I chose to use a queue for the page list because its insert and removal where both O(1) and it operates on a first in first out basis. The first in first out nature of this is perfect for the page list because the index will receive the page numbers in order, so they will not need to be sorted and they will be asked for in first in first out order. A short coming of the queue is that its worst-case search time is O(n) that means to find if a page number is in the page list it could take up to O(n). this is not too big of a deal because a page number can only have up to 4 page numbers.

```java
package proj5;
/** This is the BST ADT.  It should contain methods that allow it to
 *  insert new nodes, delete nodes, search, etc.
 *
 * @author Matthew Caulfield
 * @version 11/12/2017
 */
public class BinarySearchTree<T>
{
      //root is a node that is the root of a tree
      //insert and set key can both chang it
     private BSTNode<T> root;

     public BinarySearchTree() {
       root=null;
     }

     /**
      * inserts recursively.  I include this one so you can
      * make your own trees in your own testing class
      *
      * @param startingNode inserts into subtree rooted at staringNode
      * @param newNode node to insert
      * @return startingNode with newNode already inserted
      */
     private BSTNode<T> recursiveInsert(BSTNode<T> startingNode, BSTNode<T> newNode)
{
       if (startingNode == null) {
             return newNode;
       }
       else if (((Comparable) startingNode.key).compareTo(newNode.key) < 0) {
             startingNode.rlink = recursiveInsert(startingNode.rlink,newNode);
             return startingNode;
       }
       else {  // startingNode.key bigger than newNode.key, so newNode goes on left
             startingNode.llink = recursiveInsert(startingNode.llink,newNode);
             return startingNode;
       }
     }

     /**
      * @return returns whether or not the function is empty
      */
     public boolean isEmpty() {
       return(root == null);
```

```
  }


  /**
   * inserts recursively. Use this in your JUnit tests to
   * build a starting tree correctly
   *
   * @param newString String to insert
   */
  public void recursiveInsert(T newComparable){
    BSTNode<T> newNode = new BSTNode<T>(newComparable);
    root = recursiveInsert(root, newNode);
  }

  /**
   * Inserts a string at its correct node
   * traverses the tree iteratively
   * @param newString the string to be inserted
   */
  public void insert(T newComparable) {
    BSTNode<T> newNode = new BSTNode<T>(newComparable);
    BSTNode<T> currentNode = root;
    boolean isInserted = false;
    if(currentNode == null) {
           root = newNode;
    }
    else {
           while(!isInserted) {
                  if(((Comparable) newNode.key).compareTo(currentNode.key)>0) {
                        if(currentNode.rlink == null) {
                               currentNode.rlink = newNode;
                               isInserted = true;
                        }
                        else {
                               currentNode = currentNode.rlink;
                        }
                  }
                  else {
                         if(currentNode.llink == null) {
                         currentNode.llink = newNode;
                         isInserted = true;
                  }
                  else {
                         currentNode = currentNode.llink;
                  }
                  }
           }
    }
  }

/**
* deletes an element from the binary search tree in order
* @param victim is the target to be deleted
*/
  public void delete(T victim) {
```

```
	root = recursiveDelete(root, victim);
}

/**
 * deletes a victim from a subtree
 * and returns the new subtree
 * @param subroot
 * @param victim
 * @return new subtree
 */
private BSTNode recursiveDelete(BSTNode subroot, T victim) {
	if(subroot == null) {
		return null;
	}
	else if(((Comparable) subroot.key).compareTo(victim)>0) {
		subroot.llink = recursiveDelete(subroot.llink, victim);
	}
	else if(((Comparable) subroot.key).compareTo(victim)<0) {
		subroot.rlink = recursiveDelete(subroot.rlink, victim);
	}
	else {
		if(subroot.isLeaf()) {
			return null;
		}
