Chapter 5
Hashing

Introduction

- hashing performs basic operations, such as insertion, deletion, and finds in constant average time
  - better than other ADTs we've seen so far

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Hashing

- a hash table is merely an array of some fixed size
- hashing converts search keys into locations in a hash table
  - searching on the key becomes something like array lookup
- hashing is typically a many-to-one map: multiple keys are mapped to the same array index
  - mapping multiple keys to the same position results in a collision that must be resolved
- two parts to hashing:
  - a hash function, which transforms keys into array indices
  - a collision resolution procedure

Hashing Functions

- let K be the set of search keys
- hash functions map K into the set of M slots in the hash table

$$h: K \rightarrow \{0, 1, \dots, M - 1\}$$

  - ideally, h distributes K uniformly over the slots of the hash table, to minimize collisions
  - if we are hashing N items, we want the number of items hashed to each location to be close to N/M
- example: Library of Congress Classification System
  - hash function if we look at the first part of the call numbers (e.g., E470, PN1995)
    - collision resolution involves going to the stacks and looking through the books
    - almost all of CS is hashed to QA75 and QA76 (BAD)

Hashing Functions

- suppose we are storing a set of nonnegative integers
- given M, we can obtain hash values between 0 and $M - 1$ with the hash function

$$h(k) = k \,\%\, M$$

  - remainder when k is divided by M
  - fast operation, but we need to be careful when choosing M
- example: if $M = 2^p$, $h(k)$ is just the p lowest-order bits of k
  - are all the hash values equally likely?
- choosing M to be a prime not too close to a power of 2 works well in practice

Hashing Functions

- we can also use the hash function below for floating point numbers if we interpret the bits as an <u>integer</u>

$$h(k) = k \,\%\, M$$

  - two ways to do this in C, assuming **`long int`** and **`double`** types have the same length
  - first method uses C <u>pointers</u> to accomplish this task

```
unsigned long *k;
double x;
k = (unsigned long *) &x;
long int hash = k % M;
```

Hashing Functions

- we can also use the hash function below for floating point numbers if we interpret the bits as an integer (cont.)
- second uses a <u>union</u>, which is a variable that can hold objects of different types and sizes

```
union {
 long int k;
 double x;
} u;

u.x = 3.1416;
long int hash = u.k % M;
```

Hashing Functions

– we can hash strings by combining a hash of each <u>character</u>

```
char *s = "hello!";
unsigned long hash = 0;

for (int i = 0; i < strlen(s); i++) {
  unsigned char w = s[i];
  hash = (R * hash + w) % M;
}
```

– R is an additional parameter we get to choose

– if R is larger than any character value, then this approach is what you would obtain if you treated the string as a base-R <u>integer</u>

Hashing Functions

- K&R suggest a slightly simpler hash function, corresponding to $R = 31$

```
char *s;
unsigned hash;

for (hash = 0; *s != '\0'; s++) {
  hash = 31 * hash + *s;
}
hash = hash % M;
```

  - Weiss suggests $R = 37$

Hashing Functions

- we can use the idea for strings if our search key has <u>multiple</u> parts, say, street, city, state:

```
hash = ((street * R + city) % M) * R + state) % M;
```

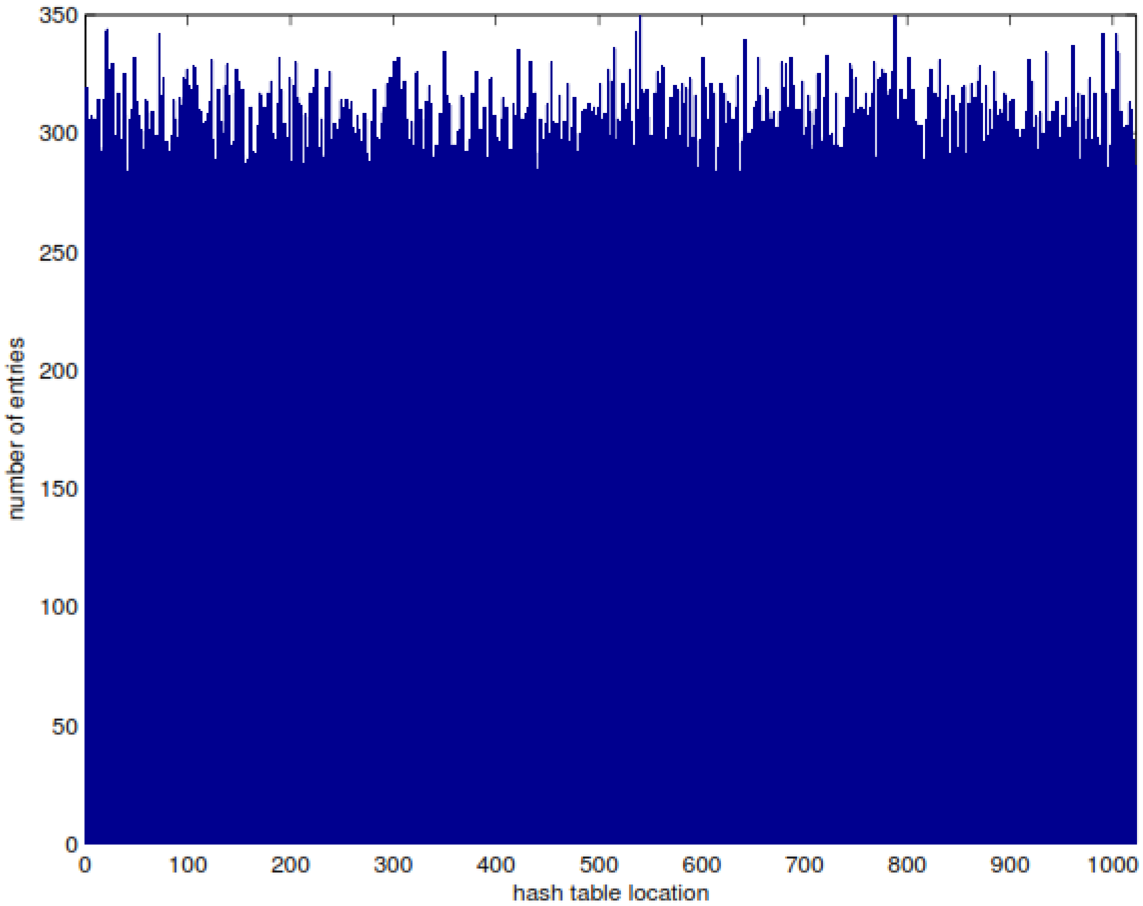
  - same ideas apply to hashing vectors

Hash Functions

- the choice of parameters can have a <u>dramatic</u> effect on the results of hashing
- compare the text string's hashing algorithm for different pairs of R and M
  - plot <u>histograms</u> of the number of words hashed to each hash table location; we use the American dictionary from the aspell program as data (305,089 words)

