Data Structures and Algorithms

Hash Functions and Hash Tables

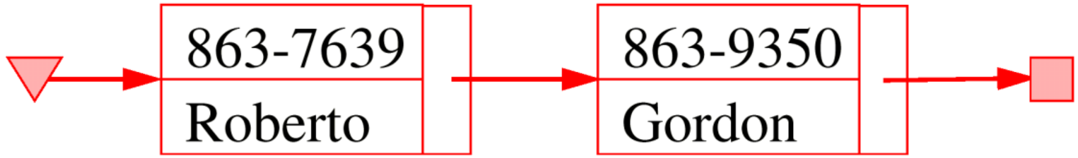
Motivation

- Your telephone company wants to provide caller ID capability:
  - given a phone number, return the caller's name
  - phone numbers are in the range R=0 to $10^7 - 1$
  - want to do this as efficiently as possible ($$$)
- A few suboptimal ways to design this dictionary:
  1. an array indexed by key: takes O(1) time, O(N+R) space
     - huge amount of wasted space

(null)	(null)	...	Roberto	...	(null)
000-0000	000-0001	...	863-7639	...	999-9999

Motivation

2. a linked list: takes O(N) time, O(N) space



3. a balanced binary tree: O(log N) time, O(N) space
    - very slow for some applications

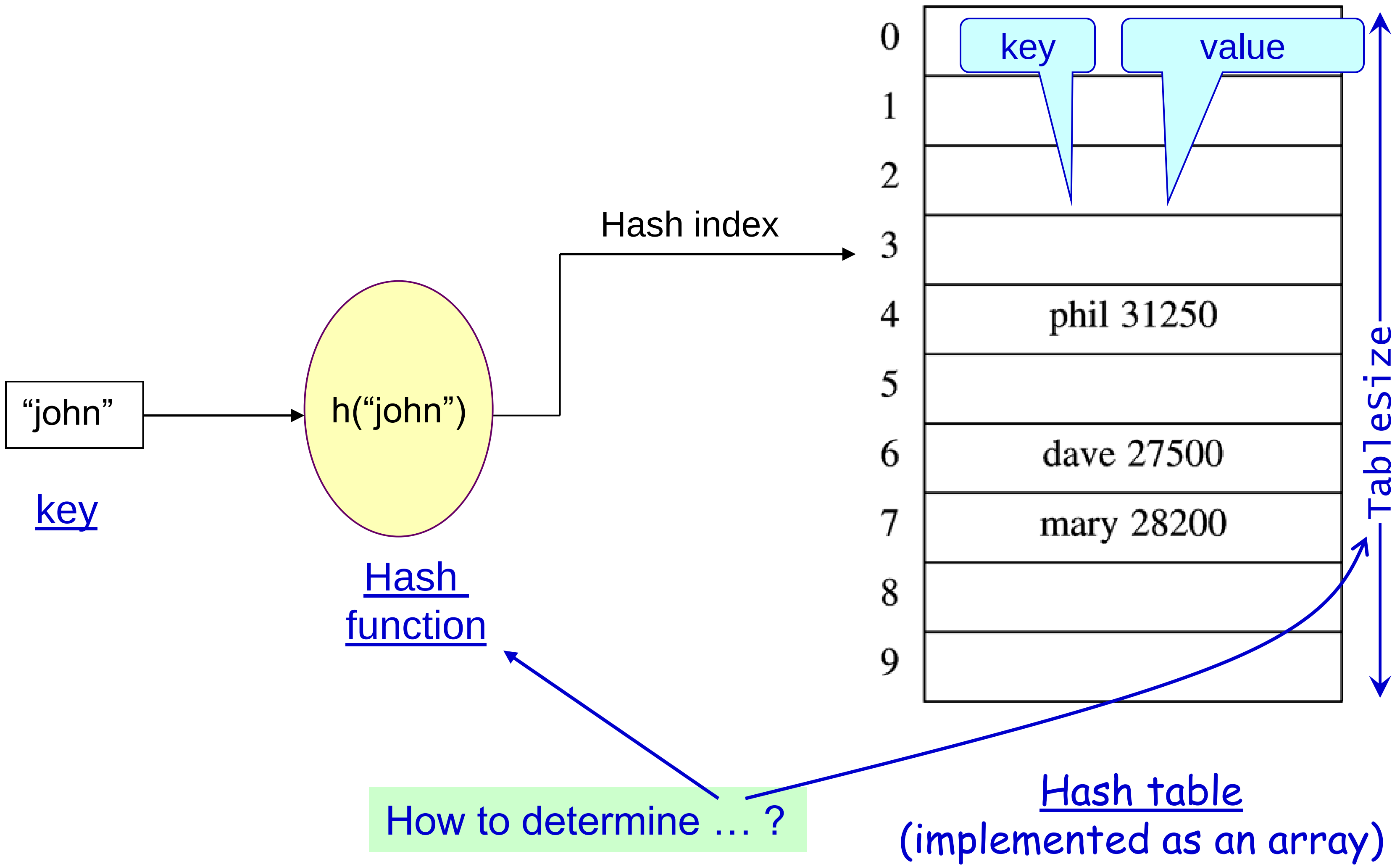
- We can do better, with a Hashtable:
    - O(1) expected time
    - O(N+M) space, where M is table size
- Like an array, but come up with a function to map the large range into one which we can manage
    - e.g., take the original key, modulo the (relatively small) size of the array, and use that as an index

