Hash tables

- Many applications: dynamic set, that supports only operations INSERT, SEARCH, DELETE.
- E.g.: symbol table in compiler, keys=character strings that correspond to identifiers.
- Searching can take extremely long ($\theta(n)$) but in practice $O(1)$.
- Hash table: generalization of ordinary array.
- Direct addressing: examine an arbitrary array in $O(1)$ time.
- When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing
- a hash table typically uses an array of size proportional to the number of keys actually stored.
- Instead of using the key as an array index directly, the array index is computed from the key.

Direct addressing

- Simple technique that works well when the number of keys small.
- Dynamic set, each element has a key from universe $U = \{0, 1, \ldots, m-1\}$, where m is not too large, no two elements have the same key.
- use an array, or direct-address table, $T[0 \ldots m-1]$, in which each position, or slot, corresponds to a key in the universe U.
- `DIRECT-ADDRESS-SEARCH(T,k)`
  `return T[k]`
- `DIRECT-ADDRESS-INSERT(T,x)`
  `T[key[x]]= x`
- `DIRECT-ADDRESS-DELETE(T,x)`
  `T[key[x]]= NIL`
- For some applications, the elements in the dynamic set can be stored in the direct-address table itself. Rather than storing an element's key and satellite data in an object external to the direct-address table, with a pointer from a slot in the table to the object, we can store the object in the slot itself.
- Need a way to state that slot is empty.

Dynamic sets by direct address table

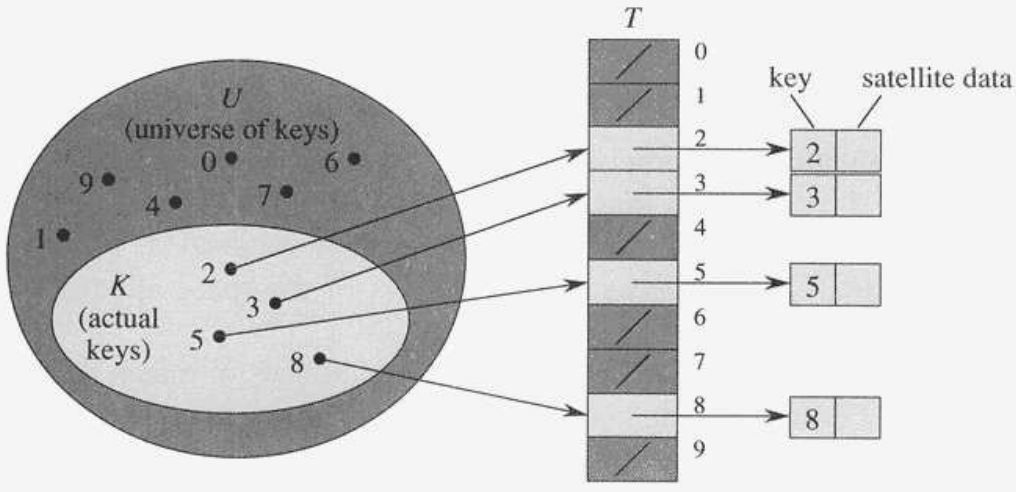


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded,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

Hash tables

- if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible.
- the set K of keys actually stored may be so small relative to U that most space allocated for T would be wasted.
- Hashing: provides $O(1)$ operations on the average.
- Storage $O(|K|)$.
- 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 used to compute the slot from the key.
- an element with key k hashes to slot $h(k)$; also say that $h(k)$ is the hash value of key k.
- collisions: two keys map to the same hash value.

