CPTS 223 Advanced Data Structure C/C++

Hashing

Overview

Average-case O(1)

- Hash Table Data Structure : Purpose
  - To support insertion, deletion and search in average-case constant time
  - Assumption: Order of elements irrelevant
  - → data structure **NOT** useful for if you want to maintain and retrieve an order of the elements
- Hash function
  - Hash["string key"] → integer value
- Hash table ADT
  - Implementations, analysis, applications
- Collision in hash tables

Hash table

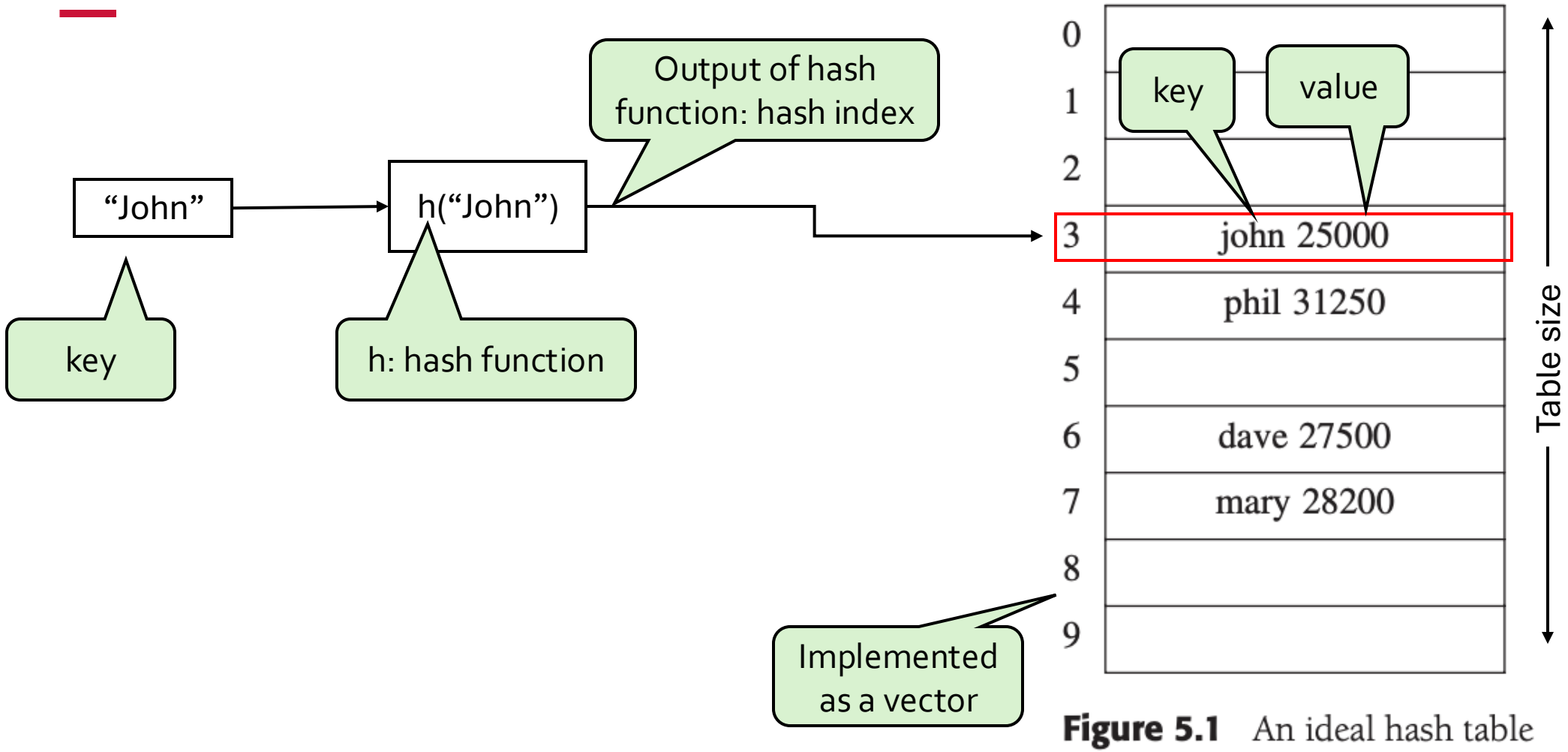


Figure 5.1 An ideal hash table

Hash table

- 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("john") = 3

Output of hash function: hash index

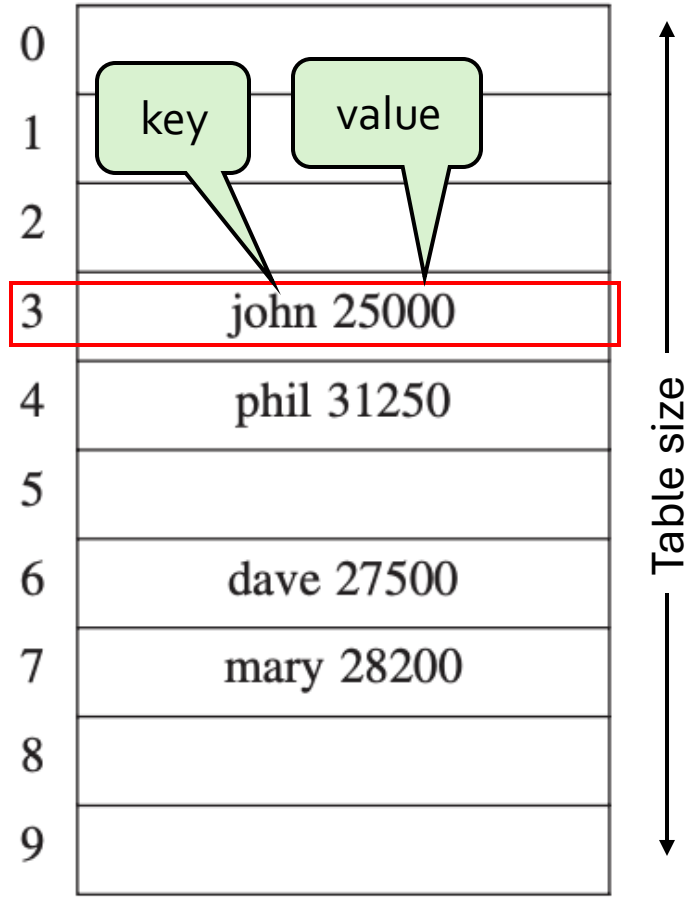


Figure 5.1 An ideal hash table

Hash table operations

hash function

hash key

Data record

- Insert
  - T [h("john")] = <"john",25000>
- Delete
  - T [h("john")] = NULL
- Search
  - T [h("john")] returns the element hashed for "john"

What happens if h("john") == h("joe")?

⇨ collision

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

Figure 5.1 An ideal hash table

Factors of hash table design

