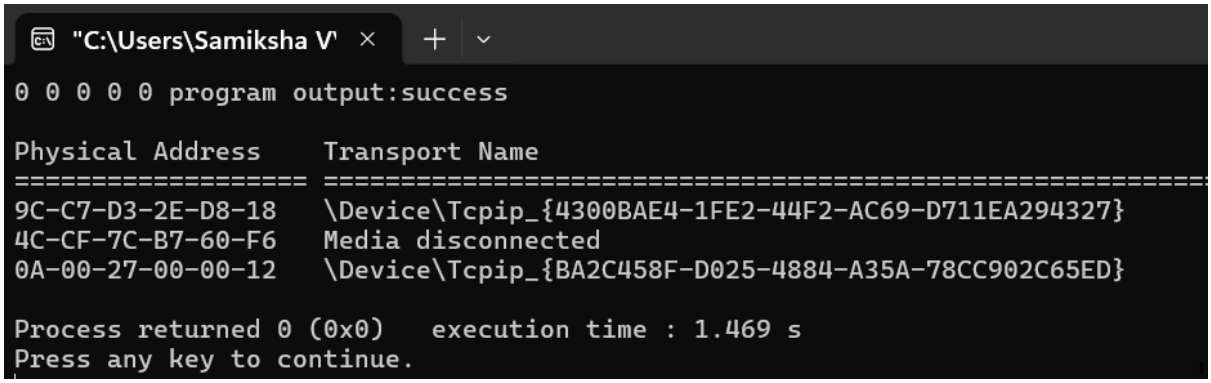
1.INITIALIZATION OF MALLOC() FUNCTION INTO ZERO.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
int n=5;
int*arr=(int *)malloc(n*sizeof(int));
if(arr==NULL)
{
printf("Memory allocation failed\n");
return 1;
}
memset(arr,0,n*sizeof(int));
for (int i = 0; i < n; i++)
{
printf("%d ",arr[i]);
}
free(arr);
printf("program output:success\n");
system("getmac");
return 0;
}
}
```

Output:

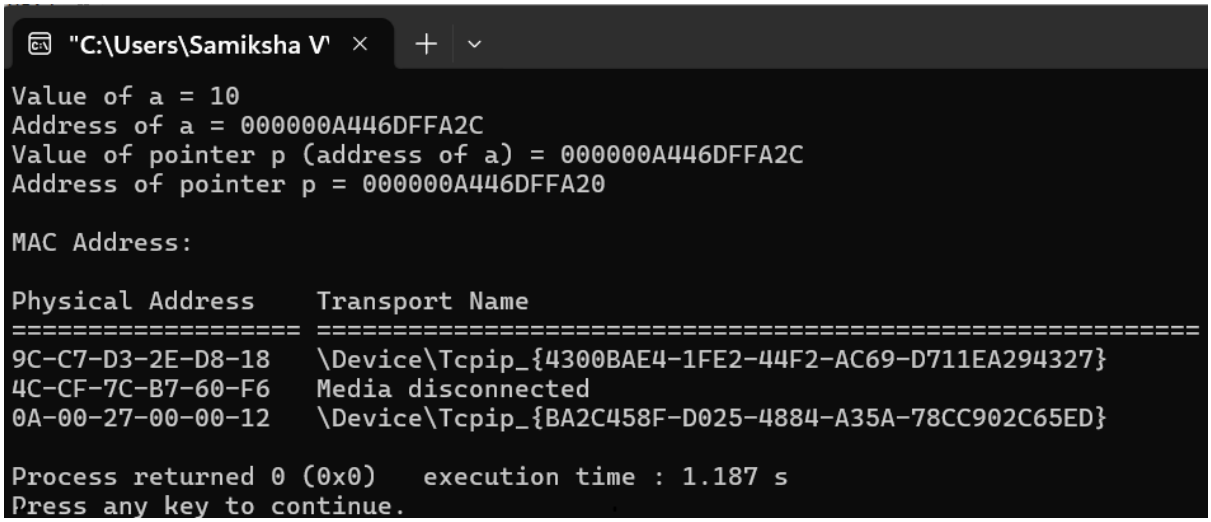


2.PRINTING THE ADDRESS OF THE POINTER.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
   int n = 5;
   int *arr = (int *)malloc(n * sizeof(int));
   if (arr == NULL)
   {
      printf("Memory allocation failed\n");
      return 1;
   }
   // Initialize malloc memory to zero
   memset(arr, 0, n * sizeof(int));

   // Print array elements
   printf("Array elements after initialization:\n");
   for (int i = 0; i < n; i++)
   {
      printf("%d ", arr[i]);
   }
   printf("\n");
   free(arr);
```

```
    printf("\nProgram output: Success\n");
    // Display MAC address
    printf("\nMAC Address:\n");
    system("getmac");


    return 0;
}
```

Output:

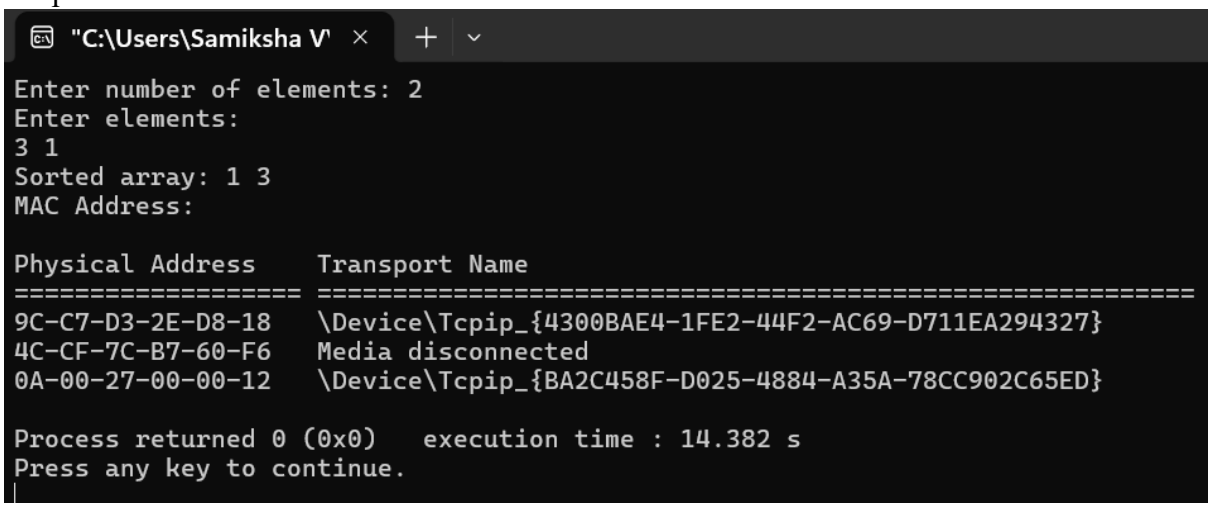


3.SELECTION SORT WITH DYNAMIC MEMORY ALLOCATION.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int *arr = (int *)malloc(n * sizeof(int));
    printf("Enter elements:\n");
    for(int i=0;i<n;i++) scanf("%d", &arr[i]);
    //Selection sort
    for(int i=0;i<n-1;i++)
{
```

```
        int min = i;
        for(int j=i+1;j<n;j++)
            if(arr[j]<arr[min]) min=j;
        int temp = arr[i];
        arr[i]=arr[min];
        arr[min]=temp;
}
    printf("Sorted array: ");
    for(int i=0;i<n;i++) printf("%d ", arr[i]);
    free(arr);
// Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");


    return 0;
}
```

Output:



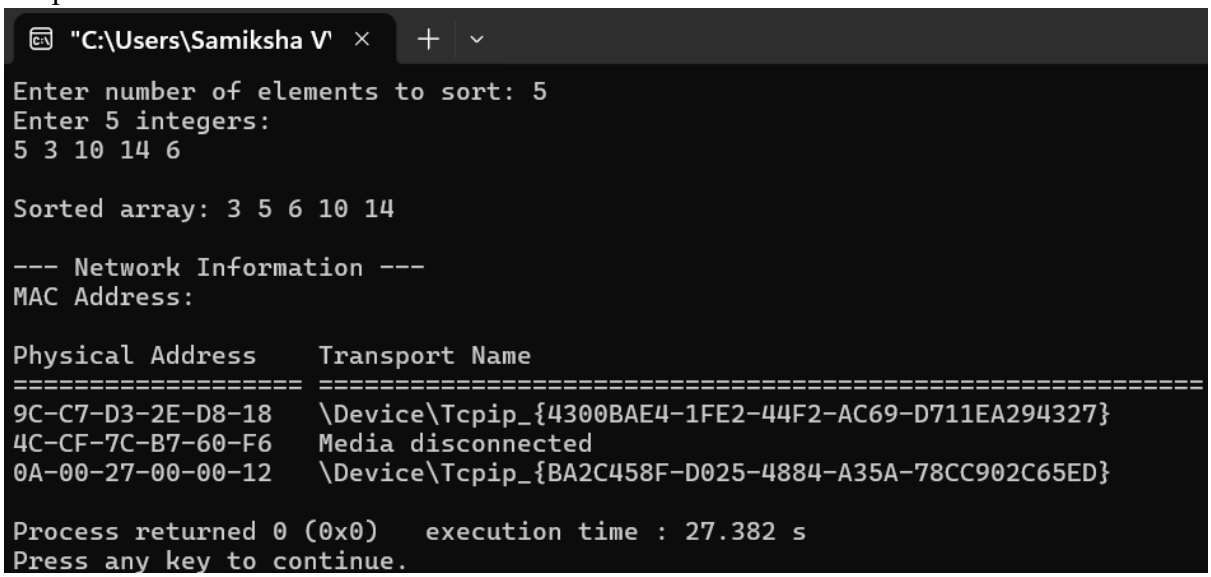
4.SELECTION SORT WITHOUT DYNAMIC MEMORY ALLOCATION.

```
#include <stdio.h>
#include <stdlib.h>// Required for system()
// Selection Sort Function
void selectionSort(int arr[], int n) {
```

```
    int i, j, min_idx, temp;
    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;
            }
        }
        // Swap
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
int main() {
    // 1. Static array (No Dynamic Memory Allocation)
    int data[100];
    int n, i;
    printf("Enter number of elements to sort: ");
    if (scanf("%d", &n) != 1) return 1;
    printf("Enter %d integers:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &data[i]);
    }
    // Perform Sort
    selectionSort(data, n);
    printf("\nSorted array: ");
    for (i = 0; i < n; i++) {
        printf("%d ", data[i]);
    }
```

```
    printf("\n");
    // 2. Display MAC Address (Windows Only)
    printf("\n--- Network Information ---");
    printf("\nMAC Address:\n");
    system("getmac");


    return 0;
}
```

Output:

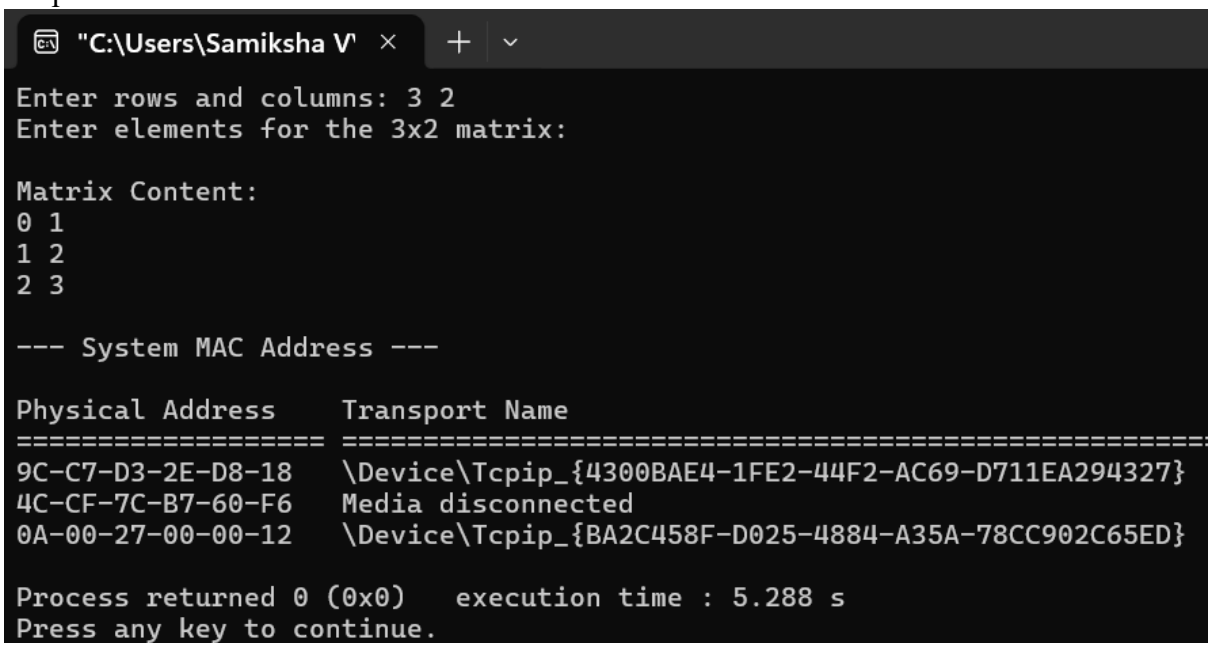


5.DECLARATION OF TWO-DIMENSIONAL ARRAY USING DYNAMIC MEMORY ALLOCATION.

