Data Structures

Overview

- Introduction to C
- Structure of a C program
- Keywords and Identifiers
- Data Types in C
- Constants and Variables
- Header files
- Input/Output Statement in C
- Operators in C
- Type Conversion And Type Casting
- Decision Control Statements
- Iterative Statements

Introduction to C

- C is a programming language developed in the early 1970s by Dennis Ritchie at Bell Laboratories.
- C is highly portable. i.e., software written for one computer can be run on another computer.
- An important feature of 'C' is its ability to extend itself.

Structure of a C Program

- C program contains one or more functions, where a function is defined as a group of C statements that are executed together.
- The statements are written in a logical sequence to perform a specific task.
- The main( ) function is the most important function and the execution of a C program begins from here.

```
main( )
  {
      Stmt 1;
      Stmt 2;
      ………
      ………
      Stmt N;
   }
Function1( )
   {
      Stmt 1;
        Stmt 2;
       ………
       ………
      Stmt N;
   }
Function2( )
    {
       Stmt 1;
       Stmt 2;
        ………
        ………
       Stmt N;
    }
```

- C program can have any number of functions depending on the tasks that have to be performed and each function can have any number of statements arranged according to specific meaningful sequence.

- Programmers can choose any name for the functions.

- But with an exception that every program must contain one function that has its name as main( ).

Keywords and Identifiers

- Every word in a C program is either a keyword or an identifier.
- Set of reserved words known as Keywords that cannot be used as an identifier.
- All keywords are basically a sequence of characters that have a fixed meaning.
- Keywords must be written in lower case letters.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

- Identifiers are distinct names given to program elements such as constants, variables, etc.,
- Identifiers may consist of an alphabet, digit or an underscore.
- Rules for forming identifier names:

  - Identifier name must begin with either a letter or an underscore.
  - No commas or blanks are allowed within a variable name.
  - The upper case and lower case letter are treated as distinct.
    i.e., Identifiers are case-sensitive.
  - An identifier can be of any length.
  - No special symbols can be used in a variable name.

Ex: roll_number, marks, name, emp_number, basic_pay

- Example of invalid identifiers:

  23_student, @name, %marks, #emp_number, auto

Data types in c

Data Type	Size in bytes	Use
char	1	To store Characters
int	4	To store integer numbers
float	4	To store floating point numbers
double	8	To store big floating point numbers

Constants and Variables

- Constants are identifiers whose value does not change.
- It is used to define fixed values like PI or the charge on an electron so that their value does not get changed.
- Declaring constants: `const float pi = 3.14`
- A Variable is defined as a meaningful name given to the data storage location in the computer memory.
- When using a variable, we actually refer to the address of the memory where the data is stored.
- Variables can change their value at any time.

$$2X + 3Y = 10$$

Here X and Y are variables. Whereas 2,3 and 10 are constants.

Header File

- A header file is a file that **allows programmers to separate certain elements of a program's source code into reusable files**.

- Header files commonly contain forward declarations of classes, subroutines, variables, and other identifiers.

- string.h : for string handling functions
- stdlib.h : for some library functions
- stdio.h : for standardized input and output functions
- math.h : for mathematical functions
- alloc.h : for dynamic memory allocation
- conio.h : for clearing the screen

- All the header files are referenced at the start of the source code file that uses one or more functions from that file.

Input / Output Statement

- The most fundamental operation in a C program is to accept input values to the program from standard input device and output the data produced by the program to a standard output device.

- The scanf function that reads data from the keyboard.

- Similarly, for outputting results of the program, printf function is used that sends results out to a terminal.

- Like printf and scanf, there are different functions in C that can carry out the input-output operations.

- These functions are collectively known as standard Input/Output Library.

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

Operators in C

- An operator is a symbol that tell the computer to perform certain mathematical or logical manipulations.
- Operators are used in program to manipulate data and variables. The data items that operators act upon are called operands.
- Some operators require two operands, while others act upon only one operand.
- The operators are classified into unary, binary and ternary depending on whether they operate on one, two or three operands respectively.

- Arithmetic operators
- Equality operators
- Unary operators
- Bitwise operators
- Comma operators
- Relational operators
- Logical operators
- Conditional operators
- Assignment operators
- Sizeof operator

Arithmetic Operators

Operation	Operator
Multiply	*
Divide	/
Addition	+
Subtraction	-
Modulus	%

Relational Operator

- A relational operator, also known as a comparison operator, is an operator that compares two values.

Operator	Meaning	Example
<	Less than	3<5 gives 1
>	Greater than	7>9 gives 0
>=	Greater than or equal to	100>=100 gives 1
<=	Less than or equal to	50<=100 gives 1

Equality Operators

- C language supports two kinds of equality operators to compare their operands for strict equality or inequality.
- They are (==) and (!=) operator.
- These operators have lower precedence than the relational operators.

Operator	Meaning
==	Returns 1 if both operands are equal, 0 otherwise
!=	Returns 1 if operands do not have the same value, 0 otherwise

Logical Operators

- Logical AND (&&)
- Logical OR (||)
- Logical NOT (!)

Unary Operators

- Unary operators act on single operands. They are

    - Unary minus(-)

    - Increment (++)

    - Decrement operators (--)

Conditional Operator

- It is also known as the ternary operator (?:) is just like if …else statement that can be within expressions.
- Syntax: exp1 ? exp2 : exp3

Bitwise Operators

- Bitwise operators are those operators that perform operations at the bit level.

  - bitwise AND (&)
  - bitwise OR (|)
  - bitwise XOR (^)
  - shift operator (<<),(>>)

Assignment Operators

- It is responsible for assigning values to the variables.
- The equal sign (=) is the fundamental assignment operator.
- Other assignment operators are,

  /=, \=, *=, += , -=, &=, ^=, <<=, >>=

Sizeof Operator

- It is a unary operator used to calculate the size of data types.
- It can be applied to all data types.
- The operator returns the size of the variable, data type, or expression in bytes.
- The sizeof operator is used to determine the amount of memory space that the variable/expression/data type will take.

```
int a = 10;
unsigned int result;
result = sizeof(a);
```

Type Conversion and Type Casting

Type Conversion:

- It is done when the expression has variables of different data types.

```
float x;
int y=2;
x=y;
```

Now, x=2.0, here integer value is converted

into its equivalent floating point representation.

Type Casting

- It is known as *forced conversion*.

- It is done when the value of a higher data type has to be converted into the value of a lower data type.

```
float salary=10000.00;
int sal;
sal = (int)salary;
```

- When floating point numbers are converted to integer, the digits after the decimal are truncated.

Decision Control Statements

- C program is executed sequentially from the first line of the program to its last line.
- That is, the second statement is executed after the first, the third statement is executed after the second, and so on.
- But in some cases, we want only selected statements to be executed. Such type of conditional processing extends the usefulness of programs.
- It allows the programmers to build logic that determine which statements of the code should be executed and which should be ignored.

- Decision control statements can alter the flow of a sequence of instructions.
- These statements help to jump from one part of the program to another depending on whether a particular condition is satisfied or not.
- It include:

  - if statement
  - if-else statement
  - if-else-if statement
  - switch-case statement

Iterative Statements

- C supports three types of iterative statements also known as looping statements. They are,

  - While loop
  - Do-while loop
  - For loop

- Iterative statements are used to repeat the execution of a sequence of statements, depending on the value of an integer expression.

UNIT - 1

LINEAR STRUCTURES AND TREES

Overview

- Introduction
- Types
- Linked Lists
- Stack ADT
- Queue ADT
- Circular queue implementation
- Applications of stack and queue
- Tree ADT
- Tree Traversals
- Binary Tree ADT
- Expression Trees

