SOUTHWEST JIAOTONG UNIVERSITY • UNIVERSITY OF LEEDS
西南交通大学－利兹学院
SWJTU-LEEDS JOINT SCHOOL

SWJTU-Leeds Joint School

Coursework Submission – Cover Sheet

Please complete ALL information

Leeds Student ID Number:201292123
SWJTU Student ID Number:2018110241
Student Name: Wu Junmou
Module Code & Name: XJCO1921 Programming Project
Title of Coursework Item: Project1 Report
For the Attention of: Xiuli Yu

Deadline Time: 12:00 pm	Deadline Date: 13th March 2020
Student Signature: 吴俊谋	**For office use:** *date stamp here*

DECLARATION of Academic Integrity

I am aware that the University defines plagiarism as presenting someone else's work, in whole or in part, as your own. Work means any intellectual output, and typically includes text, data, images, sound or performance.

I promise that in the attached submission I have not presented anyone else's work, in whole or in part, as my own and I have not colluded with others in the preparation of this work. Where I have taken advantage of the work of others, I have given full acknowledgement. I have not resubmitted my own work or part thereof without specific written permission to do so from the University staff concerned when any of this work has been or is being submitted for marks or credits even if in a different module or for a different qualification or completed prior to entry to the University. I have read and understood the University's published rules on plagiarism and also any more detailed rules specified at School or module level. I know that if I commit plagiarism I can be expelled from the University and that it is my responsibility to be aware of the University's regulations on plagiarism and their importance.

I re-confirm my consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to monitor breaches of regulations, to verify whether my work contains plagiarised material, and for quality assurance purposes.

I confirm that I have declared all mitigating circumstances that may be relevant to the assessment of this piece of work and that I wish to have taken into account. I am aware of the University's policy on mitigation and the School's procedures for the submission of statements and evidence of mitigation. I am aware of the penalties imposed for the late submission of coursework.

Report on the first project-
Extending the Quatree

Junmou Wu

Contents

1 Task1: Destroy the tree

1.1 Algorithm

In the Task 1 I was asked to create an algorithm written as **destroyTree()** for freeing all the memory used by the tree and start by freeing leaf nodes and the last node you should free is the head. Firstly, the situation was divided into two parts: whether or not the node pointed had children, thus an if-else statement. If the node's first child was null, then we only need to free its memory and assign it with null value to avoid memory leak. Otherwise, we need to go through all the nodes and use a recursive function to free from children nodes to parent, that means we need to first free all the leaf nodes using a for loop, then, the program would automatically back to parent nodes level, we could therefore free them and assigned null values as well.

1.2 Testing

The algorithm aiming to free all the memory used by the tree and start by freeing leaf nodes was tested by pictures produced from Gnuplot. For the test cases, figure1 was created by calling **growTree()** two times, and figure2 was created after calling

destroyTree(). While Figure3 was a non-uniform tree defined by calling a Level 1 tree with **growTree()** twice then use **makeChildren()** to add more nodes. The expected results were that both figures above having a blank output when calling the **destroyTree()** function because all memory were freed, and the final results were as same as expectation that figure4 and figure 2 had no remaining nodes at all. As a result, test passed.

