Computing Structures – Fall 2023
Project 2
Due: October 13th 2023 at 11:59 PM

Objective:

This project requires you to create and use a data structure for storing two dimensional points, and query the data structure with different rectangles to determine which point lie inside each rectangle queried.

Description:

The project makes use of a tree data structure which stores points in a 2D space, defined by a Rectangle. Each node of the tree will have a limited number of points stored in a vector. If the number of points exceeds the threshold, the space defined by the rectangle is divided into four quadrants, north-west, north-east, south-east and south-west. These quadrants are defined by a Rectangle and are the children of the previous node. The points are then stored in one of the new nodes created. If any of these nodes are full of points, then they will in turn be divided into four quadrants, and so on.
Once the points are stored in this tree, the tree will be queried with rectangles to determine the points that lie within the queried rectangles.

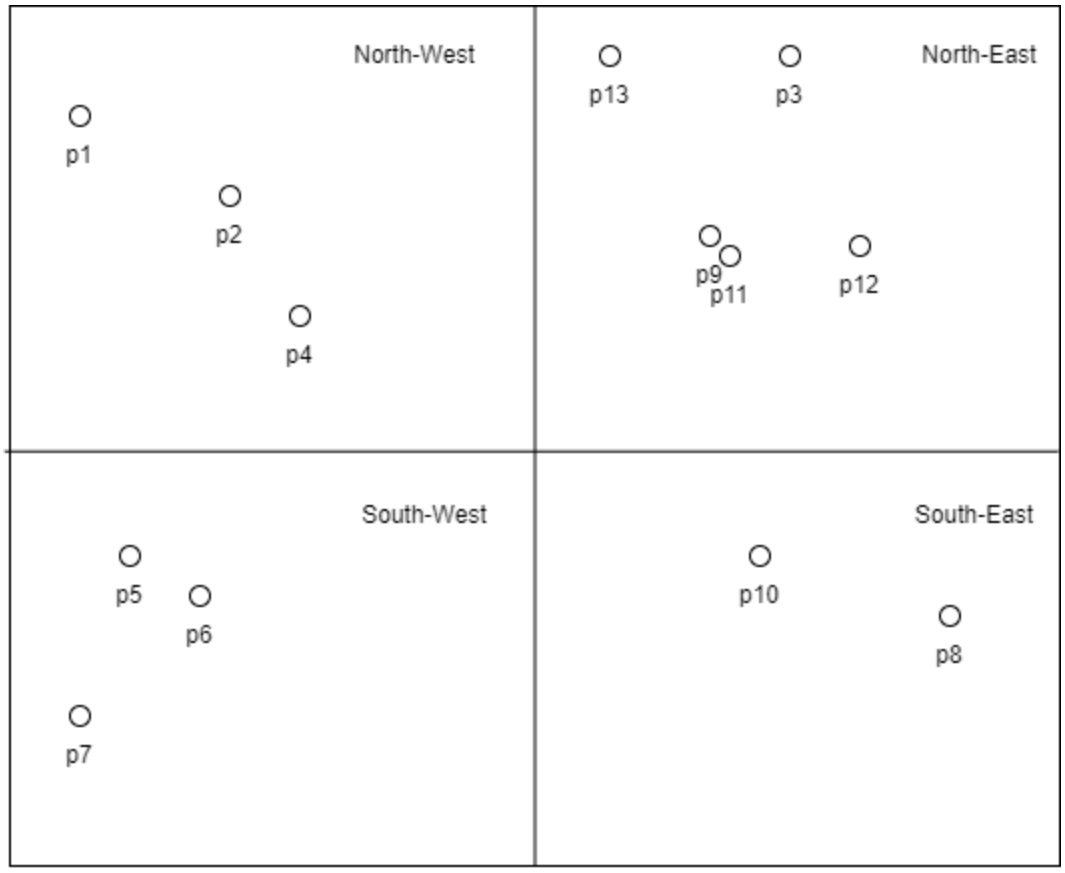

Fig. 1.

