URBAN EDGE

C Programming Notes by Code With Harry

What is Programming?

Computer Programming is a medium for us to communicate with computers. Just like we use 'Hindi' or 'English' to communicate with each other, programming is a way for us to deliver our instructions to the computer.

What is C?

C is a programming language.
C is one of the oldest and finest programming languages.
C was developed by Dennis Ritchie at AT&T's Bell Labs, USA in 1972.

Uses of C

C Language is used to program a wide variety of systems. Some of the uses of C are as follows:

1) Major parts of Windows, Linux and other operating systems are written in C.

2) C is used to write driver programs for devices like Tablets, printers etc.

3) C language is used to program embedded systems where programs need to run faster in limited memory (Microwave, Cameras etc.)

4) C is used to develop games, an area where latency is very important. ie. Computer has to react quickly on user input.

Scanned with CamScanner

URBAN EDGE

Chapter 1: Variables, Constants & Keywords

Variables

A variable is a container which stores a 'value'. In Kitchen, we have containers storing Rice, Dal, Sugar etc. Similar to that variables in C stores value of a constant. Example:

```
a = 3;      // a is assigned "3"
b = 4.7;    // b is assigned "4.7"
c = 'A';    // c is assigned 'A'
```

Rules for naming variables in C

1) First character must be an alphabet or underscore (_)
2) No commas, blanks allowed.
3) No special symbol other than (_) allowed.
4) Variable names are case sensitive.

We must create meaningful variable names in our programs. This enhances readability of our programs.

Constants

An entity whose value doesn't change is called as a constant.

A variable is an entity whose value can be changed.

Scanned with CamScanner

URBAN EDGE

Types of constants

Primarily, there are three types of constants:

1) Integer Constant → -1, 6, 7, 9
2) Real Constant → -322.1, 2.5, 7.0
3) Character Constant → 'a', '$', '@' (Must be enclosed within single inverted commas)

Keywords

These are reserved words, whose meaning is already known to the compiler. There are 32 keywords available in C.

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

Our First C Program

```
#include <stdio.h>

int main() {
printf("Hello, I am learning C with Harry");
return 0;
}
```

File: first.C

Scanned with CamScanner

URBAN EDGE ___/___/___

Basic Structure of a C Program

All C programs have to follow a basic structure.
A C program starts with a main function and executes instructions present inside it.
Each instruction is terminated with a semicolon (;)

There are some rules which are applicable to all the C programs :

1. Every program's execution starts from main() function.
2. All the statements are terminated with a semicolon.
3. Instructions are case-sensitive
4. Instructions are executed in the same order in which they are written.

Comments

Comments are used to clarify something about the program in plain language. It is a way for us to add notes to our program. There are two types of comments in C.

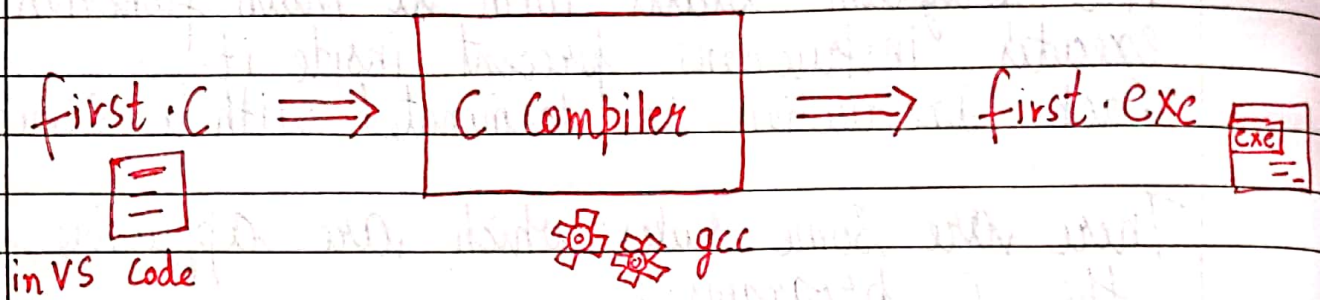
1. Single line comment : // This is a comment
2. Multi-line comment : /* This is a
multi line comment */

Comments in a C program are not executed and are ignored.

Scanned with CamScanner

URBAN EDGE

Compilation and Execution



A compiler is a computer program which converts a C program into machine language so that it can be easily understood by the computer.

A C program is written in plain text. This plain text is combination of Instructions in a particular sequence. The compiler performs some basic checks and finally converts the program into an executable.

Library Functions

C language has a lot of valuable library functions which is used to carry out certain tasks. for instance printf function is used to print values on the screen

```
printf("This is %d", i);
```

%d for integers
%f for real values
%c for characters

Scanned with CamScanner

URBAN EDGE

Types of variables

1. Integer variables → int a = 3;
2. Real variables → int a = 7.7; float a = 7.7; ← Wrong as 7.7 is real
3. Character variables → char a = 'B';

Receiving input from the User

In order to take input from the user and assign it to a variable, we use Scanf function

Syntax for using scanf :

```
Scanf ("%d", &i);
```

↳ This & is important!

& is the "address of" operator and it means that the supplied value should be copied to the address which is indicated by variable i.

Scanned with CamScanner

URBAN EDGE

Chapter 1 - Practice Set

Q1 Write a C program to calculate area of a rectangle:

(a) Using hard coded inputs
(b) Using inputs supplied by the User

Q2 Calculate the area of a circle and modify the same program to calculate the volume of a cylinder given its radius and height.

Q3 Write a program to convert Celcius (Centigrade degrees temperature to Farenheit)

Q4 Write a program to calculate simple interest for a set of values representing principal, no of years and rate of interest.

Scanned with CamScanner

URBAN EDGE

Chapter 2: Instructions and Operators

A C program is a set of Instructions. Just like a recepie - which contains Instructions to prepare a particular Dish.

Types of Instructions

1. Type declaration Instruction
2. Arithmetic Instruction
3. Control Instruction

Type declaration Instruction

```
int a;
float b;
```

Other Variations:

```
int i=10; int j=i; int a=2
int j1= a + j - i;
```

```
float b = a+3; float a=1.1
```

=> ERROR! as we are trying to use a before defining it.

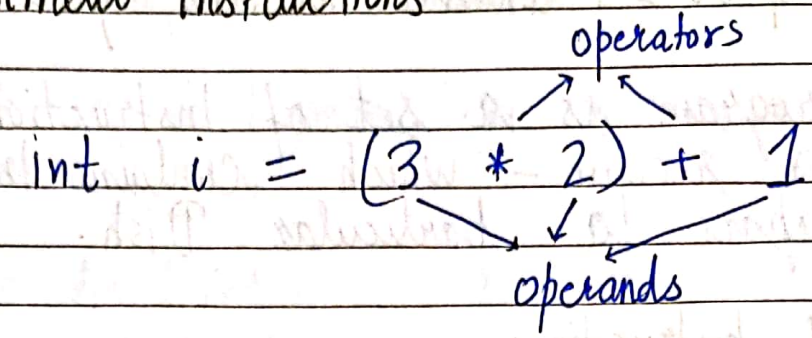
```
int a, b, c, d;
a = b = c = d = 30;
```

=> Value of a, b, c & d will be 30 each.

Scanned with CamScanner

URBAN EDGE

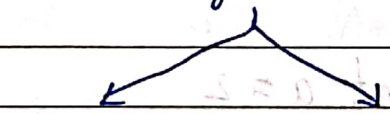
Arithmetic Instructions



Operands can be int / float etc.
+ - * / are arithmetic operators

```
int b = 2, c = 3;
int z; z = b * c;   ✓ legal
int z;  b * c = z;  ✗ Illegal (Not allowed)
```

% → Modular division operator
% → Returns the remainder
% → Cannot be applied on float
% → Sign is same as of numerator (-5 % 2 = -1)



Note :-

1. No operator is assumed to be present

```
int i = a b  → Invalid
int i = a * b → valid
```

2. There is no operator to perform exponentiation in C. However we can use pow(x, y) from <math.h> (More later)

Scanned with CamScanner

URBAN EDGE

Type Conversion

An Arithmetic operation between

Int and Int → Int
Int and float → float
float and float → float

$5/2 \rightarrow 2$ $\quad$ $5.0/2 \rightarrow 2.5$

$2/5 \rightarrow 0$ $\quad$ $2.0/5 \rightarrow 0.4$

} Important!!

Note :-

`int a = 3.5;` In this case 3.5 (float) will be demoted to 3 (int) because a is not able to store floats.

`float a = 8;` a will store 8.0
8 → 8.0 (promotion to float)

Quick Quiz :

Q) `int k = 3.0/9` Value of k? and why?

S) $3.0/9 = 0.333..$ but since k is an int, it cannot store floats & value 0.33 is demoted to 0.

Scanned with CamScanner

URBAN EDGE

Operator precedence In C

$3 * x - 8 y$ is $(3x) - (8y)$ or $3(x - 8y)$?

In C language simple mathematical rules like BODMAS, no longer applies.

The answer to the above question is provided by operator precedence & associativity.

Operator precedence :- The following table lists the operator priority in C

Priority	Operators
1st	* / %
2nd	+ -
3rd	=

Operators of higher priority are evaluated first in the absence of parenthesis.

Operator Associativity :- When operators of equal priority are present in an expression, the tie is taken care of by associativity.

$x * y / z \Rightarrow (x * y) / z$

$x / y * z \Rightarrow (x / y) * z$

*, / follows Left to right associativity

Scanned with CamScanner

URBAN EDGE

Control Instructions

Determines the flow of control in a program

Four types of control Instructions in C are:

1. Sequence Control Instruction
2. Decision Control Instruction
3. Loop Control Instruction
4. Case Control Instruction.

Scanned with CamScanner

URBAN EDGE

Chapter 2 - Practice Set

Q1 Which of the following is Invalid in C?

(i) int a; b=a;

(ii) int V = 3^3;

(iii) char dt = '21 Dec 2020';

Q2 What data type will 3.0/8 - 2 return?

Q3 Write a program to check whether a number is divisible by 97 or not.

Q4 Explain step by step evaluation of 3*x/y-z+k where x = 2 y = 3 z = 3 k = 1

Q5 3.0 + 1 will be :

(a) Integer

(b) Floating point number

(c) Character

Scanned with CamScanner

URBAN EDGE

Chapter 3 - Conditional Instructions

Sometimes we want to watch comedy videos on YouTube if the day is Sunday.
Sometimes we order junk food if it is our friend's birthday in the hostel.
You might want to buy an Umbrella if its raining and you have the money.
You order the meal if dal or your favorite bhindi is listed on the menu.

All these are decisions which depends on a condition being met.
In C language too, we must be able to execute instructions on a condition(s) being met.

Decision Making Instructions in C
→ If - else Statement
→ Switch Statement

If - else Statement
The Syntax of an If - else Statement in C looks like:

```
if (condition to be checked) {
    Statements - if - condition - true;
  }
else {
    Statements - if - condition - false;
    }
```

Scanned with CamScanner

URBAN EDGE

Code example:

```
int a = 23;

if (a > 18) {
    printf("You can drive \n");
}
```

Note that else block is not necessary but optional.

Relational Operators in C

Relational operators are used to evaluate conditions (true or false) inside the if statements. Some examples of relational operators are :

`==` , `>=` , `>` , `<` , `<=` , `!=`

- `==` : equals
- `>=` : greater than or equal to
- `!=` : not equal to

Important note : '=' is used for assignment where as '==' is used for equality check.

The condition can be any valid expression. In C a non-zero value is considered to be true.

Logical Operators

&&, || and ! are three logical operators in C. These are read as "AND", "OR" and "NOT". They are used to provide logic to our C programs.

Scanned with CamScanner

URBAN EDGE

Usage of Logical Operators:

(i) && → AND → is true when both the conditions are true

"1 and 0" is evaluated as false.
"0 and 0" is evaluated as false.
"1 and 1" is evaluated as true.

(ii) || → OR → is true when at least one of the conditions is true. (1 or 0 → 1)(1 or 1 → 1)

(iii) ! → returns true if given false and false if given true

!(3==3) → evaluates to false
!(3>30) → evaluates to true.

As the number of conditions increases, the level of indentation increases. This reduces readability. Logical operators come to rescue in such cases.

else if clause

Instead of using multiple if statements, we can also use else if along with if thus forming an if-else if-else ladder.

```
if {
// Statements;
}
else if {...
}
else {...
}
```

Scanned with CamScanner

URBAN EDGE

Using if - else if - else reduces indents
The last "else" is optional
Also there can be any number of "else if"

Last else is executed only if all conditions fail.

Operator precedence

Priority	Operator
1st	!
2nd	*, /, %
3rd	+, -
4th	<, >, <=, >=
5th	==, !=
6th	&&
7th	||
8th	=

Conditional Operators
A short hand "if - else" can be written using the conditional or ternary operators

Condition ? expression-if-true : expression-if-false

Ternary operators

Scanned with CamScanner

URBAN EDGE

Switch Case Control Instruction

Switch-Case is used when we have to make a choice between number of alternatives for a given variable.

```
Switch (integer-expression)
{
  Case C1:
      Code;

  Case C2:
      Code;

  Case C3:
      Code;

  default:
      Code;
}
```

C_1, C_2 & C_3 → Constants

Code → Any valid C code.

The value of integer-expression is matched against C_1, C_2, C_3 .... If it matches any of these Cases, that case along with all subsequent "Case" and "default" Statements are executed.

* Quick Quiz: Wrik a program to find grade of a student given his marks based on below:

→ 90 - 100 → A
→ 80 - 90 → B
→ 70 - 80 → C
→ 60 - 70 → D
→ < 70 → F.

Scanned with CamScanner

URBAN EDGE

Important Notes

1. We can use switch-case statements even by writing cases in any order of our choice. (not necessarily ascending)

2. char values are allowed as they can be easily evaluated to an integer

3. A switch can occur within another but in practice this is rarely done.

Scanned with CamScanner

URBAN EDGE

Chapter 3 - Practice Set

1. What will be the output of this program

```
int a = 10;
if (a = 11)
    printf ("I am 11");
else
    printf ("I am not 11");
```

2. Write a program to find out whether a student is pass or fail; if it requires total 40% and at least 33% in each subject to pass. Assume 3 subjects and take marks as an input from the user.

3. Calculate income tax paid by an employee to the government as per the slabs mentioned below:

Income Slab	Tax
2.5L - 5.0L	5%
5.0L - 10.0L	20%
Above 10.0L	30%

Note that there is no tax below 2.5L. Take income amount as an input from the user.

4. Write a program to find whether a year entered by the user is a leap year or not. Take year as an input from the user.

Scanned with CamScanner

URBAN EDGE

5 Write a program to determine whether a character entered by the user is lowercase or not.

6 Write a program to find greatest of four numbers entered by the user.

Scanned with CamScanner

URBAN EDGE

Chapter 4 - Loop Control Instruction

Why Loops

Sometimes we want our programs to execute few set of instructions over and over again. for ex: printing 1 to 100, first 100 even numbers etc.

Hence Loops make it easy for a programmer to tell computer that a given set of instructions must be executed repeatedly.

Types of Loops

Primarily, there are three types of loops in C Language:

1. While loop
2. do - while loop
3. for loop

We will look into these one by one

While loop

```
While (condition is true) {

    // Code
    // Code

    }
```

⇒ The block keeps executing as long as the condition is true.

Scanned with CamScanner

An example:

```
int i = 0

While (i < 10) {
    printf (" The value of i is %d", i); i++;
}
```

Note: If the condition never becomes false, the while loop keeps getting executed. Such a loop is known as an infinite loop.

Quick Quiz: Write a program to print natural numbers from 10 to 20 when initial loop counter i is initialized to 0.

The loop counter need not be int, it can be float as well.

Increment and decrement operators

i++ → i is increased by 1
i-- → i is decreased by 1

```
printf ("--i = %d", --i);
```

This first decrements i and then prints it

```
printf (" i-- = %d", i--);
```

This first prints i and then decrements it

Scanned with CamScanner

* +++ operator does not exist ⇒ Important
* += is compound assignment operator just like -=, *=, /= & %= ⇒ Also Important

do-While Loop.
The syntax of do-while loop looks like this:

```
do {
  // Code;
  // Code;
} while (condition)
```

do-while loop works very similar to while loop.
While → checks the condition & then executes the code
do-while → Executes the code & then checks the condition

do-while loop = While loop which executes at least once.

↳ Quick Quiz: Write a program to print first n natural numbers using do-while loop.

Input : 4

Output : 1
2
3
4

Scanned with CamScanner

URBAN EDGE

for Loop

The syntax of for loop looks like this:

```
for( initialize ; test ; increment
                         or decrement)
  {
   // Code;
   // Code;
   // Code;
  }
```

Initialize → Setting a loop Counter to an initial value
Test → Checking a condition
Increment → Updating the loop Counter

An example:

```
for( i=0 ; i<3 ; i++) {
    printf( "%d", i);
    printf( "\n");
  }
```

Output:

```
0
1
2
```

Quick Quiz: Write a program to print first n natural numbers using for loop

Scanned with CamScanner

URBAN EDGE

A Case of Decrementing for loop

```
for (i=5; i; i--)
    printf ("%d\n", i);
```

This for loop will keep on running until i becomes 0.
The loop runs in following steps:

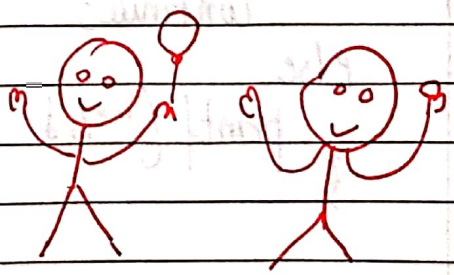
1. i is initialized to 5
2. The condition "i" (0 or none) is tested
3. The code is executed
4. i is decremented
5. Condition i is checked & code is executed if its not 0.
6. & so on until i is non 0

Quick Quiz: Write a program to print n natural numbers in reverse order.

The break Statement in C

The break statement is used to exit the loop irrespective of whether the condition is true or false.
Whenever a "break" is encountered inside the loop, the control is sent outside the loop

Let us see this with the help of an Example



Scanned with CamScanner

```
for (i=0; i<1000; i++) {
  printf ("%d\n", i);
  if (i==5) {
     break;
    }
}
```

output ⇒ 0
1
2
3
4
5
and not 0 to 100 ☺

The Continue statement in C
The Continue statement is used to immideately move to the next iteration of the loop.

The control is taken to the next iteration thus skipping everything below "Continue" inside the loop for that iteration

Let us look at an example

```
int skip = 5;
int i=0;

While (i<10) {
  if (i != skip)
     Continue;
  else
     printf ("%d", i);
  }
```

output ⇒ 5
and not 0 ... 9

Scanned with CamScanner

URBAN EDGE

Notes:

1. Sometimes, the name of the variable might not indicate the behaviour of the program.
2. break statement completely exits the loop.
3. Continue statement skips the particular iteration of the loop.

Scanned with CamScanner

URBAN EDGE

Chapter 4 - Practice Set

1. Write a program to print multiplication table of a given number n.

2. Write a program to print multiplication table of 10 in reversed order

3. A do while loop is executed:
   1. at least once
   2. at least twice
   3. at most once

4. What can be done using one type of loop can also be done using the other two types of loops - True or False?

5. Write a program to sum fist ten natural numbers using while loop.

6. Write a program to implement program 5 using for and do-while loop.

7. Write a program to calculate the sum of the numbers occuring in the multiplication table of 8. (Consider 8×1 to 8×10).

8. Write a program to calculate the factorial of a given number using a for loop.

Scanned with CamScanner

URBAN EDGE

9 Repeat 8 using while loops

10 Write a program to check whether a given number is prime or not using loops.

11 Implement 10 using other types of loops.

Scanned with CamScanner

URBAN EDGE

Project 1: Number guessing Game

Problem: This is going to be fun! We will write a program that generates a random number and asks the player to guess it. If the player's guess is higher than the actual number, the program displays "Lower number please". Similarly if the user's guess is too low, the program prints "Higher number please"
When the user guesses the correct number, the program displays the number of guesses the player used to arrive at the number.

Hint: Use loops
Use a random number generator

Scanned with CamScanner

URBAN EDGE

Chapter 5 - Functions and Recursion

Sometimes our program gets bigger in size and its not possible for a programmer to track which piece of code is doing what.
function is a way to break our code into chunks so that it is possible for a programmer to reuse them.

What is a Function ?

A function is a block of code which performs a particular task.
A function can be reused by the programmer in a given program any number of times.

Example and Syntax of a function

```
# include <stdio.h>

Void display();   => Function prototype

int main() {
  int a;
  display();      => Function call
  return 0;
  }

Void display() {          => Function definition
  printf("Hi I am display");
}
```

Scanned with CamScanner

Function prototype
Function prototype is a way to tell the compiler about the function we are going to define in the program.
Here void indicates that the function returns nothing.

Function call
Function call is a way to tell the compiler to execute the function body at the time the call is made.
Note that the program execution starts from the main function in the sequence the instructions are written

Function definition
This part contains the exact set of instructions which are executed during the function call. When a function is called from main(), the main function falls asleep and gets temporarily suspended. During this time the control goes to the function being called. When the function body is done executing main() resumes.

Quick Quiz → Write a program with three functions
1. Good morning function which prints "Good Morning"
2. Good afternoon function which prints "Good Afternoon"
3. Good night function which prints "Good night"

main() should call all of these in order 1 → 2 → 3

Scanned with CamScanner

URBAN EDGE

Important Points

→ Execution of a C program starts from main()

→ A C program can have more than one function

→ Every function gets called directly or indirectly from main()

→ There are two types of functions in C. Lets talk about them

Types of functions

1. Library functions → Commonly required functions grouped together in a library file on disk
2. User defined functions → These are the functions declared and defined by the user.

Why use functions?

1. To avoid rewriting the same logic again and again.
2. To keep track of what we are doing in a program
3. To test and check logic independently.

Scanned with CamScanner

URBAN EDGE

Passing values to functions

We can pass values to a function and can get a value in return from a function.

```
int sum(int a, int b)
```

The above prototype means that sum is a function which takes values a (of type int) and b (of type int) and returns a value of type int

function definition of sum can be:

```
int sum(int a, int b) {
    int c;
    c = a+b;
    return c;
}
```

⇒ a and b are parameters

Now we can call sum(2, 3); from main to get 5 in return.

↳ Here 2 & 3 are arguments

```
int d = sum(2, 3);
```

⇒ d becomes 5

Note:

1. Parameters are the values or variable placeholders in the function definition. Ex a & b.
2. Arguments are the actual values passed to the function to make a call. Ex 2 & 3.

Scanned with CamScanner

3. A function can return only one value at a time

4. If the passed variable is changed inside the function, the function call doesnt change the value in the calling function.

```
int change (int a) {
    a = 77;
    return 0;
}
```

⇒ Misnomer

change is a function which changes a to 77. No if we call it from main like this

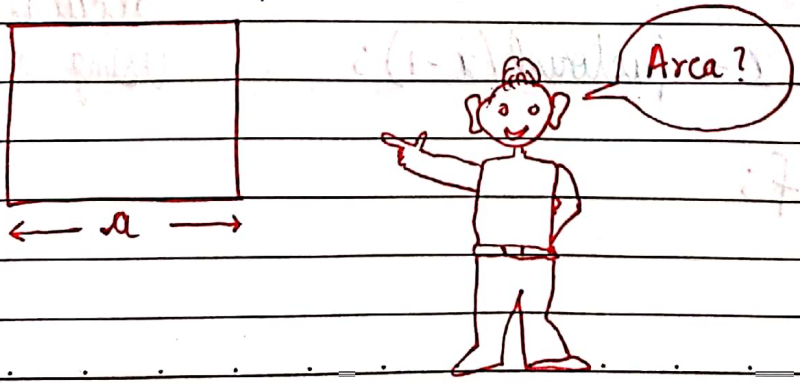
```
int b = 22
change (b);
printf("b is %d", b);
```

⇒ The value of b remains 22

⇒ prints "b is 22"

This happens because a copy of b is passed to the change function

Quick Quiz → Use the library functions to calculate the area of a square with side a.



Scanned with CamScanner

URBAN EDGE

Recursion

A function defined in C can call itself. This is called recursion.
A function calling itself is also called 'recursive' function.

Example of Recursion

A very good example of recursion is factorial

$$factorial(n) = 1 \times 2 \times 3 \cdots \times n$$

$$factorial(n) = \underbrace{1 \times 2 \times 3 \cdots n-1}_{} \times n$$

$$factorial(n) = factorial(n-1) \times n$$

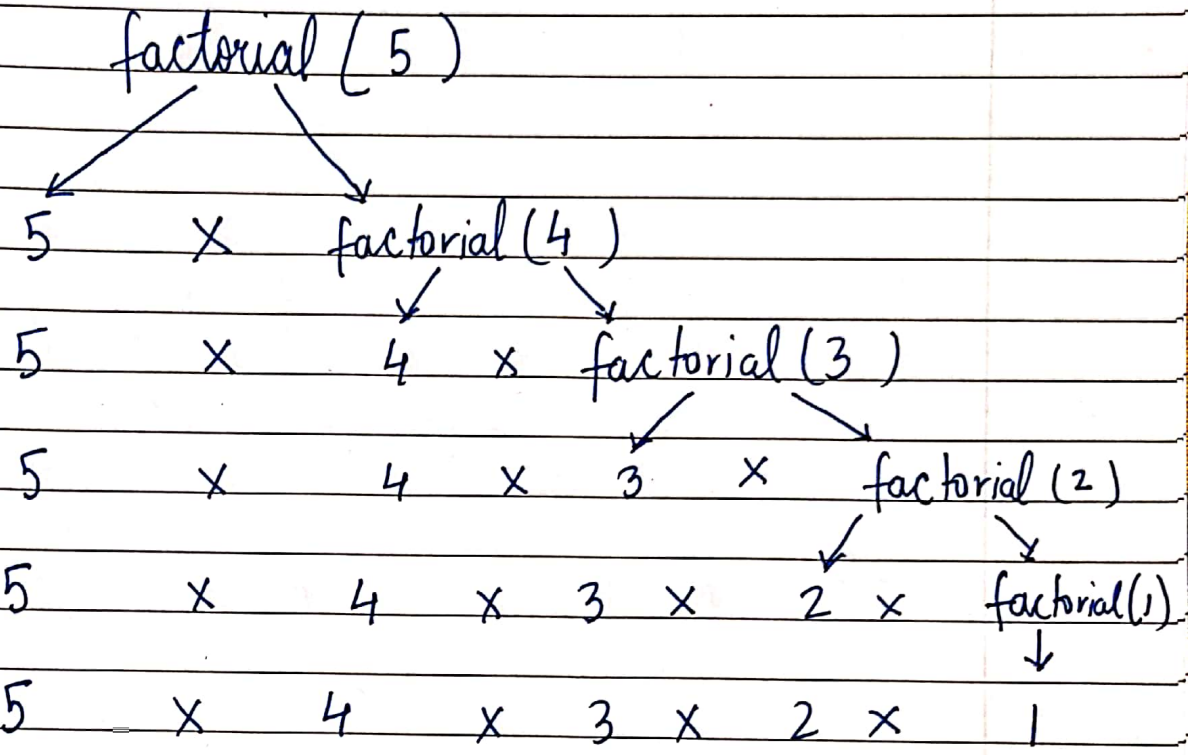
Since we can write factorial of a number in terms of itself, we can program it using recursion.

```
int factorial (int x) {
  int f;
  if (x == 0 || x == 1)
      return 1;
  else
    f = x * factorial(x-1);

  return f;

  }
```

⇒ A program to calculate factorial using recursion

Scanned with CamScanner

How does it work?



Important Notes:

1. Recursion is sometimes the most direct way to code an algorithm
2. The condition which doesnt call the function any further in a recursive function is called as the base condition.
3. Sometimes, due to a mistake made by the programmer, a recursive function can keep running without returning resulting in a memory error.

Scanned with CamScanner

URBAN EDGE

Chapter 5 - Practice Set

1. Write a program using functions to find average of three numbers.

2. Write a function to convert Celsius temperature into farenheit.

3. Write a function to calculate force of attraction on a body of mass m exerted by earth ($g = 9.8\ m/s^2$)

4. Write a program using recursion to calculate n^{th} element of fibbnacchi series.

5. What will the following line produce in a C program:

```
printf("%d %d %d\n", a, ++a, a++);
```

6. Write a recursive function to calculate the sum of first n natural numbers.

7. Write a program using functions to print the following pattern (first n lines)

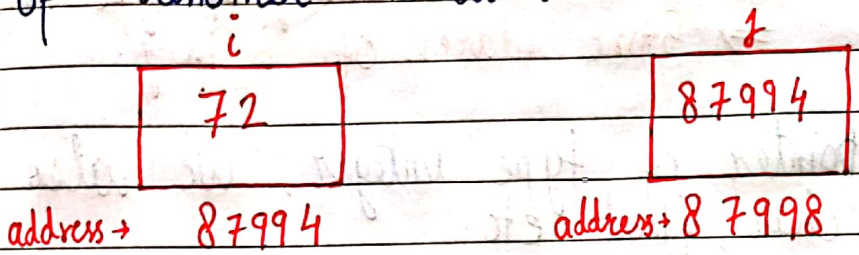
```
*
* * *
* * * * *
```

Scanned with CamScanner

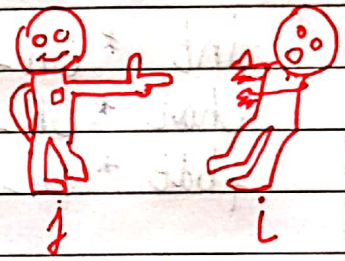
URBAN EDGE

Chapter 6 - Pointers

A pointer is a variable which stores the address of another variable



j is a pointer
j points to i



The "address of" (&) operator
The address of operator is used to obtain the address of a given variable

If you refer to the diagrams above

&i ⇒ 87994
&j ⇒ 87998

Format specifier for printing pointer address is '%u'

The 'value at address' operator (*)
The value at address or * operator is used to obtain the value present at a given memory address. It is denoted by *

*(&i) = 72
*(&j) = 87994

Scanned with CamScanner

URBAN EDGE

How to declare a Pointer ?

A pointer is declared using the following Syntax

```
int *j;     ⇒ declare a variable j of type int-pointer
j = &i      ⇒ Store address of i in j
```

Just like pointer of type integer, we also have pointers to char, float etc.

```
int *ch_ptr;   → Pointer to integer
char *ch_ptr;  → Pointer to character
fbdt *ch_ptr;  → Pointer to float
```

Although its a good practice to use meaningful variable names, we should be very careful while reading & working on porgrams from fellow programmers.

A Program to demonstrate pointers

```
#include <stdio.h>
int main() {
   int i = 8;
   int *j;
    j = &i;
   printf(" Add i = %u\n", &i);
   printf(" Add i = %u \n", j);
   printf(" Add j = %u \n", &j);
   printf(" Value i = %d \n", i);
   printf(" Value i = %d \n", *(&i));
   printf(" Value i = %d \n", *j);
   return 0;
}
```

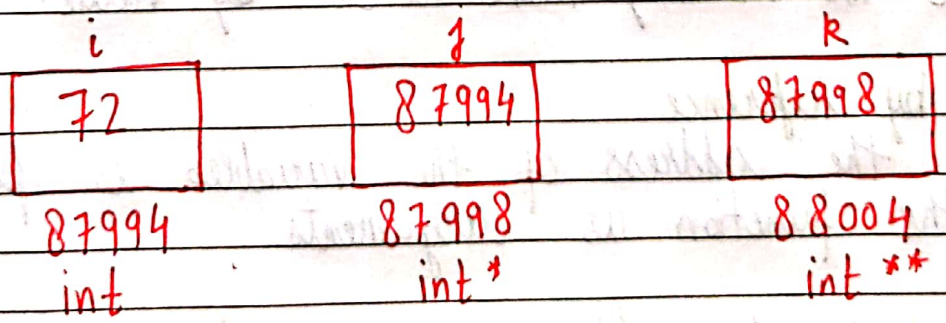
Scanned with CamScanner

Output:
Add i = 87994
Add i = 87994
Add j = 87998
Value i = 8
Value i = 8
Value i = 8

This program sums it all. If you understand it, you have got the idea of pointers.

Pointer to a pointer
Just like j is pointing to i or storing the address of i, we can have another variable k which can further store the address of j. What will be the type of k

```
int **k;
k = &j;
```



We can even go further one level and create a variable l of type int*** to store the address of k. We mostly use int* and int** sometimes in real world programs.

Scanned with CamScanner

URBAN EDGE

Types of function calls
Based on the way we pass arguments to the function, function calls are of two types.

1. Call by value → Sending the values of arguments
2. Call by reference → Sending the address of arguments

Call by value
Here the value of the arguments are passed to the function. Consider this example:

```
int c = Sum ( 3, 4 );   => assume x=3 and y=4
          x, y
```

if Sum is defined as Sum (int a, int b), the values 3 and 4 are copied to a and b. Now even if we change a and b, nothing happens to the variables x and y.
This is call by value.

In C we usually make a call by value.

Call by reference
Here the address of the variables is passed to the function as arguments

Now since the addresses are passed to the function, the function can now modify the value of a variable in calling function using * and & operators. Example:

Scanned with CamScanner

```
Void Swap (int *x, int *y)
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```

This function is capable of swapping the values passed to it. if a = 3 and b = 4 before a call to swap(a, b), a = 4 and b = 3 after calling swap.

