|  |
| --- |
| Written Report for Programming Project 1 |
| COMP1921 |
| Luke Hutton |
| SC17LJTH |

Contents

[Task 1 2](#_Toc507344845)

[New function 2](#_Toc507344846)

[Function description 2](#_Toc507344847)

[Tests 2](#_Toc507344848)

[Task 2 6](#_Toc507344849)

[New function 6](#_Toc507344850)

[Function description 6](#_Toc507344851)

[Tests 6](#_Toc507344852)

[Task 3 11](#_Toc507344853)

[Node size memory estimate 11](#_Toc507344854)

[Full tree memory estimates 11](#_Toc507344855)

[Estimated memory use for full trees of 5,6,7,8,9,10 levels 12](#_Toc507344856)

[Actual memory use for full trees of 5,6,7,8,9,10 levels 12](#_Toc507344857)

[If you would like to limit the overall memory use of the application to 20Mb what maximum level should you choose? 12](#_Toc507344858)

[Max tree level implementation 12](#_Toc507344859)

[Testing max tree level 13](#_Toc507344860)

[Task 4 16](#_Toc507344861)

[New functions 16](#_Toc507344862)

[Test 1 16](#_Toc507344863)

[Test 2 17](#_Toc507344864)

[Task 5: Reflection 18](#_Toc507344865)

# Task 1

### New function

// Destroy whole tree starting from head

void destroyTree**(**Node **\***head**)** **{**

destroyNode**(**head**);**

**}**

// Destroy node and all sub nodes recursively

void destroyNode**(**Node **\***node**)** **{**

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

**if** **(**node**->**child**[**i**]** !**=** **NULL)**

destroyNode**(**node**->**child**[**i**]);**

**}**

free**(**node**);**

**}**

### Function description

This function destroys the tree and frees up memory allocated to each of the nodes. Calling destroy tree will call the destroy node function passing the head of the tree as the argument. The destroy node function will check whether the node passed as an argument has any children, and for each child recursively run the function. Then once this process is complete it will free the node from memory and move onto the next node in the call stack.

### Tests

#### Test 1

For the first test I used this function to create a full tree at Level 2:

void task1**()** **{**

Node **\***head**;**

head **=** makeNode**(**0.0**,** 0.0**,** 0**);**

//Tree structure

makeChildren**(**head**);**

makeChildren**(**head**->**child**[**0**]);**

makeChildren**(**head**->**child**[**1**]);**

makeChildren**(**head**->**child**[**2**]);**

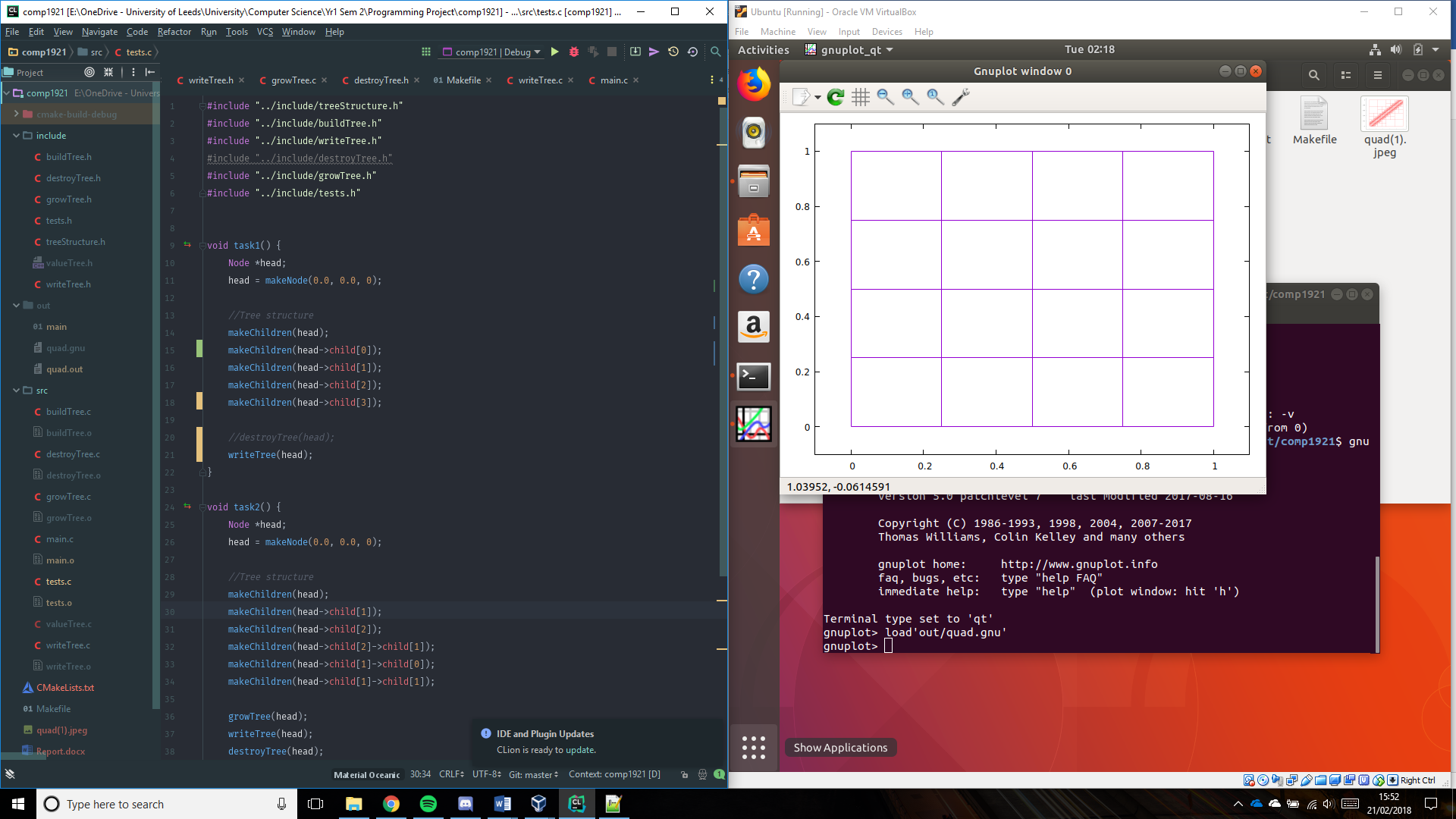
makeChildren**(**head**->**child**[**3**]);**

//destroyTree(head);

writeTree**(**head**);**

**}**

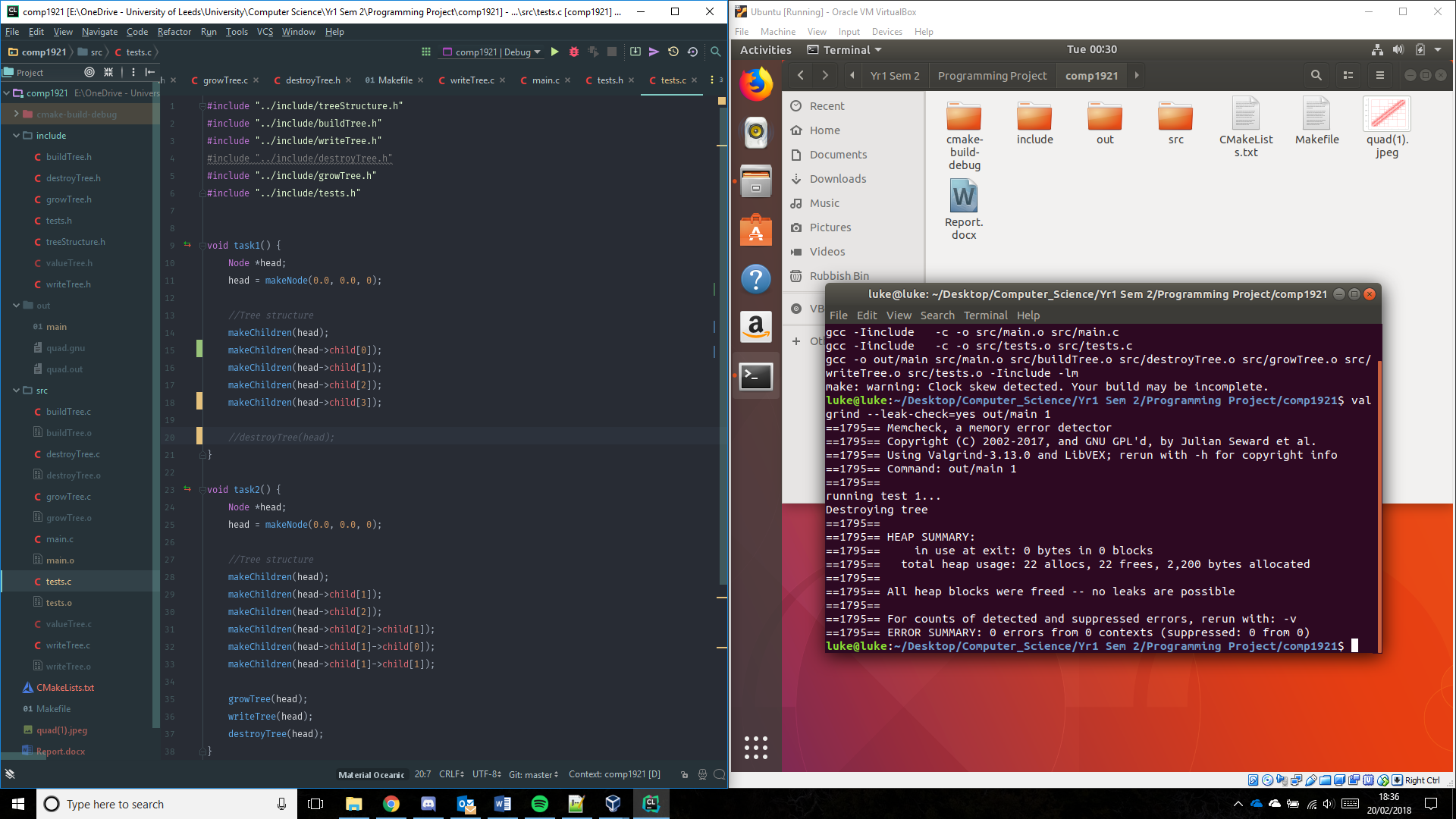
This creates the tree as shown below:



Valgrind output without the ‘destroyTree()’ function:



Valgrind output with ‘destroytree()’ function:



#### Test 2

For the second test I used this function to create a non-uniform Level 3 tree:

void task1**()** **{**

Node **\***head**;**

head **=** makeNode**(**0.0**,** 0.0**,** 0**);**

//Tree structure

makeChildren**(**head**);**

makeChildren**(**head**->**child**[**0**]);**

makeChildren**(**head**->**child**[**1**]);**

makeChildren**(**head**->**child**[**3**]);**

makeChildren**(**head**->**child**[**1**]->**child**[**1**]);**

makeChildren**(**head**->**child**[**3**]->**child**[**2**]);**

makeChildren**(**head**->**child**[**3**]->**child**[**3**]);**

//destroyTree(head);

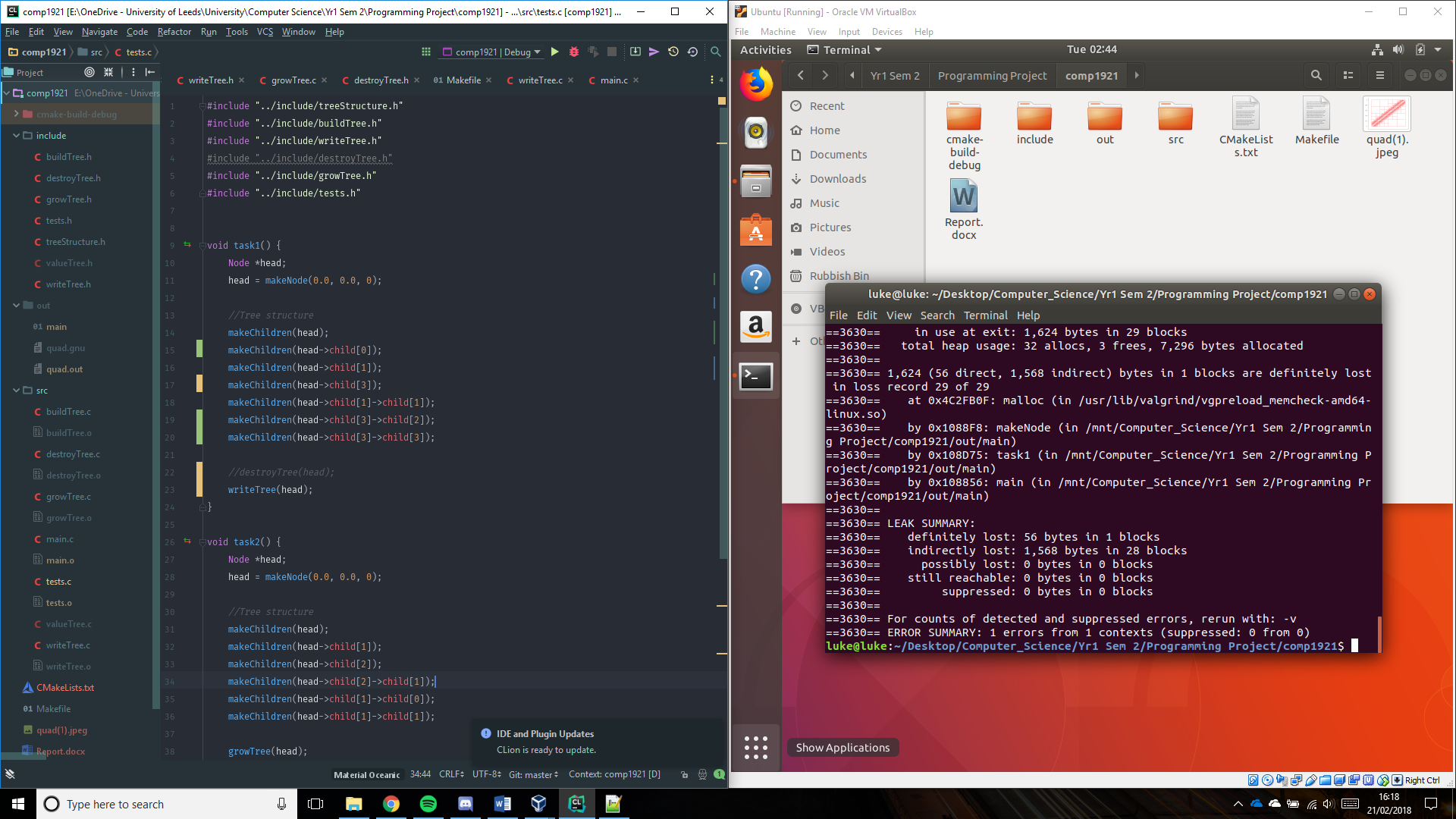
writeTree**(**head**);**

**}**

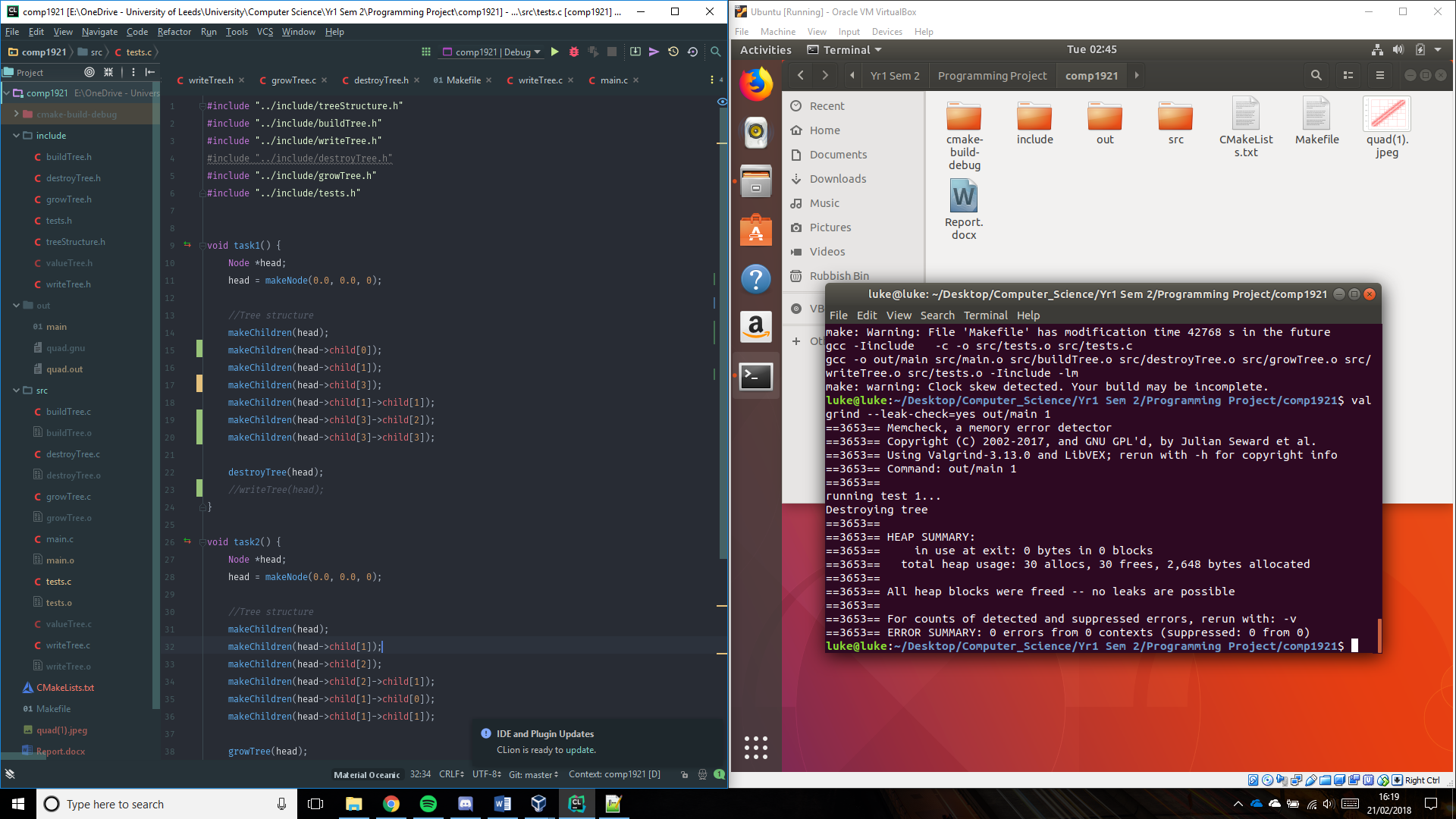
This creates the tree as shown below:



Valgrind output without the ‘destroyTree()’ function:



Valgrind output with ‘destroytree()’ function:



# Task 2

### New function

// Grow tree starting from head

void growTree**(**Node **\***head**)** **{**

printf**(**"Growing tree\n"**);**

growNode**(**head**);**

**}**

// Grow tree from starting node

void growNode**(**Node **\***node**)** **{**

**if** **(**node**->**child**[**0**]** **==** **NULL)**

makeChildren**(**node**);**

**else** **{**

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

growNode**(**node**->**child**[**i**]);**

**}**

**}**

### Function description

This function grows the tree by one level. Calling grow tree will call the grow node function passing the head of the tree as the argument. The grow node function will check whether the node passed as an argument has any children, if it doesn’t then the make children function is called on the node to grow it by one level. If it does, then the function loops through each of the nodes children and runs the function recursively.

### Tests

#### Test 1

For the first test I used this function to create a full tree at Level 2:

This test creates a new node called the head, then creates a full level 2 tree. It the grows the tree by 1 level by running the ‘growTree()’ function, writes the tree to ‘quad.out’ and destroys the tree ensuring no memory leakage.

void task2**()** **{**

Node **\***head**;**

head **=** makeNode**(**0.0**,** 0.0**,** 0**);**

//Tree structure

makeChildren**(**head**);**

makeChildren**(**head**->**child**[**0**]);**

makeChildren**(**head**->**child**[**1**]);**

makeChildren**(**head**->**child**[**2**]);**

makeChildren**(**head**->**child**[**3**]);**

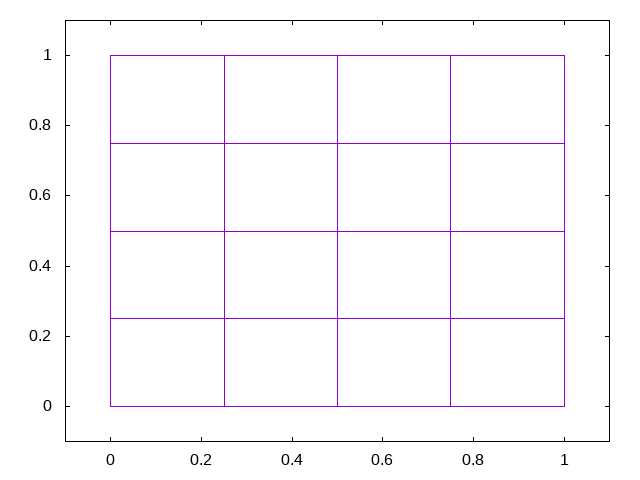
growTree**(**head**);**

writeTree**(**head**);**

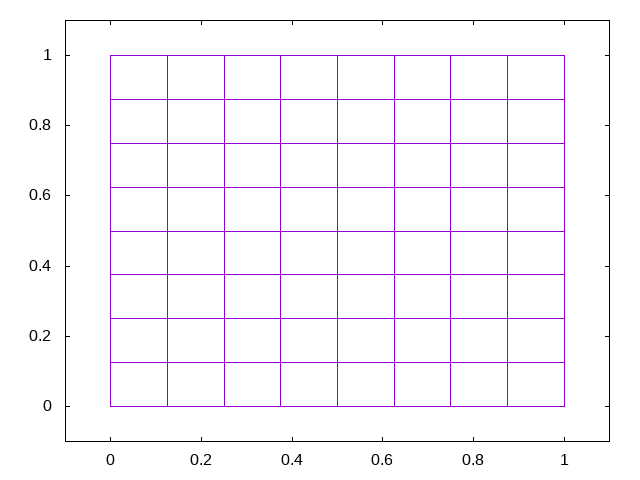
destroyTree**(**head**);**

**}**

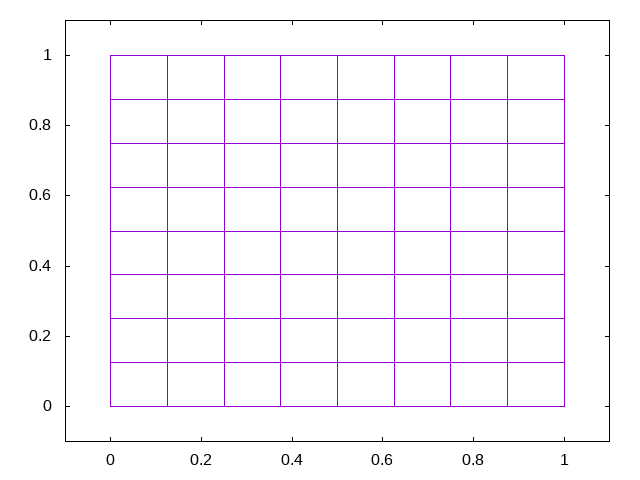
Tree before growing:



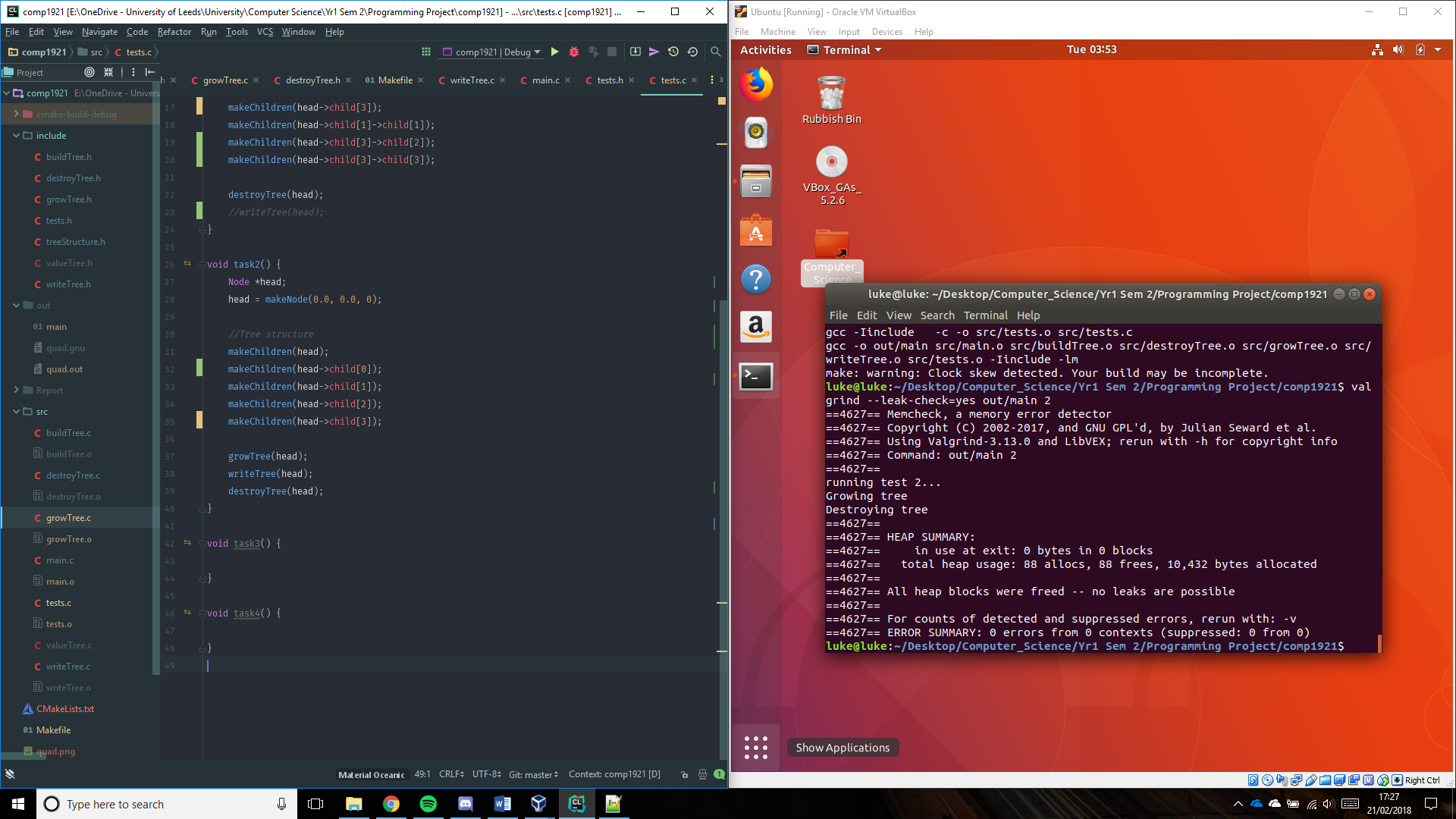
Tree after growing (Expected result):



