

# **B-Trees**

By: Matthew S Montoya

## 1 Introduction

In this lab, a program will read a file containing the English language dictionary, and store the words in a B-Tree data structure. For each word within the B-Tree that is less than 8 characters long, the program will identify all the anagrams for that word. In addition, the program will identify the word with the most number of anagrams and print them out, as well as the time (in seconds) it took to find that word.

## 2 Proposed Solution & Design Implementation

To achieve the end goal, the program will contain code similar to that given in class, with a few modifications (such as using String objects for keys [rather than integers] and comparing the Strings using *.compareToIgnoreCase*). This lab will be split across three different programs, Lab04.java, BTree.java, and BTreeNode.java.

### 2.1— Reading the file

Within the main method in the Lab04 class, the program will create a BTree data structure, calling the BTree class. It will then create an instance of BTree with the integer n value set to 100,000, and call a separate method (*fillWordsSet*) in Lab04 to read the file. Using *BufferedReader* and *FileReader*, the program will read the file and fill the BTree with the words.

### 2.2— Finding the number of anagrams in a word

Within the main method of Lab04 class, the program will ask the user for the word for which they would like to find the anagrams for. Using a scanner, the program will read the user's input and calls the *printAnagrams* method from within the same class. This method will identify anagrams with 8 or less characters, printing out the anagrams and return an integer value to the main method, where it will inform the user how many anagrams were found.

### 2.3, 2.4— Displaying the word with the most anagrams & calculate the time it took to find the word

Within the main method of the Lab04 class, the program will call the *mostAnagrams* method, passing through the parameters of the BTree, a String (representing the word with the most number of anagrams), and an integer (with the total number of anagrams). This method will be utilizing loops and recursion and upon initial call, will send a null String and int value of 0. After traversing the B-Tree and finding the word with the most number of anagrams (and the total value), the method will then display this information for the user.

To calculate the amount of time to find this information, the program will record the system time prior to calling the *mostAnagrams* method, and will record the time again

when the method returns to the main method. The difference between the two time stamps will be calculated and displayed for the user to see.

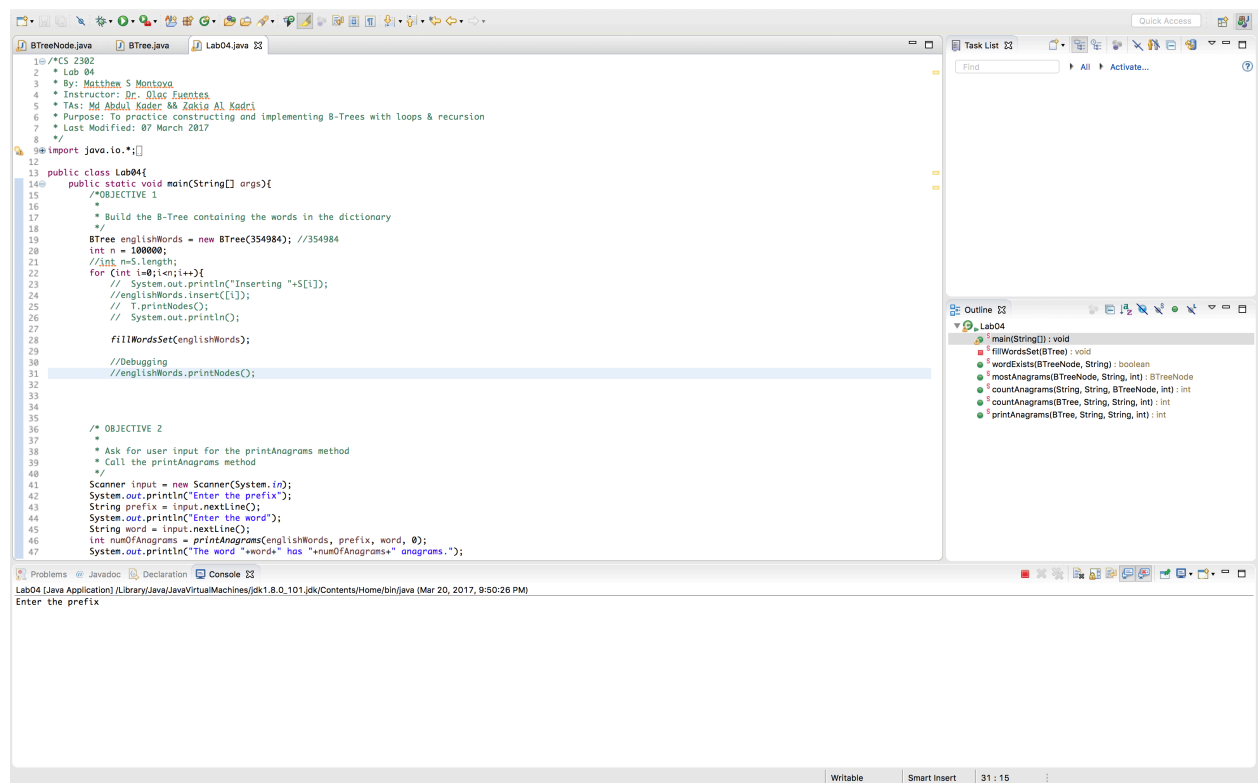
## 2.4— Printing the nodes at depth d or less

