Assignment 1: Applications of the Trie Data Structure

By Joel Pillar-Rogers Completed: 3 May 2020

Task 1: Predictive Text - Description (4%)

Child node structure

The first relevant implementation decision for this assignment was made in Lab2 - the choice of data structure for the child nodes in each TrieNode. I initially used *TrieNode[] children*. I liked this when it was limited to 26 elements (for the lowercase characters) because it was fast to access. For instance, *getChild('z')* was found at *children[25]* in constant O(1) time (and the same for *addChild('z')*). These operations are performed a lot during the creation of the Trie, so I want them as fast as possible.

```
private final int ASCII = 97;   // this is the adjustment for ascii characters so that 'a' == 0
TrieNode[] children = new TrieNode[26];

/** Lookup a child node of the current node that is associated with a ...*/
public TrieNode getChild(char label) {
    return children[label - ASCII];
}
```

To read in a dictionary of words that also contained uppercase letters and symbols, I had to change the size of my array and ASCII adjustment.

```
private final int ASCII = 32;   // this is the adjustment for ascii characters so that 'a' == 0
TrieNode[] children = new TrieNode[224]; // exclude the first 32 control characters of the extended ASCII codes

/** Lookup a child node of the current node that is associated with a ...*/
public TrieNode getChild(char label) {
    return children[label - ASCII];
}
```

I'm aware this has a large memory requirement for each TrieNode. It's likely this will need to be changed to Trent's suggested TreeMap implementation to handle the larger files later in this assignment.

Method: readInDictionary()

For this method, I copied most of the code from Lab1 (Dictionary). I decided to keep the try/catch structure since it continues to be useful to catch misspelled or missing data items in a graceful way. I also used the *String[] lineParts* and *.split()* method from the DictionaryData class, as it is a neat way to break up each input line. I had to add a *"+"* to the regex to deal with multiple spaces between input words. I discarded the ranking number at *lineParts[0]* as I don't think it's needed. I inserted an anonymous TrieData object into the *insert()* method because it will get picked up and referenced later.

```
public static Trie readInDictionary(String fileName) {
    Trie t = new Trie();
    Scanner fileScanner;
    try {
        fileScanner = new Scanner(new FileInputStream(fileName));
        while (fileScanner.hasNextLine()) {
            String[] lineParts = fileScanner.nextLine().split( regex: " +");
            t.insert(lineParts[1].trim(),new TrieData(Integer.parseInt(lineParts[2].trim())));
        }
    } catch (FileNotFoundException ex) {
        System.out.println("could not find the file " +fileName+ " in the data directory!");
        return null;
    }
    return t;
}
```

Testing: readInDictionary()

My code to this point passed these tests in IntelliJ and on FLO. I tested with both the trimmed and untrimmed version of word.freq.expanded, and also with the grow.txt.

```
4 data/word-freq.expanded.txt anomaly aba hoodie
Reading in trie...done

testing getNode
anomaly: TrieNode; isTerminal=true, data=33, #children=0
aba: TrieNode; isTerminal=false, data=null, #children=3
hoodie: null

testing get
anomaly: TrieNode; isTerminal=true, data=33, #children=0
aba: null
hoodie: null

Process finished with exit code 0
```

```
4 data/word-freq.grow.txt gre
Reading in trie...done

testing getNode
gre: TrieNode; isTerminal=false, data=null, #children=2

testing get
gre: null

Process finished with exit code 0
```

	Input	Expected	Got	
✓	4 word-freq.expanded.trim.txt anomaly aba hoodie clever	Reading in trie...done testing getNode anomaly: TrieNode; isTerminal=true, dat a=33, #children=0 aba: TrieNode; isTerminal=false, data=n ull, #children=3 hoodie: null clever: TrieNode; isTerminal=true, data =541, #children=2 testing get anomaly: TrieNode; isTerminal=true, dat a=33, #children=0 aba: null hoodie: null clever: TrieNode; isTerminal=true, data =541, #children=2	Reading in trie...done testing getNode anomaly: TrieNode; isTerminal=true, dat a=33, #children=0 aba: TrieNode; isTerminal=false, data=n ull, #children=3 hoodie: null clever: TrieNode; isTerminal=true, data =541, #children=2 testing get anomaly: TrieNode; isTerminal=true, dat a=33, #children=0 aba: null hoodie: null clever: TrieNode; isTerminal=true, data =541, #children=2	✓

Passed all tests! ✓

Method: getMostFrequentWordWithPrefix()

Initially I used the getAlphabeticalListWithPrefix method from Lab2 and iterated through the returned list to find the word with the highest frequency. This worked fine and passed all the tests. But then I thought there was a more efficient way to do this without the overhead of this method. I just needed to return one word after all. I reimplemented a similar recursive method but which just returned one string. The method relies on setting and returning the value of previously declared instance variables.

```
/** NOTE: TO BE IMPLEMENTED IN ASSIGNMENT 1 Finds the most frequently ...*/
public String getMostFrequentWordWithPrefix(String prefix) {
    prefixNode = getNode(prefix);                   // initialise/reassign these instance variables each time
    maxFreq = 0;                                    // ditto
    mostFreq = null;                                // ditto
    if(prefixNode == null) return null;             // check that the prefix exists
    getFreqWord(prefixNode,prefix);                 // call the recursive method
    return mostFreq;
}

/** Helper method, to recursively check all the words in the trie for the most frequent, from the prefix node down ...*/
private void getFreqWord(TrieNode testNode, String word){
    if(testNode.isTerminal()){
        int freq = testNode.getData().getFrequency();
        if(freq > maxFreq){
            maxFreq = freq;
            mostFreq = word;
        }
    }
    int i = 0;                                          // this counter identifies which character is associated with each loop
    for(TrieNode tn : testNode.children){               // cycle through the TrieNode child array in each TrieNode
        if(tn != null) {                                // don't go down paths with no children
            getFreqWord(tn,  word: word + (char)(i + 32));   // recursive call down each path; concatenation will make a new string
        }
        i++;                                            // increment for the next letter/child
    }
}
```

Testing: getMostFrequentWordWithPrefix()

My code passed these tests in IntelliJ and FLO. I tested with both the trimmed and untrimmed version of word.freq. You can see it is returning the most frequent word from the provided list.

```
5 data/word-freq.expanded.trim.txt aba the bbb
Reading in trie...done
PREFIX = aba
Most Frequent Word is abandoned
PREFIX = the
Most Frequent Word is the
PREFIX = bbb
Most Frequent Word is null

Process finished with exit code 0
```

```
5 data/word-freq.expanded.txt yo tha wh kn ri
Reading in trie...done
PREFIX = yo
Most Frequent Word is you
PREFIX = tha
Most Frequent Word is that
PREFIX = wh
Most Frequent Word is what
PREFIX = kn
Most Frequent Word is know
PREFIX = ri
Most Frequent Word is right
```

word-freq.expanded.txt

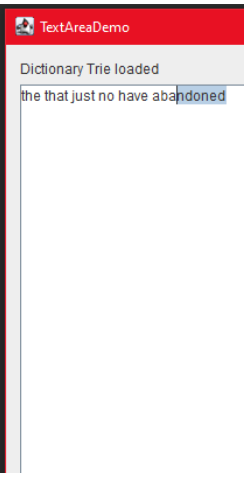
```
1 you 1222421
2 I 1052546
3 to 823661
4 the 770161
5 a 563578
6 and 480214
7 that 413389
8 it 388320
9 of 332038
10 me 312326
11 what 285826
12 is 282222
13 in 266544
14 this 249860
15 know 241548
```

	Input	Expected	Got	
✓	5 data/word-freq.expanded.trim.txt aba the bbb clev	Reading in trie...done PREFIX = aba Most Frequent Word is abandoned PREFIX = the Most Frequent Word is the PREFIX = bbb Most Frequent Word is null PREFIX = clev Most Frequent Word is clever	Reading in trie...done PREFIX = aba Most Frequent Word is abandoned PREFIX = the Most Frequent Word is the PREFIX = bbb Most Frequent Word is null PREFIX = clev Most Frequent Word is clever	✓

Passed all tests! ✓

The TextAreaDemo consistently and instantly produced the correct (most frequent) results.

```
1 you 1222421
2 I 1052546
3 to 823661
4 the 770161
5 a 563578
6 and 480214
7 that 413389
8 it 388320
9 of 332038
10 me 312326
11 what 285826
12 is 282222
13 in 266544
14 this 249860
15 know 241548
16 I'm 230304
17 for 216535
18 no 212463
19 have 210523
```



Task 2: Exact String Matching - Description (6%)

Method: insert()

Before starting this method, I first implemented the *addChild()*, *getChild()* and *addData()* methods in the SuffixTrieNode. For now, I continued to use the large array method for storing child nodes. This will likely be a problem later. The implementation is similar to the TrieNode implementation; however, instead of storing the frequency only at a terminal node, here we store the starting coordinates for the substring (in a SuffixIndex object) in every node we pass. Therefore, a SuffixTrieNode may have multiple SuffixIndex objects in its SuffixTrieData object.

```
private SuffixTrieData data = new SuffixTrieData();
private int numChildren = 0;
private final int ASCII = 32;                          // this is the adjustment for ascii characters so that 'a' == 0
SuffixTrieNode[] children = new SuffixTrieNode[224];    // exclude the first 32 control characters of the extended ASCII codes

/** Lookup a child node of the current node that is associated with the character label. ...*/
public SuffixTrieNode getChild(char label) {
    return children[label - ASCII];
}

/** Add a child node to the current node, at the position associated with the provided label. ...*/
public void addChild(char label, SuffixTrieNode node) {
    children[label - ASCII] = node;
    numChildren++;
}

/** Add a new SuffixIndex to the SuffixTrieData associated with this node. ...*/
public void addData(int sentencePos, int characterPos) {
    SuffixIndex si = new SuffixIndex(sentencePos, characterPos);
    data.addStartIndex(si);
}
```

To implement the *insert()* method in SuffixTrie, I started with the *insert()* method from Trie. There were a few differences. I used a *for* loop to iterate through each substring of the sentence passed in, then used a nested *for* loop to iterate over each character in each substring. I added nodes in a similar way to the Trie, but added data to each node, rather than just at the terminal. It was also necessary to reset the itNode node to root each time so that each substring would be added from the root of the SuffixTrie.

```
public SuffixTrieNode insert(String str, int startPosition) {
    char[] sentence = str.toCharArray();
    SuffixTrieNode itNode = root;
    for(int i = 0; i < sentence.length; i++) {
        itNode = root;
        for (int j = i; j < sentence.length; j++) {
            if (itNode.getChild(sentence[j]) == null) {
                itNode.addChild(sentence[j], new SuffixTrieNode());
            }
            itNode = itNode.getChild(sentence[j]);
            itNode.addData(startPosition, i);
        }
    }
    return itNode;
}
```

I tested these methods using some code in SuffixTrieDrive and it performed as expected.

```
    public static void main(String[] args) {

        SuffixTrie st = new SuffixTrie();
        SuffixTrieNode sn = st.insert( str: "She sells sea shells by the sea shore",  startPosition: 5);
        System.out.println(sn.toString());      // prints out the location of the last substring entered, "e".
```

```
Assignment01
SuffixTrieDriver
"C:\Program Files\Java\jdk-13.0.2\bin\java.exe" -javaagent:C:\Users\Joel\AppData\Local\JetBrains\Toolbox\apps\ID
[5.2, 5.5, 5.11, 5.16, 5.26, 5.29, 5.36]

Process finished with exit code 0
```

