

DSA Lab # 3

22F-3104



September 7, 2023

FAST CFD Campus

**Task no 1:**

Code:

#include <iostream>

using namespace std;

template <class T>

class ListADT {

public:

T arr[10]; // The array to store elements

int count; // Current number of elements in the array

ListADT() {

count = 0; // Initialize the count to 0 when an object is created

}

// Rearrange the elements of the array so that even elements come before odd elements

void Rearrange() {

T arr2[10]; // Temporary array to store rearranged elements

int evenCount = 0; // Counter for even elements

int oddCount = count; // Counter for odd elements

// Iterate through the array and move even elements to the front

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

if (arr[i] % 2 == 0) {

arr2[evenCount++] = arr[i];

}

}

// Iterate through the array again and move odd elements after even elements

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

if (arr[i] % 2 == 1) {

arr2[evenCount++] = arr[i];

}

}

// Copy the rearranged elements back to the original array

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

arr[i] = arr2[i];

}

}

// Insert an element at a specified position in the array

void insert() {

T num; // Element to be inserted

int position; // Position where the element should be inserted

cout << "\nEnter element to insert: ";

cin >> num;

cout << "\nEnter position to insert: ";

cin >> position;

if (position >= 0 && position <= count) {

// Shift elements to make space for the new element

for (int i = count; i > position; i--) {

arr[i] = arr[i - 1];

}

arr[position] = num;

count++;

cout << "\nArray after insertion: ";

Display();

}

else {

cout << "\nInvalid position!";

}

}

// Remove all elements from the array

void remove() {

cout << "After removing all elements :";

count = 0; // Set count to 0 to indicate an empty array

Display();

}

// Display the elements of the array

void Display() {

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

cout << arr[i] << " ";

}

cout << endl;

}

// Remove an element at a specified position from the array

void removeAt() {

int position; // Position from which an element should be removed

cout << "\nEnter position to remove: ";

cin >> position;

if (position >= 0 && position < count) {

// Shift elements to fill the gap left by the removed element

for (int i = position; i < count - 1; i++) {

arr[i] = arr[i + 1];

}

count--;

cout << "\nArray after removing at position: ";

Display();

}

else {

cout << "\nInvalid position!";

}

}

// Replace an element at a specified position with a new element

void Replace() {

T num; // New element to replace the existing one

int position; // Position of the element to be replaced

cout << "\nEnter element to replace: ";

cin >> num;

cout << "\nEnter position of element to replace: ";

cin >> position;

if (position >= 0 && position < count) {

arr[position] = num;

cout << "\nArray after replacing at position: ";

Display();

}

else {

cout << "\nInvalid position!";

}

}

// Display the size of the array

void size() {

cout << "The size of the array is: " << count;

}

// Check if the array is empty and display a message

void IsEmpty() {

cout << "Array is empty.\n";

}

// Check if the array is filled and display a message

void IsFill() {

cout << "Array is :";

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

if (arr[i] != NULL) {

cout << arr[i] << " ";

}

}

cout << " Filled.\n";

}

// Add an element to the array if it is not already full

void get() {

T num;

if (count < 10) {

cout << "\nEnter element to add: ";

cin >> num;

arr[count++] = num;

cout << "\nArray after adding: ";

Display();

}

else {

cout << "\nArray is already filled!";

}

}

};

int main() {

ListADT<int> obj;

int choice = 0;

// Initialize the array with 5 elements

for (int i = 0; i < 5; i++) {

cout << "Enter Element of the List [" << i + 1 << "] = ";

cin >> obj.arr[i];

obj.count++;

}

// Main menu to select the choice

while (choice != -1) {

cout << "\nPress 1 for Rearrange the List :";

cout << "\nPress 2 for Insert Element in the List :";

cout << "\nPress 3 for Removing all Element in the List :";

cout << "\nPress 4 to RemoveAt specific index in the List :";

cout << "\nPress 5 to Replace at Specific index in the Array :";

cout << "\nPress 6 to check the size of the List :";

cout << "\nPress 7 to add an element to the List :";

cout << "\nPress -1 to exit: ";

cin >> choice;

// Check if the array is empty or filled before and after each operation

int i = 0;

if (obj.arr[i] != NULL) {

obj.IsFill();

}

else if (obj.arr[i] == NULL) {

obj.IsEmpty();

}

i++;

switch (choice) {

case 1:

obj.Rearrange();

cout << "\nRearranged List: ";

obj.Display();

break;

case 2:

obj.insert();

break;

case 3:

obj.remove();

break;

case 4:

obj.removeAt();

break;

case 5:

obj.Replace();

break;

case 6:

obj.size();

break;

case 7:

obj.get();

break;

case -1:

break;

default:

cout << "\nInvalid choice!";

