CSCI 335
Software Design and Analysis III
Lecture 12: Hashing-2

Professor Anita Raja

10-13-22

Announcements

- Midterm 10/20 2:30pm-3:45pm
  - in class, closed book, closed notes, no electronic devices.
  - All material including Lecture 12. Short review on Monday 10/17.
  - Arrive for exam entrance 2:15pm on day of exam and please line up outside – do not enter exam hall until id is checked.
  - One 8x11 cheat sheet allowed.
  - Instructions will be posted on blackboard ahead of time.
  - Accommodation requests (2 weeks before exam per syllabus).
- HW3 will be released next week.

Agenda

- Hash tables
- Hash tables without Linked Lists
  - Linear Probing
  - Quadratic Probing
- Hash tables with Linked Lists
- Separate Chaining

Collision Resolution Strategies

- Open addressing
  - **Linear probing**
  - Plus 3 rehash
  - Quadratic probing (failed attempts)^2
  - Double hashing
- Closed addressing
  - Separate chaining

Probing hash tables

- Suppose x is the key.
- Try cells $h_{0(x)}$, $h_{1(x),}$ $h_{2(x)}$ in succession where

$h_{i\,(x)}$= (hash(x) + f(i)) mod Table with f(0)= 0

- Do not use additional memory outside of the table.

Problem: Primary Clustering

- Collisions in a crowded range will increase the number of collisions in that range.
- Open addressing
  - **Linear probing**
  - Plus 3 rehash
  - Quadratic probing (failed attempts)^2
  - Double hashing

Linear Probing

- Average number of probes is

$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$ for hits

$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$ for misses or inserts

λ	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{3}{4}$	$\frac{9}{10}$
Hit	1.5	2.0	3.0	5.5
Miss	2.5	5.0	8.5	55.5

Random collision resolution

Assume huge table (i.e. clustering not an issue) and each probe is independent of the the previous probes.

Theorem:

Expected # of probes in miss = Expected # of probes to find empty cell

= $1/(1- \lambda)$.

Proof

Probability{Selecting an empty cell} = 1- λ = p = Prob. of success

Probability{Selecting a non-empty cell} = λ = 1 – p = Prob. of failure

Finding an empty cell is like flipping a coin N items until success (coin is biased having probability p of selecting success)

For example if # of probes is 4, then coin provides can provide F, F, F, S

# of probes is thus a random variable X having a Geometric Probability Distribution

Expected value of X is thus 1/Prob. of success = 1/(1- λ)

Check some values: $\lambda = 0$, $\lambda = 0.3$, $\lambda = 0.5$, $\lambda = 0.7$, $\lambda = 0.9$, $\lambda = 1$

Random collision resolution

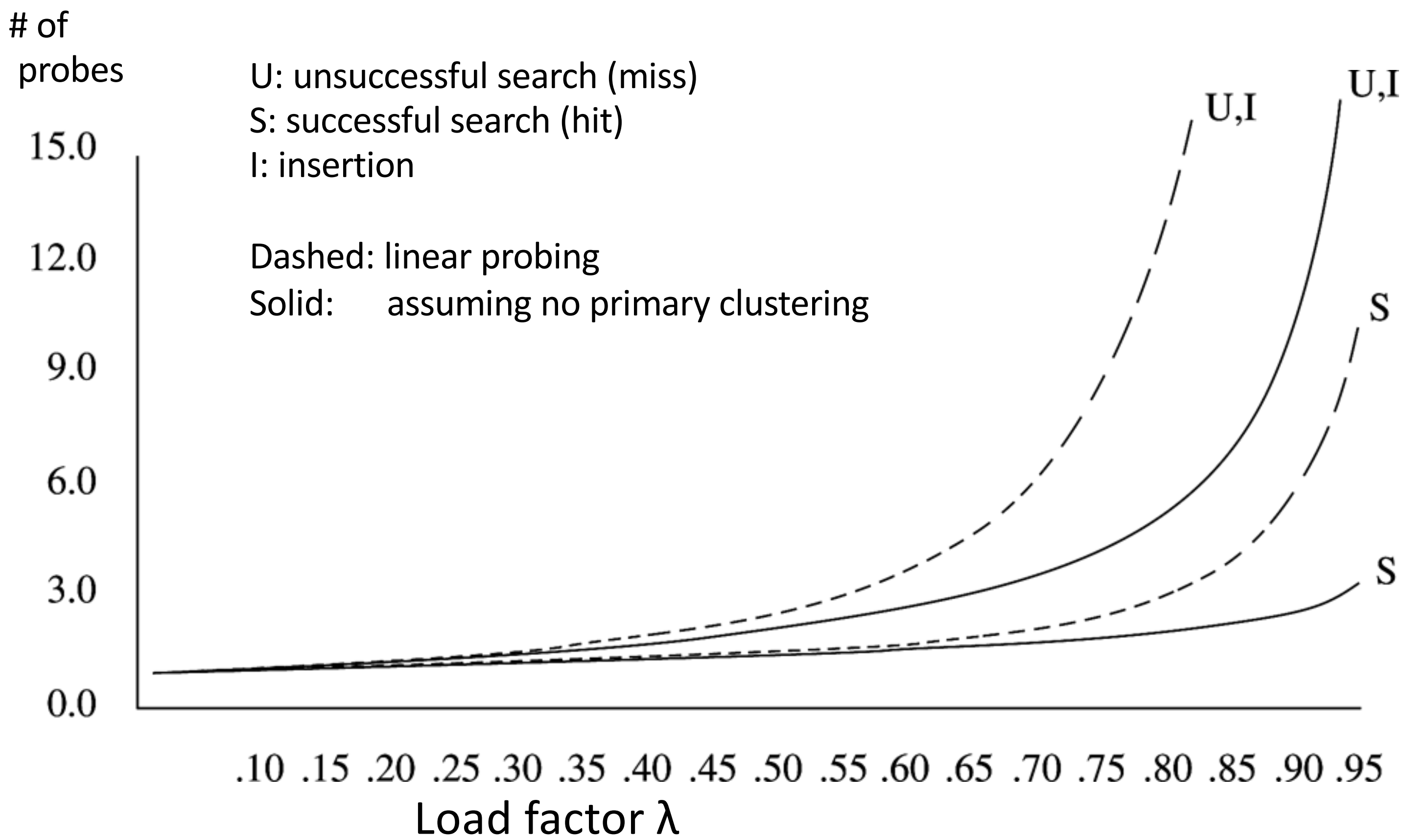
- Expected # of probes for insert = Expected # of probes for miss (why ?)