```
int main() {
int a = 3
int b = 4
swap(a,b)
return 0;
}
```

⇒ a is 3 and b is 4

⇒ Now a is 4 and b is 3

Scanned with CamScanner

URBAN EDGE

Chapter 6 - Practice Set

1. Write a program to print the address of a variable. Use this address to get the value of this variable.

2. Write a program having a variable i. Print the address of i. Pass this variable to a function and print its address. Are these addresses same? why?

3. Write a program to change the value of a variable to ten times of its current value. Write a function and pass the value by reference.

4. Write a program using a function which calculates the sum and average of two numbers. Use pointers and print the values of sum and average in main()

5. Write a program to print the value of a variable i by using "pointer to pointer" type of variable.

6. Try problem 3 using call by value and verify that it doesnt change the value of the said variable.

Scanned with CamScanner

URBAN EDGE

Chapter 7 - Arrays

An array is a collection of similar (imp) elements.

One variable ⇒ capable of storing multiple values

Syntax

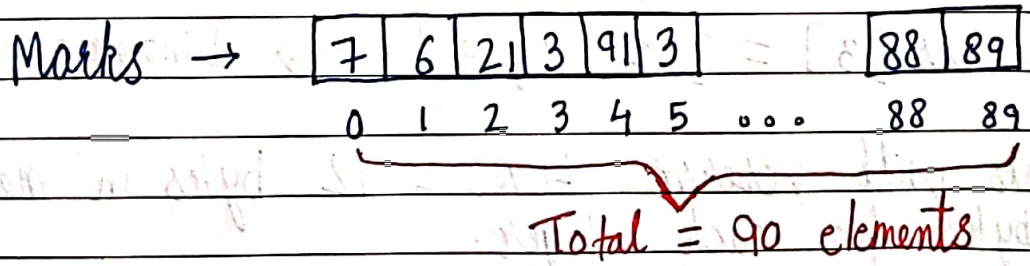
The syntax of declaring an Array looks like this:

```
int marks[90];      ⇒ Integer array
char name[20];      ⇒ Character array or String
float percentile[90]; ⇒ float array
```

The values can now be assigned to marks array like this:

```
marks[0] = 33;
marks[1] = 12;
```

Note: It is very important to note that the array index starts with 0.



Scanned with CamScanner

URBAN EDGE

Accessing elements

Elements of an array can be accessed using:

```
scanf("%d", &marks[0]);   ⇒ Input first value
printf("%d", marks[0]);   ⇒ output first value of the array
```

Quick Quiz → Write a program to accept marks of five students in an array and print them to the screen.

Initialization of an Array

There are many other ways in which an array can be initialized.

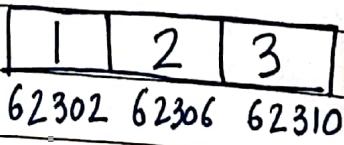
```
int cgpa[3] = {9, 8, 8}      ⇒ Arrays can be
float marks[] = {33, 40}        initialized while declaration
```

Arrays in memory

Consider this array:

```
int arr[3] = {1, 2, 3}   ⇒ 1 integer = 4 bytes
```

This will reserve 4 × 3 = 12 bytes in memory 4 bytes for each integer.



⇒ arr in memory

Scanned with CamScanner

URBAN EDGE

Pointer Arithmetic

A pointer can be incremented to point to the next memory location of that type.

Consider this example

```
int i = 32;
int *a = &i;   => a = 87994
a++;           => Now a = 87998
```

i → 32 (address → 87994)

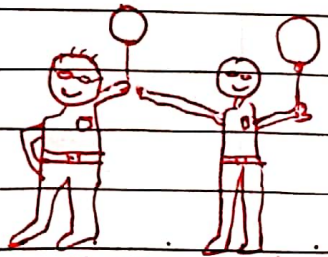
```
char a = 'A';
char *b = &a;  => b = 87994
b++;           => Now b = 87995
```

```
float i = 1.7;
float *a = &i; => Address of i or a = 87994
a++;           => Now a = 87998
```

Following operations can be performed on a pointers:

1. Addition of a number to a pointer
2. Subtraction of a number from a pointer
3. Subtraction of one pointer from another
4. Comparison of two pointer variables

Quick Quiz → Try these operations on another variable by creating pointers in a separate program. Demonstrate all the four operations.

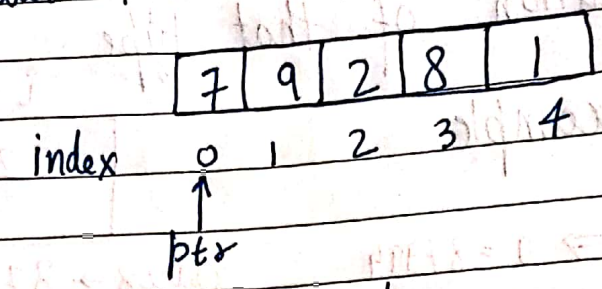


Yay! we understood pointer arithmetic

Scanned with CamScanner

Accessing Arrays using pointers

Consider this array



If ptr points to index 0, ptr++ will point to index 1 & so on...

This way we can have an integer pointer pointing to first element of the array like this:

```
int *ptr = &arr[0]; → or simply arr
ptr++;
*ptr => will have 9 as its value
```

Passing arrays to functions

Arrays can be passed to the functions like this

```
printArray(arr,n);      => function call
Void printArray(int *i, int n); => function prototype
                or
Void printArray(int i[], int n);
```

Scanned with CamScanner

URBAN EDGE

Multidimensional Arrays

An array can be of 2 dimension / 3 dimension / n dimensions

A 2 dimensional array can be defined as:

```
int arr[3][2] = { { 1, 4 }
                  { 7, 9 }
                  { 11, 22 } };
```

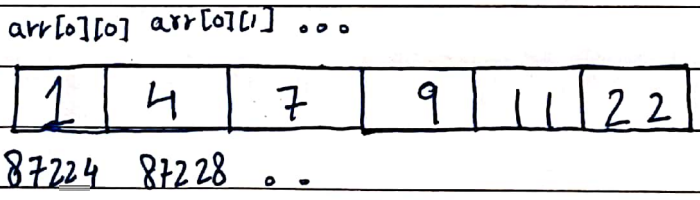
We can access the elements of this array as arr[0][0], arr[0][1] & so on...

arr[0][0] → Value = 1

arr[0][1] → Value = 4

2-D arrays in Memory

A 2d array like a 1-d array is stored in contiguous memory blocks like this:



Quick Quiz: Create a 2-d array by taking input from the user. Write a display function to print the content of this 2-d array on the screen.

Scanned with CamScanner

URBAN EDGE

Chapter 7 - Practice Set

1. Create an array of 10 numbers. Verify using pointer arithmetic that (ptr+2) points to the third element where ptr is a pointer pointing to the first element of the array.

2. If S[3] is a 1-D array of integers then *(S+3) refers to the third element:
   (i) True
   (ii) False
   (iii) Depends.

3. Write a program to create an array of 10 integers and store multiplication table of 5 in it.

4. Repeat Problem 3 for a general input provided by the user using scanf.

5. Write a program containing a function which reverses the array passed to it.

6. Write a program containing functions which counts the number of positive integers in an array.

7. Create an array of size 3 x 10 containing multiplication tables of the numbers 2, 7 and 9 respectively.

Scanned with CamScanner

8 Repeat problem 7 for a custom input given by the user.

9 Create a three-dimensional array and print the address of its elements in increasing order.

Scanned with CamScanner

Chapter 8 - Strings

A string is a 1-D character array terminated by a null ('\0')

↳ This is null character

null character is used to denote string termination characters are stored in contiguous memory locations

Initializing Strings

Since string is an array of characters, it can be initialized as follows:

```
char S[] = {'H', 'A', 'R', 'R', 'Y', '\0'};
```

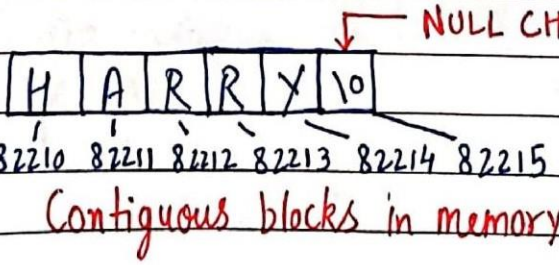
There is another shortcut for initializing strings in C language:

```
char S[] = "HARRY";
```

=> In this case C adds a null character automatically.

Strings In Memory

A string is stored just like an array in the memory as shown below



Contiguous blocks in memory

URBAN EDGE ___/___/___

Quick Quiz → Create a string using " " and print its content using a loop.

Printing Strings
A string can be printed character by character using printf and %c
But there is another convenient way to print strings in C.

```
Char St[ ] = "HARRY";
printf(" %s", St);
```
⟹ prints the entire string.

Taking string input from the user
We can use %s with scanf to take string input from the user:

```
Char St[50];
Scanf(" %s", &St);
```

Scanf automatically adds the null character when the the enter key is pressed.

Note:
1. The string should be short enough to fit into the array
2. Scanf cannot be used to input multi-word strings with spaces.

gets() and puts()

gets() is a function which can be used to receive a multi-word string.

```
Char st[30];
gets(st);
```
⇒ The entered string is stored in st!

Multiple gets() calls will be needed for multiple strings

Likewise, puts can be used to output a string.

```
puts(st);
```
⇒ prints the string
places the cursor on the next line

Declaring a string using pointers

We can declare strings using pointers

```
Char *ptr = "Harry";
```

This tells the compiler to store the string in memory and assigned address is stored in a char pointer

Note:

1. Once a string is defined using char st[] = "Harry", it cannot be reinitialized to something else.
2. A string defined using pointers can be reinitialized.

```
ptr = "Rohan";
```

URBAN EDGE

Standard library functions for Strings

C provides a set of standard library functions for string manipulation.

Some of the most commonly used string functions are:

Strlen()

This function is used to count the number of characters in the string excluding the null ('\0') character.

```
int length = Strlen(st);
```

These functions are declared under <string.h> header file

Strcpy()

This function is used to copy the content of second string into first string passed to it.

```
Char Source[] = "Harry";
char target[30];
Strcpy(target, Source);
```

⇒ target now contains "Harry"

Target string should have enough capacity to store the Source string.

URBAN EDGE

Strcat()
This function is used to concatenate two strings

```
char S1[] = "Hello";
char S2[] = "Harry";

Strcat(S1, S2);
```

⇒ S1 now contains "HelloHarry" <No space in between>

Strcmp()
This function is used to compare two strings. It returns: 0 if strings are equal
Negative value if first string's mismatching character's ASCII value is not greater than second string's corresponding mismatching character. It returns positive values otherwise.

```
Strcmp("Far", "Joke");
Strcmp("Joke", "Far");
```

⇒ Positive value
⇒ Negative value

URBAN EDGE

Chapter 8 - Practice Set

1. Which of the following is used to appropriately read a multi-word string
   (a) gets()
   (b) puts()
   (c) printf()
   (d) scanf()

2. Write a program to take string as an input from the user using %c and %s. Confirm that the strings are equal.

3. Write your own version of strlen function from <string.h>

4. Write a function slice() to slice a string. It should change the original string such tat it is now the sliced string. Take m and n as the start and ending position for slice.

5. Write your own version of strcpy function from <string.h>

6. Write a program to encrypt a string by adding 1 to the ascii value of its characters.

7. Write a program to decrypt the string encrypted using encrypt function in problem 6.

8 = Write a program to count the occurence of a given character in a string.

9 = Write a program to check whether a given character is present in a string or not.

URBAN EDGE

Chapter 9 - Structures

Arrays and strings ⇒ Similar data (int, float, char)

Structures can hold ⇒ dissimilar data

Syntax for creating Structures
A C structure can be created as follows:

```
struct employee {
    int code;
    float salary;
    char name[10];
};
```

⇒ This declares a new user defined data-type!

→ Semicolon is important

We can use this user defined data type as follows:

```
Struct employee e1;
Strcpy(e1.name, "Harry");
e1.code = 100;
e1.salary = 71.22;
```

⇒ Creating a structure variable

So a structure in C is a collection of variables of different types under a single name.

Quick Quiz: Write a program to store the details of 3 employees from user defined data. Use the structure declared above.

Why use structures?
We can create the data types in the employee structure separately but when the number of properties in a structure increases, it becomes difficult for us to create data variables without structures. In a nut shell:

(a) Structures keep the data organized.
(b) Structures make data mangement easy for the programmer.

Array of Structures
Just like an array of integers, an array of floats and an array of characters, we can create an array of structures.

```
struct employee facebook[100]; ⇒ An array of structures
```

We can access the data using:

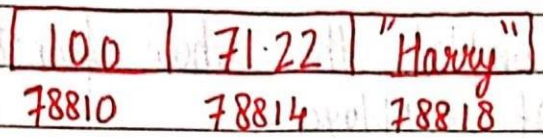
```
facebook[0].code = 100;
facebook[1].code = 101;
... & so on
```

Initializing Structures
Structures can also be initialized as follows:

```
struct employee harry = {100, 71.22, "Harry"};
struct employee shubh = {0}; ⇒ All elements set to 0
```

URBAN EDGE

Structures in memory
Structures are stored in contiguous memory locations
For the structure e1 of type struct employee, memory layout looks like this:



In an array of structures, these employee instances are stored adjacent to each other.

Pointer to structures
A pointer to structure can be created as follows:

```
struct employee *ptr;
ptr = &e1;
```

Now we can print structure elements using:

```
printf ("%d", *(ptr).code);
```

Arrow Operator
Instead of writing *(ptr).code, we can use arrow operator to access structure properties as follows

*(ptr).code or ptr -> code

Here -> is known as the arrow operator.

URBAN EDGE

Passing Structure to a function

A structure can be passed to a function just like any other data type.

```
Void show (struct employee e);
```
⇒ function prototype

Quick Quiz : Complete this show function to display the content of employee.

Typedef keyword

We can use the typedef keyword to create an alias name for data types in C. typedef is more commonly used with structures.

```
struct complex {
    float real;
    float img;
};
```
⇒ struct complex C_1, C_2;
for defining Complex numbers

```
typedef struct complex {
    float real;
    float img;
} complexNo;
```
⇒ complexNo C_1, C_2;
for defining Complex numbers

URBAN EDGE

Chapter 9 - Practice Set

1. Create a two dimensional Vector using structures in C

2. Write a function sumVector which returns the sum of two vectors passed to it. The vectors must be two-dimensional.

3. twenty integers are to be stored in memory. What will you prefer - Array or Structure?

4. Writ a program to illustrate the use of arrow operator -> in C.

5. Write a program with a structure representing a complex number.

6. Create an array of 5 complex numbers created in Problem 5 and display them with the help of a display function. The values must be taken as an input from the user.

7. Write problem 5's structure using typedef keyword.

8. Create a structure representing a bank account of a customer. What fields did you use and why?

9 = Write a structure capable of storing date. Write a function to compare those dates.

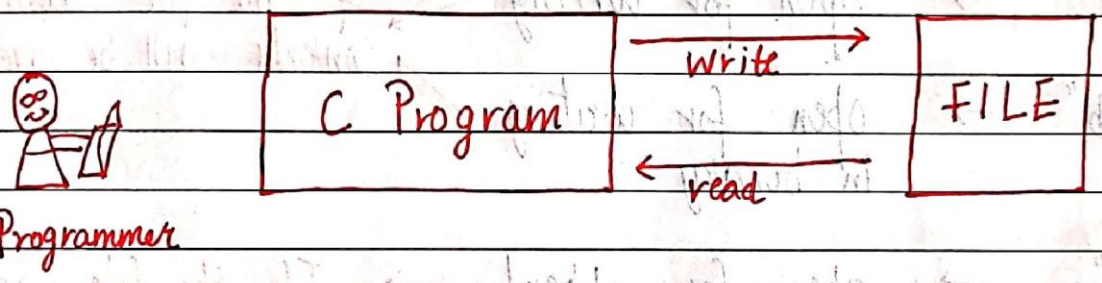
10 = Solve problem 9 for time using typedef keyword.

URBAN EDGE

Chapter 10 - File I/O

The Random Access Memory is volatile and its content is lost once the program terminates. In order to persist the data forever we use files.

A file is data stored in a storage device. A C program can talk to the file by reading content from it and writing content to it.



FILE pointer
The "FILE" is a structure which needs to be created for opening the file.
A file pointer is a pointer to this structure of the file.

FILE pointer is needed for communication between the file and the program.

A FILE pointer can be created as follows:

```
FILE *ptr;
ptr = fopen("filename.ext", "mode");
```

EDGE

File opening modes in C

C offers the programmers to select a mode for opening a file.

Following modes are primarily used in C File I/O

"r" → open for reading → If the file does not exist, fopen returns NULL

"rb" → open for reading in binary → If the file does not exist, fopen returns NULL

"w" → open for writing → If the file exists, the contents will be overwritten

"wb" → open for writing in binary → If the file exists, the contents will be overwritten

"a" → open for append → If the file does not exist, it will be created

Types of files

There are two types of files:

1. Text files (.txt, .c)
2. Binary files (.jpg, .dat)

Reading a file

A file can be opened for reading as follows:

```
FILE * ptr;
ptr = fopen ("Harry.txt", "r");
int num;
```

URBAN EDGE

Let us assume that "Harry.txt" contains an integer We can read that integer using:

```
fscanf (ptr, "%d", &num);
```

⇒ fscanf is file counterpart of scanf

This will read an integer from file in num variable.

Quick Quiz: Modify the program above to check whether the file exists or not before opening the file.

CLOSING the file

It is very important to close the file after read or write. This is acheived using fclose as follows:

```
fclose (ptr);
```

This will tell the compiler that we are done working with this file and the associated resources could be freed.

Writing to a file

We can write to a file in a very similar manner like we read the file

```
FILE *fptr;
fptr = fopen ("Harry.txt", "w");
```

```
int num = 432;
fprintf (fptr, "%d", num);

fclose (fptr);
```

fgetc() and fputc()
fgetc and fputc are used to read and write a character from/to a file

```
fgetc (ptr)
```
⟹ used to read a character from file

```
fputc ('c', ptr);
```
⟹ used to write character 'c' to the file

EOF : End of File
fgetc returns EOF when all the characters from a file have been read. So we can write a check like below to detect end of file

```
While (1) {
    ch = fgetc (ptr);
    if (ch == EOF) {
            break;
            }
    // code
    }
```

⟹ When all the content of a file has been read, break the loop!

EDGE

Chapter 10 - Practice Set

1. Write a program to read three integers from a file.

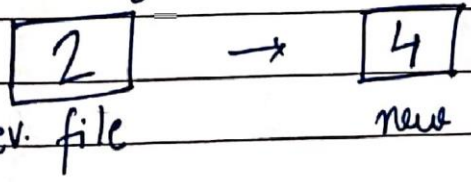
2. Write a program to generate multiplication table of a given number in text format. Make sure that the file is readable and well formatted.

3. Write a program to read a text file character by character and write its content twice in a separate file.

4. Take name and salary of two employees as input from the user and write them to a text file in the following format:

```
name1, 3300
name2, 7700
```

5. Write a program to modify a file containing an integer to double its value.



Project 2: Snake, Water, Gun

Snake, water, gun or Rock, paper, Scissors is a game most of us have played during school time. (I sometimes play it even now 😊)

Write a C program capable of playing this game with you.

Your program should be able to print the result after you choose Snake/water or gun.

URBAN EDGE

Chapter 11 - Dynamic Memory Allocation

C is a language with some fixed rules of programming. For example: changing the size of an array is not allowed.

Dynamic Memory Allocation

Dynamic memory allocation is a way to allocate memory to a data structure during the runtime. We can use DMA functions available in C to allocate and free memory during runtime.

Functions for DMA in C

Following functions are available in C to perform Dynamic memory Allocation:

1. malloc()
2. calloc()
3. free()
4. realloc()

malloc() function

malloc stands for memory allocation. It takes number of bytes to be allocated as an input and returns a pointer of type void

Syntax:

```
ptr = (int*) malloc(30 * sizeof(int))
```

- `(int*)` → casting void pointer to int
- `30` → space for 30 ints
- `sizeof(int)` → returns size of 1 int

The expression returns a null pointer if the memory cannot be allocated.

Quick Quiz: Write a program to create a dynamic array of 5 floats using malloc().

Calloc() function
Calloc stands for continuous allocation.
It initializes each memory block with a default value of 0.

Syntax:

```
ptr = (float*) calloc(30, sizeof(float));
```

↓ Allocates contiguous space in memory for 30 blocks (floats)

If the space is not sufficient, memory allocation fails and a NULL pointer is returned.

Quick Quiz: Write a program to create an array of size n using calloc where n is an integer entered by the user.

free() function
We can use free() function to de allocate the memory.
The memory allocated using calloc/malloc is not deallocated automatically.

Syntax:

free(ptr); ⟹ Memory of ptr is released.

Quick Quiz: Write a program to demonstrate the usage of free() with malloc().

realloc() function

Sometimes the dynamically allocated memory is insufficient or more than required.

realloc is used to allocate memory of new size using the previous pointer and size.

Syntax:

ptr = realloc(ptr, newSize);

ptr = realloc(ptr, 3 * sizeof(int));

⇓

ptr now points to this new block of memory capable of storing 3 integers.

Chapter 11 - Practice Set

1. Write a program to dynamically create an array of size 6 capable of storing 6 integers.

2. Use the array in problem 1 to store 6 integers entered by the user

3. Solve problem 1 using calloc()

4. Create an array dynamically capable of storing 5 integers. Now use realloc so that it can now store 10 integers.

5. Create an array of multiplication table of 7 upto 10 ($7 \times 10 = 70$). Use realloc to make it store 15 numbers (from 7×1 to 7×15).

6. Attempt problem 4 using calloc().

URBAN EDGE ___/___/___

Data Structures & Algorithms by CodeWithHarry

This course will get you prepared for placements and will teach you how to create efficient and fast algorithms.

Data structures and algorithms are two different things.

Data Structures: Arrangement of data so that they can be used efficiently in memory (data items)

Algorithms: Sequence of steps on data using efficient data structures to solve a given problem.

Other Terminology

Database :- Collection of information in permanent storage for faster retrieval and updation.

Data warehousing :- Management of huge amount of legacy data for better analysis.

Big data :- Analysis of too large or complex data which cannot be dealt with traditional data processing application.

Data Structures and Algorithms are nothing new. If you have done programming in any language like C you must have used Arrays → A data structure and some sequence of processing steps to solve a problem → Algorithm

URBAN EDGE

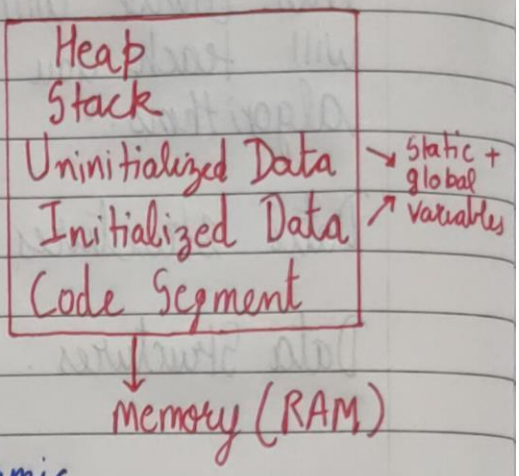
Memory layout of C programs

When the program starts, its Code is copied to the main memory.

Stack holds the memory occupied by the functions.

Heap contains the data which is requested by the program as dynamic memory.

Initialized and uninitialized data segments hold initialized and uninitialized global variables respectively



URBAN EDGE

Time Complexity & Big O notation

This morning I wanted to eat some pizzas; So I asked my brother to get me some from Dominos (3 km far)
He got me the pizza and I was happy only to realize it was too less for 29 friends who came to my house for a surprize visit!

My brother can get 2 pizzas for me on his bike but pizza for 29 friends is too huge of an input for him which he cannot handle.

2 pizzas ⟶  okay! not a big deal!

68 pizzas ⟶  Not possible! in short time

What is Time Complexity?
Time Complexity is the study of efficiency of algorithms.

(i) Time Complexity = How time taken to execute an algorithm grows with the size of the input!

Consider two developers who created an algorithm to sort n numbers. Shubham and Rohan did this independently.

When ran for input size n, following results were recorded:

no of elements (n)	Shubham's Algo	Rohan's Algo
10 elements	90 ms	122 ms
70 elements	110 ms	124 ms
110 elements	180 ms	131 ms
1000 elements	2 s	800 ms

We can see that initially Shubham's algorithm was shining for smaller input but as the number of elements increases Rohan's algorithm looks good!

Quick Quiz: Who's Algorithm is better?

Time Complexity: Sending GTA V to a friend
Let us say you have a friend living 5 kms away from your place. You want to send him a game.
Final exams are over and you want him to get this 60 GB file from you. How will you send it to him?
Note that both of you are using (JIO) 4G with 1 GB/day data limit.

The best way to send him the game is by delivering it to his house.
Copy the game to a Hard disk and send it!

Will yo do the same thing for sending a game like minesweeper which is in kbs of size?
No because you cand send it via internet.

As the file size grows, time taken by online sending increases linearly → $O(n^1)$
As the file size grows, time taken by physical sending remains constant. $O(n^0)$ or $O(1)$

Calculating Order in terms of Input size
In order to calculate the order, most impactful term containing n is taken into account.
↳ Size of input

Let us assume that formula of an algorithm in terms of input size n looks like this:

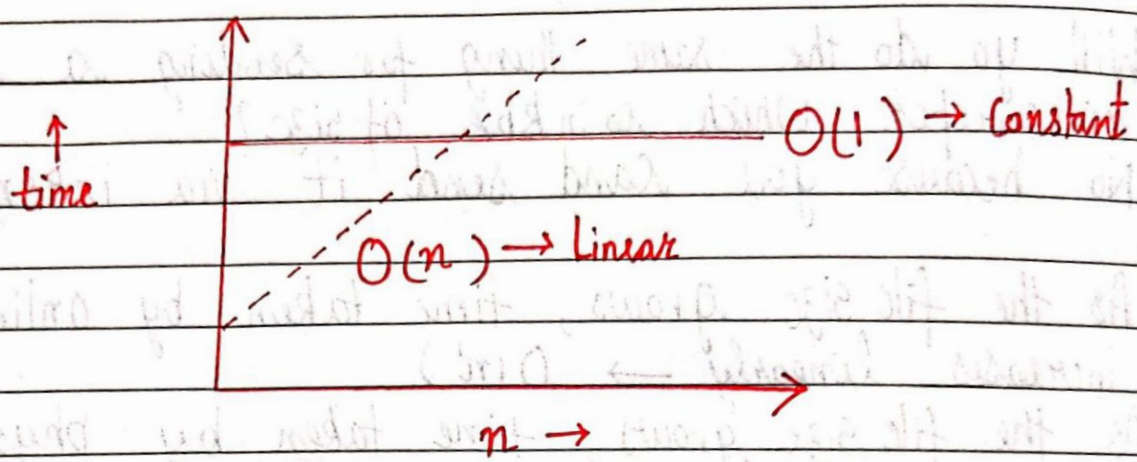
Algo 1 → $k_1 n^2 + k_2 n + 36$ ⇒ $O(n^2)$
($k_1 n^2$: Highest order term; $k_2 n + 36$: can ignore lower order terms)

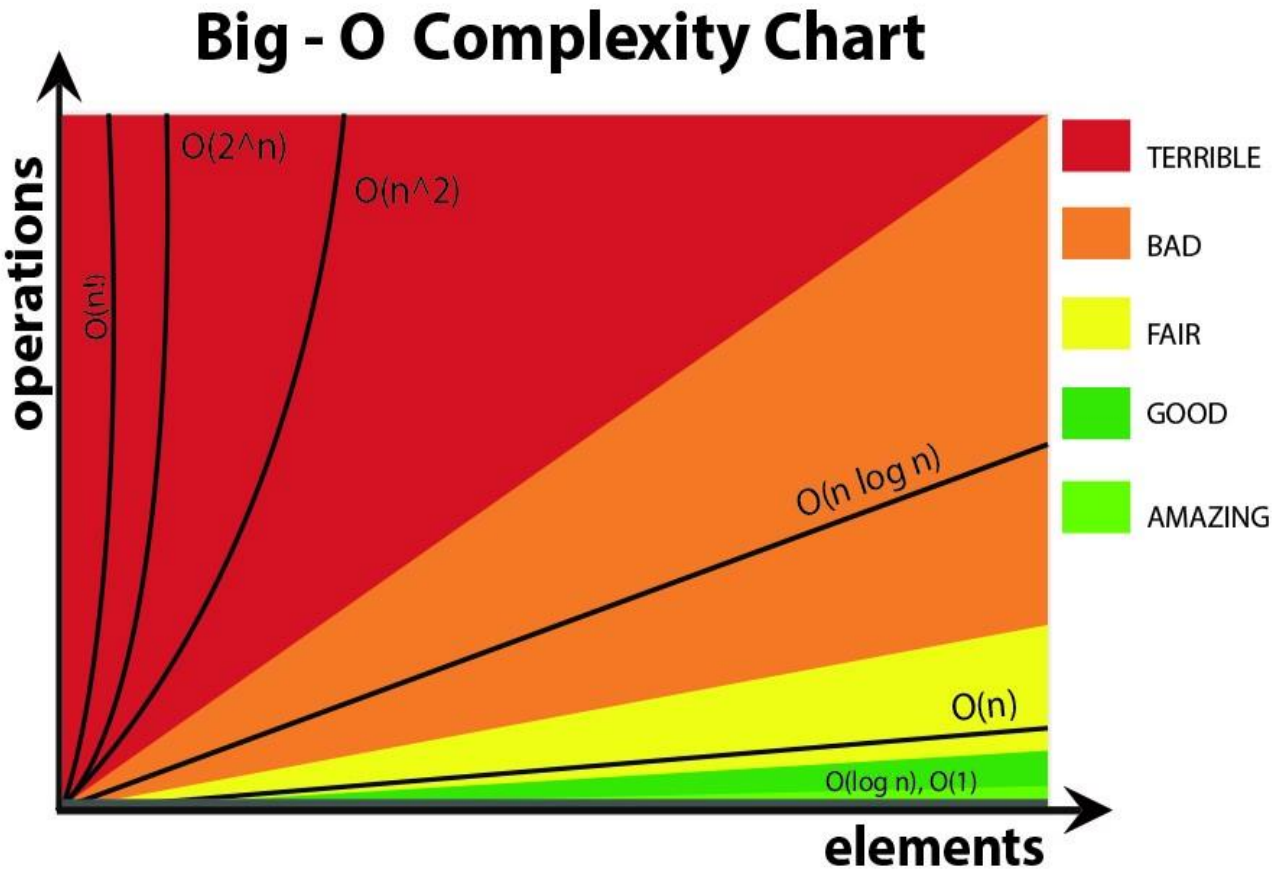
Algo 2 → $k_1 k_2^2 + k_3 k_2 + 8$
⇓
$k_1 k_2^2 n^0 + k_3 k_2 + 8$ ⇒ $O(n^0)$ or $O(1)$

Note that these are the formulas for time taken by them.

Visualising Big O

If we were to plot O(1) and O(n) on a graph, they will look something like this :





Source: https://stackoverflow.com/questions/3255/big-o-how-do-you-calculate-approximate-it

Asymptotic Notations

Asymptotic notations give us an idea about how good a given algorithm is compared to some other algorithm
Let us see the mathematical definition of "order of" now.

Primarily there are three types of widely used asymptotic notations.

1) Big oh notation (O)
2) Big omega notation (Ω)
3) Big theta notation (Θ) ⟶ Widely used one!

Big oh notation

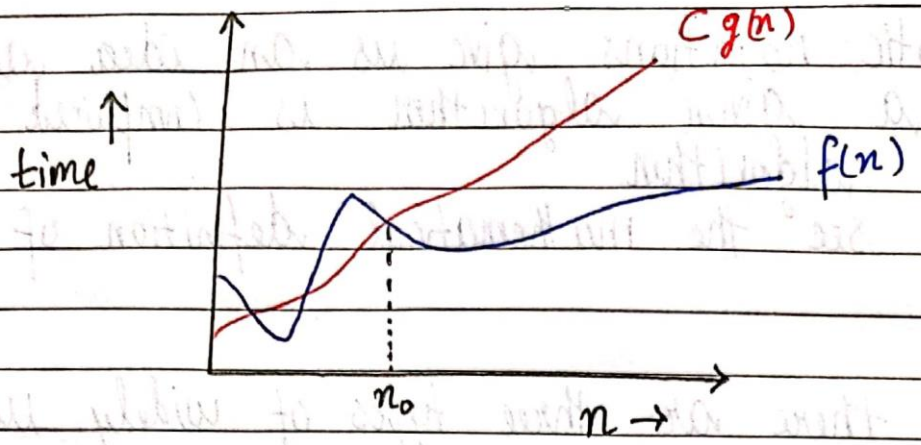
Big oh notation is used to describe asymptotic upper bound
Mathematically, if $f(n)$ describes running time of an algorithm; $f(n)$ is $O(g(n))$ iff there exist positive constants C and n_0 such that

$$0 \leq f(n) \leq C\,g(n) \quad \text{for all } n \geq n_0$$

⇓
used to give upper bound on a function.

if a function is $O(n)$, it is automatically $O(n^2)$ as well!

Graphic example for Big oh (O)



Big Omega notation

Just Like O notation provides an asymptotic upper bound, Ω notation provides asymptotic lower bound. Let $f(n)$ define running time of an algorithm;

$f(n)$ is said to be $\Omega(g n)$ if there exists positive constants C and n_0 such that

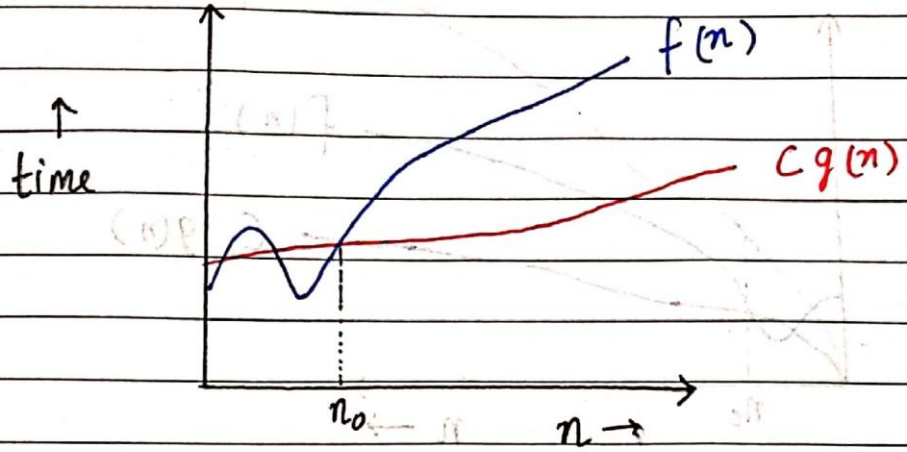
$$0 \leq c\,g(n) \leq f(n) \quad \text{for all } n \geq n_0$$

⇓ used to give lower bound on a function

if a function is $O(n^2)$ it is automatically $O(n)$ as well

URBAN EDGE

Graphic example for Big omega (Ω)



Big theta notation

Let $f(n)$ define running time of an algorithm

$f(n)$ is said to be $\Theta(gn)$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

Mathematically,

$$0 \leq f(n) \leq C_1 g(n) \quad \forall\ n \geq n_0$$

$$0 \leq C_2 g(n) \leq f(n) \quad \forall\ n \geq n_0$$

Sufficiently large value of n

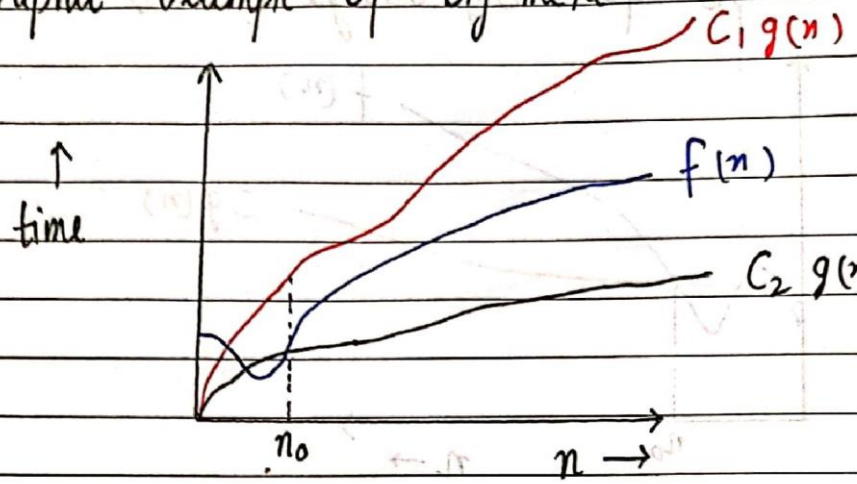
Merging both the equations, we get:

$$0 \leq C_2 g(n) \leq f(n) \leq C_1 g(n) \quad \forall\ n \geq n_0$$

The equation simply means there exist positive constants C_1 and C_2 such that $f(n)$ is sandwiched between $C_2 g(n)$ and $C_1 g(n)$

URBAN EDGE

Graphic example of Big theta



Which one of these to use ?
Since Big theta gives a better picture of runtime for a given algorithm, most of the interviewers expect you to provide an answer in terms of Big theta when they say "Order of".

Quick Quiz :- Prove that n^2+n+1 is $O(n^3)$, $\Omega(n^2)$ and $\Theta(n^2)$ using respective definitions.

Increasing order of common runtimes

$$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n^n$$

Better ⟵ ⇓ ⟶ Worse

Common runtimes from better to worse ⟶

URBAN EDGE

Best, worst and Expected Case

Sometimes we get lucky in life. Exams cancelled when you were not prepared, surprise test when you were prepared etc. ⟹ Best case

Some times we get unlucky. Questions you never prepared asked in exams, rain during sports period etc. ⇒ Worst case

But overall the life remains balance with the mixture of lucky and unlucky times. ⇒ Expected case.

Analysis of a search algorithm

Consider an array which is sorted in increasing order

1	7	18	28	50	180

We have to search a given number in this array and report whether its present in the array or not.

Algo 1 → Start from first element until an element greater than or equal to the number to be searched is found.

Algo 2 → Check whether the first or the last element is equal to the number. If not find the number between these two elements (center of the array). If the center element is greater than the number to be searched, repeat the process for first half else repeat for second half until the number is found.

Analyzing Algo 1

If we really get lucky, the first element of the array might turn out to be the element we are searching for. Hence we made just one comparison.

$$\text{Best case Complexity} = O(1)$$

If we are really unlucky, the element we are searching for might be the last one.

$$\text{Worst case Complexity} = O(n)$$

For calculating Average case time, we sum the list of all the possible case's runtime and divide it with the total number of cases.

⇓

Sometimes calculation of average case time gets very complicated

Analyzing Algo 2

If we get really lucky, the first element will be the only one which gets compared

$$\text{Best case Complexity} = O(1)$$

If we get unlucky we will have to keep dividing the array into halves until we get a single element (the array gets finished)

Worst case Complexity = $O(\log n)$

What log(n)? What is that

log(n) → Number of times you need to half the array of size n before it gets exhausted

$\log 8 = 3 \implies \frac{8}{2} \to \frac{4}{2} \to \frac{2}{2}$ → Cant break anymore.

$1 + 1 + 1$

$\log 4 = 2 \implies \frac{4}{2} \to \frac{2}{2}$ → Cant break anymore.

$1 + 1$

Log n simply means how many time I need to divide n units such that we cannot divide them (into halves) anymore.

Space Complexity

Time is not the only thing we worry about while analyzing algorithms. Space is equally important.

Creating an array of size n → O(n) space
↳ Size of input

If a function calls itself recursively n times its space complexity is O(n)

URBAN EDGE

Quick Quiz → Calculate Space Complexity of a function which calculates factorial of a given number n.

Why cant we calculate Complexity in seconds?

→ Not everyone's Computer is equally powerful

→ Asymptotic analysis is the measure of how time (runtime) grows with input

Techniques to Calculate Time Complexity

Once we are able to write the runtime in terms of size of the input (n), we can find the time complexity.

For example $T(n) = n^2 \Rightarrow O(n^2)$

$T(n) = \log n \Rightarrow O(\log n)$

Some tricks to calculate complexity

1. Drop the constants : Any thing you might think is $O(3n)$ is $O(n)$
   ↳ Better representation
2. Drop the non dominant terms : Any thing you represent as $O(n^2+n)$ can be written as $O(n^2)$
3. Consider all variables which are provided as input : $O(mn)$ & $O(mnq)$ might exist for some cases!

In most of the cases, we try to represent the runtime in terms of the input which can be more than one in number. For example :

Painting a park of dimension $m \times n \Rightarrow O(mn)$

Time Complexity – Competitive Practice Sheet

1. Fine the time complexity of the func1 function in the program show in program1.c as follows:

```c
#include <stdio.h>

void func1(int array[], int length)
{
    int sum = 0;
    int product = 1;
    for (int i = 0; i < length; i++)
    {
        sum += array[i];
    }

    for (int i = 0; i < length; i++)
    {
        product *= array[i];
    }
}

int main()
{
    int arr[] = {3, 5, 66};
    func1(arr, 3);
    return 0;
}
```

2. Fine the time complexity of the func function in the program from program2.c as follows:

```c
void func(int n)
{
    int sum = 0;
    int product = 1;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            printf("%d , %d\n", i, j);
        }
    }

}
```

3. Consider the recursive algorithm above, where the random(int n) spends one unit of time to return a random integer which is evenly distributed within the range [0,n][0,n]. If the average processing time is T(n), what is the value of T(6)?