Data Structure

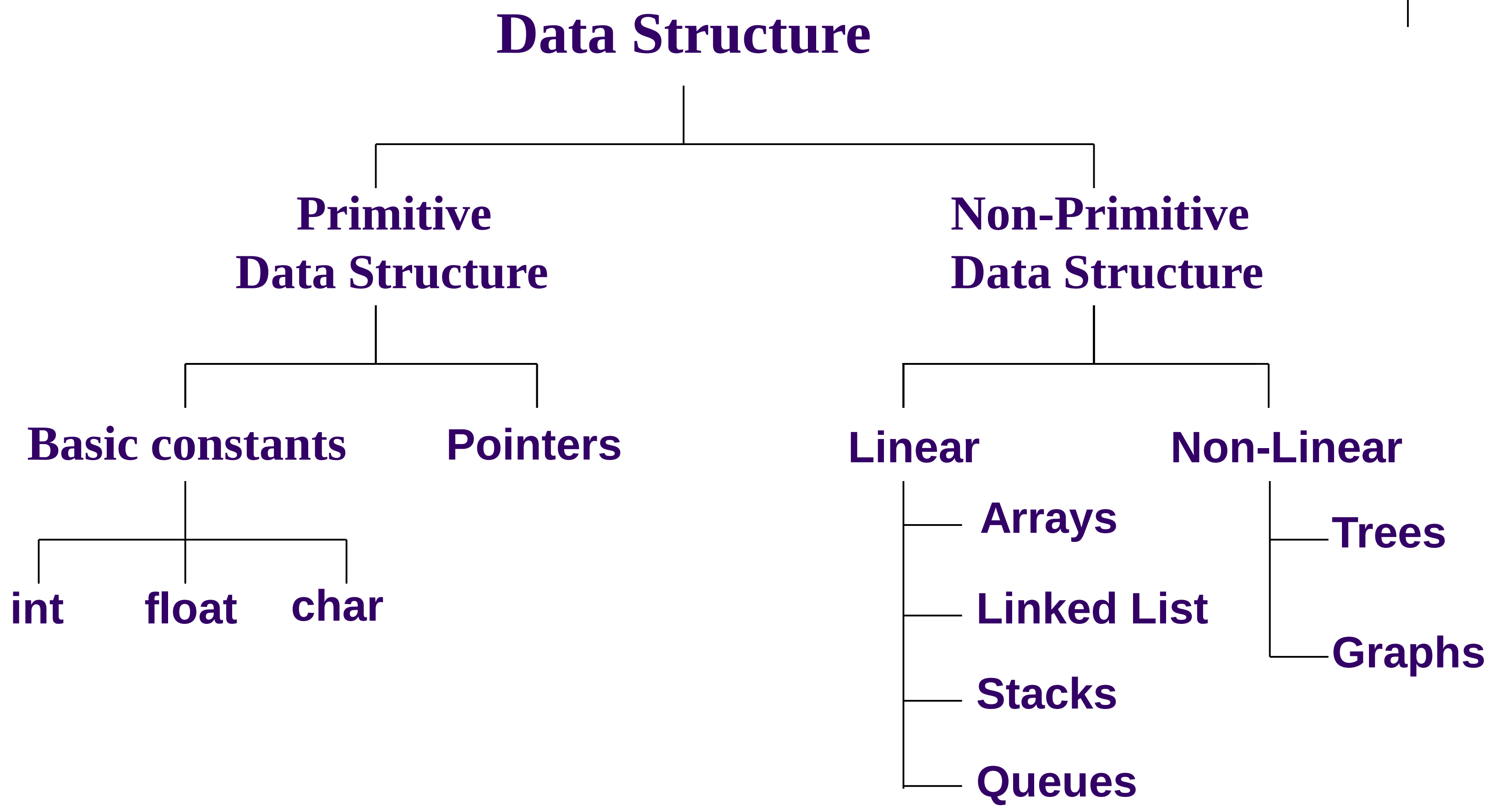
- A data structure defines a way of organizing all data items that consider not only the elements stored but also stores the relationship between elements.

Algorithm + Data structure = Program

- Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory.

- Some data structures are a programming language built-in component, and others may require the inclusion of a library or module before the structure can be used.

Types of Data Structures



- **Primitive** : Primitive data structure is a data structure that can hold a single value in a specific location.

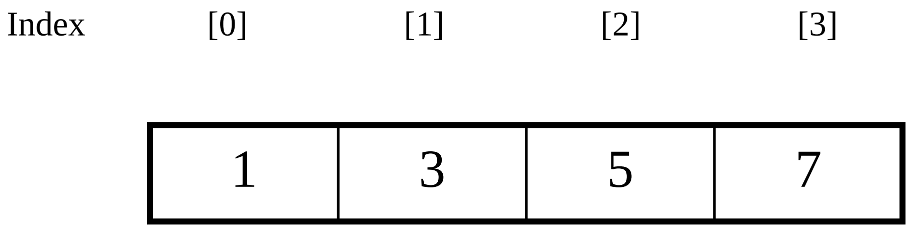
- **Non-Primitive** : It can hold multiple values either in a contiguous location or random locations.

- **Linear** : They have data elements arranged in sequential manner and each member element is connected to its previous and next element.

- **Non-Linear** : It is a form of data structure where the data elements are arranged in a non sequential / random manner supporting multi level storage.

Arrays

- It is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations.
- It works on an index system starting from 0 to (n-1), where 'n' is the size of the array.
- **Types of Arrays** :
  - **One dimensional** – It is the simplest form of array in which the elements are stored linearly and can be accessed individually by specifying the index value of each element stored in the array.



Arrays Contd…

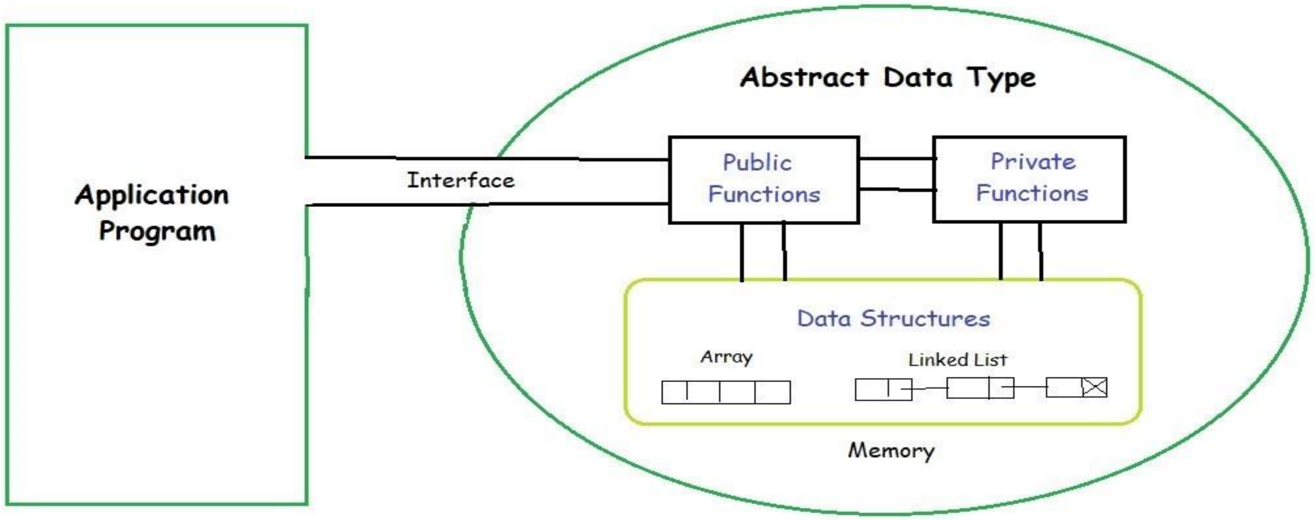
- **Multi dimensional** : It is an array with more than one level of dimension (Array of Arrays) where the data are stored in tabular form.

	C0	C1	C2
R0	a[0][0]	a[0][1]	a[0][2]
R1	a[1][0]	a[1][1]	a[1][2]
R2	a[2][0]	a[2][1]	a[2][2]

Abstract Data Types

- An ADT is **a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations**.

- An ADT specifies what each operation does, but not how it does it. Typically, an ADT can be implemented using one of many different data structures.



Lists

- List is a collection of elements in sequential order.

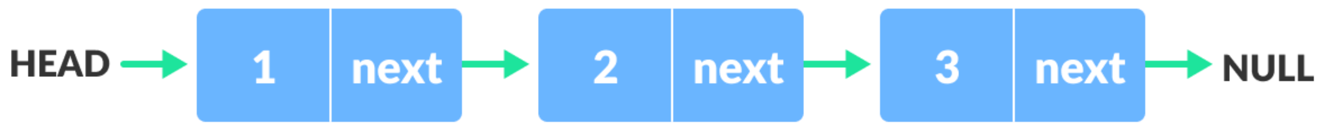
- We can store elements in the memory locations in two ways.
  - We can store the elements in sequential memory locations. This is known as Arrays.
  - We can use pointers or links to associate the elements sequentially. This is known as Linked List.

