Computer Science & IT

Programming and Data Structures

Comprehensive Theory

with Solved Examples and Practice Questions

MADE EASY
India's Best Institute for IES, GATE & PSUs

Contents

Programming & Data Structures

■■■■

Programming and Data Structures

Goal of the Subject

Computer Science is not the study of programming. Programming, however, is an important part of what a computer scientist does. Programming is often the way that we create a representation for our solutions. Therefore, this language representation and the process of creating it becomes a fundamental part of the discipline.

A data structure is a specialized format for organizing and storing data. To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the "big picture" without getting lost in the details. By creating models of the problem domain, we are able to utilize a better and more efficient problem-solving process. The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.

General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.

Introduction

In this book we tried to keep the syllabus of Software Programming and Data structures around the GATE syllabus. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into seven chapters as described below.

1. **Programming Methodology:** In this chapter we will study about the different segments and their organization, variables and their scope, flow of control in a program, function evaluation types, storage classes, and pointers and finally we discuss the application of pointers.
2. **Arrays:** In this chapter we will study properties and application of arrays, accessing methods for two and three dimensional arrays and finally we discuss the arrays in the form of special matrices.
3. **Stack:** In this chapter we will study the ADT of stack, operations on stack, applications and different types of notations evaluated by stack and finally we discuss the tower of Hanoi (application).
4. **Queue:** In this chapter we will study about the Queue, operations on queue, applications and finally we discuss different types of queues.
5. **Linked List:** In this chapter we will study types and applications of linked list, operations on linked list, priority queue and finally we discuss implementation of stack, queue and priority queue using lists.
6. **Trees:** In this chapter we introduce trees, their applications, types of trees (BST, B-tree, and AVL), different types tree traversals and finally we discuss operations on trees.
7. **Hashing Techniques:** In this chapter we introduce the Hash function, collision resolution techniques and comparisons of different collision techniques.

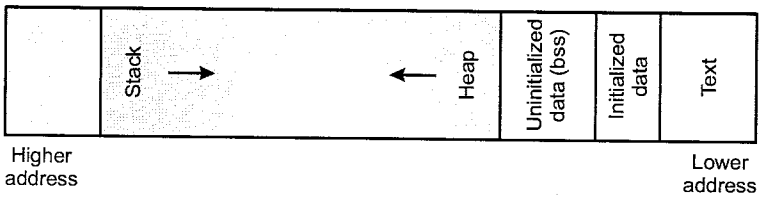
■■■■

CHAPTER 01

Programming Methodology

1.1 Data Segments in Memory

The running program stored the machine instructions (program code) and data in the same memory space. The memory is logically divided into text and data segments. A single text segment is used by modern systems to store program instructions, but to store data need more than one segment, depending upon the storage class of the data being stored there. The below figure shown the different segments in memory:



1. **Text (Code) Segment :** Text segment contains machine code of the compiled program. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors. The text segment of an executable object file is often read-only segment that prevents a program from being accidentally modified.
2. **Initialized Data Segment:** Initialized data segment stores global, static, constant, and variable with extern keywords that are initialized beforehand. Data segment is not read-only, since the values of the variables can be changed at run time.
   This segment can be further classified into initialized read-only area and initialized read-write area.

***Example*-1.1** Consider the following C code:

Code

```
#include<stdio.h>
char c[] = "Madeeasy";          /* global variable stored in initialized data
                                   segment in read-write area */
const char p[]="Madeeasy";      /* global variable stored in initialized data
                                   segment in read-only area */
int main() {
    static int a=11;            /* static variable stored in initialized data
                                   segment */
    return 0;
}
```

3. **Uninitialized Data Segment (bss) :** In this segment data is initialized to (zero) 0 before the program starts executing. Uninitialized data segment starts at the end of the data segment and contains all global and static variables either initialized to (zero) 0 or do not have explicit initialization in program.

Example-1.2 What will be output of following C code?

Code

```
#include<stdio.h>
char c;                          /* Uninitialized variable stored in basic bss */
int main()
{
    static int i;                /* Uninitialized static variable stored in bss */
    return 0;
}
```

4. **Heap:** Heap segment is usually used for dynamic memory allocation. We use malloc and calloc function to allocate memory dynamically, heap grows upward. The Heap segment is shared by all shared libraries and dynamically loaded modules in a program.

Example-1.3 What will be output of following C code?

Code

```
#include<stdio.h>
int main()
{
    char *p = (char*)malloc (sizeof (char));   /* Memory allocating in heap
                                                  segment */
    return 0;
}
```

5. **Stack:** Stack segment is used to store all local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function is over. Local variables have a scope within the defined block, they are created when function call is executed heap grows downward and free the space when execution of function is completed. All recursive function calls are added to stack.

Example-1.4 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i;` `  for(i=0;i<3;i++){` `  static int a=5;` `  printf("%d",a++);` `  }` `  return 0;` `}`	Output: 5 6 7 Static variable scope is limited to the block but its life time is upto program termination. In every iteration of the loop, a retains its previous value. First iteration a=5 Second iteration a=6 Third iteration a=7

1.2 Scope of Variable

In program a scope is the maximal size in which no bindings change. The scope rules of language determine how references to names are associated with variables.

There are two type of scoping techniques: (1) Static scoping and (2) Dynamic Scoping.

Static scoping: Static scoping is defined in terms of the physical structure of the program. The determination of scope is done at a time of compilation i.e. binding are resolved at the time of execution of program. Ada and C++, Pascal, C are the example of static scoping.

Syntax of static scoping

```
for(...){
    int i;
    {...
        {
            j = 5;
        }
    }
    ...
}
```

Here we examine the local scope and static scope until a binding is found.

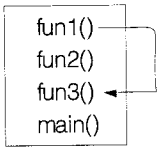
***Example*-1.5** What will be output of following program?

Code	Solution
`int count=50;` `void fun1(){` `  Printf("In fun1=%d", count);` `}` `void fun2(){` `  int count=20;` `  fun1();` `}` `int main(void){` `  fun2();` `  return0;` `}`	Output: 50 In given program with static scoping, the count variable of fun1 takes the value of global variable value (50).

Dynamic Scoping: Dynamic scope rules are usually encountered in interpreted language. Such languages do not normally have type checking at compile time because type determine is not always possible when dynamic scope rules are in effect.

In dynamic scoping, binding depends on the flow of control at run time and the order in which functions are called, refers to the closest active binding. Lisp is the example of dynamic scoping.

Example-1.6 What will be output of following program?

Code	Solution
`int count=50;` `void fun1()` `{` `  Printf("In fun1=%d", count);` `}` `void fun2()` `{` `  fun1();` `}` `void fun3()` `{` `  int count=30;` `  fun2();` `}` `int main(void)` `{` `  fun3();` `  return0;` `}`	Output: 30 The count variable of fun1take the count value of fun3(). 

1.3 C Variable

A variable is named location of data. In other word we can say variable is container of data. In real world you have used various type of containers for specific purpose. For *example* you have used suitcase to store clothes, match box to store match sticks etc. In the same way variables of different data type is used to store different types of data. For *example* integer variables are used to store integers char variables is used to store characters etc. On the basis of type of data a variable will store, we can categorize the all C variable in three groups.

(a) Variables which can store only one data at time.
Example: Integer variables, char variables, pointer variables etc.

(b) Variables which can store more than one data of similar type at a time.
Example: Array variables

(c) Variables, which can store more than one value of dissimilar type at a time.
Example: Structure or union variables.

1.3.1 Properties of C Variable

Every variable in C have three most fundamental attributes. They are Name, Value and Address.

Name of a Variable: Every variable in C has its own name. A variable without any name is not possible in C. Most important properties of variables name are its unique names.

- No two variables in C can have same name with same visibility.
- It is possible that two variable with same name but different visibility. In this case variable name can access only that variable which is more local. In C there is not any way to access global variable if any local variable is present of same name.

Example-1.7 What will be output of following program?

Code	Solution
`#include <stdio.h>` `int main(){` `  auto int a=5;  //Visibility is within main block` `  static int a=10;  //Visibility is within main block` `  /* Two variables of same name */` `  printf("%d",a);` `  return 0;` `}`	Output: compilation error Here variable 'a' is declared twice in main block. So it generate compile time error.

Example-1.8 What will be output of following program?

Code	Solution
`#include <stdio.h>` `int a=50;  //Visibility is in whole program` `int main()` `{  int a=10;  //Visibility within main block` `  printf("%d",a);` `  return 0;` `}`	Output: 10 Here variable 'a' has different visibility at two different places. So printf will print main block variable value.

NOTE: In C any name is called identifier. This name can be variable name, function name, enum constant name, micro constant name, goto label name, any other data type name like structure, union, enum names or typedef name.

1.3.2 Identifier Naming Rule in C

In C any name is called identifier. This name can be variable name, function name, enum constant name, micro constant name, goto label name, any other data type name like structure, union, enum names or typedef name.

Rule 1: Name of identifier includes alphabets, digit and underscore.

Valid name: world, addition23, sum_of_number etc.

Invalid name: factorial#, avg value, display*number etc.

Rule 2: First character of any identifier must be either alphabets or underscore.

Valid name: _calculate, _5,a_, __ etc.

Invalid name: 5_, 10_function, 123 etc.

Rule 3: Name of identifier cannot be any keyword of C program.

Valid name: INT, FLOAT, etc.

Invalid name: int, float, enum etc.

Rule 4: Name of function cannot be global identifier.

Valid name: __TOTAL__, __NAME__ , __TINY__etc.

Invalid name: __TIME__ ,__DATE__ , __FILE__ ,__LINE__ ,__STDC__, __TINY__, __SMALL__, __COMPACT__, __LARGE__, __HUHE__, __CDECL__, __PASCAL__, __MSDOS__, __TURBOC__

Rule 5: Name of identifier cannot be register Pseudo variables

Rule 6: Name of identifier cannot be exactly same as of name of function within the scope of the function.

Rule 7: Name of identifier is case sensitive i.e. num and Num are two different variables.

Rule 8: Only first 32 characters are significant of identifier name.

Example: abcdefghijklmnopqrstuvwxyz123456aaa.

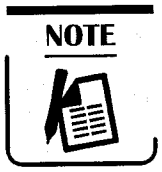
Rule 9: Identifier name cannot be exactly same as constant name which have been declared in header file of C and you have included that header files.

- Variable name cannot be exactly same as function name which have been declared any header file of C and we have included that header file in our program and used that function also in the program.
- Identifier name in C can be exactly same as data type which has been declared in any header of C.

Example-1.9 What will be output of following program?

Code	Solution
`#include <stdio.h>` `int main()` `{` `  int float=5;` `  printf("%d",float);` `return 0;` `}`	Output: compilation error Here we have used float as variable name. So it will give compile time error.

Address of a Variable in C: Location in a memory where a variable stores its data or value is known as address of variable. To know address of any variable C has provided a special unary operator and which is known as dereference operator or address operator. It is only used with variables not with the constant.



- Visibility of variables explained in the following program.

```
#include<stdio.h>
int main(){
    int a=5;
    printf("Address of variable a is: %d",&a);
    return 0;
}
```

We cannot write: &&a, because: &&a=&(&a)=&(address of variable a)=&(a constant number) and we cannot use address operator with constant.

1.3.3 Important Points about Address of Variables in C

1. Address of any variable in C is an unsigned integer. It cannot be a negative number. So in printf statement we should use: %u instead of %d, to print the address of any variable. %d: It is used to. print signed decimal number. %u: It is used to print unsigned decimal number.
   Since, if the address of any variable will beyond the range of signed short int it will print a cyclic value.
2. Address of any variable must be within the range 0000 to FFFF in hexadecimal number format or 0 to 65535 i.e. range of unsigned short int in C. To print the address of any variable in hexadecimal number format by printf function we should use %x or %X.

%x: To print a number in hexadecimal format using 0 to 9 and a, b, c, d, e, f.
%X: To print a number in hexadecimal format using 0 to 9 and A, B, C, D, E, F.

3. A programmer cannot know at which address a variable will store the data. It is decided by compiler or operating system.
4. Any two variables in C cannot have same physical address.
5. Address of any variable in C is not integer type so to assign an address to a integral variable we have to type cast the address.

Example-1.10 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int a=100;` `  unsigned int b=(unsigned)&a;` `  printf("%u",b);` `  return 0;` `}`	Output: Address of the variable a.

1.4 Operators in C

1.4.1 Bitwise Operators in C

In C bitwise operators work at bit-level. They are as follows:

(*i*) **& (bitwise AND):** '&' operator takes two numbers as input as input and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

(*ii*) **| (bitwise OR):** '|' operator takes two numbers as input and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

(*iii*) **∧ (bitwise XOR):** '&' operator takes two numbers as input and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

(*iv*) **<< (left shift):** '<<' operator takes two numbers as input, left shifts the bits of first number, the second number decides the number of places to shift.

(*v*) **>> (right shift):** '<<' operator takes two numbers as input, right shifts the bits of first number, the second number decides the number of places to shift.

(*vi*) **~(bitwise NOT):** '~' operator takes one number as input and inverts all bits of it.

Example-1.11 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `int main()` `{` `  char a = 7, b = 11;` `  printf("a = %d, b = %d\n", a, b);` `  printf("a & b = %d\n", a & b);` `  printf("a|b = %d\n", a|b);` `  printf("a ∧ b = %d\n", a ∧ b);` `  printf("~a = %d\n", a = ~a);` `  printf("a << 1 = %d\n", a << 1);` `  printf("b >> 1 = %d\n", b >> 1);` `  return 0;` `}`	Output : a = 7, b = 11 a & b = 3 a|b = 15 a ∧ b = 12 ~a = 248 a << 1 = 22 b >> 1 = 5

Some properties of bitwise operators are as follows:

1. The **left shift and right shift operators should not be used for negative numbers**. The result of '<<' and '>>' is undefined behaviour if any of the operands is a negative number.
2. If the number is shifted more than the size of integer, the behaviour is undefined. For example, 2 << 33 is undefined if integers are stored using 32 bits.
3. **The bitwise operators should not be used in-place of logical operators.**
   The result of logical operators (&&, II and !) is either 0 or 1, but bitwise operators return an integer value. Also, the logical operators consider any non-zero operand as 1.

***Example*-1.12** What will be output of following C code?

Code	Solution
`int main()` `{ int x = 2, y = 5;` `(x&y)? printf("True "):printf("False ");` `(x&&y)? printf("True "):printf("False ");` `return 0;` `}`	Output: False True

4. The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively but it will works only if numbers are positive.

***Example*-1.13** What will be output of following C code?

Code	Solution
`int main()` `{ int x = 21;` `printf ("x << 1 = %d\n", x << 1);` `printf ("x >> 1 = %d\n", x >> 1);` `return 0;` `}`	**Output :** 42 10

5. **The & operator can be used to quickly check if a number is odd or even**
   The value of expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.

***Example*-1.14** What will be output of following C code?

Code	Solution
`int main()` `{ int x = 21;` `(x & 1)? printf("Odd"): printf("Even");` `return 0;` `}`	**Output:** Odd

Properties of Operator Precedence and Associativity in C

Precedence: In an expression there are multiple operators. Precedence plays most important role, in order to decide the evaluation of there operators. ***Example:*** $1+2\times3$ is calculated as $1+(2\times3)$ and not as $(1+2)\times3$.

Associativity: In an expression some times two or more operators of same precedence appear, order of evalution is decided by associativity. Associativity can be either Left to Right or Right to Left. **Example** '×' and '/' have same precedence and their associativity is Left to Right, so the expression "1000/100 × 10" is treated as "(1000/100) × 10".

1. Associativity is compiler dependent.

Code	
`int x = 0;` `int main()` `{ int p = f1() + f2();` `printf("%d ", x);` `return 0;` `}`	`int f1()` `{ x = 5;` `return x;` `}` `int f2()` `{ x = 10;` `return x;` `}`

2. **Precedence and associativity of postfix ++ and prefix ++ are different:**
   Precedence of postfix ++ is more than prefix ++, their associativity is also different. Associativity of postfix ++ is left to right and associativity of prefix ++ is right to left.
3. Comma has the least precedence among all operators.

***Example*-1.15** What will be output of following C code?

Code	Solution
`#include<stdio.h>` `int main()` `{ int a;` `a = 1, 2, 3;` `printf("%d", a);` `return 0;` `}`	Output : Evaluated as (a = 1), 2, 3

Operators Precedence Table

S. No.	Operator	Associativity
1.	(,), [,] , →, ·	Left to right
2.	!, ~, ++, – –, + (unary), – (unary), *, &, sizeof	Right to left
3.	*, / , %	Left to right
4.	+ , –	Right to left
5.	<< , >>	Left to right
6.	<, <=, >, >=	Right to left
7.	==, !=	Left to right
8.	&	Right to left
9.	^	Left to right
10.	|	Right to left
11.	&&	Left to right
12.	||	Right to left
13.	? :	Left to right
14.	=, +=, –=, *=, /=, %=, &=, ^=, |=, <<=, >>=	Right to left
15.	,	Left to right

(Highest) ↕ (Lowest)

1.4.2 Arithmetic Operators

Arithmetic Operators perform arithmetic operations on operands.

- **Addition:** The '+' operator does addition of two operands i.e., x + y.
- **Subtraction:** The '–' operator does subtraction of two operands i.e., x – y.
- **Multiplication:** The '*' operator does multiplication of two operands i.e., x × y.
- **Division:** The '/' operator does division of the first operand by the second i.e., x/y.
- **Modulus:** The '%' operator returns the remainder when first operand is divided by the second i.e., x%y.

***Example*-1.16** What will be output of following C code?

Code

```
#include<stdio.h>
int main()
{  int a = 10, b = 4, op;
   //printing a and b
   printf("a is %d and b is %d\n", a, b);
   op = a+b; printf("a+b is %d\n", op);
   op = a-b; printf("a-b is %d\n", op);
   op = a*b; printf("a*b is %d\n", op);
   op = a/b; printf("a/b is %d\n", op);
   op = a%b; printf("a%b is %d\n", op);
   return 0;
}
```

Solution

Output :

a is 10 and b is 4
a + b is 14
a – b is 6
a*b is 40
a/b is 2
a%b is 2

Unary Arithmetic Operators

- **Increment:** The '++' operator is used to increment the value of an integer. When placed before the variable name called **pre-increment** i.e., ++ a (value increment and update immediately). When it is placed after the variable name called **post-increment** i.e., a++. Its value is preserved temporarily until the execution of this statement and it gets updated before the execution of the next statement.
- **Decrement:** The '––' operator is used to decrement the value of an integer. When placed before the variable name called **pre-decrement** i.e., –– a (value decremented and update immediately). When it is placed after the variable name called **post-decrement** i.e., a ––. Its value is preserved temporarily until the execution of this statement and it gets updated before the execution of the next statement.

***Example*-1.17** What will be output of following C code?

Code

```
#include<stdio.h>
int main()
{  int a = 10, op;
   op = a++;
   printf("a is %d and op is %d\n", a, op);
   op = a--;
   printf("a is %d and op is %d\n", a, op);
   op = ++a;
   printf("a is %d and op is %d\n", a, op);
   op = --a;
   printf("a is %d and op is %d\n", a, op);
   return 0;
}
```

Solution

Output :

a is 11 and op is 10
a is 10 and op is 11
a is 11 and op is 11
a is 10 and op is 10

1.4.3 Relational Operators

Relational operators are used for comparison of two values.

(*i*) '==' operator checks whether the two given operands are equal or not. If so, it returns true. Otherwise it returns false i.e., 5 == 5 will return true. But 5 == 6 returns false.

(*ii*) '!=' operator checks whether the two given operands are equal or not. If not, it returns true. Otherwise it returns false i.e., 5! = 5 will return false. But 5! = 6 returns true.

(*iii*) '>' operator checks whether the first operand is greater than the second operand. If so, it returns true. Otherwise it returns false i.e., 11 > 10 will return true. But 9 > 10 returns false.

(*iv*) '<' operator checks whether the first operand is lesser than the second operand. If so, it returns true. Otherwise it returns false i.e., 11 < 10 will return false. But 9 < 10 returns true.

(*v*) '>=' operator checks whether the first operand is greater than or equal to the second operand. If so, it returns true. Otherwise it returns false i.e., 10 >= 10 will return true. But 9 >= 10 returns false.

(*vi*) '<=' operator checks whether the first operand is lesser than or equal to the second operand. If so, it returns true. Otherwise it returns false i.e., 10 <= 10 will also return true. But 11 <= 10 returns false.

Example-1.18 What will be output of following C code?

Code

```
#include <stdio.h>
int main()
{
   int a=6, b=5;
   if (a > b)
      printf("a is greater than b\n");
   else printf("a is less than or equal to b\n");
   if (a >= b)
      printf("a is greater than or equal to b\n");
   else printf("a is lesser than b\n");
   if (a < b)
      printf("a is less than b\n");
   else printf("a is greater than or equal to b\n");
   if (a <= b)
      printf("a is lesser than or equal to b\n");
   else printf("a is greater than b\n");
   if (a == b)
      printf("a is equal to b\n");
   else printf("a and b are not equal\n'');
   if (a! = b)
      printf("a is not equal to b\n");
   else printf("a is equal b\n");
   return 0;
}
```

Solution

Output :

a is greater than b
a is greater than or equal to b
a is greater than or equal to b
a is greater than b
a and b are not equal
a is not equal to b

1.4.4 Logical Operators

They are used to combine two or more conditions and produce output in form of true (1) and false (0).

(*i*) Logical AND: The '&&' operator returns true when both the conditions are satisfied. Otherwise it returns false i.e., a && b returns true when both a and b are true.

(*ii*) Logical OR: The 'll' operator returns true when one (or both) of the conditions is satisfied. Otherwise it returns false i.e., a || b returns true if one of a or b is true. It returns true when both a and b are true.

(*iii*) Logical NOT: The '!' operator returns true if the condition is not satisfied. Otherwise it returns false i.e., !a returns true if a is false, i.e. when a = 0.

Example-1.19 What will be output of following C code?

Code	Solution
`#include <stdio.h>` `int main()` `{ int a = 5, b = 2, c = 5, d = 10;` `if (a>b && c == d)` `printf("a is greater than b AND c is equal to d\n");` `else printf("Logical AND condition not satisfied\n");` `if (a>b || c==d)` `printf("a is greater than b Logical OR c is equal to d\n");` `else printf("Neither a is greater than b nor c is equal " " to d\n");` `if (!a)` `printf("a is zero\n");` `else printf("a is not zero");` `return 0;` `}`	**Output** Logical AND condition not satisfied a is greater than b Logical OR c is equal to d a is not zero.

Short-Circuiting in Logical Operators

In case of **logical AND**, the second operand is not evaluated if first operand is false.

Example-1.20 What will be output of following C code?

Code	Solution
`#include <stdio.h>` `#include <stdbool.h>` `int main()` `{ int a = 5, b = 4;` `bool op = ((a == b) && printf("MADEEASY"));` `return 0;` `}`	**Output :** bool op will return 0 because of (a == b) is false. So printf statement is not evaluated.
`#include <stdio.h>` `#include <stdbool.h>` `int main()` `{ int a = 5, b = 4;` `bool op = ((a! = b) && printf("MADEEASY"));` `return 0;` `}`	**Output :** bool op will return 1 because of (a != b) is true. So printf statement is evaluated.

In case of **logical OR**, the second operand is not evaluated if first operand is true.

Example-1.21 What will be output of following C code?

Code	Solution
`#include <stdio.h>` `#include <stdbool.h>` `int main()` `{ int a = 5, b = 4;` `bool op = ((a! = b)|| printf("MADEEASY"));` `return 0;` `}`	**Output :** bool op will return 1 because of (a != b) is true. So printf statement is not evaluated.
`#include <stdio.h>` `#include <stdbool.h>` `int main()` `{ int a = 5, b = 4;` `bool op = ((a == b)|| printf("MADEEASY"));` `return 0;` `}`	**Output :** bool op will return 0 because of (a != b) is true. So printf statement is evaluated.

1.5 Address arithmetic in C

1. We can subtract address of any two variables.
2. We cannot add, multiply, divide two addresses.
3. We can add or subtract a integer number with address.
4. Other operators which can be used with addresses are:
   (a) Negation operator: !
   (b) Size operator: sizeof
   (c) Type casting operator: (Type)
   (d) Deference operator: *
   (e) Logical operator: &&, ||

Example-1.22 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `int a,b;` `printf("%d",&b-&a);` `return 0;` `}`	Output: Suppose address of b=1000, a=500 According to the solution 1000 – 500 = 500

1.6 Value of Variable in C Language

The data that a variable has is known as value of variable.

Example: int a=5; here value of variable a is 5.

By the name of variable value of the variable can be returned. C supports eleven type of assignment operator to assign any value to operator.

Those are: (a) =, (b) +=, (c) –=, (d) *=, (e) /=, (f) %=, (g) <<=, (h) >>=, (i) |=, (j) &= and (k) ^=.

1.6.1 LValue

Lvalue stands for left value. In any assignment statement LValue must be a container i.e. which have ability to hold the data. In C has only one type container and which is variable. Hence LValue must be any variable in C it cannot be a constant, function or any other data type of C.

```
#define<stdio.h>
#define max 125
struct abc{
    char *name;
    int roll;
};
enum {RED,BLUE};
int main(){
    int a, const b;
    a=5;
    //10=5; LValue cannot be an integer constant
    //max=20; //Lvalue cannot be a macro constant
    //b=11; Lvalue cannot be a constant variable
    //abc={"sachin",5}; Lvalue cannot be a data type
    //BLUE =2; Lvalue cannot be an enum constant
    return 0;
}
```

1.6.2 RValue

In any assignment statement RValue must be anything which can return and constant value or it is itself a constant. RValue can be any C constants, variables, function which is returning a value etc.

```
#define<stdio.h>
#define max 5
void display();
float sum();
enum {BLACK, WHITE};
int main(){
   int x=2;          //RValue can be a constant.
   float y=9.0;
   const int z=x;  //RValue can be a constant variable
   x=max;            //RValue can be a variable.
   x=BLACK;          //RValue can be a enum constant.
   y=sum();          //RValue can be a function.
   y=display();      //RValue can be a function which return 0;
}
```

1.7 Flow Control in C

Looping is the process of repeating of same code until a specific condition doesn't satisfy. In C there are three types of loop: (a) for loop, (b) while loop and (c) do while.

1.7.1 for Loop

This loop is used when we have to execute a part of code in finite times. It is per tested loop.

Syntax of for loop:

for (Expression 1; Expression 2; Expression 3) {

    Loop body

}

Order of movement of control in for loop:

First time: Expression 1→ Expression 2 → Loop body → Expression 3.

Second time and onward: Expression 2 → Loop body → Expression 3.

Expression 1: Only executes in the first iteration. From second iteration and onward control doesn't go to the Expression 1.

Example-1.23 What will be output of following program?

Code	Solution
`#include <stdio.h>` `int main(){` `  int i;` `  for(i=0;i<=4;i++){` `  printf("%d ",i);` `  }` `return 0;` `}`	Output: 0 1 2 3 4 Expression 1 is called initialization expression. Task of this expression is to initialize the looping variables.

Properties of Expression 1:

1. Expression 1 can initialize the more than one variable.
2. Expression 1 is optional.
3. Unlike Java, in C we cannot declare the variable at the expression 1.

Example-1.24 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i,j,k;` `  for(i=0,j=2,k=1;i<=4;i++){` `  printf("%d ",i+j+k);` `  } return 0;` `}`	Output: 3 4 5 6 7 Here expression 1 initialize 3 variables i, j, k at the same time and value j and k do not change.

Example-1.25 What will be output of following program?

Code	Solution
`#include<stdio.h>` `void main(){` `  int i=1;` `  for(;i<=4;i++){` `  printf("%d ",i);` `  } return 0;` `}`	Output: 1 2 3 4 Expression 1 is optional because we can initialize variable before for loop begin.

Example-1.26 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main()` `{ for(int i=0;i<=10;i++){` `printf("%d ",i);` `} return 0;` `}`	Output: Compilation error We can't declare variable inside the Expression 1.

Expression 2: It is called as conditional expression. Task of expression is to check the condition and if it is false then it terminates the loop.

Example-1.27 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `int i;` `for(i=1;i<=3;i++){` `printf("hi ");` `} printf("%d",i);` `return 0;` `}`	Output: hi hi hi 4 The for loop executes 3 times and print 'hi' 3 times and after that loop termination value of 'i' is printed.

Properties of Expression 2:

1. Expression 2 can have more than one checking condition and if any condition is false loop will terminate.
2. Expression 2 is also optional.
3. It can perform task of Expression 1 as well as Expression 3. That is it can initialize the variables as well as increment the variables.
4. If Expression 2 is zero means condition is false and any non zero number means condition is true.

Example-1.28 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `int i,j=2;` `for(i=0;i<=5,j>=0;i++){` `printf("%d ",i+j);` `j--;` `}` `return 0;` `}`	Output: 2 2 2 Here Expression 2 check two conditions that is i <= 5 and j >= 0 at a time then value is printed.

Example-1.29 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int j;` `  for(j=0; ;j++){` `  printf("%d ",j);` `  if(j>=2)` `  break;` `  }` `return 0;` `}`	Output: 0 1 2 Expression 2 is optional, not necessary to include every time.

Example-1.30 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i;` `  for(i=0;-5 ;i++){` `  printf("%d ",i);` `  if(i==3)` `  break;` `  } return 0;` `}`	Output: 0 1 2 3 Condition is assumed false when the value is 0. So anything except 0 will result in true. Hence loop will go for infinite looping unless we use break statements as in the given code.

Expression 3: It is called as incrementation expression. Task of this expression is to increment or decrement the variable.

Properties of Expression 3:

1. We can increment more than one variable at the same time in the Expression 3.
2. Expression 3 is also optional.

Example-1.31 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i,j,k;` `  for(i=0,j=0,k=0;i<=5,j<=4,k<=3;i++,++j,k+=2){` `  printf("%d ",i+j+k);` `  } return 0;` `}`	Output: 0 4 Here we increment 3 variables at the same time i.e. i, j, k.

Example-1.32 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i;` `  for(i=0;i<=3; ){` `  printf("%d ",i++);` `  } return 0;` `}`	Output: 0 1 2 3 Expression 3 is also optional i.e. we can increment value at other place in the for loop.

Loop Body

Loop body contains the part of code which we have to execute multiple numbers of times.

Properties of Loop Body:

1. If loop body contain only one statement then brace is optional.
2. Loop without body is possible.
3. Braces of loop body behave as block.

Example-1.33 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i,j=0;` `  for(i=0;i<=3;++i,i++,++j)` `  printf("%d %d ",i,j);` `  return 0;` `}`	Output: 0 0 2 1 Here loop body contain only statement i.e. printf statement. So braces are optional.

Example-1.34 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i;` `  for(i=0;i<=10;i++);` `  printf("%d",i);` `return 0;` `}`	Output: 11 Here for loop will be run 10 times after that value of 'i' is printed by printf function.

Example-1.35 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i;` `  for(i=0;i<=2;i++){` `  int i=8;` `  printf("%d ",i);` `  }` `  printf("%d",i);` `return 0;` `}`	Output: 8 8 8 3 Here braces of for loop works as blocks separator which decide scope. First printf is in for loop block but second printf outside for loop block.

1.7.2 While loop

It is pre-tested loop. It is used when we have to execute a part of code in unknown numbers of times till the condition is fullfilled.

Syntax of while loop:

```
while (Expression){
      Loop body
}
```

Properties of while loop:

1. Task of the expression is to check the condition. Loop will execute until condition is true otherwise loop will terminate.
2. If any expression returns zero then condition will false and if it returns any non- zero number then condition will true.
3. In while loop condition expression is compulsory.
4. While loop without any body is possible.
5. In while loop there can be more than one conditional expression.
6. If loop body contain only one statement then brace is optional.

Example-1.36 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int x=3,y=2;` `  while(x+y-1){` `  printf("%d",x=+y);` `  }` `return 0;` `}`	Output: Infinite loop Condition of while loop never goes to zero i.e. false. So program goes to infinite loop.

Example-1.37 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  while(){` `  printf("Hello world");` `  }` `  return 0;` `}`	Output: Compilation error If condition of while loop is not given then it gives compilation error. Because compiler don't know whether condition is True or False.

Example-1.38 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i=0;` `  while(i++,i<=8);` `  printf("%d",i);` `return 0;` `}`	Output: 9 We can increment and check the condition simultaneously. First condition is checked after that incrementation is done.

Example-1.39 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int x=2,y=2;` `  while(x<=5 && y<=3)` `  printf("%d %d ",++x, ++y);` `return 0;` `}`	Output: 3 3 4 4 We can check two conditions in while loop. First time 3 3 would be printed. Second time 4 4 would be printed. After that condition will fail.

Example-1.40 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  while(!printf("Hello world"));` `return 0;` `}`	Output: Hello world Here while loop doesn't have any body. So no need of braces { }.

1.7.3 Do while Loop in C

It is also called as post-tested loop. It is used when it is necessary to execute the loop at least one time.

Syntax for do while loop:

```
do { Loop body
} while (Expression);
```

Example-1.41 What will be output of following program?

Code	Solution
`void main()` `{ int num,i=0;` `do{` `printf("To enter press 1\n");` `printf("To exit press 2");` `scanf("%d",&num);` `++i;` `switch(num){` `case 1:printf("You are welcome\n");break;` `default : exit(0);` `}` `} while(i<=10);` `getch();` `}`	Output: To enter press 1 To exit press 2 // is num pressed is 1 you are welcome // else exit.

Example-1.42 What will be output of following program?

Code	Solution
`void main(){` `double i=5.5678;` `do{` `printf("hi");` `} while(!i);` `getch();` `}`	Output: hi In do-while loop its body is executed as long as while condition is True. Otherwise loop will terminate.

1.7.4 Break and Continue Keywords in C Programming

Break:

1. It is keyword of C programming. Task of this keyword is to bring the control from out of the loop in the case of looping.
2. Another task of break keyword is to switch the control from one case to another case in case of switch case control statement.

Example-1.43 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `int i;` `for(i=0;i<=4;i++){` `printf("%d",i);` `break;` `printf("Label 1");` `}` `printf("Label 2");` `return 0;` `}`	Output: 0 Label 2 Here break keyword in for loop is used to bring the control out of for loop. So first printf print 0 then Second printf outside the for loop print Label 2

Example-1.44 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i=0;` `  ++i;` `  switch(i){` `  case 1:printf("case 1");` `  case 2:printf("case 2");break;` `  case 3:printf("case 3");` `  default: printf("default");` `  } return 0;` `}`	Output: case 1 case 2 Here case 1 and case 2 is printed because there is no break statement between them. If there is no break in switch case then all cases will be printed except default case.

Continue: It is keyword of C and task of this keyword is to transfer the control of program at the beginning of loop.

Example-1.45 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i=5;` `  do{` `  printf("%d",i);` `  continue;` `  i++;` `  } while(i<=10);` `return 0;` `}`	Output: Infinite loop Control is transferred to the beginning of block (do-while block) because of "continue" keyword.

NOTE: Except looping, we cannot use continue keyword.

Example-1.46 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int x;` `  scanf("%d",&x);` `  switch(x){` `  case 1:printf("1");break;` `  case 2:printf("2");continue;` `  default:printf("3");` `  }` `return 0;` `}`	Output: Compilation error Here "continue" is part of switch case which is not allowed. Hence it gives compile time error.

1.8 Const Qualifier

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed.

Pointer to Variable

Format: `int *ptr;`

The value of ptr as well as value of object ptr pointing to can be changed. Pointer and value pointed by pointer both are stored in read-write area.

***Example*-1.47** What will be output of following C code?

Code	Solution
`#include <stdio.h>` `int main(void)` `{ int i = 15;` `int j = 20;` `int *ptr = &i;` `printf("*ptr: %d\n", *ptr);` `ptr = &j;` `printf("*ptr: %d\n", *ptr);` `*ptr = 200;` `printf("*ptr: %d\n", *ptr);` `return 0;` `}`	**Output:** *ptr: 15 *ptr: 20 *ptr: 200

Pointer to Constant

Format: `const int *ptr;` **or** `int const *ptr;`

The pointer can be changed to point to any other integer variable, but cannot change value of object (entity) pointed using pointer ptr. Pointer is stored in read-write area. Object pointed may be in read only or read write area.

***Example*-1.48** What will be output of following C code?

Code	Solution
`#include <stdio.h>` `int main(void)` `{` `int i = 15;` `int j = 20;` `const int *ptr = &i;` `printf("ptr: %d\n", *ptr);` `*ptr = 200;` `ptr = &j;` `printf("ptr: %d\n", *ptr);` `return 0;` `}`	**Output :** error: assignment of read-only location '*ptr'

Constant Pointer to Variable

Format: `int *const ptr;`

Since declaration is constant pointer to integer variable, therefore we can change value of object pointed by pointer, but cannot change the pointer to point another variable.

Example-1.49 What will be output of following C code?

Code	Solution
`#include <stdio.h>` `int main(void)` `{ int i = 15; int j = 20;` `int *const ptr = &i;` `printf("ptr: %d\n", *ptr);` `*ptr = 200;` `printf("ptr: %d\n", *ptr);` `ptr = &j;` `return 0;` `}`	**Output :** error: assignment of read-only variable 'ptr'

Constant Pointer to Constant

Format: `const int *const ptr;`

This declaration is constant pointer to constant variable which states we cannot change value pointed by pointer as well as we cannot point the pointer to other variable.

Example-1.50 What will be output of following C code?

Code	Solution
`#include <stdio.h>` `int main(void)` `{ int i = 15;` `int j = 20;` `const int *const ptr = &i;` `printf("ptr: %d\n", *ptr);` `ptr = &j;` `*ptr = 200;` `return 0;` `}`	**Output :** error : assignment of read-only variable 'ptr' error: assignment of read-only location '*ptr'

1.9 Strings in C

In C, a string can be referred either using a character pointer or as a character array.

