We begin with a simple problem: a lexicon of some natural language is given as a list of words. Suggest a data structure that will provide insertion and retrieval of data. As a first solution, we are looking for time efficiency rather than space efficiency. The solution: *trie* (word tree).

Access time: $O(|w|)$. Space requirement: $O(\sum_w |w|)$.

A trie can be augmented to store also a morphological dictionary specifying concatenative affixes, especially suffixes. In this case it is better to turn the tree into a graph.

The obtained model is that of *finite-state automata*. 
Finite-state technology

Finite-state automata are not only a good model for representing the lexicon, they are also perfectly adequate for representing dictionaries (lexicons + additional information), describing morphological processes that involve concatenation etc. A natural extension of finite-state automata – finite-state transducers – is a perfect model for most processes known in morphology and phonology, including non-segmental ones.
Formal languages are defined with respect to a given *alphabet*, which is a finite set of symbols, each of which is called a *letter*. A finite sequence of letters is called a *string*.

**Example: Strings**

Let $\Sigma = \{0, 1\}$ be an alphabet. Then all binary numbers are strings over $\Sigma$.

If $\Sigma = \{a, b, c, d, \ldots, y, z\}$ is an alphabet then *cat*, *incredulous* and *supercalifragilisticexpialidocious* are strings, as are *tac*, *qqq* and *kjshdflkwjehr*. 
The *length* of a string $w$, denoted $|w|$, is the number of letters in $w$. The unique string of length 0 is called the *empty string* and is denoted $\epsilon$.

If $w_1 = \langle x_1, \ldots, x_n \rangle$ and $w_2 = \langle y_1, \ldots, y_m \rangle$, the *concatenation* of $w_1$ and $w_2$, denoted $w_1 \cdot w_2$, is the string $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$. 

$|w_1 \cdot w_2| = |w_1| + |w_2|$.

For every string $w$, $w \cdot \epsilon = \epsilon \cdot w = w$. 
Example: Concatenation

Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet. Then $\text{master} \cdot \text{mind} = \text{mastermind}$, $\text{mind} \cdot \text{master} = \text{mindmaster}$ and $\text{master} \cdot \text{master} = \text{mastermaster}$. Similarly, $\text{learn} \cdot \text{s} = \text{learns}$, $\text{learn} \cdot \text{ed} = \text{learned}$ and $\text{learn} \cdot \text{ing} = \text{learning}$.
An *exponent* operator over strings is defined in the following way: for every string $w$, $w^0 = \epsilon$. Then, for $n > 0$, $w^n = w^{n-1} \cdot w$.

**Example: Exponent**

If $w = go$, then $w^0 = \epsilon$, $w^1 = w = go$, $w^2 = w^1 \cdot w = w \cdot w = gogo$, $w^3 = gogogo$ and so on.
The *reversal* of a string $w$ is denoted $w^R$ and is obtained by writing $w$ in the reverse order. Thus, if $w = \langle x_1, x_2, \ldots, x_n \rangle$, $w^R = \langle x_n, x_{n-1}, \ldots, x_1 \rangle$.

Given a string $w$, a *substring* of $w$ is a sequence formed by taking contiguous symbols of $w$ in the order in which they occur in $w$. If $w = \langle x_1, \ldots, x_n \rangle$ then for any $i, j$ such that $1 \leq i \leq j \leq n$, $\langle x_i, \ldots x_j \rangle$ is a substring of $w$.

