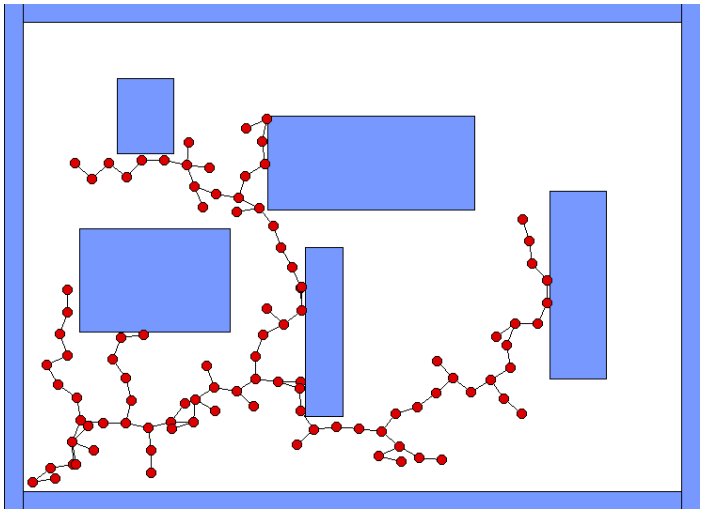
COMP2401 - Assignment #4

(Due: **Thursday, March 17th, 2022 @ 6pm**)

In this assignment, you will gain additional practice using structs, pointers and dynamic memory allocation while creating a tree with a similar structure to that of Linked-Lists.



In the world of robotics, it is ALWAYS a good idea to program a robot to move around in its environment without hitting things. If a map of the environment is given, a robot can plan paths around the obstacles. There are many ways to perform path planning and some of the solutions involve creating a graph that can be traversed. In this assignment, we will create what is called a **Rapidly-exploring Random Tree** (RRT). The tree starts at a root node and branches out a random locations based on a simple algorithm, and stops once a certain number of nodes have been added to the tree. The branches in the tree are all the same length (which is a parameter of the algorithm).

A sequence of branches in this tree, starting at the root, represents a path in the environment that does not intersect obstacles. The RRT allows us to compute reasonably-short paths in an environment from a start location (i.e., the root of the tree) to any other location in the environment. It will be your goal to construct the tree dynamically and then to compute a path in the tree dynamically as well.

We will run our code on 5 fixed environments that are composed of rectangular obstacles. These environments will be given to you. The bottom left of the environments is always coordinate (0, 0).

To begin the assignment, you should download the following files:

- **display.c** and **display.h** – code for displaying the environment, tree and path
- **obstacle.h** – structures required for this assignment
- **rrtMaker.c** – your code will be mostly written here (but there is no **main** function)
- **rrtTester.c** – this is the test program that you will run

You MUST not alter the **display.c**, **display.h** nor **obstacle.h** files. You are also NOT allowed to alter the **structs** defined in the **obstacle.h** file, NOR are you allowed to create any additional structs on this assignment. You MUST NOT CREATE ANY STATIC ARRAYS on this assignment.

(1) You MUST first create a proper **makefile** that defines the proper dependencies, compiles the files and creates an executable called **rrtTester**. The **make clean** command should also work properly. Run the program. It should display the first environment and then wait for you to press the **Enter** key (you will likely have to click somewhere in the terminal window first so that it has the focus of the cursor). The window should then close. The code will only link and run if you include the **-lX11** library (that's "minus L X eleven") as well as the **-lm** math library.

(2) You need to write code for the 3 empty/blank procedures in the **rrtMaker.c** file. Each environment is defined as indicated in the **Environment** typedef. It has a (**startyX**, **startY**) defined to be the location of the root node. The **maximumX** and **maximumY** indicate the width and height of the environment. The **maximumNodes** indicates the maximum number of nodes that the RRT will have and **growthAmount** indicates the distance between a parent node and one of its children. The environment also has a dynamically-created array of **Obstacles** a dynamically-created RRT of **TreeNodes**, and a dynamically-created path of **TreeNodes**.

