COS 226 Lecture 9: Hashing

Symbol Table, Dictionary

    records with keys

    INSERT

    SEARCH

Balanced trees, randomized trees

    use O(lgN) comparisons

Is lgN required?

    (no, and yes)

Are comparisons necessary?

    (no)

Hashing: basic plan

Save keys in a table, at a location determined
by 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

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
  hashing

Hash Functions

Short keys (fit in a word)

    treat key as integer
    h(K) = K mod M

Use prime table size

Ex: four-character keys, table size 101

```
bin    01100001011000100110001101100100
hex        6    1    6    2    6    3    6    4
ascii           a         b         c         d
```

    0x61626364 = 1633831724
    16338831724 % 101 = 11
    Key "abcd" hashes to 11

    0x64636261 = 1684234849
    1633883172 % 101 = 57
    Key "dcba" hashes to 57

"Collision"

    0x61626263 = 1633837667
    1633837667 % 101 = 57
    Key "abbc" also hashes to 57

Obvious point:

    millions of keys, small table: most collide!

Hash Functions (continued)

Long keys (strings)

  treat key as integer

  h(K) = K mod M

Multiprecision arithmetic calculation

Use Horner's method

Ex: (check with 4 chars; works for any length)

hex	**6**	**1**	**6**	**2**	**6**	**3**	**6**	**4**
ascii		**a**		**b**		**c**		**d**

```
0x61626364 = 256*(256*(256*97+98)+99)+100
take mod after each multiplication
    256*97+98  = 24930 % 101 = 84
    256*84+99  = 21603 % 101 = 90
    256*90+100 = 23140 % 101 = 11
```

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

scramble by replacing 256 by 117

Uniform hashing:

  use a different random value for each digit

Collisions

Table size M

How many insertions until the first collision?

BIRTHDAY PARADOX (classical probability theory)

Assume hash function "random"

Expected insertions to first collision

	M	sqrt(pi M/2)
.	**100**	**12**
.	**1000**	**40**
.	**10000**	**125**

N keys

Option 1:

Allow N >> M: put keys hashing to i in a list

about N/M keys per list

Option 2:

Keep N < M: put keys somewhere in table

complex collision pattern

Collisions (continued)

Experiment 1:

generate random probes between 0 and 100

84 35 45 32 89 1 58 16 38 69 5 90 16 16 53 61 ...

collision at 13th as predicted

Experiment 2:

use hash function to scatter 4-char keys

bcba	**47**	**dbcb**	**24**
bddc	**43**	**dcbd**	**60**
dabc	**85**	**dbab**	**17**
dbdb	**78**	**bccc**	**2**
ccad	**1**	**dbdd**	**80**
babb	**74**	**adbc**	**31**
cdcc	**95**	**addd**	**39**
bcbd	**50**	**bcda**	**55**
bdac	**83**	**abad**	**4**
baca	**26**	**cada**	**85**
dabb	**84**	**dabd**	**86**

collision after 20 probes

still as predicted (standard dev. not small)

Separate chaining

Simple, practical, widely used

Cuts search time by a factor of M
     over sequential search

Method: M linked lists, one for each table slot

```
.     0:    *
.     1:    L    A    A    A    *
.     2:    M    X    *
.     3:    N    C    *
.     4:    *
.     5:    E    P    E    E    *
.     6:    *
.     7:    G    R    *
.     8:    H    S    *
.     9:    I    *
.    10:    *
```

Insert cost: 1

Avg. search cost (successful): N/2M

Avg. search cost (unsuccessful): N/M

M large: CONSTANT avg. search time
     independent of how keys are distributed (!)

