```java
/**
 *
 * A generic binary search tree that holds comparables.
 *
 *  *          * I affirm that I have carried out the attached academic endeavors
 *          with full academic honesty, in accordance with the Union College
 *          Honor Code and the course syllabus.
 *
 * @author Xavier
 * @version 03.02.17
 */
public class BinarySearchTree<E extends Comparable<E>> {
      private BSTNode<E> root;

      /**
       * Default constructor for BST, holds nothing until something is inserted
       */
      public BinarySearchTree() {
            root=null;
      }

      /**
       * Removes the node with given comparable
       * @param toRemove the comparable to remove
       * @param node The node to start at, usually root
       * @return the new tree that has had the given data removed

       */
    private BSTNode remove(E toRemove, BSTNode node){
        if( node == null ) {
            return node;
        }

        if( node.key.compareTo(toRemove) > 0 ) {
            node.lLink = remove( toRemove, node.lLink );
        }

        else if( node.key.compareTo(toRemove) < 0 ) {
            node.rLink = remove( toRemove, node.rLink);
        }

        else if(node.lLink != null && node.rLink != null )
        {
            node.key = inOrderSuccessor(node).key;
            node.rLink = remove( (E)node.key, node.rLink );
        }
        else {

            if( node.lLink != null ) {
                  node=node.lLink;
            }
            else {
                  node=node.rLink; //Can be the leaf becoming null
            }
        }

        return node;
    }
```

```java
/**
 * Finds the in order successor of the given node
 * @param start The node to start at
 * @return The in order successor
 */
 private BSTNode inOrderSuccessor(BSTNode start) {
       return findMin(start.rLink);
 }

 /**
  * Finds the minimum node after the given value
  * @param start The node to start looking at
  * @return The smallest node
  */
 private BSTNode findMin(BSTNode start) {
       if(start.lLink==null) {
             return start;
       }
       else {
             return findMin(start.lLink);
       }
 }


 /**
  * Removes the node with the given data and restructures the tree
  * accordingly
  * @param toRemove The data you want to remove.
  *
  */
 public void remove(E toRemove) {
       root = remove(toRemove, root);
 }


 /**
  * Inserts information into the tree
  * @param newValue
  */
 public void insert(E newValue) {
       root = insert(root,newValue);
 }

 /**
  * Inserts the given data into the tree
  *
  * @param whatever you want to put in the tree
  * @return The new tree with the inserted node
  **/
 private BSTNode insert(BSTNode<E> node, E data) {
       if (node == null) {
             return new BSTNode<E>(data);
       }
       else if ( data.compareTo(node.key) >= 0 ) {
             node.rLink = insert(node.rLink, data);
             return node;
       }

       else {
```

```
            node.lLink = insert(node.lLink, data);
            return node;
      }
}



/**
 * Searches for a node with specific data and returns it. Returns null if
 * its not found.
 *
 * @param node
 *            The node to start at (usually root)
 * @param value
 *            The data that you're looking for
 * @return The node, or null
 */
private BSTNode<E> search(BSTNode<E> node, E value) {

      if (node == null) {
            return null;
      }

      if (node.key.compareTo(value) == 0) {
            return node;
      } else {
            BSTNode left = search(node.lLink, value);
            BSTNode right = search(node.rLink, value);

            if (left != null) {
                  return left;
            } else {
                  return right;
            }
      }
}

/**
 * Searches for some data and returns its data, or null if the node is not
 * found
 *
 * @param data
 *            Whatever you want to find
 * @return The data, or null
 */
public E search(E data) {
      BSTNode toReturn = search(root, data);
      if(toReturn!=null) {
            return (E)toReturn.key;
      }
      else {
            return null;
      }
}



/**
 * Counts how many nodes there are in the tree
```

```
	 *
	 * @return Number of nodes in the tree
	 */
	public int getSize() {
		return getSize(root);
	}

	/**
	 *
	 * @param N The node to start at, usually root
	 * @return The size of the tree
	 */
	private int getSize(BSTNode N) {
		if (N==null) {
			return 0;
		}
		else {
			return getSize(N.lLink) + getSize(N.rLink) + 1;
		}
	}



	/**
	 * A general toString method
	 *
	 * @return a string off the entire tree including parens to denote children
	 */
	public String toString() {
		return stringNodes(root);
	}

	/**
	 * Returns a string off the entire tree including parens to denote children
	 *
	 * @param N
	 *            the node to start at (usually root)
	 * @return The string version of all the nodes.