Overview

Hash["string key"] ==> integer value

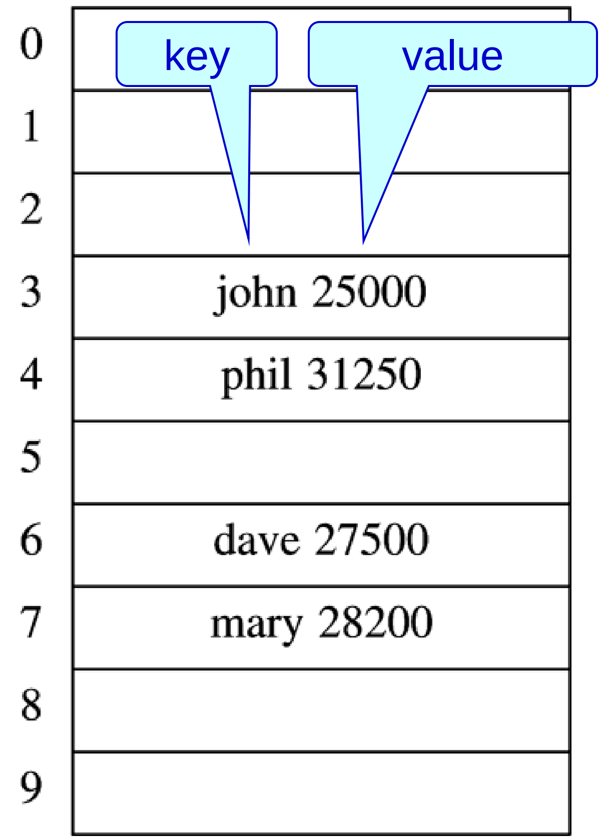
- Hash Table Data Structure
  - To support insertion, deletion and search in *average-case constant time*
  - Assumption: Order of elements irrelevant
    - Accordingly, this data structure is not useful if you want to maintain and retrieve some kind of an order of the elements.
    - In particular, operations such as `findMin`, `findMax`, or "print all elements in order" are not supported.
    - For such operations, use a BST (or an AVL)
- Hash table ADT
  - Implementations
  - Analysis
  - Applications

Hash table: Main components



Hash Table

- Hash table is an array of fixed size `TableSize`
- Array elements indexed by a <u>key</u>, which is mapped to an array index (0 … `TableSize-1`)
- Mapping (hash function) h from key to index
  - E.g., `h("john") = 3`



Hash Table

Hash function

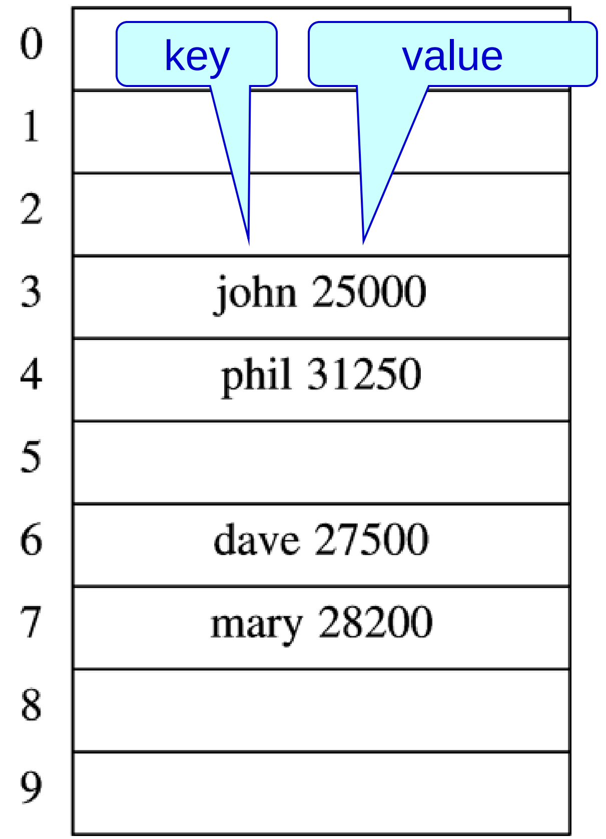
Hash key

- **Insert**
  - T [h("john")] = <"john", 25000>

Data record

- **Delete**
  - T [h("john")] = NULL
- **Search**
  - T [h("john")] returns the element hashed for "john"
- What happens if

  h("john") = h("joe") ?
  - "collision"



Factors Affecting Hash Table Design

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

Hash Function Properties

h(key) ==> hash table 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

h(key) ==> hash table index

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

Hash Function - Effective use of 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

Designing a Hash Function for String Keys

- Different Ways to Design a Hash Function for String Keys
- 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 *(see next slide)*
- Time proportional to length of the string

Another hash function example

- An example of a hash function that may not yield equitable distribution of keys when the table size is large and the string is short

```
public static int hash ( String key, int tableSize)
{
   int hashVal = 0;

   for ( int i = 0; i < key.length(); i++ )
      hashVal += key.charAt( i );

   return hashVal % tableSize;
}
```

Designing a Hash Function for String Keys

- Another Approach

  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

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

  use Horner's rule to compute h(S)
- limit L for long strings

```
/**
 * A hash routine for String objects.
 * @param key the String to hash.
 * @param tableSize the size of the hash table.
 * @return the hash value.
 */
 public static int hash( String key, int tableSize )
{
   int hashVal = 0;

   for( int i = 0; i < key.length( ); i++ )
      hashVal = 37 * hashVal + key.charAt( i );

   hashVal %= tableSize;
   if( hashVal < 0 )
      hashVal += tableSize;

   return hashVal;
}
```

Recall that modulus returns the remainder of dividing its first argument by the second (i.e., its modulus).

For negative values, the result may vary depending on the library implementation.

Resolving Collisions

- What happens when $h(k_1) = h(k_2)$?

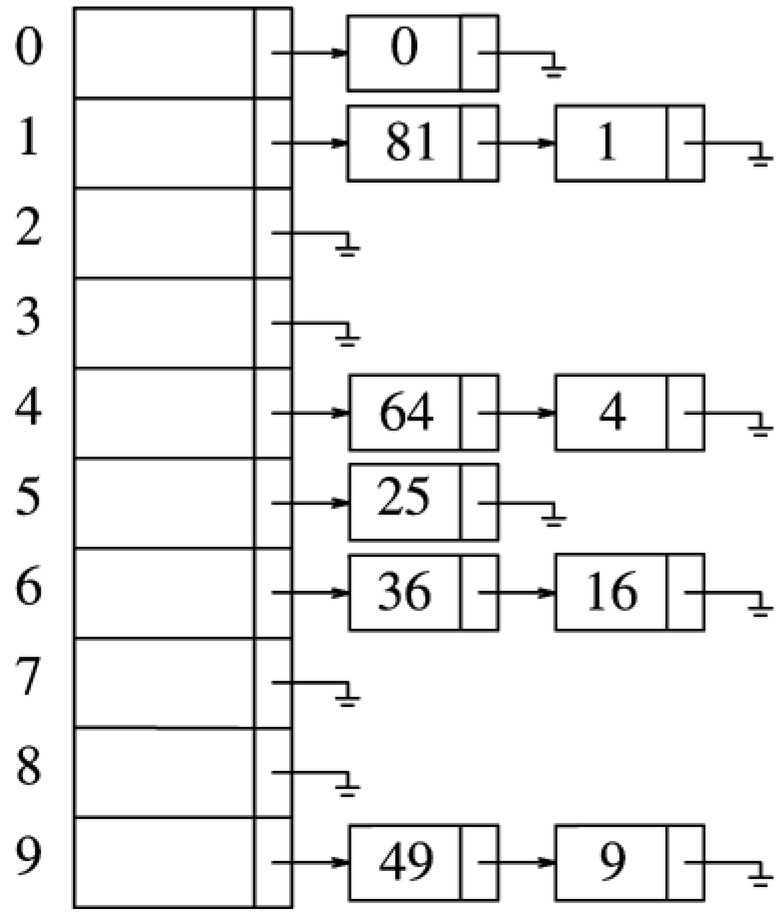
    ==> collision !

