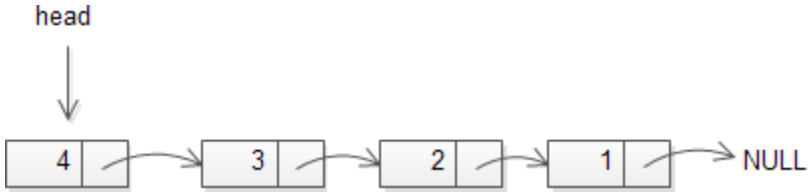
## SINGLY LINKED LIST

A singly linked list is a data structure consisting of a sequence of nodes ***where each data points to the next data.*** Each node stores,

- Data (element)
- pointer to the next node




**Singly Linked List in C**

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

The node structure has two members:

- data stores the information
- next pointer holds the address of the next node.

**Basic operation of singly list list.**

- Create Linked List
- Traverse a linked list
- Insert elements to linked list
- Delete from a linked list

**Creating a Node**

To create the node, **sizeof()** is used to determine size in bytes of an element in C and **malloc()** is used to dynamically allocate a single block of memory in C.

```
struct LinkedList *node; //Define node as pointer of data type struct LinkedList
node    createNode()
{
         node temp; // declare a node
        temp = (node)malloc(sizeof(struct LinkedList));
```

```
        temp->next = NULL;// make next point to NULL
        return temp;//return the new node
    }
```

**Traverse a linked list**

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

```
    struct node *temp = head;
    printf("\n\nList elements are - \n");
    while(temp != NULL)
    {
        printf("%d --->",temp->data);
        temp = temp->next;
    }
```

**Insert elements to linked list**

We can insert elements to either beginning, middle or end of linked list.

**Insert at beginning**

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));

newNode->data = 4;
newNode->next =
head;
head = newNode;
```

**Insert at end**

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;
struct node *temp = head;
while(temp->next != NULL)
{
        temp = temp->next;
}
temp->next = newNode;
```

**Insert to middle**

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
struct node *temp = head;
for(int i=2; i < position; i++)
{
        if(temp->next != NULL)
        {
                temp = temp->next;
```

```
        }
    }
    newNode->next = temp->next;
    temp->next = newNode;
```

**Delete from a linked list**

We can delete either from beginning, end or from a particular position.

**Delete from beginning**

- Point head to the second node

```
head = head->next;
```

**Delete from end**

- Traverse to second last element
- Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
temp = temp->next;
}
temp->next = NULL;
```

**Delete from middle**

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
 for(int i=2; i< position; i++)
 {
        if(temp->next!=NULL)
        {
                temp = temp->next;
        }
 }
```

**Program for Singly Linked List:**

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

```
struct node
{
        int data;
        struct node *next;
};
void display(struct node* head)
{
        struct node *temp = head;
        printf("\n\nList elements are - \n");
        while(temp != NULL)
        {
                printf("%d --->",temp->data);
                temp = temp->next;
        }
}
void insertAtMiddle(struct node *head, int position, int value)
{
        struct node *temp = head;
        struct node *newNode;
        newNode = malloc(sizeof(struct node));
        newNode->data = value;
        int i;
        for(i=2; inext != NULL)
        {
                temp = temp->next;
        }
        newNode->next = temp->next;
        temp->next = newNode;
}
void insertAtFront(struct node** headRef, int value)
{
```

```
        struct node* head = *headRef;
        struct node *newNode;
        newNode = malloc(sizeof(struct node));
        newNode->data = value;
        newNode->next = head;
        head = newNode;
        *headRef = head;
}
void insertAtEnd(struct node* head, int value)
{
        struct node *newNode;
        newNode = malloc(sizeof(struct node));
        newNode->data = value;
        newNode->next = NULL;
        struct node *temp = head;
        while(temp->next != NULL)
        {
                temp = temp->next;
        }
        temp->next = newNode;
}
void deleteFromFront(struct node** headRef)
{
        struct node* head =  *headRef;
        head = head->next;
        *headRef = head;
}
void deleteFromEnd(struct node* head)
{
        struct node* temp = head;
        while(temp->next->next!=NULL)
```

```
        {
                temp = temp->next;
        }
        temp->next = NULL;
}
void deleteFromMiddle(struct node* head, int position)
{
        struct node* temp = head;
         int i;
        for(i=2; inext != NULL)
        {
                temp = temp->next;
        }
        temp->next = temp->next->next;
}
void main()
{
        struct node *head;
        struct node *one = NULL;
        struct node *two = NULL;
        struct node *three = NULL;
         /* Allocate memory */
        one = malloc(sizeof(struct node));
        two = malloc(sizeof(struct node));
        three = malloc(sizeof(struct node));
        /* Assign data values */
        one->data = 1;
        two->data = 2;
        three->data = 3;
        /* Connect nodes */
        one->next = two;
```

```
    two->next = three;
    three->next = NULL;
    /* Save address of first node in head */
    head = one;
    display(head); // 1 --->2 --->3 --->

    insertAtFront(&head, 4);

    display(head); // 4 --->1 --->2 --->3 --->
    deleteFromFront(&head);
    display(head); // 1 --->2 --->3 --->
    insertAtEnd(head, 5);

    display(head); // 1 --->2 --->3 --->5 --->
    deleteFromEnd(head);
    display(head); // 1 --->2 --->3 --->
    int position = 3;
    insertAtMiddle(head, position, 10);
    display(head); // 1 --->2 --->10 --->3 --->
    deleteFromMiddle(head, position);
}   display(head); // 1 --->2 --->3 --->
```

## TYPEDEF

The typedef keyword allows us to create a new data type name from an existing data type. It does not create a new data type but introduces a new name for existing type. A typedef declaration does not reserve storage. We cannot use the typedef specifier inside a function definition.

**Syntax**

```
typedef  datatype newdatatype;
```

**Example:**

```
typedef int INTEGER;
INTEGER num=29;
```

Here, INTEGER is the new name of data type int.

**Program**

```
#include<stdio.h>
#include<conio.h>
typedef   char    uchar;
void main()
{
   uchar ch = 'a';
   printf("ch: %c\n", ch);
   getch();
}
```

**Output**

```
ch : a
```

**typedef in structure**

We can use typedef with a structure. If the typedef keyword is given before the keyword struct, then the struct becomes a new type. For example,

```
typedef struct student
{
        int regno;
        char name[20];
```

};

Now, student is the new data type. The variables of the structure student can be declared as follows:

student stud;

**Program**

```
#include <stdio.h>
#include <string.h>
typedef struct student
{
        int id;
        char name[20];
        float percentage;
} status;
void main()
{
        status record;
        record.id=1;
         strcpy(record.name, "Raju");
        record.percentage = 86.5;
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
        getch();
}
```

**Output**

```
Id is: 1

Name is: Raju
Percentage is: 86.500000
```