Output

```
15 14 12 10
Element not present in the list
15 14 12 10
15 14 12
15 12
```

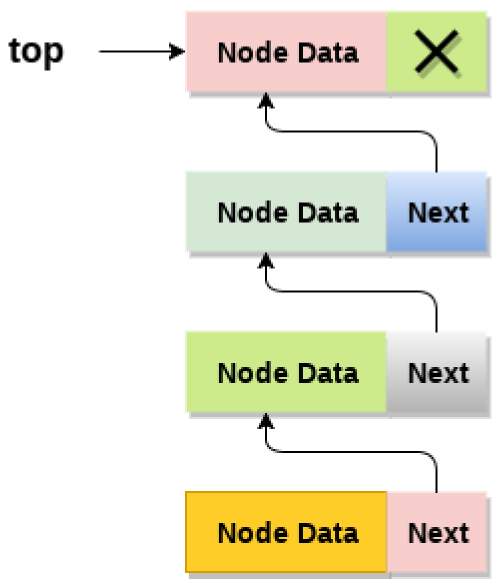
Time Complexity: O(n)
Auxiliary Space: O(n) *(due to recursion call stack)*

Linked representation of Stack and Queue:

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



Stack

The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

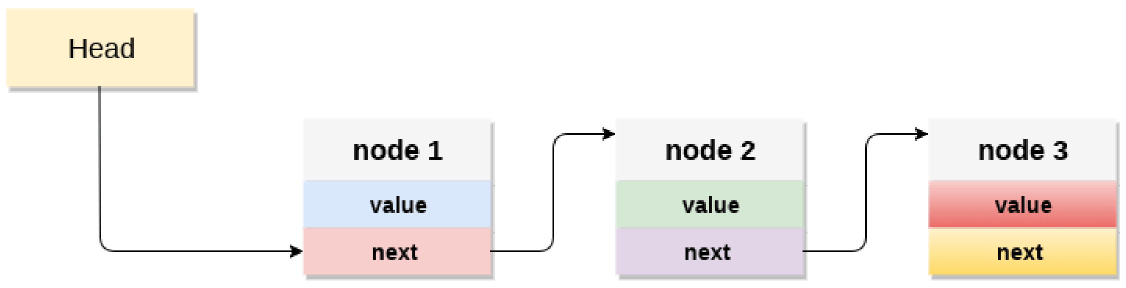
Adding a node to the stack (Push operation)

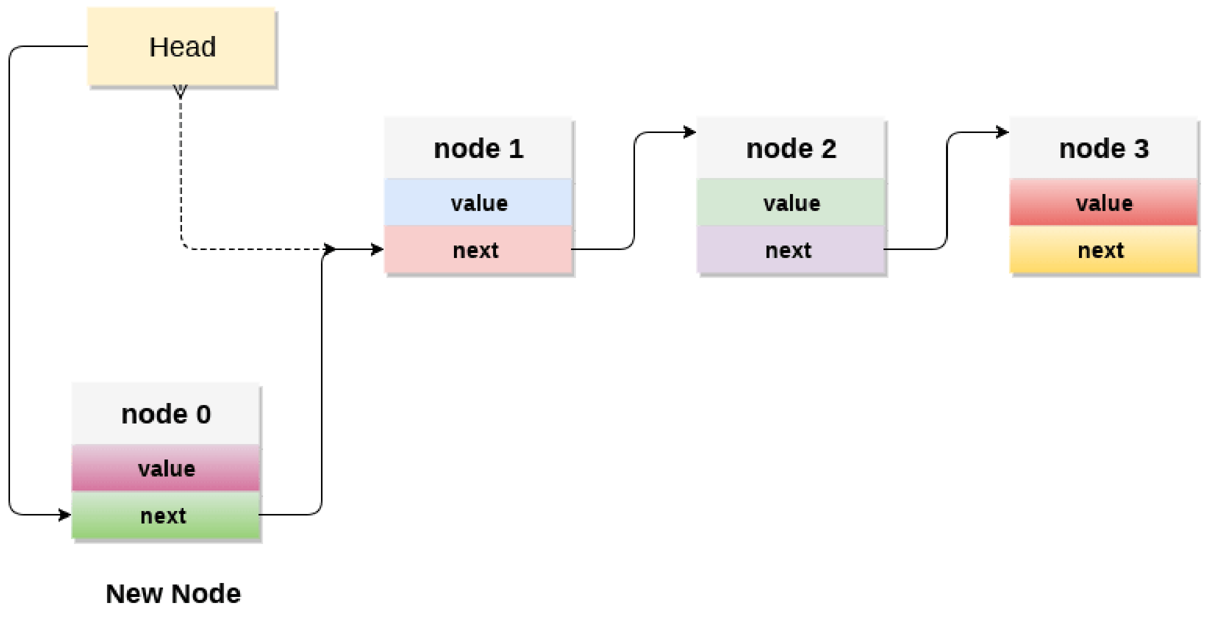
Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

