210242 : FUNDAMENTALS OF DATA STRUCTURES

(2019 Pattern) (Semester - III) (210242)

- **Chapter Wise Most IMP Answer Key –**

UNIT – 3 Searching and Sorting	Marks
Searching : Search Techniques - Sequential Search / Linear Search, Variant of Sequential Search- Sentinel Search, Binary Search, Fibonacci Search, and Indexed Sequential Search. **Sorting**: Types of Sorting-Internal and External Sorting, General Sort Concepts-Sort Order, Stability, Efficiency, and Number of Passes, Comparison Based Sorting Methods-Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Shell Sort, Non-comparison Based Sorting Methods-Radix Sort, Counting Sort, and Bucket Sort, Comparison of All Sorting Methods and their complexities.	**18 Marks**

Sr.	Questions & Answers
1	**Explain the selection sort with algorithm sort the given no. using selection sort & show the content of array after every pass. What is the time complexity of selection sort?** **Given no: 34, 9, 78, 65, 12, -8**
--	**Selection Sort :** - Selection Sort is a comparison-based sorting algorithm. - It works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the array and swapping it with the first unsorted element. - This process is repeated until the entire array is sorted. **Working** : - Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element scan the entire list to

find the smallest element and swap it with the second element. Then starting from the third element the entire list is scanned in order to find the next smallest element. Continuing in this fashion we can sort the entire list.

- Generally, on pass i ($0 \leq i \leq n-2$), the smallest element is searched among last n-i elements and is swapped with A[i]

Algorithm of Selection Sort:

1. Start from the first element (i = 0) of the array.
2. Set the first unsorted element as the minimum.
3. Compare this element with the rest of the array to find the smallest element.
4. Swap the smallest element with the first unsorted element.
5. Move the boundary of the sorted array by one element to the right.
6. Repeat the process for the remaining unsorted part of the array until the array is fully sorted.

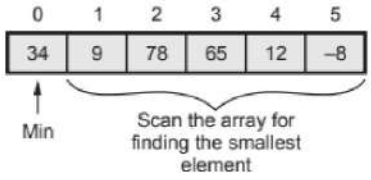
Example : Sort given array by using selection sort method 34, 9, 78, 65, 12,-8. Show step by step execution of all passes. What is the time complexity of selection sort?

Solution : Let

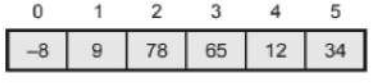
34	9	78	65	12	–8

be the given elements.

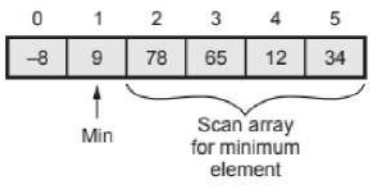
Pass 1 : Consider the elements A[0] as the first element. Assume this as the minimum element.



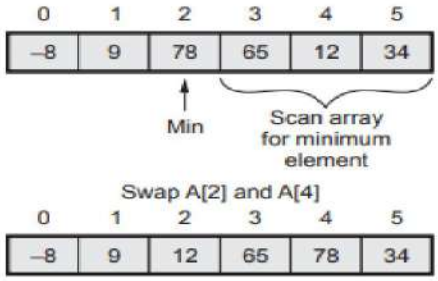
If smallest element is found, swap it with A[0]



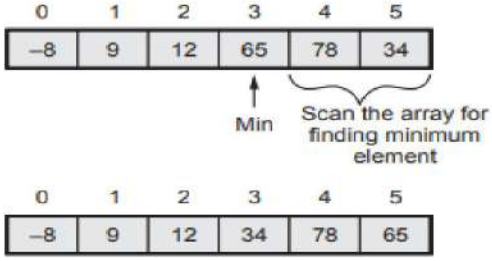
Pass 2 :



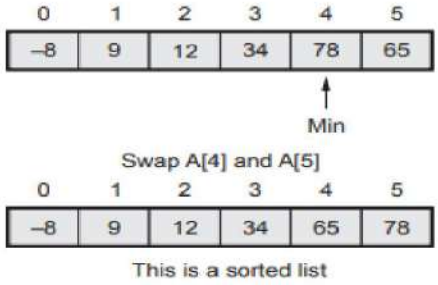
Pass 3 :



Pass 4 :



Pass 5 :



Time Complexity –

- Best Case: $O(n^2)$
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Some Practice Question elements –

1) 50, 23, 03, 18, 9, 01, 70, 21, 20, 6, 40, 04.
2) 27, 76, 17, 9, 45, 58, 90, 79, 100
3) 81, 5, 27, –6, 61, 93, 4, 8, 104, 15

Tips –

- *Practice all above question as it is asked in each example for 9 marks.*
- *While writing answers read question carefully & write according to it.*

2	**Explain in brief the different searching techniques. What is the time complexity of each of them?**
--	**1) Sequential Search (Linear Search)** - Sequential search is the simplest searching algorithm. - It works by checking each element of the array (or list) one by one, starting from the first element, until the target element is found or the entire array is traversed. **Time Complexity** : - The time complexity of this algorithm is O(n). - The time complexity will increase linearly with the value of n. - For higher value of n sequential search is not satisfactory solution. **Steps:** - Start from the first element and compare it with the target element. - If it matches, return the position of the element. - If not, continue checking the next element. - If no match is found by the end of the array, return "element not found". **Advantages**: - Simple to implement - Not require specific ordering before applying **Disadvantages**: - It is less efficient --- **2) Fibonacci Search** - In binary search method we divide the number at mid and based on mid element i.e. by comparing mid element with key element we search either the left sublist or right sublist. - Thus we go on dividing the corresponding sublist each time and comparing mid element with key element. - This algorithm is particularly useful for searching large, sorted arrays and can perform better than binary search in some cases. **Time Complexity:** - The time complexity of this algorithm is O(1).

Steps of Fibonacci Search Algorithm:

- **Find Fibonacci Number**:
  Find the smallest Fibonacci number greater than or equal to the array length n. Let this be *f(m)*.
- **Initialize Pointers**:
  Let two preceding Fibonacci numbers be f(m−1) and f(m−2).
  While the array has elements to check:

- Compare the key with the element at index f(m−2).
- If key matches, return the index.
- If key is smaller, move Fibonacci pointers two steps down to the left (reducing the search space).
- If key is larger, move Fibonacci pointers one step down to the right, and adjust the offset.

- **Single Element Check**:
  If only one element remains, check if it matches the key. Return the index if it matches.

Example:

According to the algorithm following is the example of sorting the elements using Fibonacci Search.

0	1	2	3	4	5	6
10	20	30	40	50	60	70

∴ n = 7, we want to find key = 60

Now check Fibbonacci series.

As n < 8

set f = 8

a = 5

b = 3

			(b) 3	4	(a) 5	6		(f) 8
10	20	30	40	50	60	70	...	

Set offset = – 1

∴ i = 2 ∵ 1 = min (offset + b, n – 1)

A[i] = A[2] < (key = 60)

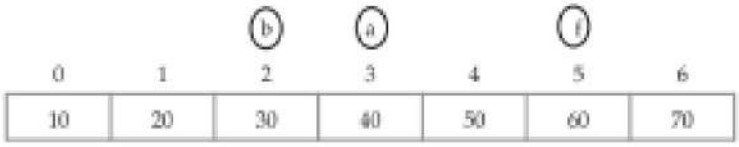
Here key is greater than the element

We move f one fibbonacci down (step 2C of algorithm)

∴ f = 5

a = 3

b = 2



Set offset = i i.e. = **2**

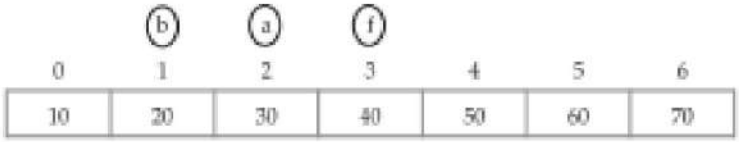
Now i = 4 ∵ i = min (offset + b, n – 1)

Here key is again greater than the element. So we move f, one fibbonacci down

∴ f = 3

a = 2

b = 1



Set offset = i i.e. **4**

Now new i = 5 ∵ i = min (offset + b, n – 1)

Here a[i] = key. Hence return value of i as the position of key element.

Note that due to fibonacci numbering the search portion is restricted and we need to compare very less number of elements.

Diagram Representation (*Optional-for understanding*)**:**

Step	Fibonacci Numbers	Offset	i (Index Checked)	Value at A[i]	Comparison
1	f = 8, a = 5, b = 3	-1	i = 2	30	60 > 30
2	f = 5, a = 3, b = 2	2	i = 4	50	60 > 50
3	f = 3, a = 2, b = 1	4	i = 5	60	60 == 60

Thus, the key **60** is found at index **5** using Fibonacci Search.

---

3) Binary Search

- Binary search is an efficient algorithm for searching in sorted arrays.
- Binary search is a searching algorithm in which the list of elements is divided into two sublists and the key element is compared with the middle element.

- If the match is found then, the location of middle element is returned otherwise, we search into either of the halves(sublists) depending upon the result produced through the match.

Algorithm for Binary Search

- if(low>high)
- return;
- mid (low+high)/2;
- if(x==a[mid])
- return (mid);
- if(x<a[mid])
- search for x in a[low] to a[mid-1];
- else
- search for x in a[mid+1] to a[high];

Example

As mentioned earlier the necessity of this method is that all the elements should be sorted. So let us take an array of sorted elements.