```
int function(int n)
{
    int i;

    if (n <= 0)
    {
        return 0;
    }
    else
    {
        i = random(n - 1);
        printf("this\n");
        return function(i) + function(n - 1 - i);
    }
}
```

4. Which of the following are equivalent to O(N)? Why?
    a) O(N + P), where P < N/9
    b) 0(9N-k)
    c) O(N + 8log N)
    d) O(N + M^2)

5. The following simple code sums the values of all the nodes in a balanced binary search tree. What is its runtime?

```
int sum(Node node)
{
    if (node == NULL)
    {
        return 0;
    }
    return sum(node.left) + node.value + sum(node.right);
}
```

6. Find the complexity of the following code which tests whether a give number is prime or not?

```
int isPrime(int n){
    if (n == 1){
        return 0;
    }

    for (int i = 2; i * i < n; i++) {
        if (n % i == 0)
            return 0;
    }
```

```
    return 1;
}
```

7. What is the time complexity of the following snippet of code?

```
int isPrime(int n){

    for (int i = 2; i * i < 10000; i++) {
        if (n % i == 0)
            return 0;
    }

    return 1;
}
isPrime();
```

URBAN EDGE

Operations on an Array

Following operations are supported by an array.

Traversal
Insertion
Deletion
Search

⇒ There can be many other operations one can perform on arrays as well!
eg: Sorting asc., Sorting desc.

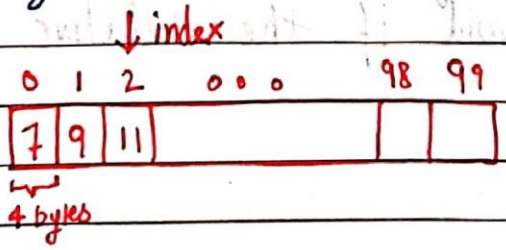
Traversal

Visiting every element of an array once → Traversal

Why traversal? → For use cases like:
→ Storing all elements → using scanf
→ Printing all elements → using printf

An important note about arrays
If we create an array of length 100 using a[100] in C language, we need not use all the elements. It is possible for a program to use just 60 elements out of these 100.
↳ But we cannot go beyond 100 elements.

An array can easily be traversed using a for loop in C language

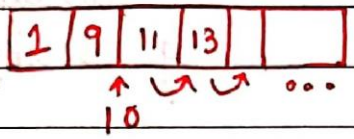


URBAN EDGE

Insertion

An element can be inserted in an array at a specified position.

In order for this operation to be successful, the array should have enough capacity

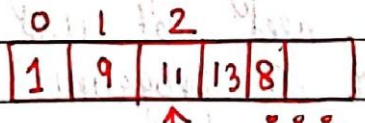


⇒ Elements need to be shifted to maintain relative order.

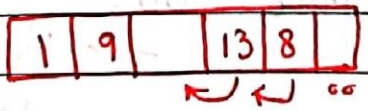
When no position is specified its best to insert the element at the end.

Deletion

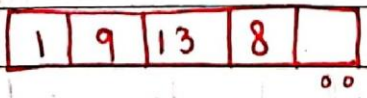
An element at specified position can be deleted creating a void which needs to be fixed by shifting all the elements to the left as follows:



Delete 11 at ind 2



Shift the elements

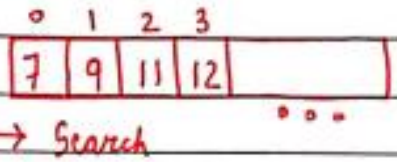


Deletion done!

We can also bring the last element of the array to fill the void if the relative ordering is not important

Searching

Searching can be done by traversing the array until the element to be searched is found



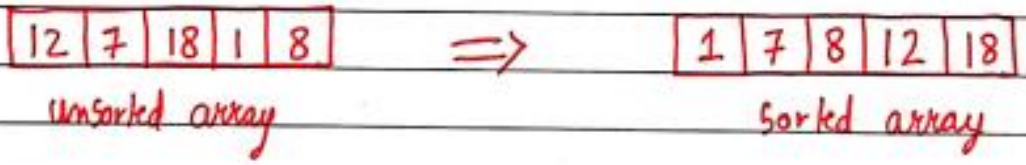
⇓

For sorted array time taken to search is much less than unsorted array!!

Sorting

Sorting means arranging an array in order (asc or desc)

We will see various sorting techniques later in the course.



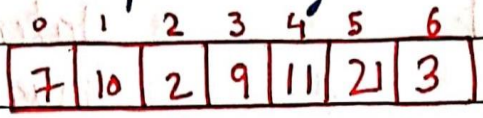
Unsorted array

Sorted array

Linear Vs Binary Search

Linear Search

Searches for an element by visiting all the elements sequentially until the element is found.



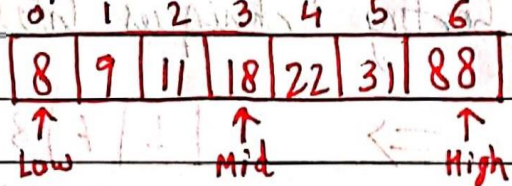
⇒ can be sorted or unsorted

Search '2' → Element found

WC Complexity : O(n)

Binary Search

Searches for an element by breaking the search space into half in a sorted array.



WC Complexity O(log n)

Search 18

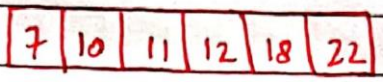
The search continues towards either side of mid based on whether the element to be searched is lesser or greater than mid.

	Linear Search	Binary Search
1.	Works on both sorted and unsorted arrays	Works only on sorted arrays
2.	Equality operations	inequality operations
3.	O(n) WC Complexity	O(log n) WC Complexity

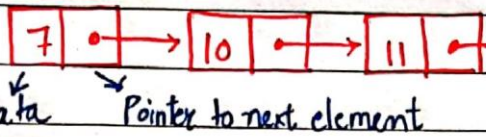
URBAN EDGE

Introduction to Linked Lists

Linked lists are similar to arrays (Linear data structures)



⇒ In Arrays elements are stored in contiguous memory locations



⇒ In linked lists, elements are stored in non contiguous memory locations

Why Linked Lists?

Memory and the capacity of an array remains fixed.
In case of linked lists, we can keep adding and removing elements without any capacity constraints

Drawbacks of Linked lists

→ Extra memory space for pointers is required (for every node 1 pointer is needed)
→ Random access not allowed as elements are not stored in contiguous memory locations.

Implementation

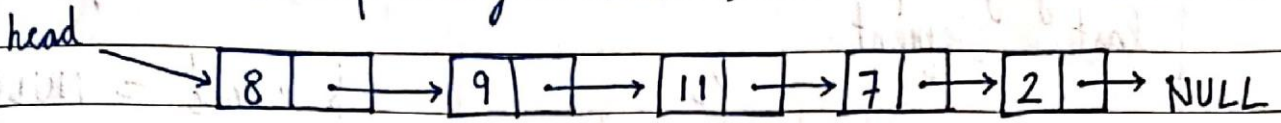
Linked list can be implemented using a structure in C language

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

⇒ Self refencing structure

Deletion in a Linked List

Consider the following Linked List

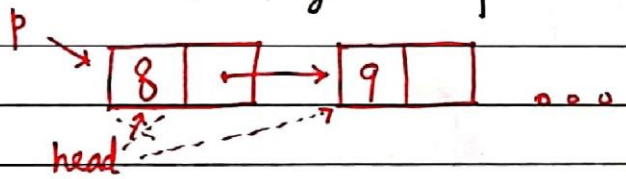


Deletion can be done for the following cases:

1. Deleting the first Node
2. Deleting the node at an index
3. Deleting the last Node
4. Deleting the first node with a given value.

The deletion just like insertion is done by rewiring the pointer connections, the only caveat being: we need to free the memory of the deleted node using free().

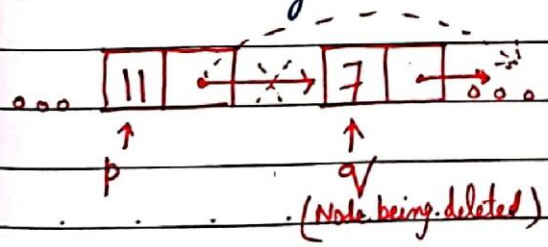
Case 1: Deleting the first node




```
Struct Node * p = head
head = head → next;
free(p);
```

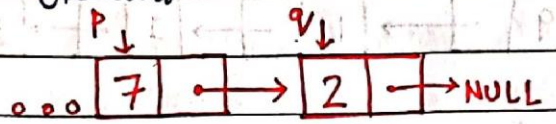
Case 2: Deleting the node at an index

for deleting a given node, we first bring a temporary pointer p before element to be deleted and q on the element being deleted



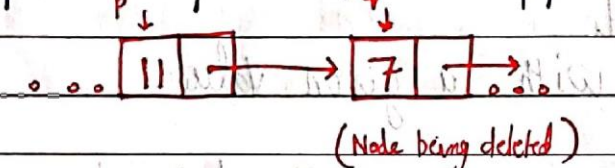

```
p → next = q → next;
free(q);
```

Case 3: Deleting the last Node
Last node can be deleted just like Case 2 by bringing p on second last element and q on last element.



p→next = NULL
free(q) → To free the memory!

Case 4: Delete the first node with a given value
This can be done exactly like Case 2 by bringing pointers p & q to appropriate positions

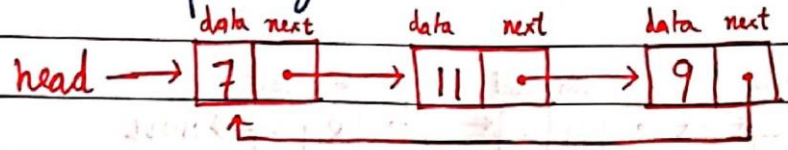


p→next = q→next;
free(q);

URBAN EDGE

Circular Linked List

A circular linked list is a linked list where the last element points to the first element (head) hence forming a circular chain



Operations on a circular linked List
operations on a circular linked lists can be performed exactly like a singly linked list.

Visit www.codewithharry.com for practice sets / code / more

URBAN EDGE

Doubly Linked List

In a doubly linked list, each node contains a data part along with the two addresses, one for the previous node and the other one for the next node

Implementation

A doubly linked list can be implemented in C language as follows:

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

Operations on a Doubly Linked List

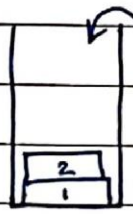
The insertion and deletion on a Doubly linked list can be performed by rewiring pointer connections just like we saw in a Singly linked list.

The difference here lies in the fact that we need to adjust two pointers (prev & next) instead of one (next) in the case of a Doubly linked list.

URBAN EDGE

Introduction to Stack Data Structure

Stack is a linear data structure. Operations on stack are performed in LIFO (Last in first out) order.



⇒ Item 2 which entered the basket last will be the first one to come out

LIFO (Last in first out)

Applications of Stack

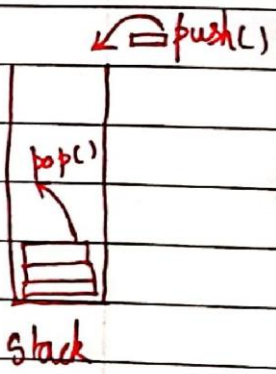
1. Used in function calls
2. Infix to postfix conversion (and other similar conversions)
3. Parenthesis matching & more...

Stack ADT

In order to create a stack we need a pointer to the topmost element along with other elements which are stored inside the stack.

Some of the operations of stack ADT are:

1. push() → push an element into the stack
2. pop() → remove the topmost element from the stack
3. peek(index) → Value at a given position is returned
4. isEmpty() / isFull() → Determine whether the stack is empty or full.



Implementation

A stack is a collection of elements with certain operations following LIFO (Last in First out) dicipline.

A Stack can be implemented using an array or a linked list

URBAN EDGE ___/___/___

Data Structures & Algorithms by CodeWithHarry

This course will get you prepared for placements and will teach you how to create efficient and fast algorithms.

Data structures and algorithms are two different things.

Data Structures: Arrangement of data so that they can be used efficiently in memory (data items)

Algorithms: Sequence of steps on data using efficient data structures to solve a given problem.

Other Terminology

Database :- Collection of information in permanent storage for faster retrieval and updation.

Data warehousing :- Management of huge amount of legacy data for better analysis.

Big data :- Analysis of too large or complex data which cannot be dealt with traditional data processing application.

Data Structures and Algorithms are nothing new. If you have done programming in any language like C you must have used Arrays → A data structure and some sequence of processing steps to solve a problem → Algorithm

URBAN EDGE

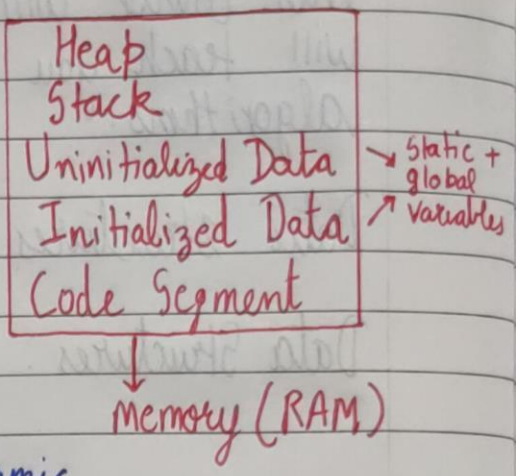
Memory layout of C programs

When the program starts, its Code is copied to the main memory.

Stack holds the memory occupied by the functions.

Heap contains the data which is requested by the program as dynamic memory.

Initialized and uninitialized data segments hold initialized and uninitialized global variables respectively



URBAN EDGE

Time Complexity & Big O notation

This morning I wanted to eat some pizzas; So I asked my brother to get me some from Dominos (3 km far)
He got me the pizza and I was happy only to realize it was too less for 29 friends who came to my house for a surprize visit!

My brother can get 2 pizzas for me on his bike but pizza for 29 friends is too huge of an input for him which he cannot handle.

2 pizzas ⟶  okay! not a big deal!

68 pizzas ⟶  Not possible! in short time

What is Time Complexity?
Time Complexity is the study of efficiency of algorithms.

(i) Time Complexity = How time taken to execute an algorithm grows with the size of the input!

Consider two developers who created an algorithm to sort n numbers. Shubham and Rohan did this independently.

When ran for input size n, following results were recorded:

no of elements (n)	Shubham's Algo	Rohan's Algo
10 elements	90 ms	122 ms
70 elements	110 ms	124 ms
110 elements	180 ms	131 ms
1000 elements	2 s	800 ms

We can see that initially Shubham's algorithm was shining for smaller input but as the number of elements increases Rohan's algorithm looks good!

Quick Quiz: Who's Algorithm is better?

Time Complexity: Sending GTA V to a friend
Let us say you have a friend living 5 kms away from your place. You want to send him a game.
Final exams are over and you want him to get this 60 GB file from you. How will you send it to him?
Note that both of you are using (JIO) 4G with 1 GB/day data limit.

The best way to send him the game is by delivering it to his house.
Copy the game to a Hard disk and send it!

Will yo do the same thing for sending a game like minesweeper which is in kbs of size?
No because you cand send it via internet.

As the file size grows, time taken by online sending increases linearly → $O(n^1)$
As the file size grows, time taken by physical sending remains constant. $O(n^0)$ or $O(1)$

Calculating Order in terms of Input size
In order to calculate the order, most impactful term containing n is taken into account.
↳ Size of input

Let us assume that formula of an algorithm in terms of input size n looks like this:

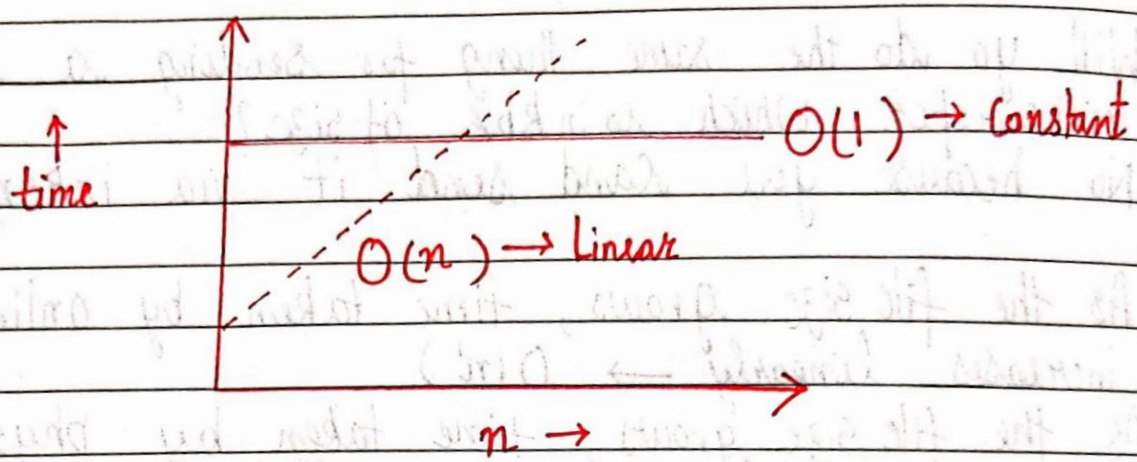
Algo 1 → $k_1 n^2 + k_2 n + 36$ ⇒ $O(n^2)$
($k_1 n^2$: Highest order term; $k_2 n + 36$: can ignore lower order terms)

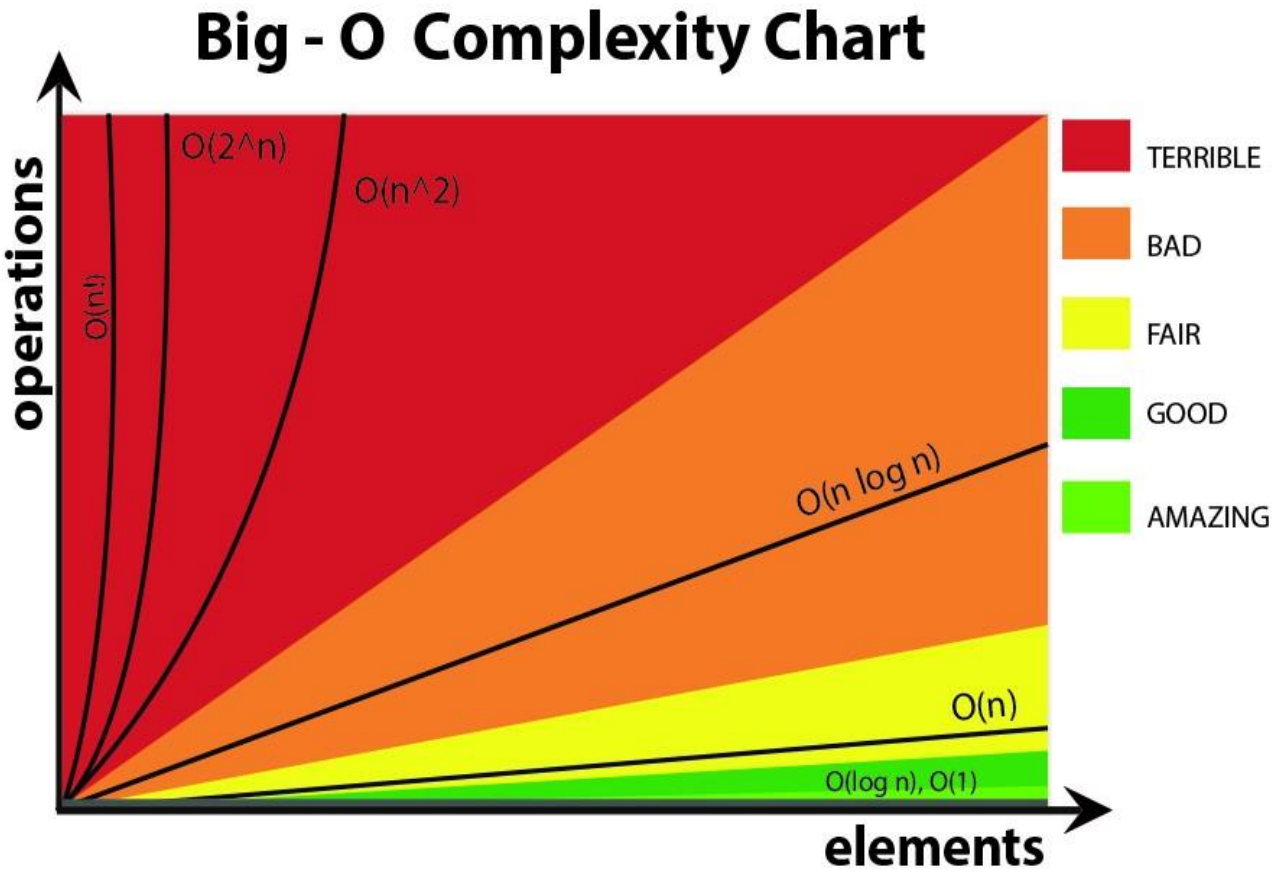
Algo 2 → $k_1 k_2^2 + k_3 k_2 + 8$
⇓
$k_1 k_2^2 n^0 + k_3 k_2 + 8$ ⇒ $O(n^0)$ or $O(1)$

Note that these are the formulas for time taken by them.

Visualising Big O

If we were to plot O(1) and O(n) on a graph, they will look something like this :





Source: https://stackoverflow.com/questions/3255/big-o-how-do-you-calculate-approximate-it

Asymptotic Notations

Asymptotic notations give us an idea about how good a given algorithm is compared to some other algorithm
Let us see the mathematical definition of "order of" now.

Primarily there are three types of widely used asymptotic notations.

1) Big oh notation (O)
2) Big omega notation (Ω)
3) Big theta notation (Θ) ⟶ Widely used one!

Big oh notation

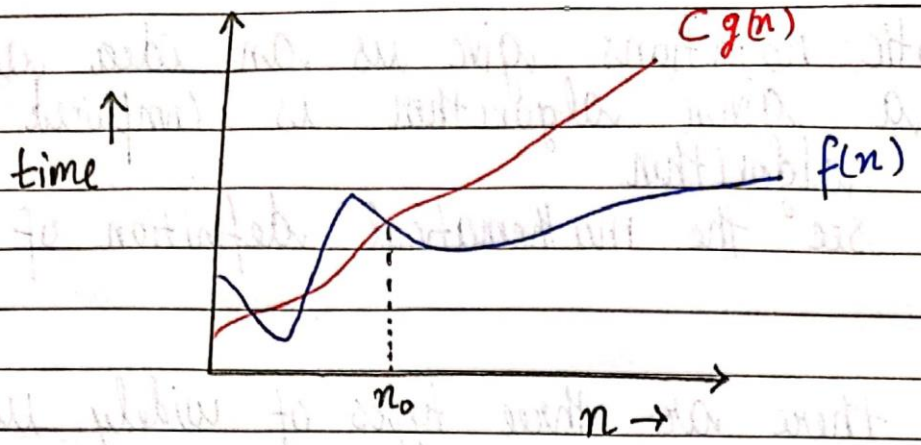
Big oh notation is used to describe asymptotic upper bound
Mathematically, if $f(n)$ describes running time of an algorithm; $f(n)$ is $O(g(n))$ iff there exist positive constants C and n_0 such that

$$0 \leq f(n) \leq C\,g(n) \quad \text{for all } n \geq n_0$$

⇓
used to give upper bound on a function.

if a function is $O(n)$, it is automatically $O(n^2)$ as well!

Graphic example for Big oh (O)



Big Omega notation

Just Like O notation provides an asymptotic upper bound, Ω notation provides asymptotic lower bound. Let $f(n)$ define running time of an algorithm;

$f(n)$ is said to be $\Omega(g n)$ if there exists positive constants C and n_0 such that

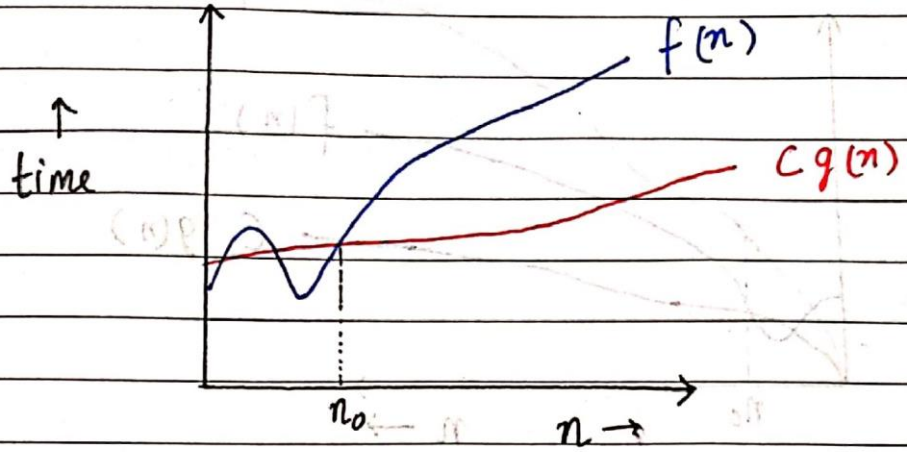
$$0 \leq c\,g(n) \leq f(n) \quad \text{for all } n \geq n_0$$

⇓ used to give lower bound on a function

if a function is $O(n^2)$ it is automatically $O(n)$ as well

URBAN EDGE

Graphic example for Big omega (Ω)



Big theta notation

Let $f(n)$ define running time of an algorithm

$f(n)$ is said to be $\Theta(gn)$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

Mathematically,

$$0 \leq f(n) \leq C_1 g(n) \quad \forall\ n \geq n_0$$

$$0 \leq C_2 g(n) \leq f(n) \quad \forall\ n \geq n_0$$

Sufficiently large value of n

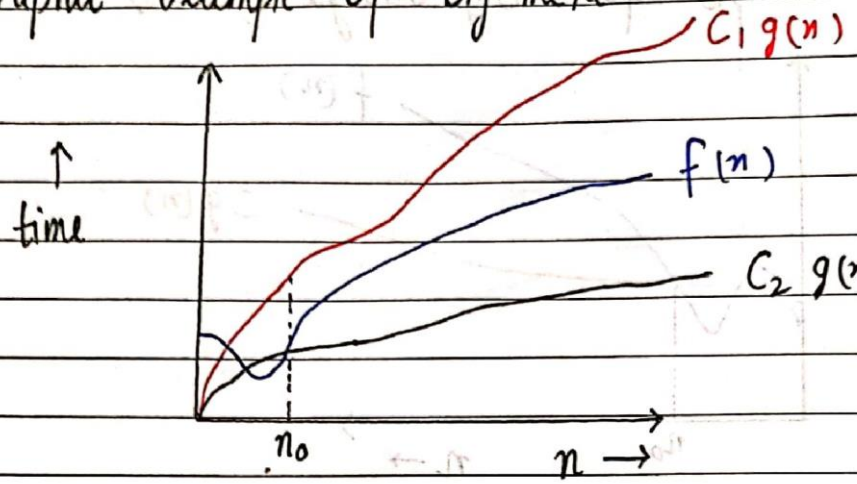
Merging both the equations, we get:

$$0 \leq C_2 g(n) \leq f(n) \leq C_1 g(n) \quad \forall\ n \geq n_0$$

The equation simply means there exist positive constants C_1 and C_2 such that $f(n)$ is sandwiched between $C_2 g(n)$ and $C_1 g(n)$

URBAN EDGE

Graphic example of Big theta



Which one of these to use ?
Since Big theta gives a better picture of runtime for a given algorithm, most of the interviewers expect you to provide an answer in terms of Big theta when they say "Order of".

Quick Quiz :- Prove that n^2+n+1 is $O(n^3)$, $\Omega(n^2)$ and $\Theta(n^2)$ using respective definitions.

Increasing order of common runtimes

$$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n^n$$

Better ⟵ ⇓ ⟶ Worse

Common runtimes from better to worse ⟶

URBAN EDGE

Best, worst and Expected Case

Sometimes we get lucky in life. Exams cancelled when you were not prepared, surprise test when you were prepared etc. ⟹ Best case

Some times we get unlucky. Questions you never prepared asked in exams, rain during sports period etc. ⇒ Worst case

But overall the life remains balance with the mixture of lucky and unlucky times. ⇒ Expected case.

Analysis of a search algorithm

Consider an array which is sorted in increasing order

1	7	18	28	50	180

We have to search a given number in this array and report whether its present in the array or not.

Algo 1 → Start from first element until an element greater than or equal to the number to be searched is found.

Algo 2 → Check whether the first or the last element is equal to the number. If not find the number between these two elements (center of the array). If the center element is greater than the number to be searched, repeat the process for first half else repeat for second half until the number is found.

Analyzing Algo 1

If we really get lucky, the first element of the array might turn out to be the element we are searching for. Hence we made just one comparison.

$$\text{Best case Complexity} = O(1)$$

If we are really unlucky, the element we are searching for might be the last one.

$$\text{Worst case Complexity} = O(n)$$

For calculating Average case time, we sum the list of all the possible case's runtime and divide it with the total number of cases.

⇓

Sometimes calculation of average case time gets very complicated

Analyzing Algo 2

If we get really lucky, the first element will be the only one which gets compared

$$\text{Best case Complexity} = O(1)$$

If we get unlucky we will have to keep dividing the array into halves until we get a single element (the array gets finished)

Worst case Complexity = $O(\log n)$

What log(n)? What is that

log(n) → Number of times you need to half the array of size n before it gets exhausted

$\log 8 = 3 \implies \frac{8}{2} \to \frac{4}{2} \to \frac{2}{2}$ → Cant break anymore.

$1 + 1 + 1$

$\log 4 = 2 \implies \frac{4}{2} \to \frac{2}{2}$ → Cant break anymore.

$1 + 1$

Log n simply means how many time I need to divide n units such that we cannot divide them (into halves) anymore.

Space Complexity

Time is not the only thing we worry about while analyzing algorithms. Space is equally important.

Creating an array of size n → O(n) space
↳ Size of input

If a function calls itself recursively n times its space complexity is O(n)

URBAN EDGE

Quick Quiz → Calculate Space Complexity of a function which calculates factorial of a given number n.

Why cant we calculate Complexity in seconds?

→ Not everyone's Computer is equally powerful

→ Asymptotic analysis is the measure of how time (runtime) grows with input

Techniques to Calculate Time Complexity

Once we are able to write the runtime in terms of size of the input (n), we can find the time complexity.

For example $T(n) = n^2 \Rightarrow O(n^2)$

$T(n) = \log n \Rightarrow O(\log n)$

Some tricks to calculate complexity

1. Drop the constants : Any thing you might think is $O(3n)$ is $O(n)$
   ↳ Better representation
2. Drop the non dominant terms : Any thing you represent as $O(n^2+n)$ can be written as $O(n^2)$
3. Consider all variables which are provided as input : $O(mn)$ & $O(mnq)$ might exist for some cases!

In most of the cases, we try to represent the runtime in terms of the input which can be more than one in number. For example :

Painting a park of dimension $m \times n \Rightarrow O(mn)$

Time Complexity – Competitive Practice Sheet

1. Fine the time complexity of the func1 function in the program show in program1.c as follows:

```c
#include <stdio.h>

void func1(int array[], int length)
{
    int sum = 0;
    int product = 1;
    for (int i = 0; i < length; i++)
    {
        sum += array[i];
    }

    for (int i = 0; i < length; i++)
    {
        product *= array[i];
    }
}

int main()
{
    int arr[] = {3, 5, 66};
    func1(arr, 3);
    return 0;
}
```

2. Fine the time complexity of the func function in the program from program2.c as follows:

```c
void func(int n)
{
    int sum = 0;
    int product = 1;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            printf("%d , %d\n", i, j);
        }
    }

}
```

3. Consider the recursive algorithm above, where the random(int n) spends one unit of time to return a random integer which is evenly distributed within the range [0,n][0,n]. If the average processing time is T(n), what is the value of T(6)?