- Collision resolution strategies
    - Chaining
        - Store colliding keys in a linked list at the same hash table index
    - Open addressing
        - Store colliding keys elsewhere in the table
- Double hashing

Chaining

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

- 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



Implementation of Chaining Hash Table

```
import java.util.LinkedList;
import java.util.List;

// SeparateChaining Hash table class
//
// CONSTRUCTION: an approximate initial size or default of 101
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// boolean contains( x )  --> Return true if x is present
// void makeEmpty( )      --> Remove all items

/**
 * Separate chaining table implementation of hash tables.
 * Note that all "matching" is based on the equals method.
 * @author Mark Allen Weiss
 */
```

Implementation of Chaining Hash Table

```
public class SeparateChainingHashTable<AnyType>
{
    private static final int DEFAULT_TABLE_SIZE = 101;

    /** The array of Lists. */
    private List<AnyType> [ ] theLists;

    private int currentSize;

    /**
     * Construct the hash table.
     */
    public SeparateChainingHashTable( )
    {
        this( DEFAULT_TABLE_SIZE );
    }
```

Array of linked lists
(this is the main hashtable)

Current #elements in
the hashtable

Implementation of Chaining Hash Table

```
/**
 * Construct the hash table.
 * @param size approximate table size.
 */
@SuppressWarnings("unchecked")
public SeparateChainingHashTable( int  size )
{
    theLists = new LinkedList[ nextPrime( size ) ];

    for( int i = 0; i < theLists.length; i++ )
        theLists[ i ] = new LinkedList<AnyType>( );
}
```

Sometimes Java Generics doesn't let you do what you want to, and you need to effectively tell the compiler that what you're doing really will be legal at execution time.

Implementation of Chaining Hash Table

```java
/**
 * Insert into the hash table. If the item is
 * already present, then do nothing.
 * @param x the item to insert.
 */
public void insert( AnyType  x )
{
    List<AnyType> whichList = theLists[ myhash( x ) ];
    if( ! whichList.contains( x ) )
    {
        whichList.add( x );

        // Rehash is discussed below
        if( ++currentSize > theLists.length )
            rehash( );
    }
}
```

Duplicate check

Later, but essentially resizes the hashtable if its getting crowded

notice that we rehash when currentSize > tableSize, regardless of the length of any of the individual chains

Implementation of Chaining Hash Table

```
/**
 * Remove from the hash table.
 * @param x the item to remove.
 */
public void remove( AnyType  x )
{
    List<AnyType> whichList = theLists[ myhash( x ) ];

    if( whichList.contains( x ) )
    {
        whichList.remove( x );
        currentSize--;
    }
}
```

This operation takes time linear in the length of the list at the hashed index location

Implementation of Chaining Hash Table

```
/**
 * Find an item in the hash table.
 * @param x the item to search for.
 * @return true if x isnot found.
 */
public boolean contains( AnyType  x )
{
    List<AnyType> whichList = theLists[ myhash( x ) ];
    return whichList.contains( x );
}


/**
 * Make the hash table logically empty.
 */
public void makeEmpty( )
{
    for( int i = 0; i < theLists.length; i++ )
        theLists[ i ].clear( );
    currentSize = 0;
}
```

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

Implementation of Chaining Hash Table

```
/**
 * A hash routine for String objects.
 * @param key the String to hash.
 * @param tableSize the size of the hash table.
 * @return the hash value.
 */
public static int hash( String key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal = 37 * hashVal + key.charAt( i );

    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;

    return hashVal;
}
```

Recall that modulus returns the remainder of dividing its first argument by the second (i.e., its modulus).

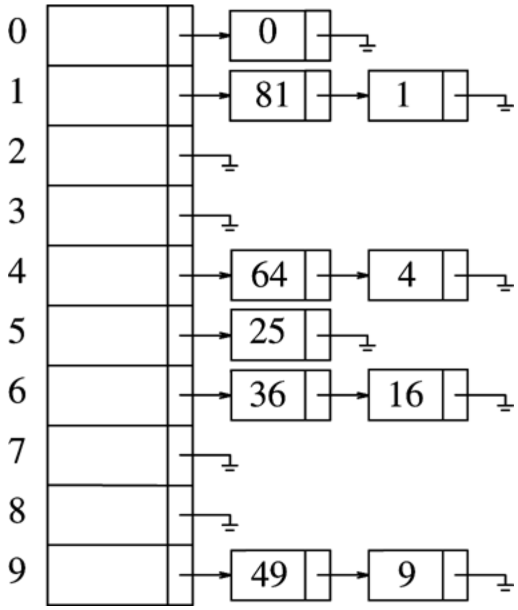
For negative values, the result may vary depending on the library implementation.

Implementation of Chaining Hash Table

```
@SuppressWarnings("unchecked")
private void rehash( )
{
    List<AnyType> [ ]  oldLists = theLists;

    // Create new double-sized, empty table
    theLists = new List[ nextPrime( 2 * theLists.length ) ];
    for( int j = 0; j < theLists.length; j++ )
        theLists[ j ] = new LinkedList<AnyType>( );

    // Copy table over
    currentSize = 0;
    for( int i = 0; i < oldLists.length; i++ )
        for( AnyType item : oldLists[ i ] )
            insert( item );
}
```



Implementation of Chaining Hash Table

```
private int myhash( AnyType x )
{
    int hashVal = x.hashCode( );

    hashVal %= theLists.length;
    if( hashVal < 0 )
        hashVal += theLists.length;

    return hashVal;
}
```

This is the hashtable's current capacity (aka "table size")

This is the hash table index for the element x

Recall that modulus returns the remainder of dividing its first argument by the second (i.e., its modulus).

For negative values, the result may vary depending on the library implementation.