Java Try Catch

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

4. **Time Complexity : o(1)**





C implementation :

```
void push ()
{
    int val;
    struct node *ptr =(struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
            head=ptr;
        }
        else
        {
            ptr->val = val;
            ptr->next = head;
            head=ptr;

        }
        printf("Item pushed");

    }
}
```

5. Deleting a node from the stack (POP operation) Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :
    1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
    2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted

and the node must be freed. The next node of the head node now becomes the head node.

6. **Time Complexity : o(n)**
C implementation

```
void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
            {
                item = head->val;
                ptr = head;
                head = head->next;
                free(ptr);
                printf("Item popped");

            }
        }
```

7. Display the nodes (Traversing)
Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.
    1. Copy the head pointer into a temporary pointer.
    2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.
8. **Time Complexity : o(n)**
C Implementation

```
void display()
```

```
{
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
        printf("Stack is empty\n");
    }
        else
        {
            printf("Printing Stack elements \n");
            while(ptr!=NULL)
            {
                printf("%d\n",ptr->val);
                ptr = ptr->next;
            }
        }
    }
```

9. Menu Driven program in C implementing all the stack operations using linked list :

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
int val;
struct node *next;
    };
    struct node *head;

    void main ()
```

```
{
    int choice=0;
    printf("\n*********Stack operations using linked list*********\n");
    printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("\n\nChose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice ");
            }
    };
}
}
void push ()
{
    int val;
    struct node *ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
            head=ptr;
        }
        else
        {
            ptr->val = val;
            ptr->next = head;
            head=ptr;

        }
        printf("Item pushed");

    }
}

void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");

    }
}
void display()
{
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
        printf("Stack is empty\n");
    }
    else
    {
        printf("Printing Stack elements \n");
        while(ptr!=NULL)
        {
            printf("%d\n",ptr->val);
            ptr = ptr->next;
        }
    }
}

```

Linked Representation of Queue:

Linked List implementation of Queue

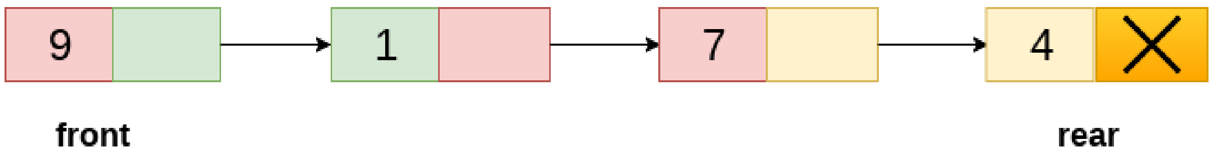
Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.
The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue
There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.
Insert operation
The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.
Firstly, allocate the memory for the new node ptr by using the following statement.

