main.c

Share | Run

```c
#include <stdio.h>
int main()
{
    int n;
    printf("Input the number of elements to store in the array: ");
    scanf("%d", &n);
    int array[n];
    printf("Input %d number of elements in the array:\n", n);
    for (int i = 0; i < n; i++) {
        printf("element - %d : ", i);
        scanf("%d", &array[i]);
    }
    printf("The elements in reverse order are:\n");
    for (int i = n - 1; i >= 0; i--) {
        printf("element - %d : %d\n", i, array[i]);
    }
    return 0;
}

```

Output | Clear

```
2
2 5 7
Input 2 number of elements in the array:
element - 0 : element - 1 : The elements in reverse order are:
element - 1 : 5
element - 0 : 2


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

main.c

Share Run Output

```c
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>
#include <string.h>
bool isNonEmpty(const char *str) {
    return str != NULL && str[0] != '\0';
}
bool isAlphanumeric(const char *str) {
    if (!isNonEmpty(str)) return false;

    for (int i = 0; str[i] != '\0'; i++) {
        if (!isalnum(str[i])) {
            return false;
        }
    }
    return true;
}
bool isAlphabetic(const char *str) {
    if (!isNonEmpty(str)) return false;

    for (int i = 0; str[i] != '\0'; i++) {
        if (!isalpha(str[i])) {
            return false;
        }
    }
    return true;
```

main.c

Share Run Outp

```c
}
bool isNumeric(const char *str) {
    if (!isNonEmpty(str)) return false
    for (int i = 0; str[i] != '\0'; i++) {
        if (!isdigit(str[i])) {
            return false;
        }
    }
    return true;
}
int main()
{
    char str[100];
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';
    printf("Checking string: '%s'\n", str);
    if (isNonEmpty(str)) {
        printf("The string is non-empty.\n");
    } else {
        printf("The string is empty.\n");
    }
    if (isAlphanumeric(str))
    {
        printf("The string is alphanumeric.\n");
    } else {
```

```
    } else {
        printf("The string is not alphanumeric.\n");
    }
    if (isAlphabetic(str))
    {
        printf("The string is alphabetic.\n");
    } else {
        printf("The string is not alphabetic.\n");
    }
    if (isNumeric(str))
    {
        printf("The string is numeric.\n");
    } else {
        printf("The string is not numeric.\n");
    }
    return 0;
}

```

main.c

Share Run Ou

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>S
typedef struct {
    int top;
    int capacity;
    int *array;
} Stack;
Stack* createStack(int capacity) {
    Stack *stack = (Stack*) malloc(sizeof(Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));
    return stack;
}
bool isFull(Stack *stack) {
    return stack->top == stack->capacity - 1;
}
bool isEmpty(Stack *stack) {
    return stack->top == -1;
}
void push(Stack *stack, int item) {
    if (isFull(stack)) {
        printf("Stack overflow\n");
        return;
    }
```

main.c

Share Run Outp

```
27      stack->array[++stack->top] = item;
28  }
29  int pop(Stack *stack) {
30      if (isEmpty(stack)) {
31          printf("Stack underflow\n");
32          return -1;
33      }
34      return stack->array[stack->top--];
35  }
36  bool validateStackSequences(int *pushed, int pushedSize, int *popped, int
        poppedSize) {
37      Stack *stack = createStack(pushedSize);
38      int pushIndex = 0;
39      int popIndex = 0;
40      while (popIndex < poppedSize) {
41          while (pushIndex < pushedSize && (isEmpty(stack) || stack->array[stack
                ->top] != popped[popIndex])) {
42              push(stack, pushed[pushIndex++]);
43          }
44          if (!isEmpty(stack) && stack->array[stack->top] == popped[popIndex]) {
45              pop(stack);
46              popIndex++;
47          } else {
48              free(stack->array);
49              free(stack);
50              return false;
```

