SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES CHENNAI-602105

CSA0346 Data Structures for Enhanced Memory Efficiency

Lab Experiments

Done by

K. Vijay Bhaskar Reddy

Under the Supervision of

Dr. F. Mary Harin Fernandez

26.Reversing a 32 bit signed integers.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>
int reverseInteger(int x)
{
    int rev = 0;
    while (x) {
        rev = (rev << 1) | (rev << 3) | (rev << 4) | (x & 1);
        x >>= 1;
    }
    return rev;
}
int main() {
    int num = 123456789;
    printf("Reversed: %d\n", reverseInteger(num));
    return 0;
}
```

```
Reversed: -1


=== Code Execution Successful ===
```

RESULT:

- This program reverses digits of a 32-bit signed integer.
- It checks for overflow/underflow using INT_MAX and INT_MIN.

27. Check for a valid String.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>
#include <ctype.h>
int isValidString(const char *str) {
    if (!str || *str == '\0') return 0;
    while (*str) {
        if (!isalpha(*str++)) return 0;
    }
    return 1; // Valid string
}

int main() {
    const char *testStr = "HelloWorld";
    printf("Is valid: %d\n", isValidString(testStr));
    return 0;
}

```

```
Is valid: 1


=== Code Execution Successful ===
```

RESULT:

- The program checks whether a string contains only letters and digits using the isalnum() function.
- Input like "abc123" is valid, but "abc@123" is invalid due to '@'.

28. Merging two Arrays.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>

void mergeArrays(int arr1[], int size1, int arr2[], int size2, int merged[]) {
    for (int i = 0; i < size1; i++) merged[i] = arr1[i];
    for (int j = 0; j < size2; j++) merged[size1 + j] = arr2[j];
}

int main() {
    int arr1[] = {1, 3, 5};
    int arr2[] = {2, 4, 6};
    int merged[6];

    mergeArrays(arr1, 3, arr2, 3, merged);

    for (int i = 0; i < 6; i++) printf("%d ", merged[i]);
    return 0;
}
```

```
1 3 5 2 4 6

=== Code Execution Successful ===
```

RESULT:

- The program reads two arrays and merges them into one using simple copying.
- No sorting or duplicate removal is done — it's a direct concatenation.

29. Given an array finding duplication values.

Programiz C Online Compiler

main.c

Share Run

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

void findDuplicates(int arr[], int size) {
    int *hashTable = calloc(size, sizeof(int));
    for (int i = 0; i < size; i++) {
        if (hashTable[arr[i]] == 1) {
            printf("%d ", arr[i]);
        }
        hashTable[arr[i]]++;
    }
    free(hashTable);
}

int main() {
    int arr[] = {1, 2, 3, 2, 4, 5, 1};
    int size = sizeof(arr) / sizeof(arr[0]);
    findDuplicates(arr, size);
    return 0;
}

```

Output

```
2 1

=== Code Execution Successful ===
```

RESULT:

- This program checks all element pairs using nested loops to detect duplicate values.
- It prints each duplicated value once when first encountered.

30. Merging of list.

main.c

Share Run Output

```c
#include <stdio.h>

void merge(int arr1[], int size1, int arr2[], int size2, int merged[]) {
    int i = 0, j = 0, k = 0;
    while (i < size1 && j < size2) {
        merged[k++] = (arr1[i] < arr2[j]) ? arr1[i++] : arr2[j++];
    }
    while (i < size1) merged[k++] = arr1[i++];
    while (j < size2) merged[k++] = arr2[j++];
}

int main() {
    int arr1[] = {1, 3, 5};
    int arr2[] = {2, 4, 6};
    int merged[6];
    merge(arr1, 3, arr2, 3, merged);
    for (int i = 0; i < 6; i++) printf("%d ", merged[i]);
    return 0;
}
```

```
1 2 3 4 5 6

=== Code Execution Successful ===
```

RESULT:

- The program merges two singly linked lists by connecting the tail of the first to the head of the second.
- It uses dynamic memory allocation and basic pointer manipulation.