Diagrammatic representation of Lists:

Array Implementation

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
10	**20**	**30**	**40**	**50**	**60**

Linked List Representation



Operations on Lists:

- Insertion
- Deletion

Ways of Implementation:

- Static Implementation (Array)
- Dynamic Implementation (Linked List)
- Cursor – Based Implementation of Lists

Static Implementation (Array)

- An array is a linear data structure which is a collection of data items having same similar data types stored in contiguous memory locations or adjacent memory locations.
- Thus array has to be finite in nature. i.e., the size of the array should be predefined.
- If the array size is n, then the array index usually starts from 0 to n-1.
- The Syntax of array declaration is,

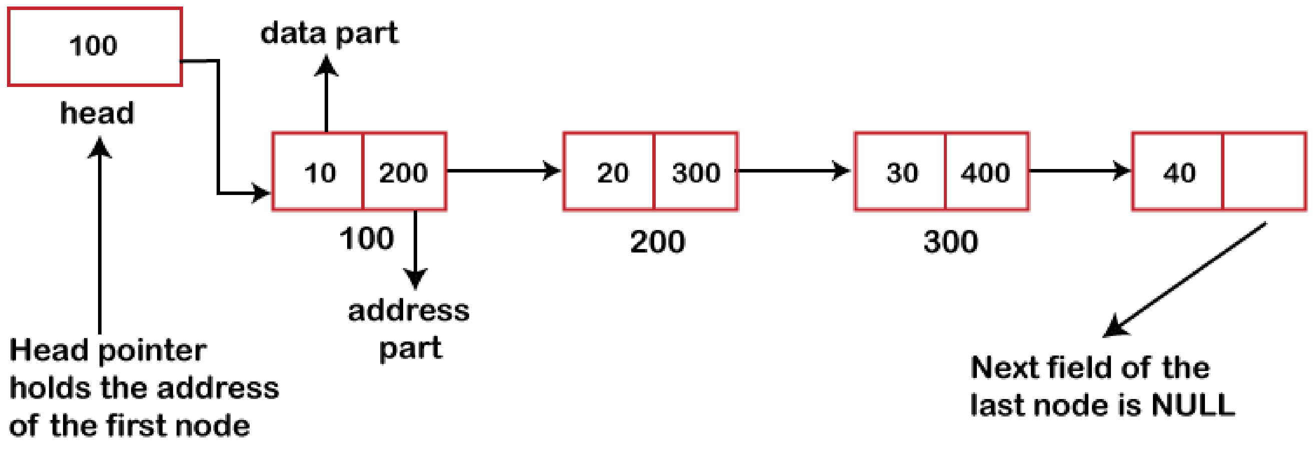
Data type array-name [size of array];

Ex: int a[6];

a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**	**a[5]**
10	**20**	**30**	**40**	**50**	**60**

Linked Lists

- Like arrays, Linked List is a linear data structure.
- Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.
- They includes a series of connected nodes. Here, each node stores the data and the address of the next node.


100
head
data part
10
200
20
300
30
400
40
100
200
300
address part
Head pointer holds the address of the first node
Next field of the last node is NULL

Why Linked List?

- Arrays can be used to store linear data of similar types, but arrays have the following limitations.

- **The size of the arrays is fixed**: Even if the array is dynamically allocated, an estimate of the maximum size of the list is required which considerably wastes the memory space.
- **Insertion of a new element / Deletion of an existing element in the array is expensive as it requires more data movement:** The room has to be created for the new elements and the existing elements have to be shifted. But in Linked list, if we have the head node then we can traverse to any node through it and insert new node at the required position.

For example,

- In a system, if we maintain a sorted list of IDs in an array id[].
- id[] = [1000, 1010, 1050, 2000, 2040].

  And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

- Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved due to this ,so much work is being done which affects the efficiency of the code.

Advantages over arrays:

- Dynamic Array.
- Ease of Insertion/Deletion.

Drawbacks:

- Random access is not allowed. We have to access elements sequentially starting from the first node (head node). So we cannot do binary search with linked lists efficiently with its default implementation.

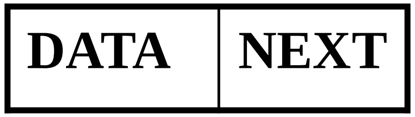
- Extra memory space for a pointer is required with each element of the list.

- Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation of Linked List

- A linked list is represented by a pointer to the first node of the linked list. The first node is called the **head**. If the linked list is empty, then the value of the head points to **NULL**.
- Each node in a list consists of at least **two parts**:
- A **Data** Item (we can store integer, strings or any type of data).
- **Next or Pointer (Or Reference)** to the next node (connects one node to another) or it holds the address of the next node.

Node Structure:



Let us create a simple linked list with 3 nodes.

Step 1: // **A linked list node**

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

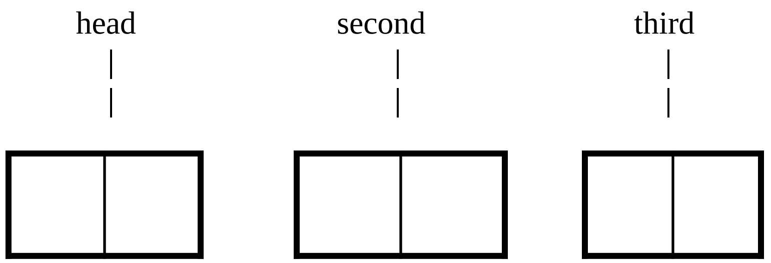
Step 2: // **Create a simple linked list with 3 nodes**

```
struct Node* head = NULL;
struct Node* second = NULL;
struct Node* third = NULL;
```

Step 3: // Allocate 3 nodes in the heap

```
head = (struct Node*)malloc(sizeof(struct Node));
second = (struct Node*)malloc(sizeof(struct Node));
third = (struct Node*)malloc(sizeof(struct Node));
```

/* Three blocks have been allocated dynamically.
We have pointers to these three blocks as head, second and third represents any random value.

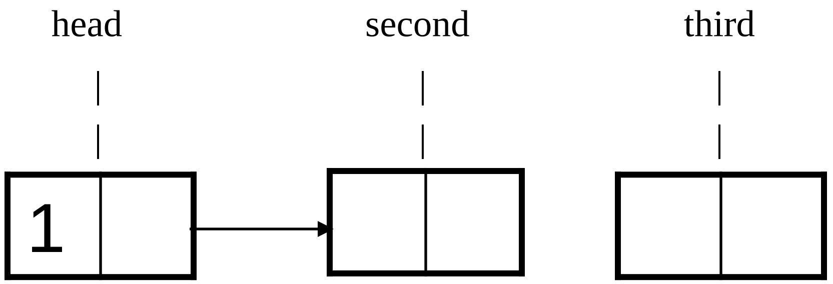


Data is random because we haven’t assigned anything yet */

Step 4:

```
head-->data = 1;          // Assign data in first node
head-->next = second;     // Link first node with the second node
```

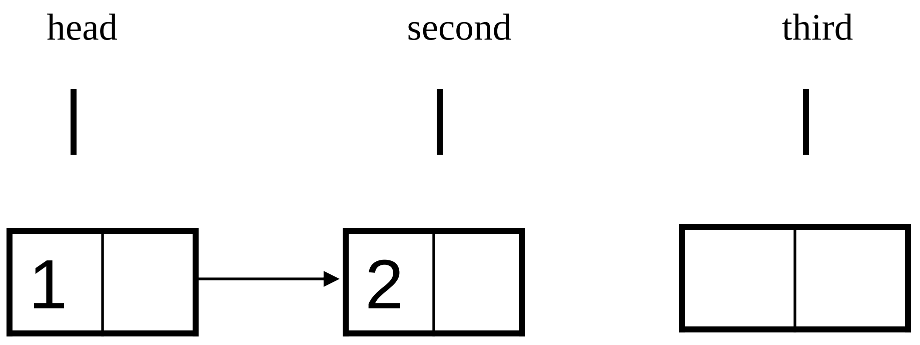
/* data has been assigned to the data part of the first block (block pointed by the head and next pointer of first block points to second. So they both are linked */



Step 5:

```
second-->data =  2            // Assign data to second node
second-->next = third;        // Link second node with the third node
```

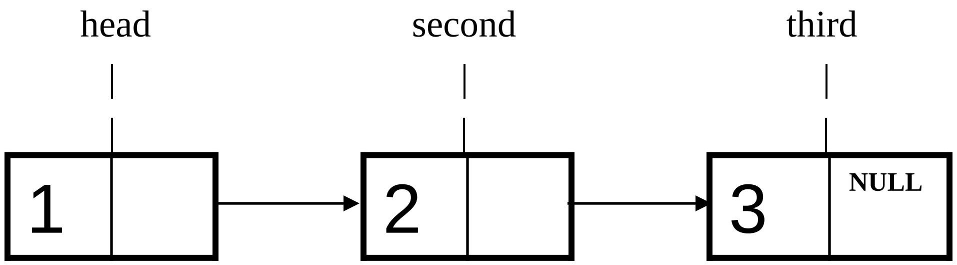
/* data has been assigned to the data part of the second block (block pointed by second) and next pointer of the second block points to the third block. So all three blocks are linked.



Step 6:

```
third-->data = 3;          // Assign data to third node
third-->next = NULL;
```

/* data has been assigned to data part of third block (block pointed by third). And next pointer of the third block is made NULL to indicate that the linked list is terminated here.



