**Software Testing Project**

**CMP461**

**Team “Sunday”**

**Team Members:**

* Radwa Samy Khattab 1 – 18
* Salma Ibrahim 1 – 26
* Mariam Hesham 2 – 21
* Nehal Abdelkader 2 – 28
* Nourhan Gamal 2 – 31

**Delivered to**

Dr. Ahmed Sobeih

Eng. Ali El-Sedeek

[I. Software Under Test (SUT) 3](#_Toc59149908)

[II. Coverages and Approach in Testing 3](#_Toc59149909)

[III. User-defined Map 4](#_Toc59149910)

[IV. User-define List 6](#_Toc59149911)

[V. User-Defined Linked Stack 9](#_Toc59149912)

[VI. User-Defined Linked Queue 10](#_Toc59149913)

[VII. Graph 12](#_Toc59149914)

# Software Under Test (SUT)

We choose various user implemented data structures to implement in C++. For example: pair, map, linked list, stack, queue, binary trees, graphs…etc.

In addition, we choose multiple sorting and searching algorithms and implemented them too.

All code is written in C++.

Then we choose Google Test (G-Test) tool to test these codes and their results.

Starting III. we would show each Data Structure/Algorithm we implemented and the code used for testing using GTest framework, and the results of these tests. Mostly all would be correct as we implemented the original code in a clean clear way and were almost sure it had no errors.

The expected output of each test case is shows in the testing code.

For example:

EXPECT\_EQ(queue.top(), 0);

Here it shows that the expected output of “queue.top()” should be equal to 0.

The actual output is showed after running the test code. If all passed that means the actual output is same as the expected output.

GTest documentation can be shown here: <https://github.com/nordlow/gtest-tutorial>

But generally, name of functions can be easy to understand their behavior, as in the previous example, “EXPECT\_EQ” means EXPECT EQUAL.

# Coverages and Approach in Testing

We mainly focused on call coverage in data structure testing, which was mentioned in one of the first lectures. When testing a class, we have to make sure to test all the member functions of that class. And in each member function, which is called at least once, we make sure of satisfying Edge Coverage, which also subsumes Node Coverage.

When needed, we try out different input for same function. Those values can be chosen like we learnt in Input Space Partition. However, it won’t be needed in all functions.

In Sorting and Searching algorithms, Edge Coverage is also guaranteed, which subsumes Node Coverage.

Each Data Structure or algorithm are shown in more details in following sections.

# User-defined Map

## User-defined Pair Class Header Code:

class Pair

{

int key;

int value;

public:

Pair(int a, int b);

Pair();

void setPair(int a, int b);

void setValue(int v);

int getKey() const;

int getValue() const;

bool operator == (const Pair & p);

};

## User-defined Map Class Header Code

#include "Pair.h"

#define MAXSIZE 100

class Map

{

Pair pairs[MAXSIZE];

int count;

public:

Map();

bool addPair(int key, int value);

int getValue(int key) const;

int getCount() const;

bool UpdateValue(const int &key, const int&value);

bool deleteKey(int key);

};

## Map Test Code

#include <gtest/gtest.h>

#include “Map.h”

TEST(MAP, CONSTRUCTOR\_AND\_GET\_COUNT) {

Map m = Map();

EXPECT\_EQ(m.getCount(), 0);

}

TEST(MAP, ADD\_AND\_GET\_Pair) {

Map m = Map();

EXPECT\_TRUE(m.addPair(1, -4));

EXPECT\_EQ(m.getCount(), 1);

EXPECT\_EQ(m.getValue(1), -4); // return value if key exist

EXPECT\_EQ(m.getValue(2), NULL); // return NULL if not exist

EXPECT\_TRUE(m.addPair(1.4, 0));

// if key is float, it should be floored

EXPECT\_EQ(m.getValue(1.4), -4); // same as key: 1

}

TEST(MAP, MAX\_CAPACITY) {

Map m = Map();

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

EXPECT\_TRUE(m.addPair(i, i));

