**Task-1:**

To facilitate the reuse of this tree in other company projects we can do the following

First we need to separate the code into respective headers and cpp files

So we create A Employee.h

And make the actual code for the database independent of the Employee struct hence we will use templates instead.

The code can also be optimized to use smart pointers such as unique pointer

// EmployeeInfo.h

#pragma once

struct EmployeeInfo {

int salary;

int age;

int emplNumber;

int sin; // Search by social insurance number

};

// AVLNode.h

#pragma once

template <typename T>

struct AVLNode {

T data;

AVLNode\* left = nullptr;

AVLNode\* right = nullptr;

int height = 0;

AVLNode(const T& data) : data(data) {}

};

// AVLTree.h

#pragma once

#include "AVLNode.h"

#include <memory>

#include <functional> // For std::function

template <typename T>

class AVLTree {

public:

AVLTree() : root(nullptr) {}

~AVLTree() { clear(root); }

void insert(const T& data);

void remove(int key); // The key can be a part of T data to identify the node

void display(std::ostream& out) const;

AVLNode<T>\* Find(int key) const;

private:

std::unique\_ptr<AVLNode<T>> root;

// Recursive utility functions to be implemented in the .cpp file

AVLNode<T>\* insert(AVLNode<T>\* node, const T& data);

AVLNode<T>\* remove(AVLNode<T>\*& node, int key);

AVLNode<T>\* findMin(AVLNode<T>\* node) const;

AVLNode<T>\* findMax(AVLNode<T>\* node) const;

AVLNode<T>\* singleRightRotate(AVLNode<T>\*& node);

AVLNode<T>\* singleLeftRotate(AVLNode<T>\*& node);

AVLNode<T>\* doubleLeftRotate(AVLNode<T>\*& node);

AVLNode<T>\* doubleRightRotate(AVLNode<T>\*& node);

void inorder(AVLNode<T>\* node, std::ostream& out) const;

void clear(AVLNode<T>\*& node);

int height(AVLNode<T>\* node) const;

int getBalance(AVLNode<T>\* node) const;

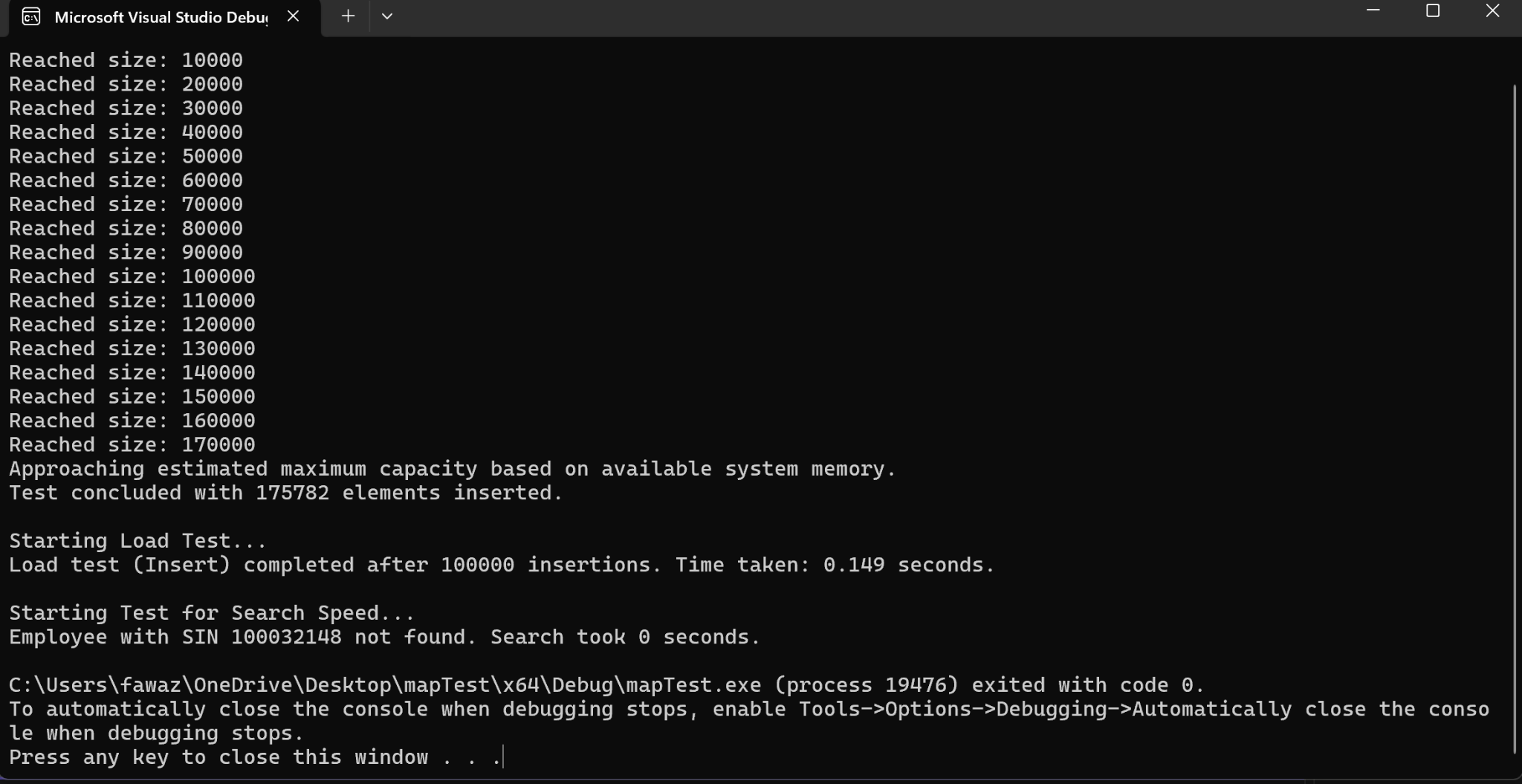
};

