



Hashing

CptS 223 – Advanced Data Structures

Larry Holder

School of Electrical Engineering and Computer Science
Washington State University



Overview

- Hashing
 - Technique supporting insertion, deletion and search in average-case constant time
 - Operations requiring elements to be sorted (e.g., FindMin) are not efficiently supported
- Hash table ADT
 - Implementations
 - Analysis
 - Applications



Hash Table

- One approach
 - Hash table is an array of fixed size TableSize
 - Array elements indexed by a key, which is mapped to an array index (0...TableSize-1)
 - Mapping (hash function) h from key to index
 - E.g., $h(\text{"john"}) = 3$

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



Hash Table

- Insert
 - $T[h(\text{"john"})] = \langle \text{"john"}, 25000 \rangle$
- Delete
 - $T[h(\text{"john"})] = \text{NULL}$
- Search
 - Return $T[h(\text{"john"})]$
- What if $h(\text{"john"}) = h(\text{"joe"})$?

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



Hash Function

- Mapping from key to array index is called a hash function
 - Typically, many-to-one mapping
 - Different keys map to different indices
 - Distributes keys evenly over table
- Collision occurs when hash function maps two keys to same array index



Hash Function

- Simple hash function
 - $h(\text{Key}) = \text{Key} \bmod \text{TableSize}$
 - Assumes integer keys
- For random keys, $h()$ distributes keys evenly over table
- What if $\text{TableSize} = 100$ and keys are multiples of 10?
- Better if TableSize is a prime number
 - Not too close to powers of 2 or 10



Hash Function for String Keys

- Approach 1
 - Add up character ASCII values (0-127) to produce integer keys
 - Small strings may not use all of table
 - $\text{Strlen}(S) * 127 < \text{TableSize}$
- Approach 2
 - Treat first 3 characters of string as base-27 integer (26 letters plus space)
 - $\text{Key} = S[0] + (27 * S[1]) + (27^2 * S[2])$
 - Assumes first 3 characters randomly distributed
 - Not true of English

Hash Function for String Keys

- Approach 3

- Use all N characters of string as an N-digit base-K integer
- Choose K to be prime number larger than number of different digits (characters)
 - I.e., K = 29, 31, 37
- If L = length of string S, then

$$h(S) = \left[\sum_{i=0}^{L-1} S[L-i-1] * 37^i \right] \bmod TableSize$$

- Use Horner's rule to compute h(S)
- Limit L for long strings

```
1  /**
2   * A hash routine for string objects.
3   */
4  int hash( const string & key, int tableSize )
5  {
6      int hashVal = 0;
7
8      for( int i = 0; i < key.length( ); i++ )
9          hashVal = 37 * hashVal + key[ i ];
10
11     hashVal %= tableSize;
12     if( hashVal < 0 )
13         hashVal += tableSize;
14
15     return hashVal;
16 }
```

