

Written Report for Programming Project 1

COMP1921

Luke Hutton

SC17LJTH

Contents

Task 1	2
New function	2
Function description	2
Tests.....	2
Task 2	7
New function	7
Function description	7
Tests.....	7
Task 3	13
Node size memory estimate	13
Estimated memory use for full trees of 5,6,7,8,9,10 levels	13
Actual memory use for full trees of 5,6,7,8,9,10 levels	13
If you would like to limit the overall memory use of the application to 20Mb what maximum level should you choose?	14
Max tree level implementation.....	14
Testing max tree level	14
Task 4	17
New functions	17
Test 1	17
Test 2	19
Task 5: Reflection.....	20
What went well with this project?	20
What was the hardest part of this work?	20
References	20

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

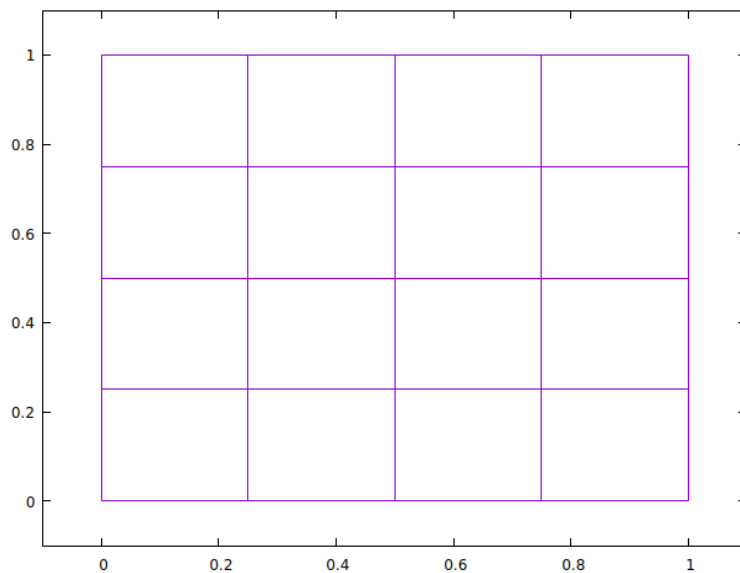
For the first test I used this function to create a full tree at Level 2:

```
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:

```
luke@luke: ~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921
File Edit View Search Terminal Help
==3295==      in use at exit: 1,176 bytes in 21 blocks
==3295==    total heap usage: 24 allocs, 3 frees, 6,848 bytes allocated
==3295==
==3295== 1,176 (56 direct, 1,120 indirect) bytes in 1 blocks are definitely lost
in loss record 21 of 21
==3295==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==3295==    by 0x1088F8: makeNode (in /mnt/Computer_Science/Yr1 Sem 2/Programmin
g Project/comp1921/out/main)
==3295==    by 0x108D75: task1 (in /mnt/Computer_Science/Yr1 Sem 2/Programming P
roject/comp1921/out/main)
==3295==    by 0x108856: main (in /mnt/Computer_Science/Yr1 Sem 2/Programming Pr
oject/comp1921/out/main)
==3295==
==3295== LEAK SUMMARY:
==3295==    definitely lost: 56 bytes in 1 blocks
==3295==    indirectly lost: 1,120 bytes in 20 blocks
==3295==    possibly lost: 0 bytes in 0 blocks
==3295==    still reachable: 0 bytes in 0 blocks
==3295==    suppressed: 0 bytes in 0 blocks
==3295==
==3295== For counts of detected and suppressed errors, rerun with: -v
==3295== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$
```

As you can see the nodes aren't destroyed and around 1176 bytes of memory is leaked.

Valgrind output with 'destroytree()' function:

```
luke@luke: ~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921
File Edit View Search Terminal Help
gcc -Iinclude -c -o src/main.o src/main.c
gcc -Iinclude -c -o src/tests.o src/tests.c
gcc -o out/main src/main.o src/buildTree.o src/destroyTree.o src/growTree.o src/
writeTree.o src/tests.o -Iinclude -lm
make: warning: Clock skew detected. Your build may be incomplete.
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$ val
grind --leak-check=yes out/main 1
==1795== Memcheck, a memory error detector
==1795== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1795== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==1795== Command: out/main 1
==1795==
running test 1...
Destroying tree
==1795==
==1795== HEAP SUMMARY:
==1795==   in use at exit: 0 bytes in 0 blocks
==1795==   total heap usage: 22 allocs, 22 frees, 2,200 bytes allocated
==1795==
==1795== All heap blocks were freed -- no leaks are possible
==1795==
==1795== For counts of detected and suppressed errors, rerun with: -v
==1795== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$
```

With the destroyTree() function all memory used by the nodes is freed and there is no memory leak.

Test 2

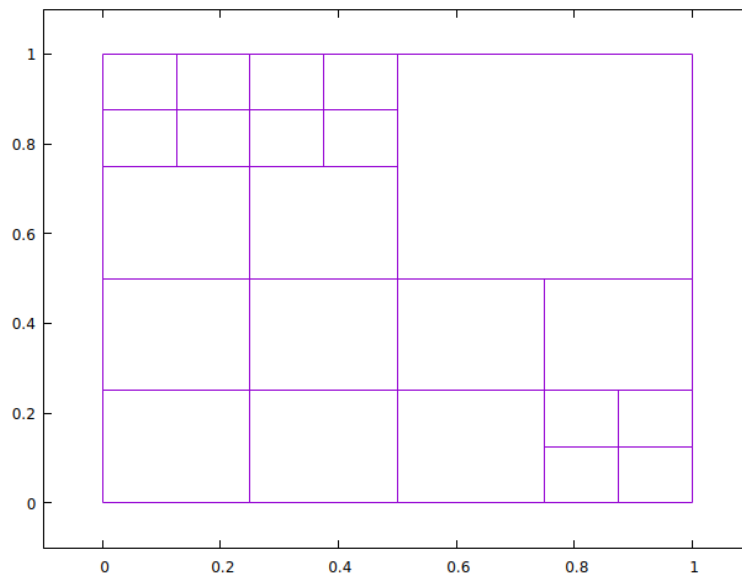
For the second test I used this function to create a non-uniform Level 3 tree:

```
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:

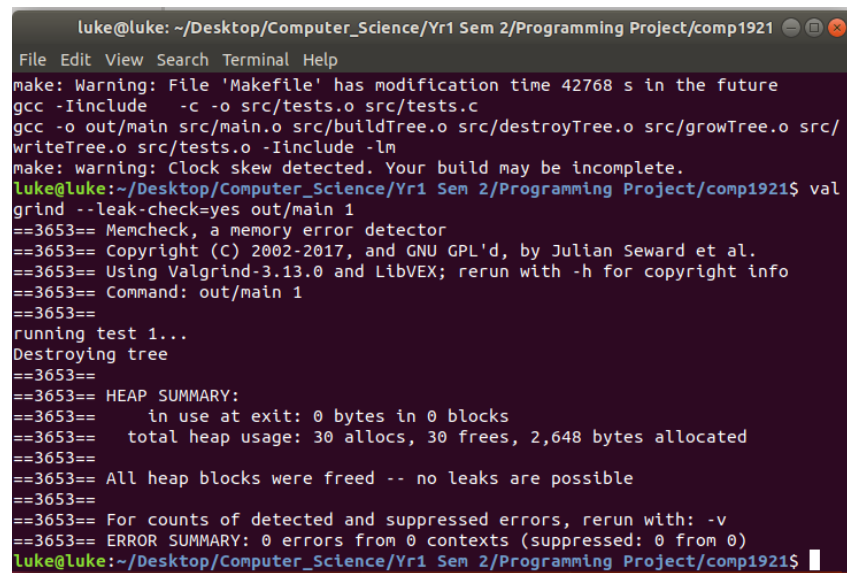


Valgrind output without the 'destroyTree()' function:

```
luke@luke: ~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921
File Edit View Search Terminal Help
==3630==      in use at exit: 1,624 bytes in 29 blocks
==3630==    total heap usage: 32 allocs, 3 frees, 7,296 bytes allocated
==3630==
==3630== 1,624 (56 direct, 1,568 indirect) bytes in 1 blocks are definitely lost
in loss record 29 of 29
==3630==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==3630==    by 0x1088F8: makeNode (in /mnt/Computer_Science/Yr1 Sem 2/Programmin
g Project/comp1921/out/main)
==3630==    by 0x108D75: task1 (in /mnt/Computer_Science/Yr1 Sem 2/Programming P
roject/comp1921/out/main)
==3630==    by 0x108856: main (in /mnt/Computer_Science/Yr1 Sem 2/Programming Pr
oject/comp1921/out/main)
==3630==
==3630== LEAK SUMMARY:
==3630==    definitely lost: 56 bytes in 1 blocks
==3630==    indirectly lost: 1,568 bytes in 28 blocks
==3630==    possibly lost: 0 bytes in 0 blocks
==3630==    still reachable: 0 bytes in 0 blocks
==3630==    suppressed: 0 bytes in 0 blocks
==3630==
==3630== For counts of detected and suppressed errors, rerun with: -v
==3630== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$
```

Without the destroyTree() function the program leaks 1624 bytes of memory as the nodes created aren't freed from memory.

