Unit 1

Implementing phonology as a list

Linguists distinguish many different aspects of language:

Phonetics the physical properties or speech, in particular the physiological production of sounds (*articulatory phonetics*) and their acoustic properties (*acoustic phonetics*)

Phonology the rules regulating the sounds of a language: which sounds are part of a given language's inventory, in which positions may they occur, how are sounds affected by the presence of other sounds, which syllables in a word are stressed, and so on

Morphology the rules regulating the structure of words, in particular how a words form can change depending on certain features such as person or number (*inflectional morphology*) and how new words can be built from other words (*derivational morphology*)

Syntax the rules regulating the structure of sentences, e.g. word order and morphosyntactic dependencies (person, gender, number agreement)

Semantics the formal study of the meaning of words (*lexical semantics*) and how the logical meaning of a sentence is derived from the meaning of its words (*compositional semantics*)

Pragmatics the study of how a sentence's intended meaning arises from the interaction of its logical meaning and the discourse context

All these areas have been looked at by computational linguists, but we will start with phonology, for several reasons. First, phonetics involves a lot of non-discrete math that is fairly demanding for the uninitiated. Similarly, syntax, semantics and pragmatics use fairly complicated linguistic formalisms. This leaves us with phonology and morphology, and since the two are very similar from a computational perspective (many computational models even handle them both at the same time), we pick the one that linguistics students usually have had greater exposure to: phonology.

1 A simple phonology problem

Word-final devoicing is a process that has been extensively studied by phonologists. As its name implies, final devoicing turns voiced consonants (/b/, /d/, /g/, /z/, ...) that

occur at the end of a word into their voiceless counterparts (/p/, /t/, /k/, /s/, ...). It usually applies only to a proper subclass of a given language's full inventory of voiced sounds. Final devoicing is attested in a rich variety of languages, from Indo-European ones like Catalan (Romance), German (Germanic), and Russian (Slavic) to Turkish (Turkic, Altaic) and Wolof (Senegambian, Niger-Congo).

Language	Voiced	Devoiced
Catalan	*grize* 'gray (F)'	*gris* 'gray (M)'
German	*räder* 'bikes'	*rat* 'bike'
Russian	*kniga* 'book (NOM.SG.)	*knik* 'book (GEN.PL.)'
Turkish	*sarabi* 'wine (ACC.SG.)'	*sarap* 'wine (NOM.SG.)'
Wolof	does anybody know	the data?

What kind of computational resources must a native speaker possess that has successfully learned this process for their language and can apply it correctly during speech? As innocent as this question may seem, it is actually very difficult to answer because it is unclear what kind of computational process underlies word-final devoicing.

The way it was described above — which is the standard view among phonologists — it is a process that takes a word as an input and returns an output where any word-final voiced consonants have been devoiced. That is a complex idea that involves three distinct components:

1. an input form,
2. an output form,
3. a process translating the former into the latter.

It is far from obvious that this is indeed what speakers are doing; all we can tell for sure is that speakers produce the correct output forms. So rather than jumping immediately into the deep waters of sophisticated phonological machinery, let's see what the **simplest empirically adequate** solution might be. It might well turn out that speakers are actually doing something more complicated, but at least we will have a better understanding of the problem and a computational baseline that we can compare speakers' behavior to.

Arguably the simplest conceivable solution is that there are no phonological processes at all, speakers simply memorize all the output forms.

Listedness model of phonology A speaker's knowledge of phonology is fully captured by a list of pronunciations.

This would reduce phonology to a long list of fully inflected words and speakers simply pick that item from the list that they want to pronounce. It explains speakers' ability to produce the correct output form purely via memorization, no computational machinery is required beyond a mechanism for storing and retrieving phonetic strings. Such a solution is certainly simple, and it has been used in NLP in the past. After all it only require a few people to go through a dictionary of English and write down the pronunciation for each word and all its fully inflected variants (e.g. *go*, *goes*, *went*, *gone*, *going*). Simple as it may be, though, the important question for us whether it is empirically adequate.

2 The Listedness model is inadequate

2.1 A failed argument about storage and retrieval

Your first gut reaction may be that the Listedness model cannot possibly be right because it would require storing an enormous amount of information. One can certainly construct a formal argument along those lines, but as we will see it is a weak one because if confuses specification and implementation.

