First project - extending the Quadtree: Report

Introduction: Modularising the tree

Even before I could start the first task, the pre-programmed quadtree that would form the core of this project had to be broken into modules. While the concept of breaking down a code into separate modules is not foreign to me- despite having completed a module in procedural programming, I had yet to do it in C.

This task took a few attempts, between compiler issues and fundamental misunderstandings when it came to the nature and relationships of .c and .h files. In my first attempt I messed up the purpose of header files, it was after some research due to a warning on the compiler that I found out about header guards and that the nature of the header files was to abstract the functions and protect them from outside users. Finally, after many attempts (the latter of which performed during task 1) I have finally modularised the code in a correct manner.

Task 1: Destroy the tree

For the first task I had to create a new function that would destroy the tree, the instructions also included that the function had to be recursive. While it was clear that I would have to free each node to destroy the tree, I had to take the time to understand how the tree worked before I could start writing a that manipulated it, this I tackled this issue by taking the original program and commenting what every line did.

With the program finally understood, I could start to make a modification of the program that recursively goes through and writes the tree. Of course the destroyTree() function has to be different as it needs to visit each node in post order to work. After some initial refinement (due to a logical error), the final code is as follows:

```c
#include "treeStructure.h"

void destroyTree(Node *node ) {

  int i;
  if( node->child[0] != NULL ){
    for ( i=3; i >-1; --i ){
      destroyTree(node->child[i] );
    }
    free( node);
  }
  else {
      free( node);
    }
  return;
}
```

1. The function itself takes the head node of the tree that you want to destroy.
2. It checks to see if the current node has children.
3. If they do, it calls the itself of each the of four children.
4. If it comes across a node that has no children it will deleted it by freeing the memory.
5. Finally it will free the original node.

Test results:

```
void task1() {

  int part=0;
  int destroy=0;

  Node *test;

  // make the head node
  test = makeNode( 0.0,0.0, 0 );

  printf("This is the first test, please pick part 1 or part 2 accordingly\n");
  scanf("%i", &part);

  if (part == 1){
    // make full tree at Level 2
    makeChildren( test );
    makeChildren( test->child[0] );
    makeChildren( test->child[1] );
    makeChildren( test->child[2] );
    makeChildren( test->child[3] );

  }

  else if (part == 2){
    //make non-uniform Level 3 tree
    makeChildren( test );
    makeChildren( test->child[0] );
    makeChildren( test->child[2] );
    makeChildren( test->child[0]->child[3]);
    makeChildren( test->child[2]->child[1]);
  }

  else
    printf("You did not pick a valid option\n");

  if (part == 1 || part ==2 ){
    printf("If you would like to destory the tree, enter 1\n");
    scanf("%i", &destroy);
    if (destroy == 1){
      destroyTree( test );
      printf("You have destoryed the tree!\n");
      printf("Valgrind is expected to show no memeory leaks\n");
    }
    else{
      printf("You did not destory the tree.\n");
      printf("Valgrind is expected to show memeory leaks\n");
    }
  }
  return;
```

This test function has a degree of automation, allowing me to test with valgrind without having to edit existing code. This allows me to enter each of the test case in the terminal and contains a line stating the expected output.

First test: The first test takes a uniform level 2 tree and destroys it. Firstly, if we run the test without destroying the tree we should get memory leaks, running the test shows this to be true:

```
This is the first test, please pick part 1 or part 2 accordingly
1
If you would like to destory the tree, enter 1
2
You did not destory the tree.
Valgrind is expected to show memeory leaks
==14911==
==14911== HEAP SUMMARY:
==14911==     in use at exit: 1,176 bytes in 21 blocks
==14911==   total heap usage: 35 allocs, 14 frees, 2,472 bytes allocated
==14911==
==14911== LEAK SUMMARY:
==14911==    definitely lost: 56 bytes in 1 blocks
==14911==    indirectly lost: 1,120 bytes in 20 blocks
==14911==      possibly lost: 0 bytes in 0 blocks
==14911==    still reachable: 0 bytes in 0 blocks
==14911==         suppressed: 0 bytes in 0 blocks
==14911== Rerun with --leak-check=full to see details of leaked memory
==14911==
==14911== For counts of detected and suppressed errors, rerun with: -v
==14911== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[ll16kdt@comp-pc6135 src]$
```

As we can see, if the tree isn't destroyed we have memory leaks, now if we choose to destroy the tree:

```
This is the first test, please pick part 1 or part 2 accordingly
1
If you would like to destory the tree, enter 1
1
You have destoryed the tree!
Valgrind is expected to show no memeory leaks
==15068==
==15068== HEAP SUMMARY:
==15068==     in use at exit: 0 bytes in 0 blocks
==15068==   total heap usage: 35 allocs, 35 frees, 2,472 bytes allocated
==15068==
==15068== All heap blocks were freed -- no leaks are possible
==15068==
==15068== For counts of detected and suppressed errors, rerun with: -v
==15068== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[ll16kdt@comp-pc6135 src]$
```

We can see that there are no memory leaks, part i) of the test works as intended.

Moving on to test 1, part ii), this time we run the same tests, but with a non-uniform level 3 tree. The tree I have created is as follows:

```
makeChildren( test );
makeChildren( test->child[0] );
makeChildren( test->child[2] );
makeChildren( test->child[0]->child[3]);
makeChildren( test->child[2]->child[1]);
```

Now if we run the same tests as before, if we don't destroy the tree we should get leaks:

```
This is the first test, please pick part 1 or part 2 accordingly
2
If you would like to destory the tree, enter 1
2
You did not destory the tree.
Valgrind is expected to show memeory leaks
==15469==
==15469== HEAP SUMMARY:
==15469==     in use at exit: 1,176 bytes in 21 blocks
==15469==   total heap usage: 35 allocs, 14 frees, 2,472 bytes allocated
==15469==
==15469== LEAK SUMMARY:
==15469==    definitely lost: 56 bytes in 1 blocks
==15469==    indirectly lost: 1,120 bytes in 20 blocks
==15469==      possibly lost: 0 bytes in 0 blocks
==15469==    still reachable: 0 bytes in 0 blocks
==15469==         suppressed: 0 bytes in 0 blocks
==15469== Rerun with --leak-check=full to see details of leaked memory
==15469==
==15469== For counts of detected and suppressed errors, rerun with: -v
==15469== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Once again, without destroying the tree the program will show leaks, now if we try again but this time choose to destroy the tree:

```
This is the first test, please pick part 1 or part 2 accordingly
2
If you would like to destory the tree, enter 1
1
You have destoryed the tree!
Valgrind is expected to show no memeory leaks
==15609==
==15609== HEAP SUMMARY:
==15609==     in use at exit: 0 bytes in 0 blocks
==15609==   total heap usage: 35 allocs, 35 frees, 2,472 bytes allocated
==15609==
==15609== All heap blocks were freed -- no leaks are possible
==15609==
==15609== For counts of detected and suppressed errors, rerun with: -v
==15609== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

No memory leaks exist, just as expected. For both cases in both parts of the test we get the expected results, therefore it is safe to assume that the destroyTree() function works as intended.

Task 2: Growing the Quadtree

Moving onto the next task, it is somewhat similar to the first, we need another recursive function that transverses the tree to modify it. This time we want to check each node in preorder and if the node has no children, call the makeChildren() function to it.

I've already gone through the quadtree in detail, there already exists a recursive function that uses the preorder traversal- the writeNode() function. This function looks for nodes with no children and writes them to a file, this means that to grow the tree by one level and every spot, all I'd need to do

is replace the part of the function that write the tree to the gnu file and replace it with the makeChildren() function. This lead the growTree() function looking like so:

```c
#include "treeStructure.h"
#include "buildTree.h"

void growTree(Node *node ) {

  int i;

  if( node->child[0] == NULL )
    makeChildren( node );
  else {
    for ( i=0; i<4; ++i ) {
      growTree(node->child[i] );
    }
  }
  return;
}
```

1. The function itself takes the head node of the tree that you want to grow.
2. It checks to see if the current node has no children.
3. If no children are found, it will make a create a child on that node.
4. If the node does have children, it will call itself onto each of the child nodes, checking to see if they have no children.
5. Unlike the destroyTree() function, due to there only being on instance of makeChildren, it will not loop endlessly.

It is also worth noting that unless I want to do every test in order whenever I wanted to test the program, it became necessary to implement a selection process in my main function, allowing me to pick one of the four test cases whenever I want, without having to run the others. For this I used a simple switch case selection as it allows me to add the next cases in very easily. It may also be a wise idea to implement a loop with my test cases, allowing me to execute more than one test at a time, but at the moment due to the need to use valgrind, I'd have to do further research to justify that functionality.

Test results:

The test is similar to the first test, it takes input to select each test part, i) or ii), and then the option to grow it or not. Each option has a sentence stating what each selection does, and each case will end by destroying the tree after it has been written in an external file, freeing the allocated memory.

Moving onto the tests themselves, each one is based on a visual comparison of the trees. For each test we will use the Gnuplot before growing the tree and use that to predict the results of output after growTree() has been used, followed by using the function to receive the actual output.

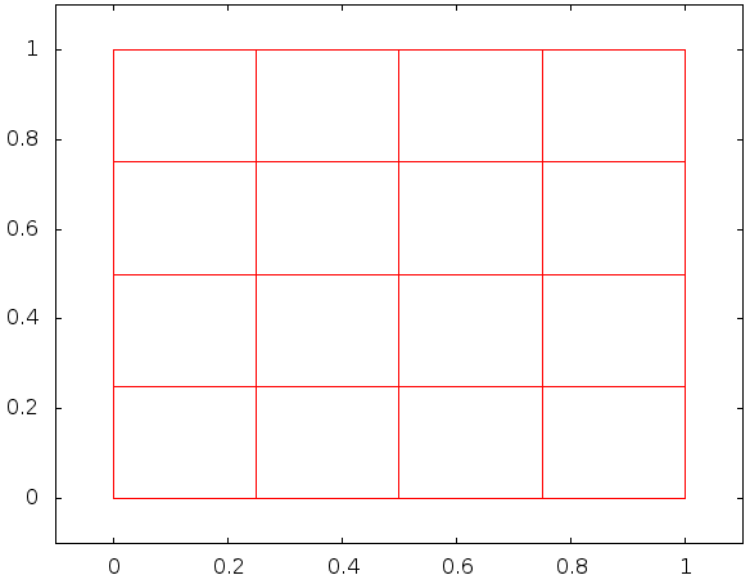
First test: For the first test we are using a uniform Level 2 quadtree, this would give us an image with 16 equal sized rectangles. With this in mind lets run the first case of the first test:

```
[ll16kdt@comp-pc6132 src]$ ./main
Please enter a test:
2
You have choosen the second test
This is the second test, please pick part 1 or part 2 accordingly
1
A full Level 2 tree has been created!
If you would like to grow the tree, enter 1
2
You have choosen not to grow the tree.
The tree should be identical to before.
[ll16kdt@comp-pc6132 src]$
```

He we can see the test selection process in action, this saves me having to modify code to run different tests, and only run the tests I want to do. With the test completed, we now use the Gnuplot from the terminal to produce a visual representation of our tree using the following commands to save it in a .png format :

```
[ll16kdt@comp-pc6132 src]$ ./main
Please enter a test:
2
You have choosen the second test
This is the second test, please pick part 1 or part 2 accordingly
1
A full Level 2 tree has been created!
If you would like to grow the tree, enter 1
2
You have choosen not to grow the tree.
The tree should be identical to before.
[ll16kdt@comp-pc6132 src]$
```

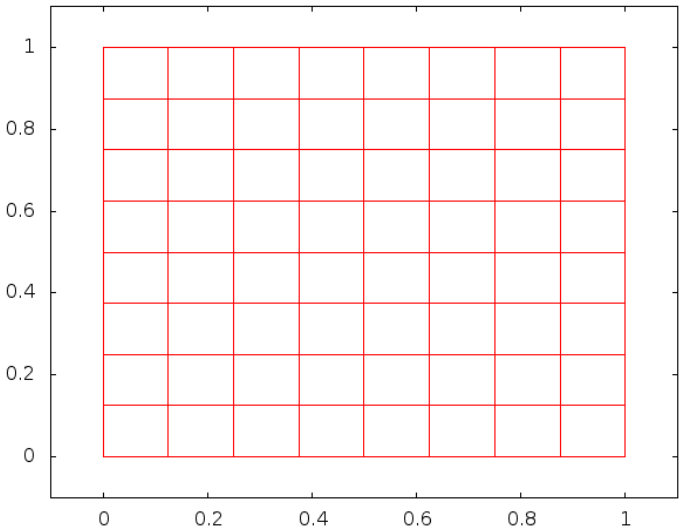
This gives us the following:



As we can see, we have a grind with 16 rectangles, each of equal size. This perfectly matches the prediction from before.