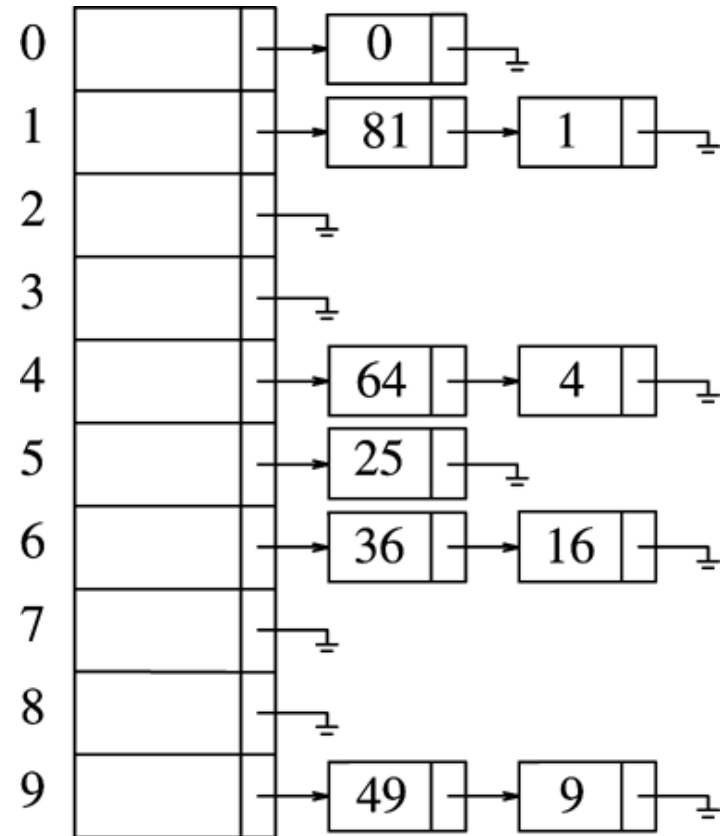


Collision Resolution

- What happens when $h(k_1) = h(k_2)$?
- Collision resolution strategies
 - Chaining
 - Store colliding keys in a linked list
 - Open addressing
 - Store colliding keys elsewhere in the table

Collision Resolution by Chaining

- Hash table T is a vector of lists
 - Only singly-linked lists needed if memory is tight
- Key k is stored in list at $T[h(k)]$
- E.g., $TableSize = 10$
 - $h(k) = k \bmod 10$
 - Insert first 10 perfect squares



Implementation of Chaining Hash Table

```
1  template <typename HashedObj>
2  class HashTable
3  {
4  public:
5      explicit HashTable( int size = 101 );
6
7      bool contains( const HashedObj & x ) const;
8
9      void makeEmpty( );
10     void insert( const HashedObj & x );
11     void remove( const HashedObj & x );
12
13 private:
14     vector<list<HashedObj> > theLists;    // The array of Lists
15     int  currentSize;
16
17     void rehash( );
18     int myhash( const HashedObj & x ) const;
19 };
20
21 int hash( const string & key );
22 int hash( int key );
```

Generic hash functions for integers and keys

Implementation of Chaining Hash Table

```
1      int myhash( const HashedObj & x ) const
2      {
3          int hashVal = hash( x );
4
5          hashVal %= theLists.size( );
6          if( hashVal < 0 )
7              hashVal += theLists.size( );
8
9          return hashVal;
10     }
```

```

1 void makeEmpty( )
2 {
3     for( int i = 0; i < theLists.size( ); i++ )
4         theLists[ i ].clear( );
5 }
6
7 bool contains( const HashedObj & x ) const
8 {
9     const list<HashedObj> & whichList = theLists[ myhash( x ) ];
10    return find( whichList.begin( ), whichList.end( ), x ) != whichList.end( );
11 }
12
13 bool remove( const HashedObj & x )
14 {
15     list<HashedObj> & whichList = theLists[ myhash( x ) ];
16     list<HashedObj>::iterator itr = find( whichList.begin( ), whichList.end( ), x );
17
18     if( itr == whichList.end( ) )
19         return false;
20
21     whichList.erase( itr );
22     --currentSize;
23     return true;
24 }

```

STL algorithm: find

Each of these operations takes time linear in the length of the list.

```
1  bool insert( const HashedObj & x )
2  {
3      list<HashedObj> & whichList = theLists[ myhash( x ) ];
4      if( find( whichList.begin( ), whichList.end( ), x ) != whichList.end( ) )
5          return false;
6      whichList.push_back( x );
7
8          // Rehash; see Section 5.5
9      if( ++currentSize > theLists.size( ) )
10         rehash( );
11
12     return true;
13 }
```

No duplicates

Later, but essentially
doubles size of table and
reinserts current elements.

```
1 // Example of an Employee class
2 class Employee
3 {
4     public:
5         const string & getName( ) const
6             { return name; }
7
8         bool operator==( const Employee & rhs ) const
9             { return getName( ) == rhs.getName( ); }
10        bool operator!=( const Employee & rhs ) const
11            { return !( *this == rhs; }
12
13            // Additional public members not shown
14
15        private:
16            string name;
17            double salary;
18            int    seniority;
19
20            // Additional private members not shown
21 };
22
23 int hash( const Employee & item )
24 {
25     return hash( item.getName( ) );
26 }
```

All hash objects must define == and != operators.