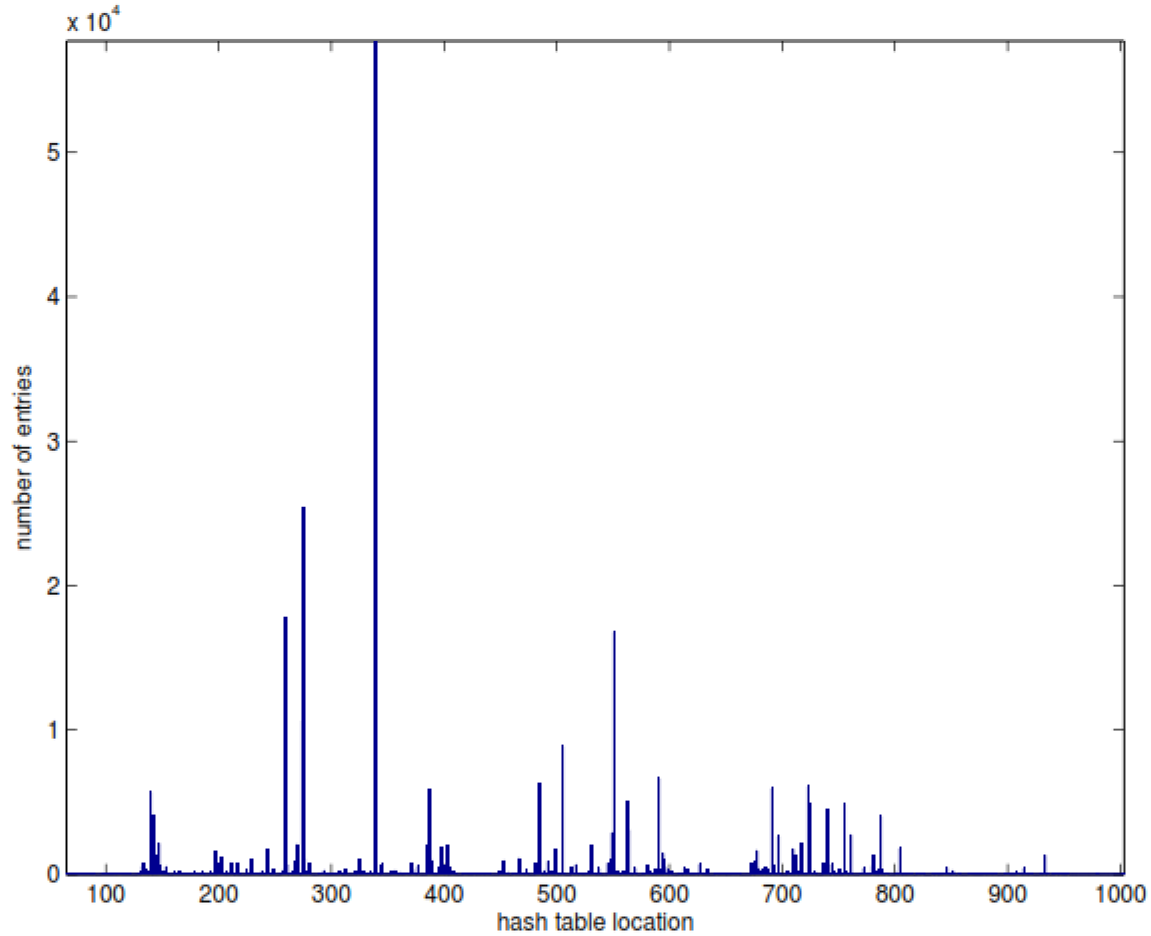
Hash Functions

- example: $R = 31,\ M = 1024$
- good: words are <u>evenly distributed</u>



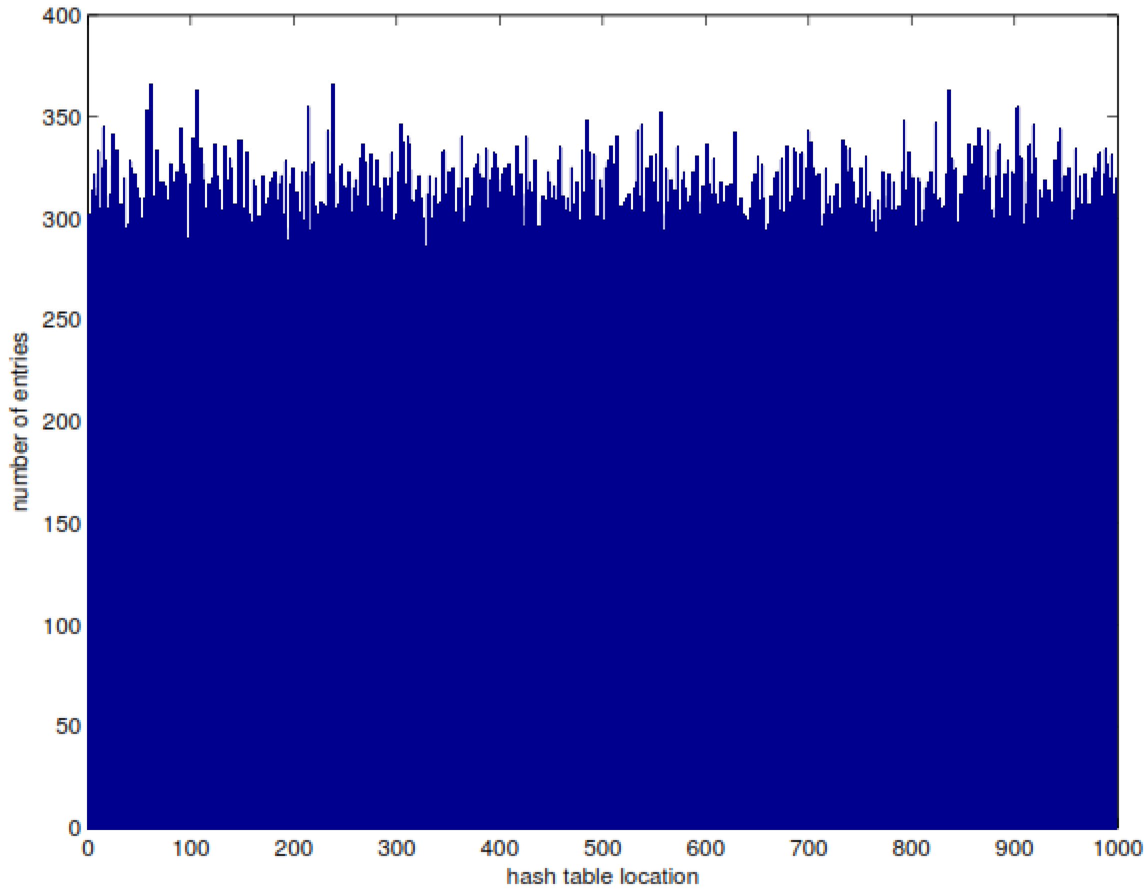
Hash Functions

- example: $R = 32,\ M = 1024$
- very bad



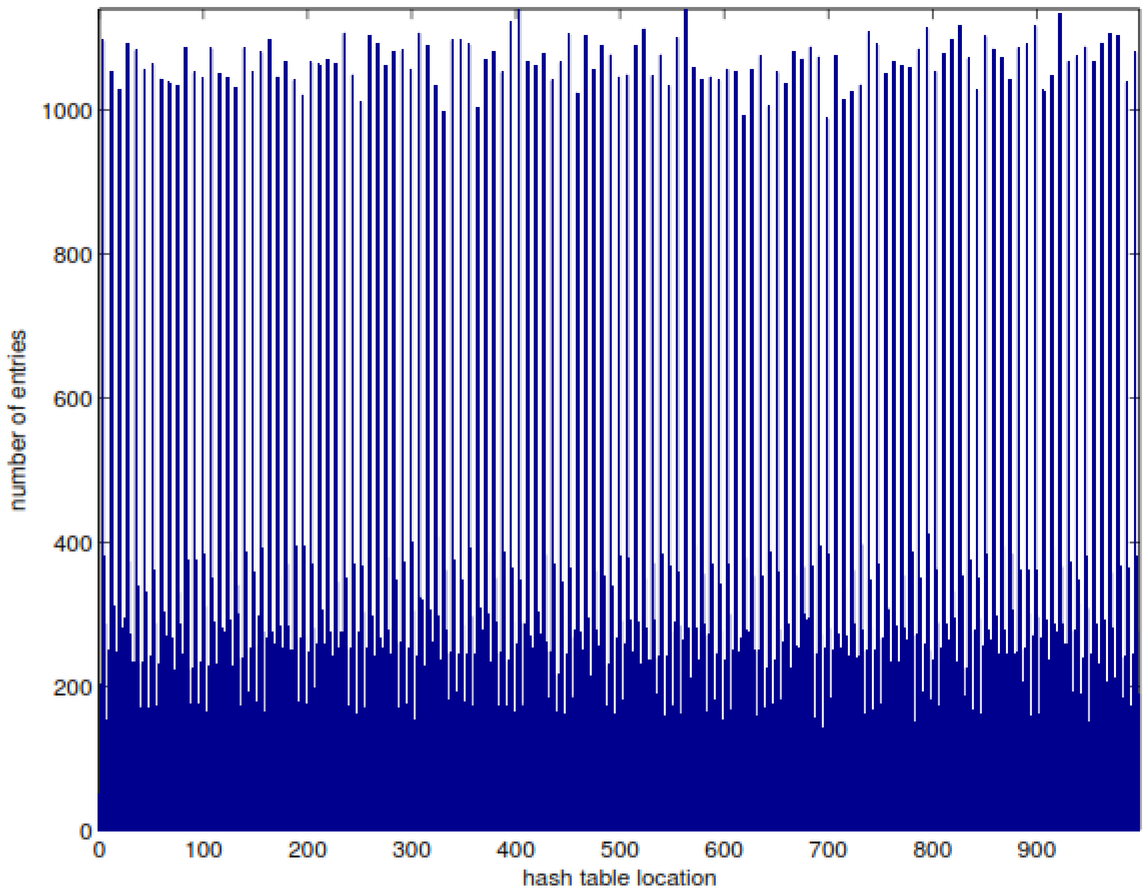
Hash Functions

- example: $R = 31,\ M = 1000$
- better



Hash Functions

- example: $R = 32, M = 1000$
- bad

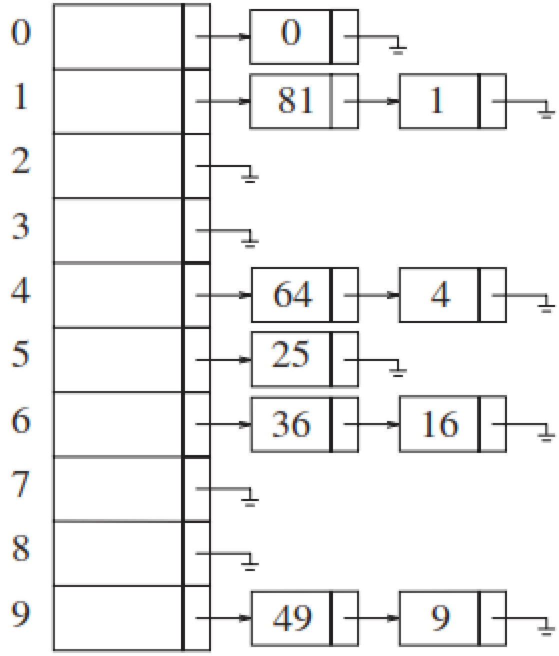


Collision Resolution

- hash table collision
  - occurs when elements hash to the same location in the table
  - various strategies for dealing with collision
    - separate chaining
    - open addressing
    - linear probing
    - other methods

Separate Chaining

- separate chaining
  - keep a list of all elements that hash to the same location
  - each location in the hash table is a linked list
  - example: first 10 squares