Let's first try to fully spell out why storage might be a problem. Suppose that an English speaker knows around 50,000 words, each one of which has approximately 2 different pronunciation forms. For instance, the verb *go* require memorizing the pronunciations of *go*, *goes*, *went*, *gone*, and *going*, the noun *cow* also has a plural *cows*, and an adjective like *red* only has that one pronounced form. Let us assume furthermore that the average word has 8 sounds, and that there are 64 different sounds — both are vast overestimates. If there are 64 distinct sounds, a single sound takes 6 bits to store. Overall, then, our list would require at least $50,000 \times 2 \times 8 \times 6 = 4,800,000$ bits of storage, which is approximately 585 kilobytes.

Background Binary numbers and bits

In daily life we use the *decimal number system*. In this system, 53 represents a number that we can analyze as $50 + 3 = (5 \times 10) + (3 \times 1) = (5 \times 10^1) + (3 \times 10^0)$. So in the decimal system, the digit in the n-th position acts as a multiplier for 10^{n-1}, and we just sum the values encoded by each position of the number. That's why it is called the decimal system, 10 acts as the base with exponent $n-1$.

This system can of course be used with any other natural number as the base. The simplest case is the *binary number system*, where the base is 2. The number 5, for instance, has the binary representation 101 since $5 = (1 \times 2^2) + (0 \times 2^1) = (1 \times 2^0)$. And 53 is written 110101, whereas 1100001 is 97 (you do the math!). So even though both numbers have two digits in decimal, one has 6 digits in binary and the other 7 digits. A binary digit is also called a *bit*.

Binary numbers are tremendously useful because they are strings of bits, and each bit can have only two values. This means that each bit can be easily instantiated in a physical system, for instance via the on and off states of a switch, or the presence or absence of an electric current. That's exactly what computers do, and that's why computers "only know zeros and ones", as the tired old saying goes. It is also why a single bit represents the smallest unit of information: the most basic distinction one can make is that between being in a specific state on the one hand, and not being in that state on the other hand. This in turn also allows us to calculate memory usage: given a binary encoding of some piece of information, its memory usage equals the number of bits used for the encoding.

In our calculations above, we need 6 bits to encode the 64 phones because that is the smallest number of bits that allows us to make 64 distinctions. With 5 bits, there's only $2^5 = 32$ distinct binary representations, which isn't enough. If the language had 65 phones, we would need 7 bits per phone instead of 6. There are ways to optimize binary encodings so that very frequent phones would have short bit representations than rare ones, but we will not go into this here. We only want a rough approximation of memory usage, and the simple binary encoding explained here is sufficient for that.

Given our (rather pessimistic) assumptions, the Listedness model of English phonology requires barely more than half a megabyte. This isn't all that much in the grand scheme of things. We could even consider more extreme examples. The language Ubykh (extinct, Northwestern Caucasian) has 84 phonemes (≈ underlying sounds). Even if we assume that each phoneme has two pronunciation variants, this leaves us with at most 172 different sounds and hence 8 bits per sound. Keeping all assumptions as before, the Listedness model would consume around 780 kilobytes, which is still less than one megabyte. To put this number into perspective, the collected works of Shakespeare take up about 5 times as much (depending on how they are encoded), and they fit on 1500 densely printed pages. If we put these numbers in direct correlation, we may say that the Listedness model only works if humans can memorize 300 pages of text.

This may seem ludicrous, but human memory is capable of some surprising feats. The Vedic Sanskrit corpus, for instance, consists of over 1000 pages and despite being over 3000 years old, wasn't written down until the first century BC. For over a thousand years, it was preserved by a purely oral tradition that relied exclusively on memorization. That requires a lot of storage capacity. Of course this isn't a perfect comparison: oral traditions rely on elaborate memorization techniques, the memorization is done by adults and not by infants acquiring the language of their environment, and so on. But if memorizing 1000 pages is feasible, it is far from obvious that the Listedness model exceeds human storage capacities.

