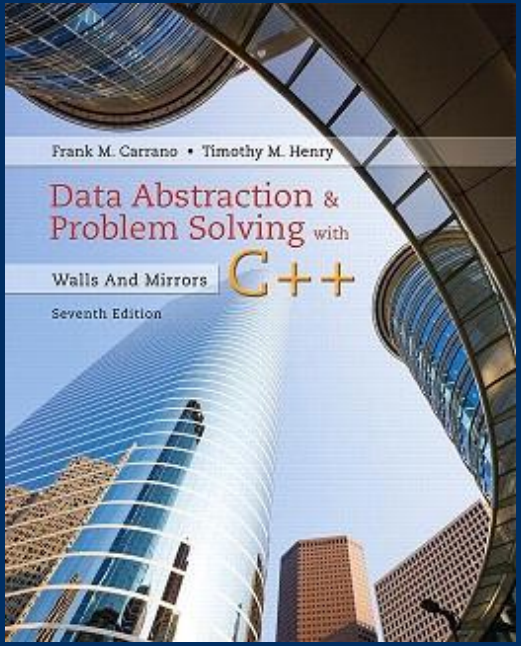
Dictionaries and Their Implementations

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

Pop Quiz

- What are the names of to TAs?
- Where do they held their office hours?
- Where is the office of the instructor?

Reminders

- Assignment 5 is available
  - Due April 11th at 2pm
- TA
  - Athanasia Katsila,
    Email: akatsila [at] nevada {dot} unr {dot} edu,
    Office Hours: Tuesday, 10:30 am - 12:30 pm at SEM 211
- Quiz 9 is available
  - Today between 4pm to 11:59pm

Contents

- The ADT Dictionary
- Possible Implementations
- Selecting an Implementation
- Hashing

The ADT Dictionary

City	Country	Population
Buenos Aires	Argentina	13,639,000
Cairo	Egypt	17,816,000
Johannesburg	South Africa	7,618,000
London	England	8,586,000
Madrid	Spain	5,427,000
Mexico City	Mexico	19,463,000
Mumbai	India	16,910,000
New York City	U.S.A.	20,464,000
Paris	France	10,755,000
Sydney	Australia	3,785,000
Tokyo	Japan	37,126,000
Toronto	Canada	6,139,000

- A collection of data about certain cities

The ADT Dictionary

- Consider need to search such a collection for
  - Name
  - Address

- Criterion chosen for search is *search key*
- The ADT dictionary uses a search key to identify its entries

Examples

- Telephone directory
- Library catalogue
- Books in print: key ISBN
- FAT (File Allocation Table)

Main Issues

- Size
- Operations: search, insert, delete,
  - < Create reports, Lists>
- What will be stored in the dictionary?
- How will be items identified?

The ADT Dictionary

```
Dictionary
----------------------------------------------------------------

----------------------------------------------------------------
+isEmpty(): boolean
+getNumberOfEntries(): integer
+add(searchKey: KeyType, newValue: ValueType): boolean
+remove(targetKey: KeyType): boolean
+clear(): void
+getValue(targetKey: KeyType): ValueType
+contains(targetKey: KeyType): boolean
+traverse(visit(value: ValueType): void): void
```

- UML diagram for a class of dictionaries

Possible Implementations

- Categories of linear implementations
  - Sorted by search key array-based
  - Sorted by search key link-based
  - Unsorted array-based
  - Unsorted link-based

Search key	Data Value

A dictionary entry

Possible Implementations

```
/** A class of entries to add to an array-based implementation of the
 ADT dictionary.
 @file Entry.h */

#ifndef ENTRY_
#define ENTRY_

template <class KeyType, class ValueType>
class Entry
{
private:
   KeyType key;
   ValueType value;

protected:
   void setKey(const KeyType& searchKey);
```

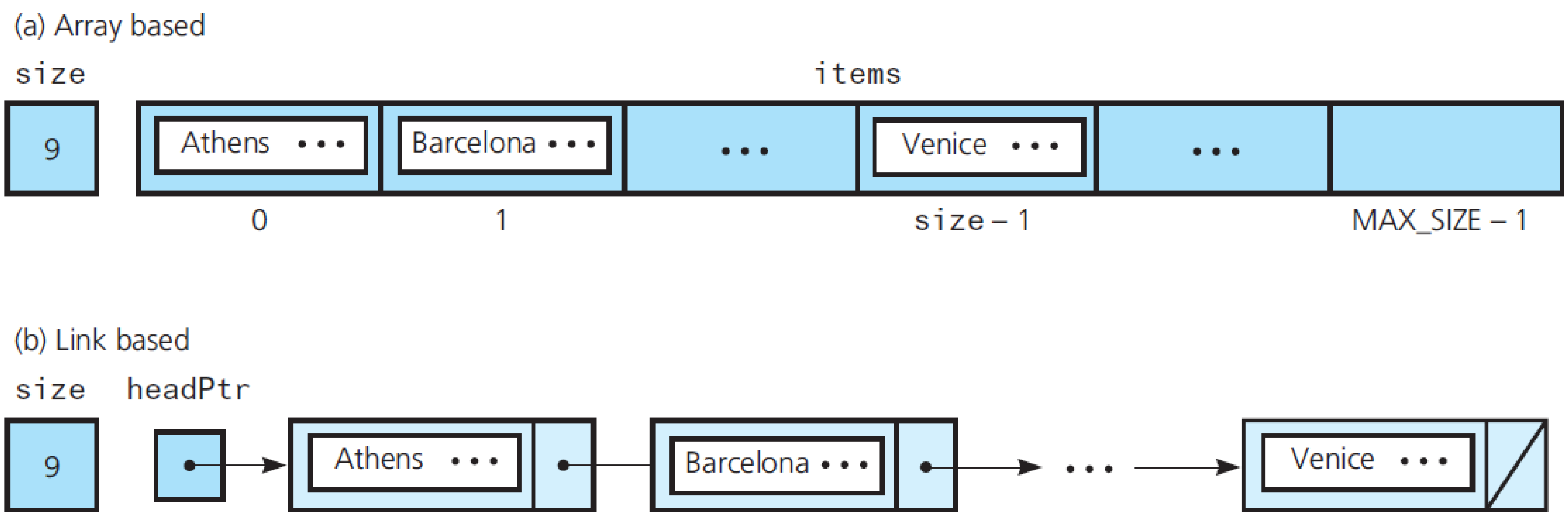
- A header file for a class of dictionary entries

Possible Implementations

```
    void setKey(const KeyType& searchKey);

public:
    Entry();
    Entry(const KeyType& searchKey, const ValueType& newValue);
    ValueType getValue() const;
    KeyType getKey() const;
    void setValue(const ValueType& newValue);

    bool operator==(const Entry<KeyType, ValueType>& rightHandValue) const;
    bool operator>(const Entry<KeyType, ValueType>& rightHandValue) const;
}; // end Entry
#include "Entry.cpp"
#endif
```

- A header file for a class of dictionary entries

Possible Implementations



- Data members for two sorted linear implementations of the ADT dictionary

Possible Implementations

```
/** An interface for the ADT dictionary.
 @file DictionaryInterface.h */

#ifndef DICTIONARY_INTERFACE_
#define DICTIONARY_INTERFACE_

#include "NotFoundException.h"

template<class KeyType, class ValueType>
class DictionaryInterface
{
public:
   /** Sees whether this dictionary is empty.
    @return  True if the dictionary is empty;
       otherwise returns false. */
   virtual bool isEmpty() const = 0;

   /** Gets the number of entries in this dictionary.
    @return  The number of entries in the dictionary. */
   virtual int getNumberOfEntries() const = 0;
```

- A header file for a class of dictionary interface

Possible Implementations

```
/** Adds a new search key and associated value to this dictionary.
 @pre  The new search key differs from all search keys presently
     in the dictionary.
 @post  If the addition is successful, the new key-value pair is in its
     proper position within the dictionary.
 @param searchKey  The search key associated with the value to be added.
 @param newValue  The value to be added.
 @return  True if the entry was successfully added, or false if not. */
virtual bool add(const KeyType& searchKey, const ValueType& newValue) = 0;

/** Removes a key-value pair from this dictionary.
 @post  If the entry whose search key equals searchKey existed in the
   dictionary, the entry was removed.
 @param searchKey  The search key of the entry to be removed.
 @return  True if the entry was successfully removed, or false if not. */
virtual bool remove(const KeyType& searchKey) = 0;

/** Removes all entries from this dictionary. */
virtual void clear() = 0;
```

- A header file for a class of dictionary interface

Possible Implementations

```
   virtual void clear() = 0;

   /** Retrieves the value in this dictionary whose search key is given
    @post  If the retrieval is successful, the value is returned.
    @param searchKey  The search key of the value to be retrieved.
    @return  The value associated with the search key.
    @throw  NotFoundException if the key-value pair does not exist. */
   virtual ValueType getValue(const KeyType& searchKey) const
                              throw (NotFoundException) = 0;

   /** Sees whether this dictionary contains an entry with a given search key.
    @post  The dictionary is unchanged.
    @param searchKey  The given search key.
    @return  True if an entry with the given search key exists in the
      dictionary. */
   virtual bool contains(const KeyType& searchKey) const = 0;
```

- A header file for a class of dictionary interface

Possible Implementations

```
   virtual bool contains(const KeyType& searchKey) const = 0;

   /** Traverses this dictionary and calls a given client function once
       for each entry.
    @post  The given function's action occurs once for each entry in the
       dictionary and possibly alters the entry.
    @param visit  A client function. */
   virtual void traverse(void visit(ValueType&)) const = 0;
   /** Destroys this dictionary and frees its assigned memory. */

   virtual ~DictionaryInterface(){ }
}; // end DictionaryInterface
#endif
```

