COMP2521 24T1
Tries

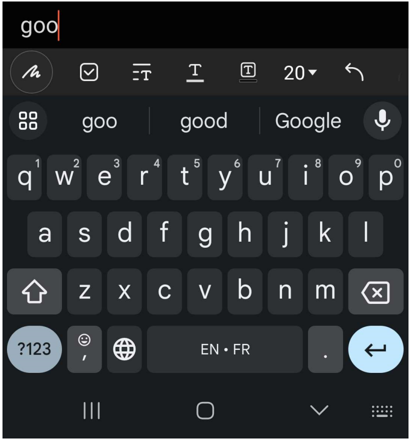
Kevin Luxa
cs2521@cse.unsw.edu.au

Motivation

Many applications require searching through a set of strings with a *pattern*

Examples:
Autocomplete
Predictive text
Approximate string matching
Spell checking

Autocomplete



Predictive text

For example, pressing “4663”
can be interpreted as the word
good, *home*, *hood* or *hoof*

Motivation

How can we implement a set of strings using data structures covered so far?

AVL tree
Performance: $O(\log n)$ worst case

Hash table
Performance: $O(1)$ average case

Motivation

AVL trees and hash tables are efficient, but...

...they are not efficient when searching for a *pattern*

Possible solution: tries

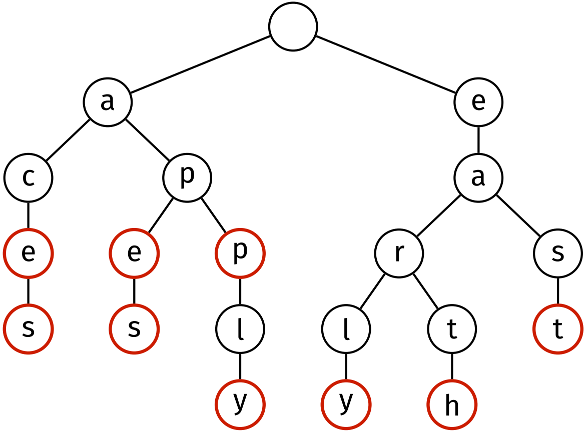
Tries

A trie...

- is a tree data structure
- used to represent a set of strings
  - e.g., all the distinct words in a document, a dictionary, etc.
  - we will call these strings *keys* or *words*
- supports string matching queries in $O(m)$ time
  - where m is the length of the string being searched for

Note: the word *trie* comes from *retrieval*, but pronounced as "try" not "tree"

Example:



Keys in the trie:
ace
aces
ape
apes
app
apply
early
earth
east

Tries

Important features of tries:

- Each link represents an individual character
- A key is represented by a path in the trie
- Each node can be tagged as a “finishing” node
  - A “finishing” node marks the end of a key
- Each node may contain data associated with key
- Unlike a search tree, the nodes in a trie do not store their associated key
  - Instead, keys are implicitly defined by their position in the trie

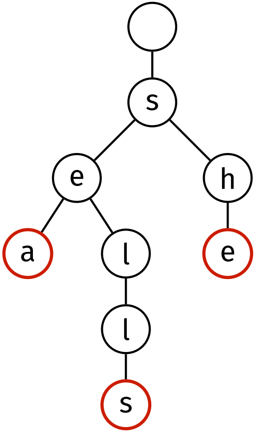
Representation

Assuming alphabetic strings:

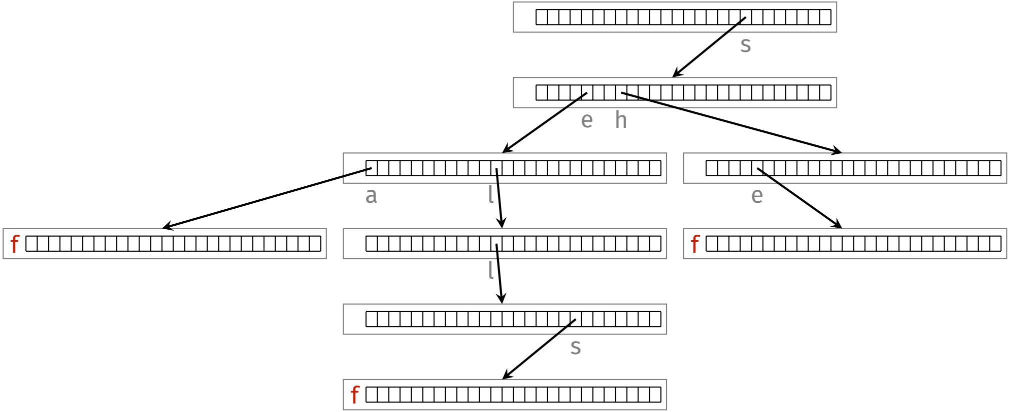
```
#define ALPHABET_SIZE 26

struct node {
    struct node *children[ALPHABET_SIZE];
    bool finish; // marks the end of a key
    Data data;   // data associated with key
};
```

