Assignment :- 1

① Classify data structures with Diagram.

Ans Data structures can be classified in various ways based on their properties and usage. One common classification is based on their organizational and access methods.

1. Primitive Data Structures :

- These are Basic and fundamental data structures directly operated upon by machine instructions.
- Examples : Integer, Float, character, Boolean.

2. Linear Data Structures :

- Data elements are arranged in a sequential manner where elements are accessed sequentially and each element is connected to its previous and next element.
- Examples : Arrays, linked lists, Stacks, Queues.

3. Non-linear Data Structures :

- Elements are not arranged sequentially, and each

element may have multiple predecessors and succes

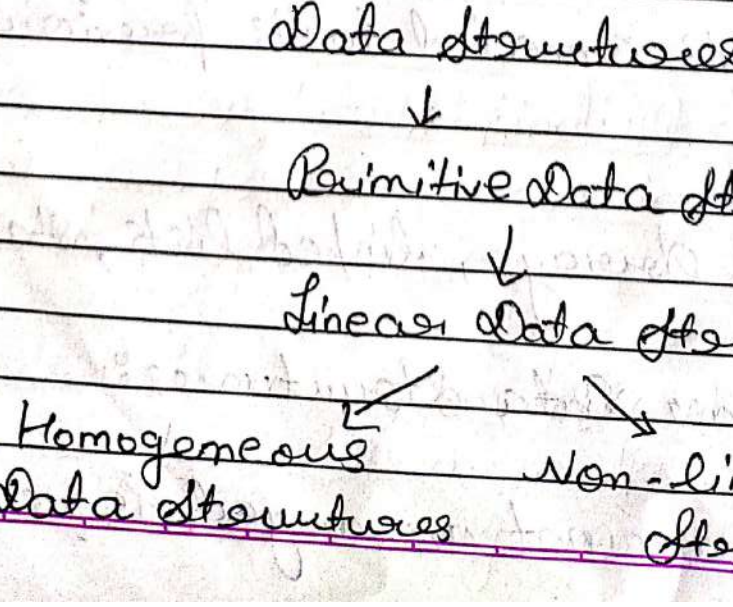
- Examples: Trees (Binary, Tree, AVL Trees, B-trees), Graphs (Directed graphs, undirected graphs)

Homogeneous Data Structures:-

- Data structures where all elements are of the same type.
- Examples: Arrays, Stacks, Queues.

5. Heterogeneous Data Structures:

- Data structures where elements may be of different types.
- Examples: Structure (Structs in C/C++, records in databases).



This diagram provides a high-level classification of data structures based on their organization and access methods. Each category can include various specific data structures that fulfill the criteria of that classification.

Que :- Interpret Big O complexity chart.

Ans :- A Big O complexity chart shows how the time or space required by an algorithm grows as the size of the input increases. It uses different curves to represent different rates of growth: constant, logarithmic, linear, quadratic etc. This helps in understanding and comparing algorithm efficiency for larger data sets.

Here's how you can interpret such a chart:

1. X-axis (Input size n)
2. Y-axis (Time or space complexity)
3. Types of curves:
   - O(1)
   - O(log n)
   - O(n)
   - O(n log n)
   - O(n^2)
   - O(2^n)
   - O(n!)

4 Interpreting the chart:
- Best case, Average case, worst case
- Comparative Analysis
- Practical implications

5. Practical use.

3. Discuss Time Complexity

Ans: Time complexity evaluates how an algorithm's runtime increases with input size n. It's expressed in Big O notation, indicating the worst-case scenario. For instance, O(1) means constant time, O(log n) logarithmic time, O(n) linear time. $O(n^2)$ quadratic time e.t.c. This analysis disregards constants and lower order terms for larger n, enabling efficient algorithm comparison and selection. understanding time complexity is crucial for optimizing software performance, ensuring applications handle larger datasets with minimal computational overhead.

4. Describe Sparse Matrix. Find the address of A[2][1]. if base address is 1024 for an integer array A[5][4] in row major order and word size is 2 byte.

Ans: A sparse matrix is a matrix in which the most of the elements are zero. In other words, it is a matrix that has a large number of zero-valued

elements. Sparse matrices are often used in scientific computing, linear algebra and machine learning to represent large matrices with a small number of non-zero elements.

## Address Calculation

Now let's calculate the address of 'A[2][1]' in the given scenario:

- Base address : 1024
- Array dimensions: A[5][4] (5 rows, 4 columns)
- Row major order (each row is stored contiguously in memory)
- word size : 2 byte (integer size)

To calculate the address, we need to calculate the offset from the base address.

