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

```
private:
  Vector<string> words;
```

containsWord?
- Binary search using ==, O(lgN) time

containsPrefix?
- Binary search using substr ==, O(lgN)

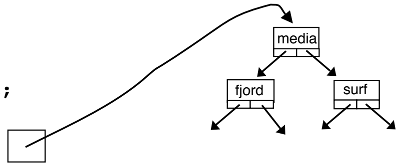
add?
- Binary search to find position, shuffle to insert, O(N)

space usage?
- Per-entry = sizeof(string) ≈ length*sizeof(char)
- ≈ 8 bytes per word (assuming words average len 8 chars)

Lexicon as binary search tree

```
private:
  struct node {
    string word;
    node *left, *right;
  };

  node *root;
```



containsWord?
- Tree search using ==, O(lgN) time

containsPrefix?
- Tree search using substr ==, O(lgN)

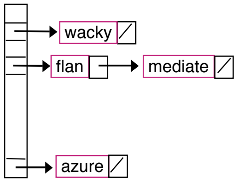
add?
- Tree search to find position, insert new leaf, O(lgN)

space usage?
- Per-entry = string + left/right ptrs + balance factor
- ≈ 18 bytes per word

Lexicon as hash table

```
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

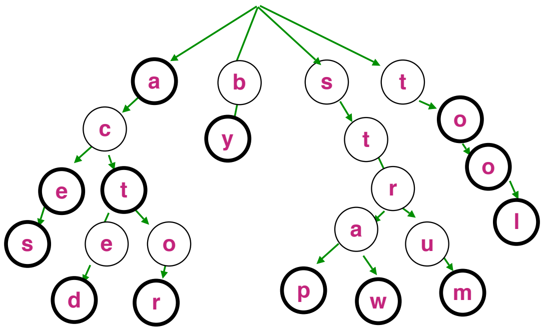
	Sorted vector	BST	Hashtable
containsWord	O(lgN)	O(lgN)	O(1)
containsPrefix	O(lgN)	O(lgN)	O(N)
add	O(N)	O(lgN)	O(1)
bytes per word	8	18	16

- 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???

Notice any patterns?

straddle
straddled
straddler
straddlers
straddles
straddling
strafe
strafed
strafer
strafers
strafes
strafing
straggle
straggled
straggler
stragglers
straggles
stragglier
straggliest
straggling
straggly
straight
straightaway
straightaways
straighted
straightedge
straightedges
straighten
straightened
straightener
straighteners
straightening
straightens
straighter
straightest
straightforward
straightforwardly
straightforwardness
straighting
straightjacket
straightjackets
straightlaced
straightly
straights
straightway
strain
strained
strainer
strainers
straining
strains
strait
straiten
straitened
straitening
straitens
straiter
straitest
straitjacket
straitjackets
straitlaced
straitly
straitness
straits
strake
straked
strakes
stramash
stramashes
stramonies
stramony
strand
stranded
strander
stranders
stranding
strands
strang
strange
strangely
strangeness
stranger
strangered
strangering
strangers
strangest
strangle
strangled
stranglehold
strangleholds
strangler
stranglers
strangles
strangling
strangulate
strangulated
strangulates
strangulating
strangulation
strap
strapless
strapped
strapper
strappers
strapping
strappings
straps
strass
strasses
strata
stratagem
stratagems
stratal
stratas
strategic
strategical
strategically
strategies
strategist
strategists
strategy
stratification
stratifications
stratified
stratifies
stratify
stratifying
stratosphere
stratospheres
stratospheric
stratum
stratums
stratus
stravage
stravaged
stravages
stravaging
stravaig
stravaiged
stravaiging
stravaigs
straw
strawberries
strawberry

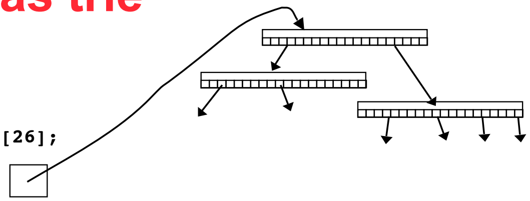
Letter trie



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

