CS302 - Data Structures
using C++

Topic: Dictionaries and their Implementations

Kostas Alexis

AUTONOMOUS ROBOTS LAB

The ADT Dictionary

- A collection of data about certain cities (slide to be referenced later)

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

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

The ADT Dictionary

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

The ADT Dictionary

- UML diagram for the class of dictionaries

UML Notation

Dictionary

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

Possible Implementations

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

Possible Implementations

- A dictionary entry

Possible Implementations

- A header file for a **class of dictionary entries**

```
// An interface for the ADT dictionary

#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 this dictionary
    virtual int getNumberOfEntries() const = 0;
```

Possible Implementations

- A header file for a **class of dictionary entries**

```
// 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;
```

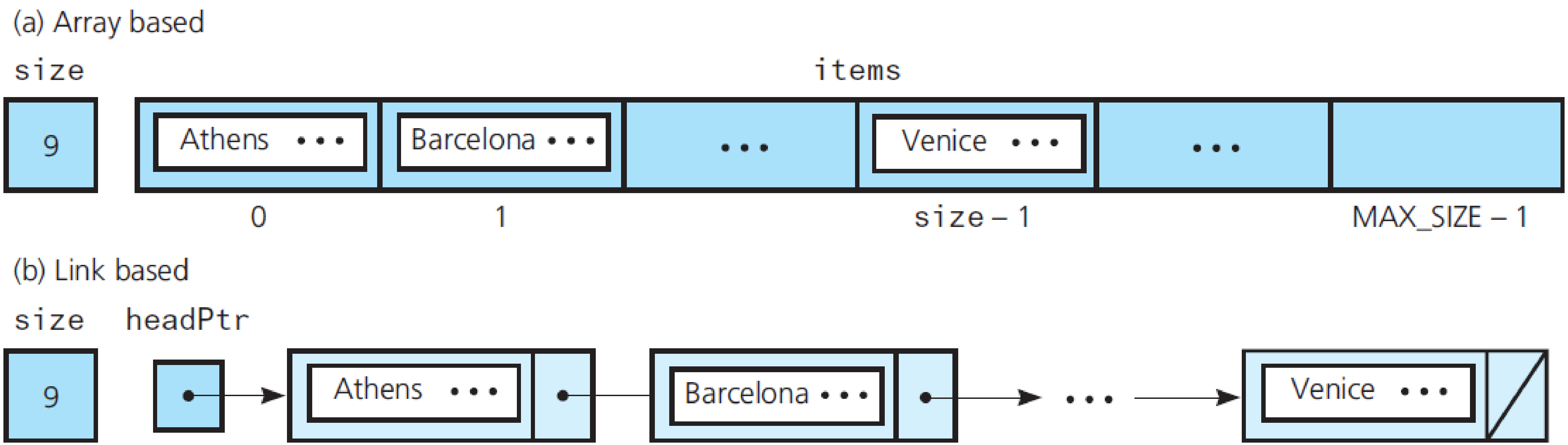
Possible Implementations

- A header file for a **class of dictionary entries**

```
    // 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;

    // 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;

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

Possible Implementations

- Data members for two sorted linear implementations of the ADT dictionary for the considered table with cities.



Possible Implementations

- A header file for a **class of dictionary entries**

```
// A clas 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);
```

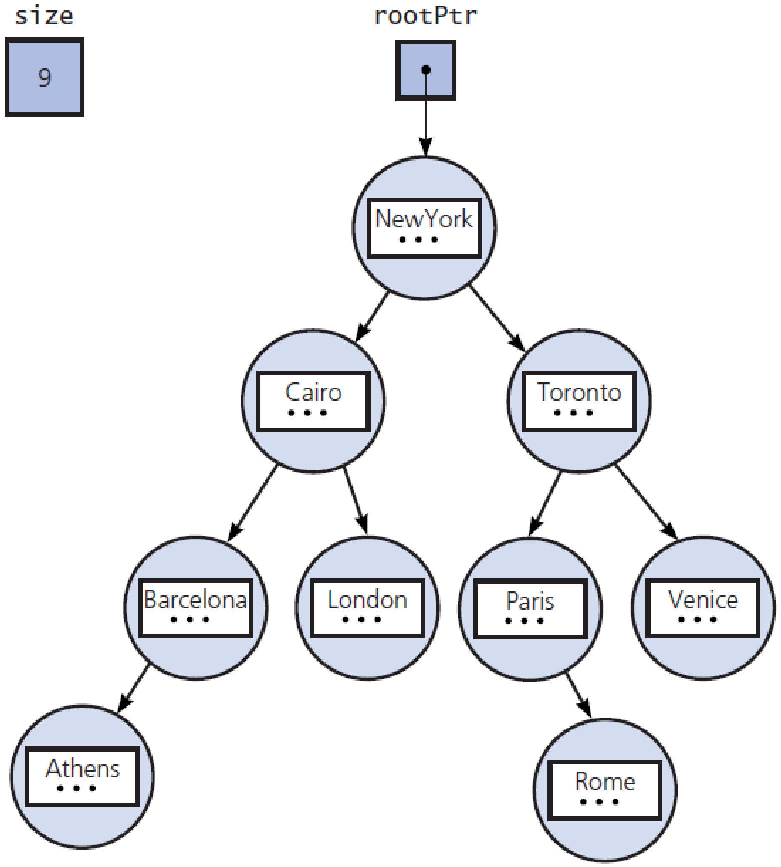
Possible Implementations

- A header file for a **class of dictionary entries**