- Hash function
- Table size
  - Usually fixed at the start
- Collision handling scheme

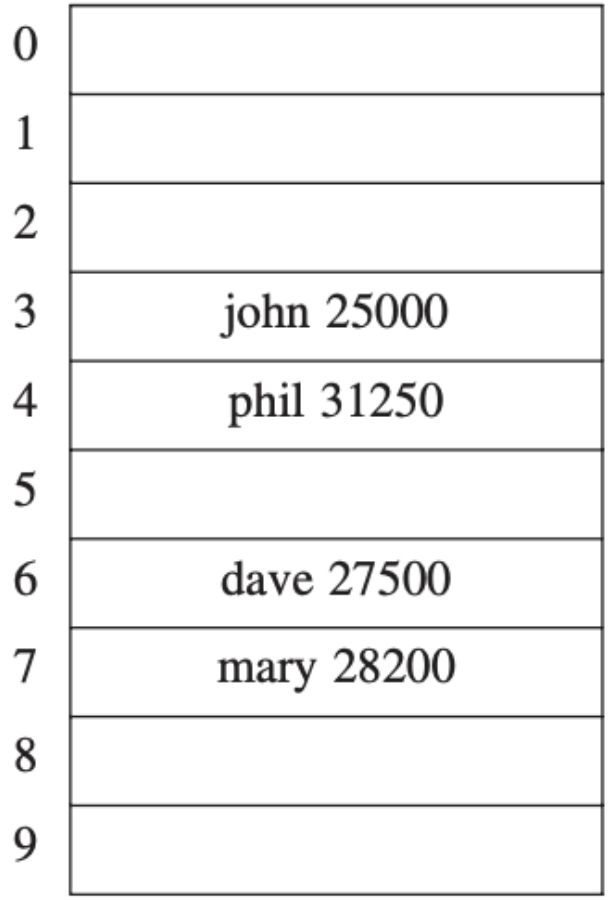


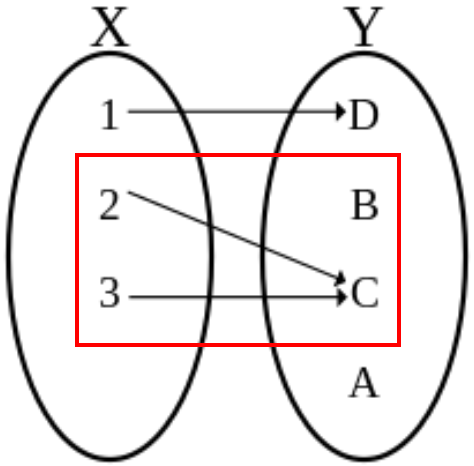
Figure 5.1 An ideal hash table

Hash function

- A hash function is one which maps an element's key into a valid hash table index
  - h(key) → hash table index
- Note that this is different from saying:
  - h(string) → int
  - Because the key can be of any type
  - e.g., "h(int) => int" is also a hash function

Hash function properties

h(key) → hash able index



- A hash function maps key to integer
  - Constraint: Integer should be between:

    [0, TableSize-1]
- A hash function can result in a many-to-one mapping (causing collision)
  - Collision occurs when hash function maps two or more keys to same array index
- Collisions cannot be avoided
- But its chances can be reduced using a "good" hash function

Hash function properties

- A "good" hash function should have the following properties:
  - Reduced chance of collision
    - Different keys should ideally map to different indices
    - Distribute keys uniformly over table
  - Should be fast to compute

