







1. #include <iostream>

#include <queue>

#include <stack>

// Function to reverse a queue using a stack

void reverseQueue(std::queue<int>& q) {

std::stack<int> s;

// Dequeue all elements from the queue and push them onto the stack

while (!q.empty()) {

s.push(q.front());

q.pop();

}

// Pop all elements from the stack and enqueue them back into the queue

while (!s.empty()) {

q.push(s.top());

s.pop();

}

}

// Function to display the elements of the queue

void displayQueue(std::queue<int> q) {

while (!q.empty()) {

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

q.pop();

}

std::cout << std::endl;

}

int main() {

std::queue<int> q;

q.push(1);

q.push(2);

q.push(3);

q.push(4);

q.push(5);

std::cout << "Original Queue: ";

displayQueue(q);

reverseQueue(q);

std::cout << "Reversed Queue: ";

displayQueue(q);

return 0;

}

1. #include <iostream>

#include <stack>

#include <stdexcept>

class MaxStack {

private:

std::stack<int> mainStack;

std::stack<int> maxStack;

public:

// Pushes an element onto the stack

void push(int value) {

mainStack.push(value);

if (maxStack.empty() || value >= maxStack.top()) {

maxStack.push(value);

}

}

// Pops the top element from the stack

void pop() {

if (mainStack.empty()) {

throw std::out\_of\_range("Stack is empty");

}

if (mainStack.top() == maxStack.top()) {

maxStack.pop();

}

mainStack.pop();

}

// Returns the top element of the stack

int top() {

if (mainStack.empty()) {

throw std::out\_of\_range("Stack is empty");

}

return mainStack.top();

}

// Retrieves the maximum element in the stack

int getMax() {

if (maxStack.empty()) {

throw std::out\_of\_range("Stack is empty");

}

return maxStack.top();

}

// Checks if the stack is empty

bool empty() const {

return mainStack.empty();

}

};

int main() {

MaxStack stack;

stack.push(3);

stack.push(5);

std::cout << "Current Max: " << stack.getMax() << std::endl; // Should print 5

stack.push(7);

stack.push(19);

std::cout << "Current Max: " << stack.getMax() << std::endl; // Should print 19

stack.pop();

std::cout << "Current Max: " << stack.getMax() << std::endl; // Should print 7

stack.pop();

std::cout << "Current Max: " << stack.getMax() << std::endl; // Should print 5

stack.pop();

std::cout << "Current Max: " << stack.getMax() << std::endl; // Should print 3

return 0;

}

1. #include <iostream>

#include <stdexcept>

class CircularQueue {

private:

int\* queue;

int front;

int rear;

int capacity;

int size;

public:

CircularQueue(int capacity) {

this->capacity = capacity;

queue = new int[capacity];

front = -1;

rear = -1;

size = 0;

}

~CircularQueue() {

delete[] queue;

}

// Enqueue an element to the rear of the queue

void enqueue(int value) {

if (isFull()) {

throw std::overflow\_error("Queue is full");

}

if (isEmpty()) {

front = 0;

}

rear = (rear + 1) % capacity;

queue[rear] = value;

size++;

}

// Dequeue an element from the front of the queue

void dequeue() {

if (isEmpty()) {

throw std::underflow\_error("Queue is empty");

}

if (front == rear) {

front = -1;

rear = -1;

} else {

front = (front + 1) % capacity;

}

size--;

}

// Get the front element of the queue

int getFront() const {

if (isEmpty()) {

throw std::underflow\_error("Queue is empty");

}

return queue[front];

}

// Check if the queue is empty

bool isEmpty() const {

return size == 0;

}

// Check if the queue is full

bool isFull() const {

return size == capacity;

}

// Get the size of the queue

int getSize() const {

return size;

}

};

int main() {

CircularQueue cq(5);

cq.enqueue(1);

cq.enqueue(2);

cq.enqueue(3);

cq.enqueue(4);

cq.enqueue(5);

std::cout << "Front element: " << cq.getFront() << std::endl;

cq.dequeue();

std::cout << "Front element after dequeue: " << cq.getFront() << std::endl;

cq.enqueue(6);

std::cout << "Front element after enqueue: " << cq.getFront() << std::endl;

while (!cq.isEmpty()) {

std::cout << "Dequeue element: " << cq.getFront() << std::endl;

cq.dequeue();

