MODULE V: POINTERS & PREPROCESSORS

INTRODUCTION

• As you know, every variable is a memory-location and every memory-location has its address defined which can be accessed using ampersand(&) operator, which denotes an address in memory.
• Consider the following example, which will print the address of the variables defined:

```
#include<stdio.h>
void  main()
{
     int var1;
     char var2;
     printf("Address of var1 variable: %d \n", &var1 );
     printf("Address of var2 variable: %d", &var2 );
     return 0;
}
```

Output:

```
Address of var1 variable: 1266
Address of var2 variable: 1268
```

POINTER

• A pointer is a variable which holds address of another variable or a memory-location.
• For ex:

```
c=300;
pc=&c;
```

Here pc is a pointer; it can hold the address of variable c
& is called reference operator

DECLARATION OF POINTER VARIABLE

• Dereference operator(*) are used for defining pointer-variable.
• The syntax is shown below:

```
data_type *ptr_var_name;
```

• For ex:

```
int *a;          // a as pointer variable of type int
float *c;        // c as pointer variable of type
```

float • Steps to access data through pointers:

1) Declare a data-variable ex: int c;
2) Declare a pointer-variable ex: int *pc;
3) Initialize a pointer-variable ex: pc=&c;
4) Access data using pointer-variable ex: printf("%d",*pc);

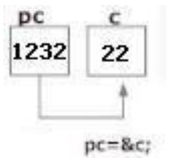
• Example: Program to illustrate working of pointers.

```
#include<stdio.h>
void  main()
{
   int *pc;
   int c;
   c=22;
   printf("Address of c: %d \n", &c);
```

```
    printf("Value of c: %d  \n", c);
   pc=&c;
    printf("Address of pointer pc: %d \n", pc);
    printf("Content of pointer pc: %d",*pc);
}
```

Output:

```
Address of c: 1232
Value of c: 22
Address of pointer pc: 1232
Content of pointer pc: 22
```



Explanation of Program and Figure

- Code int* pc; creates a pointer pc and code int c; creates normal variable c.
- Code c=22; makes the value of c equal to 22, i.e., 22 is stored in the memory-location of variable c.
- Code pc=&c; makes pointer, point to address of c. Note that, &c is the address of variable c (because c is normal variable) and pc is the address of pc (because pc is the pointer-variable).
- Since the address of pc and address of c is same, *pc will be equal to the value of c.

NULL POINTER

- A NULL pointer is defined as a special pointer value that points to '\0'(nowhere) in the memory. In other words, NULL pointer does not point to any part of the memory.
- For ex:

```
int *p=NULL;
```

POINTERS AND ARRAYS

- Consider an array:

```
int arr[4];
```

- The above code can be pictorially represented as shown below:

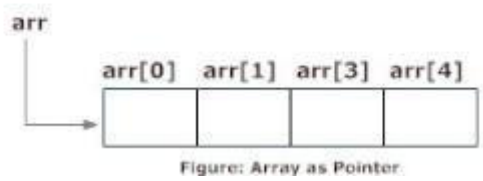


Figure: Array as Pointer

- The name of the array always points to the first element of an array.
- Here, address of first element of an array is &arr[0].
- Also, arr represents the address of the pointer where it is pointing. Hence, &arr[0] is equivalent to arr.
- Also, value inside the address &arr[0] and address arr are equal. Value in address &arr[0] is arr[0] and value in address arr is *arr. Hence, arr[0] is equivalent to *arr.
- Similarly,

```
&a[1] is equivalent to (a+1)  AND, a[1] is equivalent to *(a+1).
&a[2] is equivalent to (a+2)  AND, a[2] is equivalent to *(a+2).
&a[3] is equivalent to (a+1)  AND, a[3] is equivalent to *(a+3).
.
.
```

&a[i] is equivalent to (a+i) AND, a[i] is equivalent to *(a+i).

• You can declare an array and can use pointer to alter the data of an array.

