Hiring Task

Deadline: Nov 6th, 2020 5:00 pm

This assignment assesses your knowledge in pointers, dynamic memory management, and classes (including deep copy).

The Problem

You will complete C++ code for a tree-based data structure for storing a collection of strings. We call the data structure "Pxtree" here. (Nope, that's not its real name, I made it up.)

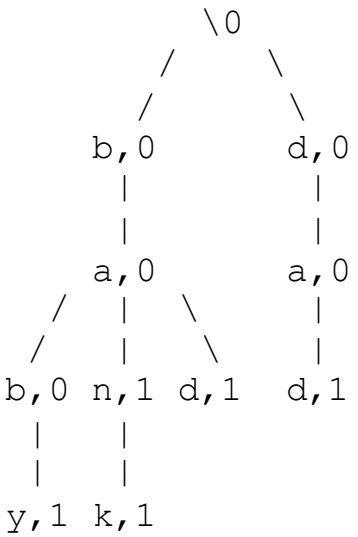
Graphical representation of the data structure

The data structure takes the form of a tree. Each node in the tree is associated with a single character (and an integer representing the counts, to be explained later). Each node can also have any number of children. The children of a node are kept in an order based on when they are inserted/removed (so not necessarily in alphabetical order).

Characters are stored in the nodes in such a way that the strings can be traced by following a path that starts from the root and choosing one of the children at each node. The characters at each node are concatenated to give the string. An alternative way of looking at this is that each string is represented by a path of individual letters, but strings that share some beginning letters (or *prefixes*) have those parts of their paths merged. As a result, no two children of a node should contain the same character.

The root node of the tree contains a special character '\0' that is not part of any string.

Here is an example of the tree containing the strings baby, ban, bank, bad, dad, inserted in this order:



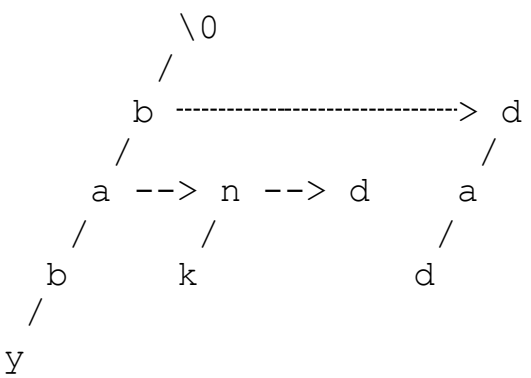
Note that a path that ends at an internal node (not a leaf) may or may not be a string that we want to represent. For example, how do we know that "ban" is a string to be stored, but "ba" is not? (Note that whether the word is a dictionary word is irrelevant here.) To distinguish this, each node is associated with an integer count, which is 0 if the string ending there is not actually represented, otherwise it is a positive number representing the number of occurrences of that string. (As explained below, the same string can be added multiple times.)

Left-child, right-sibling representation of the tree

There are many ways trees can be stored in a data structure. From an object-oriented point of view, I should not prescribe how you should implement the internal workings of the data structure as long as your implementation complies with the specified external interface; nevertheless, this is an assignment, so I am going to prescribe it.

Since we do not know how many children a node may have, we use a "leftmost child, right sibling" method which is a common way for representing trees. In this method, each node stores a pointer only

to its leftmost child (and not any other children). It also stores a pointer to its "next" sibling, i.e. the one to the right of it. The following figure shows how the same example above would be represented in this representation: (the integer counts are omitted in this picture)



This definition is specified in the Pxtree class:

```
class Pxtree {
  char  c_;
  int count_;
  Pxtree* leftmostChild_;
  Pxtree* nextSibling_;
};
```

Note that the child and the sibling, while being a node in the tree, can also be viewed as a Pxtree itself: a subtree formed by that node and all its (direct and indirect) descendents. This is reflected in the above class definition, representing them as pointer to Pxtree, instead of some kind of "Node" structure. There is no separate "Node" struct/class defined (and you should not do that); the Pxtree is defined recursively, in terms of itself.

One side effect of this recursive construction is that it is difficult to distinguish between an empty tree and a tree with one node (unless auxiliary information is stored.) Hence we assume a tree always contains at least one node (the root node with the special character '\0').

