Chapter 11 - Hashing

- Goal is to support dictionary operations:
  - insert(D, x)
  - search(D, x)
  - delete(D, x)

- We saw how to support these operations using a binary search tree. Problems: complex structure, pointers are expensive (storage), make programming hard to debug, maintain etc. Also takes O(h) to perform these operations. h can be ~ n in the worst-case.
- With balancing, we can make h = O(log n). AVL, 2-3 tree, treap etc.

- Can we achieve all these operations in O(1) steps per operation?

- Not possible in the worst-case. Possible on average assuming that the operations are performed randomly.

Overall idea behind hashing

A hash table is a generalization of an ordinary array.

- With an ordinary array, we store the element whose key is k in position k of the array.
- Given a key k, we find the element whose key is k by just looking in the kth position of the array. This is called ***direct addressing***.
- Direct addressing is applicable when we can afford to allocate an array with one position for every possible key.

We use a hash table when we do not want to (or cannot) allocate an array with one position per possible key.

- Use a hash table when the number of keys actually stored is small relative to the number of possible keys.
- A hash table is an array, but it typically uses a size proportional to the number of keys to be stored (rather than the number of possible keys).
- Given a key k, don't just use k as the index into the array.
- Instead, compute a function of k, and use that value to index into the array. We call this function a ***hash function***.