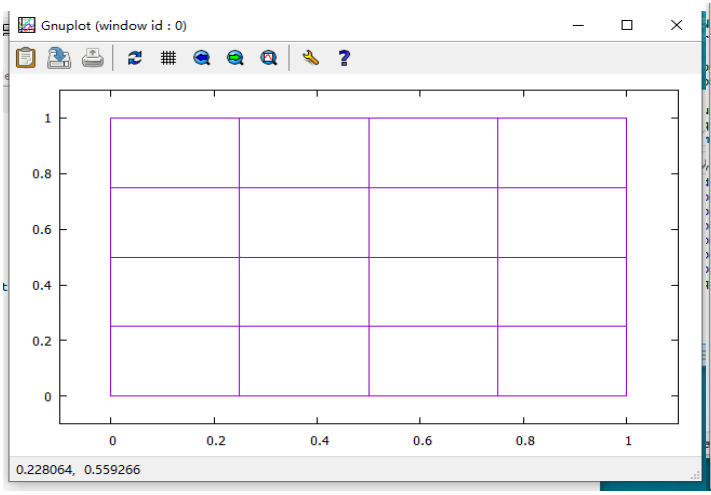

Figure1
A Full Tree at Level 2

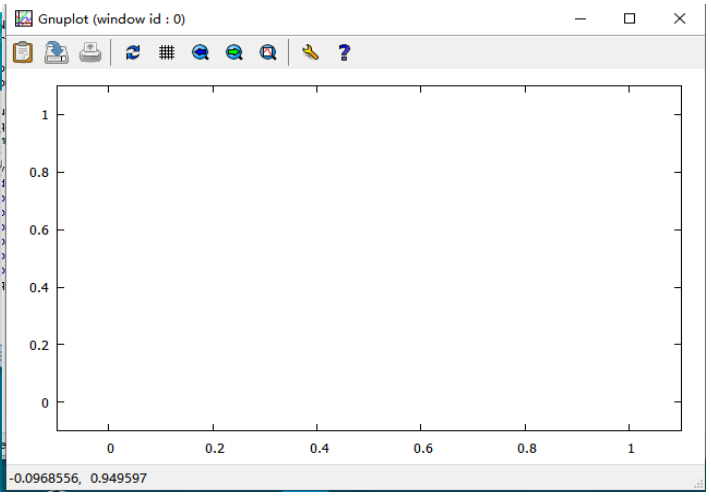

Figure2
After **destroyTree()**

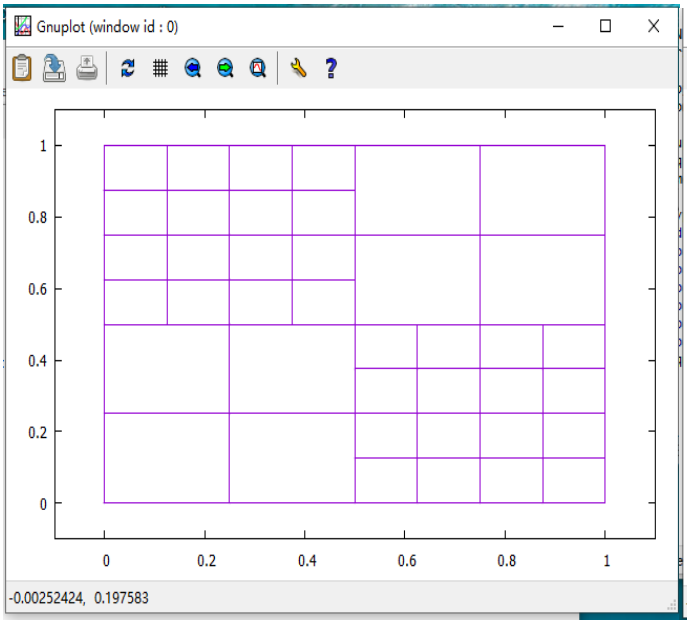

Figure3
A Non-Uniform Tree

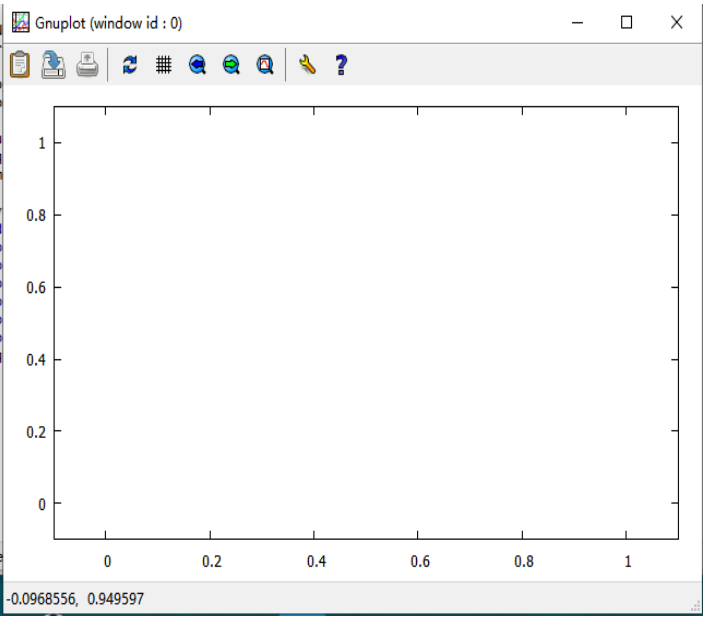

Figure4
After **destroyTree()**

2 Task2: Remove children from a node

2.1 Algorithm

For the task2 we were required to create a function that removes the 4 children from a given node and frees the memory used. Create a new function that removes children from one parent node. The situation was divided into two parts: the parent node had children and the parent node had no children. For the former, since we should assume that all children are leaf nodes, that means we just need to use a traversal algorithm to go through all leaf nodes which belonged to a particular parent node and free them, then assigned null values for every lead node. For the latter, since there were no children nodes under such a circumstance, we simply skip it.

2.2 Testing

Gnuplot was used to demonstrate the functionality of **removeChildren**() which removes the 4 children from a given node and frees the memory used. Figure5 and figure 6 were chosen as test cases. Figure5 was a full tree at Level 2 shown below, and figure6 could test the function by removing children nodes from one Level 1 node, at this stage we chose parent node[0] as parameter of the new function, the expected result was that the chosen children nodes were removed from the parent node, we could see clearly from the figure6 that the children of parent node[0] were removed. Besides, we defined a non-uniform tree as another example, which can be seen from figure7, we also expected to remove the appointed children nodes, then we used the same **removeChildren()** function to remove children from one Level 1 node. The final results can be seen easily from figure6 and figure8 that both the appointed children nodes were removed just as we expected. Therefore, test passed.

