SuffixTreeAppl.java

```
package SuffixTreePackage;
import java.util.*;
/**
 * Class with methods for carrying out applications of suffix trees
 * @author David Manlove
 */

public class SuffixTreeAppl {

	/** The suffix tree */
	private SuffixTree t;

	/**
	 * Default constructor.
	 */
	public SuffixTreeAppl () {
		t = null;
	}

	/**
	 * Constructor with parameter.
	 * 
	 * @param tree the suffix tree
	 */
	public SuffixTreeAppl (SuffixTree tree) {
		t = tree;
	}

	/**
	 * Search the suffix tree t representing string s for a target x.
	 * Stores -1 in Task1Info.pos if x is not a substring of s,
	 * otherwise stores p in Task1Info.pos such that x occurs in s
	 * starting at s[p] (p counts from 0)
	 * - assumes that characters of s and x occupy positions 0 onwards
	 * 
	 * @param x the target string to search for
	 * 
	 * @return a Task1Info object
	 */
	//modified insert
	public Task1Info searchSuffixTree(byte[] x) {
		Task1Info task1Info = new Task1Info();
		int pos, j, k;   //j:len to find
		SuffixTreeNode current, next;
		pos = 0;  // position in s
		current = t.getRoot();
		int len=x.length;

		while (true) {
			// search for child of current with left label x such that 
s[x]==s[pos]
			next = t.searchList(current.getChild(), x[pos]);
			if (next == null) {
```

```
                break;
            }
            else {
                // try to match
s[node.getLeftLabel()+1..node.getRightLabel()] with
                // segment of s starting at position pos+1
                j = next.getLeftLabel() + 1;
                k = pos + 1;

                while (j <= next.getRightLabel()) {
                    if (t.getString()[j]==x[k]) {
                        j++;
                        k++;
                    }
                    else
                        break;
                }
                //completed
                if (k >= len) {
                    task1Info.setMatchNode(next);//i did find this
useful

                    task1Info.setPos(j-len);
                    break;
                }
                if (j > next.getRightLabel()) {
                    // succeeded in matching whole segment, so go
further down tree

                    pos = k;
                    current = next;
                }
                else {
                    break;
                }
            }
        }
        return task1Info;
    }

    /**
     * Search suffix tree t representing string s for all occurrences of target x.
     * Stores in Task2Info.positions a linked list of all such occurrences.
     * Each occurrence is specified by a starting position index in s
     * (as in searchSuffixTree above).  The linked list is empty if there
     * are no occurrences of x in s.
     * - assumes that characters of s and x occupy positions 0 onwards
     *
     * @param x the target string to search for
     *
     * @return a Task2Info object
     */

    public Task2Info allOccurrences(byte[] x) {

        //use task1's matchnode to find first occurence
        Task1Info task1Info = searchSuffixTree(x);
        if(task1Info.getPos()==-1) {
            return new Task2Info();
        }
        Task2Info task2Info = new Task2Info();

        SuffixTreeNode current;
        current=task1Info.getMatchNode().getChild();
        //
        if(current!=null) {
            t2recursive(task2Info, current);
        }
        return task2Info;

    }

    public void t2recursive(Task2Info task2Info, SuffixTreeNode current) {
        SuffixTreeNode next=current.getChild(),sibling=current.getSibling();
        if(next!=null) {
            t2recursive(task2Info,next);
        }else {
            //if not a branch add position to the linked list
            task2Info.addEntry(current.getSuffix());
        }
        //continue for siblings
        if(sibling!=null) {
            t2recursive(task2Info,current.getSibling());
        }
    }



    /**
     * Traverses suffix tree t representing string s and stores ln, p1 and
     * p2 in Task3Info.len, Task3Info.pos1 and Task3Info.pos2 respectively,
     * so that s[p1..p1+ln-1] = s[p2..p2+ln-1], with ln maximal;
     * i.e., finds two embeddings of a longest repeated substring of s
     * - assumes that characters of s occupy positions 0 onwards
     * so that p1 and p2 count from 0
     *
     * @return a Task3Info object
     */
    //
    public Task3Info traverseForLrs () {
        Task3Info task3Info = new Task3Info();
        SuffixTreeNode current=t.getRoot();
        int len=0;
        //empty tree check
        if(current.getChild()!=null) {
            t3recursive(task3Info,current,len);
        }

        return task3Info;
    }
    public void t3recursive(Task3Info task3Info, SuffixTreeNode current,int len) {
