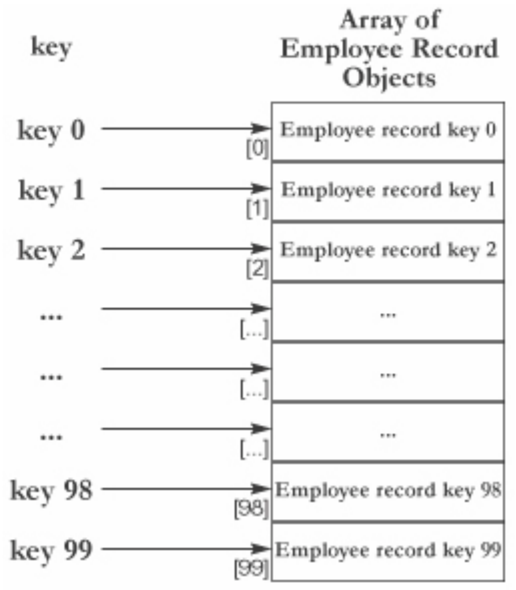
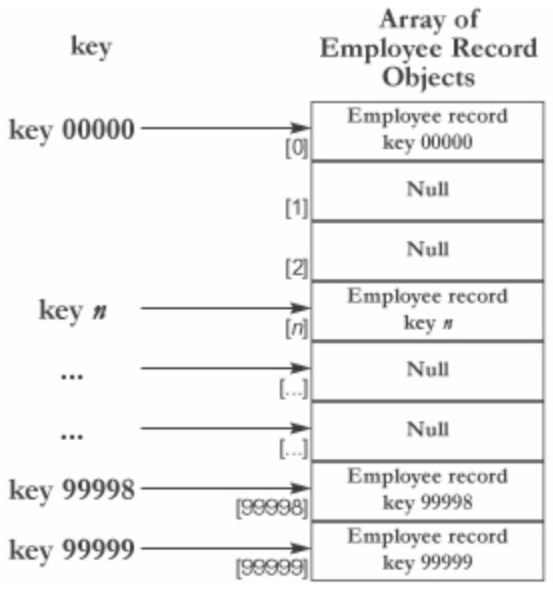
Introduction to Hash Table and Hash Function

This is a short introduction to Hashing mechanism

Introduction
Is it possible to design a search of O(1)– that is, one that has a constant search time, no matter where the element is located in the list? Let's look at an example. We have a list of employees of a fairly small company. Each of 100 employees has an ID number in the range 0 – 99. If we store the elements (employee records) in the array, then each employee's ID number will be an index to the array element where this employee's record will be stored.



In this case once we know the ID number of the employee, we can directly access his record through the array index. There is a one-to-one correspondence between the element's key and the array index. However, in practice, this perfect relationship is not easy to establish or maintain. For example: the same company might use employee's five-digit ID number as the primary key. In this case, key values run from 00000 to 99999. If we want to use the same technique as above, we need to set up an array of size 100,000, of which only 100 elements will be used:



Obviously it is very impractical to waste that much storage.

But what if we keep the array size down to the size that we will actually be using (100 elements) and use just the last two digits of key to identify each employee? For example, the employee with the key number 54876 will be stored in the element of the array with index 76. And the employee with the key 98759 will be stored in the element of the array with index 59. Note that the elements are not stored according to the *value* of the key as they were in the previous example. In fact, the record of the employee with the key number 54876 precedes the record of the employee with key number 98759, even though the *value* of its key is larger. Moreover, we need a way to convert a five-digit key number to two-digit array index. We need some *function* that will do the transformation. Using this technique we call the array a **Hash Table** and a function is called a **Hash Function**.

Hash Table, Hash Function, Collisions.

A **Hash Table** is a data structure in which keys are mapped to array positions by a hash function. This table can be searched for an item in O(1) amortized time (meaning constant time, on average) using a hash function to form an address from the key. Good Hash Table implementations also have O(1) amortized insertion and deletion time, which means that growing and shrinking the number of elements in a Hash Table is also efficient. The easiest way to conceptualize a hash table is to think of it as an array. When a program stores an element in the array, the elements key is transformed by a **hash function** that produces array indexes for that array.

