# ESO207 (Data Structures and Algorithms) Lecture Notes Set 6

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 each such list is a doubly linked list with pointers *next* and *prev*. In addition each linked list 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 ≠ nil AND p.value.key ≠ x.key do
    p := p.next;
end
if p = nil then
    Return 'Not found'
else
    Return p.value;
end
```

Following code inserts a linked-list node in $T$, which is pointed by the pointer $p$.

```
k := p.value.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 = nil then
    T[k] := p.next;
    if p.next ≠ nil then
        p.next.prev := nil;
    end
else
    p.prev.next := p.next;
    if p.next ≠ 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. We will now analyze the cost of the search operation. Let $n$ denote the number of elements currently present in the set (Hash table). The *load factor* is defined to be $\alpha = n/m$ 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 an element with key $k$ (unsuccessfully). Let $n_i$ denote the number of elements in the linked list of slot $i$. The unsuccessful search requires that we go through the entire list. The probability of any element of the set getting mapped to slot $h(k)$ is $1/m$.

So $E[T(k)] = c_1 + E[n_{h(k)}] = c_1 + \sum_{i=1}^{n} c_2.1/m = c_1 + c_2 n/m = c_1 + c_2\alpha$ where $c_1$ denotes the cost of computing the function $h(k)$. ■

**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 $x_1, x_2, \ldots, x_n$. While searching $x_j$ it will take the time proportional to the number of elements from the list $x_n, x_{n-1}, \ldots, x_{j+1}, x_j$ which were mapped to slot $i$ because the list is searched from last to first.

We will make two assumptions. First is that each element in the current dynamic set have equal probability of being hashed to any slot of the table. Second is that any element is searched with equal probability. Define a random variable $X_{ij}$ which takes value 1 if $x_j$ is mapped to slot $i$. Otherwise it is zero.

So the cost of searching $x_j$ is $T(j) = c_1 + c_2 \sum_{r=n}^{j} X_{h(j),r}$. The expected cost is

$$\begin{aligned} & E[T(j)] \\ &= c_1 + c_2 \sum_{r=n}^{j} E[X_{h(j),r}] \\ &= c_1 + c_2 \sum_{r=n}^{j} 1/m \\ &= c_1 + c_2(n-j+1)/m. \end{aligned}$$

Further since $x_j$ can be any element in the present set with equal probability,

$$\begin{aligned} & E[T(j)] \\ =& c_1 + c_2.(1/n).\sum_{j=1}^{n}(n-j+1)/m \\ =& c_1 + c_2.(n(n+1))/(2nm) \\ =& c_1 + c_2.(1/2)(1/m+\alpha). \end{aligned}$$

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 $m$?

If we take $m = 2^p$, then $h(k)$ will be the last digit in the $2^p$-base representation of $k$. This may perform reasonably well as consecutive keys will be mapped to different slots.

Suppose the key is expressed as a $2^p$ based number $\langle a_r \ldots a_2 a_1 a_0 \rangle$. 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 too is a bad choice of $m$.

It is found that this hash function work well if $m$ is taken to be a prime number.

2. Multiplication Based: For some constant $0 < A < 1$, $h(k) = \lfloor m(k.A(mod\ 1)\rfloor = \lfloor m(k.A - \lfloor kA\rfloor)\rfloor$.

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 $kP$. Thus the value of $m$ does not have any significant affect. Thus it maps reasonably randomly.

## 2.4 Universal Hashing

We have observed that the worst case performance of hashing is not good but its average case performance 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 also be very poor. So we want to introduce a random parameter in the algorithm such that the average performance remains 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 *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 number greater than or equal to $|U|$. Recall that $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field under modulo-$p$ addition and modulo-$p$ multiplication. Also note that $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$. For any $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$ define a hash function $h_{ab}$ as $h_{ab}(k) = ((ak + b) mod\ p) mod\ m$.

**Lemma 3** *Hash function family $\mathcal{H}_{p,m} = \{h_{ab} | a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$ is universal.*

***Proof*** Let $k, l$ 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 $r - s = (a(k - l)) mod\ p$. Note that $a < p$, $|k - l| < |U| \leq p$, and $p$ is prime. So $r - s \neq 0$. Since $\mathbb{Z}_p$ is a field and $k - l \neq 0$, $(k - l)^{-1}$ exists. So $a = (r - s)(k - l)^{-1} mod\ p$ 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 $(r, s)$. Thus each pair $r, s \in \mathbb{Z}_p$ forms a valid pair if $r \neq s$ and it corresponds to a unique $h_{ab}$.

