Introduction to Hash Functions

July 2011

Introduction To Hashing

Suppose that we want to store 10,000 students records (each with a 5-digit ID(say **102345**)) in a an array.

Possible storage structures could be.

- A linked list implementation which would take **O(n)** time. (increase with increase in n)
- A height balanced tree which would give **O(log n)** access time. ((increase by log n)
- Using an array of size 100,000 which would give **O(1)** access time but will lead to a lot of space wastage.
- *The fastest and most efficient way (that we could get O(1) access without wasting a lot of space) is* ***hashing.***

Introduction To Hashing

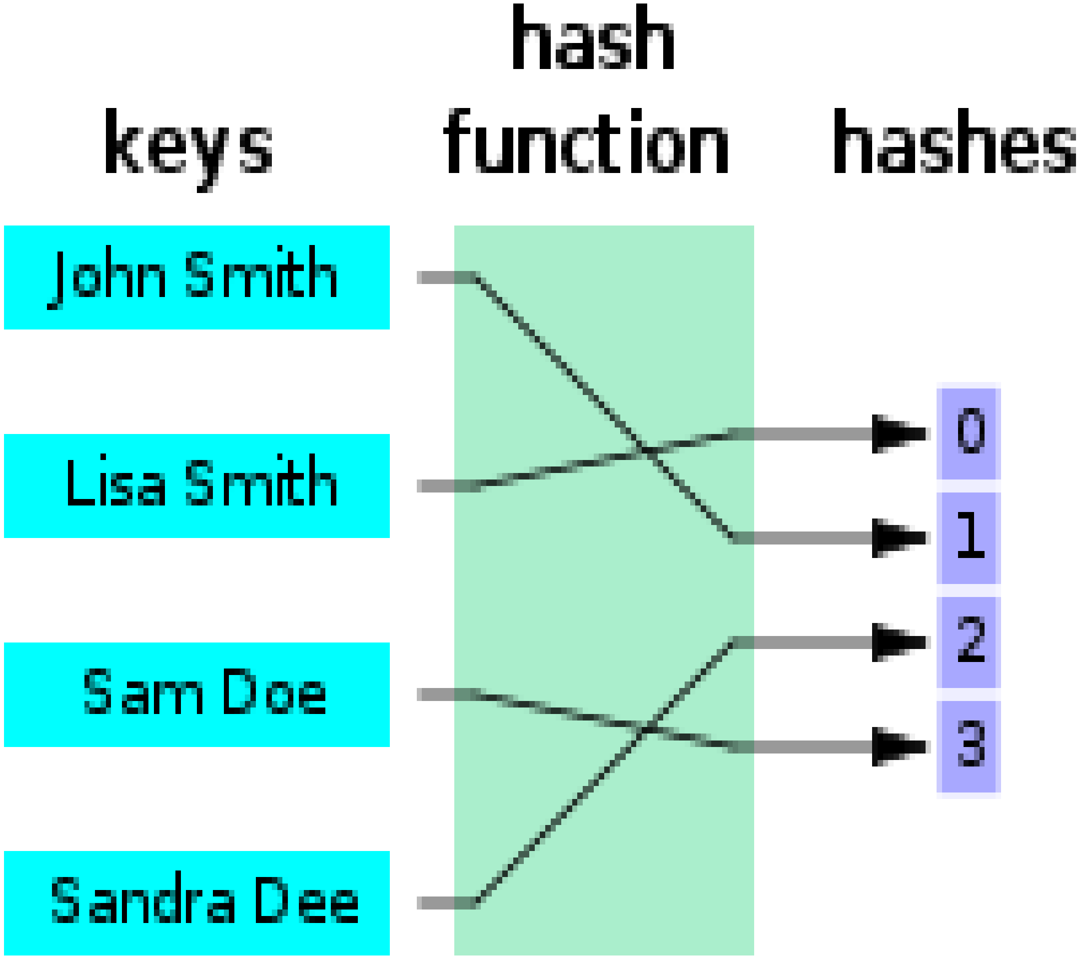
Definition

Hashing is the **use of functions** that receive an input data and use it to **generate fixed-length** output data that acts as a **shortened reference to the original input data**.

This is useful when the original data is too cumbersome to use in its entirety.

- The 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) % Tablesize.**
- Where **k** is the input parameter to the hash function h() and **tablesize** is the size of the array.

Introduction To Hashing



Example1

Suppose we have a table(an Array) of size 97 and wish 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, 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

Introduction To Hashing

Hashing Example 2

A dictionary based on a hash table for: items (SSN, **Name**)

700 persons in the database

We choose a hash table of size N = 1000 with hash function **h(x) = last three** digits of x

0	(025-611-000, Mr. X)
1	
2	
3	(987-067-003,Wayne John)
4	
......	
997	(431-763-997, Alan Okongo)
998	(007-007-998, Jane, Wangui)

Introduction To Hashing

Collision

- It is possible for different keys to hash to the same array location.
- This situation is called **collision** and the colliding keys are called **synonyms.**
  - if for **k1 , k2 h(k1) = h(k2)** then collision occur.
  - **A good hash function should:**
    - Minimize collisions.
    - Be easy and quick to compute.
    - Distribute key values evenly in the hash table.
    - Use all the information provided in the key.

