DATA STRUCTURES

UNIT -1

The term data structure is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm you might use, and choosing an algorithm affects the data structures we use.

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

Introduction to Data Structures:

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored. To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

A data structure is said to be *linear* if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be *non linear* if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category.

Data structures: Organization of data

The collection of data you work with in a program have some kind of structure or organization.No matte how complex your data structures are they can be broken down into two fundamentaltypes:

- Contiguous
- Non-Contiguous.

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements. In contrast, items in a non-contiguous structure and scattered in memory, but we linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node. Figure 1.2 below illustrates the difference between contiguous and non-contiguous structures.

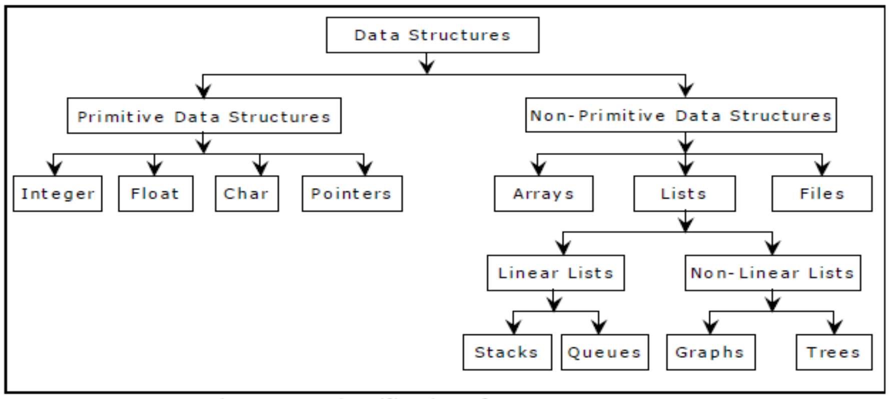


Figure 1.1. Classification of Data Structures

Contiguous structures:

Contiguous structures can be broken drawn further into two kinds: those that contain data items of all the same size, and those where the size may differ. Figure 1.2 shows example of each kind. The first kind is called the array. Figure 1.3(a) shows an example of an array of numbers. In an array, each element is of the same type, and thus has the same size.

The second kind of contiguous structure is called structure, figure 1.3(b) shows a simple structure consisting of a person's name and age. In a struct, elements may be of different data types and thus may have different sizes.

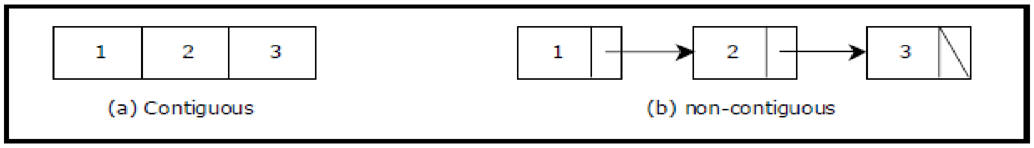


Figure 1.2 Contiguous and Non-contiguous structures compared

For example, a person's age can be represented with a simple integer that occupies two bytes of memory. But his or her name, represented as a string of characters, may require many bytes and may even be of varying length.

Couples with the atomic types (that is, the single data-item built-in types such as integer, float and pointers), arrays and structs provide all the “mortar” you need to built more exotic form of data structure, including the non-contiguous forms.

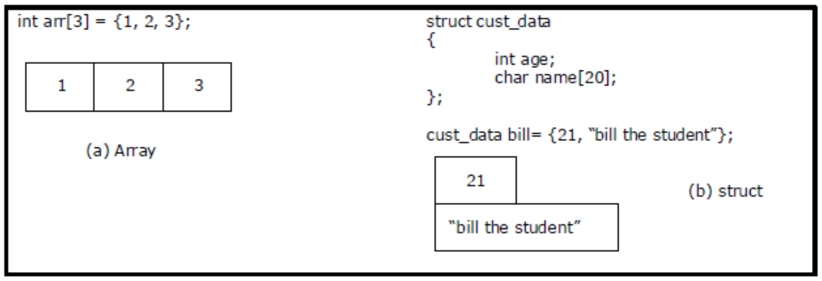


Figure 1.3 Examples of contiguous structures.

Non-contiguous structures:

Non-contiguous structures are implemented as a collection of data-items, called nodes, where each node can point to one or more other nodes in the collection. The simplest kind of non-contiguous structure is linked list.

A linked list represents a linear, one-dimension type of non-contiguous structure, where there is only the notation of backwards and forwards. A tree such as shown in figure 1.4(b) is an example of a two-dimensional non-contiguous structure. Here, there is the notion of up and down and left and right. In a tree each node has only one link that leads into the node and links can only go do the tree. The most general type of non-contiguous structure, called a graph has no such restrictions. Figure 1.4(c) is an example of a graph.

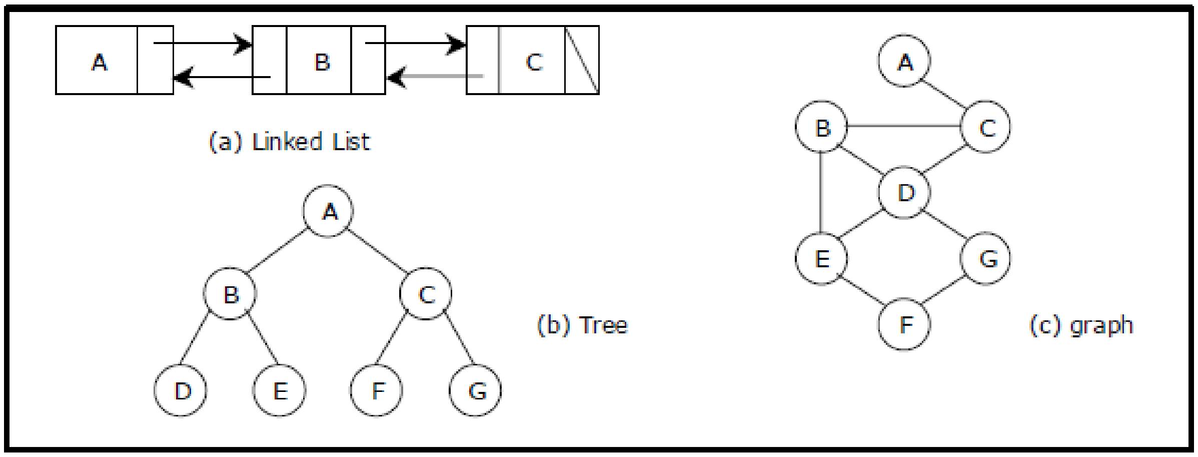


Figure 1.4. Examples of non-contiguous structures

Hybrid structures:

If two basic types of structures are mixed then it is a hybrid form. Then one part contiguous and another part non-contiguous. For example, figure 1.5 shows how to implement a double–linked list using three parallel arrays, possibly stored a past from each other in memory.

