# **Reverse a Queue**

#include <iostream>

#include <queue>

using namespace std;

void reverseQueue(queue<int>& q) {

if (q.empty()) return;

int front = q.front();

q.pop();

reverseQueue(q);

q.push(front);

}

int main() {

queue<int> q;

q.push(5);

q.push(10);

q.push(15);

q.push(20);

q.push(25);

reverseQueue(q);

while (!q.empty()) {

cout << q.front() << " ";

q.pop();

}

return 0;

}

# **Implement Queue using Stacks**

# 

# **Method 1 (Costly Enqueue Operation => Enqueue in O(n) and Dequeue in O(1))**

#include <iostream>

#include <stack>

using namespace std;

class myQueue {

stack<int> s1, s2;

public:

void enqueue(int x) {

while (!s1.empty()) {

s2.push(s1.top());

s1.pop();

}

s1.push(x);

while (!s2.empty()) {

s1.push(s2.top());

s2.pop();

}

}

void dequeue() {

if (s1.empty()) {

return;

}

s1.pop();

}

int front() {

if (s1.empty()) {

return -1;

}

return s1.top();

}

int size() {

return s1.size();

}

};

int main() {

myQueue q;

q.enqueue(1);

q.enqueue(2);

q.enqueue(3);

cout << "Front: " << q.front() << '\n';

cout << "Size: " << q.size() << '\n';

q.dequeue();

cout << "Front: " << q.front() << '\n';

cout << "Size: " << q.size() << '\n';

return 0;

}

### **Method 3. Implementing Queue Using Stacks and Recursion**

#include <iostream>

#include <stack>

using namespace std;

class myQueue {

private:

stack<int> s;

public:

void enqueue(int x) {

s.push(x);

}

void dequeue() {

if (s.empty()) {

cout << "Queue Underflow\n";

return;

}

int x = s.top();

s.pop();

if (s.empty())

return;

dequeue();

s.push(x);

return;

}

int front() {

if (s.empty()) {

cout << "Queue is empty\n";

return -1;

}

int x = s.top();

s.pop();

if (s.empty()) {

s.push(x);

return x;

}

int item = front();

s.push(x);

return item;

}

int size() {

return s.size();

}

};

int main() {

myQueue q;

q.enqueue(1);

q.enqueue(2);

q.enqueue(3);

cout << "Front: " << q.front() << endl;

cout << "Size: " << q.size() << endl;

q.dequeue();

cout << "Front: " << q.front() << endl;

cout << "Size: " << q.size() << endl;

return 0;

}

# **Reverse the first K elements of a Queue using Recursion**

#include <bits/stdc++.h>

using namespace std;

void moveKToEnd(queue<int>& q, int k) {

if (k == 0) return;

int e = q.front();

q.pop();

moveKToEnd(q, k - 1);

q.push(e);

}

queue<int> reverseFirstK(queue<int> q, int k) {

moveKToEnd(q, k);

int s = q.size() - k;

while (s-- > 0) {

int x = q.front();

q.pop();

q.push(x);

}

return q;

}

int main() {

queue<int> queue;

queue.push(1);

queue.push(2);

queue.push(3);

queue.push(4);

queue.push(5);

int k = 3;

queue = reverseFirstK(queue, k);

while (!queue.empty()) {

cout << queue.front() << " ";

queue.pop();

}

return 0;

}

# **Rearrange a linked list in to alternate first and last element**

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

int data;

Node \*next;

Node(int x) {

data = x;

next = nullptr;

}

};

void reorderList(Node\* head) {

if (head == nullptr)

return;

deque<Node\*> dq;

Node\* curr = head;

while (curr) {

dq.push\_back(curr);

curr = curr->next;

}

Node\* dummy = new Node(0);

Node\* tail = dummy;

bool fromFront = true;

while (!dq.empty()) {

if (fromFront) {

tail->next = dq.front();

dq.pop\_front();

}

else {

tail->next = dq.back();

dq.pop\_back();

}

tail = tail->next;

fromFront = !fromFront;

}

tail->next = nullptr;

head = dummy->next;

}

int main() {

Node \*head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(4);

head->next->next->next->next = new Node(5);

reorderList(head);

Node\* curr = head;

while (curr != nullptr) {

cout << curr->data << " ";

curr = curr->next;

}

cout << endl;

return 0;

}

# **Finding the Length of a Linked List**

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int new\_data) {

data = new\_data;

next = nullptr;

}

};

int countNodes(Node\* head) {

int count = 0;