// Implementations for template functions would be in the cpp file.

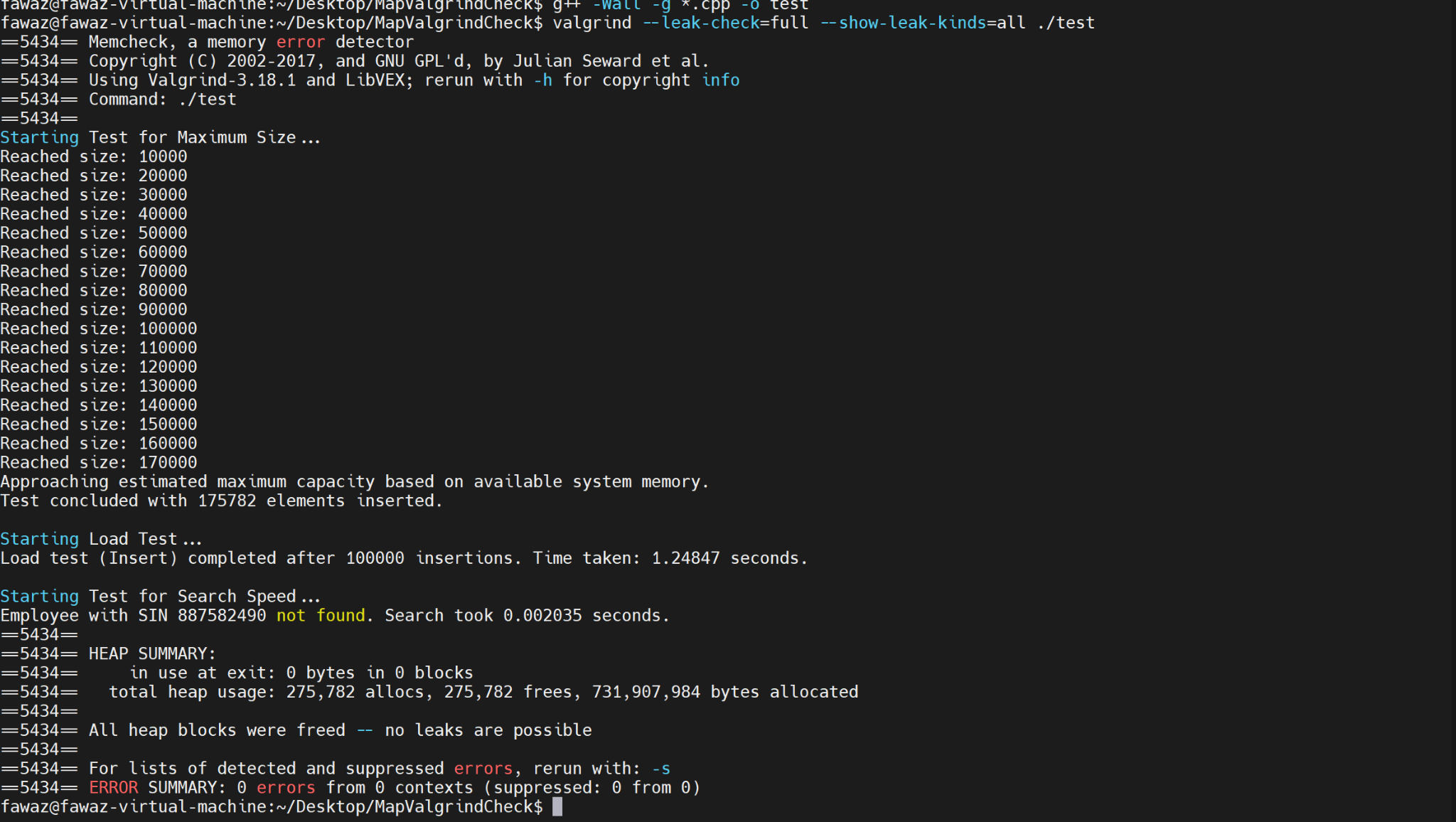
# TEST CASES OUTPUT AND VALGRIND MEMORY TEST

