```
    //Inserting at the begining
    if(position == 1){
      newNode->next=p;
      *head=newNode;
    }
    else{
      //Travese the list until the position where we want to insert
      while((p!=NULL) && (k<position)){
        k++;
        q=p;
        p=p->next;
      }
      q->next=newNode;  //more optimum way to do this
      newNode->next=p;
    }
  }
```

Note: We can implement the three variations of the *insert* operation separately.

Time Complexity: O(n). Since, in the worst we may need to insert the node at end of the list. Space Complexity: O(1), for creating one temporary variable.

Singly Linked List Deletion

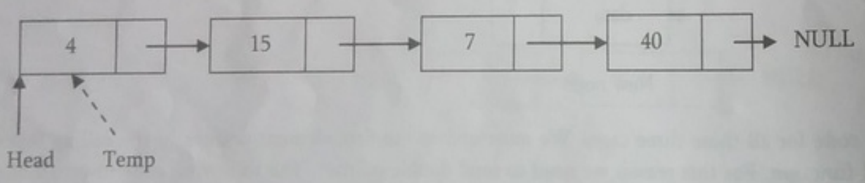
As similar to insertion here also we have three cases.

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node

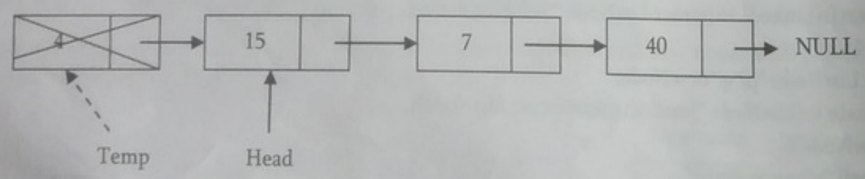
Deleting the First Node in Singly Linked List

First node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to same node as that of head.



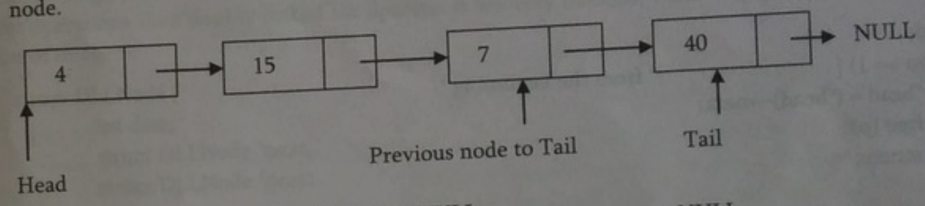
- Now, move the head nodes pointer to the next node and dispose the temporary node.



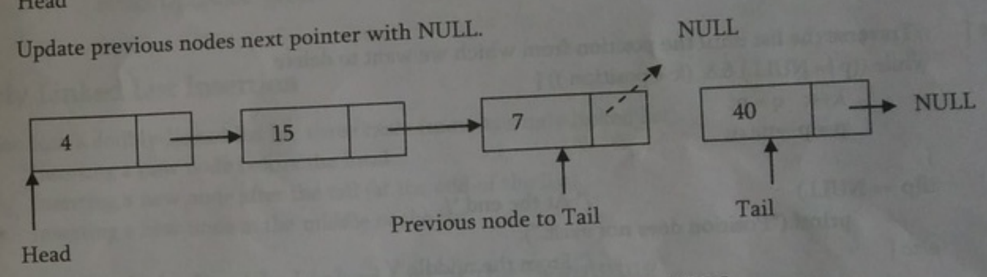
Deleting the last node in Singly Linked List

In this case, last node is removed from the list. This operation is a bit trickier than removing the first node, because algorithm should find a node, which is previous to the tail first. It can be done in three steps:

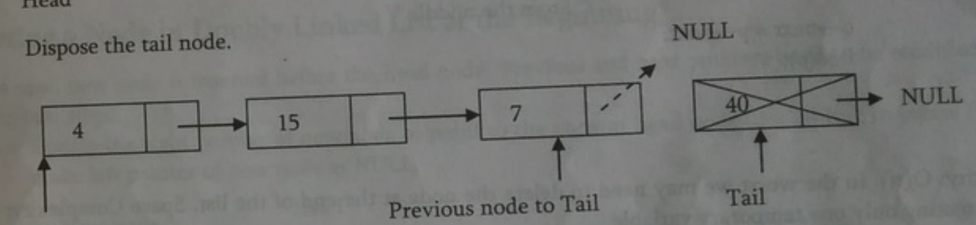
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of list, we will have two pointers one pointing to the *tail* node and other pointing to the node *before* tail node.



- Update previous nodes next pointer with NULL.



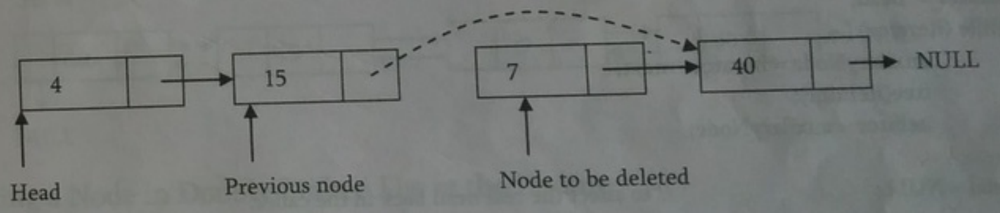
- Dispose the tail node.



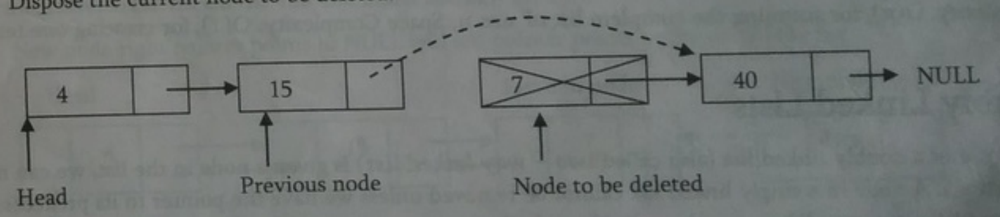
Deleting an Intermediate Node in Singly Linked List

In this case, node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- As similar to previous case, maintain previous node while traversing the list. Once we found the node to be deleted, change the previous nodes next pointer to next pointer of the node to be deleted.



- Dispose the current node to be deleted.




```
void DeleteNodeFromLinkedList (struct ListNode **head, int position) {
      int k = 1;
      struct ListNode *p, *q;
```

```
    if(*head == NULL) {
            printf ("List Empty");
            return;
    }
    p = *head;
    if(position == 1) {                    /* from the beginning */
            *head = (*head)→next;
            free (p);
            return;
    }
    else {    //Traverse the list until the position from which we want to delete
            while ((p != NULL) && (k < position )) {
                    k++;   q = p;
                    p = p→next;
            }
            if(p == NULL)                  /* At the end */
                    printf ("Position does not exist.");
            else {                         /* From the middle */
                    q→next = p→next;
                    free(p);
            }
    }
}
```

Time Complexity: O(n). In the worst we may need to delete the node at the end of the list. Space Complexity: O(1). Since, we are creating only one temporary variable.

Deleting Singly Linked List

This works by storing the current node in some temporary variable and freeing the current node. After freeing the current node go to next node with temporary variable and repeat this process for all nodes.

```
void DeleteLinkedList(struct ListNode **head) {
        struct ListNode *auxilaryNode, *iterator;
        iterator = *head;
        while (iterator) {
                auxilaryNode = iterator→next;
                free(iterator);
                iterator = auxilaryNode;
        }
        *head = NULL;                  // to affect the real head back in the caller.
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

3.7 Doubly Linked Lists

The advantage of a doubly linked list (also called *two – way linked list*) is given a node in the list, we can navigate in both directions. A node in a singly linked list cannot be removed unless we have the pointer to its predecessor. But in doubly linked list we can delete a node even if we don't have previous nodes address (since, each node has left pointer pointing to previous node and can move backward). The primary *disadvantages* of doubly linked lists are:

- Each node requires an extra pointer, requiring more space.

- The insertion or deletion of a node takes a bit longer (more pointer operations).

As similar to singly linked list, let us implement the operations of doubly linked lists. If you understand the singly linked list operations then doubly linked list operations are very obvious. Following is a type declaration for a doubly linked list of integers:

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

Doubly Linked List Insertion

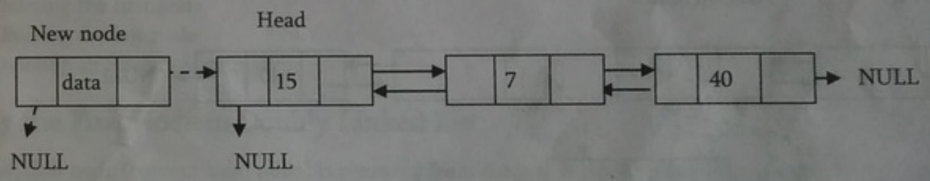
Insertion into a doubly-linked list has three cases (same as singly linked list):

- Inserting a new node before the head.
- Inserting a new node after the tail (at the end of the list).
- Inserting a new node at the middle of the list.

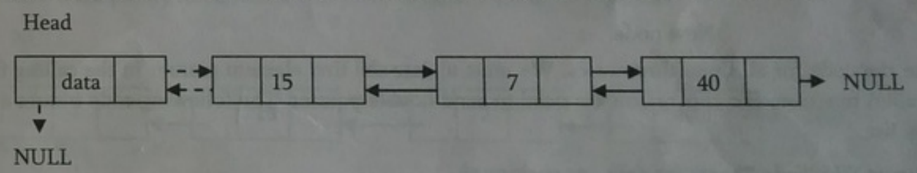
Inserting a Node in Doubly Linked List at the Beginning

In this case, new node is inserted before the head node. Previous and next pointers need to be modified and it can be done in two steps:

- Update the right pointer of new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.



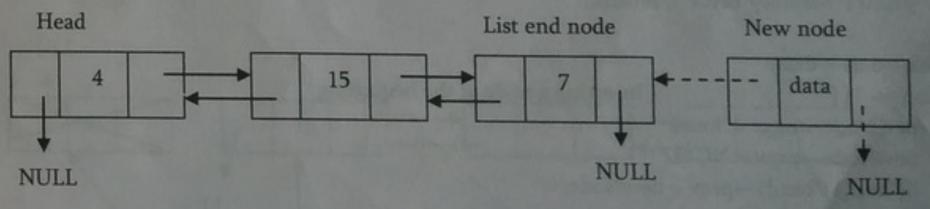
- Update head nodes left pointer to point to the new node and make new node as head.



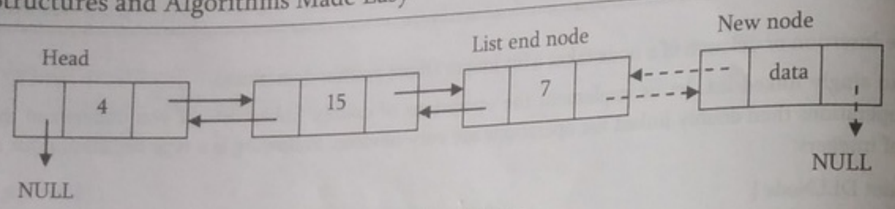
Inserting a Node in Doubly Linked List at the Ending

In this case, traverse the list till the end and insert the new node.

- New node right pointer points to NULL and left pointer points to the end of the list.



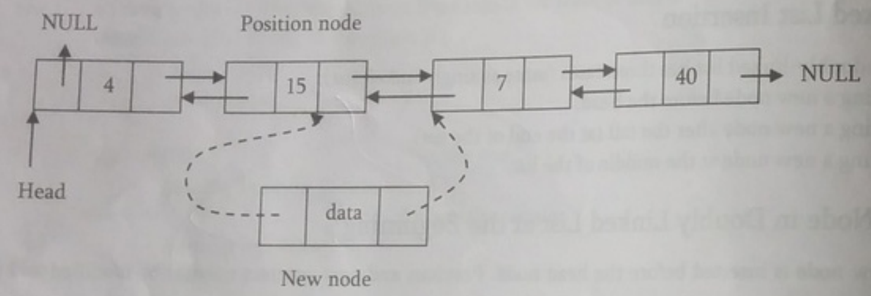
- Update right of pointer of last node to point to new node.



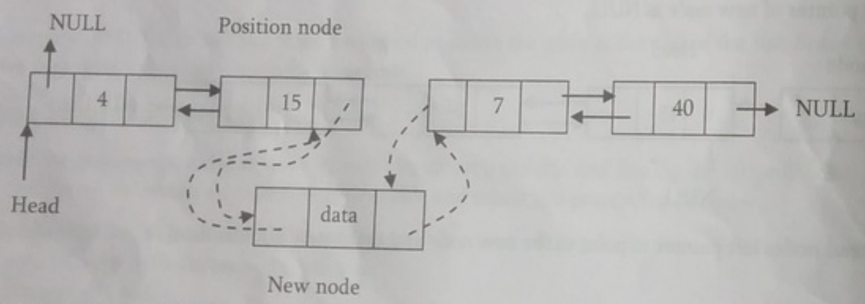
Inserting a Node in Doubly Linked List at the Middle

As discussed in singly linked lists, traverse the list till the position node and insert the new node.

- *New node* right pointer points to the next node of the *position node* where we want to insert the new node. Also, *new node* left pointer points to the *position node*.



- Position node right pointer points to the new node and the *next node* of position nodes left pointer points to new node.



Now, let us write the code for all these three cases. We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send double pointer. The following code inserts a node in the doubly linked list.