Linked List Traversal

Let us traverse the created list and print the data of each node.

```
// This function prints contents the of linked list starting from the given node

void PrintList (struct Node* ptr)
{
    while (ptr != NULL)
    {
        printf(" %d ", ptr-->data);
        ptr = ptr-->next;
     }
  }
```

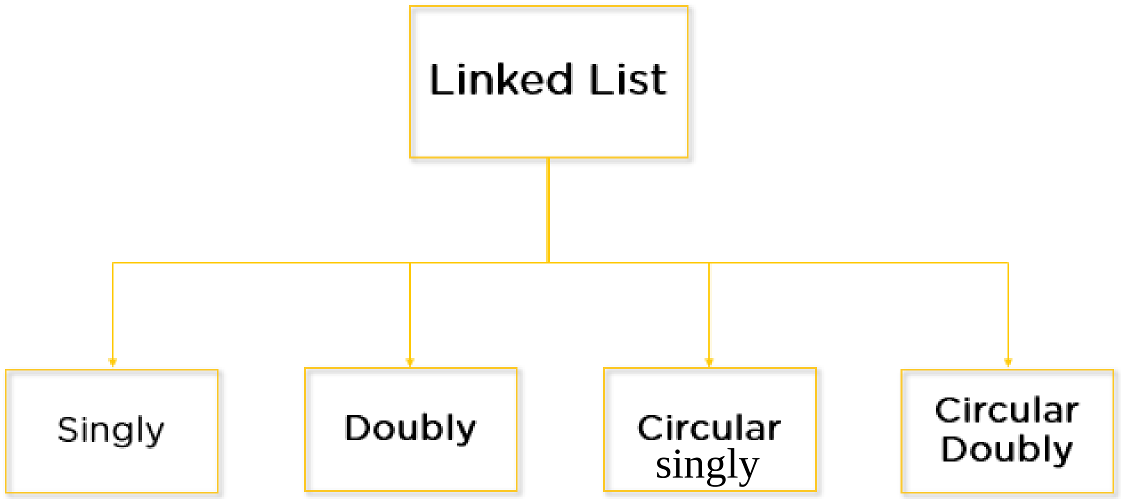
```
        // Allocate 3 nodes in the heap

head = (struct Node*)malloc(sizeof(struct Node));
second = (struct Node*)malloc(sizeof(struct Node));
third = (struct Node*)malloc(sizeof(struct Node));

head-->data = 1;              // Assign data in first node
head-->next = second;         // Link first node with second

second-->data = 2;            // Assign data to second node
second-->next = third;        // Link second node with third

third-->data = 3;             // Assign data to third node
third-->next = NULL;

PrintList(head);
```

Linked List

- A linked list is **a sequence of data structures, which are connected together via links**. Linked List consists of series of nodes. Each node contains the element and a pointer to its successor node.

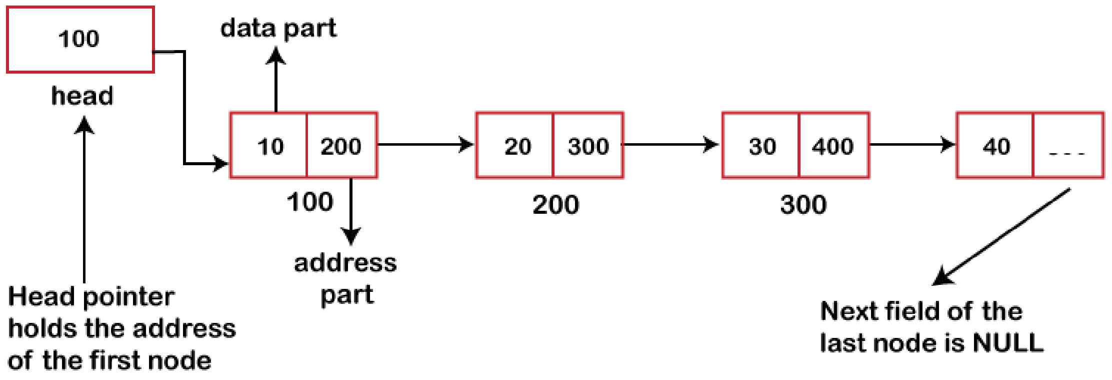
- Each link contains a connection to another link. Linked list is the second most-used data structure after array.

Types of Linked Lists:



Singly Linked List

- The singly linked list is **a linear data structure in which each element of the list contains a pointer which points to the next element in the list**.
- Each element in the singly linked list is called a node.
- Each node has two components: data and a pointer which points to the next node in the list.


100
head
data part
10
200
20
300
30
400
40
- - -
100
200
300
address part
Head pointer holds the address of the first node
Next field of the last node is NULL

Node Declaration:

```
Struct Node
{
  int element;
  Struct Node *Next;
 }
```

Insertion:

We can perform the insertion operation at any place in the linked list.

- At the beginning
- After the given node
- At the end

At the beginning:

Algorithm:

Step 1: Create a node structure for the new node.

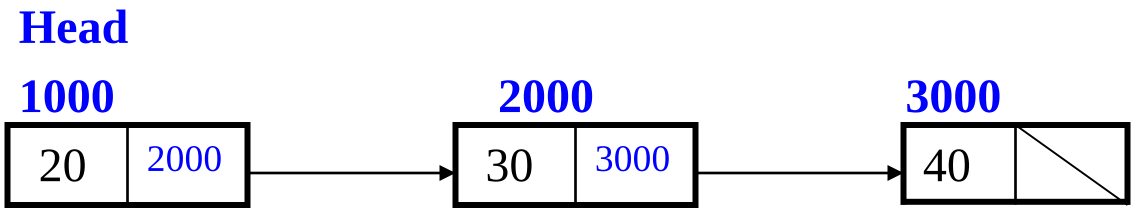
Step 2: Allocate a memory location for the new node.

Step 3: Get the data from the user and Put the data on the data field of a node.

Step 4: In the address field of new node, assign the address of Head node.

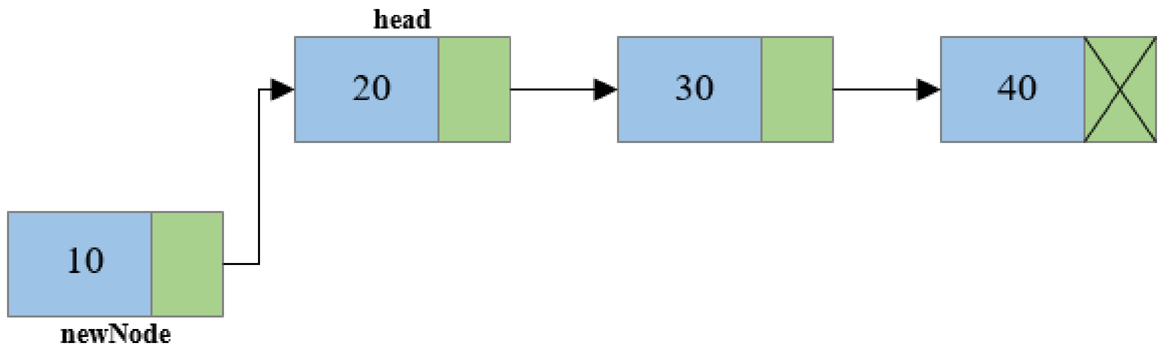
Step 5: Now, make the new node as head node.

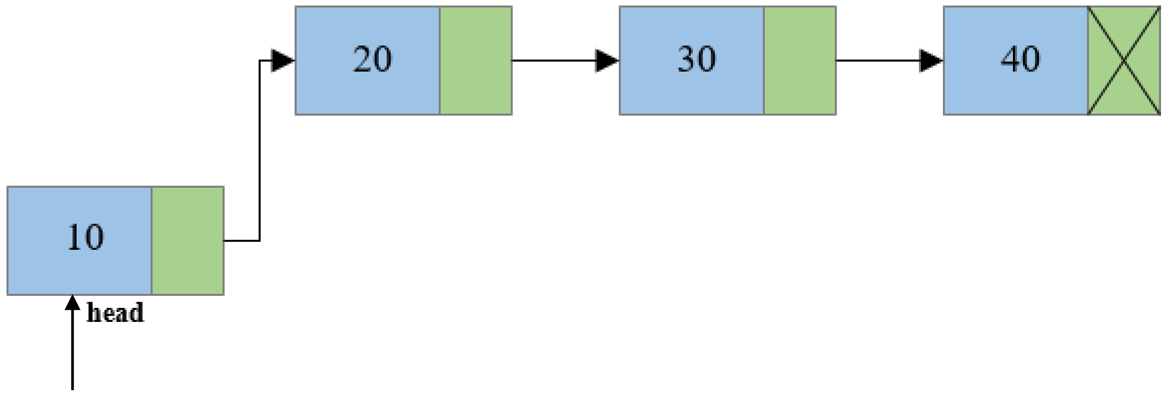
Example: Existing List



Inserting an Element X=10 at the beginning of the linked list L:

```
void insertbegin( Struct Node* head, int elt)
{
 newnode = (Struct Node*)malloc(sizeof(Struct Node))
 newnode →data = 10;
 newnode →next = head;
 head = newnode;   return head;    }
```





At the middle or any given Node (After)

Algorithm:

Step 1: Create a node structure for the new node.

