

Searching

Terence Parr
University of San Francisco

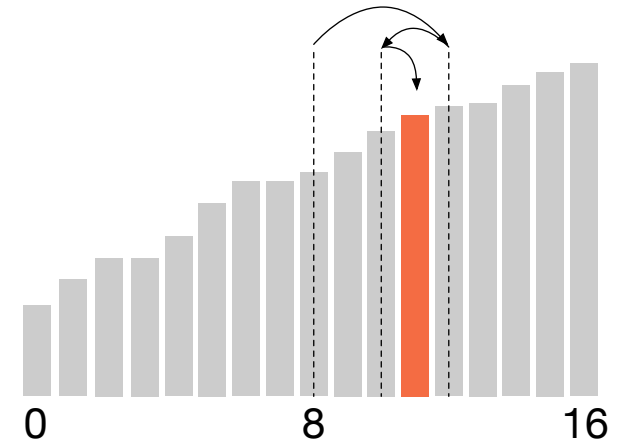
Common searching/membership strategies

- $O(mn)$ • **linear**: scan data structure looking for element(s)
- $O(m\log(n))$ • **binary search**: if in array and sorted, split recursively in half
- $O(m\log(n))$ • **binary search tree**: subtree to left has elements less than current node and subtree to right has elements greater than
- $O(m)$ • **hash table**: function maps key to bucket, linear search in bucket; recall search index project from MSDS692; for word search, not arbitrary string search in document(s)
- $O(m)$ • **state machines** (graphs)

Binary search (review sort of)

- If we know data is sorted, we can search much faster than linearly
- Means we don't have to examine every element even worst-case

```
def binsearch(a,x):  
    left = 0; right = len(a)-1  
    while left<=right:  
        mid = (left + right)//2  
        if a[mid]==x: return mid  
        if x < a[mid]: right = mid-1  
        else: left = mid+1  
    return -1
```



Compare to (tail-)recursive version

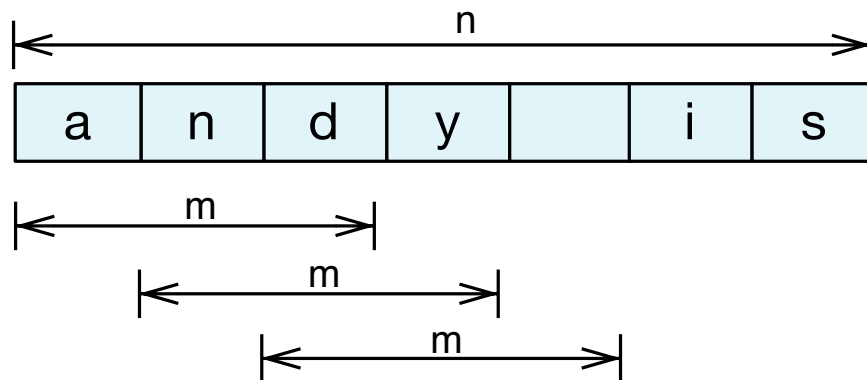
```
def binsearch(a,x,left,right):  
    if left > right: return -1  
    mid = (left + right)//2  
    if a[mid]==x: return mid  
    if x < a[mid]:  
        return binsearch(a,x,left,mid-1)  
    else:  
        return binsearch(a,x,mid+1,right)
```

```
left = 0; right = len(a)-1  
while left<=right:  
    mid = (left + right)//2  
    if a[mid]==x: return mid  
    if x < a[mid]: right = mid-1  
    else: left = mid+1
```

← Bracket region with element

String matching

- **Problem:** Given a document of length n characters and a string of length m , find an occurrence or all occurrences
- Brute force algorithm is $O(nm)$
- Theoretical best-case algorithm exists for $O(n + m)$
- **Exercise:** Describe brute force algorithm; why is it "slow"?



