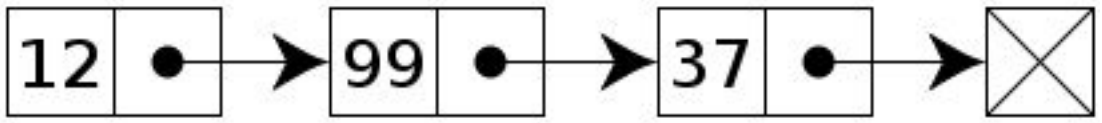
Chapter#5
Linked lists: Linear Data Structure

Introduction

- Consisting of a group of nodes which together represent a sequence.
- Under the simplest form, each node is composed of a datum and a reference (in other words, a *link*) to the next node in the sequence.
- This structure allows for efficient insertion or removal of elements from any position in the sequence.

Introduction

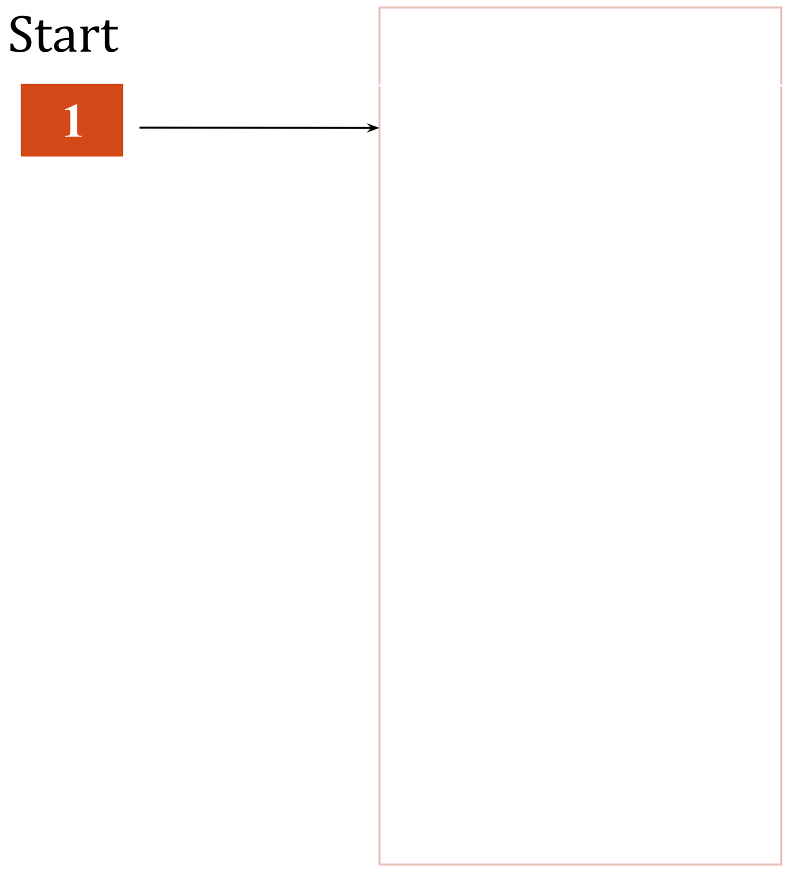


A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator (NULL pointer) used to signify the end of the list.

Introduction

- Advantage: Provides quick insert and delete operations.
- Disadvantage: Slow search operations and requires more memory space.

Memory Representation



After traversal,
we get HELLO within
this linked list

Linked List vs. Arrays

- The *size of the arrays is fixed*: So we must know the upper limit on the number of elements in advance.
- *Inserting* a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

Linked List vs. Arrays

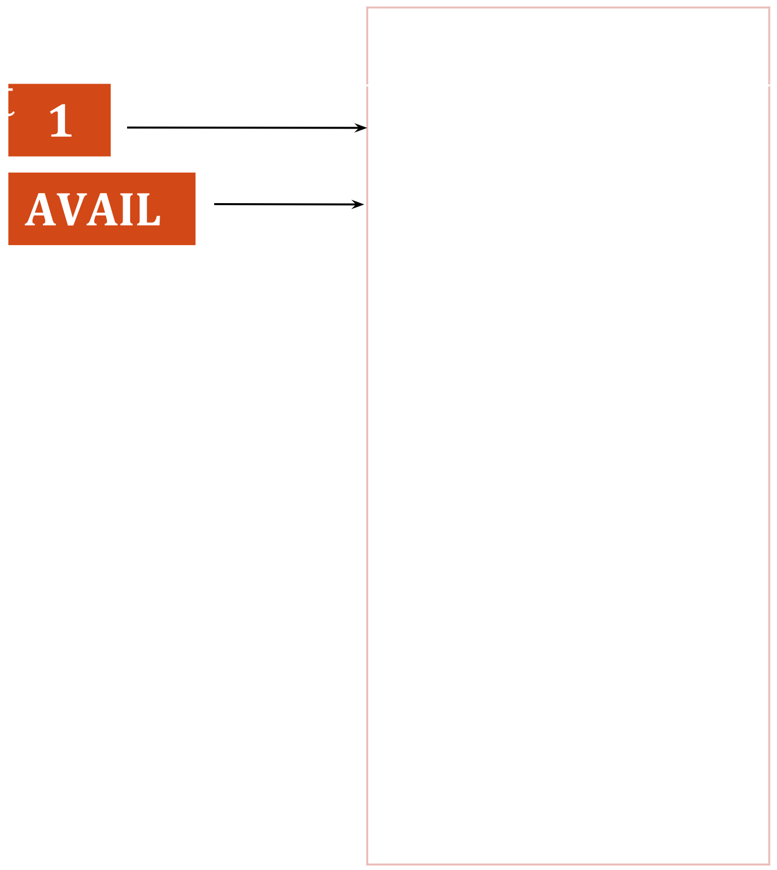
- *Random access is not allowed.* We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.

