LINKED LISTS

4.1 MEMORY MANAGEMENT

4.1.1 Basics about Memory

When a C program is compiled, the compiler translates the source code into machine code. Now the program is given a certain amount of memory to use. This memory is divided into four segments viz.

- **Code Segment**, in which the source code is stored
- **Data Segment**, that holds static and global variables
- **Heap Segment**, a free area
- **Stack Segment**, where local variables are stored

The memory organization for a C/C++ program is depicted in Figure 4.1.

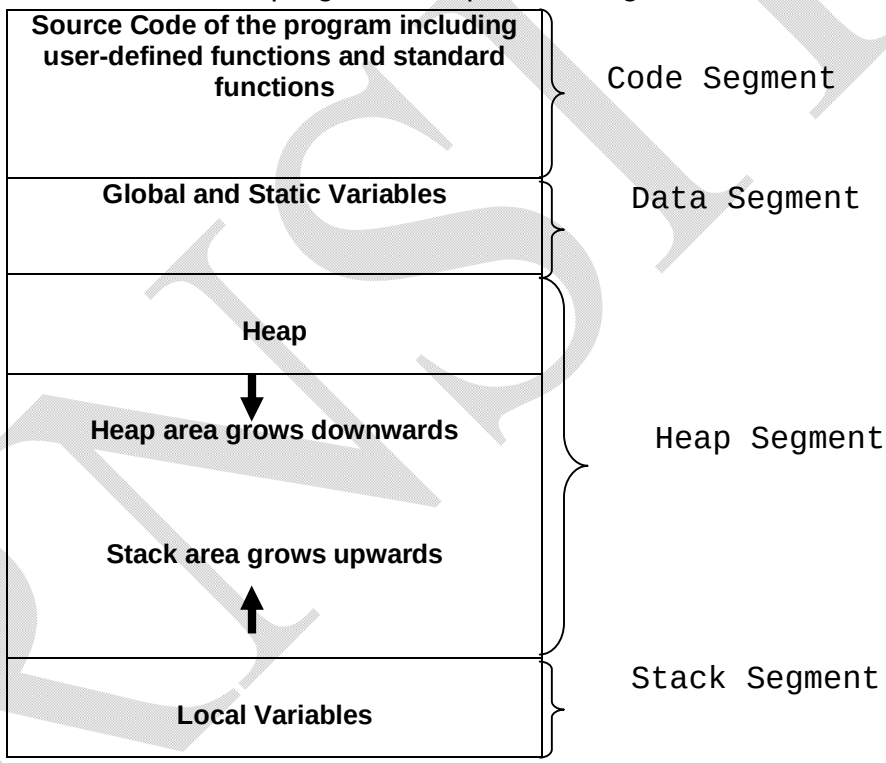


Figure 4.1 Memory Segments

Consider a block of code:

```
void test()
{
        int Var_X;
        --------
        --------
}
```

When this function is compiled and linked, an executable file will be generated. The executable file contains equivalent instructions for all the statements of the function. When this executable file is executed, memory is allocated for the variable Var_X. This is called as ***Static memory allocation*** and memory for Var_X is given from ***stack segment.***

When the compiler writes instructions to allocate memory for Var_X in executable file, it also writes the instructions to de-allocate that memory at the end of the block within which Var_X is declared. So, during execution, the memory for Var_X is de-allocated when Program Control (PC) comes out the block in which it is declared. Thus, any local variable can not be accessed outside its block. This is known as ***static memory de-allocation.*** Note that, this freed memory is returned to the Operating System (OS) by the compiler.

Static memory allocation has its own pitfalls. To understand the problems, consider the following sequence of code:

```
void test()
{
        int arr[20];
        ……
        ……
}
```

Now, 40 bytes (assuming, integer requires 2 bytes) will be allocated for the array arr, and it can store 20 integers. If the number of items to be stored exceeds 20, then the array arr will not be capable to do so. On the other hand, assume that the number of items is less than array size, say, 5. Then 10 bytes are sufficient and remaining 30 bytes of memory will be wasted. This kind of problems viz. memory overflow or wastage of memory are inevitable with static memory allocation.

To overcome these problems, one should go for ***Dynamic memory allocation***. In this method, the memory for variables will be allocated during runtime *based on the requirements arising at runtime*. And the memory blocks are taken from ***Heap segment***. When such memory locations are no longer required, the same can be de-allocated and returned to OS. This is known as ***dynamic memory de-allocation***. To do this kind of memory allocation and de-allocation, the programmer has to write specific code within the program.

The differences between static and dynamic memory allocations are listed in the following table –

Static Memory Allocation	Dynamic Memory Allocation
1. Used when the number of memory locations required is known in advance.	1. Used when number of memory locations required is unknown.
2. The size of the memory to be allocated is fixed and can not be varied during run time.	2. The size of memory can be increased or decreased during run time.
3. The variables with statically allocated memory are named ones and can be manipulated directly.	3. The variables with dynamically allocated memory are unnamed and can be manipulated indirectly only with the help of pointers.
4. The allocation and de-allocation of memory for variables is handled by the compiler automatically.	4. The memory allocation and de-allocation must be explicitly performed by the programmer.
5. As memory for the variables is decided at compile time and data manipulation is done on these locations, execution of the program is faster.	5. As memory must be allocated during run time, the execution will be slower.
6. Memory for global and static variables will be allocated in *Data Segment* and for local variables in *Stack Segment.*	*6.* Memory allocation will be in *Heap Segment.*

4.1.2 Generic Pointer/Void pointer

Dynamic memory allocation/deallocation in C is done through 4 functions viz. malloc(), calloc(), realloc() and free(). To understand the working of these functions, you should know the concept of generic pointer or void pointer.

