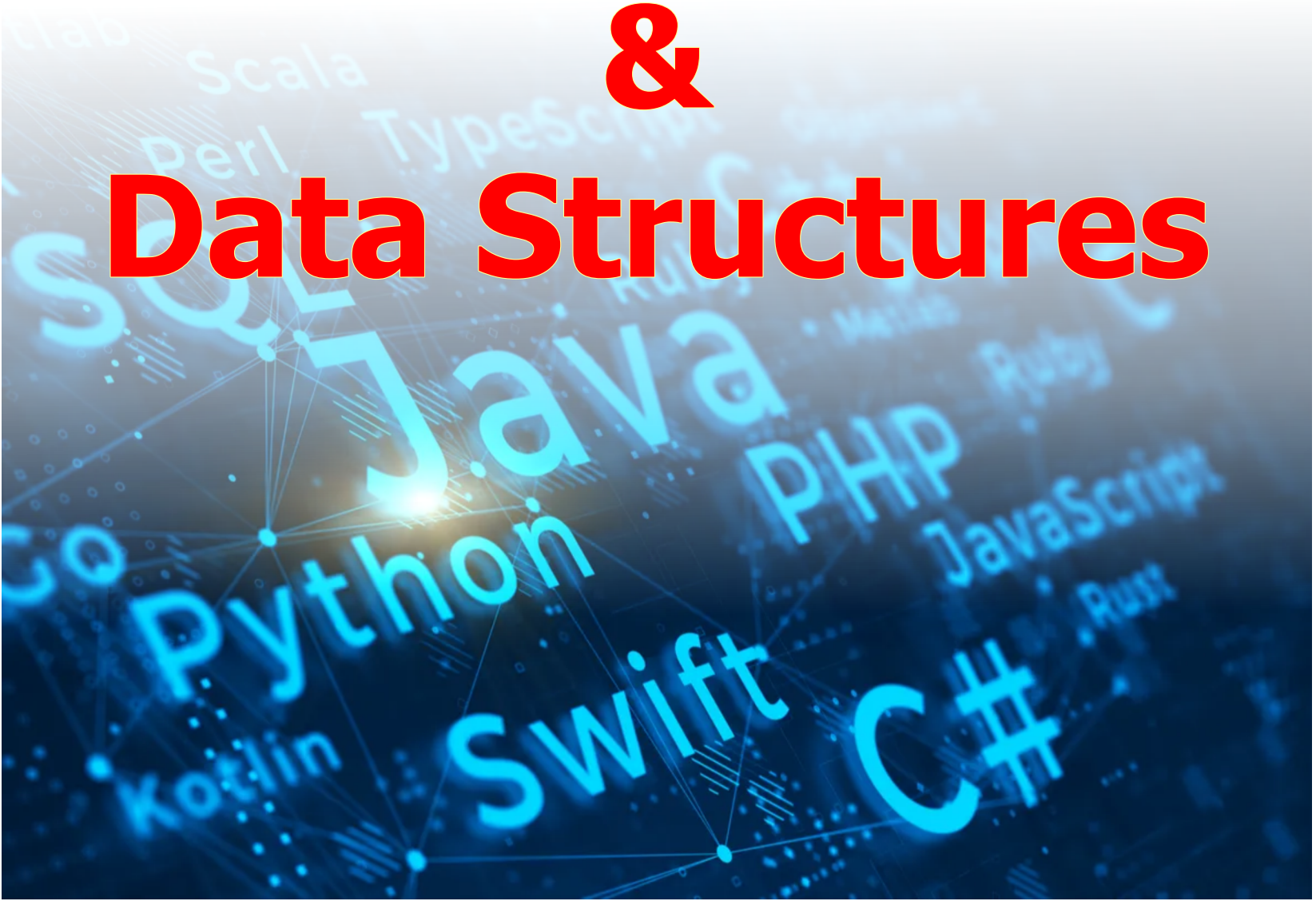
Programming & Data Structures



Published By:

Physics Wallah

ISBN: 978-93-94342-39-2

Mobile App: Physics Wallah (Available on Play Store)

Website: www.pw.live

Email: support@pw.live

Rights

Disclaimer

A team of PW experts and faculties with an understanding of the subject has worked hard for the books.

While the author and publisher have used their best efforts in preparing these books. The content has been checked for accuracy. As the book is intended for educational purposes, the author shall not be responsible for any errors contained in the book.

The publication is designed to provide accurate and authoritative information with regard to the subject matter covered.

(This Module shall only be Used for Educational Purpose.)

Programming & Data Structure

INDEX

1. Data Types and Operators ..... 5.1 – 5.5
2. Control Flow Statements ..... 5.6 – 5.8
3. Storage Class & Function ..... 5.9 – 5.12
4. Arrays and Pointers ..... 5.13 – 5.22
5. Strings ..... 5.23 – 5.26
6. Types of Data Structure, Array & Linked List ..... 5.27 – 5.31
7. Stack and Queue ..... 5.32 – 5.33
8. Tree Data Structure ..... 5.34 – 5.38
9. Graph Traversal 5.39
10. Hashing ..... 5.40 – 5.41

1 DATA TYPES AND OPERATORS

1.1 Data Types

1.1.1 Primitive Data Type

(a) Integer Types:

- ✓ short int, unsigned short int
- ✓ int, unsigned int
- ✓ long int, unsigned long int
- ✓ long long int, unsigned long long int

(b) Character Types:

- ✓ char,unsigned char

(c) Floating Types:

- ✓ float, double, long double

(d) Other:

- ✓ void, bool

1.1.2 Non – Primitive Data Type

(a) Derived data type:

- ✓ Array
- ✓ Pointer
- ✓ String

(b) User defined data type:

- ✓ Structure
- ✓ union
- ✓ Enum
- ✓ typedef

- C standard does not specify how many bits or bytes to be allocated for every type and different compiler may choose different ranges.
- Only restriction is that short and int types are at least 16 bits, long types are at least 32 bits and short is no longer than int which is no longer that long type.

1.2 Operators

Depending upon the number of operands, an operator in C language can be unary, binary or ternary.

1.2.1 Arithmetic Operators

(i) Addition (+)
(ii) Subtraction (-)
(iii) Multiplication (×)
(iv) Division (/)
(v) Modulus (%)

- Both operands for % operator must be integer types, otherwise error will be raised.
- The sign of the result for modulo operator is machine dependent.

1.2.2 Relational Operators

(i) Less than (<)
(ii) Greater than (>)
(iii) Less than equal to (< =)
(iv) Greater than equal to (> =)
(v) Equal checking (= =)
(vi) Not Equal to (! =)

- The result of these operators is always 0 to 1

  Ex.1: printf ("%d",20 > 10)
  O/P: 1
  Ex.2: printf ("%d",20 = = 10)
  O/P: 0

1.2.3 Assignment Operator (=)

- Used to assign a value or value of an expression to a variable.
- Typically, the system is
  Lvalue = Rvalue
- Lvalue must be a variable.
- Lvalue cannot be a literal or expression.
- Rvalue can be an expression, variable, literal.

Ex1.

The following are invalid

```
10 = a;
a + b = c;
```

- The result of assignment statement is the value we are assigning i.e……

```
int x;
x = 3 + (x = 10); is perfectly valid.
Firstly x = 10 evaluated and
(1) 10 is assigned to x   (2) then the result of (x = 10) will be 10
so,
x = 3 + 10
```

x = 13
Finally, 13 is assigned to x.

1.2.4 Logical Operators

(i) Logical AND (&&)
(ii) Logical OR (||)
(iii) Logical NOT (!)

- Just like rational operators, the result of every logical operator is either 0 or 1.
- Logical NOT is a unary operator
- Logical NOT converts a non-zero operand into 0 and a zero operand in 1.

1.2.5 Short Circuiting in Logical Operators

- In case of logical AND, the second operand is evaluated only if the first operand is evaluated to be true.
  If the first operand is evaluated as false then the second operand is not evaluated.
- In case of logical OR operation, the second operand is not evaluated if the first operand is evaluated as true.

Example1:

```
void main () {
printf (“Hello”) || printf (“Pankaj”);
}
```

O/P: Hello

printf function display everything written within double quotes on the monitor and the result / value returned by printf () is the number of characters successfully displayed on the screen.