• Example: Program to access elements of an array using pointer.

```
#include<stdio.h>
void main()
{
   int data[5], i;
    printf("Enter elements: ");
    for(i=0;i<5;++i)
         scanf("%d", data+i);
    printf("You entered: ");
    for(i=0;i<5;++i)
         printf("%d   ",*(data+i) );
}
```

Output:

```
Enter elements:  1 2 3 5 4
You entered:  1 2 3 5 4
```

• Example: Program to find sum of all the elements of an array using pointers.

```
#include<stdio.h>
void main()
{
         int i, a[10], n, sum=0;
         printf("Enter the size of the array:");
         scanf("%d", &n);
         printf("Enter the elements into the array: ");
         for(i=0;i<n;i++)
               scanf("%d ",&a[i]);
         for(i=0;i<n;i++)
               sum+=*(a+i);
         printf("Sum --> %d ",sum);
}
```

Output:

```
Enter the size of the array:5
Enter the elements into the array: 2 4 6 10 15 Sum
--> 37
```

POINTERS AND FUNCTIONS

• When, argument is passed using pointer, address of the memory-location is passed instead of value.

• Example: Program to swap 2 number using call by reference.

```
#include<stdio.h>
void swap(int *a,int *b)
{  // pointer a and b points to address of num1 and num2 respectively
      int temp;
      temp=*a;
      *a=*b;
     *b=temp;
}

void  main()
{
    int num1=5,num2=10;
    swap(&num1, &num2);
    //address of num1 & num2 is passed to swap function
    printf("Number1 = %d \n",num1);
    printf("Number2 = %d",num2);
}
```

Output:

```
Number1 = 10
Number2 = 5
```

Explanation

- The address of memory-location num1 and num2 are passed to function and the pointers *a and *b accept those values.
- So, the pointer a and b points to address of num1 and num2 respectively.
- When, the value of pointer is changed, the value in memory-location also changed correspondingly.
- Hence, change made to *a and *b was reflected in num1 and num2 in main function.
- This technique is known as call by reference.

POINTER ARITHMETIC

- As you know, pointer is an address, which is a numeric value.
- Therefore, you can perform arithmetic operations on a pointer just as you can a numeric value.
- There are 4 arithmetic operators that can be used on pointers: ++, --, +, and –
- To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000.
- Assuming 16-bit integers, let us perform the following arithmetic operation on the pointer:

  ```
  ptr++
  ```
- Now, after the above operation, the ptr will point to the location 1002 because each time ptr is incremented, it will point to the next integer location which is 2 bytes next to the current location.

Incrementing a Pointer

- We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer.
- Example: Program to increment the variable pointer to access each succeeding element of the array.

```
#include<stdio.h>
void main()
{
        int var[] = {10, 100, 200};
        int i, *ptr;
        ptr = var;
        for ( i = 0; i < 3; i++)
        {
              printf("Address of var[%d] = %x \n", i, ptr );
              printf("Value of var[%d] = %d \n", i, *ptr );
              ptr++;              //move to the next location
        }
```

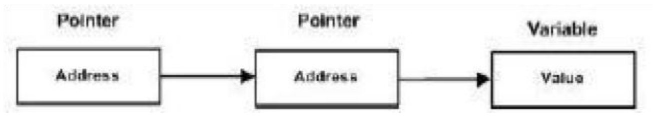
```
}
```

Output:

```
Address of var[0] = 1130
Value of var[0] = 10
Address of var[1] = 1132
Value of var[1] = 100
Address of var[2] = 1134
Value of var[2] = 200
```

POINTERS TO POINTERS

• A variable which contains address of a pointer-variable is called pointer to a pointer.



• A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name.

• For example, following is the declaration to declare a pointer to a pointer of type int:

```
int **var;
```

• When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

```
#include<stdio.h>
void main()
{
        int var;  int *ptr;
        int **pptr;
        var = 3000;     // take the address of var
        ptr = &var;     // take the address of ptr using address of operator &
        pptr = &ptr;   // take the value using pptr
        printf("Value of var = %d \n", var );
        printf("Value available at *ptr = %d \n", *ptr );
        printf("Value available at **pptr = %d ", **pptr);
        return 0;
}
```

Output:

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

PREPROCESSOR

• A preprocessor is not part of the compiler, but is a separate step in the compilation process.