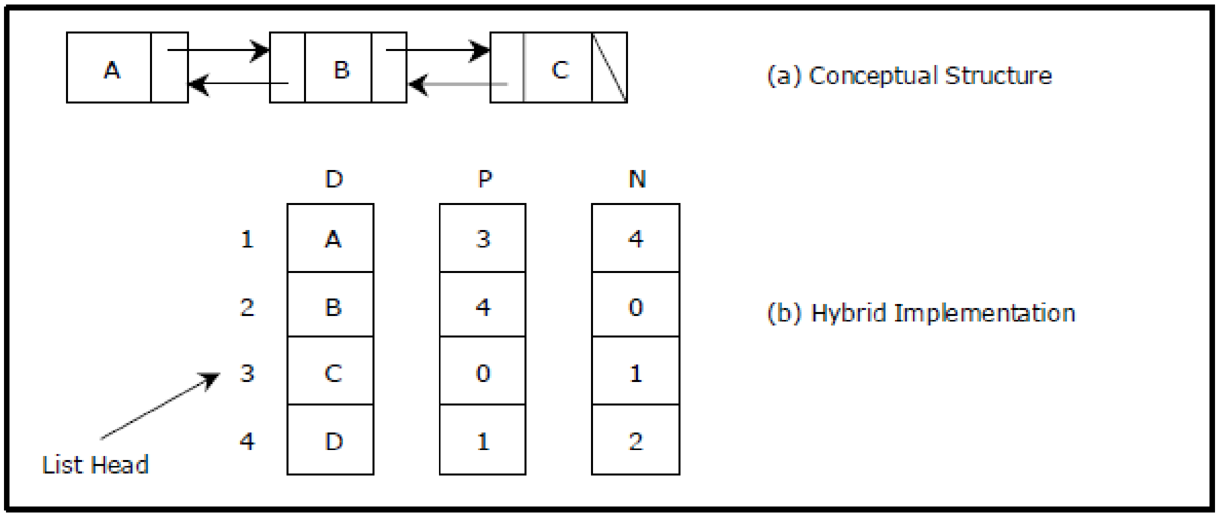


Figure 1.5. A double linked list via a hybrid data structure

The array D contains the data for the list, whereas the array P and N hold the previous andnext “pointers’’. The pointers are actually nothing more than indexes into the D array. For instance, D[i] holds the data for node i and p[i] holds the index to the node previous to i, where may or may not reside at position i–1. Like wise, N[i] holds the index to the next node in the list.

Abstract Data Type (ADT):

The design of a data structure involves more than just its organization. You also need to plan for the way the data will be accessed and processed – that is, how the data will be interpreted actually, non-contiguous structures – including lists, tree and graphs – can be implemented either contiguously or non- contiguously like wise, the structures that are normally treated as contiguously - arrays and structures – can also be implemented non-contiguously.

The notion of a data structure in the abstract needs to be treated differently from what ever is used to implement the structure. The abstract notion of a data structure is defined in terms of the operations we plan to perform on the data.Considering both the organization of data and the expected operations on the data, leads to the notion of an abstract data type. An *abstract data type* in a theoretical construct that consists of data as well as the operations to be performed on the data while hiding implementation.

For example, a stack is a typical abstract data type. Items stored in a stack can only be addedand removed in certain order – the last item added is the first item removed. We call these operations, pushing and popping. In this definition, we haven’t specified have items are store on the stack, or how the items are pushed and popped. We have only specified the valid operations that can be performed. For example, if we want to read a file, we wrote the code to read the physical file device. That is, we may have to write the same code over and over again. So we created what is knowntoday as an ADT. We wrote the code to read a file and placed it in a library for a programmer touse.

As another example, the code to read from a keyboard is an ADT. It has a data structure,character and set of operations that can be used to read that data structure.

To be made useful, an abstract data type (such as stack) has to be implemented and this is where data structure comes into ply. For instance, we might choose the simple data structure of an array to represent the stack, and then define the appropriate indexing operations to perform pushing and popping.

Selecting a data structure to match the operation:

The most important process in designing a problem involves choosing which data structure to use. The choice depends greatly on the type of operations you wish to perform. Suppose we have an application that uses a sequence of objects, where one of the main operations is delete an object from the middle of the sequence. The code for this is as follows:

```
void delete (int *seg, int &n, int posn)

// delete the item at position from an array of n elements.
{
  if (n)
  {
         int i=posn;
         n--;
        while (i < n)
         {
          seq[i] = seg[i+1];
          i++;
        }
  }
   return;
}
```

This function shifts towards the front all elements that follow the element at position *posn*. This shifting involves data movement that, for integer elements, which is too costly. However, suppose the array stores larger objects, and lots of them. In this case, the overhead for moving data becomes high. The problem is that, in a contiguous structure, such as an array the logical ordering (the ordering that we wish to interpret our elements to have) is the same as the physical ordering (the ordering that the elements actually have in memory).

If we choose non-contiguous representation, however we can separate the logical ordering from the physical ordering and thus change one without affecting the other. For example, if we store our collection of elements using a double–linked list (with previous and next pointers), we can do the deletion without moving the elements, instead, we just modify the pointers in each node. The code using double linked list is as follows:

```
void delete (node * beg, int posn)
//delete the item at posn from a list of elements.
{
        int i = posn;
        node *q = beg;
        while (i && q)
          {
            i--;
           q = q -> next;
        }
        if (q)
        { /* not at end of list, so detach P by making previous and
                next nodes point to each other */
                node *p = q -> prev;
                node *n = q -> next;
                if (p)
                p -> next = n;
                if (n)
                n -> prev = P;
        }
return;
}
```

The process of detecting a node from a list is independent of the type of data stored in the node, and can be accomplished with some pointer manipulation as illustrated in figure below:

Since very little data is moved during this process, the deletion using linked lists will often befaster than when arrays are used. It may seem that linked lists are superior to arrays. But is that always true? There are tradeoffs. Our linked lists yield faster deletions, but they take up more space because they require two extra pointers per element.

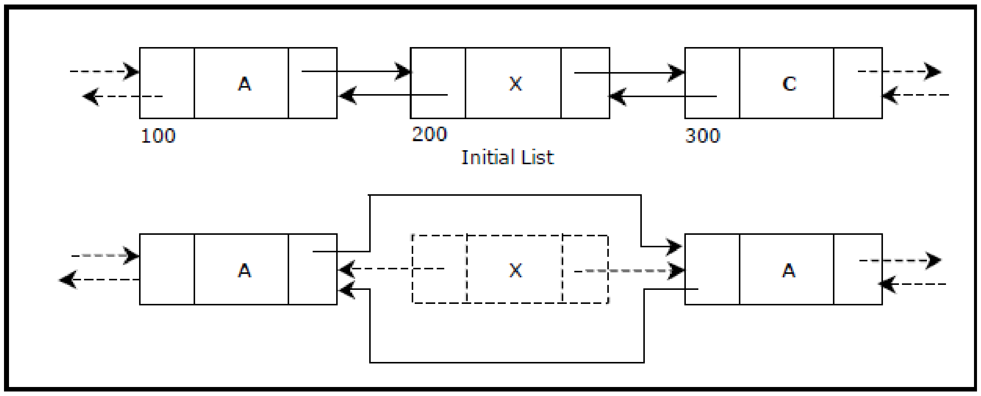


Figure 1.6 Detaching a node from a list

Linked List

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast. The disadvantages of arrays are:

• The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.

• Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

• Deleting an element from an array is not possible.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Here is a quick review of the terminology and rules of pointers. The linked list code will depend on the following functions:

malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc() and other heap functions are in stdlib.h. malloc() returns NULL if it cannot fulfill the request. It is defined by:

*void *malloc (number_of_bytes)*

Since a

void * is returned the C standard states that this pointer can be converted to any type.

For example

```
char *cp;
cp = (char *) malloc (100);
```

Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes. For example,

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

free()is the opposite of malloc(), which de-allocates memory. The argument to free() is a pointer to a block of memory in the heap — a pointer which was obtained by a malloc() function. The syntax is:

free (ptr);

The advantage of free() is simply memory management when we no longer need a block.

LINKED LIST CONCEPTS

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:
1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node. A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

Comparison between array and linked list:

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

FEATURE	ARRAYS	LINKED LISTS
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials areexpression containing terms with non zero coefficient and exponents.
Forexample:P(x) = a0 Xn + a1 Xn-1 + …… + an-1 X + an
2. Represent very large numbers and operations of the large number such asaddition, multiplication and division.
3. Linked lists are to implement stack, queue, trees and graphs.
4. Implement the symbol table in compiler construction

Singly Linked List:

A linked list allocates space for each element separately in its own block of memorycalled a "node". The list gets an overall structure by using pointers to connect all itsnodes together like the links in a chain. Each node contains two fields; a "data" field tostore whatever element, and a "next" field which is a pointer used to link to the nextnode. Each node is allocated in the heap using malloc(), so the node memorycontinues to exist until it is explicitly de-allocated using free(). The front of the list is apointer to the “start” node.

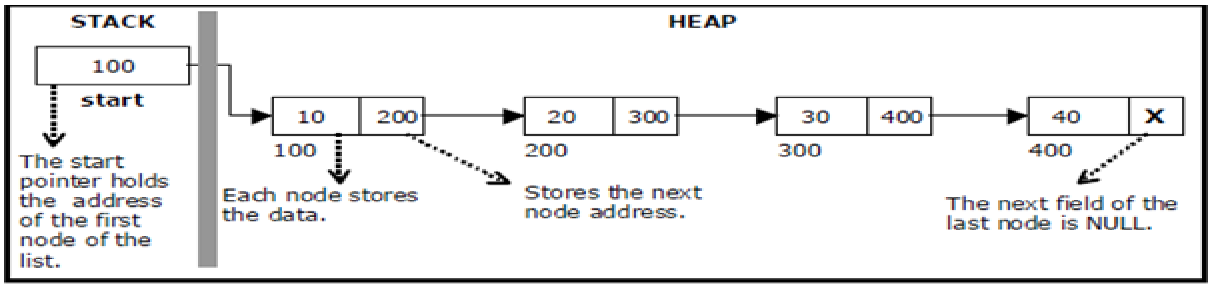

STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
10
200
100
20
300
200
30
400
300
40
X
400
Each node stores the data.
Stores the next node address.
The next field of the last node is NULL.

Figure: A single linked list

The beginning of the linked list is stored in a "**start**" pointer which points to the firstnode. The first node contains a pointer to the second node. The second node contains apointer to the third node, ... and so on. The last node in the list has its next field set toNULL to mark the end of the list. Code can access any node in the list by starting at the**start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on theleft top to show that it is in the stack. The list nodes are drawn on the right to showthat they are allocated in the heap.

Implementation of Single Linked List:

Before writing the code to build the above list, we need to create a **start** node**,** used tocreate and access other nodes in the linked list. The following structure definition will do:

• Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.

• Initialise the start pointer to be NULL.

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

The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

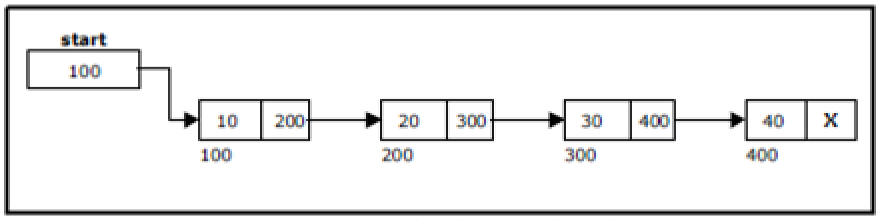
Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to beallocated for creating a node. The information is stored in the memory, allocated byusing the malloc() function. The function getnode(), is used for creating a node, afterallocating memory for the structure of type node, the information for the item (i.e.,data) has to be read from the user, set next field to NULL and finally returns theaddress of the node. Figure 3.2.3 illustrates the creation of a node for single linked list.

Creating a Singly Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:
- Get the new node using malloc().
- If the list is empty, assign new node as start.
  start = newnode;
- If the list is not empty, follow the steps given below:
- The next field of the new node is made to point the first node (i.e. start node) in the list by assigning the address of the first node.
- The start pointer is made to point the new node by assigning the address of the new node.
- Repeat the above steps 'n' times.

Figure shows 4 items in a single linked list stored at different locations in memory.



Single Linked List with 4 nodes

Insertion of a Node:

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.

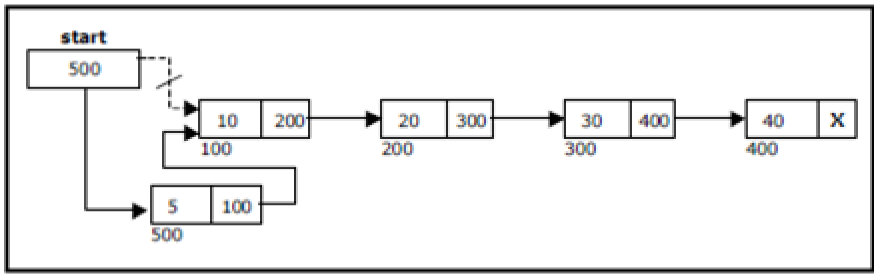
• Inserting a node at intermediate position.