So, printf (“Hello”) prints Hello and return 5.

So, the expression

```
printf (“Hello”) || printf (“Pankaj”);
```

becomes

```
5 || printf (“Pankaj”)
```

As the first operand for logical OR is evaluated as true, second printf () will never execute.

1.2.6 Increment and Decrement Operators

(i) pre increment ++x
(ii) post increment x++
(iii) pre decrement - -x
(iv) post decrement x - -

- unary operator.
- can’t be applied on constant / literals.
- Pre increment: can be viewed as 2 step operators.
  1st step: Increment the value of variable.
  2nd step: After increment, use the value of variable.
- Post increment: can be viewed as 2 step operators
  1st step: Use the value.
  2nd step: Increase the value of variable by 1.

1.2.7 Bitwise operators

(i) Bitwise AND (&)
(ii) Bitwise OR (|)
(iii) Bitwise XOR (^)
(iv) Bitwise Left Shift (<<)
(v) Bitwise Right Shift (>>)
(vi) Bitwise Not (~)

- All these operators work on binary representation of numbers.
- Typically, faster than arithmetic operators and other operators.

```
#include<stdio.h>
    int main () {
            int x = 3, y = 6;
            printf ("%d\n",x & y);    // output 2
            printf ("%d\n",x|y);      // output 7
            printf ("%d\n", x^y)      // output 5
            return 0;
            }
```

binary representation of 3: 00000000...0011
binary representation of 6: 00000000...0110

3 & 6: 00000000...0010
3 | 6: 00000000...0111
3 ^ 6: 00000000...0101

XOR of two bits is 1 if both bits are different.
XOR of two bits is 0 if both bits are same.

- ~ x has output as – (x + 1)
- printf ("%d\n", ~2) has output -3 i.e., -(2 + 1)

- Comma has lowest precedence among all operators in C language.
- sizeof() is used
  (i) To find the number of elements present in an array.
  (ii) To dynamically allocate block of memory.

1.2.8 Ternary operator (? :)

- It requires 3 operands i.e., Left, Middle, Right
  Left ? Middle : Right

- If left expression evaluates as true, then the value returned is middle argument otherwise the returned value is Right expression

int a;

a = 7 > 2 ? 3 × 2 + 5 : 6! = 7

⇒ Left expression: 7 > 2 is true

⇒ So, the value returned would be the middle argument.

i.e., 3 × 2 + 5 = 11

so, a will get 11.

Note:
(a) No of '?' and ':' should be equal.
(b) Every colon (:) should match with just before '?'.
(c) Every ? followed by : not immediately but following.

Example: int a;

a = 2 > 5 ? 10 : !5! = 2 > 5 ? 20 : 30

L_1 M_1 R_1

2 > 5 is false.

so, for Leftmost ? the value returned is its right expression.

a = ! 5 ! = 2 > 5 ? 20 : 30 :

Left Mid Right

Left expression:

!5! = 2 > 5

0! = 2 > 5

0! =0

0 i.e., false.

As left operand is false, the final value is right expression

a = 30

Operators	Associativity
() [] -> .	Left to right
! ~ ++ -- + - * *(type)* size of	Right to left
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
= = ! =	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
? :	Right to left
= += -= *= /= %= &= ^= |= <<= >>=	Right to left
,	Left to right

❐❐❐

2 CONTROL FLOW STATEMENTS

2.1 Switch Statement

- "Case Value" can be of character type and int type.
- There can be one or many number of cases.
- Statements associated with a matching case executes until a break statement is reached.
- The position of default case does not matter. It can be placed anywhere.
- Default case is optional.
- Duplicate case labels are not allowed.
- Statements written before all the cases are ignored by the compiler.
- The break statement is optional.
- Case label cannot be a variable.

2.2 Iterative statements (Loop)

An iterative statement, or loop, repeatedly executes a group of statements, known as body of loop, until the controlling expression is false.

2.2.1 For Loop

- The for statement evaluates 3 expressions and executes the loop body until second controlling expression executes to false.
- It is recommended to use for loop when the number of iterations is known in advance.
- Syntax of for loop is:

```
for (expression – 1(optional); expression – 2(optional); expression – 3(optional))
{
    statement – 1
    statement – 2
    ⋮
    statement – n
    }
```

- for loop executes the loop body 0 or more times.
- for statements works as follows:

(a) expression – 1 is evaluated once before the first iteration of the loop.

(b) expression – 2 is a expression that determines whether to terminate the loop. expression – 2 is evaluated before every iteration.
If the expression is true (non - zero), the loop body executed.
If the expression is false (zero), execution of the for statement is terminated.

(c) expression – 3 is evaluated after each iteration.

(d) The for statement executes until expression-2 is false (0), or until a jump statement terminates execution of the loop.

- All 3 expression can be omitted

(a) If we omit expression-2, the condition is considered as true for an example:

```
for (i = 0;  ; i++)
statement.
```
represent an infinite loop

2.3 While Statement

- The while statement evaluates a controlling expression before every execution of the loop body.
- If the controlling expression is true (non-zero), the loop body is executed. If the controlling expression is false (zero), then the while statement terminates.
- Syntax:

```
while (expression)
statement

OR

while (expression)
{
statement-1
statement-2
⋮
statement-n
}
```

2.4 Do-while Loops

- Both for and while loop checks the loop termination condition before every iteration but do while loop check the condition after executing the loop body.
- do while loop body executes at least 1 time, no matter whether the loop termination condition is false or true.
- Syntax is:

```
do
statement
while (expression);

OR

do {
statement-1
statement-2
⋮
statement-n
} while (expression);
```

2.5 Break Statement

- The break statement provides an early exit from for, while and do while.
- A break statement causes the innermost loop or switch to be exited immediately.
- In implementation, when we know the maximum number of repetition but some condition is there, where we require to terminate the repetition process, then use break statement.

Example:

```
void main ()
{
int i = 1;
while (i < = 15)
    {
```

```
        printf("%d\t",i);
        if (i > 4)
            break;
        i = i + 1;
        }
    printf("End");
    }
    Output: 1 2 3 4 End
```

In above code, when i becomes 5, the condition i > 4 becomes true, so break statement executes & control passed outside of the loop body i.e., printf("End") executed.

2.6 Continue Statement

- Continue statement is related to break but it is not used that much frequently.
- Whenever continue is encountered in a loop, it skips the remaining statement of the current iteration and continues with the next iteration of the loop.
- Only applicable with loops, not with switch statement.

Example:

```
    void main () {
        int i = 1;
        for (i = 1; i < = 10; i++)
        {
            if(i%2 = = 0)
            continue;
            printf("%d",i);
        }
    }
    Output: 13579 (All odd numbers between 1 to 10)
```

whenever i becomes even, condition i%2 = = 0 becomes true & continue causes the control to go to i++ & printf is skipped.

❐❐❐

3 STORAGE CLASS & FUNCTION

3.1 Memory Organization of C Program

3.1.1 Code Segment

- It is also known as text segment.
- It contains executable instructions and this segment is a read only segment.
- Usually, this section is sharable.

3.1.2 Uninitialized data segment

- This section contains all static and global variables that are not initialized by the programmer and hence initialized to zero (default value).

3.1.3 Initialized data segment

- This section contains all static and global variables that are initialized by the programmer.

3.1.4 Stack

- In this local variables are stored and some other information is also saved.
- A set of values stored for a function is called as stack frame which atleast contain return address.

3.1.5 Heap

- This area is responsible for dynamic memory allocation. The heap area is managed by malloc(), calloc(), ralloc() and free() which may be use brk() and sbrk() system calls.

3.1.6 Static Memory

- Compile time allocation
- Static variable
- Global Variable

3.2 Storage Classes

Storage Class	Scope	Life Time	Default	Storage area
auto	Local	Within function	Garbage Value	RAM
static	Local	Till end of main program	0	RAM

extern	Global	Till end of main program	0	RAM
Register	Local	Within function	Garbage	Register

- Stack Overflow: Abnormal Termination
- If conflict between global and local variable occurs, then local variable gets preference.
- Declaration of a register variable is only a recommendation/request not a command.
- Cannot apply '&' operator with register variable.
- No physical memory is allocated using 'extern' keyword.
- Extern declaration is mandatory when an external variable is referred before it is defined or if it is defined in some other source file from the file where it is being used.
- Declaration of an external variable tells about the properties like its type, while definition leads to storage allocation.