		else if(subroot.rlink != null && subroot.llink == null) {
			return subroot.rlink;
		}
		else if(subroot.rlink == null && subroot.llink != null) {
			return subroot.llink;
		}
		else {
			BSTNode substitute = subroot.llink.getRightMostDescendant();
			T substituteKey = (T) substitute.getKey();
			subroot = recursiveDelete(subroot, (T) substituteKey);
			subroot.setKey(substituteKey);
			return subroot;
		}
	}
	return subroot;
 }

/**
 * Returns the key of an node from
 * the search tree when given an element that
 * is equivalent in compare to of the node
 * @param element that needs to find key
 * @return the key of the equivalent element
 */
public T getData(T element) {
	return (T) recursiveGetData(root, element).key;
}

/**
 * Recursively searches through a tree for a node and returns the
 * node if its key matches the target in compare to
 * @param startingNode of tree to search through
 * @param target to search for
 * @return matching node or null if no match
 */
private BSTNode recursiveGetData(BSTNode startingNode, T target) {
  if (startingNode == null) {
        return null;
  }
  else if(((Comparable) startingNode.key).compareTo(target)>0) {
        return recursiveGetData(startingNode.llink, target);
  }
  else if(((Comparable) startingNode.key).compareTo(target)<0) {
        return recursiveGetData(startingNode.rlink, target);
  }
  else {
        return startingNode;
  }
}

/** private helper method that searches the tree from a starting node
 * if the current node is null then the target is not in the tree if the
 * current node is the target return true. otherwise begin searching through a
 * subtree the right subtree if the target is better than the current node
 * the left tree if equal to or worse than the current node
 *
 * @param the target string
 * @param the starting node of the tree to search
 * @return whether or not the target was in the tree
 */
private boolean recursiveSearch(T target, BSTNode<T> startingNode) {
  if (startingNode == null) {
        return false;
  }
  if (((Comparable) startingNode.key).compareTo(target)== 0) {
        return true;
  }
  else if(((Comparable) startingNode.key).compareTo(target)<0) {
        return recursiveSearch(target, startingNode.rlink);
  }
  else{
       return recursiveSearch(target, startingNode.llink);
  }
}

/**
 * searches through a tree recursively returns true if the target
 * is in the tree
 * @param target
 * @return the boolean value of if the target is in the tree
 */
public boolean search(T target) {
  return recursiveSearch(target, root);

}
```

```java
/**Private helper function for the toString method
 * uses inorder traversal to make turn a binary search tree
 * to a string
 * @param currentString part of the tree as a string that the method
 * appends to
 * @param the starting node for the to string
 * @return a portion of the tree as a string
 */
private String toString(String currentString, BSTNode<T> startingPoint) {
   if(startingPoint != null) {
        currentString += "(";
        currentString = toString(currentString, startingPoint.llink);
        currentString += startingPoint.key;
        currentString = toString(currentString, startingPoint.rlink);
        currentString += ")";
   }
   return currentString;
}

/**
 *  this is the public toString method it returns a sting
 *  of the complete Binary Search Tree in the form
 *  (( A ) B ( C ))
 *  B is the parent of A (left kid) and C (right kid).
 *
 * @return the Binary Search Tree in string form
 */
public String toString() {
   return this.toString("", root);
}

/**
 * WARNING: CRAPPY METHOD!  I wish I had toString...
 *
 * Recursive helper method of print.
 * Uses inorder tree traversal algorithm.
 * @param N subroot of tree to print
 */
private void print(BSTNode<T> N)
{
    if (N != null) {     // stop recursing when N is null
      System.out.print("(");
      print(N.llink);
      System.out.print("  " + N + "  ");
      print(N.rlink);
      System.out.print(")");
    }
}


/**
 *  WARNING: CRAPPY METHOD!  I wish I had toString...
 *
 *  prints a parenthesized version of the tree that shows
```

```
     *  the subtree structure.  Example: (( A ) B ( C )) means
     *  B is the parent of A (left kid) and C (right kid).