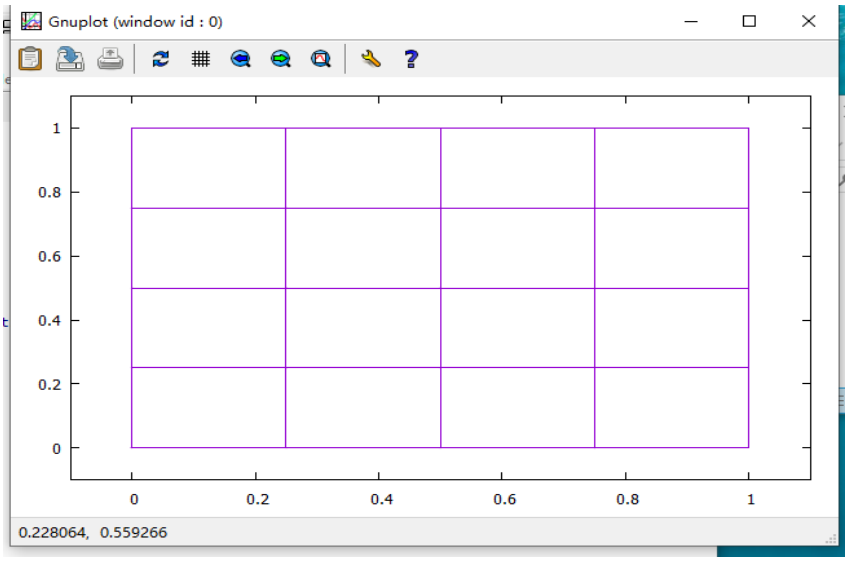

Figure5

A Full Tree at Level 2

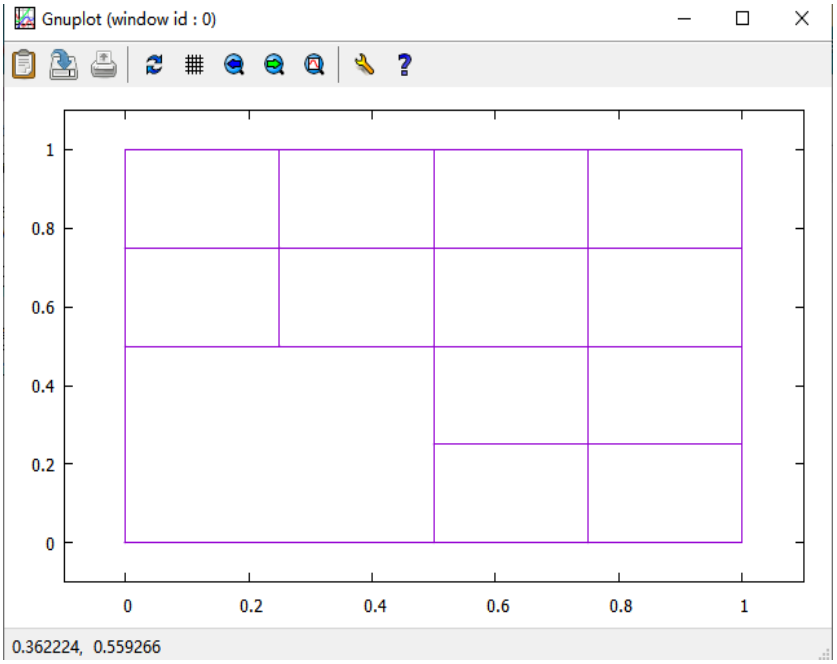

Figure6

After **removeChildren()**

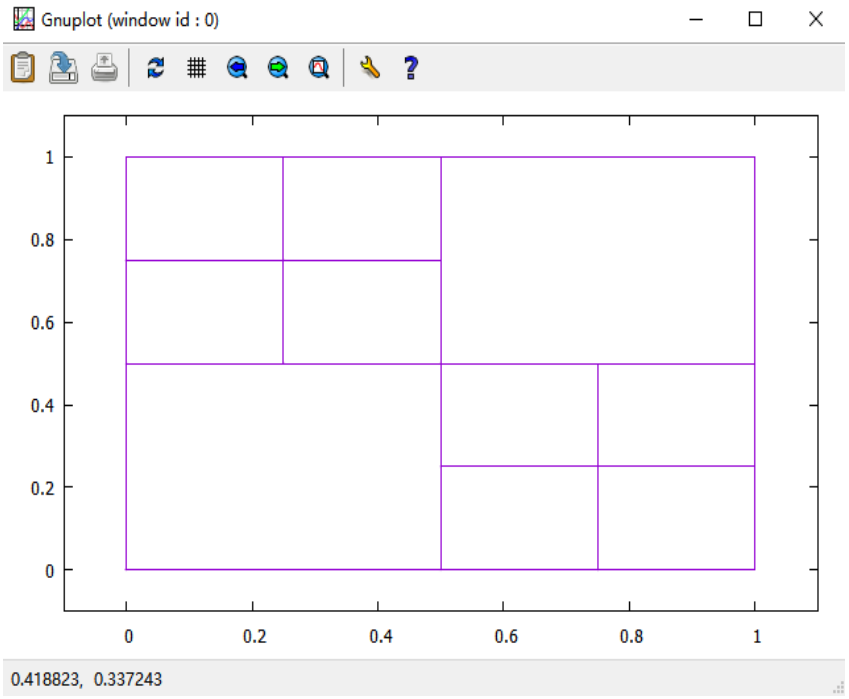

Figure7

A Non-Uniform Tree

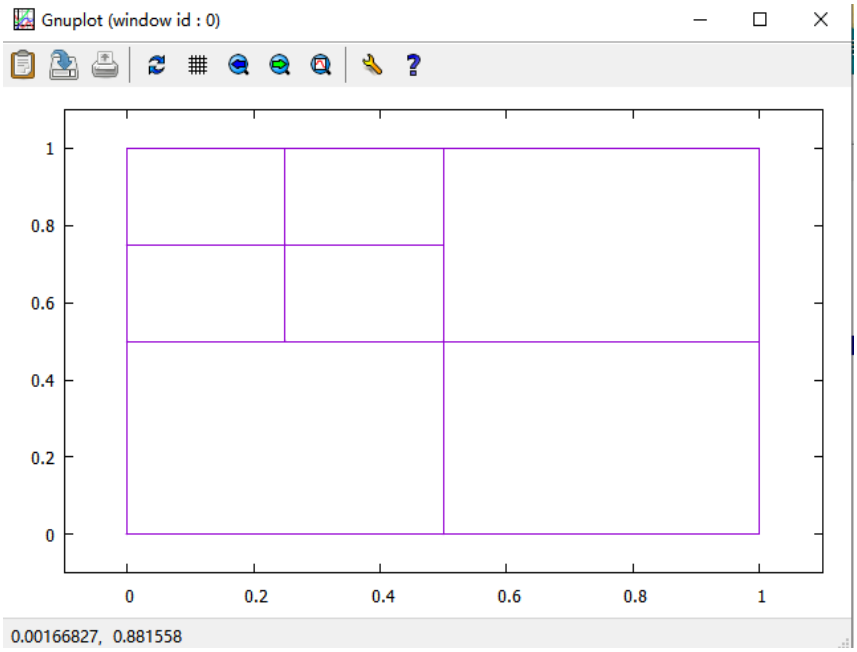

Figure8

After **removeChildren()**

3 Task3: Making a data-dependent Quadtree

3.1 Algorithm

Task3 required us to make a data-dependent quadtree, that is, to generate a quadtree that had more levels where the function is bigger than 0.5 and fewer levels where the function is less than -0.5 automatically based on a given function F(x,y) in 2D space, we had already been given two functions which defined the data on the quadtree. Thus, **makeChildren()** and **removeChildren()** functions were needed to change the tree. The algorithm was divided into two stages: first step, setting a flag to decide if you need to add or remove children, then, adding or removing nodes to change tree data structure. Therefore, for the stage1, firstly, we looked into every leaf node and set flag values according to the requirements, while all the parent nodes flags were initially set to zero because we wrote node->flag = 0 in **makeNode()** function. To achieve this, recursion algorithm was used to call function itself to enable every leaf node was set a proper flag. On the other hand, for stage2, we made two global variables, addCount and removeCount to keep count of how many nodes being added and removed. To begin with, we still used recursion algorithm to search for the required leaf nodes, when we found a leaf node with flag equals to one, **makeChildren()** was used and addCount incremented by one. After the recursion algorithm, we continued to check parent node, if the node is not a leaf node and all 4 children have flag equals to minus one, **removeChildren() was used** and removeCount incremented by one. And the tree data structure would change during each operation, thus **changeTreeStructure()** would keep calling until the tree did not change, and eventually the function would print out how many nodes added and how many nodes removed.

3.2 Testing

To test the functionality of setFlagValue() and changeTreeStructure(), we started with a fullquadtree at Level 3, as you can see from figure9 which consisted of three **growTree()** functions. After **setFlagValue()** and **changeTreeStructure(),** the final picture can be seen from figure10. Besides, from figure11, nodes added and removed at each stage were printed out, in this case, 64 nodes added and 32 nodes removed.

