Tries and Applications

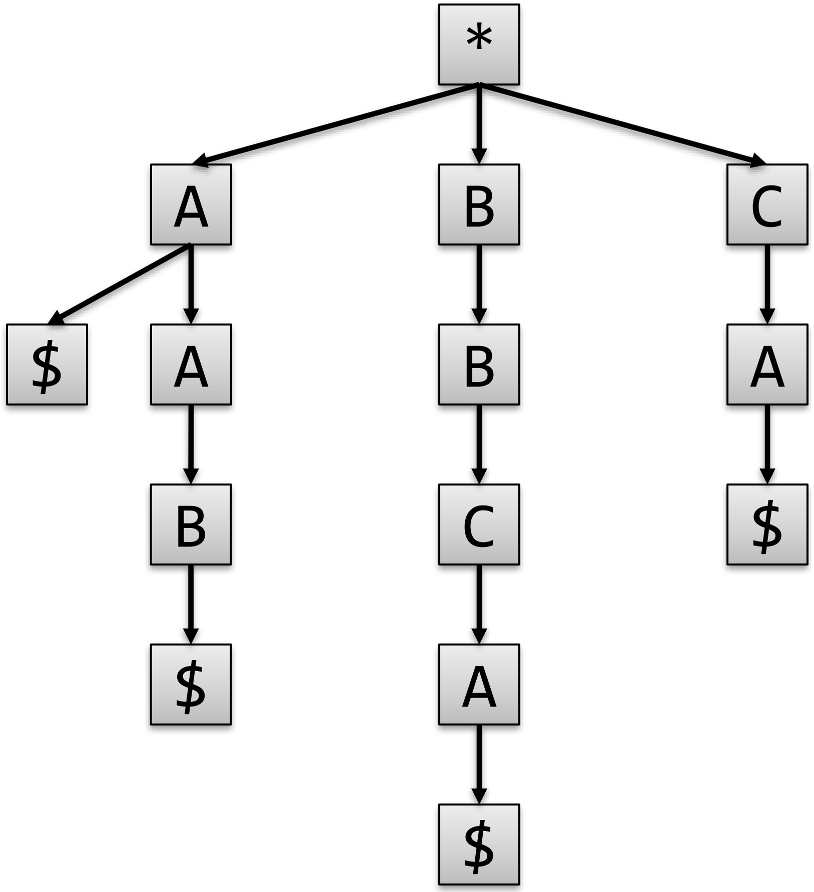
ITP 435

Week 4, Lecture 2

Tries

What's a trie?

- Usually pronounced "try"
- A tree used (typically) to store strings and allows for efficient matching of patterns and implementing things like autocomplete
- Example on the right (ignore the * and $ for now)



What to store for each node?

We're going to "rule of zero" this, so no raw pointers!

- Array of `unique_ptrs` to children – number of elements is ***alphabet size + 1*** (we'll talk about why the + 1 later)

- `char` for the letter stored at the node

How will we declare the trie?

```
template <size_t AlphabetSize, typename LetterToIdxFunc>
class Trie
```

- AlphabetSize – How many different letters there are in the alphabet the trie needs to support (for space efficiency)

- LetterToIdxFunc – A function to map a letter to a specific index in the array of children we store at each node

Sample LetterToIdxFunc

- Suppose we have an alphabet size of 3 and the three letters are A, B, and C:

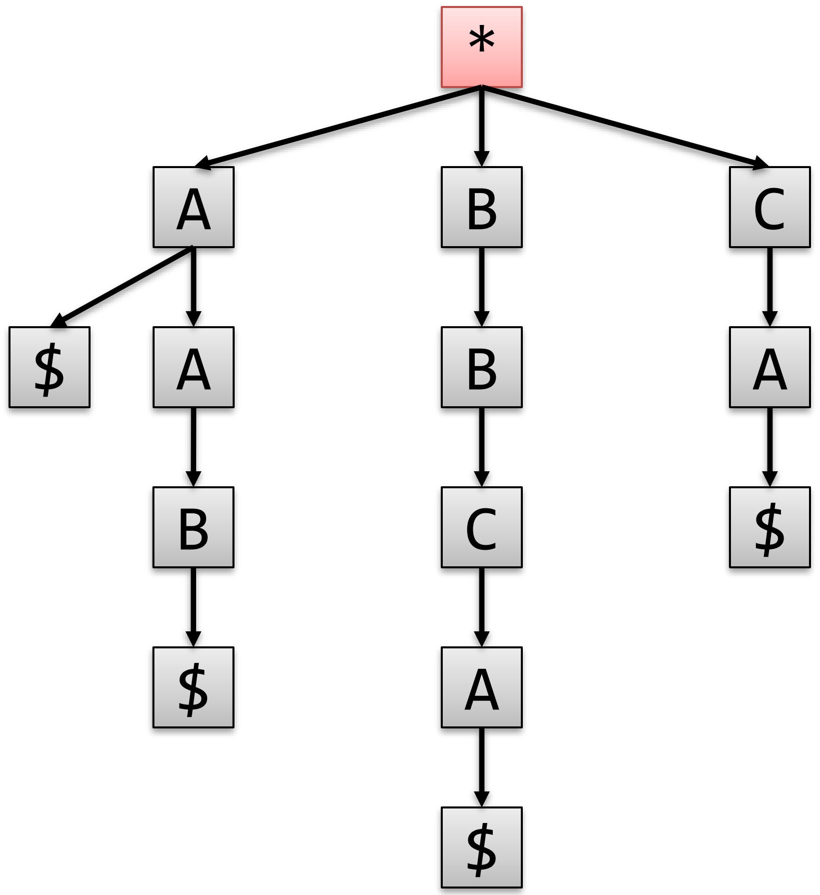
```
struct BasicTextIdx {
   size_t operator()(char c) const {
      switch (c) {
         case 'A': return 0;
         case 'B': return 1;
         case 'C': return 2;
         default: return 0;
      }
   }
};
```

What member data does the trie have?

- Pointer to root node
- Instance of LetterToIdxFunc object

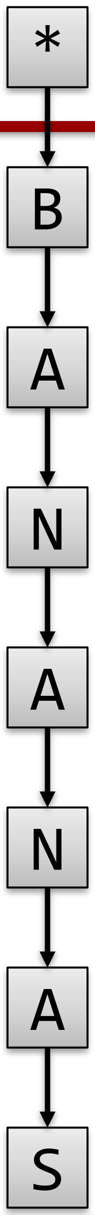
What letter do we store at the root node?

