr -> next;
        }
        return start;
}
```

Singly Linked Lists

```
struct node *insert_beg(struct node *start)
{
        struct node *new_node;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        new_node -> next = start;
        start = new_node;
        return start;
}
```

Singly Linked Lists

```
struct node *insert_end(struct node *start)
{
        struct node *ptr, *new_node;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        new_node -> next = NULL;
        ptr = start;
        while(ptr -> next != NULL)
        ptr = ptr -> next;
        ptr -> next = new_node;
        return start;
}
```

Singly Linked Lists

```
struct node *insert_after(struct node *start)
{
        struct node *new_node, *ptr, *preptr;
        int num, val;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        printf("\n Enter the value after which the data has to be inserted : ");
        scanf("%d", &val);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        ptr = start;
        preptr = ptr;
        while(preptr -> data != val)
        {
                preptr = ptr;
                ptr = ptr -> next;
        }
        preptr -> next=new_node;
        new_node -> next = ptr;
        return start;
}
```

Singly Linked Lists

```
struct node *delete_beg(struct node *start)
{
        struct node *ptr;
        ptr = start;
        start = start -> next;
        free(ptr);
        return start;
}
```

```
{
        struct node *ptr, *preptr;
        ptr = start;
        while(ptr -> next != NULL)
        {
                preptr = ptr;
                ptr = ptr -> next;
        }
        preptr -> next = NULL;
        free(ptr);
        return start;
```

Singly Linked Lists

```
struct node *delete_after(struct node *start)
{
        struct node *ptr, *preptr;
        int val;
        printf("\n Enter the value after which the node has to deleted : ");
        scanf("%d", &val);
        ptr = start;
        preptr = ptr;
        while(preptr -> data != val)
        {
                preptr = ptr;
                ptr = ptr -> next;
        }
        preptr -> next=ptr -> next;
        free(ptr);
        return start;
}
```

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 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.

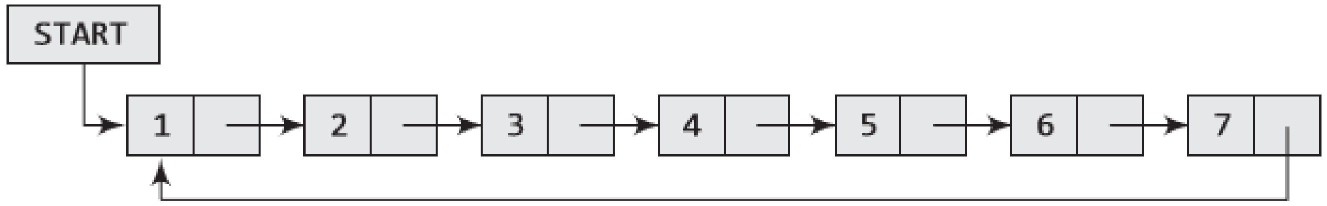


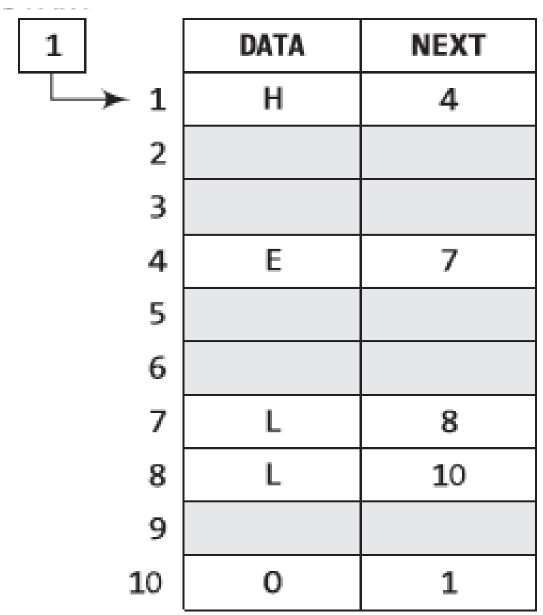
Figure 6.26 Circular linked list

Circular Linked Lists

- 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.
- Circular linked lists are widely used in operating systems for task maintenance.
- We will now discuss an example where a circular linked list is used.
- When we are surfing the Internet, we can use the Back button and the Forward button to move to the previous pages that we have already visited.
- How is this done? The answer is simple.
- A circular linked list is used to maintain the sequence of the Web pages visited. Traversing this circular linked list either in forward or backward direction helps to revisit the pages again using Back and Forward buttons.
- Actually, this is done using either the circular stack or the circular queue.

Circular Linked Lists

- 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.

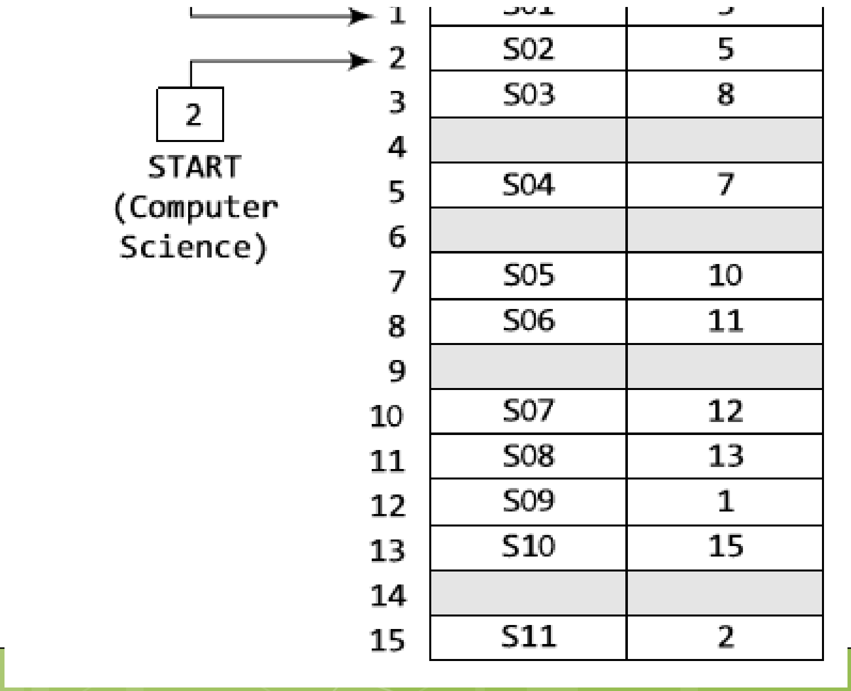


	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	1

Figure 6.27 Memory representation

Circular Linked Lists

- 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.



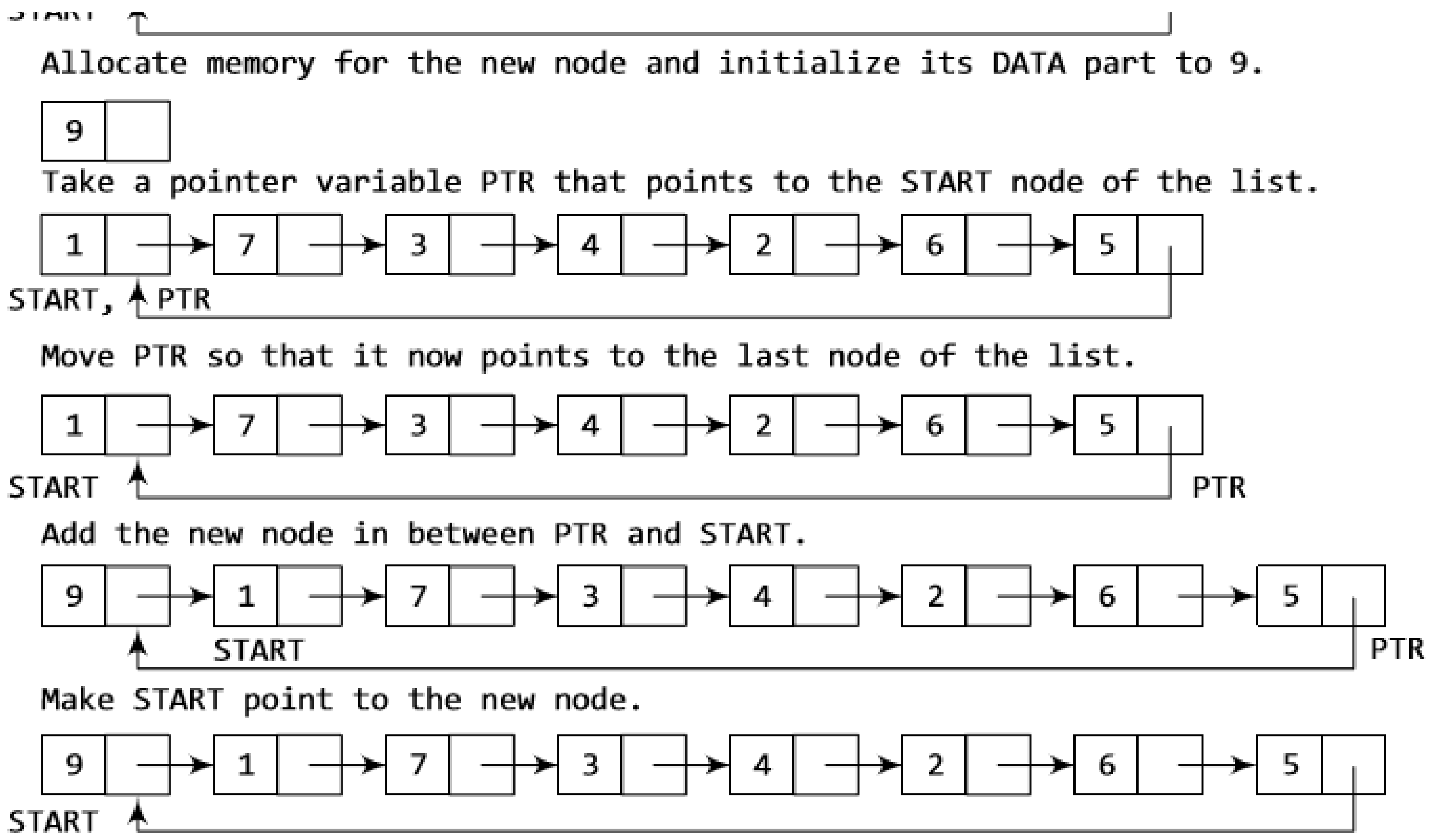
1	S01	3
2	S02	5
3	S03	8
4		
5	S04	7
6		
7	S05	10
8	S06	11
9		
10	S07	12
11	S08	13
12	S09	1
13	S10	15
14		
15	S11	2

Circular Linked Lists

- Inserting a new node in a circular Linked List
- In this section, we will see how a new node is added into an already existing 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.

Circular Linked Lists

- **Inserting a Node at the Beginning of a Circular Linked List**
- **Consider the linked list shown in Fig. 6.29.**
- **Suppose we want to add a new node with data 9 as the first node of the list.**
- **Then the following changes will be done in the 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.

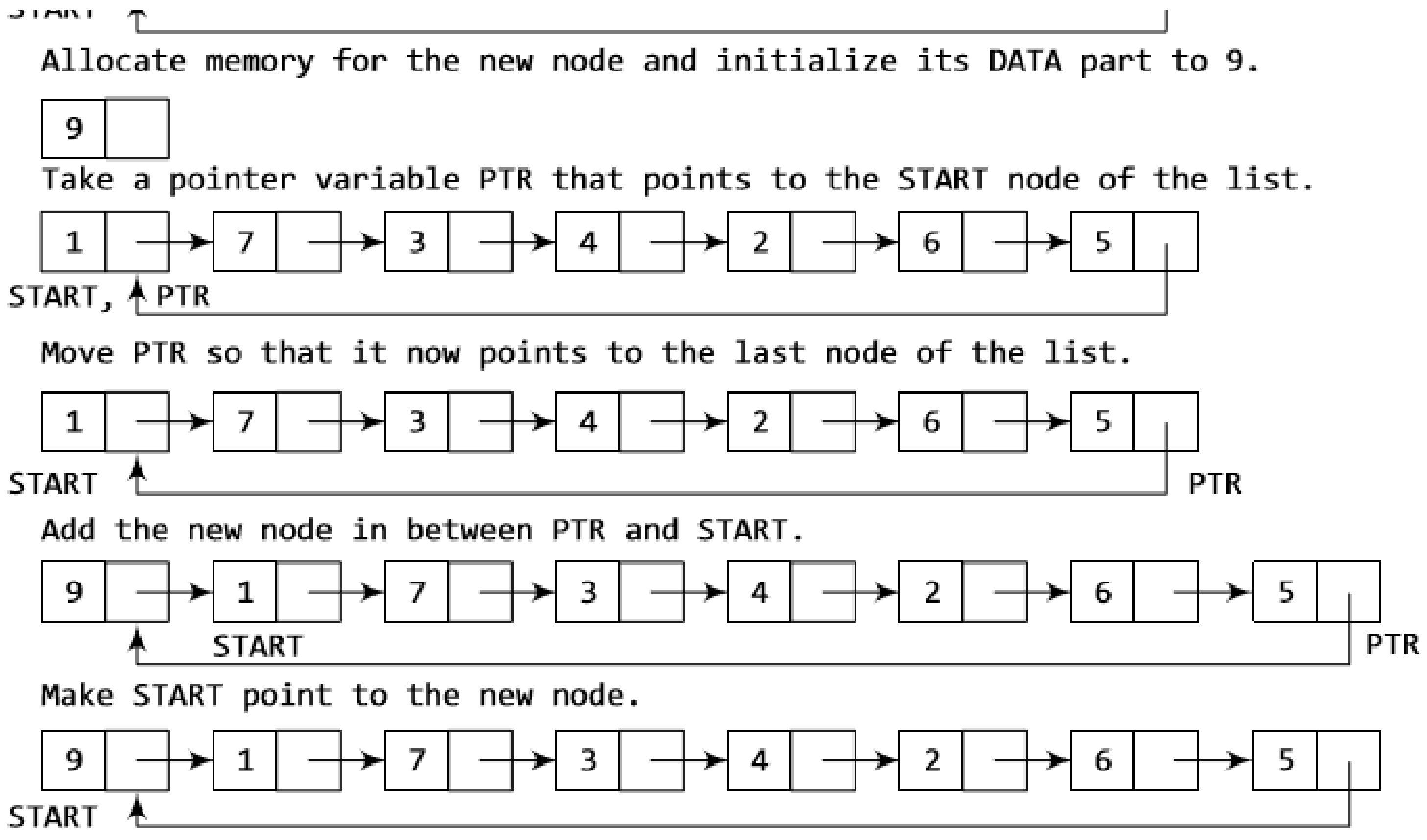
Move PTR so that it now points to the last node of the list.

Add the new node in between PTR and START.

Make START point to the new node.

Circular Linked Lists

- Inserting a Node at the Beginning of a Circular Linked List
- Consider the linked list shown in Fig. 6.29.
- Suppose we want to add a new node with data 9 as the first node of the list.
- Then the following changes will be done in the linked list.


Allocate memory for the new node and initialize its DATA part to 9.
9
Take a pointer variable PTR that points to the START node of the list.
1 7 3 4 2 6 5
START, PTR
Move PTR so that it now points to the last node of the list.
1 7 3 4 2 6 5
START
PTR
Add the new node in between PTR and START.
9 1 7 3 4 2 6 5
START
PTR
Make START point to the new node.
9 1 7 3 4 2 6 5
START

Circular Linked Lists

- Inserting a Node at the Beginning of a Circular Linked List
- Figure 6.30 shows the algorithm to insert a new node at the beginning of a linked list.
- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if free memory cell is available, then we allocate space for the new node.
- Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE.
- While inserting a node in a circular linked list, we have to use a while loop to traverse to the last node of the list.
- Because the last node contains a pointer to START, its NEXT field is updated so that after insertion it points to the new node which will be now known as START.

Circular Linked Lists

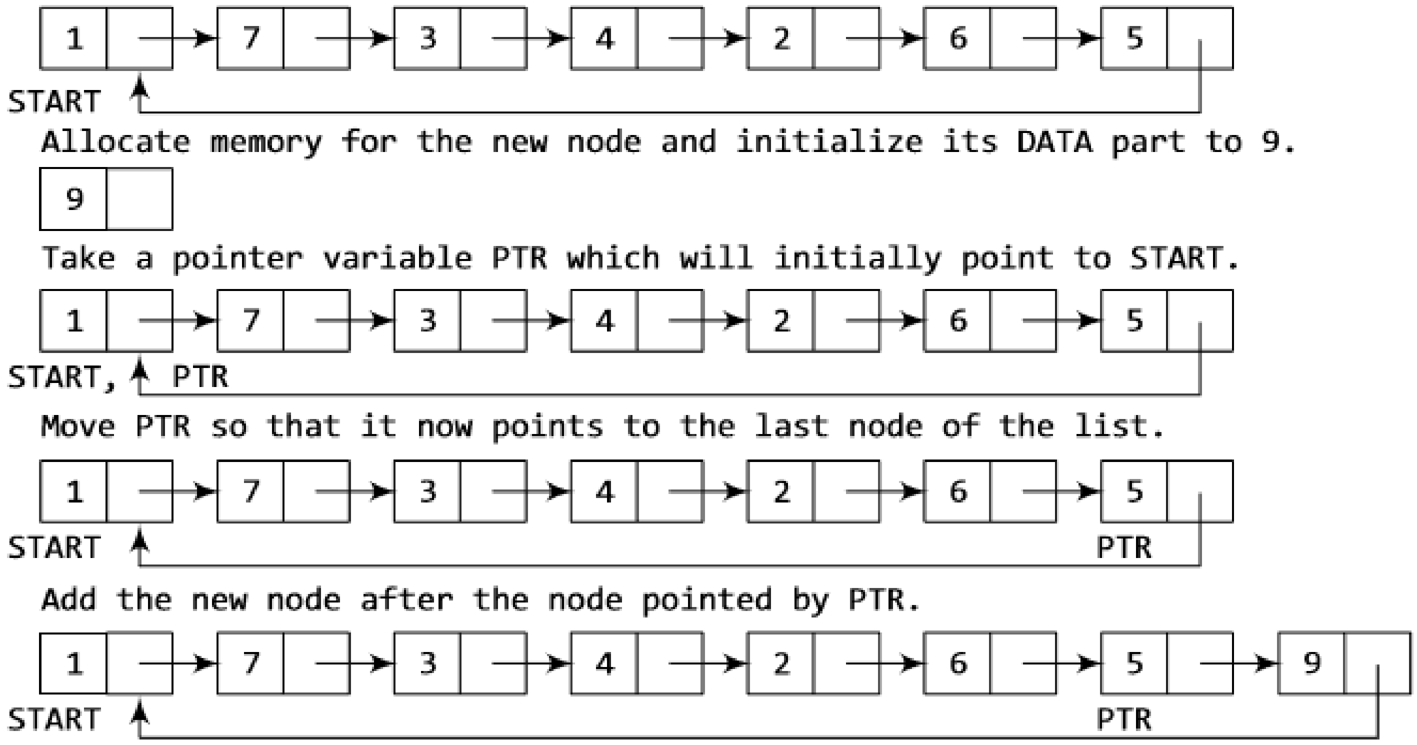
- **Inserting a Node at the Beginning of a Circular Linked List**