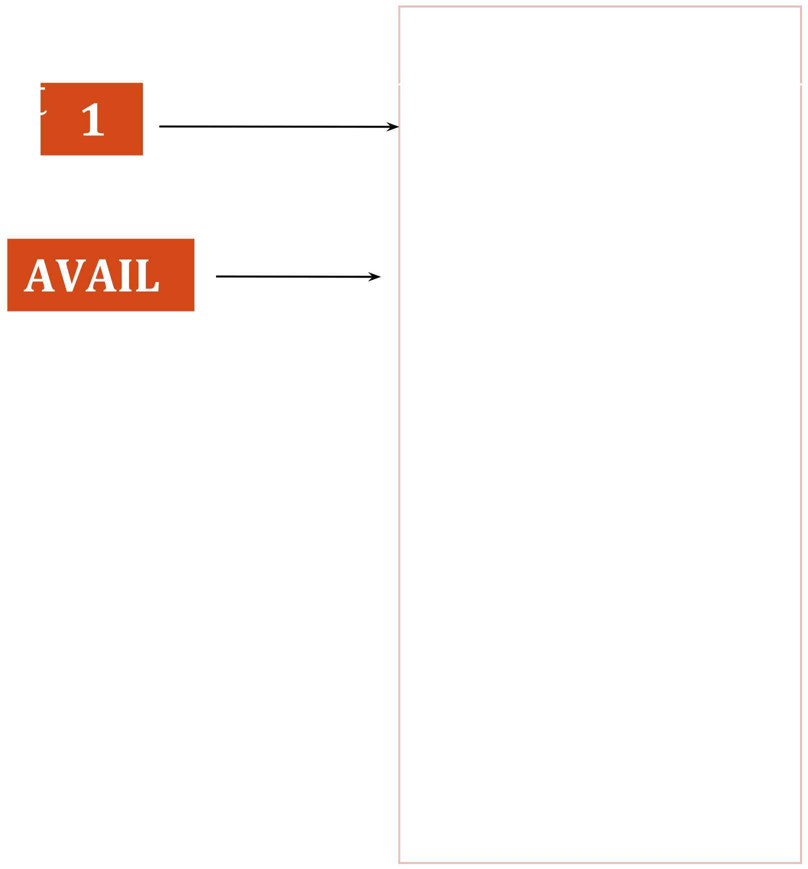
- *Extra memory space* for a pointer is required with each element of the list.

Memory allocation & de-allocation

- Operating system bears the responsibility to change the status of the memory occupied to available and vice versa without intervention from user.
- Consider, we have a pointer AVAIL which stores the address of the first free address. (Free Pool)
- AVAIL will be useful to store new value.

Memory allocation & de-allocation

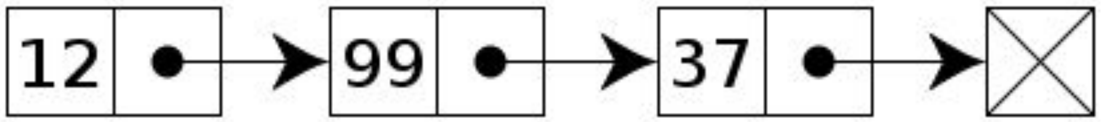




SINGLY LINKED LIST

Singly Linked-List

- Simplest type of linked list in which every node contains data and pointer.



- A singly linked list allows traversal of data only in one way.
- Example: representation of the polynomial equation is done by the mean of singly linked list.

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

```
class linked_list
{
private:
    node *head;
public:
    linked_list()
    {
        head = NULL;
    }
};
```

Algorithm for traversing a linked list

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

Algorithm to print each node of the linked list

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

Algorithm to print the number of nodes in the linked list

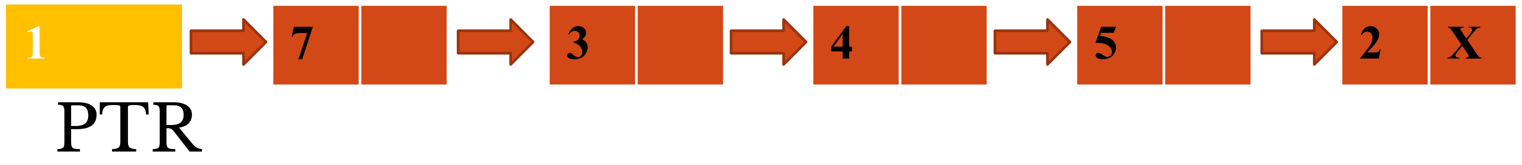
```
Step 1: [INITIALIZE] SET Count = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:     SET Count = Count + 1
Step 5:     SET PTR = PTR -> NEXT
            [END OF LOOP]
Step 6: EXIT
```

Searching an unsorted Linked List

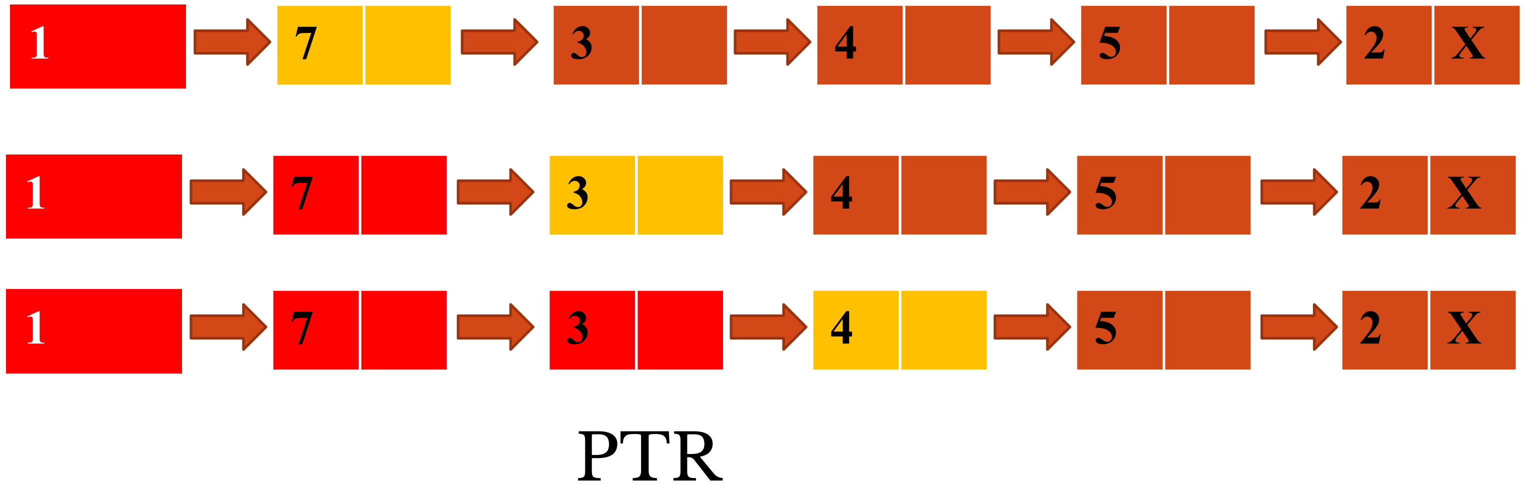
```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 while PTR != NULL
Step 3:     IF VAL = PTR -> DATA
                SET POS = PTR
                Go To Step 5
            ELSE
                SET PTR = PTR -> NEXT
            [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step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

Searching an unsorted linked list

- Find VAL = 4



(Here, PTR -> DATA != 4, so move to next node)



(Here, PTR -> DATA = 4, so POS = PTR)

Searching a sorted (ascending) Linked List

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