- We'll just use *
- (This assumes * doesn't exist in the alphabet)



Handling Common Prefixes

- Suppose we need to store “BANANA” ***and*** “BANANAS”
- The proposed trie on the right does not encode enough information

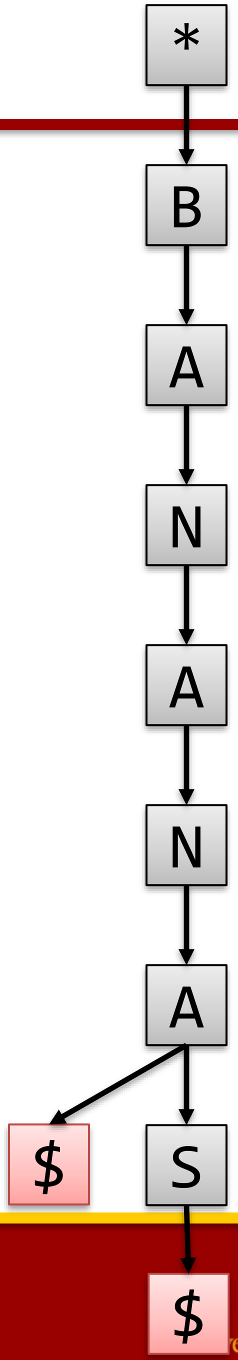


Handling Common Prefixes

- Suppose we need to store "BANANA" ***and*** "BANANAS"

- Use a special character $ to denote the end of a string

- (This assumes $ is not a valid letter in the alphabet)



Accounting for $ and the LetterToIdxFunc

- Just assume that $ is always index 0
- Add 1 to LetterToIdxFunc result to get the correct final location in the array:

```
struct BasicTextIdx {
   size_t operator()(char c) const {
      switch (c) {
         case 'A': return 0;
         case 'B': return 1;
         case 'C': return 2;
         default: return 0;
      }
   }
};
```

- Even though BasicTextIdx says ‘A’ is at 0, we’ll actually store it at 1

Insertion

Insertion

- The general idea is you get a string to insert and when you insert you either use existing nodes when appropriate or create new nodes if they don't already exist

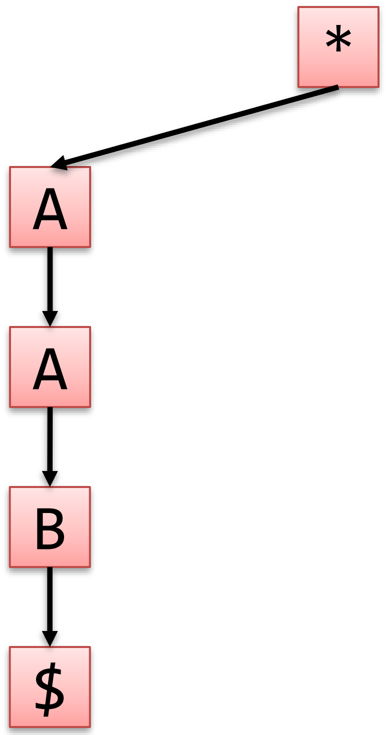
```
void Insert(std::string_view word)
```

Sample

```
Trie<3, BasicTextIdx> trie;
trie.Insert("AAB");
trie.Insert("A");
trie.Insert("BBCA");
trie.Insert("CA");
```

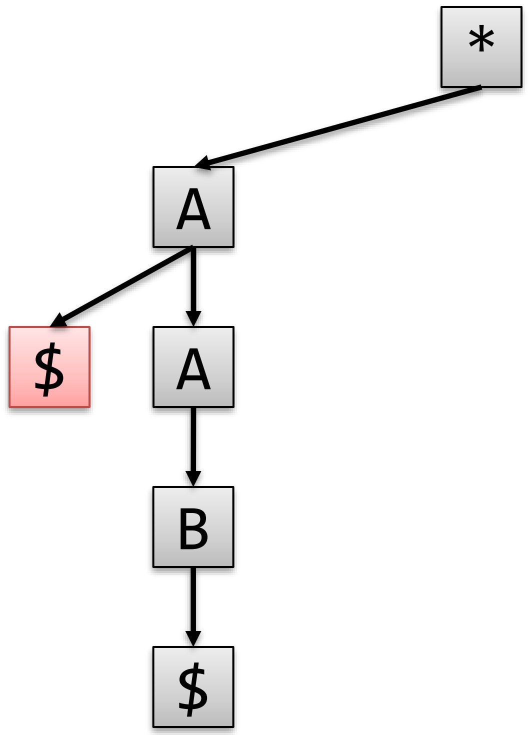
Sample

```
Trie<3, BasicTextIdx> trie;
trie.Insert("AAB");
trie.Insert("A");
trie.Insert("BBCA");
trie.Insert("CA");
```



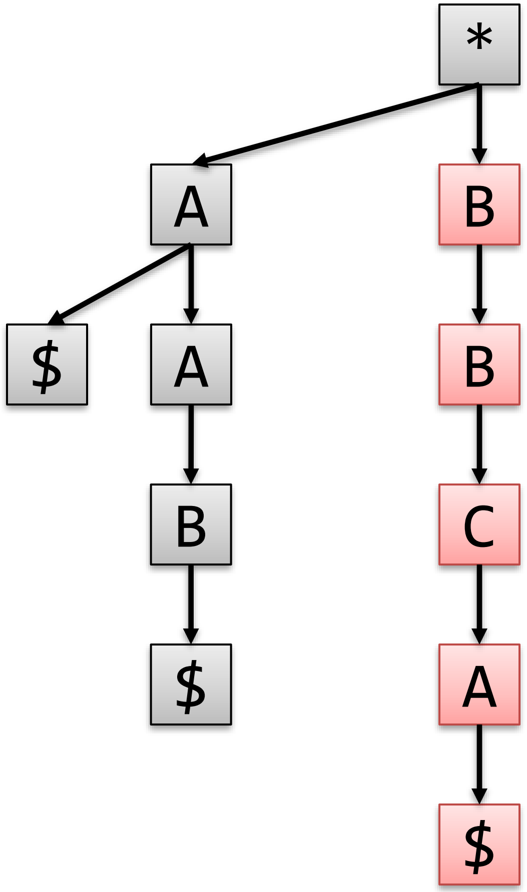
Sample

```
Trie<3, BasicTextIdx> trie;
trie.Insert("AAB");
trie.Insert("A");
trie.Insert("BBCA");
trie.Insert("CA");
```