```
void DLLInsert(struct DLLNode **head, int data, int position) {
        int k = 1;
        struct DLLNode *temp, *newNode;
        newNode = (struct DLLNode *) malloc(sizeof ( struct DLLNode ));
        if(!newNode) {                                  //Always check for memory errors
                printf ("Memory Error"); return;
        }
        newNode→data = data;
        if(position == 1) {                     //Inserting a node at the beginning
                newNode→next = *head;
                newNode→prev = NULL;
                if(*head) (*head)→prev = newNode;
                *head = newNode;
                return;
```



```
        }
        temp = *head;
        while ( (k < position – 1) && temp→next!=NULL) {
                temp = temp→next;
                k++;
        }
        if(k!=position){
          printf("Desired position does not exist\n");
        }
        newNode->next=temp->next;
        newNode->prev=temp;
        if(temp->next)
          temp->next->prev=newNode;
        temp->next=newNode;
        return;
}
```

Time Complexity: O(n). In the worst we may need to insert the node at the end of the list. Space Complexity: O(1), for creating one temporary variable.

Doubly Linked List Deletion

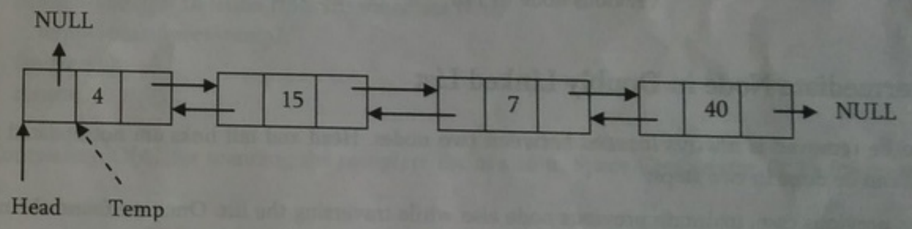
As similar to singly linked list deletion, here also we have three cases:

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node

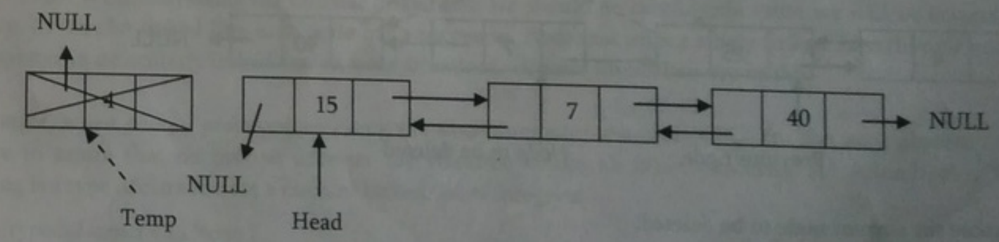
Deleting the First Node in Doubly Linked List

In this case, first node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to same node as that of head.



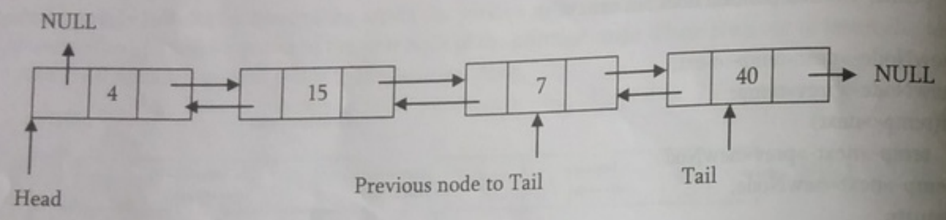
- Now, move the head nodes pointer to the next node and change the heads left pointer to NULL. Then, dispose the temporary node.



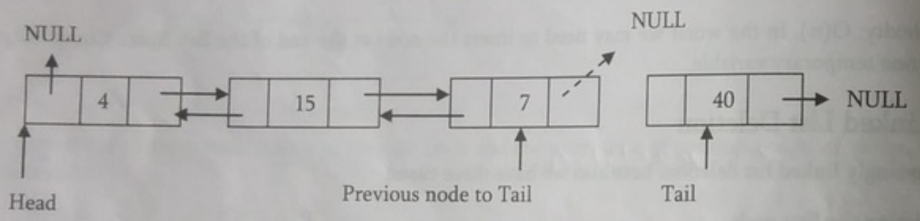
Deleting the Last Node in Doubly Linked List

This operation is a bit trickier, than removing the first node, because algorithm should find a node, which is previous to the tail first. It can be done in three steps:

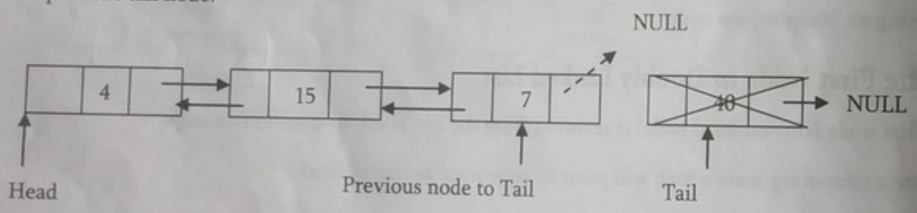
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of list, we will have two pointers one pointing to the tail and other pointing to the node before tail node.



- Update tail nodes previous nodes next pointer with NULL.



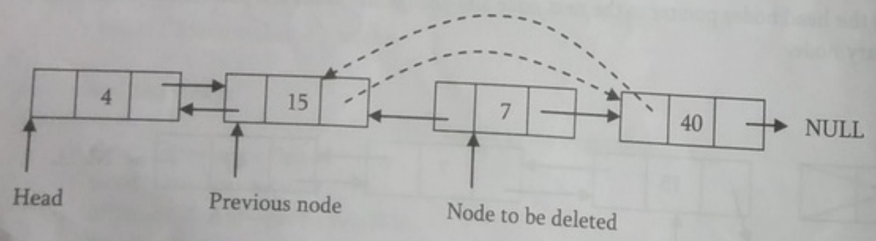
- Dispose the tail node.



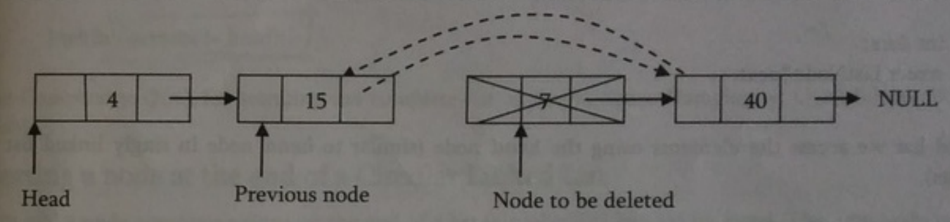
Deleting an Intermediate Node in Doubly Linked List

In this case, node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- As similar to previous case, maintain previous node also while traversing the list. Once we found the node to be deleted, change the previous nodes next pointer to the next node of the node to be deleted.



- Dispose the current node to be deleted.




```
void DLLDelete(struct DLLNode **head, int position) {
        struct DLLNode *temp, *temp2, temp = *head;
        int k = 1;
        if(*head ==  NULL) {
                printf("List is empty");
              return;
        }
        if(position == 1) {
                *head = (*head)→next;
                if(*head != NULL)
                        (*head)→prev = NULL;
                free(temp);
                return;
        }
        while((k < position) && temp→next!=NULL){
                temp = temp→next;
                k++;
        }
        if(k!=position-1){
           printf("Desired position does not exist\n");
        }

        temp2=temp->prev;
        temp2->next=temp->next;
        if(temp->next) // Deletion from Intermediate Node
          temp->next->prev=temp2;
        free(temp);
        return;
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

3.8 Circular Linked Lists

In singly linked lists and doubly linked lists the end of lists are indicated with NULL value. But circular linked lists do not have ends. While traversing the circular linked lists we should be careful otherwise we will be traversing the list infinitely. In circular linked lists each node has a successor. Note that unlike singly linked lists, there is no node with NULL pointer in a circularly linked list. In some situations, circular linked lists are useful.

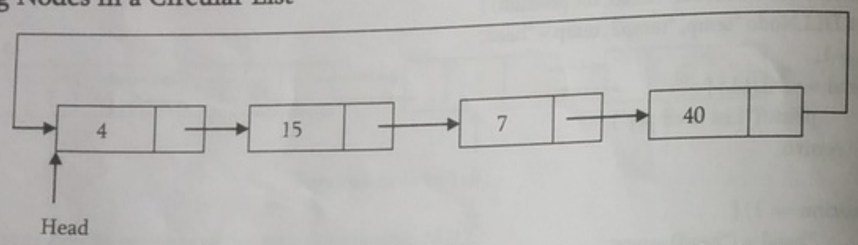
For example, when several processes are using the same computer resource (CPU) for the same amount of time, and we have to assure that no process accesses the resource before all other processes did (round robin algorithm). Following is a type declaration for a circular linked list of integers:

```
typedef struct CLLNode {
```

```
    int data;
    struct ListNode *next;
};
```

In circular linked list we access the elements using the *head* node (similar to *head* node in singly linked list and doubly linked lists).

Counting Nodes in a Circular List



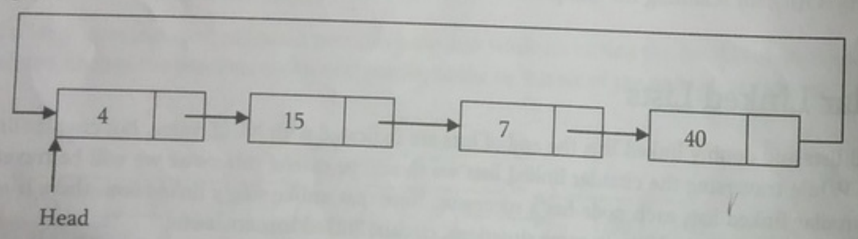
The circular list is accessible through the node marked *head*. To count the nodes, the list has to be traversed from node marked *head*, with the help of a dummy node *current* and stop the counting when *current* reaches the starting node *head*. If the list is empty, *head* will be NULL, and in that case set *count* = 0. Otherwise, set the current pointer to the first node, and keep on counting till the current pointer reaches the starting node.

```
int CircularListLength(struct CLLNode *head) {
    struct CLLNode *current = head;
    int count = 0;
    if(head == NULL) return 0;
    do {    current = current→next;
            count++;
    } while (current != head);
    return count;
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

Printing the contents of a Circular List

We assume here that the list is being accessed by its *head* node. Since all the nodes are arranged in a circular fashion, the *tail* node of the list will be the node previous to the *head* node. Let us assume we want to print the contents of the nodes starting with the *head* node. Print its contents, move to the next node and continue printing till we reach the *head* node again.




