CS 32 Winter ‘19: Project 4

What to do?

- Implement 3 classes:
  - Trie
  - Genome
  - GenomeMatcher

Class #1: Trie

- Implements a multimap abstract data type (ADT), using a trie data structure
- Trie class **must** be implemented using a trie for full credit!
- Your trie should be implemented as if you're using this multimap implementation:

```
std::multimap<string, int> myMap;                // maps strings to ints
```

- However, we always assume that the key is always a string and you can use the Trie class as such:

```
Trie<int> trie1;                                 // maps strings to ints

Trie<char> trie2;                               // maps strings to chars
```

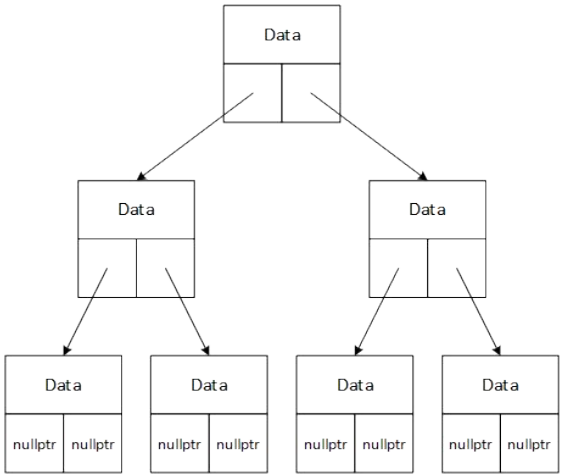
Class #1: Trie Methods

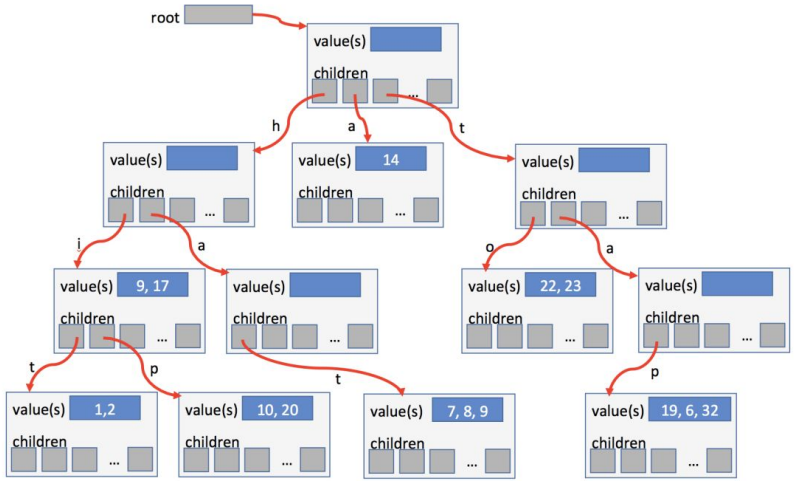
- Some functions you must write:
  - `void insert(const string& key, const ValueType& value)`
    - Insert the key and its values by adding nodes
    - Must run in O(L*C) time, with L = length of key and C = ave. number of children
    - Hint: Do NOT deal with mismatches here, do it in the find function
  - `vector<ValueType> find(const string& key, bool exactMatchOnly)const`
    - Search for values associated with the given key string
    - Return vector containing all the associated values, if no match return an empty vector
    - exactMatchOnly is set to true if we only want **exact match** or **SNiPs** only

```
std::vector<int> result1 = trie.find("hit", true);
// returns {1, 2} or {2, 1}
```

