Lecture 18

Hashing

Definition 18.1. A ***hash function*** is a function h that outputs an integer value for each key. A ***hash table*** is an array implementation of the table ADT, where each key is mapped via a hash function to an array index.

A hash function should:

- be computable quickly (constant time);
- produce hashed values that are uniformly distributed if keys are drawn uniformly at random;
- produce hash values that are "spread out" for keys that are "close".

A hash function should be deterministic, but appear "random" – in other words it should pass some statistical tests (similar to pseudo-random number generators).

Example 18.2. Suppose that we hash dictionary words by letting $h(w)$ be the address of the last letter of w, converted to lower case (so there are 26 slots in the hash table, numbered $0,\ldots,25$ to correspond to $a,b,\ldots,z$).
List two words that are very "different" yet have the same hash value. Which of the criteria listed above are satisfied by this hash function?

h(to) = 0 → 15 − 1 = 14
h(kangaroo) = 0 → 15 − 1 = 14
} Basic hash function

Four basic methods for choosing a hash function are ***division***, ***folding***, ***middle-squaring***, and ***truncation***.

Division assuming the table size is a prime number m and keys, k, are integers, set the hash function $h(k) = r(k,m) = k \mod m$, the remainder after integer division by m.

Folding an integer key k is divided into sections and the value $h(k)$ combines sums, differences, and products of the sections. Eg split $k = 013402122$, into 013, 402, and 122, and add to get $h(k) = 537$.

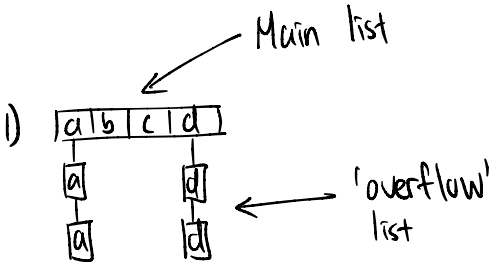
Middle-squaring a middle section of an integer key k, is selected and squared, then a middle section of the result is the value $h(k)$. Eg $k = 013402122$ results in $402^2 = 161604$so middle four digits give $h(k) = 6160$ in the range $[0,\ldots,9999]$).

Truncation parts of a key are simply cut out and the remaining digits, or bits, or characters are used as the value $h(k)$. Eg take last three digits of $k = 013402122$ to get $h(k) = 122$. Fast, but keys do not scatter randomly and uniformly over the hash table indices so use in conjunction with something else.

18.1 Collisions

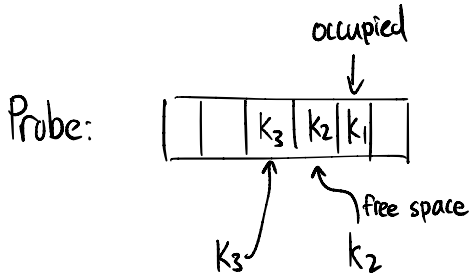
The number of possible keys is usually much larger than the actual number of keys so we do not want to allocate an array large enough to fit all possible keys. But this means that hash functions are not 1-to-1. When two keys may be mapped to the same index we call it a ***collision***.

We need a ***collision resolution policy*** to prescribe what to do when collisions occur. We assume the first-come-first served model for resolving collisions.



We consider two collision resolution policies:

1. ***Chaining*** uses an "overflow" list for each element in the hash table.
2. ***Open addressing*** uses no extra space. Every element is stored in the hash table. If it gets overfull, we can reallocate space and rehash.



When a collision occurs in open addressing, we can

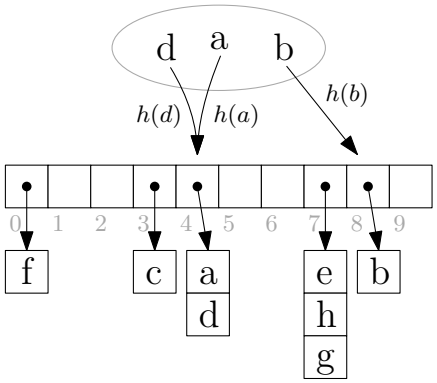
- ***probe*** nearby for a free position (linear probing, quadratic probing, ...);
- go to a "random" position by using a second-level hash function (***double hashing***).
- We **employ the convention that we always probe to the left**.

18.2 Collision resolution via chaining

- Elements that hash to the same slot are placed in a list. The slot contains a pointer to the head of this list.
- Insertion can then be done in constant time.
- Deletion can be done in constant time with a doubly linked list, for example.

- A drawback is the additional space overhead. Also, the distribution of sizes of lists turns out to be very uneven.

Example 18.3. A hash table with chaining, showing empty chains and chains of length 1, 2 and 3.



