ESO211 (Data Structures and Algorithms) Lectures 17 to 19

Shashank K Mehta

In these lectures we will discuss a data structure for storing a dynamic set on which we perform operations: *Insert an element*, *Delete an element* and *Search an element.* The elements in the set are from a *universal* set. We assume that each element of the universal set has a unique key (identity). Symbol U denotes the set of all keys of the universal set, n will denote the number of elements currently in the set.

Let $T[0 : m-1]$ denote an array in which we store the elements of the dynamic set or pointers to the elements of the dynamic set. We will call T the *Hash-Table.*

1 Direct Access Table

If the universal set is small (i.e., the table is large enough to store all the elements of the universe), then we can adopt this approach. Define the universal set of keys to be $\{0, \ldots, m-1\}$.

Then if the element corresponding to key i is present in the set currently, then we store it in $T[i]$ or store a pointer to the element in $T[i]$. Otherwise we store a special element *nil* in $T[i]$. This simple approach can search, insert, and delete an element in $O(1)$ time.

2 Closed Hashing

The problem of storing a dynamic set becomes much more complex when $|U| >> m$. In this approach we devise a function $h : U \to \{0, 1, \ldots, m-1\}$. The idea is to store an element x at $T[h(x.key)]$ or to store a pointer to x at that place. Clearly when the universal set is much bigger than m often more than one element of the current set may have keys that are mapped (hashed) to the same slot in the array. This situation is called *collision.*

There are two standard ways to deal with the problem of collision: Chaining and Closed Hashing.

2.1 Chaining

In this case we store all the elements hashed to the same slot, say i, of the table in a single linked list and store a pointer in $T[i]$ to the head of that list. Clearly this approach solves the collision problem in any situation. We assume that the each such list is a doubly linked list with pointers *next* and *prev*. In addition each node stores one element in a field *value*.

Following code searches an element x in the hash-table.

```
$p := T[h(x.key)]$;
while $p \neq nil$ AND $p.value.key \neq x.key$ do
    $p := p.next$;
end
if $p = nil$ then
    Return 'Not found'
else
    Return $p.value$;
end
```

Following code is to insert a node pointed by the pointer p.

```
$k := p.val.key$;
$q := T[h(k)]$;
$T[h(k)] := p$;
$p.next := q$;
$p.prev := nil$;
$q.prev := p$;
```

To delete a node that is being pointed by p.

```
$k := p.val.key$;
if $p.prev \neq nil$ then
    $T[k] := p.next$;
    if $p.next \neq nil$ then
        $p.next.prev := nil$;
    end
else
    $p.prev.next := p.next$;
    if $p.next \neq nil$ then
        $p.next.prev := p.prev$;
    end
end
```

2.2 Chaining: Analysis

As we see that the insertion and the deletion (given a pointer to the node to be deleted) takes $O(1)$ time. Hence we need to analyze the cost of search. Let n denote the number of elements currently present in the set (Hash table).The *load factor* is given by $\alpha = n/m$ where m is the number of slots in the table.

In the worst case all the n elements may be in the same slot making a single linked list for them then an unsuccessful search will cost $(c_1 + c_2 n)$ where c_1 is the cost of computing $h(key)$. Assuming this to be a constant the worst case time complexity is $O(n)$ which is very poor.

Now let us consider the average case cost. For the average case we assume an ideal condition that each element of the set is mapped to each slot with equal probability. Therefore while searching an element x in the Hash table we assume that $Pr(h(x.key) = i) = 1/m$ for any $i \in \{0, 1, \ldots, m-1\}$.

Lemma 1 *If in a hash table collisions are handled by chaining and if the hash function uniformly distributes the keys to the slots, then the expected cost of an unsuccessful search (a search for an element which is not in the set presently) is* $\Theta(1 + \alpha)$.

Proof Let $T(k)$ denote the time to search the element with key k (unsuccessfully) Let n_i denote the number of elements in the linked list of slot i.