```
int function(int n)
{
    int i;

    if (n <= 0)
    {
        return 0;
    }
    else
    {
        i = random(n - 1);
        printf("this\n");
        return function(i) + function(n - 1 - i);
    }
}
```

4. Which of the following are equivalent to O(N)? Why?
    a) O(N + P), where P < N/9
    b) 0(9N-k)
    c) O(N + 8log N)
    d) O(N + M^2)

5. The following simple code sums the values of all the nodes in a balanced binary search tree. What is its runtime?

```
int sum(Node node)
{
    if (node == NULL)
    {
        return 0;
    }
    return sum(node.left) + node.value + sum(node.right);
}
```

6. Find the complexity of the following code which tests whether a give number is prime or not?

```
int isPrime(int n){
    if (n == 1){
        return 0;
    }

    for (int i = 2; i * i < n; i++) {
        if (n % i == 0)
            return 0;
    }
```

```
    return 1;
}
```

7. What is the time complexity of the following snippet of code?

```
int isPrime(int n){

    for (int i = 2; i * i < 10000; i++) {
        if (n % i == 0)
            return 0;
    }

    return 1;
}
isPrime();
```

URBAN EDGE

Operations on an Array

Following operations are supported by an array.

Traversal
Insertion
Deletion
Search

⇒ There can be many other operations one can perform on arrays as well!
eg: Sorting asc., Sorting desc.

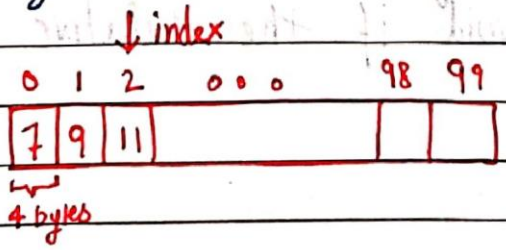
Traversal

Visiting every element of an array once → Traversal

Why traversal? → For use cases like:
→ Storing all elements → using scanf
→ Printing all elements → using printf

An important note about arrays
If we create an array of length 100 using a[100] in C language, we need not use all the elements. It is possible for a program to use just 60 elements out of these 100.
↳ But we cannot go beyond 100 elements.

An array can easily be traversed using a for loop in C language

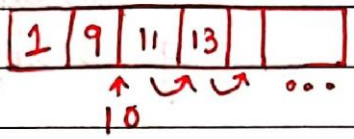


URBAN EDGE

Insertion

An element can be inserted in an array at a specified position.

In order for this operation to be successful, the array should have enough capacity

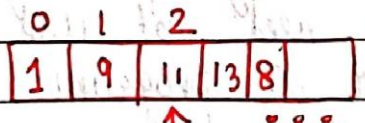


⇒ Elements need to be shifted to maintain relative order.

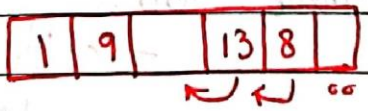
When no position is specified its best to insert the element at the end.

Deletion

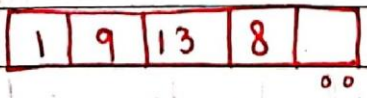
An element at specified position can be deleted creating a void which needs to be fixed by shifting all the elements to the left as follows:



Delete 11 at ind 2



Shift the elements

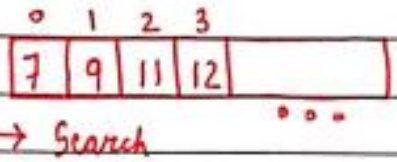


Deletion done!

We can also bring the last element of the array to fill the void if the relative ordering is not important

Searching

Searching can be done by traversing the array until the element to be searched is found



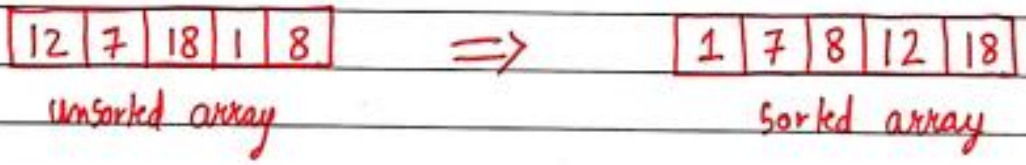
⇓

For sorted array time taken to search is much less than unsorted array!!

Sorting

Sorting means arranging an array in order (asc or desc)

We will see various sorting techniques later in the course.



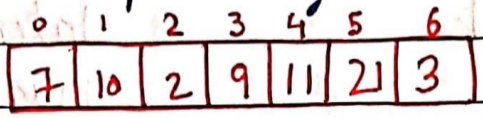
Unsorted array

Sorted array

Linear Vs Binary Search

Linear Search

Searches for an element by visiting all the elements sequentially until the element is found.



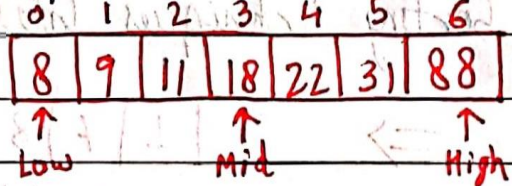
⇒ can be sorted or unsorted

Search '2' → Element found

WC Complexity : O(n)

Binary Search

Searches for an element by breaking the search space into half in a sorted array.



WC Complexity O(log n)

Search 18

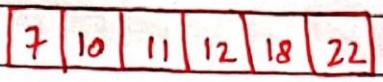
The search continues towards either side of mid based on whether the element to be searched is lesser or greater than mid.

	Linear Search	Binary Search
1.	Works on both sorted and unsorted arrays	Works only on sorted arrays
2.	Equality operations	inequality operations
3.	O(n) WC Complexity	O(log n) WC Complexity

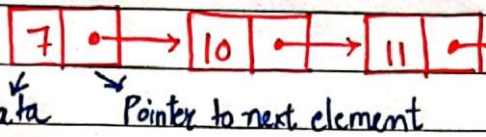
URBAN EDGE

Introduction to Linked Lists

Linked lists are similar to arrays (Linear data structures)



⇒ In Arrays elements are stored in contiguous memory locations



⇒ In linked lists, elements are stored in non contiguous memory locations

Why Linked Lists?

Memory and the capacity of an array remains fixed.
In case of linked lists, we can keep adding and removing elements without any capacity constraints

Drawbacks of Linked lists

→ Extra memory space for pointers is required (for every node 1 pointer is needed)
→ Random access not allowed as elements are not stored in contiguous memory locations.

Implementation

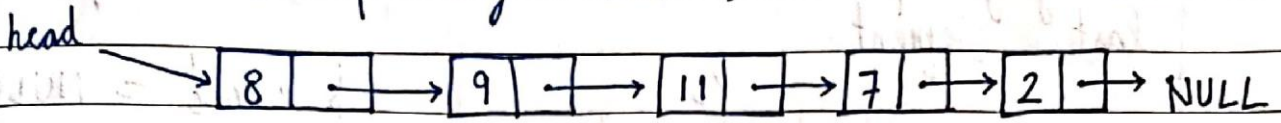
Linked list can be implemented using a structure in C language

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

⇒ Self refencing structure

Deletion in a Linked List

Consider the following Linked List

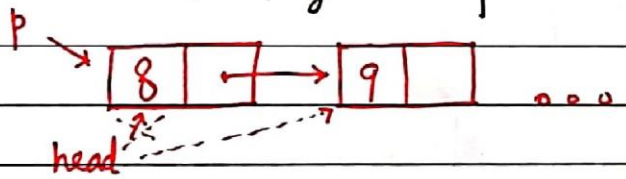


Deletion can be done for the following cases:

1. Deleting the first Node
2. Deleting the node at an index
3. Deleting the last Node
4. Deleting the first node with a given value.

The deletion just like insertion is done by rewiring the pointer connections, the only caveat being: we need to free the memory of the deleted node using free().

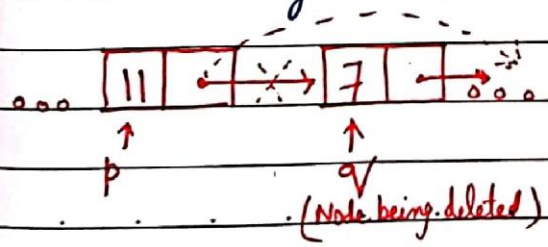
Case 1: Deleting the first node




```
Struct Node * p = head
head = head → next;
free(p);
```

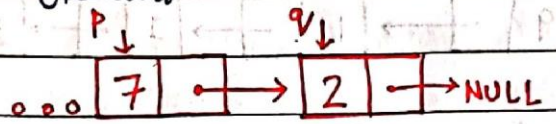
Case 2: Deleting the node at an index

for deleting a given node, we first bring a temporary pointer p before element to be deleted and q on the element being deleted



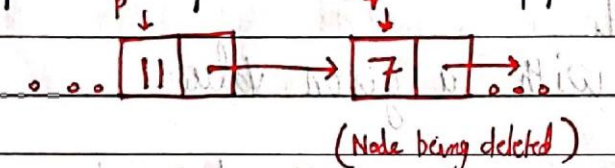

```
p → next = q → next;
free(q);
```

Case 3: Deleting the last Node
Last node can be deleted just like Case 2 by bringing p on second last element and q on last element.



p→next = NULL
free(q) → To free the memory!

Case 4: Delete the first node with a given value
This can be done exactly like Case 2 by bringing pointers p & q to appropriate positions

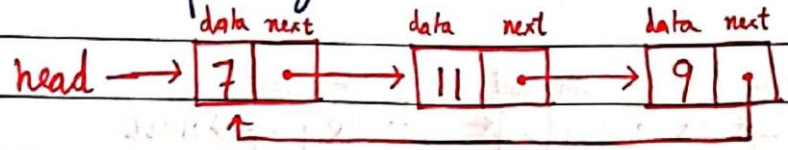


p→next = q→next;
free(q);

URBAN EDGE

Circular Linked List

A circular linked list is a linked list where the last element points to the first element (head) hence forming a circular chain



Operations on a circular linked List
operations on a circular linked lists can be performed exactly like a singly linked list.

Visit www.codewithharry.com for practice sets / code / more

URBAN EDGE

Doubly Linked List

In a doubly linked list, each node contains a data part along with the two addresses, one for the previous node and the other one for the next node

Implementation

A doubly linked list can be implemented in C language as follows:

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

Operations on a Doubly Linked List

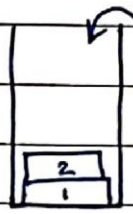
The insertion and deletion on a Doubly linked list can be performed by rewiring pointer connections just like we saw in a Singly linked list.

The difference here lies in the fact that we need to adjust two pointers (prev & next) instead of one (next) in the case of a Doubly linked list.

URBAN EDGE

Introduction to Stack Data Structure

Stack is a linear data structure. Operations on stack are performed in LIFO (Last in first out) order.



⇒ Item 2 which entered the basket last will be the first one to come out

LIFO (Last in first out)

Applications of Stack

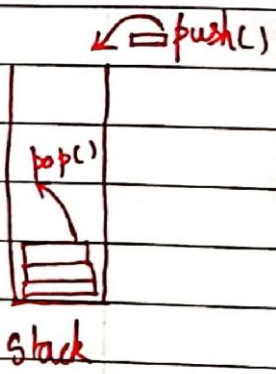
1. Used in function calls
2. Infix to postfix conversion (and other similar conversions)
3. Parenthesis matching & more...

Stack ADT

In order to create a stack we need a pointer to the topmost element along with other elements which are stored inside the stack.

Some of the operations of stack ADT are:

1. push() → push an element into the stack
2. pop() → remove the topmost element from the stack
3. peek(index) → Value at a given position is returned
4. isEmpty() / isFull() → Determine whether the stack is empty or full.



Implementation

A stack is a collection of elements with certain operations following LIFO (Last in First out) dicipline.

A Stack can be implemented using an array or a linked list

HANDWRITTEN NOTES

FREE DOWNLOAD

Data Structure & Algorithms

In C Programming

WWW.LEARNLONER.COM

Data Structures and Algorithms.

Beginner to Advanced Guide.

INDEX

LearnLoner.com

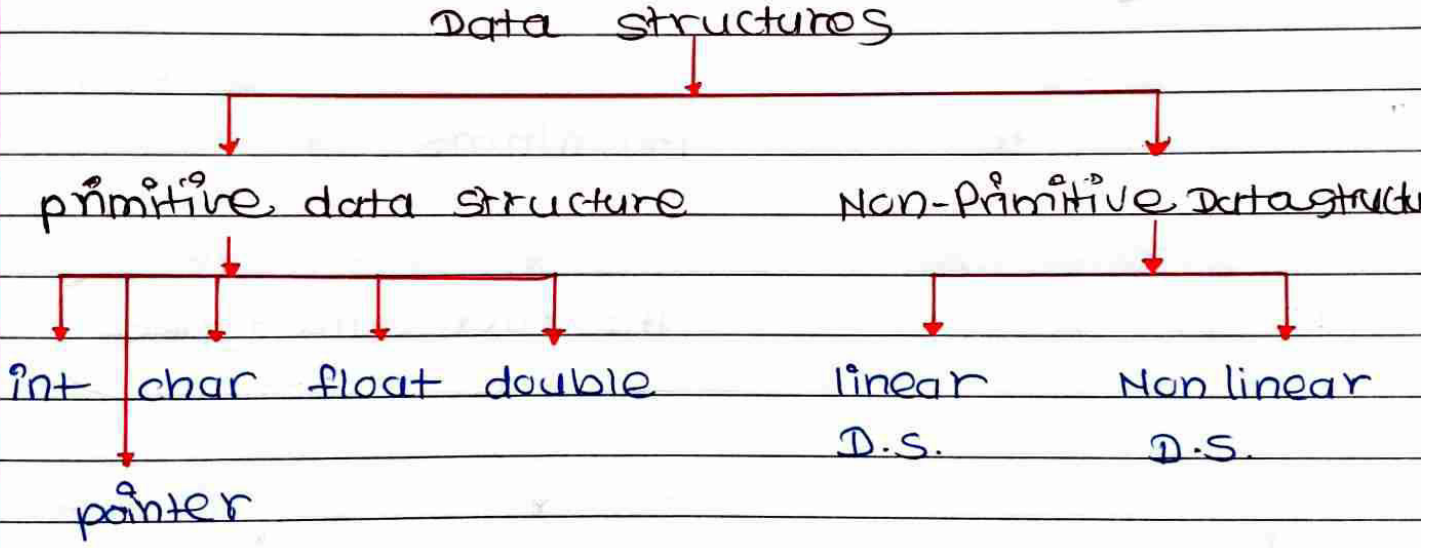
LearnLoner.com

What is Data Structure ?

→ Data structure is a way to store and organize data so that it can be used efficiently.

As per name indicates itself that organizing the data in memory.

The data structure is not any programming language like c, c++, Java etc. It is set of algorithms that we can use in any programming language to structure data in memory.



Linear Data structure :-

The arrangement of data in the sequential manner is known as linear data structure. The data structure used for this purpose are Arrays, linked list, Stacks and queues.

In this data structures, one element is connected to only one another element in a

LearnLoner.com

linear form.

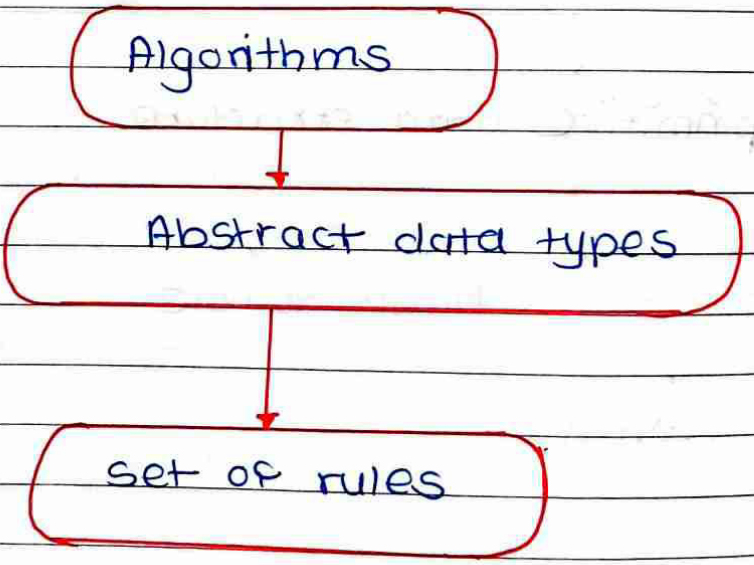
Non-linear data structure :-

When one element is connected to the 'n' number of elements known as non-linear data structures.

Example :- trees and graphs.

In this case, elements are arranged in a random manner.

Algorithms and Abstract Data types ??



Why →

To structure the data in memory, 'n' number of algorithms are proposed, and all these algorithms are known as Abstract Data Types.

LearnLoner.com

An Abstract Data Type tells what is to be done and data structure tells how is to be done ?

ADT gives us the blueprint while data structure provides the implementation part.

What is Data ?

Data can be defined as the elementary value / collection of values.
forexample :- student's name and its id are the data about student.

What is Record ?

Record can be defined as collection of various data items
example :- student entity → name, address, course and marks can be grouped together to form record.

What is File ?

File is a collection of various records of one type of entity
example :- if there are 60 employees in class, then there will be 20 records in related file where record contains info of employee

What is Attribute and Entity ?

An entity represents class of certain objects. it contains various attributes. each attribute represents particular property of that entity.

LearnLoner.com

What is need of data structures?

As applications are getting complexed and amount of data is increasing day by day, there may arise following problems :-

Processor speed :- As data is growing day by day to the billions of files per entity, processor may fail to deal with that amount of data.

Data Structure :- consider an inventory size of 106 items in store, if our application needs to search for a particular item, it needs to transverse 106 items every time, results in slowing down process.

multiple requests :- If thousands of users are searching data simultaneously on a web server, then there are chances that to be failed to search during that process.

To solve this problems, data structures are used. Data is organized to form a data structure in a such way that all items are not required to be searched and require data can be searched instantly.

Advantages of data Structure :-

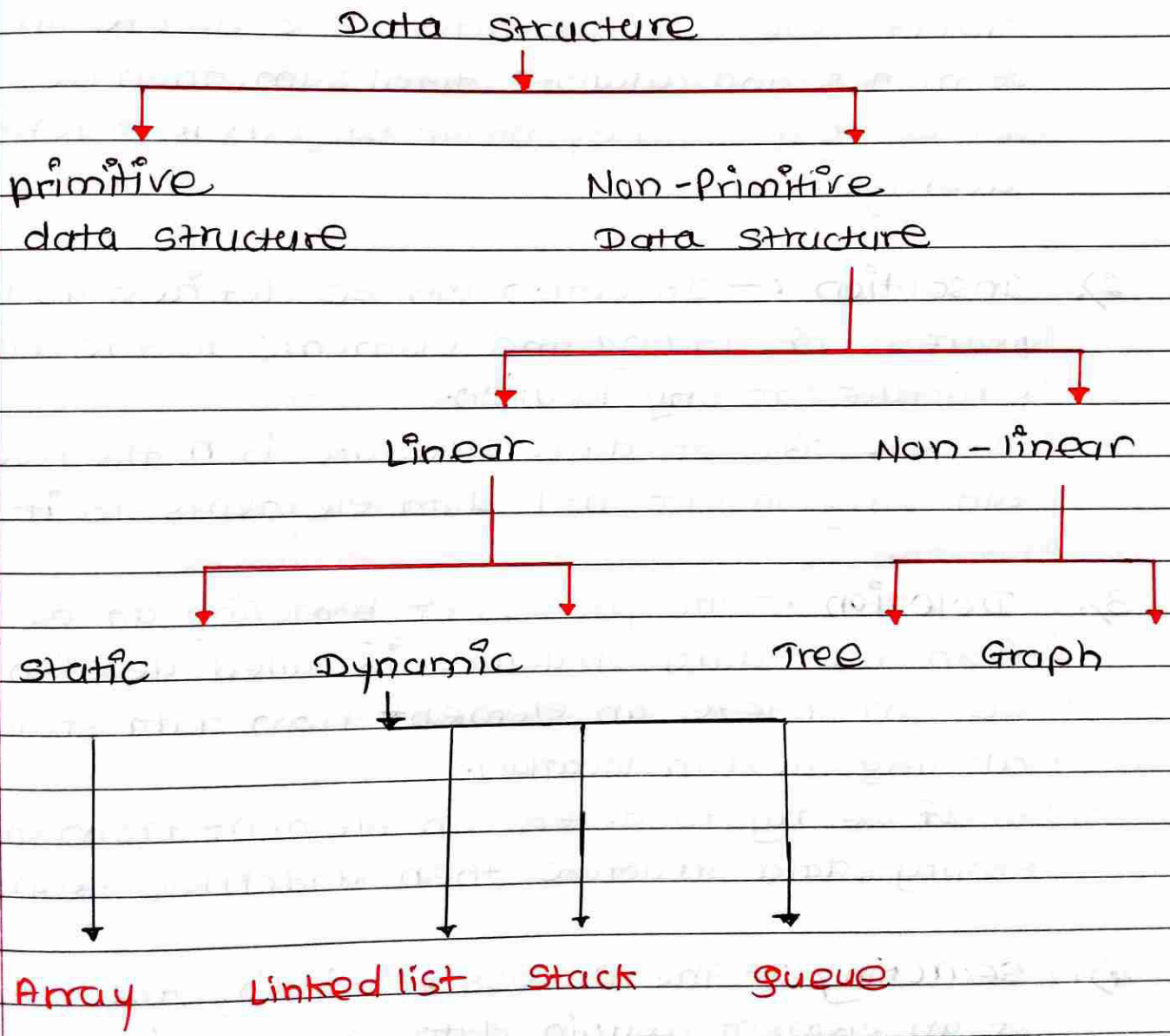
Efficiency :- If the choice of a data structure for implementing a particular ADT is proper, it makes program very efficient in terms of time and space.

Reusability :- The data structure provides reusability means that multiple client programs can use the data structure.

LearnLoner.com

Abstraction :- The data structure specified by the ADT also provides level of abstraction. The client cannot see internal working of data structure, so it does not have to worry about implementation.

* Data structure classification :-



LearnLoner.com

Operations on data structure :-

1). Traversing :- Every data structure contains a set of data elements. Traversing data structure means visiting each element of data structure in order to perform some specific operation like searching or sorting.

Example :- If we need to calculate average of marks obtained by a student in 6 different subject, we need to traverse complete array of marks and calculate total sum, then we will devide that sum by no. of subjects ie. 6 to find average.

2). Insertion :- Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is n then we can only insert n-1 data elements to it.

3). Deletion :- The process of removing an element from the data structure is called deletion. We can delete an element from data structure at any random location.

If we try to delete an element from an empty data structure then underflow occurs.

4). Searching :- The process of finding the location of an element within data structure is called searching. There are two algorithms to perform

LearnLoner.com

searching, linear search and Binary search.

5). Sorting :- The process of arranging the data structure in a specific order is called as sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort etc.

6). merging :- When two lists list A and list B of size M and N respectively, of similar type of elements, clubbed or joined to produce third list, list C of size (M+N), then this process is called merging.

DATA STRUCTURES AND ALGORITHM

What is Algorithm ?

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer.

It is not complete program or code ; it is just a solution (logic) of a problem, which can be represented either as an informal description using a flowchart or pseudocode.

characteristics of an algorithm.

Input :- An algorithm has some input values. We can pass 0 or some input value to an algorithm.

LearnLoner.com

Output :- We will get 1 | more output at end of an algorithm.

Unambiguity :- An algorithm should be unambiguous which means that instruction in an algorithm should be clear and simple.

Finiteness :- An algorithm should have finiteness. means limited number of instructions.

Effectiveness :- An algorithm should have finite as each instruction in an algorithm affects the overall process.

Approches in Algorithm :-

1). Brute force Algorithm :- The general logic structure is applied to design an algorithm. It is also known as exhaustive search algorithm that searches all possible to provide required solution.

such algorithms have two types :-

1). optimizing
finding all solutions of a problem and then take out the best solution is known then it will terminate if the best solution is known.

2). sacrificing
As soon as the best solution is found, then it will stop.

LearnLoner.com

Divide and conquer :- This breaks down the algorithm to solve the problem in different methods. It allows you to break down problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.

Greedy algorithm :- It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting best solution. It is easy to implement and has faster execution time. But there are very rare cases in which it provides the optimal solution.

The major categories of algorithms are given below:
Sort :- Algorithm developed for sorting the items in a certain order.

search :- Algorithm developed for searching the items inside a data structure.

Delete :- Algorithm developed for deleting the existing element from the data structure.

Insert :- Algorithm developed for inserting an item inside a data structure.

Update :- Algorithm developed for updating the existing element inside a data structure.

LearnLoner.com

Algorithm Analysis :-

The algorithm can be analyzed in two levels ie first is before creating the algorithm, and second is after creating the algorithm.
There are two analysis of an algorithm.

Priori Analysis :-

Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm.

Posterion Analysis :-

Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing algorithm using any programming language.

Algorithm complexity :-

The performance of the algorithm can be measured in two factors :

Time complexity :-

The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation.

Here big O notation is the asymptotic notation to represent time complexity. The time complexity is mainly calculated by counting the number of steps to finish execution.

LearnLoner.com

```
sum = 0 ;
// suppose we have to calculate the sum of n
 numbers.
for i=1 to n
sum = sum + i ;
// when the loop ends then sum holds the sum
of n numbers.
return sum;
```

In above code, the time complexity of the loop statement will be atleast n, and if value of n increases, then time complexity also increases.

We generally consider the worst-time complexity as it is maximum time taken for any given input size.

Space complexity :—

An algorithm's space complexity is the amount of space required to solve a problem and produce an output. similar to the time complexity, space complexity is also expressed in big o notation.

Space complexity = Auxiliary space + Input size.

LearnLoner.com

The following are the types of algorithms:

Search Algorithm :-

on each day, we search for something in our day to day life.

Similarly, with the case of computer, huge data is stored in a computer that whenever user asks for any data then the computer searches for that data in the memory and provides that data to the user. There are mainly two techniques available to search data in an array:

- Linear search
- Binary search

Sorting Algorithms :-

Sorting algorithms are used to rearrange elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements:

LearnLoner.com

Asymptotic Analysis :-

The time required by an algorithm comes under three types :

Worst case :- It defines the input for which the algorithm takes a huge time.

Average case :- It takes average time for the program execution.

Best case :- It defines the input for which the algorithm takes the lowest time.

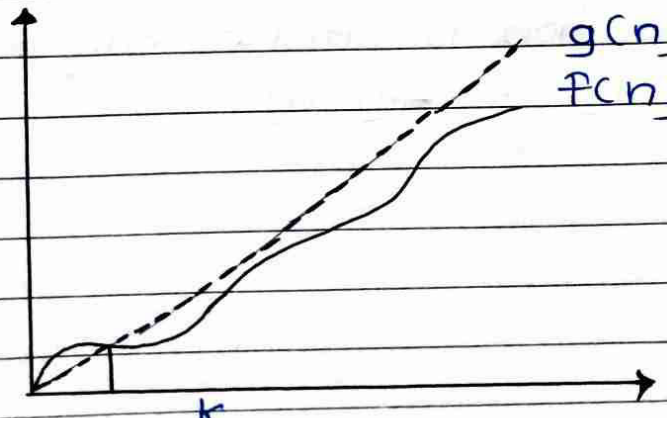
Asymptotic Notations :-

The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below :

1. Big oh notation (O) :-

This measures the performance of an algorithm by simply providing the order of growth of the function.

This notation provides an upper bound on a function which ensures that function never grows faster than the upper bound.



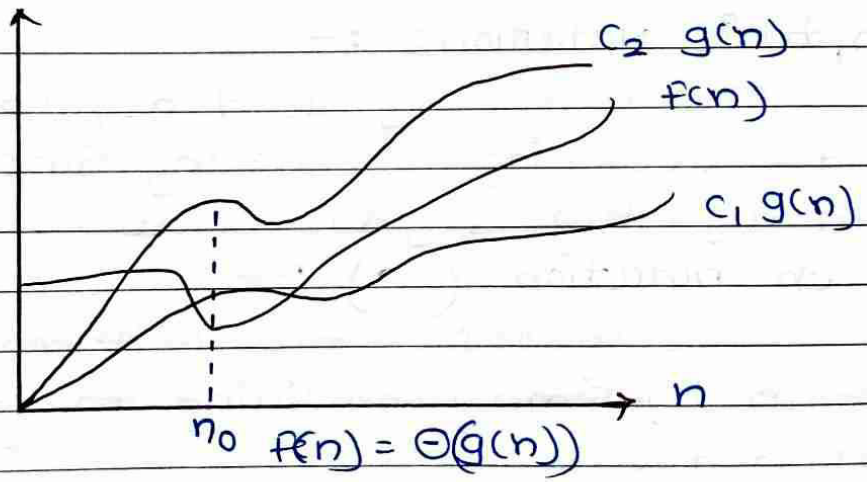
LearnLoner.com

Example :- If f(n) and g(n) are two functions defined for positive integer,
then $f(n) = O(g(n))$ as f(n) is big oh of g(n) or f(n) is on order of g(n)) if there exists constants c and n_0 such that:

$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

2). Omega Notation (Ω) :-

It basically describes best case scenario which is opposite to big o notation. It is the formal way to represent lower bound of an algorithm's running time.



Example :- let f(n) and g(n) be functions of n where n is steps required to execute program:

$f(n) = \Omega(g(n))$

The above condition is satisfied only if when:

$c1 \cdot g(n) <= f(n) <= c2 \cdot g(n)$

LearnLoner.com

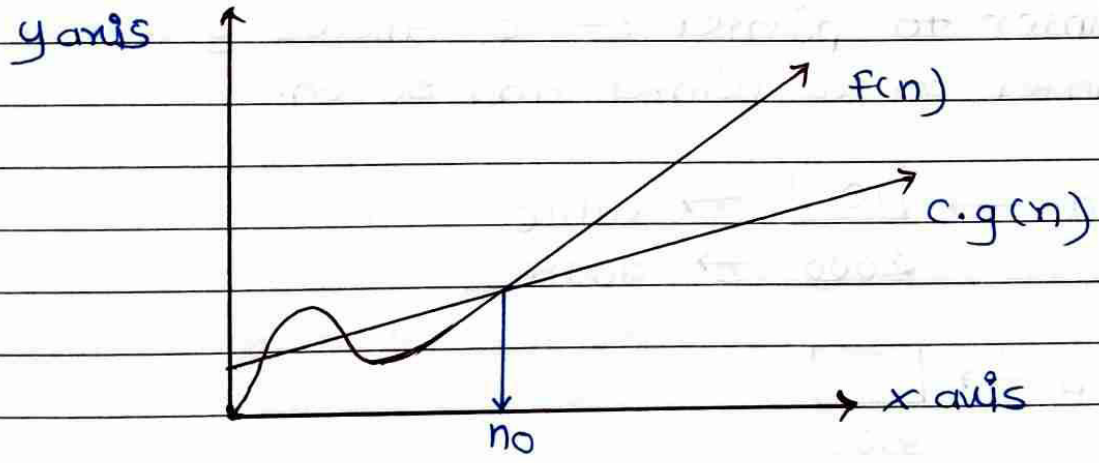
2). omega Notation (Ω)

It basically describes best-case scenario which is opposite to big-O notation It is formal way to represent lower bound to an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or best case time complexity.

Example :- If f(n) and g(n) are two functions defined for positive integers,

then $f(n) = \Omega g(n)$ as f(n) is omega of g(n) or f(n) is on the order of g(n) if there exists constants c and no such that:

$f(n) >= c \cdot g(n)$ for all $n \geq n_0$ and $c > 0$



3). Theta Notation (Θ)

The theta notation mainly describes average case scenarios.

It represents realistic time complexity of an algorithm. Big theta is mainly used when the value of worst case and best case is same.

LearnLoner.com

Pointer :-

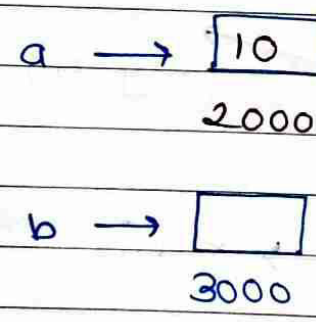
Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at location is known as dereferencing pointer.

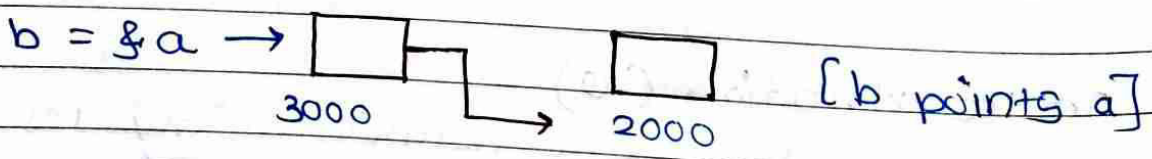
Pointer arithmatic :-

4 arithmatic operaters that can be used in pointers : ++, --, +, -

Array of pointers :- You can define array of to hold a number of pointers.

Pointer to pointer :- c allows you to have pointer on a pointer and so on.





Program
Pointer →

