Data Structures – Week #7

Hashing

Outline

- Motivation for Hashing
- Underlying Idea
- Hash Tables
- Hash Functions
- Separate Chaining
- Open Addressing
- Rehashing

Hashing

Motivation for Hashing

- Keeping a data set of dynamic (i.e., rapidly changing) nature in an array is costly.
- Cost of operations such as ***search***, ***insert*** *and* ***remove*** depends upon *how data resides* in the array (i.e., **ordered or not**)
- ***Unordered data*** in array take ***linear time*** to ***search*** and ***remove*** (and ***constant time*** to ***insert***), while
- ***Ordered sequences*** make use of ***binary search*** and can be searched in $O(\log_2 n)$ time, although ***insertion*** and ***removal*** still take $O(n)$, since a ***shift*** operation is required to follow these operations for the data to remain contiguous after these operations.
- The table on the following page summarizes the performance of operations for ordered and unordered data.

Motivation for Hashing

Operation (in arrays)	Unordered Data	Ordered Data
Insert	O(1)	O(n)
Remove	O(n)	O(n)
Search	O(n)	O(log*n*)

Motivation for Hashing

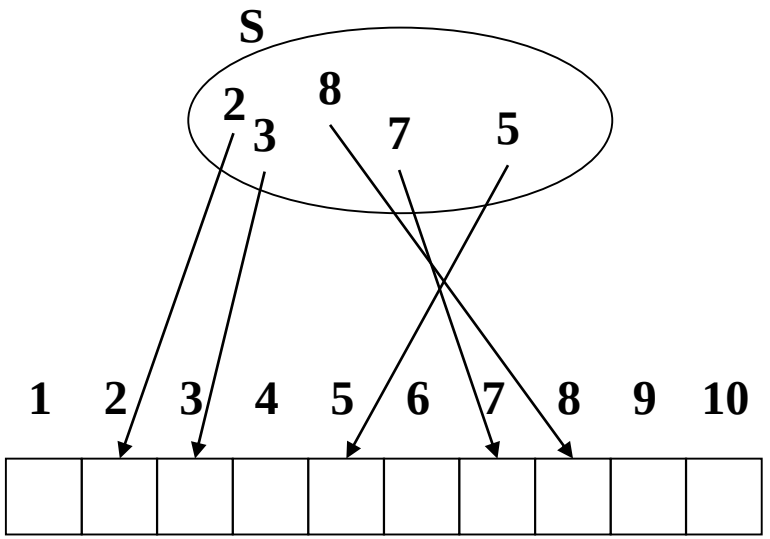
- Question: *May we find a way to perform these operations in average constant time (O(1))*?

- Hash tables or hashing is the answer to the above question.

- In the following pages, we will define what a hash table is.

Underlying Idea

Consider a data set *S={1,…,k}*, ***k* small** (e.g. at most as large as a reasonable array size). You may place each number into the corresponding cell of an array of size *k* using a *one-to-one* mapping. In the figure, this is a linear mapping. This mapping from keys to the array index is called "*direct addressing*."



One-to-one function: *f(key)=key*

Underlying Idea

- To handle many real world case, it is reasonable to assume data is generated from an inexhaustible source. Hence we *assume the infinity of data*.
- Then, no array will be capable of holding the entire data.
- **Solution is to use an array of some sufficient size *m* much less than the original data size *k* and to allow many-to-one mapping**. This array is called a *hash table* and the *many-to-one mapping* is known as the *hash function*. (Check next figure!)

