Hashing Algorithms

Hash functions

Separate Chaining

Linear Probing

Double Hashing

Symbol-Table ADT

Records with keys (priorities)

basic operations

- insert
- search
- create
- test if empty
- destroy
- copy

generic operations common to many ADTs

not needed for one-time use but critical in large systems

Problem solved (?)

- balanced, randomized trees use O(lg N) comparisons

Is lg N required?

- no (and yes)

Are comparisons necessary?

- no

ST.h

```
void STinit();
void STinsert(Item);
Item STsearch(Key);
 int STempty();
```

ST interface in C

ST implementations cost summary

"Guaranteed" asymptotic costs for an ST with N items

	insert	search
unordered array	1	N
BST	N	N
randomized BST*	lg N	lg N
red-black BST	lg N	lg N

* assumes system can produce "random" numbers

Can we do better?

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key)

Hash function

- method for computing table index from key

Collision resolution strategy

- algorithm and data structure to handle
  two keys that hash to the same index

Classic time-space tradeoff

- no space limitation:
  trivial hash function with key as address
- no time limitation:
  trivial collision resolution: sequential search
- limitations on both time and space (the real world)
  hashing

Hash function

Goal: random map (each table position equally likely for each key)

Treat key as integer, use prime table size M

- hash function: `h(K) = K mod M`

Ex: 4-char keys, table size 101

binary	01100001		01100010		01100011		01100100	
hex	6	1	6	2	6	3	6	4
ascii		a		b		c		d

26^4 ~ .5 million different 4-char keys
101 values
~50,000 keys per value

Huge number of keys, small table: most collide!

abcd hashes to 11

```
0x61626364 = 1633831724
16338831724 % 101 = 11
```

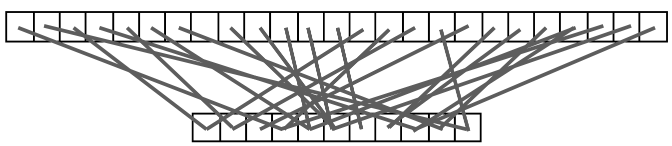
dcba hashes to 57

```
0x64636261 = 1684234849
1633883172 % 101 = 57
```

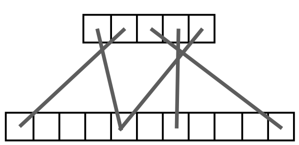
abbc also hashes to 57

```
0x61626263 = 1633837667
1633837667 % 101 = 57
```

25 items, 11 table positions
~2 items per table position



5 items, 11 table positions
~ .5 items per table position



Hash function (long keys)

Goal: random map (each table position equally likely for each key)

Treat key as long integer, use prime table size M

- use same hash function: `h(K) = K mod M`
- compute value with Horner's method

0x61

Ex: `abcd` hashes to `11`

```
0x61626364 = 256*(256*(256*97+98)+99)+100
16338831724 % 101 = 11
```

numbers too big?

OK to take mod after each op

```
256*97+98  = 24930 % 101 = 84
256*84+99  = 21603 % 101 = 90
256*90+100 = 23140 % 101 = 11
```

... can continue indefinitely, for any length key

How much work to hash a string of length N?

N add, multiply, and mod ops

hash.c

scramble by using 117 instead of 256

```
int hash(char *v, int M)
  { int h, a = 117;
    for (h = 0; *v != '\0'; v++)
      h = (a*h + *v) % M;
    return h;
  }
```

hash function for strings in C

Uniform hashing: use a different random multiplier for each digit.

Collision Resolution

Two approaches

Separate chaining

- M much smaller than N
- ~N/M keys per table position
- put keys that collide in a list
- need to search lists

Open addressing (linear probing, double hashing)

- M much larger than N
- plenty of empty table slots
- when a new key collides, find an empty slot
- complex collision patterns

Separate chaining

Hash to an array of linked lists

Hash

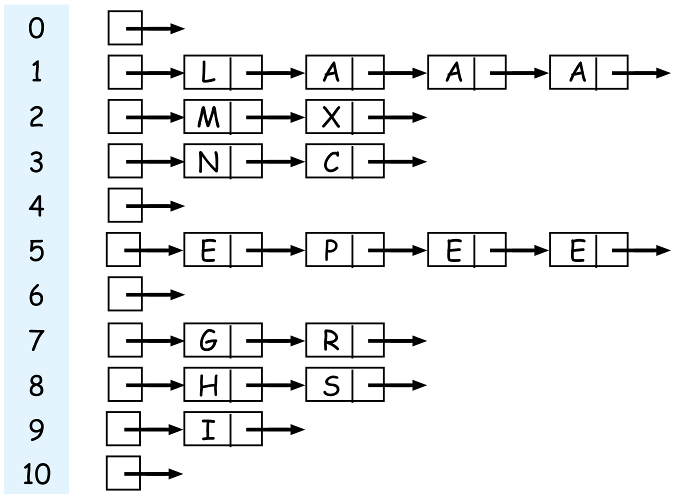
- map key to value between 0 and M-1

Array

- constant-time access to list with key

Linked lists

- constant-time insert
- search through list using elementary algorithm



Trivial: average list length is N/M

Worst: all keys hash to same list

Theorem (from classical probability theory):