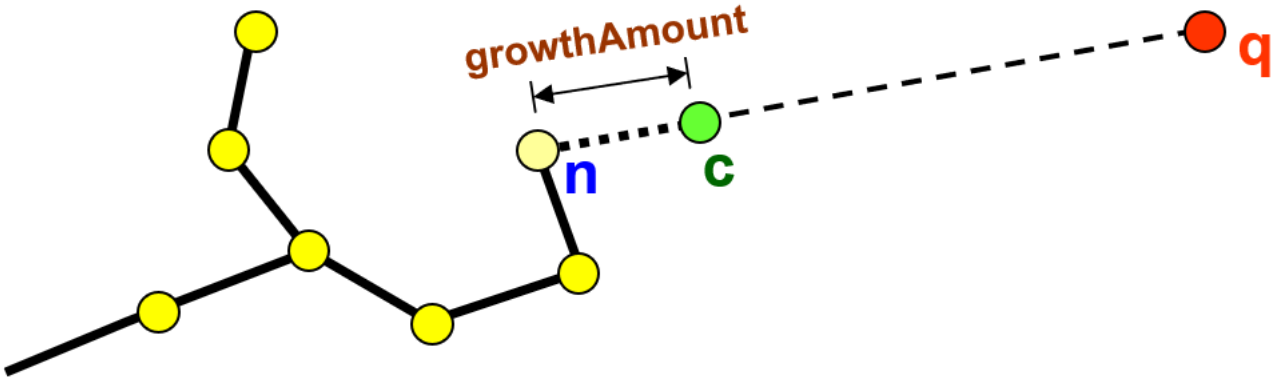
```
typedef struct {
  unsigned short     startX;
  unsigned short     startY;
  unsigned short     maximumX;
  unsigned short     maximumY;
  unsigned short     maximumNodes;
  unsigned char      growthAmount;
  Obstacle           *obstacles;
  unsigned short     numObstacles;
  TreeNode           **rrt;
  unsigned short     numNodes;
  TreeNode           **path;
} Environment;
```

The obstacles array has already been created for you. Each **Obstacle** is assumed to be rectangular and defined as an (**x**, **y**) point (i.e., it's top left corner coordinate) and a **w** width and **h** height

```
typedef struct obst {
  short      x;
  short      y;
  short      w;
  short      h;
} Obstacle;
```

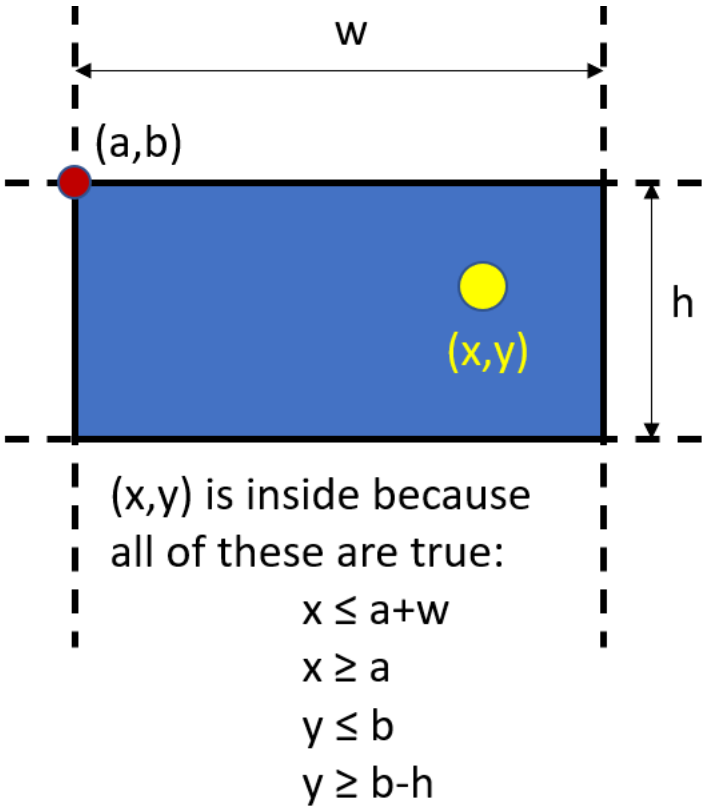
Here is the algorithm for computing the RRT:

```
RRT = an empty tree with a single root node
REPEAT
  q = a random point in the environment that is not inside an obstacle
  n = node of RRT that is closest to q
  c = point along ray from n to q that is distance growthAmount from n
  IF (n→c does not intersect any obstacles) THEN
     Add c as a child of n in RRT
UNTIL the RRT has maximumNodes nodes
```



The algorithm is quite simple … but you will need to know how to do a few things.