Hash searches

- First, note that two equal strings have same hash code so we can compare int codes quickly even for huge strings
- Rabin-Karp* algorithm uses hash function to speed up but is still $O(nm)$ worst-case; works for any substring not just words
- **Idea:** h = hash search string s ; compute hash for $\text{doc}[i:i+m]$ and compare to h ; if same, compare s to $\text{doc}[i:i+m]$, return if found; move i from 0 to $n-m$
- Key is to avoid comparing strings unless the hash codes match, but usually hash computation costs same as comparing strings

*https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm

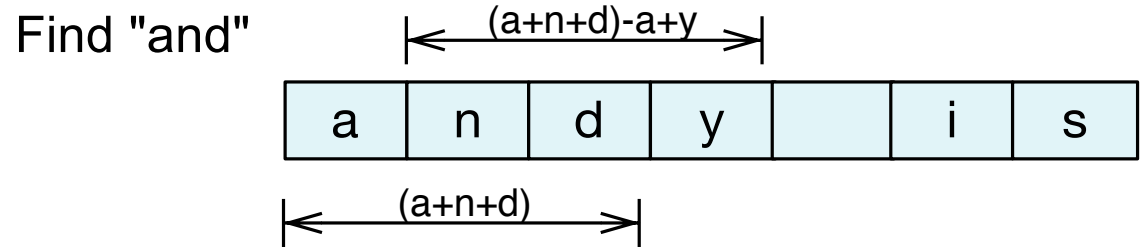
Rabin-Karp (almost)

```
def search(doc, s) -> int:
    n = len(doc); m = len(s)
    hs = hash(s)
    for i in range(0, n-m+1):
        hdoc = hash(doc[i:i+m]) # slow O(m)
        if hdoc==hs: # fast
            if s==doc[i:i+m]: # slow
                return i
    return -1
```

```
def hash(s:str)->int:
    return sum(ord(c) for c in s)
```

Additive hashcode is important here

More details



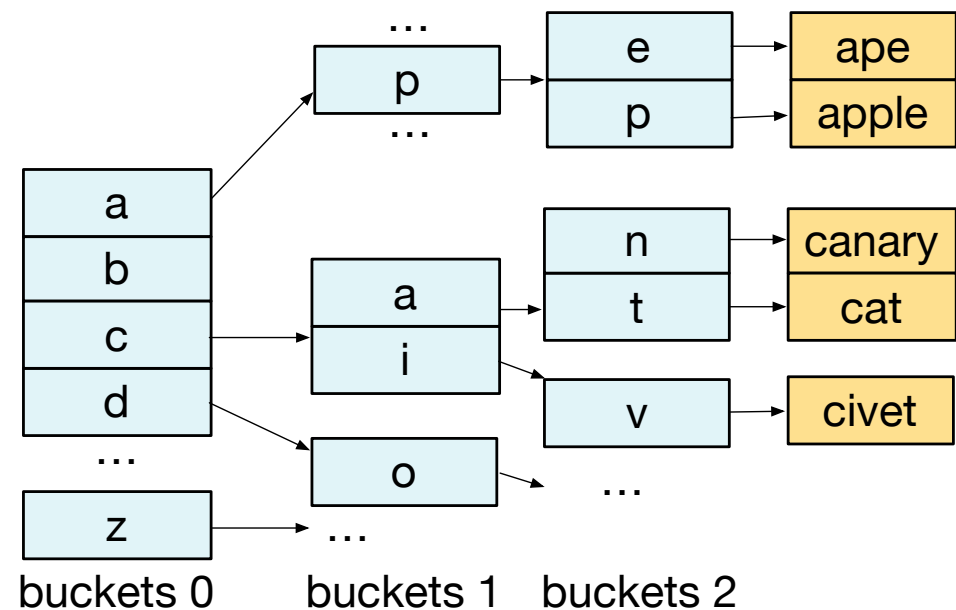
- Naïve $\text{hash}(\text{doc}[i:i+m])$ is $O(m)$ for each $i=1..n$, so use rolling hash to reuse partial hash function computations:
 - next hash is old hash minus $\text{doc}[i]$ plus $\text{doc}[i+m]$
 - drop old one off, add in new char (see improved `search()` in notebook):
 $\text{hdoc} = \text{hdoc} - \text{ord}(\text{doc}[i]) + \text{ord}(\text{doc}[i+m])$ # roll it!
- What about finding all occurrences of s in doc ?
- Can check for k strings as we go along not just 1 using $O(1)$ hashtable for each of k strings
- Algorithm is $O(nm)$ since a weak hash function could cause us to compare s at each position

Is this the best we can do?