main.c

Share Run

```
45              pop(stack);
46              popIndex++;
47          } else {
48              free(stack->array);
49              free(stack);
50              return false;
51          }
52      }
53      bool isValid = isEmpty(stack);
54      free(stack->array);
55      free(stack);
56      return isValid;
57  }
58  int main() {
59      int pushed[] = {1, 2, 3, 4, 5};
60      int popped[] = {4, 5, 3, 2, 1};
61      int pushedSize = sizeof(pushed) / sizeof(pushed[0]);
62      int poppedSize = sizeof(popped) / sizeof(popped[0]);
63      if (validateStackSequences(pushed, pushedSize, popped, poppedSize)) {
64          printf("True\n");
65      } else {
66          printf("False\n");
67      }
68      return 0;
69  }
70
```

main.c

Share Run Output

```c
#include <stdio.h>
#include <stdlib.h>
void mergeArrays(int *arr1, int size1, int *arr2, int size2, int *arr3) {
    int i, j;
    for (i = 0; i < size1; i++) {
        arr3[i] = arr1[i];
    }
    for (j = 0; j < size2; j++) {
        arr3[i + j] = arr2[j];
    }
}
int main()
{
    int arr1[] = {1, 2, 3, 4, 5};
    int arr2[] = {6, 7, 8, 9, 10};
    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int size2 = sizeof(arr2) / sizeof(arr2[0]);
    int *arr3 = (int *) malloc((size1 + size2) * sizeof(int));
    if (arr3 == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    mergeArrays(arr1, size1, arr2, size2, arr3);
    printf("Merged array: [");
    for (int i = 0; i < size1 + size2; i++) {
        printf("%d", arr3[i]);
```

```c
        printf("Memory allocation failed\n");
        return 1;
    }
    mergeArrays(arr1, size1, arr2, size2, arr3);
    printf("Merged array: [");
    for (int i = 0; i < size1 + size2; i++) {
        printf("%d", arr3[i]);
        if (i < size1 + size2 - 1) {
            printf(", ");
        }
    }
    printf("]\n");
    free(arr3);
    return 0;
}

```

main.c

Share Run

```c
#include <stdio.h>
#include <limits.h>
#define MAX_NODES 100
#define INF INT_MAX
void dijkstra(int graph[MAX_NODES][MAX_NODES], int num_nodes, int src, int
    dist[], int prev[]) {
    int visited[MAX_NODES] = {0};
    for (int i = 0; i < num_nodes; i++) {
        dist[i] = INF;
        prev[i] = -1;
    }
    dist[src] = 0;
    for (int count = 0; count < num_nodes - 1; count++) {
        int min = INF, u = -1;
        for (int i = 0; i < num_nodes; i++) {
            if (!visited[i] && dist[i] < min) {
                min = dist[i];
                u = i;
            }
        }
        if (u == -1) {
            break;
        }
        visited[u] = 1;
        for (int v = 0; v < num_nodes; v++) {
            if (!visited[v] && graph[u][v] && dist[u] != INF && dist[u] +
                graph[u][v] < dist[v]) {
```

main.c

Share Run Output

```
                graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
                prev[v] = u;
            }
        }
    }
}
void printPath(int prev[], int j) {
    if (prev[j] == -1) {
        printf("%d ", j + 1);
        return;
    }
    printPath(prev, prev[j]);
    printf("to %d ", j + 1);
}
int main() {
    int num_nodes;
    int graph[MAX_NODES][MAX_NODES];
    int dist[MAX_NODES], prev[MAX_NODES];
    int source, target;
    printf("Enter number of nodes\n");
    scanf("%d", &num_nodes);
    printf("Enter weight of all the paths in adjacency matrix form\n");
    for (int i = 0; i < num_nodes; i++) {
        for (int j = 0; j < num_nodes; j++) {
            scanf("%d", &graph[i][j]);
        }
```

```
51          }
52      }
53      printf("Enter the source\n");
54      scanf("%d", &source);
55      source--;
56      printf("Enter the target\n");
57      scanf("%d", &target);
58      target--;
59      dijkstra(graph, num_nodes, source, dist, prev);
60      printf("Shortest path from %d to %d: ", source + 1, target + 1);
61      if (dist[target] == INF) {
62          printf("No path exists\n");
63      } else {
64          printPath(prev, target);
65          printf("\n");
66      }
67      return 0;
68  }
69
```