Effective table size

- Simple hash function (assume integer keys)
  - h(Key) = Key mod TableSize
- For random keys, h() distributes keys evenly over table
  - What if TableSize = 100 and keys are ALL multiples of 10?
  - Better if TableSize is a prime number

Hash function for string keys

- Approach 1: A very simple function to map strings to integers
  - Add up character ASCII values (0-255) to produce integer keys
    - e.g., "abcd" = 97+98+99+100 = 394
    - → h("abcd") = 394 % TableSize
- Potential problems:
  - Anagrams will map to the same index
    - h("abcd") == h("dbac")
  - Small strings may not use all of table
    - Strlen(S) * 255 << TableSize
  - Time proportional to length of the string

Hash function for string keys

- Approach 2: treat first 3 characters of string as base-27 integer (26 letters plus space)
  - Key = S[0] + (27 * S[1]) + (272 * S[2])
- Potential problems:
  - Assumes first 3 characters randomly distributed?
    - Not true of English
      - Apple
      - Apply
      - Appointment
      - Apricot

collision

Better than approach 1 because … ?

e.g., anagrams will not map to the same index h("abcd") ≠ h("dbac")

Hash function for string keys

- Approach 3: use all N characters of string as an N-digit base-K number
- 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) = $[\sum_{i=0}^{L-1} S(L-1-i) * 37^i]$ mod TableSize
- Use Horner's rule to compute h(S)
- Limit L for long strings

```
1   /**
2    * A hash routine for string objects.
3    */
4   unsigned int hash( const string & key, int tableSize )
5   {
6       unsigned int hashVal = 0;
7
8       for( char ch : key )
9           hashVal = 37 * hashVal + ch;
10
11      return hashVal % tableSize;
12  }
```

Figure 5.4 A good hash function

Polynomial function of 37
Degree = L-1
Problems: potential overflow larger runtime

Techniques for collisions

- Separate chaining
- Probing
- Double hashing

With linked lists

Open addressing: without linked lists

Techniques for collisions

- What happens when h(k1) = h(k2)?
  - → collision
- Collision resolution strategies
  - Separate chaining
    - Store colliding keys in a linked list at the same hash table index
  - Open addressing
    - Store colliding keys elsewhere in the table
- Rehashing: when the hash table is too full
  - If too full
    - hash table operations take too long
    - insertions might fail for open addressing

Separate chaining

Insertion sequence: { 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 }

- Strategy: maintains a linked list at every hash index for collided elements
- Hash table T is a vector of linked lists
  - Insert element at the head (as shown here) or at the tail
- Key k is stored in list at T[h(k)]
- e.g., TableSize = 10
  - h(k) = k mod 10
  - Insert first 10 perfect squares

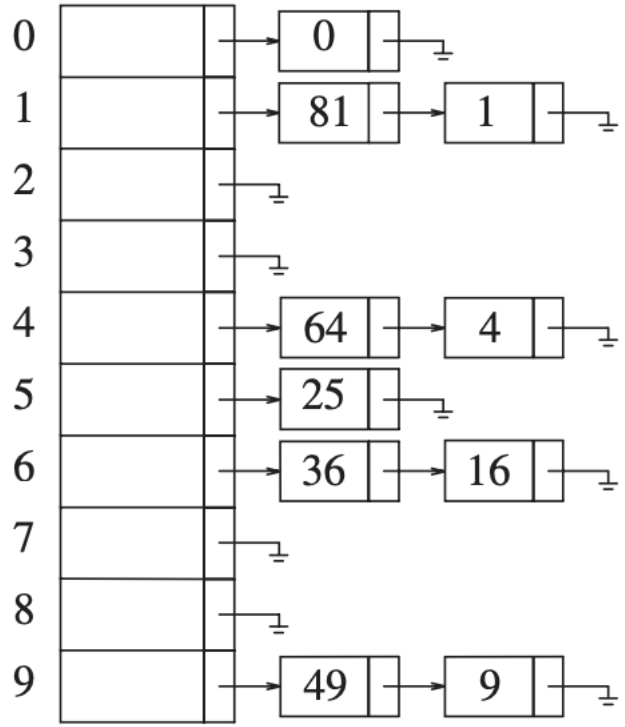


Figure 5.5 A separate chaining hash table

Separate chaining: declaration

```
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 insert( HashedObj && x );
12      bool remove( const HashedObj & x );
13
14    private:
15      vector<list<HashedObj>> theLists;     // The array of Lists
16      int currentSize;
17
18      void rehash( );
19      size_t myhash( const HashedObj & x ) const;
20  };
```

Vector of linked lists (the main hash table)

size_t: unsigned integral type that represents the size of an object

Current #elements in the hash table

Figure 5.6 Type declaration for separate chaining hash table

Separate chaining: hash function

Generic hash function

```
template <>
class hash<string>
{
  public:
    size_t operator()( const string & key )
    {
        size_t hashVal = 0;

        for( char ch : key )
            hashVal = 37 * hashVal + ch;

        return hashVal;
    }
};
```