Observe 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 $r \neq s$. So there are $p - 1$ hash functions possible with a fixed $r$. Of these pairs $(r, s)$ there are at most $\lceil p/m \rceil - 1$ pair are such that $(r - s) = 0 (mod\ p)$. Note that $lceilp/m\rceil - 1 \leq (p + m - 1)/m - 1 = (p - 1)/m$. So at most $(p - 1)/m$ hash functions out of $p - 1$ hash functions are such that $h_{ab}(k) = h_{ab}(l)$. Thus keys $k$ and $l$ collide with probability $\leq ((p - 1)/m)/(p - 1) = 1/m$.

Thus if the hash functions are picked with uniform probability, then $Pr(h(k) = h(l)) \leq 1/m$. ■

Following lemma shows how a randomly chosen hash function from a universal set of hash functions performs if available space in the hash table is large enough.

**Lemma 4** *If $n$ items are hashed into a table of size $m = n^2$ using a randomly chosen hash function from a universal family of hash function, then the probability of no collision is at least $1/2$.*

***Proof*** There are $\binom{n}{2}$ possible pairs in a set of $n$ items. Probability of a particular pair getting mapped to the same slot under a randomly chosen hash function is $1/m$. Let the random variable $X_{ij}$ indicate the collision between items $i$ and $j$, i.e., $X_{ij} = 1$ if $i$ and $j$ collide. Otherwise $X_{ij} = 0$. So $E[X_{ij}] = 1/m$. Then the total number of colliding pairs is $X = \sum_{i<j} X_{ij}$. So $E[X] = E[\sum_{i<j} X_{ij}] = \sum_{i<j} E[X_{ij}] = \binom{n}{2}.1/m = (n(n-1)/2).(1/n^2) < 1/2$.

From Markov's inequality, $Prob(X \geq 1) \leq E[X]/1 \leq 1/2$. ■

## 2.5 Two Level Hashing Scheme

Previous result shows that with a good probability hashing can be made collision free. But the price to be paid for this is also very high, namely, the demand for space in the hash table is also very high. Here we will develop a strategy which gives as low collision probability but without such a heavy memory cost.

We will devise an alternative to chaining based hashing. Consider a scheme in which a hash function $h$ maps the set of keys to a primary hash table $T$ of $m$ slot. Then all the items mapped to a particular slot $j$ are futher mapped to secondary hash table $T_j$ of $m_j$ slots using a hash function $h_j$. Then what will be the collision probability and what will be the space requirement?

This scheme is most suitable to static sets. A static set is one in which once keys are entered, no addition/deletion is performed. Suppose after the keys are mapped to the primary hash table, $n_j$ items are mapped to the $j$-th slot. Assign $m_j = n_j^2$ slots to $T_j$. Let us assume that the primary table has $m = n$ slots where $n$ is the number of elements in the static set. Let us assume that the primary hash function is randomly selected from the universal family $\mathcal{H}_{p,m}$ and the $j$-th hash function is similarly chosen from $\mathcal{H}_{p,m_j}$, for all $j$. From the previous result the probability of zero collision is at least $1/2$. The next results will give the estimate of the required memory in this scheme.

**Lemma 5** *In the two-level scheme with $m = n$ and $m_j = n_j^2$ $\forall j$, $E[\sum_j m_j] < 2n$.*

***Proof*** $\sum_{j=1}^{m} m_j = \sum_{j=1}^{n} n_j^2 = 2\sum_j \binom{n_j}{2} + \sum_j n_j = 2\sum_j \binom{n_j}{2} + n$. So the expected value of secondary space is $E[\sum_{j=1}^{m} m_j] = n + 2.E[\sum_j \binom{n_j}{2}]$. Note that $\sum_j \binom{n_j}{2}$ is the number of pairs of keys that were mapped to the same slot. The property of universal hashing is that any two keys are mapped to the same slot with probability $1/m$. So $E[\sum_j \binom{n_j}{2}] = \binom{n}{2}.1/m = n(n-1)/(2n) < n/2$. So $E[\sum_{j=1}^{m} m_j] < 2.(n/2) + n = 2n$. ■

The primary hash table uses $m = n$ slots se we have the following result.