Step 2: Allocate a memory location for the new node.

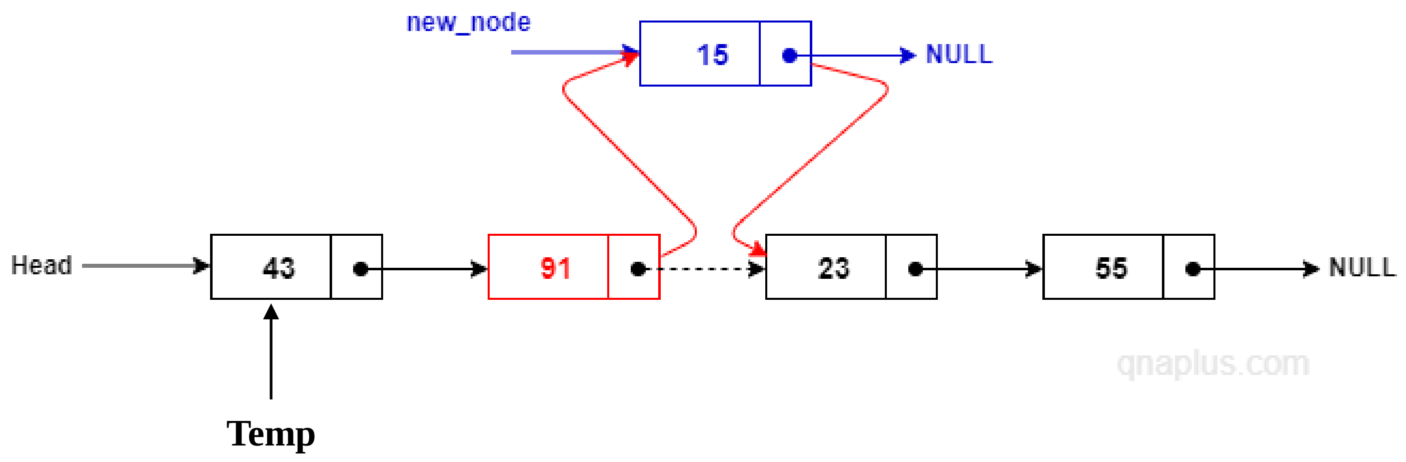
Step 3: Get the data from the user and Put the data on the data field of a node.

Step 4: Identify the new element's position. (**New element is to be inserted after which element** – **Key element)**.

Step 5: Find the key element (Traverse from starting element to the key element).

Step 6: After identifying the key element, address field of the key element is assigned to address field of the new element.

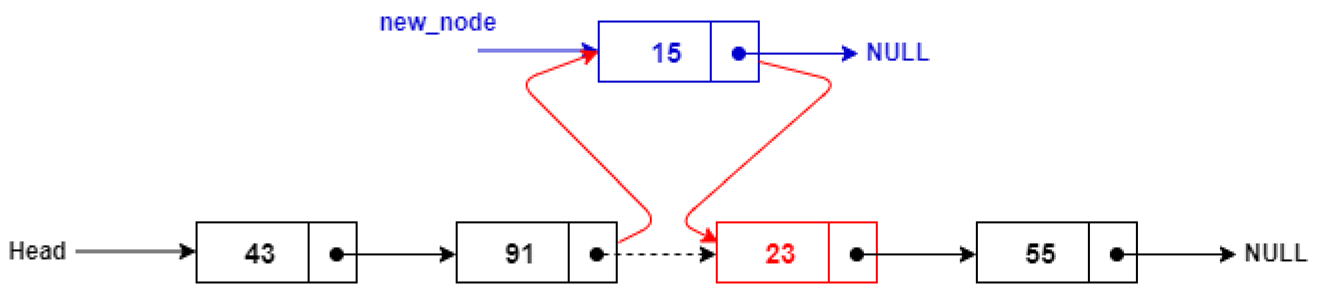
Step 7: New element's address is assigned to address field of key element.



Routine:

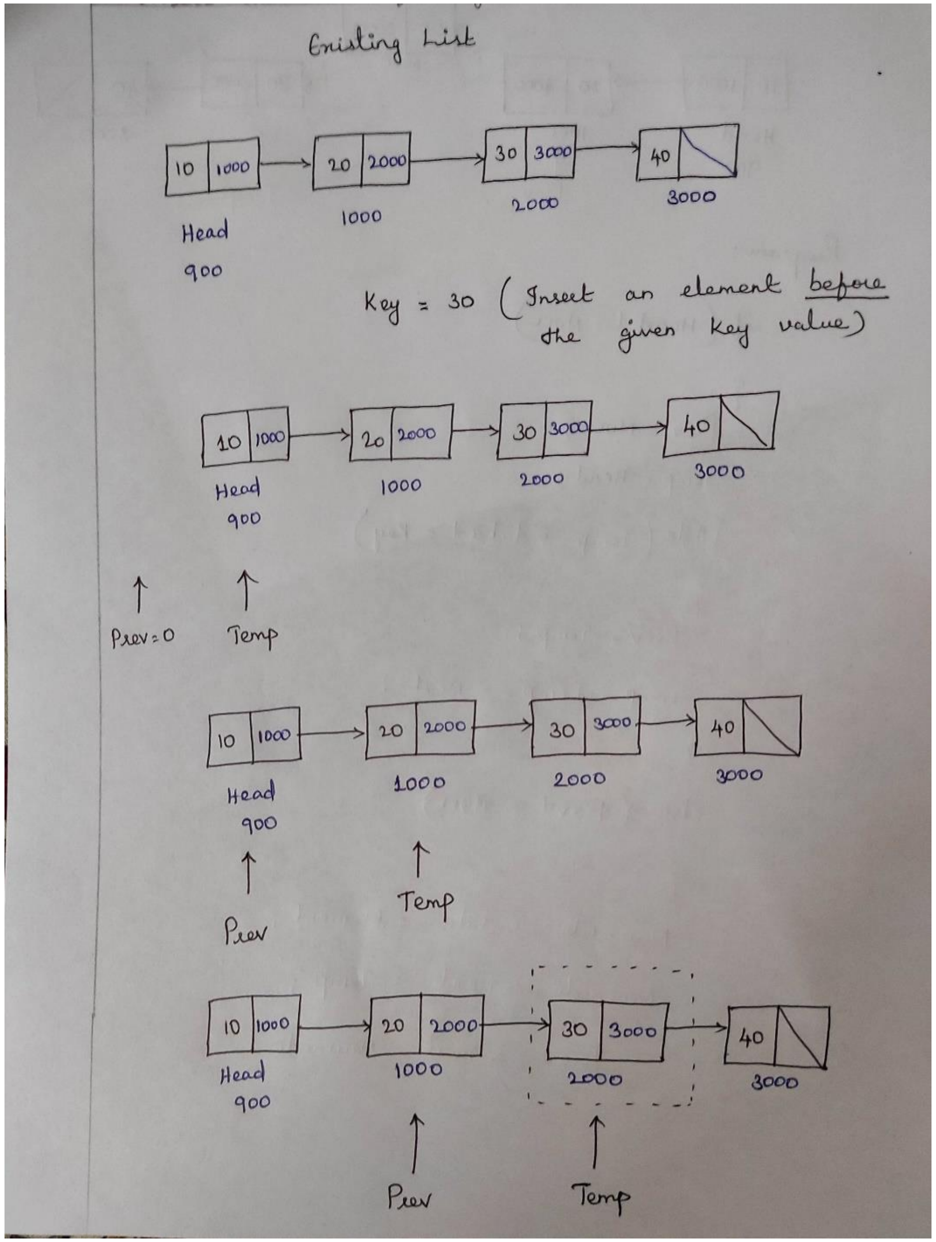
```
New = (Struct Node*) malloc ( sizeof(Struct Node));
scanf ("%d", &element)
New → data = Element;
printf ("\n After which element is to be inserted");
scanf ("%d", &key);
Temp= Head;
if (Temp →data != key)
{
 Temp = Temp → Next;
 }
else
New → Next = Temp → Next;
Temp → Next = New;
```

Insert at the Middle or any given Node (Before):

