### 

### TEST CASES A.Mallesh 192425191

### 1. Reversing a 32-bit Signed Integer

**Aim:** To write a C program that reverses the digits of a 32-bit signed integer.

**Algorithm:**

1. Initialize a long long variable reverse to 0.
2. Loop through the input number x until it becomes 0.
3. In each iteration, get the last digit using the modulo operator (pop = x % 10).
4. Append the digit to reverse (reverse = reverse \* 10 + pop).
5. Remove the last digit from x (x /= 10).

**Code:**

#include <stdio.h>  
#include <limits.h>  
  
int reverse(int x) {  
 long long rev = 0;  
 while (x != 0) {  
 rev = rev \* 10 + x % 10;  
 x /= 10;  
 }  
 if (rev > INT\_MAX || rev < INT\_MIN) return 0;  
 return (int)rev;  
}  
  
int main() {  
 int num = 123;  
 printf("Original: %d\n", num);  
 printf("Reversed: %d\n", reverse(num));  
 return 0;  
}

**Output:**

Original: 123  
Reversed: 321

**Result:** The program successfully reverses the given 32-bit signed integer.

### 2. Check for a Valid String

**Aim:** To write a C program to check if a string contains only alphanumeric characters.

**Algorithm:**

1. Get the input string.
2. Iterate through each character of the string.
3. Use a condition to check if the character is not a letter (a-z, A-Z) and not a digit (0-9).
4. If such a character is found, the string is invalid.
5. If the loop completes without finding any such character, the string is valid.

**Code:**

#include <stdio.h>  
#include <ctype.h>  
#include <stdbool.h>  
  
bool isValid(char\* str) {  
 for (int i = 0; str[i] != '\0'; i++) {  
 if (!isalnum((unsigned char)str[i])) {  
 return false;  
 }  
 }  
 return true;  
}  
  
int main() {  
 char str1[] = "ValidString123";  
 char str2[] = "Invalid-String!";  
 printf("%s is %s\n", str1, isValid(str1) ? "Valid" : "Invalid");  
 printf("%s is %s\n", str2, isValid(str2) ? "Valid" : "Invalid");  
 return 0;  
}

**Output:**

ValidString123 is Valid  
Invalid-String! is Invalid

**Result:** The program successfully validates the given strings based on the alphanumeric criteria.

### 3. Merging Two Sorted Arrays

**Aim:** To write a C program to merge two sorted arrays into a single sorted array.

**Algorithm:**

1. Create a new array with a size equal to the sum of the two input array sizes.
2. Use three pointers, i for arr1, j for arr2, and k for the merged array.
3. Compare elements from arr1 and arr2 and place the smaller one into the merged array.
4. Increment the pointer of the array from which the element was taken.
5. After one array is exhausted, copy the remaining elements from the other array.

**Code:**

#include <stdio.h>  
  
void mergeArrays(int arr1[], int n1, int arr2[], int n2, int arr3[]) {  
 int i = 0, j = 0, k = 0;  
 while (i < n1 && j < n2) {  
 arr3[k++] = (arr1[i] < arr2[j]) ? arr1[i++] : arr2[j++];  
 }  
 while (i < n1) arr3[k++] = arr1[i++];  
 while (j < n2) arr3[k++] = arr2[j++];  
}  
  
int main() {  
 int arr1[] = {1, 3, 5};  
 int arr2[] = {2, 4, 6};  
 int n1 = 3, n2 = 3;  
 int arr3[n1 + n2];  
 mergeArrays(arr1, n1, arr2, n2, arr3);  
 for (int i = 0; i < n1 + n2; i++) printf("%d ", arr3[i]);  
 printf("\n");  
 return 0;  
}

**Output:**

1 2 3 4 5 6

**Result:** The program successfully merges two sorted arrays into a single sorted array.

### 4. Finding Duplicate Values in an Array

**Aim:** To write a C program to find and print duplicate values in a given array.

**Algorithm:**

1. Get the input array.
2. Use a nested loop to compare each element with all subsequent elements.
3. The outer loop selects an element.
4. The inner loop iterates from the next position of the outer loop's selection.
5. If a match is found, print the duplicate element.

**Code:**

#include <stdio.h>  
  
void findDuplicates(int arr[], int size) {  
 printf("Duplicate elements are: ");  
 for (int i = 0; i < size; i++) {  
 for (int j = i + 1; j < size; j++) {  
 if (arr[i] == arr[j]) {  
 printf("%d ", arr[i]);  
 }  
 }  
 }  
 printf("\n");  
}  
  
int main() {  
 int arr[] = {4, 2, 4, 5, 2, 3, 1};  
 int size = sizeof(arr) / sizeof(arr[0]);  
 findDuplicates(arr, size);  
 return 0;  
}

**Output:**

Duplicate elements are: 4 2

**Result:** The program successfully identifies and prints the duplicate values in the array.

### 5. Merging of Two Sorted Lists

**Aim:** To write a C program to merge two sorted singly linked lists.

**Algorithm:**

1. Create a dummy node to act as the starting point of the merged list.
2. Create a tail pointer, initially pointing to the dummy node.
3. Iterate while both lists have nodes.
4. Compare the data of the heads of both lists.
5. Attach the smaller node to tail->next and advance the head of that list.
6. After the loop, attach the remaining non-empty list to the end of the merged list.

**Code:**

#include <stdio.h>  
#include <stdlib.h>  
  
struct Node { int data; struct Node\* next; };  
  
void printList(struct Node\* n) {  
 while (n != NULL) { printf("%d ", n->data); n = n->next; }  
 printf("\n");  
}  
  
struct Node\* mergeLists(struct Node\* l1, struct Node\* l2) {  
 if (!l1) return l2;  
 if (!l2) return l1;  
 if (l1->data < l2->data) {  
 l1->next = mergeLists(l1->next, l2);  
 return l1;  
 } else {  
 l2->next = mergeLists(l1, l2->next);  
 return l2;  
 }  
}  
  
int main() {  
 struct Node\* l1 = &(struct Node){1, &(struct Node){3, &(struct Node){5, NULL}}};  
 struct Node\* l2 = &(struct Node){2, &(struct Node){4, &(struct Node){6, NULL}}};  
 struct Node\* merged = mergeLists(l1, l2);  
 printList(merged);  
 return 0;  
}

**Output:**

1 2 3 4 5 6

**Result:** The program successfully merges two sorted linked lists into a new sorted list.

### 6. Search for a Particular Registration Number

**Aim:** To write a C program that searches for a specific registration number (string) in an array of strings.

**Algorithm:**

1. Get the array of registration numbers and the target number to search.
2. Iterate through the array from the first element to the last.
3. In each iteration, compare the current registration number with the target using strcmp.
4. If strcmp returns 0, the strings are identical and the number is found.
5. If the loop completes without finding a match, the number is not in the array.

**Code:**

#include <stdio.h>  
#include <string.h>  
  
int searchRegNo(char\* arr[], int size, char\* target) {  
 for (int i = 0; i < size; i++) {  
 if (strcmp(arr[i], target) == 0) {  
 return i;  
 }  
 }  
 return -1;  
}  
  
int main() {  
 char\* regNos[] = {"AP01AB1234", "TS02CD5678", "KA03EF9012"};  
 char\* target = "TS02CD5678";  
 int index = searchRegNo(regNos, 3, target);  
 if (index != -1) {  
 printf("Registration number found at index %d.\n", index);  
 } else {  
 printf("Registration number not found.\n");  
 }  
 return 0;  
}

**Output:**

Registration number found at index 1.

**Result:** The program successfully performs a linear search for a given string in an array of strings.

### 7. Identify Location of Element in a Given Array

**Aim:** To write a C program to find the index (location) of a given element in an array using linear search.

**Algorithm:**

1. Get the array, its size, and the element to be searched.
2. Iterate through the array from the first element (index 0).
3. Compare each element of the array with the target element.
4. If a match is found, return the current index.
5. If the loop finishes and no match is found, return -1 to indicate failure.

