Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2014-2015

Homework 3 (v1.2) - Region Quad-Trees

Due date: 30 April 2015, Thursday, 23:55

1 Objective

This homework aims to help you get familiar with the fundamental C++ programming concepts.

2 Problem Definition

A *region quad-tree* is a tree data structure which recursively partitions 2D-space into 4 quadrants at each level. Each node in the tree represents a rectangular region in 2D-space and it has either no child (if it is a leaf node) or 4 children (if it is an internal node).

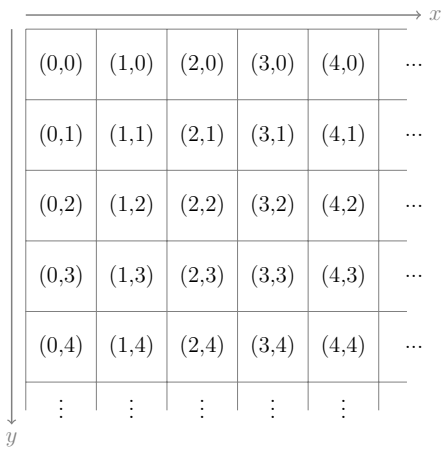

Figure 1: The 2D pixel coordinate system (x, y) used in this homework.

Region quad-trees are often used to represent a set of geometric shapes defined in 2D-space. Each node of the cells has a limited capacity of geometric shapes. While inserting a new shape to a quad-tree, if the corresponding node is full, it splits into 4 new quadrants.

In this homework, you are going to implement region quad-trees in C++ for keeping a set of rectangles defined in 2D-space.

3 Specifications

You are going to implement three C++ classes: `Coordinate`, `Rectangle` and `QTree`. For each class, we are providing you with a header file including the declarations for *public* methods and constructors. You are going to include these header files in your `.cpp` files and provide your constructor, destructor and method bodies in the `.cpp` files.

You are free to define as many *private* members, methods and constructors as you like in the header files as long as you conform with the specifications below. However, do **NOT** add, remove or modify the given *public* methods and constructors (doing so will result in severe grade deduction).

3.1 Coordinate Class

The `Coordinate` class is used for representing points in 2D-space. You are provided with the following `Coordinate.h` header file:

```
#ifndef COORDINATE_H
#define COORDINATE_H

class Coordinate{
    private:
        // add private members, methods and constructors here as you need
    public:
        // do not make any modifications below
        Coordinate();
        Coordinate(int, int);
        Coordinate(const Coordinate&);
        ~Coordinate();
        int getx() const;
        int gety() const;
        void setx(int);
        void sety(int);
        Coordinate operator+(const Coordinate&) const;
};

#endif
```

We are using the pixel coordinate system which is given in Figure 1. Descriptions for each one the public constructors and methods are given below:

Name	**Description**
`Coordinate::Coordinate()`	Empty constructor, creates a coordinate object at $(0, 0)$.
`Coordinate::Coordinate(int, int)`	Constructor with two `int` arguments. The first argument is the x-coordinate and the second one is the y-coordinate.
`Coordinate::Coordinate(const Coordinate&)`	Copy constructor.
`Coordinate::~Coordinate()`	Destructor.
`int Coordinate::getx() const`	Getter for the x-coordinate.
`int Coordinate::gety() const`	Getter for the y-coordinate.
`void Coordinate::setx(int)`	Setter for the x-coordinate.
`void Coordinate::sety(int)`	Setter for the y-coordinate.
`Coordinate Coordinate::operator+(const Coordinate&) const`	Overloaded + operator. Summation of two coordinates $(x_1, y_1) + (x_2, y_2)$ must result in a coordinate object at position $(x_1 + x_2, y_1 + y_2)$.

3.2 Rectangle Class

The `Rectangle` class is used for representing rectangle shapes. You are provided with the following `Rectangle.h` header file:

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

#include "coordinate.h"

class Rectangle{
    private:
        // add private members, methods and constructors here as you need
    public:
        // do not make any modifications below
        Rectangle(const Coordinate&, const Coordinate&);
        Rectangle(const Rectangle&);
        ~Rectangle();
        Coordinate getTopLeft() const;
        Coordinate getBottomRight() const;
        void setTopLeft(const Coordinate&);
        void setBottomRight(const Coordinate&);
        bool contains(const Coordinate&) const;
        bool intersects(const Rectangle&) const;
};

