CSCI 335 Software Design and Analysis III

Hashing

Hash Applications

- Symbol tables (compilers)
- Graphs where nodes are strings (e.g. names of cities)
- Spell-checkers
- Password checking
- …

Hashing

- Ideally O(1) find/insert/delete.
  - But no ability for sorting-based features.
- Data is stored in a table (array/vector).
  - Let us call size of table T and number of elements stored in table N.
  - Load factor $\lambda = N / T$ (helps to analyze performance).
- Hash function gives index in constant time:
  - Hash Function: Key -> {0, 1, 2, …, T-1}.
  - Key: Integer/String/etc. – depending on application.
  - Mapping from keys to indices is not unique =>
    - Collision resolution is required.

Hash table example

T: size of table
N: number of items
in table

Load factor $\lambda = N / T$

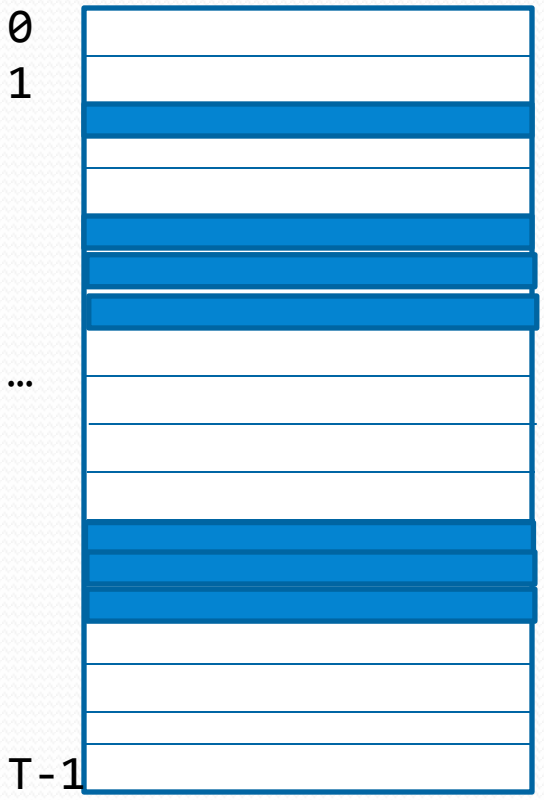
Example for T = 101:

N = 30 => $\lambda = 30 / 101 = 0.29$

N = 82 => $\lambda = 0.81$

…

N = 101 => $\lambda = 1.0$



Hash functions

- Key mod T, when Key is an integer.
  - When is this bad?
- How to hash strings?
- What are desirable properties of a hash function?

Hash table example

T = 11
Keys are integers

Insert key 3: h(3) = 3 mod 11 = 3

…

Hash table example

T = 11
Keys are integers

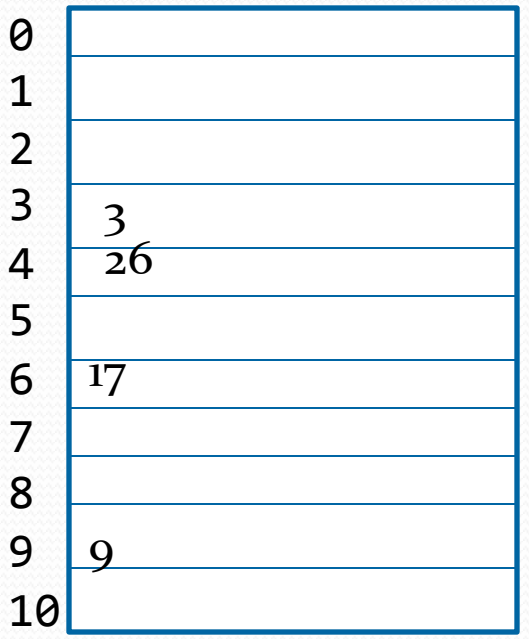
Insert key 3: h(3) = 3 mod 11 = 3

Insert 26: h(26) = 26 mod 11 = 4

Insert 9: h(9) = 9 mod 11 = 9

Insert 17: h(17) = 17 mod 11 = 6

$\lambda = 4 / 11 = 0.36$



…

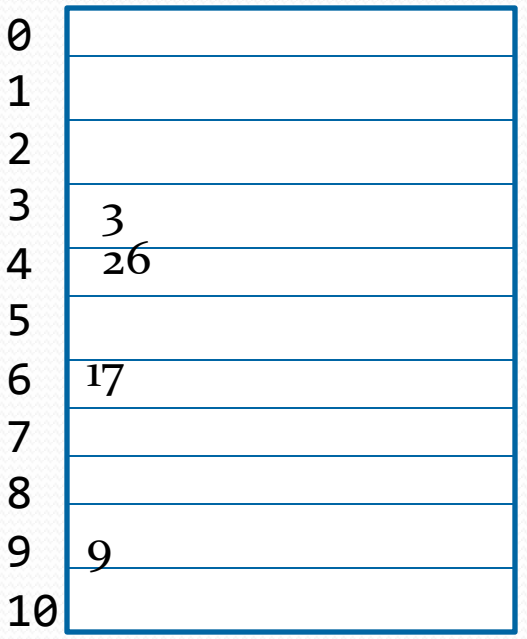
Hash table example

T = 11
Keys are integers

Find(26) ?

Find(18) ?

Find(39) ?



…

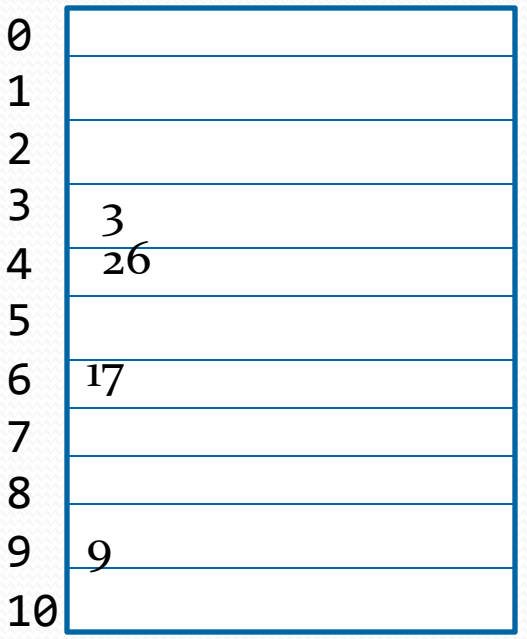
Hash table example

T = 11
Keys are integers

Find(26): h(26) = 26 mod 11 = 4
=> Found

Find(18): h(18) = 18 mod 11 = 7
=> Not Found

Find(39): h(39) = 39 mod 11 = 6
=> Not Found (why?)