	 */
	private String stringNodes(BSTNode<E> N) {
		String toReturn = "";
		if (N != null) {
			toReturn += "(";
			toReturn += stringNodes(N.lLink);
			toReturn += "  " + N.key.toString() + " ";
			toReturn += stringNodes(N.rLink);
			toReturn += ")";

		}
		return toReturn;
	}



	/**
	 * Returns just the values in the tree in order as a string (no visualization
of the trees shape)
	 * @return just the values in the tree in order as a string
	 */
```

```
	public String sortedString() {
		return sortedString(root);
	}


	/**
	 * Returns just the values in the tree in order as a string (no visualization
of the trees shape)
	 * @param N The node to start at
	 * @return just the values in the tree in order as a string
	 */
	private String sortedString(BSTNode<E> N) {
		String toReturn = "";
		if (N != null) {
			toReturn += sortedString(N.lLink);
			toReturn += N.key.toString() + "\n";
			toReturn += sortedString(N.rLink);
		}
		return toReturn;
	}

}
```

```java
/**
 * A node for a Binary Search Tree
 * Holds comparables
 * @author xavier
 */
public class BSTNode<E extends Comparable<E>>
{
    public E key;
    public BSTNode lLink;
    public BSTNode rLink;


    /** Non-default constructor
     * Makes a node for an ADT to hold
     * has two child pointers
     *
     * @param data is the comparable you want stored in the node.
     */
    public BSTNode(E data)
    {
        this.key = data;
        this.lLink = null;
        this.rLink = null;

    }


    /**
     * @return Returns the toString of the data in this node.
     */
    public String toString()
    {
      if(this.key!=null)
            return key.toString();
        return null;
    }



}
```

```
/**
 *
 */

/**
 * @author xavier
 *
 */
public class BSTTesting {
	public static void main(String[] args) {

		Testing.setVerbose(true);
		Testing.startTests();

		testInsertsLL();
		testRemoveLL();

		testSearch();
		testRemove();

		testDocument();

		testDocumentsInTree();

		testIndexer();

		testConsecutive();

		Testing.finishTests();

	}

	private static void testSearch() {
		BinarySearchTree tree = fillTree();

		Testing.assertEquals("Exists", "AAA", tree.search("AAA"));
		Testing.assertEquals("Exists", "BBB", tree.search("BBB"));
		Testing.assertEquals("does not exist", null, tree.search("ZZZ"));
		Testing.assertEquals("does not exist", null, tree.search("Ayyyy"));
		Testing.assertEquals("does not exist", null, tree.search("CCCCC"));
	}


	private static void testRemove() {
		BinarySearchTree tree = fillTree();


		tree.remove("CCB");
		Testing.assertEquals("After removal with no kids", "((  AAA (  BBA ))
BBB ((  CAA )  CCC (  DDD )))", tree.toString());

		tree.remove("AAA");
		Testing.assertEquals("After removal with one kid", "((  BBA )  BBB
((  CAA )  CCC (  DDD )))", tree.toString());

		tree.remove("CCC");
		Testing.assertEquals("After removal with no kids", "((  BBA )  BBB
((  CAA )  DDD ))", tree.toString());
```

```
            tree.insert("EEE");
            tree.remove("CAA");
            Testing.assertEquals("After removal with no kids", "((  BBA )  BBB
(  DDD (  EEE )))", tree.toString());

            tree.remove("BBB");
            Testing.assertEquals("After removal of root with two children",
"((  BBA )  DDD (  EEE ))", tree.toString());

            tree.remove("EEE");
            Testing.assertEquals("After removal of child with no kids", "((  BBA )
DDD )", tree.toString());

            tree.remove("DDD");
            Testing.assertEquals("After removal of root with one kid", "(  BBA )",
tree.toString());

            tree.remove("BBA");
            Testing.assertEquals("After removing last node", "", tree.toString());

            tree.insert("Duplicate");
            tree.insert("Duplicate");
            Testing.assertEquals("After removal of child with no kids",
"(  Duplicate (  Duplicate ))", tree.toString());

      }


      private static void testDocument() {
            Document doc = new Document("Cat",1);