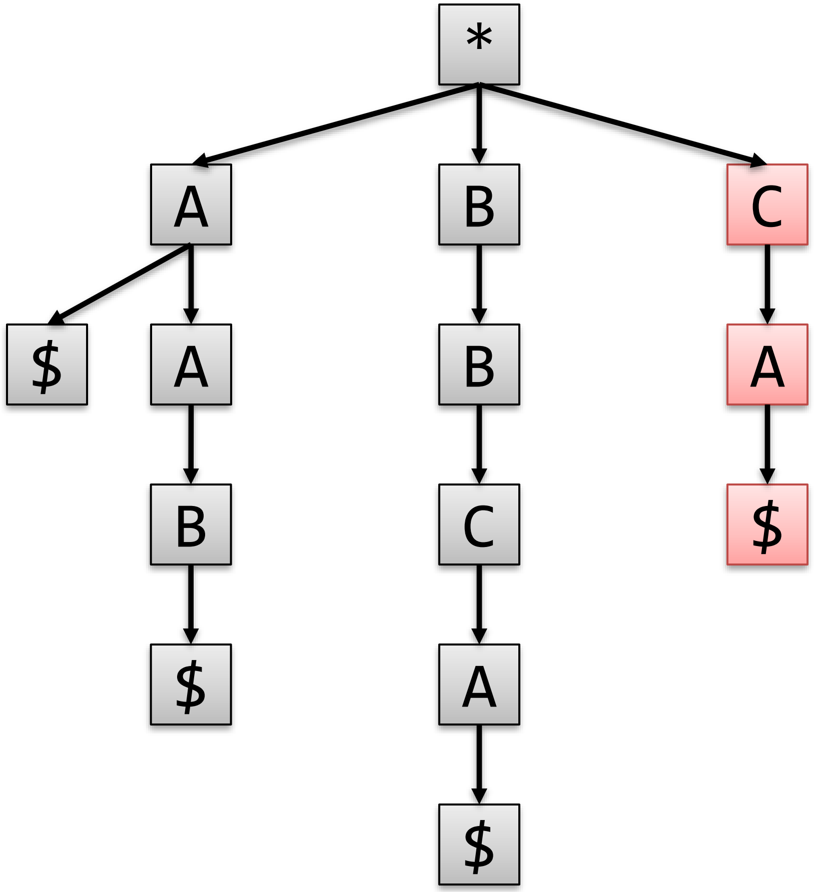
Sample

```
Trie<3, BasicTextIdx> trie;
trie.Insert("AAB");
trie.Insert("A");
trie.Insert("BBCA");
trie.Insert("CA");
```



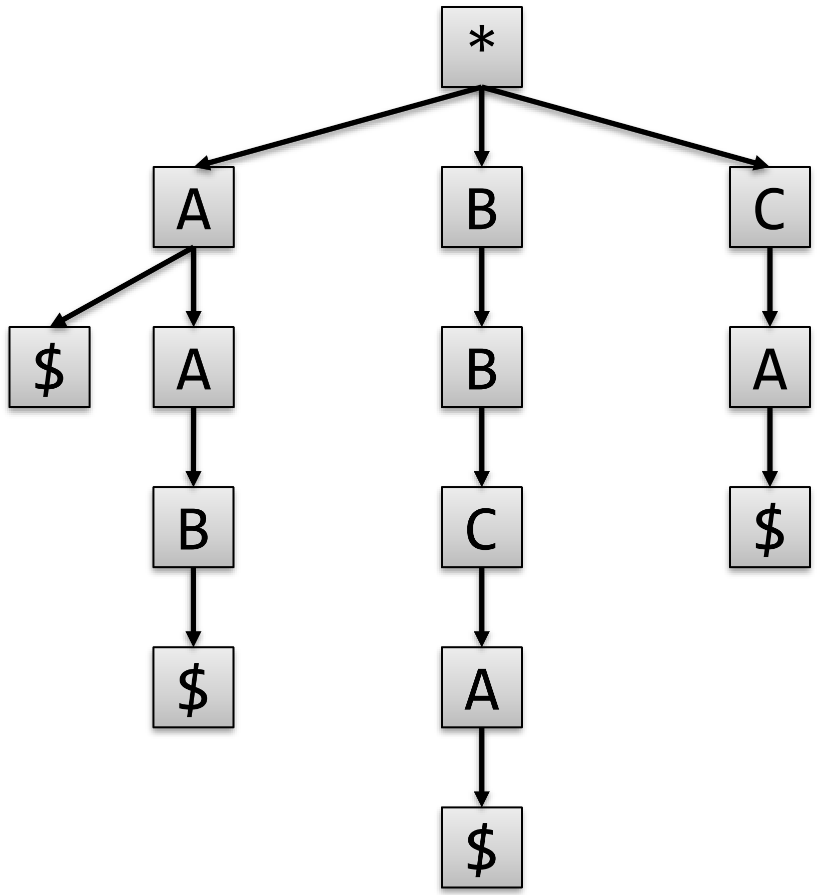
Sample

```
Trie<3, BasicTextIdx> trie;
trie.Insert("AAB");
trie.Insert("A");
trie.Insert("BBCA");
trie.Insert("CA");
```


$$\begin{tikzcd}
 & & & * \arrow[dll] \arrow[d] \arrow[dr] & \\
 & A \arrow[dl] \arrow[d] & & B \arrow[d] & C \arrow[d] \\
\$ & A \arrow[d] & & B \arrow[d] & A \arrow[d] \\
 & B \arrow[d] & & C \arrow[d] & \$ \\
 & \$ & & A \arrow[d] & \\
 & & & \$ &
\end{tikzcd}$$

Sample

```
Trie<3, BasicTextIdx> trie;
trie.Insert("AAB");
trie.Insert("A");
trie.Insert("BBCA");
trie.Insert("CA");
```


$$\begin{tikzcd}
 & & & * \arrow[dll] \arrow[d] \arrow[drr] & & \\
 & A \arrow[dl] \arrow[d] & & B \arrow[d] & & C \arrow[d] \\
\$ & A \arrow[d] & & B \arrow[d] & & A \arrow[d] \\
 & B \arrow[d] & & C \arrow[d] & & \$ \\
 & \$ & & A \arrow[d] & & \\
 & & & \$ & &
\end{tikzcd}$$

Breadth-First-Search (BFS)