Method: get()

The code for this method was identical to the code from the Trie. The *for* loop iterates down the nodes of the SuffixTrie following the characters in the sentence. If it gets to the end without encountering a *null*, the (sub)string exists and the final node is returned.

```
public SuffixTrieNode get(String str) {
    char[] sentence = str.toCharArray();
    SuffixTrieNode itNode = root;
    for (char ch : sentence) {
        if (itNode.getChild(ch) == null) {
            return null;
        }
        itNode = itNode.getChild(ch);
    }
    return itNode;
}
```

Method: readInFromFile()

I copied this method from the Trie and made a few changes. Here, each line is scanned in one at a time, split into sentences and inserted into the SuffixTrie. The regex was adjusted to split lines into sentences using [!?.]+. I no longer trim sentences, and I added a count for the method that increments each time a sentence is added. This is the sentence index passed to each SuffixTrieNode.

```
public static SuffixTrie readInFromFile(String fileName) {
    SuffixTrie st = new SuffixTrie();
    Scanner fileScanner;
    int count = 0;
    try {
        fileScanner = new Scanner(new FileInputStream(fileName));
        while (fileScanner.hasNextLine()) {
            String[] lineParts = fileScanner.nextLine().split( regex: "[!?.]+");
            for (String s : lineParts){
                st.insert(s, count);
                count++;
            }
        }
    } catch (FileNotFoundException ex) {
        System.out.println("could not find the file " +fileName+ " in the data directory!");
        return null;
    }
    return st;
}
```

Testing: readInFromFile()

I tested this method in IntelliJ. It initially worked for Frank01.txt, but quickly failed on Frank02.txt. This seems to be related to the child array in each node, with a character 8212 (8180 + 32) not falling within my ASCII range.

```
data/Frank01.txt
and
, and
, the
rejoice
ll re

[and]: [1.27, 1.87]
[, and]: [1.25]
[, the]: null
[rejoice]: [0.9]
[ll re]: [0.6]
```

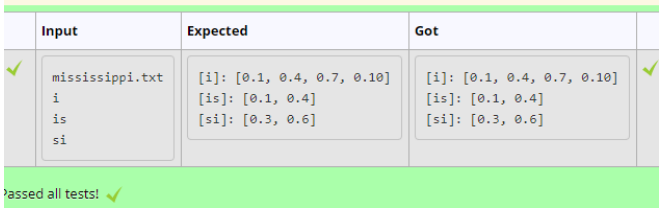
```
data/Frank02.txt
and
, the
onster
monst

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 8180 out of bounds for length 224
    at cp3.ass01.suffixtrie.SuffixTrieNode.getChild(SuffixTrieNode.java:17)
    at cp3.ass01.suffixtrie.SuffixTrie.insert(SuffixTrie.java:30)
    at cp3.ass01.suffixtrie.SuffixTrie.readInFromFile(SuffixTrie.java:77)
    at cp3.ass01.suffixtrie.SuffixTrieDriver.main(SuffixTrieDriver.java:50)
```

This character is a Unicode character, an em-dash (“—"), and when removed from Frank02.txt is parses without error. I can’t make my arrays big enough to store all the Unicode characters, so this is difficult to get around. For now, I just made a code exception for the em-dash. I added code to check if a character was a Unicode em-dash (8212) and if so, convert it to an ASCII em-dash (151). When I reran the Frank02.txt it passed with the correct results.

```
data/Frank02.txt
and
, the
onster
monst

[and]: [1.27, 1.87, 3.34, 3.153, 4.15, 6.70, 7.66, 7.148, 8.88, 9.96, 9.120, 9.173, 9.199, 10.17, 12.67, 12.91, 13.97, 13.114, 14.27, 14.90]
[, the]: [8.16]
[onster]: null
[monst]: null
```

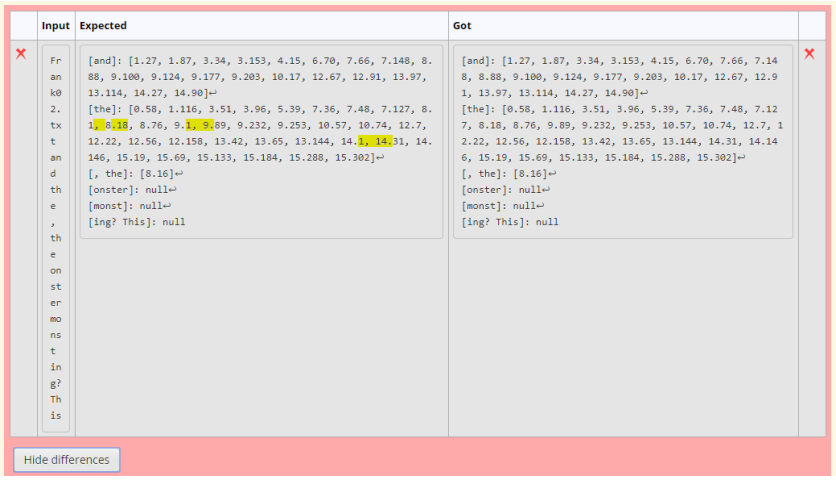
I also passed these tests on FLO.

Self-review test 3:



Self-review test 4:

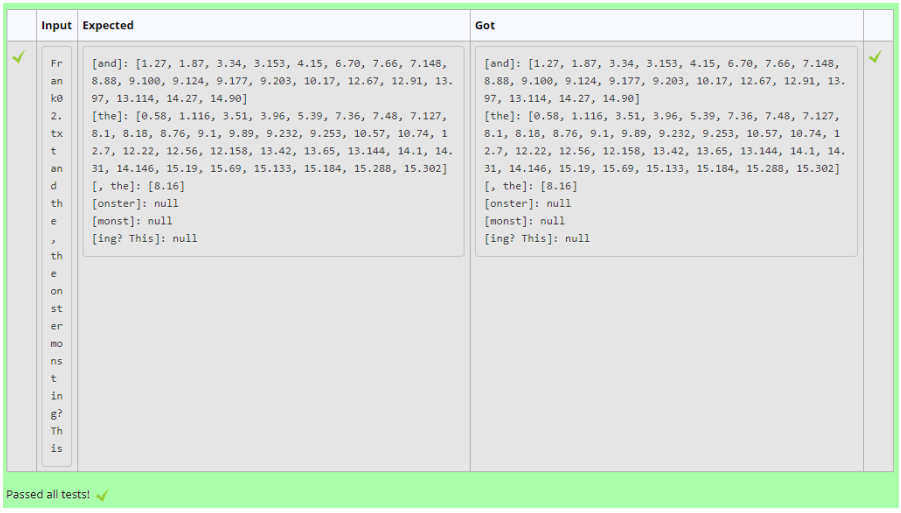
I failed the last test of the normal review. Self-review test 5:



This failed because I was capturing lowercase and uppercase letters separately. The “the” search didn’t find the “The” substrings at the start of sentences 8, 9 and 14. To minimise space, it was recommended to only use lowercase letters. I hadn’t done that, so I now made changes to that effect, reducing the size of my child array by 26 characters to 199, and adding extra code to efficiently use the space in my array for adding and getting strings. I also added code to handle the Unicode “replacement character” which sometimes popped up.

```
SuffixTrieNode[] children = new SuffixTrieNode[199];    // exclude the first 32 control characters of the extended ASCII codes

/** Lookup a child node of the current node that is associated with the character label. ...*/
public SuffixTrieNode getChild(char label) {
    if(label == 8212) label = 151;              // convert unicode em-dash to ascii em-dash
    if(label == 65533) label = 256;             // handle the unicode replacement character
    if(label >64 && label < 91) label+=32;      // convert uppercase to lowercase
    if(label > 90) label -= 26;                 // use up the space left by uppercase
    return children[label - ASCII];
}

/** Add a child node to the current node, at the position associated with the provided label. ...*/
public void addChild(char label, SuffixTrieNode node) {
    if(label == 8212) label = 151;              // convert unicode em-dash to ascii em-dash
    if(label == 65533) label = 256;             // handle the unicode replacement character
    if(label >64 && label < 91) label+=32;      // convert uppercase to lowercase
    if(label > 90) label -= 26;                 // use up the space left by uppercase
    children[label - ASCII] = node;
    numChildren++;
}
```

This code was getting unwieldy, and a switch to a TreeMap seemed imminent. The problem here is that my alphabet (R) is poorly defined and I kept having to hack extra bits onto it. I could've used the Unicode 2-byte alphabet, but an array of 65,536 elements would use up my heap memory very fast. Regardless, my code now passed the final test, and all the previous tests as well.

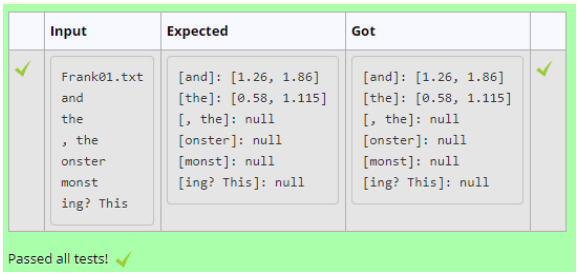


I also attempted the stricter tests on FLO with some minor changes, e.g. adding "data + separator" to the filename, ignoring empty lines and trimming each sentence).

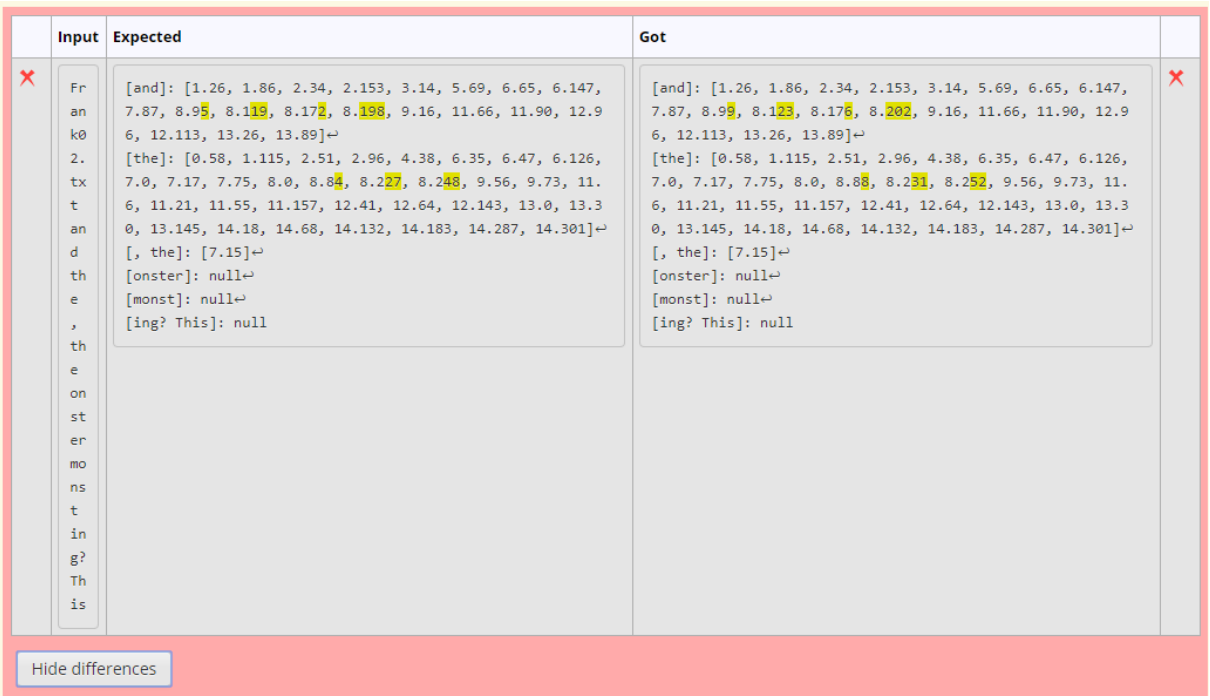
```
fileScanner = new Scanner(new FileInputStream( name: "data" + File.separator + fileName));
while (fileScanner.hasNextLine()) {
    String[] lineParts = fileScanner.nextLine().split( regex: "[!?.]+");
    if(lineParts[0].isEmpty() && lineParts.length == 1) continue;
    for (String s : lineParts){
        st.insert(s.trim(), count);
        count++;
    }
}
```

Stricter Test 1 passed, but Stricter Test 2 failed. As others have mentioned on the CP3 forums, this is likely to do with the encoding of the page and the em-dash being read as multiple characters and throwing off the character index by four places in sentence 8. I won't bother to handle this encoding issue in this assignment, and I am happy with my results.