**Code:**

#include <stdio.h>  
  
int linearSearch(int arr[], int n, int x) {  
 for (int i = 0; i < n; i++) {  
 if (arr[i] == x) {  
 return i;  
 }  
 }  
 return -1;  
}  
  
int main() {  
 int arr[] = {10, 20, 80, 30, 60, 50};  
 int x = 30;  
 int n = sizeof(arr) / sizeof(arr[0]);  
 int result = linearSearch(arr, n, x);  
 if (result != -1) {  
 printf("Element found at index %d.\n", result);  
 } else {  
 printf("Element not found.\n");  
 }  
 return 0;  
}

**Output:**

Element found at index 3.

**Result:** The program successfully finds and returns the location of the specified element in the array.

### 8. Print Odd and Even Values from an Array

**Aim:** To write a C program that separates and prints the odd and even numbers from a given array.

**Algorithm:**

1. Get the input array.
2. First, iterate through the array to find and print all even numbers.
3. For each element, use the modulo operator (% 2). If the result is 0, the number is even.
4. Next, iterate through the array again to find and print all odd numbers.
5. If the result of the modulo operation is not 0, the number is odd.

**Code:**

#include <stdio.h>  
  
void printOddEven(int arr[], int size) {  
 printf("Even numbers: ");  
 for (int i = 0; i < size; i++) {  
 if (arr[i] % 2 == 0) {  
 printf("%d ", arr[i]);  
 }  
 }  
 printf("\nOdd numbers: ");  
 for (int i = 0; i < size; i++) {  
 if (arr[i] % 2 != 0) {  
 printf("%d ", arr[i]);  
 }  
 }  
 printf("\n");  
}  
  
int main() {  
 int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};  
 int size = sizeof(arr) / sizeof(arr[0]);  
 printOddEven(arr, size);  
 return 0;  
}

**Output:**

Even numbers: 2 4 6 8   
Odd numbers: 1 3 5 7 9

**Result:** The program successfully segregates and prints the odd and even numbers from the array.

### 9. Sum of Fibonacci Series

**Aim:** To write a C program to calculate the sum of the Fibonacci series up to 'n' terms.

**Algorithm:**

1. Get the number of terms, n.
2. Handle the base cases where n is 0 or 1.
3. Initialize first two terms a = 0, b = 1 and total\_sum = 1.
4. Loop from the 3rd term up to n.
5. In each iteration, calculate the next Fibonacci number (next = a + b), add it to total\_sum, and update a and b.

**Code:**

#include <stdio.h>  
  
long long fibonacciSum(int n) {  
 if (n <= 0) return 0;  
 if (n == 1) return 0;  
 long long a = 0, b = 1, sum = 1, next;  
 for (int i = 2; i < n; i++) {  
 next = a + b;  
 sum += next;  
 a = b;  
 b = next;  
 }  
 return sum;  
}  
  
int main() {  
 int n = 10;  
 printf("Sum of Fibonacci series up to %d terms is %lld\n", n, fibonacciSum(n));  
 return 0;  
}

**Output:**

Sum of Fibonacci series up to 10 terms is 88

**Result:** The program successfully calculates the sum of the Fibonacci series for the given number of terms.

### 10. Finding Factorial of a Number

**Aim:** To write a C program to calculate the factorial of a given non-negative number.

**Algorithm:**

1. Get the input number n.
2. Initialize a variable factorial to 1.
3. If n is 0, the factorial is 1.
4. If n is greater than 0, loop from 1 to n.
5. In each iteration, multiply factorial by the loop counter.

**Code:**

#include <stdio.h>  
  
unsigned long long factorial(int n) {  
 if (n < 0) return 0; // Factorial is not defined for negative numbers  
 unsigned long long fact = 1;  
 for (int i = 1; i <= n; ++i) {  
 fact \*= i;  
 }  
 return fact;  
}  
  
int main() {  
 int num = 10;  
 printf("Factorial of %d is %llu\n", num, factorial(num));  
 return 0;  
}

**Output:**

Factorial of 10 is 3628800

**Result:** The program successfully calculates the factorial of the given number.

### 11. AVL Tree Insertion

**Aim:** To implement insertion in an AVL tree, which maintains its balance.

**Algorithm:**

1. Perform a standard Binary Search Tree (BST) insertion.
2. After insertion, update the height of the ancestor nodes.
3. Get the balance factor of each ancestor node (height of left subtree - height of right subtree).
4. If a node becomes unbalanced (balance factor > 1 or < -1), perform the appropriate rotation (LL, RR, LR, or RL) to restore balance.
5. Return the new root of the (sub)tree.

**Code:**

#include <stdio.h>  
#include <stdlib.h>  
  
struct Node { int key, height; struct Node \*left, \*right; };  
  
int height(struct Node \*N) { return N ? N->height : 0; }  
int max(int a, int b) { return (a > b) ? a : b; }  
  
struct Node\* newNode(int key) {  
 struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));  
 node->key = key; node->left = node->right = NULL; node->height = 1;  
 return node;  
}  
  
struct Node\* rightRotate(struct Node\* y) {  
 struct Node\* x = y->left; struct Node\* T2 = x->right;  
 x->right = y; y->left = T2;  
 y->height = max(height(y->left), height(y->right)) + 1;  
 x->height = max(height(x->left), height(x->right)) + 1;  
 return x;  
}  
  
struct Node\* leftRotate(struct Node\* x) {  
 struct Node\* y = x->right; struct Node\* T2 = y->left;  
 y->left = x; x->right = T2;  
 x->height = max(height(x->left), height(x->right)) + 1;  
 y->height = max(height(y->left), height(y->right)) + 1;  
 return y;  
}  
  
int getBalance(struct Node\* N) { return N ? height(N->left) - height(N->right) : 0; }  
  
struct Node\* insert(struct Node\* node, int key) {  
 if (!node) return newNode(key);  
 if (key < node->key) node->left = insert(node->left, key);  
 else if (key > node->key) node->right = insert(node->right, key);  
 else return node;  
 node->height = 1 + max(height(node->left), height(node->right));  
 int balance = getBalance(node);  
 if (balance > 1 && key < node->left->key) return rightRotate(node);  
 if (balance < -1 && key > node->right->key) return leftRotate(node);  
 if (balance > 1 && key > node->left->key) { node->left = leftRotate(node->left); return rightRotate(node); }  
 if (balance < -1 && key < node->right->key) { node->right = rightRotate(node->right); return leftRotate(node); }  
 return node;  
}  
  
void preOrder(struct Node \*root) {  
 if(root != NULL) { printf("%d ", root->key); preOrder(root->left); preOrder(root->right); }  
}  
  
int main() {  
 struct Node \*root = NULL;  
 root = insert(root, 10); root = insert(root, 20); root = insert(root, 30);  
 printf("Preorder traversal of the constructed AVL tree is \n");  
 preOrder(root); printf("\n");  
 return 0;  
}

**Output:**

Preorder traversal of the constructed AVL tree is   
20 10 30

**Result:** The program successfully inserts nodes into an AVL tree and maintains its balance through rotations.

### 12. Valid Stack (Balanced Parentheses)

**Aim:** To validate if a string of parentheses is balanced using a stack.

**Algorithm:**

1. Create a character stack.
2. Iterate through the input string character by character.
3. If the character is an opening bracket ((, {, [), push it onto the stack.
4. If it is a closing bracket (), }, ]), check if the stack is empty or if the top element is not the matching opening bracket. If so, the string is invalid.
5. After the loop, if the stack is empty, the string is balanced; otherwise, it is not.

**Code:**

#include <stdio.h>  
#include <stdbool.h>  
#define MAX 100  
  
char stack[MAX];  
int top = -1;  
  
void push(char item) { if(top < MAX-1) stack[++top] = item; }  
char pop() { if(top > -1) return stack[top--]; return '\0'; }  
char peek() { return stack[top]; }  
  
bool areBracketsBalanced(char\* exp) {  
 for (int i = 0; exp[i]; i++) {  
 if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{') push(exp[i]);  
 else if (exp[i] == ')' || exp[i] == ']' || exp[i] == '}') {  
 if (top == -1) return false;  
 char check = pop();  
 if ((exp[i] == ')' && check != '(') || (exp[i] == ']' && check != '[') || (exp[i] == '}' && check != '{')) return false;  
 }  
 }  
 return (top == -1);  
}  
  
int main() {  
 char exp[] = "{()}[]";  
 if (areBracketsBalanced(exp)) printf("Balanced\n");  
 else printf("Not Balanced\n");  
 return 0;  
}

**Output:**

Balanced

**Result:** The program successfully validates the balance of parentheses in a string using a stack.

### 13. Graph - Shortest Path (Dijkstra's Algorithm)

**Aim:** To find the shortest path from a source vertex to all other vertices in a weighted graph using Dijkstra's algorithm.

