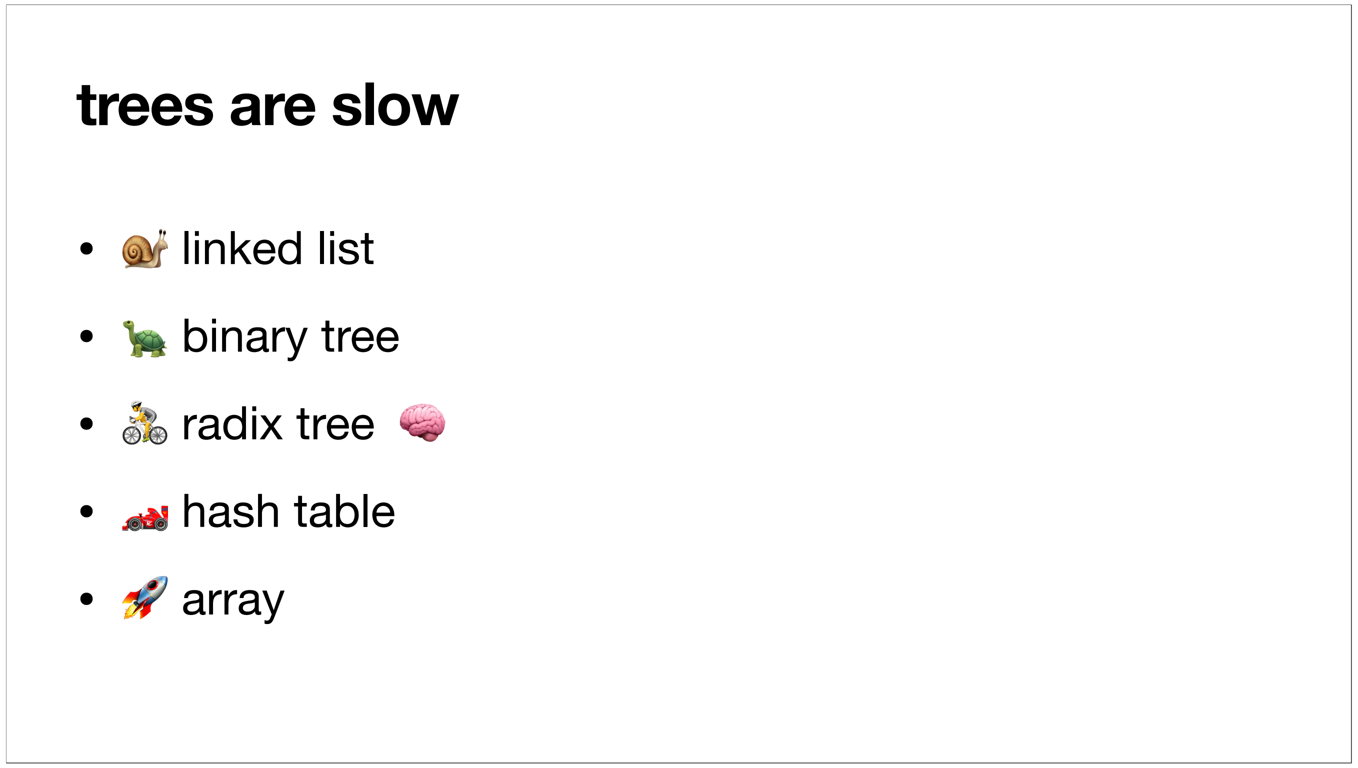
how a qp-trie works

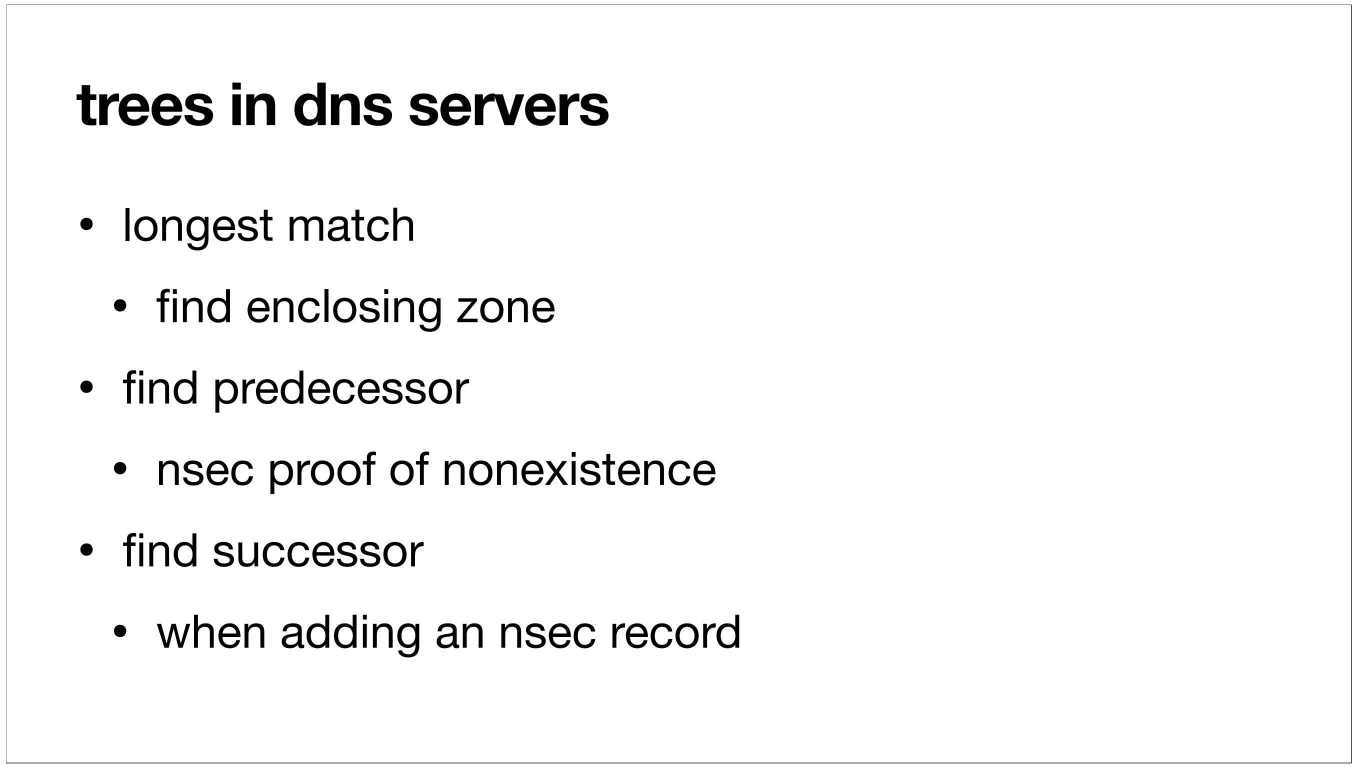
5 cool ideas and 3 revelations



trees are slow

- 🐌 linked list
- 🐢 binary tree
- 🚴 radix tree 🧠
- 🏎️ hash table
- 🚀 array

indirect memory accesses
unpredictable addresses



trees in dns servers

- longest match
  - find enclosing zone
- find predecessor
  - nsec proof of nonexistence
- find successor
  - when adding an nsec record

why we can’t use a hash table

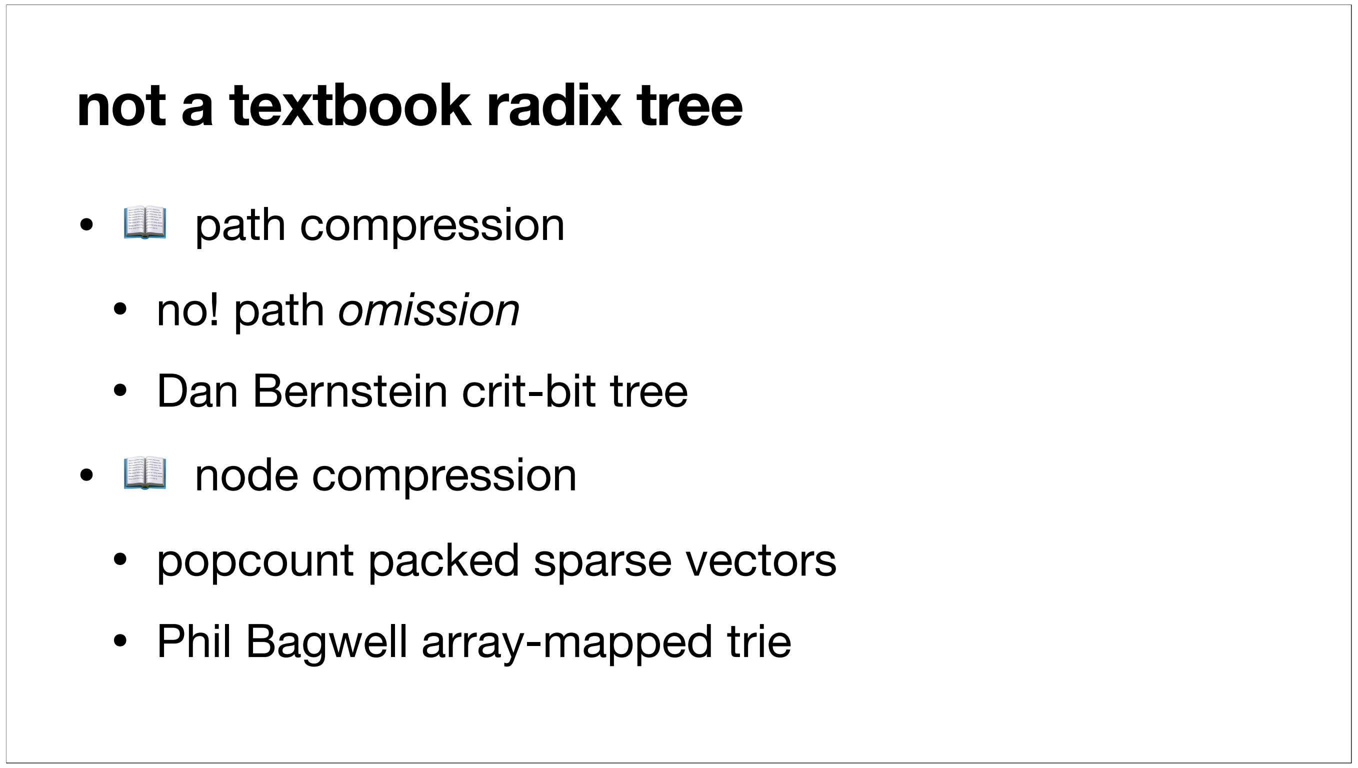
kinds of trees

- comparison tree
  - binary tree: red-black / splay / avl
  - b-tree
- radix tree
  - same sort order as memcmp()
  - trie = information reTRIEval