Hash function to handle Employee object type



Collision Resolution by Chaining: Analysis

- Load factor λ of a hash table T
 - N = number of elements in T
 - M = size of T
 - $\lambda = N/M$
- Average length of a chain is λ
- Unsuccessful search $O(\lambda)$
- Successful search $O(\lambda/2)$
- Ideally, want $\lambda \approx 1$ (not a function of N)
 - I.e., TableSize = number of elements you expect to store in the table



Collision Resolution by Open Addressing

- When a collision occurs, look elsewhere in the table for an empty slot
- Advantages over chaining
 - No need for addition list structures
 - No need to allocate/deallocate memory during insertion/deletion (slow)
- Disadvantages
 - Slower insertion – May need several attempts to find an empty slot
 - Table needs to be bigger (than chaining-based table) to achieve average-case constant-time performance
 - Load factor $\lambda \approx 0.5$



Collision Resolution by Open Addressing

- Probe sequence
 - Sequence of slots in hash table to search
 - $h_0(x), h_1(x), h_2(x), \dots$
 - Needs to visit each slot exactly once
 - Needs to be repeatable (so we can find/delete what we've inserted)
- Hash function
 - $h_i(x) = (h(x) + f(i)) \bmod \text{TableSize}$
 - $f(0) = 0$



Linear Probing

- $f(i)$ is a linear function of i
 - E.g., $f(i) = i$
- Example: $h(x) = x \bmod \text{TableSize}$
 - $h_0(89) = (h(89) + f(0)) \bmod 10 = 9$
 - $h_0(18) = (h(18) + f(0)) \bmod 10 = 8$
 - $h_0(49) = (h(49) + f(0)) \bmod 10 = 9$ (X)
 - $h_1(49) = (h(49) + f(1)) \bmod 10 = 0$



Linear Probing Example

	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: Analysis

- Probe sequences can get long
- Primary clustering
 - Keys tend to cluster in one part of table
 - Keys that hash into cluster will be added to the end of the cluster (making it even bigger)



Linear Probing: Analysis

- Expected number of probes for insertion or unsuccessful search

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

- Expected number of probes for successful search

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

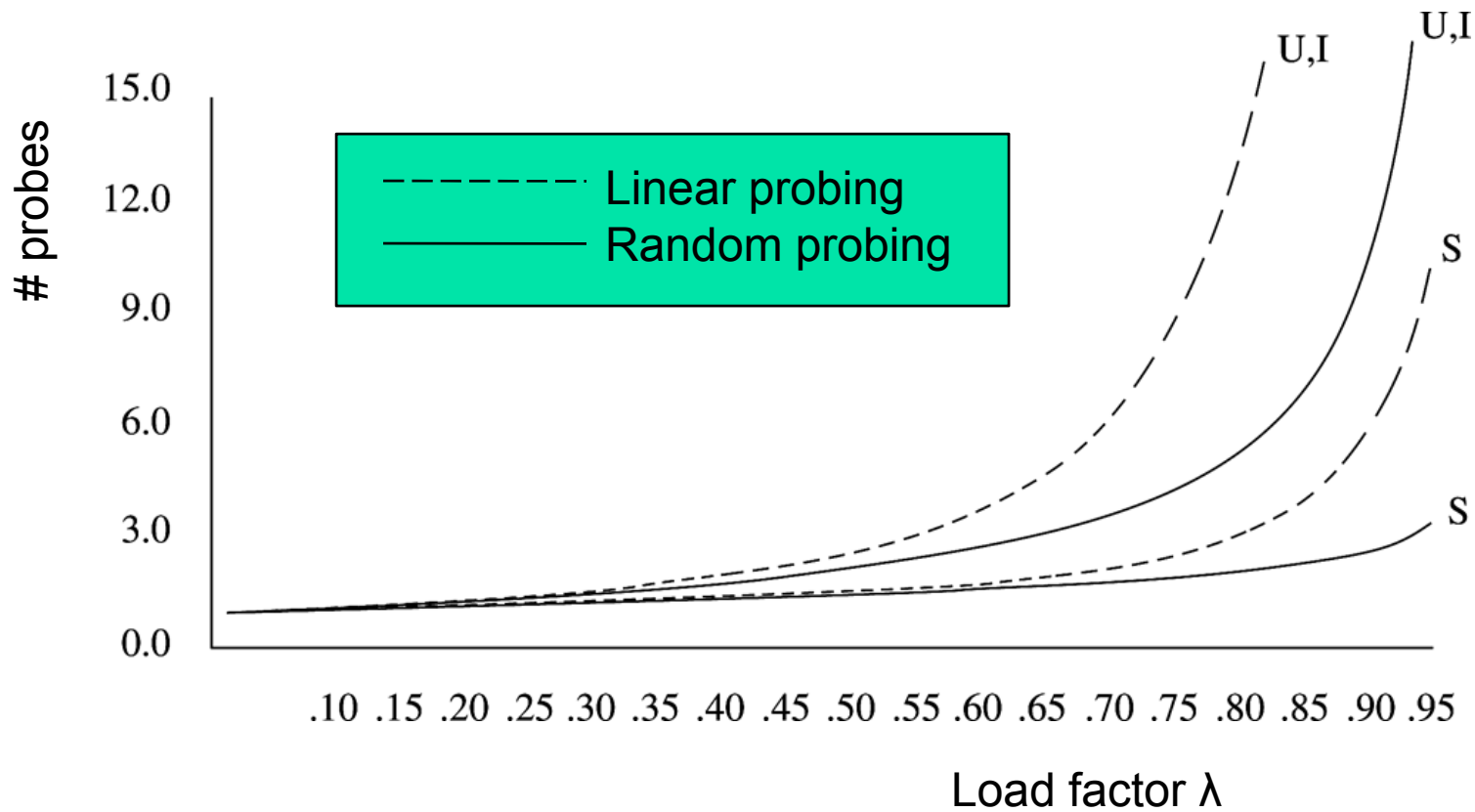
- Example ($\lambda = 0.5$)
 - Insert / unsuccessful search
 - 2.5 probes
 - Successful search
 - 1.5 probes
- Example ($\lambda = 0.9$)
 - Insert / unsuccessful search
 - 50.5 probes
 - Successful search
 - 5.5 probes



