**Lesson 22 – Hash Tables**

**Reading: Chapter 13, Section 2 of the text.**

**Learning Objectives:**

* Describe the functionality of a hash table.
* Describe methods for resolving hash table collisions.
* Describe the characteristics of a good hash function.
* Analyze the time complexity of a hash table.

**Hashing:**

* We have been discussing trees as a data structure. The benefit of using a tree, if it is balanced, is that it is more efficient to search a tree than it is to search a list. Searching a list takes *O(n)* time while searching a balanced binary tree takes *O(log2 n)* time. For both, even if we know the location of the item (recall the linked-list delete operation by index)
* A radically different strategy is necessary to locate (and insert and delete) an item virtually instantaneously.
  + Imagine an array table of *n* slots—with each array slot capable of holding a single table item—and a seemingly magical box called an “address calculator.” Whenever you have a new item that you want to insert into the table, the address calculator will tell you where you should place it in the array.
  + An insertion is O(1); that is, it requires constant time.
  + You also use the address calculator for the retrieve and delete operations. It thus appears that you can perform the operations insert, retrieve and delete all in constant time.
* If you are to implement such a scheme, you must, of course, be able to construct an address calculator that can, with very little work, tell you where a given item should be.
  + Such an address calculator is known as a **hash function**.
  + The scheme just described is an idealized form of a method known as **hashing**, and the array table is called the **hash table**.
* For example, suppose that we store a person’s information in an array based on their telephone number. Since a phone number is of the form xxx-xxx-xxxx (assume we restrict to U.S. phone numbers), we can access their information as such: person[9998887777].getRecord(). Access in an array, as such, can be done in constant time. But, what is the down side?
* Perhaps we restrict it to the last 4 numbers in the telephone number. Then, we can access their information as such: person[7777].getRecord(). This significantly reduces the size of the storage array.
  + If the phone number is of type int, how would we easily convert the full phone number into the last 4 digits?
  + But, what is the downside to the reduction in space requirements for this approach?
* Since the hashing scheme stores the items in an array, it would appear to suffer from the familiar problems associated with a fixed-size implementation.
  + The hash table must be large enough to contain all of the items that you want to store.
  + The hash function might map two or more search keys to the same index into the array, even if there are empty slots in the array. This occurrence is called a **collision**. A hash function that will never do this is called a **perfect hash function**.
  + Because reserving vast amounts of storage is usually not practical, **collision-resolution** **schemes** are necessary to make hashing feasible.
* To summarize, a typical hash function must:
  + be easy and fast to compute; and
  + place items evenly throughout the hash table.

**Hash function strategies:**

* + It is sufficient to consider hash functions that output integer values (array index value).
  + The goal is coming up with a hash function that has a uniform distribution across the range of indices.
  + One method might be to select certain digits from the key. For example, wrt using the telephone number, selecting the last 4 or last 2 would probably work well. Why not selecting the first two? What about the first and last digit?
  + One way to improve upon the previous approach is to add the digits together. What would the range of values be for telephone numbers? Is this a good approach?
  + Modulo arithmetic provides a simple and effective hash function. For example, consider the function:

For example, if table has 101 elements, then mod 100 returns a value from 0 to 99 (the indices of the array table). Usually, a size is selected that is prime. That way, even if the keys are not uniformly distributed, the hash values will be.

* + If your search key is a character string—such as a name—you could convert it into an integer before applying the hash function *h(x)*. Using the ASCII values of each character, we can perform some mathematical operation and then take the modulo of the value.

**Resolving Collisions:**

* **Open addressing**: During an attempt to insert a new item into a table, if the hash function indicates a location in the hash table that is already occupied, you probe for some other empty, or open, location in which to place the item. The sequence of locations that are examined are called the **probe sequence**.
  + **Linear probing**: In this simple scheme to resolve a collision, you search the hash table sequentially, starting from the original hash location until you find an available location. Typically, you wrap around from the last table location to the first table location if necessary.
  + **Quadratic probing**: You can virtually eliminate primary clusters simply by adjusting the linear probing scheme so that, instead of probing consecutive locations, you check locations hashIndex + 12, hashIndex + 22, hashIndex +32, etc. until an empty location is found.  
    Let hash(x) be the slot index computed using the hash function.
    - If the slot hash(x) % S is full, then we try (hash(x) + 1\*1) % S.
    - If (hash(x) + 1\*1) % S is also full, then we try (hash(x) + 2\*2) % S.
    - If (hash(x) + 2\*2) % S is also full, then we try (hash(x) + 3\*3) % S.
    - This process is repeated for all the values of i until an empty slot is found.
  + **For example:** Let us consider a simple hash function as “key mod 7” and  sequence of keys as 50, 700, 76, 85, 92, 73, 101

Calendar

Description automatically generated with low confidence

* + **Double hashing**: Double hashing defines key-dependent probe sequences. In this scheme, the probe sequence still searches the table in a linear order, starting at the location *h1(key)*, but a second hash function *h2* determines the size of the steps taken. For example, let

; and

When using double hashing, it is important to choose two hash functions that are independent of each other, meaning that the output of one function should not be predictable from the output of the other. One common way to achieve independence is to use a prime number for the second hash function.

