Q1:

#include <iostream>

#include <vector>

using namespace std;

// Defining the rows and columns of

// vector of vectors

#define ROW 4

#define COL 5

int main()

{

// Initializing the vector of vectors

vector<vector<int> > vec;

// Elements to insert in column

int num = 10;

// Inserting elements into vector

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

// Vector to store column elements

vector<int> v1;

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

v1.push\_back(num);

num += 5;

}

// Pushing back above 1D vector

// to create the 2D vector

vec.push\_back(v1);

}

// Displaying the 2D vector

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

for (int j = 0; j < vec[i].size(); j++)

cout << vec[i][j] << " ";

cout << endl;

}

return 0;

}

10 15 20 25 30

35 40 45 50 55

60 65 70 75 80

85 90 95 100 105

Q2:

#include <iostream>

#include<stack>

using namespace std;

int main()

{

stack<string> stack;

//The list of flower names are: Rose, Lily, Tulip, Orchid, Carnation

stack.push("Rose");

stack.push("Lily");

stack.push("Tulip");

stack.push("Orchid");

stack.push("Carnation");

while (!stack.empty()) {

cout<< stack.top()<<" ";

stack.pop();

}

return 0;

}

Carnation Orchid Tulip Lily Rose %

Q3:

#include <iostream>

#include <stack>

#include <string>

#include <sstream>

using namespace std;

bool isOperator(const string& input);

void performOp(const string& input, stack<double>& calcStack);

int main()

{

stack<double> calcStack;

string input;

while(true)

{

cout << ": ";

cin >> input;

double num;

if(istringstream(input) >> num)

{

calcStack.push(num);

}

else if (isOperator(input))

{

performOp(input, calcStack);

}

else if(input == "q")

{

return 0;

}

else

{

cout << "Invalid input" << endl;

}

}

}

bool isOperator(const string& input)

{

string ops[] = {"-", "+", "\*", "/"};

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

{

if(input == ops[i])

{

return true;

}

}

return false;

}

void performOp(const string& input, stack<double>& calcStack)

{

double lVal, rVal, result;

rVal = calcStack.top();

calcStack.pop();

lVal = calcStack.top();

calcStack.pop();

if(input == "-")

{

result = lVal - rVal;

}

else if (input == "+")

{

result = lVal + rVal;

}

else if (input == "\*")

{

result = lVal \* rVal;

}

else

{

result = lVal / rVal;

}

cout << result << endl;

calcStack.push(result);

}

: 10

: 2

: /

5

: -1

: \*

-5

: 2.2

: +

-2.8

: 4

: 5

: 6

: +

11

: +

15

: q

Q4:

1. **Singly Linked List**

* **Insert**
  + **Insert in the beginning**

Struct node \*newNode //Allocate memory for new node

NewNode = malloc(size(of(struct node)); // store data

newNode-> data = size();

newNode-> next = head; //change next of new node to point to head

head = newNode; // change head to point to recently created node

* + **Insert in the end**

Struct node \*newNode //Allocate memory for new node

NewNode = malloc(size(of(struct node)); // store data

newNode-> data = size();

newNode-> next = head; //change next of new node to point to head

struct node \*temp = head; //traverse to last node

while(temp->next != null)

temp = temp->next;

temp ->next = newNode; //change next of last node to recently created node

* + **Insert in the middle**

Struct node \*newNode //Allocate memory for new node

NewNode = malloc(size(of(struct node)); // store data

newNode-> data = size();

struct node \*temp = head;

for(int I = size() – 2; i< position; i++) //Traverse to node just before the required position of new node

if(temp -> next != Null)

temp = temp -> next

newNode->next = temp->next // change next pointers to include new node in between

temp->next = newNode

* **Delete**
  + **Delete from beginning**

Head = head -> next //Point head to the second node

* + **Delete from end**

Struct node\* temp = head //Traverse to second last element

While(temp->next-> next !=null) //Change its next pointer to null

Temp = temp->next

Temp->next = Null

* + **Delete from middle**

For (int I = 2; i< position; i++) //Traverse to element before the element to be deleted