**Algorithm:**

1. Create a dist array to store shortest distances, initialized to infinity except for the source vertex (dist=0).
2. Create a sptSet (shortest path tree set) boolean array, initialized to false, to track included vertices.
3. Loop for V-1 times (V is the number of vertices).
4. In each iteration, pick the vertex u not yet in sptSet with the minimum distance value.
5. Include u in sptSet. Update the dist values of all adjacent vertices of u.

**Code:**

#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];  
 int 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;  
 for (int v = 0; v < V; v++)  
 if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v])  
 dist[v] = dist[u] + graph[u][v];  
 }  
 printf("Vertex Distance from Source\n");  
 for (int i = 0; i < V; i++) printf("%d \t\t %d\n", i, dist[i]);  
}  
  
int main() {  
 int graph[V][V] = {{0, 4, 0, 0, 1}, {4, 0, 2, 0, 0}, {0, 2, 0, 3, 0}, {0, 0, 3, 0, 1}, {1, 0, 0, 1, 0}};  
 dijkstra(graph, 0);  
 return 0;  
}

**Output:**

Vertex Distance from Source  
0 0  
1 4  
2 6  
3 3  
4 1

**Result:** The program successfully computes the shortest paths from a source vertex using Dijkstra's algorithm.

### 14. Traveling Salesman Problem (TSP)

**Aim:** To solve the Traveling Salesman Problem using a brute-force recursive approach.

**Algorithm:**

1. Represent the graph with an adjacency matrix where graph[i][j] is the distance between city i and city j.
2. Start at a source city (e.g., city 0) and initialize minimum weight to a very large value.
3. Use a recursive function that explores all possible paths starting from the current city.
4. Keep track of visited cities using a boolean array.
5. When all cities have been visited, return to the starting city and update the minimum weight if the current path's weight is smaller.

**Code:**

#include <stdio.h>  
#include <limits.h>  
#define V 4  
  
int tsp(int graph[][V], int visited[], int currPos, int n, int count, int cost, int ans) {  
 if (count == n && graph[currPos][0]) {  
 ans = (ans < cost + graph[currPos][0]) ? ans : cost + graph[currPos][0];  
 return ans;  
 }  
 for (int i = 0; i < n; i++) {  
 if (!visited[i] && graph[currPos][i]) {  
 visited[i] = 1;  
 ans = tsp(graph, visited, i, n, count + 1, cost + graph[currPos][i], ans);  
 visited[i] = 0;  
 }  
 }  
 return ans;  
}  
  
int main() {  
 int graph[][V] = {{0, 10, 15, 20}, {10, 0, 35, 25}, {15, 35, 0, 30}, {20, 25, 30, 0}};  
 int visited[V] = {0};  
 visited[0] = 1;  
 printf("Minimum cost: %d\n", tsp(graph, visited, 0, V, 1, 0, INT\_MAX));  
 return 0;  
}

**Output:**

Minimum cost: 80

**Result:** The program successfully finds the minimum cost for the Traveling Salesman Problem tour.

### 15. Binary Search Tree - Search, Min, and Max

**Aim:** To implement search, find minimum, and find maximum element operations in a Binary Search Tree (BST).

**Algorithm:**

1. **Search:** Start at the root. If the target key is the same, return the node. If the key is smaller, go to the left child; if larger, go to the right child. Repeat until found or a NULL pointer is reached.
2. **Find Minimum:** Start at the root and repeatedly move to the left child until a node with no left child is found. This node is the minimum.
3. **Find Maximum:** Start at the root and repeatedly move to the right child until a node with no right child is found. This node is the maximum.

**Code:**

#include <stdio.h>  
#include <stdlib.h>  
  
struct node { int key; struct node \*left, \*right; };  
  
struct node\* newNode(int item) {  
 struct node\* temp = (struct node\*)malloc(sizeof(struct node));  
 temp->key = item; temp->left = temp->right = NULL;  
 return temp;  
}  
  
struct node\* insert(struct node\* node, int key) {  
 if (node == NULL) return newNode(key);  
 if (key < node->key) node->left = insert(node->left, key);  
 else if (key > node->key) node->right = insert(node->right, key);  
 return node;  
}  
  
struct node\* search(struct node\* root, int key) {  
 if (root == NULL || root->key == key) return root;  
 if (root->key < key) return search(root->right, key);  
 return search(root->left, key);  
}  
  
struct node\* minValueNode(struct node\* node) {  
 struct node\* current = node;  
 while (current && current->left != NULL) current = current->left;  
 return current;  
}  
  
struct node\* maxValueNode(struct node\* node) {  
 struct node\* current = node;  
 while (current && current->right != NULL) current = current->right;  
 return current;  
}  
  
int main() {  
 struct node\* root = NULL;  
 root = insert(root, 50); insert(root, 30); insert(root, 20); insert(root, 40); insert(root, 70);  
 printf("Min value is %d\n", minValueNode(root)->key);  
 printf("Max value is %d\n", maxValueNode(root)->key);  
 printf("Search for 40: %s\n", search(root, 40) ? "Found" : "Not Found");  
 return 0;  
}

**Output:**

Min value is 20  
Max value is 70  
Search for 40: Found

**Result:** The program successfully implements and demonstrates search, find minimum, and find maximum in a BST.

### 16. Array Sort - Ascending and Descending

**Aim:** To sort an array in both ascending and descending order using the Bubble Sort algorithm.

**Algorithm:**

1. **Ascending Sort:**
   * Use nested loops. The outer loop runs from i=0 to n-2.
   * The inner loop runs from j=0 to n-i-2.
   * Compare arr[j] with arr[j+1]. If arr[j] > arr[j+1], swap them.
2. **Descending Sort:**
   * Use the same nested loop structure.
   * Compare arr[j] with arr[j+1]. If arr[j] < arr[j+1], swap them.

**Code:**

#include <stdio.h>  
  
void swap(int\* xp, int\* yp) { int temp = \*xp; \*xp = \*yp; \*yp = temp; }  
  
void bubbleSortAsc(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]) swap(&arr[j], &arr[j + 1]);  
}  
  
void bubbleSortDesc(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]) swap(&arr[j], &arr[j + 1]);  
}  
  
void printArray(int arr[], int size) {  
 for (int i = 0; i < size; i++) printf("%d ", arr[i]);  
 printf("\n");  
}  
  
int main() {  
 int arr[] = {64, 34, 25, 12, 22, 11, 90};  
 int n = sizeof(arr) / sizeof(arr[0]);  
 bubbleSortAsc(arr, n);  
 printf("Ascending sorted array: "); printArray(arr, n);  
 bubbleSortDesc(arr, n);  
 printf("Descending sorted array: "); printArray(arr, n);  
 return 0;  
}

**Output:**

Ascending sorted array: 11 12 22 25 34 64 90   
Descending sorted array: 90 64 34 25 22 12 11

**Result:** The program successfully sorts the given array in both ascending and descending order.

### 17. Array Search - Linear and Binary

**Aim:** To implement both linear and binary search algorithms to find an element in an array.

**Algorithm:**

1. **Linear Search:**
   * Iterate through the array from the first element to the last.
   * Compare each element with the target value.
   * If a match is found, return the index. If the loop ends, return -1.
2. **Binary** Search (on **a sorted array):**
   * Set low to 0 and high to n-1.
   * While low <= high, calculate mid.
   * If arr[mid] is the target, return mid.
   * If arr[mid] is less than the target, set low = mid + 1.
   * Otherwise, set high = mid - 1. If the loop ends, return -1.