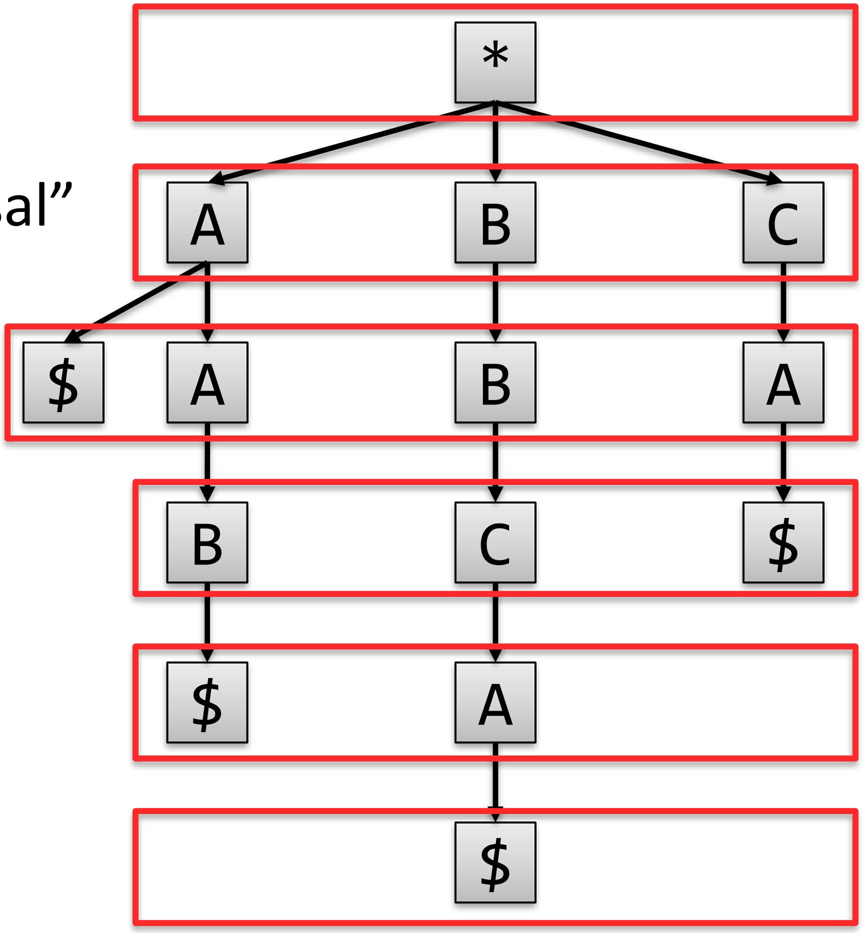
Breadth-First-Search Traversal

- This is not a particularly useful operation on a trie, but it is helpful for writing test cases to ensure the trie is constructed to spec!
- Will call `visitFunc` on each node in the tree in a BFS manner, starting at root

```
void BFS(std::function<void(char)> visitFunc)
```

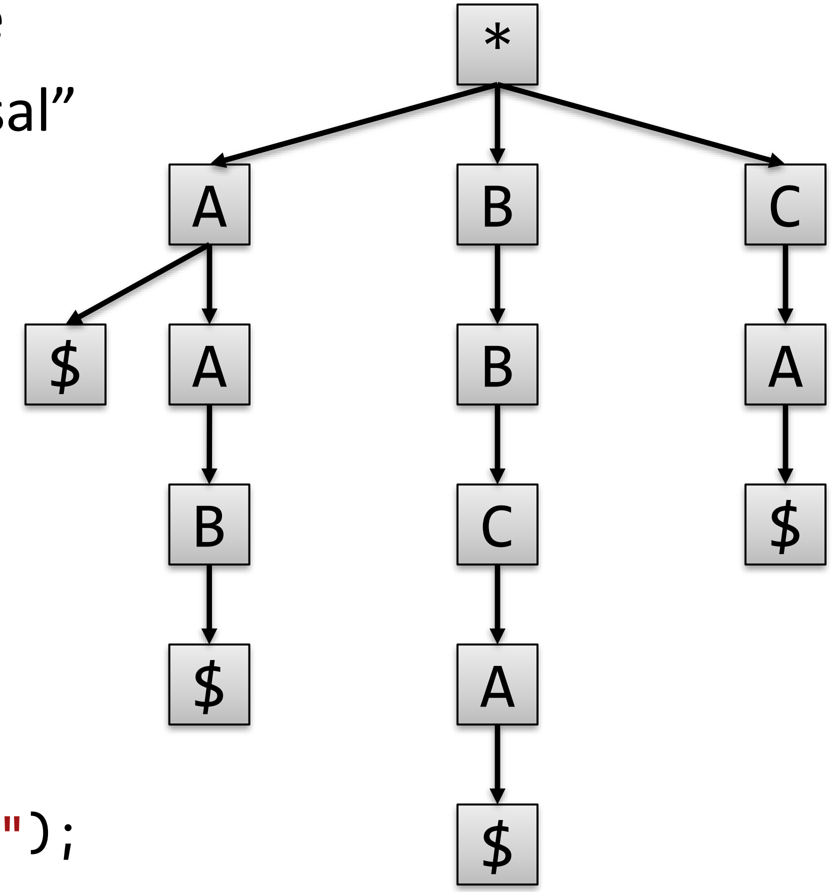
Breadth-First-Search

- Can just implement using a std::queue
- Equivalent to "level-order traversal"



Breadth-First-Search Example

- Can just implement with a queue
- Equivalent to “level-order traversal”



```
std::string result;
trie.BFS([&result](char c) {
    result += c;
});
REQUIRE(result == "*ABC$ABABC$$A$");
```

FindPrefix

FindPrefix

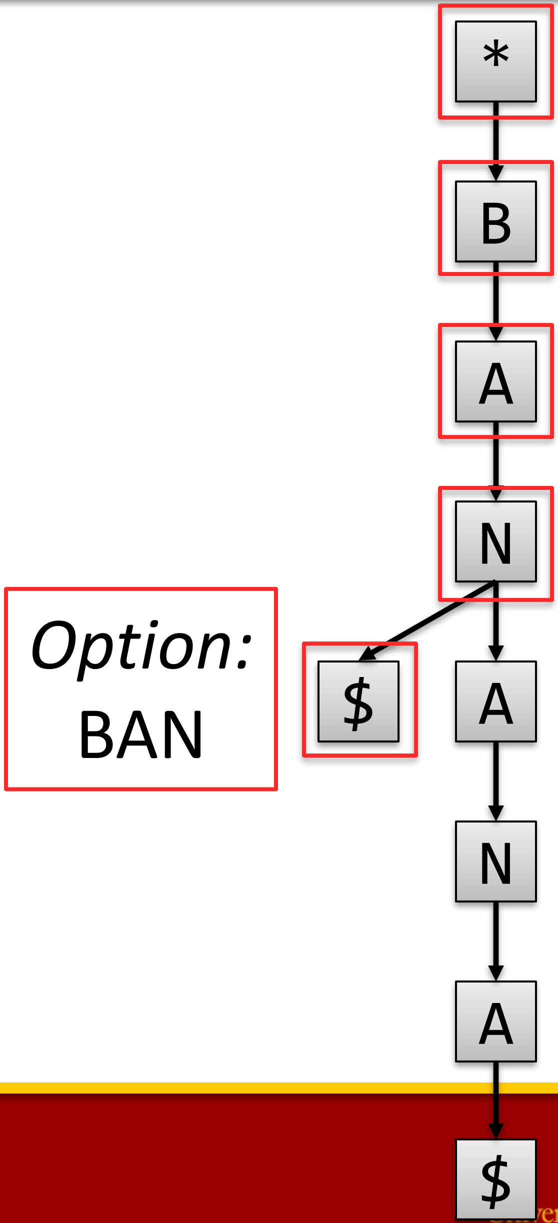
- Given a string, finds the longest prefix of that string which exists in the trie (or an empty string if none exists)

```
std::string FindPrefix(std::string_view word)
```

