Data Strcutures

Kishore Kothapalli

February 15, 2010

Chapter 1

Hash Tables

1.1 Introduction

Consider writing a compiler for a modern day language such as C/C++. These languages employ strict naming and scoping rules. For example, in the same scope, a variable name must be unique. To check for this uniqueness of variables, one has to use a data structure that stores the previously defined variables along with their scope, and upon encountering a new variable definition, has to check for uniqueness. This information is stored in a table called the *symbol table*. Given the nature of the symbol table, that data structure must support the following three operations.

- `Insert` : Insert a new variable into the symbol table.
- `Find`: Check for the occurence of a variable in the symbol table, and
- `Delete` : Delete a name from the symbol table.

Let us look at some possible solutions to support these operations. To formalize, let us say that U, called the *universe*, is the set of all possible values. For instance, for a symbol table, U would be the set of well defined variable names.

One can keep an array of length $|U|$ and implement the above operations in $O(1)$ time as follows. For simplicity, we consider that each element of U is an integer. Let A be an array of size $|U|$.

- `Insert`(i) : { if $A[i] == 1$ then report ERROR; else $A(i) = 1$; }
- `Find`(i) : { if $A[i] == 1$ then return 1 else return 0;}
- `Delete`(i) : { $A(i) = 0$; }

The following figure illustrates the above implementation for $|U| = 10$, and the table containing vales $4, 7$, and 8.

FIGURE COMES HERE

What are the drawbacks of the above solution? It uses a space of $|U|$. In a typical program, the number of identifiers does not exceed a few hundred, at a huge overcount. However, the possible set of identifiers in a language is much higher. For example, even if we restrict variables to only character strings of length 8, the number is still 26^8. So the solution is rather wasteful in terms of space. However, an advantage of the solution is the simplicty of the operations and their $O(1)$ time, even in the worst case. It is therefore worthwhile to see if there is a solution that achieves $O(1)$ time without wasting space.

Let us consider another solution based on linked lists. In this approach, we intend to keep a list of all identifiers as a linked list. To insert an element, we simply add to the front of the list. To delete and find, we traverse the list of elements as follows.

- `Insert`(i) : { checkDuplicate(i); node $-$>next = head; head = node; }
- `Find`(i) : { while (head $\neq$ NULL and head$-$>data $\neq i$) head = head $-$>next; }
- `Delete` (i) : { ... }

Let us analyse this solution. The space required is only for maintaining the list of identifiers plus the pointers. So, it is linearly proportional to the number of identifiers. But, the time taken for the operations now increases to $O(n)$ when there are n identifiers. Notice also that maintaining a sorted order among the elements also does not help reduce the cost of the above operations.

It would be wonderful if we had a solution which guarantees us an $O(1)$ time for the operations and still uses $O(n)$ space for storing n identifiers. While that is not the case, we will study one solution called *hashing* which guarantees $O(1)$ time for each operation on the average and uses $O(n)$ space.

1.2 Hashing

In our setting, we have a universe U of values with $|U| = m$. We need to store a small subset $K \subset U$ with $|K| = n$. We have that $n \ll m$. Instead of direct addressing, we create a table T of size $O(n)$ and try to assign slots to elements of K u sing a function called the *hash function*. A hash function is normally denoted by h. Thus, key $k_1 \in K$ is stored in slot $h(k_1)$. Key k_1 is also said to hash to $h(k_1)$ and $h(k_1)$ is also called the hash value of k_1. Notice that, by convention, $0 \leq h(k_1) \leq |T| - 1$. Figure **??** shows an example with $h(x) = x \bmod 7$ and $U = \{1, 2, \cdots, 100\}$ and $K = \{15, 46, 80, 96\}$.

FIGURE COMES HERE...

The operations Insert, Delete, and Find can now be implemented as follows:

- `Insert`(k) { $T(h(k)) = k$; }
- `Delete`(k) { $T(h(k)) = NIL$;}
- `Find`(k) { if $T(h(k)) == k$ return 1 else return 0;}

Notice the similarity with the solution using arrays. Instead of looking at slot k, we are now probing at slot $h(k)$. Thus, instead of creating an array of size $|U|$, we use a space of size $O(n)$.

