Data Structures

Outline

- What is an algorithm?
- Introduction to data structure
- Why data structure?
- Types of Data structures.
- Arrays.
- Representation of arrays in a memory.

What is an algorithm?

- An algorithm is a finite set of instructions that accomplishes a particular task.
- A well-defined computational procedure that takes some value, or a set of values, as input and produces some values or a set of values as output.
- Sequence of computational steps that transform the input into the output

What is a Good Algorithm ?

- Efficient :
  - Running Time.
  - Space Used.

Introduction to data structure

- Data are simply a value or a set of values of different types such as:
  - String
  - Integer
  - Char
  - etc.
- Structure is a way of organizing information.

Introduction to data structure

- In simple words we can define a data structure as:
  - A way of organizing data so that it can be used effectively.
  - Organizing the data in some way so that later on it can be quickly and easily:
    - Accessed
    - Queried
    - And Updated

Why data structure?

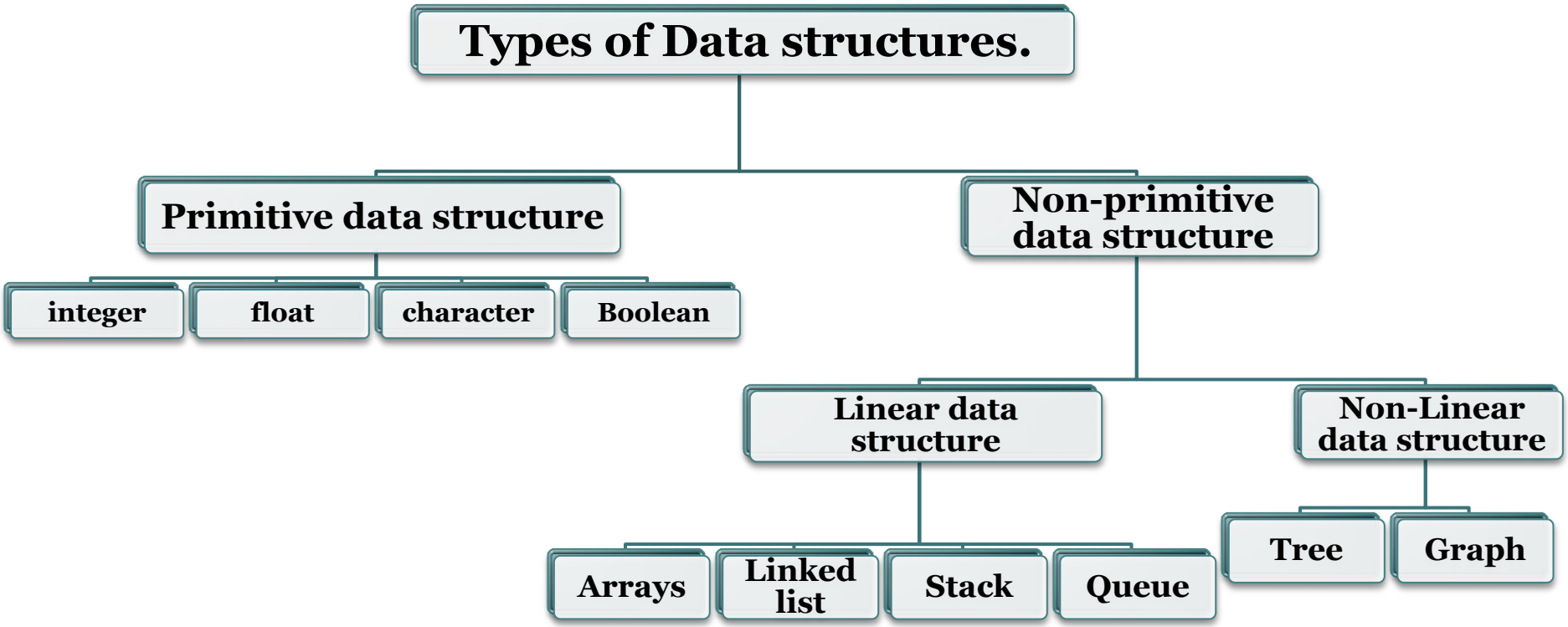
- Data structures are:
  - Essential ingredient in creating fast and powerful algorithms.
  - Help to manage and organize the data.

Advantages of organized data

- The organized data has many advantages:
  - Easy data retrieval.
  - Less data storage space.
  - Easy data operations, modification and deletion.
  - Easy to manage.
  - Avoid duplicate data.
  - Efficient software systems.

How to choose suitable data structure

- For each set of data there are different methods to organize these data in a particular data structure.
- To choose suitable data structure, we must use the following criteria:
  - Data size and the required memory
  - The dynamic nature of the data.
  - The required time to obtain any data element from the data structure.


Types of Data structures.
Primitive data structure
Non-primitive data structure
integer
float
character
Boolean
Linear data structure
Non-Linear data structure
Arrays
Linked list
Stack
Queue
Tree
Graph

Primitive Data Structure

- Some data structures are built in to the programming language itself which are readily available to the programmer for its use and these data structure are referred as primitive data structure.
- For example, the most common primitive data structures which are generally supported by almost all programming languages include
  - Integer.
  - Float.
  - Character.
  - and Boolean.

Primitive data structure

Data structure	Description	Example
Integer	Represents a number without decimal points	1, 2, 45, 1000
Float	Represents a number with decimal points	1.5, 4.5, 3.8, 2567.24
Character	Represents a single character	A, B, g, h
Boolean	Represent logical values either True or False	

Non-Primitive data structure

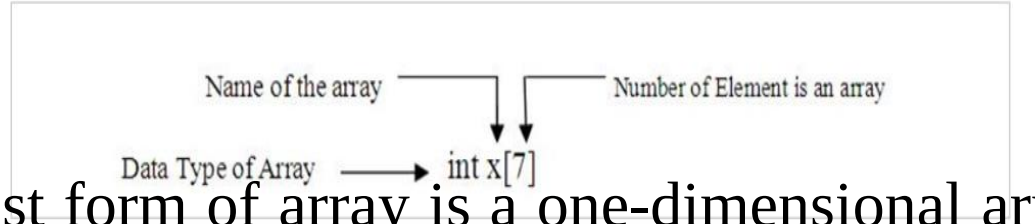
- The Non-Primitive data structure are user (programmer) defined data structure which also referred as user defined data structure.
- The Non-primitive data types are not pre-defined in the programming language, and therefor the programmer has to defined as per the program requirements.
- The Non-primitive data structure are derived from primitive data types by combining two or more primitive data structure. These Data structure divided as linear and Non-linear data structure.