Introduction To Hashing

Collision

0		(987-067-000,Mr. Z)
1		
2		
3		**(025-611-003, Mr. Y) ? (123-456-003, Mr. H)**
4		(987-067-004,Mr. T)

Introduction to Hashing

Managing Collisions

Different possibilities of handing collisions:

- Linear probing,
- Double hashing,
- Chaining,

Introduction to Hashing

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 **.**

Introduction to Hashing

Linear probing,

The pseudocode

Linear_probing_insert(K)

if (table is full) error

probe = h(K)

while (table[probe] occupied)

probe = (probe + 1) mod M

table[probe] = K

Introduction to Hashing

Linear probing,

The pseudocode

Linear_probing_insert(K)

if (table is full) error

probe = h(K)

while (table[probe] occupied)

probe = (probe + 1) mod M

table[probe] = K

Introduction to Hashing

Linear probing,

Explanation

- *Look ups (walk) along table until the key or an empty slot is found*
- *Uses less memory than chaining don't have to store all those links*
- *Slower than chaining may have to walk along table for a long way*
- *Deletion is more complex*
  - *Either mark the deleted slot or fill in the slot by shifting some elements down* ***(Expensive)***

Introduction to Hashing

Linear probing,

Example 1

h(k) = k mod 13

Insert keys: 18 41 22 44 59 32 31 73

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Introduction to Hashing

Linear probing,

Example 2

Consider a simple example with table of size 10.

After hashing keys 22 , 9 and 43, the table is shown below

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

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

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

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

Linear probing,

Example 2

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

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

Introduction to Hashing

Linear probing,

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.

Introduction to Hashing

Linear probing,

Practical Example

```
Int main(){
int v[]={18,41,22,44,59,32,31,73};
int store2[13];// store
int index=0;
cout<<"Using hashing index"<<endl;
for (int j=0;j<8;j++){
index=p[j]%13;//Geting the index using the
store2[index]=p[j];// storing in the store
}
cout<<"Reading Stored Values"<<endl;
for (int k=0;k<12;k++){
if(store2[k]){
cout<<store2[k]<<",";
}
else
cout<<"NULL"<<",";
}
cout<<endl;
Return 0;
}//end of program
```

Introduction to Hashing

Double Hashing

- *Use two hash functions and the fact that if M is prime, eventually will examine every position in the table*
- *Distributes keys more uniformly than linear probing does*
- *Double hashing uses a secondary hash function d(k):*
- *Handles collisions by placing items in the first available cell*
- ***h(k) + j · d(k)*** *for* ***j = 0, 1, . . . , N − 1.***
- *The size of the table N should be a prime.*

The pseudocode

```
double_hash_insert(K)
if(table is full) error
probe = h1(K)
offset = h2(K)
while (table[probe] occupied)
probe = (probe + offset) mod M
table[probe] = K
```

Introduction to Hashing

Double Hashing

Example

h1(K) = K mod 13

h2(K) = 8 - K mod 8

Where h2(k) is the offset to add

Introduction to Hashing

Double Hashing

Practical Example

```
Int main(){
int probe,offset;
cout<<"Loading values int store using hashing
index"<<endl;
cout<<"K\t probe h(k)\tOffsetd(k)\tProbes"<<endl;
for (int j=0;j<8;j++){
probe=p[j]%13;//Geting the index using the
offset=7-(p[j]%7);
cout<<p[j]<<"\t\t"<<probe<<"\t\t"<<offset<<"\t";
if(store2[probe]>=0){// position is occupied if true
probe=(probe+offset)%13;
cout<<probe<<endl;
}
cout<<probe<<endl;
store2[probe]=p[j];
}
cout<<"Stored
Values"<<endl;
for (int k=0;k<13;k++){
if(store2[k]){
cout<<store2[k]<<",";
}
else
cout<<"NULL"<<",";
}
cout<<endl;
Return 0;
}//end of program
```

Introduction to Hashing

Double Hashing

Loading values int store using hashing index

K	probe h(k)	Offsetd(k)	Probes
18	5	3	8
41	2	1	3
22	9	6	2
44	5	5	10
59	7	4	11

Stored Values

4197744,NULL,22,41,4196240,NULL,32,62,18,31,44,59,73,

Introduction to Hashing

Using Chaining

- The Hash table T is a vector of linked lists i.e Set up Linked lists of items with the same index
- The expected, search/insertion/removal time is O(n/N), provided the indices are uniformly distributed
- The performance of the data structure can be fine tuned by changing the table size N
- Insert element at the head (as shown here) or at the tail
- Key k is stored in list at T[h(k)]