     */
    public void print()
    {
        print(root);
        System.out.println();
    }
}
```

```
package proj5;


import static org.junit.Assert.*;


import org.junit.Test;


public class BinarySearchTreeTest {


	@Test
	/**
	 *  Testing toString for an empty BST
	 */
	public void testToStringEmpty() {
		BinarySearchTree aTree = new BinarySearchTree();
		assertEquals(aTree.toString(),"");
	}



	/**
	 * testing is empty
	 */
	@Test
	public void isEmpty() {
		BinarySearchTree aTree = new BinarySearchTree();
```

```
        assertEquals(true, aTree.isEmpty());
        aTree.insert("cat");
        assertEquals(false, aTree.isEmpty());
}

@Test
/**
 *  Testing toString for a BST with one node
 */
public void testToStringOne() {
        BinarySearchTree aTree = new BinarySearchTree();
        aTree.recursiveInsert("dog");
        assertEquals(aTree.toString(),"(dog)");
}

@Test
/**
 *  Testing toString for a BST with multiple node
 */
public void testToStringMulti() {
        BinarySearchTree aTree = new BinarySearchTree();
        aTree.recursiveInsert("dog");
        aTree.recursiveInsert("cat");
        aTree.recursiveInsert("mouse");
        aTree.recursiveInsert("hat");
        aTree.recursiveInsert("sock");
        assertEquals(aTree.toString(),"((cat)dog((hat)mouse(sock)))");
}
```

```
@Test
/**
 * testing search on an empty tree
 */
public void testSearchEmpty() {
	BinarySearchTree aTree = new BinarySearchTree();
	assertEquals(aTree.search(""), false);
	assertEquals(aTree.search("HI"), false);
}

@Test
/**
 * testing search on a tree with one node
 */
public void testSearchOne() {
	BinarySearchTree aTree = new BinarySearchTree();
	aTree.recursiveInsert("cat");
	assertEquals(aTree.search("cat"), true);
	assertEquals(aTree.search("dog"), false);
}

@Test
/**
 * testing search on a tree with multiple nodes
 */
public void testSearchMultiple(){
	BinarySearchTree aTree = new BinarySearchTree();
```

```
		aTree.recursiveInsert("dog");
		aTree.recursiveInsert("cat");
		aTree.recursiveInsert("mouse");
		aTree.recursiveInsert("hat");
		aTree.recursiveInsert("sock");
		assertEquals(aTree.search("cat"), true);
		assertEquals(aTree.search("dog"), true);
		assertEquals(aTree.search("mouse"), true);
		assertEquals(aTree.search("hat"), true);
		assertEquals(aTree.search("sock"), true);
		assertEquals(aTree.search("hello"), false);
	}

	/**
	 * testing insert on an empty tree
	 */
	@Test
	public void testInsertEmpty() {
		BinarySearchTree aTree = new BinarySearchTree();
		aTree.insert("dog");
		assertEquals(aTree.toString(),"(dog)");
	}

	/**
	 * testing insert on a tree with one node
	 */
	@Test
	public void testInsertOne() {
		BinarySearchTree aTree = new BinarySearchTree();
		aTree.insert("dog");
		aTree.insert("cat");
		assertEquals(aTree.toString(),"((cat)dog)");
	}

	/**
	 * testing insert on a tree with multiple nodes
	 * multiple times
	 */
	@Test
	public void testInsertMulti() {
		BinarySearchTree aTree = new BinarySearchTree();
		aTree.insert("dog");
		aTree.insert("cat");
		aTree.insert("mouse");
		aTree.insert("hat");
		aTree.insert("sock");
		assertEquals(aTree.toString(),"((cat)dog((hat)mouse(sock)))");
	}

	/**
	 * testing delete from an empty tree
	 */
	@Test
	public void testDeleteEmpty() {
		BinarySearchTree<String> aTree = new BinarySearchTree<String>();
		aTree.delete("cat");
		assertEquals(aTree.toString(), "");

	}

	/**
	 * testing delete an a tree with one node
	 */
	@Test
	public void testDeleteOneNode() {
		BinarySearchTree<String> aTree = new BinarySearchTree<String>();
		aTree.insert("cat");
		aTree.delete("cat");
		assertEquals("", aTree.toString());
	}

	/**
	 * testing delete for deleting leaf
	 */
	@Test
	public void testDeleteLeaf() {
		BinarySearchTree<String> aTree = new BinarySearchTree<String>();
		aTree.insert("cat");
		aTree.insert("art");
		aTree.insert("dog");
		aTree.delete("art");