Fig. 1. Shows points in a 2D space. These are stored in a tree with maximum 4 points to be stored per node. Thus, the space is divided into 4 quadrants, as shown. The North-East quadrant still has more than the threshold amount of points, so it is divided further:

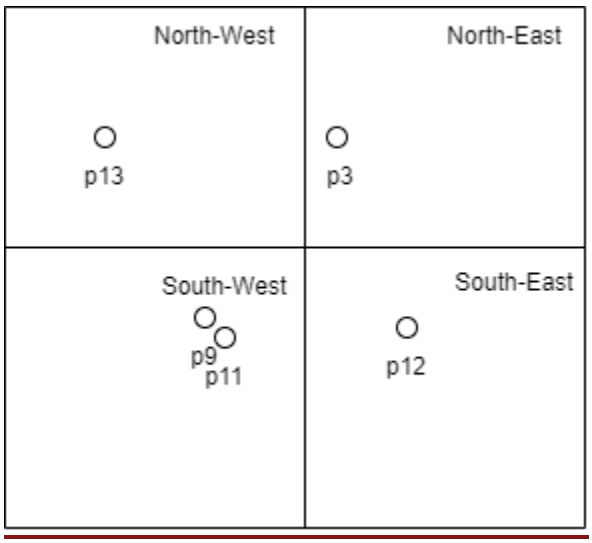

Fig. 2

You will be provided with the data structure to use for storing the points in the project boilerplate code. In addition, following is the algorithm for creating the tree structure:

1. Store points in the tree node until full.
2. If full, divide the node's 2D space into four quadrants, North-West, North-East, South-West and South-East. These are children of the said node.
3. Fill the remaining point(s) in the children, wherever applicable (Step 1).

The original node with the original rectangular space will have nodes, then the partitions of the rectangular space will also have points, if applicable. However, the points in the original space will not be copied into the new smaller partitions just because the smaller partitions represent the same space in parts.

Following is the algorithm for doing the search in the search space:

1. If a query rectangle intersects with the boundary of a search space, perform the below steps. Else return.
2. Iterate through all points in the boundary of the search space, check if they lie inside the queried rectangle. If yes, report them by storing them in a vector.
3. If the search space is further divided into quadrants, recursively check the quadrants with step 1.
4. Once the search space is exhausted till the last level, print the points in the vector after sorting them. Sorting will be done first on the 'x' coordinate. If the 'x' coordinate of two points is the same, sort on 'y' coordinate. The sorting will be ascending order.

Input explanation:

The input will be from a single file which will be read as ***a redirected input***. The file will have the following format:

```
10000  100                          //Number of points in the file and max points per node.

9 4 2 7 7 3 5 7 5 8 ……              // Points begin. The first number is 'x' coordinate
……..                               // the second number is the 'y' coordinate. Five points
……..                               // are shown.
. .
. .
. .
. .

4                                   //The number of rectangles to be queried.
1 1 4 5                             //Four numbers to define rectangle shapes.
1 3 7 2                             //Each line will have a single rectangle.
4 2 5 5
7 1 2 2
```

The input file will define the number of points in the file in the first line. The first line will have a second number which will define the maximum number of points for each node in the tree.

The next few lines will have the points defined by their (x, y) coordinates. A single line may have multiple points, but each point will have it's 'x' and 'y' coordinate on the same line (It doesn't matter if it is on the same line or not, what matters is that there be space between each unique number).
Once you read in the numbers, you will be required to create the data structure, as defined in the algorithm for the construction of the tree.
Once done with the construction of the tree, you will read the query rectangles (one on each line). The number of rectangles to be queried is provided in the file, before the rectangles are defined. The query rectangles are defined by four numbers:

The first two numbers are the (x, y) coordinates of the **top-left corner** of the rectangle, and the remaining two are the width and height of the rectangle. The width is considered along the 'x' coordinate and height is considered along the 'y' coordinate.