```
#include <stdio.h>
#include <stdlib.h> // Required for malloc and system()
int main() {
    int rows, cols, i, j;
    // 1. Get dimensions from user
    printf("Enter rows and columns: ");
    scanf("%d %d", &rows, &cols);
    // 2. DYNAMIC ALLOCATION
    // Allocate memory for 'rows' number of integer pointers
    int **arr = (int **)malloc(rows * sizeof(int *));
```

```
    // For each row, allocate 'cols' number of integers
    for (i = 0; i < rows; i++) {
        arr[i] = (int *)malloc(cols * sizeof(int));
    }
    // 3. Input and Output data
    printf("Enter elements for the %dx%d matrix:\n", rows, cols);
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            arr[i][j] = i + j;  }
    }
    printf("\nMatrix Content:\n");
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            printf("%d ", arr[i][j]);   }
        printf("\n");
    }
    // 4. FREE MEMORY (Important to prevent memory leaks)
    for (i = 0; i < rows; i++) {
        free(arr[i]);
    }
    free(arr);
    // 5. Display MAC Address (Windows)
    printf("\n--- System MAC Address ---\n");
    system("getmac");
    return 0;
}
```

Output:

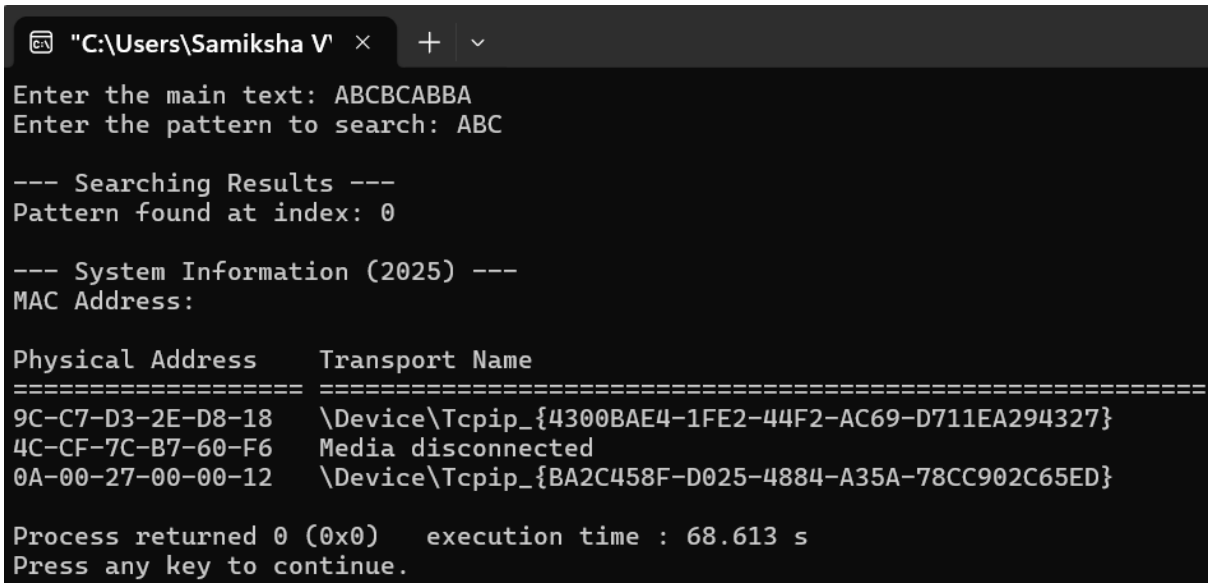


6.PATTERN MATCHING (GENERAL WAY).

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h> // Required for system()
// General Pattern Matching Function (Naive Approach)
void findPattern(char* text, char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int found = 0;
    // Slide the pattern over the text one by one
    for (int i = 0; i <= n - m; i++) {
        int j;
        // For current position i, check for pattern match
        for (j = 0; j < m; j++) {
            if (text[i + j] != pattern[j]) {
                break;
            }
        }
        // If the pattern matched
```

```
        if (j == m) {
            printf("Pattern found at index: %d\n", i);
            found = 1;
        }
    }
    if (!found) {
        printf("Pattern not found in the given text.\n");
    }
}
int main() {
    char text[100], pattern[50];
    // 1. Input Text and Pattern
    printf("Enter the main text: ");
    gets(text);

    printf("Enter the pattern to search: ");
    gets(pattern);

    // 2. Execute Matching
    printf("\n--- Searching Results ---\n");
    findPattern(text, pattern);

    // 3. Display MAC Address (Windows Only)
    printf("\n--- System Information (2025) ---\n");
    printf("MAC Address:\n");
    system("getmac");

    return 0;
}
```

OUTPUT:

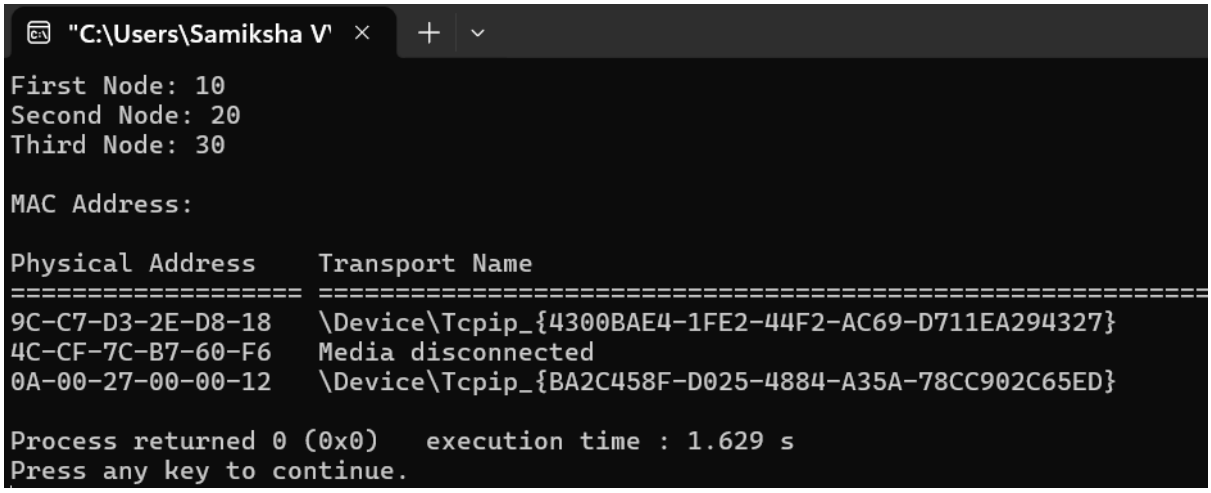


7.SELF-REFERENTIAL STRUCTURE(LINKED LIST NODE).

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

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

int main() {
    struct Node *head = NULL;
    struct Node *second = NULL;
    struct Node *third = NULL;

    // Dynamically allocate memory for nodes
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    // Assign data and link nodes
```

```
    head->data = 10;      // Assign data to first node
    head->next = second;  // Link first node with the second

    second->data = 20;    // Assign data to second node
    second->next = third;   // Link second node with the third

    third->data = 30;     // Assign data to third node
    third->next = NULL;   // Terminate the list

    // Access and print data
    printf("First Node: %d\n", head->data);
    printf("Second Node: %d\n", head->next->data);
    printf("Third Node: %d\n", second->next->data);
    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");

    // Free allocated memory to prevent memory leaks (omitted for brevity, but essential in real code)

    return 0;
}
```

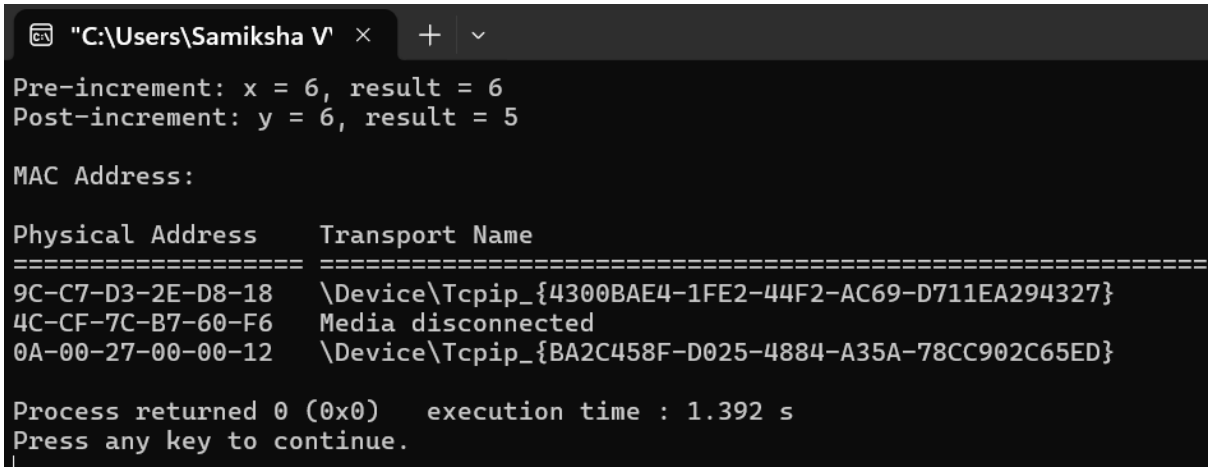
OUTPUT:



8.PRE INCREMENT AND POST INCREMENT.

```
#include <stdio.h>
#include <stdlib.h> // Needed for the system() function


int main() {
    int x = 5, y = 5;
    int result_pre, result_post;


    // 1. Pre-increment: ++x (Increment first, then use)
    result_pre = ++x;
    printf("Pre-increment: x = %d, result = %d\n", x, result_pre); // Both 6
    // 2. Post-increment: y++ (Use first, then increment)
    result_post = y++;
    printf("Post-increment: y = %d, result = %d\n", y, result_post); // y is 6, result is 5
    // 3. Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");
    return 0;
}
```

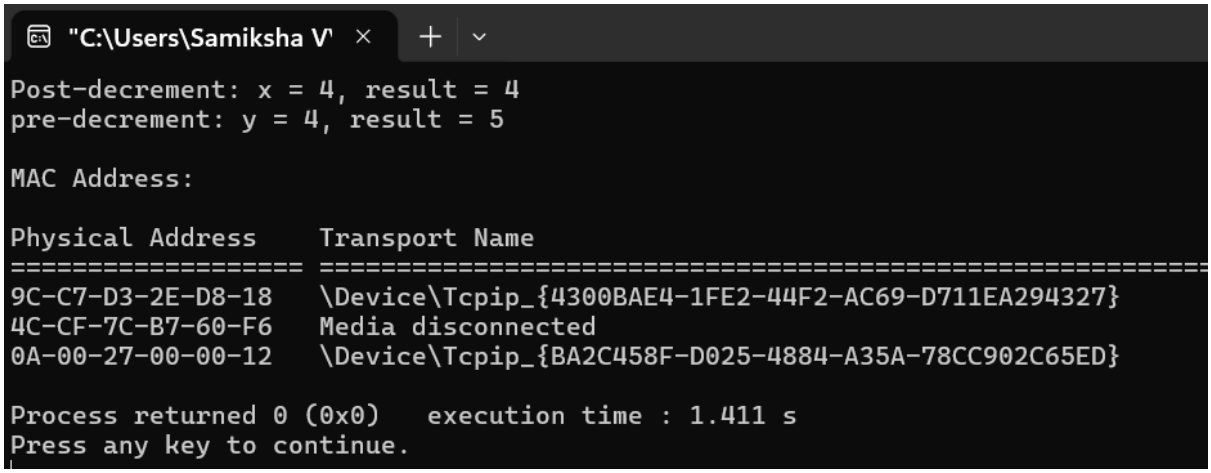
OUTPUT:



9.POST DECREMENT AND PRE DECREMENT.

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

int main() {
    int x = 5, y = 5;
    int result_pre, result_post;

    // 1. post decrement: --x (decrement first, then use)

    result_pre = --x;
    printf("Post-decrement: x = %d, result = %d\n", x, result_pre);

    // 2. Pre-decrement: y-- (Use first, then decrement)
    result_post = y--;
    printf("pre-decrement: y = %d, result = %d\n", y, result_post);

    // 3. Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");
    return 0;
}
```

OUTPUT:



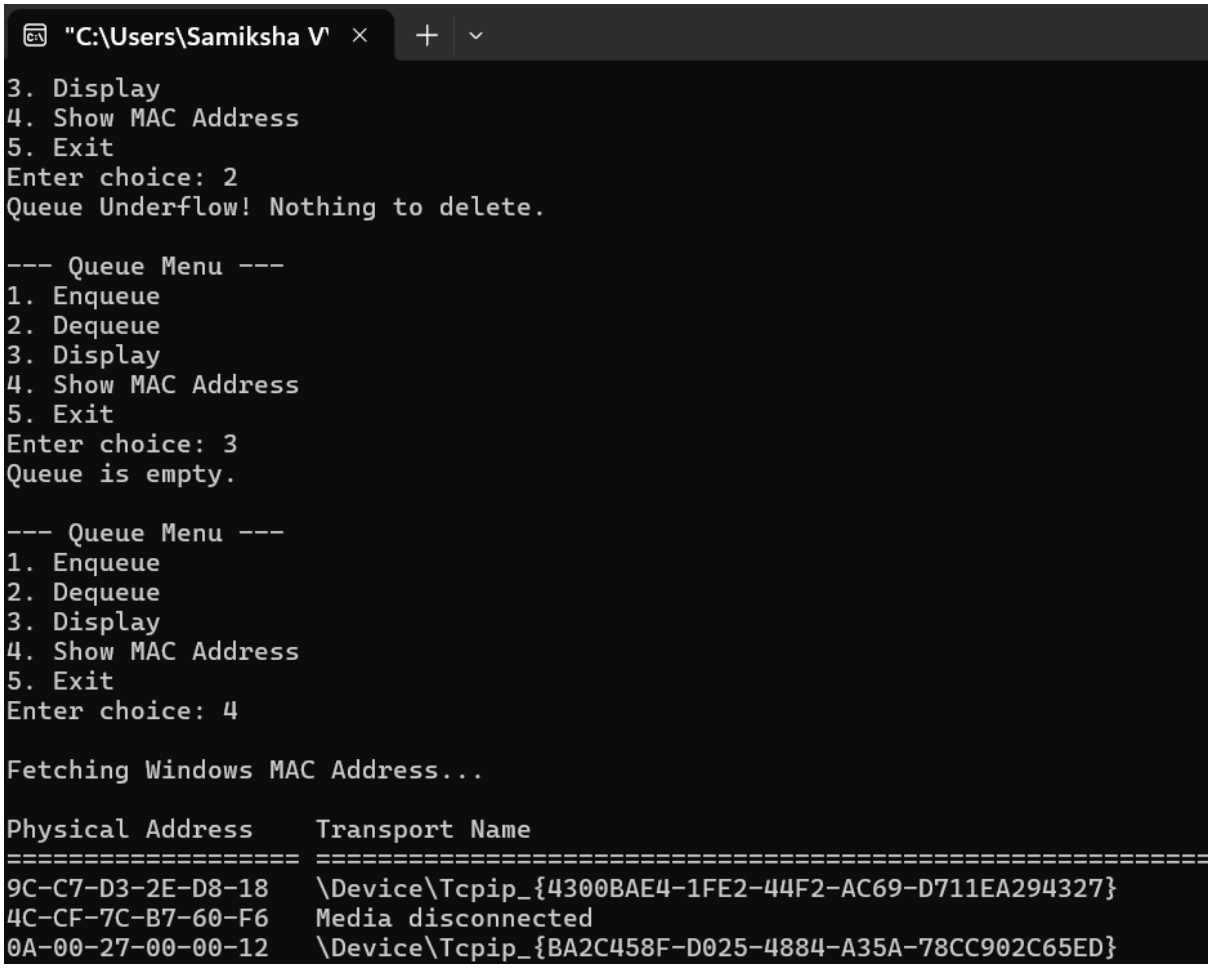
10.REGULAR QUEUE(ARRAY IMPLEMENT).

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5

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

// Function to add an element
void enqueue(int value) {
   if (rear == SIZE - 1) {
      printf("Queue Overflow! Cannot add %d\n", value);
   } else {
      if (front == -1) front = 0; // Set front to 0 for first element
      queue[++rear] = value;
      printf("Enqueued: %d\n", value);
   }
}

// Function to remove an element
void dequeue() {
```

```
    if (front == -1 || front > rear) {
        printf("Queue Underflow! Nothing to delete.\n");
    } else {
        printf("Dequeued: %d\n", queue[front++]);
        // Reset pointers if queue becomes empty
        if (front > rear) front = rear = -1;
    }
}
// Function to display queue elements
void display() {
    if (front == -1) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++)
            printf("%d ", queue[i]);
        printf("\n");
    }
}

int main() {
    int choice, val;

    while (1) {
        printf("\n--- Queue Menu ---\n");
        printf("1. Enqueue\n2. Dequeue\n3. Display\n4. Show MAC Address\n5. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
```

```
            case 1:
                printf("Enter value: ");
                scanf("%d", &val);
                enqueue(val);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("\nFetching Windows MAC Address...\n");
                // Uses the Windows 'getmac' command via system shell
                system("getmac");
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

OUTPUT:



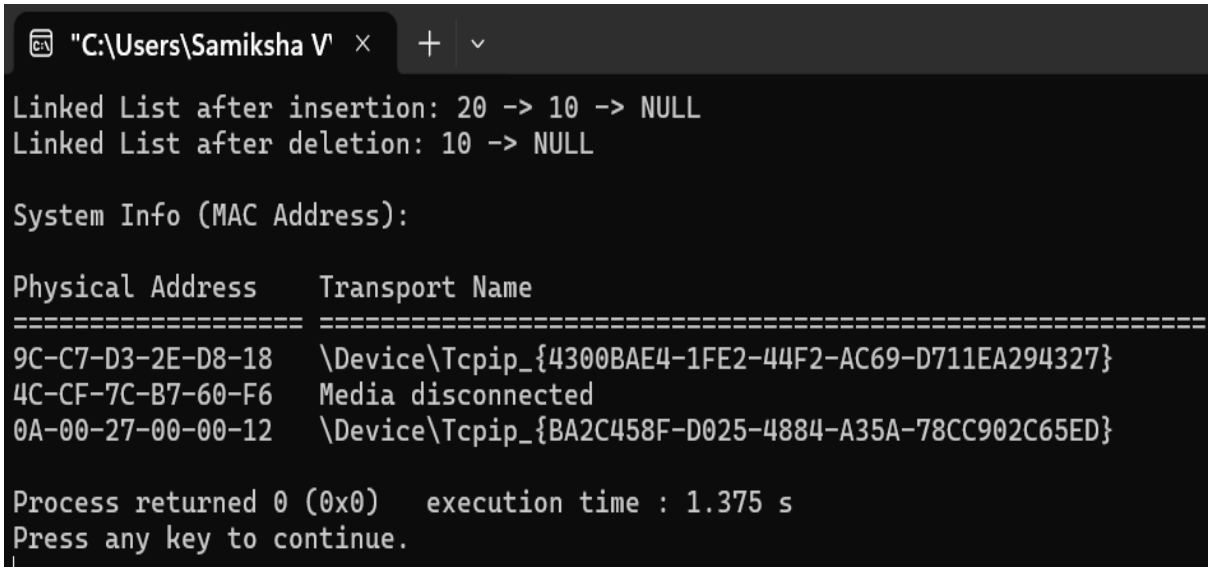
11. OPERATION OF LINKED LISTS(TRAVERSAL,INSERTION,DELETION).

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

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

// Traversal: Print all nodes
void traverse(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
```

```
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Insertion: Add to the beginning
void insertHead(struct Node** head, int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->next = *head;
    *head = newNode;
}

// Deletion: Remove the first node
void deleteHead(struct Node** head) {
    if (*head == NULL) return;
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}

int main() {
    struct Node* list = NULL;

    insertHead(&list, 10);
    insertHead(&list, 20);
    printf("Linked List after insertion: ");
    traverse(list);
```

```
    deleteHead(&list);
    printf("Linked List after deletion: ");
    traverse(list);

    // Windows MAC Address Display
    printf("\nSystem Info (MAC Address):\n");
    system("getmac");

    return 0;
}
```

OUTPUT:

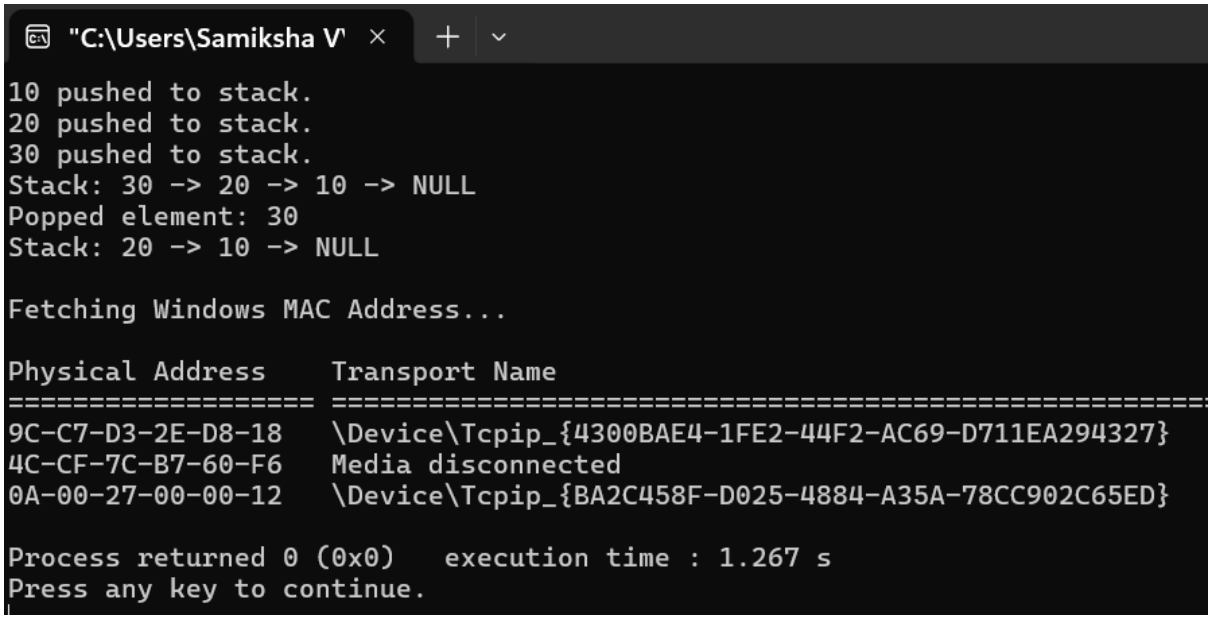


12.LINKED LIST USING STACKS.

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

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

struct Node* top = NULL; // Initialize top as empty

void push(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Heap Overflow! Cannot allocate memory.\n");
        return;
    }
    newNode->data = val;
    newNode->next = top;
    top = newNode;
    printf("%d pushed to stack.\n", val);
}

void pop() {
    if (top == NULL) {
        printf("Stack Underflow! Nothing to pop.\n");
        return;
    }
    struct Node* temp = top;
    printf("Popped element: %d\n", top->data);
```

```
    top = top->next;
    free(temp); // Free memory to avoid leaks
}

void display() {
    if (top == NULL) {
        printf("Stack is empty.\n");
        return;
    }
    struct Node* temp = top;
    printf("Stack: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main() {
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();

    printf("\nFetching Windows MAC Address...\n");
    system("getmac"); // Windows command only

    return 0;
}
```

OUTPUT:



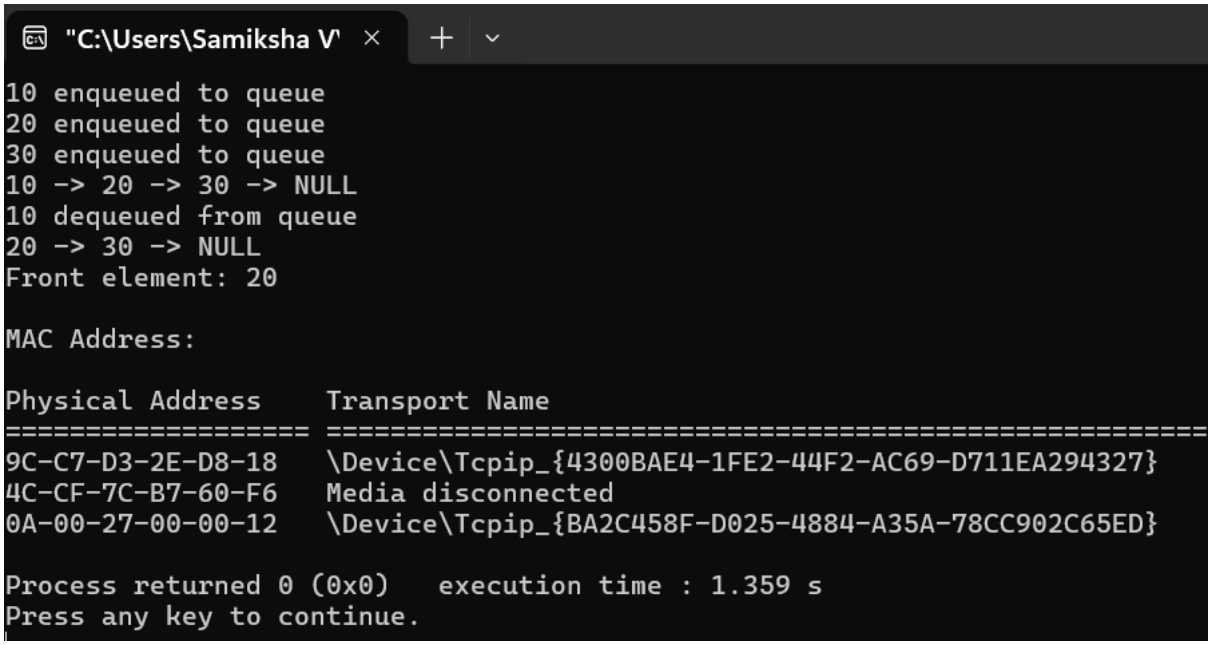
13.LINKED LIST USING QUEUES.

```
#include <stdio.h>
#include <stdlib.h>
// 1. Define the Node structure
struct Node {
    int data;
    struct Node* next;
};
// Global pointers for front and rear
struct Node* front = NULL;
struct Node* rear = NULL;

// 2. Enqueue: Add an element to the back (O(1))
void enqueue(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed (Overflow)\n");
        return;
    }
```

```
    newNode->data = val;
    newNode->next = NULL;

    if (rear == NULL) { // Queue is empty
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
    printf("%d enqueued to queue\n", val);
}

// 3. Dequeue: Remove an element from the front (O(1))
void dequeue() {
    if (front == NULL) {
        printf("Queue Underflow\n");
        return;
    }
    struct Node* temp = front;
    printf("%d dequeued from queue\n", temp->data);
    front = front->next;

    if (front == NULL) { // If queue becomes empty, update rear
        rear = NULL;
    }
    free(temp);
}
// 4. Peek: View the front element
int peek() {
    if (front == NULL) return -1;
```

```
    return front->data;
}
// 5. Display the entire queue
void display() {
    struct Node* temp = front;
    if (front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
    printf("Front element: %d\n", peek());
    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");
    return 0;
}
```

OUTPUT:



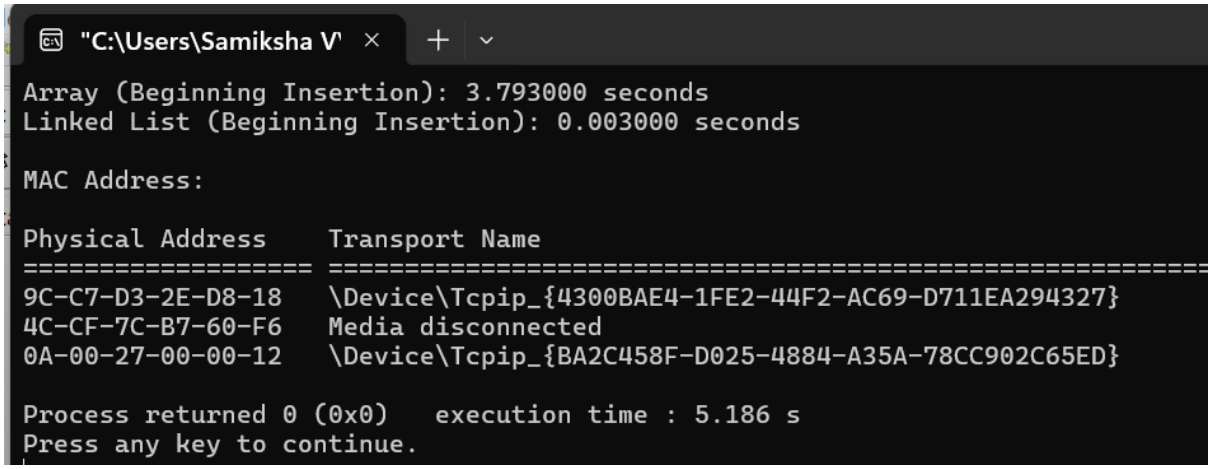
14.TIME TAKEN FOR ARRAYS AND LINKED LISTS DURING INSERTION AND DELETION.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 100000
// Linked List Node
struct Node {
    int data;
    struct Node* next;
};
// Function to insert at the beginning of an array (O(n))
void insertArray(int arr[], int *size, int value) {
    for (int i = *size; i > 0; i--) {
        arr[i] = arr[i - 1]; // Shifting elements
    }
    arr[0] = value;
    (*size)++;
}
```

```
// Function to insert at the beginning of a Linked List (O(1))
void insertList(struct Node** head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = *head;
    *head = newNode;
}
int main() {
    clock_t start, end;
    double cpu_time_used;
    // --- Array Test ---
    int *arr = (int*)malloc(N * sizeof(int));
    int currentSize = 0;
    start = clock();
    for (int i = 0; i < N; i++) {
        insertArray(arr, &currentSize, i);
    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Array (Beginning Insertion): %f seconds\n", cpu_time_used);
    // --- Linked List Test ---
    struct Node* head = NULL;
    start = clock();
    for (int i = 0; i < N; i++) {
        insertList(&head, i);
    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Linked List (Beginning Insertion): %f seconds\n", cpu_time_used);
    // Display MAC Address (Windows only)
```

```
    printf("\nMAC Address:\n");
    system("getmac");

    return 0;
}
```

OUTPUT:

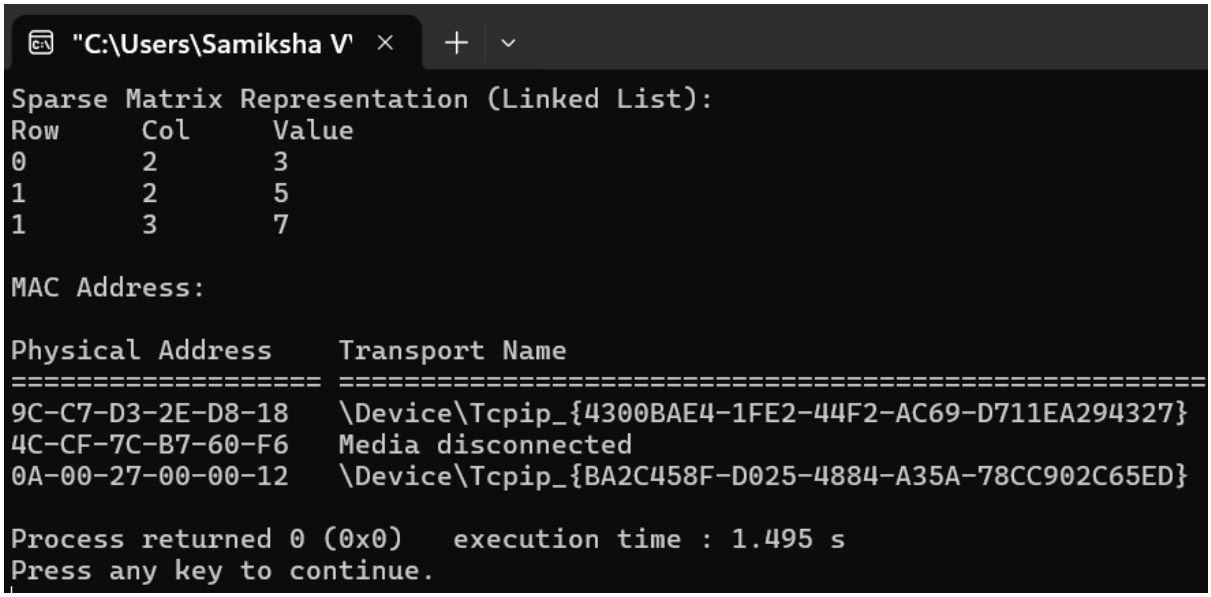


15.SPARSE MATRIX.

```
#include <stdio.h>
int main() {
    // 1. Initial 4x5 Sparse Matrix
    int sparseMatrix[4][5] = {
        {0, 0, 3, 0},
        {0, 0, 5, 7},
        {0, 0, 0, 0}
    };
    int size = 0;
    // 2. Count non-zero elements to determine size of compressed matrix
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 5; j++) {
            if (sparseMatrix[i][j] != 0) {
                size++;
```

```
            }
        }
    }
    // 3. Create Triplet Matrix (3 rows: Row index, Col index, Value)
    int triplet[3][size];
    int k = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 5; j++) {
            if (sparseMatrix[i][j] != 0) {
                triplet[0][k] = i;            // Row index
                triplet[1][k] = j;            // Column index
                triplet[2][k] = sparseMatrix[i][j]; // Non-zero value
                k++;
     }
 }
}
    // 4. Display the Triplet Representation
    printf("Triplet Representation:\n");
    printf("Row\tCol\tValue\n");
    for (int i = 0; i < size; i++) {
        printf("%d\t%d\t%d\n", triplet[0][i], triplet[1][i], triplet[2][i]);
    }
// Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");

    return 0;
}
```

OUTPUT:

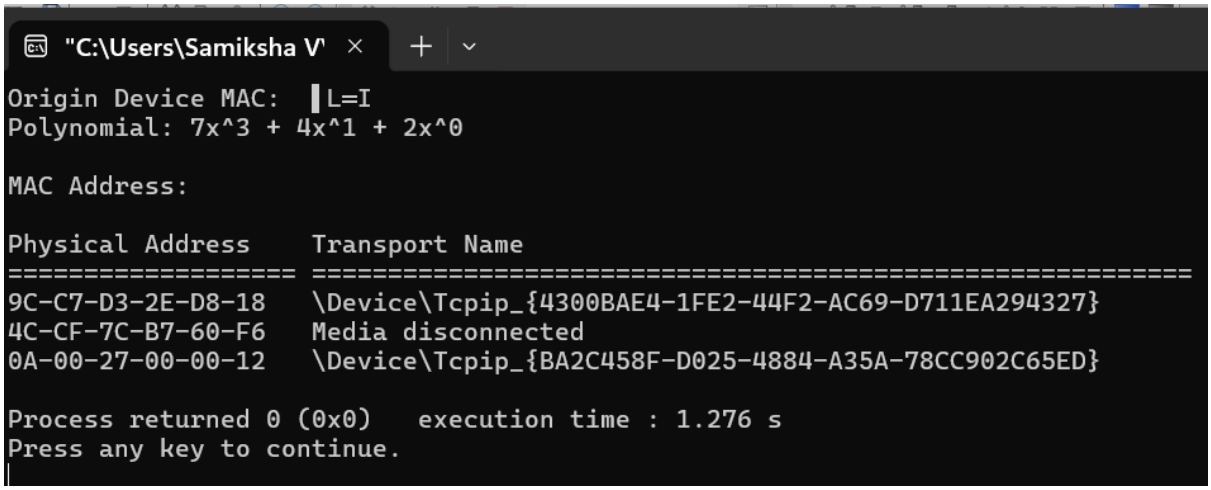


16.POLYNOMIAL REPRESENTATION .

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Node {
    int coeff;
    int exp;
    struct Node* next;
};
struct Polynomial {
    char mac_address[18];
    struct Node* head;
};
void addTerm(struct Node** head, int c, int e) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->coeff = c;
    newNode->exp = e;
```

```
    newNode->next = *head;
    *head = newNode;
}
void displayPolynomial(struct Polynomial p) {
    printf("Origin Device MAC: %s\n", p.mac_address);
    printf("Polynomial: ");
    struct Node* temp = p.head;
    while (temp != NULL) {
        printf("%dx^%d", temp->coeff, temp->exp);
        temp = temp->next;
        if (temp != NULL) printf(" + ");
    }
    printf("\n");
}
int main() {
    struct Polynomial myPoly;
    // Representing: 7x^3 + 4x^1 + 2x^0
    addTerm(&myPoly.head, 2, 0);
    addTerm(&myPoly.head, 4, 1);
    addTerm(&myPoly.head, 7, 3);
    displayPolynomial(myPoly);

    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");

    return 0;
}
```

OUTPUT:



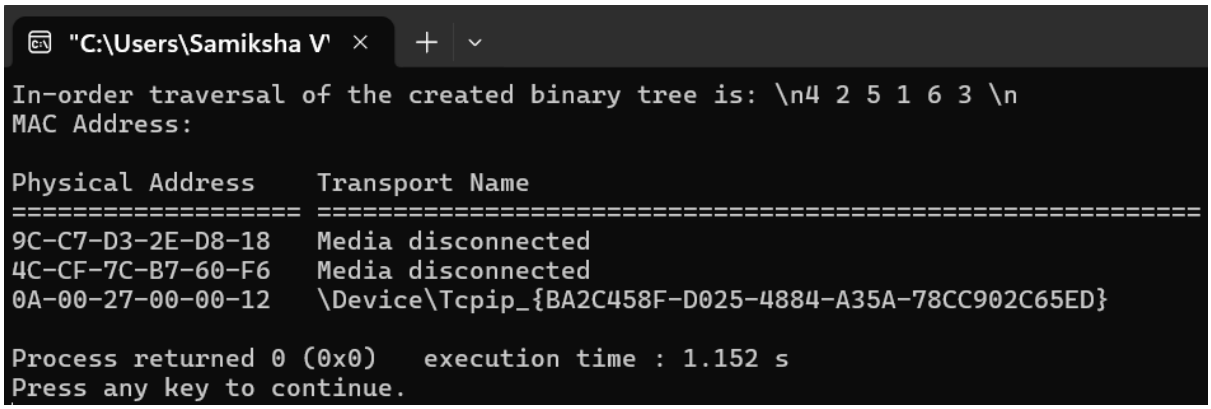
17.HOW TO CREATE TREE.

```
#include <stdio.h>
#include <stdlib.h>
// Step 1: Define the Node Structure
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
// Step 2: Function to Create a New Node
// Allocates memory for a new node and initializes it.
struct Node* createNode(int value)
 {
    // Use malloc for dynamic memory allocation.
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\\n");
        exit(1);
    }
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
```

```
    return newNode;
}
// Step 3: Function for In-Order Traversal
// Traverses the tree in the order: Left, Root, Right.
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
        // Recursively visit the left subtree.
        inOrderTraversal(root->left);
        // Print the current node's data.
        printf("%d ", root->data);
        // Recursively visit the right subtree.
        inOrderTraversal(root->right);
    }
}
// Step 4: Main Function to Build and Traverse the Tree
int main() {
    // Create the root node.
    struct Node* root = createNode(1);

    // Build the tree manually for this example.
    // root points to 1
    // 2 becomes the left child of 1
    root->left = createNode(2);
    // 3 becomes the right child of 1
    root->right = createNode(3);
    // 4 becomes the left child of 2
    root->left->left = createNode(4);
    // 5 becomes the right child of 2
    root->left->right = createNode(5);
    // 6 becomes the left child of 3
```

```
    root->right->left = createNode(6);

    printf("In-order traversal of the created binary tree is: \\n");

    inOrderTraversal(root);
    printf("\\n");
    return 0;
}
```

OUTPUT:

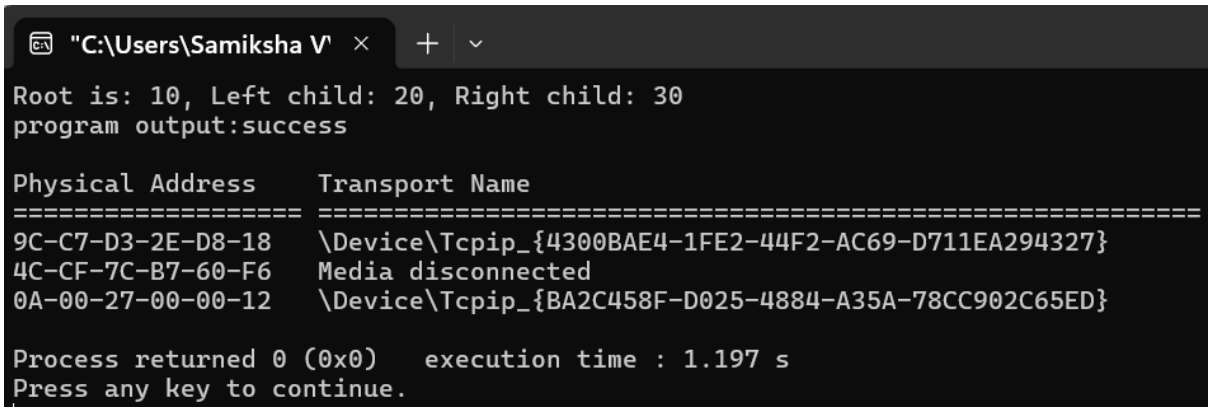


18. CONSTRUCT A TREE USING ARRAY .

```
#include <stdio.h>
int main() {
    // Imagine this tree:
    //      A (0)
    //     / \
    //    B   C (1, 2)
    char tree[] = {'A', 'B', 'C', 'D', 'E'};
    int n = 5; // Number of nodes
    printf("Node\tLeft\tRight\n");
    printf("----\t----\t-----\n");
    for (int i = 0; i < n; i++) {
        int left = (2 * i) + 1;
        int right = (2 * i) + 2;
```

```
        printf("%c\t", tree[i]);
        // Check if left child exists within array bounds
        if (left < n)
            printf("%c\t", tree[left]);
        else
            printf("-\t");

        // Check if right child exists within array bounds
        if (right < n)
            printf("%c\n", tree[right]);
        else
            printf("-\n"); }
// Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");

    return 0;
}
```

OUTPUT:



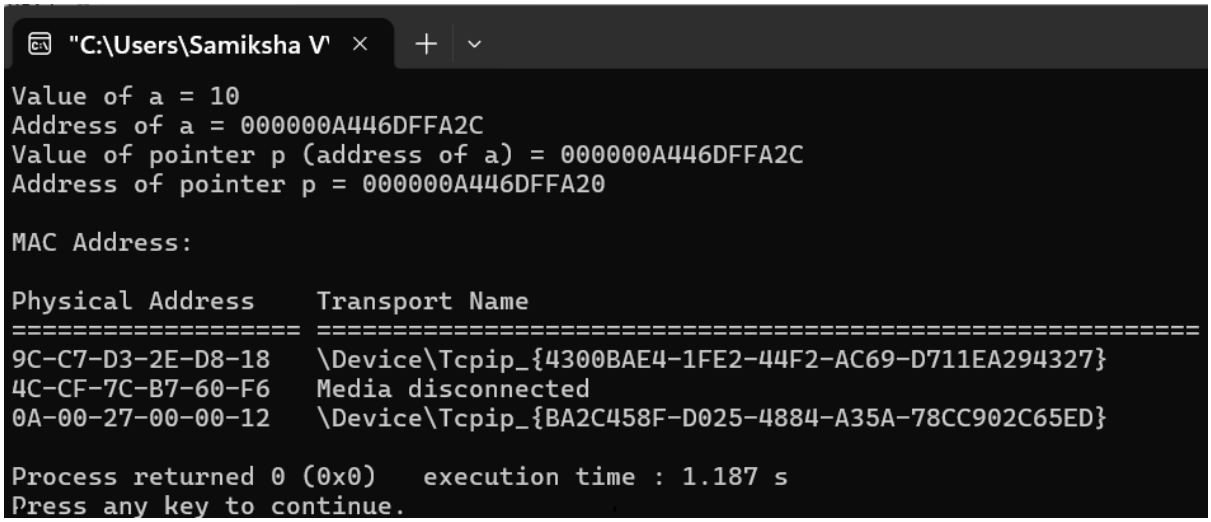
19.CONSTRUCT A BINARY TREE USING QUEUES.

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
```

```
    int data;
    struct Node *left, *right;
};
// Queue for storing node pointers
struct Node* queue[100];
int front = 0, rear = 0;
void enqueue(struct Node* n) { queue[rear++] = n; }
struct Node* dequeue() { return queue[front++]; }

struct Node* createNode(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->left = newNode->right = NULL;
    return newNode;
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = 5;
    // 1. Create root and add to queue
    struct Node* root = createNode(arr[0]);
    enqueue(root);
    int i = 1;
    while (i < n) {
        struct Node* current = dequeue();
        // 2. Add Left Child
        if (i < n) {
            current->left = createNode(arr[i++]);
            enqueue(current->left};
        // 3. Add Right Child
        if (i < n) {
```

```
            current->right = createNode(arr[i++]);
            enqueue(current->right);
        }
    }
    printf("Root is: %d, Left child: %d, Right child: %d\n",
        root->data, root->left->data, root->right->data);

    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");

    return 0;
}
```

OUTPUT:



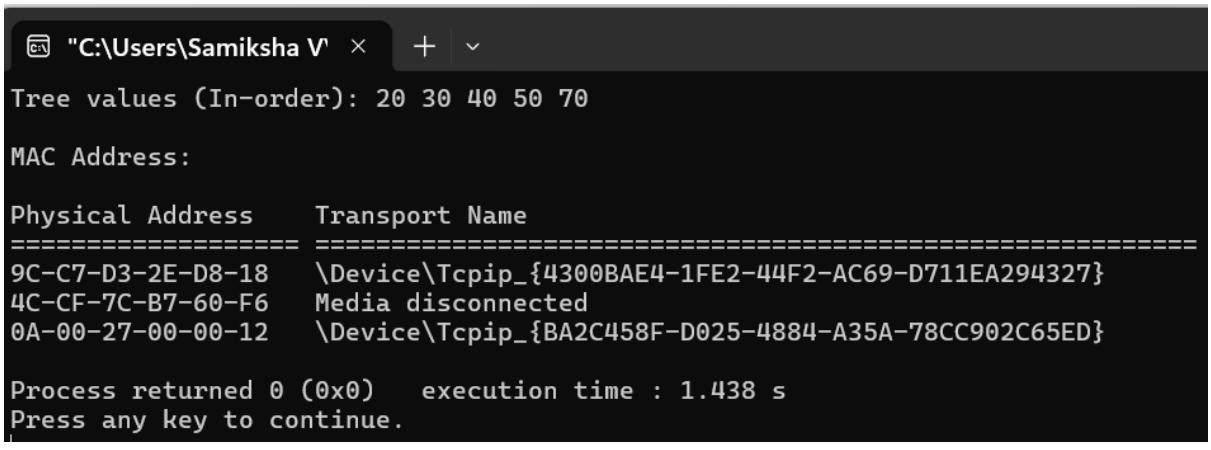
20.INSERTION OF NODES.

```
#include <stdio.h>
#include <stdlib.h>
// 1. Define the Node structure
struct Node {
    int data;
    struct Node *left, *right;
```

```
};
// 2. Function to create a new node
struct Node* createNode(int val) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->left = newNode->right = NULL;
    return newNode;
}
// 3. Function to insert a node
struct Node* insert(struct Node* root, int val) {
    // If tree is empty, return a new node
    if (root == NULL) {
        return createNode(val);
    }
    // Otherwise, recur down the tree
    if (val < root->data) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }
    return root;

// 4. Print tree (In-order) to see sorted values
void display(struct Node* root) {
    if (root != NULL) {
        display(root->left);
       printf("%d ", root->data);
        display(root->right);
    }
}
```

```
int main() {
    struct Node* root = NULL;
    // Inserting nodes
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    printf("Tree values (In-order): ");
    display(root);
    printf("\n");
    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");
    return 0;
}
```

OUTPUT:



21.DELETION OF NODES.

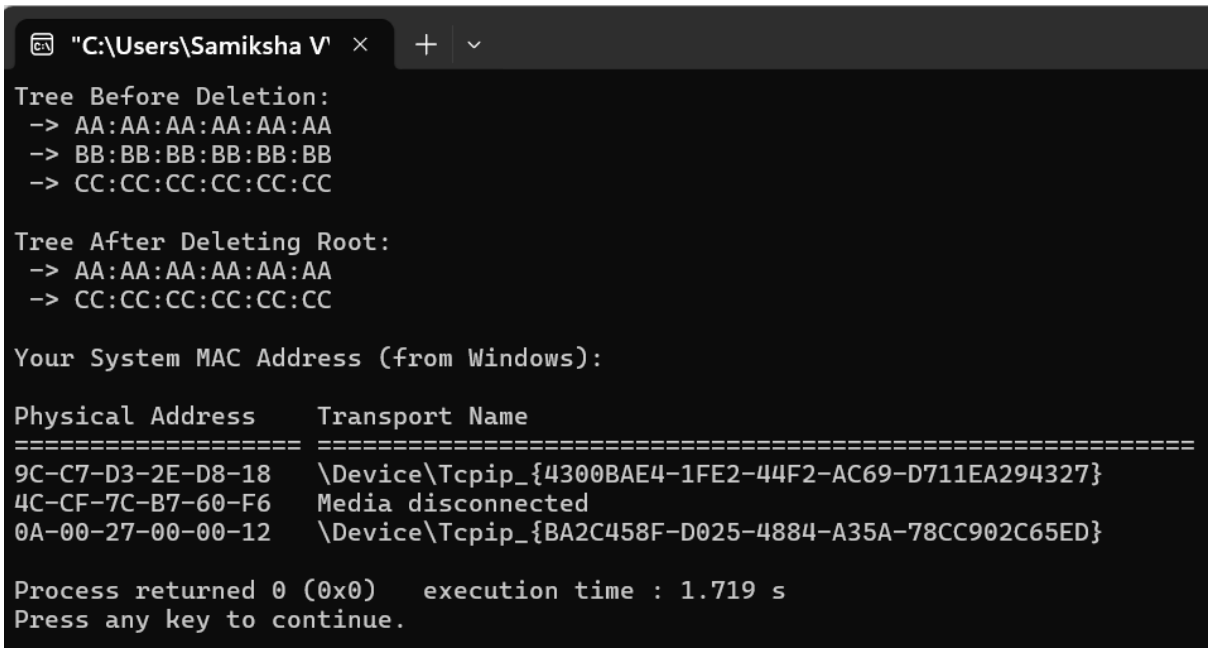
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// 1. Structure for a Tree Node
struct Node {
```

```
    char mac[18];
    struct Node *left, *right;
};
// 2. Helper to create a node
struct Node* create(char* addr) {
    struct Node* n = malloc(sizeof(struct Node));
    strcpy(n->mac, addr);
    n->left = n->right = NULL;
    return n;
}
// 3. Find smallest node (Successor)
struct Node* findMin(struct Node* root) {
    while (root->left != NULL) root = root->left;
    return root;
}
// 4. Deletion Logic
struct Node* deleteNode(struct Node* root, char* key) {
    if (root == NULL) return root;
    int cmp = strcmp(key, root->mac);
    if (cmp < 0) root->left = deleteNode(root->left, key);
    else if (cmp > 0) root->right = deleteNode(root->right, key);
    else {
        // Node found
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
```

```
            return temp;
        }
        struct Node* temp = findMin(root->right);
        strcpy(root->mac, temp->mac);
        root->right = deleteNode(root->right, temp->mac);
    }
    return root;
}
void printTree(struct Node* root) {
    if (root) {
        printTree(root->left);
        printf(" -> %s\n", root->mac);
        printTree(root->right);
    }
}
int main() {
    // Construct simple tree
    struct Node* root = create("BB:BB:BB:BB:BB:BB");
    root->left = create("AA:AA:AA:AA:AA:AA");
    root->right = create("CC:CC:CC:CC:CC:CC");
    printf("Tree Before Deletion:\n");
    printTree(root);
    // Perform Deletion
    root = deleteNode(root, "BB:BB:BB:BB:BB:BB");
    printf("\nTree After Deleting Root:\n");
    printTree(root);
    // Display Computer's MAC Address (Windows only)
    printf("\nYour System MAC Address (from Windows):\n");
    system("getmac");
    return 0;
```