As the name suggests, generic pointer is a pointer which can store the address of any type of variable. Normally, when we don't know, address of which type of variable we are going to assign to a pointer, we can declare it as a void pointer. Later, we can assign the address of a variable to the void pointer. While accessing the data via pointer, we need to use appropriate typecasting.

For example:

```
void main()
{
        void *p;        // a void pointer or generic pointer declared
        int  a= 5;
        float b= 3.4;

        p= &a;          //address of integer variable is assigned to p
        printf("%d", *(int *)p);        //dereference after typecasting
        p= &b;          //address of float variable is assigned to p
        printf("%f", *(float *)p);      //dereference after typecasting
}
```

One can note that, the same pointer *p* has been used to store the address of an integer as well as float. But, while dereferencing those values, first we should type-cast. Consider, the statement,

```
printf("%d", *(int *)p);
```

Here, *p* is type casted to the type ***pointer to integer type*** using **int *.** Later, it is dereference using ***.**

Thus, C/C++ language provides an option to programmer to make use of single generic pointer that is capable of storing address of any data type.

Note that, the memory address returned from heap area will always be of generic type. Hence, this topic is discussed here.

4.1.3 Dynamic Memory Allocation

The memory allocation and de-allocation can be done in C using the functions like malloc(), calloc(), realloc() and free. Each of these functions are discussed below. The function malloc() is discussed in more elaborative manner with examples. But, similar approach/concepts will apply to calloc() and realloc() functions also.

- **malloc(size):** This function allocates a block of 'size' bytes from the heap. It will return a void pointer if the requested memory is available in the heap area. Otherwise, it returns NUL. Syntax of this function is –

  ptr = (data_type *) malloc(size);

  Here,

  'ptr' is a pointer variable of type 'data_type',
  'data_type' is any basic C data type/user defined data type
  'size' is the number of bytes required.

  For ex.-

  ```
  int *ptr;
  ptr = (int *) malloc(sizeof(int) * n);
  ```

  Suppose n=3 and size of integer is 2 bytes, then above statement will allocate 3 x 2= 6 bytes and the address of first byte is assigned to ptr.

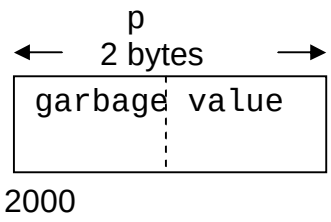
  To understand the concept clearly, let us discuss the memory map for the following set of statements:

  ```
  int *p;
  p = (int *) malloc(sizeof(int));
  *p= 10;
  ```

Now, the memory map for each of these statements may look like –

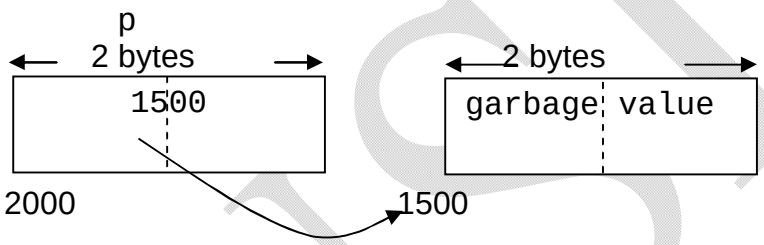
Statement:
int *p;

Memory Map:



Statement:
p = (int *) malloc(sizeof(int));

Memory Map:



Statement:
*p=10;

Memory Map:

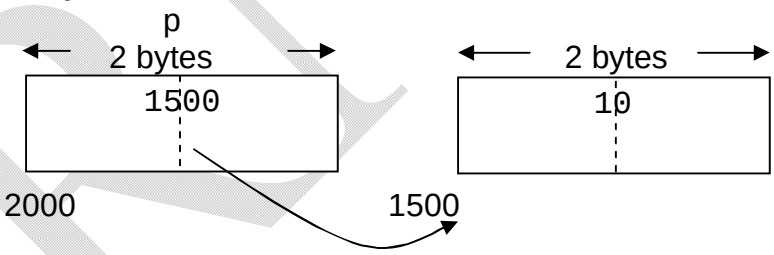


Figure 4.2 DMA memory map

It is important to note from the above figure that, the address of p viz. 2000 is allocated from the **stack segment** and the address stored at p viz. 1500 is allocated from the **heap segment.**

Note also that, if 2 bytes of memory is available in heap, then the statement,
p = (int *) malloc(sizeof(int));

assigns the base address of those memory blocks to the pointer p. Otherwise, NULL is returned. So, it is always a good practice to check whether memory is allocated or not, before using the pointer. An illustrative program for dynamic memory allocation is given below:

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

void main()
{
        int *p;
        p = (int *) malloc(sizeof(int));

        if (p == NULL)
        {
                printf("Allocation Failed");
                exit(0);
        }
        *p=10;
        printf("value is  %d", *p);
}
```

OUTPUT 1:

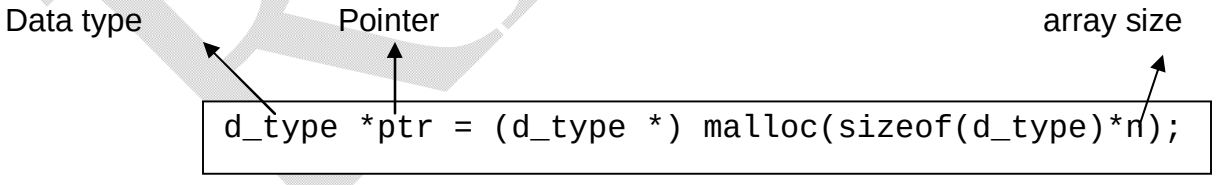
Value is 10 (if heap has sufficient free space)

OUTPUT 2:

Allocation Failed (in case of insufficient memory)

DMA for Arrays using malloc():

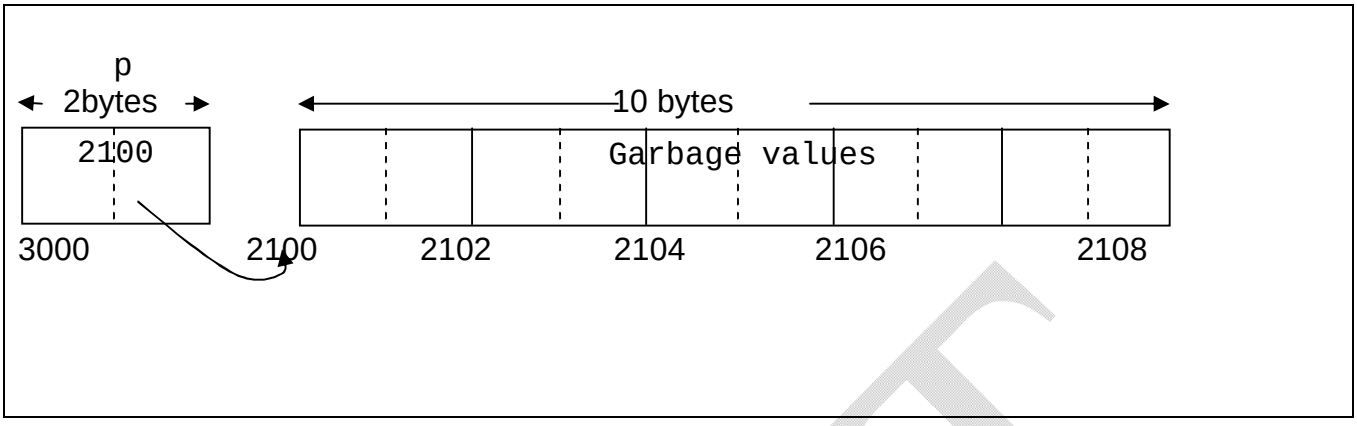
In the previous section, it has been discussed about allocating memory dynamically for a single variable. It is possible to allocate dynamic memory to arrays also. The syntax is as given below –



For example,

```
int *p = (int *) malloc(sizeof(int) *5);
```

Now, p will hold the base address of an array of 5 integers. The memory map may look like –



The following program illustrates DMA for arrays.

```
#include<stdio.h>
#include<alloc.h>
#include<stdlib.h>
void main()
{
  int *p, i,n;
  printf(”Enter size of the array:”);
  scanf(“%d”, &n);
  p= (int *) malloc(sizeof(int)*n);

  if (p == NULL)
  {
                printf(”Allocation Failed”);
                exit(0);
  }
  printf(”\nEnter array elements:”);
  for(i=0;i<n;i++)
        scanf(’%d”,  (p+i));

  printf(”\nElements are:”);
  for(i=0;i<n;i++)
        printf(“%d”, *(p+i));
}
```

The output would be –

```
Enter size of the array: 5
Enter array elements: 12    31    -4    64    10
Elements are: 12    31    -4    64    10
```

- **calloc(n, size):** This function is also used to allocate the memory from heap. But, malloc() allocates a single block of contiguous storage where as calloc() allocates multiple blocks of storage, each block of the same size. More over, calloc() initializes the memory block with the value zero, which is not the case with malloc(). Syntax is give as below –

  ptr= (data_type *) calloc(n,size);

  Here, 'ptr' is a pointer of type 'data_type', 'size' is the number of bytes required and 'n' is the number of blocks to be allocated of 'size' bytes.

  Rest of the explanation for calloc() function will be same as that for malloc().

- **realloc(ptr, size):** In some situations, we need to increase of decrease the memory blocks already allocated using malloc() or calloc(). In this case, to resize the memory, we will go for re-allocation of memory using this function. The syntax is as below –

  ptr = (data_type *) realloc(ptr, size);

  Here, 'ptr' is the starting address of allocated memory obtained before by using malloc(), calloc() or realloc(). And 'size' is the number of bytes required for re-allocation.

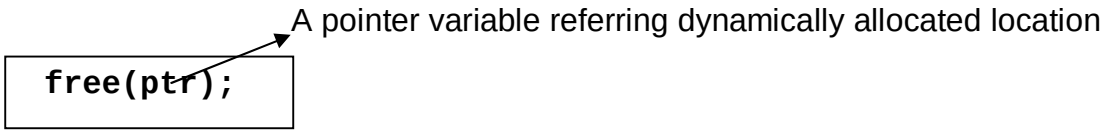
4.1.4 Dynamic Memory De-allocation

It has already been discussed that the dynamically allocated memory must be de-allocated and to be returned to OS. Let us see, what happens if such memory is not de-allocated, with the help of following example. Consider a block of code –