Implementation of Chaining Hash Table

```
/**
 * Internal method to find a prime number at least as large as n.
 * @param n the starting number (must be positive).
 * @return a prime number larger than or equal to n.
 */
private static int nextPrime( int n )
{
    if( n % 2 == 0 )
        n++;

    for ( ; !isPrime( n ); n += 2 )
       ;

    return n;
}
```

Implementation of Chaining Hash Table

```
/**
 * Internal method to test if a number is prime.
 * Not an efficient algorithm.
 * @param n the number to test.
 * @return the result of the test.
 */
private static boolean isPrime( int n )
{
    if( n == 2 || n == 3 )
        return true;

    if( n == 1 || n % 2 == 0 )
        return false;

    for( int i = 3; i * i <= n; i += 2 )
        if( n % i == 0 )
            return false;

    return true;
}
```

Implementation of Chaining Hash Table

```
// Simple main - a typical client application
public static void main( String [ ] args )
{
    SeparateChainingHashTable<Integer> H = new
                                  SeparateChainingHashTable<Integer>( );

    final int NUMS = 40000;
    final int GAP  =   37;

    System.out.println( "Checking... (no more output means success)" );

    for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
        H.insert( i );

    for( int i = 1; i < NUMS; i+= 2 )
        H.remove( i );

    for( int i = 2; i < NUMS; i+=2 )
        if( !H.contains( i ) )
            System.out.println( "Find fails " + i );

    for( int i = 1; i < NUMS; i+=2 )
    {
        if( H.contains( i ) )
            System.out.println( "OOPS!!! " +  i  );
    }
}
}
```

Collision Resolution by 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 ("table size")
  - $\lambda = N/M$ ("load factor")
    - i.e., for chaining, λ is the average length of a chain
- unsuccessful search time: $O(\lambda)$
  - same for insert time
- successful search time: $O(\lambda/2)$
- the load factor λ is more significant than the table size
- make the table size a prime number for better distribution
- ideally, we want $\lambda \approx 1$ (not a function of N)
  - if the load factor exceeds 1,
    then call rehash (see slide #24)

Potential Disadvantages of Chaining

- 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

Collision Resolution Technique #2: Open Addressing

- Collision Resolution by Open Addressing
  - When a collision occurs, look elsewhere in the table for an empty slot.

- Advantages over chaining
  - no need for list structures *(not a serious disadvantage)*
  - 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 $\lambda \approx 0.5$

What does "average-case constant-time" mean?

... Independent of N

Collision Resolution by Open Addressing

- A "<u>Probe sequence</u>" is a sequence of slots in hash table while searching for an element x
  - $h_0(x), h_1(x), h_2(x), \ldots$
  - need to visit each slot exactly once
  - need to be repeatable (so we can find/delete what we've inserted)

- <u>Hash function</u>
  - $h_i(x) = (h(x) + f(i)) \bmod TableSize$
  - $f(0) = 0$ ==> 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*".

Linear Probing

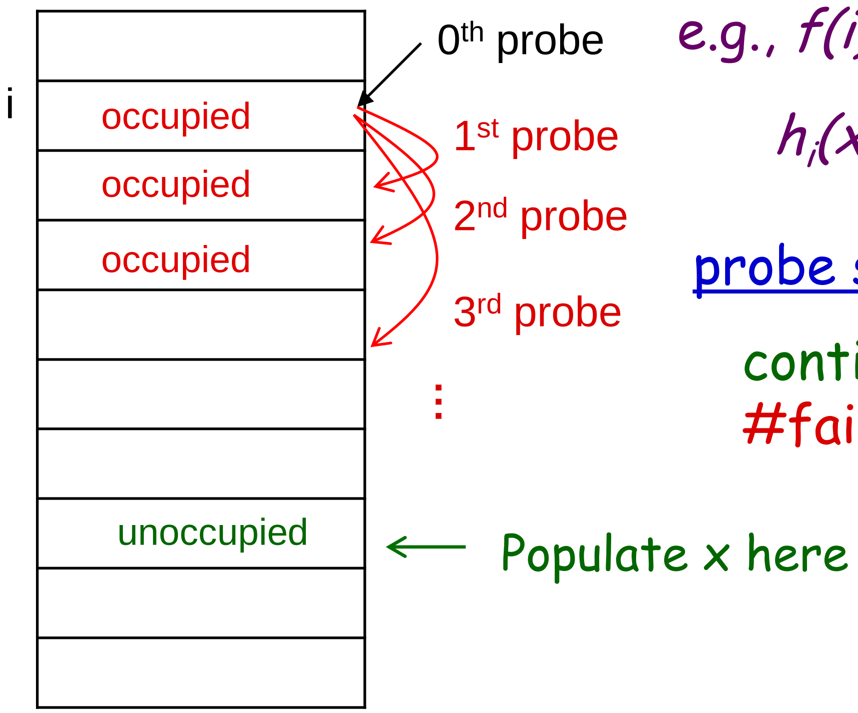
$f(i)$ = is a linear function of i,

i^{th} probe index =

0^{th} probe index

+ i

Linear probing:



e.g., $f(i) = i$

$$h_i(x) = (h(x) + i) \bmod TableSize$$

probe sequence: +0, +1, +2, +3, +4, …

continue until an empty slot is found

#failed probes is a measure of performance

Linear Probing

i^{th} probe index =

0^{th} probe index

+ i

- f(i) = is a linear function of i, e.g., $f(i) = i$

  $$h_i(x) = (h(x) + i) \bmod TableSize$$

  - Probe sequence: +0, +1, +2, +3, +4, …
- Example: h(x) = x mod 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)

    $h_1(49) = (h(49)+f(1)) \bmod 10$

    $= (h(49)+ \ 1\ ) \bmod 10 = 0$

Linear Probing Example

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

time →

	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	$\boxed{7}$ total

Linear Probing: Issues

Probe sequences can get longer with time

- *Primary clustering*
  - keys tend to cluster in some parts of the hash table
  - a cluster is a contiguous block of items
  - search through cluster using elementary algorithm for arrays
  - 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 Examples

- Linear probing: array of size M ⟸ typically twice as many slots as elements
  - Hash: map key to integer i between 0 and M-1
  - Insert: put in slot i if free, if not try i+1, i+2, etc.
  - Search: search slot i, if occupied but no match, try i+1, i+2, etc.