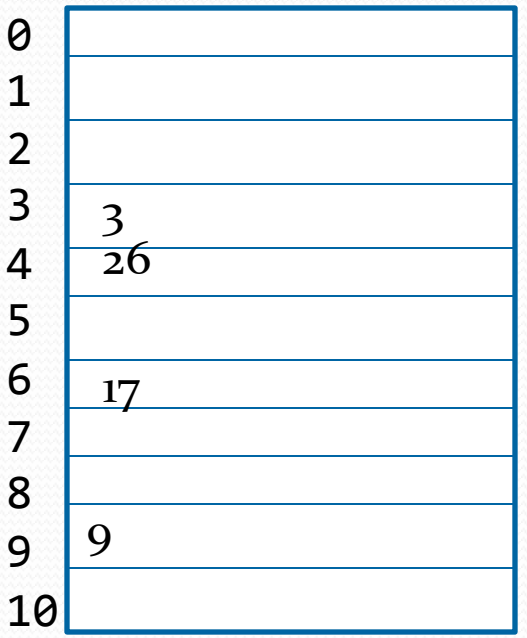


…

Hash table example

T = 11
Keys are integers

Insert 28: => ?

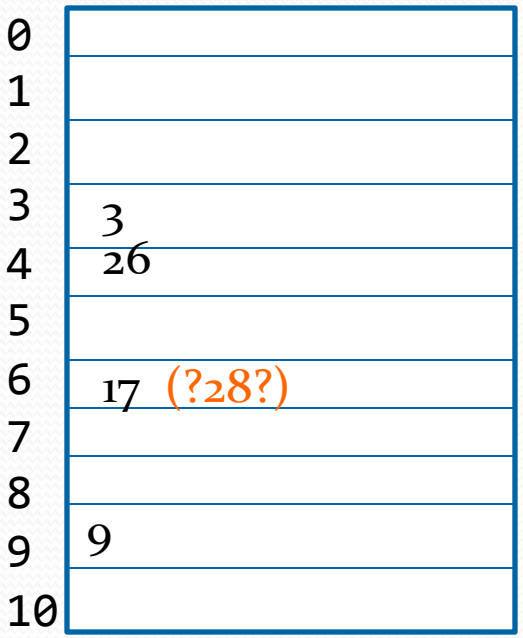


…

Hash table example

T = 11
Keys are integers

Insert 28: h(28) = 28 mod 11 = 6 :
Collision



…

Hash functions

- Key mod T, when Key is an integer.
  - When is this bad?
- How to hash strings?
- What are desirable properties of a hash function?
  - Efficient
  - Even distribution of keys in table

Hashing strings

A) Convert string **s** to integer **k**
B) Compute k mod T

i.e. h(**s**) = algorithm to convert string to positive integer

Hashing strings

- Bad idea 1:

  Sum ASCII chars

Suppose a char can range from 0 to 127 as integer

s = k0 k1 k2 k3(string of size 4)

Maximum value of hash function ?

Maximum number of strings of size 4?

Hashing strings

- Bad idea 1:

  Sum ASCII chars

Suppose a char can range from 0 to 127 as integer

s = k0 k1 k2 k3(string of size 4)

Maximum value of hash function = 4 * 127 = 508

Maximum number of strings of size 4 = 128^4 ~ 268M

Problem?

Hashing strings

- Bad Idea #1: Sum of ASCII chars mod tableSize
  - Values only range from 0 to 127 * # of chars
  - May not fully utilize the whole table
- Bad Idea #2: Quadratic function of first three chars
  - E.g. $c_0 + 37c_1 + 37^2c_2$ mod tableSize
    - (37 is number of letters, digits, and a white space character)
  - But how many different three letter prefixes are there?
  - Note: Using more characters is better but using too many will make evaluating hash function expensive.

Hashing strings

- Good: Polynomial using all the characters
  - Compute using Horner's method for $O(n)$
- $\text{hashVal} = k_3 + 37\big(k_2 + 37(k_1 + 37k_0)\big)$
  $= k_3 + 37k_2 + 37^2k_1 + 37^3k_0$

  Example: strings of 4 characters: ko k1 k2 k3
- Note: Using more characters is better but using too many will make evaluating hash function expensive.
- For long strings, compromise by using every other character. Beware of unintended consequences.

Hash function for string keys: an example of a good choice.

```
size_t hash(const string &key, size_t table_size) {
    size_t hash_value = 0;
    for (char ch : key)
        hash_value = 37 * hash_value + ch;
    return hash_value % table_size;
}

// size_t type can NOT hold negative values!
// Used for increasing loop variables, table sizes, etc.
// Can hold size of any table.
```

Hash function for string keys: not depending on table_size.

```
size_t hash(const string &key) {
    size_t hash_value = 0;
    for (char ch : key)
        hash_value = 37 * hash_value + ch;
    return hash_value;
}
// hash_value will be a large number.
// Calling function will mod with table_size.
```

Example

- S = k0 k1 k2 k3

h = 0

h = 37 * h + k0 = k0

h = 37 * h + k1 = 37 * k1 + k0

h = 37 * h + k2 = ….

h = 37 * h + k3 = …

Hash function for string keys

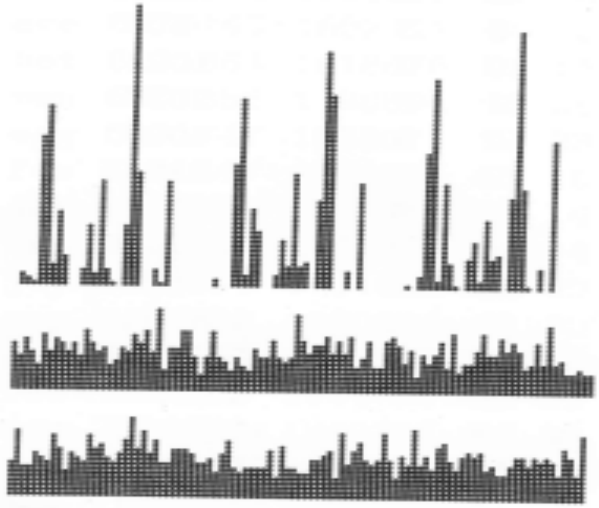


Figure 14.5
Hash functions for character strings

These diagrams show the dispersion for a set of English words (the first 1000 distinct words of Melville's Moby Dick) using Program 14.1 with
`M = 96` *and* `a = 128` (top)
`M = 97` *and* `a = 128` (center) *and*
`M = 96` *and* `a = 127` (bottom)

