Data Structure and Algorithm

Prof. Kanak Kalyani

Points to discuss

- Knowledge of C language
  - Function, Structures, Pointers

- Online Classes
  - 45 minutes lecture
  - Attendance is mandatory
  - MCQ or interactive session in between
  - Assignments will be uploaded on Google Classroom
  - PPTs will be provided at end of unit

<u>Lab Courses will be offline, on Wednesday and Thursday</u>

Basics of Programming

Characteristics of a good program

- runs correctly
- is easy to read and understand
- is easy to debug *and*
- is easy to modify.

Data Structure

- Data management is a complex
  - Collecting the data
  - Organizing the data
  - Retrieving correct information
- Data + STRUCTURE
  - Organizing the data in such a way that retrieval is fast and efficient

CLASSIFICATION OF DATA STRUCTURES

- ***Primitive***
  - integer, real, character, and boolean
- ***Non-primitive Data Structures***
  - linked lists, stacks, trees, and graphs
  - ***Linear and Non-linear Structures***
    - Linear: Sequential memory locations,
      - Example: Array, Linked List, Stacks, Queues
    - Non-Linear: Trees, Graphs

OPERATIONS ON DATA STRUCTURES

- ***Traversing***
- ***Searching***
- ***Inserting***
- ***Deleting***
- ***Sorting***
- ***Merging***

Syllabus
CDT201: DSA

UNIT – I: Data Structures and Algorithms Basics

Introduction: basic terminologies, elementary data organizations, data structure operations; abstract data types (ADT) and their characteristics.

Algorithms: definition, characteristics, analysis of an algorithm, asymptotic notations, time and space trade-offs.

Array ADT: definition, operations and representations – row-major and column-major.

UNIT – II: Stacks and Queues

Stack ADT: allowable operations, algorithms and their complexity analysis, applications of stacks – expression

conversion and evaluation (algorithmic analysis), multiple stacks.

Queue ADT: allowable operations, algorithms and their complexity analysis for simple queue and circular queue, introduction to double-ended queues and priority queues.

Syllabus
CDT201: DSA

UNIT – III: Linked Lists

Singly Linked Lists: representation in memory, algorithms of several operations: traversing, searching,

insertion, deletion, reversal, ordering, etc.

Doubly and Circular Linked Lists: operations and algorithmic analysis.

Linked representation of stacks and queues, header node linked lists

UNIT – IV: Sorting and Searching

Sorting: different approaches to sorting, properties of different sorting algorithms (Insertion, Shell, quick, merge, heap, counting), performance analysis and comparison.

Searching: necessity of a robust search mechanism, searching linear lists (linear search, binary search) and complexity analysis of search methods.

Syllabus
CDT201: DSA

UNIT – V: Trees

Trees: basic tree terminologies, binary tree and operations, binary search tree [BST] and operations with time

analysis of algorithms, threaded binary trees.

Self-balancing Search Trees: tree rotations, AVL tree and operations, B+-tree: definitions, characteristics, and

operations (introductory).

UNIT – VI: Graphs and Hashing

Graphs: basic terminologies, representation of graphs, traversals (DFS, BFS) with complexity analysis, path

finding (Dijkstra's SSSP, Floyd's APSP), and spanning tree (Prim's method) algorithms.

Hashing: hash functions and hash tables, closed and open hashing, randomization methods (division method,

mid-square method, folding), collision resolution techniques.

Course Outcome

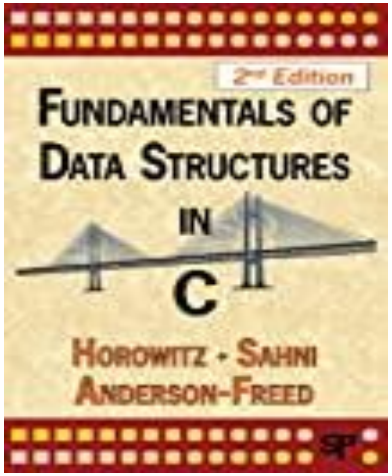
On completion of the course the student will be able to:

1. Design and realize different linear data structures.
2. Identify and apply specific methods of searching and sorting to solve a problem.
3. Implement and analyze operations on binary search trees and AVL trees.
4. Implement graph traversal algorithms, find shortest paths and analyze them.

Books

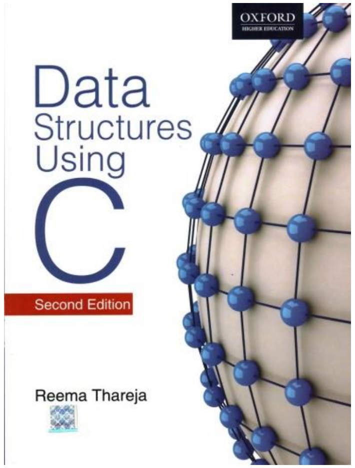
Fundamentals of Data Structures in C

By Ellis Horowitz, Sartaj Sahni & Susan Anderson-Freed



Data Structures Using C

By REEMA THAREJA



Algorithm

- An algorithm is a finite set of instructions that, if followed , accomplishes a particular task.
- In addition, all algorithms must satisfy the following criteria:
  1. Input
  2. Output
  3. Definiteness
  4. Finiteness
  5. Effectiveness

ALGORITHM SPECIFICATION

Pseudocode Convention

- Comment: //
- Block: { and }
- Identifier begins with LETTER
- Datatype of variables are not explicitly declared
- Assignment:- <variable> = <expression>
- Arrays:-
  - Single dimensional [i]
  - Two Dimensional [i,j]
  - Multi Dimensional [i, j, …]

ALGORITHM SPECIFICATION

- Loops

while ⟨*condition*⟩ **do**
{
    ⟨*statement* 1⟩
        ⋮
    ⟨*statement n*⟩
}

while $((variable - fin) * step \leq 0)$ **do**
{
    ⟨*statement* 1⟩
        ⋮
    ⟨*statement n*⟩
    $variable := variable + incr;$
}

for *variable* := *value*1 **to** *value*2 **step** *step* **do**
{
    ⟨*statement* 1⟩
        ⋮
    ⟨*statement n*⟩
}

repeat
    ⟨*statement* 1⟩
        ⋮
    ⟨*statement n*⟩
until ⟨*condition*⟩

ALGORITHM SPECIFICATION

- IF Condition

if $\langle condition \rangle$ **then** $\langle statement \rangle$
if $\langle condition \rangle$ **then** $\langle statement\ 1 \rangle$ **else** $\langle statement\ 2 \rangle$

- Switch Statement

```
case
{
      :⟨condition 1⟩: ⟨statement 1⟩
                 ⋮
      :⟨condition n⟩: ⟨statement n⟩
      :else: ⟨statement n + 1⟩
}
```

