Republic of Iraq
The Ministry of Higher Education &
Scientific Research

University: Anbar
College: CS & IT
Department: Computer Science
Stage: 2
Instructor name: Mohammed Salah Ibrahim

Course Weekly Outline

Course Name: Data Structures

<table>
<tr><td>Course Instructor</td><td colspan="5">Dr. Mohammed Salah Ibrahim</td></tr>
<tr><td>E-mail</td><td colspan="5">moh.salah@uoanbar.edu.iq</td></tr>
<tr><td>Title</td><td colspan="5">Teacher</td></tr>
<tr><td>Course Coordinator</td><td colspan="5">Dr. Mohammed Salah Ibrahim</td></tr>
<tr><td>Course Objective</td><td colspan="5">1- Learning different data structures
2- Understand why this data structure is better than the other one.
3- Learning how to choose the best data structure for your algorithm.
4- learn how to deal with your problem, building its algorithm and fitting the best data structures to it.</td></tr>
<tr><td>Course Description</td><td colspan="5">This course covers all data structure types. It starts with defining algorithms and their complexity from the time and space prospection. Then, a list of data structure and their description is presented. The course describes every data structure in detail. In addition to that, it gives the reason to why we need this data structure and where to use it. This course includes many projects that give more understanding to the data structure studied. These projects talks about real life problems that we ask student to use one of the data structure that has been presented in the course to solve it.</td></tr>
<tr><td>Textbook</td><td colspan="5">Introduction to Algorithm, third Edition, Thomas H. Cormen
Algorithms, fourth edition, Robert Sedgewick and Kevin Wayne</td></tr>
<tr><td>References</td><td colspan="5">Introduction to Algorithm, third Edition, Thomas H. Cormen
Algorithms, fourth edition, Robert Sedgewick and Kevin Wayne</td></tr>
<tr><td rowspan="2">Course Assessments</td><td>Term Tests</td><td>Laboratory</td><td>Quizzes</td><td>Project</td><td>Final Exam</td></tr>
<tr><td>%20</td><td>%10</td><td>%5</td><td>%15</td><td>%50</td></tr>
<tr><td>General Notes</td><td colspan="5"></td></tr>
</table>

Republic of Iraq
The Ministry of Higher Education &
Scientific Research

University: Anbar
College: CS & IT
Department:
Stage:
Instructor name:

Course Weekly Outline

Week	Date	Topics Covered	Lab. Experiment Assignments	Notes
1		**Introduction to Data Structures**		
2		**Algorithms and Complexity**		
3		**Arrays and Pointers**	Accountant application using arrays	
4		**Linked List 1**		
5		**Linked List 2**	Student information system using linked list	
6		**First exam**		
7		**Stack**	Color cubes games using Stack	
8		**Queue**	A snake game using queu	
9		**Tree 1**		

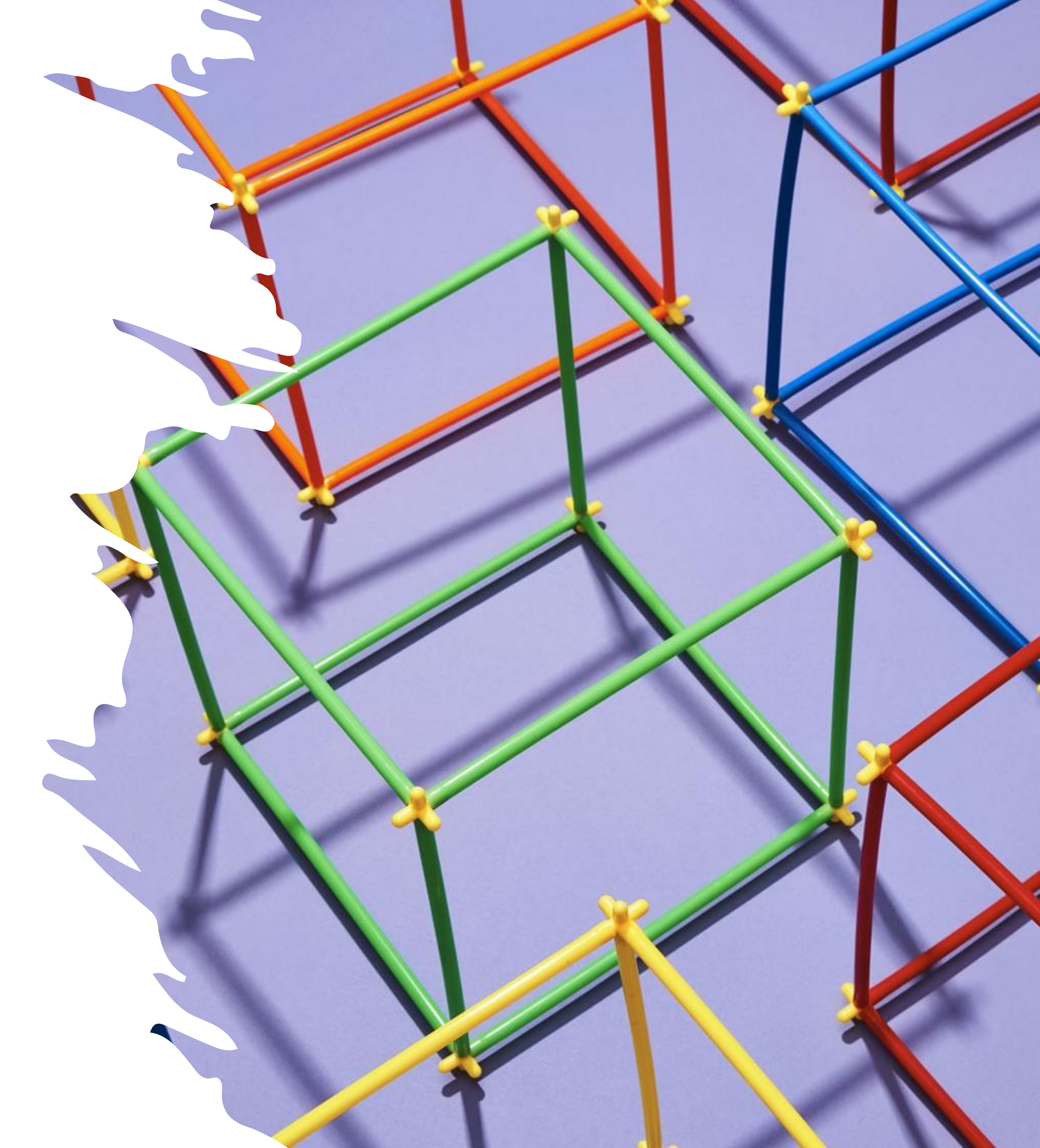
10		**Tree 2**		
11		**Graph 1**		
12		**Graph 2**	Social Media connections using Graph data structure	
13		**Hashing 1**		
14		**Hashing 2**	Simple search engine application using hashtable data structure	
15		**Second try exam**		

Instructor Signature: **Dean Signature:**

Data Structure
Lecture 1: Introduction

Prepared by

Dr. Mohammed Salah Al-Obiadi



Variables and Data Types

- **Variable** is any entity that can take on different values.
- Consider the below equation:

$$x^2 + 2y - 2 = 1$$

- This equation has variables *x* and *y*, which hold values (data).
- **Data Type** is a set of data with predefined values.
- The variables *x* and y in the above equation can take any values such as
  - Integer numbers (10, 20), Real numbers (0.23, 5.5), or Boolean (0 or 1).
- There are two types of data types:
  - System-defined data types.
  - User-defined data types.

System-defined data types:

- These are the data types that are defined by system are called primitive data types.
  - Examples of such data types are int, float, char, double, bool, etc.
- Each data type has some bytes to store data.
  - For example:
    - int may take 2 bytes or 4 bytes of memory.
    - float may take 3 bytes or 4 bytes of memory.
    - char may take 1 byte of memory.
    - Symbols in ASCII codes like +, -, *, /, @,#, etc… take 1 byte of memory. (See Appendix A for the list of ASCII codes)
  - If we have x+y, and x is int and y is float. Assume int takes 2 bytes, and float takes 3 bytes, then the equation x+y take 2+1+3=6 bytes of memory.
    - Note that the symbol '+' has 1 byte in the memory.

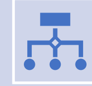
User defined data types

- If the system-defined data types are not enough, then most programming languages allow the users to define their own data types.
- Good examples of user defined data types are: structures in C/C + + and classes in Java.
- Here is an example of new defined data type called “newType”:

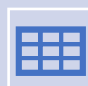
```
struct newType {
    int data1;
    float data 2;
    ...
    char data;
};
```

Data Structures

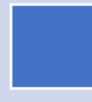
Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.



General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.



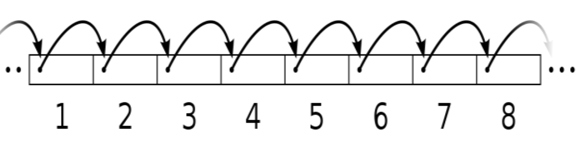
Data structures are classified into two types:



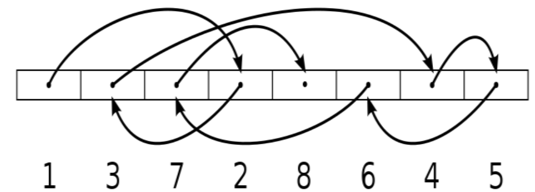
1- Linear data structures: elements can be accessed in a sequential order (e.g. Arrays, Linked Lists, Stacks and Queues).

2- Non – linear data structures: elements can be accessed in a random order (e.g. Trees, tables, sets, graphs).

Sequential access



Random access



Operations Performed in Data Structure

1- Traversing

2- Insertion

3- Deletion

4- Merging

5- Sorting

6- Searching

Appendix A:

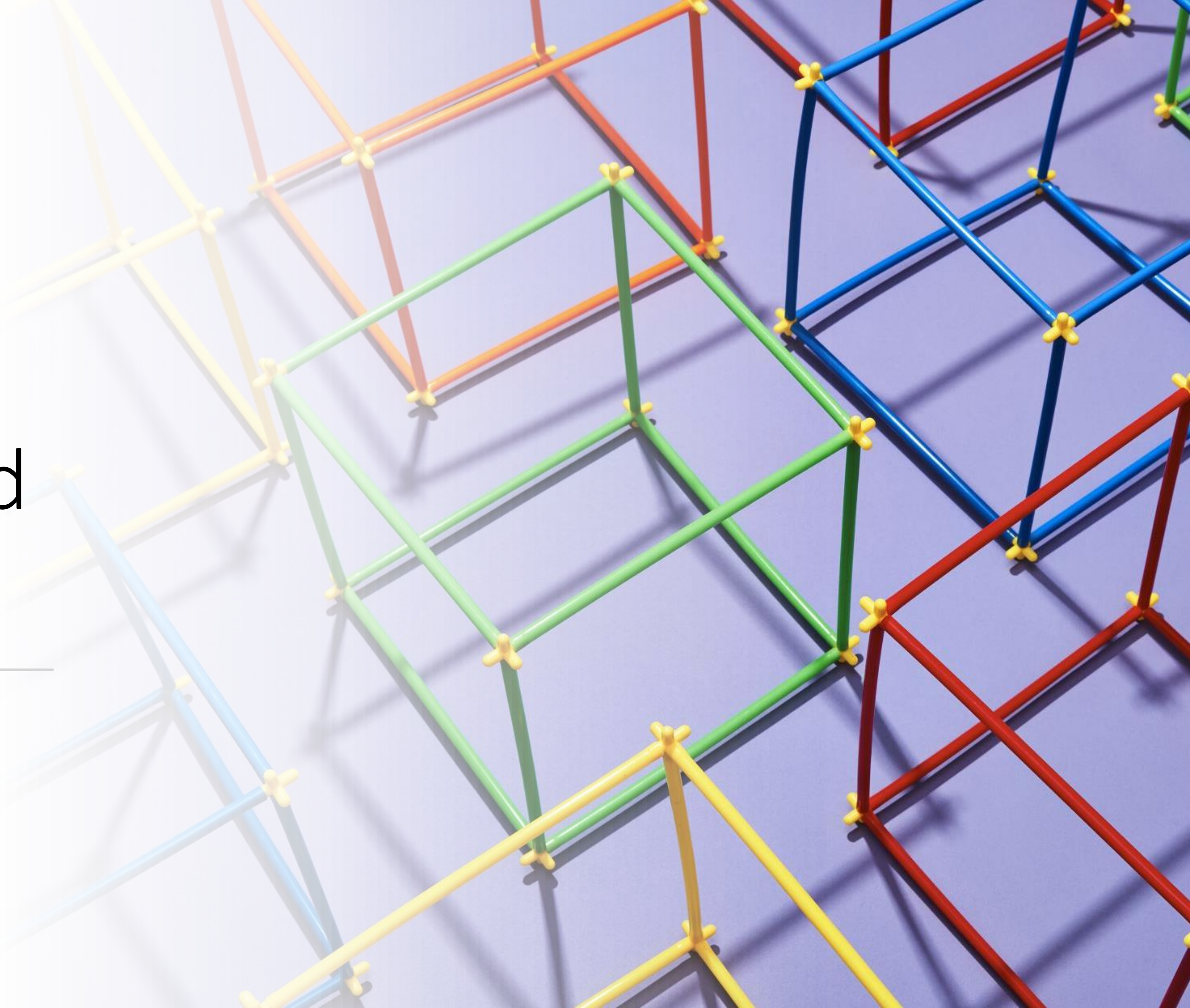
ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Data Structure Lecture 2: Algorithms and Complexity

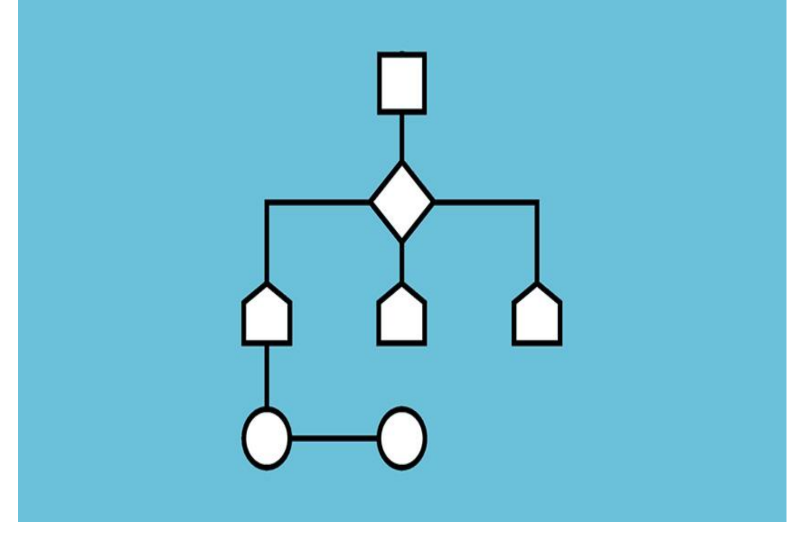
Prepared by

Dr. Mohammed Salah Al-Obiadi



What is an Algorithm?

- Steps of preparing a frying egg.
  1. Get the frying pan.
  2. Get the oil.
     a. Do we have oil?
        i. If yes, put it in the pan.
        ii. If no, do we want to buy oil?
           a. If yes, then go out and buy.
           b. If no, no egg today.
  3. Turn on the stove, etc...



An algorithm is the step-by-step clear instructions to solve a given problem.

Criteria for judging Algorithms

- There are two main criteria for judging Algorithms:
  1. Correctness: does the algorithm give solution to the problem in a finite number of steps?
  2. Efficiency: how much resources (in terms of memory and time) does it take to execute the program.

Complexity of an Algorithm

1. **Space Complexity of a program** is the amount of memory it needs to run to completion.
2. **Time complexity of a program** is the amount of computer time it needs to run to completion. The time complexity is of two types such as
   a) Compilation time
   b) Runtime

Asymptotic Notations: Big-O Notation

- **Big-O Notation [Upper Bounding Function]:** The O(g(n)) represents the upper bound computation a program can cause to the computer. f(n) = O(g(n)) (read as f of n is big oh of g of n)
- **Example-1** Find upper bound for:
  - $f(n) = 3n + 8$
    - solution $f(n) = O(n)$
  - $f(n) = n^2 + 1$
    - Solution $f(n) = O(n^2)$
  - $f(n) = 16n^3 + 45n^2 + 12n$
    - Solution $f(n) = O(max(n^3, n^2, n)) = O(n^3)$
  - $f(n) = n^4 + 100n^2 + 50$
    - Solution $f(n) = O(max(n^4, n^2)) = O(n^4)$
  - $f(n) = 410$
    - *Solution* $f(n) = O(1)$

Example a program of O(1):

- Problem: To find out the greater between two numbers

```
bool max_value (int a, int b) // function that accept two
numbers
{
   if (a> b) // Compare the two numbers
         return true; // if first is greater return true
   else
         return false. // otherwise return false
}
```

- This function does not have any loop and will not cost the computer a lot of computations, so it’s *f(n)=O(1)* means a constant computations.

Example a program of O(n):

- Problem: Program to search a number from a list of numbers

```
bool search (int arr [], int number, int n)
{
bool found=false;
for (int i=0; i<n; i++)
{
      if (arr[i] ==number)
         {
         found=true;
         break;
         } // end of if
} // end of for
return found;
} // end of function
```

- This function has a for loop that require *n* time implementations from the computer, so it's $f(n)=O(n)$.

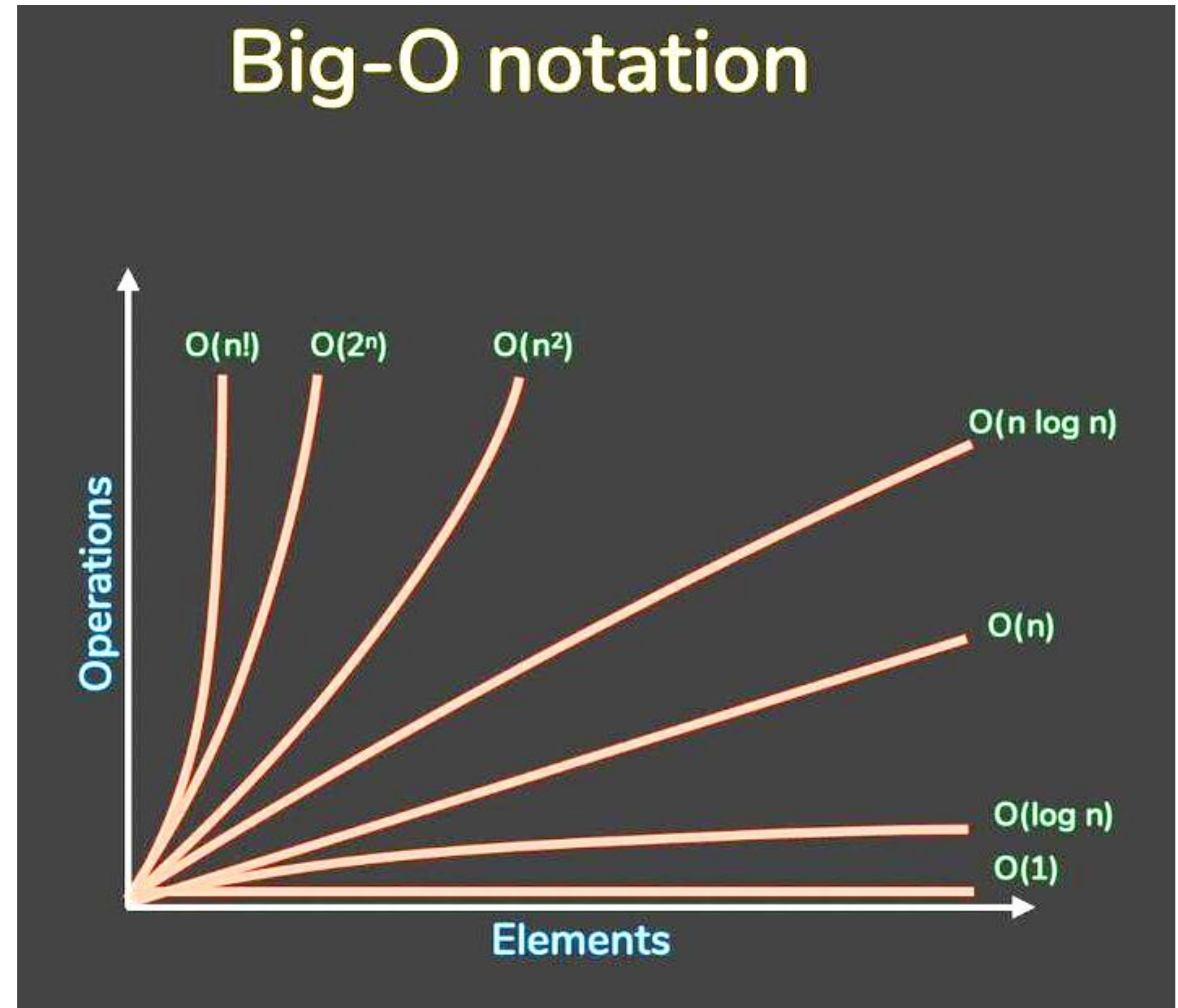
Example a program of $O(n^2)$:

Problem: Write a program to sort the series of numbers using Bubble sort

```
void array (int arr [], int n)
{
int i, j;
for (i=0; i<n; i++) // start of outer loop
{
    for (j=1; j<n-i; j++) // inner loop
    {
        if (arr [j+1] > arr[j]) // comparing the elements
            { // swapping if the adjacent is larger
            temp=arr [j+1];
            arr [j+1] =arr[j];
            arr[j] =temp;
            } // end of if
    } //end of inner for loop
} // end of outer for loop
```

- This function has two *for loops* that require n x n time implementations from the computer, so it's $f(n)=O(n \times n)=O(n^2)$.

