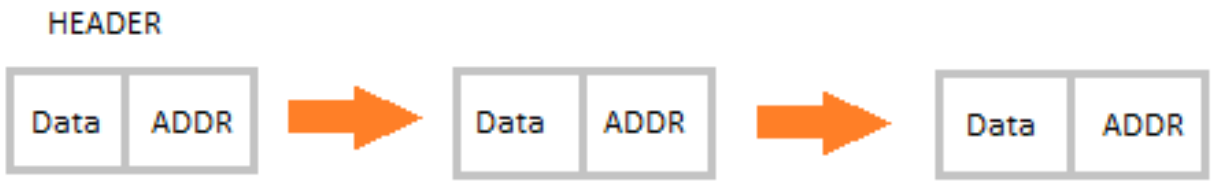
# DATA STRUCTURES – Linked List

UNIT-3

Dr. SELVA KUMAR S
ASSISTANT PROFESSOR
B.M.S. COLLEGE OF ENGINEERING

# Linked List

- Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.
- Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.
- Linked Lists are used to create trees and graphs.



# Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.

# Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

# Applications of Linked Lists

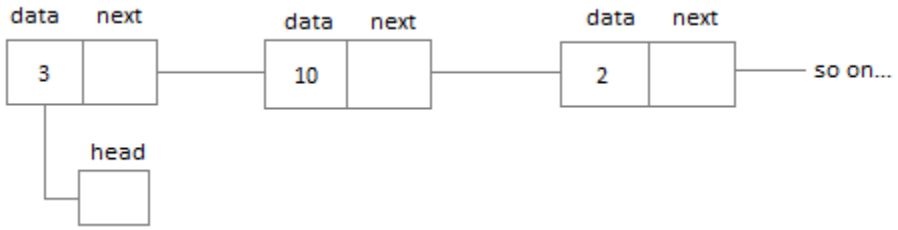
- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

# Types of Linked Lists

- Singly Linked List
- Doubly Linked List
- Circular Linked List

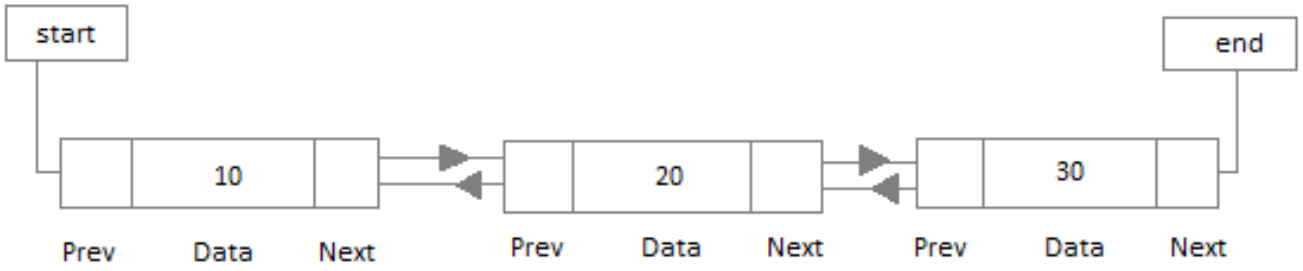
# Singly Linked List

- Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. next, which points to the next node in the sequence of nodes.
- The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.



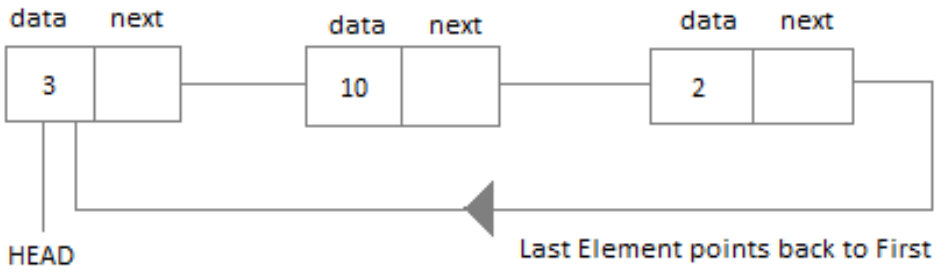
# Doubly Linked List

- In a doubly linked list, each node contains a **data** part and two addresses, one for the **previous** node and one for the **next** node.



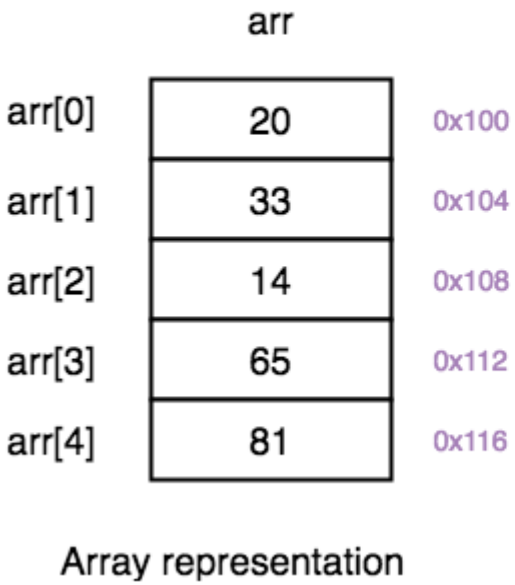
# Circular Linked List

- In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.

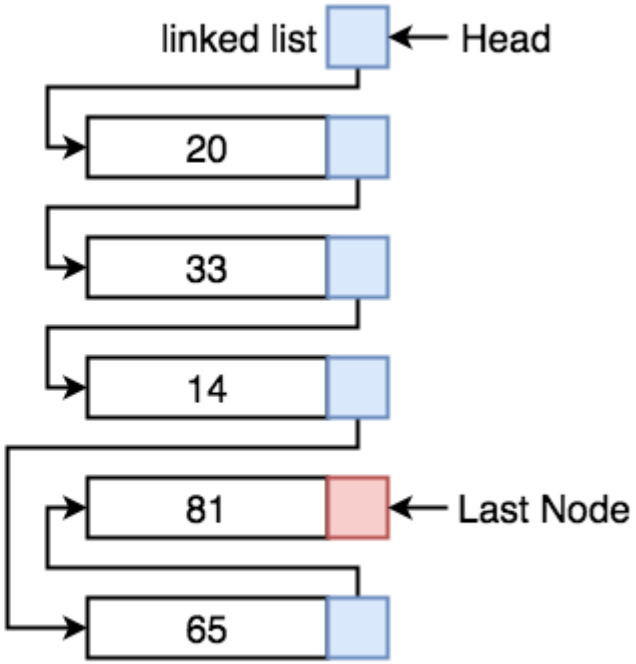


# Difference between Arrays and Linked List

| S.No. | Arrays | Linked Lists |
|---|---|---|
| 1. | Arrays is static data structure means its size can't be increased or decreased on runtime | Linked list is dynamic data structure means its size can be modified on runtime |
| 2. | If we are confirm to use n fixed block and it is not going to change in a program, so arrays is better | If we are confirm to use n fixed block then linked list use extra space for pointer to next node and it waste 2*n bytes of memory space. |
| 3. | Arrays is simpler to use | Linked list is a complex data structure and it is used basically for complex programming |
| 4. | In arrays, insertion and deletion consequences as large amount of data movements | In linked list, insertion and deletion doesn't need so much data movements |




Array representation



# Dynamic Memory Allocation Functions

| S.No. | Function Name | Meaning |
|---|---|---|
| 1. | sizeof() | This function gives the size of its arguments in terms of bytes. The arguments can be variable, arrays, structure etc. |
| 2. | malloc() | The malloc() function allocates a request size of bytes and returns a pointer to the first byte of the allocated space. |
| 3. | calloc() | The calloc() function is used to allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory. |
| 4. | realloc() | The realloc() function is used to reallocate space which is defined by the malloc() or calloc() so modifies the size of previously allocated space. |
| 5. | free() | The free() function is used for efficient use of memory we can also release the memory space that is not required. |

# Example

- Syntax for malloc():

  **ptr=(datatype *)malloc(specified-size);**

- Example 1:

  **int ptr;**

  **ptr=(int*)malloc(10*sizeof(int));**

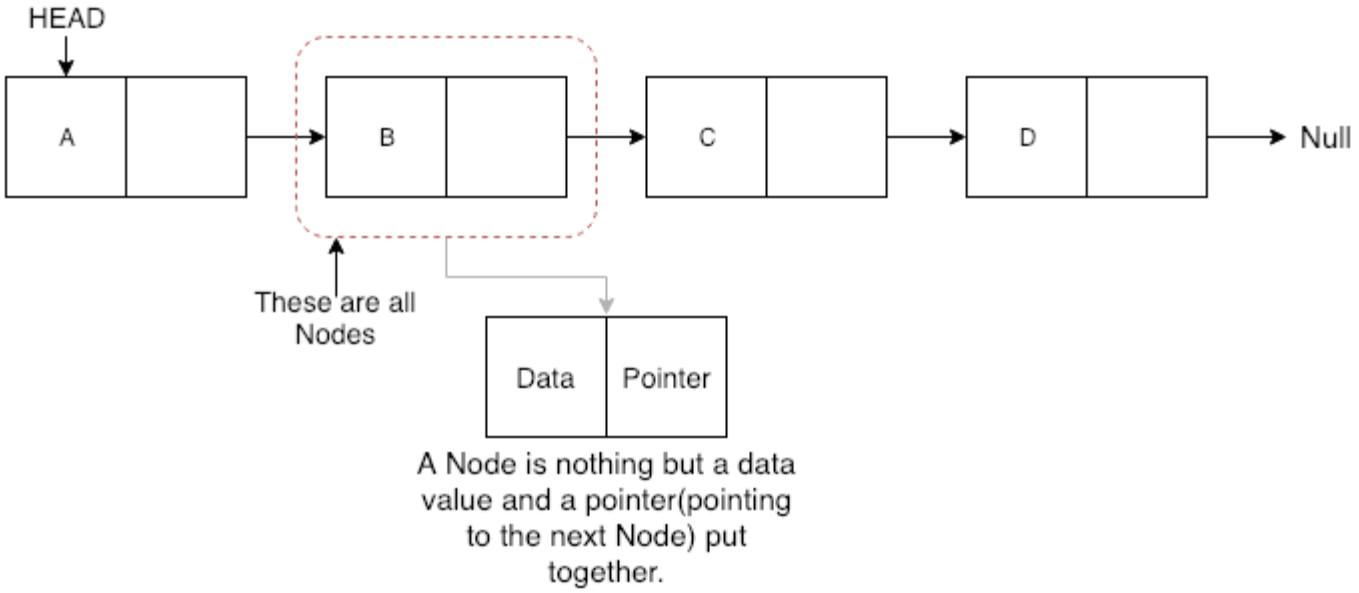
- Example 2:

  **struct *str;**

  **str=(struct node*)malloc(sizeof(struct node));**

# Singly Linked List

- What is a Node?
- A Node in a linked list holds the data value and the pointer which points to the location of the next node in the linked list.