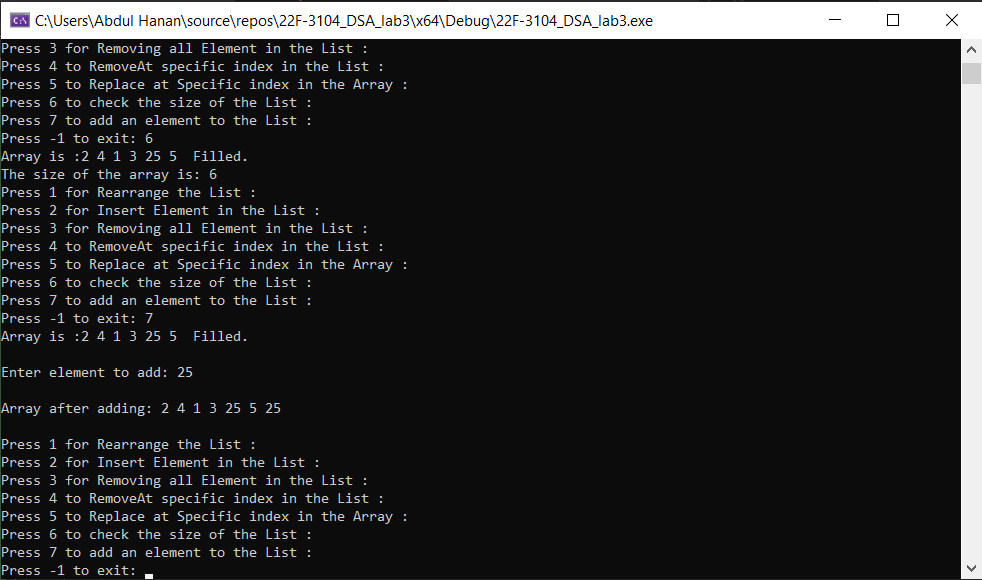
}

}

return 0;

}

Screen Shot:



**Task no 2:**

Code:

#include<iostream>

using namespace std;

class ListADT {

private:

int Arr[5];

int Arr2[5];

int Arr3[10]; // Increased size for the union array to accommodate all possible elements

public:

ListADT() {}

void Array() {

cout << "Enter elements for Array 1: ";

for (int i = 0; i < 5; i++) {

cin >> Arr[i];

}

}

void Array2() {

cout << "Enter elements for Array 2: ";

for (int i = 0; i < 5; i++) {

cin >> Arr2[i];

}

}

void intersection() {

cout << "Intersection of two lists are: ";

int count = 0; // Keep track of the number of intersection elements

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 5; j++) {

if (Arr[i] == Arr2[j]) {

Arr3[count++] = Arr[i]; // Store the intersection element

}

}

}

// Print the intersection elements

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

cout << Arr3[i] << " ";

}

cout << endl;

}

void setUnion() {

cout << "Union of two lists are: ";

int count = 0; // Keep track of the number of union elements

// Copy elements from Arr to Arr3

for (int i = 0; i < 5; i++) {

Arr3[count++] = Arr[i];

}

// Check for duplicate elements while adding elements from Arr2 to Arr3

for (int i = 0; i < 5; i++) {

bool isDuplicate = false;

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

if (Arr2[i] == Arr3[j]) {

isDuplicate = true;

break;

}

}

if (!isDuplicate) {

Arr3[count++] = Arr2[i];

}

}

// Print the union elements

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

cout << Arr3[i] << " ";

}

cout << endl;

}

};

int main() {

ListADT obj;

obj.Array();

obj.Array2();

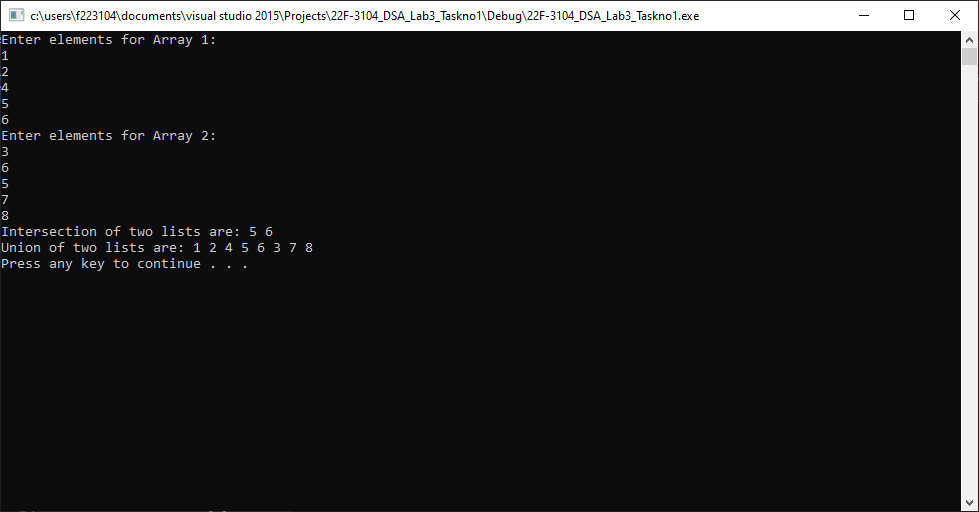
obj.intersection();

obj.setUnion();

system("pause");

return 0;