Adding a method to the BTreeNode class (which is called from the Lab04 class through the BTree class), this method will recursively call itself until the depth level is at zero, passing the child node with each call. If the depth does not exist, the user will be informed on screen, otherwise, the nodes will be listed for the user to see.

## 3 Experimental Results

### 3.1— Program Output

The output of the code (found in the appendix) is shown below. The first two images are from the same test. Note: The first image includes the beginning of the code (which can also be found in the *Appendix* section), as proof the executed code is not the work of others.



The screenshot displays an IDE with three tabs: BTreeNode.java, BTree.java, and Lab04.java. The Lab04.java tab is active, showing the following code:

```
1  //CS 2302
2  * Lab 04
3  * By: Matthew S Montoya
4  * Instructor: Dr. Olac Fuentes
5  * Tas: Md Abdul Kader, Md Zakia Al Kadri
6  * Purpose: To practice constructing and implementing B-Trees with loops & recursion
7  * Last Modified: 07 March 2017
8  */
9  import java.io.*;
10
11
12
13 public class Lab04{
14     public static void main(String[] args){
15         /*OBJECTIVE 1
16         *
17         * Build the B-Tree containing the words in the dictionary
18         */
19         BTree englishWords = new BTree(354984); //354984
20         int n = 100000;
21         //int n=5.length;
22         for (int i=0;i<n;i++){
23             // System.out.println("Inserting "+s[i]);
24             englishWords.insert(i);
25             // T.printNodes();
26             // System.out.println();
27
28             fillWordsSet(englishWords);
29
30             //Debugging
31             englishWords.printNodes();
32
33
34
35
36         /* OBJECTIVE 2
37         *
38         * Ask for user input for the printAnagrams method
39         * Call the printAnagrams method
40         */
41         Scanner input = new Scanner(System.in);
42         System.out.println("Enter the prefix");
43         String prefix = input.nextLine();
44         System.out.println("Enter the word");
45         String word = input.nextLine();
46         int numOfAnagrams = printAnagrams(englishWords, prefix, word, 0);
47         System.out.println("The word "+word+" has "+numOfAnagrams+" anagrams.");
48     }
49 }
```