- A header file for a class of dictionary interface

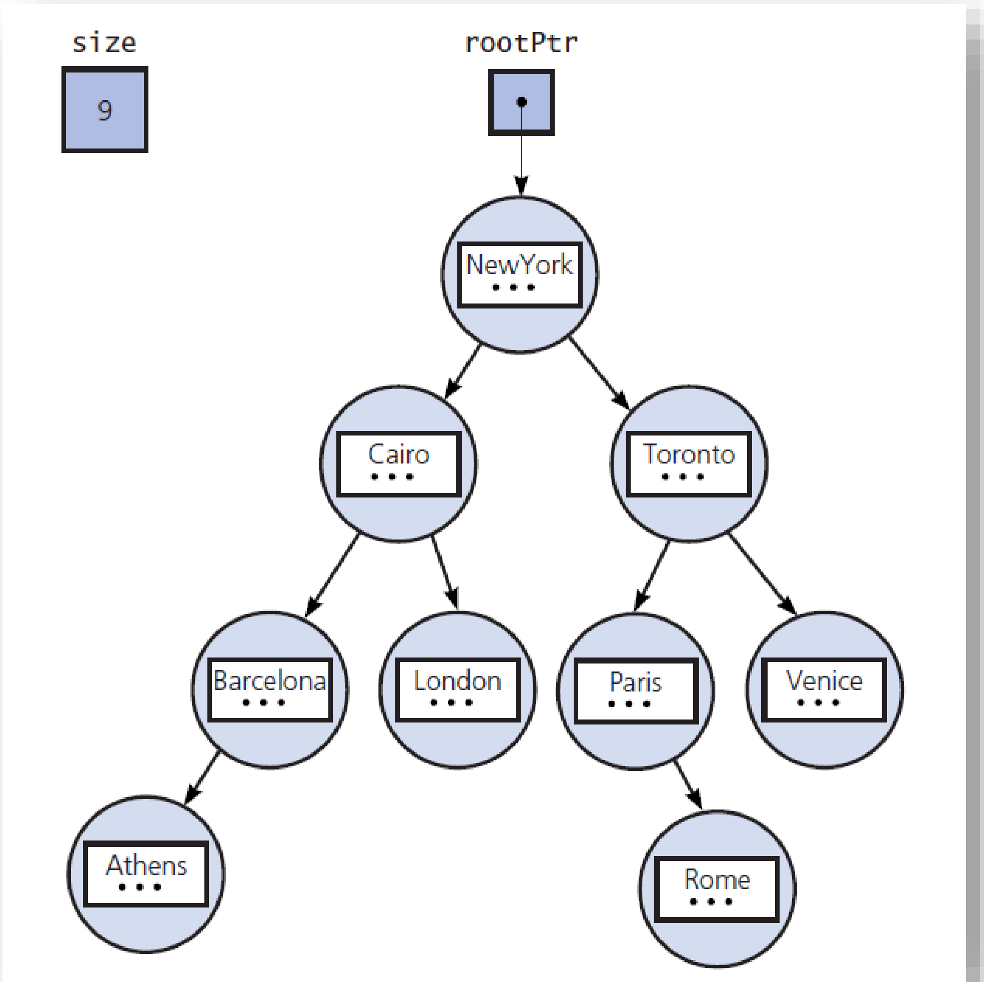
Sorted Array-Based Implementation of ADT Dictionary

```
public:
   ArrayDictionary();
   ArrayDictionary(int maxNumberOfEntries);
   ArrayDictionary(const ArrayDictionary<KeyType, ValueType>& dictionary);

   virtual ~ArrayDictionary();

   bool isEmpty() const;
   int getNumberOfEntries() const;
   bool add(const KeyType& searchKey, const ValueType& newValue) throw(PrecondViolatedExcept);
   bool remove(const KeyType& searchKey);
   void clear();
   ValueType getValue(const KeyType& searchKey) const throw(NotFoundException);
   bool contains(const KeyType& searchKey) const;

   /** Traverses the entries in this dictionary in sorted search-key order
       and calls a given client function once for the value in each entry. */
   void traverse(void visit(ValueType&)) const;
}; // end ArrayDictionary
#include "ArrayDictionary.cpp"
#endif
```

- A header file for the class ArrayDictionary

Possible Implementations

- The data members for a binary search tree implementation of the ADT dictionary



Binary Search Tree Implementation of the ADT Dictionary

```
/** A binary search tree implementation of the ADT dictionary
 that organizes its data in sorted search-key order.
 Search keys in the dictionary are unique.
 @file TreeDictionary.h */

#ifndef TREE_DICTIONARY_
#define TREE_DICTIONARY_

#include "DictionaryInterface.h"
#include "BinarySearchTree.h"
#include "Entry.h"
#include "NotFoundException.h"
#include "PrecondViolatedExcept.h"

template <class KeyType, class ValueType>
class TreeDictionary : public DictionaryInterface<KeyType, ValueType>
{
private:
```

- A header file for the class TreeDictionary

Binary Search Tree Implementation of the ADT Dictionary

```
{
private:
   // Binary search tree of dictionary entries
   BinarySearchTree<Entry<KeyType, ValueType> > entryTree;

public:
   TreeDictionary();
   TreeDictionary(const TreeDictionary<KeyType, ValueType>& dictionary);

   virtual ~TreeDictionary();

   // The declarations of the public methods appear here and are the
   // same as given in Listing 18-3 for the class ArrayDictionary.
   . . .
}; // end TreeDictionary
#include "TreeDictionary.cpp"
#endif
```

- A header file for the class TreeDictionary

Binary Search Tree Implementation of the ADT Dictionary

```
template < class KeyType, class ValueType>
bool TreeDictionary<KeyType, ValueType>::add(const KeyType& searchKey,
                                              const ValueType& newValue)
                                              throw(PrecondViolatedExcept)
{
   Entry<KeyType, ValueType> newEntry(searchKey, newValue);

   // Enforce precondition: Ensure distinct search keys
   if (!itemTree.contains(newEntry))
   {
      // Add new entry and return boolean result
      return itemTree.add(Entry<KeyType, ValueType>(searchKey, newValue));
   }
   else
   {
      auto message = "Attempt to add an entry whose search key exists in dictionary.";
      throw(PrecondViolatedExcept(message)); // Exit the method
   }  // end if
} // end add
```

- Method add which prevents duplicate keys.

Selecting an Implementation

- Linear implementations
  - Perspective
  - Efficiency
  - Motivation

- Consider
  - What operations are needed
  - How often each operation is required

Three Scenarios

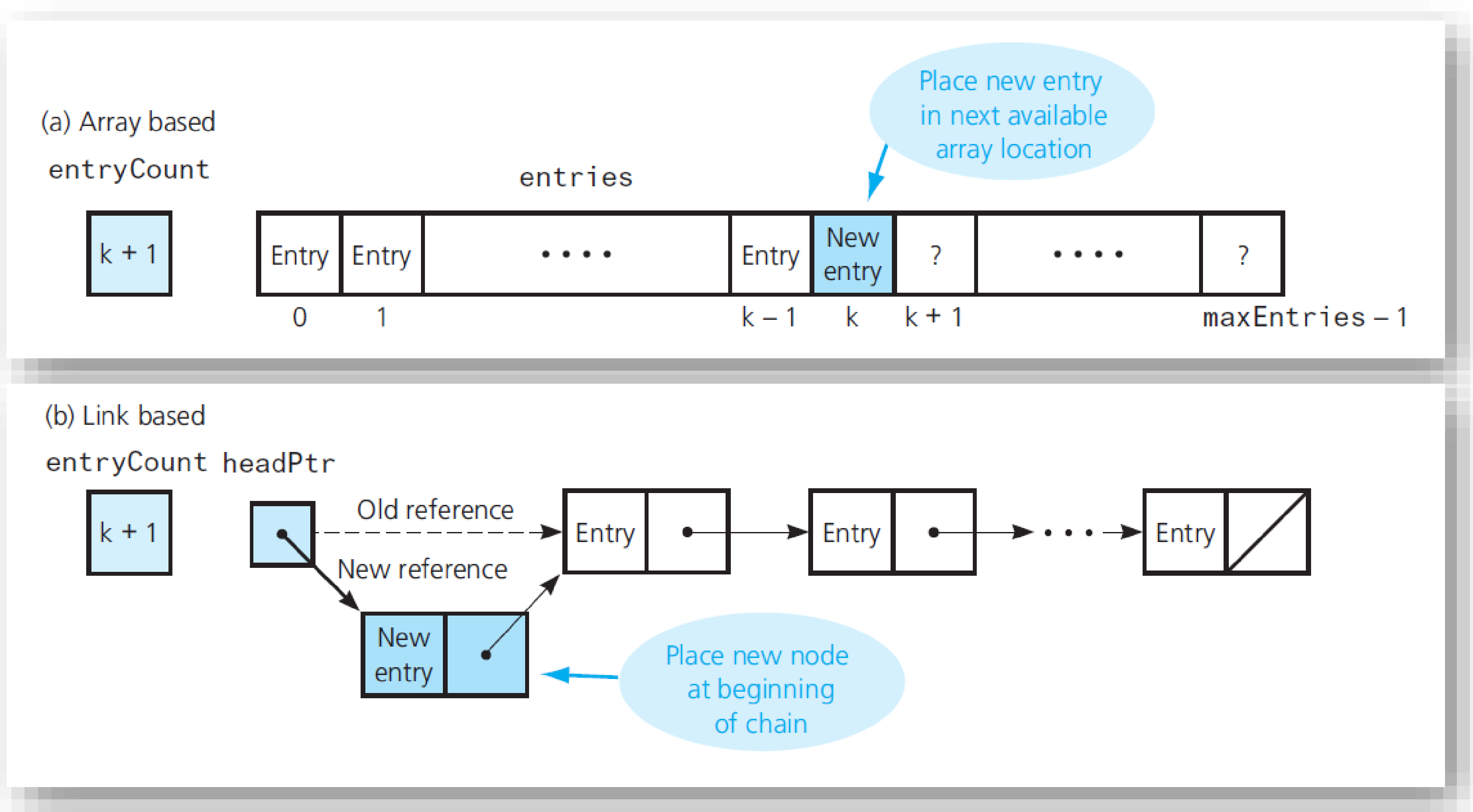
A. Addition and traversal in no particular order
   - Unsorted order is efficient
   - Array-based versus pointer-based

B. Retrieval
   - Sorted array-based can use binary search
   - Binary search impractical for link-based
   - Max size of dictionary affects choice

Three Scenarios

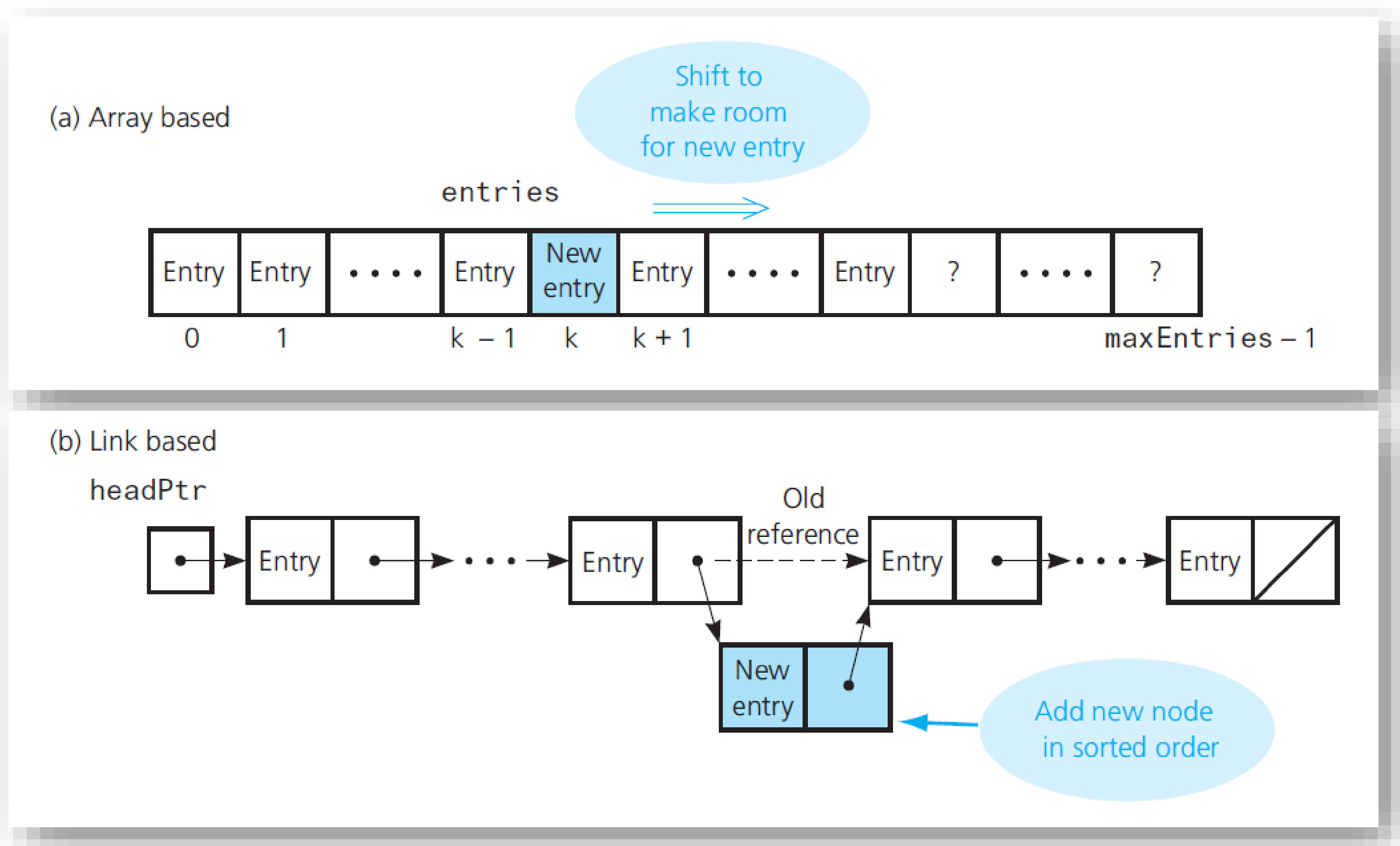
C. Addition, removal, retrieval, traversal in sorted order
   - Add and remove need to find position, then add or remove from that position
   - Array-based best for find, link-based best for addition/removal