If you worked through section P.1 on linear and binary search in Chapter 0, then you might have another objection. The longer the list, the longer it should take to retrieve pronunciations, and as speakers with a large vocabulary do not take longer to retrieve pronunciations compared to those with a small vocabulary, the Listedness model is not a plausible model of human phonology. Once again there is a lot one could quibble with here, but let us get straight to the main problem with this argument as well as the previous one: the Listedness model is a claim about how the speaker's knowledge is specified, not how it is implemented.

The Listedness model does not claim that humans have a literal list of pronunciations in their head. There's many different ways a list can be implemented, be it a singly-linked list, a doubly-linked list, a hash table, or a prefix tree (see XXX). Some of them circumvent all the problems of storage and retrieval pointed out above. Even if they didn't, there might be some neural data structure that computer scientists are unaware of that allows humans to store very long lists in a very efficient manner. Criticizing the Listedness model because lists would be a suboptimal data structure is a category mistake. The model is meant as a high-level description of phonological knowledge, not how this knowledge is stored and queried in the human mind. If we want to argue against the Listedness model, we have to show that its notion of phonological knowledge is inadequate.

2.2 Phonological knowledge is more than memorization

The Listedness model claims that phonology is just a list of licit output forms that the speaker has memorized over the years. If this is correct, then several properties should hold of the phonological systems we find in natural languages:

- **Free typology**
  Since there are no restrictions on what items may occur in a list, any given list of output forms is a possible phonological system.

- **Defeat by the unknown**
  When presented with a new word, speakers cannot find this word in their phonology list and thus do not know how to pronounce it.

- **Isolationism**
  The pronunciation of a word is always the same and does not alter depending on the preceding and following words in a sentence.

- **Finiteness**
  If phonology is simply a list that keeps track of the pronunciation of words a speaker has encountered so far, then this list must be finite — nobody has heard an infinite number of words at any given point in their life. Consequently, a speaker's phonological system contains only a finite number of words.

- **Egalitarianism**
  All phonological forms are equally easy to learn because they simply involve memorization.

These predictions are maximally wrong. There isn't a single known language where at least one of those properties holds.

Contrary to *Free typology*, languages are not only systematic, they are systematic in very specific ways. There's an infinite number of easily formulated principles that could be expressed in terms of a list yet are never found in any natural language. One might be "if the number of vowels in the word is a multiple of 3, then all vowels must be a", or "a word contains more than four phones only if it is a palindrome", or "if each consonant counts for 1 point and each vowel doubles the point total of the preceding consonants, then every word must be worth at least 17 points". We could even have conditions that reference the length of the list, e.g. "if the list has less than 20,000 words, then all vowels must be a". We can write lists that obey these highly unnatural constraints (or their opposite, or arbitrary combinations thereof), and these lists are predicted to be possible phonological systems that we should find in some language.

Defeat by the unknown is also untenable. Native speakers do have intuitions about the pronunciation of nonce words, and this has also been verified in a myriad of experiments using *wug* tests. In a *wug* test, the test subject is presented with a made-up word and asked to determine its pronunciation. The experimenter may show the subject a drawing of a bird and tell them that this bird is called a [wʌg]. The experimenter then reveals a picture with two wugs and asks the test subject to complete the sentence "On this picture, there are two…". A native speaker of English will realize that the plural *wugs* is pronounced [wʌgz] rather than [wʌgs] because the voicing of the plural morpheme is dependent on the preceding sound. A similar experiment could be performed to test German speaker's knowledge of final devoicing: the subject is first told the plural [vagə], from which they should correctly infer that the singular is not [vag] but [vak].

Isolationism does not hold, either. For example, *phone bill* is often pronounced like *foam bill* in rapid speech, and *white board* may actually sound like *wipe board*. Here the pronunciation of the last sound of the first word is partially assimilated to the first sound of the following word: the alveolar nasal [n] is changed to the bilabial nasal [m] because [b] is bilabial, while the alveolar plosive [t] becomes the bilabial plosive [p] in this context. The only way this could be handled in the list model is by listing sequences of words, rather than just individual words. But this would only exacerbate

the other problems. Such a model could, for example, require that the number of vowels in a word must exceed the number of vowels in the previous word. The only way to address Isolationism in the Listedness model is to make it perform even worse with respect to the other properties.