- Can we do better than this $O(nm)$ or even $O(n+m)$ algorithms?
- Yes, if we prepare a proper side data structure beforehand once for $O(n)$, and we search for words instead of arbitrary strings. **How?**
- First, consider a hash table, which is $O(1)$ for n words, But, relies on good hash function for good distribution and we still must search buckets of average size k ; that means $O(1)$ is really hiding $O(mk)$
- If we are counting string compares not chars then $O(mk) = O(1)$
- Constant on that complexity can be kind of high
- I claim we can search for any string in doc in $O(m)$; how is this possible?!

Revisit recursive bucket sort

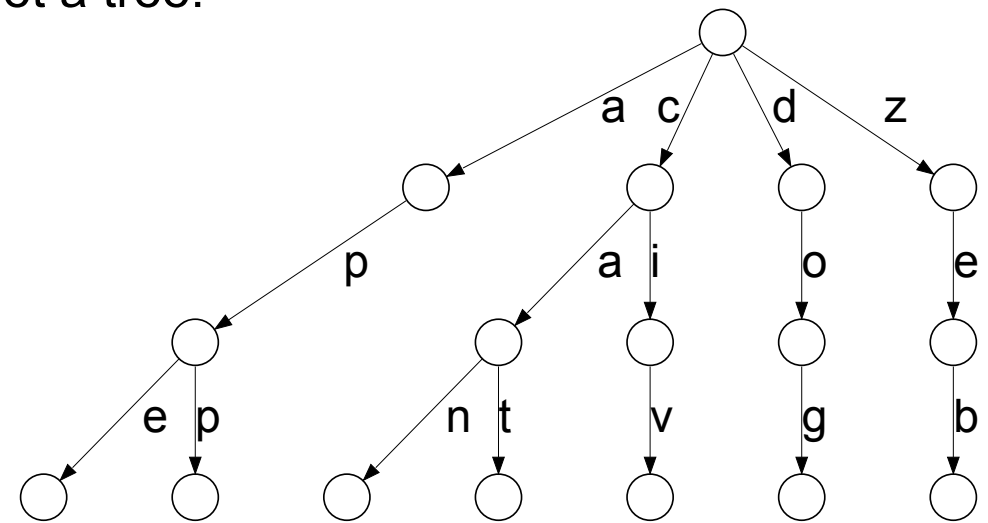
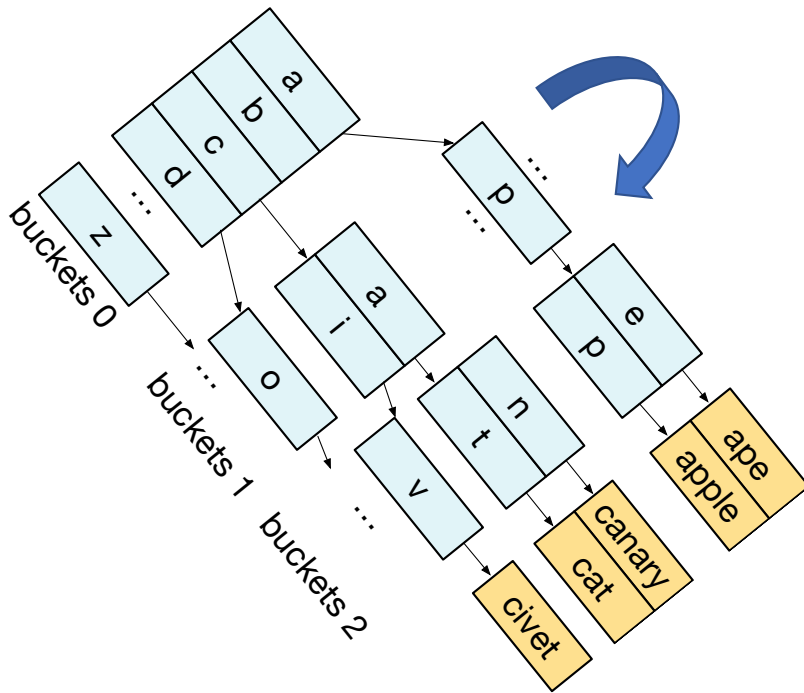
- Break up doc into words, make nested bucket structure as we saw before
- Add deeper buckets if buckets get too big
- To find word s , use $s[i]$ to navigate and find final “leaf” with list of words w/same prefix
- The index says how to navigate
- How long does it take to find s for $n=\text{len}(\text{doc})$, $m=\text{len}(s)$, $k=\text{average bucket size}$?
 $T(n,m,k) = m + k * \text{avgwordsize}$