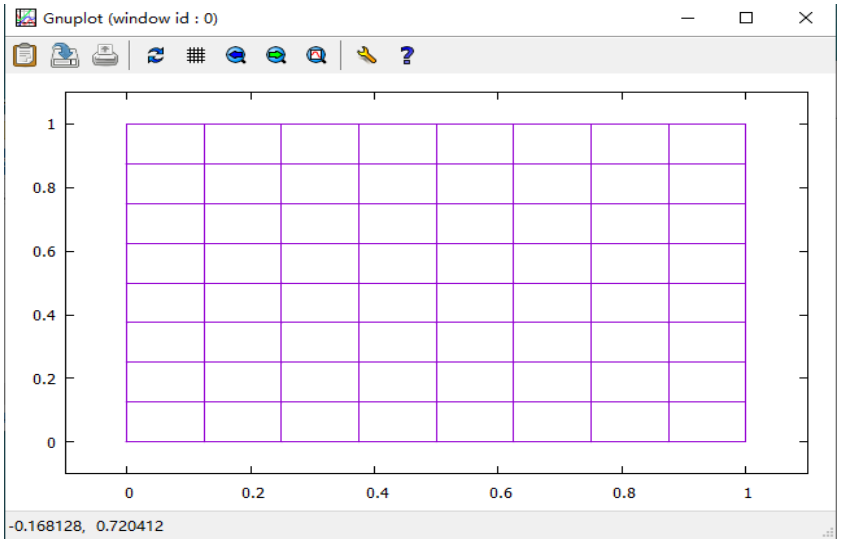

Figure9

A Full Quadtree at level 3

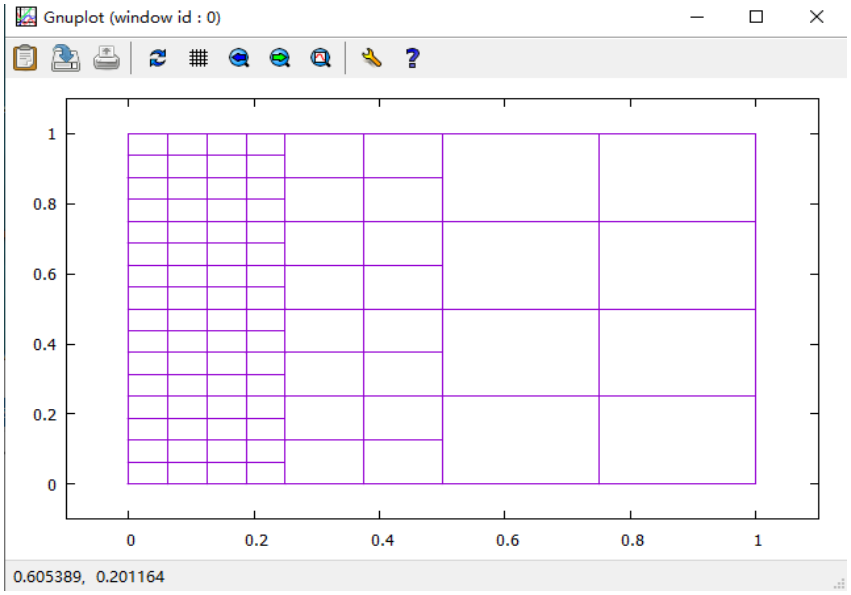

Figure10

After **setFlagValue()** and **changeTreeStructure()**

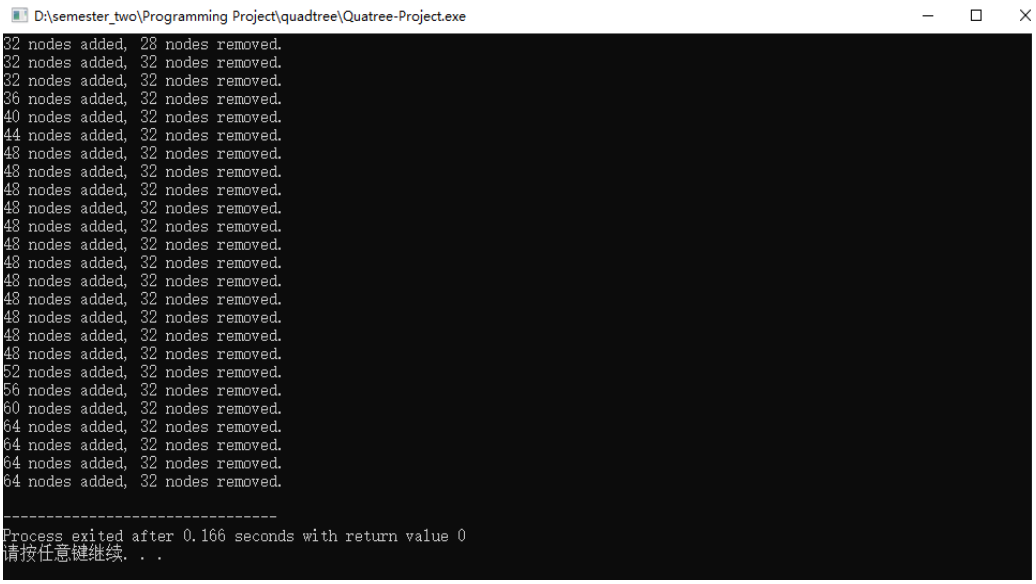

Figure 11
The Outcome of Console

3.3 Extended Algorithm

In this section we extended task3 to continue running the algorithm automatically until the tree did not change, that is, 0 nodes added and 0 nodes removed. This process was realized by a function called **adapt()**, we could consider this situation in two aspects: whether 0 nodes added and 0 nodes removed these two conditions were satisfied. Thus, we used if-else to structure the codes, on the one hand, if the two conditions above both satisfied, then we simply skip it and return. On the other hand, if only one condition satisfied or neither of them satisfied, we reset two counters to 0, with the operation of **setFlagValue()** and **changeTreeStructure(),** to show how many nodes are added or removed at each stage via recursion algorithm. We also set the global variable maxLevel to 6 since the program would keep operating without restricted level number. Moreover, global variables addCount and removeCount were used to keep on track explicitly how many nodes were added or removed. The algorithm will not stop unless 0 nodes added and 0 nodes removed, which means, the tree does not change. As a result, throughout the running of algorithm, we were looking for a state when two conditions both satisfied, nodes added or removed were recorded during the procedure of each stage.

3.4 Extended Testing

This section is aimed to test the validation of the extended algorithm for task3. Previous problem was tested by this algorithm. Staring tree was made before, a full quadtree at level3, which can be seen from figure9. And the final tree is produced below, named figure12. Besides, as you can see in figure 12, nodes added or removed at each stage are displayed, and the final stage is 0 nodes added, 0 nodes removed, shown below.