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node.
- If the list is empty then *start = newnode.*
- If the list is not empty, follow the steps given below:

```
newnode -> next = start;
start = newnode;
```



Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node
- If the list is empty then *start = newnode*.
- If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> next != NULL)
        temp = temp -> next;
temp -> next = newnode;
```

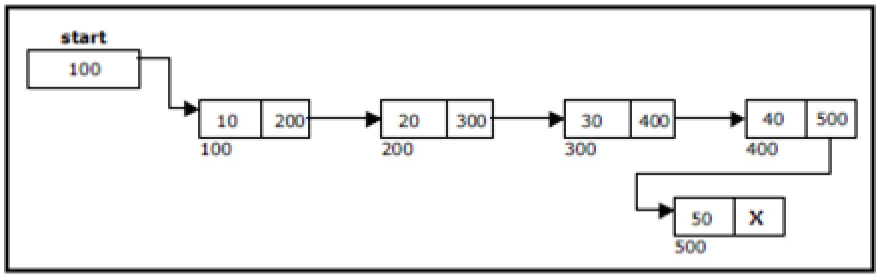


Figure ' Inserting a node at the end.

Inserting a node before a given node:

The following steps are followed, to insert a new node before a given node

- Get the new node.
- Ensure that the given node is in between first node and last node. If not, specified position is invalid.

• Store the starting address (which is in start pointer) in temp and prevpointers. Then traverse the temp pointer upto the specified node followedby prev pointer.
• After reaching the specified position, follow the steps given below:

```
prev -> next = newnode;
newnode -> next = temp;
```

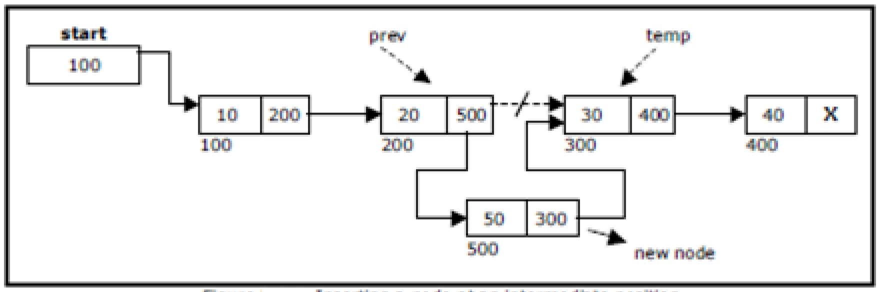


Figure : Inserting a node at an intermediate position.

Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of anode. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

• Deleting a node at the beginning.
• Deleting a node at the end.
• Deleting a node at intermediate position.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:
• If list is empty then display ‘Empty List’ message.
• If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> next;
free(temp);
```

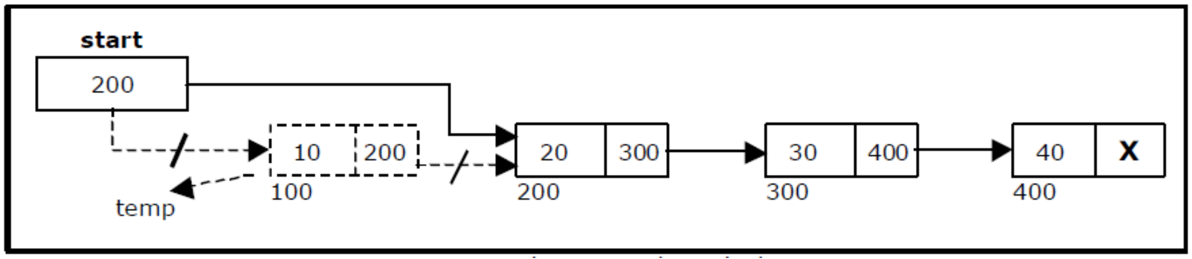


Figure: Deleting a node at the beginning.

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:
• If list is empty then display ‘Empty List’ message.
• If the list is not empty, follow the steps given below:

```
temp = prev = start;
```

```
while(temp -> next != NULL)
{
prev=temp;
temp = temp -> next;
}
prev -> next = NULL;
free(temp);
```

Figure shows deleting a node at the end of a single linked list.

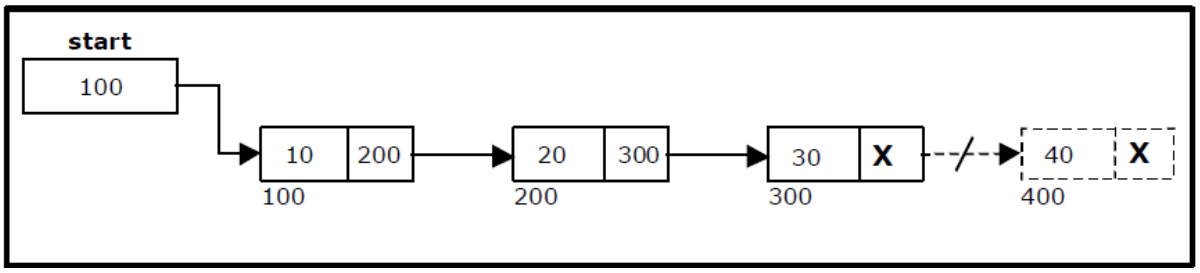


Figure . Deleting a node at the end.

Traversal and displaying a list (Left to Right):

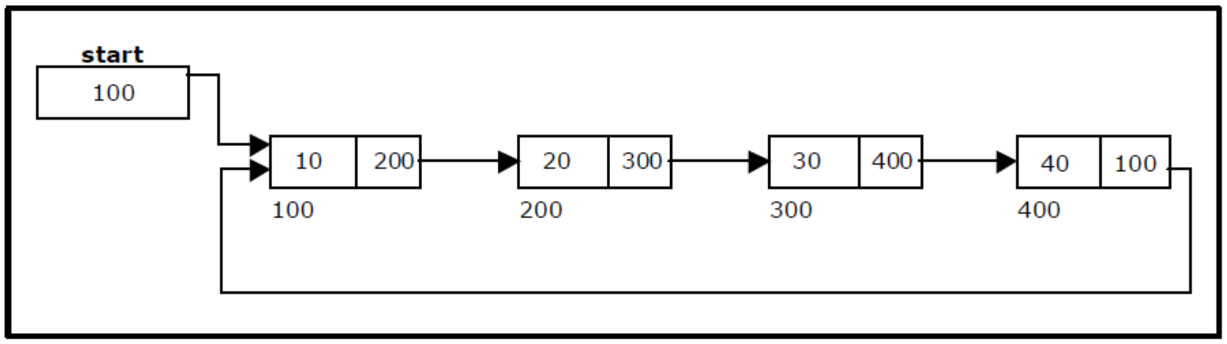
To display the information, you have to traverse (move) a linked list, node by nodefrom the first node, until the end of the list is reached. Traversing a list involves thefollowing steps:

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node.

Circular Single Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the firstnode of the list. Circular linked lists are frequently used instead of ordinary linked listbecause many operations are much easier to implement. In circular linked list no nullpointers are used, hence all pointers contain valid address.

A circular single linked list is shown in the following figure.