Typical hashing scenario

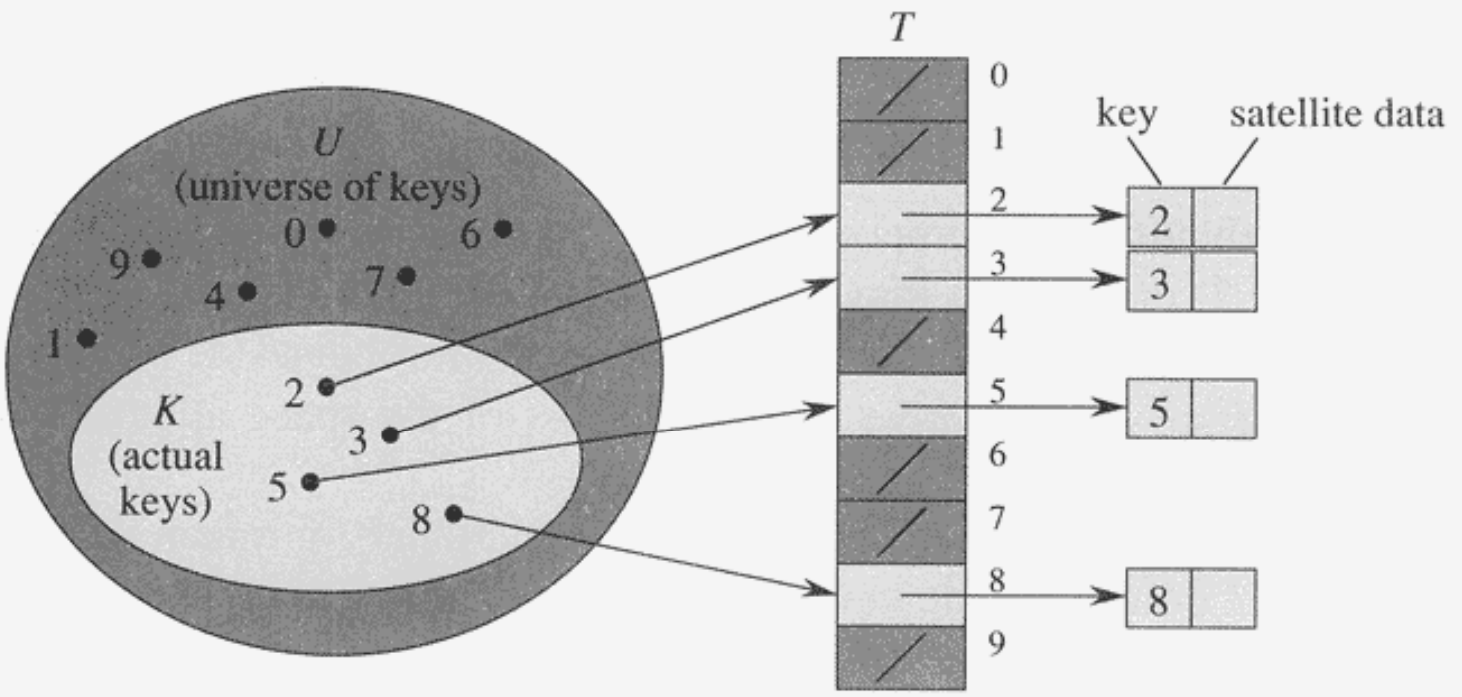


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

Collisions

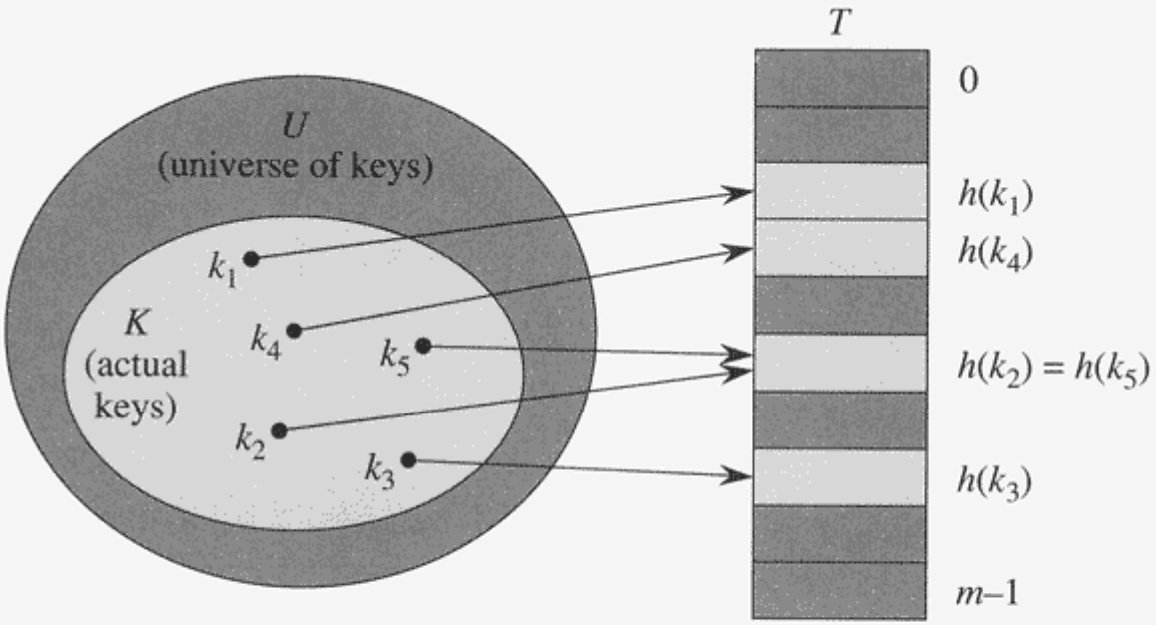


Figure 11.2 Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

Hashing

The idea is to store the keys in an array.

U - space of all possible keys.

S - set of keys actually present.

T - array in which the keys are stored.

A typical relation among these is:

$|S| << |T| << |U|$.

Basic scheme: Choose a hash function h that maps every
key from the set U to an integer between 0 and $|T| - 1$.

Insert (x): set $T[h(x)] = 1$.

Search(x): return $(T[h(x)] == 1)$.

Delete(x): T[*h(x)*] = 0;

The above methods clearly do not work if there are collisions.

We say that a collision occurs if there are x, y (x != y) such that *h(x) = h(y)*.

Open Addressing/chaining: This offers a simple solution to collisions. All the colliding keys are kept in a linked list.

Is it better to keep the list sorted? Not worth it. Too much overhead.

