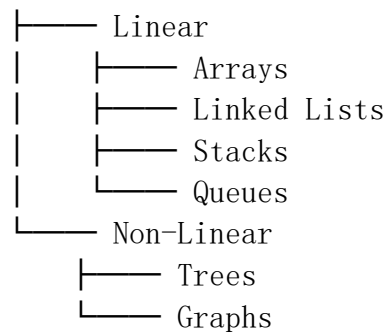
1- What is a data structure? Explain its importance.

A data structure is a specialized format for organizing, processing, retrieving, and storing data. It allows a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. The importance of data structures lies in their ability to efficiently manage large amounts of data for applications such as databases, internet indexing services, and large-scale simulations. Efficient data structures are key to designing efficient algorithms.

2- Differentiate between linear and non-linear data structures.

- **Linear Data Structures:** In linear data structures, elements are arranged sequentially and each element is connected to its previous and next element. Examples include arrays, linked lists, stacks, and queues.
- **Non-linear Data Structures:** In non-linear data structures, elements are not arranged sequentially. Each element can be connected to multiple elements, representing a hierarchical relationship. Examples include trees and graphs.



3- What are the primary operations that can be performed on a stack?

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove and return the top element of the stack.
- **Peek/Top:** Return the top element of the stack without removing it.
- **isEmpty:** Check if the stack is empty.
- **isFull:** Check if the stack is full (in case of a fixed-size stack).

Code:-

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

int stack[MAX];
int top = -1;

void push(int x) {
    if (top == MAX - 1) {
        printf("Stack overflow\n");
```

```
        return;
    }
    stack[++top] = x;
}

int pop() {
    if (top == -1) {
        printf("Stack underflow\n");
        return -1;
    }
    return stack[top--];
}

int peek() {
    if (top == -1) {
        printf("Stack is empty\n");
        return -1;
    }
    return stack[top];
}

int isEmpty() {
    return top == -1;
}
```

4- Describe the difference between a stack and a queue.

- **Stack:** Follows Last In, First Out (LIFO) principle. Elements are added and removed from the top.
- **Queue:** Follows First In, First Out (FIFO) principle. Elements are added from the rear and removed from the front.

Stack code is given in above question

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

int queue[MAX];
int front = -1, rear = -1;

void enqueue(int x) {
    if ((rear + 1) % MAX == front) {
        printf("Queue overflow\n");
        return;
    }
```

```
    if (front == -1) front = 0;
    rear = (rear + 1) % MAX;
    queue[rear] = x;
}

int dequeue() {
    if (front == -1) {
        printf("Queue underflow\n");
        return -1;
    }
    int item = queue[front];
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % MAX;
    }
    return item;
}

int peek() {
    if (front == -1) {
        printf("Queue is empty\n");
        return -1;
    }
    return queue[front];
}

int isEmpty() {
    return front == -1;
}
```

5-How does a circular queue differ from a regular queue?

In a circular queue, the last position is connected back to the first position to make a circle. This helps in efficiently utilizing the space by allowing the queue to reuse the empty spaces once elements are dequeued. A regular queue does not have this circular property, leading to potential wastage of space.

Queue code is given in the question no. 4

Circular Queue

```
#include <stdio.h>

# define max 6
int queue[max];  // array declaration
int front=-1;
```

```
int rear=-1;
// function to insert an element in a circular queue
void enqueue(int element)
{
   if(front==-1 && rear==-1)   // condition to check queue is empty
   {
      front=0;
      rear=0;
      queue[rear]=element;
   }
   else if((rear+1)%max==front)  // condition to check queue is full
   {
      printf("Queue is overflow..");
   }
   else
   {
      rear=(rear+1)%max;       // rear is incremented
      queue[rear]=element;     // assigning a value to the queue at the rear position.
   }
}

// function to delete the element from the queue
int dequeue()
{
   if((front==-1) && (rear==-1))  // condition to check queue is empty
   {
      printf("\nQueue is underflow..");
   }
 else if(front==rear)
{
   printf("\nThe dequeued element is %d", queue[front]);
   front=-1;
   rear=-1;
}
else
{
    printf("\nThe dequeued element is %d", queue[front]);
   front=(front+1)%max;
}
}
// function to display the elements of a queue
void display()
{
   int i=front;
   if(front==-1 && rear==-1)
```

```
51.             {
52.                 printf("\n Queue is empty..");
53.             }
54.             else
55.             {
56.                 printf("\nElements in a Queue are :");
57.                 while(i<=rear)
58.                 {
59.                     printf("%d,", queue[i]);
60.                     i=(i+1)%max;
61.                 }
62.             }
63.         }
64.         int main()
65.         {
66.             int choice=1,x;   // variables declaration
67.
68.             while(choice<4 && choice!=0)   // while loop
69.             {
70.             printf("\n Press 1: Insert an element");
71.             printf("\nPress 2: Delete an element");
72.             printf("\nPress 3: Display the element");
73.             printf("\nEnter your choice");
74.             scanf("%d", &choice);
75.
76.             switch(choice)
77.             {
78.
79.                 case 1:
80.
81.                 printf("Enter the element which is to be inserted");
82.                 scanf("%d", &x);
83.                 enqueue(x);
84.                 break;
85.                 case 2:
86.                 dequeue();
87.                 break;
88.                 case 3:
89.                 display();
90.
91.             }}
92.             return 0;
93.         }
```

