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. Single Linked List

# Append a list

ListAppend(list, newNode) {

if (list->head == null) {

list->head = newNode

list->tail = newNode

}

else {

list->tail->next = newNode

list->tail = newNode

}

}

# Prepend a list

ListPrepend(list, newNode) {

if (list->head == null) {

list->head = newNode

list->tail = newNode

}

else {

newNode->next = list->head

list->head = newNode

}

}

# InsertAfter

InsertAfter(list, curNode, newNode) {

# Insert as list's first node

if (list->head == null) {

list->head = newNode

list->tail = newNode

}

# Insert after list's tail node (similar to append)

else if (curNode == list->tail) {

list->tail->next = newNode

list->tail = newNode

}

# Insert in the middle after curNode

else {

newNode->next = curNode->next

curNode->next = newNode

}

}

# RemoveAfter

ListRemoveAfter(list, curNode) {

if (curNode is 0 && list->head is not null) {

sucNode = list->head-next

list->head = sucNode

# Remove last item

if (sucNode is null) {

list->tail = null

}

}

else if (curNode->next is not null) {

sucNode = curNode->next->next

curNode->next = sucNode

# Remove tail

if (sucNode is null) {

list->tail = curNode

}

}

}

1. Double Linked List

Node class

```

class Node

\*\*Initialize with input - value\*\*

value -> value

prev -> null

next -> null

```

Doubly linked list class

```

class DoublyLinkedList

\*\*Initialize with no input\*\*

head -> null

taill -> null

\*\*Method to set node as head\*\*

\*Input - node\*

if head is null

head -> node

tail -> node

return

(insert before node method with input head and node)

\*\*Method to set node as tail\*\*

\*Input - node\*

if tail is null

(set node as head method with input node)

(insert after node method with input tail and node)

\*\*Method to insert node before\*\*

\*Input - node before, node to insert\*

if node to insert = head and node to insert = tail

return

(remove node method with input node to insert)

node to insert.prev -> node before.prev

node to insert.next -> node before

if node before.prev is null

head -> node to insert

else

node before.prev.next -> node to insert

node before.prev -> node to insert

\*\*Method to insert node after\*\*

\*Input - node after, node to insert\*

if node to insert = head and node to insert = tail

return

(remove node method with input node to insert)

node to insert.prev -> node after

node to insert.next -> node after.next

if node after.next is null

tail -> node to insert

else

node after.next.prev -> node to insert

node after.next -> node to insert

\*\*Method to insert at position\*\*

\*Input - position, node to insert\*

if position = 1

(set as head method with input node to insert)

return

node -> head

curr\_position -> 1

while node is not null and curr\_position != position

node -> node.next

curr\_position += 1

if node is not null

(insert node before method with input node and node to insert)

else

(set as tail method with input node to insert)

\*\*Method to remove a node with value\*\*

\*Input - value\*

node -> head

while node is not null

node to remove -> node

node -> node.next

if node to remove.value = value

(remove node method with input node to remove)

\*\*Method to remove a node\*\*

\*Input - node to remove\*

if node to remove = head

head -> head.next

if node to remove = tail

tail -> tail.prev

(remove node bonds method with input node to remove)

\*\*Method to check if linked list contains a node with value\*\*

\*Input - value\*

node -> head

while node is not null

if node.value = value

return True

node -> node.next

return False

or

node -> head

while node is not null and node.value != value

node -> node.next

return node is not null

\*\*Method to remove node bonds\*\*

\*Input - node to remove\*

if node to remove.prev is not null

node to remove.prev.next -> node.next

if node to remove.next is not null

node to remove.next.prev -> node.prev

node to remove.prev -> null

node to remove.next -> null