```
// This function prints contents of linked list starting from head
void display()
{
    Node* ptr = start;

    if (start== NULL)
    {
      printf("\n The Linked List is empty.\n");
      return;
    }

    printf("\n The singly linked list has following nodes:");
    while (ptr != NULL)
    {
      cout<<ptr->data;
      ptr = ptr->next;
    }
}
```

```
// This function prints contents of linked list starting from
head
void display(struct Node* start_ref)
{
    struct Node* ptr = start_ref;

    if (start_ref == NULL)
    {
      printf("\n The Linked List is empty.\n");
      return;
    }

    printf("\n The singly linked list has following nodes:");
    while (ptr != NULL)
    {
      printf(" %d", ptr->data);
      ptr = ptr->next;
    }
}
```

INSERTIONS IN SINGLY LINKED LIST

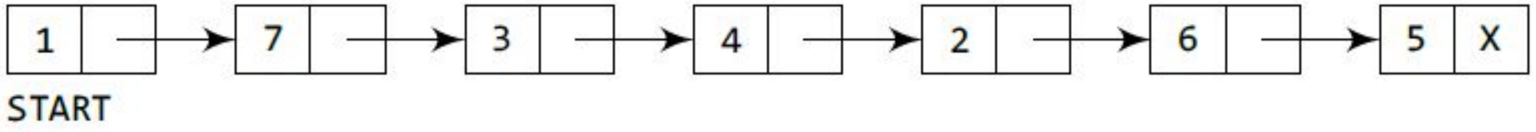
Inserting a new node in LL

Case 1: New node is inserted at the beginning
Case 2: New node is inserted at the end
Case 3: New node is inserted after a given node
Case 4: New node is inserted before a given node

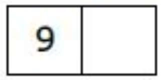
Insertion Case 1:

New node is inserted at the beginning

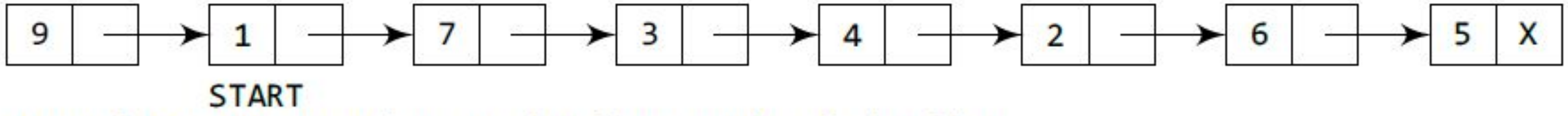
Example



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.

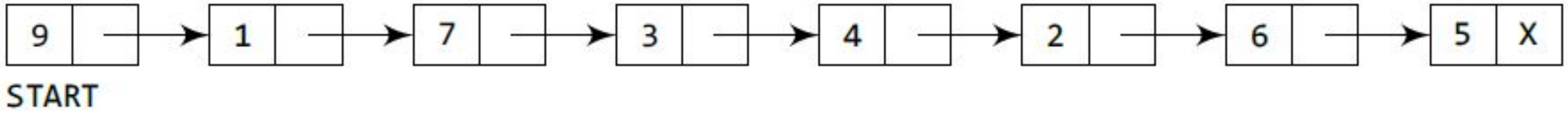


Figure 6.12 Inserting an element at the beginning of a linked list

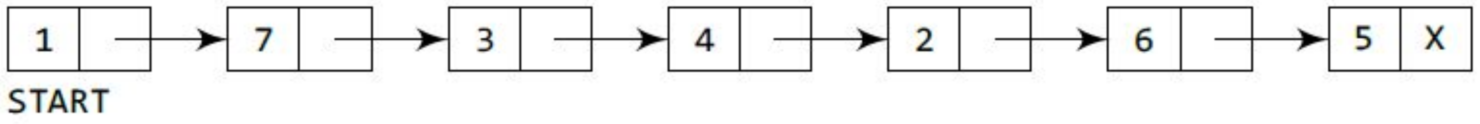
```
void linked_list::add_begin(int n)
    {
        node *tmp = new node;
        tmp->data = n;
        tmp->next = NULL;

       tmp->next = head;
       head = tmp;
}
```

```
void Insert_front(struct Node** start_ref, int new_data)
{
     /* 1. allocate node */
     struct Node* new_node = (struct Node*)
malloc(sizeof(struct Node));

     /* 2. put in the data */
     new_node->data = new_data;

     /* 3. Make next of new node as head */
     new_node->next = (*start_ref);

     /* 4. move the head to point to the new node */
     (*start_ref) = new_node;

     printf("\n The node with value %d has been successfully
inserted in the singly link list.\n",new_data);
}
```

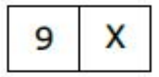
Insertion Case 2:

New node is inserted at the end

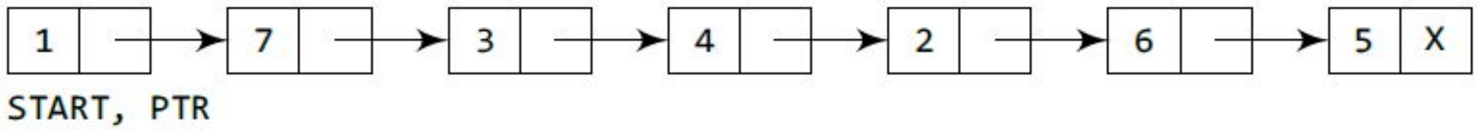
Example



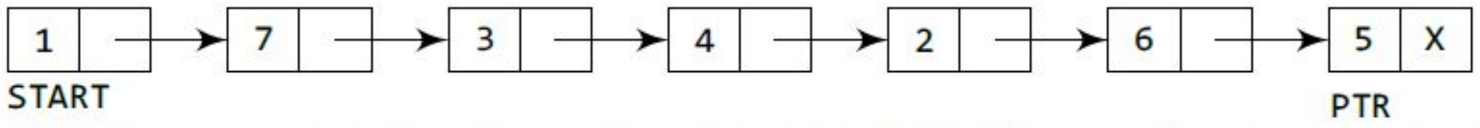
Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.



Take a pointer variable PTR which points to START.



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



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

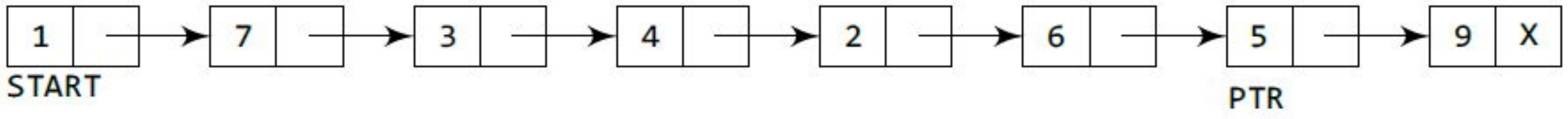


Figure 6.14 Inserting an element at the end of a linked list