**Code:**

#include <stdio.h>  
  
int linearSearch(int arr[], int n, int x) {  
 for (int i = 0; i < n; i++) if (arr[i] == x) return i;  
 return -1;  
}  
  
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;  
}  
  
int main() {  
 int arr[] = {2, 3, 4, 10, 40};  
 int n = sizeof(arr) / sizeof(arr[0]);  
 int x = 10;  
 int linear\_result = linearSearch(arr, n, x);  
 printf("Linear Search: Element found at index %d\n", linear\_result);  
 int binary\_result = binarySearch(arr, 0, n - 1, x);  
 printf("Binary Search: Element found at index %d\n", binary\_result);  
 return 0;  
}

**Output:**

Linear Search: Element found at index 3  
Binary Search: Element found at index 3

**Result:** The program successfully implements and demonstrates both linear and binary search.

### 18. Display 5th Iterated Element

**Aim:** To access and display the 5th element (at index 4) of a given array.

**Algorithm:**

1. Define an array of integers.
2. Get the size of the array.
3. Check if the size of the array is at least 5.
4. If it is, access the element at index 4 (arr[4]) and print it.
5. If the size is less than 5, print an error message indicating the array is too small.

**Code:**

#include <stdio.h>  
  
int main() {  
 int arr[] = {10, 20, 30, 40, 50, 60, 70};  
 int n = sizeof(arr) / sizeof(arr[0]);  
   
 if (n >= 5) {  
 printf("The 5th element of the array is: %d\n", arr[4]);  
 } else {  
 printf("The array has less than 5 elements.\n");  
 }  
   
 return 0;  
}

**Output:**

The 5th element of the array is: 50

**Result:** The program successfully accesses and displays the 5th element from the array.

### 19. Find Missing Element in Unsorted Array

**Aim:** To find the single missing element in an unsorted array of consecutive numbers.

**Algorithm:**

1. Given an array of size n-1 containing numbers from 1 to n with one missing element.
2. Calculate the expected sum of the first n natural numbers using the formula: expected\_sum = n \* (n + 1) / 2.
3. Calculate the actual sum of all elements present in the given array.
4. The missing number is the difference between the expected\_sum and the actual\_sum.
5. Here, n will be the size of the array plus one.

**Code:**

#include <stdio.h>  
  
int findMissingNo(int arr[], int n) {  
 int total = (n + 1) \* (n + 2) / 2;  
 for (int i = 0; i < n; i++)  
 total -= arr[i];  
 return total;  
}  
  
int main() {  
 int arr[] = {1, 2, 4, 6, 3, 7, 8};  
 int n = sizeof(arr) / sizeof(arr[0]);  
 int miss = findMissingNo(arr, n);  
 printf("The missing number is: %d\n", miss);  
 return 0;  
}

**Output:**

The missing number is: 5

**Result:** The program successfully finds the missing element from the array using the summation method.

### 20. Array Concatenation

**Aim:** To write a C program to concatenate two arrays into a third, new array.

**Algorithm:**

1. Define two input arrays and their sizes.
2. Create a third array with a size equal to the sum of the sizes of the first two arrays.
3. Use a loop to copy all elements from the first array into the new array.
4. Use a second loop to copy all elements from the second array into the new array, starting from the position after the last element of the first array.
5. Print the final concatenated array.

**Code:**

#include <stdio.h>  
  
int main() {  
 int arr1[] = {1, 2, 3};  
 int arr2[] = {4, 5, 6};  
 int n1 = sizeof(arr1) / sizeof(arr1[0]);  
 int n2 = sizeof(arr2) / sizeof(arr2[0]);  
 int result[n1 + n2];  
 int i;  
  
 for (i = 0; i < n1; i++) result[i] = arr1[i];  
 for (i = 0; i < n2; i++) result[n1 + i] = arr2[i];  
  
 printf("Concatenated array: ");  
 for (i = 0; i < n1 + n2; i++) printf("%d ", result[i]);  
 printf("\n");  
  
 return 0;  
}

**Output:**

Concatenated array: 1 2 3 4 5 6

**Result:** The program successfully concatenates two arrays into a single new array.

### 21. Haystack (Find Substring)

**Aim:** To implement the strstr function to find the first occurrence of a "needle" string within a "haystack" string.

**Algorithm:**

1. Get the haystack and needle strings.
2. Use an outer loop to iterate through the haystack string.
3. For each position in the haystack, use an inner loop to check if the substring starting at that position matches the needle.
4. A temporary pointer can be used to traverse the haystack while a second pointer traverses the needle.
5. If the inner loop completes successfully (all characters of the needle match), return the starting index from the outer loop. If the outer loop finishes, return -1.

**Code:**

#include <stdio.h>  
#include <string.h>  
  
int findSubstring(char\* haystack, char\* needle) {  
 int hLen = strlen(haystack);  
 int nLen = strlen(needle);  
 if (nLen == 0) return 0;  
  
 for (int i = 0; i <= hLen - nLen; i++) {  
 int j;  
 for (j = 0; j < nLen; j++) {  
 if (haystack[i + j] != needle[j]) {  
 break;  
 }  
 }  
 if (j == nLen) {  
 return i;  
 }  
 }  
 return -1;  
}  
  
int main() {  
 char haystack[] = "hello\_world";  
 char needle[] = "world";  
 int index = findSubstring(haystack, needle);  
 if (index != -1) {  
 printf("Needle found at index %d.\n", index);  
 } else {  
 printf("Needle not found.\n");  
 }  
 return 0;  
}

**Output:**

Needle found at index 6.

**Result:** The program successfully finds the starting index of a substring within a larger string.

### 22. Minimum Edge in a Graph

**Aim:** To represent a graph using an adjacency matrix and find the edge with the minimum weight.

**Algorithm:**

1. Represent the graph using a 2D array (adjacency matrix), where matrix[i][j] is the edge weight.
2. Initialize a variable min\_edge to a very large value (e.g., INT\_MAX).
3. Use nested loops to iterate through the upper triangle of the matrix to avoid checking each edge twice.
4. For each element matrix[i][j] that represents a valid edge (weight > 0), compare it with min\_edge.
5. If matrix[i][j] is smaller than min\_edge, update min\_edge.

**Code:**

#include <stdio.h>  
#include <limits.h>  
#define V 4  
  
int findMinEdge(int graph[V][V]) {  
 int min\_edge = INT\_MAX;  
 for (int i = 0; i < V; i++) {  
 for (int j = i + 1; j < V; j++) {  
 if (graph[i][j] != 0 && graph[i][j] < min\_edge) {  
 min\_edge = graph[i][j];  
 }  
 }  
 }  
 return min\_edge;  
}  
  
int main() {  
 int graph[V][V] = {  
 {0, 10, 5, 0},  
 {10, 0, 0, 2},  
 {5, 0, 0, 8},  
 {0, 2, 8, 0}  
 };  
 printf("The minimum edge weight is: %d\n", findMinEdge(graph));  
 return 0;  
}

**Output:**

The minimum edge weight is: 2

**Result:** The program successfully finds the minimum weight edge from the graph's adjacency matrix representation.

### 23. Print Valid Path in a Graph

**Aim:** To determine if a valid path exists between a source and a destination vertex in a graph using Depth First Search (DFS).

**Algorithm:**

1. Represent the graph using an adjacency list or matrix.
2. Use a visited array, initialized to false, to track visited nodes during traversal.
3. Start a recursive DFS function from the source vertex. Mark the source as visited.
4. In the DFS function, if the current vertex is the destination, return true.
5. Otherwise, for all adjacent and unvisited neighbors of the current vertex, recursively call the DFS function. If any recursive call returns true, return true.