Circular Single Linked List

The basic operations in a circular single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a circular single Linked List with ‘n’ number of nodes:

The following steps are to be followed to create ‘n’ number of nodes:

- Get the new node
- If the list is empty, assign new node as start.

```
start = newnode;
```

- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> next != start)
    temp = temp -> next;
temp -> next = newnode;
```

- newnode -> next = start;

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the circular list:

- Get the new node using getnode().

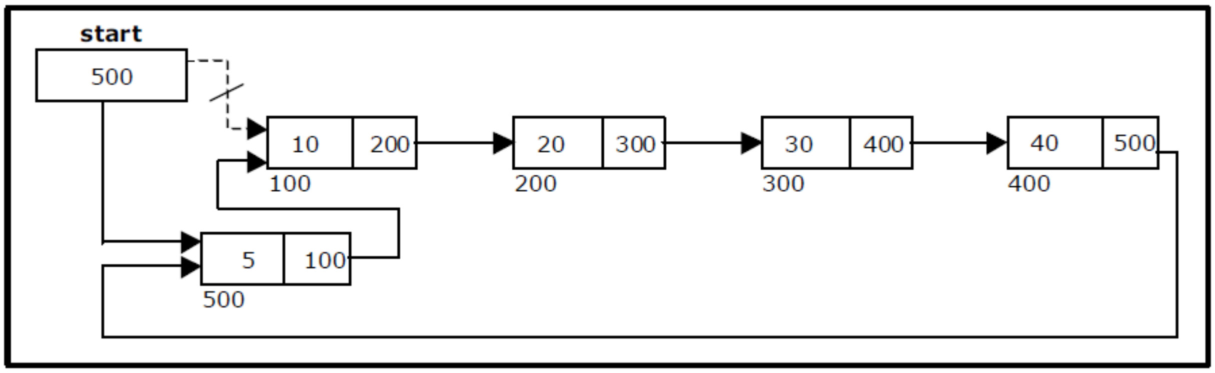
```
newnode = getnode();
```

- If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

- If the list is not empty, follow the steps given below:

```
last = start;
while(last -> next != start)
    last= last -> next;
newnode -> next = start;
start = newnode;
last -> next = start;
```



Inserting a node at the beginning

Inserting a node at the end:

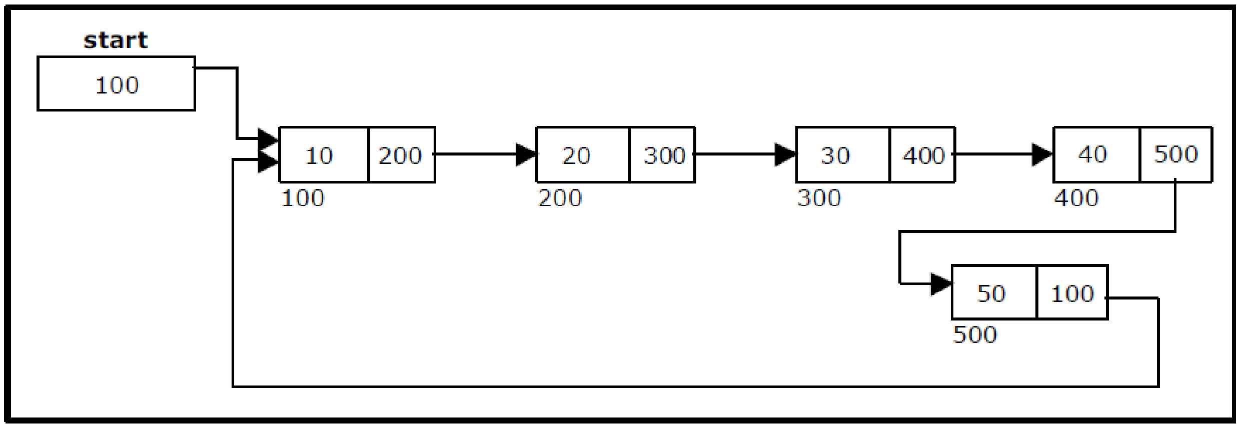
The following steps are followed to insert a new node at the end of the list:

- Get the new node
- If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

- If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> next != start)
temp = temp -> next;
temp -> next = newnode;
newnode -> next = start;
```



Inserting a node at the end

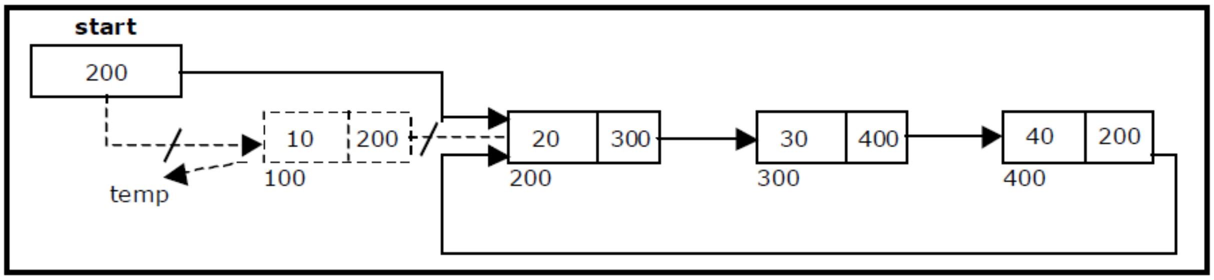
Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If the list is empty, display a message ‘Empty List’.
- If the list is not empty, follow the steps given below:

```
last = temp = start;
while(last -> next != start)
```

```
        last= last -> next;
    start = start -> next;
    last -> next = start;
```
• After deleting the node, if the list is empty then *start = NULL*.



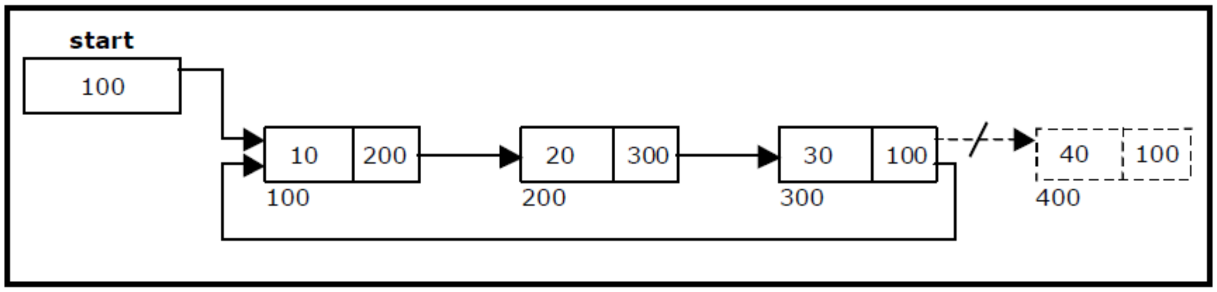
Deleting a node at beginning.

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

• If the list is empty, display a message 'Empty List'.

• If the list is not empty, follow the steps given below:
```
    temp = start;
    prev = start;
    while(temp -> next != start)
    {
        prev=temp;
        temp = temp -> next;
    }
    prev -> next = start;
```
• After deleting the node, if the list is empty then *start = NULL*.



Deleting a node at the end

Traversing a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

• If list is empty then display 'Empty List' message.

