PAs 3 and 4 Quadtrees

Due: **Friday, December 1 at 11:59 PM**

Goals and Overview

We are releasing this assignment as two dependent PAs. In the first you will implement an unfamiliar data structure, a quadtree, and in the second you'll add functionality.

Checking Out the Code

You can download to code here: `pa34.zip`. As usual, please upload the files onto the school's Linux servers and complete the assignment there.

Background Information: Quadtrees

We create a tree in the following manner. First, let us require that each node of our tree correspond to some particular region of the plane. Second, we require that the regions corresponding to the children of a node n form a partition of the region corresponding to n (i.e. no two children's regions have overlapping interiors, although overlapping boundaries are allowed), and all the children's regions put together cover the region corresponding to n exactly). Notice that given this property, for a tree T, the regions corresponding to the leaves of T form a partition of the region corresponding to the root of T (you can prove this using induction).

Now, let us construct a special tree T of the above type, as follows. First, every node of T corresponds to a square on the plane. Second, given n is a node of T (with corresponding square region s on the plane), n may have either zero or four children; if n has four children, then the regions corresponding to these four children are the four squares obtained by splitting s vertically and horizontally. We refer to these "subsquares" of s by their cardinal directions - NW (north west), NE (north east), SW (south west), and SE (south east). A tree built in this way, by selecting a single square region to corresponding to the root of T and then creating descendants according to the procedure above, is called a quadtree. See Figure 1.

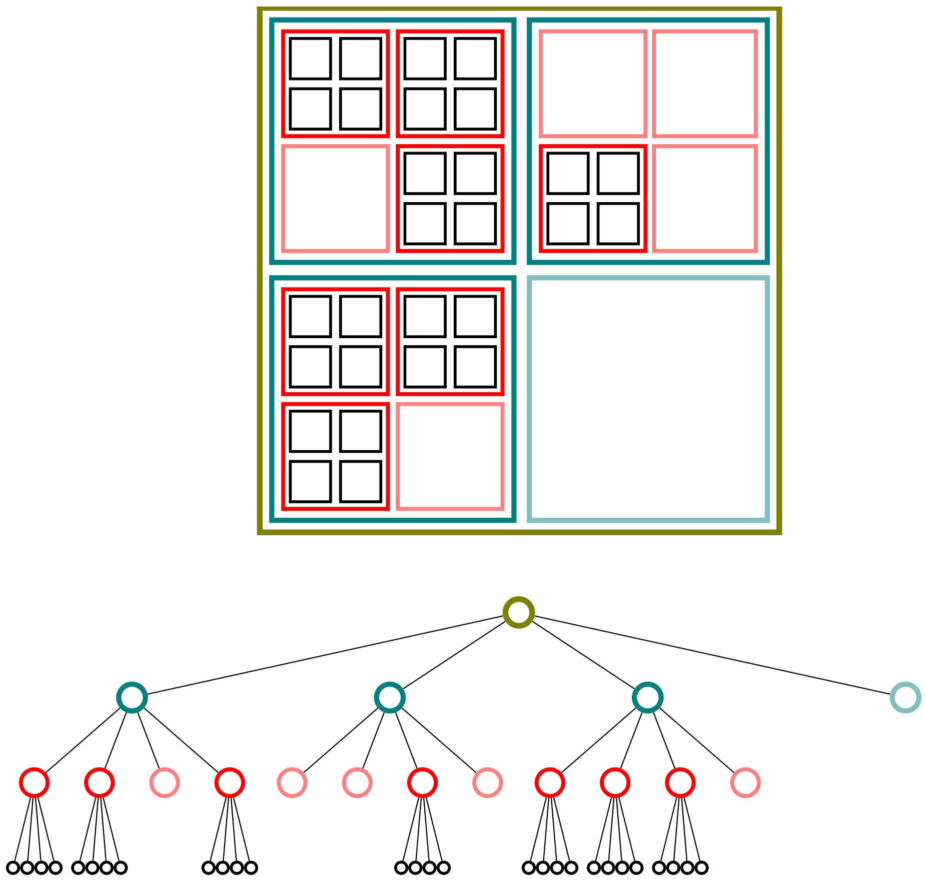


Figure 1. Quadtree, with pointers ordered NW, NE, SW, SE.

Suppose that the image is 128x128 pixels. Notice that the node at the green level of the tree corresponds to the entire 128x128 image; the nodes at the teal level of the tree correspond to the 64x64 partitions of the image; the nodes at the red level of the tree correspond to the 32x32 partitions of the image; the nodes at the black level of the tree correspond to the 16x16 partitions of the image; and so on.

In Figure 1, the 34 leaves of the tree correspond to the 34 unequal squares into which the root square is split by the diagram.