Valgrind output with 'destroytree()' function:

A terminal window with a dark background and light text. The window title is 'luke@luke: ~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921'. The terminal shows the output of a 'make' command, which includes warnings about a future modification time and a clock skew. Then, the user runs 'valgrind --leak-check=yes out/main 1'. The output shows Valgrind's version (3.13.0) and a 'running test 1...' message. It then says 'Destroying tree' and shows a 'HEAP SUMMARY' indicating 0 bytes in use at exit, 30 allocations, 30 frees, and 2,648 bytes allocated. It concludes that all heap blocks were freed and no leaks are possible. The terminal ends with the prompt 'luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921\$'.

With the destroyTree() function all nodes are destroyed and no memory is leaked.

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:

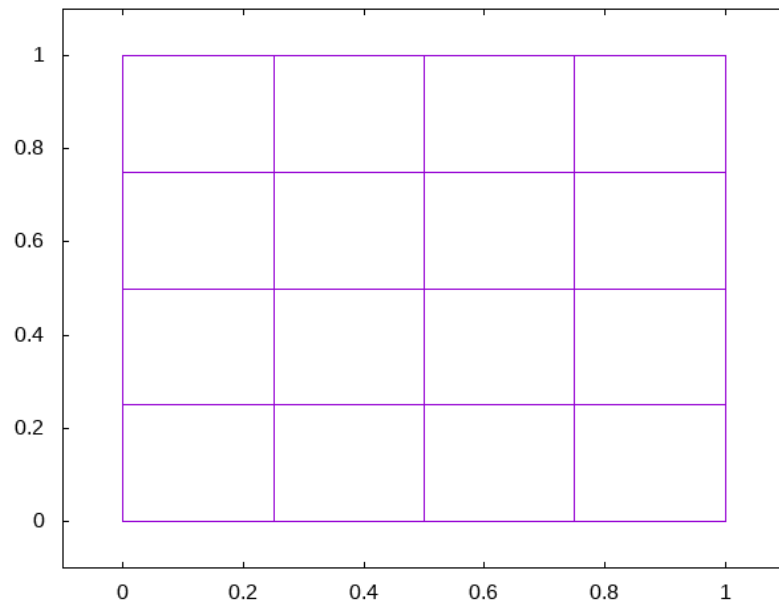
This test creates a new node called the head, then creates a full level 2 tree. It then grows the tree by 1 level by running the 'growTree()' function, writes the tree to 'quad.out' and destroys the tree ensuring no memory leakage.

```
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);
}
```

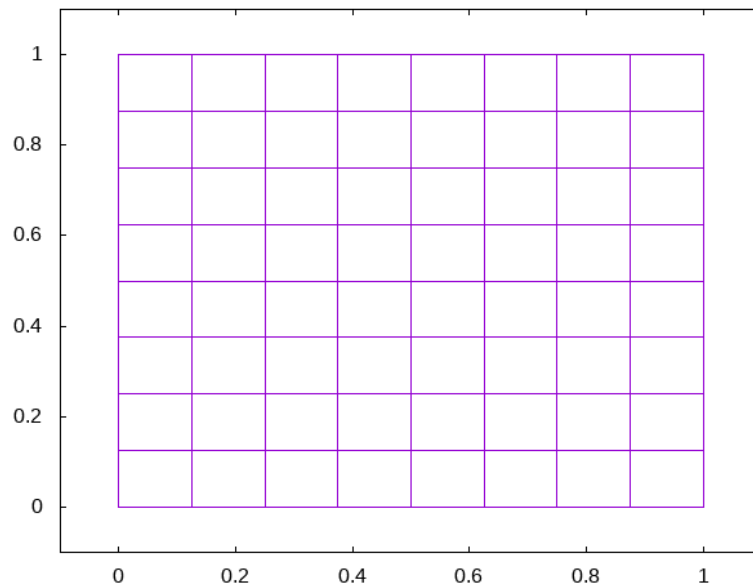
Tree before growing:



Tree after growing (Expected result):

Each node in the tree before growing should have a set of children within it. Therefore the result should be an 8x8 grid of squares.