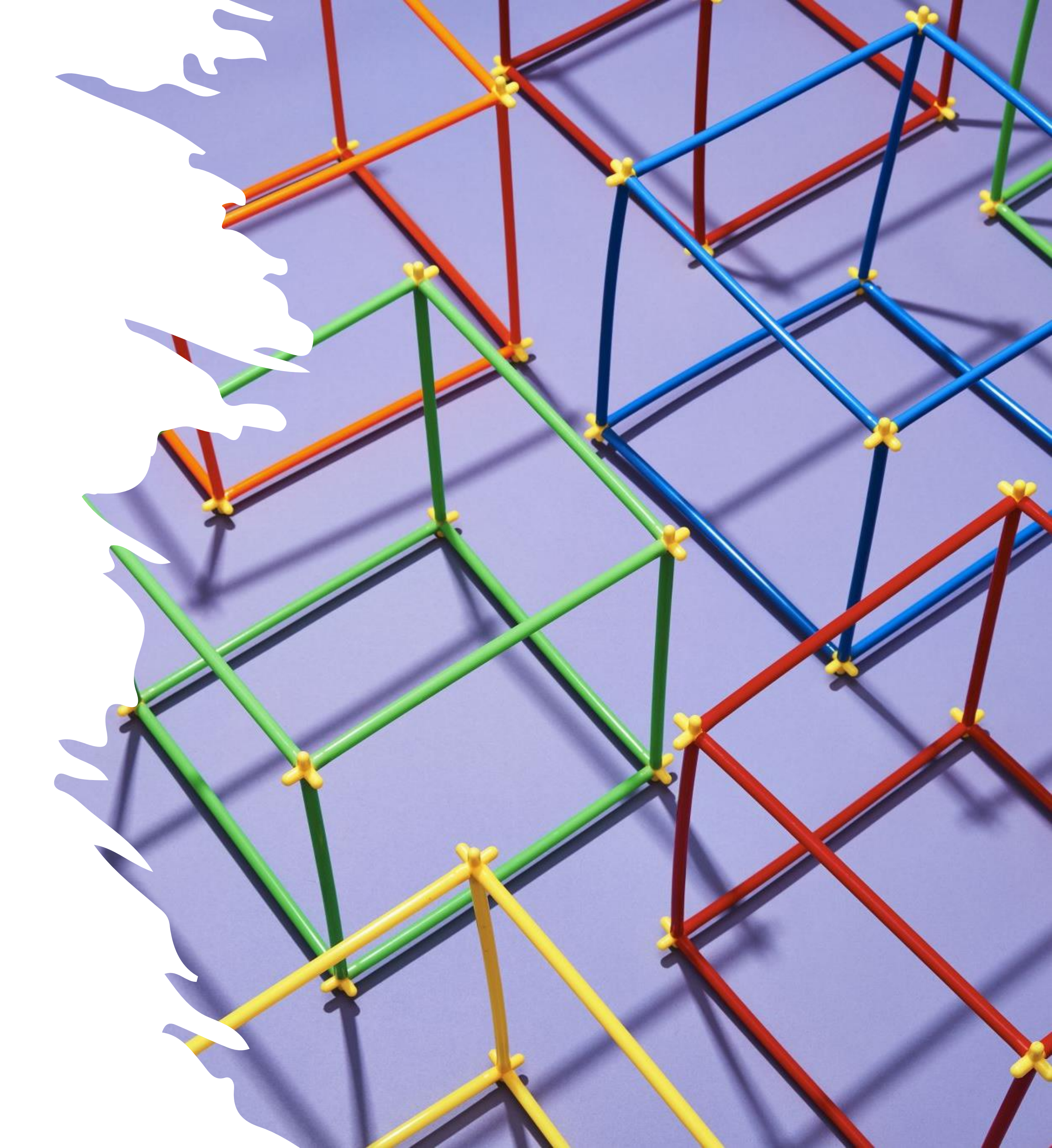
Time Compexity Chart



Data Structure Lecture 3: Arrays and Pointers

Prepared by

Dr. Mohammed Salah Al-Obiadi



Arrays data structures

- Arrays are widely used in any programming language.
- It is extremely useful in cases where we need to store the similar set of elements.
- It helps in reducing the program complexity.
- increases the programmer's productivity.
- Arrays can be categorized into the following:
    - Single Dimensional array.
    - Double Dimensional array.
    - Multidimensional array.
- We will not study the Multidimensional array.

Why we use Arrays

- Consider that we need to store grades of five students.
- In a normal way, we have to define five variables of the same type:

```
int main ()
  {
      int marks1, marks2, marks3, marks4, marks5;
      cout<<"enter marks1";
      cin>>marks1;
      cout<<"enter marks2";
      cin>>marks2;
      cout<<"enter marks3";
      cin>>marks3;
      cout<<"enter marks4";
      cin>>marks4;
      cout<<"enter marks5";
      cin>>marks5;
      return 0;
  }
```


- Complexity of the above program will grow further upon increment of subjects.
- Consider we have 200 students, how the program will look like? What is the solution?
- Here the solutions lie with the usage of arrays.

Array can be defined as:

- A data structure used to store set of similar data types.
- Elements are stored in continuous memory locations.
- Index, or subscript starts with 0.
- Size of the array should be constant.

One-Dimensional Array

Declaration:

- *Data type variable_name[bound] ;*

Examples:

- *Int arr[10]; // an integer array with 10 elements.*
- *Char arr[20]; // a character array with 20 elements.*
- *float arr[15]; // a flaot array with 20 elements*

Array Element in Memory

The array elements are stored in a consecutive manner inside the memory.

For Example: int x[7];

Let the x[0] be at the memory address 568, then the entire array can be represented in the memory as:

x[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]
568	570	572	574	576	578	580

Two-Dimensional Array

Declaration:

- *Data type variable_name[rows] [columns] ;*

Examples:

- *Int arr[4][6]; // an integer 2-D array with 4 rows and 6 columns.*
- *Char arr[20][20]; // a character 2-D array with 20 rows and 20 columns.*
- *float arr[5][10]; // a float 2-D array with 5 rows and 10 columns*

Examples of Two-dimensional arrays

```
int x[3][4]={
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {2, 4, 6, 3},
         };
char x[3][4]={
            {'h', 'a', 'f', '7'},
            {'u', 'f', 'z', 'l'},
            {'y', '8', 'j', 'm'},
         };
```

POINTERS

Pointer is a variable that is capable to hold the address of another variable.

Holding of addresses of another variable is needed in various instances that include:

1- To access the array element

2- To change the value of variable from function

3- In dynamic allocation of memory.

4- In complex programming, such as link list, tree, B tree etc.

How to know a variable is a pointer?

Pointers are preceded with the symbol *.

For instance:

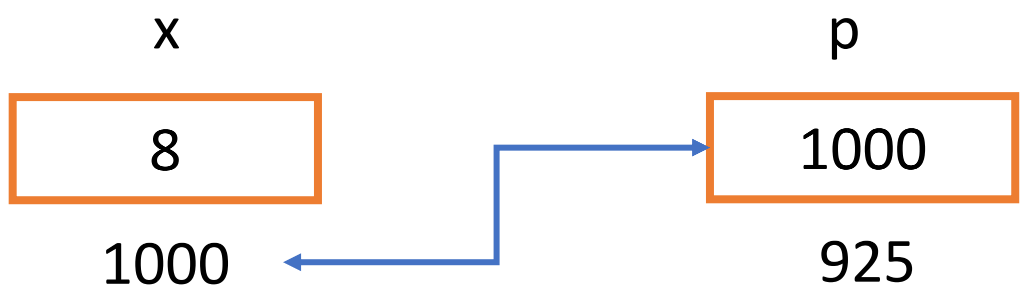
- int *x, It means that this pointer can hold the address of integer type variable.
- char *c, , It means that this pointer can hold the address of char type variable.
- float *w, It means that this pointer can hold the address of float type variable.

Example of Declaring pointer

```
int x=8;
int *p;// variable that is pointer of int type
p=&x; //p now holds the address of variable x
cout<<p; // print the address of x;
cout<<*p; // print the value pointed by p;
```

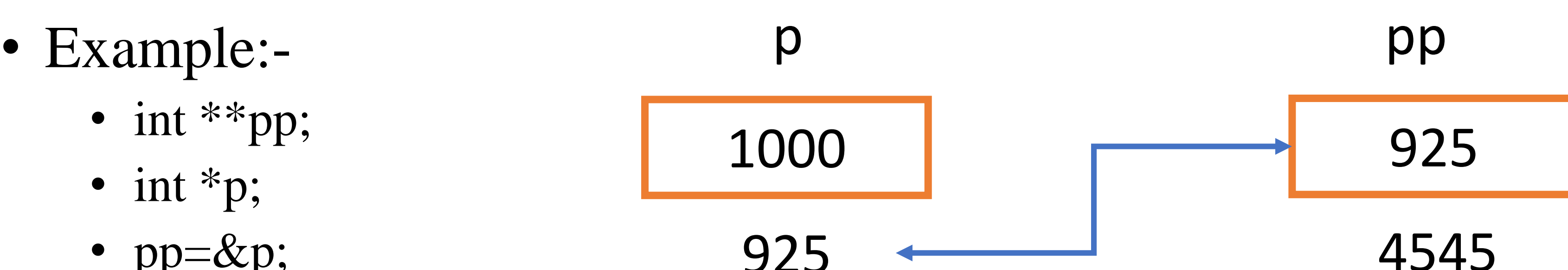
Explaining Example of Declaring pointer

- Initially, the variable "x" is declared
- Assumes that it has been allocated the address location 1000.
- when int *p is declared, it is also allocated the address 925.
- When p=&x, this means that p holds the address of variable x which is 1000.
- Printing p will print address while printing *p will print x value.



Pointer to pointer

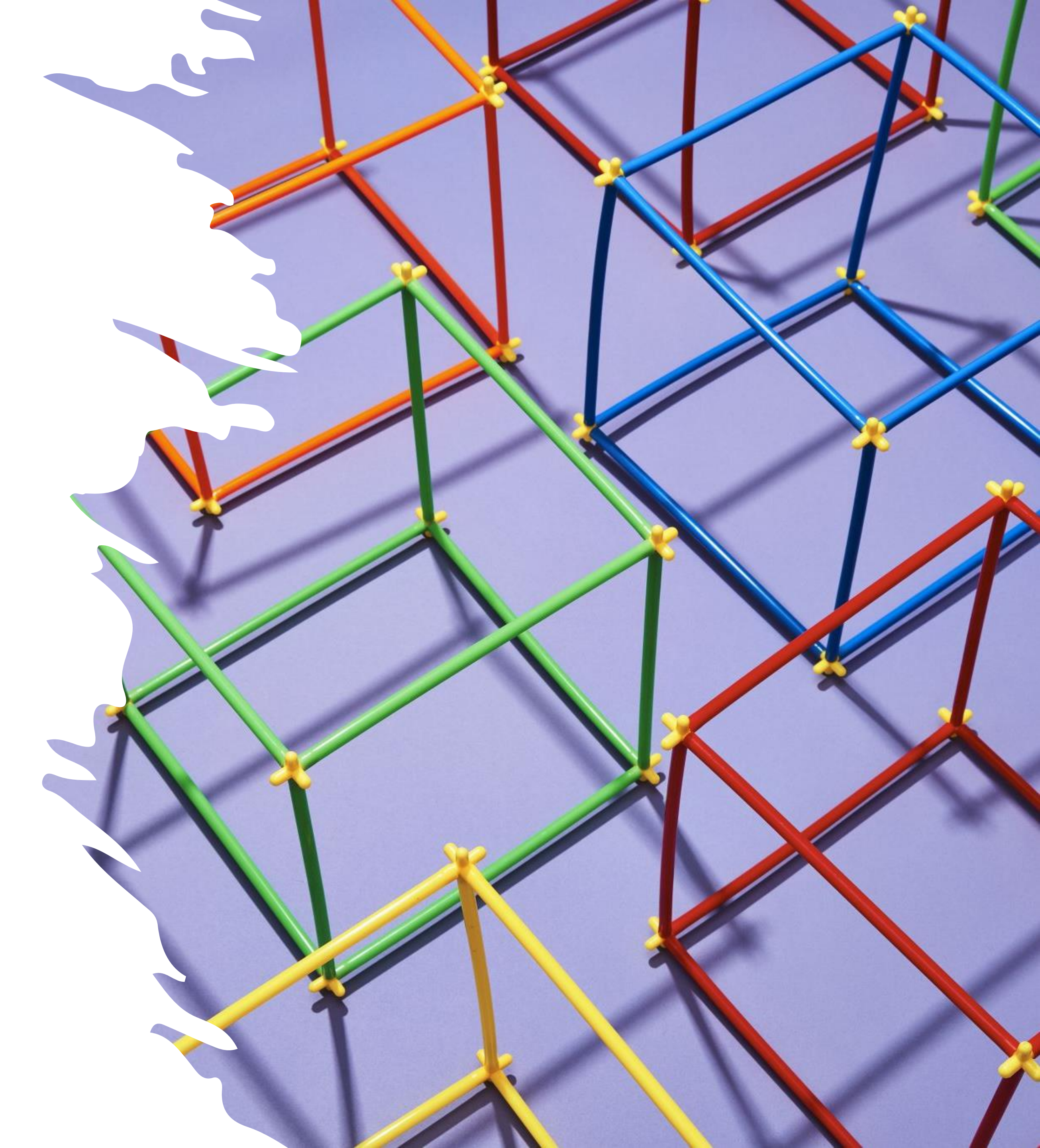
- Sometimes, we need to store the address of a pointer.
- This can be accomplished with the help of pointer to pointer.
- Pointer to pointer is a variable that holds the address of another variable that is pointer type.
- Declaring pointer to pointer is different from the normal pointer type.
- In pointer to pointer notation two asterisk (**) are preceded before the identifier.
- Example:-
  - int **pp;
  - int *p;
  - pp=&p;



Data Structure
Lecture 4: Linked List

Prepared by

Dr. Mohammed Salah Al-Obiadi

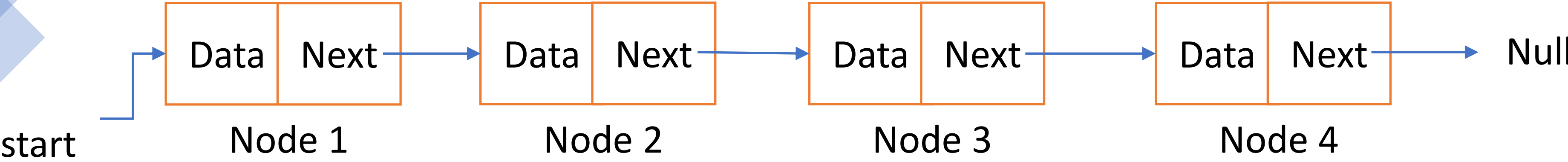


What is a Linked List?

A linked list is a data structure used for storing collections of data.

A linked list has the following properties:

1. Successive elements are connected by pointers.
2. The last element points to NULL.
3. Can grow or shrink in size during execution of a program.
4. Can be made just as long as required (until systems memory exhausts).
5. Does not waste memory space. It allocates memory as list grows.



Linked List vs Arrays?

Array	Linked list
Array elements store in a contiguous memory location.	Linked list elements can be stored anywhere in the memory
Array works with a static memory and cannot be changed at the run time.	The Linked list works with dynamic memory means memory size can be changed at the run time.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list takes less time while performing any operation like insertion, deletion, etc.
Accessing any element in an array is faster as the element in an array can be directly accessed through the index.	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time.	In the case of a linked list, memory is allocated at run time.
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements then the rest of the space will be unused.	Memory utilization is efficient as the memory can be allocated or deallocated at the run time.
Arrays take O(1) for access to an element.	Linked lists take O(n) for access to an element.

Operation on Linked List

1- Traversal: To traverse all the nodes one after another.

2- Insertion: To add a node at the given position.

3- Deletion: To delete a node.

4- Searching: To search an element(s) by value.

5- Updating: To update a node.

6- Sorting: To arrange nodes in a linked list in a specific order.

7- Merging: To merge two linked lists into one.

Types of Link List

1- Single Link List

2- Double Link List

3- Circular Link List

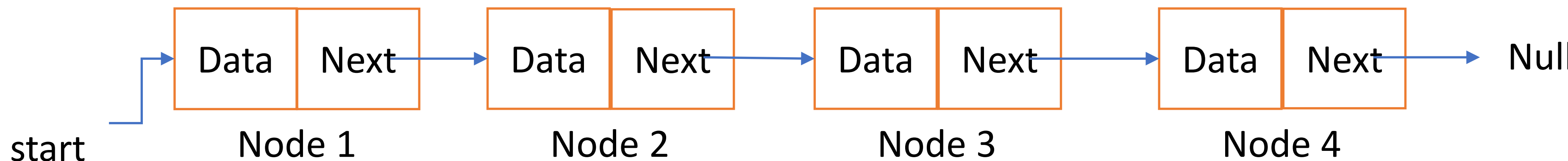
4- Doubly Circular linked list

Single Link List

Generally "linked list" means a single linked list.

This list consists of a number of nodes in which each node has a *next* pointer to the following element.

The link of the last node in the list is NULL, which indicates the end of the list.



STRUCTURE OF THE NODE OF A LINKED LIST

Struct **tagname**

```
{
    Data type member1;
    Data type member2;
    ……………………….
    ………………………
    …………………….. .
    Data type membern;
    Struct tagname *var;
};
```

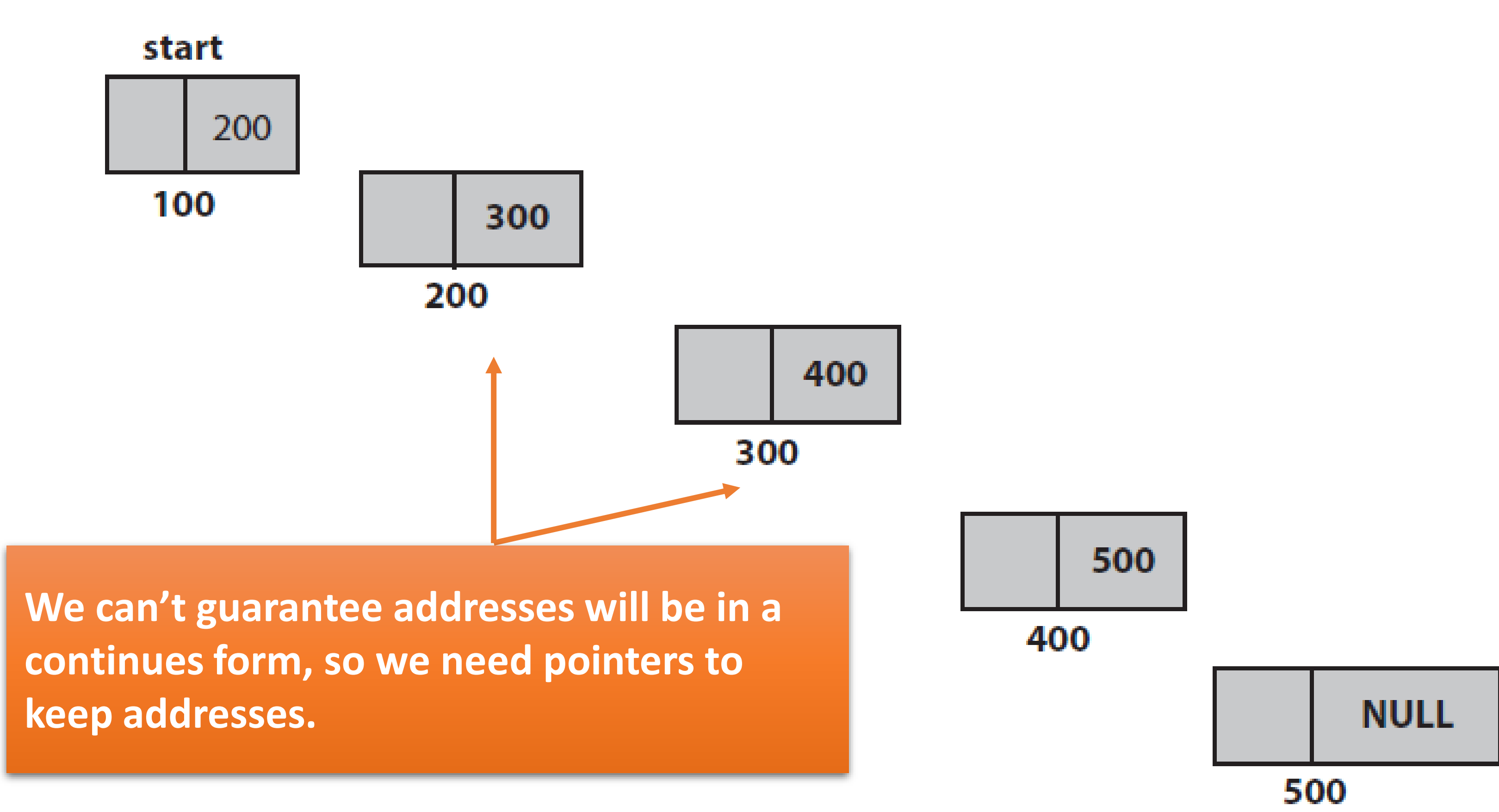
Example:

struct **link**

```
{
int info;
struct link *next;
};
```

LOGIC FOR CREATION

struct link start, *node;



Algorithm For Creation Of Single Link List

Struct link start, *node

create(start,node) [start is the structure type of variable][node is the structure type of pointer]

step-1 : node = &start

step-2 : node → next = new link() //allocate memory of size struct link for the node

node = node → next

input : node → info

node → next = null

step-3 : repeat step-2 to create more nodes

step-4 : return

Algorithm For Traversing Of Single Link List

struct link start, *node;

traverse(start,node) [start is the structure type of variable] [node is the structure type of pointer]

step-1 : node = start.next

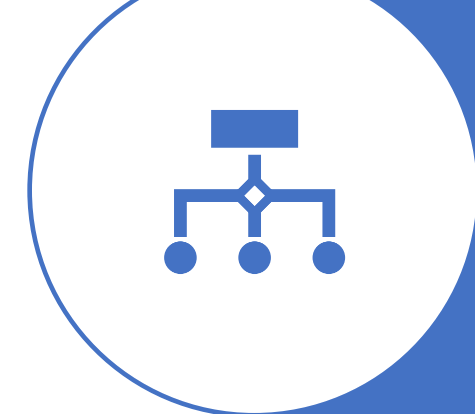
step-2 : repeat while (node!=null)

write : node → info

node = node → next

end of loop

step-3 : return



Insertion Into Linked List

The insertion process with link list can be discussed in four different ways:

1. Insertion at Beginning.
2. Insertion at End.
3. Insertion when node number is known.
4. Insertion when information is known.