```
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

Circular Linked Lists

- Inserting a Node at the End of a Circular Linked List
- Consider the linked list shown in Fig. 6.31.
- Suppose we want to add a new node with data 9 as the last node of the list.
- Then the following changes will be done in the linked lis

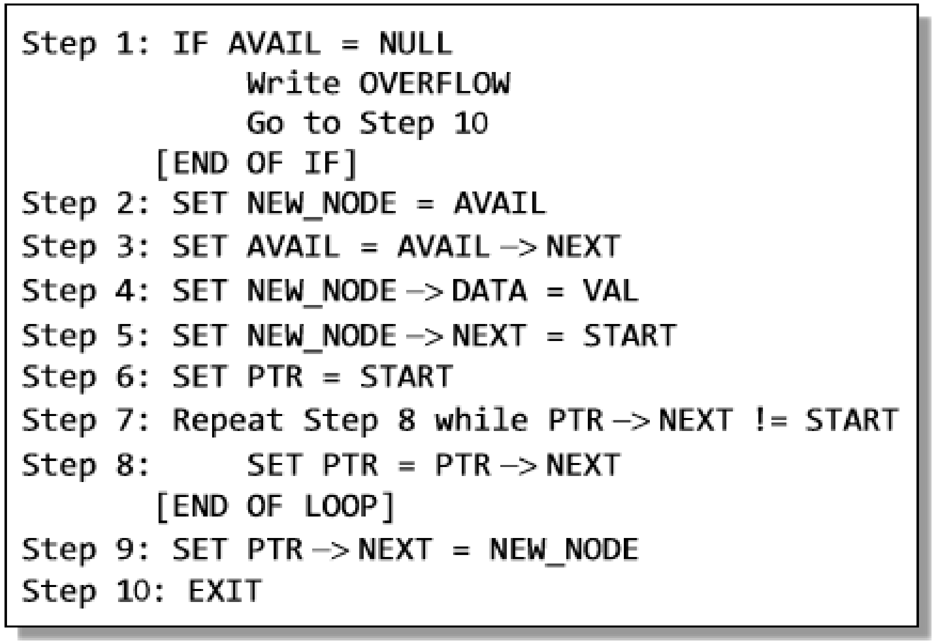


1 7 3 4 2 6 5
START
Allocate memory for the new node and initialize its DATA part to 9.
9
Take a pointer variable PTR which will initially point to START.
START, PTR
Move PTR so that it now points to the last node of the list.
PTR
Add the new node after the node pointed by PTR.
9

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

Circular Linked Lists

- Inserting a Node at the End of a Circular Linked List
- Figure 6.32 shows the algorithm to insert a new node at the end of a circular linked list.
- In Step 6, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node.
- Remember that the NEXT field of the new node contains the address of the first node which is denoted by START.



```
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
```

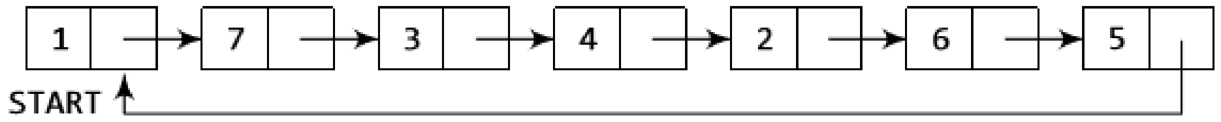
Figure 6.32 Algorithm to insert a new node at the end

Circular Linked Lists

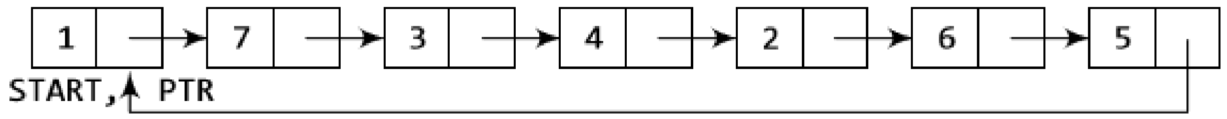
- Deleting a node from a circular Linked List
- In this section, we will discuss how a node is deleted from an already existing circular linked list.
- We will take two cases and then see how deletion is done in each case.
- Rest of the cases of deletion are same as that given for singly linked lists.
- Case 1: The first node is deleted.
- Case 2: The last node is deleted.

Circular Linked Lists

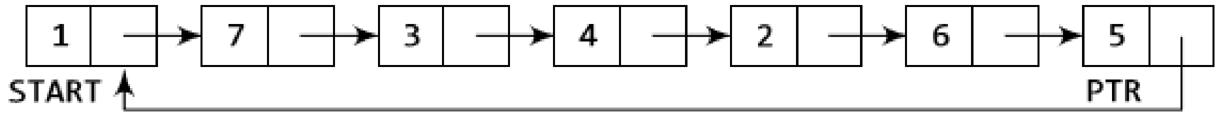
- Deleting the First Node from a Circular Linked List
- Consider the circular linked list shown in Fig. 6.33.
- When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.



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.

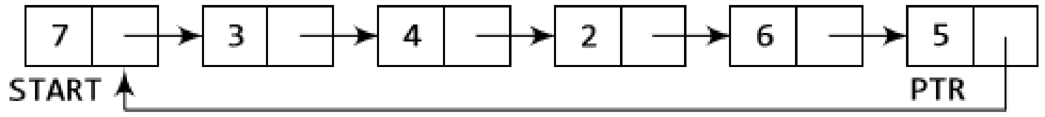


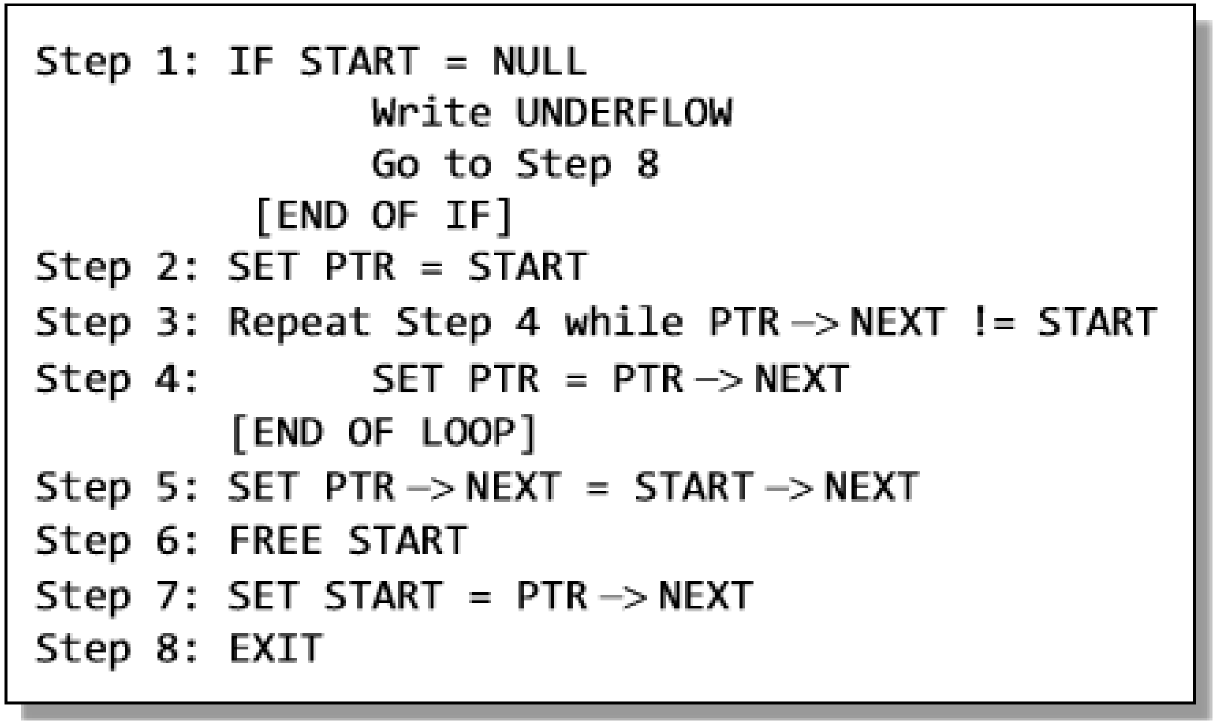
Figure 6.33 Deleting the first node from a circular linked list

Circular Linked Lists

- Deleting the First Node from a Circular Linked List
- Figure 6.34 shows the algorithm to delete the first node from a circular linked list.
- In Step 1 of the algorithm, we check if the linked list exists or not.
- If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a pointer variable PTR which will be used to traverse the list to ultimately reach the last node.
- In Step 5, we change the next pointer of the last node to point to the second node of the circular linked list.
- In Step 6, the memory occupied by the first node is freed.
- Finally, in Step 7, the second node now becomes the first node of the list and its address is stored in the pointer variable START.

Circular Linked Lists

- **Deleting the First Node from a Circular Linked List**

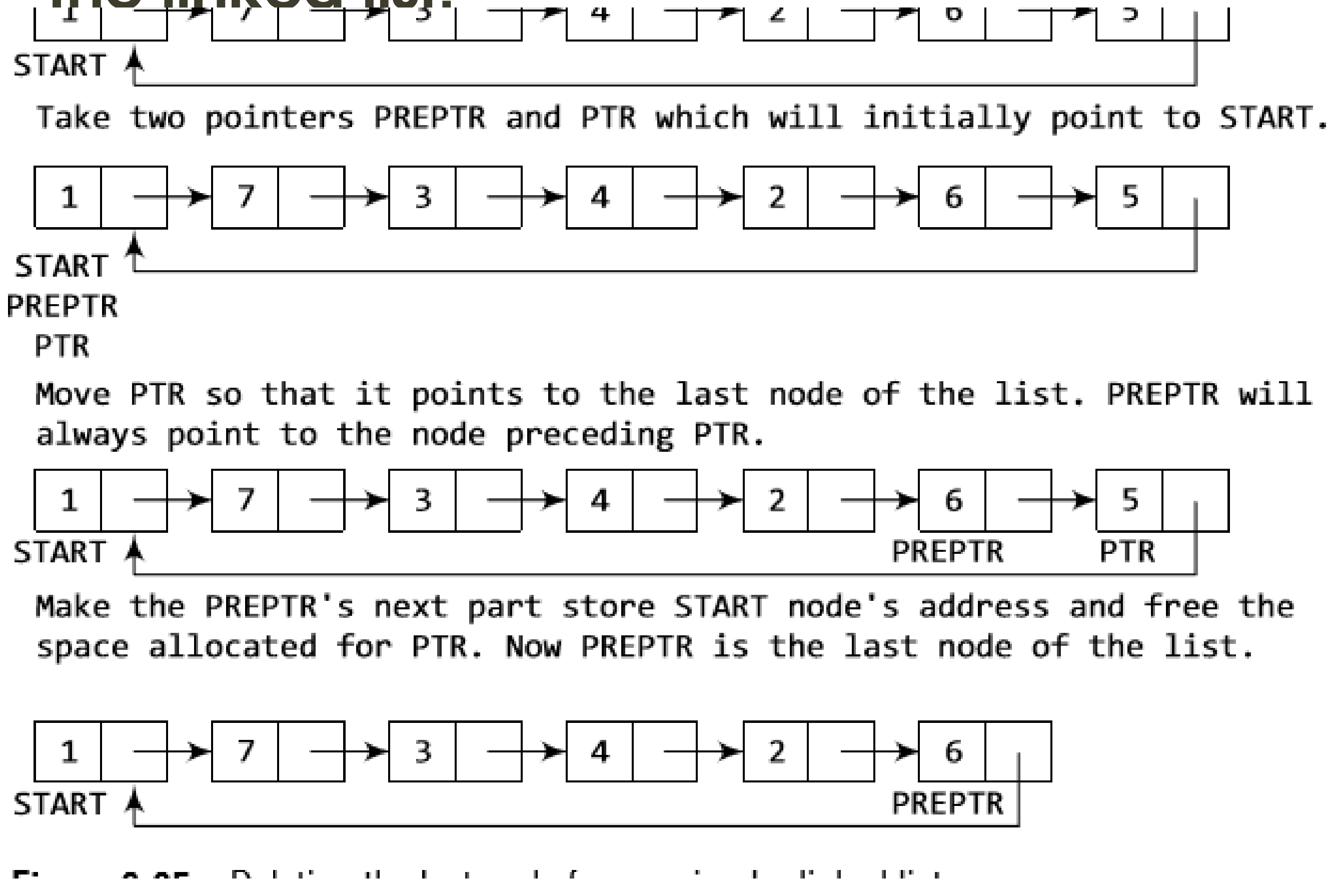


```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:         SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START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

Figure 6.34 Algorithm to delete the first node

Circular Linked Lists

- Deleting the Last Node from a Circular Linked List
- Consider the circular linked list shown in Fig. 6.35.
- Suppose we want to delete the last node from the linked list, then the following changes will be done in the 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.

Circular Linked Lists

- Deleting the Last Node from a Circular Linked List
- Figure 6.36 shows the algorithm to delete the last node from a circular linked list.
- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we take another pointer variable PREPTR such that PREPTR always points to one node before PTR.
- Once we reach the last node and the second last node, we set the next pointer of the second last node to START, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned to the free pool.

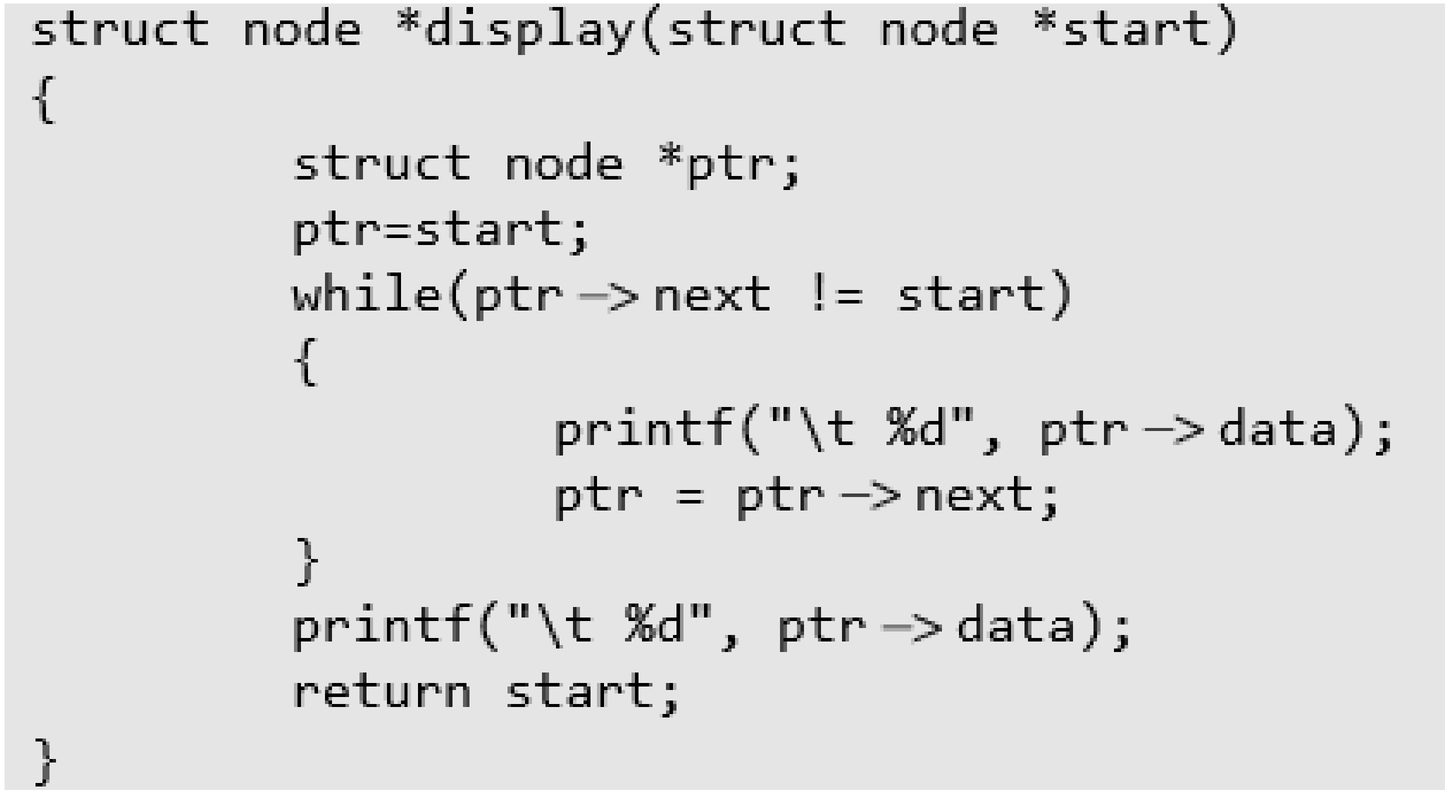
Circular Linked Lists

- **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
```

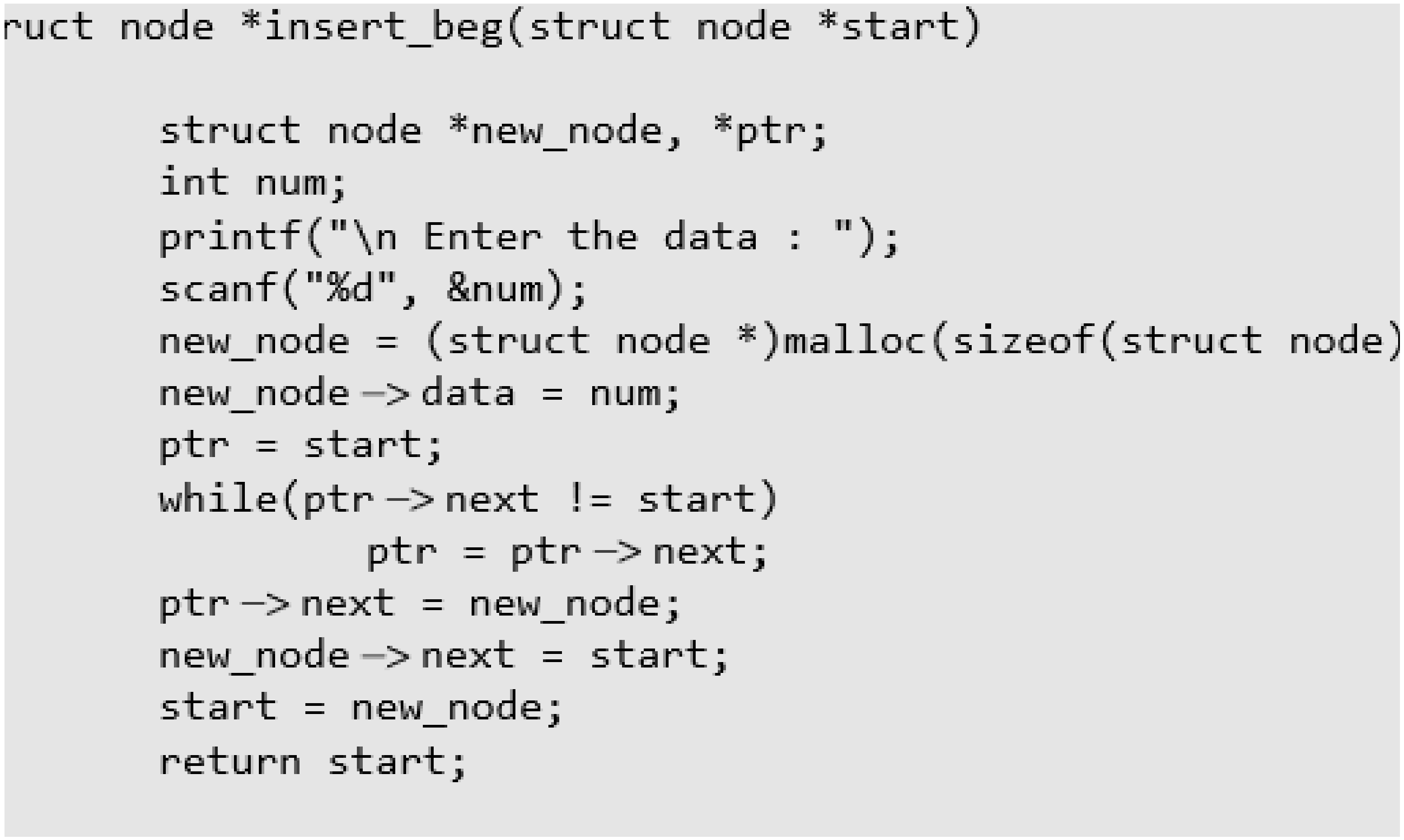
Figure 6.36 Algorithm to delete the last node

Circular Linked Lists



```
struct node *display(struct node *start)
{
        struct node *ptr;
        ptr=start;
        while(ptr -> next != start)
        {
                printf("\t %d", ptr -> data);
                ptr = ptr -> next;
        }
        printf("\t %d", ptr -> data);
        return start;
}
```

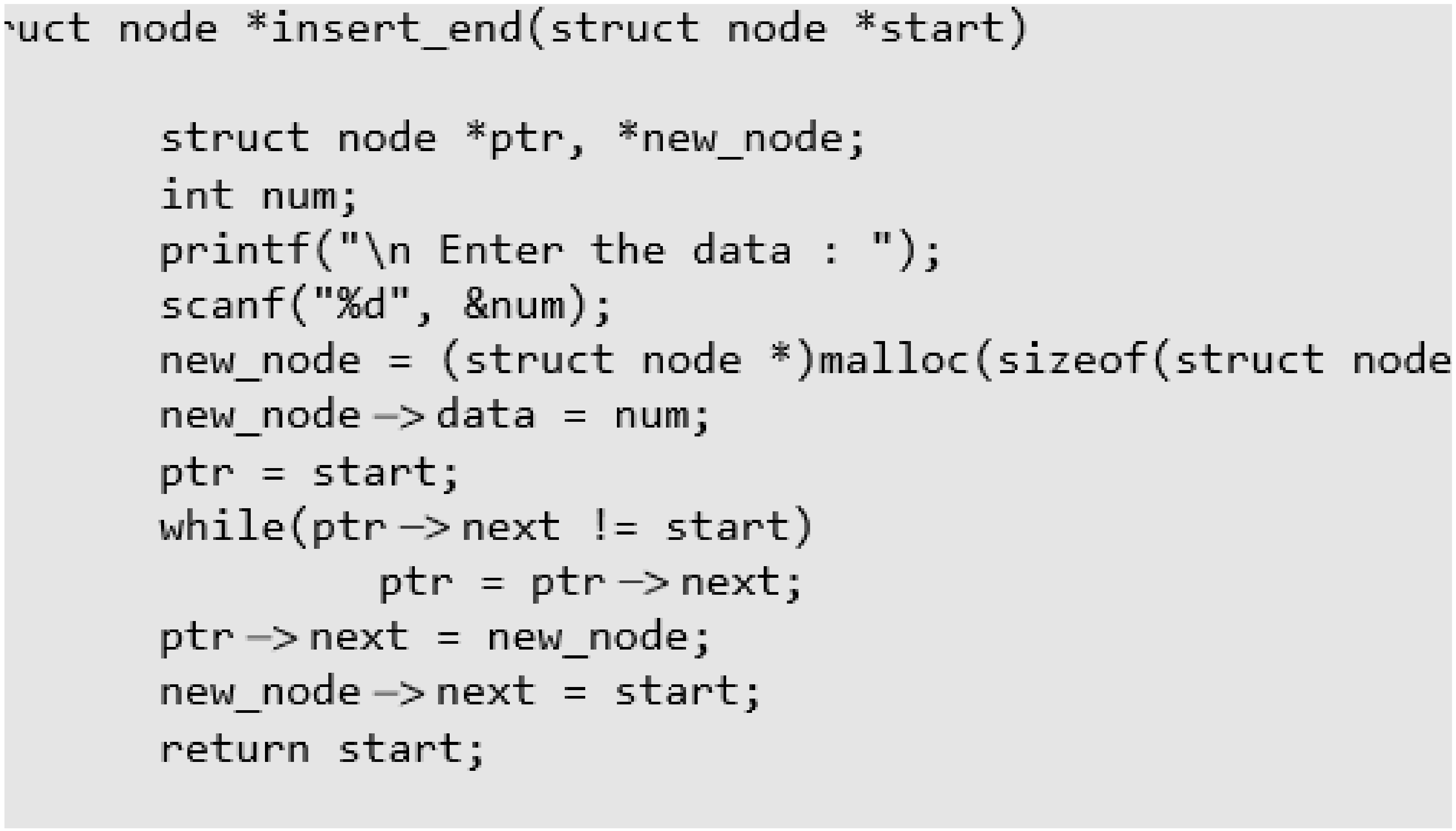
Circular Linked Lists



```
ruct node *insert_beg(struct node *start)

        struct node *new_node, *ptr;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node)
        new_node -> data = num;
        ptr = start;
        while(ptr -> next != start)
                  ptr = ptr -> next;
        ptr -> next = new_node;
        new_node -> next = start;
        start = new_node;
        return start;
```

Circular Linked Lists



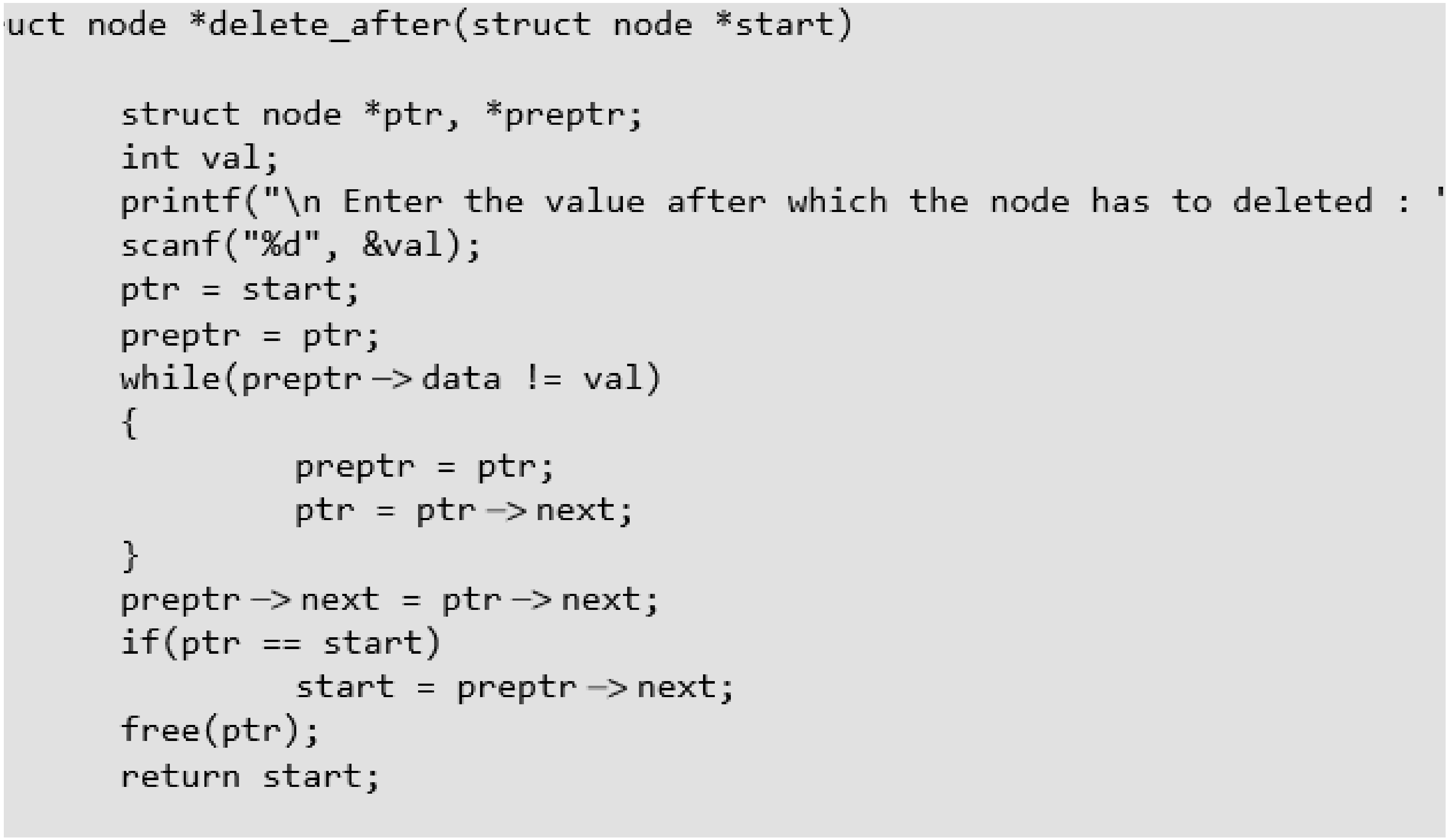
```
ruct node *insert_end(struct node *start)

      struct node *ptr, *new_node;
      int num;
      printf("\n Enter the data : ");
      scanf("%d", &num);
      new_node = (struct node *)malloc(sizeof(struct node
      new_node->data = num;
      ptr = start;
      while(ptr->next != start)
              ptr = ptr->next;
      ptr->next = new_node;
      new_node->next = start;
      return start;
```

Circular Linked Lists

```
struct node *delete_end(struct node *start)
{
        struct node *ptr, *preptr;
        ptr = start;
        while(ptr -> next != start)
        {
                preptr = ptr;
                ptr = ptr -> next;
        }
        preptr -> next = ptr -> next;
        free(ptr);
        return start;
}
```

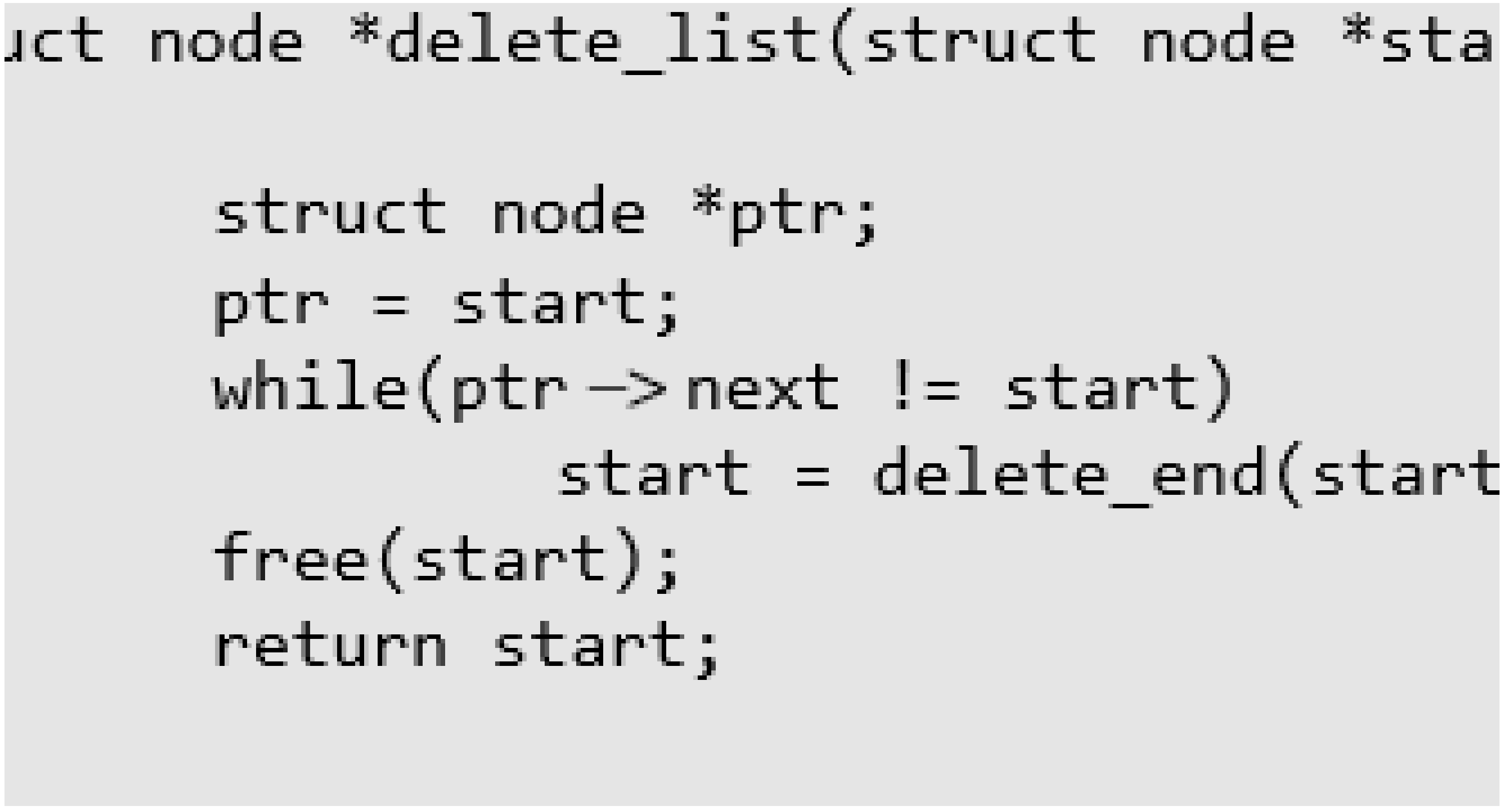
Circular Linked Lists



```
uct node *delete_after(struct node *start)

      struct node *ptr, *preptr;
      int val;
      printf("\n Enter the value after which the node has to deleted : '
      scanf("%d", &val);
      ptr = start;
      preptr = ptr;
      while(preptr -> data != val)
      {
              preptr = ptr;
              ptr = ptr -> next;
      }
      preptr -> next = ptr -> next;
      if(ptr == start)
              start = preptr -> next;
      free(ptr);
      return start;
```

Circular Linked Lists



```
uct node *delete_list(struct node *sta

     struct node *ptr;
     ptr = start;
     while(ptr -> next != start)
             start = delete_end(start
     free(start);
     return start;
```

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.

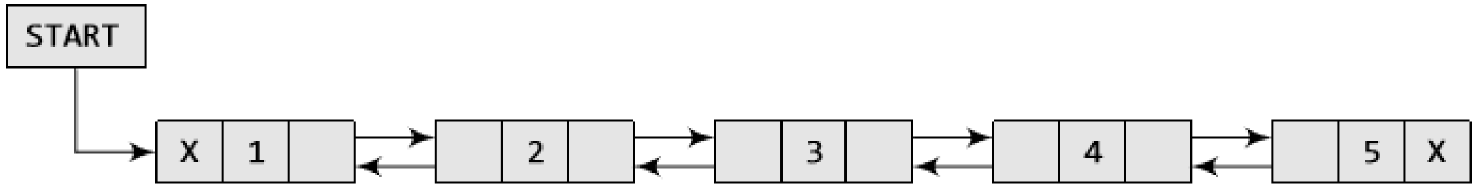
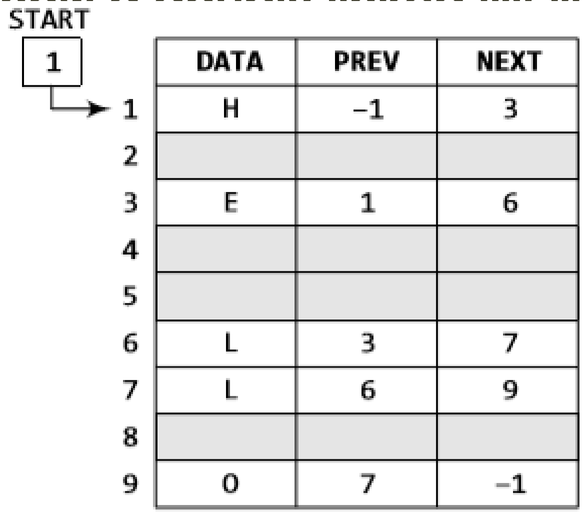


Figure 6.37 Doubly linked list

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

Doubly Linked Lists

- The PREV field of the first node and the NEXT field of the last node will contain NULL.
- The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.
- Thus, we see that a doubly linked list calls for more space per node and more expensive basic operations.
- However, a 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 doubly linked list is that it makes searching twice as efficient.
- Let us view how a doubly linked list is maintained in the memory.



START 1

	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

Doubly Linked Lists

- In the figure, we see that 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

- Inserting a new node in a doubly Linked List
- In this section, we will discuss how a new node is added into an already existing 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.

Doubly Linked Lists

- Inserting a Node at the Beginning of a Doubly Linked List
- Consider the doubly linked list shown in Fig. 6.39.
- Suppose we want to add a new node with data 9 as the first node of the list.
- Then the following changes will be done in the linked list.

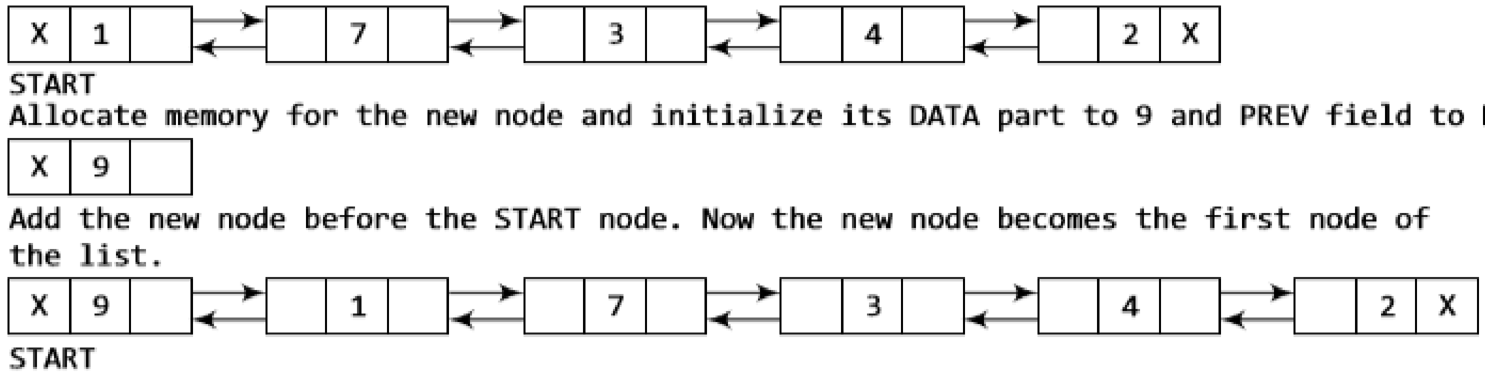
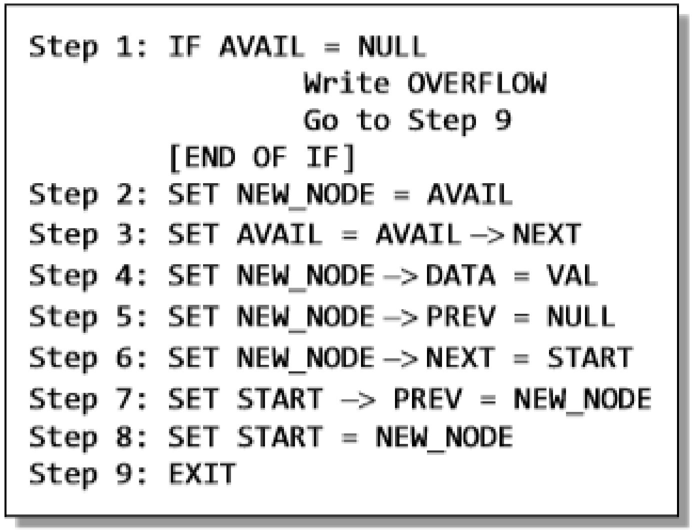

X 1 7 3 4 2 X
START
Allocate memory for the new node and initialize its DATA part to 9 and PREV field to NULL.
X 9
Add the new node before the START node. Now the new node becomes the first node of the list.
X 9 1 7 3 4 2 X
START

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

Doubly Linked Lists

- Inserting a Node at the Beginning of a Doubly Linked List
- Figure 6.40 shows the algorithm to insert a new node at the beginning of a doubly linked list.
- In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if free memory cell is available, then we allocate space for the new node.
- Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.

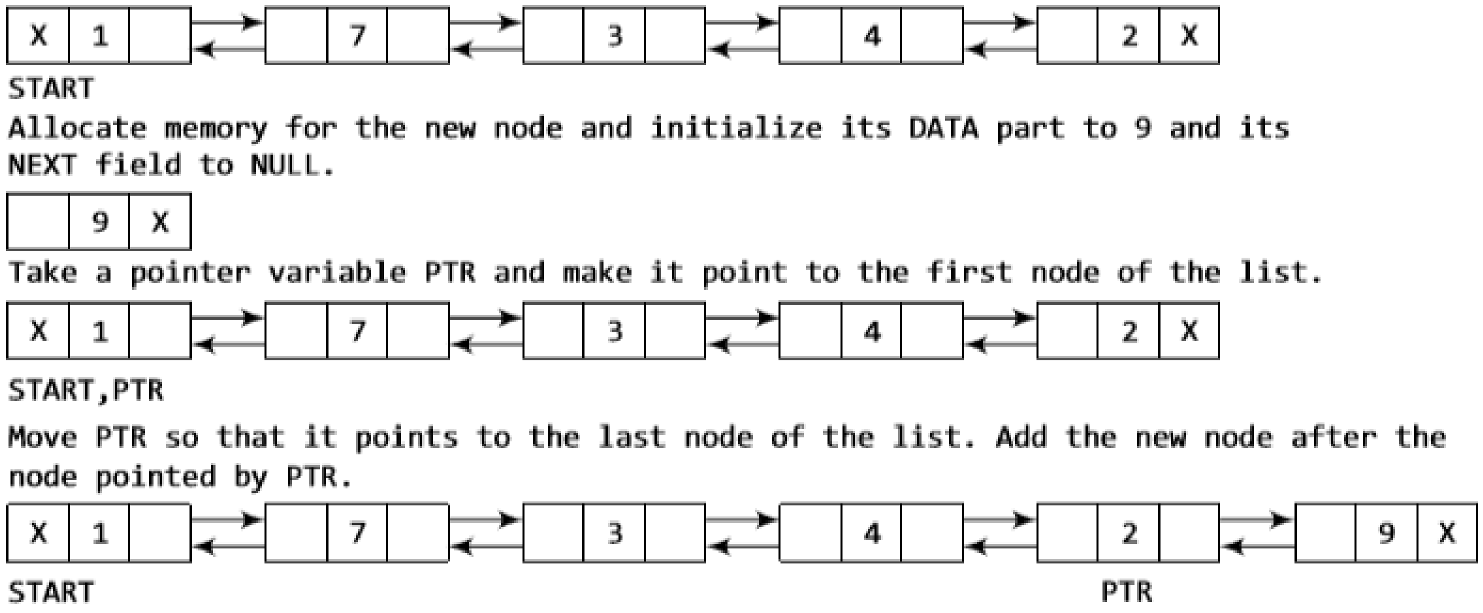


```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV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

Figure 6.40 Algorithm to insert a new node at the beginning

Doubly Linked Lists

- Inserting a Node at the End end of a Doubly Linked List
- Consider the doubly linked list shown in Fig. 6.41.
- Suppose we want to add a new node with data 9 as the last node of the list.
- Then the following changes will be done in the linked list.



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

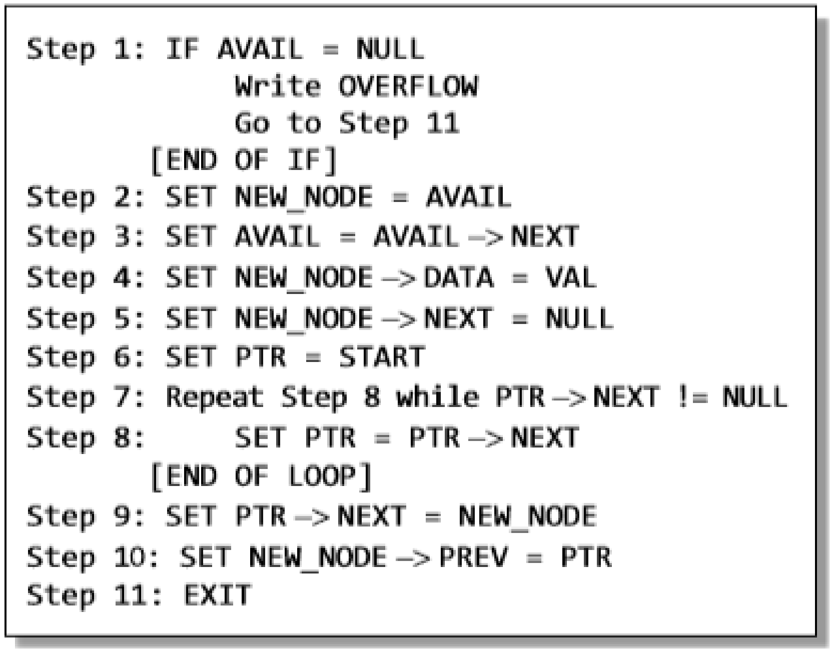
Take a pointer variable PTR and make it point to the first node of the list.

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

Doubly Linked Lists

- Inserting a Node at the End end of a Doubly Linked List
- Figure 6.42 shows the algorithm to insert a new node at the end of a doubly linked list.
- In Step 6, we take a pointer variable PTR and initialize it with START. In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node.
- Remember that the NEXT field of the new node contains NULL which signifies the end of the linked list.
- The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list)

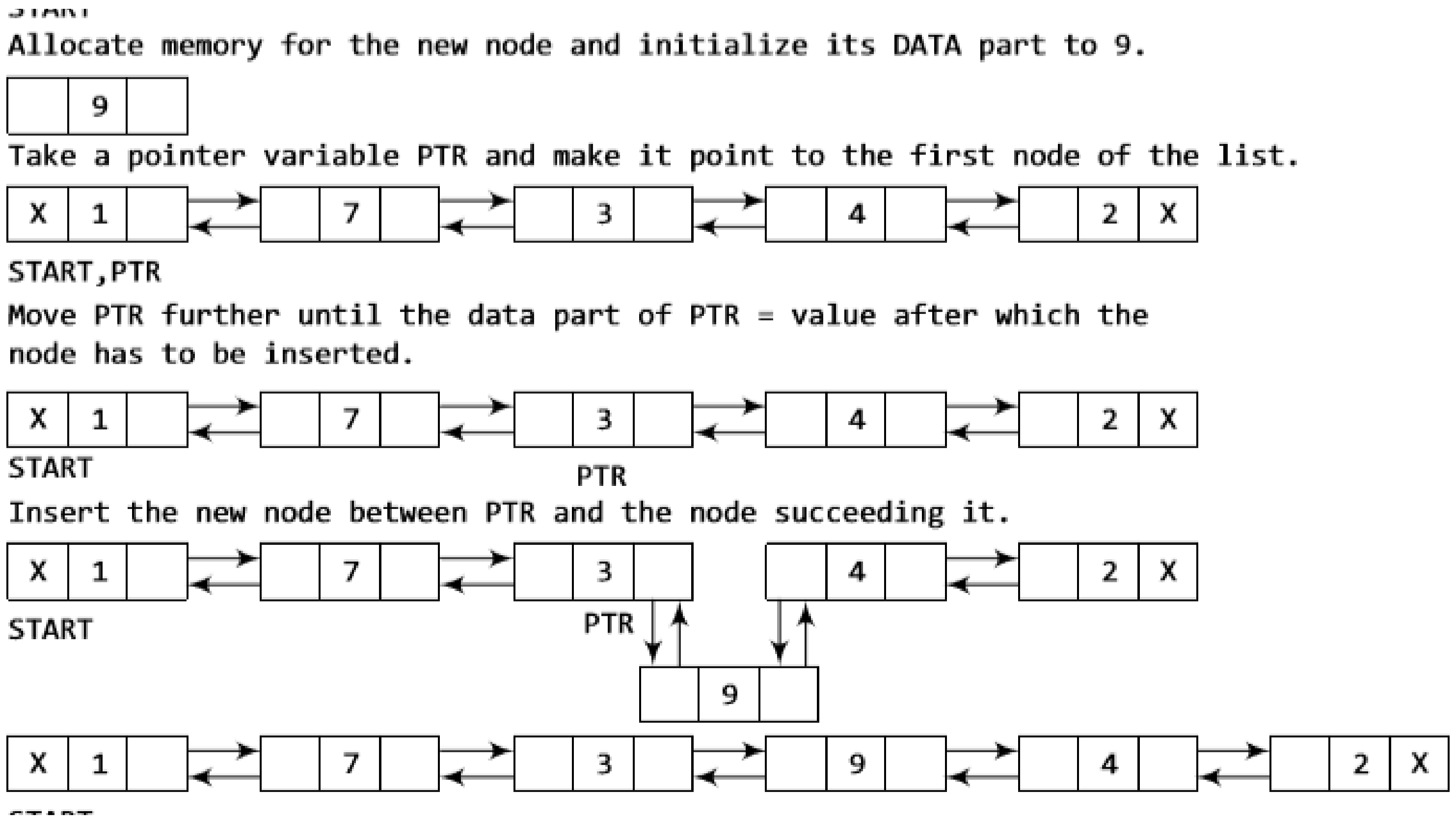


```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT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

Doubly Linked Lists

- Inserting a Node After a Given Node in a Doubly Linked List
- Consider the doubly linked list shown in Fig. 6.44.
- Suppose we want to add a new node with value 9 after the node containing 3.
- Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 6.43.