So $E[k] = 1 + E[n_{h(k)}] = 1 + \sum_x 1.1/m = 1 + n/m = 1 + \alpha$ where 1 denotes the cost of computing $h(k)$. The unsuccessful search requires that we go through the entire list. The probability of any element of the set being in slot $h(k)$ is $1/m$. ■

Lemma 2 *If in a hash table collisions are handled by chaining and if the hash function uniformly distributes the keys to the slots, then the expected cost of a successful search (a search for an element which is in the set presently) is* $\Theta(1 + \alpha)$.

Proof Suppose the present elements were entered in the table in order $x_1, x_2, \ldots, x_n$. Let the random variable X_{ij} take value 1 if x_j is mapped to slot i. While searching x_j it will take the same time as an unsuccessful search after only $x_{i+1}, x_{i+2}, \ldots, x_n$ because the search first encounters the element which was entered last. So the cost of the searching x_j is $T(k) = c_1 + c_2 \sum_{r=n}^{j} X_{h(k),r}$. So the expected cost is $E[T(k)] = c_1 + c_2 \sum_{r=n}^{j} E[X_{h(k),r}] = c_1 + c_2 \sum_{r=n}^{j} 1/m = c_1 + c_2(n - j + 1)/m$. Assuming that we search any x_j with equal probability, $E[T(k)] = c_1 + c_2.(1/n). \sum_{j=1}^{n}(n-j+1)/m = c_1 + c_2.(n(n+1))/(2nm) = c_1 + c_2.(1/2)(1/m + \alpha)$. Since $1/m \leq \alpha$, the time cost is $\leq c_1 + c_2(\alpha)$ or $\Theta(1 + \alpha)$. ■

2.3 Hash Functions

1. Division Based: We assume that the keys are positive integers. The $h(k) = k(mod m)$. The question to ponder about is: what is a good value of m?

Suppose a key is $k = \sum_j (2^p)^j a_j$, which is often the case. If we take $m = 2^p - 1$, then $h(k) = a_1 + a_2 \ldots$. In this case all permutations will map to the same slot. So this a bad choice of m. It is also found that if m is prime, then this hash function work well. So one tries to choose m to a prime which is far from $2^p - 1$ for any p.

2. Multiplication Based: For some constant $0 < A < 1$, $h(k) = \lfloor m(kA(mod 1)\rfloor = \lfloor m(kA - \lfloor kA \rfloor)\rfloor$. To understand its significance, assume that $A = P/2^q$ where P and q are integers. Then $kA(mod 1)$ is the least significant q bits of kP.

2.4 Universal Hashing

We have observed that the worst case performance of hashing is not good but its average case function is good for a low load-factor. But this claim is valid only when inputs are random and uniformly distributed over all possible inputs. If inputs are adversely biased, then the average case performance can be very poor. So we want to introduce a random parameter in the algorithm such that the average performance remain good while averaging is done over the distribution of the random parameter, not the distribution of the inputs. This way we expect to have good performance even if the inputs are biased.

The idea is to have a large family of hash functions, $\mathcal{H}$, and for each dynamic set we select one of the hash functions randomly (for one dynamic set continue to use the same hash function). We want that for arbitrary keys $k, l \in U$, the probability that $h(k) = h(l)$ be $1/m$ as the hash function is selected from $\mathcal{H}$ with uniform probability. Thus a family of hash functions is called a *universal* if $h(k) = h(l)$ holds for $|\mathcal{H}|/m$ hash functions for any $k, l \in U$.

An example of a universal hash family is as follows. Let p be a prime greater than or equal to $|U|$. Recall that $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field under modulo-p addition and multiplication. Also note that $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$. For any $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$. Then a hash function h_{ab} is given by $h_{ab}(k) = ((ak + b) mod p) mod m$.