3.3 Recursion

Definition: When the body of a function call the function itself directly or indirectly.

- Certain arguments for which the function does not call itself are called as base argument, base values.
- In general, for recursion to be non-cyclic whenever a function calls itself the formal arguments must get closer to the base argument.

Example 1:

Consider the factorial code:

```
int factorial (int n)
    {
    if (n = = 0)
    return 1;
    else
    return n * factorial (n - 1);
    }
```

If we call factorial (3), it will call factorial of 2 and so on …the argument will get closer to 0 i.e., the base argument.

- Recursion code is shorter than iterative code.
- Overhead is present.
- Some standard problems are best suited to be solved by using recursion for example- Tower of Hanoi, Factorial, merge Sort.

3.4 Static and dynamic scoping

3.4.1 Static Scoping

Static scoping is also called as lexical scoping. In this scoping, the binding of a variable can be determined by the program text. In this type of scoping compiler first looks in the current block (local), then in global variables i.e., local and then ancestors' strategy is followed.

Example 1:

```
int a = 10;
void main ()
{
    int a = 1;
    {
        int a = 2
        printf("%d",a);
        }
    }
```

Output: 2

As printf is referring to a, compiler first check in the current block & gets a variable whose value is 2.

Example 2:

```
int a = 10;
void main ()
{
int a = 1;
{
int b;
printf("%d",a);
}
}
```

Scope S_1

Scope S_2

Output: 1

Compiler looks for 'a' in scope S_1, because S_1 does not have variable a, it will look into higher scope S_2 that contains a variable name a whose value is 1.

3.4.2 Dynamic Scoping

In this type of scoping, the compiler first searches the current block and then successively searches all calling functions.

Example:

```
int i;
program main ()
{
    i = 10;
    call f ();
}
procedure f ()
{
    int c = 20;
    call g();
}
Procedure g() {
print i;
    }
```

Assuming that the above program is written in a hypothetical programming language which allows global variables

and dynamic scoping, let's try to find the output.
The order of function calls is:

$$\text{main ()} \xrightarrow{\text{calling}} \text{f()} \xrightarrow{\text{calling}} \text{g()}$$

g() is printing i, which is not present inside current block. So, because of dynamic scoping compiler will go to the function that calls g() i.e., compiler will search f() for variable i. Hence 20 is printed.

❐❐❐

4 ARRAYS AND POINTERS

4.1 Array

- An array is defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double etc.
- It has the capability to store the collection of derived data types such as pointer, structure etc.
- Each element of an array is of same data type and carries the same size example int = 4 byte.
- Elements of the array are stored at contiguous memory locations, meaning, the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of array with the given base address and size of the data element.

4.1.1 Advantages of C Array

1) Code Optimization: Less code to access the data.
2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.
3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.
4) Random Access: We can access any element randomly using the array.

4.1.2 Disadvantages of Array

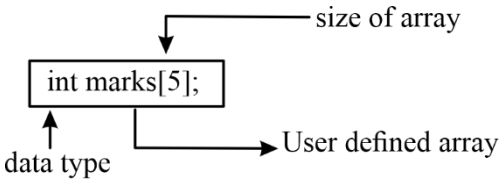
- **Fixed Size**: **1)** Whatever size, we define at the time of declaration of the array, we can't exceed the limit.
  2) It does not grow the size dynamically like linked list.

Declaration of C Array

Syntax:

```
data type array name [array_size]
```

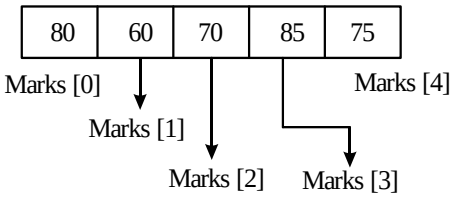
Example:



Initialization of C array:

We can initialize each element of the array by using the index. Suppose array int marks [5]. Then

```
marks [0] = 80
marks [1] = 60;
marks [2] = 70;
marks [3] = 85;
marks [4] = 75;
```



C Array Example:

```
# include <stdio.h>
int main ()
{
int i = 0;
int marks [5];
marks [0] = 80;
marks [1] = 60;
marks [2] = 70;
marks [3] = 85;
marks [4] = 75;
for(i = 0; i<5; i++)
{
    printf("%d\n",marks[i]);
}
return 0;
}
```

4.1.3 Array: Declaration with Initialization

- We can initialize the C array at the time of declaration.
  Example: int marks [5] = {20, 30, 40, 50, 60};
- In such case, there is no requirement to define the size.
  Example: int marks [ ] = {20, 30, 40, 50, 60}

Example:

```
# include<stdio.h>
int main ()
{
int i = 0;
int marks [5] =  {20, 30, 40, 50, 60}
for(i = 0; i < 5; i++)
    {
        printf("%d\n",marks[i]);
    }
return 0;
}
```

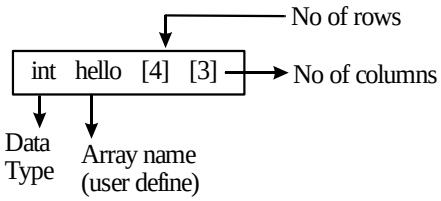
4.1.4 Two-Dimensional Array

- The two-dimensional array can be defined as an array of arrays.
- The 2D array is organized as matrices which can be represented as the collection of row and columns.

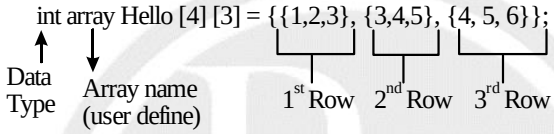
4.1.5 Declaration of two-dimensional array in C syntax

```
data_type array_name [rows] [columns]
```

Example:



Initialization of 2D Array



Example: Program of Two-Dimensional Array

```
# include<stdio.h>
int main()
{
    int i = 0, j = 0;
    int arrayHello[4][3] = {{1, 2, 3}, {2, 3, 4}, {4, 5, 6}};
    for(i = 0; i < 3; i++)
    {
    for(j = 0; j < 3; j++)
        {
            printf("arrayHello[%d][%d]=%d\n",i,j,arrayHello[i][j])
        }
    }
    return 0;
}
```

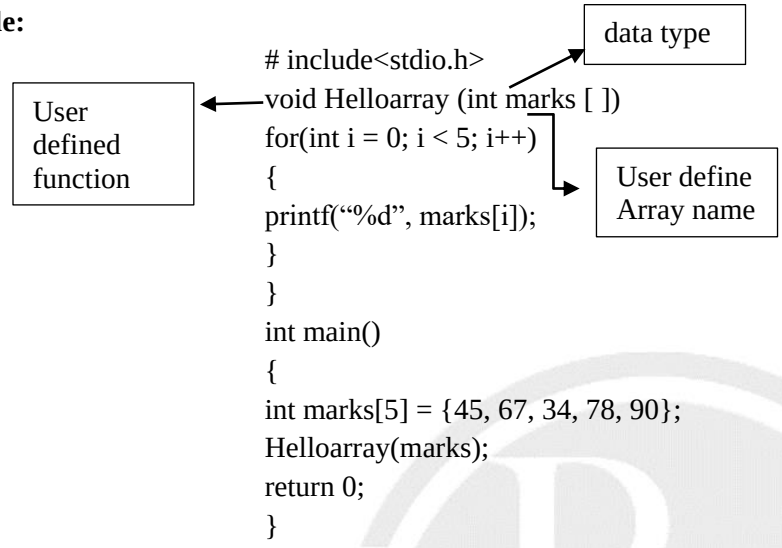
Output:

```
arrayHello[0][0] = 1
arrayHello[0][1] = 2
arrayHello[0][2] = 3
arrayHello[1][0] = 2
arrayHello[1][1] = 3
arrayHello[1][2] = 4
arrayHello[2][0] = 3
arrayHello[2][1] = 4
arrayHello[2][2] = 5
arrayHello[3][0] = 4
```

arrayHello[3][1] = 5
arrayHello[3][2] = 6

4.1.6 Passing array to a function

Example:




```
# include<stdio.h>
void Helloarray (int marks [ ])
for(int i = 0; i < 5; i++)
{
printf("%d", marks[i]);
}
}
int main()
{
int marks[5] = {45, 67, 34, 78, 90};
Helloarray(marks);
return 0;
}
```

Note: void Helloarray(int marks[ ])

Without return type function

4.1.7 Passing Array to a Functions as a pointer

Now, we will see how to pass an array to a function as a pointer.

```
# include<stdio.h>
void helloarray(char * marks)
{
    printf("Elements of array are:");
    for(int i = 0; i < 5; i++)
    {
    printf("%c", marks[i])
    }
    int main()
    {
    char marks[5] = {'A', 'B', 'C', 'D', 'E'};
    helloarray(marks);
    return 0;
    }
}
```

Note:
Whenever you pass an array, it is always call by reference. In previous 2 programs, we passed an address & formal argument in both the cases is a pointer internally.