		assertEquals("(cat(dog))", aTree.toString());
	}
```

```
/**
 * testing delete of one child
 */
@Test
public void testDeleteOneChild() {
        BinarySearchTree<String> aTree = new BinarySearchTree<String>();
        aTree.insert("cat");
        aTree.insert("art");
        aTree.insert("dog");
        aTree.insert("fog");
        aTree.delete("dog");
        assertEquals("((art)cat(fog))", aTree.toString());
}

/**
 * testing delete with two children
 */
@Test
public void testDeleteTwoKids() {
        BinarySearchTree<String> aTree = new BinarySearchTree<String>();
        aTree.insert("bat");
        aTree.insert("art");
        aTree.insert("dog");
        aTree.insert("cat");
        aTree.insert("fog");
        assertEquals("((art)bat((cat)dog(fog)))", aTree.toString());
        aTree.delete("dog");
```

```
		assertEquals("((art)bat(cat(fog)))", aTree.toString());
	}

	/**
	 * testing get data
	 */
	@Test
	public void testGetData() {
		BinarySearchTree<String> aTree = new BinarySearchTree<String>();
		aTree.insert("bat");
		aTree.insert("art");
		aTree.insert("dog");
		aTree.insert("cat");
		aTree.insert("fog");
		String aString = aTree.getData("dog");
		assertEquals("dog", aString);
	}
}
package proj5;
/**
 * Node for binary search tree can
 * only hold comparable
 * @author Matthew Caulfield
 * @version 11/12/17
 */
public class BSTNode <T>{

	public T key;
	public BSTNode<T> llink;
	public BSTNode<T> rlink;

	/**
	 * non-default constructor
	 * @param newKey string that node will hold
	 */
	public BSTNode(T newKey)
	{
		key = newKey;
```

```
          llink = null;
          rlink = null;
   }

   /**
    * returns key as printable string
    * @return the key
    */
   public String toString()
   {
          return key.toString();
   }

/**
 * returns the left most descendant of a node
 * @return the left most descendant
 */
public BSTNode<T> getLeftMostDescendant(){
          boolean stillLeftChild = true;
          BSTNode<T> currentNode = this;
          while(stillLeftChild) {
                 if(currentNode.llink == null) {
                        stillLeftChild = false;
                 }
                 else {
                        currentNode = currentNode.llink;
                 }
          }
          return currentNode;
}

/**
 * returns key
 * @return the key
 */
public T getKey() {
   return key;
}

/**
 * sets the key
 * @param newKey that key is set to
 */
public void setKey(T newKey) {
   key = newKey;
}

/**
 * returns the number of children a node has
 */
public int getNumChildren() {
       if(llink == null && rlink == null) {
              return 0;
       }
       else if(llink == null && rlink != null) {
            return 1;
        }
        else if(llink != null && rlink == null) {
            return 1;
        }
        else {
            return 2;
        }
    }

    /**
     * returns the right most descendant of a node
     * @return the right most descendant of a node
     */
    public BSTNode<T> getRightMostDescendant(){
        boolean stillRightChild = true;
        BSTNode<T> currentNode = this;
        while(stillRightChild) {
                if(currentNode.rlink == null) {
                        stillRightChild = false;
                }
                else {
                        currentNode = currentNode.rlink;
                }
        }
        return currentNode;
    }

    /**
     * checks if the node is a leaf
     * @return boolean expression of if it is a leaf
     */
    public boolean isLeaf() {
        return getNumChildren()==0;
    }

}
package proj5;

/**
 * Driver for the index maker project
 *
 * @author  Matthew Caulfield
 * @version 11/12/2017
 */
public class Client
{
    public static void main(String[] args)
    {
        makeIndex("C:/Users/Matt/eclipse-
workspace/Caulfieldproj5/src/proj5/uscons.txt");
    }

    /**
     * Makes an index out of fileName. Gradescope needs this function.