## 

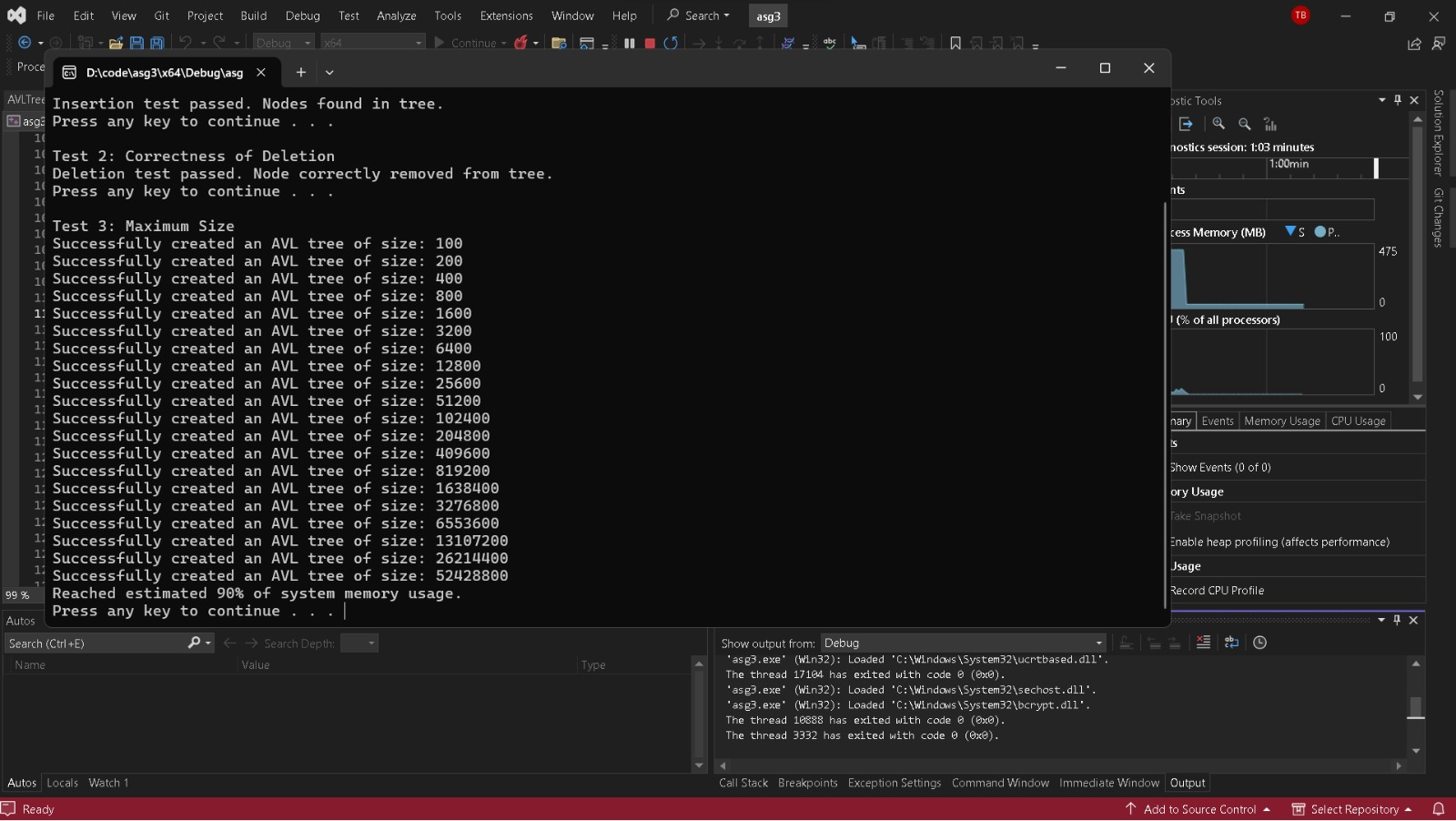
## std::map ,Windows Test

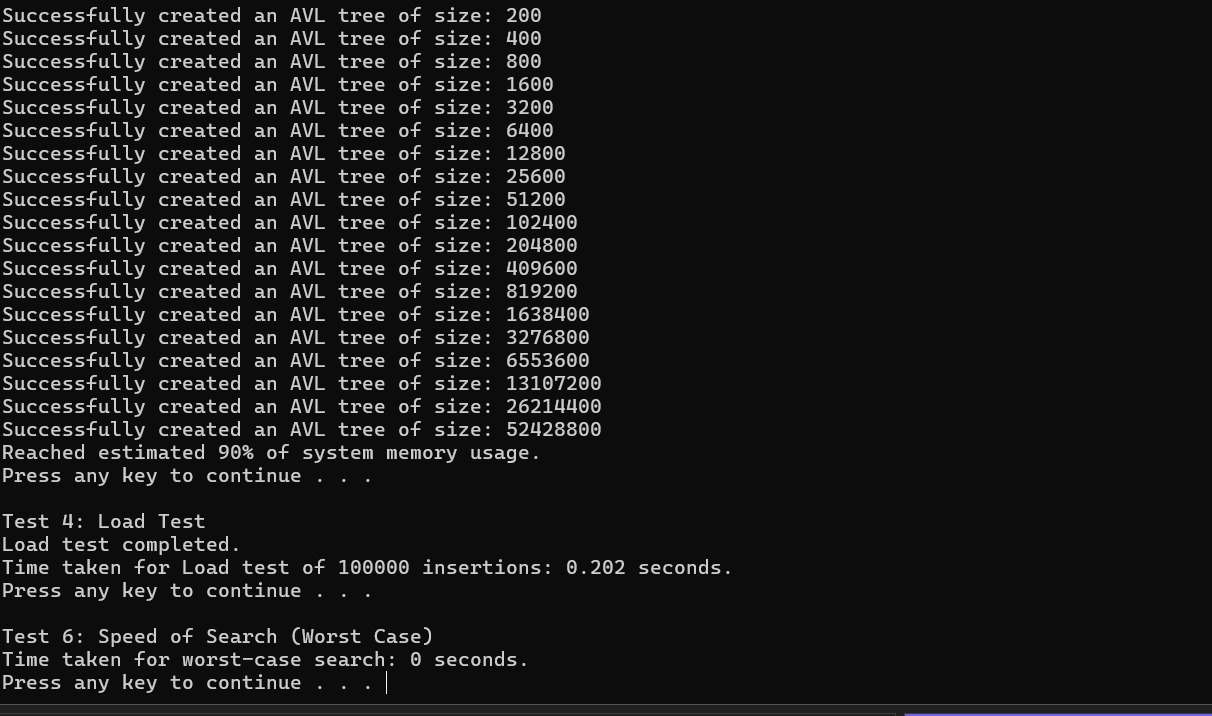


## std::map ,Linux Test