# Linked Lists#1

In this lecture, the following topics are covered:

- ❑ The Concept of Linked Lists
- ❑ Dynamic Memory Management
- ❑ Singly Linked Lists
- ❑ A Singly Linked List implementation in C

## 1. The Concept of Linked Lists

- A linked list is a linear data structure consisting of a collection of data elements. These data elements are called **nodes**.
- A linked list is a data structure which in turn can be used to implement other data structures such as stacks, queues, and their variations.
- A linked list can be perceived as a sequence of **nodes** in which each node contains one or more **data fields** and a **pointer** to the **next** node.

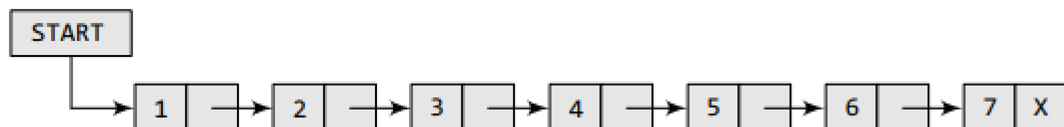



*Figure 1. A simple linked list.*

- In Fig. 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.1, the NULL pointer is represented by X.
- 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.
- 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;
};
```

- Unlike arrays, linked lists do not store their nodes in consecutive memory locations.
- Unlike an array, a linked list does not allow random access of data. Nodes in a linked list can be accessed only in a **sequentia**l manner. In other words, the processing of a linked list always starts at the first node.
- But like an array, insertions and deletions can be done at any point in the list in a constant time.
- Unlike arrays, linked lists are **dynamic**. This means that linked lists can grow and shrink at execution time.

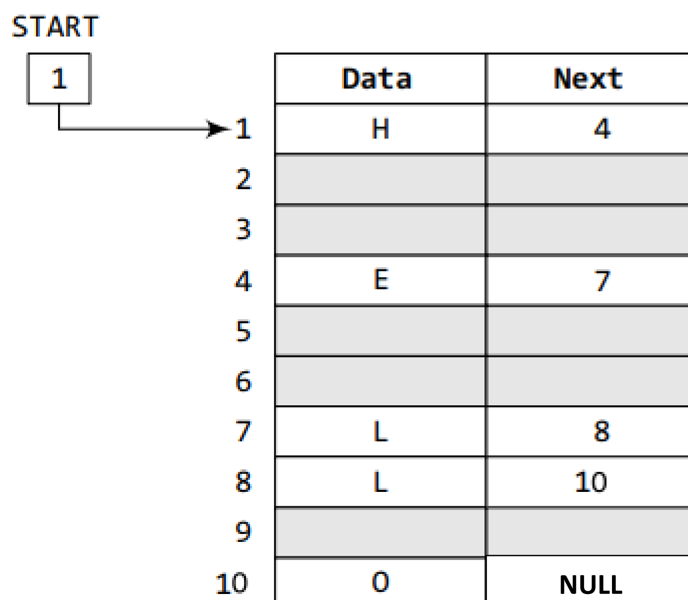
*Linked lists provide an efficient way of storing related data and perform basic operations such as insertion, deletion, and updating of information at the cost of extra space required for storing address of the next node.

## 2. Linked List Representation in Memory

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. 2. 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 **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.




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

*Figure 2. Memory representation of a simple linked list.*

- Now, look at Fig. 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 ambiguity stored in the NEXT.

- By looking at the figure, we can conclude that numbers of the students who have opted for Biology are S01, S03, S06, S08, S10, and S11. Similarly, roll numbers of the students who chose Computer Science are S02, S04, S05, S07, and S09.

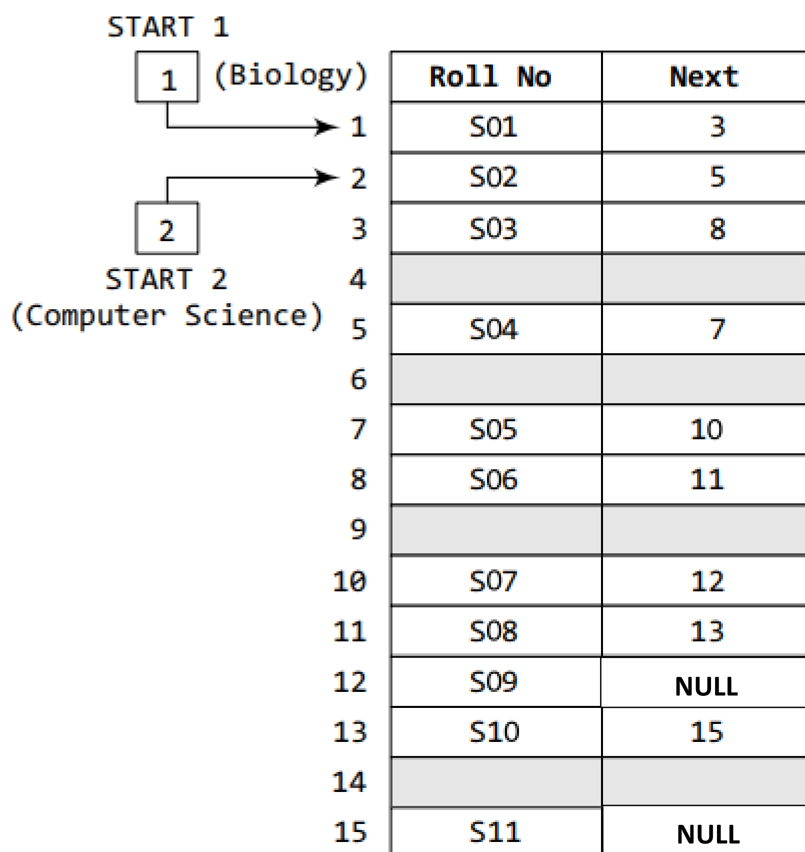



*Figure 3. Two linked lists which are simultaneously maintained in the memory.*

## 3. Dynamic Memory Management

- Creating and maintaining dynamic data structures (linked lists for example) that grow and shrink at execution time requires dynamic memory management, which has two components:
  - obtaining more memory at execution time to hold new nodes, and
  - releasing memory that is no longer needed.
- The function malloc, the function free and the operator **sizeof** are essential to dynamic memory management.

### 3.1 The malloc Function

- To request memory at execution time, pass to the function **malloc** the number of bytes to **allocate**.
- If successful, **malloc** returns a **void** * pointer to the allocated memory. Recall that a void * pointer may be assigned to a variable of any pointer type.
- Function malloc most commonly is used with **sizeof**. For example, the following statement determines a struct node object's size in bytes with **sizeof** (**struct** node), allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in **newPtr**:

```
newPtr = malloc(sizeof(struct node));
```

- The memory is not guaranteed to be initialized, though many implementations initialize it for security.
- If no memory is available, malloc returns NULL.
- Always test for a NULL pointer before accessing the dynamically allocated memory to avoid runtime errors that might crash your program.

### 3.2 The free Function

- When you no longer need a block of dynamically allocated memory, return it to the system immediately by calling the **free** function to **deallocate** the memory. This returns it to the system for potential reallocation in the future. To free the memory from the preceding malloc call, use the statement

```
free(newPtr);
```

- After deallocating memory, set the pointer to **NULL**. This prevents accidentally referring to that memory, which may have already been allocated for another purpose.
- Not freeing dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "memory leak."
- Referring to memory that has been freed is an error that typically causes a program to crash.

- Freeing memory that you did not allocate dynamically with **malloc** is an error.

## 4. 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 4 shows a singly linked list.

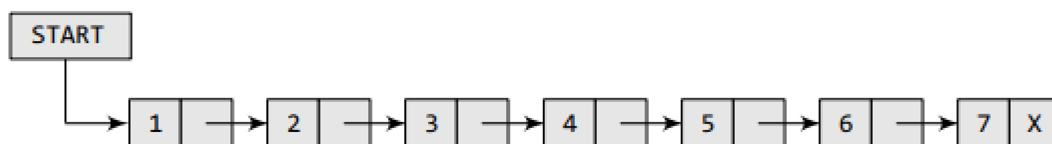



*Figure 4. A Singly Linked List.*

### 4.1 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 (Head) 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 Figure 5.

```
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 5. Algorithm for 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.