Private myHash function

```
1       size_t myhash( const HashedObj & x ) const
2       {
3           static hash<HashedObj> hf;
4           return hf( x ) % theLists.size( );
5       }
```

Figure 5.7 myHash member function for hash tables

Separate chaining: insert

```
1       bool insert( const HashedObj & x )
2       {
3           auto & whichList = theLists[ myhash( x ) ];
4           if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )
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      }
```

Find the target linked list

Hash function outputs the index of table

find returns:
the target iterator (if match) or last iterator (if not match)

Add data at end

Check if already present:
If present: do nothing

Rehash: double the size of the hash table if it gets crowded

Figure 5.10 `insert` routine for separate chaining hash table

Separate chaining

```
1     void makeEmpty( )
2     {
3         for( auto & thisList : theLists )
4             thisList.clear( );
5     }
6
7     bool contains( const HashedObj & x ) const
8     {
9         auto & whichList = theLists[ myhash( x ) ];
10        return find( begin( whichList ), end( whichList ), x ) != end( whichList );
11    }
```

Hash function outputs the index of table

Check if already present:
If present: do nothing

Separate chaining: remove

```
13      bool remove( const HashedObj & x )
14      {
15          auto & whichList = theLists[ myhash( x ) ];
16          auto itr = find( begin( whichList ), end( whichList ), x );
17
18          if( itr == end( whichList ) )
19              return false;
20
21          whichList.erase( itr );
22          --currentSize;
23          return true;
24      }
```

Not found, no removal

Each of these operations takes time linear in the length of the list at the hashed index location

Figure 5.9 makeEmpty, contains, and remove routines for separate chaining hash table

Separate chaining: analysis

- Load factor λ of a hash table T is defined as follows:
  - N = number of elements in T
    - Current size
  - M = size of T
    - TableSize
  - λ = N/M
    - load factor
  - i.e., λ is the average length of a chain
- Search time: O(1 + λ)
- Ideally, want λ ≤ 1

Separate chaining: disadvantage

- Linked lists could get long
  - Especially when N approaches M
  - Longer linked lists could negatively impact performance
- More memory because of pointers
- Absolute worst-case (even if N << M):
  - All N elements in one linked list!
  - Typically the result of a bad hash function

Open addressing

- When a collision occurs, look elsewhere in the table for an empty slot
- Advantages over chaining
  - No need for 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 $\lambda \approx 0.5$

Load factor = N/M

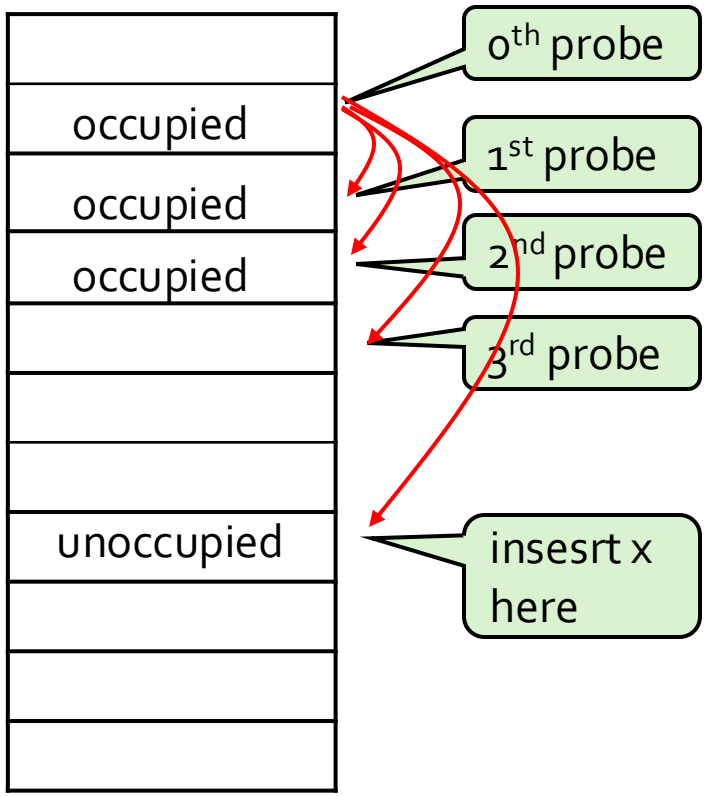
N: # of elements in hash table
M: size of T (TableSize)

Open addressing

- A "probe sequence" is a sequence of slots in hash table while searching for an element x
  - h_0(x), h_1(x), h_2(x), …
  - Needs to visit each slot exactly once
  - Needs to be repeatable (so we can find/delete what we have inserted already)
- Hash function

  f(i) can be linear, quadratic, etc.
  - h_i(x) = (h(x) + f(i)) mod TableSize
  - f(0) = 0 → position for the 0-th probe
  - f(i) is "the distance to be traveled relative to the 0-th probe position, during the i-th probe".