Two special cases of substrings are *prefix* and *suffix*: if $w = w_l \cdot w_c \cdot w_r$ then $w_l$ is a prefix of $w$ and $w_r$ is a suffix of $w$. 
Example: Substrings

Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet and $w = \text{indistinguishable}$ a string over $\Sigma$. Then $\epsilon$, $\text{in}$, $\text{indis}$, $\text{indistinguishish}$ and $\text{indistinguishable}$ are prefixes of $w$, while $\epsilon$, $e$, $\text{able}$, $\text{distinguishable}$ and $\text{indistinguishable}$ are suffixes of $w$. Substrings that are neither prefixes nor suffixes include $\text{distinguish}$, $\text{gui}$ and $\text{is}$. 
Given an alphabet $\Sigma$, the set of all strings over $\Sigma$ is denoted by $\Sigma^*$. A formal language over an alphabet $\Sigma$ is a subset of $\Sigma^*$. 
Example: Languages

Let $\Sigma = \{a, b, c, \ldots, y, z\}$. The following are formal languages:

- $\Sigma^*$;
- the set of strings consisting of consonants only;
- the set of strings consisting of vowels only;
- the set of strings each of which contains at least one vowel and at least one consonant;
- the set of palindromes;
Example: Languages

Let $\Sigma = \{a, b, c, \ldots, y, z\}$. The following are formal languages:

- the set of strings whose length is less than 17 letters;
- the set of single-letter strings ($= \Sigma$);
- the set $\{i, you, he, she, it, we, they\}$;
- the set of words occurring in Joyce’s Ulysses;
- the empty set;

Note that the first five languages are infinite while the last five are finite.
The string operations can be lifted to languages. If $L$ is a language then the *reversal* of $L$, denoted $L^R$, is the language $\{w \mid w^R \in L\}$.
If $L_1$ and $L_2$ are languages, then
$L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$.

**Example: Language operations**

$L_1 = \{i, you, he, she, it, we, they\}$, $L_2 = \{smile, sleep\}$.
Then $L_1^R = \{i, uoy, eh, ehs, ti, ew, yeht\}$ and $L_1 \cdot L_2 = \{ismile, yousmile, hesmile, shesmile, itsmile, wesmile, theysmile, isleep, yousleep, hesleep, shesleep, itsleep, wesleep, theysleep\}$. 
If $L$ is a language then $L^0 = \{\epsilon\}$.
Then, for $i > 0$, $L^i = L \cdot L^{i-1}$.

**Example: Language exponentiation**

Let $L$ be the set of words $\{bau, haus, hof, frau\}$. Then $L^0 = \{\epsilon\}$, $L^1 = L$ and $L^2 = \{baubau, bauhaus, bauhof, baufrau, hausbau, haushaus, haushof, hausfrau, hobau, hofhaus, hofhof, hoffrau, fraubau, frauhaus, frauhof, fraufrau\}$.
The Kleene closure of $L$ and is denoted $L^*$ and is defined as

$$
\bigcup_{i=0}^{\infty} L^i.
$$

$L^+ = \bigcup_{i=1}^{\infty} L^i$.

**Example: Kleene closure**

Let $L = \{\text{dog, cat}\}$. Observe that $L^0 = \{\epsilon\}$, $L^1 = \{\text{dog, cat}\}$, $L^2 = \{\text{catcat, catdog, dogcat, dogdog}\}$, etc. Thus $L^*$ contains, among its infinite set of strings, the strings $\epsilon, \text{cat, dog, catcat, catdog, dogcat, dogdog, catcatcat, catdogcat, dogcatcat, dogdogcat}$, etc.

The notation for $\Sigma^*$ should now become clear: it is simply a special case of $L^*$, where $L = \Sigma$. 
Regular expressions are a formalism for defining (formal) languages. Their “syntax” is formally defined and is relatively simple. Their “semantics” is sets of strings: the denotation of a regular expression is a set of strings in some formal language.
Regular expressions are defined recursively as follows:

- $\emptyset$ is a regular expression
- $\epsilon$ is a regular expression
- if $a \in \Sigma$ is a letter then $a$ is a regular expression
- if $r_1$ and $r_2$ are regular expressions then so are $(r_1 + r_2)$ and $(r_1 \cdot r_2)$
- if $r$ is a regular expression then so is $(r)^*$
- nothing else is a regular expression over $\Sigma$. 
Example: Regular expressions

Let $\Sigma$ be the alphabet $\{a, b, c, \ldots, y, z\}$. Some regular expressions over this alphabet are:

- $\emptyset$
- $a$
- $((c \cdot a) \cdot t)$
- $((((m \cdot e) \cdot (o)^*) \cdot w)$
- $(a + (e + (i + (o + u))))$
- $(((a + (e + (i + (o + u))))^*)$
For every regular expression $r$ its denotation, $\llbracket r \rrbracket$, is a set of strings defined as follows:

- $\llbracket \emptyset \rrbracket = \emptyset$
- $\llbracket \epsilon \rrbracket = \{ \epsilon \}$
- if $a \in \Sigma$ is a letter then $\llbracket a \rrbracket = \{ a \}$
- if $r_1$ and $r_2$ are regular expressions whose denotations are $\llbracket r_1 \rrbracket$ and $\llbracket r_2 \rrbracket$, respectively, then $\llbracket (r_1 + r_2) \rrbracket = \llbracket r_1 \rrbracket \cup \llbracket r_2 \rrbracket$, $\llbracket (r_1 \cdot r_2) \rrbracket = \llbracket r_1 \rrbracket \cdot \llbracket r_2 \rrbracket$ and $\llbracket (r_1)^* \rrbracket = \llbracket r_1 \rrbracket^*$
### Example: Regular expressions and their denotations

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Denotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( a )</td>
<td>{a}</td>
</tr>
<tr>
<td>(( (c \cdot a) \cdot t))</td>
<td>{c \cdot a \cdot t}</td>
</tr>
<tr>
<td>(( (m \cdot e) \cdot (o)^* ) \cdot w )</td>
<td>{mew, meow, meoow, meooow, \ldots}</td>
</tr>
<tr>
<td>((a + (e + (i + (o + u)))))</td>
<td>{a, e, i, o, u}</td>
</tr>
<tr>
<td>(( (a + (e + (i + (o + u))))) )^*</td>
<td>all strings of 0 or more vowels</td>
</tr>
</tbody>
</table>
Given the alphabet of all English letters, $\Sigma = \{a, b, c, \ldots, y, z\}$, the language $\Sigma^*$ is denoted by the regular expression $\Sigma^*$.

The set of all strings which contain a vowel is denoted by $\Sigma^* \cdot (a + e + i + o + u) \cdot \Sigma^*$.

The set of all strings that begin in “un” is denoted by $(un)\Sigma^*$.

The set of strings that end in either “tion” or “sion” is denoted by $\Sigma^* \cdot (s + t) \cdot (ion)$.

Note that all these languages are infinite.
A language is **regular** if it is the denotation of some regular expression.
Not all formal languages are regular.
**Closure properties:**
A class of languages $\mathcal{L}$ is said to be closed under some operation ‘•’ if and only if whenever two languages $L_1, L_2$ are in the class ($L_1, L_2 \in \mathcal{L}$), also the result of performing the operation on the two languages is in this class: $L_1 \bullet L_2 \in \mathcal{L}$.
Properties of regular languages

Regular languages are closed under:

- Union
- Intersection
- Complementation
- Difference
- Concatenation
- Kleene-star
- Substitution and homomorphism
Automata are models of computation: they compute languages. A finite-state automaton is a five-tuple $\langle Q, q_0, \Sigma, \delta, F \rangle$, where $\Sigma$ is a finite set of alphabet symbols, $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final (accepting) states and $\delta : Q \times \Sigma \times Q$ is a relation from states and alphabet symbols to states.
Example: Finite-state automaton

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{c, a, t, r\}$
- $F = \{q_3\}$
- $\delta = \{\langle q_0, c, q_1 \rangle, \langle q_1, a, q_2 \rangle, \langle q_2, t, q_3 \rangle, \langle q_2, r, q_3 \rangle\}$
The reflexive transitive extension of the transition relation $\delta$ is a new relation, $\hat{\delta}$, defined as follows:

- for every state $q \in Q$, $(q, \epsilon, q) \in \hat{\delta}$
- for every string $w \in \Sigma^*$ and letter $a \in \Sigma$, if $(q, w, q') \in \hat{\delta}$ and $(q', a, q'') \in \delta$ then $(q, w \cdot a, q'') \in \hat{\delta}$. 
Example: Paths