```
void PrintCircularListData(struct CLLNode *head) {
    struct CLLNode *current = head;
    if(head == NULL) return;
    do {    printf ("%d", current→data);
            current = current→next;
```



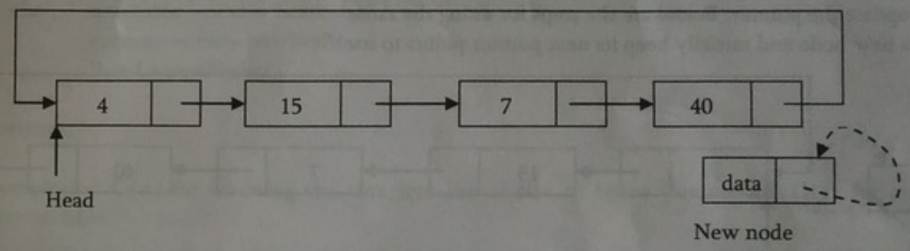
```
    } while (current != head);
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating ne temporary variable.

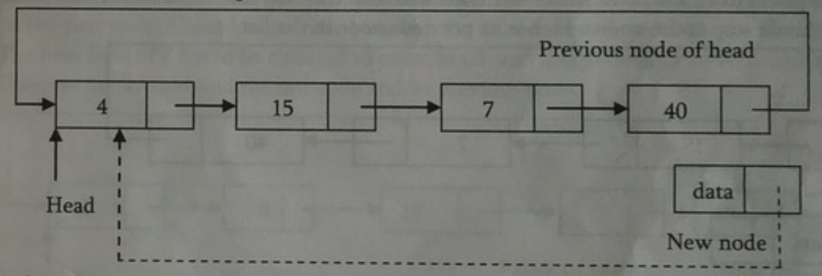
Inserting a node at the end of a Circular Linked List

Let us add a node containing *data*, at the end of a list (circular list) headed by *head*. The new node will be placed just after the tail node (which is the last node of the list), which means it will have to be inserted in between the tail node and the first node.

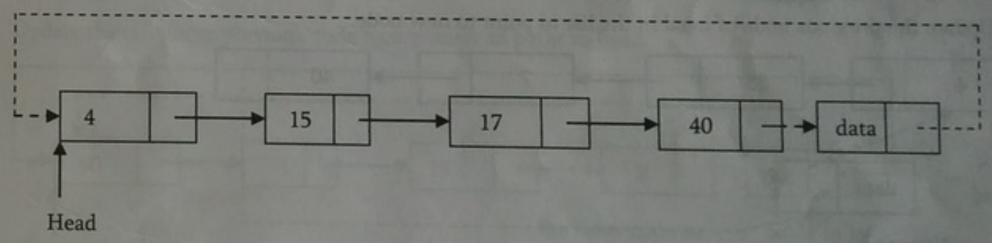
- Create a new node and initially keep its next pointer points to itself.



- Update the next pointer of new node with head node and also traverse the list until the tail. That means in circular list we should stop at a node whose next node is head.



- Update the next pointer of previous node to point to new node and we get the list as shown below.




```
void InsertAtEndInCLL (struct CLLNode **head, int data) {
    struct CLLNode current = *head;
    struct CLLNode *newNode = (struct node*) (malloc(sizeof(struct CLLNode)));
    if(!newNode) {
        printf("Memory Error"); return;
    }
    newNode→data = data;
    while (current→next != *head)
        current = current→next;
    newNode→next = newNode;
    if(*head ==NULL)
```

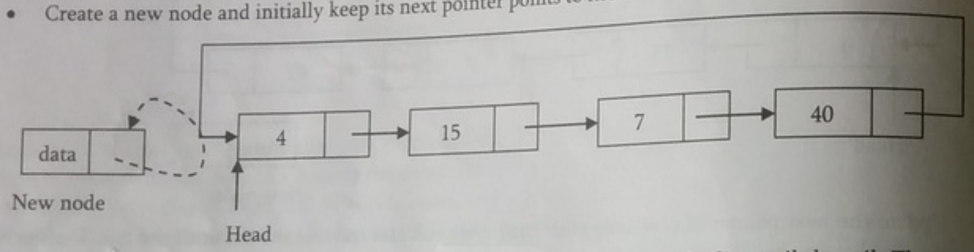
```
            *head = newNode;
    ese {   newNode→next = *head;
            current→next = newNode;
        }
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

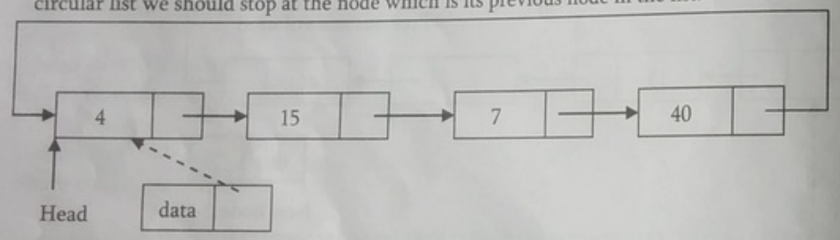
Inserting a node at front of a Circular Linked List

The only difference between inserting a node at the beginning and at the ending is that, after inserting the new node we just need to update the pointer. Below are the steps for doing the same.

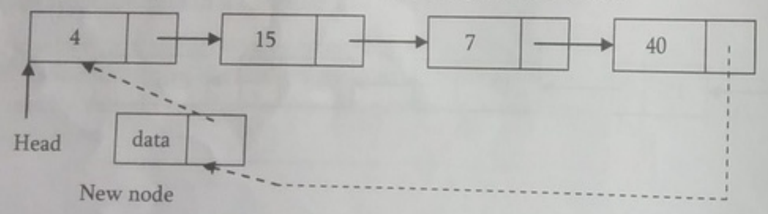
- Create a new node and initially keep its next pointer points to itself.



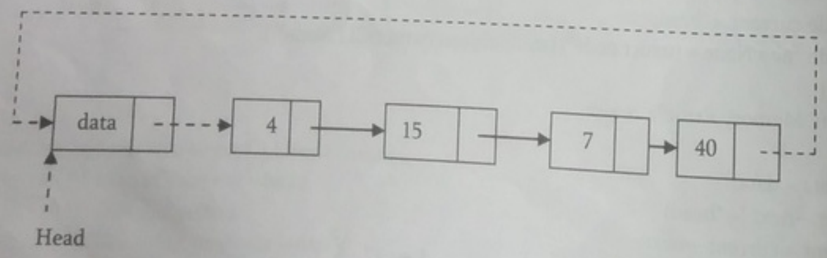
- Update the next pointer of new node with head node and also traverse the list until the tail. That means in circular list we should stop at the node which is its previous node in the list.



- Update the previous node of head in the list to point to new node.



- Make new node as head.




```
void InsertAtBeginInCLL (struct CLLNode **head, int data) {
```



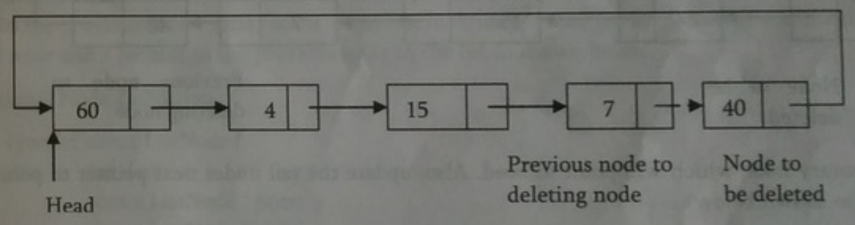
```
    struct CLLNode *current = *head;
    struct CLLNode * newNode = (struct node*) (malloc(sizeof(struct CLLNode)));
    if(!newNode) {
            printf("Memory Error"); return;
    }
    newNode→data = data;
    while (current→next != *head)
            current = current→next;
    newNode→next = newNode;
    if(*head ==NULL)  *head = newNode;
    else {  newNode→next = *head;
            current→next = newNode;
            *head = newNode;
    }
    return;
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating only one temporary variable.

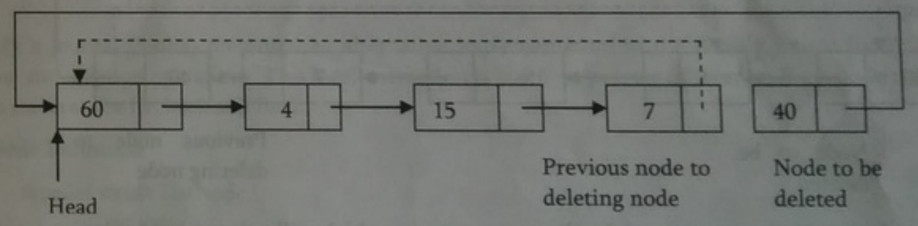
Deleting the last node in a Circular List

The list has to be traversed to reach the last but one node. This has to be named as the tail node, and its next field has to point to the first node. Consider the following list. To delete the last node 40, the list has to be traversed till you reach 7. The next field of 7 has to be changed to point to 60, and this node must be renamed *pTail*.

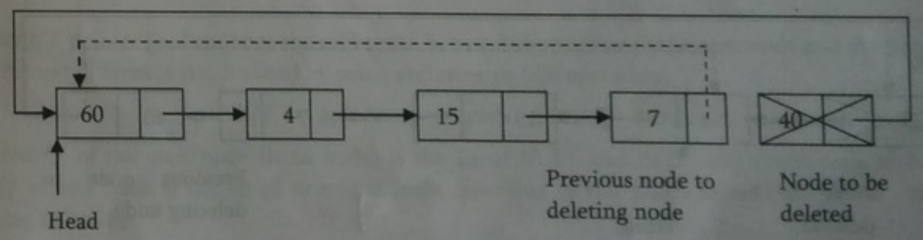
- Traverse the list and find the tail node and its previous node.



- Update the tail nodes previous node next pointer to point to head.



- Dispose the tail node.