Now for the second part of this case, we need to grow the tree evenly by one level, if each child will get four new children, than c=each of those rectangles will be broken-up into four smaller rectangles, leading us to 4*16 rectangles, therefore I predict that the quadtree after it has been grown will consist of 64 equally sized rectangles.

Skipping straight to the visual result of the test as the process is very straightforward we get the following output:



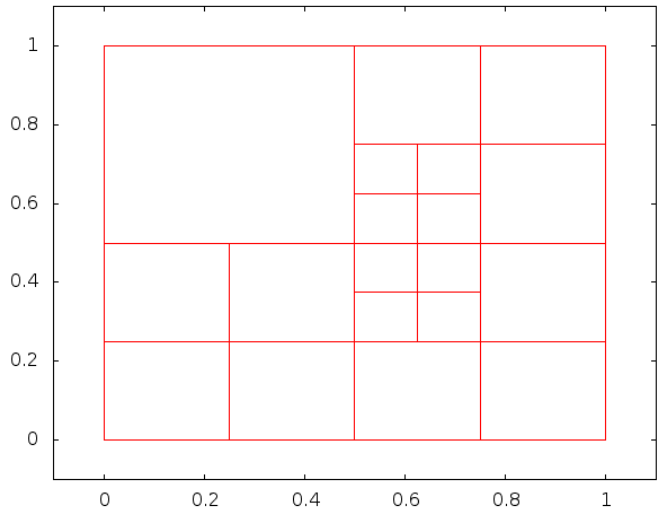
Here we can see that each of the previous rectangles have been split into four children, leading to 64 equally sized rectangles, just like predicted. The first test has been completed with no errors.

Test 2: The second test is follows the same order as the first, but with a non-uniform Level 3 tree structure that is not full. Lets look at the tree I have created for this case:

```
else if (part == 2){
  //make non-uniform Level 3 tree
  makeChildren( test );
  makeChildren( test->child[0] );
  makeChildren( test->child[1] );
  makeChildren( test->child[2] );
  makeChildren( test->child[1]->child[3]);
  makeChildren( test->child[2]->child[0]);
  printf("A non-uniform Level 3 tree has been created!\n");
}
```

The head has children, so there'll be four big quadrants, every child of head but 3 has children, so of these four quarters, one will be empty. The child 1 has children on node 3 and the child node 2 has children on node 0, this will mean two of the smaller rectangles will have one section that is further split up. Therefore there will be 1+4+3+3+4+4= 19 undivided rectangles

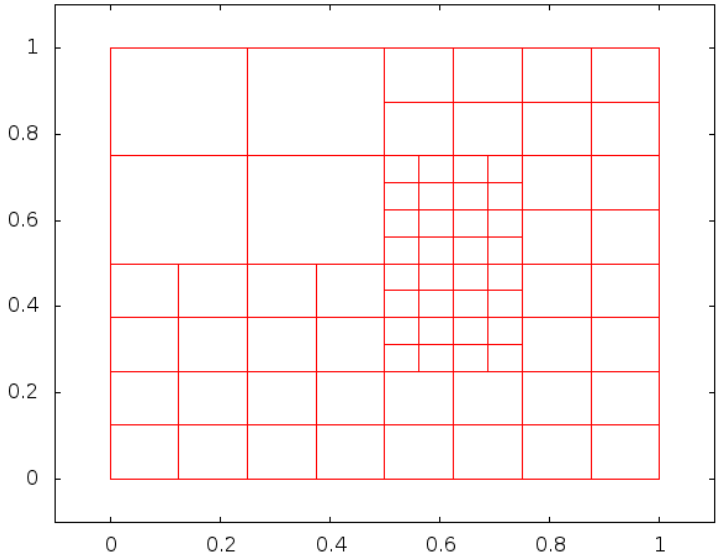
Running the test produces the following .png:



Here we see 19 rectangles of varying size, showing that the tree is non-uniform. Its characteristics match those that I just described, showing that the tree was created as intended.

If we grow the tree, each existing rectangle will be split into 4, meaning that there will be 76 rectangles after the tree has been grown.

Running the second test and growing the tree produces the following images:



This has produced the 76 rectangles as expected, therefore the growTree() function has worked under the second test conditions. The second test is now complete with everything working as expected.

Test 3: The third test wants to see if there are any memory leaks in the program. As whenever I use a function that modifies a tree, I write the end results of the modification and then I use the destroyTree() function to free the allocated memory. This case is true for my main function and my test functions.

For this test I'll run valgrind with three different cases, test 2 not ran, the first test is ran and the tree is grown and the second test is ran and the tree is grown. In each test I expect to see the line:

"All heap blocks were freed -- no leaks are possible"

Showing that all memory is freed after program execution.

First case:

```
Please enter a test:
3
==30995==
==30995== HEAP SUMMARY:
==30995==     in use at exit: 0 bytes in 0 blocks
==30995==   total heap usage: 54 allocs, 54 frees, 3,536 bytes allocated
==30995==
==30995== All heap blocks were freed -- no leaks are possible
==30995==
==30995== For counts of detected and suppressed errors, rerun with: -v
==30995== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[ll16kdt@comp-pc6132 src]$
```

3 is not a valid case and therefore all the tests have been skipped and we can see that all memory has been freed due to the line we were looking for: All heap blocks were freed -- no leaks are possible.

