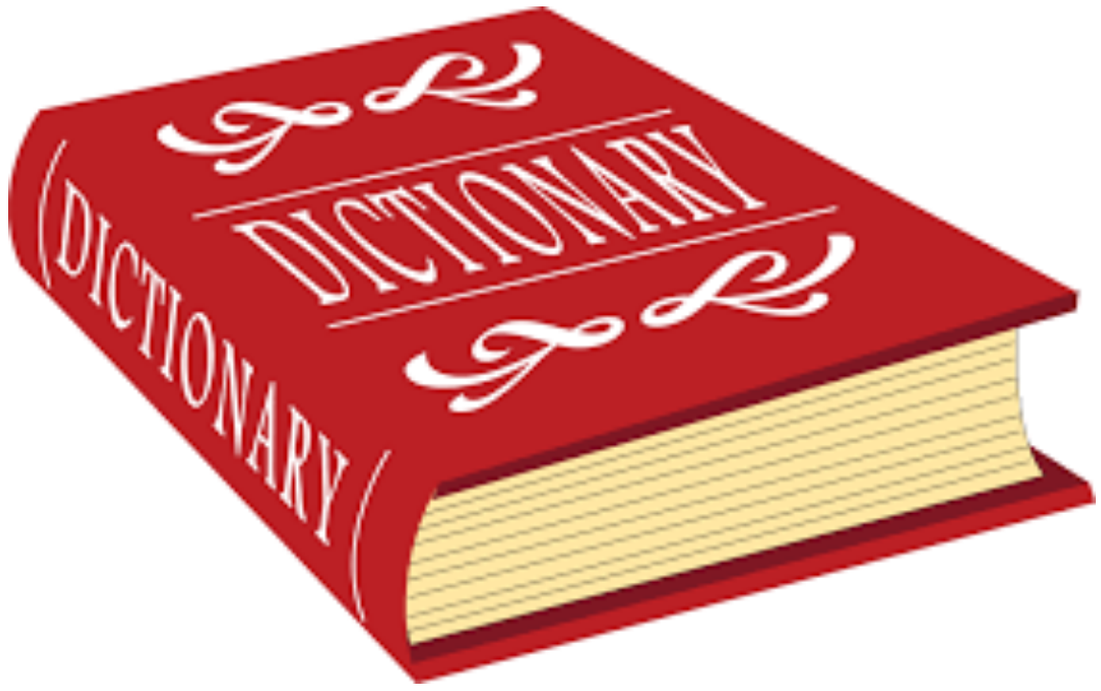
Dictionaries and Hashing



Introduction

Value-Oriented Abstract Data Types

Some ADTs we encountered already were value-oriented. This means that storage and operations on elements contained in the data structure depend on the value of the elements themselves.

The SortedList is one example if V-O ADT.

Can you provide another example?