Example - 1

- Write an algorithm to find maximum of “n” given number

```
Algorithm Max(A, n)
// A is an array of size n.
{
    Result := A[1];
    for i := 2 to n do
        if A[i] > Result then Result := A[i];
    return Result;
}
```

Example - 2

Write an algorithm to sort a given list of numbers using selection sort

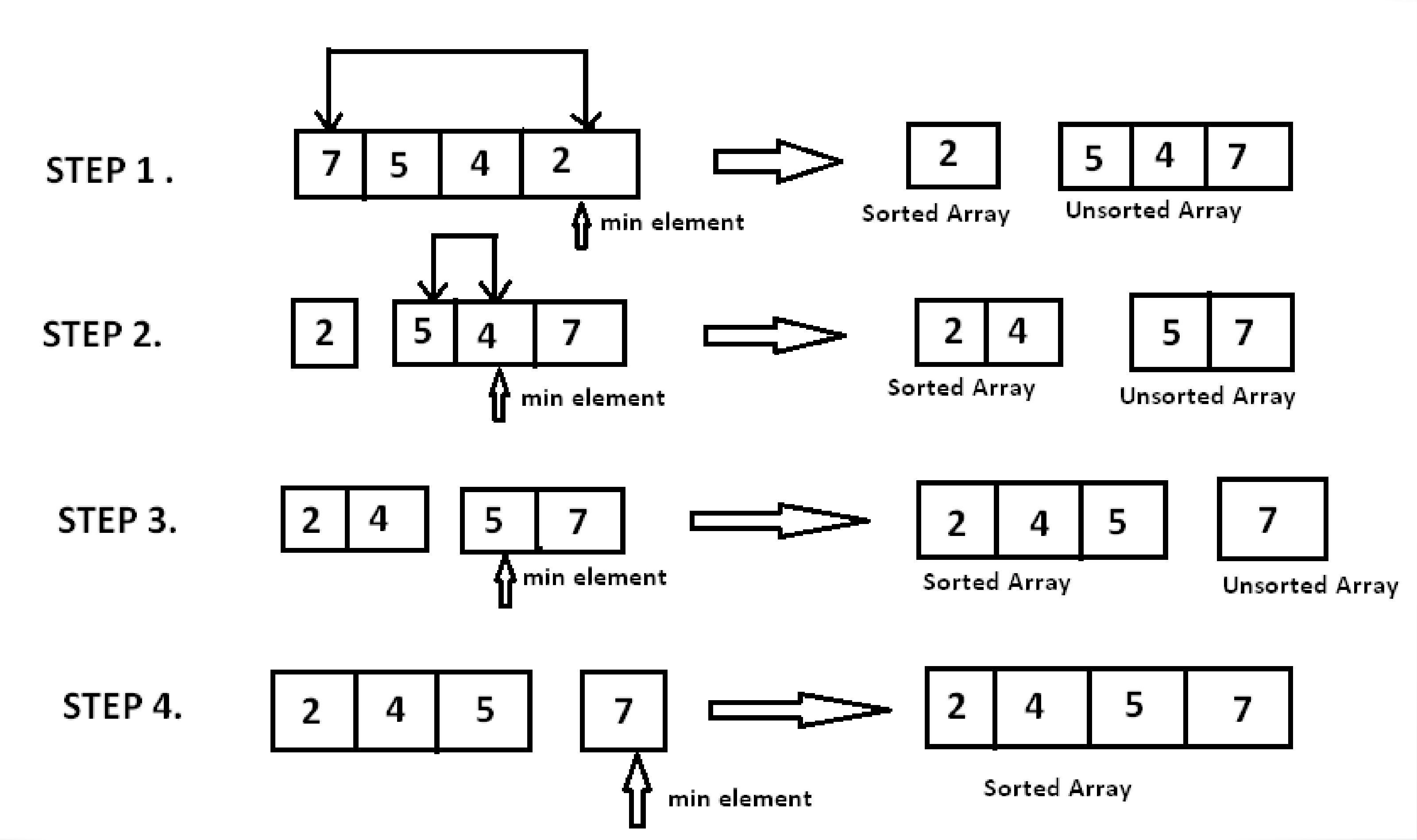
Step-1: Find Minimum Value in the List

Step-2: Swap it with the value in current position

Step-3: Repeat the process until all the numbers of list are traversed

Example-2

Consider the list:



Example-2

- Algorithm

```
Algorithm SelectionSort(a, n)
// Sort the array a[1 : n] into nondecreasing order.
{
    for i := 1 to n do
    {
        j := i;
        for k := i + 1 to n do
            if (a[k] < a[j]) then j := k;
        t := a[i]; a[i] := a[j]; a[j] := t;
    }
}
```

Another way of writing an Algorithm

Write an algorithm to find the sum of first N natural numbers.

Step 1: Input N

Step 2: SET I = 1, SUM = 0

Step 3: Repeat Step 4 while I <= N

Step 4: SET SUM = SUM + I

SET I = I + 1

[END OF LOOP]

Step 5: PRINT SUM

Step 6: END

Recursive Algorithm

- Every Recursive Problem has two MAJOR cases

- CASE -1 : **BASE Case**:- START of problem where the problem can be solved without calling it again
    - CAN BE CALLED AS TERMINATION CASE
- CASE-2 : **Recursive Case**:-
    - Problem is subdivided into simpler sub-parts
    - Function is called with subpart
    - Result is obtained by combining the subparts

Example-3

- Finding Factorial of a number

PROBLEM	SOLUTION
$5!$	$5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4!$	$= 5 \times 4 \times 3 \times 2 \times 1$
$= 5 \times 4 \times 3!$	$= 5 \times 4 \times 3 \times 2$
$= 5 \times 4 \times 3 \times 2!$	$= 5 \times 4 \times 6$
$= 5 \times 4 \times 3 \times 2 \times 1!$	$= 5 \times 24$
	$= 120$

- Base Case:- 1! = 1
- Recursive Case:- factorial(n) = n × factorial (n–1)

Example-3

Algorithm for finding Factorial of number

```
Algorithm factorial(n)
{
        if(n<=1)
                return 1
        else
                fact=n * factorial(n-1)
        return fact
}
```

- Write an recursive algorithm to find Fibonacci series

- Give an algorithm to solve the following problem: Given n, a positive integer, determine whether n is the sum of all of its divisors, that is, whether n is the sum of all t such that $1 \leq t < n$, and t divides n.