Stricter Test 1:



Stricter Test 2:



Larger file sizes

To this point, I believe my code met the basic requirements of the assignment. The code revision number was **1875**. However, I now needed to test the larger file sizes to see how my *children[199]* setup in each node will perform. I will do that in the next section, profiling.

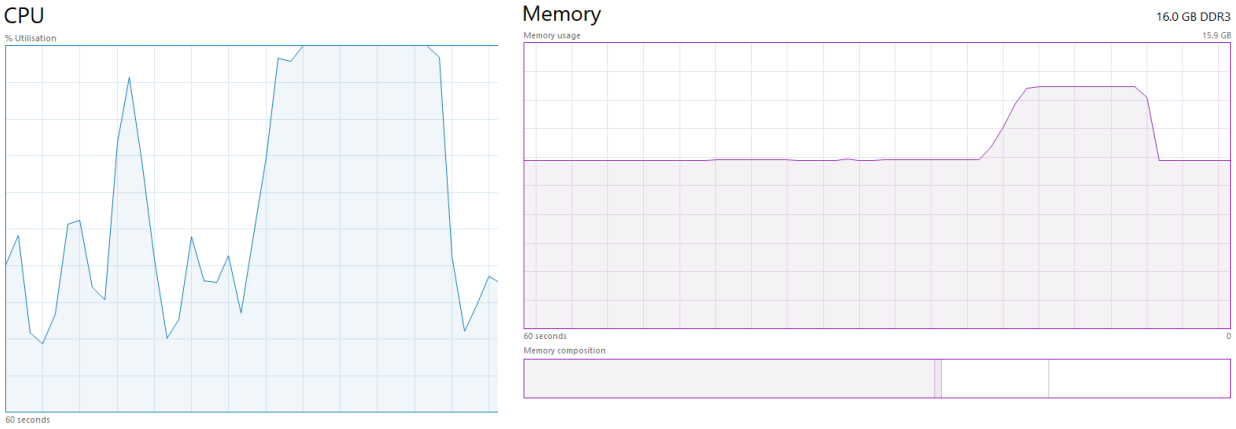
Task 2: Exact String Matching - Profiling (12%)

Initial Testing

I ran FrankChap02.txt and the program crashed.

```
FrankChap02.txt
and
the
, the
onster
monst
ing? This


Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at cp3.ass01.suffixtrie.SuffixTrieNode.<init>(SuffixTrieNode.java:8)
    at cp3.ass01.suffixtrie.SuffixTrie.insert(SuffixTrie.java:28)
    at cp3.ass01.suffixtrie.SuffixTrie.readInFromFile(SuffixTrie.java:75)
    at cp3.ass01.suffixtrie.SuffixTrieDriver.main(SuffixTrieDriver.java:46)

Process finished with exit code 1
```

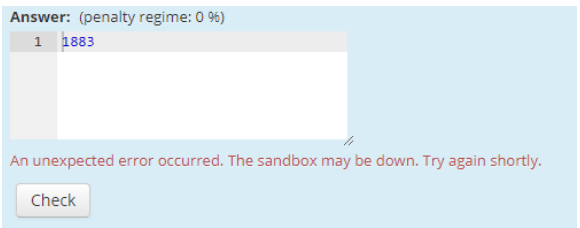
My simple task manager profiling showed the CPU maxing out for around 10 seconds and my memory usage also spiking.



Increasing the allocated heap size using *-Xmx16000m* allowed it to complete on my machine.

```
FrankChap02.txt
and
the
, the
onster
monst
ing? This

[and]: [2.13, 7.26, 7.86, 8.34, 8.153, 9.14, 11.69, 12.65, 12.147, 13.87, 14.95, 14.119, 14.172, 14.198, 15.16, 17.66, 17.90, 18.96, 18.113, 19.26, 19.89, 21
[the]: [6.58, 7.115, 8.51, 8.96, 10.38, 12.35, 12.47, 12.126, 13.0, 13.17, 13.75, 14.0, 14.84, 14.227, 14.248, 15.56, 15.73, 17.6, 17.21, 17.55, 17.157, 18.4
[, the]: [13.15, 34.246, 41.52, 64.214, 75.4, 85.118, 101.70, 120.116, 128.65, 159.16, 177.13, 221.19, 221.71, 227.14, 262.21, 288.1, 290.40, 292.70, 296.88,
[onster]: null
[monst]: [369.59]
[ing? This]: null

Process finished with exit code 0
```

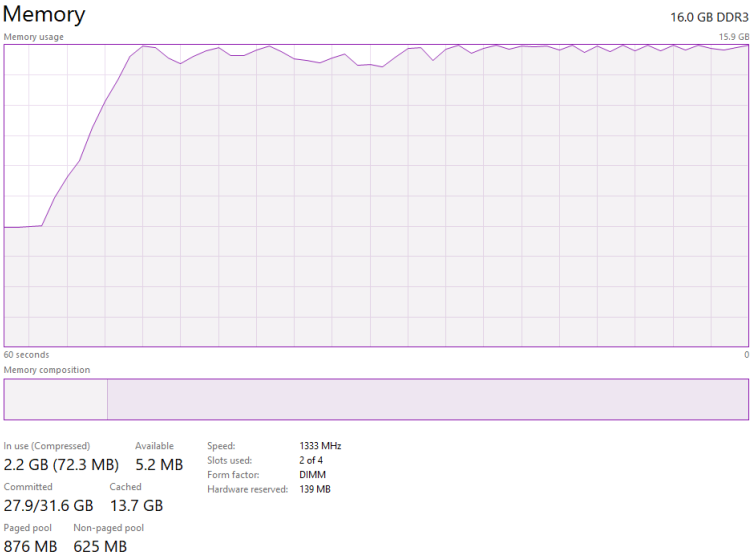
But not on FLO.



I was also able to complete FrankChap04.txt on my machine.

```
FrankChap04.txt
and
the
, the
onster
monst
ing? This

[and]: [2.13, 7.26, 7.86, 8.34, 8.153, 9.14, 11.69, 12.65, 12.147, 13.87, 14.95, 14.119, 14.172, 14.198, 15.16, 17.66, 17.90, 18.96,
[the]: [6.58, 7.115, 8.51, 8.96, 10.38, 12.35, 12.47, 12.126, 13.0, 13.17, 13.75, 14.0, 14.84, 14.227, 14.248, 15.56, 15.73, 17.6, 17
[, the]: [13.15, 34.246, 41.52, 64.214, 75.4, 85.118, 101.70, 120.116, 128.65, 159.16, 177.13, 221.19, 221.71, 227.14, 262.21, 288.1,
[onster]: null
[monst]: [369.59]
[ing? This]: null

Process finished with exit code 0
```

I wasn't able to complete FrankMed.txt, with my computer memory maxing out and paging memory for about 10 minutes before the program failed.




```
FrankMed.txt
and
the
 the
onster
monst
ing? This

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at cp3.ass01.suffixtrie.SuffixTrieNode.<init>(SuffixTrieNode.java:8)
    at cp3.ass01.suffixtrie.SuffixTrie.insert(SuffixTrie.java:28)
    at cp3.ass01.suffixtrie.SuffixTrie.readInFromFile(SuffixTrie.java:75)
    at cp3.ass01.suffixtrie.SuffixTrieDriver.main(SuffixTrieDriver.java:46)

Process finished with exit code 1
```

Re-implementing child nodes

At this point, it was time to try a different implementation to store child nodes in each node. I made a copy of my current classes and edited the child node array to be a TreeMap, similar to Trent's suggested implementation. This also greatly simplified the code.

```java
private Map<Character,SuffixTrieNode> children = new TreeMap<>();

/** Lookup a child node of the current node that is associated with the character label. ...*/
public SuffixTrieNode getChild(char label) {
    return children.get(label);
}

/** Add a child node to the current node, at the position associated with the provided label. ...*/
public void addChild(char label, SuffixTrieNode node) {
    children.put(label, node);
}
```

With these changes, I was able to successfully run both FrankMed.txt and Frankenstien.txt. My VM configuration was set to *-Xmx10000m* for these tests.

```
Frankenstien.txt
and
the
, the
onster
monst
ing? This

[and]: [2.13, 7.26, 7.86, 8.34, 8.153, 9.14, 11.69, 12.65, 12.147, 13.87, 14.95, 14.119, 14.172, 14.198, 15.16, 17.6
[the]: [6.58, 7.115, 8.51, 8.96, 10.38, 12.35, 12.47, 12.126, 13.17, 13.75, 14.84, 14.227, 14.248, 15.56, 15.73, 17.
[, the]: [13.15, 34.246, 41.52, 64.214, 75.4, 85.118, 101.70, 120.116, 128.65, 159.16, 177.13, 221.19, 221.71, 227.1
[onster]: [653.254, 698.26, 716.21, 726.17, 1189.131, 1279.80, 1293.278, 1417.10, 1656.158, 1748.16, 1890.20, 1965.2
[monst]: [369.59, 653.253, 698.25, 716.20, 726.16, 1189.130, 1279.79, 1293.277, 1417.9, 1656.157, 1748.15, 1890.19,
[ing? This]: null

Process finished with exit code 0
```

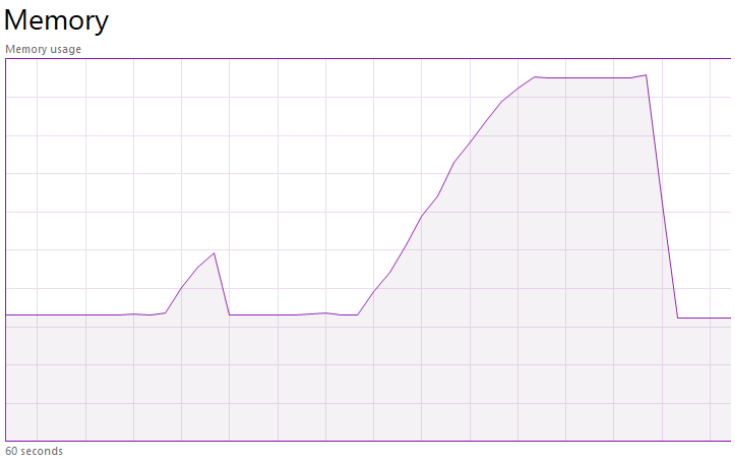
In-Program Profiling: Timing

I wanted to profile the first four .txt files using both implementations of my child array. I copied the Profiler code from Trent's video to capture time and memory use.