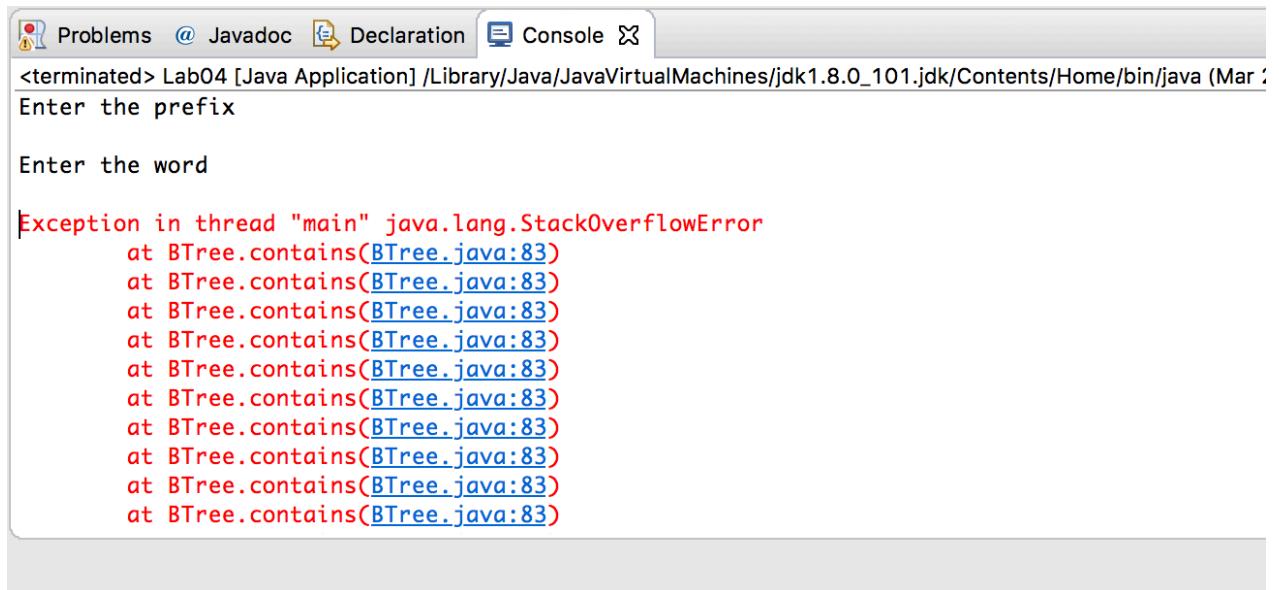
The right sidebar shows the Outline view with the following methods listed:

- main(String[]): void
- fillWordsSet(BTree): void
- wordExists(BTreeNode, String): boolean
- mostAnagrams(BTreeNode, String, int): BTreeNode
- countAnagrams(String, BTreeNode, int): int
- countAnagrams(BTree, String, String, int): int
- printAnagrams(BTree, String, String, int): int

The bottom of the IDE shows the Console view with the following output:

```
Lab04 [Java Application] /Library/Java/JavaVirtualMachines/dk1.8.0_101.jdk/Contents/Home/bin/java (Mar 20, 2017, 9:50:26 PM)
Enter the prefix
```

The program execution



The screenshot shows an IDE console window with tabs for Problems, Javadoc, Declaration, and Console. The console output indicates a terminated Java application and prompts for a prefix and word. It then displays a stack overflow error in red text, showing a recursive call to `BTree.contains` at line 83 of `BTree.java` repeated 10 times.

```
<terminated> Lab04 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (Mar :
Enter the prefix

Enter the word

Exception in thread "main" java.lang.StackOverflowError
    at BTree.contains(BTree.java:83)
    at BTree.contains(BTree.java:83)
    at BTree.contains(BTree.java:83)
    at BTree.contains(BTree.java:83)
    at BTree.contains(BTree.java:83)
    at BTree.contains(BTree.java:83)
    at BTree.contains(BTree.java:83)
    at BTree.contains(BTree.java:83)
    at BTree.contains(BTree.java:83)
    at BTree.contains(BTree.java:83)
```

A stack overflow error thrown at line 83 while running the program.

```
82         }
83     } catch (IOException e) {System.out.println("Error reading the file: " + e.getMessage());}
84 }
```

Line 83 of the code where the error was thrown

## 4 Conclusions

In this lab, I learned how to properly read text files, something I have previously been able to do in my previous CS classes. In addition, I learned how to more efficiently run a program, managing three separate classes. However, one of the areas where I did not learn as much as I had hoped was in properly implementing B-Trees. After debugging, it appears I could properly populate a B-Tree data structure by reading a text file, but not much else. The one area which I learned I need to focus on is learning how to pass references to the children nodes and distinguish them from one another (essentially, keeping track of variables while working with loops and recursion). Although I was unable to get further than the Stack Overflow error, I hope to learn how to improve my errors during the demo session, for the sake of learning for future implementations of B-Trees, both in the classroom and in the real world.