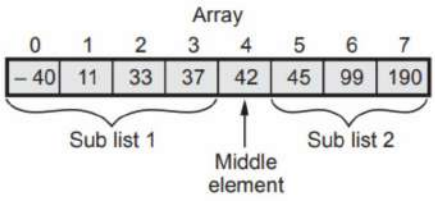
array

0	1	2	3	4	5	6	7
– 40	11	33	37	42	45	99	100

Step 1 : Now the key element which is to be searched is = 99 ∴ key = 99.

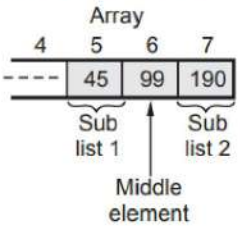
Step 2 : Find the middle element of the array. Compare it with the key

if middle ? key
i.e. if 42 ? 99
if 42 < 99 search the sublist 2

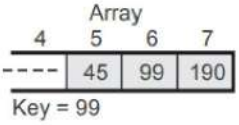


Now handle only sublist 2. Again divide it, find mid of sublist 2

if middle ? key
i.e. if 99 ? 99



So match is found at 7^{th} position of array i.e. at array [6]



Time Complexity:

- The time complexity of this algorithm is O(1).

---

4) Index Sequential Search

- Index sequential search is a searching technique in which a separate table containing the indices of the actual elements is maintained.
- The actual search of the element is done with the help of this index table.

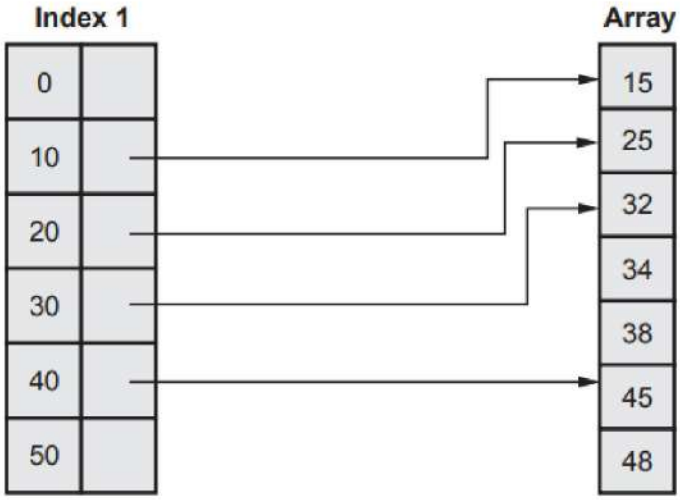
Steps:

- Create an index with key elements pointing to their positions in the dataset.
- Use the index to find the section where the target element might be.
- Perform a linear search within that section.

Time Complexity:

- The time complexity of this algorithm is O(1).

For example -



- Note that the sorted index table is maintained. First we search the index table and then with the help of an index, actual array is searched for the element.

---

5) Sentinel Search

- Sentinel search is an optimization over linear search.
- The sentinel value is a specialized value that acts as a flag or dummy data. This specialized value is used in context of an algorithm which uses its presence as a condition of termination.

	- Typical examples of sentinel values are 1. Null character used at the end of the string. 2. Null pointer at the end of the linked list. 3. End of file character. - In sentinel search, when searching for a particular value in an unsorted list, every element will be compared against this value. **Time Complexity**: - The time complexity of this algorithm is O(1). **Example:** Input: [10,20,30,40,50,60] Key: 40 Result: The element is present Input: [10,20,30,40,50,60] Key: 99 Result: The element is not present in the list.
3	**Write a pseudo code for binary search apply your algorithm on the following no. stored in an array to search no: 23 & 100. 9,17,23,40,45,52,58,80,85,95,100**

--	**Pseudo Code for Binary Search**
	def binary_search(A, low, high, key):
	if low > high:
	return -1
	mid = (low + high) // 2
	if A[mid] == key:
	return mid
	elif A[mid] > key:
	return binary_search(A, low, mid - 1, key)
	else:
	return binary_search(A, mid + 1, high, key)
	--- 	
	Execution of Binary Search on the Given Array
	A = [9, 17, 23, 40, 45, 52, 58, 80, 85, 95, 100]

- **Search for Key = 23**
  1. **Initial setup**:
     - low = 0, high = 10
     - mid = (0 + 10) // 2 = 5, so A[5] = 52
     - Since 52 > 23, update high = mid - 1 = 4
  2. **Second step:**
     - low = 0, high = 4
     - mid = (0 + 4) // 2 = 2, so A[2] = 23
     - Key found at index 2

- **Search for Key = 100**
  1. **Initial setup**:
     - low = 0, high = 10
     - mid = (0 + 10) // 2 = 5, so A[5] = 52
     - Since 52 < 100, update low = mid + 1 = 6
  2. **Second step**:
     - low = 6, high = 10
     - mid = (6 + 10) // 2 = 8, so A[8] = 85
     - Since 85 < 100, update low = mid + 1 = 9
  3. **Third step**:
     - low = 9, high = 10
     - mid = (9 + 10) // 2 = 9, so A[9] = 95
     - Since 95 < 100, update low = mid + 1 = 10
  4. **Fourth step**:
     - low = 10, high = 10
     - mid = (10 + 10) // 2 = 10, so A[10] = 100
     - Key found at index 10

Note

a. *You can write another pseudocode too*

b. *Execution of Binary Search on the Given Array can be written in another ways mentioned in Q.2.*

c. *Check Details and do practice- Following is the Practice Example*

A[0] to A[10] : 9, 17, 23,38,45,50,57,76,90,100 to search numbers 10 & 100.

4	**Explain quick sort algorithm with suitable example. What is time complexity of quick sort algorithm**

--	**Quick sort algorithm**
	- Quick Sort is a highly efficient and widely used sorting algorithm.
	- It uses the divide-and-conquer approach to sort elements in an array.
	- In this method division is dynamically carried out. The three steps of quick sort as follows:
	▪ Divide – Split the array into two sub arrays
	▪ Conquer – Recursively sort the two sub arrays
	▪ Combine – Combine all sorted elements to form sorted elements.
	
	Algorithm
	The quick sort algorithm is performed using following two important functions Quick and partition.
	
	Quick (A [0...n-1],low,high)
	//Problem Description: This algorithm performs sorting of
	//the elements given in Array A [0...n-1]
	//Input: An array A[0...n-1] in which unsorted elements are given.
	//The low indicates the leftmost element in the list
	//and high indicates the rightmost element in the list
	//Output: Creates a sub array which is sorted in ascending order
	if(low<high)**then**
	//split the array into two sub arrays
	m -- partition(A[low...high])// m is mid of the array
	Quick (A[low...m-1])
	Quick (A[mid+1...high])
	
	Example:
	Consider 25, 57, 48, 37, 12, 92, 86, 33 Sort stepwise using radix sort.
	
	Solution : Consider the first element as a pivot element.
	
	25 57 48 37 12 92 86 33
	↑ ↑ ↑
	Pivot i j

Now, if A[i] < Pivot element then increment i. And if A[j] > Pivot element then decrement j. When we get these above conditions to be false, Swap A[i] and A[j]

25	33	48	37	12	92	86	57
↑	i						j
Pivot							

25	33	48	37	12	92	86	57
↑	i			j			
Pivot							

Swap A[i] and A[j]

Low							High
25	12	48	37	33	92	86	57
	j	i					

As j > i Swap A[Low] and A[j]

After pass 1 :

[12]	25	[48	37	33	92	86	57]
		↑	↑				↑
		Pivot	i				j

12	25	[48	37	33	92	86	57]
					i		j
		Low					

12	25	[48	37	33	92	86	57]
				j	i		

As j > i, we will swap A[j] and A[Low]

After pass 2 :

12	25	[33	37]	48	[92	86	57]

After pass 3 :

12	25	33	37	48	[92	86	57]

Assume 92 to be pivot element

12	25	33	37	48	[92	86	57]
					↑	i	j
					Pivot		

12	25	33	37	48	92	86	57	
							j	i

As j > i swap 92 with 57.

After pass 4 :

12	25	33	37	48	57	[86	92]

After pass 5 :

12	25	33	37	48	57	86	92

is a sorted list.

-Time Complexity:

- **Best Case**: O (n log n)
- **Average case**: O (n log n)
- **Worst Case**: O(n^2)

Practice Question: *(must be practiced)*

1) 29, 57, 47, 39, 36, 20, 55, 28, 31, 39.
2) 27, 76, 17, 9, 57, 90, 45, 100, 79

5 **Explain insertion, bubble, radix sort algorithm with suitable example. What is time complexity of it.**

--

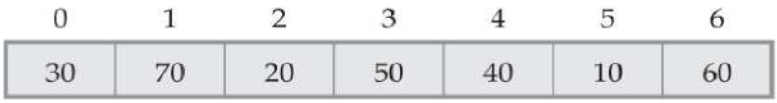
1. Insertion Sort

- In this method the elements are inserted at their appropriate place.
- Insertion Sort builds a sorted array one element at a time by repeatedly taking the next element from the unsorted portion and inserting it into its correct position in the sorted portion.