Second case:

```
Please enter a test:
2
You have choosen the second test
This is the second test, please pick part 1 or part 2 accordingly
1
A full Level 2 tree has been created!
If you would like to grow the tree, enter 1
2
You have choosen not to grow the tree.
The tree should be identical to before.
==30451==
==30451== HEAP SUMMARY:
==30451==     in use at exit: 0 bytes in 0 blocks
==30451==   total heap usage: 76 allocs, 76 frees, 5,280 bytes allocated
==30451==
==30451== All heap blocks were freed -- no leaks are possible
==30451==
==30451== For counts of detected and suppressed errors, rerun with: -v
==30451== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

We can see that I have chosen the second test and the uniform tree. After growing it we still get the

line we are looking for, showing that all memory has been freed.

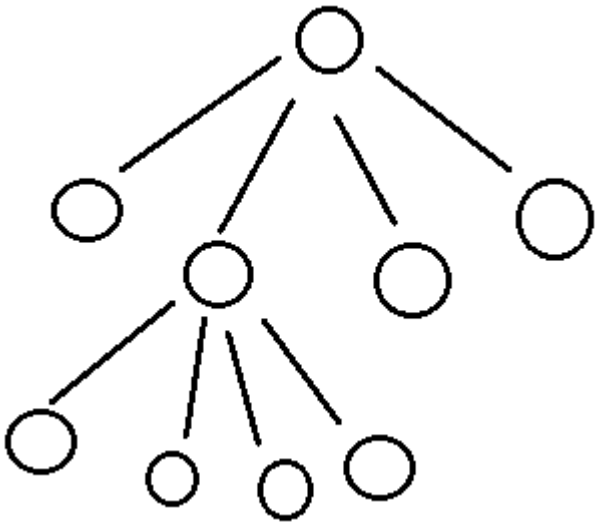
Third case:

```
Please enter a test:
2
You have choosen the second test
This is the second test, please pick part 1 or part 2 accordingly
2
A non-uniform Level 3 tree has been created!
If you would like to grow the tree, enter 1
1
You have grown the tree!
The tree should be one level deeper than before.
==31387==
==31387== HEAP SUMMARY:
==31387==     in use at exit: 0 bytes in 0 blocks
==31387==   total heap usage: 156 allocs, 156 frees, 9,760 bytes allocated
==31387==
==31387== All heap blocks were freed -- no leaks are possible
==31387==
==31387== For counts of detected and suppressed errors, rerun with: -v
==31387== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

With the third and final case once again showing that all heap blocks have been freed, we can see that the third test has been passed with the predicted results, as the final function in all lines of code is to destroy the tree, freeing memory. With this task 2 has is complete.

Task 3: A limit on tree level

The first four parts of task 3 seems to not involve coding at all, but is rather a research and mathematical exercise. Starting with part i), the first step is to create a basic quadtree that will serve as a model for the start of my calculations:



Here we have my very basic quadtree, it's a non-uniform Level 2 tree that consists of 9 nodes (that is if you include the head). Part i) will now consist of finding out how much memory this tree takes.

My approach to this problem will be to first calculate how much memory it takes to represent each node. So the first step is to look at what a node is in this program.

```c
struct qnode {
  int level;
  double xy[2];
  struct qnode *child[4];
};
typedef struct qnode Node;
```

Here we can see that a node in the program is a created type that exists as a structure consists of:

- 1 integer type
- 1 static array array that takes two double values
- a pointer that holds the address to a static array that holds four other nodes.

The first step is to find out the size of an int type. Rather than researching the internet for the answer, as the amount of memory assigned to values depends on the processor, it would be wise to step up a small test to find the size of an int on this machine:

sizeof.c

```c
#include "stdio.h"
#include "stddef.h"


int main( int argc, char **argv )
{
  printf("Starting the test!\n");

  char c;
  printf("%zu,%zu\n", sizeof c, sizeof (int));


  return 0;
}
```

The sizeof operator generates the size of a variable or data type using base units of char. This means that I can find out how much data each variable takes, but will only have to find out the exact memory that a char variable takes. Running the program gives us the following:

```
[ll16kdt@comp-pc6049 extra_stuff]$ gcc -o test sizeof.c
[ll16kdt@comp-pc6049 extra_stuff]$ ./test
Starting the test!
1,4
```

This shows that on the machines in the lab, a single int type takes up four times the memory of a character type.

Moving onto the next type, the static array, using sizeof again, we get this test:

```c
#include <stdio.h>

double xy[2];

int main(int argc, char** argv)
{
    printf("sizeof array == %zu\n", sizeof ( xy[2]));
    return 0;
}
```

The results of this test shows that the array takes the size of 8 char types.

Finally we the pointer to the static array that holds the four child nodes. This part is complicated, as

it holds nodes, the data type we're trying to find the memory allocation for. At this point it becomes very useful to know that we can use size of on whole structures, leading to the ability to calculate all the basic data types all at once, leading to this compound test:

sizeof.c

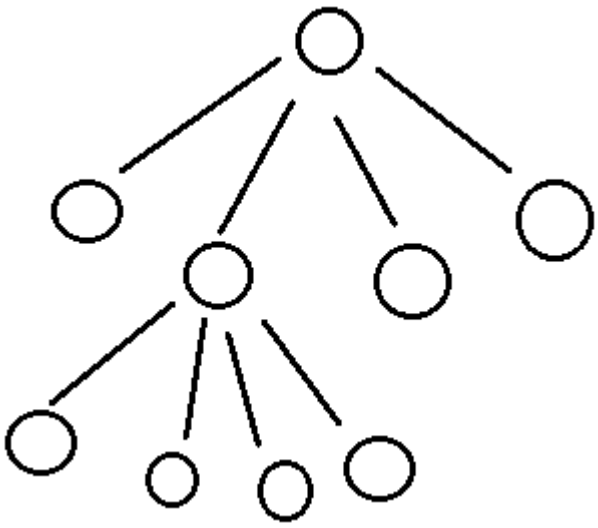
```c
#include <stdio.h>

struct qnode {
  int level;
  double xy[2];
  struct qnode *child[4];
};

int main(int argc, char** argv)
{
    printf("sizeof (struct flexarray) == %zu\n", sizeof (struct qnode));
    return 0;
}
```

This test will fully work out the memory allocated to each node in terms of a char type, this gives us a value of:

```
[ll16kdt@comp-pc6049 extra_stuff]$ ./test
sizeof (struct qnode) == 56
```

so now we know that each node takes up 56 chars worth of memory, therefore the next step is to find out how much memory a char takes on the DEC-10 machines.