Algorithm For Insertion At Beginning

struct start, *first, *node,* newnode

insbeg(start,first,node, newnode) [start is the structure variable] [node and first is the structure pointer]

step-1 : first = &start //first saves start's address

node = start.next

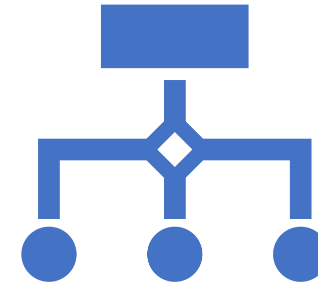
step-2 : newnode = new link()

input : newnode → info

first → next = newnode

newnode → next := node

step-3 : return



Algorithm For Insertion At Last

struct start, *last, *node,* newnode

inslast(start,last,node,newnode)

step-1 : last = &start //last's pointer saves start's address

node = start.next

step-2 : repeat while(node != null)

node = node → next

last = last → next

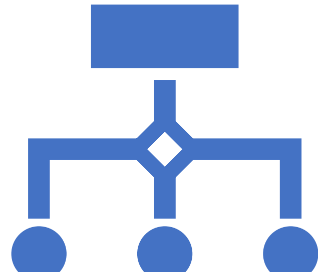
step-3 : newnode →next=new link() //allocate a memory to newnode

input : newnode → info

last → next = newnode

newnode → next = null

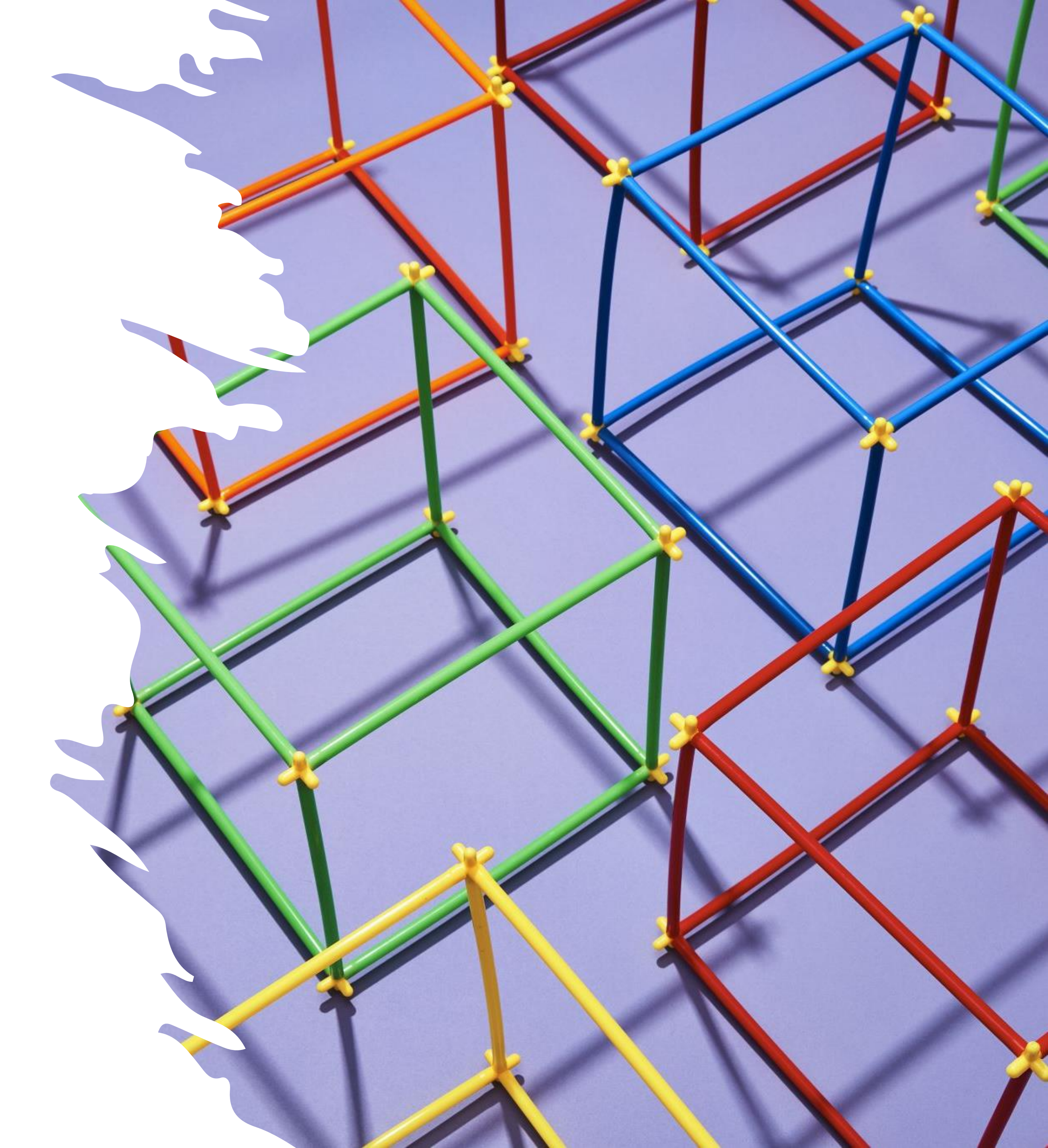
step-4 : return



Data Structure
Lecture 5: Linked List

Prepared by

Dr. Mohammed Salah Al-Obiadi

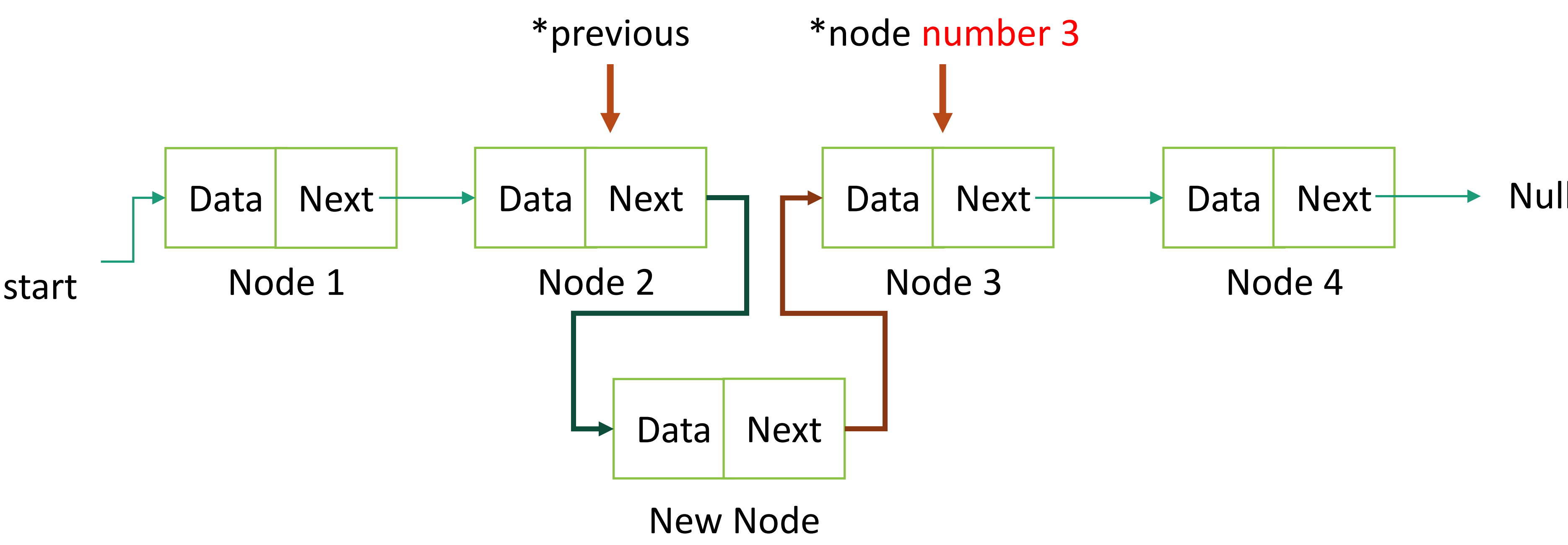


Algorithm For Insertion Of Node When Node Number Is Known

(insert in previous position)

```
struct start, *previous, *node, *newnode
insnode(start, previous,node,no,newnode) [no is the node number]
step-1 : previous := &start
         node := start.next
         count :=1
step-2 : repeat while(node != null)
            if(count = no) then:
               newnode → next = new link() \\allocate space in memory
               input : newnode → info
                previous → next = newnode
               newnode →next = node
               return
            else :
               node = node → next
                previous = previous → next
               count:=count+1
step-3 : return
```

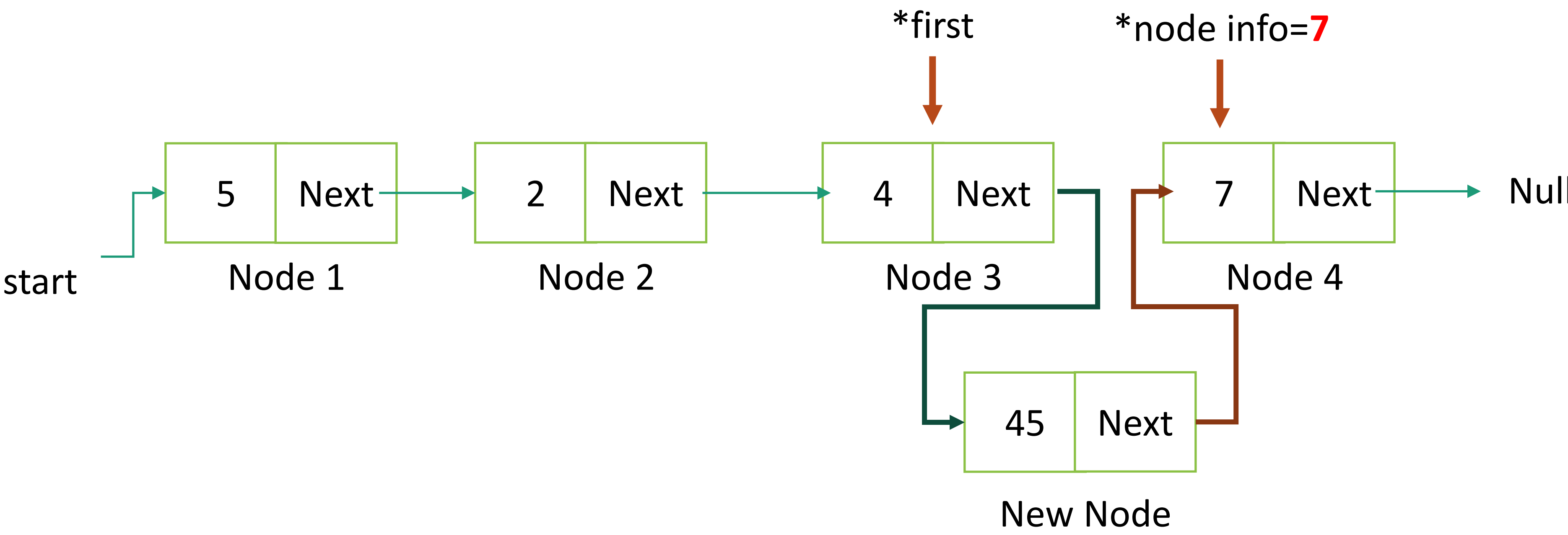
Example: insert a new node at node number 3



Algorithm For Insertion Of Node When Information Is Known

```
insertnode(start, previous,node,data,newnode) [data is information to insert]
step-1 : previous := &start
            node := start.next
step-2 : repeat while(node != null)
              if(node → info = data) then:
                   newnode →next= new link()
                   input : newnode → info /*insert data*/
                   previous → next = newnode
                   newnode →next = node
                   return
              else:
                   node = node → next
                    previous = previous → next
step-3 : return
```

Example: insert a new node at info =7

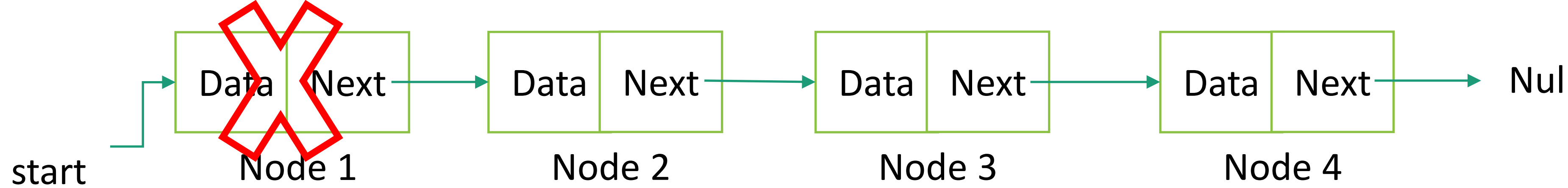


Algorithm For Deletion From Beginning

```
delbeg(start,first, node)

step-1 : first := &start
         node := start.next
step-2 : first → next = node → next
         free(node)
step-3 : return
```

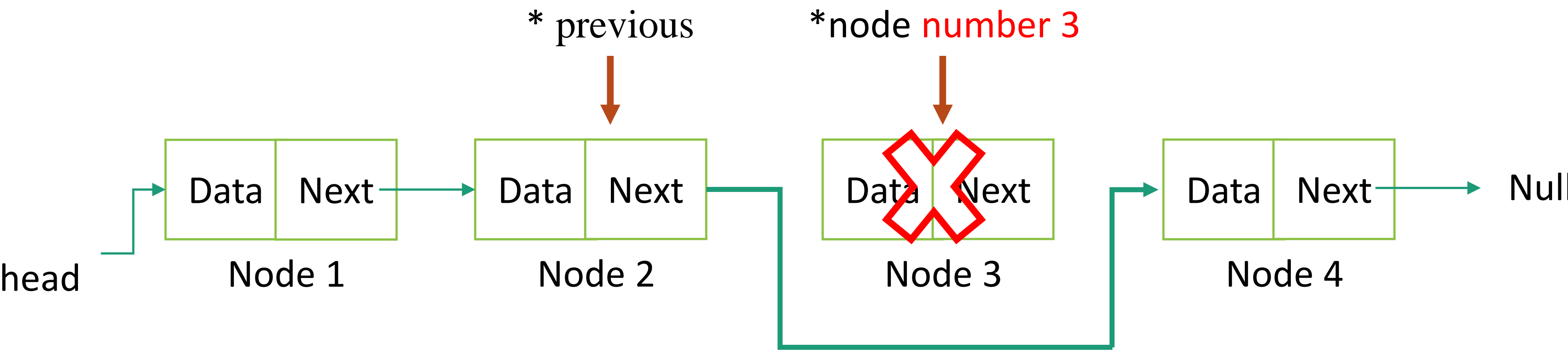
Example of Deleting a node in the beginning



Algorithm For Deletion Of Node When Node Number Is Known

```
delnode(start,previous,node,no) [no is the node number]
step-1 : previous := &start
         node := start.next
         count :=1
step-2 :  repeat while(node != null)
         if(count = no) then:
              previous → next =node → next
             free(node)
             return
         else :
             node = node → next
              previous := previous → next
             count:=count+1
step-3 : return
```

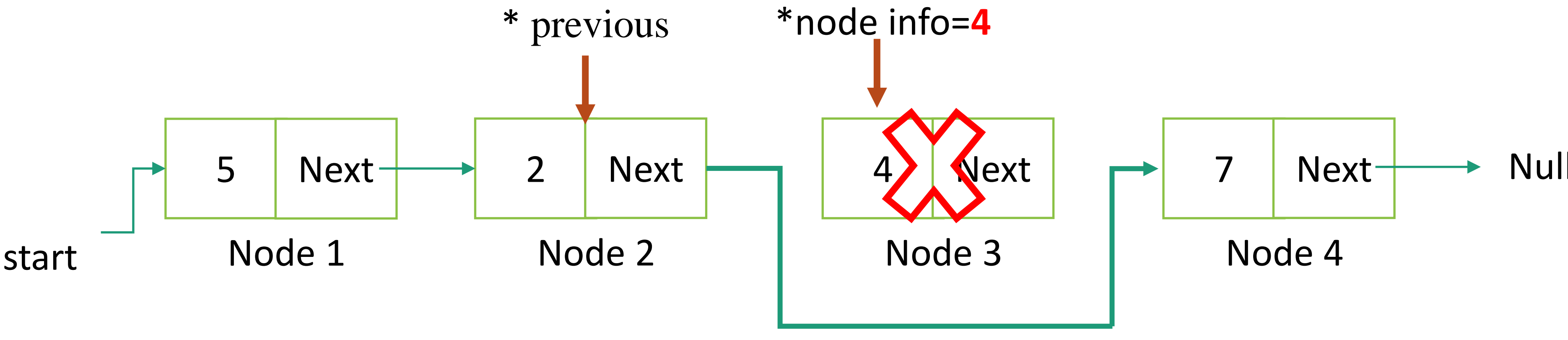
Example: delete a node at node number 3

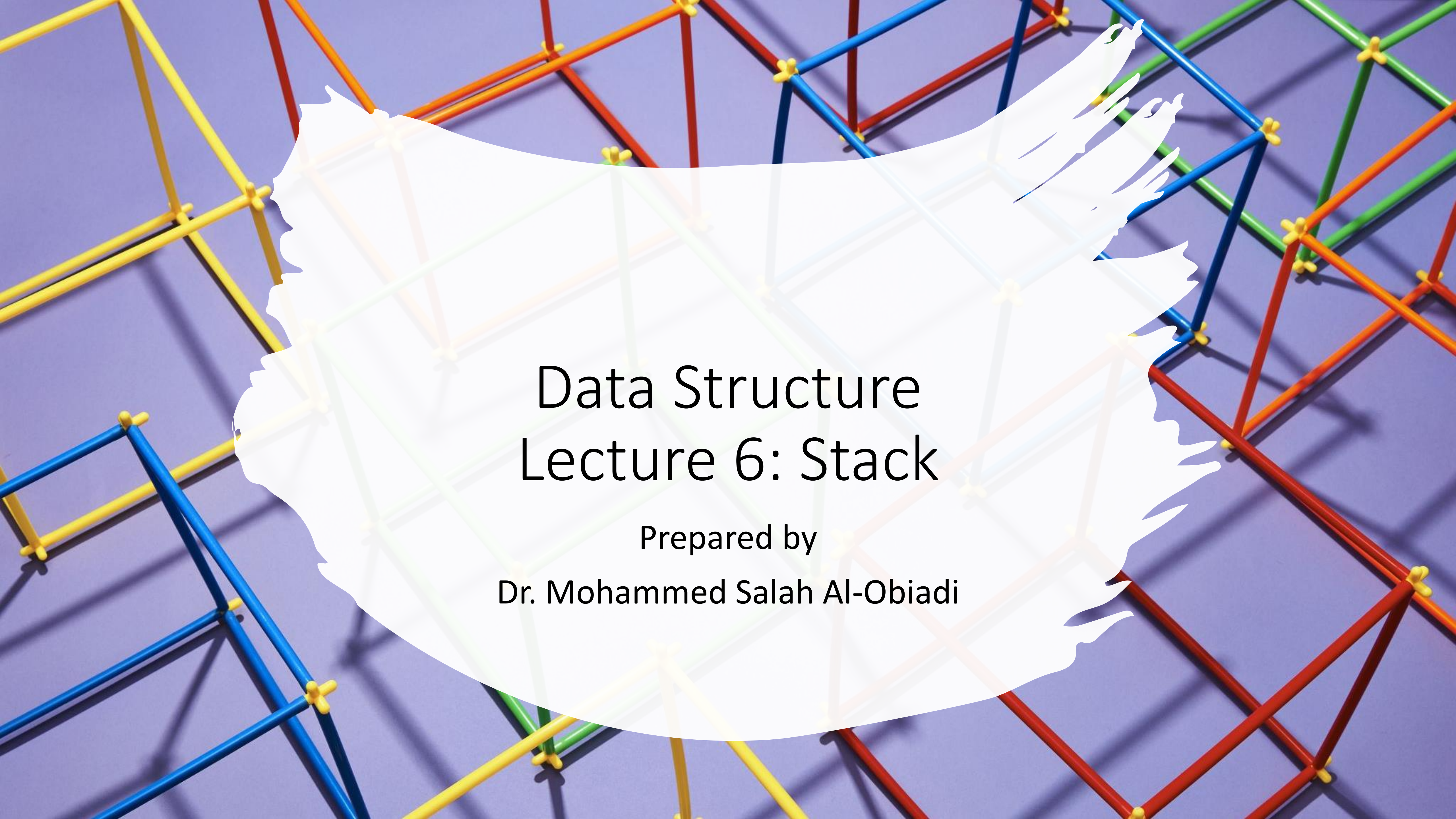


Algorithm For Deletion Of Node When Information Is Known

```
delinfo(start,previous,node,data) [data is the information to insert]
step-1 : previous = &start
         node = start.next
step-2 : repeat while(node != null)
           if(node → info = data) then:
               previous → next =node → next
               free(node)
               return
           else :
               node = node → next
               previous = previous → next
step-3 : return
```

Example: delete a node at info =4




Data Structure
Lecture 6: Stack
Prepared by
Dr. Mohammed Salah Al-Obiadi

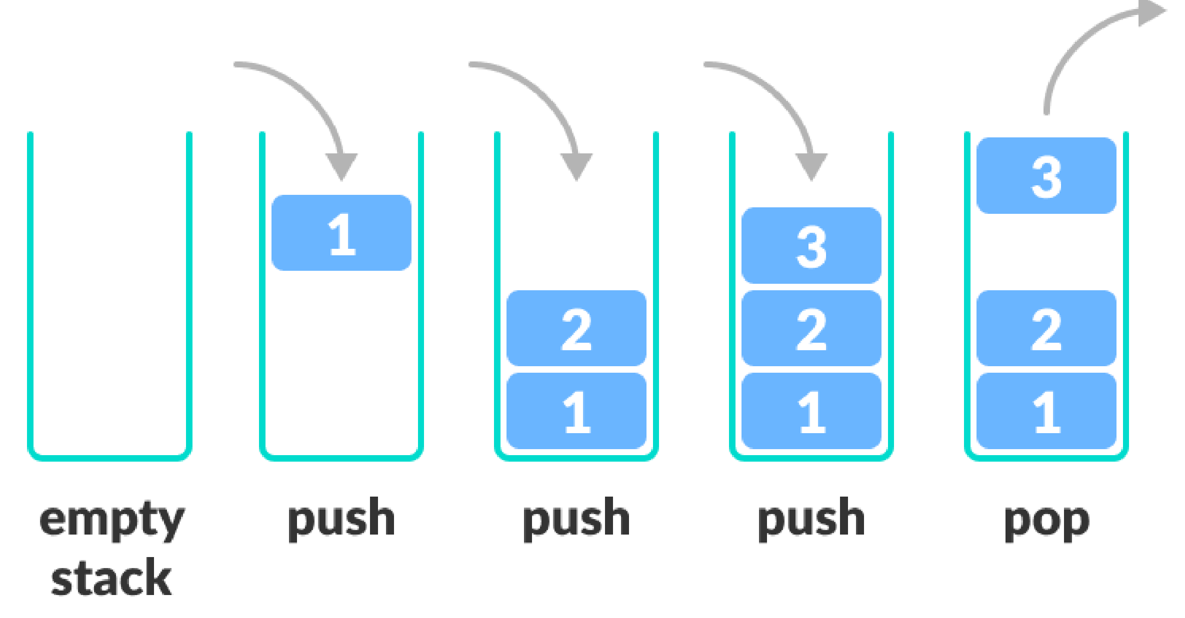
STACK

- Stack is a linear data structure.
- Follows the principle of LIFO **(Last in First Out).**
- Any data structure use the LIFO principle, it can be called as STACK.

Operations Performed With STACK

1- PUSH: which adds an element to the collection.

2- POP: which removes the most recently added element.



Overflow conditions

- During the **PUSH** (add) operation, we have to check the condition for overflow
- Condition for OVERFLOW
  - Top = size −1 (for the STACK starts with 0)
- Example of stack of size 6.
  - Now the stack has 6 items so we can't add any item.

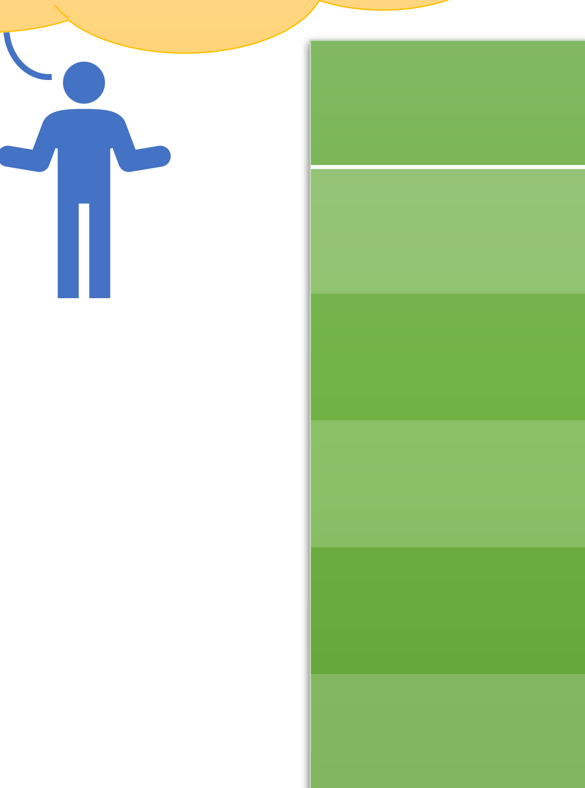
Item 6: We can't add Item 6 because Stack is full

Item 5 ← **Top**
Item 4
Item 3
Item 2
Item 1
Item 0

Underflow conditions

- During the **POP** (delete) operation, we have to check the condition for underflow.
- Condition for
  - Top =-1 (for the STACK starts with 0)
- Example of stack of size 6.
  - Now the stack is empty and Top=-1, so we can’t remove any item.