with Valgrind

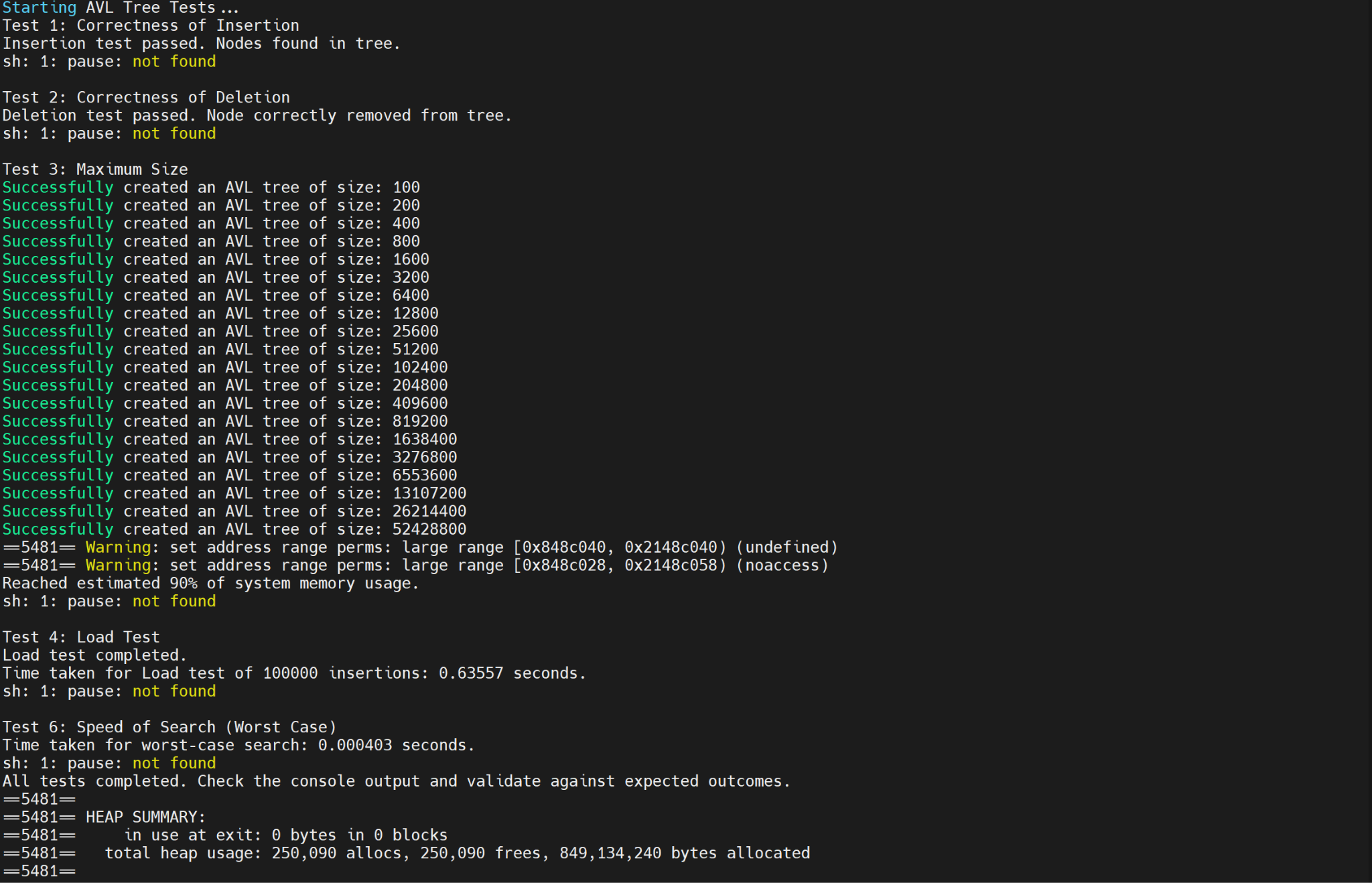


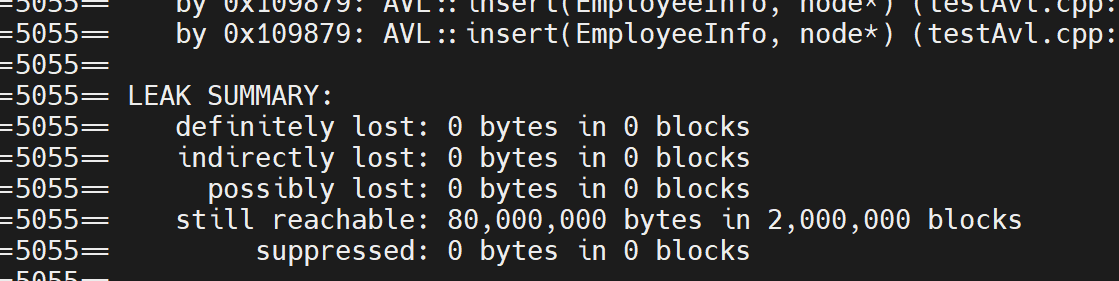
## AVL TREE:Windows





## AVL Linux TEST With Valgrind





## Test5:Memory Test

As we can see Valgrind gives us information on any possible memory leaks when we run the code.