            Testing.assertEquals("addition of same page", false,
doc.addInstance(1));
            Testing.assertEquals("addition of new page", false,
doc.addInstance(2));
            Testing.assertEquals("addition of same page", false,
doc.addInstance(2));
            Testing.assertEquals("addition of new page", false,
doc.addInstance(3));
            Testing.assertEquals("addition of same page", false,
doc.addInstance(3));
            Testing.assertEquals("addition of new page", false,
doc.addInstance(4));
            Testing.assertEquals("addition of new page", true, doc.addInstance(5));


            Testing.assertEquals("tests getWord", "cat", doc.getWord());
            Testing.assertEquals("tests toString", "cat: 1-5", doc.toString());

            Document doc2 = new Document("Bat",117);
            Document doc3 = new Document("Bat",7);

            Testing.assertEquals("tests compareTo of documents", 1,
doc.compareTo(doc2));
            Testing.assertEquals("tests compareTo of documents", -1,
doc2.compareTo(doc));

            Testing.assertEquals("tests compareTo of documents", 0,
doc2.compareTo(doc3));
```

```
            Testing.assertEquals("tests compareTo of documents", 1,
doc.compareTo("Bat"));
            Testing.assertEquals("tests compareTo of documents", 0,
doc.compareTo("Cat"));
            Testing.assertEquals("tests compareTo of documents", 0,
doc.compareTo("cat"));

      }


      private static void testDocumentsInTree() {
            BinarySearchTree<Document> index=new BinarySearchTree<Document>();
            Document doc1 = new Document("Joules",7);
            Document doc2 = new Document("damnation",100);
            Document doc3 = new Document("bork",4);
            Document doc4 = new Document("sandwich",1);
            Document doc5 = new Document("ruffles",3);
            Document doc6 = new Document("zubat",8);
            Document doc7 = new Document("Alfalfa",2);


            Document doc8 = new Document("damnation",12);
            Document doc9 = new Document("Alfalfa",77);
            Document doc10 = new Document("Joules",8);


            Testing.assertEquals("tests getSize with nothing in it", 0,
index.getSize());

            index.insert(doc1);
            index.insert(doc2);
            index.insert(doc3);
            index.insert(doc4);
            index.insert(doc5);
            index.insert(doc6);
            index.insert(doc7);


            Testing.assertEquals("toString of tree full of documents",
"((((  alfalfa: 2 )  bork: 4 )  damnation: 100 )  joules: 7 ((  ruffles: 3 )
sandwich: 1 (  zubat: 8 )))", index.toString());


            Testing.assertEquals("tests search", doc8.getWord(),
index.search(doc8).getWord());
            //Testing.assertEquals("tests search", doc8.getWord(),
index.search("damnation").getWord());



            Testing.assertEquals("tests getSize", 7, index.getSize());



            index.remove(doc9);
            System.out.println("toString" + index.toString());
```

```
            Testing.assertEquals("toString of tree full of documents", "(((  bork:
4 )  damnation: 100 )  joules: 7 ((  ruffles: 3 )  sandwich: 1 (  zubat: 8 )))",
index.toString());

            index.remove(doc10);
            Testing.assertEquals("toString of tree full of documents after
removal", "(((  bork: 4 )  damnation: 100 )  ruffles: 3 (  sandwich: 1 (  zubat:
8 )))", index.toString());

      }


      private static void testInsertsLL() {
      Testing.testSection("Tests insertAtHead, insertAtTail, and toString");
      LinkedList<String> list=new LinkedList<String>();

      LinkedList<String> list2=new LinkedList<String>();

      LinkedList<Integer> intList=new LinkedList<Integer>();



      list.insertAt(0, "One");
      Testing.assertEquals("Tests addition in empty list at start", "One",
list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 1,
list.getLength());

      list.insertAt(5, "Two");
      Testing.assertEquals("Tests addition at location longer than length", "One,
