Yazan Halawa
2/28/16
CS 312 Section 1
Lab 3 Report-Network Routing

Lab Code:

```
////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////// Priority Queue ////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////
/**
* An Abstract class that specifies the methods that a Priority Queue supports
*/
      public abstract class PriorityQueue
      {
         public PriorityQueue() { }

         public abstract void makeQueue(int numOfNodes);

         public virtual int deleteMin() { return 0; }

         public virtual int deleteMin(ref List<double> distanceArray) { return 0; }

         public virtual void decreaseKey(int targetIndex, double newKey) { }

         public virtual void decreaseKey(ref List<double> distanceArray, int targetIndex) { }

         public virtual void insert(int elementIndex, double value) { }

         public virtual void insert(ref List<double> distanceArray, int elementIndex) { }

         public abstract void printQueueContents();

         public abstract bool isEmpty();

      }

/**
* This class implements the Priority Queue methods using an array implementation. The Time
* and Space complexities are listed above the corresponding functions
*/
      public class PriorityQueueArray : PriorityQueue
      {
         private double[] queue;
```

```
        private int count;

        public PriorityQueueArray()
        {
        }

/**
* This functions returns whether the queue is empty or not. Time and Space = O(1) as it only
* involves an int comparison
*/
        public override bool isEmpty()
        {
            return count == 0;
        }

/**
* Helper function to print the contents of the queue. Time Complexity: O(n) where n is the
* count of elements in the queue.
* Space Complexity: O(1) as it does not create any extra variables that vary with the size of
the input.
*/
        public override void printQueueContents()
        {
            Console.Write("The contents of the queue are: ");
            for (int i = 0; i < count; i++)
            {
                Console.Write(queue[i] + " ");
            }
            Console.WriteLine();
        }

/**
* This function creates an array to implement the queue. Time and Space Complexities are
* both O(|V|) where |V| is
* the number of nodes. This is because you create an array of the same size and specify the
* value for each item by
* iterating over the entire array.
*/
        public override void makeQueue(int numOfNodes)
        {
            queue = new double[numOfNodes];
            for (int i = 0; i < numOfNodes; i++)
            {
                queue[i] = int.MaxValue;
```

```
            }
            count = numOfNodes;
        }

    /**
    * This method returns the index of the element with the minimum value and removes it from
     the queue.
    * Time Complexity: O(|V|) because we iterate over the entire array to find the min, where |V|
     is the number of nodes
    * Space Complexity: O(1) because we don't create any extra variables that vary with the size
     of the input.
    */
        public override int deleteMin()
        {
            double min = int.MaxValue;
            int minIndex = 0;
            for (int i = 0; i < queue.Count(); i++)
            {
                if (queue[i] < min)
                {
                    min = queue[i];
                    minIndex = i;
                }
            }
            count--;
            queue[minIndex] = int.MaxValue;
            return minIndex;
        }

    /**
    * This method updates a specific index in the array with a new key.
    * Time Complexity: O(1) as all it does is one assignment that does not vary with the size of the
      input.
    * Space Complexity: O(1) as it does not create any extra variables that vary with the size of the
      input.
    */
        public override void decreaseKey(int targetIndex, double newKey)
        {
            queue[targetIndex] = newKey;
        }
```

```
/**
 * This method inserts a new value into the queue and increments the count of nodes in the
queue
 * Time Complexity: O(1) as all it does is two assignments that do not vary with the size of the
input
 * Space Complexity: O(1) as it does not create any extra variables that vary with the size of
the input.
 */
        public override void insert(int elementIndex, double value)
        {
            queue[elementIndex] = value;
            count++;
        }
    }

    public class PriorityQueueHeap : PriorityQueue
    {
        private int capacity;
        private int count;
        private int lastElement;
        private int[] distances;
        private int[] pointers;
        public PriorityQueueHeap()
        {
        }

/**
* This functions returns whether the queue is empty or not. Time and Space = O(1) as it only
 involves an int comparison
*/
        public override bool isEmpty()
        {
            return count == 0;
        }

 /**
 * Helper method to print the contents of the queue. Time Complexity: O(|V|) where |V| is
the number of nodes.
 * Space Complexity: O(1) as it does not create any extra variables that vary with the size of
the input.
 */
        public override void printQueueContents()
        {
            Console.Write("The contents of the queue are: ");
```

```
            for (int i = 1; i < capacity; i++)
            {
                if (distances[i] != -1)
                    Console.Write(distances[i] + " ");
            }
            Console.WriteLine();
        }

/**
* This method creates an array to implement the queue. Time and Space Complexities are both