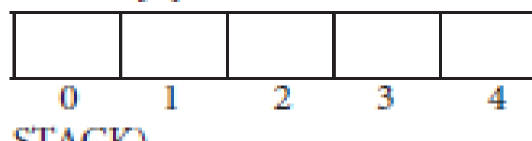
Stack is empty.
There is nothing to delete



-1 Top

EXAMPLES

STACK[5]



top = −1 (CONDITION FOR EMPTY STACK)

PUSH(5)

5				
0	1	2	3	4

top = 0

PUSH(25)

5	25			
0	1	2	3	4

top = 1

PUSH(53)

5	25	53		
0	1	2	3	4

top = 2

PUSH(78)

5	25	53	78	
0	1	2	3	4

top = 3

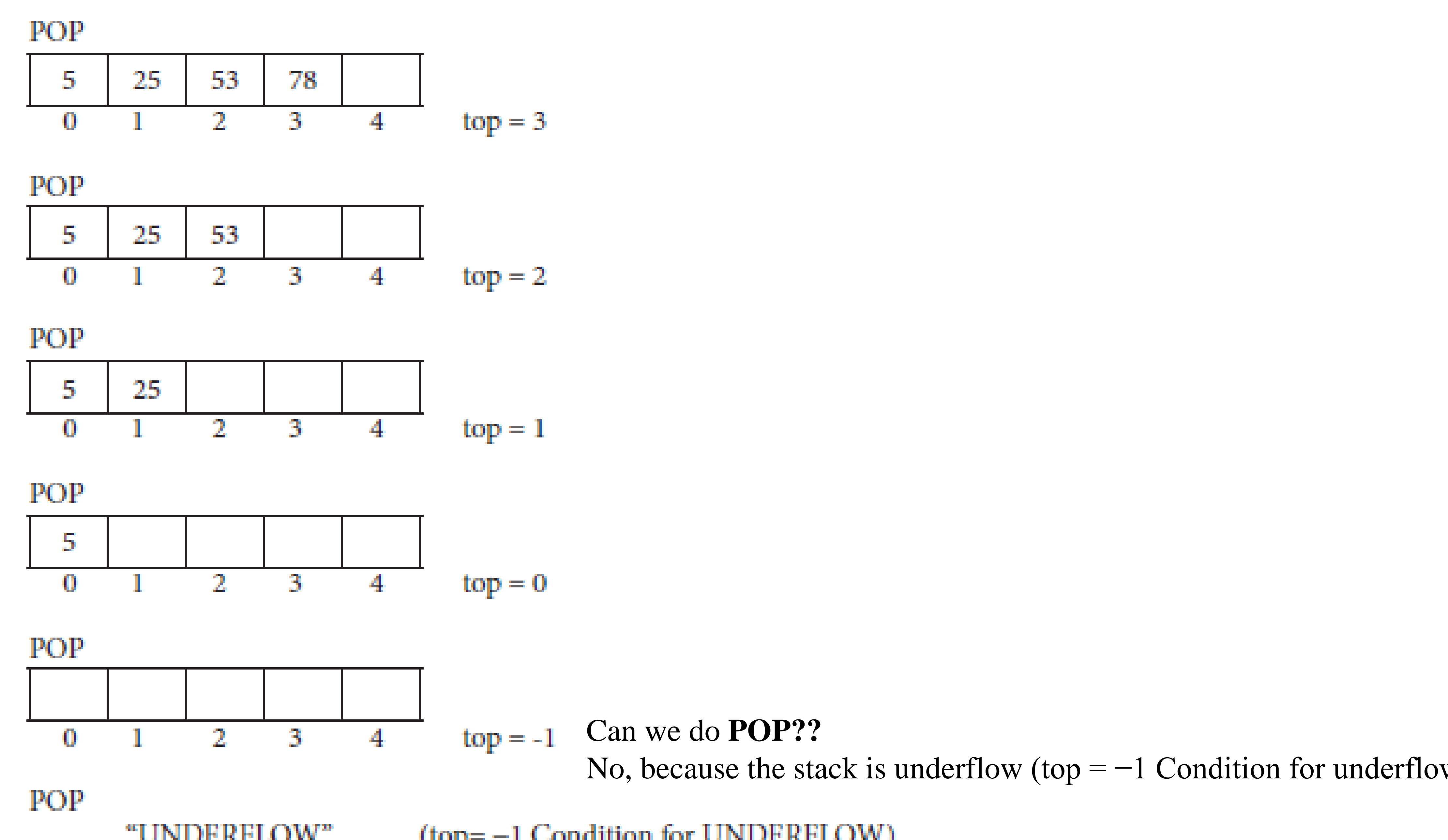
PUSH(99)

5	25	53	78	99
0	1	2	3	4

top = 4

Can we do **PUSH(76)??**

No, because OVERFLOW (top = size −1 Condition for OVERFLOW)



Can we do **POP??**

No, because the stack is underflow (top = −1 Condition for underflow)

Algorithm For Push Operation

PUSH(stack[size], no, top) [no is the number to insert] [top is the position of the stack]

step-1 : if (top = size - 1) then :

write : "overflow"

return

step-2 : top : = top +1

stack[top] := no

step-3 : return

Algorithm For POP Operation

pop(stack[size], top) [stack[size] is the stack] [top is the position of the stack]

Step-1 : if (top = - 1) then :

write : "underflow"

return

Step-2 : write : stack[top]

top := top -1

Step-3 : return

Algorithm For Traverse Operation

Traverse(stack[size], top)

Step-1 : if (top = - 1) then :

write : "stack is empty"

return

Step-2 : set i : = 0

Step-3 : repeat for i = top to 0 by -1

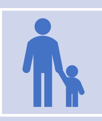
write : stack[i]

Step-4 : return

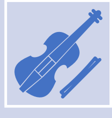
Algorithm For Update Operation

Can you do it?

Applications of STACK



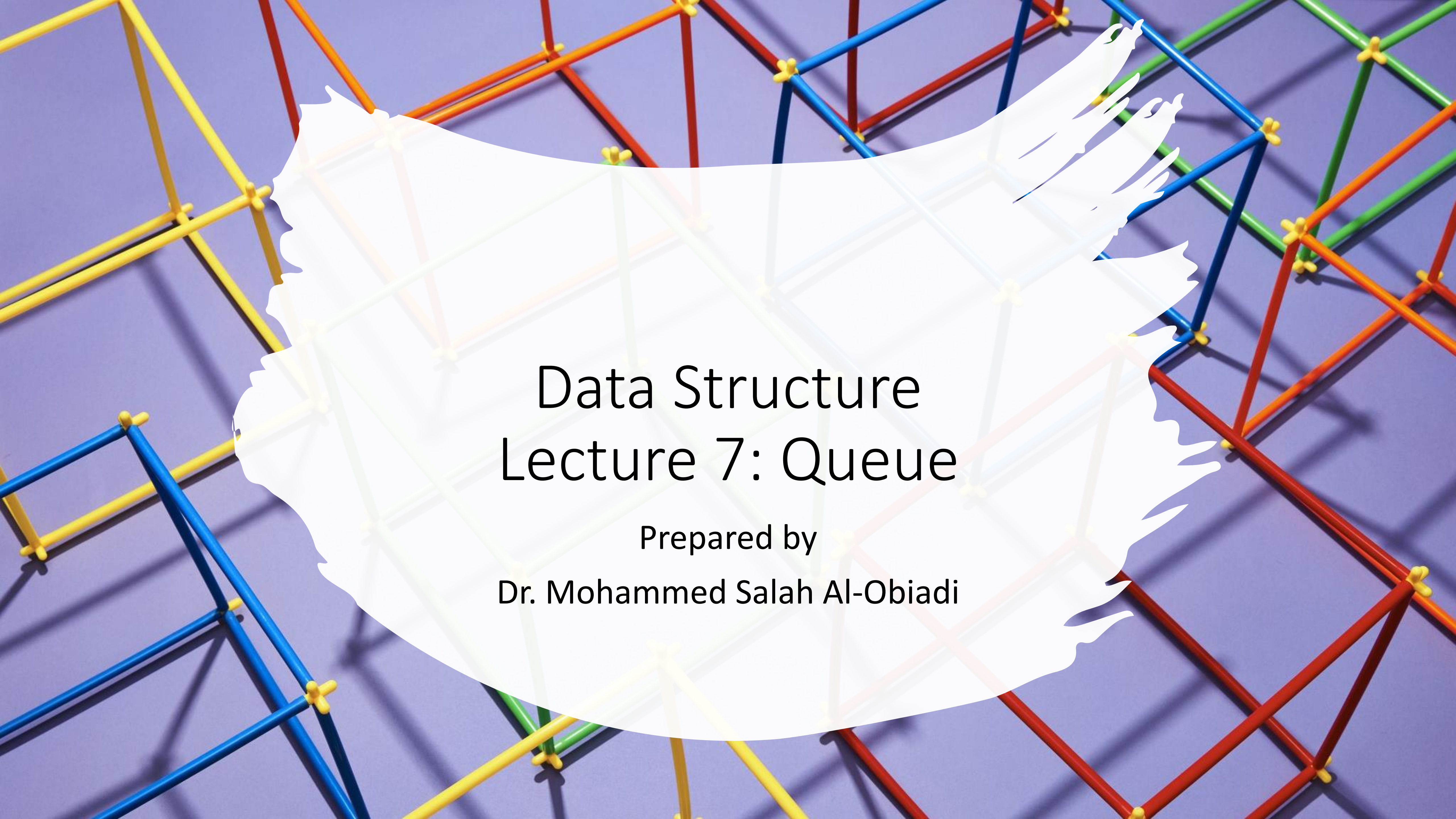
1- Checking of the parenthesis of an expression



2- Reversing of a string

3- In Recursion

4- Evaluation of Expression


Data Structure
Lecture 7: Queue
Prepared by
Dr. Mohammed Salah Al-Obiadi

Queue

In a stack, insertion and removal of the item was permitted only from one end.

Item inserted at last removed first from the stack.

How to ensure that the items are removed in the order they have inserted?? Solution is the Queue.

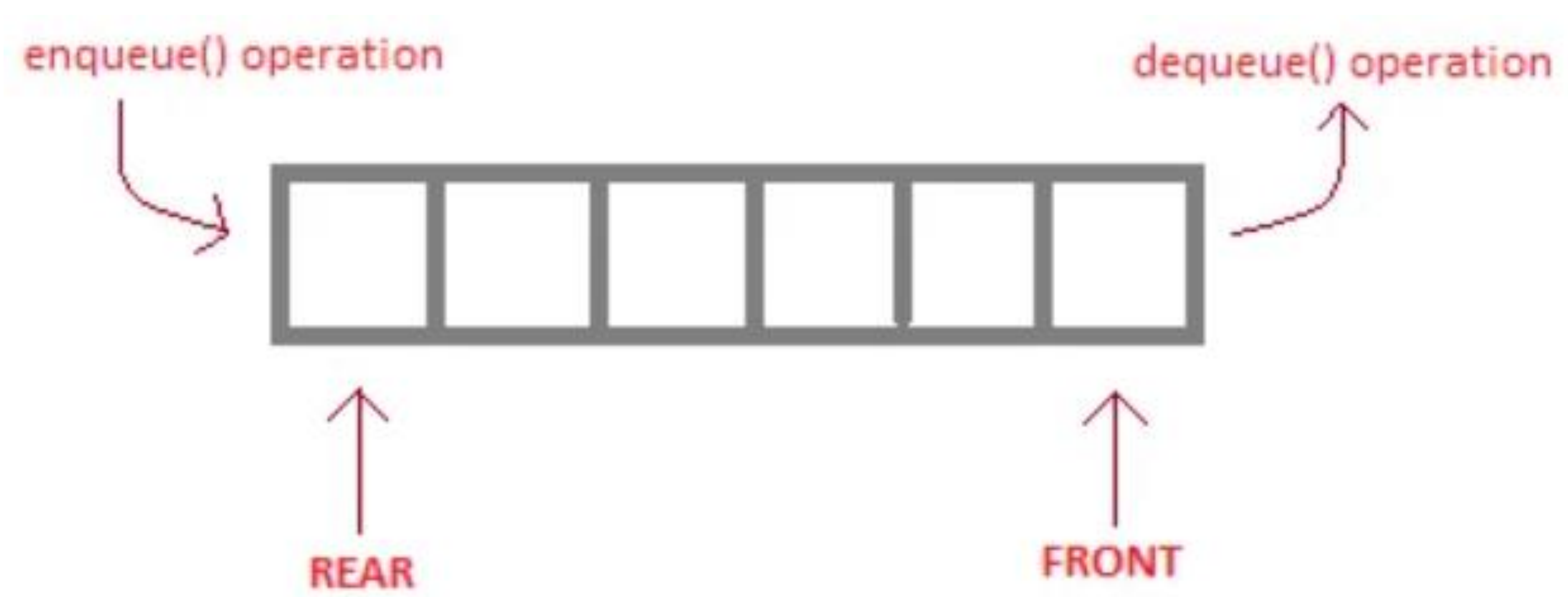
Queue is a data structure that can be considered as open from both the ends.

Queue follows the principle of **FIFO (First In First Out).**

Insertion is accomplished from one end known as **rear.**

Removal of the item is taking place on the other end known as **front**.

Queue



commonly implemented operations

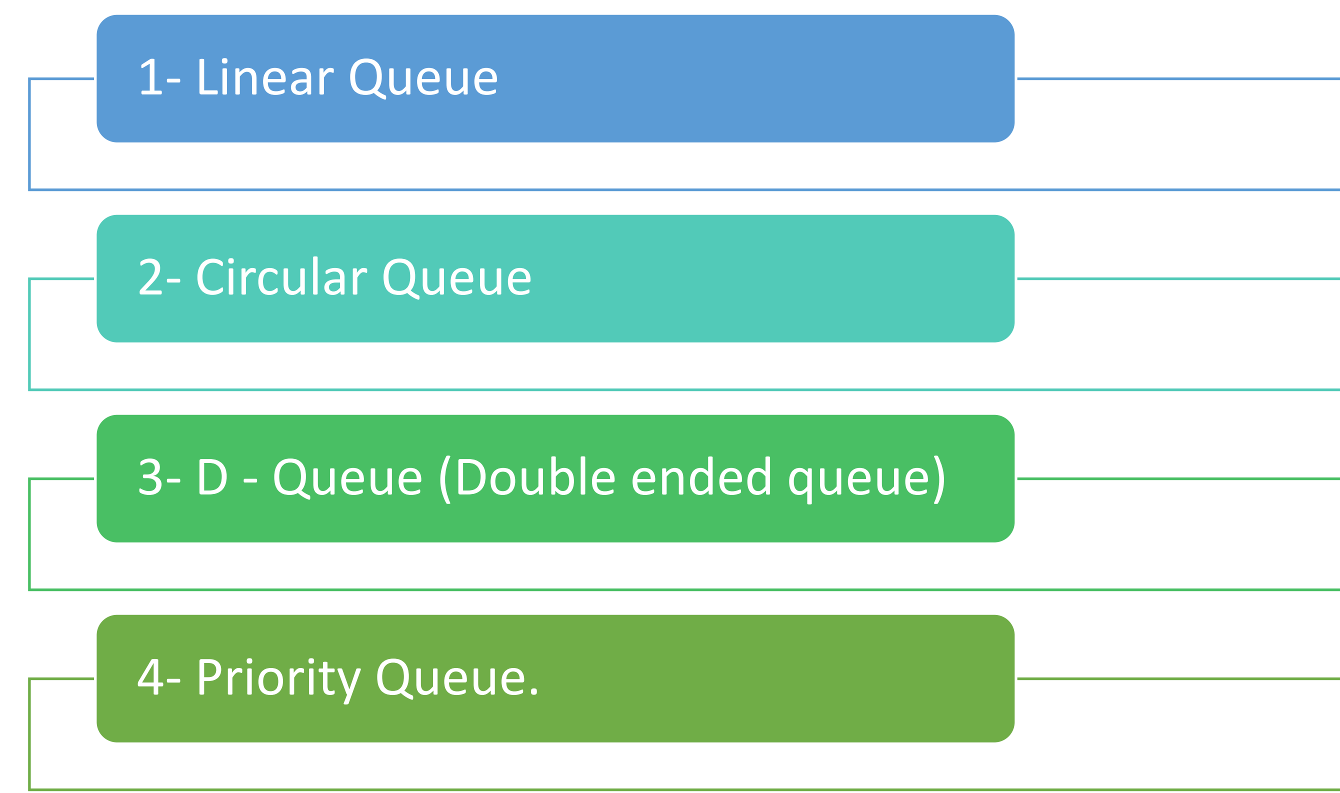
1- Insert.

- During the INSERT operation we have to check the condition for **OVERFLOW**

2- Delete.

- During the DELETE operation we have to check the condition for **UNDERFLOW**.

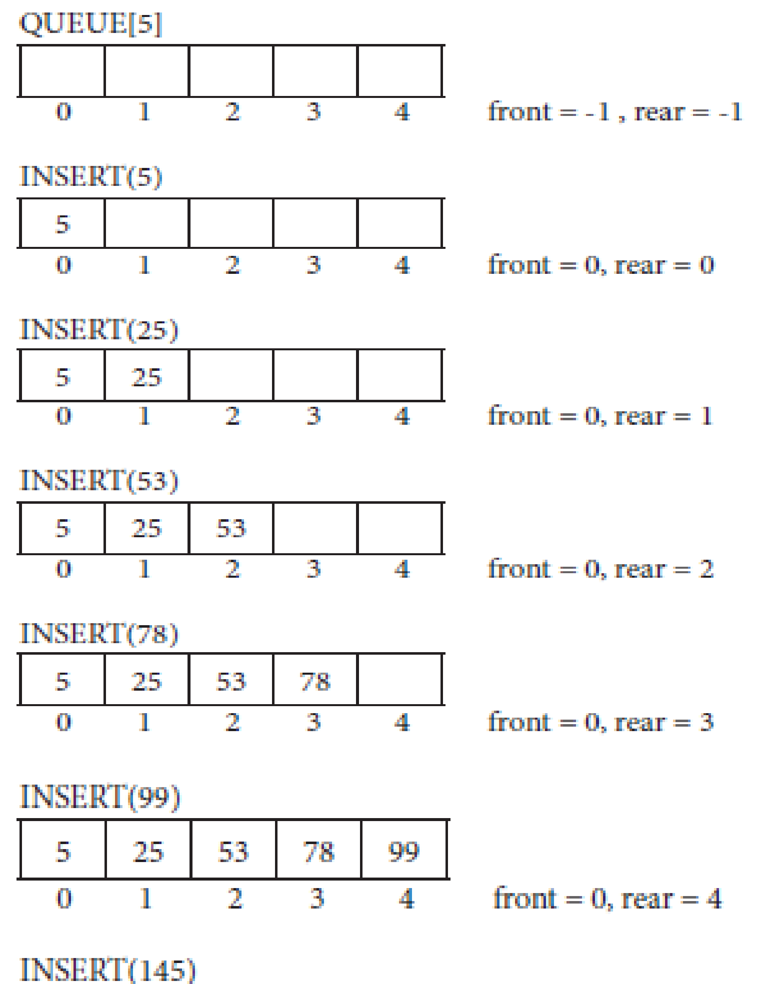
Types of Queue



Linear Queue

- A linear queue is a linear data structure that serves the request first, which has been arrived first.
- **Overflow**: insert an element with a filled QUEUE.
- Condition for OVERFLOW
  - Rear = size -1
- **UNDERFLOW:** delete an element from an empty QUEUE.
- Condition for UNDERFLOW
  - Front = -1 (for the QUEUE starts with 0)
- CONDITION FOR EMPTY QUEUE
  - Front = -1 and Rear = -1

Example: insert



Example: Delete

DELETE

	25	53	78	99
0	1	2	3	4

front = 1, rear = 4

DELETE

		53	78	99
0	1	2	3	4

front = 2, rear = 4

DELETE

			78	99
0	1	2	3	4

front = 3, rear = 4

DELETE

				99
0	1	2	3	4

front = 4, rear = 4

DELETE

0	1	2	3	4

front = -1, rear = -1

DELETE

"UNDERFLOW" (front = -1 Condition for UNDERFLOW

Algorithm For Insert Operation

- **Insert**(queue[size], front, rear, no)
- **Step 1 :** if (rear = size – 1) then :
    - write : "overflow"
    - return
- **Step 2 :** if (rear = -1) then :
    - front := 0
    - rear :=0
  - else:
    - rear :=rear+1
- **Step 3:** queue[rear] :=no
- **Step 4:** return

Algorithm For Delete Operation

- **Delet**(queue[size], front, rear)
- **Step 1 :** if (front = -1) then :
    - write : “underflow”
    - return
- **Step 2 :** write: queue[front]
- **Step 3 :** if (front ==rear) then :
        - front := -1
        - rear :=-1
    - else :
        - front := front +1
- **Step 4:** return

Algorithm For Traverse Operation

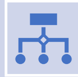
- **Traverse**(queue[size], front, rear)
- **Step 1 :** if (front = -1) then :
  - write : “ queue is empty ”
  - return
- **Step 2 :** set i:=0
- **Step 3 :** repeat for i = front to rear
  - write : queue[i]
- **Step 4:** return

Algorithm For Update Operation

- **Update**(queue[size], no, front, rear)
- **Step-1** : if (rear = - 1) then :
    - write : “stack is empty”
    - return
- **Step-2** : set i: =0
- **Step-3** : repeat for i = front to rear
    - if (no = queue[i]) then:
        - queue[i] = new no
        - return
    - if i=rear then:
        - write : “update not completed”
- **Step-4** : return

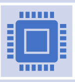
Applications of Queue

1- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).



2- Simulation of real-world queues such as lines at a ticket counter or any other first come first-served scenarios.