A new V-O ADT: The Dictionary

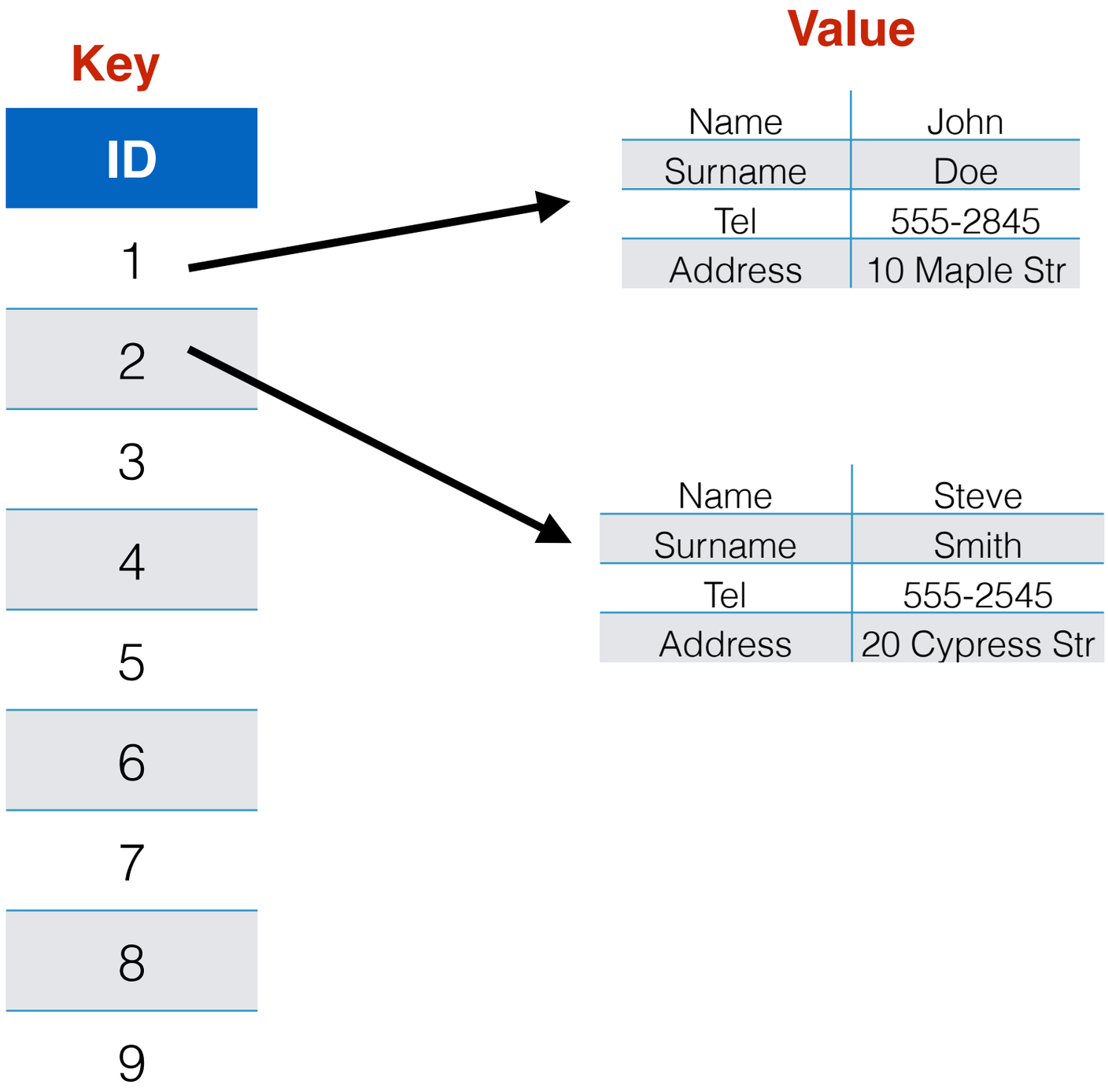
A dictionary is an ADT for storing items of a certain type. The items are accessed by an associated key.
This means that every item is associated by an unique key.

The definition of a key can give significant advantages in some operations like search.
The arrangement of the data and the choice of the key are important for the overall efficiency of this ADT.

Refer to Chp. 18

Examples

Key	Value
ID	Name
1	John Doe
2	Steve Smith
3	Mike Miller
4	Jack Stanford



Note: Sometimes Dictionaries are also called Tables or Maps (Multi-maps).

An Interface for the Dictionary ADT

```
template<class ItemType, class KeyType>
class DictionaryInterface
{
public:
    virtual bool isEmpty() const = 0;

    virtual int getNumberOfItems() const = 0;

    //Adds a new item according to the key value
    virtual bool add(const KeyType& searchKey, const ItemType& newItem) = 0;

    virtual bool remove(const KeyType& searchKey) = 0;

    virtual void clear() = 0;

    virtual ItemType getItem(const KeyType& searchKey) = 0;

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

    //Useful for applying a certain operation on every item
    virtual void traverse(void visit(ItemType&)) const = 0;
};
```

"Client Function": defined outside the class

Note that there are no methods for modifying keys: why?

Dictionary Implementations

As usual, different implementations are possible and depend on the specific problem we would like to solve with the ADT.