Linear & Non-Linear Data structure

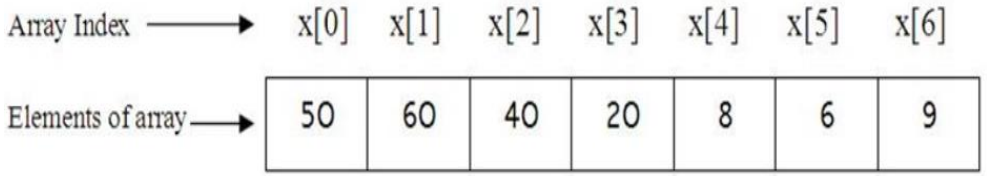
- In **Linear** data structures, the data items are arranged in memory in a linear sequential. Example **Array**, **Stack**
- In **Non-Linear** data structures, the data items are not stored in memory in sequential. Example **Tree**, graph

Arrays

- An array is a linear data structure, it is collection of elements of same data type. An array is linear data structure because the array elements (Values) are traversed in sequential manner.



- The simplest form of array is a one-dimensional array. It is organized in a single row consisting of number of pre-defined number of data elements



Two Dimensional Array

1	5	3	6
3	2	38	64
22	76	82	99
0	106	345	54

User's view (abstraction)

1	5	3	6	3	2	38	64	22	76	82	99	0	106	345	54

System's view
(implementation)

Representation of array in memory

- **One Dimensional array**
  - Location(X[I])=Base Address+(I-1)
  - **Example:**

Find X[4] location in memory where Base Address=500

Location(X[I])=Base Address+(I-1)

Location(X[4])=500 +(4 -1)

=503

Representation of array in memory

- **Two Dimensional array**
- Row –wise method:

Location (A[i , j]) = base address + N * (i-1) + (j-1)

- Column – wise method

Location (A[I,J]) = Base Address + M * (J -1) + (I -1)

- **Example:**

A[5,7], M=5, N=7, Base Address=900 Find A[4 , 6]

Representation of array in memory

By **Row** :

Location(A [4,6])= 900+7 *(4-1)+(6-1) = 900+21+5 =926

By **Column:**

Location(A [4,6])=900 +5 *(6 -1)+(4 -1) =900+25+3 =928

Thank you

DATA STRUCTURES LECTURE #2

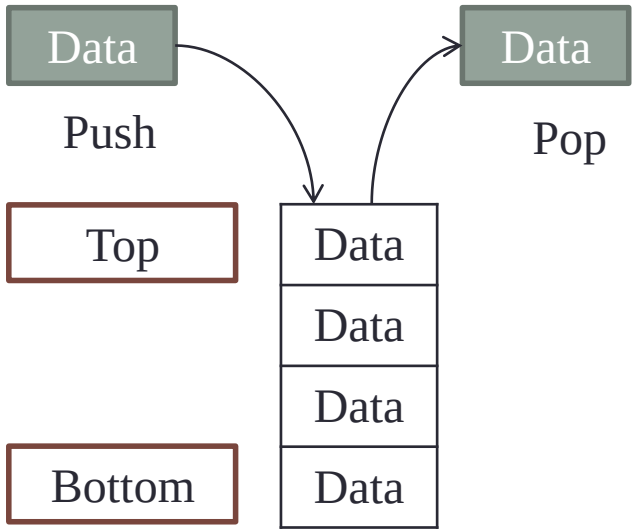
Lecturer
Nadia Kubba
nadia.mohsin@uokufa.edu.iq

Outline

- **Stack**
- **Stack Operations**
- **Arrays to implement stacks**
- **Stack Applications**

Stack

- Stack is a linear data structure which follows a particular order in which the operations are performed. The order is LIFO(Last In First Out)
- Objects are inserted into a stack at any time, but only the most recently inserted object (last one!) can be removed at any time
- New elements can be added and removed only at the top.



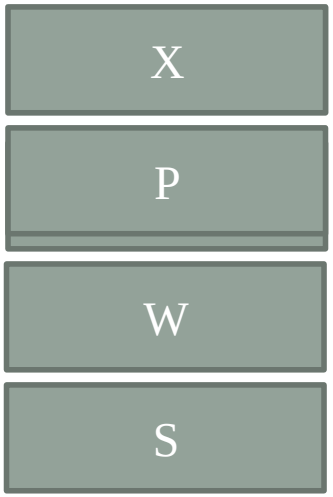
Stack Operations

- The fundamental operations involved in a stack are "**push**" and "**pop**".
  - ***push***: adds a new element on the stack
  - ***pop***: removes an element from the stack

- Checking Conditions
  - ***top>=StackSize-1 :*** Stack is full.. "Stack OVERFLOW"
  - ***top== -1 or zero***: Stack is empty.."Stack UNDERFLOW"

Example

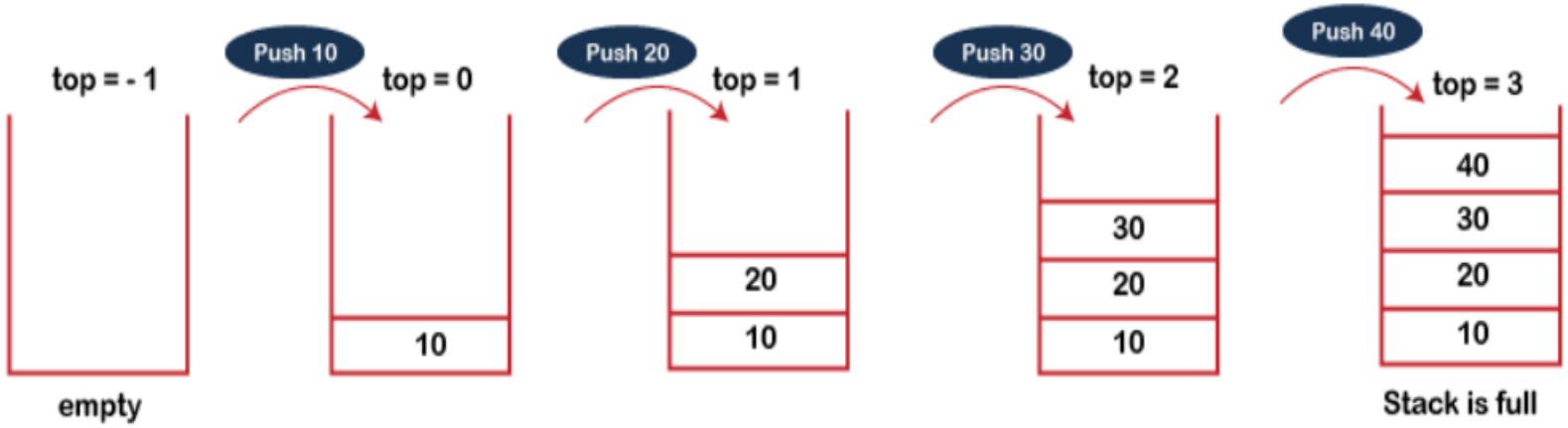
- Instruction
- Pop()
- Push (A)
- Push (D)
- Push (F)
- POP ()
- POP()
- PUSH(X)