Chaining Example

Consider a table of TableSize = 10 and

h(k) = k % 10

Insert first 10 perfect squares

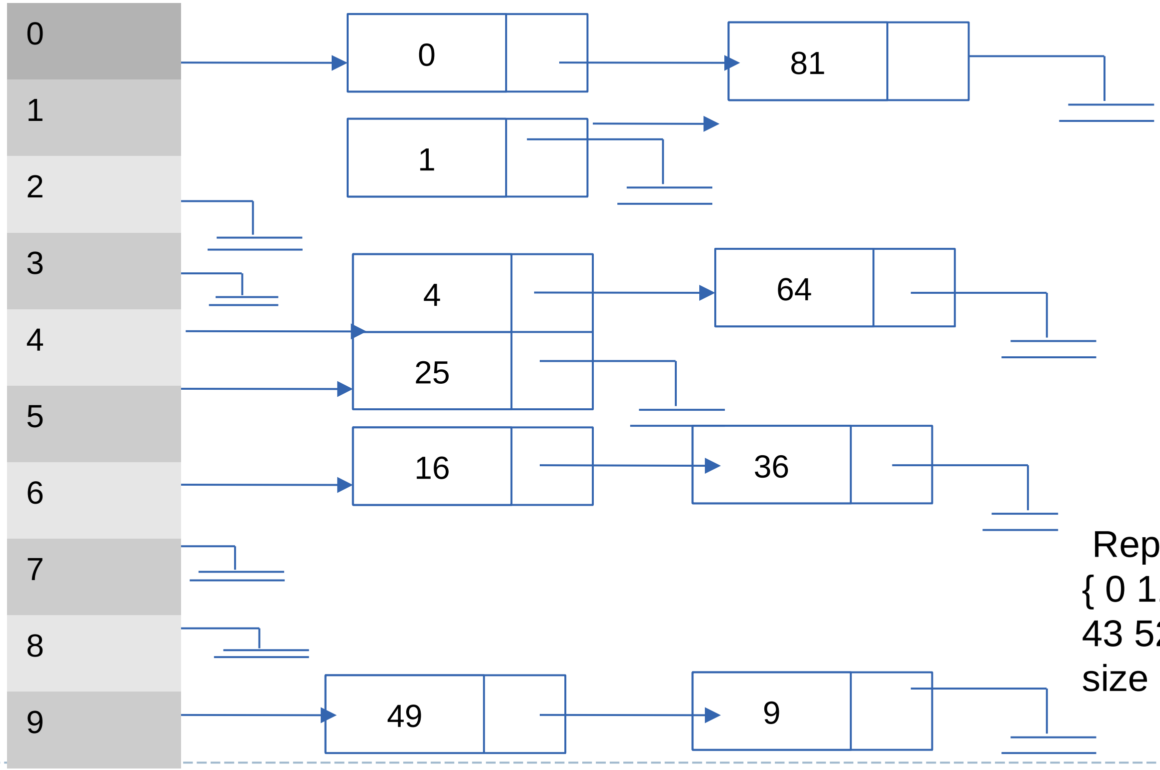
insertion sequence: { 0 1 4 9 16 25 36 49 64 81 }

Introduction to Hashing

Using Chaining

Chaining Example

insertion sequence: { 0 1 4 9 16 25 36 49 64 81 }



TASK

Repeat The task above for the set { 0 11 14 6 19 113 251 16 42 61 81 43 52 67 12} for the same table size

Introduction to Hashing

Common Hashing Functions

Division Remainder (using the table size as the divisor) .

- Computes hash value from key using the % operator.
- Table size that is a power of 2 like 32 and 1024 should be avoided, for it leads to more collisions.
- Also, powers of 10 are not good for table sizes when the keys rely on decimal integers.
- Prime numbers not close to powers of 2 are better table size values.

Truncation or Digit/Character Extraction :

- Works based on the distribution of digits or characters in the key.
- More evenly distributed digit positions are extracted and used for hashing purposes.
- For instance, students IDs or ISBN codes may contain common subsequences which may increase the likelihood of collision.
- Very fast but digits/characters distribution in keys may not be very even.

Introduction to Hashing

Folding

- It involves splitting keys into two or more parts and then combining the parts to form the hash addresses.
- To map the key 25936715 to a range between 0 and 9999, we can:
- Split the number into two as 2593 and 6715 and add these two to obtain 9308 as the hash value.
- Very useful if we have keys that are very large.
- Fast and simple especially with **bit patterns.**
- A great advantage is ability to transform **non-integer keys** into integer values.

Introduction to Hashing

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.

Introduction to Hashing

Mid-Square function:

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.

Use of a Random-Number Generator :

- Given a seed as parameter, the method generates a random number.
- The algorithm must ensure that:
- It always generates the same random value for a given key.
- It is unlikely for two keys to yield the same random value.
- The random number produced can be transformed to produce a valid hash value.

Introduction to Hashing

Hashing: Efficiency Factors

The efficiency of hashing depends on various factors:

1. The Hash function
2. Type of the keys: integers, strings,. . .
3. Distribution of the actually used keys
4. Occupancy of the hash table (how full is the hash table)
5. Method of collision handling in use

The goal of a hash function is to 'disperse' the keys in an apparently random way

Introduction to Hashing

Hash Table: The Load Factor

- The load factor **α** of a hash table is the ratio n/N, that is, the number of elements in the table divided by size of the table.
- High load factor **$\alpha \geq 0.85$** has negative effect on efficiency resulting into :
  - Lots of collisions
  - Low efficiency due to collision overhead