• In simplistic terms, a preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation.

• All preprocessor commands begin with a pound symbol(#).

• List of pre-processor directives: **#include**

This is used insert a particular header from another file.
#define, #undef
These are used to define and un-define conditional compilation symbols.
#if, #elif, #else, #endif
These are used to conditionally skip sections of source code.
#error
This is used to issue errors for the current program.

Use of #include

- Let us consider very common preprocessing directive as below: #include <stdio.h>
- Here, "stdio.h" is a header file and the preprocessor replace the above line with the contents of header file.
- Example: Program to illustrate use of #include

```
#include <stdio.h>
int main()
{
  printf("WELCOME");
  return 0;
}
```

Output:

```
WELCOME
```

Use of #define

- Analyze following examples to understand this directive.
  #define PI 3.1415
- The string 3.1415 is replaced in every occurrence of symbolic constant PI, • Example: Program to find area of a circle using #define.

```
#include<stdio.h>
#define PI 3.1415
int main()
{
    int radius;
    float area;
    printf("Enter the radius: ");
    scanf("%d", &radius);
    area=PI*radius*radius;
    printf("Area=%.2f",area);
    return 0;
}
```

Output:

```
Enter the radius: 3
Area=28.27
```

Use of #if, #elif, #else and #endif

- The preprocessor directives #if, #elif, #else and #endif allows to conditionally compile a block of code based on predefined symbols.
- Example: Program to illustrate this concept.

```
#include<stdio.h>
#define   MAX   100
void main()
{
        #if(MAX)
                printf("MAX is defined");
        #else
                printf ("MAX is not defined");
        #endif
}
```

Output:

```
 MAX is defined
```

Use of #error

- The #error directives allow instructing the compiler to generate an error.
- For example, we wish to issue a warning when a symbol is defined in the current project.
- Example: Program to illustrate use of #error.

```
#include<stdio.h>
#define  MAX   100
void main()
{
        #if(MAX)
        #error: MAX is defined by me!!!
        . .
        . .
        . .
        #endif
}
```

Output:

```
#error: MAX is defined by me!!!
```

Use of #undef

- Consider an example shown below:

```
        #undef FILE_SIZE
        #define FILE_SIZE 42
```

- This tells the preprocessor to undefine existing FILE_SIZE and define it as 42.

MEMORY ALLOCATION FUNCTIONS

- There are 2 types:

    1) Static Memory Allocation:

- If memory-space to be allocated for various variables is decided during compilation-time itself, then the memory-space cannot be expanded to accommodate more data or cannot be reduced to accommodate less data.
- In this technique, once the size of the memory-space to be allocated is fixed, it cannot be altered during execution-time. This is called static memory allocation.
- For ex,

```
            int a[5];
```

    2) Dynamic Memory Allocation

- Dynamic memory allocation is the process of allocating memory-space during execution-time i.e. run time.
- If there is an unpredictable storage requirement, then the dynamic allocation technique is used.
- This allocation technique uses predefined functions to allocate and release memory for data during execution-time.
- There are 4 library functions for dynamic memory allocation:
  1) malloc()
  2) calloc()
  3) free()
  4) realloc()
- These library functions are defined under "stdlib.h"

alloc()

- The name malloc stands for "memory allocation".
- This function is used to allocate the requirement memory-space during execution-time.
- The syntax is shown below: data_type *p; p=(data_type*)malloc(size);

  here p is pointer variable data_type can be int, float or char size is number of bytes to be allocated
- If memory is successfully allocated, then address of the first byte of allocated space is returned.