How to return an Array from a function

```
# include<stdio.h>
int *fun()
{
int marks [5];
print f (“Enter the element in an array”);
for (int i = 1; i < 5; i++)
{
scanf(“%d”,& marks[i]);
}
return marks;
}
int main()
{
int *n;
n = fun();
printf(“\n Elements of array are:”);
for(int i = 0; i < 5; i++)
{
printf(“%d”,n[i]);
}
return 0;
}
```

Note:
fun() function returns a variable ‘marks’.
It returns a local variable, but it is an illegal memory location to be returned, which is allocated with a function in the stack. Since the program control comes back to the main () function, and all the variables in the stack are fraud.
Therefore, we can say that this program is returning memory location, which is already de-allocated, so the output of the program is a **segmentation fault.**

4.1.8 Array declaration and initialization

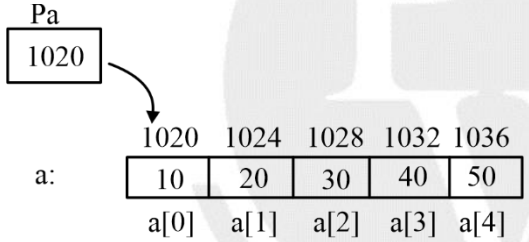
1. int A[ ] = {10, 20, 30}: valid
2. int A[3] = {10, 20, 30}: valid
3. int A[ ] ; invalid
4. int A[3] ; valid
5. int A[2][3]; valid
6. int A[ ] [3]; = {10, 20, 30}: invalid
7. int A[2] [3]; = {1, 2, 3, 4, 5, 6}: valid
8. int A[ ] [3]; = {1, 2, 3, 4, 5, 6}: valid
9. int A[2] [3] [2]; invalid
10. int A[ ] [3] [2]; invalid
11. int A[ ] [2] [ ]; invalid
12. int A[ ] [ ] [2]; invalid
13. int A[2] [3] [2]; = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,}; valid

14. int A[ ] [3] [2]; = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,}; valid
15. int A[2] [ ] [2]; = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,}; invalid
16. int A[2] [3] [ ]; = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,}; invalid

Note:
- while declaring an array, it is mandatory to provide the size of each dimension. (3), (6), (10), (11), (12) are not providing sizes of dimensions.
- While initializing an array, you have flexibility to omit only first dimension i.e., you are allowed to other dimension (1), (8),(14) are following this rule while (15), (16) are not following this rule.

4.1.9 Pointers

- In C, there is a strong relationship between arrays and pointers. An operation that can be achieved by arrays subscripting also be done with pointes.
- Consider the declaration
  int a[5] = {10, 20, 30, 40, 50};
  int* Pa;
  Pa = &a[0];
- Pointers are special variable that can hold address of other variables
  Syntax: data type * Identifier
  int *Pa: Pa is a pointer to integer variable
  i.e., Pa can hold address of integer variable



Pa contains address of a[0]

- Two operators are important to understand for understanding pointers: &, *
  i. &: address of operator
  ii. *: value at operator
- Pa is equivalent address 1020
- Pa is equivalent to value at (Memory location 1020)
  i,e *Pa is same as 10
- Pa is pointing to same element of array, then by definition Pa +1 points to next element, Pa + i points to elements after i elements before Pa.
  i.e, In our example Pa points to a[0]
  So, Pa+1 will point [1]
  Pa+2 will point to a[2]
  i,e
  Pa+i is address of a[i]
  $\Rightarrow$ Pa + i ≡ & a[i]
  $\Rightarrow$ *(Pa+i) ≡ *& a[i] ≡ a [i]
- Array name always represent address of the very first element
  i.e., Pa = &a[0];
  and

Pa = a ;

both are same.

- a[i] ≡ *(a+i) ≡ *(i + a) ≡ i [a]
  All are same and can be used interchangeably in program except in declaration.
- In declaration, we can not write
  int 3[a]; instead of int a[3]
- Array name represent a constant address.
  int a[10];

i. a++, ++a, – –a , a – – are all invalid as we cannot apply increment/decrement operator on constants:

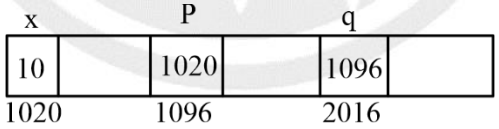
ii. Array name cannot be Lvalue of on assignment statement.
i.e.
array_name = any expression/address/value is invalid

- On the other hand, pointer being a variable, the following operations are valid.

```
int a[5]
int = *Pa;
Pa = a : valid
Pa+ + ;
Pa – –:
+ + Pa :
– – Pa :
```
} *all are valid*

- Multi-level pointer

```
int x      :   x can store value
int *P     :   p can store address of integer variable
int **q    :   q can store address of pointer variable
x = 10     :   valid
p = &x     :   valid
q = &p     :   valid
```



p ≡ 1020 i, e address of x
*P ≡ value at (memory location 1020)
*p ≡ 10
q ≡ 1096 i,e address of variable p
*q ≡ value at (memory location 1096)
≡ 1020 which is again an address
*q ≡ value at (memory address 1020)
**q ≡ 10

1. **Dangling pointer:** The pointer pointing to a deallocated memory block is known as dangling pointer.
   This situation raises an error known as dangling pointer problem. Dangling pointer occurs when a pointer pointing to a variable goes out of scope or when a variables memory gets deallocated.
   Consider the code.
   int *f( ) {

```
int a = 10; // a is a local variable and goes out of scope after execution of f( )
return &a :
}
int main () {
int * ptr = f (); //Ptr points to something which is not valid
print ("/d"*,ptr);
return 0;
}
```

- To overcome this problem, just make variable a as static when x become static it has scope throughout the program.

2. **Uninitialized pointer:** An uninitialized pointer also known as wild pointer
   A pointer which has not been initialized to anything can be dangerous

- The value saved in an uninitialized pointer could be randomly pointing anywhere in memory
- int *p

- int *p

- Storing a value using an uninitialized pointer has the potential to overwrite anything in your program including your program itself.
- System may crash.
- Always initialize with NULL.

3. **NULL pointer:** It is a pointer which is pointing to nothing It is a specially designed pointer that stores a defined value but not a valid address of any element or object.
   NULL pointer does not hold valid address and that is why if you try to dereference it, you will get an error

```
int *p = NULL:
printf ("%d",*p) :        //NULL pointer dereferencing
```

4. **void pointer**: void pointer is a pointer that points to some location in memory but is does not have any specific type.
   It can point to any type of data and any pointer type is convertible to a void pointer.
   Its declaration:

```
void *ptr :
```

   is saying that ptr is a pointer that can hold an address
   cannot be dereferenced directly
   i.e.,

```
void *ptr
int x = 10:
```

```
ptr = &x :
printf ("%d", *ptr);        //invalid
```

you need to typecast the void pointer before dereferencing.

i.e.,

```
void *ptr
int x = 10;
ptr = &x;
printf("%d",*(int*)ptr);
                 └→typecasting
```

Here, (int*)ptr does type costing of void

(int) ptr dereferences the typecasted void pointer variable

- Pointer arithmetic in not possible on void pointer.
  i,e……. ptr+ +, + + ptr, – –ptr, ptr – –, ptr +1, ptr -1 is not allowed an void pointer.

