1 2-3-4 Trees

A 2–3–4 tree (also called a 2–4 tree) is a self-balancing data structure that is used to implement dictionaries to store <key, value> pairs. They can search, insert and delete in $O(\log n)$ time, if comparison between two keys takes constant time. Work your way through this assignment to learn how to implement a 2-3-4 tree and use it to solve an interesting problem!

1.1 Implementation

In this part, your task is to implement a `2-3-4 tree` data-structure using a class `Tree` defined in `Tree.java`. In this problem the keys will be strings and values will be positive integers. Example of <key, value> pairs: ("table", 5), ("the", 12).

The `Tree` class will comprise of the following components:

- **Instance variables:**
    1. `TreeNode root`: Points to the root of the `2-3-4 tree`, which will be set to null by the constructor, representing an empty `2-3-4 tree`.
- **Member functions:**
    1. `public void insert(String s)`: Takes a string `s` as a parameter and inserts the string at the leaf of the `2-3-4 tree`. Value of the string will be set to 1. We will ensure that the string `s` given as input always contains only lower case alphabets. Also, the insert function will be called only once for a particular string.
    2. `public boolean search(String s)`: Takes a string `s` as a parameter returns `true` if it is present in the `2-3-4 tree`, and `false` otherwise.
    3. `public boolean delete(String s)`: Takes a string `s` as a parameter and returns `false` if `s` is not present in the `2-3-4 tree`, otherwise removes the string `s` from the `2-3-4 tree` and returns `true`.

       (While deleting an internal string, the string chosen to perform swap should be the *predecessor of the string s*).
    4. `public int increment(String s)`: Takes a string `s` as a parameter and increments the value of string `s` by 1. The function returns the updated value. We will ensure that `increment(s)` is called only if `s` is present in the tree.
    5. `public int decrement(String s)`: Takes a string `s` as a parameter and decrements the value of string `s` by 1. The function returns the updated value. We will ensure that `decrement(s)` is called only if `s` is present in the tree and also that `value` of a node does not go below 1.
    6. `public int getHeight()`: Returns the current height of the (balanced) `2-3-4 Tree`.
    7. `public int getVal(String s)`: Returns the `value` of the string `s`.

Each node in the `Tree` will be an object of the class `TreeNode`. The class `TreeNode` comprises of the following:

- **Instance variables:**
  1. `ArrayList<String> s`: The list of strings stored at the node, initialised to empty array list by the constructor.
  2. `ArrayList<Integer> val`: The list of values of strings in the node containing, initialised to empty array list by the constructor.
  3. `ArrayList<TreeNode>` children: Pointers to children of the current node, initialised to empty array list by the constructor.
  4. `int height`: The height of the node, initialized as 1 by the constructor.
  5. `int count`: You will use this in part 1.2 .
  6. `String max_s`: You will use this in part 1.2 .
  7. `int max_value`: You will use this in part 1.2 .

NOTE: You will need to make sure that the tree is balanced at all times. So, perform the necessary operations while performing insertion and deletion. You will have to follow the standard conventions that at every node there can be 2,3 or 4 strings present. Some convention regarding borrowing while performing delete operation : If node if the leftmost node, then borrow from the just-right sibling. Otherwise every-time borrow from the just-left sibling.

```
Test case:
Tree obj = new Tree();
obj.insert("be");
obj.insert("ce");
obj.insert("de");
obj.insert("ae");

Current Tree :-
            be
          /    \
        ae     ce, de

obj.search("ce");    // returns true
obj.search("fe");    // returns false
obj.delete("ae");    // returns true

Current Tree :-
            ce
          /    \
        be      de

obj.getHeight();     // returns 2
obj.getVal("ce");    // returns 1
obj.increment("ce");
obj.getVal("ce");    // returns 2
```

You can create other helper functions if you like, but don't change the signature of the given functions. **You are not allowed to use any inbuilt/user-defined data structures for this part.**

1.2 Passage Processor using Augmented 2-3-4 Trees

In this part, you will learn a beautiful application of the `Tree` class you created.

Use your dictionary to read an English passage, and store words with corresponding frequencies. When you read a word, say "the", for the first time, you should insert it and set value (frequency) to 1. On each subsequent occurrence of "the", you should increment value by 1. We will use value and frequency interchangeably in this part of the question.

You also need to store some auxiliary information in variables `count` and `max` to answer queries of the following type in $O(\log n)$ time:

1. Given a pair of strings, say (`str1`, `str2`), report the number of strings (counted with multiplicity) that lie between `str1` and `str2` in the lexicographic ordering.

2. Given a pair of strings, say (`str1`, `str2`), report the string `s` lying between `str1` and `str2` in the lexicographic ordering that has maximum frequency.