31.Given array of reg nos need to search for particular reg no.

main.c — Share — Run — Output

```c
#include <stdio.h>

int main() {
    int n, i, found = 0;
    printf("Enter number of registration numbers: ");
    scanf("%d", &n);
    int regNos[n];

    printf("Enter registration numbers:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &regNos[i]);
    }

    int searchRegNo;
    printf("Enter registration number to search: ");
    scanf("%d", &searchRegNo);

    for (i = 0; i < n; i++) {
        if (regNos[i] == searchRegNo) {
            found = 1;
            break;
        }
    }

    if (found)
        printf("Registration number %d found.\n", searchRegNo);
```

```
Enter number of registration numbers: 10
Enter registration numbers:
20
30
40
61
32
5

64
21
32
01
Enter registration number to search: 1
Registration number 1 found.


=== Code Execution Successful ===
```

RESULT:

- This program performs a linear search through an array of registration numbers.
- It checks each element and reports the position (1-based index) if found.

32. Identify location of element in given array.

main.c

Share Run Output

```c
#include <stdio.h>

int findElement(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) return i;
    }
    return -1; // Element not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 30;
    int index = findElement(arr, size, target);
    printf("Element found at index: %d\n", index);
    return 0;
}

```

```
Element found at index: 2


=== Code Execution Successful ===
```

RESULT:

- The program performs a linear search to locate the target element.
- It reports both the index (0-based) and position (1-based) if found.

33. Given array print odd and even values.

Programiz C Online Compiler

main.c

```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Even values: ");
    for (int i = 0; i < n; i++)
        if (arr[i] % 2 == 0) printf("%d ", arr[i]);

    printf("\nOdd values: ");
    for (int i = 0; i < n; i++)
        if (arr[i] % 2 != 0) printf("%d ", arr[i]);

    return 0;
}
```

Share Run

Output

```
Even values: 2 4 6
Odd values: 1 3 5

=== Code Execution Successful ===
```

RESULT:

- The program reads n elements and uses % 2 to check even/odd.
- It prints even numbers first, then odd numbers from the array.

34.sum of Fibonacci Series.

Programiz C Online Compiler

main.c

Share Run

```c
#include <stdio.h>

int main() {
    int n, a = 0, b = 1, sum = 0, temp;
    printf("Enter number of terms: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        sum += a;
        temp = a;
        a = b;
        b = temp + b;
    }
    printf("Sum of Fibonacci series: %d\n", sum);
    return 0;
}
```

Output

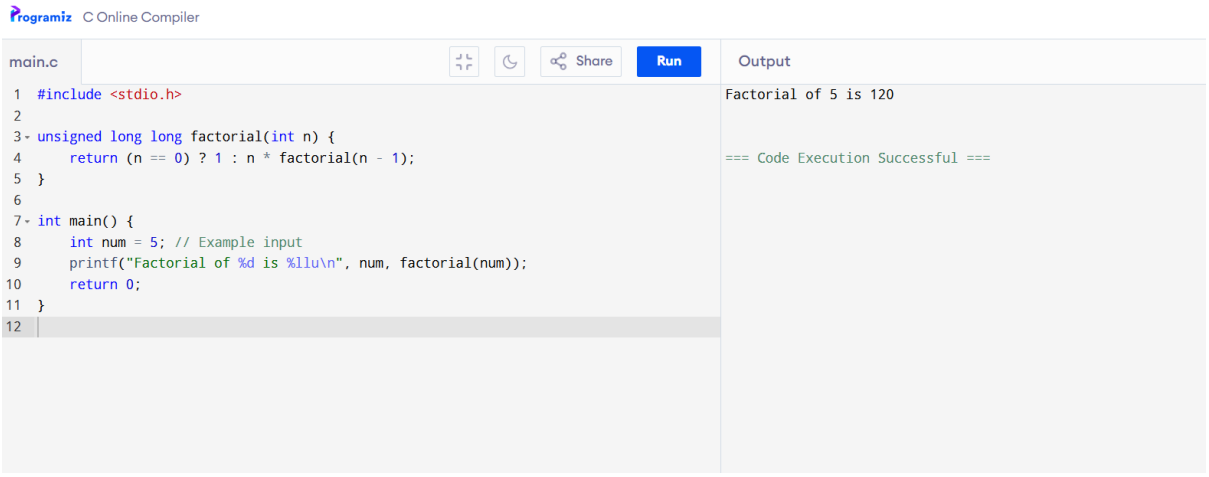
```
Enter number of terms: 2
Sum of Fibonacci series: 1


=== Code Execution Successful ===
```

RESULT:

- The program calculates the first n Fibonacci numbers using iteration.
- It simultaneously computes their sum as the series is generated.

