Hash Table:
A data structure to hold items for retrieval

Searching for a key in an unsorted array needs O(n) time, as one needs to go through all the keys in the worst case. A sorted array would need O(log n) searches using the binary search method. A binary search tree, would also need O(log n) searches if the tree is almost balanced.
A different approach would be to store the key in a specific position in a HASH TABLE. A hashing function h(.) is used on the key k to compute its position h(k) in the table. Each key thus gets stored in specific location in the Hash table. Once all the keys are stored in the able, searching for a key involves applying the same hashing function on the key to obtain its position. Then in time closer to O(1), one can retrieve the data from the Hash table, which would be independent of how many other keys are stored in the table. This holds good for many of the keys, for some of the keys the time could be much longer, but on average this turns out to be really small.

Hash Functions

Division method:

A hash function must guarantee that the number it returns is a valid index to one of the table cells. A simple way is to use h(k) modulo Tablesize.

Example: Suppose the table is to store strings. A very simple hash function would be to add up ASCII values of all the characters in the string and take modulo of table size, say 97.

Thus *cobb* would be stored at the location
(64+3 + 64+15 + 64+2 + 64+2) % 97 = 88

hike would be stored at the location
(64+8 + 64+9 + 64+11 + 64+5) % 97 = 2

ppqq would be stored at the location
(64+16 + 64+16 + 64+17 + 64+17) % 97 = 35

abcd would be stored at the location
(64+1 + 64+2 + 64+3 + 64+4) % 97 = 76

The key idea is to get numbers far from not close to each other. A better hashing function for a string $s_0\ s_2\ s_{1\ldots}\ s_N$ given a table size T_size would be

$[(\text{ascii}(s_0)* 128^0 + (\text{ascii}(s_1)* 128^1 + (\text{ascii}(s_2)* 128^2 +\ldots+ (\text{ascii}(s_N)* 128^N\ ]$ % T_size

It is very likely that the computation of the hashing function fails for large strings because of overflow in various terms. This can be avoided by using Horner's rule and using mod at each stage of computation. Use of Horner's rule would imply computing the above function in the following fashion:

$$ascii(s_0) + 128(ascii(s_1) + 128(ascii(s_2) + ...+(128(ascii(s_{N-1}) + 128\ ascii(s_N))...))$$

Whatever hash function is used, it may happen that the hash function chosen returns the same position for two strings. This problem is called collision, and methods for resolving collision are discussed later.

Folding:

In this method the key is divided in several parts(that is , the key is HASHED). The parts are combined or folded together in certain manner. For example, a social security number 123-45-6789 can be broken in three parts 123, 456, 789 and then these parts can be added to yield the position as 1368 % Table size. The folding can be done in number of ways. Another possibility is to divide the number in 4 parts 12, 34, 56, 789 and added together.

Mid-Square function:

The key is squared, and the middle part of the result is used as address for the hash table. If the key is a string, it is converted to a number. Here the entire key participates in generating the address so that there is a better chance that different addresses are generated even for keys close to each other. For example,

Key	*squaredvalue*	*middle part*
3121	9740641	406
3122	9746884	468
3123	9753129	531

In practice it is more efficient to choose a power of 2 for the size of the table and extract the middle part of the bit representation of the square of a key. If the table size is chosen in this example as 1024, the binary representation of square of 3121 is 1001010-0101000010-1100001. The middle part can be easily extracted using a mask and a shift operation.

Collision Resolution:

For almost all hash functions, it is possible that more than one key is assigned to the same table address. For example, if the hash function computes the address just based on the first letter of the key, then all keys starting with the same letter will be hashed to the same location, resulting in a collision. Collision can be resolved partially, by choosing another hash function, which computes the address based on first two letters of the key. However, even if a hash function is chosen in which all the letters of the key participate, there is still a possibility that number of keys may hash to the same location in the hash table.
Another factor that can be used to avoid collision of multiple keys is the size of the hash table. A larger size will result in fewer collisions, but that would also increase the access time during retrieval.
A number of strategies have been proposed to take care of collision of multiple keys.

Linear Probing:

When a collision takes place, search for next available position in the table, by making a sequential search. Thus the addresses are generated by
[H(k) + p(1)] mod Tsize
[H(k) +p(2)] mod Tsize
……..
[H(k) + p(i)] mod Tsize

Where p(i) is the probing function. The simplest method is linear probing, for which p(i) = i

Consider a simple example with table of size 10. After hashing keys 22 , 9 and 43, the table is shown below

0	1	2	3	4	5	6	7	8	9
		22	43						9

When keys 32 and 65 arrive, they are stored as follows:

0	1	2	3	4	5	6	7	8	9
		22	43	32	65				9

The key 54 can not be stored in its designated place as it collides with 32, so a new place for it is found by linear probing to position 6 which is empty at this point.

