7/2/24

ASSIGNMENT 1

Unit - 1&2. Introduction to Data Structure & linear Data Structures.

Q.1 Explain the algorithmic notations.

Algorithmic notations are used to describe algorithms in a clear and concise manner, often without being tied to a specific programming language. Some of the common algorithmic notations are:

i) Pseudocode

Pseudocode is a high-level description of an algorithm that uses natural language mixed with some programming language. It is designed to be easily understandable by humans and is not tied to any specific programming language syntax.

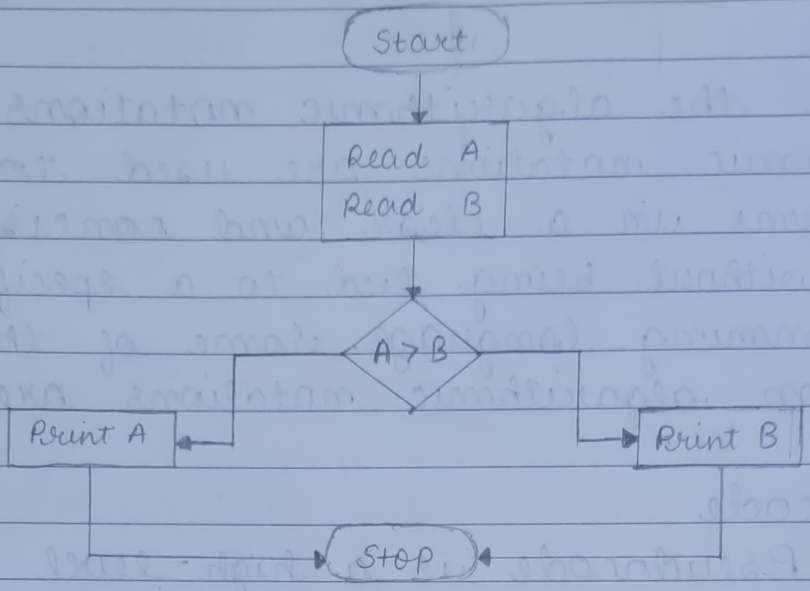
eg:

1. Initialize sum to 0
2. For each number in the list:
   Add number to the sum
3. Return the sum

ii) Flowchart

Flowcharts use graphical symbols to represent the steps of an algorithm. Each symbol represents a specific action or decision and arrows indicate the flow of control between steps.

eg:



iii) Big O Notation

Big O Notation is used to describe the upper bound of the time or space complexity of an algorithm in terms of the input size. It provides a way to classify algorithms based on how they scale with input size.

iv) Theta (Θ) Notation

Similar to Big O Notation, this describes the asymptotically tight bound of an algorithm's time complexity. It provides both upper and lower bounds, representing the best and worst-case scenarios.

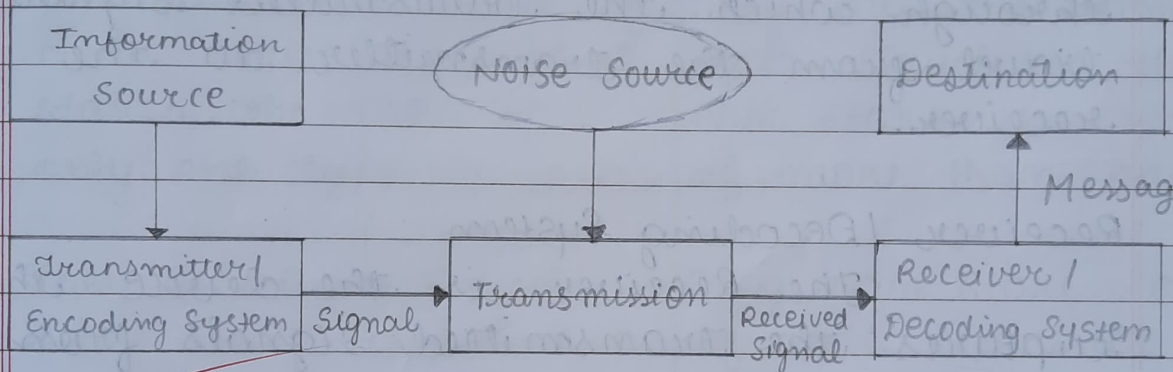
v) Omega (Ω) Notation

Omega Notation represents the lower bound of an algorithm's time complexity.