Tree after growing (Actual result):



Valgrind report after running 'growTree()':

```
luke@luke: ~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921
File Edit View Search Terminal Help
gcc -Iinclude -c -o src/tests.o src/tests.c
gcc -o out/main src/main.o src/buildTree.o src/destroyTree.o src/growTree.o src/
writeTree.o src/tests.o -Iinclude -lm
make: warning: Clock skew detected. Your build may be incomplete.
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$ val
grind --leak-check=yes out/main 2
==4627== Memcheck, a memory error detector
==4627== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4627== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4627== Command: out/main 2
==4627==
running test 2...
Growing tree
Destroying tree
==4627==
==4627== HEAP SUMMARY:
==4627==   in use at exit: 0 bytes in 0 blocks
==4627==   total heap usage: 88 allocs, 88 frees, 10,432 bytes allocated
==4627==
==4627== All heap blocks were freed -- no leaks are possible
==4627==
==4627== For counts of detected and suppressed errors, rerun with: -v
==4627== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$
```

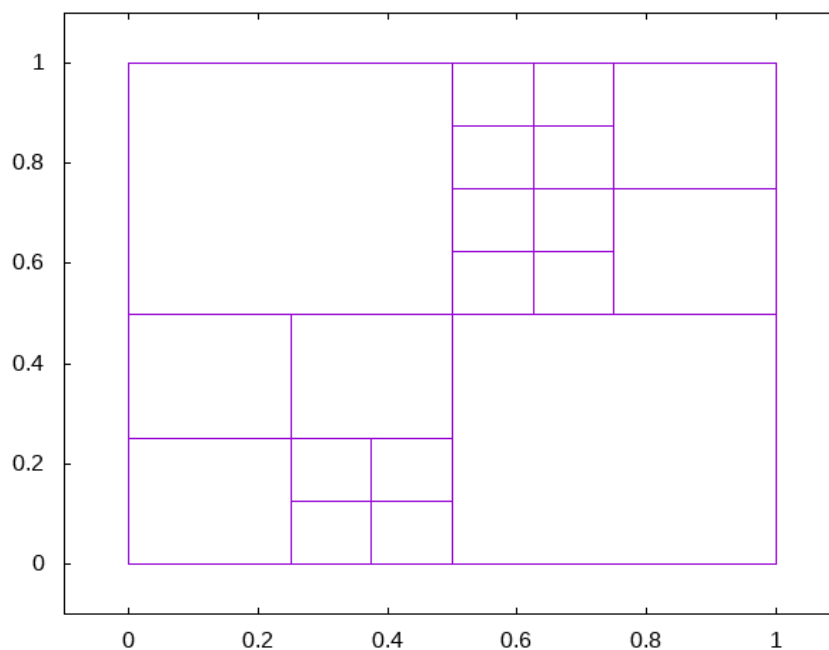
Test 2

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);  
}
```

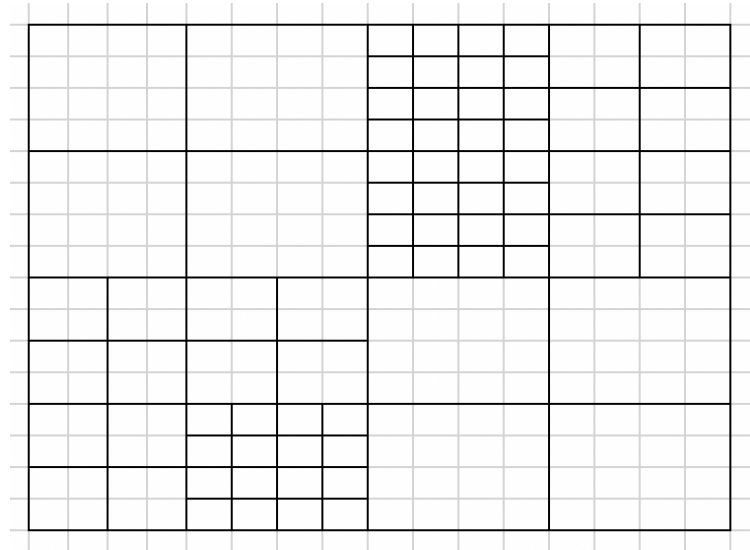
This test creates a new node called the head, then creates a non-uniform level 3 tree. It grows the tree by 1 level by running the 'growTree()' function, writes the tree to 'quad.out' and destroys the tree ensuring no memory leakage.

Tree before growing:

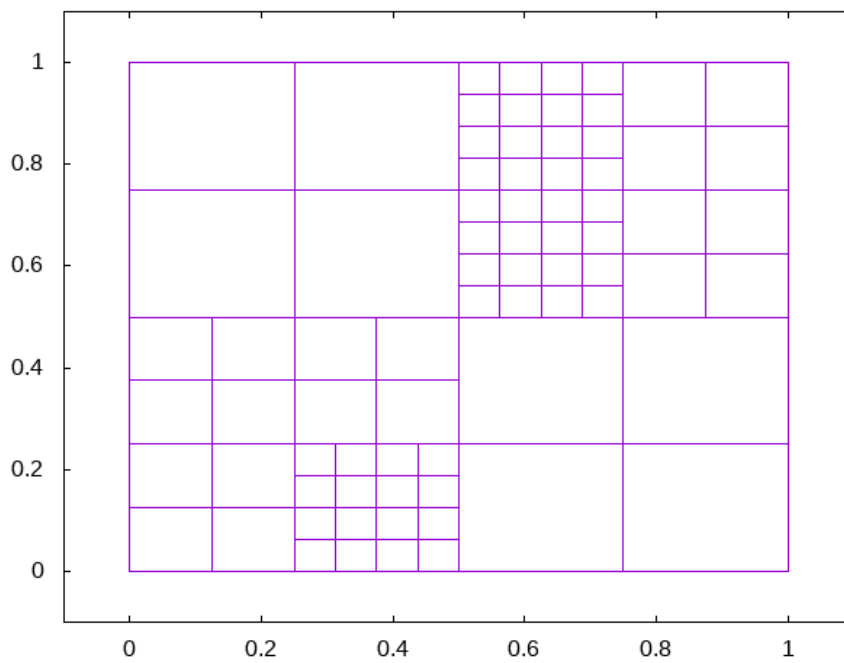


Tree after growing (Expected result):

Each node in the tree should be subdivided into 4. The result should look something like below.



Tree after growing (Actual result):



Valgrind report after running 'growTree()':

```
luke@luke: ~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921
File Edit View Search Terminal Help
gcc -Iinclude -c -o src/tests.o src/tests.c
gcc -o out/main src/main.o src/buildTree.o src/destroyTree.o src/growTree.o src/
writeTree.o src/tests.o -Iinclude -lm
make: warning: Clock skew detected. Your build may be incomplete.
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$ val
grind --leak-check=yes out/main 2