Three Scenarios



- Addition for unsorted linear implementations

Three Scenarios



- Addition for sorted linear implementations

Three Scenarios

	Addition	Removal	Retrieval	Traversal
Unsorted array-based	O(1)	O(*n*)	O(*n*)	O(*n*)
Unsorted link-based	O(1)	O(*n*)	O(*n*)	O(*n*)
Sorted array-based	O(*n*)	O(*n*)	O(log *n*)	O(*n*)
Sorted link-based	O(*n*)	O(*n*)	O(*n*)	O(*n*)
Binary search tree	O(log *n*)	O(log *n*)	O(log *n*)	O(*n*)

- The average-case order of the ADT dictionary operations for various implementations

Direct Addressing

The Search Problem

- Unsorted list
  - O(N)
- Sorted list
  - O(logN) using arrays (i.e., binary search)
  - O(N) using linked lists
- Binary Search tree
  - O(logN) (i.e., balanced tree)
  - O(N) (i.e., unbalanced tree)
- Can we do better than this?
  - Direct Addressing
  - Hashing

Direct Addressing

- Assumptions:
  - Key values are **distinct**
  - Each key is drawn from a universe $U = \{0, 1, . . . , n - 1\}$
- Idea:
  - Store the items in an array, indexed by keys

Direct Addressing (cont'd)

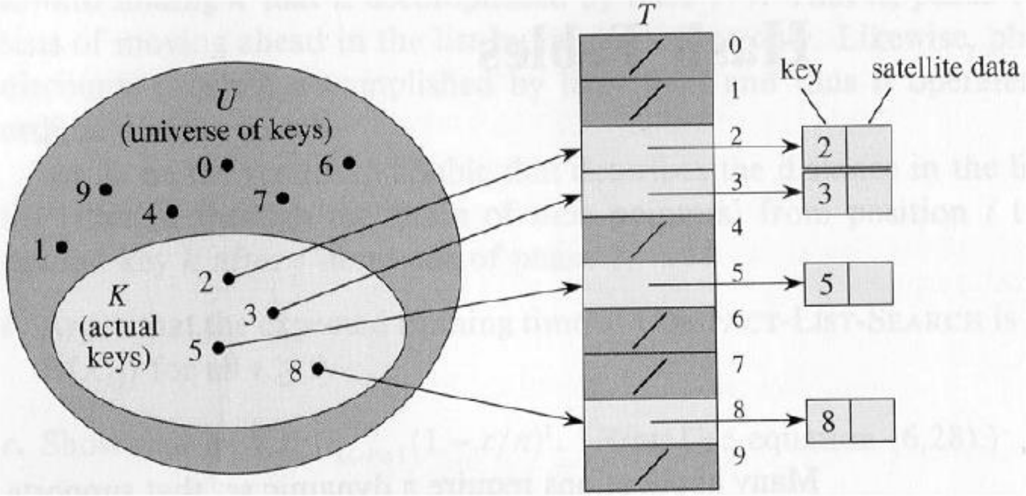
- **Direct-address table** representation:

– An array T[0 . . . n - 1]

– Each **slot**, or position, in T corresponds to a key in U

– For an element x with key k, a pointer to x will be placed in location T[k]

– If there are no elements with key k in the set, T[k] is empty, represented by NIL



Search, insert, delete in O(1) time!

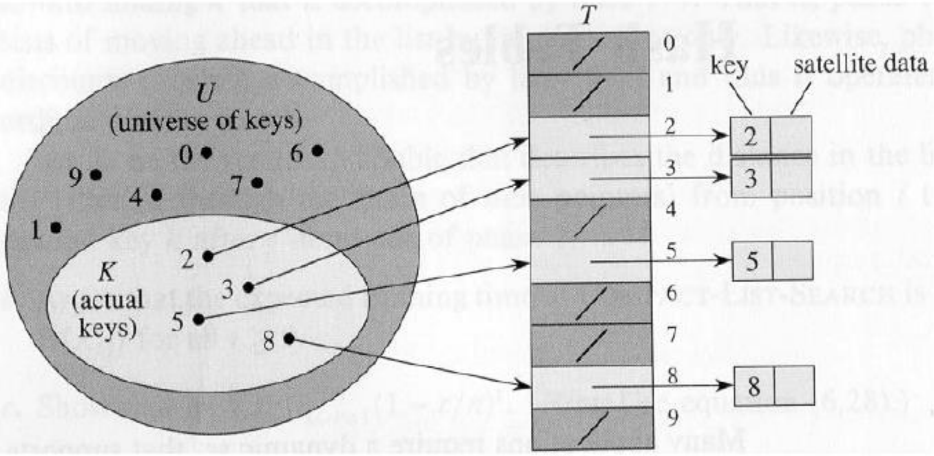
Direct Addressing (cont'd)

Example 1: Suppose that there are integers from 1 to 100 and that there are about 100 records.

Create an array A of 100 items and stored the record whose key is equal to **i** in **A[i].**

|K| = |U|

|K|: # elements in K
|U|: # elements in U

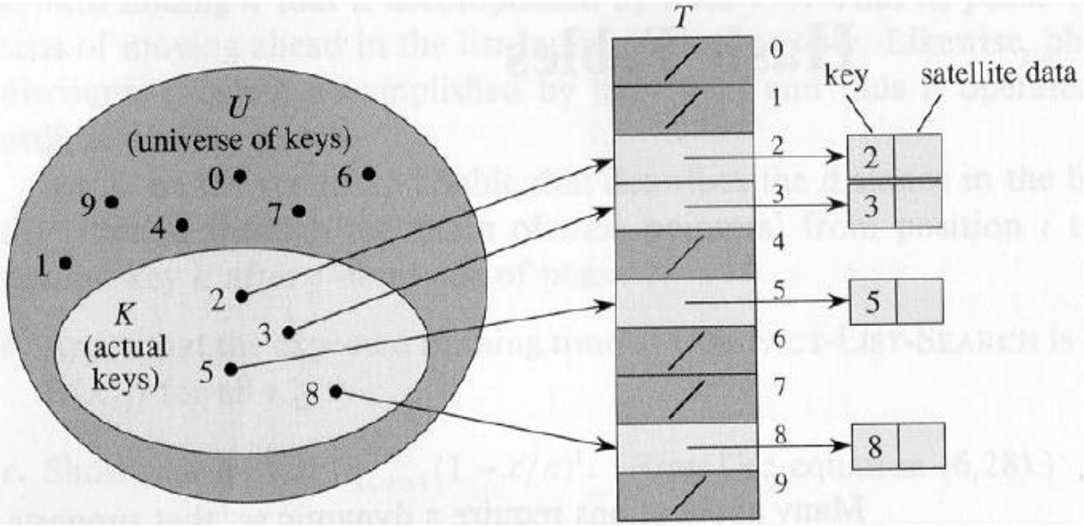


Direct Addressing (cont'd)

Example 2: Suppose that the keys are 9-digit social security numbers (SSN)

Although we could use the same idea, it would be very inefficient (i.e., use an array of 1 billion size to store 100 records)

|K| << |U|



```
/*
 * number Declaration
 */
struct number
{
    int key;
    string english_text;

    // Constructors
    number(){}
    number( int k, string e )
    {
        english_text=e;
        key = k;
    }
};
```

```
/*
 * Main Contains Menu
 */
int main()
{
    int i, key, ch;
    string str;
    number x;

    number T[65536];

    for(i = 0; i < 65536;i++)
        T[i] = number(0,"");
```

```
/*
 * Insertion of element at a key
 */
void INSERT( number T[], number x )
{
    T[ x.key ] = x;
}
```

```
/*
 * Deletion of element at a key
 */
void DELETE( number T[], number x )
{
    T[ x.key ] = number(0, "");
}
```

```
/*
 * Searching of element at a key
 */
number SEARCH( number T[], int k )
{
    return T[ k ];
}
```

```
while (1)
{
    cout<<"\n-------------------------------------"<<endl;
    cout<<"\nOperations on Direct Addressing number"<<endl;
    cout<<"\n-------------------------------------"<<endl;
    cout<<"1.Insert element into the key"<<endl;
    cout<<"2.Delete element from the number"<<endl;
    cout<<"3.Search element into the number"<<endl;
    cout<<"4.Exit"<<endl;
    cout<<"Enter your Choice: ";
    cin>>ch;
    switch(ch)
    {
    case 1:
    {
```