Random Probing: Analysis

- Random probing does not suffer from clustering
- Expected number of probes for insertion or unsuccessful search: $\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$
- Example
 - $\lambda = 0.5$: 1.4 probes
 - $\lambda = 0.9$: 2.6 probes

Linear vs. Random Probing





Quadratic Probing

- Avoids primary clustering
- $f(i)$ is quadratic in i
 - E.g., $f(i) = i^2$
- Example
 - $h_0(58) = (h(58)+f(0)) \bmod 10 = 8$ (X)
 - $h_1(58) = (h(58)+f(1)) \bmod 10 = 9$ (X)
 - $h_2(58) = (h(58)+f(2)) \bmod 10 = 2$



Quadratic Probing Example

	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: Analysis

- Difficult to analyze
- Theorem 5.1
 - New element can always be inserted into a table that is at least half empty and TableSize is prime
- Otherwise, may never find an empty slot, even if one exists
- Ensure table never gets half full
 - If close, then expand it



Quadratic Probing

- Only M (TableSize) different probe sequences
 - May cause “secondary clustering”
- Deletion
 - Emptying slots can break probe sequence
 - Lazy deletion
 - Differentiate between empty and deleted slot
 - Skip deleted slots
 - Slows operations (effectively increases λ)

Quadratic Probing: Implementation

```
1  template <typename HashedObj>
2  class HashTable
3  {
4      public:
5          explicit HashTable( int size = 101 );
6
7          bool contains( const HashedObj & x ) const;
8
9          void makeEmpty( );
10         bool insert( const HashedObj & x );
11         bool remove( const HashedObj & x );
12
```

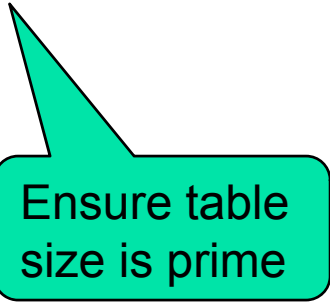
Quadratic Probing: Implementation

```
13     enum EntryType { ACTIVE, EMPTY, DELETED };
14
15     private:
16     struct HashEntry
17     {
18         HashedObj element;
19         EntryType info;
20
21         HashEntry( const HashedObj & e = HashedObj( ), EntryType i = EMPTY )
22             : element( e ), info( i ) { }
23     };
24
25     vector<HashEntry> array;
26     int currentSize;
27
28     bool isActive( int currentPos ) const;
29     int findPos( const HashedObj & x ) const;
30     void rehash( );
31     int myhash( const HashedObj & x ) const;
32 };
```

Lazy deletion

Quadratic Probing: Implementation

```
1  explicit HashTable( int size = 101 ) : array( nextPrime( size ) )
2      { makeEmpty( ); }
3
4  void makeEmpty( )
5  {
6      currentSize = 0;
7      for( int i = 0; i < array.size( ); i++ )
8          array[ i ].info = EMPTY;
9  }
```