}

Screen Shot: 

**Task no 3:**

**Code:**

#include <iostream>

using namespace std;

// Define a class for a linked list node

class Node {

public:

int data;

Node\* next;

Node(int val) {

data = val;

next = NULL;

}

};

// Define a class for a linked list

class LinkedList {

public:

Node\* head;

LinkedList() {

head = NULL;

}

// Function to insert a new node at the end of the linked list

void insert(int val) {

Node\* newNode = new Node(val);

if (head == NULL) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

// Function to concatenate the second list to the first list

void concatenate(LinkedList& secondList) {

if (head == NULL) {

head = secondList.head;

}

else {

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = secondList.head;

}

secondList.head = NULL; // Set the second list to an empty list

}

// Function to display the elements of the linked list

void display() {

Node\* temp = head;

while (temp != NULL) {

cout << temp->data << " -> ";

temp = temp->next;

}

cout << "NULL" << endl;

}

};

int main() {

LinkedList list1;

LinkedList list2;

// Insert elements into the first list

list1.insert(1);

list1.insert(2);

list1.insert(3);

// Insert elements into the second list

list2.insert(4);

list2.insert(5);

list2.insert(6);

cout << "List 1: ";

list1.display();

cout << "List 2: ";

list2.display();

// Concatenate the second list to the first list

list1.concatenate(list2);

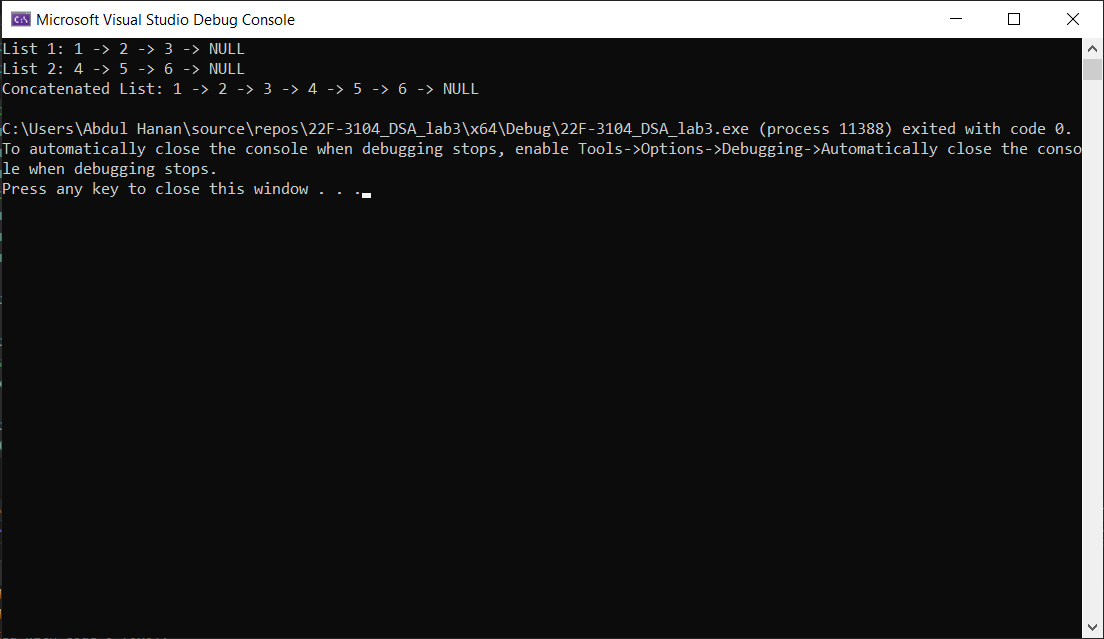
cout << "Concatenated List: ";

list1.display();

return 0;

}

**Screen Shot:**

****

**Task no 4:**

Code:

#include <iostream>

using namespace std;

// Define a class for a linked list node

class node {

public:

int data; // Data stored in the node

node\* next; // Pointer to the next node

// Constructor to initialize a node with given data

node(int val) {

data = val;

next = NULL;

}

};

// Function to insert a new node with a given value at the end of the linked list

void insert(node\*& head, int val) {

node\* n = new node(val); // Create a new node with the given value

if (head == NULL) {

head = n; // If the linked list is empty, set the new node as the head

return;

}

node\* temp = head; // Initialize a temporary pointer to traverse the list

while (temp->next != NULL) {

temp = temp->next; // Traverse to the last node in the list

}

temp->next = n; // Set the new node as the next node of the last node

}

// Function to display the elements of the linked list

void display(node\* head) {

while (head != NULL) {

cout << head->data << "->"; // Print the data of the current node

head = head->next; // Move to the next node

}

cout << "NULL" << endl; // Print "NULL" to indicate the end of the list

}

// Function to merge two sorted linked lists into a new sorted linked list

node\* merge(node\*& head1, node\*& head2) {

node\* ptr1 = head1; // Pointer to traverse the first linked list

node\* ptr2 = head2; // Pointer to traverse the second linked list

node\* temp = new node(-1); // Create a dummy node to simplify merging

node\* ptr3 = temp; // Pointer to build the merged list

while (ptr1 != NULL && ptr2 != NULL) {

if (ptr1->data < ptr2->data) {

ptr3->next = ptr1; // Append the smaller node from the first list

ptr1 = ptr1->next;