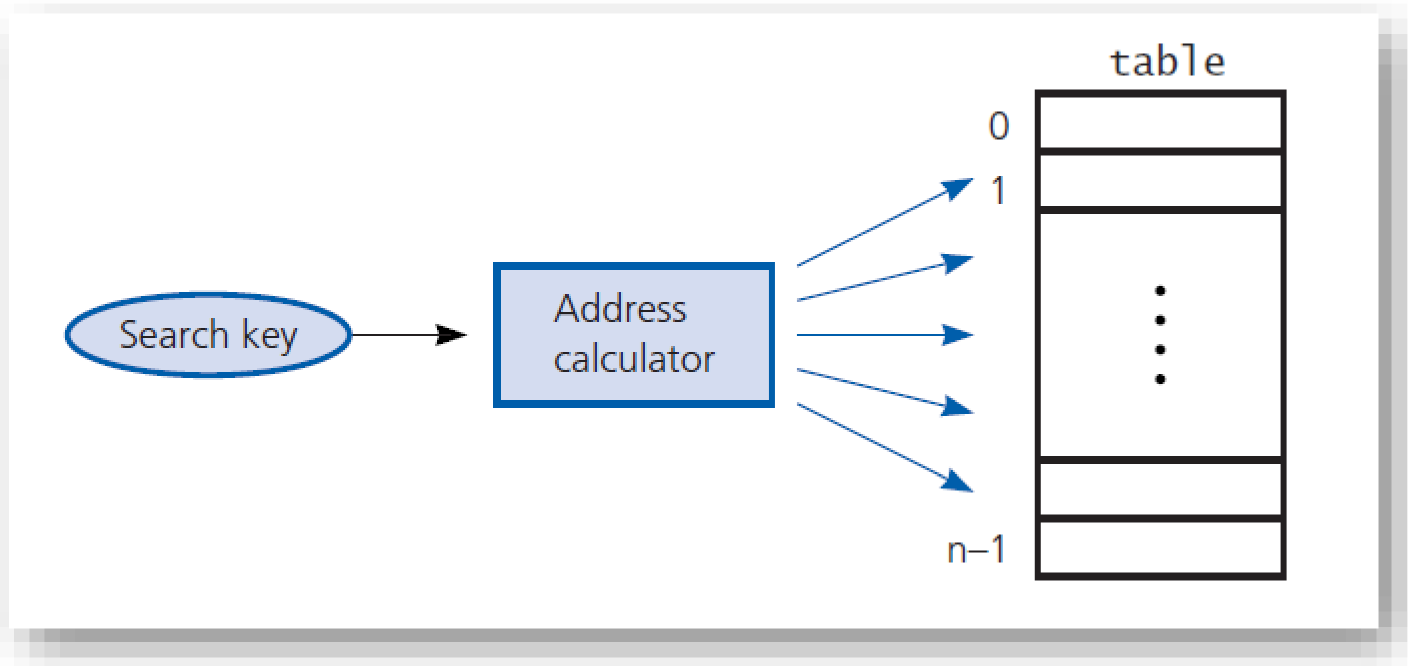
```
switch(ch)
{
case 1:
{
    string str1 = "";
    cout<<"Enter the key value: ";
    cin>>key;
    cout<<"Enter the string to be inserted: ";
    cin.ignore();
    getline(cin, str);
    INSERT(T, number(key, str));
    break;
}
case 2:
    cout<<"Enter the key of element to be deleted: ";
    cin>>key;
    x = SEARCH(T, key);
    DELETE(T, x);
    break;
case 3:
cout<<"Enter the key of element to be searched: ";
    cin>>key;
    x = SEARCH(T, key);
    if (x.key == 0)
    {
        cout<<"No element inserted at the key"<<endl;
        continue;
    }
    cout<<"Element at key "<<key<<" is-> ";
    cout<<"\""<<x.english_text<<"\""<<endl;
    break;
case 4:
    exit(1);
```

Hashing

Hashing as a Dictionary Implementation

- Situations occur for which search-tree implementations are not adequate.
- Consider a method which acts as an "address calculator" which determines an array index
  - Used for add, getValue, remove operations
- Called a hash function
  - Tells where to place item in a hash table

Hashing as a Dictionary Implementation



- Address calculator

Hashing

- is a means used to order and access elements in a list quickly by using a function of the key value to identify its location in the list.
  - the goal is O(1) time
- The function of the key value is called a hash function

Hashing

Idea:

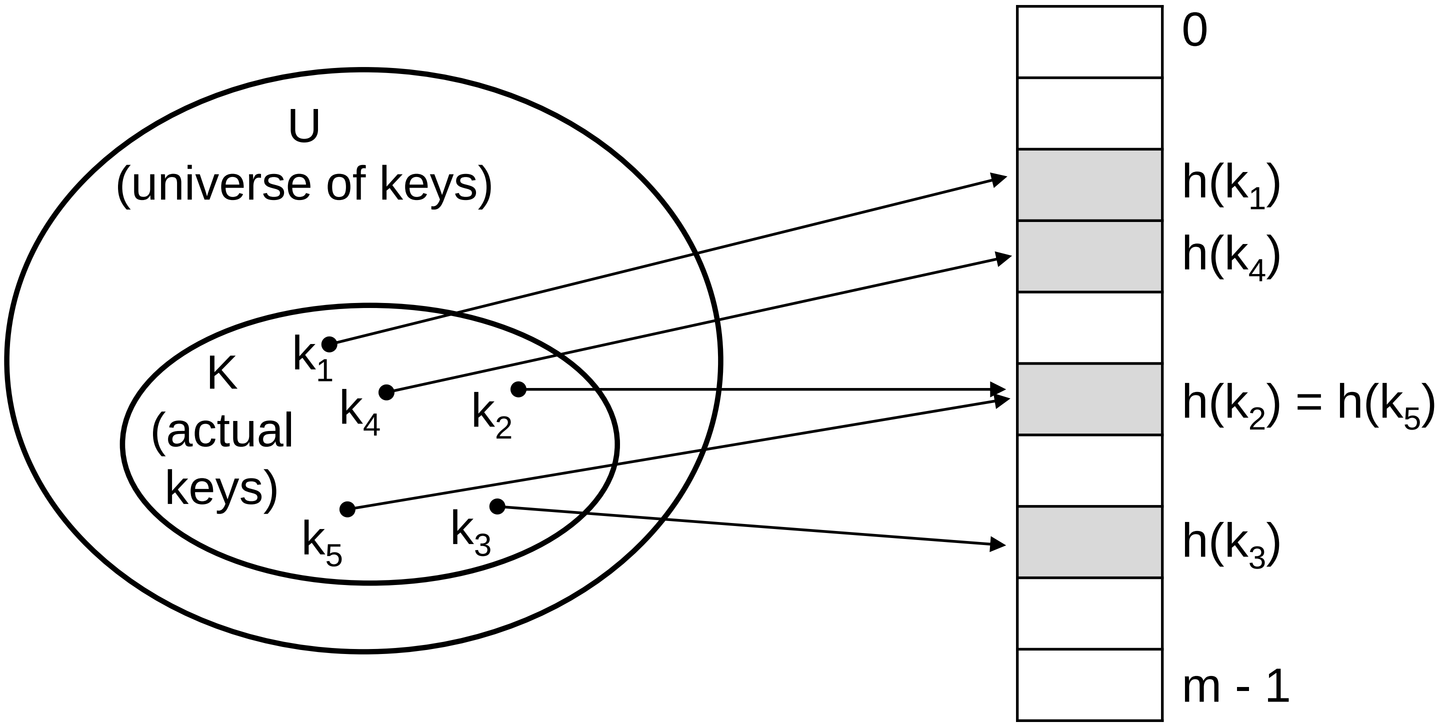
- Use a function **h** to compute the slot for each key
- Store the element in slot **h(k)**

- A **hash function** h transforms a key into an index in a hash table $T[0...m-1]$:

$$h : U \rightarrow \{0, 1, \ldots, m - 1\}$$

- We say that k **hashes** to slot $h(k)$

Hashing (cont'd)



$h : U \rightarrow \{0, 1, . . . , m - 1\}$

hash table size: **m**

Hashing (cont'd)

Example 2: Suppose that the keys are 9-digit social security numbers (SSN)

Possible hash function

$$h(ssn) = sss \text{ mod } 100 \text{ (last 2 digits of ssn)}$$

$$\text{e.g., if } ssn = 10123411 \text{ then } h(10123411) = 11)$$

Advantages of Hashing

- Reduce the range of array indices handled:

  m instead of $|U|$

  where m is the hash table size: T[0, ..., m-1]

- Storage is reduced.

- $O(1)$ search time (i.e., under assumptions).

Properties of Good Hash Functions

- **Good hash function properties**

(1)Easy to compute

(2) Approximates a random function

i.e., for every input, every output is equally likely.

(3) Minimizes the chance that similar keys hash to the same slot

i.e., strings such as pt and pts should hash to different slot.

Hashing as a Dictionary Implementation

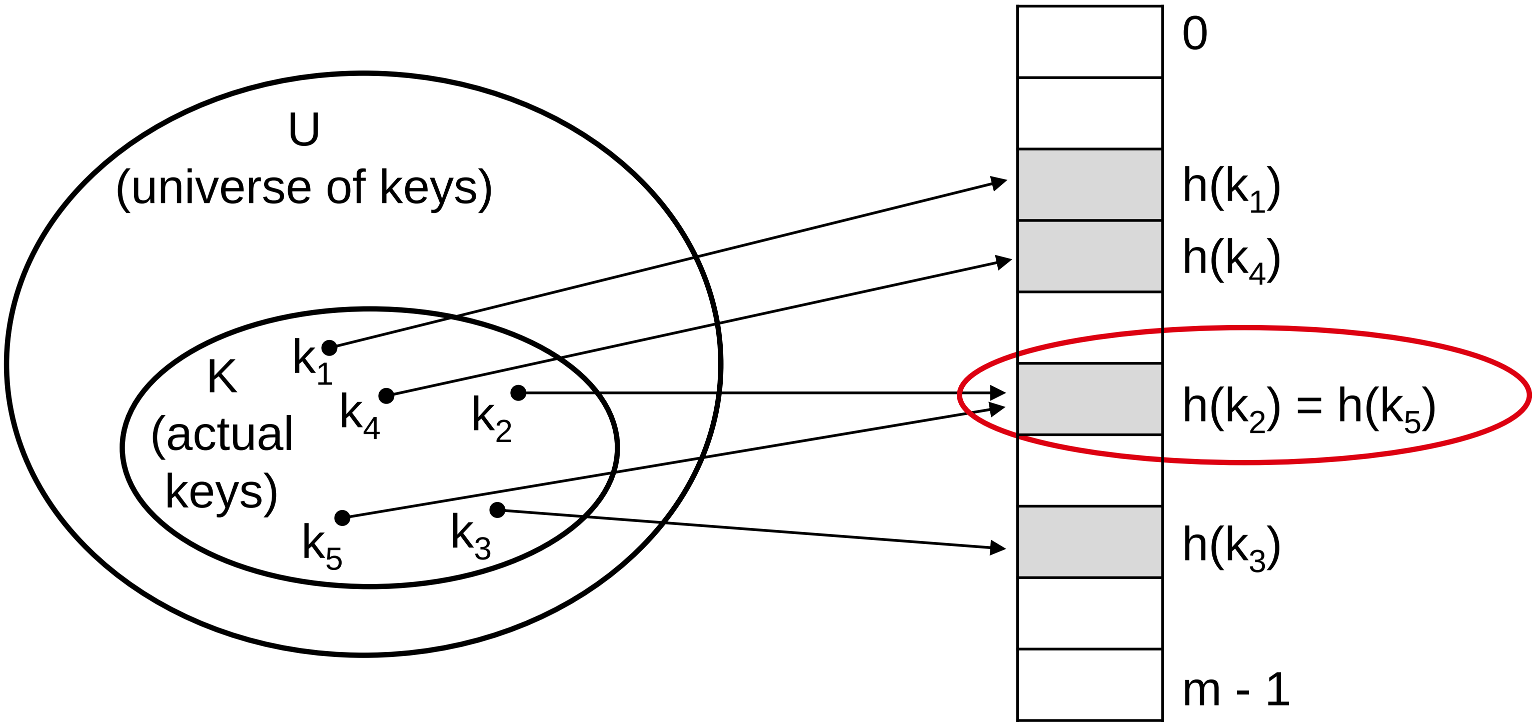
- Perfect hash function
  – Maps each search key into a unique location of the hash table
  – Possible if you know all the search keys
- Collision occurs when hash function maps more than one entry into same array location
- Hash function should
  – Be easy, fast to compute
  – Place entries evenly throughout hash table