Fibonacci Series

```
Algorithm Fibonacci(n)
{
    f0=0
    f1 = 1
    print f0 , f1
    for i:= 1 to n-1 do
    {
        fib:=  f0 + f1
        f0 :=  f1
        f1 := fib
        print fib
    }
}
```

```
Algorithm Rec_Fibonacci(n)
{

  if ((n = 1) then
          return 1
  else if(n=0)
          return  0
  else
          return
          Rec_Fibonacci(n-1)
          +Rec_Fibonacci(n-2)

}
```

Analysis of Algorithm

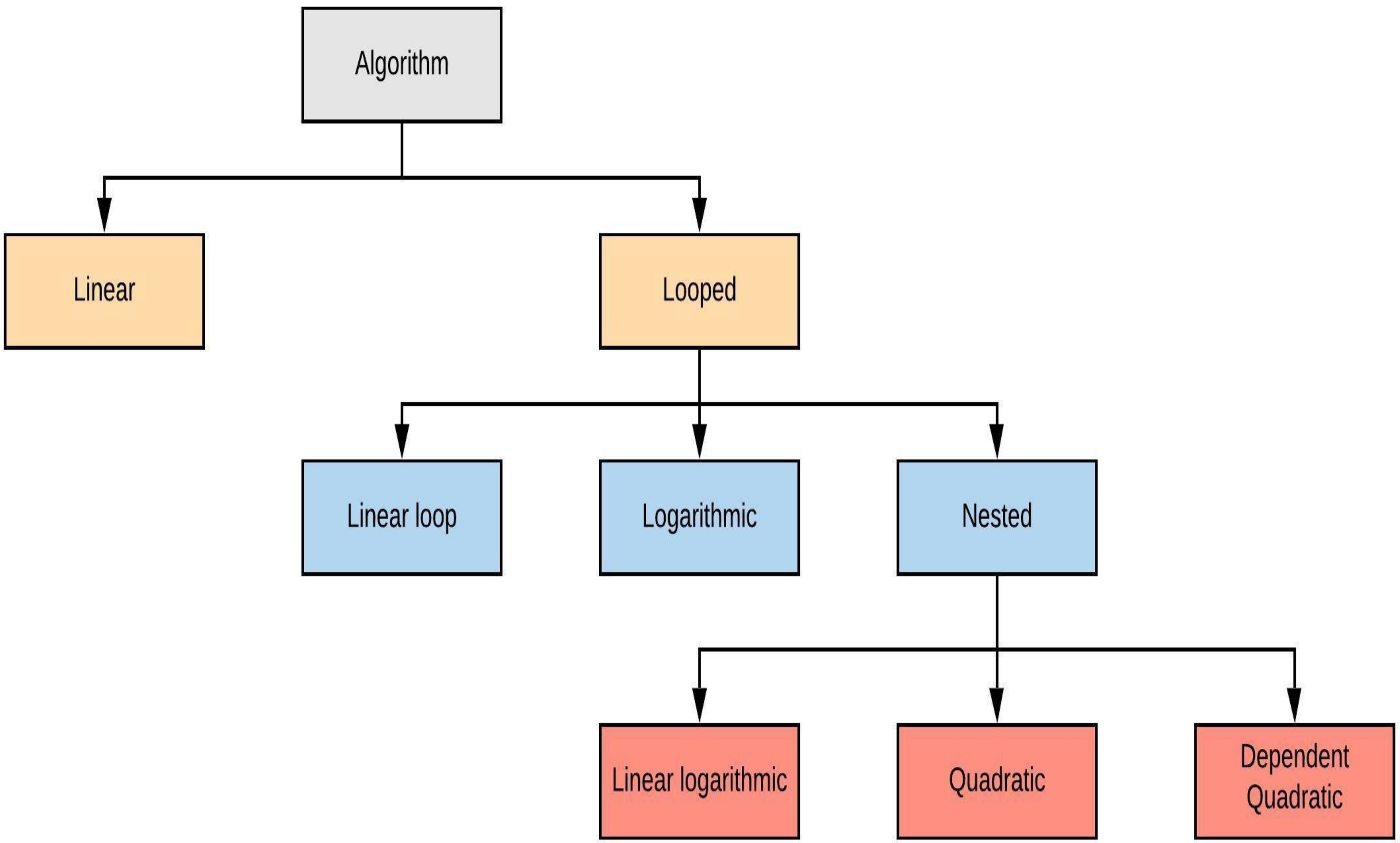
- **Time Complexity**
  - Best Case
  - Average Case
  - Worst Case
  - Amortized Case
- **Space Complexity**
  - Fixed Part
  - Variable Part

How to find Complexity?

- Linear Function: No Loops, No call to functions
  - Running Time = Number of Instructions

- Loops
  - Running Time depends upon
    - number of loops
    - Complexity of loops

Analyzing Time Complexity



Analyzing Time Complexity

Linear Loops

```
for(i=0;i<100;i++)
        statement block;
```

➔ f(n) = n

```
for(i=0;i<100; i+=2)
        statement block;
```

➔ f(n) = n/2

Logarithmic Loops

```
for(i=1;i<1000;i*=2)
    statement block;
for(i=1000;i>=1;i/=2)
    statement block
```

➔ f(n) = log n

Analyzing Time Complexity: ***Nested Loops***

Linear logarithmic loop

```
for(i=0;i<10;i++)
        for(j=1; j<10;j*=2)
                statement block;
```

➔ $f(n) = n \log n$

Quadratic loop

```
for(i=0;i<10;i++)
        for(j=0; j<10;j++)
                statement block;
```

➔ $f(n) = n^2$

Dependent quadratic loop

```
for(i=0;i<10;i++)
        for(j=0; j<=i; j++)
                statement block;
```

➔ $f(n) = n\,(n + 1)/2$

$1 + 2 + ... + 9 + 10 = 55$

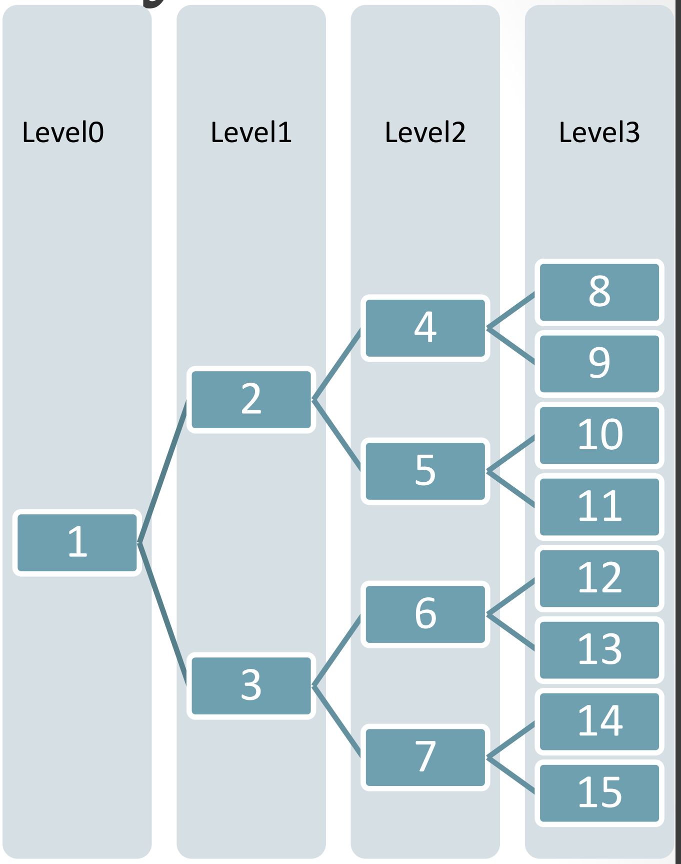
Arithmetic Series: $1 + 2 + 3 + ... + (n-2) + (n-1) = (n-1)(n)/2$