## 5 Appendix

### 5.1—Documentation & Disclosure

- Methods are documented using the following format—

```
/**METHOD NUMBER:
*
```

```
* BRIEF MENTION OF METHOD'S PURPOSE
```

```
*/
```

```
public static void methodName(parameters){  
}
```

- Non-obvious code is documented using the following format—  
for(int i=0; i<x.length-1; i++){ //summarizes the function of the loop  
x[i] = i; //may include brief logical reasoning  
}
- Disclosure: Though keeping the original digital format of the code (indention lengths, spaces, etc.) the use of Microsoft Word has reformatted the code in the document to be of different space and tab lengths. The font size has been reduced to help alleviate some visual misrepresentations.

## 5.2— Student Code (*Lab04.java*)

```
import java.io.*;  
import java.util.*;  
import java.lang.*; //Should take care of the .contains method (in theory)  
  
public class Lab04{  
    public static void main(String[] args){  
        /*OBJECTIVE 1  
        *  
        * Build the B-Tree containing the words in the dictionary  
        */  
        BTree englishWords = new BTree(354984); //354984  
        int n = 100000;  
        //int n=S.length;  
        for (int i=0;i<n;i++){  
            // System.out.println("Inserting "+S[i]);  
            //englishWords.insert([i]);  
            // T.printNodes();  
            // System.out.println();  
  
            fillWordsSet(englishWords);  
  
            //Debugging  
            //englishWords.printNodes();  
  
        }  
  
        /* OBJECTIVE 2  
        *  
        * Ask for user input for the printAnagrams method  
        * Call the printAnagrams method  
        */  
        Scanner input = new Scanner(System.in);  
        System.out.println("Enter the prefix");  
        String prefix = input.nextLine();  
        System.out.println("Enter the word");  
        String word = input.nextLine();  
        int numOfAnagrams = printAnagrams(englishWords, prefix, word, 0);  
    }  
}
```

```

        System.out.println("The word "+word+" has "+numOfAnagrams+"
anagrams.");

        /**
         * OBJECTIVES 3 + 4
         *
         * Call the mostAnagrams method
         * Time how long it will take to find and print out the result
         */
        //////////////////////////////////////
        long timeStartMillis1 = System.currentTimeMillis();           //
begin logging time
        long timeStartNano1 = System.nanoTime();                     //
        //////////////////////////////////////

        //englishWordsSet.mostAnagrams(englishWords, "null", 0);
//Sort via insertion sort

        //////////////////////////////////////
        long timeEndMillis1 = System.currentTimeMillis();           //   End
logging time
        long timeEndNano1 = System.nanoTime();                     //
        //////////////////////////////////////

        System.out.println("It took "+(timeEndMillis1-timeStartMillis1)+"ms to
find the result.");

        input.close();
    }
}

```

```

        private static void fillWordsSet(BTree englishWords) {
            try (BufferedReader br = new BufferedReader(new
FileReader("words.txt"))){
                String currentLine;
                while ((currentLine = br.readLine()) != null) {
                    englishWords.insert(currentLine);
                }
            } catch (IOException e) {System.out.println("Error reading the file:
"+ e.getMessage());}
        }
    }
}

```

```

/**
 * ADDED Method wordsExist
 *
 * This method will determine if a word exists.
 * This method will be of return type boolean.
 */

```

```

public static boolean wordExists(BTreeNode T, String str){
    if(T.isLeaf){
        for(int i=0; i<T.n; i++){
            if(T.key[i].equalsIgnoreCase(str))
                return true;
        }
        return false;
    }
    for(int j=0; j<T.n; j++){
        if(T.key[j].equalsIgnoreCase(str))
            return wordExists(T, str);
        if(T.key[j].compareToIgnoreCase(str) < 0)
            return wordExists(T.c[j], str);
    }
    return (wordExists(T.c[T.n], str));
}

/**
 * ADDED METHOD mostAnagrams
 * This method will find the word in the tree with the most amount
 * of anagrams. From the main method, this method will take in a
 * null String and an int 0
 *
 */

public static BTreeNode mostAnagrams(BTreeNode T, String maxAnagrams, int
numberOfAnagramsFromBefore){
    if(T.isLeaf){
        for(int i=0; i<T.n; i++){
            int numOfAnagramsInThisWord = countAnagrams("",T.key[i],
T, 0);
            if(numOfAnagramsInThisWord >
numberOfAnagramsFromBefore){
                maxAnagrams = T.key[i];
                numberOfAnagramsFromBefore =
numOfAnagramsInThisWord;
            }
        }
        System.out.println("The word with the most amount of anagrams
is "+maxAnagrams+" with "+numberOfAnagramsFromBefore+" anagrams.");
    }
    else if(!T.isLeaf){
        for(int i=0; i<T.n; i++){
            int numOfAnagramsInThisWord = countAnagrams("",T.key[i],
T, 0);
            if(numOfAnagramsInThisWord >
numberOfAnagramsFromBefore){
                maxAnagrams = T.key[i];
                numberOfAnagramsFromBefore =
numOfAnagramsInThisWord;
            }
        }
        return mostAnagrams(T.c[0], maxAnagrams,
numberOfAnagramsFromBefore);
    }
}

