HASH TABLES

TABLE OF CONTENTS
1. Introduction to Hashing
2. Java Implementation of Linear Probing
3. Maurer’s Quadratic Probing
4. Double Hashing
5. Separate Chaining
6. Hash Functions
7. Alphanumeric Keys
8. Hashing Efficiency
9. Summary of Major Hashing Strategies
10. Expert Recommendations on Hashing Techniques

Introduction to Hashing
Why Use Hash Tables? (page 519)
- The average running time of the find and insert operations can be held at close to O(1) under strict conditions.

Disadvantages of Hash Tables (page 519)
- Both Open and Closed Addressing:
  - It is difficult to traverse the items in sorted order.
- Open Addressing (closed hashing) only:
  - It uses an array and therefore has a fixed size, which can only be increased with some difficulty.
  - It is difficult to delete items from it.
  - Slots in the table must be initialized to be “empty” so that an empty slot can be detected.
  - There are a number of conditions on the table size, which your implementation may not be able to maintain.
  - As the table fills up, the running time slows to O(n) dropping quickly after the table reaches 50% full.

Table organization (pages 520 – 521)
A hash table can have a two-dimensional array structure. It has B rows named buckets, which are numbered 0 through B–1. Each bucket has S columns named slots, which are numbered 0 through S–1. Most often, S = 1 making the hash table a one-dimensional array.

Hashing Functions (pages 525 – 526)
- Assume that database records are being stored in the hash table and that each record has a primary key. A hashing function $f$ inputs a value for the key and returns an index (0 through B–1) within the hash table. The index returned by $f$ is called the hash address or hash code.
When the key values are integers, such as Social Security numbers or credit card account numbers, the simplest hashing function scales the key to the range 0 through $B-1$ by dividing it modulo $B$:

```java
public int f(int key)
{
    return key % B;
}
```

Algorithm for operations find and insert (page 527 – 528)
Suppose you wish to insert a record with key $X$ into a hash table. Compute $f(X) = b$ and search the slots in bucket $b$ for a record with key $X$. If the record is found, then there is no need to insert it since it is already in the table. If the record is not found and the bucket has an empty slot, then place it into the first available slot within the bucket. If the record is not found and the bucket has no empty slots, then a collision has occurred and a collision resolution (or probing) strategy must be employed.

Linear Probing (page 528)
- The simplest probing strategy is linear probing, in which you successively search buckets $b+1$, $b+2$, $b+3$, etc., wrapping around to bucket 0 if necessary, until:
  - the record with key $X$ is found, meaning you’ve found the record, OR
  - an empty slot is encountered, in which case the record is not in the table and so you insert it into the table at that position, OR
  - you return to bucket $b$ in which case the table is full and must be increased in size
- Using the same algorithm for inserting and searching guarantees that a record will be found even if it is not placed into the array element given by its hash address.

Hash Workshop Applet (pages 528 – 532)
The Hash Workshop applet demonstrates the Java implementation of a hash table using integer keys, $S = 1$, the division hashing function and linear probing.

Java Implementation of Linear Probing
Implementation Issues (pages 532 – 533)
- To delete an item in a hash table, you must overwrite it with a special value (in the applet *Del*) that indicates the slot is not really empty; otherwise the newly emptied slot interferes with the find algorithm.
- Duplicate keys are not allowed because of the same interference.
- Linear probing suffers from primary clustering (page 533).
  - Large groups of consecutive buckets become occupied.
  - Any key whose hash address lies within such a cluster requires excessive probing after which it will be inserted at the end of the cluster, making the cluster (and subsequent probe sequences) even larger.
  - Primary clustering reduces the performance of linear probing until, at worst case, it becomes an $O(n)$ linear search.
Methods **find, insert** and **delete** (pages 533 – 535)

Although your author’s implementation is very clear, the mod (%) operation to accomplish the wraparound is inefficient. You should substitute it with code that subtracts $B$ from **hashVal** if it is past the array range.

Class **HashTable** in the **hash.java** program (pages 535 – 540)

In the hash table constructor (page 536) your author takes advantage of the fact that Java initializes the hash array elements to **null**. In other languages (such as C++) you must do that with a loop.

Rehashing (page 541)

- If the hash array becomes too full, you *cannot* simply create a larger one and copy the contents of the old array into it.
- This is because a hash function calculates the bucket address based on the array size $B$ and so a key will not necessarily hash to the same bucket address in the new array as it did within the old array.
- The process of enlarging the hash array size is called **rehashing**.
- Here’s how it works:
  - Allocate a new array with the larger size.
  - Modify the hashing function to account for the larger size.
  - For each record in the current array insert it into the new array using the modified hashing function.

**Maurer’s Quadratic Probing** (pages 542 – 544)

**Quadratic probing** prevents primary clustering by probing widely separated buckets instead of adjacent buckets.

The quadratic probing technique described in your textbook is one of the earliest and was described by W. D. Maurer in 1968.¹

It works as follows. If bucket $b$ is full, then successively search buckets $b+1$, $b+4$, $b+9$, $b+16$, etc. As usual, the bucket index must be divided modulo $B$ so that it is properly scaled within the range of array indices.

