CS360 -- Lab 1

- Jian Huang
- CS360
- Url: http://www.cs.utk.edu/~huangj/cs360/360/labs/lab1/lab.html

---

This is a lab that makes sure that you have red-black trees, dllists and fields down. **If you are not yet familiar with Dr. Plank's fdr library (or need to brush up on it!), read Dr. Plank's lecture notes about libfdr.**

Your job in this lab is to write the program **famtree**. **Famtree** takes a description of people and their relationships to one another on standard input. These descriptions must be in a special format, which will be described below. An example is in the file **fam1**, which represents a family composed of Bob and Nancy, and their three chidren Frank, Lester and Diana.

Famtree takes such a description, and prints out complete information on all of the people in the file. This consist of, for each person, their sex (if known), their father, mother and children. Therefore **fam1output** contains valid output of **famtree** on **fam1**.

The format of the input file is as follows. Lines must either be blank, or have one of following first words:

- **PERSON**: This specifies a person. Following **PERSON** is the person's name, which may be any number of words. **Famtree** should assume that names consist of words with single spaces between them.
- **SEX**: This specifies the person's sex. It can be either **M** or **F**.
- **FATHER**: This specifies the person's father. Following **FATHER** is the father's name, which may be any number of words. It implies that the father is male. A person may only have one father.
- **MOTHER**: This specifies the person's mother. Following **MOTHER** is the mother's name, which may be any number of words. It implies that the mother is female. A person may only have one mother.
- **FATHER_OF**: This specifies that the person is male, and that the specified person is one of the person's children. A person may have any number of children.
- **MOTHER_OF**: This specifies that the person is female, and that the specified person is one of the person's children.

Famtree has two other features. First, it prints out the people in a structured order. That is, no person can be printed out until both of their parents have been printed out. If this is impossible (as, for example, in **cyclefam**), then **famtree** identifies this fact.

The second feature of **famtree** is that it allows for redundancy, but it infers as much as it can. For example, **redundant** has a few lines that are redundant. For example, line 3 is redundant because Fred must be male by virtue of being Joe's father. Moreover, line 7 is redundant because line 2 already specified Fred as Joe's father. The file **nonredundant** is the minimal file specifying the same family.

Working example

The class directory of lab1 is **~cosc360/lab1/**. Get everything you need from that directory. There is also a working example of **famtree**. Try it out on the input files in this directory. Other input files are:

- **fam2** a more complex family.
- **fam3** A family with 10 generations of single parents.
- **fam4** A file with an error in it.
- **fam5** A file with another error in it.

You should make your output work exactly like **famtree**'s.

Help

I'm going to give rather detailed help here. In fact, I basically "give away" how to do the lab. This is so that you understand the correct structure of the program. If you think you can do this without any help, and don't want the answer given away, please don't read this, and have fun! I'd recommend that you at least think about how you'd do it with no help before you read further. However, if you want some help, read on.

You will have a **struct** for a person (mine is called a **Person**). That struct will have the following fields:

- Name
- Sex
- Father
- Mother
- List of children
- Some stuff for depth and breadth first traversal.

Your program will work in three phases:

1. **Reading the information into the structs.** You should have a red-black tree (mine is called **people**) that contains all the people. It is keyed on each person's name, and the **val** fields are the persons' **Person** structs. You use an **IS** to read standard input.

   Each time you read a line that has a name (i.e. **PERSON**, **FATHER**, **MOTHER**, **FATHER_OF** and **MOTHER_OF**) you test to see if the person with that name is in the **people** tree. If not, you create the struct and insert it.

   Whenever you process a line that needs to create some links (i.e. **FATHER**, **MOTHER**, **FATHER_OF** and **MOTHER_OF**), you first check to see if the link exists and is correct. If incorrect, you flag an error. If correct, you do nothing. If the link doesn't exist, you should create it in both the parent and the child.

   When you're done with this phase, you'll have a red-black tree with all people in it, and all the people will have the correct links to their parents and children.

2. **Testing the graph for cycles.** The graph is messed up if there is a cycle in it. In other words, if a person can be his/her own ancestor or descendant, than there is a problem. Testing for cycles is a simple depth-first search, which you should have learned in CS302. To test if a person is his/her own descendant, the following pseudocode will work:

   ```
   /* assume that there is an integer field called "visited"
      in the Person struct, and that this field is initialized
      to zero for all people */

   is_descendant(Person *p)
   {
     if (p->visited == 1) return 0;  /* I.e. we've processed this
                                        person before and he/she's ok */
     if (p->visited == 2) return 1;  /* I.e. the graph is messed up */
     p->visited = 2;
     for all children of p do {
       if is_descendant(child) return 1;
     }
     p->visited = 1;
     return 0;
   }
   ```

3. **Printing out the graph.** If you want to forego the final 20% of the lab, just traverse the **people** tree and print out each person. Otherwise, this is a kind of breadth-first search. What you do is create a queue (which will be a **Dllist** called **toprint**. You can initially put all people into **toprint**, or you can just put all people who have no parents.

   Then you execute a loop that does the following:

```
/* assume that there is an integer field called "printed"
   in the Person struct, and that this field is initialized
   to zero for all people */

while toprint is not empty
  take p off the head of toprint
  if p has not been printed, then
    if p doesn't have parents, or if p's parents have been printed then
      print p
      for all of p's children, put the child at the end of doprint
    end if
  end if
end while
```

Enjoy!