CPSC 221: Project `proj2` due Sunday, June 18, 2017 at 11pm via `handin`

Out: June 9, 2017 Last Updated: June 9, 2017

Mimicking Book

The goal of this assignment is to create a program `mimic` that creates a model of some sample text and then produces random text according to that model. The input to `mimic` is a non-negative integer n and a text file *book* containing some sample text. The output of `mimic` is:

If $n = 0$, the model. Write m lines to the standard output (using `cout`), where m is the number of different words in *book*. The rth line of output (for $0 \leq r \leq m - 1$) is the rth new word in *book* (we say the word has *rank* r) followed by a list of all the words that immediately follow this word in the text (we call these the word's *followers*), in the order in which they occur, with duplicates.

For example, the text:

```
The rose is a rose,
And was always a rose.
But the theory now goes
That the apple's a rose,
And the pear is, and so's
The plum, I suppose.
The dear only knows
What will next prove a rose.
You, of course, are a rose -
But were always a rose.
```

produces:

```
the : rose theory apple's pear plum dear
rose : is and but and you but
is : a and
a : rose rose rose rose rose rose
and : was the so's
was : always
always : a a
but : the were
theory : now
now : goes
goes : that
that : the
apple's : a
pear : is
so's : the
plum : i
i : suppose
suppose : the
dear : only
only : knows
knows : what
what : will
will : next
next : prove
prove : a
you : of
of : course
course : are
are : a
were : always
```

Make your model output look exactly like this sample. I should be able to type "`mimic 0 book.txt`" at a Unix prompt to produce the model for `book.txt`.

If $n > 0$, a random book. Write to standard output a sequence of n words generated using the model created from the sample text *book*. Separate words by a single space or newline.

To generate the sequence, first generate a word from *book* at random, where the probability of producing the word w_1 equals the number of occurrences of w_1 in *book* divided by the total number of words in *book*. Generate the next word w_{i+1} from the current word w_i by choosing w_{i+1} at random from the multi-set of words that immediately follow w_i in *book*. (So, the probability of producing w_{i+1} equals the number of occurrences of w_i w_{i+1} in *book* divided by the number of followers (with duplicates) of w_i.)

For the example above, if w_1 is "`rose`" then w_2 will be "`is`" with probability 1/6, "`and`" with probability 2/6, "`but`" with probability 2/6, and "`you`" with probability 1/6. Note that "`rose`" occurs 7 times in *book* but has only 6 followers because "`rose`" is the last word of *book*.

If w_i has no followers (and $i < n$) then generate w_{i+1} in the same manner as w_1.

I should be able to type "`mimic 100 book.txt`" at a Unix prompt to produce 100 words generated using `book.txt` as a model.

Treaps

Treaps are special type of binary search trees, sometimes called *randomized binary search trees* that combine features of binary search trees and heaps (which explains their name "treaps"). Treaps provide an implementation for dictionary ADT that you will need to implement the `mimic` program. Each node of a treap in addition to the usual data fields: key, value, and pointers to the left and right child, stores a *priority* that is randomly assigned when the node is created. A **treap** is a binary search tree with the property that the node priorities satisfy heap order property: *Any node's priority is greater or equal to the priority of its parent.*

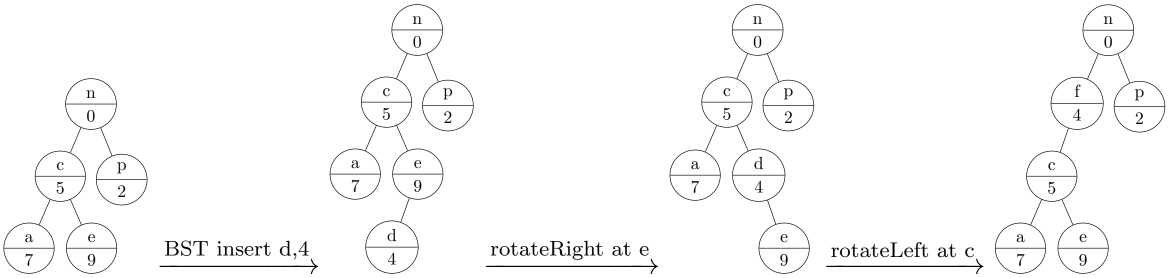
Implementing dictionary operations

- `find(key)`: works in the same way as find in a binary search tree (since a treap is a binary search tree).
- `insert(key,value)`: First insert new node as a leaf as in a BST. The new node get a random priority, and therefore, the heap property between the new node and its parent could be broken. To fix it, we need something like `swapUp`, that will bubble up the node with a small priority up the tree. The usual swapping is not going to work, because that would immediately break the binary search tree property. However, we have operations (used for AVL trees) that

  - do not break the binary search property, and
  - invert parent-child relationship between two nodes.

  Those operations are `rotateLeft` and `rotateRight`. After the rotation, the new node moves up in the tree and the heap order property between this node and its new parent might again broken. Repeatedly rotating at its parent, will move this node up, until the heap order property is not violated. At that point we are done with insertion.