Analyzing Time Complexity

Example 1:

```
int i = 1;
while (i <= n)
{
        System.out.println("*");
        i = 2 * i;
}
```

No. of times * prints $1 + \log_2 n$



Analyzing Time Complexity

Example 2:

```
for (int i = 1; i <=m; i += c)  {    Statement-1   }
for (int i = 1; i <=n; i += c) {   Statement-2 }
```

Complexity:= m + n

if n=m then 2n

Example 3:

```
int i = n;
while (i > 0) {
        for (int j = 0; j < n; j++)
                System.out.println("*");
        i = i / 2; }
```

Outer Loop: $\log_2 n$

Inner Loop: n

$F(n) = n * \log_2 n$

Analyzing Time Complexity

Example 4:

```
for (int i = 0; i < n; i++) // loop 1
        for (int j = i+1; j > i; j--) // loop 2
                for (int k = n; k > j; k--) // loop 3
                        System.out.println("*");
```

Loop1:- n

Loop2:- 1

Loop 3:- Dependent

Follows arithmetic Series:- 1 + 2 + 3 + ... + (n-2) + (n-1)

Total= $(n-1)(n)/2$

Notations for Complexity

BIG O Notation: O

- a dominant factor in the expression is sufficient to determine the order of the magnitude of the result
- O stands for ‘order of’
- When using the Big O notation, constant multipliers are ignored
- If f(n) and g(n) are the functions defined on a positive integer number n, then

$$f(n) = O(g(n))$$

if and only if positive constants c and n exist,

$$f(n) <= cg(n).$$

- TIGHT UPPER BOUND

BIG O Notation: O

Constant "c" which depends upon the following factors

- t
- t
- t
- t ... it,
- t
- t ... also
  t

g(n)	f(n) = O(g(n))
10	O(1)
$2n^3 + 1$	$O(n^3)$
$3n^2 + 5$	$O(n^2)$
$2n^3 + 3n^2 + 5n - 10$	$O(n^3)$

- f(n) = O(g(n))

Example

- $O(n^3)$ will include ➔ $n^{2.9}$, n^3, $n^3 + n$, $540n^3 + 10$.
- $O(n^3)$ will not include ➔ $n^{3.2}$, n^2, $n^2 + n$, $540n + 10$, $2n$

BIG O Notation: O

- Best case O describes an upper bound for all combinations of input.
- It is possibly lower than the worst case.
- For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case O describes a lower bound for worst case input combinations. It is possibly greater than the best case.
- For example, when sorting an array the worst case is when the array is sorted in reverse order.
- **If we simply write O, it means same as worst case O.**

Limitations of Big O Notation

- Many algorithms are simply too hard to analyse mathematically.
- There may not be sufficient information to calculate the behavior of the algorithm in the average case.
- Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider the programming effort.
- It ignores important constants.
- For example, if one algorithm takes $O(n^2)$ time to execute and the other takes $O(100000n^2)$ time to execute, then as per Big O, both algorithm have equal time complexity, but this may be a serious consideration.

Categories of Algorithms

According to the Big O notation, we have five different categories of algorithms:

- **Constant time algorithm: O(1)**
- **Linear time algorithm: O(n)**
- **Logarithmic time algorithm: as O(log n)**
- **Polynomial time algorithm: $O(n^k)$ where k > 1**
- **Exponential time algorithm: as $O(2^n)$**

OMEGA NOTATION (Ω)

- provides a tight lower bound for $f(n)$
- Ω notation is simply written as, $f(n) \in \Omega(g(n))$
- $\Omega(g(n)) = \{h(n): \exists$ positive constants $c > 0, n0$ such that

  $$0 \le cg(n) \le h(n), \forall\, n \ge n0\}.$$

- If $cg(n) \le f(n)$, $c > 0$, $\forall\, n \ge n_0$, then $f(n) \in \Omega(g(n))$ and $g(n)$ is an asymptotically tight lower bound for $f(n)$.

- Examples of functions in $\Omega(n^2)$ include: n^2, $n^{2.9}$, $n^3 + n^2$, n^3
- Examples of functions not in $\Omega(n^3)$ include: n, $n^{2.9}$, n^2

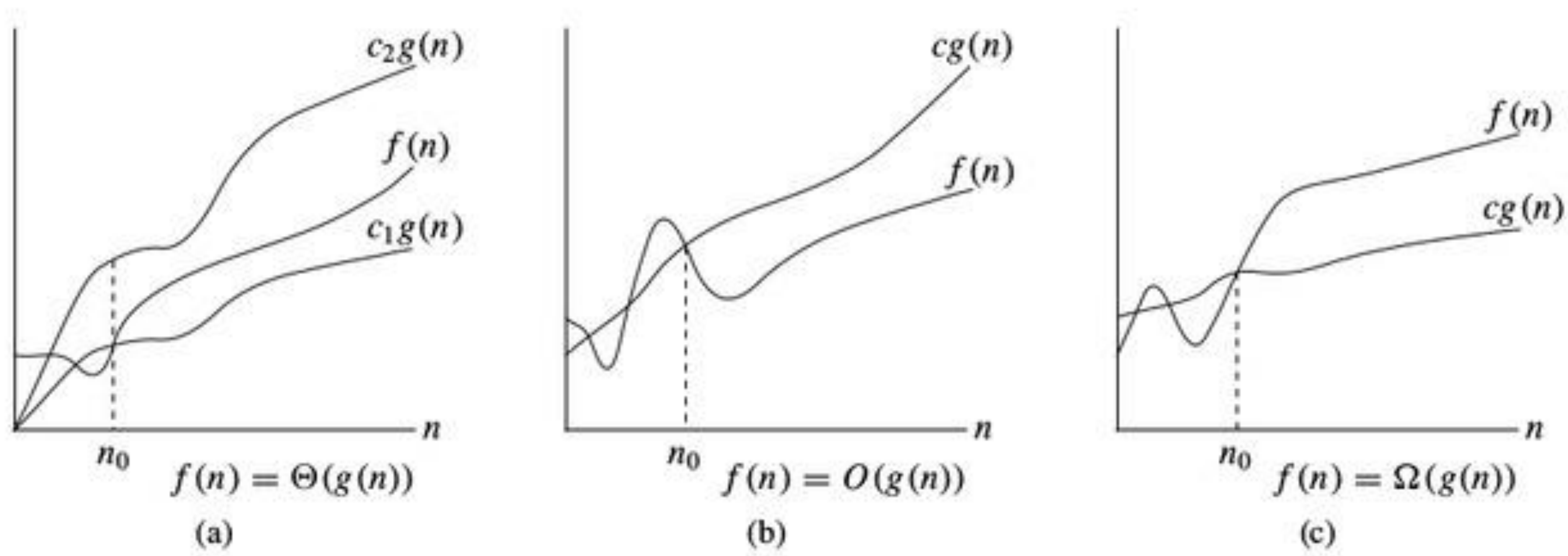
- Best case Ω describes a lower bound for all combinations of input
- Worst case Ω describes a lower bound for worst case input combinations
- **If we simply write Ω, it means same as best case Ω.**