#endif
```

Descriptions for each one of the public constructors and methods are given below:

Name	**Description**
`Rectangle::Rectangle(const Coordinate&, const Coordinate&)`	Constructor taking two arguments. The first argument is the coordinate of the top-left corner, the second argument is the coordinate of the bottom-right corner.
`Rectangle::Rectangle(const Rectangle&)`	Copy constructor.
`Rectangle::~Rectangle()`	Destructor.
`Coordinate Rectangle::getTopLeft() const`	Returns the coordinate of the top-left corner of the rectangle.
`Coordinate Rectangle::getBottomRight() const`	Returns the coordinate of the bottom-left corner of the rectangle.
`void Rectangle::setTopLeft(const Coordinate&)`	Setter for the coordinate of the top-left corner.
`void Rectangle::setBottomRight(const Coordinate&)`	Setter for the coordinate of the bottom-right corner.
`bool Rectangle::contains(const Coordinate&) const`	Returns true if the rectangle contains the given coordinate and false otherwise.
`bool Rectangle::intersects(const Rectangle&) const`	Returns true if the rectangle intersects with the given rectangle and false otherwise.

3.3 QTree Class

The `QTree` class is used for representing region quad-trees. You are provided with the following `qtree.h` header file:

```
#ifndef QTREE_H
#define QTREE_H

#include "coordinate.h"
#include "rectangle.h"
#include <iostream>

using namespace std;

class QTree{
    private:
        // add private members, methods and constructors here as you need
    public:
        // do not make any modifications below
        QTree(int);
        QTree(const QTree&);
        ~QTree();
        void insert(const Rectangle&);
        const Rectangle* operator[](const Coordinate&) const;
        friend ostream& operator<<(ostream&, const QTree&);
        QTree& operator=(const QTree&);
};

#endif
```

Descriptions for each one of the friends, public constructors and methods are given below:

Name	**Description**
`QTree::QTree(int)`	Constructor taking an `int` as its argument. The argument is the length of a side of the square region covered by the quad-tree.
`QTree::QTree(const QTree&)`	Copy constructor.
`QTree::~QTree()`	Destructor.
`void QTree::insert(const Rectangle&)`	Inserts the given rectangle to the quad-tree.
`const Rectangle* QTree::operator[](const Coordinate&) const`	Returns a pointer to the rectangle at the given coordinate. If no rectangle is present at the given coordinate, returns `NULL`.
`friend ostream& operator<<(ostream&, const QTree&)`	Overloaded << operator used for printing the quad-tree. Output syntax of this method is given in Section 3.4 below.
`QTree& QTree::operator=(const QTree&)`	Overloaded assignment operator.

1. The region covered by the root node of the quad-tree is going to be a $2^n \times 2^n$ square where n is a non-negative integer. Therefore, `QTree::QTree(int)` constructor will be called only using 2^n as its argument, where n is a nonnegative integer (i.e. `QTree(1)`, `QTree(2)`, `QTree(4)`, `QTree(8)`, `QTree(16)` etc. are valid constructor calls whereas `QTree(-1)`, `QTree(3)`, `QTree(12)` etc. are not). You do not need to perform error-checking though. This constructor will be called only using powers of 2 during grading.

2. Only the leaf nodes in the quad-tree may keep rectangles. A leaf node must keep a pointer to the rectangle object if the rectangle has at least one pixel intersection with the region of the leaf node.

3. The capacity of each node is 1. Therefore, a node may be either empty or full. If the node is full and a new rectangle is inserted, the leaf node must split recursively into 4 equal quadrants until all the new leaf nodes point to at most 1 rectangle object.

4. The inserted rectangles will not overlap.

5. The inserted rectangles will be inside the region of the quad-tree.

6. The overloaded operator `Rectangle* QTree::operator[](const Coordinate&) const` must traverse the quad-tree starting from the root node. At each level, this operator must descend into the child node whose region contains the given coordinate until a leaf node is reached. When the operator reaches the leaf node containing the given coordinate, it must return the pointer to the `Rectangle` object in that node.

3.4 Syntax for the output of `friend ostream& operator<<(ostream&, const QTree&)`

The output of the `friend ostream& operator<<(ostream&, const QTree&)` must have the following syntax:

```
(<x>,<y>) [<length>] <rectangle>
    <contents of the NorthWest child, if exists>
    <contents of the NorthEast child, if exists>
    <contents of the SouthEast child, if exists>
    <contents of the SouthWest child, if exists>
```

`<x>` and `<y>` are integers which correspond to the x and y coordinates of the top-left corner of a node in the quad-tree, respectively. `<length>` is an integer which is the length of the one side of the square region covered by the node.

If the node is empty (i.e. does not contain a rectangle) `<rectangle>` is "`...`" (without the quotes). Otherwise, it has the following syntax:

```
*** (<xr>,<yr>) [<h_length>,<v_length>] ***
```

`<xr>` and `<yr>` are x and y coordinates of the top-left corner of the rectangle, respectively. `<h_length>` and `<v_length>` are the horizontal and vertical lengths of the rectangle, respectively.

If the node has children, the contents of them are printed in the following lines, starting from the north-west child and moving in clockwise direction. The contents of the children are indented by 4 spaces at each level of the tree. Since the children are also quad-trees, the syntax given above is used recursively for printing them. The tree is traversed in depth-first order while printing (see the example in Section 4 below).

4 Sample Input and Output

There is a sample `main.cpp` code below, which provides a sample `main` function. You are **NOT** going to write a `main` function in any one of your files.

```
#include "rectangle.h"
#include "coordinate.h"
#include "qtree.h"

#include <iostream>
```

```
using namespace std;

int main(int argc, char* argv[]){
    QTree qtree(32);
    Rectangle a(Coordinate(5,5), Coordinate(15,10));
    Rectangle b(Coordinate(16,16), Coordinate(16,25));
    Rectangle c(Coordinate(22,24), Coordinate(25,30));

    qtree.insert(a);
    cout << qtree << endl;
    qtree.insert(b);
    cout << qtree << endl;
    qtree.insert(c);
    cout << qtree << endl;

    const Rectangle *d = qtree[Coordinate(23,28)];
    if (d == NULL){
        cout << "No shape at the given position." << endl;
    }
    else{
        cout << d->getTopLeft().getx() << " " << d->getTopLeft().gety() << endl;
        cout << d->getBottomRight().getx() << " " << d->getBottomRight().gety() ↩
           << endl;
    }

    return 0;
}
```

The provided `main` function creates a quad-tree object whose region size is 32×32. Then it inserts three rectangle objects one by one and calls the `print` method of the quad-tree after each insertion. Next, it uses the overloaded `[]` operator to access the shape at $(23, 28)$. See the output of this code below.

```
(0,0) [32] *** (5,5) [11,6] ***

(0,0) [32] ...
    (0,0) [16] *** (5,5) [11,6] ***
    (16,0) [16] ...
    (16,16) [16] *** (16,16) [1,10] ***
    (0,16) [16] ...

(0,0) [32] ...
    (0,0) [16] *** (5,5) [11,6] ***
    (16,0) [16] ...
    (16,16) [16] ...
        (16,16) [8] *** (16,16) [1,10] ***
        (24,16) [8] ...
        (24,24) [8] *** (22,24) [4,7] ***
        (16,24) [8] ...
            (16,24) [4] *** (16,16) [1,10] ***
            (20,24) [4] *** (22,24) [4,7] ***
            (20,28) [4] *** (22,24) [4,7] ***
            (16,28) [4] ...
    (0,16) [16] ...