EXPECT\_EQ(m.getCount(), 100);

EXPECT\_FALSE(m.addPair(100, 100));

EXPECT\_EQ(m.getCount(), 100);

}

TEST(MAP, UPDATE\_VALUE) {

Map m = Map();

EXPECT\_TRUE(m.addPair(5, 30));

EXPECT\_EQ(m.getValue(5), 30);

EXPECT\_TRUE(m.UpdateValue(5, 50)); //updating successfully

EXPECT\_EQ(m.getValue(5), 50); // m[5] should equal 50 now

EXPECT\_FALSE(m.UpdateValue(3, 4)); // key 3 doesn't exist

}

TEST(MAP, DELETE\_Key) {

Map m = Map();

EXPECT\_TRUE(m.addPair(3, 30));

EXPECT\_TRUE(m.addPair(5, 50));

EXPECT\_EQ(m.getCount(), 2);

EXPECT\_TRUE(m.deleteKey(3));

EXPECT\_EQ(m.getCount(), 1);

EXPECT\_FALSE(m.deleteKey(7));

EXPECT\_EQ(m.getCount(), 1);

}

## Map Tests Results

# User-define List

You can set any value at any specific index in a List, as long as it’s below MAX\_SIZE

## Class Header Code

#define MAXSIZE 50

class List

{

int list[MAXSIZE];

int size;

char \* nameOfList;

public:

List();

List(int maxSize, char \* name);

bool set(int index, const int &d);

void setName(char \* name);

int getIndex(const int &d) const;

void sort();

int getSize() const;

char \* getName() const;

int& operator [](int index);

List& operator = (List l);

bool operator == (List l) const ;

};

## Test Code

TEST(LIST, EMPTY\_CONSTRUCTOR) {

List l = List();

EXPECT\_EQ(l.getSize(), MAXSIZE);

EXPECT\_STREQ(l.getName(), "");

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

EXPECT\_EQ(l[i], 0);

}

TEST(LIST, INIT\_CONSTRUCTOR) {

List l = List(3, "smallList");

EXPECT\_EQ(l.getSize(), 3);

EXPECT\_STREQ(l.getName(), "smallList");

for (int i = 0; i < 3; i++)

EXPECT\_EQ(l[i], 0);

}

TEST(LIST, CHANGE\_NAME) {

List l = List(3, "smallList");

EXPECT\_STREQ(l.getName(), "smallList");

l.setName("newName");

EXPECT\_STREQ(l.getName(), "newName");

}

TEST(LIST, SET\_VALUES) {

List l = List(10, "myList");

EXPECT\_TRUE(l.set(4, 5));

EXPECT\_FALSE(l.set(-4, 20)); // negative index

EXPECT\_FALSE(l.set(13, 5)); // over capacity

}

TEST(LIST, OPERATOR\_OVERLOADING) {

List l = List(20, "operatorOverloaded");

l[3] = 30;

EXPECT\_EQ(l[3], 30);

EXPECT\_TRUE(l[3] == 30);

}

TEST(LIST, EQUAL\_LIST) {

List l = List(6, "testList");

List l1 = List(4, "list1");

List l2 = List(7, "list2");

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

l1[i] = i;

for (int i = 0; i < 7; i++)

l2[i] = 10 \* i;

l = l1;

EXPECT\_EQ(l.getSize(), 4);

EXPECT\_STREQ(l.getName(), "list1");

EXPECT\_EQ(l[0], 0);

EXPECT\_EQ(l[2], 2);

EXPECT\_EQ(l[3], 3);

l = l2;

EXPECT\_EQ(l.getSize(), 7);

EXPECT\_STREQ(l.getName(), "list2");

EXPECT\_EQ(l[0], 0);

EXPECT\_EQ(l[3], 30);

EXPECT\_EQ(l[6], 60);

}

TEST(LIST, SORT) {

List l = List(5, "unsortedList");

l[0] = 4;

l[1] = 2;

l[2] = 0;

l[3] = -4;

l[4] = 1;

l.sort();