FindPrefix Examples

```
Trie<26, EnglishTextIdx> t;
t.Insert("BAN");
t.Insert("BANANA");


t.FindPrefix("BAN")
??
"BAN"
```

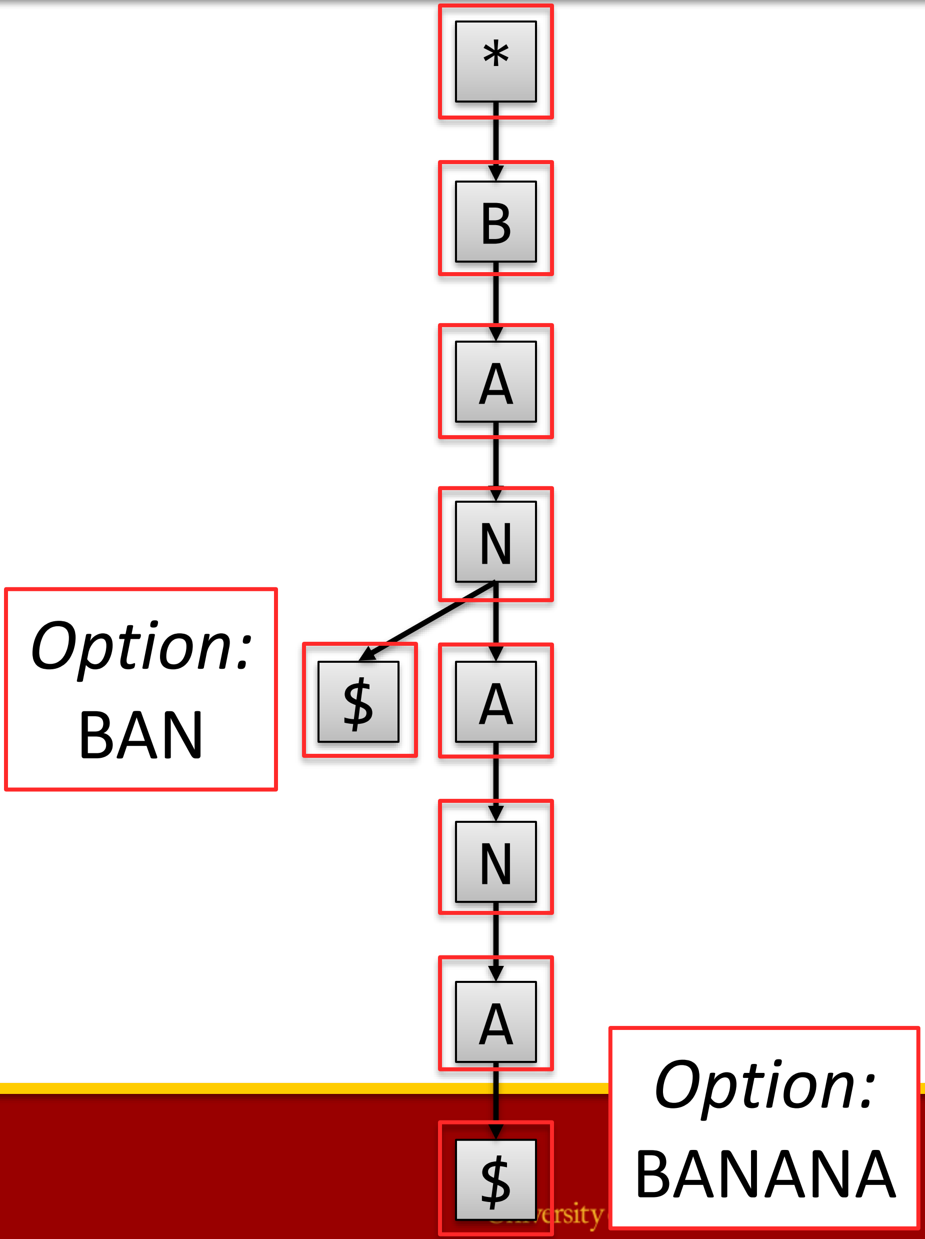


FindPrefix Examples

```
Trie<26, EnglishTextIdx> t;
t.Insert("BAN");
t.Insert("BANANA");


t.FindPrefix("BANANAS")
??
"BANANA"
```

To implement this efficiently, update the best option along the way during the search