Background Information: The `Quadtree` class

A bitmap image is composed of many identically-sized square pixels arranged in a grid. If the number of pixels is 2^k by 2^k for some k, then one can build a quadtree with its root corresponding to the entire image, and with each leaf corresponding to a pixel. This provides us with an interesting way to store an image: put an `RGBApixel` into each leaf of an appropriate quadtree.

For this PA, you will write a `Quadtree` class. In order to facilitate testing of your PAs, we have already partially implemented this class for you. In particular, we have declared a `class QuadtreeNode`, which represents a single node of a quadtree. The static data stored in a `QuadtreeNode` object are one `RGBApixel` (the "element"), and four `QuadtreeNode*` variables: `nwChild`, `neChild`, `swChild`, and `seChild`. These pointers provide links from each node object to child node objects. As this scheme suggests, every

`QuadtreeNode` of a `Quadtree` object is allocated in dynamic memory, and must be managed properly to avoid memory leaks. Additionally, we have specified in the given file `quadtree.h` that each `Quadtree` object should contain a pointer to a `QuadtreeNode`, named `root`; this pointer will point to the root node of the `Quadtree`.

We have also provided a couple of member functions for you, `printTree()` and `operator==()`. These functions do just what you might expect; the former prints out the contents of the leaves of the quadtree, using a preorder traversal, while the latter compares two `Quadtree` objects to see if they are "equivalent". Unfortunately, you will probably not find these functions too useful in implementing the remainder of the PA. However, you may find them useful for debugging.

You may implement the `Quadtree` class however you wish, with the following restrictions: you must not remove or alter the given functions, and you must use the already-declared member variables in the intended manner. That is, root should always point to the root of the quadtree, and within each `QuadtreeNode` object, the pointers `nwChild`, `neChild`, `swChild`, and `seChild` should point to the appropriate child nodes. In short, you may do whatever you like, as long as you do not break `printTree()` or `operator==()`; these functions are vital for grading.

Implementation Advice

Here are some suggestions to guide your implementation. These are just suggestions; you are not obligated to adhere to them.

- Your `Quadtree` objects will probably need some way of remembering the resolution of their underlying bitmaps.
- Remember the technique shown in class and lab for implementing recursive tree functions. In particular, you will often want to have a public version of some method, which conforms to the interface specifications above, along with a private version of the same method, which takes in additional arguments (such as a pointer to a `QuadtreeNode`, for example).
- You may find it useful to create private helper methods. In general, whenever you find that you are performing the same task in multiple places, you should consider encapsulating that task into a helper method. This can save a bit of time in the coding process, and a lot of time in the debugging process.
- You may want to ensure that, at all times, the element field of each interior (i.e. non-leaf) node of your quadtree stores the average of its children's element fields. (See the description for prune.) It is much easier to calculate all of this information once, when the tree is first constructed, than to calculate it "on demand" whenever it is needed.

In addition, be mindful of the following:

- You will be responsible for testing your code before handing it in. Using our `main.cpp` and test images to check your program, while not required, is strongly recommended. You are also strongly encouraged to perform further tests; in particular, you are encouraged to modify the given `main.cpp` file as you please, or to create a new `main.cpp` file, to further test your program.
- The `Quadtree` class implementation uses dynamic memory allocation; be sure to avoid memory leaks in your program. Memory leaks will adversely affect your grade.

- With the exception of `printTree()`, your functions should print nothing to the screen when they run correctly. You are welcome to use a debugger, or another method, to track and examine the execution of your functions during the development process. Just be sure that if your final submission functions execute without errors, they print nothing; this is to ensure your program will match the solution output during grading.

Implementation Requirements

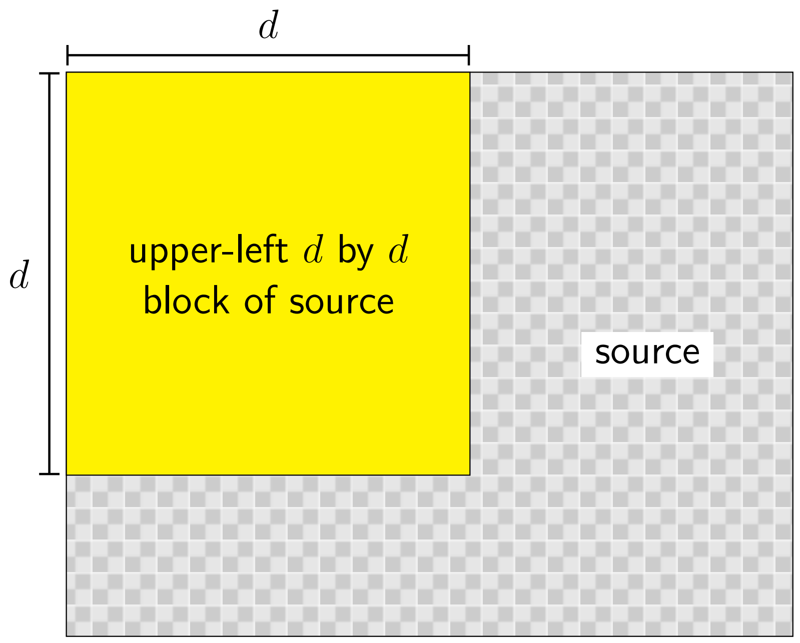
- Your functions may be graded for their asymptotic efficiency.
- You are required to comment the PA as per the commenting standard described by the Coding Style Policy.
- You must name all files, public functions, public member variables (if any exist), and executables **exactly** as we specify in this document.
- Your code must produce the **exact** output that we specify: nothing more, nothing less. Output includes standard and error output and files (such as images).
- Your code must compile on the ugrad machines using **clang++**. Being able to compile on a different machine is **not** sufficient.
- Your code must be submitted correctly by the **due date and time**. Late work is not accepted.
- Your code must not have any memory errors or leaks for full credit. ASAN tests will be performed separately from the functionality tests.
- Your public function signatures must match ours **exactly** for full credit. If using different signatures prevents compilation, you will receive a zero. Tests for `const`-correctness may be performed separately from the other tests (if applicable).

