UNIT - 1
Part - 1

Basic Concepts

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.

1.1. 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. Figure 1.1 shows the classification of data structures.

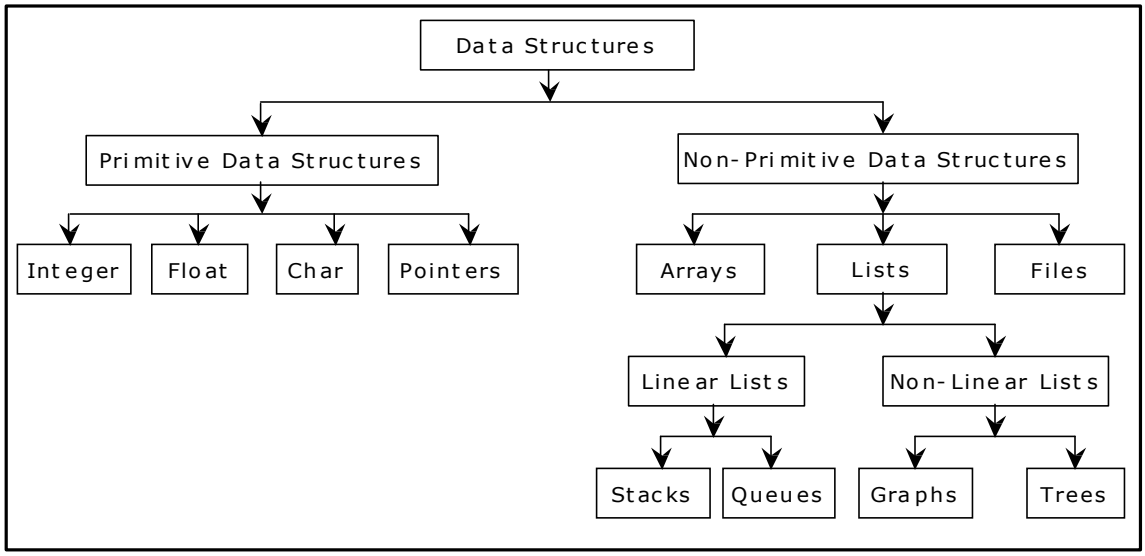


Figure 1.1. Classification of Data Structures

1.2. 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 fundamental types:

- 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.

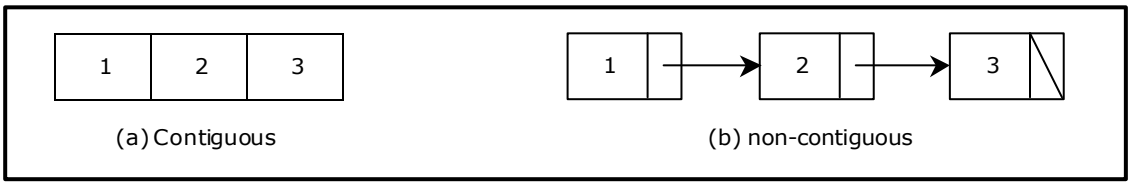


Figure 1.2 Contiguous and Non-contiguous structures compared

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.

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.

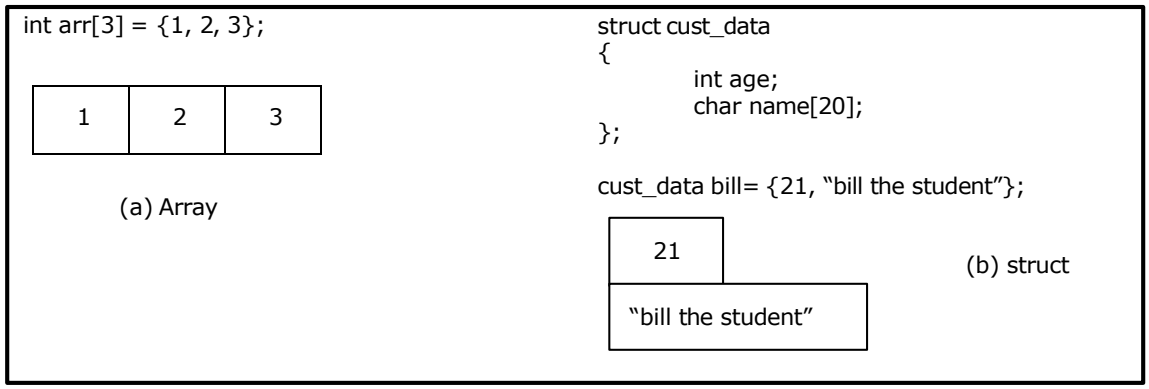


Figure 1.3 Examples of contiguous structures.

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 down 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.

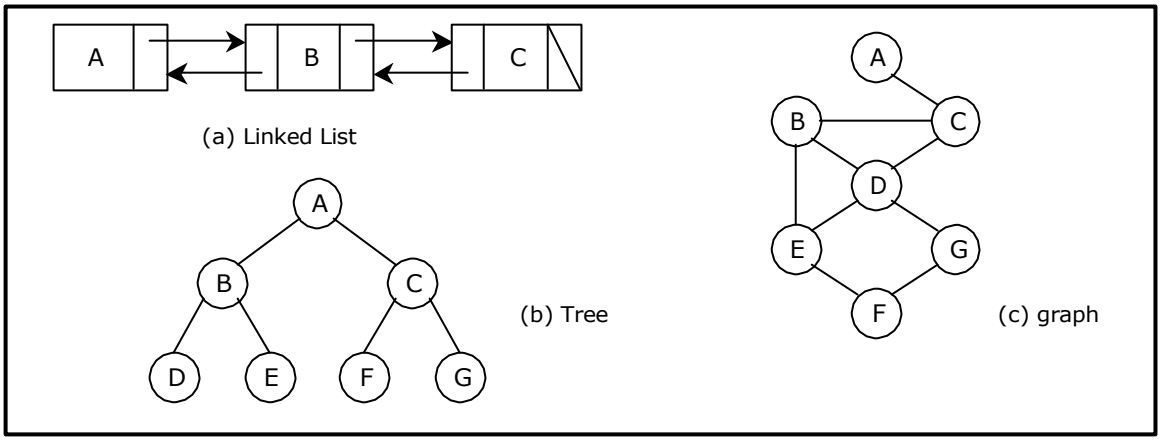


Figure 1.4. Examples of non-contiguous structures

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.

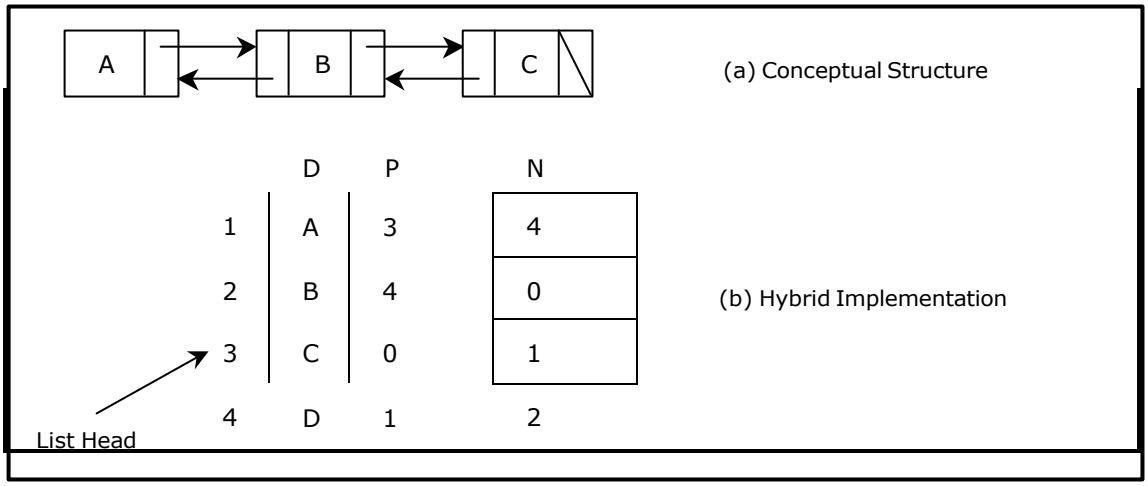


Figure 1.5. A double linked list via a hybrid data structure

The array D contains the data for the list, whereas the array P and N hold the previous and next "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.

1.3. 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 added and 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 stored 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 known

today as an ADT. We wrote the code to read a file and placed it in a library for a programmer to use.

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.

1.4. 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:

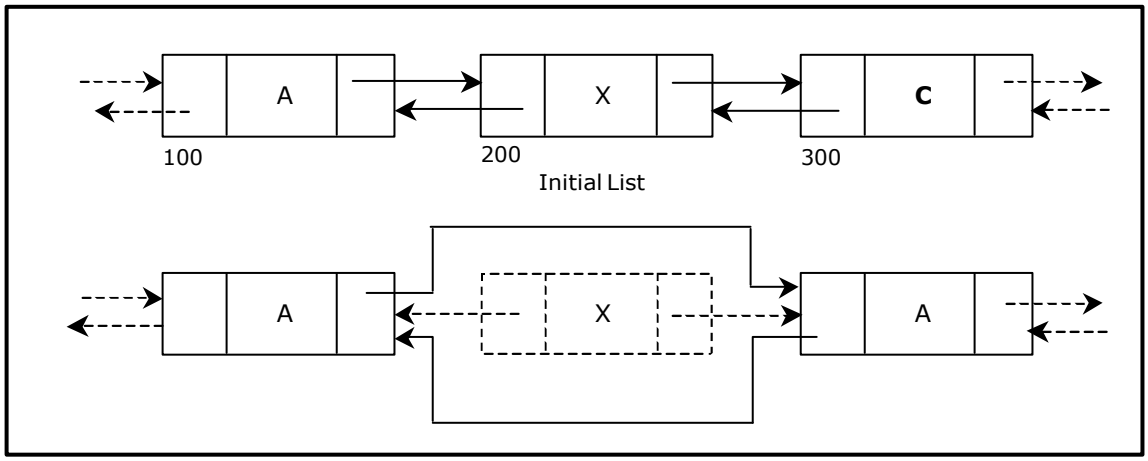


Figure 1.6 Detaching a node from a list

Since very little data is moved during this process, the deletion using linked lists will often be faster than when arrays are used.

It may seem that linked lists are superior to arrays. But is that always true? There are trade offs. Our linked lists yield faster deletions, but they take up more space because they require two extra pointers per element.

1.5. Algorithm

An **a**lgorithm 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. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent an algorithm using pseudo language that is a combination of the constructs of a programming language together with informal English statements.

1.6. Practical Algorithm design issues:

Choosing an efficient algorithm or data structure is just one part of the design process. Next, will look at some design issues that are broader in scope. There are three basic design goals that we should strive for in a program:

1. Try to save time (Time complexity).
2. Try to save space (Space complexity).
3. Try to have face.

A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to "save face" by preventing the program from locking up or generating reams of garbled data.

1.7. Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the ***TIME COMPLEXITY*** of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

1.8. Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

1 Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

Log n When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2.

n When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

n. log n This running time arises for algorithms but solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.

n^3 Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute–force" solutions to problems. Whenever n doubles, the running time squares.

1.9. Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

1.10. Rate of Growth

Big–Oh (O), Big–Omega (Ω), Big–Theta (Θ) and Little–Oh

1. $T(n) = O(f(n))$, (pronounced order of or big oh), says that the growth rate of T(n) is less than or equal ($\leq$) that of f(n)

2. $T(n) = \Omega(g(n))$ (pronounced omega), says that the growth rate of T(n) is greater than or equal to ($\geq$) that of g(n)

3. $T(n) = \Theta(h(n))$ (pronounced theta), says that the growth rate of T(n) equals (=) the growth rate of h(n) [if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$]

4. $T(n) = o(p(n))$ (pronounced little oh), says that the growth rate of T(n) is less than the growth rate of p(n) [if $T(n) = O(p(n))$ and $T(n) \neq \Theta(p(n))$].