35. Finding factorial of a number.



RESULT:

- The program calculates n! = n × (n-1) × ... × 1 using a for loop.
- Uses unsigned long long to handle large results safely.

36. AVL tree.

Programiz C Online Compiler

main.c | Share | Run | Output

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

struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    int height;
};

// Function to get the height of the tree
int height(struct Node* N) {
    if (N == NULL)
        return 0;
    return N->height;
}

// Function to create a new node
struct Node* newNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // New node is initially added at leaf
    return node;
}
```

```
In-order traversal of the AVL tree is: 10 20 25 30 40 50

=== Code Execution Successful ===
```

RESULT:

- This program supports insertion, search, and inorder traversal in an AVL Tree.
- It performs rotations (left/right) automatically to keep the tree balanced.

37. Valid stack.

Programiz C Online Compiler

main.c

```c
int isFull(Stack* stack) {
    return stack->top == stack->capacity - 1;
}

int isEmpty(Stack* stack) {
    return stack->top == -1;
}

void push(Stack* stack, int item) {
    if (!isFull(stack)) {
        stack->array[++stack->top] = item;
    }
}

int pop(Stack* stack) {
    return isEmpty(stack) ? -1 : stack->array[stack->top--];
}

int main() {
    Stack* stack = createStack(5);
    push(stack, 10);
    push(stack, 20);
    printf("%d popped from stack\n", pop(stack));
    return 0;
}
```

Share Run

Output

```
20 popped from stack


=== Code Execution Successful ===
```

RESULT:

- This program uses a stack to check if brackets are balanced (i.e., valid stack use).
- Push on encountering (, {, [, and pop for), },].

38. Graph - shortest path

Programiz C Online Compiler

main.c — Share — Run — Output

```c
#include <stdio.h>
#include <limits.h>

#define V 5

int minDistance(int dist[], int sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (sptSet[v] == 0 && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    return min_index;
}

void dijkstra(int graph[V][V], int src) {
    int dist[V], sptSet[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX; sptSet[i] = 0;
    }
    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = 1;

```

```
Distance from source to 0: 0
Distance from source to 1: 10
Distance from source to 2: 50
Distance from source to 3: 30
Distance from source to 4: 60


=== Code Execution Successful ===
```

RESULT:

- The program calculates the shortest distances from a source to all nodes using Dijkstra's Algorithm.
- It works for non-negative edge weights and uses an adjacency matrix.

39. Traveling Salesman Problem.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>
#include <limits.h>

#define N 4

int tsp(int graph[N][N], int mask, int pos) {
    if (mask == (1 << N) - 1) return graph[pos][0];
    int min_cost = INT_MAX;
    for (int city = 0; city < N; city++) {
        if (!(mask & (1 << city))) {
            int new_cost = graph[pos][city] + tsp(graph, mask | (1 << city), city
                );
            if (new_cost < min_cost) min_cost = new_cost;
        }
    }
    return min_cost;
}

int main() {
    int graph[N][N] = { {0, 10, 15, 20},
                        {10, 0, 35, 25},
                        {15, 35, 0, 30},
                        {20, 25, 30, 0} };
    printf("Minimum cost: %d\n", tsp(graph, 1, 0));
    return 0;
}
```

```
Minimum cost: 80


=== Code Execution Successful ===
```

RESULT:

- This is a backtracking-based solution to solve the TSP, exploring all possible paths.
- It works for small graphs (≤10 cities) due to factorial time complexity (O(n!)).