PA 3: Quadtree Basics

For PA 3, you will implement the constructors, the Big Three, `buildTree`, `getPixel`, `decompress`, and `clockwiseRotate` functions of the `Quadtree` class.

The `buildTree` Function

This function takes two arguments, a `PNG` object source, and an integer $resolution$, in that order. It returns nothing. It should delete the current contents of the `Quadtree` object, then turn it into a `Quadtree` object representing the upper-left $resolution$ by $resolution$ block of source. You may assume that $resolution$ is a power of two, and that the width and height of source are each at least $resolution$.



`Quadtree` Constructors

You will write two constructors for the `Quadtree` class:

- The first takes no arguments, and produces an empty `Quadtree` object, i.e., one which has no associated `QuadtreeNode` objects, and in which `root` is `NULL`.
- The second takes two arguments: a `PNG` object source, and an integer $resolution$, in that order. Its purpose is to build a `Quadtree` representing the upper-left $resolution$ by $resolution$ block of the source image. This effectively crops the source image into a $resolution \times resolution$ square.

You may assume that $resolution$ is a power of two, and that the width and height of source are each at least $resolution$. Perhaps, to implement this, you could leverage the functionality of another function you have written.

The Big Three

You will also write the Big Three (copy constructor, destructor, assignment operator) for the `Quadtree` class.

The `getPixel` Function

This function takes two arguments, x and y, in that order, both nonnegative integers. It returns the `RGBApixel` corresponding to the pixel at coordinates (x, y) in the image which the `Quadtree` represents. Note that the `Quadtree` may not contain a node specifically corresponding to this pixel (due, for instance, to pruning — see below). In this case, `getPixel` will retrieve the pixel (i.e. the color) of the square region within which the smaller query grid cell would lie. (That is, it will return the element of the nonexistent leaf's deepest surviving ancestor.) If the supplied coordinates fall outside of the bounds of the underlying bitmap, or if the current `Quadtree` is empty (i.e., it was created by the default constructor) then the returned

`RGBApixel` should be the one which is created by the default `RGBApixel` constructor.

The `decompress` Function

This function takes no arguments, and returns a `PNG` object. The return value should be the underlying `PNG` object represented by the `Quadtree`. If the current `Quadtree` is empty (i.e., it was created by the default constructor) then the returned `PNG` should be the one which is created by the default `PNG` constructor.

This function effectively "decompresses" the quadtree. A `Quadtree` object, in memory, may take up less space than the underlying bitmap image, but we cannot simply look at the `Quadtree` and tell what image it represents. By converting the `Quadtree` back into a `PNG` image, we lose the compression, but gain the ability to view the image directly.

The `clockwiseRotate()` Function

This function takes no arguments, and returns nothing. `clockwiseRotate` rotates the `Quadtree` object's underlying image clockwise by 90 degrees. (Note that this should be done using pointer manipulation, not by attempting to swap the element fields of `QuadtreeNode`s. Trust us; it's easier this way.)

PA 4: Quadtree Manipulations

For PA 4, you will implement four more functions in the `Quadtree` class: [`prune`], [`pruneSize`], and [`idealPrune`].

The `prune` Function

This function takes a single argument, an integer `tolerance`. It returns nothing.

If the color values of the leaves of a subquadtree don't vary by much, we might as well represent the entire subtree by, say, the average color value of those leaves. We may use this information to effectively "compress" the image, by strategically trimming the `Quadtree`.