==5016== Memcheck, a memory error detector
==5016== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5016== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5016== Command: out/main 2
==5016==
running test 2...
Growing tree
Destroying tree
==5016==
==5016== HEAP SUMMARY:
==5016==    in use at exit: 0 bytes in 0 blocks
==5016==   total heap usage: 104 allocs, 104 frees, 11,328 bytes allocated
==5016==
==5016== All heap blocks were freed -- no leaks are possible
==5016==
==5016== For counts of detected and suppressed errors, rerun with: -v
==5016== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$ gnu
```

Task 3

Node size memory estimate

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 [3]. A pointer can be 4-bytes or 8-bytes long [4] 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	$\left(\frac{1-4^{5+1}}{1-4}\right) = 1365$	70 KB
6	$\left(\frac{1-4^{6+1}}{1-4}\right) = 5461$	283 KB
7	$\left(\frac{1-4^{7+1}}{1-4}\right) = 21845$	1135 KB
8	$\left(\frac{1-4^{8+1}}{1-4}\right) = 87381$	4.5 MB
9	$\left(\frac{1-4^{9+1}}{1-4}\right) = 349525$	18 MB
10	$\left(\frac{1-4^{10+1}}{1-4}\right) = 1398101$	73 MB

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

To get the actual memory usage of the program I used valgrind without the destroy tree option and looked at how much memory was in use on exit.

Tree level	No nodes	Estimated Memory use	Actual memory use
5	$\left(\frac{1-4^{5+1}}{1-4}\right) = 1365$	70 KB	74 KB
6	$\left(\frac{1-4^{6+1}}{1-4}\right) = 5461$	283 KB	298 KB
7	$\left(\frac{1-4^{7+1}}{1-4}\right) = 21845$	1135 KB	1.2 MB
8	$\left(\frac{1-4^{8+1}}{1-4}\right) = 87381$	4.5 MB	4.7 MB
9	$\left(\frac{1-4^{9+1}}{1-4}\right) = 349525$	18 MB	-
10	$\left(\frac{1-4^{10+1}}{1-4}\right) = 1398101$	73 MB	-

The results show that the actual memory usage is larger than the estimated usage. This is due to the fact that when a call to malloc is made, the program allocates 56 bytes of memory to a node. This is because malloc may allocate memory in specific intervals therefore rounding 52 up to 56. Also the extra space "may be used to store metadata about the allocated block" [5].

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);
}

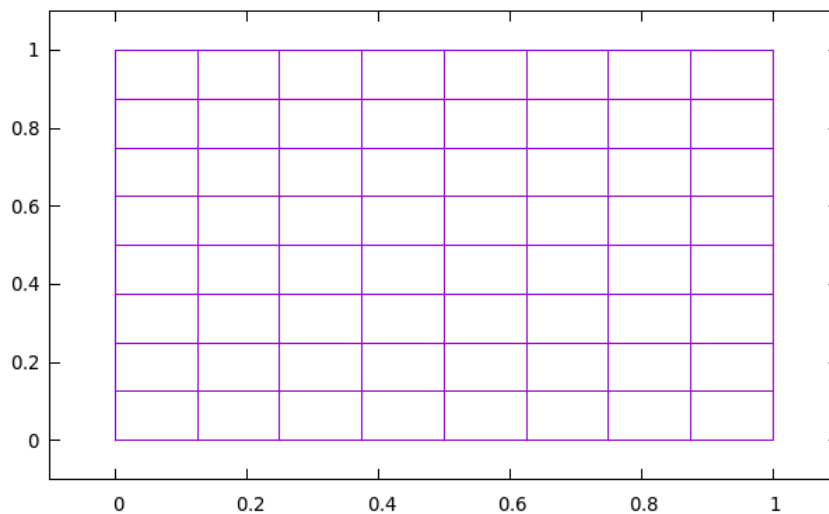
```