Some Examples:

$2n^2 + 5n - 6 = O(2^n)$
$2n^2 + 5n - 6 = O(n^3)$
$2n^2 + 5n - 6 = O(n^2)$
$2n^2 + 5n - 6 \neq O(n)$

$2n^2 + 5n - 6 \neq \Omega(2^n)$
$2n^2 + 5n - 6 \neq \Omega(n^3)$
$2n^2 + 5n - 6 = \Omega(n^2)$
$2n^2 + 5n - 6 = \Omega(n)$

$2n^2 + 5n - 6 \neq \Theta(2^n)$
$2n^2 + 5n - 6 \neq \Theta(n^3)$
$2n^2 + 5n - 6 = \Theta(n^2)$
$2n^2 + 5n - 6 \neq \Theta(n)$

$2n^2 + 5n - 6 = o(2^n)$
$2n^2 + 5n - 6 = o(n^3)$
$2n^2 + 5n - 6 \neq o(n^2)$
$2n^2 + 5n - 6 \neq o(n)$

1.11. Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

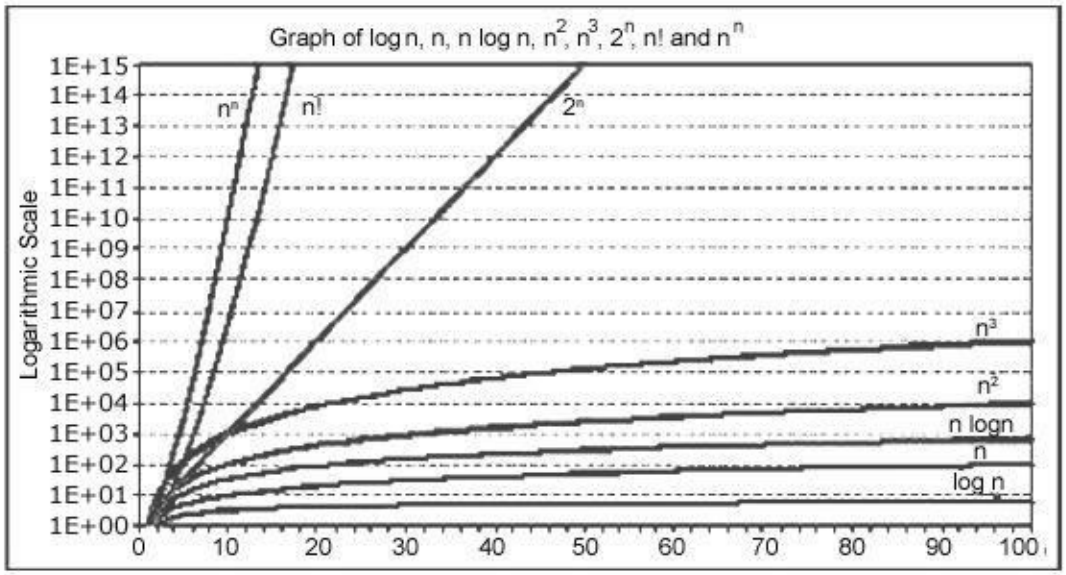
$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n. \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, $n!$ and n^n

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

S.No	log n	n	n. log n	n^2	n^3	2^n
1	0	1	1	1	1	2
2	1	2	2	4	8	4
3	2	4	8	16	64	16
4	3	8	24	64	512	256
5	4	16	64	256	4096	65536

Graph of log n, n, n log n, n^2, n^3, 2^n, n! and n^n



O(log n) does not depend on the base of the logarithm. To simplify the analysis, the convention will not have any particular units of time. Thus we throw away leading constants. We will also throw away low–order terms while computing a Big–Oh running time. Since Big-Oh is an upper bound, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.

One way to compare the function f(n) with these standard function is to use the functional 'O' notation, suppose f(n) and g(n) are functions defined on the positive integers with the property that f(n) is bounded by some multiple g(n) for almost all 'n'. Then,

f(n) = O(g(n))

Which is read as "f(n) is of order g(n)". For example, the order of complexity for:

- Linear search is O(n)
- Binary search is O(log n)
- Bubble sort is $O(n^2)$
- Quick sort is O(n log n)

For example, if the first program takes $100n^2$ milliseconds. While the second taken $5n^3$ milliseconds. Then might not $5n^3$ program better than $100n^2$ program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, $5n^3$ program be better than the $100n^2$ program.

$5\ n^3/100\ n^2 = n/20$

for inputs $n < 20$, the program with running time $5n^3$ will be faster those the one with running time $100\ n^2$.

Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was $O(n^3)$

However, as 'n' gets large, the ratio of the running times, which is n/20, gets arbitrarily larger. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function's such as O(n) or O(n log n) are always better.

Exercises

1. Define algorithm.

2. State the various steps in developing algorithms?

3. State the properties of algorithms.

4. Define efficiency of an algorithm?

5. State the various methods to estimate the efficiency of an algorithm.

6. Define time complexity of an algorithm?

7. Define worst case of an algorithm.

8. Define average case of an algorithm.

9. Define best case of an algorithm.

10. Mention the various spaces utilized by a program.

11. Define space complexity of an algorithm.

12. State the different memory spaces occupied by an algorithm.

Multiple Choice Questions

1. _____is a step-by-step recipe for solving an instance of problem. [A]
   A. Algorithm B. Complexity
   C. Pseudocode D. Analysis

2. _____is used to describe the algorithm, in less formal language. [C]
   A. Cannot be defined B. Natural Language
   C. Pseudocode D. None

3. _____of an algorithm is the amount of time (or the number of steps) needed by a program to complete its task. [D]
   A. Space Complexity B. Dynamic Programming
   C. Divide and Conquer D. Time Complexity

4. _____of a program is the amount of memory used at once by the algorithm until it completes its execution. [C]

   A. Divide and Conquer B. Time Complexity
   C. Space Complexity D. Dynamic Programming

5. _____is used to define the worst-case running time of an algorithm. [A]
   A. Big-Oh notation B. Cannot be defined
   C. Complexity D. Analysis

UNIT - 1
Part - 2

LINKED LISTS

In this chapter, the list data structure is presented. This structure can be used as the basis for the implementation of other data structures (stacks, queues etc.). The basic linked list can be used without modification in many programs. However, some applications require enhancements to the linked list design. These enhancements fall into three broad categories and yield variations on linked lists that can be used in any combination: circular linked lists, double linked lists and lists with header nodes.

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.

3.1. 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.

3.2. 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 are expression containing terms with non zero coefficient and exponents. For example:

   $P(x) = a_0 X^n + a_1 X^{n-1} + \ldots\ldots + a_{n-1} X + a_n$

2. Represent very large numbers and operations of the large number such as addition, multiplication and division.

3. Linked lists are to implement stack, queue, trees and graphs.

4. Implement the symbol table in compiler construction

3.3. Single Linked List:

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

A single linked list is shown in figure 3.2.1.

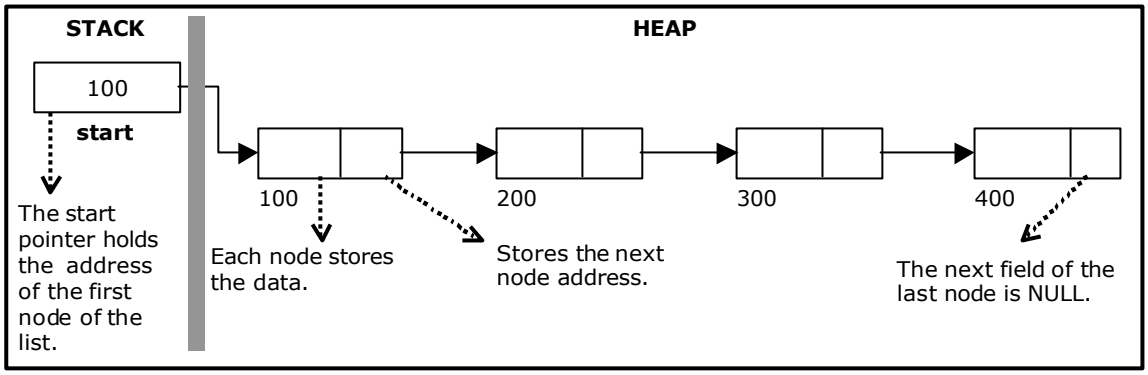


Figure 3.2.1. Single Linked List

The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL 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 the left top to show that it is in the stack. The list nodes are drawn on the right to show that 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 to create and access other nodes in the linked list. The following structure definition will do (see figure 3.2.2):

- 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 slinklist
{
        int data;
        struct slinklist* next;
};

typedef struct slinklist node;

node *start = NULL;
```

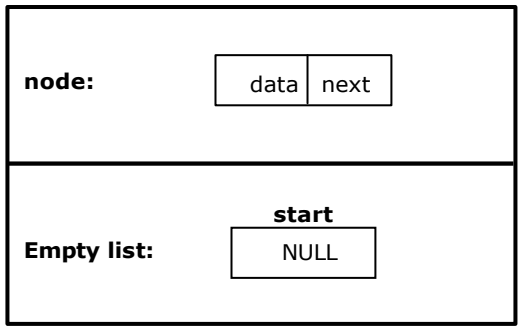


Figure 3.2.2. Structure definition, single link node and empty list

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 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, set next field to NULL and finally returns the address of the node. Figure 3.2.3 illustrates the creation of a node for single linked list.

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

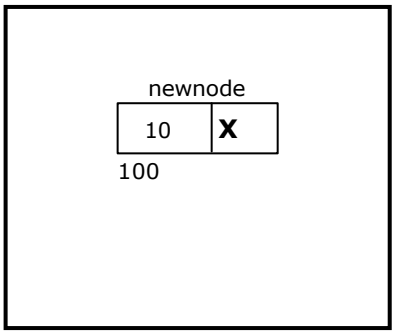


Figure 3.2.3. new node with a value of 10

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 getnode().
  newnode = getnode();
- 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 3.2.4 shows 4 items in a single linked list stored at different locations in memory.

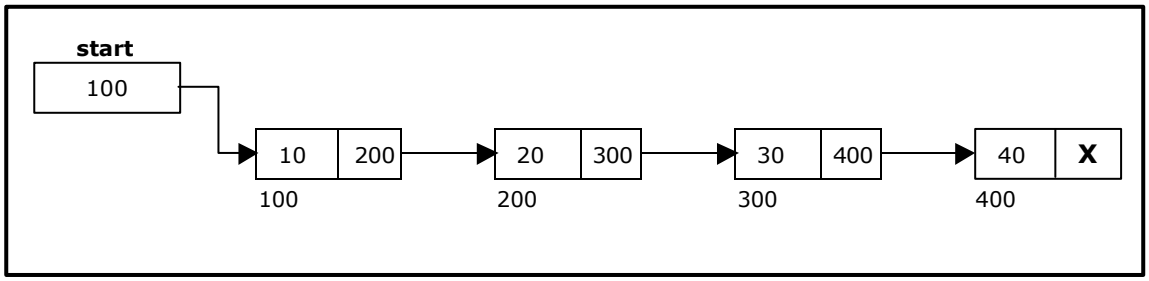


Figure 3.2.4. Single Linked List with 4 nodes

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

```
void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n ; i++)
        {
                newnode = getnode();
                if(start == NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> next != NULL)
                                temp = temp -> next;
                        temp -> next = newnode;
                }
        }
}
```

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 using getnode().
  newnode = getnode();
- If the list is empty then *start = newnode.*
- If the list is not empty, follow the steps given below:
  newnode -> next = start;
  start = newnode;

Figure 3.2.5 shows inserting a node into the single linked list at the beginning.

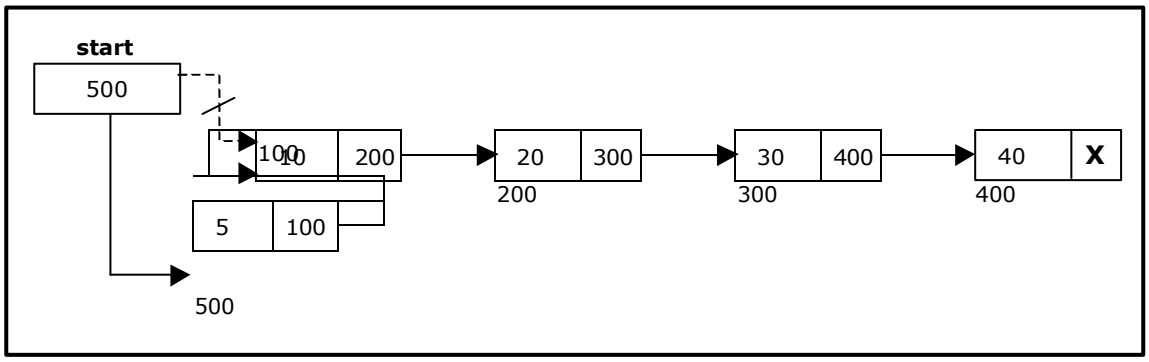


Figure 3.2.5. Inserting a node at the beginning

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

```
void insert_at_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                newnode -> next = start;
                start = newnode;
        }
}
```

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 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 -> next != NULL)
  temp = temp -> next;
  temp -> next = newnode;

Figure 3.2.6 shows inserting a node into the single linked list at the end.

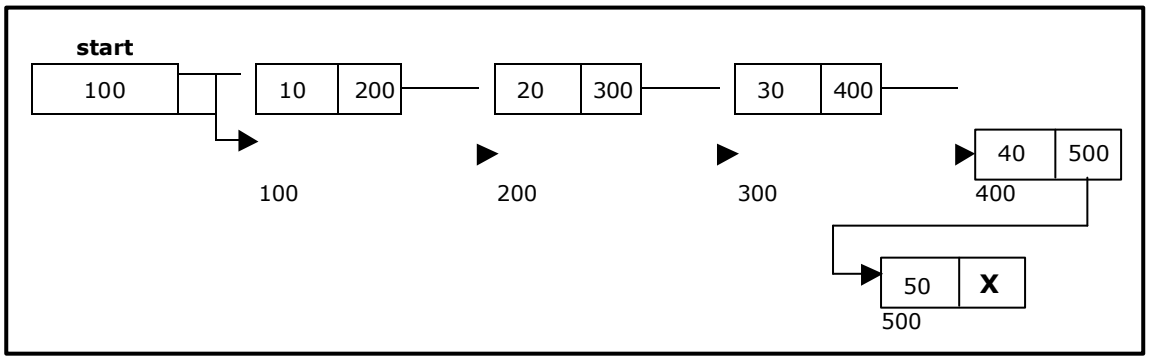


Figure 3.2.6. Inserting a node at the end.

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