In row major order, the elements of the matrix are stored as follows:

1. A[0][0], A[0][1], A[0][2], A[0][3], ---- A[1][1] ---->

The offset of 'A[2][1]' from the base address can be calculated as follows:

- Row offset: 2 (since we're in the 3rd row, indexing
- Column offset: 1 (since we're in the 2nd column,
- Total offset: (2*4)+1=9 (since each row has 4 elements, and we're in the 2nd column)

Now, multiply the total offset by the word size (2

1. offset in bytes = 9 * 2 = 18

And the offset to the base address.

11. Address of A[2][1] = 1024+18 = 1042

5. Given a two dimensional array A,(1:8, 7:14) stored in row-major order with base address 100 and size of each element is 4 bytes, find address of the element (A,(4,12).

Ans:- Address calculation:

- Base address: 100
- Array dimensions: A,(1:8, 7:14) (8 rows, 8 columns)
- Row major order
- word size: 4 bytes

To calculate the address of A,[4,12), we need to calculate the offset from the base address.

Date
Page 7

ort with 1 from 0)
ndexing starts from 0)

- Row offset: 4-1=3 (since indexing starts from 1)
- Column offset: 12-7=5 (since indexing starts from 7)

ytes):- Total offset: (3*8)+5=29 (since each row has 8 elements,

Now, multiply the total offset by the word size (4 byte

1. Offset in bytes = 29 *4=116

And the offset to the base address.

1. Address of A[4,12) = 100+116 = **216**

Therefore, the address of A[4,12) is 216.

6. Convert infix to postfix and Prefix

1. (A+B)/C-D*E

infix: (A+B)/C-D*E
Postfix: AB+C/DE*-
Prefix: -/+ABC*DE

2. P^Q^R+S/T

infix: P^Q^R+S/T
Postfix: PQ^R^T/S+
Prefix: +^^PQR/ST

3. A*B-(C/D+(E-F))^G

infix: A*B-(C/D+(E-F))^G
Postfix: AB*-+/CDF-EF^G
Prefix: -*AB-^+/CDF-EFG

Date
Page 8

7. Describe circular queue algorithm for inserting and deleting an element with diagram. Why circular queue is better than linear queue?

Ans:- Circular Queue Algorithm:- A circular Queue is a data structure that uses a single, fixed-size array to store elements. It is called "circular" because the last element of the array is connected to the first element, forming a circle.

Insertion Algorithm:-

1. Check if the queue is full by checking if '(rear+1) % max_size == front'. if true, the queue is full and insertion is not possible.
2. Calculate the new rear index: 'rear = (rear+1) % m
3. Store the new element at the calculated rear index 'queue[rear] = element'.

Deletion algorithm:-

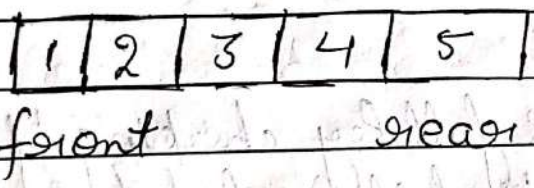
1. Check if the queue is empty by checking if 'front == rear'. if true, the queue is empty and deletion is not possible.

2. Retrieve the element at the front index: 'element = queue[front]'.

3. Calculate the new front index: 'front = (front + 1) % max_size'.

Diagram:-

Here is a diagram illustrating a circular queue with 5 elements:




In this example, the front index is 0 and the rear index is 4. The queue is not full and the next insertion would occur at index 0 (wrapping around to the beginning of the array).

Why circular Queue is better than linear Queue:

- Efficient use of Memory
- Reduced overhead
- Improved performance
- Simplified implementation.

8. Discuss Push and Pop algorithm of Stack.

Ans:- A Stack is a Last-in-first-out (LIFO) data structure that follows a specific order for adding and removing elements. The two primary operations in a Stack are:

1. Push: Adding an element to the top of the Stack.
2. Pop: Removing the top element from the Stack.

Push Algorithm:

1. check if the Stack is full by checking if 'top == max-1'. If true, the Stack is full and pushing is not possible.
2. Increment the 'top' index 'top = top+1'.
3. Store the new element at the incremented 'top' index 'Stack[top] = element'.

Pop Algorithm:

1. check if the Stack is empty by checking if 'top == -1'. If true, the Stack is empty and popping is not possible.
2. Retrieve the top element: 'element = Stack[top]'.