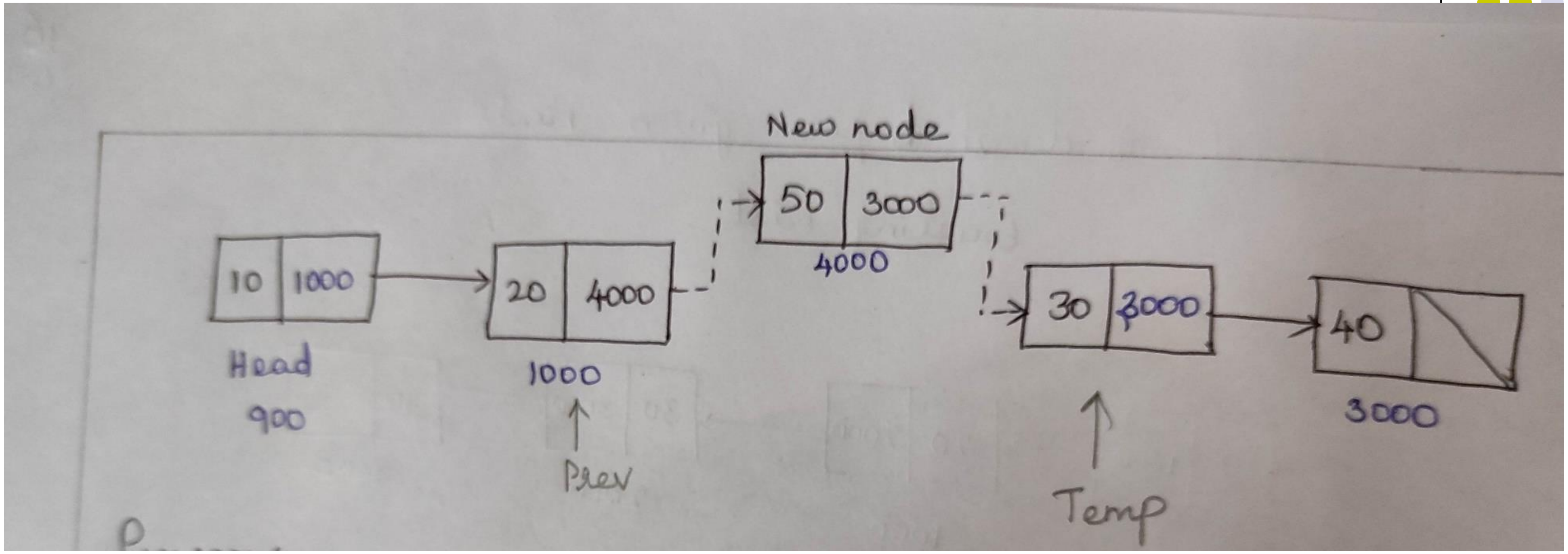


Routine:

```
if ( Head != NULL)
{
 Prev = NULL;
 Temp = Head;
 while ( Temp →data != key)

 {
  Prev = Temp;
  Temp = Temp → Next;
  }
else
{
 New → data = Element;
 New → Next = Temp;
  Prev→ Next = New;
 return Head;
  }
 }
```


Existing List

Key = 30 (Insert an element before the given key value)


New node
50
3000
4000
10
1000
Head
900
20
4000
1000
Prev
30
3000
Temp
40
3000

Inserting an Element at the End :

Algorithm:

Step 1: Create a node structure for a new node.

Step 2: Allocate a memory location for the new node.

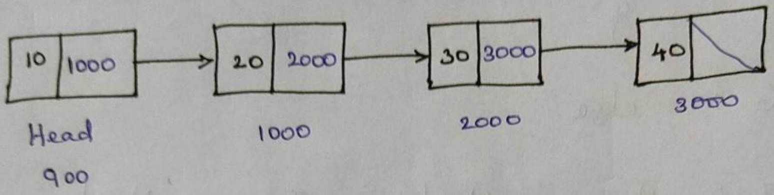
Step 3: Get the data from the user.

Step 4: Put the data in the data field.

Step 5: After identifying the last node in the list, the new node will be inserted. (If the last node's next part is NULL, then this is the end node of the list).

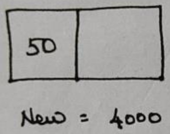
Step 6: New element's address is assigned to address field of the last node.

Example : Existing List



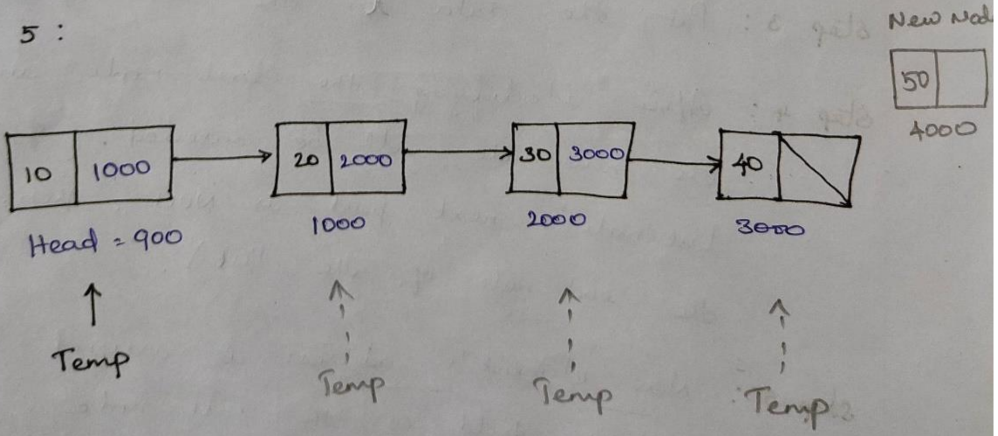
Step 1 : Data = 50 .

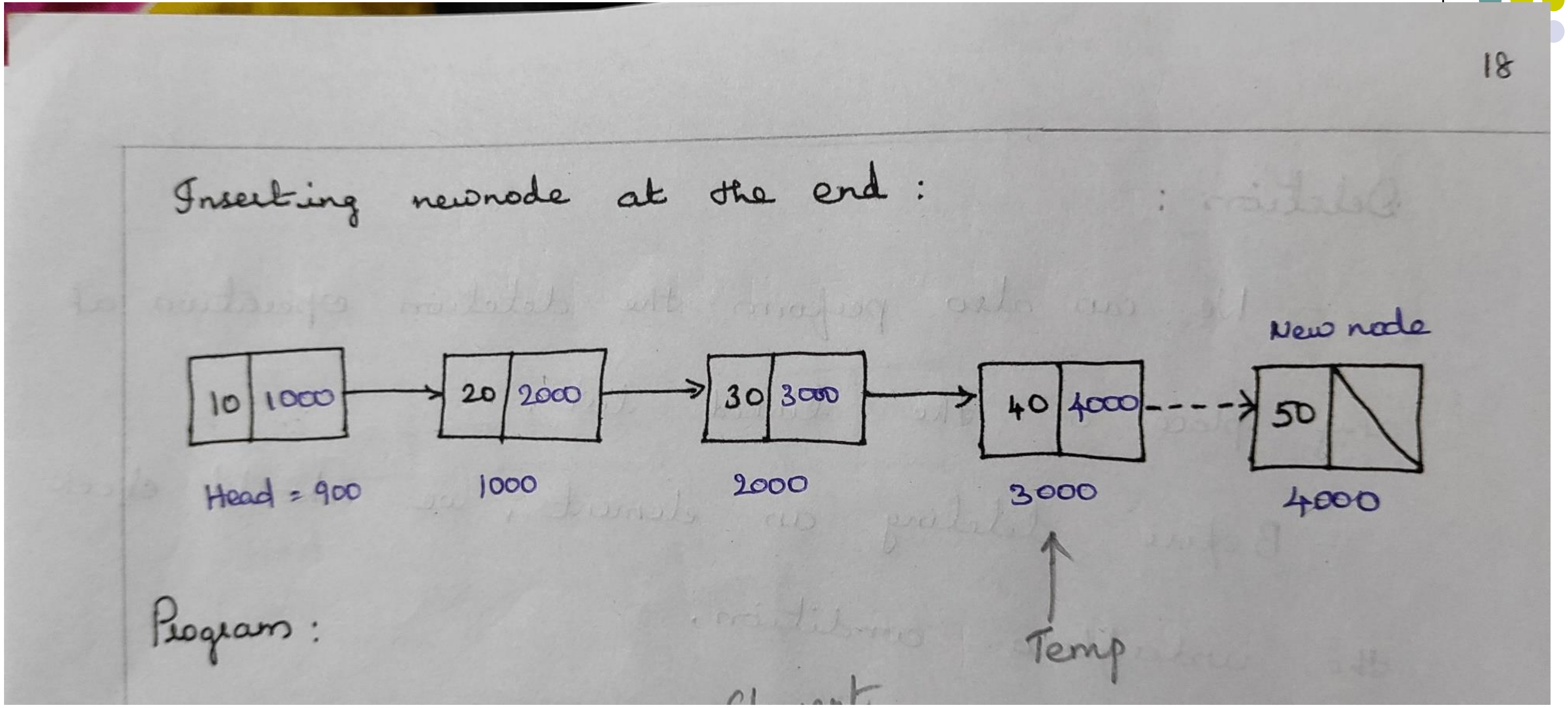
Step 2 & 3 :



Step 4 : Insert the new element at the end of the list .