Worst case: N ("probabilistically" unlikely)

Keep lists sorted?
     increases insert time to N/2M
     cuts unsuccessful search time to N/2M

Linear Probing

No links, keep everything in table

Method: start linear search at hash position

(stop when empty position hit)

Still get O(1) avg. search time if table sparse

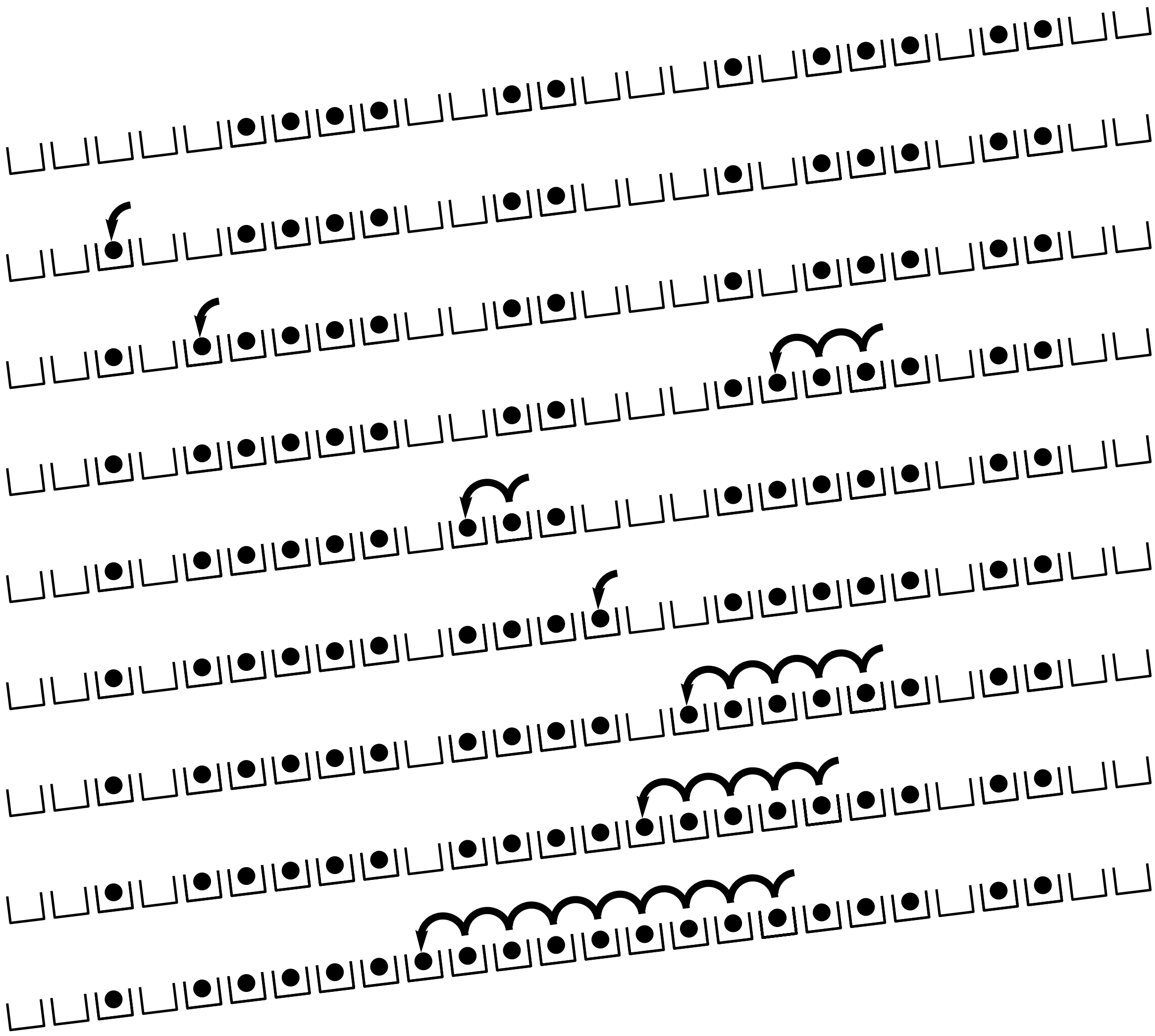
```
void STinit(int max)
  { int i;
    N = 0; M = 2*max;
    st = malloc(M*sizeof(Item));
    for (i = 0; i < M; i++) st[i] = NULLitem;
  }
void STinsert(Item item)
  { Key v = key(item);
    int i = hash(v, M);
    while (!null(i)) i = (i+1) % M;
    st[i] = item; N++;
  }
Item STsearch(Key v)
  { int i = hash(v, M);
    while (!null(i))
      if eq(v, key(st[i])) return st[i];
      else i = (i+1) % M;
    return NULLitem;
  }
```

