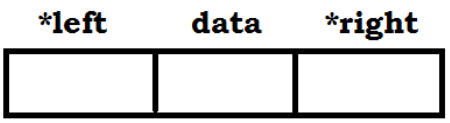
Double linked List

- It is linear data structure.
- It is used store elements in node form.
- We can represent the node using user defined data type(Structure).
- In double linked list, every node has at most 3 fields.
    - Data field
    - Link to left node
    - Link to right node
- Double linked is bi-directional, hence we can process elements in both directions.
- Compare to Single linked list, it occupies memory (one extra link).

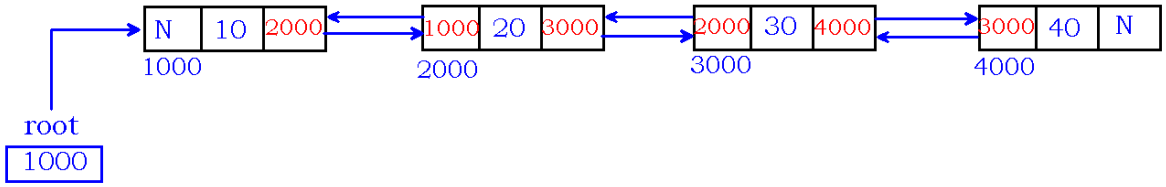
Node structure:

```
struct Node
{
    int data ;
    struct Node *left;
    struct Node *right;
};
```



Linked List representation:

- Every node has 2 pointers (left , right).
- First node left link is NULL
- Last node right link is NULL.
- Root node always points to First node in the list.



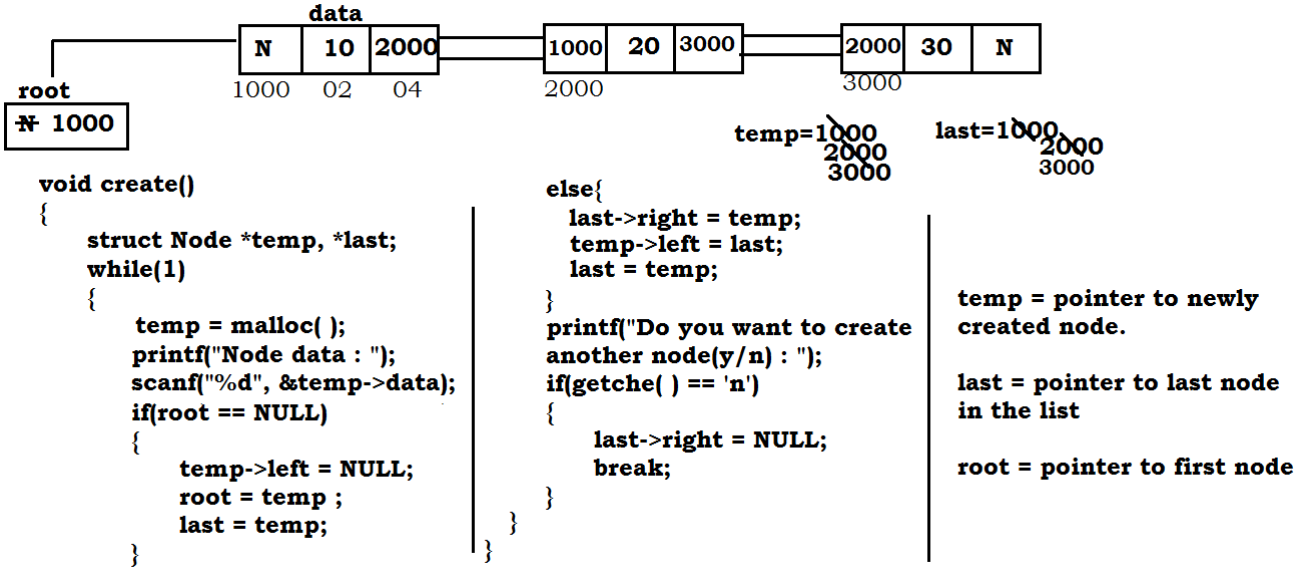
Operations on double linked list :

- Create()
- Length()
- Display()
- Append()
- addFirst()
- addAfter()
- remove()
- swap()
- sort()
- reverse()

Create():

- The function creates a list of elements using Iterators (loops).