Node\* curr = head;

while (curr != nullptr) {

count++;

curr = curr->next;

}

return count;

}

int main() {

Node\* head = new Node(1);

head->next = new Node(3);

head->next->next = new Node(1);

head->next->next->next = new Node(2);

head->next->next->next->next = new Node(1);

cout << "Count of nodes is " << countNodes(head);

return 0;

}

### **Find the Length of a Linked List (Using Recursion):**

### Recursive C++ program to find length

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int new\_data) {

data = new\_data;

next = nullptr;

}

};

int countNodes(Node\* head) {

if (head == NULL) {

return 0;

}

return 1 + countNodes(head->next);

}

int main() {

Node\* head = new Node(1);

head->next = new Node(3);

head->next->next = new Node(1);

head->next->next->next = new Node(2);

head->next->next->next->next = new Node(1);

cout << "Count of nodes is " << countNodes(head);

return 0;

}

# **Write a function to get Nth node in a Linked List**

#include <bits/stdc++.h>

using namespace std;

struct Node {

int data;

Node\* next;

Node(int x) {

data = x;

next = NULL;

}

};

int GetNth(Node\* head, int index) {

if (head == NULL)

return -1;

if (index == 1)

return head->data;

return GetNth(head->next, index - 1);

}

int main() {

Node\* head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(4);

head->next->next->next->next = new Node(5);

cout << "Element at index 3 is " << GetNth(head, 3) << endl;

return 0;

}

# **Delete Node by Position**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int data) {

this->data = data;

this->next = nullptr;

}

};

Node\* deleteNode(Node\* head, int position) {

Node\* temp = head;

if (position == 1) {

head = temp->next;

delete temp;

return head;

}

Node\* prev = nullptr;

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

prev = temp;

temp = temp->next;

}

prev->next = temp->next;

delete temp;

return head;

}

void printList(Node\* head) {

while (head != nullptr) {

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

head = head->next;

}

cout << "nullptr" << endl;

}

int main() {

Node\* head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(4);

int position = 3;

head = deleteNode(head, position);

printList(head);

return 0;

}

# **Nth node from the end of a Linked List**

#include <bits/stdc++.h>

using namespace std;

struct Node {

int data;

Node\* next;

Node(int new\_data) {

data = new\_data;

next = nullptr;

}

};

int findNthFromLast(Node\* head, int N) {

int len = 0, i;

Node\* temp = head;

while (temp != NULL) {

temp = temp->next;

len++;

}

if (len < N)

return -1;

temp = head;

for (i = 1; i < len - N + 1; i++)

temp = temp->next;

return temp->data;

}

int main() {

Node\* head = new Node(35);

head->next = new Node(15);

head->next->next = new Node(4);

head->next->next->next = new Node(20);

cout << findNthFromLast(head, 4);

return 0;

}

# **Detect Cycle in Linked List**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int x) {

this->data = x;

this->next = nullptr;

}

};

bool detectLoop(Node\* head) {

Node \*slow = head, \*fast = head;

while (slow && fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

if (slow == fast) {

return true;

}

}

return false;

}

int main() {

Node\* head = new Node(1);

head->next = new Node(3);

head->next->next = new Node(4);

head->next->next->next = head->next;

if (detectLoop(head))

cout << "true";

else

cout << "false";

return 0;

}

# **Split a Circular Linked List into two halves**

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

int data;

Node \*next;

Node (int new\_value){

data = new\_value;

next = nullptr;

}

};

pair<Node\*, Node\*> splitList(Node \*head) {

Node \*slow = head;

Node \*fast = head;

if(head == nullptr)

return {nullptr, nullptr};

while(fast->next != head &&

fast->next->next != head) {

fast = fast->next->next;

slow = slow->next;

}

if(fast->next->next == head)

fast = fast->next;

Node\* head1 = head;

Node\* head2 = slow->next;

fast->next = slow->next;

slow->next = head;

return {head1, head2};

}

void printList(Node \*head) {

Node \*curr = head;

if(head != nullptr) {

do {

cout << curr->data << " ";

curr = curr->next;

} while(curr != head);

cout << endl;

}

}

int main() {

Node \*head = new Node(1);

Node \*head1 = nullptr;

Node \*head2 = nullptr;

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(4);

head->next->next->next->next = head;

pair<Node\*, Node\*> result = splitList(head);

head1 = result.first;

head2 = result.second;

printList(head1);

printList(head2);

return 0;

}

# **Merge two sorted linked lists**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class Node {

public:

int data;

Node \*next;