Separate Chaining

- insert, search, delete in lists
  - all proportional to <u>length</u> of linked list
  - insert
    - new elements can be inserted at <u>head</u> of list
    - duplicates can increment <u>counter</u>
- other structures could be used instead of lists
  - binary search tree
  - another hash table
- linked lists good if table is <u>large</u> and hash function is good

Separate Chaining

- how long are the linked lists in a hash table?
  - expected value: N/M where N is the number of keys and M is the size of the table
  - is it reasonable to assume the hash table would exhibit this behavior?
    - load factor $\lambda = N/M$
    - average length of a list $= \lambda$
    - time to search: constant time to evaluate the hash function + time to traverse the list
      - unsuccessful search: $1 + \lambda$
      - successful search: $1 + (\lambda/2)$

Separate Chaining

- observations
  - load factor more important than table size
  - general rule: make the table as large as the number of elements to be stored, $\lambda \approx 1$
  - keep table size prime to ensure good distribution

Separate Chaining

– declaration of hash structure

```
template <typename HashedObj>
class HashTable
{
  public:
    explicit HashTable( int size = 101 );

    bool contains( const HashedObj & x ) const;

    void makeEmpty( );
    bool insert( const HashedObj & x );
    bool insert( HashedObj && x );
    bool remove( const HashedObj & x );

  private:
    vector<list<HashedObj>> theLists;     // The array of Lists
    int currentSize;

    void rehash( );
    size_t myhash( const HashedObj & x ) const;
};
```

Separate Chaining

- hash member function

```
size_t myhash( const HashedObj & x ) const
{
    static hash<HashedObj> hf;
    return hf( x ) % theLists.size( );
}
```

Separate Chaining

– routines for separate chaining

```
void makeEmpty( )
{
    for( auto & thisList : theLists )
        thisList.clear( );
}

bool contains( const HashedObj & x ) const
{
    auto & whichList = theLists[ myhash( x ) ];
    return find( begin( whichList ), end( whichList ), x ) != end( whichList );
}

bool remove( const HashedObj & x )
{
    auto & whichList = theLists[ myhash( x ) ];
    auto itr = find( begin( whichList ), end( whichList ), x );

    if( itr == end( whichList ) )
        return false;

    whichList.erase( itr );
    --currentSize;
    return true;
}
```

Separate Chaining

– routines for separate chaining

```
bool insert( const HashedObj & x )
{
    auto & whichList = theLists[ myhash( x ) ];
    if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )
        return false;
    whichList.push_back( x );

        // Rehash; see Section 5.5
    if( ++currentSize > theLists.size( ) )
        rehash( );

    return true;
}
```

Open Addressing

- linked lists incur extra costs
  - time to allocate space for new cells
  - effort and complexity of defining second data structure
- a different collision strategy involves placing colliding keys in nearby empty slots
  - if a collision occurs, try successive cells until an empty one is found
  - bigger table size needed with $M > N$
  - load factor should be below $\lambda = 0.5$
- three common strategies
  - linear probing
  - quadratic probing
  - double hashing

Linear Probing

- linear probing insert operation
  - when k is hashed, if slot $h(k)$ is open, place k there
  - if there is a collision, then start looking for an empty slot starting with location $h(k)+1$ in the hash table, and proceed linearly through $h(k)+2$, ..., $m-1$, 0, 1, 2, …, $h(k)-1$ wrapping around the hash table, looking for an empty slot
- search operation is similar
- checking whether a table entry is vacant (or is one we seek) is called a probe

Linear Probing

- example: add 89, 18, 49, 58, 69 with $h(k) = k \;\%\; 10$ and $f(i) = i$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

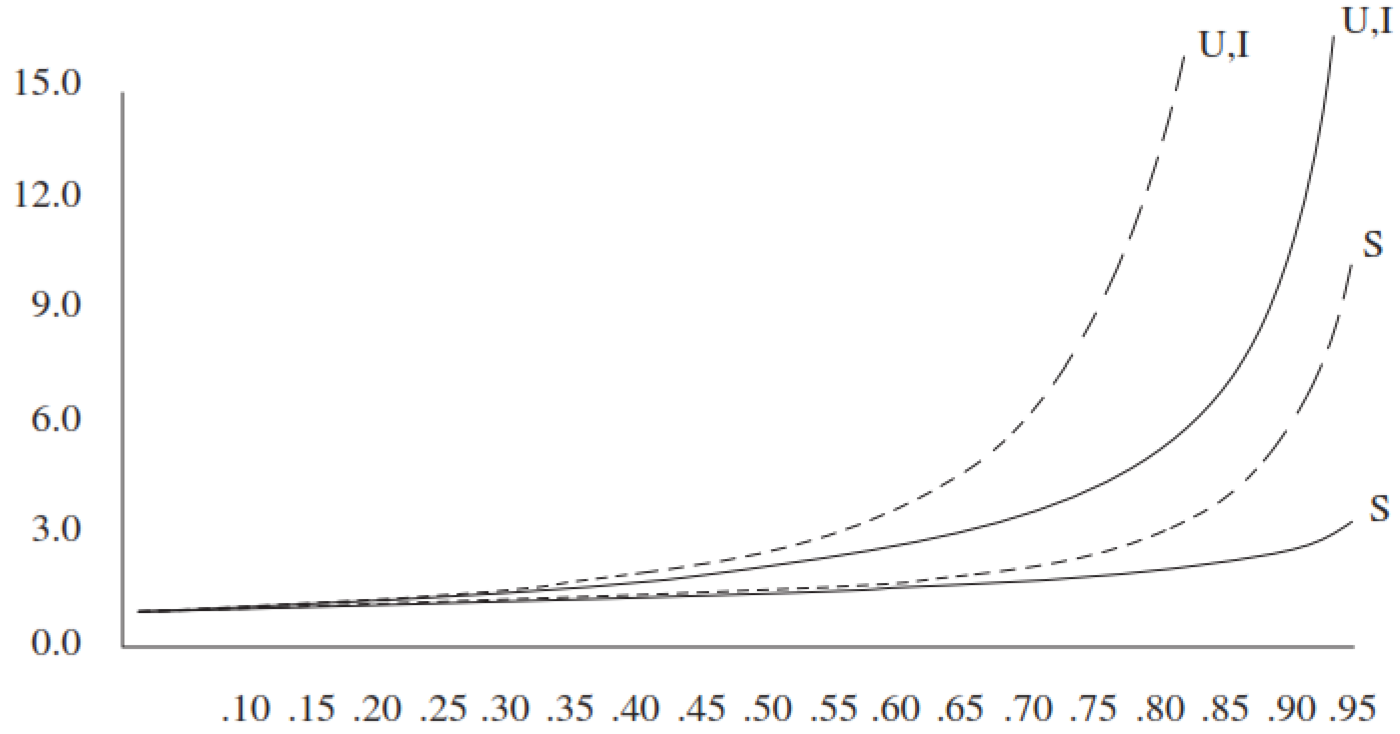
Linear Probing

- as long as the table is not full, a vacant cell can be found
  - but time to locate an empty cell can become large
  - blocks of occupied cells results in primary clustering
- deleting entries leaves holes
  - some entries may no longer be found
  - may require moving many other entries