Test when max tree level is 3

Expected result:

An 8x8 grid.

Actual result:



Valgrind report:

```

luke@luke: ~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921
File Edit View Search Terminal Help
gnuplot> exit
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$ val
grind --leak-check=yes out/main 3
==6669== Memcheck, a memory error detector
==6669== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6669== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6669== Command: out/main 3
==6669==
running test 3...
Growing tree
Growing tree
Growing tree
Growing tree
Destroying tree
==6669==
==6669== HEAP SUMMARY:
==6669==   in use at exit: 0 bytes in 0 blocks
==6669==   total heap usage: 88 allocs, 88 frees, 10,432 bytes allocated
==6669==
==6669== All heap blocks were freed -- no leaks are possible
==6669==
==6669== For counts of detected and suppressed errors, rerun with: -v
==6669== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$

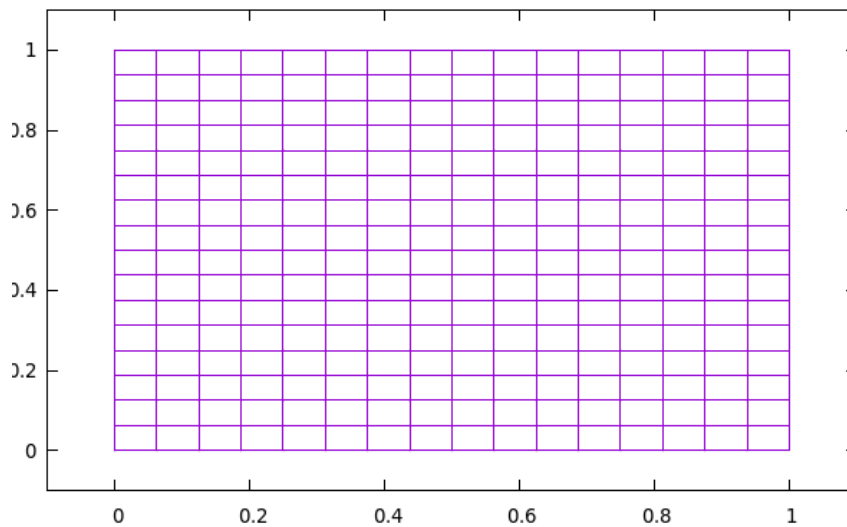
```


Test when max level is 4

Expected result:

A 16x16 grid.