Finiteness is a more subtle point because it goes beyond the pure empirical facts and invokes an additional methodological principle. Take a simple expression like *great grandfather*. Using various linguistic tests one can show that it is not a phrase. It is usually assumed that if a multi-word expression isn't a phrase, it must be a compound, and compounds are still words as far as phonology is concerned. Hence the phonology list should include an entry for *great grandfather*. But then it must also contain an entry *great great grandfather*, *great great great grandfather*, and so on. In general, native speakers know exactly how to pronounce *great*^{n} *grandfather*, even for very high values of n. Now, strictly speaking, we cannot rule out that there is a finite cut-off point for n — say, 3 trillion — where native speakers suddenly don't know how to pronounce *great*^{n} *grandfather*. There is no way to test this experimentally. If such a cutoff point exists, though, it must be fairly large. But the larger the cutoff point, the larger the list and the harder it is to work with and reason over. It is more convenient and insightful to assume that there is no finite cut-off point if that allows for a more compact and succinct description.

This leaves us with *egalitarianism*, which is at odds with language change. For example, *biopic* (= biographic picture) used to be pronounced with primary stress on the *i*, similar to *bioinformatics* or *bio-degradable*. In recent years, however, more and more speakers have switched to a pronunciation with primary stress on the *o*, apparently in an analogy to *myopic*. If phonology is just a memorized list, it is unclear what the motivation for this change could be, any form is as good as the next.

In a nutshell, the problem with the Listedness approach is that it fails to recognize the systematic nature of phonology. Any random, haphazard assortment of pronunciations should be a possible phonological system, but instead we find that phonology is largely driven by generalizations. Generalizations about what is a possible constraint on pronunciations, generalizations to nonce forms that have never been heard before, generalizations that apply across multiple words in a sentence, generalizations that may even apply to an infinite number of words that will never be uttered, generalizations that eradicate existing outliers to make phonology even more systematic. Once we contrast phonology against all logically conceivable options, it reveals itself to be driven by a narrowly circumscribed set of generalization mechanisms.

Playing devil's advocate, one may propose that this propensity for generalization need not be explained by the Listedness model because they could be due to the learning algorithm that produces the list of pronuncations. We do not see free typological variation because the learning algorithm will only entertain specific ways of inferring a phonology list from the input data. The phonology list need not be able to determine new pronunciations because the learning algorithm can take care of that based on some probabilistic metrics. Finiteness is not an issue because the learning algorithm can dynamically add new words to the list whenever they are needed. And egalitarianism does not hold because the learning algorithm can be structured in a way such that certain pronunciations are more preferable to others, e.g. by reducing the amount of idiosyncratic information that cannot be derived from more basic principles.

But the position of the devil's advocate is tantamount to admitting defeat. The goal was to present a maximally simple model of phonology. Combining a simple model of phonology with a complex learning model for phonology does not magically yield a

simple model. The complexity is still there, but it has been pushed into the learner instead of the grammar. That's just window dressing. The bottom line is that we need a more complicated model than what we started out with in this section. For this purpose it does not matter whether we put the complexity in the grammar formalism or the learner; at this point, we do not even know whether there is a meaningful difference between the two. Rather than complicating the picture by having both a grammar and learner, we should stick with one component. And since it makes no sense to have a learning algorithm without a grammar that ends up being learned, we will stick with grammars. Not much has changed, then: the simplest conceivable model has failed to capture the systematicity of phonology, so we have to find something slightly more complex that allows for phonological generalizations.

3 Summary

3.1 Key insights for Unit 1

- Phonology is the rule system that controls the sounds of a natural language.
- The Listedness model claims that a speaker's phonological knowledge can be adequately described as a finite list of licit, memorized pronunciations.
- A naive implementation of the model in terms of lists might not be cognitively plausible. But this is not a valid counterargument because the model's task is to describe phonological knowledge, not how this knowledge is actually encoded in the human mind. Alternative data structures are available that address many of the shortcomings of lists.
- The main problem of the Listedness model is that cannot explain why phonology is so systematic and ripe with generalizations.