```
void linked_list::add_end(int n)
    {
        node *temp = new node;
        temp->data = n;
        temp->next = NULL;

        if(start==NULL)
        {
                start=temp;
                return;
        }

        node *ptr=start;
        while(ptr->next!=NULL)
                ptr=ptr->next;

        ptr->next = temp;
}
```

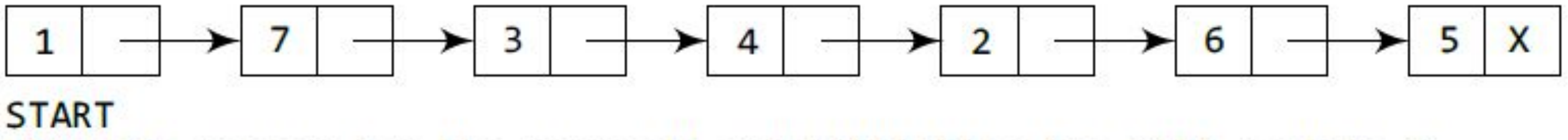
```
void Insert_end(struct Node** start_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*)
malloc(sizeof(struct Node));

    struct Node *last = *start_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so make
next of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as
head */
    if (*start_ref == NULL)
    {
        *start_ref = new_node;
        return;
    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;

    printf("\n The node with value %d has been
successfully inserted in the singly link list.\n",new_data);
}
```

Insertion Case 3:

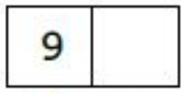
New node is inserted after the given node

Example

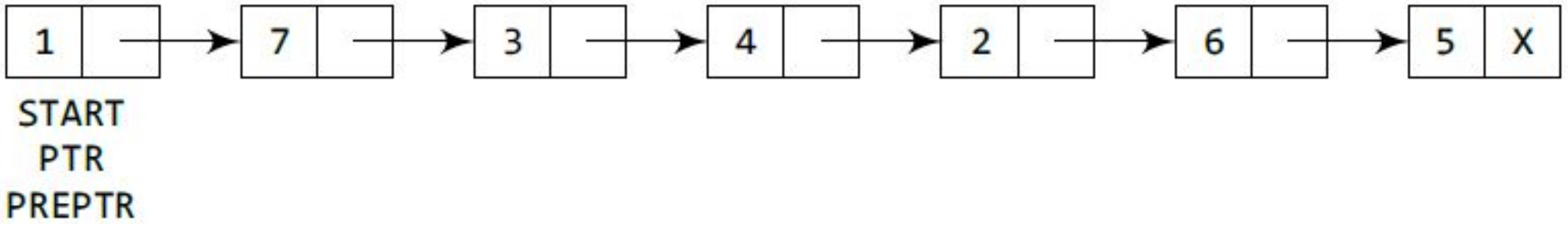
New node is inserted after a given node (after VAL=3)



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



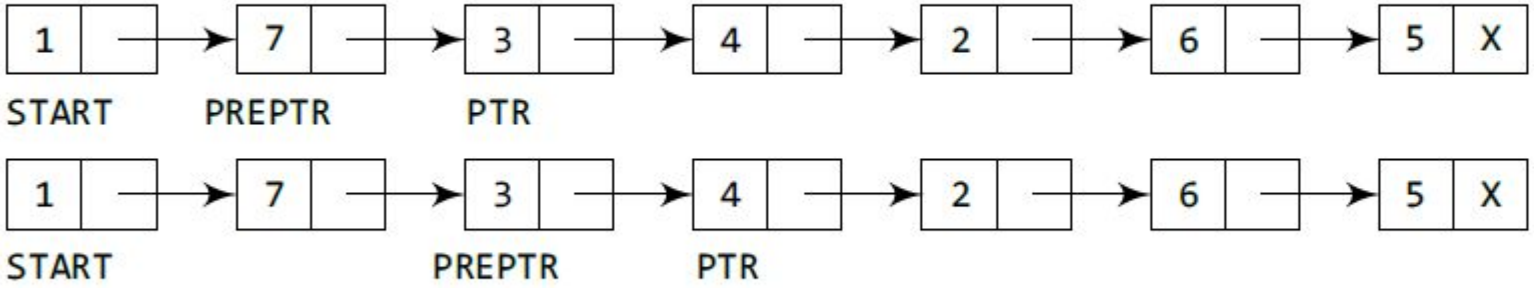
Take two pointer variables PTR and PREPTR and initialize them with START so that START, PTR, and PREPTR point to the first node of the list.



Example

New node is inserted after a given node (after VAL=3)

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.

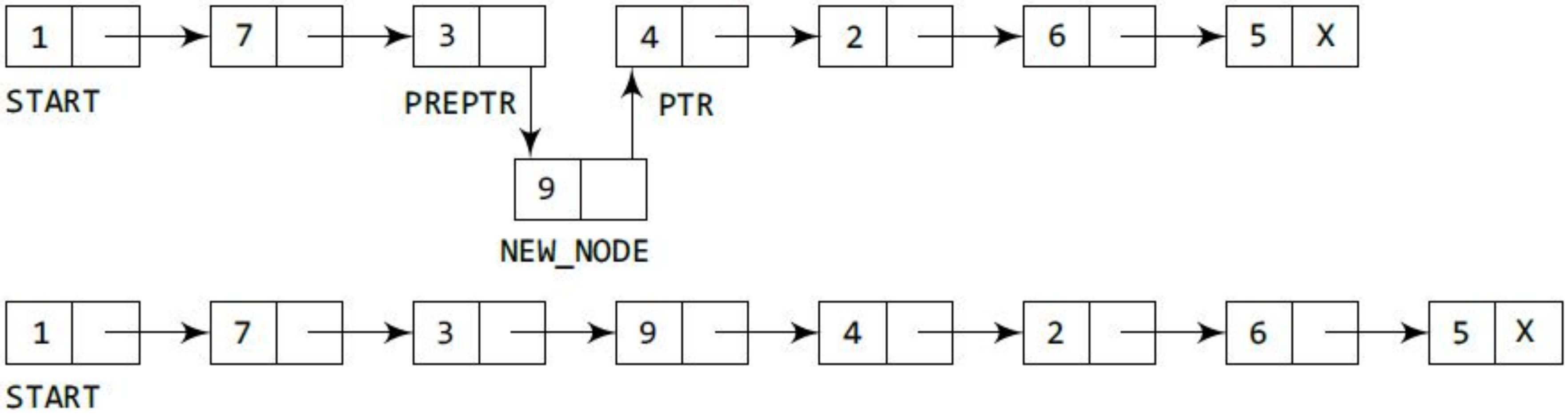


Figure 6.17 Inserting an element after a given node in a linked list