# Node Implementation

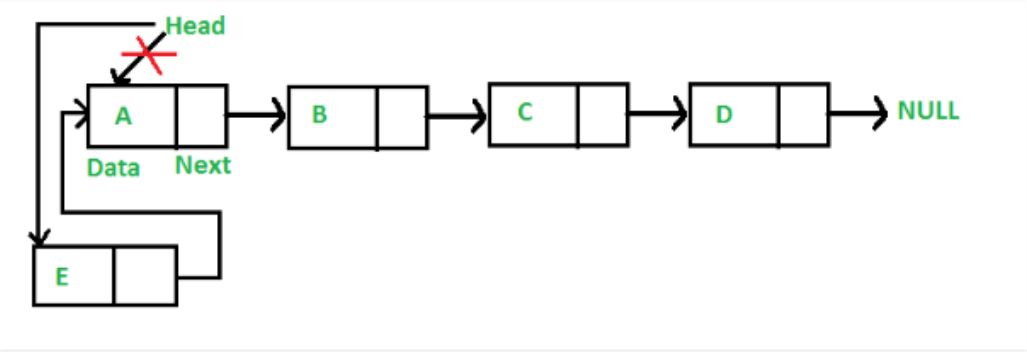
```
// A linked list node
    struct Node
     {
          int data;
          struct Node *next;
    };

    typedef struct Books {
      char title[50];
      char author[50];
      char subject[100];
      int book_id;
      struct Books *add;
    } Book;
```

# Inserting a node

- A node can be added in three ways
  **1)** At the front of the linked list
  **2)** After a given node.
  **3)** At the end of the linked list.

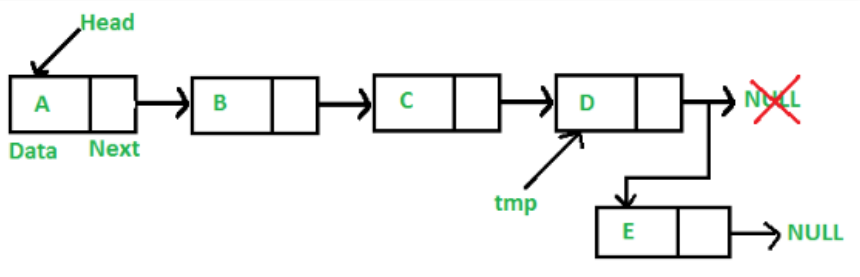
# Add a node at the front

```
/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

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

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

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



# Add a node at the end

```
/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end  */
void append(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

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

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

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

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
       *head_ref = new_node;
       return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

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

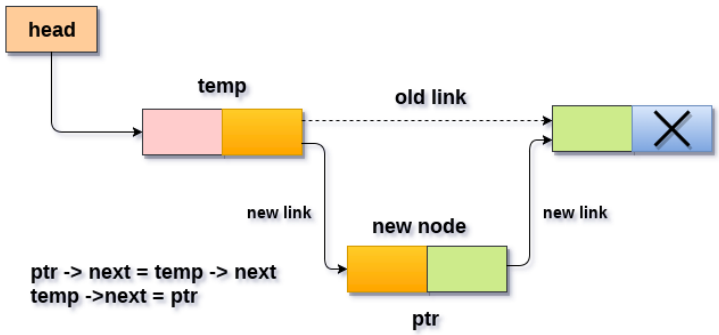


# Display the content of Linked List

```
// This function prints contents of linked list starting from head
void printList(struct Node *node)
{
  while (node != NULL)
  {
     printf(" %d ", node->data);
     node = node->next;
  }
}
```

# Add a node at specified position

```
void ins_at_pos_n(int data,int position)
  {
        struct node *ptr = (struct node*)malloc(sizeof(struct node));
        ptr->data=data;              //Creating a new node
        int i;
        struct node *temp=head;
        if(position==1)
    {
        ptr->next=temp;
                head=ptr;
                return;
        }
        for(i=1;i<position-1;i++)  //moving to the (n-1)th position node in the linked list
        {
        temp=temp->next;
        }
        ptr->next=temp->next;  //Make the newly created node point to next node of ptr temp
        temp->next=ptr;              //Make ptr temp point to newly created node in the linked list
}
```



# Complete code

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

// A linked list node
struct Node
{
int data;
struct Node *next;
};

/* Given a reference (pointer to pointer) to the head of a list and
an int, inserts a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
        /* 1. allocate node */
        struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
        /* 2. put in the data */
        new_node->data = new_data;
        /* 3. Make next of new node as head */
        new_node->next = (*head_ref);
        /* 4. move the head to point to the new node */
        (*head_ref) = new_node;
}
```

```
/* Given a reference (pointer to pointer) to the head
of a list and an int, appends a new node at the end */
void append(struct Node** head_ref, int new_data)
{
        /* 1. allocate node */
        struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
        struct Node *last = *head_ref; /* used in step 5*/
        /* 2. put in the data */
        new_node->data = new_data;
        /* 3. This new node is going to be the last node, so make next of
                it as NULL*/
        new_node->next = NULL;
        /* 4. If the Linked List is empty, then make the new node as head */
        if (*head_ref == NULL)
        {
        *head_ref = new_node;
        return;
        }
        /* 5. Else traverse till the last node */
        while (last->next != NULL)
                last = last->next;
        /* 6. Change the next of last node */
        last->next = new_node;
        return;
}
```

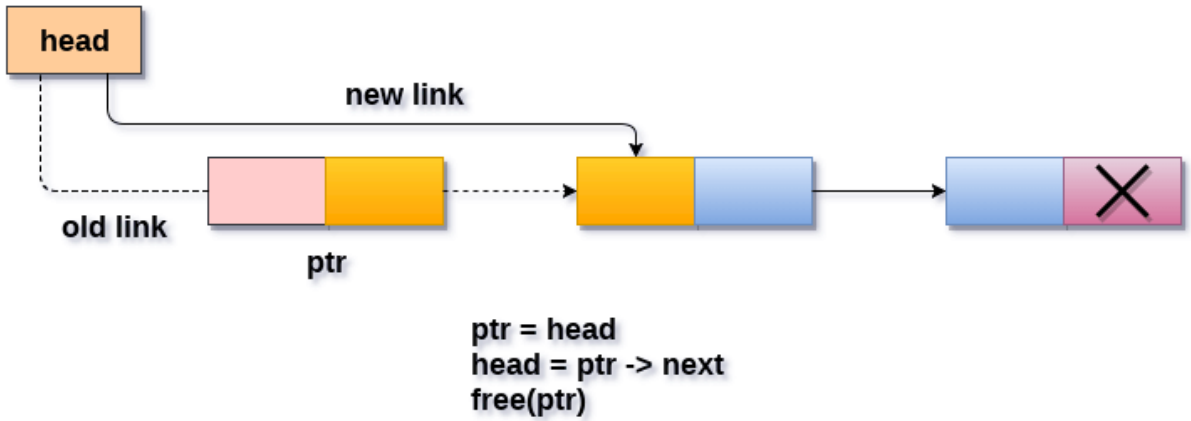
```
// This function prints contents of linked list starting from head
void printList(struct Node *node)
{
while (node != NULL)
{
        printf(" %d ", node->data);
        node = node->next;
}
}

/* Driver program to test above functions*/
int main()
{
/* Start with the empty list */
struct Node* head = NULL;
// Insert 6. So linked list becomes 6->NULL
append(&head, 6);
// Insert 7 at the beginning. So linked list becomes 7->6->NULL
push(&head, 7);
// Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
push(&head, 1);
// Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
append(&head, 4);
printf("\n Created Linked list is: ");
printList(head);
return 0;
}
```

# Singly Linked List:Deleting a node

- A node can be deleted in three ways
  **1)** At the front of the linked list
  **2)** After a given node/specified position
  **3)** At the end of the linked list.

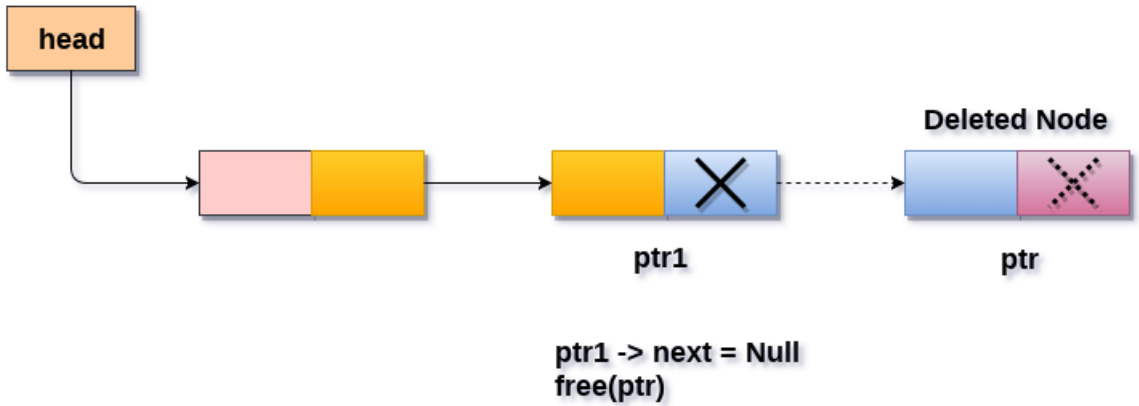
# Delete a node at the front