Can we do better than that?

Introducing “*Tries*” or *Prefix Trees*

First step, convert buckets to nodes and rotate:
we get a tree!



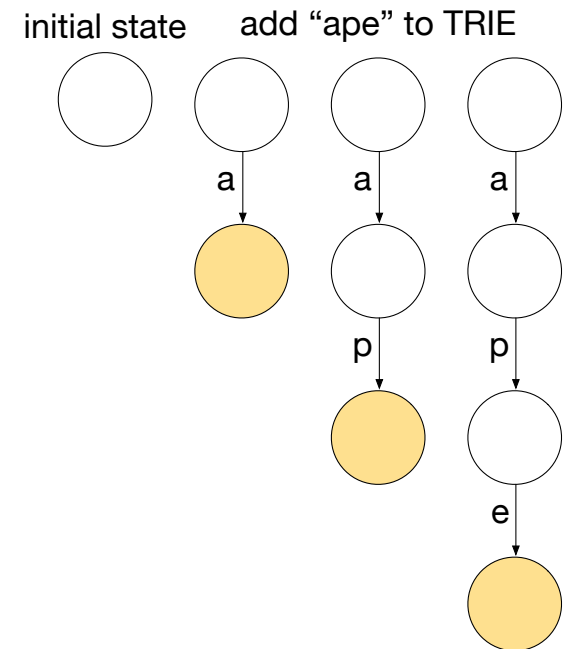
Words are edge labels on path from root to leaves

(was in a “big internet company” interview)

Adding string s to TRIE

- TRIE can hold a big set of words and we can search for a word superfast
- Like bucket sort but add nested buckets for entire length of each string: pigeonhole!
- Note: We're not sorting so order of edges is not important; can use dict()
- Starting at the root, add edge labeled with $s[0]$ pointing to new node
- Traverse edge $\text{root.edges}[s[0]]$ to child and add subtree for $s[1:]$ to that child
- Recurse until out of chars in string s
- Adding one s is $O(m)$ since we must add edge for each char

```
class TrieNode:  
    def __init__(self):  
        self.edges = {}
```

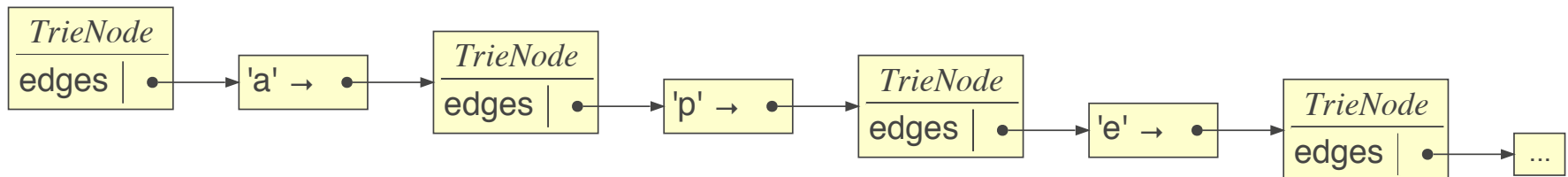


Implementation

```
class TrieNode:  
    def __init__(self):  
        self.edges = {}
```

- add(root, "ape")

```
def add(p:TrieNode, s:str, i=0) -> None:  
    if i>=len(s): return  
    if s[i] not in p.edges:  
        p.edges[s[i]] = TrieNode()  
    add(p.edges[s[i]], s, i+1)
```