THETA NOTATION (Θ)

- Theta notation provides an asymptotically tight bound for f(n).
- Θ notation is simply written as, $f(n) \in \Theta(g(n))$
- $\Theta(g(n)) = \{h(n): \exists$ positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le h(n) \le c_2 g(n), \forall\ n \ge n_0\}$.

- The best case in Θ notation is not used.
- • Worst case Θ describes asymptotic bounds for worst case combination of input values.
- • If we simply write Θ, it means same as worst case Θ.

Time Complexity



QUIZ TIME

- Time : 5 Minutes

- URL: https://forms.gle/mu9mvwALbNgbzVuf6

Abstract Data Type

- Data Type: int, float, char etc
  - What can we store
  - Operations that can be performed
- Abstract:
  - No implementation is specified
  - Generalized

- An abstract data type (ADT) is the specification of a data type within some language, **independent of an implementation**.
- An ADT does not specify *how* the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as *encapsulation*.


Real world Example
smartphone
Abstract/logical view
- 4 GB RAM
- Snapdragon 2.2GHz processor
- 5.5 inch LCD screen
- Dual Camera
- Android 8.0
- call()
- text()
- photo()
- video()
Implementation view
class Smartphone{
private:
int ramSize;
string processorName;
float screenSize;
int cameraCount;
string androidVersion;
public:
void call();
void text();
void photo();
void video();
};

Example of ADT

Lets create student record

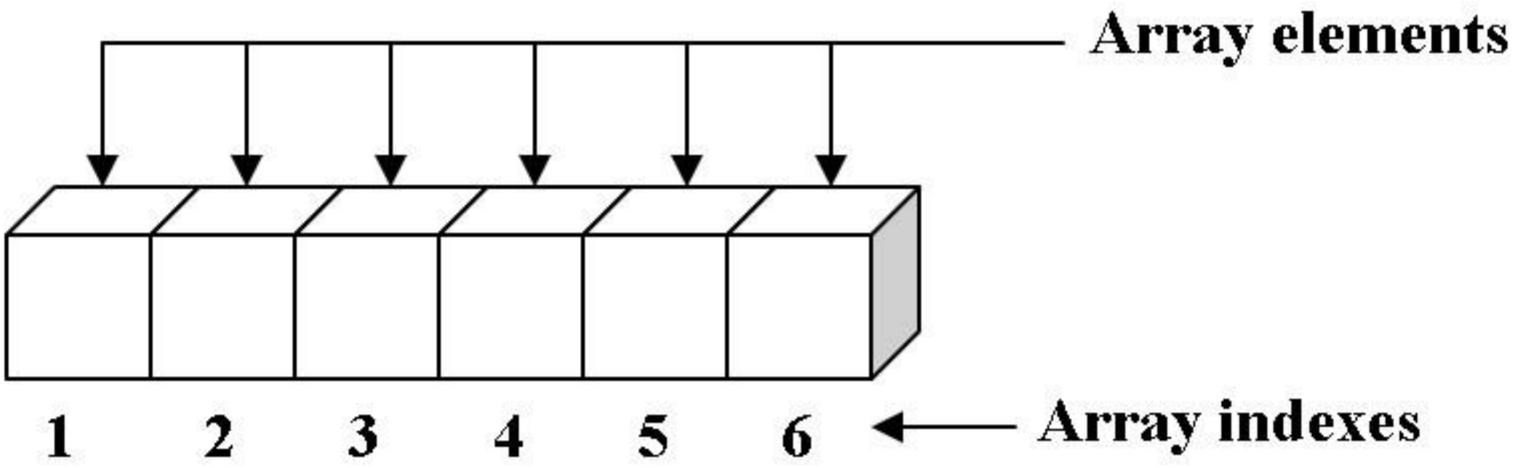
```
record{
    roll_no        number
    name           String
     a1, a2, a3    Decimal
     t1, t2, t3    Decimal
      total        Decimal
}
```

array[record] to store the records of multiple students

Functions for Student

- Find number of students in class
- Find total marks for each student
- Find class average
- Find highest marks

Array



One-dimensional array with six elements

- We can create array of any data type like int, char, struct, array etc

Array

Operations we can define on an array

- Creating the array
- Traversing elements
- Getting and setting an element at a particular position
- Inserting and deleting an element in an array
- Searching an element in an array
- Sorting the elements

Representation of an Single Dimensional Array in memory

- Consider an array of 6 elements

Index	0	1	2	3	4	5
Element	20	30	40	50	60	70
Memory Location	1000	1004	1008	1012	1016	1020

- How do we calculate memory location given the index value??
- Array stores Base Address:- 1000
- Calculate address of Arr[4]:-
  - **Address of A [I] = B + W * (I – LB)** **LB=0 in C**
  - ➔ 1000+ 4 * (4 – 0) = 1016

Representation of an Two Dimensional Array in memory

ROW MAJOR ORDER

Consider the array of size 3x3

Index	[0][0]	[0][1]	[0][2]
Element	10	20	30
Memory Location	1000	1004	1008
	[1][0]	[1][1]	[1][2]
	40	50	60
	1012	1016	1020
	[2][0]	[2][1]	[2][2]
	70	80	90
	1024	1028	1032

10	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**
1000	1004	1008	1012	1016	1020	1024	1028	1032