Node(int x) {

data = x;

next = nullptr;

}

};

Node \*sortedMerge(Node \*head1, Node \*head2) {

vector<int> arr;

while (head1 != nullptr) {

arr.push\_back(head1->data);

head1 = head1->next;

}

while (head2 != nullptr) {

arr.push\_back(head2->data);

head2 = head2->next;

}

sort(arr.begin(), arr.end());

Node \*dummy = new Node(-1);

Node \*curr = dummy;

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

curr->next = new Node(arr[i]);

curr = curr->next;

}

return dummy->next;

}

void printList(Node \*curr) {

while (curr != nullptr) {

cout << curr->data;

if (curr->next != nullptr)

cout << " -> ";

curr = curr->next;

}

cout << endl;

}

int main() {

Node \*head1 = new Node(5);

head1->next = new Node(10);

head1->next->next = new Node(15);

head1->next->next->next = new Node(40);

Node \*head2 = new Node(2);

head2->next = new Node(3);

head2->next->next = new Node(20);

Node \*res = sortedMerge(head1, head2);

printList(res);

return 0;

}

# **Rotate a Linked List**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node \*next;

Node(int new\_data) {

data = new\_data;

next = nullptr;

}

};

Node \*rotate(Node \*head, int k) {

if (k == 0 || head == nullptr)

return head;

Node \*curr = head;

int len = 1;

while (curr->next != nullptr) {

curr = curr->next;

len += 1;

}

k %= len;

if (k == 0)

return head;

curr->next = head;

curr = head;

for (int i = 1; i < k; i++)

curr = curr->next;

head = curr->next;

curr->next = nullptr;

return head;

}

void printList(Node \*node) {

while (node != nullptr) {

cout << node->data << " ";

if(node->next != NULL){

cout << "-> ";

}

node = node->next;

}

cout << endl;

}

int main() {

Node \*head = new Node(10);

head->next = new Node(20);

head->next->next = new Node(30);

head->next->next->next = new Node(40);

head = rotate(head, 6);

printList(head);

return 0;

}

**Reverse a Doubly Linked List**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node \*next;

Node \*prev;

Node(int new\_data) {

data = new\_data;

next = NULL;

prev = NULL;

}

};

Node \*reverse(Node \*head) {

// If the list is empty or has only one node,

// return the head as is

if (head == nullptr || head->next == nullptr)

return head;

Node \*prevNode = NULL;

Node \*currNode = head;

// Traverse the list and reverse the links

while (currNode != nullptr) {

// Swap the next and prev pointers

prevNode = currNode->prev;

currNode->prev = currNode->next;

currNode->next = prevNode;

// Move to the next node in the original list

// (which is now previous due to reversal)

currNode = currNode->prev;

}

return prevNode->prev;

}

void printList(Node \*node) {

while (node != nullptr) {

cout << node->data;

if(node->next != nullptr){

cout << " <-> ";

}

node = node->next;

}

cout << endl;

}

int main() {

Node \*head = new Node(1);

head->next = new Node(2);

head->next->prev = head;

head->next->next = new Node(3);

head->next->next->prev = head->next;

head = reverse(head);

printList(head);

return 0;

}

# **Insert a Node at a specific position in Doubly Linked List**

// C++ Program to insert a node at a given position

#include <bits/stdc++.h>

using namespace std;

struct Node {

int data;

Node \*next, \*prev;

Node(int new\_data) {

data = new\_data;

next = prev = nullptr;

}

};

// Function to insert a new node at a given position

Node \*insertAtPosition(Node \*head, int pos, int new\_data) {

// Create a new node

Node \*new\_node = new Node(new\_data);

// Insertion at the beginning

if (pos == 1) {

new\_node->next = head;

// If the linked list is not empty, set the prev of head to new node

if (head != NULL)

head->prev = new\_node;

// Set the new node as the head of linked list

head = new\_node;

return head;

}

Node \*curr = head;

// Traverse the list to find the node before the

// insertion point

for (int i = 1; i < pos - 1 && curr != NULL; ++i) {

curr = curr->next;

}

// If the position is out of bounds

if (curr == NULL) {

cout << "Position is out of bounds." << endl;

delete new\_node;

return head;

}

// Set the prev of new node to curr

new\_node->prev = curr;

// Set the new of new node to next of curr

new\_node->next = curr->next;

// Update the next of current node to new node

curr->next = new\_node;

// If the new node is not the last node, update prev of next node to new node

if (new\_node->next != NULL)

new\_node->next->prev = new\_node;