Two", list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 2,
list.getLength());

      list.insertAt(1, "Three");
      Testing.assertEquals("Tests addition between nodes", "One, Three, Two",
list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 3,
list.getLength());

      list.insertAt(0, "Four");
      Testing.assertEquals("Tests addition at start", "Four, One, Three, Two",
list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 4,
list.getLength());

      list.insertAt(-6, "Five");
      Testing.assertEquals("Tests addition at negative index", "Five, Four, One,
Three, Two", list.toString());

      Testing.assertEquals("Tests addition in empty list capacity", 5,
list.getLength());
      list.insertAt(6, "Six");
      Testing.assertEquals("Tests addition at end", "Five, Four, One, Three, Two,
Six", list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 6,
list.getLength());

      list2.insertAt(0, "a");
```

```
      list2.insertAt(1, null);
      list2.insertAt(2, "b");


      Testing.assertEquals("Tests addition between nodes", "a, null, b",
list2.toString());


      intList.insertAt(0, 1);
      Testing.assertEquals("Tests addition in empty list at start", "1",
intList.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 1,
intList.getLength());

    }


    private static void testRemoveLL() {
      Testing.testSection("Tests insertAtHead, insertAtTail, and toString");
      LinkedList<String> list=new LinkedList<String>();
      list.insertAt(10, "One");
      list.insertAt(10, "Two");
      list.insertAt(10, "Three");
      list.insertAt(10, "Four");
      Testing.assertEquals("Just checking", "One, Two, Three, Four",
list.toString());
      Testing.assertEquals("Tests addition in empty list capacity", 4,
list.getLength());

      Testing.assertEquals("Test removal of last", "Four", list.removeAt(3));
      Testing.assertEquals("Test removal of last", "One, Two, Three",
list.toString());
      Testing.assertEquals("Tests capacity after removal", 3, list.getLength());


      Testing.assertEquals("Test removal of first", "One", list.removeAt(0));
      Testing.assertEquals("Test removal of first", "Two, Three", list.toString());
      Testing.assertEquals("Tests capacity after removal", 2, list.getLength());

      Testing.assertEquals("Test removal of first", null, list.removeAt(-5));
      Testing.assertEquals("Test removal of first", "Two, Three", list.toString());
      Testing.assertEquals("Tests capacity after removal", 2, list.getLength());


      Testing.assertEquals("Test removal of first", null, list.removeAt(5));
      Testing.assertEquals("Test removal of first", "Two, Three", list.toString());
      Testing.assertEquals("Tests capacity after removal", 2, list.getLength());
    }


      private static void testIndexer() {


            Indexer index = new Indexer("src/proj7_input.txt");
            index.interpret();

            Indexer index2 = new Indexer("src/uscons.txt");
            index2.interpret();
```

```
		Indexer index3 = new Indexer("src/Test.txt");
		index3.interpret();


	}


	private static void testConsecutive() {
		Document doc = new Document("Cat",1);
		doc.addInstance(1);
		doc.addInstance(2);
		doc.addInstance(3);
		doc.addInstance(4);
		doc.addInstance(5);
		doc.addInstance(7);
		doc.addInstance(8);
		doc.addInstance(9);
		doc.addInstance(11);



		Testing.assertEquals("tests toString", "cat: 1-5, 7-9, 11",
doc.toString());


	}







	private static BinarySearchTree fillTree() {
		BinarySearchTree tree = new BinarySearchTree<String>();
		tree.insert("BBB");
		tree.insert("CCC");
		tree.insert("DDD");
		tree.insert("AAA");
		tree.insert("BBA");
		tree.insert("CAA");
		tree.insert("CCB");
		return tree;

	}

}
```

```java
/**
 * An ADT that holds a single string and a list of page numbers that the string occurs on.
 * There isn't really a use to this besides making an index for a document
 *
 * There is a preset limit to the number of pages this will hold before returning that it is full.