Supported operations

You should implement the Pxtree class that supports the following functions:

- Default constructor (that constructs an "empty" tree with the root containing '\0').
- Destructor, that cleans up all memory used.
- Copy constructor and copy assignment functions, that perform a deep copy of another existing Pxtree.
- `void add(string s)`: add the string s to the tree. If it is already in the tree, increase its count by 1. Any new node should be added as the rightmost child of its parent.
- `void remove(string s)`: decrease the count of the string s by 1 if it is in the tree. If this means the count becomes 0, the node is still left there, even if it is a leaf (this is dealt with separately in the `compact()` function later). If s is not in the tree, or if the count is already 0, make no changes to the tree.
- `int count(string s)`: return the count value associated with string s in the tree. If s is not in the tree, the count is 0. **This is done for you as an example.**
- `string print()`: returns a string that would print the tree, in a kind of "horizontal" way with the appropriate amount of indentation for each level of nodes (two extra spaces for each level down the tree). For example, the tree above will be printed like this:

```
b 0
  a 0
    b 0
      y 1
    n 1
      k 1
    d 1
d 0
  a 0
    d 1
```

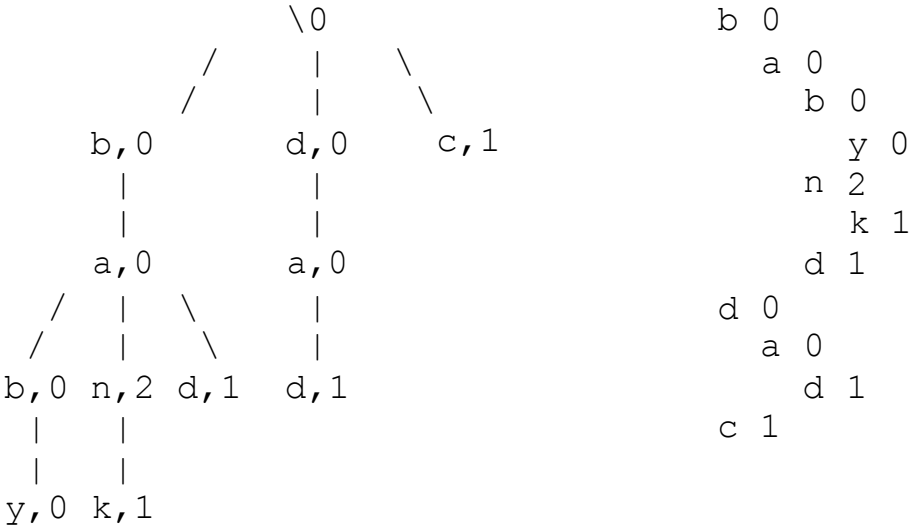
The "real" root node that contains \0 is not printed. See the "interactive" sample program (explained below) for more examples.

The whole printout is stored in one string (which may contain multiple lines, separated by the '\n' character) and returned to the caller; the function *itself* does not print anything to the screen.

- `string autoComplete(string s)`: based on the counts of the strings stored in the tree, return the string with the highest count that is an extension of s, i.e. a string that begins exactly as s but followed by some more letters. The string s itself is also a candidate (i.e., it is extended by 0 characters). If there are multiple such strings of equal highest count, you can return anyone. If no strings satisfy the criteria (i.e. s itself is not stored in the tree), return s itself. **This function is meant to be difficult.**
- `void compact()`: remove any leaves with a count of 0 from the structure of the tree. The memory of those nodes should be released correctly. Note that as a leaf is removed, some internal node with count 0 may now become a leaf, which has to be removed as well, which may lead to another internal node being exposed, and so on. Any nodes not removed should stay in the same order relative to their siblings. **This function is meant to be very difficult.** You may need to add more data members to the class and make many changes to other parts of the program.