PUSH operation

The steps involved in the PUSH operation is given below:

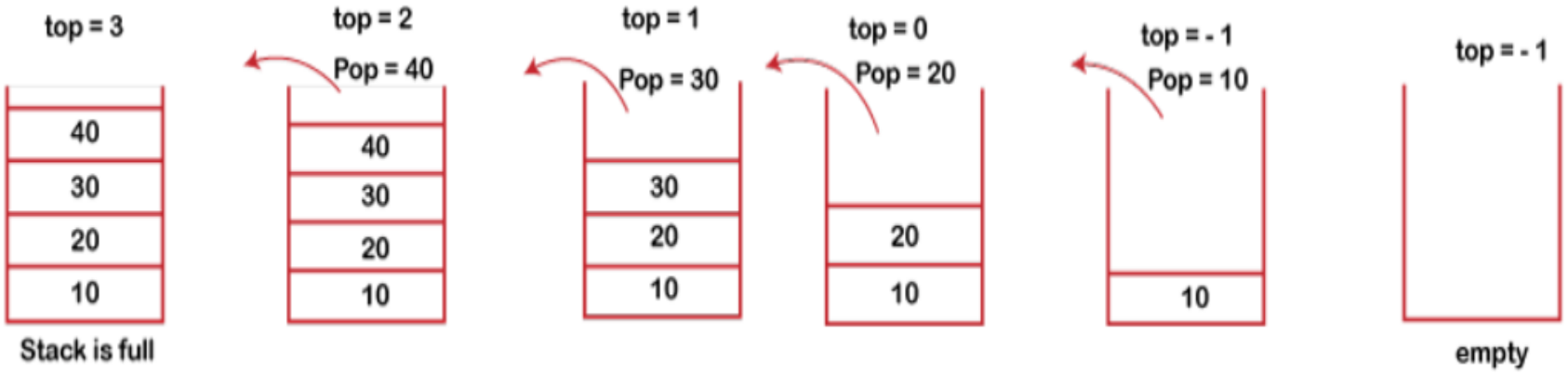
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1**, and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.



POP operation

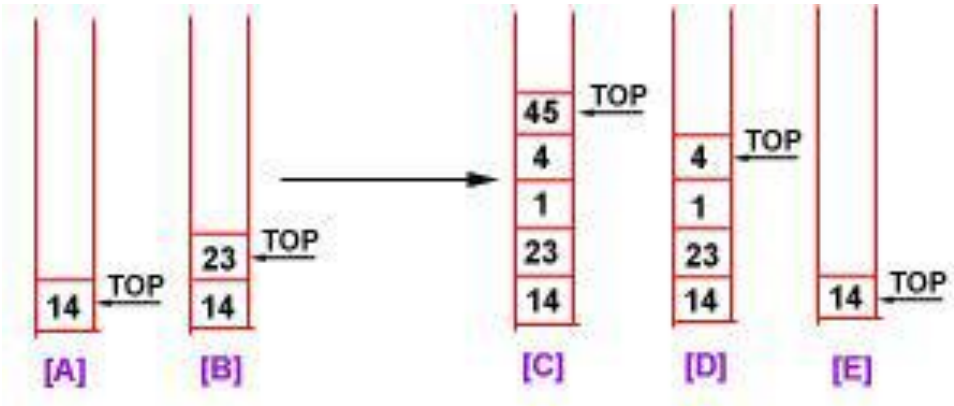
The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Operation on a stack

- Example:
- Let us consider a stack with integers 14, 23, 1, 4, 45 and upper limit set to 5.
- Note: Every time you Push() an element inside the Stack, the TOP of the Stack gets incremented by 1 and vice versa.
- When the STACK is empty, the BOTTOM and TOP of the STACK are same and pointing to the empty STACK.
- If you try to PUSH more elements than the upper limit of the STACK, it will cause in an overflow of data and vice-versa.



- Initially, the STACK is empty and the TOP of the STACK is pointing to the BOTTOM of the STACK.
- Stage [A]: We added 14 and TOP now points to it.
- Stage [B]: 23 is PUSHed and TOP is incremented by 1.
- Stage [C]: The STACK is FULL, as the upper limit was set to 5.
- Stage [D]: The TOP most element has been POPed. The TOP gets decremented by 1.
- Stage [E]: 45, 4, 1 and 23 have been POPed and TOP is now pointing to the bottom most element

Stack functions

- To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks

- **peek()** : get the top data element of the stack, without removing it.

- **isFull()** : check if stack is full.

- **isEmpty()** : check if stack is empty.

Pushing elements on the stack

- ***Push*** operation adds a new element on the stack.

Algorithm

1- Start

2- Let stack [size]

3- let top=-1

4- if top<stack.size then

5- top=top+1

6- stack[top]=data

7- else

8- Print (“ Stack Is Full “, “Data Over flow”)

9- End

Popping Elements From Stack

- ***Pop*** operation removes a element from the stack.

Algorithm

1- Start

2- If Top>=0 then

3- print stack[top]

4- top=top-1

5- else

6- Print (“ Stack Is Empty , Data Under Flow “)

7- end

Stack Applications

- Web browser: stores the addresses of recently visited sites on a stack. Each time a user visits a new site, the address of the site is *pushed* into the stack of addresses. Use the back button the user can *pop* pack to previously visited sites.
- Text editors: powerful text editors keep text changes in a stack. The user can use the undo mechanism to cancel recent editing operations.
- Arithmetic expression: following to next slide

Implementation of Stack

- There are two ways to implement a stack:

1. Using array
2. Using linked list

Using arrays to implement stacks

- A natural way of implementing a stack is with an array.
- Top represents the index in the array at which the next item pushed onto the stack is to be stored. This increases by one when another item is pushed onto the stack, and decreases by one when an item is popped.

- The stack is empty when the top =-1, and it is full if the top pointer is greater than the maximum size of the stack array. This is because top now lies beyond the bounds of the array.

Thank you

DATA STRUCTURE
LECTURE #3

Lecturer
Nadia A Kubba
nadia.mohsin@uokufa.edu.iq

OUTLINE

- Arithmetic expressions
- Precedence ,primacy, priority
- Prefix, Postfix & Infix Notation
- Algorithm to convert infix to postfix
- Examples

Arithmetic expressions

- An arithmetic expression is one which is evaluated by performing a sequence of arithmetic operations to obtain a numeric value.
- Levels of Precedence for the usual five binary operations on arithmetic operation Q .
  - Highest: Exponentiation ^
  - Next Highest: Multiplication * and division /
  - Lowest: Addition + and subtraction -

Precedence ,primacy, priority

Operator(s)	Precedence & Associativity
( )	Evaluated first. If **nested (embedded)**, innermost first. If on same level, left to right.
* / %	Evaluated second. If there are several, evaluated left to right
+ -	Evaluated third. If there are several, evaluated left to right.
=	Evaluated last, right to left.