 *
 * @author xavier
 *
 */
public class Document implements Comparable<Document> {

	private String word;
	private LinkedList<Integer> pages;
	private final int MAX_PAGES=5;

	public Document(String toStore, int page) {
		word=toStore;

		pages=new LinkedList<Integer>();
		pages.insertAt(pages.getLength(), page);
	}

	/**
	 * Adds an instance of the word, returns true if the instance has reached the limit
	 * There will never be duplicate pages
	 * @param page The page number
	 * @return True or false based on if the max of this word has been reached
	 */
	public boolean addInstance(int page) {

		if(pages.getData(pages.getLength()-1)!=page) { //If most recent page isn't the same as the new page, because they will be inserted in order

			pages.insertAt(pages.getLength(), page);

			if(pages.getLength()>=MAX_PAGES) {
				return true;
			}
		}

		return false; //I tried putting this in an else{}, but eclipse wouldn't allow it for some reason



	}

	/**
	 * Returns the word of this document
	 * @return The word
	 */
	public String getWord() {
		return this.word.toLowerCase();
	}
```

```java
	/**
	 * Compares the words of the two documents
	 * @param other The other document to be compared to
	 * @return <0 if smaller than other, 0 if other, >0 if greater
	 */
	public int compareTo(Document other) {
		return this.word.compareToIgnoreCase(other.getWord());
	}



	/**
	 * Compares the word of this document to a string
	 * @param other The String to be compared to
	 * @return <0 if smaller than other, 0 if other, >0 if greater
	 */
	public int compareTo(String other) {
		return this.word.compareToIgnoreCase(other);
	}


	/**
	 * Counts how many pages the word appears consecutively on
	 * @param index The page to start on
	 * @return The number of sequencial pages it apears on
	 */
	private int sequenceCheck(int index) {
		if(pages.getData(index+1)!=null && pages.getData(index)
+1==pages.getData(index+1)) {
			return sequenceCheck(index+1) + 1;
		}
		else {
			return 0;
		}
	}

	/**
	 * returns the word and the pages it appears on, if there are sequential
pages, it shows them as e.g 1-3
	 * @return The word and the pages it appears on
	 */
	public String toString() {
		String toReturn="";
		int length=pages.getLength();
		int start=pages.getData(0);
		int seq=0;



		for(int i=0;i<length;i++) {
			seq=sequenceCheck(i);

			if(seq>1) {
				toReturn+=pages.getData(i)+"-"+((int)pages.getData(i)+
(int)seq)+", ";
				i=i+seq;
			}
			else {
```

```
                        toReturn+=pages.getData(i)+", ";
                    }
                }

                return this.word.toLowerCase()  + ": " + toReturn.substring(0,
toReturn.length() - 2);
        }

}
```

```
/**
 * A FileReader object will read tokens from an input file.  The name of
 * the input file is given when the constructor is run.  The lone method,
 * nextToken(), will return the next word in the file as a String.
 *
 * @author Chris Fernandes
 * @version 2/27/17
 */
import java.io.*;

public class FileReader
{

   private StreamTokenizer st;     //file descriptor

   /**
    * @param fileName path to input file. Can either be a full
    *  path like "C:/project7/proj7_input.txt" or a relative one like
    *  "src/proj7_input.txt" where file searching begins inside
    *  the project folder.
    *  Be sure to use forward slashes to specify folders.  And
    *  don't forget the quotes (it's a String!)
    */
   public FileReader(String fileName)
   {
      try {
      st = new StreamTokenizer(
              new BufferedReader(
                 new InputStreamReader(
                    new FileInputStream(fileName))));
      } catch (IOException e) {}
      st.resetSyntax();                  // remove default rules
      st.ordinaryChars(0, Character.MAX_VALUE); //turn on all chars
      st.wordChars(65,90);
      st.wordChars(97,122);       //make the letters into tokens
      st.whitespaceChars(0, 34);
      st.whitespaceChars(36,64);  //make everything else except "#" into
      st.whitespaceChars(91,96);  //whitespace
      st.whitespaceChars(123, Character.MAX_VALUE);
   }

   /**
    * returns the next token as a String. Possible tokens are
    * words, the pound symbol "#" signifying the end of a page,
    * and null which is returned when the end of the file is reached.