```
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
```

Possible Implementations

- The data members for a binary search tree implementation of the ADT dictionary for the data in the considered table with cities.



Sorted Array-based Implementation of ADT Dictionary

- A header file for the class **ArrayDictionary**

```
/** An array-based implementation of the ADT dictionary
 that organizes its entries in sorted search-key order.
 Search keys in the dictionary are unique.
 @file ArrayDictionary.h */

#ifndef ARRAY_DICTIONARY_
#define ARRAY_DICTIONARY_

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

template<class KeyType, class ValueType>
class ArrayDictionary : public DictionaryInterface<KeyType, ValueType>
{
private:
   static const int DEFAULT_CAPACITY = 21;                // Small capacity to test for a full dictionary
   std::unique_ptr<Entry<KeyType, ValueType>[]> entries; // Array of dictionary entries
   int entryCount;                                        // Current count of dictionary entries
   int maxEntries;                                        // Maximum capacity of the dictionary

   void destroyDictionary();
   int findEntryIndex(int firstIndex, int lastIndex, const KeyType& searchKey) const;
```

Sorted Array-based Implementation of ADT Dictionary

- A header file for the class **ArrayDictionary**

```
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
```

Sorted Array-based Implementation of ADT Dictionary

- A header file for the class **TreeDictionary**

```
/** A binary search tree implementation of the ADT dictionary
 that organizes its entries 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>
{
```

Sorted Array-based Implementation of ADT Dictionary

- A header file for the class **TreeDictionary**

```
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
```

Sorted Array-based Implementation of ADT Dictionary

- Method **add** which prevents duplicate keys.