It describes the best-case scenario for an algorithm.

Q.2 — Explain the transmission of information.

The transmission of information refers to the process of sending data from one point to another, often over a communication channel or network. This process involves several key components and steps:



i) Information Source

The source is the entity that generates the information to be transmitted. It could be a computer system, a sensor or any device capable of producing data.

ii) Transmitter / Encoding System

The encoder is ~~impo~~ responsible for converting the message into a suitable format for transmission. This process may involve encoding the data into binary digits. Then the transmitter, which is a device or system that sends the encoded

message over the communication channel. It typically converts the encoded message into electrical signals, radio waves or light pulses, depending on the type of communication medium used.

iii) Transmission

This is the communication channel which is the physical medium through which the transmitted signals travel from the transmitter to the receiver.

iv) Receiver / Decoding System

The Receiver is the device that captures the transmitted signals from the communication channel. Then the decoder reverses the encoding process, converting the received encoded message back into its original format.

v) Destination

The destination is the intended recipient of the transmitted information.

Q.3 Define data structure and also write down the difference between primitive and non-primitive data structures.

-4 A Data Structure is a way of organizing and storing data in a computer so that

it can be accessed and manipulated efficiently. It defines the relationship between the data elements, the operations that can be performed on the data and the rules for accessing and modifying the data.

	Primitive Data Structures	Non-Primitive Data Structures
-	They are the kind of data structures that stores the data of only one type.	They are the kind of data structures that can store the data of more than one type.
-	They have a fixed size and represent a single value.	They can store collections of data elements of different types and sizes.
-	They will contain some value i.e. it cannot be NULL	They can consist of a NULL value.
-	They can be used to build more complex data structures	They can be used for organizing and managing larger sets of data.
-	Directly operate upon machine instructions	User-defined and offer more flexibility.
-	eg: int, float, char, boolean, etc	eg: arrays, lists, stacks, queues, trees, graphs, hash tables etc

Q.4 Differentiate between linear and non-linear data structure.

—↓

Linear Data Structure	Non-linear Data Structure
> Here, data elements are arranged in a sequential order	Here, data elements are not arranged in a sequential order.
> Follows a single path from one end to another	Traversal can follow multiple paths, branches.
> Elements are connected sequentially	Elements can have hierarchical or interconnected relationship
> Contiguous Memory Allocation	Non-contiguous Memory Allocation.
> Applications of this structures are mainly in application software development	Applications of this structures are in Artificial intelligence and image processing.
> eg: Arrays, Queues, Lists, Stacks	eg: Trees, Graphs.

Q.5 What are the operations to be performed on data structure?

—↓ The operations to be performed on a data

structure depend on the specific type of data structure being used. Here are some of the fundamental operations:

i) Create: This operation involves creating a new instance of a data structure. It sets up the necessary components to hold and organize data according to the structure's specifications.

ii) Destroy: This operation involves deallocating memory or resources associated with a data structure when it is no longer needed.

iii) Selection: This operation involves selecting specific elements from the data structure based on certain criteria, such as filtering or accessing elements by index.

iv) Updation: This operation involves updating the value of existing elements within the data structure, either individually or in bulk.

v) Searching: This operation involves finding a particular element within the data structure based on some search criteria such as searching for a specific value or key.

vi) Sorting: This operation involves arranging the elements within the data structure in a specific order.

vii) Merging: This operation involves combining two or more data structures into a single data structure.

viii) Splitting: This operation involves dividing a data structure into two or more separate data structure.

ix) Traversal: This operation involves visiting each element of the data structure exactly once, often used for processing or analyzing each element in sequence.

Q.6 Differentiate between data types and data structures.

Ans

Data Types	Data Structures
> Data types represent the type of data.	> Data structures organizes and manages data.
> The implementation of a data type is known as abstract implementation	> Data structure implementation is known as concrete implementation
> There is no time	> Here, time complexity

complexity in the case of data types.	plays an important role.
› Data types have typically fixed size	Data structures have dynamic memory allocation
› Operation on data type are simple assignment, comparison	Operation on data structures are insertion, deletion, traversal.
› eg: Integer, Float, String, Boolean	eg: Arrays, lists, Stacks, Queues.

