# CPSC 221: Project `proj1` due Friday, June 2, 2017 at 11pm via `handin`

Out: May 25, 2017 Last Updated: May 26, 2017

## UPGMA

**UPGMA** (Unweighted Pair Group Method with Arithmetic Mean) is a simple bottom-up hierarchical clustering method used to construct a phylogenetic tree (a tree of life) from the pairwise (evolutionary) distances between species invented by Sokal and Michener in 1958. The method works by clustering the species, at each stage joining two clusters together that have the smallest distance.

The input to the algorithm is a sequence of strings with names of the species and distances between the species represented by $n$ rows each containing $n$ numbers, where the $j$-th decimal number in a row $i$ represents the distance between the $i$-th species and the $j$-th species. You can assume that the distance between the $i$-th species and the $j$-species in the input is the same as the distances between the $j$-th species and the $i$-th species.

Here is an example of the input:

```
human mouse zebrafish C.elegans chicken
0   1.7 2.1 3.1 2.3
1.7 0   3   3.4 3.1
2.1 3   0   2.8 3.9
3.1 3.4 2.8 0   4.3
2.3 2.1 3.9 4.3 0
```

Your program should read this input from a file specified as a parameter to the program. For example,

```
./upgma species.txt
```

where `species.txt` file contains the lines described above.

### Clustering procedure

- **Initialization:** Assign each species to its own cluster. The distance $d_{C,D}$ between clusters $C = \{c\}$ and $D = \{d\}$ is initialized to the distance between species $c$ and $d$. The name of cluster $C = \{c\}$ is the same as the name of species $c$.

- **Iteration:** Find two clusters with the minimal distance, say $C$ and $D$. Add a new cluster $E$ that is a joint of clusters $C$ and $D$, and remove the original two clusters $C$ and $D$. The name of the new cluster is set to `(nameC,nameD)`, where `nameC` and `nameD` are the names of clusters $C$ and $D$. For example, if cluster $C$ has name `human` and cluster $D$ has name `mouse` then cluster $E$ will have name `(human,mouse)`. If $E$ is then joined with a cluster with name `chicken`, the new cluster will have name `((human,mouse),chicken)`.

  The distance between the new cluster $E$ and any other (current) cluster $X$ is calculated using the following formula:

$$d_{E,X} = \frac{|C| \cdot d_{C,X} + |D| \cdot d_{D,X}}{|C| + |D|}$$

where $|C|$ is the number of species cluster $C$ contains and $|D|$ is the number of species in $D$. For example cluster `human` contains one species, `(human,mouse)` two species and `((human,mouse),chicken)` contains three species.

- **Termination:** When only one cluster remains, stop.

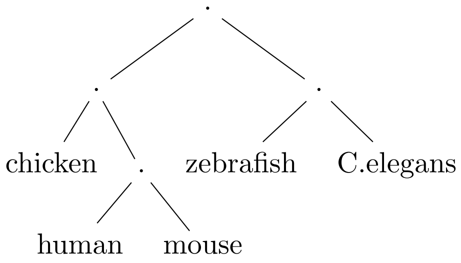
You can see a detailed work of the algorithm on a similar input here:

`wiki`https://en.wikipedia.org/wiki/UPGMA.

The output produced by the UPGMA algorithm is the name of the last cluster. For the input example above it should be the string:

```
((chicken,(human,mouse)),(zebrafish,C.elegans))
```

which your program should be print to the screen. Note that this is a string representation of the following tree:



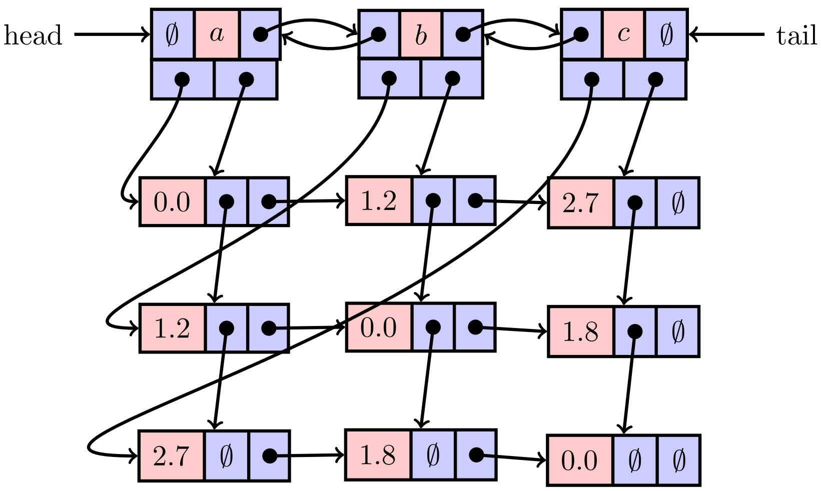
**Bonus.** You can get a small bonus if you also include a (recursive) procedure that takes the string representation of the phylogenetic tree as the input and outputs a nice tree representation to screen (it can be based on the code from Lab 4, but it should look better than formats from the lab). The **first line** printed by your program should still be the string representation of the tree. You can print a nice representation (if you wish to receive a small bonus) in the subsequent lines.