3.2 Relevant literature for Unit 1

add more info: phonology survey papers and textbooks; argument about finiteness;

Experiments have shown that frequent words are retrieved more quickly than rarer ones, but the retrieval speed is independent of overall vocabulary size (see Jurafsky 2003 and references therein). Having a larger vocabulary does not mean that it takes you longer to retrieve specific words.

This is a very simplified presentation of hash tables, see Sec. 1.5 of Dasgupta, Papadimitriou & Vazirani (2006) for some very useful details.

P Programming

P.1 Different types of lists

As mentioned in section 2.1, the Listedness model does not propose a concrete implementation of the purported list of pronunciations. In fact, even if one were to interpret it literally as postulating some kind of mental list of pronunciations, that would leave a lot of wiggle room for different implementations because there are many different types of lists.

Two of the most basic types of lists are *singly-linked lists* and *doubly-linked lists*, and those two differ in several respects. A singly-linked list is an unordered collection of data points where each data point also contains a reference to the data point that is next in the list. Yes, you read that right, a list starts out as unordered and obtains its order only through additional references that link those data points together. A concrete example in the form of a Python dictionary might make this clearer.

```
# a dictionary that implements a singly-linked list
singly_list = {
    'start': 0,
    0: {'data': 'This',
        'next': 3},
    1: {'data': 'end',
        'next': None},
    2: {'data': 'the',
        'next': 1},
    3: {'data': 'is',
        'next': 2},
}

# let's print the list from beginning to end
current = singly_list.get('start')
while current is not None:
    # print current data point, then move on to next
    print(singly_list[current].get('data'))
    current = singly_list[current].get('next')

>>> 'This'
>>> 'is'
>>> 'the'
>>> 'end'
```

While it is tempting to interpret the keys 0, 1, 2 and 3 as encoding the linear order of the list, that's not quite right. If we ordered the data points based on their key from smallest to largest, we would get *This end the is*, but the intended order is *This is the end*. The list order arises indirectly from the use of the *next* field. If we had used symbols like • and □ for the keys instead of numbers, the linear order of the items in the list would still be the same. The linear order of a singly-linked list arises only from the *next* field, and nothing else.

In a doubly-linked list, the *next* field is supplemented by a *previous* field.

```
# a dictionary that implements a doubly-linked list
doubly_list = {
    'start': 0,
    'end': 1,
    0: {'data': 'This',
        'next': 3,
        'previous': None},
    1: {'data': 'end',
        'next': None,
        'previous': 2},
    2: {'data': 'the',
        'next': 1,
        'previous': 3},
```

```
    3: {'data': 'is',
        'next': 2,
        'previous': 0},
}

# let's print the list from beginning to end
current = doubly_list.get('start')
while current is not None:
    # print current data point, then move on to next
    print(doubly_list[current].get('data'))
    current = doubly_list[current].get('next')

# and then from the end to the beginning
current = doubly_list.get('end')
while current is not None:
    # print current data point, then move on to next
    print(doubly_list[current].get('data'))
    current = doubly_list[current].get('previous')

>>> 'This'
>>> 'is'
>>> 'the'
>>> 'end'
>>> 'end'
>>> 'the'
>>> 'is'
>>> 'This'
```

While it looks like doubly-linked lists don't add much to singly-linked ones, they actually have very different properties. Suppose that we want to insert *gruesome* before *end* (key 1). In a doubly linked list, this is a quick procedure because we know that the data point before *end* is *the* (key 2). So we create a new key 4 to hold *gruesome* and insert it between *the* and *end*:

1. point *next* for *the* at *gruesome* (key 4),
2. point *previous* for *gruesome* at *the* (key 2),
3. point *next* for *gruesome* at *end* (key 1),
4. point *previous* for *end* at *gruesome* (key 4).

No matter how long the doubly-linked list is, inserting a new element always takes four steps and hence *constant time*.

Now contrast to a singly linked list. Here the procedure is exactly the same, except that we have no easy way of telling which data point occurs before *end*. As a result we have to search through the entire list until we find a data point whose *next* value points at *end*. In the worst case scenario, finding this item takes $n-1$ steps in a list with n items. The cost of insertion grows linearly with the length of the list. One also says that inserting an item into a linked list takes *linear time*. In mathematical notation, inserting an item runs in $O(1)$ for doubly-linked lists but $O(n)$ for singly-linked lists.