- expected number of probes
  - for search hits: $\sim \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right)$
  - for insertion and search misses: $\sim \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$
  - for $\lambda = 0.5$, these values are $3/2$ and $5/2$, respectively

Linear Probing

- performance of linear probing (dashed) vs. more random collision resolution
  - adequate up to $\lambda = 0.5$
- Successful, Unsuccessful, Insertion



Quadratic Probing

- quadratic probing
  - eliminates primary clustering
  - collision function is quadratic
  - example: add 89, 18, 49, 58, 69 with $h(k) = k \% 10$ and $f(i) = i^2$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Quadratic Probing

- in linear probing, letting table get nearly <u>full</u> greatly hurts performance
- quadratic probing
  - no <u>guarantee</u> of finding an empty cell once the table gets larger than half full
  - at most, <u>half</u> of the table can be used to resolve collisions
  - if table is half empty and the table size is prime, then we are always guaranteed to accommodate a new element
  - could end up with situation where all keys map to the same table location

Quadratic Probing

- quadratic probing
  - collisions will probe the same alternative cells
  - secondary clustering
  - causes less than half an extra probe per search

Double Hashing

- double hashing
  - $f(i) = i \cdot hash_2(x)$
  - apply a second hash function to x and probe across <u>longer distances</u>
  - function must never evaluate to <u>0</u>
  - make sure all cells can be probed

Double Hashing

- double hashing example
  - $hash_2(x) = R - (x \bmod R)$ with $R = 7$
  - R is a prime smaller than table size
  - insert 89, 18, 49, 58, 69
    - ex. 49: 49 % 10 = 9 (taken), so add *hash*$_2$ -> (7 - 0) % 10

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Double Hashing

- double hashing example (cont.)
  - note here that the size of the table (10) is not prime
  - if 23 inserted in the table, it would collide with 58
    - since $hash_2(23) = 7 - 2 = 5$ and the table size is 10, only one alternative location, which is taken

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Rehashing

- table may get too full
  - run time of operations may take too long
  - insertions may fail for quadratic resolution
    - too many removals may be intermixed with insertions
- solution: build a new table twice as big (with a new hash function)
  - go through original hash table to compute a hash value for each (non-deleted) element
  - insert it into the new table

Rehashing

- example: insert 13, 15, 24, 6 into a hash table of size 7
  - with $h(k) = k \,\%\, 7$

0	6
1	15
2	
3	24
4	
5	
6	13

Rehashing

- example (cont.)
  - insert 23
  - table will be over 70% full; therefore, a new table is created

0	6
1	15
2	23
3	24
4	
5	
6	13

Rehashing

- example (cont.)
  - new table is size 17
  - new hash function $h(k) = k \ \% \ 17$
  - all old elements are inserted into new table

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Rehashing

- rehashing run time $O(N)$ since N elements and to rehash the entire table of size roughly $2N$
  - must have been $N/2$ insertions since last rehash
- rehashing may run OK if in background
  - if interactive session, rehashing operation could produce a slowdown
- rehashing can be implemented with quadratic probing
  - could rehash as soon as the table is half full
  - could rehash only when an insertion fails
  - could rehash only when a certain load factor is reached
    - may be best, as performance degrades as load factor increases

Hash Tables with Worst-Case $O(1)$ Access

- hash tables so far
  - $O(1)$ average case for insertions, searches, and deletions
  - separate chaining: worst case $\Theta(\log N / \log \log N)$
    - some queries will take nearly logarithmic time
  - worst-case $O(1)$ time would be better
    - important for applications such as lookup tables for routers and memory caches
    - if N is known in advance, and elements can be rearranged, worst-case $O(1)$ time is achievable

Hash Tables with Worst-Case $O(1)$ Access

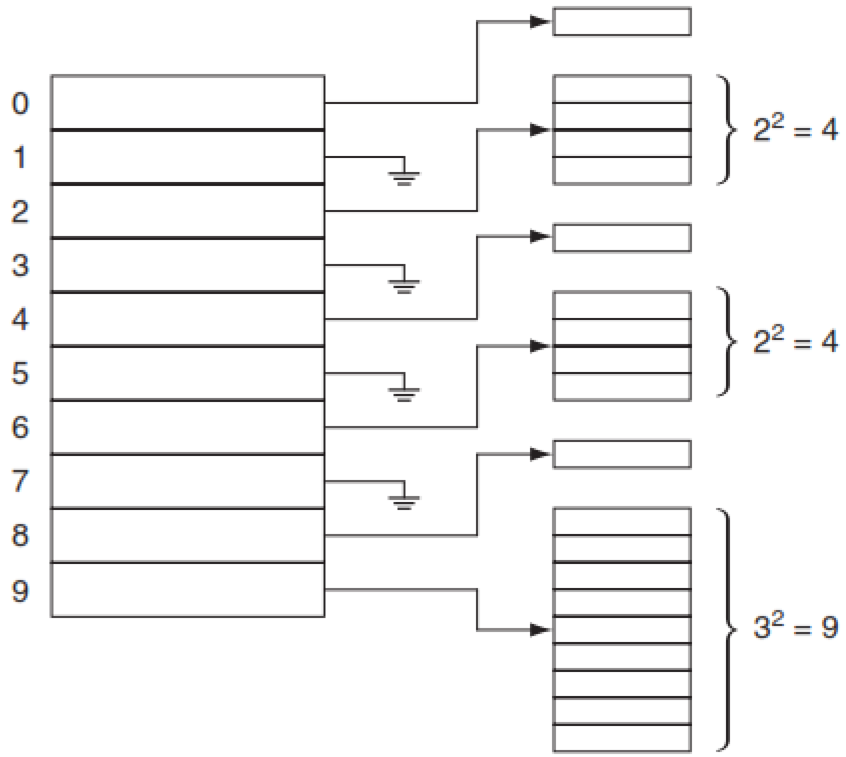
- perfect hashing
  - assume all N items known in advance
  - separate chaining
    - if the number of lists continually increases, the lists will become shorter and shorter
    - with enough lists, high probability of no collisions
    - two problems
      - number of lists might be unreasonably large
      - the hashing might still be unfortunate
        - M can be made large enough to have probability $\frac{1}{2}$ of no collisions
        - if collision detected, clear table and try again with a different hash function (at most done 2 times)