```
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
```

Selecting an Implementation

- Linear Implementations
  - Perspective
  - Efficiency
  - Motivation

Selecting an Implementation

- Linear Implementations
  - Perspective
  - Efficiency
  - Motivation
- Consider
  - What operations are needed
  - How often each operation is required

Three Scenarios

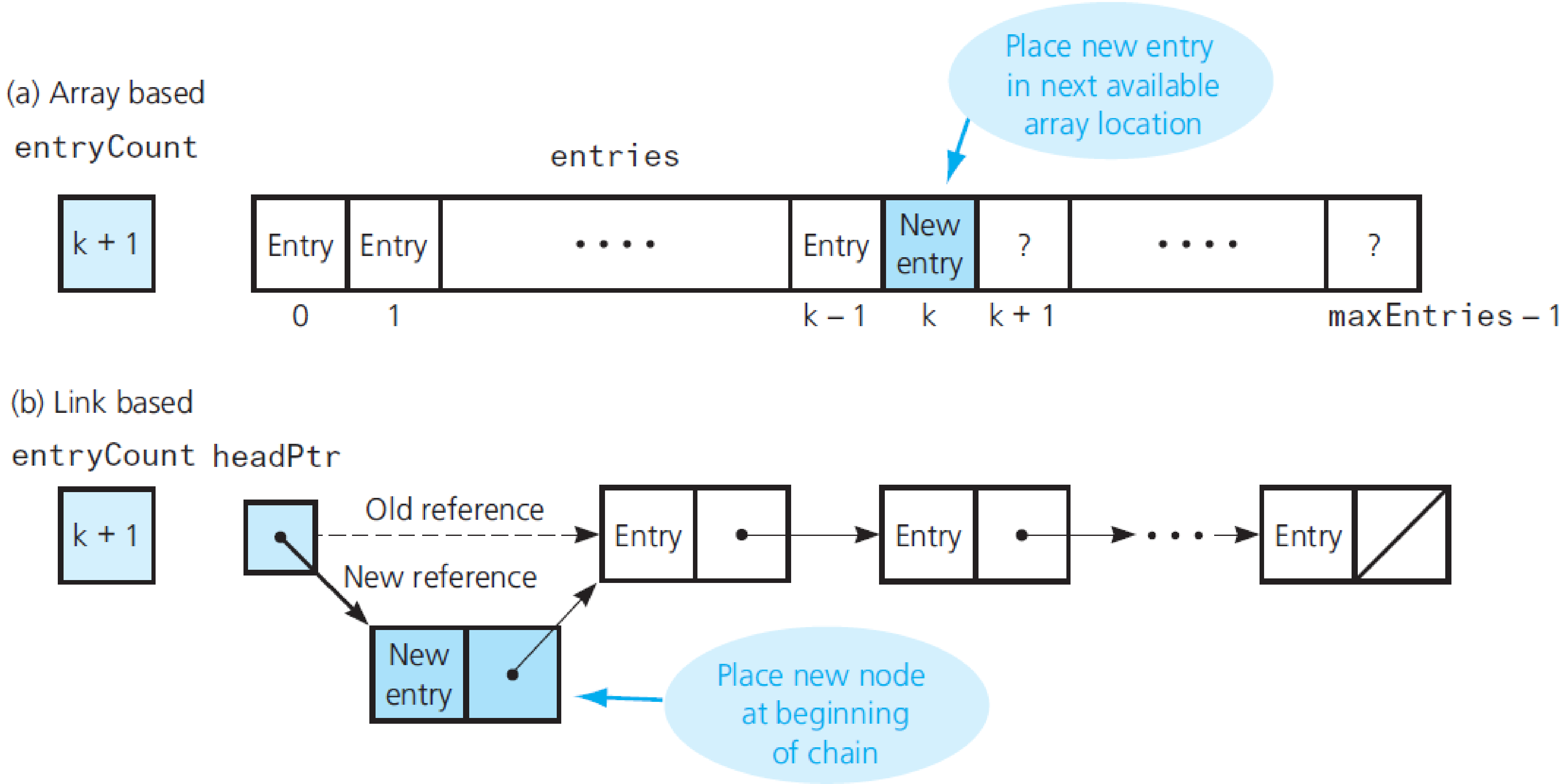
- Scenario A: Addition and traversal in no particular order
- Scenario B: Retrieval
- Scenario C: Addition, removal, retrieval, traversal in sorted order

Three Scenarios

- Scenario A: Addition and traversal in no particular order
  - Unsorted order is efficient
  - Array-based versus pointer-based
- Scenario B: Retrieval
  - Sorted array-based can use binary search