Program 14.1 Hash function for string keys

This implementation of a hash function for string keys involves one multiplication and one addition per character in the key. If we were to replace the constant 127 by 128, the program would simply compute the remainder when the number corresponding to the 7-bit ASCII representation of the key was divided by the table size, using Horner's method. The prime base 127 helps us to avoid anomalies if the table size is a power of 2 or a multiple of 2.

```
int hash(char *v, int M)
  { int h = 0, a = 127;
    for (; *v != 0; v++)
      h = (a*h + *v) % M;
    return h;
  }
```

Same function as in prev. slides. Implemented for c-strings.
Uses 127 instead of 37

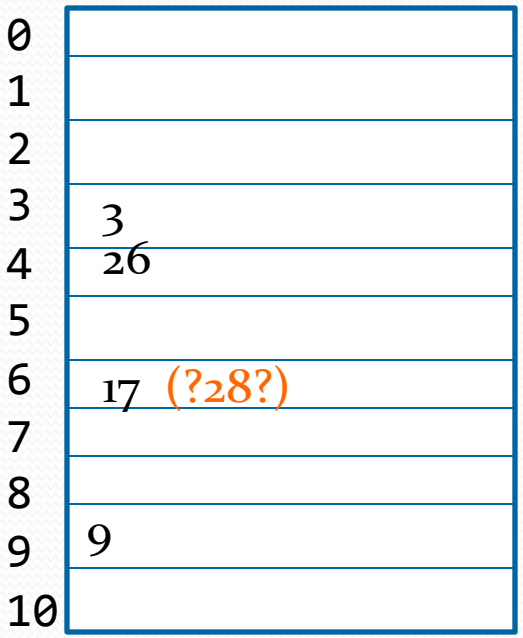
Algorithms in C++ by Robert Sedgewick

Good performance when M and a are relatively prime

Hash table example: collision

T = 11
Keys are integers

Insert 28: h(28) = 28 mod 11 = 6 :
Collision



…

Collision Resolution Strategies

Solution 1: SEPARATE CHAINING

Use a linked list for storing items that collide

Separate Chaining

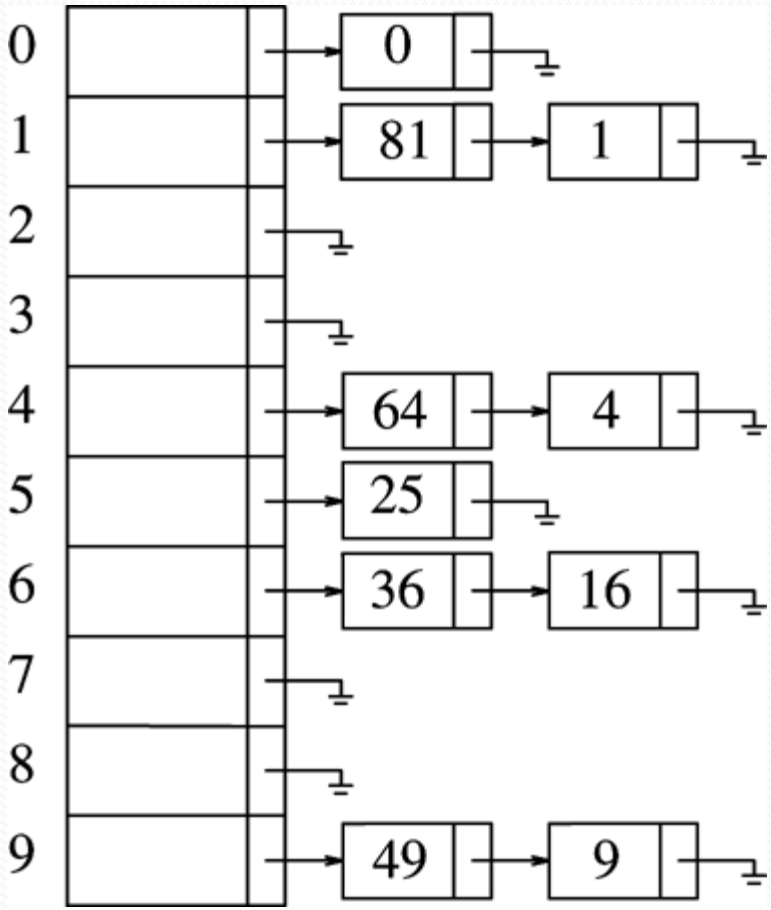


Table size T = 10

Insert: 0, 81, 25, 64, 36, 49, 4, 16, 9, 1

$\lambda = 10 / 10 = 1.0$

Separate Chaining

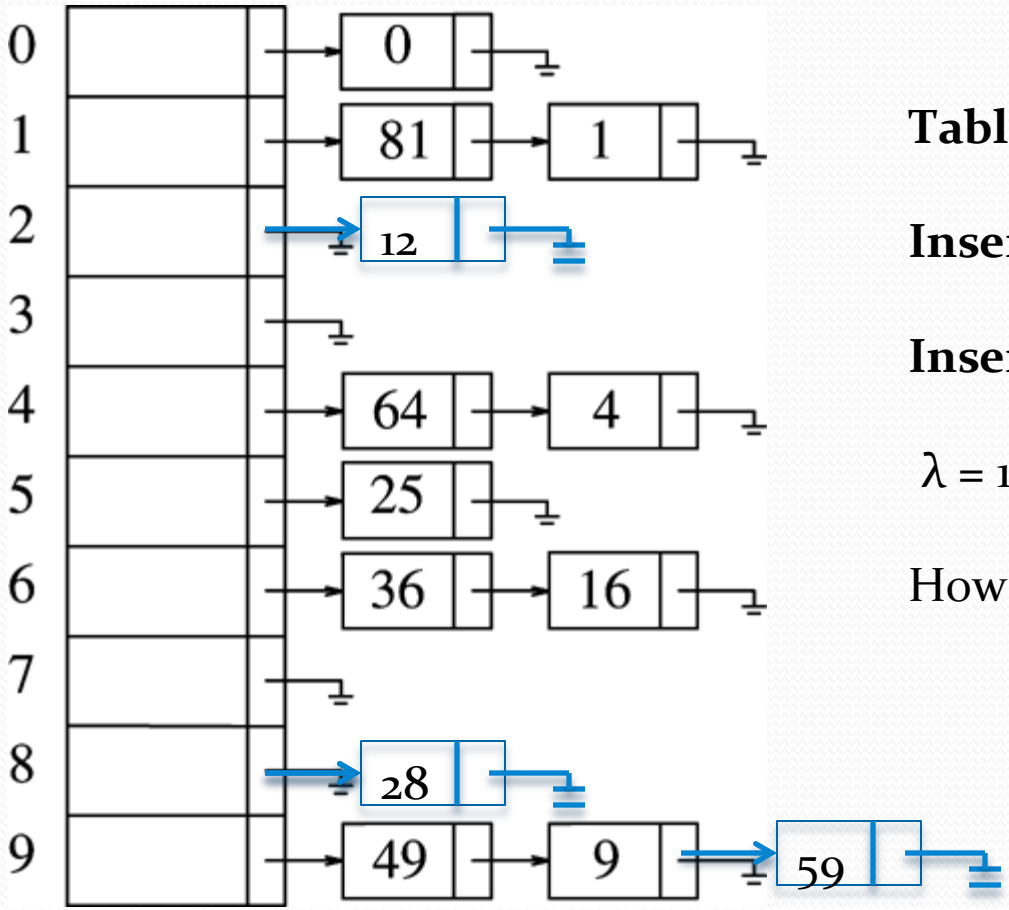


Table size T = 10

Insert: 0, 81, 25, 64, 36, 49, 4, 16, 9, 1

Insert 12, 28, 59

$\lambda = 13 / 10 = 1.3$ (greater than 1.0)

How do you search / delete?