For std::map the summary tells us that there are no possible memory leaks in our std::map code.

For AVL tree on the other hand for the code provided in the assignment question,the leak summary shows

Still reachable:80,000,000 bytes

This means the program is likely exiting without freeing all the memory it allocated, but the pointers to that memory are still available, so it could have been freed before the program exited.Still reachable" is not as dangerous as "definitely lost" memory, because it means that our program could have freed the memory but we didn't.It's more of a soft warning. This is often seen in programs where the cleanup phase is skipped, as the OS will reclaim the memory on program exit anyway.To tackle this we would create a destructor which deallocates all nodes.

**Note:Test 1,2,3,4,6 in the code file submitted**

# Questions

Ans1)The other tests I feel could be used in this case are:

**Balance Property Tests:**

After insertion and deletion we should write a test that verifies the balancing logic of the AVL tree so that the height of any two subtrees does not differ by more than one.

**Performance compared to the competitive product**

We could compare our database with some other competitive market software and see which is better

**Defect bash:**

We could bring people from other departments for them to try to break the system as they would perform random operations such as insertion, deletion etc.

**Concurrency Test**

As the database would normally be simanutanelosy-accessed,test it to make sure simultaneous inserts, deletes, and finds don't lead to race conditions or deadlocks.

—----------------------------------------------------------------------------------------------------------------------------

**Note:Since we are working with a restricted amount of memory here our device's memory we fixed it to 8GB and to avoid crashing the program we stopped the maximum size tests when we reached around 90% of our memory size.**

**—----------------------------------------------------------------------------------------------------------------------------**

Ans2)After running the tests we found out that:-

i)Both std::map and AVL correctly inserts and deletes hence test 1 and 2 are the same.

**For Windows:**

|  | **maximum size** | **load** | **memory leak** | **speed of search** |
| --- | --- | --- | --- | --- |
| **AVL** | **52428800** | **0.2 sec** | **none** | **0 sec** |
| **std::map** | **175782** | **0.149 sec** | **none** | **0 sec** |

**For Linux**

|  | **Maximum size** | **Load** | **memory leak** | **Speed of Search** |
| --- | --- | --- | --- | --- |
| **AVL** | **52428800** | **0.63 sec** | **none** | **0.000404 sec** |
| **std::map** | **175782** | **1.248 sec** | **none** | **0.002 sec** |

**Maximum Size:**

AVL Tree: It seems to support a significantly larger size compared to std::map. This might be because std::map in most implementations is a Red-Black tree, which may have additional overhead per node.

**Load**:

AVL Tree: Slightly slower load times compared to std::map in windows, faster in linux. This could be due to different balancing strategies and allocation overheads.

std::map: Generally has faster load times, potentially due to more efficient memory allocation and balancing operations.

**Memory Leak:**

Both data structures show no memory leaks.Except for warning in the AVL code.

**Speed of Search:**

AVL Tree: Very fast search times

std::map: Also very fast, but slightly slower in the Linux environment compared to the AVL tree.

ii)From the tests in both the Operating System we can choose one over the other based on our systems requirement.

If maximum size and search performance are our concern then AVL tree is preferable.

On the other hand for general-purpose use where ease of use, maintenance, and standard library support are important, std::map is recommended due to its robust implementation and integration with the C++ STL.

iii)Based on our memory tests the maximum size we could afford with the restricted environment we set i.e 8GB of memory is 52428800 for AVL tree and 175782 for std::map.This is when we use up 90-95% of our memory and anymore of it would cause out system to crash hence I would set the threshold to those values so that the system doesn't crash due to memory mismanagement.

Ans3)After we separate the files into cpp and .h and change the code into the template structure yes the code will be reusable for any other project as template can handle different data types and can be used across various projects with different types of element

Ans4)For this Assignment the test code we wrote are centered around the employee database but those are documented enough to give an idea on what the tests are doing.With minor changes in the parameters the best code can be easily replicated.But if the goal is to use the test code as is for a different project the answer would be no.The way it could be avoided was by modularization and by using parameters for test cases, such as sizes, ranges, and types of operations instead of hardcoding it to be specific to employee database.Again,as mentioned through some minor changes it could be replicated easily though.