- Example
  - divisor = b (number of buckets) = 17
  - home bucket = key % 17

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	0	45				6	23	7			28	12	29	11	30	33

  - put in pairs whose keys are:
    - 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Linear Probing Example -- Remove

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	0	45				6	23	7			28	12	29	11	30	33

- remove(0)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34		45				6	23	7			28	12	29	11	30	33

  - search cluster for pair (if any) to fill vacated bucket

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	45					6	23	7			28	12	29	11	30	33

Linear Probing Example – remove(34)

34	0	45				6	23	7			28	12	29	11	30	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

	0	45				6	23	7			28	12	29	11	30	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- search cluster for pair (if any) to fill vacated bucket

0		45				6	23	7			28	12	29	11	30	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

0	45					6	23	7			28	12	29	11	30	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Linear Probing Example – remove(29)

34	0	45				6	23	7			28	12	**29**	11	30	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

34	0	45				6	23	7			28	12		11	30	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- search cluster for pair (if any) to fill vacated bucket

34	0	45				6	23	7			28	12	11		30	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

34	0	45				6	23	7			28	12	11	30		33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

34	0					6	23	7			28	12	11	30	**45**	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Linear Probing: Analysis

- Expected number of probes for unsuccessful search and insertion

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

- Expected number of probes for successful search

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

- Example (λ = 0.5)
  - Insert and unsuccessful search
    - 2.5 probes
  - Successful search
    - 1.5 probes
- Example (λ = 0.9)
  - Insert / unsuccessful search
    - 50.5 probes
  - Successful search
    - 5.5 probes

Random Probing: Analysis

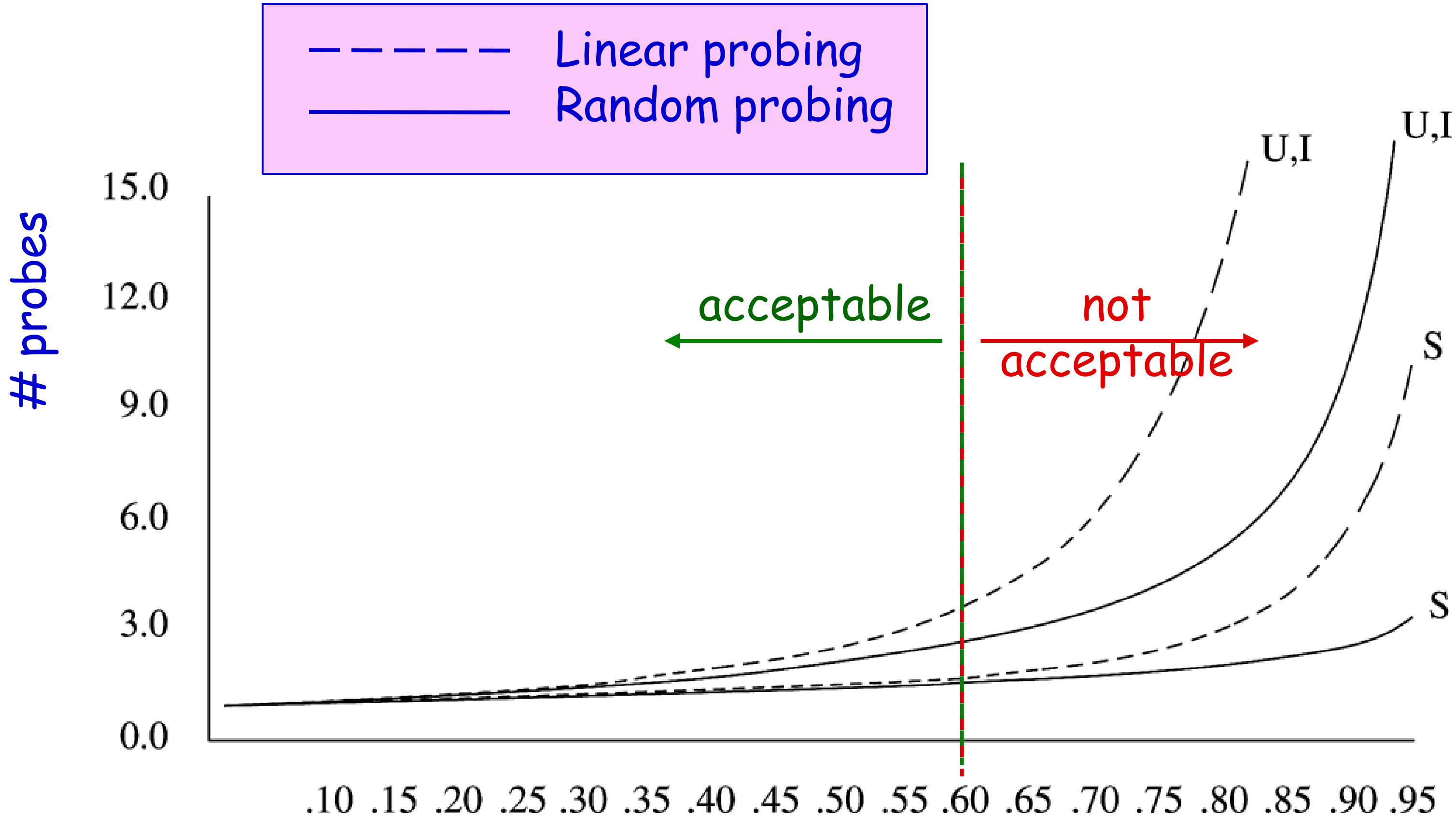
- Random Probing:
  - a pseudo-random number generator is used to obtain a random sequence R(i), 1 <= i < b where R(1), R(2), ... R(b-1) is a permutation of [1, 2, ..., b-1].
  - buckets are examined in the order f(k), (f(k)+R(i)) % b, 1 <= i < b.
- Random probing does not suffer from clustering
- Expected number of probes for insertion or unsuccessful search:

$$\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

  - This result is due to averaging and the fact that the clusters grow incrementally
  - Example
    - $\lambda = 0.5$: 1.4 probes
    - $\lambda = 0.9$: 2.6 probes