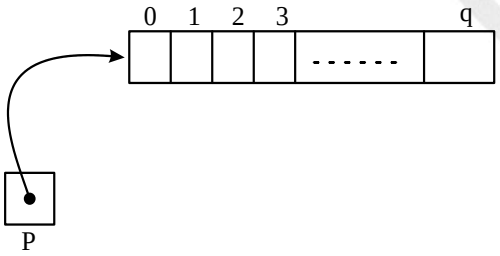
- An uninitialized pointer holds a garbage value while a NULL pointer holds a defined value but not a valid address.

4.1.10 Memory leakage problem

- Whenever a pragrammer allocated memory dynamically then it is the reponsibility of the programmer to free that memory after usage.
- Memory leakage problem occurs when a programmer allocates memory dynamically but does not de-allocate it after using it.
- It reduces the performance of the computer by reducing the amount of availbale memory.

4.1.11 Understanding Declartion

1. int *(P[10]) : P is an array of 10 pointer to integer
2. int (*P)[10] : P is a pointer to an array of 10 integers



3. int (*P)() : P is pointer to a function that takes no argument and returns an integer.
4. int (*P) (int, int) : P is a pointer to a function that takes 2 integer arguments and returns an integer.
5. int *P (char*) : P is a pointer to a function that takes a pointer to character argument and return a pointer to integer.

4.1.12 Pointer to Functions / Function Pointer

- Just like normal pointers we can have pointers to functions.

```
int(*p)(int, int)
```

P is a pointer to a function that take 2 integer arguments and returns an integer value.

- **Declaration of function pointer:** declaring a pointer to function is almost same as declaring the fuction except that in function pointer are prefix is with an asterik * symbol.
  For example: if the function is

  int add(int, int)

  Declaration of a fuction pointer for add() fuction is :

  int (*ptr)(int, int);
- How to call a function through function pointer : In order to all a function using function pointer, first we need to assign the address of function code to the pointer

Ex 1:

```
int add (int, int):
void main () {
        int (*ptr) (int, int);
        ptr = &add;       // ptr is a pointer to add() function
printf ("The sum of 10 and 20 is", (*ptr )(10, 20)); //calling add()
}
int add (int, int);
```

Ex 2:

```
void main() {
        int (*ptr) (int, int);
        ptr =  add ; // same as ptr = & add
printf (" The  sum of  10 and 20 is", (*ptr) (10, 20));
```

- A function can be called using following 4 ways using pointer to fuction.

int add (int int);
void main() {
int (*ptr) (int, int);
ptr = &add;
printf ("%d", (*ptr)(10, 20));
}
	int add (int int);
	void main() {
	int (*ptr) (int, int):
	ptr = add;
	printf ("%d", (*ptr) (10, 20));
	}

int add (int, int) ;
void main() {
int(*ptr) (int, int);
prt= &add;
printf ("%d", (*ptr)(10, 20));
}
	int add (int, int);
	void main() {
	int (*ptr) (int, int);
	prt= add;
	printf ("%d", (*ptr) (10, 20));
	}

- Using function poitnter we are able to pass a function as an argurment to other function and can also be returned from function.
  Array of function pointer
  int (*ptr[3])(int, int)
  Ptr is an array of 3 pointers to a function that takes 2 arguments and returns an integer.

❐❐❐

5 STRINGS

5.1 Strings

- Sequence of characters terminated by a null character '\0'.
  Syntax:
  char str_name[size];
- Initializing a string in c
  - char str[] = "Pankaj";
    Here, str is internally a pointer (holds the address of first character)
  - char str[20] = "Pankaj";
    Here, we predefine the size as 20 but we should always take one extra space along with size of string.
    i.e atleast 7 size must be there to store "Pankaj" (6+1)
  - char str[7] = {'P', 'a', 'n', 'k', 'a', 'j', '\0'};
    must set the end character as '\0'
  - chat str[] = {'p', 'a', 'n', 'k', 'a', 'j', '\0'};

5.1.1 Reading a string

- **scanf():**
  when we use scanf() to read, we use %s as a format specifier without using "&" to access the variable address because an array name acts as a pointer.

```
#include <stdio.h>
    int main (){
        char name [20];
        printf("enter your name \n");
        scanf("%s",name);
        printf("Hello %s",name);
          }
```

Output:

```
Enter your name
Pankaj Sharma
Hello Pankaj      // why not Hello Pankaj Sharma
```

The above code did not print Hello Pankaj Sharma as expected because scanf halts reading as soon as a whitespace or a newline character is encountered, and that is why it reads only Pankaj.

- In order to read a string containing space, we use gets() function. gets() ignores the whitespace & stops reading when a newline is found.

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

```
void main(){
        char name [20];
        printf("Enter your name :");
        gets(name);
        printf("Hello %s", name);
        }
Output :
        Enter your name: Pankaj Sharma
        Hello Pankaj Sharma
```

- A string literal always represent base address of a string. i.e address of first character.
- "Pankaj" represents address of character 'P'

Hence "Pankaj" [0] means 'P'

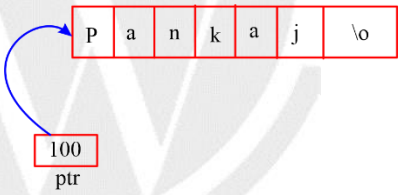
"Pankaj" [1] ≡ *("Pankaj"+1) ≡ *(Address of 'P'+1)
≡ *(Address of 'a')
≡ 'a'

5.2 Strings and pointers

- Array name represents the address of its first element. Similar to character arrays, we can create a character pointer to represent a string that will hold the starting address i.e address of first character of string.

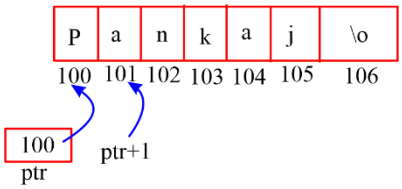
Example - 1

```
# include<stdio.h>
void main(){
        char *ptr = "pankaj"
        printf("%s",ptr);
        }
Output : pankaj
```



Example – 2

```
# include<stdio.h>
void main(){
        char *ptr = "Pankaj"
        printf("%s",ptr+1);
        }
Output: ankaj
```



5.3 Predefined functions for string

(1) strlen (): returns the length of the string passed as an argument.

(2) strcpy (): copies the contents of one string to another. strcpy (S_1,S_2) copies the content of string S_2 to string S_1.

(3) strcmp (): compares the first string with the second string. If both are same it returns 0.
strcmp (str1,str2) returns 0 if both strings are same.

(4) strcat (S_1,S_2): concat S_1 string with S_2 string and the result i.e concatenation of S_1,S_2 is stored in S_1.

5.4 Important concepts

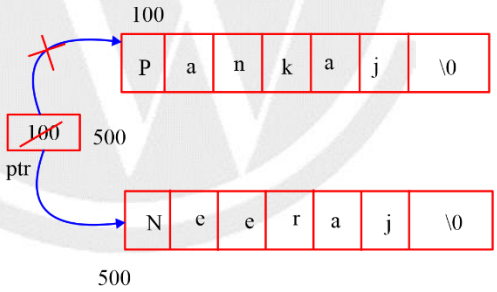
(1)
```
# include<stdio.h>
void main(){
        char name [20] = "Pankaj" ;
        name = "Neeraj" ;              // Error
        printf("%s", name);
                }
```

- Array name being a constant can't be presented at the left side of the assignment statement i.e it can't be L-value. You can't re-assign another string to array.

(2)
```
# include<stdio.h>
void main(){
                char name [20] = "Pankaj" ;
                name [1] = 'u' ;
                printf("%s", name);
                }
```
Output: Punkaj

- We are allowed to change content of array and hence we can update any character.

(3)
```
# include<stdio.h>
void main(){
                char * ptr = "Pankaj" ;
                ptr = "Neeraj" ;
                printf("%s", ptr);
                }
```
Output: Neeraj

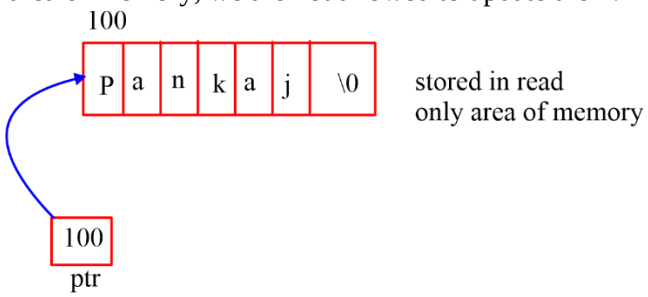


- Pointer ptr being a variable can hold different address at different time. Hence, we can assign another string to a character pointer any time.