```
1. Ptr = (struct node *) malloc (sizeof(struct node));
```

There can be the two scenario of inserting this new node ptr into the linked queue.
In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
1. ptr -> data = item;
2.         if(front == NULL)
```

```
{
    front = ptr;
    rear = ptr;
    front -> next = NULL;
    rear -> next = NULL;
}
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

```
rear -> next = ptr;
    rear = ptr;
    rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

- **Step 1:** Allocate the space for the new node PTR
- **Step 2:** SET PTR -> DATA = VAL
- **Step 3:** IF FRONT = NULL
  SET FRONT = REAR = PTR
  SET FRONT -> NEXT = REAR -> NEXT = NULL
  ELSE
  SET REAR -> NEXT = PTR
  SET REAR = PTR
  SET REAR -> NEXT = NULL
  [END OF IF]
- **Step 4:** END

C Function

```
void insert(struct node *ptr, int item; )
{


    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
            }
            else
            {
                ptr -> data = item;
                if(front == NULL)
                {
                    front = ptr;
                    rear = ptr;
                    front -> next = NULL;
                    rear -> next = NULL;
                }
                else
                {
                    rear -> next = ptr;
                    rear = ptr;
                    rear->next = NULL;
                }
            }
        }
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either

the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.
Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
ptr = front;
        front = front -> next;
        free(ptr);
```

The algorithm and C function is given as follows.
Algorithm

- **Step 1:** IF FRONT = NULL
  Write " Underflow "
  Go to Step 5
  [END OF IF]
- **Step 2:** SET PTR = FRONT
- **Step 3:** SET FRONT = FRONT -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** END

C Function

```
void delete (struct node *ptr)
{
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
{
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
```

Menu-Driven Program implementing all the operations on Linked Queue

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n*************************Main Menu*****************************\n");
        printf("\n=================================================================\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
```

```
            printf("\nEnter your choice ?");
            scanf("%d",& choice);
            switch(choice)
            {
                case 1:
                insert();
                break;
                case 2:
                delete();
                break;
                case 3:
                display();
                break;
                case 4:
                exit(0);
                break;
                default:
                printf("\nEnter valid choice??\n");
            }
        }
    }
    void insert()
    {
        struct node *ptr;
        int item;

        ptr = (struct node *) malloc (sizeof(struct node));
        if(ptr == NULL)
        {
            printf("\nOVERFLOW\n");
            return;
        }
        else
        {
            printf("\nEnter value?\n");
            scanf("%d",&item);
            ptr -> data = item;
            if(front == NULL)
            {
                front = ptr;
                rear = ptr;
                front -> next = NULL;
                rear -> next = NULL;
            }
            else
            {
                rear -> next = ptr;
                rear = ptr;
                rear->next = NULL;
            }
        }
    }
    void delete ()
    {
        struct node *ptr;
        if(front == NULL)
        {
            printf("\nUNDERFLOW\n");
            return;
        }
        else
        {
            ptr = front;
            front = front -> next;
            free(ptr);
        }
```

```
}
void display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values .....\n");
        while(ptr != NULL)
        {
            printf("\n%d\n",ptr -> data);
            ptr = ptr -> next;
        }
    }
}
```

Output:

Learn more
volume is gedempt

```
***********Main Menu**********


==============================


1.insert an element

2.Delete an element

3.Display the queue

4.Exit
```

```
Enter your choice ?1

Enter value?
123

***********Main Menu**********

===============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
90

***********Main Menu**********

===============================
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