```
void insert_at_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                temp = start;
                while(temp -> next != NULL)
                        temp  = temp -> next;
                temp -> next = newnode;
        }
}
```

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

- Let the intermediate position be 3.

Figure 3.2.7 shows inserting a node into the single linked list at a specified intermediate position other than beginning and end.

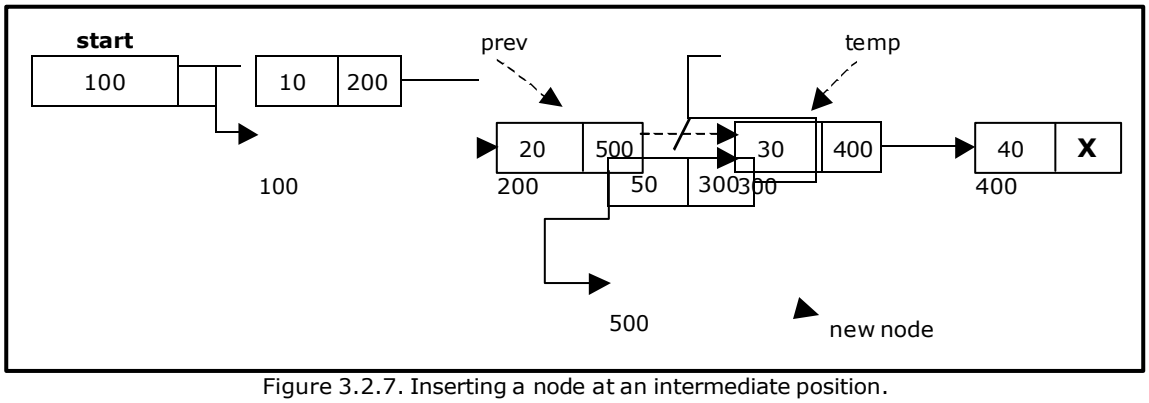


Figure 3.2.7. Inserting a node at an intermediate position.

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

```
void insert_at_mid()
{
        node *newnode, *temp, *prev;
        int  pos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
        {
                printf("position %d is not a middle position", pos);
        }
}
```

Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of a node. 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 3.2.8 shows deleting a node at the beginning of a single linked list.

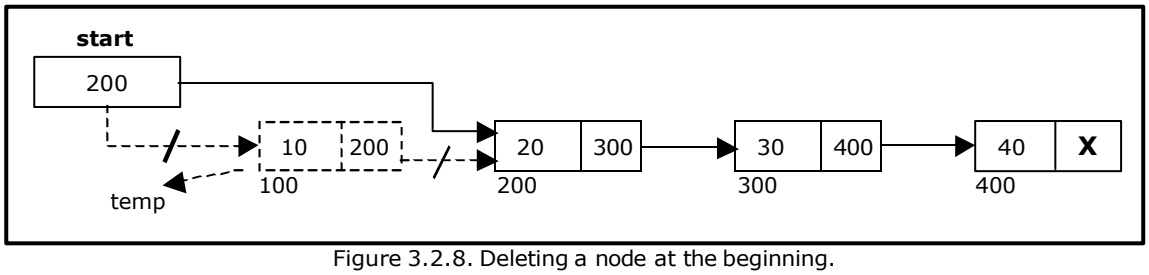


Figure 3.2.8. Deleting a node at the beginning.

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

```
void delete_at_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes are exist..");
                return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("\n Node deleted ");
        }
}
```

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

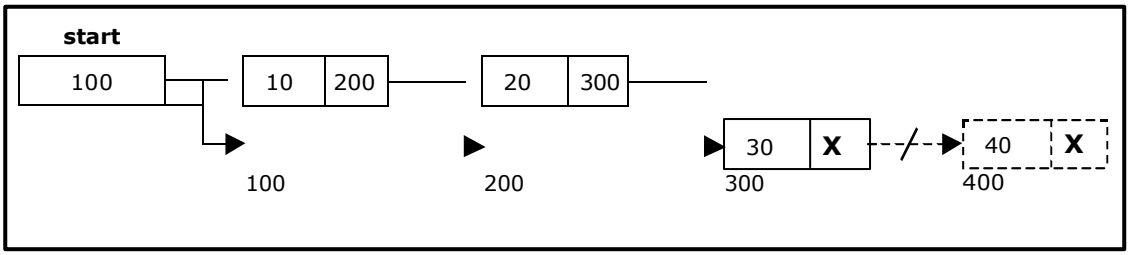


Figure 3.2.9. Deleting a node at the end.

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

```
void delete_at_last()
{
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = NULL;
                free(temp);
                printf("\n Node deleted ");
        }
}
```

Deleting a node at Intermediate position:

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

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below.

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

Figure 3.2.10 shows deleting a node at a specified intermediate position other than beginning and end from a single linked list.

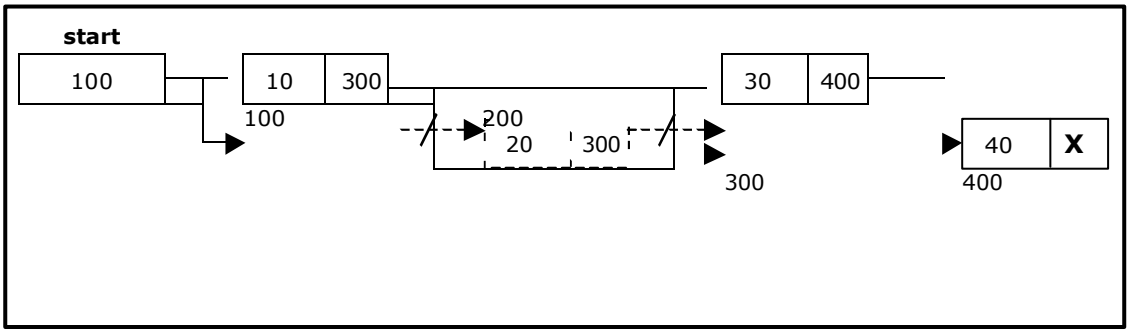


Figure 3.2.10. Deleting a node at an intermediate position.

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

```
void delete_at_mid()
{
        int ctr = 1, pos, nodectr;
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                printf("\n Enter position of node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf("\nThis node doesnot exist");
                }
```

```
                if(pos > 1 && pos < nodectr)
                {
                        temp = prev = start;
                        while(ctr < pos)
                        {
                                prev = temp;
                                temp = temp -> next;
                                ctr ++;
                        }
                        prev -> next = temp -> next;
                        free(temp);
                        printf("\n Node deleted..");
                }
                else
                {
                        printf("\n Invalid position..");
                        getch();
                }

        }
}
```

Traversal and displaying a list (Left to Right):

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

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

The function *traverse*() is used for traversing and displaying the information stored in the list from left to right.

```
void traverse()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right): \n");
        if(start == NULL )
                printf("\n Empty List");
        else
        {
                while (temp != NULL)
                {
                        printf("%d ->", temp -> data);
                        temp = temp -> next;
                }
        }
        printf("X");
}
```

Alternatively there is another way to traverse and display the information. That is in reverse order. The function rev_traverse(), is used for traversing and displaying the information stored in the list from right to left.

```
void rev_traverse(node *st)
{
	if(st == NULL)
	{
		return;
	}
	else
	{
		rev_traverse(st -> next);
		printf("%d ->", st -> data);
	}
}
```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list using *recursion*.

```
int countnode(node *st)
{
	if(st == NULL)
		return 0;
	else
		return(1 + countnode(st -> next));
}
```

3.3.1. Source Code for the Implementation of Single Linked List:

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

struct slinklist
{
	int data;
	struct slinklist *next;
};

typedef struct slinklist node;

node *start = NULL;
int menu()
{
	int ch;
	clrscr();
	printf("\n 1.Create a list ");
	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 (Left to Right)");
	printf("\n 9.Traverse the list (Right to Left)");
```

```
        printf("\n-------------------------");
        printf("\n 10. Count nodes ");
        printf("\n 11. Exit ");
        printf("\n\n Enter your choice: ");
        scanf("%d",&ch);
        return ch;
}

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

int countnode(node *ptr)
{
        int count=0;
        while(ptr != NULL)
        {
                count++;
                ptr = ptr -> next;
        }
        return (count);
}

void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n; i++)
        {
                newnode = getnode();
                if(start == NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> next != NULL)
                                temp = temp -> next;
                        temp -> next = newnode;
                }
        }
}

void traverse()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right): \n");
        if(start == NULL)
        {
                printf("\n Empty List");
                return;
        }
        else
        {
```

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

void rev_traverse(node *start)
{
        if(start == NULL)
        {
                return;
        }
        else
        {
                rev_traverse(start -> next);
                printf("%d -->", start -> data);
        }
}

void insert_at_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                newnode -> next = start;
                start = newnode;
        }
}

void insert_at_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                temp = start;
                while(temp -> next != NULL)
                        temp  = temp -> next;
                temp -> next = newnode;
        }
}

void insert_at_mid()
{
        node *newnode, *temp, *prev;
        int  pos, nodectr, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
```

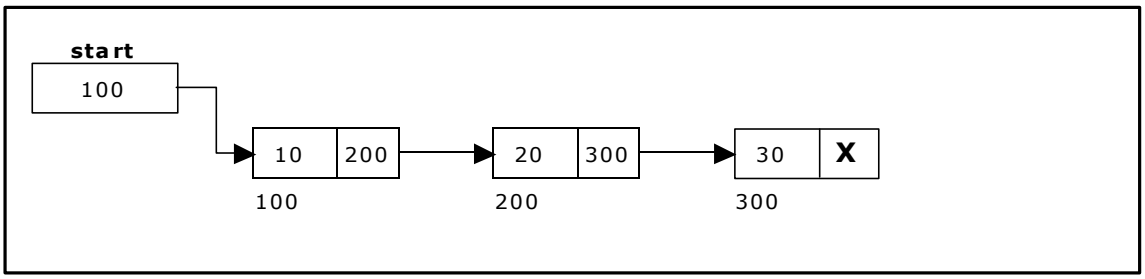
```
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
                printf("position %d is not a middle position", pos);
}

void delete_at_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes are exist..");
                return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("\n Node deleted ");
        }
}

void delete_at_last()
{
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = NULL;
                free(temp);
                printf("\n Node deleted ");
        }
}

void delete_at_mid()
{
        int ctr = 1, pos, nodectr;
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
```

```
                return ;
        }
        else
        {
                printf("\n Enter position of node to delete: ");
                scanf("%d", &pos);
                nodectr = countnode(start);
                if(pos > nodectr)
                {
                        printf("\nThis node doesnot exist");

                }
                if(pos > 1 && pos < nodectr)
                {
                        temp = prev = start;
                        while(ctr < pos)
                        {
                                prev = temp;
                                temp = temp -> next;
                                ctr ++;
                        }
                        prev -> next = temp -> next;
                        free(temp);
                        printf("\n Node deleted..");
                }
                else
                {
                        printf("\n Invalid position..");
                        getch();
                }
        }
}

void main(void)
{
        int  ch, n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch(ch)
                {
                case 1:
                        if(start == NULL)
                        {
                                printf("\n Number of nodes you want to create: ");
                                scanf("%d", &n);
                                createlist(n);
                                printf("\n List created..");
                        }
                        else
                                printf("\n List is already created..");
                                break;
                case 2:
                        insert_at_beg();
                        break;
                case 3:
                        insert_at_end();
                        break;
                case 4:
                        insert_at_mid();
                        break;
```

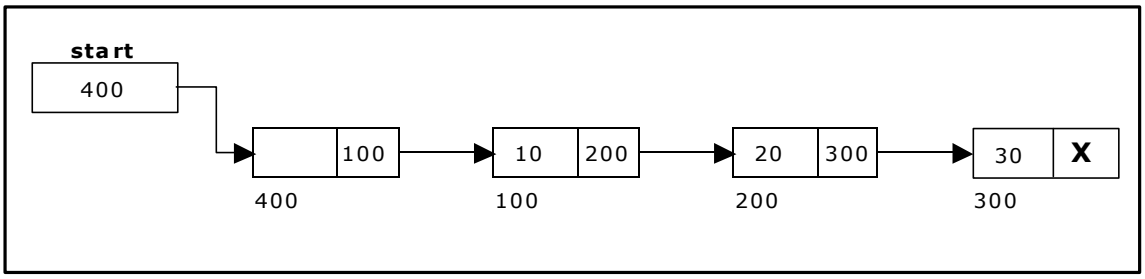
```
                case 5:
                        delete_at_beg();
                        break;
                case 6:
                        delete_at_last();
                        break;
                case 7:
                        delete_at_mid();
                        break;
                case 8:
                        traverse();
                        break;
                case 9:
                        printf("\n The contents of List (Right to Left): \n");
                        rev_traverse(start);
                        printf(" X ");
                        break;
                case 10:
                        printf("\n No of nodes : %d ", countnode(start));
                        break;
                case 11 :
                        exit(0);
                }
                getch();
        }
}
```

3.4. Using a header node:

A header node is a special dummy node found at the front of the list. The use of header node is an alternative to remove the first node in a list. For example, the picture below shows how the list with data 10, 20 and 30 would be represented using a linked list without and with a header node:



Single Linked List without a header node



Single Linked List with header node

Note that if your linked lists do include a header node, there is no need for the special case code given above for the *remove* operation; node ***n*** can never be the first node in the list, so there is no need to check for that case. Similarly, having a header node can simplify the code that adds a node before a given node ***n***.

Note that if you do decide to use a header node, you must remember to initialize an empty list to contain one (dummy) node, you must remember not to include the header node in the count of "real" nodes in the list.

It is also useful when information other than that found in each node of the list is needed. For example, imagine an application in which the number of items in a list is often calculated. In a standard linked list, the list function to count the number of nodes has to traverse the entire list every time. However, if the current length is maintained in a header node, that information can be obtained very quickly.

3.5. Array based linked lists:

Another alternative is to allocate the nodes in blocks. In fact, if you know the maximum size of a list a head of time, you can pre-allocate the nodes in a single array. The result is a hybrid structure – an array based linked list. Figure 3.5.1 shows an example of null terminated single linked list where all the nodes are allocated contiguously in an array.

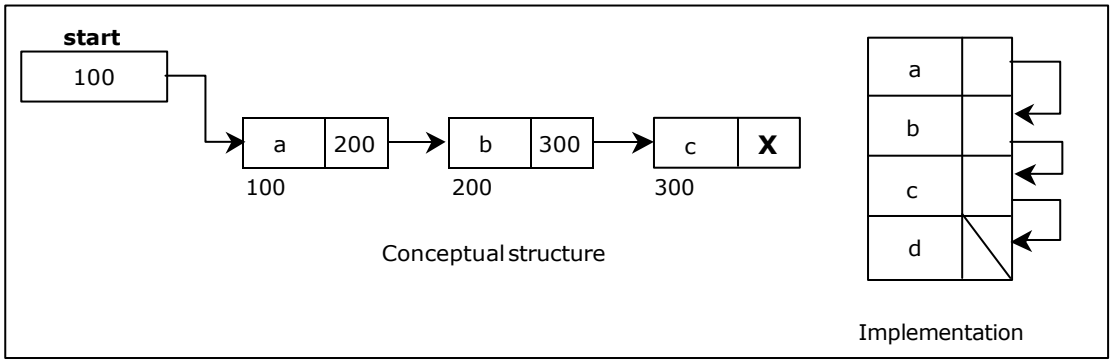


Figure 3.5.1. An array based linked list

3.6. Double 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.
- Traversing.

A double linked list is shown in figure 3.3.1.

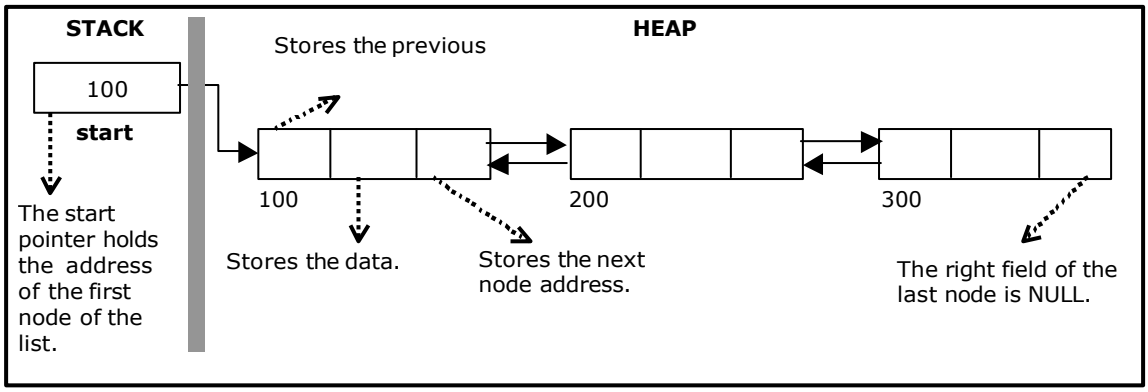


Figure 3.3.1. Double Linked List

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:

```
struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;

};

typedef struct dlinklist node;
node *start = NULL;
```

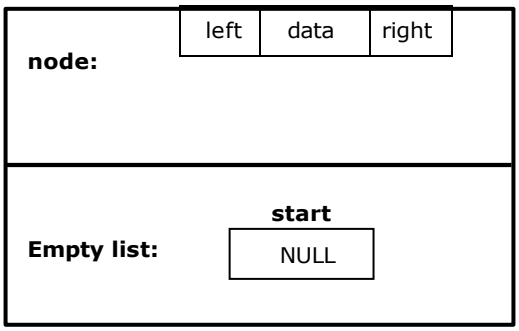


Figure 3.4.1. Structure definition, double link node and empty list

Creating a node for Double 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 (see figure 3.2.2).

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

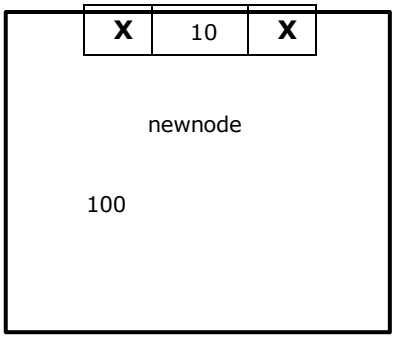


Figure 3.4.2. new node with a value of 10

Creating a Double 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 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.

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

```
void createlist(int n)
{
        int i;
        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;
                }
        }
}
```

Figure 3.4.3 shows 3 items in a double linked list stored at different locations.

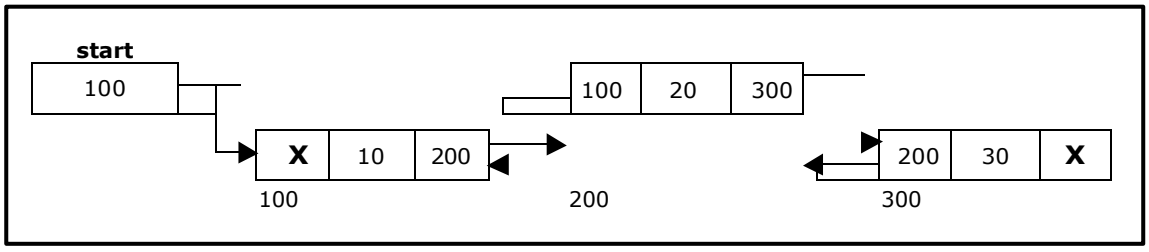


Figure 3.4.3. Double Linked List with 3 nodes

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 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 3.4.4 shows inserting a node into the double linked list at the beginning.

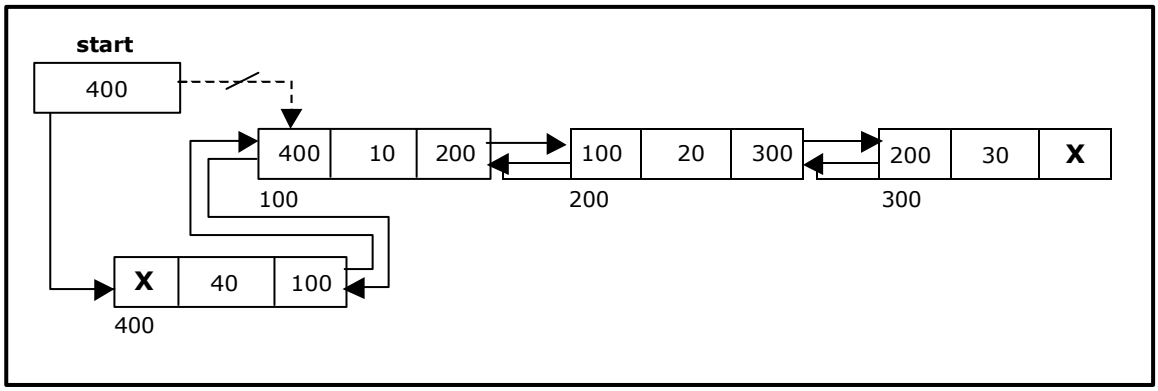


Figure 3.4.4. Inserting a node at the beginning

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 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 3.4.5 shows inserting a node into the double linked list at the end.

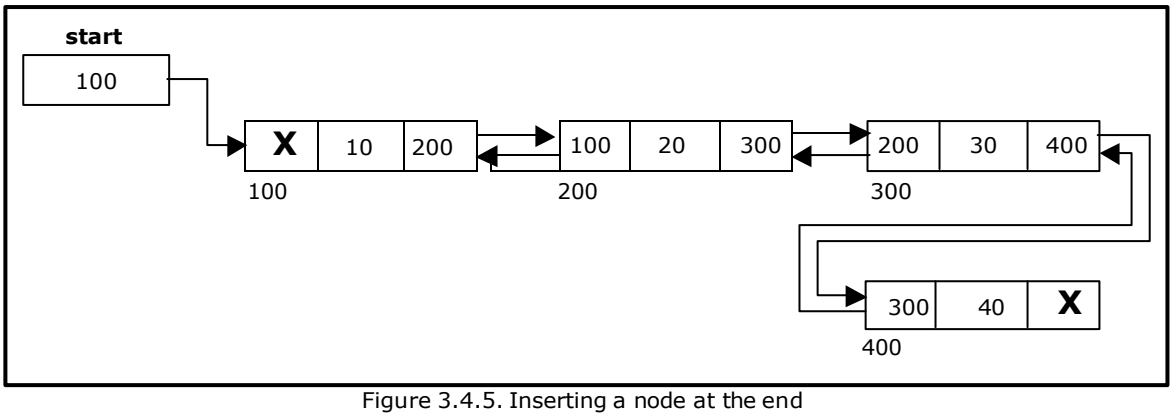


Figure 3.4.5. Inserting a node at the end

Inserting a node at an 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 3.4.6 shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.

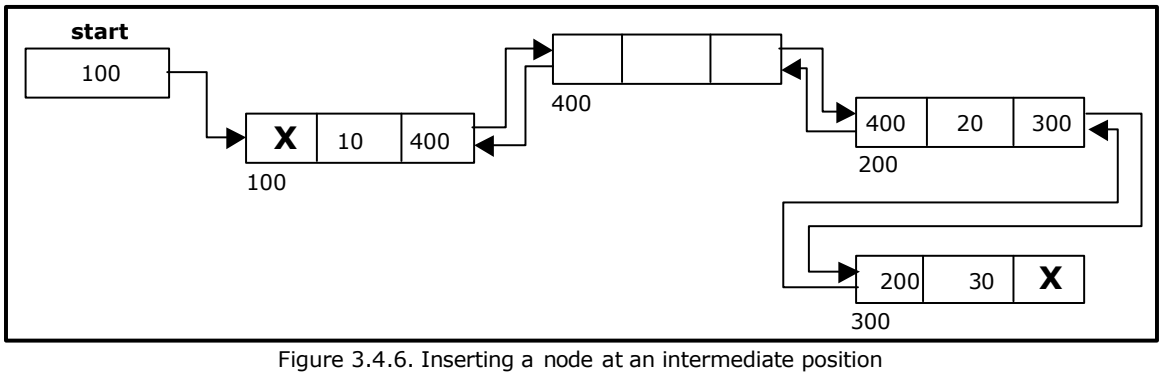


Figure 3.4.6. Inserting a node at an intermediate position

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

The function dbl_delete_beg(), is used for deleting the first node in the list. Figure 3.4.6 shows deleting a node at the beginning of a double linked list.

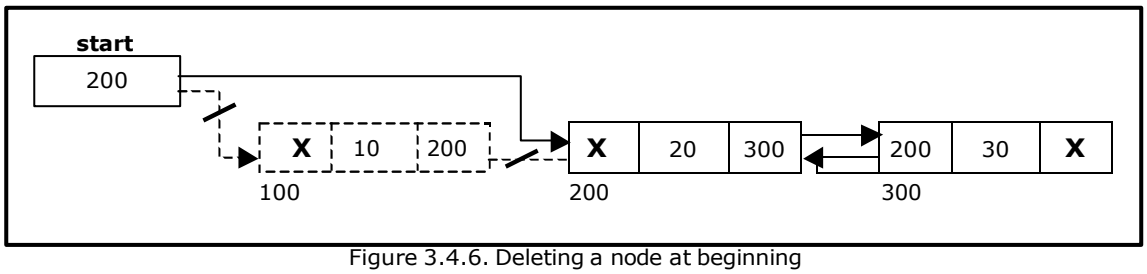


Figure 3.4.6. Deleting a node at beginning

Deleting a node at the end:

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

- If 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 3.4.7 shows deleting a node at the end of a double linked list.

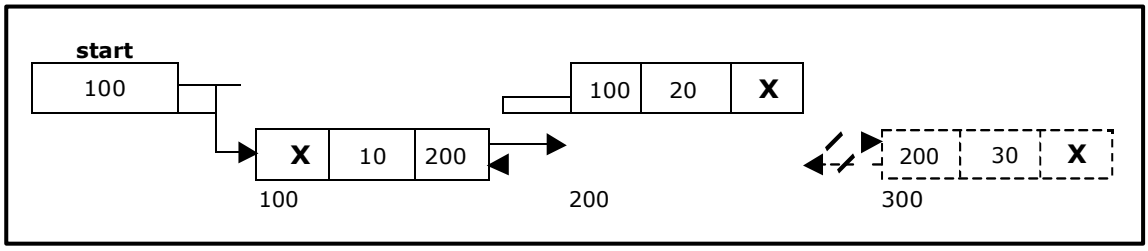


Figure 3.4.7. Deleting a node at the end

Deleting a node at Intermediate position:

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 3.4.8 shows deleting a node at a specified intermediate position other than beginning and end from a double linked list.

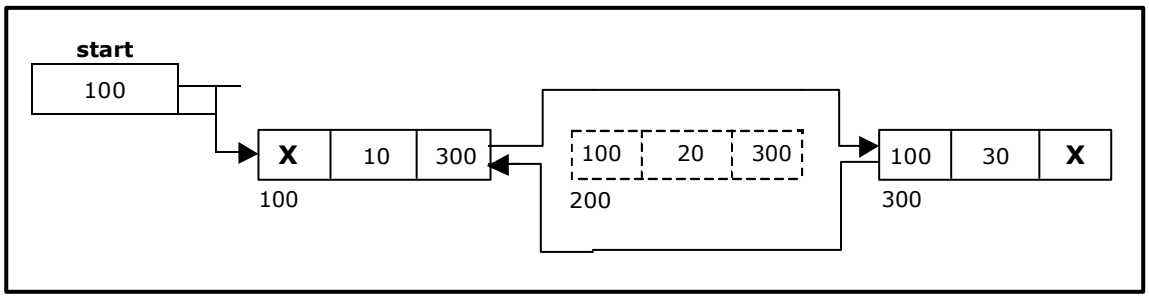


Figure 3.4.8 Deleting a node at an intermediate position

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_left_right*() is used for traversing and displaying the information stored in the 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;
while(temp != NULL)
{
        print  temp -> data;
        temp = temp -> right;
}
```

Traversal and displaying a list (Right to Left):

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_right_left*() is used for traversing and displaying the information stored in the list from right to left. The following steps are followed, to traverse a list from right to left:

- 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;
while(temp != NULL)
{
        print temp -> data;
        temp = temp -> left;
}
```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list (using recursion).