Step 5 :





Inserting newnode at the end :

Program :

Routine:

```
Struct Node* New = (Struct Node*) malloc ( sizeof (Struct Node));
New → data = Element;
New → Next = NULL;
if ( Head = = NULL)
{
  Head = New;
  return Head;
 }
 Temp = Head;
 while ( Temp → Next != NULL)
 {
  Temp = Temp → Next;
  }
  Temp → Next = New;
  return Head;
  }
```

Counting the Number of Nodes in the List:

```
int CountNodes ( Struct Node * Head)
{
 int count = 0;
 if ( Head == NULL)
     return count;
 Temp = Head;
 while ( Temp != NULL)
 {
  count++;
  Temp = Temp → Next;
  }
  return count;
  }
```

Print the Values Stored in a List:

```
void PrintElements( Struct Node * Head)
{
 if ( Head = = NULL)
      return;
 Temp = Head;
 while ( Temp != NULL)
 {
   printf ( “%d:, Temp → data);
   Temp = Temp → Next;
  }
  return;
 }
```

<u>Sorted Singly Linked List (Inserting a New Node)</u>

Algorithm: Sorting – Arranging the Elements in a particular order

Step 1: Create a node and call it as New.

Step 2: Assign the element to the data field of new node.

Step 3: If the existing list is NULL, call the new node as the list and return.

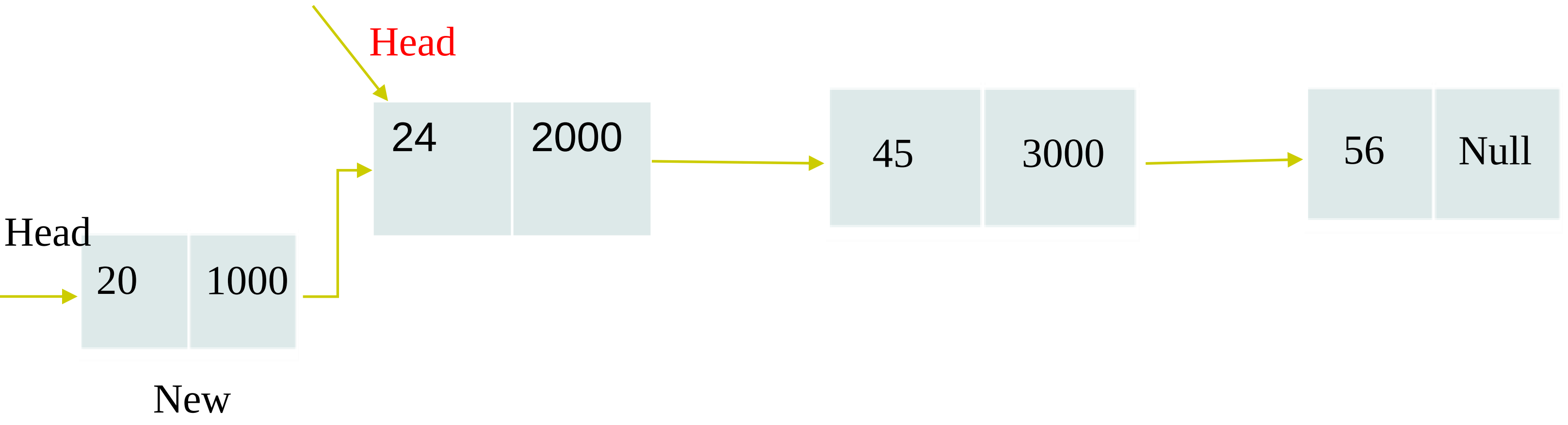
Step 4: Check if the element is less than the one stored in the first node. If so, make the new node as the first in the list and return.

Step 5: Else, use a temporary pointer and traverse the list to locate the position of the element. When this step is finished, Temp should point to the node after which the new node is to be added.

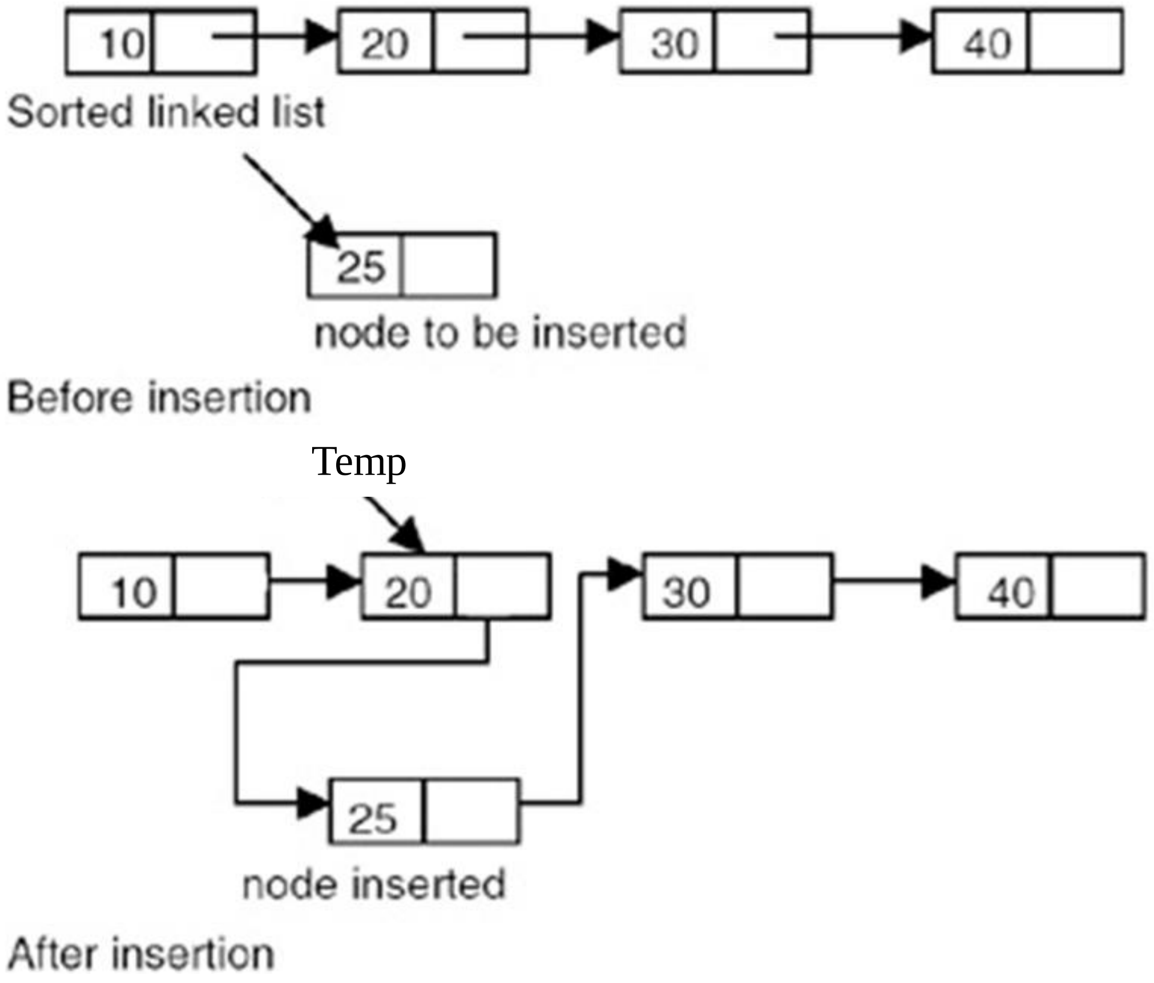
Step 6: Update the next field of New and Temp nodes suitably to make the logical link between the element.