3. Decrement the 'top' index: 'top = top - 1'.

Code :-

```
// Push algorithm

Procedure Push(stack, element)
    if top = max_size - 1 then
        error "stack is full"
    else
        top = top + 1
        stack[top] = element

ize-1'.

// Pop Algorithm

Procedure pop(stack)
    if top = -1 then
        error "stack is empty"
    else
        element = stack[top]
        top = top - 1
        return element
```

Example :-

Initial stack: '[]' (empty stack)

Push: 'A': 'top = 0' 'stack[0] = A' ⇒ '[A]'
Push 'B': 'top = 1'; 'stack[1] = B' ⇒ '[A,B]'
Pop: 'element = B', 'top = 0' ⇒ '[A]'

9. List applications of stack and convert 2*3/(2-1)+5*3 infix expression into postfix format showing stack status after every step in tabular form and evaluate that postfix notation.

Ans: Applications of stack:

1. Expression Evaluation: Stacks are used to convert infix expressions into postfix (or prefix) notation for easier evaluation.

2. Function call management: Stack manage function calls in languages that support recursion, storing return addresses and local variables.

3. Syntax Parsing: Stacks are used in compilers to parse and evaluate expressions, ensuring correct syntax and precedence.

4. Backtracking: Stacks are used in backtracking algorithms, like depth-first search, to keep of the current state and backtrack when necessary.

5. Undo Mechanisms: Stacks can be used to implement undo mechanisms in applications by storing previous states.

Conversion of infix to Postfix (RPN):

Given: infix expression: '2*3/(2-1)+5*3'.

Step-by-step conversion to convert this to postfix notation:

| Token | Stack | Output |
|---|---|---|
| 2 | [] | 2 |
| * | [*] | 2 |
| 3 | [*] | 2 3 |
| / | [*, /] | 2 3 |
| ( | [*, /, (] | 2 3 |
| 2 | [*, /, (] | 2 3 2 |
| - | [*, /, (, -] | 2 3 2 |
| 1 | [*, /, (, -] | 2 3 2 1 |
| ) | [*, /] | 2 3 2 1 |
| + | [*, /, +] | 2 3 2 1 |
| 5 | [*, /, +] | 2 3 2 1 5 |
| * | [*, /, +, *] | 2 3 2 1 5 |
| 3 | [*, /, +, *] | 2 3 2 1 5 3 |

The resulting postfix expression is: '23*21-/53*+'

## Evaluating the Postfix Expression:

To evaluate the postfix expression, we will use a stack to keep track of operands. We will iterate through the postfix expression from left to right, and for each token, we will perform the following actions:

1. If the token is an operand, push it onto the stack.
2. If the token is an operator, pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.

Here is a step by step evaluation process:

| Token | Stack |
|---|---|
| 2 | [2] |
| 3 | [2, 3] |
| * | [6] |
| 2 | [6, 2] |
| 1 | [6, 2, 1] |
| - | [6, 1] |
| 1 | [6] |
| 5 | [6, 5] |
| 3 | [6, 5, 3] |
| * | [6, 15] |
| + | [21] |

The final result is: '21'.

10. Transform the following expression to postfix and evaluate postfix expression by assuming A=1, B=2, C=3, D=4, E=6, F=6, G=1, I=3 and J=3

A+B-C*D/E+F*G/(I+J)

Ans Step 1 :- Convert infix to Postfix (RPN)

| Token | Action | stack | Postfix output |
|---|---|---|---|
| A | Add to output | A | |
| + | Push to stack | + | A |
| B | Add to output | + | AB |
| - | Push to stack | - | AB+ |
| C | Add to output | - | AB+C |
| * | Push to stack | -* | AB+C |
| D | Add to output | -* | AB+CD |
| / | Pop stack (higher precedence) | -/ | AB+CD* |
| E | Add to output | -/ | AB+CD*E |
| + | Push to stack | -+ | AB+CD*E |
| F | Add to output | -+ | AB+CD*EF |
| * | Push to stack | -+* | AB+CD*EF |
| G | Add to output | -+* | AB+CD*EFG |
| / | Pop stack (higher precedence) | -+/ | AB+CD*EFG* |
| ( | Push to stack | -+/( | AB+CD*EFG* |
| I | Add to output | -+/( | AB+CD*EFG*I |
| + | Push to stack | -+/(+ | AB+CD*EFG*I |
| J | Add to output | -+/(+ | AB+CD*EFG*IJ |
| ) | Pop stack until '(' is found | -+ | AB+CD*EFG*IJ+ |

After processing all tokens, pop all remaining operators from the stack to the output.

So, the postfix notation of 'A+B-C*D/E+F*G/(I+J)' is 'AB+CD*E/FFG*IJ+/+'.

