// Quang Le

// Project 3

// CS 474

// ContiguousSequence.h file

#ifndef CONTIGUOUSSEQUENCE\_H

#define CONTIGUOUSSEQUENCE\_H

class ContiguousSequence {

public:

// Default constructor

ContiguousSequence();

// Constructor with an initial element

ContiguousSequence(int value);

// Copy constructor

ContiguousSequence(const ContiguousSequence& other);

// Destructor

~ContiguousSequence();

// Assignment operator

ContiguousSequence& operator=(const ContiguousSequence& other);

// Return the current capacity of the contiguous sequence

int basicSize() const;

// Return the number of elements in the contiguous sequence

int size() const;

// Indexing operator (non-const version) with bounds checking

int& operator[](int index);

// Indexing operator (const version) with bounds checking

const int& operator[](int index) const;

// Insert an element at a specific index

ContiguousSequence& insertAt(int index, int value);

// Find the index of a specific element in the contiguous sequence

int find(int value) const;

// Remove an element at a specific index

ContiguousSequence& removeAt(int index);

// Apply a function to each element in the contiguous sequence

ContiguousSequence& iterate(int (\*fn)(int));

//optional

// getter and setter

int getStart() {

return this->start;

}

void setStart(int start){

this->start = start;

}

int getEnd() {

return this->end;

}

void setEnd(int end){

this->end = end;

}

private:

// Pointer to the dynamically allocated sequence

int\* sequence;

// Current capacity of the contiguous sequence

int capacity;

// Start index of the region storing elements

int start;

// End index of the region storing elements

int end;

// Non-public method to dynamically allocate a larger storage region

void grow();

};

#endif // CONTIGUOUSSEQUENCE\_H

// Quang Le

// Project 3

// CS 474

// ContiguousSequence.cpp file

#include "ContiguousSequence.h"

#include <iostream>

using namespace std;

// Default constructor

ContiguousSequence::ContiguousSequence() {

capacity = 8;

sequence = new int[capacity];

start = 3;

end = 2;

}

// Constructor with an initial element

ContiguousSequence::ContiguousSequence(int value) : ContiguousSequence() {

sequence[start] = value;

end++;

}

// Copy constructor

ContiguousSequence::ContiguousSequence(const ContiguousSequence& other) {

// Deep copy of the other object

capacity = other.capacity;

sequence = new int[capacity];

start = other.start;

end = other.end;

for (int i = start; i <= end; i++) {

sequence[i] = other.sequence[i];

}

}

// Destructor

ContiguousSequence::~ContiguousSequence() {

// Destructor to avoid memory leaks

delete[] sequence;

}

// Assignment operator

ContiguousSequence& ContiguousSequence::operator=(const ContiguousSequence& other) {

// Assignment operator for deep copy

if (this != &other) {

delete[] sequence;

capacity = other.capacity;

sequence = new int[capacity];

start = other.start;

end = other.end;

for (int i = start; i <= end; i++) {

sequence[i] = other.sequence[i];

}

}

return \*this;

}

// Return the current capacity of the contiguous sequence

int ContiguousSequence::basicSize() const {

return capacity;

}

// Return the number of elements in the contiguous sequence

int ContiguousSequence::size() const {

return end - start + 1;

}

// Indexing operator (non-const version) with bounds checking

int& ContiguousSequence::operator[](int index) {

if (index < 0 || index >= size()) {

throw out\_of\_range("Error out of bounds");

//cout"Error out of bounds"; // uncomment if you want to only print error

}

return sequence[start + index];

}

// Indexing operator (const version) with bounds checking

const int& ContiguousSequence::operator[](int index) const {

if (index < 0 || index >= size()) {

throw out\_of\_range("Error out of bounds");

//cout"Error out of bounds"; // uncomment if you want to only print error

}

return sequence[start + index];

}

// Insert an element at a specific index

ContiguousSequence& ContiguousSequence::insertAt(int index, int value) {

// if no element or out of bound

if (index < 0 || index > size()) {

cout << "Error out of bounds" << endl;

//throw out\_of\_range("Error out of bounds");// uncomment if you want to stop and throw error

return \*this;

}

// double memory size

if (size() == capacity) {

grow();

}

// Handle different insertion scenarios

if (index == 0) {

if (start == 0) {

// Shift elements to the right

for (int i = end; i >= start; i--) {

sequence[i + 1] = sequence[i];

}

end++;

} else {

start--;

}

} else if (index == size()) {

if (end == capacity - 1) {

// Shift elements to the left

for (int i = start; i <= end; i++) {

sequence[i - 1] = sequence[i];

}

start--;

} else {

end++;

}

} else {

if (end == capacity - 1) {

// Shift elements to the left

for (int i = start + index; i <= end; i++) {

sequence[i - 1] = sequence[i];

}

start--;

} else {

// Shift elements to the right

for (int i = end; i >= start + index; i--) {

sequence[i + 1] = sequence[i];

}

end++;

}

}

sequence[start + index] = value;

return \*this;

}

// Find the index of a specific element in the contiguous sequence

int ContiguousSequence::find(int value) const {

for (int i = start; i <= end; i++) {

if (sequence[i] == value) {

return i - start;

}

}

return -1;

}

// Remove an element at a specific index

ContiguousSequence& ContiguousSequence::removeAt(int index) {

if (index < 0 || index >= size()) {

cout << "Index out of bounds" << endl;

return \*this;

