

## **Dictionaries versus log files**

As with a map, a dictionary may be regarded as a log file stored in main memory. However, a log file is stored permanently on tape or disk.

Thus the *entries of a dictionary* correspond exactly to *the records of a log file*.

There are many similarities between a dictionary and a log file.

*But the important differences are:*

1. Because a log file is on disk (typically), efficiency considerations are even more important than with dictionaries in main memory. Disk access is very slow.
2. Because a dictionary is stored in main memory, and has the life only of the executing program, much more flexible storage structures are possible, as was the case with maps.

## ORDER OF ENTRIES IN THE DICTIONARY

**There are two kinds of dictionary:**

### **Unordered Dictionaries**

The entries are not stored in any key order, and the user can only compare a target key for equality with a dictionary key, using an equality tester.

Whatever order of entries exists in the unordered dictionary is normally not visible to the user of the dictionary. The user simply thinks of a dictionary as a convenient entry-storage receptacle for entry data, such as log data, to which access on the basis of a non unique key is possible, using a well-defined set of dictionary methods.

Most importantly: *an unordered dictionary will have a method that allows retrieval of all the entries with the same key value, in the form of an iterator. [This is the method `findAll(k)`, below.]*

### **Ordered Dictionaries**

The entries are stored in key order, and the user can not only compare a target key for equality with a dictionary key, but also whether a target key is greater or less than a dictionary key, using a comparator. Otherwise, it is like an unordered dictionary, with all the methods of an unordered dictionary, but also some additional methods

## The Dictionary ADT

Suppose an unordered dictionary D. The following methods are standard:

- size()* Returns number of entries in D.  
**Input:** none      **Output:** Integer
- isEmpty()* Returns true if D is empty.  
**Input:** none      **Output:** Boolean
- find (k)* Returns *some one* entry  $\langle k \ V \rangle$ .  
in D, for which target k is the key  
If target k not found in D, return **null**.  
**Input:** Object k    **Output:** Entry  $\langle k \ V \rangle$
- findAll (k)* Returns an iterator of *all* entries  
 $\langle k \ V1 \rangle, \langle k \ V2 \rangle, \langle k \ V3 \rangle \dots$   
in D, for which target k is the key  
If target k not found in D, return **null**.  
**Input:** Object k    **Output:** Iterator of entries
- insert (k, v)* Inserts an entry  $\langle k \ v \rangle$  into D,  
and returns entry  $\langle k \ v \rangle$   
**Input:** Objects k, v    **Output:** Entry  $\langle k \ v \rangle$
- remove (e)* Removes the entry e, and returns the  
removed e. If e not found in D, return **null**.  
**Input:** Entry e      **Output:** Entry e

*entries()* Returns an iterator of the entries <k v> in D  
**Input:** none **Output:** Iterator of Entries

*values()* Returns an iterator of the values in M  
**Input:** none **Output:** Iterator of Objects

*To check the operation of the above methods, see Figure 8.2 in text book*

**Note:** There are additional methods for an ordered dictionary, notably methods *first()*, *last()*, *successor()*, *predecessor()*, as listed on p. 396.

## **Techniques for Implementing an Unordered Dictionary**

These are much the same as with a map:

### ***1. Linked List Dictionary.***

Each entry is in a node of a two-way linked list. Since we do not have to be concerned about detecting a duplicate key on insertion, each new entry can be inserted at the end of the list, instead of after scanning the list, as with a map. Order in the list is thus order of entry.

### ***2. Hash Table Dictionary with Separate Chaining***

A bucket may have more than one entry assigned to it for two reasons, causing a chain to grow from it:

- (a) There are synonyms: two entries with different keys hash to the same bucket number. These go in the chain.
- (b) There are duplicate keys: two entries with the same key must hash to the same bucket number. These also go in the chain emanating from the home bucket.

### ***3. Hash Table Dictionary with Open Addressing***

A bucket may have more than one entry assigned to it as with separate chaining. Whether synonyms, or entries with the same key, they all go into empty buckets with bucket number above the home bucket number, just as with an open-addressed hash-table map.

#### **1. Linked-list Implementation of an Unordered Dictionary**

**Insertion:** *This is fast* [it's  $O(1)$  ], since we typically just insert at the end of the list, with no scan of the list.

**Retrieval:** *This is slow* with a linked list [it's  $O(n)$ ]:

(1) For *findall(k)*, [which retrieves all the entries with key  $k$ ], a scan of *all* the entries, as we look for all of the entries with the target key  $k$ , is necessary. *The entire list must be scanned.* So *findAll(k)* is always  $O(n)$ .

For *find(k)*, [which retrieves the first entries encountered with key  $k$ ], we scan until we find the first entry with the target key. On average,  $n/2$  nodes must be visited, In the best case 1 node is visited, and in the worst case  $n$  nodes are visited. So *find(k)* is  $O(n/2)$ , which is  $O(n)$ , on average.

The linked-list implementation is therefore best suited for such logs, and similar dictionaries, where *there are a lot of insertions, but few retrievals.*

## (Unordered) Linked-list Dictionary Methods

Assume a linked list dictionary class **LLDictionary** with a member object *S* that is a **List**:

```
public class LLDictionary implements Dictionary{  
    List      S = new List (-----);  
    -----
```

**LLDictionary** has the following methods:

Algorithm *insert* (*k*, *v*)

1. Create an entry  $e = \langle k \ v \rangle$
2. Insert  $e$  at the end of the list, and increment number of entries in the list too, by a call to:  
***S.insertLast* (*e*)**
3. Return  $e$ .

***O(1) even if n nodes in list.***

Algorithm *find*(*k*)

1. Search along the list until either the key  $k$  is found or not found.
2. If found return the ***entry***.
3. If not found return ***null***

***O(n/2) on average, or O(n), if n nodes in list.***

**Note:** Strictly, in the following algorithms for generic keys, we should use an equality tester *c*, to test for matching keys:

***c.isEqualTo(key1, key2)***

This was omitted for readability purposes.

Algorithm

*entries()*

1. Construct an element iterator *elemIt* over the elements of S, by means of :

*java.util.Iterator elemIt S.elements();*

2. Return this iterator by means of:

*return elemIt*

*Algorithm is O(1)*

Algorithm

*findAll(k)*

1. Create an initially empty special linked list object *eList* to hold the entries found in S, by means of

*List eList = new List (----);*

2. Use *entries()* to create an iterator *B* of all **entries** in S:

*java.util.Iterator B = this.entries();*

3. Scan *B* with *hasNext()* and *next()*, extract all **entries** with target *k*, and put them in the special list *eList*:

```
while (B.hasNext()) {  
    Entry e = B.next();  
    if (e.key() == k)  
        //should use equality tester  
        eList.insertLast(); }
```



## 2. Hash Table Dictionary with Separate Chaining

When the dictionary is going to be subjected to large numbers of retrievals, the linked-list implementation previously is not efficient, since retrievals are  $O(n)$ . A hash table is a good way to get fast retrieval, either with separate chaining or open addressing. The text book gives an example of a hash table with separate chaining only.

With separate chaining, we can have a linked list dictionary emanating from each home bucket. This means a hash-table dictionary that's a bucket array of linked-list dictionaries.

### *Bucket array of linked-list Dictionaries*

This elegant approach, described in the textbook, and similar to the map of maps earlier, uses a single array of buckets as the hash-table based dictionary container. Each bucket holds a pointer to a small linked-list dictionary, like the one previously described.

For a given bucket number, the associated linked-list dictionary holds all synonym entries (different keys but same bucket number), and entries with the same key, whose home bucket is that bucket number.