Q.7 Define:

i) Time Complexity

→ Time complexity refers to the amount of time an algorithm takes to complete as a function of the size of its input. It provides an estimate of the number of basic operations that an algorithm performs to solve a problem, relative to the size of the input. Algorithm with lower time complexity are considered more efficient as they can process larger inputs in less time.

ii) Space Complexity

→ Space Complexity refers to the amount of

memory or storage space an algorithm requires to solve a problem as a function of the size of its ~~input~~ input. It measures the maximum amount of memory needed by an algorithm to execute, including both auxiliary space (temporary storage) and space used for the input data itself. Algorithm with lower space complexity are considered more memory-efficient.

Q.8 list out the linear data structure and explain it.

→ linear data structure are those in which data elements are organized in a sequential manner, where each element is connected to its preceding and succeeding elements.

i) Arrays

> An array is a collection of elements stored at contiguous memory location.

> Elements in an array are accessed using an index that represents their position within the array.

> Arrays have a fixed size, determined at the time declaration and elements are stored in a continuous block of memory.

ii) Stacks

> A stack is a Last In, First Out (LIFO) data

structure, where elements are inserted and removed from the same end, known as the top of the stack.

- Stacks support two primary operations: push (insert element onto the top) and pop (remove top element) and additionals are peep, isEmpty.

iii) Queues.

- A Queue is a First In, First Out (FIFO) data structure, where elements are inserted at the rear end (enqueue) and removed from the front end (dequeue).
- Queues support operations like enqueue (insert an element), dequeue (to remove an element), peep and isEmpty.
- Variants of queues include priority queues (elements are removed based on priority) and double-ended queues (dequeues) that support insertion and deletion at both ends.

iv) linked lists

- A linked list is a collection of nodes where each node contains a data element and a reference (or pointer) to the next node in the sequence.
- Unlike arrays, linked lists do not require contiguous memory allocation, allowing for dynamic memory management.
- linked lists support efficient insertion and

deletion operations, especially at the beginning and end of the list, but may have a slower access times compared to arrays.

- There are different types of linked lists, such as singly, doubly and circular linked lists, depending on the number of references each node has.

Q.9 What is an array? Enlist the basic operations on arrays.

→ An array is a linear data structure that stores a collection of elements of the same data type in contiguous memory locations. Each element in the array is ~~associated~~ accessed by its index, which represents its position within the array. Arrays are widely used in programming for storing and accessing data efficiently.

Basic Operations on arrays include:

i) Accessing
ii) Insertion
iii) Update
iv) Search
v) Traversal
vi) Sorting
vii) Deletion

Q.10 What is Stack? List out the stack basic operation

→ A stack is a fundamental data structure that follows the Last In, First Out (LIFO) principle, meaning the last element added to the stack is the first one to be removed. Stacks are commonly used for tasks like expression evaluation, parsing, backtracking and managing function calls.

- Basic Operation on a stack include:

i) Push
ii) Pop
iii) Peep
iv) isEmpty
v) isFull

Q.11 Differentiate between Stack and Queue.

→

Stack	Queue
Stacks are based on the LIFO principle	Queues are based on the FIFO principle.
Insertion and deletion takes places only from one end of the list called the top	Insertion takes place at the rear and deletion takes place from the front of the list.
Stacks is used in	Queue is used in

solving problems that works on recursion	solving problems having sequential processing.
Stack does not have any types.	Queue is of three types, 1. Circular 2. Priority 3. Double-ended.
Can be considered as a vertical collection visual	Can be considered as a horizontal collection visual
Insert operation is called push operation	Insert operation is called enqueue operation
Delete operation is called pop operation	Delete operation is called dequeue operation

Q.12 Explain PUSH and POP operation of the Stack with the algorithm.

→ i) PUSH Operation

The PUSH operation is used to add an element onto the top of the stack. It involves inserting a new element into the stack.