For the finite-state automaton:

\[ \delta \] is the following set of triples:

\[
\langle q_0, \epsilon, q_0 \rangle, \langle q_1, \epsilon, q_1 \rangle, \langle q_2, \epsilon, q_2 \rangle, \langle q_3, \epsilon, q_3 \rangle, \\
\langle q_0, c, q_1 \rangle, \langle q_1, a, q_2 \rangle, \langle q_2, t, q_3 \rangle, \langle q_2, r, q_3 \rangle, \\
\langle q_0, ca, q_2 \rangle, \langle q_1, at, q_3 \rangle, \langle q_1, ar, q_3 \rangle, \\
\langle q_0, cat, q_3 \rangle, \langle q_0, car, q_3 \rangle
\]
A string \( w \) is accepted by the automaton \( A = \langle Q, q_0, \Sigma, \delta, F \rangle \) if and only if there exists a state \( q_f \in F \) such that \( (q_0, w, q_f) \in \hat{\delta} \). The language accepted by a finite-state automaton is the set of all the strings it accepts.

**Example: Language**

The language of the finite-state automaton:

![Diagram](image)

is \( \{\text{cat, car}\} \).
Finite-state automata

Example: Some finite-state automata

$q_0$ $\emptyset$
Example: Some finite-state automata

\[
\begin{array}{c}
q_0 \xrightarrow{a} q_1 \\
\end{array}
\]
Example: Some finite-state automata
Finite-state automata

Example: Some finite-state automata

\[ q_0 \xrightarrow{a} q_1 \xrightarrow{a \{a, aa, aaa, aaaa, \ldots\}} \]
Example: Some finite-state automata
Finite-state automata

Example: Some finite-state automata

$q_0 \xrightarrow{?} \Sigma^*$
Finite-state automata

An extension: $\epsilon$-moves.
The transition relation $\delta$ is extended to:

$$\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$$

Example: Automata with $\epsilon$-moves

The language accepted by the following automaton is \{do, undo, done, undone\}:
Theorem (Kleene, 1956): The class of languages recognized by finite-state automata is the class of regular languages.
Example: Finite-state automata and regular expressions

- $\emptyset$
  - $q_0$

- $a$
  - $q_0 \xrightarrow{a} q_1$

- $((c \cdot a) \cdot t)$
  - $q_0 \xrightarrow{c} q_1 \xrightarrow{a} q_2 \xrightarrow{t} q_3$
Finite-state automata

Example: Finite-state automata and regular expressions

\[((m \cdot e) \cdot (o)^*) \cdot w)\\

\(((a + (e + (i + (o + u))))))^*\n
\begin{itemize}
  \item \(q_0\) \(\rightarrow\) \(q_1\) \(m\)
  \item \(q_1\) \(\rightarrow\) \(q_2\) \(e\)
  \item \(q_2\) \(\rightarrow\) \(q_3\) \(w\)
  \item \(q_0\) \(\rightarrow\) \(a, e, i, o, u\)
\end{itemize}
Operations on finite-state automata

- Concatenation
- Union
- Intersection
- Minimization
- Determinization
Minimization and determinization

If $L$ is a regular language then there exists a finite-state automaton $A$ accepting $L$ such that the number of states in $A$ is minimal. $A$ is unique up to isomorphism.

A finite-state automaton is **deterministic** if its transition relation is a function.