Lemma 3 *Has function family $\{h_{ab} | a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$ is universal.*

Proof L ■et k, l be any two distinct keys. Suppose $h_{ab}(k) = h_{ab}(l)$. Let $r = (ak+b) mod p$ and $s = (al+b) mod p$. So $r - s = (a(k-l)) mod p$. Since p is prime and at least as large as $|\mathcal{H}|$, r and s must be distinct (i.e., r.h.s. cannot be zero). Since $\mathbb{Z}_p$ is a field, $a = (r-s)(k-l)^{-1} mod p$ and $b = (r - ak) mod p$. Thus we see that for each choice of (a, b) we have a unique (r, s) and for each choice of (r, s) there is a unique (a, b). Since there are $p(p-1)$ possible choices of (a, b), there must be $p(p-1)$ possible pairs (r, s).

See that $h_{ab}(k) = h_{ab}(l)$ implies that $(r-s) mod m = 0$. For any choice of r, there are $p-1$ possible values of s since $r \neq s$. Thus for $\lceil p/m \rceil - 1$ choices s we will have $(r-s) = 0 (mod m)$. Since $\lceil p/m \rceil - 1 \leq (p+m-1)/m - 1 = (p-1)/m$, we have at most $p(p-1)/m$ hash functions when $h(k) = h(l)$ out of $p(p-1)$ hash functions. Thus if the hash functions are picked with uniform probability, then $Pr(h(k) = h(l)) \leq 1/m$.

3 Open Hashing

Another approach to hashing is suitable only for the case when the load factor is less than 1, i.e., $n < m$. In this case we store all elements in the set the array (not in any linked list). We do not use chaining to resolve the collision. In this case we define a hash function as $h : U \times \{0, 1, \ldots, m-1\} \Rightarrow \{0, 1, \ldots, m-1\}$ such that for any key $k \in U$, $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ is a permutation of $0, 1, \ldots, m-1$. When an element with key k is to be stored in the table we first try the slot $h(k, 0)$. If it is free, then we place the element there, otherwise we try slot $h(k, 1)$, and so on. As long as $n < m$ we will eventually succeed in finding a slot for the new element.

In order to search an element x with key k, we probe $h(k, 0)$. If we do not find the element, then check $h(k, 1)$ slot, so on. We stop when we get the desired element element or when we first encounter an empty slot without getting x or when all the m slots are checked and x is not found. The second instance for stopping is justified because if x was in the table, then we would not have left this empty slot and put it at a later slot in the trail: $h(k, 0), h(k, 1), \ldots, h(k, m-1)$.

Thus far we have assume that there is not delete operation. In open hashing delete operation causes some difficulty. Suppose we insert an element x in a set which had y was encountered on the trail and x was placed at a later slot. Now suppose we delete y and then search x. In this case we will encounter a black slot (originally of y) and conclude that x is not present, while it is in the set. To avoid this error we place a special symbol 'Deleted' in the slot from where we delete an item. We treat such slots as occupied while searching an element but treat it as a empty while inserting.

The insertion $Insert(T, x)$ is given as follows:

```
$i := 0$;
$k := x.key$;
while $i < m$ do
    $j := h(k, i)$;
    if $T[j] = nil$ or $T[j] = Deleted$ then
        $T[j] := x$;
        Return $j$;
    else
        $i := i + 1$;
    end
end
Print "Overflow";
```

The search routine $Search(T, x)$ is as follows.

```
$i := 0$;
$k := x.key$;
while $i < m$ do
    $j := h(k, i)$;
    if $T[j] = Deleted\ or\ T[j].key \neq k$ then
        $i := i + 1$;
    else
        Return $j$
    end
end
Print "Not Present";
```

3.1 Hash Functions

Ideally we want that the search trail $h(k, 0), h(k, 1), dots, h(k, m-1)$ be any of the possible $m!$ sequences with equal probability. But in reality that cannot be achieved while keeping the function $h()$ simple. Here are some of the functions used in practice.