Consider this trie:



Concrete representation:
(f = finishing node)



Trie Insertion

Process for insertion:

- Start at the root
- For each character c in the key (from left to right):
  - If there is no child node corresponding to c, create one
  - Descend into the child node corresponding to c
- Mark the resulting node as a finishing node and insert data (if any)

Trie Insertion

Example

Insert the following words into an initially empty trie:

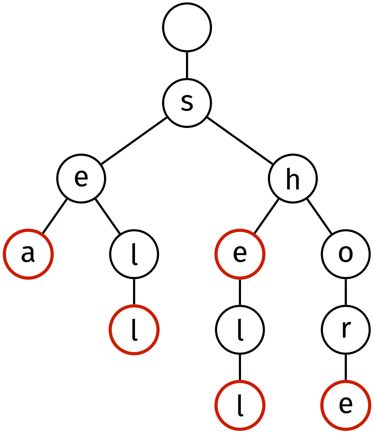
sea shell sell shore she

Trie Insertion

Example

Insert the following words into an initially empty trie:

sea shell sell shore she



Recursive method:

```
trieInsert(t, key, data):
    Input:  trie t
            key of length m and associated data
    Output: t with key and data inserted

    if t is empty:
        t = new node

    if m = 0:
        t->finish = true
        t->data = data
    else:
        first = key[0]
        rest = key[1..m - 1] // i.e., slice off first character from key
        t->children[first] = trieInsert(t->children[first], rest, data)

    return t
```

EXERCISE Try writing an iterative version.

Trie Search

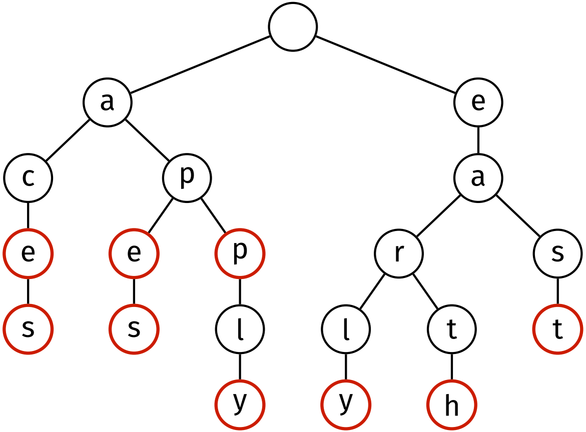
Search is similar to insertion:

- Start at the root
- For each character c in the key (from left to right):
  - If there is no child node corresponding to c, return false
  - Descend into the child node corresponding to c
- If the resulting node is a finishing node, then return true, otherwise return false

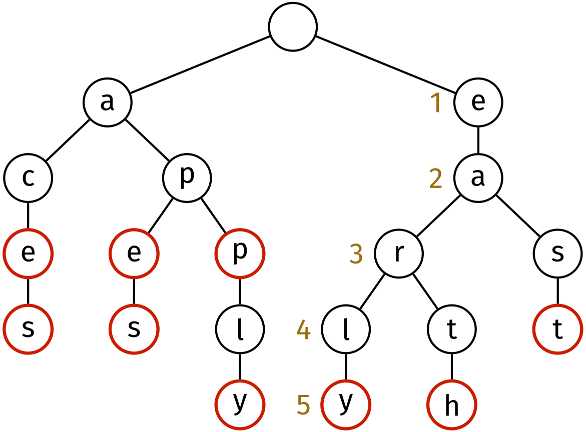
Trie Search

Example

Search for “early”



Search for “early”