Hash Functions are functions which, when applied to the key, produce an integer which can be used as an address in a hash table. The intent is that elements will be relatively randomly and uniformly distributed. In the example above the code for the hash function would look like this (assuming TABLE_SIZE was defined 100):

```
int Hashtable::hash_function(int id_num)
{
return (id_num % TABLE_SIZE);
}
```

Then, for example, to find an element in the table, a program applies hash function to the element's key, producing the array index at which the element is stored.

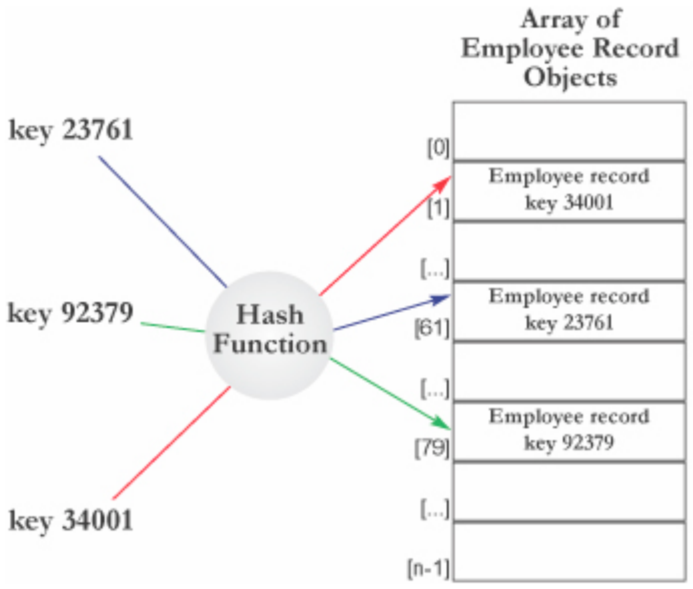
```
Type* Hashtable::find (char* key)
{
int index;

index = hash_function(key);

//finding the element

}
```

Unfortunately, this is not quite simple. For example, we have already stored several employees' records, and our table looks something like this:


Array of Employee Record Objects
key 23761
key 92379
key 34001
Hash Function
[0]
[1]
[...]
[61]
[...]
[79]
[...]
[n-1]
Employee record key 34001
Employee record key 23761
Employee record key 92379

Suppose, our next employee's key is 57879. Then our hash function will produce array index 79. But the array element with index 79 already has a value. As the array begins to fill, keys will inevitably produce the same array index when transformed by the hash function. When more than one element tries to occupy the same array position, we have a *collision*.

Collision is a condition resulting when two or more keys produce the same hash location.

Now, another question arises: can't we generate the hash function that never produces a collision? The answer is that in some cases it is possible to generate such a function. This is known as a perfect hash function.

Perfect Hash Function is a function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such function produces no collisions.

Good Hash Function minimizes collisions by spreading the elements uniformly throughout the array.

There is no magic formula for the creation of the hash function. It can be any mathematical transformation that produces a relatively random and unique distribution of values within the address space of the storage. Although the development of a hash function is trial and error, here are some hints that may make process easier:

- Set the size of the storage space to a prime number. This will help generate a more uniform distribution of addresses.
- Use modulo arithmetic (%). Transform a key in such a way that you can perform X % TABLE_SIZE to generate the addresses
- To transform a numeric key, try something like adding the digits together or picking every other digit.
- To transform a string key, try to add up the ASCII codes of the characters in the string and then perform modulo division.

However, finding a perfect hash function that works for a given data set can be extremely time consuming and very often it is just impossible. Therefore we must live with collisions and learn how to handle them.

Handling Collisions

Two popular ways of dealing with collisions are: rehashing and chaining.

Rehashing - resolving a collision by computing a new hash location (index) in the array.

Linear Probing is one type of rehash (getting a new index) - a very simple one. In case of linear probing, we are looking for an empty spot incrementing the offset by 1 every time. We explore a sequence of location until an empty one is found as follows:

index, index + 1, index + 2, index + 3, ...

In the case of linear probing the code for rehash method will look like this:

```
int Hashtable::rehash(int index)
{
int new_index;

new_index = (index + 1) % TABLE_SIZE;

return new_index;
}
```

An alternative to linear probing is quadratic probing.

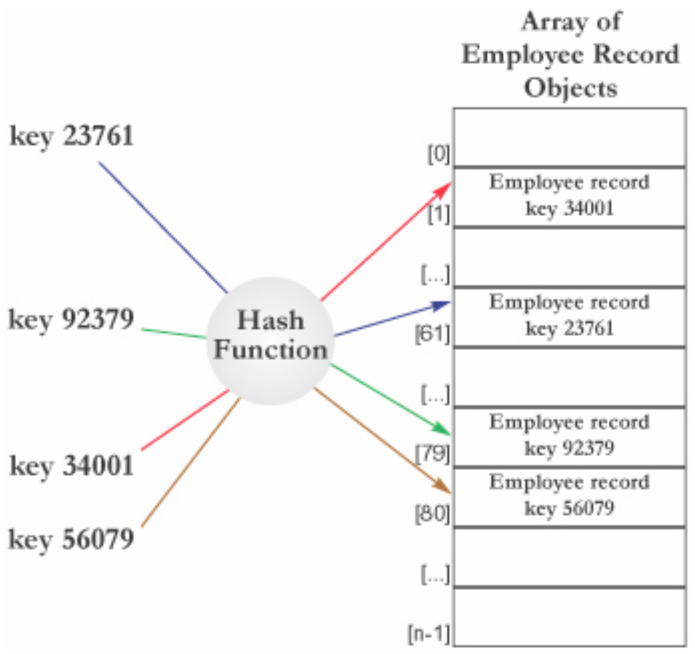
Quadratic Probing is a different way of rehashing. In the case of quadratic probing we are still looking for an empty location. However, instead of incrementing offset by 1 every time, as in linear probing, we will increment the offset by 1, 3, 5, 7, ... We explore a sequence of location until an empty one is found as follows:

    index, index + 1, index + 4, index + 9, index + 16, ...

- **Linear probing technique**

Linear Probing is resolving a hash collision by sequentially searching a hash table beginning at the location returned by the hash function.

In this case, hash table is implemented using an array. The program stores the first element that generates a specific array index at that index. For example, if the hash function generates 79, then you use array index 79 to store the element. When the hash function generates the key 79 again, the program begins a sequential search starting at location 79, looking for the next available spot. The second element whose key was transformed by hash function into 79 will be stored at the location 80, the third at 81 and so on. Of course, if 80 and 81 are already occupied, the elements will be stored farther away from the location generated by hash function.


Array of Employee Record Objects
key 23761
key 92379
key 34001
key 56079
Hash Function
[0]
[1]
[...]
[61]
[...]
[79]
[80]
[...]
[n-1]
Employee record key 34001
Employee record key 23761
Employee record key 92379
Employee record key 56079

Retrieving a value

When it comes to retrieving a value, the program recomputes the array index and checks the key of the element stored at that location. If the desired key value matches the key value at that location, then the element is found. The search time is on the order of 1, O(1) (1 comparison per data item). If the key doesn't match, then the search function begins a sequential search of the array that continues until:

- the desired element is found
- the search encounters an unused position in the array, indicating that the element is not present
- the search encounters the index which was produced by the hash function,indicating that the table is full and the element is not present

In worst case, the search takes (n-1) comparison, which is on the order of n, O(n). "Worst case" is if the table is full and the search function goes through the whole array, (n-1) elements, every time comparing the desired key value with the key at the array location. In the worst case element is either found at the last position before the one that was produced by the hash function or not found at all.

Advantages to this approach

- All the elements (or pointer to the elements) are placed in contiguous storage. This will speed up the sequential searches when collisions do occur.

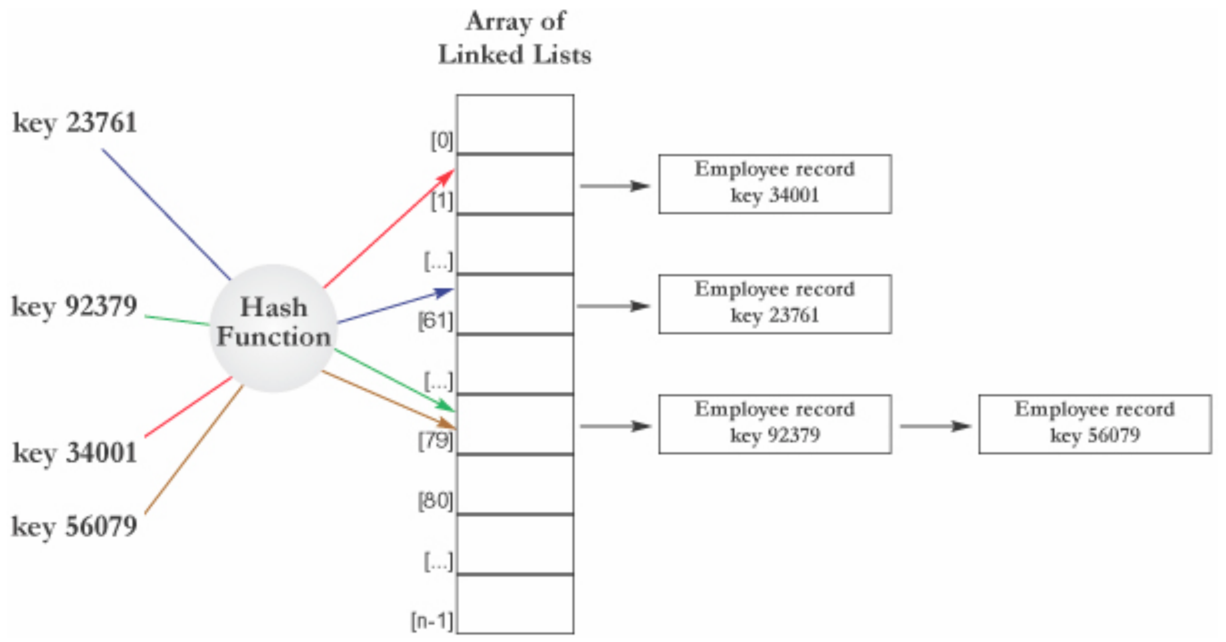
Disadvantages to this approach

- As the number of collisions increases, the distance from the array index computed by the hash function and the actual location of the element increases, increasing search time.
- Element tend to cluster around elements that produce collisions*. As the array fills, there will be gaps of unused locations.
- The hash table has a fixed size. At some point all the elements in the array will be filled. The only alternative at that point is to expand the table, which also means modify the hash function to accommodate the increased address space.

*Note: There are slight variations to the preceding scheme that attempt to space out the elements, avoiding the clustering of elements that occurs with adjacent placement. Collided elements can be placed some fixed number of locations away from the location generated by the hash function. For example, suppose we decided to place the elements four locations apart. If the first element is stored at the location 76, the second element whose hashed key generated is 76, would be stored at 80, the third at 84 and so on. If a location is occupied, the program adds four and checks again, until it finds an unused location.

- **Collision Resolution Using Linked List**

The major alternative to using adjacent storage is to use linked list. In this case table is implemented as an array of linked lists. Every element of the table is a pointer to a list. The list will contain all the elements with the same index produced by the hash function. For example, when the hash function produces key 79, the new node is created and the array element with index 79 now points to this node. When another element's key produces index 79, the new node will be created and attached to the first one and so on. The same set of data as in the previous example will be organized as follows when we use linked list for handling collisions:



Retrieving a value.

When it comes to retrieve a value, the hash function is applied to the desired key and the array index is produced. Then the search function accesses the list to which this array element points to. It compares the desired key to the key in the first node. If it matches, then the element is found. In this case the search of the element is on the order of 1, O(1). If not, then the function goes through the list comparing the keys. Search continues until the element is found or the end of the list is detected. In this case the search time depends on the length of the list.

Advantages to this approach

- The hash table size is unlimited (or limited only by available storage space). In this case you don't need to expand the table and recreate a hash function.
- Collision handling is simple: just insert colliding records into a list.

Disadvantages to this approach

- As the list of collided elements (collision chains) become long, search them for a desired element begin to take longer and longer.
- Using pointers slows the algorithm: time required is to allocate new nodes.