Separate Chaining

- On average length of list is λ, the load factor
- Unsuccessful search (miss): $O(1) + \lambda$ on average
- Successful search (hit) : $O(1) + \lambda/2$
- λ is important, should try to keep it around 1
- Why list? Why not tree or another table?
- +/- of this strategy?

Collision Resolution Strategies

Solution 1: Probing Hash Tables

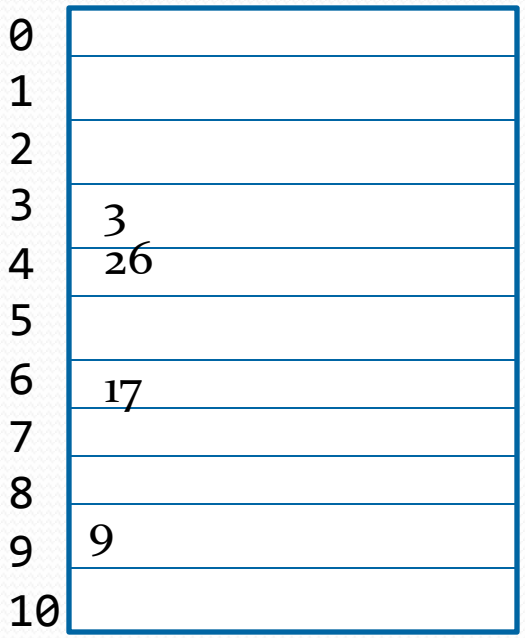
Use a simple table to store items

Hash table example: collision

T = 11
Keys are integers

Insert 28:
h(28) = 28 mod 11 = 17

Generate a probing sequence (mod T):
h(28) + 0 **COLLISION**
h(28) + f(1) ?
H(28) + f(2) ?



…

Probing Hash Tables

- Suppose that 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 TableSize

  with $f(0) = 0$

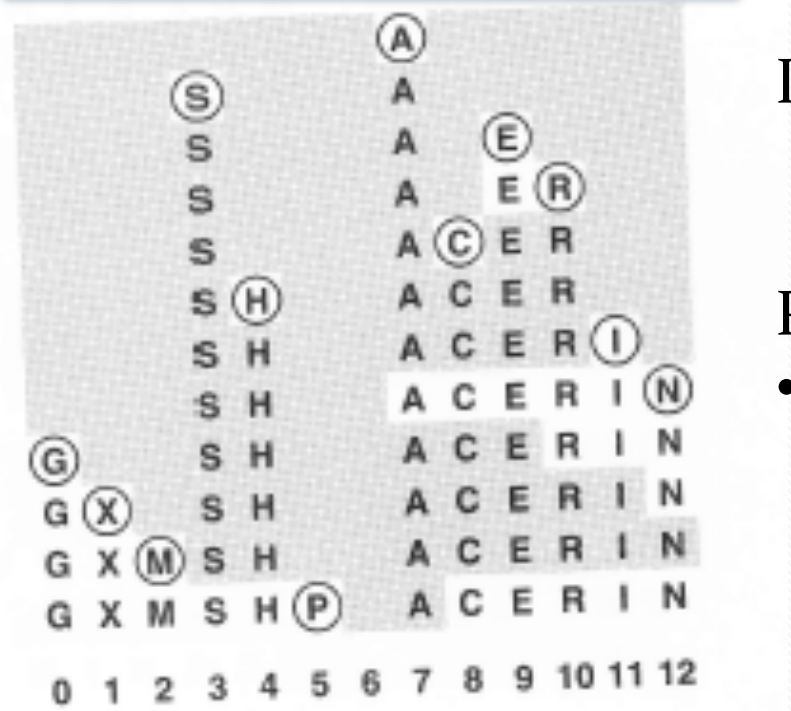
- Does not use additional memory outside of the table.

Linear Probing

Probing sequence:
$h_i(x) = (\text{hash}(x) + i) \bmod T$,
for $i = 0, 1, \ldots$, until spot is found.

hash(x):

A	S	E	R	C	H	I	N	G	X	M	P
7	3	9	9	8	4	11	7	10	12	0	8



Algorithms in C++ by Robert Sedgewick

Insert A,S,E,R,.... in table of size T =13

Problem: **Primary Clustering**

- Collisions in a crowded range will increase the number of collisions in that range.

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)

Theorem:

Expected # of probes in miss = Expected # of probes to find empty cell
= $1/(1-\lambda)$.

- **Proof**:
  - Probability{Selecting an empty cell} = $1-\lambda = p$ = Prob. of success
  - Probability{Selecting a non-empty cell} = $\lambda = 1 - p$ = 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 = $1/(1-\lambda)$
    - Check some values: $\lambda = 0$, $\lambda = 0.3$, $\lambda = 0.5$, $\lambda = 0.7$, $\lambda = 0.9$, $\lambda = 1$