6-Explain the concept of a linked list. How is it different from an array?

A linked list is a linear data structure where each element (node) contains a data part and a reference (or link) to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory locations, making them more flexible in terms of memory usage. Arrays have a fixed size and provide O(1) time complexity for accessing elements, while linked lists have dynamic sizes and O(n) time complexity for accessing elements.

- 7- **Difference between array and linked list**

- **Answer:**

Memory Allocation: Arrays use contiguous memory allocation, while linked lists use non-contiguous memory.

- **Size:** Arrays have a fixed size, whereas linked lists can grow and shrink dynamically.
- **Access Time:** Arrays provide O(1) access time, while linked lists provide O(n) access time.
- **Insertion/Deletion:** In arrays, insertion and deletion operations are expensive (O(n)), whereas in linked lists, these operations are more efficient (O(1) for insertion at the head).

8- What are the main operations of a queue? Describe their time complexities.

- **Enqueue:** Add an element to the rear. O(1)
- **Dequeue:** Remove an element from the front. O(1)
- **Peek/Front:** Get the front element without removing it. O(1)
- **isEmpty:** Check if the queue is empty. O(1)

Queue code is given in the question no. 4

9-What are the main operations of a stack?

Answer:

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove and return the top element of the stack.
- **Peek/Top:** Return the top element of the stack without removing it.
- **isEmpty:** Check if the stack is empty.
- **isFull:** Check if the stack is full (in case of a fixed-size stack)

(Code provided in question 3)

9- Describe the algorithm for evaluating postfix expressions using a stack

- Initialize an empty stack.
- Scan the postfix expression from left to right.
- If the scanned character is an operand, push it to the stack.
- If the scanned character is an operator, pop two operands from the stack, perform the operation, and push the result back to the stack.
- At the end, the stack will contain the result of the postfix expression.

Code-:

```
#include <stdio.h>

#include <ctype.h>


int evaluatePostfix(char* exp) {

    int stack[MAX];

    int top = -1;


    for (int i = 0; exp[i]; ++i) {

        if (isdigit(exp[i])) {

            stack[++top] = exp[i] - '0';

        } else {

            int val1 = stack[top--];

            int val2 = stack[top--];

            switch (exp[i]) {

                case '+': stack[++top] = val2 + val1; break;

                case '-': stack[++top] = val2 - val1; break;

                case '*': stack[++top] = val2 * val1; break;
```

```
                case '/': stack[++top] = val2 / val1; break;
            }
        }
    }
    return stack[top];
}
```

11-Explain the concept of a singly linked list and provide an example.

A singly linked list is a linear data structure where each element (node) points to the next node in the sequence. It consists of a head pointer pointing to the first node, and each node contains data and a reference to the next node.

(Code provided in question 6)

12-How does a doubly linked list differ from a singly linked list?

A doubly linked list is similar to a singly linked list but with an additional pointer in each node that points to the previous node, allowing traversal in both directions.

Code-:

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

struct Node* head = NULL;

void insert(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
```

```
    newNode->next = head;
    newNode->prev = NULL;
    if (head != NULL) {
        head->prev = newNode;
    }
    head = newNode;
}

void printList() {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

13-What is a circular linked list, and how is it implemented?