```
void Insert_after_given(struct Node** start_ref, int
search_data, int new_data)
{
   /* 1. allocate node */
   struct Node* new_node = (struct Node*) malloc(sizeof(struct
Node));

   struct Node* ptr = *start_ref;
   int flag=0;

   /* 2. put in the data  */
   new_node->data  = new_data;

   /* 3. Make next of new node as NULL*/
   new_node->next = NULL;

   /* 4. If the Linked List is empty */
   if (*start_ref == NULL)
   {
     printf("\n The Linked List is empty. The node cannot be
inserted\n");
```

```
while(ptr!=NULL)
{
    if(ptr->data==search_data)
    {
        new_node->next=ptr->next;
        ptr->next=new_node;
        flag=1;
        break;
    }
    else
    {
        ptr=ptr->next;
    }
}

if(flag==0)
{
    printf("\n No Node Found.\n");
}
```

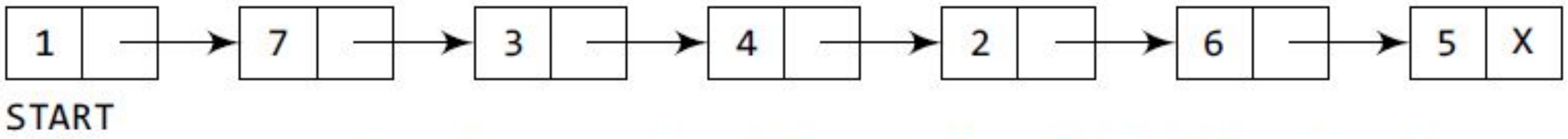
```
	if(flag==1)
	{
		printf("\n The node with value %d has been successfully inserted in the singly link list.\n",new_data);
	}

}
```

Insertion Case 4:

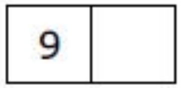
New node is inserted before the given node

Example

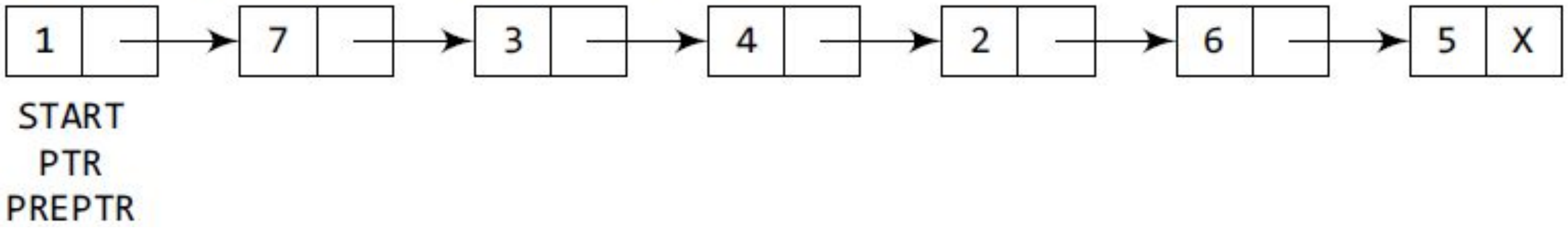
New node is inserted before a given node (Before VAL=3)



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



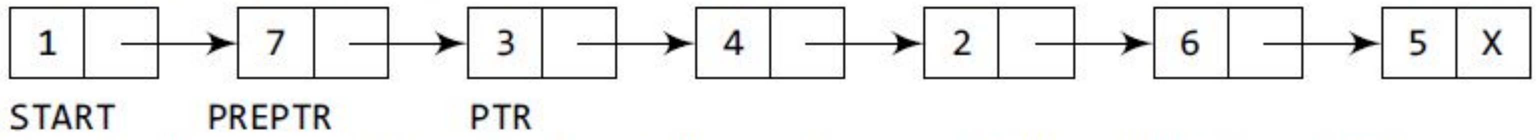
Initialize PREPTR and PTR to the START node.



Example

New node is inserted before a given node (Before VAL=3)

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.

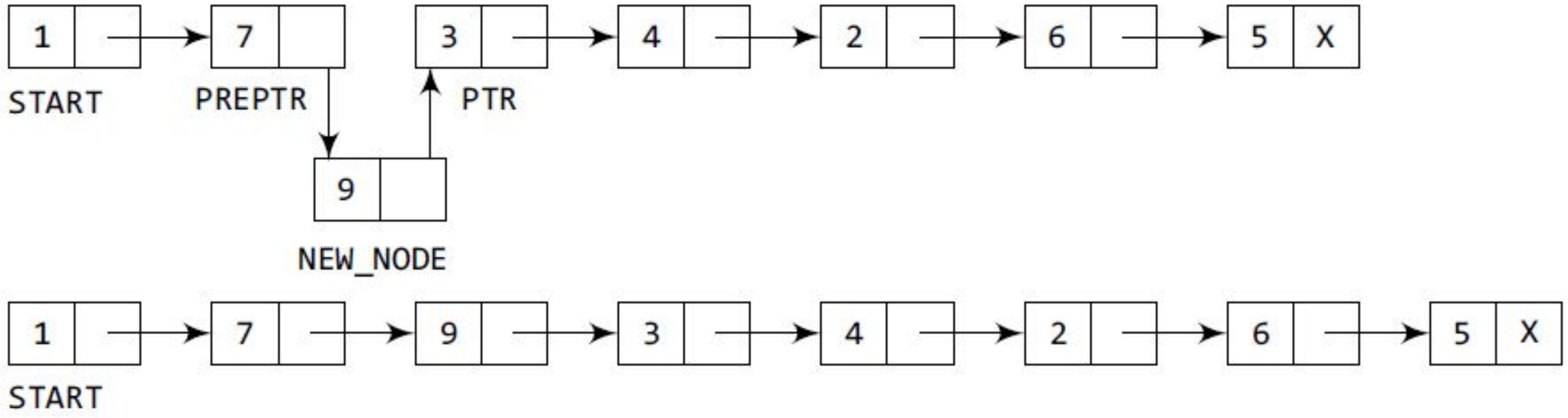


Figure 6.19 Inserting an element before a given node in a linked list