    *
    * @return a word, "#", or null
    */
   public String nextToken()
```

```
    {
        try
        {
        while (st.nextToken() != st.TT_EOF) {
            if (st.ttype < 0)
            {
                return st.sval;
            }
            else
            {
                return String.valueOf((char)st.ttype);
            }
        }
        return null;
        } catch (IOException e) {}
        return "error";
    }
}
```

```
/**
 * Given an input text, goes though and indexs the words in the text. It does not index words that are too common, and instead places them in a dictionary
 * It also does not document or index words shorter than three letters
 *
 */

/**
 * @author xavier
 *
 */
public class Indexer {

	private FileReader r;
	private BinarySearchTree dictionary;
	private BinarySearchTree index;
	private final int MIN_WORD_LENGTH=3;


	private String removed;

	/**
	 * Constructor for the indexer, which takes the source file as the input
	 * @param text The source file.
	 */
	public Indexer(String text) {
		r=new FileReader(text);
		dictionary=new BinarySearchTree<String>();
		index=new BinarySearchTree<Document>();

		removed="";
	}







	/**
	 * Reads through the text and sorts the words in the appropriate manor
	 */
	public void interpret() {
		int pageNumber=1;
		String tmp = r.nextToken();

		while(tmp!=null) {
			if(tmp.equals("#")) {
				pageNumber++;
```

```
                        }
                        else if (tmp.length()>=MIN_WORD_LENGTH &&
searchDictionary(tmp)==null) {
                                toIndex(tmp, pageNumber);
                        }
                        tmp=r.nextToken();
                }
                System.out.println("Dictionary:\n" + dictionary.sortedString());
                System.out.println("Index:\n" + index.sortedString());
        }


        /**
         * Returns the Document with the given word, or null if not found
         * @param word What to search the dictionary for
         * @return The document containing the word, or null
         */
        private String searchDictionary(String word) {
                return (String)dictionary.search(word);
        }


        /**
         * Returns the Document with the given word, or null if not found
         * @param word What to search the index for
         * @return The document containing the word, or null
         */
        private Document searchIndex(String word) {
                Document forSearch = new Document(word, 0);
                return (Document)index.search(forSearch);
        }


        /**
         * Handles checking if the data is already in the index, and if not inserting it there.
         * @param input
         * @param pageNumber
         */
        private void toIndex(String input, int pageNumber) {
                Document tmpDoc=searchIndex(input);
                if(tmpDoc!=null) { //if already in index

                        if(tmpDoc.addInstance(pageNumber)) { //If the pagelist is full
                                System.out.println("removed " + tmpDoc.toString());
                                index.remove(tmpDoc);
                                removed +=input+"\n";
                                dictionary.insert(input);
                        }
                }
```

```
            else {

                insert(input, pageNumber);
            }
        }


        /**
         * Inserts a word with the given stuff to the index
         * @param word The word
         * @param pageNumber The pagenumber
         */
        private void insert(String word, int pageNumber) {
            Document toInsert=new Document(word, pageNumber);
            index.insert(toInsert);
        }




}
```

```java
/**
 * Linked List is a collection of data nodes. All methods here relate to how one
 * can manipulate those nodes.
 *
 * @author Xavier
 * @version 02.09.17
 *
 *          * I affirm that I have carried out the attached academic endeavors
 *          with full academic honesty, in accordance with the Union College
 *          Honor Code and the course syllabus.