### 4.2 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.
- Figure 6 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. Algorithm to search a linked list.*

- Consider the linked list shown in Fig. 7. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.

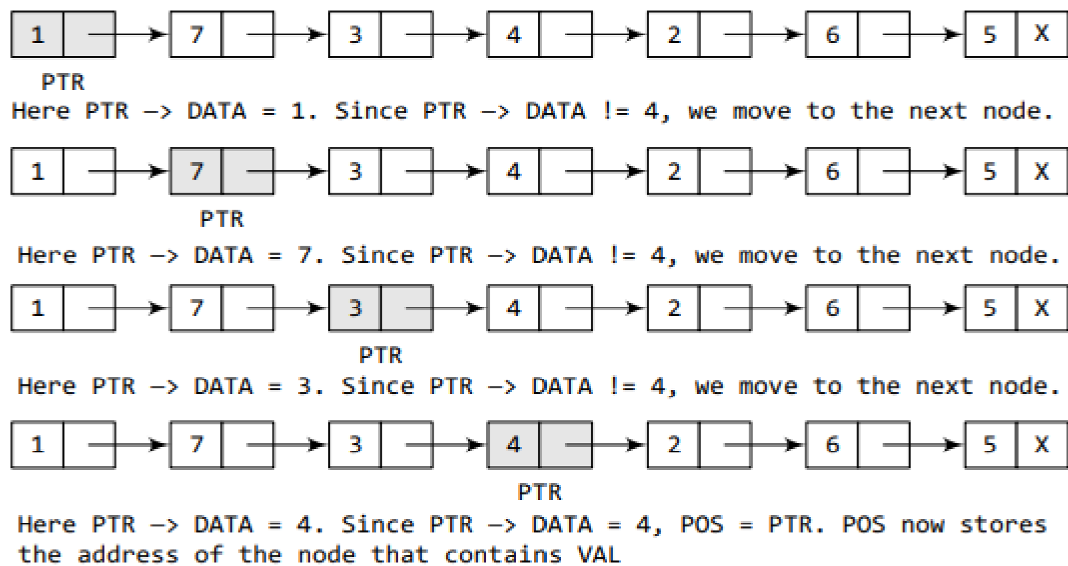



*Figure 7. Searching a linked list*

## 4.3 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.

### 4.3.1 Inserting a Node at the Beginning of a Linked List

- Consider the linked list shown in Fig. 8.
- 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.

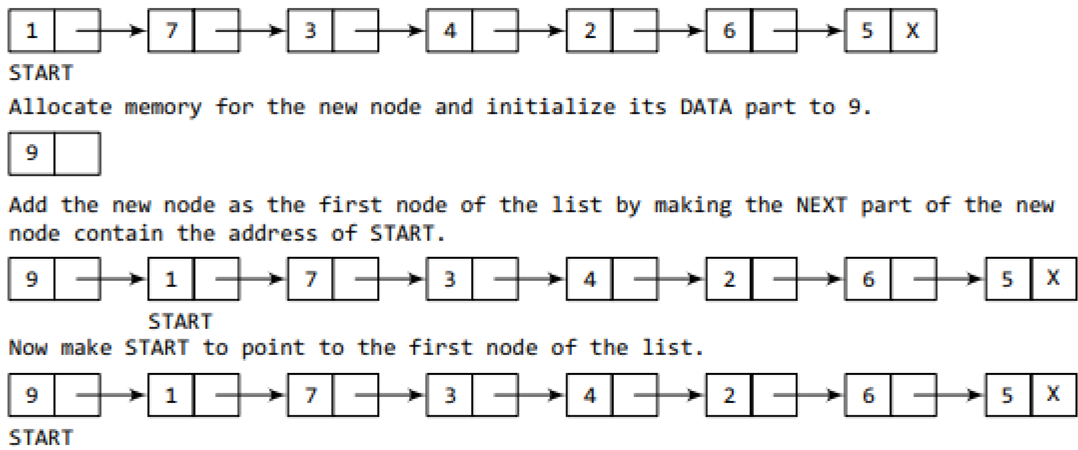



*Figure 8. Inserting an element at the beginning of a linked list.*

### 4.3.2 Inserting a Node at the End of a Linked List

- Consider the linked list shown in Fig 9.
- 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:

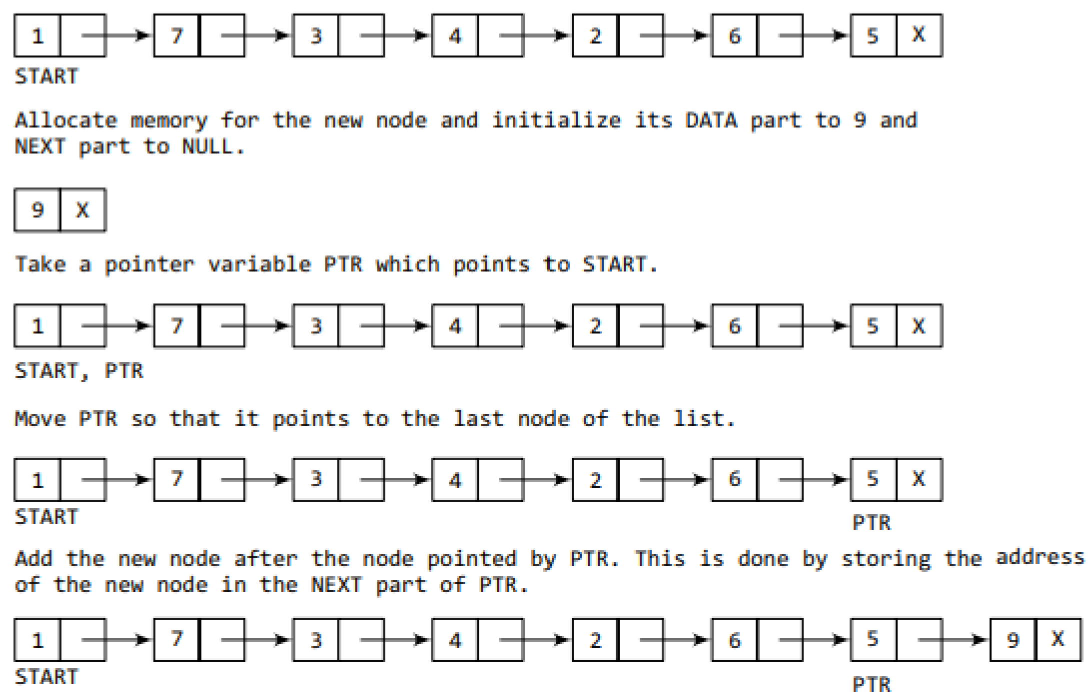



*Figure 9. Inserting an element at the end of a linked list.*

### 4.3.3 Inserting a Node After a Given Node in a Linked List

- Consider the linked list shown in Fig. 10.
- Suppose we want to add a new node with value 9 after the node containing data 3.