123

90

***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2
```

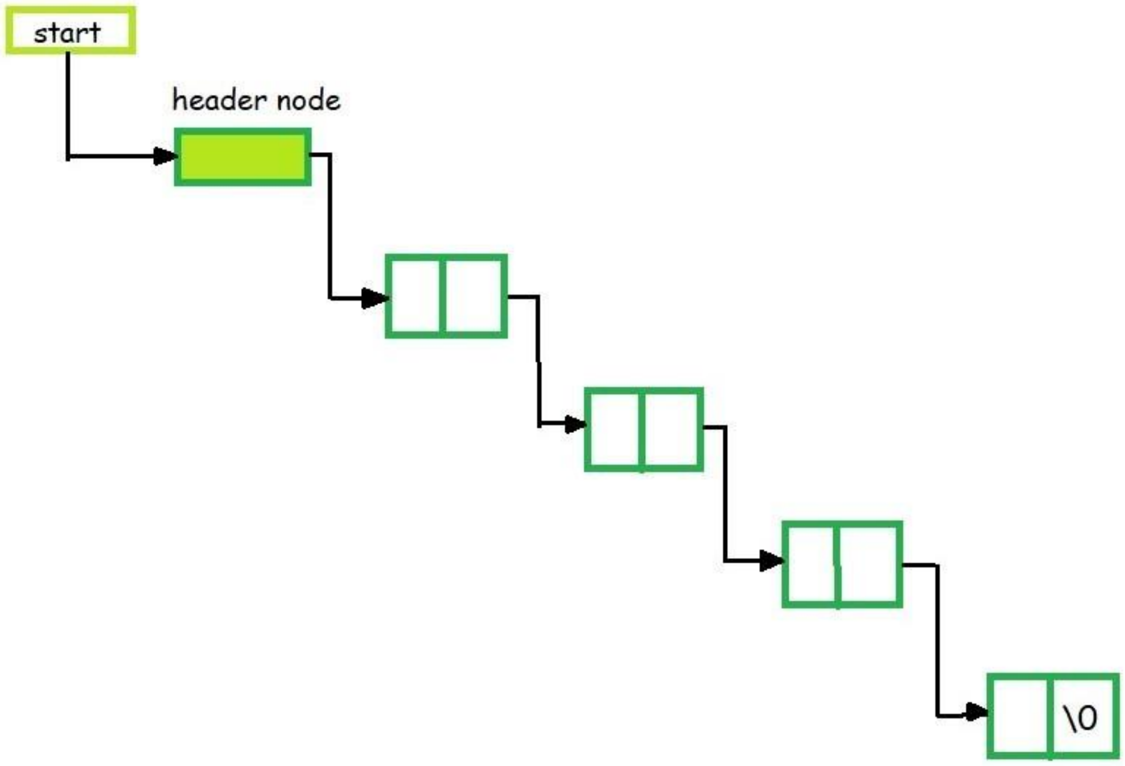
```
***********Main Menu**********

================================
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