The SuffixProfiler class is identical to Trent's class, and here is my code in main().

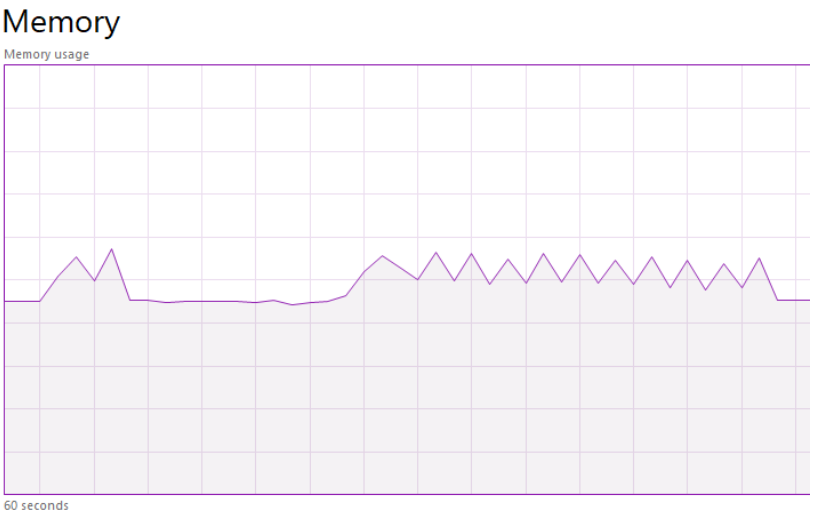
```
public static void runTime(String filename, int n) {
    System.out.println("Running the profiler");
    System.out.println("Press enter to start profiling...");
    Scanner in = new Scanner(System.in);
    in.nextLine();

    // instantiate the profiler object
    SuffixProfiler profiler = new SuffixProfiler();
    System.out.println("Running the readInFromFile() method " + n + " times");

    // run the task once to get the JVM warmed up
    SuffixTrie st = SuffixTrie.readInFromFile(filename);

    // run the task n times
    profiler.avgReset(n);
    for(int i = 0; i < n; i++){
        profiler.avgTic();
        st = SuffixTrie.readInFromFile(filename);
        profiler.avgToc();
    }

    System.out.println("Average time (ns): " + profiler.avgCalc());
    System.out.println("Average time (ms): " + SuffixProfiler.nsToms(profiler.avgCalc()));
}
```

Here is a plot of running readInFromFile() 1 time versus running it 10 times. Clearly, each suffix trie is being saved in memory. This might affect my results, so I ran a garbage collector between each loop.



With this change, the memory usage had a jagged profile, as each suffix trie was removed from memory each loop. Even though the garbage collector was not between the tic and toc, the average times below all increased a little due to the SuffixTrie elements needing to be reinitialised every time.

```
for (int i = 0; i < n; i++) {
    profiler.avgTic();
    st = SuffixTrie.readInFromFile(filename);
    profiler.avgToc();
    st = null;
    profiler.takeOutGarbage();
}
```



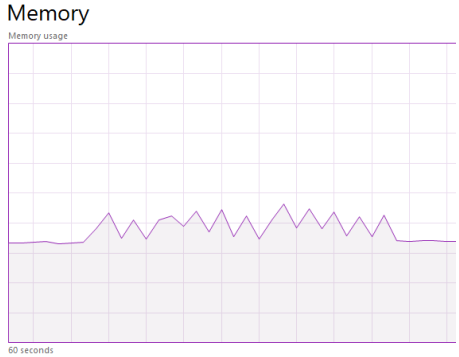
Now I ran this method for both implementations, for all .txt files. The CPU was peaking near 100% for all of these runs. These times look more like O(n), than O(n^2), which is what the algorithm is supposed to be. Perhaps this is because we're breaking the whole .txt file into sentences first.

Results Table: readInFileName() - Time Taken

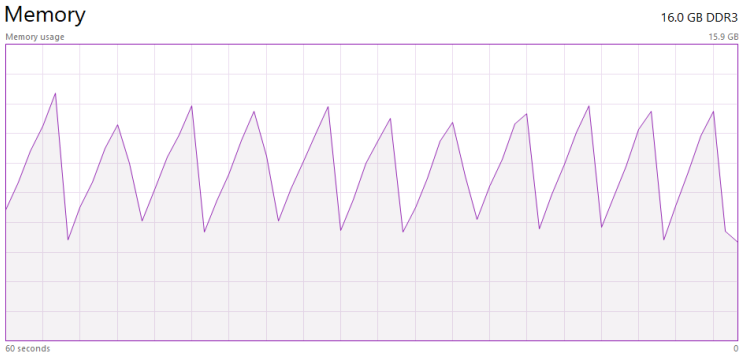
filename	**SuffixTrie with TreeMap (average of 10 runs, ms)**	**SuffixTrie with Array (average of 10 runs, ms)**	**Word Count**
Frank01.txt	10	26	49
Frank02.txt	91	187	435
FrankChap02.txt	1329	3002	9,544
FrankChap04.txt	1996	4593	14,767
FrankMed.txt	7773 (8.3 secs)	crashes (out of memory)	62,208
Frankenstien.txt	10176 (10.2 secs)	crashes (out of memory)	75,085

It was interesting to see the memory profiles of the two SuffixTries for FrankChap04.txt, as the garbage collector did its job over the 10 loops. The array is using a LOT more memory.

SuffixTrie with TreeMap (*FrankChap04.txt*)



SuffixTrie with Array (*FrankChap04.txt*)



In-Program Profiling: Memory

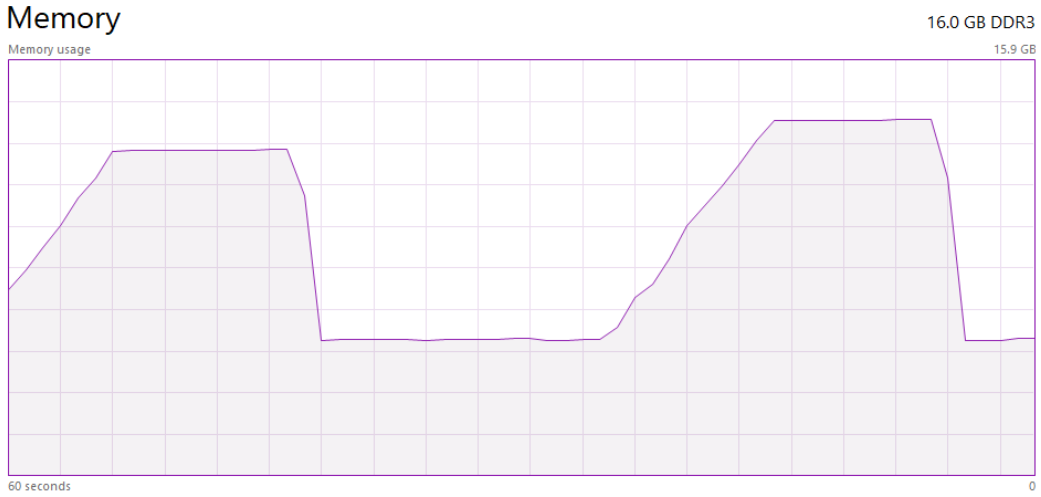
I now ran similar profiling for memory.

Results Table: readInFileName() - Memory Usage

filename	**SuffixTrie with TreeMap (megabytes)**	**SuffixTrie with Array (megabytes)**	**Word Count**
Frank01.txt	4	19	49
Frank02.txt	59 (x14.8)	244	435 (x8.9)
FrankChap02.txt	1,068 (x18.1)	4,380	9,544 (x21.9)
FrankChap04.txt	1,631 (x1.5)	6,698	14,767 (x1.5)
FrankMed.txt	6,130 (x3.8)	crashes (out of memory)	62,208 (x4.2)
Frankenstien.txt	7,316 (x1.19)	crashes (out of memory)	75,085 (x1.21)

It was interesting that time taken and memory used increased a bit more slowly than the amount of text at higher word counts (increases in brackets). I assume this is because a greater share of words is repeated as the text size increases, so as the program progresses each word is more likely to be in the suffix trie already (just a new SuffixIndex). Again, the increases suggest these are O(n) operations or potentially O(mn), with the string length ‘m’ averaging over the text.

This chart shows the memory usage for the TreeMap for *FrankMed.txt* (left) and *Frankenstien.txt* (right).



In-Program Profiling: get() Timing

Finally, I wanted to test how quickly words could be retrieved from my suffix tries. This may be the only way in which my array of children would be faster than the TreeMap. I wrote a test for this which is very similar to the test for reading in the SuffixTrie. This test finds 10 words, n times.

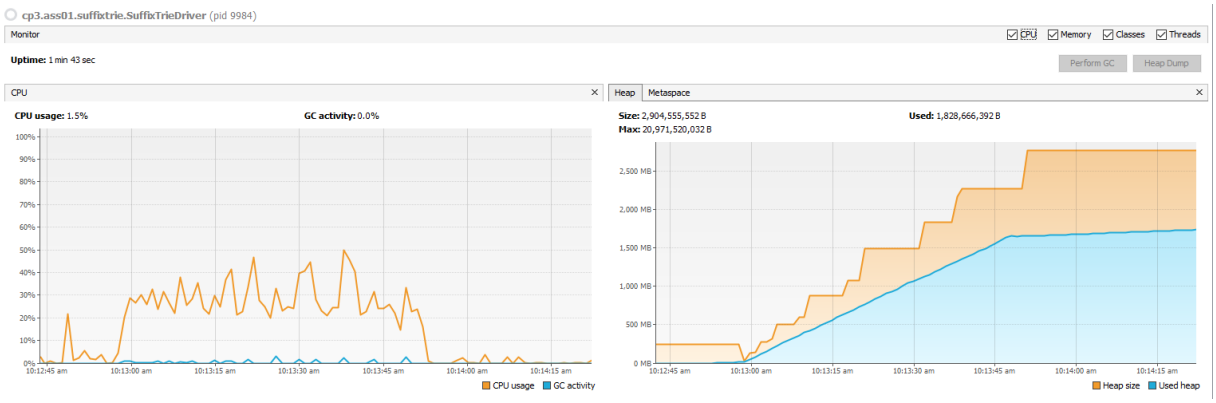
filename	**SuffixTrie with TreeMap (average 10 runs, ns)**	**SuffixTrie with Array (average of 10 runs, ns)**
Frank01.txt	6050	5760
Frank02.txt	6340	6060
FrankChap02.txt	6570	6200
FrankChap04.txt	6410	6320
FrankMed.txt	6320	crashes (out of memory)
Frankenstien.txt	6860	crashes (out of memory)

My results show the children array might be the tiniest bit quicker than the children TreeMap, but these are nano seconds, so it wouldn't be close to noticeable by humans. Also, neither of these times really increase with total word count (n). They should be O(mlogR) operations, where "R" is the size of the alphabet (at worst), which will never be very big and also doesn't scale with "n".