Briefly

Linear vs. Random Probing



U - unsuccessful search
S - successful search
I - insert

Load factor λ

Briefly

Quadratic Probing

- Avoids primary clustering
- f(i) is quadratic in i

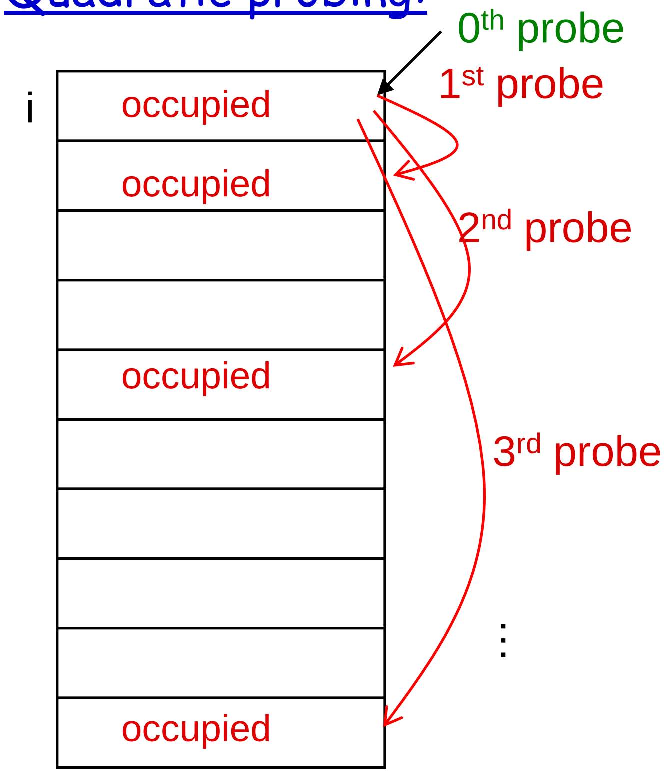
  e.g., $f(i) = i^2$

  $h_i(x) = (h(x) + i^2) \bmod TableSize$

  → Probe sequence:

  +0, +1, +4, +9, +16, …

Quadratic probing:



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., $f(i) = i^2$

  $h_i(x) = (h(x) + i^2) \bmod TableSize$

  - Probe sequence: +0, +1, +4, +9, +16, …
- 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

<u>Insert sequence:</u> 89, 18, 49, 58, 69

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

#unsuccessful probes:	0	0	1	2	2	**5** total

Quadratic Probing: Analysis

... difficult to analyze

- Theorem:
  - a 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

- Solution: Ensure table never gets half full
  - if close, then expand it

- collision sequences generated by addresses calculated with quadratic probing are called "secondary clustering"

Quadratic Probing: Deletion

- Deletion
  - emptying slots can break probe sequence and cause '`findPos`' to 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 the '`findPos`' procedure returning "not found"
  - may need compaction at some time

Quadratic Probing: Class Interface

```
// QuadraticProbing Hash table class
//
// CONSTRUCTION: an approximate initial size or default of 101
//
// ******************PUBLIC OPERATIONS*********************
// bool insert( x )       --> Insert x
// bool remove( x )       --> Remove x
// bool contains( x )     --> Return true if x is present
// void makeEmpty( )      --> Remove all items


/**
 * Probing table implementation of hash tables.
 * Note that all "matching" is based on the equals method.
 * @author Mark Allen Weiss
 */
```

Quadratic Probing: Class Interface

```
public class QuadraticProbingHashTable<AnyType>
{
    private static final int DEFAULT_TABLE_SIZE = 11;
    private HashEntry<AnyType> [ ] array;   // The array of elements
    private int currentSize;           // The number of occupied cells

    /**
     * Construct the hash table.
     */
    public QuadraticProbingHashTable( )
    {
        this( DEFAULT_TABLE_SIZE );
    }

    /**
     * Construct the hash table.
     * @param size the approximate initial size.
     */
    public QuadraticProbingHashTable( int size )
    {
        allocateArray( size );
        makeEmpty( );
    }
```

Quadratic Probing: Class Interface

```
/**
 * private nested class.
 */
private static class HashEntry<AnyType>
{
    public AnyType  element;   // the element
    public boolean isActive;   // false if marked deleted

    // construct an element
    public HashEntry( AnyType e )
    {
        this( e, true );
    }

    // construct an element
    public HashEntry( AnyType e, boolean i )
    {
        element  = e;
        isActive = i;
    }
}
```

Quadratic Probing: Class Interface

```java
/**
 * Insert into the hash table. If the item is
 * already present, do nothing.
 * @param x the item to insert.
 */
public void insert( AnyType x )
{
    // Insert x as active
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        return;

    array[ currentPos ] = new HashEntry<AnyType>( x, true );

    // Rehash
    if( ++currentSize > array.length / 2 )
        rehash( );
}
```

no duplicates

ensure table size is at least twice the number of elements

Quadratic Probing: Class Interface

```java
/**
 * Expand the hash table.
 */
private void rehash( )
{
    HashEntry<AnyType> [ ] oldArray = array;

    // Create a new double-sized, empty table
    allocateArray( nextPrime( 2 * oldArray.length ) );
    currentSize = 0;

    // Copy table over
    for( int i = 0; i < oldArray.length; i++ )
        if( oldArray[ i ] != null && oldArray[ i ].isActive )
            insert( oldArray[ i ].element );
}
```

ensure table size is prime

What about `oldArray`?

Quadratic Probing: Class Interface

```
/**
 * Method that performs quadratic probing resolution.
 * Assumes table is at least half empty and table length is prime.
 * @param x the item to search for.
 * @return the position where the search terminates.
 */
private int findPos( AnyType x )
{
    int offset = 1;
    int currentPos = myhash( x );

    while( array[ currentPos ] != null &&
             !array[ currentPos ].element.equals( x ) )
    {
        currentPos += offset;  // Compute ith probe
        offset += 2;
        if( currentPos >= array.length )
            currentPos -= array.length;
    }

    return currentPos;
}
```

skip DELETED;
no duplicates

generates
$f(i) = f(i - 1) + 2i - 1$

Quadratic Probing: Class Interface

```
/**
 * Remove from the hash table.
 * @param x the item to remove.
 */
public void remove( AnyType x )
{
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
        array[ currentPos ].isActive = false;
}
```

just mark it deleted

```
/**
 * Find an item in the hash table.
 * @param x the item to search for.
 * @return the matching item.
 */
public boolean contains( AnyType x )
{
    int currentPos = findPos( x );
    return isActive( currentPos );
}
```

Quadratic Probing: Class Interface

```
/**
 * Return true if currentPos exists and is active.
 * @param currentPos the result of a call to findPos.
 * @return true if currentPos is active.
 */
private boolean isActive( int currentPos )
{
    return array[ currentPos ] != null && array[ currentPos ].isActive;
}


/**
 * Make the hash table logically empty.
 */
public void makeEmpty( )
{
    currentSize = 0;
    for( int i = 0; i < array.length; i++ )
        array[ i ] = null;
}
```