- In the creation of list we use local variables (pointer type).
    - Temp points to newly created node
- Last points to last node in the list.

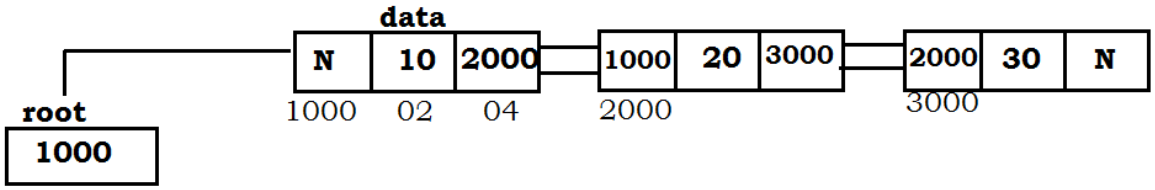



```
void create()
{
    struct Node *temp, *last;
    while(1)
    {
        temp = malloc( );
        printf("Node data : ");
        scanf("%d", &temp->data);
        if(root == NULL)
        {
            temp->left = NULL;
            root = temp ;
            last = temp;
        }
    else{
      last->right = temp;
      temp->left = last;
      last = temp;
    }
    printf("Do you want to create
    another node(y/n) : ");
    if(getche( ) == 'n')
    {
        last->right = NULL;
        break;
    }
  }
}
```

```
temp = pointer to newly
created node.

last = pointer to last node
in the list

root = pointer to first node
```

Length():

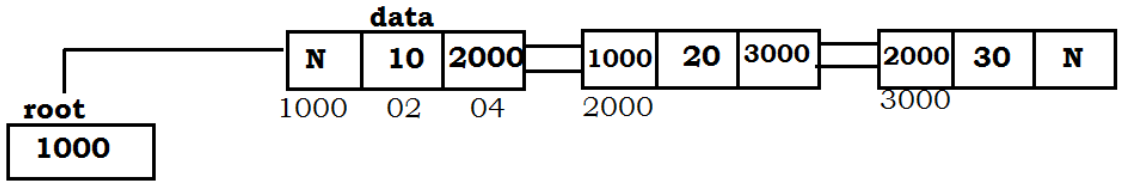
- Returns the number of nodes in the list.
- If list is empty, returns 0
- Return type is integer.




```
int length( )
{
    int count=0;
    struct Node *temp = root ;
    while(temp)
    {
        ++count ;
        temp = temp->right ;
    }
    return count ;
}
```

Display():

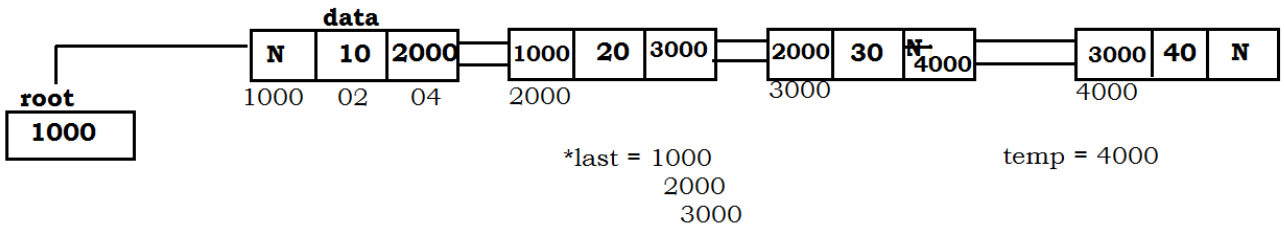
- Used to traverse all elements in the list.
- If the list has no nodes – results message “List is empty”




```
void display()
{
    if(root==NULL){
        printf("List is empty \n");
    }
    else{
        struct Node *temp = root;
        while(temp){
            printf("%d \n", temp->data);
            temp = temp->right;
        }
    }
}
```

Append():

- It is used append (add a node at end) a node to list.
- If the list has no elements, new node becomes root node.
- If list has elements, we use iterator to reach the last node in the list.
- We connect the new node to last node.