comparison trees can be based on an arbitrary user defined key ordering

radix trees must order keys bytewise lexicographically

“trie” is a pompous way to write “tree” while hinting it is some kind of radix tree



not a textbook radix tree

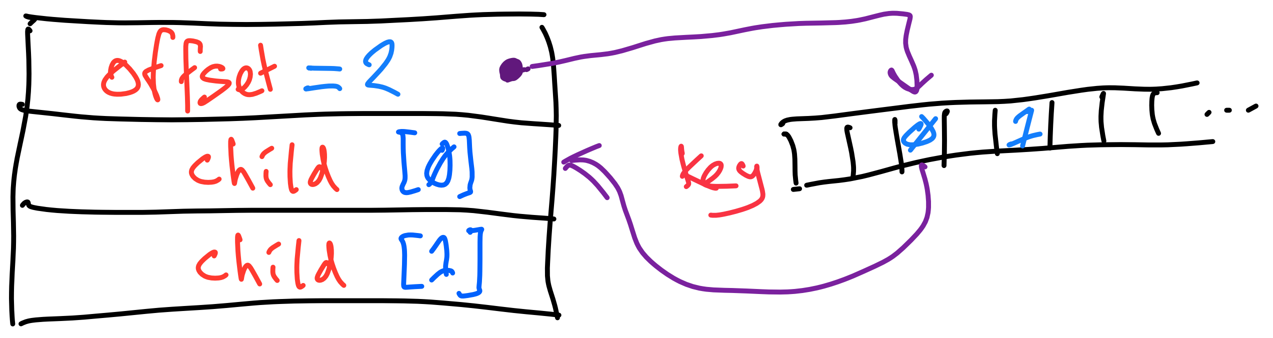
- 📖 path compression
  - no! path *omission*
  - Dan Bernstein crit-bit tree
- 📖 node compression
  - popcount packed sparse vectors
  - Phil Bagwell array-mapped trie

radix trees are known for using lots of memory

two cool ideas behind original qp-trie

a qp-trie can use as little memory as a red-black tree

crit-bit tree

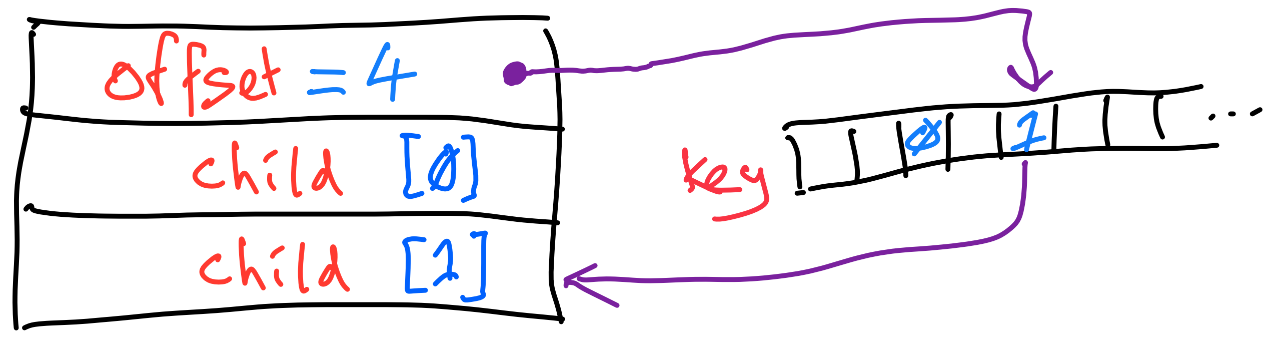


path omission: the offset lets us skip parts of the keys in the tree that are the same

but the search key can differ in one of the parts we skipped, so we compare it with the leaf key at the end to find their common prefix

fancy searches (predecessor, successor, longest match) require a second pass

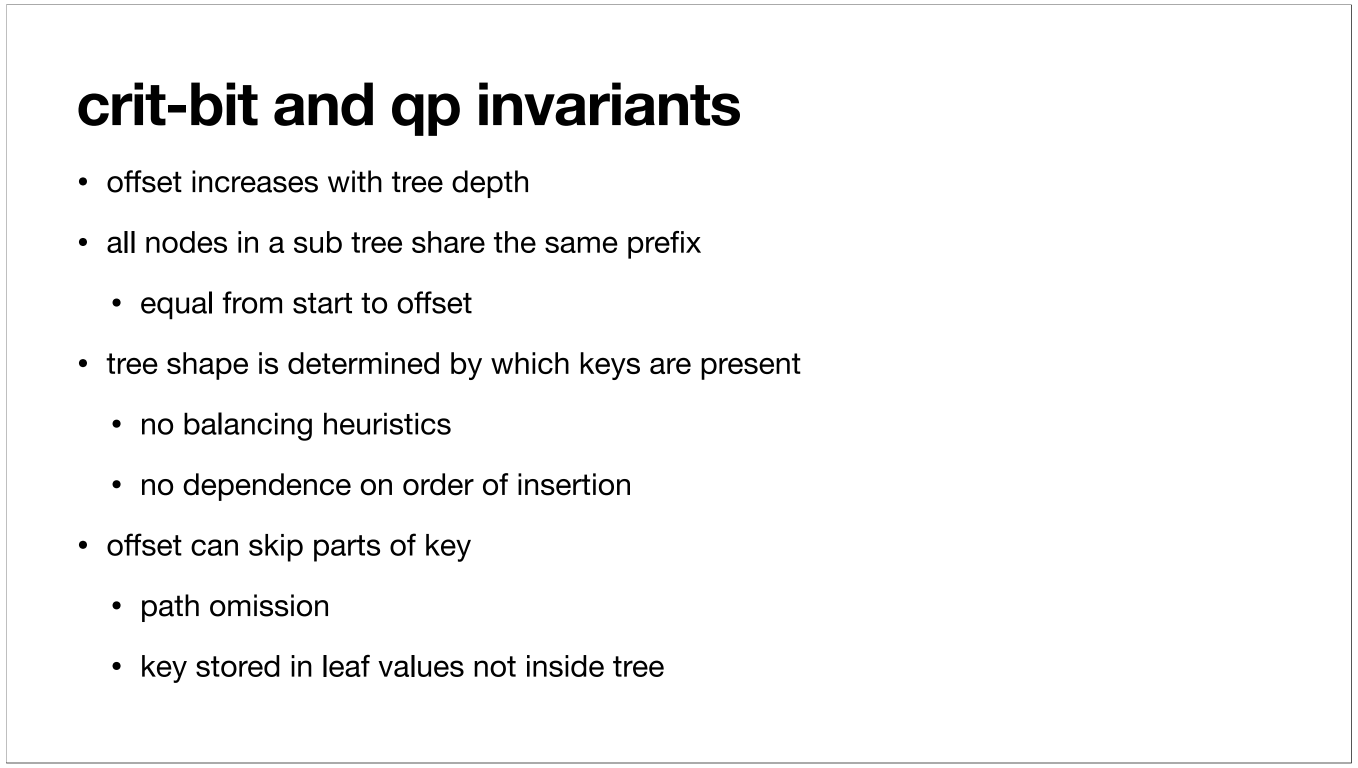
crit-bit tree



path omission: the offset lets us skip parts of the keys in the tree that are the same

but the search key can differ in one of the parts we skipped, so we compare it with the leaf key at the end to find their common prefix