VM Profiling

I had a quick try with the VisualVM profiler. These profiles were captured running the *readInFromFile()* method once only on FrankChap04.txt (almost 15,000 words). I used the TreeMap children version.

With the profiler running, the program took a LOT longer to execute. Around 50 seconds versus 2 seconds usually. This first graph shows the CPU load from my program only (my task manager CPU was closer to 90%). The right side shows the allocated heap size increasing in blocks as the used heap increases. The total memory used (1828 MB) is in line with my in-program profiler (1621 MB).



According to the CPU sampler snapshot, it took around 60 seconds to run the program, including 10 seconds waiting for me to press the enter key. The remaining 50 seconds was spent in the *insert()* method. This method called three other methods, *addChild()*, *getChild()* and *addData()*. Of the three, *addData()* had to create a new SuffixIndex object and add it to the TreeMap and *getChild()* had to traverse the TreeMap and return a SuffixTrieNode object, which may have made them take slightly longer.

Name	Total Time
main	59,789 ms (100%)
cp3.ass01.suffixtrie.SuffixTrieDriver.**main** ()	59,789 ms (100%)
cp3.ass01.suffixtrie.SuffixTrieDriver.**runMem** ()	59,789 ms (100%)
cp3.ass01.suffixtrie.SuffixTrie.**readInFromFile** ()	49,699 ms (83.1%)
cp3.ass01.suffixtrie.SuffixTrie.**insert** ()	49,605 ms (83%)
cp3.ass01.suffixtrie.SuffixTrieNode.**addData** ()	19,450 ms (32.5%)
cp3.ass01.suffixtrie.SuffixTrieData.**addStartIndex** ()	9,171 ms (15.3%)
org.graalvm.visualvm.lib.jfluid.server.ProfilerRuntimeCPUFullInstr.**rootMethodEntry** ()	5,571 ms (9.3%)
org.graalvm.visualvm.lib.jfluid.server.ProfilerRuntimeCPUFullInstr.**methodExit** ()	4,602 ms (7.7%)
Self time	104 ms (0.2%)
cp3.ass01.suffixtrie.SuffixTrieNode.**getChild** ()	19,350 ms (32.4%)
org.graalvm.visualvm.lib.jfluid.server.ProfilerRuntimeCPUFullInstr.**rootMethodEntry** ()	9,783 ms (16.4%)
org.graalvm.visualvm.lib.jfluid.server.ProfilerRuntimeCPUFullInstr.**methodExit** ()	9,255 ms (15.5%)
java.util.TreeMap.**get** ()	310 ms (0.5%)
Self time	0.0 ms (0%)
cp3.ass01.suffixtrie.SuffixTrieNode.**addChild** ()	10,698 ms (17.9%)
org.graalvm.visualvm.lib.jfluid.server.ProfilerRuntimeCPUFullInstr.**rootMethodEntry** ()	5,928 ms (9.9%)
org.graalvm.visualvm.lib.jfluid.server.ProfilerRuntimeCPUFullInstr.**methodExit** ()	4,293 ms (7.2%)
Self time	390 ms (0.7%)
java.util.TreeMap.**put** ()	85.7 ms (0.1%)
cp3.ass01.suffixtrie.SuffixTrieNode.**<init>** ()	106 ms (0.2%)
Self time	0.0 ms (0%)
java.lang.String.**split** ()	94.0 ms (0.2%)
Self time	0.0 ms (0%)
java.util.Scanner.**nextLine** ()	10,090 ms (16.9%)
Self time	0.0 ms (0%)
Self time	0.0 ms (0%)

The CPU profiler snapshot for the same run had different timings, which was strange, but adds a little more to the picture. The in-program profiler (tic, toc, etc) took almost no time. The profiler also shows the invocations, with over 7 million children added from the 15,000-word file, and around 500,000 more indexes added than children (for repeat pathways).

Name	Total Time	Total Time (CPU)	Invocations
main	15,271 ms (100%)	9,204 ms (100%)	4
cp3.ass01.suffixtrie.SuffixTrie.**readInFromFile** (String)	15,253 ms (99.9%)	9,204 ms (100%)	1
cp3.ass01.suffixtrie.SuffixTrie.**insert** (String, int)	15,119 ms (99%)	9,158 ms (99.5%)	635
Self time	6,301 ms (41.3%)	4,060 ms (44.1%)	635
cp3.ass01.suffixtrie.SuffixTrieNode.**addData** (int, int)	4,484 ms (29.4%)	2,546 ms (27.7%)	7,673,470
Self time	2,936 ms (19.2%)	1,395 ms (15.2%)	7,673,470
cp3.ass01.suffixtrie.SuffixTrieData.**addStartIndex** (cp3.ass01.suffixtrie.SuffixIndex)	1,547 ms (10.1%)	1,151 ms (12.5%)	7,673,470
cp3.ass01.suffixtrie.SuffixTrieNode.**getChild** (char)	2,841 ms (18.6%)	1,770 ms (19.2%)	15,346,940
cp3.ass01.suffixtrie.SuffixTrieNode.**addChild** (char, cp3.ass01.suffixtrie.SuffixTrieNode)	1,492 ms (9.8%)	779 ms (8.5%)	7,178,546
Self time	133 ms (0.9%)	46.6 ms (0.5%)	1
cp3.ass01.suffixtrie.SuffixProfiler.**ticMem** ()	17.5 ms (0.1%)	0.0 ms (0%)	1
cp3.ass01.suffixtrie.SuffixProfiler.**tocMem** ()	0.005 ms (0%)	0.0 ms (0%)	1
cp3.ass01.suffixtrie.SuffixProfiler.**bytesToMegabytes** (long)	0.003 ms (0%)	0.0 ms (0%)	1

The memory profiler showed around 3 million SuffixTrieNode objects (75 MB) and equivalent SuffixTrieData objects (50 MB), with around 250,000 more SuffixIndex objects (80 MB). I don't know how to reconcile this with the overall 2 gigabytes of memory used for the program, as well as the 7 million objects recorded for most classes in the other data (for instance in the sampler below).

Profiler Snapshot

Name	Live Bytes	Live Objects
cp3.ass01.suffixtrie.**SuffixIndex**	79,433,880 B (39.3%)	3,309,745 (35.1%)
java.lang.Object.**<init>** ()	79,433,880 B (39.3%)	3,309,745 (35.1%)
cp3.ass01.suffixtrie.SuffixIndex.**<init>** (int, int)	79,433,880 B (39.3%)	3,309,745 (35.1%)
cp3.ass01.suffixtrie.SuffixTrieNode.**addData** (int, int)	79,433,880 B (39.3%)	3,309,745 (35.1%)
cp3.ass01.suffixtrie.SuffixTrie.**insert** (String, int)	79,433,880 B (39.3%)	3,309,745 (35.1%)
cp3.ass01.suffixtrie.SuffixTrie.**readInFromFile** (String)	79,433,880 B (39.3%)	3,309,745 (35.1%)
cp3.ass01.suffixtrie.SuffixTrieDriver.**runMem** (String)	79,433,880 B (39.3%)	3,309,745 (35.1%)
cp3.ass01.suffixtrie.SuffixTrieDriver.**main** (String[])	79,433,880 B (39.3%)	3,309,745 (35.1%)
cp3.ass01.suffixtrie.**SuffixTrieNode**	73,576,152 B (36.4%)	3,065,673 (32.5%)
java.lang.Object.**<init>** ()	73,576,152 B (36.4%)	3,065,673 (32.5%)
cp3.ass01.suffixtrie.SuffixTrieNode.**<init>** ()	73,576,152 B (36.4%)	3,065,673 (32.5%)
cp3.ass01.suffixtrie.SuffixTrie.**insert** (String, int)	73,576,152 B (36.4%)	3,065,673 (32.5%)
cp3.ass01.suffixtrie.SuffixTrie.**readInFromFile** (String)	73,576,152 B (36.4%)	3,065,673 (32.5%)
cp3.ass01.suffixtrie.SuffixTrieDriver.**runMem** (String)	73,576,152 B (36.4%)	3,065,673 (32.5%)
cp3.ass01.suffixtrie.SuffixTrieDriver.**main** (String[])	73,576,152 B (36.4%)	3,065,673 (32.5%)
cp3.ass01.suffixtrie.**SuffixTrieData**	49,050,784 B (24.3%)	3,065,674 (32.5%)
java.lang.Object.**<init>** ()	49,050,784 B (24.3%)	3,065,674 (32.5%)
cp3.ass01.suffixtrie.SuffixTrieData.**<init>** ()	49,050,784 B (24.3%)	3,065,674 (32.5%)
cp3.ass01.suffixtrie.SuffixTrieNode.**<init>** ()	49,050,784 B (24.3%)	3,065,674 (32.5%)
cp3.ass01.suffixtrie.SuffixTrie.**insert** (String, int)	49,050,784 B (24.3%)	3,065,674 (32.5%)
cp3.ass01.suffixtrie.SuffixTrie.**readInFromFile** (String)	49,050,784 B (24.3%)	3,065,674 (32.5%)
cp3.ass01.suffixtrie.SuffixTrieDriver.**runMem** (String)	49,050,784 B (24.3%)	3,065,674 (32.5%)
cp3.ass01.suffixtrie.SuffixTrieDriver.**main** (String[])	49,050,784 B (24.3%)	3,065,674 (32.5%)
cp3.ass01.suffixtrie.**SuffixTrieDriver**	0 B (0%)	0 (0%)
cp3.ass01.suffixtrie.**SuffixProfiler**	0 B (0%)	0 (0%)
cp3.ass01.suffixtrie.**SuffixTrie**	0 B (0%)	0 (0%)

The memory sampler further broke memory into object types. Here we see the expected 7.7 million SuffixIndex objects and 7.2 million SuffixTrieNode objects, as in the CPU profiler. Each SuffixTrieNode had a TreeMap, and each SuffixTrieData has an ArrayList, so it's good to see the number of objects lining up for those.

Profiler Snapshot

Name	Live Bytes	Live Objects
org.graalvm.visualvm.lib.jfluid.server.**ProfilerRuntimeObjLiveness$ProfilerRuntimeObjLivenessWeakRef**	880,960,240 B (32.4%)	22,024,006 (30.1%)
java.lang.**Object[]**	404,019,648 B (14.8%)	7,184,194 (9.8%)
java.util.**TreeMap**	344,424,912 B (12.6%)	7,175,519 (9.8%)
java.util.**TreeMap$Entry**	287,121,920 B (10.5%)	7,178,048 (9.8%)
cp3.ass01.suffixtrie.**SuffixIndex**	184,163,280 B (6.8%)	7,673,470 (10.5%)
java.util.**ArrayList**	172,210,536 B (6.3%)	7,175,439 (9.8%)
cp3.ass01.suffixtrie.**SuffixTrieNode**	172,206,408 B (6.3%)	7,175,267 (9.8%)
java.lang.ref.**WeakReference[]**	131,604,752 B (4.8%)	2 (0%)
cp3.ass01.suffixtrie.**SuffixTrieData**	114,804,272 B (4.2%)	7,175,267 (9.8%)
byte[]	12,430,384 B (0.5%)	222,638 (0.3%)
int[]	11,077,944 B (0.4%)	1,354 (0%)
java.lang.**String**	5,307,432 B (0.2%)	221,143 (0.3%)
java.lang.**Class**	299,832 B (0%)	2,483 (0%)
char[]	170,576 B (0%)	478 (0%)
java.util.**HashMap$Node**	161,280 B (0%)	5,040 (0%)
java.lang.reflect.**Method**	151,976 B (0%)	1,727 (0%)
java.lang.**Class$ReflectionData**	123,840 B (0%)	1,935 (0%)
java.lang.**Character**	108,064 B (0%)	6,754 (0%)
java.lang.**Class[]**	94,528 B (0%)	4,309 (0%)
java.lang.ref.**SoftReference**	85,240 B (0%)	2,131 (0%)
java.lang.ref.**WeakReference**	77,920 B (0%)	2,435 (0%)
java.util.**HashMap$Node[]**	74,624 B (0%)	606 (0%)
java.util.concurrent.**ConcurrentHashMap$Node**	68,352 B (0%)	2,136 (0%)
java.io.**ObjectStreamClass$WeakClassKey**	64,064 B (0%)	2,002 (0%)
java.util.**HashMap**	35,184 B (0%)	733 (0%)
java.lang.**String[]**	32,904 B (0%)	899 (0%)

My final commit before starting the extensions was **1906**.