Implementations based on linear ADTs:

- by search-key sorted array
- by search-key sorted link-chain
- unsorted array
- unsorted link-chain

In-class exercise:

Consider the **sorted** array and link-chain implementations of a dictionary.

1) Discuss (from the time-complexity point of view) the insertion operation for these two implementations.
Is it faster in one of the two cases?

2) What is exactly the computational complexity on the insertion operation in the two cases?

3) What about a search operation?

The Item a Dictionary: the "Entry"

```
template <class KeyType, class ItemType>
class Entry
{
private:
   ItemType item;
   KeyType searchKey;

protected:
   void setKey(const KeyType& searchKey);

public:
   Entry();
   Entry(ItemType newEntry, KeyType searchKey);
   ItemType getItem() const;
   KeyType getKey() const;
   void setItem(const ItemType& newEntry);
   bool operator==(const Entry<KeyType, ItemType>& rightHandItem) const;
   bool operator>(const Entry<KeyType, ItemType>& rightHandItem) const;
};
```

Array Implementation

```
#include "DictionaryInterface.h"
#include “Entry.h"

template <class KeyType, class ItemType>
class ArrayDictionary : public DictionaryInterface<KeyType, ItemType>
{
private:
   static const int DEFAULT_CAPACITY = 21;
   Entry<KeyType, ItemType>* items; // Array of dictionary entries
   int itemCount;                   // Current count of dictionary items
   int maxItems;                    // Maximum capacity of the dictionary
   void destroyDictionary();
   int findEntryIndex(int firstIndex, int lastIndex,
                      const KeyType& searchKey) const;
public:
   ArrayDictionary();
   ArrayDictionary(int maxNumberOfEntries);
   ArrayDictionary(const ArrayDictionary<KeyType, ItemType>& dict);

   virtual ~ArrayDictionary();

   bool isEmpty() const;
   int getNumberOfItems() const;
   bool add(const KeyType& searchKey, const ItemType& newItem);
   bool remove(const KeyType& searchKey);
   void clear();
   ItemType getItem(const KeyType& searchKey) const;
   bool contains(const KeyType& searchKey) const;

   void traverse(void visit(ItemType&)) const;
};
```

What about Trees?

Is it possible to use a non-linear ADT for implementing a Dictionary?
A Sorted Binary Tree seems a good choice. **Why?**

```
#include "DictionaryInterface.h"
#include "BinarySearchTree.h"
#include "Entry.h"

template <class KeyType, class ItemType>
class TreeDictionary : public DictionaryInterface<KeyType, ItemType>
{
private:
   BinarySearchTree<Entry<KeyType, ItemType> > itemTree;

   void traversalHelper(Entry<KeyType, ItemType>& theEntry);

public:
   TreeDictionary();
   TreeDictionary(const TreeDictionary<KeyType, ItemType>& dict);
   virtual ~TreeDictionary();

   bool isEmpty() const;
   int getNumberOfItems() const;
   bool add(const KeyType& searchKey, const ItemType& newItem);
   bool remove(const KeyType& searchKey);
   void clear();
   ItemType getItem(const KeyType& searchKey) const;
   bool contains(const KeyType& searchKey) const;

   void traverse(void visit(ItemType&)) const;
};
```