```
void Pop()
  {
     struct node *ptr;
     if(head == NULL)
     {
        printf("\nList is empty");
     }
     else
     {
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\n Node deleted from the begining ...");
     }
  }
```



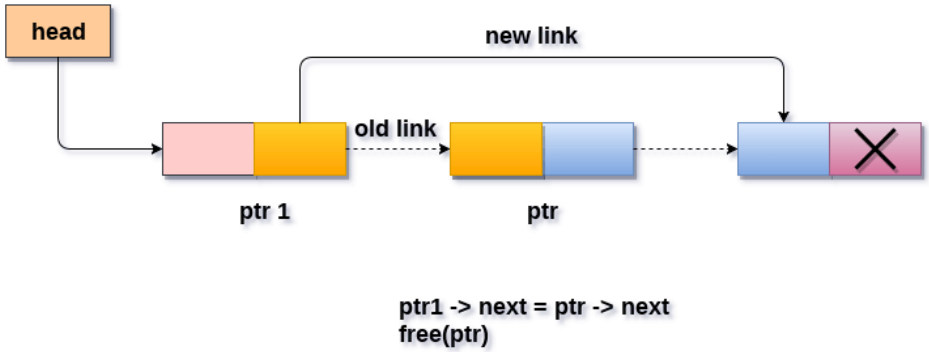
# Delete a node at the end

```
void end_delete()
  {
     struct node *ptr,*ptr1;
     if(head == NULL)
     {
        printf("\nlist is empty");
     }
     else if(head -> next == NULL)
     {
        free(head);
       head = NULL;
       printf("\nOnly node of the list deleted ...");
     }

     else
     {
        ptr = head;
        while(ptr->next != NULL)
           {
              ptr1 = ptr;
              ptr = ptr ->next;
           }
           ptr1->next = NULL;
           free(ptr);
           printf("\n Deleted Node from the last ...");
        }
     }
```



# Delete a node at specified position

```
void delete_specified()
  {
     struct node *ptr, *ptr1;
     int loc,i;
     scanf("%d",&loc);
     ptr=head;
     for(i=0;i<loc;i++)
     {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
           printf("\nThere are less than %d elements in the list..\n",loc);
           return;
        }
     }
     ptr1 ->next = ptr ->next;
     free(ptr);
     printf("\nDeleted %d node ",loc);
  }
```



# Complete Code

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
    int choice =0;
    while(choice != 9)
    {
        printf("\n\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
        printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n
        5.Delete from last\n6.Delete node after specified location\n7.Search for an element\n8.Show\n9.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            beginsert();
            break;
            case 2:
            lastinsert();
            break;
            case 3:
            randominsert();
            break;
            case 4:
            begin_delete();
            break;
            case 5:
            last_delete();
            break;
            case 6:
            random_delete();
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
```

```
void beginsert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node *));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value\n");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nNode inserted");
    }

}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            ptr -> next = NULL;
            head = ptr;
            printf("\nNode inserted");
        }
        else
        {
            temp = head;
            while (temp -> next != NULL)
            {
                temp = temp -> next;
            }
            temp->next = ptr;
            ptr->next = NULL;
            printf("\nNode inserted");

        }
    }
}
```

```
void randominsert()
{
    int i,loc,item;
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }

        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
    }
}
void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nList is empty\n");
    }
    else
    {
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\nNode deleted from the begining ...\n");
    }
}
```

```
void last_delete()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nOnly node of the list deleted ...\n");
    }

    else
    {
        ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
        ptr1->next = NULL;
        free(ptr);
        printf("\nDeleted Node from the last ...\n");
    }
}
void random_delete()
{
    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which you want to perform deletion \n");
    scanf("%d",&loc);
    ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node %d ",loc+1);
}
```

```
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }

}

void display()
{
    struct node *ptr;
    ptr = head;
    if(ptr == NULL)
    {
        printf("Nothing to print");
    }
    else
    {
        printf("\nprinting values . . . . .\n");
        while (ptr!=NULL)
        {
            printf("\n%d",ptr->data);
            ptr = ptr -> next;
        }
    }
}
```

# Singly Linked List Operations

- Search
- Count number of nodes
- Concatenation
- Merging
- Reversing

# Search

```
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }

}
```

# Count number of nodes

```
void Count_nodes()
{
/* temp pointer points to head */
      struct node* temp = head;
/* Initialize count variable  */
      int count=0;
/* Traverse the linked list and maintain the count */
      while(temp != NULL)
      {
      temp = temp->next;
      /* Increment count variable. */
      count++;
     }
/* Print the total count. */
printf("\n Total no. of nodes is %d",count);
}
```

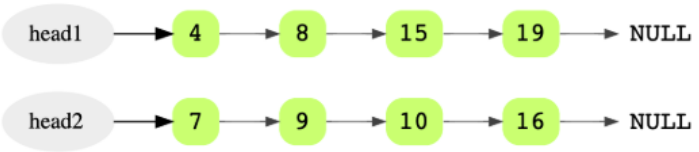
# Concatenation

```
void concatenate(struct node *a,struct node *b)
{
    if( a != NULL && b!= NULL )
    {
        if (a->next == NULL)
            a->next = b;
        else
            concatenate(a->next,b);
    }
    else
    {
        printf("Either a or b is NULL\n");
    }
}

struct node *concat( struct node *start1,struct node *start2)
{
        struct node *ptr;
        if(start1==NULL)
        {
                start1=start2;
                return start1;
        }
        if(start2==NULL)
                return start1;
        ptr=start1;
        while(ptr->link!=NULL)
                ptr=ptr->link;
        ptr->link=start2;
        return start1;
}
```

# Merging

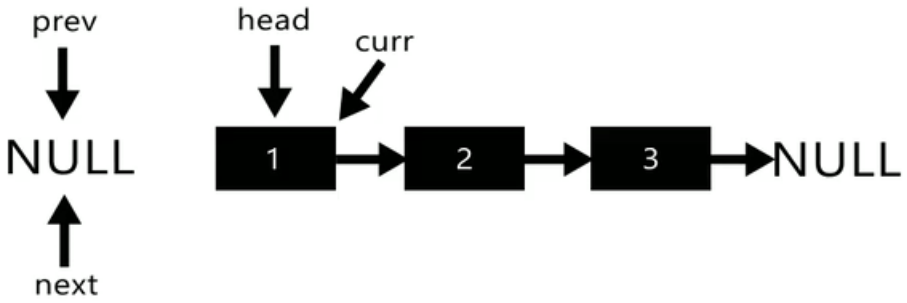
```
NodePtr merge_sorted(NodePtr head1, NodePtr head2) {

  // if both lists are empty then merged list is also empty
  // if one of the lists is empty then other is the merged list
  if (head1 == nullptr) {
    return head2;
  } else if (head2 == nullptr) {
    return head1;
  }


  NodePtr mergedHead = nullptr;
  if (head1->data <= head2->data) {
    mergedHead = head1;
    head1 = head1->next;
  } else {
    mergedHead = head2;
    head2 = head2->next;
  }

  NodePtr mergedTail = mergedHead;

  while (head1 != nullptr && head2 != nullptr) {
    NodePtr temp = nullptr;
    if (head1->data <= head2->data) {
      temp = head1;
      head1 = head1->next;
    } else {
      temp = head2;
      head2 = head2->next;
    }

    mergedTail->next = temp;
    mergedTail = temp;
  }

  if (head1 != nullptr) {
    mergedTail->next = head1;
  } else if (head2 != nullptr) {
    mergedTail->next = head2;
  }

  return mergedHead;
}
```

Consider two sorted linked lists as an example.




