ECE250: Lab Project 3

Due Date: Friday, March 18, 2016 – 11:00PM

1. Project Description

In this project, you implement a Quadtree data structure. A Quadtree is similar to a binary search tree; however, rather than being sorted on just one well-ordered element, it is sorted on a pair of well-ordered elements. Each node stores two-dimensional vectors, which we will denote as (x, y), and has up to four children, one for each direction: North East (NE), South East (SE), North West (NW) and South West (SW). A child node that is placed under a parent node following these criteria:

- Given a pair (x, y) and the current pair (x_0, y_0), if $x \geq x_0$, the pair (x, y) must be stored in one of the *east* sub-trees; otherwise the pair (x, y) must be stored in one of the *west* sub-trees.
- Given a pair (x, y) and the current pair (x_0, y_0), if $y \geq y_0$, the pair (x, y) must be stored in one of the *north* sub-trees; otherwise the pair (x, y) must be stored in one of the *south* sub-trees.

For example, if the root node is (5, 5); pairs such as (7, 7), (5, 7), and (7, 5) must be stored in the north-east sub-tree, pairs such as (3, 7) and (3, 5) are stored in the north-west sub-tree, pairs such as (7, 3) and (5, 3) are stored in the south-east sub-tree, and pairs such as (3, 3) are stored in the south-west sub-tree. This is shown in Figure 1 where the axes are grouped according to where the values are stored.

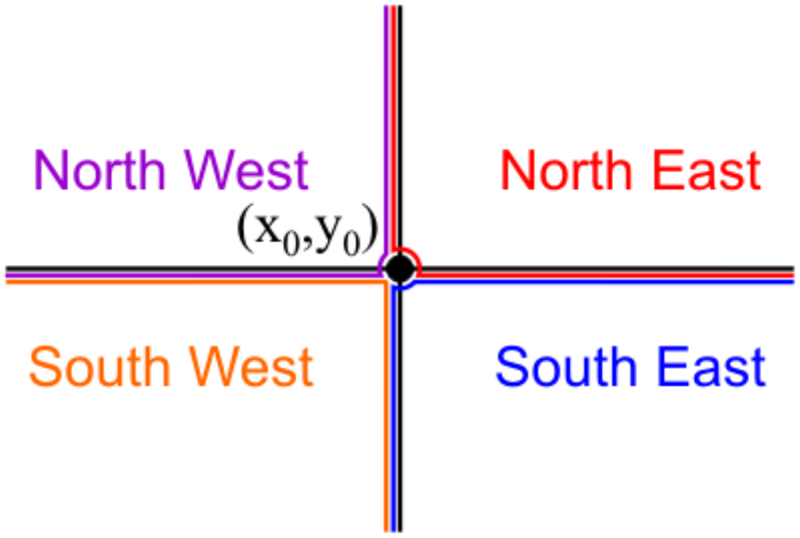

Figure 1. Two Dimensional Spaces in a Quadtree

2. Use Two Classes: Quadtree.h and Quadtree_node.h

You will implement a Quadtree using two classes, a tree class (Quadtree.h), and a node class (Quadtree_node.h). The tree class contains a root node, from where every other node in the tree can be reached. The node class, representing each node in the tree, contains a pair of values, denoted as (x,y), and pointers to four nodes.

3. Use Recursion

You will use recursive-programming techniques to perform operations on this Quadtree. For example, in order to find the sum of the *x* values under a node, you will call the function *sum_x* on the existing children of this node.

4. You will implement a C++ Template Class

You will implement the Quadtree as a C++ template class, where each node in the tree contains x and y values of variable type T. Variable type T will be replaced by a specific type such as *int*, or *double* depending on the requirements of the application using the Quadtree data structure.

5. How to Test Your Program

We use drivers and tester classes for automated marking, and provide them for you to use while you build your solution. We also provide you with basic test cases, which can serve as a sample for you to create more comprehensive test cases. You can find the testing files on the course website. You will notice that there are separate drivers to test your class template for types *int* and *double*.

6. How to Submit Your Program

Once you have completed your solution, and tested it comprehensively, you need to build a compressed file, in tar.gz format, which should contain the files:

- Quadtree.h
- Quadtree_node.h

Build your tar file using the UNIX tar command as given below:

- *tar –cvzf* ***xxxxxxxx****_p****n****.tar.gz* Quadtree.h Quadtree_node.h

where ***xxxxxxxx*** is your UW user id (ie. jsmith), and ***n*** is the project number which is 3 for this project. All characters in the file name must be lower case. Submit your tar.gz file using LEARN, in the drop box corresponding to this project.

7. Class Specifications

You will implement your solution using two classes the Quadtree.h class, and the Quadtree_node.h class. The Quadtree.h class provides access to the root of the Quadtree, so it contains a pointer to the root node. The Quadtree_node.h class stores the pair of values (x,y), and up to four pointers corresponding to the directions NE,NW,SE and SW for this node. Every node of the tree can be accessed following a path that starts at the root node.