1. **Strings as character arrays:**
   char ptr[4] = "madeeasy"; /* one extra for string ending */
   representation of above declaration,

   char ptr[4] = | m | a | d | e | e | a | s | y | /0 | /* '/0' is string terminator */

   When strings are declared as character arrays, it is stored like other types of arrays in C i.e. if ptr[] is an auto variable then string is stored in stack segment, if it's a global or static variable then stored in data segment.
2. **Strings using character pointers:** Using character pointer strings can be stored in two ways:
   (*i*) *Read only string in a shared segment:* When string value is directly assigned to a pointer, in most of the compilers, it's stored in a read only block that is shared among functions.
   char *ptr = "madeeasy"; in given declaration "madeeasy" is stored in a shared read only location, but pointer ptr is stored in a read-write memory. You can change ptr to point something else but cannot change value at present ptr. So this kind of string should only be used when we do not want to modify string at any point of time in the program.
   (*ii*) *Dynamically allocated in heap segment:* Strings are stored like other dynamically allocated things in C and can be shared among functions.

```
char *ptr;
int length = 9;          /* one extra for '/0' */      [we assume character take one byte]
ptr = (char *malloc (sizeof (char) * length);
*(ptr + 0) = 'm';  *(ptr + 1) = 'a';  *(ptr + 2) = 'd';  *(ptr + 3) = 'e';  *(ptr + 4) = 'e';
*(ptr + 5) = 'a';  *(ptr + 6) = 's';  *(ptr + 7) = 'y';  *(ptr+8) = '/0';
```

1.10 Function

A large program is divided into basic building blocks called function. Function contains set of instructions enclosed by "{ }" which performs specific operation in a program. Actually, Collection of these functions creates a program.

Uses of C functions:

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, reusability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

There are 3 aspacts in each C function. They are,

- **Function declaration or prototype:** This informs compiler about the function name, function parameters and return value's data type.
- **Function call:** This calls the actual function.
- **Function definition:** This contains all the statements to be executed.

S.No.	C Function Aspects	Syntax
1	function definition	return_type function_name (arguments list) {Body of function; }
2	function call	function_name (arguments list);
3	function declaration	return_type function_name (arguments list);

Simple example program for C function:

- As you know, functions should be declared and defined before calling in a C program.
- In the below program, function "square" is called from main function.
- The value of "m" is passed as argument to the function "square". This value is multiplied by itself in this function and multiplied value "p" is returned to main function from function "square".

***Example*-1.51** What will be output of following program?

Code	Solution
`#include<stdio.h>` `float square(float x);  //function prototype, also called function declaration` `int main(){  //main function, program starts from here` `  float m,n;` `  printf("\nEnter some number for finding square \n");` `  scanf("%f",&m);` `  n=square(m);  //function call` `  printf("\nSquare of the given number %f is %f",m,n);` `}` `float square(float x){  //function definition` `  float p;` `  p=x*x;` `  return(p);` `}`	Output: Enter some number for finding square: 2 Square of the given number 2.000000 is 4.000000

All C functions can be called either with arguments or without arguments in a C program. These functions may or may not return values to the calling function. Now, we will see simple example in C programs for each one of the following.

1. C function with arguments (parameters) and with return value
2. C function with arguments (parameters) and without return value
3. C function without arguments (parameters) and without return value
4. C function without arguments (parameters) and with return value

S.No.	C Function Aspects	Syntax
1	with arguments and with return values	int function (int); // function declaration function (a); // function call int function (int a) // function definition {statements; return a;}
2	with arguments and without return values	void function (int); // function declaration function (a); // function call void function (int a) // function definition {statements;}
3	without arguments and without return values	void function (); // function declaration function (); // function call void function () // function definition {statements;}
4	without arguments and with return values	int function (); // function declaration function (); // function call int function () // function definition {statements; return a;}

NOTE: If the return data type of a function is "void", then, it can't return any values to the calling function. If the return data type of the function is other than void such as "int, float, double etc.", then, it can return values to the calling function.

Program for with arguments and with return value: In this program, integer, array and string are passed as arguments to the function. The return type of this function is "int" and value of the variable "a" is returned from the function. The values for array and string are modified inside the function itself.

***Example*-1.52** What will be output of following C code?

Code	Solution
`#include<stdio.h>` `#include<string.h>` `int function(int, int[], char[]);` `int main(){` `  int i, a = 20;` `  int arr[5] = {10,20,30,40,50};` `  char str[30] = "madeeasy";` `  printf("values before modification\n");` `  printf("value of a is %d\n",a);` `  for(i=0;i<5;i++)` `  { printf("value of arr[%d] is %d\n", i, arr[i]);` `  //Accessing each variable` `  } printf("value of str is %s\n",str);` `  printf("\n values after modification\n");` `  a= function(a, &arr[0], &str[0]);` `  printf("value of a is %d\n",a);`	Output: values before modification value of a is 20 value of arr[0] is 10 value of arr[1] is 20 value of arr[2] is 30 value of arr[3] is 40 value of arr[4] is 50 value of str is "madeeasy" values after modification value of a is 40 value of arr[0] is 60 value of arr[1] is 70 value of arr[2] is 80

```
    for(i=0;i<5;i++)
    {   printf("value of arr[%d] is %d\n", i, arr[i]);
        // Accessing each variable
    }   printf("value of str is %s\n",str);
    return 0;
}   int function(int a, int *arr, char *str){
    int i; a = a+20;
    arr[0] = arr[0]+50;
        arr[1] = arr[1]+50;
        arr[2] = arr[2]+50;
        arr[3] = arr[3]+50;
        arr[4] = arr[4]+50;
        strcpy(str," modified string");
        return a;
}
```

value of arr[3] is 90
value of arr[4] is 100
value of str is "modified string"

Example program for with arguments and without return value: In this program, integer, array and string are passed as arguments to the function. The return type of this function is "void" and no values can be returned from the function. All the values of integer, array and string are manipulated and displayed inside the function itself.

Example-1.53 What will be output of following C code?

Code	Solution

```
#include<stdio.h>
void function(int, int[], char[]);
int main()
{   int a=20;
    int arr[5]={10,20,30,40,50};
    char str[30]="madeeasy";
    function(a,&arr[0],&str[0]);
    return 0;
}
void function(int a,int*arr,char*str)
{   int i;
    printf("value of a is %d\n\n",a);
    for(i=0;i<5;i++){
        printf("value of arr[%d] is %d\n", i, arr[i]);
            //Accessing each variable
}
printf("\nvalue of str is %s\n",str);
}
```

Output:
value of a is 20
value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
value of str is "madeeasy""

Example program for without arguments and without return value: In this program, no values are passed to the function "test" and no values are returned from this function to main function.

Example-1.54 What will be output of following C code?

Code	Solution

```
#include<stdio.h>
void test();
int main(){
    test();
    return 0;
}
void test()
{   int a = 50, b = 80;
    printf("\nvalues: a=%d and b=%d", a,b);
}
```

Output:
values : a = 50 and b = 80
Since their is no parameter passed during function call test();.
So there is no return value when function test(); complete its execution.

Example program for without arguments and with return value: In this program, no arguments are passed to the function "sum". But, values are returned from this function to main function. Values of the variable a and b are summed up in the function "sum" and the sum of these value is returned to the main function.

Example-1.55 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `int sum();` `int main(){` `  int addition;` `  addition = sum();` `  printf("\nSum of two given values =%d", addition);` `  return 0;` `}` `int sum(){` `  int a = 50, b = 80, sum;` `  sum = a + b;` `  return sum;` `}`	Output: Sum of two given values =130. Here function sum(); is passed without parameter but it is integer type. So after completion it return integer type value.

1.10.1 Library Function

- Only one value can be returned from a function.
- If you try to return more than one values from a function, only one value will be returned that appears at the right most place of the return statement.
- For example, if you use "return a,b,c" in your function, value for c only will be returned and values a, b won't be returned to the program.
- In case, if you want to return more than one values, pointers can be used to directly change the values in address instead of returning those values to the function.
- Library functions in C language are inbuilt functions which are grouped together and placed in a common place called library.
- Each library function in C performs specific operation.
- We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs.
- All C standard library functions are declared in many header files which are saved as file_name.h.
- Actually, function declaration, definition for macros are given in all header files.
- We are including these header files in our C program using "#include<filename.h>" command to make use of the functions those are declared in the header files.
- When we include header files in our C program using "#include<filename.h>" command, all C code of the header files are included in C program. Then, this C program is compiled by compiler and executed.

List of Most used Header Files in C

Check the below table to know all the C library functions and header files in which they are declared.

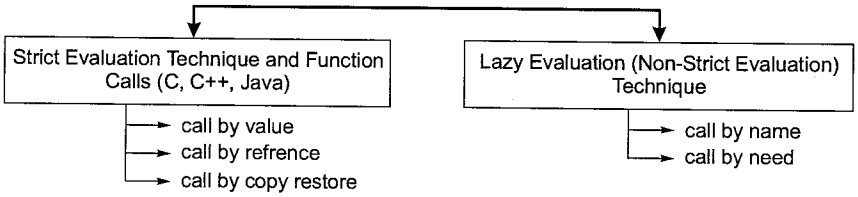
S. No.	Header file	Description
1	`stdio.h`	This is standard input/output head file in which input/Output functions are declared
2	`conio.h`	This is console input/output head file
3	`string.h`	All string related functions are defined in this head file
4	`stdlib.h`	This header file contains general functions used in C programs

5	math.h	All maths related functions are defined in this header file
6	time.h	This head file contains time and clock related functions
7	ctype.h	All character handling functions are defined in this header file
8	stdarg.h	Variable argument functions are declared in this header file
9	signal.h	Signal handling functions are declared in this file
10	setjmp.h	This file contains all jump functions
11	locale.h	This file contains locale functions
12	errno.h	Error handling functions are given in this file
13	assert.h	This contains diagnostics functions

Key Points to Remember While Writing Functions in C

- All C programs contain main() function which is mandatory.
- main() function is the function from where the execution of every C program starts.
- Name of the function is unique in a C program.
- C Functions can be invoked from anywhere within a C program.
- There can any number of functions be created in a program. There is no limit on this.
- There is no limit in calling C functions in a program.
- All functions are called in sequence manner specified in main() function.
- One function can be called within another function.
- C functions can be called with or without arguments/parameters. These arguments are nothing but inputs to the functions.
- C functions may or may not return values to calling functions. These values are nothing but output of the functions.
- When a function completes its task, program control is returned to the function from where it is called.
- There can be functions within functions.
- Before calling and defining a function, we have to declare function prototype in order to inform the compiler about the function name, function parameters and return value type.
- C function can return only one value to the calling function.
- When return data type of a function is "void", then, it won't return any values
- When return data type of a function is other than void such as "int, float, double", it returns value to the calling function.
- main() program comes to an end when there is no functions or commands to execute.
- There are 2 types of functions in C. They are, (1) Library functions and (2) User defined functions.

1.10.2 Types of Evaluation of Function



Call by value

- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter can not be modified by formal parameter.
- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

NOTE: Actual parameter = This is the argument which is used in function call.
Formal parameter = This is the argument which is used in function definition

Example program for C function (using call by value):

- In this program, the values of the variables "m" and "n" are passed to the function "swap".
- These values are copied to formal parameters "a" and "b" in swap function and used.

Example-1.56 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `//function prototype, also called function declaration` `void swap(int a, int b);` `int main(){` `  int m=22, n=44;  // calling swap function by value` `  printf("values before swap m=%d\n and n=%d", m,n);` `  swap(m,n);` `  printf("\nvalues after swap m=%d\n and n=%d",m,n);` `}` `void swap(int a, int b)` `{  int tmp;` `  tmp=a;` `  a=b;` `  b=tmp;` `}`	Output : Values before swap m = 22 and n = 44 values after swap m = 22 and n = 44.

Call by reference

- In call by reference method, the address of the variable is passed to the function as parameter.
- The value of the actual parameter can be modified by formal parameter.
- Same memory is used for both actual and formal parameters since only address is used by both parameters.

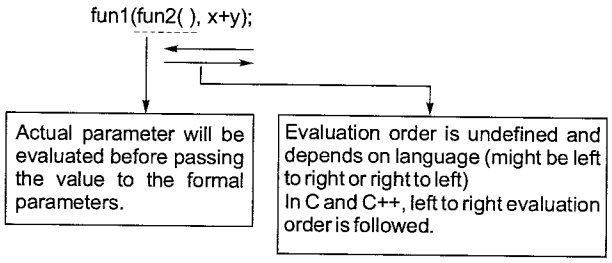
Example program for C function (using call by reference):

- In this program, the address of the variables "m" and "n" are passed to the function "swap".
- These values are not copied to formal parameters "a" and "b" in swap function.
- Because, they are just holding the address of those variables.
- This address is used to access and change the values of the variables.

Example-1.57 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `//function prototype, also called function declaration` `void swap(int *a, int *b);` `int main(){` `  int m=22, n=44;` `  //calling swap function by reference` `  printf("values before swap m=%d\n and n=%d", m,n);` `  swap(&m, &n);` `  printf("\n values after swap a=%d\n and b=%d",m, n);` `}` `void swap(int *a, int *b)` `{` `  int tmp;` `  tmp=*a;` `  *a=*b;` `  *b=tmp;` `}`	Output: Values before swap m = 22 and n = 44 values after swap m = 44 and n = 22

NOTE: (Evaluation order of sub expressions or sub functions) with Example.



1.11 Recursion

Any function which calls itself is called recursive. A recursive method solves a problem by calling a copy of itself to work on a smaller problem. This is called the recursion step. The recursion step can result in many more such recursive calls. It is important to ensure that the recursion terminates.

Recursion is a useful technique borrowed from mathematics. Recursive code is generally shorter and easier to write than iterative code. Generally, loops are turned into recursive functions when they are compiled or interpreted. Recursion is most useful for tasks that can be defined in terms of similar subtasks. For example, sort, search and traversal problems often have simple recursive solutions.

1.11.1 Format of a Recursive Function

A recursive function performs a task in part by calling itself to perform the subtasks. At some point, the function encounters a subtask that it can perform without calling itself. This case, where the function does not recur, is called the base case, the former, where the function calls itself to perform a subtask, is referred to as the recursive case. We can write all recursive functions using the format:

```
if(test for the base case)
      return some base case value
else if(test for another base case)
      return some other base case value   // the recursive case
else
      return (some work) and then (a recursive call)
```

As an example consider the factorial function: $n!$ is the product of all integers between n and 1. Definition of recursive factorial looks like:

$$n! = 1, \quad \text{if } n = 0$$
$$n! = n * (n - 1)! \quad \text{if } n > 0$$

This definition can easily be converted to recursive implementation. Here the problem is determining the value of n!, and the subproblem is determining the value of $(n-1)!$. In the recursive case, when n is greater than 1, the function calls itself to determine the value of $(n-1)!$ and multiples that with n. In the base case, when n is 0 or 1, the function simply returns 1.

This looks like the following:

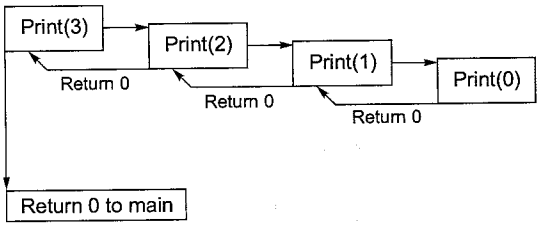
```
//calculates factorial of a positive integer
int Fact(int n){
      if(n==1)return 1;       //base cases: fact of 0 or 1 is 1
      else if(n==0) return 1;
      //recursive case: multiply n by (n - 1) factorial
      else  return n*Fact(n-1);
}
```

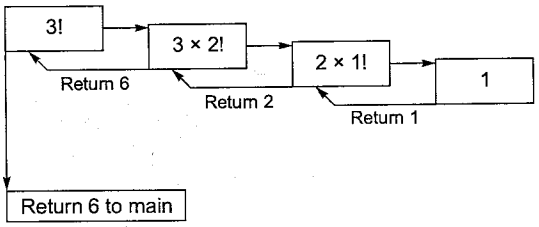
1.11.2 Recursion and Memory (Visualization)

Each recursive call makes a new copy of that method (actually only the variables) in memory. Once a method ends (*i.e.* returns some data), the copy of that returning method is removed from memory. The recursive solutions look simple but visualization and tracing takes times. For better understanding, let us consider the following example.

```
int Print(int n) { //print numbers 1 to n backward)
      if(n==0) // this is the terminating base case
            return 0;
      else{
            printf ("%d",n);
                  return Print(n-1); // recursive call to itself again
      }
}
```

For this example, if we call the print function with n = 3, visually our memory assignments. Now, let us consider our factorial function. The visualization of factorial function with n = 3.





NOTE

- Recursive algorithms have two types of cases, recursive cases and base cases.
- Every recursive function case must terminate at base case.
- Generally iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithms can be implemented without recursive function calls using a stack, but it's usually more trouble than it's worth. That means any problem that can be solved recursively can also be solved iteratively.
- For some problems, there are no obvious iterative algorithms.
- Some problems are best suited for recursive solutions while others are not.

Example of Recursion Algorithms:

- Fibonacci Series, Factorial Finding
- Merge Sort, Quick Sort
- Binary Search
- Tree Traversals and many Tree Problems: In-order, Pre-order Post-order
- Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
- Dynamics Programming Examples
- Divide and Conquer Algorithms
- Towers of Hanoi
- Backtracking algorithms.

1.12 Backtracking

Backtracking is the method of exhaustive search using divide and conquer.

- Sometimes the best algorithm for a problem is to try all possibilities.
- this is always slow, but there are standard tools that can be used to help.
- **Tools:** algorithms for generating basic objects, such as binary strings [2^n possibilities for n-bit string), permutations [n!], combinations [n!/r!(n – r)!], general strings [k – ary strings of length n has k^n possibilities], etc.
- Backtracking speeds the exhaustive search by pruning.

Example of Backtracking Algorithms

- Binary Strings: generating all binary strings
- Generating k-ary Strings
- The Knapsack Problem
- Generalized Strings
- Hamiltonian Cycles
- Graph Coloring Problem

Example-1.58 Generate all the strings of n bits. Assume A[0...n – 1] is an array of size n.

Code	Solution
`void Binary(int n)` `{ if(n < 1)` `printf("%s", A);` `// Assume array A is a global variable` `else` `{ A[n-1]=0;` `Binary(n - 1);` `A[n-1]=1;` `Binary(n - 1);` `}` `}`	Let $T(n)$ be the running time of binary (n). Assume function print f takes times O(1). $T(n) = \begin{cases} c, & \text{if } n < 0 \\ 2T(n-1)+d, & \text{otherwise} \end{cases}$ Using Divide and Conquer Master theorem we get, $T(n) = O(2^n)$. This means that the algorithm for generating bit-string is optimal.

Let us assume we keep current k-ary string in an array A[0... n – 1]. Call function k-strings (n, k):

Example-1.59 Generate all the strings of length n drawn from 0...k – 1.

Code	Solution
`int IsPrime(n)` `{` `int i, n;` `for(i=2; i<=sqrt(n); i++)` `{` `if(n%i==0)` `printf ("Not Prime\n");` `return 0;` `}` `return 1;` `}`	$T(n)$ denote the number of times the for loop is executed by the program on input n. $T(n) = \sqrt{n} + 1$ When n = 1 then best case occurs so T(n) takes constant time i.e., T(n) = $\Omega(1)$. In worst case, $T(n) = \sqrt{n} + 1$ so $T(n) = O(\sqrt{n})$

1.13 C Scope Rules

A scope in any programming language is a region of the program where a defined variable can have its existence and beyond that, variable cannot be accessed. There are three places where variables can be declared in C programming language:

- Inside a function or a block which is called local variables,
- Outside of all functions which is called global variables.
- In the definition of function parameters which is called formal parameters.

1.13.1 Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables. Here all the variables a, b and c are local to main() function.

```
#include <stdio.h>
int main (){
      int a,b,c;              /* local variable declaration */
      a=10; b=20; c=a+b;      /* actual initialization */
      printf ("value of a=%d, b=%d and c=%d\n",a,b,c);
      return 0;
}
```

1.13.2 Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

Following is the example using global and local variables:

```
#include <stdio.h>
int g;  //global variable declaration
int main (){
      int a,b;          //local variable declaration
      a=10; b=20;       // actual initialization
      g=a+b;
      printf ("value of a=%d, b=%d and g=%d\n", a,b,g);
return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference.

Example-1.60 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int g=20; //global variable declaration` `int main(){` `  int g=10; //local variable declaration` `  printf("value of g=%d\n",g);` `return 0;` `}`	Output: value of g=10 Here printf function inside main block so local value of 'g' will be printed.

1.13.3 Block Scope

- A Block is a set of statements which is enclosed within left and right braces ({ and } respectively). Blocks may be nested in C.
- A variable which is declared in a block is accessible in the block and all inner blocks of that block, but not accessible outside the block.
- If an inner block declares a variable with the same name as the variable declared by the outer block, then the visibility of the outer block variable ends at the point of declaration by inner block.

Example-1.61 What will be output of following C code?

Code	Solution
`int main()` `{` `  {` `  int x = 10, y = 20;` `  {` `  // The outer block contains declaration of x and y, so` `  // following statement is valid and prints 10 and 20` `  printf("x = %d, y = %d\n", x, y);` `  {` `  // y is declared again, so outer block y is not accessible` `  // in this block` `  int y = 40;` `  x++; // Changes the outer block variable x to 11` `  y++; // Changes this block's variable y to 41` `  printf("x = %d, y = %d\n", x, y);` `  }` `  // This statement accesses only outer block's variables` `  printf("x = %d, y = %d\n", x, y);` `  }` `  }` `  return 0;` `}`	**Output :** x = 10, y = 20 x = 11, y = 41 x = 11, y = 20

- A function itself is a block. Parameters and other local variables of a function follow the same block scope rules.
- A variable which is declared inside a block can only be accessed inside the block and all inner blocks of this block.

Example - 1.62 What will be output of following C code?

Code	Solution
`int main()` `{` `  {` `  int x = 10;` `  }` `  {` `  printf("%d", x); // Error: x is not accessible here` `  }` `  return 0;` `}`	**Output :** error: 'x' undeclared (first use in this function)

1.13.4 Formal Parameters

Function parameters, which are known as **formal parameters**, are treated as local variables within that function and are given preference over the global variables.

Example - 1.63 What will be output of following program.

Code	Solution
`#include<stdio.h>` `int a=20;  //global variable declaration` `int main(){` `  int a=10; //local variable declaration in main function` `  int b=20;` `  int c=0;` `  printf("value of a in main() =%d\n",a);` `  c=sum(a,b);` `  printf("value of c in main()=%d\n",c);` `  return 0;` `}` `int sum(int a, int b){` `  printf("value of a in sum() = %d\n",a);` `  printf("value of b in sum() = %d\n",b);` `  return a+b;` `}`	Output: using static scoping Value of a in main() = 10 Value of a in sum() = 10 Value of b in sum() = 20 Value of c in main() = 30 Local variables are given higher priority than global variable.

1.14 Storage Class

The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'. Along with the life time of a variable, storage class also determines variable's storage location (memory or registers), the scope (visibility level) of the variable, and the initial value of the variable.

There are following storage classes which can be used in a C Program

- Auto
- Register
- Static
- Extern

1.14.1 Auto - Storage Class

A variable defined within a function or block with auto specifier belongs to automatic storage class. All variables defined within a function or block by default belong to automatic storage class if no storage class is mentioned. Variables having automatic storage class are local to the block which they are defined in, and get destroyed on exit from the block.

Properties of auto storage class:

1. Default initial value of auto variable is garbage.
2. Visibility of auto variable is within the block where it has declared.
3. Scope of auto variable is within the block where it has declared.
4. From above example it is clear auto variable initialize each time.
5. An auto variable gets memory at run time.

Example-1.64 What will be output of following program?

Code	Solution
`#include<stdio.h>` `Int main(){` `  int i;` `  auto char c;` `  float f;` `  printf("%d %c %f",i,c,f);` `  return 0;` `}`	Output: Garbage Garbage Garbage Although 'auto' keyword not use for i and f, by default they belongs to auto storage class. Hence all the 3 variables get default value as Garbage.

Example-1.65 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int a=10;` `  {` `  int a=20;` `  printf("%d",a);` `  }` `  printf(" %d",a);` `  return 0;` `}`	Output: 20 10 Visibility of variable a which has declared inside inner block has scope only within that block.

Example-1.66 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i;` `  for(i=0;i<4;i++){` `  int a=20;` `  printf("%d",a);` `  a++;` `  } return 0;` `}`	Output: 20 20 20 20 Variable 'a' declared inside the for loop block has scope only within that block. After the first iteration variable a becomes dead and it looses its incremented value. In second iteration variable 'a' is again declared and initialized and so on.

Example-1.67 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i=0;` `  {` `  auto int a=20;` `  XYZ:;` `  printf("%d",a);` `  a++;` `  i++;` `  }` `  if (i<3)` `  goto xyz;` `  return 0;` `}`	Output: Compilation error Variable 'a' which declared inside inner block has scope only within that block. Once program control comes out of that block variable will be dead. If with the help of goto statement we will go to inside that inner block in the printf statement complier will not known about variable a because it has been destroyed already. Hence complier will show an error message: undefined symbol a. But if you will write goto statement label before the declaration of variable then there is not any problem because variable a will again declared and initialize.

Example-1.68 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int i=0;` `  { XYZ:;` `  auto int a=20;` `  printf("%d",a);` `  a++;` `  i++;` `  } if(i<3)` `  goto xyz;` `  return 0;` `}`	Output: 20 20 20 Every time the control goes to XYZ: A new variable 'a' is declared and initialized to 20. Hence it doest not given error like previous program.

1.14.2 Register Storage Class

The register specifier declares a variable of register storage class. Variables belonging to register storage class are local to the block which they are defined in, and get destroyed on exit from the block. A register declaration is equivalent to an auto declaration, but hints that the declared variable will be accessed frequently; therefore they are placed in CPU registers, not in memory. Only a few variables are actually placed into registers, and only certain types are eligible; the restrictions are implementation-dependent. However, if a variable is declared register, the unary &(address of) operator may not be applied to it, explicitly or implicitly. Register variables are also given no initial value by the compiler.

Important points about register storage class:

1. In following declaration: `register int a;`
   We are only requesting and not forcing the compiler to store variable a in CPU. Compiler will decide where to store in the variable a.

2. A register variable execute faster than other variables because it is stored in CPU so during the execution compiler has no extra burden to bring the variable from memory to CPU.
3. Since a CPU have limited number of register so it is programmer responsibility which variable should declared as register variable i.e. variable which are using many times should declared as a register variable.
4. We cannot deference register variable since it has not any memory address.
5. Default initial value of register variable is garbage.
6. Scope and visibility of register variable is block.

Example - 1.69 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  register int a=10;` `  int *p;` `  p=&a;` `  printf("%u",p);` `}`	Output: Compilation error Here we are trying to reference a register variable which give a compilation error.

1.14.3 Static Storage Class

The static specifier gives the declared variable static storage class. Static variables can be used within function or file. Unlike global variables, static variables are not visible outside their function or file, but they maintain their values between calls. The static specifier has different effects upon local and global variables. See the following flavours of static specifier.

When static specifier is applied to a local variable inside a function or block, the compiler creates permanent storage for it, much as it creates storage for a global variable but static local variable remains visible only to the function or block in which it is defined. In simple terms, a static local variable is a local variable that retains its value between function calls.

When static specifier is applied to a global variable or a function then compiler makes that variable or function known only to the file in which it is defined. A static global variable has *internal linkage* that means even though the variable is global; routines in other files have no knowledge of it and cannot access and alter its contents directly.

Important points about static keyword:

- It is not default storage class of global variables. For example, analyze the following programs and its output.
- Default initial value of static integral type variables are zero otherwise null.
- A same static variable can be declared many times but we can initialize at only one time.
- We cannot write any assignment statement globally. For example:
- A static variable initializes only one time in whole program.
- If we declared static variable locally then its visibility will within a block where it has declared.
- If declared a static variable or function globally then its visibility will only the file in which it has declared not in the other files.

Example-1.70 What will be output of following program?

Code	Solution
`#include<stdio.h>` `static char c;` `static int i;` `static float f;` `static char *str;` `int main(){` `  printf("%d %d %f %s",c,i,f,str);` `  return 0;` `}`	Output: 0 0 0.000000 (null) Since c, i, f and str are declaration. So by default zero is stored in these variable.

Example-1.71 What will be output of following program?

Code	Solution
`#include<stdio.h>` `static int i;  //Declaring the variable i.` `static int i=25;  //Initializing the variable.` `static int i;  //Again declaring the variable i.` `int main(){` `  static int i;  //Again declaring the variable i.` `  printf("%d",i);` `  return 0;` `}`	Output: 25 Here variable 'i' is declared many times but memory assigned only one time.

Example-1.72 What will be output of following program?

Code	Solution
`#include<stdio.h>` `static int i;  //Declaring the variable` `static int i=25; //Initializing the variable` `int main(){` `  printf("%d",i);` `  return 0;` `}` `static int i=20; //Again initializing the variable`	Output: Compilation error Multiple initialization of variable 'i'.

Example-1.73 What will be output of following program?

Code	Solution
`#include<stdio.h>` `Static int i=10; //Initialization statement` `i=25;  //Assignment statement` `int main(){` `  printf("%d",i);` `  return 0;` `}`	Output: Compilation error Here we assigned value 25 globally. So its compilation error.

NOTE: Assigning any value to the variable at the time of declaration is known as initialization while assigning any value to variable not at the time of declaration is known assignment.

Example-1.74 What will be output of following program?

Code	Solution
`#include<stdio.h>` `static int i=10;` `int main(){` `  i=25;  //Assignment statement` `  printf("%d",i);` `  return 0;` `}`	Output: 25 Here memory allocated to variable 'i' is 10 globally which is replaced by 25 inside the main block.

Example-1.75 What will be output of following program?

Code	Solution
`#include<stdio.h>` `static int i=10;` `int main(){` `  i=5;` `  for(i=0;i<5;i++){` `  static int a=10;  //This statement will execute only time.` `  printf("%d",a++);  //This statement will execute five times.` `  }` `  return 0;` `}`	Output: 10 11 12 13 14 'i' is declared globally (static too). For variable (static) 'a' declared inside the for loop, memory will be allocated only once. Hence subsequent initialization do not have any effect on 'a'

Example-1.76 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `{` `  static int a=5;` `  printf("%d",a);` `}` `  //printf("%d",a); variable a is not visible here` `  return 0;` `}`	Output: 5 Here variable 'a' is declared locally with in the block. So it prints value of 'a' for first printf function.

1.14.4 External Storage Class

The extern specifier gives the declared variable external storage class. The principal use of extern is to specify that a variable is declared with *external linkage* elsewhere in the program. To understand why this is important, it is necessary to understand the difference between a declaration and a definition. A declaration declares the name and type of a variable or function. A definition causes storage to be allocated for the variable or the body of the function to be defined. The same variable or function may have many declarations, but there can be only one definition for that variable or function.

When extern specifier is used with a variable declaration then no storage is allocated to that variable and it is assumed that the variable has already been defined elsewhere in the program. When we use extern specifier the variable cannot be initialized because with extern specifier variable is declared, not defined.

Important points about extern keyword:

1. It is default storage class of all global variables as well all functions. For example, Analyze following two C code and its output:
2. When we use extern modifier with any variables it is only declaration i.e. memory is not allocated for these variable. Hence in second case compiler is showing error unknown symbol i. To define a variable i.e. allocate the memory for extern variables it is necessary to initialize the variables.
3. If you will not use extern keyword with global variables then compiler will automatically initialize with default value to extern variable.
4. Default initial value of extern integral type variable is zero otherwise null.
5. We cannot initialize extern variable locally i.e. within any block either at the time of declaration or separately. We can only initialize extern variable globally.
6. If we declare any variable as extern variable then it searches that variable either it has been initialized or not. If it has been initialized which may be either extern or static then it is okay otherwise compiler will show an error.
7. A particular extern variable can be declared many times but we can initialize at only one time.
8. We cannot write any assignment statement globally.
9. If declared an extern variables or function globally then its visibility will whole the program which may contain one file or many files.

Example-1.77 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int i;  //By default it is extern variable` `int main(){` `  printf("%d",i);` `  return 0;` `}`	Output: 0 Here by default global declaration is comes under external storage class.

Example-1.78 What will be output of following program?

Code	Solution
`#include<stdio.h>` `extern int i;//extern variable` `int main(){` `  printf("%d",i);` `  return 0;` `}`	Output: Compilation error Here globally 'i' is declared using extern keyword which means don't create memory, memory already created. So by the time of printf function it give compilation error. Undefined symbol.

Example-1.79 What will be output of following program?

Code	Solution
`#include<stdio.h>` `void sum(int,int)  //By default it is extern` `int main(){` `  int a=5,b=10;` `  sum(a,b);` `  return 0;` `}` `void sum(int a,int b){` `  printf("%d"",a+b);` `}`	Output: 15 Here we not used extern keyword globally when declared variable. So by default it behaves as a extern storage class.

Example-1.80 What will be output of following program?

Code	Solution
`#include<stdio.h>` `extern int i=10;  //extern variable` `int main(){` `  printf("%d",i);` `  return 0;` `}`	Output: 10 Here we initialize the variarble using extern keyword. So it will not give any error and print global value of variable 'i'.

Example-1.81 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  extern int i=10; //Try to initialize extern variable locally` `  printf("%d",i);` `  return 0;` `}`	Output: Compilation error Compilation error: Cannot initialize extern variable.

Example-1.82 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  extern int i; //It will search the any initialized variable i which may be static or extern` `  printf("%d",i);` `  return 0;` `}` `extern int i=20; //Initialization of extern variable i`	Output: 20 Here we declare variable as extern then it searches the variable either it has been initialize or not. If it has been initialize which may be either extern or static then its okay otherwise compilation will come.

Example-1.83 What will be output of following program?

Code	Solution
`extern int i; //Declaring the variable i` `int i=25;  //Initializing the variable` `extern int i; //Again declaring the variable i` `#include <stdio.h>` `int main()` `{  extern int i; //Again declaring the variable i` `  printf("%d",i);` `  return 0;` `}`	Output: 25 Here memory is created for variable 'i' at a time of initializing again declaring the variable 'i' with keyword extern point the same memory location. So it will print the value of memory location pointed by extern keyword variable 'i'.

Example-1.84 What will be output of following program?

Code	Solution
`extern int i; //Declaring the variable` `int i=25;  //Initializing the variable` `#include <stdio.h>` `int main(){` `  printf("%d",i);` `  return 0;` `}` `int i=20;  //Initializing the variable`	Output: Compilation error Here we initialize the variable multiple time so it will give compilation error.

NOTE: Assigning any value to the variable at the time of declaration is known as initialization while assigning any value to variable not at the time of declaration is known assignment.

Example-1.85 What will be output of following program?

Code	Solution
`#include <stdio.h>` `extern int i;` `int main(){` `  i=25;  //Assignment statement` `  printf("%d",i);` `  return 0;` `}` `int i=10;  //Initialization statement`	Output: 25 Here assignment statement is given locally (in main function) so there is no compilation error. It will print value of 'i' i.e. 25.

NOTE: Extern keyword says don't create memory for me because for me memory is already created.

1.15 Pointers

Pointer is a variable just like other variables of C but only difference is unlike other variable it stores the memory address of any other variables of C. This variable may be type of int, char, array, structure, function or any other pointers.

For *examples*:

- Pointer p which is storing memory address of a int type variable:
```
int i=50;
int *p=&i;
```
- Pointer p which is storing memory address of an array:
```
int arr[20];
int (*p)[20]=&arr;
```
- Pointer p which is storing memory address of a function:
```
void display(char);
void(*p)(char)=&display;
```
- Pointer p which is storing memory address of struct type variable:
```
struct abc
{  int a;
  float b;
}var;
struct abc *p=&var;
```

Precedence of Postfix ++ and Prefix ++ in C/C++

Precedence of Prefix ++ (or Prefix – –) and dereference (*) operators is same, and precedence of Postfix ++ (or Postfix –) is higher than both Prefix ++ and *. If p is a pointer then *p++ is equivalent to *(p++) and ++*p is equivalent to ++(*p) (both Prefix ++ and * are right associative).

Example - 1.86 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `int main()` `{ char arr[] = "madeeasy";` `  char *p = arr;` `  ++*p;` `  printf(" %c", *p);` `  getchar();` `  return 0;` `}`	Output: *n*
`#include<stdio.h>` `int main()` `{ char arr[] = "madeeasy";` `  char *p = arr;` `  *p++;` `  printf("%c", *p);` `  getchar();` `  return 0;` `}`	Output: *a*

1.15.1 How to Read Complex Pointers in C Programming

Rule 1: Assign the priority to the pointer declaration considering precedence and associative according to following table.

Operator	Precedence	Associative
( ), [ ]	1	Left to right
* , Identifier	2	Right to left
Data type	3	

Where

( ) : This operator behaves as bracket operator or function operator.

[ ] : This operator behaves as array subscription operator.

*** :** This operator behaves as pointer operator not as multiplication operator.

Identifier:

It is not an operator but it is name of pointer variable. You will always find the first priority will be assigned to the name of pointer.

Data type:

It is also not an operator. Data types also includes modifier (like signed int, long double etc.)

You will understand it better by examples:

Example: How to read following pointer?

```
char(*ptr)[3]
```

Solution:

Step 1: () and [] enjoys equal precedence. So rule of associative will decide the priority. Its associative is left to right So first priority goes to ().

```
char (*ptr)    [3]
      1        2
```

Step 2: Inside the bracket * and ptr enjoy equal precedence. From rule of associative (right to left) first priority goes to ptr and second priority goes to *.

```
char(*ptr)     [3]
      2        1
```

Step 3: Assign third priority to [].

```
char (*ptr)    [3]
2      1       3
```

Step 4: Since data type enjoys least priority so assign fourth priority to char.

```
char (* ptr) [3]
  4   2 1 3
```

Example: void (*ptr)(int (*)[2],int (*) void))

Solution: Assign the priority considering rule of precedence and associative.

```
void (*ptr)(int (*)[2],int (*) void))
              2 1     2 3 1
  4   2 1             3
```

Now read it following manner: ptr is **pointer** to such **function** whose first parameter is **pointer** to one dimensional **array** of size two having content of **int** type data and second parameter is pointer to such **function** whose parameter is void and return type is **int** data type and return type is **void.**

1.15.2 Special Pointers

Dangling pointer: A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

There are multiple ways by which a pointer can act as dangling pointer:

(*i*) **De-allocation of memory:** Here the memory pointed by ptr is deallocated which results in dangling pointer.

Code

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
      int *ptr = (int *)malloc(sizeof(int));
      // After below free call, ptr becomes dangling pointer
      free(ptr);
      ptr = NULL;    // No dangling pointer
}
```

(*ii*) **Function Call:** The pointer pointing to local variable becomes dangling when local variable is static.

Example-1.87 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `int *fun()` `{` `  /* x is local variable and goes out of scope after an execution of fun() is over */` `  int x = 5;` `  return &x;` `}` `int main()` `{` `  int *p = fun();` `  fflush(stdin);` `  /* p points to something which is not valid anymore*/` `  printf("%d", *p);` `  return 0;` `}`	**Output :** Invalid address

The above problem can be solved as: the pointer pointing to local variable doesn't become invalid when local variable is static.