Prefix, Postfix & Infix Notation

- Prefix, Postfix & Infix Notation
- Infix : the operator is placed between operands.
  - Example: (A+B)*C parentheses necessary
- Prefix: the operator is placed before the operands
  - Example: *+ABC
- Postfix (Reveres polish notation): the operator is placed after the operands .
  - Example: AB+C*

Why?

- Why to use PREFIX and POSTFIX notations when we have simple INFIX notation?
- INFIX notations are not as simple as they seem especially while evaluating them. To evaluate an infix expression we need to consider Operators' Priority and Associative property
  - Example:
  - expression 3+5*4 evaluate to 32 i.e. (3+5)*4 or to 23 i.e. 3+(5*4).
- To solve this problem Precedence or Priority of the operators were defined. Operator precedence governs evaluation order. An operator with higher precedence is applied before an operator with lower precedence.

Algorithm to convert infix to postfix

- A summary of the rules follows:

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.

5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Rules

- Four rules you should know to convert from infix to postfix

1. Priority of operands
2. No two operands of same priority can stay in a stack
3. Lowest priority can not be placed after highest priority
4. If the symbol is a close parenthesis then the operands in between should be popped out the stack.

Example 1: (A+B)/D

Symbol	Stack	Postfix
(	(	
A	(	A
+	(+	A
B	(+	AB
)	(+)	AB+
/	/	AB+
D	/	AB+D
		AB+D/

Example 2: (A+B)/(D+E)

Symbol	Stack	Postfix
(	(	
A	(	A
+	(+	A
B	(+	AB
)	(+)	AB+
/	/	AB+
(	/(	AB+
D	/(	AB+D
+	/(+	AB+D
E	/(+	AB+DE
)	/(+)	AB+DE+
		AB+DE+/

Example 3: A-B/(C*D^E)

Symbol	Stack	Postfix
A		A
-	-	A
B	-	AB
/	-/	AB
(	-/(	AB
C	-/(	ABC
*	-/(*	ABC
D	-/(*	ABCD
^	-/(*^	ABCD
E	-/(*^	ABCDE
)	-/(*^)	ABCDE^*/-

Examples of infix to prefix and postfix

Infix	Prefix	Postfix
(A+B)/D	/+ABD	AB+D/
(A+B)/(D+E)	/+AB+DE	AB+DE+/
(A-B/C+E)/(A+B)	/+-A/BCE+AB	ABC/-E+AB+/
B^2-4*A*C	-^B2**4AC	B2^4A*C*-
A-B/(C*D^E)	-A/B*C^DE	ABCDE^*/-

Example: postfix expressions

- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.
  - infix: 2+5 postfix: 2 5 +
- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never nee

Infix	Postfix	evaluation
2 - 3 * 4 + 5	234*-5+	-5
(2 - 3) * (4 + 5)	23-45+*	-9
2- (3 * 4 +5)	234*5+-	-15

Converting between notations

- **1. Infix to Prefix:**
- (A+B)-(C*D)
- - Do the first brace: (A+B) ,the prefix is +AB
- - Do the second brace : (C*D) , the prefix is *CD
- - The end is operator-: +AB - *CD
- The prefix is: - + A B * C D

Converting between notations

- **2. Infix to Postfix:**
- (A+B)-(C*D)
- - Do the first brace: (A+B) , the Postfix is AB+
- - Do the second brace : (C*D), the Postfix is CD*
- - The end is operator-: AB+ - CD*
- The postfix is A B + C D * -

Converting between notations

- **3. Prefix to Infix:**
- **+ / *A B C D**
- **-** Find the first operator: *, take 2 operands before the operator (A and B), the Infix is (A * B)
- **-** Find the second operator : / , take 2 operands before the operator (A*B and C), the Infix is ((A*B)/C)
- **-** find the third operator-: + , take 2 operands before the operator (((A*B)/C) and D), the Infix is ((A*B)/C)+D)

Thank you

DATA STRUCTURES LECTURE #4

Lecturer
Nadia A. Kubba
nadia.mohsin@uokufa.edu.iq

OUTLINE

- What is a queue
- Queue operations
- Applications of Queue
- Types Of queue
- Circular Queue
- Double ended queue
- Priority queue

What is a queue

- Is a linear data structure.
- Contains elements that are inserted and removed according to the first-in-first-out (FIFO) principle .
- The new elements are added at the end (“the rear”) and elements are removed from the other end (“the front”).

Queues Operations

- It supports two fundamental methods:
  - Enqueue: inserts an element at the end of the queue
  - Dequeue: removes an element from the front of the queue

- Checking Conditions:
  - Queue Overflow :If rear = maxsize-1
  - Queue Empty: If front = -1

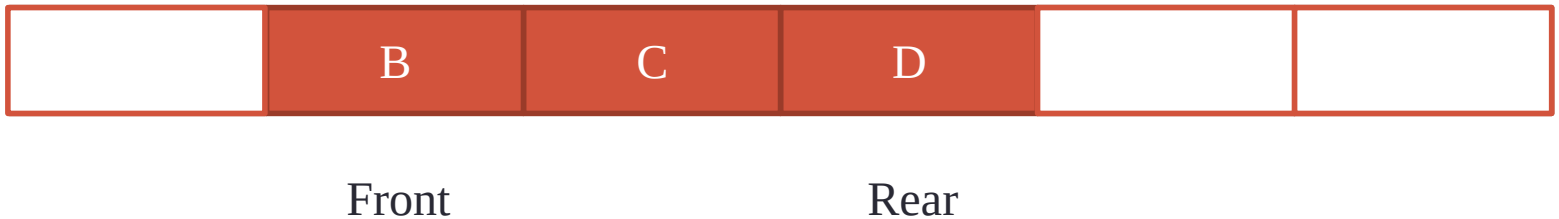
Operations on Queues … Cont

- The queue has two pointers:
  - FRONT: containing the location of the front element.
  - REAR: containing the location of the rear element.
- The new element will add itself at the 'rear' end, then Queue's **‘REAR' value increments by one.**
- The element leaves the queue from the 'front' end, so the Queue's **‘FRONT' value increments by one.**

Using Arrays to Represent Queues

Enqueue (D)

Dequeue ()



Algorithm for inserting an element into a queue

Step1: Start

Step2: let queue[Size]

Step3: let front=-1 , Rear=-1 // Initialization Queue

Step4: If (rear = queue . Size -1) Then

print (Queue Is Full“Data Data Over flow “)

Else

rear=rear+1

queue[rear]=item

Step5: If front = -1 Then

Front= 0

Step6: End

Algorithm for deleting an element from a queue

Step1: Start

Step 2: IF (front = -1) then

Print (“ Queue Is Empty “ , “ Data Underflow “)

ELSE

print(queue[front])