```
void test()
{
        int *ptr;
        ptr=(int *) malloc(sizeof(int));
        *ptr=10;
        ……
        ……
}
```

The memory map for ptr may be as shown in the **Figure 4.2.** As it has been discussed earlier, ptr is a local variable and the memory location 2000 is allocated from stack segment. The location with base address 1500 is allocated from heap. It is locked, and can be referred only with the help of ptr. When the program control goes out of the function test(), these two bytes of memory (2000 and 2001) will be de-allocated and returned to OS automatically. But, the location 1500, which is taken from heap, remains as an un-referenced location. Since ptr is destroyed, there is no means for referring the location 1500. As these locations (1500 and 1501) are locked, the OS can not allocate them to any other variable later, though they are no longer in use. This is known as ***memory leak.***

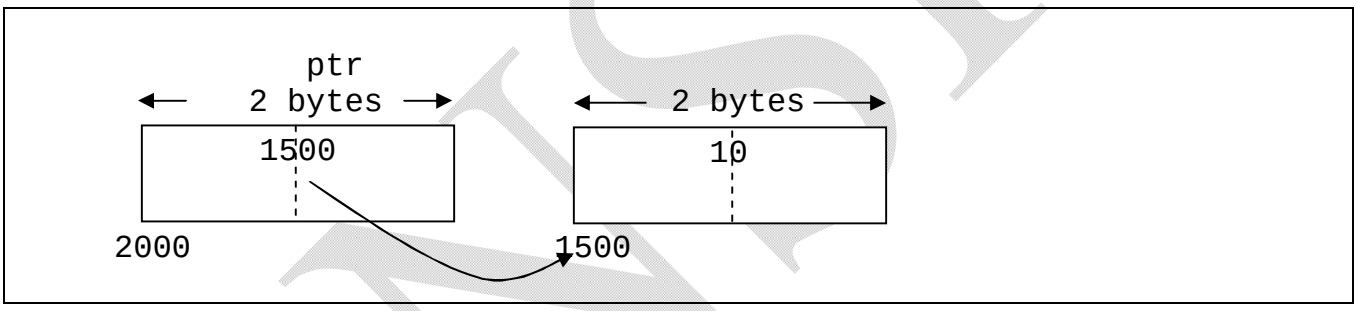
To overcome this problem, one has to de-allocate the memory manually in the program. This is achieved with the help of ***free()*** function. The syntax is –



For example, consider a code segment –

```
void test()
{
        int *ptr= (int *) malloc(sizeof(int)
        *ptr=10;
        printf("Value is %d ",  *ptr);
        free(ptr);
}
```

Here, the memory allocation is as shown below.



When the last statement

```
free(ptr);
```

is executed, the location with base address 1500 is released from the hands of ptr and is returned to OS. So, this location can then be allocated to some other variable by OS, if needed, thus preventing memory leak. Then ptr (location 2000) is de-allocated and returned to OS when the program control comes out of the function test().

4.2 LIMITATIONS OF ARRAY IMPLEMENTATION

Array is the most popular and frequently used data structure. Though we have developed the data structures like stack and queue using array, it has certain limitations as discussed below.

The conventional array uses static memory allocation. For example, the declaration

```
int a[100];
```

will allocate the memory for 100 integers, say 200 bytes. During runtime of the program, neither the size of the array can be reduced, nor can it be increased. In case, the program uses only 10 integers out of 100, the space allocated for rest of 90 integers (180 bytes) will be wasted. On the other hand, if the program requires more integers, say 120, during run time, it can not allocate and it will face shortage of memory.

Thus, the static memory allocation for arrays may create the problem of either shortage of memory or wastage of memory. This problem can be avoided by using dynamic arrays. For example, we can consider dynamic memory allocation for array as –

```
int *p, n;
p=(int *) malloc(sizeof(int)*n);
```

Now, memory for n integers, as requested by the user during execution time, will be allocated on heap. If the program requires more space, one can use *realloc()* function.

But, the problem still persists: array requires contiguous memory blocks and *malloc()* and *realloc()* may fail **if heap doesn't contain enough free space continuously** !!

To understand this problem, consider the following illustration:

```
int *p;
p=(int *) malloc(sizeof(int)*100);
```

The above allocation requires 200 bytes of memory (if int requires 2 bytes) in a contiguous blocks. Now, assume that heap has total of 300 bytes free space, but with chunks of 50, 80, 90, 70 and 10 bytes at different locations. But, our request of 200 bytes cannot be served, as heap doesn't contain 200 bytes at a stretch.

So, *malloc()* or *realloc()* will return NULL and hence, the program cannot be continued further. Thus, in a nutshell, we can say that usage of arrays either with static memory allocation or with dynamic memory allocation does not serve for all practical applications.

To overcome these problems with arrays, a new data structure called as **linked list** has been designed. Linked list uses dynamic memory allocation and each element can be stored any where in the heap.

In linked list, we have different categories based on their structure and path of accessibility of elements:

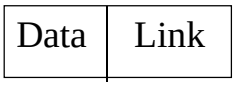
- Singly linked list
- Circular single linked list
- Doubly linked list
- Circular double linked list

Also, based on type of data that we store in the linked list, we categorize a list as homogeneous list and heterogeneous list. A list which contains single type of data (like int or char or float etc) is called as **homogeneous** list. Where as, a list with multiple types of data

like combination of int, char, float etc. is called as **heterogeneous** list. In a list, each element is called as a ***node.***

4.3 SINGLE (or SINGLY) LINKED LIST

In a singly linked list, every node consists of two parts viz. data field and link field as shown:



The data field consists of the item to be stored in the list. The link field of every node contains the address of next node in the list, and the link field of last node contains NULL to indicate end of the list. Thus, every time we need a new node, we request memory from heap.

The memory required for one node =
memory required for the item + memory required for a pointer

The diagrammatic representation of singly linked list may look like as shown in Figure 4.3.

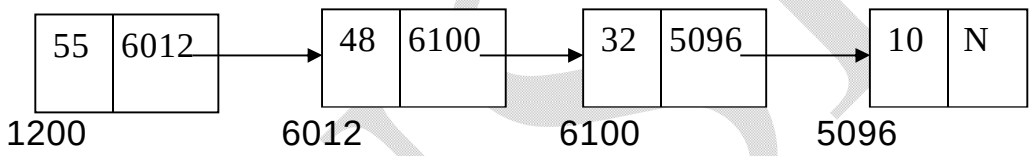


Figure 4.3 Representation of singly linked list

Various operations that can be performed on any linked list:

- Insert at front
- Insert at rear
- Delete from front
- Delete from rear
- Display the contents
- Insert at any position
- Delete from any position
- Search for a particular item
- Delete a particular item
- Creating ordered (ascending or descending) list

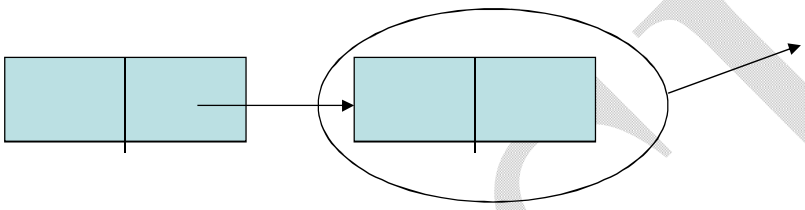
To perform various operations, first we should construct a linked list, or better to say a node. As discussed earlier, a node in singly linked list consists of a data part and link part.

For the initial stage of discussion, let us assume that we are going to create a linked list of integers. The one element (that is, node) in the list needs to contain one integer and one **pointer to another node.**

Since we need two entities which are related to each other, but are of different types (integer and a pointer), we use a structure to design a node. That is, we can use something similar to the following:

```
struct node
{
        int data;
        _____ *link;
};
```

Here, we have to think, **what is the type of the pointer *"link"*?**. The question is: the pointer *link* is going to store the address of what type of element?. The answer is: it is going to store the address of another node. Because, the linked list looks like this:



Thus, the structure looks like:

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

Now, refer Figure 4.3. It clearly indicates that, we are interested in the address of every node of linked list, which is the most important information needed to maintain any linked list.

So, for doing any operation, we need to create a pointer to ***struct node*.** For example,

```
struct node *n1;
struct node *n2;  etc.
```

To reduce the typing job, we can use *typedef* as –

```
typedef struct node *NODE;
```

Now, whenever we need to create a node in linked list, we can just declare –

```
NODE n1;
NODE n2;        etc.
```

We know that, each time we need a new node, we should request memory from the heap using *malloc()* function. To avoid repetitive task, we will write a function called ***getnode()*** to get heap memory and thus to create a new node.

```
NODE getnode()
{
        NODE x;
        x=(NODE) malloc(sizeof(struct node));

        if(x==NULL)
        {
                printf("no memory in heap");
                exit(0);
        }
        return x;
}
```

On successful allocation of memory, the above function returns the address of one block of memory, which is equivalent to the size of one node.

Similarly, to free a node, we can create a function like:

```
void freenode(NODE x)
{
        free(x);
}
```

But, since the ***freenode()*** function contains only one statement, it makes no sense to call this function, instead, it is better to use a built-in function ***free().***

By referring to the Figure 4.3, it can be observed that, if we have the address of first node in our hand, we can trace the entire list. Thus, for doing all operations on singly linked list, we keep starting node as a reference, and we declare that node as –

NODE start;

Now, we will start implementing the code for various operations on singly linked list. Initially, we will implement 5 basic operations viz. insert_front, insert_rear, delete_front, delete_rear and display.

Program for Implementation of Singly linked list

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

struct node
{
        int data;
        struct node *link;
};
typedef struct node *NODE;

NODE getnode()
{
        NODE x;
        x=(NODE) malloc(sizeof(struct node));
        if(x==NULL)
        {
                printf("no memory in heap");
                exit(0);
        }
        return x;
}

NODE insert_front(int item, NODE start)
{
        NODE temp;
        temp = getnode();
        temp->data=item;
        temp->link=start;
        return temp;
}

NODE delete_front(NODE start)
{
    NODE temp;

    if(start==NULL)
    {
        printf("no element to delete\n");
        return start;
    }
```

```
    temp=start;
    printf("Deleted item=%d", temp->data);
    start=start->link;
    free(temp);
    return start;
}

NODE insert_rear(int item, NODE start)
{
        NODE temp, cur;
        temp=getnode();
        temp->data=item;
        temp->link=NULL;

        if (start==NULL)
                return temp;

        cur=start;
        while(cur->link!=NULL)
                cur=cur->link;

        cur->link=temp;
        return start;
}

void display(NODE start)
{
        NODE temp;
        if(start==NULL)
        {
            printf("No element to display\n");
            return ;
        }
        printf("The contents of list:\n");

        temp=start;

        while(temp!=NULL)
        {
                printf("%d\n", temp->data);
                temp=temp->link;
        }
}
```

```
NODE delete_rear(NODE start)
{
        NODE prev, cur;

        if(start==NULL)
        {
                printf("no element to delete\n");
                return start;
        }

        if(start->link==NULL)
        {
                printf("\nDeleted element is%d", start->data);
                free(start);
                return NULL;
        }
        prev=NULL;
        cur=start;

        while(cur->link!=NULL)
        {
                prev=cur;
                cur=cur->link;
        }
        printf("\nDeleted element is %d", cur->data);      .
        free(cur);
        prev->link=NULL;
        return start;
}

void main()
{
        int opt, item;
        NODE start=NULL;

        for(;;)
        {
                printf("1.Insert Front\n 2.Insert Rear\n 3. Display\n");
                printf(" 4.Delete Front\n  5.Delete Rear\n");
                printf("enter your option:");
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1: printf("\nenter item");
```

```
                        scanf("%d",&item);
                         start=insert_front(item,start);
                        break;
                case 2: printf("\nenter item");
                        scanf("%d",&item);
                        start=insert_rear(item,start);
                        break;
                case 3: display(start);
                         break;
                case 4: start=delete_front(start);
                        break;
                case 5: start=delete_rear(start);
                        break;
                default: exit(0);
            }
        }
}
```