```
int countnode(node *start)
{
       if(start == NULL)
              return 0;
       else
              return(1 + countnode(start ->right ));
}
```

3.5. A Complete Source Code for the Implementation of Double Linked List:

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

struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *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;
        return newnode;
}

int countnode(node *start)
{
        if(start == NULL)
                return 0;
        else
                return 1 + countnode(start -> right);
}

int menu()
{
        int ch;
        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);
        return ch;
}

void createlist(int n)
{
        int i;
        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;
                }
        }
}
```

```
void traverse_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;
                }
        }
}
void traverse_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;
        }
}
void dll_insert_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
                start = newnode;
        else
        {
                newnode -> right = start;
                start -> left = newnode;
                start = newnode;
        }
}

void dll_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;
        }
}
```

```
void dll_insert_mid()
{
        node *newnode,*temp;
        int pos, 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);
}


void dll_delete_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n Empty list");
                getch();
                return ;
        }
        else
        {
                temp = start;
                start = start -> right;
                start -> left = NULL;
                free(temp);
        }
}


void dll_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;
        }
}

void dll_delete_mid()
{
        int i = 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)
{
        int ch, 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();
        }
}
```

3.7. 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 first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular single linked list is shown in figure 3.6.1.

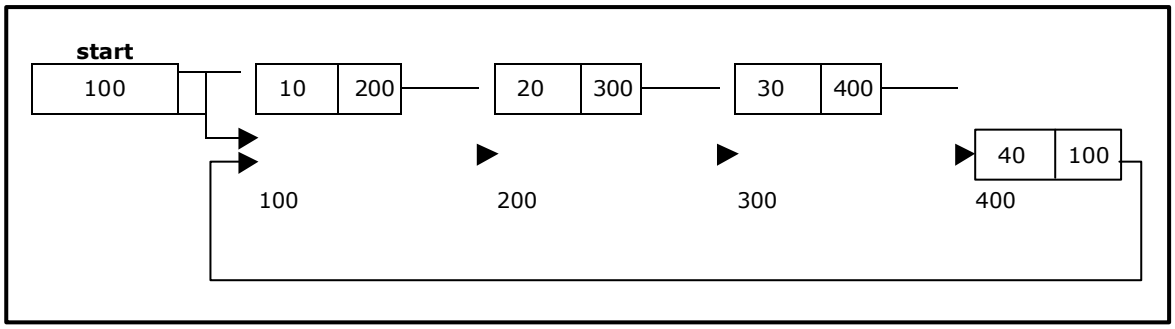


Figure 3.6.1. Circular Single Linked List

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 using getnode().

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

- 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 != NULL)
        temp = temp -> next;
temp -> next = newnode;
```

- Repeat the above steps 'n' times.

- newnode -> next = start;

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

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;
```

The function cll_insert_beg(), is used for inserting a node at the beginning. Figure 3.6.2 shows inserting a node into the circular single linked list at the beginning.

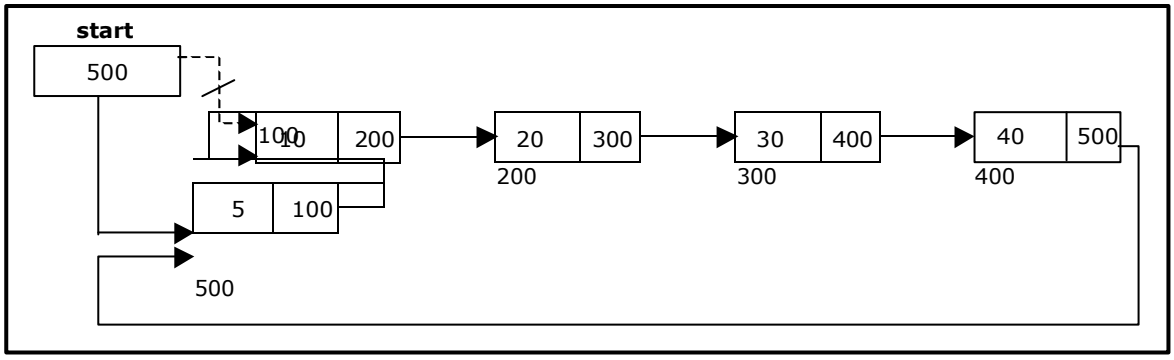


Figure 3.6.2. Inserting a node at the beginning

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 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:

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

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

Figure 3.6.3 shows inserting a node into the circular single linked list at the end.

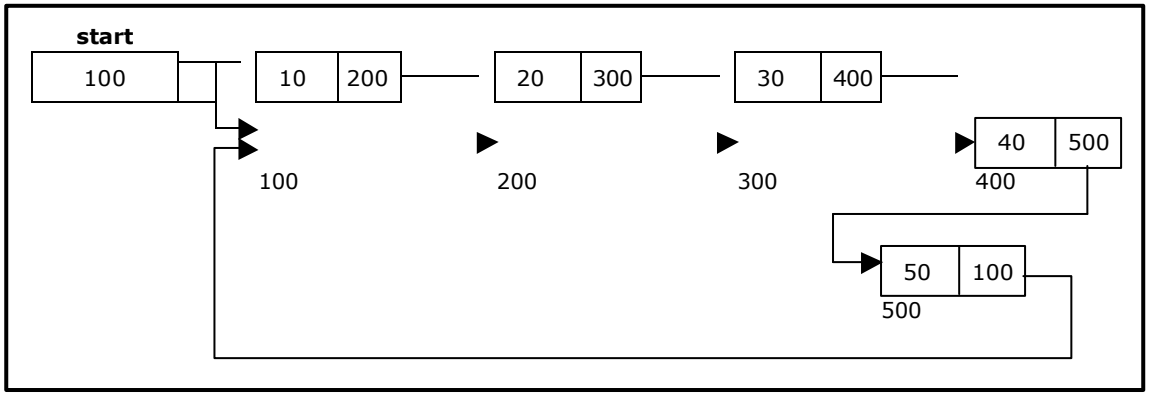


Figure 3.6.3 Inserting a node at the end.

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.*

The function cll_delete_beg(), is used for deleting the first node in the list. Figure 3.6.4 shows deleting a node at the beginning of a circular single linked list.

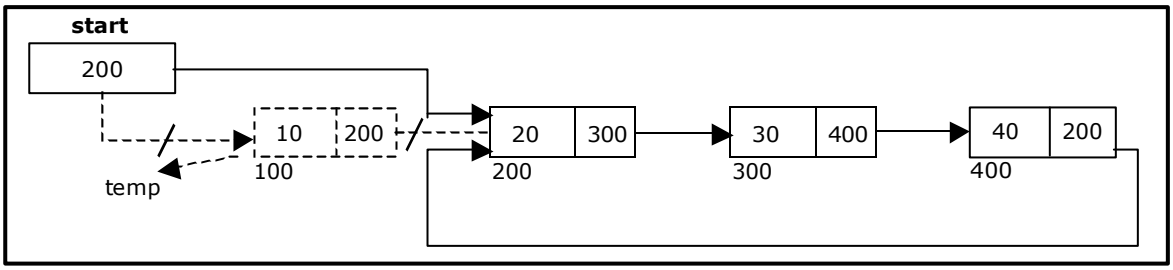


Figure 3.6.4. Deleting a node at beginning.

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.*

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

Figure 3.6.5 shows deleting a node at the end of a circular single linked list.

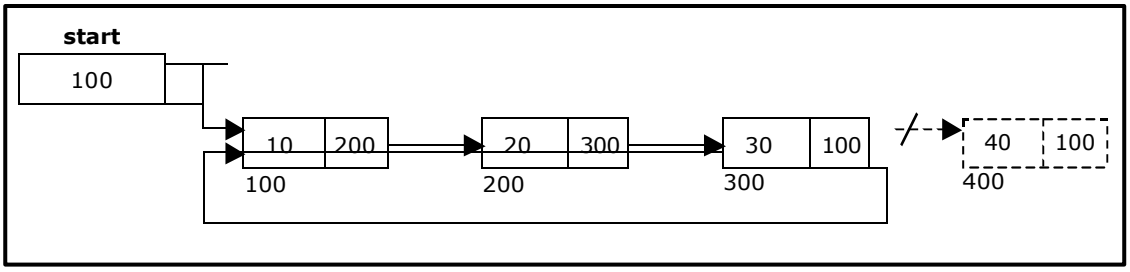


Figure 3.6.5. Deleting a node at the end.

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);
```

3.7.1. Source Code for Circular Single Linked List:

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

struct cslinklist
{
        int data;
        struct cslinklist *next;
};

typedef struct cslinklist node;

node *start = NULL;

int nodectr;

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

```
int menu()
{
        int ch;
        clrscr();
        printf("\n 1. Create a list ");
        printf("\n\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\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\n-------------------------");
        printf("\n 8. Display the list");
        printf("\n 9. Exit"); printf("\n\n-------
        ------------------");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}

void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        nodectr = n;
        for(i = 0; i < n ; i++)
        {
                newnode = getnode();
                if(start == NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> next != NULL)
                                temp = temp -> next;
                        temp -> next = newnode;
                }
        }
        newnode ->next = start;         /* last node is pointing to starting node */
}

void display()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right): ");
        if(start == NULL )
                printf("\n Empty List");
        else
        {
                do
                {
                        printf("\t %d ", temp -> data);
                        temp = temp -> next;
                } while(temp != start);
                printf(" X ");
        }
}
```

```
void cll_insert_beg()
{
        node *newnode, *last;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
                newnode -> next = start;
        }
        else
        {
                last = start;
                while(last -> next != start)
                        last = last -> next;
                newnode -> next = start;
                start = newnode;
                last -> next = start;
        }
        printf("\n Node inserted at beginning..");
        nodectr++;
}

void cll_insert_end()
{
        node *newnode, *temp;
        newnode = getnode();
        if(start == NULL )
        {
                start = newnode;
                newnode -> next = start;
        }
        else
        {
                temp = start;
                while(temp -> next != start)
                        temp = temp -> next;
                temp -> next = newnode;
                newnode -> next = start;
        }
        printf("\n Node inserted at end..");
        nodectr++;
}

void cll_insert_mid()
{
        node *newnode, *temp, *prev;
        int i, pos ;
        newnode = getnode();
        printf("\n Enter the position:  ");
        scanf("%d", &pos);
        if(pos > 1 && pos < nodectr)
        {
                temp = start;
                prev = temp;
                i = 1;
                while(i < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        i++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
```

```
                nodectr++;
                printf("\n Node inserted at middle..");
        }
        else
        {
                printf("position %d of list is not a middle position ", pos);
        }
}

void cll_delete_beg()
{
        node *temp, *last;
        if(start == NULL)
        {
                printf("\n No nodes exist..");
                getch();
                return ;
        }
        else
        {
                last = temp = start;
                while(last -> next != start)
                        last = last -> next;
                start = start -> next;
                last -> next = start;
                free(temp);
                nodectr--;
                printf("\n Node deleted..");
                if(nodectr == 0)
                        start = NULL;
        }
}

void cll_delete_last()
{
        node *temp,*prev;
        if(start == NULL)
        {
                printf("\n No nodes exist..");
                getch();
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != start)
                {
                        prev = temp;
                        temp = temp -> next;
                }
                prev -> next = start;
                free(temp);
                nodectr--;
                if(nodectr == 0)
                        start = NULL;
                printf("\n Node deleted..");
        }
}
```

```
void cll_delete_mid()
{
        int i = 0, pos;
        node *temp, *prev;

        if(start == NULL)
        {
                printf("\n No nodes exist..");
                getch();
                return ;
        }
        else
        {
                printf("\n Which node to delete: ");
                scanf("%d", &pos);
                if(pos > nodectr)
                {
                        printf("\nThis node does not exist");
                        getch();
                        return;
                }
                if(pos > 1 && pos < nodectr)
                {
                        temp=start;
                        prev = start;
                        i = 0;
                        while(i < pos - 1)
                        {
                                prev = temp;
                                temp = temp -> next ;
                                i++;
                        }
                        prev -> next = temp -> next;
                        free(temp);
                        nodectr--;
                        printf("\n Node Deleted..");
                }
                else
                {
                        printf("\n It is not a middle position..");
                        getch();
                }
        }
}

void main(void)
{
        int result;
        int ch, n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch(ch)
                {
                        case 1 :
                                if(start == NULL)
                                {
                                        printf("\n Enter Number of nodes to create: ");
                                        scanf("%d", &n);
                                        createlist(n);
                                        printf("\nList created..");
                                }
```

