

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: $h(K) = K \bmod M$

Ex: 4-char keys, table size 101

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

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

Huge number of keys, small table: **most collide!**

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

abcd hashes to **11**

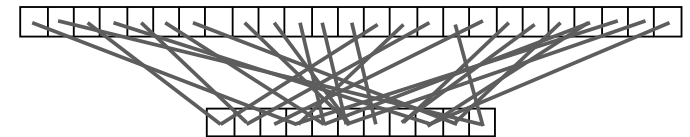
$$\begin{aligned} 0x61626364 &= 1633831724 \\ 1633831724 \% 101 &= 11 \end{aligned}$$

dcba hashes to **57**

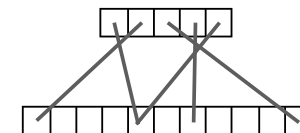
$$\begin{aligned} 0x64636261 &= 1684234849 \\ 1684234849 \% 101 &= 57 \end{aligned}$$

abbc also hashes to **57**

$$\begin{aligned} 0x61626263 &= 1633837667 \\ 1633837667 \% 101 &= 57 \end{aligned}$$



5 items, 11 table positions
 $\sim .5$ 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: $h(K) = K \bmod M$
- compute value with Horner's method

Ex: **abcd** hashes to **11**

0x61



$$0x61626364 = 256*(256*(256*97+98)+99)+100$$

$$16338831724 \% 101 = 11$$

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

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.

How much work to hash a string of length N ?

N add, multiply, and mod ops

Collision Resolution

Two approaches

Separate chaining

- M much smaller than N
- $\sim N/M$ 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

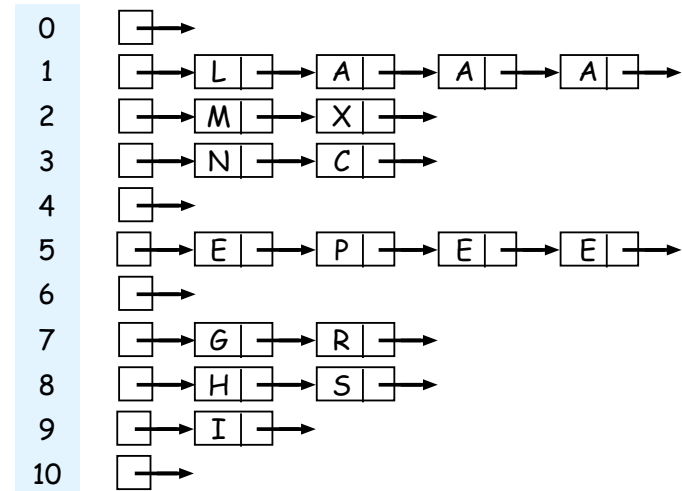
Linked lists

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

M too large: too many empty array entries

M too small: lists too long

Typical choice $M \sim N/10$: **constant-time** search/insert



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



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

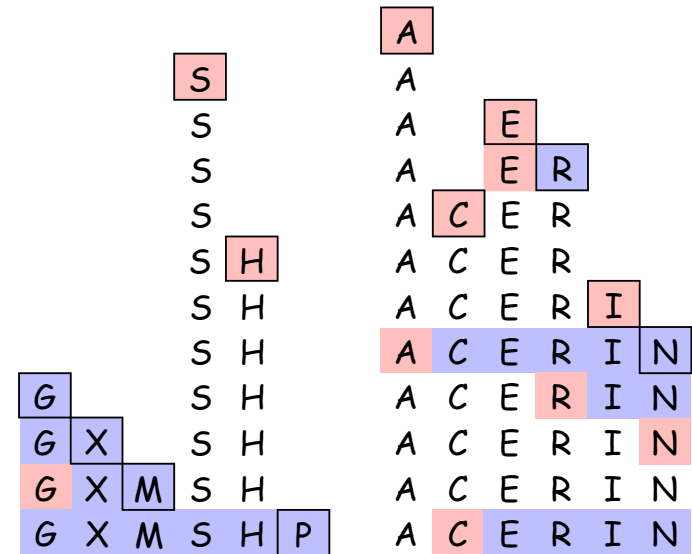
Cluster

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

M too large: too many empty array entries

M too small: clusters *coalesce*

Typical choice $M \sim 2N$: *constant-time* search/insert



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

Worst: all keys hash to same list

Theorem (beyond classical probability theory):

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

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

↑
← 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

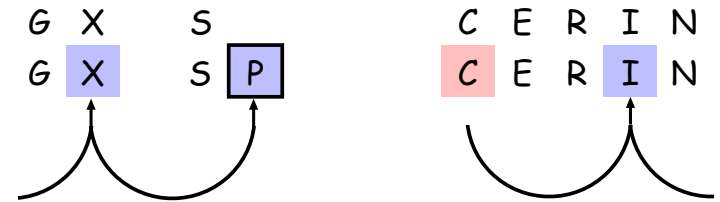
- map key to nonzero skip value (best if relatively prime to M)
- quick hack OK
Ex: $1 + (k \bmod 97)$

Avoids clustering

- skip values give different search paths for keys that collide

Typical choice $M \sim 2N$: **constant-time** search/insert

Disadvantage: **delete** cumbersome to implement



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

Worst: all keys hash to same list and same skip

Theorem (deep):

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

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



Guarantees depend on hash functions being random map

Double hashing ST implementation

```
static Item *st;
```

← code assumes Items are pointers, initialized to NULL

```
void STinsert(Item x)
```

insert

```
{ Key v = ITEMkey(x);
```

```
  int i = hash(v, M);
```

```
  int skip = hashtwo(v, M);
```

linear probing:
take skip = 1

```
  while (st[i] != NULL) i = (i+skip) % M; probe loop
```

```
  st[i] = x; N++;
```

```
}
```

```
Item STsearch(Key v)
```

search

```
{
```

```
  int i = hash(v, M);
```

```
  int skip = hashtwo(v, M);
```

```
  while (st[i] != NULL) probe loop
```

```
    if eq(v, ITEMkey(st[i])) return st[i];
```

```
    else i = (i+skip) % M;
```

```
  return NULL;
```

```
}
```

Hashing tradeoffs

Separate chaining vs. linear probing/double hashing

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

Linear probing vs. double hashing

		load factor (α)			
		50%	66%	75%	90%
linear	search	1.5	2.0	3.0	5.5
probing	insert	2.5	5.0	8.5	55.5
double	search	1.4	1.6	1.8	2.6
hashing	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	$N \lg N$	N
BST	N	N	N	N	N	N
randomized BST*	$\lg N$	$\lg N$	$\lg N$	$\lg N$	N	$\lg N$
red-black BST	$\lg N$	$\lg N$	$\lg N$	$\lg N$	N	$\lg N$
hashing*	1	1	1	N	$N \lg N$	N

Not really: need $\lg N$ bits to distinguish N keys

- * assumes system can produce "random" numbers
- * assumes **our** hash functions can produce random values **for all keys**

Can we do better?

tough to be sure....