Step 3: IF (front=rear) then

front = -1 , rear=-1

Else

front =front+1

Step 4:End

Applications of Queue

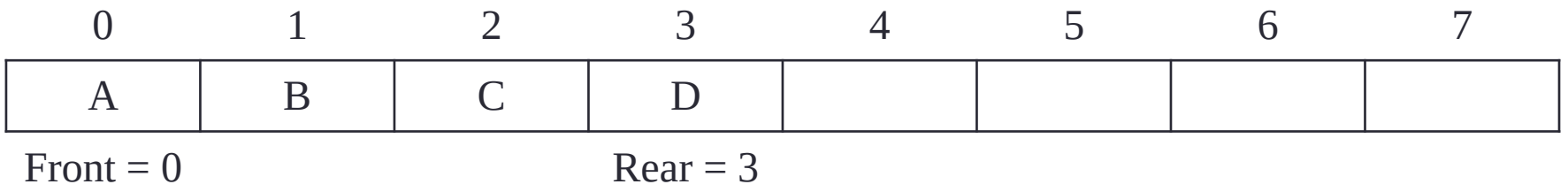
- Used in scheduling the jobs to be processed by the processor.
- A queue schedules the order of the print files to be printed.
- A server maintains a queue of the client requests to be processed

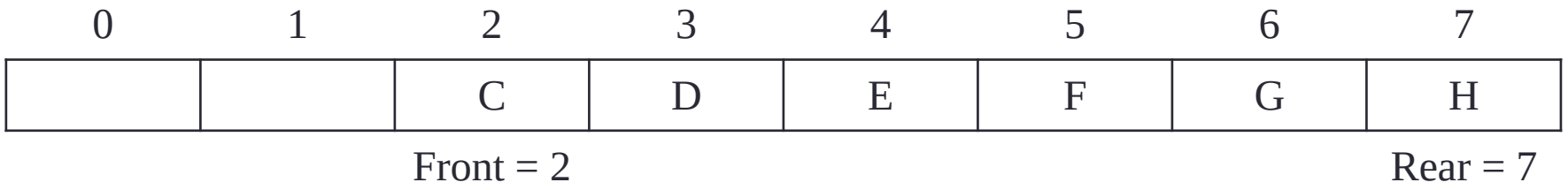
Queue Types

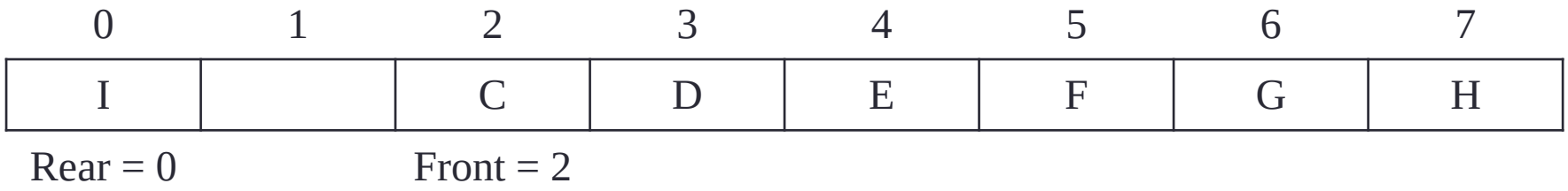
- **Linear Queue**
- **Circular Queue**
- **Double ended Queue**
- **Priority Queue**

Circular Queue

- A circular queue is a Queue but with a particular implementation of a queue.
  - They have a circular structure.
  - There is no space lost.
- The properties of this type of queues is :-
  - Front pointing to the first item.
  - Rear pointing to the last item.
  - when Rear arrives to the end of the queue make it wrap to the beginning of queue (rear =0) , also this with front.