A circular linked list is a linked list where the last node points to the first node, forming a circle. It can be singly or doubly linked.

Code-:

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

```
struct Node* head = NULL;

void insert(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = head;

    if (head == NULL) {
        newNode->next = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    head = newNode;
}

void printList() {
    if (head != NULL) {
        struct Node* temp = head;
        do {
            printf("%d -> ", temp->data);
```

```
            temp = temp->next;
        } while (temp != head);
        printf("HEAD\n");
    }
}
```

13-What is linear search, and what is its time complexity?

Linear search is a simple search algorithm that checks each element of the array sequentially until the desired element is found or the array ends. Its time complexity is O(n).

```
int linearSearch(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x)
            return i;
    }
    return -1;
}
```

15-Explain binary search. How does it improve search efficiency compared to linear search?

Binary search is an efficient algorithm for finding an element in a sorted array by repeatedly dividing the search interval in half. Its time complexity is O(log n), making it much faster than linear search for large arrays.

```
int binarySearch(int arr[], int l, int r, int x) {
  while (l <= r) {
    int m = l + (r - l) / 2;
    if (arr[m] == x)
```

```
            return m;
        if (arr[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }
    return -1;
}
```

16-How does selection sort work?

Selection sort is a simple comparison-based sorting algorithm. It works by repeatedly finding the minimum element from the unsorted part of the array and swapping it with the first unsorted element. This process continues until the array is sorted. Its time complexity is O(n^2).

```
void selectionSort(int arr[], int n) {
    int i, j, min_idx;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

17-Classify data structures with diagram

Data structures can be classified into:

- **Primitive Data Structures:** int, char, float, double, etc.
- **Non-Primitive Data Structures:** Arrays, Linked Lists, Stacks, Queues, Trees, Graphs, Hash Tables, etc.
- Data Structures
- ├── Primitive
- │ ├── int
- │ ├── char
- │ ├── float
- │ └── double
- └── Non-Primitive
- ├── Arrays
- ├── Linked Lists
- ├── Stacks
- ├── Queues
- ├── Trees
- ├── Graphs
- └── Hash Tables

Draw Digram also

18-Interpret Big O complexity chart.

Big O complexity chart helps in understanding the time and space efficiency of algorithms. Common complexities include:

- **O(1):** Constant time
- **O(log n):** Logarithmic time
- **O(n):** Linear time
- **O(n log n):** Linearithmic time
- **O(n^2):** Quadratic time
- **O(2^n):** Exponential time
- **O(n!):** Factorial time

19-Discuss Time complexity

Time complexity is a measure of the amount of time an algorithm takes to process as a function of the input size. It helps in analyzing the efficiency of an algorithm and predicting its performance.

20-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.

A sparse matrix is a matrix with a majority of its elements being zero.

Address calculation:

- Base address = 1024
- Row-major order: Address of A[i][j] = Base_address + (i * number_of_columns + j) * element_size
- Number of columns = 4

- Element size = 2 bytes

Address of A[2][1] = 1024 + (2 * 4 + 1) * 2 = 1024 + 18 = 1040

20-Given a two-dimensional array A1(1:8, 7:14) stored in row-major order with base address 100 and size of each element is 4 bytes, find the address of the element A1(4, 12).

- Base address = 100
- Row-major order: Address of A[i][j] = Base_address + [(i - lower_bound_row) * number_of_columns + (j - lower_bound_col)] * element_size
- Lower bound for rows = 1
- Lower bound for columns = 7
- Number of columns = 14 - 7 + 1 = 8
- Element size = 4 bytes

Address of A1[4][12] = 100 + [(4 - 1) * 8 + (12 - 7)] * 4 = 100 + [3 * 8 + 5] * 4 = 100 + 41 * 4 = 100 + 164 = 264

22-Define dynamic memory allocation?

Answer: Dynamic memory allocation is the process of allocating memory during the runtime of the program using functions such as `malloc()`, `calloc()`, `realloc()`, and `free()` in C.

23-Define referential structure?

A referential structure is a data structure where elements refer to other elements through pointers or references. Examples include linked lists, trees, and graphs.

1. 24-**Array is a heterogeneous data type. (True/False). Justify your answer.**

Answer: False. An array is a homogeneous data type, meaning all elements in an array must be of the same type. This ensures uniformity and allows for efficient memory allocation and access.

25-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 ?

Such a matrix is known as a **sparse matrix**. Sparse matrices are used to save memory and computational time when dealing with matrices that have a high proportion of zero elements.

1. 26-**Convert infix to Postfix and Prefix**

Answer:

(A + B) / C – D * E

1. Postfix: AB + C / DE * -
2. Prefix: - / + A B C * D E

P ^ Q ^ R + S / T

1. Postfix: P Q ^ R ^ S T / +
2. Prefix: + ^ P ^ Q R / S T

A * B – (C / D + (E – F)) ^ G

1. Postfix: AB * CD / EF - + G ^ -
2. Prefix: - * A B ^ + / C D - E F G

List applications of stack and Convert 2 * 3 / (2-1) + 5 * 3 infix expression into postfix format. Showing stacks status after every step in tabular form and evaluates that postfix notation.