Hash tables

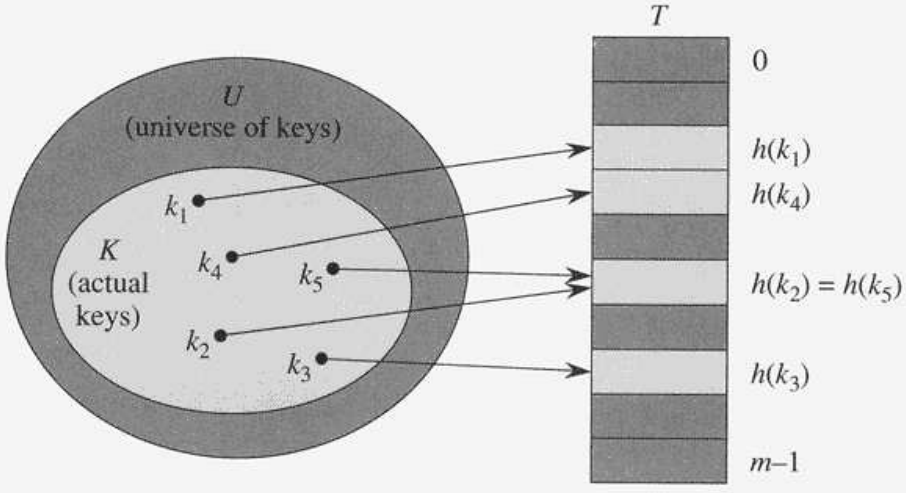


Figure 11.2 Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

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.
- `CHAINED-HASH-INSERT(T,x)`
  `insert` x `at the head of list` $T[h(key[x])]$
- `CHAINED-HASH-SEARCH(T,k)`
  `search for an element with key` k `in list` $T[h(k)]$
- `CHAINED-HASH-DELETE(T,x)`
  `delete` x `from the list` $T[h(key[x])]$
- The worst-case running time for insertion is O(1).
- For searching, the worst-case running time is proportional to the length of the list;
- Deletion of an element x can be accomplished in O(1) time if the lists are doubly linked.
- If the lists are singly linked, we must first find x in the list $T[h(key[x])]$, so that the next link of x's predecessor can be properly set to splice x out; in this case, deletion and searching have essentially the same running time.

Collision resolution by chaining

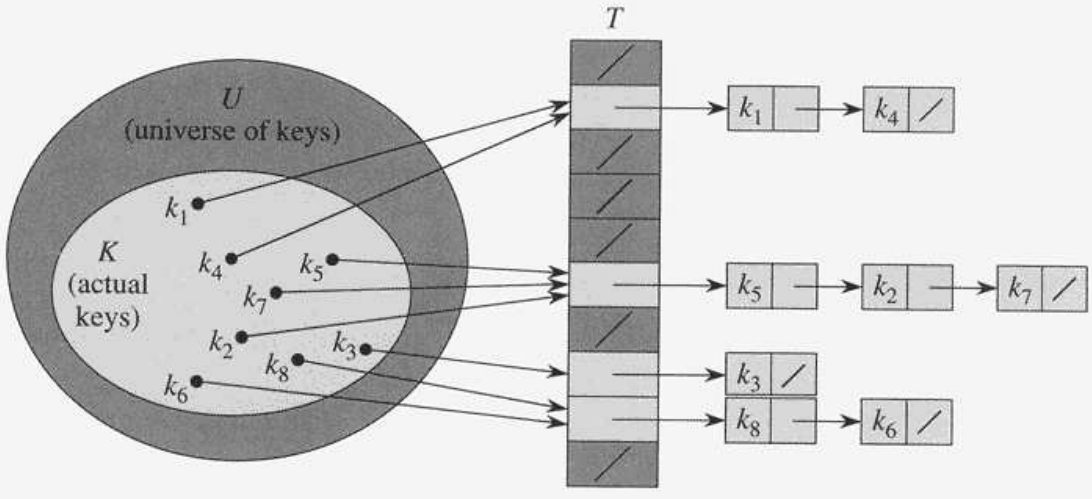


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Performance of collision resolution by chaining

- Given a hash table with n slots that stores m elements, define load factor $\alpha = m/n$.
- Average number of elements stored in a chain.
- Simple uniform hashing: every element is equally likely to hash into any of the m slots, irrespective of where previous elements have hashed.
- Theorem: under simple uniform hashing an unsuccessful search takes $O(1 + \alpha)$ time on the average.
- Theorem: under simple uniform hashing a successful search takes $O(1 + \alpha)$ time on the average.

Good hash functions

- A good hash function satisfies (approximately) the assumption of uniform hashing.
- Unfortunately, usually we don't know probability distribution of the keys, and keys might not be drawn independently.
- Good case: random real numbers k uniformly distributed in $[0, 1)$, $h(k) = \lceil km \rceil$ satisfies simple uniform hashing conditions.
- Most hash functions assume universe of keys are the natural numbers.
- E.g. character string = integer in base 128 notation.
- Identifier pt. ASCII $p = 112$, $t = 116$, becomes $112 \cdot 128 + 116 = 14452$.
- Division method: $h(k) = k$ mod m. Avoid some values of m,e.g. powers of two. Indeed, if $m = 2^p$ then $h(k) =$ the p lowest bits of k. Unless we know that p lowest bits of keys are uniform not a good idea.
- Prime not too close to an exact power of two = often a good choice.

Good hash functions: Multiplication method

- Multiplication method: Two stage procedure
- First, multiply key by constant A in range $0 < A < 1$, extract fractional part.
- Then multiply by m, extract floor.
- $h(k) = \lfloor m(kA \bmod 1) \rfloor$.
- Value of m not critical. $m = 2^p$.
- Easy implementation. Restrict $A = s/2^w$, w=machine word size.
- Better with some values of A than other. Knuth suggests $A \sim (\sqrt{5} - 1)/2$ will work well.