```
                    else
                        printf("\n List is already Exist..");
                    break;
                case 2 :
                    cll_insert_beg();
                    break;
                case 3 :
                    cll_insert_end();
                    break;
                case 4 :
                    cll_insert_mid();
                    break;
                case 5 :
                    cll_delete_beg();
                    break;
                case 6 :
                    cll_delete_last();
                    break;
                case 7 :
                    cll_delete_mid();
                    break;
                case 8 :
                    display();
                    break;
                case 9 :
                    exit(0);
            }
            getch();
        }
}
```

3.8. Circular Double Linked List:

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the *right* link of the right most node points back to the *start* node and *left* link of the first node points to the last node. A circular double linked list is shown in figure 3.8.1.

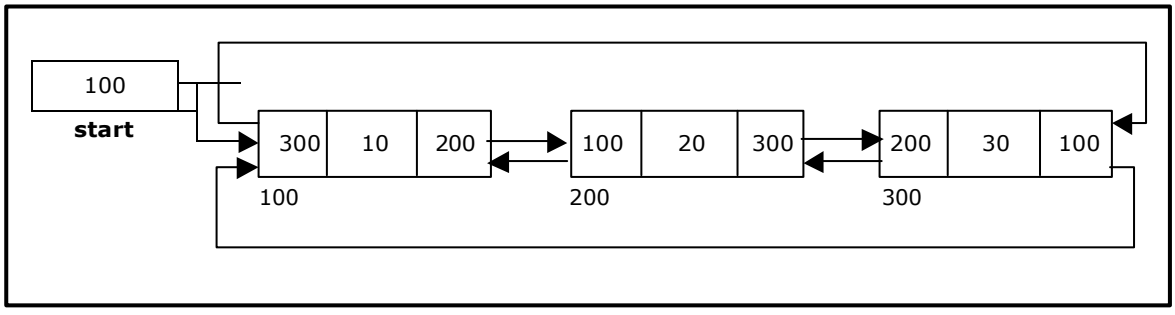


Figure 3.8.1. Circular Double Linked List

The basic operations in a circular double linked list are:

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

Creating a Circular Double 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 getnode().
  newnode = getnode();

- If the list is empty, then do the following
  start = newnode;
  newnode -> left = start;
  newnode ->right = start;

- If the list is not empty, follow the steps given below:
  newnode -> left = start -> left;
  newnode -> right = start;
  start -> left->right = newnode;
  start -> left = newnode;

- Repeat the above steps 'n' times.

The function cdll_createlist(), is used to create 'n' number of nodes:

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 using getnode().
  newnode=getnode();

- If the list is empty, then
  start = newnode;
  newnode -> left = start;
  newnode -> right = start;

- If the list is not empty, follow the steps given below:
  newnode -> left = start -> left;
  newnode -> right = start;
  start -> left -> right = newnode;
  start -> left = newnode;
  start = newnode;

The function cdll_insert_beg(), is used for inserting a node at the beginning. Figure 3.8.2 shows inserting a node into the circular double linked list at the beginning.

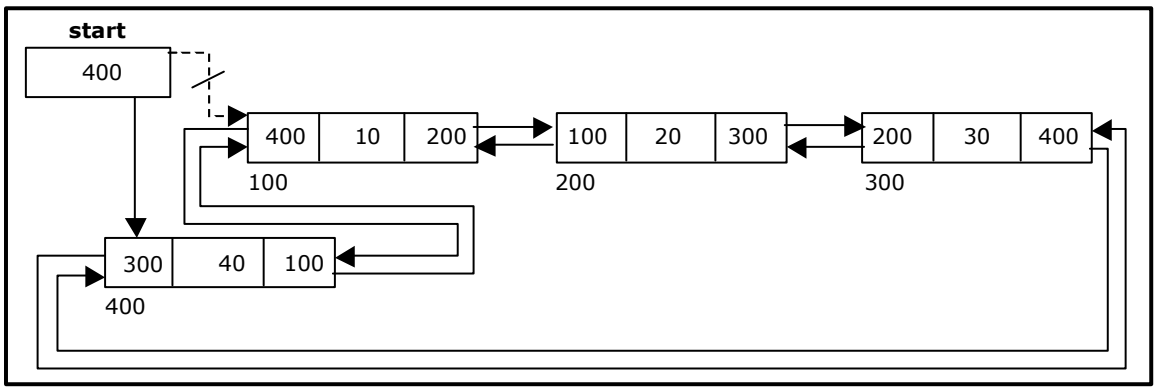


Figure 3.8.2. Inserting a node at the beginning

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 using getnode()
  newnode=getnode();

- If the list is empty, then
  start = newnode;
  newnode -> left = start;
  newnode -> right = start;

- If the list is not empty follow the steps given below:
  newnode -> left = start -> left;
  newnode -> right = start;
  start -> left -> right = newnode;
  start -> left = newnode;

The function cdll_insert_end(), is used for inserting a node at the end. Figure 3.8.3 shows inserting a node into the circular linked list at the end.

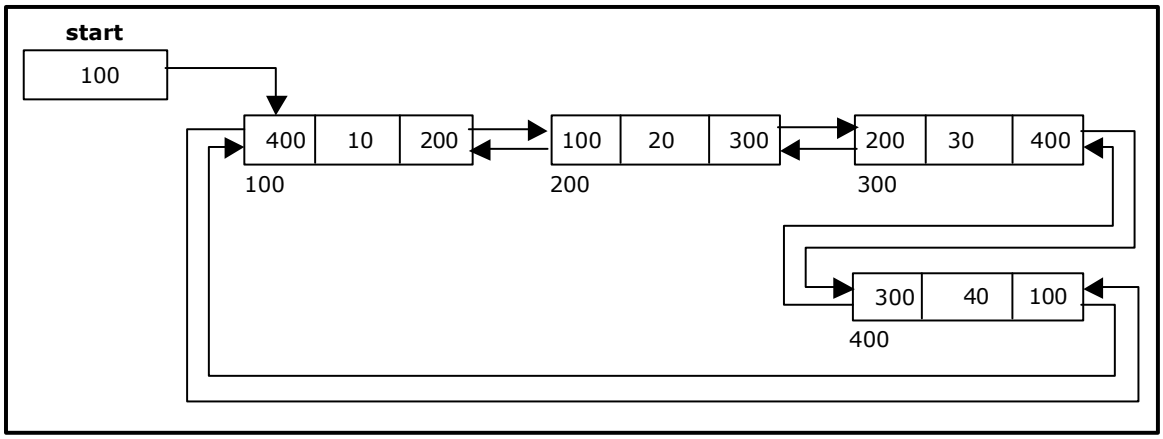


Figure 3.8.3. Inserting a node at the end

Inserting a node at an 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. Then traverse the temp pointer upto the specified position.

- After reaching the specified position, follow the steps given below:
  newnode -> left = temp;
  newnode -> right = temp -> right;
  temp -> right -> left = newnode;
  temp -> right = newnode;
  nodectr++;

The function cdll_insert_mid(), is used for inserting a node in the intermediate position. Figure 3.8.4 shows inserting a node into the circular double linked list at a specified intermediate position other than beginning and end.

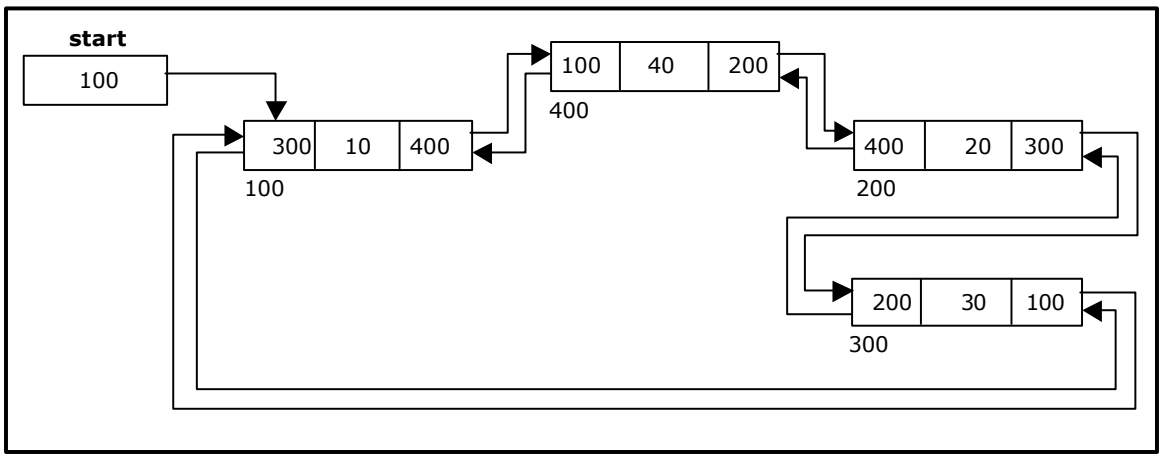


Figure 3.8.4. Inserting a node at an intermediate position

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

The function cdll_delete_beg(), is used for deleting the first node in the list. Figure 3.8.5 shows deleting a node at the beginning of a circular double linked list.

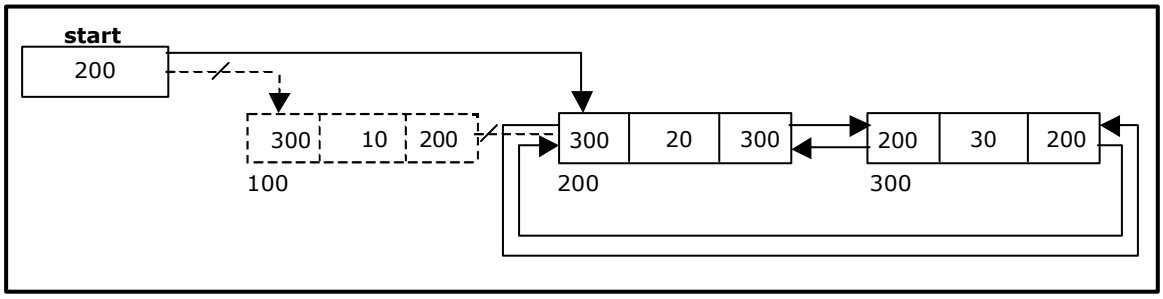


Figure 3.8.5. Deleting a node at beginning

Deleting a node at the end:

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

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below:

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

The function cdll_delete_last(), is used for deleting the last node in the list. Figure 3.8.6 shows deleting a node at the end of a circular double linked list.

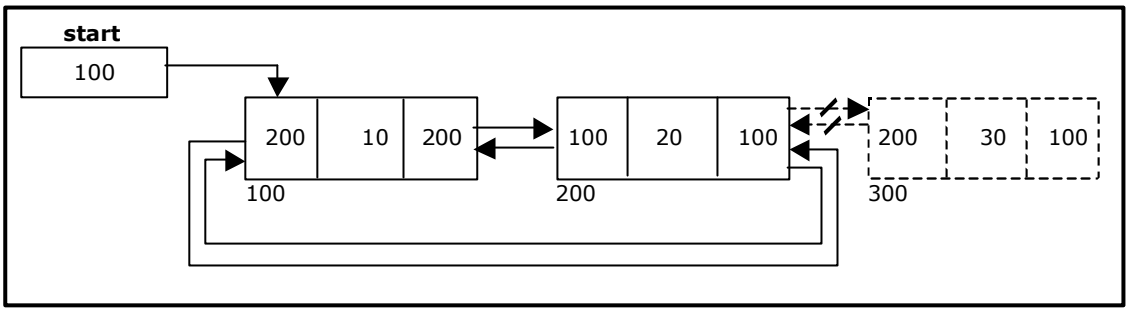


Figure 3.8.6. Deleting a node at the end

Deleting a node at Intermediate position:

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

- 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..");
        nodectr--;
}
```

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

Figure 3.8.7 shows deleting a node at a specified intermediate position other than beginning and end from a circular double linked list.

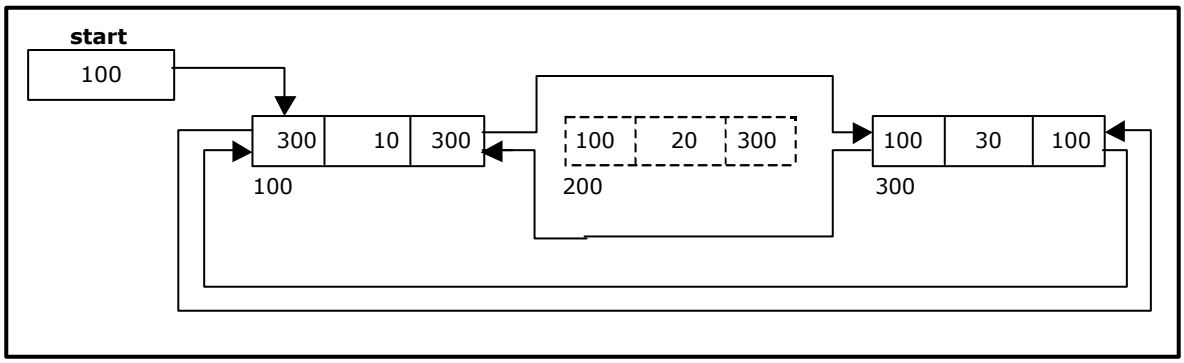


Figure 3.8.7. Deleting a node at an intermediate position

Traversing a circular double 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;
Print  temp -> data;
temp = temp -> right;
while(temp != start)
{
        print  temp -> data;
        temp = temp -> right;
}
```

The function cdll_display_left _right(), is used for traversing from left to right.

Traversing a circular double linked list from right to left:

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

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
do
{
        temp = temp -> left;
        print temp -> data;
} while(temp != start);
```

The function cdll_display_right_left(), is used for traversing from right to left.