  O(|V|) where |V| is
* the number of nodes. This is because you create an array of the same size and specify the
  value for each item by
* iterating over the entire array.
*/
        public override void makeQueue(int numOfNodes)
        {
            distances = new int[numOfNodes+1];
            pointers = new int[numOfNodes];
            for (int i = 1; i < numOfNodes + 1; i++)
            {
                distances[i] = i-1;
                pointers[i - 1] = i;
            }
            capacity = numOfNodes;
            count = 0;
            lastElement = capacity;
        }

 /**
 * This method returns the index of the element with the minimum value and removes it from
  the queue.
 * Time Complexity: O(log(|V|)) because removing a node is constant time as we have its
  position in
 * the queue, then to readjust the heap we just bubble up the min value which takes as long as
 * the depth of the tree which is log(|V|), where |V| is the number of nodes
 * Space Complexity: O(1) because we don't create any extra variables that vary with the size
  of the input.
 */
        public override int deleteMin(ref List<double> distanceArray)
        {
            // grab the node with min value which will be at the root
            int minValue = distances[1];
            count--;
```

```
            //Console.WriteLine("last element is " + lastElement);
            if (lastElement == -1)
               return minValue;
            distances[1] = distances[lastElement];
            pointers[distances[1]] = 1;
            lastElement--;


            // fix the heap
            int indexIterator = 1;
            while (indexIterator <= lastElement)
            {
               // grab left child
               int smallerElementIndex = 2*indexIterator;

               // if child does not exist, break
               if (smallerElementIndex > lastElement)
                  break;

               // if right child exists and is of smaller value, pick it
               if (smallerElementIndex + 1 <= lastElement &&
distanceArray[distances[smallerElementIndex + 1]] <
distanceArray[distances[smallerElementIndex]])
               {
                  smallerElementIndex++;
               }

               if (distanceArray[distances[indexIterator]] >
distanceArray[distances[smallerElementIndex]])
               {
                  // set the node's value to that of its smaller child and update the iterator
                  int temp = distances[smallerElementIndex];
                  distances[smallerElementIndex] = distances[indexIterator];
                  distances[indexIterator] = temp;

                  pointers[distances[indexIterator]] = indexIterator;
                  pointers[distances[smallerElementIndex]] = smallerElementIndex;
               }

               indexIterator = smallerElementIndex;
            }
            // return the min value
            return minValue;
         }
```

```
/**
* This method updates a specific index in the array with a new key.
* Time Complexity: O(log(|V|)) as it finds the value whose position needs to be updated and
bubbles
* up the min value to the top which takes as long as the depth of the tree which is log|V|.
* Space Complexity: O(1) as it does not create any extra variables that vary with the size of
the input.
*/
        public override void decreaseKey(ref List<double> distanceArray, int targetIndex)
        {
            // find the node with the old value
            int indexToHeap = pointers[targetIndex];
            count++;

            // reorder the heap by bubbling up the min to top
            int indexIterator = indexToHeap;
            while (indexIterator > 1 && distanceArray[distances[indexIterator / 2]] >
distanceArray[distances[indexIterator]])
            {
                // swap the two nodes
                int temp = distances[indexIterator / 2];
                distances[indexIterator / 2] = distances[indexIterator];
                distances[indexIterator] = temp;

                // update the pointers array
                pointers[distances[indexIterator / 2]] = indexIterator / 2;
                pointers[distances[indexIterator]] = indexIterator;

                indexIterator /= 2;
            }

        }

/**
* This method updates the nodes in the queue after inserting a new node
* Time Complexity: O(log(|V|)) as reording the heap works by bubbling up the min value to
 the top
* which takes as long as the depth of the tree which is log|V|.
* Space Complexity: O(1) as it does not create any extra variables that vary with the size of the
 input.
*/
        public override void insert(ref List<double> distanceArray, int elementIndex)
        {
```

```
            // update the count
            count++;

            // as long as its parent has a larger value and have not hit the root
            int indexIterator = pointers[elementIndex];
            while (indexIterator > 1 && distanceArray[distances[indexIterator/2]] >
distanceArray[distances[indexIterator]])
            {
               // swap the two nodes
               int temp = distances[indexIterator / 2];
               distances[indexIterator / 2] = distances[indexIterator];
               distances[indexIterator] = temp;

               // update the pointers array
               pointers[distances[indexIterator / 2]] = indexIterator / 2;
               pointers[distances[indexIterator]] = indexIterator;

               indexIterator /= 2;
            }
         }
      }
```