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

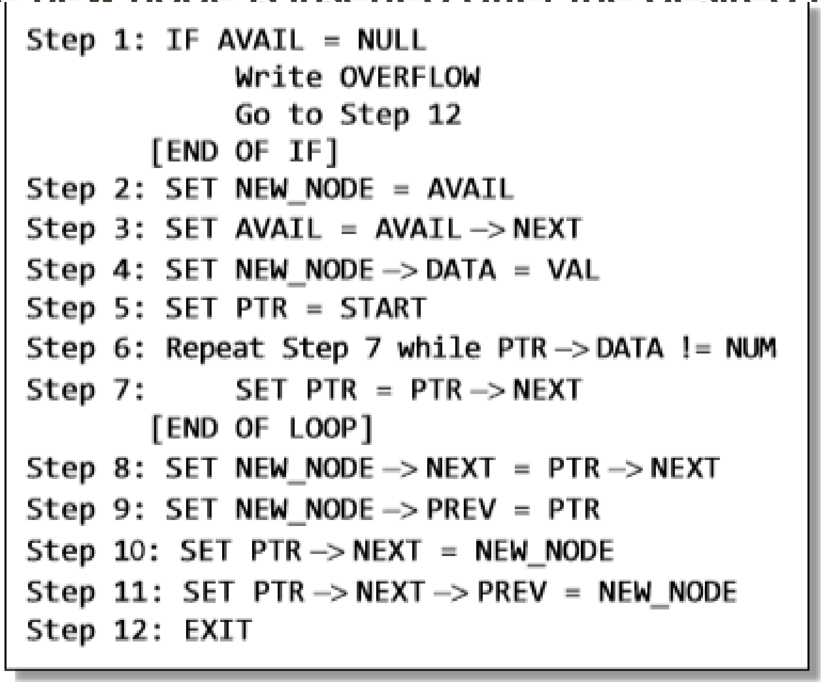
Take a pointer variable PTR and make it point to the first node of the list.

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.

Doubly Linked Lists

- Inserting a Node After a Given Node in a Doubly Linked List
- Figure 6.43 shows the algorithm to insert a new node after a given node in a doubly linked list.
- In Step 5, we take a pointer PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted after this node.
- Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted after the desired 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: 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

Doubly Linked Lists

- Inserting a Node Before a Given Node in a Doubly Linked List
- Consider the doubly linked list shown in Fig. 6.46.
- Suppose we want to add a new node with value 9 before the node containing 3.
- Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 6.45.
- In Step 1, we first check whether memory is available for the new node.
- In Step 5, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted before this node.
- Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted before the desired node.

Doubly Linked Lists

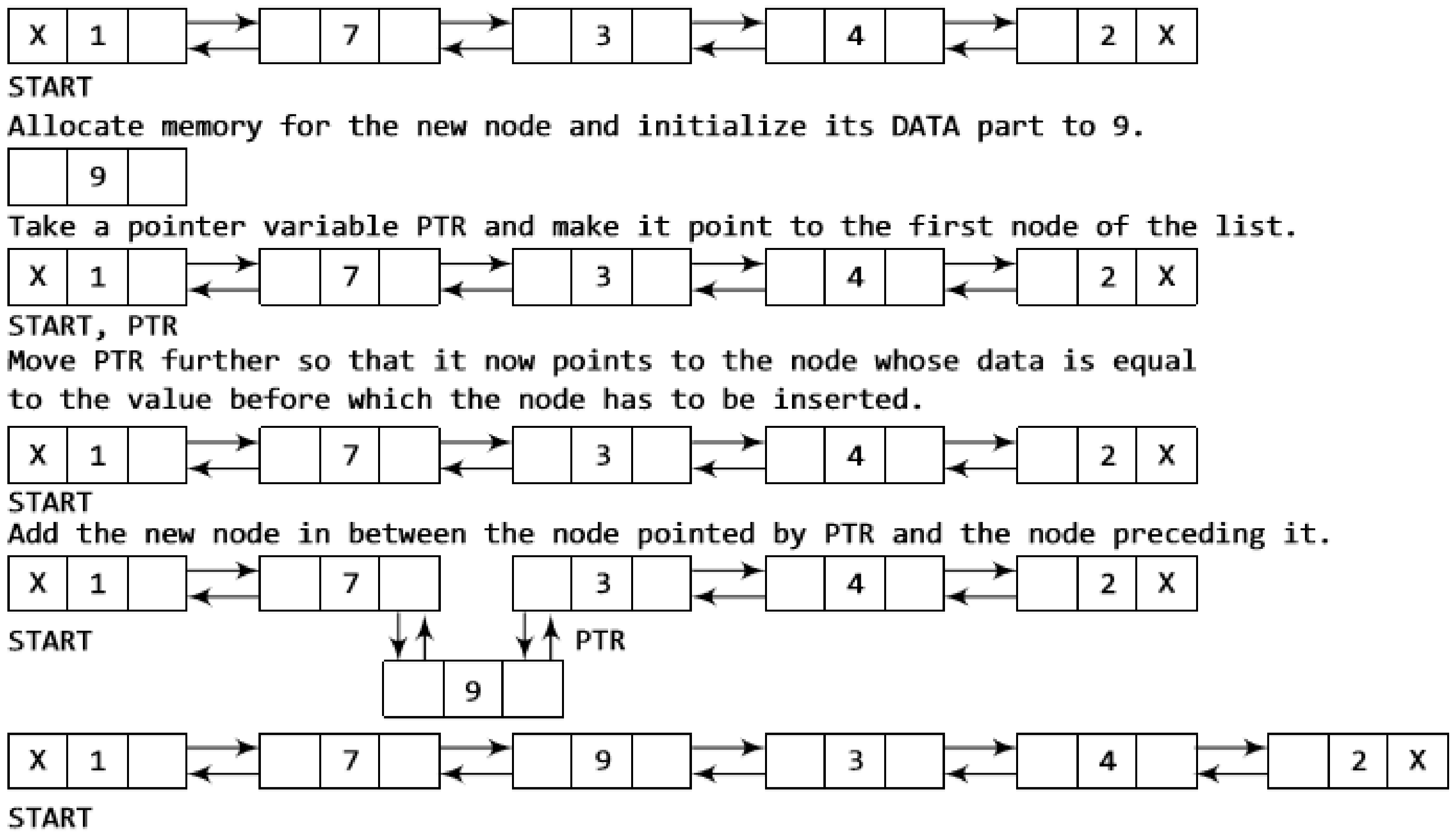
- **Inserting a Node Before a Given Node in a Doubly Linked List**

```
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

Doubly Linked Lists

- **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

Doubly Linked Lists

- Deleting a node from a doubly Linked List
- In this section, we will see how a node is deleted from an already existing 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.

Doubly Linked Lists

- Deleting the First Node from a Doubly Linked List
- Consider the doubly linked list shown in Fig. 6.47.
- When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

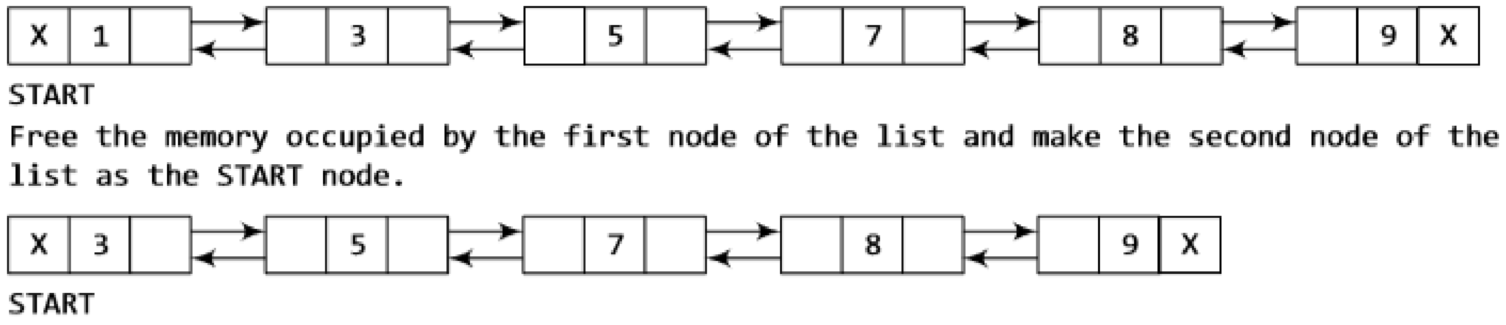
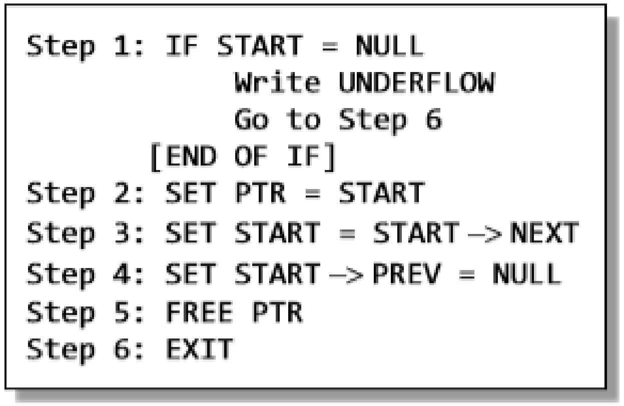

X 1 3 5 7 8 9 X
START
Free the memory occupied by the first node of the list and make the second node of the list as the START node.
X 3 5 7 8 9 X
START

Figure 6.47 Deleting the first node from a doubly linked list

Doubly Linked Lists

- Deleting the First Node from a Doubly Linked
- ListFigure 6.48 shows the algorithm to delete the first node of a doubly linked list.
- In Step 1 of the algorithm, we check if the linked list exists or not.
- If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list.
- For this, we initialize PTR with START that stores the address of the first node of the list.
- In Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

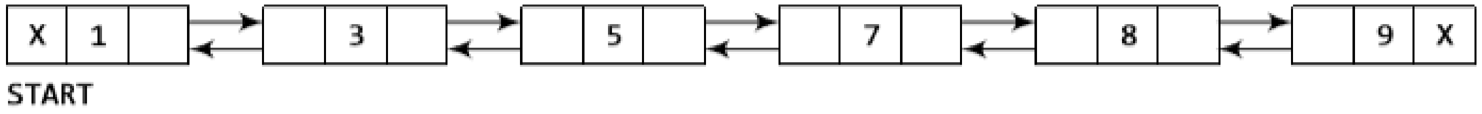


```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 6
           [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

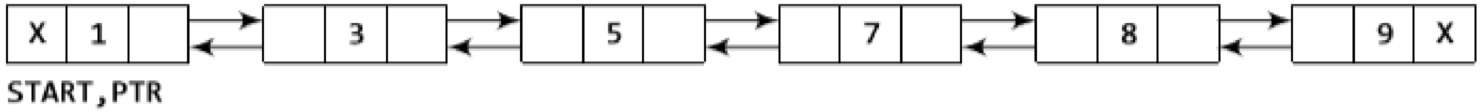
Figure 6.48 Algorithm to delete the first node

Doubly Linked Lists

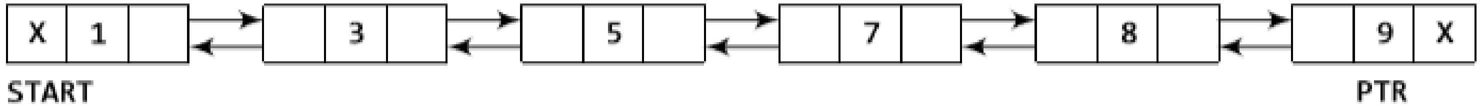
- **Deleting the Last Node from a Doubly Linked List**
- **Consider the doubly linked list shown in Fig. 6.49.**
- **Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.**



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.

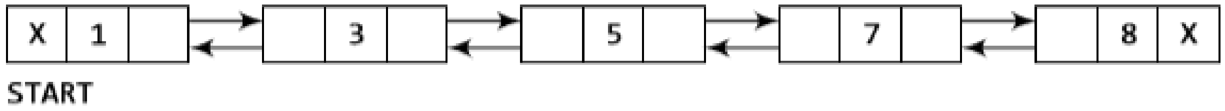
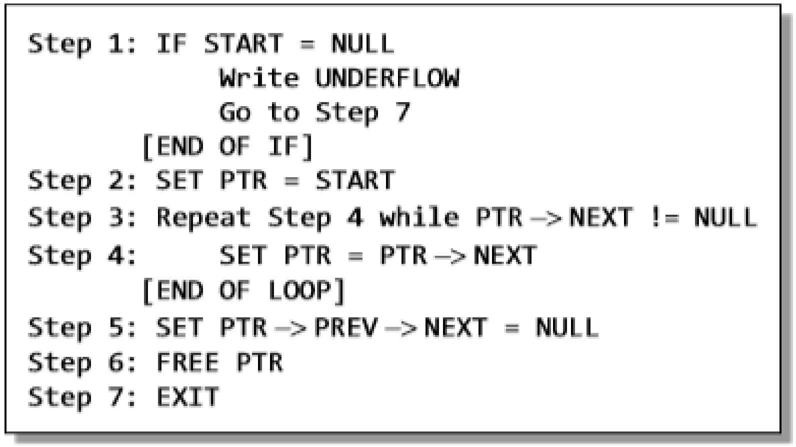