Open addressing: linear probing



- f(i): a linear function of i
- e.g., f(i) = i
- h_i(x) = (h(x) + f(i)) mod TableSize
  = (h(x) + i) mod TableSize
- Probe sequence: +0, +1, +2, …
- Continue until an empty slot is found
- #failed probes is a measure of performance (smaller → better)

Open addressing: linear probing

- f(i): a linear function of i, e.g., f(i)=i
  - hi(x) := (h(x) + i) mod TableSize
  - Probe sequence: +0, +1, +2, +3, +4, …
- Example: h(x) = x mod TableSize
  - ho(89) = (h(89)+f(0)) mod 10 = 9
  - ho(18) = (h(18)+f(0)) mod 10 = 8
  - ho(49) = (h(49)+f(0)) mod 10 = 9 (X)
  - h1(49) = (h(49)+f(1)) mod 10
    = (h(49)+ 1) mod 10 = 0

Hash table at index 9 has been occupied by 89 (the first insert)

Open addressing: linear probing

Insert sequence:
89, 18, 49, 58, 69

	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
#unsuccessful probes:		0	0	1	3	3

Totally 7

Figure 5.11 Hash table with linear probing, after each insertion

Linear probing: issues

- Probe sequences can get longer with time
- Primary clustering (blocks of occupied cells)
  - If probe occurs, 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)
  - Side effect: Other keys could also get affected if mapping to a crowded neighborhood

Linear probing: analysis

- Expected number of probes for insertion or unsuccessful search

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

Target element is not present in hash table

- Expected number of probes for successful search

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

Target element is present in hash table

- 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

Too many probes
Can we avoid?

Proof by Don Knuth in 1963, notes at https://jeffe.cs.illinois.edu/teaching/datastructures/2011/notes/knuth-OALP.pdf

Random probing

- Instead of using f(i)=i
- h_i(x) = (h(x) + probe_positions[i]) mod TableSize
  - probe_positions is an array that contains a list of random numbers
- The array should be generated beforehand and used in both search and insert
- Avoids primary clustering

Why?

Random probing

- Instead of using f(i)
- h_i(x) = (h(x) + probe_positions[i]) mod TableSize
  - probe_positions is an array that contains a list of random numbers
- The array should be generated beforehand and used in both search and insert
- Avoids primary clustering

Why?

Recall: probe sequence needs to be repeatable (so we can find/delete what we have inserted already)

Random probing

- 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
  - $\lambda = 0.5 \rightarrow 1.4$ probes
  - $\lambda = 0.9 \rightarrow 2.6$ probes