The merged linked list should look like this:

# Reversing



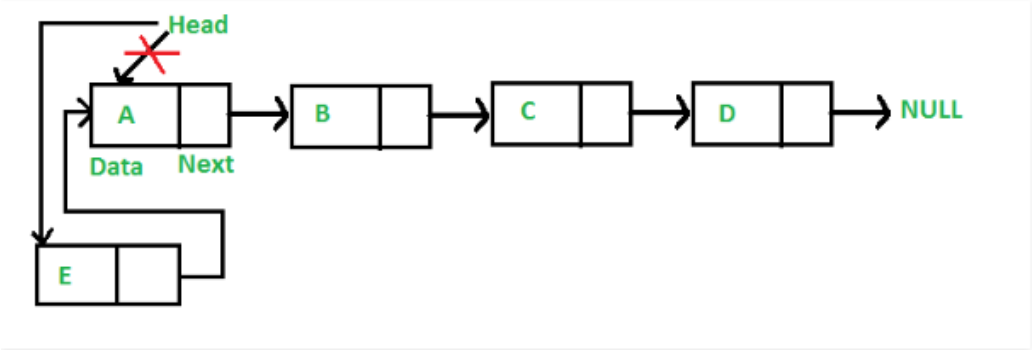

```
while (current != NULL)
     {
          next  = current->next;
          current->next = prev;
          prev = current;
          current = next;
     }
     *head_ref = prev;
```

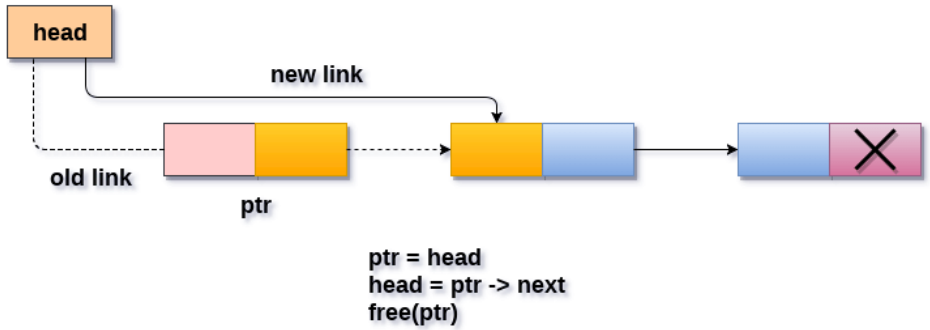
# Applications of Linked List

- Stack
- Queue implementation

# Stack Implementation using Linked List

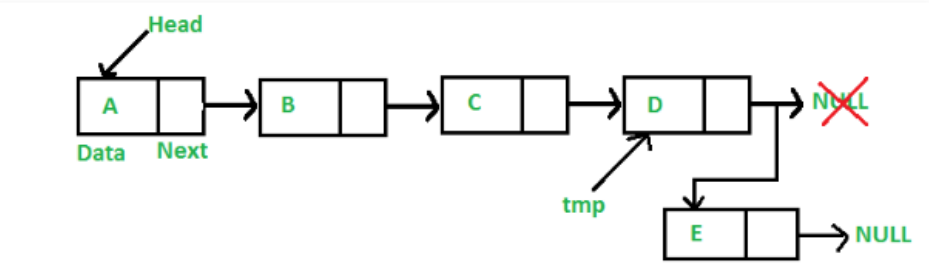
```
void push(struct Node** head_ref, int new_data)
{
        struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
        new_node->data = new_data;
        new_node->next = (*head_ref);
        (*head_ref) = new_node;
}

void Pop()
   {
      struct node *ptr;
      if(head == NULL)
      {
         printf("\nList is empty");
      }
      else
      {
         ptr = head;
         head = ptr->next;
         free(ptr);
         printf("\n Node deleted from the begining ...");
      }

   }
```

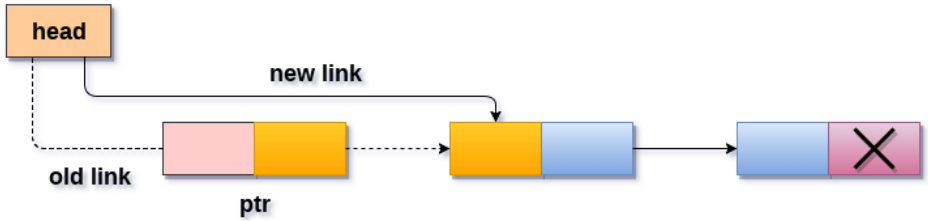


# Queue implementation using Linked List

```
void Enqueue(item)
{
   struct node *ptr,*temp;

   ptr = (struct node*)malloc(sizeof(struct node));
   ptr->data = item; ptr -> next = NULL;
      if(head == NULL)
      {
         head = ptr;
         printf("\nNode inserted");
      }
      else
      {
         temp = head;
         while (temp -> next != NULL)
         {
            temp = temp -> next;
         }
         temp->next = ptr;
         printf("\nNode inserted");
        }
   }
  Void Dequeue()
   {
      struct node *ptr;
      if(head == NULL)
      {
         printf("\nList is empty");
      }
      else
      {
         ptr = head;
         head = ptr->next;
         free(ptr);
         printf("\n Node deleted from the begining ...");
      }
```

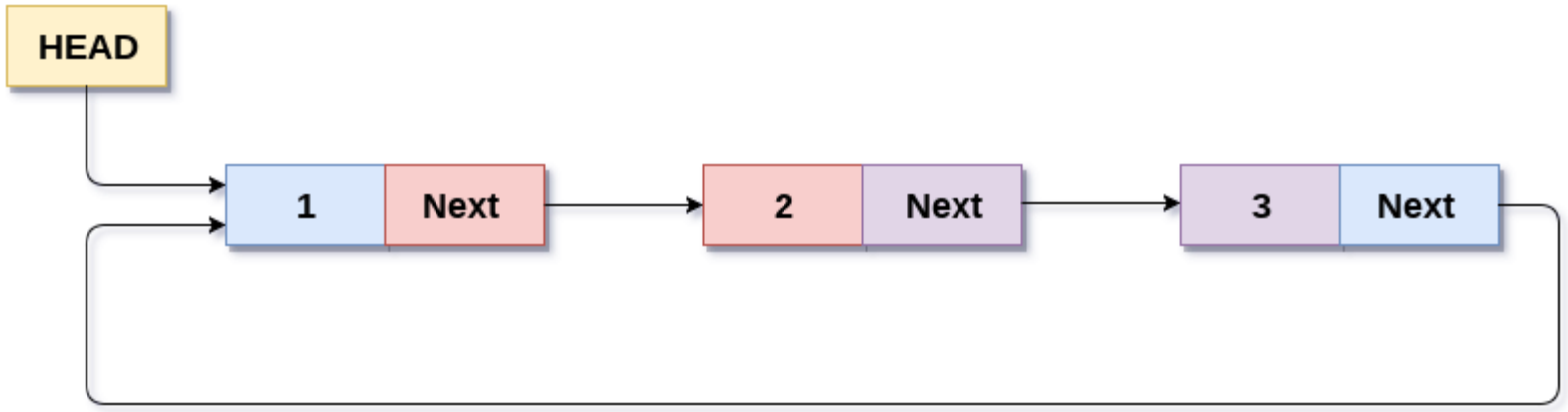



ptr = head
head = ptr -> next
free(ptr)

# Circular Linked list

- In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
- We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

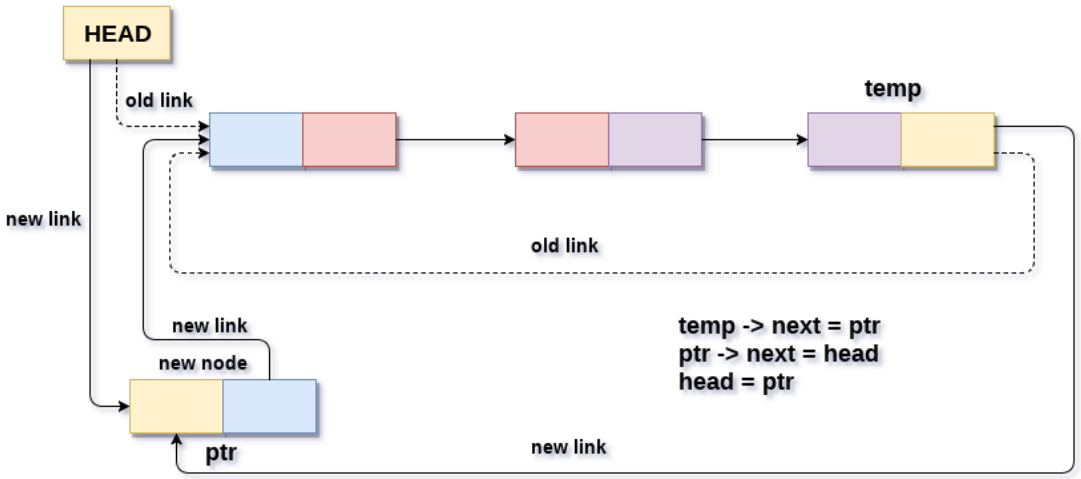


# Operations on Circular List

- Insertion at the beginning
- Insertion at the end
- Deletion at the beginning
- Deletion at the end
- Searching
- Traversing

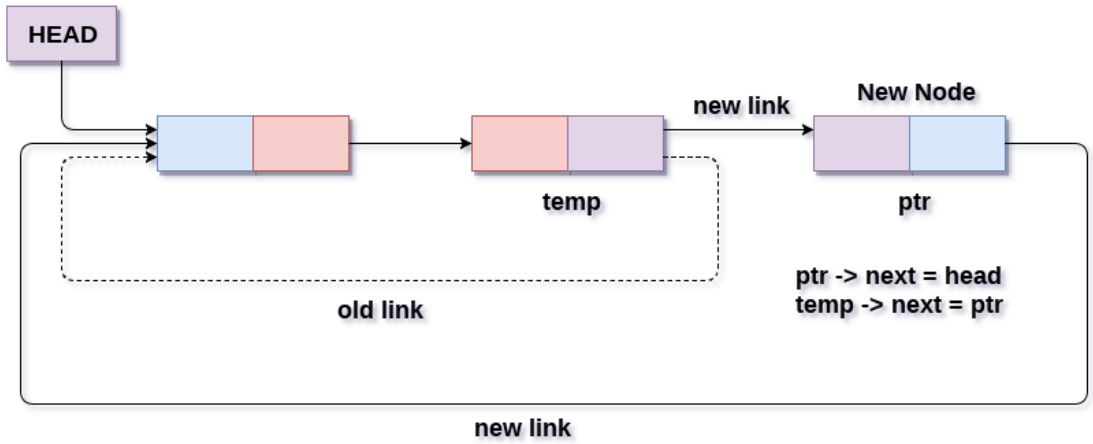
# Insertion at the beginning

```
void beg_insert(int item)
{

   struct node *ptr = (struct node *)malloc(sizeof(struct node));
   struct node *temp;
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      ptr -> data = item;
      if(head == NULL)
      {
         head = ptr;
         ptr -> next = head;
      }
      else
      {
         temp = head;
         while(temp->next != head)
            temp = temp->next;
         ptr->next = head;
         temp -> next = ptr;
         head = ptr;
      }
   printf("\nNode Inserted\n");
   }
```



