Task

- ✓ Store all words, get 20 word w/prefix
- ✓ Store words not in the trie
- ✓ Explain

check folder for complete java files

```
Enter a prefix: pre
pre
prep
pres
prey
preen
preps
press
prexy
preys
prearm
preach
precut
precis
precox
preens
prefab
prefer
prefix
prelaw
premed
Pick: prelaw
       "prelaw" is learned
```

```
Enter a prefix: pre
prelaw
pre
prep
pres
prey
preen
preps
press
prexy
preys
prearm
preach
precut
precis
precox
preens
prefab
prefer
prefix
prelaw
Pick: prepare
       "prepare" is learned.
```

```
Enter a prefix: pre
prepare
prelaw
pre
prep
pres
prey
preen
preps
press
prexy
preys
prearm
preach
precut
precis
precox
preens
prefab
prefer
prefix
Pick: prep
      "prep" is learned.
```

Store all words, retrieve 20 words w/prefix - public List<String> getCandidates(String prefix)

In this method I create a List<String> that is an arraylist that will be filled with candidates. I then retrieve the node with the given prefix from my trie that is entered by the user. I need this node so that I can then traverse the trie.

I chose to take a recursive path to fill the list of candidates.

getWordsDownstream(foundNode, list, prefix);

> The first thing I do is retrieve any previous words chosen by the user recently if they had used that prefix before. I stored these values in a hashmap called recentPrefixes. After populating as many as I can with recent words, I then continue with getWordsDownstream(Node, list);

getWordsDownstream(Node, list);

> This is the main recursive part. It begins by making a map called childmap that will be used later on when assessing the children of the current node. The real first action is to check if the current node is an end state (my base case). If so it then adds the word to my list(by traversing up the trie recursively through my getWordupstream) as long as it currently has less than 20 candidates.

getWordupstream

This method is simple in nature. It continues getting the value of the parent until it reaches head. Concatenates the letters, and returns the word(to then be added to the candidate list).

Back to getWordsDownstream. After checking the base case, my program then checks if the current node has children.

If the current node has children then we must get map of the children and save it into childmap. Afterwards, we then take the set of all children and add each one singly into a deque called myDeq. This is how my program uses breadth first search. After pushing each child node into the stack, as long as the deque is not empty we recursively call the same method getWordsDownstream but this time entering the node from the bottom of the stack. This way my code traverses the trie layers at a time as opposed to digging really deep (depth first search).

If the current node did not have any children then the program would continue recursively calling getWordsDownstream but with the bottom most node (first in, first out) – As long as the deque is not empty.

If the deque is empty, then you've traversed all the possible nodes checking for true endstates, so the program returns whatever list has been populated so far.

After all the recursion, I then clear the deque right before returning the list of 20 candidates.

Store words not in the trie - public void pickCandidate(String prefix, String candidate)

In this method I was halfway to implementing the version where more frequent words got a weight higher than just those picked most recent. I was going to do this with a hashMap<String, Double>() to keep track of the number of times a word had been a candidate to reorder the List<String> in the HashMap<String,List<String>>() of recentPrefixes. Maybe I'll get partial extra credit :D for being close but running out of time to continue playing with it.

pickCandidate

The practical part of this method right now merely just adds the candidates picked into the recentPrefixes hashmap List if the prefix for the candidate has been used before. If the prefix hasn't been used before then the method creates a new key in the hashmap recentPrefixes that equals the prefix and adds the candidate as its only word in the list.

As mentioned before, this method also contains some counter stuff that I was using to attempt the extra credit.

Closing Thoughts

I was close to getting the weighted recent list by using most frequent, but it didn't happen. My program does as it's supposed to in terms of retrieving 20 candidates matching the prefix as well as having the most recently selected candidate from the prefix be at the top.