     *
     * @param fileName path to text file that you want to index
     */
    public static void makeIndex(String fileName) {
      Parser aParser = new Parser(fileName);
      aParser.parse();
    }
}
package proj5;
/**
 * The dictionary class stores all words
 * in alphabetical order
 * @author Matt
 *@version 11/12/2017
 */
public class Dictionary {
	//wordTree stores the tree in alphabetical order
	//can only be changed by insert
	BinarySearchTree<String> wordTree;
	Dictionary(){
		wordTree = new BinarySearchTree<String>();
	}

	/**
	 * inserts a string into the dictionary
	 * @param toInsert
	 */
	public void insert(String toInsert) {
		wordTree.recursiveInsert(toInsert);
	}

	/**
	 * returns the dictionary as a string
	 * @return dictionary as a string
	 */
	public String toString() {
		String indexString = wordTree.toString();
		String [] arrayOfIndex= indexString.split("\\(|\\)");
		int indexLength = arrayOfIndex.length;
		String toReturn = "";
		for(int i = 0; i<indexLength; i++) {
			if(!arrayOfIndex[i].equals("")) {
				toReturn+=arrayOfIndex[i] + "\n";
			}
		}
		return toReturn;
	}

	/**
	 * returns true if a target word is in the tree
	 * false otherwise
	 * @param target word to check for
	 * @return boolean for if the target is in the tree
	 */
	public boolean search(String target) {
		return wordTree.search(target);
	}
}
package proj5;

import static org.junit.Assert.*;

import org.junit.Test;

public class DictionaryTest {

	@Test
	public void testInsert() {
		Dictionary aDictionary = new Dictionary();
		aDictionary.insert("cat");
		assertEquals("cat\n", aDictionary.toString());
	}

	@Test
	public void testToStringEmpty() {
		Dictionary aDictionary = new Dictionary();
		assertEquals("", aDictionary.toString());
	}

	@Test public void testToStringMultiple() {
		Dictionary aDictionary = new Dictionary();
		aDictionary.insert("cat");
		aDictionary.insert("dog");
		aDictionary.insert("rat");
		aDictionary.insert("bat");
		assertEquals("bat\ncat\ndog\nrat\n", aDictionary.toString());
	}


	@Test
	public void testSearchEmpty() {
		Dictionary aDictionary = new Dictionary();
		assertEquals(false, aDictionary.search("cat"));
	}


	@Test
	public void testSearchMultiple() {
		Dictionary aDictionary = new Dictionary();
		aDictionary.insert("cat");
		aDictionary.insert("dog");
		aDictionary.insert("rat");
		aDictionary.insert("bat");
		assertEquals(true, aDictionary.search("cat"));
	}


}
```

```
package proj5;
/**
 * The Index Class which contains words and pagenumbers that they
 * appear on
 * the words are in alphabetical order
 * @author Matthew Caulfield
 * @version 11/12/2017
 */
public class Index {
	//Binary Search Tree of Index Elements which hold
	//a word and its associated page list
	BinarySearchTree<IndexElement> indexTree;

	public Index() {
		indexTree = new BinarySearchTree<IndexElement>();
	}
```

```java
/**
 * adds a word and the page number that it appears on to
 * the index
 * @param aWord the word
 * @param pageNum the page number
 */
public void insert(String aWord, int pageNum) {
      IndexElement anIndexElement = new IndexElement(aWord, pageNum);
      indexTree.recursiveInsert(anIndexElement);
}

/**
 * adds a page number to the page list of the associated word
 * @param target the word
 * @param pageNum the page number
 */
public void addPageNumber(String target, int pageNum) {
      IndexElement anIndexElement = getIndexElement(target);
      anIndexElement.addPageNumber(pageNum);
}

/**
 * returns true if the word is in the index
 * false otherwise
 * @param target the word that is being searched for
 * @return if the word is in the index
 */
public boolean search(String target) {
      IndexElement anIndexElement = new IndexElement(target, -1);
      return indexTree.search(anIndexElement);
}

/**
 * deletes an index element from the index
 * @param victim the word of the associated index
 */
public void delete(String victim) {
      IndexElement victemIndexElement = getIndexElement(victim);
      indexTree.delete(victemIndexElement);