Example-1.88 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `#include<stdio.h>` `int *fun()` `{` `  // x now has scope throughout the program` `  static int x = 5;` `  return &x;` `}` `int main()` `{` `  int *p = fun();` `  fflush(stdin);` `  /* Not a dangling pointer as it points to static variable.*/` `  printf("%d",*p);` `}`	Output : 5

(*iii*) **Variable goes out of scope:** Void pointer is a specific pointer type. **void*** is a pointer that points to some data location in storage, which doesn't have any specific type. The type of data that it can point can be any. For instance, if we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Since any pointer type is convertible to a void pointer hence it can point to any value.

NOTE: 1. void pointers cannot be dereferenced. It can however be done using typecasting the void pointer
2. Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

```
void main()
{
   int *ptr;
   .......
   .......
   {
      int ch;
      ptr = &ch;
   }
   .......
   // Here ptr is dangling pointer
}
```

Example - 1.89 What will be output of following C code?

Code	Solution
`#include<stdlib.h>` `int main()` `{` `  int x = 4;` `  float y = 5.5;` `  void *ptr; ptr = &x;` `  printf("Integer variable is = %d", *((int*) ptr));` `  float ptr = &y;` `  printf("\nFloat variable is = %f",*((float*)ptr));` `  return 0;` `}`	**Output :** Integer variable is = 4 Float variable is= 5.500000

NULL Pointer

NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

Example - 1.90 What will be output of following C code?

Code	Solution
`#include <stdio.h>` `int main()` `{` `  int *ptr = NULL;` `  printf("The value of ptr is %u", ptr);` `}`	**Output :** The value of ptr is 0

NOTE: 1. **NULL vs Uninitialized pointer:** An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.

2. **NULL vs Void Pointer:** Null pointer is a value, while void pointer is a type.

Wild Pointer

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

Code
int main() { int *p; int x = 10; p = &x; return 0; }

Precedence Rule between Prefix ++, Postfix ++ and Dereferencing (*)

The output of above statements inprograms can be easily guessed by remembering following simple rules about postfix ++, prefix ++ and * (dereference) operators

1. Precedence of prefix ++ and * is same. Associativity of both is right to left.
2. Precedence of postfix ++ is higher than both * and prefix ++. Associativity of postfix ++ is left to right.

Example-1.91 What will be output of following C code?

Code	Solution
#include <stdio.h> int main(void) { int arr[]={10, 20}; int *p=arr; ++*p; printf("arr[0]=%d, arr[1] = %d, *p=%d", arr[0], arr[1], *p); return 0; }	Output: arr[0] = 11, arr[1] = 20, *p = 11

Example-1.92 What will be output of following C code?

Code	Solution
#include <stdio.h> int main(void) { int arr[]={10, 20}; int *p=arr; *p++; printf("arr[0]=%d, arr[1]=%d, *p=%d", arr[0], arr[1], *p); return 0; }	Output: arr[0] = 10, arr[1] = 20, *p = 20

Example-1.93 What will be output of following C code?

Code	Output
#include <stdio.h> int main(void) { int arr[]={10, 20}; int *p=arr; *++p; printf("arr[0]=%d, arr[1]=%d, *p=%d", arr[0], arr[1], *p); return 0; }	Output: arr[0] = 10, arr[1] = 20, *p = 20

Void Pointer in C

A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typcasted to any type.

```
int a = 10;
char b = 'x';
void *p = &a;          // void pointer holds address of int 'a'
p = &b;                // void pointer holds address of char 'b'
```

Advantages of void pointers:

1. malloc() and calloc() return void * type and this allows these functions to be used to allocate memory of any data type (just because of void *)

```
int main(void)
{
   // Note that malloc() returns void * which can be
   // typecasted to any type like int *, char *,...
   int *x = malloc(sizeof(int)*n);
}
```

2. void pointers in C are used to implement generic functions in C.

Facts about Void

1. void pointers cannot be dereferenced.

Example - 1.94 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `int main()` `{` `  int a = 10; void *ptr = &a;` `  printf("%d", *ptr);` `  return 0;` `}`	Compiler Error: 'void*' is not a pointer-to-object type

Solution to the above problem.

Example - 1.95 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `int main()` `{` `  int a = 10; void *ptr = &a;` `  printf("%d", *(int *)ptr);` `  return 0;` `}`	Output : 10

2. The C standard doesn't allow pointer arithmetic with void pointers. However, in C it is allowed by considering the size of void is 1.

Example-1.96 What will be output of following C code?

Code	Solution
`#include<stdio.h>` `int main()` `{` `  int a[2] = {1, 2};` `  void *ptr = &a;` `  ptr = ptr + sizeof(int);` `  printf("%d", *(int *)ptr);` `  return 0;` `}`	Output : 2

What are the Default Values of Static Variables in C?

In C, if an object that has static storage duration is not initialized explicitly, then:

- if it has pointer type, it is initialized to a NULL pointer;
- if it has arithmetic type, it is initialized to (positive or unsigned) zero;

Memory Leak

Memory-leak generally occure when programmer forget to delete the memory i.e., free (m), created in heap area using malloc (m).

Example-1.97 What will be output of following C code?

Code	Solution
`#include <stdlib.h>` `void f()` `{` `  int *ptr = (int *) malloc(sizeof(int));` `  return;` `}`	Output : Here there is memory leak problem because we had not freed the memory which is created in heap area.

Example-1.98 What will be output of following C code?

Code	Solution
`#include <stdlib.h>` `void f()` `{` `  int *ptr = (int *) malloc(sizeof(int));` `  free(ptr);` `  return;` `}`	Output : The memory leak problem which was there in above example is resolved by making free (ptr).

1.15.3 Arithmetic Operation with Pointer in C Programming

Rule 1: Addition arithmetic with pointers

Address + Number= Address — Address – – Number= Address

Address++ = Address — Address– – = Address

++Address = Address — – –Address = Address

If we will add or subtract a number from an address result will also be an address.

New address will be:

New Address = Old Address + Number × Size of Data Type Which Pointer Is Pointing.

New Address = Old Address – Number × Size of Data Type Which Pointer Is Pointing.

Example-1.99 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main()` `{ int *ptr=(int*)1000;` `ptr=ptr+1;` `printf("%u",ptr);` `return 0;` `}`	Output: 1002 Suppose address of ptr is 1000 the next value of address of ptr is 1000 + 2 = 1002.

Example-1.100 What will be output of following program?

Code	Solution
`#include <stdio.h>` `typedef union abc` `{ char near*a;` `long double d;` `unsigned int i;` `} ABC;` `int main()` `{ ABC *ptr=(ABC *)1000;` `ptr=ptr-4;` `printf("%u",ptr);` `return 0;` `}`	Output: 960 Here ptr is union (one character = 10 byte) which contain value 1000 the statement ptr = ptr – 4; makes its value 1000 – 40 = 960.

Rule 2: Difference arithmetic with pointers

Address – Address = Number

If you will subtract two pointers result will be a number but number will not simple mathematical subtraction of two addresses but it follow following rule.

If two pointers are of same type then:

Address2– Address1= (Simple Substruction of Two Address) / Size of Data Type Which Pointer Point.

Example-1.101 What will be output of following program?

Code	Solution
`#include <stdio.h>` `int main(){` `int *p=(int *)1000;` `int *temp;` `temp=p; p=p+2;` `printf("%u %u\n",temp,p);` `printf("difference= %d",p-temp);` `return 0;` `}`	Output: 1000 1004 Difference = 2 Here two pointer p and temp are of same type and both are pointing to int data type variable. p-temp = (1004 – 1000)/size of (int) = 4/2 = 2

Rule 3: illegal arithmetic with pointers

Address + Address=illegal Address * Address=illegal

Address / Address=illegal Address % Address=illegal

Rule 4: Bit wise arithmetic with pointers

We can perform bitwise operation between two pointers like

Address & Address=illegal Address | Address=illegal

Address ^ Address=illegal ~Address=illegal

Rule 5: We can find size of a pointer using sizeof operator.

***Example*-1.102** What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int *p;` `  printf("%d",sizeof(p));` `  return 0;` `}`	Output: 4 Size of pointer will always be not same. We assume it 4 B.

1.15.4 Pointer to Function in C Programming

A pointer which keeps address of a function is known as function pointer.

***Example*-1.103** What will be output of following program?

Code	Solution
`#include<stdio.h>` `int * function();` `int main(){` `  auto int *x;` `  int (*ptr)();` `  ptr=&function;` `  x=(*ptr)();` `  printf("%d",*x);` `  return 0;` `}` `int *function(){` `  static int a=10;` `  return &a;` `}`	Output: 10 Here function is function whose parameter is void data type and return type is pointer to int data type. X = (*ptr)() ⇒ x = (*&functyion)() //ptr=&function ⇒ x = function() //From rule*&p=p ⇒ x = &a So, *x = *&a = a =10

Example-1.104 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int find(char);` `int(*function())(char);` `int main(){` `  int x;` `  int(*ptr)(char);` `  ptr=function();` `  x=(*ptr)('A');` `  printf("%d",x);` `  return 0;` `}int find(char c){` `  return c;` `}int(*function())(char){` `  return find;` `}`	Output: 65 Here function whose name is function which passing void data type and returning another function whose parameter is char data type and return type is int data type. x = (*ptr)('A') ⇒ x = (*function ()) ('A') //ptr=function () //&find=function () i.e. return type of function () ⇒ x = (* &find) ('A') ⇒ x = find ('A') //From rule*&p=p ⇒ x = 65

1.15.5 Pointer to Array of Function in C

Array of function means array whose content is address of function and pointer to array of function means pointer is pointing to such array. In other word we can say pointer to array of functions is a pointer which is pointing to an array whose contents are pointers to a function.

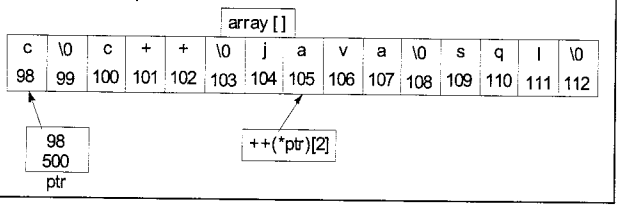
Example-1.105 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int display();` `int(*array[3])();` `int(*(*ptr)[3])();` `int main(){` `  array[0]=display;` `  array[1]=getch;` `  ptr=&array;` `  printf("%d",(**ptr)());` `  (*(*ptr+1))();` `  return 0;` `}` `int display(){` `  int x=5;` `  return x++;` `}`	Output: 5 Array []: It is array of pointer to such function which parameter is void and return type is int data type. ptr: It is pointer to array which contents are pointer to such function which parameter is void and return type is int type data. (**ptr)() = (** (&array)) () //ptr=&array = (*array) () // from rule *&p=p = array [0] () //from rule *(p+i)=p[i] = display () //array[0]=display (*(*ptr+1))() = (*(*&array+1))() //ptr=&array = *(array+1) () // from rule *&p=p = array [1] () //from rule *(p+i)=p[i] = getch () //array[1]=getch

1.15.6 Pointer to Array of String in C Programming

Pointer to array of string: A pointer which pointing to an array which content is string, is known as pointer to array of strings.

Example-1.106 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `char*array[4]={"c","c++","java",` `"sql"};` `char *(*ptr)=&array;` `printf("%s",++(*ptr)[2]);` `return 0;` `}`	Output: ava ptr: It is pointer to array of string of size 4. array[4]: It is an array and its content are string. Pictorial representation: 

In the above example upper part of box represent content and lower part represent memory address. We have assumed arbitrary address.

++(*ptr)[2]

=++(*&array)[2] //ptr=&array

=++array[2]

=++"java"

="ava" //Since ptr is character pointer so it will increment only one byte

%s is used to print stream of characters up to null (\0) character.

1.15.7 Pointer to Structure in C Programming

Pointer to structure: A pointer which is pointing to a structure is known as pointer to structure.

Example-1.107 What will be output of following program?

Code	Solution
`#include<stdio.h>` `struct address{` `char *name;` `char street[10];` `int pin;` `} cus={"madeeasy","H-2",456003};` `*p=&cus;` `int main(){` `printf("%s%s",p →name,(*p).street);` `return 0;` `}`	Output: madeeasy H-2 p is pointer to structure address. → and (*). Both are same thing. These operators are used to access data member of structure by using structure's pointer.

1.15.8 Pointer to Union in C Programming

A pointer which is pointing to a union is know as pointer to union.

Example-1.108 What will be output of following program?

Code	Solution
`#include<stdio.h>` `union address{` `  char *name;` `  char street[10];` `  int pin;` `};` `int main()` `{  union address emp,*p;` `  emp.name="ja\0pan";` `  p=&emp;` `  printf("%s%s",p →name,(*p).name);` `  return 0;` `}`	Output: ja ja p is pointer to union address. → and (*). Both are same thing. These operators are used to access data member of union by using union's pointer. %s is used to print the string up to null character i.e. '\0'

1.15.9 Multilevel Pointers in C Programming

A multi level pointer is pointer to another pointer which is pointing to other pointers i.e. pointer A points to some address which contain address of another location and this location contain address of any other location. We can have any level of pointers.

Example-1.109 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int s=2,*r=&s,**q=&r,***p=&q;` `  printf("%d",p[0][0][0]);` `  return 0;` `}`	Output: 2 As we know p[i] =*(p+i) So, p[0][0][0] = *(p[0][0]+0)=**p[0]=***p Another rule is: *&I = i So, ***p=***(&q) =**q=**(&r) =*r=*(&s)=s=2

1.15.10 Pointer to Array of Character in C

A pointer to such an array whose contents are character constants is known as pointer to array of character constant.

Example-1.110 What will be output of following program?

Code	Solution
`#include<stdio.h>` `char display(char (*)[]);` `int main(){` `  char c;` `  char character[]={65,66,67,68};` `  char (*ptr)[]=&character;` `  c=display(ptr);` `  printf("%c",c);` `  return 0;` `}`	Output: C Here function display is passing pointer to array of characters and returning char data type. **s+=2 ⇒ **s=**s+2 ⇒ **ptr=**ptr+2 //s=ptr ⇒ **&character=**&character+2 //ptr=&character ⇒ *character=*character+2 //from rule *&p =p

`char display(char (*s)[]){` `  **s+=2;` `  return **s;` `}`	⇒ character[0]=character[0]+2 //from rule *(p+i)=p[i] ⇒ character [0] =67 **s=character [0] =67

NOTE: ASCII value of 'C' is 67.

1.16 Sequence Points in C

A sequence point is defined as a point in the execution of C program at which it is sure that, all side effects of previous evaluations are performed, and no side effects from subsequent evaluations have yet been performed.

Example-1.111 What will be output of following C code?

Code	Solution
`#include <stdio.h>` `int f1()` `{ printf ("made");` `  return 1;` `}` `int f2()` `{ printf ("easy");` `  return 1;` `}` `int main()` `{ int p = f1() + f2();` `  return 0;` `}`	Output : Either f1() or f2() may be executed first. So output can be either of "madeeasy" or "easymade".

The operators '+', most of the other operators like '–', '/', '*', Bitwise AND '&', Bitwise OR 'I',... etc., does not have any predefined standard for evaluation for its operands.

Example-1.112 What will be output of following C code?

Code	Solution
`#include <stdio.h>` `int x = 20;` `int f1()` `{ x = x+10;` `  return x;` `}` `int f2()` `{ x = x-5;` `  return x;` `}` `int main()` `{ int p = f1() + f2();` `  printf ("p = %d", p);` `  return 0;` `}`	Output : Depending on the order of expression evaluation, if f1() executes first, the value of p will be 55, otherwise 40.

Example-1.113 What will be output of following C code?

Code	Solution
`#include <stdio.h>` `int main()` `{` `  int i = 8;` `  int p = i++*i++;` `  printf("%d\n", p);` `}`	Output : The output of this program is also undefined. It may be 64, 72, or may be something else depending on the order of evaluation.

1.17 Declarations and Notations

- **int*p;** p can be a pointer to an integer.
- **int*p[10];** p is a 10 element array of pointers to integer.
- **int (*p)[10];** p is a pointer to a 10-element integer array.
- **int *p();** p is a function that returns a pointer to an integer.
- **int p(char*a);** p is a function that takes an argument as pointer to a character and returns an integer.
- **int*p(char *a);** p is a function that takes an argument as pointer to a character and returns a pointer to an integer.
- **int (*p)(char *a);** p is a pointer to a function that takes an argument as pointer to a character and returns an integer.
- **int (*p(char *a))[10];** p is a function that takes an argument as pointer to a character and returns a pointer to a 10 element integer array.
- **int p(char (*a)[]);** p is a function that takes an argument as pointer to a character array and returns an integer.
- **int p(char *a[]);** p is a function that takes an argument as array of pointers to characters and returns an integer.
- **int *p(char a[]);** p is a function that takes an argument as character array and returns a pointer to an integer.
- **int *p(char (*a) []);** p is a function that takes an argument as pointer to a character array returns a pointer to an integer.
- **int *p(char *a[]);** p is a function that takes an argument as array of pointers to characters and returns a pointer to an integer .
- **int (*p)(char (*a)[]);** p is a pointer to a function that takes an argument as pointer to a character array and returns an integer.
- **int *(*p)(char (*a)[]);** p is a pointer to a function that takes an argument as pointer to a character array and returns a pointer to an integer.
- **int *(*p)(char *a[]);** p is a pointer to a function that takes an argument as array of pointers to characters and returns a pointer to an integer.
- **int (*p[10])();** p is a 10 element array of pointers to functions, each function returns an integer.

- int (*p[10])(char a); p is a 10 element array of pointers to functions each functions takes an argument as character, and returns an integer.
- int *(*p[10])(char a); p is a 10 element array of pointers to functions, each function takes an argument as character, and returns a pointer to an integer.
- int *(*p[10])(char *a); p is a 10 element array of pointers to functions, each function takes an argument as pointer to a character, and returns a pointer to an integer.

Summary

- **'Auto' Storage Class:** The auto storage class is implicitly the default storage calls used simply specifies a normal local variable which is visible within its own code block only and which is created and destroyed automatically upon entry and exit respectively from the code block.
- **'Static' Storage Class:** The static storage class causes a local variable to become permanent within its own code block i.e. it retains its memory space and hence its value between function calls.
- **'Register' Storage Class:** The register storage class also specifies a normal local variable but it also requests that the compiler store a variable so that it may be accessed as quickly as possible, possible from a CPU register.
- **'Extern' Storage Class:** An extern class which informs the compiler for the existence of the global variable which enables it can be accessed in more that one source file.
- **Control Statements:** 1. **if** statement, 2. Nested **if** statement, 3. Conditional operator?, 4. **switch** statement, 5. **for** statement, 6. **while** statement, 7. **do-while** statement, 8. **break** statement and 9. **continue** statement.
- *Call by value:* "Actuals copied to formal", but not formals to actuals.
- *Call by reference:* "Actuals and formals uses same address space".
- *Call by value result (Restore):* "Actuals copied to formals and formals copied to actuals".
- *Call by Result:* "Actuals not copied to formals", but formals copied to actuals.
- *Call by Constant:* Formal values are never changed.
- *Call by Name:* The textual substitution of every occurrence of a formal parameter in the called routines body by the corresponding actual parameter.
- *Call by Text:* Same as call by name, but if variables are named same as local variables then it prefers local.
- *Call by Need:* Memory allocates only if formals are used in function.
- **Static Scoping:** The method of binding names to non local variables called static scoping. Scope of variable can be statically determined prior to execution.
- **Dynamic Scoping:** It is based on the calling sequence of subprograms. Scope can be determined at runtime.
- **Pointer:** A pointer is a variable that is used to store a memory address of another variable in memory. If one variable holds the address of another then it is said to the second variable.

Student's Assignment

Q.1 Consider the following C-program

```
int f(int num)
{
int result = 0:
if (num <=1)
  return 1;
      else
{
for (i=num;i>=1;i—)
{
Result+=f(i/3);
}
}
return result;
}
```

When num = 6 then the return value of the function is _____

Q.2 What is the meaning of the following declaration?

int * p(char *a[]);

(a) P is a pointer to a function that accepts an argument which is a pointer to a character array and returns an integer quantity.

(b) P is a function that accepts an argument which is an array of pointers to characters and returns a pointer to an integer quantity.

(c) P is a function that accepts an argument which is an array of pointers to characters and returns an integer quantity.

(d) P is a pointer to a function that accepts an argument which is an array of pointers to characters and returns a pointer to an integer quantity.

Q.3 Consider the following program:

```
#define funct(x) x*x+x
int main()
{
  int x;
  x=36+funct(5)*funct(3);
  printf("%d",x);
  return 0;
}
```

What will be the output of the above program?

(a) 73 (b) 396

(c) 109 (d) 360

Q.4 Consider the following function given below:

```
int function(int n){
  if(n-1)
  return 2*function(n-1)+n;
  else
  return 0;
}
```

What is the value returned by function (5)?

(a) 33 (b) 41

(c) 57 (d) 65

Q.5 What is the output of the following program if dynamic scoping is used?

```
int a, b, c;
void func1(){
  int a,b;
  a=6;
  b=8;
  func2();
  a=a+b+c;
  print(a);
}
  void func2(){
  int b,c;
  b=4;
  c=a+b;
  a+= 11;
  print(c);
}
  void main(){
  a=3;
  b=5;
  c=7;
  func1();
}
```

Output of program:

(a) 7 19 (b) 10 1

(c) 10 23 (d) 10 32

Q.6 Output of the following program will be

```
int main
{
  int i,a[8]=000, 001, 010, 011, 100,
  101, 110, 111;
```

```
  for(i=0;i=8;i++)
  printf("%d",a,i);
  return 0;
}
```

(a) 0,1,2,3,4,5,6,7
(b) 0,1,10,11,100,101,110,110
(c) 0,1,8,9,100,101,110,111
(d) None of these.

Q.7 Assuming only numbers and letters given in the input, what does the following program do?

```
#include<stdio.h>
int main()
{
  int i,j, ascii[128];
  char ip[30];
  printf("Enter Input string: ");
  scanf("%s",ip);
  for(i=0;i<128;i++)
{
  ascii[i]=0;
}
  i=0;
  while(ip[i]!='\0')
{
  j=(int)ip[i];
  ascii[j]++;
  if(ascii[j]>1)
{
  printf("%c",ip[i]);
  return 0;
}
   i++;
}
  return 0;
}
```

(a) Prints the position of first repeated character in the string
(b) Prints the first repeated character in the string
(c) Prints the position of last repeated character in the string
(d) Prints the last repeated character in the string

Q.8 Consider the following code segment.

```
void foo(int x, int y){
  X+=y;
  Y+=x;
}
  main(){
  int x=4.5;
  foo(x,x);
}
```

What is the final value of x in both call by value and call by reference respectively?

(a) 4 and 12 (b) 5 and 12
(c) 12 and 16 (d) 4 and 16

Q.9 What will be the output of following C-code?

```
int main(){
  int a[3]={67,43,23};
  int *p=a;
  printf("%d",++*p);
  printf("%d",*++p);
  printf("%d",*p++);
  return 0;
}
```

(a) 68 43 43 (b) 37 43 43
(c) 67 43 23 (d) 68 43 23

Q.10 Consider following C-code.
What is the value returned by fun (10)?

```
int fun (int n){
  Static int i=10;
  if (n>=200)
  return (n+i);
  else{
      n=n+i;
      i=n+i;
      return fun(n);
    }
}
```

(a) 890 (b) 210
(c) 340 (d) none of these.

Q.11 What will be the output of following C-code?

```
void fun(void *p);
static int i;
int main()
{
  void *ptr;
  ptr &i;
  fun ptr ;
  return 0;
}
  void fun( void *p)
{
```

```
int **q;
q=(int**)&p;
printf("%d",**q);
```

(a) Garbage value (b) 0
(c) 1 (d) compile-time error

Q.12 Output of the following program will be

```
int main()
{
 char *p1="Graduate";
 char *p2=p1;
 printf ("%c,%d",*++p2,sizeof(p2));
}
```

(a) G, 8 (b) r, 7
(c) r, 4 (d) r, 8

Q.13 Consider the following program.

```
begin
integer m, n;
Procedure gate();
 begin
 print("gate n=",n);
 end;
Procedure exam(n:integer);
 begin
 print("exam m=",m);
 print("exam n=", n);
 gate();
 end;
m=5;
n=1;
print("main n=",n);
exam(10);
gate();
end;
```

Find the output using static scope?

(a) main n = 1
exam m = 5
exam n = 10
gate n = 1
gate n = 1

(b) main n = 1
exam m = 5
exam n = 1
gate n = 1
gate n = 10

(c) main n = 1
exam m = 5
exam n = 10
gate n = 10
gate n = 1

(d) None of these

Q.14 What is the output of following program?

```
int f(int a)
{  printf("%d", a++);
   return(++a);
}
   main()
{
   int b=1;
   b=f(b);
   b=f(b);
   b=f(1+f(b));
}
```

(a) 1 3 3 5 (b) 1 3 5 7
(c) 2 3 3 5 (d) 2 4 6 8

Q.15 Consider the following C code

```
int*P,A[3]={0,1,2};
P=A;
*(P+2)=5;
P=A++;
*P=7;
```

What are the values stored in the array A from index 0 to index 2 after execution of the above code?

(a) 7, 5, 2 (b) 7, 1, 5
(c) 0, 7, 5 (d) None of these

Q.16 Consider the following code:

```
int f(int a, int b)
{
   if(b==0) return 1;
   else if(b%2==0)
   {
      return(f(a,b/2)*f(a,b/2));
   }
   else
   {
      return(a*f(a,b/2)*f(a,b/2));
   }
}
```

What is the return value of f(2, 10)?

Answer Key:

1. (7)	**2.** (b)	**3.** (c)	**4.** (b)	**5.** (a)
6. (c)	**7.** (b)	**8.** (d)	**9.** (a)	**10.** (a)
11. (c)	**12.** (c)	**13.** (a)	**14.** (b)	**15.** (d)
16. (1024)				

■■■■

CHAPTER 02

Arrays

2.1 Definition of Array

- A collection of items having same data type stored in contiguous memory allocation.
- An array is derived data type in c programming language which can store similar type of data in continuous memory location. Data may be primitive type (int, char, float, double...), address of union, structure, pointer, function or another array.
- **Importance:** Array implementation is important because:
  (a) Most assembly languages have no concept of arrays.
  (b) From an array, any other data structure we might want, can be built.

2.2 Properties of Array

- Each element is of the same size (char = 1 byte, integer = 1 word)
- elements are stored continuously, with the first element stored at the smallest memory address.

NOTE: Thus the whole trick in assembly language is **(a)** To allocate correct amount of space for an array and **(b)** An address tells the location of an element.

Example: `int arr[5];` , `char arr[5];` , `float arr[5];` , `long double arr[5];` , `char * arr[5];` , `int (arr[])();` and `double ** arr[5];`

2.2.1 Array is useful when

We have to store large number of data of similar type. If we have large number of similar kind of variable then it is very difficult to remember name of all variables and write the program. Below program without array and its equivalent program with array is shown.

Program without Array	Program with Array
`#include<stdio.h>` `int main(){` `  int ax=1, b=2, cg=5, dff=7,` `  am=8, raja=0, rani=11, xxx=5,` `  yyy=90, avg;` `  avg=(ax+b+cg+dff+am+raja+rani` `  +xxx+yyy)/12;` `  printf("%d",avg);` `  return 0;` `}`	`#include<stdio.h>` `int main(){` `  int arr[]={1,2,5,7,8,0,11,5,90};` `  int i,avg;` `  for(i=0;i<12;i++){` `  avg=avg+arr[i];` `  }` `  printf("%d",avg/12);` `  return 0;` `}`

2.2.2 Advantage of Using Array

1. An array provides single name .So it easy to remember the name of all element of an array.
2. Array name gives base address of an array .So with the help increment operator we can visit one by one all the element of an array.
3. Array has many application data structure.

2.2.3 Array of Pointers in C

Array whose content is address of another variable is known as array pointers.

Example-2.1 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  float a=0.0,b=1.0,c=2.0;` `  float *arr[]={&a,&b,&c};` `  b=a+c;` `  printf("%f",b);` `  return 0;` `}`	Output: 2.0 Here variable a, b, c are float data type. Arr[ ] is pointer array which store address of variable a, b, c then printf function will print the value of 'b' after addition of 'a' and 'c'.

2.2.4 Complex Arrays in C

- Declaration of an array of size five which can store address such functions whose parameter is void data type and return type is also void data type: `void(arr[5])();`
- Declaration of an array of size five which can store address such function which has two parameter of int data type and return type is float data type: `float(arr[5])(int,int);`
- Declaration of an array of size two which can store the address of printf or scanf function: `int(arr[2])(const char*,...);`

NOTE: Prototype of printf function is: `int printf(const char*,...);`

2.2.5 Different Type of Array in C

- **Array of integer:** An array which can hold integer data type is known as array of integer.
- **Array of character:** An array which can hold character data type is known as array of character.
- **Array of union:** An array which can hold address of union data type is known as union of integer.

Example-2.2 What will be output of following program?

Code	Solution
`#include<stdio.h>` `union A{` `  char p;` `  float *const q;` `};` `int main(){` `  union A arr[10];` `  printf("%d",sizeof arr);` `  return 0;` `}`	Output: 40 Here 'Union' is used to create the user define data type i.e. 'A' which take 4 byte i.e. take maximum size in the definition of union. Array of union contain 10 elements. So size = 10 × 4 = 40

- **Array of structure:** An array which can hold address of structure data type is known as array of structure.

Example-2.3 What will be output of following program?

Code	Solution
`#include<stdio.h>` `typedef struct madeeasy{` `  char *name;` `  int roll;` `  }s;` `int main(){` `  s arr[2]={{"made",10},{"easy",11}};` `  printf("%s %d",arr[0]);` `  return 0;` `}`	Output: made 10 Here we create new data type 's' by using structure which has to fields i.e. (pointer) name and roll. On printf function first entry i.e. made (name), second entry 10 (roll) will be printed.

- **Array of string:** An array which can hold integer data type is known as array of integer.
- **Array of array:** An array which can hold address of another array is known as array of array.
- **Array of address of integer:** An array which can hold address integer data type is known as array of address of integer.

2.2.6 Pointer to Array

A pointer which holds base address of an array or address of any element of an array is known as pointer to array.

Example-2.4 What will be output of following program?

Code	Solution
`#include<stdio.h>` `int main(){` `  int arr[5]={100,200,300};` `  int *ptr1=arr;` `  printf("%d",*(ptr1+2),);` `return 0;` `}`	Output: 300 Here arr (array) contain 3 values i.e. 100, 200, 300 and integer pointer ptr1 contain address of arr. Ptr1+2 i.e. skip to elements in the array from starting location of the array. Hence answer will be 300.

2.3 Accessing Elements of an Array

In general we need to know: (a) Where the array starts (called the Base address) (b) Size of an element in bytes (to get a byte address) and (c) What the first element is numbered.

2.3.1 One Dimensional Array

Byte address of element [X] = base address + size (x-first index)

Example-2.5 A[–5 ... + 5], Base address (BA) = 999, Size of element = 100 bytes.

Find the location of A[–1], A[+5]?

Solution:

$$L(A[-1]) = 999 + [(-1) - (-5)] \times 100 = 999 + 100 \times 4 = 1399$$

$$L(A[5]) = 999 + [5 - (-5)] \times 100$$

$$= 999 + 10 \times 100 = 1999$$

NOTE: Total number of element = Last Index – First Index + 1

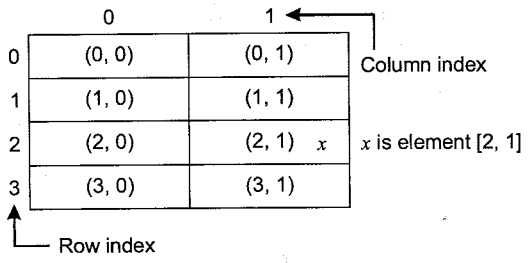
2.3.2 Two-Dimensional Arrays

How to map a 2-D array into a 1-D array memory:

Terminology $r \times c$ array, ***where*** r = rows and c = columns

element $[y, x]$, ***where*** y is row number and x is column number.

Example: 4 × 2 arrays



Mapping this 4 × 2 array into memory.

There are two possibilities:

- **Row Major Ordering:** Rows are all together.

0, 0	0, 1	1, 0	1, 1	2, 0	2, 1	3, 0	3, 1

If array is declared by a $[r_1][r_2]$ then to access the element ar $[i_1]$ $[i_2]$ we have base (ar) + $(i_1 * r_2 + i_2) *$ e-size. ***where*** base (ar) = Base address of array.

e-size = Memory size allocated to data type of array.

(Row-major order) LOC(A[J, K]) = Base(A) + w[N(J – 1) + (K – 1)]

where Base(A): The address of the first element A[0, 0] of A.

Example-2.6 a[5...25, 50...75], Base address (BA) = 0, Size of element = 5 bytes

Find the location of a[20] [70]?

Solution:

$$LOC(a[20][70]) = 0 + ((20 - 5) \times (75 - 50 + 1) + (70 - 50)) \times 5$$

$$= 0 + (15 \times 26 + 20) \times 5$$

$$= (390 + 20) \times 5 = 2050$$

- **Column Major Ordering:** Columns are all together.

0, 0	1, 0	2, 0	3, 0	0, 1	1, 1	2, 1	3, 1

$$LOC(A[J, K]) = Base(A) + w[M(K - 1 + (J - 1)].$$

Example-2.7 A[–5...+20] [25...75], Base address (BA) = 999, Size of element = 10 bytes. Find the location of a[0] [74]?

Solution:

$$\begin{aligned} LOC(A[0][74]) &= 999 + ((74 - 25) \times (20 - (-5) + 1) + (0-(-5))) \times 10 \\ &= 999 + (49 \times 26 + 5) \times 10 \\ &= 999 + 12790 = 13789 \end{aligned}$$

Bound Checking: Many high level language's offer some form of bounds checking. Your program crashes, or you get an error message if an array index is out of bounds.

Examples: x : array [1...6] of integer;

code....

y=x [8];

Assembly languages offer no implied bounds checking.

Reason: If your program calculates an address of an element, and then loads that element (by the use of the address), there is no checking to see that the address calculated was actually within the array.

2.3.3 Multidimensional Array

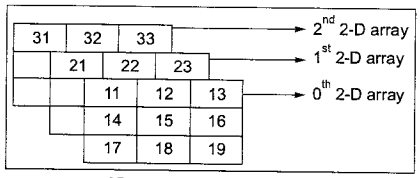
C allows array of two or more dimensions and maximum number of dimension a C program an have is depend upon the compiler we are using.

Syntax: Type array_name $[d_1]$ $[d_2]$ $[d_3]$ ... $[d_n]$ ***where*** d_n = size of last element

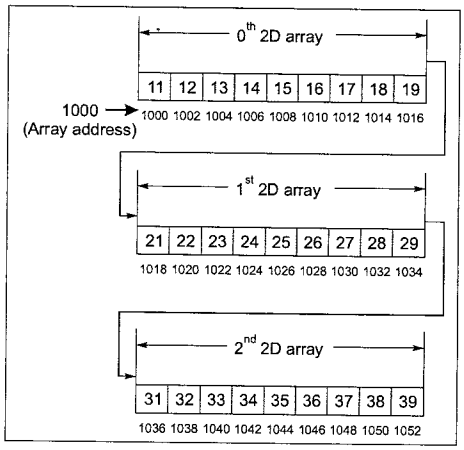
Example: int table [5] [5] [20]; Array "table" is a 3D (A 3D array is an array of arrays of arrays) array which can hold 500 integer type element.

NOTE: To make this multidimensional array example simple we will discuss 3D array for the sake of simplicity. Once you get the logic how 3D array works than you can handle 4D array or any multidimensional array easily.

Declaration and initialization 3D array: A 3D array (collection) of 2D arrays and you know 2D array itself is array of 1-D array.



3D Array Conceptual View



3D Array Actual View

2.3.4 Lower Triangular Matrix

A lower triangular matrix L

$$L_{i,j} = \begin{cases} a_{i,j} & \text{for } i \geq j \\ 0 & \text{for } i < j \end{cases}$$

$$\text{Written explicitly } L = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

There are two possibilities:

- **Row Major Ordering:** Rows are all together if array is declared by $\underset{r_1}{a[lb_1.....ub_1]}$ $\underset{r_2}{[lb_2.....ub_2]}$ array is stored at location.

$$\text{Loc}[a_{i,j}] = ar + \left[\frac{r_1 * (r_1+1)}{2} + (j - lb_2)\right] * \text{esize}$$

Where
ar = Base address of an array
e-size = Memory size allocated to data type of array
r_1 = Number of rows and r_2 = Number of columns

Example - 2.8 a[–25...+25, –25...+25], Base address (BA) = 0, size of element = 100 bytes.

Find the location of a[–20] [–21]?

Solution:

$$\text{LOC}(a[-20][-21]) = 0 + \left(\frac{(-20+25)(-20+25+1)}{2} + (-21+25)\right) \times 100$$
$$= 0 + \left(\frac{5 \times 6}{2} + 4\right) \times 100$$
$$= 0 + (19) \times 100 = 1900$$

- **Column major ordering:** Columns are all together of array is declared by $\underset{r_1}{a[lb_1.....ub_1]}$ $\underset{r_2}{[lb_2.....ub_2]}$ array is stored at location

$$\text{LOC}[a_{i,j}] = ar + \left(\frac{r_2(r_2+1)}{2} - \frac{(ub_2 - j + 1)(ub_2 - j + 1 + 1)}{2} + (j - i)\right) * e\text{-size}$$

Where
ar = Is base address of an array
e-size = Memory size allocated to data type of array
r_1 = Number of row and r_2 = Number of column

Example - 2.9 a[–5...+5, –5...+5], Base address (BA) = 1000, size of element = 10 bytes.

Find the location of a[2] [–1]?

Solution:

$$\text{LOC}(a[2][-1]) = 1000 + \left(\frac{(5-(5)+1)(5-(-5)+2)}{2} - \frac{(5-(-1)+1)(5-(-1)+2)}{2} + (2-(-1))\right) \times 10$$
$$= 1000 + (66 - 28 + 3) \times 10$$
$$= 1000 + (41) \times 10 = 1410$$

2.3.5 Upper Triangular Matrix

A upper triangular matrix U of form

$$U_{ij} = \begin{cases} a_{i,j} & \text{for } i \le j \\ 0 & \text{for } i > j \end{cases}$$

Written explicitly

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & & \cdots & a_{nn} \end{bmatrix}$$

There are two possibilities

- **Row Major Ordering:** Rows are all together if array is declared by $\underset{r_1}{a[lb_1 \ldots ub_1]}\underset{r_2}{[lb_2 \ldots ub_2]}$ array is stored at location

$$\text{Loc}[a_{i,j}] = ar + \left[\frac{r_1(r_1+1)}{2} + (j-i)\right] * e\text{-size}$$

Example-2.10 a[–25...+25, –25...+25], Base address (BA) = 0, Size of element = 100 bytes.