• If the list is not empty, follow the steps given below:
```
temp = start;
```

```
do
{
        printf("%d", temp -> data);
        temp = temp -> next;
} while(temp != start);
```

Doubly Linked List:

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

- Left link.
- Data.
- Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data. Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

The basic operations in a double linked list are: Creation, Insertion, Deletion and Traversing.

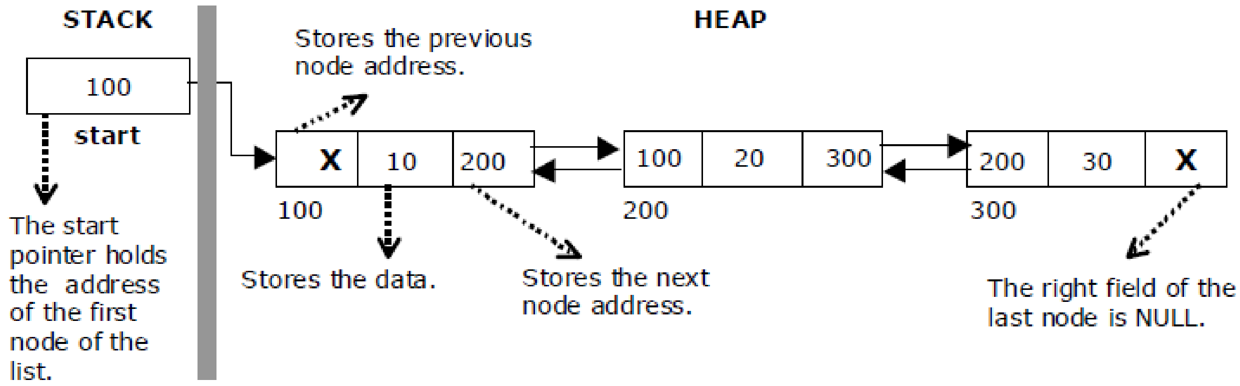

STACK
HEAP
100
start
Stores the previous node address.
X
10
200
100
20
300
200
30
X
100
200
300
The start pointer holds the address of the first node of the list.
Stores the data.
Stores the next node address.
The right field of the last node is NULL.

Figure: Doubly Linked List

The beginning of the double linked list is stored in a "start" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

The following code gives the structure definition:

```
structdlinklist
{
        structdlinklist *left;
        int data;
        structdlinklist *right;
};
typedefstructdlinklist node;
node *start=NULL
```

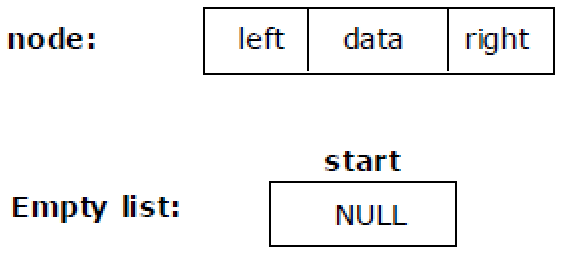


Fig: Doubly linked node and Empty node

Creating a node for a doubly linked list

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL.

```
                    node* getnode()
                    {
                              node* newnode;
                              newnode = (node *) malloc(sizeof(node));
                              printf("\n Enter data: ");
                              scanf("%d", &newnode -> data);
                              newnode -> left = NULL;
                              newnode -> right = NULL;
                              returnnewnode;
}
```

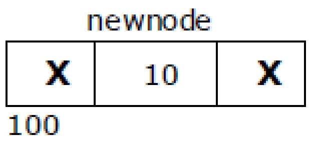


Figure:New Node with value 10

Creating a doubly linked list with 'n' nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using getnode().

```
                    newnode =getnode();
```

- If the list is empty then start = newnode.
- If the list is not empty, follow the steps given below:
- The left field of the new node is made to point the previous node.
- The previous nodes right field must be assigned with address of the new node.

• Repeat the above steps ‘n’ times.

```
voidcreatelist( int n)
{
          inti;
          node * newnode;
          node *temp;
          for( i = 0; i < n; i++)
          {
                    newno de = getno de();
                    if(start == NULL)
                    {
                              start = newno de;
                    }
                    els e
                    {
                              temp = start;
                              while(temp -> r ight)
                                        temp = temp -> r ight;
                              temp -> r ight = newno de;
                              newno de -> left = temp;
                    }
          }
}
```

The function createlist(), is used to create ‘n’ number of nodes:

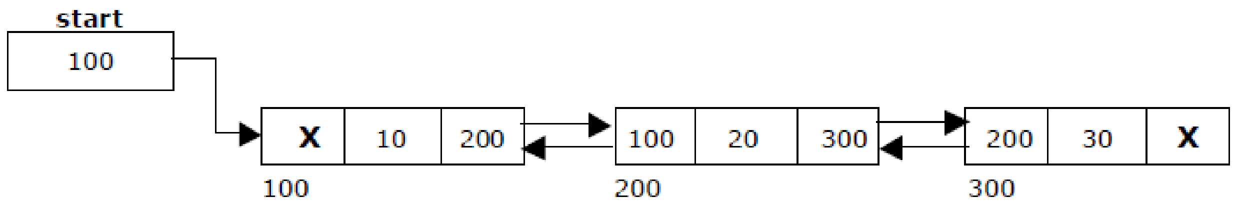


Figure:Doubly Linked List with Three nodes

Inserting a node in doubly linked list:

The following steps are to be followed to insert a new node at the beginning of the list:

• Get the new node using getnode().

newnode=getnode();

• If the list is empty then start = newnode.

• If the list is not empty, follow the steps given below:

```
newnode -> right = start;
start -> left = newnode;
start = newnode;
```

The function dbl_insert_beg(), is used for inserting a node at the beginning.

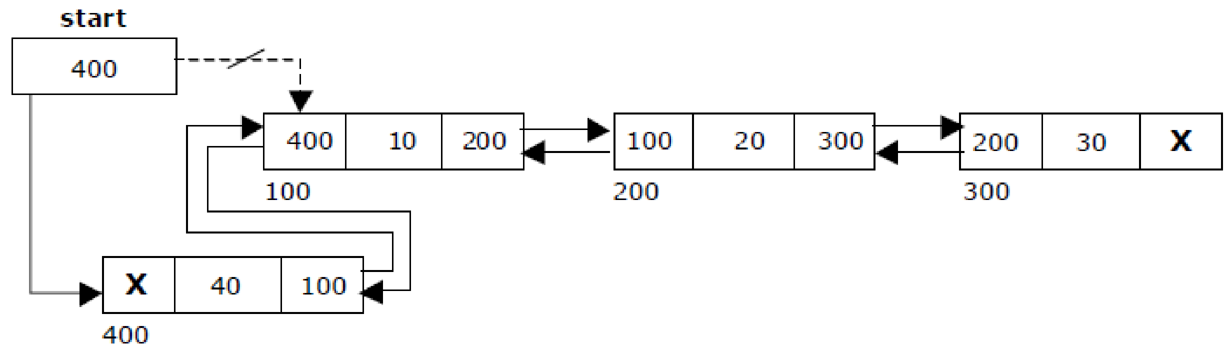


Figure:Insert a new node at beginning

Insert a node at the end:

The following steps are followed to insert a new node at the end of the list:

• Get the new node using getnode()

```
newnode=getnode();
```

• If the list is empty then start = newnode.