Random probing

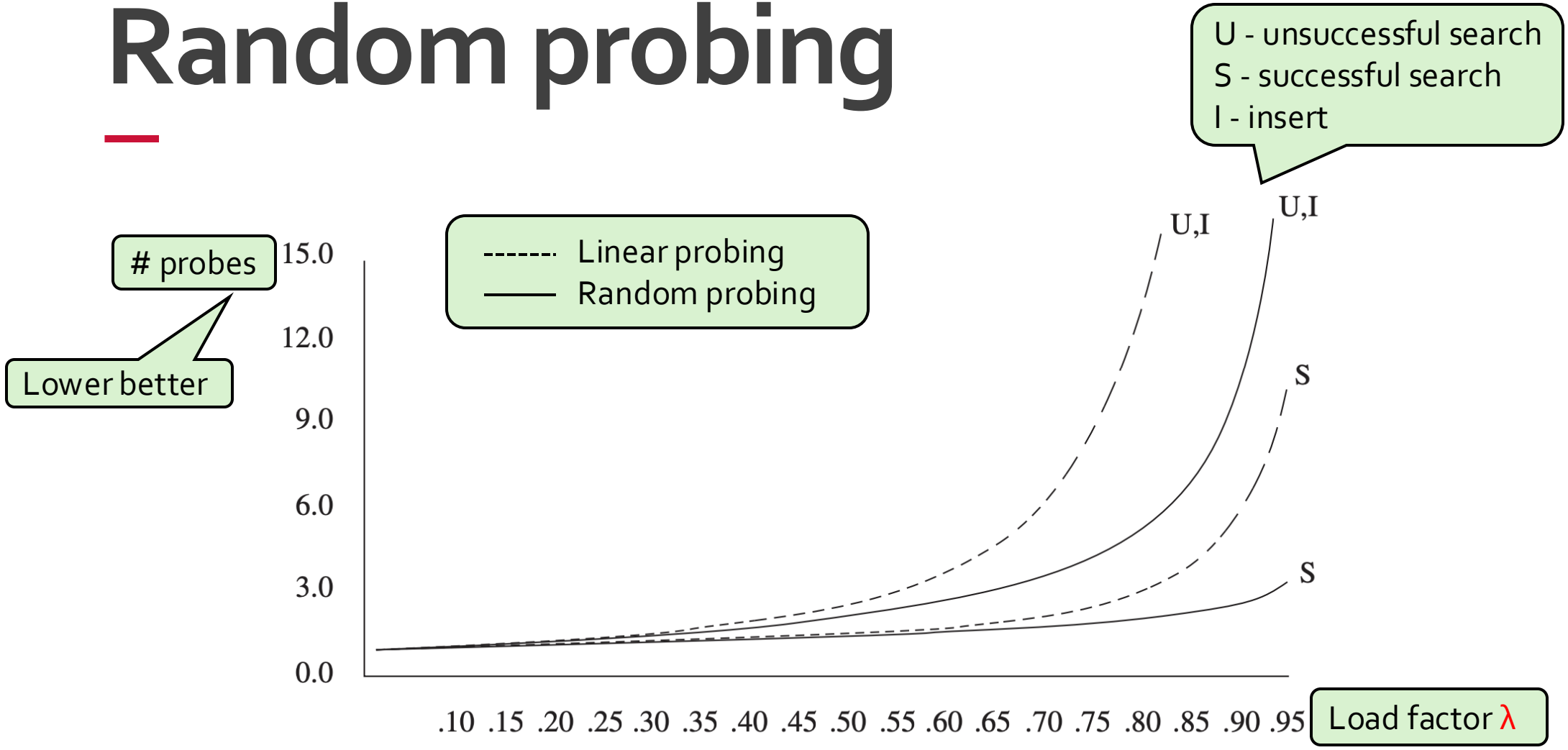
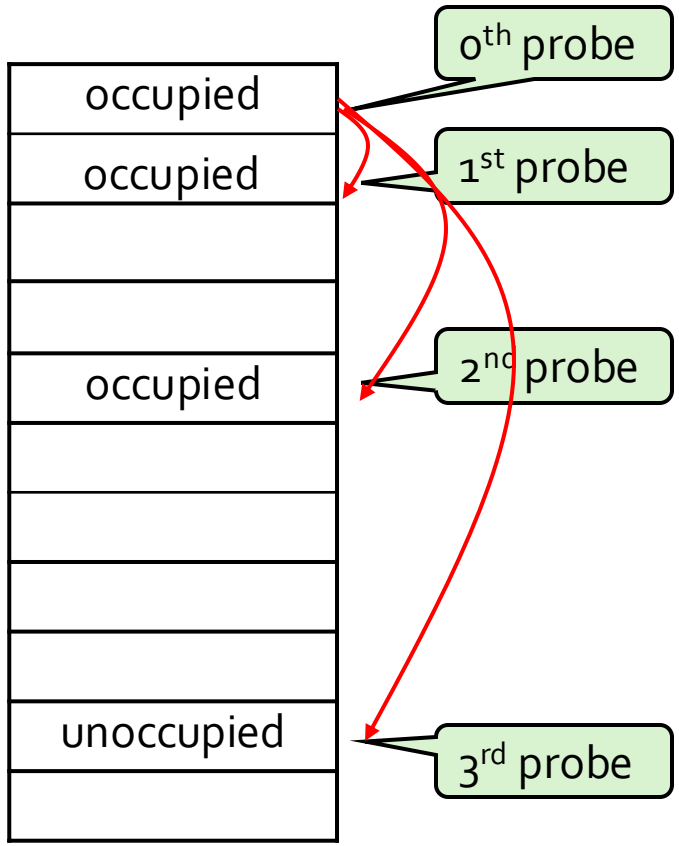


Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (*S* is successful search, *U* is unsuccessful search, and *I* is insertion)

Quadratic probing



Why?

- Avoids primary clustering
- f(i) is quadratic in i
  - e.g., f(i) = i^2
  - h_i(x) = ((h(x) +i^2) mode TableSize
- Probe sequence:
  - +0, +1, +4, +9, +16, …
- Continue until an empty slot is found
- #failed probes is a measure of performance

Quadratic probing

- Avoids primary clustering
- f(i) is quadratic in i, e.g., f(i) = i
  - h_i(x) = (h(x) + i^2) mod TableSize
  - Probe sequence: +0, +1, +4, +9, +16, …
- Example: insert 89, 18, 49, 58

h0(89) = (h(89)+f(0)) mod 10 = 9
h0(18) = (h(18)+f(0)) mod 10 = 8
h0(49) = (h(49)+f(0)) mod 10 = 9 (X)
h1(49) = (h(49)+f(1)) mod 10 = 0

9 has been occupied

h0(58) = (h(58)+f(0)) mod 10 = 8 (X)
h1(58) = (h(58)+f(1)) mod 10 = 9 (X)
h2(58) = (h(58)+f(2)) mod 10 = 2

8 has been occupied

9 has been occupied

Quadratic probing

	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
#unsuccessful probes:		0	0	1	2	2