// Return the head of the doubly linked list

return head;

}

void printList(Node \*head) {

Node \*curr = head;

while (curr != NULL) {

cout << curr->data << " ";

curr = curr->next;

}

cout << endl;

}

int main() {

// Create a harcoded doubly linked list:

// 1 <-> 2 <-> 4

Node \*head = new Node(1);

head->next = new Node(2);

head->next->prev = head;

head->next->next = new Node(4);

head->next->next->prev = head->next;

// Print the original list

cout << "Original Linked List: ";

printList(head);

// Insert new node with data 3 at position 3

cout << "Inserting Node with data 3 at position 3: ";

int data = 3;

int pos = 3;

head = insertAtPosition(head, pos, data);

// Print the updated list

printList(head);

return 0;

}

# **Count nodes in Circular linked list**

// C++ program to count number of nodes

// in a circular linked list.

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node(int x) {

data = x;

next = nullptr;

}

};

// Function to count nodes in a given Circular

// linked list

int countNodes(Node\* head) {

if (head == nullptr) return 0;

Node\* curr = head;

int result = 0;

do {

curr = curr->next;

result++;

} while (curr != head);

return result;

}

int main() {

// Create list: 1->2->3->4->5--->1

Node\* head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

head->next->next->next = new Node(4);

head->next->next->next->next = new Node(5);

head->next->next->next->next->next = head;

cout << countNodes(head);

return 0;

}

# **Insertion at specific position in circular linked list**

#include <iostream>

using namespace std;

struct Node{

int data;

Node \*next;

Node(int value){

data = value;

next = nullptr;

}

};

// Function to insert a node at a specific position in a circular linked list

Node \*insertAtPosition(Node \*last, int data, int pos){

if (last == nullptr){

// If the list is empty

if (pos != 1){

cout << "Invalid position!" << endl;

return last;

}

// Create a new node and make it point to itself

Node \*newNode = new Node(data);

last = newNode;

last->next = last;

return last;

}

// Create a new node with the given data

Node \*newNode = new Node(data);

// curr will point to head initially

Node \*curr = last->next;

if (pos == 1){

// Insert at the beginning

newNode->next = curr;

last->next = newNode;

return last;

}

// Traverse the list to find the insertion point

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

curr = curr->next;

// If position is out of bounds

if (curr == last->next){

cout << "Invalid position!" << endl;

return last;

}

}

// Insert the new node at the desired position

newNode->next = curr->next;

curr->next = newNode;

// Update last if the new node is inserted at the end

if (curr == last) last = newNode;

return last;

}

void printList(Node \*last){

if (last == NULL) return;

Node \*head = last->next;

while (true){

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

head = head->next;

if (head == last->next) break;

}

cout << endl;

}

int main(){

// Create circular linked list: 2, 3, 4

Node \*first = new Node(2);

first->next = new Node(3);

first->next->next = new Node(4);

Node \*last = first->next->next;

last->next = first;

cout << "Original list: ";

printList(last);

// Insert elements at specific positions

int data = 5, pos = 2;

last = insertAtPosition(last, data, pos);

cout << "List after insertions: ";

printList(last);

return 0;

}

# **Next Greater Element in Array**

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

vector<int> nextLargerElement(vector<int> &arr) {

int n = arr.size();

vector<int> res(n, -1);

stack<int> stk;

for (int i = n - 1; i >= 0; i--) {

// Pop elements from the stack that are less

// than or equal to the current element

while (!stk.empty() && stk.top() <= arr[i]) {

stk.pop();

}

// If the stack is not empty, the top element

// is the next greater element

if (!stk.empty()) {

res[i] = stk.top();

}

// Push the current element onto the stack

stk.push(arr[i]);

}

return res;

}

int main() {

vector<int> arr = {6, 8, 0, 1, 3};

vector<int> res = nextLargerElement(arr);

for (int x : res) {

cout << x << " ";

}

return 0;

}

# **Previous Smaller Element**

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

vector<int> prevSmaller(vector<int>& arr) {

int n = arr.size();

vector<int> result(n, -1);

stack<int> st;

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

// pop elements from stack until a smaller

// element is found or stack becomes empty

while (!st.empty() && st.top() >= arr[i]) {

st.pop();

}

// if stack is not empty, top is nearest smaller

if (!st.empty()) {

result[i] = st.top();

}

// push current element to stack

st.push(arr[i]);

}

return result;

}

int main() {

vector<int> arr = {1, 5, 0, 3, 4, 5};

vector<int> ans = prevSmaller(arr);

for (int x : ans) cout << x << " ";

return 0;