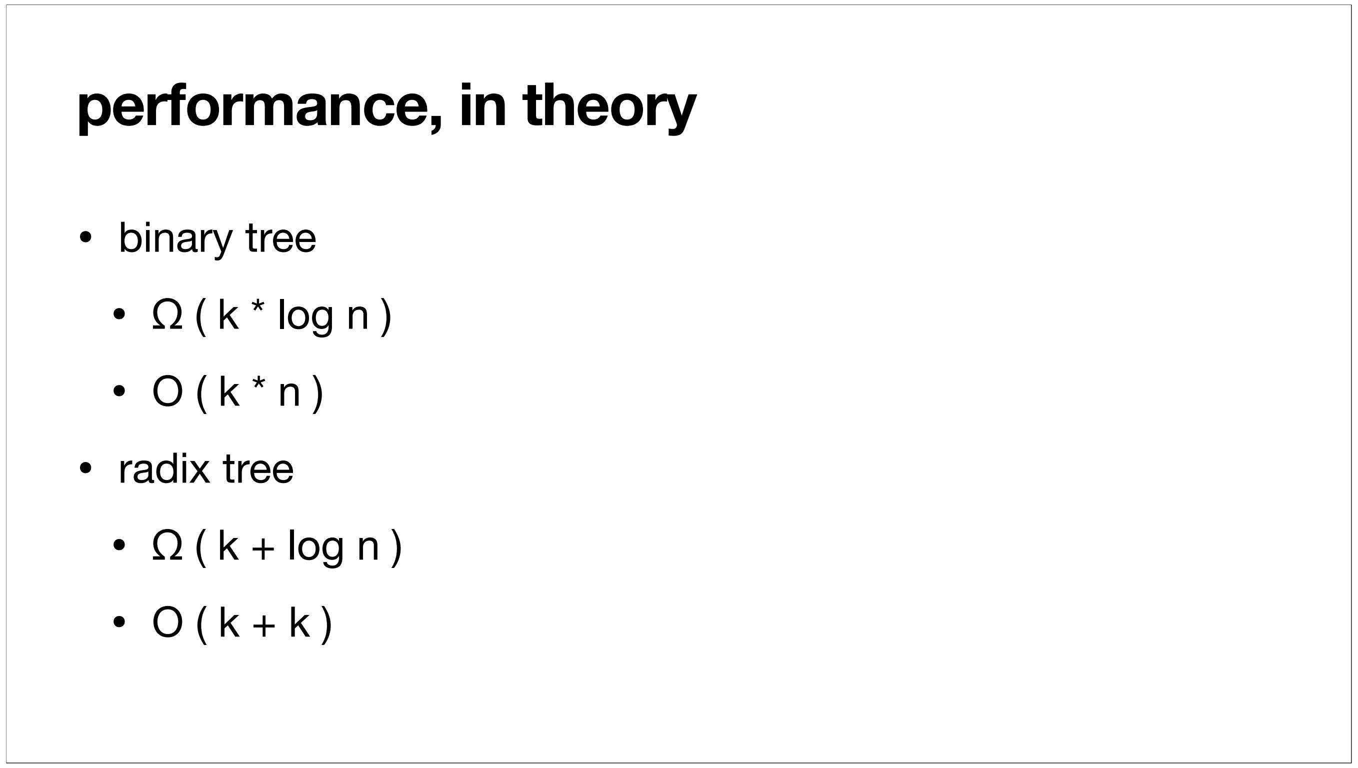
fancy searches (predecessor, successor, longest match) require a second pass



crit-bit and qp invariants

- offset increases with tree depth
- all nodes in a sub tree share the same prefix
  - equal from start to offset
- tree shape is determined by which keys are present
  - no balancing heuristics
  - no dependence on order of insertion
- offset can skip parts of key
  - path omission
  - key stored in leaf values not inside tree

a crit-bit tree is basically a one-bit qp-trie, so they share a lot of invariants



performance, in theory

- binary tree
  - $\Omega (k * \log n)$
  - $O (k * n)$
- radix tree
  - $\Omega (k + \log n)$
  - $O (k + k)$

so, how can i make a crit-bit tree faster?

$\Omega()$ lower bound
$O()$ upper bound

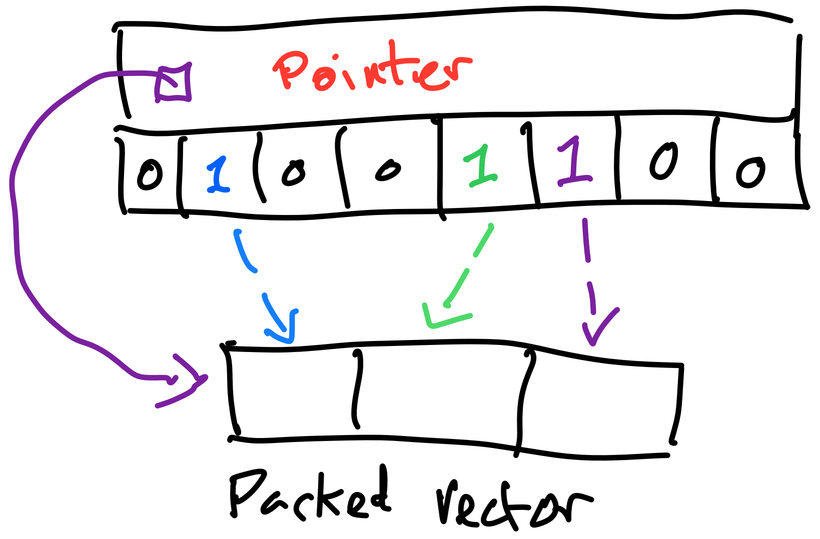
k length of keys
n number of keys

left term is key comparison, right term is tree depth

hidden constant factor in the base of the logarithm

increase base of logarithm to go faster: wider, shallower tree

popcount packed sparse vector



fat pointer: pointer and bitmap instead of pointer and length

bitmap indicates which elements are present

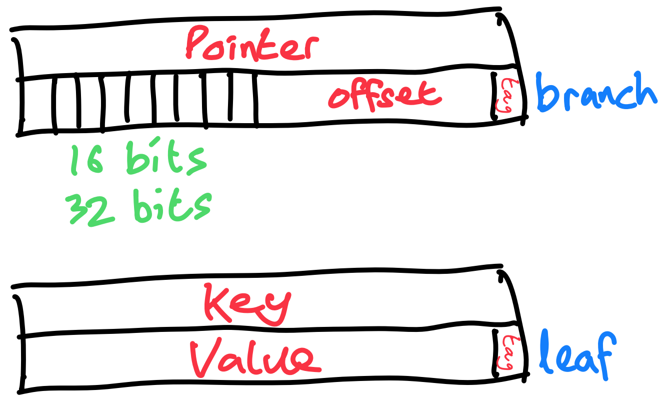
works best when bitmap fits in a word, in a Bagwell tree, 32 or 64 bits

```
  // size of vector
size = popcount(node->bitmap);
  // bit position
bit =  1 << key[node->offset];
  // is element present?
if (node->bitmap & bit) { ... }
  // position of element in packed vector
pos = popcount(node->bitmap & (bit - 1));
  // step to child node
node = node->children + pos;
```

the p in qp stands for popcount, population count, Hamming weight, “sideways add” according to Knuth

gives us quick random access into packed array, very nice for radix tree node compression