Random collision resolution

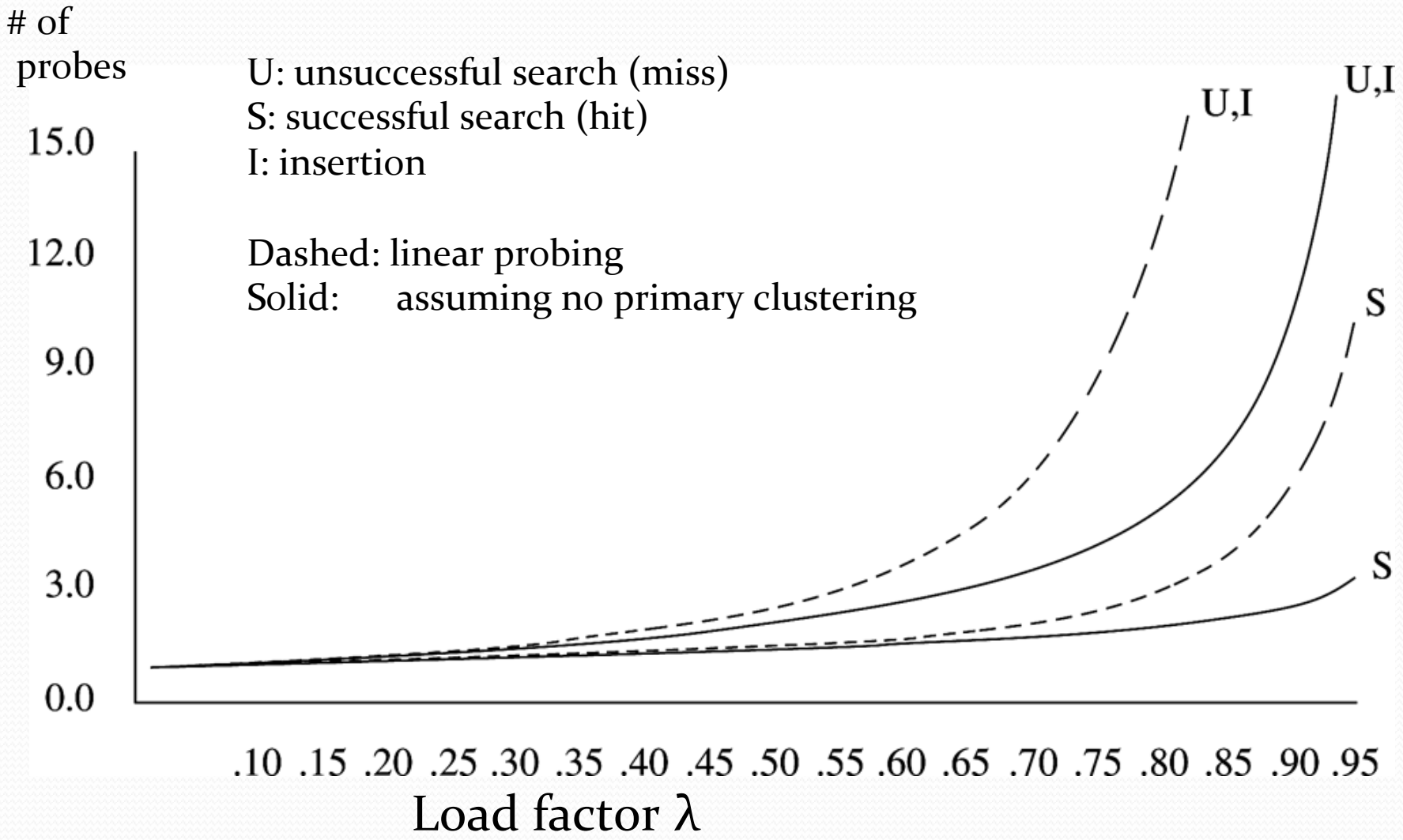
- Expected # of probes for insert = Expected # of probes for miss (why ?)

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

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:
  $hi(x) = (hash(x) + i^2) \bmod T$, for $i = 0, 1, \ldots$, 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 $\lceil TableSize/2 \rceil$ alternative locations, including the initial $h_0(x)$, are all distinct.

Quadratic probing

Two of these locations are $h(x) + i^2 \pmod{TableSize}$ and $h(x) + j^2 \pmod{TableSize}$, where $0 \leq i, j \leq \lfloor TableSize/2 \rfloor$. Assume towards contradiction that these locations are the same but $i \neq j$. Then

$$\begin{aligned} h(x) + i^2 &= h(x) + j^2 \pmod{TableSize} \\ i^2 &= j^2 \pmod{TableSize} \\ i^2 - j^2 &= 0 \pmod{TableSize} \\ (i - j)(i + j) &= 0 \pmod{TableSize} \end{aligned}$$

Quadratic probing

Since $TableSize$ is prime then either $(i - j)$ or $(i + j)$ is equal to 0 mod $TableSize$. Since $i \neq j$ the first option is not possible.

Since $0 \leq i, j \leq \lfloor TableSize/2 \rfloor$, the second option is also impossible.

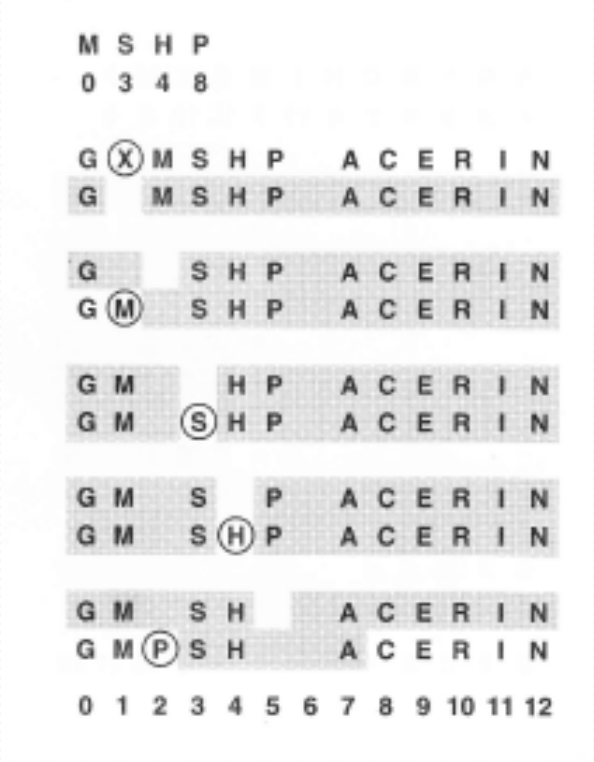
Thus the first $\lceil TableSize/2 \rceil$ alternate locations are distinct. If at most $\lfloor TableSize/2 \rfloor$ positions are taken, an empty spot can always be found.

□

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.

Removal (linear probing)



- Without lazy deletion
- What needs to be fixed when we remove X?

Algorithms in C++ by Robert Sedgewick

Double Hashing

Example of second hash function:
hash2(x) = R – (x mod R)
R: prime
R < table_size

- Sequence of probes:
  Probe(i) = (hash(x) + i * hash2(x)) mod T
  - needs care
  - Should not evaluate to zero
  - R should also be prime
  - What if we insert 23 next?