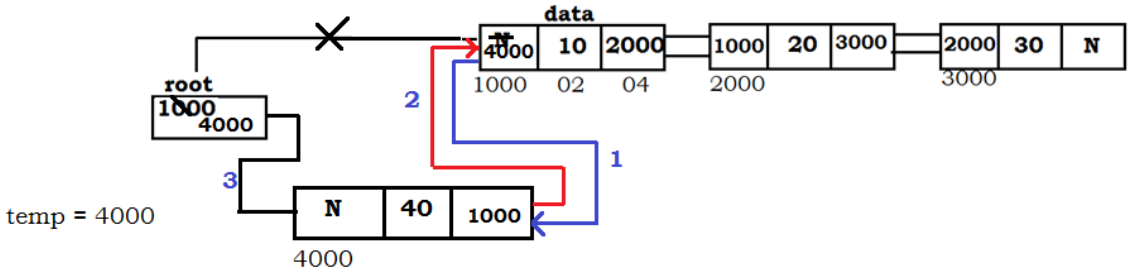

```
void append()
{
    struct Node *temp, *last;
    temp = malloc( );

    printf("Node data : ");
    scanf("%d" , &temp->data);
    temp->right = NULL;
```

```
if(root == NULL)
{
    temp->left = NULL;
    root = temp ;
}
else
{
    last = root ;
```

```
    while(last->right)
    {
        last = last->right;
    }
    last->right = temp ;
    temp->left = last ;
}
}
```

addFirst():

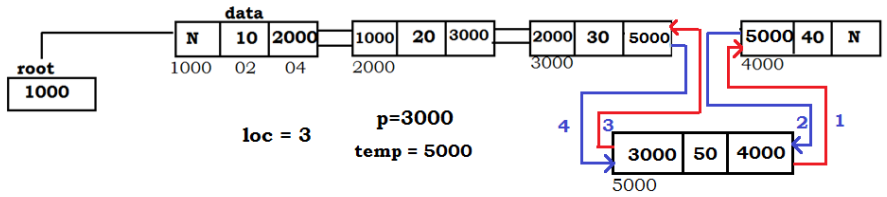
- This function is used to add the new node in the beginning of list.
- If list is empty, new node become the root node.




```
void addFirst()
{
    struct Node *temp;
    temp = malloc( );

    printf("Node data : ");
    scanf("%d", &temp->data);

    temp->left = NULL;

    if(root != NULL){
       root -> left = temp ;
    }
    temp -> right = root ;
    root = temp ;
}
```

addAfter():

- This function is used to add a node after specified node(location).
- If list has no such location results “Error”.



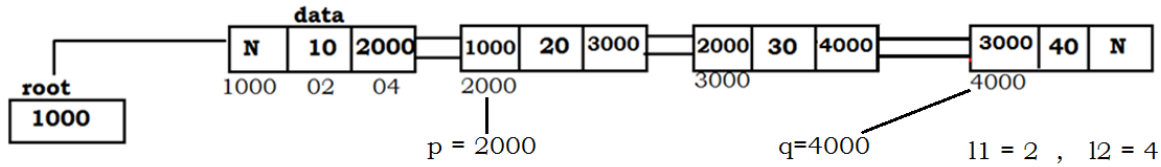

```
void addAfter()
{
    int loc, len, i=1;
    struct Node *temp, *p;

    len = length();
    printf("Enter loc : ");
    scanf("%d", &loc);

    if(loc > len){
        printf("Invalid
            location")
        return;
    }
```

```
temp = malloc( );
printf("Node data :");
scanf("%d", &temp->data);

p = root ;
while(i<loc)
{
    p=p->right;
    i++;
}
```

```
temp->right = p->right ;  ---- 1
if(p->right != NULL){
    p->right->left = temp ;   ----- 2
}
temp-> left = p ; ----- 3
p->right = temp ; ---- 4

}
```

Swapping:

- This function is used to swap the data of 2 specified nodes.
- If the list has no such location, results Error.
- Read 2 locations and swap the node data.