Hash Functions

- Sufficient for hash functions to operate on integers – examples:
  - Select digits from an ID number
  - Folding – add digits, sum is the table location
  - Modulo arithmetic $h(x) = x \bmod tableSize$
  - Convert character string to an integer – use ASCII values

Collisions

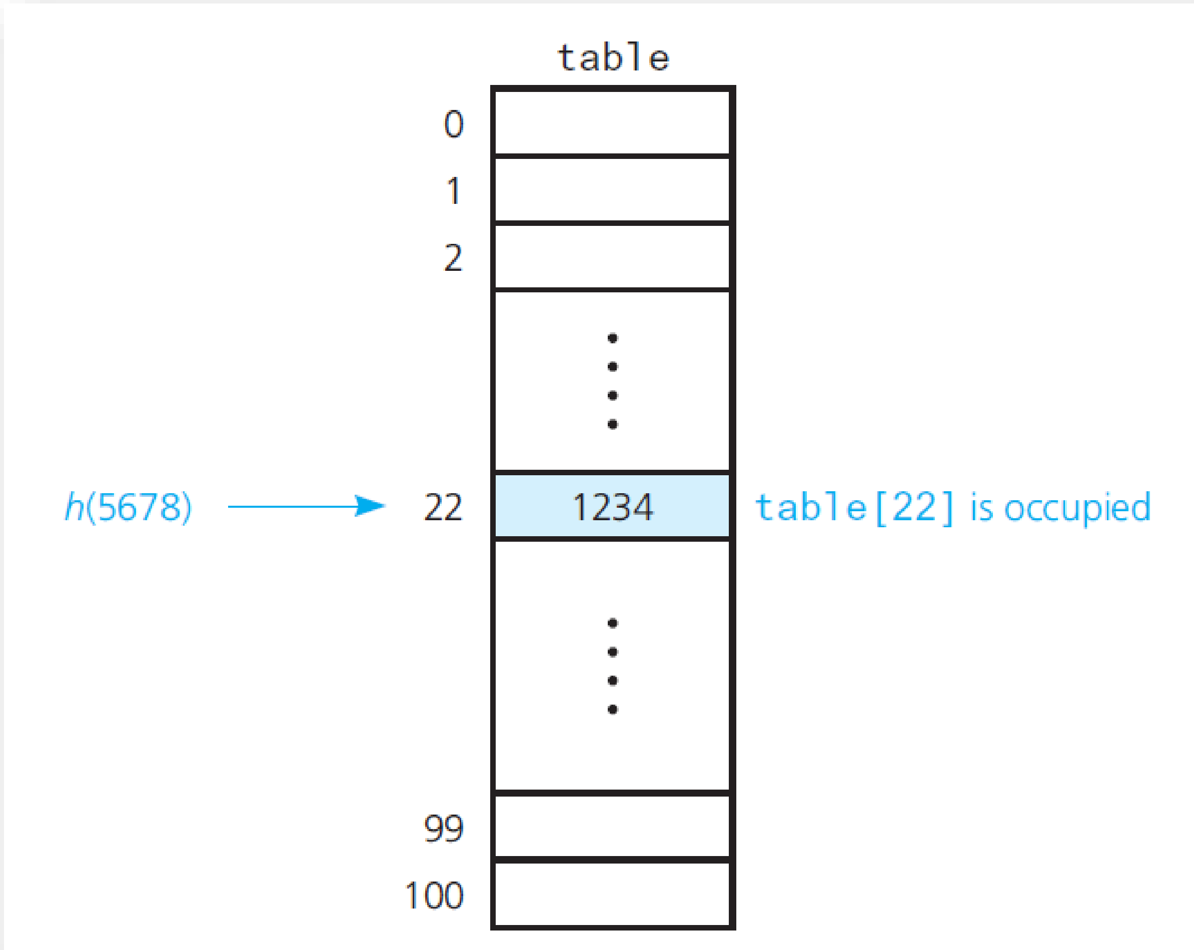
Collisions occur when $h(k_i)=h(k_j)$, $i \neq j$



Collisions (cont'd)

- For a given set K of keys:
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function!
  - If $|K| > m$, collisions will definitely happen
    - i.e., there must be at least two keys that have the same hash value
- Avoiding collisions completely might not be easy.

Resolving Collisions with Open Addressing



- A collision

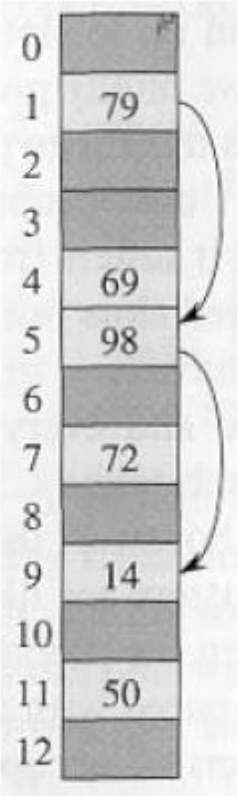
Resolving Collisions with Open Addressing

- Approach 1: Open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing
  - Increase size of hash table

Open Addressing

- Idea: store the keys in the table itself
- No need to use linked lists anymore
- Basic idea:
  - Insertion: if a slot is full, try another one, until you find an empty one.
  - Search: follow the same probe sequence.
  - Deletion: need to be careful!
- Search time depends on the length of probe sequences!

e.g., insert 14



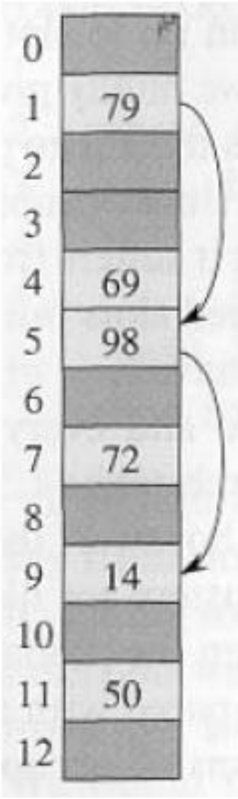
probe sequence: <1, 5, 9>

Generalize hash function notation:

- A hash function contains two arguments now:

  (i) key value, and (ii) probe number

e.g., insert 14



$$h(k,p), \quad p=0,1,\ldots,m-1$$

- Probe sequence:

  <h(k,0), h(k,1), h(k,2), ... >

- Example:

Probe sequence: <h(14,0), h(14,1), h(14,2)>=<1, 5, 9>

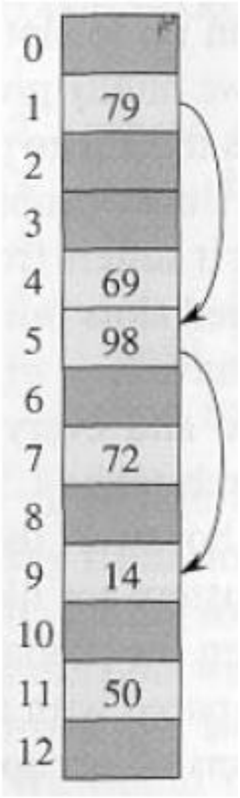
Generalize hash function notation:

– Probe sequence must be a permutation of

<0,1,...,m-1>

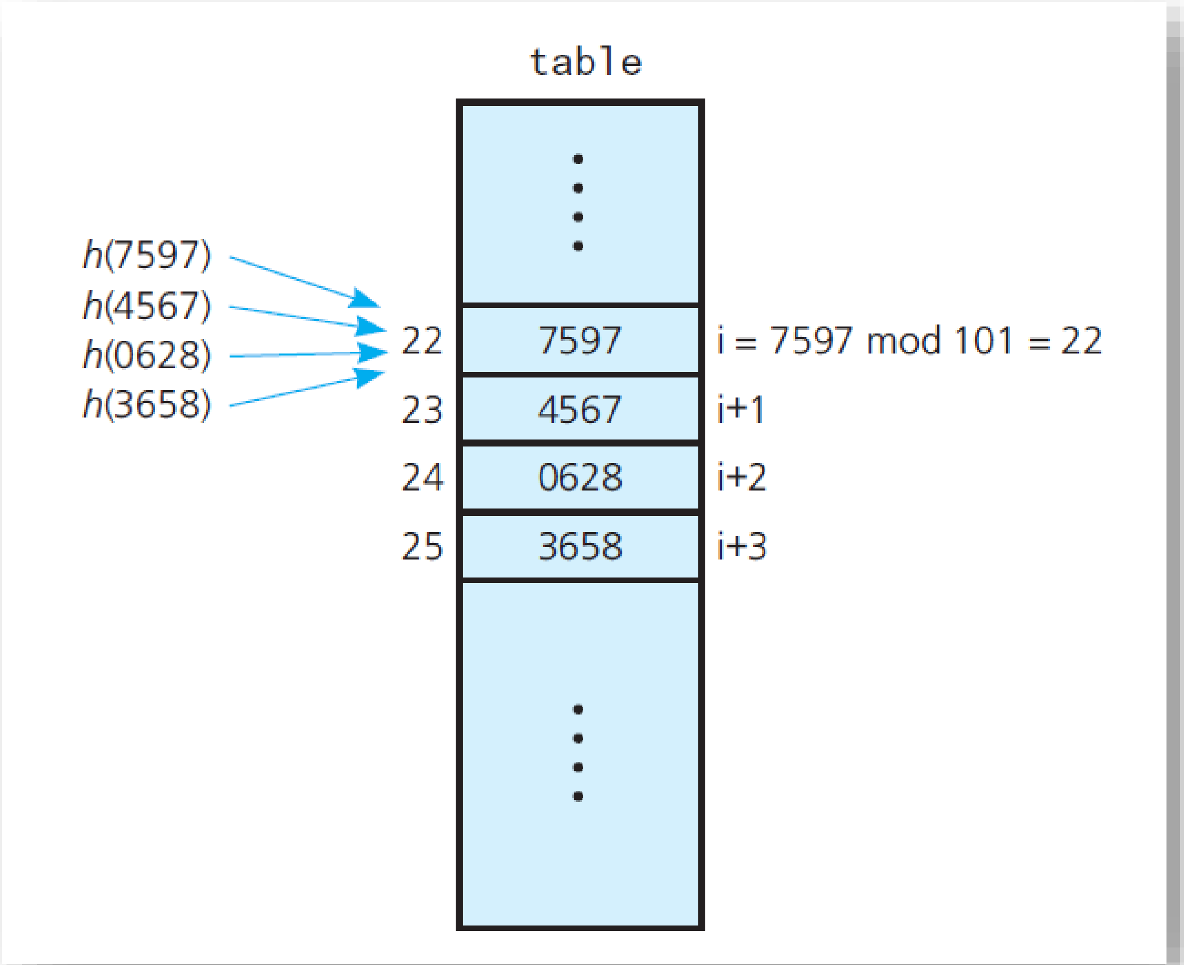
– There are m! possible permutations

e.g., insert 14



Probe sequence: <h(14,0), h(14,1), h(14,2)>=<1, 5, 9>

Resolving Collisions with Open Addressing



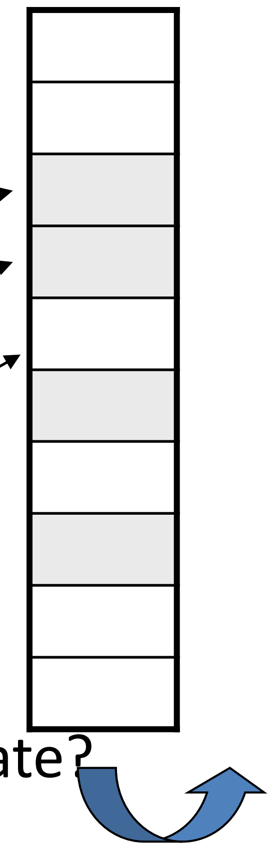
- Linear probing with h(x) = x mod 101

Linear probing: Inserting a key

- **Idea:** when there is a collision, check the next available position in the table:

$$h(k,i) = (h_1(k) + i) \bmod m$$

$$i=0,1,2,\ldots$$



- i=0: first slot probed: $h_1(k)$
- i=1: second slot probed: $h_1(k) + 1$
- i=2: third slot probed: $h_1(k)+2$, and so on

probe sequence: < $h_1(k)$, $h_1(k)+1$, $h_1(k)+2$, ...>

- How many probe sequences can linear probing generate?