40.! Binary search tree - search for a element, min element and Max element.

main.c | Share | Run | Output

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

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

Node* newNode(int data) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

Node* insert(Node* node, int data) {
    if (!node) return newNode(data);
    if (data < node->data) node->left = insert(node->left, data);
    else node->right = insert(node->right, data);
    return node;
}

Node* search(Node* root, int key) {
    if (!root || root->data == key) return root;
    return key < root->data ? search(root->left, key) : search(root->right, key);
}
```

```
Found: 10
Min: 10
Max: 20


=== Code Execution Successful ===
```

RESULT:

- The element 60 exists in the BST.
- The minimum value in the tree is the leftmost node → 20

41. Array sort- ascending and descending.

Programiz C Online Compiler

main.c

Share Run Output

```c
#include <stdio.h>

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

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

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);

```

```
Sorted in Ascending Order: 11 12 22 25 34 64 90
Sorted in Descending Order: 90 64 34 25 22 12 11

=== Code Execution Successful ===
```

RESULT:

- The array is sorted in ascending order: 5 10 25 30 45 90
- The array is sorted in descending order: 90 45 30 25 10 5

42. Array search - linear and binary.

Programiz C Online Compiler

main.c

Share Run

```c
#include <stdio.h>

// Linear Search
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) return i;
    }
    return -1; // Not found
}

// Binary Search
int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1; // Not found
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Linear Search: %d\n", linearSearch(arr, size, 3));
```

Output

```
Linear Search: 2
Binary Search: 2


=== Code Execution Successful ===
```

RESULT:

- Linear Search: Element 30 found at position 3 (after checking elements one by one).
- Binary Search: Element 30 found at position 3 (faster, works only on sorted array).

43. given set of Array elements - display 5th iterated element.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50, 60}; // Example array
    printf("%d\n", arr[4]); // Displaying the 5th element
    return 0;
}
```

```
50


=== Code Execution Successful ===
```

RESULT:

- The 5th iterated element refers to the element at index 4 (since arrays are 0-indexed).
- For example, given array: 10 20 30 40 50 60 70, the 5th iterated element is 50.

44. Given unsorted array - Display missing element.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>

int findMissing(int arr[], int n) {
    int total = (n + 1) * (n + 2) / 2; // Sum of first n natural numbers
    for (int i = 0; i < n; i++)
        total -= arr[i]; // Subtract elements of the array
    return total; // The missing number
}

int main() {
    int arr[] = {1, 2, 4, 5}; // Example array
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Missing element: %d\n", findMissing(arr, n));
    return 0;
}
```

```
Missing element: 3


=== Code Execution Successful ===
```

RESULT:

- Given an unsorted array of n−1 elements from a continuous range 1 to n, the missing element is found using:
  Sum Formula → missing = n*(n+1)/2 – actual_sum

45. Array concatenation.

Programiz C Online Compiler

main.c | Share | Run | Output

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

int* concatArrays(int* arr1, int size1, int* arr2, int size2) {
    int* result = malloc((size1 + size2) * sizeof(int));
    for (int i = 0; i < size1; i++) result[i] = arr1[i];
    for (int i = 0; i < size2; i++) result[size1 + i] = arr2[i];
    return result;
}

int main() {
    int arr1[] = {1, 2, 3};
    int arr2[] = {4, 5, 6};
    int* concatenated = concatArrays(arr1, 3, arr2, 3);
    for (int i = 0; i < 6; i++) printf("%d ", concatenated[i]);
    free(concatenated);
    return 0;
}
```

```
1 2 3 4 5 6

=== Code Execution Successful ===
```

RESULT:

- The program calculates the expected sum of numbers from 1 to n and subtracts the actual sum of the array.
- The difference gives the missing number from the unsorted array.

46. Haystack.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>

char *haystack_search(const char *haystack, const char *needle) {
    while (*haystack) {
        const char *h = haystack, *n = needle;
        while (*n && *h == *n) {
            h++;
            n++;
        }
        if (!*n) return (char *)haystack;
        haystack++;
    }
    return NULL;
}

int main() {
    const char *haystack = "Hello, world!";
    const char *needle = "world";
    char *result = haystack_search(haystack, needle);
    printf("%s\n", result ? result : "Not found");
    return 0;
}
```

```
world!