4.3.1 Creating Ordered Linked List

A linked list in which all the items are stored in some specified order viz. ascending or descending is known as an ordered linked list. Note that after the insertion of one item into an ordered list, the order should be maintained. Thus, the insertion process must be done as explained below:

- If the item to be inserted is less than the first item of the existing list, then this 'new node' should become the 'start' of resulting list.
- If the item to be inserted is grater than the last item of the existing list, then 'new node' must be the last node of the resulting list.
- But if the item is somewhere between the linked list then it must be inserted at appropriate position using the 'link' field of the nodes.

NOTE: Instead of insert_front() and insert_rear() functions in the linked list program, if you use the following function i.e. insert_order(), then you will get an ordered linked list.

Code Snippet for creating ordered linked list:

```
NODE insert-order(int item, NODE start)
{
        NODE temp, prev, cur;
        temp=getnode();
        temp->data = item;
        temp->link = NULL;

        if(start == NULL)
                return temp;
```

```
        if(item < start ->data)
        {
                temp->link =start;
                return temp;
        }
        prev = NULL;
        cur = start;
        while(cur != NULL && item >= cur->data)
        {
                prev = cur;
                cur = cur->Link;
        }

        prev->link = temp;
        temp->link=cur;
        return start;
}
```

4.3.2 Stack and Queue using Linked lists

We have discussed earlier that stack is LIFO data structure and queue is a FIFO data structure. And, they can be implemented using either arrays or linked lists. Array implementation of these data structures have been discussed in the previous chapters. The linked-list representation can be done as give below:

- **Stack:** Implement *insert_front(), delete_front()* and *display()* functions using singly linked list. That is, insertion and deletion from the same end is nothing but stack.
- **Queue:** Implement *insert_rear(), delete_front()* and *display()* functions.
- **Priority Queue:** Implement *insert_order(), delete_front()* and *display()* functions.
- **Deque:**
  - **General Deque:** Implement *insert_front(),insert_rear(), delete_front(), delete_rear()* and *display()* functions.
  - **Input Restricted Deque:** Implement *insert_rear(), delete_front(), delete_rear()* and *display()* functions.
  - **Output Restricted Deque:** Implement *insert_rear(),insert_front(), delete_front()* and *display()* functions.

4.3.3 Searching for a particular node

This operation is used to find whether a given key item is present in the list or not. If the item is there in the list, it is known as successful search, otherwise an unsuccessful search. A function to search for an item in the existing list is given below. Use appropriate *main()* function and *insert()* functions and then use 'search' function for a working of full program.

Code snippet for searching a particular node:

```
void search( int key, NODE start)
{
        NODE cur;
        int pos;

        if (start = = NULL)
        {
                printf("List is empty");
                return;
        }
        cur = start;
        pos = 1;

        while(cur!=NULL && key!=cur->data)
        {
                cur = cur->link;
                pos++;
        }
        if(cur==NULL)
        {
                printf("key not found");
                return;
        }
        printf("Key is found at position %d", pos);
}
```

4.3.4 Deletion of a node whose data field is given

Sometimes, we may need to delete a particular node from the list, based on its value. That is, when we are given a 'data' field of some node, then the entire list must be traversed to find that node, and then it must be deleted.

Following function is for deleting a particular node. Use appropriate *main()* function and *insert()* function for the complete implementation.

```
NODE delete_data(int item, NODE start)
{
        NODE prev, cur;

        if(start = = NULL)
        {
                printf("No item to delete");
                return start;
        }
```

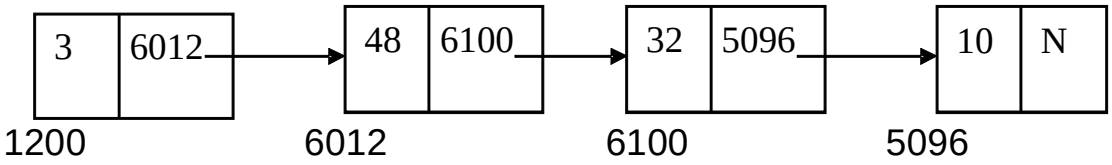
```
        if(item= = start->data)
        {
                cur = start;
                start=start->link;
                free (cur);
                return start;
        }

        prev = NULL;
        cur = start;

        while(cur!=NULL && item !=cur->data)
        {
                prev = cur;
                cur= cur->link;
        }

        if(cur = = NULL)
        {
                printf("Item not found);
                return start;
        }

        prev->link = cur->link;
        free(cur);

        return start;
}
```