}

/**
 * gets the index element associated with a string
 * @param target string of the index element its searching for
 * @return the found index element
 */
private IndexElement getIndexElement(String target) {
      IndexElement searchElement = new IndexElement(target, -1);
      IndexElement data = indexTree.getData(searchElement);
      return data;
}

/**
```

```
         * returns the index entry of a word as a string
         * @param targetWord word of associated entry
         * @return the index entry as a string in form word {page list}
         */
        public String stringOfIndexEntry(String targetWord) {
               IndexElement data = getIndexElement(targetWord);
               return data.toString();
        }

        /**
         *checks if the index contains a page number for
         *a word
         * @param element the word
         * @param target the page number
         * @return
         */
        public boolean containsPageNumber(String element, int target) {
               IndexElement data = getIndexElement(element);
               return data.containsPageNum(target);
        }

        /**
         * @return the index as a string with each element in form
         * word {page list}
         */
        public String toString() {
               String indexString = indexTree.toString();
               String [] arrayOfIndex= indexString.split("\\(|\\)");
               int indexLength = arrayOfIndex.length;
               String toReturn = "";
               for(int i = 0; i<indexLength; i++) {
                      if(!arrayOfIndex[i].equals("")) {
                             toReturn+=arrayOfIndex[i] + "\n";
                      }
               }
               toReturn = toReturn.trim();
               return toReturn;
        }

        /**
         * checks of the page list is full
         * @param the word of the associated page list
         * @return boolean expression of if the page list is full
         */
        public boolean isPageListFull(String target) {
               IndexElement anIndexElement = getIndexElement(target);
               return anIndexElement.isFull();
        }

}
package proj5;

/**
 * An Index Element that holds a string that is the word
 * and a page list associated to that word
 * @author Matthew Caulfield
 * @version 11/12/2017
 */
public class IndexElement<T> implements Comparable<T> {
	//the string that is the word for the index
	private String key;
	//the pages that the word appears on
	private Queue pageList;
	//the maximum number of pages it can appear on
	//before the page list is full
	private static int capacity = 4;
	//the number of pages in the page list
	private int size;
	public IndexElement(String newKey, int pageNumber){
		key = newKey;
		size = 0;
		pageList = new Queue(capacity);
		pageList.insert(pageNumber);
		size ++;
	}

	/**
	 * adds a page number to the page list
	 * @param pageNumber
	 */
	public void addPageNumber(int pageNumber) {
		pageList.insert(pageNumber);
		size++;
	}

	/**
	 * @return the number of pages in the page list
	 */
	public int getSize() {
		return size;
	}

	/**
	 * @return true if the number of pages is equal to
	 * or greater than the capacity otherwise returns false
	 */
	public boolean isFull(){
		return getSize() >= capacity;
	}

	/**
	 *
	 * @return the capacity of the queue
	 */
	public int getCapacity() {
		return capacity;
	}

	/**
	 * @return the key
	 */
	public String getKey(){
		return key;
	}


	/**
	 * @return the IndexElement as a String in form word {page list}
	 */
	public String toString() {
		String toReturn = key + " " + pageList.toString();
		return toReturn;
	}

	/**
	 * @return whether the page list contains a page number
	 */
	public boolean containsPageNum(int pageNum) {
		return pageList.contains(pageNum);
	}

	/**
	 * @param the object to be compared
	 * @return a negative integer, zero,
	 *  or a positive integer as this object is less than, equal to,
	 *  or greater than the specified object.