qp-trie



rough idea of the original qp-trie node layout

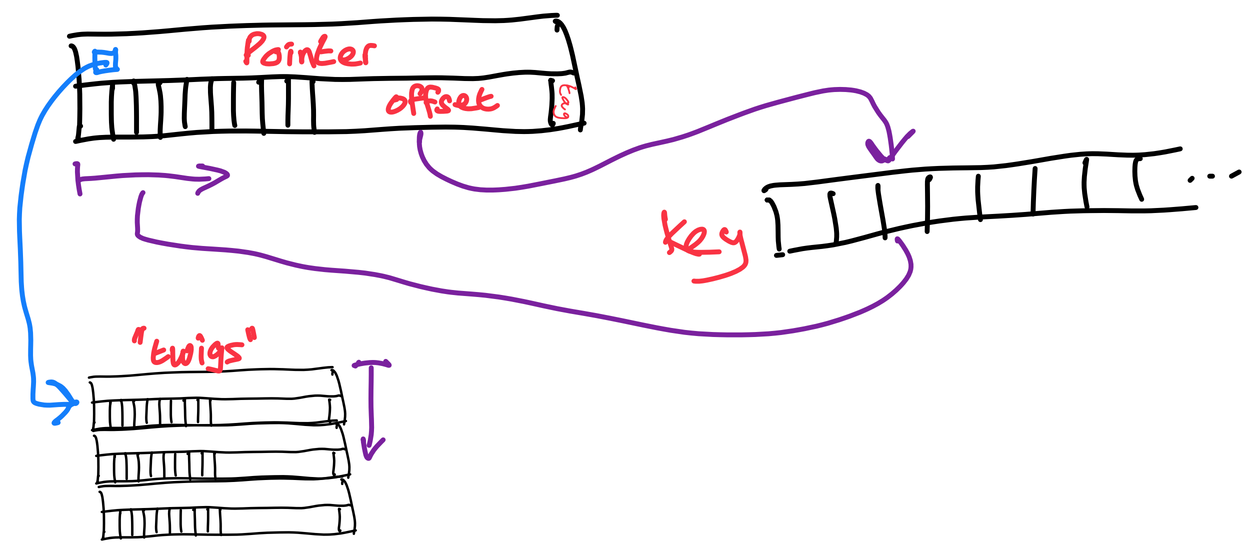
2 words of 64 bits each

smuggle a one bit discriminant tag into a pointer

bitmap and children vector from Bagwell, key offset from Bernstein

qp-trie in knot dns uses a 16 wide bitmap

qp-trie

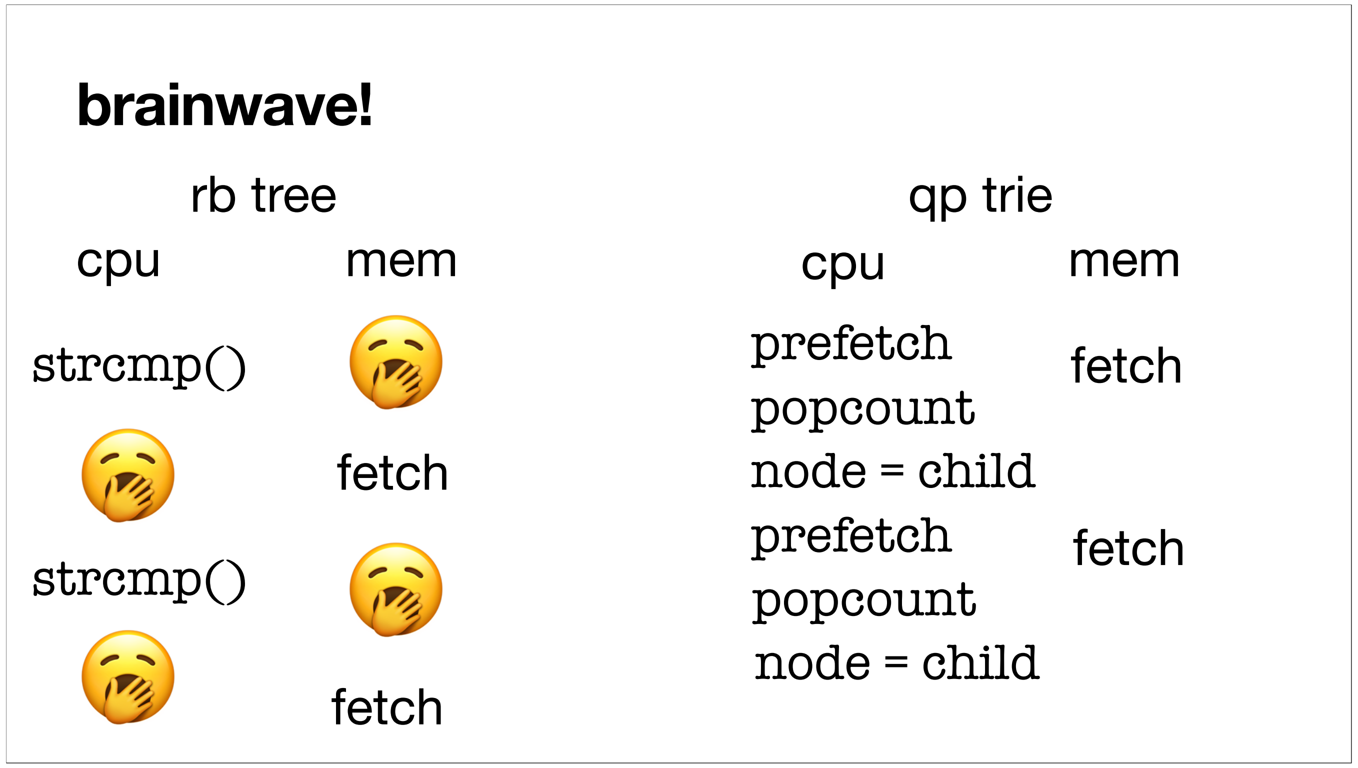


rough idea of the original qp-trie node layout

2 words of 64 bits each

smuggle a one bit discriminant tag into a pointer

bitmap and children vector from Bagwell, key offset from Bernstein



brainwave!

rb tree

cpu	mem
strcmp()	🥱
🥱	fetch
strcmp()	🥱
🥱	fetch

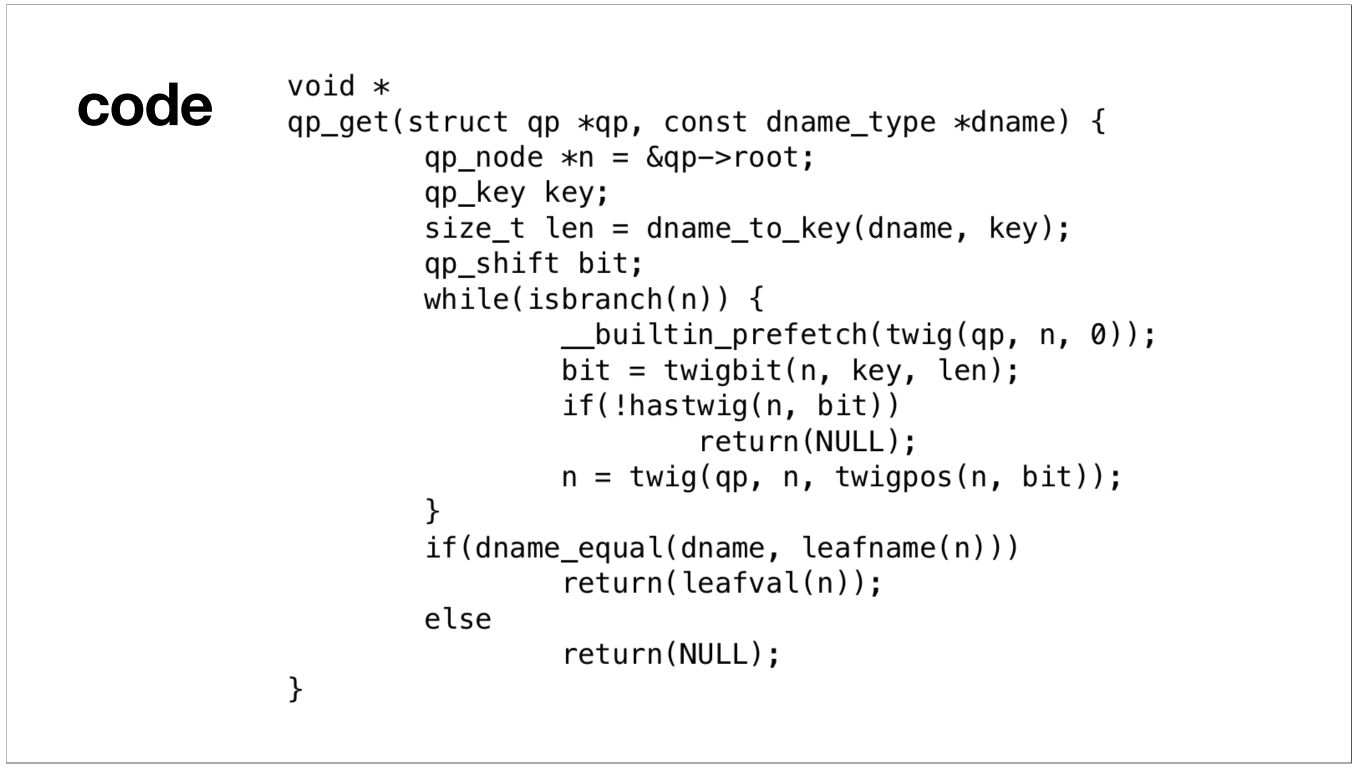
qp trie

cpu	mem
prefetch	fetch
popcount	
node = child	
prefetch	fetch
popcount	
node = child	

received wisdom is that prefetch is not much use for data structures with lots of indirection

but a qp-trie benefits from it a lot! old cpus benefit from a prefetch hint, new ones can work it out for themselves

helps keep both memory and cpu working concurrently



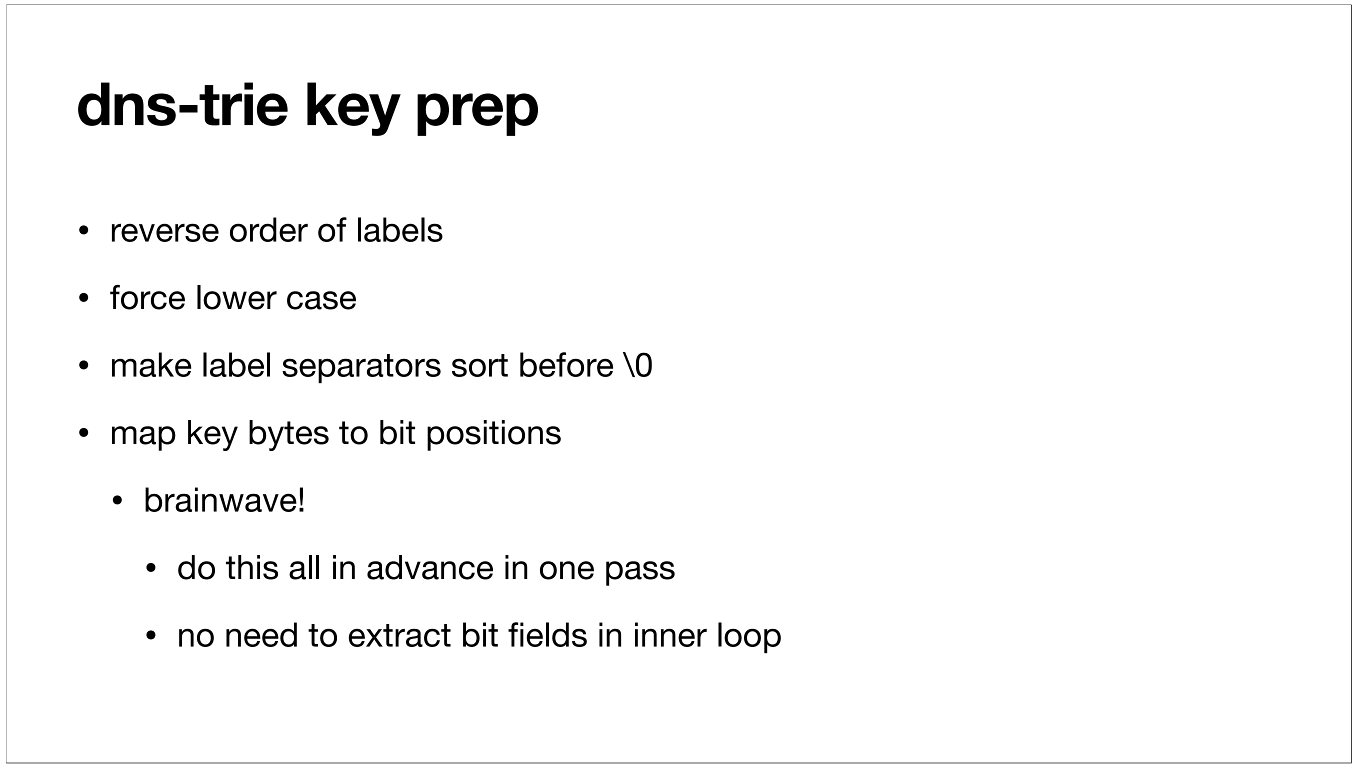
code

```
void *
qp_get(struct qp *qp, const dname_type *dname) {
        qp_node *n = &qp->root;
        qp_key key;
        size_t len = dname_to_key(dname, key);
        qp_shift bit;
        while(isbranch(n)) {
                __builtin_prefetch(twig(qp, n, 0));
                bit = twigbit(n, key, len);
                if(!hastwig(n, bit))
                        return(NULL);
                n = twig(qp, n, twigpos(n, bit));
        }
        if(dname_equal(dname, leafname(n)))
                return(leafval(n));
        else
                return(NULL);
}
```