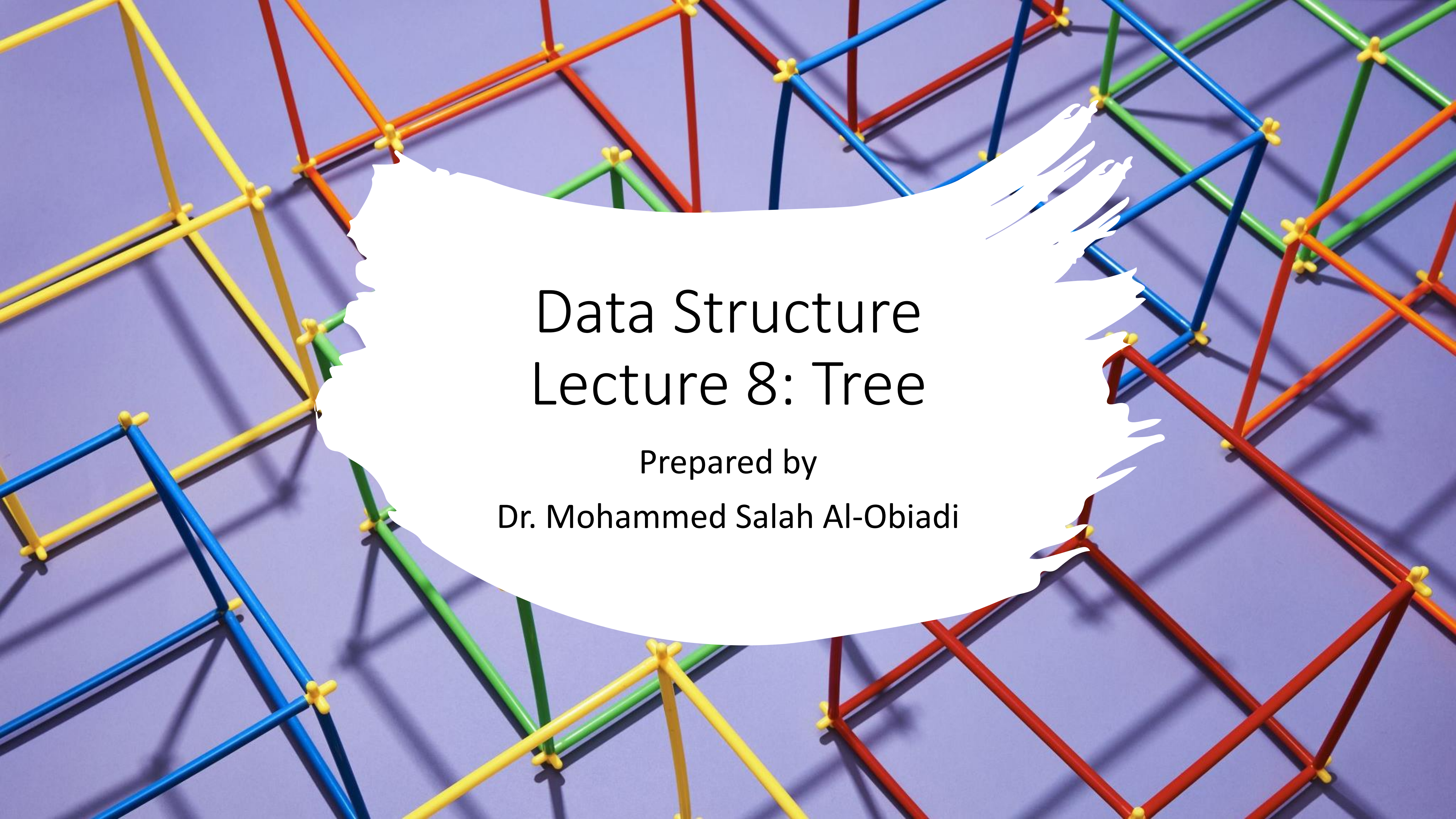
3- Multiprogramming.



4- Asynchronous data transfer (file IO, pipes, sockets).

5- Waiting times of customers at call center.

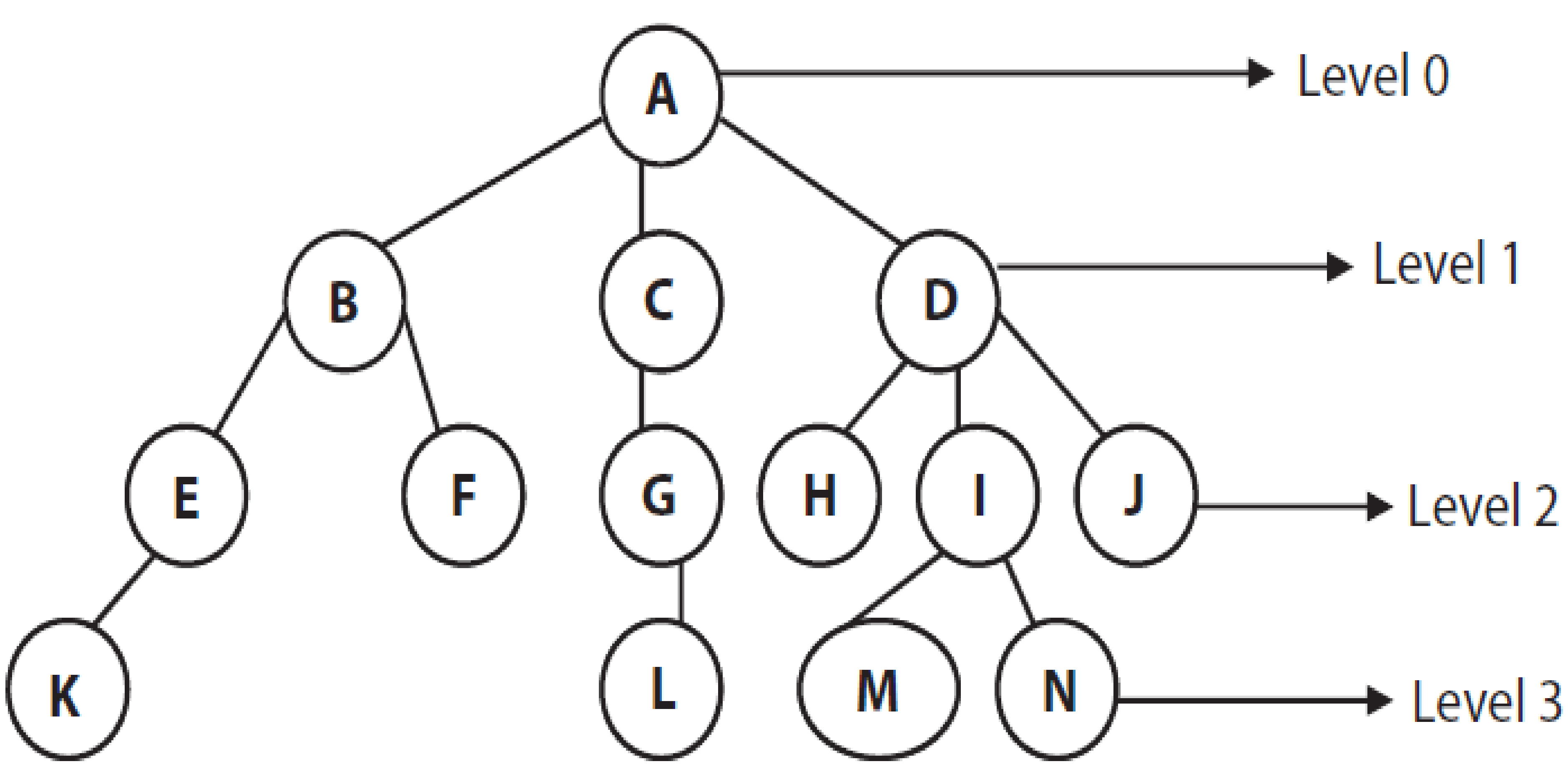
6- Determining number of cashiers to have at a supermarket.


Data Structure
Lecture 8: Tree
Prepared by
Dr. Mohammed Salah Al-Obiadi

TREE

- A TREE is a dynamic data structure that represents the hierarchical relationships between individual data items.
- It’s a data structure in which the elements are arranged in the parent and child relationship manner.
- In a tree, nodes are organized in a hierarchical way in such a way that:
  - Root is the beginning of the tree.
  - Branches are Lines that connecting the nodes.
  - Leaf nodes are nodes that have no children.

Figure 1 shows Example of a Tree



Tree Terminologies

Node: Each element of a tree is called as node. In the previous figure there are 14 nodes.

Root is the beginning of the tree. In figure 1: A is the root node.

Parent: Parent of a node is the immediate predecessor of a node. In figure 1: B is the parent of E and F.

Child: Each immediate successor of a node is known as child. In figure 1: B, C, D are children of A.

Siblings: The child nodes of a given parent node are called siblings. In figure 1: H, I, J are siblings.

Degree of a Node: The number of sub-trees of a node in a given tree. In figure 1:

- The degree of node A is 3
- The degree of node B is 2
- The degree of node G is 1
- The degree of node F is 0

Tree Terminologies

Degree of Tree: The maximum degree of nodes in a given tree. In the figure the maximum degree of nodes A and D is 3. So the degree of Tree is 3.

Terminal Node: A node with degree zero is called terminal node or a leaf.

Level: The entire tree structured is leveled in such a way that the root is always at the level 0, then its immediate children are at level 1, and their immediate children are at level 2 and so on up

Path: Path is the sequence of consecutive edges from the source node to the destination node path between A and M is (A,D),(D,I),(I,M).

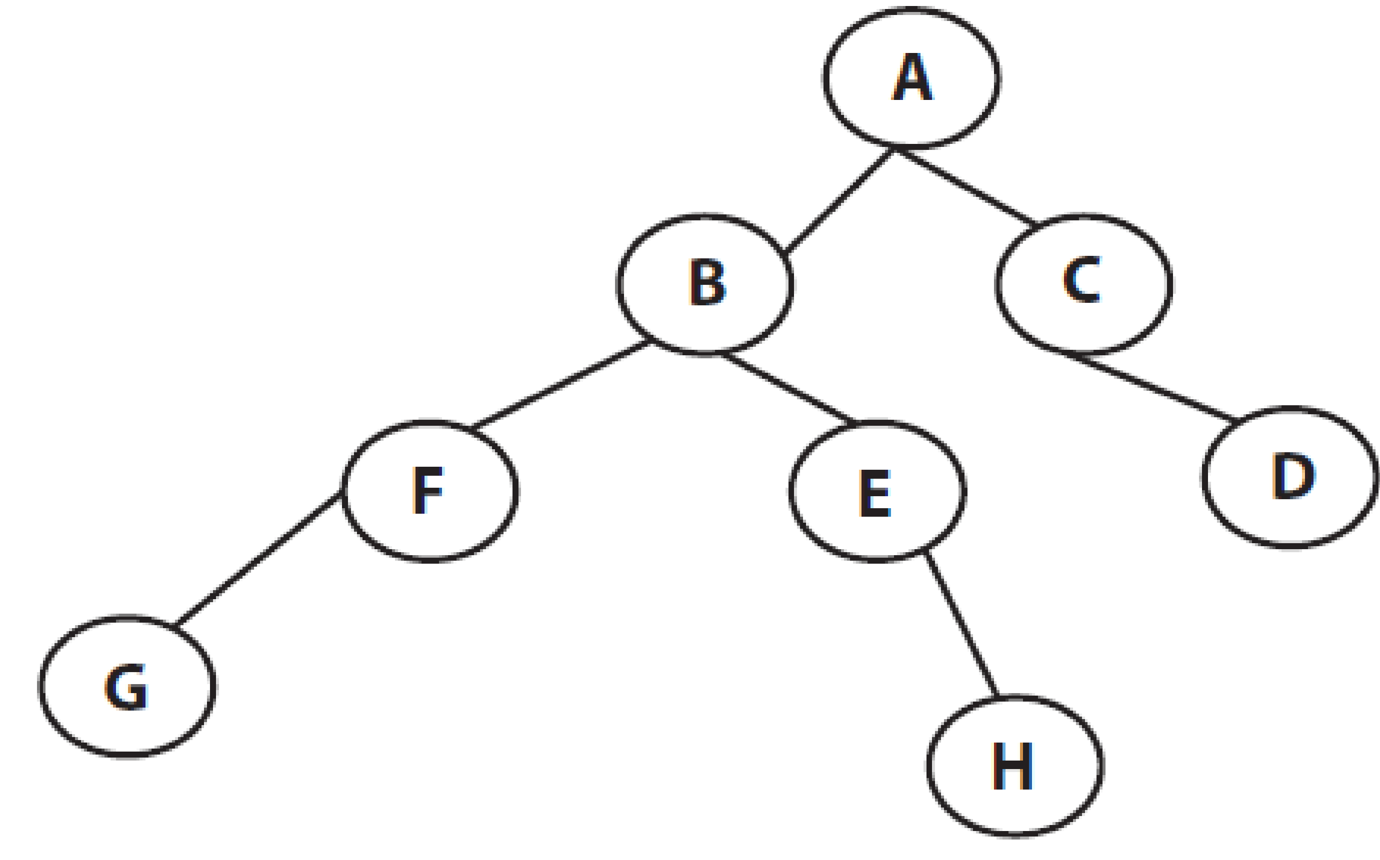
Height: The height of node n is the length of the longest path from n to leaf. The height of B is 2 and F is 0.

BINARY TREE

- A binary tree is a special form of a tree in which every node of the tree can have at most two children.

OR

- In a binary tree the degree of each node is less than or equal to 2.

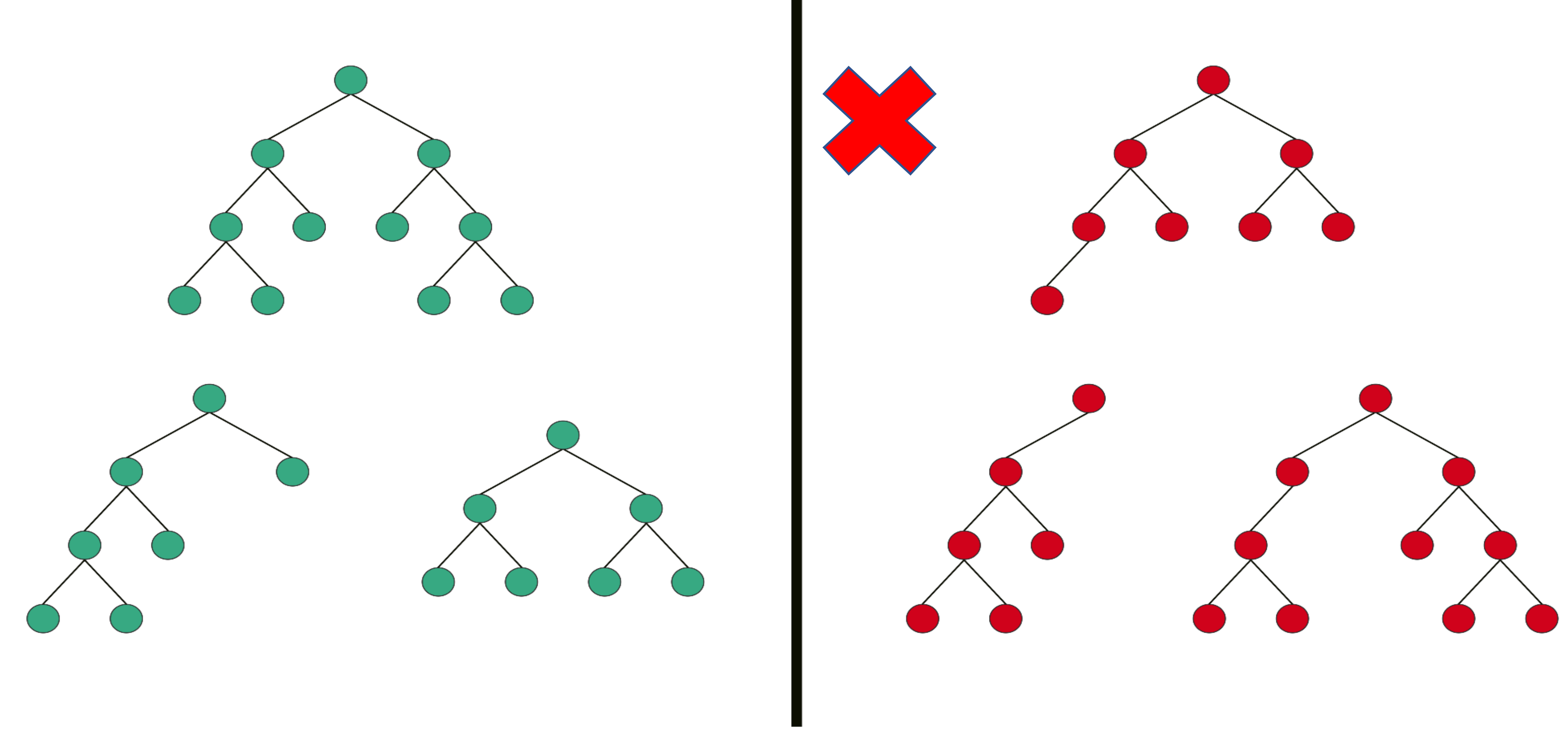


Types of Binary Tree

1. Full Binary Tree
2. Perfect Binary Tree
3. Pathological Binary Tree

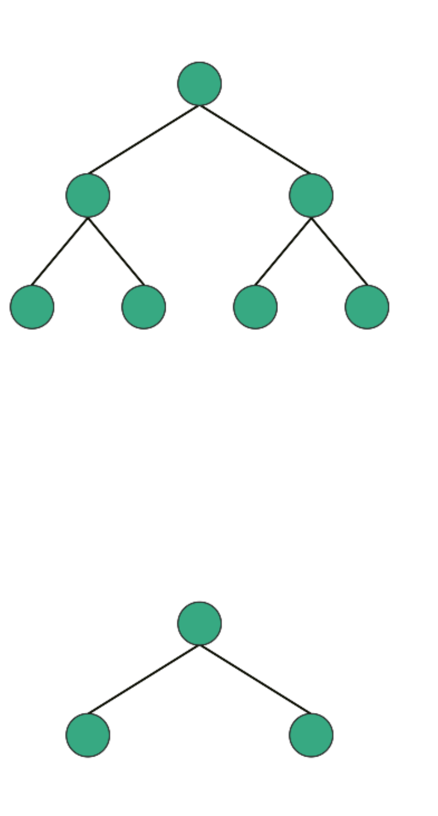
Full Binary Tree

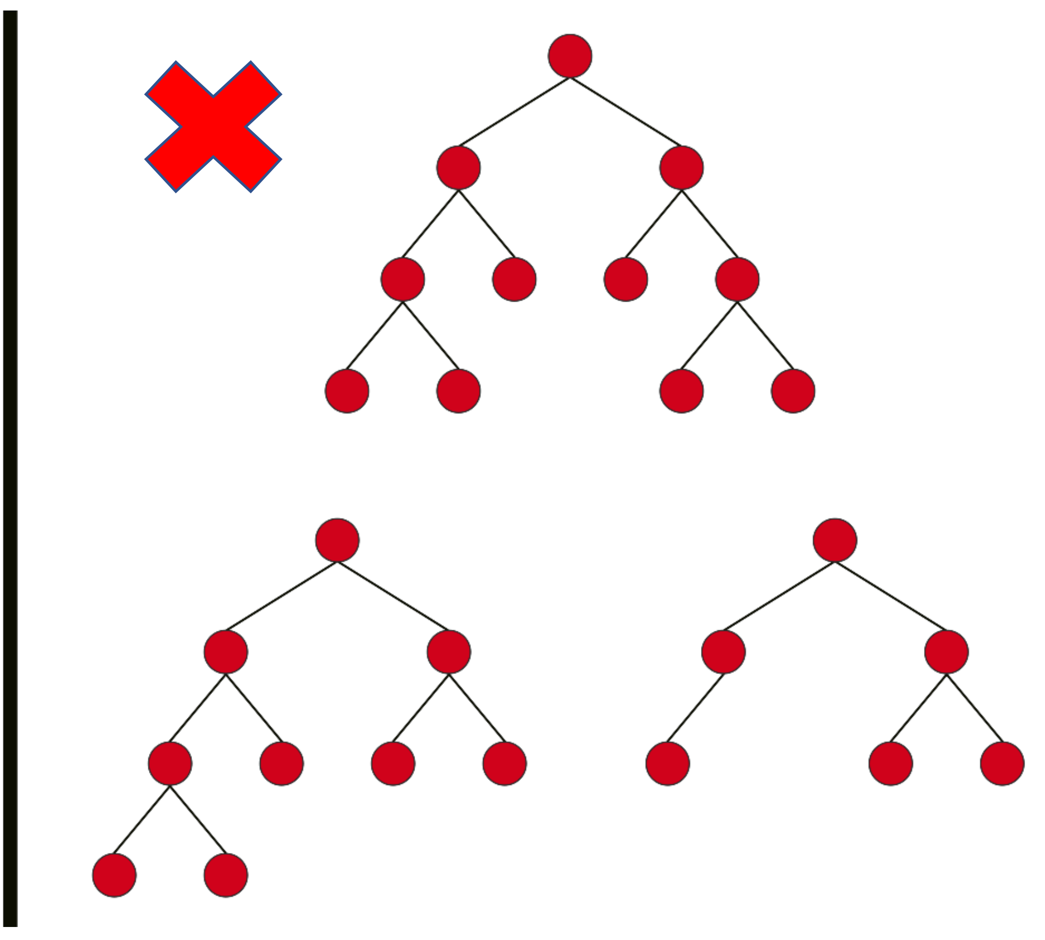
In a Full Binary Tree the out degree of every node is either 2 or Nil.



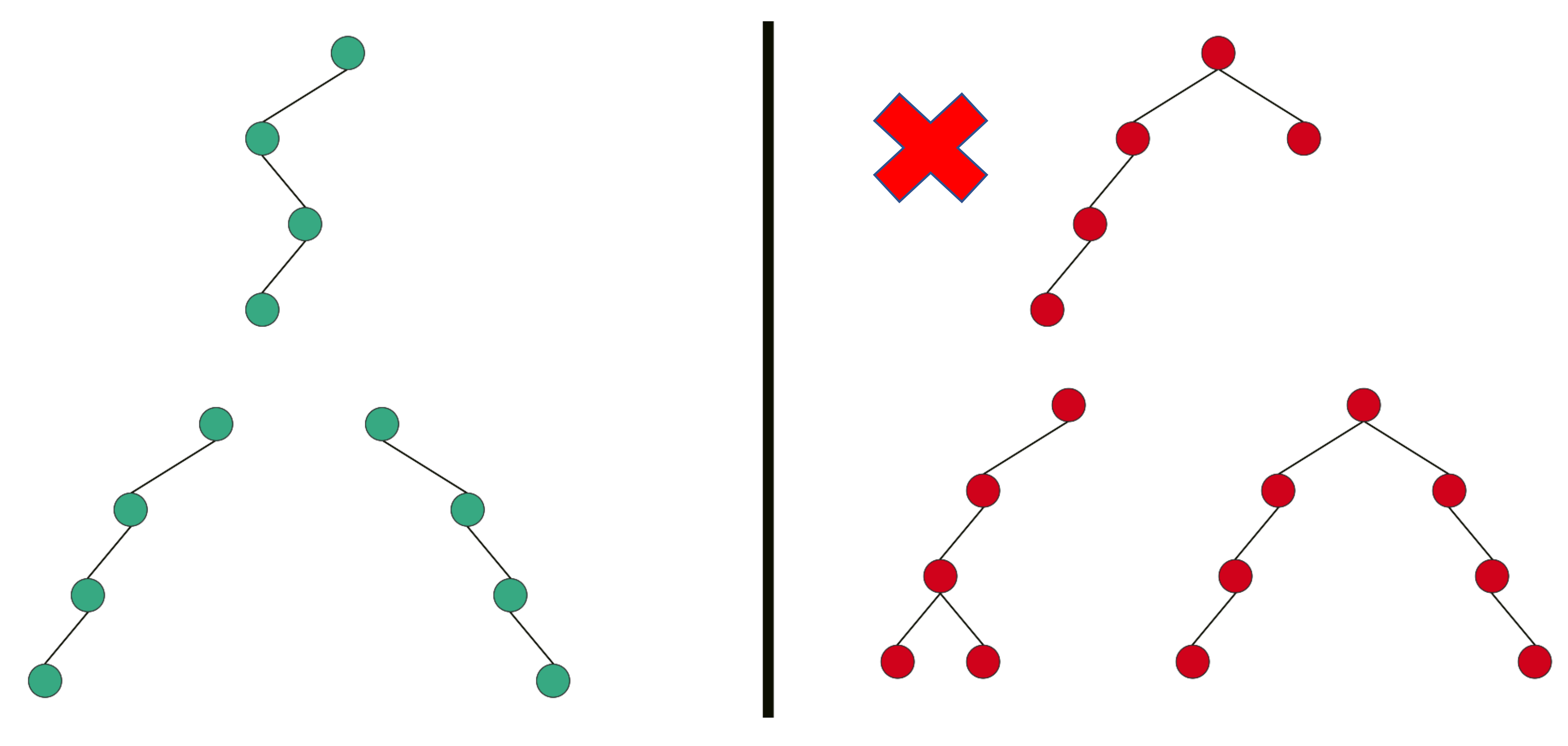
Perfect Binary Tree

Perfect Binary Tree is a Binary Tree in which all nodes have 2 children and all the leaf nodes are at the same depth or same level.