If $L$ is a regular language then there exists a deterministic, $\epsilon$-free finite-state automaton which accepts it.
Example: Equivalent automata

A₁

A₂

A₃
Finite-state automata are efficient computational devices for generating regular languages. An equivalent view would be to regard them as recognizing devices: given some automaton $A$ and a word $w$, applying the automaton to the word yields an answer to the question: Is $w$ a member of $L(A)$, the language accepted by the automaton? This reversed view of automata motivates their use for a simple yet necessary application of natural language processing: dictionary lookup.
Applications of finite-state automata in NLP

Example: Dictionaries as finite-state automata

- **go**: go → o → ⊗
- **go, gone, going**: go → o → n → e → ⊗
- **go, gone, going**: go → o → n → e → ⊗
Applications of finite-state automata in NLP

Example: Adding morphological information

Add information about part-of-speech, the number of nouns and the tense of verbs:

\[ \Sigma = \{ a, b, c, \ldots, y, z, -N, -V, -sg, -pl, -inf, -prp, -psp \} \]

![Diagram of finite-state automata with labels for different parts of speech and morphological information.]
The appeal of regular languages for NLP

- Most phonological and morphological process of natural languages can be straight-forwardly described using the operations that regular languages are closed under.
- The closure properties of regular languages naturally support modular development of finite-state grammars.
- Most algorithms on finite-state automata are linear. In particular, the recognition problem is linear.
- Finite-state automata are reversible: they can be used both for analysis and for generation.
While regular expressions are sufficiently expressive for some natural language applications, it is sometimes useful to define relations over two sets of strings.
Regular relations

Part-of-speech tagging:

I know some new tricks
PRON V DET ADJ N

said the Cat in the Hat
V DET N P DET N
Regular relations

Morphological analysis:

<table>
<thead>
<tr>
<th>I</th>
<th>know</th>
<th>some</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-PRON-1-sg</td>
<td>know-V-pres</td>
<td>some-DET-indef</td>
<td>new-ADJ</td>
</tr>
<tr>
<td>tricks</td>
<td>said</td>
<td>the</td>
<td>Cat</td>
</tr>
<tr>
<td>trick-N-pl</td>
<td>say-V-past</td>
<td>the-DET-def</td>
<td>cat-N-sg</td>
</tr>
<tr>
<td>in</td>
<td>the</td>
<td>Hat</td>
<td></td>
</tr>
<tr>
<td>in-P</td>
<td>the-DET-def</td>
<td>hat-N-sg</td>
<td></td>
</tr>
</tbody>
</table>
Regular relations

Singular-to-plural mapping:

- cat  hat  ox  child  mouse  sheep  goose
- cats  hats  oxen  children  mice  sheep  geese
A finite-state transducer is a six-tuple $\langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F \rangle$. Similarly to automata, $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final (or accepting) states, $\Sigma_1$ and $\Sigma_2$ are alphabets: finite sets of symbols, not necessarily disjoint (or different). $\delta : Q \times \Sigma_1 \times \Sigma_2 \times Q$ is a relation from states and pairs of alphabet symbols to states.
Finite-state transducers

Shorthand notation:

Adding $\epsilon$-moves:
A finite-state transducer defines a set of pairs: a binary relation over $\Sigma_1^* \times \Sigma_2^*$. The relation is defined analogously to how the language of an automaton is defined: A pair $\langle w_1, w_2 \rangle$ is accepted by the transducer $A = \langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F \rangle$ if and only if there exists a state $q_f \in F$ such that $(q_0, w_1, w_2, q_f) \in \hat{\delta}$. The transduction of a word $w \in \Sigma_1^*$ is defined as $T(w) = \{ u \mid (q_0, w, u, q_f) \in \hat{\delta} \text{ for some } q_f \in F \}$. 
Finite-state transducers

Example: The uppercase transducer

\[ a : A, b : B, c : C, \ldots \]
Finite-state transducers

Example: English-to-French

\[ c : c \]
\[ d : c \]
\[ o : h \quad g : i \quad \epsilon : e \quad \epsilon : n \]
\[ \epsilon : h \quad a : a \quad t : t \]
Properties of finite-state transducers

Given a transducer \( \langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F \rangle \),

- its **underlying automaton** is \( \langle Q, q_0, \Sigma_1 \times \Sigma_2, \delta', F \rangle \), where
  \((q_1, (a, b), q_2) \in \delta' \) iff \((q_1, a, b, q_2) \in \delta \)

- its **upper automaton** is \( \langle Q, q_0, \Sigma_1, \delta_1, F \rangle \), where
  \((q_1, a, q_2) \in \delta_1 \) iff for some \( b \in \Sigma_2 \), \((q_1, a, b, q_2) \in \delta \)

- its **lower automaton** is \( \langle Q, q_0, \Sigma_2, \delta_2, F \rangle \), where
  \((q_1, b, q_2) \in \delta_2 \) iff for some \( a \in \Sigma_a \), \((q_1, a, b, q_2) \in \delta \)
A transducer $T$ is *functional* if for every $w \in \Sigma_1^*$, $T(w)$ is either empty or a singleton.

