CSC 378 Lecture 5

June 14, 2001

For another reference, see chapter 12 of the textbook (CLR).

1 Direct Addressing

Recall that a dictionary is an ADT that supports the following operations on a set of elements with well-ordered key-values: INSERT, DELETE, SEARCH. If we know the key-values are integers from 1 to $K$, for instance, then there is a simple and fast way to represent a dictionary: just allocate an array of size $K$ and store an element with key $i$ in the $i$th cell of the array.

This data structure is called direct addressing and supports all three of the important operations in worst-case time $\Theta(1)$. There is a major problem with direct addressing, though. If the key-values are not bounded by a reasonable number, the array will be huge! Remember that the amount of space that a program requires is another measure of its complexity. Space, like time, is often a limited resource in computing.

Example 1: A good application of direct addressing is the problem of reading a textfile and keeping track of the frequencies of each letter (one might need to do this for a compression algorithm such as Huffman coding). There are only 256 ASCII characters, so we could use an array of 256 cells, where the $i$th cell will hold the count of the number of occurrences of the $i$th ASCII character in our textfile.

Example 2: A bad application of direct addressing is the problem of reading a datafile (essentially a list of 32-bit integers) and keeping track of the frequencies of each number. The array would have to be of size $2^{32}$, which is pretty big!

2 Hashing

A good observation about example 2 or about any situation where the range of key-values is large, is that a lot of these might not occur very much, or maybe even not at all. If this is the case, then we are wasting space by allocating an array with a cell for every single key-value.

Instead, we can build a hash table: if the key-values of our elements come from a universe (or set) $U$, we can allocate a table (or an array) of size $m$ (where $m < |U|$), and use a function $h : U \rightarrow \{0, \ldots, m-1\}$ to decide where to store a given element (that is, an element with key-value $x$ gets stored in position $h(x)$ of the hash table). The function $h$ is called a hash function.

2.1 Chaining

If $m < |U|$, then there must be $k_1, k_2 \in U$ such that $k_1 \neq k_2$ and yet $h(k_1) = h(k_2)$. This is called a collision; there are several ways to resolve it. One is to store a linked list at each entry in the hash
table, so that an element with key $k_1$ and an element with key $k_2$ can both be stored at position $h(k_1) = h(k_2)$ (see figure). This is called chaining.

Assuming we an compute $h$ in constant time, then the **INSERT** operation will take time $\Theta(1)$, since, given an element $a$, we just compute $i = h(key(a))$ and insert $a$ at the head of the linked list in position $i$ of the hash table. **DELETE** also takes $\Theta(1)$ if the list is doubly-linked (given a pointer to the element that should be deleted).

The complexity of **SEARCH($S,k$)** is a little more complicated. If $|U| > m(n - 1)$, then any given hash function will put at least $n$ key-values in some entry of the hash table. So, the **worst case** is when every entry of the table has no elements except for one entry which has $n$ elements and we have to search to the end of that list to find $k$ (see figure). This takes time $\Theta(n)$ (not so good).

For the **average case**, the sample space is $U$ (more precisely, the set of elements that have key-values from $U$). Whatever the probability distribution on $U$, we assume that our hash function $h$ obeys a property called **simple uniform hashing**. This means that if $A_i$ is the event (subset of $U$) \{ $k \in U \mid h(k) = i$ \}, then

$$
\Pr(A_i) = \sum_{k \in A_i} \Pr(k) = 1/m.
$$

In other words, each entry in the hash table is used just as much as any other. So the expected number of elements in any entry is $n/m$. We will call this the load factor, denoted by $a$.

To calculate the average-case running time, let $T$ be a random variable which counts the number of elements checked when searching for key $k$. Let $L_i$ be the length of the list at entry $i$ in the hash
table. Then the average-case running time is:

\[ E(T) = \sum_{k \in U} \Pr(k)T(k) \]  
\[ = \sum_{i=0}^{m-1} \sum_{k \in A_i} \Pr(k)T(k) \]  
\[ \leq \sum_{i=0}^{m-1} \Pr(A_i)L_i \]  
\[ = 1/m \sum_{i=0}^{m-1} L_i \]  
\[ = n/m \]  
\[ = a \]

So the average-case running time of \texttt{SEARCH} under simple uniform hashing with chaining is \( O(a) \). We generally consider \( a \) to be constant since we can make \( m \) bigger when we know that \( n \) will be large. When this is the case, \texttt{SEARCH} takes time \( O(1) \) on average.