 */
public class LinkedList<E extends Comparable<E>> {
	private int length; // number of nodes
	private ListNode<E> firstNode; // pointer to first node

	public LinkedList() {
		length = 0;
		firstNode = null;
	}


	/**
	 * Inserts at specified location
	 * @param The index at which to insert, starting at 0
	 * @param The thing to insert
	 */
	public void insertAt(int place, E toInsert) {
		ListNode<E> newNode = new ListNode<E>(toInsert);

		if (getLength() == 0) {
			firstNode = newNode; //If theres nothing in the list, insert at
start
		}

		else {
			ListNode<E> n;
			n = firstNode;
			int index = 0;
			while (index != (this.getLength()-1) && index < (place - 1)) {
//goes until right before place or end
				n = n.next;
				index++;
			}

			if (index == this.getLength()) { //adds to end if place is above
end
				n.next = newNode;
			}
			else if(place<=0) { //if place is 0 or less, insert at start
				newNode.next=firstNode; //
				firstNode=newNode;
			}

			else { //n.next is 'place', adds before place.
				newNode.next = n.next;
				n.next = newNode;
			}
		}
```

```
			length++;
		}

		/**
		 * Removes the data from the given location and returns it
		 * @param The location to remove, starting at 0
		 * @return The removed information
		 */

		public E removeAt(int place) {
			if(place<0 || place>=this.getLength())
				return null;

			ListNode<E> n;
			n = firstNode;
			E toReturn;

			if(place==0) { //If its the first node
				toReturn=n.data;
				firstNode=n.next;
				length--;
				return toReturn;
			}

			int index = 0;
			while (index != (this.getLength()-1) && index < (place - 1)) { //Goes
until before the location
				n = n.next;
				index++;
			}

			toReturn = (E)(n.next.data); //returns the data at the place
			n.next=n.next.next;


			length--;
			return toReturn;
		}


		/**
		 * Turn entire chain into a string
		 *
		 * @return return linked list as printable string of format
		 *         (string, string, string)
		 */
		public String toString() {
			String toReturn = "";
			ListNode<E> n;
			n = firstNode;
			while (n != null) {
				toReturn = toReturn + n.toString();
				n = n.next;
				if (n != null) {
					toReturn = toReturn + ", ";
				}
			}
			toReturn = toReturn + "";
			return toReturn;
```

```
	}

	/**
	 * getter for number of nodes in the linked list
	 *
	 * @return length of LL
	 */
	public int getLength() {
		return length;
	}

	/**
	 * Gets the information from the specified location
	 * @param The index of what to get, starting at 0
	 * @return the information
	 */
	public E getData(int place) {
		if(place<0 || place>=this.getLength())
			return null;

		ListNode<E> n;
		n = firstNode;
		E toReturn;
		if(place==0) {
			toReturn=n.data;
			return toReturn;
		}

		int index = 0;
		while (index != (this.getLength()-1) && index < (place - 1)) {
			n = n.next;
			index++;
		}
		toReturn = (E)(n.next.data);
		return toReturn;
	}

	/**
	 * Clears the linked list of everything
	 */
	public void clear() {
		firstNode=null;
		length=0;
	}



}
```

```java
/**
 * ListNode is a building block for a linked list of data items
 *
 * This is the only class where I'll let you use public instance variables.
 * It's so we can reference information in the nodes using cascading dot
 * notation, like
 *          N.next.data instead of
 *          N.getNext().getData()
 *
 * @author C. Fernandes and G. Marten
 * @version 2/6/2012
 */
public class ListNode<E extends Comparable<E>>
{
    public E data;      // a "reservation" of the conference room
    public ListNode next;   // pointer to next node

    /** Non-default constructor
     *
     * @param String a reservation you want stored in this node
     */
    public ListNode(E String)
    {
        this.data = String;
        this.next = null;
    }

    // if you say "System.out.println(N)" where N is a ListNode, the
    // compiler will call this method automatically to print the contents
    // of the node.  It's the same as saying "System.out.println(N.toString())"
    public String toString()
    {
      if(this.data!=null)
            return data.toString();
        return null;  // call the toString() method in String class
    }

}
```

```
frog # frog # frog # frog #
dog # dog  # dog  # dog # dog #
goat # goat # goat # fish # goat  # tape
```

I used a linkedList that I had already written
This linkedList has code to insert the data at any point
but I only inserted it at the front.
I chose this because it has both the advantage of
rapid insertion and rapid removal, and good memory usage.

Even with the implementation of the sequencial page printing,
it only goes over the values once, which keeps the printing at
O(n) where n is the number of pages to print.
This would be the same for an array, but this has the additional benifit
of better memory usage.