Task 2: Exact String Matching - Extensions (12%)

In these extensions, I wanted to try three things:

1. Extend my Suffix Trie to capture the whole text in each suffix
2. Build my own Suffix Tree from scratch
3. Attempt to build a Ukkonen Suffix Tree

For each extension, I wanted to be able to return exactly the same data, i.e. the locations of a substring in the original text, by sentence and character.

Extension 1: Suffix Trie without sentence breaks

I wanted to make a suffix trie that treated a whole document as one string. This would mean I could search for multi-sentence substrings and search across the sentence markers (.!?).

This would make much longer suffixes and much bigger suffix tries overall. There should be the same number of substrings, but each substring would continue to the end of the document. The read-in operation would be in $O(n^2)$ time and memory required would also be $O(n^2)$. This would be used to compare with my own suffix tree and Ukkonen's suffix tree, which I implement below.

I made changes to the SuffixTrie_FullText to read in the full text in one string.

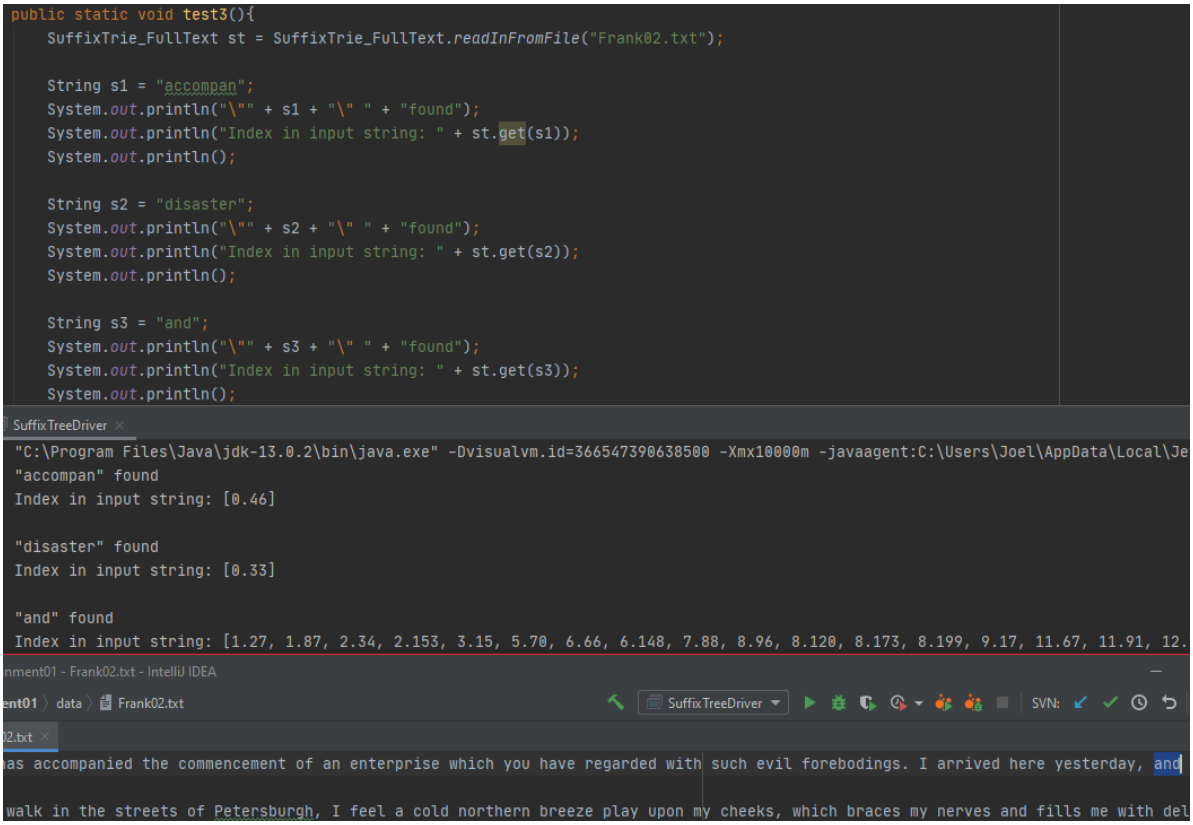
```
/** Helper/Factory static method to build a SuffixTrie object from the text in the given file. ...*/
public static SuffixTrie_FullText readInFromFile(String fileName) {
    SuffixTrie_FullText st = new SuffixTrie_FullText();
    Scanner fileScanner;
    try {
        fileScanner = new Scanner(new FileInputStream( name: "data" + File.separator + fileName));
        StringBuilder sb = new StringBuilder();
        while(fileScanner.hasNextLine()){
            sb.append(fileScanner.nextLine());
        }
        st.input = sb.toString();
        st.insert( startPosition: 0);
    } catch (FileNotFoundException ex) {
        System.out.println("could not find the file " +fileName+ " in the data directory!");
        return null;
    }
    return st;
}
```

I also added some code to maintain the sentence and character data indexing for each suffix, so I could compare the results with the previous SuffixTrie.

```
if(input.substring(i,i+1).matches( regex: "[!?.]")) {
    sentence++;
    ch = 0;
}
else{
    ch++;
}
```

I continued to use the same SuffixTrieData and SuffixIndex classes for this SuffixTrie.

The results of my test show this working as expected.



I moved on to implementing my own suffix tree.

Extension 2: Suffix Tree

I had read a little about Ukkonen's Algorithm already, but I wanted to try to design this from scratch. I did take the idea of each node just holding indexes to a string, rather than the actual characters, to significantly decrease memory required. This also required a nested Node, that could directly access the input string.

I didn't want to make it too complicated, so I resigned myself to looping over each character in an input string, and then for each character, looping over the remaining string. This would make my input algorithm $O(n^2)$ at worst, although in practice it should be a fair bit better for two reasons:

1. When I reach a Node without the right child, I will just add the rest of the string by setting the ending index to the end of the string. This will be a single operation.
2. As I move through the string in my outer loop, there are less characters remaining. So, on average, my inner loop will run n/2 times.

There are three additional challenges to making this suffix tree versus making the suffix trie.

1. Searching for characters and prefixes within and across Nodes with multiple characters inside
2. Splitting Nodes
3. Saving and returning the starting location of suffixes

The first task was to copy over useful code from my SuffixTrie_FullText. This included most of the readIn() function and a substantial amount of the insert() function.

I had to make some changes to the Nodes for the SuffixTree. To begin, these would now be nested inside the SuffixTree class, so they could directly access the input string. The Node also needed *begin* and *end* ints to represent the Node substring in the input string. I still used a TreeMap to hold links to child Nodes and SuffixTrieData and SuffixIndexes to store the starting location of my suffixes.

```
class Node {
    private int begin;
    private int end;
    private SuffixTrieData data = new SuffixTrieData();
    private Map<Character, Node> children = new TreeMap<>();
```

I also needed two constructors, one to handle nodes that run to the end of input, and one to handle internal nodes that only run for a subset of the input.

```
//constructor for nodes that run to the end of input
Node(int begin){
    this.begin = begin;
    this.end = length;                                          // length will be 1 past the end of the string;
}

//constructor for internal nodes that run for a subset of input
Node(int begin, int end){
    this.begin = begin;
    this.end = end;                                            // these internal nodes don't go to end of input
}
```

Finally, I needed the methods to add and get children, and to add suffix starting location data to the Node.

```
/** Lookup a child node of the current node that is associated with the character label. ...*/
public Node getChild(char label) {
    return children.get(label);
}

/** Add a child node to the current node, at the position associated with the provided label. ...*/
public void addChild(char label, Node node) {
    children.put(label, node);
}

/** Add a new SuffixIndex to the SuffixTrieData associated with this node. ...*/
public void addData(int sentencePos, int characterPos) {
    SuffixIndex si = new SuffixIndex(sentencePos, characterPos);
    data.addStartIndex(si);
}
```

Before I wrote the insert() code, I wanted to be able to check my suffix tree, so I wrote the get() method and the printTree() method. This needed to be able to find the end of a substring inside a Node.

```
public Node get(String str){
    Node itNode = root;
    for(int i = 0; i < str.length(); i++){
        if(itNode.getChild(str.charAt(i)) == null){
            return null;
        }
        itNode = itNode.getChild(str.charAt(i));
        int min = Math.min(str.length() - i, itNode.length());
        if(str.substring(i,i+min).equals(itNode.substr(min))){
            if(min == str.length() - i){
                return itNode;
            }
            else { i += min -1; }
        }
        else { return null; }
    }
    return null;
}
```

```
public void printTree(){
    printNode = root;
    if(!printNode.getChildren().isEmpty()){
        System.out.println("\"" + printNode.substr() + "\": ");
        System.out.println(printNode.getChildren());
    }
    for(Map.Entry<Character, Node> child : printNode.getChildren().entrySet()){
        printNode = child.getValue();
        printTree();
    }
}
```

Writing the insert algorithm was really difficult (took around 12 hrs), but by carefully stepping through each case I came up with the following insert() algorithm. It has an out loop (*i*), inner loop (*j*) and a while loop that iterates through the strings inside nodes but also advances *j* (so it should still be $O(n^2)$).