main.c

Share Run Output

```c
#include <stdio.h>
int main() {
    int num_elements;
    printf("Input the number of elements to be stored in the array: ");
    scanf("%d", &num_elements);
    int array[num_elements];
    printf("Input %d elements in the array:\n", num_elements);
    for (int i = 0; i < num_elements; i++) {
        printf("element - %d : ", i);
        scanf("%d", &array[i]);
    }
    int count_duplicates = 0;
    int visited[num_elements];
    for (int i = 0; i < num_elements; i++) {
        visited[i] = 0;
    }
    for (int i = 0; i < num_elements; i++) {
        if (visited[i] == 1) {
            continue;
        }
        int found_duplicate = 0;
        for (int j = i + 1; j < num_elements; j++) {
            if (array[i] == array[j]) {
                found_duplicate = 1;
                visited[j] = 1;
            }
        }
```

```
            if (array[i] == array[j]) {
                found_duplicate = 1;
                visited[j] = 1;
            }
        }
        if (found_duplicate) {
            count_duplicates++;
        }
    }
    printf("Total number of duplicate elements: %d\n", count_duplicates);
    return 0;
}

```

main.c

Share Run Output

```c
#include <stdio.h>
#include <limits.h>D
#define MAX_CITIES 10
#define INF INT_MAX
int tsp(int dist[MAX_CITIES][MAX_CITIES], int path[], int visited[], int n,
    int pos, int count, int cost, int minCost) {
    if (count == n && dist[pos][0]) {
        return cost + dist[pos][0] < minCost ? cost + dist[pos][0] : minCost;
    }
    for (int i = 0; i < n; i++) {
        if (!visited[i] && dist[pos][i]) {
            visited[i] = 1;
            path[count] = i;
            minCost = tsp(dist, path, visited, n, i, count + 1, cost +
                dist[pos][i], minCost);
            visited[i] = 0;
        }
    }
    return minCost;
}
int main() {
    int n;
    int dist[MAX_CITIES][MAX_CITIES];
    int path[MAX_CITIES];
    int visited[MAX_CITIES] = {0};
    int minCost = INF;
    printf("Enter number of cities: ");
```

```
            visited[i] = 0;
        }
    }
    return minCost;
}
int main() {
    int n;
    int dist[MAX_CITIES][MAX_CITIES];
    int path[MAX_CITIES];
    int visited[MAX_CITIES] = {0};
    int minCost = INF;
    printf("Enter number of cities: ");
    scanf("%d", &n);
    printf("Enter distance matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &dist[i][j]);
        }
    }
    visited[0] = 1;
    path[0] = 0;
    minCost = tsp(dist, path, visited, n, 0, 1, 0, minCost);
    printf("The minimum cost of visiting all cities is %d\n", minCost);
    return 0;
}

```

main.c

Share Run

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
struct Node* mergeLists(struct Node* l1, struct Node* l2) {
    struct Node* mergedHead = NULL;
    struct Node* mergedTail = NULL;
    struct Node dummy;
    mergedTail = &dummy;
    dummy.next = NULL;
    while (l1 != NULL && l2 != NULL) {
```

main.c

Share Run

```
28          if (l1->data <= l2->data) {
29              mergedTail->next = l1;
30              l1 = l1->next;
31          } else {
32              mergedTail->next = l2;
33              l2 = l2->next;
34          }
35          mergedTail = mergedTail->next;
36      }
37      if (l1 != NULL) {
38          mergedTail->next = l1;
39      } else {
40          mergedTail->next = l2;
41      }
42      return dummy.next;
43  }
44  void appendNode(struct Node** headRef, int data) {
45      struct Node* newNode = createNode(data);
46      if (*headRef == NULL) {
47          *headRef = newNode;
48          return;
49      }
50      struct Node* temp = *headRef;
51      while (temp->next != NULL) {
52          temp = temp->next;
53      }
54      temp->next = newNode;
```

main.c

Share Run Outp