  - Binary search impractical for link-based
  - Max size of dictionary affects choice
- Scenario 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 tasks, link-based best for addition/removal

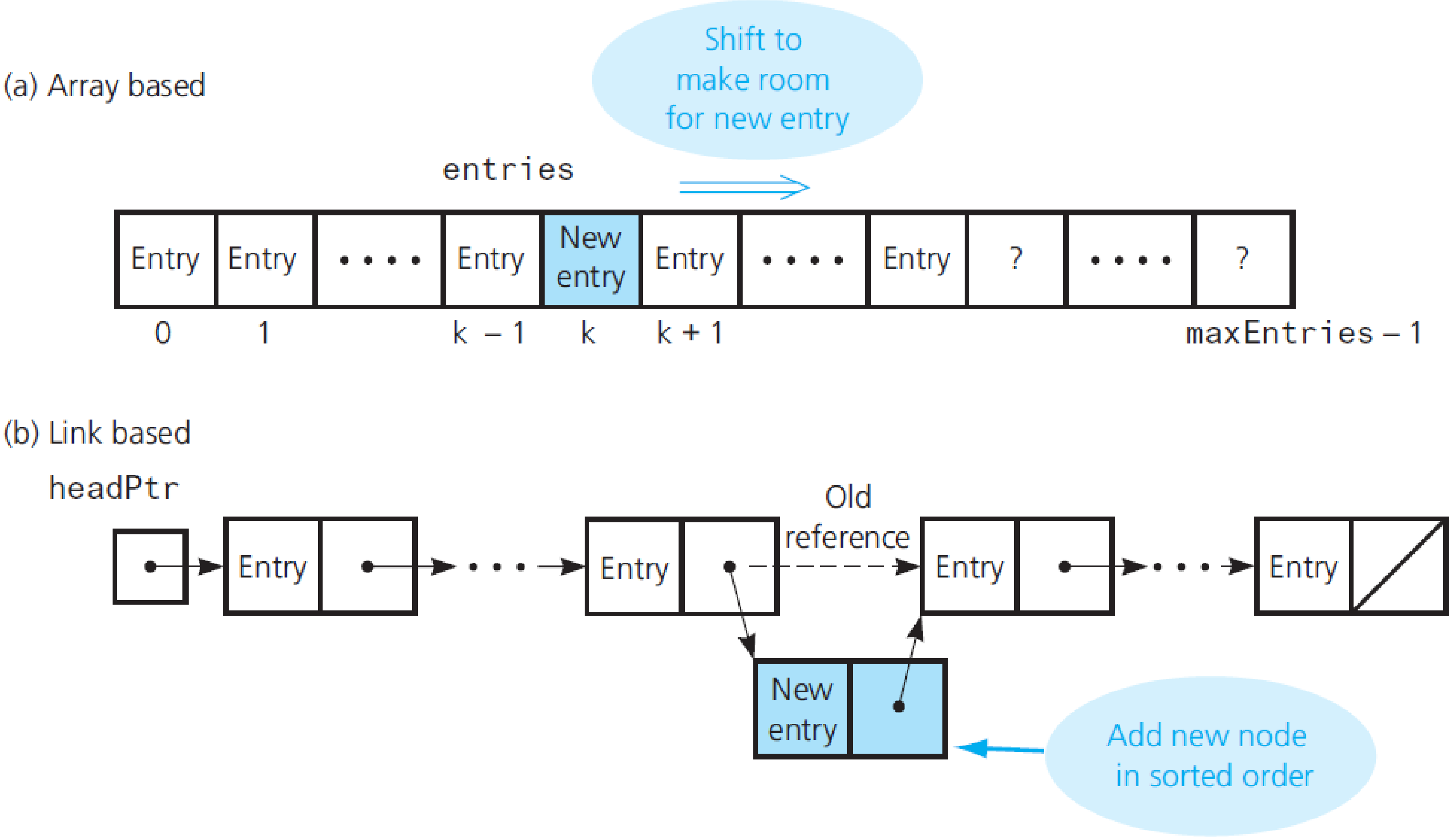
Three Scenarios

- Addition for unsorted linear implementations



Three Scenarios

- Addition for sorted linear implementations



Three Scenarios

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

Blank	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)

Hashing as a Dictionary Implementation

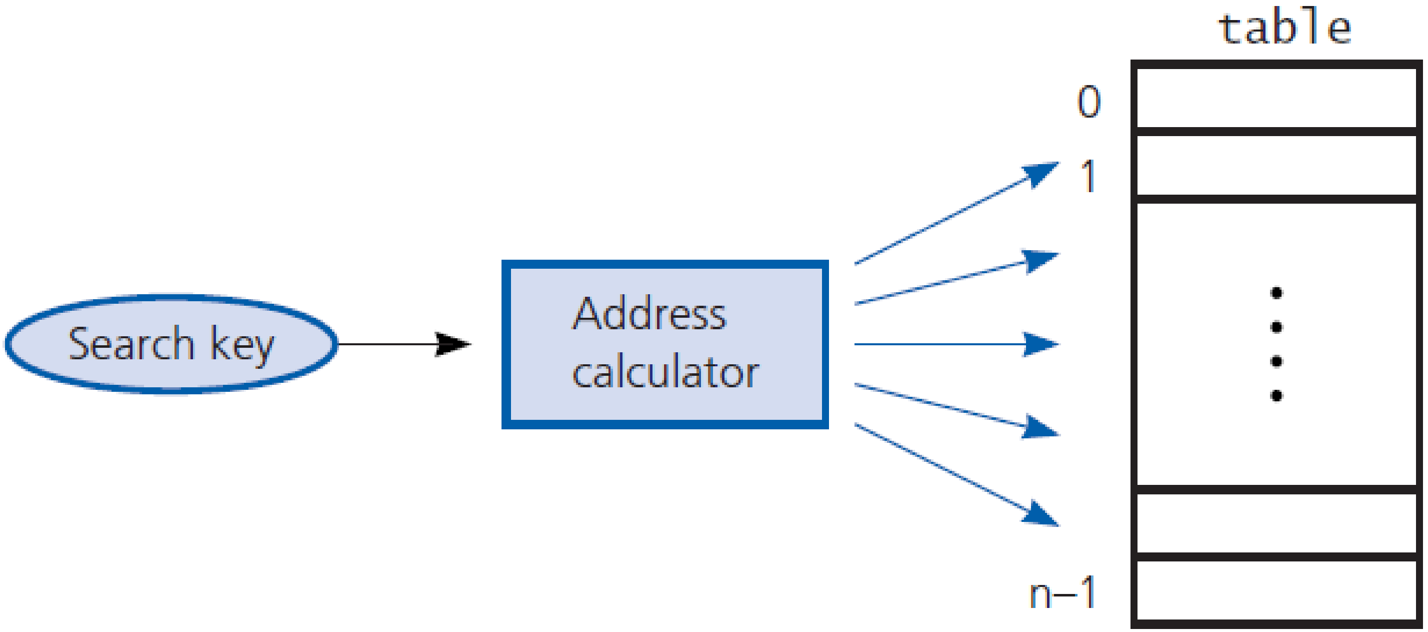
- Situations occur for which search-tree implementations are not adequate.
- Consider a method which acts as an "address calculator" which determines an array index