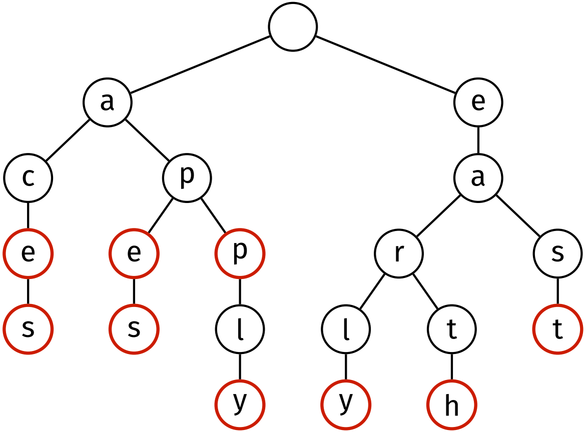


Found!

Trie Search

Example

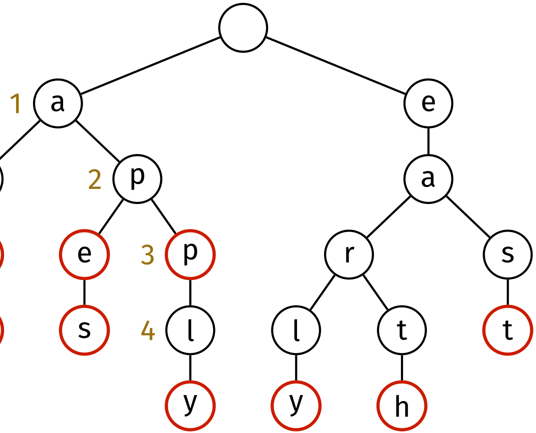
Search for “apple”



Trie Search

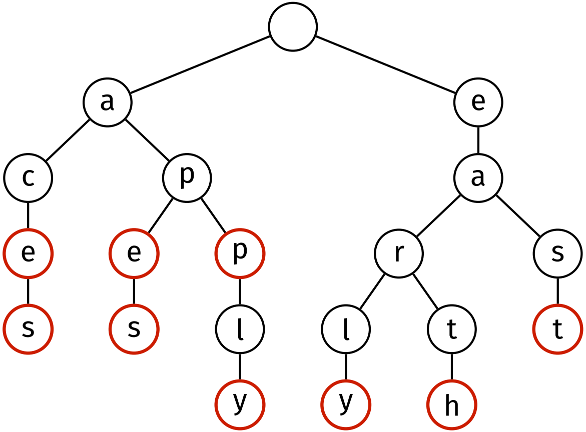
Example

Search for “apple”



Not found - node for “appl” has no child node for ‘e’

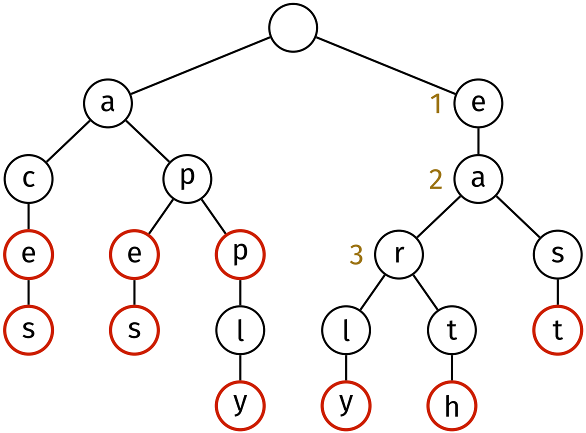
Search for “ear”



Trie Search

Example

Search for “ear”



Not found - node for “ear” is not a finishing node

Recursive method:

```
trieSearch(t, key):
    Input:  trie t
            key of length m
    Output: true if key is in t
            false otherwise

    if t is empty:
        return false
    else if m = 0:
        return t->finish = true
    else:
        first = key[0]
        rest = key[1..m - 1]
        return trieSearch(t->children[first], rest)
```

EXERCISE Try writing an iterative version.