  Here is an example of the treap insertion operation (the upper part of each circle represent the key, the lower part the priority):



- `delete(key)` is not needed for this project. But if you are curious how it works, here it is. First, you use `find` to find the node you want to delete. Then you change the priority of this node to infinity (or something big). This will break the heap property, so you need a `swapDown`-like function, that uses rotations to move the node down until it restores the heap property. The node to be deleted is now a leaf, so it can be easily removed from its parent without breaking anything.

Note about running times: the worst-case performance of all three dictionary operations is in $O(n)$, however, the expected running time of all three operations is $\Theta(\log n)$.

Your first task

Implement data structure with keys of type `string` and values of type `int`. Use `rand()` function to generate random priorities of new nodes. You only need to implement functions `find` and `insert`. Template for your implementation is provided in file `treap.h`.

Overview of implementation of mimic

What follows is an outline of my program. Your program can be different, but you must use a treap to look up words. The keys in the treap are words and values are their ranks. In addition you can use `vector` to store additional information (`vector` is like an array, but it's resizable). In the following I am using these vectors to build a model of the input text:

```
vector<string> word; // word[r] is the r-th word
vector<int> all_word_ranks; // ranks of all words in the text
vector<vector<int> > follower_ranks;
// follower_ranks[r] is a vector of ranks of followers of word[r]
```

Create the model

1. Read the input file one line at a time and break it into words as in the previous assignment.
2. For each word w in the line:
   (a) Look up (the key) w in a treap to get its rank (the value) r.
   (b) If w is not found, this is its first occurrence so assign it a new rank r (it is the rth new word) and store the key, value pair (w, r) in the treap. Also set `word[r]` $= w$.
   (c) Add r to the previous word's vector of followers.
   (d) Add r to the vector of *all* word ranks.

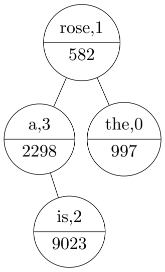
3. If $n = 0$ output the model and return.

Example: Content of all data structures for the text

```
The rose is a rose is
```

is:

```
    word = {"the", "rose", "is", "a"}
    all_word_ranks = {0,1,2,3,1,2}
    follower_ranks[0]= {1}
    follower_ranks[1]= {2,2}
    follower_ranks[2]= {3}
    follower_ranks[3]= {1}
```



Note that the exact shape of the treap depends on the randomly generated priorities! The treap on the right above is just one of many (in fact, $4! = 24$) possible shapes of this treap.

Generate text Note that the treap is not needed to generate text.

4. Set `A` to be the vector of *all* word ranks.

5. Repeat n times:

   (a) Choose a random index `i` into `A`. (`i = rand() % A.size()`)

   (b) Output `word[A[i]]`.

   (c) Set `A` to be the vector `followers[A[i]]` (or set `A` to be the vector of *all* word ranks if `word[A[i]]` has no followers).

Comment your code so that someone else (as well as you) can understand it.

Provided Code

The following files are provided:

- `treap.h` contains the headers of functions of the treap ADT and definition of `TreapNode`;
- `mimic.cc`: the main file, where you should implement the UPGMA algorithm;
- `book.txt`: the sample input.

In addition, `mimic.cc` contains code that will help you read the parameter and read the input file specified as a parameter.

Deliverables

Using `handin proj2`, you should submit:

- Your source code (.cc and .h files).
- A `Makefile` so that typing `make` in your handin directory on an undergrad Unix server will produce an executable called `upgma`.
- A `README` file containing:
  1. Your name or, if a team submission, both your names.
  2. Approximately how long the project took you to complete.
  3. Acknowledgment of any assistance you received from anyone but your team members, the 221 staff, or the 221 textbooks, but please cite code quoted or adapted directly from the texts (per the course's Collaboration policy).
  4. A list of the files in your submission with a brief description of each file.
  5. Any special instructions for the marker.
- DO NOT HAND IN: .o files, executables, core dumps, irrelevant stuff.

How to `handin` this assignment

1. Create a directory called `~/cs221/proj2` (i.e., create directory `cs221` in your home directory, and then create a subdirectory within `cs221` called `proj2`).

2. Move or copy all of the files that you wish to hand in, to the `proj2` directory that you created in Step 1.

3. Before the deadline, hand in your directory electronically, as follows: `handin cs221 proj2`

   Note that you will receive a set of confirmation messages. If you don't get any kind of an acknowledgment, then something went wrong. Please re-read the instructions and try again.

4. You can overwrite an earlier submission by including the `-o` flag, and re-submitting, as follows: `handin -o cs221 proj2`

   You can hand in your files electronically as many times as you want, up to the deadline.

5. Additional instructions about `handin`, if you need them, are listed in the man pages (type: `man handin`). At any time, you can see what files you have already handed in (and their sizes) by typing the command: `handin -c cs221 proj2`

   If your files have zero bytes, then something went wrong and you should run the original `handin` command again.