}

else {

ptr3->next = ptr2; // Append the smaller node from the second list

ptr2 = ptr2->next;

}

ptr3 = ptr3->next; // Move to the next node in the merged list

}

// Append any remaining nodes from both lists

while (ptr1 != NULL) {

ptr3->next = ptr1;

ptr1 = ptr1->next;

ptr3 = ptr3->next;

}

while (ptr2 != NULL) {

ptr3->next = ptr2;

ptr2 = ptr2->next;

ptr3 = ptr3->next;

}

return temp->next; // Return the merged linked list (excluding the dummy node)

}

int main() {

node\* head1 = NULL; // Initialize the first linked list

node\* head2 = NULL; // Initialize the second linked list

int arr1[4] = { 1, 4, 5, 7 }; // Elements for the first list

int arr2[3] = { 2, 3, 6 }; // Elements for the second list

// Insert elements from the arrays into the respective linked lists

for (int i = 0; i < 4; i++) {

insert(head1, arr1[i]);

}

for (int i = 0; i < 3; i++) {

insert(head2, arr2[i]);

}

// Display the original lists

cout << "List 1: ";

display(head1);

cout << "List 2: ";

display(head2);

// Merge the two sorted lists and display the merged list

node\* mergedList = merge(head1, head2);

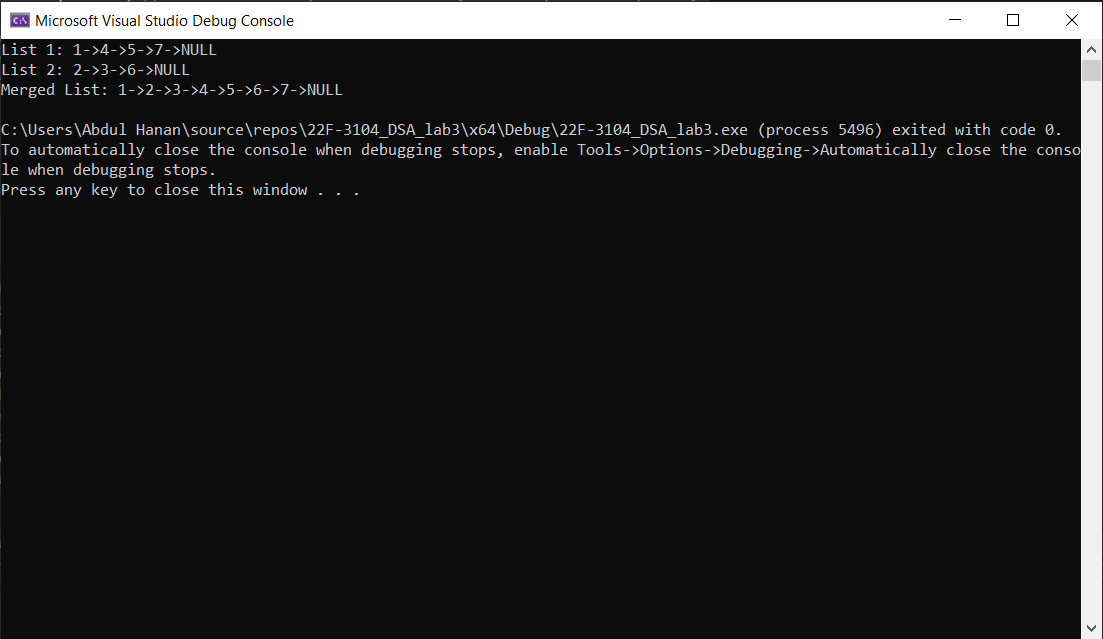
cout << "Merged List: ";

display(mergedList);

return 0;

}

Screen Shot:



**Task no 5:**

**Code:**

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

class node {

public:

int data;

node\* next;

node(int val) {

data = val;

next = NULL;

}

};

// Function to insert a node with 'val' at the end of the linked list 'head'

void insertAtTail(node\*& head, int val) {

node\* n = new node(val);

if (head == NULL) {

head = n;

return;

}

node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = n;

}

// Function to display the elements in the linked list 'head'

void display(node\* head) {

while (head != NULL) {

cout << head->data << " -> ";

head = head->next;

}

cout << "NULL" << endl;

}

int main() {

node\* head = NULL;

int RandArray[30];

// Initialize the random number generator with the current time

srand((time(0)));

// Generate 30 random numbers and add them to both an array and a linked list

for (int i = 0; i < 30; i++) {

int randomNum = rand() % 1001; // Generate random numbers between 0 and 1000

RandArray[i] = randomNum;

insertAtTail(head, randomNum);