(4)
```
# include<stdio.h>
void main()
        {
        char * ptr = "pankaj" ;
        ptr [1] = 'u' ;  // Invalid
        printf("%s", ptr);
        }
```

- String literals are stored in read only memory area i.e "pankaj" is stored first & then the address of character 'p' is given to ptr.

- As they are stored in read only area of memory, we are not allowed to update them.



❐❐❐

6 TYPES OF DATA STRUCTURE, ARRAY & LINKED LIST

6.1 Introduction

6.1.1 Linear data structure

Every element can have almost 2 neighbours. Ex. arrays, linked list, stack, queue.

6.1.2 Non-Linear data structure

Element can have more than 2 neighbours. Ex. Tree, graph.

6.2 Arrays

6.2.1 1-D array

Theoretically index can start from any integer value. Let A be a 1-D array of n elements and the size of each element is w bytes, then the address of A[i] element.

$$\text{Base Address}\,(A) + (i - \text{starting index}) * w$$

6.2.2 2-D Array

Analogous to matrix with rows and columns. Can be implemented in RMO (Row-major order) or CMO (Column-major order).

Let A[M] [N] be a 2-D array, where size of each element is w-bytes, then the address of A[i] [j] is :

(a) In RMO

Address (A[i] [j]) = Base Address (A) + [(j-starting index) + (i – starting index) × M]xw

(b) In CMO

Base Address (A) + [(i – starting index) + (j – starting index) × M] × w

6.2.3 Sparse Matrices

Most of the element of the matrix have 0 value and representing such matrix by a 2D array leads to wastage of memory and that is why, it is better than we will only store non-zero elements (Efficient method).

6.2.4 Lower Triangular Matrix

$n \times n$

$A_{ij} = 0 \quad \text{if} \quad i < j$

$A_{ij} \neq 0 \quad \text{if} \quad i > j$

For ex.

$$A=\begin{bmatrix} A_{11} & 0 & 0 & 0 \\ A_{21} & A_{22} & 0 & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

6.2.5 Upper triangular Matrix

$$A=\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ 0 & A_{22} & A_{23} & A_{24} \\ 0 & 0 & A_{33} & A_{34} \\ 0 & 0 & 0 & A_{44} \end{bmatrix}$$

6.2.6 Tri-diagonal Matrix

Matrix that has non-zero elements only in the main diagonal, diagonal just below main diagonal and diagonal just above main diagonal. All other elements are zero.

$$\begin{bmatrix} A_{11} & A_{12} & 0 & 0 & 0 \\ A_{21} & A_{22} & A_{23} & 0 & 0 \\ 0 & A_{32} & A_{33} & A_{34} & 0 \\ 0 & 0 & A_{43} & A_{44} & A_{45} \\ 0 & 0 & 0 & A_{54} & A_{55} \end{bmatrix}$$

6.2.7 Efficient method to store sparse matrix

Using Row-major order for string lower triangular matrix, the element at index (i, j) can be represented as :

$$\text{Index of } A_{ij}=\left[\frac{i(i-1)}{2}+(j-1)\right]$$

Using column-major order for storing upper triangular matrix, the element at index (i, j) is sparse matrix can be represented as:

$$\text{Index of } A_{ij}=(i-j)+\left[(j-1)N-\frac{(j-i)(i-2)}{2}\right]$$

Matrix order

N × N

Using Row-major order for storing upper triangular matrix, the element at index (i, j) in sparse matrix can be represented as :

$$\text{Index of } A_{ij}=(j-i)+\left[(i-1)N-\frac{(i-i)(i-2)}{2}\right]$$

Where N : Number of rows/columns

Using column-major order for starting upper-triangular matrix the element at index (i, j) can be represented as

$$\text{Index of } A_{ij} = \left[(i-1) + \frac{j(j-1)}{2} \right]$$

Using Row-major order for storing tri-diagonal matrix,

Index of element A_{ij} = $2i + j - 3$

Using column-major order for storing tri-diagonal matrix,

Index of element A_{ij} = $i + 2j - 3$

6.3 Linked List

6.3.1 Linked List

Linked list is collection of elements called node, where each node contains atleast 2 fields.

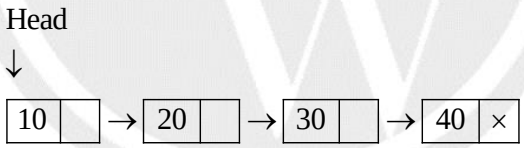
(i) Data field : that contains information.

(ii) Link field : contains address of next note.

6.4 Types of Linked List

6.4.1 Singly Linked List

Every node contains data and a pointer to the next node in the linked list, we can traverse the linked list only in forward direction.



- Head : Pointer that contains address of 1st node.
- Head contains NULL represent empty Linked List.
- If Ptr contains address of some note (Ptr is a pointer to node) then to go to the next node, we need.

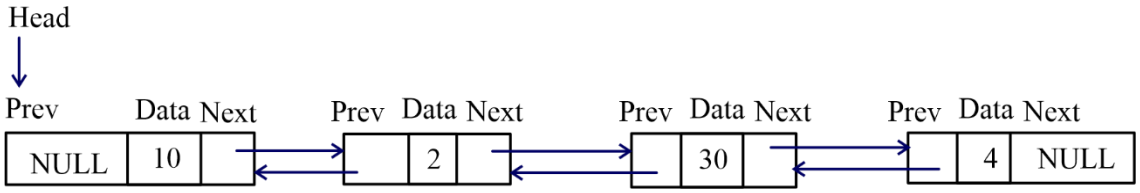
$$\text{Ptr} = \text{Ptr} \rightarrow \text{Link}$$

- Node can be implemented by structure in C

```
Struct Node {
        int data;
Struct Node    * Link;
        }
```

6.4.2 Doubly Linked List

A two-way linked list in which every node contains a pointer to the next node as well as pointer to previous node in the list we can traverse it in backward direction also.



Structure of a Node :

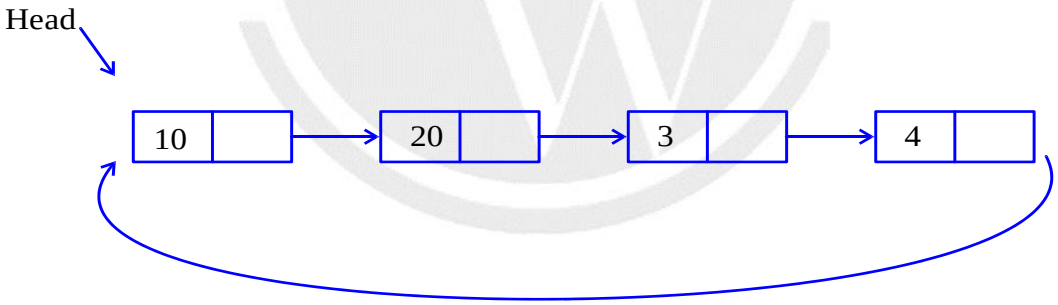
```
Struct Node {
        struct Node * Prev;
        int data;
        struct node * next;

        };
```

6.4.3 Circular Linked List

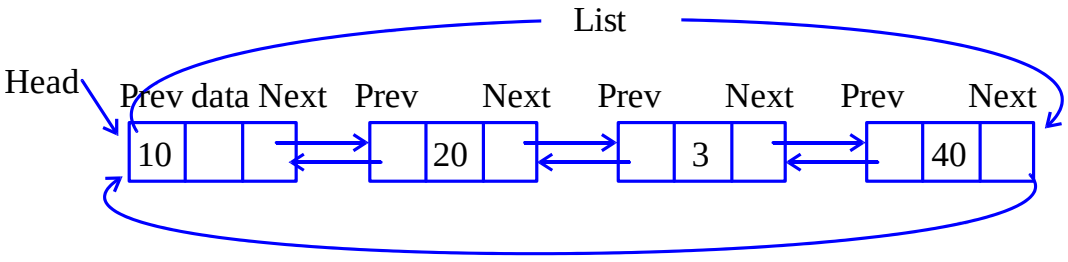
A linked list in which last node points back to the first node in the list.