Answer: Applications of Stack:

- Expression evaluation and conversion (infix, postfix, prefix)
- Syntax parsing
- Backtracking algorithms
- Function call management in recursion
- Undo mechanisms in text editors

Conversion to Postfix:

- Infix: 2 * 3 / (2 - 1) + 5 * 3
- Postfix: 2 3 * 2 1 - / 5 3 * +

Stack status for each step:

Step	Stack	Output
2	2	
*	* 2	

Step	Stack	Output
3	*	2 3
/	/	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
End		2 3 * 2 1 - / 5 3 * +

Evaluation of Postfix:

- 2 3 * = 6
- 6 2 1 - / = 6 1 /
- 6 / 1 = 6
- 6 5 3 * + = 6 15 +
- 6 + 15 = 21

So, the value of the expression is 21.

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

Sure, here are detailed answers to the questions from the question bank:

1.

What is a data structure? Explain its importance.

Answer: A data structure is a specialized format for organizing, processing, retrieving, and storing data. It allows a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. The importance of data structures lies in their ability to efficiently manage large amounts of data for applications such as databases, internet indexing services, and large-scale simulations. Efficient data structures are key to designing efficient algorithms.

Differentiate between linear and non-linear data structures.

Answer:

1. **Linear Data Structures:** In linear data structures, elements are arranged sequentially and each element is connected to its previous and next element. Examples include arrays, linked lists, stacks, and queues.
2. **Non-linear Data Structures:** In non-linear data structures, elements are not arranged sequentially. Each element can be connected to multiple elements, representing a hierarchical relationship. Examples include trees and graphs.

What are the primary operations that can be performed on a stack?

Answer:

1. **Push:** Add an element to the top of the stack.
2. **Pop:** Remove and return the top element of the stack.
3. **Peek/Top:** Return the top element of the stack without removing it.
4. **isEmpty:** Check if the stack is empty.
5. **isFull:** Check if the stack is full (in case of a fixed-size stack).

    2.

Describe the difference between a stack and a queue.

Answer:

1. **Stack:** Follows Last In, First Out (LIFO) principle. Elements are added and removed from the top.
2. **Queue:** Follows First In, First Out (FIFO) principle. Elements are added from the rear and removed from the front.

How does a circular queue differ from a regular queue?

Answer: In a circular queue, the last position is connected back to the first position to make a circle. This helps in efficiently utilizing the space by allowing the queue to reuse the empty spaces once elements are dequeued. A regular queue does not have this circular property, leading to potential wastage of space.

Explain the concept of a linked list. How is it different from an array?

Answer: A linked list is a linear data structure where each element (node) contains a data part and a reference (or link) to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory locations, making them more flexible in terms of memory usage. Arrays have a fixed size and provide O(1) time complexity for accessing elements, while linked lists have dynamic sizes and O(n) time complexity for accessing elements.

Difference between array and linked list

Answer:

1. **Memory Allocation:** Arrays use contiguous memory allocation, while linked lists use non-contiguous memory.
2. **Size:** Arrays have a fixed size, whereas linked lists can grow and shrink dynamically.
3. **Access Time:** Arrays provide O(1) access time, while linked lists provide O(n) access time.
4. **Insertion/Deletion:** In arrays, insertion and deletion operations are expensive (O(n)), whereas in linked lists, these operations are more efficient (O(1) for insertion at the head).

What are the main operations of a queue? Describe their time complexities.

Answer:

1. **Enqueue:** Add an element to the rear of the queue (O(1)).
2. **Dequeue:** Remove and return the front element of the queue (O(1)).
3. **Front/Peek:** Return the front element without removing it (O(1)).
4. **isEmpty:** Check if the queue is empty (O(1)).
5. **isFull:** Check if the queue is full (in case of a fixed-size queue) (O(1)).

What are the main operations of a stack?

Answer:

1. **Push:** Add an element to the top of the stack.
2. **Pop:** Remove and return the top element of the stack.
3. **Peek/Top:** Return the top element of the stack without removing it.
4. **isEmpty:** Check if the stack is empty.
5. **isFull:** Check if the stack is full (in case of a fixed-size stack).

Describe the algorithm for evaluating postfix expressions using a stack

Answer:

Create an empty stack.

Read the postfix expression from left to right.

For each character:

1. If it is an operand, push it onto the stack.
2. If it is an operator, pop the top two elements from the stack, apply the operator, and push the result back onto the stack.

After the entire expression has been processed, the value remaining in the stack is the result of the postfix expression.

Explain the concept of a singly linked list and provide an example.

Answer: A singly linked list is a linear data structure where each node contains a data part and a reference to the next node in the sequence. The last node points to null, indicating the end of the list.

Example:

```
Node1 (data=1) -> Node2 (data=2) -> Node3 (data=3) -> null
```

1. **How does a doubly linked list differ from a singly linked list?**

Answer: A doubly linked list contains an extra reference in each node to the previous node, in addition to the reference to the next node. This allows traversal in both directions (forward and backward).

Example:

```
null <- Node1 (data=1) <-> Node2 (data=2) <-> Node3 (data=3) -> null
```

1. **What is a circular linked list, and how is it implemented?**

Answer: In a circular linked list, the last node points back to the first node instead of null, forming a circle. This allows for continuous traversal without reaching an end.

Implementation:

```
Node1 (data=1) -> Node2 (data=2) -> Node3 (data=3) -> Node1
```

1. **What is linear search, and what is its time complexity?**

Answer: Linear search is a simple search algorithm that checks each element of the list sequentially until the desired element is found or the list ends. Its time complexity is O(n), where n is the number of elements in the list.

1. **Explain binary search. How does it improve search efficiency compared to linear search?**

Answer: Binary search is an efficient algorithm for finding an element in a sorted list. It works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeat until the value is found or the interval is empty. Its time complexity is O(log n), which is much faster than linear search for large datasets.

1. **How does selection sort work?**

Answer: Selection sort is a simple comparison-based sorting algorithm. It works by repeatedly finding the minimum element from the unsorted part of the array and swapping it with the first unsorted element. This process continues until the array is sorted. Its time complexity is O(n^2).

1. **Classify data structures with diagram.**

Answer: Data structures can be classified as follows:

- **Linear Data Structures:**
    - Arrays
    - Linked Lists
    - Stacks
    - Queues

- **Non-Linear Data Structures:**
  - Trees
  - Graphs

Diagram:

markdown
Copy code
Data Structures
├── Linear
│ ├── Arrays
│ ├── Linked Lists
│ ├── Stacks
│ └── Queues
└── Non-Linear
├── Trees
└── Graphs

1. **Interpret Big O complexity chart.**

Answer: Big O notation is used to describe the performance or complexity of an algorithm. It specifically describes the worst-case scenario and helps to understand the upper limits of algorithm performance. Common complexities include:

- O(1): Constant time
- O(log n): Logarithmic time
- O(n): Linear time
- O(n log n): Linearithmic time
- O(n^2): Quadratic time
- O(2^n): Exponential time
- O(n!): Factorial time

1. **Discuss Time complexity**

Answer: Time complexity is a measure of the amount of computational time that an algorithm takes to complete as a function of the length of the input. It helps in analyzing the efficiency of an algorithm and is typically expressed using Big O notation. Understanding time complexity is crucial for optimizing code and ensuring that algorithms run efficiently even with large inputs.

1. **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 bytes.**

Answer: A sparse matrix is a matrix in which most of the elements are zero. It is typically used to save memory when the matrix has a large number of zero elements.

To find the address of A[2][1] in a row-major order array:

- Formula: Address = Base Address + (i * number_of_columns + j) * element_size
- Here, i = 2, j = 1, number_of_columns = 4, element_size = 2 bytes, base_address = 1024.

- Address = 1024 + (2 * 4 + 1) * 2 = 1024 + 9 * 2 = 1024 + 18 = 1042.

So, the address of A[2][1] is 1042.

1. **Given a two dimensional array A1(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 A1(4, 12).**

