Hash Tables

What is a Dictionary?

Container class

- Stores key-element pairs

Allows "look-up" (find) operation

Allows insertion/removal of elements

May be *unordered* or *ordered*

Dictionary Keys

Must support equality operator

- For ordered dictionary, also support comparator operator
  - useful for finding neighboring elements

Keys sometimes required to be unique

In this case, referred to as a MAP

Dictionary Examples

Natural language dictionary

- word is key
- element contains word, definition, pronunciation, etc.

Web pages (a Map)

- URL is key
- html or other file is element

Any typical database (e.g. student record, also a map)

- has one or more search keys
- each key may require own organizational dictionary

Map ADT

Size(), isEmpty()

get(k): Return element with key k or null

put(k,e): Insert element e with key k; if this key exists replace element and return old element

remove(k): Remove element with key k; if no such entry return null

keySet(), values(), entrySet(): iterators over keys and values and entries (key-value pairs) in Map

Implemented using Trees (so far)

Hash Table

Provides efficient implementation of **unordered** dictionary

- Insert, remove, and find all $O(1)$ expected time

Bucket array

- Provides storage for elements

Hash function

- Maps keys to buckets (ranks)
- For each operation, evaluate hash function to find location of item

Bucket Array

Each array element holds 1 or more dictionary elements

Capacity is number of array elements

Load is percent of capacity used

- N is capacity of hash table
- n is size of dictionary
- n/N is load of hash table

Collision is mapping of multiple dictionary elements to the same array element

Simplest Hash Table

Keys are unique integers in range [0, N-1]

Trivial hash function

- $h(k) = k$

Uses $O(N)$ space (can be very large)

- okay if $N = O(n)$
- bad if key can be any 32-bit integer
  - table has 2^{32} entries = 4 gigaentries

find( ), insert( ), and remove( ) all take $O(1)$ time

Hash Function

Maps each key to an array rank

- *h*(*k*): K → R
- array rank is integer in [0, *N*-1]

Decomposed into two parts

- *hash code* generation
  - converts key to an integer
- *compression map*
  - converts integer hash code to valid rank
- *h*(*k*) = *cm*(*hc*(*k*))

"Good" hash function

Want to "spread out" values to avoid collisions

Ideally, keys act as random distribution of ranks

- Probability($h(k) = i$) = $1/N$ for all i in $[0, N\text{-}1]$
- Expected keys in bucket i is n/N
  - this is $O(1)$ if $n = O(N)$

If no collision, operations are $O(1)$

- so *expected* time is $O(1)$ for all operations

Note: worst case time is still $O(n)$

Generating Hash Codes: Java's Object.hashCode()

generates integer for any object

generates same integer for two objects as long as equals( ) method evaluates to true

- different instances with same value are not equal according to Object.equals( )
  - won't always give expected hashing behavior

exact method is implementation dependent

Generating Hash Codes: Cast to Integer

Works well if key is byte, short, or char type

- can use Float.floatToIntBits() for floats

Disadvantages

- High order bits ignored for longs/doubles
  - May result in collisions
- Cannot handle more complex keys

Generating Hash Codes: Summing Components

Add up multiple integers to get a single integer

- Ignore overflows
- $hc(x_0, x_1, x_2, \ldots, x_{k-1}) = \sum_{i=0}^{k-1} x_i$

Examples

- Long or double may be converted to two ints (high order and low order) and summed
- Strings may be broken into multiple characters and summed

Disadvantage

- Ordering of integers is ignored
  - May result in collisions

Generating Hash Codes: Polynomial Hash Codes

Multiply each component by some constant to a power

- $hc(x_0, x_1, x_2, \ldots, x_{k-1}) = \sum_{i=0}^{k-1} a^i x_i$

  $= x_0 + a(x_1 + a(x_2 + \ldots x_{k-1}))\ldots)$
- Makes hash code dependent on order of components

Disadvantages

- *k*-1 multiplies in hash evaluation
- Choice of *a* makes big difference in "goodness" of hash function

Generating Hash Codes: Cyclic Shift

Cyclic Shift Hash Codes

- Rotates bits of current code by some number of positions before adding each new component
- $hc(x_0, x_1, x_2, \ldots, x_{k-1}) =$
  $\text{rotate}(x_{k-1} + \text{rotate}(x_{k-2} + \ldots(x_1 + \text{rotate}(x_0))\ldots))$
- no multiplication
  - only addition and bitwise shifts and ORs

Disadvantages

- Choice of rotation size still makes big difference in "goodness"

Compression Maps

Division Method

- $h(k) = |k| \bmod N$
- N works best if it is a prime number
- Even then, multiples of N map to same position
    - $h(iN) = 0$, $h(iN+j) = j \bmod N$

MAD (multiply, add, and divide) Method

- $h(k) = |ak+b| \bmod N$
    - $h(iN) = |aiN + b| \bmod N = b \bmod N$
    - $h(iN+j) = |aiN + aj + b| \bmod N$
      $= |aj + b| \bmod N$
- Not clear that this is much better...