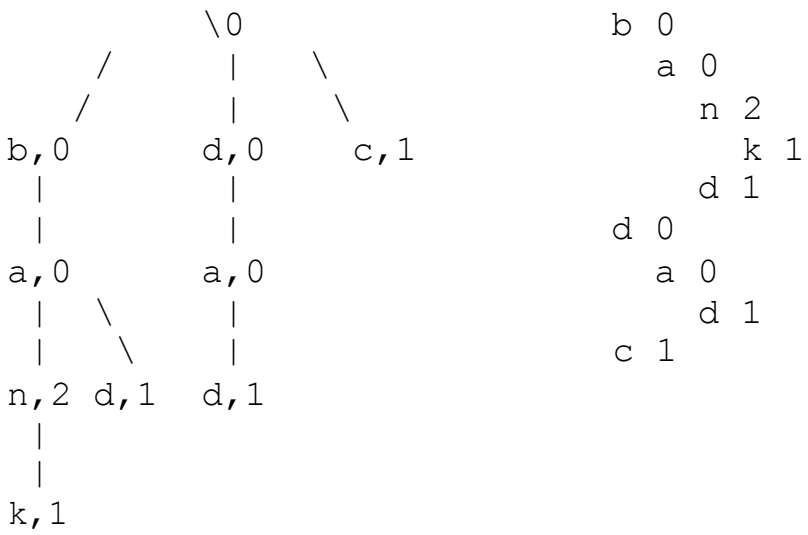
In all the above functions, you can assume the input string s is non-empty and only contains "normal", printable ASCII characters.

For illustration, this is what the tree will become if, starting from the above example, we called `add("ban"), add("c"), remove("baby")`. The left is the conceptual tree, the right is what `print()` should produce:



And at this point, the result of `autoComplete("ba")` would be "ban", `autoComplete("d")` would be "dad" and `autoComplete("z")` would be "z".

And finally if `compact()` is called at this point the result would be



Please refer to the Pxtree.h and Pxtree.cpp files for the exact specification of what each function should do.

Hint: recursion will be very useful or even a must in implementing most of these functions.

What can be changed

You must not change the existing public interface of the class, and you must not change the existing private data members of the class. You are allowed to add public or private member functions, or private data members, should you want to.

Files Provided

- Pxtree.h
- Pxtree.cpp

These are the only two files you need to modify/submit. All code written by you should be in there.

- main.cpp

This is just an example that illustrates how the functions can be called.

- PxtreeTester.h
- PxtreeTester.cpp
- PxtreeTesterMain.cpp

They are used for the execution testing part (see the next section).

- makefile

This is a makefile that will compile the main executable and the testing suite executable.

- InteractivePxtree

This program is given for your reference and convenience only. It is a Linux executable that shows you how the trees are supposed to look like (output of `print()`) after add, remove or compact operations. Just start the program and repeatedly type "add xxx", "remove xxx" or "compact" (where xxx is the string you want to add/remove). We will demonstrate its use in class.

Absolutely no error handling was attempted, so if you don't enter the commands exactly as stated, anything may happen. And even if you enter them correctly, it is not guaranteed to be bug-free and so, in case of disagreement with the assignment specifications, alert me about it!

It should work on the departmental linux system (and probably only on it; it probably won't work on your home linux and certainly not in Windows!) You may have to type "chmod u+x InteractivePxtree" to make it executable before running it. (If there is enough demand I may try to make a Windows version...)

To use the test suite (which is used in the execution testing part of the marking), simply type "make" in a linux terminal (with all the above files in the same folder). It will (assuming you did not break Pxtree.h or Pxtree.cpp) produce a PxtreeTester executable file. Run the program by typing

```
./PxtreeTester a
```
or
```
./PxtreeTester a b c d
```
which runs a single test case or multiple test cases respectively. We will also demonstrate its use in class.

The test suite may not cover all corner cases, so passing all test cases does not guarantee your program is 100% correct. Also, unfortunately we cannot isolate the testing of the various functions. For example to pass those test cases meant for `print()` you need to also implement `add()` at least partially correctly.

Submission Instructions

Submit only the files Pxtree.h and Pxtree.cpp. Just upload them as two separate files. DO NOT put them into a zip archive, and DO NOT upload the entire project folder of whatever IDE you are using.

While you may want to change the main.cpp file for your own testing, it is not part of the submission. The test suites and the makefiles also should not be submitted. If you submit them, they will be ignored and I will use my own version for testing.