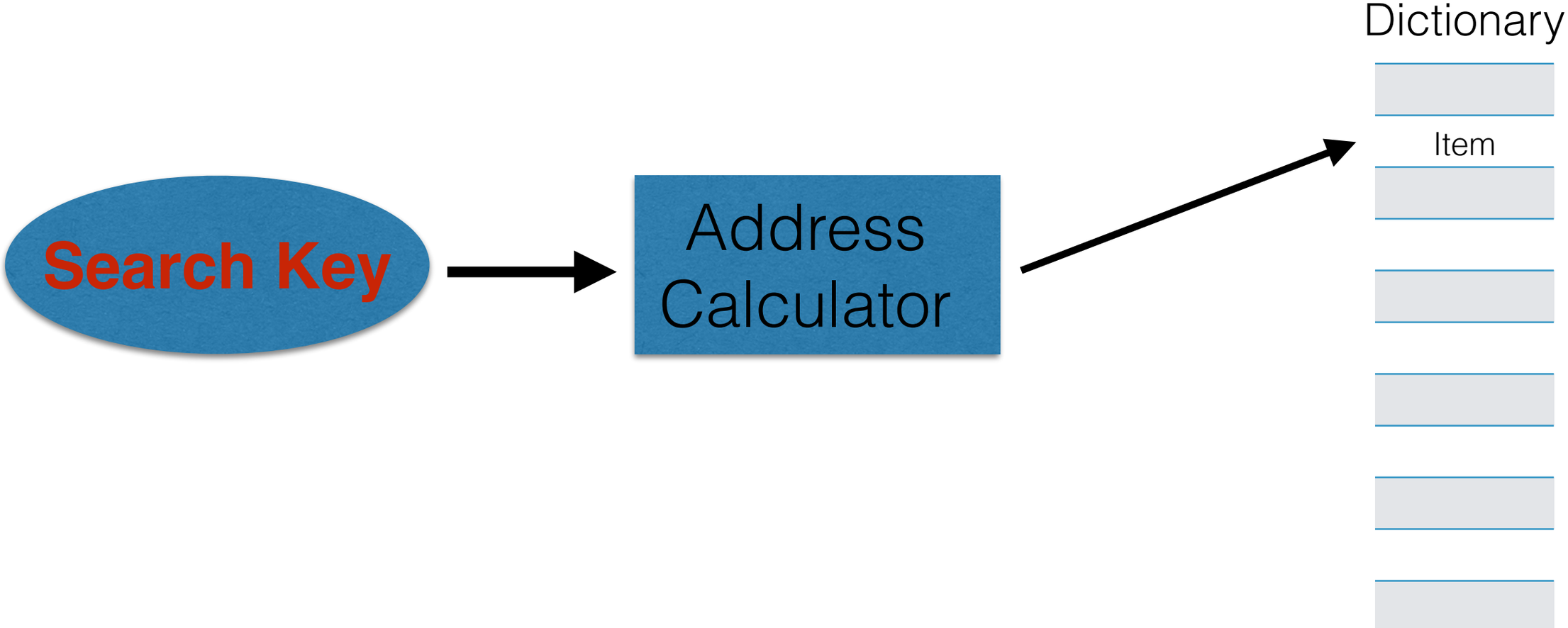
Computational Complexity Considerations

ADT	Insertion	Removal	Retrieval	Traversal
Unsorted Array	O(1)	O(N)	O(N)	O(N)
Unsorted Linked Chain	O(1)	O(N)	O(N)	O(N)
Sorted Array	O(N)	O(N)	O(log	O(N)
Sorted Linked Chain	O(N)	O(N)	O(N)	O(N)
Binary Search Tree	O(log	O(log	O(log	O(N)

Hashing

Is it possible to do better than a search tree for locating an item in a dictionary?
This means to be faster than $O(\log_2 N)$!
A technique called **Hashing** allows retrieval times of order O(1).

The idea is to have an "address calculator" which, given the search key, returns after O(1) calculations the exact address of the item in the dictionary.



All this means to realize a mapping between keys and dictionary positions.
What desirable properties should such map (function) have?

A Simple Hashing Function

Example 1: Storing of a telephone address book.
We can use the telephone numbers (they are unique!) as keys and store names and addresses as dictionary items.

If the dictionary is an array, the hash function can be very simple:

The phone number can be directly interpreted as the array position!

555-2748 —> items[5552748]

Other Simple Hashing Techniques

Digit Selection:
Considering again the previous example, if your phone book must store only a certain number of phone numbers (say, 1000), you can map the phone numbers into a 3-digit number selecting only certain digits from the number:

55**5**-2**8**4**7** —-> 587 —-> items[587]

Folding:
Folding combines in a simple way the digits of the key.
Example 1: sum of the digits of the key.
Example 2: sum of groups of digits of the key.