  - Used for add, getValue, remove operations

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 as a Dictionary Implementation

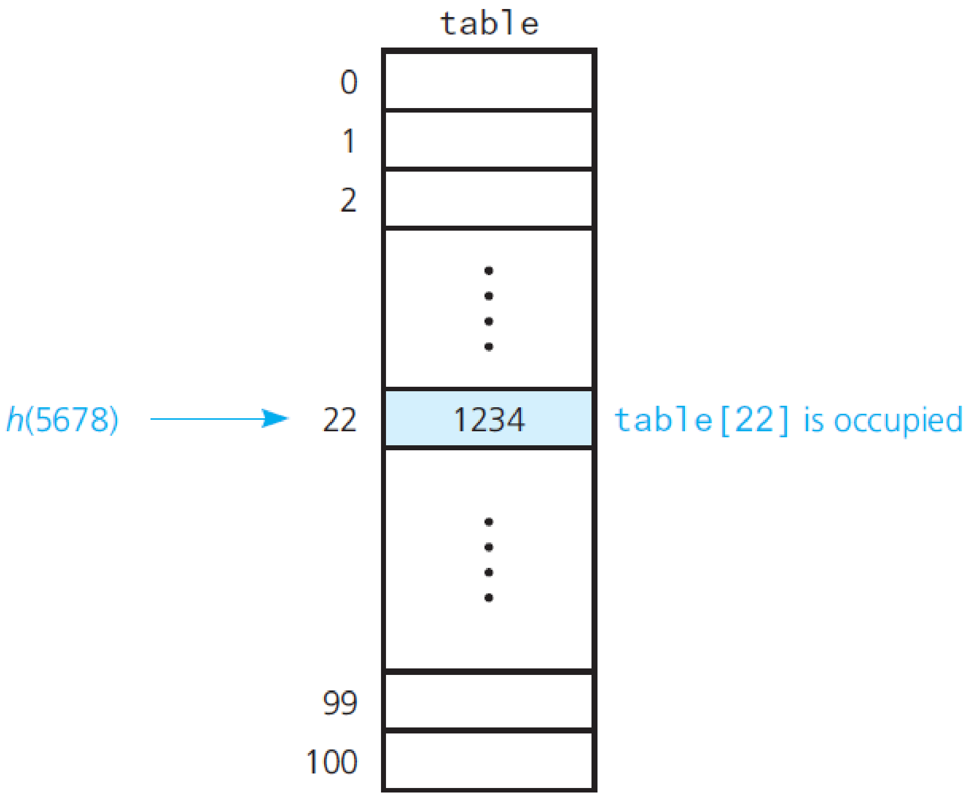
- Perfect hash function
  - Maps each search key into a unique location of the hash table
  - Possible if you know all the search keys
- Collision occurs when hash function maps more than one entry into the same array location
- Hash function should
  - Be easy, fast to compute
  - Place entries evenly 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
  - Module arithmetic h(x) = x mod tableSize
  - Convert character string to an integer – use ASCII values

Resolving Collisions with Open Addressing

- A collision

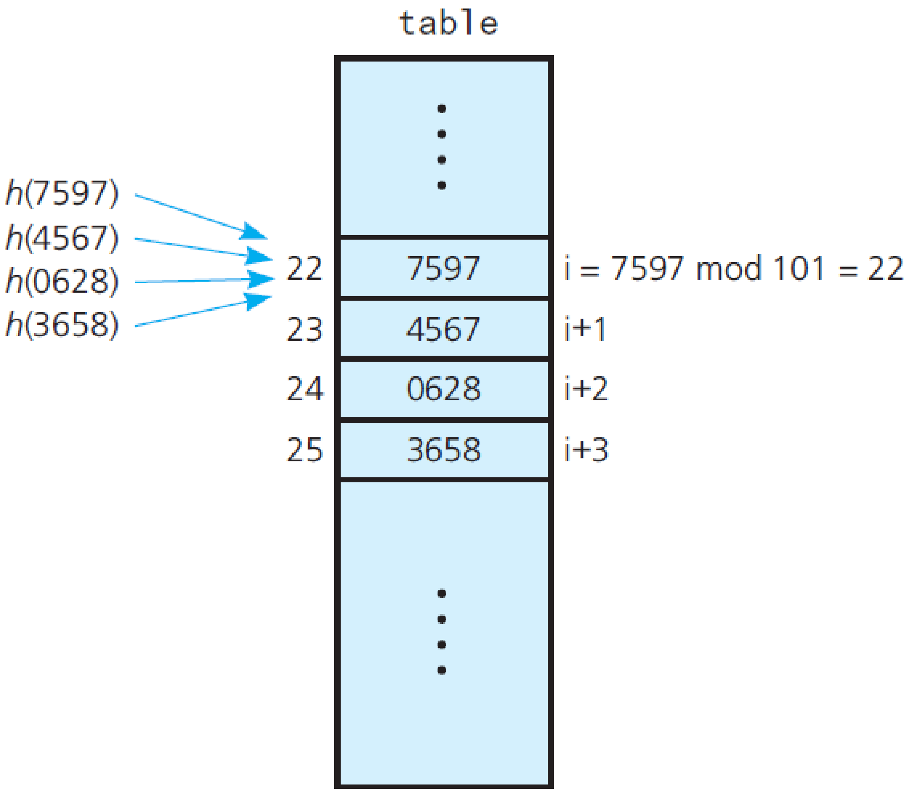


Resolving Collisions with Open Addressing