The code to implement quadratic probing is not in your textbook. The algorithm would be very similar to that of the linear probe on page 533. The only difference is that within the while loop that performs the probe, change **hashVal++** to **hashVal+=2*k+1** where $k$ is a variable initialized to 0 and incremented at the end of each loop cycle. Also, as with the linear probing algorithm substitute the use of % with code that subtracts $B$ from **hashVal** if it is past the array range.

It is clear that linear probing inspects consecutive buckets until (a) finding the sought key, (b) finding an empty slot or (c) finding that the table is full. It is not clear that the same thing happens with quadratic probing – that the find algorithm doesn't fall into an infinite loop of searching the same buckets over and over again. The following theorem,

the proof of which is found in Maurer’s article, gives the conditions under which items are safely inserted:\footnote{Your textbook, on page 544, states the table size requirement but neglects to mention that the table be no more than half full.}

**Maurer’s Theorem:** If table size $B$ is prime and the table is no more than half full, then the quadratic probing sequence $b, b+1, b+4, b+9, b+16$, etc. will not probe the same bucket twice and a new record can always be inserted.

Although quadratic probing solves the primary clustering problem, it suffers from the problem of secondary clustering (page 544). This means that any two keys with the same hash address follow the same probe sequence so that each additional collision at that address will initiate a probe sequence that is one greater in length.

**Double Hashing (pages 544 – 552)**

- **Double hashing** is an open addressing method that eliminates both forms of clustering and is no harder to code.
- Here’s how it works. If bucket $b$ is full, then successively probe buckets $b+h, b+2h, b+3h$, etc., where $h$ is an increment value given by a secondary hashing function. The idea is that the probe sequence depends on the key rather than being the same for every key. Thus, even if two keys hash to the same bucket address $b$ using the primary hashing function, they won’t hash to the same increment $h$ using the secondary one and therefore, their probe sequences will differ.
- *Do not work* with the HashDouble applet in double mode (page 545) as it is not clear what the author is using for his secondary hashing function.
- As with quadratic probing, we need a guarantee that we do not keep searching the same buckets over and over again (see page 551). The following theorem provides it:

**Double Hashing Theorem:** If table size $B$ is relatively prime to the increment value $h$, then the double hashing probing sequence $b, b+h, b+2h, b+3h$, etc. will eventually examine every bucket of the hash table.

Thus, the secondary hashing function must return an increment $h$ that is:

1. Different from the primary one
2. Greater than 0 and less than the table size $B$
3. Relatively prime to the table size $B$

The easiest way to insure the third requirement is to make the table size $B$ prime. If $c$ is a constant smaller than the table size $B$, then the other two requirements are guaranteed by the following function:

```java
public int η( int key )
{
    return c - (key % c);
}
```
Your author (on page 545) says that the constant $c$ must be prime. He chooses $c = 5$ to get this secondary hashing function shown on page 547:

```java
public int hashFunc2( int key )
{
    return 5 - (key % 5);
}
```

Weiss\(^3\) also suggests choosing a prime number for $c$ but doesn’t imply that it is necessary. On the other hand, Sedgwick\(^4\) suggests choosing $c = 8$ so that you can efficiently implement the secondary hashing function by simply extracting the rightmost three bits of the key.

The code to implement double hashing is on pages 546 – 548. The probing loops within methods `find()`, `insert()` and `delete()` are identical to those of linear probing except for the increment value. Again, for greater efficiency you should substitute the mod (\%\) operation to accomplish the wraparound with code that subtracts $B$ from `hashVal` if it is past the array range.

**Separate Chaining (pages 552 – 561)**

- In *separate chaining* each bucket of the hash table is a linked list.
- To insert a record with key $X$ into the hash table compute $f(X) = b$ and search the linked list at bucket $b$ for the record, inserting it into the list if it isn’t already present.

**Hash Functions (pages 561 – 563)**

- A good hashing function has two properties.
  - 1. It should have quick running time so that you don’t lose efficiency and destroy the advantage of using a hash table.
  - 2. It should uniformly distribute the keys across the entire range of possible hash addresses.
- The theoretical ideal is a *uniform hashing function*, which, given a randomly selected key $X$, hashes it to any bucket address with equal probability.
- Several factors affect the uniformity of hashing functions, including:
  - The nature of the hashing function.
  - The nature of the keys.
  - The size $B$ of the hash table.

---


Alphanumeric Keys (pages 563 – 566)
For databases that use alphanumeric keys (such as VIN numbers), the string must be converted to an integer before or during the application of the hashing function. The conversion algorithm uses the fact that each character of a string is stored as an integer. A common integer code is the ASCII code (American Standard Code for Information Interchange) where each character is stored as an eight-bit binary number. For example:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Decimal</td>
</tr>
<tr>
<td>A</td>
<td>65</td>
</tr>
<tr>
<td>B</td>
<td>66</td>
</tr>
<tr>
<td>E</td>
<td>69</td>
</tr>
<tr>
<td>F</td>
<td>70</td>
</tr>
</tbody>
</table>

The best strategy guarantees that no two different strings are converted to the same integer, thus insuring that you haven't introduced any non-uniformity into the hashing scheme through the string to integer conversion.

