COMP 251: Data Structures and Algorithms	Fall 2006

Hash Tables

Lecturer/TA: Ethan Kim *October 3rd, 2006*

1 Abstract Data Types (ADTs)

Definition 1. A*n Abstract Data Type(ADT) is a set of data values and associated operations that are precisely specified independent of any particular implementation.*

Examples:

1. Stack: **push, pop**
2. Queue: **enqueue, dequeue**
3. Priority Queue: **insert, find_max, delete, . . .**
4. Dictionary: stores (key, value) pairs, and supports **insert, find, delete**

We use various data structures to implement these ADTs. For example..

- Binary Search Trees
- Arrays
- Linked Lists
- **Hash Tables**

2 Hash Tables

What are hash tables? Suppose we want to implement an ADT that supports *insert, delete,* and *search*. In particular, suppose each data contains two entries: *key* and *value*. Note that the key serves as a way to identify the entry, whereas the value can be any combination of information. For example:

- student information: key is student ID, where the value can be the student's name, address, etc.
- credit card information: key is the credit card number, and the value can be the client's information..
- car information: key is the license plate number, and the value can be its maker, model, year, . . .

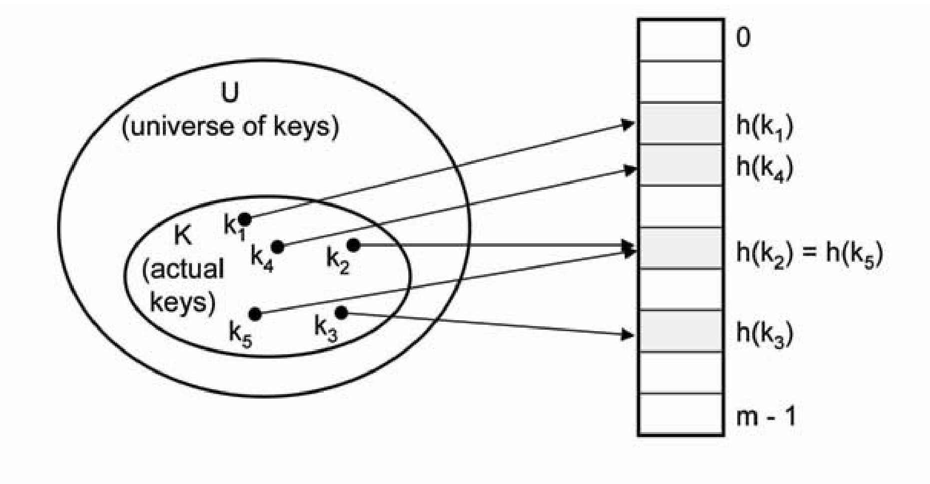

Figure 1: a hash table; collisions are not handled

Now, consider the credit card information. VISA card numbers are 16 digits long, so there are 10^{16} possible keys.

1. If we build an array capable of holding 10^{16} accounts, it would require Petabytes of RAM even if each entry is only one byte. → *extremly inefficient!*

2. If we build a linked list to hold only the existing accounts, the space required won't be an issue, but now it takes $O(n)$ time to search.

Hash tables let us implement these operations in $O(1)$ running time on average.

2.1 Basic Concepts

Let U be the *universe* of possible key values. (As for the VISA card example, there are 10^{16} key values in this universe.) Let $T[0 \dots m-1]$ be the *hash table*. We define the *hash function* as $h : U \to \{0, 1, \dots, m-1\}$.

What the hash function does is basically mapping the key to some index of the hash table. Ideally, m is much smaller than $|U|$, so that we do not waste any space. But, as we have more entries in the hash table, problems may occur. See Figure 1.

If m is smaller than the number of keys stored in the hash table, there is a *collision*. The two main challenges when designing a hash function are:

1. Minimize the number of collisions

2. How to handle the collisions

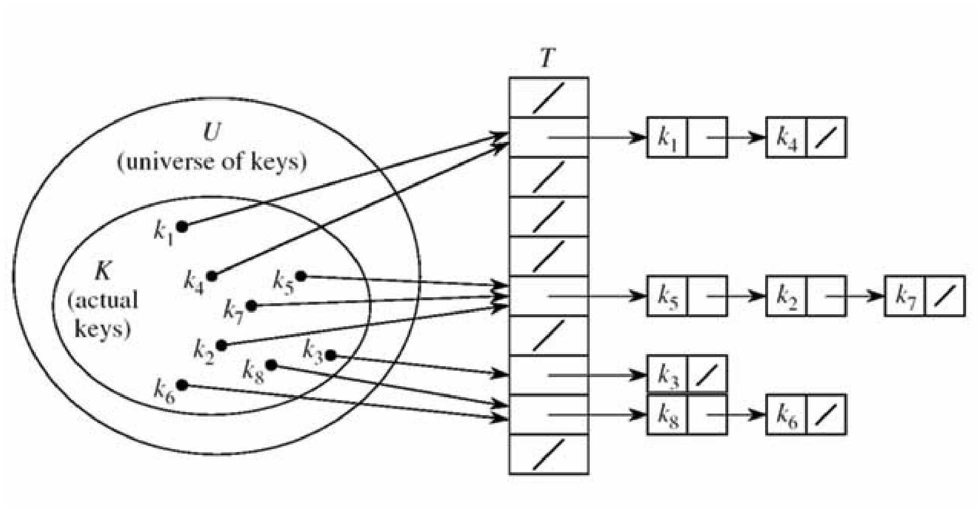

Figure 2: a hash table with separate chaining

2.2 Hash table with separate chaining method

One way to resolve collisions is to create a linked list for each entry in the hash table. See Figure 2. We may implement this as follows:

1. CHAINED-HASH-INSERT(T, x)
   insert x at the head of the list T[h(key[x])]
2. CHAINED-HASH-SEARCH(T, k)
   search for element with key k in T[h(k)]
3. CHAINED-HASH-DELETE(T, x)
   delete x from the list T[h(key[x])]

Observe that both the INSERT and DELETE operations can be done in $O(1)$ time, worst-case. For SEARCH operation, it takes $O(n)$ time. Why? In the worst-case scenario, the hash function may decide to put all the entries into a single cell in the hash table, in which case we simply have all the elements in a linked list.

But, we can say a little more..

2.3 Analysis of hashing with chaining

We want to analyze the expected running time for CHAINED-HASH-SEARCH operation. In order to make our analysis easy, we assume that our hash function perfectly distributes the keys into the hash table.

Given a hash table T with m slots that stores n elements, we define the *load factor* α as $\frac{n}{m}$. This is the average number of elements stored in a single slot in T. For now, assume that every element is equaly likely to hash into any of the m slots, independently of other element's positions in the hash table. This assumption is called *simple uniform hashing.* This allows us to figure out the average length of the linked lists attached to T.

For $j \in \{0, \ldots, m-1\}$, let n_j = length of list $T[j]$. Thus, we have $n = \sum_{j=0}^{m-1} n_j$. If we take the expected value of both sides:

$$\begin{aligned} n = E[n] &= E[\sum_{j=0}^{m-1} n_j] \\ &= \sum_{j=0}^{m-1} E[n_j] \\ &= mE[n_j] \qquad \text{by SUH} \end{aligned}$$

So we have that $E[n_j] = \frac{n}{m} = \alpha$.

Now that we know the expected length of the chains, we can analyze the time it takes to find an arbitrary element x in the hash table T. There are two scenarios: 1) we find x in T. 2) we don't find x in T. We look at each case separately.

Unsuccessful Search First, we need to hash the key into the hash table, i.e., compute h(key[x]). This can be done (usually) in $O(1)$ time. Then, we need to run down the linked list looking for that key. The expected length of the list is α. So the overall search takes $O(1+\alpha)$ time.

Successful Search Again, we need to hash the key into the hash table, so this takes $O(1)$ time. Then, we look for the key in the corresponding chain in the hash table. On average we look at halfway through a list, i.e., $\frac{\alpha}{2}$. This gives the overall search takes $O(1+\alpha)$ time.

We could do this a little more formally. The time it takes for a successful search is the number of elements before x in the list plus 1. Notice, however, the elements before x are all inserted *after* x. (Recall the implementation of CHAINED-HASH-INSERT.) Let x_i denote i^{th} element inserted into the table, and let k_i denote the key for x_i. Consider the indicator random variable $X_{i,j} = I\{h(k_i) = h(k_j)\}$ for collisions between x_i and x_j. By SUH, we have that $Pr\{h(k_i) = h(k_j)\} = \frac{1}{m}$. Thus, we have E$[X_{i,j}] = \frac{1}{m}$. Now, we can compute the number of elements searched as below:

$$\begin{aligned} E[\frac{1}{n}\sum_{i=1}^{n}(1 + \sum_{j=i+1}^{n} X_{i,j})] &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} E[X_{i,j}] \\ &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{1}{m} \\ &= 1 + \frac{n}{m} \\ &= 1 + \alpha \end{aligned}$$

So we can do the searches in $O(1+\alpha)$ time. Even better, if the size of hash table is proportional to the number of elements stored, $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$. Therefore, the expected running time for CHAINED-HASH-SEARCH is $O(1)$.