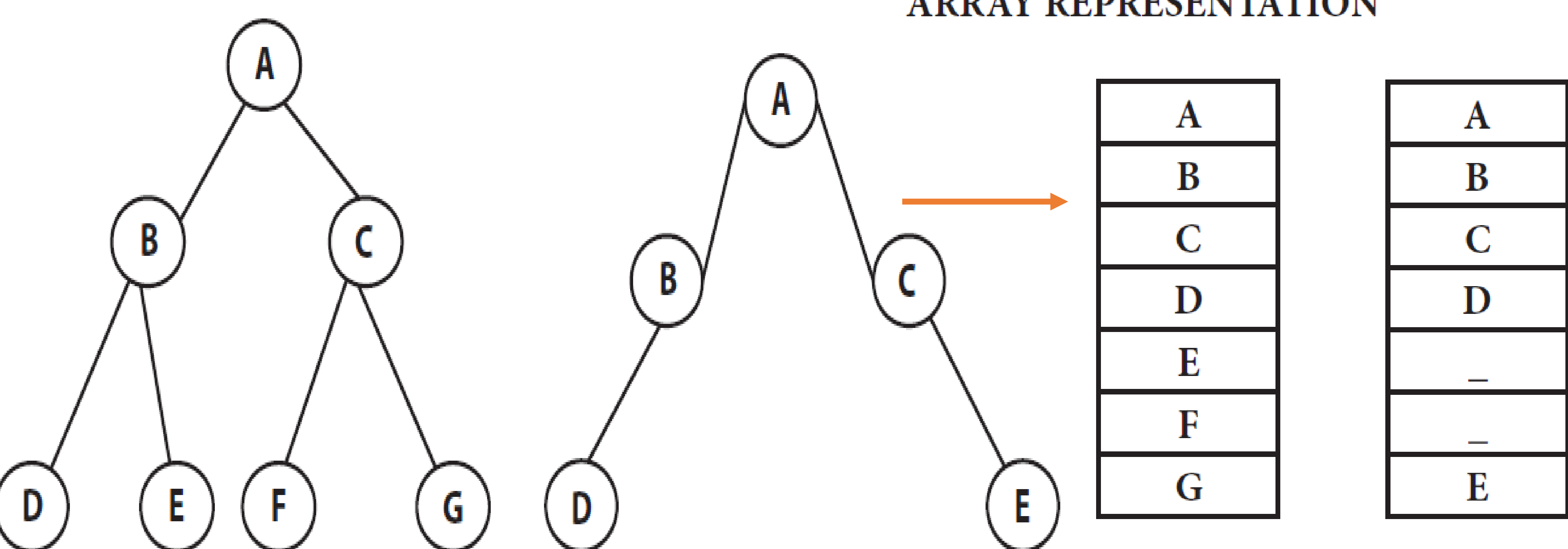


Pathological Binary Tree

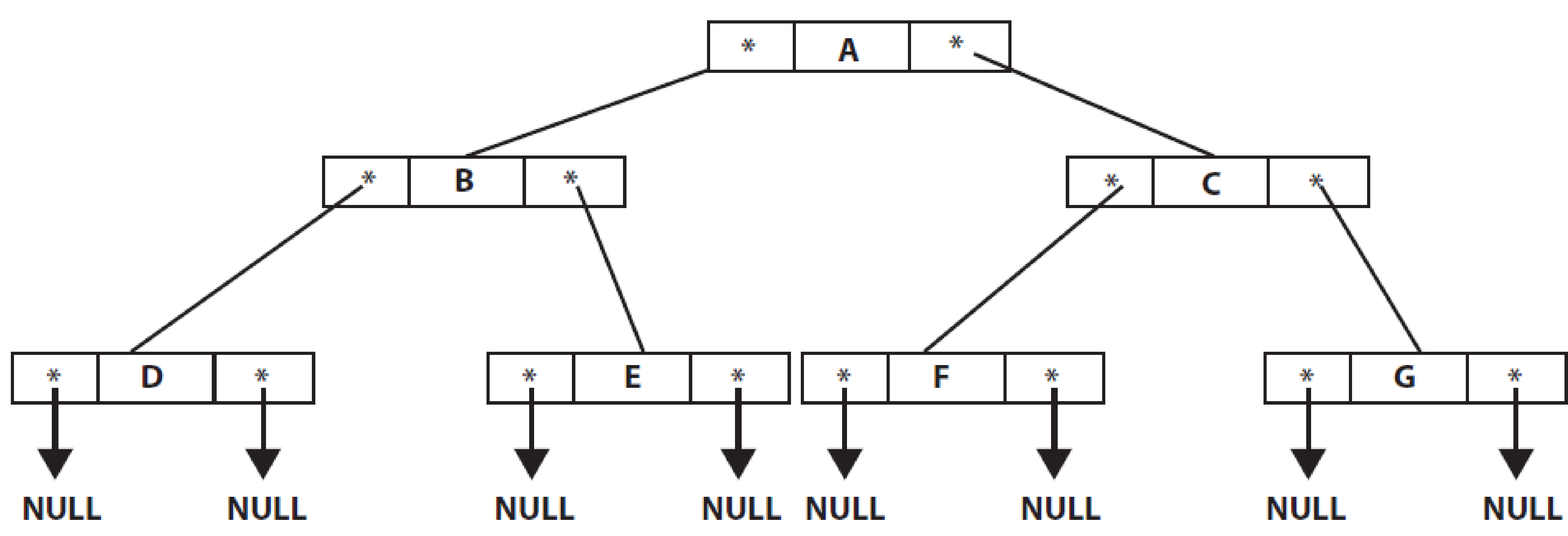


Array Representation of a Tree

- The ROOT node is always kept as the FIRST element of the array i.e/ in the 0-Index the root node will be store. Then, in the successive memory locations the left child and right child are stored.
- Example:



Linked List Representation of a Tree (Double Linked List)

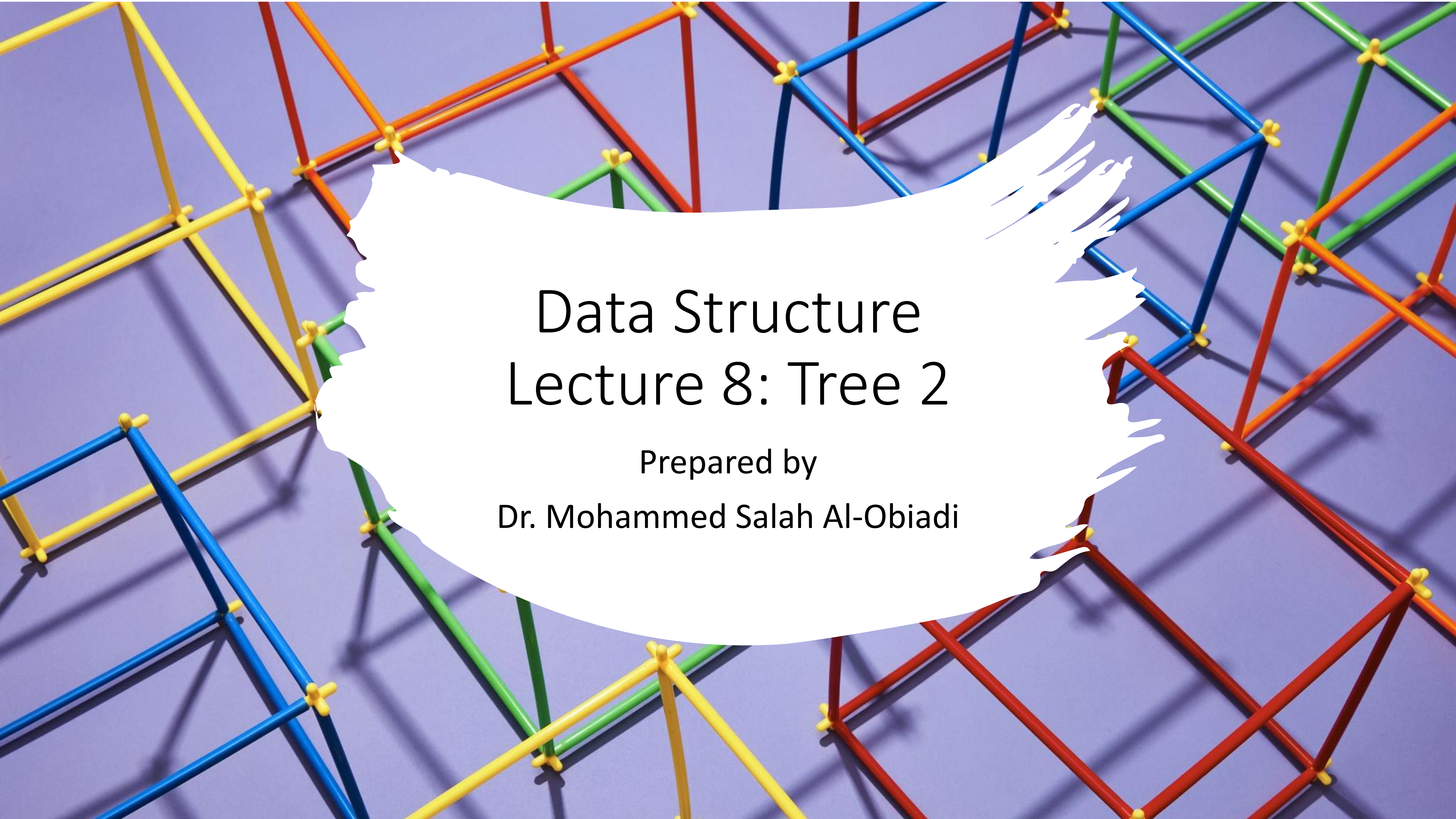


Operations Performed With the Binary Tree

- ➢Creation
- ➢Insertion
- ➢Deletion
- ➢Searching
- ➢Copying
- ➢Merging
- ➢Updating

Algorithm for Creation of Binary Tree

Create (node, info) [node is the structure having both left and right pointer. info is data]
Step-1 : if (node = null) then:
    Node := new Node() allocate a memory to node
    Node → info := info
    Node → left := null
    Node → right := null
    return
Step-2 : if node → info>= info then:
        create(node → left, info)
    else:
        create(node → right, info)
Step-3 : return(node)


Data Structure
Lecture 8: Tree 2
Prepared by
Dr. Mohammed Salah Al-Obiadi

Traversing With Tree

- The tree traversing is the way to visit all the nodes of the tree on a specific order.
- The Tree traversal can be accomplished in four different ways:
  - Inorder Traversal
  - Pre Order Traversal.
  - Post Order Traversal
  - Level Order Traversal

Inorder traversal

- Traverse the Left Subtree in **INORDER(Left)**
- Visit the Root node
- Traverse the Right Subtree in **INORDER(Right)**

Preorder Traversal

- Visit the Root Node
- Traverse the Right Subtree in **PREORDER(Left)**
- Traverse the Right Subtree in **PREORDER(Right)**

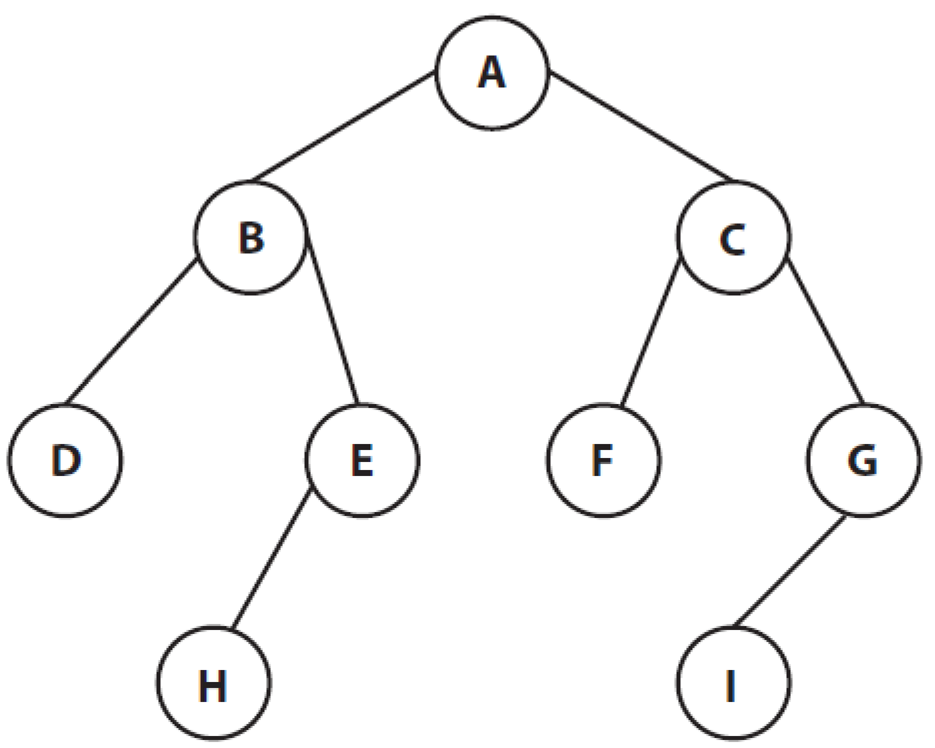
Postorder Traversal

- Traverse the Right Subtree in **POSTORDER(Left)**
- Traverse the Right Subtree in **POSTORDER(Right)**
- Visit the Root Node

Level Order Traversal

- In this type of traversal the elements will be visited according to level wise but it is not so far used.

Examples



1. Inorder : **D B H E A F C I G**
2. Preorder : **A B D E H C F G I**
3. Post Order : **D H E B F I G C A**
4. Level Order : **A B C D E F G H I**

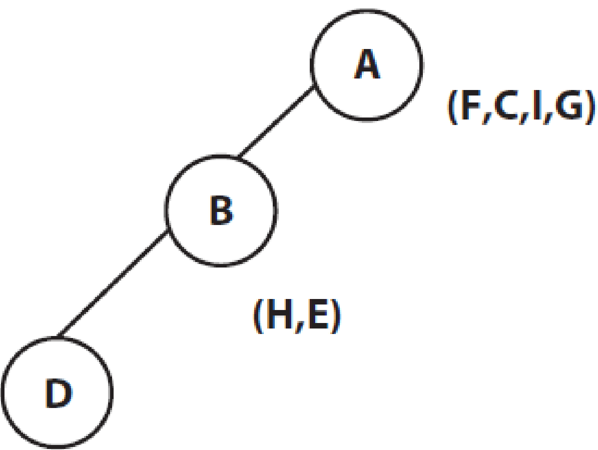
Conversion Of A Tree From Inorder And Preorder

- **INORDER : D B H E A F C I G**
- **PREORDER : A B D E H C F G I**
  - Choose the ROOT from the preorder and from inorder find the nodes in left and right and this process will continue up to all the elements are chosen from the preorder/inorder.
  - **STEP1:** From preorder A is the root and from inorder we will find that in the left of A (**D,B,H,E**) and in the right **(F,C,I,G):**

A

(D,B,H,E) (F,C,I,G)

  - **STEP2:** Again from Preorder 'B' will be chosen as PARENT and from Inorder in the left of B (**D**) and in the right **(H,E):**



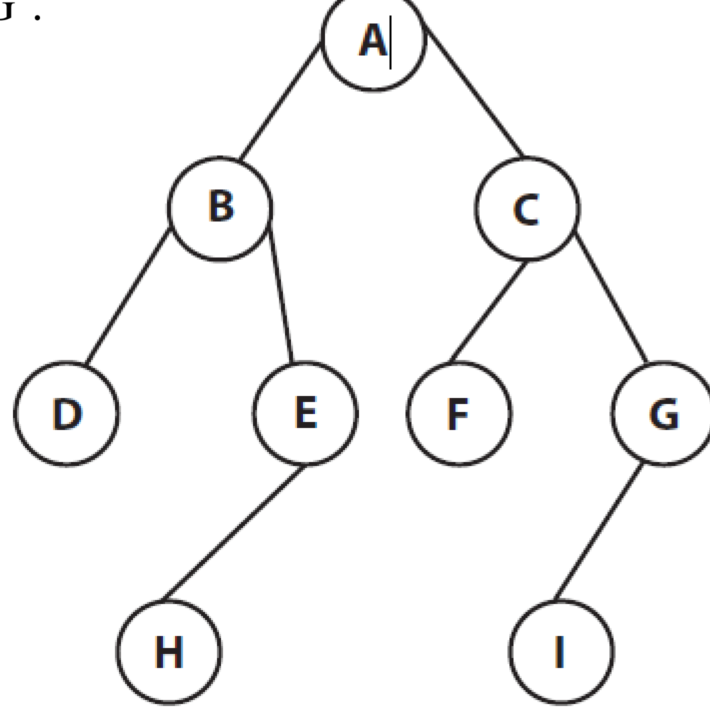
- **STEP3:** From Preorder 'E' will chosen as the PARENT and from inorder on its left 'H' is present:

(F,C,I,G)

- **STEP4:** From Preorder we will choose 'C' as the PARENT and from inorder we observe that in the left of 'C' (**F**) will placed and in the right (**I,G**)

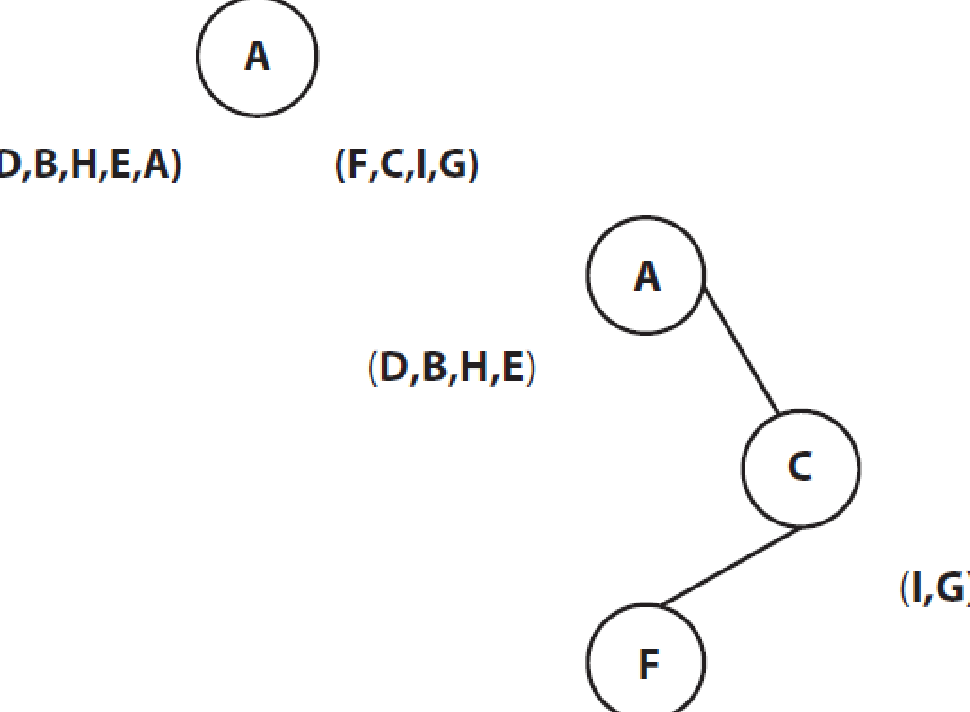
(I,G)

- **STEP5:** From the PREORDR we observe that 'G' is the parent and from the INORDER **I** will be used as the Left child of 'G'.



Conversion Of A Tree From Inorder And Postorder

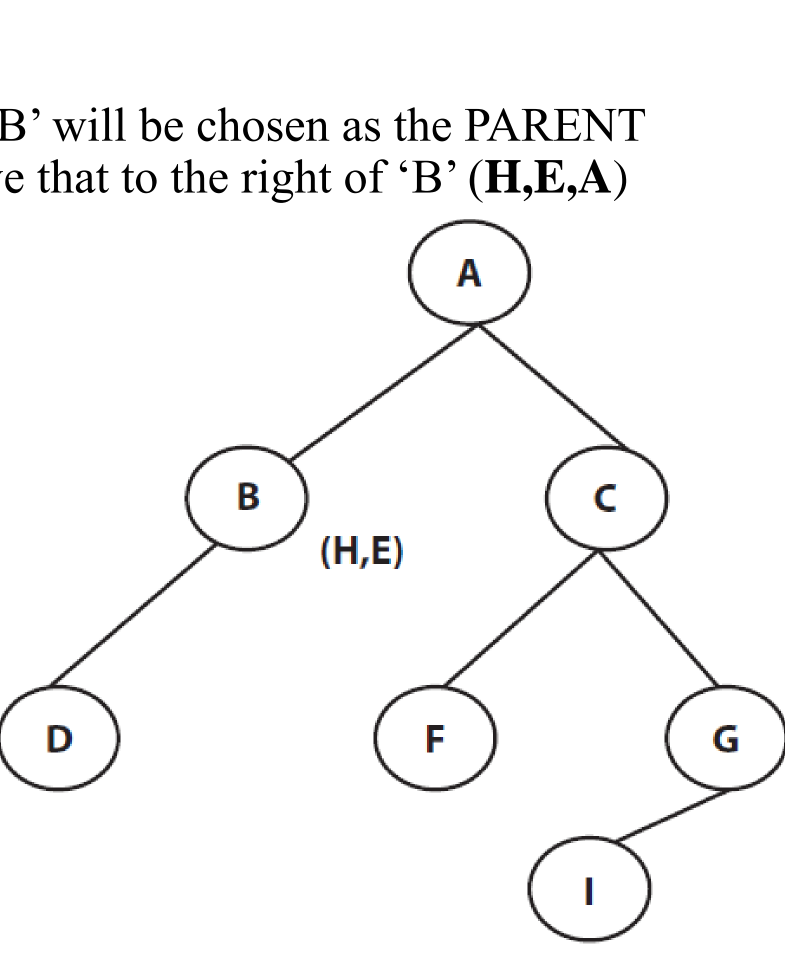
- **INORDER : D B H E A F C I G**
- **POST ORDER : D H E B F I G C A**
- Choose the **ROOT** from the postorder **(from the right)** and from inorder find the nodes in left and right and this process will continue up to all the elements are chosen from the postorder/inorder.
  - **STEP1 :** From the right of POSTORDER 'A' will be chosen as the ROOT and from INORDER we observe that in the left of A (**D,B,H,E,A**) and in the right (**F,C,I,G**) will be there.
  - **STEP2:** From the POSTORDER 'C' will be chosen as the PARENT and from INORDER we observe that in the right of 'C' (**I,G**) and to the left (**F**) will be used.