Figure 6.49 Deleting the last node from a doubly linked list

Doubly Linked Lists

- Deleting the Last Node from a Doubly Linked List
- Figure 6.50 shows the algorithm to delete the last node of a doubly linked list.
- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- The while loop traverses through the list to reach the last node.
- Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node.
- To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned to the free pool.



```
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

Doubly Linked Lists

- Deleting the Node After a Given Node in a Doubly Linked List
- Consider the doubly linked list shown in Fig. 6.51.
- Suppose we want to delete the node that succeeds the node which contains data value 4.
- Then the following changes will be done in the linked list.

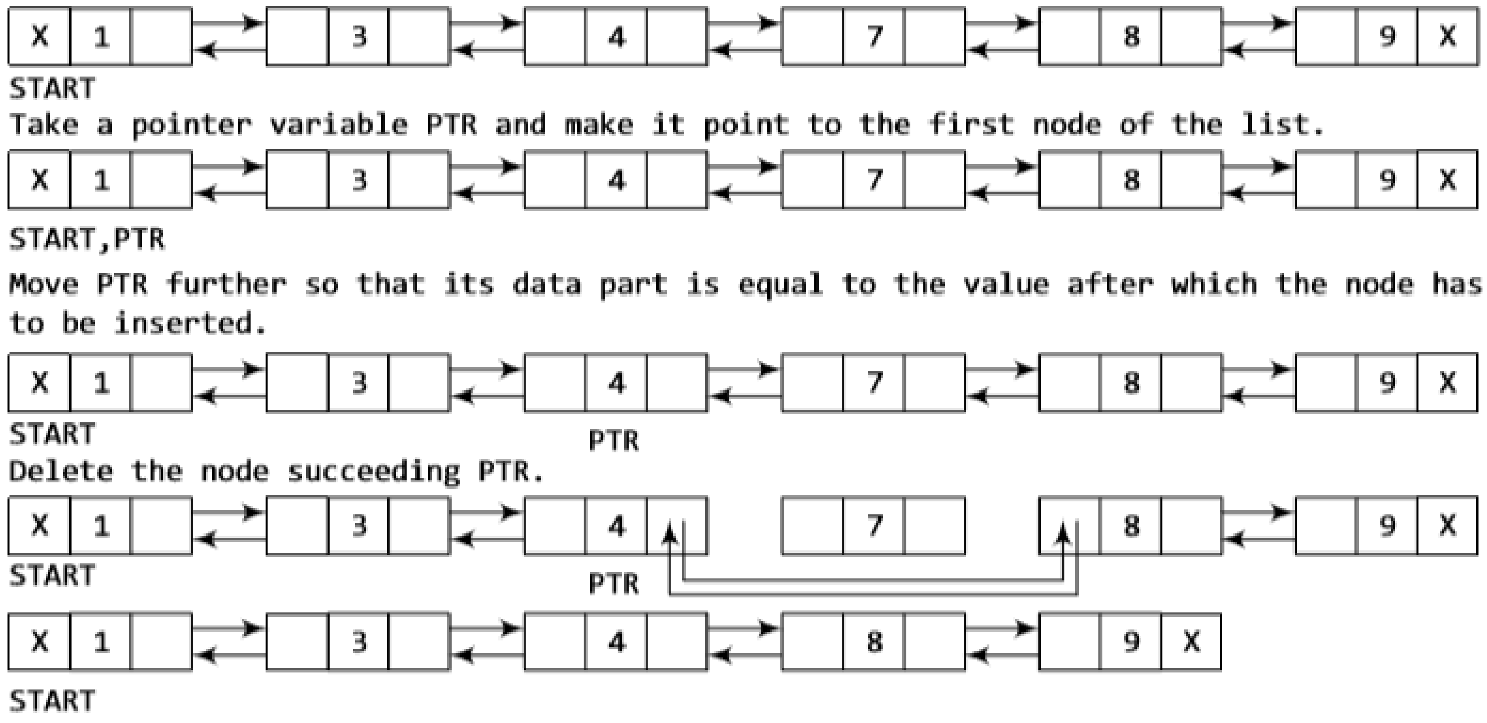
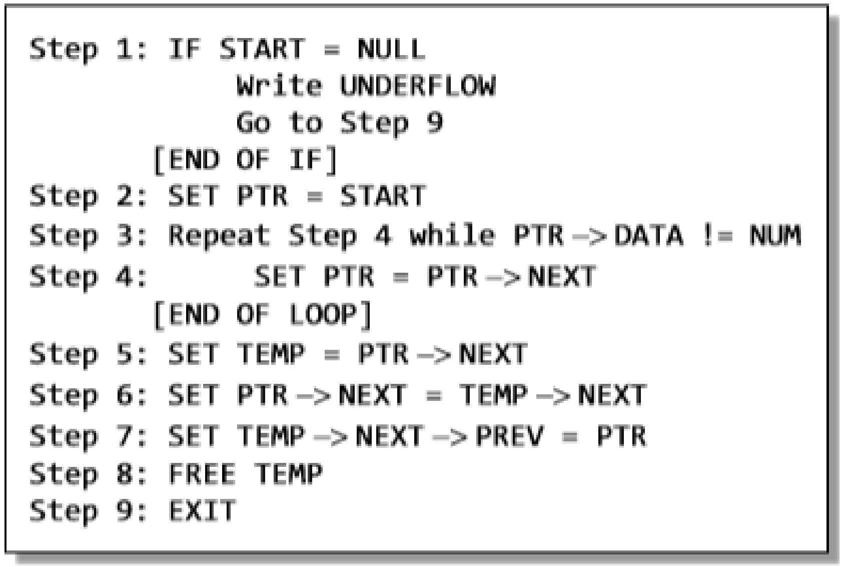

X 1 3 4 7 8 9 X
START
Take a pointer variable PTR and make it point to the first node of the list.
START,PTR
Move PTR further so that its data part is equal to the value after which the node has to be inserted.
START
PTR
Delete the node succeeding PTR.
START
PTR
X 1 3 4 8 9 X
START

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

Doubly Linked Lists

- Deleting the Node After a Given Node in a Doubly Linked List
- Figure 6.52 shows the algorithm to delete a node after a given node of a doubly linked list.
- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the doubly linked list.
- The while loop traverses through the linked list to reach the given node.
- Once we reach the node containing VAL, the node succeeding it can be easily accessed by using the address stored in its NEXT field.
- The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node.
- Finally, the memory of the node succeeding the given node is freed and returned to th



```
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

Doubly Linked Lists

- Deleting the Node Before a Given Node in a Doubly Linked List
- Consider the doubly linked list shown in Fig. 6.53.
- Suppose we want to delete the node preceding the node with value 4.
- Before discussing the changes that will be done in the linked list, let us first look at the algorithm.

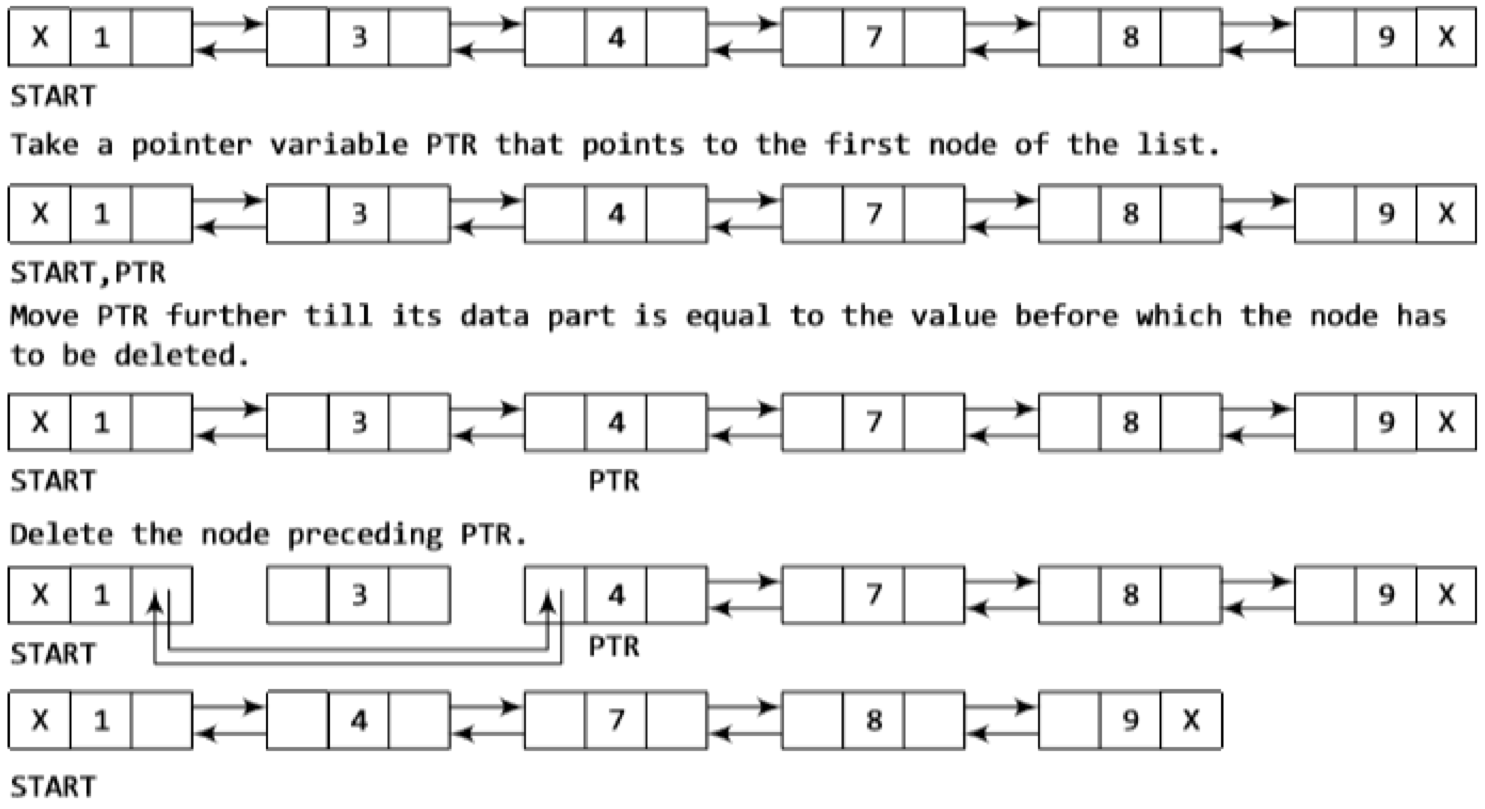
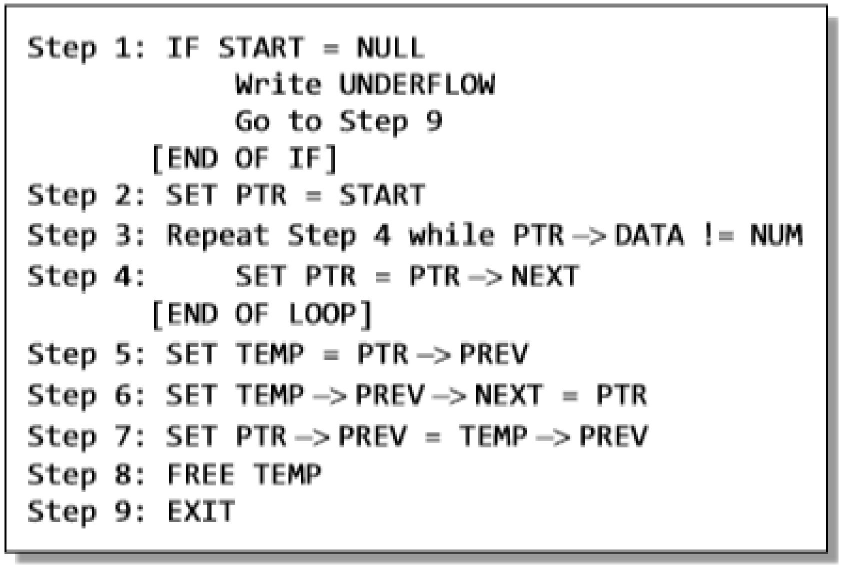

X 1 3 4 7 8 9 X
START
Take a pointer variable PTR that points to the first node of the list.
X 1 3 4 7 8 9 X
START,PTR
Move PTR further till its data part is equal to the value before which the node has to be deleted.
X 1 3 4 7 8 9 X
START
PTR
Delete the node preceding PTR.
X 1 3 4 7 8 9 X
START
PTR
X 1 4 7 8 9 X
START

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

Doubly Linked Lists

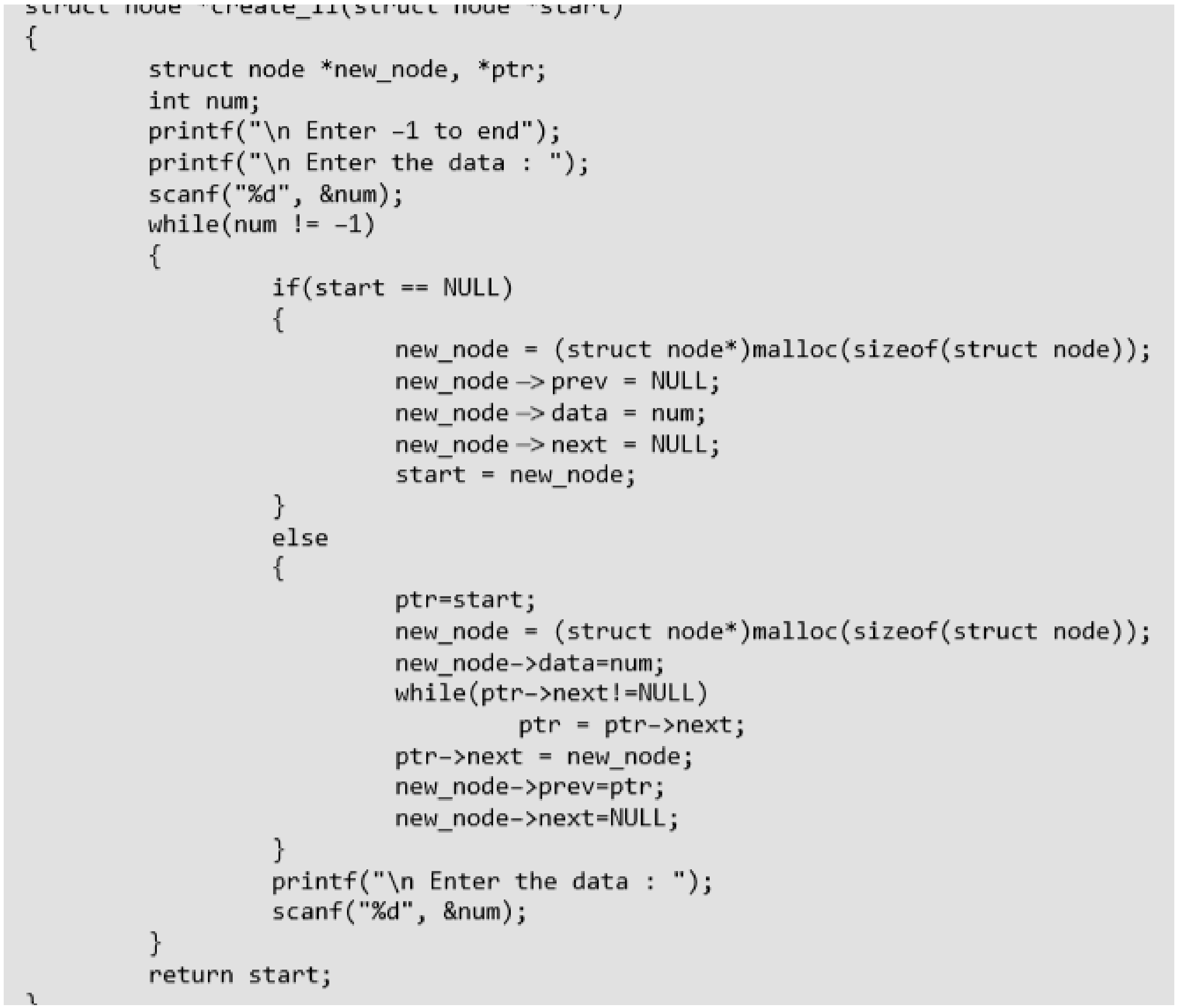
- Deleting the Node Before a Given Node in a Doubly Linked List
- Figure 6.54 shows the algorithm to delete a node before a given node of a doubly linked list.
- In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.
- The while loop traverses through the linked list to reach the desired node.
- Once we reach the node containing VAL, the PREV field of PTR is set to contain the address of the node preceding the node which comes before PTR.
- The memory of the node preceding PTR is freed and returned to the free pool. Hence, we see that we can insert or delete a node in a constant number of operations given only that node's address.
- Note that this is not possible in the case of a singly linked list which requires the previous node's address also to perform the same operation.



```
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

Doubly Linked Lists



```
struct node *create_ll(struct node *start)
{
        struct node *new_node, *ptr;
        int num;
        printf("\n Enter -1 to end");
        printf("\n Enter the data : ");
        scanf("%d", &num);
        while(num != -1)
        {
                if(start == NULL)
                {
                        new_node = (struct node*)malloc(sizeof(struct node));
                        new_node -> prev = NULL;
                        new_node -> data = num;
                        new_node -> next = NULL;
                        start = new_node;
                }
                else
                {
                        ptr=start;
                        new_node = (struct node*)malloc(sizeof(struct node));
                        new_node->data=num;
                        while(ptr->next!=NULL)
                                ptr = ptr->next;
                        ptr->next = new_node;
                        new_node->prev=ptr;
                        new_node->next=NULL;
                }
                printf("\n Enter the data : ");
                scanf("%d", &num);
        }
        return start;
}
```

Doubly Linked Lists