}

// Shift elements to fill the gap caused by removal

for (int i = start + index; i < end; i++) {

sequence[i] = sequence[i + 1];

}

// Adjust start and end pointers

if (index == 0) {

start++;

} else if (index == size() - 1) {

end--;

}

return \*this;

}

// Apply a function to each element in the contiguous sequence

ContiguousSequence& ContiguousSequence::iterate(int (\*fn)(int)) {

for (int i = start; i <= end; i++) {

sequence[i] = fn(sequence[i]);

}

return \*this;

}

// Non-public method to dynamically allocate a larger storage region

void ContiguousSequence::grow() {

int newCapacity = capacity \* 2;

int\* newSequence = new int[newCapacity];

// Copy elements to the new region

for (int i = start; i <= end; i++) {

newSequence[i] = sequence[i];

}

// Deallocate old storage and update pointers

delete[] sequence;

sequence = newSequence;

capacity = newCapacity;

}

// Quang Le

// Project 3

// CS 474

// main.cpp file

#include "ContiguousSequence.h"

#include <iostream>

using namespace std;

void printListAndSize(const ContiguousSequence& cs) {

cout << "Elements in the list:";

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

cout << " | " << cs[i];

}

if (cs.size() != 0) {

cout << " |";

}

cout << endl;

cout << "Entire allocated space: " << cs.basicSize() << endl;

}

void getStartAndEnd () {

ContiguousSequence cs;

cout << "\n+Test getter and setter:\n";

cout << "Start is:" << cs.getStart()<< endl;

cout << "End is:" << cs.getEnd()<< endl;

cs.setStart(4);

cs.setEnd(3);

cout << "Modify Start to 4:" << cs.getStart() << endl;

cout << "Modify End to 3:" << cs.getEnd() << endl;

}

void testDefaultConstructor() {

ContiguousSequence cs;

cout << "\n+Test Default Constructor:\n";

printListAndSize(cs);

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

cout << "Basic Size: " << cs.basicSize() << endl;

}

void testParameterizedConstructor() {

ContiguousSequence cs(42);

cout << "\n+Test Parameterized Constructor:\n";

printListAndSize(cs);

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

cout << "Basic Size: " << cs.basicSize() << endl;

}

void testCopyConstructor() {

ContiguousSequence original;

original.insertAt(0, 5);

original.insertAt(1, 10);

original.insertAt(2, 15);

ContiguousSequence copy(original);

cout << "\n+Test Copy Constructor:\n";

printListAndSize(copy);

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

cout << "Basic Size: " << copy.basicSize() << endl;

}

void testAssignmentOperator() {

ContiguousSequence original;

original.insertAt(0, 5);

original.insertAt(1, 10);

original.insertAt(2, 15);

ContiguousSequence assigned;

assigned = original;

cout << "\n+Test Assignment Operator:\n";

printListAndSize(assigned);

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

cout << "Basic Size: " << assigned.basicSize() << endl;

}

void testInsertAndRemove() {

ContiguousSequence cs;

cs.insertAt(0, 5);

cs.insertAt(1, 10);

cs.insertAt(2, 15);

cs.insertAt(1, 20);

cout << "\n+Test Insert and Remove:\n";

printListAndSize(cs);

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

cout << "Basic Size: " << cs.basicSize() << endl;

cs.removeAt(1);

cout << "After removeAt(1):\n";

printListAndSize(cs);

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

cout << "Basic Size: " << cs.basicSize() << endl;

}

void testIterate() {

ContiguousSequence cs;

cs.insertAt(0, 5);

cs.insertAt(1, 10);

cs.insertAt(2, 15);

cout << "\n+Test Iterate:\n";

cout << "Before\n";

printListAndSize(cs);

cout << "\nAfter\n";

cs.iterate([](int value) { return value \* 2; });

printListAndSize(cs);

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

cout << "Basic Size: " << cs.basicSize() << endl;

}

void testFind() {

ContiguousSequence cs;

cs.insertAt(0, 5);

cs.insertAt(1, 10);

cs.insertAt(2, 15);

std::cout << "\n+Test Find:\n";

printListAndSize(cs);

int index = cs.find(10);

cout << "Index of value 10: " << index << endl;

index = cs.find(25);

cout << "Index of value 25: " << index << endl;

}

void testBound() {

ContiguousSequence cs;

cs.insertAt(0, 5);

cout << "\n+Test Throw error at index 20:\n";

cout << "Throw error" << cs[20] << endl;

}

void testMemorysize() {

ContiguousSequence original;

original.insertAt(0, 5);

original.insertAt(0, 10);

original.insertAt(0, 15);

original.insertAt(0, 20);

original.insertAt(0, 25);

original.insertAt(0, 30);

original.insertAt(1, 35);

original.insertAt(0, 90);

original.insertAt(3, 45);

ContiguousSequence assigned;

assigned = original;

cout << "\n+Test Double Size:\n";

printListAndSize(assigned);

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

cout << "Basic Size: " << assigned.basicSize() << endl;

}

int main() {

// Test Default Constructor

testDefaultConstructor();

// Test Parameterized Constructor

testParameterizedConstructor();

// Test Copy Constructor

testCopyConstructor();

// Test Assignment Operator

testAssignmentOperator();

// Test Insert and Remove

testInsertAndRemove();

// Test Iterate

testIterate();

// Test Double Memorysize

testMemorysize();

//Test Find

testFind();

//Test getter and setter

getStartAndEnd ();

// test out bound case

testBound();

return 0;

}