```
#include <stdio.h>
int main()
```

LearnLoner.com

```
{
int a = 5;
int *b;
b = &a;
printf ("value of a = %d \n", a);
printf ("value of a = %d \n", *(&a));
printf ("value of a = %d \n", *b);
printf ("address of a = %u \n", &a);
printf ("address of a = %d \n", b);
printf ("address of b = %u \n", &b)
printf ("value of b = address of a = %u ", b);
return 0;
}
```

output

```
value of a = 5
value of a = 5
address of a = 3010491492
address of a = -1284473004
address of b = 3010491496
value of b = address of a = 3010491492.
```

Program :—
Pointer to pointer :—

```
#include <stdio.h>
int main ()
{
int a = 5;
int *b;
int **c;
```

LearnLoner.com

```
b = &a;
c = &b;
printf ("value of a = %d \n", a);
printf ("value of b = address of a = %u \n", b);
printf ("value of c = address of b = %u \n", c);
printf ("address of b = %u \n", c);
printf ("address of c = %u \n", &c);
return 0;
}
```

output →

```
value of a = 5
value of b = address of a = 2831685116
value of c = address of b = 2831685120
address of b = 2831685120
address of c = 2831685128.
```

Structure :-

A structure is a composite data type that defines a grouped list of variables that are to be placed under one name in block of memory.

Program :-

structure →

```
struct structure_name
{
  data-type member 1;
  data-type member 2;
```

LearnLoner.com

:
:
:

data-type member ;
};

Advantages of structure :-
- It can hold variables of different data types.
- We can create objects containing different types of attributes.
- It allows us to re-use the data layout across programs.
- It is used to implement other data structure like linked list, queues, trees and graphs.

Program :-
how to use structure in program →

```
#include <stdio.h>
#include <conio.h>
Void main ()
{
struct employee
{
int id ;
float salary ;
int mobile ;
};
```

LearnLoner.com

```
struct employee e1, e2, e3;
printf ("\n Enter ids, salary & mobile no. \n");
scanf ("%d %f %d", &e1.id, &e1.salary, &e1.mobile);
scanf ("%d %f %d ", &e2.id, &e2.salary, &e2.mobile);
printf (%d %f %d", &e3.id, &e3.salary, &e3.mobile);
printf ("\n Entered result ");
printf ("\n %d %f %d", e1.id, e1.salary, e1.mobile);
printf ("\n %d %f %d ", e2.id, e2.salary, e2.mobile);
printf ("\n %d %f %d ", e3.id, e3.salary, e3.mobile);
getch ();
}
```

output →

guess the output

And write it here....

LearnLoner.com

Array :- Arrays are defined as collection of similar type of data items stored at contigous memory locations.

Array is the simplest data structure where each data element can be randomly accessed by using its index number.

Array declaration :-

```
int arr[10] ; char arr[10] ; float arr[5]
```

Program without Array :-

```
#include <stdio.h>
void main ()
{
 int marks-1 = 56 ; marks-2 = 78, marks-3 = 89 ;
 float avg = (marks-1 + marks-2 + marks-3)/3 ;
 print (avg) ;
}
```

Program by using Array :-

```
#include <stdio.h>
void main
 {
 int marks [3] = { 56, 78, 89} ;
 int i ;
 float avg ;
 for (i=0 ; i<3 ; i++)
```

LearnLoner.com

```
    {
    avg = avg + marks[i];
    }
  printf(avg);
}
```

Complexity of Array operations :-

1). Time complexity :-

Algorithm	Average case	worst case
Access	O(1)	O(1)
search	O(n)	O(n)
insertion	O(n)	O(n)
Deletion	O(n)	O(n)

2). Space complexity :-

In Array space complexity for worst case is O(n)

Memory Allocation of the Array :-

Each element in Array represented by indexing Indexing of array can be defined in three ways:

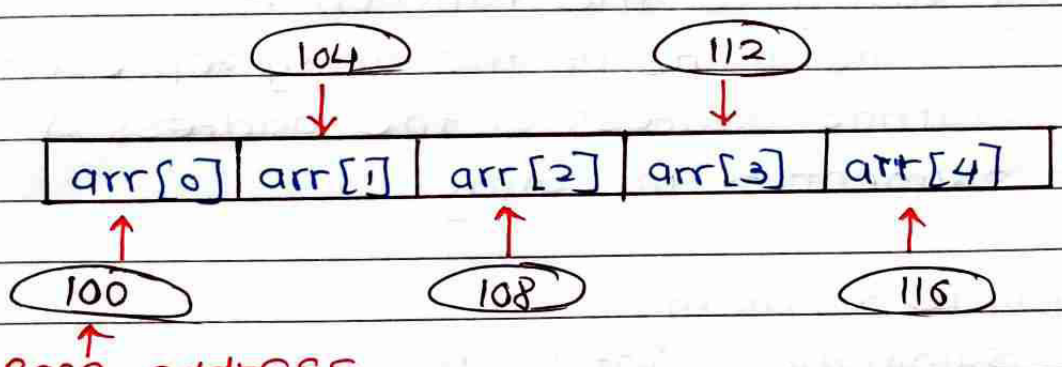
1. 0 (Zero Based indexing) :-

The first element of the array will be arr[0].

LearnLoner.com

2. 1 (one-based indexing) :-
The first element of array will be arr[1].

3. n (n-based indexing) :-
The first element of array can reside at any random index number.



(fig : int arr[5])

Accessing elements of an Array :-
To access any random element of an array we need the following information :
1. Base address of the array
2. size of an element in bytes.
3. which type of indexing, array follows.

Address of any element of 1D array can be calculate

Byte address of element A[i] = base address + size * (first - index)

Example: In an array, A[-10 .... +2] Base address (BA) = ggg, size of an element = 2 bytes, find location of A[-1].

LearnLoner.com

Solution: $L(A[-1]) = ggg + [(-1) - (-10)] \times 2$.

$= ggg + 18$

$= 1017$.

$\therefore$ location of $A[-1] = 1017$.

Passing array to the function :-

The name of the array represents the starting address or the address of the first element of the array.

Program:

```
#include <stdio.h>
int summation (int []);
void main ()
{
 int arr [5] = {0,1,2,3,4};
 int sum = summation (arr);
 printf ("%d ", sum);
}
int summation (int arr[])
 {
  int sum =0, i;
  for(i=0 ;i<5 ;i++)
    {
     sum = sum + arr [i];
    }
  return sum ;
 }
```

LearnLoner.com

2D Array :- 2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as collection of rows and columns.

How to declare 2D Array :-

The syntax for declaration of two dimensional array is as follows :

```
int arr[max-rows][max-columns];
```

However, it produces the data structure which looks like following :

	0	1	2	...	n-1
0	a[0][0]	a[0][1]	a[0][2]	.....	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	....	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	...	a[2][n-1]
:	:	:	:	...	:
n-1	a[n-1][0]	a[n-1][1]	a[n-1][2]		a[n-1][n-1]

a[n][n]

(Fig : a[n][n])

How to access data in 2D-array :-

Due to fact that elements of 2D arrays can be random accessed.

LearnLoner.com

```
int x = a[i][j];
```

where i, j are the rows and columns respectively.

Initializing 2D arrays :-

The syntax to declare and initialize the 2D array is given as follows :

```
int arr[2][2] = {0,1,2,3};
```

number of elements in 2D arrays
= number of rows * number of columns.

Mapping 2D array to 1 D array :-

The size of a two dimensional array is equal to the multiplication of number of rows and number of columns present in the array.

A 3x3 two dimensional array is as shown:-

	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

column index

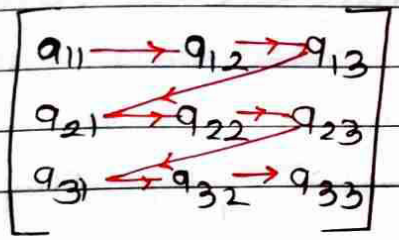
row index

There are two main techniques of storing 2D array elements into memory.

LearnLoner.com

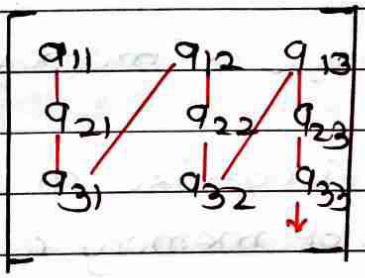
1. Row major ordering :—

In row major ordering, all the rows of 2D array are stored into memory contiguously.



2. Column major ordering :—

According to column major ordering, all the columns of 2D array are stored into the memory contigously.



Calculating address of random element of a 2D array:—

1). By row major order :—

If array is declared a[m][n] where m is the number of rows while n is number of columns. then address of an element a[i][j] is calculated as,

Address (a[i][j]) = B.A + (i * n + j) * size

B.A → Base Address

2). By column major order :—

Address (a[i][i] = (j * m) + i) * size + B.A.

LearnLoner.com

Linked list :-

✓ Why there is a need of linked list ?

If we declare an array of size 3. As we know that all the values of an array are stored in a continous manner, so all three values of an array are stored in a sequential fashion.

Then, total memory space occupied by array would be $3 * 4 = 12$ bytes.

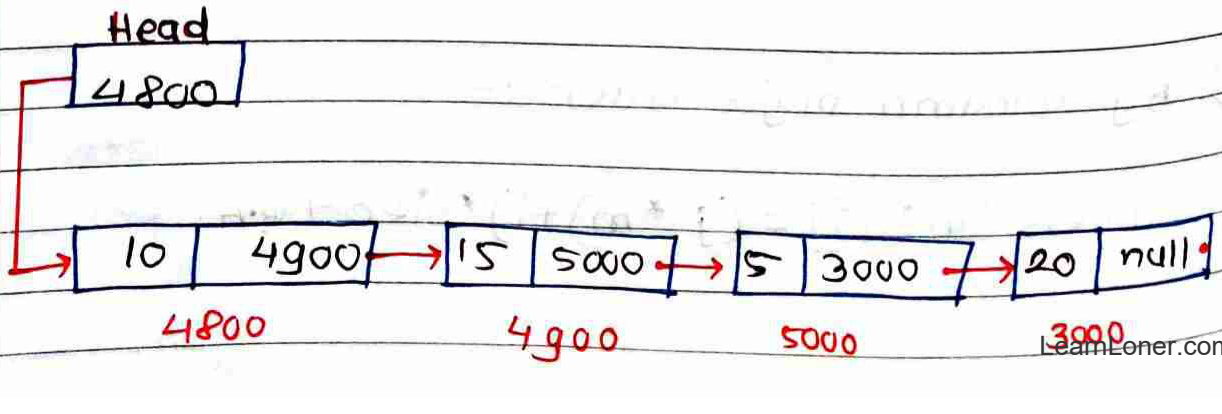
Drawbacks of using array :-

- we cannot insert more than 3 elements in above example because only 3 spaces are allocated by 3 elements.
- In case of array, the wastage of memory can occur.
- In array, we are providing fixed-size at compile time, due to which wastage of memory occurs.

The solution to this problem is to use linked list.

What is Linked list ?

A linked list is also a collection of elements, but the elements are not stored in a consecutive location. or linked list is a collection of the nodes in which one node is connected to another node and node consists of two parts i.e. one is data part and second one is the address part.



LearnLoner.com

declaration of linked list :-

In linked list, one is variable and second one is pointer variable. we can declare linked list by using user-defined data type called as structure.

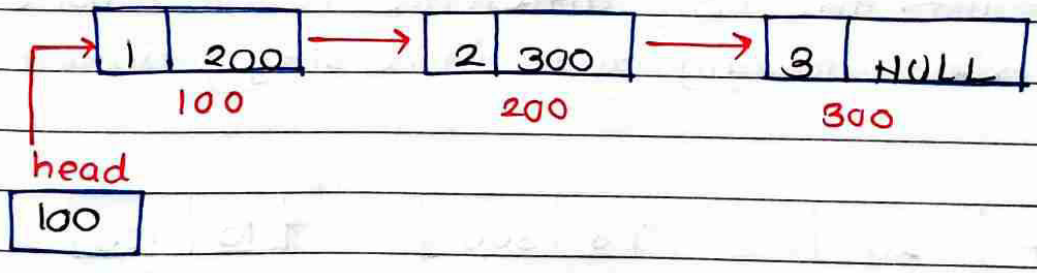
```
struct node
{
  int data;
  struct node *next;
}
```

Types of linked list :-

1). singly linked list :-

The singly linked list is most common. which consists of data part and address part. The address part in the node is known as a pointer.

Example :- suppose we have three nodes and addresses of these three nodes are 100, 200 and 300 :

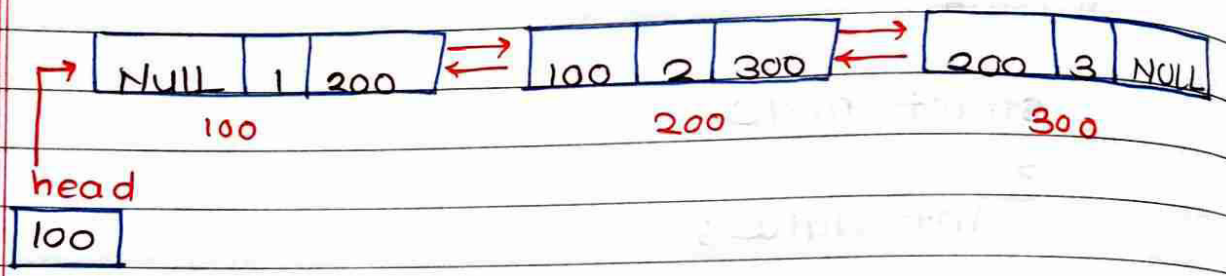


NULL means its address part does not point to any node. The pointer that holds the address of the initial node is known as a head pointer.

LearnLoner.com

2. Doubly linked list :-

As name suggests, the doubly linked list contains two pointers. We define it in three parts the data part and the two address part.

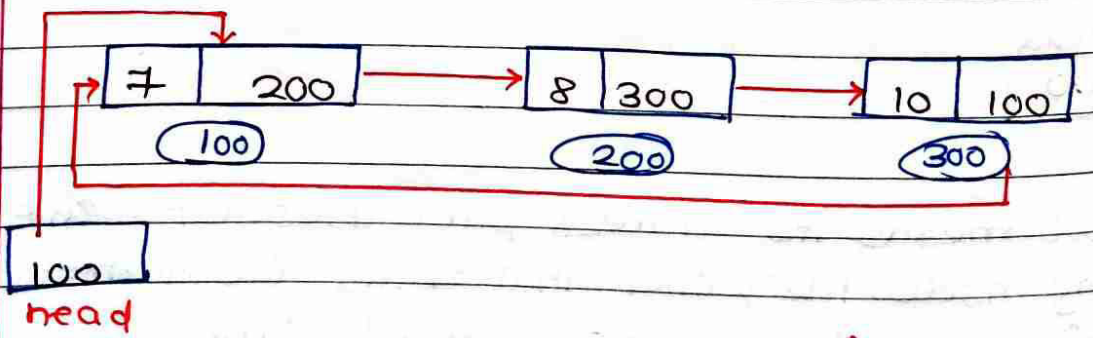


Representation of doubly linked list :-

```
struct node
 {
  int data ;
  struct node * next ;
  struct node * prev;
 }
```

3. Circular linked list :-

A circular linked list is a variation of a singly linked list. The only difference is "last node does not point to any node in a singly linked list".



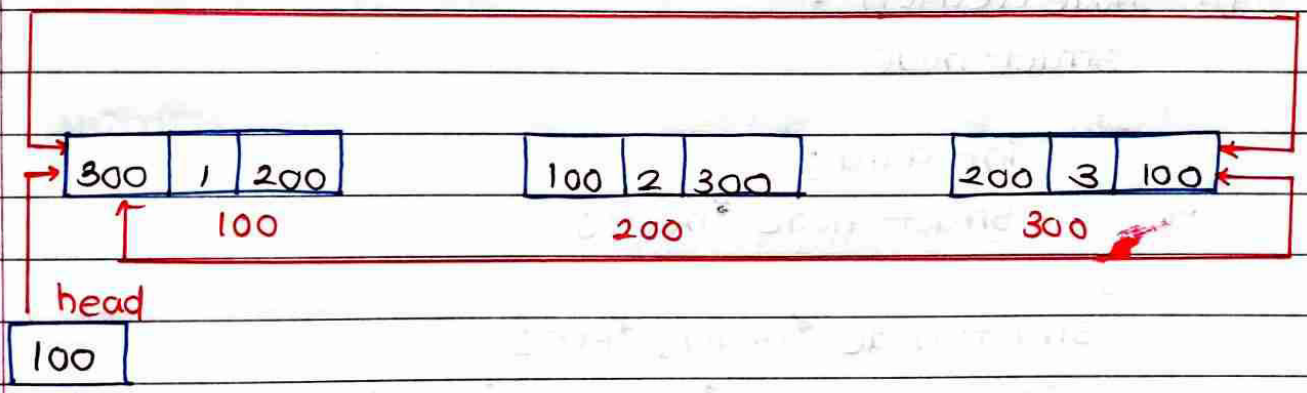
LearnLoner.com

Representation of circular linked list :-

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

4. Doubly circular linked list :-

The doubly circular linked list has the features of both the circular linked list and doubly linked list.



The last node is attached to the first node and thus creates a circle.
The main difference is that doubly circular linked list does not contain NULL value in previous field of the node.

Representation of doubly circular linked list :-

```
struct node
{
  int data;
  struct node *next;
  struct node *prev;
}
```

LearnLoner.com

Complexity :-

	Average				Space complexity
	Access	search	Insertion	deletion	Worst
Singly linked list	O(n)	O(n)	O(1)	O(1)	O(n)

	Worst			
	Access	search	Insertion	deletion
singly linked list	O(n)	O(n)	O(1)	O(1)

Operations on singly linked list :-

1). Node creation :-

```
struct node
{
    int data;
    struct node *next;
};
struct node *head, *ptr;
ptr = (struct node*) malloc (sizeof (struct node*)
```

2). Insertion :-

① Insertion at beginning :- It involves inserting any element at the front of the list. We just need a few link adjustment to make new node as head of list.

② Insertion at end of list :- The new node can be inserted as the only node in the list / it can be inserted as last one.

③ Insertion after specified node :- we need to skip desired number of nodes in order to reach node after which the new node will be inserted.

learnl.oher.com

3). 3) Deletion and Traversing :-

① Deletion at beginning :- It just needs few adjustments in the node pointers

② Deletion at end of list :- The list can either be empty or full. Different logic is implemented for different scenario's.

Traversing :- In traversing, we simply visit each node of the list at least once in order to perform some specific operation in it, for example, printing data part of each node present in the list.

Searching :- In searching, we match each element of the list with the given element. If the element is found on any of the location of that element is returned otherwise null is returned.

operations on doubly linked list :-

1) Node creation :-

```
struct node
{
  struct node *prev;
  int data;
  struct node *next;
};
struct node *head;
```

2). Insertion :-

① Insertion at beginning :- Adding the node into the linked list at beginning.

② Insertion at end :- Adding the node into the linked list to the end.

LearnLoner.com

3). Deletion and Traversing :-

① Deletion at beginning :- Removing the node from beginning of the list

② Deletion at end :- Removing the node from end of the list.

Traversing :- visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display etc.

Searching :- comparing each node data with the item to be searched and return location of the item in the list if the item found else return null.

Skip list :-

* What is a skip list ?

A skip list is a probalistic data structure. The skip list is used to store a linked list of elements or data with a linked list. In one single step, it skips several elements of the entire list, which is why it is known as skip list.

Structure of skip list :-

skip list is built in two layers : The lowest layer and the top layer. The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are the like an "express line" where elements are skipped.

LearnLoner.com

complexity table :-

Sr.No.	complexity	Average case	Worst case
1).	Access complexity	O(log n)	O(n)
2).	search comple.	O(log n)	O(n)
3).	delete comple.	O(log n)	O(n)
4).	Insert comple.	O(log n)	O(n)
5).	space comple.	-	O(n log n).

Basic operations and its algorithms :-

1). Insertion operation :- It is used to add new node to a particular location in a specific situation.

2). Deletion operation :- It is used to delete a node in a specific situation.

3). search operation :- The search operation is used to search a particular node in a skip list.

Algorithm of insertion operation :-

```
Insertion (L, key)
 local update [0 .... max - level + 1]
 q = L → header
 for i = L → level down to 0 do.
      while q → forward [i] → key forward [i]
 update [i] = q
```

LearnLoner.com

```
a = a → forward[0]
lvl = random_level()
if lvl > L → level then
for i = L → level + 1 to lvl do
    update[i] = L → header
    L → level = lvl
a = make node (lvl, key, value)
for i = 0 to level do
    a → forward[i] = update[i] → forward[i]
update[i] → forward[i] = a
```

Algorithm of deletion operation :-

```
Deletion (L, key)
local update[0 .... max level + 1]
a = L → header
for i = L → level down 0 to do
    while a → forward[i] → key forward[i]
        update[i] = a
a = a → forward[0]
if a → key = key then
    for i = 0 to L → level do
    if update[i] → forward[i] ≠ a then break
    update[i] → forward[i] → forward[i]
free(a)
while L → level > 0 and L → header → forward[L → level]
                                    = NIL do
L → level = L → level - 1.
```

LearnLoner.com

Algorithm of searching operation :-

searching (L, skey)

q = L → header

loop invariant : q → key level down to 0 do.

while q → forward [i] → key < forward [i]

q = q → forward [q]

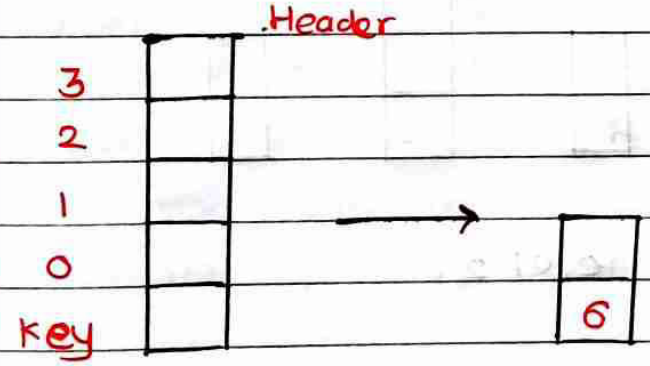
if q → key = skey then return q → value

else return failure.

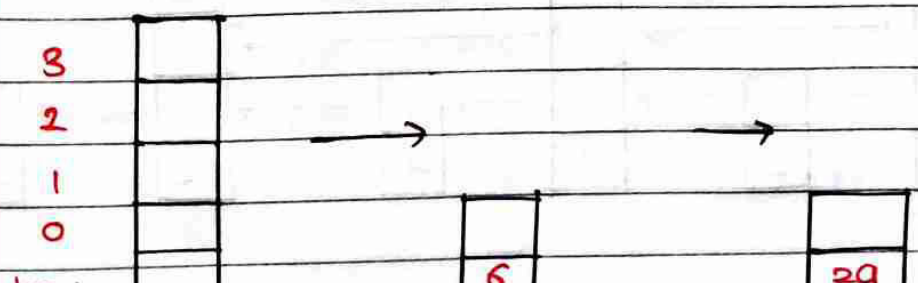
Example : create a skip list, we want to insert these following keys in empty skip list.

1. 6 with level 1
2. 29 with level 1
3. 22 with level 4.
4. 9 with level 3.
5. 17 with level 1.
6. 4 with level 2.

→ solution :- Insert 6 with level 1.

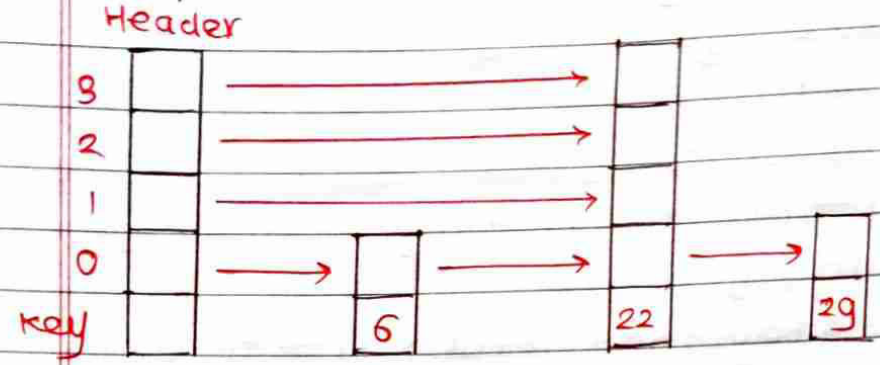


step 2 :- Insert 29 with level 1.

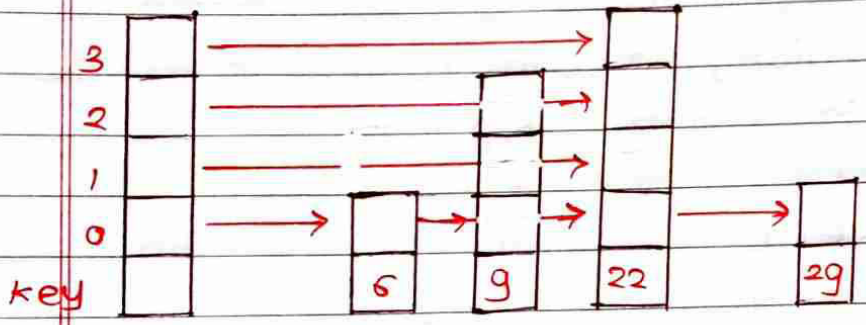


LearnLoner.com

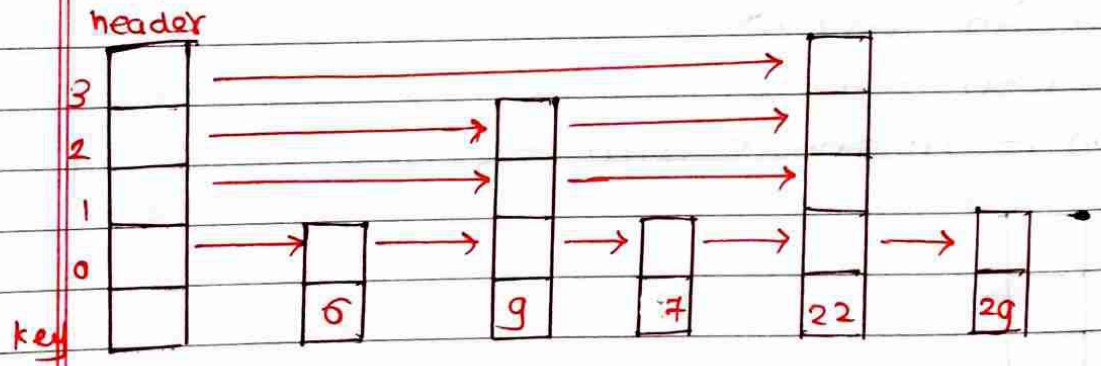
Step 3 : Insert 22 with level 4.



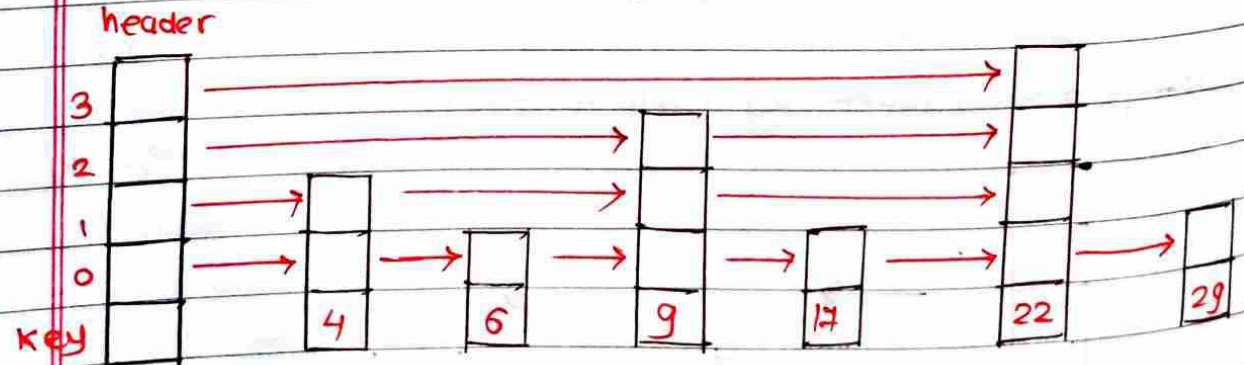
Step 4: Insert 9 with level 3.



Step 5: Insert 17 with level 1



Step 6: Insert 4 with level 2.



LearnLoner.com

Stack :- A stack is a linear data structure that follows LIFO (Last-In-First-Out) principle. Stack has one end, whereas queue has two ends (front and rear).

A stack is a container in which insertion and deletion can be done from the end (one) known as the top of the stack.

A stack is an Abstract Data Type with a pre-defined capacity, which means that it can store elements of limited size.

Operations on the stack :-

1). push ( ) :- When we insert an element in a stack then the operation is known as push. If stack is full overflow condition occurs.

2). pop ( ) :- when we delete an element from stack, the operation is called as pop ( ). If stack is empty means no element exists in the stack, this state is known as an underflow state.

3). peek ( ) :- It returns the element at a given position.

4). count ( ) :- It returns the total number of elements available in a stack.

5). change ( ) :- It changes the element at the given position.

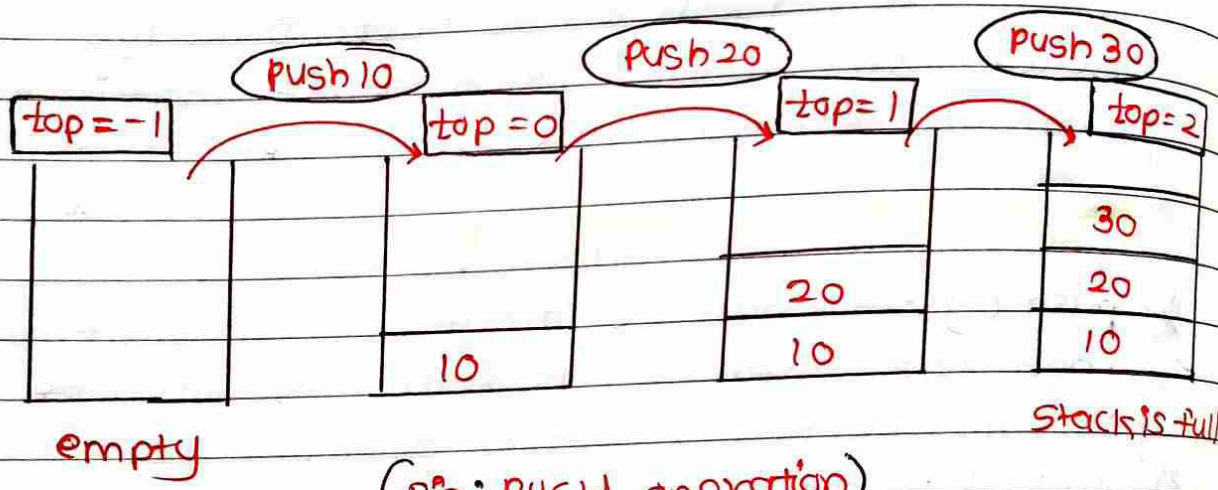
6). display ( ) :- It prints all the elements available in the stacks.

PUSH operation :-

Steps — Before inserting an element in the a stack, we check whether the stack is full.

LearnLoner.com

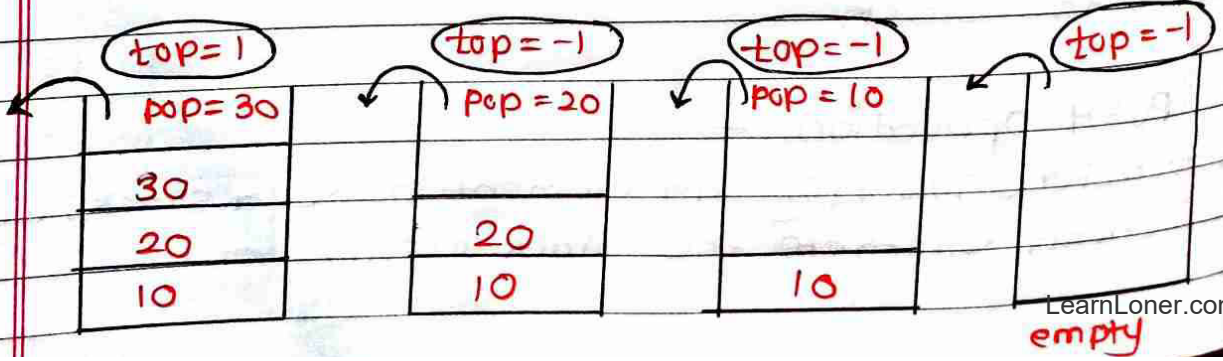
- If we try to insert element in a stack, and the stack is full, then overflow condition occurs.
- when we initialised a stack, we set the value of top as -1 to check that stack is empty
- The elements will be inserted until we reach the max size of the stack. top = top + 1.



(fig: PUSH operation)

POP Operation :-

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from empty stack, then underflow condition occurs.
- first access the element which is pointed by top.
- once the top operation is performed, top is decremented by 1 i.e. top = top - 1.



LearnLoner.com

Applications of stack :-

1) Recursion :- The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all previous records of function are maintained.

2) DFS (Depth first search) :- This search is implemented on a graph, graph uses stack d.s.

3) Backtracking :- If we have to create a path to solve maze problem, If we are moving in particular path and we realise that we come on the wrongway. In order to come at beginning of the path to create a new path, we use stack d.s.

4) Memory management :- The stack manages the memory. The memory is assign in the contiguous memory blocks.

Algo :- push operation :-