0 1 2 3 4 5 6 7
A B C D
Front = 0
Rear = 3

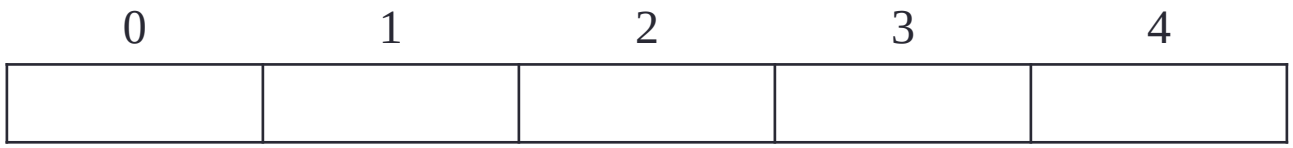

0 1 2 3 4 5 6 7
C D E F G H
Front = 2
Rear = 7


0 1 2 3 4 5 6 7
I C D E F G H
Rear = 0
Front = 2

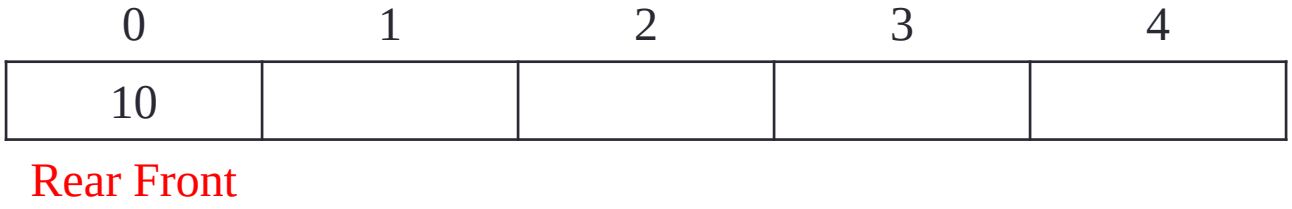
Example

- Consider the following circular queue with size 5

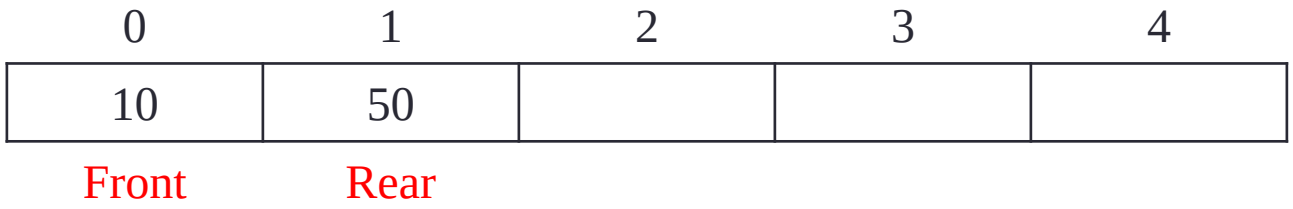
1- Initially Rear =-1 Front = -1



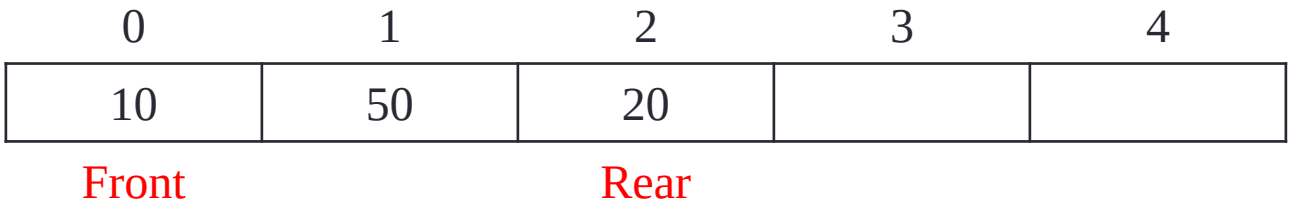
2- Insert 10, Rear = 1 Front =1



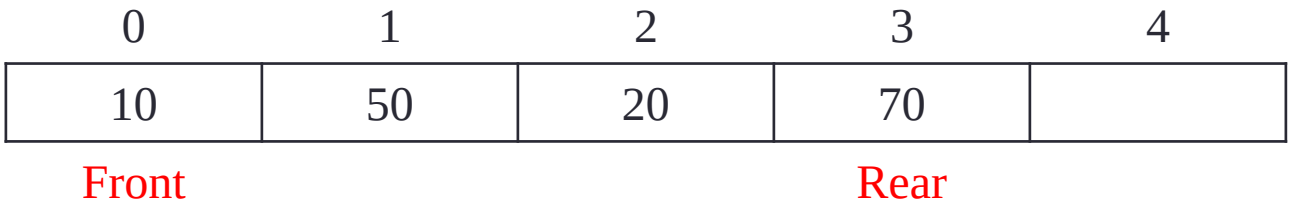
3- Insert 50 Rear = 1 Front =0



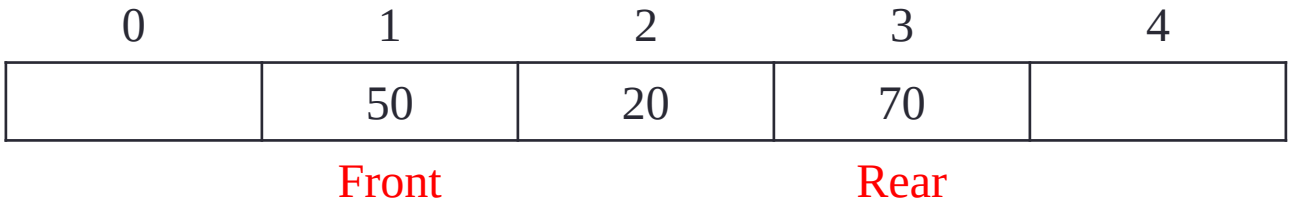
4- Insert 20, Rear = 2, Front =0



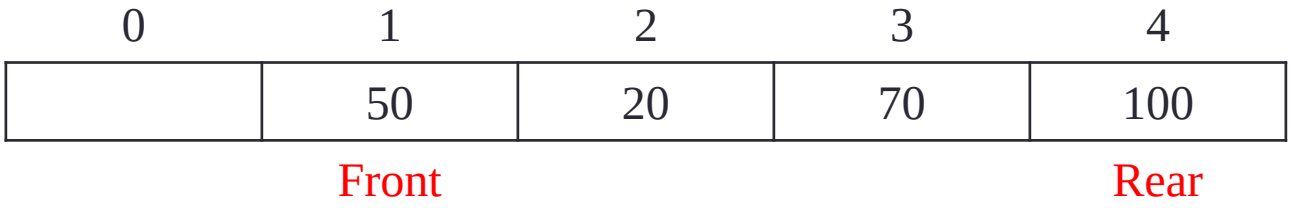
5- Insert 70, Rear = 3, Front =0



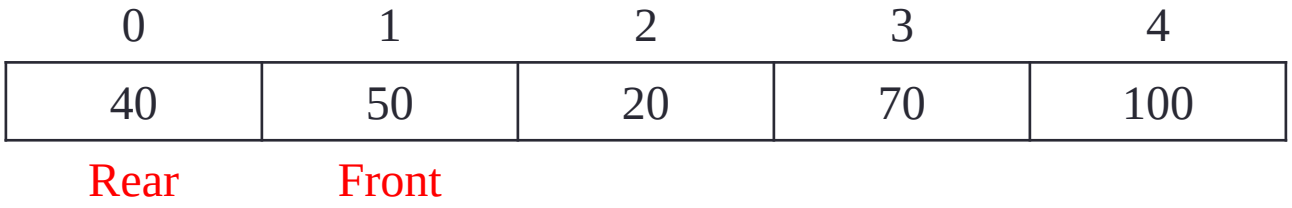
6- Delete front, Rear = 3, Front =1



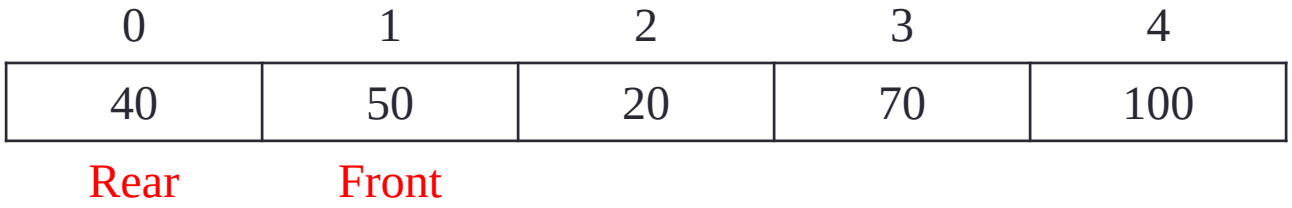
7- Insert 100, Rear = 4, Front =1



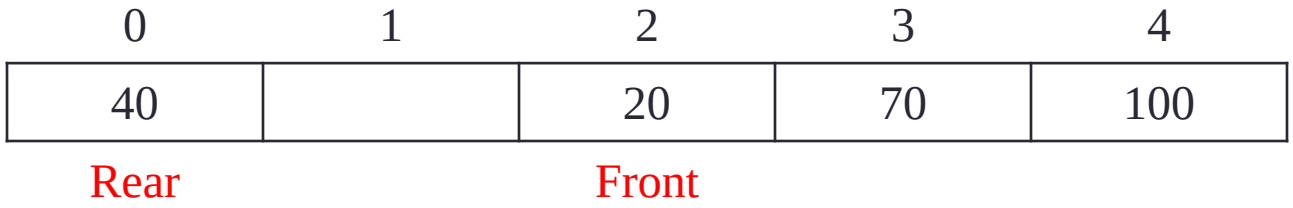
8- Insert 40, Rear =0, Front =1



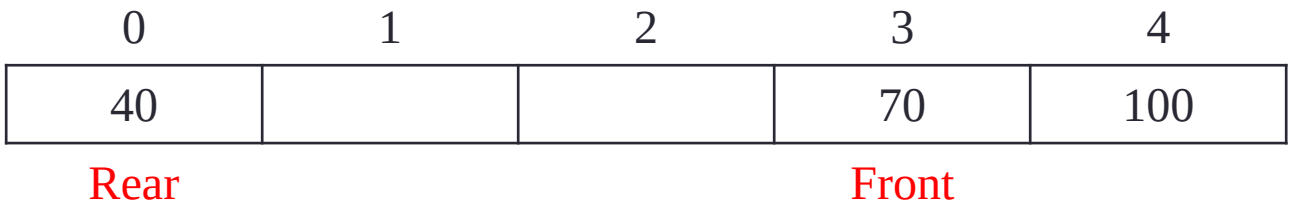
9- Insert 140, Rear = 0, Front =1, As Front = Rear – 1 so the queue is full (overflow)