m probe sequences maximum

wrap around

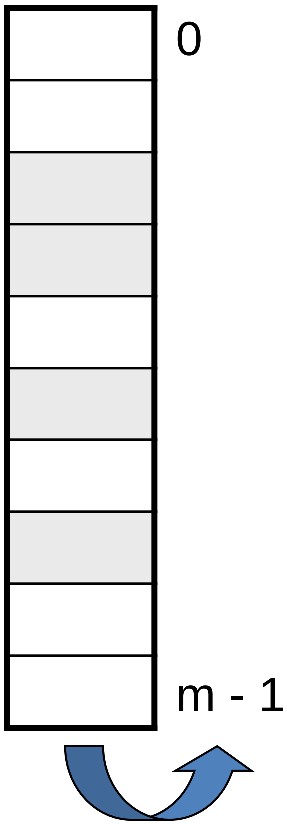
Linear probing: Searching for a key

- Given a key, generate a probe sequence using the same procedure.
- Three cases:

  (1) Position in table is occupied with an element of equal key→ FOUND

  (2) Position in table occupied with a different element → KEEP SEARCHING

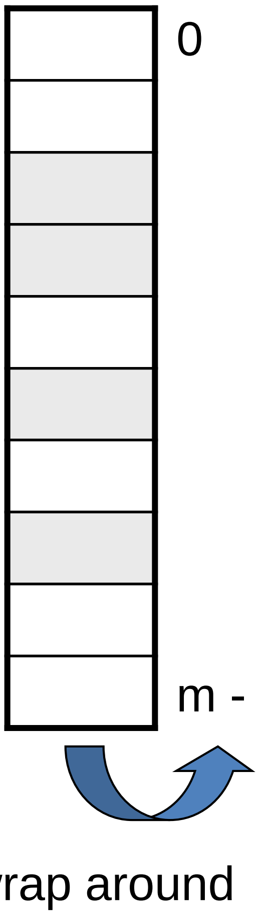
  (3) Position in table is empty→ NOT FOUND



wrap around

Linear probing: Searching for a key

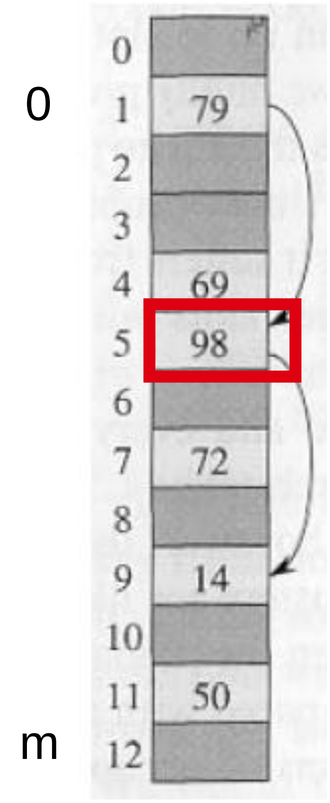
- Running time depends on the length of the probe sequences.
- Need to keep probe sequences short to ensure fast search.



Linear probing: Deleting a key

- First, find the slot containing the key to be deleted.
- Can we just mark the slot as empty?
  – It would be impossible to retrieve keys inserted after that slot was occupied!
- Solution
  – "Mark" the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion.

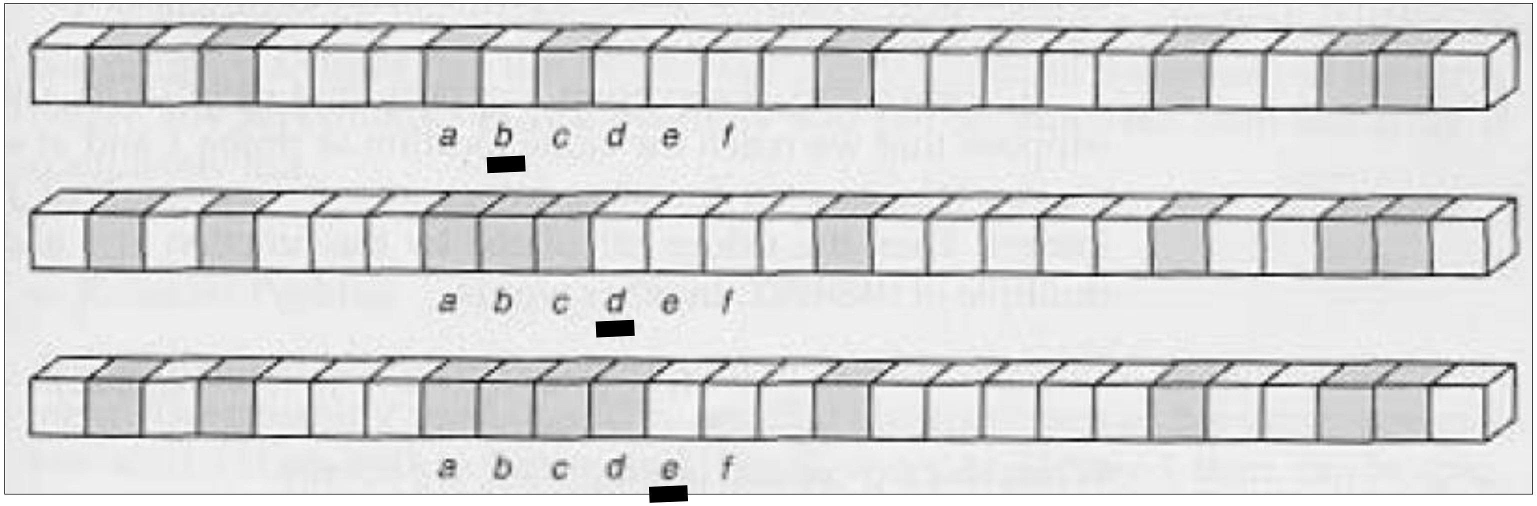
e.g., delete 98



Primary Clustering Problem

- Long chunks of occupied slots are created.
- As a result, some slots become more likely than others.
- Probe sequences increase in length. $\Rightarrow$ search time increases!!