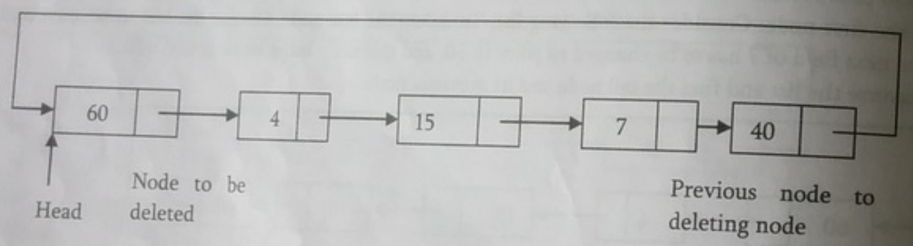
```
void DeleteLastNodeFromCLL (struct CLLNode **head) {
        struct CLLNode *temp = *head, *current = *head;
        if(*head == NULL) {
                printf( "List Empty");
                return;
        }
        while (current→next != *head) {
                temp = current;
                current = current→next;
        }
        temp→next = current->next;
        free(current);
        return;
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

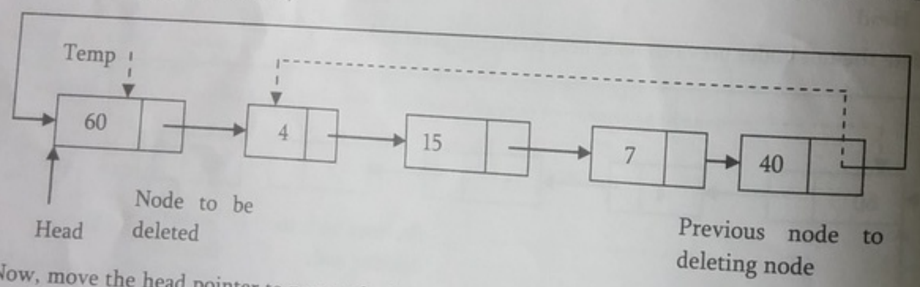
Deleting the first node in a Circular List

The first node can be deleted by simply replacing the next field of tail node with the next field of the first node.

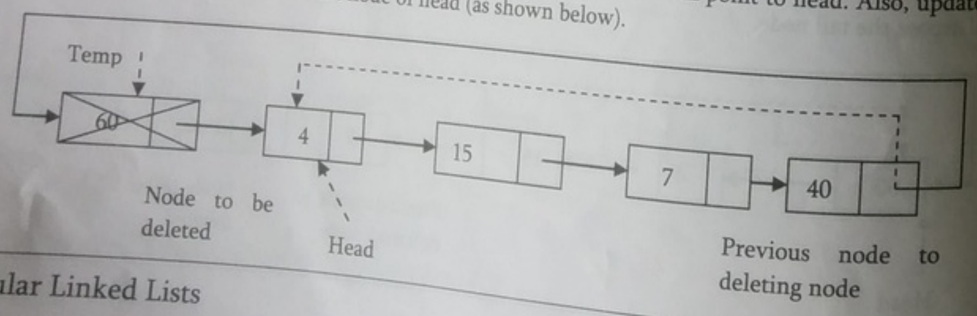
- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.



- Create a temporary node which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



- Now, move the head pointer to next node. Create a temporary which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).




```
void DeleteFrontNodeFromCLL (struct CLLNode **head) {
        struct CLLNode *temp = *head;
        struct CLLNode *current = *head;
        if(*head == NULL) {
                printf("List Empty");
                return;
        }
        while (current→next != *head)
                current = current→next;
        current→ next = *head→next;
        *head = *head→next;
        free(temp);
        return;
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

Applications of Circular List

Circular linked lists are used in managing the computing resources of a computer. We can use circular lists for implementing stacks and queues.

3.9 A Memory-Efficient Doubly Linked List

In conventional implementation, we need to keep a forward pointer to the next item on the list and a backward pointer to the previous item. That means, elements in doubly linked list implementations consists of data, a pointer to the next node and a pointer to the previous node in the list as shown below.

Conventional Node Definition

```
typedef struct ListNode {
        int data;
        struct ListNode * prev;
        struct ListNode * next;
};
```

Recently a journal (Sinha) presented an alternative implementation of the doubly linked list ADT, with insertion, traversal and deletion operations. This implementation is based on pointer difference. Each node uses only one pointer field to traverse the list back and forth.

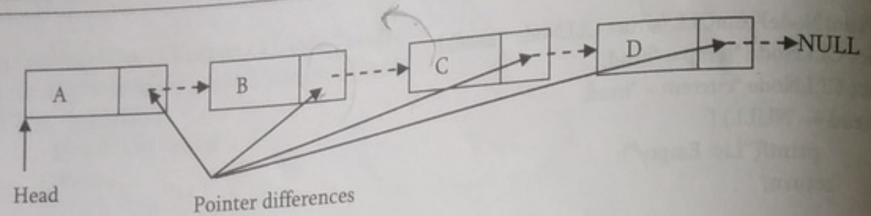
New Node Definition

```
typedef struct ListNode {
        int data;
        struct ListNode * ptrdiff;
};
```

The $ptrdiff$ pointer field contains the difference between the pointer to the next node and the pointer to the previous node. Pointer difference is calculated by using exclusive-or ($\oplus$) operation.

$$ptrdiff = pointer\ to\ previous\ node \oplus pointer\ to\ next\ node.$$

The $ptrdiff$ of the start node (head node) is the $\oplus$ of NULL and $next$ node (next node to head). Similarly, the $ptrdiff$ of end node is the $\oplus$ of $previous$ node (previous to end node) and NULL. As an example, consider the following linked list.



In the above example,

- The next pointer of A is: NULL ⊕ B
- The next pointer of B is: A ⊕ C
- The next pointer of C is: B ⊕ D
- The next pointer of D is: C ⊕ NULL

Why does it work?

To have answer for this question let us consider the properties of ⊕:

X ⊕ X = 0
X ⊕ 0 = X
X ⊕ Y = Y ⊕ X (symmetric)
(X ⊕ Y) ⊕ Z = X ⊕ (Y ⊕ Z) (transitive)

For the above example, let us assume that we are at C node and want to move to B. We know that Cs *ptrdiff* is defined as B ⊕ D. If we want to move to B, performing ⊕ on Cs *ptrdiff* with D would give B. This is due to fact that,

(B ⊕ D) ⊕ D = B (since, D ⊕ D=0)

Similarly, if we want to move to D, then we have to applying ⊕ to Cs *ptrdiff* with B would give D.

(B ⊕ D) ⊕ B = D (since, B ⊕ B=0)

From the above discussion we can see that just by using single pointer, we are able to move back and forth. A memory-efficient implementation of a doubly linked list is possible to have without compromising much timing efficiency.

3.10 Problems on Linked Lists

Problem-1 Implement Stack using Linked List.

Solution: Refer *Stacks* chapter.

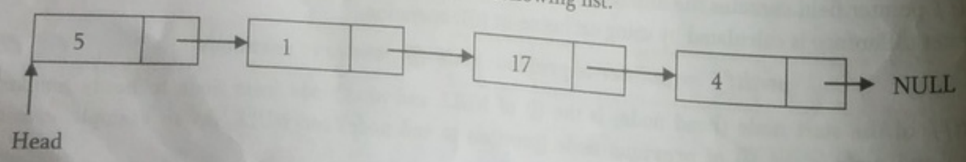
Problem-2 Find n^{th} node from the end of a Linked List.

Solution: Brute-Force Method: Start with the first node and count how many nodes are there after that node. If the number of nodes are $< n-1$ then return saying "fewer number of nodes in the list". If the number of nodes are $> n-1$ then go to next node. Continue this until the numbers of nodes after current node are $n-1$.

Time Complexity: $O(n^2)$, for scanning the remaining list (from current node) for each node. Space Complexity: $O(1)$.

Problem-3 Can we improve the complexity of Problem-2?

Solution: Yes, using hash table. As an example consider the following list.



In this approach, create a hash table whose entries are < *position of node, node address* >. That means, key is the position of the node in the list and value is the address of that node.

Position in List	Address of Node
1	Address of 5 node
2	Address of 1 node
3	Address of 17 node
4	Address of 4 node

By the time we traverse the complete list (for creating hash table), we can find the list length. Let us say, the list length is M. To find n^{th} from end of linked list, we can convert this to $M-n+1^{th}$ from the beginning. Since we already know the length of the list, it is just a matter of returning $M-n+1^{th}$ key value from the hash table.

Time Complexity: Time for creating the hash table, $T(m) = O(m)$. Space Complexity: Since, we need to create a hash table of size m, $O(m)$.

Problem-4 Can we use Problem-3 approach for solving Problem-2 without creating the hash table?

Solution: Yes. If we observe the Problem-3 solution, what actually we are doing is finding the size of the linked list. That means, we are using hash table to find the size of the linked list. We can find the length of the linked list just by starting at the head node and traversing the list. So, we can find the length of the list without creating the hash table. After finding the length, compute $M-n+1$ and with one more scan we can get the $M-n+1^{th}$ node from the beginning. This solution needs two scans: one for finding the length of list and other for finding $M-n+1^{th}$ node from the beginning.

Time Complexity: Time for finding the length + Time for finding the $M-n+1^{th}$ node from the beginning. Therefore, $T(n = O(n) + O(n) \approx O(n)$. Space Complexity: $O(1)$. Since, no need of creating the hash table.

Problem-5 Can we solve Problem-2 in one scan?

Solution: Yes. Efficient Approach: Use two pointers *pNthNode* and *pTemp*. Initially, both points to head node of the list. *pNthNode* starts moving only after *pTemp* made n moves. From there both moves forward until *pTemp* reaches end of the list. As a result *pNthNode* points to n^{th} node from end of the linked list.

Note: at any point of time both moves one node at time.

