Lexicon case study

- Special case of Set/Map
  - Keys always strings, in fact, English words
  - No associated value
- Core operations
  - add
  - containsWord
  - containsPrefix

Lexicon as sorted vector

```cpp
private:
    Vector<string> words;
    ... abacus abate ...
    ... zygote
```

containsWord?
- Binary search using ==, O(lgN) time
containsPrefix?
- Binary search using substr ==, O(lgN)
add?
- Binary search to find position, shuffle to insert, O(N)
space usage?
  - Per-entry = sizeof(string) ~ length*sizeof(char)
  - ~8 bytes per word (assuming words average len 8 chars)

Lexicon as binary search tree

```cpp
private:
    struct node {
        string word;
        node *left, *right;
    };
    node *root;
containsWord?
- Tree search using ==, O(lgN) time
containsPrefix?
- Tree search using substr ==, O(lgN)
add?
- Tree search to find position, insert new leaf, O(lgN)
space usage?
  - Per-entry = string + left/right ptrs + balance factor
  - ~18 bytes per word

Lexicon as hash table

```cpp
private:
    struct cell{
        string word;
        cell *next;
    };
    cell *buckets[NBuckets];
```

containsWord?
- Hash to bucket, search bucket, O(N/B)
containsPrefix?
- Search all buckets, O(N)
add?
- Hash to bucket, search/add to bucket, O(N/B)
space usage?
  - Per-entry = string + 4-byte pointer + 4-byte bucket ptr
  - ~16 bytes per word
Summary so far

<table>
<thead>
<tr>
<th>containsWord</th>
<th>Sorted vector</th>
<th>BST</th>
<th>Hashtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(lgN)</td>
<td>O(lgN)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>containsPrefix</td>
<td>O(lgN)</td>
<td>O(lgN)</td>
<td>O(N)</td>
</tr>
<tr>
<td>add</td>
<td>O(N)</td>
<td>O(lgN)</td>
<td>O(lgN)</td>
</tr>
<tr>
<td>bytes per word</td>
<td>O(1)</td>
<td>O(lgN)</td>
<td>O(lgN)</td>
</tr>
</tbody>
</table>

- Trie from retrieval (but pronounced "try")
- 26-way branching tree
- Paths are prefixes
- Path that ends at thick circle is word

Notice any patterns?

- OSPD2.txt file has 125,000 words
  - Average length 8 characters, file is 1.1 MB
- At 8-18 bytes per word, total is 1MB to 2.3 MB
- Boggle lexicon actually only takes 350K (.3 MB)....
  - How???

Lexicon as trie

```c
private:
struct node {
    char letter;
    bool isWord;
    node *children[26];
};
node *root;
containsWord?
    Trace path, check isWord, O(length)
containsPrefix?
    Trace path O(length)
add?
    Trace path, add new nodes as needed O(length)
space usage?
    Per-node = 106 bytes
    Trie for OSPD2 has ~250,000 nodes = 26.5 MB!!!
```
Dynamic array of children

Replace static array of children with dynamically-sized array

- Most of 26 entries are NULL anyway
- Per-node = 10 bytes + NumChildren*4 bytes
- Each node averages 6 children, so total 34 bytes
- Trie for OSPD2 has ~250,000 nodes = 8.5 MB...

Flatten tree into array

Flatten tree into array

- Like we did for heap, but not fixed location
- Saves all space used for pointers
- Makes data structure much less flexible (insert/delete words)

Space usage?

- Per-node = 10 bytes
- Trie for OSPD2 has ~250,000 nodes = 2.5 MB
- (Now same ballpark as bst/hash)

Where to go next?

- adapted
- basted
- depicted
- impacted
- interrupted
- baster
- baster
- depicter
- impacter
- interrupter
- basters
- basters
- depicters
- impacters
- interrupters
- basing
- basing
- depiction
- impaction
- interruption
- basions
- basions
- depictions
- impactions
- interruptions
- basits
- basits
- depists
- impacts
-interrupts
- adopted
- billed
- deserted
- indented
- invented
- adopter
- biller
- deserter
- indenter
- inventor
- adopters
- billers
- deserters
- indenters
- inventers
- adopting
- billing
- deserting
- indenting
- inventing
- adoption
- billion
- descretion
- indentation
- invention
- adoptions
- billions
- descretions
- indentions
- inventions
- adopt
- bils
- deserts
- intend
- invent
- affector
- camper
- detector
- inserted
- perfected
- affecter
- campers
- detectors
- inserters
- perfecters
- affecting
- camping
- detecting
- inserting
- perfecting
- affection
- campion
- detection
- insertion
- perfection
- affections
- campions
- detections
- insertions
- perfections
- affects
- camps
- detects
- inserts
- perfects
- asserted
- corrupted
- fruited
- intercepted
- portal
- asserter
- corruptor
- fruiter
- interceptor
- porter
- asserters
- corrupters
- fruiters
- intercepters
- porters
- asserting
- computing
- fruiling
- intercepting
- porting

Dawg: directed acyclic word graph

Unify suffixes as well as prefixes!

Directed
- ♦ Arcs go one-way only

Acyclic
- ♦ No cycles
  - banana, bananan...a

Graph
- ♦ Generalized tree
- ♦ Can be more than one path between nodes
Lexicon as dawg

```c
private:
    struct node {
        char letter;
        bool isWord;
        int childIndex;
        int nChildren;
    };
node *root;
```

Unify suffixes as well as prefixes
- Trie had 250,000 nodes, dawg has just 80,000
- 125,000 words — so many words have no unique nodes

Space usage?
- Per-node = 10 bytes
- Dawg for OSPD2 has ~80,000 nodes = .8MB

The final result

```c
private:
    struct node {
        5 bits letter;
        1 bit isWord;
        1 bit for lastChild
        25 bits childIndex;
    };
node *root;
```

Bitpack node into smallest size
- Only 26 letters, don’t need full ASCII, squeeze in 5 bits
- isWord/last each 1 bit
- Remaining 25 bits used for index
- Each node shrinks from 10 bytes down to 4

Space usage?
- Dawg for OSPD2 has ~80,000 nodes = .32 MB
- Just ~30% of the size of word file!

Cool facts about the dawg

Easily written/read to disk
- Just take the entire array and write it out as is
- Indexes don’t change, no pointers!
- Easy to restore
- That’s what the binary ”lexicon.dat” file is

Data structure facilitates other interesting operations
- Tweak width of ”beam” as you traverse path to explore just near neighbors
  - Regex, spelling corrections, slight permutations
- Solving puzzles
  - Anagrams, hangman, scrabble, etc.

Original reference