Class #1: Trie

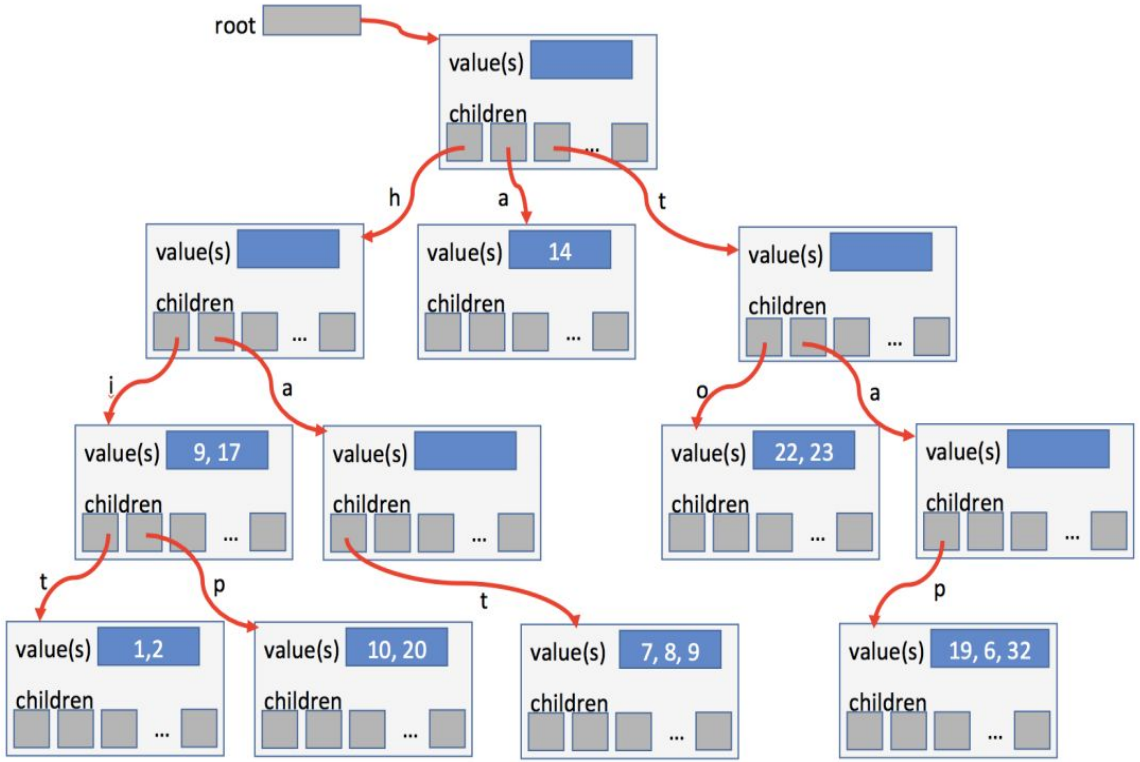
- Tree vs. Trie:
  - Trie: a node can have multiple values and many children (a regular tree only contain a value and a pointer to each child)
  - Tree: a node can only have one value and a pointer to one child





Class #1: Trie

- Search for:
  - “hip” : {10, 20}
  - “tap” : {19, 6, 32}
  - “hat” : {7, 8, 9}
  - “hi” : {9,17}



Class #1: Trie

- How to implement Trie for this project?
  - We know that there are only 4 DNA bases: A, C, T, G
  - Each DNA patterns start with these 4 bases followed by one of the 4 bases then another one of the 4 bases, etc.

    ACTAAGTAGGATGA….

    CTGAATGGACTGAA….

    TTAAGCTAGGCTAC….

  - How to implement that using a Trie?

Class #2: Genome

- This class holds data of an organism’s complete genome and allows you to extract DNA **subsequences** from different positions of the genome
- Restrictions:
  - Do not add public member functions or data members
  - If you need to add more functions/ members, add it in your private method

Class #2: Genome Methods

- `static bool load(std::istream& genomeSource, vector<Genome>& genomes)`
  - Should load a text file in FASTA format (read more in specs pp. 17-18)
  - Must run in O(N) time
  - How the load function works:

1. Extract genome name from a line in the file that begins with > sign

2. Extract the sequence of DNA bases and turn it into one string

3. Create a *Genome* object with the name of the organism and its DNA sequence. Add it to the *genomes* vector that is passed

- `int length() const`
  - Returns length of the DNA sequence, running in O(1) time
  - Use string.length() method → it is of O(1) complexity
- `string name() const`
  - Returns the name of the genome in O(S) time
  - Name of genome can be seen in the FASTA file