After researching for an answer, it seems that for most modern machines, a char type is one byte. This means that each node is 448 bits. This now allows us to work out the amount of memory used in the quadtree from before:



This tree has 9 nodes, meaning that it would take 9*448 bits, or 4032 bits in memory. This falls under the assumption that even if the node has the children, it still uses the same amount of memory as a node with children, and that all the memory in the program is assigned to the nodes.

Moving on to part ii), I now use this data to make assumptions of bigger trees. Seeing that I use the

number of nodes of each tree to predict the memory, it would be wise to figure out a general rules to calculate the nodes of a full tree:

Level 0: 1 node
Level 1: 5 nodes
Level 3: 21 nodes-

It seems that the rule is that each full level increases the number of nodes by a factor of 4. This is a simple calculation, although the number of nodes will rapidly increase per level.

Number of Levels	Number of Nodes	Predicted Memory (bytes)	Actual Memory
5	341	19096	
6	1365	76440	
7	5461	305816	
8	21845	1223320	
9	87381	4893336	
10	349525	19573400	

The next step is part 3, I'll have to modify my main program to produce uniform trees of increasing levels.

```
//allows me to grow the tree without editing the code
int loop =0;
int loopCounter= 1;

printf("If you would like to grow the tree, enter 1\n");
scanf("%i", &loop);

while (loop==1){
  growTree( head );
  loopCounter++;
  printf("The tree is now at level %i\n", loopCounter);
  printf("If you would like to grow the tree, enter 1\n");
  scanf("%i", &loop);
}
```

This block of code now lets me choose to grow my tree, two other modifications to my code has been made: the tree before this point is now only a level one tree, and the destroyTree() function has been commented out to allow valgrind to work.

This now gives me everything I need to finish part iii)

Number of Levels	Number of Nodes	Predicted Memory (bytes)	Actual Memory (bytes)
5	341	19096	77008
6	1365	76440	306384
7	5461	305816	1223888

8	21845	1223320	4893904
9	87381	4893336	N/A
10	349525	19573400	N/A

After doing some mathematical comparisons, my values are consistently five times smaller than the values given by valgrind. While I predicted that each node would take 56 bytes of memory, if we do some average ratios between the valgrind value and the number of nodes, you get a an average of 300 bytesper node. As for explanations of the of the differences in the values, only two ideas really come to mind.

The first would be that the program assigns memory to just the nodes. My reasoning for this is that all memory was just assigned to nodes, than when I worked out the ratio for valgrind, I would have received the exact same value when I divided the memory allocation give by valgrind by the number of nodes. The other reason for this might that there is a difference in memory allocation to nodes that have children. The fuel for this idea that is for the smallest tree, the Level 5 one, it would have the least nodes that don't have children and has the highest memory allocation ratio, with the level 8 tree having the least.

The second idea is that the structure contained more than just the pointer and actually contained the array with the 4 other nodes, and this wasn't represented in the sizeof test. This makes sense as if this was the case, for each node I'd only be representing one set of memory, rather than five. This matches closely with the amount of data I predicted, it's consistently 25% of the actual value, meaning that if 5% of the data is spent elsewhere (like in the writeTree() function), then we would have only a fifth of the memory that was required.

With my two ideas of the differences explored, it's time to move on the part iv). If we wanted to limit the memory use of the application to 20Mb, than we'd have to find the level that first breaches that allocated amount and than set the limit to the level below that.

Firstly, 1 Mb is equal to 10^6 bytes. 10^6 means 1,000,000, therefore we have 20,000,00 bytes.

Seeing that valgrind already gives us memory in bytes, there is now reason to convert into bits if we use that to calculated it, but if we use our average ratio of 2395bits per node we can work out an estimate increase the leaks are too great to run on valgrind (as the delay was increasing greatly with every increase of level).

There are 8 bits in a byte, therefore one node has the ratio of 2395/8= 299.375 bytes.

20,000,000/299.375= 66806 nodes (rounded up on the nearest node).

From this we can already see that our limit would be level 9, as somewhere between a full level 9 tree and a full level 10 tree our value of 66806 nodes is breached.

Now when running valgrind we get the following values from the two latter trees:

Level 9: 19,573,968 bytes;

Level 10: 78,294,244 bytes;

Here we can see that our estimate was correct, therefore the maximum tree level to keep the memory use below 20Mb is level 9.

Part v) tasks us with implementing a cap on the based on our maximum level from the last part. That means that we need to put a condition on a function that grows the tree. The growTree() function itself is not deep enough to place this limit, as if I placed it there, I would still be able to create a level 10 tree at the start using the makeChildren() function. This means that I'd need to put the condition in the makeChildren() function itself.

```
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));

  if (level < 10){
    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 );
  }4

  else{
    printf("You have reached the limit!\n");

  }
  return;
}
```