**Code:**

#include <stdio.h>  
#include <stdbool.h>  
#define V 4  
  
bool isReachable(int graph[V][V], int src, int dest, bool visited[]) {  
 if (src == dest) return true;  
 visited[src] = true;  
 for (int i = 0; i < V; i++) {  
 if (graph[src][i] && !visited[i]) {  
 if (isReachable(graph, i, dest, visited)) {  
 return true;  
 }  
 }  
 }  
 return false;  
}  
  
int main() {  
 int graph[V][V] = {{0, 1, 1, 0}, {0, 0, 1, 0}, {1, 0, 0, 1}, {0, 0, 0, 1}};  
 int u = 0, v = 3;  
 bool visited[V] = {false};  
 if (isReachable(graph, u, v, visited))  
 printf("Path exists from %d to %d\n", u, v);  
 else  
 printf("No path exists from %d to %d\n", u, v);  
 return 0;  
}

**Output:**

Path exists from 0 to 3

**Result:** The program successfully determines if a path exists between two vertices in a graph.

### 24. Heap, Merge, Insertion, and Quick Sort

**Aim:** To provide implementations for Heap Sort, Merge Sort, Insertion Sort, and Quick Sort.

**Algorithm:**

1. **Heap Sort:** Build a max heap. Repeatedly extract the maximum element from the heap (the root) and place it at the end of the array.
2. **Merge Sort:** A divide-and-conquer algorithm. It recursively divides the array in half, sorts the halves, and then merges them.
3. **Insertion Sort:** Iterates through the input elements, and for each element, it finds its correct position in the sorted part of the array and inserts it there.
4. **Quick Sort:** A divide-and-conquer algorithm. It picks a 'pivot' element and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

**Code:**

#include <stdio.h>  
  
void swap(int\* a, int\* b) { int t = \*a; \*a = \*b; \*b = t; }  
  
// --- Quick Sort ---  
int partition(int arr[], int low, int high) {  
 int pivot = arr[high]; int i = (low - 1);  
 for (int j = low; j <= high - 1; j++) { if (arr[j] < pivot) { i++; swap(&arr[i], &arr[j]); } }  
 swap(&arr[i + 1], &arr[high]); return (i + 1);  
}  
void quickSort(int arr[], int low, int high) {  
 if (low < high) { int pi = partition(arr, low, high); quickSort(arr, low, pi - 1); quickSort(arr, pi + 1, high); }  
}  
  
// --- Insertion Sort ---  
void insertionSort(int arr[], int n) {  
 int i, key, j;  
 for (i = 1; i < n; i++) { key = arr[i]; j = i - 1; while (j >= 0 && arr[j] > key) { arr[j + 1] = arr[j]; j = j - 1; } arr[j + 1] = key; }  
}  
  
// --- Merge Sort ---  
void merge(int arr[], int l, int m, int r) {  
 int i, j, k; int n1 = m - l + 1; int n2 = r - m;  
 int L[n1], R[n2];  
 for (i = 0; i < n1; i++) L[i] = arr[l + i];  
 for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];  
 i = 0; j = 0; k = l;  
 while (i < n1 && j < n2) { arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++]; }  
 while (i < n1) { arr[k++] = L[i++]; } while (j < n2) { arr[k++] = R[j++]; }  
}  
void mergeSort(int arr[], int l, int r) { if (l < r) { int m = l + (r - l) / 2; mergeSort(arr, l, m); mergeSort(arr, m + 1, r); merge(arr, l, m, r); } }  
  
// --- Heap Sort ---  
void heapify(int arr[], int n, int i) {  
 int largest = i; int l = 2 \* i + 1; int r = 2 \* i + 2;  
 if (l < n && arr[l] > arr[largest]) largest = l;  
 if (r < n && arr[r] > arr[largest]) largest = r;  
 if (largest != i) { swap(&arr[i], &arr[largest]); heapify(arr, n, largest); }  
}  
void heapSort(int arr[], int n) {  
 for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);  
 for (int i = n - 1; i > 0; i--) { swap(&arr[0], &arr[i]); heapify(arr, i, 0); }  
}  
  
void printArray(int A[], int size) { for (int i = 0; i < size; i++) printf("%d ", A[i]); printf("\n"); }  
  
int main() {  
 int arr[] = {12, 11, 13, 5, 6, 7};  
 int n = sizeof(arr) / sizeof(arr[0]);  
 quickSort(arr, 0, n - 1); // Example usage of one sort  
 printf("Sorted array: \n"); printArray(arr, n);  
 return 0;  
}

**Output:**

Sorted array:   
5 6 7 11 12 13

**Result:** The program provides the C code for four major sorting algorithms. The output shows the result for Quick Sort.

### 25. Count Nodes in a Linked List

**Aim:** To write a C program that counts the total number of nodes in a given singly linked list.

**Algorithm:**

1. Initialize a count variable to 0.
2. Create a temporary pointer, current, and point it to the head of the list.
3. Traverse the list by moving current to current->next until current becomes NULL.
4. For each node visited, increment the count.
5. After the loop terminates, return the final count.

**Code:**

#include <stdio.h>  
#include <stdlib.h>  
  
struct Node { int data; struct Node\* next; };  
  
void push(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;  
}  
  
int getCount(struct Node\* head) {  
 int count = 0;  
 struct Node\* current = head;  
 while (current != NULL) {  
 count++;  
 current = current->next;  
 }  
 return count;  
}  
  
int main() {  
 struct Node\* head = NULL;  
 push(&head, 1); push(&head, 3); push(&head, 1); push(&head, 2); push(&head, 1);  
 printf("Count of nodes is %d\n", getCount(head));  
 return 0;  
}

**Output:**

Count of nodes is 5

**Result:** The program successfully counts and prints the number of nodes in the linked list.

### 26. Largest Element in a 2D Matrix

**Aim:** To find and print the largest element in a given 2D matrix (array).

**Algorithm:**

1. Get the input 2D matrix.
2. Initialize a variable maxElement with the value of the very first element (matrix[0][0]).
3. Use nested loops to iterate through every element of the matrix.
4. The outer loop iterates through rows, and the inner loop iterates through columns.
5. In the inner loop, compare the current element with maxElement. If the current element is larger, update maxElement.

**Code:**

#include <stdio.h>  
#define R 3  
#define C 3  
  
int findLargest(int matrix[R][C]) {  
 int maxElement = matrix[0][0];  
 for (int i = 0; i < R; i++) {  
 for (int j = 0; j < C; j++) {  
 if (matrix[i][j] > maxElement) {  
 maxElement = matrix[i][j];  
 }  
 }  
 }  
 return maxElement;  
}  
  
int main() {  
 int matrix[R][C] = {{1, 2, 3}, {4, 9, 6}, {7, 8, 5}};  
 printf("The largest element in the matrix is: %d\n", findLargest(matrix));  
 return 0;  
}

**Output:**

The largest element in the matrix is: 9

**Result:** The program successfully finds and prints the largest element from the 2D matrix.

### 27. Sort a String in Alphabetical Order

**Aim:** To sort the characters of a given string in alphabetical order.

**Algorithm:**

1. Get the input string.
2. Use a simple sorting algorithm like Bubble Sort.
3. Use nested loops to iterate through the string.
4. The outer loop selects a character, and the inner loop compares it with the subsequent characters.
5. If string[i] > string[j], swap the characters.

**Code:**

#include <stdio.h>  
#include <string.h>  
  
void sortString(char\* str) {  
 int n = strlen(str);  
 char temp;  
 for (int i = 0; i < n-1; i++) {  
 for (int j = i+1; j < n; j++) {  
 if (str[i] > str[j]) {  
 temp = str[i];  
 str[i] = str[j];  
 str[j] = temp;  
 }  
 }  
 }  
}  
  
int main() {  
 char str[] = "programming";  
 printf("Original string: %s\n", str);  
 sortString(str);  
 printf("Sorted string: %s\n", str);  
 return 0;  
}

**Output:**

Original string: programming  
Sorted string: aggimmnoprr

**Result:** The program successfully sorts the characters of the given string alphabetically.

### 28. Print Index of Repeated Characters in an Array

**Aim:** To find characters that are repeated in a string (char array) and print all their indices.

**Algorithm:**

1. Get the input string.
2. Use a frequency array (e.g., count[256]) initialized to zero to store character counts.
3. Iterate through the string once to populate the frequency array.
4. Iterate through the frequency array from 0 to 255.
5. If the count for any character is greater than 1, iterate through the original string again, and print the index j whenever string[j] matches the repeated character.