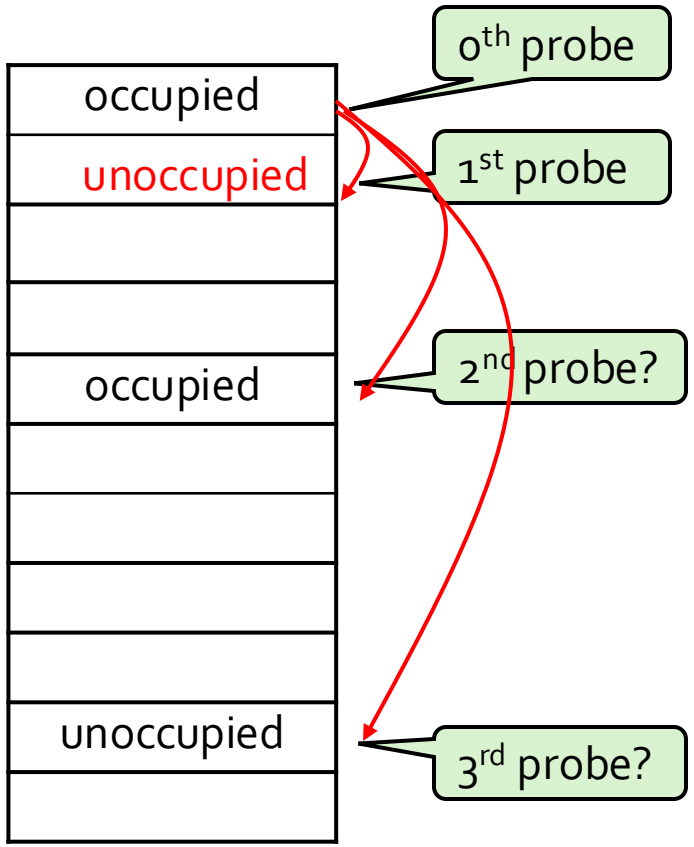
Totally 5

Figure 5.13 Hash table with quadratic probing, after each insertion

Quadratic probing: analysis

- 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 is one exists
- Ensure table never gets half full
  - If close, then expand it (rehash)
- May cause "secondary clustering":
  - elements that hash to the same position will probe the same alternative cells
  - Take long time to find empty slots

Quadratic probing: delete



- Deletion
  - Emptying slots can break probe sequence and could cause find stop prematurely
  - Lazy deletion
    - Differentiate between empty and deleted slot
    - When finding, skip and continue beyond deleted slots
    - If you hit a non-deleted empty slot, then stop find procedure returning "not found"
- When does it really (physically) delete the data?
  - Delete upon insert
  - Compaction at some time

Quadratic probing

```
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 insert( HashedObj && x );
12      bool remove( const HashedObj & x );
13
14      enum EntryType { ACTIVE, EMPTY, DELETED };
```

For lazy deletion

Figure 5.14

Qu

```
16      private:
17        struct HashEntry
18        {
19            HashedObj element;
20            EntryType info;
21
22            HashEntry( const HashedObj & e = HashedObj{ }, EntryType i = EMPTY )
23              : element{ e }, info{ i } { }
24            HashEntry( HashedObj && e, EntryType i = EMPTY )
25              : element{ std::move( e ) }, info{ i } { }
26        };
27
28        vector<HashEntry> array;
29        int currentSize;
30
31        bool isActive( int currentPos ) const;
32        int findPos( const HashedObj & x ) const;
33        void rehash( );
34        size_t myhash( const HashedObj & x ) const;
35    };
```

Initialize by EMPTY

Figure 5.14 Class interface for hash tables using probing strategies, including the nested `HashEntry` class

Quadratic probing

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

Clear hash table: set to EMPTY

Figure 5.15 Routines to initialize quadratic probing hash table

Qu

```
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; }
```

Quadratic probing position

Figure 5.16 `contains` routine (and `private` helpers) for hashing with quadratic probing

Qu

```
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; }
```

Sum of i odd numbers == i^2

currentPos: sum of odd numbers

offset: a sequence of odd numbers: 1, 3, 5, ...

Skip DELETED

Figure 5.16 `contains` routine (and `private` helpers) for hashing with quadratic probing

Qua


```
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 ].element = x;
9           array[ currentPos ].info = ACTIVE;
10
11          // Rehash; see Section 5.5
12          if( ++currentSize > array.size( ) / 2 )
13              rehash( );
14
15          return true;
16      }
17
18      bool remove( const HashedObj & x )
19      {
20          int currentPos = findPos( x );
21          if( !isActive( currentPos ) )
22              return false;
23
24          array[ currentPos ].info = DELETED;
25          return true;
26      }
```

Find quadratic probing position

Labeled as ACTIVE

Find quadratic probing position

Labeled as DELETED

Figure 5.17 Some `insert` and `remove` routines for hash tables with quadratic probing

Double hashing

- Use a second hash function
- Good choices for h_2(x) ?
  - Should never evaluate to 0
  - e.g., h_2(x) = R – (x mod R)
  - R is prime number less than TableSize
- Previous example with R=7
  - h_2(x) = 7 – (x mod 7)
  - f(i) = i * h_2(x)
  - h0(49) = (h(49)+f(0)) mod 10 = 9 (X)
  - h1(49) = (h(49)+1*(7 – 49 mod 7)) mod 10 = 6

h_i(x) = (h(x) + f(i))