Random collision resolution

- Expected # of probes for insert = ?
- λ changes after each insert, so for hits we take the mean over λ :

$$I(\lambda) = \frac{1}{\lambda}\int_0^{\lambda} \frac{1}{1-x}\,dx = \frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

Example


of
probes
U: unsuccessful search (miss)
S: successful search (hit)
I: insertion
Dashed: linear probing
Solid: assuming no primary clustering
15.0
12.0
9.0
6.0
3.0
0.0
U,I
U,I
S
S
.10 .15 .20 .25 .30 .35 .40 .45 .50 .55 .60 .65 .70 .75 .80 .85 .90 .95
Load factor λ

Summary of Linear probing

- Quite competitive, though, when the load factors are in the range 30-70% as clusters tend to stay small.
- In addition, a few extra probes is mitigated when sequential access is much faster than random access, as in the case of caching.
- Because of primary clustering, sensitive to
  - quality of the hash function or
  - the particular mix of keys that result in many collisions or clumping.
- Therefore, it may not be a good choice for general purpose hash tables.

Quadratic probing

f(i) = i^2

i.e.

f(0) = 0

f(1) = 1

f(2) = 4

f(3) = 9

…

Insert 89, 18, 49, 58, 69.

	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

- Probing sequence:

  hi(x) = (hash(x) + i^2) mod T, for i = 0, 1, … , until spot is found.

Search for: 18, 69, 79.

Quadratic Probing

Theorem

If Q.P. is used, and TableSize is prime, then a new element can always be inserted if the table is at least half empty.

Proof

Let TableSize be a prime greater than 3.

We show that the first $\left\lceil \frac{TableSize}{2} \right\rceil$ alternative locations, including the initial $h_0(x)$ are all distinct.

Quadratic Probing

Consider two alternative locations in the first $\left\lceil \frac{TableSize}{2} \right\rceil$ set:

$h(x) + i^2$ (mod TableSize) and

$h(x) + j^2$ (mod TableSize)

where $0 \leq i,j \leq \left\lfloor \frac{TableSize}{2} \right\rfloor$

Assume towards contradiction that these locations are the same but $i \neq j$.

Then $h(x) + i^2$ (mod TableSize) = $h(x) + j^2$ (mod TableSize)

i^2 (mod TableSize) = j^2 (mod TableSize)

$i^2 - j^2 = 0$ (mod TableSize)

$(i - j)\ (i + j) = 0$ (mod TableSize)

Quadratic Probing

Since TableSize is prime, then either (i-j) = 0 (mod TableSize) OR (i+j)=0 (mod TableSize).

Since i≠j, (i-j) ≠ 0 (mod TableSize)

Since $0 \le i,j \le \lfloor \frac{TableSize}{2} \rfloor$, (i+j) ≠ 0 (mod TableSize).

Thus the two alternate locations we selected are not the same (they are distinct).

We have thus proved that if atmost $\lfloor \frac{TableSize}{2} \rfloor$ positions are taken, an empty spot can always be found.