//
//////////////////////////////////// Helper Functions //
//

```
/**
* Helper function to compute distance between two points
* Time Complexity: O(1) as all it does is some arithmetic which does not vary with the size of
the input
* Space Complexity: O(1) as it does not create an extra variables that vary with the size of the
input
*/
      private double computeDistance(PointF point1, PointF point2)
      {
         double deltaX = Math.Pow(point2.X - point1.X, 2);
         double deltaY = Math.Pow(point2.Y - point1.Y, 2);
         return Math.Sqrt(deltaX + deltaY);
      }
```

```
 /**
* Helper function to calculate the midpoint between two points
* Time Complexity: O(1) as all it does is some arithmetic which does not vary with the size of
 the input
* Space Complexity: O(1) as it does not create an extra variables that vary with the size of the
 input
*/
        private PointF findMidPoint(int firstIndex, int secondIndex)
        {
            PointF midPoint = new PointF();
            midPoint.X = (points[secondIndex].X + points[firstIndex].X) / 2;
            midPoint.Y = (points[secondIndex].Y + points[firstIndex].Y) / 2;
            return midPoint;
        }

 /**
 * Helper function to draw the path between the list of points
 * Time Complexity: O(n) as the worse case is it iterates over the entire the path. n is the size
 of the path list.
 * Space Complexity: O(1) as it does not create any extra variables that vary with the size of
 the input.
 */
        private void drawPath(ref List<int> path, bool isArray)
        {
            // Create variables to iterate through the path
            int currIndex = stopNodeIndex;
            int prevIndex = currIndex;
            double totalPathCost = 0;
            // Keep looping until the path from start node to end node is drawn
            while (true)
            {
                currIndex = path[currIndex];
                if (currIndex == -1) // if hit start node, exit cause the path is done
                    break;
                // Draw line
                Pen pen;
                if (isArray)
                    pen = new Pen(Color.Black, 2);
                else
                    pen = new Pen(Color.Red, 5);
                graphics.DrawLine(pen, points[currIndex], points[prevIndex]);

                // Label it with the distance
                double distance = computeDistance(points[currIndex], points[prevIndex]);
```

```
            totalPathCost += distance;
            graphics.DrawString(String.Format("{0}", (int)distance), SystemFonts.DefaultFont,
Brushes.Black, findMidPoint(prevIndex, currIndex));

            // Update the iterator
            prevIndex = currIndex;
        }
        pathCostBox.Text = String.Format("{0}", totalPathCost);
    }


////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////  Dijktra's Algorithm ////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////
/**
* This function will implement Dijkstra's Algorithm to find the shortest path to all the nodes
from the source node
* Time Complexity: O((|V| + |E|) log|V|) with a heap as it iterates over all the nodes and all
the edges and uses a queue to go
* through them where the heap queue has a worst case of log|V|. Whereas if the queue was
implemented with an array, the complexity
* would be O((|V|^2) since the queue has a worst case of |V| and |E| is upper bounded by
|V|^2 and so |V|^2 dominates.
* Space Complexity: O(|V|) as it creates arrays as big as the number of nodes in the graph
*/
    private List<int> Dijkstras(ref PriorityQueue queue, bool isArray)
    {
        // Create Queue to track order of points
        queue.makeQueue(points.Count);
        // Set up prev node list
        List<int> prev = new List<int>();
        List<double> dist = new List<double>();
        for (int i = 0; i < points.Count; i++)
        {
            prev.Add(-1);
            dist.Add(double.MaxValue);
        }

        // Initilize the start node distance to 0
        dist[startNodeIndex] = 0;

        // Update Priority Queue to reflect change in start point distance
        if (isArray)
            queue.insert(startNodeIndex, 0);
```

```
            else
                queue.insert(ref dist, startNodeIndex);

            // Iterate while the queue is not empty
            while (!queue.isEmpty())
            {
                // Grab the next min cost Point
                int indexOfMin;
                if (isArray)
                    indexOfMin = queue.deleteMin();
                else
                    indexOfMin = queue.deleteMin(ref dist);

                PointF u = points[indexOfMin];

                // For all edges coming out of u
                foreach (int targetIndex in adjacencyList[indexOfMin])
                {
                    PointF target = points[targetIndex];
                    double newDist = dist[indexOfMin] + computeDistance(u, target);
                    if (dist[targetIndex] > newDist)
                    {
                        prev[targetIndex] = indexOfMin;
                        dist[targetIndex] = newDist;
                        if (isArray)
                            queue.decreaseKey(targetIndex, newDist);
                        else
                            queue.decreaseKey(ref dist, targetIndex);
                    }
                }
            }
            return prev;
        }

////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////// Solve Button Function ////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////
        private void solveButton_Clicked()
        {
            // As default, solve the problem using the heap
            PriorityQueue queue = new PriorityQueueHeap();
            Stopwatch watch = Stopwatch.StartNew();
            List<int> pathHeap = Dijkstras(ref queue, false);
            watch.Stop();
```