Answer: To find the address of A1(4, 12) in a row-major order array with non-zero lower bounds:

- Convert indices to zero-based: A1(4, 12) -> A1(3, 5)
- Formula: Address = Base Address + [(i - lower_row_bound) * number_of_columns + (j - lower_column_bound)] * element_size
- Here, i = 4, j = 12, lower_row_bound = 1, lower_column_bound = 7, number_of_columns = 8, element_size = 4 bytes, base_address = 100.
- Address = 100 + [(4 - 1) * 8 + (12 - 7)] * 4
- Address = 100 + [3 * 8 + 5] * 4 = 100 + (24 + 5) * 4 = 100 + 29 * 4 = 100 + 116 = 216.

So, the address of A1(4, 12) is 216.

1. **Define dynamic memory allocation?**

Answer: Dynamic memory allocation is the process of allocating memory storage during the runtime of a program. It allows for flexible use of memory by allocating and deallocating memory as needed. Functions like `malloc`, `calloc`, `realloc`, and `free` in C and `new` and `delete` in C++ are used for dynamic memory allocation.

1. **Define referential structure?**

Answer: A referential structure in computer science is a data structure that uses references (or pointers) to connect data elements. Linked lists, trees, and graphs are examples of referential structures, where nodes or elements reference other nodes or elements to establish relationships and connections within the structure.

1. **Array is a heterogeneous data type. (True/False). Justify your answer.**

Answer: False. An array is a homogeneous data type, meaning all elements in an array must be of the same type. This ensures uniformity and allows for efficient memory allocation and access.

1. **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 ?**

Answer: Such a matrix is known as a **sparse matrix**. Sparse matrices are used to save memory and computational time when dealing with matrices that have a high proportion of zero elements.

1. **Convert infix to Postfix and Prefix**

Answer:

(A + B) / C – D * E

- Postfix: AB + C / DE * -
- Prefix: - / + A B C * D E

P ^ Q ^ R + S / T

- Postfix: P Q ^ R ^ S T / +
- Prefix: + ^ P ^ Q R / S T

A * B – (C / D + (E – F)) ^ G

- Postfix: AB * CD / EF - + G ^ -
- Prefix: - * A B ^ + / C D - E F G

List applications of stack and Convert 2 * 3 / (2-1) + 5 * 3 infix expression into postfix format. Showing stacks status after every step in tabular form and evaluates that postfix notation.

Answer: Applications of Stack:

- Expression evaluation and conversion (infix, postfix, prefix)
- Syntax parsing
- Backtracking algorithms
- Function call management in recursion
- Undo mechanisms in text editors

Conversion to Postfix:

- Infix: 2 * 3 / (2 - 1) + 5 * 3
- Postfix: 2 3 * 2 1 - / 5 3 * +

Stack status for each step:

Step	Stack	Output
2		2
*	*	2
3	*	2 3
/	/	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
End		2 3 * 2 1 - / 5 3 * +

Evaluation of Postfix:

- 2 3 * = 6
- 6 2 1 - / = 6 1 /
- 6 / 1 = 6
- 6 5 3 * + = 6 15 +
- 6 + 15 = 21

So, the value of the expression is 21.

1. **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. INFIX- A + B – C * D / E + F * G / (I + J)**

Answer:

- Infix: A + B – C * D / E + F * G / (I + J)
- Postfix: A B + C D * E / - F G * I J + / +

Evaluation with given values:

- A = 1, B = 2, C = 3, D = 4, E = 6, F = 6, G = 1, I = 3, J = 3
- Postfix: 1 2 + 3 4 * 6 / - 6 1 * 3 3 + / +
- Steps:
    - 1 2 + = 3
    - 3 4 * = 12
    - 12 6 / = 2
    - 3 2 - = 1
    - 6 1 * = 6
    - 3 3 + = 6
    - 6 / 6 = 1
    - 1 6 + = 7

So, the value of the expression is 7.

29 . Differentiate between LIFO and FIFO access mechanism.

Answer:

- **LIFO (Last In, First Out):** The last element added is the first to be removed. This mechanism is used in stacks.
- **FIFO (First In, First Out):** The first element added is the first to be removed. This mechanism is used in queues.