Find the location of a[20] [22]?

Solution: $\text{LOC}(a[20][22]) = 0 + \left[\frac{(25-(-25))(25-(-25)+1)}{2} + (22-20)\right] \times 100$

$$= 0 + \left[\frac{50 \times 51}{2} + 2\right] \times 100$$
$$= 0 + [1275 + 2] \times 100$$
$$= 1277 \times 100$$
$$= 127700$$

- **Column Major Ordering:** Columns are all together of array is declared by $\underset{r_1}{a[lb_1 \ldots ub_1]}\underset{r_2}{[lb_2 \ldots ub_2]}$ array is stored at location.

$$\text{LOC}[a_{i,j}] = ar + \left[(j - lb_2)\frac{(j + lb_2 + 1)}{2} + (i - lb_2)\right] * e\text{-size}$$

Example-2.11 a[–25...+25, –25...+25], Base address (BA) = 0, Size of element = 100 bytes.

Find the location of a[20] [22]?

Solution: $\text{LOC}(a[20][22]) = 0 + \left(\frac{(22-(-25))(22-(-25)+1)}{2} + (20-(-25))\right) \times 100$

$$= 0 + \left(\frac{2256}{2} + 45\right) \times 100$$
$$= (1128 + 45) \times 100$$
$$= 117300$$

NOTE: All the notations same as the lower triangular matrix.

2.3.6 Strictly Lower Triangular Matrix

A lower triangular matrix having as along with the diagonal as well as the upper portion *i.e.*, a matrix $A = [a_{i,j}]$ such that $a_{i,j} = 0$ for $i \geq j$.

$$\text{Written explicitly } L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}$$

2.3.7 Strictly Upper Triangular Matrix

A upper triangular matrix as along the diagonal as well as the upper portion i.e., a matrix $A = [a_{i,j}]$ such that $a_{i,j} = 0$ for $i \leq j$

$$\text{Written explicit } U = \begin{bmatrix} 0 & a_{12} & \cdots & a_n \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

Summary

- Array is a collection of homogeneous elements stored in contiguous memory location.
- **Properties:** Static in nature, Compile time early binding and user friendly.
- **1-D Array:** Let '*A*' be an array of *n* elements. Address of an element *A*[*i*]:
  Base(A) + (*i* – start index) × size of element.
- **2-D Array:** Let *A*[*m*][*n*] be a 2-*D* array with *m* rows and *n* columns.
  Address of an element *A*[*i*][*j*] in Row Major order:
  Base (*A*) + (*j* – start index) × size of element + (*i* – start index) × size of element × *n*
  Address of an element A[*i*][*j*] in Column Major order:
  Base (*A*) + (*i* – start index) × size of element + (*j* – start index) × *m* × size of element.
- **Insertion:**
  [best case] At end takes constant time i.e. $\Omega(1)$
  [worst case] At beginning takes O(*n*) time
  [Average case] In middle takes $\theta(n)$ time
- **Deletion:**
  [best case] At end takes constant time i.e. $\Omega(1)$
  [worst case] At beginning takes O(*n*) time
  [Average case] In middle takes $\theta(n)$ time
- **Lower Triangular** $[\triangle]_{n\times n}$:

  (*i*) Size = $n + \frac{n^2 - n}{2} = \frac{n(n+1)}{2}$

  (*ii*) Row Major Order (RMO): $(j-1) + \frac{i(i-1)}{2}$

  (*iii*) Column Major Order (CMO): $a[i][j] = (i-j) + \left[(j-1)n - \frac{(j-1)(j-2)}{2}\right]$

- **Upper Triangular** $[\nabla]_{n\times n}$:

  (*i*) Size $= \dfrac{n(n+1)}{2}$

  (*ii*) Row Major Order (RMO): $a[i][j] = (j-i)+\left[(i-1)n-\dfrac{(i-1)(i-2)}{2}\right]$

  (*iii*) Column Major Order (CMO): $a[i][j] = \left[(i-1)+\dfrac{j(j-1)}{2}\right]$

- **Strictly lower triangular** $[\Delta]_{n\times n}$

  (*i*) Size $= \dfrac{n^2-n}{2}$

  (*ii*) Row Major Order (RMO): $a[i][j] = (j-1)+\dfrac{(i-1)(i-2)}{2}$

  (*iii*) Column Major Order (CMO): $a[i][j] = (i-1)+\dfrac{(j-1)(j-2)}{2}$

Student's Assignment

Q.1 Advantage of array is
(a) Linear access (b) Random access
(c) Sequential access (d) All the above

Q.2 Consider the following single dimensional array
int a[5] = {10, 20, 30, 40, 50}
What is location is a[4]
(a) 1000 (b) 1008
(c) 1005 (d) 1002

Q.3 Consider the following single dimensional array declaration
A[1... 1000]
Base address 1000 and size of element 4. Find the location of A[50]
(a) 1186 (b) 1126
(c) 1096 (d) 1196

Q.4 Consider the 2 dimensional array. A [– 8...12, –4...16]. Calculate the address of A[1, 3]. Assume that it is stored in a row major order. Each element occupies 4-byte and starting address of array 2000.
(a) 2780 (b) 2784
(c) 2776 (d) 2782

Q.5
```
Main ()
{
    int a[3] [4] = [1  2  3  4
                    5  6  7  8
                    9 10 11 12]
    printf("\n% u% u% u", a[0] + 1, * (a[0] + 1),
    *(*(a + 0) + 1) ));
}
```
What is output of the above program? Assume array begin at address 10.
(a) 12, 2, 2 (b) 10, 2, 2
(c) 12, 2, 4 (d) 12, 4, 4

Q.6 Professor Pradhumn decides to make quick sort stable by changing each key A[*i*] in array A[1: n] to (n*A[*i*]) + *i* – 1, so that all the new keys are distinct (call the modified array A′[1 : n] and then sorting A′ (Assume A is subset of integers). Then which of the following is true?
(a) A′ contains distinct elements and the original keys can be restored by computing $\left\lfloor \dfrac{A'[i]}{n} \right\rfloor$ for each *i*.
(b) A′ contains distinct elements and the original keys can be restored by computing $\left\lfloor \dfrac{A'[i]}{n} \right\rfloor$ for each *i*.

(c) A′ does not always contain distinct keys.
(d) A′ contains distinct elements but it is not possible to restore original keys.

Q.7 In a compact single dimensional array representation for lower triangular matrices (i.e. all the elements above the diagonal are zero of size n × n, non-zero elements (i.e., elements of the lower triangle) of each row are stored one after another, starting from the first row, the index of the $(i, j)^{th}$ element of the lower triangular matrix in the new representation is

(a) $i + j$ (b) $i + 1 - 1$

(c) $j + \frac{i(i-1)}{2}$ (d) $i + \frac{j(j-1)}{2}$

Q.8
```
Main( )
{
    int a[3] [4] = [1  2  3  4
                    5  6  7  8
                    9  10 11 12]
    print f("\ n% u% u", a +1, & a + 1);
}
```

what is the output of the above program? Assume array begin at address 54572.

(a) 54572, 54580 (b) 54580, 54596
(c) 54572, 54596 (d) 54572, 65490

Q.9 Suppose we want to arrange the n numbers stored in an array such that all negative value occur before all positive ones, minimum number of exchanges required in the worst case is

(a) n – 1 (b) n

(c) n + 1 (d) $\frac{n+(n+1)}{2}$

Q.10 Let A[1 : n] be an array such that A[i] = i. An algorithm randomly permutes the elements of A, call the resulting array A′. Let X denote the number of locations such that A′[i] = i. What is expectation of X?

(a) n^2 (b) n/2
(c) n (d) 1

Answer Key:

1. (b)	**2.** (b)	**3.** (d)	**4.** (c)	**5.** (a)
6. (b)	**7.** (c)	**8.** (b)	**9.** (d)	**10.** (a)

CHAPTER 03

Stack

3.1 Introduction

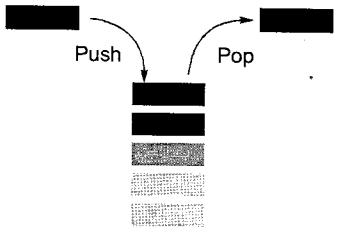
"A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top*". Stacks are sometimes referred to as Last In First Out (LIFO) lists.

3.2 Operation on Stack

Stacks have some useful terminology associated with them:

- **Push** To add an element to the stack
- **Pop** To remove an element from the stack
- **Peek** To look at elements in the stack without removing them
- **LIFO** Refers to the last in, first out behavior of the stack
- **FILO** Equivalent to LIFO

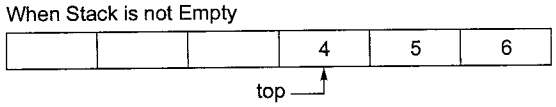
Visual Representation of Stack:



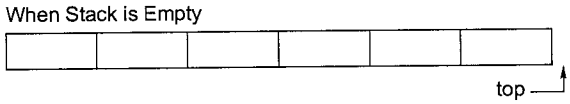
Consider Stack with following details:

Field	Value
Size of the Stack	6
Maximum Value of Stack Top	5
Minimum Value of Stack Top	0
Value of Top when Stack is Empty	–1
Value of Top when Stack is Full	5

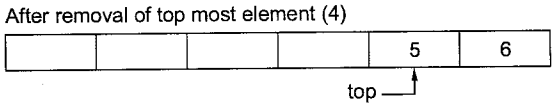
View 1: When Stack is Empty: When Stack is said to empty then it does not contain any element inside it. Whenever the Stack is Empty the position of topmost element is –1.



View 2: When Stack is Not Empty: Whenever we add very first element then topmost position will be incremented by 1. After adding First Element top = 0.



View 3 : After Deletion of 1 Element Top Will be Decremented by 1



NOTE: Stack can be represented either Horizontal or Vertical.

Position of top and its value:

Position of Top	Status of Stack
–1	Stack is Empty
0	First Element is Just Added into Stack
N – 1	Stack is said to Full
N	Stack is said to be Overflow

3.3 Simple Representation of a Stack

Given a stack S = (a[1], a[2],... a[n]) then we say that a1 is the bottom most element and element a[i]) is on top of element a[$i-1$], $1< i \leq n$.

Implementation of stack: (*i*) Array (static memory) and **(*ii*)** Linked list (dynamic memory).

3.4 ADT of Stack

A stack S is an abstract data type (ADT) supporting the following three methods:

- **push(n):** Inserts the item ***n*** at the top of stack.
- **pop( ):** Removes the top element from the stack and returns that top element. An error occurs if the stack is empty.
- **peek( ):** Returns the top element and an error occurs if the stack is empty.

3.5 Operations of Stack

- PUSH operations
- POP operations
- PEEK operations

3.5.1 Adding an Element Into a Stack (PUSH Operations)

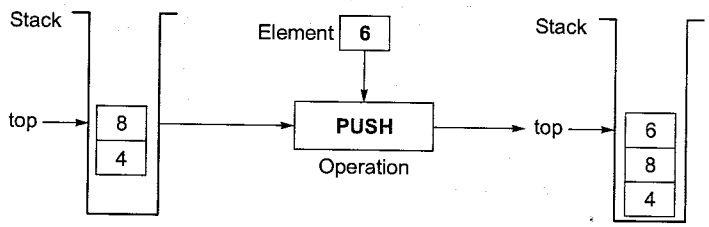
Adding element into the TOP of the stack is called PUSH operation.

1. Push Refers as "Adding Elements onto Stack".
2. Push Operation carried out in following 2 steps:
   (a) First Increment Variable "top" so that it now refers to next memory location.
   (b) Secondly Add Element on that incremented top (add new element at the top of stack).
3. Main Function Should ensure that stack is not full before making call to push() in order to prevent "Stack Overflow".

NOTE: Check conditions: TOP = N, then STACK FULL, where N is maximum size of the stack.

Adding into stack (push algorithm):

```
procedure add (item : items);
{  add item to the global stack; top is the current top of stack
   and n is its maximum size
}
begin
   if top = n then stack full;
   top := top+1;
   stack(top) := item;
end: {of add}
```



Implementation in C using array:

```
/* here, the variables stack, top and size are global variables */
void push (int item)
{
    if (top == size-1)
    printf("Stack is Overflow");
    else
    {
        top = top + 1;
        stack[top] = item;
    }
}
```

NOTE

Stack Overflow condition:

- Pushing element onto stack which is already filled is refer as "Stack Overflow".
- Using array representation of stack ,variable "MAX" is used to keep track of "Number of Maximum Elements".

3.5.2 Deleting an Element from a Stack (POP Operations)

Deleting or Removing element from the TOP of the stack is called POP operations.

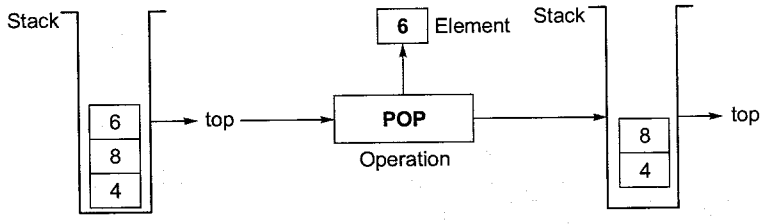
Steps in pop operation:

1. Store Topmost Element in another Variable.
2. Decrement Top by 1
3. Return Topmost Element.

NOTE: Check Condition: TOP = 0, then STACK EMPTY.

Deletion in Stack (pop operation)

```
Procedure delete(var item : items);
{remove top element from the stack stack and put it in the item}
begin
   if top = 0 then stack empty;
   item := stack(top);
   top := top-1;
end; {of delete}
```



Implementation in C using array:

```
/* here, the variables stack, and top are global variables */
int pop( )
{
   if (top == -1){
      printf("Stack is Underflow");
      return (0);
   }else
   {
      return (stack[top - - ]);
   }
}
```

Pop operation arguments and return type:

1. **Argument:** Variable of Type Stack.
2. **Return Type:** Integer [Removed Element]

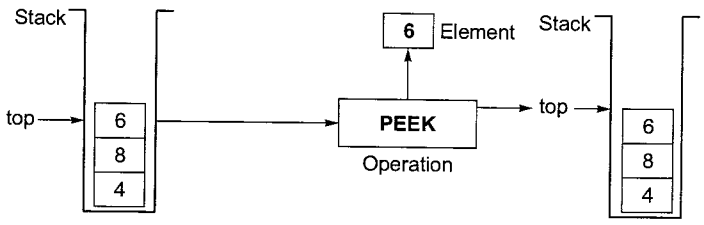
NOTE

Underflow Condition:

1. Before Removing or Popping element from stack ensure that stack is not empty.
2. Removing element from empty stack leads to problem called "**Underflow of Stack**".

3.5.3 Peek Operation

- Returns the item at the top of the stack but does not delete it.
- This can also result in **underflow** if the stack is empty.



PEEK algorithm:

```
PEEK(STACK, TOP)
begin
     /*Check, Stack is empty? */
     if(top == -1) then
          print "underflow" and return 0.
     else
          item = Stack[top] /* stores the top element into a local variable*/
          return item       /* returns the top element to the user*/
end
```

Implementation in C using array:

```
/* here, the variables stack, and top are global variables */
int pop( )
{
     if (top == -1)
     {
          printf("Stack is Underflow");
          return (0);
     }else
     {
          return (stack[top]);
     }
}
```

3.6 Applications of Stack

1. It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
2. Infix to Postfix Transformation
3. It is useful during the execution of recursive programs
4. A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
5. A stack (memory stack) can be used in function calls including recursion.
6. Reversing Data
7. Reverse a List
8. Convert Decimal to Binary

9. Parsing – It is a logic that breaks into independent pieces for further processing
10. Backtracking

1.	Infix notation	A+(B*C)	Equivalent	Postfix notation	ABC* +
2.	Infix notation	(A+B)*C	Equivalent	Postfix notation	AB + C*

3.7 Expression Evaluation and Syntax Parsing

Calculators employing reverse Polish notation (also known as **postfix notation**)use a stack structure to hold values.

Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based machines. Note that natural languages are context sensitive languages and stacks alone are not enough to interpret their meaning.

Infix, Prefix and Postfix Notation

The notations used to represent the arithmetic expressions.

1. **Infix:** <operand> operator <operand>
   Ex: A + B
2. **Prefix (polish notation):** operator <operand> <operand>
   Ex: +AB
3. **Postfix:** <operand> <operand> operator
   Ex: AB+

We are accustomed to write arithmetic expressions with the operation between the two operands: **a+b** or **c/d**. If we write **a+b*c**, however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?).

There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c – d)	* + a b – c d	a b + c d – *

Arithmetic expressions: polish notation

- An arithmetic expression will have operands and operators.
- Operator precedence listed below:
  Highest : ($)
  Next Highest : (*) and (/)
  Lowest : (+) and (–)
- For most common arithmetic operations, the operator symbol is placed in between its two operands. This is called **infix notation**. *Example:* A + B , E * F
- Parentheses can be used to group the operations. *Example:* (A + B) * C

- Accordingly, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.
- Polish notation refers to the notation in which the operator symbol is placed before its two operands. This is called **prefix notation**. *Example:* +AB, *EF
- The fundamental property of polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression.
- Accordingly, one never needs parentheses when writing expressions in Polish notation.
- **Reverse Polish Notation** refers to the analogous notation in which the operator symbol is placed after its two operands. This is called **postfix notation**. *Example:* AB+, EF*
- Here also the parentheses are not needed to determine the order of the operations.
- The computer usually evaluates an arithmetic expression written in infix notation in two steps,
  (a) It converts the expression to **postfix notation**.
  (b) It evaluates the postfix expression.
- In each step, the stack is the main tool that is used to accomplish the given task.

3.8 Evaluation of an Infix

Follow the algorithm to calculate the infix expression.

```
1. Read one line character
2. Actions at end of each input
   (a)      Opening brackets  push into stack and then goto step (1)
   (b)      Number                push into stack and then goto step (1)
   (c)      Operator              push into stack and then goto step (1)
   (d)      Closing brackets  pop from character stack
          if it is closing bracket, discard it, goto step (1)
          pop is used four times.
          The first popped element is assigned to op2
          The second popped element is assigned to op
          The third popped element is assigned to op1
        The fourth popped element is assigned the remaining.
        Opening bracket, which can be discarded.
        Evaluate op1 and op2.
        Convert the result into character and push into the stack. Go to
        step (d).
   (e)      New line character      pop from stack and print the answer stop.
```

Example-3.1 Evaluate the Infix expression (((2 * 5) – (1 * 2)) / (11 – 9) ?

Solution:

Five type of input characters: (*i*) Opening bracket (*ii*) Numbers (*iii*) Operators (*iv*) Closing bracket and (*v*) New line character

Data Structure Requirement: A character stack

Input string: (((2 * 5) – (1 * 2)) / (11 – 9)

Input symbol	Stack (from bottom to top)	Operation
(	(	
(	((	
(	(((	
2	(((2	
*	(((2*	
5		(((2*5
)	((10	2*5 = 10 and push
–	((10–	
(	((10–(	
1	((10–(1	
*	((10–(1*	
2	((10–1*2	
)	((10–2	1*2 = 2 and push
)	(8	10–2 = 8 and push
/	(8/	
(	(8/(	
11	(8/(11	
–	(8/(11–	
9	(8/(11–9	
)	(8/2	11–9 = 2 and push
)	4	8/2 = 4 and push
New line	Empty	pop & print

3.9 Evaluation of Prefix Expression

Follow the algorithm to evaluate prefix expression

1. Read the character input at a time and keep pushing it into the character stack until the new line character is reached.
2. Perform pop from the character stack. If the stack is empty, go to step (3)
   (a) Number: Push in to the integer stack and then, go to step-1.
   (b) Operator: Assign the operator to op
   Pop a number from the integer stack and assign it to op_1
   pop another number from integer stack and assign it to op_2
   Calculate op_1 and op_2 and push the output into the integer stock. Go to step-2.
3. Pop the result from the integer stack and display the result.

Example-3.2 Evaluate the Infix expression /–* 25 * 12 – 119 ?

Solution:

There are 3 types of input characters.

1. Numbers
2. Operators
3. New line character (\n)

Data Structure Requirements: A character stack and integer stack.

Input string: /– * 25 * 12 –119

Input Symbol	Character Stack (from bottom to top)	Integer Stack (from bottom to top)	Operation Performed
/			
-	/		
*	/-*		
2	/-*2		
5	/-*25		
*	/-*25*		
1	/-*25*1		
2	/-*25*12		
-	/-*25*12 -		
11	/-*25*12-11		
9	/-*25*12-119		
/n	/-*25*12-11	9	
	/-*25*12 -	911	
	/-*25*12	2	11 - 9 = 2
	/-*25*1	22	
	/-*25*	221	
	/-*25	22	1 * 2 = 2
	/-*2	225	
	/-*	2252	
	/-	2210	5 *2 = 10
	/	28	10 - 2 =8
	stack is empty	4	8 / 2 = 4
		stack is empty	print 4

3.10 Postfix Evaluation

The algorithm for Evaluating a postfix expression (using Stack).

- Initialize an empty stack
- While token remain in the input stream
  - (a) Read next token
  - (b) If token is a number, push it into the stack
  - (c) Else, if token is an operator, pop top two tokens off the stack, apply the operator, and push the answer back into the stack
- Pop the answer off the stack.

Algorithm postfix expression:

```
Initialize a stack, opndstk to be empty.
{scan the input string reading one element at a time into symb }
While (not end of input string)
{
      Symb := next input character;
      If symb is an operand Then
            push (opndstk,symb)
      Else
```

```
            [symbol is an operator]
        {
            Opnd1:= pop(opndstk);
            Opnd2:= pop(opndnstk);
        Value := result of applying symb to opnd1 & opnd2.
        Push(opndstk,value);
}
Result := pop (opndstk);
```

Example-3.3 Evaluate the postfix expression 6 2 3 + –3 8 2 / + * 2 $ 3 + ?

Solution:

Symbol	Operand 1 (A)	Operand 2 (B)	Value (A ⊗ B)	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
–	6	5	1	1
3				1, 3
8				1, 3, 8
2				1, 3, 8, 2
/	8	2	/	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2				7, 2
$	7	2	49	49
3				49, 3
+	49	3	52	52

The Final value in the STACK is 52. This is the answer for the given expression.

Run time stack for function calls:

- Push local data and return address onto stack.
- Return by popping off local data and then popping off address and returning to it.
- Return value can be pushed onto stack before returning, popped off by caller.

3.11 Infix to Postfix Conversion

Infix expressions are often translated into postfix form in which the operators appear after their operands.

Steps:

1. Initialize an empty stack.
2. Scan the Infix Expression from left to right.
3. If the scanned character is an operand, add it to the Postfix Expression.
4. If the scanned character is an operator and if the stack is empty, then push the character to stack.
5. If the scanned character is an operator and the stack is not empty, then
   (a) Compare the precedence of the character with the operator on the top of the stack.

(b) While operator at top of stack has higher precedence over the scanned character & stack is not empty. (i) POP the stack. (ii) Add the Popped character to Postfix String.

(c) Push the scanned character to stack.

6. Repeat the steps 3-5 till all the characters
7. While stack is not empty,
   (a) Add operator in top of stack
   (b) Pop the stack.
8. Return the Postfix string.

Algorithm infix to postfix conversion:

```
1.   Opstk = the empty stack;
2.   while (not end of input)
     {
         symb = next input character;
3.       if (symb is an operand)
             add symb to the Postfix String
4.       else
         {
5.           While(! empty (opstk) && prec (stacktop (opstk), symb))
             {
                 topsymb = pop (opstk)
                 add topsymb to the Postfix String;
             }   / * end of while */
             Push(opstk, symb);
         } /* end else */
6.   } /* end while */
7.   While(! empty (opstk))
     {
         topsymb = pop (opstk)
         add topsymb to the Postfix String
     }   /* end of while */
8.   Return the Postfix String.
```

Example-3.4 Conversion from Infix to Postfix of the given string ((A – (B+C))*D) $ (E+F) ?

Solution:

Symb	Postfix String	Opstk
(		(
(		((
A	A	((
–	A	((–
(	A	((–(
B	AB	((–(
+	AB	((–(+

)	$ABC+$	((−
)	$ABC+-$	(
*	$ABC+-$	(*
D	$ABC+-D$	(*
)	$ABC+-D*$	
$	$ABC+-D*$	$
(	$ABC+-D*$	$(
E	$ABC+-D*E$	$(
+	$ABC+-D*E$	$(+
F	$ABC+-D*EF$	$(+
)	$ABC+-D*EF+$	$
	$ABC+-D*EF+\$$	

NOTE: For postfix expression evaluation operand stack is used.

For prefix, postfix, infix conversation uses operator stack.

3.12 Implement Stack using Queues

We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue. A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q_1' and 'q_2'. Stack 's' can be implemented in two ways:

Method 1: This method makes sure that newly entered element is always at the front of 'q_2', so that pop operation just dequeues from 'q_2'. 'q_2' is used to put every new element at front of 'q_2'.

```
push(s, x)                    // x is the element to be pushed and s is stack
1.  Enqueue x to q2
2.  One by one dequeue everything from q1 and enqueue to q2.
3.  Swap the names of q1 and q2   // Swapping of names is done to avoid one
                                  more movement of all elements from q2 to q1.

pop(s)
1. Dequeue an item from q1 and return it.
```

Method 2: In push operation, the new element is always enqueued to q_1. In pop() operation, if q_2 is empty then all the elements except the last, are moved to q_2. Finally the last element is dequeued from q_1 and returned.

```
push(s, x)
1. Enqueue x to q1.
pop(s)
1. One by one dequeue everything except the last element from q1 and enqueue
   to q2.
2. Dequeue the last item of q1, the dequeued item is result, store it.
3. Swap the names of q1 and q2
4.  Return the item stored in step 2. // Swapping of names is done to avoid
                                  one more movement of all elements from q2 to q1.
```

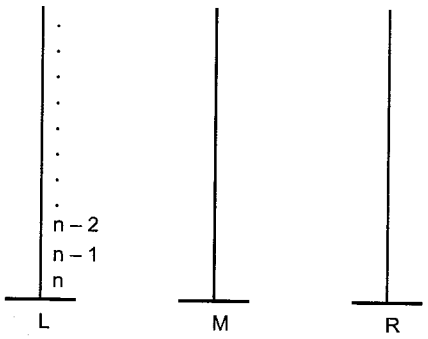
3.13 Tower of Hanoi

The tower of Hanoi also called the tower of Brahma or Lucas tower.

It consists of 3 rods, and the number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of sizes on one rod, the smallest at the top, thus make a conical shape.

The objective of the puzzle is to move the entire stack to another rod obeying the following rule.

1. Only one disk move at a time.
2. Each move consists of taking the upper disk from the one of the rods and sliding it onto another rod, on top of the other disk that may already be present on that rod.
3. No disk may be placed on top of a smeller disk and all disk will be there in 3-towers only.



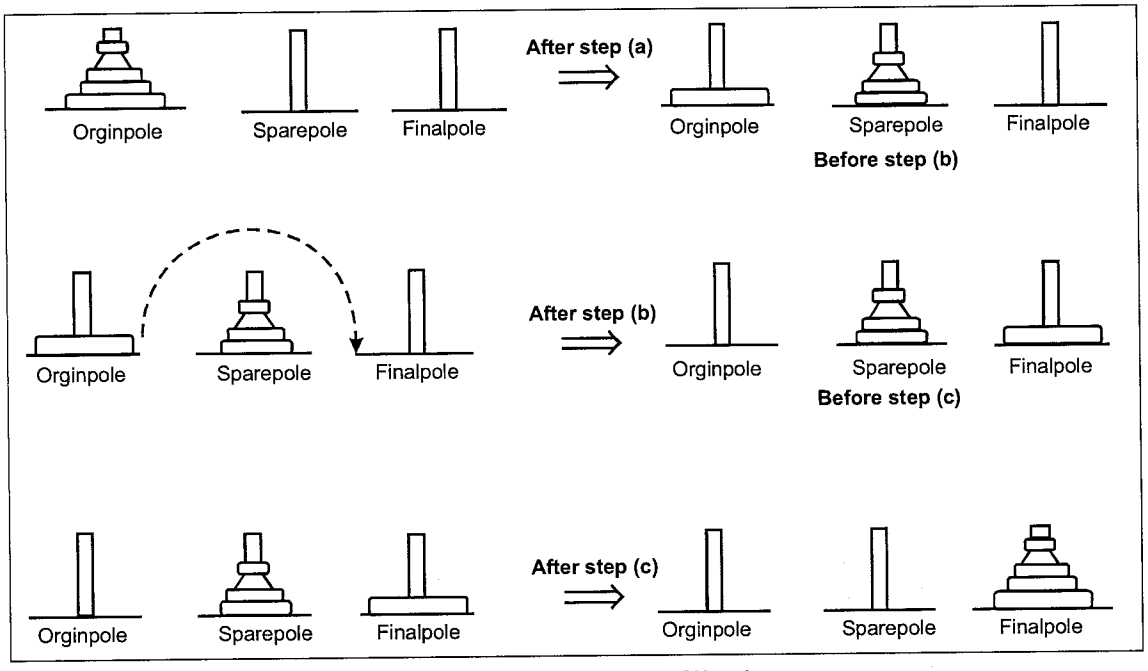

```
1. TOH (n - 1, R, M)
2. MOVE (L - R)
3. TOH (n - 1, M, L, R)
```

Algorithm for Tower of Hanoi:

```
TOH (n, L, M, R)
{
      if (n = = 0) return
      else
1.    TOH (n - 1, L, R, M)
2.    MOVE (L - R)
3.    TOH (n - 1, M, L, R)
}
```

NOTE

- After (n + 1) function call first move will be taken place.
- In TOH number of function call = $2^{n+1} - 1$
- In TOH number of move will be TOH (n) = $2^n - 1$



Working Procedure Tower of Hanoi

Summary

- A stack is an ordered list in which all insertion and deletions are made at one end. Top is a pointer pointing to the top most element of the stack.
- **Push** To add an element to the stack
- **Pop** To remove an element from the stack
- **Peek** To look at elements in the stack without removing them
- **LIFO** Refers to the last in, first out behavior of the stack
- **FILO** Equivalent to LIFO
  1. **Infix:** <operand> operator <operand>
  2. **Prefix (polish notation):** operator <operand> <operand>
  3. **Postfix:** <operand> <operand> operator
- **Applications of Stack:**
  1. Recursion
  2. Post fix notation (Evaluation of expression)
  3. Conversion of arithmetic expression from infix to polish (prefix/postfix)
  4. Tower of hanoi
  5. Fibonacci series
  6. Permutation
  7. Balancing of symbols
  8. Subroutines
- After (n + 1) function call first move will be taken place.
- In TOH number of function call = $2^{n+1} - 1$
- In TOH number of move will be TOH (n) = $2^n - 1$

Student's Assignments

Q.1 In a stack, the data item placed on the stack first is,
(a) The first data item to be removed
(b) The last data item to be removed
(c) Given as index zero
(d) Not given as index number

Q.2 The postfix form of A $ B ≠ C – D + E|F|(G + H) is
(a) AB $ C ≠ D – EF|GH +|+
(b) AB $ ≠ C – D + EF|GH|+
(c) AB $ ≠ C + D – EF|GH|+
(d) AB $ C – D ≠ EF|GH I ++

Q.3 To remove recursion from a program we have to use the following data structures
(a) Array (b) Stack
(c) Queue (d) List

Q.4 What is the maximum size of the operand stack while evaluating the postfix expression 6 2 3 + –3 8 2/+ *?
(a) 1 (b) 2
(c) 3 (d) 4

Common Data Questions (5 and 6):
Consider a stack processor which has 2 stack operations PUSH and POP.

PUSH	a
PUSH	b
PUSH	c
ADD	
PUSH	d
MUL	
PUSH	e
DIV	
PUSH	f
SUB	
ADD	

Q.5 What will be the post order expression equivalent to the above instructions?
(a) abc + de * f/– + (b) abc + de * / f + –
(c) abc + d * e/f – + (d) ab + c + d * e/f–

Q.6 What will be the final content of the stack if a = 2, b = 7, c = 3, d = 4, e = 5 and f = 4?
(a) 6 (b) – 2
(c) 0 (d) – 6

Q.7 Consider the following program:

```
Queue q=new Queue();
Stack S=new Stack();
S.push('A');
S.push('B');
S.push('C');
S.push('D');
S.push('E');
q.enque(S.pop());
q.enque(S.pop());
q.enque(S.pop());
S.push(q.deque());
S.push(q.deque());
q.enque('D');
q.enque('E');
```

What will be the top element of the stack, rear and front of the queue respectively.
(a) top = E, rear = E, front C
(b) top = D, rear = E, front E
(c) top = D, rear = E, front C
(d) top = D, rear = C, front E

Q.8 Which of the following is invalid stack operation?
(a) pop (pop (pop (push (b, push (a, S))))) is under flow.
(b) Get top (pop (push (c, push (b, push (a, S))))) = b
(c) pop (push (c, push (b, push (a, S))) = c
(d) Is empty (pop (push (b, pop (push (a, S))))) = False

Q.9 The following require of operations is performed on a stack; push (10), push (20), pop, push (10), push (20), pop, pop, pop, push (20), pop (10). What is output
(a) 20, 10, 20, 10, 20 (b) 20, 20, 10, 10, 20
(c) 10, 20, 20, 10, 20 (d) 20, 20, 10, 20, 10

Q.10 Assume that the operators × –,+ are left associative and ∧ is right associative. The order of precedence (from highest to lowest) is ∧, ×, +, – the postfix expression corresponding to the infix expression a + b × c – d ∧ e ∧ f is

(a) abc × + def ∧ × −

(b) abc × + de ∧ f ∧

(c) ab + c × d − e ∧ f ∧

(d) − + a × bc ∧ ∧ def

Q.11 Suppose STACK is allocated N = 6 memory cells and initially STACK is empty. Then find the output of the following module.

```
1. Set AAA=2 and BBB=5
2. Call PUSH(STACK,AAA)
   Call PUSH(STACK,4)
   Call PUSH(STACK,BBB+2)
   Call PUSH(STACK,9)
   Call PUSH(STACK,AAA+BBB)
3. Repeat while TOP≠0
   Call POP(STACK,ITEM)
   Write : ITEM
   [End of loop]
4. Return
```

(a) 2, 4, 7, 9, 7 (b) 7, 9, 7, 4, 2

(c) 2, 4, 9, 11, 9 (d) 9, 11, 9, 4, 2

Q.12 What is the output of the following program?

```
Queue q=new Queue();
Stack s=new Stack();
s.push(new Integer(8));
s.push(new Integer(9));
s.push(new Integer(10));
q.enqueue(s.pop());
q.enqueue(new Integer(8));
q.enqueue(new Integer(9));
s.push(q.dequeue());
System.out.print(s.pop ));
System.out.print(s.pop());
```

(a) 10, 8, 9 (b) 8, 9, 10

(c) 9, 8, 10 (d) 10, 9, 8

Answer Key:

1. (b)	**2.** (a)	**3.** (b)	**4.** (d)	**5.** (c)
6. (a)	**7.** (c)	**8.** (d)	**9.** (b)	**10.** (a)
11. (b)	**12.** (a)			

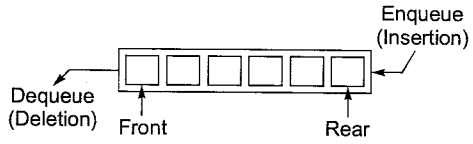
■■■■

CHAPTER 04

Queue

4.1 Introduction

"A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT". *Queue*s are sometimes referred to as First In First Out (FIFO) lists.



Example: (a) The people waiting in line at a bank cash counter form a queue. (b) In computer, the jobs waiting in line to use the processor for execution. This queue is called Job Queue.

4.2 Operations of Queue

There are two basic queue operations.

- **Enqueue:** Inserts an item / element at the rear end of the queue. An error occurs if the queue is full.
- **Dequeue:** Removes an item / element from the front end of the queue, and returns it to the user. An error occurs if the queue is empty.

Addition into a queue: (assume queue as linear array)

```
Void QInsert(int queue[],max,front,rear,item)
{  if ((rear + 1 == max)
   {
      printf ("queue is overflow");
      exit(-1);
   }
   else {
      if (front == rear == -1)
         front = rear = 0;
      else
         rear = rear + 1;
         queue[rear] = x;
   }
}
```

Deletion in a Queue: (assume queue as linear array)

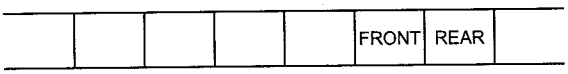
```
int QDelete(queue[],max,front,rear)
{  int y;
   if(front == rear == -1) or if(front == -1)
   {  printf("circular queue is underflow");
      exit(-1);
   }
   else {
      y = queue[front];
      if(front == rear)
         front = rear = -1;
      else
         front = front+1;
         return (y);
   }
}
```

4.3 Application of Queue

Queues remember things in first-in-first-out (FIFO) order. Good for fair (first come first served) ordering of actions.

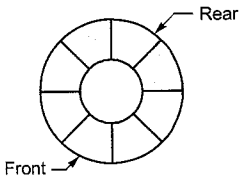
Application of stack (a) Scheduling processing of GUI events printing request. (b) Simulation orders the events models real life queues (e.g. supermarkets checkout, phone calls on hold).

Why do we need circular queue? In a queue, to show that the queue is full the condition is REAR == N, but their might be the case when queue has some empty slots while the condition REAR == N is full field so, we need FRONT == N –1 and REAR == N, which shows queue is actually full, hence we need a circular queue.



4.4 Circular Queue

Location of queue are viewed in a circular form. The first location is viewed after the last one. Overflow occurs when all the locations are filled.



Note: We assume max is a variable which tells us last position in the circular queue.