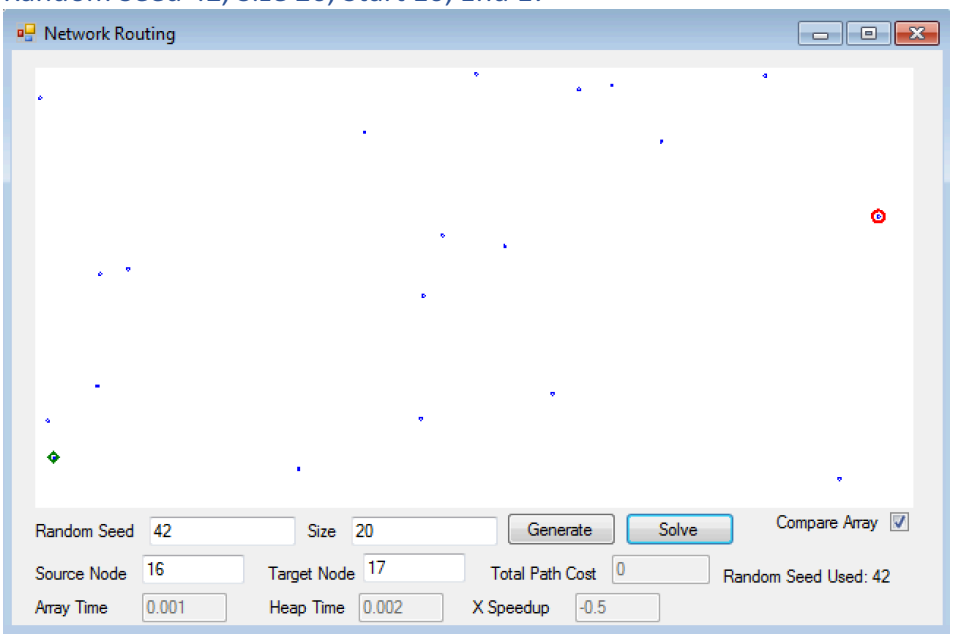
```
        // Calculate time for heap
        double heapTime = (double)watch.ElapsedMilliseconds / 1000;
        heapTimeBox.Text = String.Format("{0}", heapTime);

        List<int> pathArray = new List<int>();
        // Now If the Array box is checked, solve it again using an array, and compare answers
        if (arrayCheckBox.Checked)
        {
            PriorityQueue queueArray = new PriorityQueueArray();
            watch = Stopwatch.StartNew();
            pathArray = Dijkstras(ref queueArray, true);
            watch.Stop();

            // Calculate time for array
            double arrayTime = (double)watch.ElapsedMilliseconds / 1000;
            arrayTimeBox.Text = String.Format("{0}", arrayTime);
            differenceBox.Text = String.Format("{0}", (arrayTime - heapTime)/heapTime);

            // check if the two paths are the same
            for (int i = 0; i < pathArray.Count(); i++)
            {
                if (pathArray[i] != pathHeap[i])
                    Console.WriteLine("At index " + i + " pathArray was :" + pathArray[i] + " and
pathHeap was " + pathHeap[i]);
            }
        }

        // Draw the final minimum cost path
        drawPath(ref pathHeap, false);
        drawPath(ref pathArray, true);
    }
```

Space and Time Complexities:

The time and Space complexities are explained and measured above each function in the attached lab code. Please refer to the comments above the functions in the code.