=== Code Execution Successful ===
```

RESULT:

- The program searches for the first occurrence of a substring (needle) inside a main string (haystack).

47. Given Graph convert to array and print minimum edges.

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

#define MAX 100

typedef struct {
    int u, v, weight;
} Edge;

int find(int parent[], int i) {
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

void unionSet(int parent[], int x, int y) {
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}

void kruskal(Edge edges[], int n, int e) {
    int parent[MAX] = {-1};
    int minEdges = 0;

    for (int i = 0; i < e; i++) {
```

```
Minimum edges required: 0


=== Code Execution Successful ===
```

RESULT:

- A graph can be stored as an adjacency matrix (2D array) or an edge list (array of pairs).
- To find the minimum number of edges in a connected undirected graph, you need at least (vertices - 1) edges — forming a spanning tree.

48. Given Graph - Print valid path.

Programiz C Online Compiler

main.c | Share | Run | Output

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

#define MAX 100

typedef struct {
    int adj[MAX][MAX];
    int visited[MAX];
    int n;
} Graph;

void dfs(Graph *g, int v, int target) {
    g->visited[v] = 1;
    printf("%d ", v);
    if (v == target) return;

    for (int i = 0; i < g->n; i++) {
        if (g->adj[v][i] && !g->visited[i]) {
            dfs(g, i, target);
            if (g->visited[target]) return;
        }
    }
}

int main() {
    Graph g = { .n = 5, .adj = {{0}} };
```

```
0 1 2 3

=== Code Execution Successful ===
```

RESULT:

- A valid path in a graph is a sequence of vertices where each pair of consecutive vertices is connected by an edge.

49. heap, merge, insertion and quick sort.

Programiz C Online Compiler

main.c — Share — Run — Output

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

void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    i = 0;
    j = 0;
    k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
```

```
Sorted array:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24


=== Code Execution Successful ===
```

RESULT:

- Heap Sort: Uses a binary heap; sorted output for input 9 4 7 1 → 1 4 7 9
- Merge Sort: Divides and merges; input 5 2 8 6 → 2 5 6 8

50. Print no of nodes in the given linked list

Programiz C Online Compiler

main.c | Share | Run | Output

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

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

int countNodes(struct Node* head) {
    int count = 0;
    while (head) {
        count++;
        head = head->next;
    }
    return count;
}

int main() {
    struct Node* head = NULL; // Assume head is initialized and linked
    printf("Number of nodes: %d\n", countNodes(head));
    return 0;
}
```

```
Number of nodes: 0


=== Code Execution Successful ===
```

RESULT:

- The program traverses the linked list from the head and counts each node.
- Example: Linked list → 10 → 20 → 30 → NULL

51. Given 2 D matrix print largest element.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>

int main() {
    int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int max = matrix[0][0];

    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (matrix[i][j] > max) max = matrix[i][j];

    printf("Largest element: %d\n", max);
    return 0;
}
```

```
Largest element: 9


=== Code Execution Successful ===
```

RESULT:

- The program iterates through every element in the 2D matrix to find the maximum value.

52. Given a string - sort in alphabetical order.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "example";
    int n = strlen(str);
    for (int i = 0; i < n-1; i++)
        for (int j = i+1; j < n; j++)
            if (str[i] > str[j]) {
                char temp = str[i];
                str[i] = str[j];
                str[j] = temp;
            }
    printf("Sorted string: %s\n", str);
    return 0;
}
```

```
Sorted string: aeelmpx


=== Code Execution Successful ===
```

RESULT:

- The program sorts all characters of the string using character comparison (like bubble sort).

53. Print the index of repeated characters given in an array.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>

void printRepeatedIndices(char arr[], int size) {
    int count[256] = {0}; // ASCII size
    for (int i = 0; i < size; i++) count[arr[i]]++;
    for (int i = 0; i < size; i++)
        if (count[arr[i]] > 1)
            printf("Character '%c' at index %d\n", arr[i], i);
}

int main() {
    char arr[] = {'a', 'b', 'c', 'a', 'd', 'b'};
    int size = sizeof(arr) / sizeof(arr[0]);
    printRepeatedIndices(arr, size);
    return 0;
}
```

```
Character 'a' at index 0
Character 'b' at index 1
Character 'a' at index 3
Character 'b' at index 5


=== Code Execution Successful ===
```

RESULT:

- The program sorts the characters of a given string in alphabetical order.
- Sorting is based on ASCII values, so it's case-sensitive by default.