There is however, one small problem. Consider the following example. $K = \{18, 5, 26, 38, 42\}$ and $h(k) = k \bmod 7$. We fix that $U = \{1, 2, \cdots, 100\}$ in the example. It is the case that $h(5) = 5 = h(26)$. So how do we store both 5 and 26 in the same slot? Such a situation is called a *collision*. For a hash table to be useful, we have to deal with these collisons. There are several ways to resolve collisons.

1.3 Collison Resolution Techniques

There are several collison resolution techniques. In this section, we will review some of them.

1.3.1 Chaining

The chaining method of resolving collisons treats each cell as a linked list. All the elements of K that hash to the same cell are maintained as a linked list. So the above example can be treated as follows.

FIGURE HERE Pg 4 END.

Thus the operations `Insert, Delete`, and `Find` take the following form.

```
Algorithm Insert($(k)$)
begin
$v = h(k)$; node −>data = k;
if head$[v]$ = NIL
    head[v] = node;
else
    node −>next = (heads$[v]$)−>next;
    head = node;
End;
```

```
Algorithm Delete$(k)$
begin
$v = h(k)$;
Traverse the list pointed by heads$[v]$ to locate $x$;
delete $k$;
end
```

```
Algorithm Find$(k)$
begin
$v = h(k)$;
Search for $k$ in the list pointed by head$[v]$;
end
```

There is noting specific about using linked lists to resolve collisons. We could have used any other data structure such as trees as will be seen later. But with a good hash function, it is expected that there will be few collisons so that each list will be very small in length.

To analyze hasing with chaining, we introduce a definition. We defined the load factor λ of a hash table T as the ratio of the size of the hash table T to the number of elements in the hash table.

By definition, λ also denotes the average size of each list as each cell on the average contains λ elements. The insert and the delete operations depend on searching a list. So the time taken by the Find operation provides a clue to estimate the time taken the operations Insert and Delete. To analyze the Find operation, the time required is the time taken to find the hash value of the key k, denoted $v = h(k)$, and then traverse the linked list at the vth cell of T. Given that each list has an average length of λ, the average number of elements that will be searched for a successful search is $\lambda/2$.

TO FINISH FROM PAGE 6

1.3.2 Open Addressing

One disadvantage of hashing with chaining is the use of linked lists. This introduces additional program complexity of dealing with pointers and allocating and deallocating memory. An alternative is *open addressing*.

In open addressing schemes, collisons are resolved by tyring alternative cells until an empty cell is found. Formally, let $f : \mathbb{N} \rightarrow \{0, 1, 2, \cdots, |T| - 1\}$ be a function. If cell $h_0 = h(k)$ is not empty, then cells $h_1(k), h_2(k), \cdots$, are tried in succession until an empty cell is found. In the above, $h_i(k) = (h(k) + f(i)) \bmod |T|$, for $i \geq 1$. The function f is called the resolution function. Depending on $f(.)$, several resolution strategies can be defined. We study three strategies: linear probing, quadratic probing, and double hashing.

Linear Probing

In linear probing the function $f(i) = i$. Thus the slots that are probed for placing an element k are $h_0(k) = h(k)$, $h_1(k) = h_0(k)+1, \cdots$, $h_i(k) = h_{i-1}(k)+1$, while the additions are done modulo $|T|$. The following example illustrates the method.

Let $h(x) = x \bmod 10$ and $K = \{34, 52, 76, 84, 98\}$ with $U = \{1, 2\cdots, 100\}$. Notice that key 84 is placed in cell 5 even though $h(84) = 4$. Since cell 4 is already occupied by 34 before inserting 84, we probe cell 5. Since cell 5 is empty, we insert key 84 in cell 5.

Now if we insert 65, it will be inserted in cell 7, since at the time of inserting 65, cells 5 and 6 are already occupied.