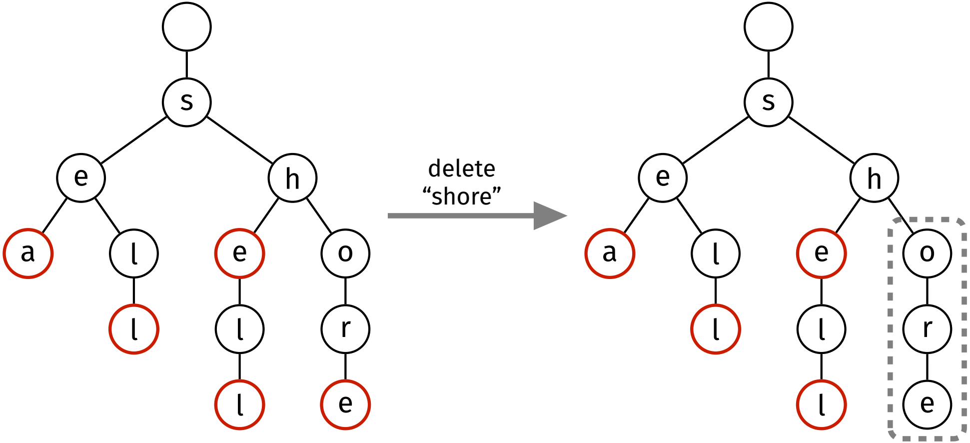
Trie Deletion

Deletion is trickier...

- Can simply find node corresponding to given key and mark it as a non-finishing node
- ...but this can leave behind dead branches
  - i.e., branches that don't contain any finishing nodes
  - dead branches waste memory

Trie Deletion

Example of dead branch:



Trie Deletion

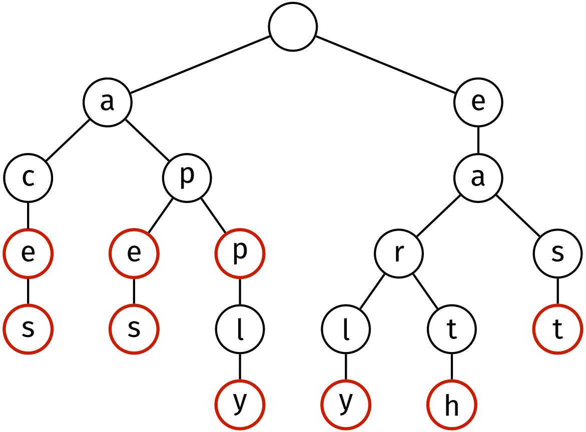
Process for deletion:

- Find node corresponding to given key
  - If node doesn't exist, do nothing
- Mark the node as a non-finishing node
- While current node is not a finishing node and has no child nodes:
  - Delete current node and move up to parent
    - Handled recursively

Trie Deletion

Example

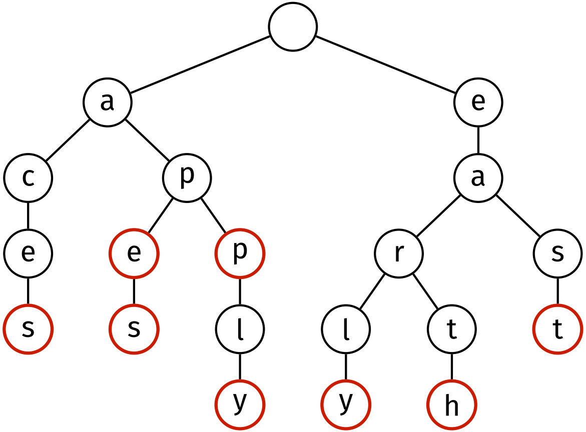
Delete "ace"



Trie Deletion

Example

Delete “ace”