Routine for Inserting an Element at the Beginning in a Sorted List

```
Struct Node * InsertSortList ( Struct Node * Head, int Element)
{
 Struct Node * New, *Temp;
 New = (Struct Node *) malloc ( sizeof(Struct Node));
 New → data = Element;
 New → Next = NULL;
 if ( Head = = NULL || Key < Head → data)
{
 New → Next = Head;
 Head = New;
 return Head;
}
```



Sorted Linked List – Insert an Element at the Middle



```
Struct Node * InsertSortListMid ( Struct Node * Head, int Element)
{
 Struct Node * Newnode, *Temp;
 Newnode = (Struct Node *) malloc ( sizeof(Struct Node));
 Newnode →data = Element;
 Newnode →Next = NULL;
  Temp = Head;
  int  Key = Newnode → data;
  while ( Temp → Next != NULL  && Temp → Next → data < Key)
   {
       Temp → Temp → Next;
    }
     Newnode → Next = Temp → Next;
    Temp → Next = Newnode;  return Head;
   }
```

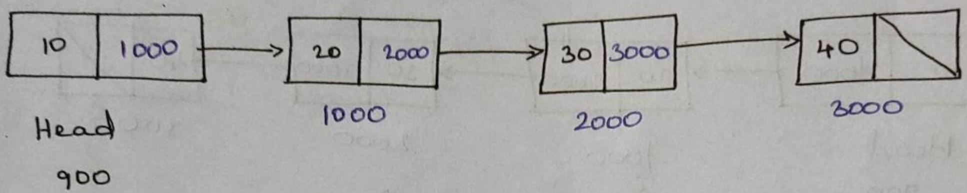
Deleting a Node from the List:

- We can also perform the deletion operation at any place in the linked list.
- Before deleting an element, we should check the underflow condition.

Algorithm:

Step 1: Get the element (key) to be deleted.
Step 2: Check the Underflow condition. If the list is underflow, then print "List is Empty".
Step 3: Otherwise, find an element which is previous or before of the deleting element.
For that we have to traverse from the starting element.
Step 4: If the previous element is NULL, address field of the starting element is assigned to "Head".
Step 5: Otherwise, deleting elements address field is assigned to previous element's address field.

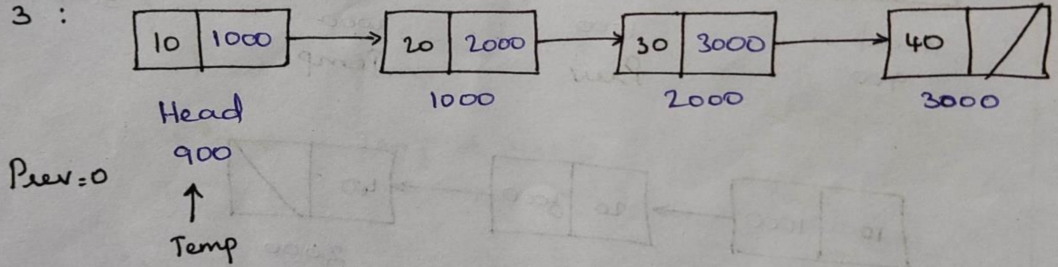
step 1 :

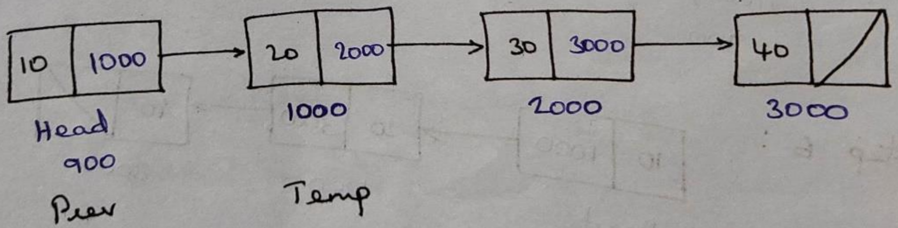


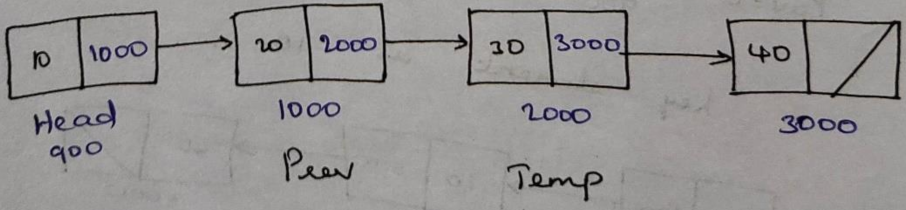
Key element is 30 .

step 2 : Head ≠ NULL , So proceed next step .

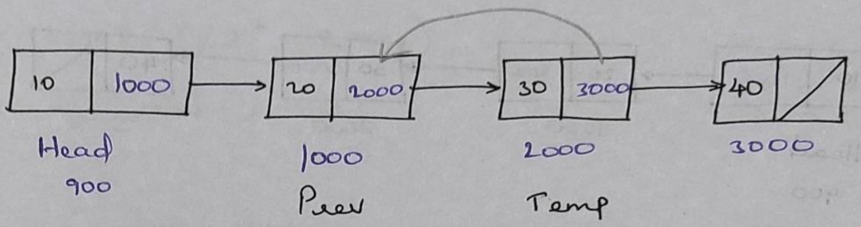
step 3 :

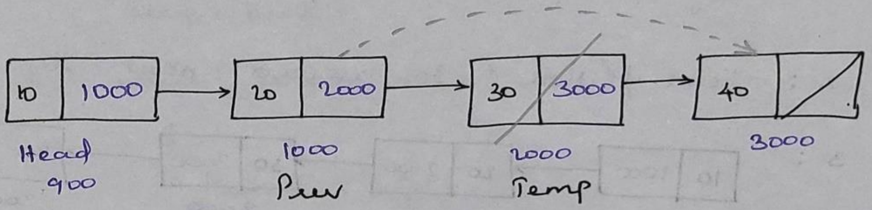


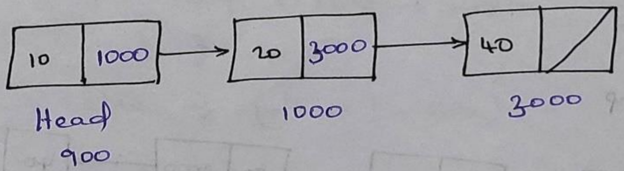




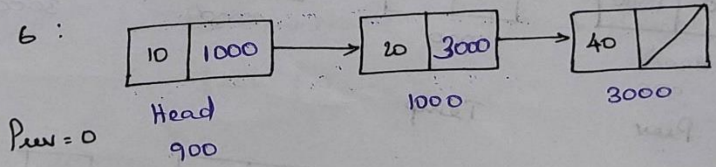
step 5 :



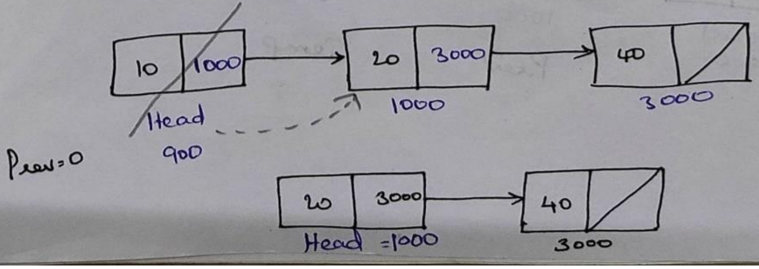




step 6 :



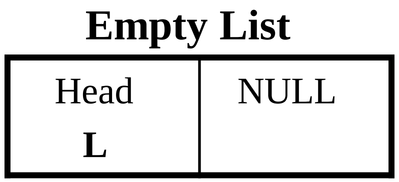
Key element is 10.



Routine :

```
scanf ( “%d”, &key);
if ( Head != NULL)
{
 Prev = NULL;        // See Step 3
 Temp = Head;
 while ( Temp != NULL)
 {
   if ( Temp → data != Key)
   {
   Prev = Temp;        //  See Step 4
   Temp = Temp → Next;
    }
    else if ( Prev != NULL)
    {
     Prev → Next = Temp → Next;       // See Step 5
     free ( Temp);
     }
    else
     {
       Head = Head → Next;       // See Step 6
     }
    }
  }
```

Routine to check whether the List is Empty:



```
int IsEmpty ( List  L)       // L = Head
{
 if ( L → Next = = NULL)
 return (1);
 }
```

Routine to check whether the current position is Last:

```
int IsLast ( position  P, List  L)         // L = Head
{
 if ( P → Next = = NULL)
 return ( 1);
 }
```

Routine to Find an element:

```
position Find ( int X, List L);
{
 position P;
 P = L → Next;
 while ( P != NULL  &&  P → Element != X)
 {
  P = P → Next;
 }
  return P;
 }
```

Routine for Finding Previous Element:

```
position FindPrevious ( int X , List L)
{
 position P;
 P = L;
 while ( P → Next != NULL && P → Next → Element != X)
 {
  P = P → Next;
 }
 return P;
 }
```

Routine for Finding Next Element in the List:

```
position FindNext ( int X, List L)
{
 P = L → Next;
 while ( P → Next != NULL  && P → Element != X)
 {
   P = P → Next;
  }
  return  P → Next;
 }
```

Routine to Delete the Entire List:

```
void DeleteList ( List  L)
{
 position  P, Temp;
 P = L → Next;
 L → Next = NULL;
 while ( P != NULL)
 {
  Temp = P → Next;
   free ( P );
   P = Temp;
 }
 }
```