Step 2: Evaluate the postfix expression

Now, let's evaluate the postfix expression 'AB+CD*E/FFG*IJ+/+' using the given variable assignments.

A=1, B=2, C=3, D=4, E=6, F=6, G=1, I=3, J=3

| Token | Action | Stack |
|---|---|---|
| A | Push 1 | 1 |
| B | Push 2 | 12 |
| + | Pop 2, Pop 1, calculate 1+2 | 3 |
| C | Push 3 | 33 |
| D | Push 4 | 34 |
| * | Pop 4, Pop 3, calculate 3*4 | 12 |
| E | Push 6 | 126 |
| / | Pop 6, Pop 12, calculate 12/6 | 2 |
| + | Pop 2, Pop 2, calculate 2+2 | 4 |
| F | Push 6 | 46 |
| G | Push 1 | 461 |
| * | Pop 1, Pop 6, calculate 6*1 | 46 |
| I | Push 3 | 463 |
| J | Push 3 | 4633 |

| | | |
|---|---|---|
| + | Pop 3, Pop 3, calculate 3+3 | 4 6 6 |
| / | Pop 6, Pop 4, calculate 6/4 | 1 |
| + | Pop 1, Pop 1, calculate 1+1 | 2 |

The evaluated value of the postfix expression 'A+B-C*D/E+F*G/(I+J)' is '2'.

12. The initial configuration of the queue is having the elements 'x, y, z, a' forming a queue with size 4 to get the new configuration 'a, z, y, x' one needs a minimum of

a) 3 additions and 4 deletion.
b) 3 deletions and 4 additions.
c) 3 deletions and 3 additions.
d) 4 deletions and 4 additions.

Ans To transform the initial configuration 'x, y, z, a' into the desired configuration 'a, z, y, x' using operations on a queue, we need to consider the operations of addition (enqueue) and deletion (dequeue).

Here's the step-by-step reasoning:

1. Initial Queue configuration: x, y, z, a (front to rear)
2. Desired Queue configuration: a, z, y, x (front to rear)

To achieve the desired configuration:

- 'a' needs to be at the front of the queue.
- 'z' needs to follow 'a'.
- 'y' needs to follow 'z'.
- 'x' needs to be at the rear of the queue.

Operation Needed:

- Deletion (Dequeues): we need to remove elements from the front of the initial queue until we reach 'a' at the front. This requires 3 deletions (to remove 'x', 'y', 'z').
- Additions (Enqueues): After positioning 'a' at the front, we need to add 'z', 'y' and 'x' to the rear of the queue. This requires 3 additions.

Conclusion:

- Therefore, the minimum operations required are:
  - 3 deletions (to remove 'x', 'y', 'z')
  - 3 additions (to add 'z', 'y', 'x')

Hence the correct answer is C) 3 deletions and 3 additions

Short Questions:-

1. Define Dynamic memory allocation?

Ans: Dynamic memory allocation allows programs to request and release memory during execution using functions like 'malloc', 'calloc', 'realloc' and 'free'. It's essential for managing varying data sizes and structures efficiently, preventing memory leaks and optimizing resource usage in languages like C and C++.

2. Define referential structure?

Ans: A referential structure defines relationships between elements by referencing memory addresses or pointers, rather than embedding data directly. It's integral to dynamic data structures like linked lists and trees, enabling efficient traversal and manipulation of interconnected data elements in programming.

3. Array is a heterogeneous data types. (True/False). Justify your answer.

Ans:- False. Arrays are homogeneous data structures in most programming languages, meaning they store elements of the same type. Each element occupies contiguous memory allocation/locations, facilitating

efficient acess using indices. Heterogeneous collections can bee simulated using structures or classes in languages like C or C++.

4. A m*n matrix which contains very few non-zero elements. A matrix contains more number of zero values than Non-zero values. Such matrix is known as?

Ans:- Such a matrix is known as Sparse matrix. Sparse matrices have a majority of their elements as zeros. They are efficiently stored and manipulated using specialized data structures or techniques that focus only on the non-zero elements, optimizing memory usage and computational efficiency for operations like matrix multiplication and addition.

5. Insert an element in array at index k will take how much time?

Ans b) O(n-k).

6. State True / False: Arrays have better cache locality that can make them better in terms of performance.

Ans:- Issue: Arrays generally have better cache locality compared to other data structures like linked lists.

7. Differentiate between LIFO and FIFO access mechanism.

Ans:- LIFO (Last in, first out) stacks use a push-pop method, while FIFO (first in, first out) queues use an enqueue - dequeue method, affecting item retrieval order: last versus first placed.