```
begin
  if top = n then stack full
  top = top + 1
  stack(top) := item ;
end
```

Time Complexity : O(1)

pop operation :-

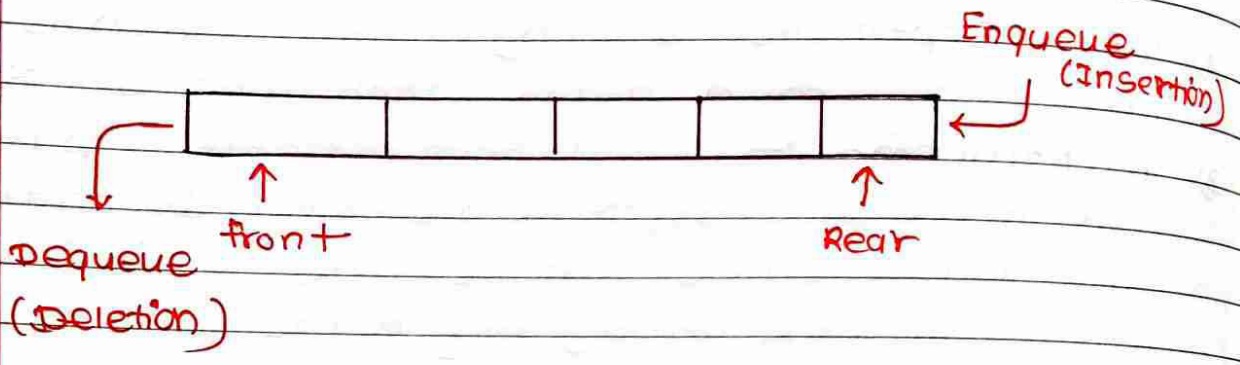
```
begin
  if top = 0 then empty
  item := stack(top) ;
  top = top - 1 ;
end.
```

Time complexity : O(1)

LearnLoner.com

Queue :- A queue can be defined as ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.

- Queue can be referred as to be first In first out list.



Complexity of queue :-

	Average				Space comp.
	Access	search	Deletion	Insertion	Worst
Queue	O(n)	O(n)	O(1)	O(1)	O(n)

	Worst			
	Access	search	Insertion	Deletion
Queue	O(n)	O(n)	O(1)	O(1)

Operations on queue :-

1). Enqueue :- Enqueue is used to insert element at rear end of the queue. It returns void.

2). Dequeue :- dequeue operations performs the deletion from front end of queue. The deque operation can also be designed to void.

LearnLoner.com

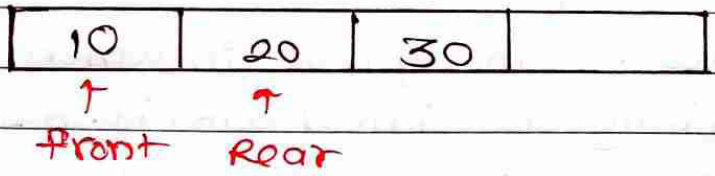
3). peek :- This returns, element which is pointed by front pointer in the queue but does not delete it.

4). queue overflow (is full) :- when queue is completely full, then it shows overflow condition.

5). queue underflow (isempty) :- when there is no element in the queue then it throws underflow condition.

Types of queue :-

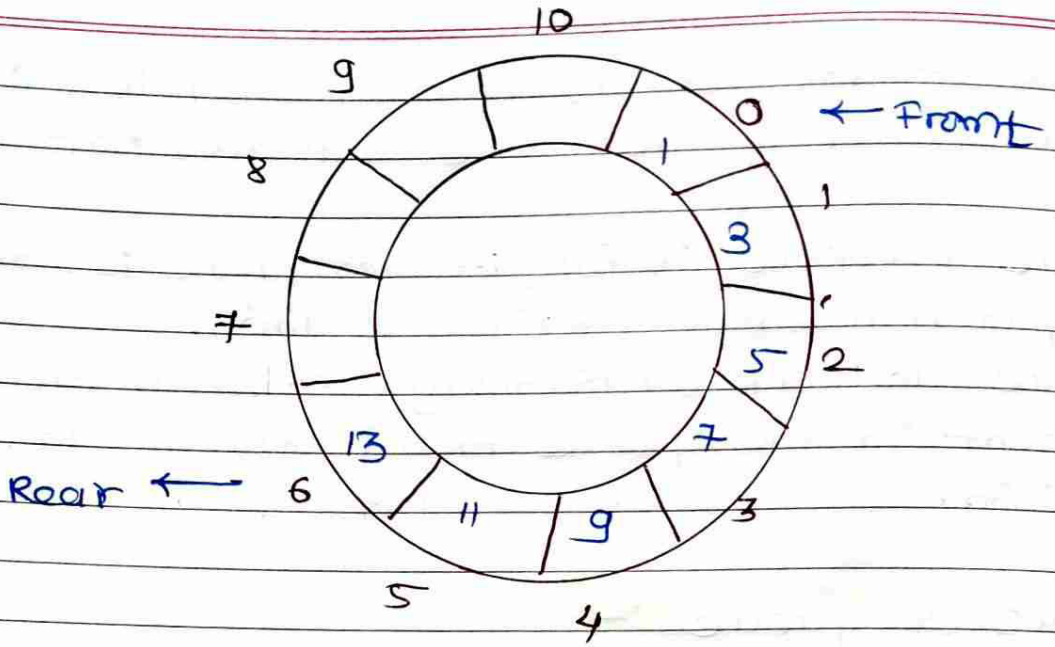
1). Linear queue :- In linear queue, an insertion takes place from one end while deletion occurs from another end. It strictly follows FIFO rule. The linear queue can be represented, as shown :



The elements are inserted from rear end, and if we insert more elements in queue, then rear value gets incremented on every insertion.
drawback is using linear queue is : insertion is done only from rear end. The linear queue shows the overflow condition as rear is pointing to last element of the queue.

2). circular queue :- In circular queue, all nodes are represented as circular. It is similar to linear queue except that last element of the queue is connected to the first element. It is also known as ring buffer. as all ends are connected to another end.

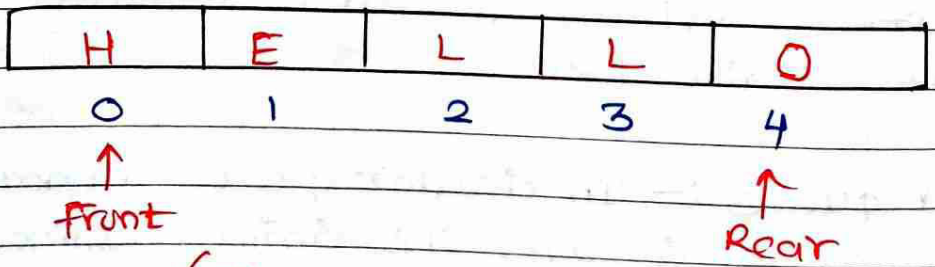
LearnLoner.com



Drawback of linear queue is overcomes in circular queue. If empty space is available, new element can be added in empty space by simply incrementing value of rear.

3). Priority queue :— The queue in which each element has some priority associated with it. Based on priority of the element, elements are arranged in a priority queue. If elements occur with same priority, then they are served according to FIFO principle.

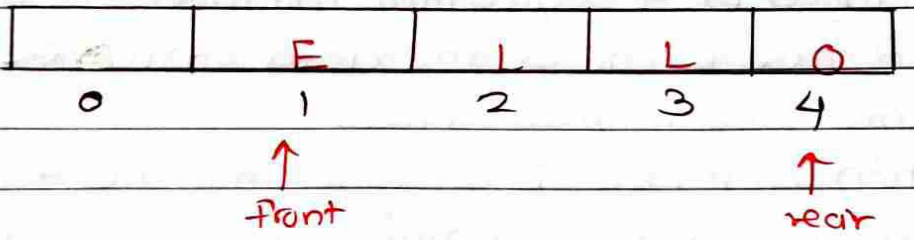
* Array representation of queue :—



(fig : queue after inserting an element)

LearnLoner.com

After deleting element, value of front will increase from -1 to 0, the queue will look like :



(Fig : queue after deleting an element)

Algorithm to insert any element in a queue :-
check if queue is already full by comparing rear to max -1.

Algo:
step 1 :- IF REAR = MAX -1
write OVERFLOW
Go to step [END OF IF]
step 2 :- IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1 [END OF IF].
step 3 :- SET QUEUE [REAR] = NUM
step 4 :- EXIT.

Algorithm to delete an element from queue :-

Algo:
step 1 :- IF FRONT = -1 or FRONT > REAR
write UNDERFLOW
ELSE
SET VAL = QUEUE [FRONT]
SET FRONT = FRONT + 1
[END OF IF]
step 2 :- EXIT.

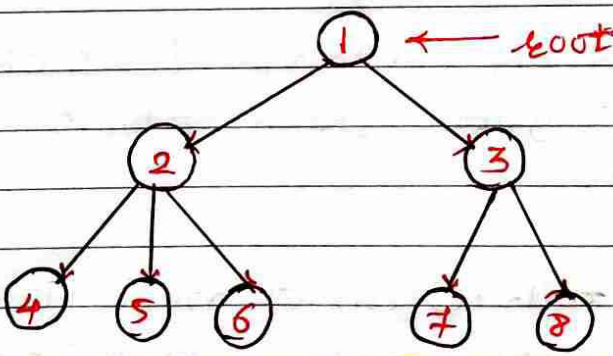
LearnLoner.com

Tree :- We read data structure, like an array, linked list, stack and queue in which all elements are arranged in a sequential manner.

A tree is one of the data structures that represents hierarchical data.

defination :- A tree is a data structure defined as collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.

- A tree is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in tree are arranged in multiple levels.
- In tree data structure topmost node is called as root node. Each node contains some data & data can be of any type.
- Each node contains some data & links or reference of other nodes that can be called children.



Some Basic terms of tree :-

1). **link :-** Each node is labeled with some number. Each array shown in fig is known as link between two nodes.

2). **Root :-** The root node is top most node in tree hierarchy. root node is one that doesn't have any parent. If node is directly linked to some other

LearnLoner.com

node, then it would be called a parent-child relation-ship.

3). child node :- If the node is a descendant of any node, then node is called as child node.

4). parent :- If node contains any sub-node, then node is said to be parent of that sub-node.

5). sibling :- The nodes that have same parents are called siblings.

6). leaf node :- node which doesn't have any child node, a leaf a bottom-most node of tree.

7). ancestor node :- It is any predessor node on a path from root to that node. In the given fig, 1, 2, 5 are ancestors of node 10.

8). Descendant :- The immediate successor of given node is known as descendant of a node.

* Properties of tree data structures :-

1). Recursive data structure :- Tree is also known as recursive data structure. Recursion means reducing something in a self-similar manner.

2). Number of edges :- If there are (n) nodes, then there would be (n-1) edges. each node, except root node, will have atleast one incoming line known as an edge.

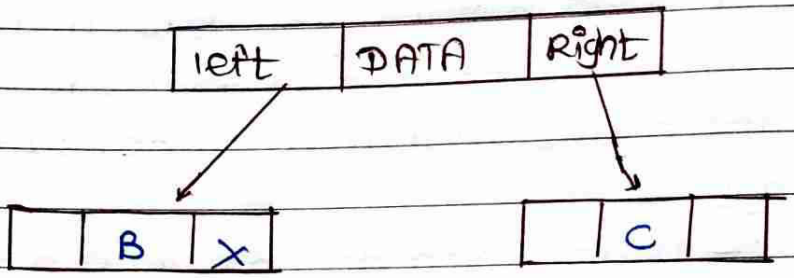
3). Depth of node x :- It can be defined as length of path from root to node x. one edge contributes one unit length in the path, depth can be defined as no. of edges between root node and node (x)
The root node has depth 0.

4). Height of node x :- It is defined as longest path from node x to leaf node.

LearnLoner.com

Implementation of tree :-

The tree data structure can be created nodes dynamically with help of pointers. The tree in memory can be represented as shown :




```
Struct node
 {
  int data;
 struct node *left;
 struct node *right;
 }
```

The above structure can only be defined for binary trees because binary tree can have utmost two children, and generic trees.

Application of trees :-

1). storing naturally hierarchical data :- File system., stored on disc drive, file and folder are in form of naturally heirarchical data and store in form of trees.

2). organize data :- It is used to organize data for efficient insertion, deletion and searching.

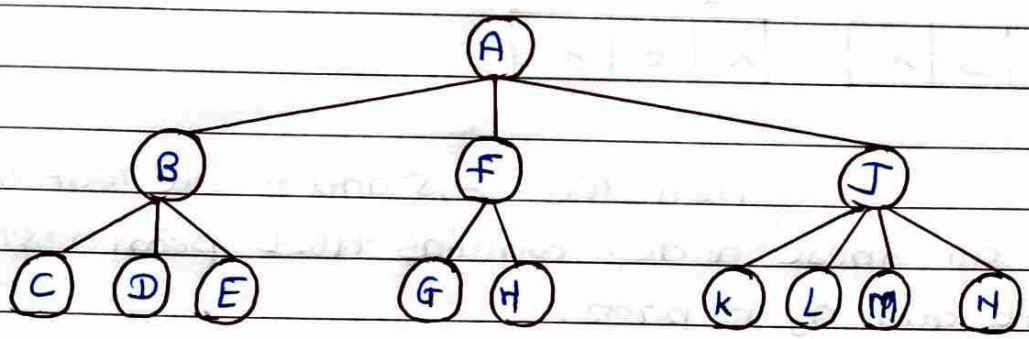
3). Trie :- It is special kind of tree that is used to store dictionary. It is fast and efficient way for dynamic spell checking.

4). Heap :- It is also a tree data structure implemented using arrays. It is used to implement priority queues.

LearnLoner.com

Types of Tree data structure :—

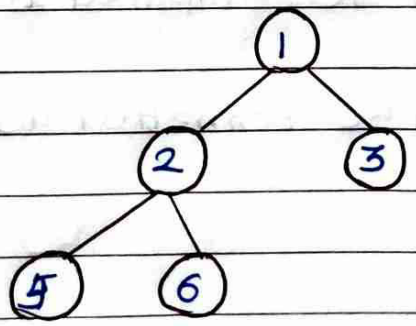
1). General Type :— In a general tree, a node can have either 0 or maximum n number of nodes. There is no restrictions imposed on the degree of node (number of nodes that a node can contain) The topmost node in a general tree is known as root node. The children of parent node are known as subtree.



There can be n number of subtrees in general tree. In general tree, subtrees are unordered as nodes in subtree cannot be ordered.

Every non empty tree has a downward edge, and these edges are connected to nodes known as child nodes. The nodes that have same parent are known as siblings.

2). Binary Tree :— Binary tree means that the node can have maximum two children.

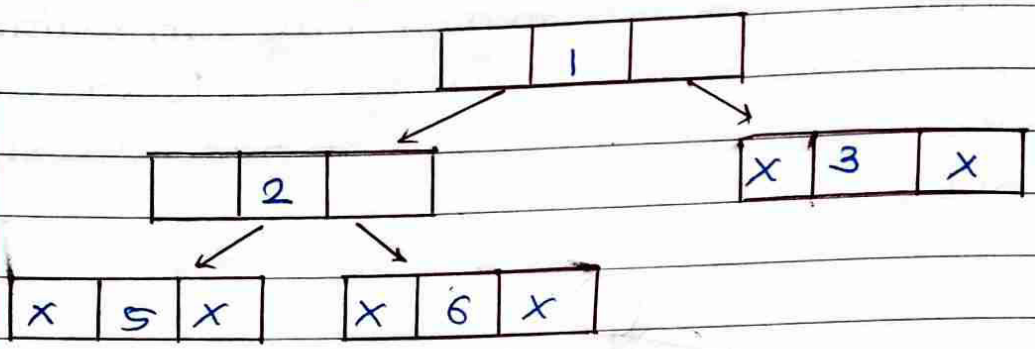


← given tree is a binary tree because
- each node contains atmost two children.

LearnLoner.com

logical representation of above tree :-

In above tree, node 1 contains two pointers i.e. left and right pointer pointing to left and right node respectively.



The nodes 3, 5 and 6 are leaf nodes, so all these nodes contains NULL pointer on both left and right parts.

Properties of Binary tree :-

- At each level of i, the maximum number of nodes is 2^i.
- The height of tree is longest path from root node to leaf node. In general, maximum number of nodes possible at height is $(2^0 + 2^1 + 2^2 + .. 2^h)$
- The minimum number of nodes possible at height h is equal to h+1.
- If number of nodes is minimum, then height of tree would be maximum.
- minimum height can be computed as:
  $h = \log_2(n+1) - 1$
- maximum height can be computed as:
  $h = n - 1$.

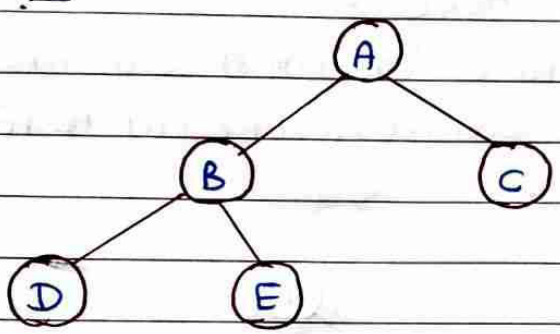
LearnLoner.com

Types of Binary tree :-

1). full / proper / strict Binary tree :-

If each node contains either 0 or two children. The tree in which each node must contain 2 children except left nodes.

Example :-



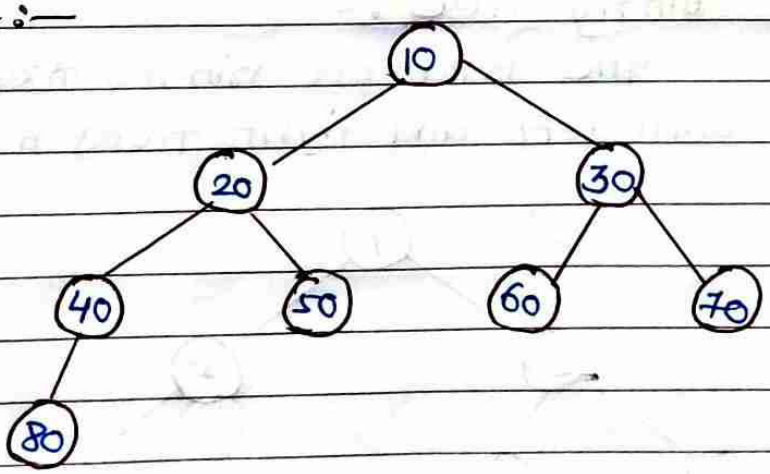
Properties :-

- maximum number of nodes : $2^{h+1} - 1$.
- minimum number of nodes : $2 * h - 1$
- minimum height $\log_2 (n+1) - 1$
- maximum height $h = \frac{n+1}{2}$

2). complete Binary Tree :-

The tree in which all nodes are completely filled except the last level. In complete Binary tree nodes should be added from left.

Example :-



LearnLoner.com

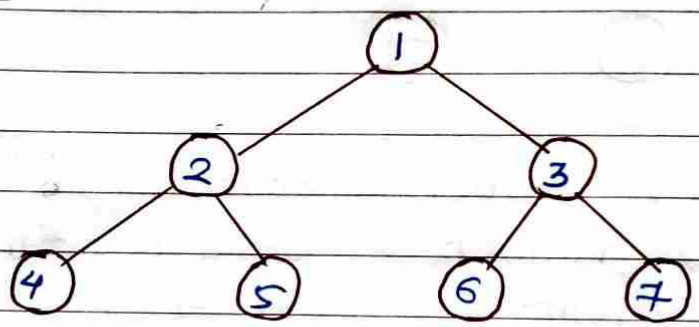
properties :-

- maximum number of nodes → $2^{h+1} - 1$.
- minimum number of nodes → 2^h
- minimum height → $\log_2 (n+1) - 1$.

3). Perfect Binary tree :-

A tree in which all the internal nodes have 2 children. and all leaf nodes are at the same level.

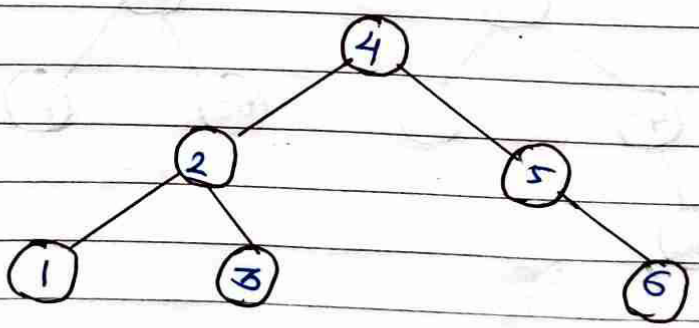
Example :-



Note :- All the perfect Binary trees are complete binary trees as well as the full Binary trees as But, vice versa is not true, all complete binary trees and full binary trees are the perfect Binary trees

4). Balanced Binary Tree :-

The balanced binary tree is a tree in which both left and right trees by almost 1.



above tree is balance : diff bet^n left subtree & right S.T. is zero

LearnLoner.com

Binary Tree implementation :-

```
struct node
 {
  int data;
  struct node *left, *right;
 }
```

Tree Traversal :-

The process of visiting nodes is called as tree traversal.

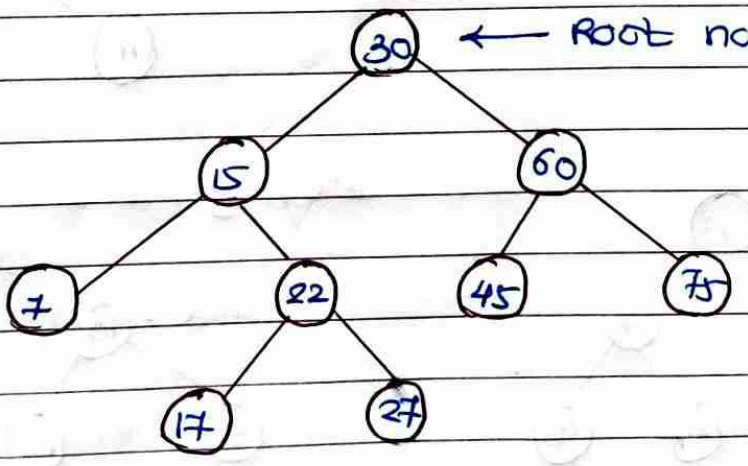
There are three types of traversals used to visit a node :

1). Inorder Traversal
2). preorder Traversal
3). postorder Traversal.

3). Binary Search Tree :-

defn :- Binary search tree can be defined as a class of binary trees, in which a nodes are arranged in a specific order. also called as ordered BinaryTree.

- similary value of all nodes in right subtree is greater than or equal to value of root.



LearnLoner.com

Example : create binary search tree using following data elements :-

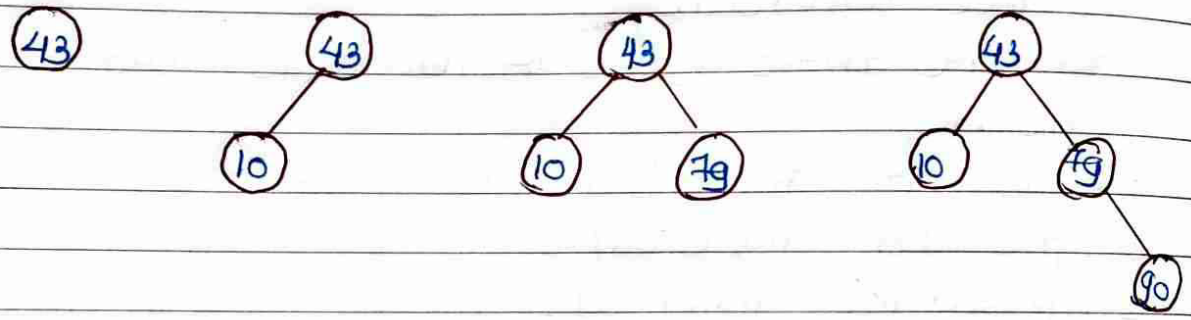
43, 10, 79, 90, 12, 54, 11, 9, 50.

→ 1). Insert 43 into tree as root of tree.

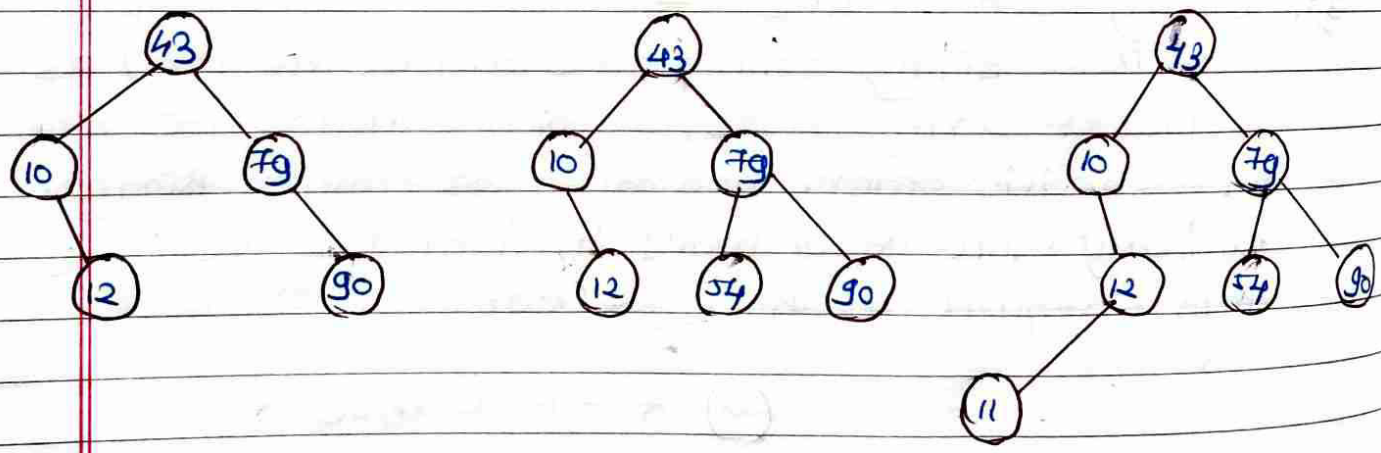
2). Read next element, if it is lesser root node element, insert it as root of left sub-tree.

3). otherwise, insert it as root of right of right subtree.

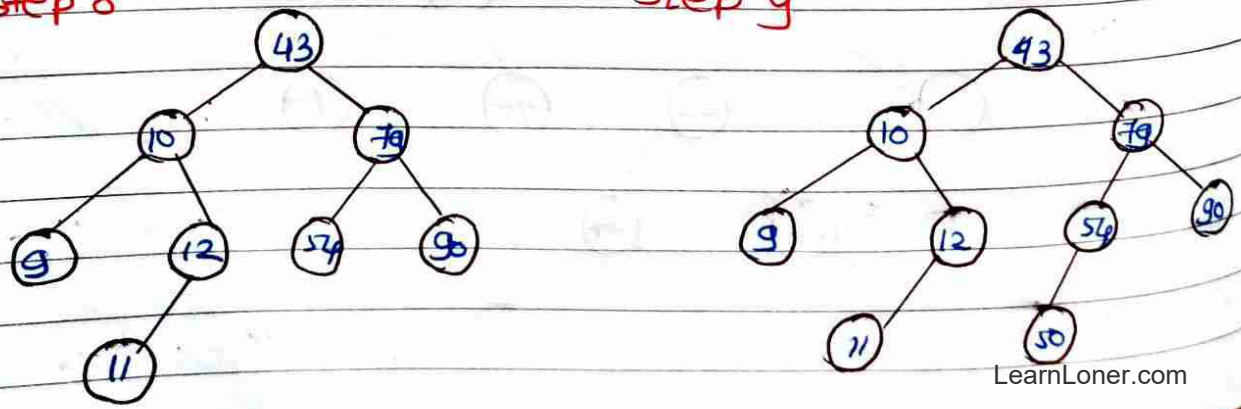
step 1 step 2 step 3 step 4



step 5 step 6 step 7



step 8 step 9



LearnLoner.com

Operations on Binary Search Tree (BST) :—

Sr.No.	Operation	Description
1).	searching in BST	finding location of some specific element in a Binary search tree.
2).	Insertion in BST	Adding a new element to the binary search tree at appropriate location so that property of BST do not violate.
3).	Deletion in BST.	Deleting some specific node from a BST, However, tree there can be various cases in deletion depending upon number of children, node have.

4). AVL Tree :— AVL tree is invented by GM Adelson -Velsky and EM Landis in 1962. The tree is named as AVL in honour of its inventors.

AVL tree is defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtra -cting the height of its R. subtree from its left subtree.

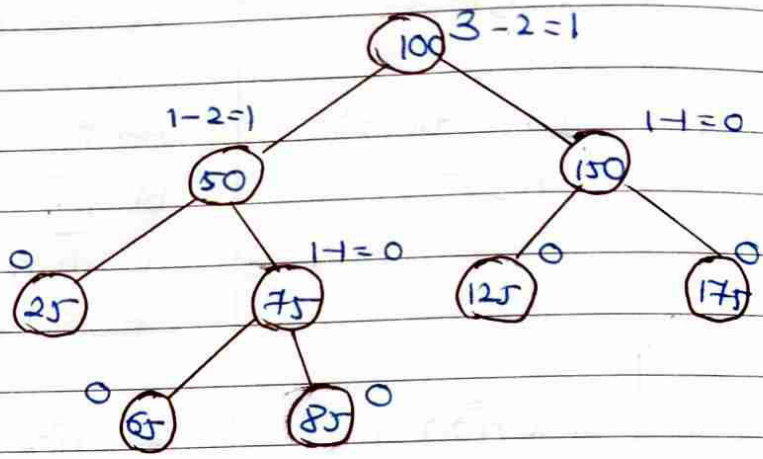
Balance factor (k) = height(left(k)) − height(right(k))

— If balance factor of any node is 1, it means that left sub-tree is one level higher than right subtree.

LearnLoner.com

- If balance factor of any node is 0, it means that left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that left sub-tree is one level lower than right subtree.

Example :-



Here we see that, balance factor associated with each node is between -1 and +1.
∴ It is an example of AVL tree.

Complexity :-

Algorithm	Average case	Worst case
space	O(n)	O(n)
search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Why AVL Tree ? → AVL tree controls height of binary search tree by not letting it to be skewed. The time taken by all operations in BST is O(h). However it will be extended to O(n) if BST became

LearnLoner.com

skewed (worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be O(log n), where n is number of nodes.

Operations on AVL tree :-

Sr. No.	Operation	Description.
1).	Insertion	Insertion is performed in same way it performed in BST. However, it may lead to violation in the AVL tree property and so tree may need balancing. and tree can be balanced by rotation.
2).	Deletion	Deletion is also same way performed as BST It can be also disturb balance of tree, so various types of rotations are used to rebalance tree.

AVL Rotations :-

We perform rotations in AVL tree only in case if Balance factor is other than -1, 0 and 1.
There are Basically four types of rotations which are as follows :

LearnLoner.com

1). L-L rotation :- Inserted node is in the left subtree of left subtree of A.

2). R-R rotation :- Inserted node is in the right subtree of right subtree of A.

3). L-R rotation :- Inserted node is in right subtree of left subtree of A.

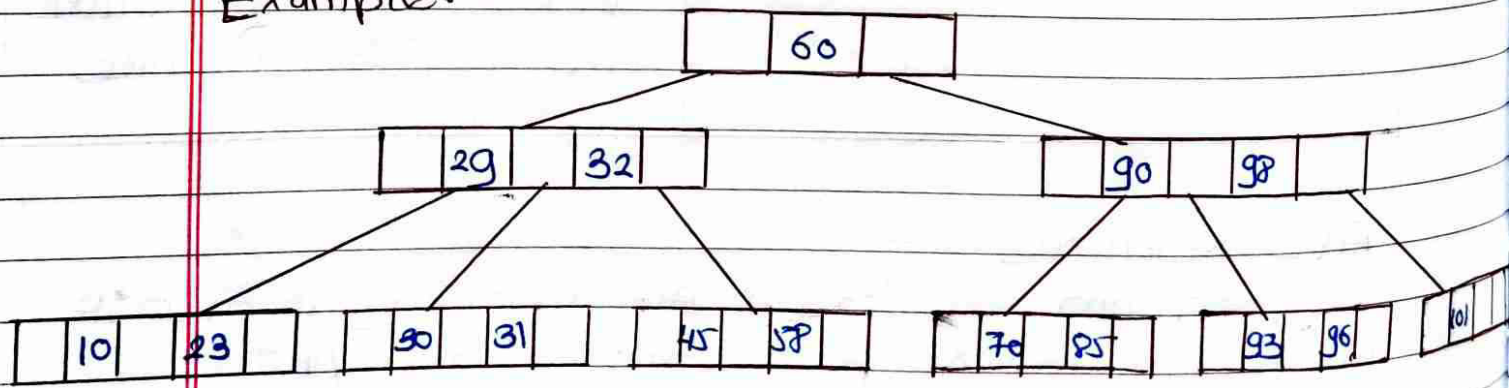
4). R-L rotation :- Inserted node is in the left subtree of right subtree of A.

5). B Tree :- B Tree is specialized m-way tree that can be widely used for disk access. A B-tree of order m can have at most m-1 keys and m children.

Properties :-

1. Every node in B-Tree contains at most m children.
2. Every node in B-Tree except root node and leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

Example :-



LearnLoner.com

Operations :-

1). searching :- The searching in B tree is similar to searching in Binary tree. For example, we search for an item 49 in following B Tree. The process will be :

(1). compare item 49 with root node 78. since $49 < 78$ hence, move its left sub-tree.

(2). since, $40 < 49 < 56$, traverse right subtree of 40.

(3). $49 > 45$, move to right compare 49.

(4). match found, return.

Searching in B tree depends upon height of the tree. The search algorithm takes $O(\log n)$ time to search any element in B tree.

2). Inserting :- Insertion are done at leaf node level. The following algorithm needs to be followed in order to insert an item into B tree.

(1). Traverse B tree in order to find appropriate leaf node at which node can be inserted.

(2). If leaf node contains less than m-1 keys then insert element in increasing order.

(3). Else, if leaf node contains m-1 keys, then follow following steps:

- Insert new element in increasing order of elements.
- split node into two nodes at median.
- Push median element up to its parent node.
- If parent node also contain m-1 number of keys, then split it too by steps.

Application of B tree :-

- B tree is used to index data and provides fast access to actual data stored on disks since, the

LearnLoner.com

to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time.

6). B + Tree :-

B+tree is an extension of B tree which allows efficient insertion, deletion and search operations.

The leaf nodes of B+ tree are linked together in form of the singly linked list to make search queries more efficient.

Advantages of B+ tree :-

1). Records can be fetched in equal number of disk accesses.

2). Height of tree remains balanced and less as compare to B tree.

3). We can access data stored in B+ tree sequentially as well as directly.

4). Keys are used for indexing.

Graph :-

A graph can be defined as group of vertices and edges that are used to connect these vertices.

Defination :-

A graph G can be defined as an ordered set G(V, E) where V(G) represents set of vertices and E(G) represents set of edges.

LearnLoner.com

which are used to connect these vertices.

Directed and Undirected Graph :-

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with directions with them.

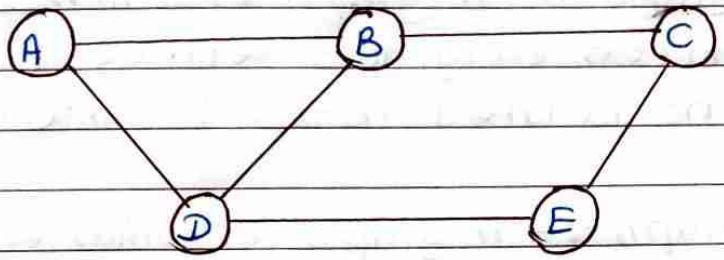


fig : Undirected graph

As above figure edges are not attached with any of the directions.

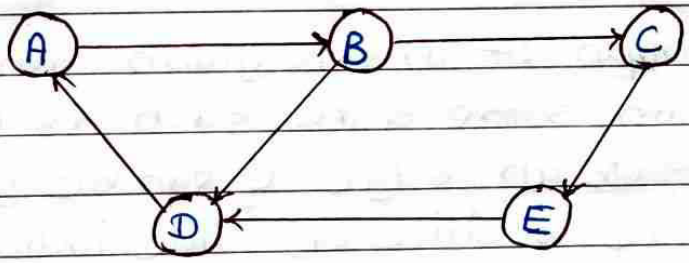


fig: directed graph.

In above figure, directed graph edges form an ordered pair.

Graph Terminology :-

1). Path :- A path can be defined as sequence of nodes that are followed in order to reach some terminal node v from initial node v.

2). closed path :- A path will be called as closed if initial node is same as terminal node. $V_0 = V_N$

LearnLoner.com

3). Simple path :- If all nodes of graph are distinct with an exception $V_0 = V_N$, then such path P is called as closed simple path.

4). Cycle :- A cycle is a path which has no repeated edges or vertices except first and last vertices.

5). Connected graph :- A graph in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

6). Complete graph :- A graph in which every node is connected with all other nodes. A complete graph contain $\frac{n(n-1)}{2}$ edges where n is number of nodes in graph.

7). Weighted graph :- In this graph each node is assigned with some data such as length or width The weight of an edge e can be given as w(e) which must be positive (+) value indicating cost of traversing edge.

8). Diagraph :- A diagraph is directed graph in which each edge is associated with some direction and traversing can be done only in specified direction.

9). Loop :- An edge that is associated with the similar end points can be called as loop.

10). Adjacent Nodes :- If two nodes u and v are connected via an edge e, then nodes u and v

LearnLoner.com

are called as neighbours or adjacent nodes.

ii). Degree of a Node :- A degree of a node is a number of edges that are connected with that node. A node with degree 0 is called isolated.

Graph Representation :-

We simply mean, technique which is to be used to in order to store some graph into the computers memory.

(1). Sequential Representation :- In this we use adjacency matrix to store mapping represented by vertices and edges. A graph having n vertices, will have a dimension n x n.

An entry Mij in adjacency matrix representation of an ~~not~~ undirected graph G will be 1 if there exists an edge between Vi and Vj.

An undirected graph and its adjacency matrix representation is shown in following :

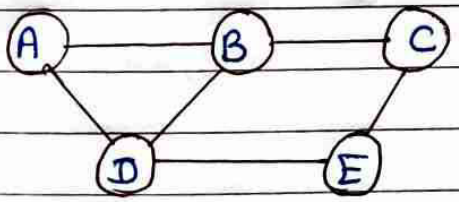


fig : Undirected graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Fig: Adjacency matrix

In above figure, we can see mapping among vertices (A, B, C, D, E) is represented by using adjacency matrix which is also shown in fig.

LearnLoner.com

A directed graph and its adjacency matrix representation is shown in figure :

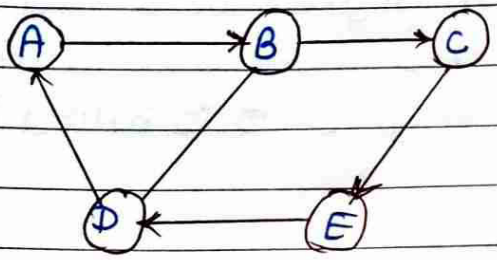


Fig : Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

fig: Adjacency matrix

Representation of weighted directed graph is different. Instead of filling entry by 1, non zero entries of adjacency matrix are represented by weight of respective edges.

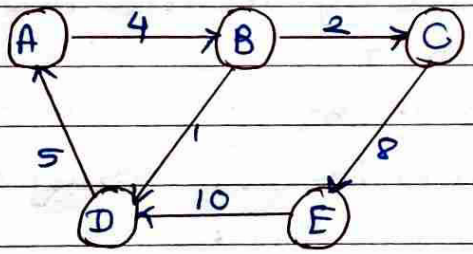


Fig : weighted directed graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

fig: Adjancy matrix

(2) linked representation :-

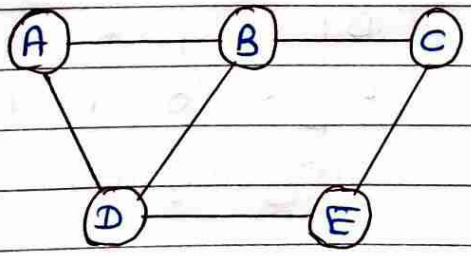


fig : undirected graph

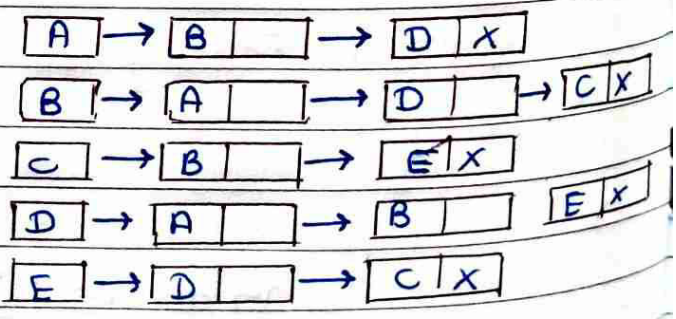


fig : Adjacency list.

LearnLoner.com

An adjacency list is maintained for each node present in graph which stores node value and a pointer to next adjacent node to respective node.

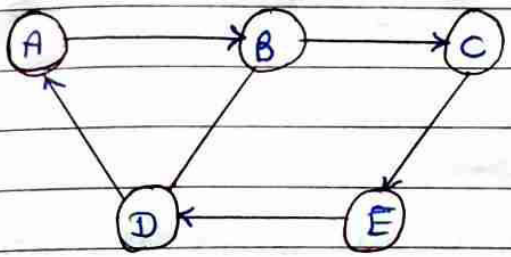


fig : Directed graph

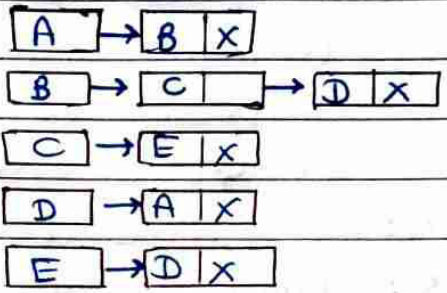


fig : Adjacy list

In directed graph, sum of lengths of all the adjacy lists is equal to the number of edges preseht in the graph.

Graph Traversal Algorithm :-

In this tutorial we will learn all techniques by using which, we can traverse all the vertices of the graph. Traversing means examining all nodes and vertices of graph. There are two standard methods by using which, we can traverse graphs.

- Breadth first search
- Depth first search

1. Breadth first search (BFS) algorithm :-

Breadth first search is a graph traversal algorithm that starts traversing graph from root node and explores all the neighbouring nodes.

Then, it selects nearest node and explore all unexplored nodes. The algorithm follows same process for each of nearest node until it finds goal.

LearnLoner.com

Algorithm :-

step 1: SET STATUS = 1 (ready state) for each node in G.

step 2: Enqueue starting node A & set its STATUS = 2 (waiting state).

step 3: Repeat steps 4 and 5 until QUE E is empty.

step 4: Dequeue a node N. Process it & set its STATUS = 3

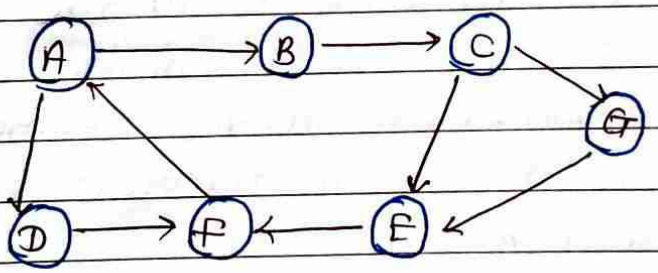
step 5: Enqueue all neighbours of N that are in ready state (whose STATUS = 1) & set (STATUS = 2) [END OF LOOP].

step 6: EXIT.

Example :-

Consider graph G shown in following image, calculate minimum path P from node A to node E. Given that each edge has a length of 1.

→



Adjacency lists :

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F.

Solution :-

Minimum Path P can be found by applying Breadth first search algorithm that will begin at node A and will end at E.

A → B → C → E

LearnLoner.com

Depth first search Algorithm :-

DFS algorithm starts with initial node of graph G, & goes to deeper & deeper until we find goal node / node which has no children. The data structure used in DFS is stack.

Algorithm :-

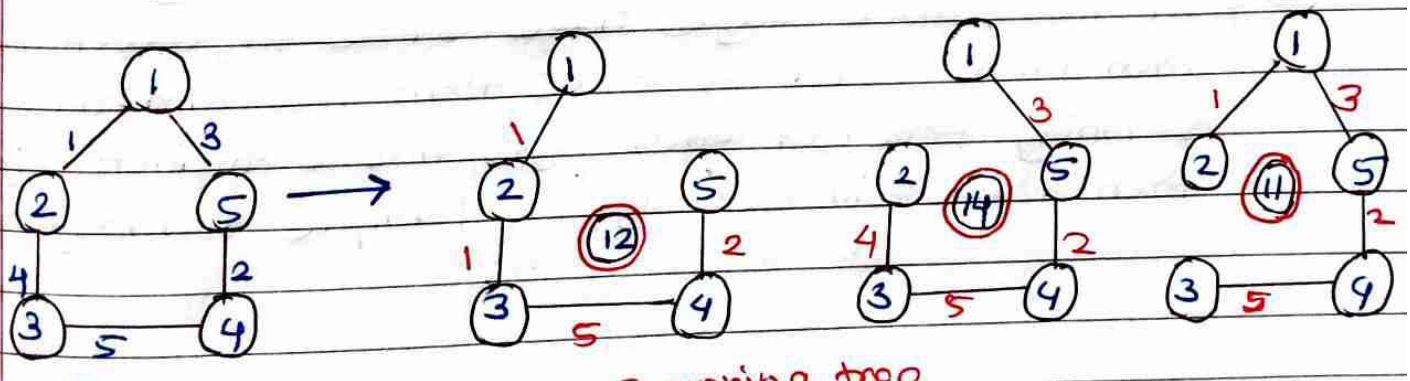
step 1: SET STATUS = 1 (ready state) for each node in G.
step 2: Push starting node A on stack & set its STATUS = 2 (waiting state).
step 3: Repeat steps 4 and 5 until stack is empty.
step 4: Pop top node N. Process it & set its STATUS = 3.
step 5: Push on stack all neighbours of N that are in ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP].
step 6: EXIT.

Spanning Tree :-

If we have a graph containing V vertices and E edges, then graph can be represented as: G(V, E). If we create spanning tree from above graph, then spanning tree would have same number of vertices as the graph, but vertices are not equal.

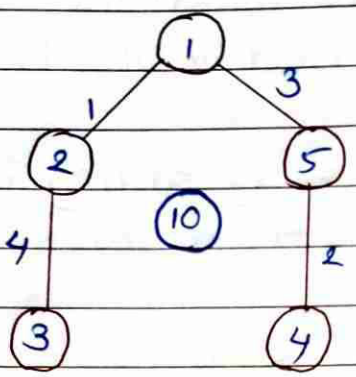
edges (spanning tree) = no. of edge (in graph) - 1.

Example :-



Minimum Spanning Trees :-

The minimum spanning tree is a tree whose sum of edge weights is minimum.



In above tree, total edge weight is less than above spanning trees, therefore a minimum spanning tree is a tree which is having an edge weight as 10.

Properties of Spanning tree :-

- A connected graph can contain more than one spanning tree.
- All possible spanning trees that can be created from given graph G would have same number of vertices in given graph minus 1.
- Spanning tree does not contain any cycle, let's understand this property through an example.
- Spanning tree cannot be disconnected. If we remove one more edge from any of above spanning trees as.
- If two / more edges have same edge weight, then there would be more than two minimum spanning tree. If each edge has a distinct weight, then there will be only one / unique spanning tree.

LearnLoner.com

Applications of Spanning tree :-

1). Building a network :- Suppose there are many routers in network connected to each other, so there might be a possibility that it forms a loop.

2). Clustering :- clustering means that grouping set of objects in such way that similar objects belong to same group than to different group. our goal is to divide the n objects into k groups such that distance between different groups gets maximised.

Searching :-

Searching is a process of finding some particular element in list. If the element is present in the list, then process is called successful and process returns location of that element, otherwise search is called unsuccessful.

There are two methods widely used as below:

- Linear search
- Binary search

1). Linear search :-

linear search is a simplest sequential search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of list with item whose location is to be found.

linear search is mostly used to search an unordered list in which items are not sorted. The algorithm is given as follows :

LearnLoner.com

Algorithm :-

LINEAR_SEARCH (A, N, VAL).

step 1 :- [INITIALIZE] SET POS = -1

step 2 :- [INITIALIZE] SET I = 1

step 3 :- Repeat step 4 while I <= N

step 4 :- IF A[I] = VAL
SET POS = I
PRINT POS
Go to step 6
[END OF IF]
SET I = I+1
[END OF LOOP].

step 5 :- IF POS = -1
PRINT "VALUE IS NOT PRESENT IN ARRAY"
[END OF IF].

step 6 :- EXIT.

Complexity of an Algorithm :-

Complexity	Best case	Average case	Worst case
Time	O(1)	O(n)	O(n)
space			O(1)

C program of linear search :-