Actual result:



Valgrind report:

```
luke@luke: ~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921
File Edit View Search Terminal Help
gnuplot> exit
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$ val
grind --leak-check=yes out/main 3
==6740== Memcheck, a memory error detector
==6740== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6740== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6740== Command: out/main 3
==6740==
running test 3...
Growing tree
Growing tree
Growing tree
Growing tree
Destroying tree
==6740==
==6740== HEAP SUMMARY:
==6740==    in use at exit: 0 bytes in 0 blocks
==6740==   total heap usage: 344 allocs, 344 frees, 24,768 bytes allocated
==6740==
==6740== All heap blocks were freed -- no leaks are possible
==6740==
==6740== For counts of detected and suppressed errors, rerun with: -v
==6740== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
luke@luke:~/Desktop/Computer_Science/Yr1 Sem 2/Programming Project/comp1921$
```

Task 4

New functions

```
// Visit leaf nodes and determine whether child nodes need adding
void visitLeafNodes(Node *node, double tolerance, int choice, int
*falseResults) {
    if (node->child[0] == NULL) {
        // Indicator returns false...
        if (!indicator(node, tolerance, choice)) {
            (*falseResults)++;
            makeChildren(node);
        }
    } else {
        for (int i = 0; i < 4; ++i)
            visitLeafNodes(node->child[i], tolerance, choice,
falseResults);
    }
}

// Grow the tree using the predefined functions
void growTreeUsingData(Node *head, double tolerance, int choice) {
    // keep track of how many false results there are
    int falseResults;

    do {
        falseResults = 0;
        visitLeafNodes(head, tolerance, choice, &falseResults);
    } while (falseResults != 0);
}
```

Test 1

When choice = 0, and tolerance = 0.5

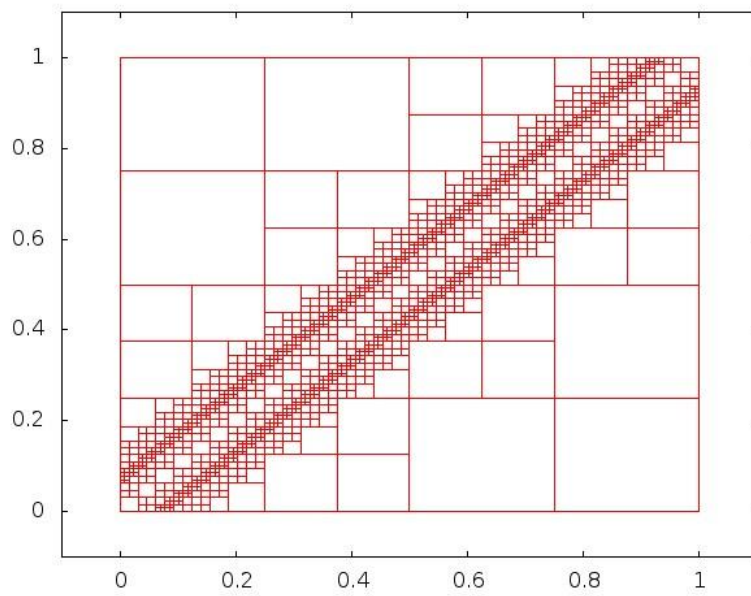
```
void task4() {
    Node *head;
    head = makeNode(0.0, 0.0, 0);

    makeChildren(head);
    growTree(head);

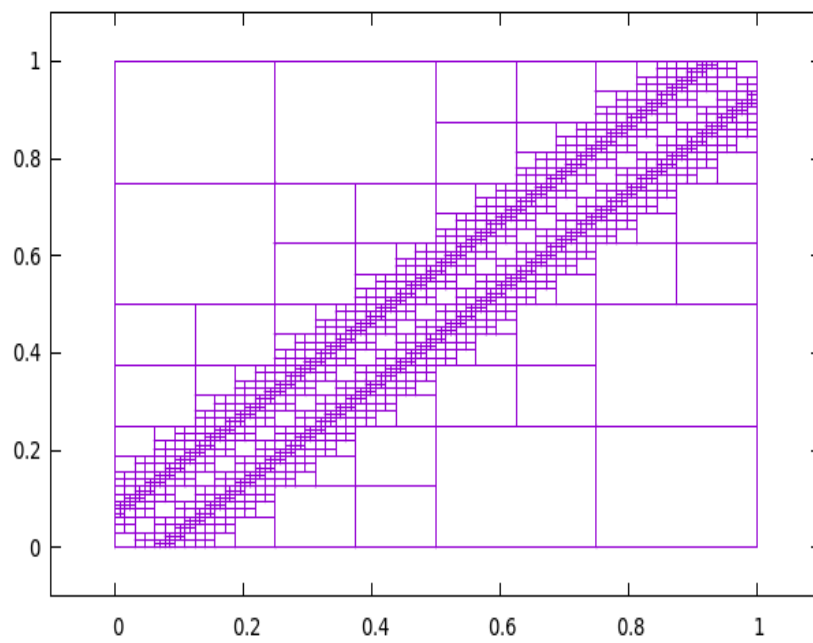
    growTreeUsingData(head, 0.5, 0);

    writeTree(head);
    destroyTree(head);
}
```