As mentioned, the two main additions from the SuffixTrie are the node splitting and the suffix starting index data management.

```
for(int i = 0; i < input.length(); i++) {                                    //
    itNode = root;                                                           //
    for (int j = i; j < input.length(); j++) {                               //
        while(j < itNode.end){                                               //
            char oldChar = input.charAt(itNode.begin + offset);              //
            if(input.charAt(j) != oldChar){                                  //
                // split the nodes
                    Node splitA = new Node(itNode.begin + offset, itNode.end);
                    splitA.children = itNode.children;
                    itNode.children = new TreeMap<>();
                    itNode.addChild(input.charAt(itNode.begin + offset),splitA)
                    itNode.end = itNode.begin + offset;
                // copy suffix index data across, except most recent one added
                    for(SuffixIndex si : itNode.data.getArray()){
                        splitA.data.addStartIndex(si);                       //
                    }
                    splitA.data.removeLastIndex();                           //
                    break;
            }
            offset++;                                                        //
            j++;                                                             //
        }
        // after this while loop, we will be at the end of the substring for th
        if(itNode.getChild(input.charAt(j)) == null) {                       //
            itNode.addChild(input.charAt(j), new Node(j));                   //
            itNode = itNode.getChild(input.charAt(j));                       //
            itNode.addData(sentence, ch);                                    //
            break;                                                           //
        }
        itNode = itNode.getChild(input.charAt(j));                           //
        offset = 1;                                                          //
        itNode.addData(sentence, ch);                                        //
    }
```

I also added sentence and character indexing:

```
if(input.substring(i,i+1).matches( regex: "[!?.]")) {
    sentence++;
    ch = 0;
}
else{
    ch++;
}
```

Some quick testing showed my suffix tree performing as expected. I entered this code:

```
SuffixTree st = new SuffixTree();
st.setInput("abcd.cde");
st.insert();
st.printTree();
System.out.println(st.get("cd"));
```

It created this output. If you work through each node you can see it is the correct structure.

```
Node string: "root":
Starting indexes: []
Children: [., a, b, c, d, e]

Node string: ".cde":
Starting indexes: [0.4]
Children: []

Node string: "abcd.cde":
Starting indexes: [0.0]
Children: []

Node string: "bcd.cde":
Starting indexes: [0.1]
Children: []

Node string: "cd":
Starting indexes: [0.2, 1.0]
Children: [., e]

Node string: ".cde":
Starting indexes: [0.2]
Children: []

Node string: "e":
Starting indexes: [1.0]
Children: []

Node string: "d":
Starting indexes: [0.3, 1.1]
Children: [., e]

Node string: ".cde":
Starting indexes: [0.3]
Children: []

Node string: "e":
Starting indexes: [1.1]
Children: []

Node string: "e":
Starting indexes: [1.2]
Children: []

[0.2, 1.0]
```

I now tested the code on larger input strings. My “test_data.txt” contained multiple words and lines:

```
mississippi. minimizes.

missing. minniminimo.
```

Searching for “mi” returned this, which you can see is the correct result.

```
[0.0, 1.1, 1.5, 2.0, 3.1, 3.6]
```

Next I tested it on Frank01.txt, searching for “de”. This is also the correct result.

```
[0.112, 1.61, 1.107, 1.136]
```

Finally, I compared the results of SuffixTrie_FullText with SuffixTree, both testing on Frank02.txt.

```
//TESTING the SuffixTrie_FullText readIn, insert() and get()
test1();

//TESTING the readIn(), insert() and get() function of the SuffixTree
test2();
```

```
SuffixTreeDriver
"C:\Program Files\Java\jdk-13.0.2\bin\java.exe" -Dvisualvm.id=371508480255000 -Xmx10000m -javaagent:C:\Users\Joel\AppData\Local\JetBra
Testing the SuffixTrie_FullText
Reading in "Frank02.txt"...

"accompan" found
Index in input string: [0.46]

"disaster" found
Index in input string: [0.33]

"and" found
Index in input string: [1.27, 1.87, 2.34, 2.153, 3.15, 5.70, 6.66, 6.148, 7.88, 8.96, 8.120, 8.173, 8.199, 9.17, 11.67, 11.91, 12.97,

Testing the SuffixTree
Reading in "Frank02.txt"...

"accompan" found
Index in input string: [0.46]

"disaster" found
Index in input string: [0.33]

"and" found
Index in input string: [1.27, 1.87, 2.34, 2.153, 3.15, 5.70, 6.66, 6.148, 7.88, 8.96, 8.120, 8.173, 8.199, 9.17, 11.67, 11.91, 12.97,
```

I considered my SuffixTree completed and did some profiling. As mentioned, I expected the SuffixTrie_FullText to perform much worse than before. My SuffixTree should read much faster. For these tests I again set the VM options to -Xmx10000m.

Results Table: readInFromFile () - Time

filename	**SuffixTree (average of 10 runs, ms)**	**SuffixTrie_FullText (average of 10 runs, ms)**	**Word Count**
Frank01.txt	<1	16	49
Frank02.txt	5	776	435
FrankChap02.txt	95	crashed (out of heap space)	9,544
FrankChap04.txt	170	crashed (out of heap space)	14,767
FrankMed.txt	977	crashed (out of heap space)	62,208
Frankenstien.txt	1145	crashed (out of heap space)	75,085

Results Table: readInFromFile () - Memory

filename	**SuffixTree (megabytes)**	**SuffixTrie_FullText (megabytes)**	**Word Count**
Frank01.txt	<1	9	49
Frank02.txt	3	701	435
FrankChap02.txt	33	crashed (out of heap space)	9,544
FrankChap04.txt	49	crashed (out of heap space)	14,767
FrankMed.txt	242	crashed (out of heap space)	62,208
Frankenstien.txt	284	crashed (out of heap space)	75,085

These results show my SuffixTree performing better than I expected. I thought it would have readIn time and memory space approaching $O(n^2)$, but it seems closer to O(n). I think because I'm using indexing into the input string, it is very space and time efficient. For example, if I'm adding an external node from character 10 to 400,000, this is just one simple operation and the memory required is two integers.

I moved on to implementing Ukkonen's suffix tree.

Extension 3: Ukkonen's Algorithm for building a Suffix Tree

I started reading up on this algorithm, and quickly realised how difficult it might be to implement. To construct it, I followed a really good answer on StackOverflow.[1] This didn't provide code but had a step by step explanation that helped explain practically what was happening.

When I got really stuck, I referenced a Ukkonen Suffix Tree built in Java by Daniel Mariselli on GitHub.[2] I wrote mine from scratch however, I didn't copy any code in.

I started by building my nodes. I knew they would need a beginning and ending index, that related to the position of a substring in the input string. The beginning index would be set at creation, but the ending index would need to be updated every step of the parsing process. One trick of the Ukkonnen algorithm is that if all the ending indexes were pointers, they could point to the same location that would only need to be updated once. As Java doesn't have pointers, I used the same method as Mariselli, which was to set all the ending indexes to -1 initially, and only update them when reading the node.

My nodes would also need a way to hold child nodes. As before, I used a TreeMap for this. My nodes should also hold the starting indexes for any substrings that pass the node. As before I used SuffixTrieData and SuffixIndexes for this.

```
class UkkonenNode {
    private int begin;
    private int end;
    private List<Integer> data = new ArrayList<>();
    private Map<Character, UkkonenNode> children = new TreeMap<>();

    //constructor for nodes;
    UkkonenNode(int i){
        begin = i;
        end = -1;
    }
```

Next, I added some getter and setter methods to the node, including *addChild()*, *getChild()*, *addData()*, *getData()*, *subStr()* and *toString()*. I implemented an insertion method in UkkonenTree. I kept it simple at this stage, just inserting all characters at root.

```
public class UkkonenTree {

    private String input;
    private UkkonenNode root;
    private int step;

    public UkkonenTree(){
        root = new UkkonenNode( i: -1);
        step = 0;
    }

    public void insert(String str){
        this.input = str;
        UkkonenNode itNode = root;
        for(int i = 0; i < input.length(); i++){
            itNode.addChild(input.charAt(i), new UkkonenNode(i));
            itNode.getChild(input.charAt(i)).addData(i);
            step++;
        }
        System.out.println("step = " + step);
    }
```

[1] https://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english/9513423
[2] https://github.com/dmariselli/Ukkonen-Suffix-Tree

Finally, I implemented a *get()* method for getting nodes out of the suffix tree. This code was similar to my SuffixTree code. It should work whether *str* is longer or shorter than the substring referenced by each node it passes.

```
public UkkonenNode get(String str){
    UkkonenNode itNode = root;
    for(int i = 0; i < str.length(); i++){
        if(itNode.getChild(str.charAt(i)) == null){
            return null;
        }
        itNode = itNode.getChild(str.charAt(i));
        int min = Math.min(str.length() - i, itNode.length());
        if(str.substring(i,min).equals(itNode.subStr(min))){
            if(min == str.length() - i){
                return itNode;
            }
            else { i += min -1; }
        }
        else { return null; }
    }
    return null;
}
```

This passed my testing in the Ukkonen driver.

```
//TESTING the UkkonenTree
UkkonenTree uke = new UkkonenTree();
uke.insert( str: "abcdefg");

String s1 = "bcd";
System.out.println("\"" + s1 + "\" " + "found in node: " + uke.get(s1));
System.out.println("Index in input string: " + uke.get(s1).getData());
System.out.println();

String s2 = "ab";
System.out.println("\"" + s2 + "\" " + "found in node: " + uke.get(s2));
System.out.println("Index in input string: " + uke.get(s2).getData());
System.out.println();

String s3 = "ba";
System.out.println("\"" + s3 + "\" " + "found in node: " + uke.get(s3));
try {
    System.out.println("Index in input string: " + uke.get(s3).getData());
} catch (NullPointerException n) {
    System.out.println("String not found");
}
```

```
UkkonenDriver
"C:\Program Files\Java\jdk-13.0.2\bin\java.exe" -Dvisualvm.id=100010185885200 -ja
step = 7
"bcd" found in node: bcdefg
Index in input string: [1]

"ab" found in node: abcdefg
Index in input string: [0]

"ba" found in node: null
String not found

Process finished with exit code 0
```

Next, I needed to handle repeated characters in the input string. My insert function needed to get a lot more sophisticated. First, I needed four new variables. The three *ActivePoint* variables keep track of where I need to insert the next node. The remainder variable keeps track of how many suffixes I still need to insert. It's possible not to insert a suffix each step if the start of that suffix already exists.

```
private UkkonenNode activeNode;
private char activeChar;
private int activeLength;
private int remainder;

public UkkonenTree(){
    root = new UkkonenNode( begin: -1);
    step = 0;
    activeNode = root;
    activeChar = '\0';
    activeLength = 0;
    remainder = 1;
}
```

Implementing the remaining logic of Ukkonen's algorithm proved very difficult. I started to refer to Mariselli's example more and more. I wrote my own code and filled in comments about what was happening at each step, but it became very similar in structure to Mariselli's. I decided to focus on making it work and then extending it for my purposes.

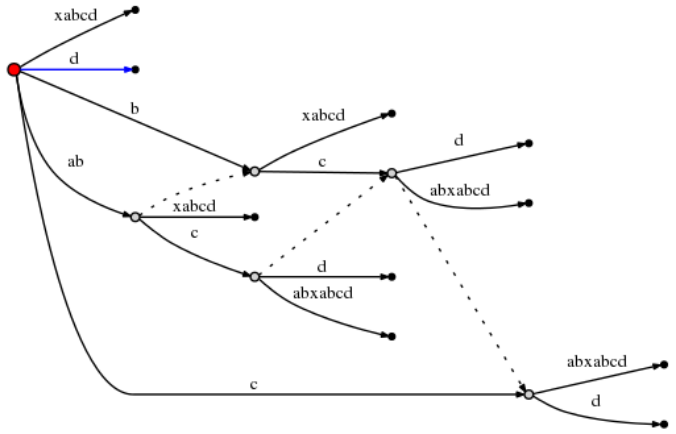
My first extension was to update my *get()* function above, as it had some errors in it. When this was fixed, I was able to see what my Tree had in it after reading in a string. I was also able to see some error's in Mariselli's code. For example, when he split a node, he only added one outgoing pathway, when a split is supposed to create two outgoing pathways.

```
// splitting nodes to make internal node
int splitEnd = next.begin + activeLength - 1;                    // create ending index for internal node
UkkonenNode split = new UkkonenNode(next.begin, splitEnd);       // create internal node
activeNode.addChild(activeChar, split, i);                       // add internal node to active node children
next.begin += activeLength;                                      // increase the next node beginning index by activeLength
split.addChild(input.charAt(next.begin), next, i);               // add next node to the internal node children
split.addChild(input.charAt(i), new UkkonenNode(i), i);          // this child wasn't being created before.
```

I copied over my simple recursive ***printTree()*** method from my SuffixTree to see what was in the Tree.