	Empty Table	After 89	After 18	After 49	After 58	After 69
0	hash2(x) = 7 – (x mod 7)					69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Effects of clustering

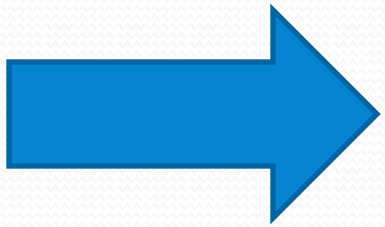
Algorithms in C++ by Robert Sedgewick

Top: key distribution
Middle: Linear Probing
Bottom: Double Hashing

Rehashing

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

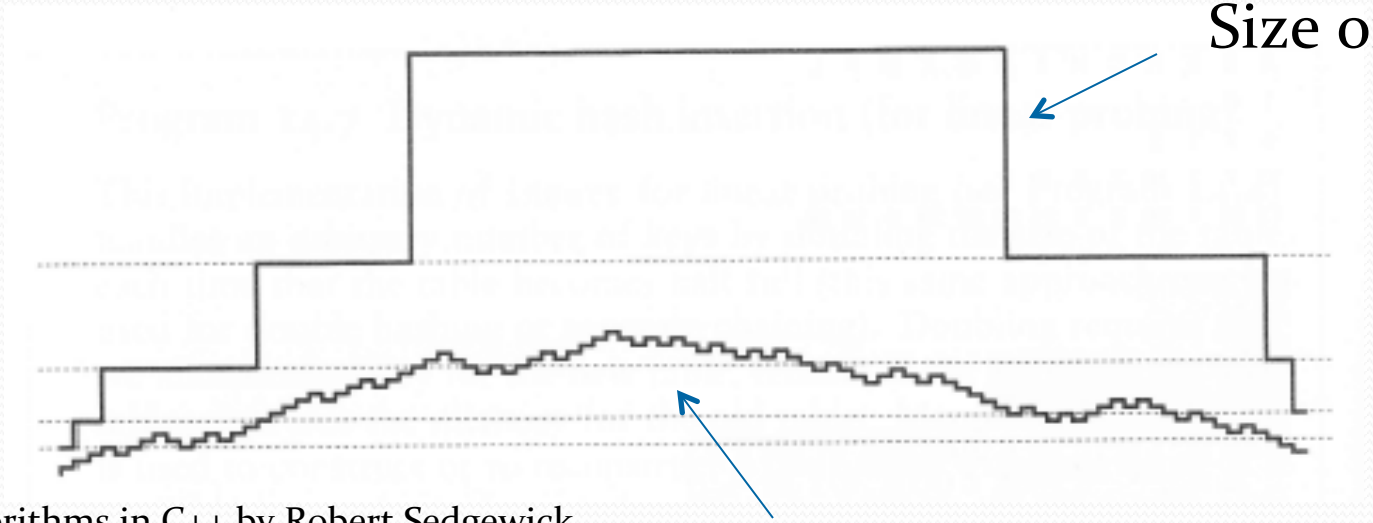
0	6
1	15
2	23
3	24
4	
5	
6	13



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

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

Rehashing example



Algorithms in C++ by Robert Sedgewick

Keeping a hash table between 1/8 to 1/2 full

Hash function for a general key type (C++11) – function object template

```
template <typename Key>
class hash {
 public:
  size_t operator()(const Key &key) {
     // Return a hash_value before performing
     // mod with the table_size.
     // table_size is uknown.
     // We will see example in a few slides.
  }
}
```

Hash function for string keys (C++11)

: CHANGING DEFAULT HASH FUNCTION

Without this code the default STL hash function for strings will be used.

```
template <>
class hash<string> {
size_t operator()(const string & key) {
    size_t hash_value = 0;
    for (char ch : key)
        hash_value = 37 * hash_value + ch;
    return hash_value;
}

// How to use the above?
```

Defining a hash functions for a user-defined type in C++11.

```
// Assume that a class Employee exists.
// Defining a hash function for a user-defined type Employee.
template<>
class hash<Employee> {
  public:
    size_t operator()(const Employee &item) const {
        static hash<string> hf;
        return hf(item.GetName());
    }
};

// This is within the implementation of a hash-table.
// See complete code. (called myhash in book).
size_t InternalHash(const HashedObj & x) const {
    static hash<HashedObj> hash_functional;  // Why static?
    return hash_functional(x) % table_size_;
}
```

```
// Example of very simple class Employee. This class can be a HashedObj type in
// the implementation of hash tables. You can for instance declare a hash table
// HashTable<Employee> hash_of_employees
// provided that its hash function has been declared as in the previous slide.
// Note that you can also use STL's hash-like type:
// unordered_set<Employee> unordered_set_of_employees;
class Employee {
  public:
    Employee(const string & name, double salary):  name_{name}, salary_{salary} { }
    // = default, for big five.
    const string & GetName( ) const { return name_; }
    void print(ostream & out) const { out << name_ << " (" << salary_ << ")"; }
    bool operator==(const Employee & rhs) const {  // <- required for unordered_set/map
          return name_ == rhs.name_;
    }
    // bool operator!=(const Employee & rhs) const {  <-not required
    //    return !(*this == rhs);
    // }
    // More public functions maybe needed.

  private:
    // This is the key and it should be unique for each Employee.
    string name_;
    double salary_;
    // .. More information possible needed.
};
```

Stl’s unordered_set/map

- unordered_set
- unordered_map
- Same functionality as set and map, but no ordered capabilities.