- Circular linked list has no beginning and no end.



6.4.4 Doubly circular linked list

It is a 2-way circular linked list.

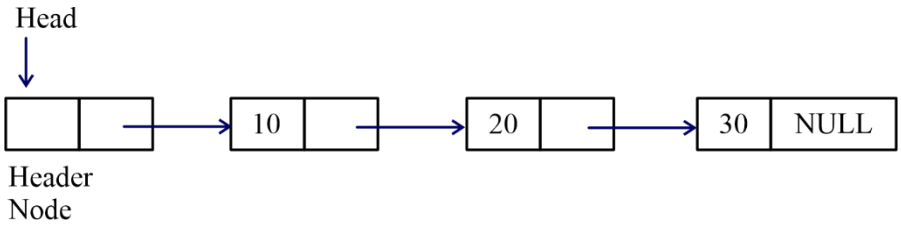


6.4.5 Header Linked List

A linked list that contains a special node called header node at the beginning of the list.

- Head will not point to first node.

- Head points to the header node.



6.4.6 Applications

- used to implement other data structure link stack, queue data structure.
- used for dynamic memory allocation.

6.4.7 Advantages and Disadvantage

- Insertion and deletion are efficient.
- Using pointers in every node, require more memory.
- Random accessing in not possible.
- Circular linked list can be used to implement Fibonacci Heap.

❑❑❑

7 STACK AND QUEUE

7.1 Introduction

Linear data structure in which both insertion and deletion operation are performed at one end called TOP of the stack. Works on LIFO (Last In First Out) Policy.

7.1.1 Application

To post-pone certain decision (To wait/To delay)

- **Stack Permutation :** Order of insertion of key is fixed but we can pop an element any time.

  Number of possible stack permutation with n elements $= \frac{^{2n}C_n}{n+1}$

- Infix to postfix
- Infix to prefix
- Prefix Evaluation
- Postfix Evaluation
- Recursion
- Tower of Hanoi

7.2 Queue Data Structure

Linear data structure in which insertion is done at one end called rear of the queue and deletion is performed at other end called front end of the queue.

Works on FIFO (First In First Out) policy.

7.2.1 Applications of Queue Data Structure

- Whenever a resource is being shared among many users.
- When data is transferred asynchronously.
- FCFS scheduling
- Algorithms like BFS.

7.3 Implementation

Can be implemented using arrays, linked list.

(a) Using Array :

- Easy to implement and memory efficient.
- Not dyanmic

(b) Using Linked List :

- Dynamic
- Require extra memory because of pointer field.

7.4 Standard Operations

7.4.1 Stack

(a) Push ( ) : Insertion of an element onto stack.

(b) Pop ( ) : Deletion of an element.

(c) Is_empty ( ) : Returns true if stack is empty otherwise return false.

7.4.2 Queue

(a) Enqueue : Insertion of an element.

Performed at rear end of the queue.

(b) Dequeue : Deletion of an element.

Performed at front end of the queue.

7.5 Circular Queue

Extended version of simple queue in which last element is connected to the first element.

In simple queue, even though space is available, we are not able to insert a new element and declaring it a overflow. Circular queue solves this problem.

Queue Full

(i) Front = = 0 & & Rear = = SIZE – 1 OR

(ii) Front = = (Rear + 1)% SIZE

7.6 Priority Queue

A queue in which a priority is associated with every element and elements are processed according to their priorities and if two elements have same priority then they are processed as per their arrival in the queue.

❑❑❑

8 TREE DATA STRUCTURE

8.1 Introduction

8.1.1 Terminology

(i) Internal Node : Node with atleast 1 child.

(ii) Leaf Node : Node without any child.

(iii) Root Node : Only node that does not have any parent.

(iv) Complete binary Tree : Binary tree in which nodes are inserted from left to right at every level and we can not insert at node at (K+ 1)th level until all levels upto K are fully filled.

(v) Full Binary Tree : A binary tree in which every internal node has exactly two children and all the leaf nodes are at some level.

Binary tree that contains maximum number of nodes.

(vi) Strict Binary Tree : A binary tree in which every internal node has exactly two children.

(vii) Complete K-ary tree : Tree in which every internal node has exactly K-childs.

8.1.2 Mathematical Result

(i) For a binary tree of height h

- Maximum number of nodes = $2^{(h+1)} - 1$
- Minimum number of nodes = h + 1

(ii) For a complete binary tree of height h

- Maximum number of nodes = $2^{(h+1)} - 1$
- Minimum number of nodes = 2^h

(iii) For a full binary tree :

Number of nodes = $2^{h+1} - 1$

(iv) Total number of unlabelled binary trees with n nodes = $\frac{{}^{2n}C_n}{n+1}$

(v) Total number of labelled binary tree with no keys = $\frac{{}^{2n}C_n}{n+1} \times n!$

(vi) Total number of binary trees with a given preorder (n length) = $\frac{^{2n}C_n}{n+1}$

(vii) Total number of binary trees with a given proorder/postorder/inorder = $\frac{^{2n}C_n}{n+1}$

(viii) Number of binary trees with a given preorder and inorder = 1

(ix) Number of binary trees with a given postorder and inorder = 1

(x) Number of leaf nodes = Number of nodes with two children + 1

8.2 Binary Tree Traversal

(1) Preorder :

(i) Process the root.

(ii) Traverse the left subtree of root in preorder.

(iii) Traverse the right subtree of root in preorder.

(2) Inorder

(i) Traverse the left subtree of root in inorder.

(ii) Process the root.

(iii) Traverse the right subtree of root in inorder.

(3) Post-order

(i) Traverse the left subtree of root in post-order.

(ii) Traverse the right subtree of root in post-order.

(iii) Process the root.

In-order

- **Morrison Traversal :**

  In-place traversal of a binary tree

  No recursion

  Constant extra space.

8.3 Binary Search Tree

A binary tree in which every node satisfy the property that all the keys in the left subtree of the node are smaller than node's key and all the keys in the right subtree of the node are greater than node's key.

- Inorder traversal of a binary search tree is ascending order of keys in the BST.
- Insertion in a BST
  (a) O (log n) best case
  (b) O (n) worst case.

- Deletion from a BST :

  (a) Deletion of leaf node : can be done directly

  (b) Deletion of node with one child : copy the child to the node and delete the child.

  (c) Deletion of a node with two children : first find the inorder successor of the node, copy the contents of this successor to the node and dlete the inorder successor.

  We can also use inorder predecessor.

  - TC : O (log n) best case

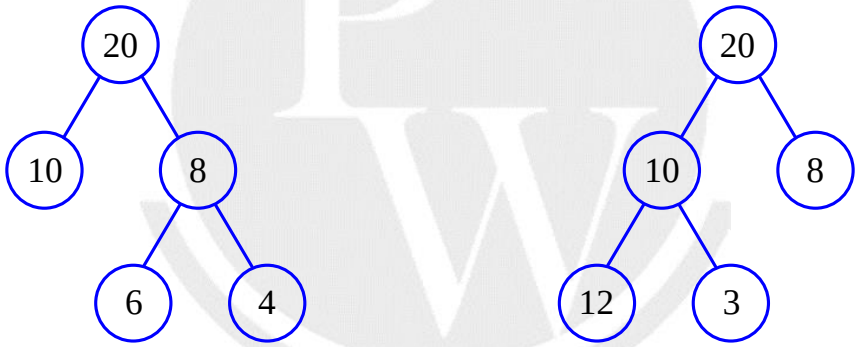
    O (n) worst case

8.4 Heap

A complete binary tree in which every node satisfy heap property (min heap or max heap)

(i) Min heap : A complete binary tree in which every node satisfy the property that the value of the node is smaller than both its children.

(ii) Max heap : A complete binary tree in which every node satisfy the property that the value in the node is greater than both its children.



Not a heap Not a heap

- **Construction of Heap :**

  (i) By inserting keys one after another : O (n log n)

  (ii) Using build-heap method/heapify algo : O(n)

- Number of min-heaps possible with n keys

  $T(n) = T(k) \cdot T(n - k - 1) \cdot {}^{n-1}C_k$

  K : Number of elements in the left subtree of root node.