}

// Calculate the sum of the elements in the array

int sum = 0;

for (int i = 0; i < 30; i++) {

sum += RandArray[i];

}

// Calculate the average of the elements in the array

int Average = 0;

Average = sum / 30;

// Find the maximum and minimum values in the array

int Max = 0;

int Min = RandArray[0];

for (int i = 0; i < 30; i++) {

if (RandArray[i] > Max) {

Max = RandArray[i];

}

if (RandArray[i] < Min) {

Min = RandArray[i];

}

}

// Display the linked list and the calculated statistics

cout << "Random number generated Linked List: ";

display(head);

cout << "\n\nSum of linked list is : " << sum;

cout << "\n\nAverage of linked list is : " << Average;

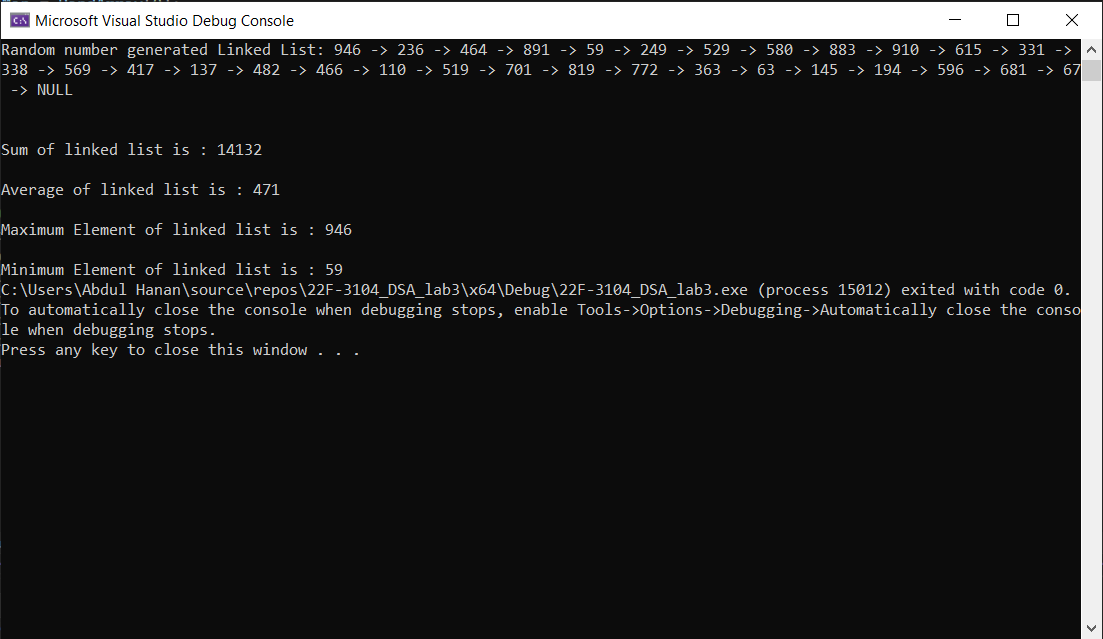
cout << "\n\nMaximum Element of linked list is : " << Max;

cout << "\n\nMinimum Element of linked list is : " << Min;

return 0;

}

**Screen Shot:**

****

**Task no 6:**

**Code:**

#include <iostream>

using namespace std;

class node {

public:

char data;

node\* next;

// Constructor to initialize a node with a given value

node(char val) {

data = val;

next = NULL;

}

};

class LinkedList {

public:

node\* head;

// Constructor to initialize the linked list with an empty head

LinkedList() {

head = NULL;

}

// Function to insert a new node at the end of the linked list

void insert(char val) {

node\* newNode = new node(val);

// If the list is empty set the new node as the head

if (head == NULL) {

head = newNode;

return;

}

// Traverse the list to find the last node and link the new node

node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

// Function to reverse the linked list

void reverse() {

node\* prev = NULL;

node\* current = head;

node\* nextNode = NULL;

// Reverse the links between nodes to reverse the list

while (current != NULL) {

nextNode = current->next;

current->next = prev;

prev = current;

current = nextNode;

}

// Update the head to point to the new first node

head = prev;

}

// Function to create a copy of the linked list

LinkedList copy() {

LinkedList newList;

node\* temp = head;

// Traverse the original list and insert each element into the new list

while (temp != NULL) {

newList.insert(temp->data);

temp = temp->next;

}

return newList;

}

// Function to display the elements of the linked list

void display() {

node\* temp = head;

// Traverse the list and print each element

while (temp != NULL) {

cout << temp->data << " -> ";

temp = temp->next;

}

cout << "NULL" << endl;

}

};

int main() {

LinkedList list1;

LinkedList list2;

// Insert 15 characters into the first list

for (char i = 'A'; i <= 'O'; i++) {

list1.insert(i);

}

// Display the original list

cout << "Original List: ";

list1.display();

// Create a copy of the first list in reverse order

list2 = list1.copy();

list2.reverse();