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

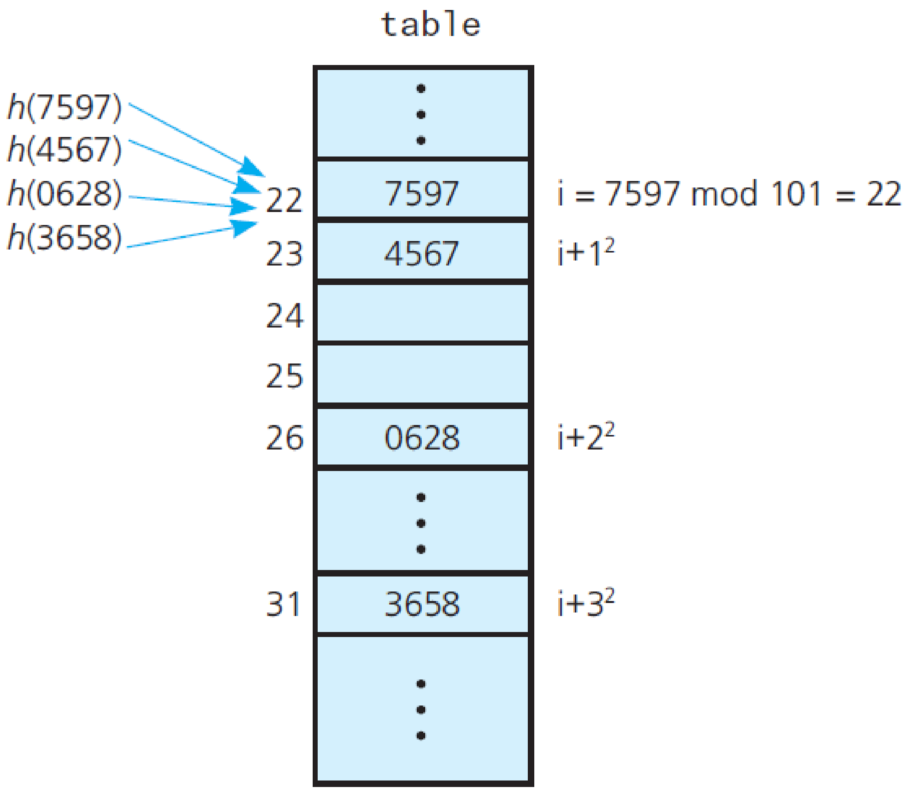
Resolving Collisions with Open Addressing

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



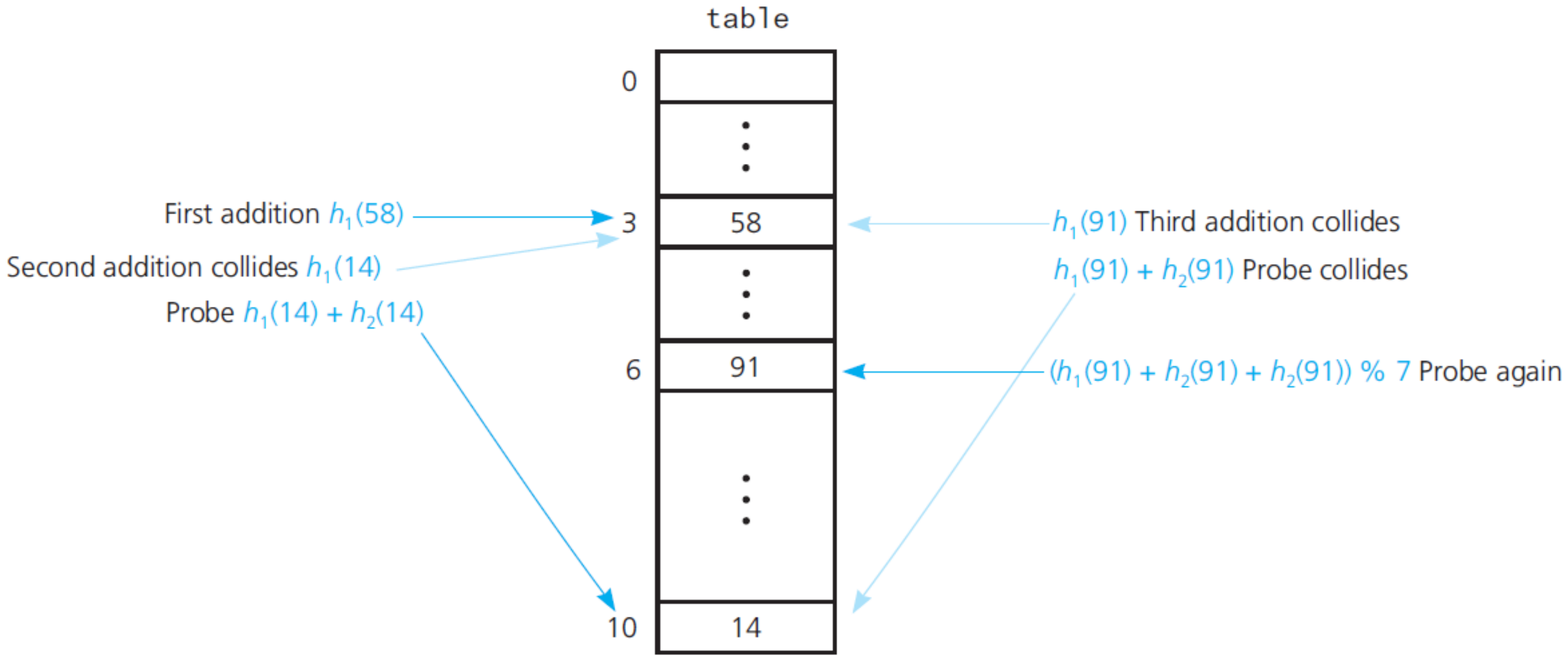
Resolving Collisions with Open Addressing

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



Resolving Collisions with Open Addressing

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

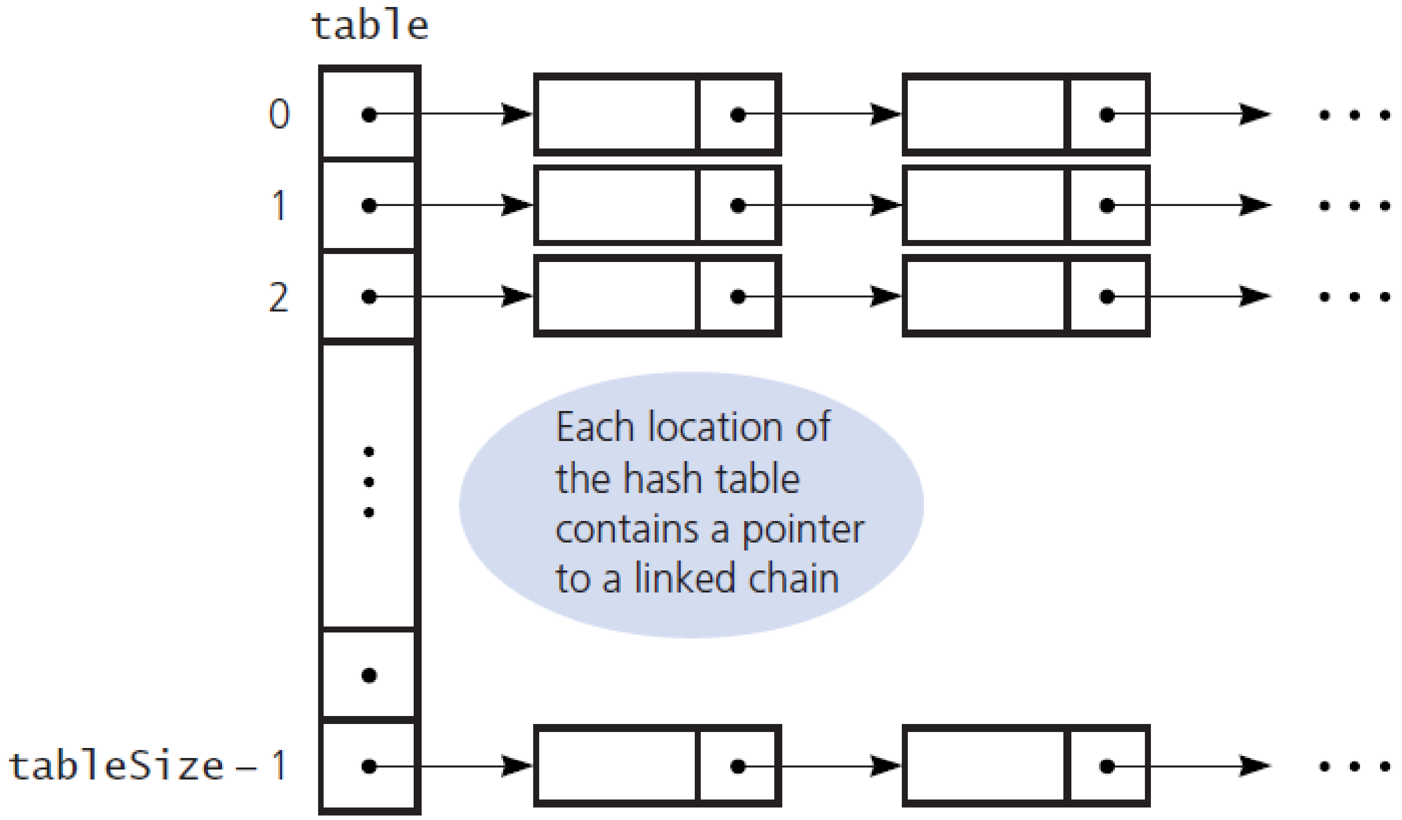


Resolving Collisions with Open Addressing

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

Resolving Collisions with Open Addressing

- Separate chaining



The Efficiency of Hashing

- Load factor measures how full a hash table is

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

- Unsuccessful searchers