Ensure table
size is prime

Quadratic Probing: Implementation

```
1  bool contains( const HashedObj & x ) const
2      { return isActive( findPos( x ) ); }
3
4  int findPos( const HashedObj & x ) const
5  {
6      int offset = 1;
7      int currentPos = myhash( x );
8
9      while( array[ currentPos ].info != EMPTY &&
10           array[ currentPos ].element != x )
11      {
12          currentPos += offset; // Compute ith probe
13          offset += 2;
14          if( currentPos >= array.size( ) )
15              currentPos -= array.size( );
16      }
17
18      return currentPos;
19  }
20
21  bool isActive( int currentPos ) const
22      { return array[ currentPos ].info == ACTIVE; }
```

Find

Skip DELETED;
No duplicates

Quadratic probe
sequence (really)

Quadratic Probing: Implementation

```
1  bool insert( const HashedObj & x )
2  {
3      // Insert x as active
4      int currentPos = findPos( x );
5      if( isActive( currentPos ) )
6          return false;
7
8      array[ currentPos ] = HashEntry( x, ACTIVE );
9
10     // Rehash; see Section 5.5
11     if( ++currentSize > array.size( ) / 2 )
12         rehash( );
13
14     return true;
15 }
16
17 bool remove( const HashedObj & x )
18 {
19     int currentPos = findPos( x );
20     if( !isActive( currentPos ) )
21         return false;
22
23     array[ currentPos ].info = DELETED;
24     return true;
25 }
```

Insert

No duplicates

Remove

No deallocation
needed



Double Hashing

- Combine two different hash functions
- $f(i) = i * h_2(x)$
- Good choices for $h_2(x)$?
 - Should never evaluate to 0
 - $h_2(x) = R - (x \bmod R)$
 - R is prime number less than TableSize
- Previous example with $R=7$
 - $h_0(49) = (h(49)+f(0)) \bmod 10 = 9$ (X)
 - $h_1(49) = (h(49)+(7 - 49 \bmod 7)) \bmod 10 = 6$



Double Hashing Example

	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: Analysis

- Imperative that TableSize is prime
 - E.g., insert 23 into previous table
- Empirical tests show double hashing close to random hashing
- Extra hash function takes extra time to compute



Rehashing

- Increase the size of the hash table when load factor too high
- Typically expand the table to twice its size (but still prime)
- Reinsert existing elements into new hash table

Rehashing Example

$h(x) = x \bmod 7$
 $\lambda = 0.57$

0	6
1	15
2	
3	24
4	
5	
6	13

Insert 23
 $\lambda = 0.71$

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

$h(x) = x \bmod 17$
 $\lambda = 0.29$

Rehashing
→

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



Rehashing Analysis

- Rehashing takes $O(N)$ time
- But happens infrequently
- Specifically
 - Must have been $N/2$ insertions since last rehash
 - Amortizing the $O(N)$ cost over the $N/2$ prior insertions yields only constant additional time per insertion



Rehashing Implementation

- When to rehash
 - When table is half full ($\lambda = 0.5$)
 - When an insertion fails
 - When load factor reaches some threshold
- Works for chaining and open addressing



Rehashing for Chaining

```
20     /**
21     * Rehashing for separate chaining hash table.
22     */
23     void rehash( )
24     {
25         vector<list<HashedObj> > oldLists = theLists;
26
27         // Create new double-sized, empty table
28         theLists.resize( nextPrime( 2 * theLists.size( ) ) );
29         for( int j = 0; j < theLists.size( ); j++ )
30             theLists[ j ].clear( );
31
32         // Copy table over
33         currentSize = 0;
34         for( int i = 0; i < oldLists.size( ); i++ )
35         {
36             list<HashedObj>::iterator itr = oldLists[ i ].begin( );
37             while( itr != oldLists[ i ].end( ) )
38                 insert( *itr++ );
39         }
40     }
```

Rehashing for Quadratic Probing

```
1      /**
2      * Rehashing for quadratic probing hash table.
3      */
4      void rehash( )
5      {
6          vector<HashEntry> oldArray = array;
7
8          // Create new double-sized, empty table
9          array.resize( nextPrime( 2 * oldArray.size( ) ) );
10         for( int j = 0; j < array.size( ); j++ )
11             array[ j ].info = EMPTY;
12
13         // Copy table over
14         currentSize = 0;
15         for( int i = 0; i < oldArray.size( ); i++ )
16             if( oldArray[ i ].info == ACTIVE )
17                 insert( oldArray[ i ].element );
18     }
```



