CSC 241
Exercise 4

Write a $O(\mathrm{N} + \log \mathrm{N})$ **contains** method for the BST such that it is not recursive. The current provided BST **contains** method is recursive.
Use the following startup code for the method in the **BinarySearchTree** class:

```
public boolean nonRecContains(T target){
	LinkedQueue<T> q = new LinkedQueue<T>();
	inOrder(root, q);	//O(N)
	return q.contains(target);
}
```

Since the **LinkedQueue** calls its contains method, you must implement the contains method inside the **LinkedQueue** class. Here is the startup code for the contains method in the **LinkedQueue** class:

```
public boolean contains(T target){
		//return true if target is found and false if target is not found.
		T[] elements = (T[]) new Object[size()];
		for(int i = 0; i < elements.length; i++){  //O(N)
			elements[i] = (T) dequeue();
			System.out.println(elements[i]);
		}
		//the code for the O(log N) search here:
		T item;

}
```

Please note the use of the **compareTo** method on the type T data type requires the use of the following syntax, due to the unknow data type T:
`((Comparable<? super T>) target).compareTo(item)`
In the case above, T must be a comparable object, and if it has a super class, the '?' will resolve any data type binding issues during runtime.

The O(log N) search should involve cutting the elements array in half as each time the target isn't found.
Suppose you have the following array for **elements** and you are testing contains with 'S':

start **end = mid - 1** **mid** **start = mid + 1** **end**

0	1	2	3	4	5	6	7	8	9
'A'	'F'	'G'	'J'	'P'	'R'	'S'	'T'	'W'	'Z'

You should start by finding the starting and ending index of your search:
start = 0, end = elements.length;
Now compute the mid point:
mid = (end – start) / 2; *This is not exactly correct for your coding purposes, but gets the job done to explain how this algorithm works.
So, in this example, **mid** = 5.

Next, the **compareTo** method will be used to compare 'S' with 'R', because 'R' is the value at the 5th index. Since, 'S' is greater than 'R', the **compareTo** method will return a value greater than 0.

The algorithm needs to continue looking for 'S' in the right half of the array. If **compareTo** returns a negative value, then the algorithms will continue to look for 'S' in the left half. If the **compareTo** returns 0, then the target is found.

Now a new **mid** will be computed and the algorithm will continue to look for 'S' in a similar fashion. Note that **start** and **end** will be updated accordingly; and, the original calculation of **mid** should probably be adjusted for the loop structure of this algorithm to work properly.

Use the following driver code to test your method:

```java
public class BSTExample
{
  public static void main(String[] args)
  {
    BinarySearchTree<Character> example = new BinarySearchTree<Character>();

    example.add('P'); example.add('F'); example.add('S'); example.add('B');
    example.add('H'); example.add('R'); example.add('Y'); example.add('G');
    example.add('T'); example.add('Z'); example.add('W');

    System.out.println(example.nonRecContains('T'));
    System.out.println(example.nonRecContains('E'));

  }
}
```