Collision resolution by chaining

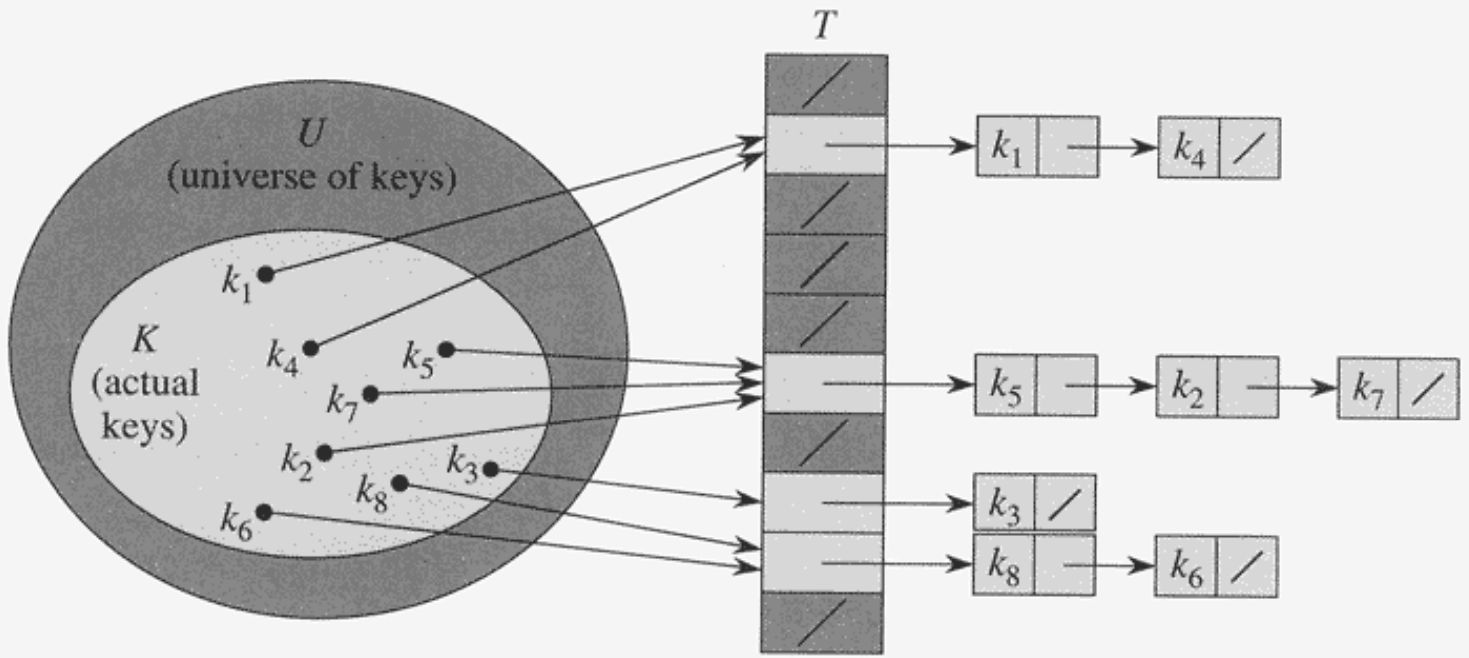


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Assuming that the keys are uniformly drawn, what is the expected cost of searching for a key?

Suppose there are n keys and the table size is m. Then, the cost averaged over all keys can be determined as follows:The expected length of each list is m/n. The expected cost of searching a list of length L is:

$m/2n$ (if the search is successful)

$m/n + 1$ (if the search is NOT successful).

Closed Hashing

The array itself is used to store the keys. Problem is collisions.

Collision resolution strategy: To store x, try the locations *h(x), h(x)+1, h(x)+2, …,* until an empty slot is found. (Should use wrap around.)

Apparently it is better to use a shift by some *d* > 1. i.e., use the probe sequence *h(x), (h(x)+d) mod N, (h(x)+2d) mod N …* Why is it better?

Avoids clustering.

Closed hashing - Insertion

```
HASH-INSERT(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = NIL
4             then T[j] ← k
5                  return j
6             else i ← i + 1
7     until i = m
8  error "hash table overflow"
```

Successive positions in which we attempt to insert k is called a probe sequence.

Even better (although the overhead is slightly greater) is to use double hashing. i.e., let *h'* be a second hash function. Use probe sequence *h(x), (h(x)+h'(x)) mod N, (h(x)+2h'(x)) mod N …* *etc.*

$$h(x, i) = h(x) + (i - 1)\, h'(x)$$

Closed hashing - Analysis

We will assume that for any input key x, the successive slots probed is a permutation of <0,1, …, m-1> so that every permutation is equally likely. *Clearly this assumption does not hold for any of the resolution strategies presented above.* But this assumption makes the analysis much easier.

Under this assumption, it is easy to estimate the cost of insertion (which is the same as the cost of unsuccessful search). Suppose there are currently n keys in the array.

(successful) insertion cost = cost of unsuccessful search

Cost of unsuccessful Search

The number of probes before an empty slot is found is geometrically distributed with probability of success = $n/m = 1 - \alpha$.

Theorem 11.6:
The expected number of probes is thus given by
$m/n = 1/(1-\alpha)$

Proof: a direct consequence of expected value of a geometric distribution.