Hash Tables in C++ STL

- Hash tables not part of the C++ Standard Library
- Some implementations of STL have hash tables (e.g., SGI's STL)
 - `hash_set`
 - `hash_map`

Hash Set in SGI's STL

```
#include <hash_set>

struct eqstr
{
    bool operator()(const char* s1, const char* s2) const
    {
        return strcmp(s1, s2) == 0;
    }
};

void lookup(const hash_set<const char*, hash<const char*>, eqstr>& Set,
            const char* word)
{
    hash_set<const char*, hash<const char*>, eqstr>::const_iterator it
        = Set.find(word);
    cout << word << ": "
         << (it != Set.end() ? "present" : "not present")
         << endl;
}

int main()
{
    hash_set<const char*, hash<const char*>, eqstr> Set;
    Set.insert("kiwi");
    lookup(Set, "kiwi");
}
```

Key

Hash fn

Key equality test

Hash Map in SGI's STL

```
#include <hash_map>
```

```
struct eqstr
```

```
{
```

```
    bool operator() (const char* s1, const char* s2) const
```

```
    {
```

```
        return strcmp(s1, s2) == 0;
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    hash_map<const char*, int, hash<const char*>, eqstr> months;
```

```
    months["january"] = 31;
```

```
    months["february"] = 28;
```

```
    ...
```

```
    months["december"] = 31;
```

```
    cout << "january -> " << months["january"] << endl;
```

```
}
```

Key

Data

Hash fn

Key equality test



Problem with Large Tables

- What if hash table is too large to store in main memory?
- Solution: Store hash table on disk
 - Minimize disk accesses
- But...
 - Collisions require disk accesses
 - Rehashing requires a lot of disk accesses