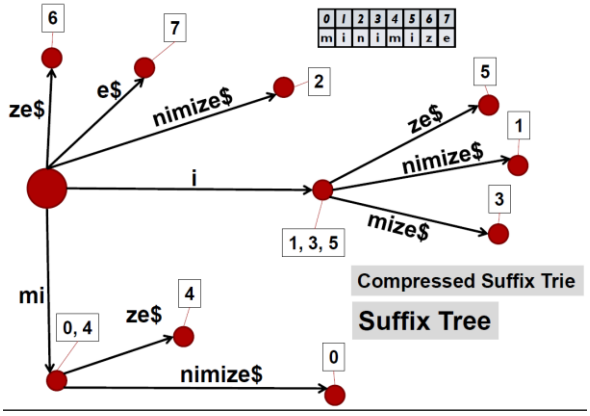
```
public void printTree(){
    if(!printNode.getChildren().isEmpty()){
        System.out.println("\"" + printNode.toString() + "\": ");
        System.out.println(printNode.getChildren());
    }
    for(Map.Entry<Character,UkkonenNode> child : printNode.getChildren().entrySet()){
        printNode = child.getValue();
        printTree();
    }
}
```

According to the StackOverflow article, the final Suffix Tree for the input string "abcabxabcd" should be like the image below. My *printTree()* output is listed for comparison (I use a $ to terminate the input).




```
"root":
{$=$, a=ab, b=b, c=c, d=d$, x=xabcd$}
"ab":
{c=c, x=xabcd$}
"c":
{a=abxabcd$, d=d$}
"b":
{c=c, x=xabcd$}
"c":
{a=abxabcd$, d=d$}
"c":
{a=abxabcd$, d=d$}

Process finished with exit code 0
```

Alternatively, using the lecture example below also shows it is working as expected.




```
"root":
{$=$, e=e$, i=i, m=mi, n=nimize$, z=ze$}
"i":
{m=mize$, n=nimize$, z=ze$}
"mi":
{n=nimize$, z=ze$}

Process finished with exit code 0
```

I also added a *readInFromFile()* method so I could read in the Frankenstien texts. This is similar to my SuffixTree which reads in a whole file as one string.

```
public void readInFromFile(String fileName){
    try {
        Scanner fileScanner = new Scanner(new FileInputStream( name: "data" + File.separator + fileName));
        StringBuilder sb = new StringBuilder();
        while(fileScanner.hasNextLine()){
            sb.append(fileScanner.nextLine());
        }
        sb.append("$");                          // this character marks the end of the input string.
        insert(sb.toString());
    } catch (FileNotFoundException ex) {
        System.out.println("could not find the file " +fileName+ " in the data directory!");
    }
}
```

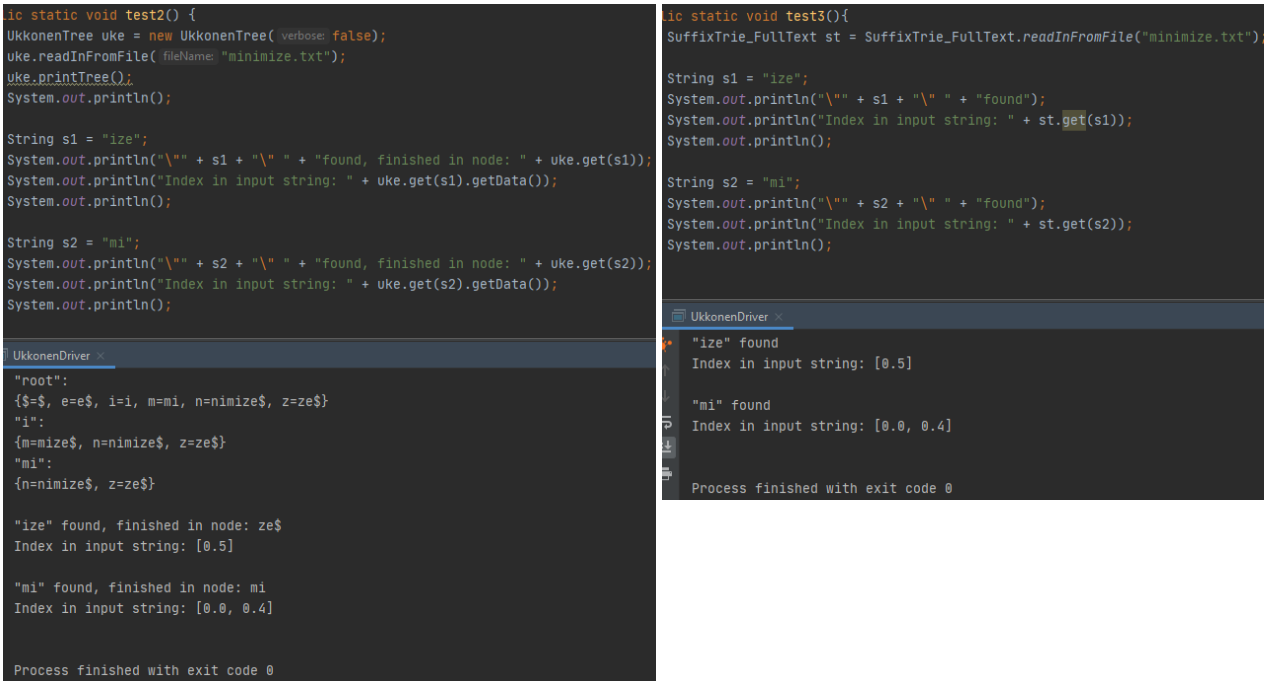
I now wanted to add indexes to where the substrings could be found in the original input, so I could compare this to the SuffixTrie results. This was quite difficult with Ukkonen's algorithm, particularly with the splitting nodes. An index needed to be registered anytime a character was added or node split. There were multiple places to add this code. Firstly, when the activeNode didn't have the activeChar.

```
// if the child nodes of the activeNode don't have the activeChar, make a new node for that path
if (!activeNode.contains(activeChar)) {
    activeNode.addChild((activeChar), new UkkonenNode(i));
    activeNode.getChild(activeChar).addData(i);
    if(activeNode != root) activeNode.addData( index: i - remainder + 1);  // add relevant starting index
```

Then in the splitting process:

```
// splitting nodes to make internal node
int splitEnd = next.begin + activeLength - 1;
UkkonenNode split = new UkkonenNode( begin: splitEnd + 1);
next.end = splitEnd;
activeNode.addChild(activeChar, next);
next.addData( index: i - remainder + 1);
next.addChild(input.charAt(split.begin), split);
split.addData(next.begin);
next.addChild(input.charAt(i), new UkkonenNode(i));
next.getChild(input.charAt(i)).addData( index: i - remainder + 1);
```

The indexing of substrings worked correctly in my tests. Here is the comparison with SuffixTrie_FullText.



```
lic static void test2() {
UkkonenTree uke = new UkkonenTree( verbose: false);
uke.readInFromFile( fileName: "minimize.txt");
uke.printTree();
System.out.println();

String s1 = "ize";
System.out.println("\"" + s1 + "\" " + "found, finished in node: " + uke.get(s1));
System.out.println("Index in input string: " + uke.get(s1).getData());
System.out.println();

String s2 = "mi";
System.out.println("\"" + s2 + "\" " + "found, finished in node: " + uke.get(s2));
System.out.println("Index in input string: " + uke.get(s2).getData());
System.out.println();
```

```
UkkonenDriver
 "root":
 {$=$, e=e$, i=i, m=mi, n=nimize$, z=ze$}
 "i":
 {m=mize$, n=nimize$, z=ze$}
 "mi":
 {n=nimize$, z=ze$}

 "ize" found, finished in node: ze$
 Index in input string: [0.5]

 "mi" found, finished in node: mi
 Index in input string: [0.0, 0.4]


 Process finished with exit code 0
```

```
lic static void test3(){
SuffixTrie_FullText st = SuffixTrie_FullText.readInFromFile("minimize.txt");

String s1 = "ize";
System.out.println("\"" + s1 + "\" " + "found");
System.out.println("Index in input string: " + st.get(s1));
System.out.println();

String s2 = "mi";
System.out.println("\"" + s2 + "\" " + "found");
System.out.println("Index in input string: " + st.get(s2));
System.out.println();
```

```
UkkonenDriver
 "ize" found
 Index in input string: [0.5]

 "mi" found
 Index in input string: [0.0, 0.4]


 Process finished with exit code 0
```

I profiled my two suffix tree algorithms, with results below.

Results Table: readInFromFile () - Time

filename	**Ukkonen's Algorithm (average of 10 runs, ms)**	**SuffixTree (average of 10 runs, ms)**	**Word Count**
Frank01.txt	1	<1	49
Frank02.txt	2	5	435
FrankChap02.txt	9	95	9,544
FrankChap04.txt	13	170	14,767
FrankMed.txt	27	977	62,208
Frankenstien.txt	33	1145	75,085

Results Table: readInFromFile () - Memory

filename	**Ukkonen's Algorithm (megabytes)**	**SuffixTree (megabytes)**	**Word Count**
Frank01.txt	1	<1	49
Frank02.txt	2	3	435
FrankChap02.txt	3	33	9,544
FrankChap04.txt	2	49	14,767
FrankMed.txt	5	242	62,208
Frankenstien.txt	4	284	75,085

Clearly there is something wrong with my Ukkonen algorithm. It seems to stop working after a certain number of words. At this point I think I am beyond my ability (and time) to troubleshoot it.

I ran the tests again, this time with Daniel Mariselli's ExampleUke to get these results.

Results Table: readInFromFile () - Time

filename	**Mariselli's Ukkonen Tree (average of 10 runs, ms)**	**SuffixTree (average of 10 runs, ms)**	**Word Count**
Frank01.txt	0	<1	49
Frank02.txt	1	5	435
FrankChap02.txt	39	95	9,544
FrankChap04.txt	56	170	14,767
FrankMed.txt	251	977	62,208
Frankenstien.txt	374	1145	75,085

Results Table: readInFromFile () - Memory

filename	**Mariselli's Ukkonen Tree (megabytes)**	**SuffixTree (average of 10 runs, ms)**	**Word Count**
Frank01.txt	0.2	<1	49
Frank02.txt	0.5	3	435
FrankChap02.txt	12	33	9,544
FrankChap04.txt	19	49	14,767
FrankMed.txt	78	242	62,208
Frankenstien.txt	93	284	75,085

Even though Mariselli's suffix tree built with the Ukkonen Algorithm is definitely faster and uses less memory, I am still pleased at the performance of my algorithm. The difference in memory used is interesting, because the two suffix trees should have a fairly similar node structure at the end. This is possibly due to my SuffixTree having quite a bit of overhead, particularly for storing the starting index data of each suffix.

My final commit is: **2372**.