Algorithm :-

→ PUSH(stack, element):

1. // if stack is full:

```
        return "Stack Overflow"
    else:
2.      top = top + 1
3.      stack[top] = element
4.      return.
```

ii) POP Operation

The POP operation is used to remove the top element from the stack. It involves retrieving and removing the element that was most recently added.

Algorithm :-

```
POP(stack):
1.  if stack is empty:
        return "Stack Underflow"
    else:
2.      top = top - 1
3.      return stack[top+1]
```

Q.13 Explain the types of expression. Convert the following infix expression into the postfix expression. Show the stack traces.
A/B$C+D*E/F-G+H

Ans Expressions are statements that represent calculations or relationships between variables, constants and operators.

i) Infix Expression : Operators are placed

between operands like 'A+B', '5 * (C-D)'. It may require the use of parentheses to indicate the order of operations, following standard mathematical rules.

ii) Prefix (Polish) Expression : Operators precede their operands like '+AB', '*5-CD'. It is named after the Polish mathematician Jan ~~Łukasiewicz~~ Łukasiewicz.

iii) Postfix (Reverse Polish) Expression : Operators follow their operands like 'AB+', '5CD-*'. They do not require parentheses to indicate the order of operations

- Infix Expression : A/B$C+D*E/F-G+H

Symbol	Stack	Postfix
A		A
/	/	A
B	/	AB
$	/ $	AB
C	/ $	ABC
+	+	ABC$/
D	+	ABC$/D
*	+ *	ABC$/D
E	+ *	ABC$/DE
/	+ /	ABC$/DE*
F	+ /	ABC$/DE*F

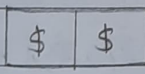
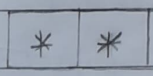
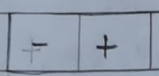
-		ABC$/DE*F/+
G		ABC$/DE*F/+G
+		ABC$/DE*F/+G-
H		ABC$/DE*F/+G-H+

⇒ Final Postfix Expression : ABC$/DE*F/+G-H+

Q.14 Translate the following into polish notation and trace the stack. A+B-C*D*EFG.

→ • Infix Expression : A+B-C*D*EFG.

• Reverse Infix Expression : GFE*D*C-B+A.

Symbol	Stack	Postfix
G		G
$		G
F		GF
$		GF
E		GFE
*		GFE$$
D		GFE$$D
*		GFE$$D
C		GFE$$DC
-		GFE$$DC**
B		GFE$$DC**B
+		GFE$$DC**B-
A		GFE$$DC**BA

• Final Postfix Expression : GFE$$DC**BA+-

⇒ Prefix Expression : -+AB**CD$$EFG

Q.15 Write an algorithm for infix to postfix conversion.

—> An algorithm for converting an infix expression using the Shunting Yard Algorithm:

1. Initialize an empty stack for operators and an empty string for the output.
2. Start scanning the infix expression from left to right.
3. For each symbol encountered:
   - If it is an operand (letter or number), append it to the output.
   - If it is an operator:
     - While the stack is not empty and the precedence of the operator at the top of the stack is greater than or equal to the precedence of the current operator, pop operators from the stack and append them to the output.
     - And if the precedence is lower than the current operator, push it onto the stack.
   - If it is a left parenthesis, push it onto the stack.
   - If it is a right parenthesis, pop operators from the stack and append them to the output until a left parenthesis is encountered. Pop and discard the left parenthesis.
4. After scanning the entire infix expression, pop any remaining operator from the stack and append them to the output.
5. The output string is the postfix expression.

Q.16 Write an algorithm to evaluate postfix expressions. Explain working of the algorithm using appropriate example.