Algorithm circular queue insert:

```
Void CQInsert(int queue[],max,front,rear,item)
{  if ((rear + 1) % max == front)
   {  printf ("circular queue is overflow");
      exit(-1);
   }
   else {
      if (rear == -1)
         front = rear = 0;
      else
         rear = (rear + 1) % max;
         queue[rear] = x;
   }
}
```

Algorithm circular queue delete:

```
int CQDelete(queue[],max,front,rear)
{  int y;
   if(front == rear == -1) or if(front == -1)
   {
      printf("circular queue is underflow");
      exit(-1);
   }
   else {
      y = queue[front];
      if(front == rear)
         front = rear = -1;
      else
         front = (front+1) % max;
         return (y);
   }
}
```

4.5 Implement Queue using Stacks

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1: This method makes sure that newly entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

enQueue(q,x)

```
1. While stack1 is not empty, push everything from satck1 to stack2.
2. Push x to stack1 (assuming size of stacks is unlimited).
3. Push everything back to stack1.
```

deQueue(q)

```
1. If stack1 is empty then error.
2. Pop an item from stack1 and return it.
```

Method 2: In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

enQueue(q,x)

```
1. Push x to stack1.
```

deQueue(q)

```
1. If both stacks are empty then error.
2. If stack2 is empty
   While stack1 is not empty, push everything from satck1 to stack2.
3. Pop the element from stack2 and return it.
```

Queue can also be implemented using one user stack and one Function Call Stack: Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

enQueue(x)

```
1. Push x to stack1.
```

deQueue:

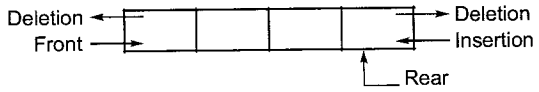
```
1. If stack1 is empty then error.
```

```
2. If stack1 has only one element then return it.
3. Recursively pop everything from the stack1, store the popped item.
In a variable "A", push the "A" back to stack1 and return "A".
```

The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in stack1 (step 2), we get the last element of stack1 in dequeue() and all other items are pushed back in step 3.

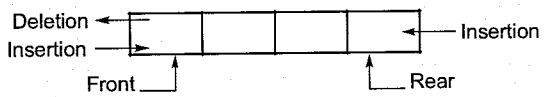
4.6 Types of Queue

1. **Input Restricted Queue:**



- Not FIFO.
- Insertion will be done only one side.
- Deletion will be done both side.

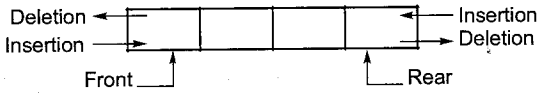
2. **Output restricted queue**



- Not FIFO.
- Insertion will be done both side.
- Deletion will be done only one side.

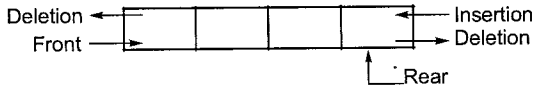
4.7 Double Ended Queue

Double ended queue is a linear list in which insertion and delection are possible at either end



Two types of Double Ended Queue

1. **Input Restricted dqueue:**



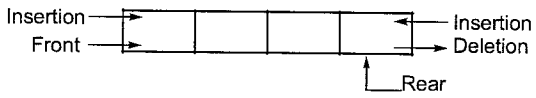
NOTE: M = size of queue.

F = M/2

R = (M/2) – 1

Delete Left	Insert Right	Delete Right
`if(F==(M/2))empty` `else` `{` `  Y=Q[F];` `  F++;` `}` `return (y);`	`if(R==M-1) full` `else` `{` `  R++;` `  Q[R]=x;` `}`	`if(R==(M/2)-1)empty` `else` `{` `  y=Q[R];` `  R;` `}` `return (y);#`

2. Output Restricted Dqueue:



R==M

Insert Left	Delete Left	Delete Right
`if(R==F+1) full` `else` `{` `F++;` `Q[F]=x;` `}`	`if(F==-1) empty` `else` `{` `y=Q[F];` `F --;` `}` `Return(y);`	`if(R==M) empty` `else` `{` `y=Q[R];` `R++;` `}` `return(y);`

NOTE: M = size of queue.

4.8 Priority Queue

- The **priority queue** is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations.
- There are two types of priority queue: an ascending priority queue and descending priority queue.
  (a) An **ascending priority queue** is a collection of items into which the items can be inserted arbitrarily and from which only the smallest item can be removed.
  (b) A **descending priority queue** is similar but allows deletion of the largest item.
- A queue may be similarly viewed as an ascending priority queue whose elements are ordered by time of insertion.

NOTE

- A stack may be viewed as a descending priority queue whose elements are ordered by time of insertion.
- The element that was inserted last has the greatest insertion-time value and is the only item that can be retrieved.

4.8.1 Array Implementation of a Priority Queue

Suppose that the n elements of a priority queue pq are maintained in positions 0 to n-of an array pq.items of size max pq, and suppose that pq. rear equals the first empty array position, n.

Then pq insert (pq, x) can be written as:

```
if(pq.rear>=maxpq){
   printf("priority queue overflow");
   exit(1);
} /*end if*/
pq.items[pq.rear]=x;
pq.rear++;
```

4.8.2 Problems with Array Implementation

As long as only insertions takes place, this implementation works well. Suppose, however, that we attempt the operation pq mindelete (pq) on an ascending priority queue.

This raises two issues:

- First, **to locate the smallest element,** every element of the array from pq. items [0] through pq. items [pq.rear–1] must be examined. Therefore a deletion requires accessing every element of the priority queue.
- Second, how can an element in the middle of the array be deleted? Stack and queue deletions involve removal of an item from one of the two ends and do not require any searching. Priority queue under this implementation requires both searching of the element to be deleted and removal of an element in the middle of an array.

Solutions:

1. A special empty indicator can be placed into a deleted position.
   Disadvantages:
   - The search process to locate the maximum or minimum element must examine all the deleted array positions in addition to the actual priority queue elements. If many items have been deleted but no compaction has yet taken place, the deletion operation accesses many more array elements than exist in the priority queue.
   - Second, once in a while insertion requires accessing every single position of the array, as it runs out of room and begins compaction.
2. The deletion operation labels a position as empty as in the previous solutions but insertion is modified to insert a new item in the first "empty" position. Insertion then involves accessing every array element up to the first one that has been deleted. This decreased efficiency of insertion and is a major draw back of this solution.
3. Each deletion can compact the array by shifting all elements past the deleted element by one position and then decrementing **pq.rear** by 1. Insertion remains unchanged. On the average, half of all priority queue elements are shifted for each deletion, so that deletion becomes quite inefficient and expensive. A slightly better alternative is to shift all proceeding elements forward or all succeeding elements backward, depending on which group is smaller.
4. Instead of maintaining priority queue as an unordered array, maintain it as an ordered, circular array as follows:

```
# max PQ 100
Struct pqueue{
    int items[MAX PQ];
    int front,rear;
}
Struct pqueue pq;
```

   - pq. front is the position of the smallest element, **pq.rear** is 1 greater than the position of the largest.
   - Deletion involves merely increasing **pq.front** (for the ascending queue) or decreasing pq. rear (for descending queue).
   - This method removes the work of searching and shifting from the deletion operation to the insertion operation.
   - However, since the array is ordered, the search for the position of the new element in an ordered array is only half as expensive on the average as finding the maximum or minimum of the unordered array, and a binary search might be used to reduce the cost even more.
   - Other technique that involve leaving gaps in the array between the elements of the priority queue to allow for subsequent insertions is also possible.

Summary

- A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT. *Queues* are sometimes referred to as First In First Out (FIFO) lists.
- **Enqueue:** Inserts an item / element at the rear end of the queue. An error occurs if the queue is full.
- **Dequeue:** Removes an item / element from the front end of the queue, and returns it to the user. An error occurs if the queue is empty.
- Location of queue are viewed in a circular form. The first location is viewed after the last one. Overflow occurs when all the locations are filled.
- Double ended queue is a linear list in which insertion and delection are possible at either end.
- The **priority queue** is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations.
- There are two types of priority queue: (a) An **ascending priority queue** is a collection of items into which the items can be inserted arbitrarily and from which only the smallest item can be removed. (b) A **descending priority queue** is similiar but allows deletion of the largest item.

Student's Assignments

Q.1 Queues serve a major role in
(a) Simulation of recursion
(b) Simulation of arbitrary linked list
(c) Simulation of limited resource allocation
(d) Expression evaluation

Q.2 In queue an element can be inserted and deleted by using
(a) Rear and front (b) Front and rear
(c) Top (d) None of these

Q.3 A sorted array of n elements contains 0 and 1 to find out majority of 0 and 1. How much time it will take?
(a) O(1) (b) O(logn)
(c) O(n) (d) $O(n^2)$

Q.4 Implementation of priority queue is
(a) Binary heap (b) Hashing
(c) Dqueue (d) None of these

Q.5 Which of the following data structure in which the intrinsic ordering of the elements does determine the result of its basic operation.
(a) Stack (b) Queue
(c) Priority queue (d) All of these

Q.6 An array A contains n characters from a,b,c,...z. How much time it will take to find first repeated character
(a) O(N log N) (b) O(N)
(c) O(log N) (d) None of these

Q.7 The initial configuration of circular queue is as follows

What is the status of Queue contents after the following sequence of steps
(*i*) enqueue 'x' (*ii*) dequeue
(*iii*) enqueue 'y' (*iv*) dequeue
(*v*) enqueue 'z' (*vi*) dequeue
(a) x, y, z, –, – (b) x, –, y, z, –
(c) x, –, –, y, z (d) –x, y, z, –

Q.8 Suppose a circular queue of capacity (n – 1) elements is implemented with an array of n-elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as array index variables, respectively.

Initially, REAR = FRONT = 0. The condition to detect queue full and queue empty are

(a) full : (REAR + 1) mod n = = FRONT
empty : REAR = FRONT

(b) full : (REAR + 1) mod n = = FRONT
empty : (FRONT + 1) mod n = = REAR

(c) full : REAR = = FRONT
empty : (REAR +1) mod n = = FRONT

(d) full : (FRONT + 1) mod n = = REAR
empty : REAR = = FRONT

Common Data Questions (9 and 10):
Suppose each data structure is stored in a circular array with N-memory cells.

Q.9 Find the number NUMB of elements in a queue in terms of FRONT and REAR with FRONT ≤ REAR and REAR < FRONT.

(a) NUMB = N + REAR – FRONT + 1
NUMB = REAR + FRONT – 1

(b) NUMB = N + REAR – FRONT + 1
NUMB = N + REAR – FRONT + 1

(c) NUMB = REAR – FRONT + 1
NUMB = N + REAR – FRONT + 1

(d) NUMB = N + REAR – FRONT – 1
NUMB = N + REAR + FRONT + 1

Q.10 With reference to the results obtained in (Q.9). Find NUMB with

(*i*) FRONT ≤ REAR, N = 12
FRONT = 3, REAR = 9

(*ii*) REAR < FRONT, N = 12
FRONT = 9, REAR = 4

(a) 19, 11 (b) 7, 8
(c) 13, 19 (d) 18, 26

Q.11 A priority queue Q is used to implement a stack that stores characters push (C) to implemented INSERT (Q,C,K) where K is an appropriate integer key chosen by the implementation. Pop is implemented as DELETEMIN (Q). For a sequence of operations, the key chosen are in

(a) Non increasing order
(b) Non decreasing order
(c) Strictly increasing order
(d) Strictly decreasing order

Q.12 Consider a stack S with element 1, 0, 2, 4, 5, 7 with top of the stack as 7 and two queues Q_1 and Q_2. Q_1 contains elements 3, 6, 7, 8, 9, 0 in that order from front to rear and Q_2 is initially empty. What are the contents of Q_1 and Q_2 if we execute the following code?

```
void foo()
{
   int x,y;
   while(!empty(S))
   {
      x=pop(S);
      y=dequeue (Q1);
      x=x+y;
      enqueue(Q1,x);
      enqueue(Q2,y);
   }
}
```

(a)

Q_1	10	11	11	10	9	1

Q_2	3	6	7	8	9	0

(b)

Q_1	10	11	11	10	9	1

Q_2	3	6	7	8	9	10

(c)

Q_1	10	11	10	11	9	1

Q_2	3	6	7	8	9	0

(d) None of these

Answer Key:

1. (c)	**2.** (a)	**3.** (a)	**4.** (a)	**5.** (c)
6. (b)	**7.** (d)	**8.** (a)	**9.** (c)	**10.** (c)
11. (d)	**12.** (a)			

CHAPTER 05

Linked Lists

5.1 Introduction

We have seen representation of linear data structures by using sequential allocation method of storage, as in, arrays. But this is unacceptable in cases like:

(a) **Unpredictable storage requirements:** The exact amount of data storage required by the program varies with the amount of data being processed. This may not be available at the time we write programs but are to be determined later.

Example: Linked allocations are very beneficial in case of polynomials. When we add two polynomials, and none of their degrees match, the resulting polynomial has the size equal to the sum of the two polynomials to be added. In such cases, we can generate nodes (allocate memory to the data member) whenever required, if we use linked representation (dynamic memory allocation).

(b) **Extensive data manipulation takes place:** Frequently many operations like insertion, deletion etc., are to be performed on the linked list.

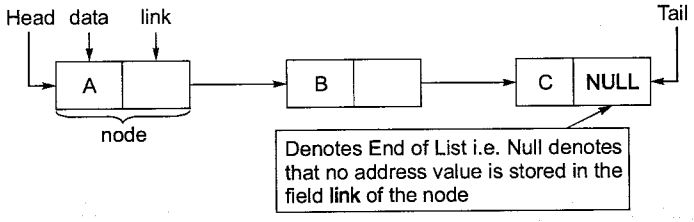
Comparison of Arrays, Vectors and Linked Lists

1. Arrays can be **useful** because:
   - The convenient "[]" notation allows immediate access to any element in the array.
   - Arrays can directly hold primitive types, as well as objects.
2. **Arrays** can be a **problem** because:
   - The size of the array must be known or at least estimated before the array can be used.
   - Increasing the size of an array can be very time consuming (create another array and copy contents).
   - Arrays can only use a contiguous block of memory. When a new element is inserted into an array, all elements above the new one must be shifted up.
3. **Vectors** are **useful** because:
   - Sizing is no longer a problem.
   - They are built-in to the java.util class.
   - They inherit many useful methods.

4. A Vector can be a **problem** because:
   - Arrays are still used (in the background).
   - Every re-sizing of the Vector causes a time lag, since a new array must be created in another block of memory and all the elements copied over.
   - Element insertion is just as time-consuming as with arrays, and possibly even worse if the Vector has to be resized.
   - It can only store Objects, not primitive types.
   - Do not have the handy "[]" notation, only methods.

5.2 Linked Lists

A singly linked list consists of Objects called "nodes" that contain data and a link to the next node in the list. A node is defined in one class, the linked list class, contains the "**head**" and "**tail**" pointers.



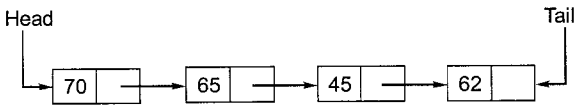
5.3 Uses of Linked lists

1. Link list can store primitive types or Objects.
2. A node can be anywhere in memory. The list does not have to occupy a contiguous memory space.
3. List size is only limited by available memory, and does not have to be declared first.
4. No empty nodes.
5. The adding, insertion or deletion of list elements or nodes can be accomplished with the minimal disruption of neighbouring nodes.

5.4 Singly Linked List or One Way Chain

This is a list, which may consist of an ordered set of elements that may vary in number. Each element in this linked list is called as *node*. A node in a singly linked list consists of two parts, a *information part* where the actual data is stored and a *link part*, which stores the address of the successor(next) node in the list. The order of the elements is maintained by this explicit link between them. The typical node is as shown:

Consider an example where the marks obtained by the students are stored in a linked list as shown in the Figure:



In Figure, the arrows represent the links. The **data** part of each node consists of the marks obtained by a student and the **next** part is a pointer to the next node. The NULL in the last node indicates that this node is the last node in the list and has no successors at present. In the above the example the data part has a single element marks but you can have as many elements as you require, like his name, class etc.

5.4.1 Operations on Singly Linked List

Structure declaration:

```
typedef struct node
{  int data;
   struct node *link;
}  NODE;
   # include<stdio.h>
   # include<alloc.h>   /*required for dynamic memory allocation*/
```

Function to add a node at the end of the linked list

```
append(NODE**q,int num)
{
   NODE*temp, *r;
   temp=*q;
   if(*q==NULL)/*list empty, create the first node*/
   {
      temp=malloc(sizeof(NODE));
      temp→data=num;
      temp→link=NULL;
      *q=temp;
   }
   else
   {
      temp=*q;
      /*go to the end of the list*/
      while(temp→link !=NULL)
      temp = temp→link;
      r=malloc(sizeof(NODE));
      r→data=num;
      /*add node at the r→link=NULL,i.e. end of the list*/
      temp→link=r;
      r→link=NULL
   }
}
```

The append( ) function has to deal with two situations:

(a) The node is being added to an empty list.

(b) The node is being added to the end of the linked list.

- In the First Case, the Condition:

  `if(*q==NULL)` gets satisfied. Hence space is allocated for the node using malloc( ). Data and the link part of this node are set up using the statements:

```
temp → data=num;
temp →link=NULL;
```

- Lastly p is made to point to this node, since the first node has been added to the linked list and p must always point to the first node. Note that *q is nothing but equal to p.

  In the other case, when the linked list is not empty, the condition:

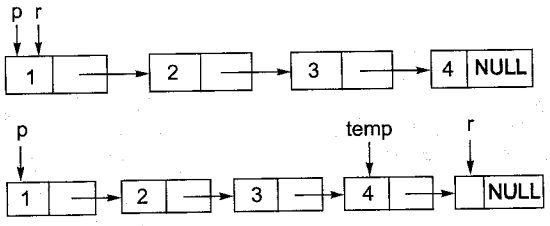
  `if( *q==NULL)` would fail, since *q (i.e. p is non-NULL). Now temp is made to point to the first node in the linked list through the statement,

  ```
  temp=*q;
  ```

  Then using temp we have traversed through the entire linked list using the statements:

  ```
  while(temp→link !=NULL)
  temp=temp→link;
  ```

  The position of the pointer before and after traversing the linked list is shown below:



Node being added at the end of Singly Linked List

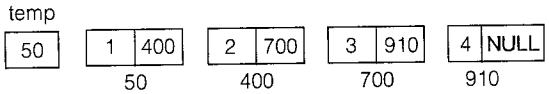
- Each time through the loop the statement `temp=temp→link` makes temp point to the next node in the list. When temp reaches the last node the condition `temp→link != NULL` would fail.
- Once outside the loop we allocate the space for the new node through the statement

  ```
  r=malloc(sizeof(NODE));
  ```

- Once the space has been allocated for the new node its data part is filled with num and the link part with NULL. Note that this node is now going to be the last node in the list.
- All that now remains is connecting the previous last node to this new last node. The previous node is being pointed to by temp and the new last node is by r, they are connected through the statement

  ```
  temp→link=r;
  ```

- There is often a confusion among the beginners as to how the statement `temp=temp→link` makes temp point to the next node in the linked list. Let us understand this with the help of an example. Suppose in a linked list containing 4 nodes temp is pointing to the first node. This is shown in the figure below:



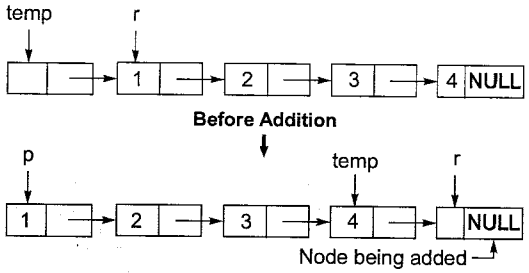
Actual representation of a Singly Linked List in Memory

- Instead of showing the links to the next node the above diagram shows the addresses of the next node in the link part of each node. When we execute the statement `temp=temp→link`, the right hand side yields 50. This address is now stored in temp. As a result, temp starts positioning nodes present at address 50. In effect the statement has shifted temp so that it has started positioning to the next node in the linked list.

5.4.2 Function to Add a Node at the beginning of the Linked List

```
add_beg(NODE **q,int num)
{
        temp= malloc(sizeof(NODE));
        temp→data=num;
        temp→link=*q;
        *q=temp;
}
```

Example: Suppose there are already 5 nodes in the list and we wish to add a new node at the beginning of this existing linked list. This situation is shown in the Figure.



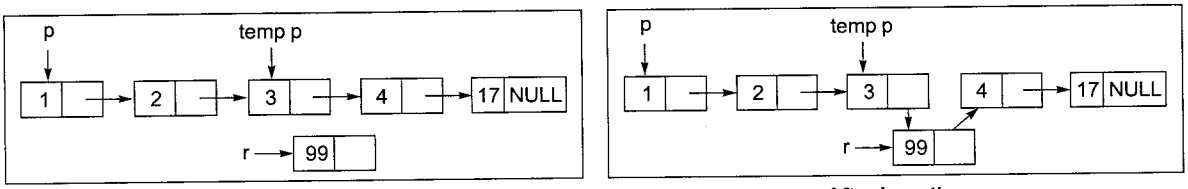
Addition of the Node at the beginning of a Singly Link List

- For adding a new node at the beginning, firstly space is allocated for this node and data is stored in it through the statement `temp→data=num;`
- Now we need to make the link part of this node point to the existing first node. This has been achieved through the statement `temp→link=*q;`
- Lastly this new node must be made the first node in the list. This has been attained through the statement `*q=temp;`

5.4.3 Function to Add a Node after the Specified Node

```
add_after(NODE *q,int loc,int num)
{
   NODE *temp,*t;
   int i;
      temp=q;
         for(i=0;i<loc;i++)
         {
            temp=temp→link;
            if(temp==NULL)
            {
                  printf("There are less than %d elements in the list",loc);
            return;
            }
         }
   r=malloc(sizeof(NODE));
   r→data=num;
   r→link=temp→link;
   temp→link=r;
}
```

- The add_after() function permits us to add a new node after a specified number of nodes in the linked list.
- To begin with, through a loop we skip the desired number of nodes after which a new node is to be added. Suppose we wish to add a new node containing data as 99 after the third node in the list. The position of pointers once the control reaches outside the **for** loop is shown as Figure:



Before Insertion **After Insertion**

Insertion of a node in the specified position

- The space is allocated for the node to be inserted and 99 is stored in the data part of it. All that remains to be done is readjustment of links such that 99 goes in between 3 and 4, this is achieved through the statements

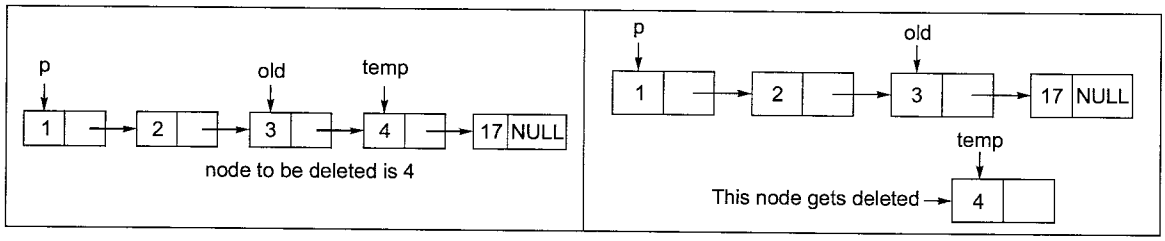
```
r→link=temp→link;
temp→link=r;
```

- The first statement makes link part of node containing 99 to point to the node contains 4, the second statement ensures that the link part of the node containing 3 points to the new node.

5.4.4 Function to Delete the Specified Node from the List

```
delete(NODE *q, int num)
{
   NODE *old, *temp;
   temp=*q;
   while(temp!=NULL){
      if(temp→data==num){
         if(temp==q){   /*if it is the first node*/
            *q=temp→link;
            free(temp); /*release the memory*/
         return;
         }else
         {  old→link==temp→link;
            free(temp);
            return;
         }
      }else
      {
         old=temp;
         temp=temp→link;
      }
   }
   printf("\n Element %d not found",num);
}
```

- In this function through the **while** loop , we have traversed through the entire linked list, checking at each node, whether it is the node to be deleted. If so, we have checked if the node is the first node in the linked list. If it is so, we have simply shifted **p** to the next node and then deleted the earlier node.
- If the node to be deleted is an intermediate node, then the position of various pointers and links before and after deletion are shown below.

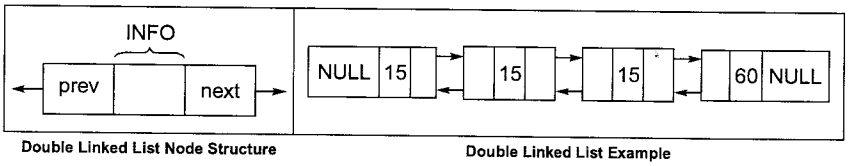


Before Deletion **After Deletion**

- Though the above linked list depicts a list of integers, a linked list can be used for storing any similar data. For example, we can have a linked list of floats, character array, structure etc.

5.5 Doubly Linked Lists or Two-way chain

- In a singly linked list we can traverse in only one direction (forward), i.e. each node stores the address of the next node in the linked list. It has no knowledge about where the previous node lies in the memory. If we are at the 12^{th} node(say) and if we want to reach 11^{th} node in the linked list, we have to traverse right from the first node. This is a cumbersome process. Some applications require us to traverse in both forward and backward directions. Here we can store in each node not only the address of the next node but also the address of the previous node in the linked list. This arrangement is often known as a **Doubly linked list.**



Double Linked List Node Structure **Double Linked List Example**

- The left pointer of the leftmost node and the right pointer of the rightmost node are NULL indicating the end in each direction.

Program Implements the Doubly Linked List

```
/*program to maintain a doubly linked list*/
#include<malloc.h>
typedef struct node
{  int data;
   struct node *prev, *next;
} NODE;
main()
{
   NODE *p;
   p=NULL;                    /*empty doubly linked list*/
   d_append(&p,11);
```

```
    d_append(&p,21);
    clrscr();
    display(p);
    printf("\n No of elements in the doubly linked list=%d",count(p));
    d_add_beg(&p,33);
    d_add_beg(&p,55);
    display(p);
    printf("\n No of elements in the doubly linked list=%d",count(p));
    d_add_after(p,1,4000);
    d_add_after(p,2,9000);
    display(p);
    printf("\n No of elements in the linked list=%d",count(p));
    d_delete(&p,51);
    d_delete(&p,21);
    display(p);
    printf("\n number of elements in the linked list=%d");
}
/*adds a new node at the beginning of the list*/
d_add_beg(NODE **s,int num)
{
    NODE *q;
    /*create a new node*/
    q=malloc(sizeof(NODE));
    /*assign data and pointers*/
    q→prev=NULL;
    q→data=num;
    q→next=*s;
    /*make the new node as head node*/
    (*s)→prev=q;
    *s=q;
}
/*adds a new node at the end of the doubly linked list*/
d_append(NODE **s,int num)
{
    NODE *r,*q=*s;
    if(*s==NULL)     /*list empty, create the first node*/
    {
        *s=malloc(sizeof(NODE));
        (*s)→data=num;
        (*s)→next=(*s)→prev=NULL;
    }
    else
```

```
    {
        while(q→next !=NULL) /*goto the end of list*/
        q=q→next;
        r=malloc(sizeof(NODE));
        r→data=num;  /*add node at the end of the list*/
        r→next=NULL;
        r→prev=q;
        q→next=r;
    }
}
/*adds a new node after the specified number of nodes*/
d_add_after(NODE *q,int loc,int num)
{
    NODE *temp;
    int i;
    /*skip to the desired position*/
    for(i=0;i<loc;i++)
    {
        q=q→next;
        if(q==NULL)
        {
            printf("There are less than %d elements in the list",loc);
            return;
        }
    }
    /*insert a new node*/
    q=q→prev;
    temp=malloc(sizeof(NODE));
    temp→data=num;
    temp→prev=q;
    temp→next=q→next;
    temp→next→prev=temp;
    q→next=temp;
    }
    /* to delete the specified node from the list.*/
    d_delete(NODE **s,int num)
    {
        NODE *q=*s;
        /*traverse the entire linked list*/
        while(q !=NULL)
        {
            if(q→data==num)
```

```
        {
            if(q==*s)  /*if it is the first node*/
            {
                *s=(*s)→next;
                (*s)→prev=NULL;
            }
            else
            {
                /*if the node is last node*/
                if(q→next==NULL)
                q→prev→next=NULL;
                else
                 /*node is intermediate*/
                {
                    q→prev→next=q→next;
                    q→next→prev=q→prev;
                }
                free(q);
            }
            return;    /*after deletion*/
        }
    q=q→next;        /*goto next node if not found*/
  }
  printf("\n Element %d not found",num);
}
```

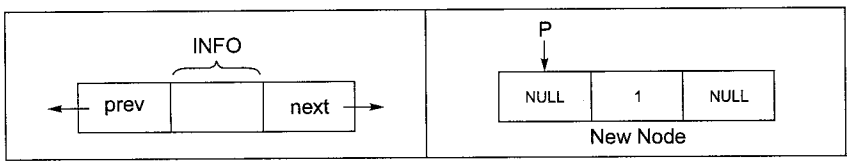
As you must have realized by now any operation on linked list involves adjustments of links. Since we have explained in detail about all the functions for singly linked list, it is not necessary to give step-by-step working anymore. We can understand the working of doubly linked lists with the help of diagrams.

5.5.1 Addition of New Node to an Empty Linked List

Case 1: Addition to an empty list

Related function : d_append( )

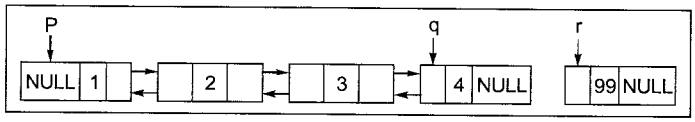
p = *s = NULL;



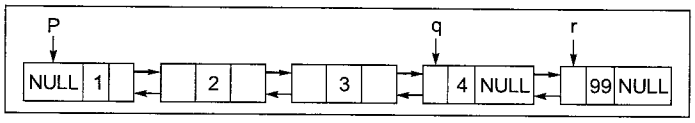
Before Addition **After Addition**

Case 2: Addition to an existing linked list

Related function : d_append( )



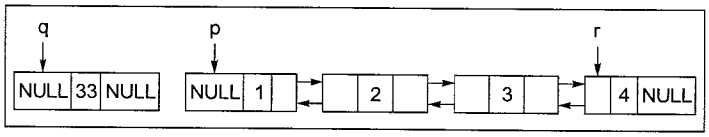
Before Appending



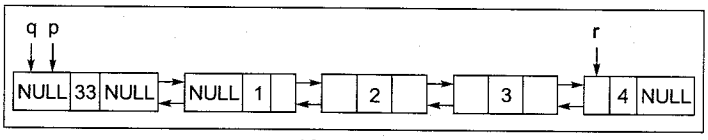
After Appending

Addition of new node at the beginning

Related Function: d_add_beg( )



Before Addition

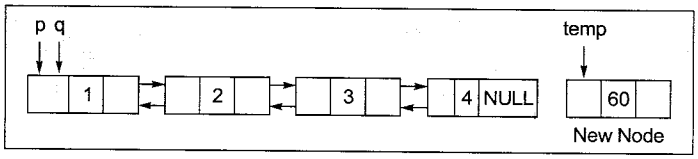


After Addition

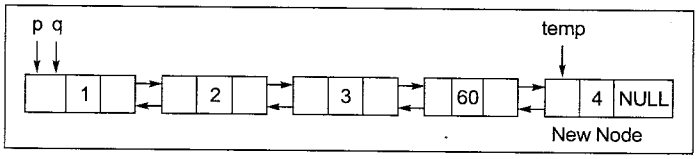
NOTE: Time complexity for adding a new node in beginning of linked list is O(1).

Insertion of a new node after a specified node

Related function : d_add_after( )



Before Insertion



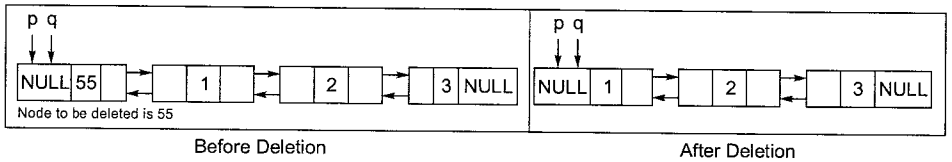
After Insertion

NOTE: Time complexity for adding a new node after a spefic node of linked list is O(n).

5.5.2 Deletion of a Node

Case 1: Deletion of first node

Related function : d_delete( )



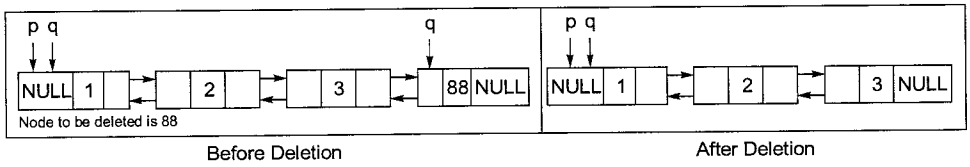
Before Deletion

After Deletion

NOTE: Time complexity for delete a node in beginning of linked list is O(1).

Case 2: Deletion of the last node

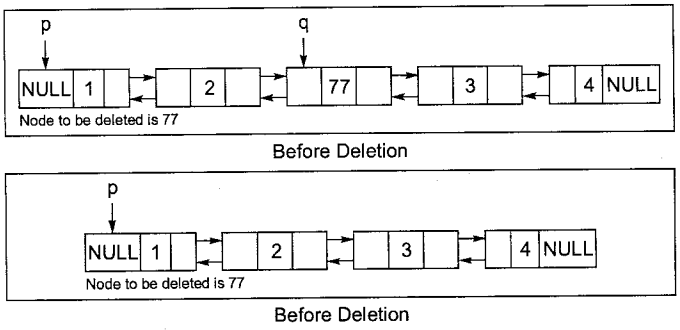
Related function : d_delete( )



NOTE: Time complexity for delete last node of linked list is O(n).

Case 3: Deletion of the intermediate node

Related function : d_delete( )

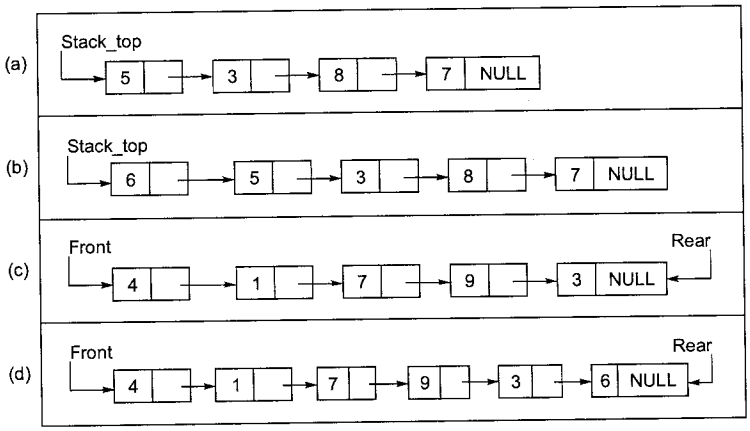


NOTE: Time complexity for delete intermediate node of linked list is O(n).

5.6 List

The operation of adding an element in front of a linked list is quite similar do that of pushing an element onto a stack. In both cases, a new item is added as the only immediately accessible item in a collection.

Push Operation	Pop Operation
P=getnode(); info=x; next(p)=S; S=P;	if(empty(s)){ printf ("stack under flow"); exit(1); } Else { P=S; S=next(P); x=info(P); }



NOTE: Here next (p) represents p → next and info (p) represents p → info.

Advantages of List

- All stacks being used by a program can share the same available list. When any stack needs a node, it can obtain from the single available list.
- When any stack no larger needs a node, it can obtain it from the single available list.

5.7 List Implementation of Queues

Under the list representation, a queue q consists of a list and two pointers, q-front and q-rear. The operations empty(a) and x=remove (q) are completely analogous to empty(s) and x = pop (s), with the pointer q. front replacing s.

Remove Operation	Insert Operation
`if(empty(q)){` `  printf("queue underflow");` `  exit(1);` `}` `  p=q.front;` `  x=info(p);` `  q.front=next(p);` `if (q.front==null)` `  q.rear=null;` `  freenode(p);`	`p=getnode( );` `info(p)=x;` `next(p)=null;` `if (q.rear==null)` `  q.front=p;` `else` `  next(q.rear)=p;` `  q.rear=p;`

5.8 List Implementation of Stacks

Operation of adding element to the front of a linked list is quite similar to that of posting an element onto a stack. In both cases, a new item is added as the only immediately accessible item in a collection.

Stack can be accessed only through its top element, and a list can be accessed only from the pointer to its first element. Similarly, the operation of removing the first element from a linked list is analogous to poping a stack.

Remove Operation	Insert Operation
`pop(struct node *top){` `  struct node *p;` `  int y;` `  if(top==null){` `  printf("stack is underflow");` `  exit(1);` `}` `  y=top→data;` `  p=top;` `  top=top→next;` `  free(p);` `  p=null;` `  return(y);` `}`	`push(struct node *top, int x)` `{` `  struct node *p;` `  p=(struct node*)malloc(size of (struct node));` `  if(p==null)` `  {` `  printf("stack is overflow");` `  exit(1);` `  }` `  p→data=x;` `  p→next=top;` `  top=p;` `}`

Disadvantages of Representing Stack or Queue by Linked List

A node in a linked list occupies more storage than a corresponding element in an array, since two pieces of information per element are necessary in a list node (info and next), whereas only one piece of information is needed in the array implementation.

- However, the space used for a list node is usually not twice the space used by an array element, since the elements in such a list usually consist of structured with many subfields.
- Another disadvantage is the additional time spent in managing the available list. Each addition and deletion of an element from a stack or a queue involves a corresponding deletion or addition to the available list.

Advantage of using Linked List Representation

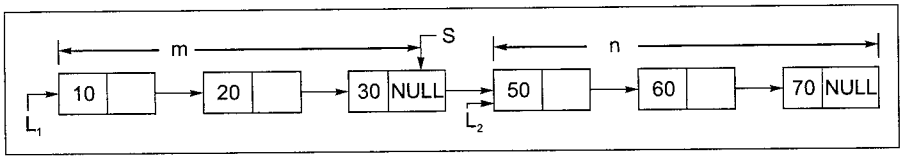
All the stacks and queues of a program have access to the same free list of nodes. Nodes not used by one stack may be used by another, as long as the total number of nodes in use at any one time is not greater than the total number of nodes available.

5.9 List Implementation of Priority Queues

- An ordered list can be used to represent a priority queue for an ascending priority queue, insertion (pa insert) is implemented by the place operation, which keeps the list ordered and deletion of the minimum element (pqmindelete) is implemented by the pop operation, which removes the first element from the list.
- A descending priority queue can be implemented by keeping the list in descending, rather than ascending, order or by using remove to implement pa max delete.
- A priority queue implemented as an ordered linked list requires examining an average of approximately n/2 nodes for insertion, but only one node for deletion.
- An unordered list may also be used as a priority queue. Such a list requires examining only one node for insertion (by implementing p q insert using push or insert) but always requires examining n-elements for deletion (traverse the entire list to find the minimum or maximum and then delete the node).
- Thus an ordered list is some what more efficient than an unordered list in implementing a priority queue.
- The advantage of a list over an array for implementing a priority queue is that no shifting of elements or gaps are necessary in a list. An item can be inserted into a list without moving any other items, whereas this is impossible for an array unless extra space is left empty.