```
#include <stdio.h>
 void main ()
{
```

LearnLoner.com

```
int a[10] = { 10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
int item, i, flag;
printf("\n Enter Item which is to be searched \n");
scanf("%d", item);
for (i=0; i<10; i++)
  {
   if (a[i] == item)
    {
     flag = i+1;
     break;
    }
  else
     flag = 0;
}
  if (flag != 0)
   {
   printf("\n item found at location %d \n", flag);
   }
   else
   {
     printf("\n Item not found \n");
   }
}
```

Output :- Enter Item which is to be searched
20
Item not found
Enter Item which is to be searched
23
Item found at location 2.

LearnLoner.com

2). Binary Search :-

Binary search is a search technique which works on efficiently on sorted lists. Hence in order to search an element into some list by using binary search technique, we must ensure that list is sorted.

Binary search follows divide and conquer approach in which, list is divided into two halves and item is compared with middle element of list.

Binary search Algorithm :-

```
BINARY_SEARCH (A, lower-bound, upper-bound, VAL)
step 1 :- [INITIALIZE] SET BEG = lower-bound
          END = upper-bound, pos = -1.

step 2 :- Repeat steps 3 and 4 while BEG<=END

step 3 :- SET MID = (BEG + END)/2

step 4 :- IF A[MID] = VAL
          SET POS = MID
          PRINT POS  Go to step 6
          ELSE IF A[MID] > VAL
          SET END = MID - 1
          ELSE SET BEG = MID + 1

step 5 :- IF POS = -1
          PRINT "VALUE IS NOT PRESENT IN ARRAY"

step 6 :- EXIT.
```

LearnLoner.com

Complexity :-

Sr.No.	Performance	complexity
1).	Worst case	O(log n)
2).	Best case	O(1)
3).	Average case	O(log n)
4).	Worst case space complexity	O(1)

Example: Let us consider an array arr = {1, 5, 7, 8, 13, 19, 20, 23, 29}. find location of item 23 in the array.

→ In 1st step :-

BEG = 0

END = 8 or n

MID = 4

a[mid] = a[4] = 13 < 23, therefore;

In second step :-

Beg = mid + 1 = 5

End = 8

mid = 13/2 = 6

a[mid] = a[6] = 20 < 23, therefore;

In third step :-

beg = mid + 1 = 7

End = 8

mid = 15/2 = 7

a[mid] = a[7].

LearnLoner.com

a[7] = 23 = item ;
therefore, set location = mid ;
The location of item will be 7.

item to be searched : 23

step 1 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

step 2 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

step 3 →

1	5	7	8	13	19	20	23	29
0	1	2	3	4	5	6	7	8

In step 1: a[mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end) / 2 = 13/2 = 6

In step 2 :- a[mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end) / 2 = 15/2 = 7.

step 3 :- a[mid] = 23
23 = 23
loc = mid.

C program : Binary search

```
#include<stdio.h>
int binarysearch (int [], int, int, int);
void main ()
```

LearnLoner.com

```
{
int arr[10] = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
int item, location = -1;
printf(" Enter the item which you want to search");
scanf(" %d ", &item);
location = binarysearch (arr, 0, 9, item);
if (location != -1)
{
printf( "Item found at location %d ", location);
}
else
{
printf ("item not found");
}
}
int Binary search (int a[], int beg, int end, int item)
{
int mid;
if (end >= beg)
{
mid = (beg + end)/2;
if (a [mid] == item)
{
return mid + 1;
}
else if (a [mid] < item)
{
return binary search (a, mid + 1, end, item);
}
else
```

LearnLoner.com

```
{
return binary search (a, beg, mid-1, item);
}
}
return -1;
}
```

output: Enter item which you want to search
19
Item found at location 2.

Sorting Algorithm :-

1) Bubble sort Algorithm :-

Bubble sort algorithm is a simplest sorting algorithm. Bubble sort works on repeatedly swapping of adjacent elements until they are not in intended order. It is called as Bubble sort because moment of array elements is just like movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly the array elements in bubble sort move to end in each iteration.

It is not suitable for large data sets. The average and worst case complexity of bubble sort is $O(n^2)$, where n is number of items.

Bubble sort is majorly used where :

- complexity does not matter
- simple and shortcode is preferred.

LearnLoner.com

Algorithm :-

```
begin BubbleSort(arr)
  for all array elements
  "if arr[i] > arr[i+1]
    swap(arr[i], arr[i+1])
  end if
 end for
 return arr
end Bubble sort.
```

Bubble sort complexity :-

case	Time complexity	space complexity
Best case	$O(n)$	$O(1)$.
Average case	$O(n^2)$	
Worst case	$O(n^2)$	

Implementation of Bubble Sort :-

C language Implementation :-

```
#include<stdio.h>
void print(int a[], int n)
 {
  int i;
  for(i=0; i<n; i++)
   {
    printf("%d", a[i]);
   }
```

LearnLoner.com

```
}
void bubble (int a[], int n)
 {
 int i,j,temp;
 for(i=0;i<n;i++)
  {
   for (j=(i+1) ; j<n ;j++)
   {
   if (a[j]<a[i])
     {
     temp = a[i];
     a[i] = a[j];
     a[j] = temp;
     }
  }
}
}
void main ()
 {
  int i,j, temp;
  int a[5] = { 10, 35, 32, 13, 26 };
  int n = sizeof (a) / sizeof (a0);
  printf ("Before sorting array elements are :\n");
  print (a,n);
  bubble (a,n);
  printf ("\n after sorting array elements - \n");
  print (a,n);
}
```

LearnLoner.com

output :-
Before sorting array elements are -
10 35 32 13 26
After sorting array elements are -
10 13 26 32 35.

Bucket Sort Algorithm :-

The data items in the bucket sort are distributed in term of buckets.

Bucket sort is a sorting algorithm that seprates elements into multiple groups said to be buckets. Elements in bucket sort are first uniformly divided into groups called buckets, and then they are sorted by any other sorting algorithm. After that, elements are gathered in sorted manner.

Advantages of bucket sort are :-
- Bucket sort reduces no. of comparisons
- It is asymptotically fast because of uniform distribution of elements.

Limitations of bucket sort are :
- It may or may not be a stable sorting algorithm
- It is not useful if we have a large array bcz it increases the cost.
- It is not an in-place sorting algorithm, because more some extra space is required to sort the buckets.

LearnLoner.com

The best and average-case complexity of bucket sort is $O(n+k)$, worst-case complexity of bucket sort is $O(n^2)$, where n is number of items.

Bucket sort is commonly used :

- with floating-point values.
- when input is distributed uniformly over a range.

Algorithm :-

Bucket sort (A[.])

1. Let B[0 .... n-1] be a new array.
2. n = length [A].
3. for i = 0 to n-1
4. make B[i] an empty list
5. for i = 1 to n
6. do insert A[i] into list B[n a[i]]
7. for i = 0 to n-1.
8. do sort list B[i] with insertion sort.
9. concatenate lists B[0], B[1] .... B[n-1] together in order.
10. END.

Complexity :-

1. Time complexity :-

case	time complexity
Best case	$O(n+k)$.

LearnLoner.com

Average case	$O(n+k)$
worst case	$O(n^2)$

2. space complexity :-

Space complexity	$O(n*k)$
stable	YES

Implementation of bucket sort in C :-

```
#include <stdio.h>
int getmax (int a[], int n)
{
 int max = a[0];
 for (int i=1; i<n; i++)
   if (a[i] > max)
      max = a[i];
   return max;
}
void bucket (int a[], int n)
 {
 int max = getmax (a,n)
 int bucket [max], i;
 for ( int i = 0 ; i <= max ; i++)
  {
```

LearnLoner.com

```
        bucket [i]=0 ;
    }
for(int i=0 ;i<n ;i++)
    {
     bucket [a[i]]++;
    }
for (int i= 0; j=0 ; i<=max ;i++)
    {
    while (bucket [i] >0)
      {
      a[j++] =i ;
      bucket [i]-- ;
      }
}
}
void printArr (int a[],int n)
  {
   for (int i=0; i<n ; i++)
   printf("%d", a[i]);
  }
int main ()
{
 int a[]= {54, 12, 84, 57, 69, 41, 9, 5};
 int n = size of(a) / size of a[0] ;
 printf ("Before sorting array elements are:-\n");
 printArr(a,n);
  bucket (a,n);
 printf ("\n After sorting array elements are:-\n");
  printArr(a,n);
 }
```

LearnLoner.com

output :-
Before sorting array elements are :-
54 12 84 57 69 41 9 5
After sorting array elements are :-
5 9 12 41 54 57 69 84.

Heap Sort Algorithm :-

Heap sort processes the elements by creating min-heap or max-heap using the elements of the given array.

Heap sort basically recursively performs two main operations :

- Build a heap H, using the element of array.
- Repeatedly delete the root element of heap formed in 1st phase.

What is heap ?

A heap is a complete binary tree, and binary tree is a tree in which node can have utmost two children.

Algorithm :-

```
Heap Sort (arr)
BuildMaxHeap(arr)
for i = length (arr) to 2
   swap arr [i] with arr [i]
    heap-size [arr] = heap-size [arr] ? 1
    MaxHeapify (arr, 1)
End.
```

LearnLoner.com

```
BuildMaxHeap(arr):
    BuildMaxHeap(arr)
        heap-size(arr) = length(arr)
        for i = length(arr)/2 to 1.
    MaxHeapify(arr, i)
End.
```

Complexity :-

case	Time complexity	space complexity
Best	o(n log n)	o(1).
Average	o(n log n)	
worst	o(n log n)	

Implementation of Heap sort :-

```
#include<stdio.h>
/* function to heapify a subtree. Here is 'i' the
index of root node in array a[], and 'n' is size
of heap */
void heapify(int a[], int n, int i)
 {
  int largest = i
  int left = 2*i+1
  int right = 2*i+2
  if(left<n && a[left]> a[largest]
    largest = left;
```

LearnLoner.com

```
if (right<n && a[right]>a[largest]
  largest = right ;
if (largest != i)
{ int temp = a[i] ;
  a[i] = a[largest];
  a[largest] = temp;
  heapify (a, n, largest);
}
}
void heapsort (int a[], int n)
{
  for(int i = n/2-1 ; i>=0, i--)
    heapify (a, n, i);
  for (int i=n-1; i>=0 ; i--)
  { int temp = a[0];
    a[0] = a[i];
    a[i] = temp;
  heapify (a, i, 0);
}
} void printArr (int arr[], int n).
{
  for(int i= 0; i<n ; ++i)
    {
      printf ("%d ", arr[i]);
      printf (" ");
    }
} int main ()
{
 int a[] = { 48, 10, 23, 43, 28, 26, 1};
 int n = sizeof (a) / sizeof (a[0]);
 printf ("Before sorting array elements are - \n");
```

LearnLoner.com

```
printArr(a,n);
heapSort(a,n);
printf("\n After sorting array elements are -\n");
printArr(a,n);
return 0;
}
```

output: Before sorting array elements are :-
48 10 23 43 28 26 1
After sorting array elements are -
1 10 23 26 28 43 48.

Insertion sort Algorithm :-

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card. The idea behind the insertion sort is that first make/take one element, iterate it through sorted array. complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is number of items.

Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort and merge sort etc.

Insertion sort has various advantages such as :

- simple implementation.
- Efficient for small data sets.
- Adaptive i.e it is appropriate for data sets that are already substantially sorted.

LearnLoner.com

complexity :-
case time complexity :
Best case $O(n)$
Avergge case $O(n^2)$
worst case $O(n^2)$.

space complexity $O(1)$.

Implementation of insertion sort :-

```
#include <stdio.h>
void insert (int a[], int n)
{
  int i, j, temp;
  for (i=1; i<n, i++) {
    temp = a[i];
    j = i-1;
   while (j>=0 && temp <= a[j])
   {
    a[j+1] = a[j];
    j = j-1;
   }
   a[j+1] = temp;
 }
}
void printArr (int a[], int n)
 {
   int i;
   for(i=0; i<n; i++)
    printf ("%d", a[i]);
 }
int main()
{
  int a[] = { 12, 31, 25, 8, 32, 17};
```

LearnLoner.com

```
int n = sizeof (a) / sizeof a[0];
printf ("Before sorting array elements are -\n");
printArr (a, n);
insert (a, n);
printf ("\nAfter sorting array elements are -\n");
printArr(a, n);
return 0;
}
```

output: Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32.

Merge Sort Algorithm :-

Merge sort is the sorting technique that follows divide and conquer approach. This will be very helpful and interesting.

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort elements.

Algorithm :-

arr is given array, beg is starting element and end is last element of array.

```
MERGE - SORT (arr, beg, end)
if beg < end
Set mid = (beg + end)/2.
MERGE - SORT (arr, beg, mid)
MERGE - SORT (arr, mid+1, end)
MERGE (arr, beg, mid, end)
```

LearnLoner.com

end of if
End MERGE_SORT.

implementation of merge sort:-

```
/* function of merge the subarrays of a[] */
void merge (int a[], int beg, int mid, int end)
{
  int i,j,k;
  int n1 = mid - beg+1;
  int n2 = end - mid;
  int LeftArray [n1], RightArray [n2];
  /* copy data to temp arrays */
  for(int i=0;i<n1;i++)
  LeftArray [i] = a[beg +i];
  for (int j=0; j<n2;j++)
  RightArray[j] = a[mid+1 +j];
  i=0;
  j=0;
  k=beg;
while (i<n1 && j<n2)
{
  if(LeftArray [i]<=RightArray [j])
  {
    a[k] = LeftArray[i];
    i++;
  }
else
{
a[k]=RightArray [j];
j++;
}
```

LearnLoner.com

```
    }
    k++;
    }
    while (i<n1)
    {
    a[k] = LeftArray[i];
     i++;
     k++;
    }
    while (j<n2)
    {
     a[k] = RightArray[j];
     j++;
     k++;
    }
}
```

Complexity :-

case	Time complexity	space complexity
Best case	$o(n*\log n)$	$o(n)$.
Avg. case	$o(n*\log n)$	
worst case	$o(n*\log n)$	

LearnLoner.com

DATA STRUCTURE CODING QUE.

Arrays by using c :

1). program to demonstrate arrays in c

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
# define NUM_EMPLOYEE 10
int main (int argc, char *argv[])
{ int salary [NUM_EMPLOYEE], lcount=0,
    gcount=0 , i=0;
  printf ("Enter employee salary (MAX 10)\n");
     for (i=0; i<NUM_EMPLOYEE ; i++)
  {
    printf ("\n Enter employee salary : %d -",
            i+1) ;
    scanf ("%d ", &salary [i]);
  }
     for (i=0; i<NUM_EMPLOYEE ; i++)
  {  if (salary [i] <3000)
     lcount ++ ;
    else
     gcount ++;
  }
  printf ("\n There are {%d} employee with
          salary more than 3000 \n", gcount);
  printf ("There are {%d} employee with salary
          less than 3000 \n", lcount);
  printf ("press Enter to continue .. \n");
```

LearnLoner.com

```
    getchar ();
    return 0;
}
```

27. Linked list in c++ :

```
using namespace std;
template <typename T>
class node
{
  public :
   T value;
   Node *next;
   Node *previous;
   Node (T value)
   {
     this->value = value;
   }
};
  template <typename T>
  class linked list
  {
    private :
    int size;
    Node<T> *head_ = NULL;
    Node<T> *tail_= NULL;
    Node<T> *itr = NULL;
    public;
    Linked list()
    {
     this->size_ = 0;
    }
```

LearnLoner.com

```
void append (T value)
{
 if (this->head == NULL)
{
 this->head - = new Node <T> (value);
 this->tail = this->head - ;
 }
else
 {
this - > tail_-> next = new node <T> (value)
this -> tail -> next -> previous = this_
                              ->tail ;

 }
this->size - + = 1;
 }
void prepend (T value)
 {
void resetIterator ()
 {
 tail - = NULL ;
 }
int main (int argc, char **argv)
 {
    LinkedList <int> llist ;
    llist. append (10);
    llist. append (3);
    llist. append (1);
  cout<< "printing linked list <<endl;
  cout<<endl ;
  return 0;
 }
```

LearnLoner.com

3). stack implementation in c :

```
#include <stdio.h>
int MAXSIZE = 8;
int stack [8];
int top = -1;
int isempty() {
   if (top == -1)
      return 1;
   else
      return 0;
   }
int isfull() {
   if (top == MAXSIZE)
      return 1;
   else
      return 0; }
int peek() {
   return stack [top]; }
int pop() {
   int data;
   if(!isempty()) {
      data = stack [top];
      top = top-1;
      return data; }
   else {
      printf("could not retriere data, stack is empty\n");
   }
}
int push (int data) {
   if (!isfull()) {
      top = top+1;
```

LearnLoner.com

```
    stack[top] = data;
  } else {
    printf("could not insert data, stack is full\n");
  }
}
int main() {
  // push items on to the stack
  push(3);
  push(5);
  push(9);
  push(1);
  push(12);
  push(15);
  printf("Element at top of the stack : %d\n", peek());
  printf("Elements : \n");
  // print stack data
  while (!isempty()) {
    int data = pop();
    printf("%d\n", data);
  }
  printf("stack full : %s\n", isfull() ? "true" : "false");
  printf("stack empty : %s\n", isempty() ? "true" : "false");
  return 0;
}
```

LearnLoner.com

DATA STRUCTURES INTERVIEW QUESTIONS WITH ANSWERS

Q.1. What is data structure ?

→ A data structure is a way of organizing data that considers not only items stored, but also their relationship to each other.

Q.2. List out the areas in which data structure are applied extensively ?

→ • compiler design , • operating system,
• database system , • statistical analysis,
• numerical analysis , • artificial intelligence

Q.3. What are major data structures used in following areas RDBMS, network data model and Hierarchical data model.

→ RDBMS = array (array of structures).
network data model = graph.
Hierarchical data model = tree.

Q.4. If you are using C language to implement the heterogeneous linked list, what pointer type will you use ?

→ The heterogeneous linked list contains different data types in its nodes and we need a link, pointer to connect them. It is not possible to use ordinary pointer for this. so we go for void pointer. Void pointer is capable of storing pointer to any

LearnLoner.com

type as it is a generic pointer type.

Q.5. minimum number of queues needed to implement the priority queue?

→ two. one queue is used for actual storing of data and another for storing priorities.

Q.6. What is data structure used to perform recursion?

→ stack. because of its LIFO (Last IN first Out) property it remembers its 'caller'

Q.7. What are notations used in evaluation of arithmatic expressions using prefix & postfix forms?

→ Polish and Reverse polish notations.

Q.8. convert expression ((a+b)*c-(d-e)^(f+g)) to equivalent prefix and postfix notations.

→ prefix notation : - * + abc ^ - de + fg
postfix notation : ab+c*de-fg+^-

Q.9. What are methods available in storing sequential files?

→ 1. straight merging,
2. natural merging,
3. polyphase sort,
4. distribution of initial runs.

Q.10. Whether linked list is a linear or non-linear data structure?

LearnLoner.com

→ According to access strategies linked list is a linear one. according to storage linked list is a non linear one.

Q.11. define doubly linked list.

→ It is collection of data elements called nodes, where each node is divided into three parts:
- an info field that contains information stored in the node.
- left field that contain pointer to node on left side.
- Right field that contain pointer to node on right side.

Q.12. What are the Issues that hampers efficiency in sorting a file?

→
- length of time required by programmer in coding a particular sorting program.
- amount of machine time necessary for running the particular program.
- amount of space necessary for particular pgm.
- object oriented analysis and design.

Q.13. calculate efficiency of sequential search?

→ The number of comparisons depends on where the record with argument key appears in table
- If it appears at first position then one comparison.
- If it appears at last position then n comparison.
- average = $\frac{n+1}{2}$ comparisons.

LearnLoner.com

• number of comparisons in any case is O(n).

Q.14. Is any implicit arguments are passed to a function when it is called ?

→ yes, there is a set of implicit arguments that contain information necessary for function to execute and return correctly, one of them is return address which is stored within the function's data area, at time of returning to calling program address is retrived and function branches to that location.

Q.15. Paranthesis is never required in postfix or prefix expressions ? Why

→ parenthesis is not required because order of the operators in postfix / prefix expressions determines actual order of operations in evaluating expression.

Q.16. List out few of applications of tree data structure ?

→ The manipulation of arithmatic expression, symbol table construction & syntax analysis.

Q.17. List out few of applications that make use of multilinked structures ?

→ sparse matrix, Index generation.

Q.18. What is type of the algorithm used in solving 8 queens problem ?

LearnLoner.com

→ backtracking.

Q.19. In an AVL Tree, at what condition balancing is to be done?

→ If 'pivotal value' or height factor is greater than 1 or less than -1.

Q.20. In Rdbms, what is the efficient data structure in internal storage representation.

→ b+tree. because bt tree, all the data is stored in only in leaf nodes, that makes searching easier. this corresponds to records that shall be stored in leaf nodes.

Q.21. What is difference between array and a stack?

→ Stack follows LIFO. thus the item that is first entered would be last to be removed.

In the array, items can be entered or removed by in any order. basically, each member access is done using index. no strict order is to be followed here to remove a particular element.

Q.22. How to check whether a linked list is circular?

→ create two pointers, each set to start of list.
update each as follows:

```
while (pointer 1)
{
  pointer 1 = pointer 1 -> next;
  pointer 2 = pointer 2 -> next;
  if (pointer 2) pointer 2 = pointer 2 -> next;
```

LearnLoner.com

```
if (pointer 1 == pointer 2)
{
print("circularn");
}
}
```

Q.23. What is a node class?

→ A node class is class that, relies on the base for service and implementation, provides a wider interface to users than its base class, relies primarily on virtual functions in its public interface depends on all its direct and indirect base class.

Q.24. When can you tell that a memory leak will occur?

→ a memory leak occurs when a program loses the ability to free a block of dynamically allocated memory.

Q.25. What are types of collision Resolution techniques and methods used in each of the type?

→ open addressing (closed hashing), methods used include: overflow block. closed addressing (open hashing) methods used include: linked list, binary tree

Q.26. Which is simplest file structure? (sequential, index, random).

→ Sequential is the simplest file structure.

LearnLoner.com

Q.27. What are the notations, used in evaluation of arithmatic expression, using prefix and postfix forms?

→ Polish and Reverse polish notations.

Q.28. list out few of applications of tree data structure?

→ The manipulation of arithmatic expressions, symbol table construction and syntax analysis.