90

***********Main Menu**********

================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?4
```

[redacted]

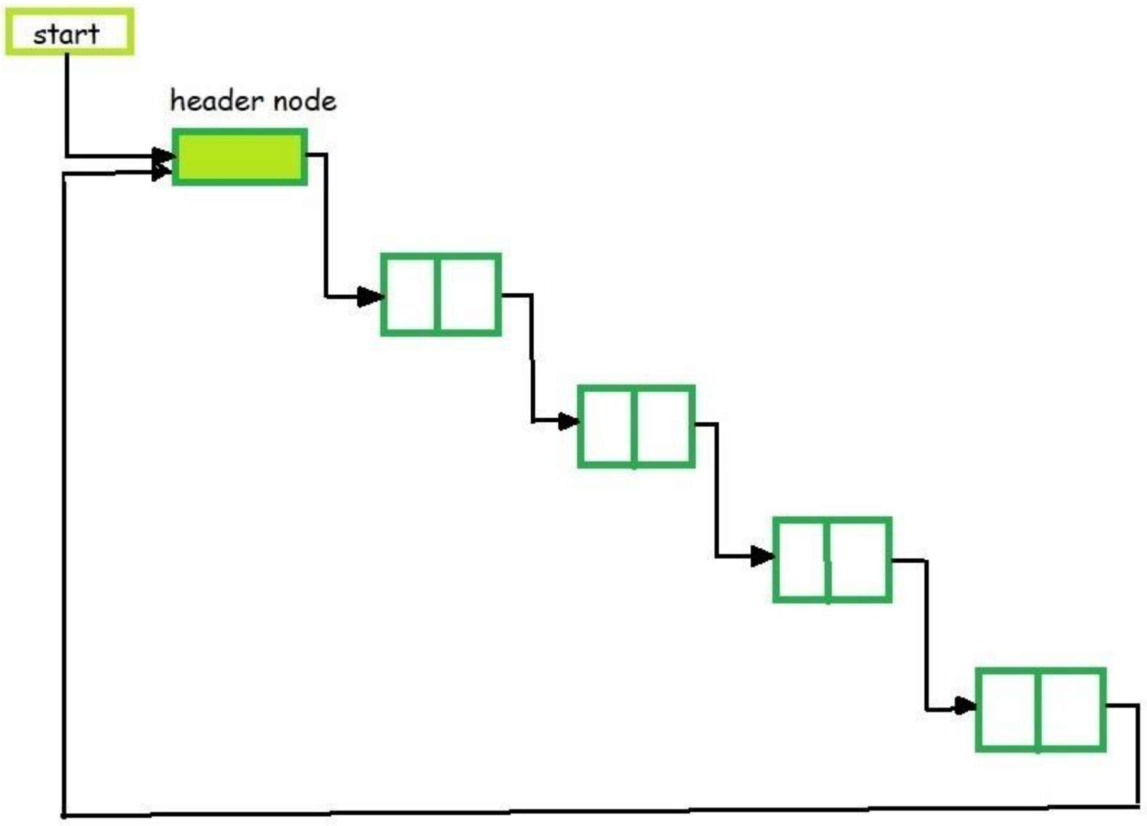
Header nodes:

A *header node* is a special node that is found at the *beginning* of the list. A list that contains this type of node, is called the header-linked list. This type of list is useful when information other than that found in each node is needed.
For example, suppose there is an application in which the number of items in a list is often calculated. Usually, a list is always traversed to find the length of the list. However, if the current length is maintained in an additional header node that information can be easily obtained.
Types of Header Linked List

1. **Grounded Header Linked List**
   It is a list whose *last node* contains the *NULL* pointer. In the header linked list the **start** pointer always points to the header node. **start -> next = NULL** indicates that the grounded header linked list is *empty*. The operations that are possible on this type of linked list are *Insertion*, *Deletion*, *and Traversing*.



2. **Circular Header Linked List**
A list in which *last node* points back to the *header node* is called circular linked list. The chains do not indicate first or last nodes. In this case, external pointers provide a frame of reference because last node of a circular linked list does **not contain** the **NULL** pointer. The possible operations on this type of linked list are *Insertion, Deletion and Traversing*.


start
header node

```
// C program for a Header Linked List
#include <malloc.h>
#include <stdio.h>

// Structure of the list
struct link {
    int info;
    struct link* next;
};

// Empty List
struct link* start = NULL;

// Function to create a header linked list
struct link* create_header_list(int data)
{

    // Create a new node
    struct link *new_node, *node;
    new_node = (struct link*)
        malloc(sizeof(struct link));
    new_node->info = data;
    new_node->next = NULL;
    // If it is the first node
    if (start == NULL) {

        // Initialize the start
        start = (struct link*)
            malloc(sizeof(struct link));
        start->next = new_node;
    }
    else {

        // Insert the node in the end
        node = start;
        while (node->next != NULL) {
            node = node->next;
        }
        node->next = new_node;
    }
    return start;
}

// Function to display the
// header linked list
struct link* display()
{
    struct link* node;
    node = start;
    node = node->next;
    while (node != NULL) {
        printf("%d ", node->info);
        node = node->next;
    }
    printf("\n");
    return start;
}

// Driver code
int main()
{

    // Create the list
    create_header_list(11);
    create_header_list(12);
    create_header_list(13);