initially, all slots have probability 1/m

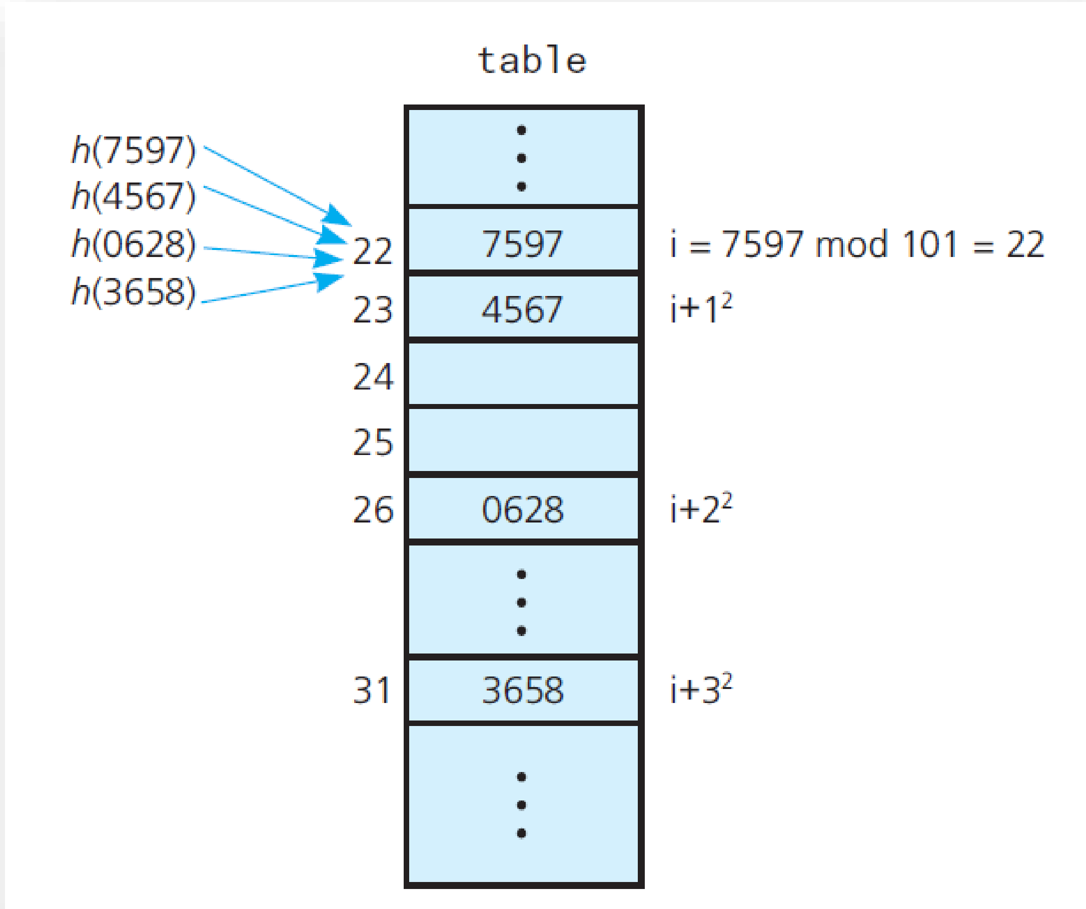


Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

Resolving Collisions with Open Addressing



- Quadratic probing with h(x) = x mod 101

Quadratic probing

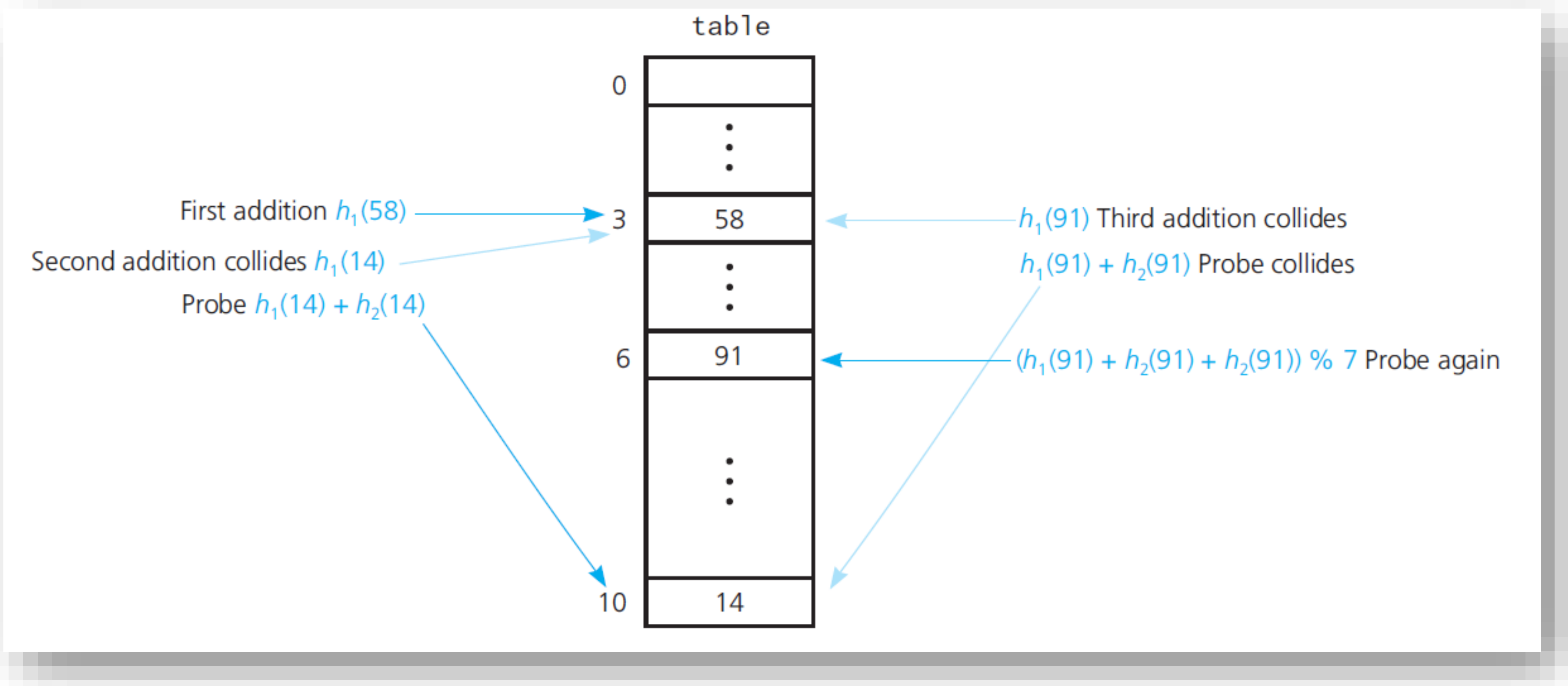
$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \text{ where } h' : U --> (0, 1, \ldots, m-1)$$

i=0,1,2,...

- Clustering is less serious but still a problem
  - *secondary clustering*
- How many probe sequences can quadratic probing generate?

m -- the initial position determines probe sequence

Resolving Collisions with Open Addressing



- Double hashing during the addition of 58, 14, and 91

Double Hashing

(1) Use one hash function to determine the first slot.

(2) Use a second hash function to determine the increment for the probe sequence:

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m, \quad i=0,1,\ldots$$

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \bmod m$, so on ...
- Advantage: handles clustering better
- Disadvantage: more time consuming
- How many probe sequences can double hashing generate?

m^2

Double Hashing: Example

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 13$

- Insert key 14:

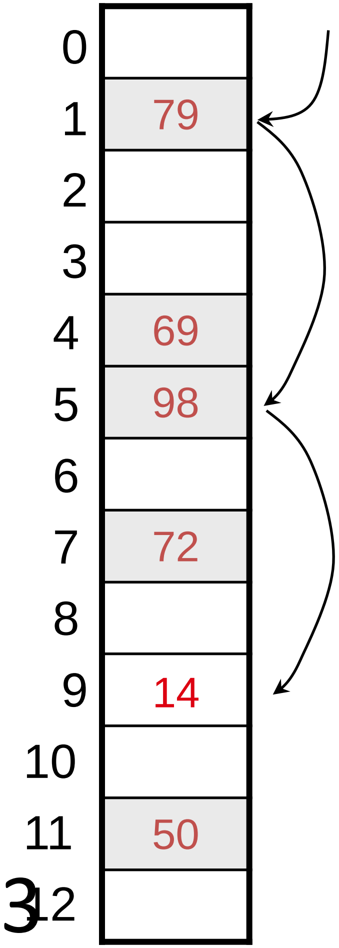
i=0: $h(14,0) = h_1(14) = 14 \bmod 13 = 1$

i=1: $h(14,1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

i=2: $h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

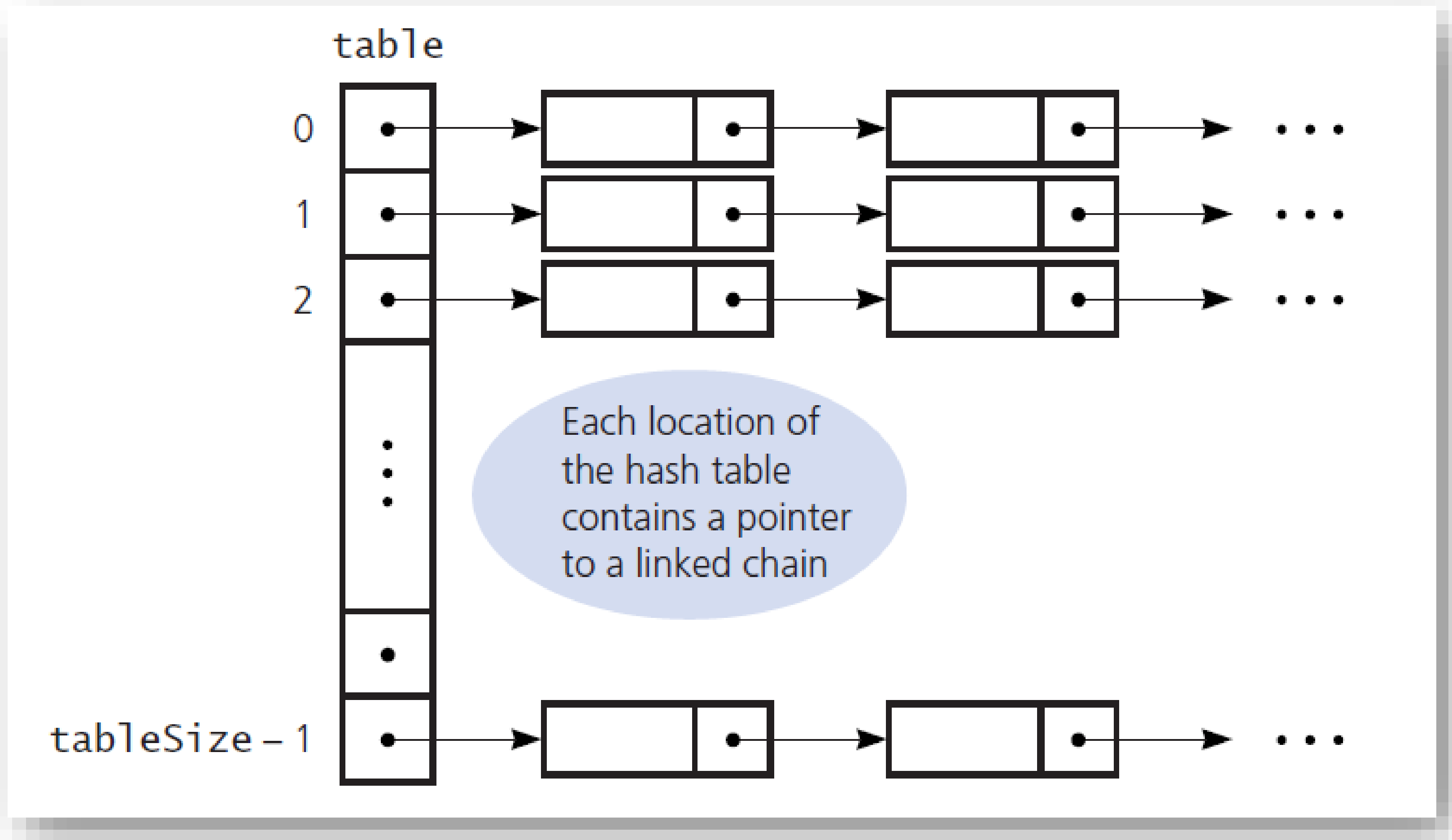
$= (1 + 8) \bmod 13 = 9$



Resolving Collisions with Open Addressing

- Approach 2: Resolving collisions by restructuring the hash table
  - Buckets
  - Separate chaining

Resolving Collisions with Open Addressing



- Separate chaining

Chaining

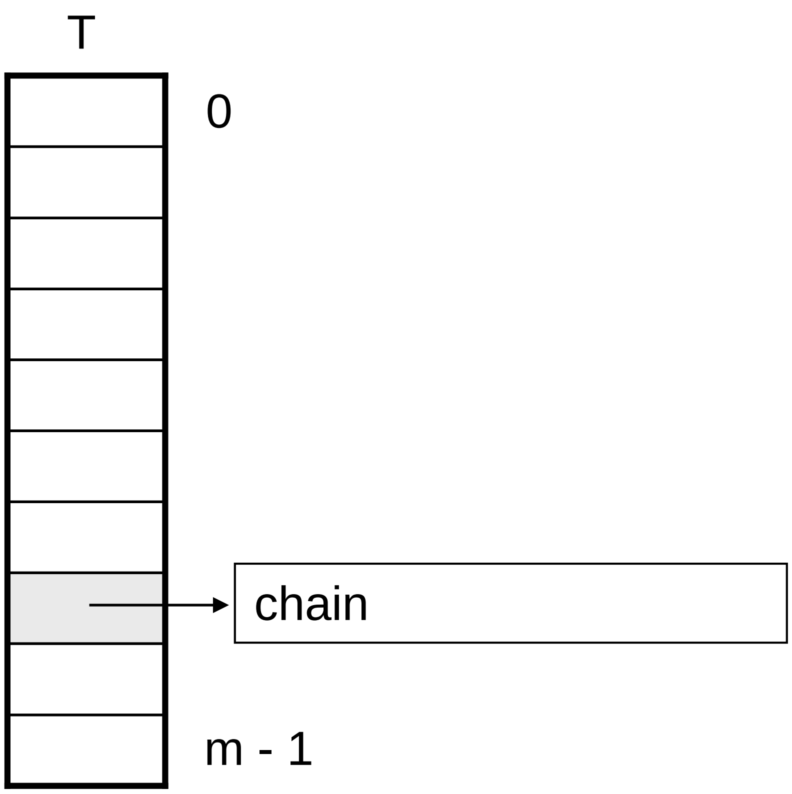
- How to choose the size of the hash table **m**?
  - Small enough to avoid wasting space
  - Large enough to avoid many collisions and keep linked-lists short
  - Typically 1/5 or 1/10 of the total number of elements
- Should we use sorted or unsorted linked lists?
  - Unsorted
    - Insert is fast
    - Can easily remove the most recently inserted elements

Analysis of Hashing with Chaining: Worst Case

- How long does it take to search for an element with a given key?