}

OUTPUT:



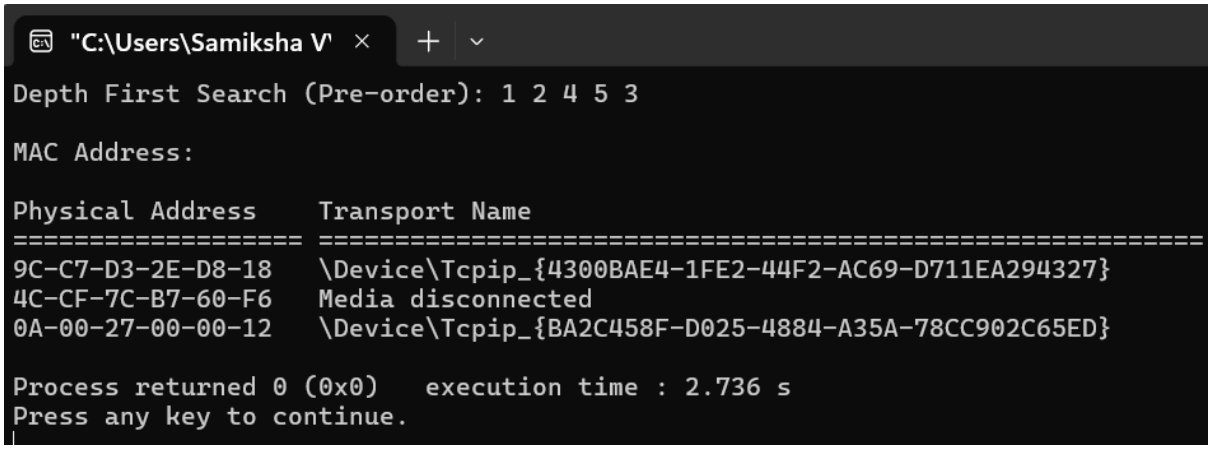
22.DEPTH SERACH TREE.

```
#include <stdio.h>
#include <stdlib.h>
// 1. Define the Tree Node
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};
// 2. Function to create a new node
struct Node* createNode(int val) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```
// 3. DFS Traversal (Pre-order: Root -> Left -> Right)
void DFS(struct Node* root) {
    if (root == NULL) return;
    // Visit the current node
    printf("%d ", root->data);
    // Go deep into left branch
    DFS(root->left)
    // Go deep into right branch
    DFS(root->right);

int main() {
    /* Create this tree:
          1
         / \
        2   3
       / \
      4   5
    */
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5)

    printf("Depth First Search (Pre-order): ");
    DFS(root);
    printf("\n");
// Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
```

```
    system("getmac");


    return 0;
}
```

OUTPUT:



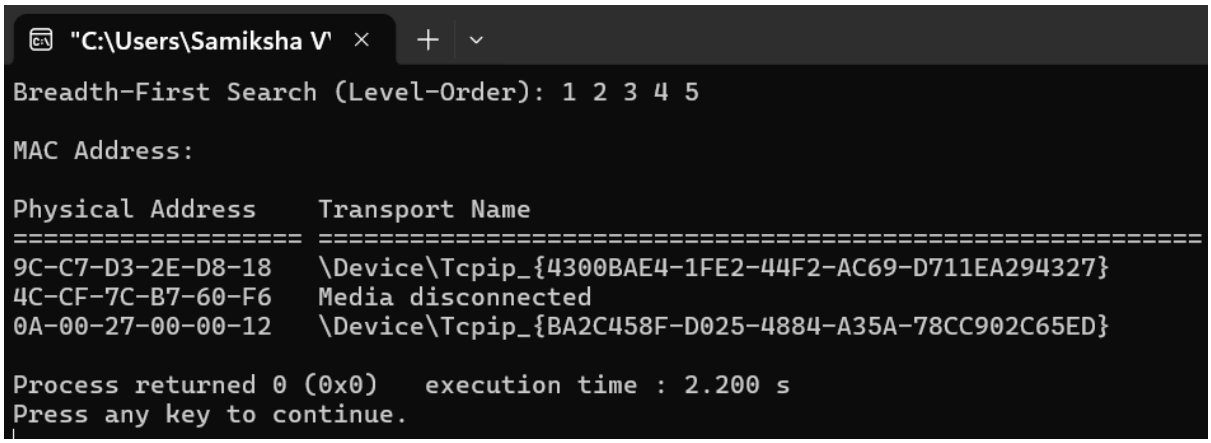
23.BREADTH SEARCH TREE.

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


// 1. Define the Tree Node structure
struct Node {
    int data;
    struct Node *left, *right;
};
// 2. Simple Queue for storing Node pointers
struct Node* queue[100];
int front = 0, rear = 0;


void enqueue(struct Node* n) {
    if (n != NULL) queue[rear++] = n;
}
struct Node* dequeue() {
    return queue[front++];
```

```
}

// 3. Helper function to create a new node
struct Node* createNode(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    newNode->data = val;
    newNode->left = newNode->right = NULL;
    return newNode;
}
// 4. BFS Logic (Level-Order Traversal)
void BFS(struct Node* root) {
    if (root == NULL) return;

    enqueue(root); // Start with root

    while (front < rear) {
        struct Node* current = dequeue();
        printf("%d ", current->data);

        // Add children to the queue to be visited in the next level
        if (current->left) enqueue(current->left);
        if (current->right) enqueue(current->right);
    }
}
int main() {
    /* Constructing the following tree:
          1
         / \
        2   3
```

```
     / \
    4   5
*/
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Breadth-First Search (Level-Order): ");
    BFS(root);
    printf("\n");

    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");

    return 0;
}
```

OUTPUT:



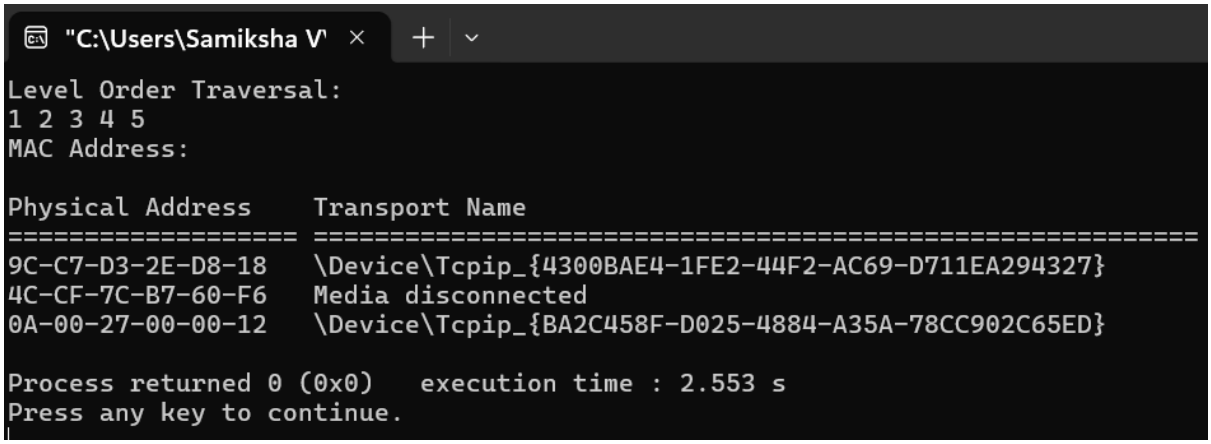
24.LEVEL ORDER.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
// Node structure
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};
// Queue for level order traversal
struct Queue {
    struct Node* items[MAX];
    int front;
    int rear;
};
// Initialize queue
void initQueue(struct Queue* q) {
    q->front = -1;
    q->rear = -1;
}
// Check if queue is empty
int isEmpty(struct Queue* q) {
    return q->front == -1;
}
// Enqueue
void enqueue(struct Queue* q, struct Node* node) {
    if (q->rear == MAX - 1) return;
    if (isEmpty(q)) q->front = 0;
    q->rear++;
```

```
    q->items[q->rear] = node;
}
// Dequeue
struct Node* dequeue(struct Queue* q) {
    if (isEmpty(q)) return NULL;
    struct Node* temp = q->items[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front++;
    }
    return temp;
}
// Create a new node
struct Node* createNode(int data) {
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}
// Level Order Traversal
void levelOrder(struct Node* root) {
    struct Queue q;
    initQueue(&q);
    if (!root) return;
    enqueue(&q, root);
    while (!isEmpty(&q)) {
        struct Node* curr = dequeue(&q);
        printf("%d ", curr->data);
        if (curr->left) enqueue(&q, curr->left);
```

```
        if (curr->right) enqueue(&q, curr->right);  }
}
int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    printf("Level Order Traversal:\n");
    levelOrder(root);
    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");
    return 0;}
```

OUTPUT:



25.DFS and BFS USING ADJACENCY LIST(USE STACKS AND QUEUES IN PROGRAM).

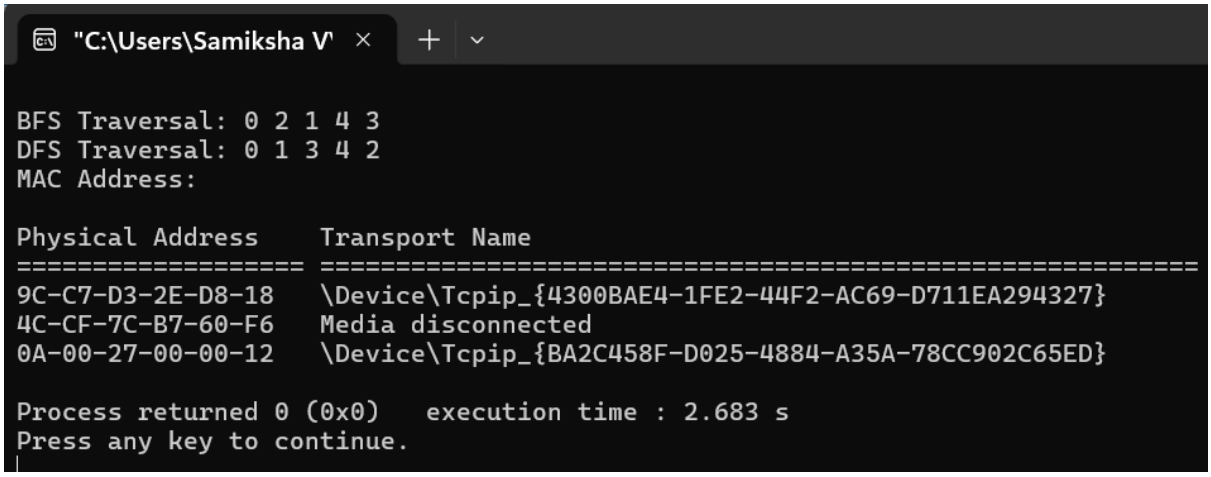
```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
struct Node {
    int data;
    struct Node* next;
```

```
};
struct Graph {
    struct Node* adjLists[MAX];
    int visited[MAX];
};
// Create a new node
struct Node* createNode(int v) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->data = v;
    newNode->next = NULL;
    return newNode;
}
// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
// BFS Implementation (Using Queue)
void BFS(struct Graph* graph, int startNode, int n) {
    int queue[MAX], front = 0, rear = 0;

    for (int i = 0; i < n; i++) graph->visited[i] = 0;
    graph->visited[startNode] = 1;

    queue[rear++] = startNode;
```

```
    printf("\nBFS Traversal: ");
    while (front < rear) {
        int current = queue[front++];
        printf("%d ", current);

        struct Node* temp = graph->adjLists[current];
        while (temp) {
            if (!graph->visited[temp->data]) {
                graph->visited[temp->data] = 1;
                queue[rear++] = temp->data;
            }
            temp = temp->next; }}
}

// DFS Implementation (Using Stack)
void DFS(struct Graph* graph, int startNode, int n) {
    int stack[MAX], top = -1;

    for (int i = 0; i < n; i++) graph->visited[i] = 0;
    stack[++top] = startNode;

    printf("\nDFS Traversal: ");
    while (top != -1) {
        int current = stack[top--];
        if (!graph->visited[current]) {
            printf("%d ", current);
            graph->visited[current] = 1;
        }
        struct Node* temp = graph->adjLists[current];
```

```
        while (temp) {
            if (!graph->visited[temp->data]) {
                stack[++top] = temp->data;
            }
            temp = temp->next;}}
}
int main() {
    int n = 5; // Number of vertices
    struct Graph* graph = malloc(sizeof(struct Graph));
    for (int i = 0; i < n; i++) graph->adjLists[i] = NULL;

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);

    BFS(graph, 0, n);
    DFS(graph, 0, n);
    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");

    return 0;
}
```