	 */
	public int compareTo(T other) {
		String otherKey = ((IndexElement) other).getKey();
		return key.compareTo(otherKey);
	}
}
```

```
package proj5;


import static org.junit.Assert.*;


import org.junit.Test;


public class IndexTest {


	@Test

	public void testToStringEmpty() {
        Index aIndex = new Index();
        assertEquals("", aIndex.toString());
    }

    @Test
    public void testToStringMultiple() {
        Index aIndex = new Index();
        aIndex.insert("bat", 1);
        aIndex.insert("cat", 1);
        aIndex.insert("dog", 2);
        aIndex.insert("rat", 3);
        assertEquals("bat {1}\ncat {1}\ndog {2}\nrat {3}", aIndex.toString());
    }

    @Test
    public void testaddPageNumber() {
        Index aIndex = new Index();
        aIndex.insert("bat", 1);
        aIndex.addPageNumber("bat", 3);
        aIndex.addPageNumber("bat", 4);
        aIndex.addPageNumber("bat", 7);
        assertEquals("bat {1, 3, 4, 7}", aIndex.toString());
    }

    @Test
    public void testContainsPageNumberTrue() {
        Index aIndex = new Index();
        aIndex.insert("bat", 1);
        aIndex.addPageNumber("bat", 3);
        aIndex.addPageNumber("bat", 4);
        aIndex.addPageNumber("bat", 7);
        assertEquals("bat {1, 3, 4, 7}", aIndex.toString());
        assertEquals(true, aIndex.containsPageNumber("bat", 1));
        assertEquals(true, aIndex.containsPageNumber("bat", 3));
        assertEquals(true, aIndex.containsPageNumber("bat", 4));
        assertEquals(true, aIndex.containsPageNumber("bat", 7));
}

@Test
public void testContainsPageNumberFalse() {
        Index aIndex = new Index();
        aIndex.insert("bat", 1);
        aIndex.addPageNumber("bat", 3);
        aIndex.addPageNumber("bat", 4);
        aIndex.addPageNumber("bat", 7);
        assertEquals("bat {1, 3, 4, 7}", aIndex.toString());
        assertEquals(false, aIndex.containsPageNumber("bat", 5));
}

@Test
public void testSearch() {
        Index aIndex = new Index();
        aIndex.insert("bat", 1);
        assertEquals(true, aIndex.search("bat"));
        assertEquals(false, aIndex.search("cat"));
}
```

```
@Test
public void testDelete() {
        Index aIndex = new Index();
        aIndex.insert("bat", 1);
        aIndex.insert("cat", 1);
        aIndex.insert("dog", 2);
        aIndex.insert("rat", 3);
        assertEquals("bat {1}\ncat {1}\ndog {2}\nrat {3}", aIndex.toString());
        aIndex.delete("dog");
        assertEquals("bat {1}\ncat {1}\nrat {3}", aIndex.toString());
}
@Test
public void testStringIndexElement() {
        Index aIndex = new Index();
        aIndex.insert("bat", 1);
        aIndex.insert("cat", 1);
        aIndex.insert("dog", 2);
        aIndex.insert("rat", 3);
        assertEquals("bat {1}", aIndex.stringOfIndexEntry("bat"));
}

@Test
public void testIsPageListFull() {
        Index aIndex = new Index();
        aIndex.insert("bat", 1);
        assertEquals(false, aIndex.isPageListFull("bat"));
        aIndex.addPageNumber("bat", 3);
```

```
                        assertEquals(false, aIndex.isPageListFull("bat"));

                        aIndex.addPageNumber("bat", 4);

                        assertEquals(false, aIndex.isPageListFull("bat"));

                        aIndex.addPageNumber("bat", 7);

                        assertEquals(true, aIndex.isPageListFull("bat"));

            }

}
```

```
package proj5;
/**
 * class that parses an input file and makes an Index
 * and dictionary for that file
 * @author Matthew Caulfield
 * @version 11/13/2017
 */
public class Parser {
      //a dictionary of words that have been seen more than
      //four times in alphabetical order
      private Dictionary aDictionary;
      //an index that stores words with their page numbers
      //in alphabetical order
      private Index anIndex;
      //reads in a text file
      private FileReader aFileReader;
      public Parser(String fileName){
            aDictionary = new Dictionary();
            anIndex = new Index();
            aFileReader = new FileReader(fileName);
      }

      /**
       * parses through a file and creates an index and dictionary
       * for the text file
       */
      public void parse() {
            int pageNumber = 1;
            String currentString = aFileReader.nextToken();
            while(currentString != null) {
                  if(currentString.equals("#")) {