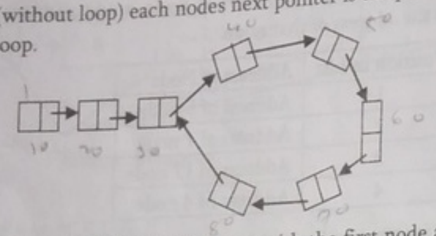
```
struct ListNode *NthNodeFromEnd(struct ListNode *head , int NthNode) {
        struct ListNode *pTemp = NULL, *pNthNode = head;
        int count = 0;
        for (pTemp = head; pTemp!= NULL;) {
                count++;
                if(NthNode – count > 0)
                        pNthNode = pNthNode→pNext;
                pTemp = pTemp→pNext;
        }
        if(count >= NthNode)
                return pNthNode;
        return NULL;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Problem-6 Check whether the given linked list is either NULL-terminated or ends in a cycle (cyclic).

Solution: Brute-Force Approach. As an example consider the following linked list which has a loop in it. The difference between this list and regular list is that, in this list there are two nodes whose next pointers are same. In

regular singly linked lists (without loop) each nodes next pointer is unique. That means, the repetition of next pointers indicates the existence of loop.



One simple and brute force way of solving this is, start with the first node and see whether there is any node whose next pointer is current nodes address. If there is a node with same address then that indicates that some other node is pointing to the current node and we can say loops exists. Continue this process for all the nodes of the linked list.

Does this method works? As per the algorithm we are checking for the next pointer addresses, but how do we find the end of the linked list (otherwise we will end up in infinite loop)?

Note: If we start with a node in loop, this method may work depending on the size of the loop.

Problem-7 Can we use hashing technique for solving Problem-6?

Solution: Yes. Using Hash Tables we can solve this problem.
Algorithm:

- Traverse the linked list nodes one by one.
- Check if the nodes address is there in the hash table or not.
- If it is already there in the hash table then that indicates that we are visiting the node which was already visited. This is possible only if the given linked list has a loop in it.
- If the address of the node is not there in the hash table then insert that nodes address into the hash table.
- Continue this process until we reach end of the linked list *or* we find loop.

Time Complexity: $O(n)$ for scanning the linked list. Note that we are doing only scan of the input. Space Complexity: $O(n)$ for hash table.

Problem-8 Can we solve the Problem-6 using sorting technique?

Solution: No. Consider the following algorithm which is based on sorting. And then, we see why this algorithm fails.
Algorithm:

- Traverse the linked list nodes one by one and take all the next pointer values into some array.
- Sort the array which is having next node pointers.
- If there is a loop in the linked list, definitely two nodes next pointers will pointing to the same node.
- After sorting if there is a loop in the list, the nodes whose next pointers are same will come adjacent in the sorted list.
- If there is any such pair exists in the sorted list then we say the linked list has loop in it.

Time Complexity: $O(nlogn)$ for sorting the next pointers array. Space Complexity: $O(n)$ for the next pointers array.

Problem with above algorithm? The above algorithm works only if we can find the length of the list. But if the list is having loop then we may end up in infinite loop. Due to this reason the algorithm fails.

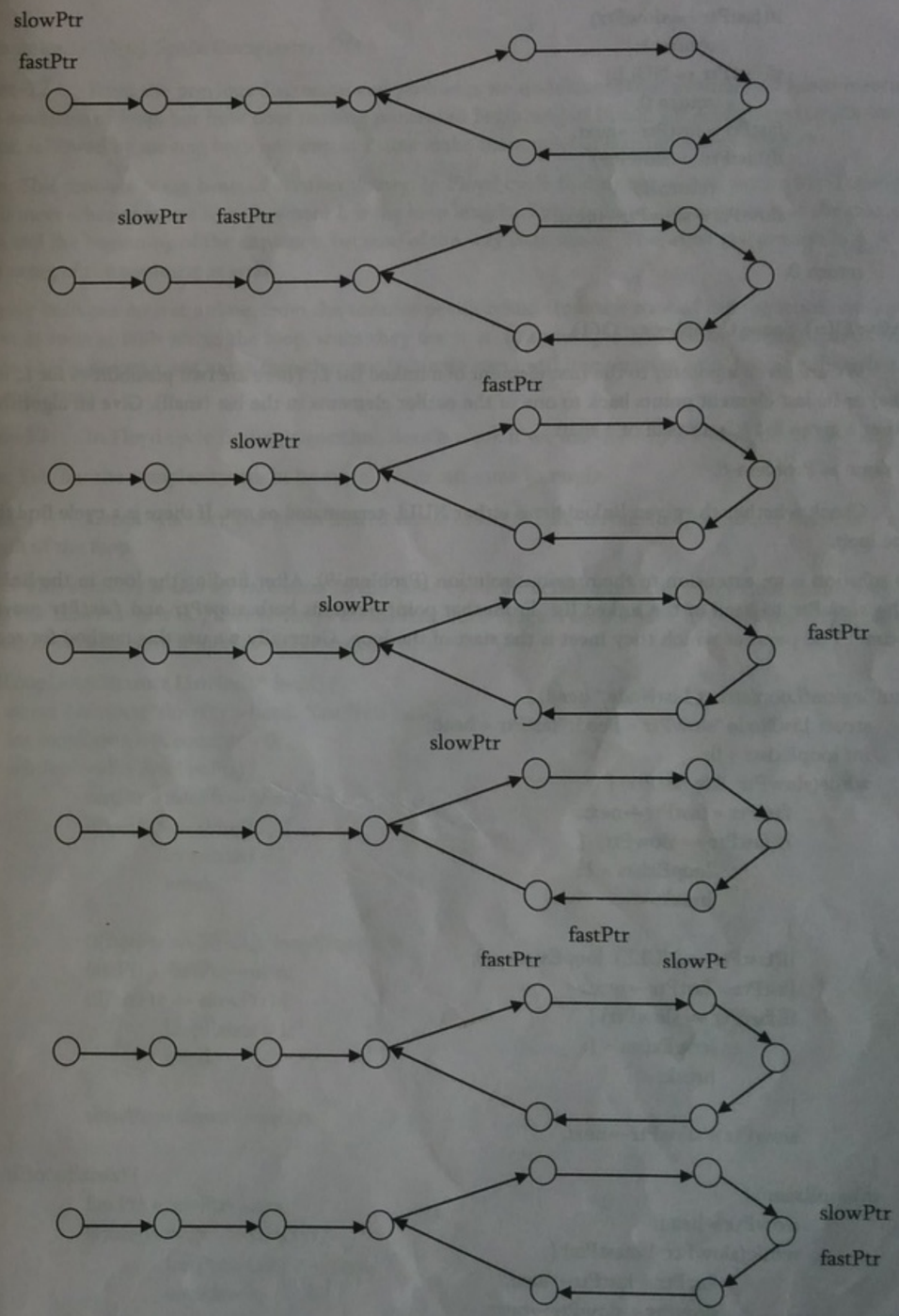
Problem-9 Can we solve the Problem-6 in $O(n)$?

Solution: Yes. **Efficient Approach (Memory less Approach)**: This problem was solved by *Floyd*. The solution is named as Floyd cycle finding algorithm. It uses 2 pointers moving at different speeds to walk the linked list. Once they enter the loop they are expected to meet, which denotes that there is a loop. This works because the only way a faster moving pointer would point to the same location as a slower moving pointer is, if somehow the entire list or a part of it is circular.

Think of a tortoise and a hare running on a track. The faster running hare will catch up with the tortoise if they are running in a loop. As an example, consider the following example and trace out the Floyd algorithm. From the below

diagrams we can see that after the final step they are meeting at some point in the loop which may not be the starting of the loop.

Note: *slowPtr* (*tortoise*) moves one pointer at a time and *fastPtr* (*hare*) moves two pointers at a time.



```
int IsLinkedListContainsLoop(struct ListNode * head) {
        struct ListNode *slowPtr = head, *fastPtr = head;
        while(slowPtr  && fastPtr) {
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr)
                        return 1;
                if(fastPtr == NULL)
                        return 0;
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr)
                        return 1;
                slowPtr = slowPtr→next;
        }
        return 0;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-10 We are given a pointer to the first element of a linked list L. There are two possibilities for L, it either ends (snake) or its last element points back to one of the earlier elements in the list (snail). Give an algorithm that tests whether a given list L is a snake or a snail.

Solution: It is same as Problem-6.

Problem-11 Check whether the given linked list is either NULL-terminated or not. If there is a cycle find the start node of the loop.

Solution: The solution is an extension to the previous solution (Problem-9). After finding the loop in the linked list, we initialize the *slowPtr* to head of the linked list. From that point onwards both *slowPtr* and *fastPtr* moves only one node at a time. The point at which they meet is the start of the loop. Generally we use this method for removing the loops.

```
int FindBeginofLoop(struct ListNode * head) {
        struct ListNode *slowPtr = head, *fastPtr = head;
        int loopExists = 0;
        while(slowPtr  && fastPtr) {
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr)  {
                        loopExists = 1;
                        break;
                }
                if(fastPtr == NULL)  loopExists = 0;
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr) {
                        loopExists = 1;
                        break;
                }
                slowPtr = slowPtr→next;
        }
        if(loopExists) {
                slowPtr = head;
                while(slowPtr  != fastPtr) {
                        fastPtr = fastPtr→next;
                        slowPtr = slowPtr→next;
```



```
                }
                return slowPtr;
        }
        return NULL;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-12 From the previous discussion and problems we understand that tortoise and hares meeting concludes the existence of loop, but how does moving tortoise to beginning of linked list while keeping the hare at meeting place, followed by moving both one step at a time make them meet at starting point of cycle?

Solution: This problem is the heart of number theory. In Floyd cycle finding algorithm, notice that the tortoise and the hare will meet when they are $n \times L$, where L is the loop length. Furthermore, the tortoise is at the midpoint between the hare and the beginning of the sequence, because of the way they move. Therefore the tortoise is $n \times L$ away from the beginning of the sequence as well.

If we move both one step at a time, from the tortoise position and from the start of the sequence, we know that they will meet as soon as both are in the loop, since they are $n \times L$, a multiple of the loop length, apart. One of them is already in the loop, so we just move the other one in single step until it enters the loop, keeping the other $n \times L$ away from it at all times.

Problem-13 In Floyd cycle finding algorithm, does it work if we use the step 2 and 3 instead of 1 and 2?

Solution: Yes, but the complexity might be more. Trace out some example.

Problem-14 Check whether the given linked list is either NULL-terminated or not. If there is a cycle find the length of the loop.

Solution: This solution is also an extension to the basic cycle detection problem. After finding the loop in the linked list, keep the *slowPtr* as it is. *fastPtr* keeps on moving until it again comes back to *slowPtr*. While moving *fastPtr*, use a counter variable which increments at the rate of 1.

```
int FindLoopLength(struct ListNode * head) {
        struct ListNode *slowPtr = head, *fastPtr = head;
        int loopExists = 0, counter = 0;
        while(slowPtr && fastPtr) {
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr)  {
                        loopExists = 1;
                        break;
                }
                if(fastPtr == NULL)  loopExists = 0;
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr) {
                        loopExists = 1;
                        break;
                }
                slowPtr = slowPtr→next;
        }
        if(loopExists) {
                fastPtr = fastPtr→next;
                while(slowPtr  != fastPtr) {
                        fastPtr = fastPtr→next;
                        counter++;
```

```
        }
        return counter;
    }
    return 0;                                   //If no loops exists
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-15 Insert a node in a sorted linked list

Solution: Traverse the list and find a position for the element and insert it.

```
struct ListNode *InsertInSortedList(struct ListNode * head, struct ListNode * newNode) {
    struct ListNode *current = head, temp;
    if(!head)
        return newNode;
    // traverse the list until you find item bigger the new node value
    while (current != NULL && current→data < newNode→data){
        temp = current;
        current = current→next;
    }
    //insert the new node before the big item
    newNode→next = current;
    temp→next = newNode;
    return head;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-16 Reverse a singly linked list

Solution: // Iterative version

```
struct ListNode *ReverseList(struct ListNode *head) {
    struct ListNode *temp = NULL, *nextNode = NULL;
    while (head) {
        nextNode = head→next;
        head→next = temp;
        temp = head;
        head = nextNode;
    }
    return temp;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Recursive version: We can find it easier to start from the bottom up, by asking and answering tiny questions (this is the approach in The Little Lisper):

- What is the reverse of NULL (the empty list)? NULL.
- What is the reverse of a one element list? The element itself.
- What is the reverse of an n element list? The reverse of the second element on followed by the first element.

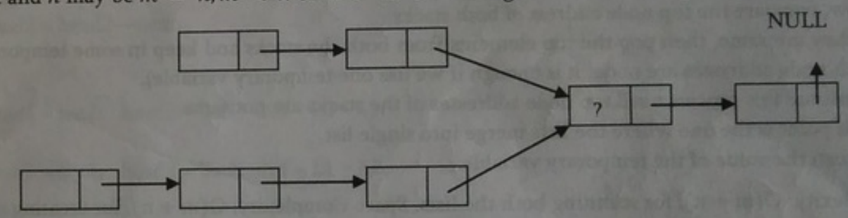
```
struct ListNode * RecursiveReverse(struct ListNode *head) {
  if (head == NULL)
      return NULL;
  if (head→next == NULL)
      return list;
```



```
  ListNode secondElem = head→next;
  head→next = NULL;              // Need to unlink list from the rest or you will get a cycle
  struct ListNode *reverseRest = RecursiveReverse(secondElem);   // reverse everything from the second element on
  secondElem→next = head;        // then we join the two lists
  return reverseRest;
}
```

Time Complexity: O(n). Space Complexity: O(n), for recursive stack.

Problem-17 Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the list before they intersect are unknown and both list may have it different. $List1$ may have n nodes before it reaches intersection point and $List2$ might have m nodes before it reaches intersection point where m and n may be $m = n, m < n$ or $m > n$. Give an algorithm for fidning the merging point.



Solution: Brute-Force Approach: One easy solution is to compare every node pointer in the first list with every other node pointer in the second list by which the matching node pointers will lead us to the intersecting node. But, the time complexity in this case will O(mn) which will be high.

Time Complexity: O(mn). Space Complexity: O(1).

Problem-18 Can we solve Problem-17 using sorting technique?

Solution: No. Consider the following algorithm which is based on sorting and see why this algorithm fails.

Algorithm

- Take first list node pointers and keep in some array and sort them.
- Take second list node pointers and keep in some array and sort them.
- After sorting, use two indexes: one for first sorted array and other for second sorted array.
- Start comparing values at the indexes and increment the index whichever is having lower value (increment only if the values are not equal).
- At any point, if we were able to find two indexes whose values are same then that indicates that those two nodes are pointing to the same node and we return that node.

Time Complexity: Time for sorting lists + Time for scanning (for comparing)=O($mlogm$) +O($nlogn$) +O($m + n$). We need to consider the one which gives the maximum value. Space Complexity: O(1).