  If memory allocation fails, then NULL is returned.
- For ex:

  ptr=(int*)malloc(100*sizeof(int));
- The above statement will allocate 200 bytes assuming sizeof(int)=2 bytes.
- Example: Program to find sum of n elements entered by user. Allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int n, i, *ptr, sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));
    //memory allocated using malloc
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
   {
      scanf("%d  ",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
    free(ptr);
}
```

Output:

```
Enter number of elements: 3
Enter elements of array: 2  5  1
Sum= 8
```

calloc()

- The name calloc stands for "contiguous allocation".
- This function is used to allocate the required memory-size during execution-time and at the same time, automatically initialize memory with 0's.
- The syntax is shown below:

```
data_type *p;
p=(data_type*)calloc(n,size);
```

- If memory is successfully allocated, then address of the first byte of allocated space is returned.
  If memory allocation fails, then NULL is returned.
- The allocated memory is initialized automatically to 0's.
- For ex:

```
ptr=(int*)calloc(25,sizeof(int));
```

- The above statement allocates contiguous space in memory for an array of 25 elements each of size of int, i.e., 2 bytes.
- Example: Program to find sum of n elements entered by user. Allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int n,i,*ptr,sum=0;
     printf("Enter number of elements: ");
     scanf("%d",&n);
     ptr=(int*)calloc(n,sizeof(int));
     printf("Enter elements of array: ");
     for(i=0;i<n;++i)
    {
       scanf("%d  ",ptr+i);
       sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
     free(ptr);
}
```

Output:

```
Enter number of elements: 3
Enter elements of array: 2  5  1
Sum= 8
```

free()

- Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.
- The syntax is shown below: free(ptr);
- This statement causes the space in memory pointed by ptr to be deallocated.

realloc()

- If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory-size previously allocated using realloc().
- The syntax is shown below:

```
ptr=(data_type*)realloc(ptr,newsize);
```

- Example: Program to illustrate working of realloc().

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
     int *ptr, i, n1, n2;
    printf("Enter size of array: ");
    scanf("%d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i=0;i<n1;i++)
        printf("%u \n", ptr+i);
    printf("\n Enter new size of array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);
    printf("Address of newly allocated memory: ");
    for(i=0;i<n2;i++)
         printf("%u \n", ptr+i);
}
```

Output:

```
Enter size of array: 3
Address of previously allocated memory:
1022
1024
1026
Enter new size of array: 5
Address of newly allocated memory:
1022
1024
1026
1028
1030
```

MODULE V(CONT.): INTRODUCTION TO DATA STRUCTURES

PRIMITIVE AND NON-PRIMITIVE DATA TYPES

• Data type specifies the type of data stored in a variable.
• The data type can be classified into two types:
  1) Primitive data type and
  2)Non-Primitive data type

Primitive Data Type

• The primitive data types are the basic data types that are available in most of the programming languages.
• The primitive data types are used to represent single values.

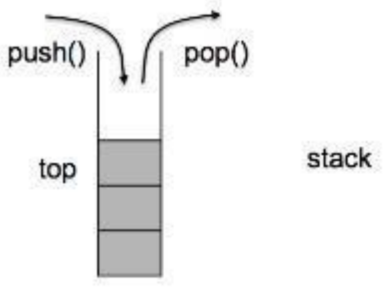
Integer: This is used to represent a number without decimal point.
Eg: 12, 90
Float: This is used to represent a number with decimal point.
Eg: 45.1, 67.3
Character: This is used to represent single character
Eg: „C", „a"
String: This is used to represent group of characters.
Eg: "M.S.P.V.L Polytechnic College"
Boolean: This is used represent logical values either true or false.

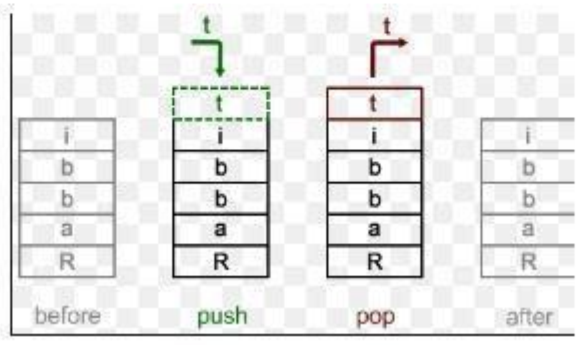
Non Primitive Data Type

- The data types that are derived from primary data types are known as non-Primitive data types.
- These datatypes are used to store group of values.
- The non-primitive data types are
  - Arrays
  - Structure
  - Stacks
  - Linked list
  - Queue
  - Binary tree

STACKS

- A stack is a special type of data structure where elements are inserted from one end and elements are deleted from the same end.
- Using this approach, the Last element Inserted is the First element to be deleted Out, and hence, stack is also called LIFO data structure.
- The various operations performed on stack:
  - Insert: An element is inserted from top end. Insertion operation is called push operation.
  - Delete: An element is deleted from top end. Deletion operation is called pop operation.
  - Overflow: Check whether the stack is full or not.
  - Underflow: Check whether the stack is empty or not.
- This can be pictorially represented as shown below:





APPLICATIONS OF STACK

1) Conversion of expressions: The compiler converts the infix expressions into postfix expressions using stack.
2) Evaluation of expression: An arithmetic expression represented in the form of either postfix or prefix can be easily evaluated using stack.
3) Recursion: A function which calls itself is called recursive function.
4) Other applications: To find whether the string is a palindrome, to check whether a given expression is valid or not.

Program to implement a stack using array.