// Display the reversed copy

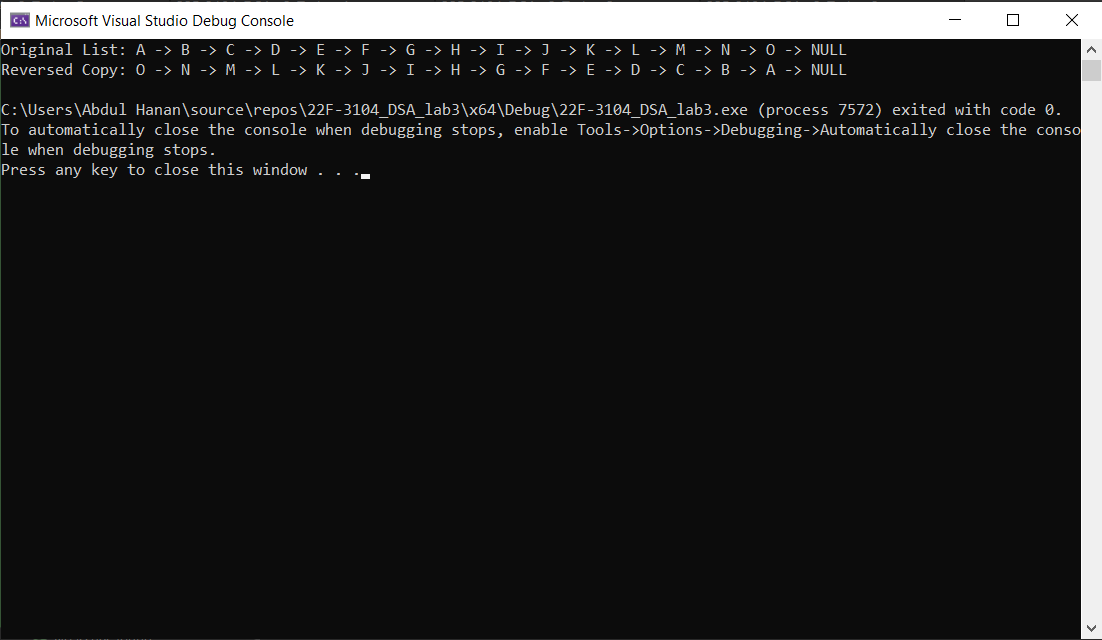
cout << "Reversed Copy: ";

list2.display();

return 0;

}

**Screen Shot:**

****

**Task no 7:**

**Code:**

#include <iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node(int val) {

data = val;

next = NULL;

}

};

// Function to insert a new node at the end of the linked list

void insert(node\*& head, int val) {

node\* nn = new node(val); // Create a new node with the given value

if (head == NULL) {

head = nn; // If the list is empty set the new node as the head

return;

}

node\* temp = head;

while (temp->next != NULL) {

temp = temp->next; // Traverse to the end of the list

}

temp->next = nn; // Connect the last node to the new node

}

// Function to display the elements of the linked list

void display(node\* head) {

while (head != NULL) {

cout << head->data << " -> "; // Print the data of the current node

head = head->next; // Move to the next node

}

cout << "NULL" << endl; // Print NULL to indicate the end of the list

}

// Function to insert a new node in the middle of the linked list

void insertInMiddle(node\*& head, int val) {

int count = 0;

node\* temphead = head;

while (temphead != NULL) {

temphead = temphead->next;

count++;

}

int mid = count / 2; // Calculate the middle position

if (mid == 0) {

node\* nn = new node(val);

nn->next = head;

head = nn; // If there is only one node insert before it

}

else {

int c = 0;

temphead = head;

while (temphead != NULL) {

if (c == mid - 1) {

node\* nn = new node(val); // Create a new node with the given value

nn->next = temphead->next; // Connect the new node to the next node

temphead->next = nn; // Connect the current node to the new node

break;

}

temphead = temphead->next; // Traverse the list

c++;

}

}

}

int main() {

node\* head = NULL; // Initialize an empty linked list

int nn;

cout << "Enter the number of nodes for the linked list: ";

cin >> nn;

if (nn % 2 == 0) {

cout << "The number of nodes should be odd." << endl; // Check for an odd number of nodes

return 1;

}

for (int i = 1; i <= nn; i++) {

int val;

cout << "Enter the value for node " << i << ": ";

cin >> val;

insert(head, val); // Insert each node at the end of the linked list

}

cout << "Original Linked List: ";

display(head); // Display the original linked list

// Check if the number of nodes is odd and insert a new node in the middle

int newVal1;

cout << "Enter the value to insert in the middle: ";

cin >> newVal1;

if ((nn % 2) != 0) {

insertInMiddle(head, newVal1); // Insert a new node in the middle

cout << "Updated Linked List (after inserting a new node due to odd nodes): ";