```

```

        for(int i=0; i<T.n; i++){
            int numOfAnagramsInThisWord = countAnagrams("",T.key[i], T, 0);
            if(numOfAnagramsInThisWord > numberOfAnagramsFromBefore){
                maxAnagrams = T.key[i];
                numberOfAnagramsFromBefore = numOfAnagramsInThisWord;
            }
        }
        return mostAnagrams(T.c[T.n], maxAnagrams,
        numberOfAnagramsFromBefore);
    }

```

/\*\*

\* ADDED METHOD countAnagrams  
 \* This method will find the number of anagrams in a word  
 \* This method will be of int return type.  
 \*  
 \*/

```

    public static int countAnagrams(String prefix, String word, BTreeNode
    englishWordsSet, int count) {
        if(word.length()>8){
            String str = prefix + word;
            if (englishWordsSet.contains(englishWordsSet, str)){
                count++;
            }
        }
        else {
            for(int i = 0; i < word.length(); i++) {
                String cur = word.substring(i, i + 1);
                String before = word.substring(0, i); // letters before
                String after = word.substring(i + 1); // letters after

                if (!before.contains(cur)) // Check if permutations of
                cur have not been generated.
                    countAnagrams(prefix + cur, before + after,
                    englishWordsSet, count);
            }
        }
        return count;
    }

```

/\*

```

    public static void printAnagrams(String word) {
        printAnagrams("",word);
    }

```

```

    public static int countAnagrams(String word, BTree englishWordsSet){
        if (word.length()>8)
            return 1;
        return countAnagrams("",word,englishWordsSet);
    }

```



```

        */

        public static int countAnagrams(BTree englishWords, String prefix, String
word, int count) {
            if(word.length() <= 1){
                String str = prefix + word;
                if (englishWords.contains(englishWords, str)){
                    System.out.println(str);
                    count++;
                }
            }
            else {
                for(int i = 0; i < word.length(); i++) {
                    String cur = word.substring(i, i + 1);
                    String before = word.substring(0, i); // letters before
cur
                    String after = word.substring(i + 1); // letters after
cur
                    if (!before.contains(cur)) // Check if permutations of
cur have not been generated.
                        printAnagrams(englishWords, prefix + cur, before
+ after, count);
                }
            }
            return count;
        }
    }

```

```

        public static int printAnagrams(BTree englishWords, String prefix, String
word, int count) {
            if(word.length() <= 1) {
                String str = prefix + word;
                if (englishWords.contains(englishWords, str)){
                    System.out.println(str);
                    return count;
                }
            }
            else {
                for(int i = 0; i < word.length(); i++) {
                    String cur = word.substring(i, i + 1);
                    String before = word.substring(0, i); // letters before
cur
                    String after = word.substring(i + 1); // letters after
cur
                    if (!before.contains(cur)) // Check if permutations of
cur have not been generated.
                        printAnagrams(englishWords, prefix + cur, before
+ after, count);
                }
            }
            return count;
        }
    }
}