Address Calculation in Row Major

Address of A [I][J] = B + W * [N * (I – Lr) + (J – Lc)]

B = Base address

I = Row subscript

J = Column subscript

W = Storage Size of one element stored in the array (in byte)

Lr = Lower limit of row

Lc = Lower limit of column

M = Number of row of the given matrix

N = Number of column of the given matrix

Find address of int arr[1][2]

arr[1][2] = 1000 + 4 * [3 * (1 – 0) + (2-0)]

= 1020

Representation of an Two Dimensional Array in memory

COLUMN MAJOR ORDER

Consider the array of size 3x3

Index	[0][0]	[0][1]	[0][2]
Element	10	20	30
Memory Location	1000	1012	1024
	[1][0]	[1][1]	[1][2]
	40	50	60
	1004	1016	1028
	[2][0]	[2][1]	[2][2]
	70	80	90
	1008	1020	1032

10	**40**	**70**	**20**	**50**	**80**	**30**	**60**	**90**
1000	1004	1008	1012	1016	1020	1024	1028	1032

Address Calculation in Column Major

Address of A [I][J] = B + W * [(I – Lr) + M*(J – Lc)]

B = Base address

I = Row subscript

J = Column subscript

W = Storage Size of one element stored in the array (in byte)

Lr = Lower limit of row

Lc = Lower limit of column

M = Number of row of the given matrix

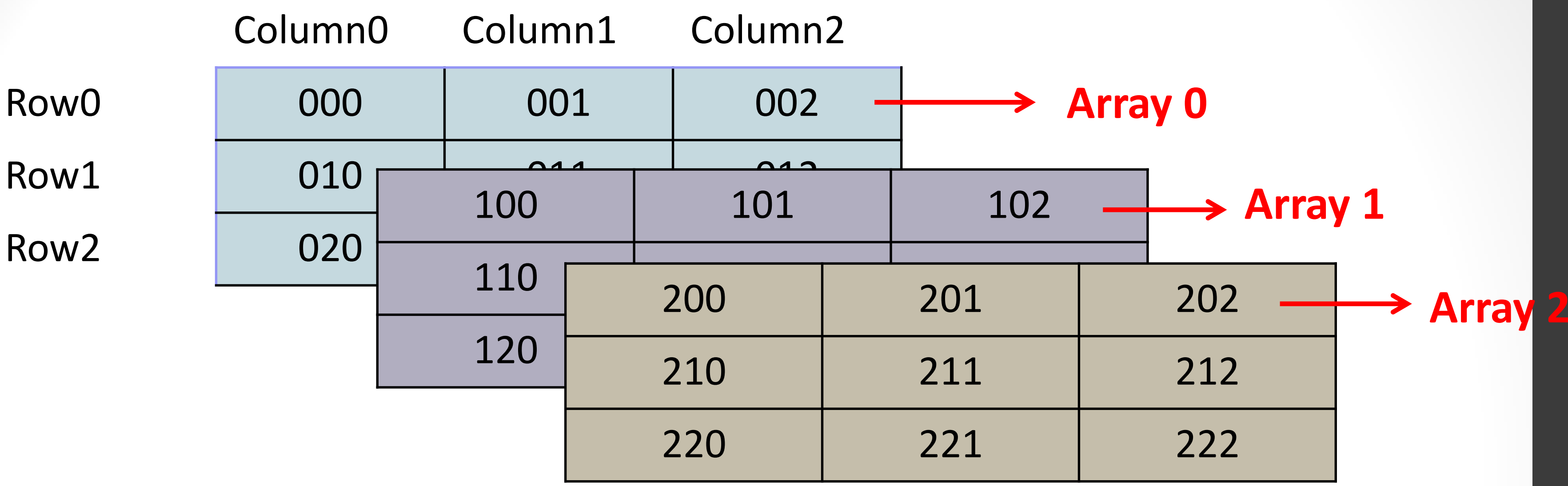
N = Number of column of the given matrix

Find address of int arr[1][2]

arr[1][2] = 1000 + 4 * [(1 – 0) + 3 * (2-0)]

= 1028

Address Calculation for 3 – Dimensional Array

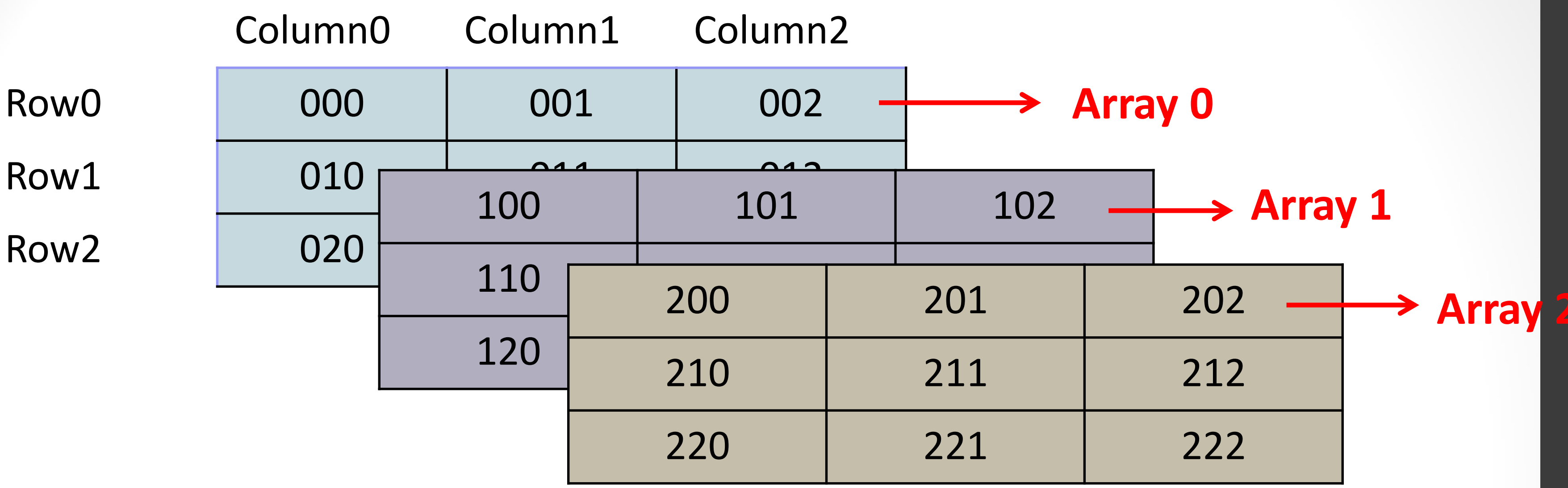


- **To calculate address of element arr[i,j,k] using row-major order :**

  A[I][J][K]= B + W(MN (K-L) + (I-L) + N*(J-L))*

arr[2][1][2] =1000 + 4* (3*3 *(2-0) + (2-0) + 3*(1-0))
= 1000 + 4 * (18 + 2 + 3)
= 1000 + 4 * (23)