Hash Tables

Here, the data size, *K*, is much greater than the size of the array, *M* or

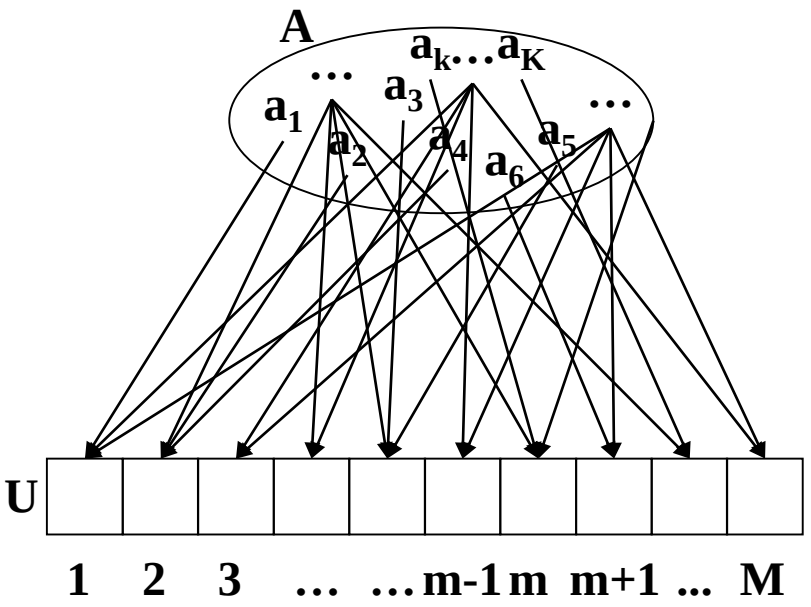
K >> M.

Hence, several keys may get mapped to the same array cell according to the given *many-to-one* mapping mechanism.

The mapping mechanism used is called the hash function.

The attempt to "hash" a key to an occupied location of the hash table due to the many-to-one nature of the hash function is called a *collision*.

Following the discussion regarding hash functions, we will discuss strategies about how to avoid and resolve collisions.



Many-to-one mapping function

$m=h(a_k)$;

m is the hash value;

a_k is the key.

Hash Functions

- A hash function should
  - be *easy to compute*;
  - *distribute keys evenly* within the hash table
  - *ensure equally likely hash values*.
- The performance of hashing depends on the effectiveness of the hash function.

Making a Hash Function

- Typically,
  1. the table size *M* is chosen to be a *prime number* that is the *first larger one than the necessary size of the table* if it is known (e.g., choose 11 if 10 is enough);
  2. some "*natural*" way is selected to *convert keys to large numbers r* (*i.e.*, $a_k \rightarrow r$), and
  3. *modulo M of this large number* (*r mod M*) is obtained as the *hash value of the key*.

Examples to Hash Functions

- "*Natural*" Ways to convert **string keys** to large numbers *(i.e., $a_k \rightarrow r$)*
- **Adding up ASCII values of characters in a string**

  Example: ali $\rightarrow$ 97+108+105=310

  A good hash function?
- **Another method:**

  f(k)=key[0]+27*key[1]+729*key[2];

  26^3 combinations possible for the first three letters

  However, only around 2850 are meaningful.
- **Horner's Rule**

A Hash Function

- Adding up the ASCII values of the characters in the string.
- Any problems with that strategy?
- Assume we chose a big hash table size (considering that we will place strings or words in the structure, a large hash table is not unreasonable at all) such as 10000 (or 10007 if you want to make it prime).
- We usually use words composed at most of eight characters.
- This means that the first 1016 (why?) cells are most likely to be allocated. The rest of the hash table space will mostly remain empty.
- Hence, the data are not evenly distributed.

Another Hash Function

- Consider a hashing mechanism considering only the first three characters of a word and processing it as follows:
- *key[0]+key[1]*27+key[2]*272 mod tablesize*
- Here 27 is selected regarding the fact that the English alphabet has 26 letters.
- This is a good selection provided that the occurrence of the first three characters of English words are quite uniformly distributed over the set of all three-character strings.
- Unfortunately, this is not true. Out of a total of 263=17576 possible combinations, only 2851 three-character strings are meaningful, and hence, encountered in an online English dictionary.
- Hence, even if no collisions happen in a table chosen as above, 28% of the hash table would be full.
- For large tables this is not a good function to use.