Very sparse table: like separate chaining

As table fills up: CLUSTERING occurs

(infinite loop on full table)

Linear probing performance



CLUSTERING

bad phenomenon: items clump together

long clusters tend to get longer

avg. search cost grows to M as table fills

Precise analysis very difficult.

THM (Knuth)

Insert cost: approx. (1+ 1/(1-N/M)^2)/2

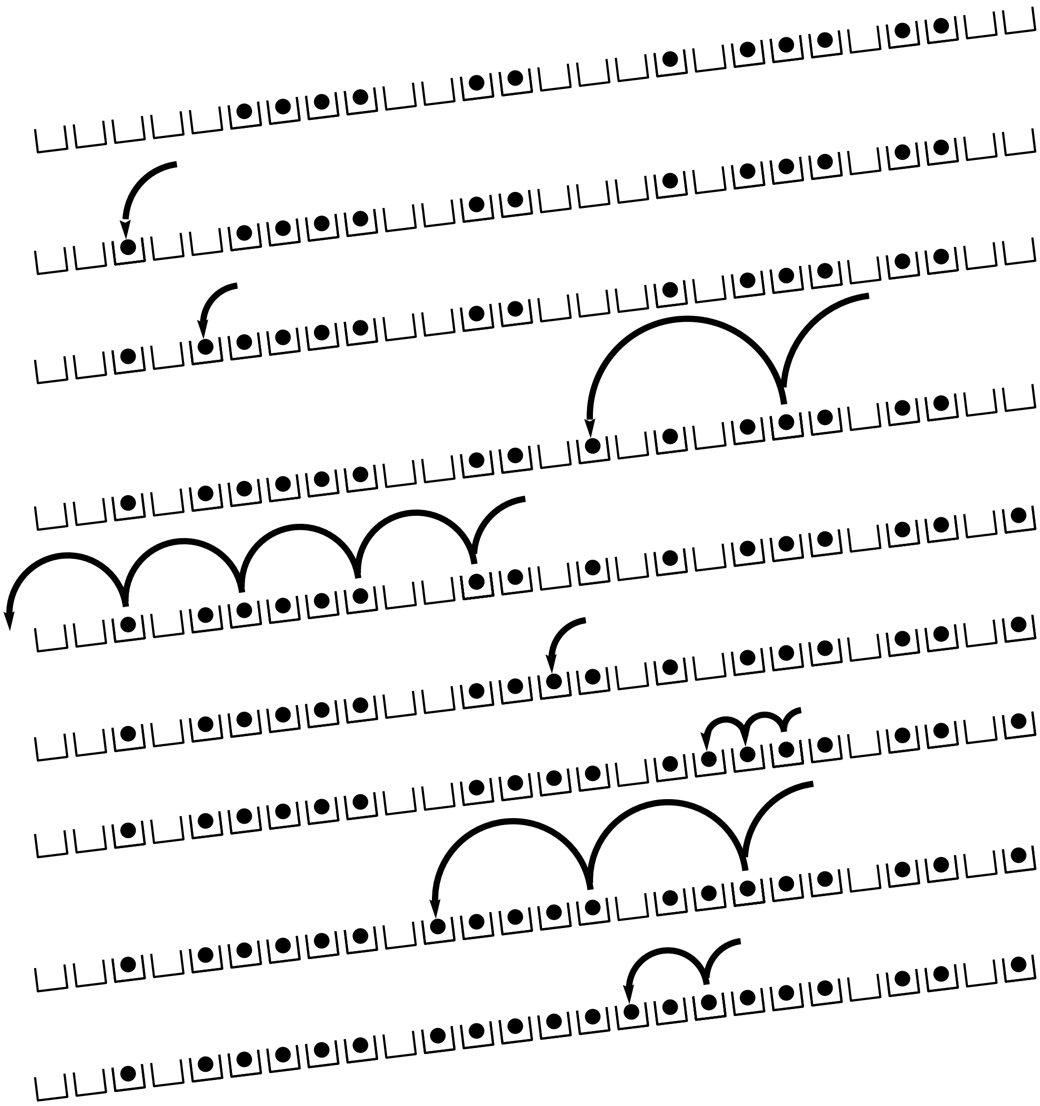
Search cost (hit): approx. (1+ 1/(1-N/M))/2