```
void Insert_before_given(struct Node** start_ref, int
search_data, int new_data)
{
   /* 1. allocate node */
   struct Node* new_node = (struct Node*) malloc(sizeof(struct
Node));

   struct Node* ptr = *start_ref;
   struct Node* pre_ptr = ptr;

   int flag=0;

   /* 2. put in the data  */
   new_node->data  = new_data;

   /* 3. Make next of new node as NULL*/
   new_node->next = NULL;

   if (*start_ref == NULL)
   {
     printf("\n The Linked List is empty. The node cannot be
inserted\n");
```

```
while(ptr!=NULL)
{
    if(ptr->data==search_data)
    {
        if(ptr==*start_ref)
        {
            new_node->next=*start_ref;
            *start_ref=new_node;
            flag=1;
            break;
        }
        else
        {
            new_node->next=ptr;
            pre_ptr->next=new_node;
            flag=1;
            break;
        }
    }
    else
    {
        pre_ptr=ptr;
        ptr=ptr->next;
    }
}
```

```
    if(flag==0)
    {
        printf("\n No Node Found.\n");
    }

    if(flag==1)
    {
        printf("\n The node with value %d has been successfully
inserted in the singly link list.\n",new_data);
    }
}
```

DELETIONS IN SINGLY LINKED LIST

Deleting node from LL

Case 1: First node is deleted

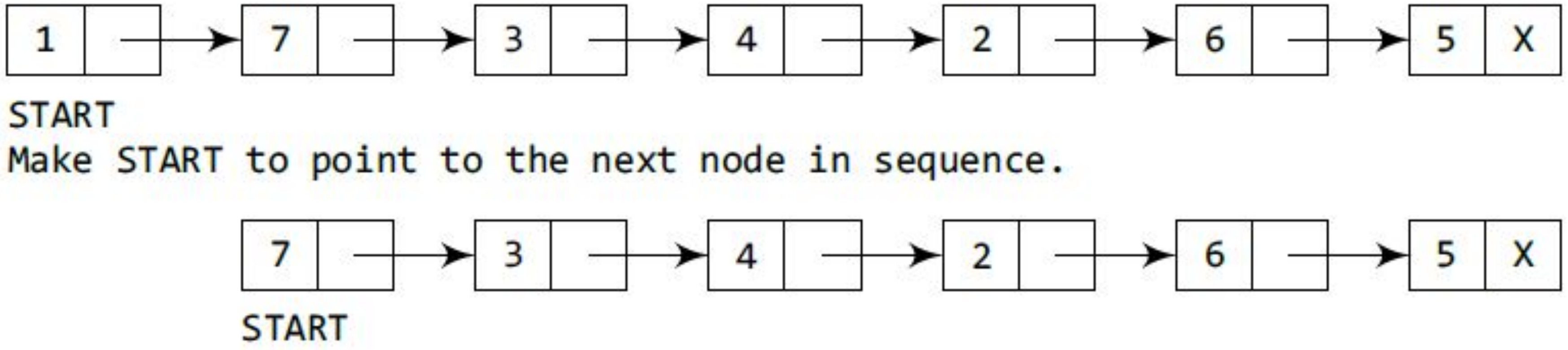
Case 2: Last node is deleted

Case 3: Node after a given node is deleted

Deletion Case 1:

First Node is deleted

Example



Make START to point to the next node in sequence.

Figure 6.20 Deleting the first node of a linked list

Algorithm

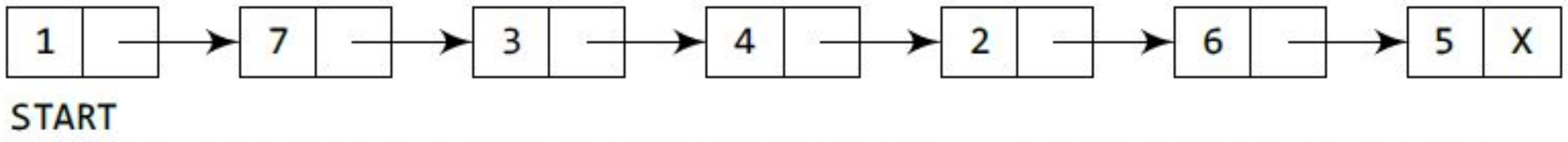
```
Step 1: IF START = NULL then
            Write UNDERFLOW
            GO TO Step 7
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: PTR -> NEXT = NULL
Step 6: PTR = NULL
Step 7: EXIT
```

```
void Delete_front(struct Node **start_ref)
{
        int val;
        struct Node *ptr=*start_ref;

        /* If the Linked List is empty */
        if (*start_ref == NULL)
        {
                printf("\n The Linked List is empty. The node cannot be 
deleted.\n");
                return;
        }

        /* move the head to point to the new node */
        (*start_ref) = ptr->next;

        val=ptr->data;
        ptr->next=NULL;
        free(ptr);   ptr=NULL;

