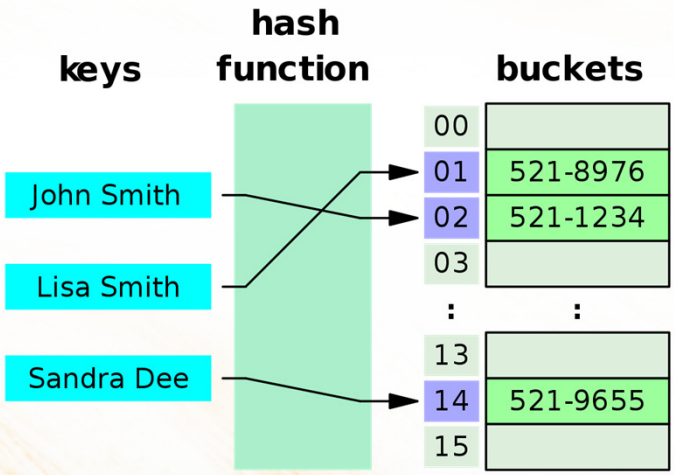
Chapter 5 - Introduction to Hash Tables



1

1

Chapter 5 - Introduction to Hash Tables

- Up to now, the main process used by searching techniques was comparing values of keys to values of other keys (such as sequentially through a list) until we found a match
- A different way to search can be based on using the key's value to determine the place where the key would be within a table. This use of a key value in this manner is known as hashing and creates a hash of the key.
- The hash value is then used to place an object in a "bucket", or to search for a value within that "bucket". The bucket contains all the objects that result in a particular hash value.
- If a hashing function results in a uniform distribution across all possible hash values, then searching/inserting/deleting time can be minimized, and even be done in constant time in some cases!

2

2

Hash Functions

- Since the value of the key is the only indication of position, if the key is known, we can access the table directly via some kind of lookup or offset mechanism
- This can reduce the search time from $O(n)$ or $O(\lg n)$ to $O(1)$, especially if there are more "buckets" in the hash table than there are objects being placed into the table, so that each key goes to a different bucket
- No matter how many elements there are, the run time is the same – that's why can be called "constant" time!
- Unfortunately, this is just an ideal; in real applications we can only approximate this, especially if multiple objects are routinely placed into the same hash bucket (thus forcing a search within the bucket)
- The task is to develop a function, *h*, that can transform a key, *K*, into an index, or "hash", $h(K)$. The value of $h(K)$ determines which bucket to look in.
- The function *h* is called a ***hash function***

3

3

Hash Functions

- Consider a symbol table for a compiler, that needs to store all the variable names used in a program. Given the nature of the variable names typically used, a hash table with 1000 positions or "buckets" may be more than adequate
- In this case, any hash function that can reasonably spread out variable names across the buckets would be more than sufficient for this purpose, resulting in quicker compile times for code

4

4

(Bad) Example Hash Function

- EXAMPLE: we could define *h* to be the sum of the ASCII values (8-bit binary values) of letters in the variable name
- If we restrict variables to having 31 or fewer characters, we will need only a few thousand positions, since a variable with of 31 characters all "z" would sum to 31 · 122 (the ASCII code for "z") = 3782
- HOWEVER, the function will not produce unique values, for *h*("abc") = 97 + 98 + 99 = 294, and *h*("acb") = 97 + 99 + 98 = 294, and thus be mapped to the same bucket
- This is called a ***collision*** in the hash function, and is a measure of the usefulness of a hash function. The more collisions that occur, the worse the hash function is!
- An ideal hash function would distribute keys uniformly across all hash values, and avoid patterns that might cause collisions
- Collisions can be reduced by making the function *h* better than this simple example

5

5

Modulo Arithmetic Hash Functions

- Using Modulo Arithmetic is one possibility
- Hash functions must guarantee that the value they produce is a valid index to the table. An easy way to ensure this is to use Modular Division, and divide the keys by the size of the table: ***h*(*K*) = *K* mod *TSize*** where ***TSize* = *sizeof* (*table*)**
- This works best if the table size is a prime number, but if not, we can use ***h*(*K*) = (*K* mod *p*) mod *TSize*** for a prime $p \geq TSize$
- However, nonprimes often work for the divisor as well, provided they do not have any prime factors less than 20
- This modulo method is often used when little is known about the key values themselves