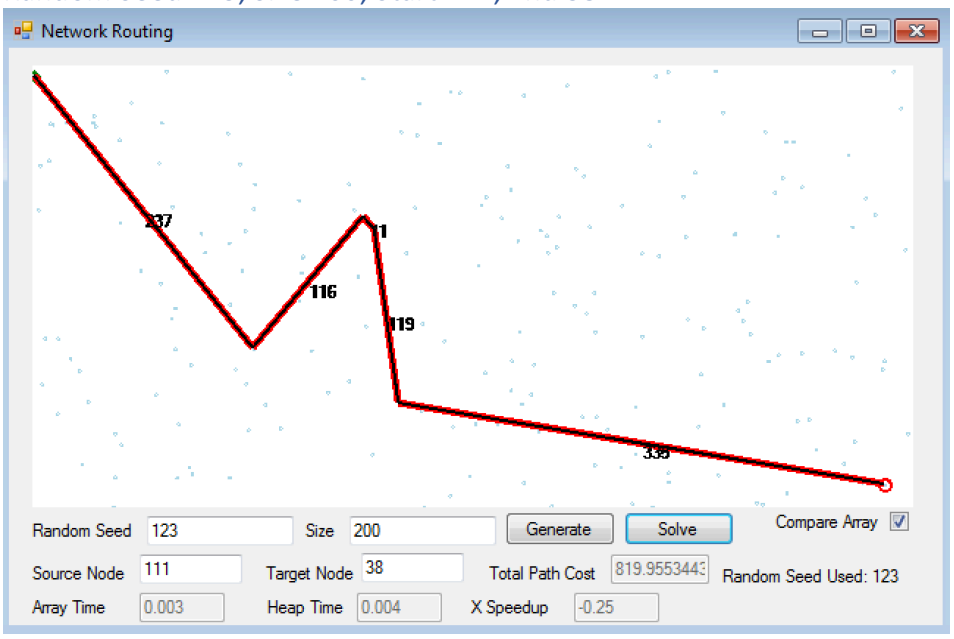
Screenshots Showing Correct Implementation:

a) Random Seed 42, Size 20, Start 16, End 17



This Screen shot shows that under the specified input, there is no path between the source and target Node.

b) Random Seed 123, Size 200, Start 111, End 38



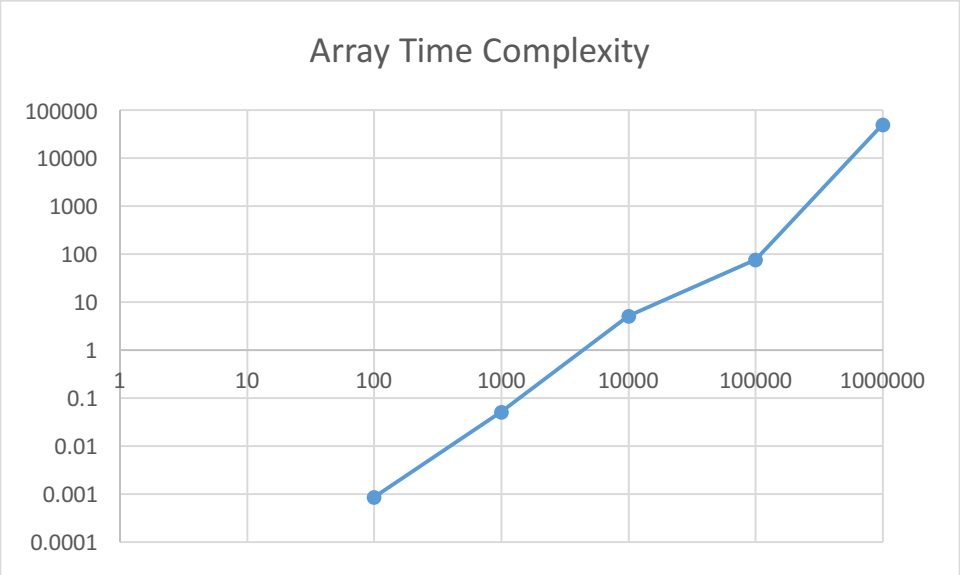
This Screen shot shows the path between the two nodes, the heap path is colored in black, and the array one is in red. This shows that the two paths match.