OUTPUT:

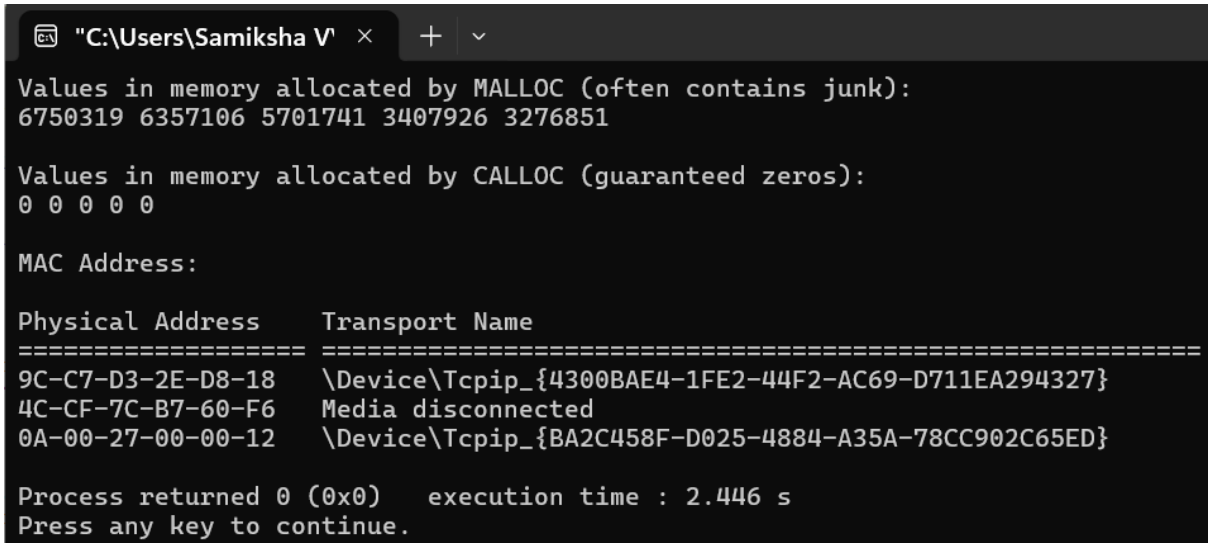


26.CALLOC AND MALLOC FUNCTIONS CHECKING IF JUNK/ZERO IS INITIALIZED TO THEM

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n = 5;
    int *mPtr, *cPtr;
    // 1. MALLOC: Allocates memory but DOES NOT initialize it
    mPtr = (int*)malloc(n * sizeof(int));
    // 2. CALLOC: Allocates memory and INITIALIZES all bits to ZERO
    cPtr = (int*)calloc(n, sizeof(int));
    // Check for allocation failure
    if (mPtr == NULL || cPtr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    printf("Values in memory allocated by MALLOC (often contains junk):\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", mPtr[i]); }
    printf("\n\nValues in memory allocated by CALLOC (guaranteed zeros):\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", cPtr[i]);    }
    Printf("\n");
```

```
    free(mPtr);
    free(cPtr);
    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");
    return 0;
}
```

OUTPUT:



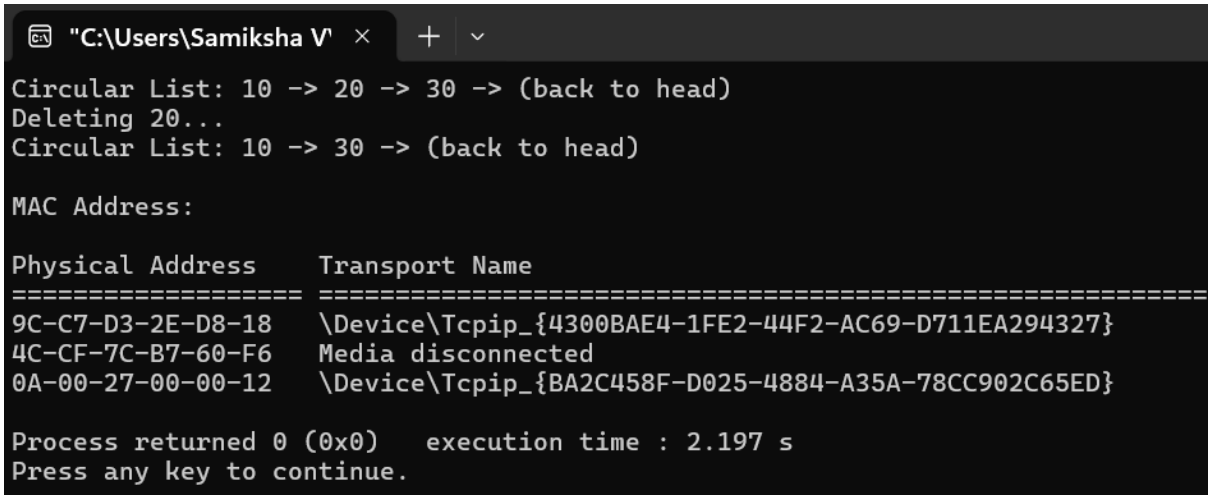
27. CIRCULAR LINKED LIST BASIC OPERATIONS.

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
// Insert at the beginning
struct Node* insert(struct Node* tail, int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;

    if (tail == NULL) {
```

```
        newNode->next = newNode; // Points to itself
        return newNode;
    } else {
        newNode->next = tail->next;
        tail->next = newNode;
        return tail;
}
}
// Delete a node by value
struct Node* deleteNode(struct Node* tail, int key) {
    if (tail == NULL) return NULL;
    struct Node *curr = tail->next, *prev = tail;
{
        if (curr->data == key) {
            if (curr == tail && curr->next == tail) { // Only one node
                free(curr);
                return NULL;
 }
            prev->next = curr->next;
            if (curr == tail) tail = prev;

            free(curr);
            return tail;
        }
        prev = curr;
        curr = curr->next;
    } while (curr != tail->next);

    printf("Node %d not found.\n", key);
    return tail;
```

```
}
// Display the list
void display(struct Node* tail) {
    if (tail == NULL) {
        printf("List is empty.\n");
        return; }
    struct Node* temp = tail->next; // Start from head
    printf("Circular List: ");
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != tail->next);
    printf("(back to head)\n");
}
int main() {
    struct Node* tail = NULL;
    tail = insert(tail, 30);
    tail = insert(tail, 20);
    tail = insert(tail, 10);
    display(tail);
    printf("Deleting 20...\n");
    tail = deleteNode(tail, 20);
    display(tail);
    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");
    return 0;
}
```

OUTPUT:

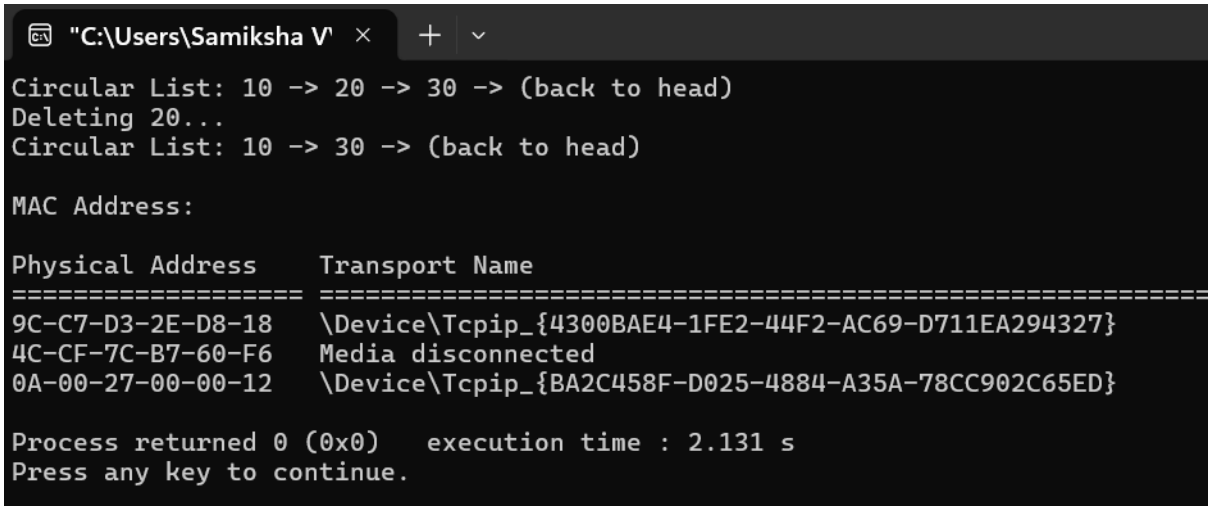


28. how to implement comaprision of 2 strings using built in function

```
#include <stdio.h>
#include <string.h> // Required for strcmp()

int main() {
    char str1[] = "apple";
    char str2[] = "apple";
    char str3[] = "banana";

    // Comparing str1 and str2 (Identical)
    if (strcmp(str1, str2) == 0) {
        printf("str1 and str2 are equal.\n");
    }

    // Comparing str1 and str3 (Different)
    int result = strcmp(str1, str3);
    if (result < 0) {
        printf("\"%s\" comes before \"%s\" lexicographically.\n", str1, str3);
    } else if (result > 0) {
        printf("\"%s\" comes after \"%s\" lexicographically.\n", str1, str3);
    }
    // Display MAC Address (Windows only)
```

```
    printf("\nMAC Address:\n");
    system("getmac");

    return 0;
}
```

OUTPUT:



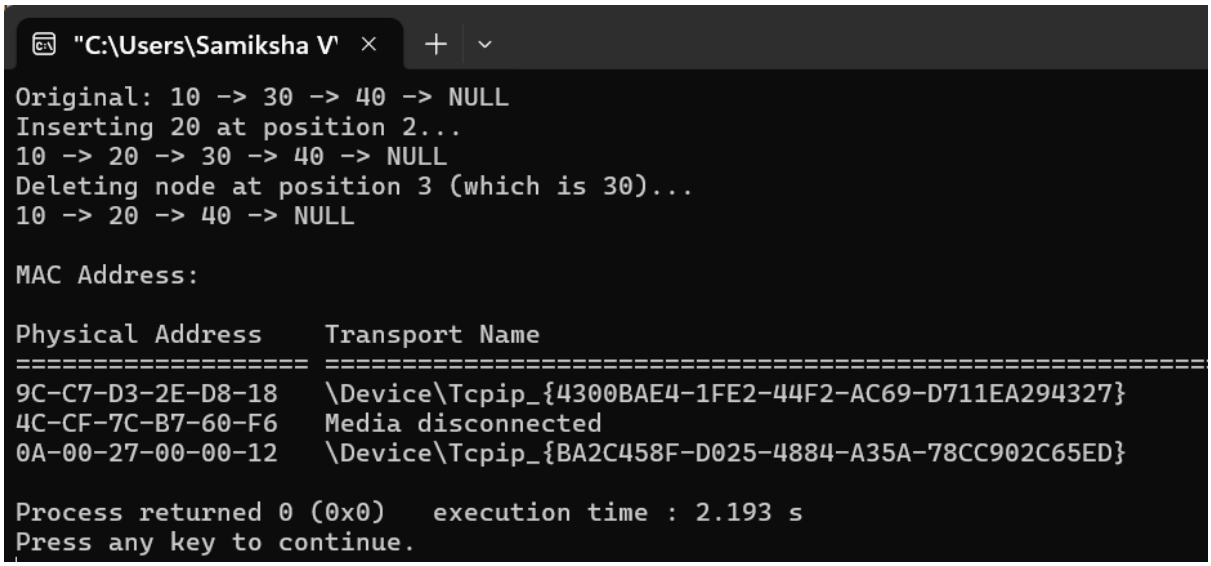
29.In linked list insertion in the middle and deletion in the middle.

