Dynamic Hash Tables

ECE 1762 Algorithms and Data Structures
Spring Semester, 2004 — U Toronto

1 Hash Tables Revisited

Our analysis for hash tables presumed that m, the hash table size, is equal to or greater than n, the number of elements we need to hash. However, n is usually not known in advance. For example, a compiler has no way of knowing in advance the number of program identifiers (n) so that it allocates the appropriate space (m) to the symbol table.

A reasonable way to handle the situation when n is unknown is to be prepared to construct a sequence of hash tables T_0, T_1, T_2, ...

We choose some suitable value for the size m of the initial hash table T_0. Once the number of inserted elements exceeds m, we create a new hash table T_1 of size $2m$ and rehash the keys with a new hashing function that returns values from 0 to $2m-1$. The old table T_0 is discarded. We can continue inserting more elements in T_1 until the number of elements inserted becomes bigger than $2m$. At that point we rehash all elements into a new table T_2 of size $4m$. To generalize the idea, we create a new hash table T_k of size $2^k m$ as soon as T_{k-1} contains $2^{k-1}m$ elements. We continue in this fashion until we have inserted all elements.

What is the expected time required to insert 2^k elements, assuming that $m = 1$? We see that this process, as described above, can be modeled by the recurrence:

$$T(1) = 1$$

$$T(2^k) = T(2^{k-1}) + 2^k$$

with solution: $T(2^k) = 2^{k+1} - 1 = O(\#keys)$.

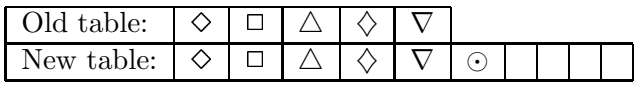
We conclude that a sequence of n `INSERT`, `FIND`, and `DELETE` operations can be processed in $O(n)$ total expected time through hashing.

(*Adapted from A.Aho, J.Hopcroft, and J.Ullman, "The Design and Analysis of Computer Algorithms,", Addison–Wesley, 1974.*)

2 Dynamic Hash Tables

Now we will give an amortized analysis of the problem above.

Like before, suppose that each time we attempt to insert an item into a table that is already full (with n items), we allocate $2n$ new memory cells, copy the original table to this new location, and then insert our item accordingly. As an example, consider the insertion of the item $\odot$:



Since every item of the old table must be moved to enlarge the table, this operation needs time $\Theta(n)$. Insertion when there is room in the table, on the other hand, can be accomplished in $O(1)$ time.

A naive worst-case analysis of n insertions would give a time bound of $\Theta(n) \times n = \Theta(n^2)$. By now, however, you should be fairly suspicious of this naive analysis and insist on an amortized calculation.

Aggregate method Consider inserting n items into the table. We want to analyze when we will be required to enlarge the table, and how many items will have to be moved. The first time we enlarge the table is after we have inserted 1 item (assuming the table starts out with size 1); this will require moving 1 item. The second time we enlarge the table is after we have inserted 2 items and will require 2 moving operations. The third time we enlarge the table will be when 8 items have been inserted into the table.

In general, after 2^{j-1} items have been inserted into the table, we will have to perform the j'th table enlargement and move all the items into the new table. Since we are inserting n items in total, it must be that $2^{j-1} \leq n$ which implies that $j \leq \lg(n) + 1$. Thus, the total number of items that will be moved by the enlargement operation would be:

$$\sum_{j=1}^{\lg n+1} 2^{j-1} \approx 2n$$

If we add the n operations needed for actually inserting items into a non-full table, then we get an overall running time of $3n$, or an amortized running time of $\frac{3n}{n} = 3$ steps per insertion.

Accounting We can analyze the same problem using the accounting problem. Specifically, we can charge \$3 per insertion. When a customer comes to us with an insertion request, he will pay \$1 for the labor and staple \$2 to the item.

When it is required to enlarge the table, we will pay for the movements of the various items with the \$2 stapled to the items (we will call these items the *wealthy items*). The question is: will we always have enough money in our hand to pay for this copy process?

The key idea lies in the fact that every time we enlarge the table, we double its size. Thus, if the table has n elements: half of the elements must be indigent and have no dollars stapled to them; the other half of the elements (wealthy elements) can pay for their own movement with their stapled money. However, the wealthy elements when they were inserted after the previous table enlargement, had $2\frac{n}{2} = n$ dollars stapled on them. This amount can pay for copying *all* the n elements ($n/2$ indigent and $n/2$ wealthy) in the new hash table. Observe that in the new, half-filled hash table all elements are now indigent. It is a good exercise of your understanding of the accounting method to go through the previous paragraph and convince yourself of how the accounting method works in this case.

Potential functions A final method of attacking our problem is with the use of potential functions. In this case, a potential function that works is:

$$\Phi(T) = 2 \times \text{the number of items in the table} - \text{the size of the table}$$

As a good exercise, prove that this potential function guarantees an amortized price of \$3 per insertion. The complete answer can be found in the textbook.

Deletion Just as we considered inserting elements into a tree, we may also consider deleting them from the tree. A simple implementation would merely delete a specified item from the table when requested. However, it is often advantageous to contract the table upon a deletion if possible. For example, when deleting the $\odot$ from the previous hash table example, it would be a good idea to contract the resulting table from 8 cells to 4 cells.

The question of when to contract the table is a difficult one. If we contract the table whenever it is half full, then we could end up with a very bad situation of repeated insert-delete combinations that cause the table to expand and contract. In this case, n operations could cost $\Theta(n^2)$ time.

Thus, it might make more sense to contract the table only when it is a quarter full so as to prevent this problem. Using a new potential function, we can see that insertion and deletion can be done in an amortized $O(1)$ time. For the interested reader, this is discussed in further detail in the textbook.