Q.29. difference between calloc and malloc?

→ malloc : allocate n bytes.
calloc : allocate m times n bytes initialized to 0.

Q.30. Which file contains the definition of member function

→ defination of member function for the linked list class are contained in linkedlist.cpp file.

Q.31. How is the front of the queue calculated?

→ The front of the queue is calculated by
front = (front + 1) % size.

Q.32. Why is the Isempty() member method called?

→ the isempty() member method is called within the dequeue process to determine if there is an item in dequeue to be removed i.e. isempty() is called to decide whether queue has atleast one element. This method is called by dequeue() method before returning front element.

Q.33. Which process places data at back of queue?

LearnLoner.com

→ enque is a process that places data at back of the queue.

Q.34. What is queue ?

→ A queue is sequential organization of data. a queue is a first in first out type of date structure. an element is inserted at last position and an element is always taken out from first position.

Q.35. What does isempty() member method determine

→ isempty() checks if stack has at least one element. this method is called by pop() before retrieving and returning top element.

Q.36. What method removes value from top of a stack ?

→ The pop() member method removes value from top of a stack, which is then returned by the pop() member method to statement that calls Pop() member method.

Q.37. What method is used to place a value onto the top of a stack ?

→ push() method, push is the direction that data is being added to stack. push() member method places a value onto the top of a stack.

Q.38. How do you assign an address to an element

LearnLoner.com

of a pointer array ?

→ We can assign a memory address to an element of a pointer array by using the address operator, which is ampersand (&), in an assignment statement such as ptemployee[0] = &projects[2];

Q.39. How many parts are there in a declaration statement ?

→ There are two main parts, variable, identifier & data type and third type is optional which is type qualifier like signed / unsigned.

Q.40. list some of the static data structures in c ?

→ some of the static data structures in c are arrays, pointers, structures etc.

Q.41. define dynamic data structure ?

→ A data structure formed when number of data items are not known in advance is known as dynamic data structure or variable size data structure.

Q.42. list some of dynamic data structures in c ?

→ some of dynamic data structures in c are linked lists, stack, queues, trees etc.

Q.43. define linear data structure.

→ linear data structures are data structures having a linear relationship between its adjacent elements.
eg : linked list.

LearnLoner.com

Q.44. define nonlinear data structures.

→ Non linear data structure are the data structures are data structure that don't have a linear relationship between its adjacent elements but have a hierarchical relationship between the elements.
eg : trees and graphs.

Q.45. state the different types of linked lists?

→ The different types of linked list include singly linked list, doubly linked list and circular linked list.

Q.46. List the basic operations carried out in a linked list?

→
- creation of a list
- Insertion of a list.
- deletion of a node.
- modification of a node.
- traversal of a node.

Q.47. define a stack.

→ Stack is an ordered collection of an elements in which insertion and deletions are restricted to one end. The end from which elements are added and or removed is referred as top of stack.

Q.48. List out the basic operations that can be performed on a stack.

→
- push operation.

LearnLoner.com

- pop operation
- peek operation
- empty check
- fully occupied check.

Q.49. State the different ways of representing expression

→
- Infix notation.
- prefix notation
- postfix notation.

Q.50. What is sequential search?

→ In sequential search each item in the array is compared with the item being searched until a match occurs.

LearnLoner.com

THANK YOU !!!

keep following us on our

Social Media Channels

for More Helpful Contents !

LearnLoner.com

URBAN EDGE ___/___/___

Data Structures & Algorithms by CodeWithHarry

This course will get you prepared for placements and will teach you how to create efficient and fast algorithms.

Data structures and algorithms are two different things.

Data Structures: Arrangement of data so that they can be used efficiently in memory (data items)

Algorithms: Sequence of steps on data using efficient data structures to solve a given problem.

Other Terminology

Database :- Collection of information in permanent storage for faster retrieval and updation.

Data warehousing :- Management of huge amount of legacy data for better analysis.

Big data :- Analysis of too large or complex data which cannot be dealt with traditional data processing application.

Data Structures and Algorithms are nothing new. If you have done programming in any language like C you must have used Arrays → A data structure and some sequence of processing steps to solve a problem → Algorithm

URBAN EDGE

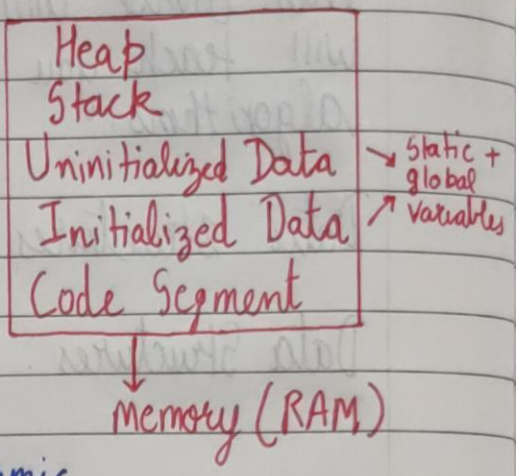
Memory layout of C programs

When the program starts, its Code is copied to the main memory.

Stack holds the memory occupied by the functions.

Heap contains the data which is requested by the program as dynamic memory.

Initialized and uninitialized data segments hold initialized and uninitialized global variables respectively



URBAN EDGE

Time Complexity & Big O notation

This morning I wanted to eat some pizzas; So I asked my brother to get me some from Dominos (3 km far)
He got me the pizza and I was happy only to realize it was too less for 29 friends who came to my house for a surprize visit!

My brother can get 2 pizzas for me on his bike but pizza for 29 friends is too huge of an input for him which he cannot handle.

2 pizzas ⟶  okay! not a big deal!

68 pizzas ⟶  Not possible! in short time

What is Time Complexity?
Time Complexity is the study of efficiency of algorithms.

(i) Time Complexity = How time taken to execute an algorithm grows with the size of the input!

Consider two developers who created an algorithm to sort n numbers. Shubham and Rohan did this independently.

When ran for input size n, following results were recorded:

no of elements (n)	Shubham's Algo	Rohan's Algo
10 elements	90 ms	122 ms
70 elements	110 ms	124 ms
110 elements	180 ms	131 ms
1000 elements	2 s	800 ms

We can see that initially Shubham's algorithm was shining for smaller input but as the number of elements increases Rohan's algorithm looks good!

Quick Quiz: Who's Algorithm is better?

Time Complexity: Sending GTA V to a friend
Let us say you have a friend living 5 kms away from your place. You want to send him a game.
Final exams are over and you want him to get this 60 GB file from you. How will you send it to him?
Note that both of you are using (JIO) 4G with 1 GB/day data limit.

The best way to send him the game is by delivering it to his house.
Copy the game to a Hard disk and send it!

Will yo do the same thing for sending a game like minesweeper which is in kbs of size?
No because you cand send it via internet.

As the file size grows, time taken by online sending increases linearly → $O(n^1)$
As the file size grows, time taken by physical sending remains constant. $O(n^0)$ or $O(1)$

Calculating Order in terms of Input size
In order to calculate the order, most impactful term containing n is taken into account.
↳ Size of input

Let us assume that formula of an algorithm in terms of input size n looks like this:

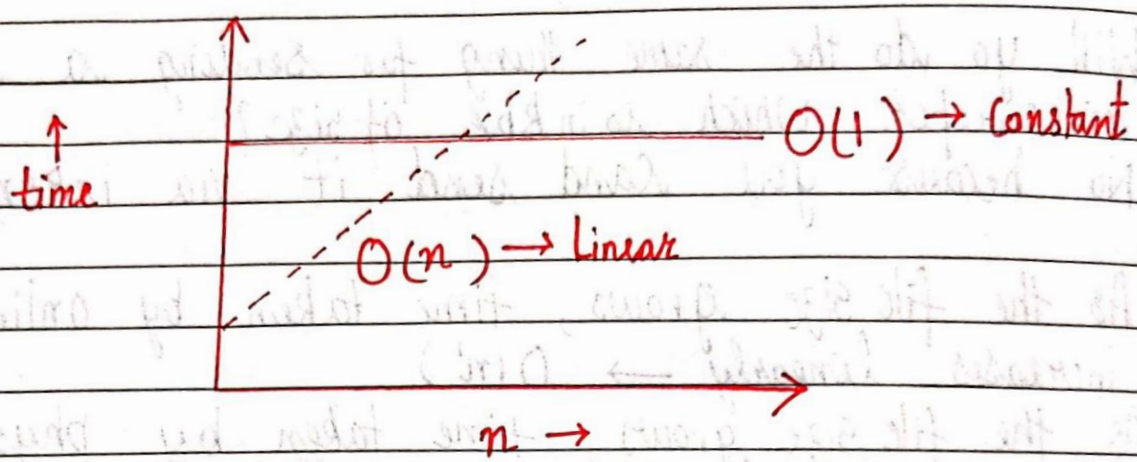
Algo 1 → $k_1 n^2 + k_2 n + 36$ ⇒ $O(n^2)$
($k_1 n^2$: Highest order term; $k_2 n + 36$: can ignore lower order terms)

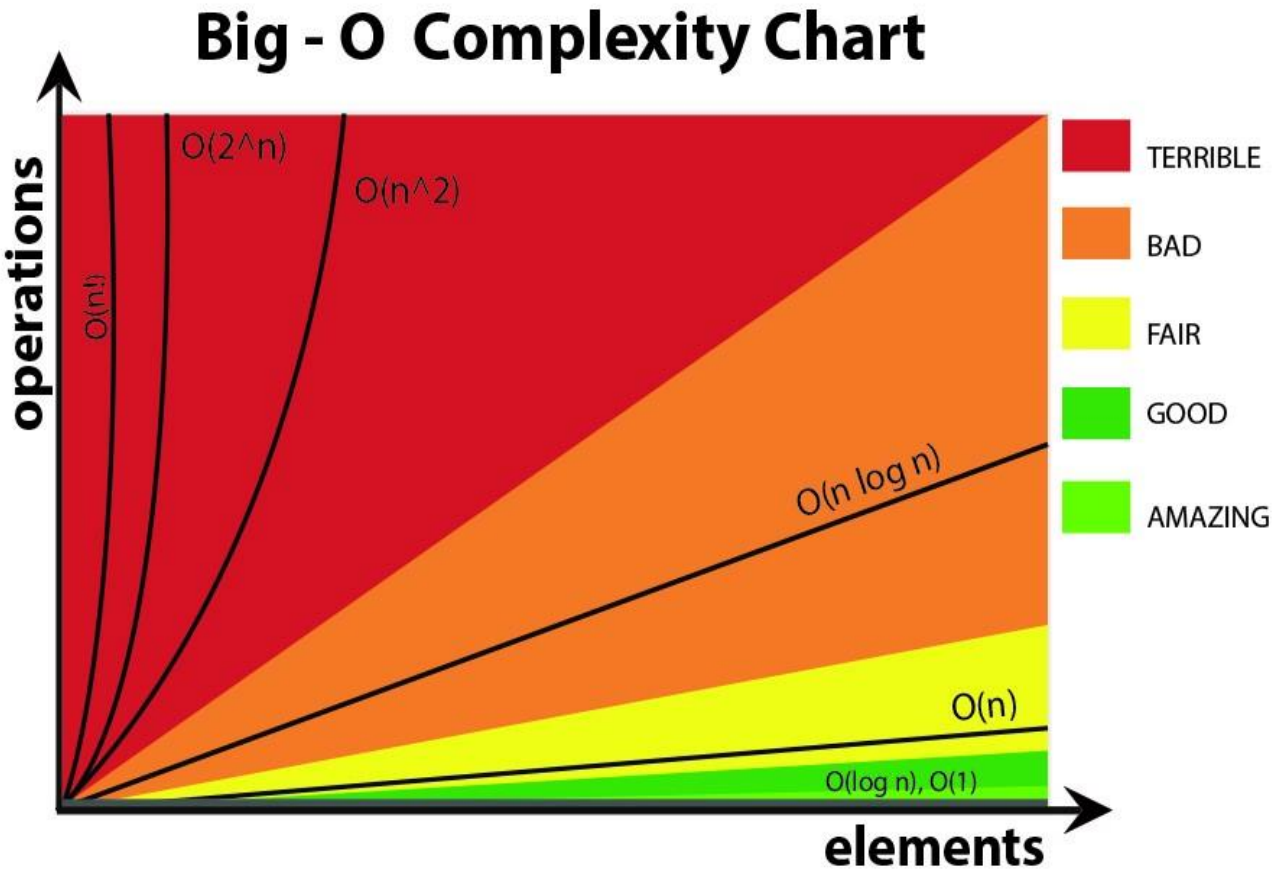
Algo 2 → $k_1 k_2^2 + k_3 k_2 + 8$
⇓
$k_1 k_2^2 n^0 + k_3 k_2 + 8$ ⇒ $O(n^0)$ or $O(1)$

Note that these are the formulas for time taken by them.

Visualising Big O

If we were to plot O(1) and O(n) on a graph, they will look something like this :





Source: https://stackoverflow.com/questions/3255/big-o-how-do-you-calculate-approximate-it

Asymptotic Notations

Asymptotic notations give us an idea about how good a given algorithm is compared to some other algorithm
Let us see the mathematical definition of "order of" now.

Primarily there are three types of widely used asymptotic notations.

1) Big oh notation (O)
2) Big omega notation (Ω)
3) Big theta notation (Θ) ⟶ Widely used one!

Big oh notation

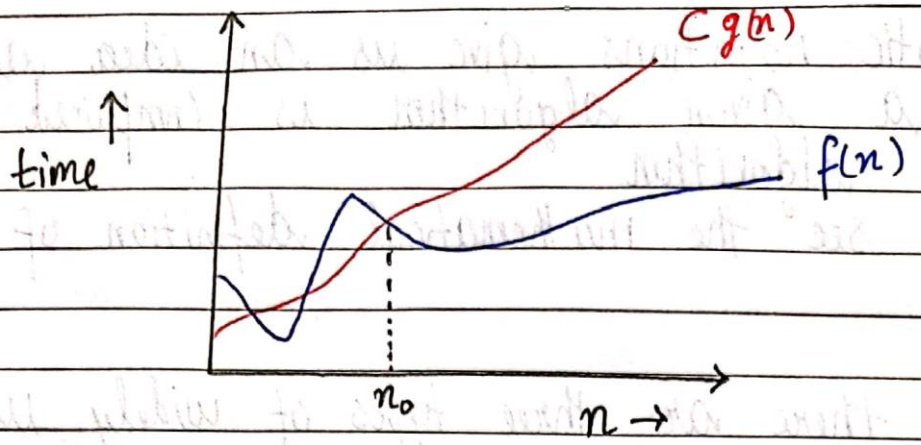
Big oh notation is used to describe asymptotic upper bound
Mathematically, if $f(n)$ describes running time of an algorithm; $f(n)$ is $O(g(n))$ iff there exist positive constants C and n_0 such that

$$0 \leq f(n) \leq C\,g(n) \quad \text{for all } n \geq n_0$$

⇓
used to give upper bound on a function.

if a function is $O(n)$, it is automatically $O(n^2)$ as well!

Graphic example for Big oh (O)



Big Omega notation

Just Like O notation provides an asymptotic upper bound, Ω notation provides asymptotic lower bound. Let $f(n)$ define running time of an algorithm;

$f(n)$ is said to be $\Omega(g n)$ if there exists positive constants C and n_0 such that

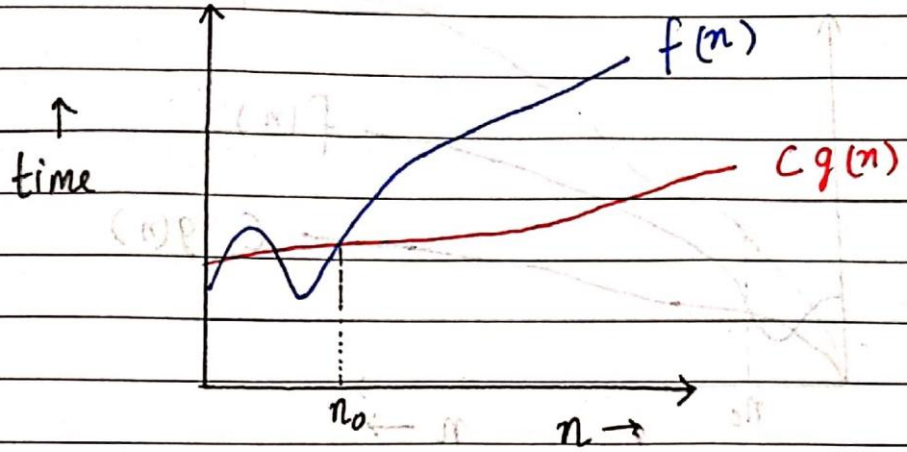
$$0 \leq c\,g(n) \leq f(n) \quad \text{for all } n \geq n_0$$

⇓ used to give lower bound on a function

if a function is $O(n^2)$ it is automatically $O(n)$ as well

URBAN EDGE

Graphic example for Big omega (Ω)



Big theta notation

Let $f(n)$ define running time of an algorithm

$f(n)$ is said to be $\Theta(gn)$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

Mathematically,

$$0 \leq f(n) \leq C_1 g(n) \quad \forall\ n \geq n_0$$

$$0 \leq C_2 g(n) \leq f(n) \quad \forall\ n \geq n_0$$

Sufficiently large value of n

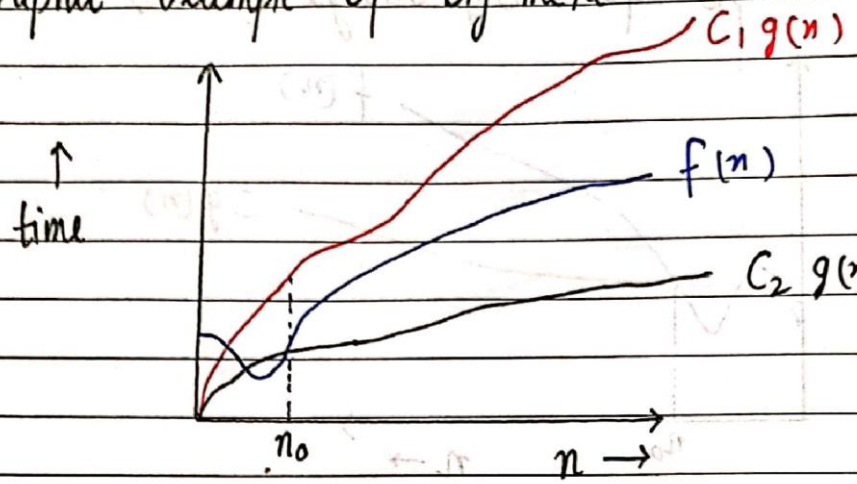
Merging both the equations, we get:

$$0 \leq C_2 g(n) \leq f(n) \leq C_1 g(n) \quad \forall\ n \geq n_0$$

The equation simply means there exist positive constants C_1 and C_2 such that $f(n)$ is sandwiched between $C_2 g(n)$ and $C_1 g(n)$

URBAN EDGE

Graphic example of Big theta



Which one of these to use ?
Since Big theta gives a better picture of runtime for a given algorithm, most of the interviewers expect you to provide an answer in terms of Big theta when they say "Order of".

Quick Quiz :- Prove that n^2+n+1 is $O(n^3)$, $\Omega(n^2)$ and $\Theta(n^2)$ using respective definitions.

Increasing order of common runtimes

$$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n^n$$

Better ⟵ ⇓ ⟶ Worse

Common runtimes from better to worse ⟶

URBAN EDGE

Best, worst and Expected Case

Sometimes we get lucky in life. Exams cancelled when you were not prepared, surprise test when you were prepared etc. ⟹ Best case

Some times we get unlucky. Questions you never prepared asked in exams, rain during sports period etc. ⇒ Worst case

But overall the life remains balance with the mixture of lucky and unlucky times. ⇒ Expected case.

Analysis of a search algorithm

Consider an array which is sorted in increasing order

1	7	18	28	50	180

We have to search a given number in this array and report whether its present in the array or not.

Algo 1 → Start from first element until an element greater than or equal to the number to be searched is found.

Algo 2 → Check whether the first or the last element is equal to the number. If not find the number between these two elements (center of the array). If the center element is greater than the number to be searched, repeat the process for first half else repeat for second half until the number is found.

Analyzing Algo 1

If we really get lucky, the first element of the array might turn out to be the element we are searching for. Hence we made just one comparison.

$$\text{Best case Complexity} = O(1)$$

If we are really unlucky, the element we are searching for might be the last one.

$$\text{Worst case Complexity} = O(n)$$

For calculating Average case time, we sum the list of all the possible case's runtime and divide it with the total number of cases.

⇓

Sometimes calculation of average case time gets very complicated

Analyzing Algo 2

If we get really lucky, the first element will be the only one which gets compared

$$\text{Best case Complexity} = O(1)$$

If we get unlucky we will have to keep dividing the array into halves until we get a single element (the array gets finished)

Worst case Complexity = $O(\log n)$

What log(n)? What is that

log(n) → Number of times you need to half the array of size n before it gets exhausted

$\log 8 = 3 \implies \frac{8}{2} \to \frac{4}{2} \to \frac{2}{2}$ → Cant break anymore.

$1 + 1 + 1$

$\log 4 = 2 \implies \frac{4}{2} \to \frac{2}{2}$ → Cant break anymore.

$1 + 1$

Log n simply means how many time I need to divide n units such that we cannot divide them (into halves) anymore.

Space Complexity

Time is not the only thing we worry about while analyzing algorithms. Space is equally important.

Creating an array of size n → O(n) space
↳ Size of input

If a function calls itself recursively n times its space complexity is O(n)

URBAN EDGE

Quick Quiz → Calculate Space Complexity of a function which calculates factorial of a given number n.

Why cant we calculate Complexity in seconds?

→ Not everyone's Computer is equally powerful

→ Asymptotic analysis is the measure of how time (runtime) grows with input

Techniques to Calculate Time Complexity

Once we are able to write the runtime in terms of size of the input (n), we can find the time complexity.

For example $T(n) = n^2 \Rightarrow O(n^2)$

$T(n) = \log n \Rightarrow O(\log n)$

Some tricks to calculate complexity

1. Drop the constants : Any thing you might think is $O(3n)$ is $O(n)$
   ↳ Better representation
2. Drop the non dominant terms : Any thing you represent as $O(n^2+n)$ can be written as $O(n^2)$
3. Consider all variables which are provided as input : $O(mn)$ & $O(mnq)$ might exist for some cases!

In most of the cases, we try to represent the runtime in terms of the input which can be more than one in number. For example :

Painting a park of dimension $m \times n \Rightarrow O(mn)$

Time Complexity – Competitive Practice Sheet

1. Fine the time complexity of the func1 function in the program show in program1.c as follows:

```c
#include <stdio.h>

void func1(int array[], int length)
{
    int sum = 0;
    int product = 1;
    for (int i = 0; i < length; i++)
    {
        sum += array[i];
    }

    for (int i = 0; i < length; i++)
    {
        product *= array[i];
    }
}

int main()
{
    int arr[] = {3, 5, 66};
    func1(arr, 3);
    return 0;
}
```

2. Fine the time complexity of the func function in the program from program2.c as follows:

```c
void func(int n)
{
    int sum = 0;
    int product = 1;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            printf("%d , %d\n", i, j);
        }
    }

}
```

3. Consider the recursive algorithm above, where the random(int n) spends one unit of time to return a random integer which is evenly distributed within the range [0,n][0,n]. If the average processing time is T(n), what is the value of T(6)?

```
int function(int n)
{
    int i;

    if (n <= 0)
    {
        return 0;
    }
    else
    {
        i = random(n - 1);
        printf("this\n");
        return function(i) + function(n - 1 - i);
    }
}
```

4. Which of the following are equivalent to O(N)? Why?
    a) O(N + P), where P < N/9
    b) 0(9N-k)
    c) O(N + 8log N)
    d) O(N + M^2)

5. The following simple code sums the values of all the nodes in a balanced binary search tree. What is its runtime?

```
int sum(Node node)
{
    if (node == NULL)
    {
        return 0;
    }
    return sum(node.left) + node.value + sum(node.right);
}
```

6. Find the complexity of the following code which tests whether a give number is prime or not?

```
int isPrime(int n){
    if (n == 1){
        return 0;
    }

    for (int i = 2; i * i < n; i++) {
        if (n % i == 0)
            return 0;
    }
```

```
    return 1;
}
```

7. What is the time complexity of the following snippet of code?

```
int isPrime(int n){

    for (int i = 2; i * i < 10000; i++) {
        if (n % i == 0)
            return 0;
    }

    return 1;
}
isPrime();
```

URBAN EDGE

Operations on an Array

Following operations are supported by an array.

Traversal
Insertion
Deletion
Search

⇒ There can be many other operations one can perform on arrays as well!
eg: Sorting asc., Sorting desc.

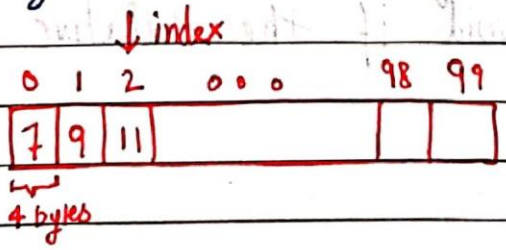
Traversal

Visiting every element of an array once → Traversal

Why traversal? → For use cases like:
→ Storing all elements → using scanf
→ Printing all elements → using printf

An important note about arrays
If we create an array of length 100 using a[100] in C language, we need not use all the elements. It is possible for a program to use just 60 elements out of these 100.
↳ But we cannot go beyond 100 elements.

An array can easily be traversed using a for loop in C language

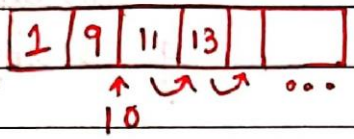


URBAN EDGE

Insertion

An element can be inserted in an array at a specified position.

In order for this operation to be successful, the array should have enough capacity

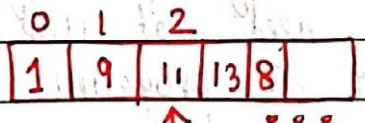


⇒ Elements need to be shifted to maintain relative order.

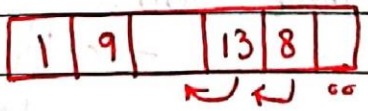
When no position is specified its best to insert the element at the end.

Deletion

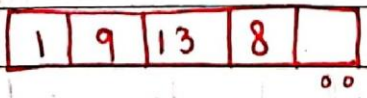
An element at specified position can be deleted creating a void which needs to be fixed by shifting all the elements to the left as follows:



Delete 11 at ind 2



Shift the elements

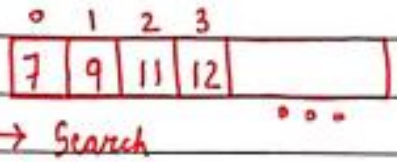


Deletion done!

We can also bring the last element of the array to fill the void if the relative ordering is not important

Searching

Searching can be done by traversing the array until the element to be searched is found



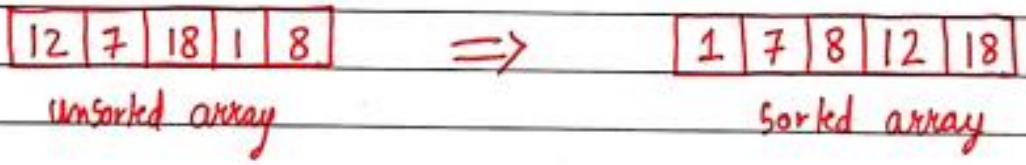
⇓

For sorted array time taken to search is much less than unsorted array!!

Sorting

Sorting means arranging an array in order (asc or desc)

We will see various sorting techniques later in the course.



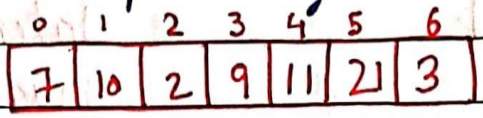
Unsorted array

Sorted array

Linear Vs Binary Search

Linear Search

Searches for an element by visiting all the elements sequentially until the element is found.



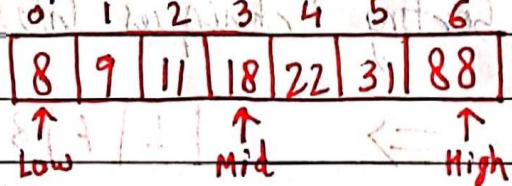
⇒ can be sorted or unsorted

Search '2' → Element found

WC Complexity : O(n)

Binary Search

Searches for an element by breaking the search space into half in a sorted array.



WC Complexity O(log n)

Search 18

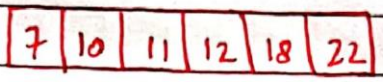
The search continues towards either side of mid based on whether the element to be searched is lesser or greater than mid.

	Linear Search	Binary Search
1.	Works on both sorted and unsorted arrays	Works only on sorted arrays
2.	Equality operations	inequality operations
3.	O(n) WC Complexity	O(log n) WC Complexity

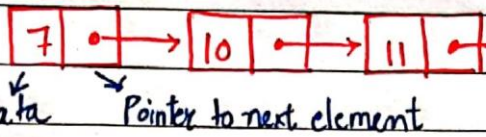
URBAN EDGE

Introduction to Linked Lists

Linked lists are similar to arrays (Linear data structures)



⇒ In Arrays elements are stored in contiguous memory locations



⇒ In linked lists, elements are stored in non contiguous memory locations

Why Linked Lists?

Memory and the capacity of an array remains fixed.
In case of linked lists, we can keep adding and removing elements without any capacity constraints

Drawbacks of Linked lists

→ Extra memory space for pointers is required (for every node 1 pointer is needed)
→ Random access not allowed as elements are not stored in contiguous memory locations.

Implementation

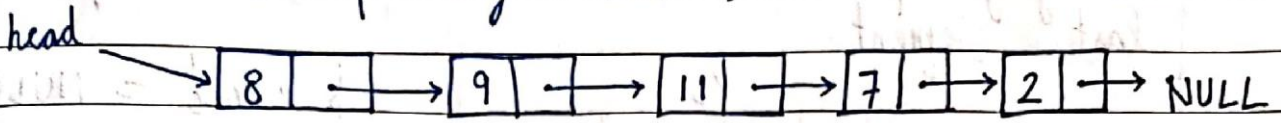
Linked list can be implemented using a structure in C language

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

⇒ Self refencing structure

Deletion in a Linked List

Consider the following Linked List

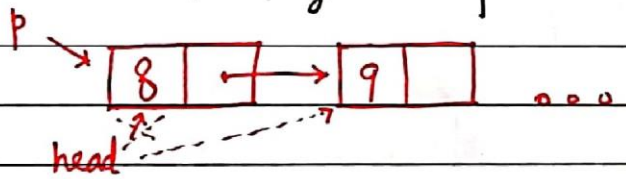


Deletion can be done for the following cases:

1. Deleting the first Node
2. Deleting the node at an index
3. Deleting the last Node
4. Deleting the first node with a given value.

The deletion just like insertion is done by rewiring the pointer connections, the only caveat being: we need to free the memory of the deleted node using free().

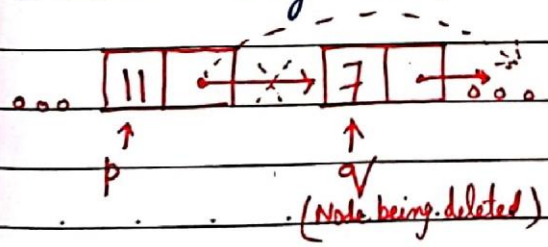
Case 1: Deleting the first node




```
Struct Node * p = head
head = head → next;
free(p);
```

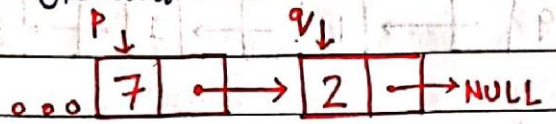
Case 2: Deleting the node at an index

for deleting a given node, we first bring a temporary pointer p before element to be deleted and q on the element being deleted



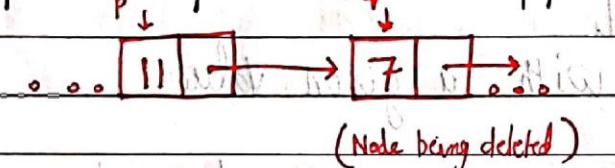

```
p → next = q → next;
free(q);
```

Case 3: Deleting the last Node
Last node can be deleted just like Case 2 by bringing p on second last element and q on last element.



p→next = NULL
free(q) → To free the memory!

Case 4: Delete the first node with a given value
This can be done exactly like Case 2 by bringing pointers p & q to appropriate positions

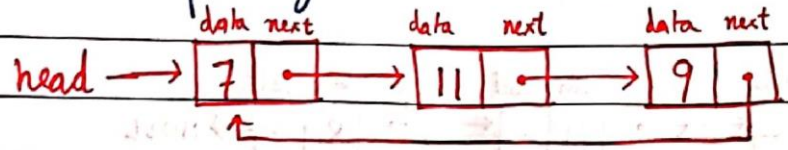


p→next = q→next;
free(q);

URBAN EDGE

Circular Linked List

A circular linked list is a linked list where the last element points to the first element (head) hence forming a circular chain



Operations on a circular linked List
operations on a circular linked lists can be performed exactly like a singly linked list.

Visit www.codewithharry.com for practice sets / code / more

URBAN EDGE

Doubly Linked List

In a doubly linked list, each node contains a data part along with the two addresses, one for the previous node and the other one for the next node

Implementation

A doubly linked list can be implemented in C language as follows:

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

Operations on a Doubly Linked List

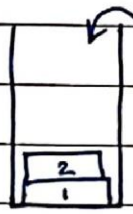
The insertion and deletion on a Doubly linked list can be performed by rewiring pointer connections just like we saw in a Singly linked list.

The difference here lies in the fact that we need to adjust two pointers (prev & next) instead of one (next) in the case of a Doubly linked list.

URBAN EDGE

Introduction to Stack Data Structure

Stack is a linear data structure. Operations on stack are performed in LIFO (Last in first out) order.



⇒ Item 2 which entered the basket last will be the first one to come out

LIFO (Last in first out)

Applications of Stack

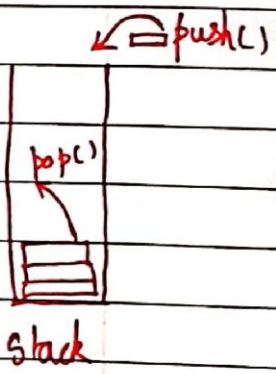
1. Used in function calls
2. Infix to postfix conversion (and other similar conversions)
3. Parenthesis matching & more...

Stack ADT

In order to create a stack we need a pointer to the topmost element along with other elements which are stored inside the stack.

Some of the operations of stack ADT are:

1. push() → push an element into the stack
2. pop() → remove the topmost element from the stack
3. peek(index) → Value at a given position is returned
4. isEmpty() / isFull() → Determine whether the stack is empty or full.



Implementation

A stack is a collection of elements with certain operations following LIFO (Last in First out) dicipline.

A Stack can be implemented using an array or a linked list