This is the limiting condition I added to the function, it should be noted that

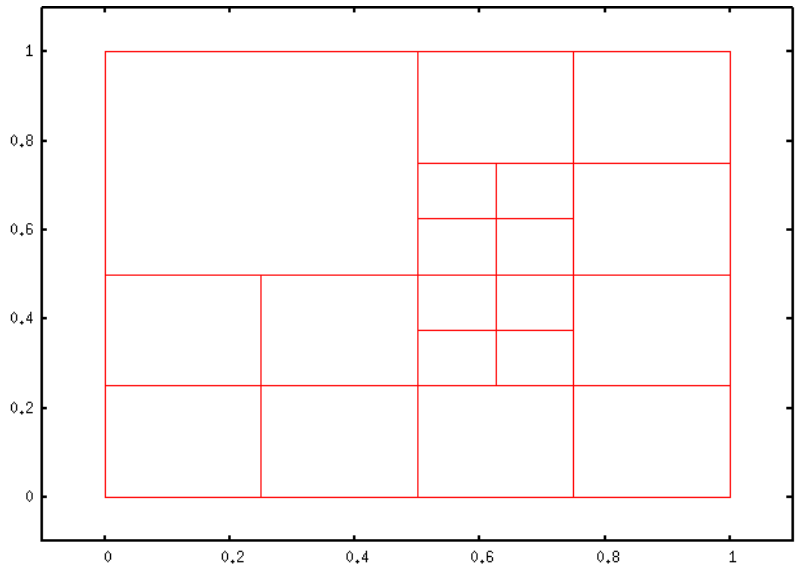
a) The else condition needs to be removed, as it'll print it out thousands of times,

b) This can't be tested visually, as Gnuplot shows a Level 9 and a Level 10 tree both as solid red blocks.

Test results:

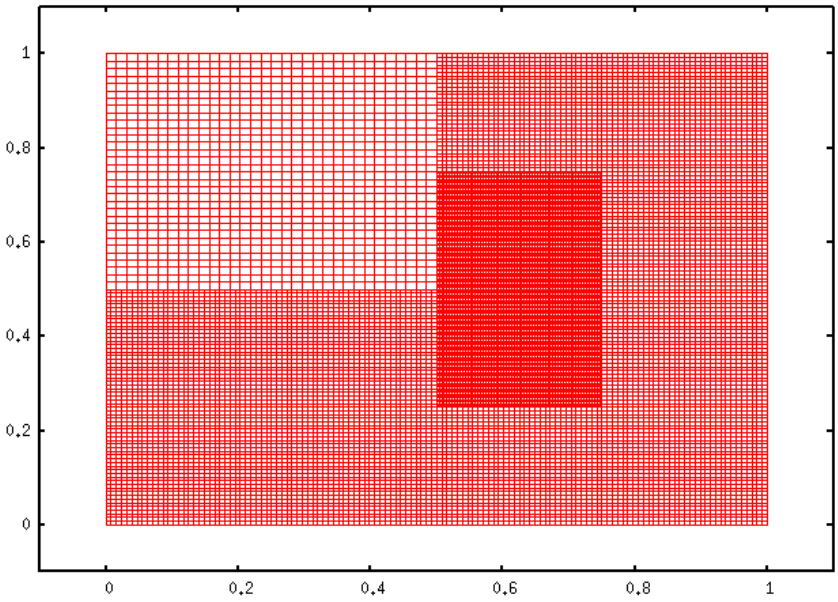
Much to my dismay the testing for this part cannot be done without editing the code itself. To combat this this I have implemented a “soft lock” function to allow me to limit the growTree() function from within the main and test, although due to it's macro approach it is limited in its use, but al we've already set a hard lock on level 9, this is just extra functionality.

Using the same tree as before, we will use visual results with Gnuplot again to confirm the results of this test. To remind ourselves of the starting tree:



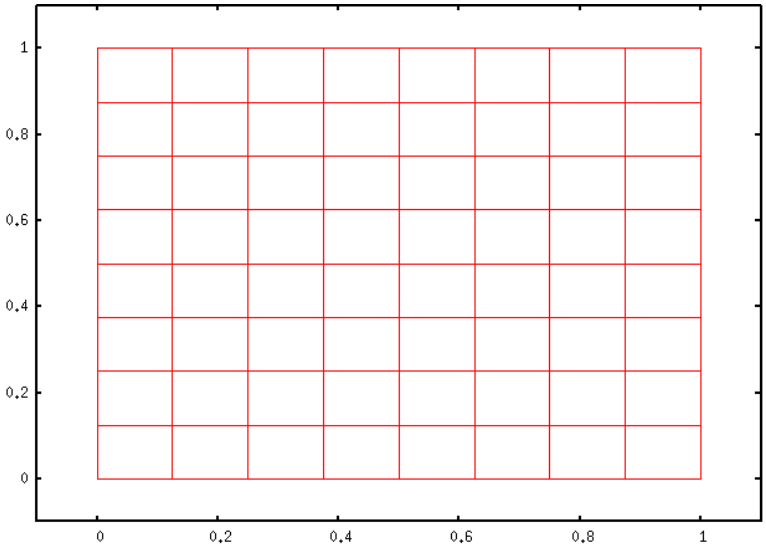
For the first test, I'll edit the maximum level in the makeChildren() to 3, and then try growing the tree five times. To start with I won't change the maximum level cap, meaning that the results with should get will produce rectangles smaller than the existing ones with have originally, compared to after the level 3 cap where the level shouldn't increase at the cap, meaning that two smallest divided rectangles won't be divided any more. Additionally, as I'm trying to increase the level by 5, we'll end with a uniform Level 3 graph, symbolised by all the rectangles being equal.

Starting with no limit:



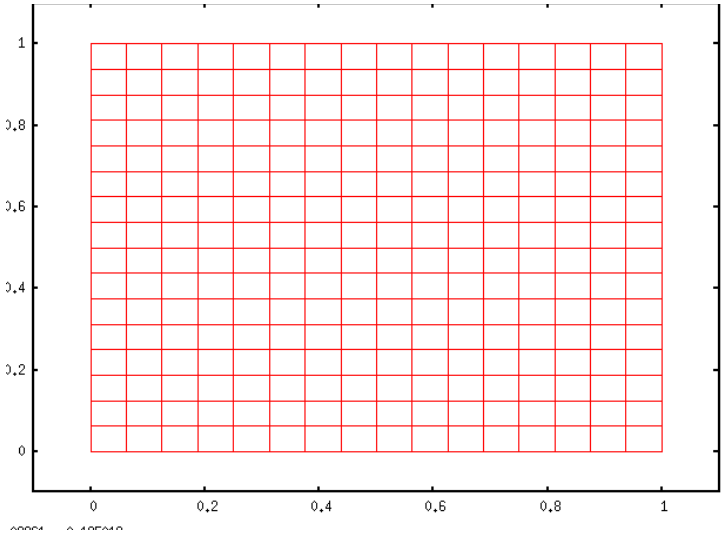
As we can see, with no additional limit, the quadtree has gone to level 8 at the lowest level.