Implementation in C++

- Lazy deletion is preferred strategy
- Clever way of computing probing sequence in Q.P. without doing multiplication
  - The difference between consecutive square numbers is an odd number.
  - $(i+1)^2 - i^2 = 2i + 1$
  - So $f(i) = f(i-1) + 2i + 1$
  - The difference between consecutive odd numbers is 2.

Double Hashing

Example of second hash function:

$hash_2(x) = R - (x \bmod R)$

R: prime

R < table_size

- Sequence of probes:

  $Probe(i) = (hash(x) + i * hash_2(x)) \bmod T$

  - needs care
  - Should not evaluate to zero
  - R should also be prime
  - What if we insert 23 next?

$hash_2(x) = 7 - (x \bmod 7)$

	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

- Like Q.P, it is a collision resolution method.

- If table size is not prime, it is possible to run out of alternative locations prematurely.

- However if double hashing is correctly implemented,
  - simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy.
  - This makes double hashing theoretically interesting.

- Quadratic probing however
  - does not require the use of a second hash function and
  - is thus likely to be simpler and faster in practice, especially for keys like strings.

Rehashing

- If table gets too full,
  - running time for operations will take too long
  - Insertions might fail for open addressing hashing with quadratic resolution
  - Can happen if there are too many removals intermixed with insertions.
- Solution
  - When table is over 70 % full, build another table that is twice as big with associated hash function
  - Scan the entire original table
  - Compute new hash value for each non-deleted element and insert it in the new table.

Rehashing

- Increase T, and re-hash elements
- Expensive operation

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

After insertion: 13,15,6,24,23 (mod 7 hash).

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

h(x)=x mod 17 .

Rehashing with Q.P.

- Can be implemented in several ways
  - Rehash as soon as table is half full
  - When insertion fails
  - When the table reaches a certain load factor.

STL's unordered_set/map

- Hashtable implementations of sets and maps
- Same functionality as set and map, but no ordered capabilities.
  - Items in ordered_set and keys unordered_map must provide an overloaded operator== and a hash function
  - Ordered_set and map templates can be instantiated with a function object that provides comparison function,
  - Unordered sets and maps can be instantiated with function objects that provide hash function and equality operators

Word changing example from Chapter 4:

- Method 1:
  - Map in which the key is a word and the value is a collection of all words that differ in only one character from that word.
  - Unordered_map unless we want printHighChangeables to alphabetically list the subset of words that it can be changed into
- Method 2:
  - key is word length and value is a collection of all words of that word length.
  - Unordered_map since order in which word lengths are processed does not matter.
- Method 3:
  - key is representative and value is a collection of all words with that representative.
  - Unordered_map since order in which word lengths are processed does not matter.

Stl's unordered_set: How to provide your own hash function?

```
// Usage:
//   CaseInsensitiveStringHashFunction case_insensitive_hash;
//   string input_str; cin >> input_str;
//   cout << case_insensitive_hash(input_str);  // Will get the hash
value for given string.

class CaseInsensitiveStringHashFunction {

  public:

    size_t operator() (const string &input_string) const {

       static hash<string> hash_functional;

       string to_lower_case = input_string;

       std::transform(to_lower_case.begin(),
to_lower_case.end(), to_lower_case.begin(),

                       [](unsigned char c) {return
std::tolower(c);});

       return hash_functional(to_lower_case);

    }

};
```

```
// Usage:
//    Used for overloaded equality operator
//    CaseInsensitiveStringEquality case_insentitive_equality;
//    string str1, str2; cin >> str1; cin >> str2;
//    cout << case_insentitive_equality (str1, str2); // Returns true
if strings are equal ignoring case.
class CaseInsensitiveStringEquality {

  public:

    bool operator()(const string &lhs, const string &rhs) const
{

        return EqualIgnoreCase(lhs, rhs);  // Implement this.

  }

};

// This is how you can now declare your unordered_set.

unordered_set<string,

               CaseInsensitiveStringHashFunction,

               CaseInsensitiveStringEquality> my_hash_set;

my_hash_set.insert("an input string");
```

Stl's unordered_set: How to provide your own hash function? more concise

```
// Usage:
//   CaseInsensitiveStringHash case_insensitive_hash;
//   string input_str; cin >> input_str;
//   cout << case_insensitive_hash(input_str);  // Will get the hash value for given string.
//   string other_str; cin >> other_str;
//   cout << case_insensitive_hash(input_str, other_str); // Will return true if strings are equal.
class CaseInsensitiveStringHash {
  public:
    // Hash overload.
    size_t operator() (const string &input_string) const {
       static hash<string> hash_functional;
       string to_lower_case = input_string;
       std::transform(to_lower_case.begin(), to_lower_case.end(), to_lower_case.begin(),
                    [](unsigned char c) {return std::tolower(c);});
       return hash_functional(to_lower_case);
    }
    // Equality overload.
    bool operator()(const string &lhs, const string &rhs) const {
        return EqualIgnoreCase(lhs, rhs); // EqualIgnoreCase() is implemented elsehwere.
    }
};
```

```
//
// This is how you can now declare your unordered_set.
unordered_set<string,
                CaseInsensitiveStringHash,
                CaseInsensitiveStringHash> my_hash_set;
my_hash_set.insert("an input string");
```

Worst-Case Access

- Hashtable we have examined so far
  - with reasonable load factors and appropriate hash functions,
  - expect O(1) cost on average for insertions, deletions and search.
- If use separate chaining, and assume load factor 1, what is the worst-case access time?
- Worst case analysis problem is formulated as:

  Given N balls to be placed (randomly) in N bins,

  what is expected number of balls in most occupied bin?