Horner's Rule

- **Horner's rule:**
  Another hash function proposed by Horner and called *Horner's rule*, has the formulation below:
- This mechanism is better than the two former functions. If the strings are too long, it takes long for hash values to compute. Then a certain substring of the key may be used.

$$\sum_{i=0}^{keysize-1} key[keysize-i-1]*37^{i} \bmod tablesize$$

Hash Functions for Integer Keys

Truncation Method: Take the first few or last few characters or digits as the hash code. This method is easy and fast. **e.g.**, Consider Example2, only a subset of the id digits can be used. If 3 high-order digits are used then a table of size 1000 can be created. Collisions?

- **Division Method**: We map a key *k* into one of *m* slots by taking the remainder of *k* divided by *m*. Quite fast

  e.g., *k*=34752 |1000 (maxItems = 1000)

  34

  ———

  752

- That is the hash function is $h(k) = k \bmod m$

More Examples to Hash Functions…

- **Multiplication Method:** Operates in two steps. First, we multiply the key, k by a constant in the range $0<A<1$ and extract the fractional part. Then, we multiply this value by m and take the floor of the result.

  That is the hash function is $h(k) = \lfloor m\ (k\ A \bmod 1) \rfloor$

  e.g., k=123456, m=10000, and $A = (\sqrt{5}-1)/2$

  $$h(k) = \lfloor 10000 * (123456 * 0.61803\ldots \bmod 1) \rfloor$$
  $$= \lfloor 10000 * 0.0041151\ldots \rfloor$$
  $$= \lfloor 41.151\ldots \rfloor$$
  $$= 41$$

More Examples to Hash Functions…

- **Midsquare Method**: The key is multiplied by itself (squared) and then the middle few digits of the result are selected as the hash code.

- **e.g.**, $k = 510324$ $k^2 = 260430584976$

  $h(510324) = 058$

More Examples to Hash Functions…

- Key is partitioned or divided into several pieces. Pieces are operated upon in some way. Adding them together and taking the required number of digits as the hash code is one of the possibilities.

  e.g., $k = 510324$

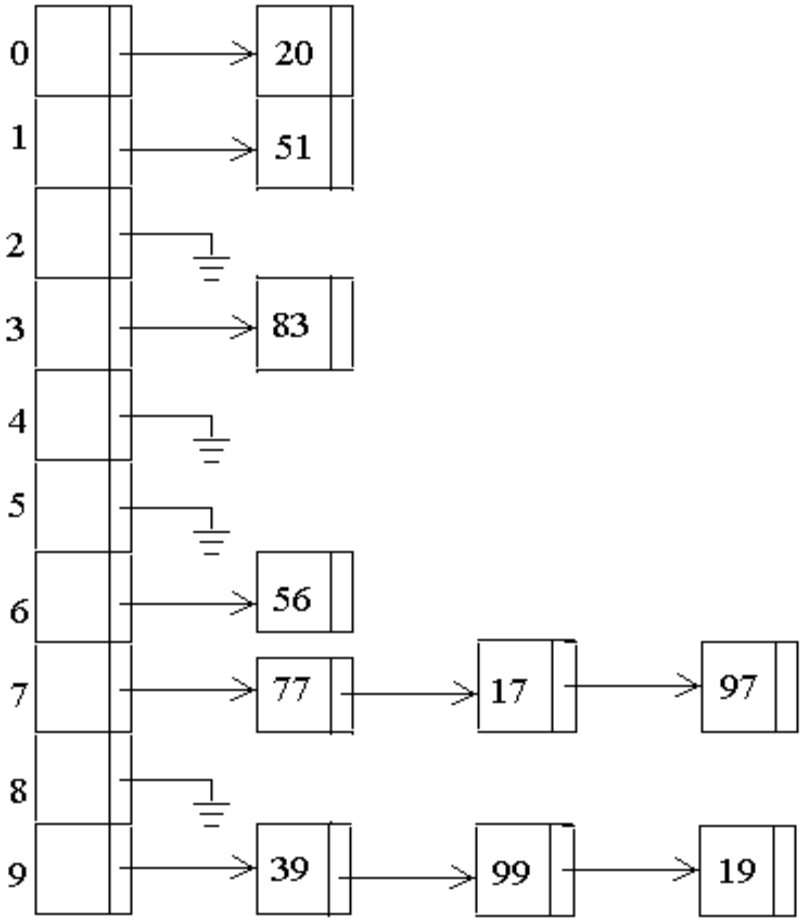
- **Folding method 1**: 51
  03
  24
  78 For m=1000, $h(k)$ = 078