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
#unsuccessful probes:		0	0	1	1	2

collision 2

collision

collision

collision 1

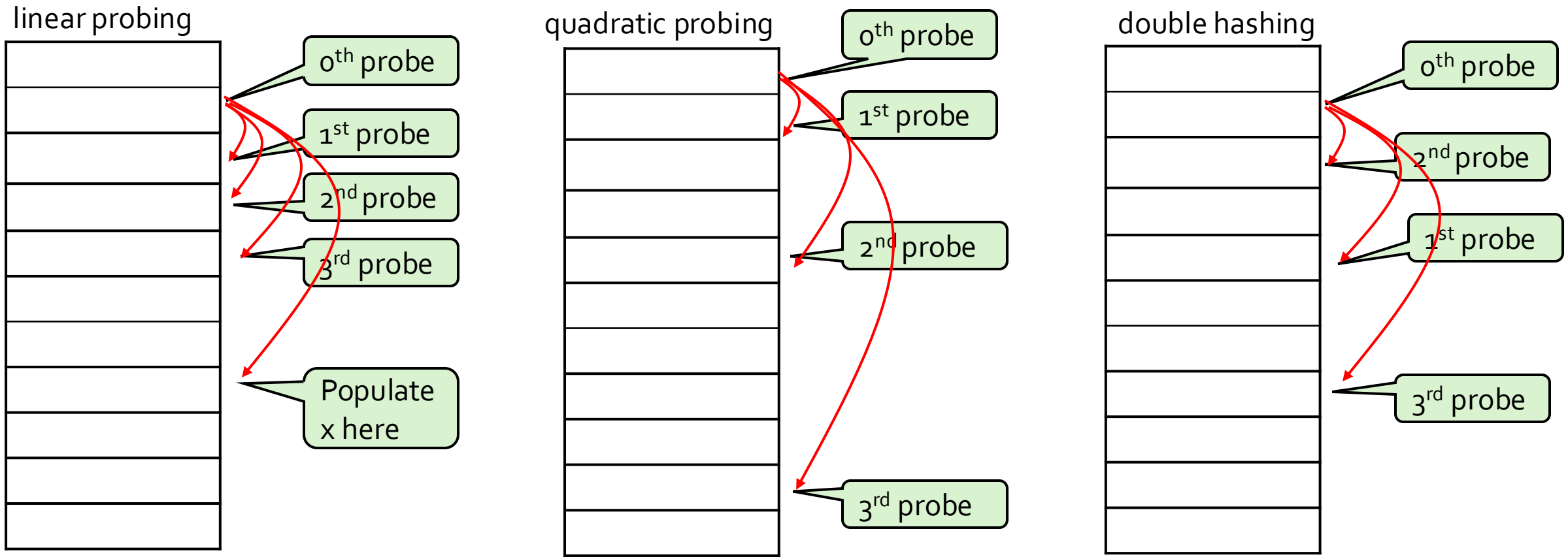
Totally 4

Figure 5.18 Hash table with double hashing, after each insertion

Double hashing: analysis

- TableSize must be prime
- Empirical tests show double hashing close to random probing
- Extra hash function takes extra time to compute

Probing techniques: review



Rehashing

- Increases the size of the hash table when load factor becomes "too high" (defined by a cutoff)
  - Anticipating that prob(collisions) would become higher
- Typically expand the table to twice its size (but still prime)
- Need to reinsert all existing elements into new hash table

Rehashing: example

h(x) = x mod 7
λ = 0.57

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

Figure 5.20 Hash table with

Insert 23

λ = 0.71

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

Figure 5.20 Hash table with

rehashing

h(x) = x mod 17
λ = 0.29

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

Figure 5.21 Hash table after rehashing

Rehashing: analysis

- Rehashing takes time to do N insertions
  - O(N)
- Therefore, should do it 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 load factor reaches some threshold (e.g,. $\lambda \geq 0.5$), OR
  - When an insertion fails
- Applies across collision handling schemes

Rehashing for separate 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( auto & thisList : theLists )
30              thisList.clear( );
31
32              // Copy table over
33          currentSize = 0;
34          for( auto & thisList : oldLists )
35              for( auto & x : thisList )
36                  insert( std::move( x ) );
37      }
```

Double the size

Empty each list

Each list

Each element

Figure 5.22 Rehashing for both separate chaining hash tables and pr

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( auto & entry : array )
11          entry.info = EMPTY;
12
13          // Copy table over
14      currentSize = 0;
15      for( auto & entry : oldArray )
16          if( entry.info == ACTIVE )
17              insert( std::move( entry.element ) );
18      }
```

Double the size

Empty each element

Each element

Hash tables in C++ STL

- Some implementations of STL have hash tables (e.g., SGI STL)
  - hash_set
  - hash_map

STL unordered_map

- unordered_map: A better option for hash table
- Supported in C++ 11 and later
- Supports a standard interface for common hash table operations
- Operations on average are O(1)

Problems 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

Solution:
extendible hashing

Hash table applications

- 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)
- Heavily used in text processing languages
  - e.g., Python dictionary

Hashing: summary

- Hash tables support fast insert and search
  - O(1) insert, search, delete avg performance
  - Deletion possible, but degrades performance
- Not suited if ordering of elements is important
- Many applications

Hashing: checklist to remember

- Table size prime
- Table size much larger than number of inputs (to maintain λ closer to 0 or < 0.5)
- Tradeoffs between chaining vs. probing
- Collision chances decrease in this order:

linear probing > quadratic probing > {random probing, double hashing}

- Rehashing required to resize hash table at a time when λ exceeds a threshold (usually 0.5)
- Good for searching. Not good if there is some order implied by data