Worst-Case Access

Given N balls to be placed (randomly) in N bins,

what is expected number of balls in most occupied bin?

- Result from Probability & Statistics theory:

  $\Theta(\log N / \log\log N)$

- Meaning on average we expect find queries to take nearly log time.
  - Not O(1)

Worst-Case Access O(1)

- Perfect Hashing provides a solution, sect. 5.7
  - the primary hash table is constructed several times if the number of collisions that are produced is higher than required.
  - We will not cover it.

Extendible Hashing

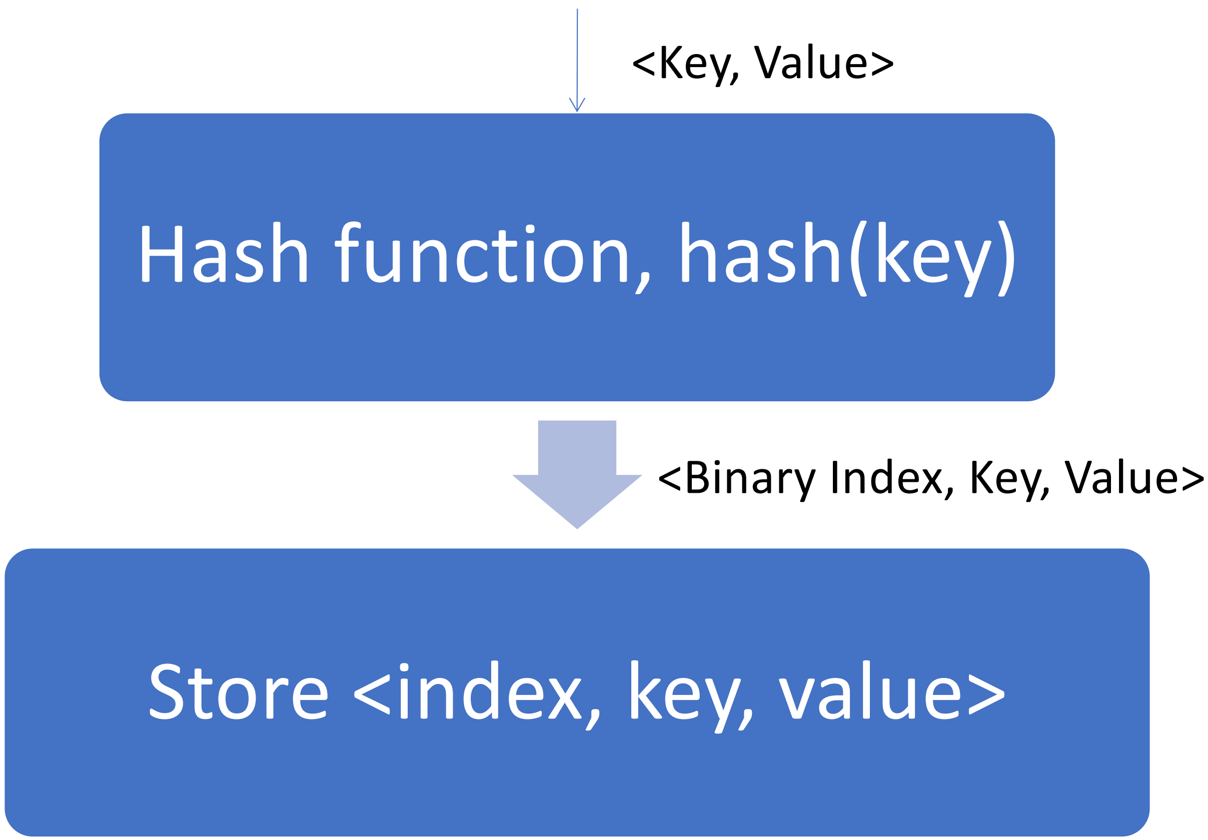
- Hash table is huge => store on disk
  - N records to store
  - M records fit in one disk block
    - M < N
- Solution?
  - Regular hashing?
    - Collisions may require many disk accesses
    - Rehashing is extremely expensive
      - O(N) disk accesses.
  - Extendible hashing
    - 2 disk accesses for search.

B-tree approach for Extendible Hashing

- Depth $\log_{k/2} N$ – k is the branching factor
- Can we make its depth be 2?