Probability that any list length is > tN/M is exponentially small in t

M too large: too many empty array entries

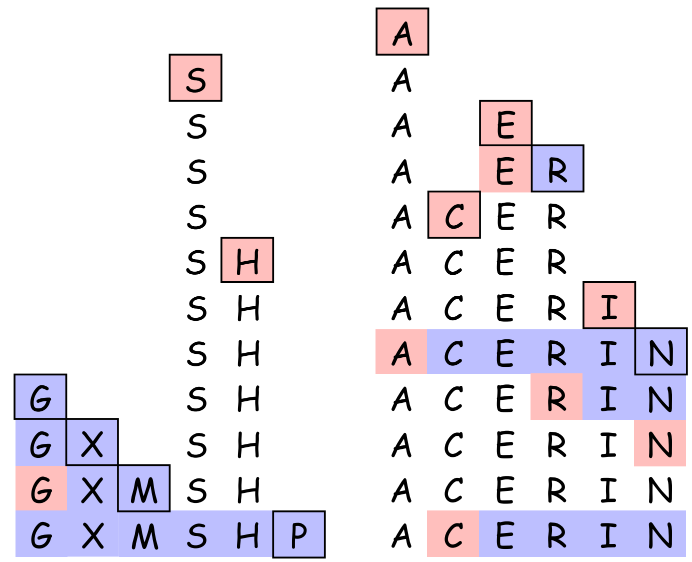
M too small: lists too long

Typical choice M ~ N/10: constant-time search/insert

Guarantee depends on hash function being random map

Linear probing

Hash to a large array of items, use sequential search within clusters



Hash

- map key to value between 0 and M-1

Large array

- at least twice as many slots as items

Cluster

- contiguous block of items
- search through cluster using elementary algorithm for arrays

Trivial: average list length is $N/M \equiv \alpha$

Worst: all keys hash to same list

Theorem (beyond classical probability theory):

insert: $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$

search: $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$

M too large: too many empty array entries

M too small: clusters coalesce

Typical choice M ~ 2N: constant-time search/insert

Guarantees depend on hash function being random map

Double hashing

Avoid clustering by using second hash to compute skip for search

Hash

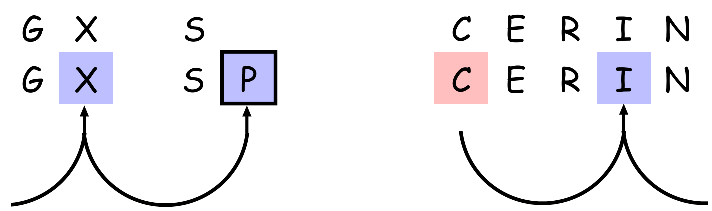
- map key to array index between 0 and M-1

Second hash

- map key to nonzero skip value (best if relatively prime to M)
- quick hack OK
  Ex: `1 + (k mod 97)`

Avoids clustering

- skip values give different search paths for keys that collide



Trivial: average list length is $N/M \equiv \alpha$

Worst: all keys hash to same list and same skip

Theorem (deep):

insert: $\frac{1}{1-\alpha}$

search: $\frac{1}{\alpha} \ln(1+\alpha)$

Typical choice M ~ 2N: constant-time search/insert

Guarantees depend on hash functions being random map

Disadvantage: delete cumbersome to implement

Double hashing ST implementation

```
static Item *st;
```

code assumes Items are pointers, initialized to NULL

insert

```
void STinsert(Item x)
  { Key v = ITEMkey(x);
    int i = hash(v, M);
    int skip = hashtwo(v, M);
    while (st[i] != NULL) i = (i+skip) % M;
    st[i] = x; N++;
  }
```

linear probing: take skip = 1

probe loop

search

```
Item STsearch(Key v)
  {
    int i = hash(v, M);
    int skip = hashtwo(v, M);
    while (st[i] != NULL)
      if eq(v, ITEMkey(st[i])) return st[i];
      else i = (i+skip) % M;
    return NULL;
  }
```

probe loop

Hashing tradeoffs

Separate chaining vs. linear probing/double hashing

- space for links vs. empty table slots
- small table + linked allocation vs. big coherant array

Linear probing vs. double hashing

load factor (α)

		50%	66%	75%	90%
linear probing	search	1.5	2.0	3.0	5.5
	insert	2.5	5.0	8.5	55.5
double hashing	search	1.4	1.6	1.8	2.6
	insert	1.5	2.0	3.0	5.5

Hashing vs. red-black BSTs

- arithmetic to compute hash vs. comparison
- hashing performance guarantee is weaker (but with simpler code)
- easier to support other ST ADT operations with BSTs

ST implementations cost summary

"Guaranteed" asymptotic costs for an ST with N items

	insert	search	delete	find kth largest	sort	join
unordered array	1	N	1	N	NlgN	N
BST	N	N	N	N	N	N
randomized BST*	lg N	lg N	lg N	lgN	N	lgN
red-black BST	lg N	lg N	lg N	lg N	N	lg N
hashing*	1	1	1	N	NlgN	N

Not really: need lgN bits to distinguish N keys

* assumes system can produce "random" numbers

* assumes our hash functions can produce random values for all keys

tough to be sure....

Can we do better?