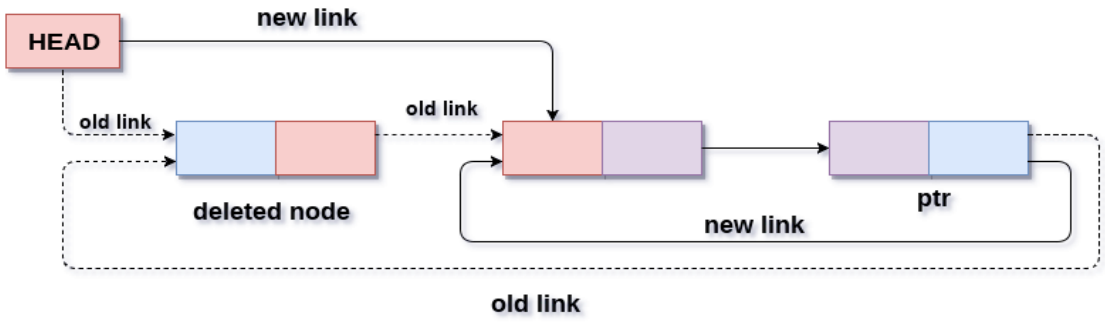
# Insertion at the end

```
void lastinsert(struct node*ptr, struct node *temp, int item)
{
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW\n");
   }
   else
   {
      ptr->data = item;
      if(head == NULL)
      {
         head = ptr;
         ptr -> next = head;
      }
      else
      {
         temp = head;
         while(temp -> next != head)
         {
            temp = temp -> next;
         }
         temp -> next = ptr;
         ptr -> next = head;
      }
   }
```



# Deletion at the beginning

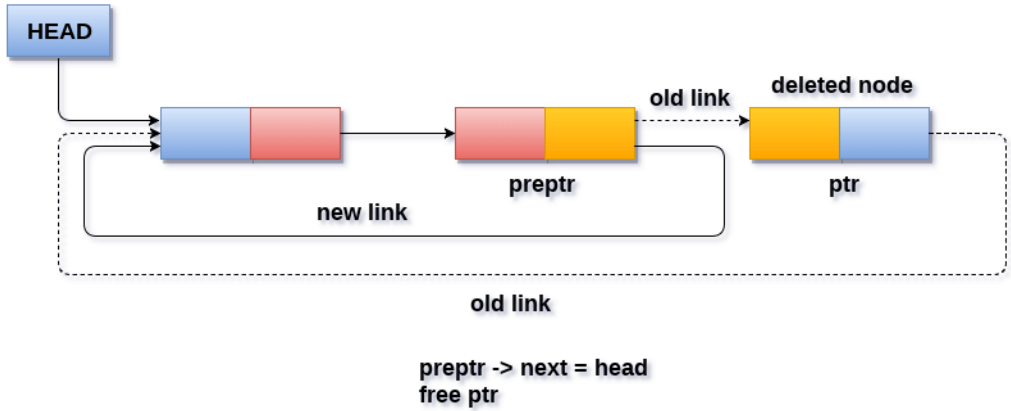
```
void beg_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\nUNDERFLOW\n");
   }
   else if(head->next == head)
   {
      head = NULL;
      free(head);
      printf("\nNode Deleted\n");
   }

   else
   {
      ptr = head;
      while(ptr -> next != head)
         ptr = ptr -> next;
      ptr->next = head->next;
      free(head);
      head = ptr->next;
      printf("\nNode Deleted\n");
   }
}
```




**ptr -> next = head -> next**
**free head**
**head = ptr -> next**

# Deletion at the end

```
void last_delete()
{
struct node *ptr, *preptr;
   if(head==NULL)
   {
      printf("\nUNDERFLOW\n");
   }
   else if (head ->next == head)
   {
      head = NULL;
      free(head);
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = head;
      while(ptr ->next != head)
      {
         preptr=ptr;
         ptr = ptr->next;
      }
      preptr->next = ptr -> next;
      free(ptr);
      printf("\nNode Deleted\n");
   }
}
```



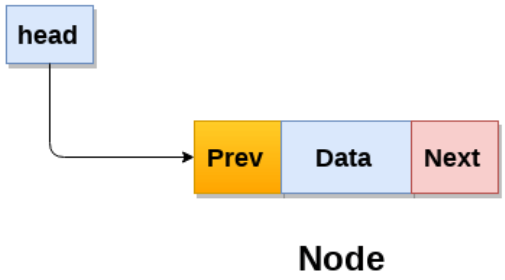
# Searching

```
void search()
{
   struct node *ptr;
   int item,i=0,flag=1;
   ptr = head;
   if(ptr == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      printf("\nEnter item which you want to search?\n");
      scanf("%d",&item);
      if(head ->data == item)
      {
      printf("item found at location %d",i+1);
      flag=0;
      return;
      }
      else
      {
      while (ptr->next != head)
      {
         if(ptr->data == item)
         {
            printf("item found at location %d ",i+1);
            flag=0;
            return;
         }
         else
         {
            flag=1;
         }
         i++;
         ptr = ptr -> next;
      }
      }
      if(flag != 0)
      {
         printf("Item not found\n");
         return;
      }
•           }
```

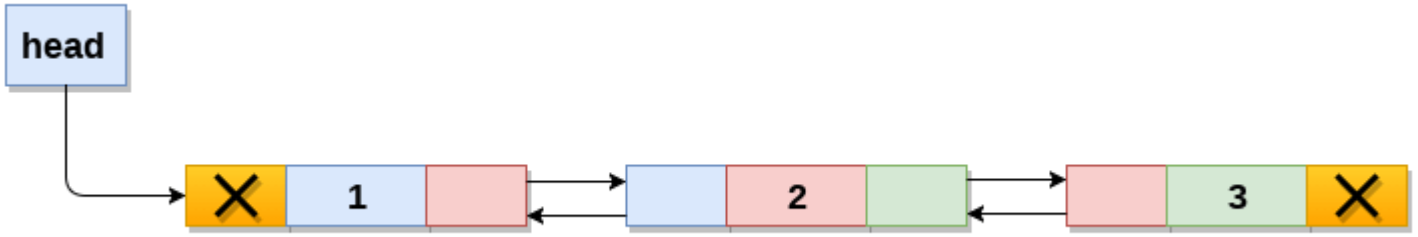
# Traversing

```
void traverse()
{
   struct node *ptr;
   ptr=head;
   if(head == NULL)
   {
      printf("\nnothing to print");
   }
   else
   {
      printf("\n printing values ... \n");

      while(ptr -> next != head)
      {

         printf("%d\n", ptr -> data);
         ptr = ptr -> next;
      }
      printf("%d\n", ptr -> data);
   }

}
```

# Doubly Linked List




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

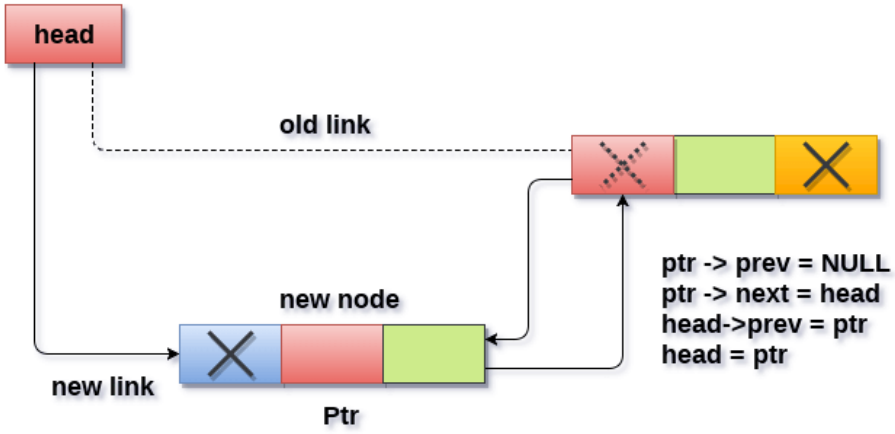


# Operations on Doubly Linked List

- Insertion at beginning
- **Insertion at end**
- Insertion after specified node
- Deletion at beginning
- Deletion at end
- **Deletion of the node given data**
- Searching
- Traversing