Tree after growing (Actual result):



Valgrind report after running ‘growTree()’:



#### Test 2

For the second test I used this function to create a non-uniform tree at Level 3:

void task2**()** **{**

Node **\***head**;**

head **=** makeNode**(**0.0**,** 0.0**,** 0**);**

//Tree structure

makeChildren**(**head**);**

makeChildren**(**head**->**child**[**0**]);**

makeChildren**(**head**->**child**[**2**]);**

makeChildren**(**head**->**child**[**0**]->**child**[**1**]);**

makeChildren**(**head**->**child**[**2**]->**child**[**0**]);**

makeChildren**(**head**->**child**[**2**]->**child**[**3**]);**

growTree**(**head**);**

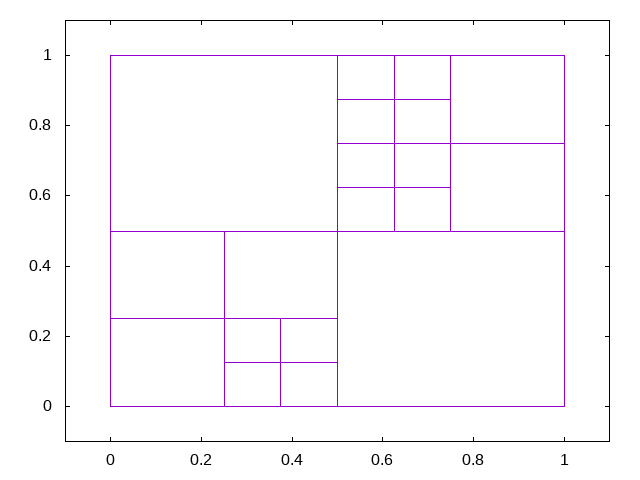
writeTree**(**head**);**

destroyTree**(**head**);**

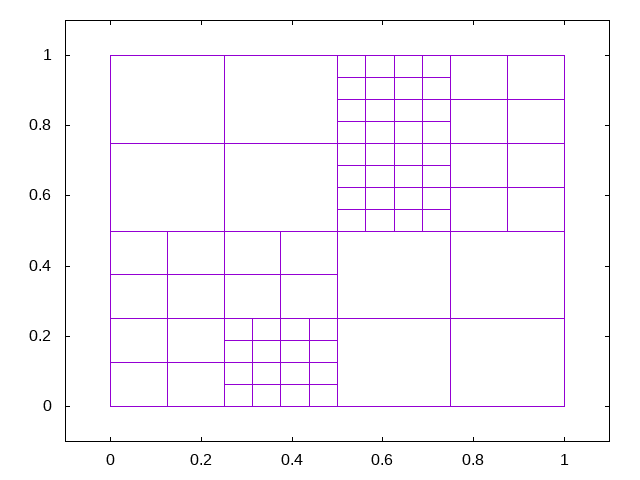
**}**

This test creates a new node called the head, then creates a non-uniform level 3 tree. It the grows the tree by 1 level by running the ‘growTree()’ function, writes the tree to ‘quad.out’ and destroys the tree ensuring no memory leakage.

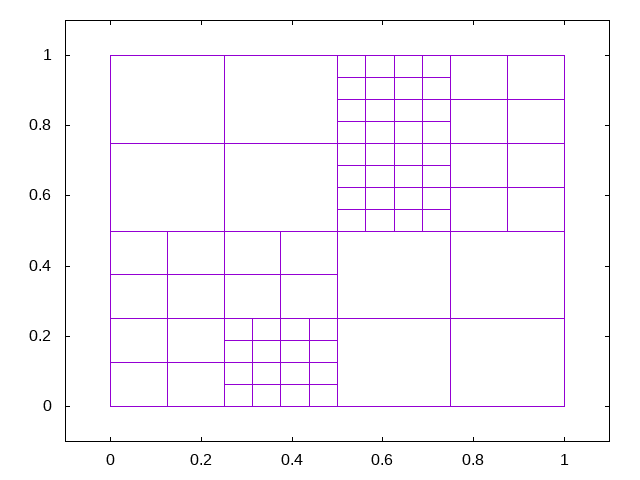
Tree before growing:



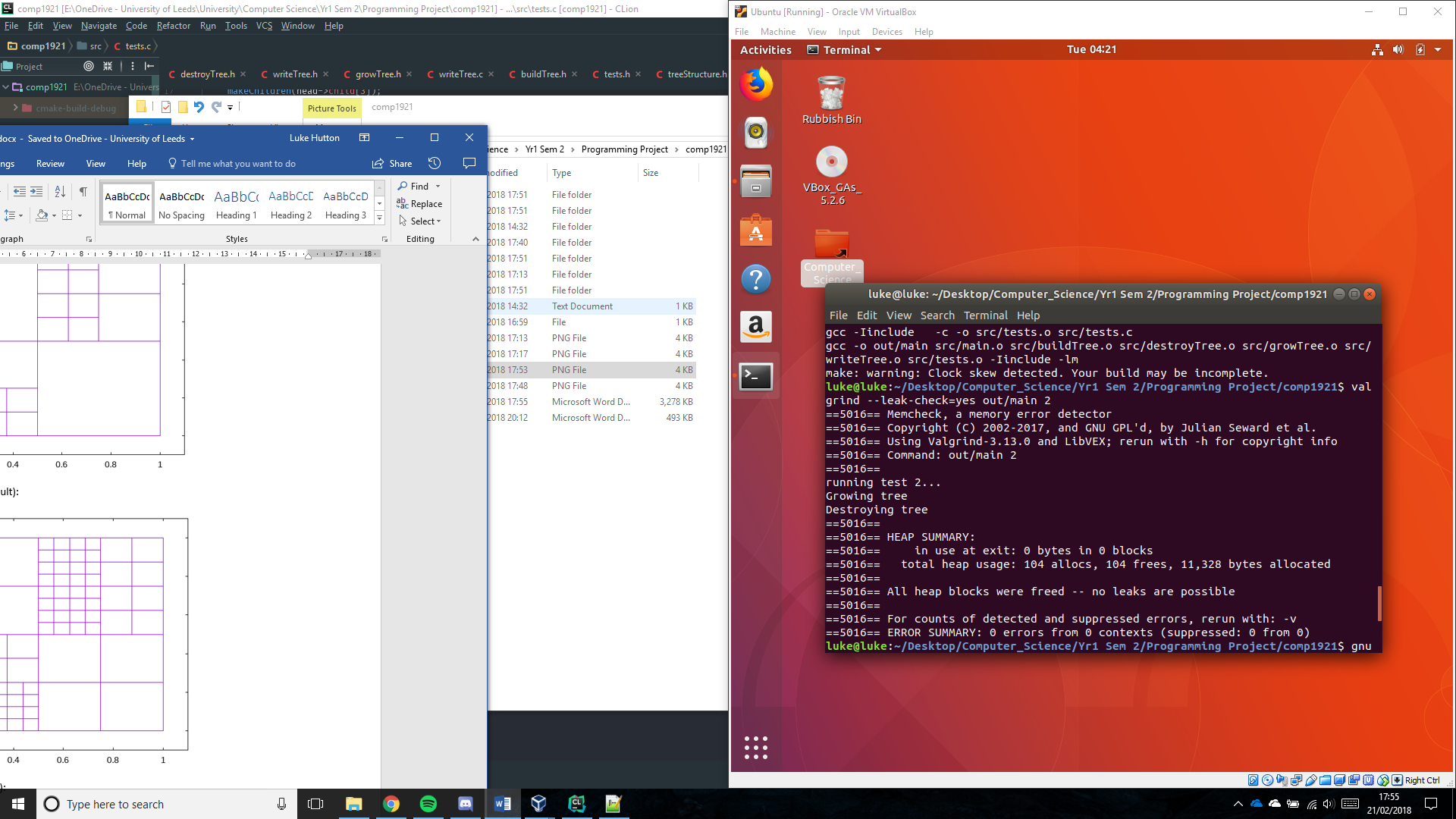
Tree after growing (Expected result):



Tree after growing (Actual result):



Valgrind report after running ‘growTree()’;



# Task 3

### Node size memory estimate

### Full tree memory estimates

A node is defined as follows:

struct qnode **{**

int level**;**

double xy**[**2**];**

struct qnode **\***child**[**4**];**

**};**

The node struct contains an integer ‘level’, 2 doubles ‘xy[]’ and 4 pointers to the same struct.