Problem with the above algorithm? Yes. In the algorithm, we are storing all the node pointers of both the lists and sorting. But we are forgetting the fact that, there can be many repeated elements. This is because after the merging point all node pointers are same for both the lists. The algorithm works fine only in one case and it is when both lists have ending node at their merge point.

Problem-19 Can we solve Problem-17 using hash tables?

Solution: Yes.

Algorithm:

- Select a list which is having less number of nodes (If we do not know the lengths beforehand then select one list randomly).
- Now, traverse the other list and for each node pointer of this list check whether the same node pointer exists in the hash table or not.

- If there a merge point for the give lists then we will definitely encounter the node pointer in the hash table.

Time Complexity: Time for creating the hash table + Time for scanning the second list = $O(m) + O(n)$ (or $O(n) + O(m)$, depends on which list we select for creating the hash table). But in both cases the time complexity is same. Space Complexity: $O(n)$ or $O(m)$.

Problem-20 Can we use stacks for solving the Problem-17?

Solution: Yes.
Algorithm:

- Create two stacks: one for the first list and one for the second list.
- Traverse the first list and push all the node address on to the first stack.
- Traverse the second list and push all the node address on to the second stack.
- Now both stacks contain the node address of the corresponding lists.
- Now, compare the top node address of both stacks.
- If they are same, then pop the top elements from both the stacks and keep in some temporary variable (since both node addresses are node, it is enough if we use one temporary variable).
- Continue this process until top node addresses of the stacks are not same.
- This point is the one where the lists merge into single list.
- Return the value of the temporary variable.

Time Complexity: $O(m + n)$, for scanning both the lists. Space Complexity: $O(m + n)$, for creating two stacks for both the lists.

Problem-21 Is there any other way of solving the Problem-17?

Solution: Yes. Using "finding the first repeating number" approach in an array (for algorithm refer *Searching* chapter).

Algorithm:

- Create an array A and keep all the next pointers of both the lists in the array.
- In the array find the first repeating element in the array [Refer *Searching* chapter for algorithm].
- The first repeating number indicates the merging point of the both lists.

Time Complexity: $O(m + n)$. Space Complexity: $O(m + n)$.

Problem-22 Can we still think for finding alternative solution for the Problem-17?

Solution: Yes. By combining sorting and search techniques we can reduce the complexity.

Algorithm:

- Create an array A and keep all the next pointers of the first list in the array.
- Sort these array elements.
- Then, for each of the second list element, search in the sorted array (let us assume that we are using binary search which gives $O(logn)$).
- Since we are scanning the second list one by one, the first repeating element which appears in the array is nothing but the merging point.

Time Complexity: Time for sorting + Time for searching = $O(Max(mlogm, nlogn))$. Space Complexity: $O(Max(m, n))$.

Problem-23 Can we improve the complexity for the Problem-17?

Solution: Yes.
Efficient Approach:

- Find lengths (L1 and L2) of both list -- $O(n) + O(m) = O(max(m, n))$.
- Take the difference d of the lengths -- $O(1)$.
- Make d steps in longer list -- $O(d)$.

- Step in both lists in parallel until links to next node match -- $O(min(m, n))$.
- Total time complexity = $O(max(m, n))$.
- Space Complexity = $O(1)$.

```
struct ListNode* FindIntersectingNode(struct ListNode* list1, struct ListNode* list2) {
    int L1=0, L2=0, diff=0;
    struct ListNode *head1 = list1, *head2 = list2;
    while(head1!= NULL) {
            L1++;
            head1 = head1→next;
    }
    while(head2!= NULL) {
            L2++;
            head2 = head2→next;
    }
    if(L1 < L2) {
            head1 = list2;  head2 = list1;  diff = L2 - L1;
    }else{
            head1 = list1;  head2 = list2;  diff = L1 – L2;
    }
    for(int i = 0; i < diff; i++)
            head1 = head1→next;
    while(head1 != NULL && head2 != NULL) {
            if(head1 == head2)
                    return head1→data;
            head1= head1→next;
            head2= head2→next;
    }
    return NULL;
}
```

Problem-24 How will you find the middle of the linked list?

Solution: Brute-Force Approach: For each of the node count how many nodes are there in the list and see whether it is the middle.

Time Complexity: $O(n^2)$. Space Complexity: $O(1)$.

Problem-25 Can we improve the complexity of Problem-24?

Solution: Yes.
Algorithm:

- Traverse the list and find the length of the list.
- After finding the length, again scan the list and locate $n/2$ node from the beginning.

Time Complexity: Time for finding the length of the list + Time for locating middle node = $O(n) + O(n) \approx O(n)$. Space Complexity: $O(1)$.

Problem-26 Can we use hash table for solving Problem-24?

Solution: Yes. The reasoning is same as that of Problem-3.

Time Complexity: Time for creating the hash table. Therefore, $T(n) = O(n)$. Space Complexity: $O(n)$. Since, we need to create a hash table of size n.

Problem-27 Can we solve Problem-24 just in one scan?

Solution: Efficient Approach: Use two pointers. Move one pointer at twice the speed of the second. When the first pointer reaches end of the list, the second pointer will be pointing to the middle node.

Note: If the list has even number of nodes, the middle node will be of $\lfloor n/2 \rfloor$.

```
struct ListNode * FindMiddle(struct ListNode *head) {
        struct ListNode *ptr1x, *ptr2x;
        ptr1x = ptr2x = head;
        int i=0;
        // keep looping until we reach the tail (next will be NULL for the last node)
        while(ptr1x→next != NULL) {
                if(i == 0) {
                        ptr1x = ptr1x→next; //increment only the 1st pointer
                        i=1;
                }
                else if( i == 1) {
                        ptr1x = ptr1x→next; //increment both pointers
                        ptr2x = ptr2x→next;
                        i = 0;
                }
        }
        return ptr2x;      //now return the ptr2 which points to the middle node
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-28 How will you display a linked list from the end?

Solution: Traverse recursively till end of the linked list. While coming back, start printing the elements.

```
//This Function will print the linked list from end
void PrintListFromEnd(struct ListNode *head) {
        if(!head)
                return;
        PrintListFromEnd(head→next);
        printf("%d ",head→data);
}
```

Time Complexity: O(n). Space Complexity: O(n)→ for Stack.

Problem-29 Check whether the given Linked List length is even or odd?

Solution: Use a $2x$ pointer. Take a pointer which moves at $2x$ [two nodes at a time]. At the end, if the length is even then pointer will be NULL otherwise it will point to last node.

```
int IsLinkedListLengthEven(struct ListNode * listHead) {
        while(listHead && listHead→next)
            listHead = listHead→next→next;
        if(!listHead)
                return 0;
        return 1;
}
```

Time Complexity: O($\lfloor n/2 \rfloor$) ≈O(n). Space Complexity: O(1).

Problem-30 If the head of a linked list is pointing to kth element, then how will you get the elements before kth element?

Solution: Use Memory Efficient Linked Lists [XOR Linked Lists].

Problem-31 Given two sorted Linked Lists, how to merge them into the third list in sorted order?

Solution:

```
struct ListNode *MergeList(struct ListNode *a, struct ListNode *b) {
        struct ListNode *result = NULL;
        if(a == NULL)
                return b;
        if(b == NULL)
                return a;
        if(a→data <= b→data) {
                result =a;
                result→next = MergeList(a→next, b);
        }
        else {
                rresult =b;
                result→next = MergeList(b→next,a);
        }
        return result;
}
```

Time Complexity – O($n + m$), where n and m are lengths of two lists.

Problem-32 Reverse the linked list in pairs. If you have a linked list that holds $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow X$, then after the function has been called the linked list would hold $2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow X$.

Solution: //Recursive Version

```
void ReversePairRecursive(struct ListNode *head) {
        struct ListNode *temp;
        if(head ==NULL || head→next ==NULL)
                return;                          //base case for empty or 1 element list
        else {  //Reverse first pair
                temp = head→next;
                head→next = temp→next;
                temp→next = head;
                //Call the method recursively for the rest of the list
                ReversePairRecursive(head→next);
        }
}
/*Iterative version*/
void ReversePairIterative(struct ListNode *head) {
        struct ListNode *temp, *temp2, *current = head;
        while(current != NULL && current→next != NULL) {
                //Swap the pair
                temp = current→next;
                temp2 = temp→next;
                temp→next = current;
                current→next = temp2;
                //Advance the current pointer
                if(current)
                    current = current→next;
        }
}
```

Time Complexity – O(n). Space Complexity - O(1).

Problem-33 Given a binary tree convert it to doubly linked list.

Solution: Refer *Trees* chapter.

Problem-34 How do we sort the Linked Lists?

Solution: Refer *Sorting* chapter.

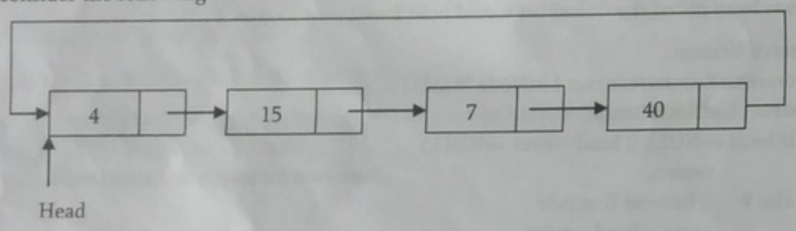
Problem-35 If we want to concatenate two linked lists which of the following gives O(1) complexity?
1) Singly linked lists 2) Doubly linked lists 3) Circular doubly linked lists

Solution: Circular Doubly Linked Lists. This is because for singly and doubly linked lists, we need to traverse the first list till the end and append the second list. But in case of circular doubly linked lists we don't have to traverse the lists.

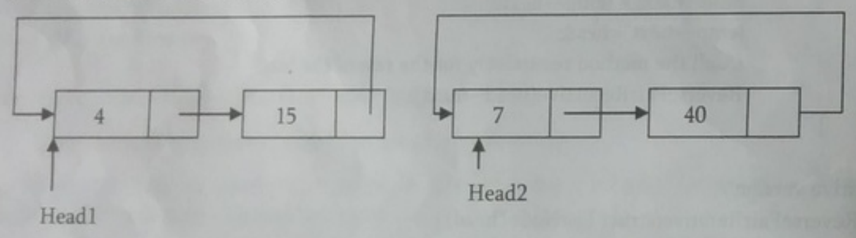
Problem-36 Split a Circular Linked List into two equal parts. If the number of nodes in the list are odd then make first list one node extra than second list.

Solution:
Algorithm
- Store the mid and last pointers of the circular linked list using Floyd cycle finding algorithm.
- Make the second half circular.
- Make the first half circular.
- Set head pointers of the two linked lists.

As an example, consider the following circular list.



After the split, the above list will look like:




```
// structure for a node
struct ListNode {
        int data;
        struct ListNode *next;
};
void SplitList(struct ListNode *head, struct ListNode **head1, struct ListNode **head2) {
        struct ListNode *slowPtr = head;
        struct ListNode *fastPtr = head;
        if(head == NULL)
                return;
        /* If there are odd nodes in the circular list then fastPtr→next becomes
         head and for even nodes fastPtr→next→next becomes head */
```



```
        while(fastPtr→next != head && fastPtr→next→next != head) {
                fastPtr = fastPtr→next→next;
                slowPtr = slowPtr→next;
        }
        // If there are even elements in list then move fastPtr
        if(fastPtr→next→next == head)
                fastPtr = fastPtr→next;
        // Set the head pointer of first half
        *head1 = head;
        // Set the head pointer of second half
        if(head→next != head)
                *head2 = slowPtr→next;
        // Make second half circular
        fastPtr→next = slowPtr→next;
        // Make first half circular
        slowPtr→next = head;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-37 How will you check if the linked list is palindrome or not?