0	1	2	3	4	5	6	7	8	9
		22	43	32	65	54			9

When the search reaches end of the table, it continues from the first location again. Thus the key 59 will be stored as follows;

0	1	2	3	4	5	6	7	8	9
59		22	43	32	65	54			9

This shows the problem with the linear probing scheme. The keys start forming clusters, and the clusters have a tendency to grow faster, as more and more collisions take place and the new keys get attached to one end of the cluster. These are called primary clusters. The problem with such clusters is with unsuccessful searches. The search must go through to the end of the table and start from the beginning of the table.

Quadratic Probing:

To overcome the primary clustering problem, quadratic probing places the elements not in immediate succession, but further away. One choice is the following quadratic function

$p(i) = h(k) +(-1)^{i-1} ((i+1)/2)^2$ for $i = 1,2,..,Tsize-1$

This results in following sequence of addresses

$h(k) +i^2$
$,h(k) -i^2$
for i = 1,2, …

The size of the table is chosen as a prime number, preferably of the form 4j+3 where j is an integer. Thus for j=4, the table size is 19. Now if the key to be inserted in this table is 9, it will start looking at following table positions, till it finds a free location.

9,10, 8, 13, 5, 18, 0,…

Although using quadratic probing gives much better results than linear probing, the problem of cluster buildup is not avoided altogether. Such clusters are called secondary clusters.

Double Hashing:

The problem of secondary clustering is best addressed with double hashing. A second function is used for resolving conflicts. The probing sequence becomes

$h(k), h(k)+h_p(k), h(k)+ 2h_p(k) ,.. . .$

One choice for the secondary function is
$h_p(k)= i.h(k) +1$

Thus if the key is hashed to position j, the resulting probing sequence is
$j,2j+1,5j+2,. ...$

Whatever scheme is used for hashing, it is obvious that the search time depends on how much of the table is filled up. The search time increases with the number of elements in the table. In the worst case, one may have to go through all the table entries.

Separate chaining:

A popular and space efficient alternative to above schemes is separate chaining hashing. Each position of the table is associated with a linked list or chain of structures whose data field stores the keys. The hashing table is a table of references to these linked lists. Thus if we consider the example we have taken at the beginning, keys 78, 8, 38, 28, 58 would be referenced from position 8 in the reference hash table, instead of visiting other cells of the hash table.

In this scheme, the table can never overflow, because the linked lists are extended only upon the arrival of new keys. A new key is always added to the front of the linked list, thus minimizing storage time. Many unsuccessful searches may end up in empty lists, reducing the search time, compared with other hashing schemes. This is of course at the expense of extra storage for linkedlist references.

Comparison with Binary Search Trees:

Binary search trees need O(log n) for all operations – search, insert, delete, findmax, findmin, provided the data is not presented to it in sorted or almost sorted manner. Hash tables do not support findmax, findmin operations. Even delete is very difficult to support. Suppose the table contains 3 entries 44 at position 4, 54 at position 6 and 24, at position 8. At some stage, let us say we delete 44 from the hash table, leaving an empty slot at position 4. Later if were to search for 24, we first check location 4 and finding it empty may lead us to conclude that the table does not contain any entries ending in 4. If

we suspect the item to be there, we have to search the entire table to discover it. A possible solution is *lazy deletion*, where the element is simply marked as deleted, instead of physically removing it from the table.

However, binary search trees are useful only when these are balanced. Maintaining balanced BST can be very expensive to implement. Thus if we have a suspicion that the input data stream is partially sorted, hashing would be a data structure of choice.

Hashing applications:

Compilers use hash tables to keep track of declared variables in source codes, (symbol table). Hash tables are ideal application for this problem as only insert and find operations are performed. Identifiers are short so hash function can be computed quickly. Most searches are successful.

Another common use of hash tables is in game programs. As the program searches through different lines of play, it keeps track of positions that it has encountered by computing a hash function based on the position (and storing its move for that position). If the same position recurs, usually by a simple transposition of moves, the program can avoid expensive re-computation.

A third use of hashing is online spell checkers. An entire dictionary can be pre-hashed and words can be checked in constant time. Note words do not have to be alphabetized.