In the specifications below, when indicating runtime requirements, ***n*** represents the number of elements in the Quadtree.

7.1 Quadtree Class

Member Variables

The *Quadtree* class has at two members variables:

- ***Quadtree_node<T>*** **tree_root* – a pointer to the root node of the tree
- *int count* - The number of elements currently in Quadtree (same as n)

__Constructor__
Quadtree( )
Set both member variables to 0.

__Destructor__
~ Quadtree ()
The destructor must delete each of the nodes in the Quadtree. (**O**(*n*))

__Accessors__
This class has ten accessors. Note that the first three member functions should be implemented entirely in the Quadtree class; the other member functions should be implemented by calling the appropriate member functions of the root node. ie. min_x() should be calculated by calling min_x() on the root node.

- *int size() const* – Returns the number of items in the Quadtree. (**O**(1))
- *Quadtree_node<T> *root() const* - Returns the address of the root node. If the tree is empty, the root node should be 0. (**O**(1))
- *bool empty() const* - Returns true if the Quadtree is empty, false otherwise. (**O**(1))
- **max and min functions** - Returns the minimum-or-maximum x-or-y value within the quadtree. Throw an underflow exception if the tree is empty. (**O**(n) but **O**(√n) if balanced)
  T min_x() const;
  T max_x() const;
  T min_y() const;
  T max_y() const;
- **sum functions** - Returns the sum of the x and y values within the Quadtree, respectively. The sum of the nodes of an empty tree is 0. (**O**(n))
  T sum_x() const;
  T sum_y() const;
- *bool member(T const &x, T const &y) const* - Returns true if the pair (x,y) is stored in one of the nodes of the Quadtree and false otherwise. (**O**(*n*) but **O**(ln(*n*)) if balanced)

__Mutators__
This class has two mutators:

- *void insert(T const &x, T const &y)* - Inserts the pair (x, y) into the Quadtree. If the root is 0, a new quadtree node is created; otherwise, the task of insertion is passed to the root node. (**O**(n) but **O**(ln(n)) if balanced)
- *void clear()* - Calls clear on the root if necessary and sets the root and count to 0. (**O**(*n*))

7.2 Quadtree_node Class

The Quadtree_node class stores the pair of values (x,y), and up to four pointers corresponding to the directions NE, NW, SE and SW for this node.

__Member Variables__

- *T x_value* – x_value is the first value in the pair (x,y) (*x_value* is of template type T)
- *T y_value* - y_value is the second value in the pair (x,y) (*y_value* is of template type T)
- Four pointers to Quadtree_node:
  *Quadtree_node *north_west;*
  *Quadtree_node *north_east;*
  *Quadtree_node *south_west;*
  *Quadtree_node *south_east;*

Constructor
Quadtree_node(T const &x, T const &y)
This constructor sets the x and y values to the arguments, respectively, and sets all the sub-trees to zero value. (**O**(1))

Destructor
~ Quadtree_node()
The destructor must call clear on the four sub-trees and delete those nodes. (**O**(n))

Accessors
This class has thirteen accessors:

- ***retrieve functions*** - Return the x and y values of the current node, respectively. (**O**(*1*))
  T retrieve_x() const;
  T retrieve_y() const;
- ***child pointer functions*** - Return the north-west, north-east, south-west, and south-east pointers, respectively. (**O**(1))
  *Quadtree_node *nw() const*
  *Quadtree_node *ne() const;*
  *Quadtree_node *sw() const;*
  *Quadtree_node *se() const;*
- ***max and min functions*** - Return the minimum-or-maximum x-or-y value within the quadtree. (**O**(n) but **O**($\sqrt{n}$) if balanced)
  T min_x() const;
  T min_y() const;
  T max_x() const;
  T max_y() const;
- ***sum functions*** - Return the sum of the x and y values within the Quadtree, respectively (**O**(n))
  T sum_x() const;
  T sum_y() const;
- *bool member(T const &x, T const &y) const* - Returns true if the pair (x,y) is stored in the current node, or in the Quadtree starting at the current node (in nodes connected to this node through its pointers). It returns false otherwise. (**O**(n) but **O**(ln(n)) if balanced)

Mutators
This class has two mutators:

- *bool insert(T const &x, T const &y)* - If the pair (x,y) equals the pair stored in the current node, return false; otherwise, insert the pair into the appropriate sub-tree either by creating a new node and returning true or recursively calling insert and returning that call's return value. (**O**(n) but **O**(ln(n)) if balanced).
- *void clear()* - On any non-zero sub-tree, call clear, and then delete that node. (**O**(n))

NOTE: the **O**($\sqrt{n}$) run times are the natural run times of a balanced Quadtree: the minimum x value is the minimum x value of the current node and the minimum x values of the two west sub trees. You do not have to implement special code to achieve the required run time.