Cost of successful search

Theorem

The expected number of probes in a successful search is at most $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$.

Proof A successful search for key k follows the same probe sequence as when key k was inserted.

By the previous corollary, if k was the $(i+1)$st key inserted, then α equaled i/m at the time. Thus, the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$.

That was assuming that k was the $(i+1)$st key inserted. We need to average over all n keys:

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n}) , \end{aligned}$$

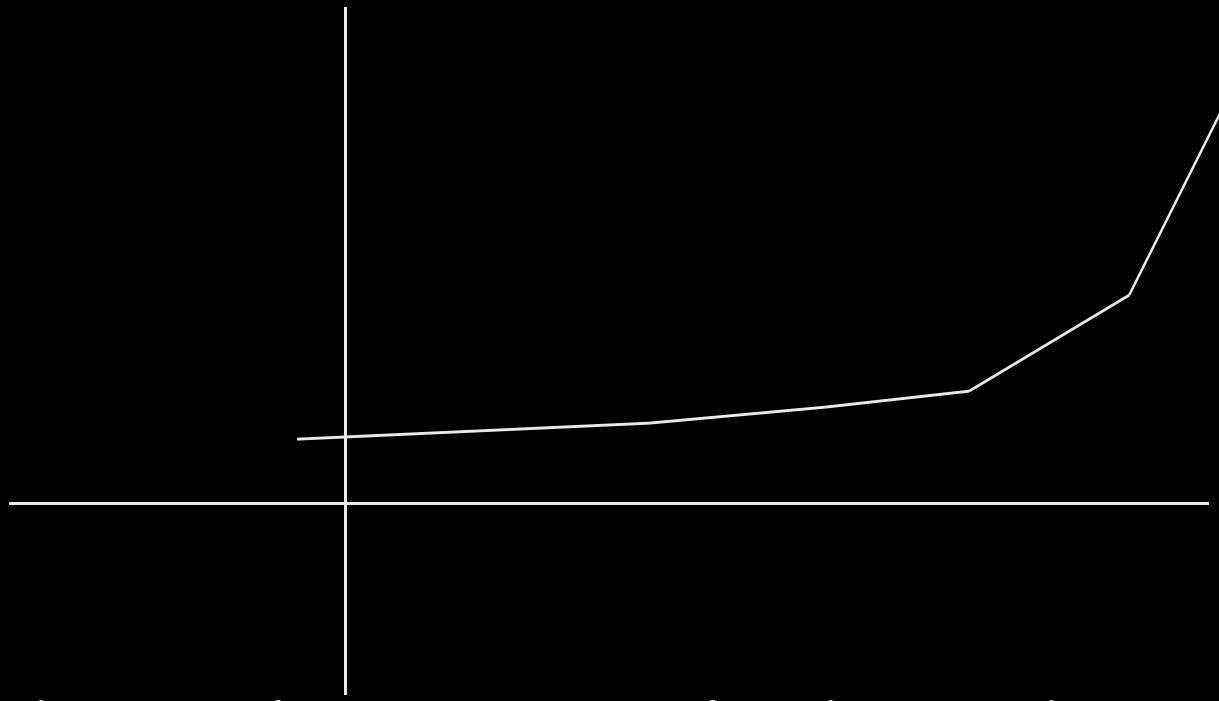
Cost of successful search

where $H_i = \sum_{j=1}^{i} 1/j$ is the ith harmonic number.

Simplify by using the technique of bounding a summation by an integral:

$$\begin{aligned}
\frac{1}{\alpha}(H_m - H_{m-n}) &= \frac{1}{\alpha} \sum_{k=m-n+1}^{m} 1/k \\
&\leq \frac{1}{\alpha} \int_{m-n}^{m} (1/x)\, dx \quad \text{(inequality (A.12))} \\
&= \frac{1}{\alpha} \ln \frac{m}{m-n} \\
&= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \qquad \blacksquare \text{ (theorem)}
\end{aligned}$$

How does this expression grow as a function of α?



So long α as is small (say below 0.5), this remains a constant. As grows large, say 0.8 or above, there is a rapid transition to a non-linear behavior. At this point, typically, one doubles the table size. Of course, this is a major task since all the keys should be reinserted into the new table.

Choice of a good hash function

- It should be simple to compute.

- It should distribute the keys uniformly. This requirement is hard to meet since it is application dependent.