-4 An algorithm to evaluate postfix expressions:

1. Initialize an empty stack for operands.
2. Start scanning the postfix expression from left to right
3. For each symbol encountered:
   - If it is an operand (number), push it onto the stack.
   - If it is an operator:
     - Pop the top two operands from the stack.
     - Perform the operation using the operator and the two operands.
     - Push the result back onto the stack.
4. After scanning the entire postfix expression, the result will be the only element left on the stack, which is the final result of the evaluation.

eg: Postfix Expression : 3 4 + 2 * ⇒ 14

i) Start scanning the postfix expression from left to right.

ii) Encounter operand 3 ; Push 3 onto the stack.
   - Stack : [3]

iii) Encounter operand 4 : Push 4 onto the stack.
   - Stack : [3, 4]

iv) Encounter operator '+': Pop operands 4 and 3, perform addition and push the result (7) onto the stack.
→ Stack : [7]

v) Encounter operand '2': Push 2 onto the stack.
→ Stack : [7, 2]

vi) Encounter operator '*': Pop operands 2 and 7 from the stack, perform multiplication and push the result (14)
→ Stack : [14]

vii) Finish scanning the expression. The result is the only element left on the stack : 14.

Q.17 Convert the following infix expression into the prefix expression. Show the stack traces.
(A * B) $ (C+D) * (E/F-G+H).

→ Infix Expression : (A*B) $ (C+D) * (E/F-G+H)
Reverse Infix Expression : (H+G-F/E)*(D+C) $ (B*A)

Symbol	Stack	Postfix
'('	'('	
H	'('	H
+	'(' +	H
G	'(' +	HG
-	'(' + -	HG+
F	'(' + -	HGF
/	'(' + - /	HGF
E	'(' + - /	HGFE
')'		HGFE/-+

*	*	HGFE/-+
'('	* '('	HGFE/-+
D		HGFE/-+D
+	* '(' +	HGFE/-+D
C		HGFE/-+DC
')'	*	HGFE/-+DC+
$	* $	HGFE/-+DC+
'('	* $ '('	HGFE/-+DC+
B		HGFE/-+DC+B
*	* $ '(' *	HGFE/-+DC+B
A		HGFE/-+DC+BA
')'	* $	HGFE/-+DC+BA*

Final Postfix Expression : HGFE/-+DC+BA*$*

⇒ Prefix Expression : *$*AB+CD+-/EFGH

Q.18 Evaluate the postfix expression using stack:
AB*CD$-EF/G/+ (A=5, B=2, C=3, D=2, E=8, F=2, G=2).

→ Postfix Expression : AB*CD$-EF/G/+ (replace with value)

Symbol	Stack	Operation
5		5
2		5, 2
*	*	5, 2
3		10, 3
2		10, 3, 2
$	$	10, 3, 2
-	-	10, 9
8		1, 8

2		1, 8, 2.
/	/	1, 8, 2
2		1, 4, 2.
/	/	1, 4, 2
+	+	1, 2

∂→ 52*32$-82/2/+ ⇒ 3.

Q.19 What is the queue? Explain insert and delete operation in a simple queue with an algorithm.

—↓ A Queue is a linear data structure that follows the First In, First Out (FIFO) principle, where elements are added at the rear (enqueue) and removed from the front (dequeue).

i) Insert (Enqueue) Operation

Here, a new element is added to the rear end of the queue.

• Algorithm :-

—↓ ENQUEUE (queue, element):

1. if queue is full:
   return "Queue Overflow"

   else:
2. rear = rear + 1
3. queue[rear] = element

4. return

iii) Delete (Dequeue) Operation

Here, the element at the front end of the queue is removed.

- Algorithm :-

→ DEQUEUE (queue) :

1. if queue is empty :
   return "Queue Underflow"
   else :
2. element = queue [front]
3. front = front + 1
4. return element.

Q.20 What is a circular queue ? Compare linear queue and circular queue.

→ A circular queue, also known as a circular buffer or a ring buffer, is a type of queue that uses a fixed-size array to store elements. Unlike a traditional linear queue, where elements are added and removed from opposite ends, in a circular queue, both the front and rear of the queue "wrap around" to the beginning of the array when they reach the end.

	Linear Queue	Circular Queue
>	Arranges the data in a linear pattern	Arranges the data in a circular order where the rear end is connected with the front.
>	It requires more memory space.	It requires less memory space.
>	Can lead to overflow if the rear reaches the end of the array.	Rear wraps around to the beginning of the array, preventing overflow.
>	The insertion and deletion operations are fixed	Insertion and deletion are not fixed and it can be done in any position
>	The order of operations performed on any element is fixed i.e. FIFO.	The order of operations performed on an element may change

Q.21 Explain insert and delete algorithm in a circular queue.