6

6

"Folding" Hash Functions

- Another approach is called Folding
- In folding, the keys are divided into parts which are then combined (or "folded") together and often transformed into the address
- Two types of folding are used, ***shift folding*** and ***boundary folding***
- In shift folding, the parts are placed underneath each other and then processed (for example, by adding them together)
- Using a Social Security number, say 123-45-6789, we can divide it into three parts - 123, 456, and 789 – and add them to get 1368
- This can then be divided using modulo *TSize* to get a hash value
- With boundary folding, the key is visualized as being written on a piece of paper and "folded" on the boundaries between the parts

7

7

"Folding" Hash Functions

- One aspect of this "folding" technique is that alternating parts of the key are reversed, so the Social Security number part would be 123, 654, 789, totaling 1566
- As can be seen, in both versions, the key is divided into even length parts of some fixed size, plus any leftover digits
- Then these are added together and the result is divided modulo the table size
- Consequently this is very fast and efficient, especially if bit strings are used instead of numbers
- With character strings, one approach is to exclusively-or the individual character together and use the result
- For example, *h*("abcd") = "a" ⊕ "b" ⊕ "c" ⊕ "d"

8

8

Mid-Square Hash Function

- In the "Mid-Square" approach, the numeric value of a key is squared and the middle part is extracted to serve as the address
- If the key is non-numeric, some type of preprocessing needs to be done to create a numeric value (such as folding)
- Since the entire key participates in generating the address, there is a better chance of generating different addresses for different keys

- For example, if the key is 3121, 3121^2 = 97**406**41, and if the table has 1000 locations, $h(3121) = 406$, which is the middle part of 3121^2
- In application, powers of two are more efficient for the table size and the middle of the bit string of the square of the key is used
- Assuming a table size of, say, 1024, 3121^2 is represented by the bit string 1001010 **0101000010** 1100001, and the key, 322, is used

9

9

Hash Tables - Collision Resolution

- The hashing we've looked at so far might have problems with multiple keys hashing to the same location in the table, or hashing to locations in a non-uniform way
- So in addition to using more efficient functions, we also need to consider the size of the table being hashed into
- Even then, we cannot guarantee to eliminate collisions; we are forced to consider approaches that assure a solution
- A number of methods have been developed; we will consider a few of them

10

10

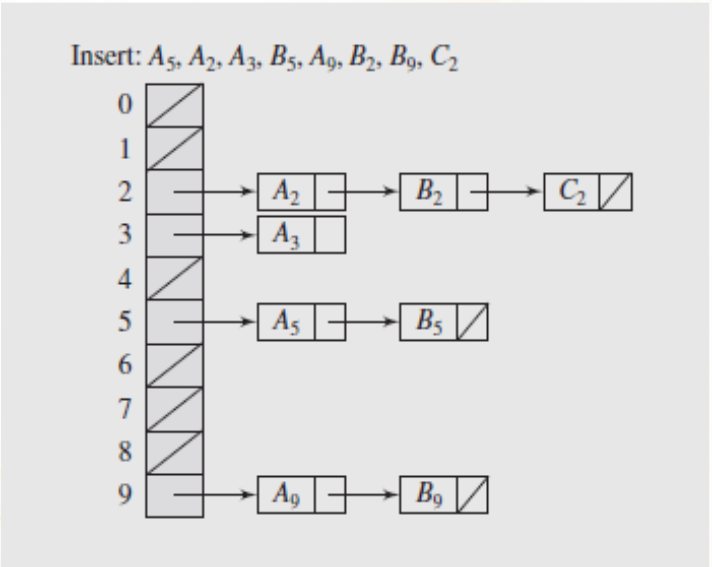
Collision Resolution – Chaining

- One method of handling the case when multiple keys hash to the same bucket is called ***chaining***
- In ***chaining***, the keys are stored in the `info` portion of a linked list of nodes associated with each bucket
- In this way the table never overflows, as the lists are extended whenever new keys are inserted, as can be seen on the next slide
- This is very fast for short lists, but as they increase in size, performance can degrade sharply, with a complexity of O(n)
- Specifically, if there are n keys and k buckets, and n becomes much greater than k, we end up with n/k items in each bucket, which requires O(n) time to search for items in the worst case
- Gains in performance can be made if the lists are ordered so unsuccessful searches don't traverse the entire linked list, but that also requires extra time, so there's still decreased efficiency

11

11

Collision Resolution - Chaining



12

12