Any string $s_n s_{n-1} \ldots s_1 s_0$ can be converted to a unique integer using the formula:

$$v(s_n) r^n + v(s_{n-1}) r^{n-1} + \ldots + v(s_1) r + v(s_0)$$

$v(s_k)$ is the integer encoding for character $s_k$ and $r$ is the radix of the character set. Since ASCII has 256 characters (represented from 00000000 to 11111111) its radix is 256.

**Example:** For string ABE, $A = s_2$, $B = s_1$, $E = s_0$. Thus,

$$v('A') r^2 + v('B') r + v('E') =$$
$$65 \cdot 256^2 + 66 \cdot 256 + 69 =$$
$$4,276,805$$

Effectively this algorithm simply stores the ASCII codes for each letter consecutively. For example, the binary representation of 4,276,805 is 0100 0001 0100 0010 0100 0101.

\[0100\ 0001\ 0100\ 0010\ 0100\ 0101\]

\[A\quad B\quad E\]

Since $256 = 2^8$, multiplying a binary number by 256 shifts it 8 bits to the left, multiplying by $256^2$ shifts it 16 bits to the left, and so on. Thus, the algorithm effectively shifts each character’s ASCII code to its corresponding position within the string and adds the results together.
Without improvement, this algorithm likely violates our requirement of having a quick running time. For a string of length \( n \), the polynomial \( v(s_n) r^n + v(s_{n-1}) r^{n-1} + \ldots + v(s_1) r + v(s_0) \) has \( n+1 \) terms. If we calculate the powers by successive multiplication then the total computation time is \( O(n^2) \). We could use *Horner’s method* to calculate the polynomial, which has time complexity \( O(n) \). Also, to prevent integer overflow resulting from the computation of such large integers, we incorporate the division of the result by the table size into each step of the computation (using \( \% \) \( B \)). The resulting algorithm looks like this coded in Java:

```java
public int hashString( String key )
{
    int hashVal = 0;
    for ( int j=0; j<key.length(); j++ )
    {
        int letter = key.charAt( j );
        hashVal = (( hashVal * 256 ) + letter ) % B;
    }
    return hashVal;
}
```

The call `key.charAt(j)` returns the ASCII code of the character in string `key` at position `j`. Your textbook (on page 565) presents the same algorithm as `hashString`. His algorithm uses a radix of 27 and subtracts 96 from the ASCII code because he is assuming keys are made up only of blanks and the twenty-six lower-case letters. The ASCII code for the lower-case letter ‘a’ is 97. Thus, by subtracting 96 from the ASCII code in the first statement, he scales each lower-case letter to an integer in the range 1 through 26.
Hashing Efficiency (pages 566 – 571)

The book presents a number of graphs that compare the performance of the probing strategies. These graphs present the theoretical best possible performance for each strategy because they assume randomly selected keys that are hashed to any bucket address with equal probability.

Summary of Major Hashing Strategies

<table>
<thead>
<tr>
<th>Class</th>
<th>AKA</th>
<th>Characteristic</th>
<th>Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Addressing</td>
<td>Closed</td>
<td>If a collision occurs then alternate buckets are tried until an empty one is</td>
<td>Linear Probing Quadratic Probing Double Hashing</td>
</tr>
<tr>
<td></td>
<td>Hashing</td>
<td>found.</td>
<td></td>
</tr>
<tr>
<td>Closed Addressing</td>
<td>Open</td>
<td>If a collision occurs then the new item is inserted into a linked list at the</td>
<td>Separate Chaining</td>
</tr>
<tr>
<td></td>
<td>Hashing</td>
<td>bucket address.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Probing Sequence</th>
<th>Requires</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linear Probing</strong></td>
<td>(b, b+1, b+2, b+3), etc.</td>
<td></td>
<td>Primary clustering</td>
</tr>
<tr>
<td><strong>Maurer’s Quadratic Probing</strong></td>
<td>(b, b+1, b+4, b+9, b+16), etc.</td>
<td>Table cannot exceed half full Table size must be a prime number</td>
<td>Secondary clustering</td>
</tr>
<tr>
<td><strong>Double Hashing</strong></td>
<td>(b, b+h, b+2h, b+3h), etc.</td>
<td>Table size must be a prime number Possible restrictions on secondary hashing function</td>
<td>Computation cost of secondary hashing function</td>
</tr>
</tbody>
</table>

Expert Recommendations on Hashing Techniques

- Both your author (on page 552) and Sedgewick\(^5\) claim that double hashing is the open addressing method of choice.
- Weiss\(^6\), on the other hand, refers to double hashing as “theoretically interesting” and recommends quadratic probing.
- Sedgewick\(^7\) recommends:
  - Use separate chaining if the number of records to be placed into the table is unknown.
  - If you can roughly estimate in advance the number of records and the size of the keys then use double hashing and make the table about twice as large as the number of records.
  - Never let an open addressing hash table become more than 60% full.
  - If it reaches 60% full, rehash it to at least twice its current size.