—> i) Insert (Enqueue) Operation

A new element is added to the rear end of the circular queue.

1. First, check if the circular queue is full

by comparing the rear pointer with the front pointer.

2. If the queue is not full, increment the rear pointer to the next position.
3. Insert the new element at the position pointed to by the rear pointer.

Algorithm :-

```
ENQUEUE (circularQueue, value):
1.      if ((rear + 1) % maxSize) == front:
            return error (overflow condition)
        else:
2.          rear = (rear + 1) % maxSize
3.          circularQueue [rear] = value.
```

ii) Delete (Dequeue) Operation.

The element at the front end of the circular queue is removed.

1. Check if the circular queue is empty by comparing the front pointer with rear pointer.
2. If the queue is not empty, retrieve the element at the front of the queue and store it.
3. Move the front pointer to the next position.

Algorithm :-

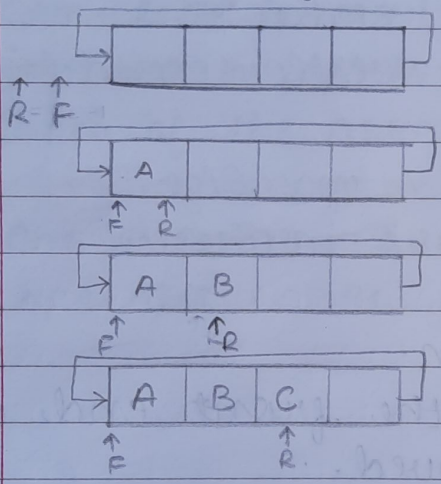
```
DEQUEUE (circularQueue):
1.      if front == rear:
            return error (underflow condition)
```

```
else:
    front = (front + 1) % maxSize
    return circularQueue[front]
```

Q.22 Perform the following operation in a circular queue of length 4 and give the front, rear and size of the queue after each operation.

i) Insert A, B, C.

— 4 Initializing the Circular Queue.



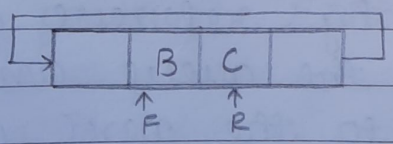
F=-1 R=-1 S=0

F=0 R=0 S=1

F=0 R=1 S=2

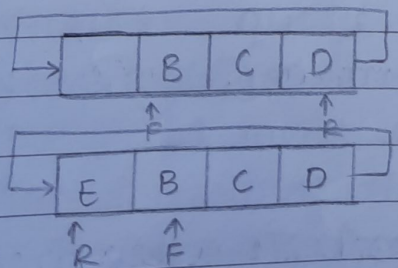
F=0 R=2 S=3

ii) Delete



F=1 R=2 S=2

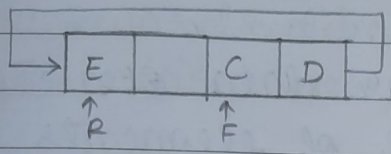
iii) Insert D, E.



F=1 R=3 S=3

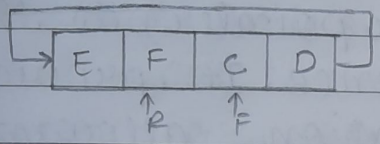
F=1 R=0 S=4

iv) Delete



F = 2 R = 0 S = 3

v) Insert F



F = 2 R = 1 S = 4

Q.23 & Q.24 write an algorithm to insert and delete elements in a circular queue. Design an algorithm to perform insert operation in a circular queue.

→ Same answer as Q.21

Q.25 Design an algorithm to merge two linked lists.

→

```
mergeLists(list1, list2):
1.      if list1 is empty:
            return list2
2.      if list2 is empty:
            return list1

3.      mergedList = null
4.      if list1.value <= list2.value:
            mergedList = list1
            mergedList.next = mergeLists(list1.next, list2)
        else:
5.          mergedlist = list2
            mergedList.next = mergeLists(list1, list2.next)
6.      return mergedlist
```

Q.26 What is linked list? list out and explain the types of linked list.