Quadratic Probing: Class Interface

```java
private int myhash( AnyType x )
{
    int hashVal = x.hashCode( );

    hashVal %= array.length;
    if( hashVal < 0 )
        hashVal += array.length;

    return hashVal;
}


/**
 * Internal method to allocate array.
 * @param arraySize the size of the array.
 */
 @SuppressWarnings("unchecked")
private void allocateArray( int arraySize )
{
    array = new HashEntry[ nextPrime( arraySize ) ];
}
```

Quadratic Probing: Class Interface

```
/**
 * Internal method to find a prime number at least as large as n.
 * @param n the starting number (must be positive).
 * @return a prime number larger than or equal to n.
 */
private static int nextPrime( int n )
{
    if( n <= 0 )
        n = 3;

    if( n % 2 == 0 )
        n++;

    for( ; !isPrime( n ); n += 2 )
        ;

    return n;
}
```

Quadratic Probing: Class Interface

```
/**
 * Internal method to test if a number is prime.
 * Not an efficient algorithm.
 * @param n the number to test.
 * @return the result of the test.
 */
private static boolean isPrime( int n )
{
    if( n == 2 || n == 3 )
        return true;

    if( n == 1 || n % 2 == 0 )
        return false;

    for( int i = 3; i * i <= n; i += 2 )
        if( n % i == 0 )
            return false;

    return true;
}
```

Quadratic Probing: Class Interface

```
    // Simple main - client application
    public static void main( String [ ] args )
    {
        QuadraticProbingHashTable<String> H = new QuadraticProbingHashTable<String>( );

        final int NUMS = 400000;
        final int GAP  =   37;

        System.out.println( "Checking... (no more output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            H.insert( ""+i );
        for( int i = 1; i < NUMS; i+= 2 )
            H.remove( ""+i );

        for( int i = 2; i < NUMS; i+=2 )
            if( !H.contains( ""+i ) )
                System.out.println( "Find fails " + i );

        for( int i = 1; i < NUMS; i+=2 )
        {
            if( H.contains( ""+i ) )
                System.out.println( "OOPS!!! " +  i  );
        }
    }
}
```

Double Hashing: keep two hash functions h_1 and h_2

- Use a second hash function for all tries i other than 0:
  $f(i) = i * h_2(x)$
- 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)
  - $h_1(49) = (h(49) + 1*(7 - 49 \bmod 7)) \bmod 10 = 6$

f(1)

Double Hashing Example

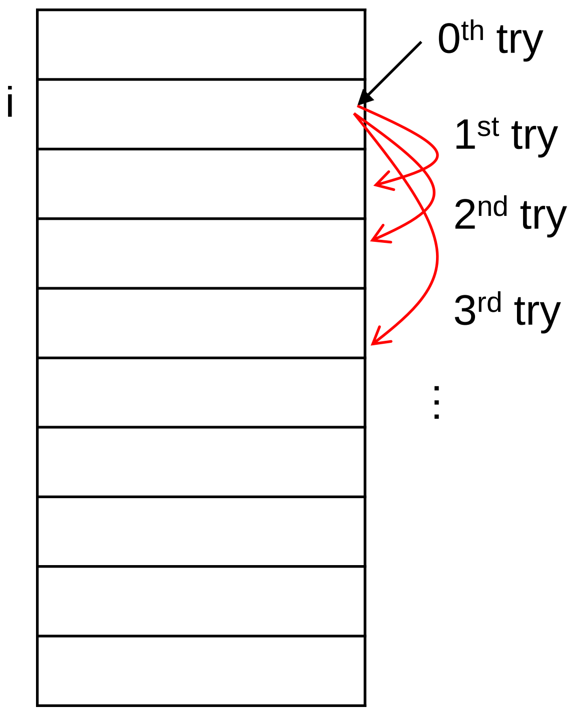
	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Double Hashing: Analysis

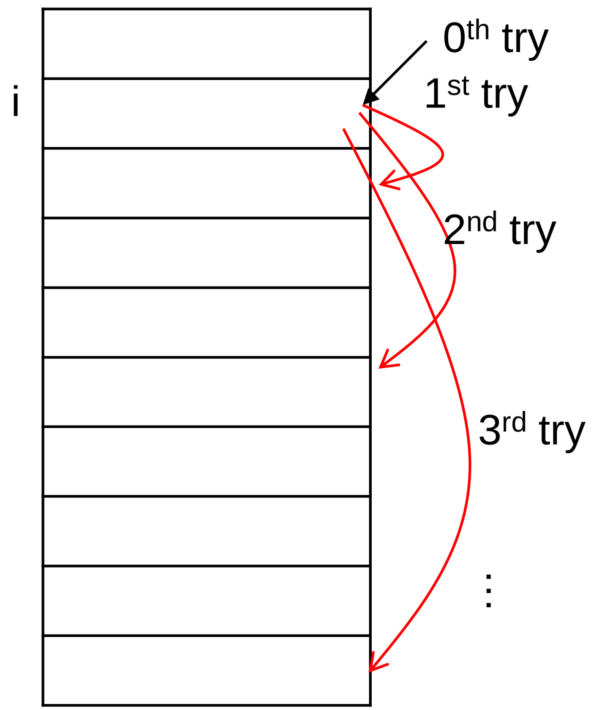
- Imperative that TableSize is prime
  - E.g., insert 23 into previous table
- Empirical tests show double hashing close to random hashing
- Extra hash function takes extra time to compute

Probing Techniques - Review

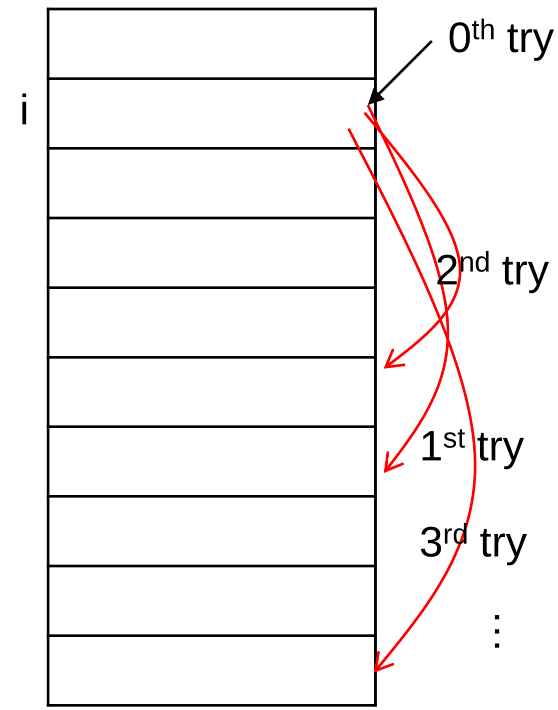
Linear probing:



Quadratic probing:



Double hashing*:

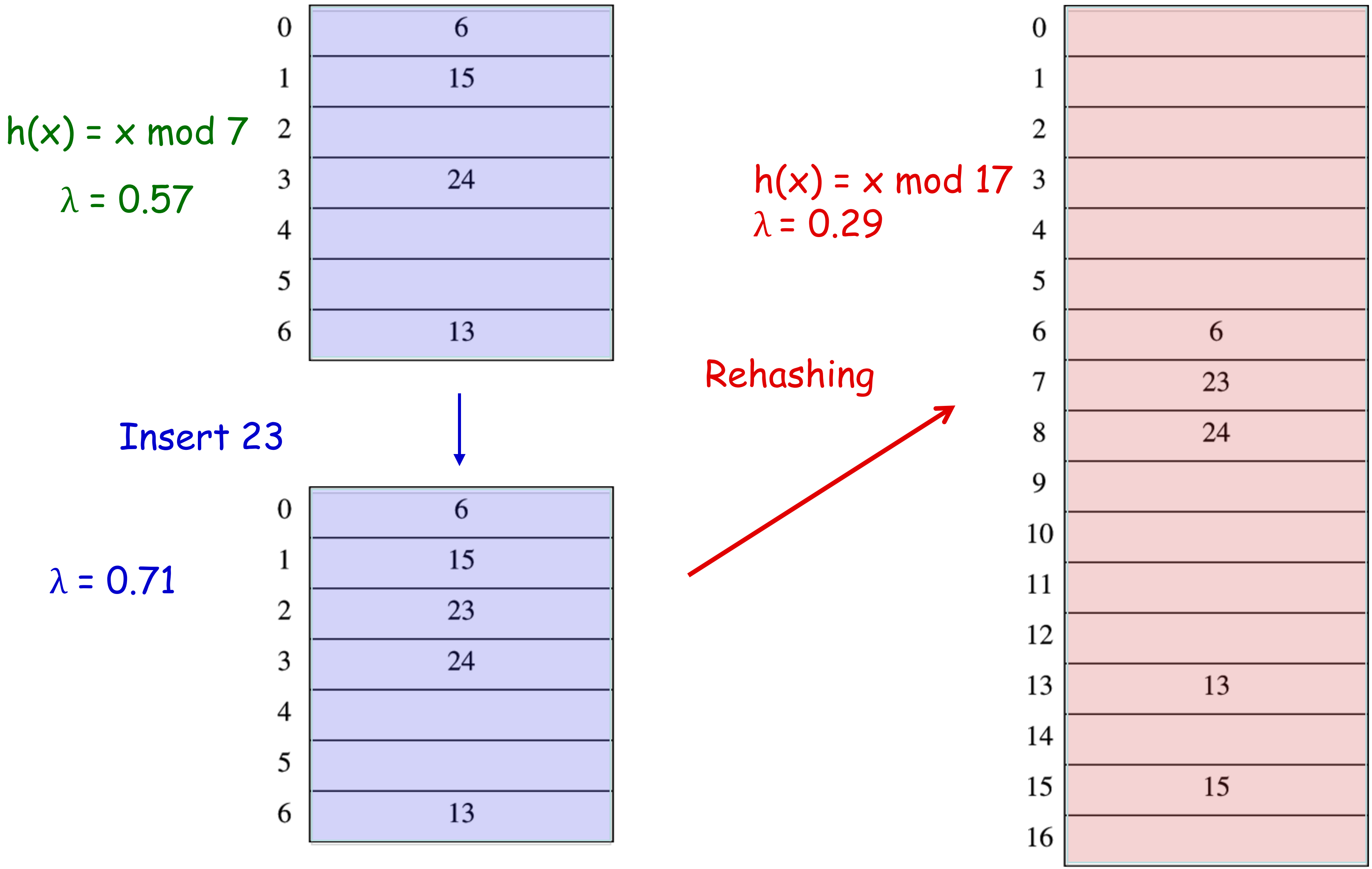


*(determined by a second hash function)

Rehashing

- Increases the size of the hash table when load factor becomes "too high" (defined by a cutoff)
  - Anticipating that probability of 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



Rehashing Analysis

- Rehashing takes time to do N insertions
- 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 Chaining

```
/**
 * Rehashing for separate chaining hash table.
 */
void rehash( )
{
    vector<list<HashedObj> > oldLists = theLists;

        // Create new double-sized, empty table
    theLists.resize( nextPrime( 2 * theLists.size( ) ) );
    for( int j = 0; j < theLists.size( ); j++ )
        theLists[ j ].clear( );

        // Copy table over
    currentSize = 0;
    for( int i = 0; i < oldLists.size( ); i++ )
    {
        list<HashedObj>::iterator itr = oldLists[ i ].begin( );
        while( itr != oldLists[ i ].end( ) )
            insert( *itr++ );
    }
}
```

Rehashing for Quadratic Probing

```
/**
 * Rehashing for quadratic probing hash table.
 */
void rehash( )
{
    vector<HashEntry> oldArray = array;

        // Create new double-sized, empty table
    array.resize( nextPrime( 2 * oldArray.size( ) ) );
    for( int j = 0; j < array.size( ); j++ )
        array[ j ].info = EMPTY;

        // Copy table over
    currentSize = 0;
    for( int i = 0; i < oldArray.size( ); i++ )
        if( oldArray[ i ].info == ACTIVE )
            insert( oldArray[ i ].element );
}
```

gets rid of elements with a "deleted" tag

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

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., Perl, Python, etc.

Symbol Table: Implementations Cost Summary

	Worst Case			Average Case		
implementation	Search	Insert	Delete	Search	Insert	Delete
Sorted Array	log N	N	N	log N	N / 2	N / 2
Unsorted List	N	1	1	N / 2	1	1
Binary Search Tree	N	N	N	log N	log N	sqrt(N)
Hashing	N	1	N	1	1	1

assumes that hash function is random

Summary

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

Points 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 or double hashing
- Rehashing required to resize hash table at a time when λ exceeds 0.5
- Good for searching.
- Not good if there is some order implied by data.