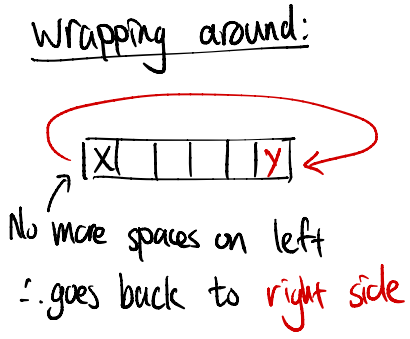
Example 18.4. Using the hash function from Example 18.2 to hash the following 14 words using chaining. What is the length of the longest chain formed? The words are:
abacus, abaci, acrid, grampus, radii, celery, homely, comely, xylitol, animal, seminar, seminal, radar, box

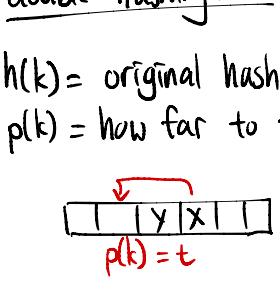
s ī d y l r x
s ī y l r
y l

Length of longest chain: 3 ('y' and 'l')

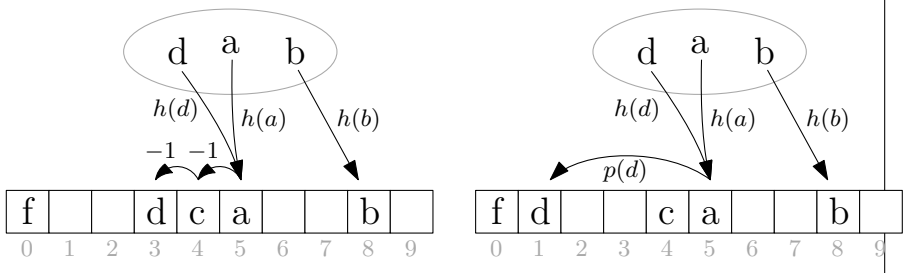
18.3 Collision resolution via open addressing

- Every key is stored somewhere in the array; no extra space is required.
- If a key k hashes to a value $h(k)$ that is already occupied, we probe (look for an empty space).
  - The most common probing method is ***linear probing***, which moves left one index at a time, wrapping around if necessary, until it finds an empty address. This is easy to implement but leads to ***clustering***.
  - Another method is ***double hashing***. Move to the left by a fixed step size t, wrapping around if necessary, until we find an empty address. The difference is that t is not fixed in advance, but is given by a second hashing function $p(k)$.





Example 18.5. The hash table on the left uses linear probing, so d is inserted two left of $h(d) = h(a)$, because $h(d) - 1$ is already occupied. The hash table on the right uses double hashing, so d is moved $p(d)$ to the left of $h(d) = h(a)$.



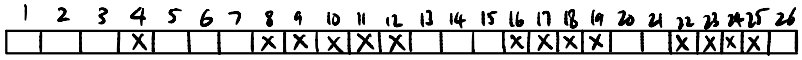
Example 18.6. If we use the hash function for dictionary words, open addressing with linear probing, and the list of 14 words as above, where does the word "comely" hash to in the table? If instead of linear probing we use double hashing with $p(w)$ defined to be the number of vowels in w, where does "comely" end up?

Linear probing = 23

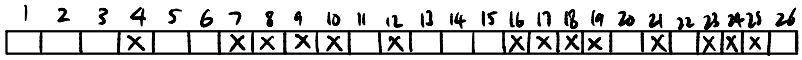
Double hashing = 21

s=19 i=9 d=4 y=25 l=12 r=18 x=24

LP:



DH:



Linear probing is simple but is prone to *clustering* – large stretches filled entries form making average search times longer.

18.4 Analysis of hashing

Cost is measured by the number of key comparisons or probes.

Insertion has the same cost as an unsuccessful search, plus one, provided the table is not full. Thus the average cost of a successful search is the average of all insertions needed to construct the table from empty.

We often use the ***simple uniform hashing*** model. That is, each of the n keys is equally likely to hash into any of the m slots. So we are considering a "balls in bins" model.

If n is much smaller than m, collisions will be few and most slots will be empty. If n is much larger than m, collisions will be many and no slots will be empty. The most interesting behaviour is when m and n are of comparable size. We define the ***load factor*** to be $\lambda := n/m$.

$n > m$ (eg chaining)

Example 18.7. Compute the load factor for the hash tables shown in Examples 18.3 and 18.5.

18.3) $m = 10$, $n = 8$, $\therefore \lambda = \frac{8}{10} = 0.8$

18.5) $m = 10$ slots, $n = 5$ items $\therefore \lambda = \frac{5}{10} = 0.5$