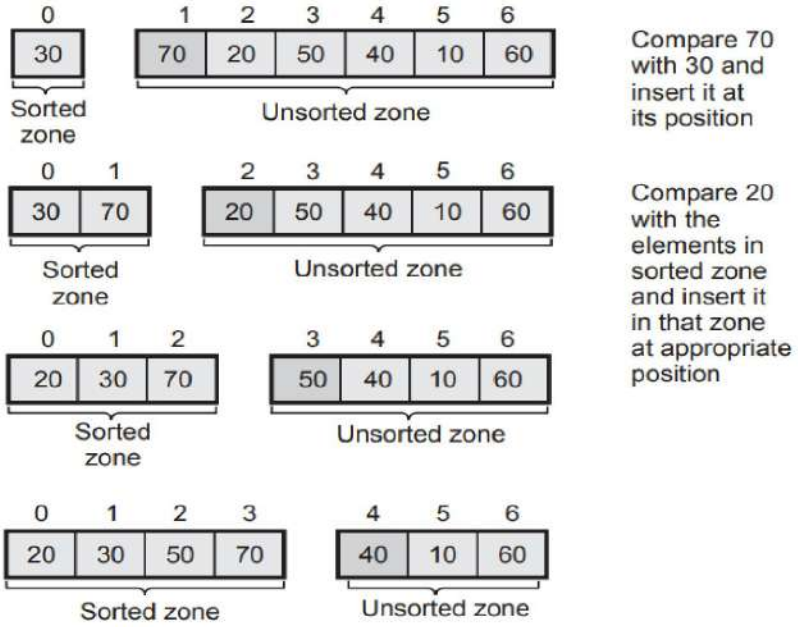
Example:

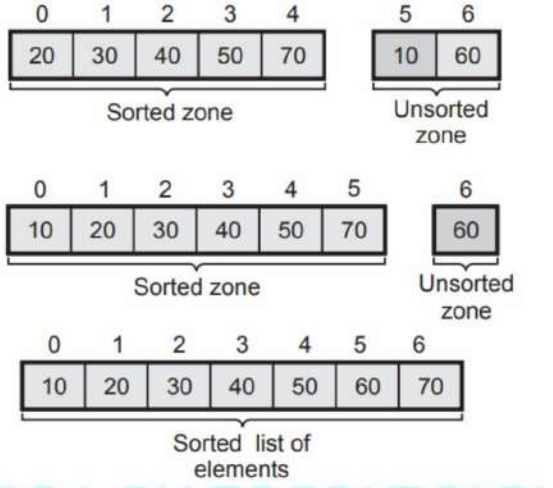
A [30, 70, 20, 50, 40, 10, 60]

Consider a list of elements as,



The process starts with first element





Time Complexity:

- Best Case: O(n)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Practice question: *55, 85, 45, 11, 34, 5, 89, 99, 67*

---

2. Bubble Sort

- This is the simplest kind of sorting method in this method.
- Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is sorted.

Example:

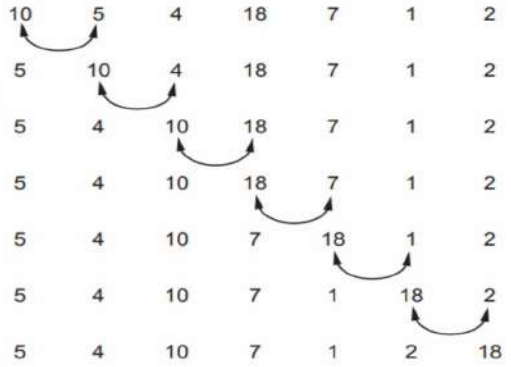
A [10, 5, 4, 18, 17, 1, 2]

Solution: Let, 10, 5, 4, 18, 17, 1, 2 be the given list of elements. We will adjacent elements say A[i] and A[j]. If A[i]>A[j] then swap the elements.

Pass 1

Compare 10 and 5: Swap → 5,10,4,18,17,1,2
Compare 10 and 4: Swap → 5,4,10,18,17,1,2
Compare 10 and 18: No Swap → 5,4,10,18,17,1,2
Compare 18 and 17: Swap → 5,4,10,17,18,1,2
Compare 18 and 1: Swap → 5,4,10,17,1,18,2
Compare 18 and 2: Swap → 5,4,10,17,1,2,18

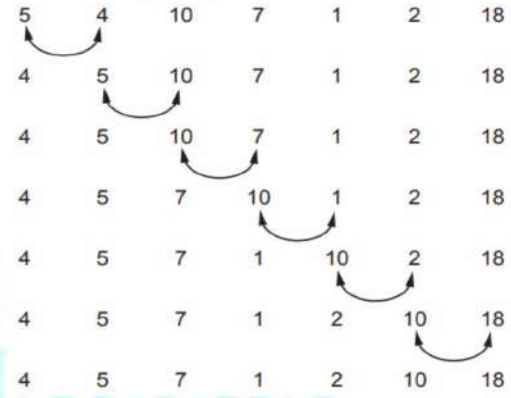
End of Pass 1: 5, 4, 10, 17, 1, 2, 18



Pass 2:

Compare 5 and 4: Swap → 4,5,10,17,1,2,18
Compare 5 and 10: No Swap → 4,5,10,17,1,2,18
Compare 10 and 17: No Swap → 4,5,10,17,1,2,18
Compare 17 and 1: Swap → 4,5,10,1,17,2,18
Compare 17 and 2: Swap → 4,5,10,1,2,17,18
Compare 17 and 18: No Swap → 4,5,10,1,2,17,18

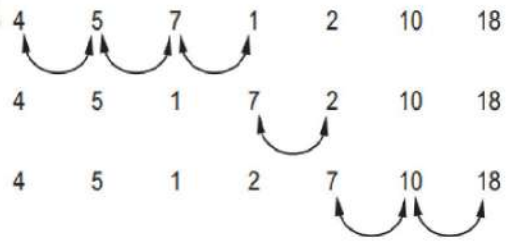
End of Pass 2: 4,5,10,1,2,17,18



Pass 3:

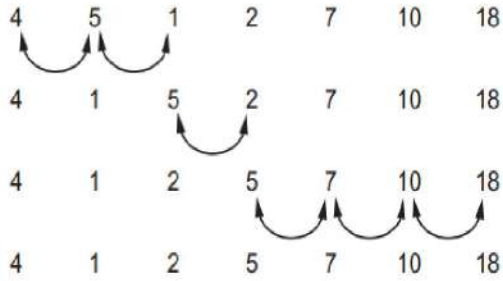
Compare 4 and 5: No Swap → 4,5,10,1,2,17,18
Compare 5 and 10: No Swap → 4,5,10,1,2,17,18
Compare 10 and 1: Swap → 4,5,1,10,2,17,18
Compare 10 and 2: Swap → 4,5,1,2,10,17,18
Compare 10 and 17: No Swap → 4,5,1,2,10,17,18
Compare 17 and 18: No Swap → 4,5,1,2,10,17,18

End of Pass 3: 4,5,1,2,10,17,18



Pass 4:

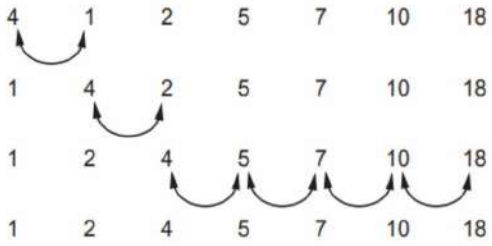
Compare 4 and 5: No Swap → 4,5,1,2,10,17,18
Compare 5 and 1: Swap → 4,1,5,2,10,17,18
Compare 5 and 2: Swap → 4,1,2,5,10,17,18
Compare 5 and 10: No Swap → 4,1,2,5,10,17,18
Compare 10 and 17: No Swap → 4,1,2,5,10,17,18
Compare 17 and 18: No Swap → 4,1,2,5,10,17,18



End of Pass 4: 4,1,2,5,10,17,18

Pass 5:

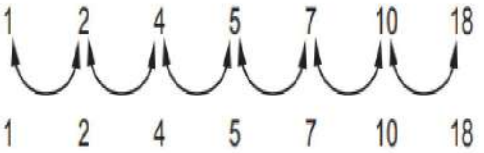
Compare 4 and 1: Swap → 1,4,2,5,10,17,18
Compare 4 and 2: Swap → 1,2,4,5,10,17,18
Compare 4 and 5: No Swap → 1,2,4,5,10,17,18
Compare 5 and 10: No Swap → 1,2,4,5,10,17,18
Compare 10 and 17: No Swap → 1,2,4,5,10,17,18
Compare 17 and 18: No Swap → 1,2,4,5,10,17,18



End of Pass 5: 1,2,4,5,10,17,18

Pass 6:

Compare 1 and 2: No Swap → 1,2,4,5,10,17,18
Compare 2 and 4: No Swap → 1,2,4,5,10,17,18
Compare 4 and 5: No Swap → 1,2,4,5,10,17,18
Compare 5 and 10: No Swap → 1,2,4,5,10,17,18
Compare 10 and 17: No Swap → 1,2,4,5,10,17,18
Compare 17 and 18: No Swap → 1,2,4,5,10,17,18



End of Pass 6: 1,2,4,5,10,17,18

Sorted Array: 1,2,4,5,10,17,18

Time Complexity:

- Best Case: O(n)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

***Practice question**: 64, 34, 25, 12, 22, 11, 90*

**Note: This question can be asked for 9 marks as it is asked in may_june-2024 exam so all steps may mandatory for 9 marks.*

---

3. Radix Sort

- Radix Sort is a non-comparison-based sorting algorithm that sorts numbers by processing individual digits.
- In this method sorting can be done digit by digit and thus all the elements can be sorted.

Example:

A [25, 06, 45, 60, 40, 50]

Solution:

Step 1: Sort the elements according to last digit and sort them.