}

# **Next Greater Element in Array**

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

vector<int> nextLargerElement(vector<int> &arr) {

int n = arr.size();

vector<int> res(n, -1);

stack<int> stk;

for (int i = n - 1; i >= 0; i--) {

// Pop elements from the stack that are less

// than or equal to the current element

while (!stk.empty() && stk.top() <= arr[i]) {

stk.pop();

}

// If the stack is not empty, the top element

// is the next greater element

if (!stk.empty()) {

res[i] = stk.top();

}

// Push the current element onto the stack

stk.push(arr[i]);

}

return res;

}

int main() {

vector<int> arr = {6, 8, 0, 1, 3};

vector<int> res = nextLargerElement(arr);

for (int x : res) {

cout << x << " ";

}

return 0;

}

# **Infix to Postfix Expression**

#include <iostream>

#include <stack>

using namespace std;

int prec(char c) {

if (c == '^')

return 3;

else if (c == '/' || c == '\*')

return 2;

else if (c == '+' || c == '-')

return 1;

else

return -1;

}

bool isRightAssociative(char c) {

return c == '^';

}

string infixToPostfix(string &s) {

stack<char> st;

string res;

for (int i = 0; i < s.length(); i++) {

char c = s[i];

if ((c >= 'a' && c <= 'z') ||

(c >= 'A' && c <= 'Z') ||

(c >= '0' && c <= '9'))

res += c;

else if (c == '(')

st.push('(');

else if (c == ')') {

while (!st.empty() && st.top() != '(') {

res += st.top();

st.pop();

}

st.pop();

}

else {

while (!st.empty() && st.top() != '(' &&

(prec(st.top()) > prec(c) ||

(prec(st.top()) == prec(c) && !isRightAssociative(c)))) {

res += st.top();

st.pop();

}

st.push(c);

}

}

while (!st.empty()) {

res += st.top();

st.pop();

}

return res;

}

int main() {

string exp = "a\*(b+c)/d";

cout << infixToPostfix(exp);

return 0;

}

# **Infix To Prefix Notation**

#include <iostream>

#include <stack>

#include <algorithm>

#include <cctype>

using namespace std;

int precedence(char c) {

if (c == '^') return 3;

else if (c == '\*' || c == '/') return 2;

else if (c == '+' || c == '-') return 1;

else return -1;

}

bool isOperator(char c) {

return (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^');

}

bool isRightAssociative(char op) {

return (op == '^');

}

string infixToPrefix(string s) {

reverse(s.begin(), s.end());

for (int i = 0; i < (int)s.length(); i++) {

if (s[i] == '(') s[i] = ')';

else if (s[i] == ')') s[i] = '(';

}

stack<char> st;

string result = "";

for (int i = 0; i < (int)s.length(); i++) {

char c = s[i];

if (isalnum(c)) {

result += c;

}

else if (c == '(') {

st.push(c);

}

else if (c == ')') {

while (!st.empty() && st.top() != '(') {

result += st.top();

st.pop();

}

if (!st.empty()) st.pop();

}

else if (isOperator(c)) {

while (!st.empty() && st.top() != '(' &&

((precedence(st.top()) > precedence(c)) ||

(precedence(st.top()) == precedence(c) && isRightAssociative(c)))) {

result += st.top();

st.pop();

}

st.push(c);

}

}

while (!st.empty()) {

result += st.top();

st.pop();

}

reverse(result.begin(), result.end());

return result;

}

int main() {

string s = "a\*(b+c)/d";

cout << infixToPrefix(s);

return 0;

}

# **Implement Stack using Queues**

#include <iostream>

#include <queue>

using namespace std;

class myStack {

queue<int> q1, q2;

public:

// insert element

void push(int x) {

q1.push(x);

}

void pop() {

if (q1.empty())

return;

while (q1.size() != 1) {

q2.push(q1.front());

q1.pop();

}

q1.pop();

swap(q1, q2);

}

int top() {

if (q1.empty())

return -1;

while (q1.size() != 1) {

q2.push(q1.front());

q1.pop();

}

int temp = q1.front();

q1.pop();

q2.push(temp);

swap(q1, q2);

return temp;

}

int size() {

return q1.size();

}

};

int main() {

myStack st;

st.push(1);

st.push(2);

st.push(3);

cout << st.top() << endl;

st.pop();

cout << st.top() << endl;

st.pop();

cout << st.top() << endl;

cout << st.size() << endl;

return 0;

}