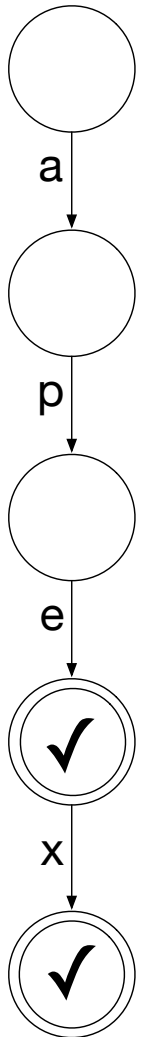
Note that nodes have no values, edges contain the letters

Words that are prefixes of other words

- What about when we have two words “**ape**” and “**apex**”?
- “ape” stops before being a leaf, so we must mark as accept state, which is sometimes called a stop state

```
class TrieNode:  
    def __init__(self):  
        self.isword = False # set to true if accept state  
        self.edges = {}
```

```
def add(p:TrieNode, s:str, i=0) -> None:  
    if i>=len(s): p.isword=True; return  
    if s[i] not in p.edges:  
        p.edges[s[i]] = TrieNode()  
    add(p.edges[s[i]], s, i+1)
```



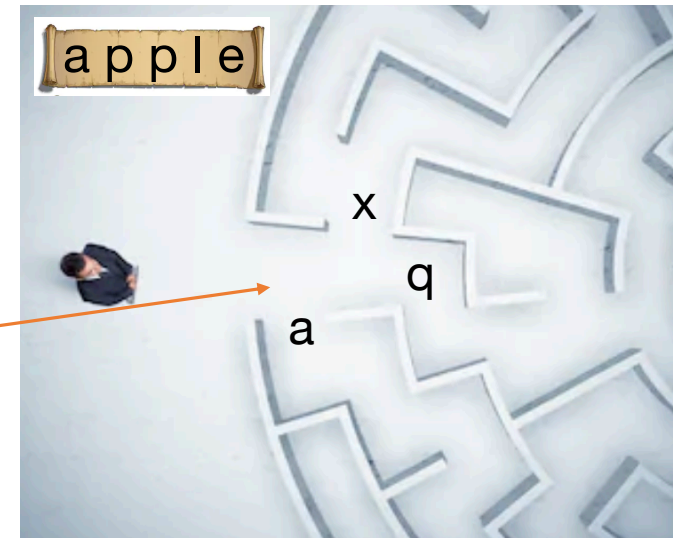
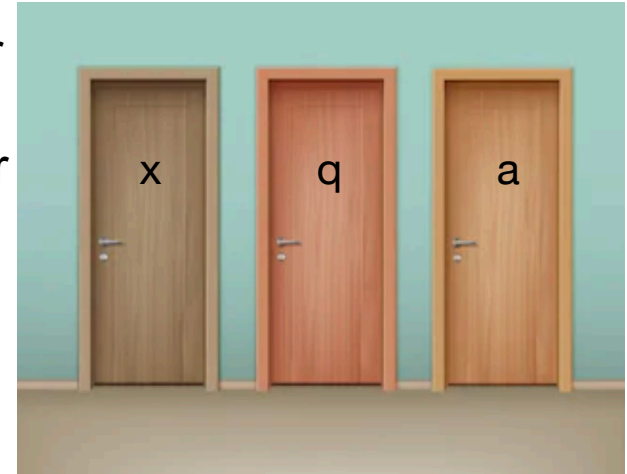
Searching a Trie

(with analogies)

- Return true if s is prefix of word in Trie or full word in Trie
- Note that the search depends on len(s) NOT num words n in the vocabulary!!!