Hash Tables with Worst-Case $O(1)$ Access

- perfect hashing (cont.)
  - how large must M be?
    - theoretically, M should be N^2, which is <u>impractical</u>
  - solution: use N lists
    - resolve collisions by using hash tables instead of linked lists
      - each of these lists can have n^2 elements
  - each secondary hash table will use a different hash function until it is <u>collision-free</u>
    - can also perform similar operation for primary hash table
  - total size of secondary hash tables is at most $2N$

Hash Tables with Worst-Case $O(1)$ Access

- perfect hashing (cont.)
  - example: slots 1, 3, 5, 7 empty; slots 0, 4, 8 have 1 element each; slots 2, 6 have 2 elements each; slot 9 has 3 elements



Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing
  - $\Theta(\log N / \log \log N)$ bound known for a long time
  - researchers surprised in 1990s to learn that if one of two tables were randomly chosen as items were inserted, the size of the largest list would be $\Theta(\log \log N)$, which is significantly smaller
  - main idea: use 2 tables
    - neither more than half full
    - use a separate hash function for each
    - item will be stored in one of these two locations
    - collisions resolved by displacing elements

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - example: 6 items; 2 tables of size 5; each table has randomly chosen hash function

Table 1	
0	B
1	C
2	
3	E
4	

Table 2	
0	D
1	
2	A
3	
4	F

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

  - A can be placed at position 0 in Table 1 or position 2 in Table 2
  - a search therefore requires at most 2 table accesses in this example
  - item deletion is trivial

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - insertion
    - ensure item is not already in one of the tables
    - use first hash function and if first table location is <u>empty</u>, insert there
    - if location in first table is occupied
      - <u>displace</u> element there and place current item in correct position in first table
      - displaced element goes to its alternate hash position in the second table

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - example: insert A

Table 1	
0	A
1	
2	
3	
4	

Table 2	
0	
1	
2	
3	
4	

A: 0, 2

  - insert B (displace A)

Table 1	
0	B
1	
2	
3	
4	

Table 2	
0	
1	
2	A
3	
4	

A: 0, 2
B: 0, 0

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - insert C

Table 1	
0	B
1	C
2	
3	
4	

Table 2	
0	
1	
2	A
3	
4	

A: 0, 2
B: 0, 0
C: 1, 4

  - insert D (displace C) and E

Table 1	
0	B
1	D
2	
3	E
4	

Table 2	
0	
1	
2	A
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - insert F (displace E)

Table 1	
0	B
1	D
2	
3	**F**
4	

Table 2	
0	
1	
2	A
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

  - (E displaces A)

Table 1	
0	B
1	D
2	
3	**F**
4	

Table 2	
0	
1	
2	**E**
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

  - (A displaces B)

Table 1	
0	**A**
1	D
2	
3	**F**
4	

Table 2	
0	
1	
2	**E**
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

  - (B relocated)

Table 1	
0	**A**
1	D
2	
3	**F**
4	

Table 2	
0	**B**
1	
2	**E**
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - insert G

Table 1	
0	B
1	C
2	
3	E
4	

Table 2	
0	D
1	
2	A
3	
4	F

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4
G: 1, 2

Table 1	
0	**A**
1	D
2	
3	**F**
4	

Table 2	
0	**B**
1	
2	**E**
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

  - displacements are <u>cyclical</u>
    - G D B A E F C G
    - can try G's second hash value in second table, but it also results in a displacement cycle

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - cycles
    - if table's load value < 0.5, probability of a cycle is very <u>low</u>
      - insertions should require < $O(\log N)$ displacements
    - if a certain number of displacements is reached on an insertion, tables can be <u>rebuilt</u> with new hash functions

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - variations
    - higher number of tables (3 or 4)
    - place item in second hash slot immediately instead of <u>displacing</u> other items
    - allow each cell to store <u>multiple</u> keys
      - space utilization increased

	1 item per cell	2 items per cell	4 items per cell
2 hash functions	0.49	0.86	0.93
3 hash functions	0.91	0.97	0.98
4 hash functions	0.97	0.99	0.999

Hash Tables with Worst-Case $O(1)$ Access

- cuckoo hashing (cont.)
  - benefits
    - worst-case <u>constant</u> lookup and deletion times
    - avoidance of <u>lazy deletion</u>
    - potential for <u>parallelism</u>
  - potential issues
    - extremely sensitive to choice of hash functions
    - time for insertion increases rapidly as load factor approaches 0.5

Hash Tables with Worst-Case $O(1)$ Access

- hopscotch hashing
  - improves on linear probing algorithm
    - linear probing tries cells in sequential order, starting from hash location, which can be long due to primary and secondary clustering
    - instead, hopscotch hashing places a bound on <u>maximal length</u> of the probe sequence
      - results in worst-case constant-time lookup
      - can be parallelized

Hash Tables with Worst-Case $O(1)$ Access

- hopscotch hashing (cont.)
  - if insertion would place an element too far from its hash location, go backward and <u>evict</u> other elements
    - evicted elements cannot be placed farther than the maximal length
  - each position in the table contains information about the current element inhabiting it, plus others that <u>hash</u> to it

Hash Tables with Worst-Case $O(1)$ Access

- hopscotch hashing (cont.)
  - example: MAX_DIST = 4

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1000
10	E	0000
11	G	1000
12	F	1000
13		0000
14		0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11

  - each bit string provides 1 bit of information about the current position and the next 3 that follow
    - 1: item hashes to current location; 0: no