- Consider the bits of the hash index:
  - <binary number> = hash key
  - Store these binary numbers in a clever way.

Extendible Hashing



Do not use simple array

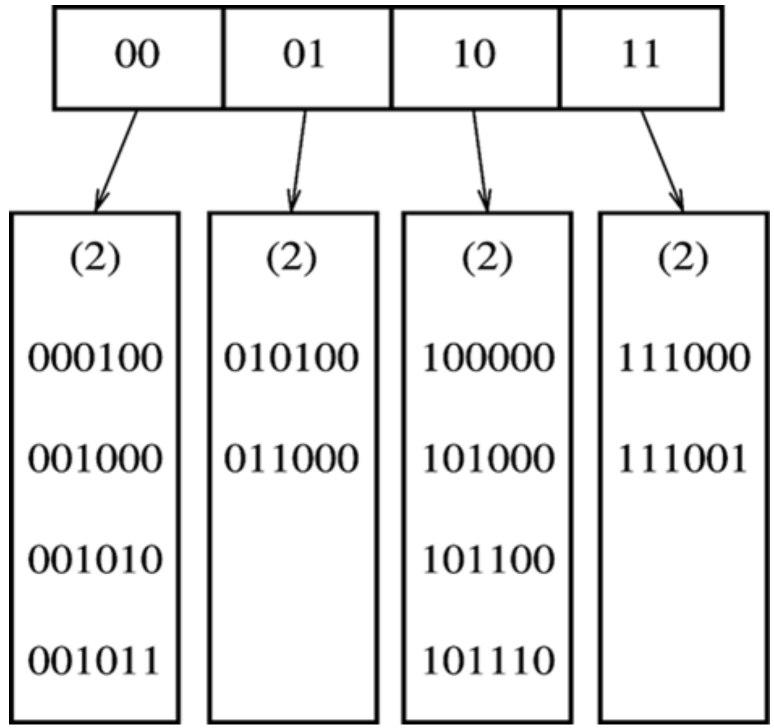
Extendible Hashing

- Example: Store 6-bit integers
- Root level: directory
- D is # of bits used by root