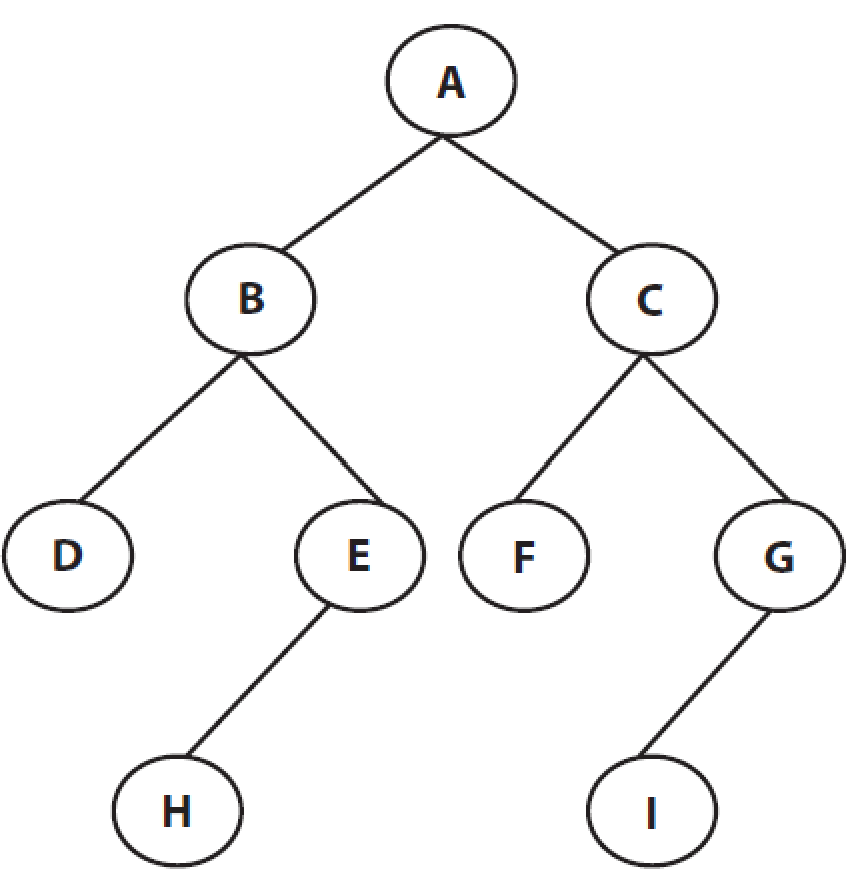
- **STEP3:** From the right of POSTORDER ‘G’ will be chosen as the PARENT and from inorder to the left of ‘G’ (**I**) will be used.

(D,B,H,E)

- **STEP4:** From the right to postorder ‘B’ will be chosen as the PARENT and from the INORDER to we observe that to the right of ‘B’ (**H,E,A**) and to the left (**D**) will be used.



- **STEP5:** From the right to postorder we will choose ‘E’ as PARENT and from the Inorder to the left of ‘E’ (**H**) will be used.



Applications of Binary Tree

- Expression Tree
- Binary Search Tree
- Height Balanced Tree (AVL Tree)
- Threaded Binary Tree
- Heap Tree
- Huffman Tree
- Decision Tree
- Red Black Tree

Expression Tree

- An expression tree is a Binary Tree which stores/represents the mathematical (arithmetic) expressions.
- The leaves of an expression tree are operands, such as constants or variable names and all the internal nodes are the operators.
- Formally we can define an expression Tree as a special kind of binary tree in which:
  - Each leaf is an operand. Examples: a, b, c, 6, 100
  - The root and internal nodes are operators. Examples: +, -, *, /, ^
  - Subtrees are subexpressions with the root being an operator.

EXAMPLE

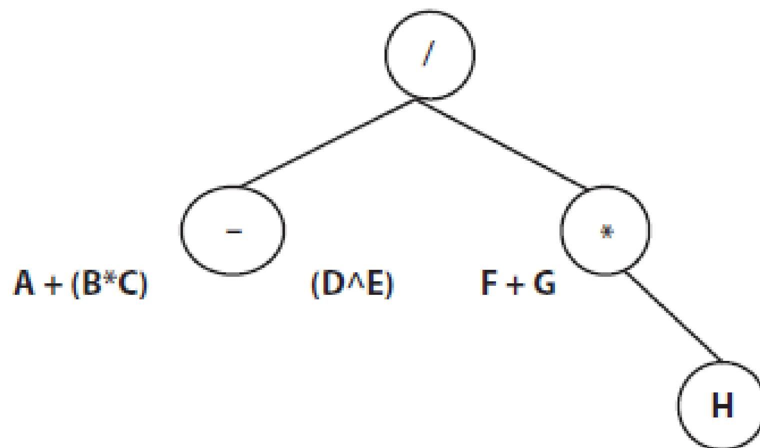
- Represent an Expression Tree
    - **A + (B*C) – (D^E) / F + G * H**
- choose an operator in such a way that the terms in parenthesis will be in a side
- Choose a operator having higher precedence.

STEP1:

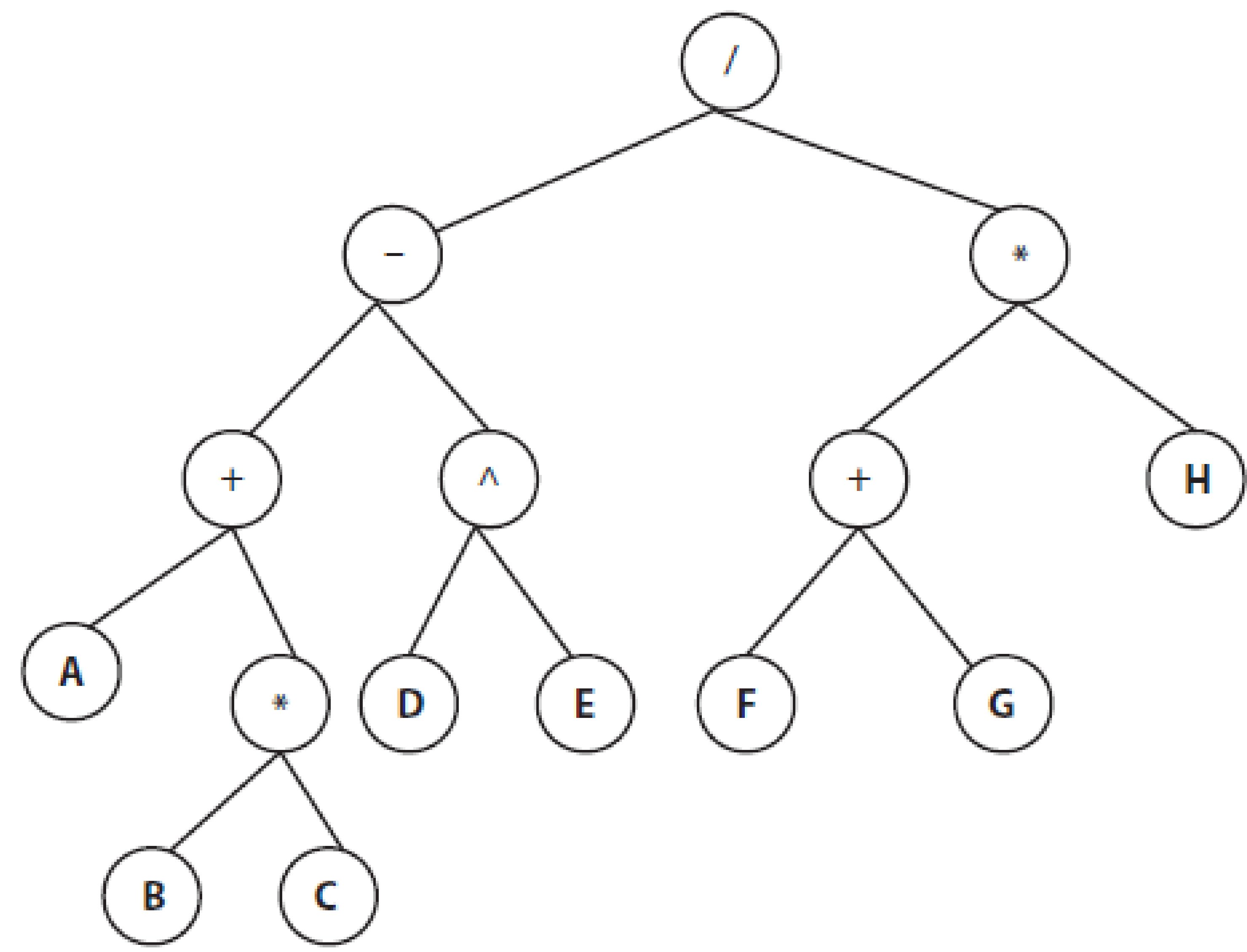
/

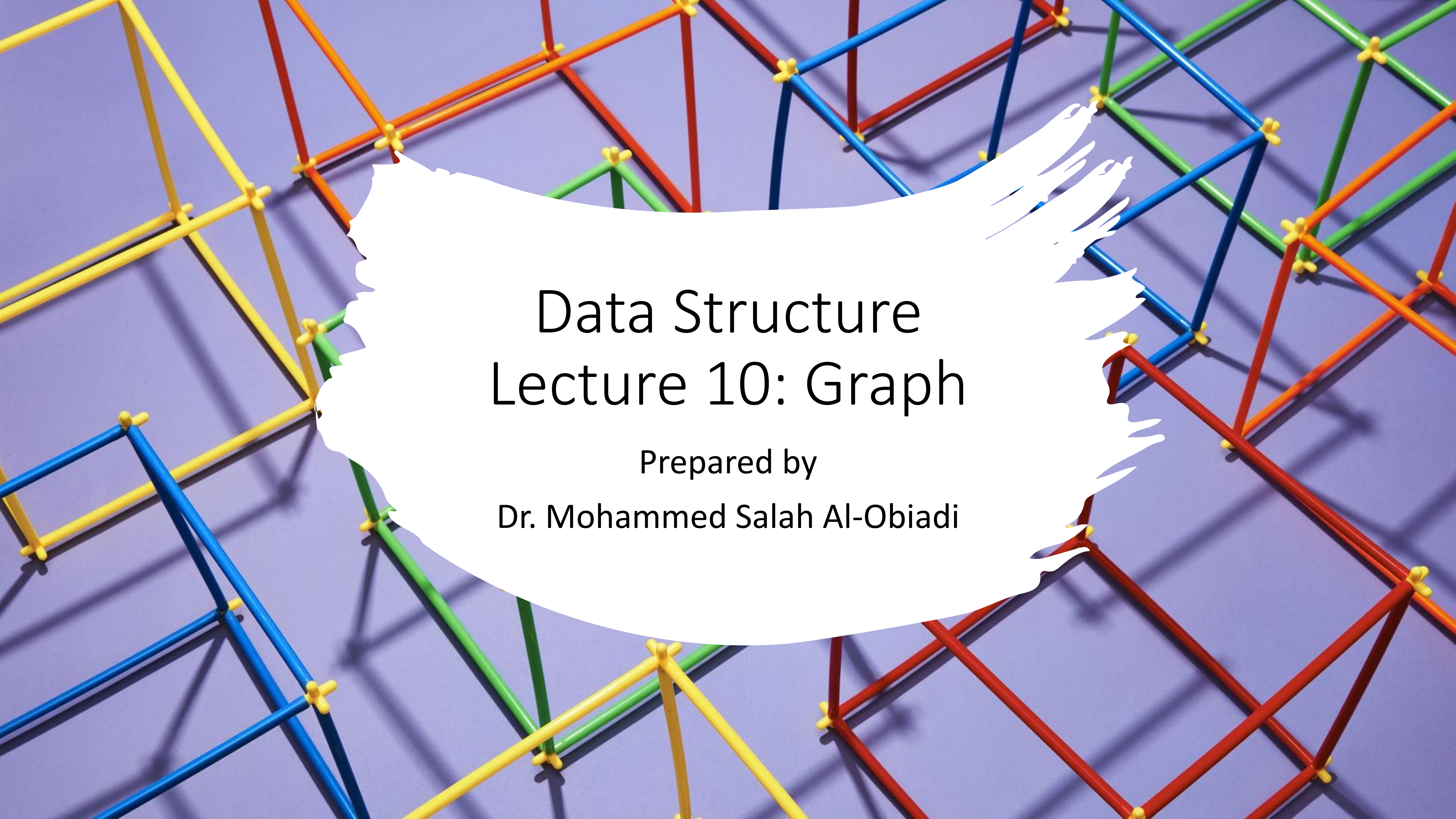
A + (B*C) – (D^E)   F + G * H

STEP2:



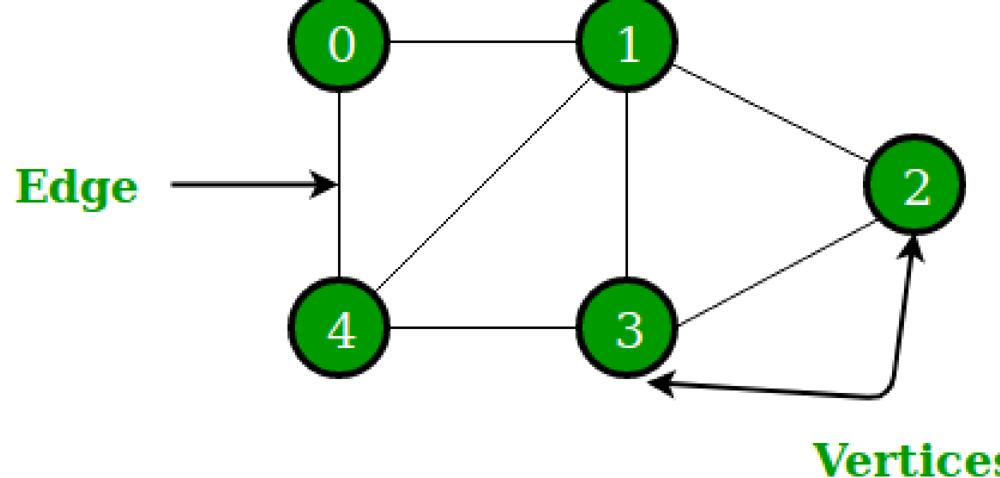
Step3:




Data Structure
Lecture 10: Graph
Prepared by
Dr. Mohammed Salah Al-Obiadi

Graph

- GRAPH is a non-linear data structure in which the elements are arranged randomly inside the memory and are interconnected with each other
- A graph G is an ordered pair of sets (V,E) where
  - V is the set of vertices and
  - E is the edges which connect the vertices.

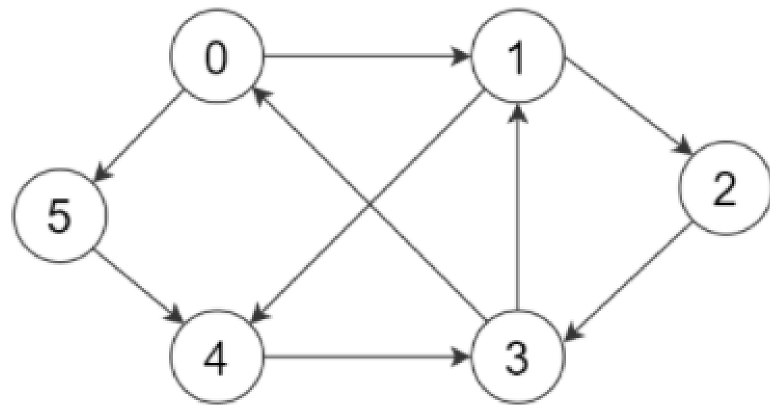


Applications of Graph

- **Google maps** uses graphs for building transportation systems.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them.
- In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u.
- **Path Optimization Algorithms**, Path optimizations are primarily occupied with finding the best connection that fits some predefined criteria.
- **Recommendation Search Engines:** google uses graph to represent pages and their importance.

Graph Terminologies

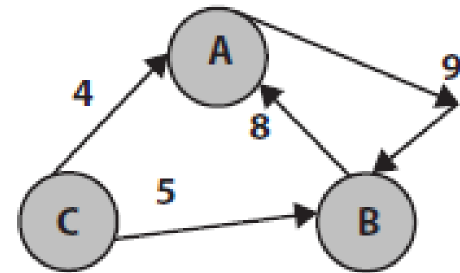
- Directed Graph: A graph in which every edge is directed is called undirected graph.

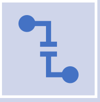


- Undirected Graph: A graph in which every edge is undirected

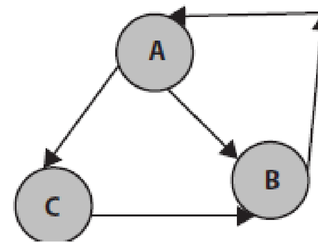
Graph Terminologies

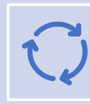
WEIGHTED GRAPH: A graph is said to be weighted if its edges have been assigned some non-negative value as weight.



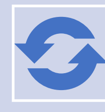


Path is the sequence of consecutive edges from the source node to the destination node.

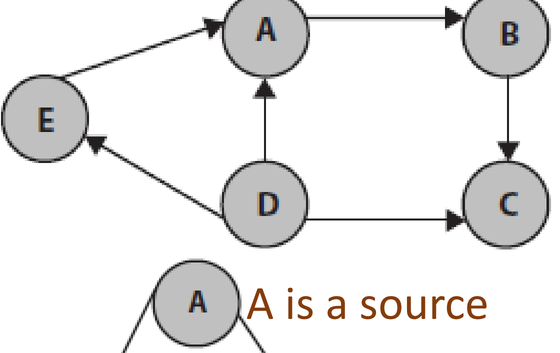


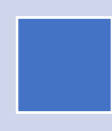


CYCLIC GRAPH A graph that has cycles is called as cyclic graph.



ACYCLIC GRAPH A graph that has no cycle is known as acyclic graph.



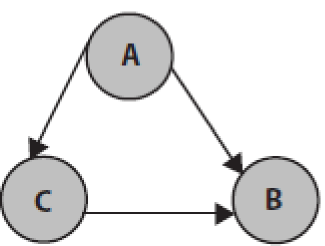


SOURCE A node which has no incoming edges, but has outgoing edges

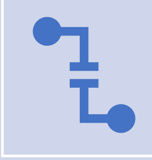
A is a source

Graph Terminologies

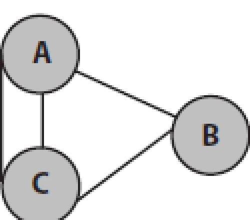
SINK A node, which has no outgoing edges but has incoming edges



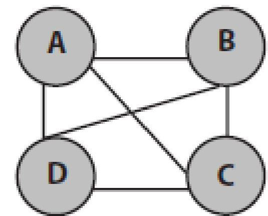
B is a Sink node



DEGREE In an undirected graph the number of edges connected to a node is called the degree of that node. In graph-3 the degree of the node A is 3 and the degree of the node B is 2.



REGULAR GRAPH A graph is regular if every node is adjacent to the same number of nodes



Representation of Graph

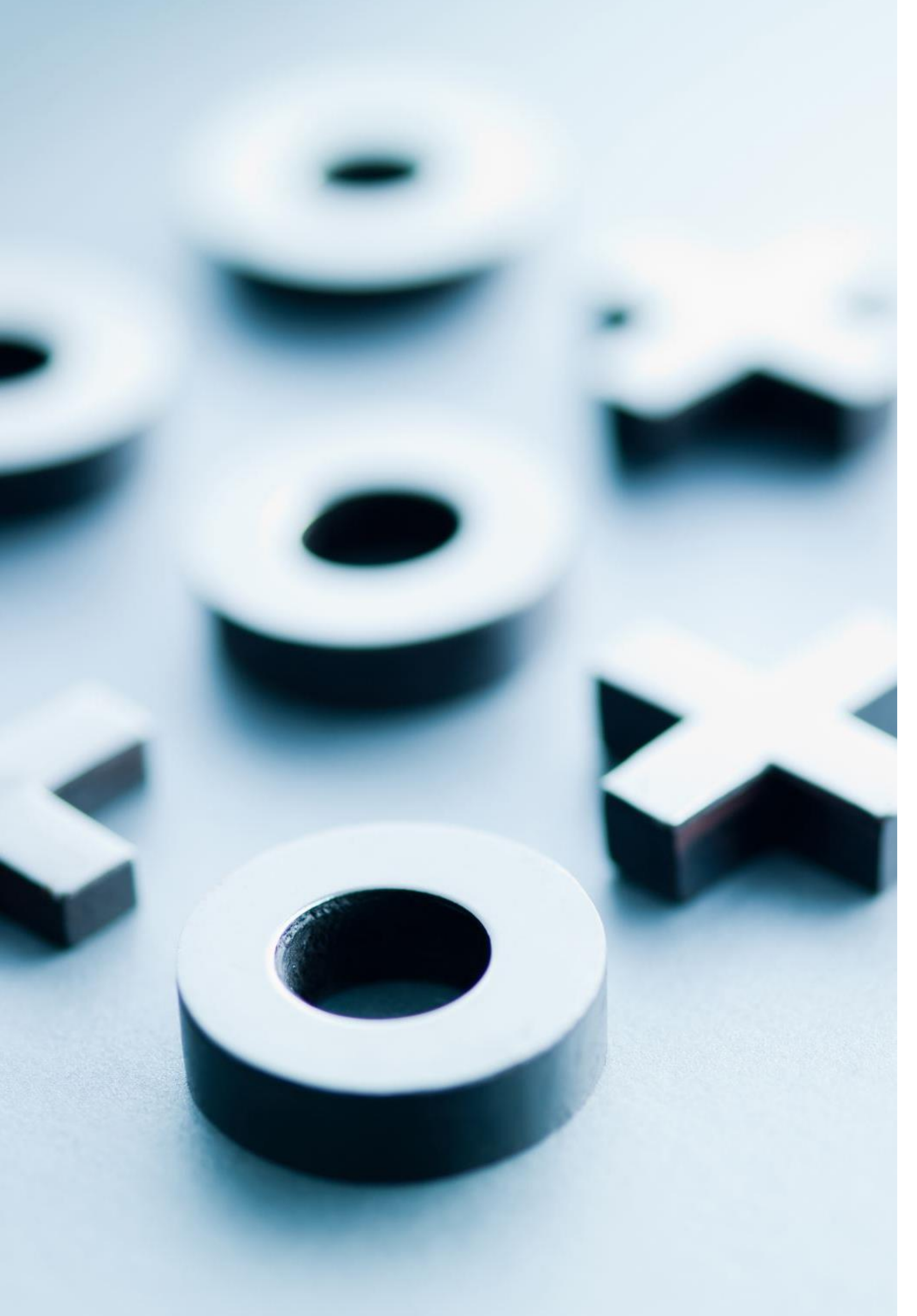
The major components of the graph are node and edges.

