

## Hash Tables

Hashing 1

A hash table employs a function,  $H$ , that maps key values to non-negative integers (or to table index values).

Ideally, the hash function,  $H$ , can be used to determine the location (table index) of any record, given its key value.

For example, student records for a class could be stored in an array  $C$  of dimension 10000 by truncating the student's ID number to its last four digits:

$$H(\text{IDNum}) = \text{IDNum} \% 10000$$

Given an ID number  $X$ , the corresponding record would be found at  $C[H(X)]$ .

Of course, it can't really be that simple... what obvious problem will probably occur?

## The Birthday Problem

Hashing 2

Suppose you have a group of  $N$  people. Using the Gregorian calendar and disregarding February 29, each person in the group has a birthday between January 1 and December 31 (or 0 and 364 if you prefer).

Of course, it's possible that two or more people will have the same birthday.

Question: how large must the group be in order for the probability that there are two or more people with the same birthday to be  $1/2$  or higher?

Surprise answer: 23 people.

In hashing, this is known as a hash collision...it's OK for two people to have the same birthday, but you can't store two different data items in the same physical space.

The fundamental problems in hashing fit into two categories:

- What properties should a hash function have? And how can I pick or design a function that has those properties?
- How can I deal with the problem of collisions in the hash table?

We will first consider the issues of hash function design...

A good hash function should:

- be easy and quick to compute
- achieve an even distribution of the key values that actually occur across the index range supported by the table

Typically a hash function will take a key value and:

- chop it up into pieces, and
- mix the pieces together in some fashion, and
- compute an index that will be uniformly distributed across the available range.

Note: hash functions are NOT random in any sense.

## Truncation:

- ignore a part of the key value and use the remainder as the table index
- e.g., 21296876 maps to 976

## Folding:

- partition the key, then combine the parts in some simple manner
- e.g., 21296876 maps to  $212 + 968 + 76 = 1256$  and then mod to 256

## Modular Arithmetic:

- convert the key to an integer, and then mod that integer by the size of the table
- e.g., 21296876 maps to 876

It is usually desirable to have the entire key value affect the hash result (so simply chopping off the last k digits of an integer key is NOT a good idea in most cases).

Consider the following function to hash a string value into an integer range:

```
unsigned int strHash(string toHash) {  
  
    unsigned int hashValue = 0;  
    for (int Pos = 0; Pos < toHash.length(); Pos++) {  
        hashValue = hashValue + int(toHash.at(Pos));  
    }  
    return hashVal;  
}
```

```
Hashing: hash  
h: 104  
a: 97  
s: 115  
h: 104  
Sum: 420
```

```
Mod by table  
size to get the  
index: 33
```

This takes every element of the string into account... a string hash function that truncated to the last three characters would send "hash", "stash", "mash", "trash", etc., all to the same table index.

In most cases the table size is used to mod a computed value to obtain the final index value.

This is a simple way to guarantee that the computed index value is in the desired range.

This can also pose problems if the table size is poorly chosen. For example, suppose the table size is 1000. The mod operation will produce an index value that depends only on the three low order digits of the previous value. It's less obvious, but similar issues arise if the table size is a power of 2 (or any other small integer).

Generally, making the table size a prime integer produces better results by reducing the probability of "clumping" of index values.

Any usable hash function is likely to map two or more key values to the same index, in at least some cases.

A little bit of design forethought can often reduce this:

```
unsigned int strHash(string toHash) {
    unsigned int hashValue = 0;
    for (int Pos = 0; Pos < toHash.length(); Pos++) {
        hashValue = 4*hashValue + int(toHash.at(Pos));
    }
    return hashVal;
}
```

**Hashing: hash**

h: 104

a: 97

s: 115

h: 104

**Sum:** 8772**Index:** 0

The original version would have hashed both of these strings to the same table index.

Flaw: it didn't take element position into account.

**Hashing: shah**

s: 115

h: 104

a: 97

h: 104

**Sum:** 9516**Index:** 13

## Resolving Collisions

Hashing 9

Despite careful design, it is almost always true that a hash function will create some collisions.

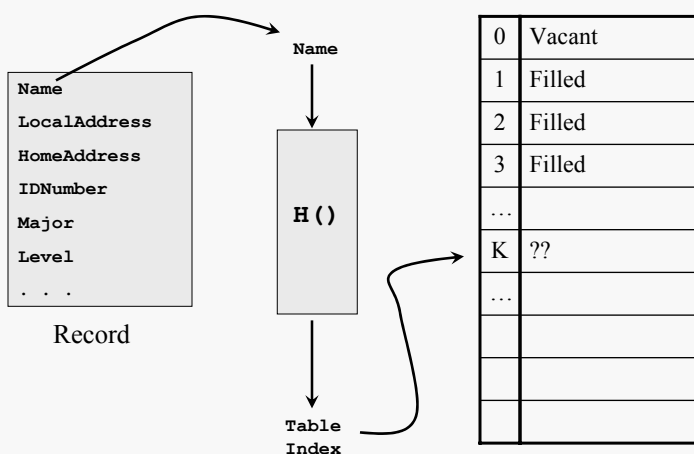
When that happens the hash table implementation must provide some mechanism to resolve the collision:

- No strategy. Just reject the insertion. Unacceptable.
- Linear Probing: start with the original hash index, say  $K$ , and search the table sequentially from there until an empty slot is found. If no empty slot is found...
- Quadratic Probing: search from the original hash index by considering indices  $K + 1$ ,  $K + 4$ ,  $K + 9$ , etc., until an empty slot is found (or not...).
- Other: apply some other strategy for computing a sequence of alternative table index values.

## Hash Table Insertion

Hashing 10

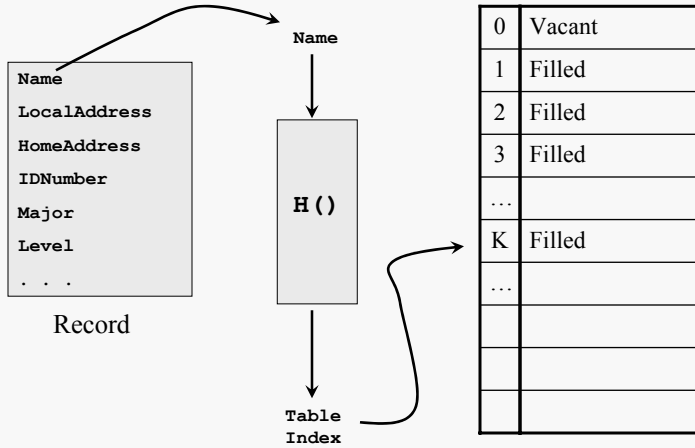
Simple insertion of an entry to a hash table involves two phases:



First the appropriate record key value must be hashed to yield a table index...

# Hash Table Insertion

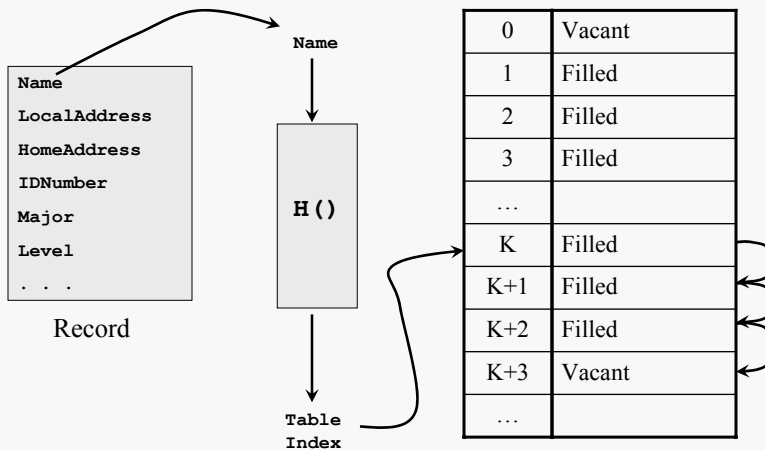
If the hash location is vacant, then the record may be stored there and we are done:



If the hash location is filled, then a third phase is required: we must find another location for the record...

# Linear Probing

If the hash location is not vacant, then:



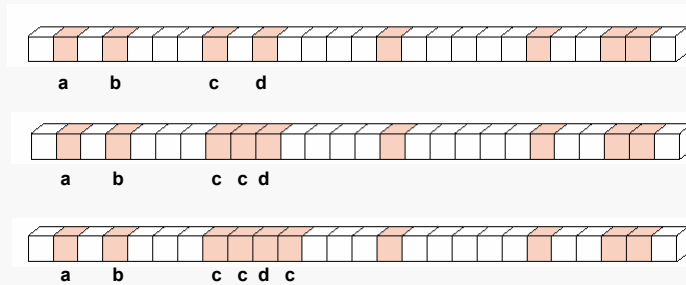
If the hash location is filled, then we must find another location for the record...

## Clustering

Hashing 13

Linear probing is guaranteed to find a slot for the insertion if there still an empty slot in the table.

However, linear probing also tends to promote clustering within the table:

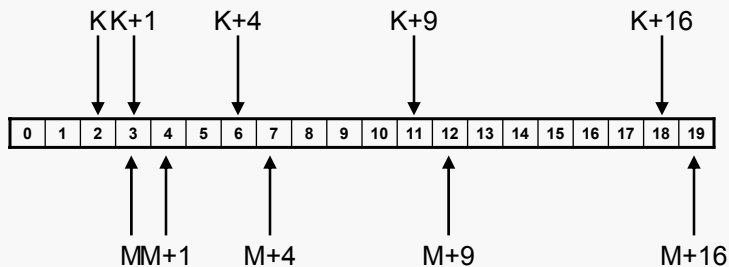


The problem here is that the probabilities that a slot will be hit are no longer uniform. If there are  $N$  slots in the table, the probability that the slot just after the last 'c' will be hit next is  $5/N$  instead of the ideal  $1/N$ .

## Quadratic Probing

Hashing 14

Quadratic probing is an attempt to scatter the effect of collisions across the table in a more distributed way:



Now, the probe sequences from near misses don't overlap completely.

Problem: will this eventually try every slot in the hash table? No. If the table size is prime, this will try approximately half the table slots.

## General Increment Probing

Hashing 15

Quadratic probing can be generalized by picking some function, say  $S(i)$ , to generate the step sizes during the probe. Then we have the index sequence:

$$K, K + S(1), K + S(2), K + S(3), K + S(4), \text{ etc.}$$

Letting  $S(i) = i^2$  yields quadratic probing.

The primary concern is that the probe sequence not cycle back to  $K$  too soon, because once that happens the same sequence of index values will be generated a second time.

## Key Dependent Probing

Hashing 16

It also seems reasonable to use a probe function that takes into account the original key value:

$$K, S(1, K), S(2, K), S(3, K), S(4, K), \text{ etc.}$$

If done well, this could prevent the adjacencies that increment probing creates in the probe sequences for adjacent slots (see slide 14 again).

One must be careful to make sure that the calculation of the probe indices is relatively cheap, and that the probe sequence covers a reasonable fraction of the table slots.

The more complex the function used becomes, the harder it is to satisfy those concerns.

Aside from the theoretical issues already presented, there are practical considerations that will influence the design of a hash table implementation.

- The table should, of course, be encapsulated as a template.
- The size of the table should be configurable via a constructor.
- The hash function does NOT, perhaps surprisingly, logically belong as a member function of the hash table template.

The table implementation should be as general as possible, but the choice of a particular hash function should take into account both the type and the expected range of the key values.

Hence, it is natural to make the choice of hash function the responsibility of the designer of the data element (key) being hashed. From this perspective, the table simply asks a data element to hash itself. The hash function may either return an integer, which the table then mods by its size, or it may take the table size as a parameter.

- The probing strategy IS the responsibility of the hash table, since it requires knowing the internal table configuration.

Deleting a record poses a special problem: what if there has been a collision with record being deleted? In that case, we must be careful to ensure that future searches will not be disrupted.

Solution: replace the deleted record with a "tombstone" entry that indicates the cell is available for an insertion, but that it was once filled so that a search will proceed past it if necessary.

Problem: this increases the average search cost since probe sequences will be longer than strictly necessary.

We could periodically re-hash the table or use some other reorganization scheme.

Question: how do tombstones affect the logic of hash table searching.

Question: can tombstones be "recycled" when new elements are inserted?

## A HashTable Class

Hashing 19

Here is a sample interface for a hash table class:

```
enum probeOption {LINEAR, QUADRATIC};

template <typename T, typename H> class HashTableT {
private:
    enum slotState {EMPTY, FULL, TOMBSTONE};
    T*      Table;
    slotState*  Status;
    int     Size;
    int     Usage;
    probeOption Opt;

    unsigned int Probe(int Step);

    // continues . . .
```

Naturally the table is allocated dynamically.

The hash table does not provide a hash function; the second template parameter is required to implement a public function with the interface:

```
unsigned int H::Hash(T);
```

Two client-selectable probe strategy options are provided.

## A HashTable Class

Hashing 20

The public interface provides the expected functions, as well as table resizing:

```
// . . .continued . . .

public:
    HashTableT(unsigned int Sz = 97, probeOption O = LINEAR);
    HashTableT(const HashTableT<T, H>& Source);
    HashTableT<T, H> operator=(const HashTableT<T, H>& RHS);
    ~HashTableT();

    T* Insert(const T& Elem);
    T* Find(const T& Elem);
    T* Delete(const T& Elem);

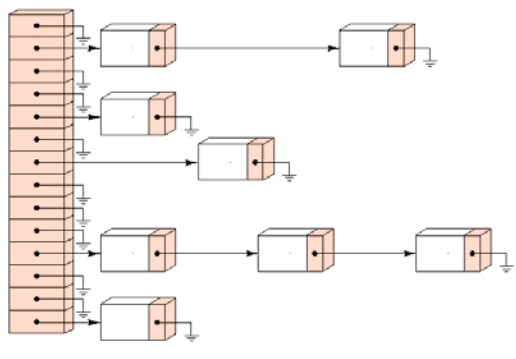
    void Clear();
    bool Resize(unsigned int Sz);
    unsigned int tableSize() const;
    void Display(ostream& Out);
};
```

For testing purposes it would also be natural to provide instrumentation.

## Alternative: Open Hashing

Hashing 21

The clustering problem can also be handled by viewing each slot in the table as a sort of "bucket" into which a number of records may be dropped:



Here, the "buckets" are linked lists which could hold any number of colliding records. Alternatively each table slot could be large enough to store several records directly... in that case the slot may overflow, requiring a fallback...

## Hash Table as an Index

Hashing 22

The entries in the hash table do not have to be data records.

For example, if we have a large disk file of data records we could use a hash table to store an index for the file. Each hash table entry might store a key value and the byte offset within the file to the beginning of the corresponding record.

Or, the hash table entries could be pointers to data records allocated dynamically on the system heap.

# A Better String Hash Function

Consider the following function to hash a string value into an integer:

```
unsigned int elfHash(const string& toHash) {  
  
    unsigned int hashVal = 0;  
    for (int Pos = 0; Pos < toHash.length(); Pos++) { // use all elements  
  
        hashVal = (hashVal << 4) + int(toHash.at(Pos)); // shift/mix  
  
        unsigned int hiBits = hashVal & 0xF0000000; // get high "nibble"  
  
        if (hiBits != 0) {  
            hashVal ^= hiBits >> 24; // xor high nibble with second nibble  
        }  
  
        hashVal &= ~hiBits; // clear high nibble  
    }  
  
    return hashVal;  
}
```

# Details

Here's a trace:

| Character     | hashVal  |
|---------------|----------|
| d: 64         | 00000064 |
| i: 69         | 000006a9 |
| s: 73         | 00006b03 |
| t: 74         | 0006b0a4 |
| r: 72         | 006b0ab2 |
| i: 69         | 06b0ab89 |
| b: 62         | 0b0ab892 |
| u: 75         | 00ab8925 |
| t: 74         | 0ab892c4 |
| i: 69         | 0b892c09 |
| o: 6f         | 0892c04f |
| n: 6e         | 092c05de |
| distribution: | 15       |

|                   |            |
|-------------------|------------|
| hashVal           | : 06b0ab89 |
| hashVal << 4:     | 6b0ab890   |
| add 62            | : 6b0ab8f2 |
| hiBits            | : 60000000 |
| hiBits >> 24:     | 00000060   |
| hashVal ^ hiBits  | : 6b0ab8f2 |
| hiBits            | : 00000060 |
| hashVal & ~hiBits | : 6b0ab892 |

|    |      |
|----|------|
| f: | 1111 |
| 6: | 0110 |
| ^: | 1001 |