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

- **Dynamic Size:** Linked lists can grow and shrink dynamically, unlike arrays that have a fixed size.
- **Efficient Insertions/Deletions:** Linked lists allow for efficient O(1) insertions and deletions at any position, unlike arrays that require shifting elements.

- **Memory Utilization:** Linked lists use memory more efficiently because they do not require a contiguous block of memory.

Dynamic Memory Allocation: Linked lists allocate memory as needed for each new element, which can be anywhere in the heap memory. This contrasts with arrays that require a contiguous block of memory, leading to potential wastage.

31-In which type of scenario, linear queue (simple queue) is better than a circular queue?

A linear queue is better when the number of enqueues and dequeues is small and predictable, ensuring that the queue will not fill up frequently. It is also simpler to implement and understand when circular behavior is not needed.

32- **After evaluation of 3,5,4,*,+ result is?**

Answer: The expression 3 5 4 * + in postfix notation:

- 5 4 * = 20
- 3 + 20 = 23

So, the result is 23.

33-What will be the value of Front and Rear pointers when the queue is empty?

Answer: When the queue is empty, typically:

- **Front:** -1
- **Rear:** -1

However, implementations may vary.

34-Apply selection sort algorithm on the following input: 12, 29, 25, 8, 32, 17, 40. Explain step by step.

Answer:

- Initial Array: [12, 29, 25, 8, 32, 17, 40]

Step-by-step:

1-Find the minimum element (8) and swap it with the first element.

[8, 29, 25, 12, 32, 17, 40]

2-Find the minimum element (12) from the remaining array and swap with the second element.

[8, 12, 25, 29, 32, 17, 40]

3-Find the minimum element (17) from the remaining array and swap with the third element.

[8, 12, 17, 29, 32, 25, 40]

4-Find the minimum element (25) from the remaining array and swap with the fourth element.

[8, 12, 17, 25, 32, 29, 40]

Find the minimum element (29) from the remaining array and swap with the fifth element.

- [8, 12, 17, 25, 29, 32, 40]

The array is now sorted.

35- **Write an algorithm for bubble sort. Apply it on random 8 input data.**

Answer: Algorithm for Bubble Sort:

1. Start from the first element, compare the current element with the next element.
2. If the current element is greater than the next element, swap them.
3. Move to the next element and repeat step 2 until the end of the array.
4. Repeat steps 1-3 for all elements until no swaps are needed.

Example:

- Input: [5, 1, 4, 2, 8, 3, 7, 6]

Step-by-step:

1. [1, 5, 4, 2, 8, 3, 7, 6]
2. [1, 4, 5, 2, 8, 3, 7, 6]
3. [1, 4, 2, 5, 8, 3, 7, 6]
4. [1, 4, 2, 5, 3, 8, 7, 6]
5. [1, 4, 2, 5, 3, 7, 8, 6]
6. [1, 4, 2, 5, 3, 7, 6, 8]
7. [1, 2, 4, 5, 3, 7, 6, 8]
8. [1, 2, 4, 3, 5, 7, 6, 8]
9. [1, 2, 3, 4, 5, 7, 6, 8]
10. [1, 2, 3, 4, 5, 6, 7, 8]

Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8]

36- **Write Merge Sort algorithm. Apply the algorithm to the following elements: 10, 5, 28, 7, 39, 310, 55, 15, 1**

Answer: Merge Sort Algorithm:

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two halves to produce a sorted array.

Example:

- Input: [10, 5, 28, 7, 39, 310, 55, 15, 1]

Step-by-step:

1. Divide: [10, 5, 28, 7, 39] and [310, 55, 15, 1]
2. Further divide: [10, 5] and [28, 7, 39]; [310, 55] and [15, 1]
3. Further divide: [10] and [5]; [28] and [7, 39]; [310] and [55]; [15] and [1]
4. Merge: [5, 10]; [7, 28, 39]; [55, 310]; [1, 15]
5. Merge: [5, 10] and [7, 28, 39]; [1, 15] and [55, 310]
6. Merge: [5, 7, 10, 28, 39]; [1, 15, 55, 310]
7. Merge: [1, 5, 7, 10, 15, 28, 39, 55, 310]

Sorted Array: [1, 5, 7, 10, 15, 28, 39, 55, 310]