```

### 5.3— Student Code (*BTree.java*)

```
public class BTree{
    private BTreeNode root;
    private int t; //t is the maximum number of children a node can have
    private int height;

    public BTree(int t){
        root = new BTreeNode(t);
        this.t = t;
        height = 0;
    }

    public void printHeight(){
        System.out.println("Tree height is "+height);
    }

    public void insert(String newKey){
        if (root.isFull()){//Split root;
            split();
            height++;
        }
        root.insert(newKey);
    }

    public void print(){
        // Wrapper for node print method
        root.print();
    }

    public void printNodes(){
        // Wrapper for node print method
        root.printNodes();
    }

    public void split(){
        // Splits the root into three nodes.
        // The median element becomes the only element in the root
        // The left subtree contains the elements that are less than the median
        // The right subtree contains the elements that are larger than the median
        // The height of the tree is increased by one

        // System.out.println("Before splitting root");
        // root.printNodes(); // Code used for debugging
        BTreeNode leftChild = new BTreeNode(t);
        BTreeNode rightChild = new BTreeNode(t);
        leftChild.isLeaf = root.isLeaf;
        rightChild.isLeaf = root.isLeaf;
        leftChild.n = t-1;
        rightChild.n = t-1;
        int median = t-1;
        for (int i = 0; i<t-1; i++){
            leftChild.c[i] = root.c[i];
            leftChild.key[i] = root.key[i];
        }
    }
}
```

```

        leftChild.c[median]= root.c[median];
        for (int i = median+1;i<root.n;i++){
            rightChild.c[i-median-1] = root.c[i];
            rightChild.key[i-median-1] = root.key[i];
        }
        rightChild.c[median]=root.c[root.n];
        root.key[0]=root.key[median];
        root.n = 1;
        root.c[0]=leftChild;
        root.c[1]=rightChild;
        root.isLeaf = false;
        // System.out.println("After splitting root");
        // root.printNodes();
    }

    /**
     * ADDED Method .contains
     *
     * This method will determine if a string is detected in a key
     * This method will be of return type String
     */
    public boolean contains(BTree root, String str){
        return root.contains(root, str);
    }

    public String mostAnagrams(BTree root){
        return root.mostAnagrams(root);
    }
}

```

#### 5.4— Student Code (*BTreeNode.java*)

```

import java.lang.*;

public class BTreeNode{
    private int t;          // BTree parameter, each node has at least t-1 and at most
    2t-1 keys
    public int n;           // Actual number of keys on the node
    public boolean isLeaf;  // Boolean indicator
    public String[] key;    // Keys stored in the node. They are sorted in
    ascending order
    public BTreeNode[] c;  // Children of node. Keys in c[i] are less than key[i] (if
    it exists) and greater than key[i+1] if it exists

    public BTreeNode(int t){ // Build empty node
        this.t = t;
        isLeaf = true;
        key = new String[2*t-1]; // Array sizes are set to maximum possible value
        c = new BTreeNode[2*t];
        n=0; // Number of elements is zero, since node is empty
    }

    public boolean isFull(){
        return n==(2*t-1);
    }

    public void insert(String newKey){

```

```

        //System.out.println("inserting " + newKey); // Debugging code
        int i=n-1;
        if (isLeaf){
            while ((i>=0) && (newKey.compareToIgnoreCase(key[i]) < 0)){ // Shift key
greater than newKey to left
                key[i+1] = key[i];
                i--;
            }
            n++;
            key[i+1]=newKey; // Update number of keys in node
            // Insert new key
        }
        else{
            while ((i>=0)&& (newKey.compareToIgnoreCase(key[i]) < 0)) {
                i--;
            }
            int insertChild = i+1; // Subtree where new key must be inserted
            if (c[insertChild].isFull()){
                n++;
                c[n]=c[n-1];
                for(int j = n-1;j>insertChild;j--){
                    c[j] =c[j-1];
                    key[j] = key[j-1];
                }
                key[insertChild]= c[insertChild].key[t-1];
                c[insertChild].n = t-1;

                BTreeNode newNode = new BTreeNode(t);
                for(int k=0;k<t-1;k++){
                    newNode.c[k] = c[insertChild].c[k+t];
                    newNode.key[k] = c[insertChild].key[k+t];
                }

                newNode.c[t-1] = c[insertChild].c[2*t-1];
                newNode.n=t-1;
                newNode.isLeaf = c[insertChild].isLeaf;
                c[insertChild+1]=newNode;

                if (newKey.compareToIgnoreCase(key[insertChild]) < 0){
                    c[insertChild].insert(newKey);
                }
                else{
                    c[insertChild+1].insert(newKey);
                }
            }
            else
                c[insertChild].insert(newKey); //No need to split node
        }
    }

//Prints all keys in the tree in ascending order
public void print(){
    if (isLeaf){
        for(int i =0; i<n;i++)
            System.out.print(key[i]+" ");
        System.out.println();
    }
    else{
        for(int i =0; i<n;i++){
            c[i].print();
        }
    }
}