```
struct node *display(struct node *start)
{
        struct node *ptr;
        ptr=start;
        while(ptr!=NULL)
        {
                printf("\t %d", ptr -> data);
                ptr = ptr -> next;
        }
        return start;
}
```

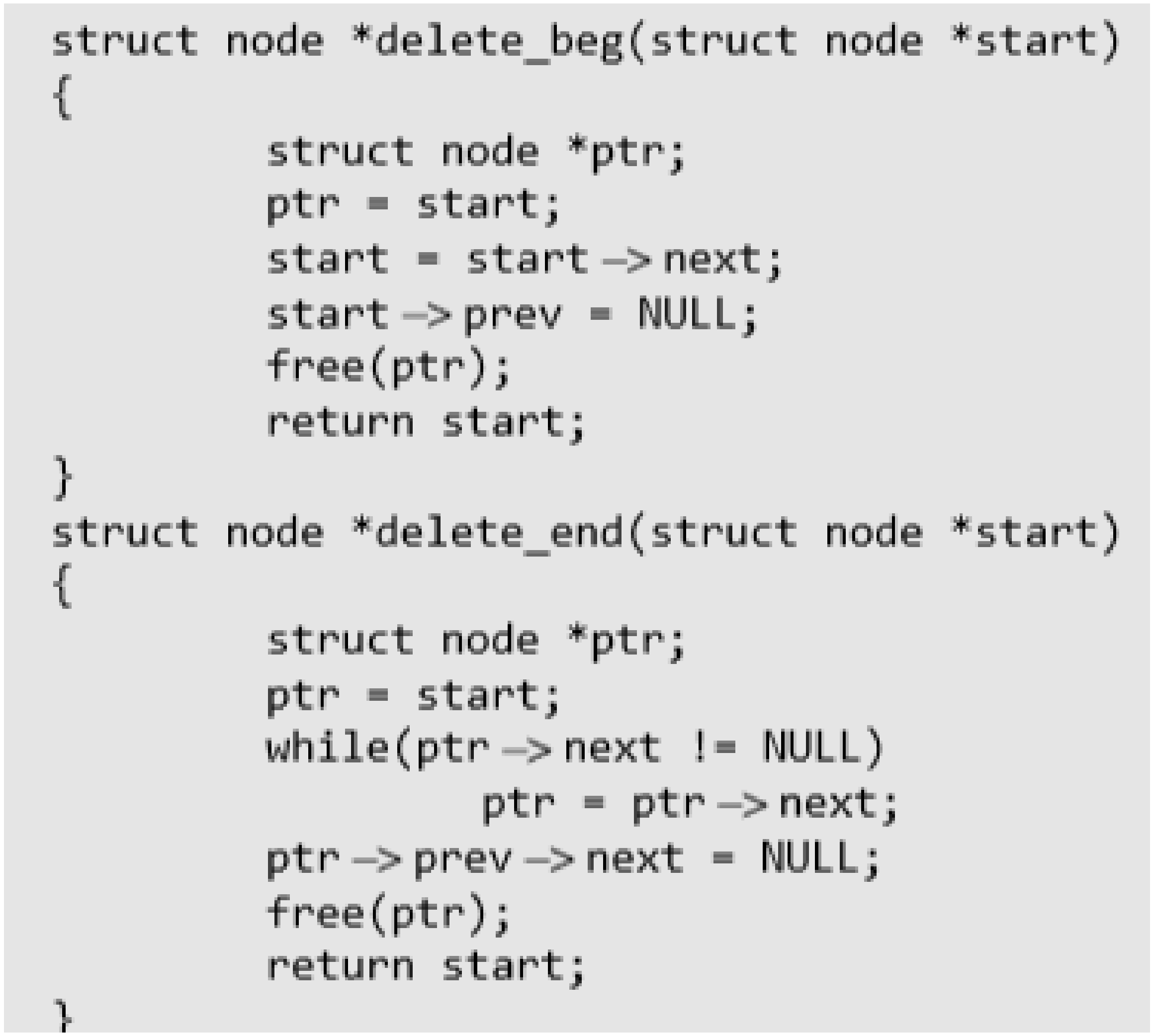
Doubly Linked Lists

```
struct node *insert_end(struct node *start)
{
        struct node *ptr, *new_node;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        ptr=start;
        while(ptr -> next != NULL)
                ptr = ptr -> next;
        ptr -> next = new_node;
        new_node -> prev = ptr;
        new_node -> next = NULL;
        return start;
}
```

Doubly Linked Lists

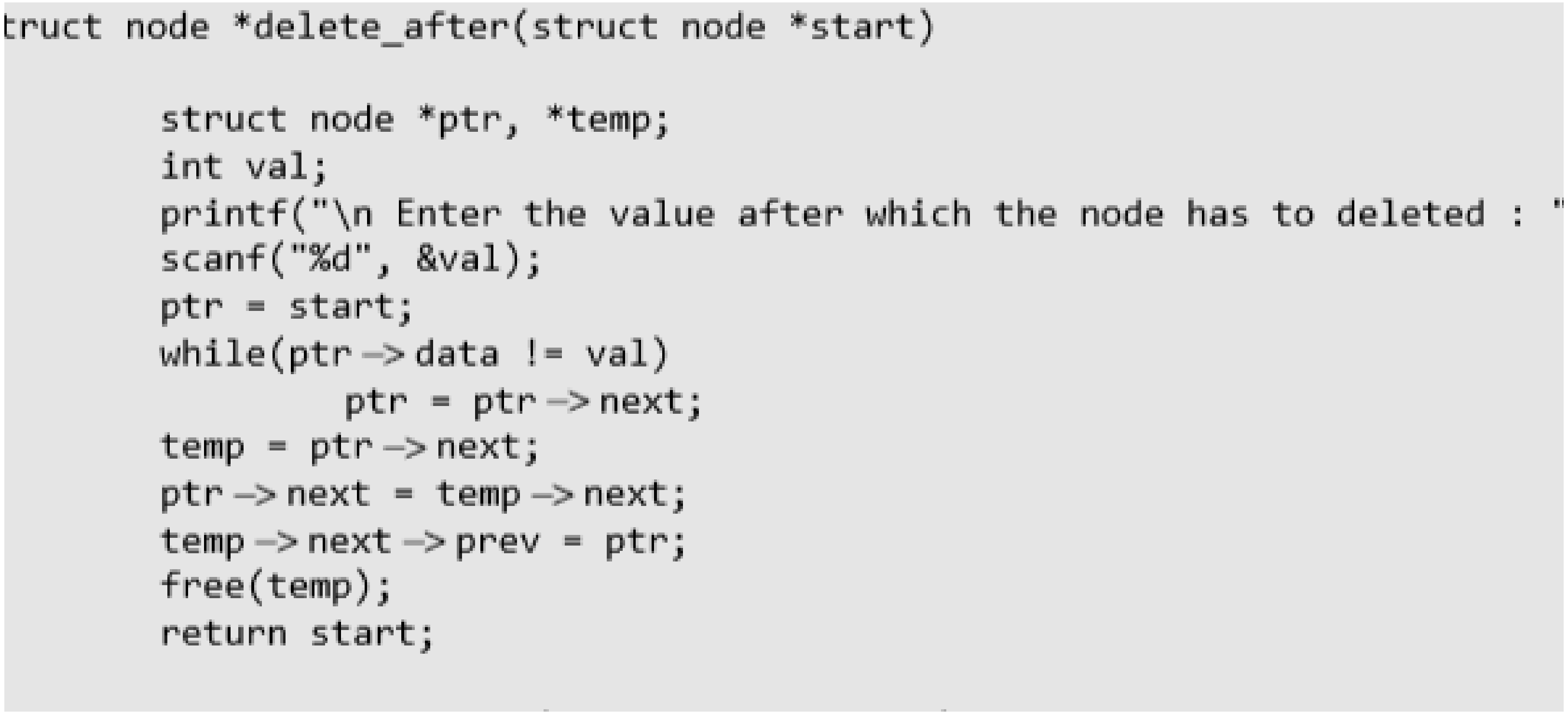
```
struct node *insert_before(struct node *start)
{
        struct node *new_node, *ptr;
        int num, val;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        printf("\n Enter the value before which the data has to be inserted:");
        scanf("%d", &val);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        ptr = start;
        while(ptr -> data != val)
                ptr = ptr -> next;
        new_node -> next = ptr;
        new_node -> prev = ptr-> prev;
        ptr -> prev -> next = new_node;
        ptr -> prev = new_node;
        return start;
}
```

Doubly Linked Lists



```
struct node *delete_beg(struct node *start)
{
         struct node *ptr;
         ptr = start;
         start = start -> next;
         start -> prev = NULL;
         free(ptr);
         return start;
}
struct node *delete_end(struct node *start)
{
         struct node *ptr;
         ptr = start;
         while(ptr -> next != NULL)
                  ptr = ptr -> next;
         ptr -> prev -> next = NULL;
         free(ptr);
         return start;
}
```

Doubly Linked Lists



```
truct node *delete_after(struct node *start)

       struct node *ptr, *temp;
       int val;
       printf("\n Enter the value after which the node has to deleted : "
       scanf("%d", &val);
       ptr = start;
       while(ptr -> data != val)
               ptr = ptr -> next;
       temp = ptr -> next;
       ptr -> next = temp -> next;
       temp -> next -> prev = ptr;
       free(temp);
       return start;
```

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.

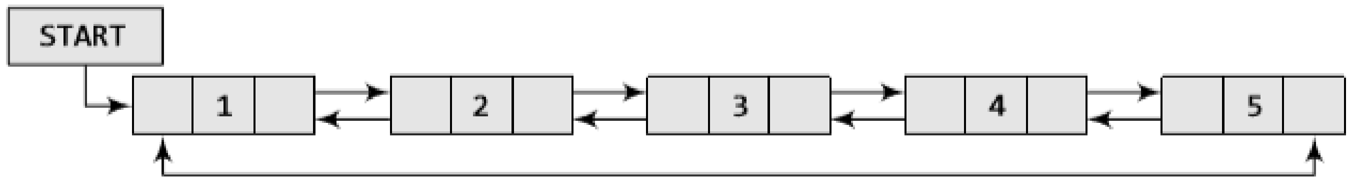
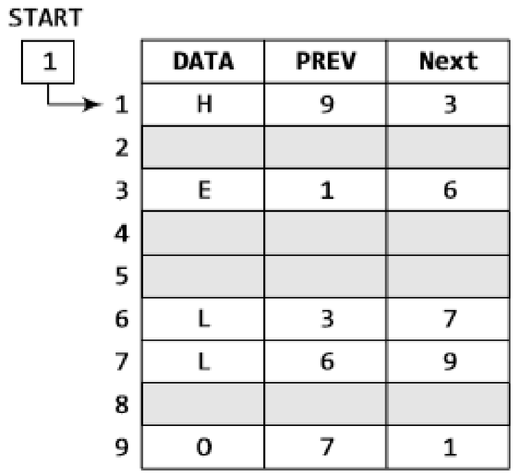


Figure 6.55 Circular doubly linked list

Circular Doubly Linked Lists

- 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.
- Let us view how a circular doubly linked list is maintained in the memory. Consider Fig. 6.56



	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

Circular Doubly Linked Lists

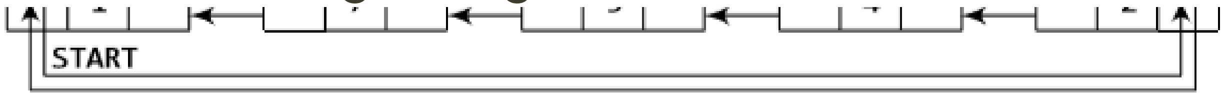
- In the figure, we see that a variable START is used to store the address of the first node.
- Here in this example, START = 1, so the first data is stored at address 1, which is H.
- Since this is the first node, it stores the address of the last node of the list in its previous field.
- The corresponding 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 Lists

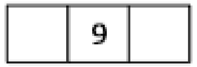
- Inserting a new node in a circular doubly Linked List
- In this section, we will see how a new node is added into an already existing circular doubly linked list.
- We will take two cases and then see how insertion is done in each case.
- Rest of the cases are similar to that given for doubly linked lists.
- Case 1: The new node is inserted at the beginning.
- Case 2: The new node is inserted at the end.

Circular Doubly Linked Lists

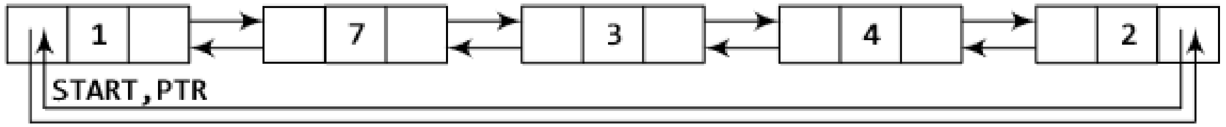
- Inserting a Node at the Beginning of a Circular Doubly Linked List
- Consider the circular doubly linked list shown in Fig. 6.57.
- Suppose we want to add a new node with data 9 as the first node of the list.
- Then, the following changes will be done in the linked list.



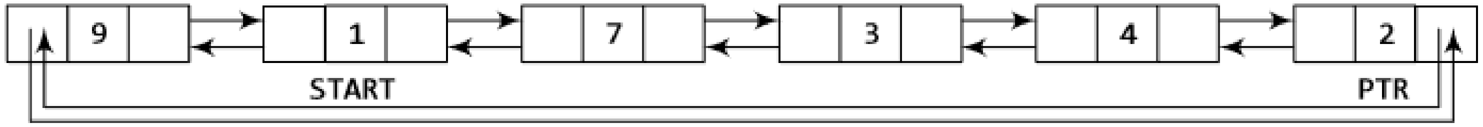
Allocate memory for the new node and initialize its DATA part to 9.



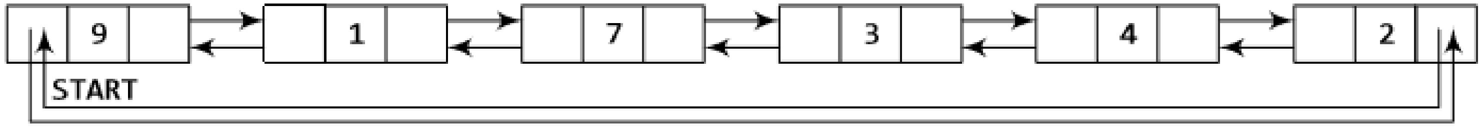
Take a pointer variable PTR that points to the first node of the list.



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.



Circular Doubly Linked Lists

- Figure 6.58 shows the algorithm to insert a new node at the beginning of a circular doubly linked list.
- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, we allocate space for the new node. Set its data part with the given VAL and its next part is initialized with the address of the first node of the list, which is stored in START.
- Now since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.
- Since it is a circular doubly linked list, the PREV field of the NEW_NODE is set to contain the address of the last node.

Circular Doubly Linked Lists

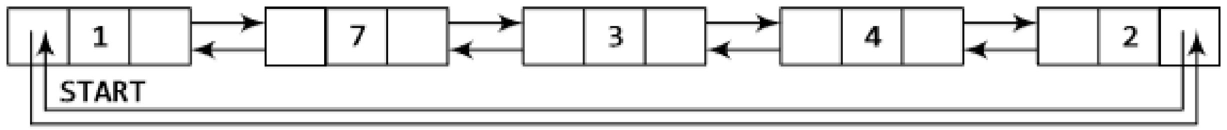
- Inserting a Node at the Beginning of a Circular Doubly Linked List

```
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
```

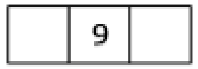
Figure 6.58 Algorithm to insert a new node at the beginning

Circular Doubly Linked Lists

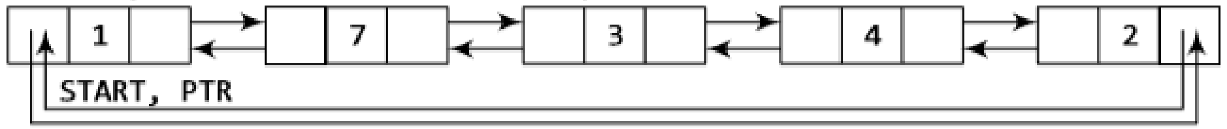
- Inserting a Node at the End of a Circular Doubly Linked List
- Consider the circular doubly linked list shown in Fig. 6.59.
- Suppose we want to add a new node with data 9 as the last node of the list.
- Then the following changes will be done in the linked list.



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.

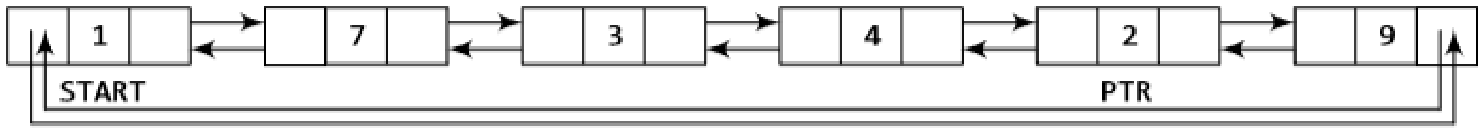


Figure 6.59 Inserting a new node at the end of a circular doubly linked list

Circular Doubly Linked Lists

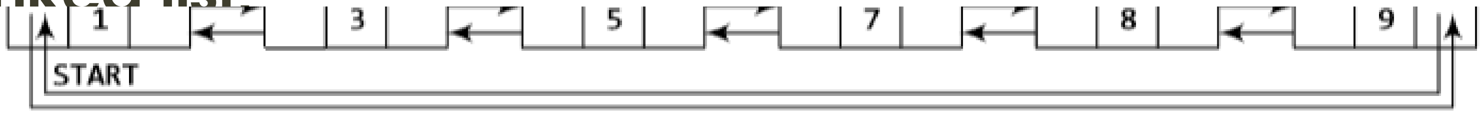
- Inserting a Node at the End of a Circular Doubly Linked List
- Figure 6.60 shows the algorithm to insert a new node at the end of a circular doubly linked list.
- In Step 6, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node.
- The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the 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
```

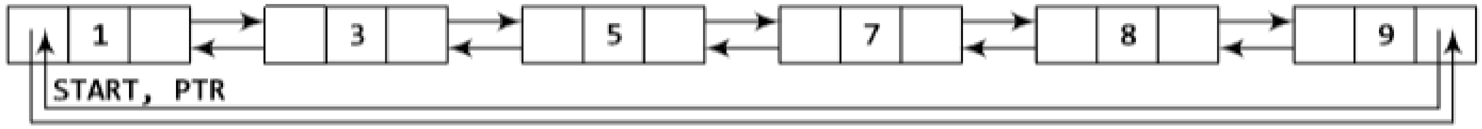
Figure 6.60 Algorithm to insert a new node at the end

Circular Doubly Linked Lists

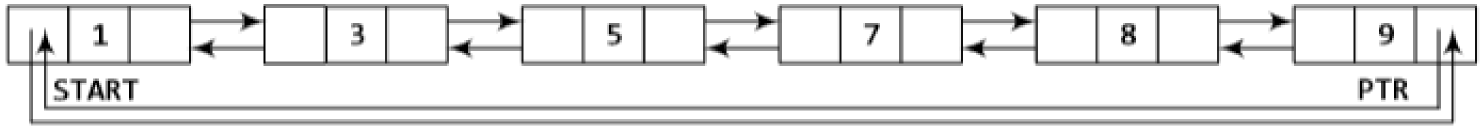
- **Deleting the First Node from a Circular Doubly Linked List**
- **Consider the circular doubly linked list shown in Fig. 6.61.**
- **When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.**



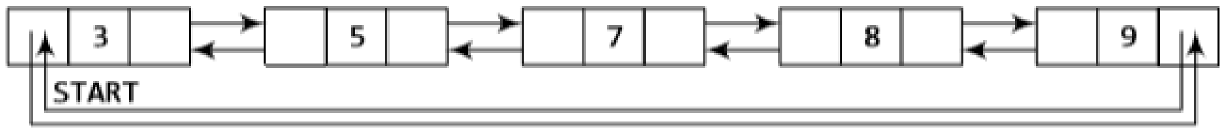
Take a pointer variable PTR that points to the first node of the list.