Modulo Arithmetic
A Hash Function can be constructed with the modulo (%) operation:

h(key) = key % TableSize

For example, key%101 maps key into [0-100].
The TableSize number must be carefully chosen for avoiding collisions.
Can you see when a collision might happen?

String to Numbers Conversion

Since we can always covert a string into a number, it is sufficient to study only hash functions operating on integers.

Example1: Assign to every letter a number.

Example2: Example1 + apply folding

Example3: Assign to every letter a binary code and then concatenate the numbers.

Example4: Assign to every letter a binary ASCII code + concatenation.

String->Numbers conversion

Example: convert "NOTE" to a number using the ASCII encoding + concatenation:

$$N \text{ —> } 01110_2\ (14_{10})$$
$$O \text{ —> } 01111_2\ (15_{10})$$
$$T \text{ —> } 10100_2\ (20_{10})$$
$$E \text{ —> } 00101_2\ (05_{10})$$

Result:

NOTE —> $01110011111010000101_{\mathbf{2}} = \mathbf{474757_{10}}$

NOTE: The folding by addition results in collisions. For example, the word "TONE" will result in the same address in the dictionary!

A computationally faster way to compute the hash function: **Horner's Rule**.

Instead of converting the binary number to a decimal, we can do the following. After observing that our binary numbers are represented by 5 digits (2^5=32) we can write:

$$474757 = 14x32^3+15x32^2+20x32^1+5x32^0 = ((14x32+15)x32+20)x32 +5$$

according to the Horner's rule:

$$p(x) = \sum_{i=0}^{N} a_i x^i = a_0 + x(a_1 + x(a_2 + ... + x(a_{n-1} + a_n x)))$$

Collisions

When two keys are mapped by the hash function into the same location, we have a potential **collision**.
The collision is indeed realized if the spot in the dictionary is already occupied.

There are different techniques for resolving collisions:

Approach 1: **Open Addressing**

- Linear Probing
- Quadratic Probing
- Double Hashing
- Increase hash table size

Approach 2: **Table Restructuring**

- Buckets Technique
- Separate Chaining

Linear and Quadratic Probing

In the case of collision, a simple strategy is **linear probing**: you probe if the next spot is empty and you keep trying in a sequential way increasing the locating index linearly.

```
dict[h(key)] -> dict[h(key)+1] -> dict[h(key)+2] -> …
```

The **retrieval** operation works in the exact same way (indeed calculating a key->position map is exactly the same operation you do while retrieving).

The **removal** is more complicated: if you place an item after a collision and then remove the previous item, you loose the probing sequence!
The problem is solved placing a code in the empty entries, for example:

- occupied , empty , removed

Linear probing has the drawback that after many insert/removal operations, items tend to cluster in the same locations of the dictionary.
This phenomenon is known as **primary clustering**.

A simple solution to primary clustering is **quadratic probing**:

```
dict[h(key)] -> dict[h(key)+1²] -> dict[h(key)+2²] -> …
```

NOTE: Quadratic probing can result in **secondary clustering**, since with items hashed in the same location, quadratic probing uses the same probing sequence for both (this is common in all the methods where the sequence is independent form the key). The result is a slower search process. This problem is anyway smaller than primary clustering.

Double Hashing

Double hashing is designed for greatly reducing clustering.
The idea is to construct key-dependent probe sequences (see for comparison the note at the end of the previous slide).

The method is based on two hash functions:

h_1(key) determines the starting location of the probing
h_2(key) determines the size of the steps taken (h_2(key) must be always non-zero).

Example:

h_1(key) = key % **11**
h_2(key) = **7** - (key % **7**)

Your probe sequences will visit all the locations of the table if the table size and the step are prime among each other.

Exercise: Find the probing sequence in the example given before, given key = 22.

Dictionary Size Change

As you fill up the hash table, the chance of collision increases. A solution could be to increase the size of the table.

There are some problems with this: you have to increase the size to another prime number (just doubling it is not a good idea: why?)

Increasing the table size, means recalculating new positions for the already present items.
This process is known as **rehashing**.