Last digit	Elements
0	50.60,40
1	
2	
3	
4	
5	25,45
6	06
7	
8	
9	

Sorted Elements after Step 1: 50, 60, 40, 25, 45, 06

Step 2: Sort the elements according to second last digits and sort them.

Second Last digit	Elements
0	06
1	
2	25
3	
4	40,45
5	50
6	60
7	
8	
9	

Sorted Elements after Step 2: 06, 25, 40, 45, 50, 60

	Sorted Array: 06, 25, 40, 45, 50, 60 **Time Complexity**: • Best Case: O(nlogn) • Average Case: O(nlogn) • Worst Case: O(nlogn) ***Practice question***: *29, 57, 47, 39, 36, 20, 55, 28, 31, 39* **Note: This question can be asked for 9 marks as it is asked in nov_dec-2023 exam so all steps may mandatory for 9 marks.*

UNIT – 4 Linked List	Marks
Introduction to Static and Dynamic Memory Allocation, **Linked List:** Introduction, of Linked Lists, Realization of linked list using dynamic memory management, operations, Linked List as ADT, **Types of Linked List:** singly linked, linear and Circular Linked Lists, Doubly Linked List, Doubly Circular Linked List, Primitive Operations on Linked List- Create, Traverse, Search, Insert, Delete, Sort, Concatenate. Polynomial Manipulations-Polynomial addition. Generalized Linked List (GLL) concept, Representation of Polynomial using GLL	**18 Marks**

6	**Explain Generalized Linked List with example Explain the representation of polynomial using GLL. Explain node structure**
--	**Generalized Linked List** - A Generalized Linked List (GLL) is a flexible data structure that allows for the representation of heterogeneous data types, including other lists. - This makes GLLs particularly useful for representing complex structures like trees, graphs, and mathematical expressions. **<u>Concept:</u>** - A generalized linked list A, is defined as a finite sequence of n>=0 elements, a_1, a_2, a_3, ..., a_n, such that ai are either atoms or the list of atoms. Thus A = (a_1, a_2, a_3, ..., a_n) - Where n is total number of nodes in the list. - Now to represent such a list of atoms we will have certain assumptions about the node structure

Flag	Data	Down Pointer	Next Pointer

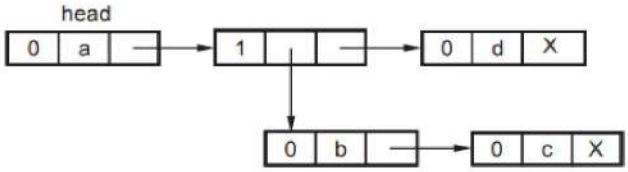
Flag = 1 means down pointer exists.
= 0 means next pointer exists.

- Data means the atom
- Down pointer is address of node which is down of the current node.
- Next pointer is the address of the node which is attached as the next node.

With this typical node structure let us represent the generalized linked list for the above. Let us take few example to learn how actually the generalized linked list can be shown,

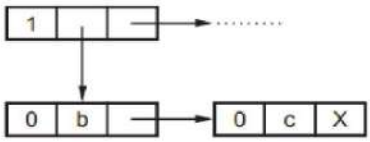
Example

1. (a , (b , c), d)



In above example the head node is

In this case the first field is 0, it indicates that the second field is variable. If first field is 1 means the second field is a down pointer, means some list is starting.



The above figure indicates (b , c). In this way the generalised linked list can be built. The X in the next pointer field indicates NULL value.

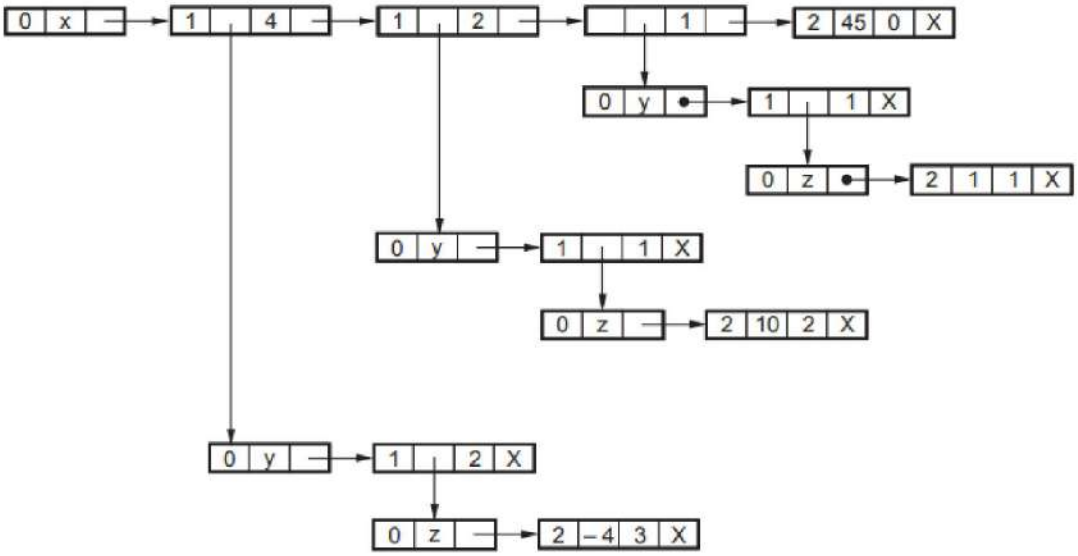
Representation of polynomial using GLL

- A Generalized Linked List (GLL) provides an efficient way to represent polynomials, particularly those involving multiple variables.
- By using GLLs, each term of the polynomial can be treated as a node, which can also contain a nested list for its exponents.
- This structure allows for dynamic representation, making it easy to handle various polynomial forms.

- The typical node structure for representation of polynomial is –

- Flag = 0 means variable is present
  Flag = 1 means down pointer is present.
  Flag = 2 means coefficient and exponent is present

Example: – $4x^4y^2z^3 + 10x^2yz^2 + 7xyz + 45$



GLL Node structure:

A typical GLL node may have the following components:

1. **Data Field**: This can store either a simple data value (like an integer or float) or a pointer to another GLL, allowing for nesting.
2. **Next Pointer**: A pointer to the next node in the same list.
3. **Down Pointer**: A pointer to another GLL, used when the node points to a nested list.

Example: You can write above example if not given
(Above mentioned diagram is node structure actually)

Note: This question can be asked with different types so don't be confused, If ask GLL write above or if only polynomial representation, then write second only.

Example for practice: 5x7 + 7xy6 + 11xz.

7	**What is doubly linked list? Write pseudo code for following function using Doubly Linked List of integer numbers.** **i) Insert given value as last value in the list.** **ii) Delete first node from the list.** **iii) Delete last node from the list.**

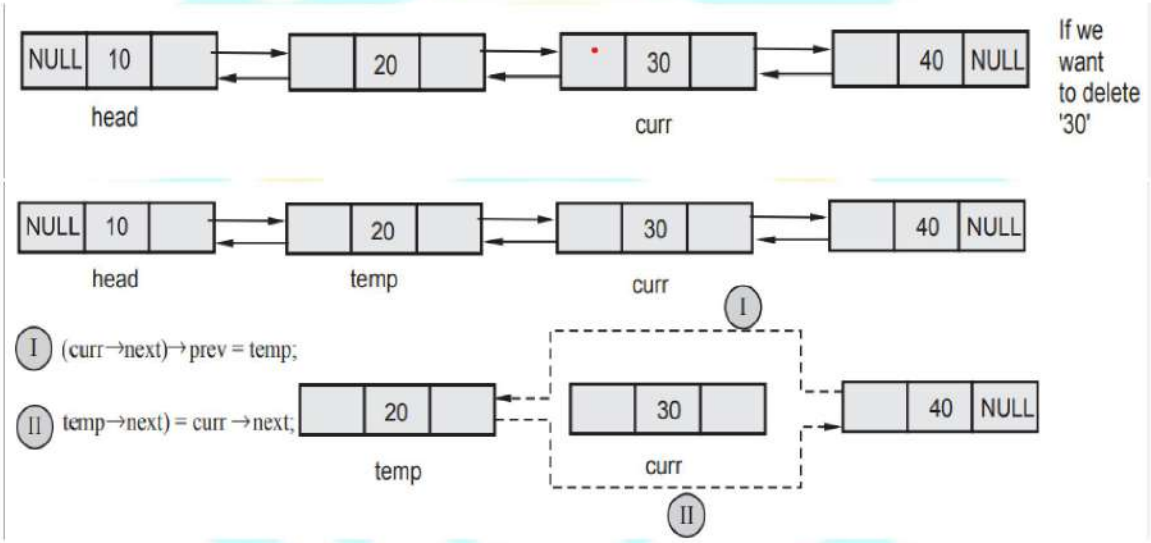
--	**Doubly linked list**
	- This structure allows traversal in both directions (forward and backward) and facilitates easier deletion and insertion operations compared to singly linked lists.
	- A Doubly Linked List is a data structure that consists of nodes, where each node contains three parts:
	1. **Data**: The value stored in the node.
	2. **Next Pointer**: A pointer to the next node in the list.
	3. **Previous Pointer**: A pointer to the previous node in the list.
	- Typical Structure of each node in doubly in linked list
	Prev | Data | Next
	Node Structure
	- Here's a typical structure for a node in a doubly linked list in pseudo code:
	Node {
	int data
	Node* prev
	Node* next
	}
	Operations on Doubly Linked List –
	1. Creation of node
	2. Display of Doubly linked list
	3. Deletion of node
	4. Insertion of node
	--- 	
	Q. Explain the process of deletion of an element from doubly linked list with ex –
	- Deletion of node in doubly linked list

- Steps of deletion of an element from doubly linked list.

