Hashing

August 7, 2020

1 Hashing

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/HashIntro.html

1.1 Introduction

- method for storing and retrieving records from a database
- lets you search/insert/delete records based on key
- when properly implemented, these operations can be done in $O(1)$
  - as opposed to $O(logn)$ taken by binary search or BST
- hashing principle is simple but proper implementation is difficult

1.2 Terminologies

1.2.1 Hash Table

- records are stored in an array called **hash table**, let's say **HT**

1.2.2 Hash Function

- finds the position of search key K in HT containting the record associated with K
- the goal is to arrange things such that, for any K and hash function $hash$, $i = hash(K)$
- locations are usually numbered from 0 to **N-1**
- See: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/HashIntro.html for a simple demo

1.3 Pros and Cons of Hashing

- not good for applications with duplicate keys or multiple records with same key
- not good for answering range queries; e.g., retrieve all records where key >= value and <= another_value
- great choice for exact key matching
- suitable for both in-memory and disk-based searching (most databases use this technique besides B-tree)

1.4 Hash Function Principles

- when large range of values are hashed and stored into small number of slots, collision will likely occur

- collisions occurs when two records hash to the same slot/index in the table
- E.g. Birthday Paradox: if there are 23 students in a class, there's 50% probability that two will share a birthday
  - even though there are 365 days; 100% when there are 367
- **HT** collusion depends on the distribution of the keys which we typically do not have control over
- generally, pick a hash function that maps keys to slots in a way that makes each slot in the hash table have equal probability of being filled for the actual set of keys being used

1.5 Sample Hash Functions

1.5.1 1. Simple Mod Function

- hash function to store integers in a table with 10 slots

[2]:
```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono> //sleep thread
#include <thread>
#include <unistd.h>

using namespace std;
```

[3]:
```
// 10 slots: 0-9
int intHash(int num) {
    return num%10;
}
```

[4]:
```
int nums[10] = {};
```

[5]:
```
int getRandomNumber() {
    srand(time(NULL)); // use current time to generate random number
    return rand()%100;
}
```

[8]:
```
// get 10 random numbers and insert into nums table
for (int i=0; i<10; i++) {
    int num = getRandomNumber();
    // wait for a second so; we get new random number
    //std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    usleep(1000); // sleep for microseconds
    int index = intHash(num);
    nums[index] = num;
}
```

[9]:
```
// see all the numbers stored
for (int i=0; i<10; i++) {
```

```
    cout << nums[i] << " ";
}
```

```
30 81 2 23 74 95 16 37 88 79
```

1.5.2 2. Binning

- if keys are in the given range say 0 to 999, and we've a hash table of size 10
- simply divide the key value by 100
- so all keys in the range 0-99 -> 0, 100-199 -> 1, and so on
  - record with K is stored at index K/X from some value X (using integer division)
- this technqiue is called binning

1.5.3 3. Mid-Square Method

- good hash function for integer key values
- method:
  1. square the key value
  - take out the middle r bits of the result
    - gives the value in the range 0 to $2^r - 1$
- e.g.: say records are 4-digit numbers in base 10
  - the goal is to hash these key values to a table of size 100 (range of 0-99)
  - equivalent to 2 digits in base 10; so $r = 2$
  - say $K = 4567$
  - $K^2 = 20\mathbf{57}489$ - the highlighted middle two digits is the index
  - middle digits are affected by everydigit of the original key value

1.6 Simple Hash Function for Strings

- sum of ASCII value of all the characters in the string

[ ]:
```
#include <cstring>
#include <string>
#include <iostream>
using namespace std;
```

[ ]:
```
// hash str to table of size M
int strHash(string str, int M) {
    int sum = 0;
    for (char c: str)
        sum += int(c);
    return sum%M;
}
```

[ ]:
```
cout << strHash("Hello World!", 5);
```

[ ]:
```
cout << strHash("John Legend", 5);
```

1.6.1 String Folding

- process string N bytes/chars at a time
- integer values for the N-byte chunks are added together
- convert the resulting sum to 0-M using modulus operator

[ ]:
```
// use folding on a string, summed 4 bytes at a time
int strHashFold(string str, int M) {
    long long unsigned int sum = 0;
    int mul = 1;
    for (int i=0; i<str.size(); i++) {
        mul = (i%4 == 0)? 1: mul*256;
        sum += int(str[i]) * mul;
    }
    return sum%M;
}
```

[ ]:
```
cout << strHashFold("Hello World!", 5);
```

[ ]:
```
cout << strHashFold("John Legend", 5);
```

[ ]:
```
cout << strHashFold("this is a long sentence!", 5);
```

1.7 Collision Resolutions

- two classes: Open Hashing and Closed Hashing
- main difference is:
  - open hashing: collisions are stored outside the table
  - close hashing: collisions are stored in the table in one of the available slots

1.8 Open Hashing

- technique that tries to minimize collisions
- also called separate chaining
- e.g., vector of linked list where each records with duplicate keys are pushed backed into the linked list the key is hashed into
- records within the slot's list can be ordered in several ways:
  - insertion order; key value order; frequency of access, etc.
  - ordering provides an advantage in the case of an unsuccessful search (stop early)
- open hashing is most appropriate when the hash table is kept in main memory

1.9 Closed Hashing

- bucket hashing is used to store all records directly in the hash table
- each record, R has a **home position**; i.e., $\mathbf{h}(K)$
- if another record alread occupies where R needs to be inserted, use collision resolution policy to determine another available slot in the table

1.9.1 Bucket Hashing

- group hash table into **buckets**
- M slots of hash table are divided into B buckets; each bucket with M/B slots
- hash function assigns each record to the first slot within the one of the buckets
  - if this slot is already occupied, bucket slots are searched sequentially until an open slot is found
  - if a bucket is entirely full, the record is stored in **overflow bucket** of infinite capacity at the end of the table
  - all buckts share the same overflow bucket
  - a good implementation will use a hash function that distributes the records evenly among the buckets so that as few records as possible go into the overflow bucket
- Visualize bucket hashing demo here: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BucketHash.html#id1

1.9.2 Alternate Approach

- pretend there are no buckets; so use N whole slots as home position
- if the home position is full, then search through the rest of the bucket to find an empty slot
- his reduces initial collisions as each slot can be a home position rather than just the first slot in the bucket
- visualize alternate closed hashing technique here: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BucketHash.html#an-alternate-approach

[ ]:

```

```