Solution:
Algorithm
1. Get the middle of the linked list.
2. Reverse the second half of the linked list.
3. Compare the first half and second half.
4. Construct the original linked list by reversing the second half again and attaching it back to the first half.

Time Complexity: O(n). Space Complexity: O(1).

Problem-38 For a given K value ($K > 0$) reverse blocks of K nodes in a list.
Example: Input: 1 2 3 4 5 6 7 8 9 10. Output for different K values:
For $K = 2$: 2 1 4 3 6 5 8 7 10 9 For $K = 3$: 3 2 1 6 5 4 9 8 7 10 For $K = 4$: 4 3 2 1 8 7 6 5 9 10

Solution:
Algorithm: This is an extension of swapping nodes in a linked list.
1) Check if remaining list has K nodes.
   a. If yes get the pointer of $K + 1^{th}$ node.
   b. Else return.
2) Reverse first K nodes.
3) Set next of last node (after reversal) to $K + 1^{th}$ node.
4) Move to $K + 1^{th}$ node.
5) Go to step 1.
6) $K - 1^{th}$ node of first K nodes becomes the new head if available. Otherwise, we can return the head.

```
struct ListNode * GetKPlusOneThNode(int K, struct ListNode *head) {
        struct ListNode *Kth;
        int i = 0;
        if(!head)
                return head;
        for (i = 0, Kth = head; Kth && (i < K); i++, Kth = Kth→next);
        if(i == K && Kth != NULL)
                return Kth;
```

```
        return head→next;
}
int HasKnodes(struct ListNode *head, int K) {
        int i =0;
        for(i = 0; head && (i < K); i++, head = head→next);
        if(i == K)
                return 1;
        return 0;
}
struct ListNode *ReverseBlockOfK-nodesInLinkedList(struct ListNode *head, int K) {
        struct ListNode *cur = head, *temp, *next, newHead;
        int i;
        if(K==0 || K==1)
                return head;
        if(HasKnodes(cur, K-1))
                newHead = GetKPlusOneThNode(K-1, cur);
        else
                newHead = head;
        while(cur && HasKnodes(cur, K)) {
                temp = GetKPlusOneThNode(K, cur);
                i=0;
                while(i < K) {
                        next = cur→next;
                        cur→next=temp;
                        temp = cur;
                        cur = next;
                        i++;
                }
        }
        return newHead;
}
```

Problem-39 Is it possible to get O(1) access time for Linked Lists?

Solution: **Yes**. Create a linked list at the same time keep it in a hash table. For n elements we have to keep all the elements into hash table which gives preprocessing time of $O(n)$. To read any element we require only constant time $O(1)$ and to read n elements we require $n * 1$ unit of time $= n$ units. Hence by using amortized analysis we can say that element access can be performed within $O(1)$ time.

Time Complexity – $O(1)$ [Amortized]. Space Complexity - $O(n)$ for Hash.

Problem-40 **JosephusCircle:** N people have decided to elect a leader by arranging themselves in a circle and eliminating every M^{th} person around the circle, closing ranks as each person drops out. Find which person will be the last one remaining (with rank 1).

Solution: Assume the input is a circular linked list with N nodes and each node has a number (range 1 to N) associated with it. The head node has number 1 as data.

```
struct ListNode *GetJosephusPosition(){
        struct ListNode *p, *q;
        printf("Enter N (number of players): "); scanf("%d", &N);
        printf("Enter M (every M-th payer gets eliminated): "); scanf("%d", &M);
```



```
        // Create circular linked list containing all the players:
        p = q = malloc(sizeof(struct node));
        p→data = 1;
        for (int i = 2; i <= N; ++i) {
                p→next = malloc(sizeof(struct node));
                p = p→next;
                p→data = i;
        }
        p→next = q;  // Close the circular linked list by having the last node point to the first.
        // Eliminate every M-th player as long as more than one player remains:
        for (int count = N; count > 1; --count) {
                for (int i = 0; i < M - 1; ++i)
                        p = p→next;
                p→next = p→next→next;  // Remove the eiminated player from the circular linked list.
        }
        printf("Last player left standing (Josephus Position) is %d\n.", p→data);
}
```

Problem-41 Given a linked list consists of data, next pointer and also a random pointer which points to a random node of the list. Give an algorithm for cloning the list.

Solution: We can use a hash table to associate newly created nodes with the instances of node in the given list.

Algorithm:

- Scan the original list and for each node X, create a new node Y with data of X, then store the pair (X, Y) in hash table using X as a key. Note that during this scan set $Y \rightarrow next$ and $Y \rightarrow random$ to $NULL$ and we will fix them in the next scan.
- Now for each node X in the original list we have a copy Y stored in our hash table. We scan again the original list and set the pointers buildings the new list.

```
struct ListNode *Clone(struct ListNode *head){
        struct ListNode *X, *Y;
        struct HashTable *HT = CreateHashTable();
        X = head;
        while (X != NULL) {
                Y = (struct ListNode *)malloc(sizeof(struct ListNode *));
                Y→data = X→data;
                Y→next = NULL;
                Y→random = NULL;
                HT.insert(X, Y);
                X = X→next;
        }
        X = head;
        while (X != NULL) {
                // get the node Y corresponding to X from the hash table
                Y = HT.get(X);
                Y→next = HT.get(X→next);
                Y.setRandom = HT.get(X→random);
                X = X→next;
        }
        // Return the head of the new list, that is the Node Y
        return HT.get(head);
}
```

Time Complexity: O(n). Space Complexity: O(n).

Problem-42 Can we solve Problem-41 without any extra space?

Solution: Yes. Follow the comments in below code and traceout.

```
void Clone(struct ListNode *head){
        struct ListNode *temp, *temp2;
        //Step1: put temp→random in temp2→next, so that we can reuse the temp→random field to point to temp2.
        temp = head;
        while (temp != NULL) {
                temp2 = (struct ListNode *)malloc(sizeof(struct ListNode *));
                temp2→data = temp→data;
                temp2→next = temp→random;
                temp→random = temp2;
                temp = temp→next;
        }
        //Step2: Setting temp2→random.  temp2→next is the old copy of the node that
        // temp2→random should point to, so temp→next→random is the new copy.
        temp = head;
        while (temp != NULL) {
                temp2 = temp→random;
                temp2→random = temp→next→random;
                temp = temp→next;
        }
        //Step3: Repair damage to old list and fill in next pointer in new list.
        temp = head;
        while (temp != NULL) {
                temp2 = temp→random;
                temp→random = temp2→next;
                temp2→next = temp→next→random;
                temp = temp→next;
        }
}
```

Time Complexity: O($3n$) ≈O(n). Space Complexity: O(1).

Problem-43 Given a linked list with even and odd numbers, give an algorithm for making changes to list in such a that all even numbers appear at the beginning.

Solution: To solve this problem, we can use the splitting logic. While traversing the list, split the linked list into two: one containing all even nodes and other containing all odd nodes. Now, to get the final list, we can simply append the odd node linked list after the even node linked list.

To split the linked list, traverse the original linked list and move all odd nodes to a separate linked list of all odd nodes. At the end of loop, the original list will have all the even nodes and the odd node list will have all the odd nodes. To keep the ordering of all nodes same, we must insert all the odd nodes at the end of the odd node list.

Time Complexity: O(n). Space Complexity: O(1).

Chapter-4

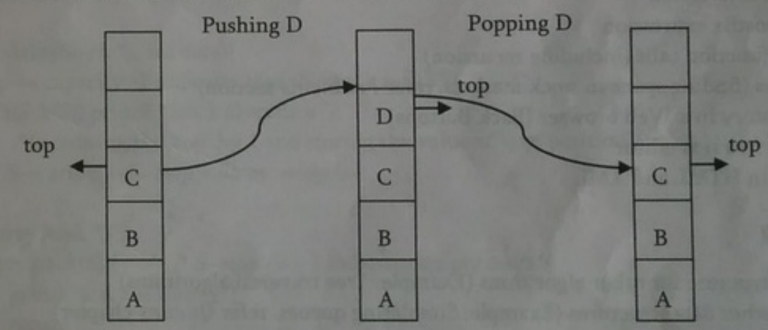
STACKS

4.1 What is a Stack?

A stack is a simple data structure used for storing data (similar to Linked Lists). In stack, the order in which the data arrives is important. The pile of plates of a cafeteria is a good example of stack. The plates are added to the stack as they are cleaned. They are placed on the top. When a plate is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used.

Definition: A *stack* is an ordered list in which insertion and deletion are done at one end, where the end is called as *top*. The last element inserted is the first one to be deleted. Hence, it is called Last in First out (LIFO) or First in Last out (FILO) list.

Special names are given to the two changes that can be made to a stack. When an element is inserted in a stack, the concept is called as *push*, and when an element is removed from the stack, the concept is called as *pop*. Trying to pop out an empty stack is called as *underflow* and trying to push an element in a full stack is called as *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshots of the stack.



4.2 How Stacks are used?

Consider a working day in the office. Let us assume a developer is working on a long-term project. The manager then gives the developer a new task, which is more important. The developer places the long-term project aside and begins work on the new task. The phone then rings, this is the highest priority, as it must be answered immediately. The developer pushes the present task into the pending tray and answers the phone. When the call is complete the task abandoned top answer the phone is retrieved from the pending tray and work progresses. If another call comes in, it may have to be handled in the same manner, but eventually the new task will be finished, and the developer can draw the long-term project from the pending tray and continue with that.

4.3 Stack ADT

The following operations make a stack an ADT. For simplicity assume the data is of integer type.

Main stack operations

- Push (int data): Inserts *data* onto stack.

- int Pop(): Removes and returns the last inserted element from the stack.

Auxiliary stack operations

- int Top(): Returns the last inserted element without removing it.
- int Size(): Returns the number of elements stored in stack.
- int IsEmptyStack(): Indicates whether any elements are stored in stack or not.
- int IsFullStack(): Indicates whether the stack is full or not.

Exceptions

Attempting the execution of an operation may sometimes cause an error condition, called an exception. Exceptions are said to be "thrown" by an operation that cannot be executed. In the Stack ADT, operations pop and top cannot be performed if the stack is empty. Attempting the execution of pop (top) on an empty stack throws an exception. Trying to push an element in a full stack throws an exception.

4.4 Applications

Following are the some of the applications in which stacks plays an important role.

Direct applications

- Balancing of symbols
- Infix-to-postfix conversion
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Finding of spans (finding spans in stock markets, refer *Problems* section)
- Page-visited history in a Web browser [Back Buttons]
- Undo sequence in a text editor
- Matching Tags in HTML and XML

Indirect applications

- Auxiliary data structure for other algorithms (Example: Tree traversal algorithms)
- Component of other data structures (Example: Simulating queues, refer *Queues* chapter)

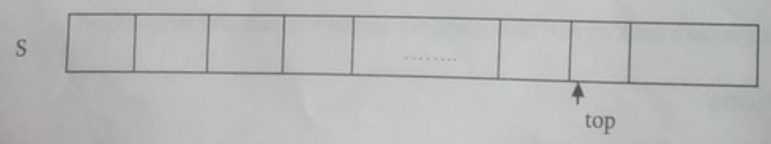
4.5 Implementation

There are many ways of implementing stack ADT and below are the commonly used methods.

- Simple array based implementation
- Dynamic array based implementation
- Linked lists implementation

Simple Array Implementation

This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the top element.