—4 A linked list is a linear data structure consisting of a sequence of elements, called nodes, where each node contains a data value and a reference (or pointer) to the next node in the sequence. It provides dynamic memory allocation, allowing elements to be efficiently inserted or removed without the need to reallocate memory.

Types of linked lists :-

i) Singly linked lists
- Here, each node contains a data value and a single pointer that points to the next node in the sequence.
- The last node's pointer points to null, indicating the end of the list.
- It allows traversal only in the forward direction, starting from the head node.

ii) Doubly linked lists
- Here, each node contains a data value, a pointer to the next node, and a pointer to the previous node in the sequence.
- The first node's previous pointer and the last node's next pointer point to null, indicating the beginning and end of the list.

- It allow traversal in both forward and backward directions, providing more flexibility compared to singly linked lists.

iii) Circular linked list

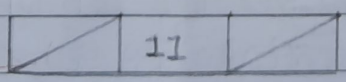
- The last node's pointer points back to the first node (head) in the sequence, forming a circular structure.
- It can be either singly or doubly linked.
- They are useful in applications where continuous access to elements is required, such as in scheduling algorithms.

Q.27 Explain creation, insertion and deletion of doubly linked lists with example.

→ i) Creation of Doubly Linked List

- To create, you start with an empty list where each node has a pointer to the next node and a pointer to previous node.

eg: Create a node with data '11'.



ii) Insertion in Doubly Linked List

- Insertion can happen at the beginning, end or at any specified position in between.

- Insertion at the beginning :-

Firstly create a new node then make the next of the ~~next to~~ new node as the current head.

* Deletion Part is at pg-36.

Make the previous of the current head as the new node and update the head to point to the new node.

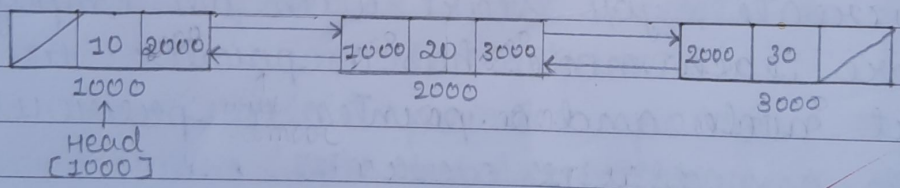
- Insertion at the End :-

Firstly create a new node, traverse the list to last node then make the next of the last node point to new node. Make the previous of new node point to last node and update the new node as the last node.

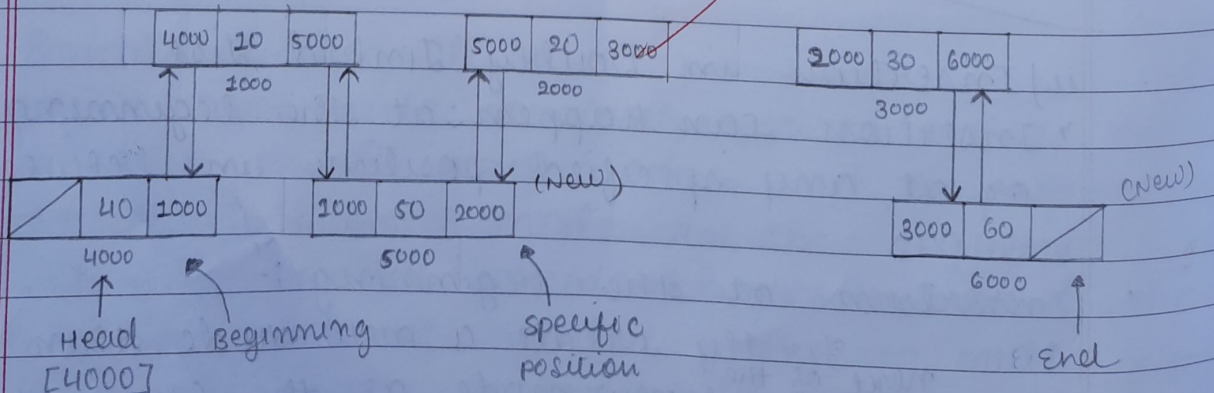
- Insertion at a Specific Position :-

Firstly create a new node, traverse the list to the desired position and adjust pointers of current node and its neighbors to insert the new node.

eg:



- Now, insert three nodes at beginning, at specific position and at end.