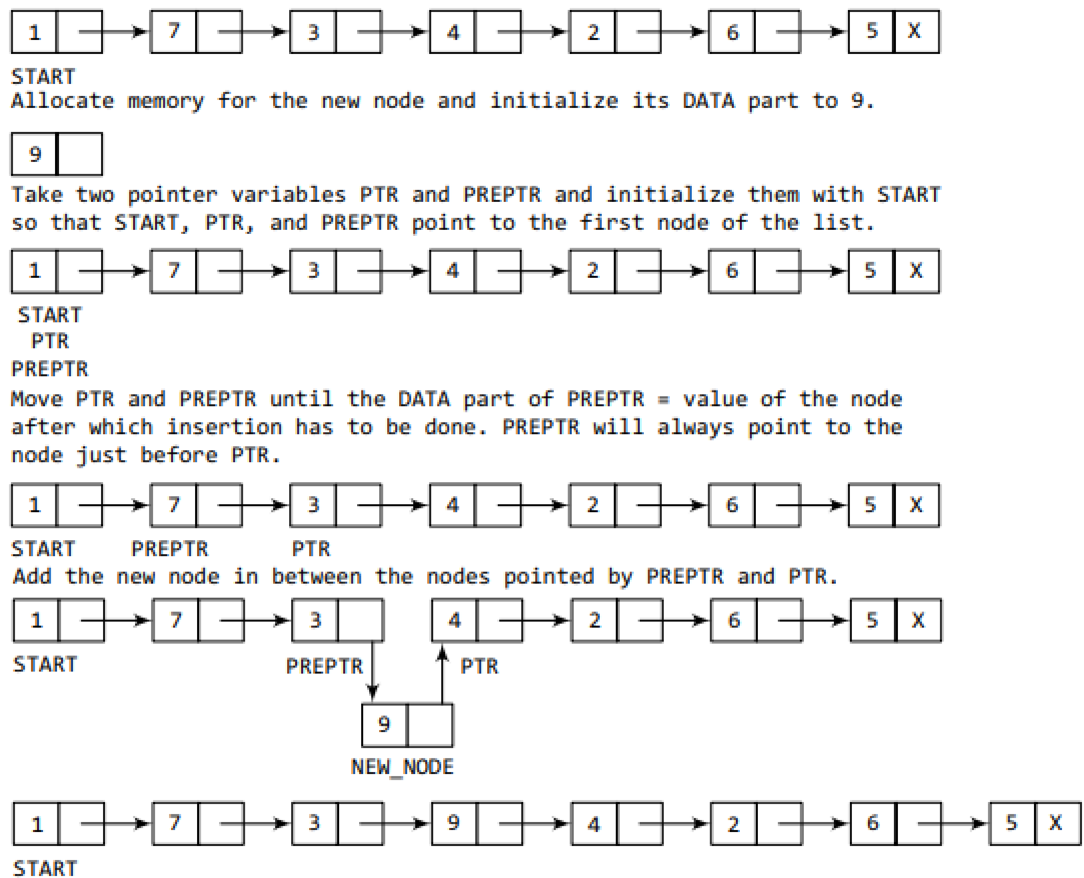



*Figure 10. Inserting an element after a given node in a linked list.*

### 4.3.4 Inserting a Node Before a Given Node in a Linked List

- Consider the linked list shown in Fig. 11.
- Suppose we want to add a new node with value 9 before the node containing 3.