Class definition:

The user defined classes will have the following layout:

```
class Point {
    public:
    int x, y;
    Point (int x, int y);
};
```

The class Point will store the (x, y) coordinates of the points. You don't need any member functions here.

```
class Rectangle {
    public:
    int x_top_left, y_top_left, width, height; //variables for defining the
rectangle.

    //Constructor
    Rectangle(int x, int y, int width, int height);

    //For determining if the rectangle contains the point p.
    bool contains(Point p);

    //For determining if this rectangle intersects with other.
    bool intersects(Rectangle other);
};
```

The class Rectangle will store the rectangle shape. Prototypes for the member functions are provided which you have to implement.

```
class twoDimSpace {
Rectangle boundary;                    //Defines the boundary of this node
    int capacity;                      //Defines the number of points the node
can hold.
    std::vector<Point> points;         //Storing the points in a templated
vector (like Project 1)
    bool divided = false;              //A variable to store whether the node is
further divided into quadrants
    twoDimSpace *northWest, *northEast; //Pointers to child nodes.
    twoDimSpace *southWest, *southEast; //Pointers to child nodes.

public:

    //Constructor for creating the node
    twoDimSpace(Rectangle boundary, int capacity);

    //Destructor
    ~twoDimSpace();

    //Method to divide the boundary of this node into four equal quadrants.
    //The four quadrants will be used to instantiate the four children mentioned.
    void subdivide();

    //Method for inserting a point in the node. If the node is full, it's
children
    //are to be checked recursively.
    bool insert(Point point);

    //Method for querying a Rectangle. The method will check if the node's
boundary and
    //give rectangle intersect. If yes, iterate through all points in the node.
If node
    //has children, check them recursively for points.
    void query(Rectangle rectangle, std::vector<Point>& found);
};
```

The twoDimSpace class will store the points and will form a tree structure.
The methods shown will have to be implemented.

A note on the ***query*** **method**: This method has the second argument passed as a reference. This is deliberate because this function will be recursive, and we want our findings in one call to the

function to stick. Passing a vector by reference makes it so that each function call adds points to the same vector. Otherwise, if the argument was passed as value, each function call would result in the instantiation of a new vector, and the points found would be stored in different vectors, to whom we would have no handle in the main function. The function can not be used to print the points as we need to sort them first.

Output:

The output will have the points inside each rectangle queried. The output file will be provided shortly.

Submission:

Submission will be through *GradeScope*. Your program will be autograded. You may submit how many ever times to check if your program passes the test cases provided. One test case will be released at the beginning and later other test cases will be released while grading. For the autograder to work, the program you upload must be named as ***projec2.cpp***.

Your final submission must contain your source file named project2.cpp. You access GradeScope using the tab on the left in our course page on canvas.

Constraints:

1. None of the projects is a group project. Consulting on programming projects with anyone is strictly forbidden and plagiarism charges will be imposed on students who do not follow this.
2. Please review academic integrity policies in the modules.

Programming advice:

1. Before working on data manipulation, please make sure that you are reading the data correctly. Use cout statements to print out the values you are storing. Data properly read and stored is 70% of the job done, in my opinion.
2. Consistency in effort will reduce the workload significantly.
3. If your program works for reading data from the input files provided, try creating your own files and see if your program reads them properly too.
4. The autograder runs on the GNU C++ library, which may behave slightly differently than the ones installed with Microsoft Visual Studio, for example. Thus, if your program works correctly on your machine, please make sure to test it on Autograder as well. Your program needs to work on Autograder.

How to ask for help:

1. Please attend the help sessions. You can even stay there while you work.

2. Please always have a lookout for announcements on canvas. This class has video lectures that were recorded in the past, so the current information on the course will be mainly through announcements.
3. Email the TA with your ***precise*** questions. Please attach a snapshot of the error and your source file to the email if the question is regarding error(s) in compilation or runtime.