Consider a node n and the subtree, T_n rooted at n, and let a denote the component-wise average color value of all the leaves of T_n. Component-wise average means that every internal node in the tree calculates its value by averaging its immediate children. This implies that the average must be calculated in a "bottom-up" manner. (Due to rounding errors, using the component-wise average is not equivalent to using the true average of all leaves in a subtree.) If a node n is pruned, the children of n and the subtrees for which they are the roots are removed from the `Quadtree`. Node n is pruned if the color value of no leaf in T_n, differs from a by more than `tolerance`. (Note: for all average calculations, just truncate the value to integer.)

We define the "difference" between two colors (r_1, g_1, b_1) and (r_2, g_2, b_2) to be:

$$(r_2 - r_1)^2 + (g_2 - g_1)^2 + (b_2 - b_1)^2$$

To be more complete, if the tolerance condition is met at a node n, then the block of the underlying image which n represents contains only pixels which are "nearly" the same color. For each such node n, we remove from the `Quadtree` all four children of n, and their respective subtrees (an operation we call a pruning). This means that all of the leaves that were deleted, corresponding to pixels whose colors were similar, are now replaced by a single leaf with color equal to the average color a over that square region.

The `prune` function, given a tolerance value, prunes the `Quadtree` as extensively as possible. (Note, however, that we do not want the `prune` function to do an "iterative" `prune`. It is conceivable that by pruning some mid-level node n, an ancestor p of n then becomes prunable, due to the fact that the `prune` changed the leaves descended from p. Your `prune` function should evaluate the prunability of each node based on the presence of all nodes, and then delete the subtrees based at nodes deemed prunable.)

You should start pruning from the root of the `Quadtree`.

The `pruneSize` Function

This function takes a single argument, an integer `tolerance`. It returns an integer. This function is similar to `prune`; however, it does not actually prune the `Quadtree`; rather, it returns a count of the total number of leaves the `Quadtree` would have if it were pruned as in the prune function.

The `idealPrune` Function

This function takes a single argument, a positive integer `numLeaves`. It returns an integer. This function should calculate and return the minimum `tolerance` necessary to guarantee that upon pruning the tree as above, no more than `numLeaves` leaves remain in the `Quadtree`. Essentially, this function is an inverse of `pruneSize`; for any `Quadtree` object `theTree`, and for any positive integer `tolerance`, the following inequality should hold:

```
theTree.pruneSize(theTree.idealPrune(numLeaves)) <= numLeaves
```

Once you understand what this function is supposed to do, you will probably notice that there is an "obvious" implementation. This is probably not the implementation you want to use! There is a fast way to implement this function, and a slow way; you will need to find the fast way. (If you doubt that it makes a significant difference, the tests in the given `main.cpp` should convince you.)

Hint

The "obvious" implementation involves a sort of linear search over all possible tolerances. What if you tried a binary search instead?

Testing Your Code

We have provided you with a sample `main.cpp` file, and corresponding test images. However, to grade your program, we will use different `main.cpp` files that implement different tests. Consequently, you will not submit `main.cpp`. You are strongly encouraged to write your own `main.cpp` file to more thoroughly test your program.

You can compile your code with the following command:

```
make
```

You can run it with the following command:

```
./pa3
```

As usual, an ASAN version is also produced:

```
./pa3-asan
```

The output it produces can be compared with the expected (`soln_pa3.out`) and the images created can be compared with the solution images using a `diff` utility (e.g., `diff` and `compare`).

As usual, passing these tests does not ensure correctness. You should augment these tests with your own.

Handing In Your Code

To facilitate anonymous grading, do not include any personally-identifiable information (like your name, your CWL, or your CSID) in any of your source files. Instead, before you hand in this assignment, create a file called `partners.txt` that contains only the CSIDs of people in your collaboration group (if it exists), one per line, with your name on line 1. If you worked alone, include only your own CSID in this file. We will be automatically processing this information, so do not include anything else in the file. As always, if you're working in a group, each group member must hand in the assignment.

Stay tuned for handin instructions.

Grading Information

The following files are used to grade PA3 and PA4:

- `quadtree.cpp`
- `quadtree.h`
- `partners.txt`

All other files (including your `main.cpp`) will not be used for grading.

Further Study

One can think of many ways to define distances between colors. We defined the distance between two colors simply by the distance between the colors' corresponding points in the 3-dimensional space of the colors' RGB representations. However, this distance doesn't represent the human perception of the difference between colors especially well. A search on "CIE color" provides a starting point to learn more about color schemes that endeavor to better model actual human color perception. Using a better concept of intercolor distance could yield images that, when compressed with the pruning function in this PA, look more natural; or from another perspective, images that can be compressed more yet still look as natural.

Quadtrees have uses reaching far beyond storing of images. One can store a set of points within a quadtree, and then manipulate the structure in various ways. The k-d tree is a binary variant of the quadtree in which nodes correspond to rectangles (and child and parent regions again follow a certain set of rules). Quadtrees can be generalized to three-dimensional space, where they are called octrees.

Good luck!