Hash Tables with Worst-Case $O(1)$ Access

- hopscotch hashing (cont.)
  - example: insert H in 9

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1000
10	E	0000
11	G	1000
12	F	1000
13		0000
14		0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1000
10	E	0000
11		**0010**
12	F	1000
13	**G**	0000
14		0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	**1010**
10	E	0000
11	**H**	0010
12	F	1000
13	G	0000
14		0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11
H: 9

  - try in position 13, but too far, so try candidates for eviction (10, 11, 12)
  - E would move too far away, so evict G in 11

Hash Tables with Worst-Case $O(1)$ Access

- hopscotch hashing (cont.)
  - example: insert I in 6

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0010
12	F	1000
13	G	0000
14		0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	**0001**
12	F	1000
13		0000
14	**G**	0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12		**0100**
13	**F**	0000
14	G	0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11
H: 9
I: 6

  - position 14 too far, so try positions 11, 12, 13
  - H would move too far away, but G can move down one
  - position 13 still too far; F can move down one ***

Hash Tables with Worst-Case $O(1)$ Access

- hopscotch hashing (cont.)
  - example: insert I in 6

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12		0100
13	F	0000
14	G	0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9		**0011**
10	E	0000
11	H	0001
12	**B**	0100
13	F	0000
14	G	0000
...		

→

	Item	Hop
...		
6	C	**1001**
7	A	1100
8	D	0010
9	**I**	0011
10	E	0000
11	H	0001
12	B	0100
13	F	0000
14	G	0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11
H: 9
I: 6

  - position 12 still too far, so try positions 9, 10, 11
  - B can move down three
  - now slot is open for I, fourth from 6

Hash Tables with Worst-Case $O(1)$ Access

- universal hashing
  - in principle, we can end up with a situation where all of our keys are hashed to the same location in the hash table (bad)
  - more realistically, we could choose a hash function that does not evenly distribute the keys
  - to avoid this, we can choose the hash function randomly so that it is independent of the keys being stored
  - yields provably good performance on average

Hash Tables with Worst-Case $O(1)$ Access

- universal hashing (cont.)
  - let H be a finite collection of hash functions mapping our set of keys K to the range $\{0,1,\ldots,M-1\}$
  - H is a universal collection if for each pair of distinct keys $k,l \in K$, the number of hash functions $h \in H$ for which $h(k)=h(l)$ is at most $|H|/M$
  - that is, with a randomly selected hash function $h \in H$, the chance of a collision between distinct k and l is not more than the probability $(1/M)$ of a collision if $h(k)$ and $h(l)$ were chosen randomly and independently from $\{0,1,\ldots,M-1\}$

Hash Tables with Worst-Case $O(1)$ Access

- universal hashing (cont.)
  - example: choose a prime p sufficiently large that every key k is in the range 0 to $p - 1$ (inclusive)
  - let $A = \{0, 1, \ldots, p - 1\}$ and $B = \{1, \ldots, p - 1\}$

    then the family

    $$h_{a,b}(k) = \big((ak + b)\ mod\ p\big)\ mod\ M \quad a \in A, b \in B$$

    is a universal class of hash functions

Hash Tables with Worst-Case $O(1)$ Access

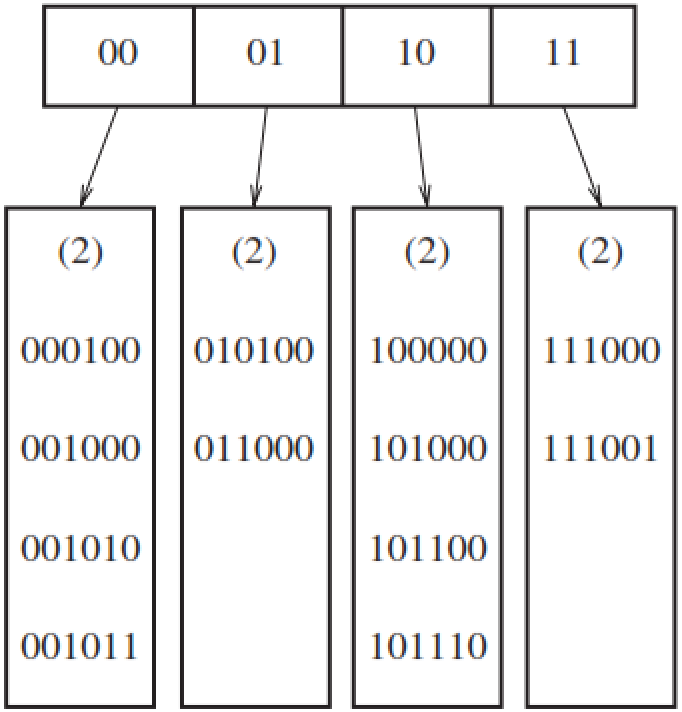
- extendible hashing
  - amount of data too large to fit in <u>memory</u>
    - main consideration is then the number of disk accesses
  - assume we need to store N records and $M = 4$ records fit in one disk block
  - current problems
    - if probing or separate chaining is used, collisions could cause <u>several blocks</u> to be examined during a search
    - rehashing would be expensive in this case

Hash Tables with Worst-Case $O(1)$ Access

- extendible hashing (cont.)
  - allows search to be performed in <u>2</u> disk accesses
    - insertions require a bit more
  - use B-tree
    - as M increases, height of B-tree <u>decreases</u>
      - could make height = 1, but multi-way branching would be extremely high