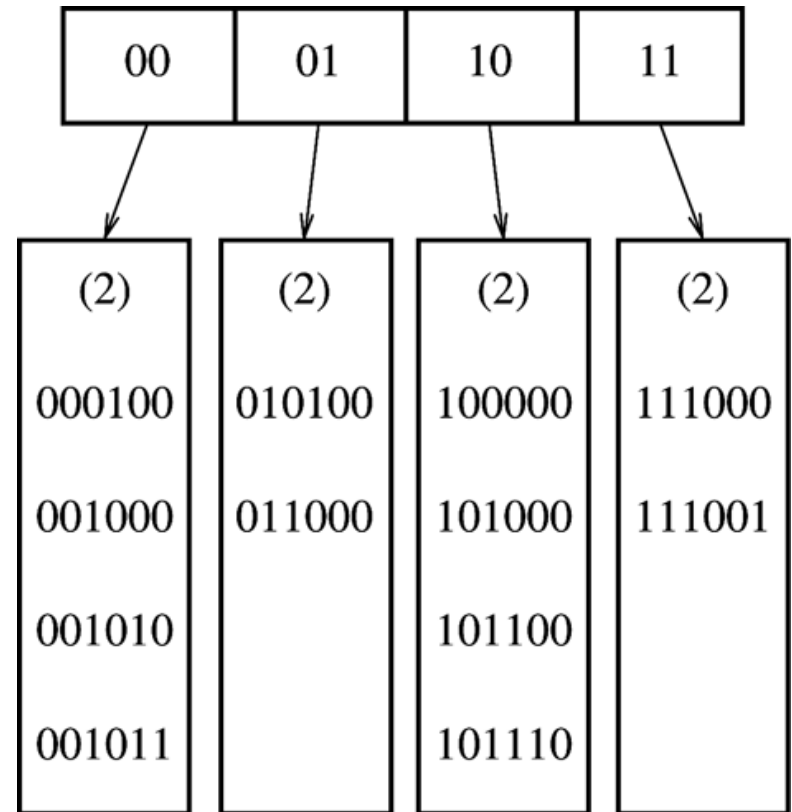


Extendible Hashing

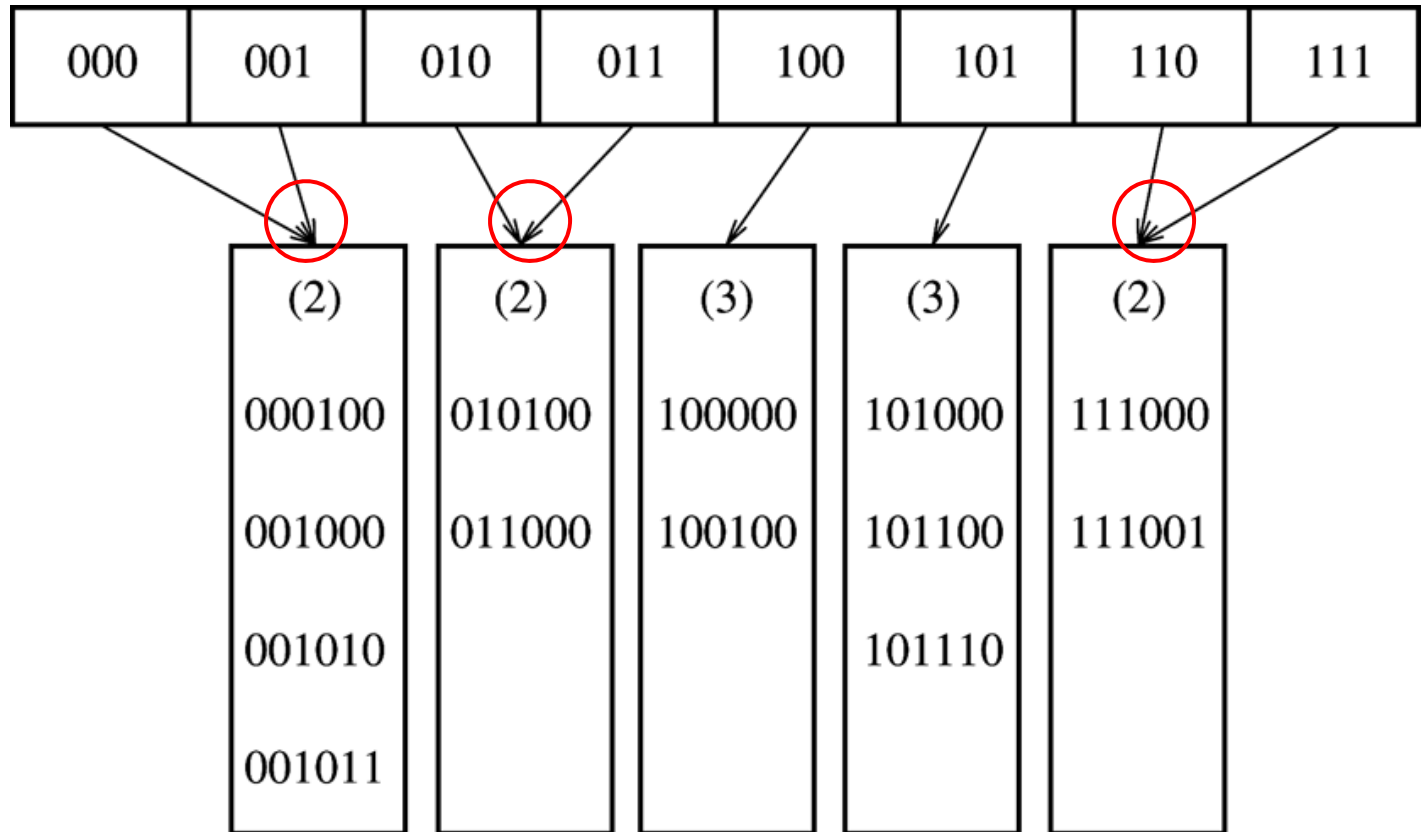
- Store hash table in a depth-1 tree
 - Every search takes 2 disk accesses
 - Insertions require few disk accesses
- Hash the keys to a long integer (“extendible”)
- Use first few bits of extended keys as the keys in the root node (“directory”)
- Leaf nodes contain all extended keys starting with the bits in the associated root node key

Extendible Hashing Example

- Extendible hash table
- Contains $N = 12$ data elements
- First $D = 2$ bits of key used by root node keys
 - 2^D entries in directory
- Each leaf contains up to $M = 4$ data elements
 - As determined by disk page size
- Each leaf stores number of common starting bits (d_L)