1. Identify the Node to be Deleted:
2. Adjust the Previous Node's Next Pointer:
3. Adjust the Next Node's Previous Pointer:
4. Delete the Target Node:

Example:

- Consider,



Then delete **curr**.

Q. Write pseudo code for following function using Doubly Linked List of integer numbers.

i) Insert given value as last value in the list.
ii) Delete first node from the list.
iii) Delete last node from the list.

i) Insert given value as last value in the list.

```
void insertAtEnd(Node*& head, int value) {
   Node* newNode = new Node(value);
   if (!head) { head = newNode; return; }
   Node* current = head;
   while (current->next) current = current->next;
   current->next = newNode;
   newNode->prev = current;
}
```

ii) Delete first node from the list

```
void deleteFirst(Node*& head) {
  if (!head) return;
  Node* temp = head;
  head = head->next;
  if (head) head->prev = nullptr;
  delete temp;
}
```

iii) Delete last node from the list.

```
void deleteLast(Node*& head) {
  if (!head) return;
  if (!head->next) { delete head; head = nullptr; return; }
  Node* current = head;
  while (current->next) current = current->next;
  current->prev->next = nullptr;
  delete current;
}
```

8 **What is singly linked list? Write pseudocode to perform –**

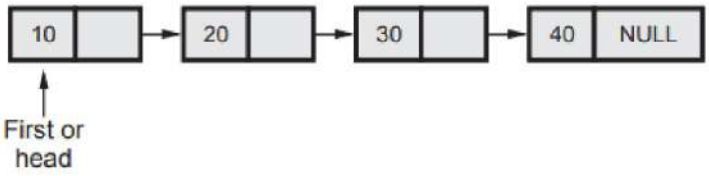
i) Write a pseudo code to insert new node in to singly link list.

ii) Merging of two sorted singly linked lists of integers into third list. Write complexity of it.

--

Singly linked list

- It is called singly linked list because it contains only one link which points to the next node.
- The very first node is called head or first.
- It contains two parts:
  1. **Data**: Stores the value of the element.
  2. **Next Pointer**: A pointer that points to the next node in the list.



i) Write a pseudo code to insert new node in to singly link list.

```
struct Node {
  int data;
  Node* next;
  Node(int value) : data(value), next(nullptr) {}
};

void insertAtEnd(Node*& head, int value) {
  Node* newNode = new Node(value);
  if (!head) { head = newNode; return; }
  Node* current = head;
  while (current->next) current = current->next;
  current->next = newNode;
}
```

Complexity:

- Time Complexity – O(n)
- Space Complexity – O(1)

---

ii) Merging of two sorted singly linked lists of integers into third list.

```
Node* mergeSortedLists(Node* list1, Node* list2) {
  Node* mergedList = nullptr;
  Node** tail = &mergedList;

  while (list1 && list2) {
    if (list1->data < list2->data) {
      *tail = list1;
      list1 = list1->next;
    } else {
      *tail = list2;
      list2 = list2->next;
    }
    tail = &((*tail)->next);
  }

  *tail = (list1) ? list1 : list2;
  return mergedList;
}
```

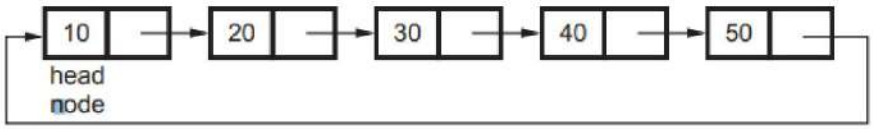
Complexity:

- Time Complexity – O(n+m)
- Space Complexity – O(1)

9 **Explain circular linked list with it's basic operation.**
Explain node structure of Circular Singly Linked List

Circular linked list

- A circular linked list is a variation of the linked list where the last node points back to the first node, forming a circle.
- In a circular singly linked list, each node has:
  1. **Data**: The value stored in the node.
  2. **Next Pointer**: A pointer that points to the next node in the list.
- The circular linked list is as shown below –



Node Structure of Circular linked list

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

--

Various operations of Circular linked list

1. Creation of circular linked list

- To create a circular linked list, we start with an empty list and insert the first node.
- This node's next pointer should point to itself, making it circular

Code:

```
void createCircularList(Node*& head, int value) {
   Node* newNode = new Node(value);
   head = newNode;
   newNode->next = head;    // Points to itself, making the list circular
}
```

2. Display of Circular linked list

- To display a circular linked list, we need to traverse the list starting from the head node and continue until we come back to the head.

Code:

```
void displayCircularList(Node* head) {
  if (!head) return;
  Node* current = head;
  do {
    cout << current->data << " ";
    current = current->next;
  } while (current != head);
  cout << endl;
}
```

3. Insertion of circular linked list

- Can insert at the beginning or end of the list by updating pointers.

Code:

```
void insert(Node*& head, int value, Node* afterNode = nullptr) {
  Node* newNode = new Node(value);
  if (!head) {
    head = newNode;
    newNode->next = head;
    return;
  }
  if (!afterNode) {
    // Insertion at the beginning
    Node* last = head;
    while (last->next != head) last = last->next;
    newNode->next = head;
    last->next = newNode;
    head = newNode;
  } else {
    // Insertion after a specific node
    newNode->next = afterNode->next;
    afterNode->next = newNode;
  }
}
```

4. Deletion of any node

- To delete a specific node, we need to search for the node to delete and adjust the pointers accordingly.

Code:

```
void deleteNode(Node*& head, int value) {
    if (!head) return;
    Node *current = head, *previous = nullptr;

    do {
        if (current->data == value) {
            if (current == head) {
                Node* last = head;
                while (last->next != head) last = last->next;
                if (head == head->next) {
                    delete head;
                    head = nullptr;
                } else {
                    last->next = head->next;
                    head = head->next;
                    delete current;
                }
            } else {
                previous->next = current->next;
                delete current;
            }
            return;
        }
        previous = current;
        current = current->next;
    } while (current != head);
}
```

5. Searching a node from circular linked list

- To search for a node with a specific value, we traverse the list starting from the head and continue until we circle back to the head.

Code:

```
bool searchNode(Node* head, int value) {
    if (!head) return false;
```

```
        Node* current = head;
        do {
            if (current->data == value) return true;
            current = current->next;
        } while (current != head);
        return false;
    }
```

Note: You can also write example for explanations in place of code & also it may be compulsory according to question asked.

10	**Write pseudocode to perform addition of two polynomials using singly linked list & doubly linked lists into third list.**
--	**1) pseudocode to perform addition of two polynomials using singly linked list**

```
Node* addPolynomials(Node* poly1, Node* poly2) {
    Node* result = nullptr;
    Node** tail = &result;

    while (poly1 && poly2) {
        if (poly1->exponent == poly2->exponent) {
            *tail = new Node(poly1->coefficient + poly2->coefficient, poly1->exponent);
            poly1 = poly1->next;
            poly2 = poly2->next;
        } else if (poly1->exponent > poly2->exponent) {
            *tail = new Node(poly1->coefficient, poly1->exponent);
            poly1 = poly1->next;
        } else {
            *tail = new Node(poly2->coefficient, poly2->exponent);
            poly2 = poly2->next;
        }
        tail = &((*tail)->next);
    }

    while (poly1) {
        *tail = new Node(poly1->coefficient, poly1->exponent);
        poly1 = poly1->next;
        tail = &((*tail)->next);
    }
```

```
        while (poly2) {
            *tail = new Node(poly2->coefficient, poly2->exponent);
            poly2 = poly2->next;
            tail = &((*tail)->next);
        }

        return result;
    }
```

---

2) pseudocode to perform addition of two polynomials using doubly linked lists into third list

```
    DNode* addPolynomials(DNode* poly1, DNode* poly2) {
        DNode* result = nullptr;
        DNode* tail = nullptr;

        while (poly1 && poly2) {
            if (poly1->exponent == poly2->exponent) {
                DNode* newNode = new DNode(poly1->coefficient + poly2->coefficient,
    poly1->exponent);
                poly1 = poly1->next;
                poly2 = poly2->next;
            } else if (poly1->exponent > poly2->exponent) {
                DNode* newNode = new DNode(poly1->coefficient, poly1->exponent);
                poly1 = poly1->next;
            } else {
                DNode* newNode = new DNode(poly2->coefficient, poly2->exponent);
                poly2 = poly2->next;
            }
            if (!result) result = tail = newNode;
            else {
                tail->next = newNode;
                newNode->prev = tail;
                tail = newNode;
            }
        }

        while (poly1) {
            DNode* newNode = new DNode(poly1->coefficient, poly1->exponent);
            if (!result) result = tail = newNode;
            else {
```

```
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
          }
          poly1 = poly1->next;
        }

        while (poly2) {
          DNode* newNode = new DNode(poly2->coefficient, poly2->exponent);
          if (!result) result = tail = newNode;
          else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
          }
          poly2 = poly2->next;
        }

        return result;
      }
```

Note: You can write another code also.