  2^D # of entries in dir
- d_L # of common leading bits in a leaf L

  $d_L \leq D$

- ....Insert 100100

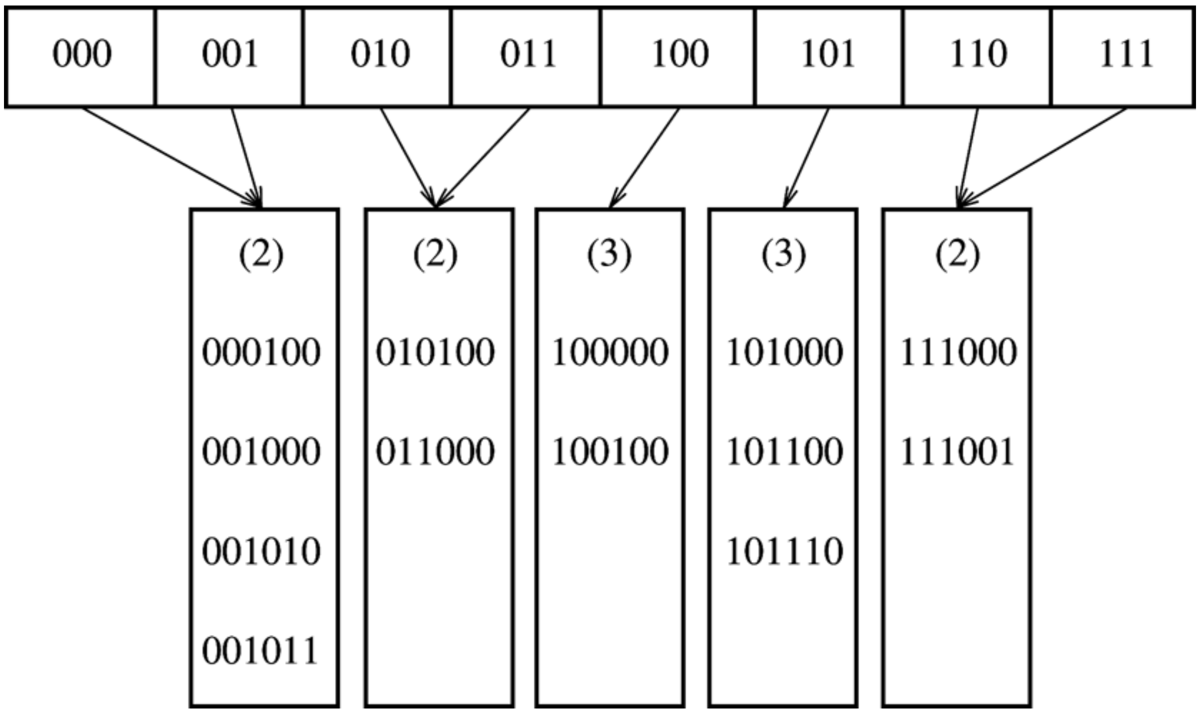


Extendible Hashing

- Insert 100100

=> Directory split

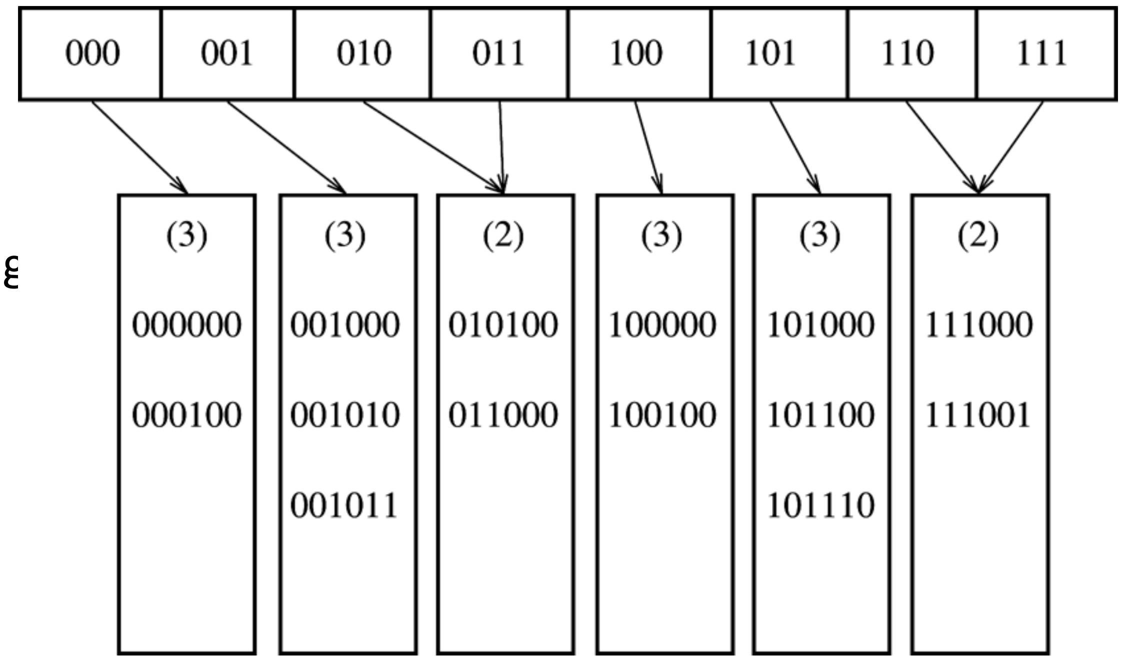
- Changes?



Extendible Hashing

- Changes?
  - D = 3
  - Up to 8 entries in Directory

  Upto 3 common leading bits In a leaf



…Insert 000000

Extendible Hashing

- Treats hash as a bit string
  - Very simple strategy that provides quick access times for insert and search operations on large databases.
- Insertions may require more than 1 split.
  - Example: Insert 111010,111011, 111100 in initial table
- How do we handle collisions?
  - In this case we have non-unique binary indices
    - Note that indices are the result of hash() operation
    - For example two records could hash to 010100
- What if more than M collisions? (M is maximum number of elements stored in a leaf)
  - E.g. more than M records hash to 010100
- Bits need to be fairly random => hash(key) should be fairly long integer

Extendible Hashing performance

- Assume that bit patterns are uniformly distributed
- Expected number of leaves is :

  $(N/M)\ \log_2(e) = (N/M)\ \ (1/\ln(2)) = (N/M)*1.442..$

  N=1,000,000,000 records (billion)

  M = 500

  => $2.88.. * 10^6$ leaves
- Average leaf is ln(2) = 0.69 full (like B-tree)
  - Not surprising since for both data structures new nodes are created when the (M+1)th entry is added.

Extendible Hashing performance

- Surprising result is Expected size of directory:

  $O(N^{1+1/M} / M)$

  N=1,000,000,000 records (billion)