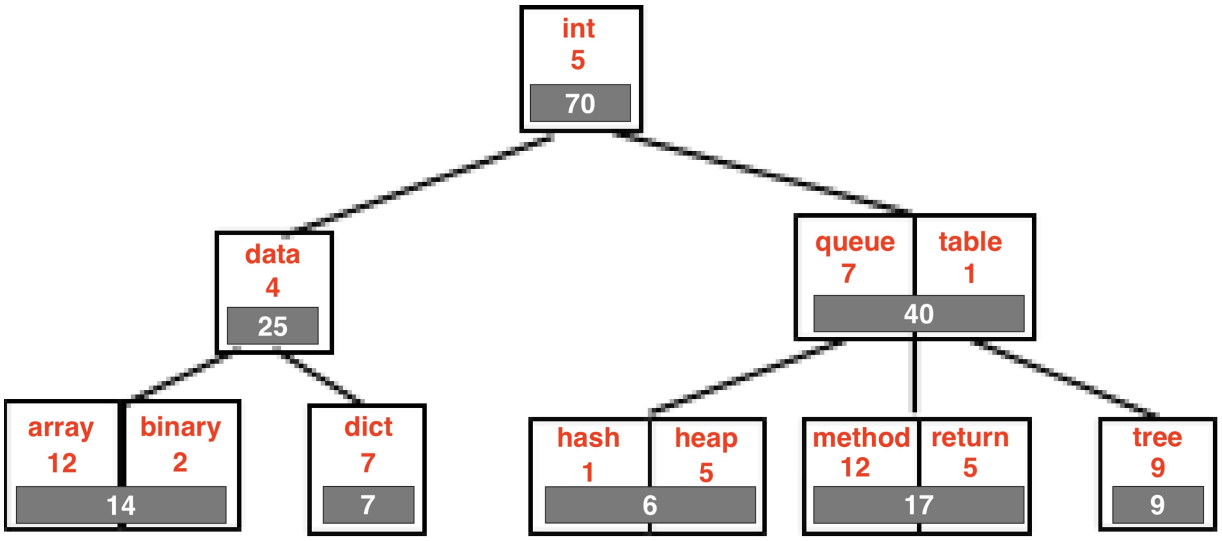

Figure 1: An example depicting a 2-3-4 tree. The fields highlighted in grey color provides a hint for the information that must be stored in the `count` variable.

Your task is to implement the following functions in Application.java file which extends `Tree` class :

1. `public void insert(String s)` : Takes a string `s` as a parameter and inserts string `s` at the leaf of the 2-3-4 tree. Value of the string must be set to 1. You will also need to handle the `count`, `max_s`, `max_value` variables.

2. `public int increment(String s)` : Takes a string `s` as a parameter and increments the `value` by 1. It should return the updated value. We will ensure that `increment(s)` is called only if `s` is present in the tree. You will also need to handle the `count`, `max_s`, `max_value` variables.

3. `public void buildTree(String fileName)` : Takes a string *fileName* as parameter which stores the path of the file which you need to read. You will need to build a `2-3-4 Tree` using the strings (space-separted) in the file. You need to use the insert and increment functions for this. This function will be called only once in the very beginning.

4. `public int cumulativeFreq(String s1, String s2)` : The function takes two strings `s1`, `s2`. It returns the total frequency of strings between `s1` and `s2` (inclusive). You need to use the count variable and perform the task in $O(\log n)$ time. We will make sure that `s1` is lexicographically smaller or same as `s2`.

5. `public String maxFreq(String s1, String s2)` : The function takes two strings `s1`, `s2`. It returns the string `s` lying between `s1` and `s2` (inclusive) in the lexicographic ordering that has maximum frequency. You need to use the `max_s` and `max_value` variables and perform the task in $O(\log n)$ time. If there are two or more strings with the maximum frequency, return the lexicographically smaller one. We will make sure that `s1` is lexicographically smaller or same as `s2`.

This question can be solved with various data structures. However, for an optimal solution, you require $O(\log n)$ time for search, insert and delete. You must use the 2-3-4 tree data structure from part 1 here. **You are not allowed to use any other inbuilt/user defined data structure for this question.**

NOTE: We will ensure that the file only contains spaces and strings which are lower case letters.

Example:

```
Test case:
Assume(hypothetically) that in your tree is such as the one given in the
   figure 1. Now, the following function calls are performed :

Inputs :
cumulativeFreq("dict","queue");          // returns 25
cumulativeFreq("array", "binary");       // returns 14
maxFreq("hash","return");                // returns "method"
maxFreq("array","binary");               // returns "array"


Explanation for cumulativeFreq:
Strings between dict and queue : dict, int, queue, hash, heap.
Adding the frequency of the strings gives answer as 7+5+6+7 = 25.
Note : You need to make use of "count" auxiliary variables so that you can
   do the computation in O(log n) time.

Explanation for maxFreq:
Strings between hash and return : hash, heap, queue, method, return.
String with the maximum frequency is : method.
Note : You need to make use of "max_s" and "max_value" variables so that
   you can do the computation in O(log n) time.
```