EXPECT\_EQ(l[0], -4);

EXPECT\_EQ(l[1], 0);

EXPECT\_EQ(l[2], 1);

EXPECT\_EQ(l[3], 2);

EXPECT\_EQ(l[4], 4);

}

TEST(LIST, GET\_INDEX) {

List l = List(9, "testList");

for (int i = 0; i < 9; i++)

l[i] = i \* 10;

EXPECT\_EQ(l.getIndex(30), 3);

EXPECT\_EQ(l.getIndex(0), 0);

EXPECT\_EQ(l.getIndex(40), 4);

EXPECT\_EQ(l.getIndex(80), 8);

EXPECT\_EQ(l.getIndex(18), -1);

}

## List Tests Results

# User-Defined Linked Stack

## Node Class Header Code

class Node

{

int item;

Node \* next;

public:

Node();

Node(const int & newItem);

Node(const int& newItem, Node \*nextNodePtr);

void setItem(const int & newItem);

void setNextNode(Node \*nextNodePtr);

int getItem() const ;

Node \* getNextNode() const ;

};

## LinkedStack Class Header Code

class LinkedStack

{

Node\* top; // Index to top of stack

public:

LinkedStack(); // Default constructor

LinkedStack(LinkedStack &aStack); // Copy constructor

~LinkedStack(); // Destructor

bool isEmpty() const ;

void push( const int & newEntry );

bool pop();

int peek() const ;

}; // end ArrayStack

## Linked Stack Test Code

TEST(STACK, CONSTRUCTOR\_AND\_IsEQUAL) {

LinkedStack s = LinkedStack();

EXPECT\_TRUE(s.isEmpty());

}

TEST(STACK, PUSH\_AND\_PEEK) {

LinkedStack s = LinkedStack();

s.push(1);

EXPECT\_EQ(s.peek(), 1);

s.push(3);

EXPECT\_EQ(s.peek(), 3);

}

TEST(STACK, POP) {

LinkedStack s = LinkedStack();

s.push(1);

s.push(3);

EXPECT\_TRUE(s.pop()); // bec. not empty

EXPECT\_FALSE(s.isEmpty());

EXPECT\_EQ(s.peek(), 1);

s.pop();

EXPECT\_TRUE(s.isEmpty());

EXPECT\_FALSE(s.pop()); // bec. empty

}

TEST(STACK, COPY\_CONSTRUCTOR) {

LinkedStack s = LinkedStack();

s.push(1);

LinkedStack s2 = LinkedStack(s);

EXPECT\_EQ(s2.peek(), 1);

EXPECT\_EQ(s.peek(), 1); // to make sure it didn't change

}

## Linked Stack Tests Results

# User-Defined Linked Queue

## Linked Queue Header Code

#include "Node.h"

class LinkedQueue {

// The queue is implemented as a chain of linked nodes that has

// two external pointers, a head pointer for the front of the queue

// and a tail pointer for the back of the queue.

Node\* backPtr;

Node\* frontPtr;

public :

LinkedQueue();

LinkedQueue(LinkedQueue & aQueue);

~LinkedQueue();

bool isEmpty() const ;

bool enqueue(int newEntry);

bool dequeue();

int peekFront() const;

}; // end Queue

## Linked Queue Test Code

TEST(QUEUE, CONSTRUCTOR\_AND\_IsEQUAL) {

LinkedQueue q = LinkedQueue();

EXPECT\_TRUE(q.isEmpty());

}

TEST(QUEUE, ENQUEUE\_AND\_PEEK\_FRONT) {

LinkedQueue q = LinkedQueue();

q.enqueue(1);

EXPECT\_EQ(q.peekFront(), 1);

q.enqueue(3);

EXPECT\_EQ(q.peekFront(), 1);

}

TEST(QUEUE, DEQUEUE) {

LinkedQueue q = LinkedQueue();