4.3.5 Header Nodes

To simplify the design of linked list, sometimes we will have a special node at the beginning of the list. Usually, the 'data' field of this node would be empty, and it will not represent any item of the linked list. Such a node is called as ***header node*.**

Sometimes the integer value representing the number of nodes present in the list will be stored in header node. But, in this case each time an insertion or deletion occurs, the 'data' field of header node must be updated to keep track of the actual information. If the list is empty, then link field of header node contains NULL or else it will contains the address of first node of the linked list.

For example, following diagram shows a singly linked list with a header node, where the data field of header node contains total number of nodes in the list.



Here, the first node with the address 1200 is a header node. Its data field contains the number 3 indicating there are 3 nodes in the list.

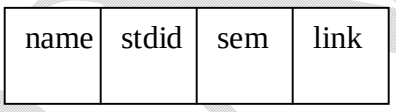
4.3.6 Non-homogenous List

All the examples that we discussed till now were having the 'data' field of a node as an integer variable. But, in fact, a 'data' field of a node in a singly linked list may also contain the variables of some other data type. Moreover, a node can contain more than one value in its data field. That is, it is capable of storing various information.

For example, we can define our structure as-

```
struct student
{
        char name[20];
        int stdid;
        int sem;
        struct student *link;
};
```

Then, each of the member field can be stored in 'data' field of a linked list. That is, one node of the linked list may look like –



Suppose we have declared a variable like-

```
struct student *temp;
```

Then, the member variables can be accessed as

```
temp -> name;
temp -> stdid;
temp -> sem;        etc.
```

The operations for non-homogenous or non-integer linked list are same as those of integer linked list. But the functions like *insert-front()* and *insert-rear()* should have the parameters like char, int etc. instead of only one parameter.

Consider following example program to illustrate non-homogenous list.

Program for Illustration of non-homogenous (or heterogenous) list

Question: Write a C program to construct a singly linked list consisting of the following information in each node: student_id (integer), student_name (character string) and semester (integer). Perform insert front and display operations on this non-homogenous linked list

Answer:

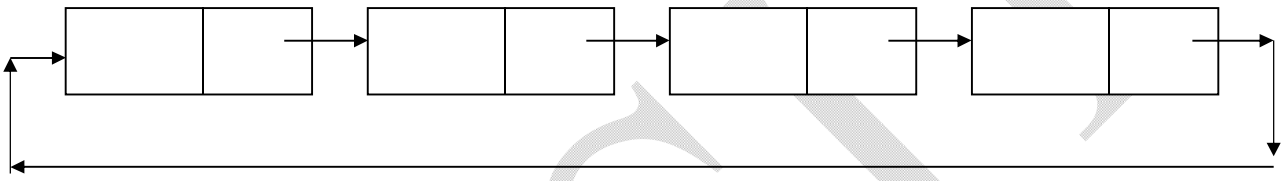
```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>

struct node
{
        int sid;
        char sname[15];
        int sem;
        struct node *link;
};
typedef struct node *NODE;

NODE getnode()
{
        NODE x;
        x = (NODE) malloc (sizeof (struct node));
        if (x == NULL)
        {
                printf("No memory space\n");
                exit(0);
        }
        return x;
}

NODE insert_front (int stuid, char stuname[], int semester, NODE start)
{
        NODE temp;
        temp = getnode();
        temp->sid = stuid;
        strcpy(temp->sname, stuname);
        temp->sem = semester;
        temp->link = start;

        return temp;
}
```

```
void display (NODE start)          //Display the contents of the list.
{
        NODE temp;

        if (start == NULL)
        {
                printf("List is empty\n");
                return;
        }

        printf("\nThe contents of the list are :\n");
        temp = start;

        printf("S-ID \t\t          S-NAME \t\t    SEMESTER\n");

        while (temp != NULL)
        {
                printf("%d \t %s \t \t %d\n", temp->sid, temp->sname, temp->sem);
                temp = temp->link;
        }
}

void main()
{
        NODE start = NULL;
        int opt, stuid,semester;
        char stuname[15];

        for(;;)
        {
                printf("1: Insert \n 2.Display \n 3: EXIT\n");
                printf("Enter your option\n");
                scanf("%d",&opt);

                switch (opt)
                {
                        case 1: printf("Enter the student id \n");
                                scanf("%d",&stuid);
                                printf("Enter the student name \n");
                                scanf("%s", stuname);
                                printf("Enter the semester\n");
                                scanf("%d",&semester);
                                start = insert_front(stuid, stuname, semester, start);
                                break;
                        case 2: display(start);   break;
```

```
                    case 3:
                    default: exit(0);
                }
            }
        }
```

4.4 CIRCULAR LINKED LIST

Circular singly linked list is quite similar to singly linked list. The only difference is – the 'link' field of a last node in a circular singly linked list contains the address of first node, instead of NULL. The diagrammatic representations is -



In a singly linked list, we can trace the list in one direction. That is, if we are at 10^{th} node, we can't trace back to access 9^{th} node, instead, we have trace from the beginning once again. This is time consuming. Hence, we make use of circular list to avoid this problem up to some extent.