because it fits on a slide

verbatim from NSD

the little helper functions typically compile down to a few ALU ops; the only hidden conditional is a bounds check on the length of the key



dns-trie key prep

- reverse order of labels
- force lower case
- make label separators sort before \0
- map key bytes to bit positions
  - brainwave!
    - do this all in advance in one pass
    - no need to extract bit fields in inner loop

dns-trie is a qp-trie specialized for domain names as keys

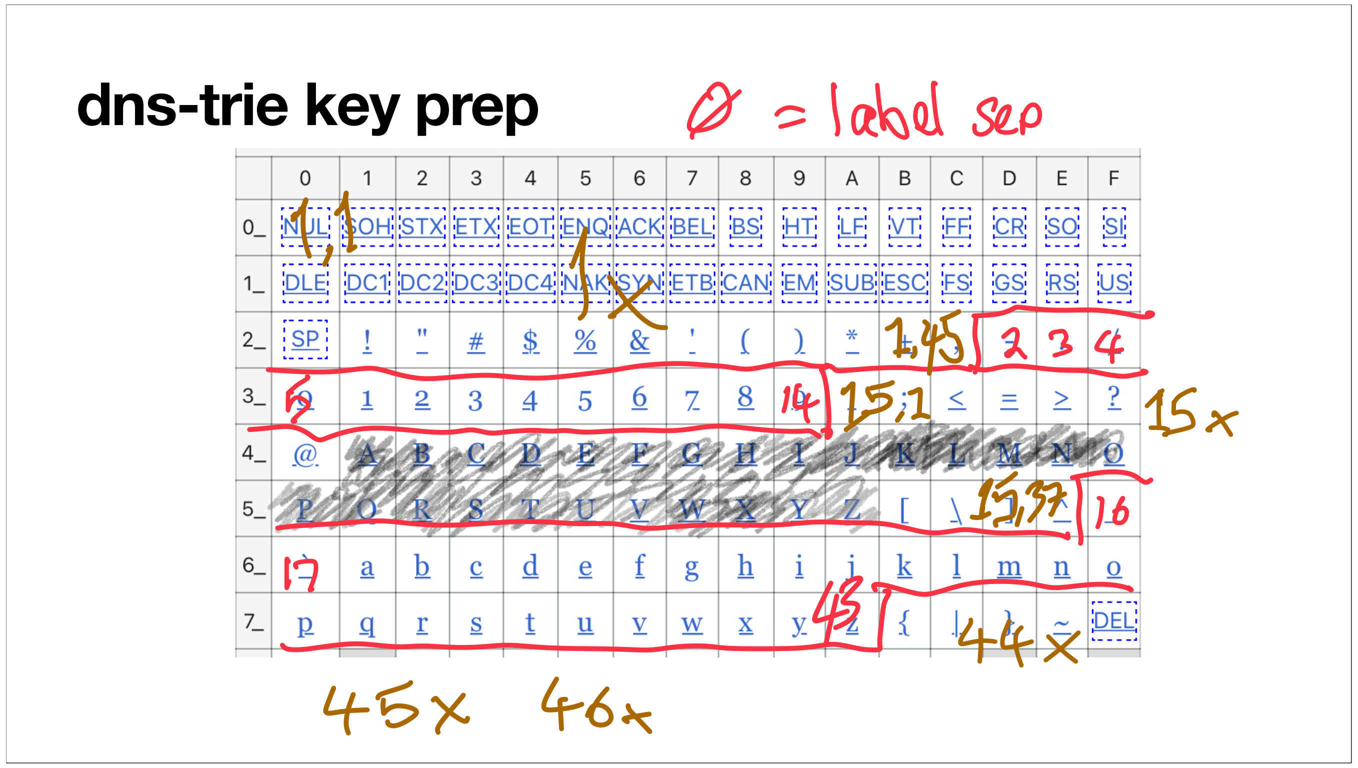
goal: one byte of name per indirection (instead of 4 or 5 bits in classic qp-trie)

domain names need to be massaged before they can be used as radix tree keys (first three points on the slide)

if we massage them more cleverly, we can make lookups faster and simpler!

if we try to use a byte at a time without cleverness, we would need a 256 wide bitmap, and our nodes would become bloated and slow

revelation from summer 2020



dns-trie key prep

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0_	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1_	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2_	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3_	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4_	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5_	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6_	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7_	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

∅ = label sep
1,1
1x
1,45
2 3 4
5
14
15,1
15x
15,37
16
17
43
44x
45x 46x

instead we make use of the fact that almost all domain names follow host name syntax

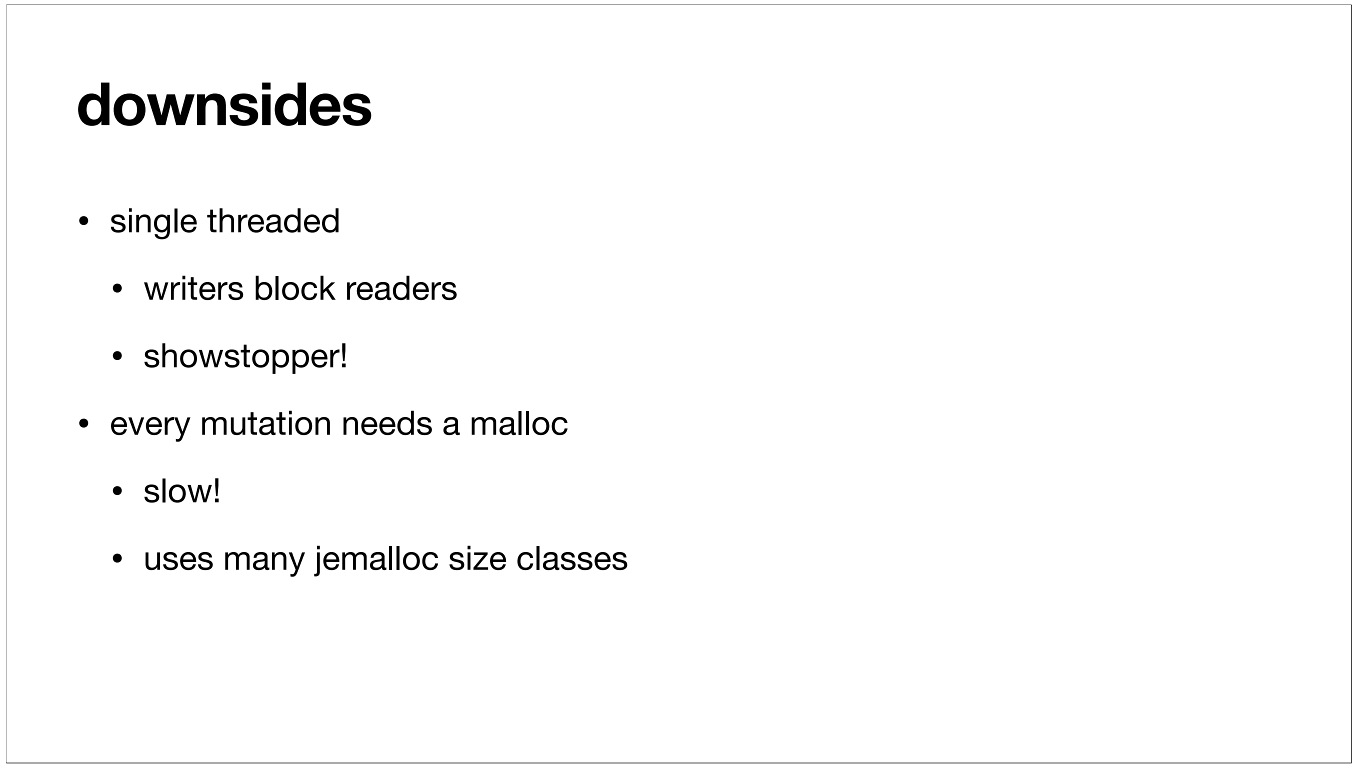
so we only need a 40 wide bitmap to support the vast majority of bytes we will actually encounter: hyphen to 9, underscore to z, and the label separator

we use a few more bits for escape characters, so rare bytes in domain name turn into two bytes in our lookup key, eg 15, 7 for ‘@’

the escaping scheme preserves DNSSEC canonical sorting order

ideas so far

- use a radix tree
  - faster in theory and practice
- Bernstein’s path omission
- Bagwell’s popcount node compression
  - smaller size
- my prefetch discovery
  - sometimes you can predict the future
- use hostname syntax
  - fewer faster steps to climb the tree