Like tree the graph can also be represented in two different ways such as

- ARRAY REPRESENTATION
- LINKED REPRESENTATION

Overall, there are four major approaches to represent the graph as

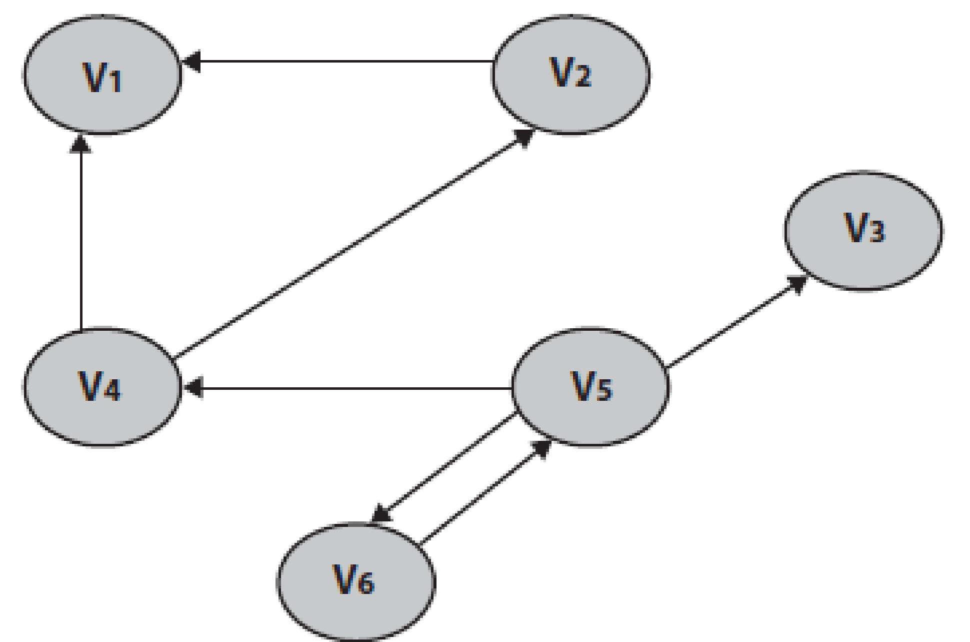
- Adjacency Matrix
- Adjacency Lists
- Adjacency Multilists
- Incedince Matrix



Adjacency Matrix

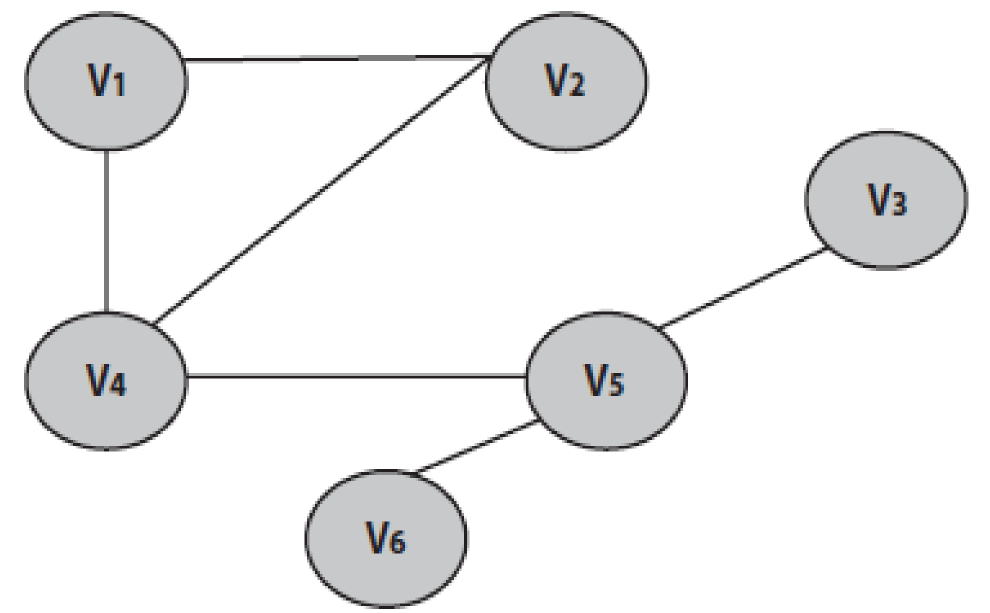
- The nodes that are adjacent to one another are represented as matrix.
- The adjancy matrix of the graph G is a two-dimensional array of size n * n(Where n is the number of vertices in the graph) with the property that A[I][J] = 1, if the edge (VI, VJ) is in the set of edges and A[I][J] = 0 if there is no such edge

Example:



	V1	V2	V3	V4	V5	V6
V1	0	0	0	0	0	0
V2	1	0	0	0	0	0
V3	0	0	0	0	0	0
V4	1	1	0	0	0	0
V5	0	0	1	1	0	1
V6	0	0	0	0	1	0

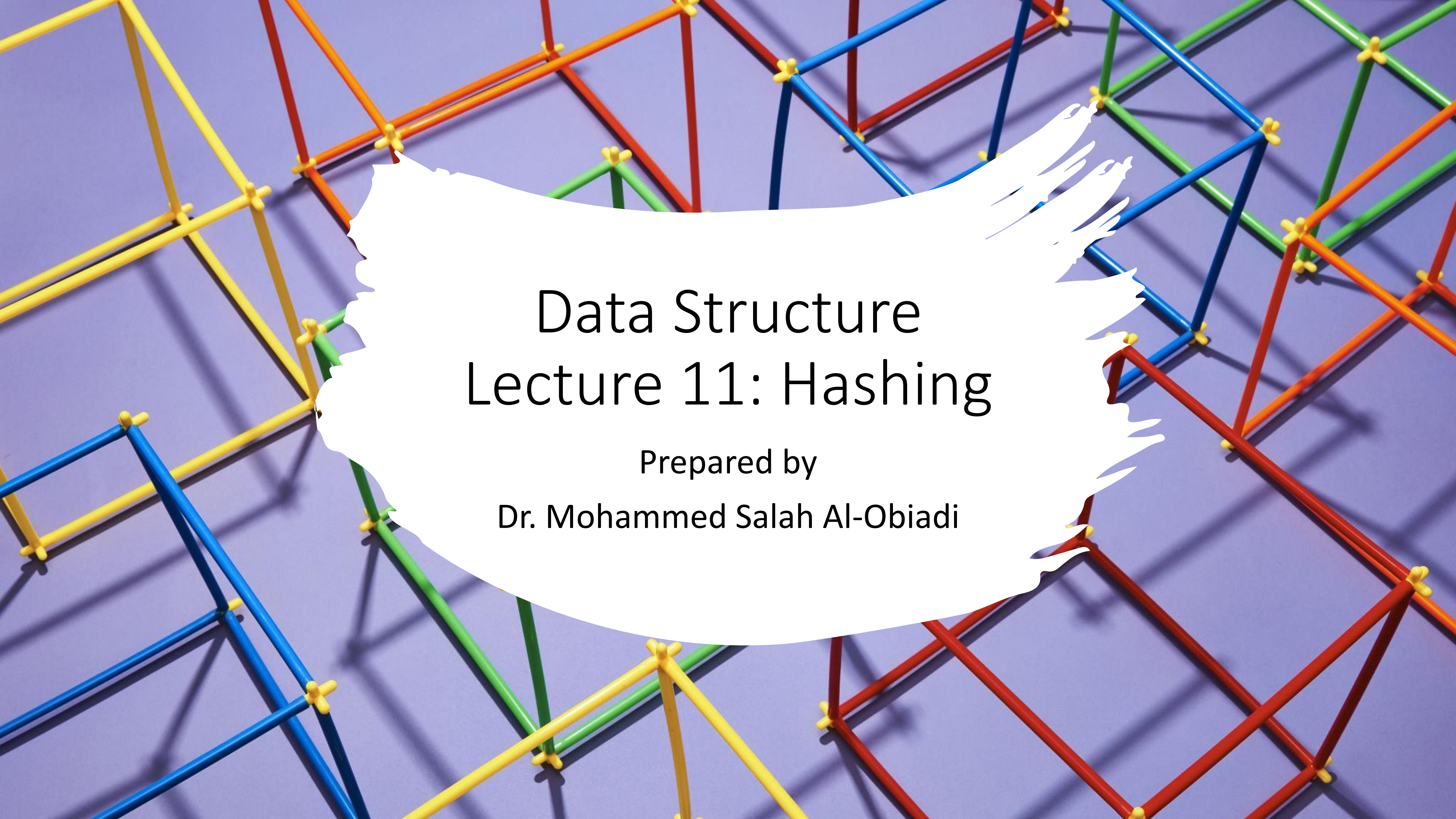
If the graph was Undirected:



	V1	V2	V3	V4	V5	V6
V1	0	1	0	1	0	0
V2	1	0	0	1	0	0
V3	0	0	0	0	1	0
V4	1	1	0	0	1	0
V5	0	0	1	1	0	1
V6	0	0	0	0	1	0

Traversal of Graph

- The Graph Traversal is of two types such as
  - Breadth First Search (BFS).
  - Depth First Search (DFS).

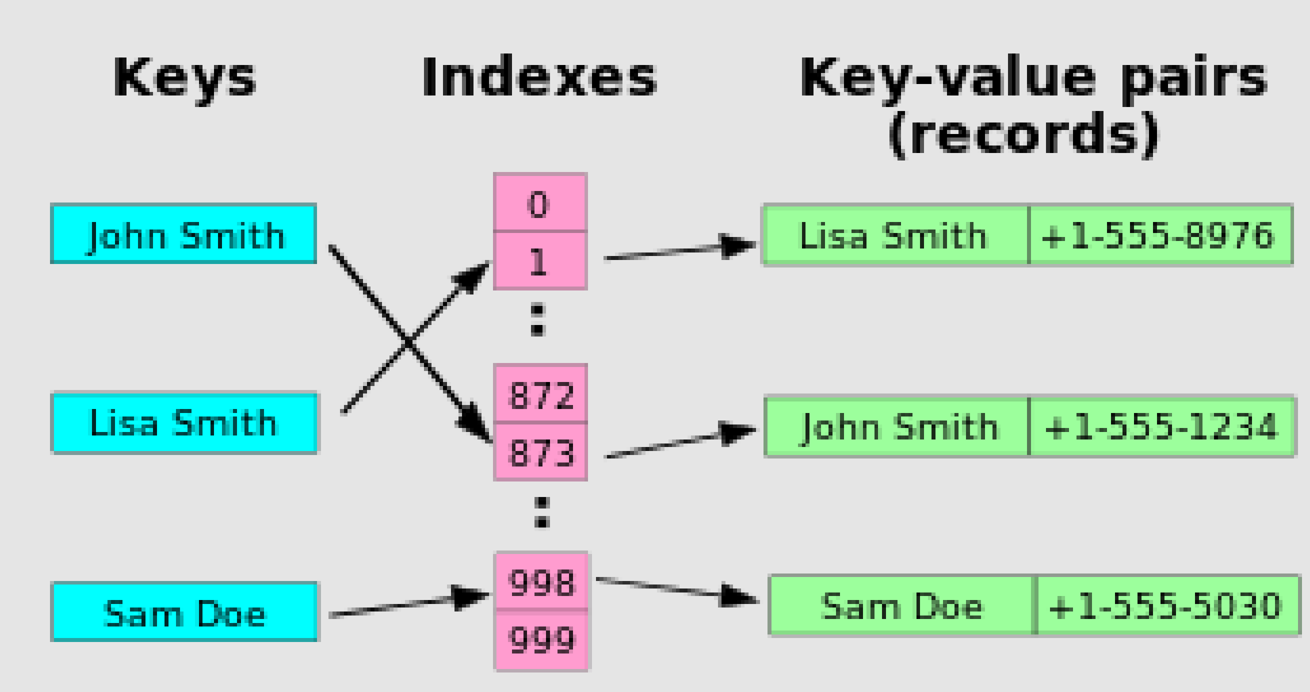


Data Structure
Lecture 11: Hashing

Prepared by

Dr. Mohammed Salah Al-Obiadi

Hash Table



- Hashing is a technique used for storing and retrieving information as quickly as possible.
- It is used to perform optimal searches and is useful in implementing symbol tables.
- Arrays, linked list, trees requires O(N) or log(N) to, search, traverse, add or delete, while with hashtable it requires only O(1) time to do these operations.
- A hash table is a *array* with mapping function.

Why hash table is important

- If we have a set of names Ahmed, Yaser, Yasien, Mina, Mustafa, Lina.
- Let's suppose that you put these names in an array or a linked list.
- Let’s suppose that you are looking for the name Lina.
- Then, you will need to match all the array or linked list elements to find that name because it’s the last name in the list.
- This name costs loop of 6 to find it and if the list of names = n, then it costs O(n) to find it.
- With hash table, you give the name Lina, and you get it immediately in O(1). How is that? We will see in the next slides.

Time Complexity in Big O notation

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

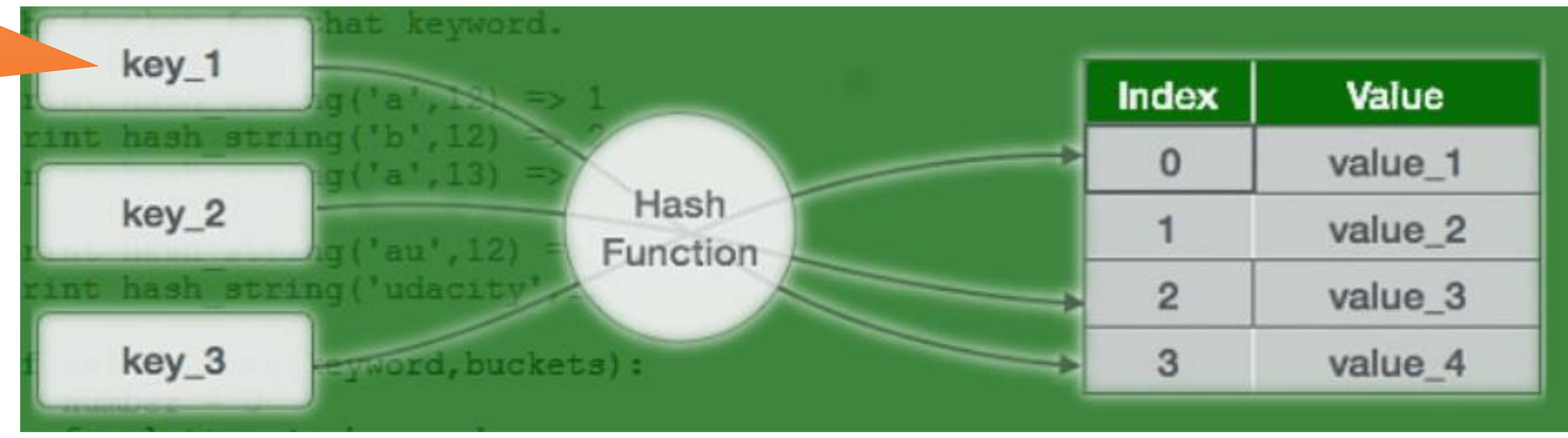
Components of Hashing

- Hashing has four key components:
  1. Hash Table
  2. Hash Functions
  3. Collisions
  4. Collision Resolution Techniques

Hash Table

- Hash table or hash map is a data structure that stores the keys and their associated values.
- Hash table is a kind of array.
- It uses a hash function to map keys to their associated values.
- The terms widely used with hash table are Key, value. See Figure down.

Key here can be number, name, text, picture!!



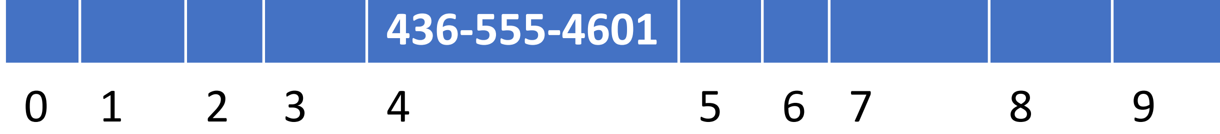
Hash Functions

- The hash function is used to transform the key into the index.
- The hash function should map each possible key to a unique slot index.
- A hash function that maps each item into a unique slot is referred to as a *perfect hash function.*
- A hash function can lead to collisions.
- Collisions happen when a hash function for two keys produce same index.
- Our goal is to create a hash function that minimizes the number of collisions

How to choose your hash function?

- Example of phone number 436-555-4601.
- Consider the hash function h(x) = sum of numbers % hashtable size.
- The % is the mode or division reminder and x is the phone number.
- Suppose hashtable size=10.

- So, h(436-555-4601)= (4+3+6+5+5+5+4+6+0+1) %10 = 4



- Another phone (479-345-6537)
- h(479-345-6537)= (4+7+9+3+4+5+6+5+3+7) %10 = 3

479-345-6537 436-555-4601

0 1 2 3 4 5 6 7 8 9

- h(564-767-6537)= (5+6+4+7+6+7+6+5+3+7) %10 = 6

0	1	2	3	4	5	6	7	8	9
			479-345-6537	436-555-4601		564-767-6537			

- h(565-387-9865)= (5+6+5+3+8+7+9+8+6+5) %10 = 2

0	1	2	3	4	5	6	7	8	9
		565-387-9865	479-345-6537	436-555-4601		564-767-6537			

- h(343-387-9865)= (3+4+3+3+8+7+9+8+6+5) %10 =6, here we have collision because index 6 already full.

- Problems with hashtable:
  1. Number of elements can be larger than hashtable size.
  2. An input that can lead to same index which is called collision

Characteristics of Good Hash Functions

1. Minimize collision
2. Be easy and quick to compute
3. Distribute key values evenly in the hash table
4. Use all the information provided in the key

Collisions

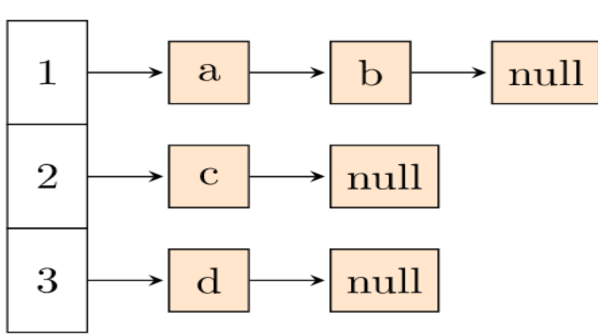
Hash functions are used to map each key to a different address space

But it is not possible to create such a hash function and the problem is called *collision.*

Collision is the condition where two records are stored in the same location

Collision Resolution Techniques

- Collision resolution is The process of finding an alternate location.
- There are a number of collision resolution techniques:
  - **Direct Chaining:** An array of linked list application
    - Separate chaining



  - **Open Addressing:** Array-based implementation
    - Linear probing (linear search)
    - Quadratic probing (*n*onlinear search)
    - Double hashing (use two hash functions)

Linear probing

The interval between probes is fixed at 1.

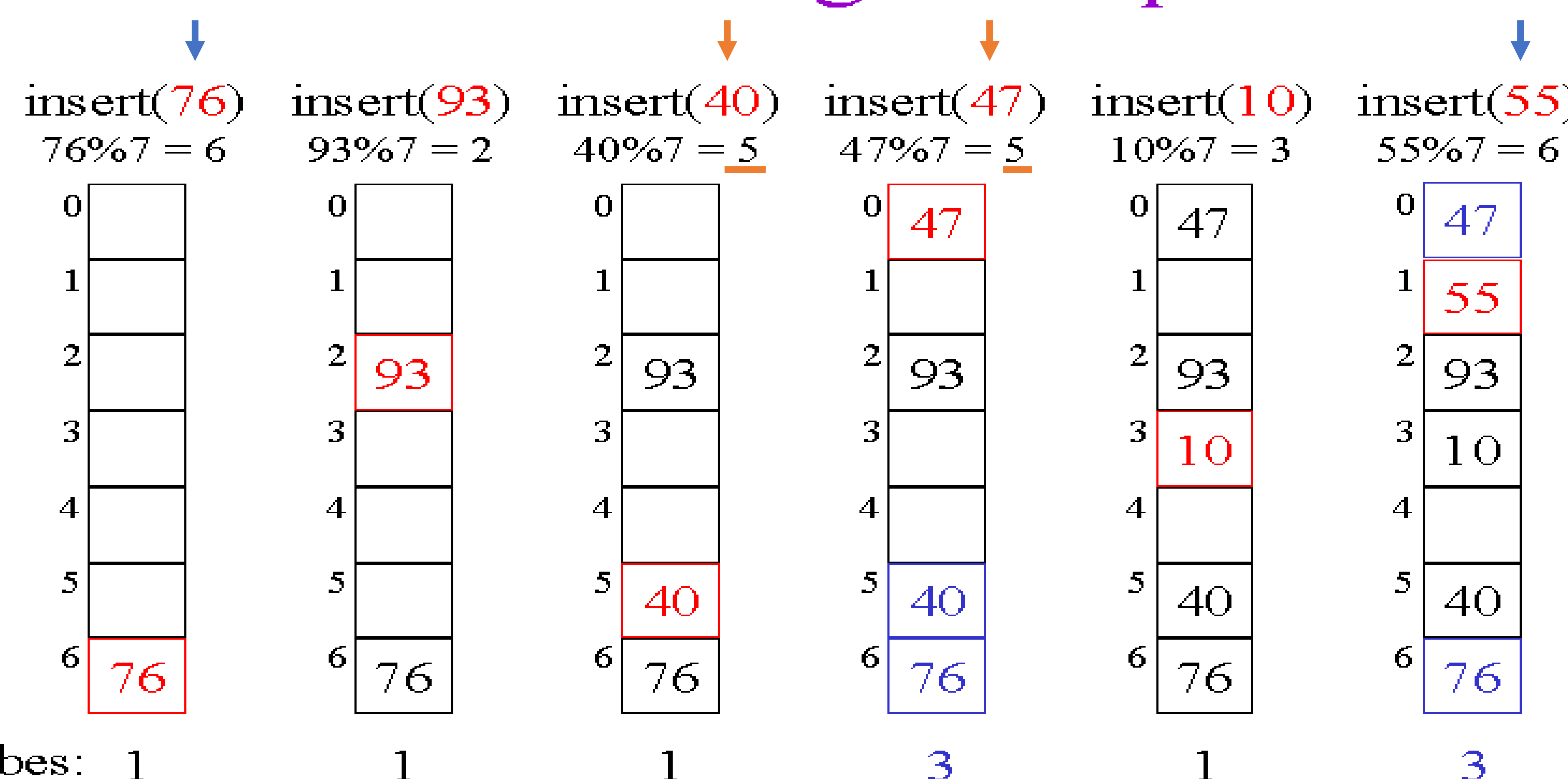
In linear probing, we search the hash table sequentially, starting from the original hash location.

If a location is occupied, we check the next location.

$$rehash(key) = (n + 1)\% \; tablesize$$

Where n is the current location found for key

Linear Probing Example



Quadratic Probing

Linear probing can lead to cluster of keys together.

The problem of Clustering can be eliminated if we use the quadratic probing method

In quadratic probing, we start from the original hash location n.

If a location is occupied, we check the next location.

rehash(key) = (n + k²)% tablesize

$$rehash(key) = (n + k^2) \% \text{ tablesize}$$

Where n is the current location found for key

If a location is occupied, we check the locations $i + 1^2$, $i + 2^2$, $i + 3^2$, $i + 4^2$

Example of Quadratic Probing

- Let us assume that the table size is 11 (0..10)
- Insert keys
  - $31 \text{ } mod \text{ } 11 = 9$
  - $19 \text{ } mod \text{ } 11 = 8$
  - $2 \text{ } mod \text{ } 11 = 2$
  - $13 \text{ } mod \text{ } 11 = 2 \rightarrow 2 + 1^2 \text{ mod } 11 = 3$
  - $25 \text{ } mod \text{ } 11 = 3 \rightarrow 3 + 1^2 \text{ mod } 11 = 4$
  - $24 \text{ } mod \text{ } 11 = 2 \rightarrow 2 + 1^2 \text{ mod } 11 \text{ , } 2 + 2^2 = 6$
  - $21 \text{ } mod \text{ } 11 = 10$
  - $9 \text{ } mod \text{ } 11 = 9 \rightarrow 9 + 1^2, 9 + 2^2 \text{ } mod \text{ } 11, 9 + 3^2 \text{ } mod \text{ } 11 = 7$

Index	Value
0	
1	
2	2
3	13
4	25
5	5
6	24
7	9
8	19
9	31
10	21