        SuffixTreeNode next=current.getChild(),sibling=current.getSibling();




        if(next!=null) {
            //move down p keeping track of current length
            t3recursive(task3Info,next,len+current.getRightLabel()-
current.getLeftLabel()+1);
        }else {

            //System.out.println("gll"+current.getLeftLabel());
                //check if repeated if so its valid
                if(sibling !=null) {

                    //overwrite old longest
                    if(task3Info.getLen()<=len-1) {
                        //
                        task3Info.setLen(len-1);

                        task3Info.setPos1(current.getSuffix());
                        task3Info.setPos2(sibling.getSuffix());
                    }
                }
        }
        //check sibling
        if (sibling!=null) {
            t3recursive(task3Info,sibling,len);
        }
    }

    /**
     * Traverse generalised suffix tree t representing strings s1 (of length
     * s1Length), and s2, and store ln, p1 and p2 in Task4Info.len,
     * Task4Info.pos1 and Task4Info.pos2 respectively, so that
     * s1[p1..p1+ln-1] = s2[p2..p2+ln-1], with len maximal;
     * i.e., finds embeddings in s1 and s2 of a longest common substring
       * of s1 and s2
     * - assumes that characters of s1 and s2 occupy positions 0 onwards
     * so that p1 and p2 count from 0
     *
     * @param s1Length the length of s1
     *
     * @return a Task4Info object
     */
    public Task4Info traverseForLcs (int s1Length) {
        Task4Info task4Info = new Task4Info();
        SuffixTreeNode current=t.getRoot();


        //empty tree check.
        if(current.getChild()!=null) {
            t4recursive(task4Info,current,0,s1Length);
        }
        return task4Info;
    }


    public void t4recursive(Task4Info task4Info, SuffixTreeNode current,int
len,int s1Length) {
        SuffixTreeNode next=current.getChild(),sibling=current.getSibling();

        //branch v
        if(current.getSuffix()==-1) {
        //bi(v)
            if (current.getLeafNodeString1()) {
                task4Info.setString1Leaf(true);
            }if(current.getLeafNodeString2()) {
                task4Info.setString2Leaf(true);
            }
        }

        if(next!=null) {
            //move down
            t4recursive(task4Info,next,len+current.getRightLabel()-
current.getLeftLabel()+1,s1Length);
        }else {

            //System.out.println("gll"+current.getLeftLabel());
                //b1(v)and b2(v)
                if(task4Info.getString1Leaf() &&
task4Info.getString2Leaf()) {

                    //common so overwrite old longest
                    if(task4Info.getLen()<=len-1) {
                        task4Info.setLen(len-1);

                        //task4Info.setPos1(current.getSuffix());
                        //task4Info.setPos2(sibling.getSuffix());
                    }

                }
        }
        //check sibling
        if (sibling!=null) {

            t4recursive(task4Info,sibling,len,s1Length);
        }
    }
}
```

Main.java

```
      public static SuffixTreeAppl helper(String file) {
             FileInput one = new FileInput(file);
             byte[] s = one.readFile();
             SuffixTree tree = new SuffixTree(s);
             SuffixTreeAppl appl = new SuffixTreeAppl(tree);
             return appl;
      }



      public static void main(String args[]) {
              String errorMessage = "Required syntax:\n";
                errorMessage += "  java Main SearchOne <filename> <query string> for
Task 1\n";
                errorMessage += "  java Main SearchAll <filename> <query string> for
Task 2\n";
                errorMessage += "  java Main LRS <filename> for Task 3\n" ;
                errorMessage += "  java Main LCS <filename1> <filename2> for Task 4";

             if (args.length < 2)
                    System.out.println(errorMessage+"trigger");
             else {
                    // get the command from the first argument
                    String command = args[0];

                    switch (command) {
                          case "SearchOne": {
                                 if (args.length < 3) {
                                  System.out.println(errorMessage);
                           }
                                 SuffixTreeAppl sTree1 = helper(args[1]);
                                 Task1Info
task1=sTree1.searchSuffixTree(args[2].getBytes());
                                 if(task1.getPos()<0) {
                                        System.out.println("Search string
\""+args[2]+"\" not found in "+args[1] );
                                 }else {
                                        System.out.println("Search string