Background Asymptotic notation

Computer scientists measure performance in terms of *asymptotic worst-case complexity*. That is to say, how long does it take to solve a given task if everything goes wrong that could go wrong within the specified parameters of the problem (e.g. the item we are looking for is at the very end of the list) and if there are no restrictions on the size of the input (e.g. there is no limit on how many items a list may contain).

Asymptotic worst-case complexity is expressed via the "Big O" notation. If an algorithm has time complexity $O(n^3)$, this means that it takes at most n^3 steps for it to find a solution (or determine that there is no solution), where n is the size of the input problem (e.g. the number of items in a list). The Big O notation ignores parameters whose impact on complexity diminishes as the size of the problem approaches infinity. For instance, when running a search algorithm you wrote in Python there is a fixed cost for reading in the actual code you wrote. But since this only takes a fixed amount of steps, say 5, that cost is completely negligible once we deal with lists containing millions of items — it does not matter whether your computer has to do 3 million computations or 3 million and 5.

Without going into too much detail, we can rank an algorithm's efficiency according to which of the following classes it belongs to:

constant solving the problem always takes a fixed number of steps
e.g. $O(5) = O(1)$

logarithmic the amount of time is logarithmically bounded by the size of the input
e.g. $O(2\log n + 5) = O(2\log n) = O(\log n)$

linear the amount of time scales linearly with the size of the input
e.g. $O(5n\log n + 5) = O(5n) = O(n)$

polynomial the amount of time is bounded by a polynomial of the input (in simpler terms: a function where n has an exponent)
e.g. $O(8n^4 + 3n^2 + 5n\log n + 5) = O(8n^4) = O(n^4)$

exponential the amount of time grows exponentially with input size
e.g. $O(4^n + 2^n + 8n^4 + 3n^2 + 5n\log n + 5) = O(4^n)$

factorial the amount of time grows factorially with the input, where the factorial of n is $n! := n \times (n-1) \times \cdots \times 2 \times 1$ (for instance, $4! = 4 \times 3 \times 2 \times 1$)
e.g. $O(8n! \times n^{10^{100}}) = O(8n!) = O(n!)$

In practice, problems that can be solved in polynomial time are considered efficiently solvable. In the worst case, you may have to wait a few years for hardware to catch up, but since computing power doubles approximately every two years (*Moore's law*), you will eventually catch up with the problem. The difficulty of exponential problems, on the other hand, grows so fast with input size that it vastly outpaces technological progress and how many resources you can throw at it.

This short excursion into data structures shows that there is uniform notion of lists when it comes to implementations. The list in the Listedness model is just a convenient abstraction and simplification with no commitments about how such a list might be implemented. Criticizing it for its storage cost or retrieval speed misses the point.

In Marrian terms, the Listedness model operates at the computational level, not the algorithmic one, so we can't use algorithmic concerns to debunk it. And as we will see in the next section, there are efficient list implementations that sidestep the storage and retrieval issues.

By the way, if you're wondering whether Python's list are singly-linked or doubly-linked, the answer is neither: Python's lists are implemented as *dynamic arrays*, which is yet another data structure with very specific advantages and disadvantages. The possibilities truly are endless.

P.2 Hash tables for efficient retrieval

Quick access and storage of information is crucial in various applications, and thus it is no surprise that computer scientists have developed data structures which allow any given item to be stored and retrieved in constant time. A particularly popular one is the *hash table* or *hash map*.

The basic idea behind a hash table is that the problem with lists is the arbitrary and volatile connection between an item in a list and its key. Consider the example of a Python list below (remember, Python lists are dynamic arrays and hence have nothing to do with the singly-linked or doubly-linked lists we just encountered). The example uses the sci.lang ASCII transliteration of IPA, which is easier to type and can be used on systems without support for Unicode.

```
german_phonology = ['Ra:t', 'Re:dV"', 'blint', 'blind@',\
                    'iC', 'du:', 'EV"', 'si:', 'Es']
```

As you probably know, elements of a Python lists can be retrieved instantaneously if their position in the list is already known.

```
# this list look-up runs in constant time
german_phonology[4]
>>> 'iC'
```

But if you do not know the index of an item, you have to search through the list to find the item, and that is slow — linear search takes linear time, whereas binary search runs in logarithmic time but only works for sorted lists. If there were a way to compute the position of an item without searching through the list, we could always use indices to retrieve items from a Python list. This would allow constant time retrieval for all items, and that is exactly what hash tables were designed for.