Class #2: Genome Methods

- `bool extract(int position, int length, string& fragment) const`
  - Finds a substring of size *length* starting at *position* and sets it to *fragment*
  - Returns true if successfully extract the string, false otherwise (i.e. goes beyond the length of the genome, or position is more than the size)

```
Genome g("oryx",
"GCTCGGNACACATCCGCCGCGGACGGGACGGGATTCGGGCTGTCGATTGTCTCACAGATCGTCGACGTACATGACTGGGA");

string f1, f2, f3;

bool result1 = g.extract(0, 5, f1);     //   result1 =  true, f1 = “GCTCG”;

bool   result2 = g.extract(74, 6, f2); //    result2 =  true, f2 = “CTGGGA”;

bool result3 =g.extract(74,      7, f3); // result3 =  false, f3 is unchanged
```

Class #3: GenomeMatcher

- Maintains the library of genomes and allows to search through the genomes for DNA sequences
- Can identify genomes in the library that are related to the queried genome
- Implementation **must** use the Trie class template
- May use other containers as long as they are not used to hold DNA sequence data
- Can use functions from the `<algorithm>` library

Class #2: GenomeMatcher Methods

- `void addGenome(const Genome& genome)`
  - Add new genome to the library of genomes maintained by GenomeMatcher object
  - Must do two things:
    i. Add the genome to a collection of genome (hint: use a vector) held by GenomeMatcher object
    ii. Index the DNA sequences of newly-added genome by adding *every* substring of length minSearchLength of that genome's DNA sequence into a Trie maintained by GenomeMatcher
  - Must run in O(L*N) time, where L= minSearchLength and N = length of added Genome's DNA sequence
- `int minimumSearchLenght() const`
  - Returns minimum search length passed to the constructor to determine min. Length of strings can be searched for
  - Runs in constant time, i.e. O(1)

Class #2: GenomeMatcher Methods

- `void addGenome(const Genome& genome)`

Genome 1: ACTG
Genome 2: TCGACT
Genome 3: TCTCG

For Genome 1:

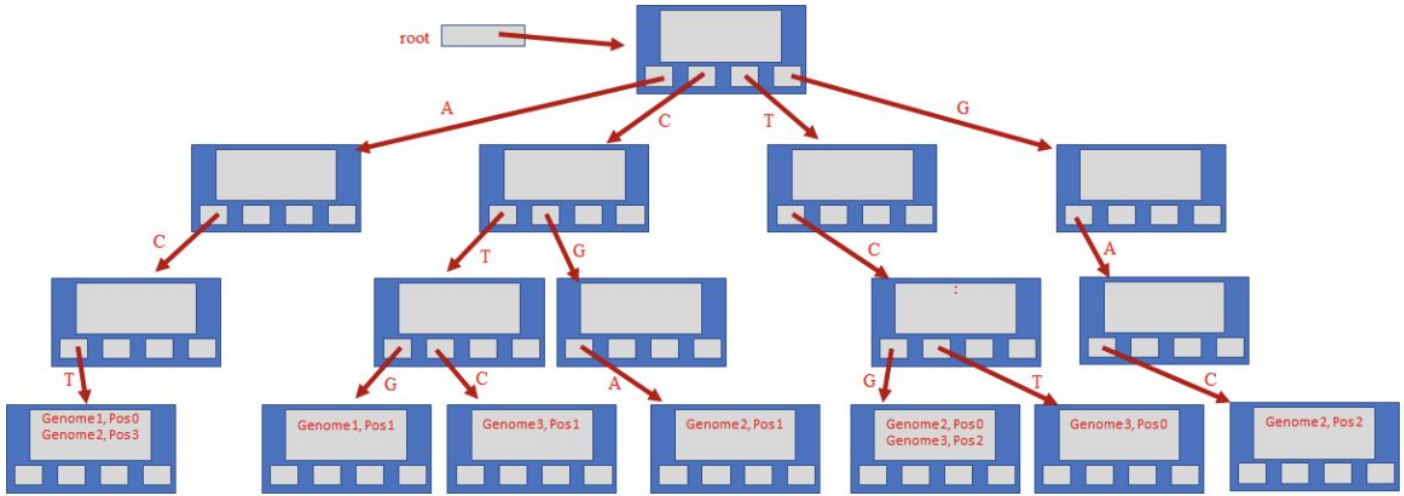
ACT → (Genome 1, position 0)
CTG → (Genome 1, position 1)