**Code:**

#include <stdio.hh>  
#include <string.h>  
#define NO\_OF\_CHARS 256  
  
void printIndicesOfRepeated(char \*str) {  
 int count[NO\_OF\_CHARS] = {0};  
 int len = strlen(str);  
  
 for (int i = 0; i < len; i++) count[(int)str[i]]++;  
  
 for (int i = 0; i < NO\_OF\_CHARS; i++) {  
 if (count[i] > 1) {  
 printf("Character '%c' is repeated at indices: ", i);  
 for (int j = 0; j < len; j++) {  
 if (str[j] == i) printf("%d ", j);  
 }  
 printf("\n");  
 }  
 }  
}  
  
int main() {  
 char str[] = "hellohello";  
 printIndicesOfRepeated(str);  
 return 0;  
}

**Output:**

Character 'e' is repeated at indices: 1 6   
Character 'h' is repeated at indices: 0 5   
Character 'l' is repeated at indices: 2 3 7 8   
Character 'o' is repeated at indices: 4 9

**Result:** The program successfully identifies repeated characters and prints all their corresponding indices.

### 29. Count of Frequently Repeated Numbers

**Aim:** To find the number that is most frequently repeated in an array and its count.

**Algorithm:**

1. Sort the input array. This brings all identical elements together.
2. Iterate through the sorted array.
3. Keep track of the current element's frequency (current\_count) and the maximum frequency found so far (max\_count).
4. As you iterate, if the current element is the same as the previous one, increment current\_count.
5. If the element changes, compare current\_count with max\_count. If current\_count is greater, update max\_count and store the element. Reset current\_count to 1.

**Code:**

#include <stdio.h>  
  
// Simple sort for the algorithm  
void sort(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 findMostFrequent(int arr[], int n) {  
 sort(arr, n);  
 int max\_count = 0, res = arr[0], curr\_count = 1;  
 for (int i = 1; i < n; i++) {  
 if (arr[i] == arr[i - 1]) curr\_count++;  
 else {  
 if (curr\_count > max\_count) { max\_count = curr\_count; res = arr[i - 1]; }  
 curr\_count = 1;  
 }  
 }  
 if (curr\_count > max\_count) { max\_count = curr\_count; res = arr[n - 1]; }  
 printf("Most frequent number is %d, repeated %d times.\n", res, max\_count);  
}  
  
int main() {  
 int arr[] = {1, 3, 2, 1, 4, 1, 3, 3};  
 int n = sizeof(arr) / sizeof(arr[0]);  
 findMostFrequent(arr, n);  
 return 0;  
}

**Output:**

Most frequent number is 1, repeated 3 times.

**Result:** The program successfully finds the most frequent element and its count in the array.

### 30. Palindrome using SLL

**Aim:** To check if a Singly Linked List (SLL) is a palindrome.

**Algorithm:**

1. Find the middle of the linked list using a slow and fast pointer approach.
2. Reverse the second half of the list starting from the node after the middle.
3. Compare the first half of the list with the reversed second half.
4. If all corresponding nodes have the same data, the list is a palindrome.
5. (Optional but good practice) Restore the original list by reversing the second half again.

**Code:**

#include <stdio.h>  
#include <stdlib.h>  
#include <stdbool.h>  
  
struct Node { int data; struct Node\* next; };  
  
void reverse(struct Node\*\* head\_ref) {  
 struct Node \*prev = NULL, \*current = \*head\_ref, \*next = NULL;  
 while (current != NULL) { next = current->next; current->next = prev; prev = current; current = next; }  
 \*head\_ref = prev;  
}  
  
bool isPalindrome(struct Node\* head) {  
 struct Node \*slow\_ptr = head, \*fast\_ptr = head, \*second\_half, \*prev\_of\_slow\_ptr = head;  
 struct Node\* midnode = NULL;  
 bool res = true;  
 if (head != NULL && head->next != NULL) {  
 while (fast\_ptr != NULL && fast\_ptr->next != NULL) {  
 fast\_ptr = fast\_ptr->next->next; prev\_of\_slow\_ptr = slow\_ptr; slow\_ptr = slow\_ptr->next;  
 }  
 if (fast\_ptr != NULL) { midnode = slow\_ptr; slow\_ptr = slow\_ptr->next; }  
 second\_half = slow\_ptr; prev\_of\_slow\_ptr->next = NULL;  
 reverse(&second\_half);  
 struct Node\* p1 = head; struct Node\* p2 = second\_half;  
 while (p1 && p2) { if (p1->data != p2->data) res = false; p1 = p1->next; p2 = p2->next; }  
 }  
 return res;  
}  
  
int main() {  
 struct Node\* head = &(struct Node){1, &(struct Node){2, &(struct Node){2, &(struct Node){1, NULL}}}};  
 if (isPalindrome(head)) printf("Is a palindrome.\n");  
 else printf("Not a palindrome.\n");  
 return 0;  
}

**Output:**

Is a palindrome.

**Result:** The program successfully checks if the given singly linked list is a palindrome.

### 31. Binary Tree Implementation

**Aim:** To create a basic binary tree structure and demonstrate node creation.

**Algorithm:**

1. Define a struct Node containing an integer data, and two pointers, left and right, to other nodes.
2. Create a helper function newNode(data) that allocates memory for a new node, assigns the data, and initializes its left and right children to NULL.
3. In the main function, create the root node.
4. Manually create child nodes and link them to the root and each other to form a simple tree structure.
5. This demonstrates the fundamental structure without complex insertion logic like in a BST.

**Code:**

#include <stdio.h>  
#include <stdlib.h>  
  
struct Node {  
 int data;  
 struct Node\* left;  
 struct Node\* right;  
};  
  
struct Node\* newNode(int data) {  
 struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));  
 node->data = data;  
 node->left = NULL;  
 node->right = NULL;  
 return node;  
}  
  
int main() {  
 struct Node\* root = newNode(1);  
 root->left = newNode(2);  
 root->right = newNode(3);  
 root->left->left = newNode(4);  
   
 printf("Binary tree created successfully.\n");  
 printf("Root: %d, Left Child: %d, Right Child: %d\n", root->data, root->left->data, root->right->data);  
   
 return 0;  
}

**Output:**

Binary tree created successfully.  
Root: 1, Left Child: 2, Right Child: 3

**Result:** The program successfully demonstrates the creation of a simple binary tree.

### 32. BST - Kth Smallest Value

**Aim:** To find the Kth smallest value in a Binary Search Tree (BST).

**Algorithm:**

1. The in-order traversal of a BST visits nodes in ascending order.
2. We can use this property to find the Kth smallest element.
3. Write a recursive function that performs an in-order traversal.
4. Pass a counter k by reference to the function.
5. In the function, first recur for the left subtree. Then, decrement k. If k becomes 0, the current node is the Kth smallest. Finally, recur for the right subtree if needed.

**Code:**

#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;  
}  
  
struct Node\* insert(struct Node\* node, int data) {  
 if (node == NULL) return newNode(data);  
 if (data < node->data) node->left = insert(node->left, data);  
 else if (data > node->data) node->right = insert(node->right, data);  
 return node;  
}  
  
void kthSmallestUtil(struct Node\* root, int k, int\* count, int\* result) {  
 if (root == NULL || \*count >= k) return;  
 kthSmallestUtil(root->left, k, count, result);  
 (\*count)++;  
 if (\*count == k) { \*result = root->data; return; }  
 kthSmallestUtil(root->right, k, count, result);  
}  
  
int kthSmallest(struct Node\* root, int k) {  
 int count = 0, result = -1;  
 kthSmallestUtil(root, k, &count, &result);  
 return result;  
}  
  
int main() {  
 struct Node\* root = NULL;  
 root = insert(root, 20); insert(root, 8); insert(root, 22); insert(root, 4); insert(root, 12);  
 int k = 3;  
 printf("The %dth smallest value is %d\n", k, kthSmallest(root, k));  
 return 0;  
}

**Output:**

The 3th smallest value is 12

**Result:** The program successfully finds the Kth smallest element in the BST using in-order traversal logic.

### 33. Intersection of Two SLLs

**Aim:** To find the node at which two singly linked lists intersect.

**Algorithm:**

1. Get the lengths of both lists, len1 and len2.
2. Calculate the difference in lengths, d = abs(len1 - len2).
3. Move the head pointer of the longer list forward by d nodes.
4. Now, both lists have an equal number of nodes from the current pointers to the end.
5. Traverse both lists simultaneously, one node at a time. The first node where the pointers become equal is the intersection point.