Expected result:



Actual result:

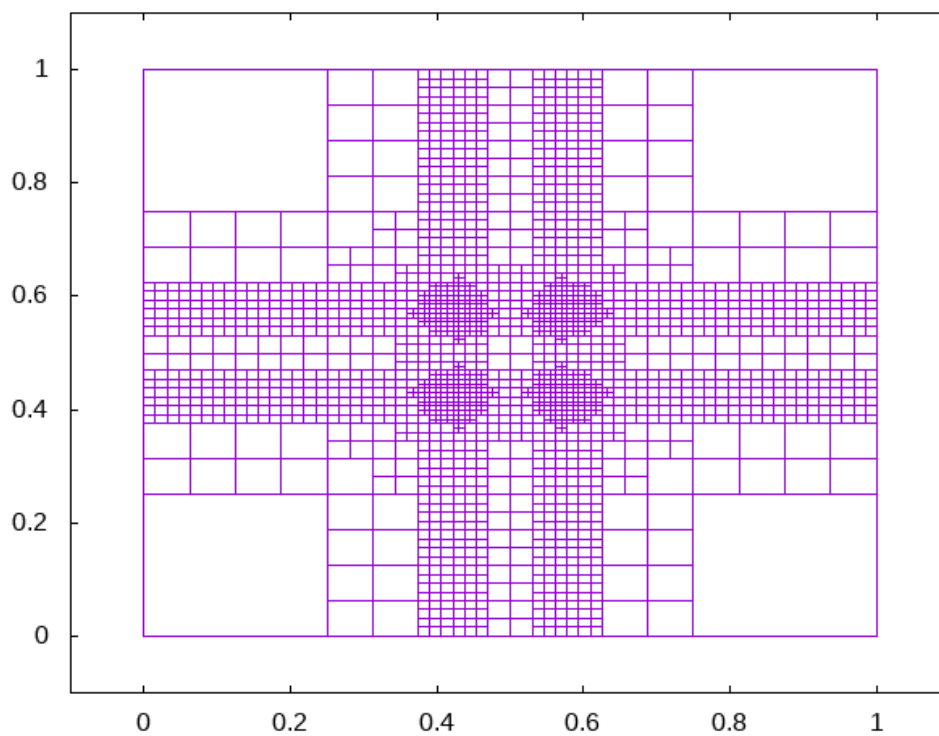


Test 2

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?

After the initial work of setting up all the files to compile correctly, the tasks were fairly straight forwards to complete. Most involved defining a function to perform an operation on the graph and didn't require that much programming or background knowledge to get working.

Other than using valgrind, testing was also simple. The expected result is often really clear so checking to make sure the program has produced the correct output went well. Creating the test functions in a separate '.c' file made the overall structure of the program better as it keeps the tests separate from the main source code. I structured the main() function so that when the program is run from the command line, it allows you to select which test to run. For example typing: ./out/main 3 will run task 3.

What was the hardest part of this work?

The hardest part of this project was getting the make file to compile all of the source files and link them together. This was because it uses syntax that is hard to understand and not very intuitive. It was especially hard to get the make file to compile the header files when I placed them in a separate directory compared to the '.c' files. After research online ^[1], I was eventually able to get the makefile working correctly. Once this was complete it was very easy to compile code and work on the project.

Another hard part of the project was getting valgrind to work on a mac. Whilst osx is built on top of UNIX, valgrind is a linux program and doesn't work very well on osx. It gives false positive results, claiming that memory had been leaked by the program when it hasn't ^[2]. I eventually chose to install a virtual machine running Ubuntu to overcome this problem.

References

- [1] <http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>
- [2] <https://stackoverflow.com/questions/5074308/valgrind-memcheck-reports-false-positive>
- [3] <https://www.geeksforgeeks.org/data-types-in-c/>
- [4] <https://www.safaribooksonline.com/library/view/understanding-and-using/9781449344535/pointer-size-and-types.html>
- [5] <https://stackoverflow.com/questions/14923157/c-malloc-seems-to-allocate-more-then-im-requesting-array>