```
#include <stdio.h> #define
MAXSIZE 5 int stack[MAXSIZE];
int top;
void push()           // Function to add an element to the stack
{
   int num;
   if (top==(MAXSIZE-1))
   {
      printf ("Error: Overflow ");
   }
    else
    {
       printf ("Enter the element to be pushed \n");
       scanf ("%d", &num);
       top = top + 1;
       stack[top] = num;
   }
}

void pop()       //Function to delete an element from the stack
{
    int num;
    if (top==-1)
   {
      printf ("Error: Stack Empty\n");
   }
    else
    {
         num = stack[top];
         printf ("popped element is = %d \n", num);
         top=top-1;
   }
}

void display()     //Function to display the status of the stack
{
   int i;
   if (top == -1)
   {
       printf ("Error: Stack Empty");
   }
    else
    {
         printf ("\n Items in Stack \n");
         for (i = top; i >= 0; i--)
         {
             printf ("%d \n", stack[i]);
        }
    }
}
```

```
void main()
{
         int element, choice;
          top = -1;
         while (1)
         {
                printf ("1. PUSH \n");
                printf ("2. POP \n");
                printf ("3. DISPLAY \n");
                printf ("4. EXIT \n");
                printf ("Enter your choice \n");
                scanf   ("%d", &choice);
                switch (choice)
               {
                     case 1: push();
                                break;
                      case 2: pop();
                                break;
                      case 3: display();
                                break;
                      case 4: return;
               }
        }
}
```

Output:

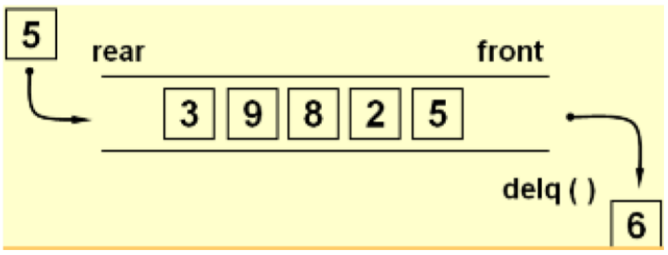
```
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice
1
Enter the element to be pushed
11

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice
1
Enter the element to be pushed
22

1. PUSH
2. POP
3. DISPLAY
4. EXIT
```

```
Enter your choice
1
Enter the element to be pushed
33
```

```
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice
3
Items in Stack
33
22
11

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice
2
popped element is = 33

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice
2
popped element is = 22
```

QUEUES

- A queue is a special type of data structure where elements are inserted from one end and elements are deleted from the other end.
- The end at which new elements are added is called the rear and the end from which elements are deleted is called the front.
- The first element inserted is the first element to be deleted out, and hence queue is also called FIFO data structure.
- The various operations performed on queue are 1) Insert: An element is inserted from rear end.
  2) Delete: An element is deleted from front end.
  3) Overflow: If queue is full and we try to insert an item, overflow condition occurs.
  4) Underflow: If queue is empty and try to delete an item, underflow condition occurs.
- This can be pictorially represented as shown below:



Program to implement a queue using an array.