        printf("\n The value deleted from the linked list is: %d\n",val);
}
```

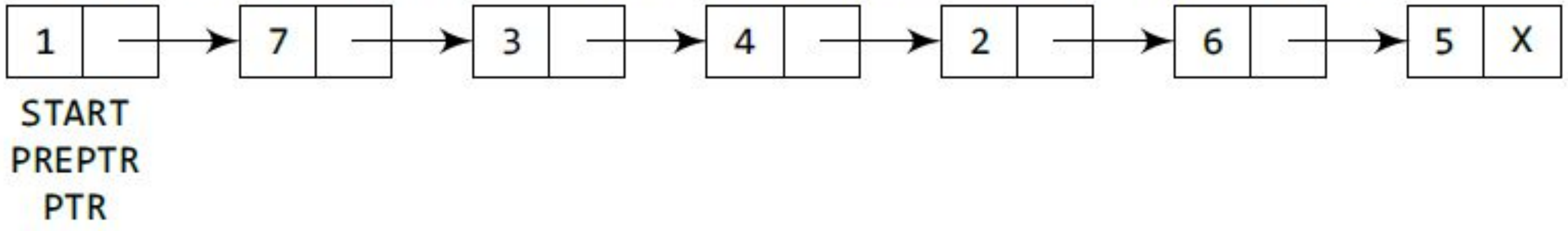
Deletion Case 2:

Last Node is deleted

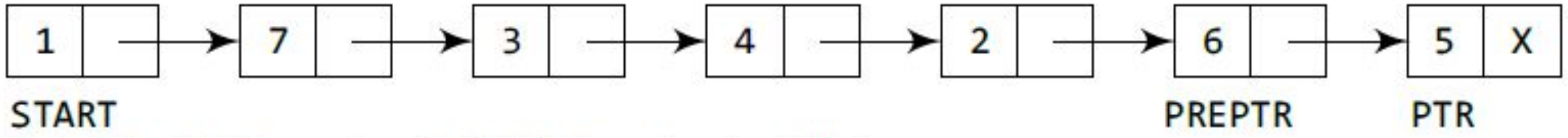
Example



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.

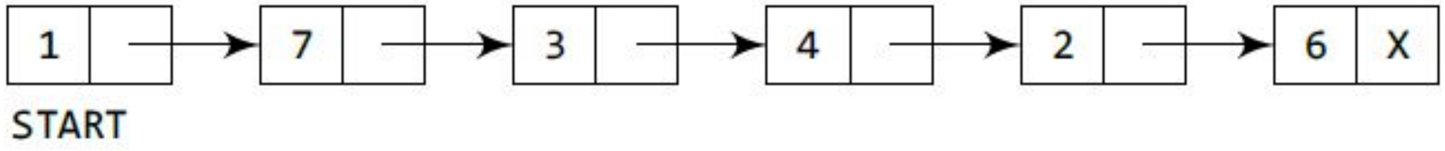


Figure 6.22 Deleting the last node of a linked list

Algorithm

```
Step 1: IF START = NULL then
            Write UNDERFLOW
            GO TO Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 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: PTR = NULL
Step 9: EXIT
```

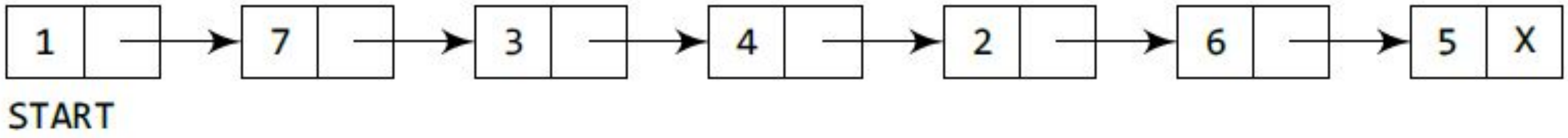
```
void Delete_end(struct Node** start_ref)
{
    int val;
    struct Node *preptr,*ptr = *start_ref;


    /* If the Linked List is empty */
    if (*start_ref == NULL)
    {
        printf("\n The Linked List is empty. The node cannot be
deleted.\n");
        return;
    }

    preptr=ptr;

    /* Traverse till the last node */
    while (ptr->next != NULL)
    {
        preptr=ptr;
        ptr = ptr->next;
    if(preptr==ptr)
    {
        *start_ref = NULL;
        val=ptr->data;
        free(ptr);  ptr=NULL;
        preptr=NULL;
    }
    else
    {
        preptr->next=NULL;
        val=ptr->data;
        ptr->next=NULL;
        free(ptr);   ptr=NULL;
    }

    printf("\n The value deleted from the linked list is: %d\n",val);
}
```

Deletion Case 3:

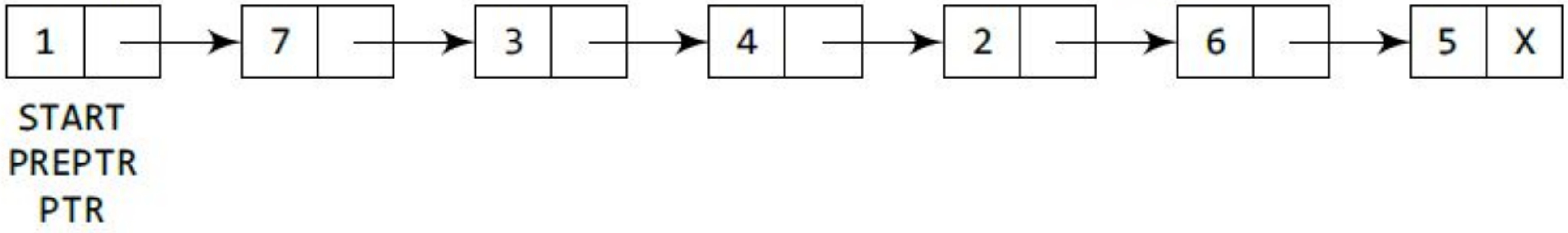
Node after a given node is deleted

Example

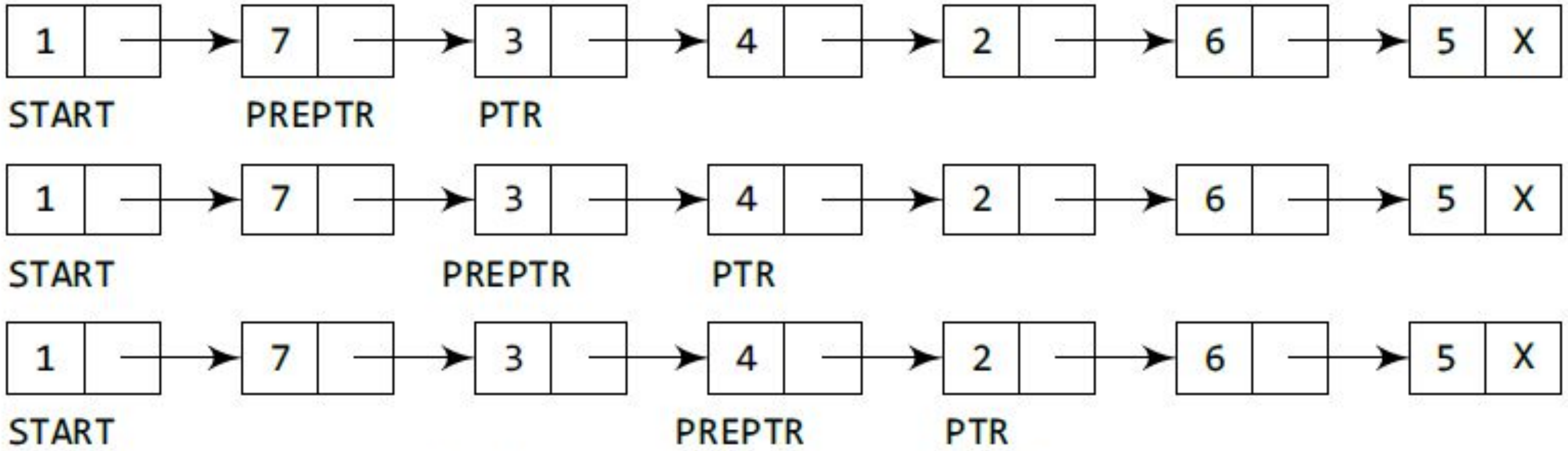
The node is deleted after a given node (After VAL=4)



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.

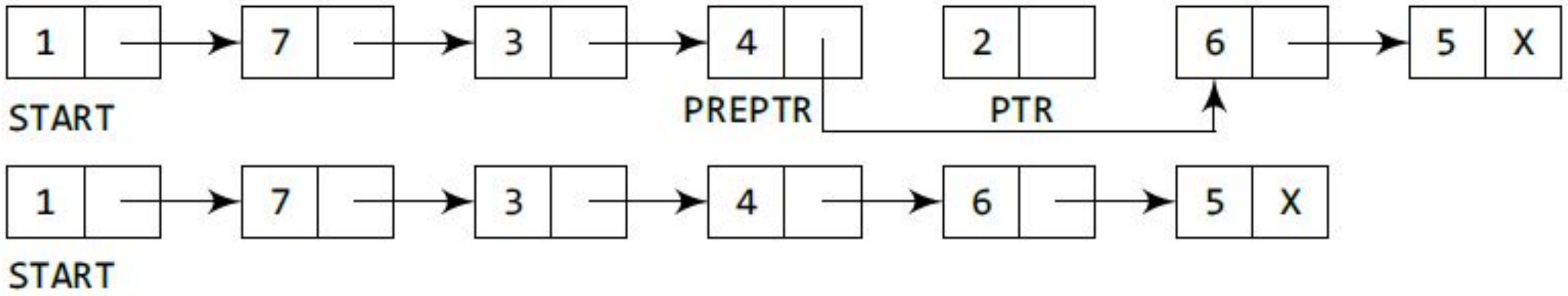


Figure 6.24 Deleting the node after a given node in a linked list

Algorithm