    // Print the list
    display();
    create_header_list(14);
    create_header_list(15);
    // Print the list
    display();

    return 0;
}
```

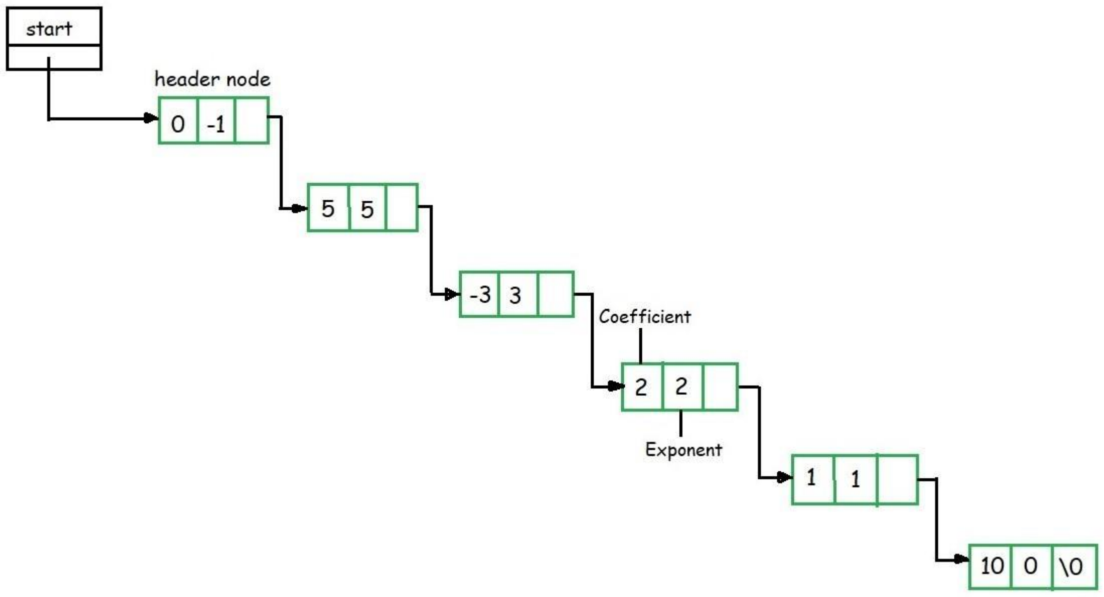
Output:

```
11 12 13
11 12 13 14 15
```

Applications of Header Linked List

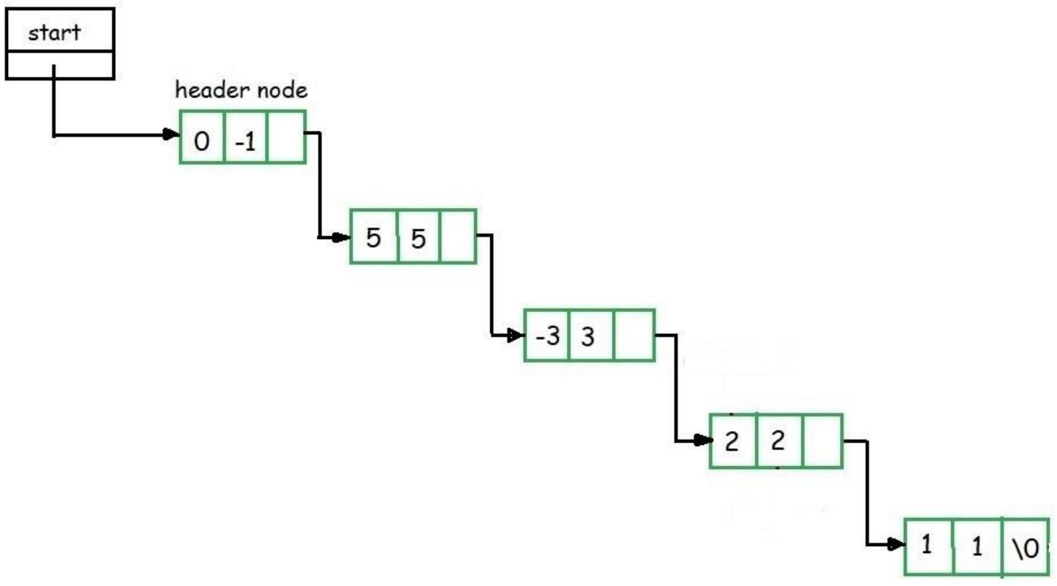
Polynomials

- The header linked lists are frequently used to maintain the polynomials in memory. The *header node* is used to represent the *zero polynomial*.
- Suppose we have **F(x) = 5x5 – 3x3 + 2x2 + x1 +10x0**
- From the polynomial represented by F(x) it is clear that this polynomial has two parts, **coefficient** and **exponent**, where, **x** is **formal parameter**. Hence, we can say that a polynomial is sum of terms, each of which consists of a coefficient and an exponent.
- The computer implementation requires implementing polynomials as a *list of pair of coefficient and exponent*. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures.
- If one wants to represent F(x) with help of linked list then the list will contain 5 nodes. When we link each node we get a linked list structure that represents polynomial **F(x)**.



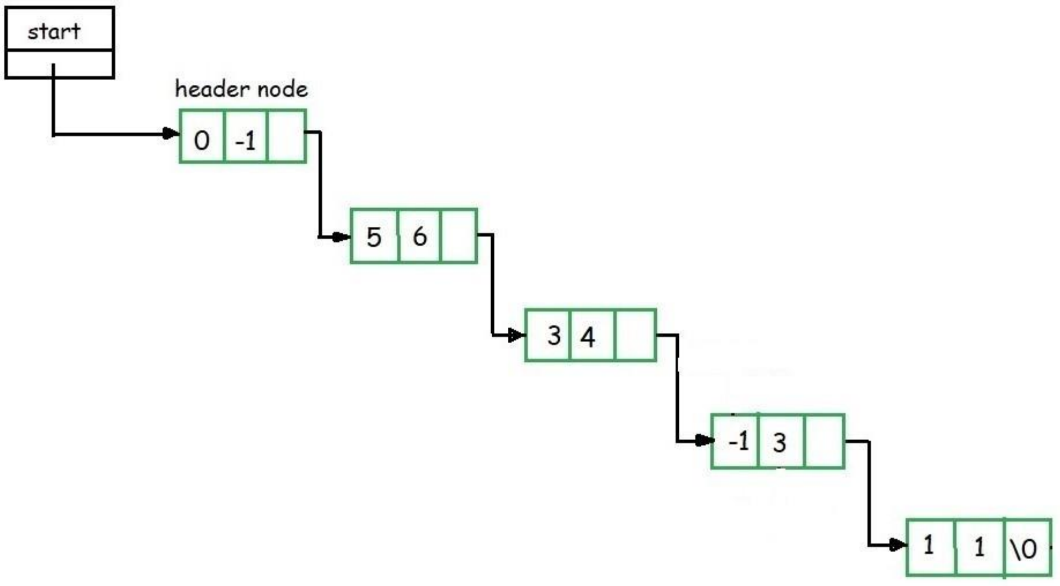
Addition of polynomials

1. To add two polynomials, we need to scan them once.
2. If we find terms with the same exponent in the two polynomials, then we add the coefficients, otherwise, we copy the term of larger exponent into the sum and go on.
3. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.
4. Suppose we have two polynomials as illustrated and we have to perform addition of these polynomials.



First Polynomial

5.



Second Polynomial

When we scan first node of the two polynomials, we find that exponential power of first node in the second polynomial is greater than that of first node of the first polynomial.

6. Here the exponent of the first node of the second polynomial is greater hence we have to copy first node of the second polynomial into the sum.
7. Then we consider the first node of the first polynomial and once again first node value of first polynomial is compared with the second node value of the second polynomial.
8. Here the first node exponent value of the first polynomial is greater than the second node exponent value of the second polynomial. We copy the first node of the first polynomial into the sum.
9. Now consider the second node of the first polynomial and compare it with the second node of the second polynomial.
10. Here the exponent value of the second node of the second polynomial is greater than the second node of the first polynomial, hence we copy the second node of the second list into the sum.
11. Now we consider the third node exponent of the second polynomial and compare it with second node exponent value of the first polynomial. We find that both are equal, hence perform addition of their coefficient and copy in to the sum.
12. This process continues till all the nodes of both the polynomial are exhausted. For example after adding the above two polynomials, we get the following *resultant polynomial* as shown.