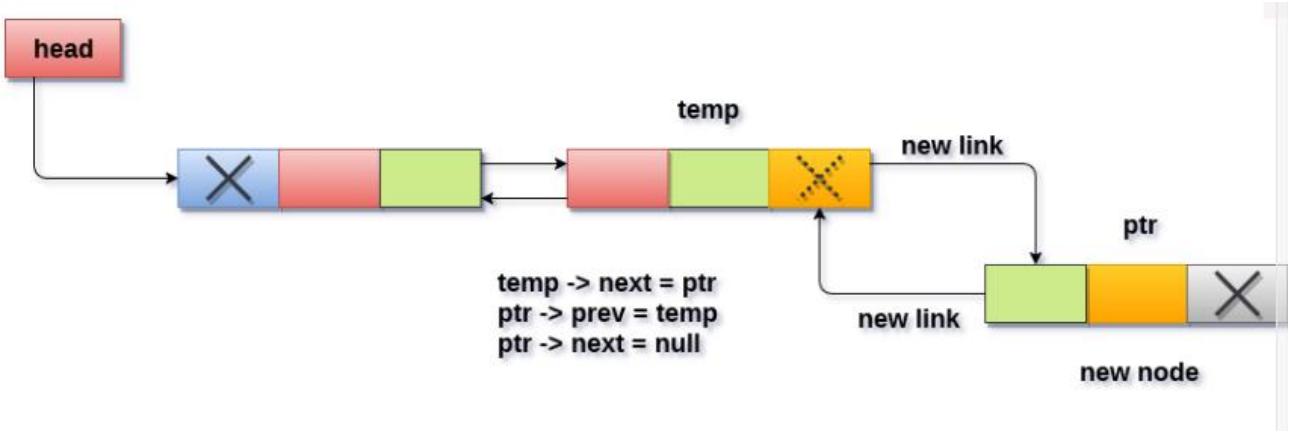
# Insertion at beginning

```
void insertbeginning(int item)
{

  struct node *ptr = (struct node *)malloc(sizeof(struct node));

  if(head==NULL)
  {
     ptr->next = NULL;
     ptr->prev=NULL;
     ptr->data=item;
     head=ptr;
  }
  else
  {
     ptr->data=item;
     ptr->prev=NULL;
     ptr->next = head;
     head->prev=ptr;
     head=ptr;
  }
}
```



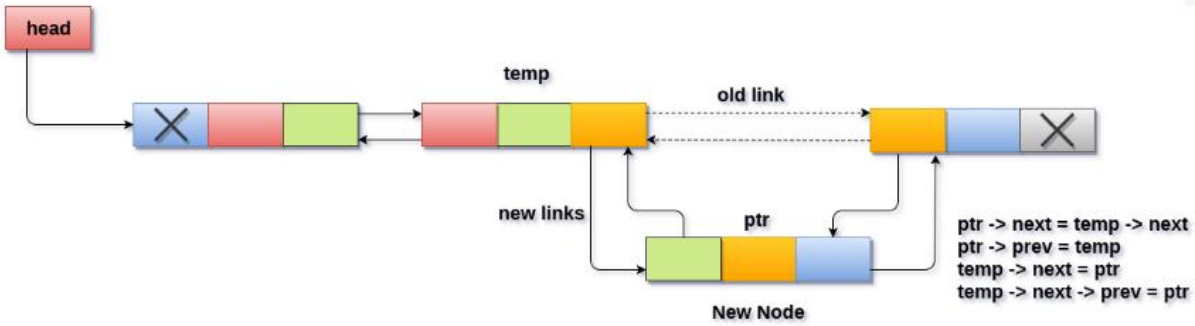
# Insertion at end

```
void insertlast(int item)
{

  struct node *ptr = (struct node *) malloc(sizeof(struct node));
  struct node *temp;

      ptr->data=item;
     if(head == NULL)
     {
        ptr->next = NULL;
        ptr->prev = NULL;
        head = ptr;
     }
     else
     {
       temp = head;
       while(temp->next!=NULL)
       {
          temp = temp->next;
       }
       temp->next = ptr;
       ptr ->prev=temp;
       ptr->next = NULL;
     }
printf("\nNode Inserted\n");

}
```



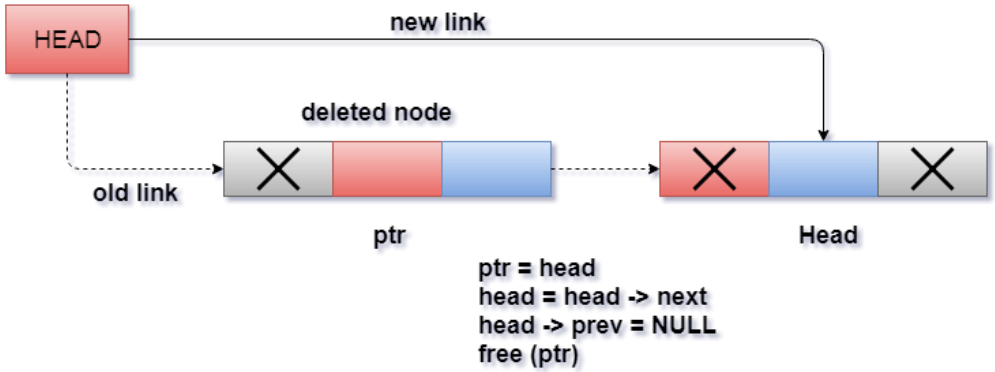
# Insert after specified node

```
void insert_specified(int item)
{
  struct node *ptr = (struct node *)malloc(sizeof(struct node));
  struct node *temp;
  int i, loc;
   printf("\nEnter the location\n");
     scanf("%d",&loc);
     temp=head;
     for(i=0;i<loc;i++)
     {
        temp = temp->next;
        if(temp == NULL)
        {
           printf("\ncan't insert\n");
           return;
        }
     }
     ptr->data = item;
     ptr->next = temp->next;
     ptr -> prev = temp;
     temp->next = ptr;
     temp->next->prev=ptr;
     printf("Node Inserted\n");
  }
}
```



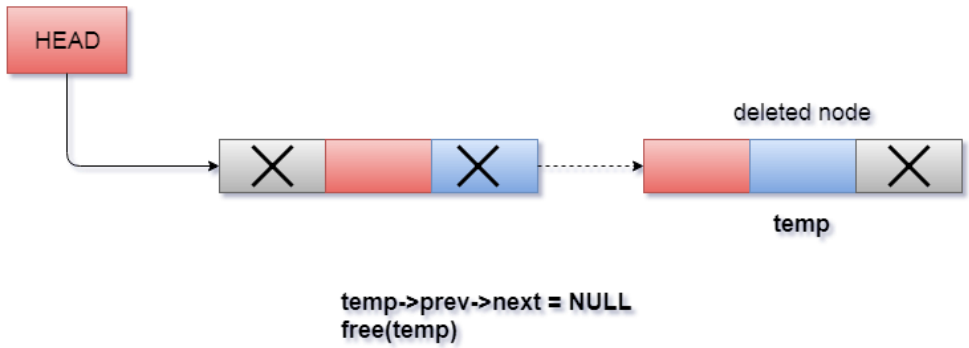
# Deletion at beginning

```
void beginning_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW\n");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = head;
      head = head -> next;
      head -> prev = NULL;
      free(ptr);
      printf("\nNode Deleted\n");
   }
}
```



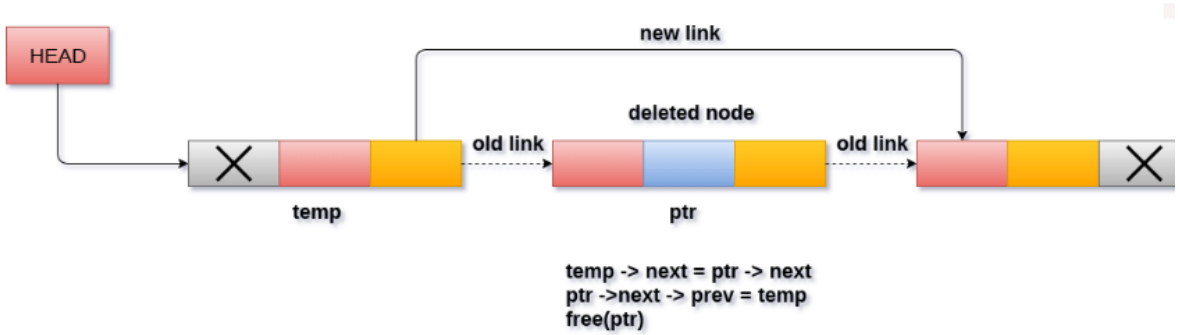
# Deletion at end

```
void last_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW\n");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = head;
      if(ptr->next != NULL)
      {
         ptr = ptr -> next;
      }
      ptr -> prev -> next = NULL;
      free(ptr);
      printf("\nNode Deleted\n");
   }
}
```



# Deletion of a specified node

```
void delete_specified( )
{
   struct node *ptr, *temp;
   int val;
   printf("Enter the value");
   scanf("%d",&val);
   temp = head;
   while(temp -> data != val)
   temp = temp -> next;
   if(temp -> next == NULL)
   {
      printf("\nCan't delete\n");
   }
   else if(temp -> next -> next == NULL)
   {
      temp ->next = NULL;
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = temp -> next;
      temp -> next = ptr -> next;
      ptr -> next -> prev = temp;
      free(ptr);
      printf("\nNode Deleted\n");
   }
}
```



# Searching

```
void search()
{
   struct node *ptr;
   int item,i=0,flag;
   ptr = head;
   if(ptr == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      printf("\nEnter item which you want to search?\n");
      scanf("%d",&item);
      while (ptr!=NULL)
      {
         if(ptr->data == item)
         {
            printf("\nitem found at location %d ",i+1);
            flag=0;
            break;
         }
         else
         {
            flag=1;
         }
         i++;
         ptr = ptr -> next;
      }
      if(flag==1)
      {
         printf("\nItem not found\n");
      }
   }

}
```