An integer is 4 bytes on most 32-bit and 64-bit systems. A double is 8 bytes and there are 2 so that’s 16 bytes. A pointer can be 4-bytes or 8-bytes long depending on the processor architecture (if it’s a 32-bit it is 4 bytes long, if it’s a 64-bit architecture it is 8 bytes long). So lets take it to be 8 bytes, that’s 32 bytes for 4 child nodes.

Overall, 4 + 16 + 32 = 52 bytes of memory per node.

### Estimated memory use for full trees of 5,6,7,8,9,10 levels

|  |  |  |
| --- | --- | --- |
| Tree level | No nodes | Estimated Memory use |
| 5 | 45 = 1024 | 52 KB |
| 6 | 46 = 4096 | 208 KB |
| 7 | 47 = 16384 | 832 KB |
| 8 | 48 = 65536 | 3 MB |
| 9 | 49 = 262144 | 13 MB |
| 10 | 410 = 1048576 | 52 MB |

### Actual memory use for full trees of 5,6,7,8,9,10 levels

|  |  |  |  |
| --- | --- | --- | --- |
| Tree level | No nodes | Estimated Memory use | Actual memory use |
| 5 | 45 = 1024 | 52 KB |  |
| 6 | 46 = 4096 | 208 KB |  |
| 7 | 47 = 16384 | 832 KB |  |
| 8 | 48 = 65536 | 3 MB |  |
| 9 | 49 = 262144 | 13 MB | - |
| 10 | 410 = 1048576 | 52 MB | - |

### If you would like to limit the overall memory use of the application to 20Mb what maximum level should you choose?

To limit the memory use of the application to 20 Mb = 2.5 MB we need to choose a level that doesn’t exceed that amount. Therefore, we need to limit the tree to level 7.

### Max tree level implementation

#### New code

In buildTree.c:

void makeChildren**(**Node **\***parent**)** **{**

double x **=** parent**->**xy**[**0**];**

double y **=** parent**->**xy**[**1**];**

int level **=** parent**->**level**;**

double hChild **=** pow**(**2.0**,** **-(**level **+** 1**));**

// Limit child node level creation to MAX\_LEVEL

**if** **(**level + 1 **>** MAX\_LEVEL**)**

**return;**

parent**->**child**[**0**]** **=** makeNode**(**x**,** y**,** level **+** 1**);**

parent**->**child**[**1**]** **=** makeNode**(**x **+** hChild**,** y**,** level **+** 1**);**

parent**->**child**[**2**]** **=** makeNode**(**x **+** hChild**,** y **+** hChild**,** level **+** 1**);**

parent**->**child**[**3**]** **=** makeNode**(**x**,** y **+** hChild**,** level **+** 1**);**

**}**

In tests.h:

extern const int MAX\_LEVEL**;**

In tests.c:

const int MAX\_LEVEL **=** 7**;**

#### Function description

In order to limit the tree size to a certain level I added an if statement to the makeChildren function. I chose to add it to this function as it will only check once when the children are being made and not 4 individual times when a node is being created for each of the children. This if statement checks whether the level the nodes are going to be added at is more than the maximum level, if it is then a message is printed and the function returns without creating the child nodes. To make it easier to change the value of the constant I added it to the tests.c file where it can easily be changed with the tests.

### Testing max tree level

Test used to test max level implementation:

void task3**()** **{**

Node **\***head**;**

head **=** makeNode**(**0.0**,** 0.0**,** 0**);**

//Tree structure

makeChildren**(**head**);**

growTree**(**head**);**

growTree**(**head**);**

growTree**(**head**);**

//Test

growTree**(**head**);**

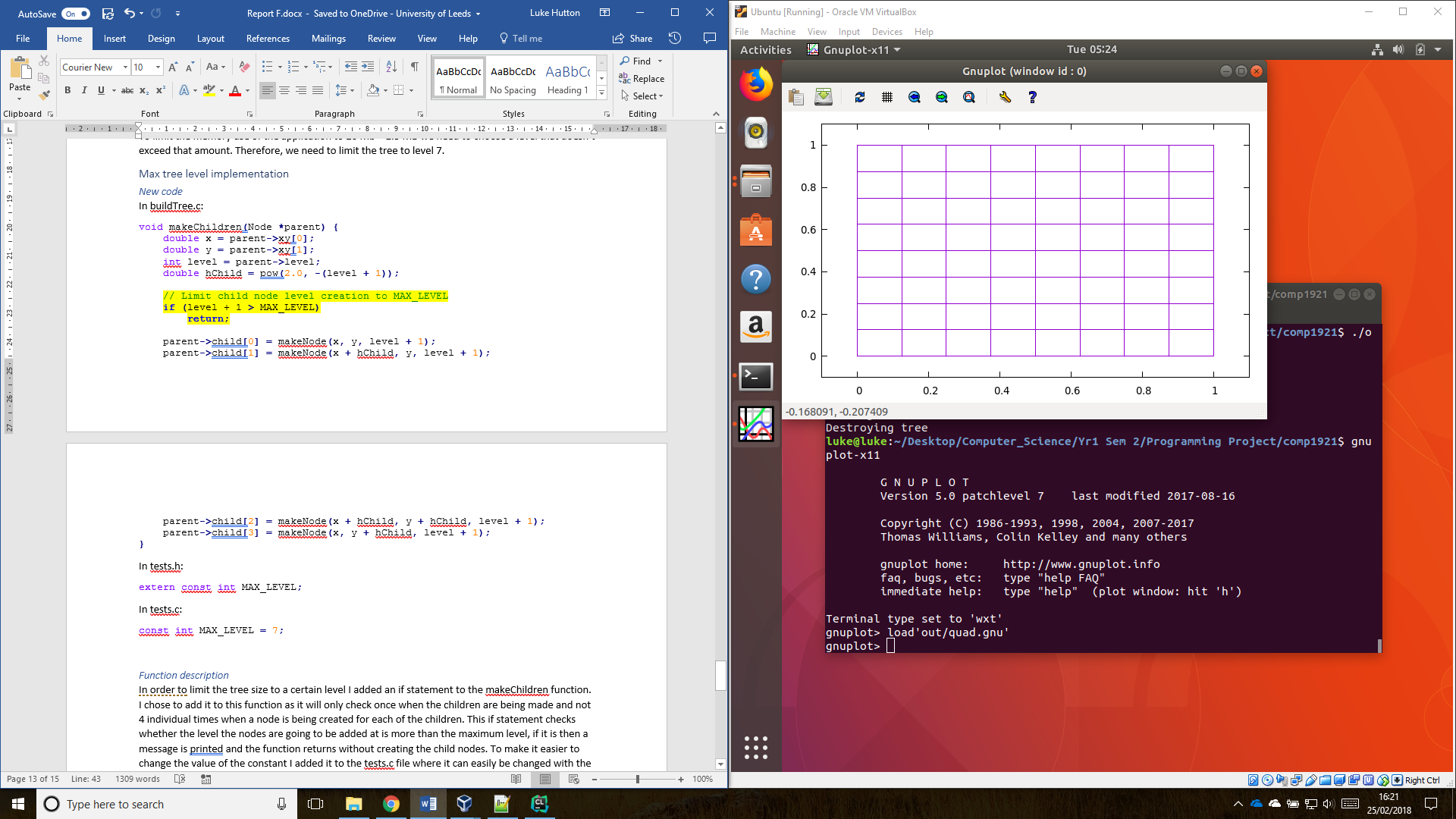
writeTree**(**head**);**

destroyTree**(**head**);**

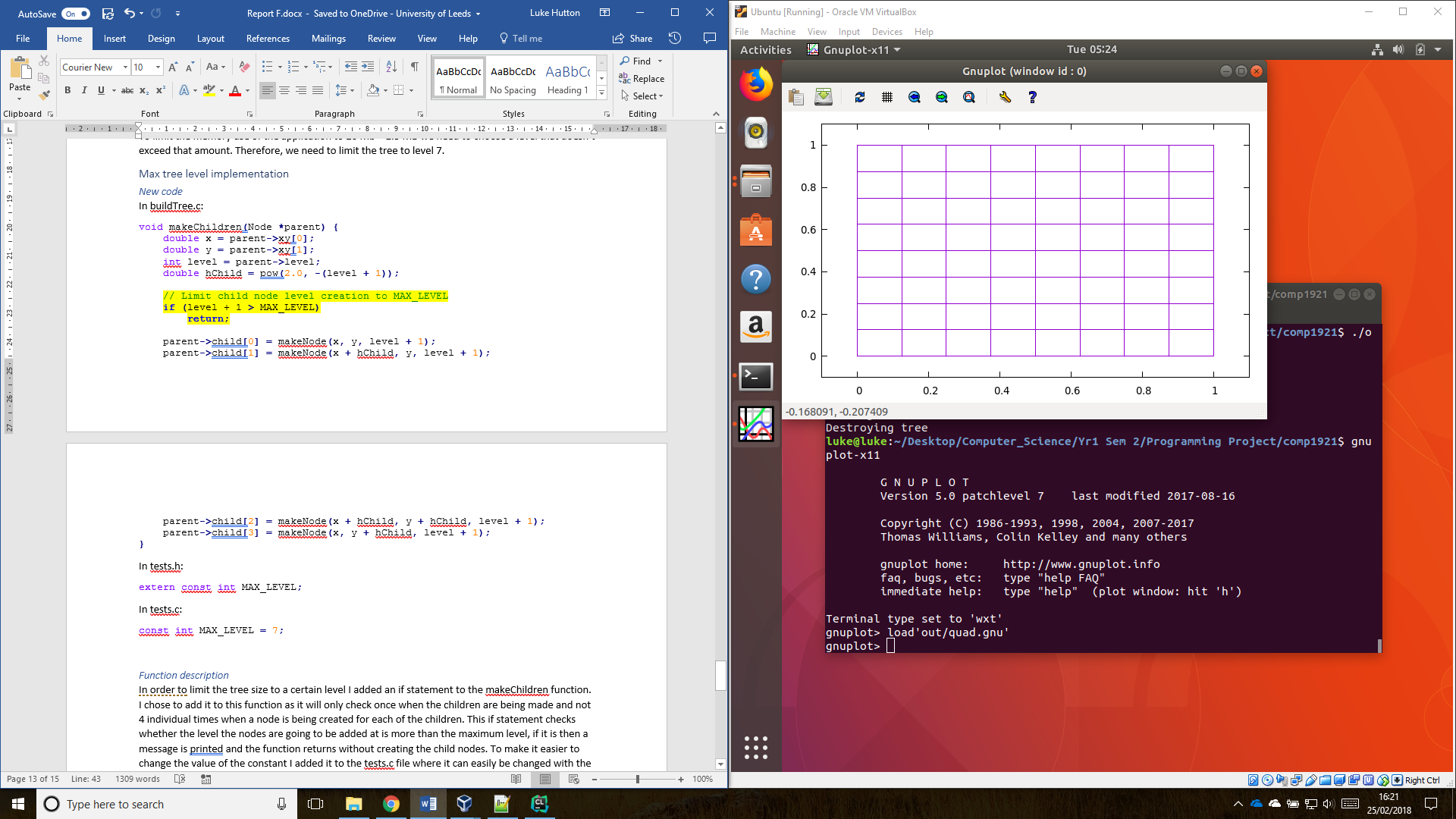
**}**

#### Test when max tree level is 3

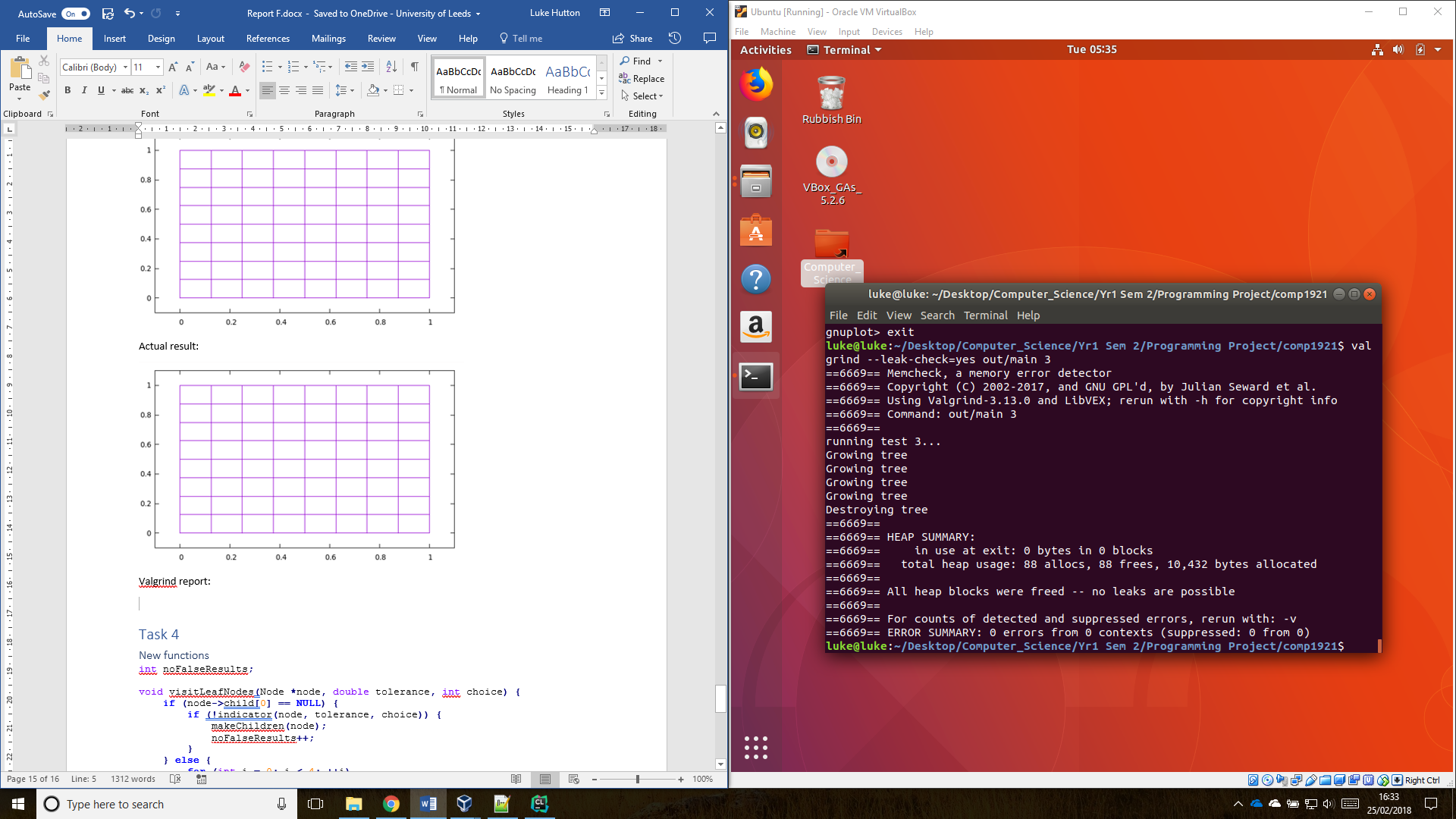
Expected result:



Actual result:

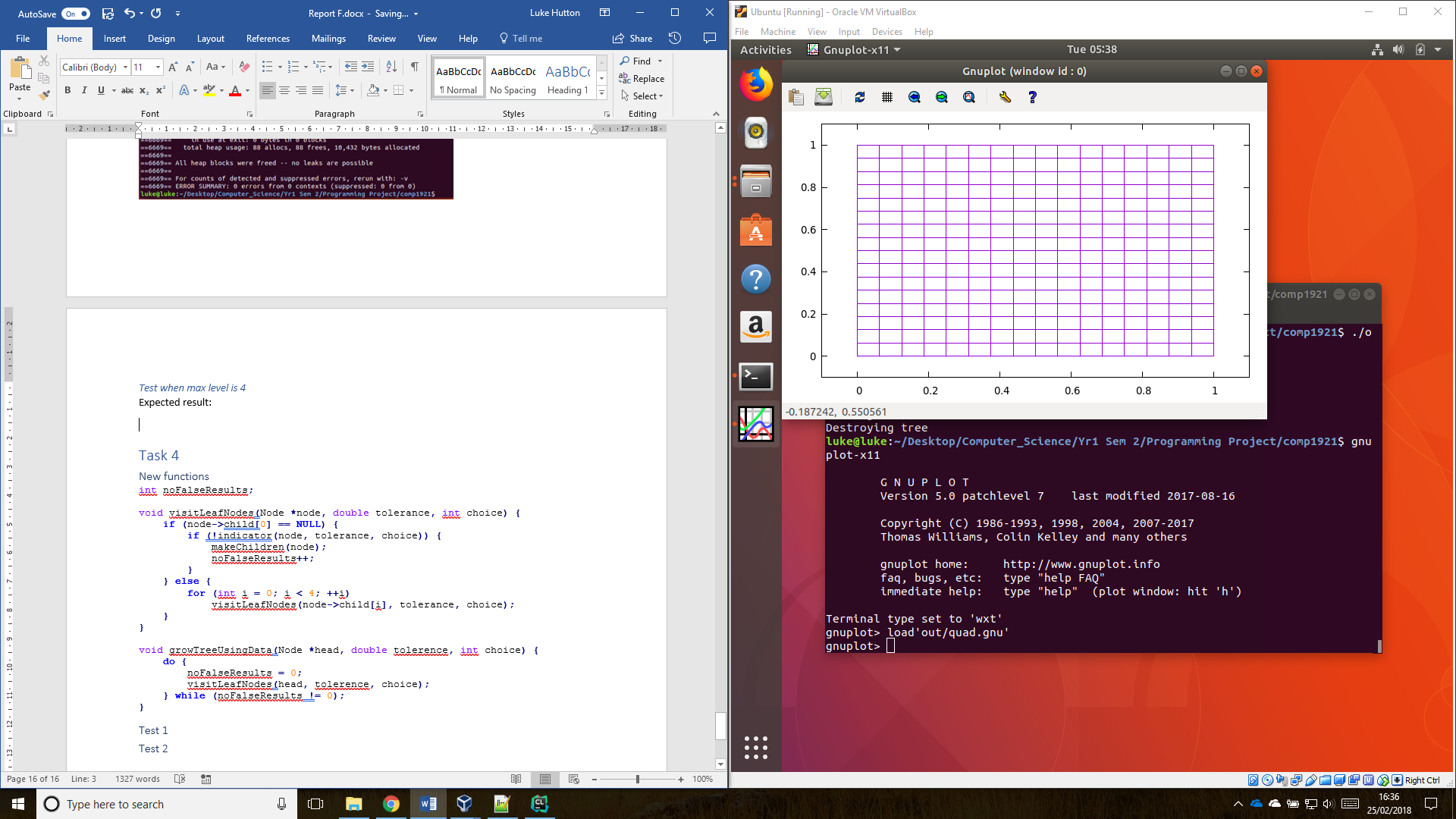


Valgrind report:

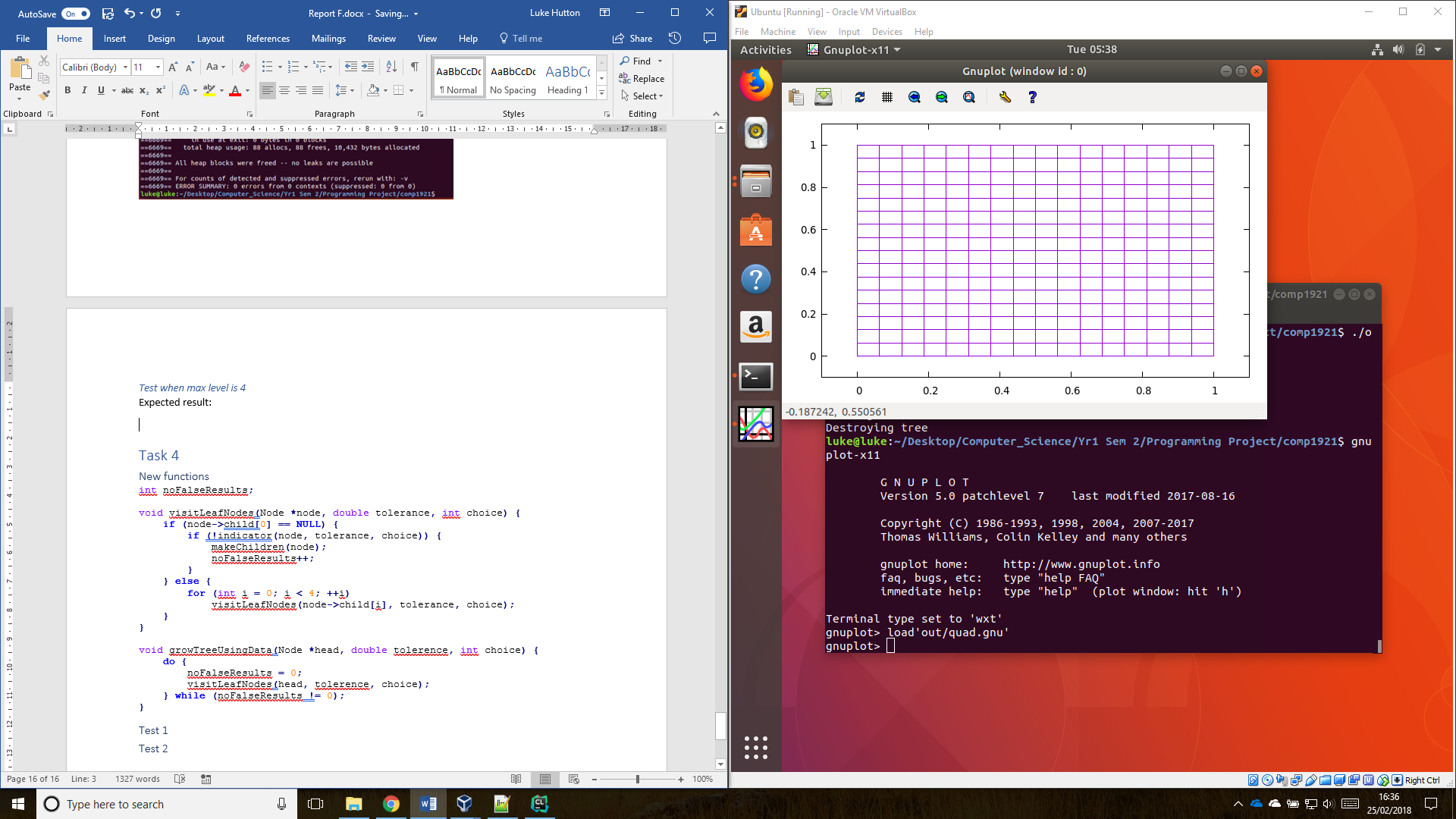


#### Test when max level is 4

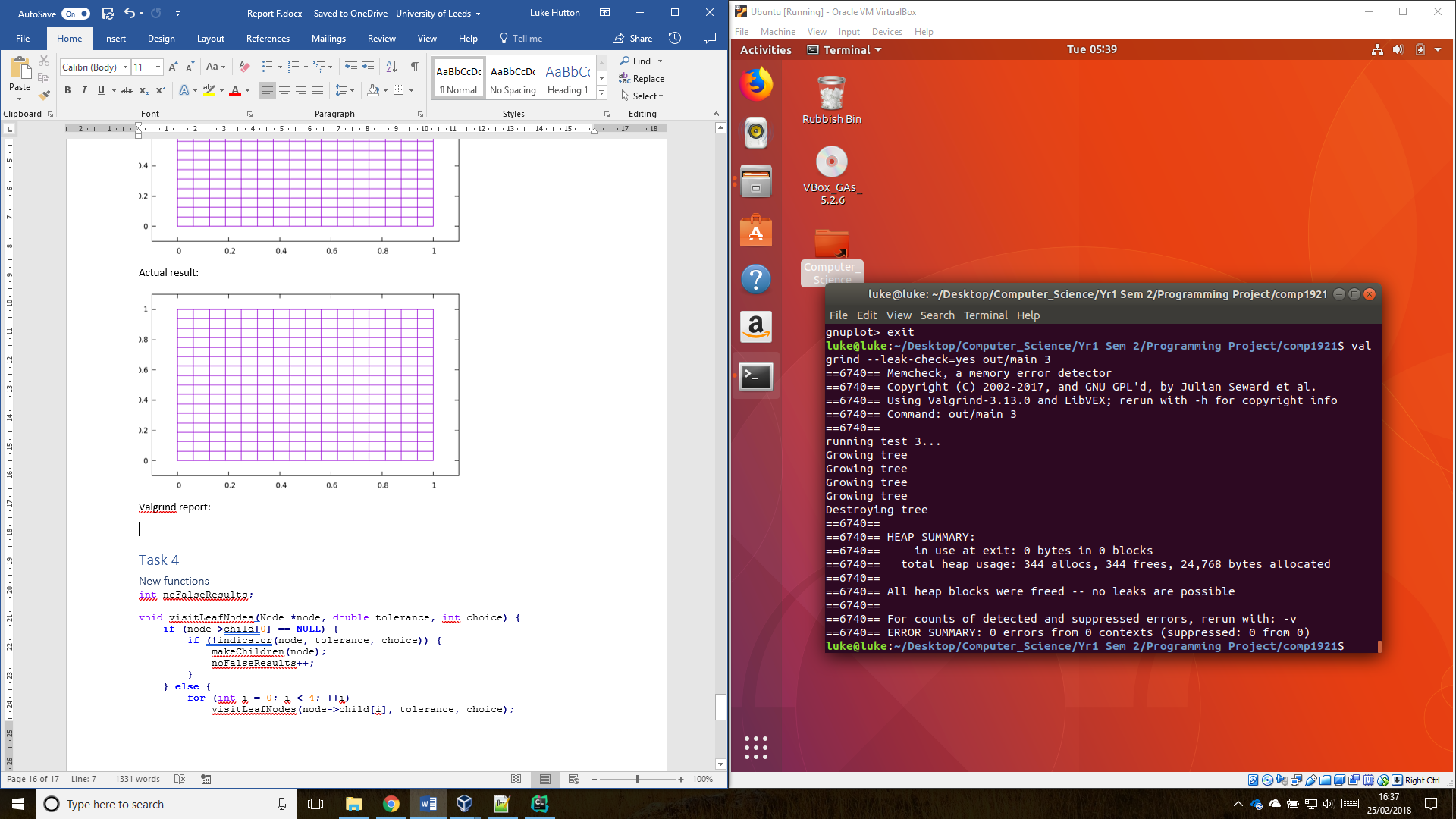
Expected result:



Actual result:



Valgrind report:



# Task 4

### New functions

int noFalseResults**;**

void visitLeafNodes**(**Node **\***node**,** double tolerance**,** int choice**)** **{**

**if** **(**node**->**child**[**0**]** **==** **NULL)** **{**

**if** **(!**indicator**(**node**,** tolerance**,** choice**))** **{**

noFalseResults**++;**

makeChildren**(**node**);**

**}**

**}** **else** **{**

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

visitLeafNodes**(**node**->**child**[**i**],** tolerance**,** choice**);**

**}**

**}**

void growTreeUsingData**(**Node **\***head**,** double tolerence**,** int choice**)** **{**

**do** **{**

noFalseResults **=** 0**;**

visitLeafNodes**(**head**,** tolerence**,** choice**);**

**}** **while** **(**noFalseResults **!=** 0**);**

**}**

### Test 1

When choice = 0, and tolerance = 0.5

void task4**()** **{**

Node **\***head**;**

head **=** makeNode**(**0.0**,** 0.0**,** 0**);**

makeChildren**(**head**);**

growTree**(**head**);**

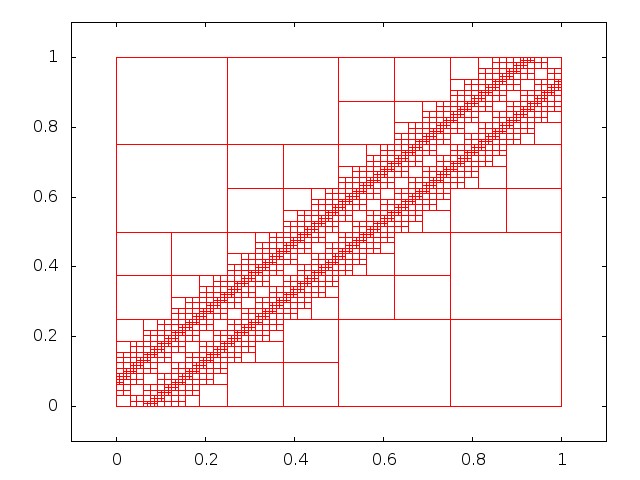
growTreeUsingData**(**head**,** 0.5**,** 0**);**

writeTree**(**head**);**

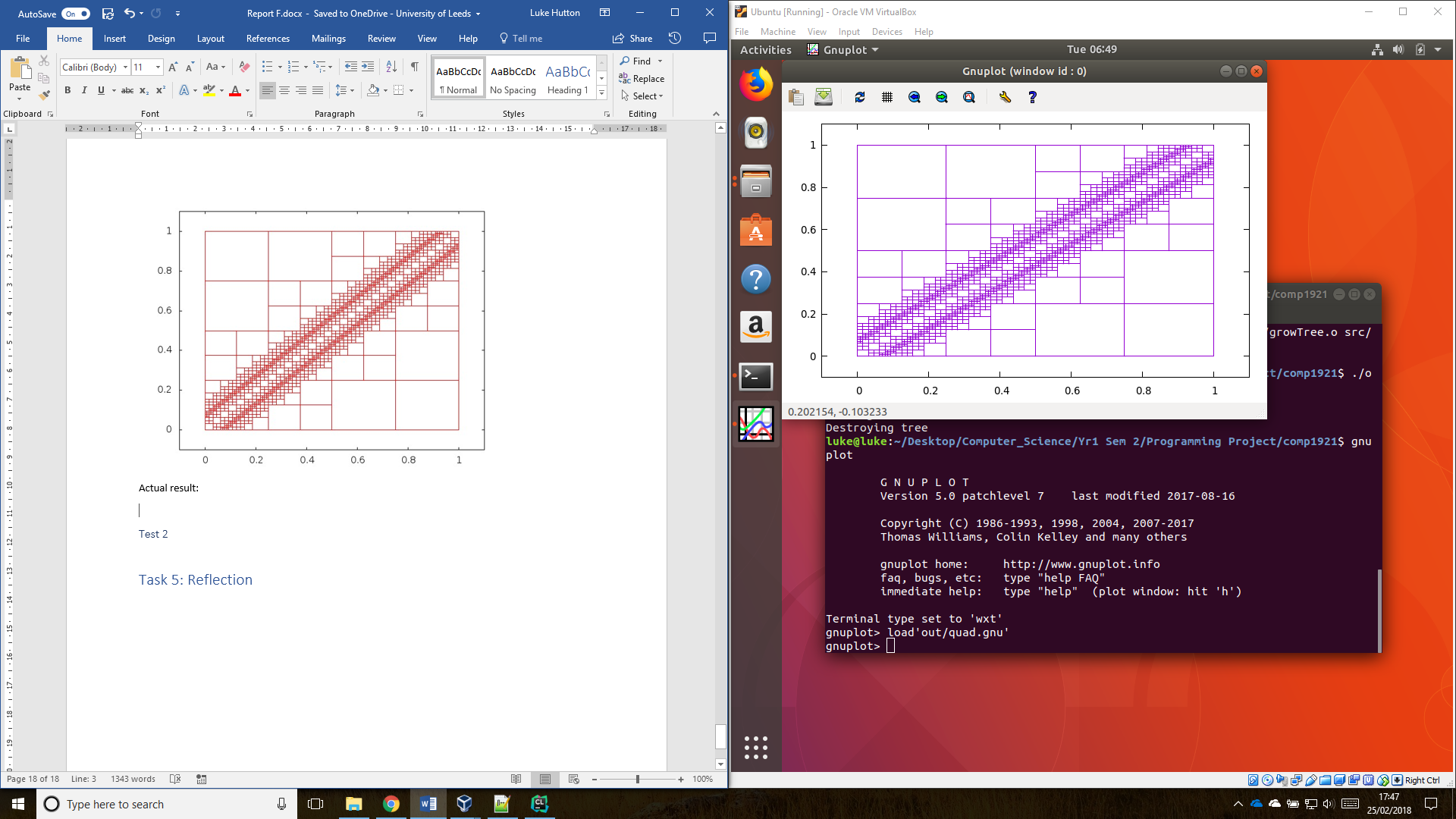
destroyTree**(**head**);**

**}**

Expected result:



Actual result:



### Test 2

When choice = 1 and tolerance = 0.2

void task4**()** **{**

Node **\***head**;**

head **=** makeNode**(**0.0**,** 0.0**,** 0**);**

makeChildren**(**head**);**

growTree**(**head**);**

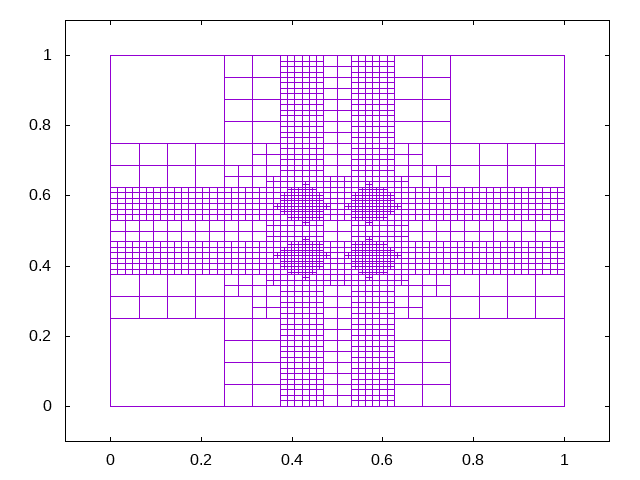
growTreeUsingData**(**head**,** 0.2**,** 1**);**

writeTree**(**head**);**

destroyTree**(**head**);**

**}**

Result:



# Task 5: Reflection

## What went well with this project?

## What was the hardest part of this work?

There were multiple things were hard with this project. The first was getting all the applications to work correctly. The second was getting the make file to compile each of the source files and output to the correct place.