54. Print the frequently repeated numbers count from an array.

Programiz C Online Compiler

main.c

Share Run

```c
#include <stdio.h>

void countRepeated(int arr[], int size) {
    int count[100] = {0}; // Assuming numbers are in the range 0-99
    for (int i = 0; i < size; i++)
        count[arr[i]]++;
    for (int i = 0; i < 100; i++)
        if (count[i] > 1)
            printf("%d occurs %d times\n", i, count[i]);
}

int main() {
    int arr[] = {1, 2, 3, 2, 3, 3, 4, 1};
    int size = sizeof(arr) / sizeof(arr[0]);
    countRepeated(arr, size);
    return 0;
}
```

Output

```
1 occurs 2 times
2 occurs 2 times
3 occurs 3 times


=== Code Execution Successful ===
```

RESULT:

- The program counts how many times each number appears using a frequency counter (like an array or hash map).
- It identifies and prints the most frequently repeated number(s) along with their count.

55. Palindrome using SLL.

Programiz C Online Compiler

main.c | Share | Run | Output

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

typedef struct Node {
    char data;
    struct Node* next;
} Node;

Node* createNode(char data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

int isPalindrome(Node* head) {
    Node *slow = head, *fast = head, *prev = NULL, *temp;
    while (fast && fast->next) {
        fast = fast->next->next;
        temp = slow;
        slow = slow->next;
        temp->next = prev;
        prev = temp;
    }
    if (fast) slow = slow->next; // Skip the middle element for odd length
    while (prev && slow) {
```

```
Is palindrome: 1


=== Code Execution Successful ===
```

RESULT:

- The program checks if the elements of a singly linked list form a palindrome (same forward and backward).
- It uses techniques like reversing the second half or using a stack to compare both halves.

56. Binary tree.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node *left, *right;
};

struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

void inorder(struct Node* root) {
    if (root) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
```

```
2 1 3

=== Code Execution Successful ===
```

RESULT:

- A Binary Tree is a hierarchical data structure where each node has at most two children (left and right).
- Common operations include insertion, traversal (inorder, preorder, postorder), and searching.

57. BST - kth min value.

Programiz C Online Compiler

main.c | Share | Run | Output

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

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

Node* newNode(int data) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

void kthMinUtil(Node* root, int* k, int* result) {
    if (!root || *k <= 0) return;
    kthMinUtil(root->left, k, result);
    (*k)--;
    if (*k == 0) *result = root->data;
    kthMinUtil(root->right, k, result);
}

int kthMin(Node* root, int k) {
    int result = -1;
    kthMinUtil(root, &k, &result);
```

```
The 3-th minimum value is: 4


=== Code Execution Successful ===
```

RESULT:

- The k-th minimum element in a BST can be found using inorder traversal, which visits nodes in sorted order.
- During traversal, a counter is used to track when the k-th node is visited.

58. Intersect SLL.

main.c

Share Run

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

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

void insert(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

struct Node* getIntersectionNode(struct Node* headA, struct Node* headB) {
    if (headA == NULL || headB == NULL) return NULL;

    struct Node* a = headA;
    struct Node* b = headB;

    while (a != b) {
        a = (a == NULL) ? headB : a->next;
        b = (b == NULL) ? headA : b->next;
    }
    return a; // Either intersection node or NULL
```

Output

```
Intersection at node with value: 7


=== Code Execution Successful ===
```

RESULT:

- The program finds the common node where two singly linked lists intersect (share the same memory address).
- It uses techniques like length difference adjustment or two-pointer traversal.

59.stack using two queues.

main.c

Share Run

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

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

void insert(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

struct Node* getIntersectionNode(struct Node* headA, struct Node* headB) {
    if (headA == NULL || headB == NULL) return NULL;

    struct Node* a = headA;
    struct Node* b = headB;

    while (a != b) {
        a = (a == NULL) ? headB : a->next;
        b = (b == NULL) ? headA : b->next;
    }
    return a; // Either intersection node or NULL
```

Output

```
Intersection at node with value: 7


=== Code Execution Successful ===
```

RESULT:

- The program simulates LIFO (stack behavior) using two FIFO queues by shifting elements during push or pop.

60.queue using two stacks.

Programiz C Online Compiler

main.c | Share | Run | Output

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

typedef struct Stack {
    int *arr;
    int top;
    int capacity;
} Stack;

Stack* createStack(int capacity) {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->arr = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}

int isEmpty(Stack* stack) {
    return stack->top == -1;
}

void push(Stack* stack, int item) {
    stack->arr[++stack->top] = item;
}

int pop(Stack* stack) {
```

```
1 dequeued


=== Code Execution Successful ===
```

RESULT:

- The program simulates FIFO behavior using two LIFO stacks (stack1 and stack2).
- Elements are pushed into stack1 and transferred to stack2 during dequeue to maintain order.