5.10 Other Operation on Linked List

Concatenation of 2 Linked List

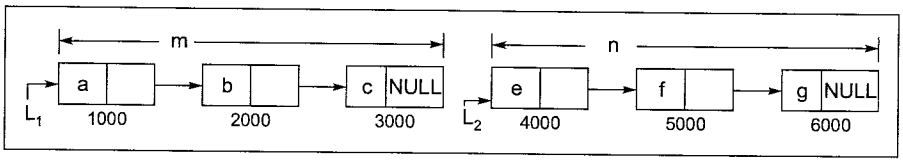



```
S=L1;
while(S→next!=null)
S=S→next;
S→next=L2;
return(L1);
```

O(m)

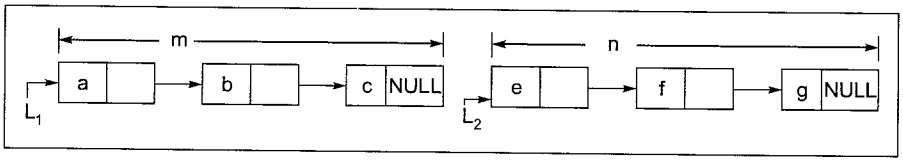
NOTE: Concatenation of 2 linked list take T.C. = O(m+n).

Intersection of 2 Linked List



NOTE: For every element of first linked list compare total second linked list. So, T.C. = $O(n^2)$.

Union of 2 Linked List



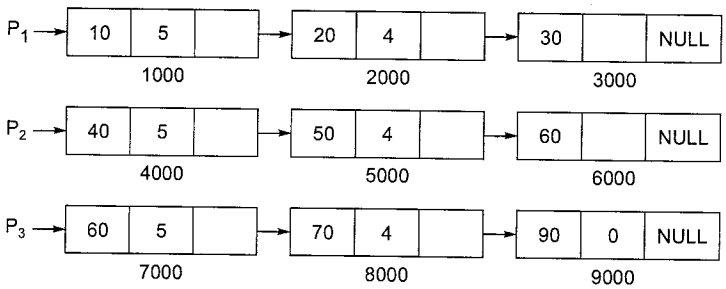
Algorithm:

1. Add first linked list to the resultant third linked list.
2. Using first linked list remove common element from the second linked list.
3. Add modified second linked list to third linked list.

NOTE: 1. Union of 2 linked list T.C. = $O(n^2)$. 2. Binary search on linked list will be done in O(n) time because finding middle itself take O(n) time. So we not preferred binary search on linked list.

5.11 Polynomial Addition Using Linked List

```
Struct poly{
    int coe;
    int pow;
    struct poly, * next;
};
```




```
Poly addition (struct poly*P1, struct poly*P2, struct poly*P3){
S3=P3;
    while(P!=Null&&P2!=Null){
        if(P1→pow==P2→pow){
            P3→pow=P1→pow;
            P3→coe=P1→coe + P2→coe;
            P1=P1→next; P2=P2→next;
            P3=P3→next;
```

```
        }
        else{
            if(P1→pow > P2→pow){
                P3→pow=P1→pow;
                P3→coe=P1→coe;
                P1=P1→next;
                P3=P3→next;
            }
            else{
                P3→pow=P2→pow;
                P3→coe=P2→coe;
                P3=P3→next;
                P2=P2→next;
            }
        }
        if(P1==Null&&P2==Null){
            return(P3);
        }
        else{
            if(P1==Null){
                add(P2, P3)
            }
            else{
                add(P1, P3)
            }
        }
}
```

Remember: Polynomial addition take 0(1) time complexity.

5.12 Polynomial Multiplication Using Linked List

$S_1 \rightarrow P_1 : 2x^5 + 10x^2 + 100$

$S_2 \rightarrow P_2 : 5x^5 + 3x^3 + 4x^2 + 10$

$S_3 \rightarrow P_3 : 10x^{10} + 6x^8 + 58x^7 + 550x^5 + 40x^4 + 30x^3 + 500x^2 + 1000$

```
Polymultiplication (struct poly *P1, *P2, *P3)
S3=P3; S2=P2;
while (P1!=Null){
    while(P2!=Null){
        P3→pow=P1→pow + P2→pow;
        P3→coe=P1→coe * P2→coe;
        P2=P2→next;
        P3=P3→next;
    }P1=P1→next;
    P2=S2;
}
```

Remember: Polynomial multiplication T.C. = $O(n^2)$.

Summary

- A singly linked list consists of Objects called "nodes" that contain data and a link to the next node in the list. A node is defined in one class, the linked list class, contains the "**head**" and "**tail**" pointers.
- We can store in each node not only the address of the next node but also the address of the previous node in the linked list. This arrangement is often known as a **Doubly linked list.**
- The left pointer of the leftmost node and the right pointer of the rightmost node are NULL indicating the end in each direction.
- Number of pointers modified to insert a new node at end = 2
  `node → next = newnode;`
  `newnode → next = null;`
- Number of pointers modified to insert a new node at beginning = 2
  `newnode → next = root;`
  `root = newnode;`
- Number of pointers modified to insert a new node in mid = 2
  `newnode → next = node → next;`
  `node → next = newnode;`
- Number of pointers modified to insert a new node at beginning = 1
  `head → next = head → next → next;`
- Number of pointers are modified to insert a node at beginning = 4
  `newnode → next = root;`
  `newnode → prev = null;`
  `root → prev = newnode;`
  `root = newnode;`
- Number of pointers are modified to insert a node at the end = 3
  `newnode → next = null;`
  `newnode → prev = node;`
  `node → next = newnode;`
- Number of pointers are modified to insert a node in between = 4
  `newnode → next = node → next;`
  `newnode → prev = node;`
  `(node → next) → prev = newnode;`
  `node → next = newnode;`
- Number of pointers are modified to delete a node in middle = 2
  `(node → prev) → next = node → next;`
  `(node → next) → prev = node → prev; free (node);`
- Binary search on linked list take O(n) time.
- Union operation of two linked list take $O(n^2)$.
- Concatination of two linked list will take O(m+n).

Student's Assignments

Q.1 In a linked list

(*i*) Each link contains a pointer to the next link
(*ii*) An array of pointers point to the links
(*iii*) Each link containing data and pointer to next
(*iv*) The links are stored in an array

Which of the following is correct.

(a) (*i*, *iii*) (b) only (*i*)
(c) only (*iii*) (d) (*ii*, *iv*)

Q.2 Which of the following operation is performed more efficiently by double linked list than by linear linked list

(a) deleting nodes whose location is given
(b) searching an unsorted list for a given item
(c) inserting a node after the node with a given location
(d) traveling the list to process each node

Q.3 Linked lists are not suitable for

(a) Insertion sort
(b) Binary search
(c) Polynomial addition
(d) Polynomial multiplication

Q.4 A list can be initialized to the empty list by which operation

(a) list = 1; (b) list = 0;
(c) list = null; (d) None of these

Q.5 In the given n elements linear linked list to find k^{th} element from the end, how much time it will take?

(a) O(n) (b) $O(n^2)$
(c) O(nlogn) (d) none

Q.6 If we want to find last node of a linked list then the correct coding is :

(a) if (temp → link ! = null)
temp = temp → link;
(b) if (temp → data = num
temp = temp → link;
(c) while (temp → link ! = data)
temp = temp → link;
(d) while (temp → link! = null)
temp = temp → link;

Q.7 Consider the following program with a linked list called head :

```
int func(list *head){
  list *a, *b;
  if(head==null||head→next==null)
  return 0;
  a=head;
  b=head→next;
  while (a!=b)
  {
    if (a==null||b==null||
    b→next==null)
    return 0;
    a=a→next;
    b=b→next;
    if(b→next==NULL)
    return 0;
    else
    b=b→next;
  }
  return 1;
}
```

What does the function func() do?

(a) Finds if the length of the loop is odd
(b) Finds whether any loop is present in linked list
(c) Finds whether the length of the linked list is even
(d) None of these

Q.8 The most appropriate matching for the following pairs

List-I

A: m = malloc(5); m = null;
B: free (n); n → value = 5;
C: char * p; *p = 'a';

List-II

1: Using dangling pointer
2: Using uninitialized pointers
3: Lost memory

	A	B	C
(a)	1	3	2
(b)	2	1	3
(c)	3	2	1
(d)	3	1	2

Q.9 What does do() on a single linked list :

```
void do (struct node*p,struct node*c)
{
  if(c)
  {
    Do(c,c→link);
  }
  else
First=p;
}
void main()
  {
    Do(null,first);
  }
```

(a) Print the singly linked list in reverse
(b) Reverse the singly linked list
(c) Duplicates each node of the list
(d) Delete duplicate nodes

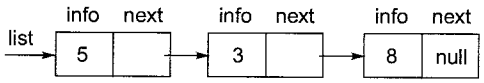
Q.10 Consider the following given code:

```
Void deletenode from LL(struct
LLnode**head)
{
  struct LLnode*temp=*head;
  struct LLnode*current=*head;
  if(*head==NULL)
  {
    printf("list empty");
    return;
  }
  while(current→next!=head)
  {
    current=current→next;
  }
  current→next=*head→next;
  *head=*head→next;
  free(temp);
  return;
}
```

Which of the following is true for the above given code?

(a) The given code is used to delete the last node in a circular linked list.
(b) The given code is used to delete all the nodes in a circular linked list
(c) The given code is used to delete the first node in a circular linked list
(d) The given code is used to delete the middle node in a circular linked list.

Q.11 Consider the following sequence of operation on given link list:




```
P=getnode();
info(P)=6;
next(P)=list;
list=P;
  :
  :
P=list;
list=next(P);
X=info(P);
freenode(P);
```

The above sequence of operation to linked list results into

(a) Updation of a node
(b) Deletion of a node
(c) Linked list remains the same
(d) None of the above

Q.12

```
Void main()
{
  int *mptr, *cptr;
  mptr=(int*)malloc(size of(int));
  printf("%d",*mptr);
  int*cptr=(int*)calloc(size of (int));
  printf("%d",*cptr);
}
```

What is the output of the above programe?

(a) grabagl, 0 (b) 0, grabagl
(c) grabagl, grah (d) 0,0

Answer Key:

1. (a)	**2.** (a)	**3.** (b)	**4.** (c)	**5.** (a)
6. (c)	**7.** (b)	**8.** (d)	**9.** (b)	**10.** (c)
11. (c)	**12.** (a)			

CHAPTER 06

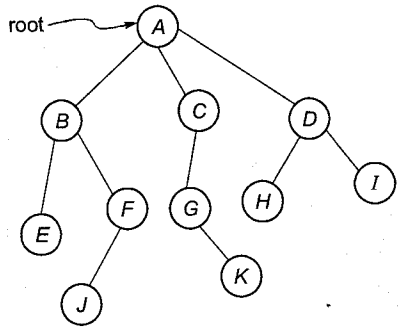
Trees

6.1 Introduction

A Tree is a data structure similar to linked lists but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes. Trees is an example of non-linear data structures. A tree structure is a way of representing the hierarchical nature of a structure in a graphical form.

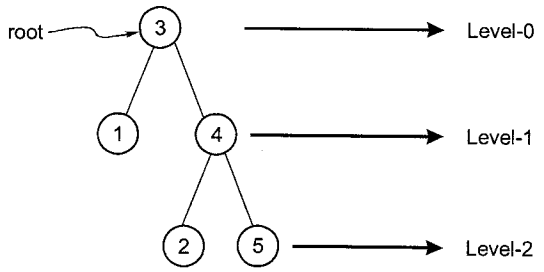
In trees ADT (Abstract Data Type), order of the elements is not important. If we need ordering information linear data structures like linked lists, stacks, queues, etc. can be used.

6.2 Glossary

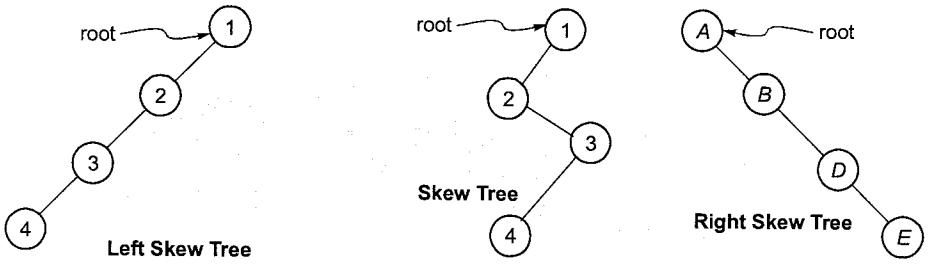


- The root of a tree is the node with no parents. There can be at most one root node in a tree (node A in the above example).
- An edge refers to the link from parent to child (all links in the figure).
- A node with no children is called leaf node (*E*, *J*, *K*, *H* and *I*).
- Children of same parent are called siblings (*B*,*C*,*D* are siblings of *A* and *E*, *F* are the siblings of *B*).
- A node *p* is an ancestor of a node *q* if there exists a path from root to *q* and *p* appears on the path. The node *q* is called a descendant of *p*. For example, *A*, *C* and *G* are the ancestors for *K*.
- The depth of a node is the length of the path from the root to the node (depth of *G* is 2, *A*-*C*-*G*).
- The height of a node is the length of the path from the root to the node to the deepest node. The height of a tree is the length of a path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero. In the previous example, height of *B* is 2(*B*-*F*-*J*).

- Height of the tree is the maximum height among all the nodes in the tree and depth of the tree is the maximum depth among all the nodes in the tree. For a given tree depth and height returns the same value. But for individual nodes we may get different results.
- Size of a node is the number of descendants it has including itself (size of the subtree *C* is 3).
- Set of all nodes at a given depth is called level of the tree (*B*, *C* and *D* are same level). The root node is at level zero.

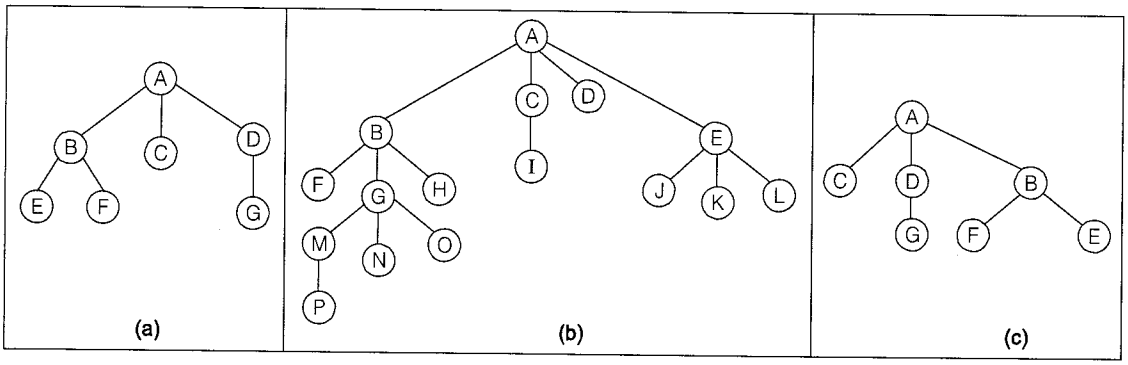


- If every node in a tree has only one child (except leaf nodes) then we call such trees as skew trees. If every node has only left child then we call them as left skew trees. Similarly, if every node has only right child then we call them as right skew trees.



6.3 Applications of Tree

- A tree is a finite non-empty set of elements in which the element is called the root and the remaining elements are partitioned into M>=0, disjoint subsets, each of which is itself a tree.
- Each element is called the node of the tree.
- An ordered tree is defined as a tree in which the subtrees of each node form an order set.
- A forest is an ordered set of ordered trees.
- Thus every binary tree is a tree but every tree is not a binary tree since it can have more than two sons.



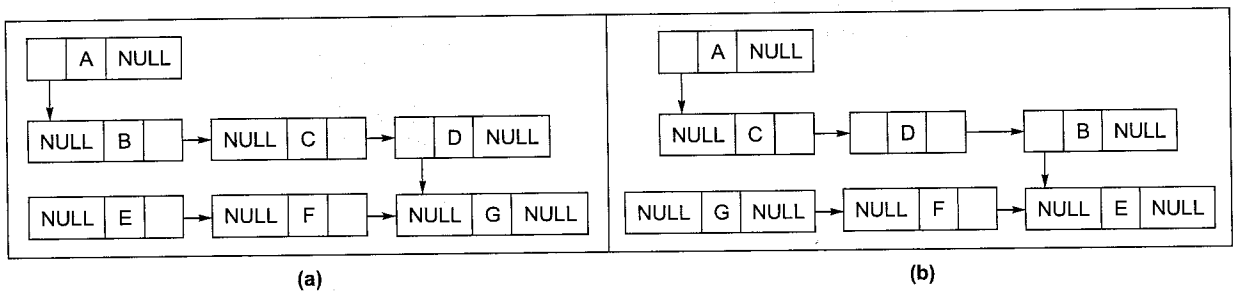
Static Representation of Tree

We can link all the sons of a node together in a link list. Thus:

```
#define MAXNODES 500
struct tree node{
     int root;
     int father;
     int son;
     int next;
};
struct treenode node[MAXNODES]
```

Dynamically Representation of Tree

```
struct tree node
{
     int root;
     struct tree node*father;
     struct tree node*son;
     struct tree node*next;
};
Typedef struct treenode*NODEPTR;
```



(a) (b)

6.4 Tree Traversals for Forests

The tree traversals of a forest may be defined as the tree traversals of its corresponding binary tree. Routines for preorder, postorder and Inorder.

(a) Preorder:

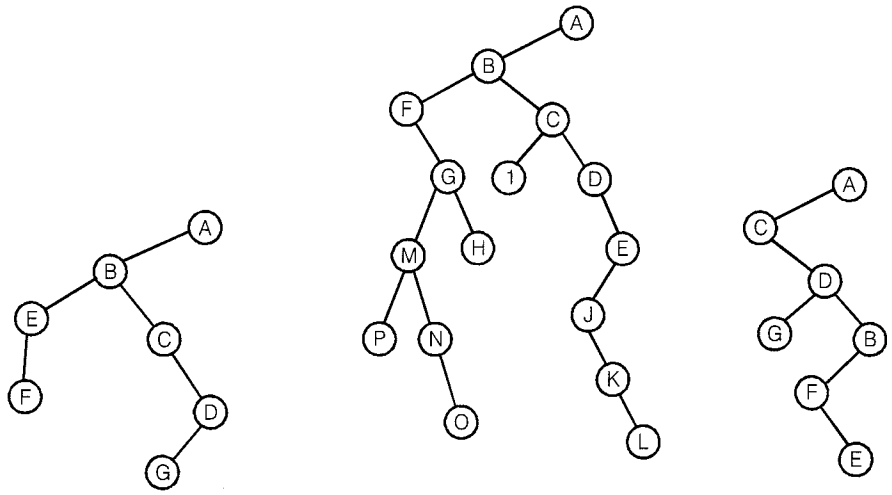
1. Visit the root of the first tree in the forest
2. Traverse in preorder the forest formed by the subtrees of the first tree, if any
3. Traverse in preorder the forest formed by the remaining trees in the forest if any.

(b) Inorder:

1. Traverse in inorder the forest formed by the subtrees of the first tree in the forest if any
2. Visit the root of the first tree
3. Traverse in inorder the forest formed by the remaining trees in the forest, if any

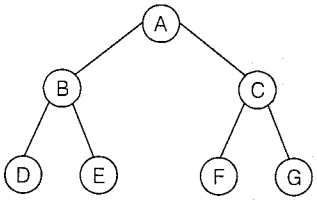
(c) Postorder:

- Traverse in postorder the forest formed by the subtrees of the first tree in the forest, if any
- Traverse in postorder the forest formed by the subtrees of the first forest, if any
- Visit the root of the first tree in the forest



6.5 Binary Trees

- A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets.
- The first subset contains a single element called the **root** of the tree.
- The other two subsets are themselves binary trees, called the left and right substrees of the original tree.
- Each element of a binary tree is called a node of the tree.



Binary tree

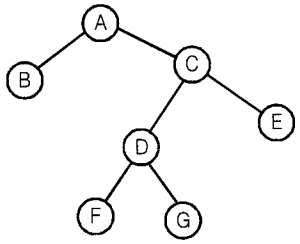
NOTE

- **Level:** The level of a node in a binary tree is the number of edges traversed from the root to that node. i.e., root has level 0 and level of any other node is one more than the level of its father.
- **Depth:** The depth of a binary tree is the maximum level of any leaf in the tree.
- The longest simple path from the root to any leaf node. i.e. leaves has depth or height = 1 and root has the maximum depth or height.

6.6 Types of Binary Trees

Strictly Binary Tree

- If every non leaf node in a binary tree has non empty left and right substrees, the tree is termed as strictly binary tree

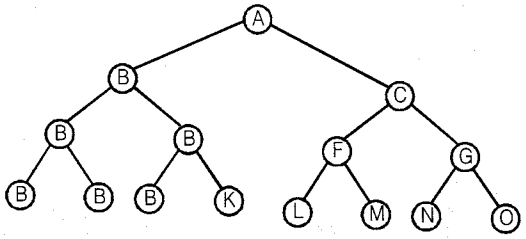


Strictly binary tree

- Every non-leaf node has degree 2.
- A strictly binary tree with n leaves has (2n – 1) nodes.
- Thus a strictly binary tree has odd number of nodes.

Complete Binary Tree

- A complete binary tree of depth d is the strictly binary tree - all of whose leaves are at level d.



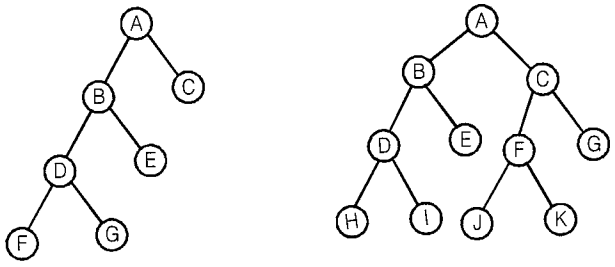
- A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^l nodes at each level *l* between 0 and d.
- Thus the total number of nodes in a complete binary tree are $2^{d+1} - 1$ where leaf nodes are 2^d and non-leaf are $2^d - 1$.
- Because a complete binary tree is also a strictly binary tree, thus if it has n leaves then it has 2n–1 nodes. Also follows from this is the previous assertion that if $n = 2^d$ then total nodes are

$$2*2^d - 1 = 2^{d+1} - 1$$

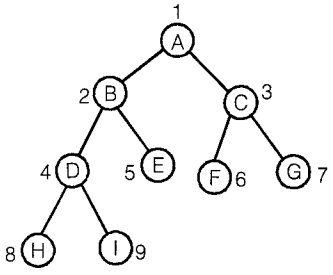
Almost Complete Binary Tree

- **A binary tree of depth d is an almost complete binary tree if:** At any node in the tree with a right descendent at level d, node must have a left son and every left descendent of node is either a leaf at level d or has two sons i.e. the tree must be left filled.

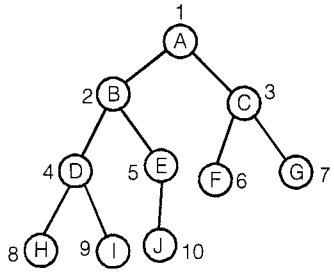
NOTE: ACBT Property says that if there are n nodes in the tree than leaf nodes have numbers $\left\lceil \frac{n}{2}+1 \right\rceil, \left\lceil \frac{n}{2}+2 \right\rceil, \ldots n$.



(a) (b)

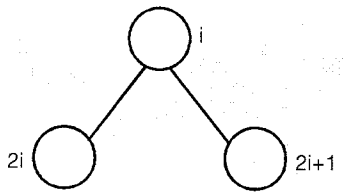


(c)

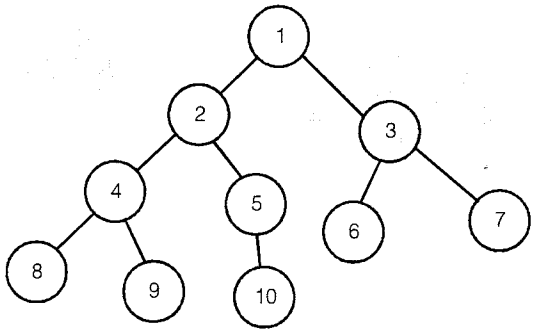


(d)

- Figure (a) violates property (a) and figure (b) violates properties (b) but figures (c) and (d) satisfy both the properties.
- In ACBT the numbering of nodes is done as



∴ *Examples:*



NOTE: We can see that the tree has n = 10 nodes.

∴ The leaves are from $\left\lceil \frac{10}{2}+1 \right\rceil, \left\lceil \frac{10}{2}+2 \right\rceil \ldots 10$.

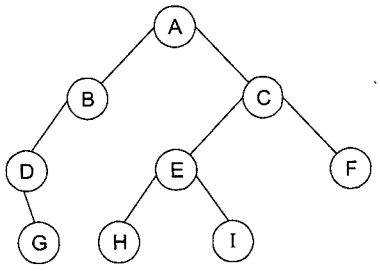
- An ACBT with n leaves has 2n nodes if it is not an strict binary tree like Figure (d).
- An ACBT which is also an strict binary tree like Figure (c) has 2n – 1 nodes for n leaves.
- Also an ACBT is an strict binary tree only if the number of nodes are odd.
- An ACBT of depth d is intermediate between the complete binary tree of depth d – 1, that contains $2^d - 1$ nodes, and the complete binary tree of depth d, which contains $2^{d+1} - 1$ nodes.

6.7 Applications of Binary Tree

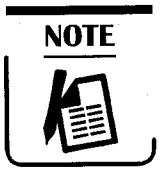
A binary tree is a useful data structure when two-way decisions must be made at each point of process.

Tree Traversals

1. **Preorder** (also known as depth-first order):
   (a) Visit the root
   (b) Traverse the left subtree in preorder
   (c) Traverse the right subtree in preorder
2. **Inorder** (Also known as symmetric order):
   (a) Traverse the left subtree in inorder
   (b) Visit the root
   (d) Traverse the right subtree in inorder
3. **Postorder:**
   (a) Traverse the left subtree in postorder
   (b) Traverse the right subtree in postorder
   (c) Visit the root



Preorder: ABDGCEHIF
Inorder: DGBAHEICF
Postorder: GDBHIEFCA



- A binary tree having the property that all the elements in the right subtree are greater than the root and in the left subtree are smaller than the root is known as a Binary Search Tree.
- The inorder traversal of a Binary search tree gives us the sorted list and the nodes are printed in the ascending order.
- Number of binary trees that can be built with n nodes are $\binom{2n}{n} * \frac{1}{n+1}$.

6.7.1 Binary Tree Traversals in C

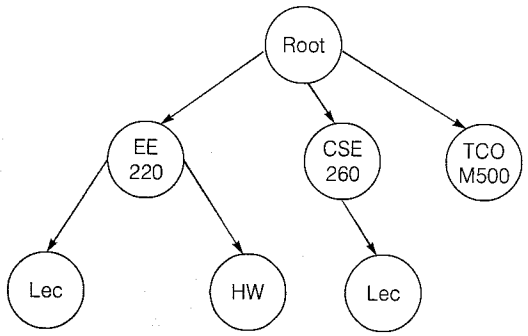
```
• void pretrav(NODEPTR tree)
  {
     if(tree!=NULL)
     {
        printf("%d\n",tree→root);  /*visit the root*/
        pretrav(tree→left);        /*traverse left subtree*/
        pretrav(tree→right);       /*traverse right subtree*/
     }   /*end if*/
  }  /*end pretrav*/
• void intrav(NODEPTR tree)
  {
     if(tree!=NULL){
        intrav(tree→left);          /*traverse left subtree */
        printf("%d\n",tree→root);   /*visit the root*/
        intrav(tree→right);         /*traverse right subtree*/
     }   /*end if*/
  }
```

- ```
  void posttrav(NODEPTR tree)
  {
     if(tree!=NULL)
     {
        posttrav(tree→left);      /*traverse left subtree*/
        posttrav(tree→right);     /*traverse right subtree*/
        printf("%d/n",tree→info) /*visit the root*/
     } /*end if*/
  } /*end posttrav*/
  ```

Preorder Example

Need to find the depth of each node in the tree representing the unix file system.

```
Listdepth(node,depth)
{
      print(node→name,depth);
      for each child C of node
      Listdepth(C,depth+1)
}
```



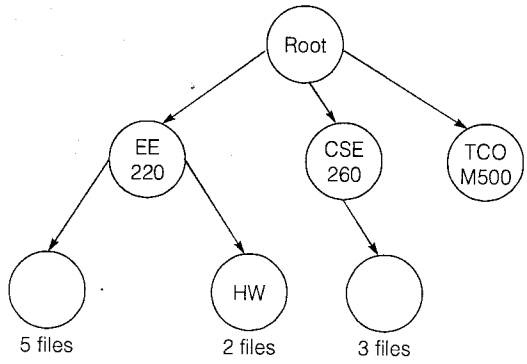
Print sequence:

Root, 0	EE220 1	Lec 2	HW 2
CSE260 1	Lec 2	TOC500 1	

Postorder Example

Need to find the size of each directory in the unix file system

```
Dirsize(node)
{
      if(node=leaf)return(file num);
      size=0;
      for each child C of node
      size=size+Dirsize(C);
      printf(node→name,size);
}
```

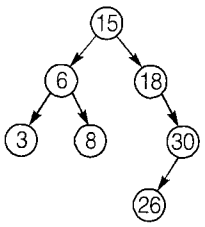


Print sequence:

Lec: 5,	HW: 2	EE220: 7	Lec: 3
CSE 260: 3	TCOM 500: 0	Root: 10	

In-Order Traversal

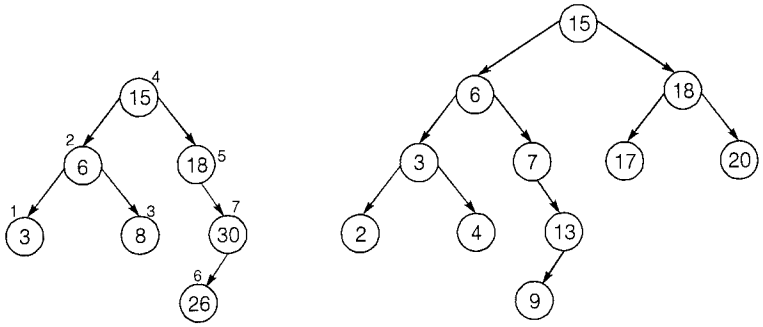
- Is this tree sorted?
  If you visit in the right order it is sorted.
- Algorithm Inorder(x)
  Input: x is the root of a subtree.




```
1. if x≠NULL
2. then Inorder(left(x));
3. output key(x);
```
```

```
4. Inorder(right(x));
```

Tracing the algorithm



Inorder : 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20
Preorder : 15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20
Postorder : 2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15

- Pre-Order Algorithm `Preorder(x)`
  Input: x is the root of a subtree.

```
1. if x≠NULL
2. then output key(x);
3. Preorder(left(x));
4. Preorder(right(x));
```

- In-Order Algorithm `Inorder(x)`
  Input: x is the root of a subtree.

```
1. if x≠NULL
2. then Inorder(left(x));
3. output key(x);
4. Inorder(right(x));
```

- Post-Order Algorithm `Postorder(x)`
  Input: x is the root of a subtree.

```
1. if x≠NULL
2. then Postorder(left(x));
3. Postorder(right(x));
4. output key(x);
```

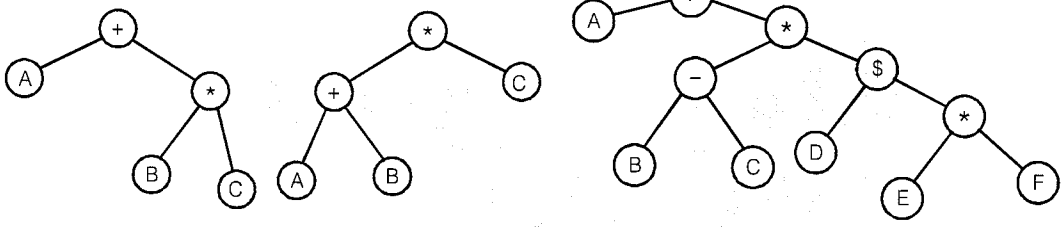
6.7.2 Non Recursive Inorder Traversal Routine

```
#define MAXSTACK 100
void intrav2(NODEPTR tree)
{
    struct stack
    {
       int top;
       NODEPTR item[MAXSTACK];
    } s;
    NODEPTR p;
    s.top=-1;
```

```
        p=tree;
        do
        {  /*travel down left branches as far as possible*/
           /*saving pointers to nodes passed*/
        }
}
```

6.8 Expression Trees

- We represent an expression containing operands and binary operators by a strictly binary tree.
- The root of the strictly binary tree contain an operator that is to be applied to the result of evaluating the expressions represented by the left and right subtrees.
- A node representing an operator is a non-leaf, where as a node representing an operand is a leaf.



(a) A + B * C (b) (A + B) * C (c) A + (B – C) * D$(E * F)

- Preorder traversal of an expression tree gives prefix, postorder gives the postfix and inorder gives the infix.

Prefix	Postfix:	Infix:
Fig(a): + A * BC Fig(b): * + ABC Fig(c): + A * – BC $ D * EF	Fig(a): ABC * + Fig(b): AB + C * Fig(c): ABC – DEF * $ * +	Fig(a): A + B * C Fig(b): A + B * C Fig(c): A + B – C * D$ * E * Ft

NOTE

If both routines, recursive and non recursive are executed, the recursive routine executes faster than non recursive routine. This goes against the accepted rule that recursion is slower than iteration. This is due to the following factors.

(a) Superfluous overflow and under flow tests.
(b) There is no extra recursion as in the case of fibonacci numbers.
(c) The recursion stack cannot be entirely eliminated as in case of factorial function. Thus automatic stacking and unstacking of built-in-recursion is more efficient.
(d) The automatic stacking of recursion does not stack any more variables than are necessary.

6.9 Binary Tree Representations

There are two types of node representation of binary tree:

1. **Linked Array Representation:** A node can be defined as

```
# define NUMNODES 500
Struct node type
```

```
{
    int info;
    int left;
    int right;
    int father;
} ;
Struct node type node[NUMNODES];
```

Once the array of nodes is declared, we could create an available list by executing the following statements.