- **Folding method 2**: Fold left and right sections

  1 03 2
  5 4

  15
  03
  42
  60

Separate Chaining



Open Addressing (Closed Hashing)

- The hash table has a fixed size, M.
- Any data point $a_k \in A$ under consideration is placed in the hash table.
- The hash table cell, m, the data point a_k is placed is determined by the hash function, h, or $m=h(a_k)$.
- Since $K>>M$, sometimes collisions occur. To resolve the collisions, we use collision resolution strategies.

Analysis of Open Addressing

- We will express the performance of hash tables in terms of the load factor *a* of the hash table.
- *Load Factor*: The number that specifies how many elements of the hash table are full, or

$$a = n/m,$$

where

n is the *number of occupied table cells*, and

m is the *table size*.

Cost of Unsuccessful Search & Insert

- In an *unsuccessful search*, at the end of the search, we find out that the key is *not in the hash table* once we find an *empty table cell*.
- This is also what we do to insert a key into the hash table. We *insert* the key when we *find an available space* (unoccupied or *empty table cell*).
- Hence, for both operations, we will attempt to *find* the *expected number of checks (probes)* we make before we find an available table cell.

Cost of Unsuccessful Search & Insert

- Assume *X* is an RV and *represents the number of probes made in an unsuccessful search*.
- The probability that *X is at least i* (i.e., $x \geq i$) probes before an empty slot is found is:

$$p(x \geq i) = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\left(\frac{n-2}{m-2}\right)\cdots\left(\frac{n-i+2}{m-i+2}\right) \leq \left(\frac{n}{m}\right)^{i-1} = a^{i-1}$$

Cost of Unsuccessful Search & Insert

The expected number of probes made in an unsuccessful search is *at most*:

$$E[X] = \sum_{i=1}^{\infty} i p(X = i) = \sum_{i=1}^{\infty} i\{p(X > i) - p(X \geq i+1)\}$$

$$\begin{aligned} E[X] = {} & p(X > 1) - \quad p(X \geq 2) + \\ & 2p(X > 2) - 2p(X \geq 3) + \\ & 3p(X > 3) - 3p(X \geq 4) + \\ & 4p(X > 4) - 4p(X \geq 5) + \\ & \vdots \end{aligned}$$

$$E[X] = \sum_{i=1}^{\infty} p(X \geq i) \leq \sum_{i=1}^{\infty} a^{i-1} = \sum_{i=0}^{\infty} a^{i} = \frac{1}{1-a}$$

Cost of Successful Search

From the last slide we remember that the expected number of probes for the insertion of a key is at most *1/(1-a)*, *a* being the load factor. If, for instance, the key inserted is the *(i+1)st* key, then, the expected number of probes cannot exceed:

$$\frac{1}{1-a} = \frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$$

Cost of Successful Search

Using this info, the expected number of probes for a successful search can be found by averaging *m/(m-i)* for the *(i+1)st* key over *n* keys as follows:

$$t(n) = \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{a}\left(\frac{1}{m} + \frac{1}{m-1} + \frac{1}{m-2} + \cdots + \frac{1}{m-n+1}\right)$$