## Representation of distances

To store distances between clusters we will use **Dynamic Matrix** data structure (that's a new data structure invented for this assignment only, so I do not recommend googling for it, as probably things you will find are not related) that allows to remove/add rows/columns from the matrix[1].

Let's start with an illustration of this data structure that stores three clusters named $a, b, c$ and the corresponding distances between them:

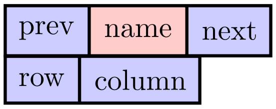
[1] Do not worry, we only need matrix to store 2D distance data, we are not going to do any matrix operations (like multiplication) with the matrix.



For example, the distance between cluster $b$ and $c$ can be found in the second column and the last row (or alternatively, in the last column and the second last row): 1.8.

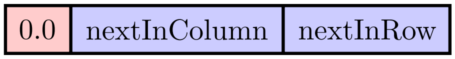
There are two types of nodes: `ClusterNode` and `DistanceNode`. The `ClusterNode` is defined as follows:

```
struct ClusterNode {
  string name;
  ClusterNode* prev;
  ClusterNode* next;
  DistanceNode* row;
  DistanceNode* column;
};
```

Here is a diagram of `ClusterNode`: 

The `DistanceNode` is defined as follows:

```
struct DistanceNode {
  double distance;
  DistanceNode* nextInColumn;
  DistanceNode* nextInRow;
};
```

Here is a diagram of `DistanceNode`: 

One way how to look at this structure is the following: we have a doubly-linked list of clusters. Each cluster has pointers to two linked list: one for the corresponding row in the distance matrix and one for the corresponding column in the distance matrix. Hence, each `DistanceNode` is included in two linked list (one linked through `nextInRow` and other through `nextInColumn`).

You will first need to implement the following operations on Dynamic Matrix data structure:

```
void addCluster(ClusterNode *&head,ClusterNode *&tail,const std::string& name);
// adds a cluster (at the tail) and the corresponding row and column to data structure
// distance of all added DistanceNodes should be initialized to 0.0
// at the end, tail should point to the newly added ClusterNode

void removeCluster(ClusterNode *&head,ClusterNode *&tail,ClusterNode *toBeRemoved);
// removes a cluster pointed to by toBeRemoved and the corresponding row and column
// if toBeRemoved is the first or last cluster then head or tail needs to be updated

void findMinimum(ClusterNode *head,ClusterNode *&C,ClusterNode *&D);
// finds the minimum distance (between two different clusters) in the data structure
// and returns the two clusters via C and D
```

Then you should use these functions to implement the UPGMA algorithm.

### Provided Code

The following files are provided:

- `dynmatrix.h` contains the headers of functions described above (feel free to add additional functions you might need, for example `printDynMatrix(ClusterNode *tail)` might be useful for debugging);
- `dynmatrix.cc` contains these functions' bodies, but you need to provide the actual code for the bodies;
- `main.cc`: the main file, where you should implement the UPGMA algorithm;
- `Makefile` that can compile the project assuming you don't add more files (which you are free to do so, but you will need to update `Makefile` if you do);
- `species.txt`: the sample input.

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

## Deliverables

Using `handin proj1`, 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/proj1` (i.e., create directory `cs221` in your home directory, and then create a subdirectory within `cs221` called `proj1`).

2. Move or copy all of the files that you wish to hand in, to the `proj1` directory that you created in Step 1.

3. Before the deadline, hand in your directory electronically, as follows: `handin cs221 proj1`

   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 proj1`

   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 proj1`

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