- To determine if a point is inside an obstacle, you can simply check the x and y coordinates:



- To determine if a line segment (x1, y1)→(x2, y2) intersects an obstacle, just check if it intersects any of the 4 obstacle edges. A function has been provided for you that determines if two line segments intersect.
- Given point **n** and point **q**, you will need to compute point **c**. You need to first compute the **angle** that ray **n**→**q** makes as **atan2**($q_x - n_x$, $q_y - n_y$) then you can compute **c** as being

(n_x + cos(**angle**)*__growthAmount__, n_y + sin(**angle**)*__growthAmount__);

You will also need to understand the **TreeNode** and **Child** structures. A **TreeNode** keeps the (**x,y**) coordinate of the node in the environment. It also keeps a pointer to its **parent** in the tree as well as a singly-linked list of its children nodes. It really just keeps the **firstChild** in this linked-list … which is the head of the list.

```
typedef struct node {
    short          x;             // Location of the node in the environment
    short          y;
    struct node   *parent;        // Parent of this node in the tree
    struct child  *firstChild;    // The first child in a singly-linked list of children
} TreeNode;
```

The singly-linked list of children is make up of **Child** types:

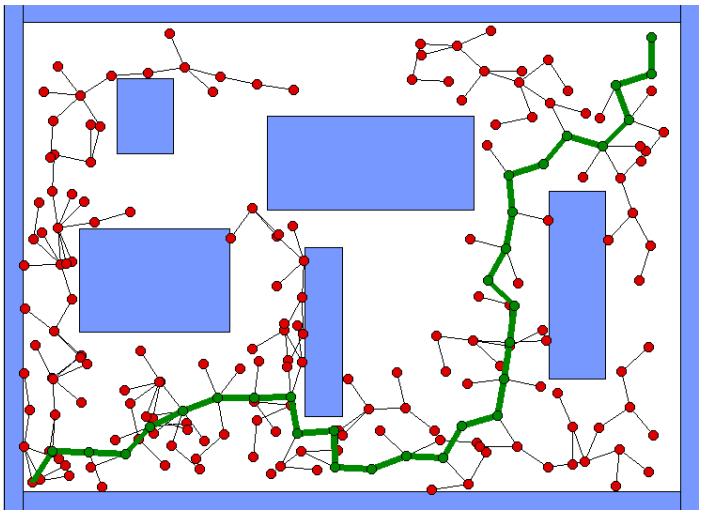
```
typedef struct child {
    TreeNode      *node;          // A child of a tree node
    struct child  *nextSibling;   // The next child (i.e., sibling) in the list
} Child;
```

(3) Write the **createRRT()** procedure. Add code so that it allocates a dynamic array that can hold pointers of up to a **maximumNodes** number of tree nodes. This array MUST be stored into the **rrt** attribute of the **Environment** structure. Make sure to create the root of the tree as the first node in this array with coordinate (**startX**, **startY**), which are **Environment** attributes as well.

Next, implement the RRT creation algorithm described above. Each time you create a new **TreeNode**, you should add it to the environment's dynamically-created **rrt** array as well as connect it appropriately to the tree by connecting the parent and children pointers accordingly. Make sure that whenever you add a child to a parent node, the children must remain in the same order that they were created. That means once you create the children list for a node by adding its **firstChild** as the head … then the head of that list NEVER changes.

Run your code to make sure that it works. Your tree branches should never intersect any obstacles and none of your tree nodes should be inside an obstacle. Keep in mind though that the tree nodes are drawn larger than a single point, so just make sure that the center pixel of the node is not inside an obstacle.

(4) Write the **tracePath()** procedure. It should first find the node in the tree that is closest to the given (x,y) coordinate. You must then dynamically-allocate an array to store a sequence of **TreeNode** pointers that represents the path from that closest node back to the root of the tree. The array MUST be the exact size needed to store the nodes of the path (which will vary each time that you run the program). You will likely want to count the nodes beforehand, from the end location to the root, so that you know how big to make the array. To get the path, then you just follow the parent pointers from the end of the path back to the root.



Run your code to make sure that it works. If your path was created properly, you should see the darker green thicker lines along that path as well a green Nodes instead of red.

(5) You now want to ensure that your code works for varying **growthAmount** and **maximumNodes** values as well as on each of the 5 environments. Go to the **rrtTester.c** program and alter the **main()** function so that the program takes three command-line arguments. The 1st must be an integer in the range from 5 to 100 which will represent the **growthAmount** (in pixels) to be used when creating the RRT. The 2nd command-line argument must be a number from 5 to 5000 and this represents the maximum number of nodes that the RRT will have once completed. The 3rd command-line argument must be a number from 1 to 5 and this represents the environment to be used for testing. Make sure to handle the situation where there are not exactly 3 command line arguments available and make sure to check if they are in valid ranges … otherwise you should display an error message that explains what the command line arguments should be … and quit the program. Test your code with different values sizes to make sure that it works properly.

(6) Finally, write code in the **cleanupEverything()** procedure so that your program has no memory leaks **nor errors**. Use `valgrind ./rrtTester 25 200 1 -leakcheck==yes` to do this … although you should try various combinations of grid sizes and environments as well.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit:

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted
   - any specific instructions for compiling and/or running your code

2. All of your **.c source** files and all other files needed for testing/running your programs.

3. Any output files required, if there are any.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).