- Deletion :

  (1) Swap A[1], A[n]

  (2) Apply Heapify on A[1] considering there are only (n – 1) nodes.

  T.C = O (log n)

- Heapsort (Assuming max heap)

  (i) Build Max Heap O(n)

  (ii) Replace root element with last element $\left(A[1] \leftrightarrow A[n]\right)$ reduce the heap size by 1 and then apply heapify at root node.

  (iii) Repeat step 2 while heap size is greater than 1.

8.5 AVL Search Tree

- Height balanced BST.
- Balance factor (B_f) is given by

$$B_f = |h_L - h_R| \leq 1$$

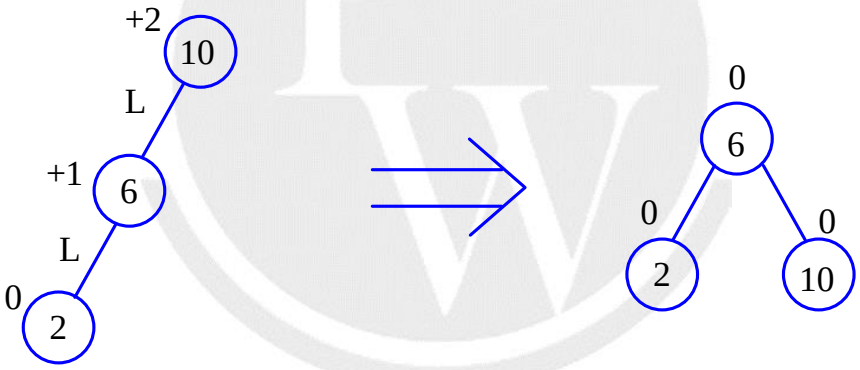
i.e., Balancing factor of a node can be +1, –1 or 0.

8.5.1 Insertion

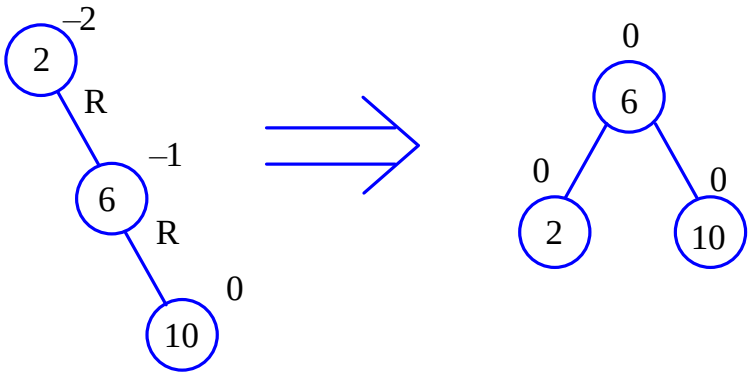
To ensure that the tree remains AVL tree, after insertion some re-balancing is performed i.e., certain rotations are needed.

8.5.2 Types of Rotation

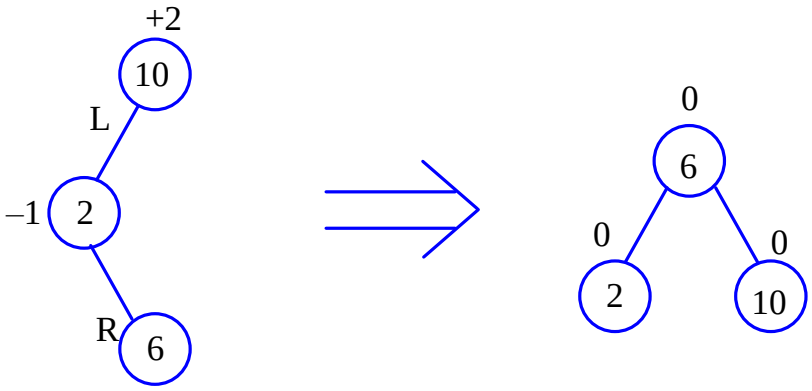
(i) Left-Left (LL) Rotation :



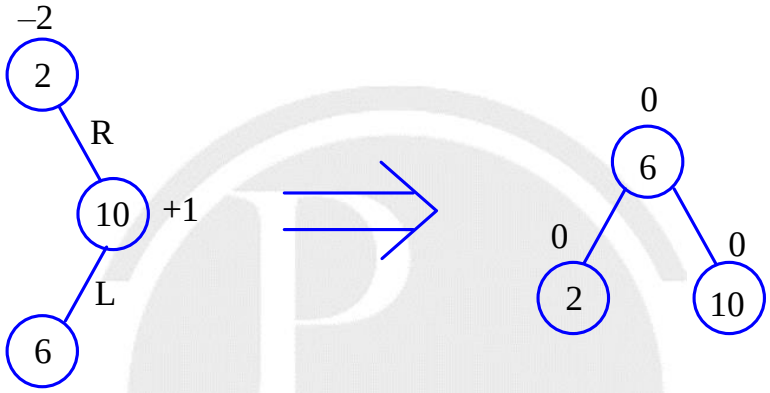
(ii) Right-Right (RR) Rotation :



(iii) Left-Right (LR) Rotation :



(iv) Right-Left (RL) Rotation :



- LL, RR are single rotation.
- LR, RL are double rotation.
- Just work no tri-node structure.
- Minimum number of nodes in an AVL tree of height h is given by recurrence.

  $n(h) = 1 + n(h - 1) + n(h - 2)$ $\quad h \geq 2$

  $n(0) = 1$

  $n(1) = 2$

  i.e., it also forms a Fibonacci series

  1, 2, 4, 7, 12, …….

❑❑❑

09 GRAPH TRAVERSAL

9.1 Introduction

9.1.1 Breadth First Search

- Uses a queue data structure.
- To find whether a graph is connected or not.
- To find number of connected components in a given graph (Breadth First traversal is used)
- To detect cycle in a graph.
- To find whether a given graph is bipartite or not.
- T.C. = O (V + E) using Adjacency List

  = $O(V^2)$ using Adjacency matrix.
- To find shortest path in undirected graph without weights.

9.1.2 Depth First Search

- Uses stack.
- To detect a cycle in a graph.
- To find whether a graph is connected or not.
- To find whether given graph is bipartite or not.
- To find number of connected components in the graph.
- To find bridges in the graph.
- To find articulation points in the graph.
- To find topological sort of a graph.
- To find biconnected components.
- To find strongly connected components.

❑❑❑

10 HASHING

10.1 Introduction

- Searching technique.
- Mapping keys into hash table using a hash function.
- Efficiency based on hash function used for mapping.

10.1.1 Terminology

(i) Collision : When two keys mapped to same location, collision occurs.

$H(k_1) = H(k_2)$, where H(k) is the hash function used.

(ii) Load factor $\lambda = \dfrac{\text{no. of keys}}{\text{Hash table size}}$

10.1.2 Collision resolution technique

(i) Open addressing (closed hashing)

(a) Linear probing
(b) Quadratic probing
(c) Double hashing

$\Big\} \lambda \le 1$

(ii) Separate chaining

10.1.3 Linear Probing

Whenever there is a collision at memory location 'L', then we will search for empty slot sequentially starting from 'L' then L + 1, L + 2, L + 3, …….

10.1.4 Problem with linear Probing

Linear probing suffers with primary clustering problem, consecutive keys forms a cluster and the time to find a free slot increases.

10.1.5 Quatratic Probing

Hash function $h(x) = x$ modm. It leads to a collision and it is i^{th} collision for key x then the collision resolution function is given as:

$H(x, i) = (h(x) + i^2)$ modm

Searching order : L, L + 1, L + 4, L + 9, ……

- Free from primary clustering
- Suffers with secondary clustering problem.

10.1.6 Double Hashing : Using 2 hash functions

(hash – 1 (key) + *i* * hash-2 (key))% m

M : table size

Hash-2 (key) : Secondary hash function must not give output 0.

* One of the best probing.
* Uniform distribution.
* Tree from primary and secondary clustering.
* Computation overhead because of two hash function computation.

10.2 Chaining

Uses the concept of linked list (chain) when more than one element are hashed to same location (slot) then elements are inserted into a singly linked list (chain).

- Deletion is easy compared to open addressing.
- Deleting in open addressing require rehashing remaining keys.

❑❑❑