The array storing the stack elements may become full. A push operation will then throw a *full stack exception*. Similarly, if we try deleting an element from empty stack then it will throw *stack empty exception*.

```
struct ArrayStack {
        int top;
        int capacity;
        int *array;
};
struct ArrayStack *CreateStack() {
        struct ArrayStack *S = malloc(sizeof(struct ArrayStack));
        if(!S) return NULL;
        S→capacity = 1;
        S→top = -1;
        S→array= malloc(S→capacity * sizeof(int));
        if(!S→array) return NULL;
        return S;
}
int IsEmptyStack(struct ArrayStack *S) {
        return (S→top == -1);      // if the condition is true then 1 is returned else 0 is returned
}
int IsFullStack(struct ArrayStack *S){
        //if the condition is true then 1 is returned else 0 is returned
        return (S→top == S→capacity - 1);
}
void Push(struct ArrayStack *S, int data){
        /* S→top == capacity -1 indicates that the stack is full*/
        if(IsFullStack(S)) printf( "Stack Overflow");
        else      /*Increasing the 'top' by 1 and storing the value at 'top' position*/
                S→ array[++S→top]= data;
}
int Pop(struct ArrayStack *S){
        if(IsEmptyStack(S)){      /* S→top == - 1 indicates empty stack*/
                printf("Stack is Empty");
                return 0;
        }
        else /* Removing element from 'top' of the array and reducing 'top' by 1*/
                return (S→ array[S→top--]);
}
void DeleteStack(struct DynArrayStack *S){
        if(S) {   if(S→array) free(S→array);
                free(S);
        }
}
```

Performance & Limitations

Performance

Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$

Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStack()	O(1)
Time Complexity of DeleteStack()	O(1)

Limitations

The maximum size of the stack must be defined in prior and cannot be changed. Trying to push a new element into a full stack causes an implementation-specific exception.

Dynamic Array Implementation

First, let's consider how we implemented a simple array based stack. We took one index variable *top* which points to the index of the most recently inserted element in the stack. To insert (or push) an element, we increment *top* index and then place the new element at that index. Similarly, to delete (or pop) an element we take the element at *top* index and then decrement the *top* index. We represent empty queue with *top* value equal to −1. The issue still need to be resolved is that what we do when all the slots in fixed size array stack are occupied?

First try: What if we increment the size of the array by 1 every time the stack is full?

- Push(): increase size of S[] by 1
- Pop(): decrease size of S[] by 1

Problems with this approach?

This way of incrementing the array size is too expensive. Let us a see the reason for this. For example, at $n = 1$, to push an element create a new array of size 2 and copy all the old array elements to new array and at the end add the new element. At $n = 2$, to push an element create a new array of size 3 and copy all the old array elements to new array and at the end add the new element.

Similarly, at $n = n - 1$, if we want to push an element create a new array of size n and copy all the old array elements to new array and at the end add the new element. After n push operations the total time $T(n)$ (number of copy operations) is proportional to $1 + 2 + \ldots + n \approx O(n^2)$.

Alternative Approach: Repeated Doubling

Let us improve the complexity by using array *doubling* technique. If the array is full, create a new array of twice the size, and copy items. With this approach, pushing n items takes time proportional to n (not n^2).

For simplicity, let us assume that initially we started with $n = 1$ and moved till $n = 32$. That means, we do the doubling at $1, 2, 4, 8, 16$. The other way of analyzing the same is, at $n = 1$, if we want to add (push) an element then double the current size of array and copy all the elements of old array to new array.

At, $n = 1$, we do 1 copy operation, at $n = 2$, we do 2 copy operations, and n = 4, we do 4 copy operations and so on. By the time we reach $n = 32$, the total number of copy operations is $1 + 2 + 4 + 8 + 16 = 31$ which is approximately equal to $2n$ value (32). If we observe carefully, we are doing the doubling operation *logn* times.

Now, let us generalize the discussion. For n push operations we double the array size *logn* times. That means, we will have *logn* terms in below expression. The total time $T(n)$ of a series of n push operations is proportional to

$$1 + 2 + 4 + 8 \ldots + \frac{n}{4} + \frac{n}{2} + n = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots + 4 + 2 + 1$$
$$= n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right)$$
$$= n(2) \approx 2n = O(n)$$

$T(n)$ is $O(n)$ and the amortized time of a push operation is $O(1)$.


```
struct DynArrayStack {
        int top;
        int capacity;
        int *array;
};
struct DynArrayStack *CreateStack(){
        struct DynArrayStack *S = malloc(sizeof(struct DynArrayStack));
        if(!S) return NULL;
        S→capacity = 1;
        S→top = -1;
        S→array = malloc(S→capacity * sizeof(int));        // allocate an array of size 1 initially
        if(!S→array) return NULL;
        return S;
}
int IsFullStack(struct DynArrayStack *S){
        return (S→top == S→capacity-1);
}
void DoubleStack(struct DynArrayStack *S){
        S→capacity *= 2;
        S→array = realloc(S→array, S→capacity);
}
void Push(struct DynArrayStack *S, int x){
        // No overflow in this implementation
        if(IsFullStack(S))
                DoubleStack(S);
        S→array[++S→top] = x;
}
int IsEmptyStack(struct DynArrayStack *S){
        return S→top == -1;
}
int Top(struct DynArrayStack *S){
        if(IsEmptyStack(S))
                return INT_MIN;
        return S→array[S→top];
}
int Pop(struct DynArrayStack *S){
        if(IsEmptyStack(S))
                return INT_MIN;
        return S→array[S→top--];
}
void DeleteStack(struct DynArrayStack *S){
        if(S) {
                if(S→array) free(S→array);
                free(S);
        }
}
```

Performance

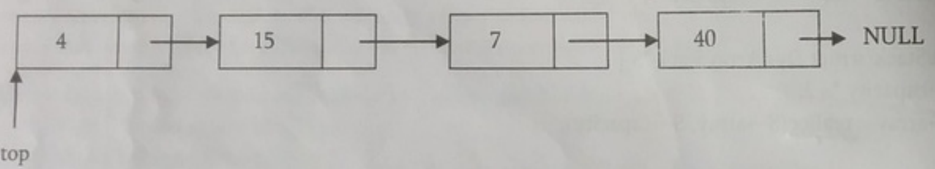
Let n be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of CreateStack()	O(1)
Time Complexity of Push()	O(1) (Average)
Time Complexity of Pop()	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmptyStack()	O(1))
Time Complexity of IsFullStack()	O(1)
Time Complexity of DeleteStack()	O(1)

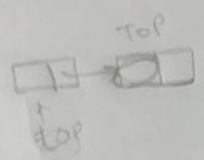
Note: Too many doublings may cause memory overflow exception.

Linked List Implementation

The other way of implementing stacks is by using Linked lists. Push operation is implemented by inserting element at the beginning of the list. Pop operation is implemented by deleting the node from the beginning (the header/top node).




```
struct ListNode{
        int data;
        struct ListNode *next;
};
struct Stack *CreateStack(){
        return NULL;
}
void Push(struct Stack **top, int data){
        struct Stack *temp;
        temp = malloc(sizeof(struct Stack));
        if(!temp) return NULL;
        temp→data = data;
        temp→next = *top;
        *top = temp;
}
int IsEmptyStack(struct Stack *top){
        return top == NULL;
}
int Pop(struct Stack **top){
        int data;
        struct Stack *temp;
        if(IsEmptyStack(top))
                return INT_MIN;
        temp = *top;
        *top = *top→next;
        data = temp→data;
        free(temp);
        return data;
```




```
}
int Top(struct Stack * top){
        if(IsEmptyStack(top))  return INT_MIN;
        return top→next→data;
}
void DeleteStack(struct Stack **top){
        struct Stack *temp, *p;
        p = *top;
        while( p→next) {
                temp = p→next;
                p→next = temp→next;
                free(temp);
        }
        free(p);
}
```

Performance

Let n be the number of elements in the stack. Let n be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of CreateStack()	O(1)
Time Complexity of Push()	O(1) (Average)
Time Complexity of Pop()	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of DeleteStack()	$O(n)$

4.6 Comparison of Implementations

Comparing Incremental Strategy and Doubling Strategy

We compare the incremental strategy and doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations. We start with an empty stack represented by an array of size 1. We call *amortized* time of a push operation is the average time taken by a push over the series of operations, i.e., $T(n)/n$.

Incremental Strategy: The amortized time (average time per operation) of a push operation is $O(n)$ [$O(n^2)/n$].

Doubling Strategy: In this method, the amortized time of a push operation is O(1) [$O(n)/n$].

Note: For reasoning, refer implementation section.

Comparing Array Implementation and Linked List Implementation

Array Implementation

- Operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of n operations (starting from empty stack) -- "amortized" bound takes time proportional to n.

Linked list Implementation

- Grows and shrinks gracefully.
- Every operation takes constant time O(1).

- Every operation uses extra space and time to deal with references.

4.7 Problems on Stacks

Problem-1 Discuss how stacks can be used for checking balancing of symbols?

Solution: Stacks can be used to check whether the given expression has balanced symbols or not. This algorithm is very much useful in compilers. Each time parser reads one character at a time. If the character is an opening delimiter like (, {, or [- then it is written to the stack. When a closing delimiter is encountered like),}, or]- is encountered the stack is popped. The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line. A linear-time $O(n)$ algorithm based on stack can be given as:

Algorithm

a) Create a stack.
b) while (end of input is not reached) {
   1) If the character read is not a symbol to be balanced, ignore it.
   2) If the character is an opening symbol like (, [, {, push it onto the stack
   3) If it is a closing symbol like),],}, then if the stack is empty report an error. Otherwise pop the stack.
   4) If the symbol popped is not the corresponding opening symbol, report an error.

   }
c) At end of input, if the stack is not empty report an error

Examples:

Example	Valid?	Description
(A+B)+(C-D)	Yes	The expression is having balanced symbol
((A+B)+(C-D)	No	One closing brace is missing
((A+B)+[C-D])	Yes	Opening and immediate closing braces correspond
((A+B)+[C-D]}	No	The last closing brace does not correspond with the first opening parenthesis

For tracing the algorithm let us assume that the input is: () (() [()])

Input Symbol, A[i]	Operation	Stack	Output
(	Push (	(	
)	Pop (Test if (and A[i] match? YES		
(	Push (	(	
(	Push (	((	
)	Pop (Test if (and A[i] match? YES	(	
[	Push [	([	
(	Push (	([(	
)	Pop (Test if(and A[i] match? YES	([	
]	Pop [Test if [and A[i] match? YES	(	

)	Pop (Test if(and A[i] match? YES		
	Test if stack is Empty? YES		TRUE

Time Complexity: $O(n)$. Since, we are scanning the input only once. Space Complexity: $O(n)$ [for stack].

Problem-2 Discuss infix to postfix conversion algorithm using stack?

Solution: Before discussing the algorithm, first let us see the definitions of infix, prefix and postfix expressions.

Infix: An infix expression is a single letter, or an operator, proceeded by one infix string and followed by another Infix string.

A
A+B
(A+B)+ (C-D)

Prefix: A prefix expression is a single letter, or an operator, followed by two prefix strings. Every prefix string longer than a single variable contains an operator, first operand and second operand.

A
+AB
++AB-CD

Postfix: A postfix expression (also called Reverse Polish Notation) is a single letter or an operator, preceded by two postfix strings. Every postfix string longer than a single variable contains first and second operands followed by an operator.

A
AB+
AB+CD-+

Prefix and postfix notions are methods of writing mathematical expressions without parenthesis. Time to evaluate a postfix and prefix expression is $O(n)$, were n is the number of elements in the array.

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B-C	-+ABC	AB+C-
(A+B)*C-D	-*+ABCD	AB+C*D-

Now, let us concentrate on the algorithm. In infix expressions, the operator precedence is implicit unless we use parentheses. Therefore, for the infix to postfix conversion algorithm we have to define the operator precedence (or priority) inside the algorithm. The table shows the precedence and their associatively (order of evaluation) among operators.

Token	Operator	Precedence	Associatively
( ) [ ] → .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	Left-to-right