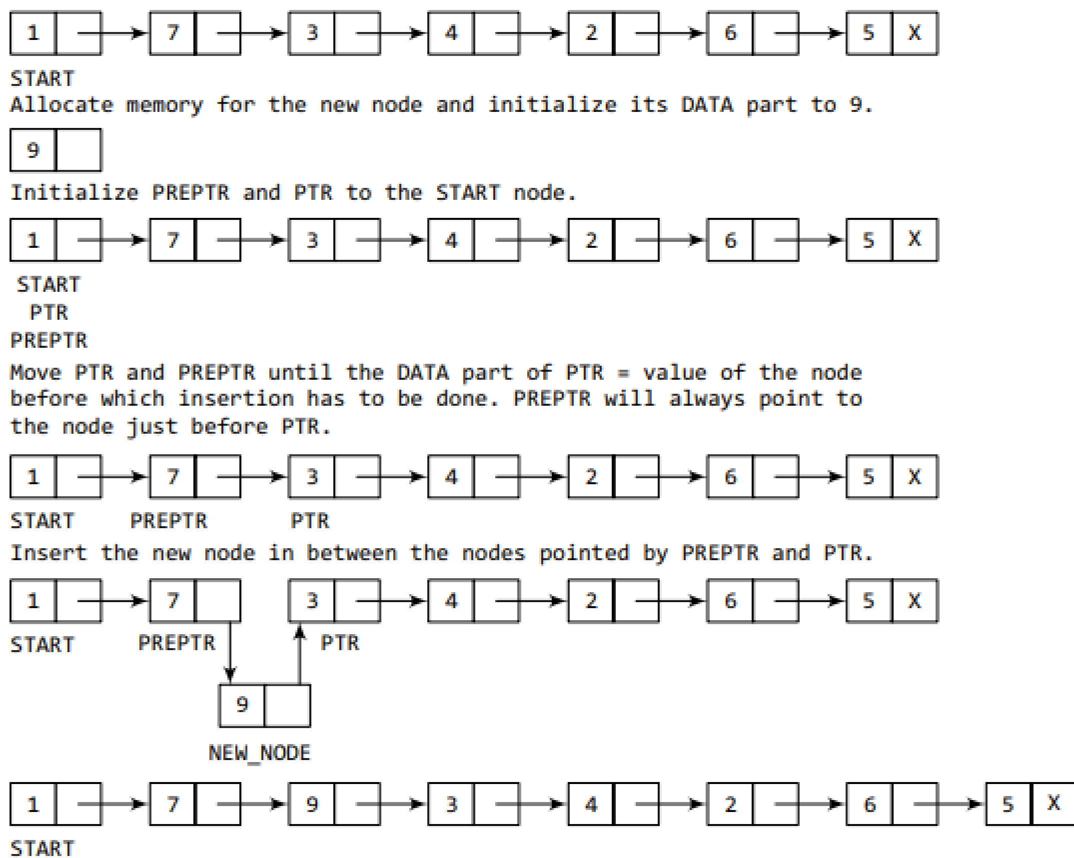



*Figure 11. Inserting an element before a given node in a linked list.*

### 4.4 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.

### 4.4.1 Deleting the First Node from a Linked List

- Deleting the First Node from a Linked List Consider the linked list in Fig. 12.
- When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list:

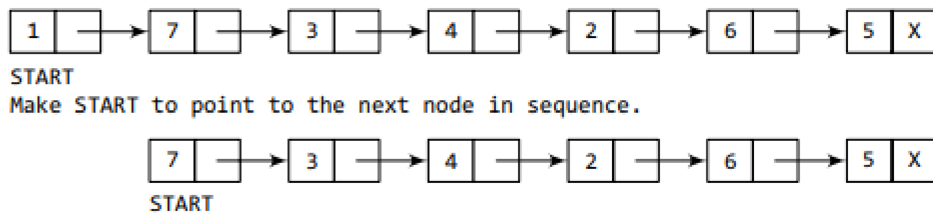



*Figure 12. Deleting the first node of a linked list.*

- Figure 13 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, as shown in Step 2, 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 13. Algorithm to delete the first.*

### 4.4.2 Deleting the Last Node from a Linked List

- Consider the linked list shown in Fig. 14. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list:

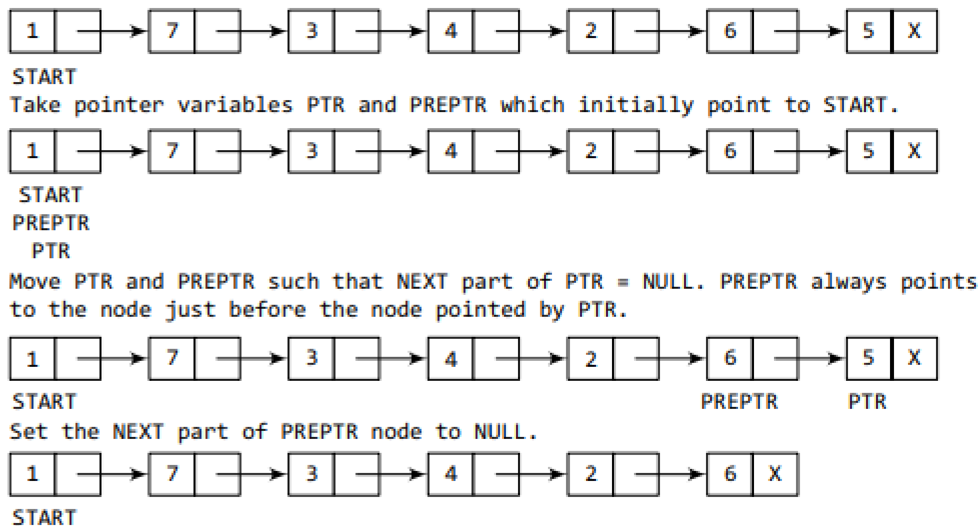



*Figure 14. Deleting the last node of a linked list.*

- Figure 15 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 15. Algorithm to delete the last node.*

### 4.4.3 Deleting the Node After a Given Node in a Linked List

- Consider the linked list shown in Fig. 16. 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:

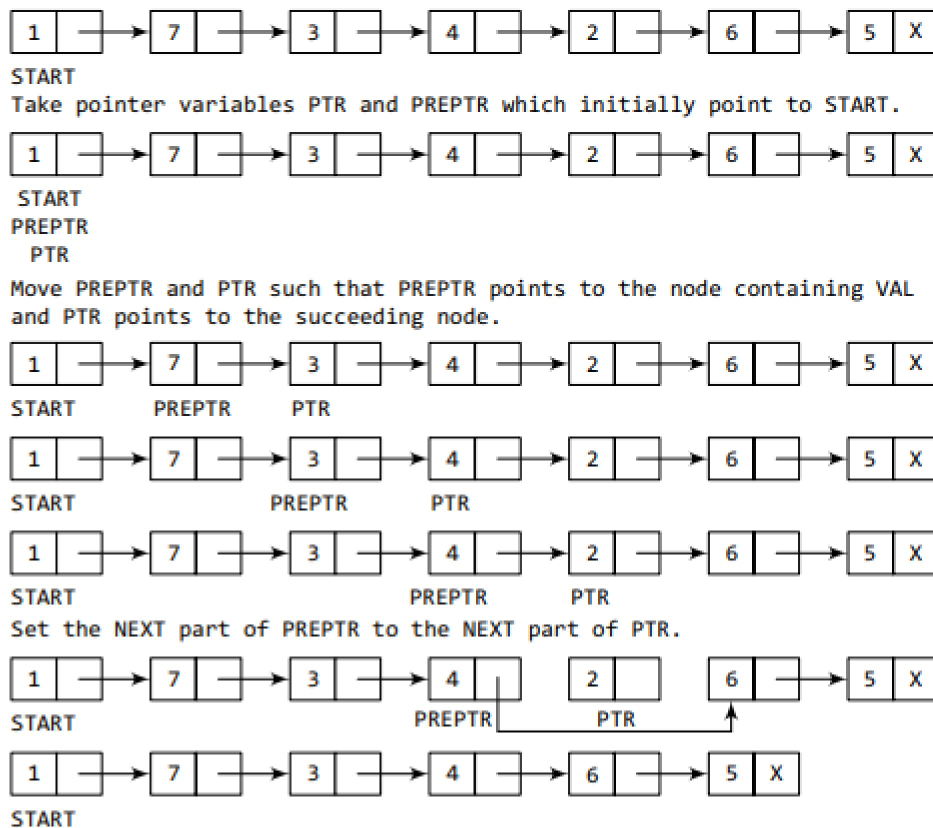



*Figure 16. Deleting the node after a given node in a linked list.*

- Figure 17 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.

```
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 17. Algorithm to delete the node after a given node.*

## 5. Linked lists in C

The following C program shows how to implement the different operations (insertion, deletion and search) that can be performed on linked lists which have been explained earlier:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct node* addAtStart(struct node* head, int data);
void addAtEnd(struct node* head, int data);
bool addAfterNode(struct node* head, int data, int ref);
bool addBeforeNode(struct node* head, int data, int ref);

struct node* deleteFromStart(struct node* head);
struct node* deleteFromEnd(struct node* head);
struct node* deleteNode (struct node* head, int data);
struct node* search (struct node* head, int data);

void printLinkedList (struct node* head);
struct node* lastNode(struct node* head);
bool isEmpty (struct node* head);

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

```c
int main() {
    struct node* head = NULL;
    int val, ref, option;
do
{
    printf("\n *****MAIN MENU*****");
    printf("\n 1. Add node at beginning of list");
    printf("\n 2. Add node at end of list");
    printf("\n 3. Add node after specific Node");
    printf("\n 4. Add node before specific Node");
    printf("\n 5. Delete node from beginning of list");
    printf("\n 6. Delete node from end of list");
    printf("\n 7. Delete specific node from list");
    printf("\n 8. Print linked list");
    printf("\n 9. EXIT");
    printf("\n Enter your option: ");
    scanf("%d", &option);
    switch(option)
    {
    case 1:
        printf("Enter int value for new node:");
        scanf("%d", &val);
        head = addAtStart(head, val);
        break;

 case 2:
     printf("Enter int value for new node:");
     scanf("%d", &val);
     if (isEmpty(head)){
         head = addAtStart(head, val);
     }
     else{
         addAtEnd(head, val);
     }
     break;
 case 3:
     printf("Enter int value for the node to add after:");
     scanf("%d", &ref);
     printf("Enter int value for the node to be added:");
     scanf("%d", &val);
     if (addAfterNode(head, val,  ref)){
            printf("OK. node is added!\n");
            printf("Head node is %d\n", head->data);
         }
     else{
         printf("Sorry node %d does not exist!\n", ref);
     }
     break;
```