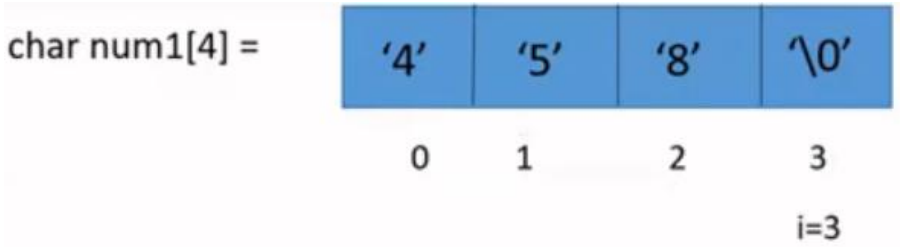
# Traversing

```
int traverse()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      ptr = head;
      while(ptr != NULL)
      {
         printf("%d\n",ptr->data);
         ptr=ptr->next;
      }
   }
}
```

# Example1: Addition of long positive integers

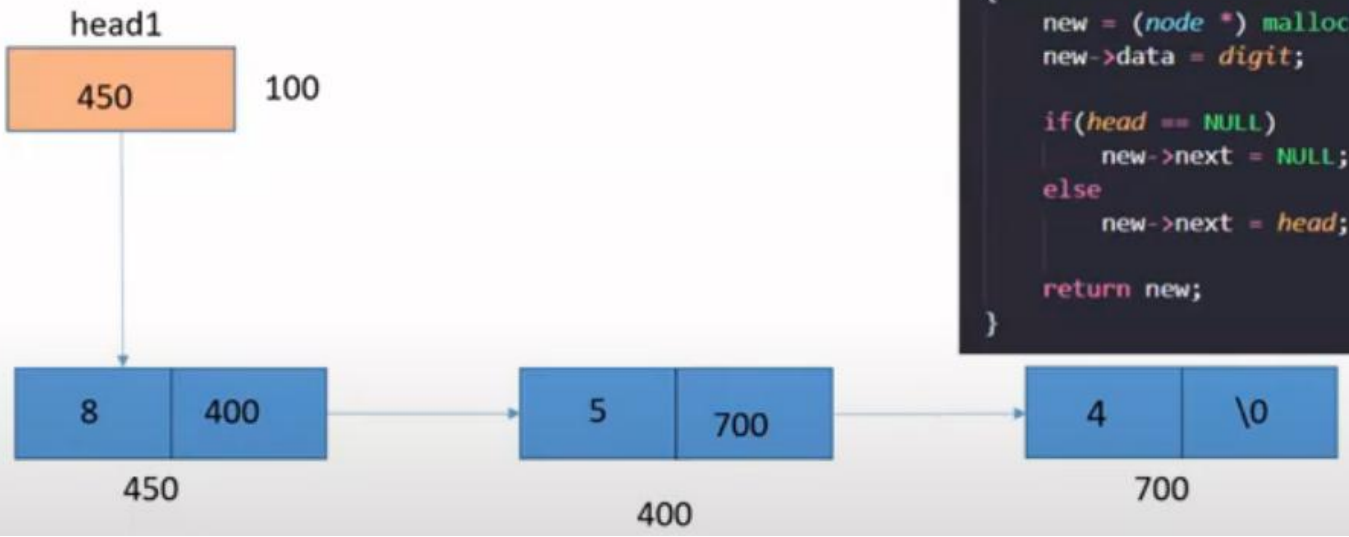
- Algorithm:
  - Read the first long integers as string.
    - Read each character from the string
    - Convert the character to integer
    - Create a new node and store each integer in that node
    - Insert each node at the beginning of the linked list.
  - Repeat the first step for the second number.

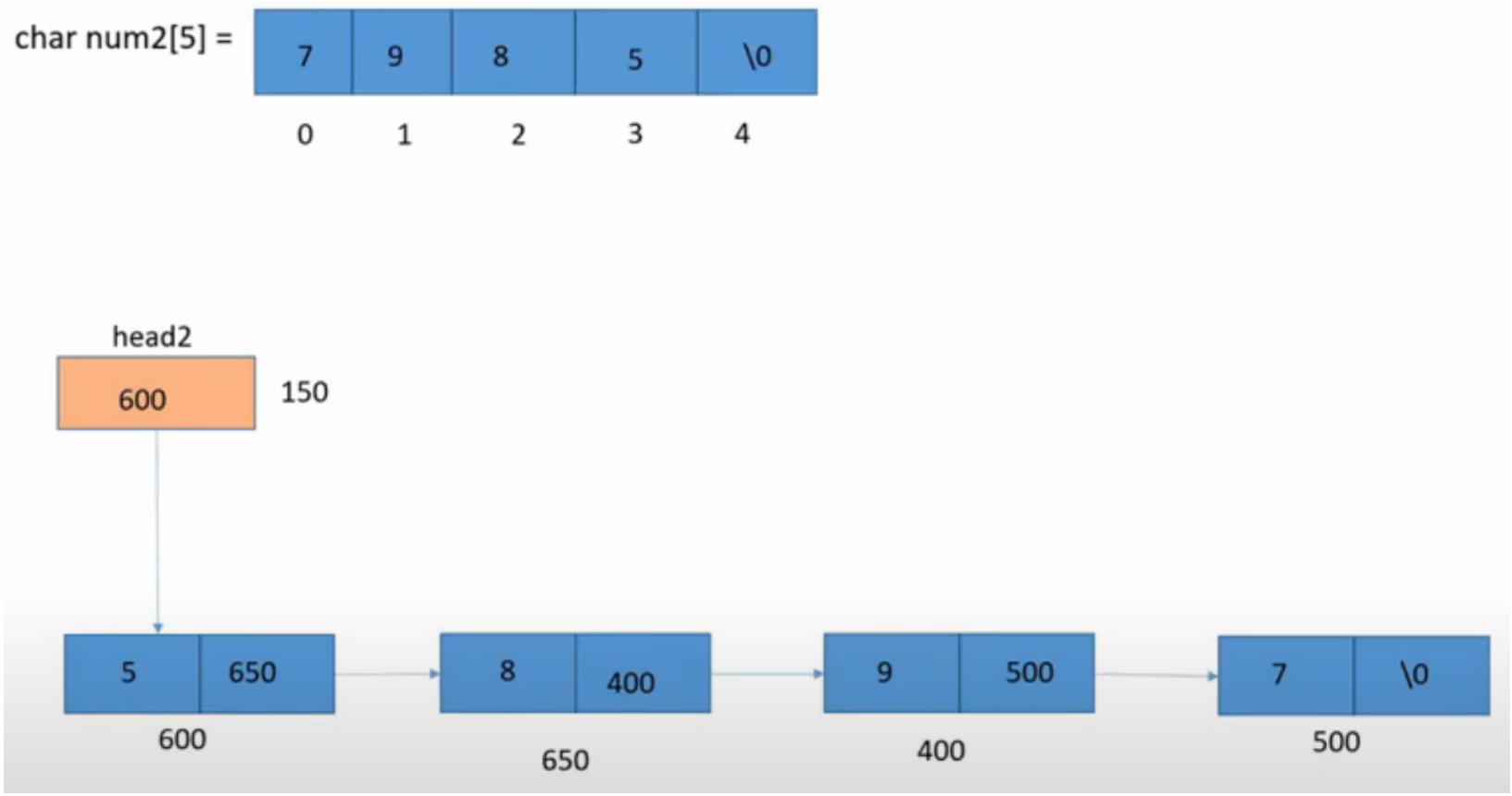
- Traverse the two linked lists from start to end
- Add the two digits each from respective linked lists.
- If one of the list has reached the end then take 0 as its digit.
- Continue it until both the lists end.
- If the sum of two digits is greater than 9 then set carry as 1 and the current digit as *sum % 10*

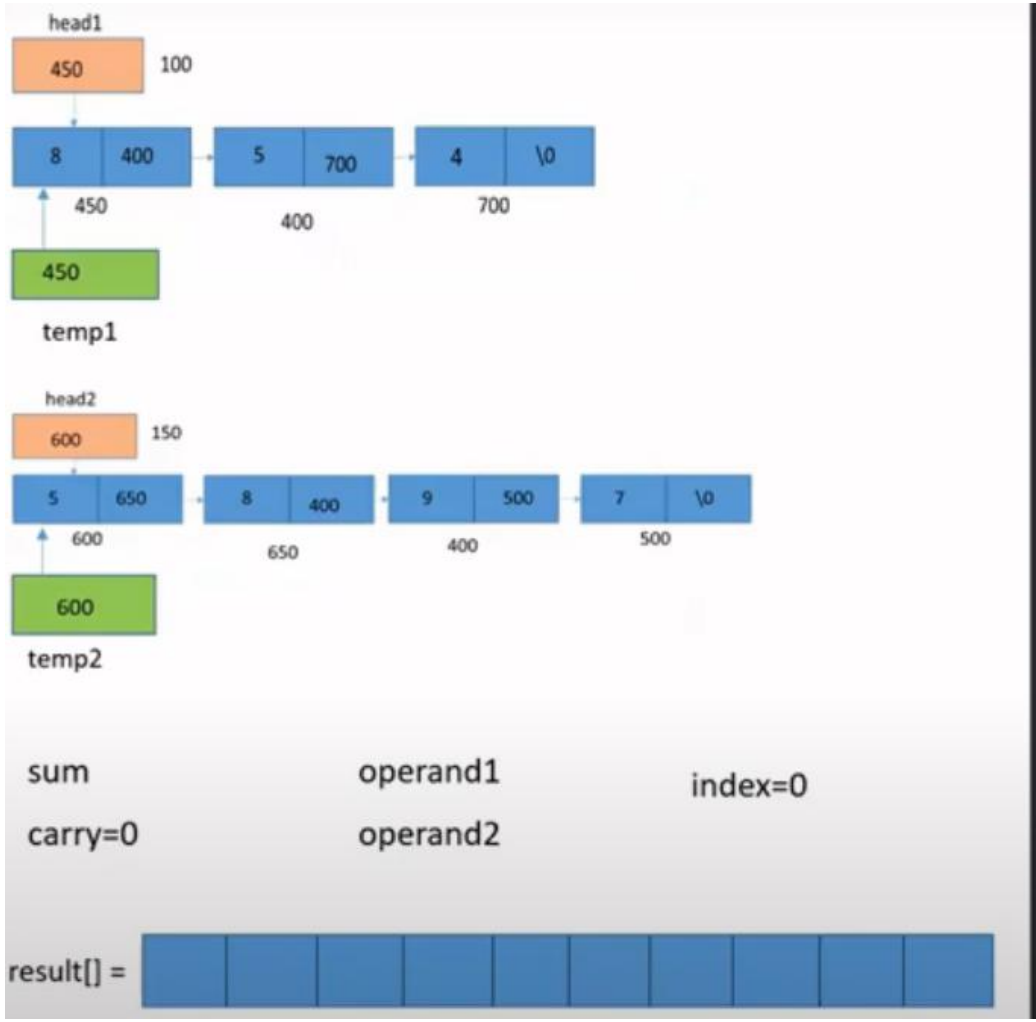



```c
i=0;
while(num1[i]!='\0')   //creating linked list for 1st no.
{
    digit = num1[i]-48;
    head1 = insertNode(head1,digit);
    i++;
}
```