```
    temp->next = newNode;
}
int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;
    struct Node* mergedList = NULL;
    printf("Enter number of elements in the first sorted list: ");
    int n1;
    scanf("%d", &n1);
    printf("Enter %d elements in the first sorted list:\n", n1);
    for (int i = 0; i < n1; i++) {
        int data;
        scanf("%d", &data);
        appendNode(&list1, data);
    }
    printf("Enter number of elements in the second sorted list: ");
    int n2;
    scanf("%d", &n2);
    printf("Enter %d elements in the second sorted list:\n", n2);
    for (int i = 0; i < n2; i++) {
        int data;
        scanf("%d", &data);
        appendNode(&list2, data);
    }
    mergedList = mergeLists(list1, list2);
    printf("Merged sorted list:\n");
    printList(mergedList);
```

```
    for (int i = 0; i < n2; i++) {
        int data;
        scanf("%d", &data);
        appendNode(&list2, data);
    }
    mergedList = mergeLists(list1, list2);
    printf("Merged sorted list:\n");
    printList(mergedList);
    return 0;
}

```

main.c

Share Run Output

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insertNode(root->left, data);
    } else if (data > root->data) {
        root->right = insertNode(root->right, data);
    }
    return root;
}
struct Node* searchNode(struct Node* root, int key) {
    if (root == NULL || root->data == key) {
```

main.c

Share Run Output

```c
struct Node* searchNode(struct Node* root, int key) {
    if (root == NULL || root->data == key) {
        return root;
    }
    if (key < root->data) {
        return searchNode(root->left, key);
    }
    return searchNode(root->right, key);
}
struct Node* findMin(struct Node* root) {
    struct Node* current = root;
    while (current && current->left != NULL) {
        current = current->left;
    }
    return current;
}
struct Node* findMax(struct Node* root) {
    struct Node* current = root;
    while (current && current->right != NULL) {
        current = current->right;
    }
    return current;
}
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
```

main.c

Share Run

```
51          inorderTraversal(root->left);
52          printf("%d ", root->data);
53          inorderTraversal(root->right);
54      }
55  }
56  int main() {
57      struct Node* root = NULL;
58      int choice, value;
59      while (1) {
60          printf("\nBinary Search Tree Operations:\n");
61          printf("1. Insert a Node\n");
62          printf("2. Search for a Node\n");
63          printf("3. Find Minimum Element\n");
64          printf("4. Find Maximum Element\n");
65          printf("5. Print In-Order Traversal\n");
66          printf("6. Exit\n");
67          printf("Enter your choice: ");
68          scanf("%d", &choice);
69          switch (choice) {
70              case 1:
71                  printf("Enter value to insert: ");
72                  scanf("%d", &value);
73                  root = insertNode(root, value);
74                  break;
75              case 2:
76                  printf("Enter value to search: ");
77                  scanf("%d", &value);
```

main.c

Share | Run | Output

```
75          case 2:
76              printf("Enter value to search: ");
77              scanf("%d", &value);
78              if (searchNode(root, value) != NULL) {
79                  printf("Value %d found in the BST.\n", value);
80              } else {
81                  printf("Value %d not found in the BST.\n", value);
82              }
83              break;
84          case 3:
85              {
86                  struct Node* minNode = findMin(root);
87                  if (minNode != NULL) {
88                      printf("Minimum value in the BST is %d.\n", minNode
                            ->data);
89                  } else {
90                      printf("The BST is empty.\n");
91                  }
92              }
93              break;
94          case 4:
95              {
96                  struct Node* maxNode = findMax(root);
97                  if (maxNode != NULL) {
98                      printf("Maximum value in the BST is %d.\n", maxNo
99
```

```
93                  break;
94              case 4:
95                  {
96                      struct Node* maxNode = findMax(root);
97                      if (maxNode != NULL) {
98                          printf("Maximum value in the BST is %d.\n", maxNo)
99                          printf("Invalid choice. Please try again.\n");
100         }
101     }
102     return 0;
103 }
104
```