Tries
Apply to Section Lead!

http://cs198.stanford.edu
Implementing **Lexicon**

- The **Lexicon** represents a set of English words.

- Main operations:
  - Add word.
  - Remove word.
  - Is word contained?
  - Is prefix contained?

- How can we efficiently implement the **Lexicon**?
An Inefficient Implementation

- We could implement the **Lexicon** as a list of all the words it contains.

- To add a word:
  - Check if the word already exists.
  - If not append it.

- To remove a word:
  - Find and remove it from the list.

- To see if a word exists:
  - Search the list for the word.

- To see if a prefix exists:
  - Check every word in the list to see if the word is a prefix.
A Better Implementation

• Use a hash table.
• Adding, removing, and checking for a word now runs very quickly (O(1) comparisons needed).
• How would you implement containsPrefix?
A Better Implementation

- Use a hash table.
- Adding, removing, and checking for a word now runs very quickly (O(1) comparisons needed).
- How would you implement `containsPrefix`?
- Would have to check all words in all buckets.
Rethinking Hashing

- Our motivation behind hashing was to put values into places where we would know to look for them.
- When storing strings as our keys, one initial idea was to break strings apart by their first letter.
- Let's look at that idea again.
An Initial Hashing Idea


Dumbledore  Harry  Lily  Minerva  Ron  Voldemort

Draco  Hermione  Snape

Hagrid
A
AB
ABOUT
AD
ADAGE
ADAGIO

BAR
BARD
BARN
BE
BED
BET
BETA

CAN
CANE
CAT

DIKDIK
DIKTAT
ADAGIO
A

B

C

D

CAN

CANE

CAT

DIKDIK

DIKTAT

BAR

BARD

BARN

BE

BED

BET

BETA

O

U

T

E

I

O
ABOUT
Tries

- The data structure we have just seen is called a trie.
- Comes from the word retrieval.
- Pronounced “try,” not “tree.”
Trie Nodes

- The pieces of the trie are called **nodes**.
- Each node stores two pieces of information:
  - Whether, at this point in the trie, you have arrived at a word, and
  - Pointers to child nodes in the trie.
- The node at the top of the trie is called the **root node**.
Let's trie coding up Lexicon!
Analyzing the Trie

- How efficient are the operations on the trie?
- Every operation takes time proportional to the length of the string, which we'll denote $L$.
- Time to add or look up an element is $O(L)$.
- Time to check if a prefix exists is $O(L)$. 
Removing from a Trie

- Mark the node as no longer containing a word.
- If the node has no children:
  - Remove that node.
  - Repeat this process at the node one level higher up in the tree.