```
#include <stdio.h>
#define MAX 50
int queue_array[MAX], rear = - 1, front = - 1;
insert()
{
      int add_item;
      if (rear == MAX - 1)
          printf("Queue Overflow ");
      else
      {
           if (front == - 1)        //If queue is initially empty
                 front = 0;
          printf("Insert the element in queue : ");
          scanf("%d", &add_item);
          rear=rear+1;
          queue_array[rear]=add_item;
     }
}         //End of insert()

delete()
{
      if (front == - 1 || front > rear)
     {
          printf("Queue Underflow \n");
          return ;
     }
      else
     {
          printf("Element deleted from queue is : %d\n", queue_array[front]);
          front = front + 1;
     }
}       //End of delete()
```

```
display()
{
     int i;
     if (front == - 1)
          printf("Queue is empty \n");
      else
     {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
           printf("%d ", queue_array[i]);
        printf("\n");
    }
}

main()
{
    int choice;
     while (1)
    {
        printf("1.Insert element to queue \n");
        printf("2.Delete element from queue \n");
        printf("3.Display all elements of queue \n");
        printf("4.Quit \n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch (choice)
        {
           case 1: insert();
                    break;
          case 2: delete();
                    break;
          case 3: display();
                    break;
          case 4: exit(1);
          default: printf("Wrong choice \n");
       }       //End of switch
    }          //End of while
}              //End of main()
```

Output:

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Insert the element in queue : 10

1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Insert the element in queue : 15
```

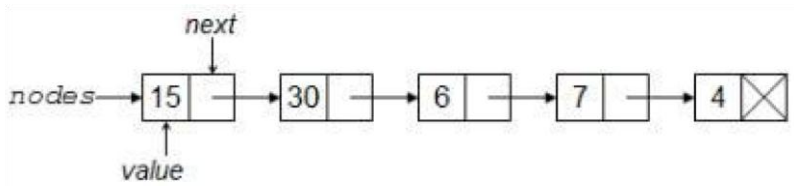
```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 20

1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 30

1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 2
Element deleted from queue is : 10

1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
15 20 30

1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 4
```

LINKED LIST

- A linked list is a data structure which is collection of zero or more nodes where each node has some information.
- Normally, node consists of 2 fields
    1) info field which is used to store the data or information to be manipulated 2) link field which contains address of the next node.
- Different types of linked list are SLL & DLL
- Various operations of linked lists
    1) Inserting a node into the list
    2) Deleting a node from the list
    3) Search in a list
    4) Display the contents of list

SINGLY LINKED LIST

- This is a collection of zero or more nodes where each node has two or more fields and only one link field which contains address of the next node.
- This can be pictorially represented as shown below:



- In C, this can be represented as shown below: struct node // Node Declaration

```
{
     int value;
     struct node *next;
};
typedef struct node snode;
```

- A pointer variable can be declared as shown below

```
snode *first;
```

Program to implement singly linked list using dynamic memory allocation

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

struct node         // Node Declaration
{
     int value;
     struct node *next;
};



typedef struct node snode; snode
*newnode, *ptr, *prev, *temp; snode
*first = NULL, *last = NULL;



snode* create_node(int val)       // Creating Node
{
      newnode = (snode *)malloc(sizeof(snode))
      if (newnode == NULL)
     {
         printf("\nMemory was not allocated");
         return 0;
     }
      else
     {
        newnode->value = val;
         newnode->next = NULL;
         return newnode;
     }
}


void insert_node_first()      // Inserting Node at First
{
     int val;
      printf("\nEnter the value for the node:");
      scanf("%d", &val);
     newnode = create_node(val);
     if (first == last && first == NULL)
     {
        first = last = newnode;
         first->next = NULL;
        last->next = NULL;
     }
      Else
      {
           temp = first;
           first = newnode;
           first->next = temp;
     }
}
```