**Code:**

#include <stdio.h>  
#include <stdlib.h>  
  
struct Node { int data; struct Node\* next; };  
  
int getCount(struct Node\* head) {  
 struct Node\* current = head; int count = 0;  
 while (current != NULL) { count++; current = current->next; }  
 return count;  
}  
  
int getIntersectionNode(struct Node\* head1, struct Node\* head2) {  
 int c1 = getCount(head1); int c2 = getCount(head2); int d;  
 struct Node \*current1 = head1, \*current2 = head2;  
 d = (c1 > c2) ? c1 - c2 : c2 - c1;  
 if (c1 > c2) for (int i = 0; i < d; i++) current1 = current1->next;  
 else for (int i = 0; i < d; i++) current2 = current2->next;  
 while (current1 != NULL && current2 != NULL) {  
 if (current1 == current2) return current1->data;  
 current1 = current1->next; current2 = current2->next;  
 }  
 return -1;  
}  
  
int main() {  
 struct Node\* newNode;  
 struct Node\* head1 = (struct Node\*)malloc(sizeof(struct Node)); head1->data = 10;  
 struct Node\* head2 = (struct Node\*)malloc(sizeof(struct Node)); head2->data = 3;  
 newNode = (struct Node\*)malloc(sizeof(struct Node)); newNode->data = 6; head2->next = newNode;  
 newNode = (struct Node\*)malloc(sizeof(struct Node)); newNode->data = 9; head2->next->next = newNode;  
 newNode = (struct Node\*)malloc(sizeof(struct Node)); newNode->data = 15; head1->next = newNode; head2->next->next->next = newNode;  
 newNode = (struct Node\* head2 = (struct Node\*)malloc(sizeof(struct Node)); head2->data = 3;

newNode = (struct Node\*)malloc(sizeof(struct Node)); newNode->data = 6; head2->next = newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node)); newNode->data = 9; head2->next->next = newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node)); newNode->data = 15; head1->next = newNode; head2->next->next->next = newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node)); newNode->data = 30; head1->next->next = newNode; head1->next->next->next = NULL;

printf("The intersection point is %d\n", getIntersectionNode(head1, head2));

return 0;

}

**Output:**

The intersection point is 15

**Result:** The program successfully identifies the data of the intersecting node in two linked lists.

**34. Stack using Two Queues**

**Aim:**To implement a stack data structure using two queues.

**Algorithm:**

1. Use two queues, `q1` (main) and `q2` (helper).

2. \*\*Push:\*\* To push an element, first enqueue the new element to `q2`.

3. Then, dequeue all elements from `q1` and enqueue them one by one into `q2`.

4. Finally, swap the names of `q1` and `q2`. The new element is now at the front of `q1`.

5. \*\*Pop:\*\* To pop, simply dequeue from `q1`.