If (temp -> next != Null)

Temp = temp->next

Temp->next = temp->next->next // Change next pointers to exclude the node from the chain

* **Search**

Bool searchNode(struct Node\*\* head\_ref, int key) //search a node

Struct Node\* current = \*head\_ref //make head as the current node

While (current != Null)

If (current -> data == key) return true

Current = current->next

Return false

* **Traversing**

Struct node \*temp = head //When temp is null, we have reached the end of the linked list so we get out of the while loop

While(temp != Null)

Printf(temp->data)

Temp = temp->next

1. **Double Linked List**

* **Insert**
  + **Insert in the beginning**

Void insertFront(struct Node\*\* head, int data) //insert node in the front

Struct Node\* newNode = new Node // allocate memory for newNode

newNode->data = data; // Assgin data to newNode

newNode->new = (\*head)

newNode->prev = Null

if ((\*head) != Null)

(\*head)->prev = newNode

(\*head) = newNode

* + **Insert between two nodes**

Void insertAfter(struct Node\* prev\_node, int data) // insert a node after a specific node

If (prev\_node == Null) // check if previous node is null

Return

Struct Node\* newNode = new Node // allocate memory for newNode

newNode->data = data // assign data to newNode

newNode->next = prev\_node->next // set next of newNode to next of prev node

prev\_node->next = newNode // set next of prev node to newNode

newNode->prev = prev\_node // set prev of newNode to the previous node

if (newNode->next != Null)

newNode->next->prev = newNode // set prev of newNode’s next to newNode

* + **Inset in the end**

Void insertEnd(struct Node\*\* head, int data) // insert a newNode in the end of list

Struct Node\* newNode = new Node // allocate memory for node

newNode->data =data // assign data to newNode

newNode->next = Null //assign null to next of newNode

struct Node\* temp = \*head // store the head node temporarily

if (\*head == Null) // if the linked list is empty, make the newNode as head node

newNode -> prev = Null

\*head = newNode

Return

While (temp->next != null) // if the linked list is not empty, traverse to the end of the linked list

Temp = temp->next

Temp-> next = newNode //point the next of the last node to newNode

newNode->prev = temp // assign prev of newNode to temp

* **Delete**
  + **Delete the first node of Doubly Linked List**

If (\*head == del\_node)

\*head = del\_node->next

If (del\_node->prev != null)

Del\_node->prev->next = del\_node->next

Free(del)

* + **Delete of the inner node**

If (del\_node->next != null)

Del\_node->next->prev = del\_node->prev

If (del\_node->prev != null)

Del\_node->prev->next = del\_node->next

* + **Delete the last node of doubly linked list**

If (del\_node->prev != null)

Del\_node->prev->next = del\_node->next

* **Search**

Node\* temp = head //create a temp node pointing to head

Int found = 0 //create two variables: found – to track

Int I = 0 // search, index – to track current index

// if the temp node is not null check the node value with searchValue, if found update variables and break the loop, else continue searching till temp node is not full

If (temp!= null)

While (temp != null)

I++

If (temp->data == searchValue)

Found++

Break

Temp = temp->next

if (found == 1)

cout <<serachValue<< “is found at index = <<i<<

else

cout<<searchValue<< “is not found in the list.”

Else // If the temp node is null in the beginning, the list is empty

Cout<< “The list is empty”

* **Traversing**

Node\* temp = head; // create a temp node pointing to head

// if the temp node is not null continue displaying the content and move to the next node till the temp becomes null

If (temp != null)

Cout << “The list contains:”

While (temp != null)

Cout << temp->data<< “ ”

Temp = temp ->next

Cout<<endl

Else

//If the temp node is null at the start, the list is empty

Cout<<”The list is emty”