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 consise

```
// 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

- If use separate chaining, and assume load factor 1.
- What is the worst-case access time?

Worst-Case Access

- If use separate chaining, and assume load factor 1.
- What is the worst-case access time?
- 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

- If use separate chaining, and assume load factor 1.
- What is the worst-case access time?
- Problem is formulated as:

  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:

  $\Theta(\log N / \log\log N)$
- Is this O(1) ?

Worst-Case Access O(1)

- Perfect Hashing provides a solution, sect. 5.7
  - We will not cover it.

Extensible Hashing

- Hash table is huge => store on disk
  - N records to store
  - M records fit in one disk block
    - M < N
  - Solution?

Extendible Hashing

- Hash table is huge => stored on disk
- N records to store
- M records fit in one disk block
- Regular hashing?
  - (a) Collisions may require many disk accesses
  - (b) Rehashing is extremely expensive

    O(N) disk accesses
- **Extendible hashing**: 2 disk accesses for search

Extendible Hashing

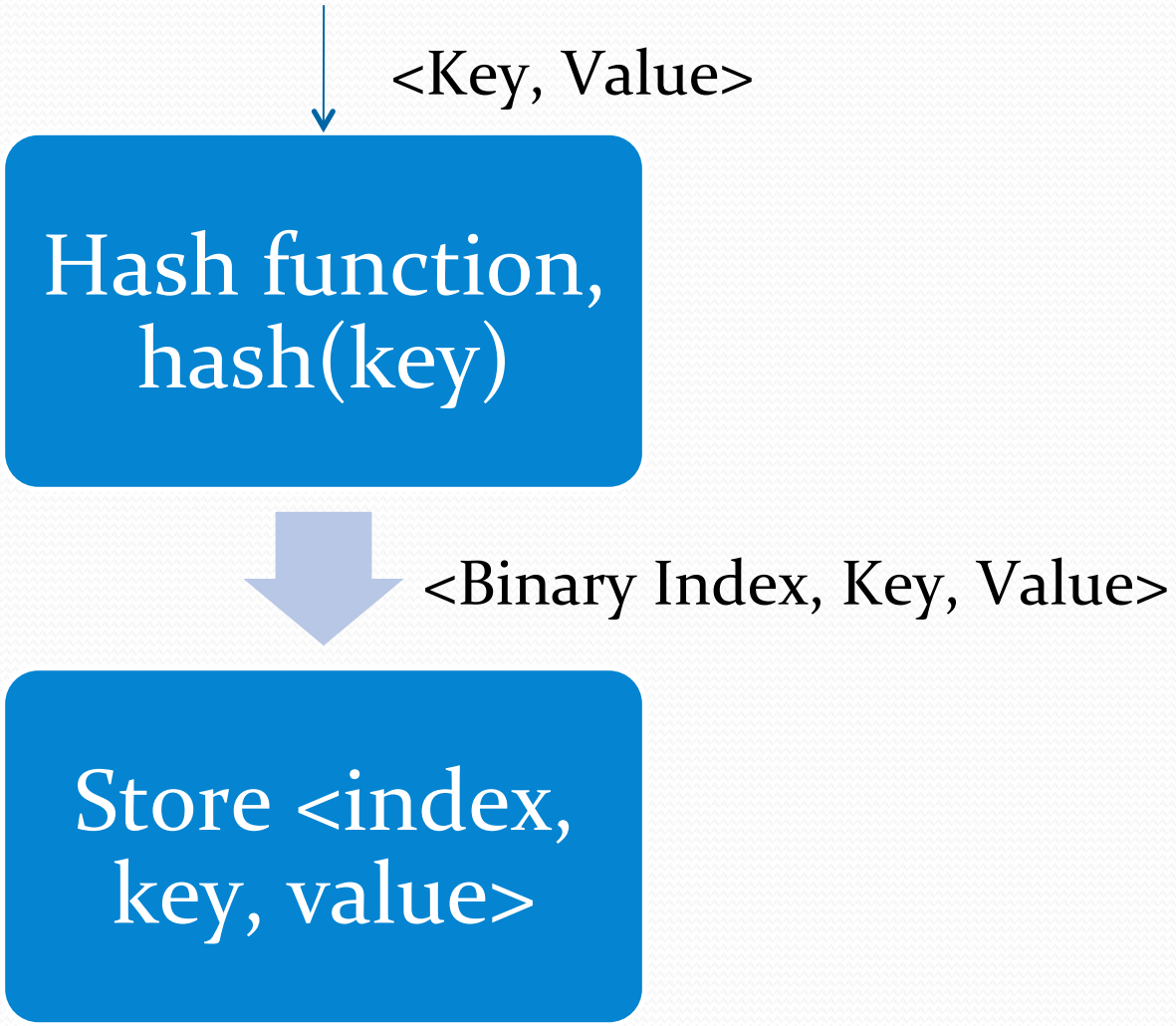
- B-tree approach?
  - Depth is $\log_{\frac{K}{2}} N$, (K is the branching factor)
  - Can we make its depth be 2?