Collision Handling: Chaining

For each bucket, store a sequence of elements that map to the bucket

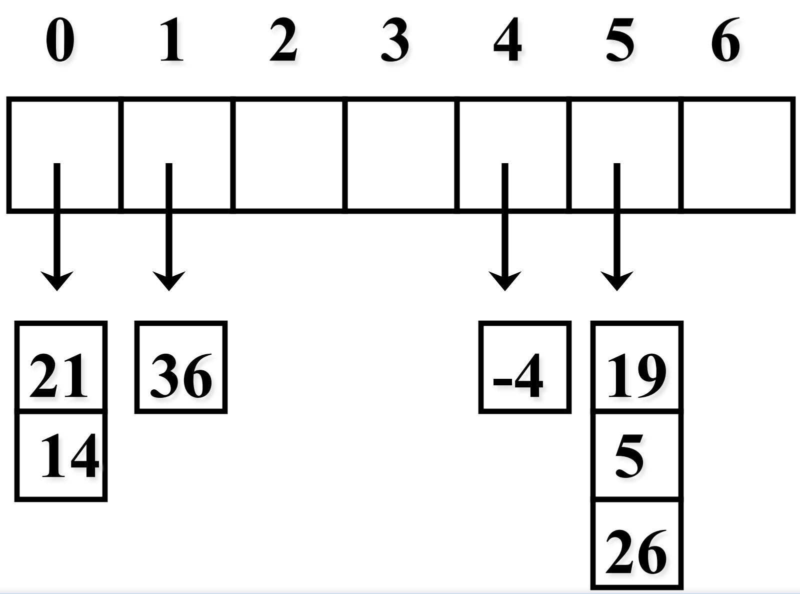
- effectively a much smaller, auxiliary dictionary

Linearly search sequence to find correct element

Chaining Example

$$N = 7, \quad h(k) = |k| \bmod N$$

Insert 19 36 5 21 -4 26 14 **(load = 1)**



Map Methods with Separate Chaining used for Collisions

- Delegate operations to a list-based map at each cell:

Algorithm get(k):
Output: The value associated with the key k in the map, or **null** if there is no
entry with key equal to k in the map
return $A[h(k)]$.get(k) {delegate the get to the list-based map at $A[h(k)]$}

Algorithm put(k,v):
Output: If there is an existing entry in our map with key equal to k, then we
return its value (replacing it with v); otherwise, we return **null**
$t = A[h(k)]$.put(k,v) {delegate the put to the list-based map at $A[h(k)]$}
if t = **null then** {k is a new key}
$n = n + 1$
return t

Algorithm remove(k):
Output: The (removed) value associated with key k in the map, or **null** if there
is no entry with key equal to k in the map
$t = A[h(k)]$.remove(k) {delegate the remove to the list-based map at $A[h(k)]$}
if $t \neq$ **null then** {k was found}
$n = n - 1$
return t

Collision Handling: Probing Hash Tables

Store only 1 element per bucket

- No additional space, but requires smaller load

If multiple elements map to same bucket, use some method to find empty bucket

- Linear probing
  - $h'(k) = (\ h(k) + j\ ) \bmod N \quad j = 0, 1, 2, 3, \ldots$
    - » Keep adding 1 to rank to find empty bucket
- Quadratic probing
  - $h'(k) = (\ h(k) + j^2\ ) \bmod N \quad j = 0, 1, 2, 3, \ldots$
- Double hashing
  - $h'(k) = (\ h(k) + j * h''(k)\ ) \bmod N \quad j = 0, 1, 2, 3, \ldots$

Linear Probing

- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing Example

$$N = 7, \quad h(k) = |k| \bmod N$$

Insert 19 36 5 21 -4 26 14 **(load = 1)**

0	1	2	3	4	5	6
21	36	26	14	-4	19	5

Time for probing?

I(l) = 1/l ln 1/(1-l) where l is load factor

E.g. l = .75 ➔ 8-9 probes

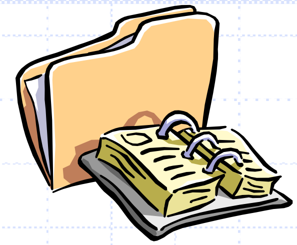
l = .90 ➔ 50 probes

Double Hashing

Note that in double-hashing, the second hash function cannot evaluate to zero!

hash'' (x) = R – (x mod R), R prime and R<N

Search with Linear Probing



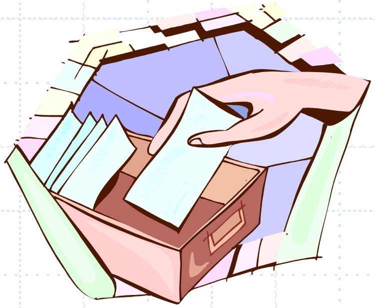
- Consider a hash table ***A*** that uses linear probing
- get(***k***)
  - We start at cell ***h***(***k***)
  - We probe consecutive locations until one of the following occurs
    - An item with key ***k*** is found, or
    - An empty cell is found, or
    - ***N*** cells have been unsuccessfully probed