This is because a prime number is unlikely to be a factor of the size of the hash table, which means that the resulting index values from the second hash function are more likely to be uniformly distributed across the table. If a non-prime number were used for the second hash function, it could introduce biases and lead to poor performance of the hash table.

h(key, i) = (firstHashfunction(key) + i \* secondHashFunction(key)) % tableSize

firstHashFunction(key) = key % tableSize

secondHashFunction(key) = PRIME - (key % PRIME)

Let us consider the same example in which we choose R = 7.

A screenshot of a phone

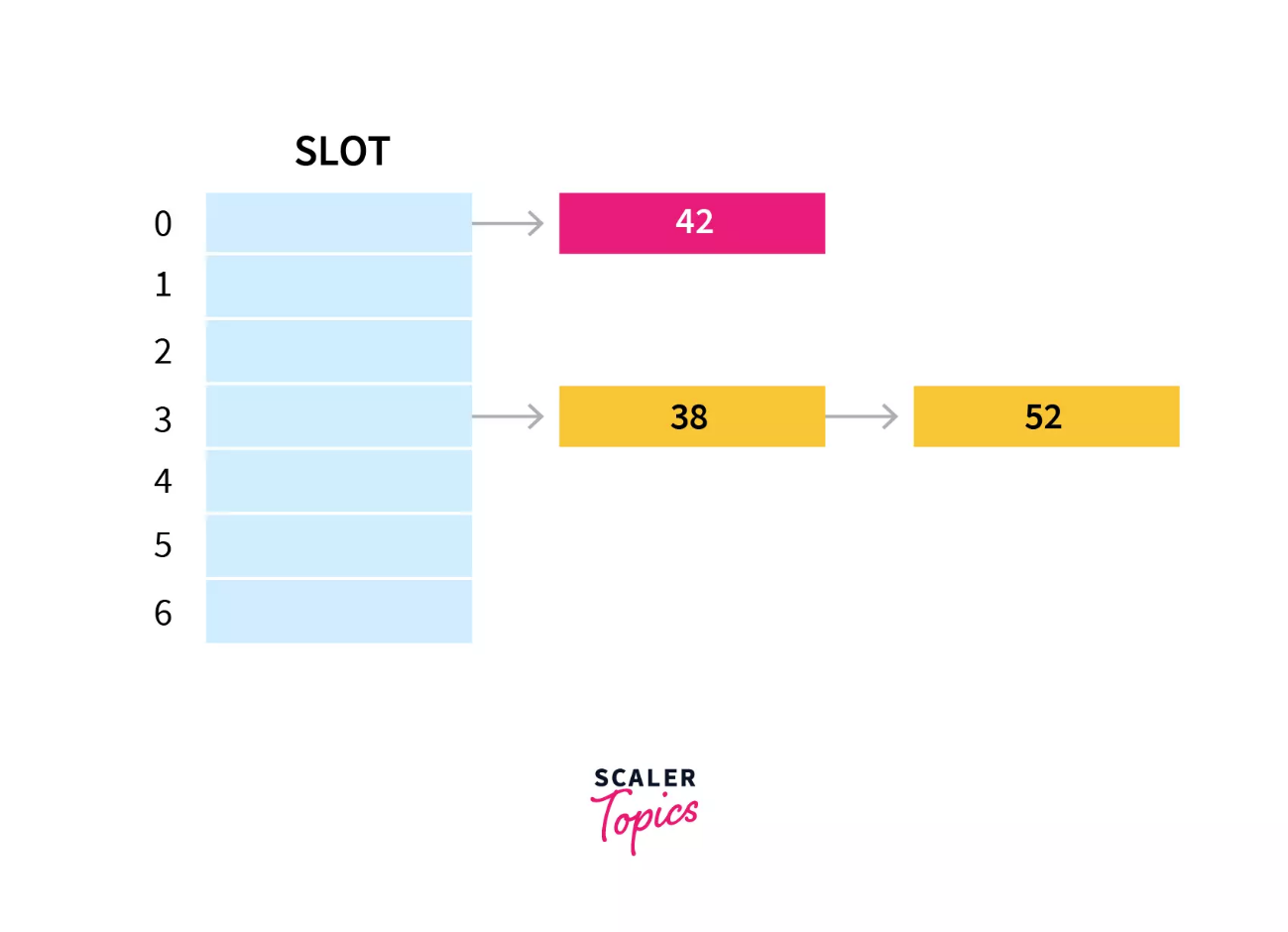
Description automatically generated with low confidence

A Closed Hash Table using Double Hashing

Table

Description automatically generated

* **Increasing the size of the hash table:** If you use a resizable array, you can increase its size whenever the table becomes too full.
  + You can’t simply double the size of the array because the size of the array must remain prime.
  + Also, you can’t simply copy the values of the old array to the same location in the new array. You need to apply your new hash function to every item in the old hash table before placing it into the new hash table.
* **Buckets:** In this approach, each hash table location contains another array (the bucket) so that each colliding item is stored at the same location in the hash table within the bucket.
  + The problem with this approach is choosing the size B of each bucket. If B is too small, then collisions are only postponed. If B is too large, then a large amount of storage will be wasted.
* **Separate chaining:** A better approach is to design the hash table as an array of linked lists. In this arrangement, each location in the hash table is a reference to the front of a linked list—the chain.

****

**The Efficiency of Hashing:**

* An analysis of the average-case efficiency of hashing involves the **load factor** α, which is:
* Linear probing:

for a successful search; and

for an unsuccessful search

* Quadratic probing and double hashing:

for a successful search; and

for an unsuccessful search

* Separate chaining:

for a successful search; and

for an unsuccessful search