11 Hash tables: chaining

11.1 Lower bound for binary trees

- Recall that '$f(n)$ is $O(g(n))$' means that for some real number c and index N, $f(n) \leq cg(n)$ for all $n \geq N$.
- Or: $f(n) \leq cg(n)$ for almost every n.
- We say that f is $o(g)$ if for *every* $c > 0$, $f(n) < cg(n)$ for almost every n.
- We write 'f is $\Omega(g)$' when f *is not* $o(g)$:
  - for *every* positive c,
  - for infinitely many n,
  - $f(n) \geq cg(n)$.

(11.1) Corollary *The depth of an n-node binary tree is* $\Omega(\log n)$. ∎

Therefore storing keys in binary search trees has worst-case cost $\Omega(\log n)$.
Something stronger can be said:

(11.2) Lemma *The* average depth *of nodes in an n-node binary tree is* $\Omega(\log n)$.
Equivalently: the internal path length is $\Omega(n \log n)$. ∎

11.2 Hash tables

So there is a lower bound to the average-case performance of binary search trees used to store and retrieve keys.

Hash tables are completely different. The idea is simple: store keys in an array $A[0 \ldots m-1]$, where m is suitably chosen. Use a *hash function* $h()$ which converts a key into an index in the range $0 \ldots m-1$; then to find if a key x is stored in the array, look at $A[h(x)]$. The best plan would be to ensure that h is an injective function (on the given supply of keys). This is called 'perfect hashing.'

It is highly unlikely, though it can be arranged with hindsight, that is, if all the keys are known in advance [Fredman, Komlós, Szemerédi,'Storing a sparse table with $O(1)$ worst-case access time'].

What we hope for is that the hash function is suitably 'random,' or unbiased. We just make this assumption and derive average-case performance of various strategies.

We shall not worry about computing the hash address. Sometimes $x \mod M$ is good enough, where x is a bitstring regarded as an integer and M is the array size. The point is that the hash function should be fairly random, i.e., the values of $h(x)$ are uniformly distributed in the array's range.

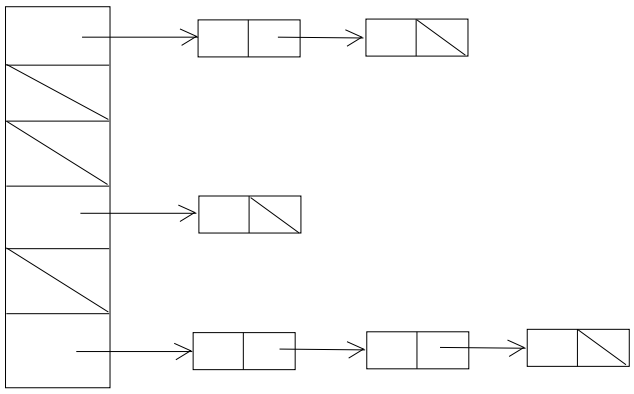

Figure 1: chaining.

11.3 Chaining

So we must expect to store groups of keys with the same hash-value. The simplest way to do this is with *chaining*, where the array contains pointers to linked lists, and when a key x with hash-value i is stored, it is stored in the i-th list in the array (Figure 1).

Put another way, the keys are stored in 'buckets' (linked lists).

Also the general practice is to insert a key, which is not stored, at the place where the search ends. For chaining that means at the end of the list searched.

(11.3) Successful and unsuccessful searches. When analysing hashing strategies, there is often a difference between the cost of an 'unsuccessful search,' where a key is not found and is then inserted, or a 'successful search' where the key has been inserted previously.

(11.4) Analysis of chaining. Suppose that beginning with an empty table, the keys $x_1, \ldots, x_n$ are inserted. When the r-th key is inserted, there are $r-1$ stored; and with suitable assumptions of randomness, the length of the 'bucket' searched for the r-th key is roughly $(r-1)/m$. Summing from $r = 1$ to n, and dividing by n to get the average search, we get

$$\frac{1}{n}\frac{n(n-1)}{2}\frac{1}{m} \approx \frac{n}{2m}.$$

This is the average cost of unsucessful search, which is roughly the same as the cost of successful searching as well.