• If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
temp = temp -> right;
temp -> right = newnode;
newnode -> left = temp;
```

The function dbl_insert_end(), is used for inserting a node at the end.

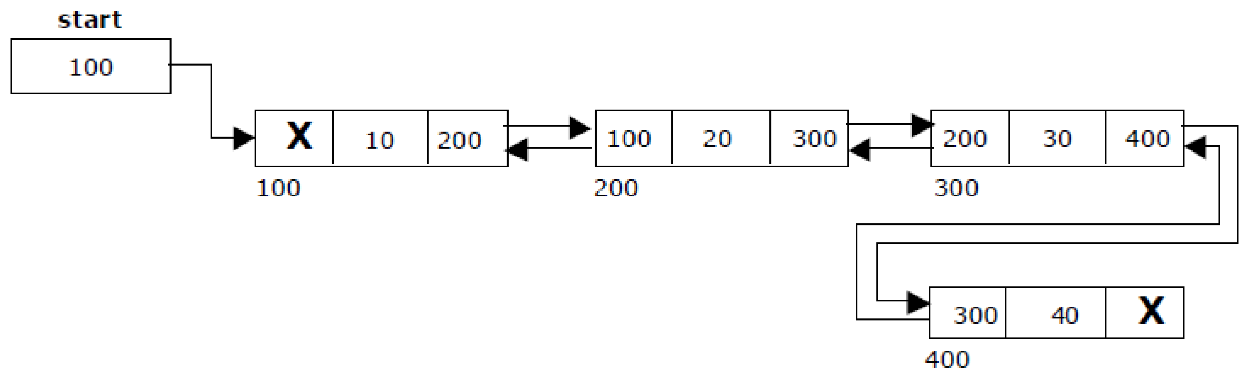


Figure:Insert the node at the End

Inserting a node at intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

• Get the new node using getnode().

```
newnode=getnode();
```

• Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
• Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
• After reaching the specified position, follow the steps given below:

```
newnode -> left = temp;
newnode -> right = temp -> right;
temp -> right -> left = newnode;
temp -> right = newnode;
```

The function dbl_insert_mid(), is used for inserting a node in the intermediate position.

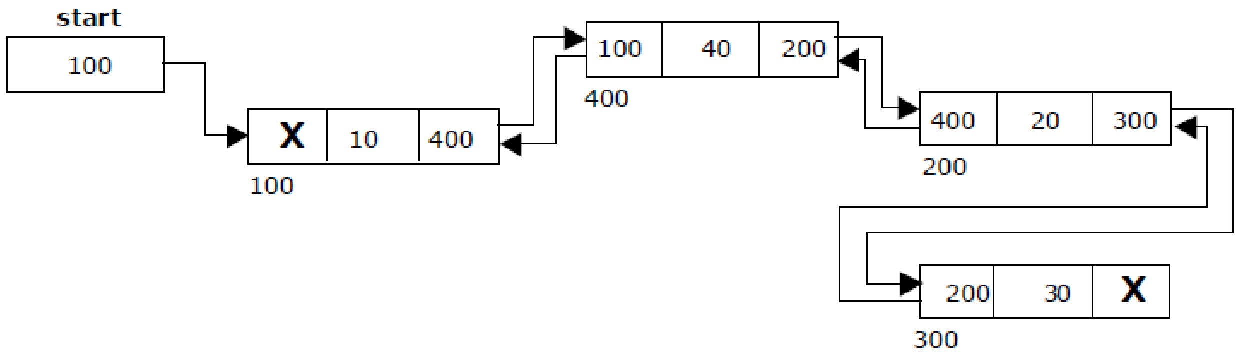


Figure:Inserting a node at Intermediate position

Delete a node at beginning:

The following steps are followed, to delete a node at the beginning of the list:
• If list is empty then display ‘Empty List’ message.
• If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```

The function dbl_delete_beg(), is used for deleting the first node in the list.

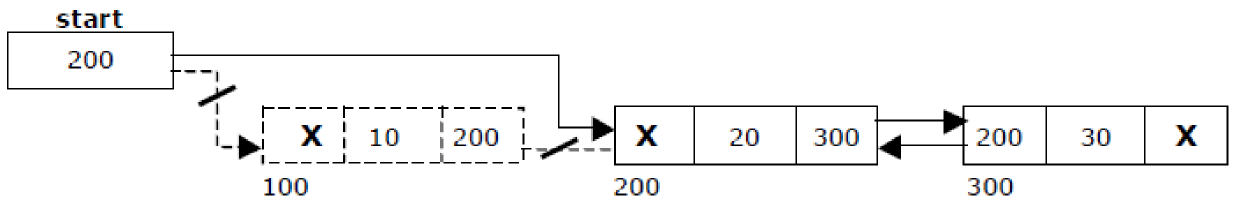


Figure: Delete a node at beginning

Delete a node at the end

The following steps are followed to delete a node at the end of the list:
• If list is empty then display ‘Empty List’ message

• If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
{
        temp = temp -> right;
}
temp -> left -> right = NULL;
free(temp);
```

The function dbl_delete_last(), is used for deleting the last node in the list.

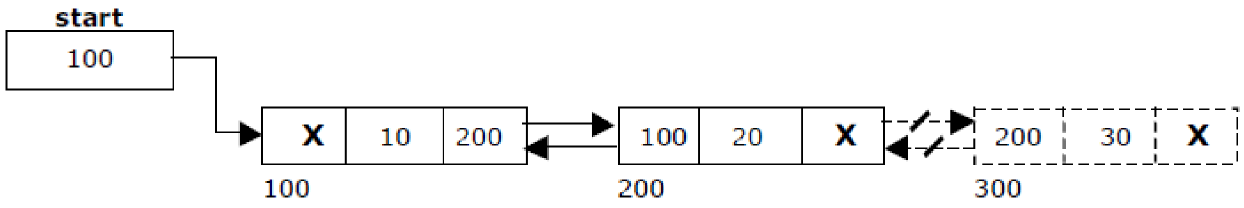


Figure: Delete the node at the end.

Delete the intermediate node:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

• If list is empty then display ‘Empty List’ message.

• If the list is not empty, follow the steps given below:

• Get the position of the node to delete.

• Ensure that the specified position is in between first node and last node. If not, specified position is invalid.

• Then perform the following steps:

```
if(pos> 1 &&pos<nodectr)
{
        temp=start;
        i=1;
        while(i<pos)
        {
                temp =temp -> right;
                i++;
        }
        temp -> right -> left = temp -> left;
        temp -> left -> right = temp -> right;
        free(temp);
        printf("\n node deleted..");
}
```

The function delete_at_mid(), is used for deleting the intermediate node in the list.