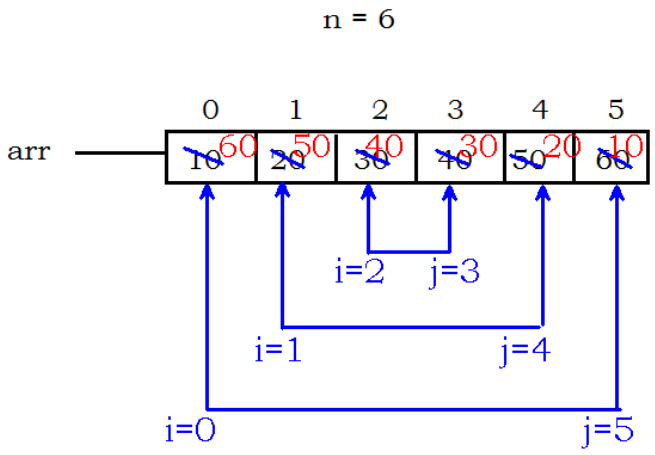

```
void swap( )
{
    int l1, l2, i=1, j=1 , temp ;
    struct Node *p, *q;
    int len = length();

    printf("Enter locations : ");
    scanf("%d%d", &l1, &l2);

    if(l1>len || l2>len){
        printf("Invalid locs")
        return ;
    }
```

```
    p = q = root;
    while(i<l1)
    {
        p=p->right;
        i++;
    }
    while(j<l2)
    {
        q=q->right;
        j++;
    }
```

```
    temp = p->data ;
    p->data = q->data ;
    q->data = temp;

}
```

Reverse Array:

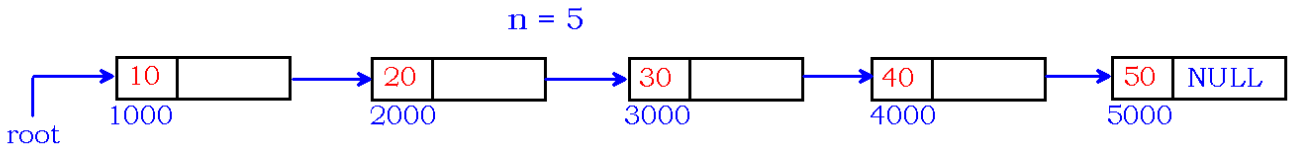
- Array elements can process using index.
- We access locations in both the directions(forward and backward)
- Hence we can easily reverse an array using loop.




```
i=0;
j=n-1;
while( i < j )
{
   temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
   i++;
   j--;
}
```

Reverse list: Single linked list:

- Single linked list can reverse like we reverse the array.
- Single linked list doesn't support indexing concept.
- Single linked list is not bi-directional.
- It is bit complex to reverse the single linked list elements.
- The program as follows:



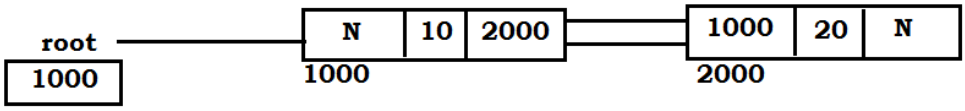

```
void reverse( )
{
   int i, j,k, n, temp;
   struct Node *p, *q;
   n = length( );
   i = 0 ;
   j = n-1 ;
   p=root;

   while(i<j)
   {
      q=root;
      k=0;
      while(k<j){
         q=q->link;
         k++;
      }
      temp = p->data;
      p->data = q->data;
      q->data = temp;
      i++;
      j-- ;
      p = p->link;
   }
}
```

Reverse list: Double linked list:

- Double linked list can reverse easily like array.
- Single linked list is bi-directional.
- The program as follows:

```
void reverse()
{
        int i, j, n, temp ;
        struct node *p, *q;

        n=length();
        i=0;
        j=n-1;

        p=root;
        q=root;

        while(q->right)
        {
                q=q->right;
        }
        while(i<j)
        {
                temp=p->data;
                p->data=q->data;
                q->data=temp;
                p=p->right;
                q=q->left;
                ++i;
                --j;
        }
}
```

Remove First node in double linked list:



temp = 1000

```
void removeFirst()
{
    struct Node *temp = root;
    if(root == NULL)
    {
        printf("List is empty \n");
        return;
    }
    printf("Deleted : %d \n", root->data);
    root = root->right ;
    if(root)
    {
        root->left = NULL ;
    }
    temp->right = NULL ;
    free(temp);

}
```

Remove specified node in double linked list: