Dictionary

Dictionaries permit content-based retrieval, unlike the position-based retrieval of stacks and queues. They support three primary operations –

- **Insert(x,d)** — Insert item x into dictionary d.
- **Delete(x,d)** — Remove item x (or the item pointed to by x) from dictionary d.
- **Search(k,d)** — Return an item with key k if one exists in dictionary d.

A data structures course may well present a dozen different ways to implement dictionaries, including sorted/unsorted linked lists, sorted/unsorted arrays, and a forest full of random, splay, AVL, and red-black trees – not to mention all the variations on hashing.

The primary issue in algorithm analysis is performance, namely, achieving the best possible trade-off between the costs of these three operations. But what we usually want in practice is the simplest way to get the job done under the given time constraints. The right answer depends on how much the contents of your dictionary change over the course of execution:

1. Static Dictionaries — These structures get built once and never change. Thus they need to support search, but not insertion on deletion. The right answer for static dictionaries is typically an array. The only real question is whether to keep it sorted, in order to use binary search to provide fast membership queries. Unless you have tight time constraints, it probably isn't worth using binary search until n > 100 or so. You might even get away with sequential search to n = 1, 000 or more, provided you will not be doing too many searches.

2. Semi-dynamic Dictionaries — These structures support insertion and search queries, but not deletion. If we know an upper bound on the number of elements to be inserted we can use an array, but otherwise we must use a linked structure.
Hash tables are excellent dictionary data structures, particularly if deletion need not be supported. The idea is to apply a function to the search key so we can determine where the item will appear in an array without looking at the

other items. To make the table of reasonable size, we must allow for collisions, two distinct keys mapped to the same location. The two components to hashing are (1.) defining a hash function to map keys to integers in a certain range, and (2.) setting up an array as big as this range, so that the hash function value can specify an index. The basic hash function converts the key to an integer, and takes the value of this integer mod the size of the hash table. Selecting a table size to be a prime number (or at least avoiding obvious composite numbers like 1,000) is helpful to avoid trouble. Strings can be converted to integers by using the letters to form a base "alphabet-size" number system. To convert "steve" to a number, observe that e is the 5th letter of the alphabet, s is the 19th letter, t is the 20th letter, and v is the 22nd letter. Thus "steve" ⇒ 264×19+263×20+262×5+261×22+260×5 = 9, 038, 021. The first, last, or middle ten characters or so will likely suffice for a good index. The absence of deletion makes open addressing a nice, easy way to resolve collisions. In open addressing, we use a simple rule to decide where to put a new item when the desired space is already occupied. Suppose we always put it in the next unoccupied cell. On searching for a given item, we go to the intended location and search sequentially. If we find an empty cell before we find the item, it does not exist anywhere in the table. Deletion in an open addressing scheme is very ugly, since removing one element can break a chain of insertions, making some elements inaccessible. The key to efficiency is using a large-enough table that contains many holes. Don't be cheap in selecting your table size or else you will pay the price later.

3. Fully Dynamic Dictionaries — Hash tables are great for fully dynamic dictionaries as well, provided we use chaining as the collision resolution mechanism. Here we associate a linked list with each table location, so insertion, deletion, and query reduce to the same problem in linked lists. If the hash function does a good job the m keys will be distributed uniformly in a table of size n, so each list will be short enough to search quickly.

Source:

http://www.learnalgorithms.in/#