3/14

Hashing:

A hash table is a generalization of the simpler notion of an ordinary array. Many applications require a dynamic set that supports only the dictionary operations Insert, Search, Delete. A hash table is an effective data structure for implementing dictionaries.

If an application needs dictionary operations with keys drawn from the universe $U = \{0, 1, \dots, u-1\}$, where m is not too large, then we can direct-address table method by using an array $T[0, \dots, u-1]$.

what about when the universe U is so large that storing a table T of size $|U|$ may be impractical and in addition the set K of keys actually stored may be so small relative to U (e.g. keys in a dictionary)

With hashing, we can reduce space to $\theta(|K|)$ with search time only $O(1)$ (though in average).

With direct addressing, an element with key k is stored in slot k. with hashing, this element is stored in slot $h(k)$; that is, a hash function h is used to compute the slot from the key k. Here h maps the universe U of keys into the slots of a hash table $T[0, \dots, m-1]$. ($h: U \rightarrow \{0, 1, \dots, m-1\}$). we say that an element with key k hashes to slot $h(k)$ or $h(k)$ is the hash value of key k.

The main issue is that two keys may hash to the same slot, a collision, for which we see effective techniques in this session. One idea is to make h appear to be "random" to avoid collision

Hashing functions: The average performance of hashing depends how well it distributes the keys in the m slots, i.e. we consider integers, fixed length arrays (of integers) (characters) and variable length arrays, "strings",

* hashing integers: The original proposal of Cater and Wegman was to pick a prime $p \geq u$ and define $h_{a,b}(x) = ((ax+b) \bmod p) \bmod m$, where a, b are randomly chosen integers mod p with $a \neq 0$.

* hashing fixed-length array: The input is a vector $\bar{x} = (x_0, \dots, x_{h-1})$ of h integers (or bytes/characters).

$h(\bar{x}) = \left[\sum_{i=0}^{h-1} h^i_{a,b}(x_i)\right] \bmod m$, where each $h^i_{a,b}$ is constructed by choosing a, b randomly mod p with $a \neq 0$; where $p \geq u$ again is a prime.

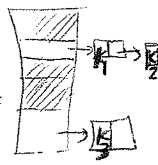
* hashing strings: if the length of the string can be bounded by a small number it is best to use the fixed-length array solution. Now assume that we want to hash $\bar{x} = (x_0, \dots, x_l)$, where a good bound on l is not known a priori. Again if $x_i \in [0, \dots, u-1]$, let $p \geq \max\{u, m\}$ be a prime and define $h_c(\bar{x}) = h_{a,b}\left(\left(\sum_{i=0}^{l} x_i c^i\right) \bmod p\right)$, where a, b, and c are randomly chosen integers mod p with $a, c \neq 0$.

A hash function is called universal if $\forall x, y \in U, x \neq y: \Pr[h(x) = h(y)] \leq \frac{1}{m}$. Note that this is the ideal case in which any two keys of the universe collide with prob. at most $\frac{1}{m}$. All functions above provide such guarantee. Note, that without randomness, we cannot obtain such functions. Universal hashing guarantees any given element is equally likely to hash into any slots; "uniform hashing"

Any how, if we use any approach then there is a chance of collision. so what are the solutions:

1) Collision resolution by chaining:

In chaining, we put all the elements that hash to the same slot in a linked list. slot j contains a pointer to the head of the list of all stored elements that hash to j; if there are no such elements, slot j contains NIL.



The dictionary operations are as follows:

```
Hash_Insert (T, x)
    insert x at the head of list T[h(key[x])]

Hash_Delete (T, x)
    delete x from the list T[h(k)]

Hash_Search (T, K)
    Search for an element with key k in list T[h(k)]
```

The worst case running time for insertion is $O(1)$ though for delete and search is $O(size(T[h(x)]))$. Because of the uniformity property, the average length of each list is $\frac{n}{m}$, where n is # of keys and m is the size of T. Thus running time for delete and search is in $O(1+\frac{n}{m})$.
Thus if m is at least proportional to n, then $n = O(m)$ and the running time is $O(1)$ for all operatio

2) Open addressing: linear Probing:

In open addressing, all elements are stored in the hash table itself and it avoids pointers altogether. Thus in open addressing, the hash table can fill up so that no farther insertion can be made (and thus waste of memory)

Instead in the case of collision, we need to examine, probe, a sequence of slots.
In the simplest form of open addressing, we are using the method of linear probing as follows: given a key k, the first slot probed is $T[h(k)]$. We next prob slot $T[h(k)+1]$ and so on up to slot $T[m-1]$. Then we wrap around to slots $T[0], T[1], \dots,$ until we finally prob slot $T[h(k)-$
Note that here the intial probe position determines the entire probe. In insertion, we probe according to this order until we find an empty space or give error "hash table overflow".
for search, we probe accordingly until we find the element or an empty slot.
Deletion from an open-address hash-table is difficult. When we delete a key from slot i, we need to mark it as "Deleted" instead of NIL such that search skips over it, but insert can insert it. Again if the load factor $\alpha = \frac{n}{m}$ is constant insert and search can take $O(1)$ in average. However Delete can .. cause problems and make operations no longer dependent in α.
For this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted so often. Sometime if the number of deletions is a lot we can do re-hash (i.e. hash it in a new array).