$$t(n) = \frac{1}{a}\left\{\sum_{i=1}^{m}\frac{1}{i} - \sum_{i=1}^{m-n}\frac{1}{i}\right\} = \frac{1}{a}\sum_{i=m-n+1}^{m}\frac{1}{i} \le \frac{1}{a}\int_{m-n}^{m}\frac{dx}{x} = \frac{1}{a}\ln\frac{m}{m-n} = \frac{1}{a}\ln\frac{1}{1-\frac{n}{m}}$$

$$t(n) = \frac{1}{a}\ln\frac{1}{1-a}$$

Resolving Collisions in Hash Tables

- Collision Resolving Strategies
  - *Linear Probing*
  - *Quadratic Probing*
  - *Double Hashing*

Linear Probing

- Given an ordinary hash function

  h: $A \rightarrow \{0,1,\ldots, m\text{-}1\}$, the method of linear probing uses the hash function

$$h'(a_k,i) = (h(a_k) + i) \bmod m$$

where i is the number of collisions occurred for the current key.

Linear Probing: Example

i\keys	Empty	24	116	50	66	26	259	144	247	40	51
0				50	50	50	50	50	50	50	50
1					66	66	66	66	66	66	66
2						26	26	26	26	26	26
3							259	259	259	259	259
4								144	144	144	144
5									247	247	247
6										40	40
7											51
8											
9											
10											
11		24	24	24	24	24	24	24	24	24	24
12			116	116	116	116	116	116	116	116	116
CF		0	0	2	0	2	4	3	5	5	8

Quadratic Probing

- **Quadratic probing**: uses a hash function of the form

$$h'(a_k, i) = (h(a_k) + c_1 i + c_2 i^2) \bmod m$$

where

- i is the number of collisions occurred for the current key, and
- c_is are the coefficients of the quadratic function.

Quadratic Probing: Example

i\keys	Empty	24	116	50	66	26	259	144	247	40	51
0						26	26	26	26	26	26
1					66	66	66	66	66	66	66
2				50	50	50	50	50	50	50	50
3							259	259	259	259	259
4									247	247	247
5								144	144	144	144
6											
7											
8											51
9											
10										40	40
11		24	24	24	24	24	24	24	24	24	24
12			116	116	116	116	116	116	116	116	116
CF		0	0	2	0	0	2	2	2	3	3

Double Hashing

- **Double hashing**: is one of the best methods available. It uses a hash function of the form

$$h'(a_k,i) = (h_1(a_k) + i\, h_2(a_k)) \bmod m$$

where

- i is the number of collisions occurred for the current key, and
- $h_2(a_k)$ is the second hash function involved in case of a collision.

Selecting the second Hash Function

- A popular form of the second hash function is:

$$h_2(a_k) = R - (a_k \bmod R)$$

  where R is usually selected as the closest smaller prime number than the table size.
- The reason for selecting R this way is to obtain any second hash value equally likely.

Double Hashing: Example

h2 (a_k)=R – (a_k mod R); R=11

i\keys	Empty	24	116	50	66	26	259	144	247	40	51
0						26	26	26	26	26	26
1					66	66	66	66	66	66	66
2											
3				50	50	50	50	50	50	50	50
4							259	259	259	259	259
5										40	40
6									247	247	247
7											51
8								144	144	144	144
9											
10											
11		24	24	24	24	24	24	24	24	24	24
12			116	116	116	116	116	116	116	116	116
CF		0	0	1	0	0	1	2	1	1	2
$h_2(a_k)$		---	---	5	---	---	5	10	6	4	4

Rehashing

- If more than half of the current hash table is loaded, a new and larger hash table is constructed.
- All keys are placed in this new table using a new hash function.
- This is called ***rehashing***.
- A typical example to the selection of the size of the new table is the first prime that is greater than two times the size of the current hash table.