```
Step 1: IF START = NULL then
                    Write UNDERFLOW
                 GO TO Step 11
          [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Step 5 and 6 while PREPTR -> DATA !=
   NUM
Step 5:       SET PREPTR = PTR
Step 6:            SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 7: SET PREPTR -> NEXT = PTR -> NEXT
Step 8: FREE PTR
Step 9: PTR -> NEXT = NULL
Step 10: PTR = NULL
Step 11: EXIT
```

```
void Delete_after_given(struct Node** start_ref, int search_data)
{
    int val, flag=0;
    struct Node *ptr,*preptr = *start_ref;


    /* If the Linked List is empty */
    if (*start_ref == NULL)
    {
        printf("\n The Linked List is empty. The node cannot be
deleted.\n");
        return;
    }

    ptr=preptr->next;
```

```
while(preptr!=NULL)
{
    if(preptr->data==search_data)
    {
        if(ptr!=NULL)
        {
            preptr->next=ptr->next;
            val=ptr->data;
            ptr->next=NULL;
            ptr=NULL;
            flag=1;
            break;
        }
        else // For case if LL has one node whose value matches with the search value.
            break;
    }
    else
    {
        preptr=ptr;
        ptr=ptr->next;
    }
}
```

```
     if(flag==0)
     {
          printf("\n No Node Found.\n");
     }

     if(flag==1)
     {
          printf("\n The value deleted from the linked list is:
%d\n",val);

     }
}
```

Void Main Function in C

```
void main()
{
     /* Start with the empty list */
     struct Node* head = NULL;
     int choice, new_val, search_val;
     clrscr();

   do
   {
     printf("\n\n SINGLY LIKED LIST OPERATIONS");
          printf("\n---------------------------------");
          printf("\n 1.INSERT AT THE FRONT”);
          printf("\n 2.INSERT AT THE END”);
          printf("\n 3.INSERT AFTER GIVEN NODE”);
          printf("\n 4.INSERT BEFORE GIVEN NODE\n");
          printf("\n 5.DELETE THE FIRST NODE”);
          printf("\n 6.DELETE THE LAST NODE”);
          printf("\n 7.DELETE AFTER GIVEN NODE");
          printf("\n 8. DISPLAY\n 9.EXIT");
     printf("\n\n Enter the Choice: ");
     scanf("%d",&choice);
```

```
switch(choice)
{
    case 1:
    {
      printf("\n Enter the value to be inserted: ");
      scanf("%d",&new_val);
      Insert_front(&head,new_val);
      break;
    }
    case 2:
    {
      printf("\n Enter the value to be inserted: ");
      scanf("%d",&new_val);
      Insert_end(&head,new_val);
      break;
    }
```

```
        case 3:
        {
          printf("\n Enter the value to be inserted: ");
          scanf("%d",&new_val);
          printf("\n Enter the value after which the new node is to be
inserted: ");
          scanf("%d",&search_val);
          Insert_after_given(&head,search_val,new_val);
          break;
        }
        case 4:
        {
          printf("\n Enter the value to be inserted: ");
          scanf("%d",&new_val);
          printf("\n Enter the value before which the new node is to be
inserted: ");
          scanf("%d",&search_val);
          Insert_before_given(&head,search_val,new_val);
          break;
        }
        case 5:
        {
          Delete_front(&head);
          break;
        }
        case 6:
        {
          Delete_end(&head);
          break;
        }
        case 7:
        {
          printf("\n Enter the value after which the new node is to be
deleted: ");
          scanf("%d",&search_val);
          Delete_after_given(&head,search_val);
          break;
        }
        case 8:
        {
          display(head);
          break;
        }
        case 9:
        {
          printf("\n EXIT POINT.\n");
          break;
        }
        default:
        {
          printf("\n Please Enter a Valid Choice(1-9).");
        }

    }
    printf("\n Press ENTER ...");
    getch();
  }
  while(choice!=9);
}
```

Void Main Function in C++

```
void main()
{
	linked_list L1;

	L1. insert_begin(10);
	L1. insert_begin(20);
	L1. insert_end(30);
	L1. insert_after_given(20,40);
	L1. insert_before_given(20,50);

	L1. display();

	L1. delete_front();
	L1. delete_end();
	L1. delete_after_given(40);

	L1. display();

}
```