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

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

// Function to print the list
void display(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
```

```
}

// 1. INSERT AT MIDDLE (at a given position)
void insertMiddle(struct Node* head, int val, int pos) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    struct Node* temp = head;

    // Move to the node just before the insertion point
    for (int i = 1; i < pos - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range\n");
    } else {
        newNode->next = temp->next;
        temp->next = newNode;
    }
}
// 2. DELETE FROM MIDDLE (at a given position)
void deleteMiddle(struct Node** head, int pos) {
    if (*head == NULL) return;

    struct Node* temp = *head;
    // Special case: deleting the head
    if (pos == 1) {
        *head = temp->next;
        free(temp);
        return;
    }
```

```
    // Move to the node just before the one to be deleted
    for (int i = 1; temp != NULL && i < pos - 1; i++) {
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL) {
        printf("Position out of range\n");
    } else {
        struct Node* nextNode = temp->next->next;
        free(temp->next); // Free the target node
        temp->next = nextNode; // Link previous to next-next
    }
}
int main() {
    // Creating a simple list: 10 -> 30 -> 40
    struct Node* head = (struct Node*)malloc(sizeof(struct Node));

    head->data = 10;
    head->next = (struct Node*)malloc(sizeof(struct Node));
    head->next->data = 30;
    head->next->next = (struct Node*)malloc(sizeof(struct Node));
    head->next->next->data = 40;
    head->next->next->next = NULL;

    printf("Original: ");
    display(head);

    printf("Inserting 20 at position 2...\n");
    insertMiddle(head, 20, 2);
    display(head);
```

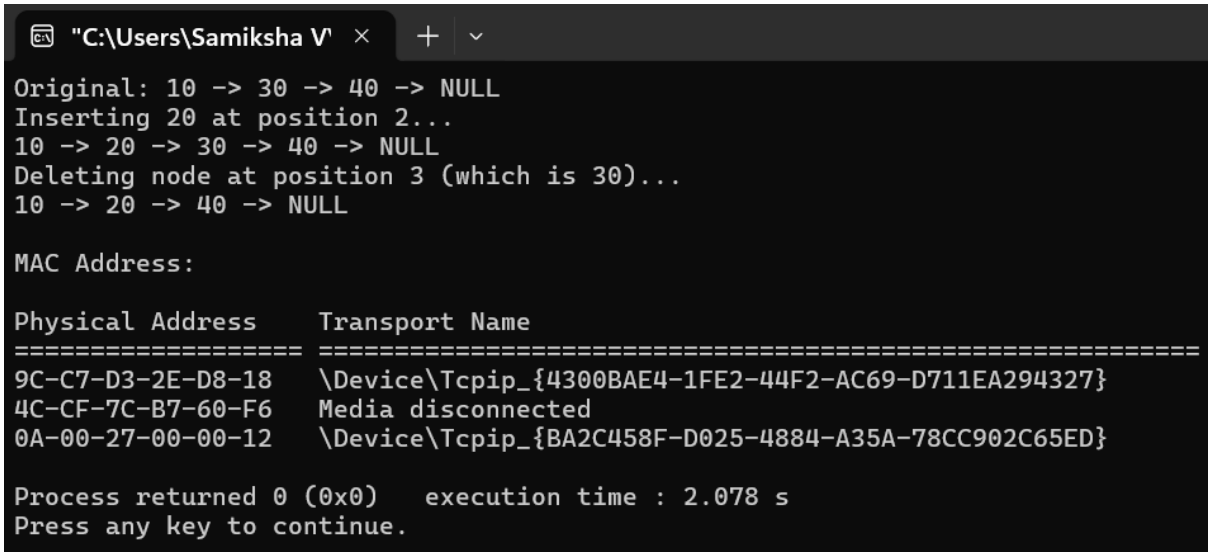
```
    printf("Deleting node at position 3 (which is 30)...\n");

    deleteMiddle(&head, 3);
    display(head);

    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");
    return 0;
}
```



30.BINARY TREE TRAVERSAL.

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

// 1. Define the Node
struct Node {
    int data;
    struct Node *left, *right;
};
// 2. Simple Inorder Traversal (Recursion)
void printTree(struct Node* root) {
```

```
    if (root == NULL) return;     // Stop if node is empty
    printTree(root->left);        // Go Left
    printf("%d ", root->data);    // Print Root
    printTree(root->right);       // Go Right
}
// 3. Simple Node Creator
struct Node* add(int val) {
    struct Node* n = malloc(sizeof(struct Node));
    n->data = val;
    n->left = n->right = NULL;
    return n;
}
int main() {
    // Manually build a small tree:
    //      10
    //     / \
    //    5   20
    struct Node* root = add(10);
    root->left = add(5);
    root->right = add(20);

    printf("Inorder Traversal: ");
    printTree(root);
    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");

    return 0;
}
```



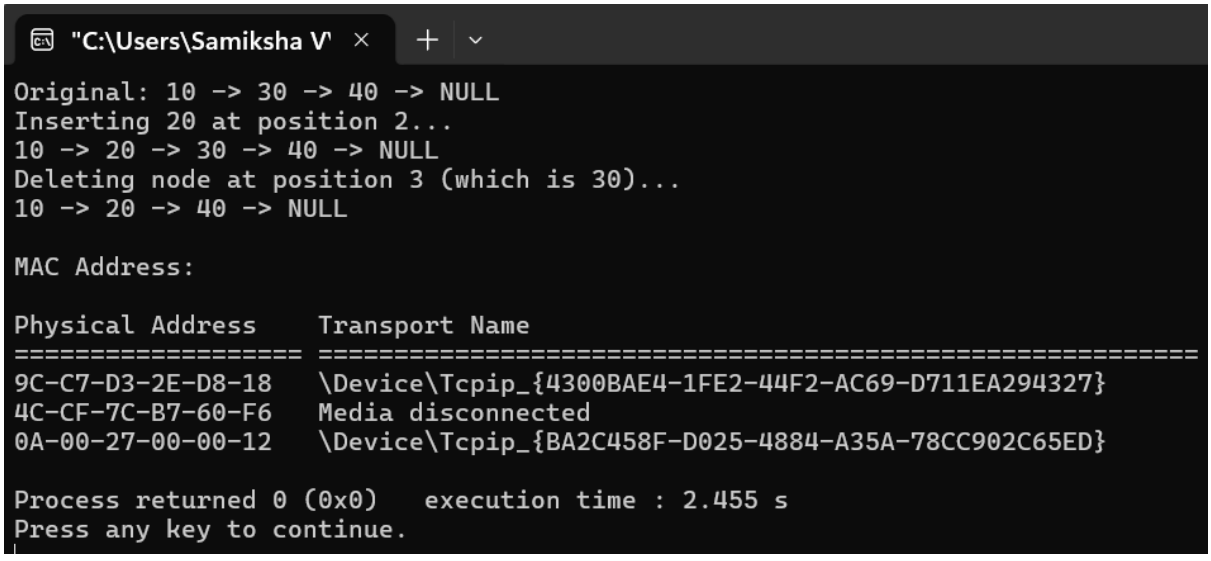
31.CIRCULAR QUEUES USING ARRAY(MODULO DIVISION,QUEUE FULL,QUEUE EMPTY).

```
#include <stdio.h>
#define MAX 5  // Size of the queue
int queue[MAX];
int front = -1, rear = -1;
// 1. Check if Queue is FULL
int isFull() {
   return (rear + 1) % MAX == front;
}

// 2. Check if Queue is EMPTY
int isEmpty() {
   return front == -1;
}
// 3. ENQUEUE (Add item)
void enqueue(int value) {
   if (isFull()) {
      printf("Queue Full!\n");
      return;
   }
```

```
    if (isEmpty()) front = 0; // First element
    rear = (rear + 1) % MAX;  // Circular increment
    queue[rear] = value;
    printf("Inserted: %d\n", value);
// 4. DEQUEUE (Remove item)
void dequeue() {
    if (isEmpty()) {
        printf("Queue Empty!\n");
        return;
    }
    printf("Deleted: %d\n", queue[front]);
    if (front == rear) { // Reset if only one element was left
        front = rear = -1;
    } else {
        front = (front + 1) % MAX; // Circular increment
    }
}
// 5. DISPLAY
void display() {
    if (isEmpty()) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue: ");
    int i = front;
    while (1) {
        printf("%d ", queue[i]);
        if (i == rear) break;
        i = (i + 1) % MAX;
    }
```

```
    printf("\n");
}
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    enqueue(40);
    enqueue(50); // Queue is now full

    display();

    dequeue();
    dequeue();

    enqueue(60); // This works because of modulo division!
    display();

    return 0;
}
```

OUTPUT:

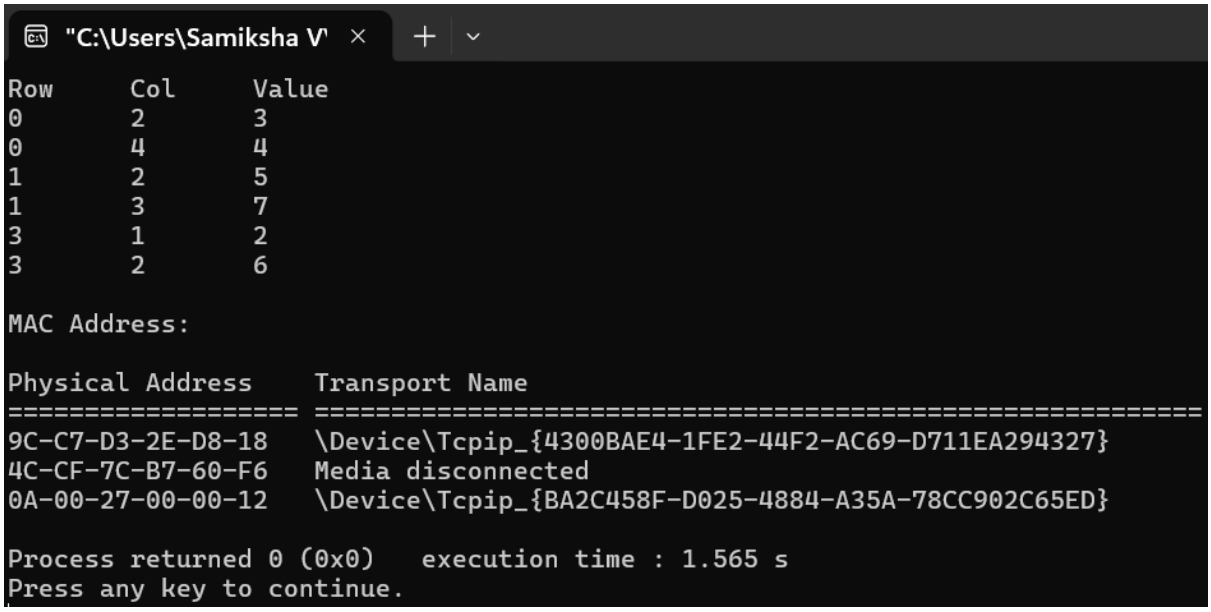


32.SPARSE MATRIX REPRESENTATION USING LINKED LIST.

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

struct Node {
    int row, col, value;
    struct Node* next;
};

// Function to create a new node and add it to the list
void create_node(struct Node** head, int r, int c, int val) {
    struct Node *temp, *ptr;
    temp = (struct Node*)malloc(sizeof(struct Node));
    temp->row = r;
    temp->col = c;
    temp->value = val;
    temp->next = NULL;

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

// Function to display the linked list representation
void display(struct Node* head) {
    struct Node* temp = head;
```

```
    printf("Row\tCol\tValue\n");
    while (temp != NULL) {
        printf("%d\t%d\t%d\n", temp->row, temp->col, temp->value);
        temp = temp->next;
    }
}

int main() {
    int matrix[4][5] = {
        {0, 0, 3, 0, 4},
        {0, 0, 5, 7, 0},
        {0, 0, 0, 0, 0},
        {0, 2, 6, 0, 0}
    };

    struct Node* head = NULL;
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 5; j++) {
            if (matrix[i][j] != 0) {
                create_node(&head, i, j, matrix[i][j]);
            }
        }
    }
    display(head);
    // Display MAC Address (Windows only)
    printf("\nMAC Address:\n");
    system("getmac");
    return 0;
}
```

OUTPUT:


"C:\Users\Samiksha V\
Row Col Value
0 2 3
0 4 4
1 2 5
1 3 7
3 1 2
3 2 6
MAC Address:
Physical Address Transport Name
9C-C7-D3-2E-D8-18 \Device\Tcpip_{4300BAE4-1FE2-44F2-AC69-D711EA294327}
4C-CF-7C-B7-60-F6 Media disconnected
0A-00-27-00-00-12 \Device\Tcpip_{BA2C458F-D025-4884-A35A-78CC902C65ED}
Process returned 0 (0x0) execution time : 1.565 s
Press any key to continue.