}

return 0;

}

1. #include <iostream>

#include <stack>

// Function to sort a stack

void sortStack(std::stack<int>& stack) {

std::stack<int> auxStack;

while (!stack.empty()) {

// Pop the top element from the original stack

int temp = stack.top();

stack.pop();

// While the auxiliary stack is not empty and the top element of auxStack is greater than temp

while (!auxStack.empty() && auxStack.top() > temp) {

stack.push(auxStack.top());

auxStack.pop();

}

// Push the temp element onto the auxiliary stack

auxStack.push(temp);

}

// Transfer the sorted elements back to the original stack

while (!auxStack.empty()) {

stack.push(auxStack.top());

auxStack.pop();

}

}

// Function to print the elements of a stack

void printStack(std::stack<int> stack) {

while (!stack.empty()) {

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

stack.pop();

}

std::cout << std::endl;

}

int main() {

std::stack<int> stack;

stack.push(34);

stack.push(3);

stack.push(31);

stack.push(98);

stack.push(92);

stack.push(23);

std::cout << "Original Stack: ";

printStack(stack);

sortStack(stack);

std::cout << "Sorted Stack: ";

printStack(stack);

return 0;

}

1. #include <iostream>
2. #include <list>
3. int main() {
4. // Create a list
5. std::list<int> myList;
6. // Insert elements at the end
7. myList.push\_back(10);
8. myList.push\_back(20);
9. myList.push\_back(30);
10. // Insert elements at the front
11. myList.push\_front(5);
12. myList.push\_front(1);
13. // Display elements
14. std::cout << "List after push\_back and push\_front: ";
15. for (int val : myList) {
16. std::cout << val << " ";
17. }
18. std::cout << std::endl;
19. // Insert element at a specific position
20. auto it = myList.begin();
21. std::advance(it, 2);
22. myList.insert(it, 15);
23. std::cout << "List after insert: ";
24. for (int val : myList) {
25. std::cout << val << " ";
26. }
27. std::cout << std::endl;
28. // Erase element at a specific position
29. it = myList.begin();
30. std::advance(it, 3);
31. myList.erase(it);
32. std::cout << "List after erase: ";
33. for (int val : myList) {
34. std::cout << val << " ";
35. }
36. std::cout << std::endl;
37. // Remove elements by value
38. myList.remove(10);
39. std::cout << "List after remove: ";
40. for (int val : myList) {
41. std::cout << val << " ";
42. }
43. std::cout << std::endl;
44. // Remove elements based on a condition
45. myList.remove\_if([](int n) { return n < 10; });
46. std::cout << "List after remove\_if: ";
47. for (int val : myList) {
48. std::cout << val << " ";
49. }
50. std::cout << std::endl;
51. // Sorting the list
52. myList.sort();
53. std::cout << "List after sort: ";
54. for (int val : myList) {
55. std::cout << val << " ";
56. }
57. std::cout << std::endl;
58. // Reversing the list
59. myList.reverse();
60. std::cout << "List after reverse: ";
61. for (int val : myList) {
62. std::cout << val << " ";
63. }
64. std::cout << std::endl;
65. // Merging two lists
66. std::list<int> otherList = {40, 50, 60};
67. myList.merge(otherList);
68. std::cout << "List after merge: ";
69. for (int val : myList) {
70. std::cout << val << " ";
71. }
72. std::cout << std::endl;
73. // Clearing the list
74. myList.clear();
75. std::cout << "List after clear: ";
76. for (int val : myList) {
77. std::cout << val << " ";
78. }
79. std::cout << std::endl;
80. // Checking if the list is empty
81. if (myList.empty()) {
82. std::cout << "List is empty." << std::endl;
83. }
84. // Adding elements again
85. myList.push\_back(100);
86. myList.push\_back(200);
87. // Accessing front and back elements
88. std::cout << "Front element: " << myList.front() << std::endl;
89. std::cout << "Back element: " << myList.back() << std::endl;
90. return 0;
91. }