Restructuring the Hash Table

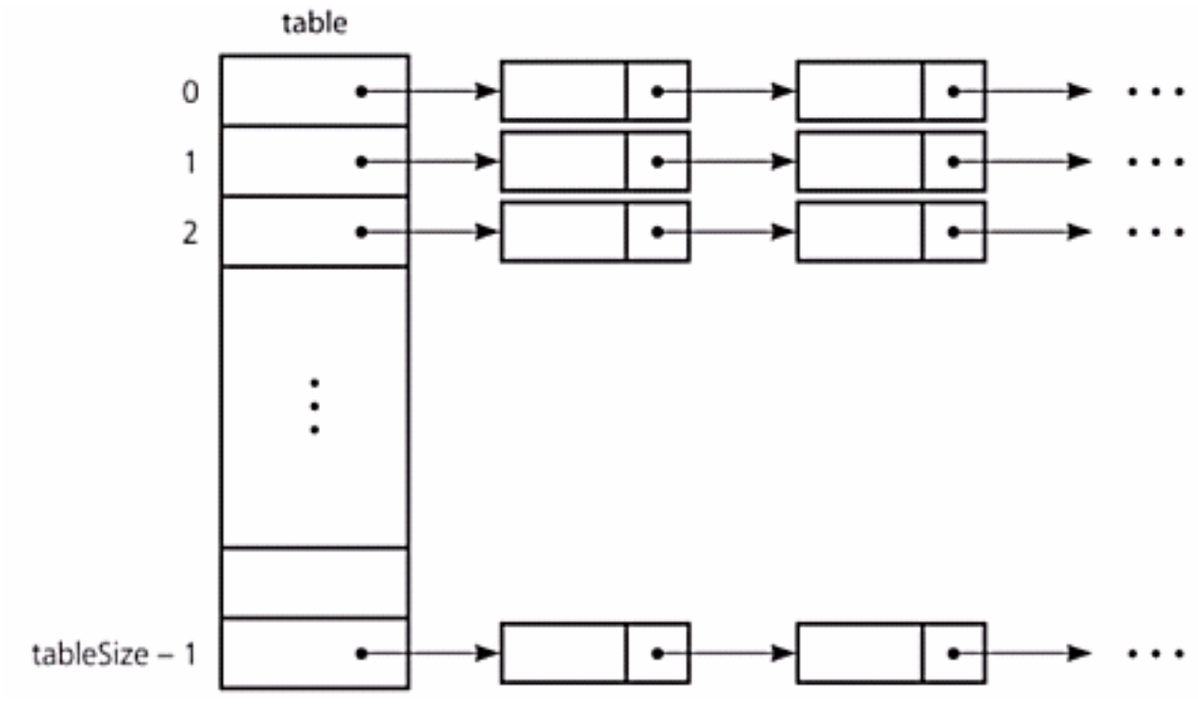
Another way to resolve collisions is to restructure the hash table. This can be done with different techniques. The basic idea is to store more items at the same location.

Use of buckets:

We can turn the location within the table array into an array too: in this way we can store more items. The array at a specific location is called a “bucket”.
The size of the bucket must be chosen carefully: if too small, it allows collisions, if too large, it degrades hash table performance.

Separate Chaining:

This approach is similar to the buckets one, but it is fully dynamic.
The dictionary itself is organized as a linked chain and also the buckets are linked chains. I this way a collision can be resolved in any case.