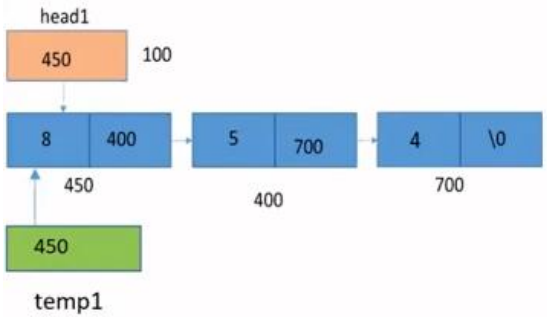
```c
node* insertNode(node *head, int digit)
{
    new = (node *) malloc(sizeof(node));
    new->data = digit;

    if(head == NULL)        //for inserting the first node
        new->next = NULL;
    else
        new->next = head;

    return new;
}
```

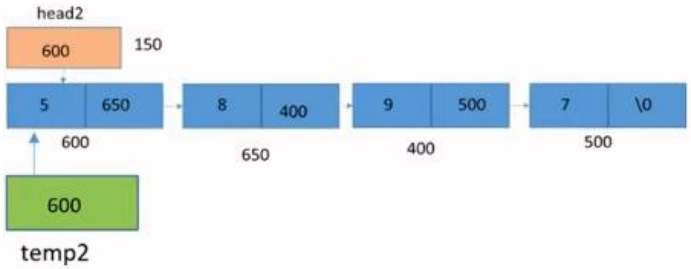



char num2[5] =
7
9
8
5
\0
0
1
2
3
4
head2
600
150
5
650
8
400
9
500
7
\0
600
650
400
500




```
void add(char result[])
{
    int sum,carry=0,operand1,operand2,index=0;

    temp1 = head1;
    temp2 = head2;
    while (temp1!=NULL || temp2!=NULL)
    {
        (temp1==NULL)? operand1=0 : (operand1=temp1->data);
        (temp2==NULL)? operand2=0 : (operand2=temp2->data);

        sum = operand1 + operand2 + carry;

        if(sum>=10)
        {
            carry = 1;
            sum = sum%10;
        }
        else
            carry = 0;

        result[index] = sum + '0';
        index++;

        if(temp1!=NULL)
            temp1 = temp1->next;
        if(temp2!=NULL)
            temp2 = temp2->next;
    }
    result[index] = '\0';
```



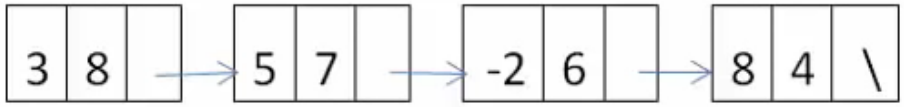

sum = 3 operand1 = 8 index=1

carry= 1 operand2 = 5

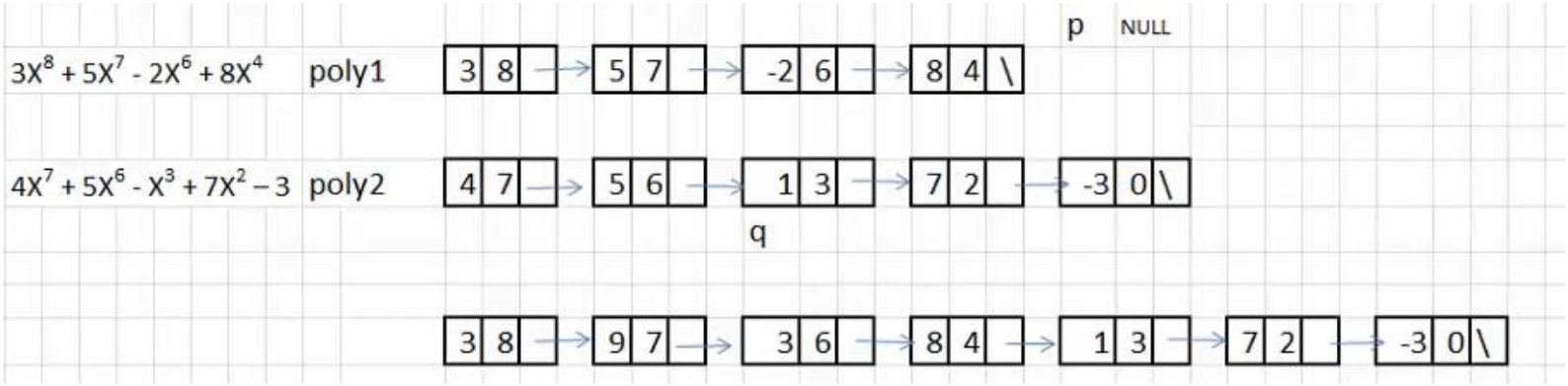

```c
void add(char result[])
{
    int sum,carry=0,operand1,operand2,index=0;

    temp1 = head1;
    temp2 = head2;
    while (temp1!=NULL || temp2!=NULL)
    {
        (temp1==NULL)? operand1=0 : (operand1=temp1->data);
        (temp2==NULL)? operand2=0 : (operand2=temp2->data);

        sum = operand1 + operand2 + carry;

        if(sum>=10)
        {
            carry = 1;
            sum = sum%10;
        }
        else
            carry = 0;

        result[index] = sum + '0';
        index++;

        if(temp1!=NULL)
            temp1 = temp1->next;
        if(temp2!=NULL)
            temp2 = temp2->next;
    }
    result[index] = '\0';
    strrev(result);
}
```

# Example2: Adding Polynomials

$3X^8 + 5X^7 - 2X^6 + 8X^4$




$4X^7 + 5X^6 - X^3 + 7X^2 - 3$



p
NULL
$3X^8 + 5X^7 - 2X^6 + 8X^4$
poly1
3 8
5 7
-2 6
8 4 \
$4X^7 + 5X^6 - X^3 + 7X^2 - 3$
poly2
4 7
5 6
1 3
7 2
-3 0 \
q
3 8
9 7
3 6
8 4
1 3
7 2
-3 0 \

```c
struct node
{
    int coef;
    int exp;
    struct node *next;
};
```

```c
struct node * GetNode()
{
    struct node *p;
    p=(struct node *)malloc(sizeof(struct node));
    return p;
}
```

```c
InsBeg(struct node **list, int c,int e)
{
struct node *temp;
temp=GetNode();
temp->coef=c;
temp->exp=e;
temp->next=*list;
*list=temp;
}
```

```c
Traverse(struct node *list)
{
    struct node *t;
    t=list;
    while(t!=NULL)
    {
        printf("\t %dX%d +",t->coef,t->exp);
        t=t->next;
    }
}
```

```c
void main()
{
    struct node *Start1,*Start2,*Start3;
    Start1=NULL;
    Start2=NULL;
    Start3=NULL;

    int x;

    InsEnd(&Start1,3,8);
    InsEnd(&Start1,5,7);
    InsEnd(&Start1,-2,6);
    InsEnd(&Start1,8,4);
    printf("\nFirst Polynomial is:=> ");
    Traverse(Start1);

    InsEnd(&Start2,4,7);
    InsEnd(&Start2,5,6);
    InsEnd(&Start2,-1,3);
    InsEnd(&Start2,7,2);
    InsEnd(&Start2,-3,0);
    printf("\n\n");
    printf("Second Polynomial is:=> ");
    Traverse(Start2);

    START3=AddPoynomial(START1,START2);

    printf("\n\n")
    printf("Result of Polynomial Addition is:=> ")
    Traverse(Start3);
}
```

```
struct node * AddPolynomial(struct node *poly1, struct node *poly2
{
    struct node *poly3=NULL;
    struct node *p,*q;
    p=poly1;
    q=poly2;

    while(p!=NULL&&q!=NULL)
    {
        if(p->exp==q->exp)
        {
            InsEnd(&poly3,p->coef+q->coef,p->exp);
            p=p->next;
            q=q->next;
        }
        else
        {
            if(p->exp>q->exp)
            {
                InsEnd(&poly3,p->coef,p->exp);
                p=p->next;
            }
            else
            {
                InsEnd(&poly3,q->coef,q->exp);
                q=q->next;
            }

        }
    }

    while(p!=NULL)
    {
        InsEnd(&poly3,p->coef,p->exp);
            p=p->next;
    }

    while(q!=NULL)
    {
        InsEnd(&poly3,q->coef,q->exp);
            q=q->next;
    }
```