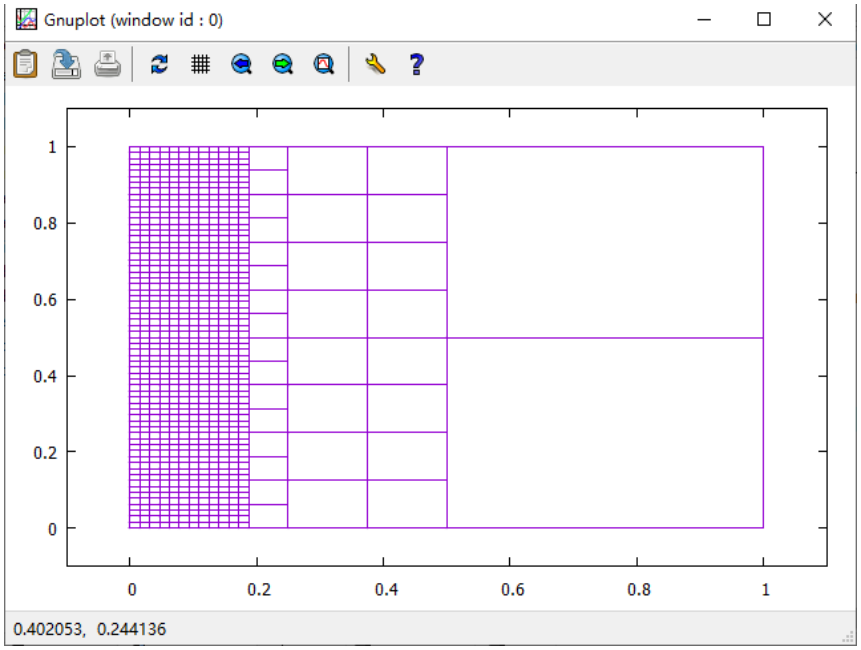

Figure12

After **adapt()**

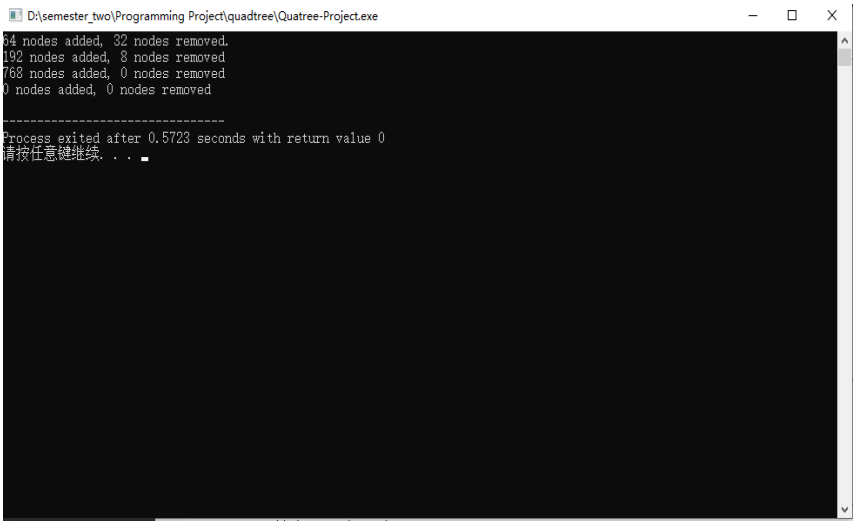

Figure13
Nodes Added or Removed at Each Stage

4. Task4: **Reflection**

4.1 What Went Well With this project?

Through the whole project, there were many aspects went quite well.

Modularity, with the help of previous exercised, I found it easy and logic to design the overall modularity of the software. Every written function was placed into an appropriate manner, **destroyTree()** and **removeChildren()** were integrated into a new file based on their generality, naming disableTree.c, similarly, **setFlagValue()**, **changeTreeStructure()** and **adapt()** were put together into changeTree.c.

Clarity, to avoid unnecessary misunderstanding of others, I named all the functions and variables based on their functionality, such as addCount and removeCount, which were used to keep track on how many nodes were added or removed. Besides, annotations were made to enable clarity and readability.

Formatting, consistent indentation and spacing were used during this project after studying how to indent and keep space.

Version Control, every version of the work was pushed to GitHub to provide back-up.

Design, initial designs of task1 and task2 worked successfully with a clear logic and precise delivery.

Testing, since Valgrind is restricted to Linux, Gnuplot is the only tool to test the result. Throuth testing, test cases were given clearly, expected results and final results were made to adjust the algorithm.

4.2 The hardest part of this work? Why? What will you do to change this in the future?

The hardest part of this work were the designing and programming of task3 and its extended part, although Minerva has relevant description of the design and algorithm, I found it hard to figure out the implementation of how to design the code, I knew the structure, but when it comes to specific function realization, I even did not know where to start. It was teacher's answers of recursion algorithm that gave me some inspirations.

The occurrence of this confused status probably because of the weak knowledge mastery and a lack of doing exercises, I just listened to the class and did nothing else after class, which is a very bad habit for programmers.

In the future, I will review and revise the things learnt from class, especially those I do not understand. Besides, as a student majoring in computer science, algorithm and data structure cannot be mastered easily, I need to learn more from practicing on computer than just reviewing in brain. Skill comes from practice, as a result, I will code a lot to improve my coding ability as well as my thinking ability.