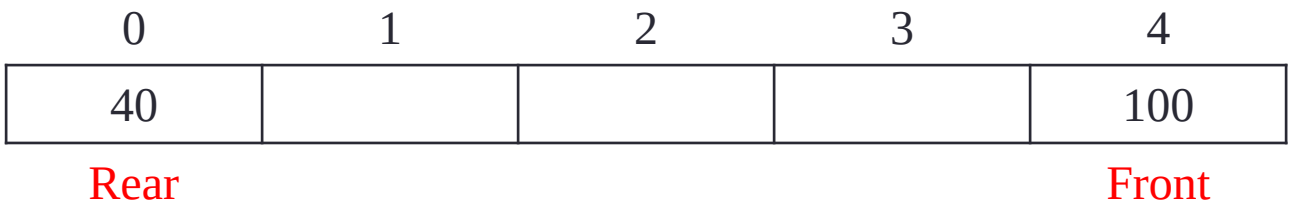
10- Delete front, Rear = 0, Front = 2



11- Delete front, Rear = 0, Front =3



12- Delete front, Rear = 0, Front=4



Algorithm to insert an element in a circular queue

Step1: Start

Step2: p = (Rear+1) MOD CQueue.length;

Step3: If (p = Front) Then

print(“ Queue Overflow ");

else

Rear = p;

CQueue[Rear] = Item;

Step4: if (Front = -1)

Front = 0;

Step5: End

Algorithm to delete an element from a circular

Step1: Start

Step2: If(Front =-1)

print ("CQUEUE IS EMPTY")

Else

Print(CQueue[Front])

Step3: If (Front == Rear)

Front = Rear = -1

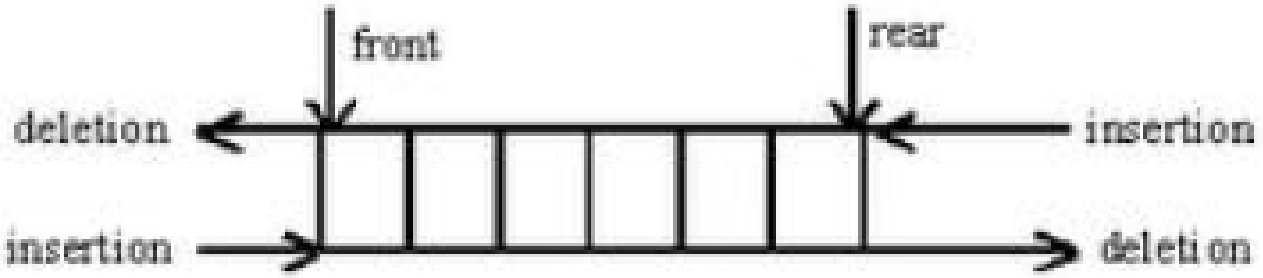
Else

Front=(Front+1) Mod CQueue.length

Step4: END

Double Ended queue

- It is a linear list in which elements are added or removed at either end but not in middle.
- A double-ended queue is a data structure that supports the following operations:
  - enq_front
  - enq_back
  - deq_front
  - deq_back



Double Ended queue

- Double Ended Queue can be represented in two ways:

1) Input restricted De-Queue :- allows insertions at only one end but allows deletions on both ends of the list .

2) Output-restricted de-queue:- allows deletions at only one end but allows insertions at both ends of the list.

Priority queue

- A priority queue is a collection of elements such that each element has been assigned a priority and the order in which elements are deleted and processed comes from the following
- rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

DATA STRUCTURES LECTURE #5

Lecturer
Nadia A. Kubba
nadia.mohsin@uokufa.edu.iq

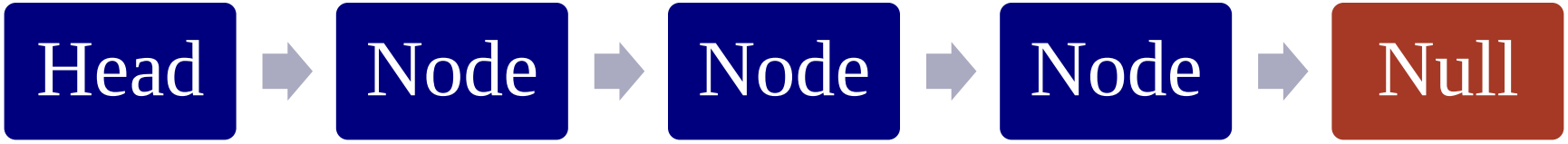
OUTLINE

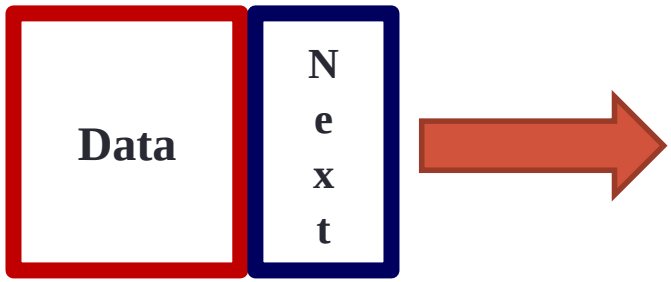
- What is a linked list
- Parts of a Linked List
- Architecture of Linked list
- Applications of linked list in computer science
- Advantages and disadvantages of Linked lists
- Array vs Linked List
- Operations On Linked List
- Types of linked list
- Singly linked list

What is a linked list

- It is another type of data structure which are dynamically allocated.
- It is a collection of especially designed data elements called **nodes** linked to one another by means of **pointers**.
- Each node is divided into two parts first part contains the **Data** and the second contains **Pointer** which points to the next node.

