COP 5536 Project Report

Junrui Lin 99099174

1. Objective

   This project is to implement an event counter using AVL tree. For each node, it needs to records its ID and event count at least. ID is used to specific identification of one event and count is the number of active event, i.e. if it's smaller than 1 it will be removed.

   In this project, I use java to program my project. first I implement a AVL tree with its basic functions, and then make it accessible for other six functions: *increase, reduce, count, inRange, next* and *previous*.

2. Compiler

   For development, the environment was as follows:

   OS: Windows 8.1 professional

   Compiler: standard JDK

3. Structure

   Here I will list all of the function I used. And for complete documentation, please read the comments in the code.

   a. AvlTree.java

      This file contains several methods to build AVL tree and provide basic function and required functions to achieve sorts of purposes as required in this project.

      Public class: AvlTree. This class is to complete a AVL tree and test it.

      Following I will explain each functions and methods, for details, please check the comments in the corresponding codes.

      - Public class AvlNode (): this method is used to define nodes in AVL tree, it contains its basic value, i.e. *key* (means *ID*) and count. Also, it records its left node, right node and parent node as well as its balance factor.
      - Public void initial (int [], int [], int n): this method is able to do initialization of a AVL tree. The first array stores each *key* value, i.e. *ID*, and second array keeps corresponding *count* number, and *n* is the length of this two arrays which is equal to this first number of test txt file.
      - Public AvlNode initalTree (int [], int [], int start, int end): this method is actual initialization function. As described above, the first array stores each *key* value, i.e. *ID*, and second array keeps corresponding *count* number. And start and end are the first index and the last index of this two array respectively.
      - Public void insert (k, c): to call functions to insert a node with key *k* and count *c* to an existed tree.
      - Public void insertAVL (AvlNode p, AvlNode q): able to recursively insert a node into a tree. *P* is the node currently compared and usually we start with the root, q is the node with k and c needed to be inserted. After insertion, we need to check balance and decide whether we to do balancing.
      - Public void recursiveBalance (AvlNode cur): is used to check the balance for each node recursively, if it was not balanced, it would call corresponding method to do balancing, i.e. corresponding rotation.

- Public void remove (k): to remove a node with key *k*.
- Public void removeAVL (AvlNode p, int q): to search whether there is a node with key *q*. If so, call removeFoundNode function to delete it.
- Public void removeFounNode (AvlNode q): when we call this method, that means we have found the node we need to remove, so just remove it and then change other information, like which node should be put its place, which nodes should be left and right child, who's the parent. Besides, check balance and decide whether need to do balancing.
- Public rotateLeft: to do left rotation, that's RR rotation.
- Public rotateRight: to do right rotation, that's LL rotation.
- Public doubleRotateRightLeft: to do RL rotation.
- Public doubleRotateLeftRight: to do LR rotation.
- Public AvlNode search (k): this method is a search function, which is used to find whether there is a node with key *k* existing in the tree.
- Public void increase (i, j): it is able to increase the count number of the node with key *i* by j. Firstly to call search function, if we found the node then do increase, if not insert it.
- Public void reduce (i, j): it is used to decrease the count number of the node with key *i* by j. also, we need to call search function, if we found the node then do reduce. Furthermore, if its count number is smaller or equal to 0, delete this node, i.e. call remove function. If we cannot find the node, print *0*.
- Public void next (i): this method is able to print key and count of the node with lowest key that is greater than *i*. if there is no such node, print *0 0*.
- Private void searchNext: it's actual method to do next operation described above.
- Public void count (i): to print the count number of a node with key *i*.
- Public void previous (i): this method is able to print key and count of the node with greatest key that is less than *i*. if there is no such node, print *0 0*.
- Private void searchPrevious: it's actual method to do previous operation described above.
- Public void inRange (x, y): assuming $x \leq y$. It is a method to print all count numbers of nodes whose keys are between x and y inclusively.
- Private void searchRange: it's actual method to do inRange operation described above.
- Public void successor (AvlNode q): it is not the successor for inorder traversal. It is used in remove function for searching which node will be in the position where the node is removed.
- Private int height: it is able to calculate the height of a node. We will use it when we check balance.
- Private int maximum: it is a usual method to result in the maximum of two numbers.
- Private void setBalance: this method can calculate the balance factor of a node.