```
case 4:
    printf("Enter int value for the node to add before:");
    scanf("%d", &ref);

    printf("Enter int value for the node to be added:");
    scanf("%d", &val);

    if (addBeforeNode(head, val,  ref)){
            printf("OK. node is added!\n");
            printf("Head node is %d\n", head->data);
        }
    else{
        printf("Sorry node %d does not exist!\n", ref);
    }
    break;
case 5:
   if (!isEmpty(head)){
        head = deleteFromStart(head);
   }
   else{
       printf("Sorry linked list is empty!\n");
   }

    break;

case 6:
   if (!isEmpty(head)){
         head = deleteFromEnd(head);
         printf("Last node deleted..\n");
   }
   else{
       printf("Sorry linked list is empty!\n");
   }
    break;
case 7:
   if (!isEmpty(head)){
        printf("Enter int value for the node to be deleted:");
        scanf("%d", &val);
        head =  deleteNode ( head, val);
   }
   else{
        printf("Sorry linked list is empty!\n");
   }
    break;
case 8:
    printLinkedList (head);
    break;
 }//End of switch
```

```
}while(option != 9);
  return 0;
}

//Functions for insertion and deletion
  struct node* lastNode(struct node* head){
      struct node *last;
      last = head;
      while(last->next!= NULL ){
       last = last->next;
      }
  return last;
  }
   struct node* addAtStart(struct node* head, int data){
      struct node *newNode;
      newNode = malloc(sizeof(struct node));
     newNode->data = data;
    newNode->next = head;
    head = newNode;
    return head;
  }
  void addAtEnd(struct node* head, int data){
      struct node *newNode, *last;
      newNode = malloc(sizeof(struct node));
      last = lastNode(head);
      newNode->data = data;
      newNode->next = NULL;
      last->next = newNode;
  }

bool addAfterNode(struct node* head, int data, int ref){
      struct node *newNode, *current;
     current = head;
    while(current!= NULL ){
            if (current->data == ref){
                newNode = malloc(sizeof(struct node));
                newNode->data = data;
                newNode->next = current->next;
                current->next = newNode;
                return true;
            }

   current = current->next;
  }

    return false;
  }
```

```
bool addBeforeNode(struct node* head, int data, int ref){
     struct node *newNode, *current;
     current = head;
     while(current!= NULL ){
           if (current->next->data == ref){
               newNode = malloc(sizeof(struct node));
               newNode->data = data;
                newNode->next = current->next;
               current->next = newNode;

               return true;
           }
      current = current->next;
     }
    return false;
}

 bool isEmpty (struct node* head){
   return (head == NULL);
 }

struct node* deleteNode (struct node* head, int data)
  {
       struct node *previous, *current ;
       current = head;
       if (current->data == data){
         head = deleteFromStart(head);
         return head;
       }
       else{
        while (current->data != data){
             previous = current;
             current = current->next;
        }
         previous->next = current->next;
         free (current);
          return head;
       }
  }
  struct node* deleteFromStart(struct node* head){
      struct node* temp;
      temp = head;
      head = head->next;
      free(temp);
      return head;
  }
```

```c
struct node* deleteFromEnd(struct node* head){

        struct node *previous,*current;
               current = head;
               previous = head;
          while(current->next != NULL){
                   previous = current;
                   current = current->next;
          }
          if (current == head){
               head = NULL;

          }

          else{
               previous->next = NULL;
               free(current);
          }
          return head;
 }

struct node* search (struct node* head, int data){
    struct node *current;
     current = head;
     while(current!= NULL ){
           if (current->data == data){
               return current;
           }
      current = current->next;
}
  return NULL;
}

void printLinkedList(struct node* head){
  if (isEmpty(head))
      printf("Sorry, linked list is empty!\n");
  else{
     struct node *current;
     current = head;
     while(current!= NULL){
         printf("%d ",current->data);
         current = current->next;
     }
  }
printf("\n");
}
```