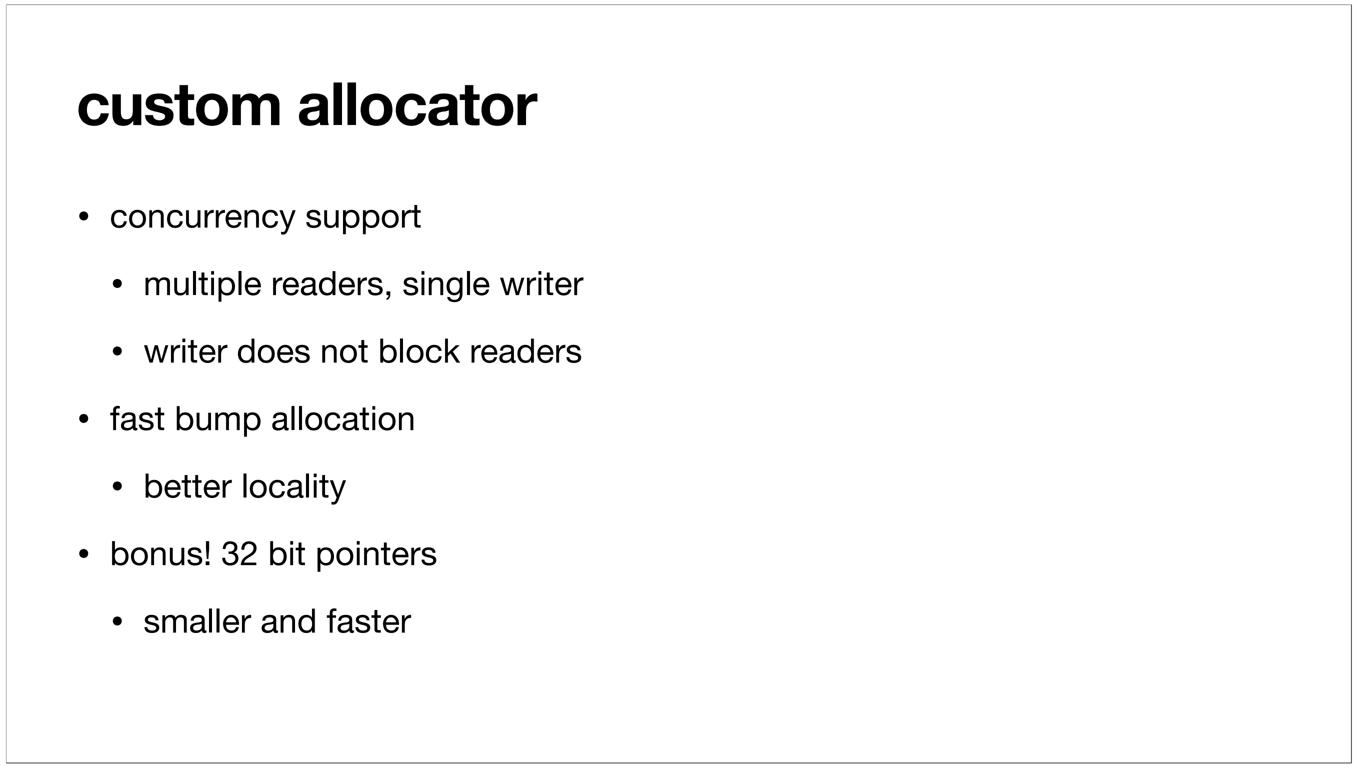


downsides

- single threaded
  - writers block readers
  - showstopper!
- every mutation needs a malloc
  - slow!
  - uses many jemalloc size classes

a write can be complicated: UPDATE or IXFR

size classes segregated in to different pages, greater TLB usage



custom allocator

- concurrency support
  - multiple readers, single writer
  - writer does not block readers
- fast bump allocation
  - better locality
- bonus! 32 bit pointers
  - smaller and faster

i was very wary of writing a custom allocator, as Bagwell suggested, because it seemed like over-engineering

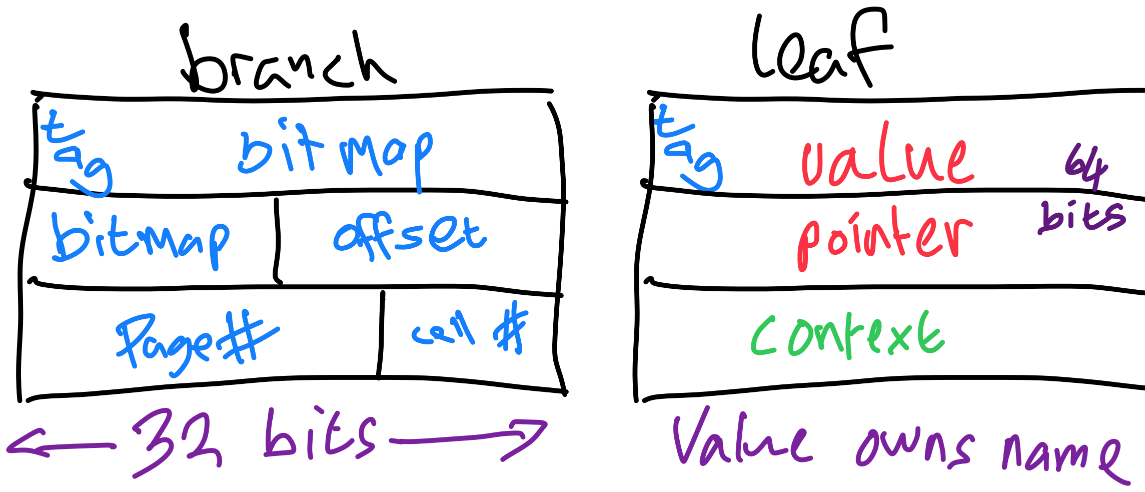
but it deals with both major downsides, and gives us an extra bonus

the cost is about 1.75x the code size

(Bagwell's allocator is very different from mine and does not address concurrency)

revelation: simpler than i expected, and much easier than my previous refcount-based cow implementation for knot dns

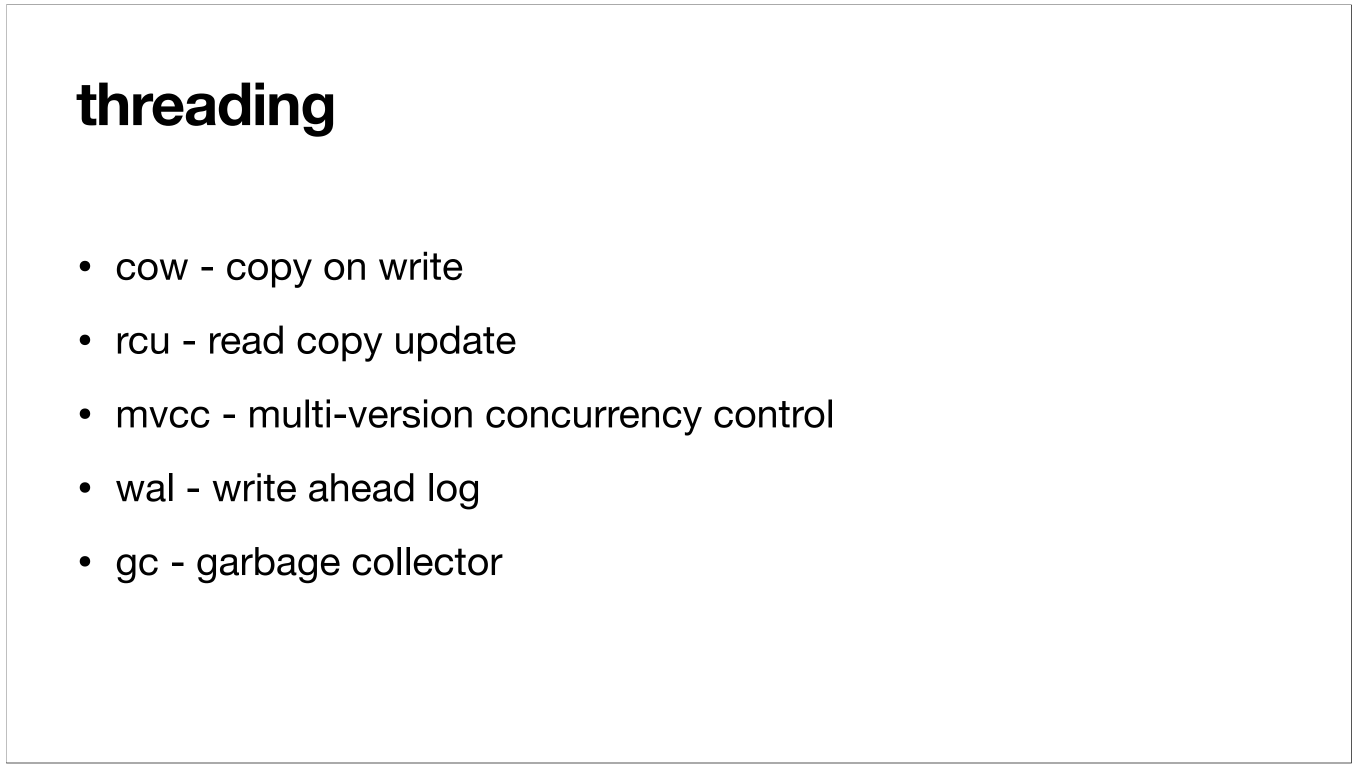
qp-trie



summer 2021 version

12 bytes per node

much less than one branch node per leaf node, so about 20 bytes per element, plus space for names



threading

- cow - copy on write
- rcu - read copy update
- mvcc - multi-version concurrency control
- wal - write ahead log
- gc - garbage collector

cow - general technique to allow readers to keep working while an update is in progress

WHY? because qp-trie nodes are too small for lock-based concurrency! (i guess it could use out-of-tree locks like rbtdb?)