(1) Linear Probe: $h(k, i) = (h'(k) + i)(modm)$, where $h'()$, an auxiliary hash function, is any hash function used for chaining. In this case only m trails are possible, namely, $(0, 1, \ldots, m-1), (1, 2, \ldots, m-1, 1), (2, 3, \ldots, n-1, 0, 1), \ldots, (m-1, 1, 2, \ldots, m-2)$. Often data may have neighboring keys. In that case there will be heavy overlap. Suppose $m = 100$ and there are only 6 elements in the set with keys $0, 1, 2, am, am+1, am+2$. Then slots $0, 1, 2, 3, 4, 5$ will be occupied. The average search time will be $15/6 = 2.5$. This is called *primary clustering.*

(2) Quadratic Probe: $h(k, i) = (h'(k) + c_1.i + c_2, i^2)(modm)$. This also has only m distinct trails. But if the keys of the elements in the set do not map to neighboring slots, then the clustering problem is less severe. Here the clustering problem is called *secondary clustering.*

(3) Double Hashing: $h(k, i) = (h_1(k) + ih_2(k))(modm)$. This is a significantly better function compared to the previous functions. It has m^2 distinct trails. But, on the down side, it requires the computation of two auxiliary functions.

Two most useful sets of auxiliary hash functions are as follows. In each h_1 is any good hash function used in chaining.

(i) $h_2(k)$ is an odd valued function and $k = 2^r$.

(ii) $h_2(k)$ is any good hash function used in chaining and m is prime.

3.2 Analysis

For our analysis we will define α' as the number of elements plus the number of 'Deleted' symbols divided by m. So $\alpha' \geq \alpha$. We assume that $\alpha' < 1$. By n we will denote the number of elements plus the number of 'Deleted' symbols.

Lemma 4 *The average number of probes in an unsuccessful search is at most* $1/(1-\alpha')$*, assuming the ideal condition (i.e., all* $m!$ *trails are equally probable).*

Proof Let X denote the number of probes (checking a slot if it contains the element we are searching). In this case we are searching an element which is not in the set. A search is successful when we encounter a blank slot because in that case we stop the search. Let A_i denote the probability if i-th probe is unsuccessful (finds a non-empty slot).

Then $Pr(X \geq i) = Pr(A_1 \cap A_2 \cap \cdots \cap A_{i-1})$ because if the $i-1$st probe is unsuccessful, then we will make at least i probes. SO $Pr(X \geq i) = Pr(A_1) \cdot Pr(A_2|A_1) \cdot Pr(A_3|A_1 \cap A_2) \ldots Pr(A_{i-1}|A_1 \ldots A_{i-2})$.

There are n elements in the set so the first probe will be successful if we hit one of the empty slots. SO $Pr(A_1) = n/m$. In general $Pr(A_j/A_1 \cap \ldots A_{j-1}) = (n-j+1)/(m-j+1)$ because we have already seen $j-1$ slots and the success happens if we hit one of the $m-n$ vacant slots out of remaining $m-j+1$ slots. So $Pr(X \geq i) = (n/m)(n-1)/(m-1)\ldots(n-i+1)/(m-i+1) \leq (n/m)^i = \alpha'^i$.

The expected number of probes is $E[X] = \sum_{i=1}^{m} i.Pr(X = i) = \sum_{i=1}^{\infty} Pr(X \geq i) \leq \sum_{i=1}^{\infty} \alpha'^i = 1/(1-\alpha')$ ■

Lemma 5 *The average number of probes in a successful search is at most $1/\alpha' \log 1/(1-\alpha')$, assuming the ideal condition (i.e., all $m!$ trails are equally probable).*

Proof Search for an element probes exactly those slots which were probed during its insertion. Suppose we have inserted $x_1, x_2, \ldots, x_n$ in that order. It is possible that some of them have been replaced by 'Deleted" symbol. So the number of probes for searching x_i is same as the probes in an unsuccessful search after insertion of $x_1, \ldots, x_{i-1}$. So the expected number of probes in searching x_i is at most $1/(1-(i-1)/m)$. So the expected number of probes for an arbitrary element of the set is at most $(1/n)\sum_{i=0}^{n-1} 1/(1-i/m) = (1/n)\sum_{i=0}^{n-1} m/(m-i) = (1/\alpha')\sum_{i=m-n+1}^{m} 1/i \leq (1/\alpha')\log(m/(m-n+1)) = (1/\alpha')\log(1/(1-\alpha'-(1/m)))$. ■