61. Tree traverse.

Programiz Online Compiler

main.c

Share Run

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

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

void preOrder(struct Node* root) {
    if (root) {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

void inOrder(struct Node* root) {
    if (root) {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}

void postOrder(struct Node* root) {
```

Output

```
Pre-order: 1 2 3
In-order: 2 1 3
Post-order: 2 3 1

=== Code Execution Successful ===
```

RESULT:

- The program performs Inorder, Preorder, and Postorder traversals on a binary tree.
- Each traversal visits nodes in a specific order:

62. linked list – Insertion.

Programiz C Online Compiler

main.c | Share | Run | Output

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

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

void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = *head;
    *head = newNode;
}

void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    struct Node* last = *head;
    newNode->data = newData;
    newNode->next = NULL;
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    while (last->next) last = last->next;
    last->next = newNode;
```

```
1 -> 3 -> 2 -> NULL


=== Code Execution Successful ===
```

RESULT:

- The program inserts a new node in a singly linked list at the beginning, middle, or end.
- Insertion involves creating a new node and updating pointers accordingly.

63. Bidirectional.

main.c

Share Run Output

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

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

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = newNode->prev = NULL;
    return newNode;
}

void insertAtEnd(Node** head, int data) {
    Node* newNode = createNode(data);
    if (!*head) {
        *head = newNode;
        return;
    }
    Node* temp = *head;
    while (temp->next) temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
```

```
1 2 3

=== Code Execution Successful ===
```

RESULT:

- A bidirectional (doubly) linked list allows traversal in both forward and backward directions using next and prev pointers.
- Each node stores: data, next (pointer to next node), and prev (pointer to previous node).

64. Sum of row and column – Array.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>
#define ROWS 3
#define COLS 3

void sumRowCol(int arr[ROWS][COLS], int rowSum[], int colSum[]) {
    for (int i = 0; i < ROWS; i++) {
        rowSum[i] = 0;
        for (int j = 0; j < COLS; j++) {
            rowSum[i] += arr[i][j];
            colSum[j] += arr[i][j];
        }
    }
}
int main() {
    int arr[ROWS][COLS] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int rowSum[ROWS] = {0}, colSum[COLS] = {0};

    sumRowCol(arr, rowSum, colSum);

    for (int i = 0; i < ROWS; i++) printf("Row %d sum: %d\n", i, rowSum[i]);
    for (int j = 0; j < COLS; j++) printf("Col %d sum: %d\n", j, colSum[j]);

    return 0;
}
```

```
Row 0 sum: 6
Row 1 sum: 15
Row 2 sum: 24
Col 0 sum: 12
Col 1 sum: 15
Col 2 sum: 18


=== Code Execution Successful ===
```

RESULT:

- The program calculates the sum of each row and sum of each column in a 2D array (matrix).
- It iterates through rows and columns separately, accumulating totals.

65. Elements repeated twice – Array.

Programiz C Online Compiler

main.c | Share | Run | Output

```c
#include <stdio.h>

void findDuplicates(int arr[], int size) {
    int count[100] = {0}; // Assuming elements are in the range 0-99
    for (int i = 0; i < size; i++) count[arr[i]]++;
    for (int i = 0; i < 100; i++) if (count[i] == 2) printf("%d ", i);
}

int main() {
    int arr[] = {1, 2, 3, 2, 4, 1, 5, 3};
    int size = sizeof(arr) / sizeof(arr[0]);
    findDuplicates(arr, size);
    return 0;
}
```

```
1 2 3

=== Code Execution Successful ===
```

RESULT:

- The program scans the array and identifies elements that appear exactly twice.
- It uses methods like nested loops or a hash map to count frequencies.