- At the least, the hash function should depend on all the bits (digits) of the key. (For example, suppose we use the last 4 digits of the integer as the address. If your application generates only even numbers, half of the hash table will never be used.

- Another requirement: table size should be a prime number.

Two popular choices

Multiplication method: Let A be a real constant, $0 < A < 1$.

$$h(x) = [\, m\, (\, k\, A \bmod 1)]$$

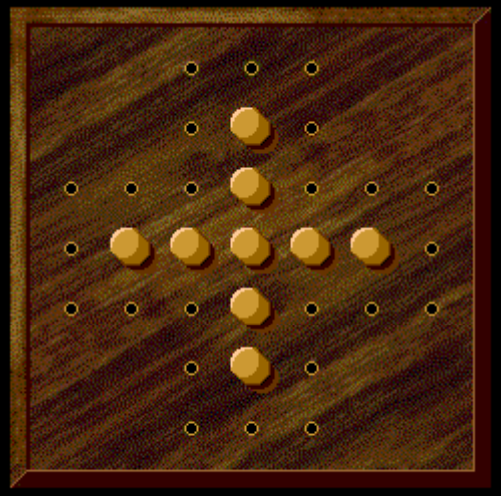
Division method: simpler

$$h(x) = x \bmod m$$

Both assume that the keys are integers. If the key is a string, we should convert them to integers.

Application of hashing to a search problem

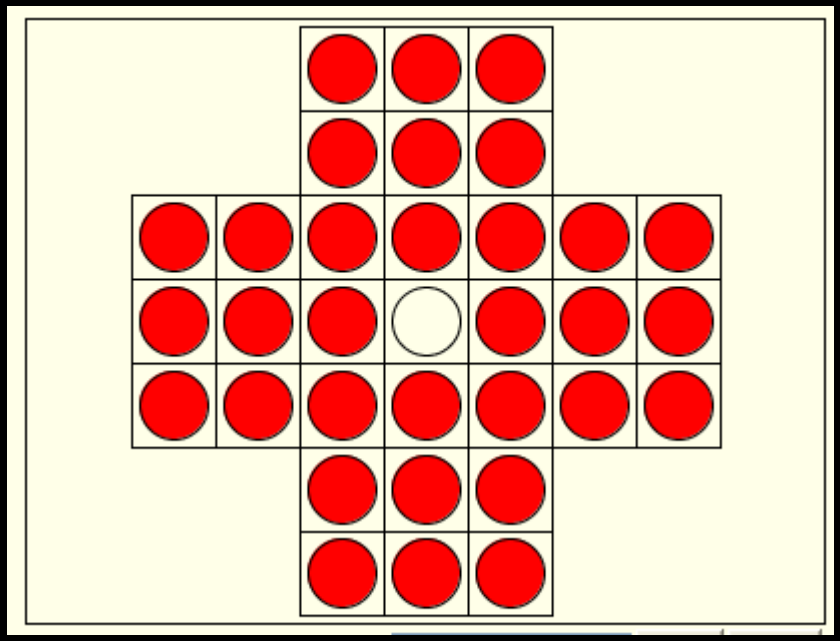
Peg Solitaire: A board with pegs as shown below:



Move: if there are two pegs on a line and an empty slot next, remove the two pegs and put one on the hole.

Goal is to find a sequence of moves that leaves exactly one peg on the board.

Standard starting position – how to win?



Applet for the game can be found in many sites, including the one in

http://www.cut-the-knot.org/proofs/pegsolitaire.shtml

Basic algorithm to solve the problem

```
Backtracking algorithm:

search (board B)
// returns a sequence of moves leading to solution, starting
// from B. A special message will be printed if no solution
Movelist = f ; // empty set
if (B is a final board position) then return Movelist;
L = list of successors of B;
while (L != f) do
  {
    Btemp = Delete(L);  // remove the first item from L
    M = the move that resulted in B1 (from B);
    call search(Btemp);
    if the output is a valid move sequence Mst, then
       {Movelist = M concatenated with Mst;
         return Movelist;
       }
   }
return “no solution”
end;
```

How can hashing improve the performance?

The idea is to save the board positions that have no solution in a hash-table. When exploring a node, first check if it is already in the table. If so, don't insert it.

Significantly improves the performance of the algorithm.

The idea can be applied to any backtrack searching algorithm.

We may try this for peg-solitaire or another similar problem.