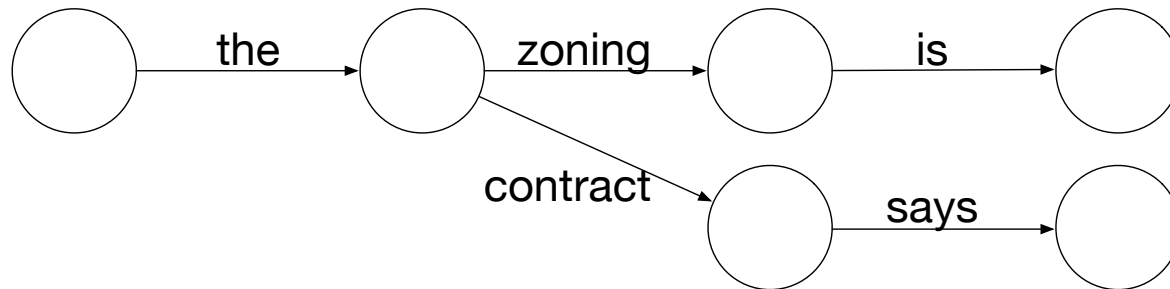
```
def search(root:TrieNode, s:str, i=0) -> bool:
    p = root
    while p is not None:
        if i>=len(s): return True
        if s[i] not in p.edges: return False
        p = p.edges[s[i]]
        i += 1
    return True
```

choose door
based upon
current letter



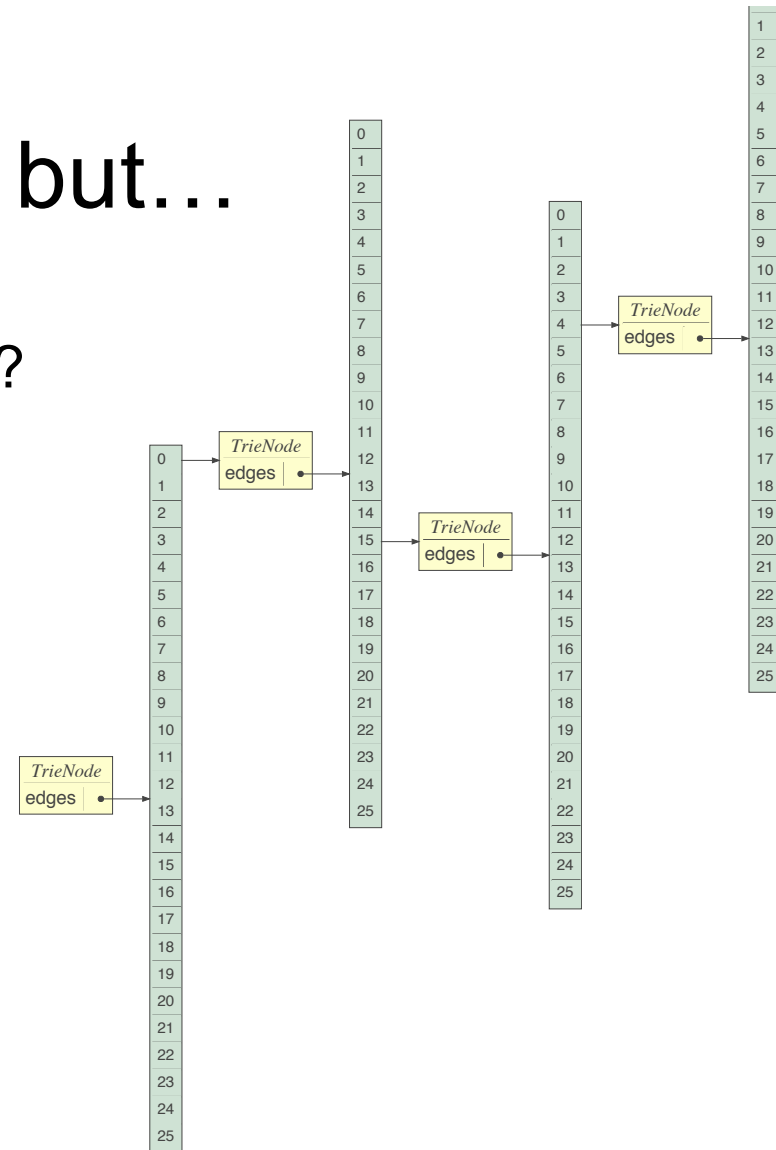
Can search for word sequences too

- TRIE remembers set of sentences not words, in this case
- Tokenize document into words then add sentence sequence to TRIE or just bigrams, trigrams etc...



Edge dictionaries are $O(1)$ but...