Lexicon as trie

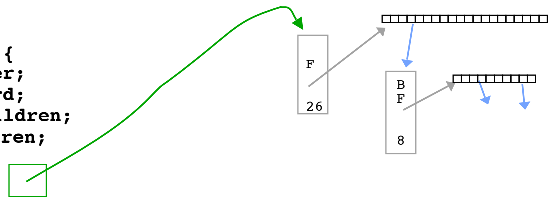
```
private:
 struct node {
   char letter;
   bool isWord;
   node *children[26];
 };
 node *root;
```



- 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

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



Replace static array of children with dynamically-sized array

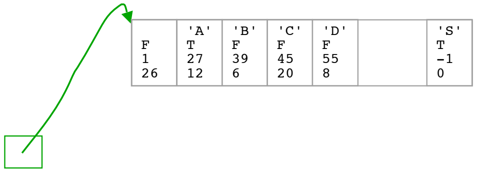
- Most of 26 entries are NULL anyway

Space usage?

- 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

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



	'A'	'B'	'C'	'D'		'S'
F	T	F	F	F		T
1	27	39	45	55		-1
26	12	6	20	8		0

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
adapter
adapters
adapting
adaption
adaptions
adapts
adopted
adopter
adopters
adopting
adoption
adoptions
adopts
affected
affecter
affecters
affecting
affection
affections
affects
asserted
asserter
asserters
asserting

basted
baster
basters
basting
bastion
bastions
basts
billed
biller
billers
billing
billion
billions
bills
camped
camper
campers
camping
campion
campions
camps
corrupted
corrupter
corrupters
corrupting

depicted
depicter
depicters
depicting
depiction
depictions
depicts
deserted
deserter
deserters
deserting
desertion
desertions
deserts
detected
detecter
detecters
detecting
detection
detections
detects
fruited
fruiter
fruiters
fruiting

impacted
impacter
impacters
impacting
impaction
impactions
impacts
indented
indenter
indenters
indenting
indention
indentions
indents
inserted
inserter
inserters
inserting
insertion
insertions
inserts
intercepted
intercepter
intercepters
intercepting

interrupted
interrupter
interrupters
interrupting
interruption
interruptions
interrupts
invented
inventer
inventers
inventing
invention
inventions
invents
perfected
perfecter
perfecters
perfecting
perfection
perfections
perfects
ported
porter
porters
porting

Dawg: directed acyclic word graph

Unify suffixes as well as prefixes!

Directed

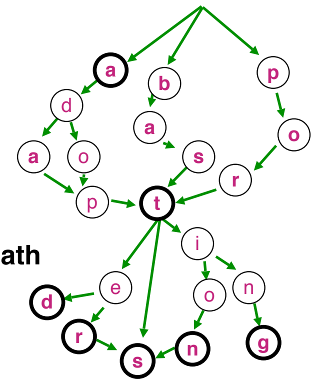
- Arcs go one-way only

Acyclic

- No cycles
  - banana, bananan…a

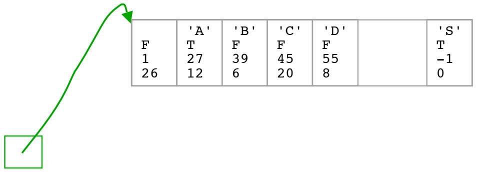
Graph

- Generalized tree
- Can be more than one path between nodes



Lexicon as dawg

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



	'A'	'B'	'C'	'D'		'S'
F	T	F	F	F		T
1	27	39	45	55		-1
26	12	6	20	8		0

Unify suffixes as well as prefixes

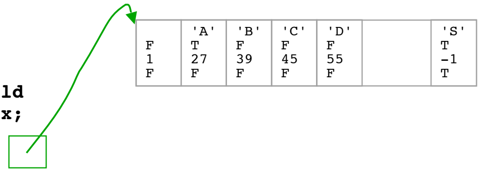
- Trie had 250,000 nodes, dawg has just 80,000
- 125,000 words — so many words have no unique nodes

Space usage?

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

The final result

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



	'A'	'B'	'C'	'D'		'S'
F	T	F	F	F		T
1	27	39	45	55		-1
F	F	F	F	F		T

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

- "The World's Fastest Scrabble Player", Appel & Jacobsen, CACM May 1988