EXPECT\_TRUE(q.isEmpty());

q.enqueue(1);

q.enqueue(3);

EXPECT\_TRUE(q.dequeue()); // bec. not empty

EXPECT\_FALSE(q.isEmpty());

EXPECT\_EQ(q.peekFront(), 3);

q.dequeue();

EXPECT\_TRUE(q.isEmpty());

EXPECT\_FALSE(q.dequeue()); // bec. empty

}

TEST(QUEUE, COPY\_CONSTRUCTOR) {

LinkedQueue q = LinkedQueue();

q.enqueue(3);

q.enqueue(2);

q.enqueue(1);

LinkedQueue s2 = LinkedQueue(q);

EXPECT\_EQ(s2.peekFront(), 3);

EXPECT\_EQ(s2.peekFront(), 3);

}

## Linked Queue Tests Results

# Graph

## Edge Class Header Code

class Edge

{

int destination;

Edge \* NextEdge; //ptr to next Edge

long weight; //weight of Edge

public:

//constructors

Edge(int d);

Edge(int d, long w);

Edge(int d, Edge \* nxtEdge);

Edge(int d, Edge \* nxtEdge, long w);

//setters

void setDestination(int d);

void setNext(Edge \* nxt);

void setWeight(long w);

//getters

int getDestination() const ;

Edge \* getNext() const ;

long getWeight() const ;

};

## Vertex Class Header Code

#include "Edge.h"

class Vertex

{

int data;

Vertex \* nextVertex;

int inDegree, outDegree;

Edge \* firstEdge;

int processed;

public:

//constructors

Vertex(int d);

//setters

void setData(int d);

void setNextVertex(Vertex \* v);

void setFirstEdge( Edge \* Edge );

//getters

int getData() const ;

Vertex \* getNextVertex() const ;

Edge \* getFirstEdge() const ;

int getProcessedCount() const ;

//another functions

void incProcessed();

void resetProcessed();

bool isConnected(int v2) const ;

void addEdge(int v, int weight);

void removeEdge(int d);

int getEdgeWeight(int destination) const ;

bool HasEdges() const ;

//destructor

~Vertex();

};

## Graph Header Code

#include "Vertex.h"

#include <string>

using namespace std;

class Graph

{

Vertex \* headVertex;

Vertex \* tailVertex;

int numVertices;

int numEdges;

//utility functions

Vertex \* & getVertexPtr(int v) const ;

Vertex \* getPrevVertex(int data) const ;

public:

//constructors

Graph();

//main operations

bool add(int vertex1, int vertex2, int edgeWeight) ;

bool remove(int vertex1, int vertex2);

void clear() ;

bool isExist(int vertex) const ;

bool isEmpty() const ;

bool isConnected(int v1, int v2) const ;

//getters

int getNumVertices() const ;

int getNumEdges() const ;

int getEdgeWeight(int start, int end) const ;

//Traversals

string depthFirstTraversal(int start);

string breadthFirstTraversal(int start);

};

## Graph Tests Code

#include "pch.h"

#include "Graph.h"

TEST(GRAPH, EMPTY\_CONSTRUCTOR\_CHECKS\_EMPTY) {

Graph g = Graph();

EXPECT\_TRUE(g.isEmpty());

EXPECT\_EQ(g.getNumEdges(), 0);

EXPECT\_EQ(g.getNumVertices(), 0);

}

TEST(GRAPH, IS\_EXIST) {

Graph g = Graph();

EXPECT\_TRUE(g.add(1, 2, 10));

EXPECT\_TRUE(g.isExist(1));

EXPECT\_FALSE(g.isExist(4));

}

TEST(GRAPH, IS\_CONNECTED) {

Graph g = Graph();

EXPECT\_TRUE(g.add(1, 2, 10));

EXPECT\_TRUE(g.isConnected(1, 2));

EXPECT\_FALSE(g.isConnected(3, 2));

}

TEST(GRAPH, ADD\_VERTEX) {

Graph g = Graph();