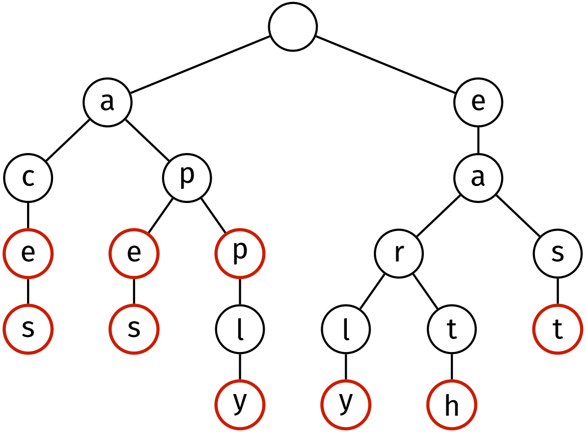


Deleted - node for “ace” is no longer marked as a finishing node

Trie Deletion

Example

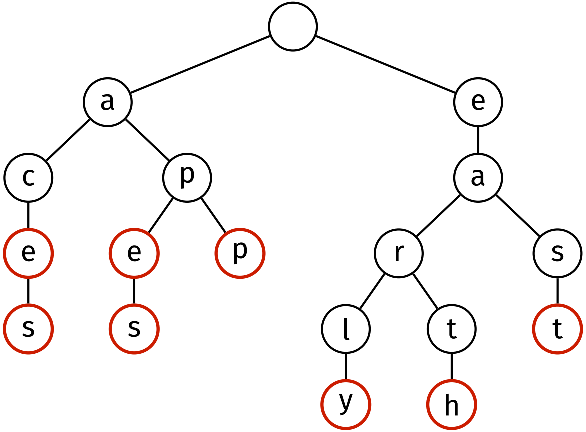
Delete “apply”



Trie Deletion

Example

Delete “apply”

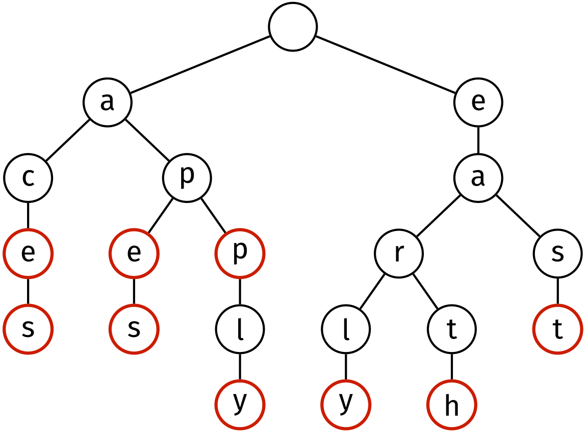


Deleted - deleted nodes corresponding to “apply” and “appl”

Trie Deletion

Example

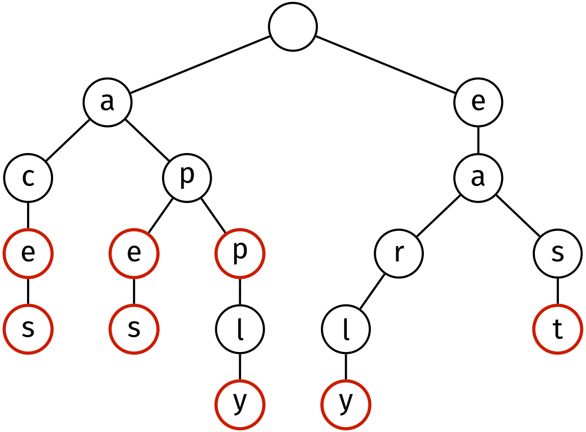
Delete "earth"



Trie Deletion

Example

Delete "earth"



Deleted - deleted nodes corresponding to "earth" and "eart"

Recursive method:

```
trieDelete(t, key):
    Input:  trie t
            key of length m
    Output: t with key deleted

    if t is empty:
        return t
    else if m = 0:
        t->finish = false
    else:
        first = key[0]
        rest = key[1..m - 1]
        t->children[first] = trieDelete(t->children[first], rest)

    if t->finish = false and t has no child nodes:
        return NULL
    else:
        return t
```

Analysis

Analysis of standard trie:

- $O(m)$ insertion, search and deletion
    - where m is the length of the given key
    - each of these needs to examine at most m nodes
- $O(nR)$ space
    - where n is the total number of characters in all keys
    - where R is the size of the underlying alphabet (e.g., 26)

Variants

Simple trie representation consumes an enormous amount of memory

- Each node contains `ALPHABET_SIZE` pointers
  - If keys are alphabetic, then this is 26 pointers...
    - ...which is $8 \times 26 = 208$ bytes on an 64-bit machine!
  - If keys can contain any ASCII character, then this is 128 pointers!
- Even if trie contains many keys, most child pointers will be unused

Variants

Different representations exist to reduce memory usage at the cost of increased running time:

- Use a singly linked list to store child nodes
- Alphabet reduction - break each character into smaller chunks, and treat these chunks as the characters

Variants

Linked list of children

One technique to reduce memory usage:

Have each node store a linked list of its children instead of an array of `ALPHABET_SIZE` pointers