```
int avail,i;
{
    avail=1;
    for(i=0;i<NUMNODES;i++)
    node[i].left=i+1;
    node[NUMNODES-1];
    left=0;
}
```

Note that the available list is not a binary tree but a linear list whose nodes are connected or linked together by the left field. Each node in a tree is taken from the available pool when needed and returned to the available pool when no longer in use.

2. **Dynamic Node Representation:** Alternatively a node may be defined by

```
struct node type
{
    int info;
    struct node type*left;
    struct node type*right;
    struct node type*father;
} ;
Type def struct node type*NODEPTR;
```

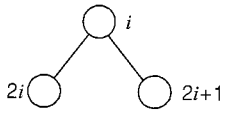
Under this implementation, there is no explicit available free list. The routines get node and free node simply allocate and free node using the routines malloc and free.

NOTE: Both the linked array representation and the dynamic node representation are implementations of an abstract linked representation (also called the node representation) in which explicit pointers link together the nodes of a binary tree.

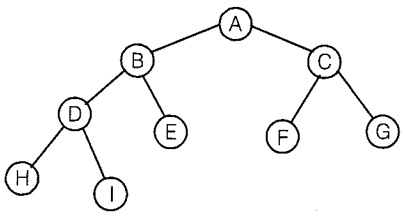
6.10 Internal and External Nodes

- Leaf nodes have no sons. Thus, in linked representation of binary trees, left and right pointers are needed only in non-leaf nodes.
- Thus using this distinction, non leaves are called **internal nodes** and leaves are known as **external nodes**.

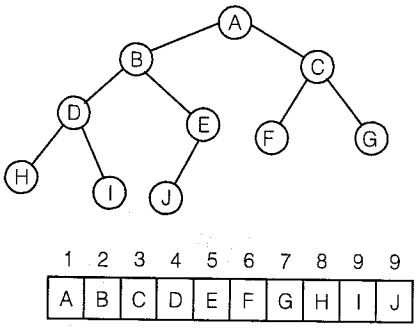
6.11 Implicit Array Representation of Binary Trees



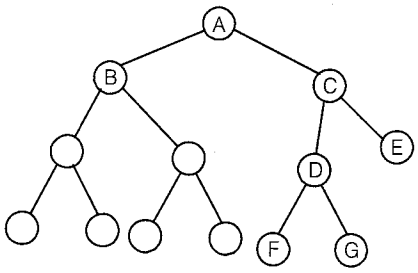
- We can extend the above implict array representation of almost complete binary trees to an implicit array representation of binary trees generally.
- We do this by identifying an almost complete binary tree that contains the binary tree being represented.

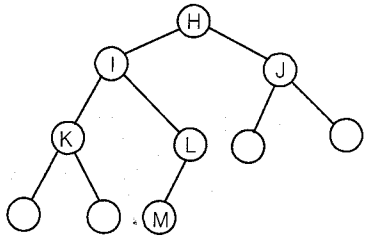


(a)



(b)





(b) Almost complete extensions

- The implicit array representation is also called the sequential representation, as contrasted with the linked representation presented earlier, because it allow a tree to be implemented in a contiguous block of memory (an array) rather than via pointers connecting widely separated nodes.

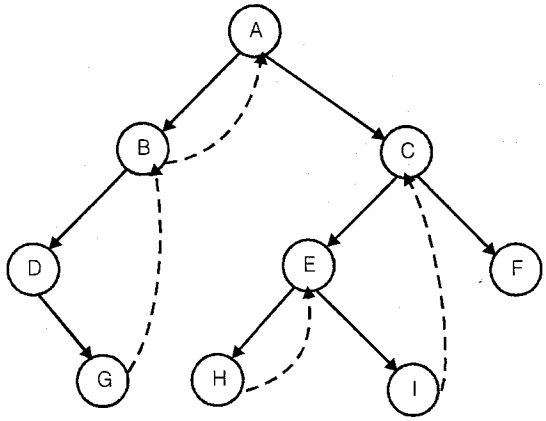
Choosing a Binary Tree Representation-Trade off's between sequential and linked implementation:

1. **Sequential:** The sequential representation is some what simpler, although it is necessary to ensure that all pointers are within the array bounds.
   - The sequential representation saves storage space for trees known to be almost complete, since it eliminates the need for the fields left, right and father and does not even require a used field.
   - It is also space efficient for trees that are only few nodes short of being almost complete, or when nodes are being successively eliminated from a tree that originates as almost complete, although a used field might then be required.
   - The sequential representation can only be used in the context in which only a single tree is required, or where the number of trees needed and each of their maximum sizes is fixed in advance.

2. **Linked Representation:**
   - Linked representation requires left, right and father fields but allows much more flexible use of the connection of nodes.
   - In linked representation, a particular node may be placed at any location in any tree, where as in sequential representation a node can be utilized only if it is needed at a specific location in a specific tree.
   - Under the dynamic node representation the total number of nodes and trees is limited only to the amount of allocated memory but it is not the case with linked representation.

6.12 Threaded Binary Trees

- Instead of containing a NULL pointer in the right field, a node with an empty right subtree contained in its right field, a pointer to the node that would be on the top of the stack at that point in the algorithm (that is, a pointer to its inorder successor).
- Such a pointer is called a thread and must be differentiable from a tree pointer that is used to link a node to its left or right subtree.
- Thus there is no longer a need for a stack, since the last node visited during a traversal of a left subtree points directly to its inoder successor.
- **Note that** the right most node in each tree still has a NULL pointer, since it has no inorder successor. Such trees are called **right-in-threaded** binary trees.



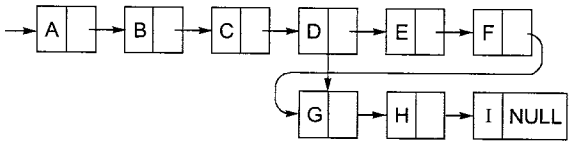
- A **left-in-threaded binary tree** may be defined similarly, as one in which each NULL left pointer is altered to contain a thread to that node is inorder predecessor.
- An in-threaded binary tree may then be defined as a binary tree that is both left-in threaded and right in-threaded.
- We may also define right and left pre-threaded binary trees, in which NULL right and left pointers of nodes are replaced by their preorder successor and predecessor respectively.
- A right pre-threaded binary tree may be traversed efficiently in preorder without the use of a stack.
- A right in-treaded binary tree may also be traversed in preorder without the use of a stack.

6.13 Traversing using Father's Field

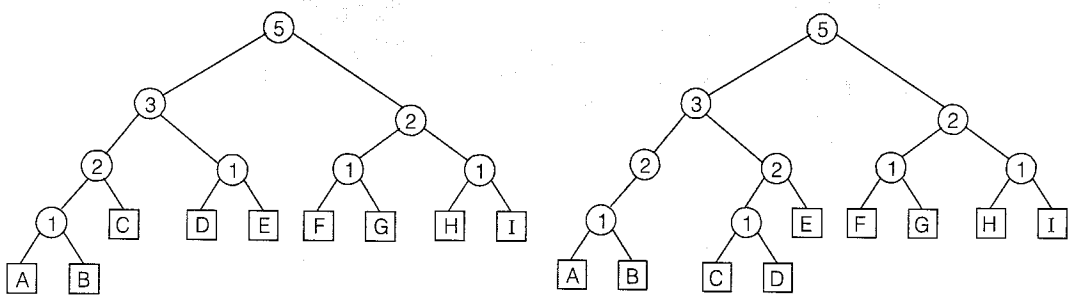
- If each tree node contains a father field, neither a stack nor threads are necessary for non-recursive traversal.
- Instead, when the traversal process reaches the left node, the father field can be used to climb back up the tree.

- Traversal using father pointers for backing up is less time efficient than traversal of a threaded tree. A thread points directly to a node's successor whereas a whole series of father pointers may have to be followed to reach the successor in an unthreaded tree.
- It is difficult to compare the time efficiency of stack based traversal and father-based traversal since the former includes the overhead of stacking and unstacking.
- Another technique for non-recursive traversal without father field is that simply reverse the son pointer on the way down the tree so that it can be used to find a way back up. But this also is inefficient in a multiuser environment because if one user is temporarily modifying the pointers then another user will not see the tree as coherent structure.

6.14 Representing Lists as Binary Trees



(a)

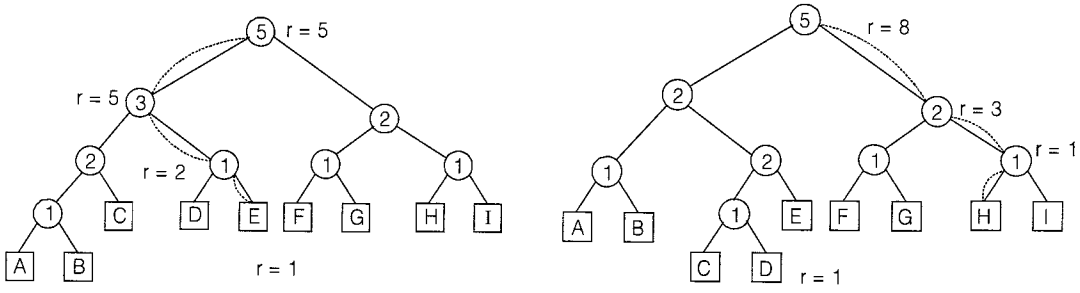


(b) (c)

Representation

- Element of the original list are represented by the leaves of the tree, whereas the non-leaf nodes of the tree are present as a part of the internal tree structure.
- Associated with each leaf node are the content of the corresponding list element.
- Associated with each non-leaf node is a count representing the number of leaves in the node's left subtree.
- The elements of the list in their original sequence are assigned to the leaves of the tree in the inorder sequence of the leaves.

Finding the Kth Element



(a) (b)

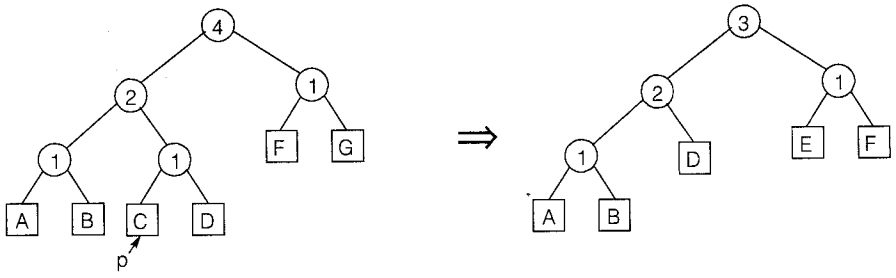
- Let tree points to the root of the tree
- count(p) represent the count associated with the non-leaf node pointed by p.

```
r=k;  //r keeps the count and K is the element to find
  p=tree;
  while(P is not a leaf node)
  if(r≤count(p))
  P=left(p)
  else{
       r-=count(p);
       p=right(p);
  }    /*end if */
  find=P;
```

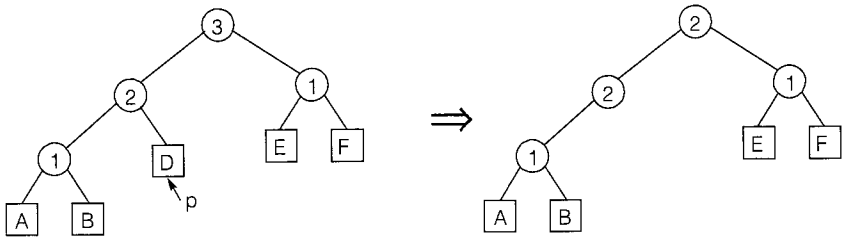
- The number of nodes examined in finding the 5th element is less than equal to the depth of the tree.
- **Comparison between binary tree and list as a tree:**
  (a) Although list as a binary tree is not a very impressive saving, consider a list with 1000 elements. A binary tree of depth 10 is sufficient to represent such a list, since $\log_2 1000$ is less than 10, thus finding the Kth element using such a tree requires no more than 11 nodes accesses.
  (b) If an almost complete tree is used, the kth element of an n-element list can be found in almost $\log_2 n+1$ node accesses, whereas k accesses would be required if a linear linked list were used.

Deleting an Element

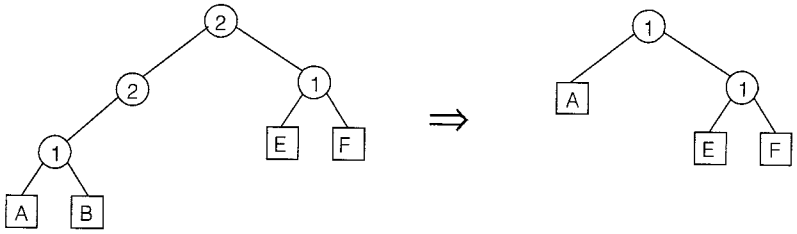
- The deletion algorithm involves inspection of up to two nodes (the ancestor of the node being deleted and the ancestor's brother) at each level. Thus, the operation of deleting the Kth element of a list represented by a tree (which involves finding the element and then deleting it) requires a number of node accesses approximately equal to three times of the tree depth
- Although deletion from a linked list requires accesses to only three nodes, deleting the Kth element requires a total of k+2 access ((k – 1) of which are to located the node preceding the kth.)
- For large lists, therefore, the tree representation is more efficient.



(a)



(b)



(c)

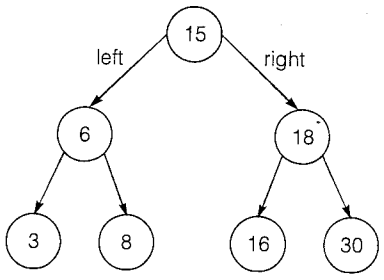
- Comparison of efficiency of tree represented lists with array represented lists:
  (a) If an n-element list is maintained in the first n-elements of an array, finding the k-th element involves only a single array access, but deleting it requires shifting (n-k) elements that had followed by the deleted element.
  (b) If gaps are allowed in the array so that selection can be implemented efficiently (by setting a flag in the array position of the deleted element without shifting any subsequent elements), finding the k^{th} element requires at least k array accesses. The reason is that it is no longer possible to know the array position of the Kth element in the best since gaps may exist among the elements in the array.

Inserting an Element

Inserting a new Kth element into a tree-represented list (between the $(k-1)^{st}$ and the previous k^{th}] is also relatively efficient operation. The insertion consists of allocating the vth element, replacing it with a new non leaf that has a leaf containing the new element as it's left son and the leaf containing the old Kth elements as its right son, and adjusting the count among its ancestors.

6.15 Binary Search Tree

- Node x
- Data in Left(x) is less than x
- Data in Right(x) is greater than x
- **Notation:** Key(x) means the data is in Node x.



- Expression Trees
  (a) (a+(b*c)) + (((d*e)+f)*g)
  (b) General evaluation strategy: inorder traversal
  Left, op, Right
  (c) Postorder
  Left, Right, op
  (d) Preorder:
  Op, Left, Right

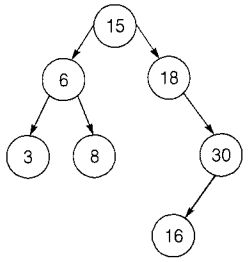
ADT for BST

BST as Abstract Data Type

- Insert(x) // Insert item
- Find(x) // find item
- Delete(x) // delete item
- Traverse() // visit nodes in tree (several different ways)
- Minimum() // find minimum
- Maximum() // find maximum
- Successor(x) // find x's successor

Data Structure

```
Struct Node
{
   int data;
   Node*left;
   Node*right;
} ;
```

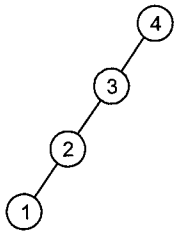


- Keep a pointer called "root" to the tree.
- All you need to maintain is node root to access any nodes in the tree!
- This is like a linked-list. Remember for linked list we only need to store the "head" pointer
  Reminds you of linked-list? Remember, x = x → next;
  Now we have:
  if (x→left)
  x = x→left; (more to left)
  if (x→right)
  x = x→right; (more to right)

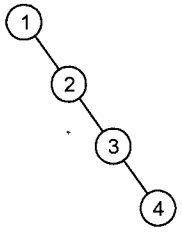
Skew Binary Search Tree

When the hight of a BST goes till 'n' with 'n' elements then the tree is said to be skew BST. There are two types of skew binary search tree:

1. Left skewed BST
2. Right skewed BST



Left skew BST with element 1,2,3 and 4



Right skew BST with element 1,2,3 and 4

NOTE

Prove that the depth of a complete binary tree has depth log(N)

- Prove by induction that number of nodes at depth d is 2^d
- Total number of nodes of a complete binary tree of depth d is $1+2+4+...2^d = 2^{d+1} - 1$
  Thus $2^{d+1} - 1 = N$
  $d = \log(N + 1) - 1$

6.15.1 Searching the Tree

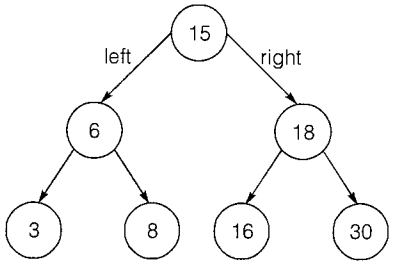
Search(x,k)

Check if "k" is in the Tree.

Algorithm Search(x, k)

Input: x is the root of the tree and k is the input search key.

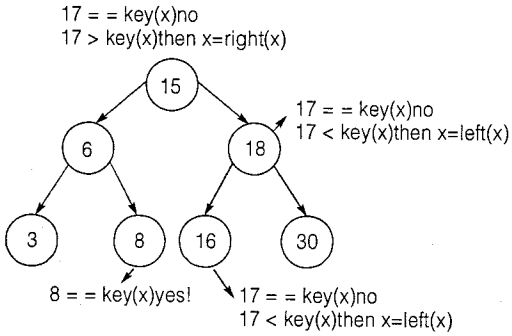
Output: the node containing k or NULL.




```
1. while x≠NULL
2. do if k=key(x)
3. then return x;
4. else
5. if k<key(x)
6. then x:=left(x);
7. else x:=right(x);
8. return NULL;
```

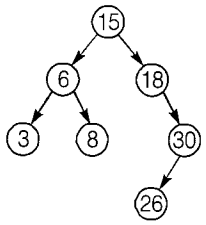
Running time = O(tree height)

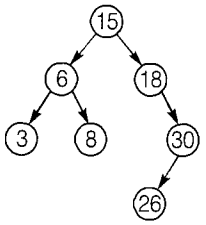
Example: Search(root,17)




```
X==NULL
Stop...Not found
```

6.15.2 Find Min and Max





- Minimum element is always the left-most node
- Maximum element is always the right most node
- Time complexity
- Worst case-the height of the tree
- Algorithm Minimum(x)

  Input: x is the root.

  Output: the node containing the minimum key.

  ```
  1. while left(x)≠NULL
  2. do x:=left(x)
  3. return x;
  ```

- Algorithm Maximum(x)

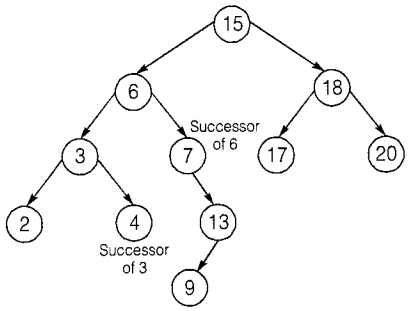
  Input: x is the root.

  Output: the node containing the maximum key.

  ```
  1. while right(x)≠NULL
  2. do x:=right(x)
  3. return x;
  ```

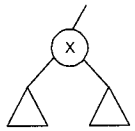
6.15.3 Successor

- The successor of a node x is defined as: The node y, whose key(y) is the successor of key(x) in sorted order of this tree. (2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20)

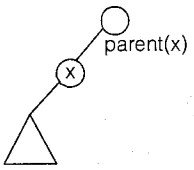


Three Scenarios to Determine Successor:

- **Scenario-1:** Node x has a right subtree.
  By definition of BST, all items greater than x are in this right sub-tree.
  Successor is the minimum (right(x)).

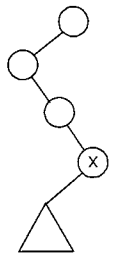


- **Scenario-2:**
  Node x has no right subtree.
  X is the left child of parent(x).
  Successor is the parent(x).



  Why?
  The successor is the node whose key would appear in the next sorted order. Think about inorder traversal.

- **Scenario-3:**
  Node x is not a left child of an immediate parent.
  Keep moving up the tree until you find a parent which branches from the left( ).
  Must traverse up the tree until we find a suitable parent.
  Stated as in Pseudocode::




```
y=parent(x);
while y≠NULL and x=right(y)
do x:=y;
 x=parent(y);
```

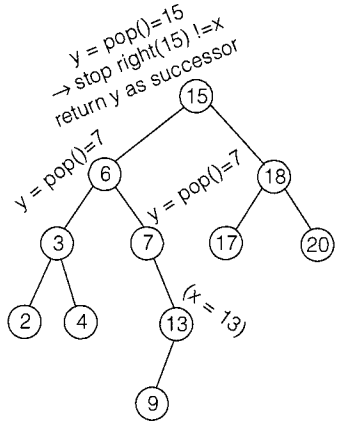
- **Algorithm Successor(x)**
  Input: x is the input node.

```
1. if right(x)≠NULL
2. then return Minimum(right(x)); [Scenario-1]
3. else
4. y:=parent(x);
5. while y≠NULL and x=right(y)
6. do x:=y; [Scenario-2 and 3]
7. y:=parent(y);
8. return y;
```

- Successor (r, x)

  Input: r is the root of the tree and x is the node.

```
1.  initialize an empty stack S;
2.  while key(r) ≠ key(x)
3.  do push(S,r);
4.  if key(x)<key(r)
5.  then r:=left(r);
6.  else r:=right(r);
7.  if right(x) ≠ NULL
8.  then return Minimum(right(x));
9.  else
10. if S is empty
11. then return NULL;
12. else
13. y:=pop(S);
14. while y≠NULL and x=right(y)
15. do x:=y ;
16. if S is empty
17. then y:=NULL;
18. else y:=pop(S);
19. return y;
```



Successor(root, 13) Part-II

```
Find Parent (Scenario-3)
y=s.pop()
while y!=NULL and x=right(y)
x=y;
if s.is empty()
y=NULL;
else
y=s.pop();
loop
return y;
```

7
6
15

Stack S

6.15.4 Binary Search Tree Insertion

Algorithm Insert (T, z)

Input: T is the tree and z is the node to be inserted.

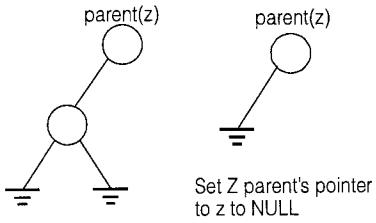
```
/*assume that left(z)=right(z)=NULL.*/
/*assume that key(z) is not in T.*/
1. y:=NULL;
2. x:=root(T);
3. while x≠NULL
4. do y:=x;
5. if key(z)<key(x)
6. then x:=left(x);
7. else x:=right(x);
8. parent(z):=y;
9. if y:=NULL
10.       then root(T):=z;
11.                    else
12.       if key(z)<key(y)
13.       then left(y):=z;
14.      else right(y):=z;
```

6.15.5 Binary Search Tree Deletion

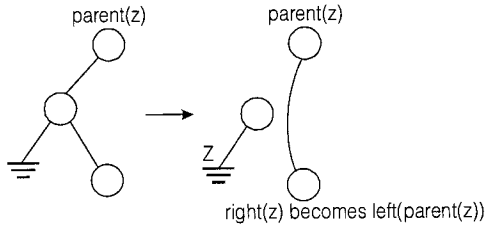
Delete node "Z" from Tree.

Three Cases for Deletion:

Case-1: Node z is a leaf

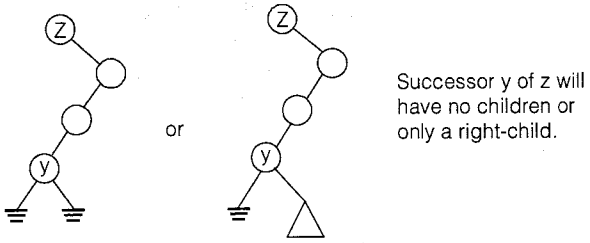


Case-2: Node z has exactly 1 child

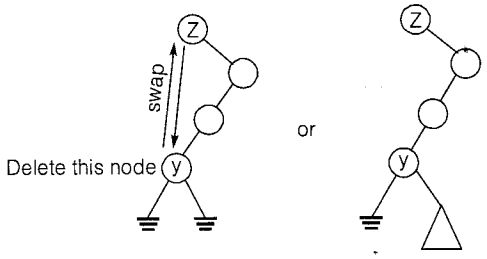


Case-3: Node z has 2 children

- **Step 1:**Find succesor y of 'z' (i.e. y = successor(Z))
  Since z has 2 children, successor is y=minimum(right(z))



- **Step 2:** Swap keys of z and y. Now delete node y (which now has value z)! This deletion is either case 1 or 2.

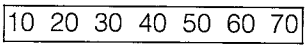


(deletion of node "y" is always going to be Case I or II)

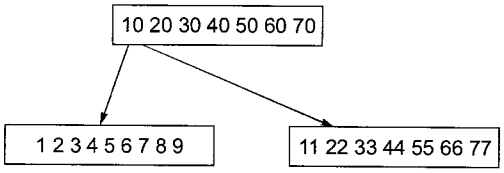
6.16 B-Tree

- Up to now, all data that has been stored in the tree has been in memory.
- If data gets too big for main memory, what do we do?
- If we keep a pointer to the tree in main memory, we could bring in just the nodes that we need.
- For instance, to do an insert with a BST, if we need the left child, we do a disk access and retrieve the left child.
- If the left child is NULL, then we can do the insert, and store the child node on the disk.
- Of course, this is silly for a BST.
- The problem with BST or even RB(Red Black) Tree: storing the data requires disk accesses, which is expensive, compared to execution of machine instructions.
- If we can reduce the number of disk accesses, then the procedure runs faster.

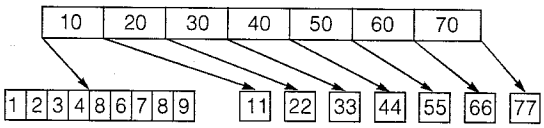
- The only way to reduce the number of disk accesses is to increase the number of keys in a node.
- The BST allows only one key per leaf.
- If we increase the number of keys in the nodes, how will we do any tree operations effectively?



- Above is a node with 7 keys. How do we add children?



- Clearly, the tree above is useless.
- So, just like the BST, place the keys in child nodes



- So, now we can easily find any value.
- We need to create operations, which require rules on what makes a tree a B-Tree.
- Clearly, having one key per node would be very bad.
- We need a mechanism to increase the height of the tree (since the number of keys in any node can get very high) so we can shift keys out of a node, making the nodes smaller.
- A B-Tree is a rooted tree (whose root is root[T]) having the following properties:

Every internal node x has the following fields:

$n[x]$	$key_1[x]$	$key_2[x]$	$key_3[x]$	$key_4[x]$	$key_5[x]$	$key_6[x]$	$key_7[x]$	key_8	$leaf[x]$
$c_1[x]$	$c_2[x]$	$c_3[x]$	$c_3[x]$	$c_4[x]$	$c_5[x]$	$c_6[x]$	$c_7[x]$	$c_8[x]$	$c_9[x]$

- n[x] is the number of keys in the node, n[x] = 8 above.
- leaf[x] = false for internal nodes, since x is not a leaf.
- The $key_i[x]$ are the values of the keys, where $key_i[x] \le key_{i+1}[x]$.
- $c_i[x]$ are pointers to child nodes. All the keys in $c_i[x]$ have values that are between $key_{i-1}[x]$ and $key_i[x]$.
- Leaf nodes have no child pointers.
- leaf[x] = true for leaf nodes.
- All leaf nodes are at the same level.

$n[x]$	$key_1[x]$	$key_2[x]$	$key_3[x]$	$key_4[x]$	$key_5[x]$	$key_6[x]$	$key_7[x]$	key_8	$leaf[x]$

- There are lower and upper bounds on the number of keys a node can contain. This depends on the **"minimum degree" t ≥ 2**, which we must specify for any given B-Tree.
  (a) Every node other than the root must have at least t – 1 keys. Every internal node other than the root thus has t children. If the tree is nonempty, the root must have at least one key.
  (b) Every node can contain at most 2t – 1 keys. Therefore, an internal node can have at most 2t children. A node is full if it contains exactly 2t – 1 keys.

- Height of B-Tree:
  (a) If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, height = $h \leq \log_t[(n+1)/2]$
  (b) The important thing to notice is that the height of the tree is log base t. So, as t increases, the height, for any number of nodes n, will decrease.
  (c) Using the formula $\log_a x = (\log_b x) / (\log_b a)$, we can see that
  $\log_2 10^6 = (\log_{10} 10^6)/(\log_{10} 2) \approx 6/0.30102999566398 \approx 19 = \log_{10} 10^6 = 6$
  So, 13 less disk accesses to get to the leaves.

6.16.1 Basic Operations on B-Tree

- The root of the B-tree is always in main memory, so that a Disk-Read on the root is never required; a Disk-Write of the root is required, however, whenever the root node is changed.
- Any nodes that are passed as parameters must already have had a Disk-Read operation performed on them.

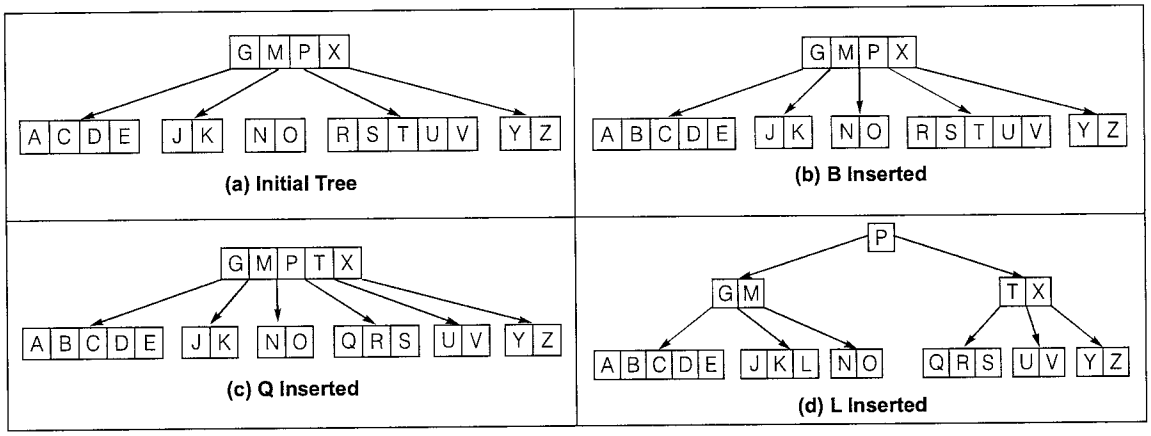
Searching a B-Tree

```
B-TREE-SEARCH(x,k)
1. i←1
2. while i≤n[x] and k>key_i[x]
3. do i←i+1
4. if i≤n[x] and k=key_i[x]
5. then return(x,i)
6. if leaf[x]
7. then return NIL
8. else DISK-READ(c_i[x])
9. return B-TREE-SEARCH(c_i[x],k)
```

Inserting into B-trees

Just keep in mind that you are starting at the root, and then finding the subtree where the key should be inserted, and following the pointer. But you must keep in mind that a deletion may eventually occur, and sometimes deletions force keys into their parents. So, if we encounter a full node on our way to the node where the insertion will take place, we must split that node into two.

B-Tree-Insert



B-Tree-Split-Child

```
B-tree split-child (x,i,y)
1. z←Allocate-node()          ← z is the new node
2. leaf[z]←leaf [y]           ← If y is a leaf so is z, and z will have t–1 keys.
3. n[z]←t-1
4. for j←1 to t - 1           ← This is how z get the t – 1 keys (from y)
5. do key_j[z]←key_j+1[y]
6. If not leaf[y]             ← If y has children, some must be moved over to z
7. then for j←1 to t
8. do c_j[z]← c_j+1[y]
9. n[y]← t -1                 ← y will have t–1 keys when done
10.for j←n[x]+1
   down to i+1
11.do c_j+1[x]←c_j[x]         ← Move the child pointers to the right to make
                                room for z, then place z.
12.c_j+1[x]←z
13.for j←n[x]down to i
14.do key_j +1[x]←key_i[x]    ← Shift over the keys in x to make room for the key
                                moving up to x from y, and then increase the degree of x.
15.key_i[x]←key_t[y]
16.n[x]←n[x]+1
17.Disk-write(y)
18.Disk-write(z)
19.Disk-write(x)
```

B-Tree-Insert-Nonfull

```
B-TREE-INSERT-NONFULL(x,k)
1. i←n[x]
2. if leaf[x]
3. then while i≥1 and k<key_i[x]
4. do key_i + 1[x]←key_i[x]
5. i←i-1
6. key_i+1[x]←k
7. n[x]←n[x]+1
8. DISK-WRITE(x)
9. else while i≤1 and k<key_i[x]
10.                do i←i-1
11.                   i←i+1
12.        DISK-READ(c_i[x])
13.        if n[c_i[x]]=2t-1
14.then B-TREE-SPLIT-CHILD(x,i,c_i[x])
15.             if k>key_i[x]
16.              then i←i+1
17.B-TREE-INSERT-NONFULL(c_i[x],k)
```

6.16.2 Deleting Keys from the Nodes

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following.
   (a) If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)
   (b) Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass).

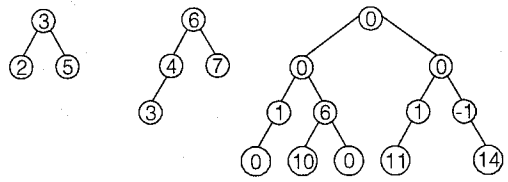
(c) Otherwise, if both y and z have only t – 1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t – 1 keys. Then, free z and recursively delete k from y.

3. If the key k is not present in internal node x determine the root $c_i[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. If $c_i[x]$ has only t – 1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.
   (a) If $c_i[x]$ has only t – 1 keys but has an immediate sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$ moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $c_i[x]$.
   (b) If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have t – 1 keys, merge $c_i[x]$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

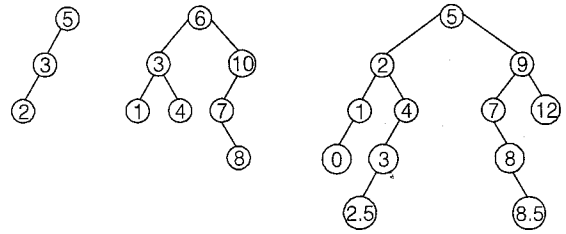
6.17 AVL Tree

An AVL tree (also called an "admissible tree") is a tree in which the height of the left and right subtrees of *every* node differ by *at most one* - referred to as "height-balanced".

Example:



Example of non-AVL trees:



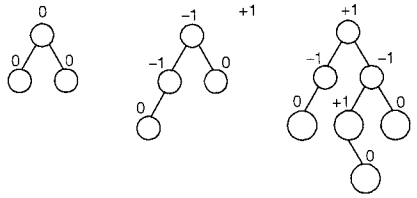
In order to indicate the differences between the heights of the right and left subtrees of a given (root) node, a **balance factor** is defined for that node of the subtree. We define the balance factor, BF:

BF = (height of right subtree – height of left subtree)

So, BF = –1, 0 or +1 for an AVL tree.

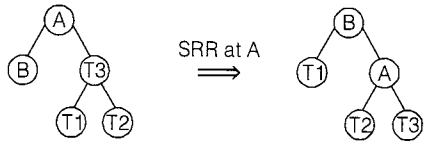
Balance factors for example AVL trees (node key values not shown): +1, –1 and 0.

Balance factors for an example non-AVL tree (node key values not shown): 2, –2, +3, –3+. . .

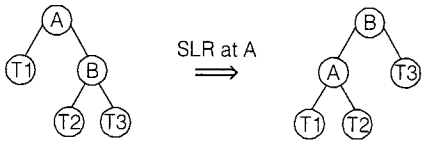


When the AVL property is lost we can rebalance the tree via one of four rotations:

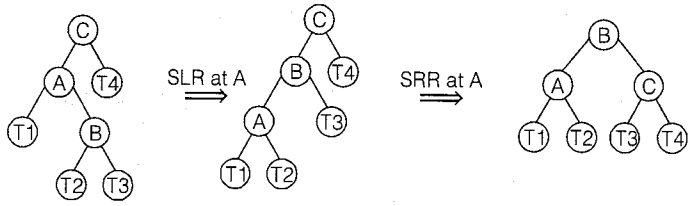
1. **Single Right Rotation (SRR):** A is the node that the rotation is performed on. This rotation is performed when A is unbalanced to the left (the left subtree is 2 higher than the right subtree) and B is left-heavy (the left subtree of B is 1 higher than the right subtree of B). T1, T2 and T3 represent subtrees (a node was added to T1 which made B left heavy and unbalanced A).



2. **Single Left Rotation (SLR):** A is the node that the rotation is performed on. This rotation is performed when A is unbalanced to the right (the right subtree is 2 higher than the left subtree) and B is right-heavy (the right subtree of B is 1 higher than the left subtree of B). T1, T2 and T3 represent subtrees (a node was added to T3 which made B right-heavy and unbalanced A).

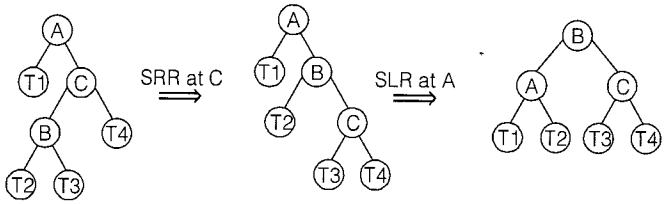


3. **Double Left Rotation (DLR):** C is the node that the rotation is performed on. This rotation is performed when C is unbalanced to the left (the left subtree is 2 higher than the right subtree), A is right-heavy (the right subtree of A is 1 higher than the left subtree of A) and B is left-heavy. T1, T2, T3, and T4 represent subtrees (a node was added to T2 which made B left-heavy, made A right-heavy and unbalanced C). This consists of a single left rotation at node A, followed by a single right at node C.



That is,

4. **Double right rotation:** A is the node that the rotation is performed on. This rotation is performed when A is unbalanced to the right (the right subtree is 2 higher than the left subtree), C is leftheavy (the left subtree of C is 1 higher than the right subtree of C) and B is right-heavy. T1, T2, T3, and T4 represent subtrees (a node was added to T3 which made B right-heavy, made C left-heavy and unbalanced A). This consists of a single right at node C, followed by a single left at node A.



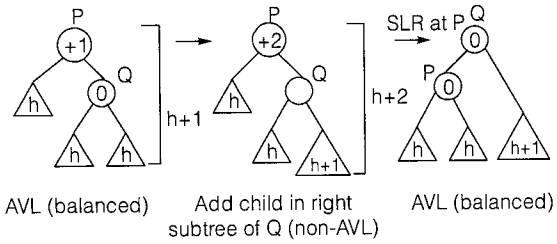
That is, DRR equiv SRR + SLR

6.17.1 Operations on AVL-Tree

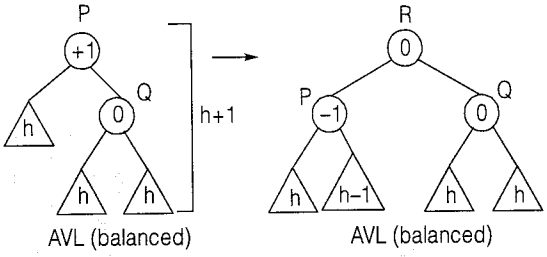
Insertion in a AVL-Tree

An AVL tree *may* become out of balance in two basic situations: (1) After inserting a node in the right subtree of the right child. (2) After inserting a node in the left subtree of the right child.

1. **Insertion of a node in the right subtree of the right child:**
   This involves a SLR about node P:

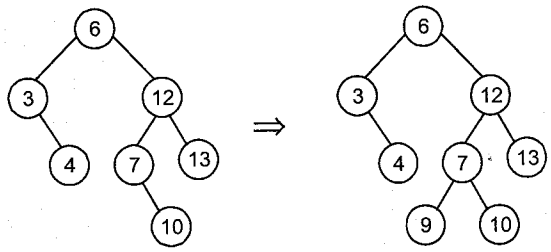


2. **Insertion of a node in the left subtree of the right child:**
   This involves a DRR:



In each case the tree is rebalanced locally (since there is no need to modify the balance factors higher in the tree). Since rebalancing is local, insertion requires one single or one double rotation, i.e. O(constant) for rebalancing. Of course, there is still the cost of search for insertion (see below). Experiments have shown that 53% of insertions do not bring the tree out of balance.

Example: Given the following AVL tree, insert the node with value 9:



Deletion in a AVL-Tree

It can be shown that deletion requires O(log N) rotations in the worst-case. However, experiments show that 78% of deletions do not require rebalancing.

6.17.2 Performance for Searching in an AVL Tree

We can show that the maximum number of comparisons in an *optimum shaped tree* is:

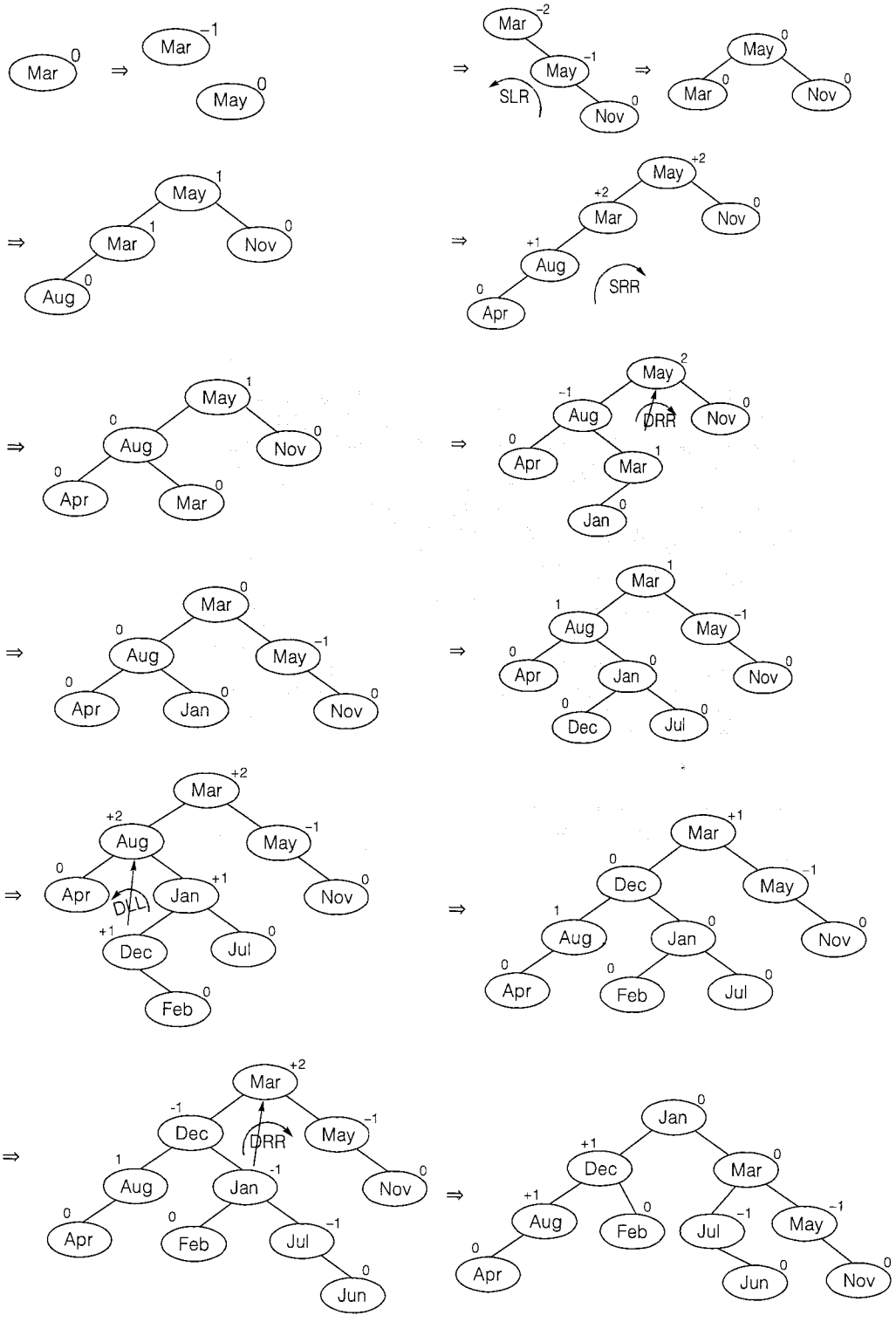
$$C > 2\lceil \log(N + 1) \rceil + 1$$

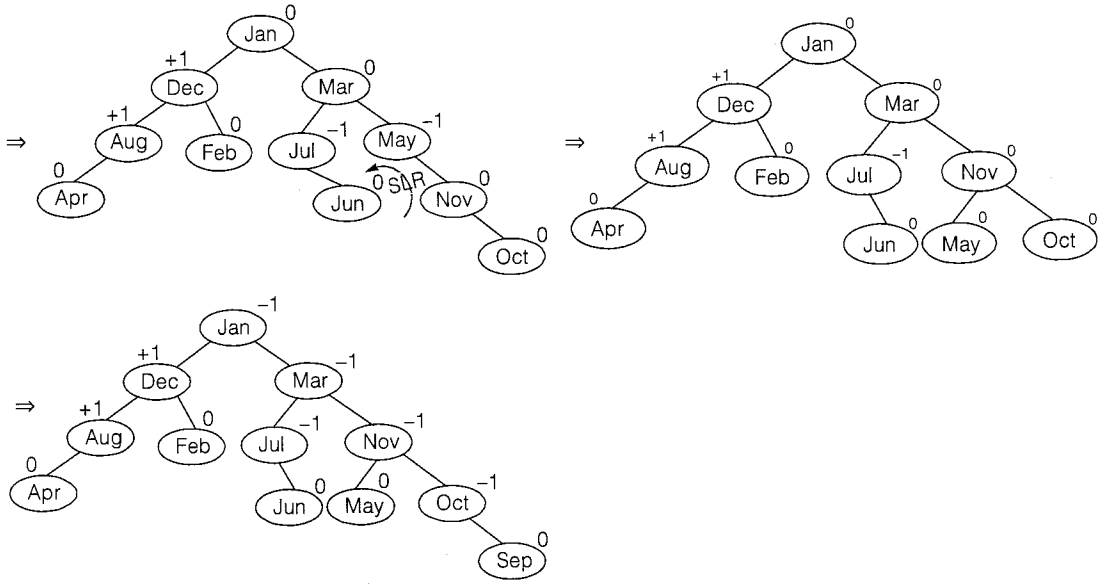
The number of comparisons for a worst-case AVL tree is obtained by evaluating an AVL tree with the greatest possible height and minimum number of nodes.

This type of AVL tree is called a *Fibonacci AVL trees*. The maximum number of comparisons in the deepest possible Fibonacci AVL tree is $C' = 2h + 1 < 2.88 \times \log (N + 2) - 1.66$

That is, AVL searches can require no more than about 44% more comparisons than required in the most costly search of an optimum shaped AVL tree. Also, much better than the O(N) worst-case result for a normal (non-AVL) binary search tree (Skew BST).

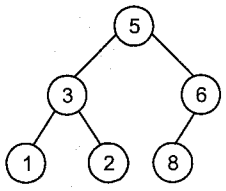
Example of AVL Tree Balancing with the following insertions:
Mar, May, Nov, Aug, Apr, Jan, Dec, Jul, Feb, Jun, Oct, Sep.





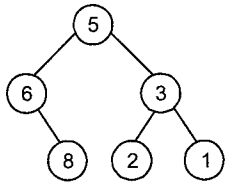
Mirror Image of Binary tree

Input: Take a binary tree i.e., need not binary search tree.