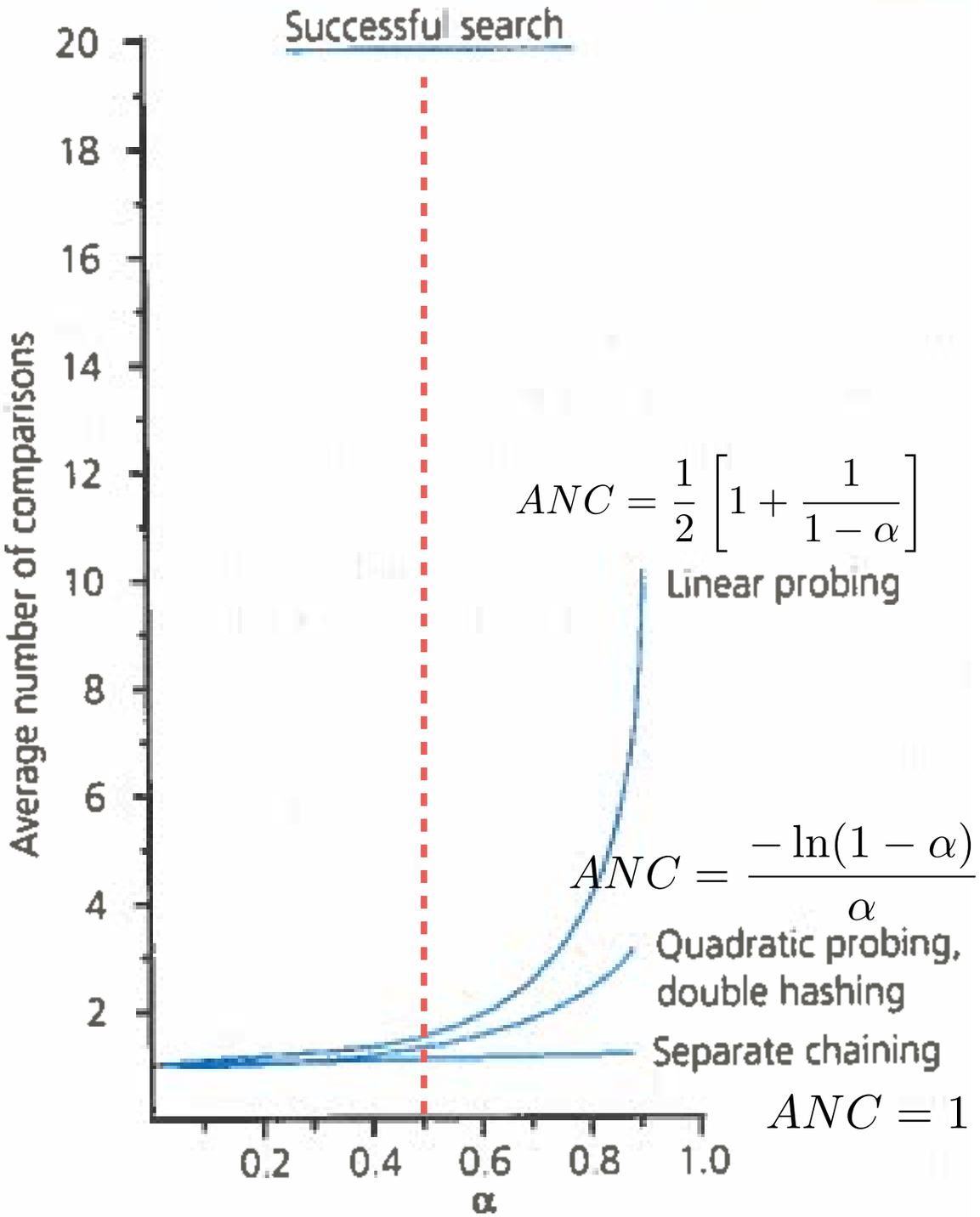


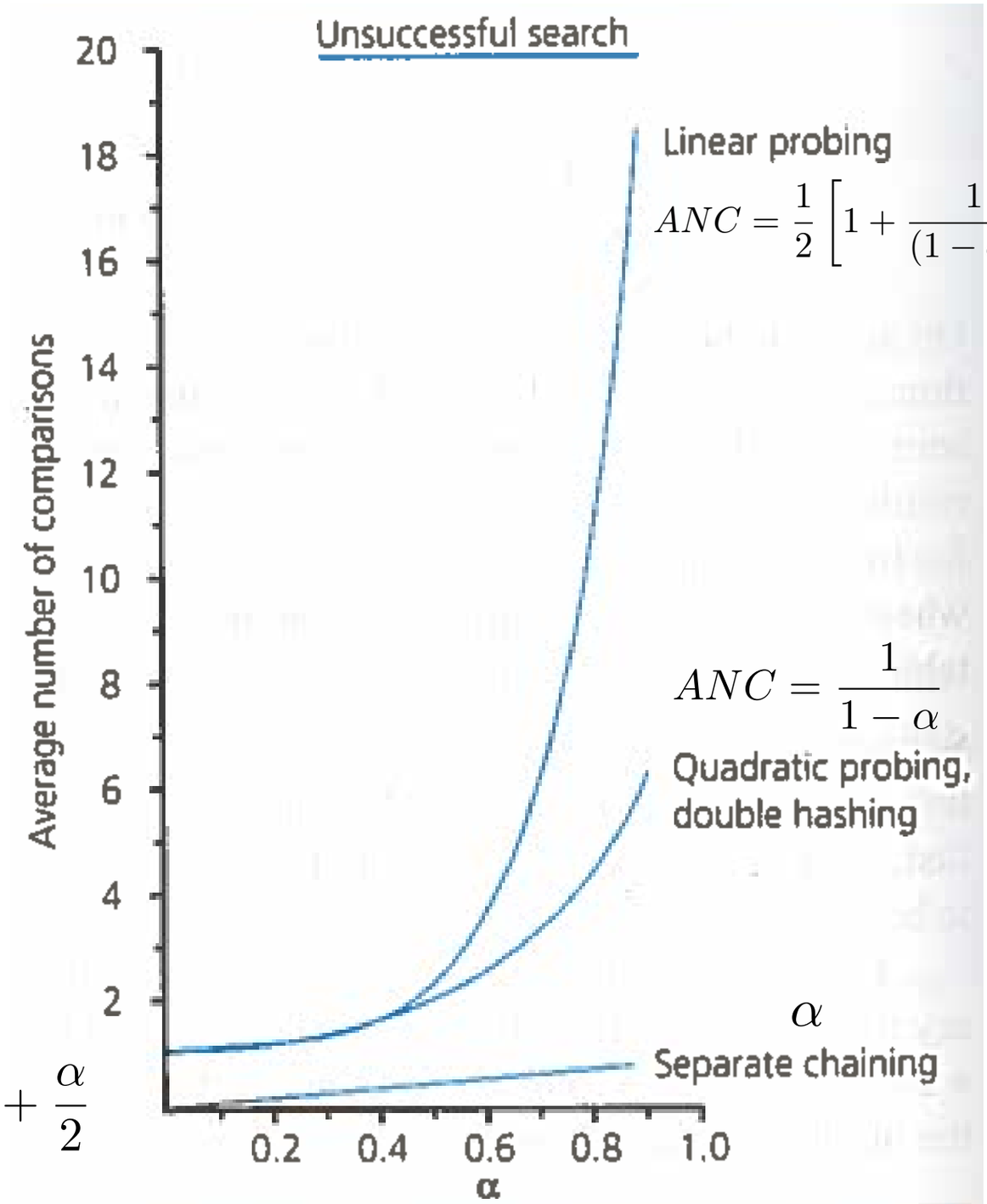
Hashing Efficiency Considerations

Load Factor: $\alpha = \dfrac{\text{Number of table items}}{\text{table size}}$

The hashing efficiency decreases as the load factor increases, since the collision probability is higher for higher occupancy.

Hashing Efficiency Considerations





Hashing Function Properties

In constructing a hash function (which is rarely “perfect”), the following questions are relevant:

1) Is the function easy to computer?

2) Does the function scatter the data evenly?

3) How well are random data scattered?

4) How well are non-random data scattered?

Two final comments:
Although hash tables are designed for fast retrieval (O(1), typically), there is an operation which is quite inefficient: the traversal in a certain order. Do you understand why?

Hashing ultimately calculates where to look, eliminating the need of searching!

A STL Implementation of an Hash Table

```
template < class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Alloc = allocator< pair<const Key,T> >
           > class unordered_map;
```

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main(){

  unordered_map<int,char> hash;

  hash[0] = 'a';
  hash[1] = 'b';
  hash[2] = 'c';
  hash[3] = 'd';

  for (auto &pair : hash) cout << pair.first << " " << pair.second << endl;

  return 0;

}
```