b. bbst.java

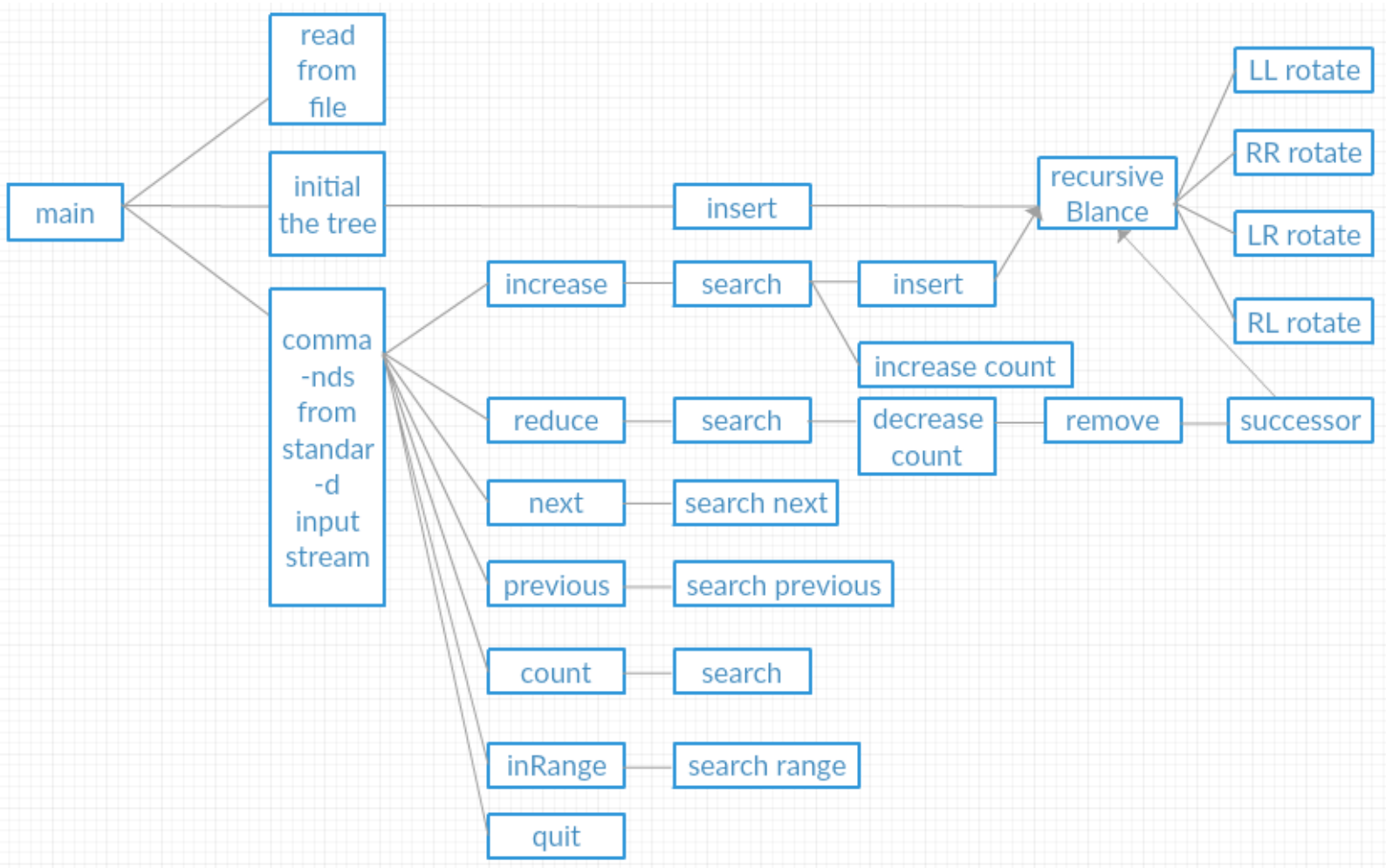
the main class contains the entry point and main application logic. It reads input from a file given as the command line argument, reads them into an array and then one by

one inserts them into a new built AVL tree.

After initialized insertion, the commands given from the standard input decides on which function is to be called. According to the following functions: *increase, reduce, count, inRange, next* and *previous*, the corresponding methods are called on the tree and the required output is shown as a standard output. Functions are explained in last part. The quit command simply exits from the program by calling exit (0), i.e. successful termination.

- scanner input {args [0]}: this method is able to read input file with event key and count through command line and initiate a AVL tree building process.
- Scanner command line: this method is used to do the same as "scanner input file". It would quickly read the commands line by line and execute them all and give output finally.

The diagram Following shows the basic structure of program and how the important functions are being used and where. The diagram is self-explanatory. Some getters/setter functions have been skipped, which were basic and obvious. All the important functions used and called by the program have been mentioned. The full documentation can be found in the code comments. The order to read process in this program is from left to right without arrows, if they are connected with an arrow, follow the arrow direction.



Correction: the method of "*insert*" after "*initial the tree*" is different from that in "*increase*", of course it actually means initial method, besides, I rewrite initial methods in *AvlTree*.*java* to be called by *main* method in *bbst*.*java* for better structure. So again, it is just a fundamental graph to illustrate basic idea for the programming.

4. Control Flow in the code

   When first input the event file using command line, all events ID and count will be accepted by *bbst*.*java* file. Then program will call *insert* () method to initiate the AVL tree.

   After initiating the AVL tree, program will accept the *command*.*txt* file, which contains multiple lines of functions regulated by requirements. According to those commands, program will automatically call *increase, reduce, count, inRange, next* and *previous* function, which have been described in above section.

   In the end, the program will make out a out.txt file that contains results of each command line. Then we can compare it with the standard answer and see it the answer is right or wrong.

5. Steps to execute the project in terminal

   I use Windows OS, so take command prompt for instance.

   a. Unzip the "*LIN_JUNRUI.zip*" file
   b. Copy the input files and commands files into the unzipped folder, assuming their names are "*test*.*txt*" and "*commands.txt*" respectively
   c. Go into terminal, use command "*cd*" to the file directory and execute "*make*" command;
   d. After you see several "*.class*" files come out, type command as following format to check result (there are a space between every words):

      java bbst test.txt <commands.txt> output.txt

   e. Then you can get a txt file named "*output*.*txt*" back as my result
   f. Now you can check whether it is same as your final result