15869 Homework Assignment 1:
Due at the start of class, Thursday Oct 20

Question 1: (10 points) In class we discussed a number of ways a processor might implement the SIMT abstraction for code containing nested conditions. Describe how you would extend this implementation to handle programs containing `for` or `while` loops with data-dependent loop bounds. Be precise in your description of details such as how lane masks at set in each loop iteration, how masks are reset after leaving the loop body. Be sure to describe any state you keep around to do this.

Question 2: (10 points)

A common operation in computer graphics is to find intersections between a point and scene objects. During the rasterization lecture we discussed algorithms for solving the following instance of this problem: "given a primitive in 2D, find all points contained within the primitive." This question flips the problem around. Given a point and a set of line segments you need to efficiently find all the line segments that contain the point. To simply things, you only need to carry out the computation in 1D.

Figure 1 shows a collection of line segments in 1D (the start and end of each segment is given). The figure also shows a binary tree data structure organizing the segments into a spatial hierarchy. Leaves of the tree correspond to the line segments. Each interior node of the hierarchy represents a spatial extent spanned by its children. Notice that sibling leaves can overlap. Using this tree data structure, it is possible to answer the question "what segments contain a specified point" without testing the point against all segments in the scene.

The function `find_largest_segment_1` uses the tree data structure in Figure 1 to quickly find all line segments containing a point in 1D. It returns the result of `very_expensive_function` called on the *largest* of the line segments containing the point. For example, if this was a simple renderer, one possible implementation of `very_expensive_function` might compute the color of the line segment at the intersection point. For simplicity, assume that `very_expensive_function` is a straight-line block of code with no conditionals or data-dependent control.

Study the algorithm, and understand how it works. For example, given the point 0.1, the algorithm will perform the following sequence of operations: (I-test,N0), (I-hit,N0), (I-test,N1), (I-hit,N1), (I-test,N2), (I-hit,N2) (L-test, N3), (VEF, N3), (I-test, N4), (I-hit, N4), (L-test,N5), (VEF, N5), (L-test, N6), (L-miss, N6), (L-test,N7), (L-miss,N7) , (I-test, N8), (I-miss, N8)

where:

(I-test, Nx) represents a point-interior node test against Node X.
(I-hit, Nx) represents logic of traversing to the child nodes after it is determined the query point is contained within Node X.
(I-miss, Nx) represents logic of traversing to sibling/ancestor nodes when the point is not contained within node X.
(L-test,Nx) represents a point-leaf node test against the segment represented by Node X.
(VEF, Nx) represents `very_expensive_function` executed on node X.

Now consider simultaneous SIMT-style execution of `find_largest_segment_1` on a 4-wide system using the four points 0.1, 0.4, 0.7, and 0.75 as inputs. Using the notation established above, chart the utilization of each "lane" of the processor in the 4-column matrix below (columns indicate behavior of each of the four SIMT lanes, and rows correspond to processor behavior at a particular point in time). Note that the first column of the matrix should contain the values given in the example for point 0.1 above. It may be helpful to use --- to indicate that a lane's operation is masked at a particular time.

```
struct Node {
   float min, max;
   bool leaf;
   Node* left;
   Node* right;
};

// returns the value of very_expensive_function(node, pt_x) for the largest
// segment containing pt_x.  If no segment contains pt_x, returns NO_SEGMENT

float find_largest_segment_1(float pt_x, Node* root_node)
{

  Stack<Node*> stack;
  Node* node;
  float max_extent = 0.0;
  float result = NO_SEGMENT;

  stack.push(root_node);

  while(!stack.size() == 0)
  {

    node = stack.pop();

    while (!node->leaf)
    {

      // I-test: test to see if point is contained within interior node
      if (pt_x >= node->min && pt_x <= node->max)
      {
        // I-hit: continue to child nodes
        push(node->right);
        node = node->left;
      }
      else
      {
        // I-miss: point not contained within node
        if (stack.size() == 0)
          return NO_SEGMENT;
        else
          node = stack.pop();
      }
    }

    // L-test: test to see if point is contained within line segment (leaf node)
    if (pt_x >= node->min && pt_x <= node->max && (node->max-node->min) > max_extent)
    {
       // this basic block is referred to as VEF in problem description:
       result = very_expensive_function(node, pt_x);
       max_extent = node->max – node->min;
    }
  }

  return result;
}
```

pt_x = 0.1	pt_x = 0.4	pt_x = 0.7	pt_x = 0.75
(I-test, N0)	(I-test, N0)	(I-test, N0)	(I-test, N0)
(I-hit, N0)	(I-hit, N0)	(I-hit, N0)	(I-hit, N0)

Question 3: (10 points)

The function `find_largest_segment_2` produces the same output as `find_largest_segment_1`.

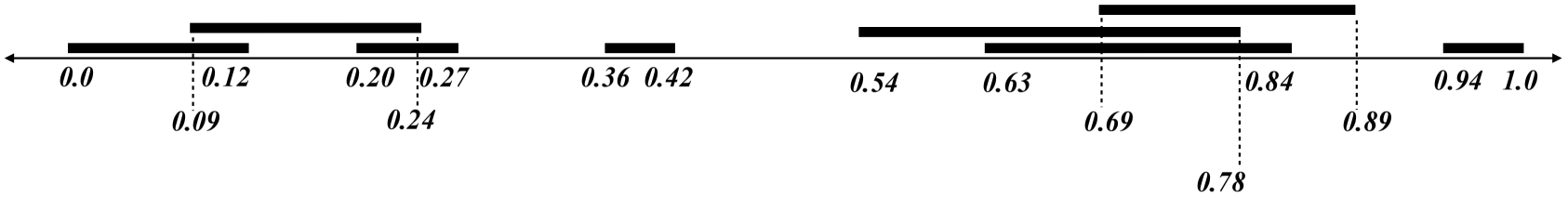
Chart its SIMT execution behavior on the same four rays as in question 2 and then intuitively describe the differences in how `find_largest_segment_1` and `find_largest_segment_2` execute. Are there advantages and disadvantages of the two approaches? Although I only provided one example set of segments and point queries in this assignment, it will be helpful to consider the execution behavior of these functions under varying characteristics of the binary tree (e.g., consider very large, unbalanced trees), different costs of `very_expensive_function`, and even different point queries.

```
float find_largest_segment_2(float pt_x, Node* root_node)
{

  Stack<Node*> stack;
  Node* node;
  float max_extent = 0.0;
  float result = NO_SEGMENT;

  stack.push(root_node);

  while(!stack.size() == 0)
  {
    node = stack.pop();

    if (!node->leaf)
    {
       // I-test: test to see if point is contained within interior node
       if (pt_x >= node->min && pt_x <= node->max)
       {
          // I-hit: continue to child nodes
          push(node->right);
          push(node->left);
       }
    }
    else
    {
       // L-test: test to see if point is contained within line segment (leaf node)
       if (pt_x >= node->min && pt_x <= node->max && (node->max-node->min) > max_extent)
       {
          // this basic block is referred to as VEF in the problem description:
          result = very_expensive_function(node, pt_x);
          max_extent = node->max – node->min;
       }
    }
  }

  return result;
}
```

pt_x = 0.1	pt_x = 0.4	pt_x = 0.7	pt_x = 0.75
(I-test, N0)	(I-test, N0)	(I-test, N0)	(I-test, N0)
(I-hit, N0)	(I-hit, N0)	(I-hit, N0)	(I-hit, N0)

Figure 1

Line segments in 1D



Corresponding binary tree