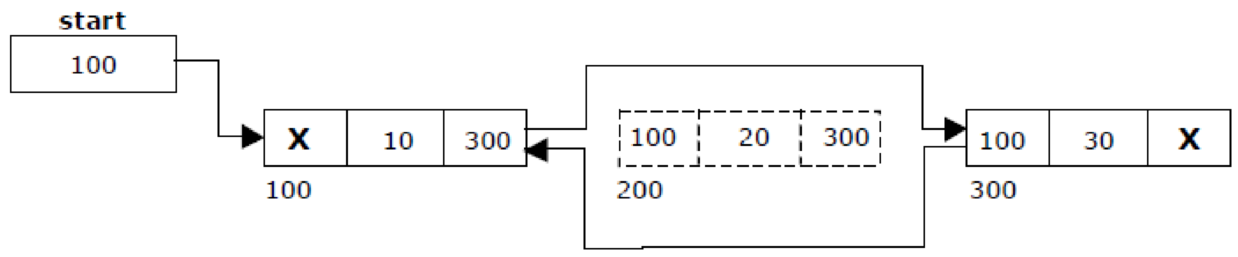


Figure : Delete the intermediate node using doubly linked list

PROGRAM FOR DOUBLY LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct dlinklist
{
        structdlinklist *left;
        int data;
        structdlinklist *right;
};
typedef  struct dlinklist node;
node *start = NULL;
node* getnode()
{
        node * newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> left = NULL;
        newnode -> right = NULL;
        returnnewnode;
}
Int countnode(node *start)
{
        if(start == NULL)
                return 0;
        else
                return 1 + countnode(start -> right);
}
int menu()
{
        intch;
        clrscr();
        printf("\n 1.Create");
        printf("\n------------------------------");
        printf("\n 2. Insert a node at beginning ");
        printf("\n 3. Insert a node at end");
        printf("\n 4. Insert a node at middle");
        printf("\n------------------------------");
        printf("\n 5. Delete a node from beginning");
        printf("\n 6. Delete a node from Last");
        printf("\n 7. Delete a node from Middle");
        printf("\n------------------------------");
        printf("\n 8. Traverse the list from Left to Right ");
        printf("\n 9. Traverse the list from Right to Left ");
        printf("\n------------------------------");
        printf("\n 10.Count the Number of nodes in the list");
        printf("\n 11.Exit");
        printf("\n\n Enter your choice: ");
        scanf("%d", &ch);
        returnch;
}
voidcreatelist(int n)
{
        inti;
        node *newnode;
        node *temp;
        for(i = 0; i < n; i++)
        {
                newnode = getnode();
                if(start == NULL)
                        start = newnode;
                else
                {
                        temp = start;
                        while(temp -> right)
                                temp = temp -> right;
                        temp -> right = newnode;
                        newnode -> left = temp;
                }
        }
}
```

```
voidtraverse_left_to_right()
{
	node *temp;
	temp = start;
	printf("\n The contents of List: ");
	if(start == NULL )
		printf("\n Empty List");
	else
	{
		while(temp != NULL)
		{
			printf("\t %d ", temp -> data);
			temp = temp -> right;
		}
	}
}
voidtraverse_right_to_left()
{
	node *temp;
	temp = start;
	printf("\n The contents of List: ");
	if(start == NULL)
		printf("\n Empty List");
	else
	{
		while(temp -> right != NULL)
			temp = temp -> right;
	}
	while(temp != NULL)
	{
		printf("\t%d", temp -> data);
		temp = temp -> left;
	}
}
voiddll_insert_beg()
{
	node *newnode;
	newnode = getnode();
	if(start == NULL)
		start = newnode;
```

```
        else
        {
                newnode -> right = start;
                start -> left = newnode;
                start = newnode;
        }
}
voiddll_insert_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
                start = newnode;
        else
        {
                temp = start;
                while(temp -> right != NULL)
                        temp = temp -> right;
                temp -> right = newnode;
                newnode -> left = temp;
        }
}
voiddll_insert_mid()
{
        node *newnode,*temp;
        intpos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos - nodectr>= 2)
        {
                printf("\n Position is out of range..");
                return;
        }
        if(pos> 1 &&pos<nodectr)
        {
                temp = start;
                while(ctr<pos - 1)
                {
```

```
                temp = temp -> right;
                ctr++;
            }
            newnode -> left = temp;
            newnode -> right = temp -> right;
            temp -> right -> left = newnode;
            temp -> right = newnode;
        }
        else
            printf("position %d of list is not a middle position ", pos);
}
voiddll_delete_beg()
{
        node *temp;
        if(start == NULL)
        {
            printf("\n Empty list");
            getch();
            return ;
        }
        else
        {
            temp = start;
            start = start -> right;
            start -> left = NULL;
            free(temp);
        }
}
voiddll_delete_last()
{
        node *temp;
        if(start == NULL)
        {
            printf("\n Empty list");
            getch();
            return ;
        }
        else
        {
            temp = start;
```

```
                while(temp -> right != NULL)
                        temp = temp -> right;
                temp -> left -> right = NULL;
                free(temp);
                temp = NULL;
        }
}
voiddll_delete_mid()
{
        inti = 0, pos, nodectr;
        node *temp;
        if(start == NULL)
        {
                printf("\n Empty List");
                getch();
                return;
        }
        else
        {
                printf("\n Enter the position of the node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos>nodectr)
                {
                        printf("\nthis node does not exist");
                        getch();
                        return;
                }
                if(pos> 1 &&pos<nodectr)
                {
                        temp = start;
                        i= 1;
                        while(i<pos)
                        {
                                temp = temp -> right;
                                i++;
                        }
                        temp -> right -> left = temp -> left;
                        temp -> left -> right = temp -> right;
                        free(temp);
```

```
                    printf("\n node deleted..");
                }
                else
                {
                    printf("\n It is not a middle position..");
                    getch();
                }
        }
}
void main(void)
{
        intch, n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch(ch)
                {
                    case 1 :
                        printf("\n Enter Number of nodes to create: ");
                        scanf("%d", &n);
                        createlist(n);
                        printf("\n List created..");
                        break;
                    case 2 :
                        dll_insert_beg();
                        break;
                    case 3 :
                        dll_insert_end();
                        break;
                    case 4 :
                        dll_insert_mid();
                        break;
                    case 5 :
                        dll_delete_beg();
                        break;
                    case 6 :
                        dll_delete_last();
                        break;
                    case 7 :
                        dll_delete_mid();
                        break;
                case 8 :
                        traverse_left_to_right();
                        break;
                case 9 :
                        traverse_right_to_left();
                        break;
                case 10 :
                        printf("\n Number of nodes: %d", countnode(start));
                        break;
                case 11:
                        exit(0);
            }
        getch();
        }
}
```