In-class Week 4 Lecture 2

- FindPrefix practice

CompleteFromPrefix

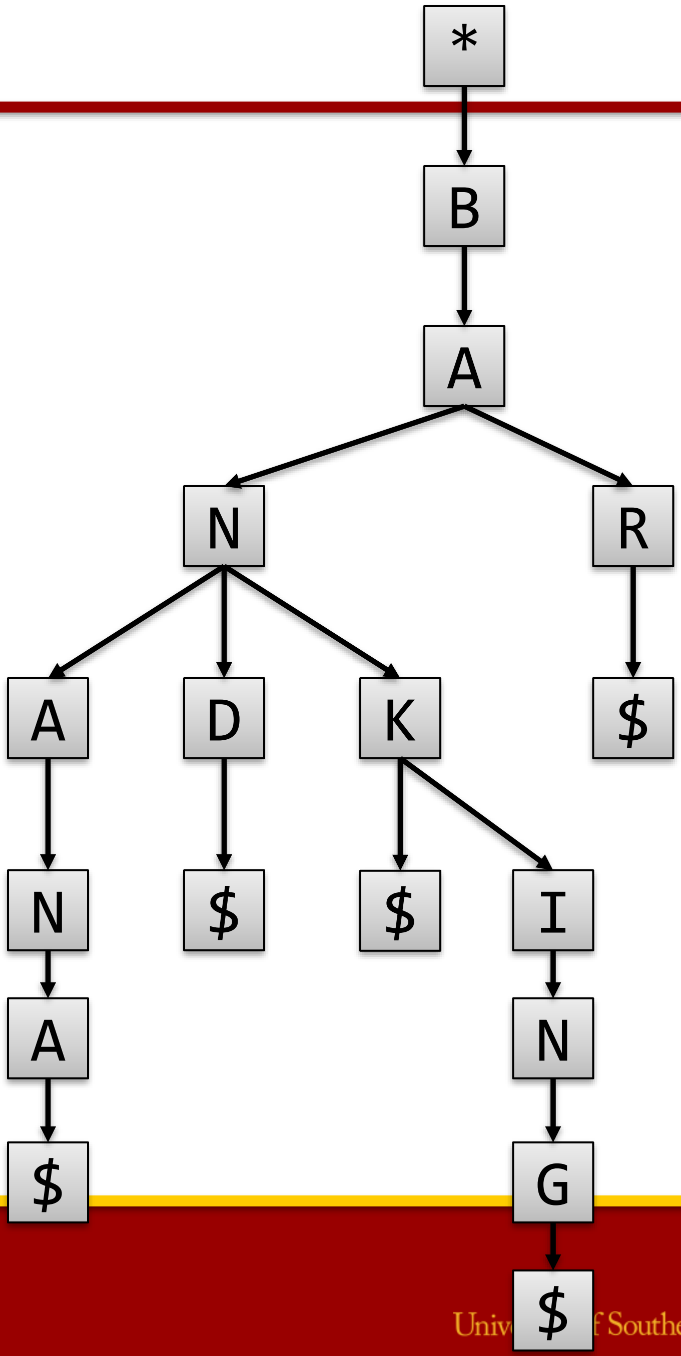
CompleteFromPrefix

- Given a string, finds the X shortest words in the trie that match the prefix

```
std::vector<std::string>
CompleteFromPrefix(std::string_view prefix,
                                 size_t count = 3)
```

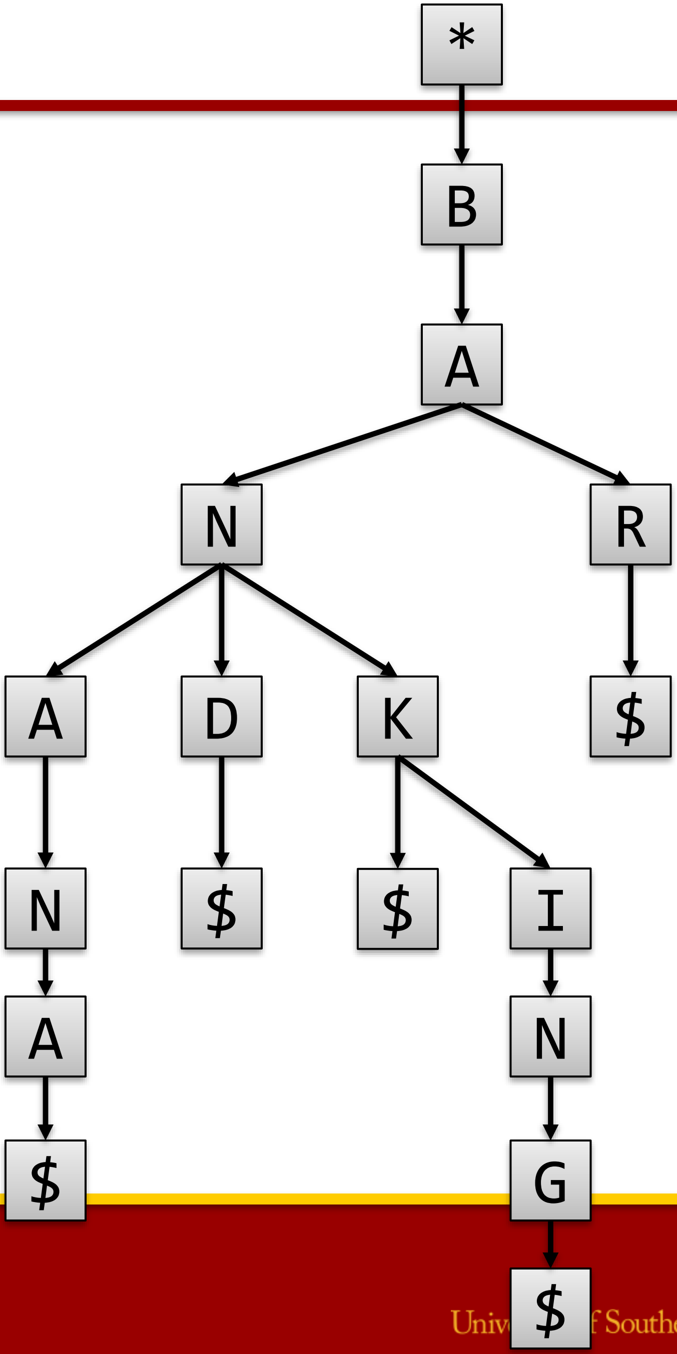
CompleteFromPrefix Example

- Given this trie



CompleteFromPrefix Example

```
t.CompleteFromPrefix("BAN");
```

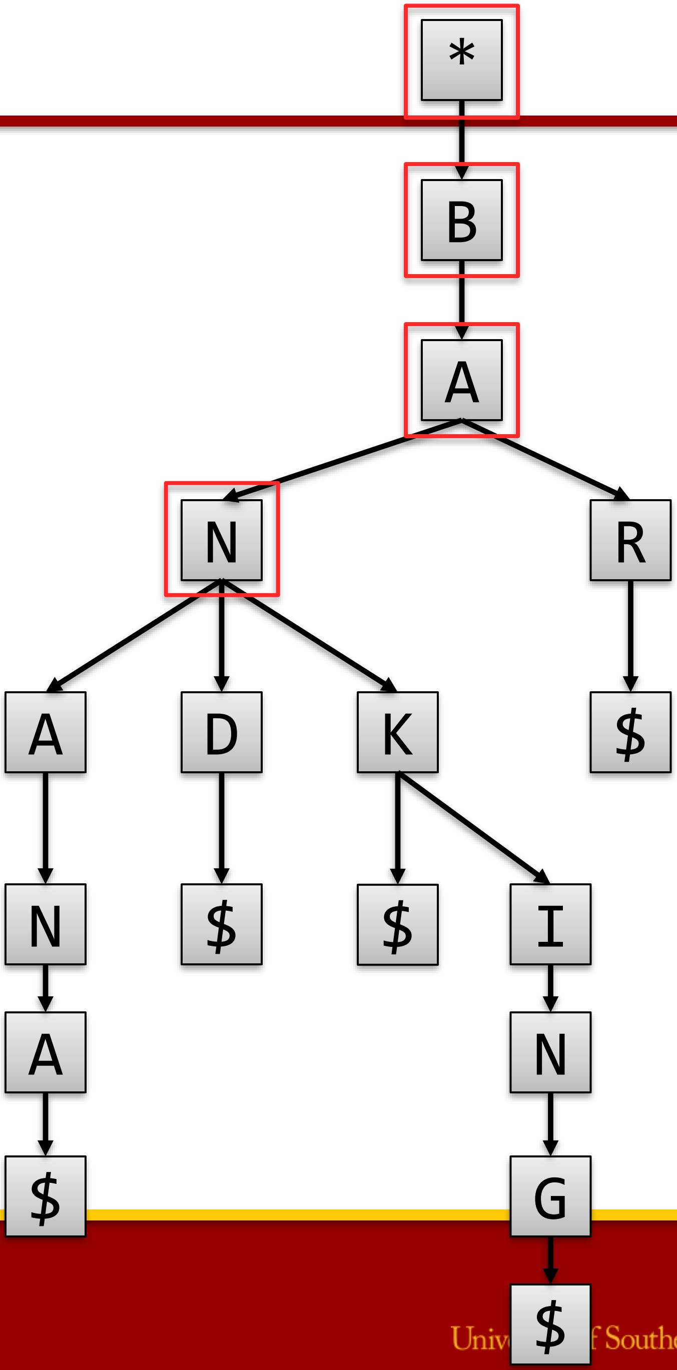


CompleteFromPrefix Example

```
t.CompleteFromPrefix("BAN");
```

First, match "BAN"