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
         else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until  p = N
    return null
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- remove(***k***)
  - We search for an entry with key ***k***
  - If such an entry (***k, o***) is found, we replace it with the special item ***AVAILABLE*** and we return element ***o***
  - Else, we return ***null***
- put(***k, o***)
  - We throw an exception if the table is full
  - We start at cell ***h***(***k***)
  - We probe consecutive cells until one of the following occurs
    - A cell ***i*** is found that is either empty or stores ***AVAILABLE***, or
    - ***N*** cells have been unsuccessfully probed
  - We store entry (***k, o***) in cell ***i***

Double Hashing



- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

$$(i + jd(k)) \bmod N$$

for $j = 0, 1, \ldots, N - 1$

- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Common choice of compression function for the secondary hash function:

$$d_2(k) = q - k \bmod q$$

- where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are

$$1, 2, \ldots, q$$

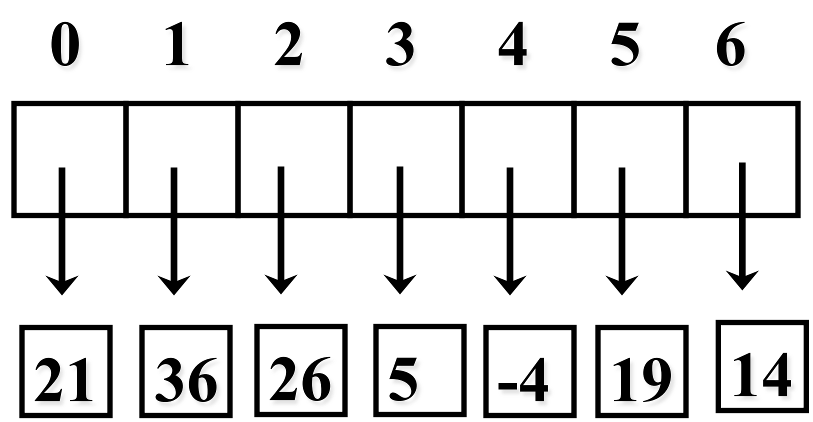
Double Hashing Example

$$N = 7, \quad h(k) = |k| \bmod N$$

$$R = 5, \quad h''(k) = R - k \bmod R$$

Insert 19 36 5 21 -4 26 14

(load = 1)



Other Open Addressing Difficulties

Searching

- For NO_SUCH_KEY, must search until empty bucket found

Removing

- Cannot just empty the bucket
  - —could disconnect colliding keys
- Easiest method is setting with special DELETED_KEY sentinal
  - —insert( ) can reuse bucket
  - —find( ) must continue searching beyond bucket

Rehashing

When load of hash table gets too large

- Allocate new hash table
- Generate new hash function
- Re-hash old elements into new table
- Time cost may be amortized as in dynamic array
  - —must increase size by $O(n)$ each time

Extendible Hashing

Recall disk access is expensive; probing is thus expensive

B-tree reduced disk access but still grow in size

Extendible hashing is a mashup of B-trees and hashing

Extendible Hashing

- external storage
- N records in total to store,
- M records in one disk block

No more than two blocks are examined.

Extendible Hashing

Idea:

- Keys are grouped according to the first m bits in their code.
- Each group is stored in one disk block.
- If some block becomes full, each group is split into two , and m+1 bits are considered to determine the location of a record.

Example

4 disk blocks, each can contain 3 records

4 groups of keys according to the first two bits

directory

00	01	10	11
00010	01001	10001	11000
00100	01010	10100	11010
	01100		

Modified from lydia.sinapova

Example (cont.)

New key to be inserted: 01011.
Block2 is full, so we start considering 3 bits

directory

000/001 (still on same block)	010	011	100/101	110/111
00010	01001	01100	10001	11000
----	01010		---	11010
00100	01011		10100	

Extendible Hashing

Size of the directory : 2^D

$2^D = \quad O(N^{(1+1/M)} / M)$

D - the number of bits considered.
N - number of records
M - number of disk blocks

Unordered Dictionary ADT

size, isEmpty

find(k): Return element with key k; else null

findAll(k): Iterator of all elements with key k

insert(k,e): Insert element e with key k; return entry

remove(k): Remove element with key k; return entry or null

entries(): iterator over all entries

Some interesting facts

Time in linear case is about ½ (1 + 1/(1-load^2))

Quadratic can always insert if less than half full

Assume not; think about the first N/2 possiblities and suppose that i^2%N = j^2%N and both i and j < N/2

i^2 – j^2 = 0 ➔ (i+j)(i-j) = 0 but i+ j < N, so cannot be equal to zero. So first n/2 alternatives are unique