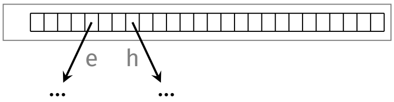
Variants

Linked list of children

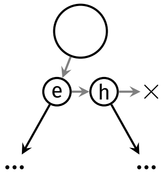
```
struct node {
    struct child *children;
    bool finish;
    Data data;
};

struct child {
    char c;
    struct node *node;
    struct child *next;
};
```

Instead of:



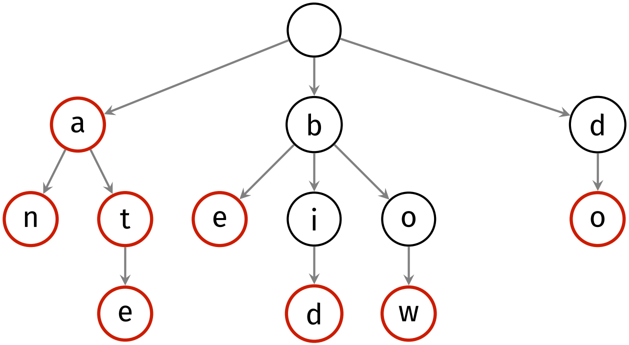
We have:



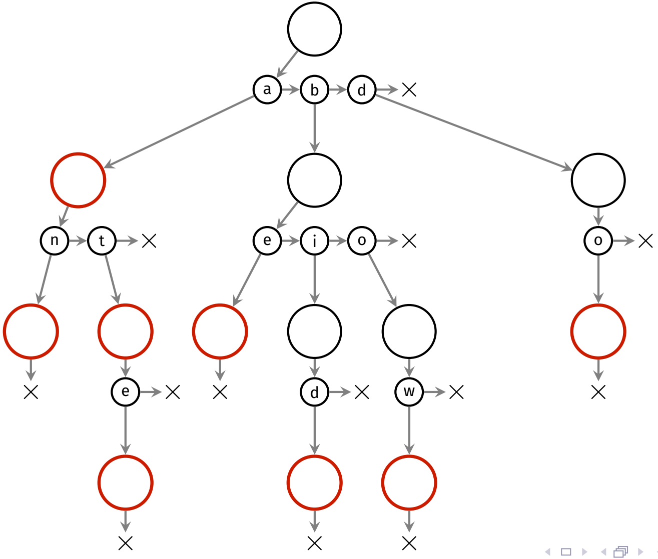
Variants

Linked list of children

Consider the following trie:



Its concrete representation:



Variants

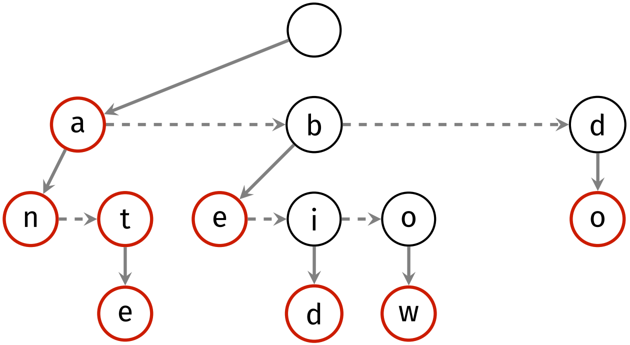
Left-child right-sibling binary tree

We can simplify this representation
by merging each linked list node with its corresponding trie node

This produces the left-child right-sibling binary tree representation

```
struct node {
    char c;
    struct node *children;
    struct node *sibling;
    bool finish;
    Data data;
};
```

Concrete representation of above trie:



Analysis:

- This representation uses much less space
  - Each node just stores one extra pointer to its sibling instead of `ALPHABET_SIZE` pointers
- But this is at the expense of running time
  - Need to traverse up to `ALPHABET_SIZE` nodes before reaching desired child

Variants

Alphabet reduction

Another technique to reduce memory usage:
alphabet reduction

Break each 8-bit character into two 4-bit nybbles

This reduces the branching factor,
i.e., the number of pointers in each node

For example, the word “sea” consists of the following bytes:

s	e	a
01110011	01100101	01100001

We break it into 4-bit nybbles like so:

s		e		a	
01110011		01100101		01100001	
0111	0011	0110	0101	0110	0001

Instead of storing the word “sea”, we now insert the following word:

`0111 0011 0110 0101 0110 0001`

Analysis:

- This representation uses much less space
    - Much fewer pointers per node
- But this is at the expense of running time
    - Path to each key is twice as long - lookups need to visit twice as many nodes

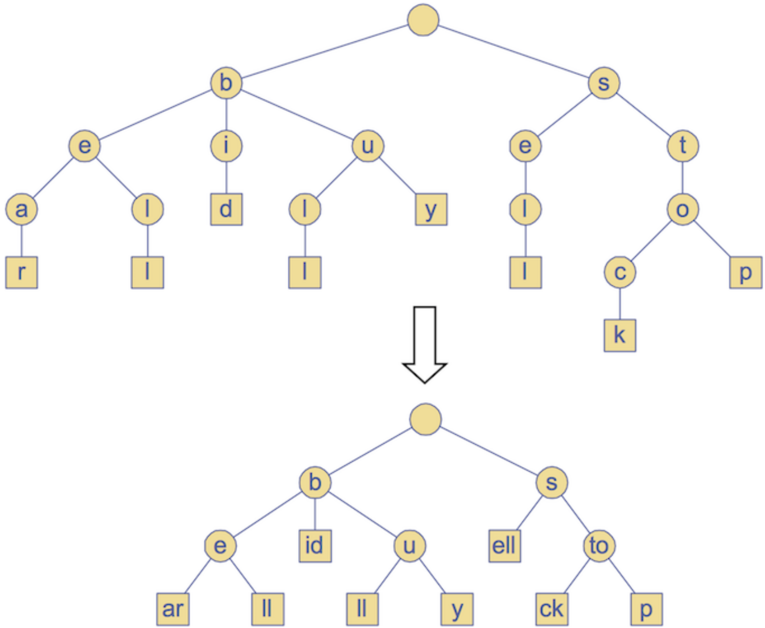
Another technique to reduce memory usage:
use a compressed trie

In a compressed trie, each node contains ≥ 1 character

Obtained by merging non-branching chains of nodes

Specifically, non-finishing nodes with only one child are merged with their child

Variants
Compressed tries



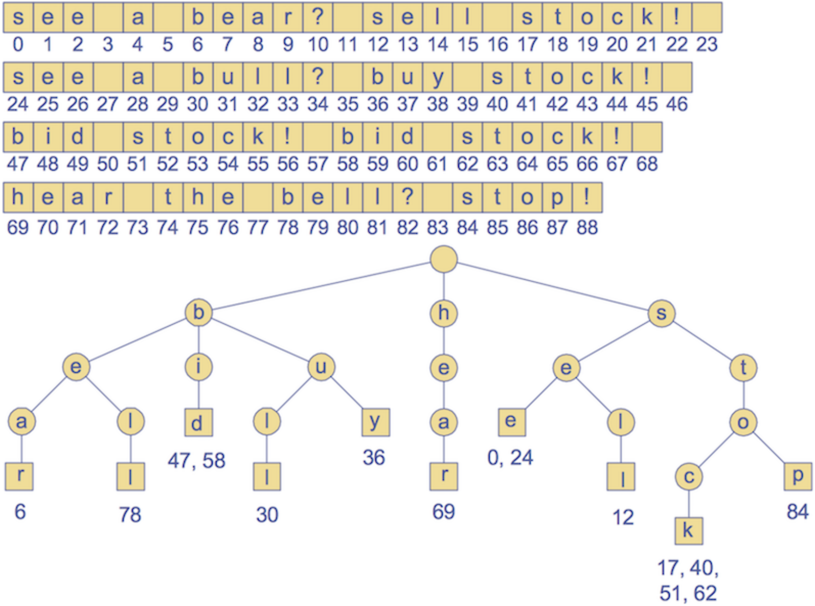
Idea:

Given a document, preprocess it
by storing all words in a trie,
and for each word, store the location of all its occurrences

When user searches for a word,
can query the trie instead of scanning entire document