Q.28 Write an algorithm to insert an element into a singly linked list.

—↓ The insertion operation involves adding a new element (or node) into the list at a specific position. There are various scenarios for insertion, including at the beginning, at the end or at a specific position in the middle of the list.

• Algorithm :-

```
INSERT(head, value):
1.      newNode = createNode(value)

2.      if head == NULL:
            head = newNode
        else:
3.          current = head
4.          while current.next != NULL:
                current = current.next

5.      current.next = newNode
```

Q.29 Differentiate between Array and linked list.

—↓

Array	linked list
They are stored in contiguous block of memory	They ~~are~~ can be scattered in memory, and ~~not~~ it is non-contiguous.

They are fixed in size and need resizing for dynamic changes	They are dynamic in size and can easily grow or shrink.
They require typically less memory overhead.	They requires additional memory for pointers.
They are better for random access and fixed-size collections	They are better for frequent insertion/deletion and dynamic collections.
Constant Time O(1) for accessing elements by index.	Linear Time O(n) for accessing elements, as you have to traverse from the beginning.
Insertion and deletion can be slow if resizing is required, O(n)	Fast insertion/deletion at beginning or end (O(1)) but O(n) for insertion/deletion in middle due for traversal.

Q.30 Explain circular linked lists in brief.

A circular linked list is a variation of linked list in which the last node points back to the first node, forming a circular structure. In other words, the next pointer of the last node in the list does not point to

null but instead points to the first node, creating a loop.

- In a circular linked list:

1. Each node contains a data value and a pointer to the next node in the sequence.
2. The last node's next pointer points back to the first node, closing the loop and forming the circular structure.
3. Traversal can start from any node in the list but typically, there is a designated "head" node from which traversal ~~of~~ begins.
4. Circular linked lists can either be singly or doubly linked, meaning that each node can have either one or two pointers.

- Advantages :-

> Efficient Operations
> Implementation of Circular Buffers.
> Memory Utilization.

- Disadvantages :-

> Memory Overhead
> Insertion and Deletion in the Middle
> Complexity.

Q.27. (iii) Deletion in from Doubly linked list

> Deletion can ~~be~~ also happen at the beginning, end, or at any specific position

- Deletion at the beginning :-

Firstly update the head to point to the next node and make the previous of the new head as None.

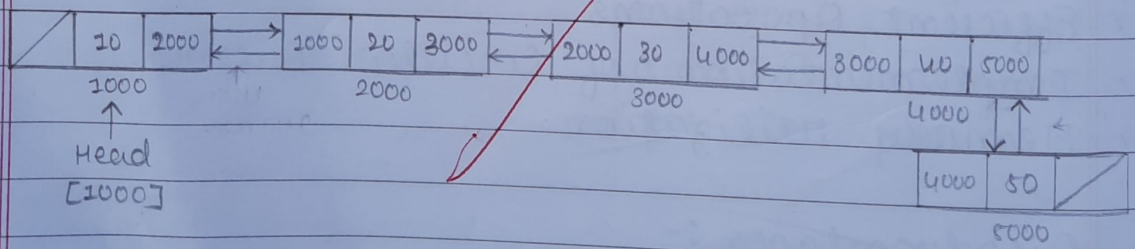
- Deletion at the end :-

Firstly traverse the list to the last node and make the second last node's next as None. Then free the memory of last node.

- Deletion at a specific position :-

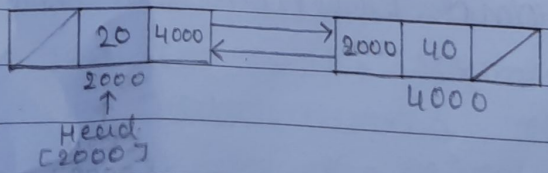
Firstly traverse the list to the desired position, then adjust pointers of the previous and next nodes to skip the node to be deleted. Then free the memory of the node to be deleted.

eg:



Q.A. 5.13

Now, delete three nodes at beginning, at specific position and at end.



xxx x xxx