22 24
25 30
```

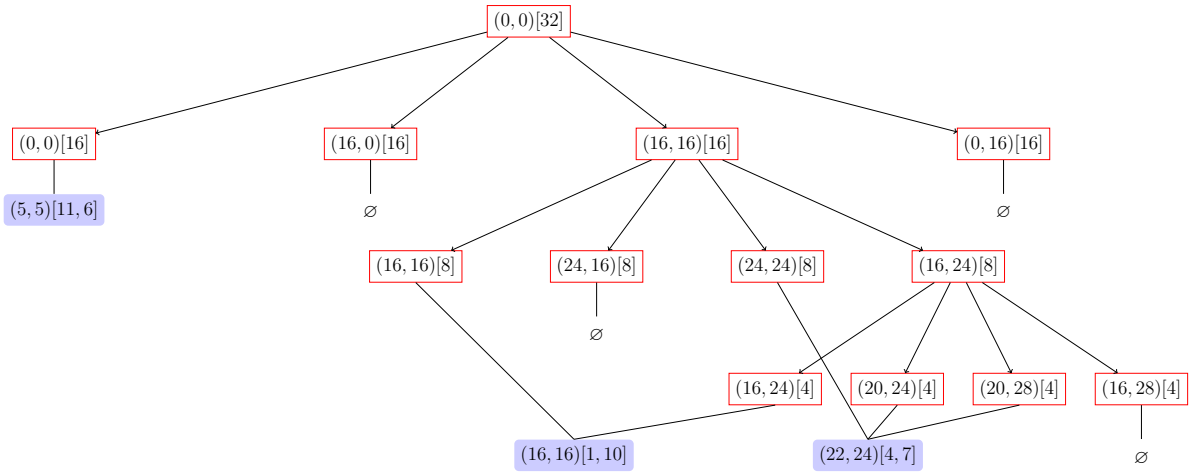

Figure 2: The quad-tree created in the sample `main` function.

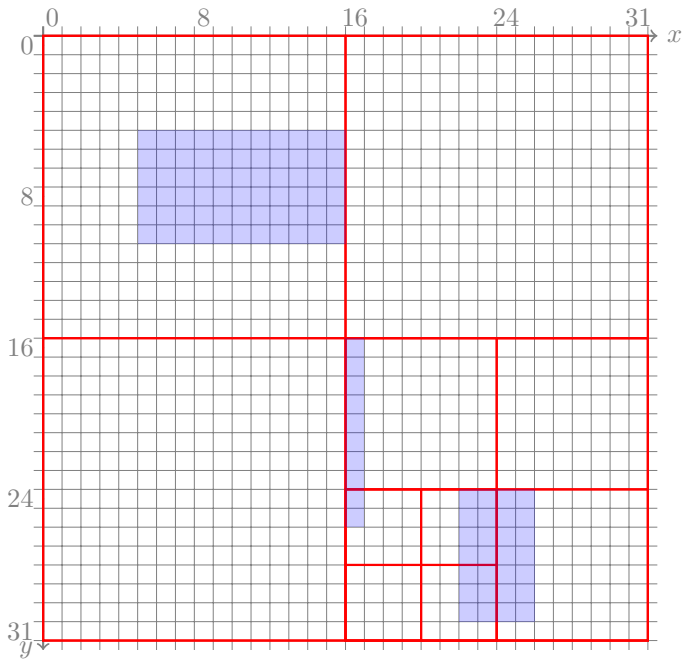

Figure 3: The regions of quad-tree nodes (red) and inserted rectangles (blue).

5 Regulations

- **Memory-leak:** The class destructors must free all of the used heap memory. Any heap block, which is not freed at the end of the program will result in grade deduction. Please check your codes using `valgrind --leak-check=full` for memory-leaks.
- **Performance:** Time complexity of a call to `void QTree::insert(const Rectangle&)` and `Rectangle* QTree::operator[](const Coordinate&) const` must be $\Theta(\log n)$ on average where n is the number of nodes in the quad-tree. During grading, the time complexity will be checked by inserting and querying large number of rectangles to quad-trees defined on large regions.
- **Allowed Libraries:** You may include and use C++ Standard Library. Use of any other library (especially the external libraries found on the internet) is forbidden.

- **Programming Language:** You must code your program in C++. Your submission will be compiled and graded with `g++` on department lab computers. You are expected make sure your code compiles successfully with `g++` using the flags `-ansi -pedantic`.
- **Late Submission:** A penalty of $5 * day * day$ is applied.
- **Cheating:** Cheating will result in receiving 0 from all assignments and the university regulations will be applied.
- **Newsgroup:** You must follow the Ceng242 newsgroup for discussions and possible updates on a daily basis.
- **Grading:** This homework will be graded out of 100.

6 Submission

Submission will be done via Moodle. Do **NOT** write a `main` function in any one of the files. You are going to submit the following files:

- `coordinate.h`
- `coordinate.cpp`
- `rectangle.h`
- `rectangle.cpp`
- `qtree.h`
- `qtree.cpp`

Your codes must successfully compile and run using the command sequence below:

```
$ g++ -ansi -pedantic coordinate.cpp rectangle.cpp qtree.cpp main.cpp -o hw3
$ ./hw3
```

You are not going to submit a `main.cpp` file. We will use our own `main.cpp` during grading. There is a sample `main.cpp` provided in Section 4 above. You may use it while developing and debugging your codes, if you like.