UNIT – 5 Stack	Marks
Basic concept, stack Abstract Data Type, Representation of Stacks Using Sequential Organization, stack operations, Multiple Stacks, **Applications of Stack** - Expression Evaluation and Conversion, Polish notation and expression conversion, Need for prefix and postfix expressions, Postfix expression evaluation, Linked Stack and Operations. **Recursion**- concept, variants of recursion- direct, indirect, tail and tree, backtracking algorithmic strategy, use of stack in backtracking.	**17 Marks**

11	**Write algorithm for postfix expression evaluation. Explain with suitable example.**

--

Algorithm for postfix expression evaluation

1. Read the postfix expression from left to right.
2. If the input symbol read is an operand, then push it on to the stack.
3. If the operator is read POP two operands and perform arithmetic operations if operator is

   \+ then result = operand 1 + operand 2

   \- then result = operand 1 operand 2

   * then result = operand 1 * operand 2

   / then result = operand 1 / operand 2
4. Push the result onto the stack.
5. Repeat steps 1-4 till the postfix expression is not over.

For Example –

Consider postfix expression –

AB + C – C$ – for A = 1, B = 2, C = 3

Given: AB + C - C $ -

- A = 1
- B = 2
- C = 3

- Substituting the values into the expression:

  1 2 + 3 - 3 $ -

Step-by-Step Evaluation:

1. Initial Expression: 1 2 + 3 - 3 $ -
2. Initialize an empty stack.

Step-by-Step Evaluation:

1. **1** → Operand → Push onto the stack.
   - **Stack**: [1]
2. **2** → Operand → Push onto the stack.
   - **Stack**: [1, 2]
3. **+** → Operator:
   - Pop the top two operands: 2 and 1.
   - Compute: 1 + 2 = 3.

- Push result back onto the stack.
- **Stack**: [3]

4. **3** → Operand → Push onto the stack.
   - **Stack**: [3, 3]
5. - → Operator:
   - Pop the top two operands: 3 and 3.
   - Compute: 3 - 3 = 0.
   - Push result back onto the stack.
   - **Stack**: [0]
6. **3** → Operand → Push onto the stack.
   - **Stack**: [0, 3]
7. **$** → Operator (Assuming $ represents **exponentiation**):
   - Pop the top two operands: 3 and 0.
   - Compute: 0 ^ 3 = 0 (0 raised to the power of 3 is 0).
   - Push result back onto the stack.
   - **Stack**: [0]
8. **-** → Operator:
   - Pop the top two operands: 0 (only one value is present).
   - Result of 0 - 0 = 0.
   - Push result back onto the stack.
   - **Stack**: [0]

Final Result:

- The final result is 0.

Explanation

- In the given example, $ was assumed to be an **exponentiation operator**, commonly denoted by ^.
- The postfix expression was evaluated step-by-step using a stack, where operators performed computations using the topmost stack elements and pushed the results back onto the stack.

***Note**: You can write another example with another steps as it is acceptable.*

12	**Write rules to convert given infix expression to postfix expression using stack. Explain procedure with Example.**
--	**Rules for Conversion** 1) **Operands**: Add directly to the output. 2) **Left Parenthesis [( ]**: Push onto the stack. 3) **Right Parenthesis [ )]**: Pop from the stack to the output until a left parenthesis is encountered. Discard the left parenthesis. 4) **Operators**: While the stack is not empty and the top has equal or greater precedence than the current operator, pop the stack to the output; then push the current operator onto the stack. 5) **End of Expression:** Pop all operators from the stack to the output. **Operator Precedence** • ^ (highest) • *, / • +, - (lowest) **Associativity** • Left-to-right for +, -, *, / • Right-to-left for ^ **Example:** (A* B – (C + D * E) ^ (F * G / H))

Step	Input	Action	Stack	Output
1	(	Push (on to the stack.	(	
2	A	Add A to output.	(	A
3	*	Push * onto the stack.	(*	A
4	B	Add B to output.	(*	A B
5	-	Pop * to output; push - onto stack.	(	A B *
6	(	Push (on to the stack.	(- (	A B *
7	C	Add C to output.	(- (	A B * C
8	+	Push + on to the stack.	(- (+	A B * C
9	D	Add D to output.	(- (+	A B * C D
10	*	Push * onto the stack.	(- (+ *	A B * C D
11	E	Add E to output.	(- (+ *	A B * C D E
12	)	Pop * and + to output; pop (.	(-	A B * C D E *
13	^	Push ^ onto the stack.	- ^	A B * C D E *
14	(	Push (onto the stack.	- ^ (	A B * C D E *
15	F	Add F to output.	- ^ (	A B * C D E * F

16	*	Push * onto the stack.	- ^ (*	A B * C D E * F
17	G	Add G to output.	- ^ (*	A B * C D E * F G
18	/	Pop * to output; push /.	- ^ (/	A B * C D E * F G *
19	H	Add H to output.	- ^ (	A B * C D E * F G * H
20	)	Pop / and * to output; pop (.	- ^	A B * C D E * F G * H /
21	)	Pop ^ to output; pop (.	-	A B * C D E * F G * H / ^
22		Pop - to output.		A B * C D E * F G * H / ^ -

---

*Practice Question :1. (P * Q – (L + M * N) ^ (X * Y / Z)*

*2. (a+b) * d + e/(f + a*d) + c*

13 What is infix, prefix and postfix expression? Give each example

--

Infix Expression:

- An expression where operators are placed between operands.
- In this type of expressions the arrangement of operands and operator is as follow

Infix expression = operand1 operator operand2

- Infix expression are the most natural way of representing the expressions.
- Parenthesis can be used in these expressions.
- For example
  1. (a+b)
  2. (a+b) * (c-d)
  3. a + b * c

Postfix Expression (also known as Polish Notation):

- An expression where operators follow their operands.
- In this type of expressions the arrangement of operands and operator is as follow

Postfix expression = operand1 operand2 operator

- All the corresponding operands come first and then operator can be placed.
- There is no Parenthesis used in these expressions.
- For example
  1. ab+
  2. ab + cd – *
  3. ab + e / df +*

Prefix Expression (also known as Polish Notation):

- An expression where operators precede their operands.
- In this type of expressions the arrangement of operands and operator is as follow

Prefix expression = operator operand1 operand2

- All the corresponding operators come first and then operands are arranged
- There is no Parenthesis used in these expressions.
- For example
  1. + ab
  2. + a * b c
  3. *+ ab – cd

14	**What is concept of recursion?** **Explain with example three different types/ variants of recursion.**

--	**Recursion:**
	- Recursion is a programming concept where a function calls itself in order to solve a problem.
	- It typically involves breaking down a problem into smaller subproblems, each of which is similar to the original problem.
	- By recursion the same task can be performed repeatedly.
	- Recursion consists of two main parts:
	1. The base case, which stops the recursion,
	2. The recursive case, which continues the recursion.
	--- 	
	Different types/ variants of recursion.
	1. Direct Recursion
	- In direct recursion, a function calls itself directly.
	- This is the most straightforward form of recursion.
	- A C function is directly recursive if it contains an explicit call to itself.
	- For example
	int fun1(int n)
	{
	if(n<=0)
	return n:
	else

```
        return fun1(n-1)
}
```

2. Indirect Recursion

- In indirect recursion, a function calls another function, which in turn calls the original function.
- This creates a cycle of function calls.
- A C function is indirectly recursive if functionl calls function2 and function2 ultimately calls function1.
- For example

```
int fun1(int n)
{
    if(n<=0)
        return n;
    else
        return fun2(n);
}
int fun2(int m)
{
        return fun1(n-1);
}
```

3. Tail Recursion

- In tail recursion, the recursive call is the last operation in the function.
- This allows for optimizations by some compilers and interpreters, as it can reduce the amount of stack space needed.
- For example

```
int tailRecursiveFactorial(int n, int accumulator = 1) {
    if (n == 0) return accumulator;
    return tailRecursiveFactorial(n - 1, n * accumulator);
}
int main() {
    int number;
    cout << "Enter a non-negative integer: ";
    cin >> number;
```

```
    if (number < 0) {
        cout << "Factorial is not defined for negative numbers." << endl;
    } else {
        cout << "Factorial of " << number << " is: " <<
      tailRecursiveFactorial(number) << endl;
    }
    return 0;
}
```

15	**Explain the pseudo-C/C++ code to implement stack with overflow and underflow conditions. or Basic operation of stack**
--	Basic operation of stack – **1. Push**: - Push is a function which inserts new element at the top of the stack. **2. Pop:** - It deletes the element at the top of the stack. **3. Peek:** - The peek operation retrieves the top element without removing it. - We check if the stack is empty before returning the value.

```
bool push(Stack &s, int value) {
  if (s.top >= MAX_SIZE - 1) {          // Check for overflow
    return false;                       // Stack overflow
  }
  s.arr[++s.top] = value;               // Increment top and add value
  return true;                          // Successful push
```

```
bool push(Stack &s, int value) {
  if (s.top >= MAX_SIZE - 1) {          // Check for overflow
    return false;                       // Stack overflow
  }
  s.arr[++s.top] = value;               // Increment top and add value
  return true;                          // Successful push
}
```

```
bool peek(Stack &s, int &value) {
  if (s.top < 0) {                      // Check for underflow
    return false;                       // Stack is empty
  }
  value = s.arr[s.top];                 // Get the top value
  return true;                          // Successful peek
}
```

```
#define MAX_SIZE 100
struct ArrayStack {
   int arr[MAX_SIZE];
   int top;
};

void initialize(ArrayStack &s) {
   s.top = -1;
}
bool push(ArrayStack &s, int value) {
   if (s.top >= MAX_SIZE - 1) return false;
   s.arr[++s.top] = value;
   return true;
}
bool pop(ArrayStack &s, int &value) {
   if (s.top < 0) return false;
   value = s.arr[s.top--];
   return true;
}
bool peek(ArrayStack &s, int &value) {
   if (s.top < 0) return false;
   value = s.arr[s.top];
   return true;
}
```