```
void display()   // Displays non-empty List from Beginning to End
{
     if (first == NULL)
     {
        printf(":No nodes in the list to display\n");
     }
else
{
        for (ptr = first;ptr != NULL;ptr = ptr->next)
        {
           printf("%d\t", ptr->value);
        }
     }
}

int main()
{
     int ch;
     char ans = 'Y';

     while (1)
     {
        printf("\n Operations on singly linked list\n");
printf("\n 1.Insert node at first");           printf("\n
2.Display List from Beginning to end");           printf("\n
3.Exit\n");          printf("\n Enter your choice: ");
scanf("%d", &ch);

        switch (ch)
        {
         case 1:     printf("\n...Inserting node at first...\n");
                insert_node_first();
                break;
         case 2:     printf("\n...Displaying List From Beginning to End...\n");
        display();
                break;
         case 3:     printf("\n...Exiting...\n");
        return 0;
                break;
        default:     printf("\n...Invalid Choice...\n");
        break;
        }
}
    return 0;
}
```

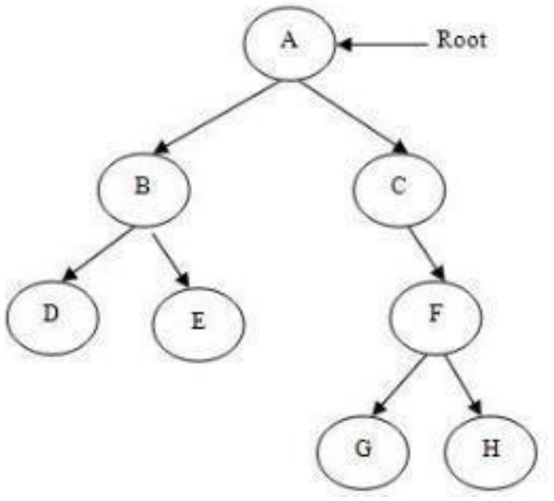
Output:

```
Operations on singly linked list
1.Insert node at first
2.Delete Node from any Position
3.Exit
Enter your choice: 1
...Inserting node at first...
Enter the value for the node: 100
```

```
Operations on singly linked list
1.Insert node at first
2.Delete Node from any Position
3.Exit
Enter your choice: 1 ...Inserting node at first...
Enter the value for the node: 200

Operations on singly linked list
1.Insert node at first
2.Delete Node from any Position
3.Exit
Enter your choice: 1 ...Inserting node at first...
Enter the value for the node: 300

Operations on singly linked list
1.Insert node at first
```

```
2.Delete Node from any Position
3.Exit
Enter your choice: 2
...Displaying List From Beginning to End...
100     200   300
```

BINARY TREE
• A tree in which each node has either zero, one or two subtrees is called a binary tree.
• figure
• Each node consists of three fields llink: contains address of left subtree
info: this field is used to store the actual data or information to be manipulated
rrlink: contains address of right subtre
• This can be pictorially represented as shown below:



BINARY TREE APPLICATIONS
• Tree traversal refers to process of visiting all nodes of a tree exactly once (Figure 5.16).
• There are 3 techniques, namely: 1) Inorder traversal(LVR);
2) Preorder traversal(VLR);
3) Postorder traversal(LRV). (Let L= moving left, V= visiting node and R=moving right).
• In postorder, we visit a node after we have traversed its left and right subtrees.
In preorder, the visiting node is done before traversal of its left and right subtrees.
In inorder, firstly node"s left subtrees is traversed, then node is visited and
• For above tree, tree traversal is as follows
Inorder traversal → DBEACGFH
Preorder traversal → ABDECFGH
Postorder traversal → DEBGHFCA