\""+args[2]+"\" occurs at position "+task1.getPos()+" of "+args[1] );
                                 }
                                 break;
                          }




                          case "SearchAll": {
                           if (args.length < 3) {
                                  System.out.println(errorMessage);
                           }
                             SuffixTreeAppl sTree2 = helper(args[1]);
                    Task2Info
task2=sTree2.allOccurrences(args[2].getBytes());
                    if(task2.getPositions().isEmpty()) {
                        System.out.println("Search string
\""+args[2]+"\" not found in "+args[1] );
                    }else {
                        System.out.println("Search string
\""+args[2]+"\" occurs in "+args[1]+" at positions:" );
                        int i=0;
                        while(!task2.getPositions().isEmpty()) {

    System.out.println(task2.getPositions().pop());
                            i++;
                        }
                        System.out.println("The total number of
occurrences is "+i);

                    }
                    break;
                }




                case "LRS": {
                    FileInput three = new FileInput(args[1]);
                    byte[] s3 = three.readFile();
                    SuffixTree tree3 = new SuffixTree(s3);
                    SuffixTreeAppl appl3 = new SuffixTreeAppl(tree3);
                    Task3Info task3=appl3.traverseForLrs();
                    int len3 =task3.getLen();
                    int pos13 = task3.getPos1();


                    if(len3<=0) {
                        System.out.println("No LRS in "+args[1]);
                    }
                    else {
                        System.out.print("An LRS in "+args[1]+" is
\"");

                        for(int i=0;i<len3;i++) {
                            System.out.print((char)s3[pos13+i]);
                        }
                        System.out.print("\"\n");

                        System.out.println("Its length is "+len3);
                        System.out.println("Starting position of one
occurence is "+pos13);

                        System.out.println("Starting position of
another occurence is "+task3.getPos2());
                    }
                    break;
                }





                case "LCS": {
                if (args.length < 3) {
                    System.out.println(errorMessage);
                    break;
                }
                FileInput four1 = new FileInput(args[1]);
                FileInput four2 = new FileInput(args[2]);
                byte[] s41 = four1.readFile();
                byte[] s42 = four2.readFile();
                SuffixTree tree4 = new SuffixTree(s41,s42);
                SuffixTreeAppl appl4 = new SuffixTreeAppl(tree4);
                Task4Info task4=appl4.traverseForLcs(s41.length);

                int len4 =task4.getLen();
                int pos14 = task4.getPos1();
                int pos24 = task4.getPos2();

                if(len4<=0) {
                    System.out.println("No LCS of "+args[1]+" and 
"+args[2]);
                }
                else {
                    System.out.print("An LCS of "+args[1]+" and 
"+args[2]+" is \"");

                    for(int i=0;i<len4;i++) {
                        System.out.print((char)s41[pos14+i]);
                    }
                    System.out.print("\"\n");

                    System.out.println("Its length is "+len4);
                    System.out.println("Starting position in "+args[1]+" 
is "+pos14);
                    System.out.println("Starting position in "+args[2]+" 
is "+pos24);
                }
                break;
            }
            default: System.out.println(errorMessage+"trigger2");
            }
        }
```

SuffixTree.java

```java
    /**
     * Builds a generalised suffix tree for two given strings.
     *
     * @param sInput1 the first string
     * @param sInput2 the second string
     * - assumes that '$' and '#' do not occur as a character anywhere in sInput1 or sInput2
     * - assumes that characters of sInput1 and sInput2 occupy positions 0 onwards
     */
    public SuffixTree (byte[] sInput1, byte[] sInput2) {
        root = new SuffixTreeNode(null, null, 0, 0, -1);  // create root node of suffix tree;
        int l1 = sInput1.length;
        int l2 = sInput2.length;
        stringLen =l1+l2 +1;
        s = new byte[stringLen + 1];
        System.arraycopy(sInput1, 0, s, 0, l1);
        s[l1] = (byte) '#';
        System.arraycopy(sInput1, 0, s, l1, l2);
        s[stringLen] = (byte) '$';
        buildSuffixTree();
    }
```