rcu - cow in the linux kernel, with clever techniques allowing readers to be completely unlocked; commits just swap a pointer and unlike rwlock do not block readers

mvcc - cow in databases; snapshot isolation; dbversion in bind

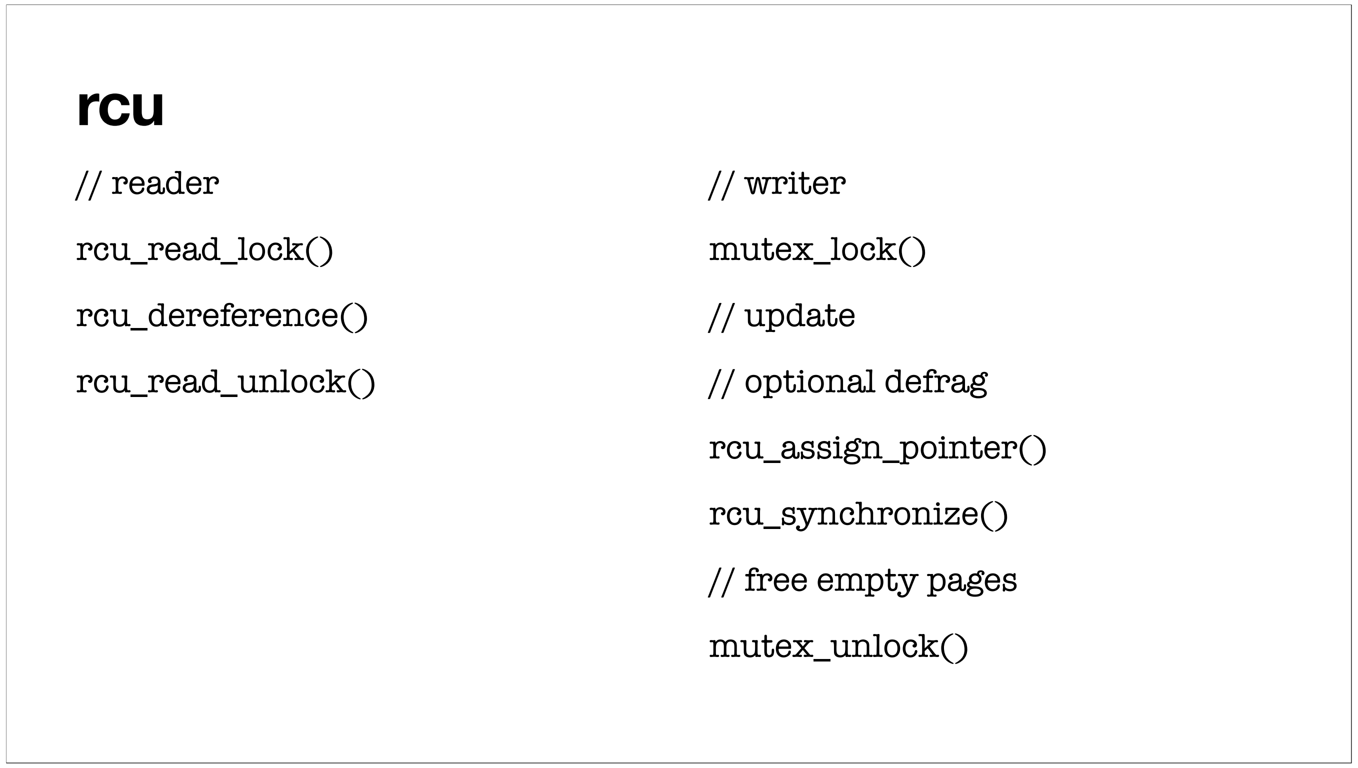
wal - a way to implement mvcc; closely related to bump allocators in programming language runtimes (better locality than jemalloc size classes)

gc - all of these need some kind of gc

rcu is nonblocking because it relies on being able to free unused memory later

postgres vacuum is a gc for its wal

gc does not have to be bad: compacting gc gives good locality; we can choose when it runs and combine with (eg) cache cleaning



rcu

```
// reader
rcu_read_lock()
rcu_dereference()
rcu_read_unlock()
```

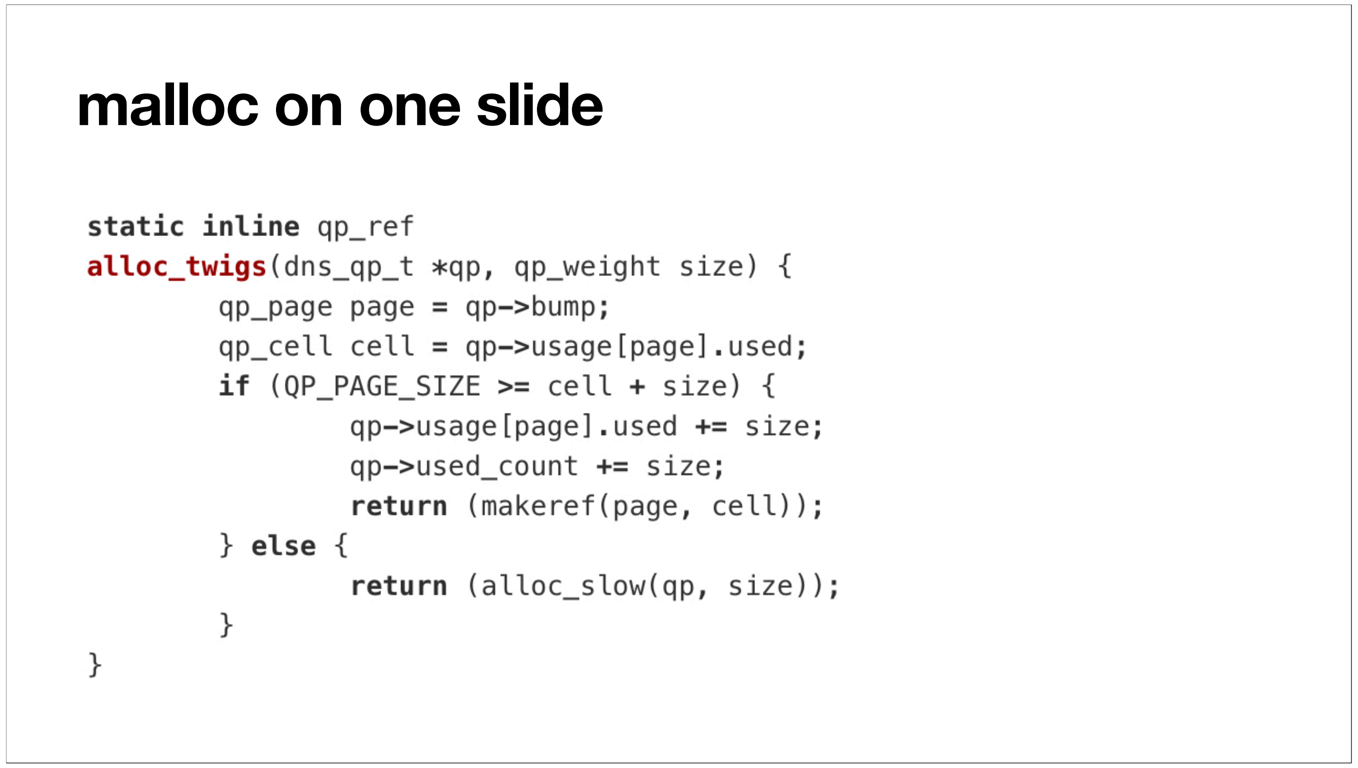
```
// writer
mutex_lock()
// update
// optional defrag
rcu_assign_pointer()
rcu_synchronize()
// free empty pages
mutex_unlock()
```

reader is nonblocking

read locks are close to no-ops

synchronize is where writer waits for readers to exit critical sections

can use rwlock instead of rcu, but the rcu pattern is a helpful design guide



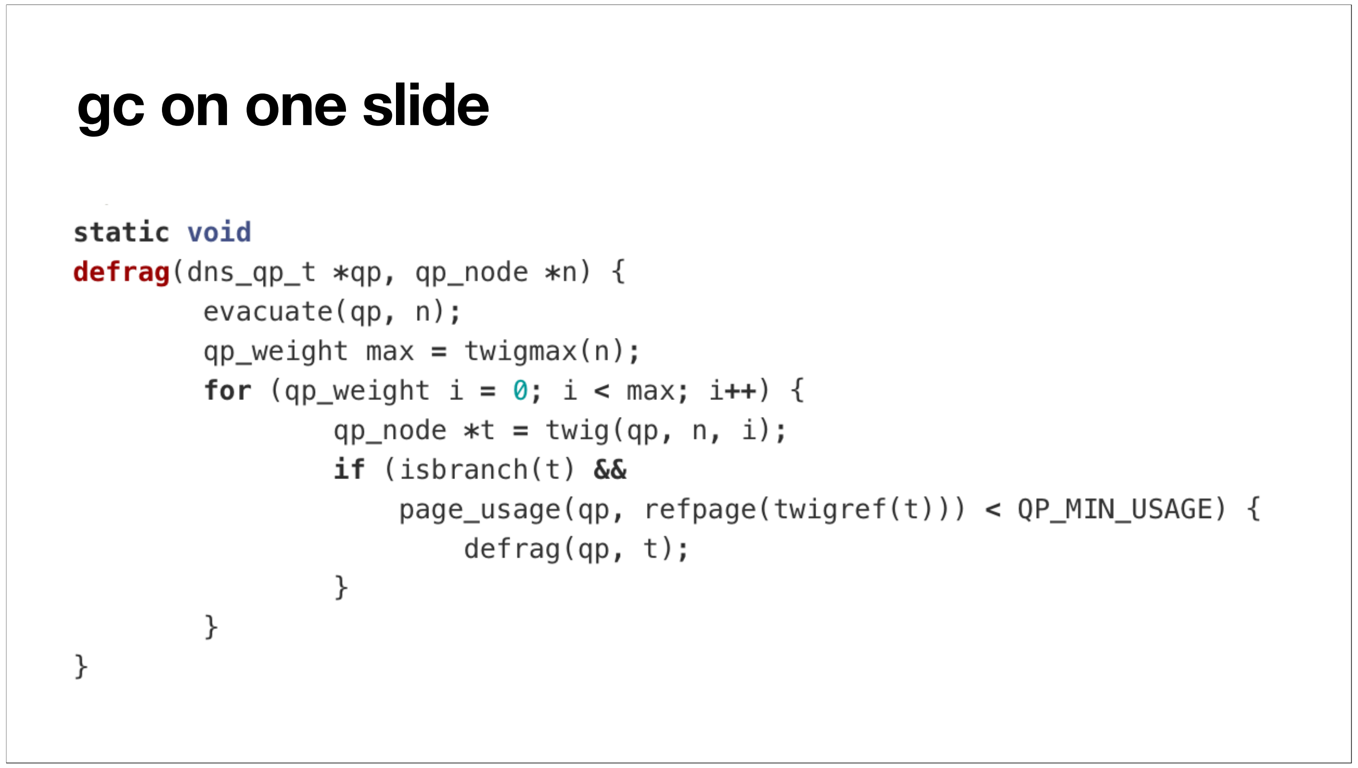
malloc on one slide

```
static inline qp_ref
alloc_twigs(dns_qp_t *qp, qp_weight size) {
        qp_page page = qp->bump;
        qp_cell cell = qp->usage[page].used;
        if (QP_PAGE_SIZE >= cell + size) {
                qp->usage[page].used += size;
                qp->used_count += size;
                return (makeref(page, cell));
        } else {
                return (alloc_slow(qp, size));
        }
}
```

a bump allocator is very simple!