In a hash table, each entry has a key that uniquely identifies it, and this key is translated into the correct index via a *hash function*. So in a sense a hash table is a list that is equipped with a mechanism for quickly computing every item's position in the list without ever searching through the list. If item i has key k, then the hash function will convert k to the position where i occurs in the underlying list and retrieve i from this position. A smart hash function can do this conversion from keys to positions in constant time for any given key irrespective of how many keys there are or how long they are — the conversion from indices to keys always takes a fixed number of steps. It follows that lookup in hash table takes constant time, much more efficient than the linear time lookup in a list.

Hash tables are ubiquitous in modern programming. Python dictionaries are an instance of hash tables. Python sets are a subtype of hash tables where every item is also its own key. But what is the cognitive status of hash tables? Would the Listedness model be more plausible if it used a hash table instead of a list?

Our understanding of the human mind is too limited to say for sure. As lookup in hash tables takes constant time instead of linear time, speakers with larger vocabularies would no longer be predicted to have slower retrieval speeds. But a hash table might be too efficient in light of the psycholinguistic reality that not all words are equally easy to retrieve. It is conceivable, though, that humans use a hash function that prioritizes quick retrieval of common items to the detriment of less frequent ones. After all, a hash function that takes one step on very frequent items and five steps on rare ones might be preferable to one that takes 3 steps for all of them. From the perspective of computational complexity the two are the same because $O(1) = O(3) = O(5)$, but that does not preclude that one is noticeably faster than the other in practice, at least when run on *wetware*, i.e. the human brain. Or maybe frequent words are stored in a faster type of memory (just like you might have your OS installed on an SSD while keeping your media files on a conventional hard drive). Or maybe the memory architecture of the human brain is so fundamentally different from anything else we know that the very notion of hash tables does not make any sense. Once again the level of algorithmic implementation furnishes so many knobs and parameters to tune that it is difficult to draw any firm conclusions either way.

P.3 Prefix trees for compact storage

The cognitive argument against the Listedness model also pointed out the storage needs of large lists. But this does not take into account that many items in the list would look very similar, which allows us to represent the list more compactly as a *prefix tree*. For the sake of simplicity, the following discussion will use English orthography instead of phonetic transcriptions.

Consider two English words like *ban* and *banana*. The latter is an extension of the former, or the other way round, *ban* is a *prefix* of *banana*. And *ban* is also a prefix of *bandana*, making it a common prefix of *banana* and *bandana*. This usage of prefix has nothing to do with its linguistic meaning as an affix that precedes the stem of a word. Given a string w, any initial substring of w is a prefix of w, including the empty string ε and w itself.