Because, in a circular list, from any node we can trace rest of the nodes processing in a forward direction (refer diagram given in previous slide). Theoretically, any node in a circular list can be treated as a first node, and its previous node as a last node. But, since, there won't be NULL in any node to indicate end of list, the circular lists have to be processed properly, otherwise, it may lead to infinite loop.

In most of the real time applications, we will be interested in first node and the last node of a linked list. So, if we keep the first node in our hand, we have to trace the whole list to get a last node, as we did in singly linked list. So, in a circular list, **we will keep last node as a tool in our hand**, whose next node itself will be the first node, and hence, we can get both the nodes without much trace.

All the operations performed on an ordinary singly linked list can be implemented on circular singly linked list.

Program for Operations on Circular linked lists

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

struct node
{
        int data;
        struct node *link;
};
typedef struct node *NODE;

NODE getnode()
{
        NODE x;
        x=(NODE) malloc(sizeof(struct node));
        if(x==NULL)
        {
                printf("no memory in heap");
                exit(0);
        }
        return x;
}

NODE insert_front(int item, NODE last)
{
   NODE temp;
   temp = getnode();

   temp->data=item;
   temp->link=temp;

    if(last==NULL)
        return temp;

  temp ->link = last->link;
  last ->link=temp;
  return last;
}

NODE insert_rear(int item, NODE last)
{
        NODE temp;
        temp=getnode();
```

```
        temp->data=item;
        temp->link=temp;

        if (last==NULL)
                return temp;

        temp->link=last->link;
        last->link=temp;
        return temp;
}

void display(NODE last)
{
    NODE temp;

    if(last==NULL)
    {
        printf("No element to display\n");
        return ;
    }

    temp=last->link;
    printf("The contents of list:\n");

    while(temp!=last)
    {
        printf("%d\n" temp->data);
        temp=temp->link;
    }

    printf("%d", temp->data);
}

NODE delete_front(NODE last)
{
        NODE temp;
        if(last==NULL)
        {
                printf("no element to delete\n");
                return NULL;
        }
        if(last->link==last)
        {
                printf("The item deleted is %d", last->data);
```

```
                free(last);
                return NULL;
        }
        temp=last->link;
        last->link=temp->link;
        printf("Item deleted is %d", temp->data);
        free(temp);
        return last;
}

NODE delete_rear(NODE last)
{
        NODE prev;
        if(last==NULL)
        {
                printf("no element to delete\n");
                return NULL;
        }
        if(last->link==last)
        {
                printf("The item deleted is %d", last->data);
                free(last);
                return NULL;
        }
        prev=last->link;
        while(prev->link!=last)
                prev=prev->link;

        prev->link=last->link;
        printf("\nDeleted element is %d", last->data);
        free(last);
        return prev;
}

void main()
{
        int opt, item;
        NODE last=NULL;

        for(;;)
        {
                printf("1.Insert Front \n 2.Insert Rear\n 3.Display\n")
                printf(" 4.Delete Front \n  5.Delete Rear\n");
                printf("Enter your option:");
                scanf("%d",&opt);
```

```
                switch(opt)
                {
                        case 1: printf("\nenter item");
                                scanf("%d",&item);
                                last=insert_front(item,last);
                                break;
                        case 2: printf("\nenter item");
                                scanf("%d",&item);
                                 last=insert_rear(item,last);
                                break;
                        case 3: display(last);
                                break;
                        case 4: last=delete_front(last);
                                break;
                        case 5: last=delete_rear(last);
                                 break;
                        default: exit(0);
                }
        }
}
```

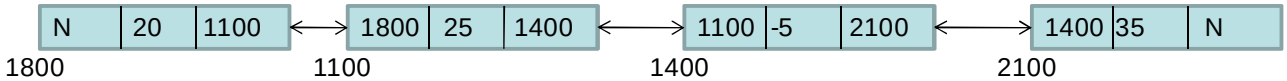
4.5 DOUBLE LINKED LIST

In a singly linked list and a circular singly linked list, we can traverse a list in a forward direction. That is, when we are at n^{th} node and would like to access $(n-1)^{th}$ node, then we have to traverse in a forward direction only. This is time consuming.

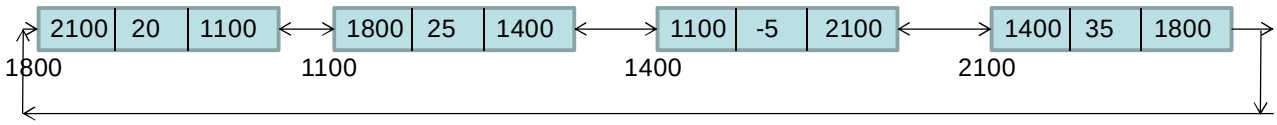
To avoid such problem, we will go for double linked list in which each node consists of two link fields (viz. left link and right link) and one data field as shown:

Here, right link is used to store the address of next node and left link is used to store the address of previous node. In ordinary doubly linked list, the left link field of first node and the right link field of last node contain NULL.

A typical doubly linked list may look like this –



A circular doubly linked list may look like this –



The structure declaration for a node will be –

```
struct node
{
        struct node *llink;
        int data;
        struct node *rlink;
};
```

NOTE: All the primitive operations done using singly linked list can be applied for doubly linked list and circular doubly linked list. Implementation of doubly linked list is illustrated in Lab Program 13.