Address Calculation for 3 – Dimensional Array



- **To calculate address of element arr[i,j,k] using row-major order :**

  A[I][J][K]= B + W(MN (K-L) + M*(I-L) + (J-L))*

arr[2][1][2] =1000 + 4* (3*3 (2-0) + 3*(2-0) + (1-0))

= 1000 + 4 * (18 + 6 +1)

= 1000 + 4 * (25)

Array — Abstract Data Type

Operations we can define on an array

- **Creating the array**

```
int arr[5];
```

- **Traversing elements**

```
for (int i=0;i<5;i++)
        printf("%d", arr[i])
```

- **Getting and setting an element at a particular position**

```
arr[2]=20;
printf("%d", a[2]);
```

Array — Abstract Data Type

Inserting an element in an array

Consider the array arr [5]

10	20	30	40	

- Insert element 50 at index 2

10	20		30	40

Create a blank space for 50

10	20	50	30	40

- Insert the element 70
  - This should give error as no space is left

```
void insert(int arr[], int size, int element, int index)
{
        if(size==max){
                printf("Array is Full");
                return;}
        for(int i=max-1; i>=index;i--)
                arr[i+1]=arr[i];
         arr[index]=element;
        size+=1;
        for(int i=0; i<max;i++)
                printf("%d ", arr[i]);
}
```

Array — Abstract Data Type

Delete an element 30

10	20	30	40	50

10	20	30	40	50

10	20	40	50	

```
int delete(int arr[], int size, int index)
{
        if(size<0 || size>index)
        {       printf("index element is not in array");
                return 0;
        }
        for(int i=index; i>max;i++)
                arr[i]=arr[i+1];
        for(int i=0; i<max;i++)
                printf("%d ", arr[i]);
        return 1;
}
```

ARRAY - ADT

Arrays:-

Disadvantage:- Static Memory Allocation

Size needs to be specified at Compile Time

int a[100]

Partial Solution to the Problem:- **Create Dynamic Allocated Arrays**

```
int *arr = (int*) malloc(5 * sizeof(int));
```

Advantage:- Size can be specified at Runtime

Disadvantage :- Resizing of Array is not possible

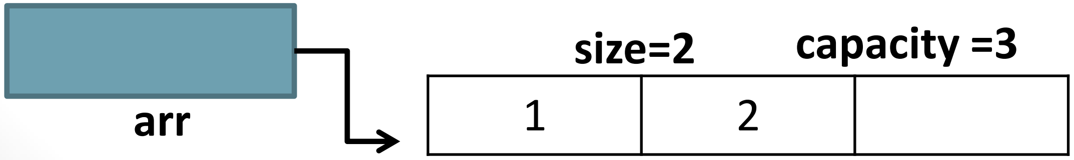
ARRAY - ADT

Complete Solution to the Problem:-

CREATE DYNAMIC ARRAY

Dynamic Array (Resizable Array)

➔ Store a pointer to dynamically created array and replace with newly created Array.



Pointers

Declaration : int* p;

Initialization:

```
int *pc, c;
c = 5;
pc = &c;
```

Display

```
printf("%d", *pc);    // print 5
printf("%p", pc);    // prints address of c
```

```
c = 1;
OR
*pc = 1;
```

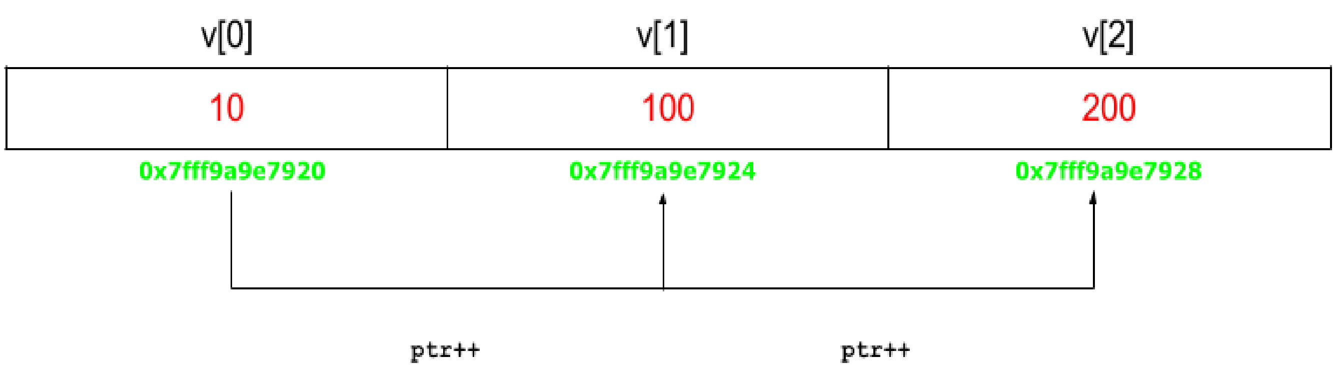
```
printf("%d", c); // OUTPUT 1
printf("%d", *pc);// OUTPUT 1
```

Pointer Arithmetic

```
int *ptr;
ptr++;
```

```
int v[3] = {10, 100, 200};
ptr=v

for (int i = 0; i < 3; i++)
  {
    printf("Value of *ptr = %d\n", *ptr);
    printf("Value of ptr = %p\n\n", ptr);
    ptr++;
  }
```



Structures and Pointers

Syntax of struct

```
struct structureName
{ dataType member1;
dataType member2; ... };

struct Person {
        char name[50];
        int citNo;
        float salary; }
person1, *personptr, p[20];
int main()
{ struct Person person1, person2, p[20];
return 0; }
```

Accessing members

Member Operator : - .