Now if we change the limit from 9 and 3 and try the test again:



Here we see a uniform Level 3 tree, this shows that the level 3 limit is working as intended, as the tree has been limited, but execution was not stopped.

The second part of the test would be to to change the limit to Level 4 and run the same condition as before. This should lead to a tree with one more level than the one above. That will be symbolised by each of the rectangles above being divided into 4. Now if we run the test with the edited limit:



Here we can see that the tree visual is exactly like we predicted, meaning that the limiter is working as expected. This method of implementation comes with the benefit of stopping the the tree being initialised pass the limit, but comes with the drawback of not being able to be changed during runtime. Of course this has been combated with the maxLevel() function in the main stopping the growTree() function from growing a tree pass a wanted level, with a hard limit on Level 9.

Task 4: Generating a data-dependent Quadtree

The first step is to implement the the two new functions, This is a straightforward task as I can just download them into into my source folder. The next step was to replace the *'#include "quadtree.h"'* with the relevant modules.

From my understanding I'll have to create a function similar to growTree(), but with some changes for the two new functions. This lead to the creation of the autoGrow() function:

```c
#include "treeStructure.h"
#include "buildTree.h"
#include "valueTree.h"

void autoGrow(Node *node ) {

  int i;
  bool x;
  int choice= 0;
  double tolerance= 0.5;

  if( node->child[0] == NULL )
     x=indicator(node, tolerance, choice );
      if (x == false){
        makeChildren( node );
      }
  else {
    for ( i=0; i<4; ++i ) {
      autoGrow(node->child[i] );
    }
  }
  return;
}
```

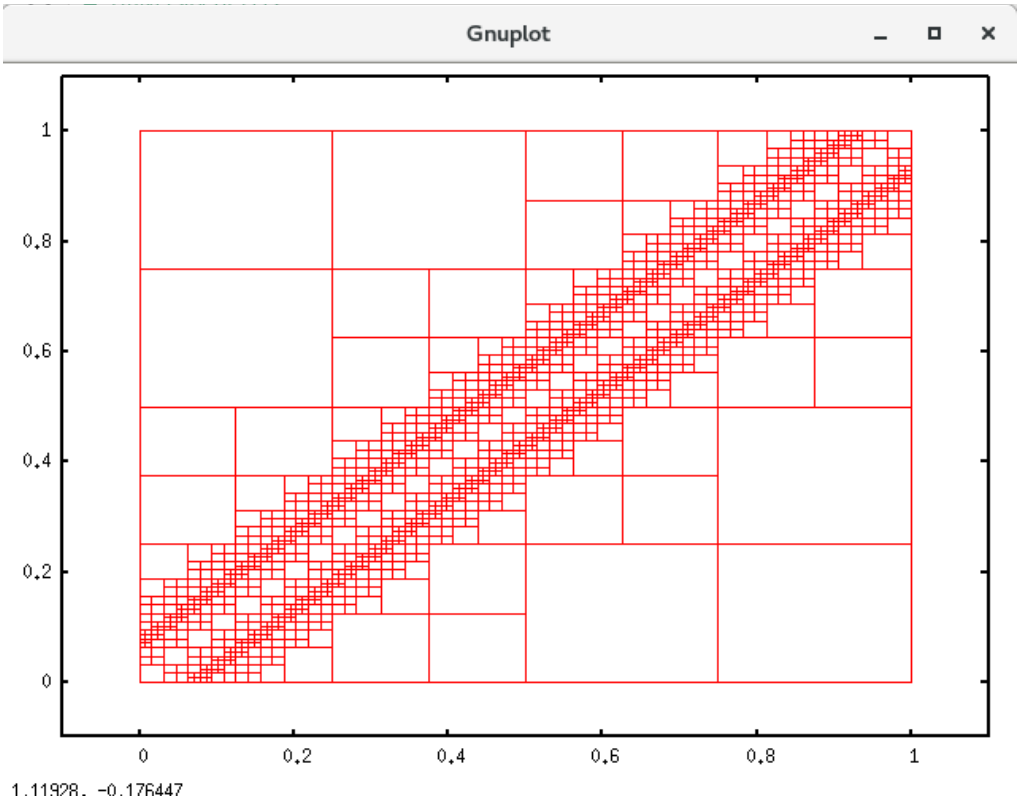
here we can see that we use skeleton of the growTree() function to recursively check every leaf node of the tree and runs the indicator() function on them (it should be noted that type bool has been defined in treeStructure.h), if the function returns false, then that node is expanded.

The next step was to implement the rule about a check for false values. The implementation of this sounded a lot like the one found in bubble sort, leading to the following code:

```
void autoGrow(Node *node ) {

  int i;
  bool x;
  int choice= 0;
  int falseCounter=1;
  double tolerance= 0.5;

  while (falseCounter> 0){
    falseCounter= 0;
  if( node->child[0] == NULL ){
    x=indicator(node, tolerance, choice );
     if (x == false){
       makeChildren( node );
       falseCounter++;
     }
  }
  else {
    for ( i=0; i<4; ++i ) {
      autoGrow(node->child[i] );
    }
  }
}
  return;
}
```

Much like the bubble sort algorithm, this function will now loop if any child nodes are created, as they will have produced a false reading from the indicator() function.