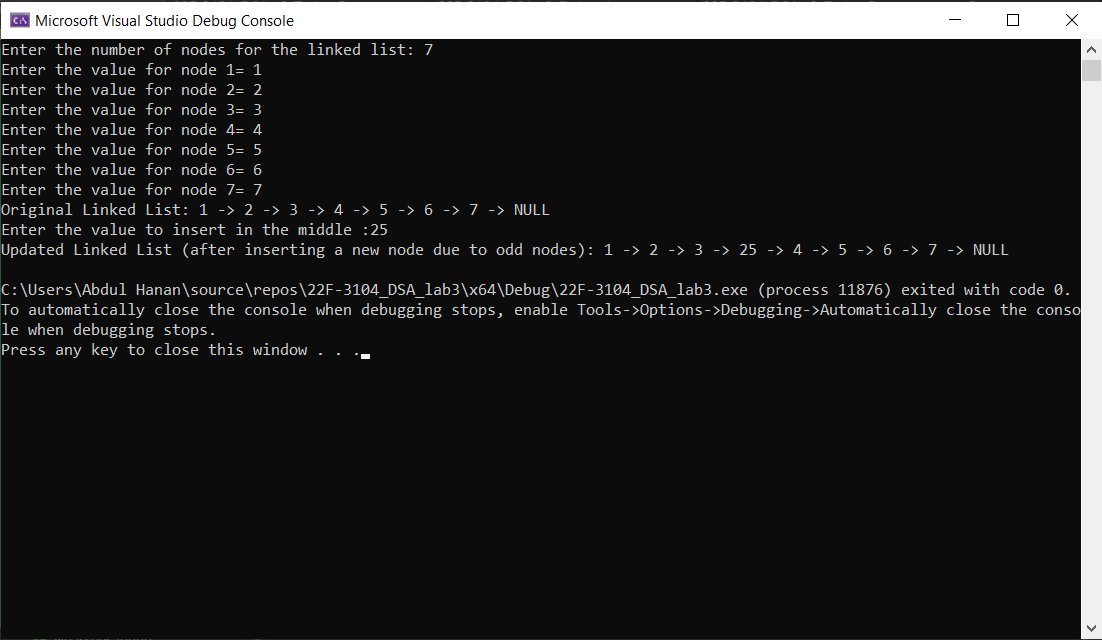
display(head); // Display the updated linked list

}

return 0;

}

**Screen Shot:**

****

**Task no 8:**

**Code:**

#include <iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node(int val) {

data = val;

next = NULL;

}

};

// Function to insert a new node at the end of the linked list

void insert(node\*& head, int val) {

// Create a new node with the given value

node\* nn = new node(val);

// If the linked list is empty set the new node as the head

if (head == NULL) {

head = nn;

return;

}

// Traverse the linked list to find the last node

node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

// Append the new node to the end of the linked list

temp->next = nn;

}

// Function to display the elements of the linked list

void display(node\* head) {

// Traverse the linked list and print each element

while (head != NULL) {

cout << head->data << " -> ";

head = head->next;

}

cout << "NULL" << endl;

}

// Function to remove duplicate elements from a sorted linked list

void remove\_Duplicates(node\* head) {

if (head == NULL) {

return;

}

node\* current = head;

int duplicate\_count = 0;

while (current->next != NULL) {

// If the current nodes data is the same as the next node data remove the next node

if (current->data == current->next->data) {

node\* duplicate = current->next;

current->next = current->next->next;

delete duplicate;

duplicate\_count++;

}

else {

current = current->next;

}

}

// Display the number of duplicates removed

cout << "Number of duplicates removed: " << duplicate\_count << endl;

}

int main() {

node\* head = NULL;

// Insert elements in decreasing order (sorted)

int arr[] = { 10, 8, 8, 6, 4, 4, 2 };

for (int i = 0; i < sizeof(arr) / sizeof(arr[0]); i++) {

insert(head, arr[i]);

}

// Display the original linked list

cout << "Original Linked List: ";

display(head);

// Remove duplicates from the linked list

remove\_Duplicates(head);