Parts of a Linked List

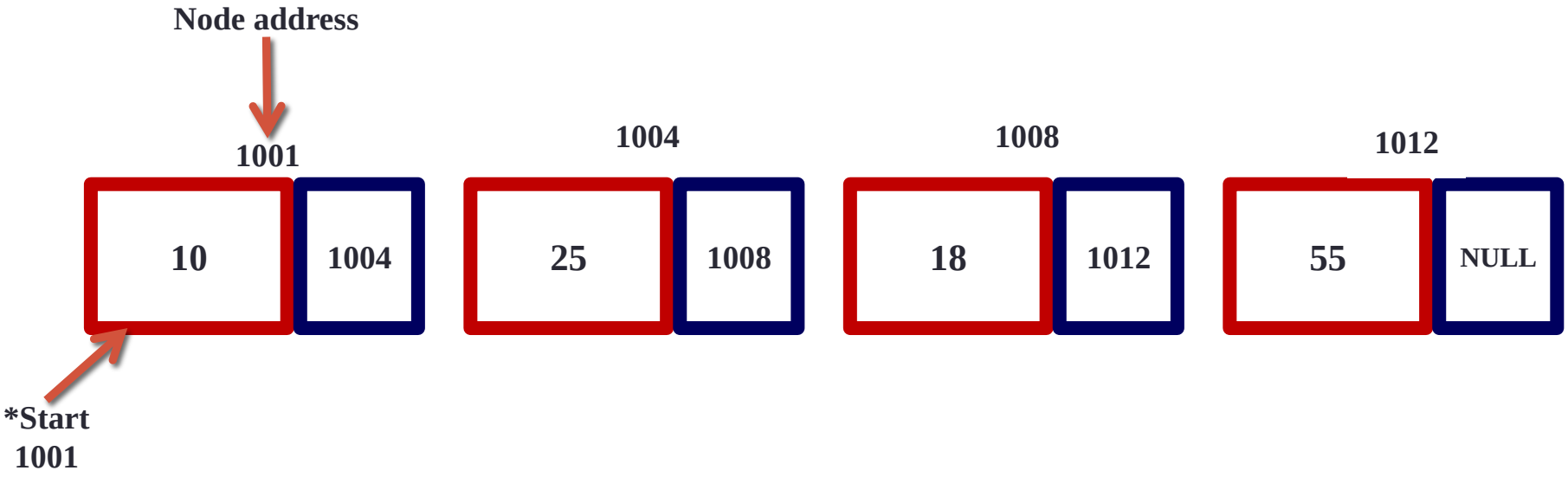




One node is made up of two parts: some Data that it holds, and a reference to the next node

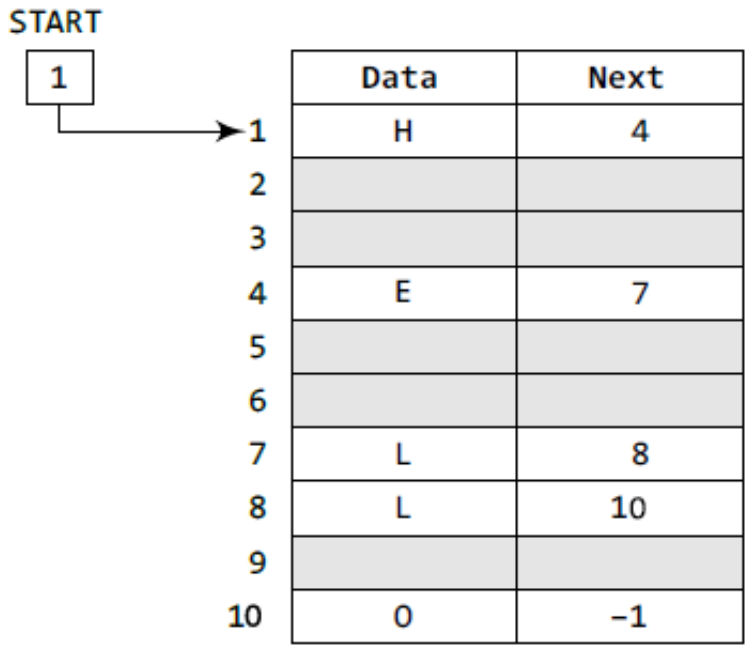
Architecture of Linked lists

- The linked list is a set of nodes, each one in it consists of two parts as follow:
  - First Part: contains data (String, Integer, Etc)
  - Second Part: Contains a pointer that refer to the next address in the memory



Example of a linked list

- START is used to store the address of the first node
- START = 1 so the first data is stored at address 1, which is H.
- The corresponding NEXT stores the address of the next node
- NEXT = 4 The second data element obtained from address 4 is E
- Next = 7 The third data element obtained from address 7 is L
- We repeat this procedure until we reach
- a position where the NEXT entry contains –1 or NULL as this would denote the end of the linked list



	Data	Next
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	–1

Applications of linked list in computer science

- Implementation of stacks and queues.
- Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
- Dynamic memory allocation : We use a linked list of free blocks.
- Maintaining directory of names.
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list
- Representing sparse matrices

Advantages of Linked lists

- It is a dynamic data structure, i.e. a linked list can grow and shrink in size during its lifetime.
- The nodes of linked list (elements) are stored at different memory locations .
- Insertion and deletion in linked list is easy because it does not requires shifting of elements in it.

Disadvantage of Linked List

- They use more memory because of the storage used by their pointers.
- Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.
- Nodes are stored noncontiguous, greatly increasing the time periods required to access individual elements within the list.
- Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards, while doubly linked lists are somewhat easier to read, memory is consumed in allocating space for a back-pointer.
- Searching a particular element in a linked list is difficult and also consuming time.

Array vs Linked List

Array	Linked List
Fixed Size	Dynamic Size
Insertions and deletions are inefficient: elements are usually shifted	Insertions and Deletions are efficient: no shifting.
Random Access: efficient indexing	No random access Not suitable for operations requiring accessing elements using index such as sorting
No memory waste if the array is full or almost full. Otherwise may result in much memory waste	Since memory is dynamically allocated (according to our need) there is no waste of memory
Sequential access is faster [reason: elements are in contiguous memory locations]	Sequential access is slow [reason: elements are not in contiguous memory locations

Array vs Linked List cont.

Array	Linked List
It is necessary to specify the number of elements during declaration (during compile time)	It is not necessary to specify the number of elements during declaration time (memory is allocated during run time)
Occupies less memory for the same number of elements.	Occupies more memory for the same number of elements
Insertion elements at the front of the array is expensive because existing elements need to be shifted	Inserting new elements at any position can be carried out easily

Operations On Linked List

- **Creation**: Creation operation is used to create a linked list with one node.
- **Insertion**: Insertion operation is used to insert a new node at any specified location in the linked list.
- A new node may be inserted.
  a) At the beginning of the linked list
  b) At the end of the linked list
  c) At any specified position in between in a linked list
- **Deletion**: Same as Above

Operations On Linked List cont.

- **Traversing:** is the process of going through all the nodes from one end to another end of a linked list.

- **Searching:** usually searching operations is employed not only while a data item is needed but in case of insertion and deletion at specified location search operation is performed before nodes can be inserted or deleted.

Types of Linked List

- Singly linked list
- Doubly linked list
- Circular linked list

Thank You