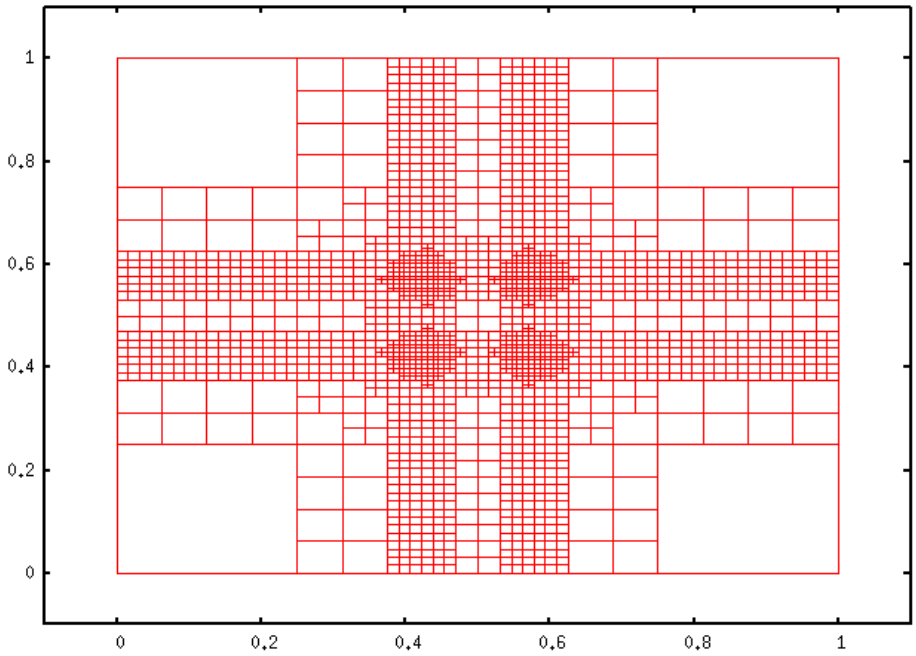
For this to run I needed to implement values, so I took the testing values and got the following output:



Leading me to believe that my autoGrow() function works as intended!

Test results:

The first part of the test was completed during development, the image above is the result and it matches quad.jpeg, an image given for comparison. The next part of the test requires me to change the choice and tolerance values of the autoGrow() function. I'll assume the test will be successful if I get a quadtree that starts with a low block density at the edges and that density increases at a chosen area, like the tree in the image above. When I run the program I get the following:



The above tree converges at 4 central points. This falls under my prediction of what the tree would look like, so I'm satisfied about the outcome of this test.

The final part of task 4 tasks me to describe the second case. The first thing that comes to mid is the symmetry of the tree, there is both a vertical and horizontal line of symmetry in this tree at 0.5 on both axis. If you look at the pattern diagonally, then you notice that the level of the tree increases as you move down the image the level of the tree increases, this happens four times, once for each corner.

Task 5: Reflection

Starting what went well, the design aspect of the project was vastly simplified, due to the ability to cannibalise existing code. The greatest example of this could would be the writeNode() function, as once I took the time to fully understand how the code works, it was the foundation for destroyTree(), virtually the same as growTree() and autoGrow() is just it modified. Certain test also had their complexity reduced due to Gnuplot, as it now because possible to use visual data as a form of black box testing rather than having to to look through the program and check the logic, if you know what the tree should look like when plotted, than you can compare visuals in tests, rather than using white box methods. This all became very evidence in task 4, where without even planning the solution, I read “a function that visits every leaf node”, and immediately knew it would be a modification of the writeNode() function, with the adding children part meaning that I would be using the growTree() function. That combined with the instruction of looping if a change has been made (a concept that is shared with bubble sort), lead to the final part of the project going very well.

On the other hand, the hardest part of the work had to be attempting to change the original tree. This was already evidenced at the start, as destroying the tree was impossible until I spent hours looking through it, something that wouldn't have been the case if I designed and created the tree myself. But when it came to things like trying to change the makeChildren() function to be able to edit the maximum level at run time, I couldn't manage it without breaking the whole program. This also happened with task 4, even with the rapid and easy implementation, all attempts to set the choice and tolerance during run time would break the program again. There was also a small difficulty when planning tests, as there is some difficulty in describing the results of Gnuplot. The easiest way would be to draw the predicted tree, but as I wouldn't be able to prove that I drew the graph and didn't just plot it first, that idea didn't feel valid, therefore I chose to use words to describe the visuals, focusing on the size and number of the rectangles produced as the result of nodes having children. Finally, my biggest problem with this project was a lack of information on the finer details. Preferably, when working on a project you'd either meet with a client or ask who every handed you the project about any details you aren't sure about. Due to the strikes this wasn't possible at the later stages, meaning that there is the possibility that I have misunderstood some of the later tasks, leading to unintended implementations producing the intended results. While on paper that may not seem as a bad thing as you still get the desired results, in a professional setting this can lead to unforeseen bugs and unoptimised code.

References:

1. Walkley, M (2018). *COMP1921 Programming Project, Lecture 4: First project: The Quadtree data structure*[Power Point slides]. Retrieved from *https://minerva.leeds.ac.uk/bbcswebdav/pid-5408666-dt-content-rid-10194327_2/courses/201718_32442_COMP1921/prog-04%281%29.pdf*
2. Walkley, M (2018). *COMP1921 Programming Project, Lecture 5: First project: The Quadtree data structure* [Power Point slides]. Retrieved from *https://minerva.leeds.ac.uk/bbcswebdav/pid-5413419-dt-content-rid-10216441_2/courses/201718_32442_COMP1921/prog-05%281%29.pdf*
3. Leonetti, M (2017). *COMP1711 Procedural Programming, Working with Larger Programs* [Power

Point slides]. Retrieved from *https://minerva.leeds.ac.uk/bbcswebdav/pid-5359928-dt-content-rid-10070688_2/courses/201718_32439_COMP1711/16-modularity.pdf*

4. GeeksforGeeks. [no date]. *Tree Traversals (Inorder, Preorder and Postorder)*. [Online] [Accessed 24 Feb. 2018]. Available at: *https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/*
5. En.wikipedia.org. (2018). *Sizeof*. [online] [Accessed 25 Feb. 2018]. Available at: https://en.wikipedia.org/wiki/Sizeof
6. MQL4 (2018). *Char, Short, Int and Long Types.* [Online] [Accessed 25 Feb. 2018] Available at: https://docs.mql4.com/basis/types/integer/integertypes .