- Worst case:
  - All n keys hash to the same slot

then $O(n)$ plus time to compute the hash function



Analysis of Hashing with Chaining: Average Case

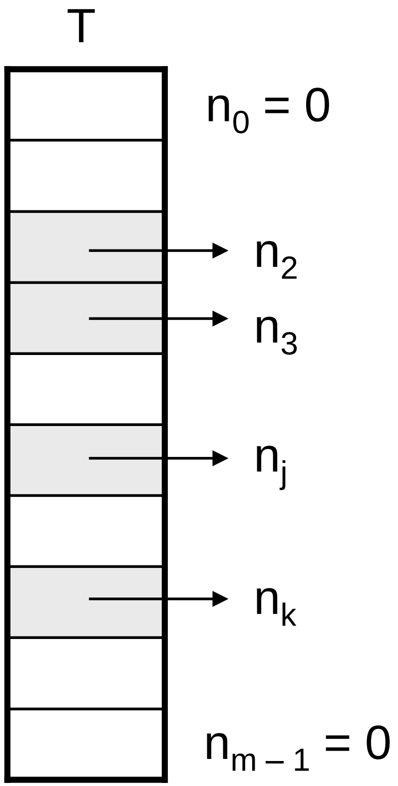
- It depends on how well the hash function distributes the n keys among the m slots
- Under the following assumptions:

  (1) **n = O(m)**

  (2) any given element is **equally likely** to hash into any of the m slots

  i.e., simple uniform hashing property

then → O(1) time plus time to compute the hash function



The Efficiency of Hashing

- Load factor measures how full a hash table is

$$\alpha = \frac{\text{Current number of table entries}}{\text{tableSize}}$$

- Unsuccessful searches
  - Generally require more time than successful
- Do not let the hash table get too full

The Efficiency of Hashing

- Linear probing – average number of comparisons

$$\frac{1}{2}\left[1 + \frac{1}{1 - \alpha}\right]$$ for a successful search, and

$$\frac{1}{2}\left[1 + \frac{1}{(1 - \alpha)^2}\right]$$ for an unsuccessful search

The Efficiency of Hashing

- Quadratic probing and double hashing – average number of comparisons

$$\frac{-\log_e(1-\alpha)}{\alpha} \qquad \text{for a successful search,}$$

$$\frac{1}{1-\alpha} \qquad \text{for an unsuccessful search}$$

The Efficiency of Hashing

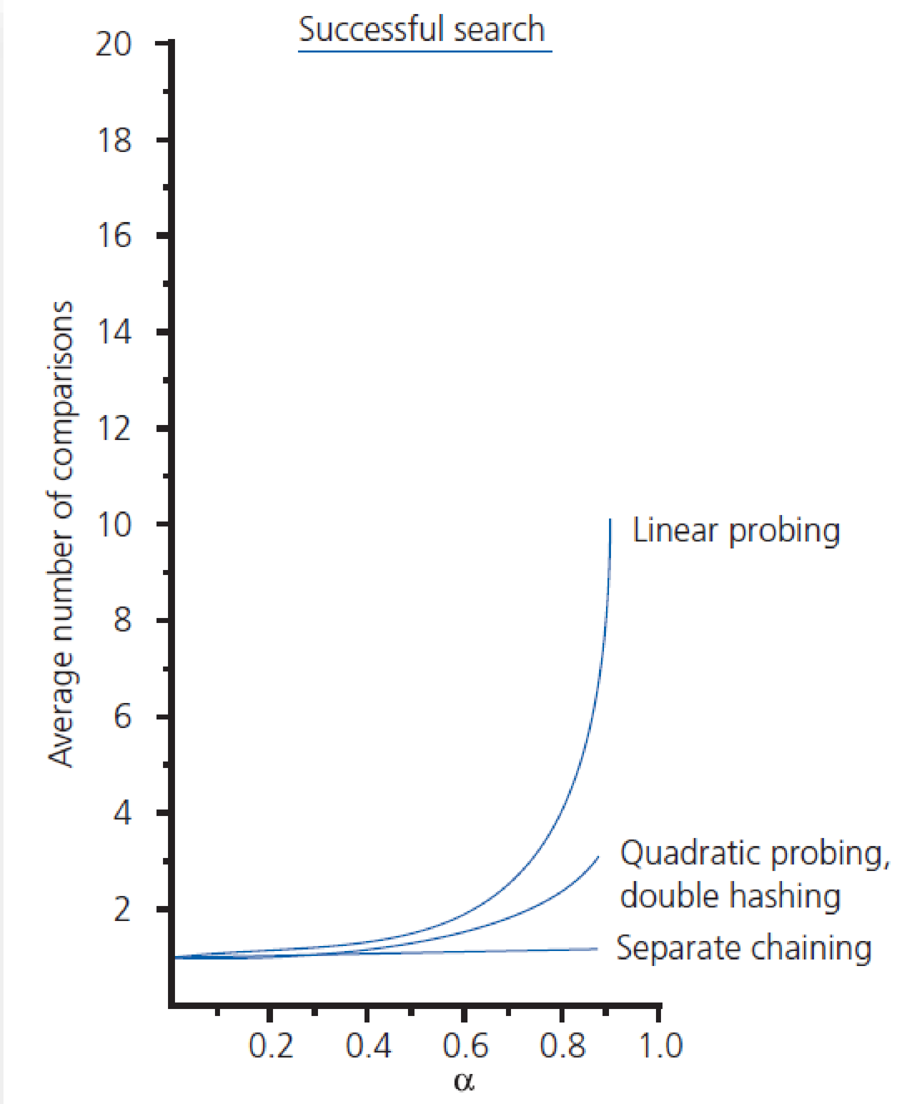
- Efficiency of the retrieval and removal operations under the separate-chaining approach

$$1 + \frac{\alpha}{2} \quad \text{for a successful search,}$$

$$\alpha \quad \text{for an unsuccessful search}$$

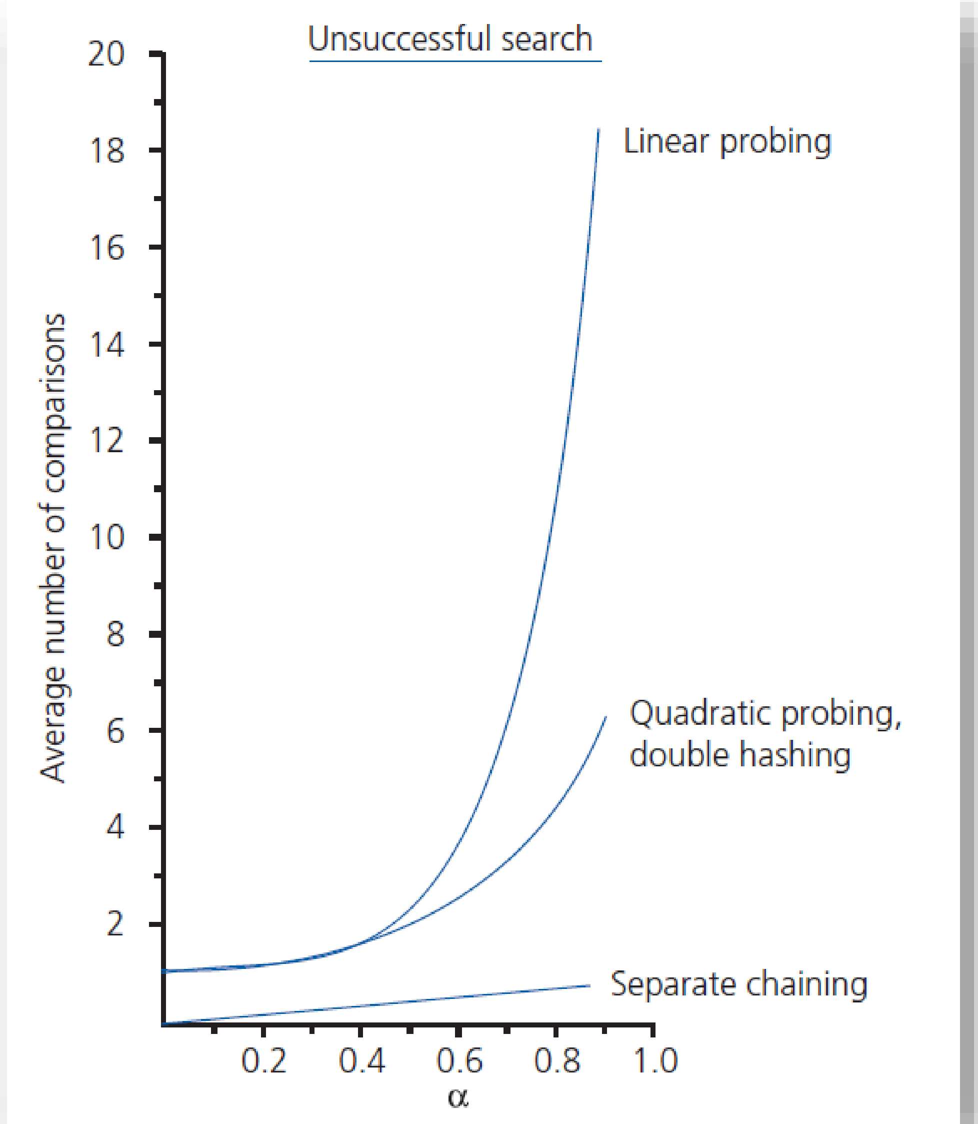
The Efficiency of Hashing

- The relative efficiency of four collision-resolution methods



The Efficiency of Hashing

- The relative efficiency of four collision-resolution methods



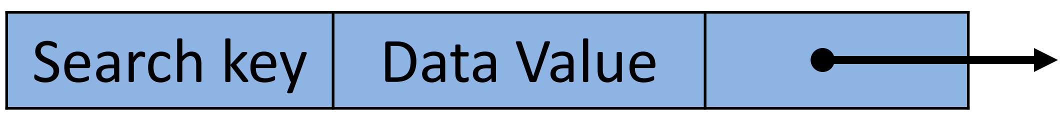
What Constitutes a Good Hash Function?

- Is hash function easy and fast to compute?
- Does hash function scatter data evenly throughout hash table?
- How well does hash function scatter random data?
- How well does hash function scatter *non-*random data?

Dictionary Traversal: An Inefficient Operation Under Hashing

- Entries hashed into table[i] and table[i+1] have no ordering relationship
- Hashing does not support well traversing a dictionary in sorted order
  - Generally better to use a search tree
- In external storage possible to see
  - Hashing implementation of getValue
  - And search-tree for ordered operations simultaneously

Using Hashing, Separate Chaining to Implement ADT Dictionary



- A dictionary entry when separate chaining is used

Using Hashing, Separate Chaining to Implement ADT Dictionary

```
/** A class of entry objects for a hashing implementation of the
    ADT dictionary.
 @file HashedEntry.h */

#ifndef HASHED_ENTRY_
#define HASHED_ENTRY_

#include "Entry.h"

template<class KeyType, class ValueType>
class HashedEntry : public Entry<KeyType, ValueType>
{
private:
   std::shared_ptr<HashedEntry<KeyType, ValueType>> nextPtr;
public:
```

- The class HashedEntry

Using Hashing, Separate Chaining to Implement ADT Dictionary

```
   std::shared_ptr<HashedEntry<KeyType, ValueType>> nextPtr;
public:
   HashedEntry();
   HashedEntry(KeyType searchKey, ValueType newValue);
   HashedEntry(KeyType searchKey, ValueType newValue,
               std::shared_ptr<HashedEntry<KeyType, ValueType>> nextEntryPtr);

   void setNext(std::shared_ptr<HashedEntry<KeyType, ValueType>>
                                       nextEntryPtr nextEntryPtr);
   auto getNext() const;
}; // end HashedEntry

#include "HashedEntry.cpp"
#endif
```

- The class HashedEntry