**Corollary 1** *In the two-level scheme with $m = n$ and $m_j = n_j^2$ $\forall j$, expected size of the total memory is $3n$*

The next result gives the tail estimate.

**Corollary 2** *In this scheme the probability that secondary space exceeds $4n$ is less than $1/2$*

***Proof*** We want to know the probability $Pr(\sum_j m_j > 4n)$. Using Markov's inequality $Pr(\sum_j m_j \geq 4n) \leq E[\sum_j m_j]/4n \leq 2n/4n = 1/2$. ■

# 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 of the set in 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 assumed that there is not delete operation. In open hashing delete operation causes some difficulty. While inserting a new element $x$, suppose another element $y$ was encountered in the trail. Subsequently $y$ is deleted and later $x$ is searched. In this case we will encounter a blank slot (originally of $y$) and conclude that $x$ is not present, in spite of the fact that 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 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";
```

**Algorithm 1:** $InsertT, x$

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 ≠ k then
        i := i + 1;
    else
        return j
    end
end
Print "Not Present";
```

**Algorithm 2:** $SearchT, x$

## 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: $h(k,i) = (h'(k) + i)(mod\ m)$, where $h'()$, an auxiliary hash function, is any hash function used for chaining. In this case only $m$ trails are possible, namely, $(0, 1, \ldots, m-1), (1, 2, \ldots, m-1, 0), (2, 3, \ldots, n-1, 0, 1), \ldots, (m-1, 0, 1, 2, \ldots, m-2)$. Note that there is a big overlap between any two trails. If $h'$ values to two keys are closer, the then their trails have higher overlap.

Often the dynamic set may have neighboring keys. In that case there will be heavy overlap among the trails of these data items. Suppose $m = 100$ and there are only 6 elements in the set with $h'$-keys $0, 1, 2, 0, 1, 2$. Then slots $0, 1, 2, 3, 4, 5$ will be occupied. The average successful search time will be $(1 + 1 + 1 + 4 + 4 + 4)/6 = 15/6 = 2.5$. This form of overlap of trails is called *primary clustering*.

(2) Quadratic Probe: $h(k,i) = (h'(k) + c_1.i + c_2, i^2)(mod\ m)$. Let us first show that quadratic probe also results in a trail visiting all them slots, for a suitable choice of $m$. Suppose $h(k,i) = h(k,j)$ for some $0 \leq i < j \leq m-1$. Then $c_1.(j-i) + c_2.(j-i).(j+i) = 0(mod\ m)$ or $(j-i)(c_1 + c_2(i+j)) = 0(mod\ m)$. Suppose $m = 2^r$, $c_1$ is odd and $c_2$ is even. Then $c_1 + c_2(i+j)$ will always be odd. So $m$ must divide $j - i$. But both, $i$ and $j$, are less than $m$ so $m$ cannot divide $j - i$ unless $j = i$. Hence $h(k,i) = h(k,j)$ if and only if $j = i$.

This scheme also has only $m$ distinct trails. But two trails are unlikely to overlap at more than one consecutive locations. Hence the primery clustering problem is not likely. There might be some clustering problem in this situation too. It is called *secondary clustering*.

(3) Double Hashing: $h(k,i) = (h_1(k) + ih_2(k))(mod\ m)$. 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 $m = 2^r$.

(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 the extended load-factor $\alpha'$ as the number of elements in the table 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 so $\alpha' = n'/m$.

**Lemma 6** *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. Let $A_i$ denote the probability if $i$-th probe finds a non-empty slot which either contains some other element or the "Deleted"-label.

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

The number of elements plus "Deleted" labels is $n'$ in the set so the first probe will terminate the search 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 each contains an element and they are out of further consideration. 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 7** *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 $(1/n')\sum_{i=0}^{n'-1} 1/(1-i/m) = (1/n')\sum_{i=0}^{n'-1} m/(m-i) = (1/\alpha')\sum_{i=m-n'+1}^{m} 1/i \leq (1/\alpha')\log\,(m/(m-n'+1)) = (1/\alpha')\log\,(1/(1-\alpha'+(1/m))) \leq 1/\alpha')\log\,(1/(1-\alpha'))$. ■