```
struct node
{   int value;
    struct node * left;
    struct node * right;
};
void mirror (struct node * node)
{   if (node == NULL)
    return '0';
    else
    {   struct node * temp;
        mirror (node → left);
        mirror (node → right);
        temp = node → left;
        node → left = node → right;
        node → right = temp;
    }
}
```

Output:



Time complexity to find mirror image of binary tree will take O(n) time.

Summary

- Tree is defined as T (V, E) where V = set of vertices and E = set of edges.
- In a tree there is exactly one edge between a pair of vertices and there are no cycles and self loops.
- The degree of a node in the tree is defined as the maximum number of children that node can hold.
- A tree can never be empty, but binary tree may be empty.
- Internal node (*I*): A node having minimum of 1 child is known as internal node.
- Leaf node (*L*): A node having no children.
- External node (*E*): The replacement of null leaves are external nodes.
- For '*n*' nodes there are $(n-1)$ edges.
- Let n_0 = number of nodes with degree 0.
  n_2 = number of nodes with degree 2.
  $n_0 = n_2 + 1$
- (In heap array) If child is at location *i*(index starts from 0) then parent is at location $\lfloor (i-1)/2 \rfloor$.
  If parent is at location *i*(index starts at 0) then left child is at location $2i + 1$ and Right child is at location $2i+2$.
- (In heap array) If parent is at location *i*(indexing at 1) then Left child is at location $2i$ and Right child is at location $2i+1$.
  If child is at location *i*(indexing at 1) then parent is at location $\lfloor i/2 \rfloor$.
- In a linked representation of binary tree, if there are '*n*' nodes then number of NULL links = n +1.
- **Complete Binary Tree** is a tree in which nodes are inserted/filled from left to right. All the leaves may not be at the same level.
- **Full Binary Tree** is a tree in which every node has exactly 2 children and all the leaves are at the same level. All full binary trees are complete binary trees but vice versa is not true.
- **Strict Binary Tree** is a tree in which each node has exactly 0 or 2 children.
- A complete *n*-ary tree is one in which every node has '0' or '*n*' sons. If '*x*' is the number of internal nodes of a complete *n*-any tree, the number of leaves in it is: $x(n-1)+1$.
- Let T(n) be the number of different binary search trees on *n*-distinct elements, then
  $$T(n) = \sum_{K=1}^{n} T(k-1)\,T(n-k+1).$$
- In binary tree, Rank (or) Index of any node: Rank (i) = (Number of nodes in left subtree of i) + 1
- In a *m*-ary tree (degree = *m*), if n_i is number of nodes of degree 'i'
  (i = 0, 1,..., *m*) then n_0(leaf nodes) = $1+\sum_{i=2}^{m}(i-1)n_i$
- Maximum size of array to store a binary tree with '*n*' nodes = $2^n - 1$.
- Minimum size of array to store a binary tree with '*n*' nodes = $2^{\lceil \log(n-1) \rceil} - 1$.

- Maximum height possible for a binary tree with 'n' nodes = n
- Minimum height possible for a binary tree with 'n' nodes = $\log_2 n$
- Maximum number of nodes in a binary tree of height 'h' = $2^{h+1} - 1$.
- Total number of binary trees possible with n nodes = ${}^{2n}C_n / (n+1)$
- For non-empty binary tree, if n is number of nodes and e is the number of edges, then $n = e + 1$.
- **Traversal Techniques:** (1) Preorder (Root, Left, Right). (2) Inorder (Left, Root, Right). (3) Postorder (Left, Right, Root).
- **Binary Search Tree (BST):** (a) Left child < Root < Right Child. (b) Recursion of left child from the root gives the minimum element and Recursion of right child from the root gives the maximum element.
- Inorder traversal of BST is called "Binary Sort". Sorted order of data results in ascending order.

	Average Case	Worst Case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Build		$O(n^2)$

- **"Deletion of a node" in BST:** (*i*) If node has no child, simply delete that node. (*ii*) If node has 1 child, delete that node and replace it with child. (*iii*) If node has 2 children, delete that node and replace it with inorder successor or predecessor of that node.
- **AVL Search Tree:** Devised by *Adelson Velski Landis*

  Height balanced binary search tree

  Balance factor (b_f) is :

  $|b_f| = |h_L - h_R| \le 1$

  $b_f = -1$ or 0 or 1
- Rotation is a technique used in the AVL tree to balance the height.
- **Types of rotations:** (*i*) Right-Right [RR] Rotation (Single Rotation). (*ii*) Left-Left [LL] Rotation (Single Rotation). (*iii*) Left-Right [LR] Rotation (Double Rotation). (*iv*) Right-Left [RL] Rotation (Double Rotation).
- Minimum number of nodes in a AVL tree of height 'h'

$$N_{min}(H) = \begin{cases} 1 & ; \ L = 0 \\ 2 & ; \ L = 1 \\ 1 + N(H-1) + N(H-2) & ; \ L > 1 \end{cases}$$

- Time complexity to find height of a tree is O(n).
- Time complexity to check given tree is BST or not is O(n).
- Time complexity to find levels of a tree is O(n).

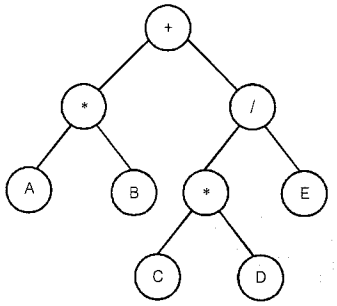
Student's Assignments

Q.1 The maximum number of nodes in a binary tree of height k is

(a) $2^{k+1} - 1$; k > = 1 (b) 2^{k-1}, k > = 0

(c) $2^k + 1$, k > = 1 (d) $2^{k+1} + 1$, k > = 1

Q.2 From the following tree write the inorder traversal



(a) AB * CD * E/+ (b) A * B + /C * DE

(c) A * B + C*D/E (d) None of these

Q.3 Which of the following represents pre-order traversing of the given tree?

(a) DBEFAGCHI (b) ABDEFCGHI

(c) ABEFDCIG (d) DBAFEGCIH

Q.4 In tree construction, which one will be suitable and efficient data structure?

(a) array (b) linked list

(c) stack (d) queue

Q.5 Binary heap data structure look like

(a) Tree

(b) Complete binary tree

(c) Graph

(d) None of these

Q.6 Which of the following is max-heap property if A is an array and i is the index of array?

(a) A[parent (i)] ≥ A[i]

(b) A[parent (i)] > A[i]

(c) A[parent (i)] ≤ A[i]

(d) A[parent (i)] < A[i]

Q.7 A complete n-ary tree is one in which every node has 0 or n sons. If x is the number of internal nodes of a complete n-ary tree the number of leaves in it is given by

(a) x(n –1) (b) n(x – 1)

(c) xn + 1 (d) x(n + 1)

Q.8 If a binary tree transvered inorder then numbers of nodes will printed in

(a) Ascending order (b) Descending order

(c) Random order (d) None of these

Q.9 Let T be a binary search tree with n nodes and S_n be the average number of comparisons required for a successful search and u_n be the average number of comparisons required for an unsuccessful search. Then which of the following relation holds true?

(a) $S_n = \frac{u_n + n}{n-1}$ (b) $S_n = \left(\frac{n+1}{n}\right)u_n - 1$

(c) $S_n = \left(\frac{n-1}{n}\right)u_n + 1$ (d) $S_n = \left(\frac{n-1}{n}\right)u_n - 1$

Common Data Questions (10 and 11):

Consider the following elements in the construction of AVL-Tree 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7.

Q.10 How many RR and RL imbalances occurred during the construction of AVL-Tree?

(a) 2, 4 (b) 2, 3

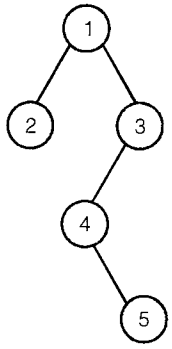
(c) 1, 3 (d) 1, 4

Q.11 What is root element of AVL-Tree?

(a) 21 (b) 4

(c) 14 (d) 15

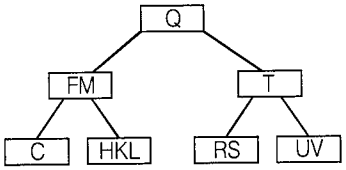
Q.12 In the given binary tree, using array (the array starts from index 1), you can store the node 4 at location



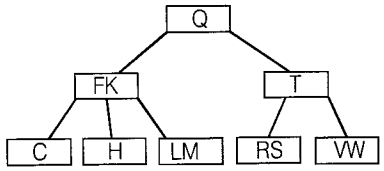
(a) 6th (b) 4th
(c) 5th (d) 3rd

Q.13 Show the B-tree of order 4 after insertion of F, S, Q, K, C, L, H, T, V, W, M, R

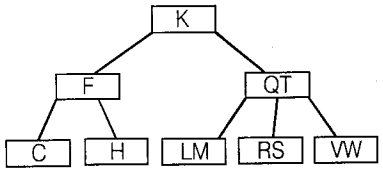
(a)



(b)



(c)



(d) None of the above

Q.14 The worst-case height of AVL tree with n nodes is
(a) 2 log n (b) n log (n + 1)
(c) 1.44 log (n + 2) (d) 1.44 n log n

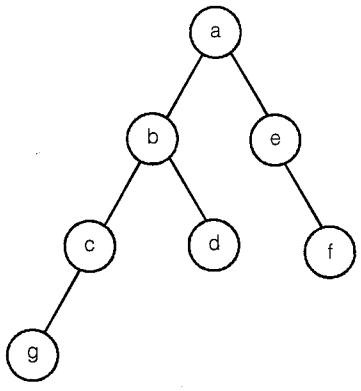
Q.15 What is primary distinction between B-tree indices and B^+ – tree indices?
(a) B-tree eliminates the redundant storage of search key values.
(b) B^+ tree eliminates the redundant storage of search key indices.
(c) Deletion in a B-tree is more complicated.
(d) Lookup B-tree is proportional to the logarithm of the number of search keys.

Q.16 A B-tree is of order p and consists of n keys. Find the maximum height.

(a) $\log_{\left(\frac{p}{2}\right)} \frac{n+1}{2}$ (b) $\log_p n$

(c) $\log_{(p/2)}(n+1)$ (d) None of these

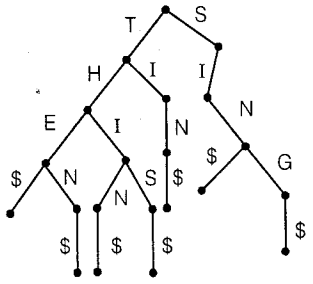
Q.17 In the balanced binary-tree in figure given below, how many nodes will be come unbalanced when a node is inserted as a child of the node "g".



(a) 1 (b) 3
(c) 7 (d) 8

Q.18 Which of the following sequences denotes the post order traversal sequence of the tree of question 17?
(a) fegcdba (b) gcbdafe
(c) gcdbfea (d) fedgcba

Q.19 Below structure represents a



(a) Binary tree (b) Tree
(c) Balance tree (d) All of these

Q.20 If one uses straight two way merge sort algorithm to sort the following elements in ascending order
20, 47, 15, 8, 9, 4, 40, 30, 12, 17
Then the order of these elements after second pass of the algorithm is
(a) 8, 9, 15, 20, 47, 4, 12, 17, 30, 40
(b) 8, 15, 20, 47, 4, 9, 30, 40, 12, 17
(c) 15, 20, 47, 4, 8, 9, 12, 31, 40, 17
(d) 4, 8, 9, 15, 20, 47, 12, 17, 30, 40

Q.21 Which of the following is correct?

(a) B-trees are for storing data on disk and B^+ trees are for main memory.

(b) Range queries are faster on B^+ trees.

(c) B-trees are for primary indexes and B^+ trees are for secondary indexes.

(d) The height of a B^+ tree is independent of the number of records.

Q.22 Consider the following nested representation of binary trees : (x, y, z) indicates y and z are the left and right sub trees, respectively, of node x, note that y and z may be null, or further nested, which of the following represents a valid binary tree?

(a) (1, 2 (4, 5, 6, 7))

(b) (1(2, 3, 4) 5, 6)7)

(c) (1 (2, 3, 4) (5, 6, 7))

(d) (1 (2, 3 null) (9, 5))

Q.23 B^+ – trees are preferred to binary trees in databases because

(a) Disk capacities are greater than memory capacities

(b) Disk access is much slower than memory access

(c) Disk data transfer rates are much less than memory data transfer rates

(d) Disks are more reliable than memory

Q.24 Which of the following lists of numbers produce the most balanced tree, when inserted, in the order given, into a binary search tree?

(a) 2, 5, 1, 4, 0, 3 (b) 5, 1, 2, 6, 3, 4

(c) 2, 4, 7, 5, 8, 11 (d) 6, 4, 1, 8, 10, 5

Q.25 In a sequential representation of a binary tree in memory, let TREE be an array which is linear in nature. If any Node N occupies the position Tree [K] then its left child is stored in _____ and its right child is stored in _____.

(a) TREE [K + 1], TREE [K + 2]

(b) TREE [2 * K], TREE [2 * K + 1]

(c) TREE [2 * K + 1], TREE [2 * K + 2]

(d) TREE [K + 2], TREE [K + 1]

Q.26 Let T_n be a comlete binary tree which has 1000000 nodes. Then its depth

(a) 100 (b) 99

(c) 21 (d) 101

Q.27 The number 1, 2, ... n are inserted in binary search tree in some order. In the resulting tree, the right subtree of the root contains p nodes. The first number to be inserted in the tree must be

(a) p (b) p + 1

(c) n – p (d) n – p + 1

Q.28 In quick-sort, for sorting n elements, the $(n/4)^{th}$ smallest element is selected as pivot using an O(n) time algorithm. What is the worst case time complexity of the quick sorts?

(a) $\theta(n)$ (b) $\theta(n \log n)$

(c) $\theta(n^2)$ (d) $\theta(n^2 \log n)$

Q.29 Consider the table of 15 items as given below:

AL, EX, FN, FU, IF, IW, LE, LO, NC, OP, OR, RD, RN, TE, TI.

By using binary search method, after how many passes will the search for the item IF be terminated?

(a) 3-pass (b) 4-pass

(c) 5-pass (d) 6-pass

Q.30 The height of a tree is defined as the number of edges on the longest path in the tree. The function shown in the pseudocode below is invoked as the height (root) to compute the height of a binary tree rooted at the tree pointer root.

```
int height(treeptr n)
{
   if(n==NULL)return -1;
   if(n→left==NULL)
   if(n→right==NULL) return 0;
   else return[B1]; //Box1
   else
   {
      h1=height(n→left);
      if(n→right==NULL)
      return(1+h1);
```

```
        if
        {
            h2=height(n→right);
            return [B2]; //Box2
        }
    }
}
```

The appropriate expressions for the two boxes B1 and B2 are

(a) B1: (1 + height (n → right))
B2: (1 + max(h1, h2))

(b) B1: (height(n → right))
B2: (1 + max(h1, h2))

(c) B1: height(n → right)
B2: max(h1, h2)

(d) B1: (1 + height(n → right))
B2: max(h1, h2))

Answer Key:

1. (a)	**2.** (c)	**3.** (c)	**4.** (d)	**5.** (d)
6. (a)	**7.** (b)	**8.** (a)	**9.** (b)	**10.** (c)
11. (c)	**12.** (a)	**13.** (b)	**14.** (c)	**15.** (a)
16. (a)	**17.** (b)	**18.** (c)	**19.** (b)	**20.** (b)
21. (b)	**22.** (c)	**23.** (b)	**24.** (d)	**25.** (b)
26. (c)	**27.** (d)	**28.** (b)	**29.** (b)	**30.** (a)

■■■■

CHAPTER 07

Hashing Techniques

7.1 Introduction

Hash table is a generation of array. With an array, we stroke the element whose key is k at a position k of the array. That means, given a key k, we find the element whose key is k by just looking in the k^{th} position of the array. This is called direct addressing.

Direct addressing is applicable when we can afford to allocate array with the positions for every possible key. Suppose if we do not have enough space to allocate a location for each possible key then we need a mechanism to handle this case. Other way of defining the scenario is, if we have less locations and more possible keys then simple array implementation is not enough.

In these cases one option is to use hash tables. Hash table or hash map is a data structure that stores the keys and their associated values. Hash table uses a hash function to map keys to their associated values. General convention is that we use a hash table when the number of keys actually stored is small relative to the number of possible keys.

7.2 Hash Function

The hash function is used to transform the key into the index. Ideally, the hash function should map each possible key to a unique slot index, but it's difficult to achieve in practice.

How to Choose Hash Function?

The basic problems associated with the creation of hash tables are:

- An efficient hash function should be designed so that it distributes the index values of inserted objects uniformly across the table.
- An efficient collision resolution algorithm should be designed so that it computes an alternative index for a key whose hash index corresponds to a function to a location previously inserted in the hash table.
- We must choose a hash function which can be calculated quickly, returns values within the range of locations in our table, and minimize collisions.

Characteristics of Good Hash Functions

- Minimize collision
- Be easy and quick to compute
- Distribute key values evenly in the hash table
- Use all the information provided in the key
- Have a high load factor for a given set of keys

Load Factor

The load factor of a non empty hash table is the number of items stored in the table divided by the size of the table. This is the decision parameter used when we want to rehash or expand the existing hash table entries. This also helps us in determining the efficiency of the hashing function. That means, it tells whether the hash function is distributing the key uniformly or not.

$$\text{Load Factor} = \frac{\text{Number of elements in hash table}}{\text{Hash Table Size}}$$

7.3 Collisions

Hash functions are used to map each key to different address space but practically it is not possible to create such a hash function and the problem called collision. Collision is the condition where two records are stored in the same location.

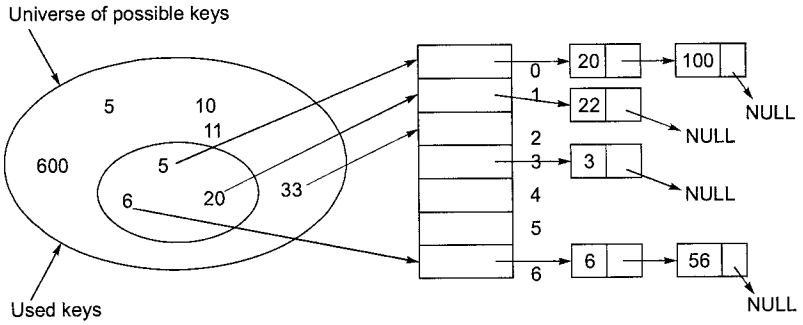
7.4 Collision Resolution Techniques

The process of finding an alternate location is called collision resolution. Even though hash tables are having collision problem, they are more efficient in many cases comparative to all other data structures like search trees. There are a number of collision resolution techniques, and the most popular are open addressing and chaining.

- **Direct Chaining:** An array of linked list application
  - (*i*) Separate chaining
- **Open Addressing:** Array based implementation
  - (*i*) Linear probing (linear search)
  - (*ii*) Quadratic probing (non linear search)
  - (*iii*) Double hashing (use two hash functions)

7.4.1 Separate Chaining

Collision resolution by chaining combines linked representation with hash table. When two or more records hash to the same location, these records are constituted into a singly-linked list called a chain.



7.4.2 Open Addressing

In open addressing all keys will be stored in the hash table itself. This approach is also known as closed hashing. This procedure is based on probing. A collision is resolved by probing.

- **Linear Probing:** Interval between probes is fixed at 1. In linear probing, we search the hash table sequentially starting from the original hash location. If a location is occupied, we check the next location. We wrap around from the last table location to the first table location if necessary. The function for the rehashing is the following:

  rehash (key) = $(n + 1)\%$ table size

  One of the problems with linear probing is that table items tend to cluster together in the hash table. This means that, table contains groups of consecutively occupied locations and called as clustering. Clusters can get close to one another, and merge into a larger cluster. Thus, the one part of the table might be quite dense, even though another part has relatively few items. Clustering causes long probe searches and therefore decrease the overall efficiency.

  The next location to be probed is determined by the step size, where other step-sizes (than one) are possible. The step-size should be relatively prime to the table size *i.e.* their greatest common divisor should be equal to 1. If we chose the table size to be a prime number, then any step-size is relatively prime to the table size. Clustering cannot be a avoided by larger step-sizes.

  Here, $\Theta(m)$ probe sequences are used.

- **Quadratic Probing:** Interval between probes increases proportional to the hash value (the interval thus increasing linearly and the indices are described by a quadratic function). Clustering problem can be eliminated if we used quadratic probing method.

  In quadratic probing, we start from the original hash location i. If a location is occupied, we check the locations $i + 1^2$, $i + 2^2$, $i + 3^2$, $i + 4^2$ ... We wrap around from the last table location to the first table location if necessary. The function for the rehashing is the following:

  rehash (key) = $(n + k^2)\%$ table size

  Example: Let us assume that the table size is 11 (0..10)

  Hash Function: h(key) = key mod 11

  Insert keys:

  31 mode 11 = 9
  19 mode 11 = 8
  2 mode 11 = 2
  13 mode 11 = 2 → 2 + 12 = 3
  25 mode 11 = 3 → 3 + 12 = 4
  24 mode 11 = 2 → 2 + 12, 2 + 22 = 6
  21 mode 11 = 10
  9 mode 11 = 9 → 9 + 12, 9 + 2^2 mod 11, 9 + 3^2 mod 11 = 7

0	
1	
2	2
3	13
4	25
5	5
6	24
7	9
8	19
9	31
10	21

  Even though clustering is avoided by quadratic probing. Still there are chances of clustering. Clustering is caused by multiple search keys mapped to the same hash key. Thus, the probing sequence for such search keys is prolonged by repeated conflicts along the probing sequence. Both linear and quadratic probing use a probing sequence that is independent of the search key.

  Here, $\Theta(m)$ probe sequences are used.

- **Double Hashing:** Interval between probes is computed by another hash function. Double hashing reduces clustering in a better way. The increments for the probing sequence are computed by using a second hash function.

The second hash function $h2$ should be:

$$h2(\text{key}) \neq 0 \text{ and } h2 \neq h1$$

We first probe the location $h2$ (key). If the location is occupied, we probe the location $h1$(key) + $h2$(key), $h1$(key) + 2 * $h2$(key), ...

Example:

Table size is 11 (0.10)

Hash Function: Assume $h2(\text{key}) = \text{key mod } 11$ and

$h2(\text{key}) = 7 - (\text{key mod } 7)$

Insert Keys:

58 mode 11 = 3

14 mode 11 = 3 → 3 + 7 = 10

91 mode 11 = 3 → 3 + 7, 3 + 2 * 7 mod 11 = 6

24 mode 11 = 3 → 3 + 3, 3 + 2 * 3 = 9

0	
1	
2	
3	58
4	25
5	
6	91
7	
8	
9	25
10	14

(a) Here, $\Theta(m)^2$ probe sequences are used.

(b) In an open addressing scheme, the expected number of probes in an unsuccessful search is at most $1 / (1 - \alpha)$, assuming uniform hashing.

(c) Expected number of probes in a successful search is at most $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$, assuming uniform hashing and that each key is equally likely to be searched.

7.5 Hashing Function

A hash function f, transforms an identifier, x, into a bucket address in the hash table. We want to hash function that is easy to compute and that minimizes the number of collisions. We know that identifiers, whether they represent variable names in a program, word in a dictionary, or names in a telephone book, cluster around certain letters of the alphabet. To avoid collisions, the hash function should depend on all the characters in an identifier. It also should be unbiased. That is, if we randomly choose an identifier, x, from the identifier space (the universe of all possible identifiers), the probability that $f(x) = i$ is $1/\alpha$ for all buckets i. This means that a random x has an equal property a uniform hash function.

There are several types of uniform hash functions, and we shall describe four of them. We assume that the identifiers have been suitably transformed into a numerical equivalent.

7.5.1 Mid-square

The middle of square hash function is frequently used in symbol table application. We compute the function f_m by squaring the identifier and then using an appropriate number of bits from the middle of the square to obtain the bucket address. Since the middle bits of the square usually depend upon all the characters in an identifier, there is a high probability that different identifiers will produce different hash addresses, even when some of the characters are the same. The number of bits used to obtain the bucket address depends on the table size. If we use r bits, the range of the values is 2^r. Therefore, the size of the hash table should be a power of 2 when we use this scheme.

7.5.2 Division

This hash function is using the modulus (%) operator. We divide the identifier x by some number M and use the remainder as the hash address of x. The hash function is: $f_D(x) = x\%$ M. This gives bucket address that range from 0 to M – 1, where M = the table size. The choice of M is critical. In the division function, if M is a power

of 2, then $f_D(x)$ depends only on the least significant bits of x. Such a choice for M results in a biased use of the hash table when several of the identifiers in use have the same suffix. If M is divisible by 2, then odd keys are mapped to odd buckets, and even keys are mapped to even buckets. Hence, an even M results in a biased use of the table when a majority of identifiers are even or when majorities are odd.

7.5.3 Folding

In this method, we partition the identifier x into several parts. All parts, except for the last one have the same length. We then add the parts together to obtain the hash address for x. There are two ways of carrying out this addition. In the first method, we shift all parts except for the last one, so that the least significant bit of each part lines up with the corresponding bit of the last part. We then add the parts together to obtain $f(x)$. This method is known as shift folding. The second method, know as folding at the boundaries, reverses every other partition before adding.

7.5.4 Digit Analysis

The last method we will examine, digit analysis, is used with static files. A static file is one in which all the identifiers are known in advance. Using this method, we first transform the identifiers into numbers using some radix, r. We then examine the digits of each identifier, deleting those digits that have the most skewed distributions. We continue deleting digits until the number of remaining digits is small enough to give an address in the range of the hash table. The digits used to calculate the hash address must be the same for all identifiers and must not have abnormally high peaks or valleys (the standard deviation must be small).

7.6 Comparison of Collision Resolution Techniques

Comparisons: Linear Probing Vs. Double Hashing

The choice between linear probing and double hashing depends on the cost of computing the hash function and on the load factor [number of elements per slot] of the table. Both use few probes but double hashing take more time because it hashes to compare two hash functions for long keys.

Comparisons: Open Addressing vs. Separate Chaining

It's somewhat complicated because we have to account the memory usage. Separate chaining uses extra memory for links. Open addressing needs extra memory implicitly within the table to terminate probe sequence. Open addressed hash tables cannot be used if the data does not have unique keys. An alternative is use separate chained hash tables.

Comparisons: Open Addressing methods

Linear Probing	Quadratic Probing	Double hashing
Fastest among three	Easiest to implement and deploy	Makes more efficient use of memory
Use few probes	Uses extra memory for links and it does not probe all locations in the table	Use few probes but take more time
A problem occurs known as primary clustering	A problem occurs known as secondary clustering	More complicated to implement
Interval between probes is fixed-often at 1.	Interval between probes increases proportional to the hash value	Interval between probes is computed by another hash function

7.7 Various Hash Function

1. **Division Method:** h(k) = k mod m.
   - Hash k into a table with m slots using the slot given by the remainder of k divided by m.
   - We pick up a table size m = prime number, not too close to a power of 2(or 10).
2. **Multiplication Method:**
   - $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ where A is a constant $0 < A < 1$ i.e. calculate fractional part of KA and then multiply it with m, the table size.
   - $m = 2^p$
3. **Universal Hash function:** We pick a random hash function, when the algorithm begins, not on every insert.
4. **Perfect Hashing:** It the technique in which the best case number of memory accesses required to perform a search is O(1).

Summary

- Hashing is a searching technique in which the searching is constant O(1). It is based on indexing mechanism. It uses a function called HASH FUNCTION.
- **Components in hashing:** Hash table, Hash functions, Collisions and Collision Resolution Techniques.
- Load factor (α) = $\frac{\text{number of elements in hash table}}{\text{Hash table size}}$
- **Direct address table:** In direct address table key is the address without any manipulation. Even though number of keys are very less but one of key may contain 64 bits then size of hash table should be $2^{64} - 1$. The range of keys determines the size of the direct address table.
- **Types of Hash Functions:**
  (*i*) Division modulo method:
  $H(X) = X\%\ m$, where m = Hash table size and X = Key.
  (*ii*) Digit extraction method (Truncation method)
  (*iii*) Mid Square method
  (*iv*) Folding Method: Fold shifting method and Fold boundary
- **Collision Resolution Techniques:**
  (*i*) Open addressing.
  (*ii*) Chaining.
- **Collision Resolution Techniques:** Collision will take place if a new element is getting mapped where an element is already present.
- **Open Addressing:**
  (*i*) Linear Probing: If there is a collision at location '*L*' then look for empty location successively.
  Disadvantage:
  Primary clustering: The trend is for long sequence of preoccupied positions still become longer, primarily at one place.
  (*ii*) Quadratic Probing: Hash function = $h+i^2$ [Quadratic in nature]
  Disadvantage: Secondary clustering.
  (a) The maximum number of comparisons performed for successful search = size of largest cluster + 1.

(b) In Quadratic probing, all buckets are not compared because of quadratic nature.

(*iii*) Double Hashing/Rehashing: '*C*' is changed in random probing in order to get new sequence.

Advantage: Clustering is reduced.

Disadvantage: Clustering is not avoided.

- In open addressing scheme $0 \le \alpha \le 1$.
- Average number of probes (comparisons) in successful search

Successful Seach	Unsuccessful Seach
Linear probing: $\frac{1}{2}+\frac{1}{2(1-\alpha)}$	$\frac{1}{2}+\frac{1}{2(1-\alpha)^2}$
Chaining: $(\alpha+1)/2$	$1+\alpha/2$

Student's Assignments

Q.1 A hash table is used when

(a) Number of keys to be stored is greater than the number of records to be stored

(b) A large number of keys are duplicate keys

(c) Number of keys actually stored it any point of time is small as compared to the number of possible keys

(d) None of these

Q.2 An hash table with chaining as a collision resolution technique degenerates to a

(a) Queue (b) Linked list

(c) Three (d) Array

Q.3 Let, k = Key value, m = some multiple of 2, A = some constant fraction, 0 < A < 1.
Which of the following created by the multiplication method?

(a) [km mod A] (b) $\lfloor m\,(kA - \lfloor kA \rfloor) \rfloor$

(c) $\lfloor kA \bmod m \rfloor$ (d) None of these

Q.4 Let H be a finite collection of hash functions that map a universe U of keys into {0, 1, ..., m – 1}. H is said to be universal if for each pair of distinct keys, k and I ∈ U, the number of hash functions h ∈ H for which h(k) = n(I) is at most.

(a) $|H|/m^2$ (b) $\frac{1}{m^2 \log m}$

(c) $|H|m^2$ (d) $|H|/m$

Common Data Questions (5 and 6):

Insert the characters at the string K, R, P, C, S, N, Y, T, J, M into a hash table of size 10 use has function

H(x) = ord (x) – ord ('a') + 1 mod 10

and linear probing to resolve collisions.

Q.5 Which insertions cause collisions?

(a) S, N (b) T, J

(c) J, M (d) None of these

Q.6 Display the final hash table?

(a)

0	1	2	3	4	5	6	7	8	9
T	K	J	C	R	S	P	M	N	Y

(b)

0	1	2	3	4	5	6	7	8	9
T	K	J	C	N	Y	P	M	R	S

(c)

0	1	2	3	4	5	6	7	8	9
T	K	J	C	Y	N	P	M	R	S

(d)

0	1	2	3	4	5	6	7	8	9
T	K	J	C	N	Y	P	M	S	R

Common Data Questions (7 and 8):

Consider a hash table with n buckets, where external (overflow) chaining is used to resolve collision. The hash function is such that the probability that a key values is hashed to a particular bucket is (1/n). The hash table is initially empty and k distinct values are inserted in the table.

Q.7 What is the probability that bucket number 1 is empty after the k^{th} insertion?

(a) n (b) $(n-1)^k$

(c) $\left(\frac{n-1}{n}\right)^k$ (d) $\left(\frac{n-1}{k}\right)^n$

Q.8 What is the probability that no collision has occurred in any of the k insertions?

(a) $\frac{n(n-1)(n-2)\ldots(n-k+1)}{n^k}$

(b) $\frac{n(n-1)(n-2)(n-k-1)}{n^k}$

(c) Both (a) and (b)

(d) None of these

Q.9 A hash table of length 10 uses open addressing with hash function h(k) = k mod 10 and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Given a hash table with size = 11, hash function h (key) = key % 11, where collisions are resolved using linear probing; following operations are performed sequentially:

Insert (17)

Insert (37)

Insert (59)

Insert (70)

Insert (59)

Insert (16)

What is the number of collisions while doing insert operation on the hash table?

(a) 3 (b) 4

(c) 5 (d) 6

Q.10 Consider a hash table with 9 slots. The hash function is $h(k) = k$ mod 9. The collisions are resolved by chaining. The following 9 keys are inserted in the order: 5, 28, 19, 15, 20, 33, 12, 17, 10. The maximum, minimum, and average chain lengths in the hash table, respectively, are

(a) 3, 0, and 1 (b) 3, 3, and 3

(c) 4, 0, and 1 (d) 3, 0, and 2

Q.11 An advantage of chaining over open addressing scheme is

(a) Worst case complexity of search operation is less

(b) Space used is less

(c) Deletion easier

(d) None of these

Q.12 Search tables used by compiler for efficient searching generally use

(a) hash table

(b) list of records

(c) binary search table

(d) binary search tree

Answer Key:

1. (c)	**2.** (b)	**3.** (b)	**4.** (d)	**5.** (c)
6. (b)	**7.** (c)	**8.** (a)	**9.** (b)	**10.** (a)
11. (c)	**12.** (a)			

■■■■