// Display the linked list after removing duplicates

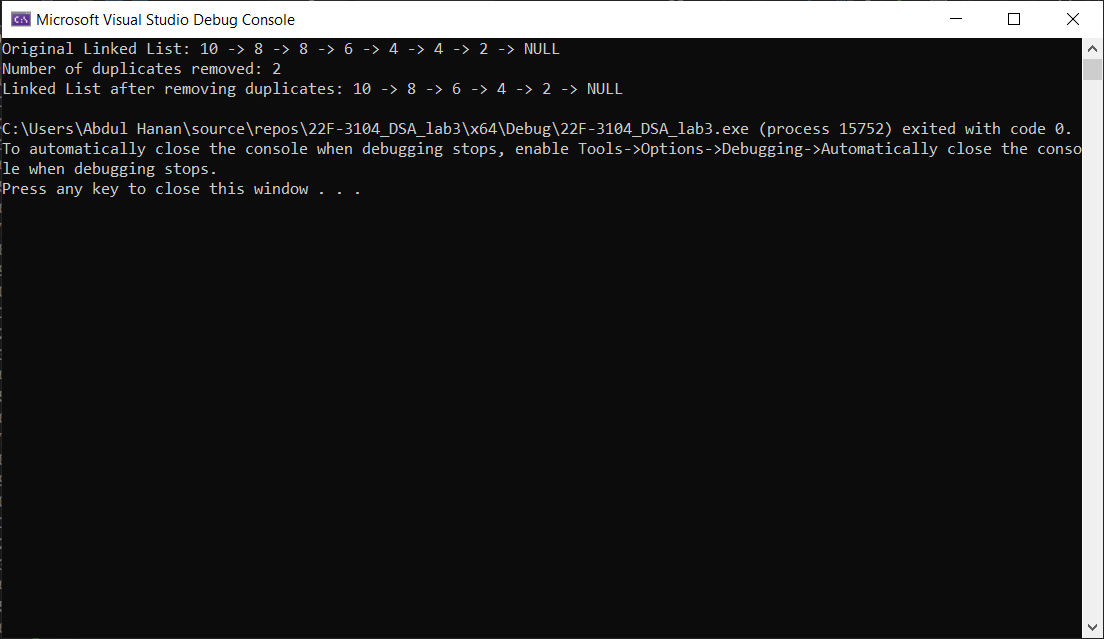
cout << "Linked List after removing duplicates: ";

display(head);

return 0;

}

**Screen Shot:**

****

**Task no 9:**

**Code:**

#include <iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node(int val) {

data = val;

next = NULL;

}

};

class LinkedList {

public:

node\* head;

LinkedList() {

head = NULL;

}

// Function to insert a new node at the end of the linked list

void insert(int val) {

node\* nn = new node(val);

if (head == NULL) {

head = nn;

return;

}

node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = nn;

}

// Function to insert a node into the list while maintaining the decreasing order

void SortedInsert(node\* newNode) {

if (head == NULL || head->data < newNode->data) {

// If the list is empty or the new node should be inserted at the beginning

newNode->next = head;

head = newNode;

}

else {

// Find the position to insert the new node

node\* current = head;

while (current->next != NULL && current->next->data > newNode->data) {

current = current->next;

}

newNode->next = current->next;

current->next = newNode;

}

}

// Function to display the elements of the linked list

void display() {

node\* temp = head;

while (temp != NULL) {

cout << temp->data << " -> ";

temp = temp->next;

}

cout << "NULL" << endl;

}

};

int main() {

LinkedList list1;

// Insert elements into list1 sorted in decreasing order

list1.insert(5);

list1.insert(3);

list1.insert(1);

cout << "Original List 1: ";

list1.display();

// Create a new node and insert it into the sorted position

node\* nn = new node(4);

list1.SortedInsert(nn);

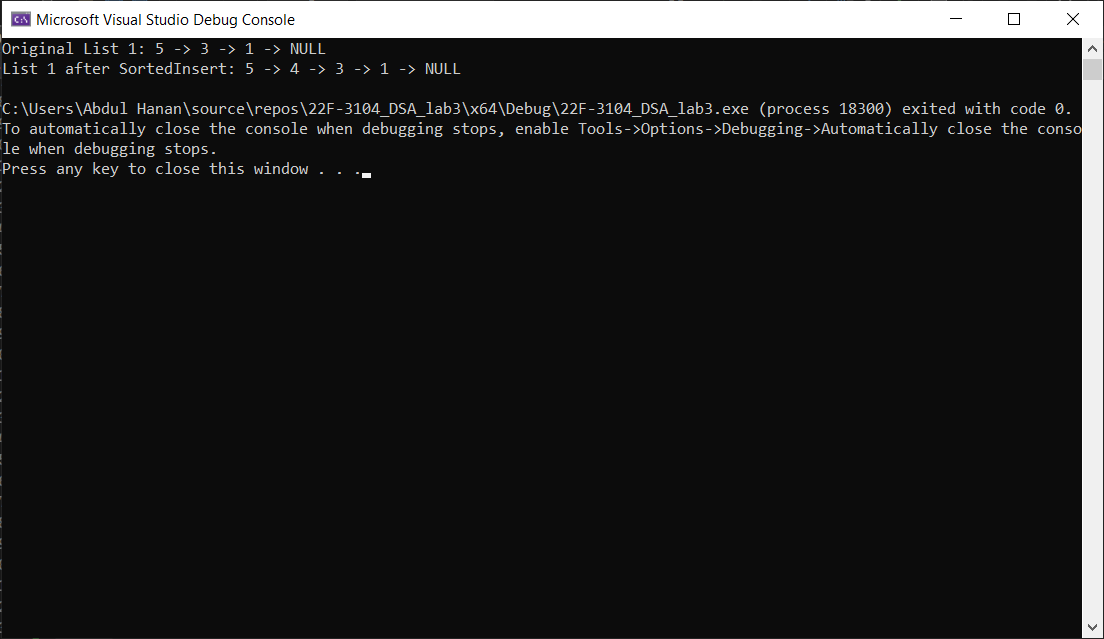
cout << "List 1 after SortedInsert: ";

list1.display();

return 0;

}

**Screen Shot:**

****