The procedures for insert, delete, and find now can be implemented as follows.

```
Algorithm Insert–LinearProbing($k$)
begin
$v = h(k)$;
if $T(v)$ is empty then
    $T(v) = k$
else
        while not done
        begin
            $v = v + 1$;
            if $T[v]$ is empty then
            $T[v] = k$;
                done = true;
        end-while
    end-if
End
```

```
Algorithm Find($k$)
begin
    $v = h(k)$;
    while not done and $v \leq |T|$ do
        if $T[v] = k$
        done = true; return found
        else $v = v + 1$;
    end-while;
end
```

When using linear probing to resolve collisons, it holds that for any hash function h and an element $k \in K$, a slot can always be found as long as the table is not full.

The performance of linear probing thus depends on λ, the load factor. Another practical problem with linear probing is that there may appear blocks of occupied cells even though the table is relatively less loaded. This can happen due to bad inputs or a poor hash function. This problem is called as *primary* *clustering*, indicating that inserting elements into that area is going to take considerably more time than inserting elements elsewhere in the table.

However, it is shown that the average number of probes for insertions and unsuccessful searches is $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$ and $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$ for successful searches.

Quadratic Probing

The collison resolution function $f(x)$ in quadractic probing is simply $f(x) = x^2$. This technique also helps in overcoming the primary clustering problem. Thus to insert an element, the sequence of cells probed is $h_0(k) = h(k)$, $h_1(k) = h(k) + 1$, $h_2(k) = h(x) + 4$, $\cdots$, $h_i(k) = h(k) + i^2$. All the above calculations are done modulo $|T|$.

As an example, if $h(k) = k \bmod 10$ and $K = \{12, 34, 53, 62, 89, 76\}$, the sequence of probes for inserting the element 62 are 2, 3, and 6. Similarly, key 76 is inserted at cell 7 as cell 6 is occupied.

The operations `insert` and `find` can be implemented as follows.

```
Algorithm Insert($k$)
begin
$v = h(k)$;
if $T(v)$ is empty then $T(v) = k$;
else
    $i = 1$;
    while not done
        $v = v + i^2$;
        if $T(v)$ is empty then $T(v) = k$, done = true;
        else $i = i + 1$;
    end-while;
    end-if;
end
```

```
Algorithm Find($k$)
begin
    $v = h(k)$;
    if $T(v)$ is empty then return found;
    $i = i + 1$;
    while not found and $i < |T|$
        $v = v + (i * i) \bmod |T|$;
        if $T[v] = k$
        done = true; return found
        $v = v + 1$;
    end-while;
end
```

One pitfall with quadratic probing is that with a poor choice of hash function, even though the table is not full, an empty cell may not be found for some elements. For example, let $h(k) = k \bmod 16$ and the current table is

TODO

Fortunately however, it can be shown that if the table size is a prime number, and the hash function is $h(k) = k \bmod |T|$, tehn a new element can always be inserted if the table is at least half-empty.

The following theorem formalizes the above.

Theorem 1.3.1 *If quadratic probing is used, and the table size is a prime, then a new element can always be inserted if the table is at least half-empty.*

Proof. Let the tablesize $|T$ be an odd prime. We show that the first $|T|/2$ entries that are probed are distinct as follows. Let i and j be two probed cells so that $h(k) + i^2 = h(k) + j^2$.Then,

$$h(k) + i^2 = h(k) + j^2 \bmod |T| \Rightarrow i^2 - j^2 = 0 \bmod |T|, \text{ or } , (i + j)(i - j) = 0 \bmod |T|.$$

Since $i \neq j$, it can only be the case that $i + j = 0 \bmod |T|$. Since $0 \leq i, j \leq |T|/2$, $i + j \neq 0|T|$. Hence the cells being probed are distinct. $\square$

The proof holds for any k and including the $|T|/2$ alternative cells.

Double Hashing

The collison resolution function for double hashing is $f(k) = i \cdot h_2(k)$ where $h_2(k)$ is another hash function. Thus, having observed a collison at $h(k)$, the sequence of cells probed is $h_2(x), h_3(x), \cdots$, so on. The following example illustrates the procedure.

EXAMPLE TO DO

Thus, the choice of $h_2(k)$ is very crucial in this case. It is also important to ensure that all cells will be probed. A good choice of $h_2(k)$ is $h_2(k) = r - (k \bmod r)$ where r is a prime smaller than $|T|$. The following example provides an illustration.