good locality, eg, when we need to cow we copy the path from the root to the new node, sequentially in memory (the parts that have not already been copied)

i have cheated by leaving out the slow path page allocator



gc on one slide

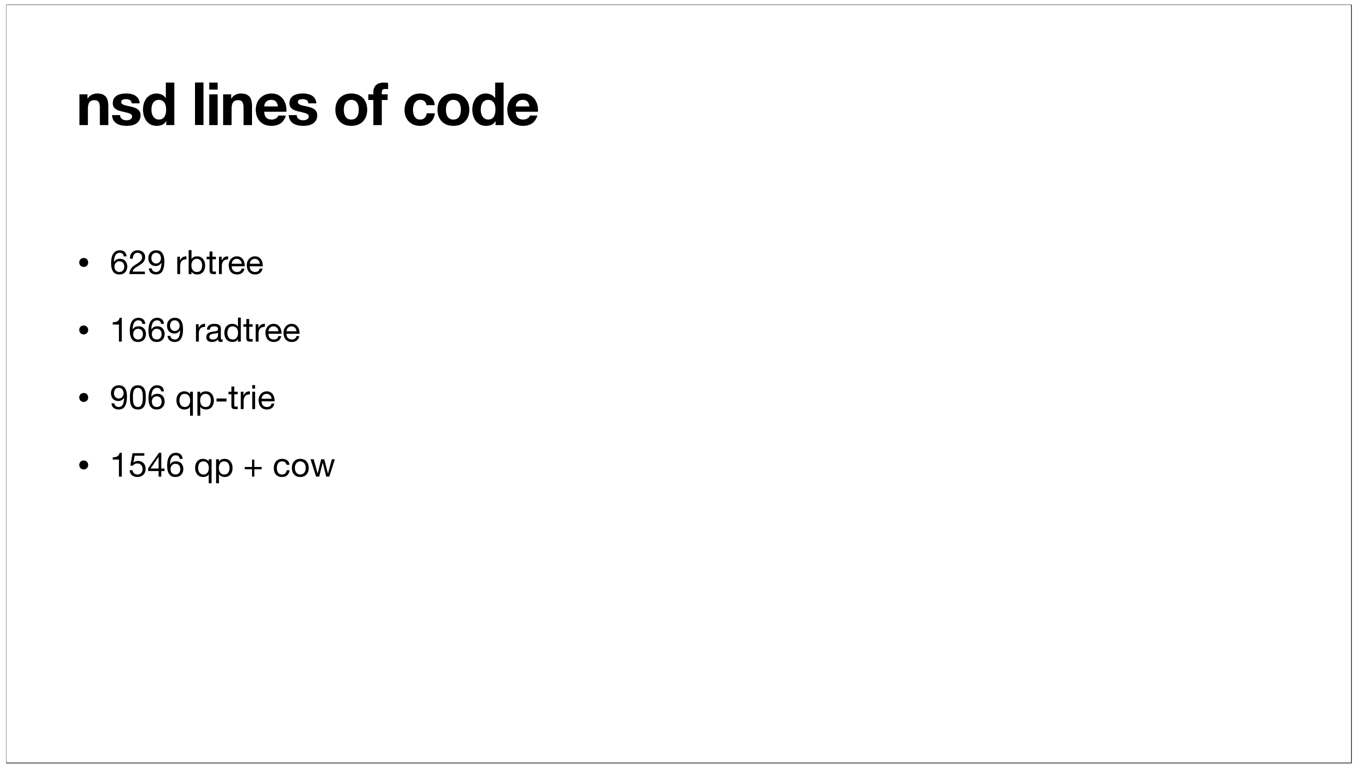
```
static void
defrag(dns_qp_t *qp, qp_node *n) {
        evacuate(qp, n);
        qp_weight max = twigmax(n);
        for (qp_weight i = 0; i < max; i++) {
                qp_node *t = twig(qp, n, i);
                if (isbranch(t) &&
                    page_usage(qp, refpage(twigref(t))) < QP_MIN_USAGE) {
                        defrag(qp, t);
                }
        }
}
```

we know it is a tree (not a graph) and we know it can’t be pathologically deep, so a copying collector is easy

for a small tree (a small zone) this naturally compacts it into a small contiguous section of a page, which we can realloc() to fit perfectly, so no need for a large minimum memory usage

this algorithm is can keep fragmentation very low; i tried several alternatives; clever ones were worse!

i have cheated by leaving out page reclamation



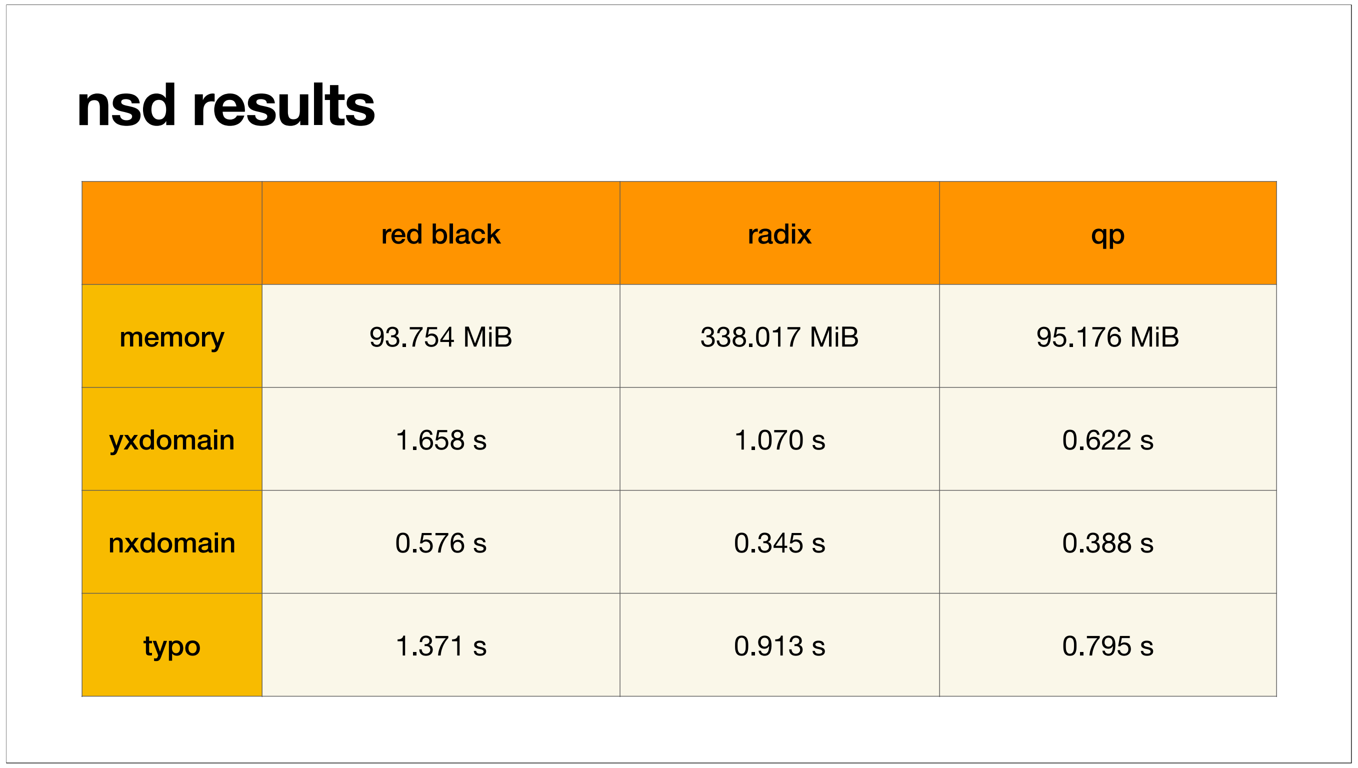
nsd lines of code

- 629 rbtree
- 1669 radtree
- 906 qp-trie
- 1546 qp + cow

apples to apples comparison except the last one

total count for .c and .h files that implement each data structure

the benchmark used the COW version (the non-COW version uses about 150 MiB)



nsd results

	red black	radix	qp
memory	93.754 MiB	338.017 MiB	95.176 MiB
yxdomain	1.658 s	1.070 s	0.622 s
nxdomain	0.576 s	0.345 s	0.388 s
typo	1.371 s	0.913 s	0.795 s

memory usage competitive with red-black tree

usually fastest

1,000,000 queries, one thread

yxdomain = known to exist in tree

nxdomain = completely random

typo = small random change to known domain