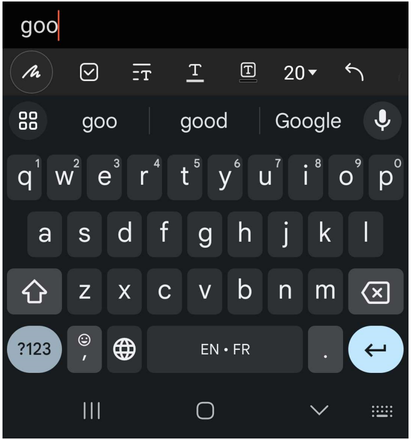
Applications

Word finding



Autocomplete

Given a series of letters,
find all words that start with it



Predictive text

Given a series of button presses (e.g., on a keypad), where each button can represent multiple letters, find all possible matching words

Feedback

https://forms.office.com/r/5c0fb4tvMb

Appendix

Trie Insertion Example

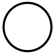
Insert the following words into an initially empty trie:

sea shell sell shore she

Trie Insertion Example

Insert the following words into an initially empty trie:

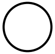
sea shell sell shore she



Trie Insertion Example

Insert the following words into an initially empty trie:

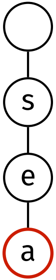
sea shell sell shore she



Trie Insertion Example

Insert the following words into an initially empty trie:

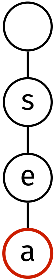
sea shell sell shore she



Trie Insertion Example

Insert the following words into an initially empty trie:

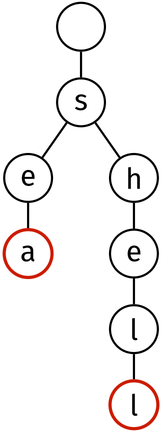
sea shell sell shore she



Trie Insertion Example

Insert the following words into an initially empty trie:

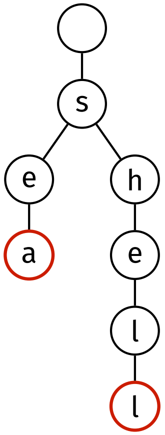
sea shell sell shore she



Trie Insertion Example

Insert the following words into an initially empty trie:

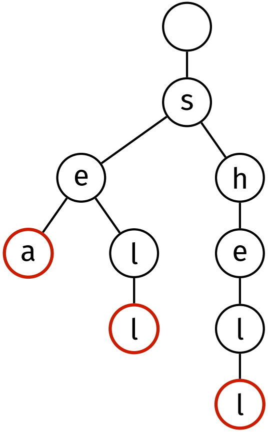
sea shell sell shore she



Trie Insertion Example

Insert the following words into an initially empty trie:

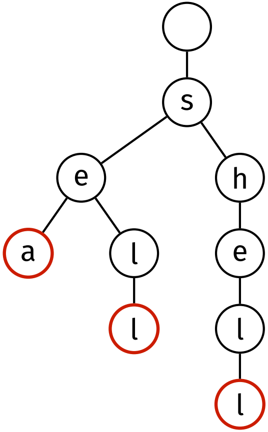
sea shell sell shore she



Trie Insertion Example

Insert the following words into an initially empty trie:

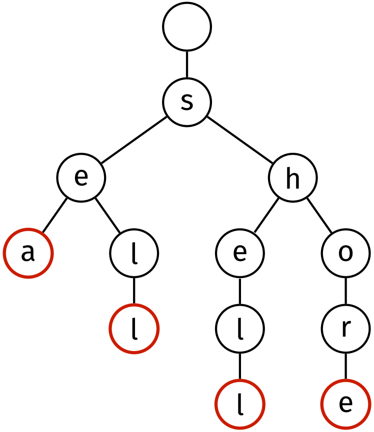
sea shell sell shore she



Trie Insertion Example

Insert the following words into an initially empty trie:

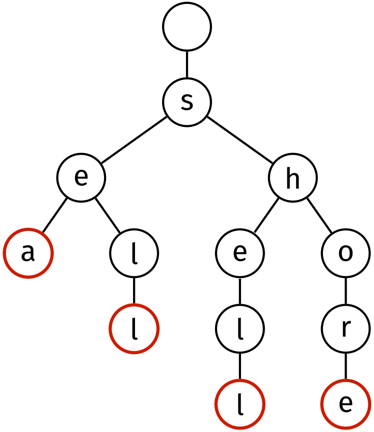
sea shell sell shore she



Trie Insertion Example

Insert the following words into an initially empty trie:

sea shell sell shore she



Trie Insertion Example

Insert the following words into an initially empty trie:

sea shell sell shore she