Extendible Hashing Example

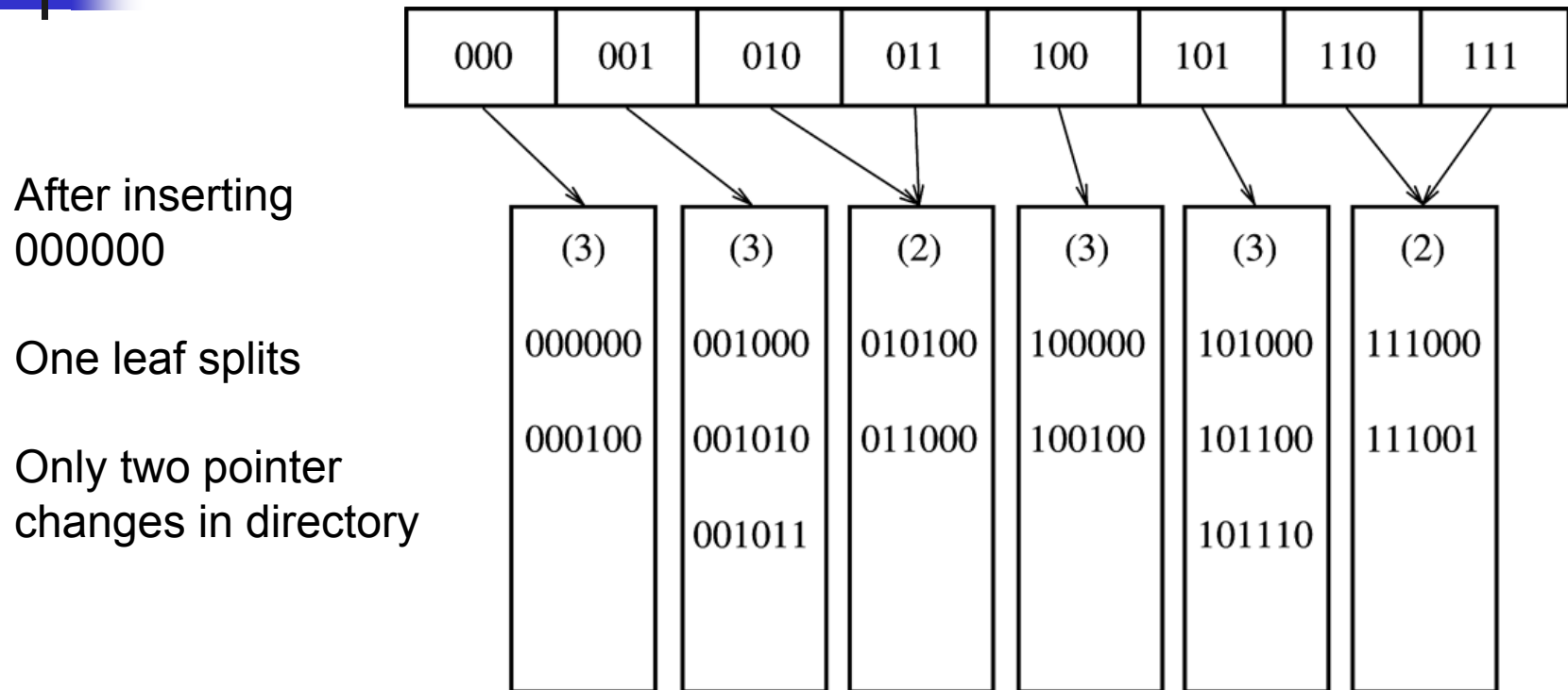


After inserting
100100

Directory split
and rewritten

○ Leaves not involved in split now pointed to by two adjacent directory entries. These leaves are not accessed.

Extendible Hashing Example





Extendible Hashing Analysis

- Expected number of leaves is $(N/M) * \log_2 e = (N/M) * 1.44$
- Average leaf is $(\ln 2) = 0.69$ full
 - Same as for B-trees
- Expected size of directory is $O(N^{(1+1/M)}/M)$
 - $O(N/M)$ for large M (elements per leaf)



Hash Table Applications

- Maintaining symbol table in compilers
- Accessing tree or graph nodes by name
 - E.g., city names in Google maps
- Maintaining a transposition table in games
 - Remember previous game situations and the move taken (avoid re-computation)
- Dictionary lookups
 - Spelling checkers
 - Natural language understanding (word sense)



Summary

- Hash tables support fast insert and search
 - $O(1)$ average case performance
 - Deletion possible, but degrades performance
- Not good if need to maintain ordering over elements
- Many applications