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



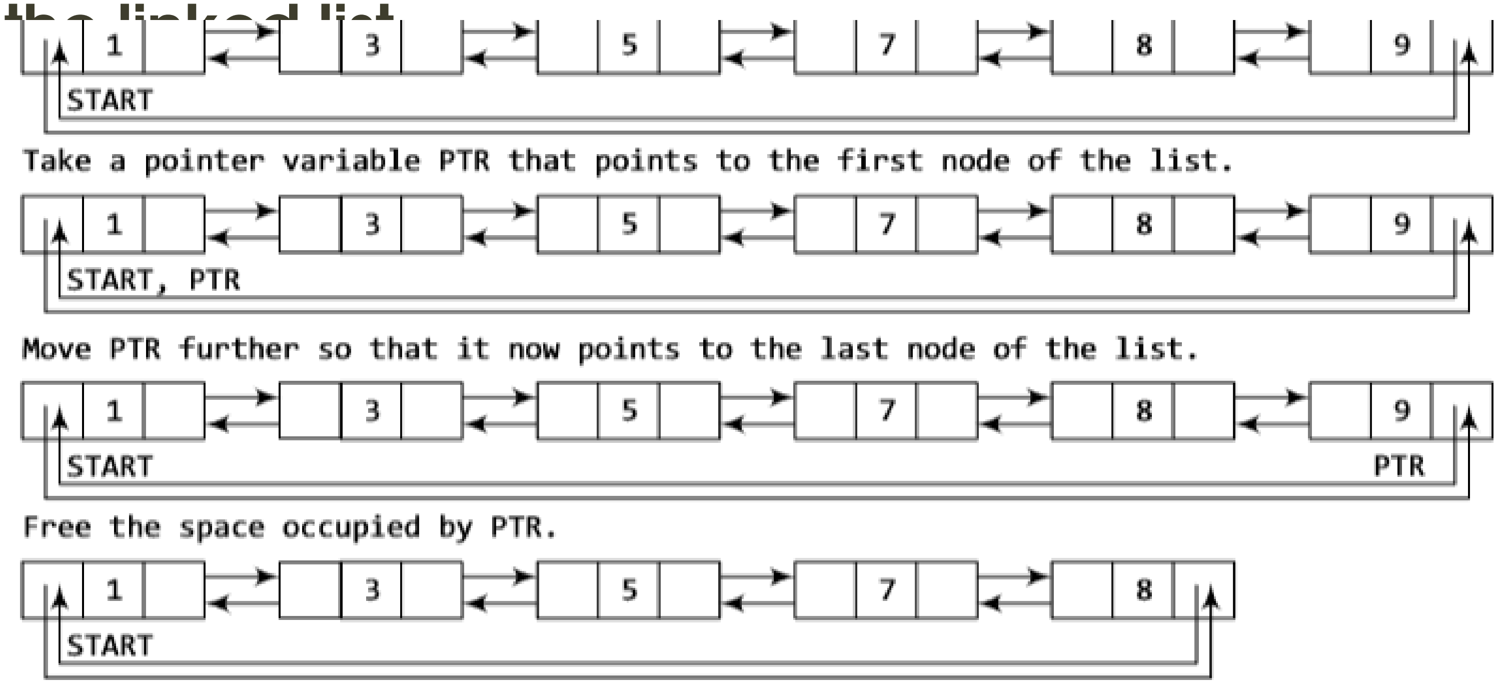
Make START point to the second node of the list. Free the space occupied by the first node.



[illegible]

Circular Doubly Linked Lists

- Deleting the Last Node from a Circular Doubly Linked List
- Consider the circular doubly linked list shown in Fig. 6.63.
- Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list



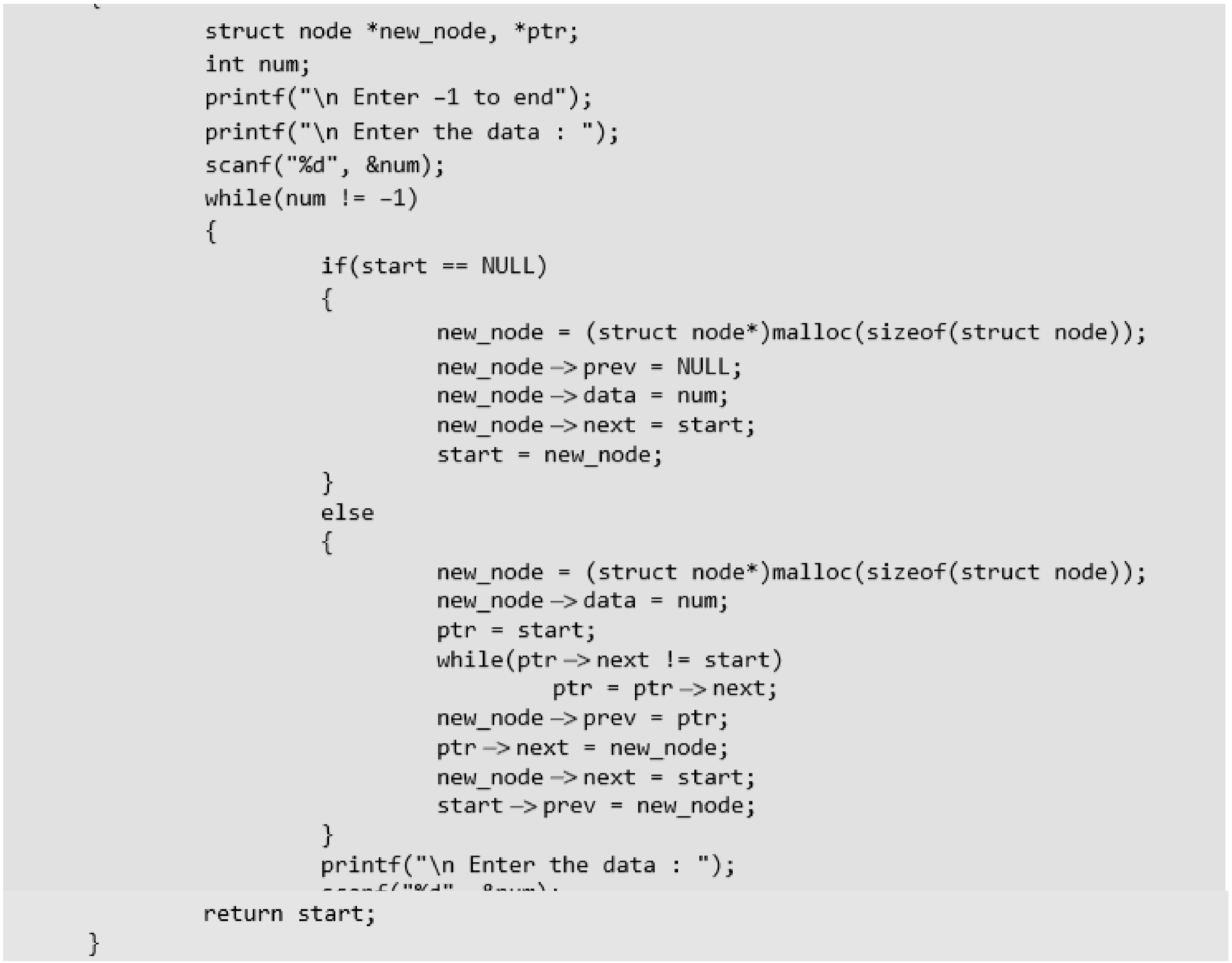
Take a pointer variable PTR that points to the first node of the list.

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

Circular Doubly Linked Lists



```
    {
            struct node *new_node, *ptr;
            int num;
            printf("\n Enter -1 to end");
            printf("\n Enter the data : ");
            scanf("%d", &num);
            while(num != -1)
            {
                    if(start == NULL)
                    {
                            new_node = (struct node*)malloc(sizeof(struct node));
                            new_node -> prev = NULL;
                            new_node -> data = num;
                            new_node -> next = start;
                            start = new_node;
                    }
                    else
                    {
                            new_node = (struct node*)malloc(sizeof(struct node));
                            new_node -> data = num;
                            ptr = start;
                            while(ptr -> next != start)
                                    ptr = ptr -> next;
                            new_node -> prev = ptr;
                            ptr -> next = new_node;
                            new_node -> next = start;
                            start -> prev = new_node;
                    }
                    printf("\n Enter the data : ");
                    scanf("%d", &num);
            return start;
    }
```

Circular Doubly Linked Lists

```
struct node *insert_beg(struct node *start)
{
        struct node *new_node, *ptr;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node-> data = num;
        ptr = start;
        while(ptr -> next != start)
                ptr = ptr -> next;
        new_node -> prev = ptr;
        ptr -> next = new_node;
        new_node -> next = start;
        start -> prev = new_node;
        start = new_node;
        return start;
}
```

Circular Doubly Linked Lists

```
struct node *delete_node(struct node *start)
{
        struct node *ptr;
        int val;
        printf("\n Enter the value of the node which has to be deleted : ");
        scanf("%d", &val);
        ptr = start;
        if(ptr -> data == val)
        {
                start = delete_beg(start);
                return start;
        }
        else
        {
                while(ptr -> data != val)
                        ptr = ptr -> next;
                ptr -> prev -> next = ptr -> next;
                ptr -> next -> prev = ptr -> prev;
                free(ptr);
                return start;
        }
}
```

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.

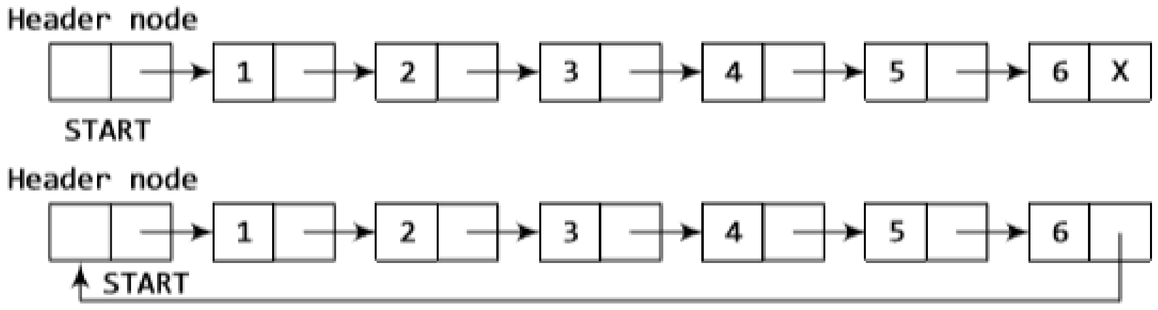
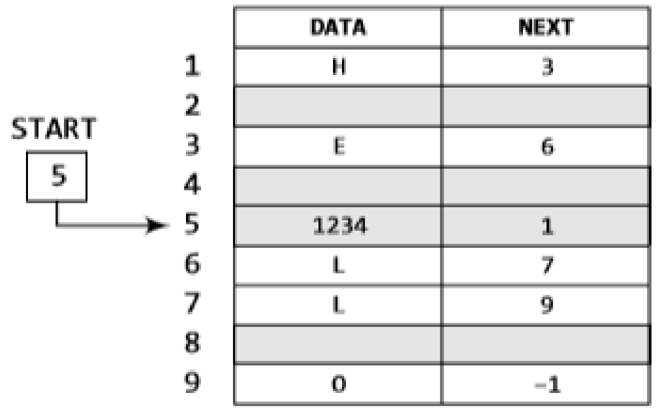


Figure 6.65 Header linked list

Header Linked Lists

- As in other linked lists, if START = NULL, then this denotes an empty header linked list.
- Let us see how a grounded header linked list is stored in the memory.
- 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.



	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

Header Linked Lists

- 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.
- 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.

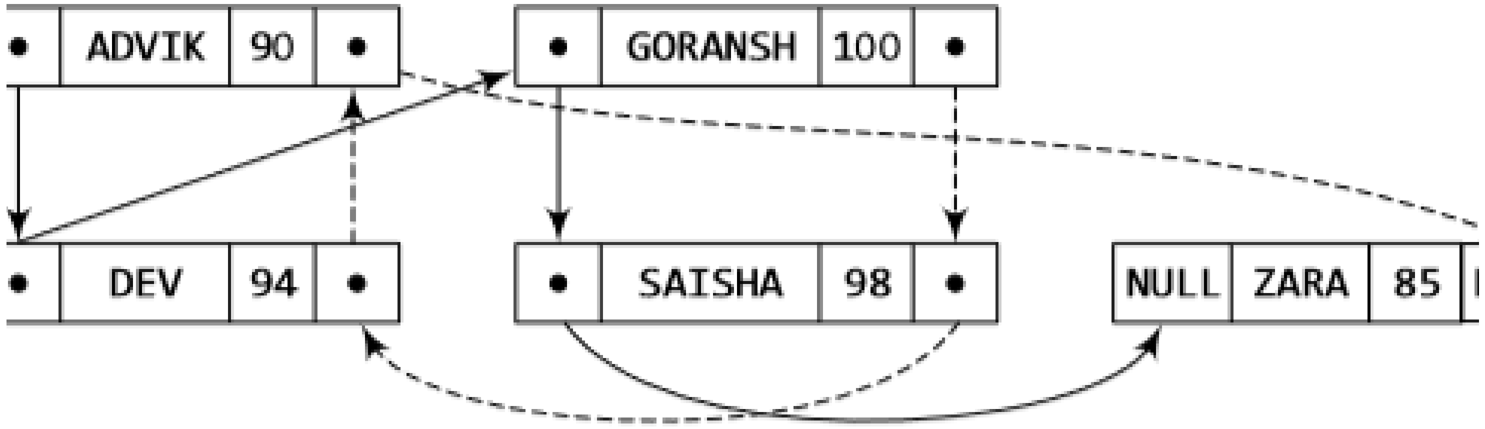
Multi-Linked Lists

- 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.

Multi-Linked Lists



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

Applications of Linked Lists

- Polynomial representation
- Let us see how a polynomial is represented in the memory using a linked list.
- 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.

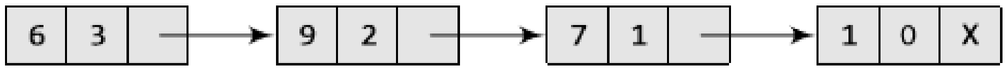


Figure 6.74 Linked representation of a polynomial

Applications of Linked Lists

- **Polynomial representation**

```
scanf("%d", &n);
printf("\t Enter its coefficient : ");
scanf("%d", &c);
while(n != -1)
{
        if(start==NULL)
        {
                new_node = (struct node *)malloc(sizeof(struct node));
                new_node -> num = n;
                new_node -> coeff = c;
                new_node -> next = NULL;
                start = new_node;
        }
        else
        {
                ptr = start;
                while(ptr -> next != NULL)
                        ptr = ptr -> next;
                new_node = (struct node *)malloc(sizeof(struct node));
                new_node -> num = n;
                new_node -> coeff = c;
                new_node -> next = NULL;
                ptr -> next = new_node;
        }
        printf("\n Enter the number : ");
        scanf("%d", &n);
        if(n == -1)
                break;
```