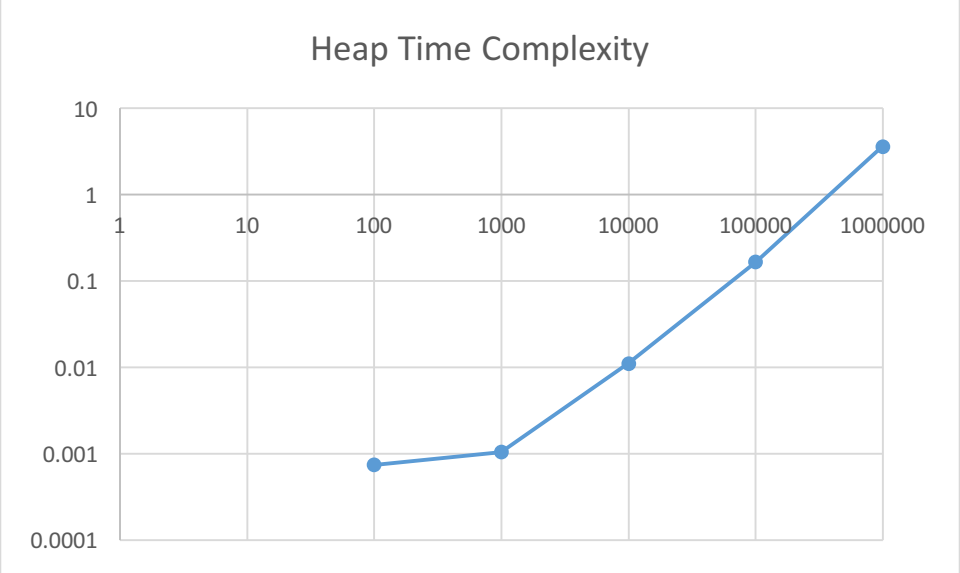
Empirical Analysis:

Raw Data:

	Array Time						Heap Time					
n	Seed = 4	Seed = 123	Seed = 1123	Seed = 7723	Seed = 5714	Average	Seed = 4	Seed = 123	Seed = 1123	Seed = 7723	Seed = 5714	Average
100	0.001	0.001	0.001	0.00061	0.000651	0.0008522	0.003	0.003	0.003	0.00000911	0.000108	0.00074342
1000	0.0487	0.05187	0.0528	0.0526	0.0485	0.050894	0.0009557	0.00119	0.00074	0.00136	0.0009514	0.00104
10000	5.086	5.067	5.067	5.087	5.0623	5.07386	0.00976	0.0112	0.0098	0.0113	0.0134	0.011092
100000	86.138	75.503	86.7077	75.496	58.972	76.563	0.1653	0.1657	0.1677	0.1734	0.1561	0.16564
1000000	NA	NA	NA	NA	NA	50,000	3.803	3.447	3.6308	3.6183	3.5479	3.6094

Graphing the n column with the averages from each container type, in logarithmic scale, I get:





The expected running time for the Array implementation is O(|V|^2), so theoretically for n = 1000, the running time should be 1,000,000. However, in our data, it is 0.0508. So there is a constant involved. If we let the constant be 0.0508/1,000,000 = approx. $\mathbf{5 * 10^{-8}}$
So now using this we can estimate the values for n = 1,000,000:

ArrayTime(n = 1,000,000) = k * (1,000,000)^2 = 50,000

Since the expected running time for the Array implementation is $O(|V|^2)$ which is much larger than the running time for the Heap implementation of O((|V| + |E|)*log|V|), it is natural that the array time values would scale in a much worse way that the heap ones would. For small values of n, the difference is practically unnoticeable, and in some cases, the array implementation was better. We should take into account that these data were based off of running the program on a virtual machine, so they might not be an accurate depiction of how the program functions. Furthermore, the way the time of the algorithms was measured was by using a System Stop watch to calculate system ticks and divide by the equivalence of ticks to seconds. Thus, there is even more chance of error.