Structure pointer operator :- ->

person1.salary = 2000;

personptr->salary = 3000;

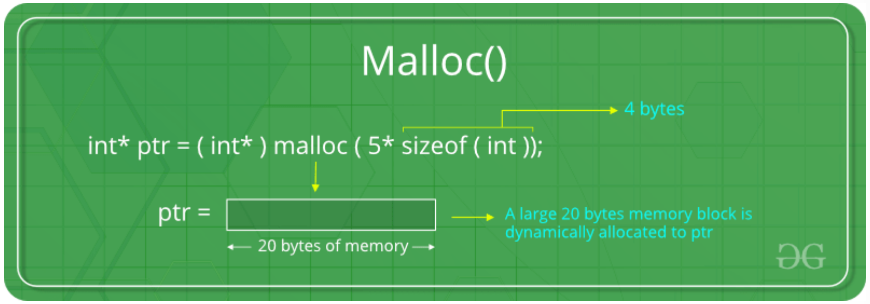
(*personptr).salary=300

Dynamic Memory Allocation

4 library functions provided by C in **<stdlib.h>**

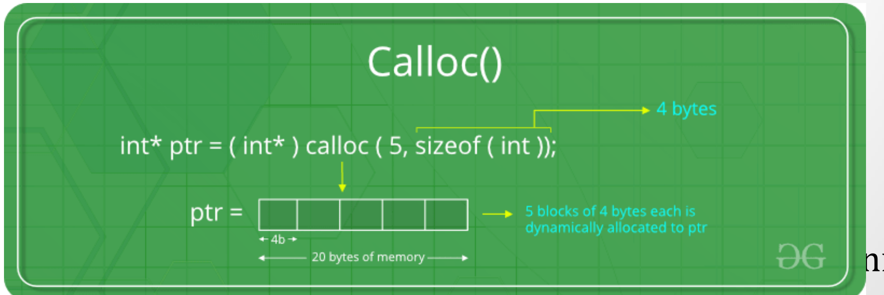
- malloc()   : ptr = (cast-type*) malloc(byte-size)

  ptr = (int*) malloc(5 * sizeof(int));



- calloc() : ptr = (cast-type*)calloc(n, element-size);

ptr = (float*) calloc(5, sizeof(float));



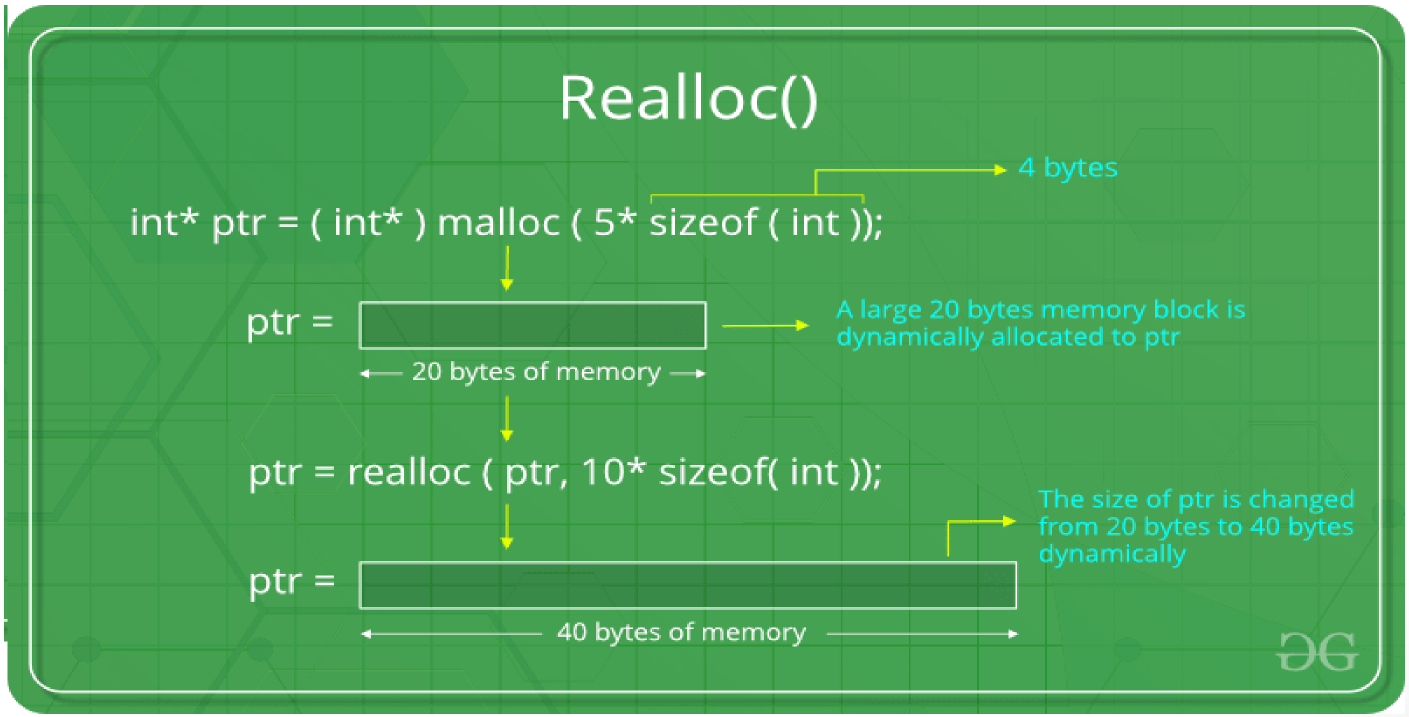
ni

Dynamic Memory Allocation

free: - **free(ptr);**

realloc() method

ptr = realloc(ptr, newSize);



Declaration of struct Array

```
struct Array
   {
      int *a;
      int size;
      int max;
   };
```

Initializing the array

```
void create(){
 SArray = (struct Array *)malloc(sizeof(struct Array));
printf("Enter max elements that can be stored in an Array");
   scanf("%d",&SArray->max);
   SArray->a=(int *)malloc(SArray->max*sizeof(int));
   SArray->size=0;
SArray->a[0]=1;
SArray->a[1]=2;
SArray->a[2]=3; }
```

Displaying all elements of an array

```
void display(struct Array *SArray)
{
   printf("\nThe elements are : ");
   for(int i=0;i<SArray->size;i++)
      printf("%d ",SArray->a[i]);

}
```

Get an element at a given index

```
int Get(struct  Array *SArray, int index){
          return SArray->a[index];
 }
```

Set an element at a given index

```
void Set(struct  Array *SArray, int index, int key){
          SArray->a[index] = key;
}
```

Resizing the Array

```
void resize(struct Array *a)
{
    int *temp=(int *)malloc (2*a->size * sizeof(int));
    for(int i=0;i<a->length;i++)
        temp[i]=a->A[i];
    a->A=temp;
    a->size *= 2;
}
```

OR

```
void resize(struct Array *a)
{
    a->A=(int *)realloc(a->A, 2*a->max*sizeof(int));
        a->max *=2;
}
```

Lab Assignment

E-1: To study an Array ADT and to implement various operations on an Array ADT.

Create an array and implement the operations – traverse(), insert_element(), delete_element(), sort(), search(), copy(), create().

Write a C program to demonstrate an array ADT using defined operations appropriately using a menu-driven approach. Your program should be able to print the array contents appropriately at any or all instances (as required may be).

Note:- You must also ensure that no input is acquired within the body of functions, nor (preferably) display any prompts/results.