**Code:**

```c

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct { int arr[MAX]; int front, rear; } Queue;

void enqueue(Queue\* q, int val) { q->arr[q->rear++] = val; }

int dequeue(Queue\* q) { return q->arr[q->front++]; }

typedef struct { Queue q1, q2; } Stack;

void push(Stack\* s, int x) {

enqueue(&s->q2, x);

while (s->q1.front != s->q1.rear) enqueue(&s->q2, dequeue(&s->q1));

Queue temp = s->q1; s->q1 = s->q2; s->q2 = temp;

}

int pop(Stack\* s) { return dequeue(&s->q1); }

int main() {

Stack s = {{{0}, 0, 0}, {{0}, 0, 0}};

push(&s, 1); push(&s, 2); push(&s, 3);

printf("%d ", pop(&s));

printf("%d ", pop(&s));

printf("\n");

return 0;

}

**Output:**

3 2

**Result:** The program successfully implements stack operations using two queues, demonstrating the LIFO (Last-In, First-Out) principle.

### 35. Queue using Two Stacks

**Aim:** To implement a queue data structure using two stacks.

**Algorithm:**

1. Use two stacks, s1 (for enqueuing) and s2 (for dequeuing).
2. **Enqueue:** To enqueue an item, simply push it onto s1.
3. **Dequeue:** To dequeue an item, first check if s2 is empty.
4. If s2 is empty, pop every element from s1 and push it onto s2. This reverses the order.
5. Finally, pop the top element from s2. This will be the oldest element, following the FIFO principle.

**Code:**

#include <stdio.h>

#define MAX 100

typedef struct { int arr[MAX]; int top; } Stack;

void push(Stack\* s, int x) { s->arr[++s->top] = x; }

int pop(Stack\* s) { return s->arr[s->top--]; }

typedef struct { Stack s1, s2; } Queue;

void enqueue(Queue\* q, int x) { push(&q->s1, x); }

int dequeue(Queue\* q) {

if (q->s2.top == -1) {

while (q->s1.top != -1) push(&q->s2, pop(&q->s1));

}

return pop(&q->s2);

}

int main() {

Queue q = {{{0}, -1}, {{0}, -1}};

enqueue(&q, 1); enqueue(&q, 2); enqueue(&q, 3);

printf("%d ", dequeue(&q));

printf("%d ", dequeue(&q));

printf("\n");

return 0;

}

**Output:**

1 2

**Result:** The program successfully implements queue operations using two stacks, demonstrating the FIFO (First-In, First-Out) principle.

### 36. Tree Traversal

**Aim:** To implement and demonstrate In-order, Pre-order, and Post-order traversals of a binary tree.

**Algorithm:**

1. Create a binary tree structure.
2. **In-order:** Recursively traverse the left subtree, visit the root node, then recursively traverse the right subtree.
3. **Pre-order:** Visit the root node, recursively traverse the left subtree, then recursively traverse the right subtree.
4. **Post-order:** Recursively traverse the left subtree, recursively traverse the right subtree, then visit the root node.
5. Implement separate functions for each traversal type.

**Code:**

#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 printPostorder(struct Node\* node) {

if (node == NULL) return;

printPostorder(node->left); printPostorder(node->right); printf("%d ", node->data);

}

void printInorder(struct Node\* node) {

if (node == NULL) return;

printInorder(node->left); printf("%d ", node->data); printInorder(node->right);

}

void printPreorder(struct Node\* node) {

if (node == NULL) return;

printf("%d ", node->data); printPreorder(node->left); printPreorder(node->right);

}

int main() {

struct Node\* root = newNode(1);

root->left = newNode(2); root->right = newNode(3);

root->left->left = newNode(4); root->left->right = newNode(5);

printf("Preorder traversal: "); printPreorder(root); printf("\n");

printf("Inorder traversal: "); printInorder(root); printf("\n");

printf("Postorder traversal: "); printPostorder(root); printf("\n");

return 0;

}

**Output:**

Preorder traversal: 1 2 4 5 3

Inorder traversal: 4 2 5 1 3

Postorder traversal: 4 5 2 3 1

**Result:** The program successfully demonstrates the three primary DFS-based traversals of a binary tree.

### 37. Linked List - Insertion

**Aim:** To implement functions for inserting a new node at the beginning, at the end, and after a given node in a singly linked list.

**Algorithm:**

1. **Insert at Beginning:** Create a new node. Set its next pointer to the current head of the list. Update the head to point to this new node.
2. **Insert at End:** Create a new node. Traverse the list to find the last node. Set the next pointer of the last node to the new node.
3. **Insert After Node:** Create a new node. Set the new node's next to the next node of the given previous node. Set the previous node's next to the new node.
4. Handle the edge case where the list is empty.
5. A helper function to print the list is useful for verification.

**Code:**

#include <stdio.h>

#include <stdlib.h>

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

void push(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;

}

void append(struct Node\*\* head\_ref, int new\_data) {

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

struct Node\* last = \*head\_ref;

new\_node->data = new\_data; new\_node->next = NULL;

if (\*head\_ref == NULL) { \*head\_ref = new\_node; return; }

while (last->next != NULL) last = last->next;

last->next = new\_node;

}

void printList(struct Node\* node) {

while (node != NULL) { printf(" %d ", node->data); node = node->next; }

printf("\n");

}

int main() {

struct Node\* head = NULL;

append(&head, 6);

push(&head, 7);

push(&head, 1);

append(&head, 4);

printf("Created Linked list is:"); printList(head);

return 0;

}

**Output:**

Created Linked list is: 1 7 6 4

**Result:** The program successfully demonstrates the insertion of nodes at the beginning and end of a linked list.

### 38. Bidirectional Linked List (Doubly)

**Aim:** To implement a doubly linked list and demonstrate forward and backward traversal.

**Algorithm:**

1. Define a struct Node with data, a next pointer, and a prev pointer.
2. Implement an insertion function (e.g., at the beginning).
3. When inserting, carefully update the next of the new node, prev of the old head, next of the new node, and the new head itself.
4. Create two traversal functions: one that uses the next pointer for forward traversal.
5. Create another traversal function that finds the tail and uses the prev pointer for backward traversal.

**Code:**

#include <stdio.h>

#include <stdlib.h>

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

void push(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); new\_node->prev = NULL;

if ((\*head\_ref) != NULL) (\*head\_ref)->prev = new\_node;

(\*head\_ref) = new\_node;

}

void printList(struct Node\* node) {

struct Node\* last;

printf("\nTraversal in forward direction \n");

while (node != NULL) { printf(" %d ", node->data); last = node; node = node->next; }

printf("\nTraversal in reverse direction \n");

while (last != NULL) { printf(" %d ", last->data); last = last->prev; }

printf("\n");

}

int main() {

struct Node\* head = NULL;

push(&head, 6); push(&head, 7); push(&head, 1);

printList(head);

return 0;

}

**Output:**

Traversal in forward direction

1 7 6

Traversal in reverse direction

6 7 1

**Result:** The program successfully implements a doubly linked list and demonstrates traversal in both forward and reverse directions.

### 39. Sum of Row and Column - Array

**Aim:** To calculate the sum of elements for each row and each column in a 2D matrix.

**Algorithm:**

1. Use nested loops to iterate through the matrix. The outer loop for rows and the inner loop for columns.
2. **Row Sum:** For each row i, initialize row\_sum to 0. In the inner loop (for columns j), add matrix[i][j] to row\_sum. Print the sum after the inner loop finishes.
3. **Column Sum:** For each column j, initialize col\_sum to 0. In the inner loop (for rows i), add matrix[i][j] to col\_sum.
4. Print the column sum after the inner loop.
5. This requires two separate sets of nested loops.

**Code:**

#include <stdio.h>

#define R 3

#define C 3

void sumRowCol(int mat[R][C]) {

int i, j, sum;

for (i = 0; i < R; ++i) {

sum = 0;

for (j = 0; j < C; ++j) sum = sum + mat[i][j];

printf("Sum of the elements of row %d = %d\n", i, sum);

}

for (i = 0; i < C; ++i) {

sum = 0;

for (j = 0; j < R; ++j) sum = sum + mat[j][i];

printf("Sum of the elements of column %d = %d\n", i, sum);

}

}

int main() {

int mat[R][C] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

sumRowCol(mat);

return 0;

}

**Output:**

Sum of the elements of row 0 = 6

Sum of the elements of row 1 = 15

Sum of the elements of row 2 = 24

Sum of the elements of column 0 = 12

Sum of the elements of column 1 = 15

Sum of the elements of column 2 = 18

**Result:** The program successfully calculates and prints the sum of each row and column in the given 2D array.

### 40. Elements Repeated Twice - Array

**Aim:** To find and print all elements in an array that are repeated exactly twice.

**Algorithm:**

1. Use a hash map or a frequency array to store the count of each element.
2. Iterate through the input array. For each element, increment its corresponding count in the frequency map.
3. After populating the map, iterate through the map's keys (or the frequency array's indices).
4. If the count for any element is exactly 2, print that element.
5. This approach efficiently handles the counting in linear time.

**Code:**

#include <stdio.h>

#include <stdlib.h>

void printRepeating(int arr[], int size) {

printf("Elements repeated twice are: ");

for (int i = 0; i < size; i++) {

int count = 0;

for (int j = 0; j < size; j++) {

if (arr[i] == arr[j]) {

count++;

}

}

if (count == 2) {

// To avoid printing the same duplicate twice

int already\_printed = 0;

for (int k = 0; k < i; k++) {

if (arr[k] == arr[i]) already\_printed = 1;

}

if (!already\_printed) printf("%d ", arr[i]);

}

}

printf("\n");

}

int main() {

int arr[] = {4, 2, 4, 5, 2, 3, 1, 5};

int arr\_size = sizeof(arr) / sizeof(arr[0]);

printRepeating(arr, arr\_size);

return 0;

}

**Output:**

Elements repeated twice are: 4 2 5

**Result:** The program successfully identifies and prints all elements that appear exactly two times in the array.

### 41. Sum of Stack Top and Bottom Elements

**Aim:** To find the sum of the top-most element of one stack and the bottom-most element of a second stack.

**Algorithm:**

1. **Get Top:** Getting the top element of the first stack is a simple peek operation.
2. **Get Bottom:** To get the bottom element of the second stack, a recursive helper function is needed.
3. The function pops an element, recursively calls itself, and if the stack becomes empty after its call, the popped element is the bottom one.
4. After finding the bottom element, the function must push the popped element back onto the stack to restore it.
5. Sum the two retrieved values.

**Code:**

#include <stdio.h>

#define MAX 100

typedef struct { int arr[MAX]; int top; } Stack;

void push(Stack\* s, int x) { s->arr[++s->top] = x; }

int pop(Stack\* s) { return s->arr[s->top--]; }

int peek(Stack\* s) { return s->arr[s->top]; }

int getBottom(Stack\* s) {

if (s->top == 0) return peek(s);

int temp = pop(s);

int bottom = getBottom(s);

push(s, temp);

return bottom;

}

int main() {

Stack s1 = {{0}, -1};

Stack s2 = {{0}, -1};

push(&s1, 1); push(&s1, 2); push(&s1, 3); // Top is 3

push(&s2, 4); push(&s2, 5); push(&s2, 6); // Bottom is 4

int top1 = peek(&s1);

int bottom2 = getBottom(&s2);

printf("Top of Stack 1: %d\n", top1);

printf("Bottom of Stack 2: %d\n", bottom2);

printf("Sum: %d\n", top1 + bottom2);

return 0;

}

**Output:**

Top of Stack 1: 3

Bottom of Stack 2: 4

Sum: 7

**Result:** The program successfully finds the top and bottom elements from two different stacks and calculates their sum.

### 42. Reverse - SLL

**Aim:** To write a C program to reverse a singly linked list.

**Algorithm:**

1. Initialize three pointers: prev as NULL, current as head, and next as NULL.
2. Iterate through the linked list while current is not NULL.
3. Inside the loop, store the next node: next = current->next.
4. Reverse the current node's pointer: current->next = prev.
5. Move pointers one position ahead: prev = current and current = next.
6. After the loop, prev will be pointing to the new head of the reversed list.

**Code:**

#include <stdio.h>

#include <stdlib.h>

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

void reverse(struct Node\*\* head\_ref) {

struct Node\* prev = NULL;

struct Node\* current = \*head\_ref;

struct Node\* next = NULL;

while (current != NULL) {

next = current->next;

current->next = prev;

prev = current;

current = next;

}

\*head\_ref = prev;

}

void push(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;

}

void printList(struct Node\* head) {

struct Node\* temp = head;

while (temp != NULL) {

printf("%d ", temp->data);

temp = temp->next;

}

printf("\n");

}

int main() {

struct Node\* head = NULL;

push(&head, 20); push(&head, 4); push(&head, 15); push(&head, 85);

printf("Given linked list\n"); printList(head);

reverse(&head);

printf("\nReversed linked list \n"); printList(head);

return 0;

}

**Output:**

Given linked list

85 15 4 20

Reversed linked list

20 4 15 85

**Result:** The program successfully reverses the given singly linked list using an iterative approach.