8. How linked list is better compared to stack, queue, and array? Explain with concept of dynamic memory allocation.

Ans:- Linked lists are advantageous over stack, queues and arrays due to dynamic memory allocation, allowing efficient insertion, deletion and size flexibility, unlike arrays with fixed size and stack, queue implementations with bounded structures.

9. In which type of scenario, linear queue (simple queue) is better than circular queue?

Ans A linear queue (simple queue) is preferable over a circular queue when memory efficiency and sequential processing without circular wrap-around are prioritized. It's suitable for scenarios with fixed or predictable queues sizes where straight

forward implementation and maintenance are beneficial.

10. After evaluation of 3, 5, 4, *, + result is ?

Ans :- (1.) Push operands onto the stack as they appear.

- Push '3'
- Push '5'
- Push '4'

Stack '3, 5, 4'

(2.) Encounter the '*' operator:

- Pop '4' and '5' from the stack.
- Calculate '4 * 5' = 20.
- Push the result ('20') back onto the stack

Stack '3, 20'.

(3.) Encounter the '+' operator.

- Pop '20' and '3' from the stack.
- Calculate '20 + 3' = 23'.
- Push the result ('23') back onto the stack.

(4) After processing all tokens, the stack contains the final result, which is '23'.

Ans, result is '23'.

(11) what will be the value of front and Rear pointers when Queue is Empty?

Ans When a Queue is empty:

- Front Pointer: Typically set to '-1' or 'Null'.
- Rear Pointer: Also set to '-1' or 'Null'.

These values indicate there are no elements in the queue, helping manage enqueue and dequeue operations effectively.

## MCQ Questions (Unit-1)

(1) A program P reads in 500 integers in the range [0..100] representing the score of 500 students. It then print the frequency of each score above 50. what would be the best way P to store the frequencies

Ans An array of 100 numbers (✓)

(2.) Consider the following c program.

```
#include <stdio.h>
int main() {
    int a[4][5] = {(1,2,3,4,5)},
                  {6,7,8,9,10},
```

```
{11,12,13,14,15},
{16,17,18,19,20};
printf("%d\n", *(*(a+**a+2)+3));
return(0);
}
```

The output of the program is (19)

3. If Array A is made to hold the string "abcde", which of the below four test cases will be successful in exposing the flaw in this procedure?

(a) None (b) 2 only (c) 3 and 4 only (d) 4 only

4. Consider the following Pseudo code.

```
function Example (n: integer)
    sum := 0
    for i := 1 to n do
        for j := 1 to i do
            sum := sum + 1
    return sum
```

5. What is the time complexity of the above Pseudo code?

Ans $O(n^2)$ (✓)

## MCQ Questions (unit 2)

1. Which one of the following is an application of Stack data structure?

Ans:- All of the above (✓)

2. Consider the following sequence of operations on an initially empty stack:

1. Push(10):- The stack is initially empty so 10 is pushed onto the stack. The stack now looks like this: [10].

2. Push(20):- 20 is pushed onto the stack. The stack now looks like this: [10,20].

3. Push(30):- 30 is pushed onto the stack. The stack now look like this: [10,20,30].

4. Pop():- The top element 30 is popped from the stack. The stack is now look like this [10,20].

5. Push(40):- 40 is pushed onto the stack. The stack now look like this [10,20,40].

6. Pop():- The top element 40 is popped from the stack. The stack now look like this [10,20].

7. pop(): The top element 20 is popped from the stack. The stack now looks like this [10]

8. pop(): The top element 10 is popped from the stack. The stack now looks like empty: [ ].

what is the sequence of values popped from the stack?

Ans 30, 40, 20, 10 ✓

3. what is the time complexity of an infix to postfix conversion algorithm?

Ans: O(N) ✓

4. The recurrence relation capturing the optimal execution time of the towers of Hanoi problem with n discs is

Ans. $T(n) = 2T(n-1) + 1$ (✓)

5. which one of the following is an application of stack data structure?

Ans All of the above (✓)

6. Consider a linear queue implemented using an array Q of size N. The queue uses two variables front and rear to keep track of the front and rear positions of the queue respectively. Initially the queue is empty with front = -1 and rear = -1.

Which of the following conditions correctly checks if the queue is full?

(a) rear == N-1
(b) front == N-1
(c) (rear+1)%N == front ✓
(d) rear == front

7. In circular queue, to manage rear pointer, which formula is used? where rear is a pointer to insert an element and N is size of queue.

Ans rear = (rear+1)%N (✓)

8. What is the main advantage of using a priority queue?

Ans :- Fast access to the largest (or smallest) element (✓).