- `self.edges = {}` using general hashtable;
can we do faster version of `self.edges['x']`?
- Yes, use array access via perfect hash function $f(c) = \text{ord}(c) - \text{ord}('a')$
- But we use 26 slots even for one edge
- How can we reduce memory costs?
 - Many nodes will have just one outgoing edge so we can optimize for that case with single pointer instead of an array
 - Switch to 26-element edge array if we need more than one edge



Exercise: Brute force dictionary search

- Load words from **/usr/share/dict/words** file (one per line) into list
- Search for each word in list of words; what is complexity?
- This takes almost 5 minutes on my fast computer. Ugh
- For 50k words, takes 13s (still brutally slow)

A
a
aa
aal
aalii
aam
Aani
aardvark
aardwolf
Aaron
Aaronic
Aaronical
...

Exercise: Build Trie from dictionary of words

- From searching notebook, get Trie implementation
- Add each word to a trie, which takes about 7s on my machine
- Search the trie for each of 235,886 words; takes 0.70s for me!!
- Rejoice in your new super powers
- Cool interview question/task:
How can you do fast spell checking on big documents?

Exercise: find all words starting with prefix

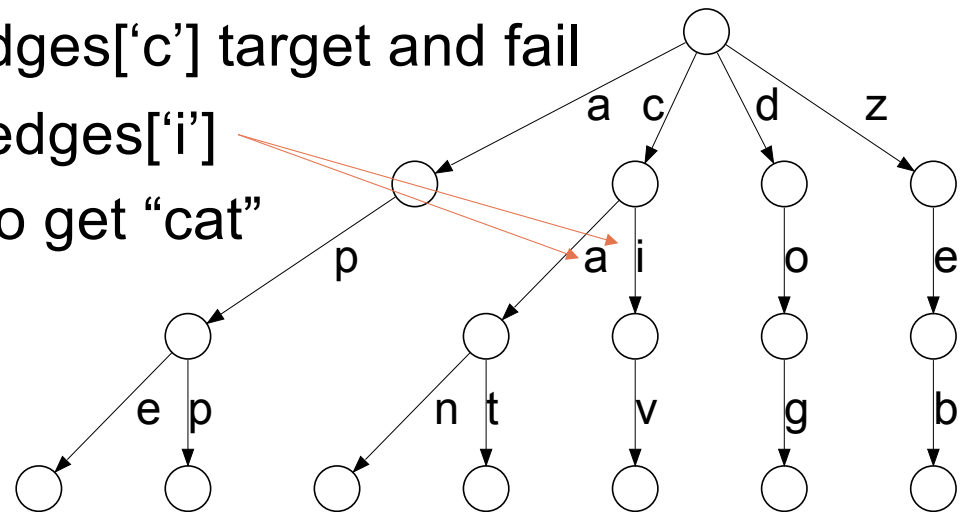
- Create a trie again from the word list
- Write a function that prints all words in trie that begin with a specific prefix like “app”; it should get “apple”, “application”, ...
- How would this work
- Trace prefix into trie, reaching specific non-leaf node p; find all reachable leaves; track string as recursion parameter for each path; print the string when you reach a leaf

Exercise: How to build a suffix tree?

- Simple: create trie from reversed strings or modify add() method to walk backwards through string

Exercise: Given misspelled words off by 1 letter only, find all possible words

- Trace word into trie until no edge exists for $s[i]$; this is node p
- Get list of words reachable from each node targeted by p starting with $s[i+1]$
- E.g., “cxt” would get to $p = \text{root.edges}['c']$ target and fail
- Find “t” from $p.\text{edges}['a']$ and $p.\text{edges}['i']$
- We find only “t” matches via ‘a’ to get “cat”



Summary

- Lots of ways to search beyond linear and binary search
- String searching has some really efficient solutions such as Rabin-Karp; idea is to compare hash codes before doing string comparisons and do a rolling hash for the document substrings
- If we are willing to build a graph data structure in $O(nm)$, the TRIE is pretty hard to beat complexity and performance; looking up a word in the TRIE is $O(m)$ for m -character string!
- TRIE is just a nested pigeonhole sort turned into a graph
- Useful as prefix and suffix trees; can find misspelled words