Question 2

I realized this question needed an array to be made as well as a linked list. In the main method, I declared an integer variable “size” so you only have to change the size in one place rather than everywhere the number is used. I also created an array of size “size” and called a function called “insertionSort” which would take in the array we want to use, and the size of it. The function first creates an array of the chosen size and fills each index with a randomly generated number. The next thing it does is sort the array using the insertion sort method. We have two variables, one to hold the position in the array we want to be at (“trail”), and another to hold the value within the position we want (“temporary”). Using a for loop, we iterate through the array one by one. An example of one iteration would be for us to start at index 1 which is the second position in the array. We take trail to hold the index (1), and temporary holds the value in position 1. As long as “trail” is not a negative number (meaning within the bounds of the array) and the value at our position 1 is less than the previous position (0), then we put the value that is in the position 0 into our current index since it is greater. We decrement trail so the next test we can check the position even further back. We then put the value we had originally at position 1 into position 0.

The next thing we need to do is an iterative binary search and a recursive binary search. The idea behind a binary search is we are given a number we want to search for in a sorted array. Instead of starting from the beginning of the array and checking through each index, binary search goes straight to the middle of the array and checks if the number is greater or less than the middle value. If it is less, we disregard the greater half of the array and find the midpoint in the lesser half (the same goes for the opposite). In our function for iterative binary search, we pass in the array, the number we want to look for, and the size of the array. We declare 3 variables: left, right, and midpoint to keep track of the positions, respectively. The left will be set to 0 which is the beginning of the array, and right is set to size, which is the end of the array, and midpoint is the sum of the two divided by 2 to find the middle. We loop to continuously search, and the condition is as long as the right does not pass the left (meaning we checked all the spaces in between the right and left side) then we check the circumstances. The three conditions within the loop are if the search number is less than the middle, we change the right side to be one less than the midpoint, and we adjust the midpoint accordingly. The same happens if the search number is greater than the midpoint, but instead we change left to be one greater than the midpoint while right stays the same, and we adjust the midpoint. This is so we continue to half the array as long as we are still searching for the number within bounds of the array. The last condition is if the midpoint is equal to the search number, we just return the midpoint. Outside of this loop, however, means that we could not find the number.

Recursive binary search follows the same concept as iterative but is shorter because we want to call the function within itself. We take in the array, the number we are searching for, and the positions of the left and right of the array. This is important because for the iterative function, we needed to continuously change the left and right. We initialize the midpoint again and have the same loop condition. Our conditions within the loop start with a base case (where the solution happens). If the value in the midpoint is equal to the search number, we just return the midpoint. We then have the two conditions for when the search number is greater than or less than the midpoint. If it is less than, we recall the function, but change the parameters so that we have the array, search number, and a new left or right. If the number is less than the midpoint value, we change right to one less than the midpoint, and if the number is greater than the midpoint, we change left to be one greater than the midpoint.

The last thing we need to do is a linked-type binary search. In order to do this, we need to create a linked list. At the beginning of the code, I created a node class so we can make node objects, which are what makes up a linked list. The nodes will have data and will also have a pointer to the next node in the list. In the function called linkedListCreate to create the list, I instantiated a node called “head” because it will serve as the first node in the list. The function takes in the array we made from earlier and its size because we are just going to create new nodes and pass in the values in the array to the nodes (just to keep the testing consistent). We set the head data to the first value in the array and initialize a new node for the next node after the head. We then loop through the size of the array, put the array value at that index, then initialize a new node. Increment then continue putting the array values into new nodes that are linked to each other. This function returns the head node, and in main, I pass that head into another function for the linked binary search, which takes in a node, the search number, and the size. We initialize the left, right, and middle which we will call “index.” We loop to first find the index AKA the middle of the list. Next, we check the conditions, similarly to the iterative binary search, depending on if the search number is greater than or less than the index (adjust the right or left and the index).