3.8.1. Source Code for Circular Double Linked List:

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

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

typedef struct cdlinklist node;
node *start = NULL;
int nodectr;

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

int menu()
{
        int ch;
        clrscr();
        printf("\n 1. Create ");
        printf("\n\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\n-------------------------");
        printf("\n 5. Delete a node from Beginning");
        printf("\n 6. Delete a node from End");
        printf("\n 7. Delete a node from Middle");
        printf("\n\n-------------------------");
        printf("\n 8. Display the list from Left to Right");
        printf("\n 9. Display the list from Right to Left");
        printf("\n 10.Exit");
        printf("\n\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}

void cdll_createlist(int n)
{
        int i;
        node *newnode, *temp;
        if(start == NULL)
        {
                nodectr = n;
                for(i = 0; i < n; i++)
                {
                        newnode = getnode();
                        if(start == NULL)
                        {
                                start = newnode;
                                newnode -> left = start;
                                newnode ->right = start;
                        }
                        else
                        {
                                newnode -> left =  start -> left;
```

```
                                newnode -> right = start;
                                start -> left->right = newnode;
                                start -> left = newnode;
                        }
                }
        }
        else
                printf("\n List already exists..");
}

void cdll_display_left_right()
{
        node *temp;
        temp = start;
        if(start == NULL)
                printf("\n Empty List");
        else
        {
                printf("\n The contents of List: ");
                printf(" %d ", temp -> data);
                temp = temp -> right;
                while(temp != start)
                {
                        printf(" %d ", temp -> data);
                        temp = temp -> right;
                }
        }
}

void cdll_display_right_left()
{
        node *temp;
        temp = start;
        if(start == NULL)
                printf("\n Empty List");
        else
        {
                printf("\n The contents of List: ");
                do
                {
                        temp = temp -> left;
                        printf("\t%d", temp -> data);
                } while(temp != start);
        }
}

void cdll_insert_beg()
{
        node *newnode;
        newnode = getnode();
        nodectr++;
        if(start == NULL)
        {
                start = newnode;
                newnode -> left = start;
                newnode -> right = start;
        }
        else
        {
                newnode -> left = start -> left;
                newnode -> right = start;
                start -> left -> right = newnode;
                start -> left = newnode;
```

```
                start = newnode;
        }
}

void cdll_insert_end()
{
        node *newnode,*temp;
        newnode = getnode();
        nodectr++;
        if(start == NULL)
        {
                start = newnode;
                newnode -> left = start;
                newnode -> right = start;
        }
        else
        {
                newnode -> left = start -> left;
                newnode -> right = start;
                start -> left -> right = newnode;
                start -> left = newnode;
        }
        printf("\n Node Inserted at End");
}

void cdll_insert_mid()
{
        node *newnode, *temp, *prev;
        int pos, ctr = 1;
        newnode = getnode();
        printf("\n Enter the position: ");
        scanf("%d", &pos);
        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;
                nodectr++;
                printf("\n Node Inserted at Middle.. ");
        }
        else
                printf("position %d of list is not a middle position", pos);
        }
}

void cdll_delete_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes exist..");
```

```
                getch();
                return ;
        }
        else
        {
                nodectr--;
                if(nodectr == 0)
                {
                        free(start);
                        start = NULL;
                }
                else
                {
                        temp = start;
                        start = start -> right;
                        temp -> left -> right = start;
                        start -> left = temp -> left;
                        free(temp);
                }
                printf("\n Node deleted at Beginning..");
        }
}

void cdll_delete_last()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes exist..");
                getch();
                return;
        }
        else
        {
                nodectr--;
                if(nodectr == 0)
                {
                        free(start);
                        start = NULL;
                }
                else
                {
                        temp = start;
                        while(temp -> right != start)
                                temp = temp -> right;
                        temp -> left -> right = temp -> right;
                        temp -> right -> left = temp -> left;
                        free(temp);
                }
                printf("\n Node deleted from end ");
        }
}

void cdll_delete_mid()
{
        int ctr = 1, pos;
        node *temp;
        if( start == NULL)
        {
                printf("\n No nodes exist..");
                getch();
                return;
        }
```

```
        else
        {
                printf("\n Which node to delete: ");
                scanf("%d", &pos);
                if(pos > nodectr)
                {
                        printf("\nThis node does not exist");
                        getch();
                        return;
                }
                if(pos > 1 && pos < nodectr)
                {
                        temp = start;
                        while(ctr < pos)
                        {
                                temp = temp -> right ;
                                ctr++;
                        }
                        temp -> right -> left = temp -> left;
                        temp -> left -> right = temp -> right;
                        free(temp);
                        printf("\n node deleted..");
                        nodectr--;
                }
                else
                {
                        printf("\n It is not a middle position..");
                        getch();
                }
        }
}

void main(void)
{
        int ch,n;
        clrscr();
        while(1)
        {
                ch = menu();
                switch( ch)
                {
                        case 1 :
                                printf("\n Enter Number of nodes to create: ");
                                scanf("%d", &n);
                                cdll_createlist(n);
                                printf("\n List created..");
                                break;
                        case 2 :
                                cdll_insert_beg();
                                break;
                        case 3 :
                                cdll_insert_end();
                                break;
                        case 4 :
                                cdll_insert_mid();
                                break;
                        case 5 :
                                cdll_delete_beg();
                                break;
                        case 6 :
                                cdll_delete_last();
                                break;
```

```
                        case 7 :
                                cdll_delete_mid();
                                break;
                        case 8 :
                                cdll_display_left_right();
                                break;
                        case 9 :
                                cdll_display_right_left();
                                break;
                        case 10:
                                exit(0);
                }
                getch();
        }
}
```

3.9. Comparison of Linked List Variations:

The major disadvantage of doubly linked lists (over singly linked lists) is that they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the *prev* fields as well as the *next* fields; the more fields that have to be maintained, the more chance there is for errors.

The major advantage of doubly linked lists is that they make some operations (like the removal of a given node, or a right-to-left traversal of the list) more efficient.

The major advantage of circular lists (over non-circular lists) is that they eliminate some extra-case code for some operations (like deleting last node). Also, some applications lead naturally to circular list representations. For example, a computer network might best be modeled using a circular list.

3.10. Polynomials:

A polynomial is of the form: $\sum_{i=0}^{n} c_i x^i$

Where, c_i is the coefficient of the i^{th} term and
n is the degree of the polynomial

Some examples are:

$5x^2 + 3x + 1$
$12x^3 - 4x$
$5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$

It is not necessary to write terms of the polynomials in decreasing order of degree. In other words the two polynomials 1 + x and x + 1 are equivalent.

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials $5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$ illustrates in figure 3.10.1.

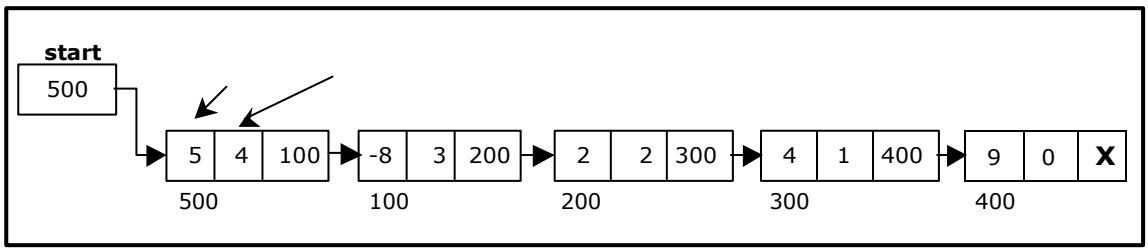


Figure 3.10.1. Single Linked List for the polynomial $F(x) = 5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$

3.10.1. Source code for polynomial creation with help of linked list:

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

struct link
{
        float coef;
        int expo;
        struct link *next;
};

typedef struct link node;
node * getnode()
{
                node *tmp;
                tmp =(node *) malloc( sizeof(node) );
                printf("\n Enter Coefficient : ");
                fflush(stdin);
                scanf("%f",&tmp->coef);
                printf("\n Enter Exponent : ");
                fflush(stdin);
                scanf("%d",&tmp->expo);
                tmp->next = NULL;
                return tmp;
}
node * create_poly (node *p )
{
                char ch;
                node *temp,*newnode;
                while( 1 )
                {
                        printf ("\n Do U Want polynomial node (y/n): ");
                        ch = getche();
                        if(ch == 'n')
                                break;
                        newnode = getnode();
                        if( p == NULL )
                                p = newnode;
                        else
                        {
                                temp = p;
                                while(temp->next != NULL )
                                        temp = temp->next;
                                temp->next = newnode;
                        }

                }
                return p;
}
```

```
void display (node *p)
{
		node *t = p;
		while (t != NULL)
		{
			printf("+ %.2f", t -> coef);
			printf("X^ %d", t -> expo);
			t =t -> next;
		}
}

void main()
{
		node *poly1 = NULL ,*poly2 = NULL,*poly3=NULL;
		clrscr();
		printf("\nEnter First Polynomial..(in ascending-order of exponent)");
		poly1 = create_poly (poly1);
		printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
		poly2 = create_poly (poly2);
		clrscr();
		printf("\n Enter Polynomial 1: ");
		display (poly1);
		printf("\n Enter Polynomial 2: ");
		display (poly2);
		getch();
}
```

3.10.2. Addition of Polynomials:

To add two polynomials we need to scan them once. If we find terms with the same exponent in the two polynomials, then we add the coefficients; otherwise, we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.

To add two polynomials follow the following steps:

- Read two polynomials.
- Add them.
- Display the resultant polynomial.

3.10.3. Source code for polynomial addition with help of linked list:

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

struct link
{
	float coef;
	int expo;
	struct link *next;
};

typedef struct link node;

node * getnode()
{
		node *tmp;
```

```
                tmp =(node *) malloc( sizeof(node) );
                printf("\n Enter Coefficient : ");
                fflush(stdin);
                scanf("%f",&tmp->coef);
                printf("\n Enter Exponent : ");
                fflush(stdin);
                scanf("%d",&tmp->expo);
                tmp->next = NULL;
                return tmp;
}

node * create_poly (node *p )
{
                char ch;
                node *temp,*newnode;
                while( 1 )
                {
                        printf ("\n Do U Want polynomial node (y/n): ");
                        ch = getche();
                        if(ch == 'n')
                                break;
                        newnode = getnode();
                        if( p == NULL )
                                p = newnode;
                        else
                        {
                                temp = p;
                                while(temp->next != NULL )
                                        temp = temp->next;
                                temp->next = newnode;
                        }

                }
                return p;
}

void display (node *p)
{
                node *t = p;
                while (t != NULL)
                {
                        printf("+ %.2f", t -> coef);
                        printf("X^ %d", t -> expo);
                        t = t -> next;
                }
}

void add_poly(node *p1,node *p2)
{

        node *newnode;
        while(1)
        {
                if( p1 == NULL || p2 == NULL )
                        break;
                if(p1->expo == p2->expo )
                {
                                printf("+ %.2f X ^%d",p1->coef+p2->coef,p1->expo);
                                p1 = p1->next; p2 = p2->next;
                }
                else
                {
                        if(p1->expo  < p2->expo)
```

```
                        {
                                printf("+ %.2f X ^%d",p1->coef,p1->expo);
                                p1 = p1->next;
                        }
                        else
                        {
                                printf(" + %.2f X ^%d",p2->coef,p2->expo);
                                p2 = p2->next;
                        }
                }
        }
        while(p1 != NULL )
        {
                printf("+ %.2f X ^%d",p1->coef,p1->expo);
                p1 = p1->next;
        }
        while(p2 != NULL )
        {

                printf("+ %.2f X ^%d",p2->coef,p2->expo);
                p2 = p2->next;
        }
}

void main()
{
                node *poly1 = NULL ,*poly2 = NULL,*poly3=NULL;
                clrscr();
                printf("\nEnter First Polynomial..(in ascending-order of exponent)");
                poly1 = create_poly (poly1);
                printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
                poly2 = create_poly (poly2);
                clrscr();
                printf("\n Enter Polynomial 1: ");
                display (poly1);
                printf("\n Enter Polynomial 2: ");
                display (poly2);
                printf( "\n Resultant Polynomial : ");
                add_poly(poly1, poly2);
                display (poly3);
                getch();
}
```

Exercise

1. Write a "C" functions to split a given list of integers represented by a single linked list into two lists in the following way. Let the list be L = (l_0, l_1, ….., l_n). The resultant lists would be R_1 = (l_0, l_2, l_4, …..) and R_2 = (l_1, l_3, l_5, …..).

2. Write a "C" function to insert a node "t" before a node pointed to by "X" in a single linked list "L".

3. Write a "C" function to delete a node pointed to by "p" from a single linked list "L".

4. Suppose that an ordered list L = (l_0, l_1, …..,l_n) is represented by a single linked list. It is required to append the list L = (l_n, l_0, l_1, ….., l_n) after another ordered list M represented by a single linked list.

5. Implement the following function as a new function for the linked list toolkit.

   Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

   Postcondition: The return value is the number of occurrences of 42 in the data field of a node on the linked list. The list itself is unchanged.

6. Implement the following function as a new function for the linked list toolkit.

   Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

   Postcondition: The return value is true if the list has at least one occurrence of the number 42 in the data part of a node.

7. Implement the following function as a new function for the linked list toolkit.

   Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

   Postcondition: The return value is the sum of all the data components of all the nodes. NOTE: If the list is empty, the function returns 0.

8. Write a "C" function to concatenate two circular linked lists producing another circular linked list.

9. Write "C" functions to compute the following operations on polynomials represented as singly connected linked list of nonzero terms.

   1. Evaluation of a polynomial
   2. Multiplication of two polynomials.

10. Write a "C" function to represent a sparse matrix having "m" rows and "n" columns using linked list.

11. Write a "C" function to print a sparse matrix, each row in one line of output and properly formatted, with zero being printed in place of zero elements.

12. Write "C" functions to:

    1. Add two m X n sparse matrices and
    2. Multiply two m X n sparse matrices.

    Where all sparse matrices are to be represented by linked lists.

13. Consider representing a linked list of integers using arrays. Write a "C" function to delete the i^{th} node from the list.

Multiple Choice Questions

1. Which among the following is a linear data structure: [D]
   A. Queue C. Linked List
   B. Stack D. all the above

2. Which among the following is a dynamic data structure: [A]
   A. Double Linked List C. Stack
   B. Queue D. all the above

3. The link field in a node contains: [A]
   A. address of the next node C. data of next node
   B. data of previous node D. data of current node

4. Memory is allocated dynamically to a data structure during execution by ------- function. [D]
   A. malloc() C. realloc()
   B. Calloc() D. all the above

5. How many null pointer/s exist in a circular double linked list? [D]
   A. 1 C. 3
   B. 2 D. 0

[ ]

6. Suppose that p is a pointer variable that contains the NULL pointer. What happens if your program tries to read or write *p?
   A. A syntax error always occurs at compilation time.
   B. A run-time error always occurs when *p is evaluated.
   C. A run-time error always occurs when the program finishes.
   D. The results are unpredictable.

[A]

7. What kind of list is best to answer questions such as: "What is the item at position n?"
   A. Lists implemented with an array.
   B. Doubly-linked lists.
   C. Singly-linked lists.
   D. Doubly-linked or singly-linked lists are equally best.

8. In a single linked list which operation depends on the length of the list. [A]
   A. Delete the last element of the list
   B. Add an element before the first element of the list
   C. Delete the first element of the list
   D. Interchange the first two elements of the list

9. A double linked list is declared as follows: [A]