Hash Tables with Worst-Case $O(1)$ Access

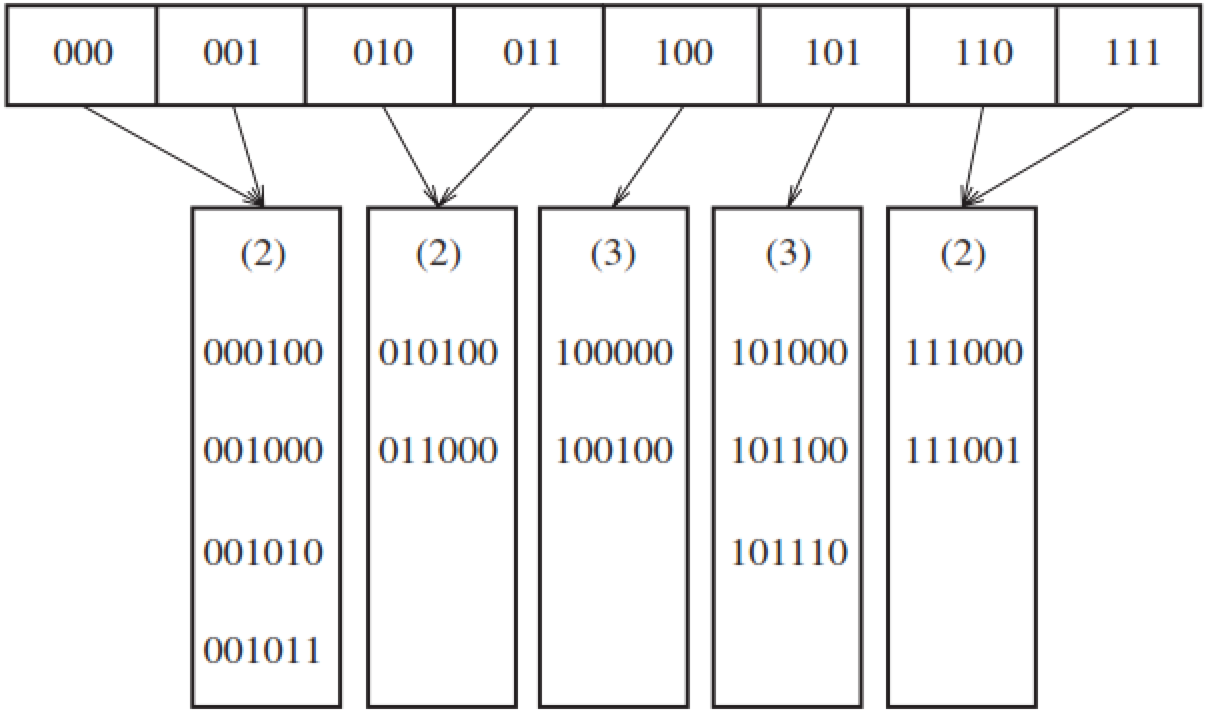
- extendible hashing (cont.)
  - example: 6-bit integers



  - root contains 4 pointers determined by first 2 bits
  - each leaf has up to 4 elements

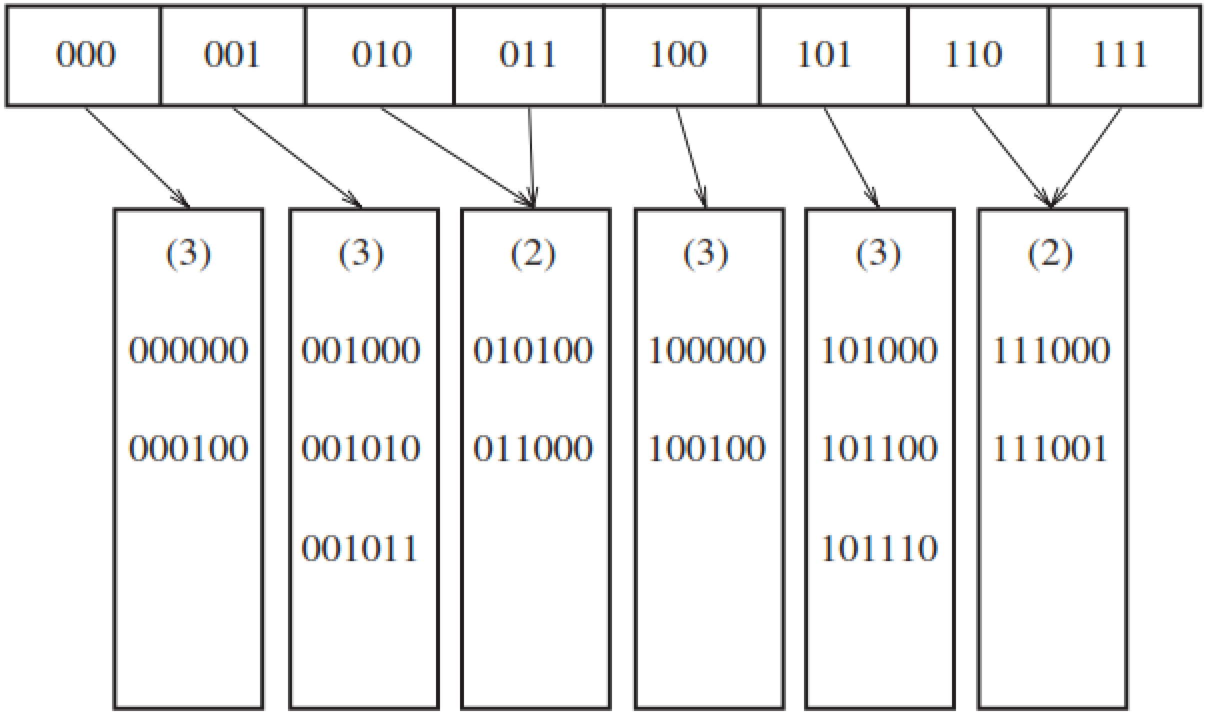
Hash Tables with Worst-Case $O(1)$ Access

- extendible hashing (cont.)
  - example: insert 100100
    - place in third leaf, but full
    - split leaf into 2 leaves, determined by 3 bits



Hash Tables with Worst-Case $O(1)$ Access

- extendible hashing (cont.)
  - example: insert 000000
    - first leaf split



Hash Tables with Worst-Case $O(1)$ Access

- extendible hashing (cont.)
  - considerations
    - several directory splits may be required if the elements in a leaf agree in more than D+1 leading bits
      - number of bits to distinguish bit strings
      - does not work well with duplicates ($> M$ duplicates: does not work at all)

Hash Tables with Worst-Case $O(1)$ Access

- final points
  - choose hash function carefully
  - watch load factor
    - separate chaining: close to 1
    - probe hashing: 0.5
  - hash tables have some limitations
    - not possible to find min/max
    - not possible to search for a string unless the exact string is known
      - binary search trees can do this, and $O(\log N)$ is only slightly worse than $O(1)$

Hash Tables with Worst-Case $O(1)$ Access

- final points (cont.)
  - hash tables good for
    - symbol table
    - gaming
      - remembering locations to avoid recomputing through transposition table
    - spell checkers