```

```

        System.out.print(key[i]+" ");
    }
    c[n].print();
}
}

public void printNodes(){
    //Prints all keys in the tree, node by node, using preorder
    //It also prints the indicator of whether a node is a leaf
    //Used mostly for debugging purposes
    for(int i =0; i<n;i++){
        System.out.println(key[i]+" ");
        System.out.println(isLeaf);
        if (!isLeaf){
            for(int i =0; i<=n;i++){
                c[i].printNodes();
            }
        }
    }
}

/**
 * ADDED Method 4
 *
 * This method will print the contents of all the nodes at
 * depth d or less.
 * This method will be of type NON-RETURN.
 */
public void printDepthNodes(BTreeNode T, int depth){
    if(depth==0){
        System.out.println();
        for(int i=0; i<= T.n; i++)
            System.out.print(T.key[i]+" ", );
    }
    else if(T.isLeaf)
        System.out.println("No keys in depth "+depth);
    else{
        for(int i=0; i<T.n; i++){
            System.out.println(T.key[i]+" ", );
            printDepthNodes(T.c[0], depth-1);
        }
    }
}

/**
 * ADDED Method 1
 *
 * This method will search the tree using sequential search
 * This method will be of return type BTreeNode, returning
 * the node at which the string is found.
 */
public BTreeNode sequentialSearch(BTreeNode T, String k){
    if(T.isLeaf){
        for(int i=0; i<T.n; i++){
            if(T.key[i].equalsIgnoreCase(k))
                return T;
        }
    }
}

```

```

        return null;
    }
    for(int j=0; j<T.n; j++){
        if(T.key[j].equalsIgnoreCase(k))
            return T;
        if(T.key[j].compareToIgnoreCase(k) < 0)
            return sequentialSearch(T.c[j], k);
    }
    return (sequentialSearch(T.c[T.n], k));
}

/**
 * ADDED Methods
 *
 * These methods will search the tree using binary search
 * This method will be of type type BTreeNode, returning
 * the node at which the string is found.
 */
public BTreeNode binarySearch(BTreeNode T, String k){
    BTreeNode found = binaryContains(T, k, 0, T.n); //last two: int starting
position, int ending position
    return found;
}
public BTreeNode binaryContains(BTreeNode T, String k, int start, int end){
    if(T.isLeaf){
        if(start<end){
            int middle = (end/2);
            if(k.compareToIgnoreCase(T.key[middle]) < 0)    //if the key is before
the middle
                return binaryContains(T, k, start, middle);
            if(k.compareToIgnoreCase(T.key[middle]) > 0)    //if the key is after
the middle
                return binaryContains(T, k, middle+1, end);
            if(k.compareToIgnoreCase(T.key[middle]) == 0)    //if the key is after
the middle
                return T;
        }

        if(start == end){
            if (k.compareToIgnoreCase(T.key[start]) ==
0)    //if the key is the middle
                return T;
        }

        if(!T.isLeaf){
            for(int i=0; i<T.n; i++){
                if(k.compareToIgnoreCase(T.key[i]) < 0)
                    return binaryContains(T.c[i], k, 0, T.n);
                return binaryContains(T.c[T.n], k, 0, T.n);
            }
        }
    }
    return null;
}

/**

```

```

* ADDED Method
*
* This method will search the tree and determine
* if what we're looking for is contained in the
* B-Tree.
* This method will be of a boolean return type.
*/
public boolean contains(BTreeNode root, String str){
    if(root.isLeaf){
        for(int i=0; i<root.n; i++){
            if(root.key[i].equalsIgnoreCase(str))
                return true;
        }
        return false;
    }
    for(int j=0; j<root.n; j++){
        if(root.key[j].equalsIgnoreCase(str))
            return true;
        if(root.key[j].compareToIgnoreCase(str) < 0)
            return contains(root.c[j], str);
    }
    return (contains(root.c[root.n], str));
}

/**
* ADDED Method
*
* This method will search the tree and find the
* word with the most amount of anagrams.
* This method will be of a String return type.
*/
public String mostAnagrams(BTreeNode root){
    if(root.isLeaf){

    }

    return "null";
}
}

```