Stack Implementation Using Singly Linked List

```
struct Node {
   int data;
   Node* next;
};
struct LinkedListStack {
   Node* top;
};
void initialize(LinkedListStack &s) {
   s.top = nullptr;
}
bool push(LinkedListStack &s, int value) {
```

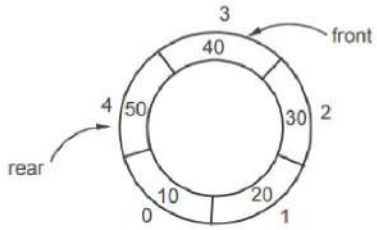
```
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = s.top;
    s.top = newNode;
    return true;
}
bool pop(LinkedListStack &s, int &value) {
    if (s.top == nullptr) return false;
    Node* temp = s.top;
    value = temp->data;
    s.top = s.top->next;
    delete temp;
    return true;
}
bool peek(LinkedListStack &s, int &value) {
    if (s.top == nullptr) return false;
    value = s.top->data;
    return true;
}
```

UNIT – 6 Queue	Marks
Basic concept, Queue as Abstract Data Type, Representation of Queue using Sequential organization, Queue Operations, Circular Queue and its advantages, Multi-queues, Linked Queue and Operations. **Deque**-Basic concept, types (Input restricted and Output restricted), Priority Queue- Basic concept, types (Ascending and Descending).	**17 Marks**

16	**What is circular queue? Draw and explain implementation of Circular queue using array. Explain it's basic operation. Explain the advantages of circular queue over linear queue.**
--	**Circular queue:** - A circular queue is a linear data structure that follows the FIFO (First In, First Out) principle but connects the last position back to the first position, making it circular. - This structure efficiently utilizes space, especially when elements are added and removed frequently.

Formula for setting the front & rear pointers,

rear = (rear +1) % size front = (front + 1) % size



Q. Write pseudocode for Add, Remove operations. ***or***
Q. Write pseudo code to implement circular queue using array

/* insert function */

```
void Queue::insert(int item)
{
   if (front==(rear+1)%MAX)
   {
     cout << "Queue is full\n";
   }
   else
    {
     //setting front pointer for a single element in Queue
       if(front==-1)
       front=rear=0;
       else
       rear=(rear+1)%MAX;
       Que[rear]=item;
       }
   }
```

/* delete function */

```
int Queue::delete()
{
int val;
if(front==-1)
{
  cout << "Queue is empty\n";
  return 0;     // return null on empty Queue
```

```
    }
    val= Que[front];      //item to be deleted
    if(front==rear)       //when single element is present
    {
       front=rear= -1;
    }
    else
      front(front+1)%MAX;

    return val;
    }
```

Advantages of circular queue over linear queue.

1) **Efficient Space Utilization**: In a linear queue, once elements are dequeued, the allocated space cannot be reused, leading to wasted space. Circular queues allow the reuse of empty slots by connecting the end of the array back to the front.
2) **No Overflow Until Full**: Circular queues can handle more insertions until all positions are filled, while linear queues may indicate they are full even if there is available space at the beginning.
3) **Constant Time Operations**: Both enqueue and dequeue operations in a circular queue are performed in O(1) time, as they involve simple arithmetic operations. Linear queues might require shifting elements, which increases time complexity.
4) **Better Performance for Queuing Tasks**: Circular queues are particularly useful in scenarios like scheduling or buffering, where tasks are frequently added and removed, ensuring better performance.
5) **Simplified Implementation**: The circular nature simplifies the implementation of the queue operations by eliminating the need for resizing or shifting elements.
6) **Better Handling of Wraparound**: In a circular queue, when the end of the array is reached, the pointers wrap around seamlessly, whereas linear queues can become inefficient as they fill up.

	Basic Operation of Circular queue 1. Enqueue (Add an Element) 2. Dequeue (Remove an Element) 3. Peek (View the Front Element) 4. isEmpty 5. isFull
17	**Draw and explain implementation of Linear Queue using Singly Linked List. Explain Add, Remove, Queue Full and Queue Empty operations**
--	**Implementation of a Linear Queue Using Singly Linked List –** A linear queue can be implemented using a singly linked list, which allows for dynamic sizing. In this implementation, each node in the linked list represents an element of the queue. **Structure** 1. **Node Structure**: Each node contains: ○ A data field to store the value. ○ A pointer to the next node. 2. **Queue Structure**: The queue will maintain two pointers: ○ front: Points to the first node (front of the queue). ○ rear: Points to the last node (rear of the queue). **Pseudocode for Operations** **1. Node Definition** STRUCT Node INTEGER data Node next **2. Queue Definition** STRUCT Queue Node front Node rear **3. Initialize Queue** FUNCTION initializeQueue() RETURNS Queue CREATE Queue SET front = NULL SET rear = NULL RETURN Queue

Operations

1. Add (Enqueue)

- **Purpose**: To add an element at the rear of the queue.
- **Steps**:
  - Create a new node with the given value.
  - If the queue is empty (front == NULL), set both front and rear to this new node.
  - If the queue is not empty, link the new node to the next of the current rear and update rear to the new node.

```
FUNCTION enqueue(queue: Queue, value: INTEGER) RETURNS BOOLEAN
  CREATE newNode
  SET newNode.data = value
  SET newNode.next = NULL

  IF queue.front == NULL THEN
    SET queue.front = newNode
    SET queue.rear = newNode
  ELSE
    queue.rear.next = newNode
    SET queue.rear = newNode
  ENDIF
  RETURN TRUE
```

---

2. Remove (Dequeue)

- **Purpose**: To remove and return the front element of the queue.
- **Steps**:
  - If the queue is empty (front == NULL), return an error or indicate that the queue is empty.
  - Retrieve the value from the front node.
  - Move the front pointer to the next node.
  - If the queue becomes empty after the operation, set rear to NULL as well.

```
FUNCTION dequeue(queue: Queue) RETURNS INTEGER
  IF queue.front == NULL THEN
    PRINT "Queue is empty"
    RETURN -1  // or some error code
  ENDIF
```

```
    INTEGER value = queue.front.data
    SET queue.front = queue.front.next

    IF queue.front == NULL THEN
      SET queue.rear = NULL
    ENDIF

    RETURN value
```

---

3. Queue Full

- **Purpose**: To check if the queue is full.
- **Implementation**:
  Since a linked list can grow dynamically, a linear queue implemented using a linked list is generally never full unless memory is exhausted. Thus, this operation is not typically implemented in this case.

```
FUNCTION isFull(queue: Queue) RETURNS BOOLEAN
  // Not applicable for linked list-based implementation
  RETURN FALSE
```

---

4. Queue Empty

- **Purpose**: To check if the queue is empty.
- **Steps**:
  - Check if front is NULL

```
FUNCTION isEmpty(queue: Queue) RETURNS BOOLEAN
  RETURN queue.front == NULL
```

Write pseudocode for Linear Queue Implementation using array.

```
CONSTANT MAX_SIZE = 5

STRUCT Queue
  ARRAY queue[MAX_SIZE]
  INTEGER front
  INTEGER rear

FUNCTION initializeQueue() RETURNS Queue
  CREATE Queue
  SET front = -1
  SET rear = -1
  RETURN Queue

FUNCTION isEmpty(queue: Queue) RETURNS BOOLEAN
```

```
  RETURN front == -1
FUNCTION isFull(queue: Queue) RETURNS BOOLEAN
  RETURN rear == MAX_SIZE - 1
FUNCTION enqueue(queue: Queue, value: INTEGER) RETURNS BOOLEAN
  IF isFull(queue) THEN
    PRINT "Queue is full"
    RETURN FALSE
  ENDIF
  IF isEmpty(queue) THEN
    SET front = 0
  ENDIF
  SET rear = rear + 1
  queue.queue[rear] = value
  RETURN TRUE
FUNCTION dequeue(queue: Queue) RETURNS INTEGER
  IF isEmpty(queue) THEN
    PRINT "Queue is empty"
    RETURN -1
  ENDIF
  INTEGER value = queue.queue[front]
  FOR i FROM front TO rear - 1
    queue.queue[i] = queue.queue[i + 1]
  ENDFOR
  SET rear = rear - 1
  IF rear == -1 THEN
    SET front = -1
  ENDIF
  RETURN value
FUNCTION peek(queue: Queue) RETURNS INTEGER
  IF isEmpty(queue) THEN
    PRINT "Queue is empty"
    RETURN -1
  ENDIF
  RETURN queue.queue[front]
```


18	**Draw and explain Priority Queue? with application. Explain array implementation of priority queue with all basic operation**
--	

Priority Queue

- The priority queue is a data structure having a collection of elements which are associated with specific ordering.
- There are two types of priority queues –

Types of Priority Queue –

The elements in the priority queue have specific ordering. There are two types of priority queues –

1. **Ascending Priority Queue** – It is a collection of items in which the items can be inserted arbitarily but only smallest element can be removed.
2. **Descending Priority Queue** – It is a collection of items in which insertion of items can be in any order but only largest element can be removed.

- In priority queue, the elements are arranged in any order and out of which only the smallest or largest element allowed to delete each time.
- The implementation of priority queue can be done using arrays or linked list. The data structure heap is used to implement the priority queue effectively.