EXPECT\_TRUE(g.add(1, 2, 10)); // graph is empty

EXPECT\_TRUE(g.add(1, 3, 20)); // v1 exists only

EXPECT\_TRUE(g.add(4, 3, 30)); // v2 exists only

EXPECT\_TRUE(g.add(3, 2, 40)); // both exists but not connected

EXPECT\_TRUE(g.add(5, 6, 50)); // neither exist & not empty

EXPECT\_FALSE(g.add(1, 2, 10)); // bec. already exits

}

TEST(GRAPH, CHECK\_SIZE) {

Graph g = Graph();

EXPECT\_TRUE(g.add(1, 2, 10));

EXPECT\_TRUE(g.add(1, 3, 20));

EXPECT\_FALSE(g.isEmpty());

EXPECT\_EQ(g.getNumEdges(), 2);

EXPECT\_EQ(g.getNumVertices(), 3);

}

TEST(GRAPH, CLEAR) {

Graph g = Graph();

EXPECT\_TRUE(g.add(1, 2, 10));

EXPECT\_TRUE(g.add(1, 3, 20));

EXPECT\_FALSE(g.isEmpty());

EXPECT\_EQ(g.getNumEdges(), 2);

EXPECT\_EQ(g.getNumVertices(), 3);

g.clear();

EXPECT\_TRUE(g.isEmpty());

EXPECT\_EQ(g.getNumEdges(), 0);

EXPECT\_EQ(g.getNumVertices(), 0);

}

TEST(GRAPH, REMOVE) {

Graph g = Graph();

EXPECT\_TRUE(g.add(1, 2, 10));

EXPECT\_TRUE(g.add(1, 3, 20));

EXPECT\_EQ(g.getNumEdges(), 2);

EXPECT\_TRUE(g.isConnected(1, 2));

EXPECT\_EQ(g.getNumVertices(), 3);

EXPECT\_FALSE(g.remove(4, 5)); // nodes don't exist

EXPECT\_FALSE(g.remove(3, 2)); // not connected

EXPECT\_TRUE(g.remove(1, 2));

EXPECT\_EQ(g.getNumEdges(), 1);

EXPECT\_FALSE(g.isConnected(1, 2));

EXPECT\_EQ(g.getNumVertices(), 2); // vertex 2 is removed

}

TEST(GRAPH, EDGE\_WEIGHT) {

Graph g = Graph();

EXPECT\_TRUE(g.add(1, 4, 10));

EXPECT\_EQ(g.getEdgeWeight(1, 4), 10);

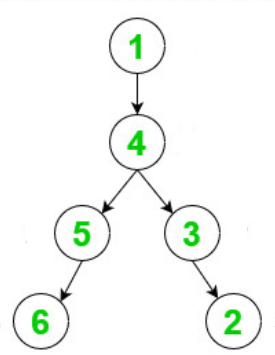
EXPECT\_EQ(g.getEdgeWeight(1, 3), -1); // don't exist

}

TEST(GRAPH, DFS) {

Graph g = Graph();

EXPECT\_TRUE(g.add(1, 4, 10));

 EXPECT\_TRUE(g.add(4, 5, 20));

EXPECT\_TRUE(g.add(4, 3, 10));

EXPECT\_TRUE(g.add(5, 6, 20));

EXPECT\_TRUE(g.add(3, 2, 10));

string dfs = g.depthFirstTraversal(1);

EXPECT\_EQ(dfs, "143256");

}

TEST(GRAPH, BFS) {

Graph g = Graph();

EXPECT\_TRUE(g.add(1, 4, 10));

EXPECT\_TRUE(g.add(4, 5, 20));

EXPECT\_TRUE(g.add(4, 3, 10));

EXPECT\_TRUE(g.add(5, 6, 20));

EXPECT\_TRUE(g.add(3, 2, 10));

string dfs = g.breadthFirstTraversal(1);

EXPECT\_EQ(dfs, "145362");

}

## Graph Tests Results