(*If there is no match in the trie then there are no results*)

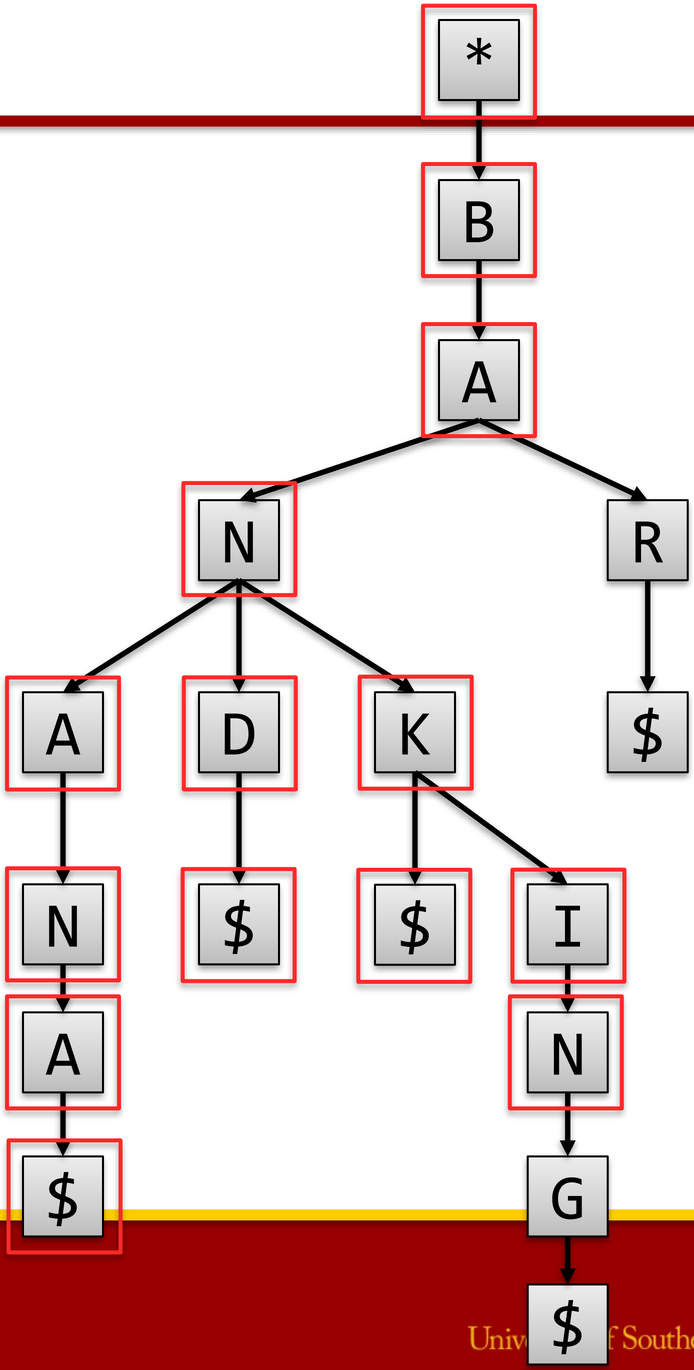


CompleteFromPrefix Example

```
t.CompleteFromPrefix("BAN");
```

Next, do a BFS from that node stopping when you hit count complete words (or stopping if the BFS runs out)

BAND
BANK
BANANA

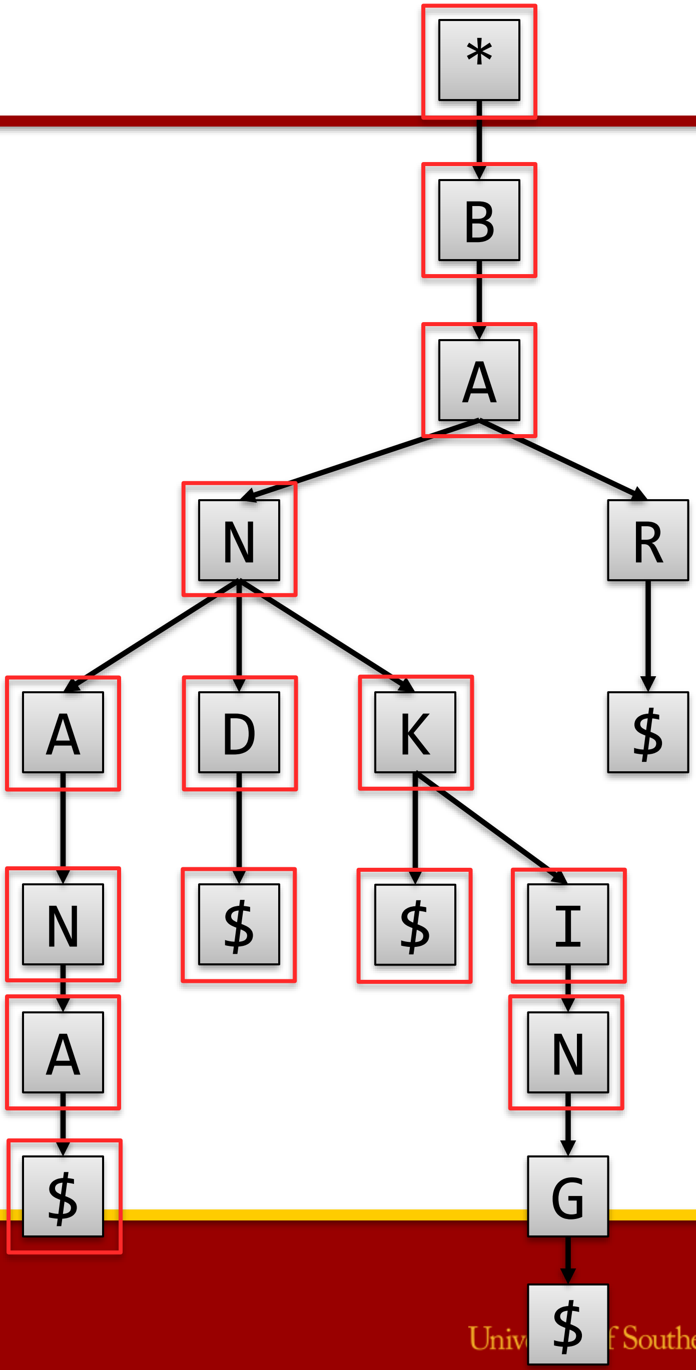


CompleteFromPrefix Example

How to reconstruct the words from the trie?