Application of Priority Queue –

1. The typical example of priority queue is scheduling the jobs in operating system. Typically operating system allocates priority to jobs. The jobs are placed in the queue and position1 of the job in priority queue determines their priority.
2. In network communication, to manage limited bandwidth for transmission the priority queue is used.
3. In simulation modeling, to manage the discrete events the priority queue is used.

Basic Operations:

1. **Create()** - The queue is created by declaring the data structure for it.
2. **insert()** - The element can be inserted in the queue
3. **delet()** - If the priority queue is ascending priority queue then only smallest element is deleted each time. And if the priority queue is descending priority queue then only largest element is deleted each time.

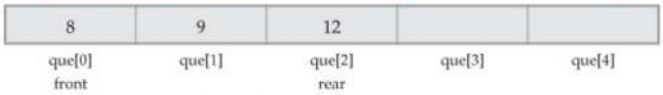
4. **Display()** - The elements of queue are displayed from front to rear.

Array implementation of priority queue –

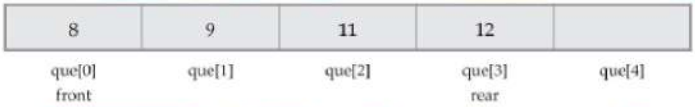
1. Insertion operation

- While implementing the priority queue we will apply a simple logic.
- That is while inserting the element we will insert the element in the array at the proper position.
- For example if the elements are placed in the queue as -

And now if an element 8 is to be inserted in the queue then it will be at 0th location as –



If the next element comes as 11 then the queue will be -

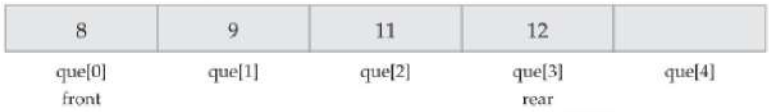


The C++ function for this operation is as given below –

```
int Pr_Q::insert(int rear, int front)
{
        int item,j;
        cout<<"\nEnter the element: ";
        cin>>item;
        if(front ==-1)
                front++;
        j=rear;
        while(j>=0 && item<que[j])
        {
        que[j+1]=que[j];
        j--;
        }
        que[j+1]=item;
        rear=rear+1;
        return rear;
}
```

2. Deletion Operation

- In the deletion operation we are simply removing the element at the front.
- For example if queue is created like this –



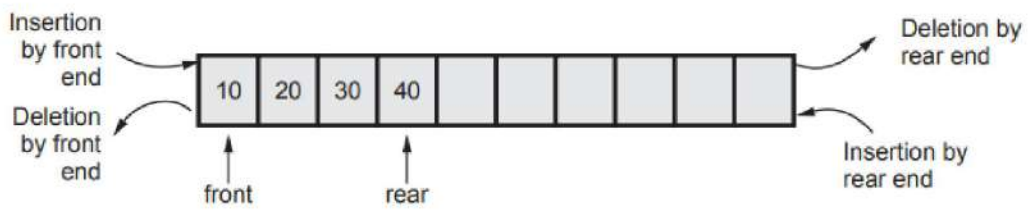
Then the element at que[0] will be deleted first

and then new front will be que[1].

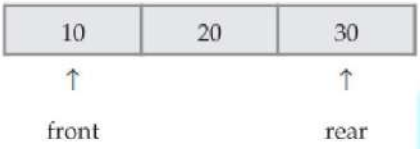
The deletion operation in C++ is as given below –

```
int Pr_Q::delet(int front)
{
        int item;
        item=que[front];
        cout<<"\n The item deleted is "<<item;
        front++;
        return front;
}
```

19	**What is Doubly Ended Queue? Draw Diagram with labelling four basic operations at appropriate places. Which two data structures are combined in it and how?**
--	**Dequeue** - Dequeue is a data structure in which we can insert the element both by front and rear end. - Similarly we can delete the element from dequeue by both the rear and front ends. - Diagram with labelling four basic operations are –

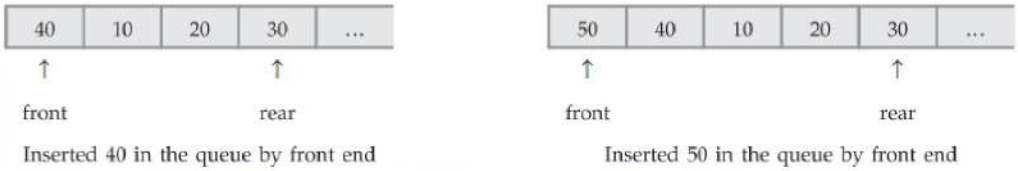


As we know, normally we insert the elements by rear end and delete the elements from front end. Let us say we have inserted the elements 10, 20, 30 by rear end.



Now if we wish to insert any element from front end then first we have to shift all the elements to the right.

For example if we want to insert 40 by front end then, the deque will be



(a) Insertion by front end

(b) Deletion by rear end

We can place -1 for the element which has to be deleted.

Combined Data Structures

- A Doubly Ended Queue (Deque) effectively combines the functionalities of two primary data structures:

1. Queue

- **Functionality**: A queue operates on a First In First Out (FIFO) principle, where elements are added to the rear and removed from the front.
- **Integration in Deque**:
  - In a deque, you can perform standard queue operations:
    - **Enqueue (Insert Rear)**: Add an element to the rear.
    - **Dequeue (Delete Front)**: Remove an element from the front.
- This allows the deque to function as a standard queue while also supporting operations at both ends.

2. Stack

- **Functionality**: A stack operates on a Last In First Out (LIFO) principle, where elements are added and removed from the same end.
- **Integration in Deque**:
  - In a deque, you can perform standard stack operations:
    - **Push (Insert Front)**: Add an element to the front.
    - **Pop (Delete Front)**: Remove an element from the front.
- This allows the deque to function as a standard stack while supporting both insertion and deletion from either end.

Q. Write pseudo C++ code to represent dequeue and perform the following operations on dequeue: i) Create ii) Insert iii) Delete iv) Display

```
#include <iostream>
using namespace std;

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

class Deque {
private:
  Node* front;
  Node* rear;

public:
  Deque() : front(nullptr), rear(nullptr) {}

  void insertFront(int value) {
    Node* newNode = new Node{value, nullptr, front};
    if (front) front->prev = newNode;
    front = newNode;
    if (!rear) rear = newNode;
  }

  void insertRear(int value) {
    Node* newNode = new Node{value, rear, nullptr};
    if (rear) rear->next = newNode;
    rear = newNode;
```

```
        if (!front) front = newNode;
    }
    void deleteFront() {
        if (!front) { cout << "Deque is empty.\n"; return; }
        Node* temp = front;
        front = front->next;
        if (front) front->prev = nullptr;
        else rear = nullptr; // Deque is empty
        delete temp;
    }
    void deleteRear() {
        if (!rear) { cout << "Deque is empty.\n"; return; }
        Node* temp = rear;
        rear = rear->prev;
        if (rear) rear->next = nullptr;
        else front = nullptr; // Deque is empty
        delete temp;
    }
    void display() {
        for (Node* curr = front; curr; curr = curr->next)
            cout << curr->data << " ";
        cout << "\n";
    }
};
int main() {
    Deque deque;
    deque.insertRear(10);
    deque.insertFront(20);
    deque.insertRear(30);
    deque.display();
    deque.deleteFront();
    deque.display();
    deque.deleteRear();
    deque.display();
    return 0;
}
```

**Note - Write code only if asked else its not compulsory wo write.*

20	**Define queue as an ADT. Write pseudo C++ code to represent queue. Comparison of Circular Queue with Linear queue**

--	**Queue as Abstract Data Type**
	- The ADT for queue is as given below –
	
	AbstractDataType Queue
	{
	Instances:
	Que[MaX] is a finite collection of elements in which insertion of element is by rear end and deletion of element is by front end.
	Precondition:
	The front and rear should be within the maximum size MAX.
	Before insertion operation, whether the queue is full or not is checked.
	Before any deletion operation, whether the queue is empty or not is checked.
	Operations:
	1. Create() - The queue is created by declaring the data structure for it.
	2. insert() - The element can be inserted in the queue by rear end.
	3. delet() - The element at front end deleted each time.
	4. Display() - The elements of queue are displayed from front to rear.
	}
	--- 	
	Pseudo C++ code to represent queue.
	#include <iostream>
	using namespace std;
	const int MAX_SIZE = 5;
	class Queue {
	private:
	int data[MAX_SIZE];
	int front, rear;
	public:
	Queue() : front(-1), rear(-1) {}
	bool isEmpty() { return front == -1; }
	bool isFull() { return (rear + 1) % MAX_SIZE == front; }
	bool enqueue(int value) {

```
        if (isFull()) return false;
        if (isEmpty()) front = 0;
        rear = (rear + 1) % MAX_SIZE;
        data[rear] = value;
        return true;
    }
    int dequeue() {
        if (isEmpty()) return -1;
        int value = data[front];
        if (front == rear) front = rear = -1;
        else front = (front + 1) % MAX_SIZE;
        return value;
    }
    int peek() {
        return isEmpty() ? -1 : data[front];
    }
};
int main() {
    Queue queue;
    queue.enqueue(10);
    queue.enqueue(20);
    cout << "Front: " << queue.peek() << endl;
    cout << "Dequeue: " << queue.dequeue() << endl;
    cout << "Front: " << queue.peek() << endl;
    return 0;
}
```


- **Instruction**
  1. For the Customized Micro PDFs Solution of All Paper contact us on telegram.
  2. Feel free to message us on Each single doubt, we will definitely help you regarding.....!!