  - 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

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

For an unsuccessful search

The Efficiency of Hashing

- Quadratic probing and double hashing – average number of comparisons

$$\frac{-\log_e(1-\alpha)}{\alpha}$$ For a successful search

$$\frac{1}{1-\alpha}$$ For an unsuccessful search

The Efficiency of Hashing

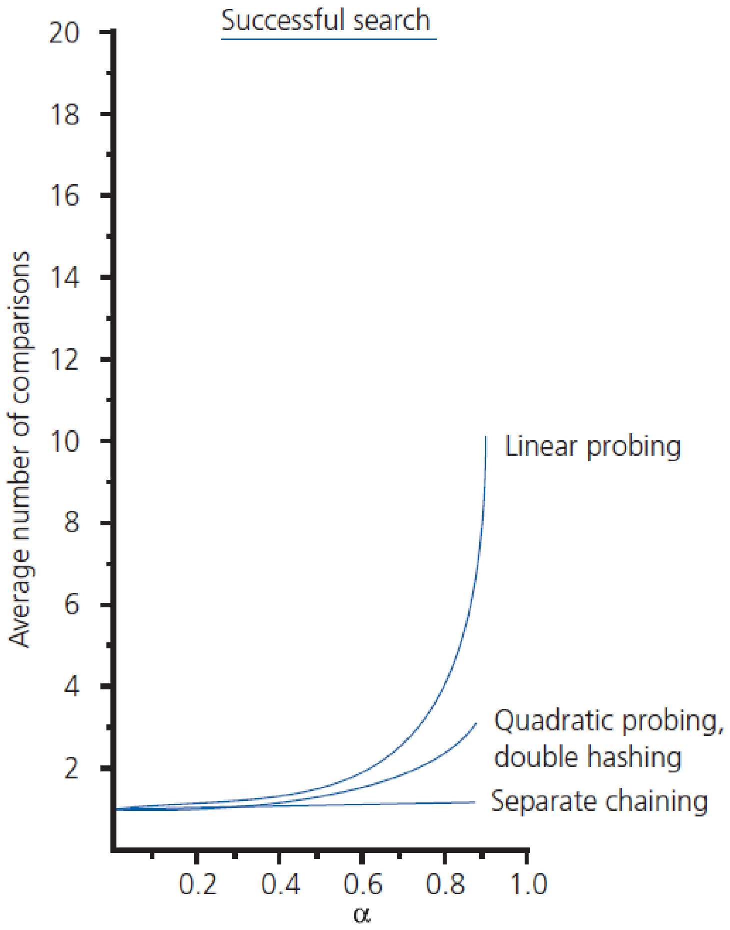
- Efficiency of the retrieval and removal operations under the separate-chaining approach

$$1 + \frac{\alpha}{2}$$ **For a successful search**

$$\alpha$$ **For an unsuccessful search**

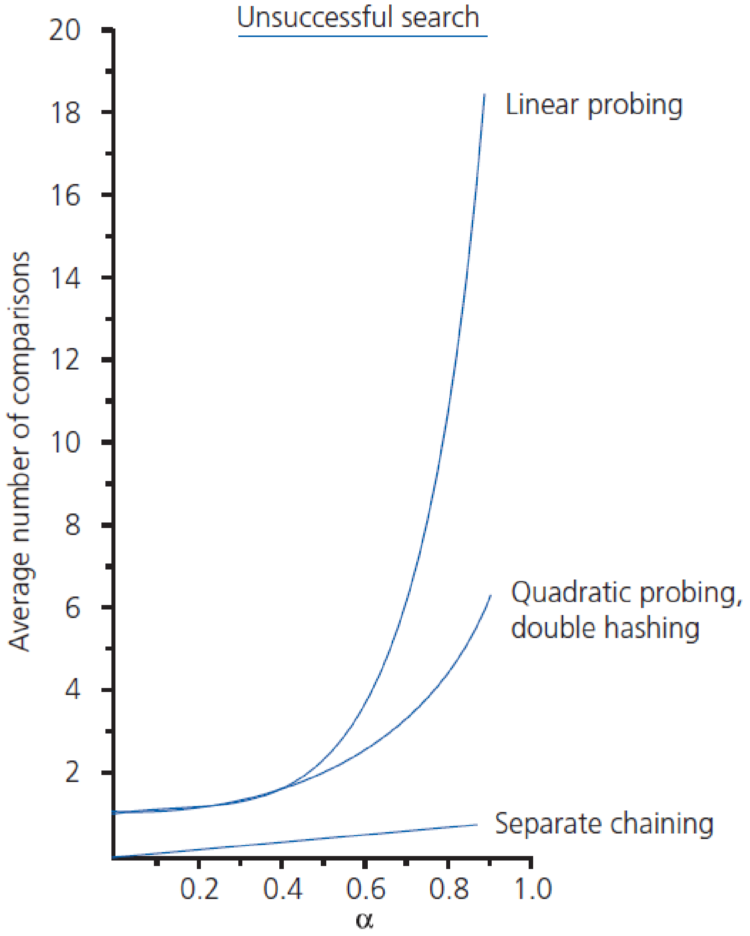
The Efficiency of Hashing

- 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 the hash function easy and fast to compute?

What Constitutes a Good Hash Function?

- Is the hash function easy and fast to compute?
- Does the hash function scatter data evenly throughout the hash table?

What Constitutes a Good Hash Function?

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

What Constitutes a Good Hash Function?

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

- A dictionary entry when separate chaining is used

Using Hashing, Separate Chaining to Implement the ADT Dictionary

- A header file for the class **HashedEntry**

```
/** 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;
```

Sorted Array-based Implementation of ADT Dictionary

- A header file for the class **HashedEntry**

```
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
```

Thank you