```
struct dllist
{
        struct dllist *fwd, *bwd;
        int data;
 }
```

Where fwd and bwd represents forward and backward links to adjacent elements of the list. Which among the following segments of code deletes the element pointed to by X from the double linked list, if it is assumed that X points to neither the first nor last element of the list?

A. X -> bwd -> fwd = X -> fwd;
   X -> fwd -> bwd = X -> bwd
B. X -> bwd -> fwd = X -> bwd;
   X -> fwd -> bwd = X -> fwd
C. X -> bwd -> bwd = X -> fwd;
   X -> fwd -> fwd = X -> bwd
D. X -> bwd -> bwd = X -> bwd;
   X -> fwd -> fwd = X -> fwd

10. Which among the following segment of code deletes the element pointed to by X from the double linked list, if it is assumed that X points to the first element of the list and ***start*** pointer points to beginning of the list? [B]
A. X -> bwd = X -> fwd;
   X -> fwd = X -> bwd
B. start = X -> fwd;
   start -> bwd = NULL;
C. start = X -> fwd;
   X -> fwd = NULL
D. X -> bwd -> bwd = X -> bwd;
   X -> fwd -> fwd = X -> fwd

11. Which among the following segment of code deletes the element pointed to by X from the double linked list, if it is assumed that X points to the last element of the list? [C]
A. X -> fwd -> bwd = NULL;
B. X -> bwd -> fwd = X -> bwd;
C. X -> bwd -> fwd = NULL;
D. X -> fwd -> bwd = X -> bwd;

12. Which among the following segment of code counts the number of elements in the double linked list, if it is assumed that X points to the first element of the list and *ctr* is the variable which counts the number of elements in the list? [A]
A. for (ctr=1; X != NULL; ctr++)
        X = X -> fwd;
B. for (ctr=1; X != NULL; ctr++)
        X = X -> bwd;
C. for (ctr=1; X -> fwd != NULL; ctr++)
        X = X -> fwd;
D. for (ctr=1; X -> bwd != NULL; ctr++)
        X = X -> bwd;

13. Which among the following segment of code counts the number of elements in the double linked list, if it is assumed that X points to the last element of the list and *ctr* is the variable which counts the number of elements in the list? [B]
A. for (ctr=1; X != NULL; ctr++)
        X = X -> fwd;
B. for (ctr=1; X != NULL; ctr++)
        X = X -> bwd;
C. for (ctr=1; X -> fwd != NULL; ctr++)
        X = X -> fwd;
D. for (ctr=1; X -> bwd != NULL; ctr++)
        X = X -> bwd;

14. Which among the following segment of code inserts a new node pointed by X to be inserted at the beginning of the double linked list. The ***start*** pointer points to beginning of the list? [B]

A. X -> bwd = X -> fwd;
   X -> fwd = X -> bwd;
B. X -> fwd = start;
   start -> bwd = X;
   start = X;
C. X -> bwd = X -> fwd;
   X -> fwd = X -> bwd;
   start = X;
D. X -> bwd -> bwd = X -> bwd;
   X -> fwd -> fwd = X -> fwd

15. Which among the following segments of inserts a new node pointed by X to be inserted at the end of the double linked list. The ***start*** and ***last*** pointer points to beginning and end of the list respectively? [C]

A. X -> bwd = X -> fwd;
   X -> fwd = X -> bwd
B. X -> fwd = start;
   start -> bwd = X;
C. last -> fwd = X;
   X -> bwd = last;
D. X -> bwd = X -> bwd;
   X -> fwd = last;

16. Which among the following segments of inserts a new node pointed by X to be inserted at any position (i.e neither first nor last) element of the double linked list? Assume ***temp*** pointer points to the previous position of new node. [D]

A. X -> bwd -> fwd = X -> fwd;
   X -> fwd -> bwd = X -> bwd
B. X -> bwd -> fwd = X -> bwd;
   X -> fwd -> bwd = X -> fwd
C. temp -> fwd = X;
   temp -> bwd = X -> fwd;
   X ->fwd = x
   X ->fwd->bwd = temp
D. X -> bwd = temp;
   X -> fwd = temp -> fwd;
   temp ->fwd = X;
   X -> fwd -> bwd = X;

17. A single linked list is declared as follows: [A]

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

Where next represents links to adjacent elements of the list.

Which among the following segments of code deletes the element pointed to by **X** from the single linked list, if it is assumed that X points to neither the first nor last element of the list? **prev** pointer points to previous element.

A. prev -> next = X -> next;
   free(X);
B. X -> next = prev-> next;
   free(X);
C. prev -> next = X -> next;
   free(prev);
D. X -> next = prev -> next;
   free(prev);

18. Which among the following segment of code deletes the element pointed to by X from the single linked list, if it is assumed that X points to the first element of the list and ***start*** pointer points to beginning of the list? [B]

A. X = start -> next;
   free(X);
B. start = X -> next;
   free(X);
C. start = start -> next;
   free(start);
D. X = X -> next;
   start = X;
   free(start);

19. Which among the following segment of code deletes the element pointed to by X from the single linked list, if it is assumed that X points to the last element of the list and **prev** pointer points to last but one element? [C]

A. prev -> next = NULL;
   free(prev);
B. X -> next = NULL;
   free(X);
C. prev -> next = NULL;
   free(X);
D X -> next = prev;
   free(prev);

20. Which among the following segment of code counts the number of elements in the single linked list, if it is assumed that X points to the first element of the list and *ctr* is the variable which counts the number of elements in the list? [A]

A. for (ctr=1; X != NULL; ctr++)
        X = X -> next;
B. for (ctr=1; X != NULL; ctr--)
        X = X -> next;
C. for (ctr=1; X -> next != NULL; ctr++)
        X = X -> next;
D. for (ctr=1; X -> next != NULL; ctr--)
        X = X -> next;

21. Which among the following segment of code inserts a new node pointed by X to be inserted at the beginning of the single linked list. The ***start*** pointer points to beginning of the list? [B]

A. start -> next = X;
   X = start;
B. X -> next = start;
   start = X
C. X -> next = start -> next;
   start = X
D. X -> next = start;
   start = X -> next

22. Which among the following segments of inserts a new node pointed by X to be inserted at the end of the single linked list. The ***start*** and ***last*** pointer points to beginning and end of the list respectively? [C]

A. last -> next = X;
   X -> next = start;
B. X -> next = last;
   last ->next = NULL;
C. last -> next = X;
   X -> next = NULL;
D. last -> next = X -> next;
   X -> next = NULL;

23. Which among the following segments of inserts a new node pointed by X to be inserted at any position (i.e neither first nor last) element of the single linked list? Assume ***prev*** pointer points to the previous position of new node. [D]

A. X -> next = prev -> next;
    prev -> next = X -> next;
B. X = prev -> next;
    prev -> next = X -> next;
C. X -> next = prev;
    prev -> next = X;
D. X -> next = prev -> next;
   prev -> next = X;

24. A circular double linked list is declared as follows: [A]

```
struct cdllist
{
        struct cdllist *fwd, *bwd;
        int data;
 }
```

Where fwd and bwd represents forward and backward links to adjacent elements of the list.

Which among the following segments of code deletes the element pointed to by X from the circular double linked list, if it is assumed that X points to neither the first nor last element of the list?

A. X -> bwd -> fwd = X -> fwd;
   X -> fwd -> bwd = X -> bwd;
B. X -> bwd -> fwd = X -> bwd;
   X -> fwd -> bwd = X -> fwd;
C. X -> bwd -> bwd = X -> fwd;
   X -> fwd -> fwd = X -> bwd;
D. X -> bwd -> bwd = X -> bwd;
   X -> fwd -> fwd = X -> fwd;

25. Which among the following segment of code deletes the element pointed to by X from the circular double linked list, if it is assumed that X points to the first element of the list and ***start*** pointer points to beginning of the list? [D]

A. start = start -> bwd;
   X -> bwd -> bwd = start;
   start -> bwd = X -> bwd;
B. start = start -> fwd;
   X -> fwd -> fwd = start;
   start -> bwd = X -> fwd
C. start = start -> bwd;
   X -> bwd -> fwd = X;
   start -> bwd = X -> bwd
D. start = start -> fwd;
   X -> bwd -> fwd = start;
   start -> bwd = X -> bwd;

26. Which among the following segment of code deletes the element pointed to by X from the circular double linked list, if it is assumed that X points to the last element of the list and ***start*** pointer points to beginning of the list? [B]

A. X -> bwd -> fwd = X -> fwd;
   X -> fwd -> fwd= X -> bwd;
B. X -> bwd -> fwd = X -> fwd;
   X -> fwd -> bwd = X -> bwd;
C. X -> fwd -> fwd = X -> bwd;
   X -> fwd -> bwd= X -> fwd;
D. X -> bwd -> bwd = X -> fwd;
   X -> bwd -> bwd = X -> bwd;

27. Which among the following segment of code counts the number of elements in the circular double linked list, if it is assumed that X and **start** points to the first element of the list and *ctr* is the variable which counts the number of elements in the list? [A]

A. for (ctr=1; X->fwd != start; ctr++)
        X = X -> fwd;

B. for (ctr=1; X != NULL; ctr++)
        X = X -> bwd;

C. for (ctr=1; X -> fwd != NULL; ctr++)
        X = X -> fwd;

D. for (ctr=1; X -> bwd != NULL; ctr++)
        X = X -> bwd;

28. Which among the following segment of code inserts a new node pointed by X to be inserted at the beginning of the circular double linked list. The ***start*** pointer points to beginning of the list? [B]

A.
```
X -> bwd = start;
X -> fwd = start -> fwd;
start -> bwd-> fwd = X;
start -> bwd = X;
start = X
```

B.
```
X -> bwd = start -> bwd;
X -> fwd = start;
start -> bwd-> fwd = X;
start -> bwd = X;
start = X
```

C.
```
X -> fwd = start -> bwd;
X -> bwd = start;
start -> bwd-> fwd = X;
start -> bwd = X;
start = X
```

D.
```
X -> bwd = start -> bwd;
X -> fwd = start;
start -> fwd-> fwd = X;
start -> fwd = X;
X = start;
```

29. Which among the following segment of code inserts a new node pointed by X to be inserted at the end of the circular double linked list. The ***start*** pointer points to beginning of the list? [C]

A.
```
X -> bwd = start;
X -> fwd = start -> fwd;
start -> bwd -> fwd = X;
start -> bwd = X;
start = X
```

B.
```
X -> bwd = start -> bwd;
X -> fwd = start;
start -> bwd -> fwd = X;
start -> bwd = X;
start = X
```

C.
```
X -> bwd= start -> bwd;
X-> fwd = start;
start -> bwd -> fwd = X;
start -> bwd = X;
```

D.
```
X -> bwd = start -> bwd;
X -> fwd = start;
start -> fwd-> fwd = X;
start -> fwd = X;
X = start;
```

30. Which among the following segments of inserts a new node pointed by X to be inserted at any position (i.e neither first nor last) element of the circular double linked list? Assume ***temp*** pointer points to the previous position of new node. [D]

A.
```
X -> bwd -> fwd = X -> fwd;
X -> fwd -> bwd = X -> bwd;
```

B.
```
X -> bwd -> fwd = X -> bwd;
X -> fwd -> bwd = X -> fwd;
```

C.
```
temp ->  fwd = X;
temp -> bwd  = X -> fwd;
X -> fwd = X;
X -> fwd -> bwd = temp;
```

D.
```
X ->  bwd = temp;
X -> fwd  = temp -> fwd;
temp -> fwd = X;
X -> fwd -> bwd = X;
```