  M = 500

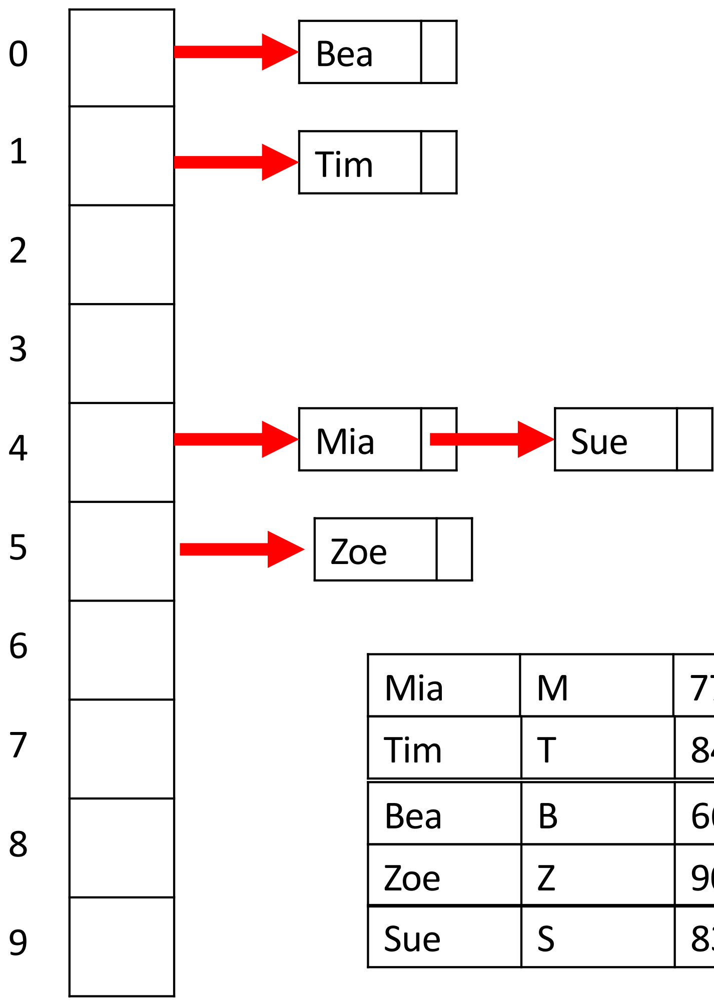
  => $\leq c * 2.08.. * 10^6$ entries (expected # of leaves)
- M=10

  $\Rightarrow \leq c * 7.94.. * 10^8$ entries
- The smaller the M the larger the directory size

Extendible Hashing performance

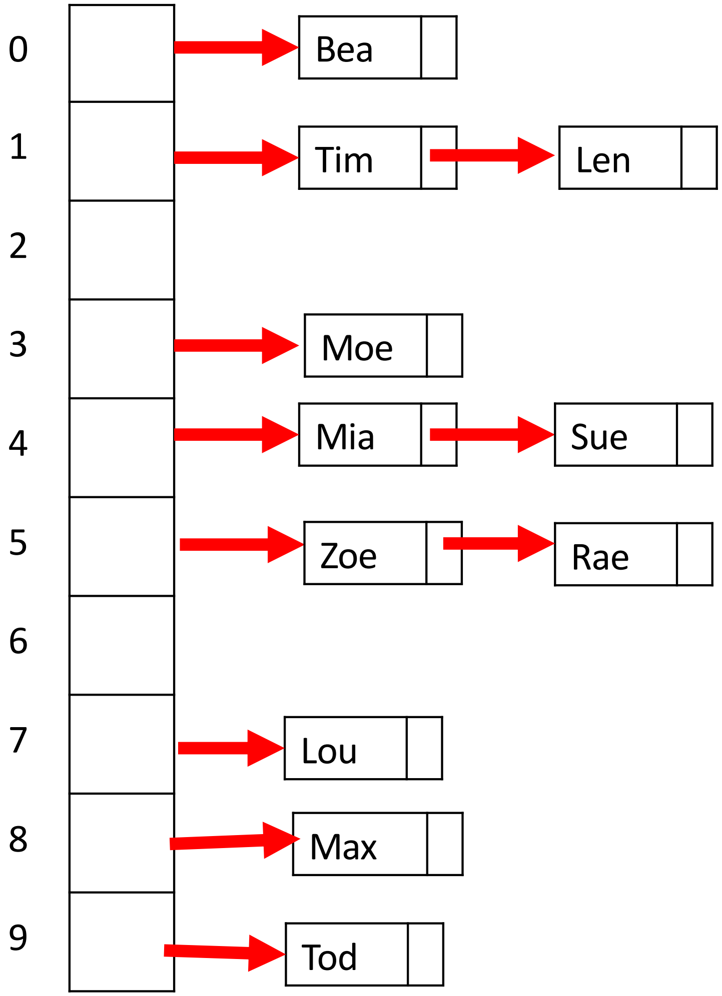
- Because of the hierarchical nature of the system, re-hashing is an incremental operation (done one bucket at a time, as needed).
- =>time-sensitive applications are less affected by table growth than by standard full-table rehashes.
- Practically all modern filesystems use either extendible hashing or B-trees.