For Genome 2:

TCG → (Genome 2, position 0)
CGA → (Genome 2, position 1)
GAC → (Genome 2, position 2)
ACT → (Genome 2, position 3)

For Genome 3:

TCT → (Genome 3, position 0)
CTC → (Genome 3, position 1)



Class #2: GenomeMatcher Methods

```
bool findGenomeswithThisDNA(const string& fragment, int minimumLength, bool
exactMatchOnly, vector<DNAMatch>& matches) const
```

- Find all genomes in library that contain a specified DNA *fragment* or one of its SNiPs
- These must be of length *minimumLength* or more bases long
- Pass back a vector of longest match found
- If there are two exact matches, pass the earlier one
- It is **case-sensitive!**
- Read more of the specifics of what a match is and how/ why it returns false in specs pp. 23-24
- The vector contains DNAMatch's that is a struct defined as:

```
struct DNAMatch
{
        std::string genomeName;
        int position;
        int length;
};
```

Class #2: GenomeMatcher Methods

```
bool findRelatedGenomes(const Genomes& query, int fragmentMatchLength, bool
exactMatchOnly,double matchPercentThreshold, vector<GenomeMatch>& results) const
```

- Compared the query with all genomes in the library and passed back a vector of GenomeMatch's that are at least *matchPercentThreshold* of the base sequence
- Must use the algorithm given to receive full credit! See specs pp. 27-28
- Must run in O(Q*X) time, where Q = length in DNA bases of the *query* sequence and X = big-O of your *findGenomesWithThisDNA* function

Class #2: GenomeMatcher Methods

```
bool findRelatedGenomes(const Genomes& query, int fragmentMatchLength, bool
exactMatchOnly,double matchPercentThreshold, vector<GenomeMatch>& results) const
```

- Questions about the algorithm?

We will consider sequences of length *fragmentMatchLength* from the *query* genome starting at positions 0, 1* *fragmentMatchLength*,2* *fragmentMatchLength*, etc. (e.g., if *fragmentMatchLength* were 12, the start positions would be 0, 12, 24, 36). If the length of the *query* genome is not a multiple of *fragmentMatchLength*, we ignore the final sequence that is shorter than *fragmentMatchLength*. Let S be the number of sequences we will consider. For example, if the *query* genome were 800 bases long and *fragmentMatchLength* were 12, then since 800/12 is 66.6667, S will be 66 (since we ignore the final 8 base long sequence).

For each such sequence:

1. Extract that sequence from the queried genome.
2. Search for the extracted sequence across all genomes in the library (using *findGenomesWithThisDNA()*), allowing SNiP matches if *exactMatchOnly* is *false*).
3. If a match is found in one or more genomes in the library, then for each such genome, increase the count of matches found thus far for it.

For each genome g in the library that contained at least one matching sequence from the query genome:

1. Compute the percentage p of sequences from the query genome that were found in genome g by dividing the number of matching sequences found in genome g by S (e.g., if S is 66, and 15 of the 66 sequences were found in the genome, then 15/66 or 22.73% of the sequences from the *query* genome were found in that genome, so p will be 22.73).
2. If p is greater than the *matchPercentThreshold* parameter (a percentage in the range 0 though 100), then genome g is a match for the query.

Tips

- You do NOT need to use a hash tables/ unordered maps (you can if you want to, but it will overcomplicate things)
- Remember, it can be written in <400 lines of code! More lines does **not** mean better code.
  - If you find yourself writing >=500 lines, come to OH asap
- Just like project 3, think of your design first before writing any code. Drawing things out is extremely important, especially since we're trying to implement a Trie
- Know your STL containers and algorithms in the `<algorithm>` library, along with their big-O
  - Compare different containers/ algorithms to make sure you hit the required time complexity
- Read pp. 30-31 of the specs, it is extremely important for you to receive full credits
- Know how to run things on g32
  - You should know how by now, but if you don't talk to me or any of the TAs
  - There are resources on the class website about running your code on the Linux server
- Do not spend too much time in your report-- be clear and concise :)

Good Luck :)