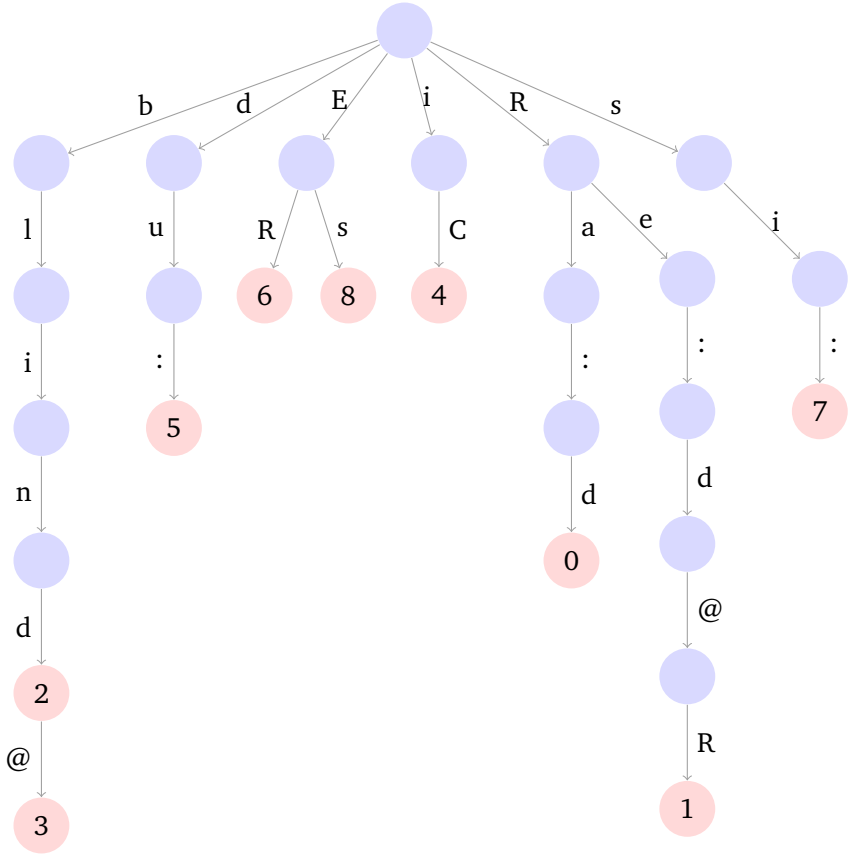
Example 1.1 Prefixes of strings

All of the following are prefixes of *band*:

1. ε,
2. *b*,
3. *ba*,
4. *ban*,
5. *band*

But *bard* is not a prefix of *band*.

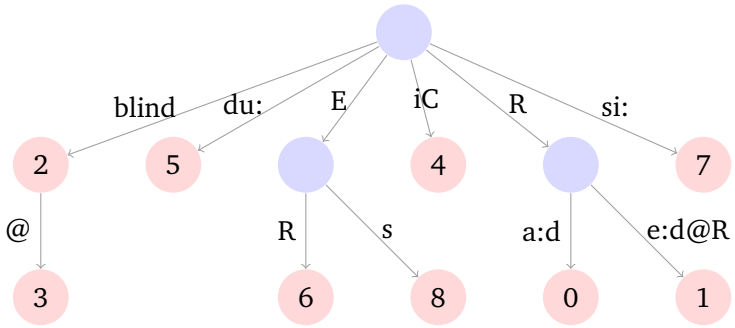
Large lists where many items share a common prefix can be represented more efficiently via a *prefix tree* (also called *trie*). fixme: change tree



Blue circles indicate non-final nodes, whereas the red square represent final nodes. Every path from the root down to final node corresponds to a specific item in the list. For instance, the red node labeled 2 corresponds to *blind* as the branches along this path spell out b-l-i-n-d. If we move one branch further down we get the item *blind*@. Notice that the cost of storing both *blind* and *blind*@ is almost exactly the same as storing both of them thanks to the large amounts of structure sharing. In many cases, adding a new item may only require changing a node from non-final to final, which only requires changing a single bit from 0 to 1 and comes with no extra memory cost.

Prefix trees also offer fast retrieval without the need for a hash function. In order to determine if some item i is in the list, one does not need to search through the entire tree. Instead, it is sufficient to check if the path described by i ends in a final node. This takes as many steps as there are symbols in i. Technically this makes the look-up linear time. But the look-up time scales linearly with the length of the item i, and the items in a list tend to be much shorter than the list itself. So short, in fact, that look-up is effectively done in constant time.

Prefix trees are a wonderful data structure for lists where many items share common prefixes. For sparse lists with very little overlap between items, they might not be worth the memory overhead of labeled branches, nodes, and which nodes are final. We can somewhat mitigate the problem by truncating sequences of unary branches into a single branch, yielding a *compacted prefix tree* (also known as *radix trie*).



For vocabulary lists, which tend to have lots of common prefixes between items, prefix trees combine efficient storage with fast search. Hence it is not true that the Listedness model necessarily commits us to unrealistic assumptions about human memory capacity and retrieval speeds. But there is also precious little evidence that the human brain uses anything resembling prefix trees. As has been stated multiple times before in this chapter, such algorithmic issues are very hard to evaluate, in particular compared to the questions that arise at the computational level. The latter provides much more solid footing, so we would do well to stick with it as much as possible.