                        pageNumber++;
                  }
                  else {
                        if(currentString.length()>2 &&
!aDictionary.search(currentString)) {
                              if(anIndex.search(currentString)) {
                                    if(!anIndex.containsPageNumber(currentString,
pageNumber)) {
	if(!anIndex.isPageListFull(currentString)) {

	anIndex.addPageNumber(currentString, pageNumber);
							}
							else {
								System.out.println("Deleting '"
+ anIndex.stringOfIndexEntry(currentString) + "' from index");
								anIndex.delete(currentString);

	aDictionary.insert(currentString);
							}
						}
					}
					else {
						anIndex.insert(currentString, pageNumber);
					}
				}
			}
			currentString = aFileReader.nextToken();
		}
		System.out.println(anIndex.toString());
		System.out.println(aDictionary.toString());
	}
}

package proj5;
/**
 * Queue class of integers only
 * operates on the first in first out principle
 * @author Matt
 *@version 11/12/2017
 */
public class Queue {
	//index of the first element in the queue
	//only remove will change this
	private int front;
	//index of the last element in the queue +1
	//only insert will change this
	private int rear;
	//the number of elements in the queue
	//only insert and remove will change it
	private int count;
	//how many ints the queue could potentially hold
	//the queue needs to bee made bigger when the
	//count is the capacity
	//only ensure capacity can change it
	private int capacity;
	//If the capacity is full the amount to increase the capacity
	//setCapacityIncrement can change it
	private int capacityIncrement;
	//array holding all integers in the queue
	private int[] itemArray;

	public Queue(int cap) {
        front = 0;
        rear = 0;
        count = 0;
        capacity = cap;
        capacityIncrement = 5;
        itemArray = new int[capacity];
}

/**
 * @return the number of elements in the queue
 */
public int size() {
        return count;
}

/**
 *
 * @return whether the queue is empty
 */
public boolean isEmpty() {
        return (size() == 0);
}

/**
 * inserts an integer into the back of the queue
 * @param element to be inserted
 */
public void insert(int element) {
        ensureCapacity();
        itemArray[rear] = element;
        count++;
        rear = (rear+1)%capacity;
}

/**
 * changes the capacity increment
 * @param num the new capacity increment
 */
public void setCapacityIncrement(int num) {
        capacityIncrement = num;
}

/**
 * removes and returns the first item in
 * the queue
 * @return the first item in the queue
 */
public int remove() {
        if(!isEmpty()) {
                int toRemove = itemArray[front];
                front = (front+1)%capacity;
                count--;
                return toRemove;
        }
        else {
```

```
			return -1;
		}

	}

	/**checks if the queue contains an integer
	 * @return true if it does contain false otherwise
	 */
	public boolean contains(int target) {
		if(front<rear) {
			for(int i = front; i <= rear; i++) {
				if(itemArray[i]==target) {
					return true;
				}
			}
		}
		else {
			for(int i = 0; i<= rear; i++) {
				if(itemArray[i]==target) {
					return true;
				}
			}
			for(int i = front; i<count; i++) {
				if(itemArray[i]==target) {
					return true;
				}
			}
		}
		return false;
	}

	/**
	 * ensures that the queue size will not be larger than
	 * the capacity if it will be makes the capacity larger by the
	 * capacity increment
	 *
	 */
	private void ensureCapacity() {
		if(count+1 == capacity) {
			int[] cloneArray = itemArray.clone();
			capacity += capacityIncrement;
			itemArray = new int[capacity];
			for(int i = 0; i<count; i++) {
				itemArray[i] = cloneArray[i];
			}
		}
	}

	/**
	 * @return the capacity
	 */
	public int getCapacity() {
		return capacity;
	}
```

```java
	/**
	 * @return the queue as a string in form {1, 2, 3, ..., etc}
	 */
	public String toString() {
		if(isEmpty()) {
			return "";
		}
		else {
			String toReturn = "{"+ itemArray[front];
			if(front<rear) {
				for(int i = front+1; i < rear; i++) {
					toReturn += ", " + itemArray[i];
				}
			}
			else {
				for(int i = front+1; i<count; i++) {
					toReturn += ", " + itemArray[i];
				}
				for(int i = 0; i< rear; i++) {
					toReturn += ", " + itemArray[i];
				}

			}
			toReturn += "}";
			return toReturn;
		}
	}

}
```