- 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

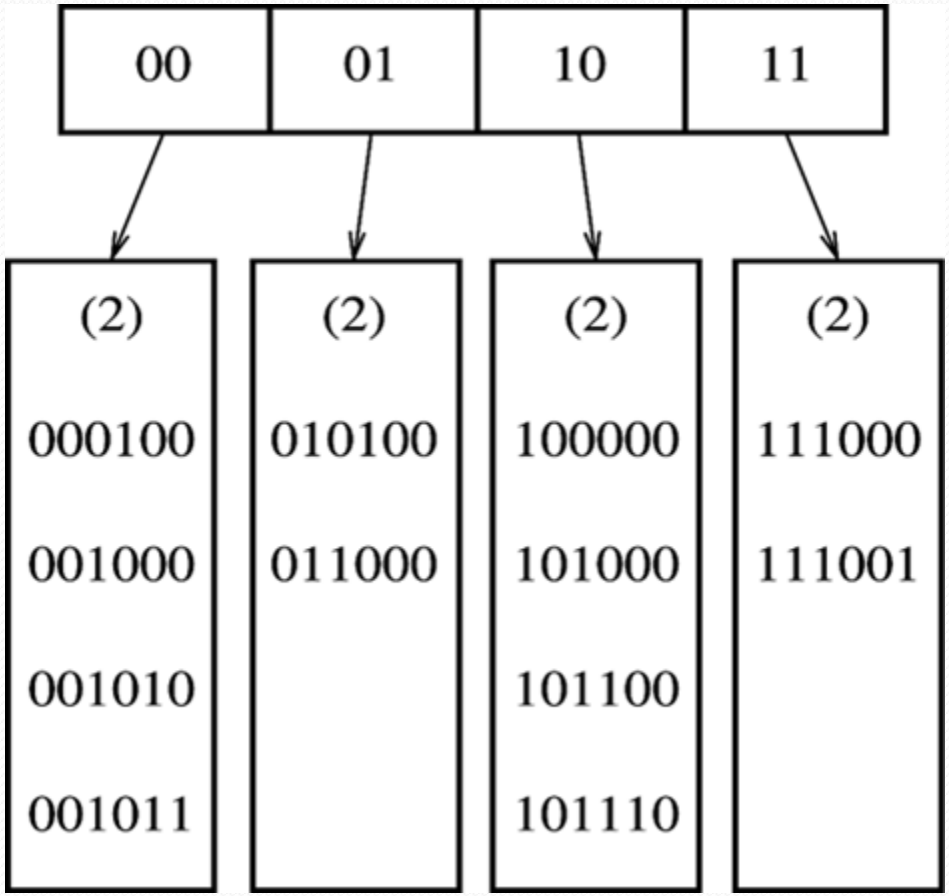
E.H.

- 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

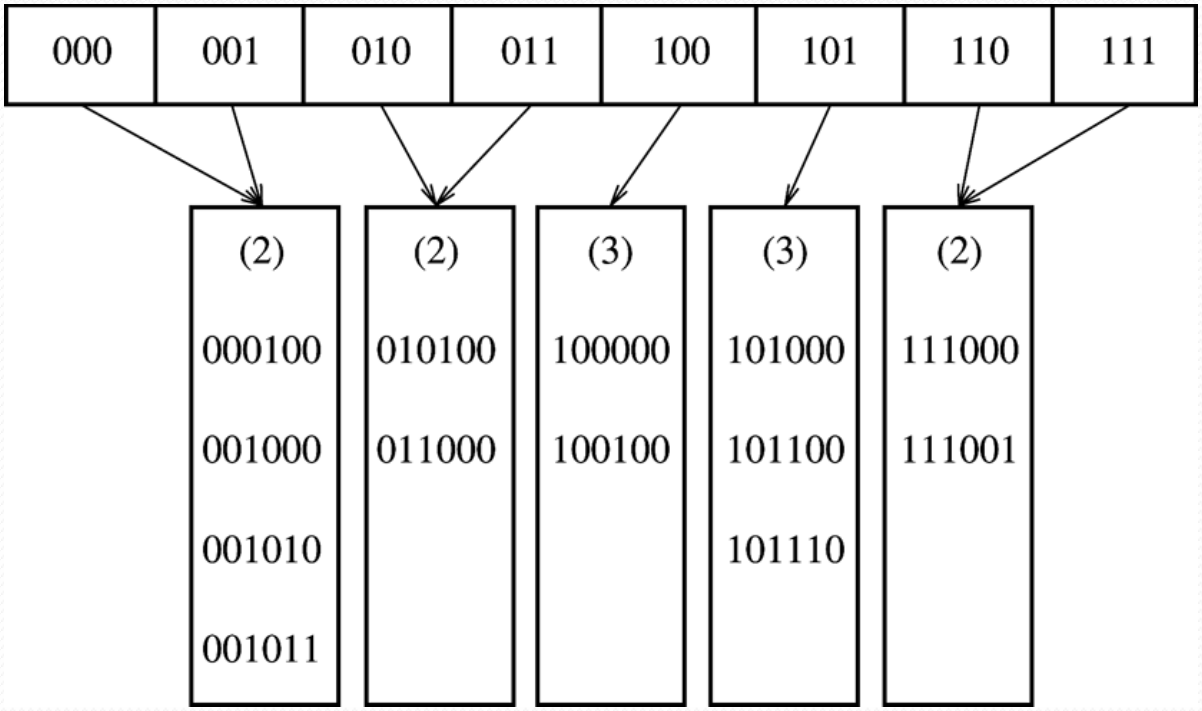


E.H.

- Insert 100100

=> Directory split

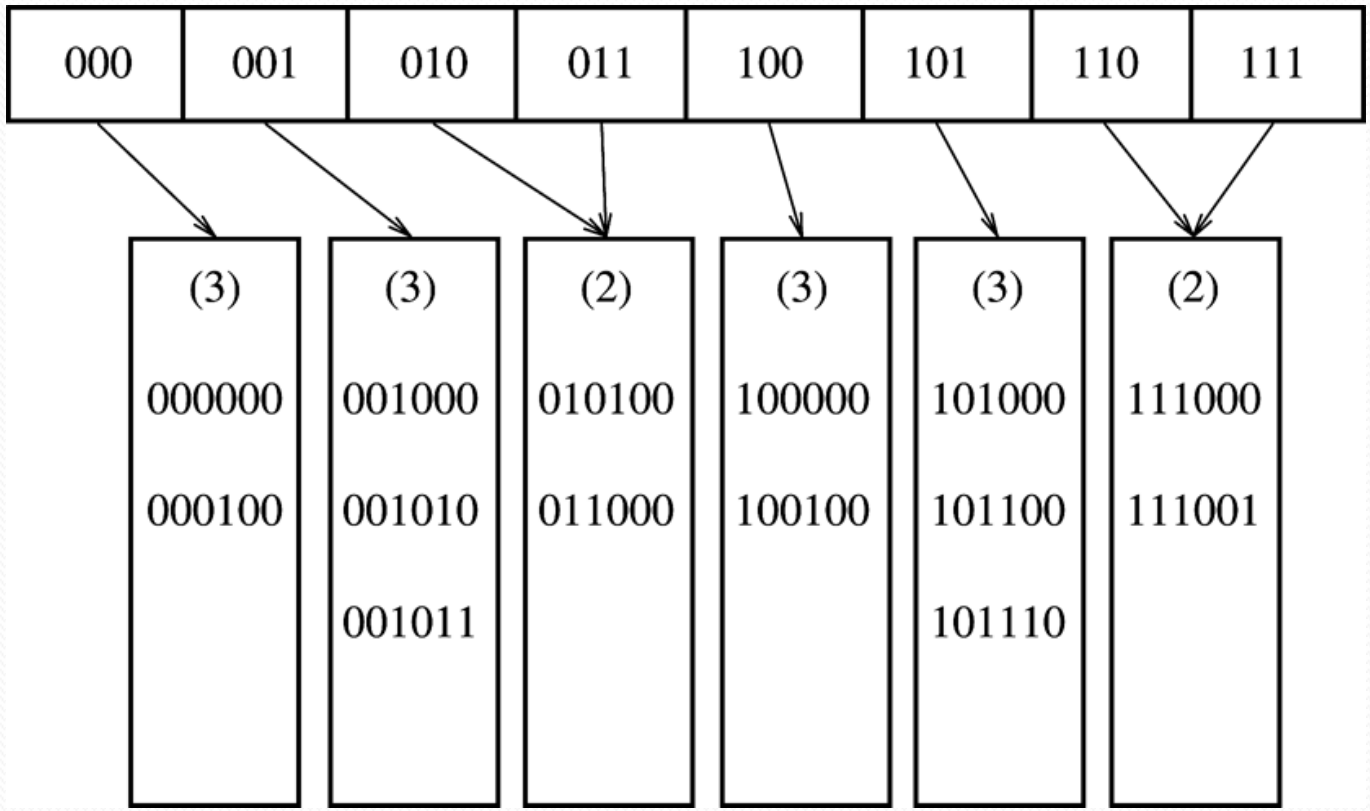
- Changes?



- ...Insert oooooo

E.H.

- Changes?


000
001
010
011
100
101
110
111
(3)
000000
000100
(3)
001000
001010
001011
(2)
010100
011000
(3)
100000
100100
(3)
101000
101100
101110
(2)
111000
111001

E.H.

- 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

E.H. 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)

E.H. performance

- 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

- M=10

  $\Rightarrow \leq c * 7.94.. * 10^8$ entries

- The smaller the M the larger the directory size

E.H. performance

- Leaves could contain pointers to records.

Performance?

- A huge directory can be stored in disk

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)
    - In BST search no need for expensive mult/div operations
- If no ordering is required, hash is probably better
  - Can try both to see which is better in practice.
  - 1 second difference for the 1-letter replacement word problem.

Hash Applications

- Symbol tables (compilers)
- Graphs where nodes are strings (e.g. names of cities)
- Spell-checkers
- Password checking
- …