Search cost (miss): same as insert

Too slow when table gets 70%-80% full

Double Hashing

Avoid clustering by using 2nd hash
to compute increment for seq. search



Analysis extremely difficult

Roughly equivalent to ideal (random probe)

THM (Guibas-Szemeredi)

Insert cost: approx. 1/(1-N/M)

Search cost (hit): approx. ln(1+N/M)/(N/M)

Search cost (miss): same as insert

Not too slow until table gets 90%-95% full

Amortized analysis of algorithms

Measure running time for X operations by
(total cost of all X operations)/ X

Ex:
- insert N elements in a heap:
  - (lg1 + lg2 + ... + lgN) / N = lgN + O(1)
- insert N elements in a binomial queue:
  - (1*N/2 + 2*N/4 + 3*N/8 +...)/N < 2

Hashing:
- grow table while keeping search cost O(1)
- when number of keys in table doubles
  - rebuild to double the size of the table

Ex: separate chaining, avg search cost < 2
- 4M keys in table of size M
- proof by induction: amortized cost < 2
  - cost to build: x*4M
  - cost to rebuild to new table size 2M: 4M
  - amortized cost of first 8M insertions:
    - (x*4M + 4M + 4M)/8M
    - x/2 + 1 < x

Same argument works for other basic ADTs!
Ex: stacks, queues in arrays, double hashing

Another way to evaluate algorithms
- worst-case > average-case > amortized

Separate chaining vs. double hashing

Assume same amount space for keys, links
    pointers for long or variable-length keys

Space for separate chaining w/ rehashing
    4M keys (or links to keys)
    4M links in nodes
    Total space: 9M words for 4M items
    Avg search time: 2

Double hashing in same space
    4M items
    table size 9M
    avg search time: 1/(1-4/9) = 1.8
    10% faster

Double hashing in same time
    4M items
    avg search time 2
    space needed: 8M words (1/(1-4/8) = 2)
    11% less space

Separate chaining advantages
    idiot-proof (doesn't break)
    no large chunks of memory (is that good?)
    could avoid links to nodes

Other ST ADT operations

Deletion

- Separate chaining: trivial
- Linear probing: rehash keys in cluster
  - or use indirect method (see below)

- Double hashing: no easy direct method
  - mark deleted nodes as "deadwood"
  - rebuild periodically to clear deadwood

Sort, find kth largest

- Separate chaining w/ sorted lists
- Linear probing/double hashing
  - have to do full sort

Join

- Separate chaining: easy
- Linear probing/double hashing:
  - rehash whole table

Reasons not to use hashing

Hashing implements ST ADT search and insert
in constant time.

Why use anything else?

* no performance guarantee

* too much arithmetic on long keys

* takes extra space

* doesn’t support all ADT ops efficiently

* compare abstraction works for partial order
(searching without keys)

Other hashing variants

Perfect hashing

- fixed set of keys
- hash function with no collisions
- good hack for small tables
- not practical for large tables
- totally static

Coalesced hashing

- properly account for link space
- mix hash table, storage allocation

Ordered hashing

- cut costs in half as with ordered lists

Brent's variation

- guarantee constant search cost
- up to M insert cost