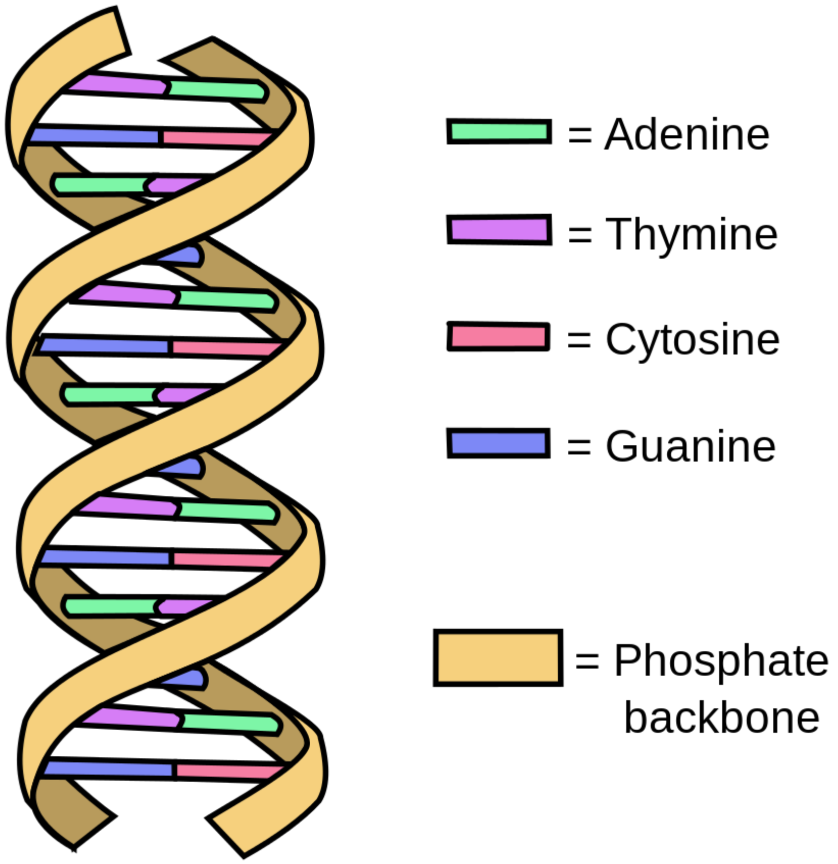
Option 1: During the BFS also save the partially-completed word in the queue, so you can just add the additional letter as it continues (fastest)

Option 2: Once you hit a $, traverse the parents back to root and reverse the word



DNA Pattern Matching

DNA



- A, T, C, and G are called ***nucleotides***

FASTA File Format

- A simple, text-based file format used to describe (among other things) DNA nucleotide sequences

- First line has a comment/description of the file, and subsequent lines have a sequence of nucleotides

- Example:

```
>gi|319999821:c124527448-124526573 Pan troglodytes isolate Yerkes chimp
ATGATACCCATCCAACTCACTGTCTTCTTCATGATCATCTATGTGCTTGAGTCCTTGACAATTATTGTGCAG
AGCAGCCTAATTGTTGCAGTGCTGGGCAGAGAATGGCTGCAAGTCAGAAGGCTGATGCCTGTGGACATGATT
CTCATCAGCCTGGGCATCTCTCGCTTCTGTCTACAGTGGGCATCAATGCTGAACAATTTTTGCTCCTATTTT
AATTTGAATTATGTACTTTGCAACTTAACAATCACCTGGGAATTTTTAATATCCTTACATTCT
```

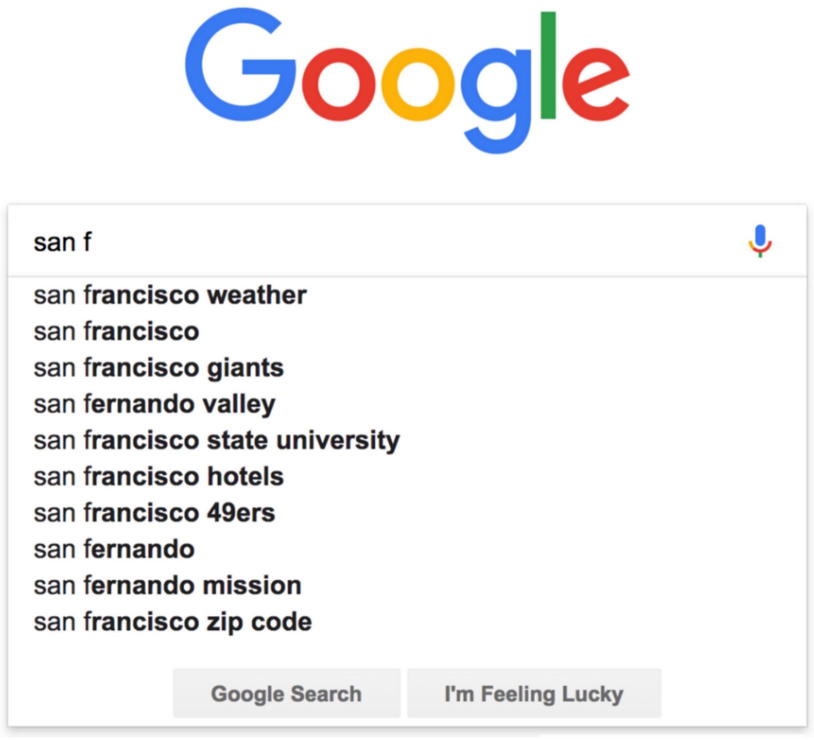
DNA Pattern Matching

- Given a long DNA sequence, find the largest prefix in the trie that matches
- Start at letter 0 of the sequence, `FindPrefix`, then go on to letter 1, `FindPrefix`, etc
- Will use to find some transcription factor proteins

Autocomplete

Autocomplete

- Essentially just have a large dictionary you load into the trie, and given the prefix use `CompleteFromPrefix`



san f

san francisco weather
san francisco
san francisco giants
san fernando valley
san francisco state university
san francisco hotels
san francisco 49ers
san fernando
san fernando mission
san francisco zip code