Transducers are closed under union: if $T_1$ and $T_2$ are transducers, there exists a transducer $T$ such that for every $w \in \Sigma_1^*$,

$$T(w) = T_1(w) \cup T_2(w).$$

Transducers are closed under inversion: if $T$ is a transducer, there exists a transducer $T^{-1}$ such that for every $w \in \Sigma_1^*$,

$$T^{-1}(w) = \{ u \in \Sigma_2^* \mid w \in T(u) \}.$$

The inverse transducer is $\langle Q, q_0, \Sigma_2, \Sigma_1, \delta^{-1}, F \rangle$, where

$$(q_1, a, b, q_2) \in \delta^{-1} \text{ iff } (q_1, b, a, q_2) \in \delta.$$
Example: Operations on finite-state relations

\[
R_1 = \{ \text{tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, pineapple:Ananas, coconut:Koko} \}
\]

\[
R_2 = \{ \text{grapefruit:pampelmuse, coconut:Kokusnuß} \}
\]

\[
R_1 \cup R_2 = \{ \text{tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, grapefruit:pampelmuse, pineapple:Ananas, coconut:Koko, coconut:Kokusnuß} \}
\]
Transducers are closed under composition: if $T_1$ is a transduction from $\Sigma_1^*$ to $\Sigma_2^*$ and and $T_2$ is a transduction from $\Sigma_2^*$ to $\Sigma_3^*$, then there exists a transducer $T$ such that for every $w \in \Sigma_1^*$, $T(w) = T_2(T_1(w))$.

The number of states in the composition transducer might be $|Q_1 \times Q_2|$. 
Example: Composition of finite-state relations

\[ R_1 = \{ \text{tomato: } \text{Tomate}, \text{cucumber: } \text{Gurke}, \text{grapefruit: } \text{Grapefruit}, \text{grapefruit: } \text{pampelmuse}, \text{pineapple: } \text{Ananas}, \text{coconut: } \text{Koko}, \text{coconut: } \text{Kokusnuß} \} \]

\[ R_2 = \{ \text{tomate: } \text{tomato}, \text{ananas: } \text{pineapple}, \text{pampelmousse: } \text{grapefruit}, \text{concombre: } \text{cucumber}, \text{cornichon: } \text{cucumber}, \text{noix-de-coco: } \text{coconut} \} \]

\[ R_2 \circ R_1 = \{ \text{tomate: } \text{Tomate}, \text{ananas: } \text{Ananas}, \text{pampelmousse: } \text{Grapefruit}, \text{pampelmousse: } \text{Pampelmuse}, \text{concombre: } \text{Gurke}, \text{cornichon: } \text{Gurke}, \text{noix-de-coco: } \text{Koko}, \text{noix-de-coco: } \text{Kokusnuße} \} \]
Properties of finite-state transducers

Transducers are not closed under intersection.

\( T_1 \)

\( T_2 \)

\( T_1(c^n) = \{ a^m b^n \mid m \geq 0 \} \)

\( T_2(c^n) = \{ a^m b^n \mid m \geq 0 \} \Rightarrow \)

\( (T_1 \cap T_2)(c^n) = \{ a^n b^n \} \)

Transducers with no \( \epsilon \)-moves are closed under intersection.
Properties of finite-state transducers

- Computationally efficient
- Denote regular relations
- Closed under concatenation, Kleene-star, union
- Not closed under intersection (and hence complementation)
- Closed under composition
- Weights