2-3-4 Tree Class Implementation (Part 2)
Cpt S 223 Homework Assignment
Washington State University

Assignment Instructions (read all instructions *carefully* before you write any code):

This is the second part of a two-part assignment. In this assignment you will finish the 2-3-4 class implementation by adding removal/deletion functionality. The requirements from the previous assignment are listed here, with new requirements listed in red.

You will implement the 2-3-4 tree as a template class, but for this assignment you will always use **Student** as the template type. Implement a simple Student class or structure that consists of a string for a name and an integer for an ID number. As is mentioned a few more times later on in this assignment, the ID is what you use to compare one student structure to another.

You MUST use the preemptive split scheme that we discussed in class. You must also use the borrowing and fusion schemes described later on. Your output must EXACTLY match the expected output for the application. There is no variation expected across implementations if you correctly followed the preemptive split and specified borrowing/fusion rules.

Set up a main function to read commands from stdin:

In this assignment you will read simple commands from standard input. You will display your name and ID number on a single line to start with and then enter the command processing loop. There will be one command per line, so your main function (pseudo-code) will be something like:

```
cout << "Student Name, ID#\n";
while (true)
{
  string cmd = ReadLine();
  ProcessCmd(cmd);
}
cout << "Done\n";
```

Command strings will be much like C function calls in terms of syntax. You must support the following commands:

add(student name,student_id) : Adds a student structure to the tree. The name string may consist of any characters except for commas, parentheses, and non-displayable characters. The student ID is an integer that is used as the key for comparing one student structure to another as you build the tree.

- The is no space after the comma and before the student ID
- Do not display any output when processing this command
- Do NOT allow addition of two students with the same student ID
- DO allow addition of two students with the same name and different ID numbers

countLevels() : Displays an integer value indicating the number of levels in the tree. An empty tree has zero levels, a tree with only node, which implicitly is the root and the root is a leaf in this case, has 1 level, and so on.

displayInOrder() : Displays the contents of the entire tree, in order by student ID numbers. For each student the name is displayed, followed by a comma, then the student ID, and then a semicolon. There is a semicolon after every name,ID pair including the very last one on the line.

- If the student name contained spaces then those spaces will be displayed, but other than that there must not be any spaces on the line

getName(student_id) : Does a search within the tree for a student with the specified ID number and displays their name if found. If no student with the specified ID exists within the tree then then **(student not found)** is displayed.

getLevel(student_id) : Finds a node within the tree that contains a student with the specified ID number, then displays the level of the tree that this node is on. The root node is on level 0, its immediate children are on level 1 and so on. If the tree does not contain a student with the specified ID, then **-1** is displayed.

countValuesInNode(student_id) : Finds a node within the tree that contains a student with the specified ID number, then displays an integer value for the number of values that reside within that node. This will be **1**, **2**, or **3** if the node is found, or **-1** if the tree does not contain a student with the specified ID.

remove(student_id) : Removes the student structure with the specified ID from the tree. If no such student exists then no action is taken. This command does not display anything on screen. Removal scheme details are VERY important and you must follow them exactly. Your results will be exactly the same as the expected output if you follow these rules:

- Use the preemptive merge scheme we discussed in class. If you are traversing down the tree during a deletion/removal and you encounter a node with only 1 value in it, then you must borrow/fuse.
  - First check to see if you can borrow from the adjacent sibling on the left, and if so, do it.
  - If you cannot borrow from the adjacent left sibling, check the right and if you can borrow from that then do so.
  - If neither of the adjacent siblings have a value that you can borrow, fuse by bringing down a value from the parent.
    - If you're at child L or M1, then fuse by bringing down the minimum ('A' value) from the parent.
    - If you're at child M2 then fuse by bringing down the parent B value (will work in the case when the parent has 3 values as well as the case when the parent has 2 values).
    - If you're at child R then fuse by bringing down the maximum ('C' value) from the parent.
- Recall that when the node with the value to remove is found it could be either a leaf or internal node.
  - In the leaf node case, the value is just removed from the node.
  - In the internal node case take the minimum value in the right subtree, make a temporary copy of it, recursively delete it, and then swap it back in for the value that was originally to be deleted.

Remember that if you follow all these rules you should get an output that exactly matches the expected.

quit() : Breaks the command processing loop and terminates your program. Display the line “Done” as the very last thing before returning from main.

Notes:

- Use the sample input/output text files included in the zip for testing. You may want to develop your own additional test cases, as the included tests don’t necessarily exercise all of the relevant functionality.

- Put comments in your code! Give a brief explanation for each function at a minimum.
- Your code must compile with g++ on Linux and you must test it with ALL included input files on Linux as well. If your code crashes for any input file, do not submit it. Fix the crash, even if it means just displaying “not implemented” for a particular command.