Separate Chaining (Closed addressing)



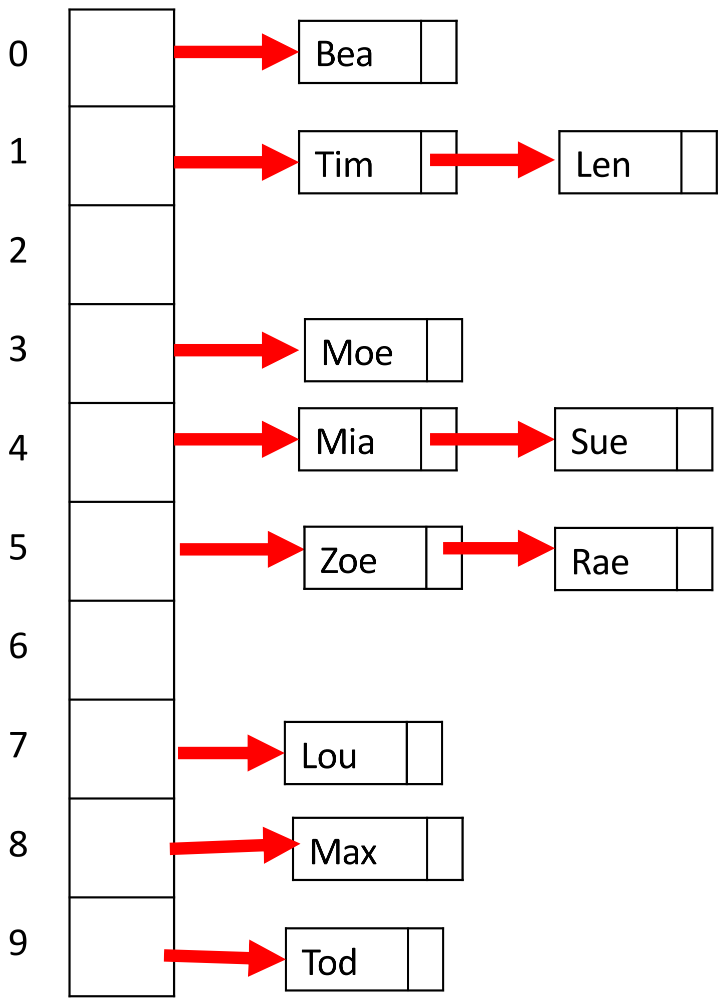
Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Jan	J	74	a	97	n	110	281	6
Ada	A	65	d	100	a	97	262	9
Leo	L	76	e	101	o	111	288	2
Sam	S	83	a	97	m	109	289	3
Lou	L	76	o	111	u	117	304	7
Max	M	77	a	97	x	120	294	8
Ted	T	84	e	101	d	100	285	10



Find Rae 280 Mod 11 = 5

myData = Array(5)


0
1
2
3
4
5
6
7
8
9
Bea
Tim
Len
Moe
Mia
Sue
Zoe
Rae
Lou
Max
Tod

Separate Chaining Analysis

- On average length of list is λ, the load factor
  - Ratio of the number of elements in the hash table to table size.
- Unsuccessful search (miss): $O(1) + \lambda$ on average
- Successful search (hit) : $O(1) + \frac{\lambda}{2}$
- λ is important, should try to keep it around 1

Why is successful list $O(1) \;\; + \frac{\lambda}{2}$?

- List being searched contains the one node that stores the match +zero or more other nodes
- Expected # of other nodes in a table of N elements and M lists is
  - (N-1)/M = λ-1/M which is essentially λ, since M is presumed to be large.
- On average half the other nodes are searched, so combined with the matching node, we obtain the average cost of 1+ $\lambda/2$
- Analysis shows that table size is not really important but λ is.

General Rule for Separate Chaining

- Make table size about as large as the number of elements expected (i.e. let $\lambda \sim 1$).
- In code, if $\lambda > 1$, we expand the table size by calling rehash
- It is a good idea to keep the table size prime to ensure a good distribution.

Applications

- When log(*n*) is just too big…
  - Symbol tables in interpreters
  - Real-time databases (in core or on disk)
    - air traffic control
    - packet routing
    - graphs where nodes are strings (e.g. names of cities)
    - password checking
  - Spell-checkers
- When associative memory is needed…
  - Dynamic programming
    - cache results of previous computation

      f(x) →if (Find(x)) then Find(x) else f(x)
    - Chess endgames
  - Many text processing applications – e.g. Web

    $Status{$LastURL} = "visited";

Separate Chaining Summary

- Used to index large amounts of data
- Address of each key calculated using the key itself.
- Collisions resolved when open or closed addressing
- Hashing is widely used in database indexing, compilers, caching, password authentication and more
- Insertion, deletion and retrieval occur in constant time.

Hash Summary

- Constant average time for insert/find
- Load factor λ is crucial
  - ~1 for separate chaining
  - <0.5 for probing
  - Can change it with rehashing (expensive)
- BSTrees could also be used
  - Sort, findMin/Max
  - Search within a range
  - O(log n) is not always larger than O(1)
- If no ordering is required, hashtable set/map is probably better
  - Can try both to see which is better in practice.
  - 1 second difference for the 1-letter replacement word problem.