Multiplication-method of hashing

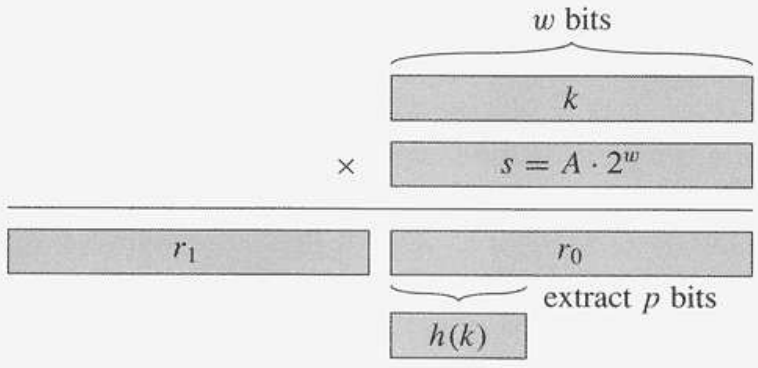


Figure 11.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

Open addressing

- All elements stored in the hash table itself.
- When searching for an element: systematically examine table slots until found, or report NOT FOUND.
- Load factor: cannot exceed 1.
- Instead of storing pointers we *compute* the sequence of slots to be examined.
- Insertion: probe the hash table until find an empty slot to put the key.
- Instead of being fixed in the order $0, 1, \ldots, m-1$, the sequence of positions probed depends on the key being inserted.
- To determine which slot to probe, extend hash function with probe number (starting from zero) as second input.
- $h : U \times \{0, \ldots, m-1\} \rightarrow \{0, \ldots, m-1\}$.
- Require that for every key k, probe sequence $h(k, 0), h(k, 1) \ldots h(k, m-1)$ is permutation of $\{0, \ldots, m-1\}$.

Procedure HASH-INSERT

```
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"
```

Procedure HASH-SEARCH

```
HASH-SEARCH(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = k
4             then return j
5          i ← i + 1
6     until T[j] = NIL or i = m
7  return NIL
```

Open-address hashing

- Deletion: difficult. Marking NIL does not work.
- Doing so might make it impossible to retrieve any key during whose insertion probed slot i and found it occupied.
- One solution: DELETED instead of NIL. Problem: search time no longer dependent on load factor.
- Techniques for probing: linear probing, quadratic probing and double hashing.
- Linear probing: given auxiliary hash function $h' : U \to \{0, \ldots, m-1\}$, use hash function

$$h(k, i) = (h'(k) + i) \text{ mod } m.$$

- Easy to implement but suffers from problem called primary clustering.
- Long runs of occupied slots build up, increasing average search time.

Open-address hashing

- Quadratic probing

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \text{ mod } m.$$

- Works much better than linear probing, but to make use of full hash table the values of c_1, c_2, m are constrained.
- Suffers from secondary clustering: if two keys have the same initial probe position then their probe sequences are the same.
- Double hashing

$$h(k,i) = (h_1(k) + i h_2(k)) \text{ mod } m.$$

- Among the best methods for open addressing.
- $h_2(k)$ must be relative prime to m. One solution is m a power of two and $h_2(k)$ odd.
- Another one: m prime, $h_2(k) < m$.
- Given an open address hash table with load factor $\alpha = n/m < 1$ the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Double hashing

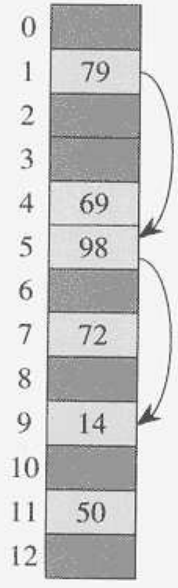


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Universal hashing

- Any *fixed* hash function vulnerable to worst case behavior:
- if "adversary" chooses n keys that all hash to the same slot, this yields average search time of $\theta(n)$.
- Solution: choose hash function randomly, independent of the keys that are going to be stored.
- $\mathcal{H}$ finite collection of hash functions that map universe U into $\{0, 1, \ldots, m-1\}$.
- Such a collection is called universal if for every keys $k \neq l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}/m|$.
- Suppose a hash function h is chosen from a universal collection of hash functions, and is used to hash n keys into a table T of size m (using chaining).
- If key is not in the table expected length of the list that k hashes to is at most α.
- If key is not in the table expected length of the list that k hashes to is at most $1 + \alpha$.

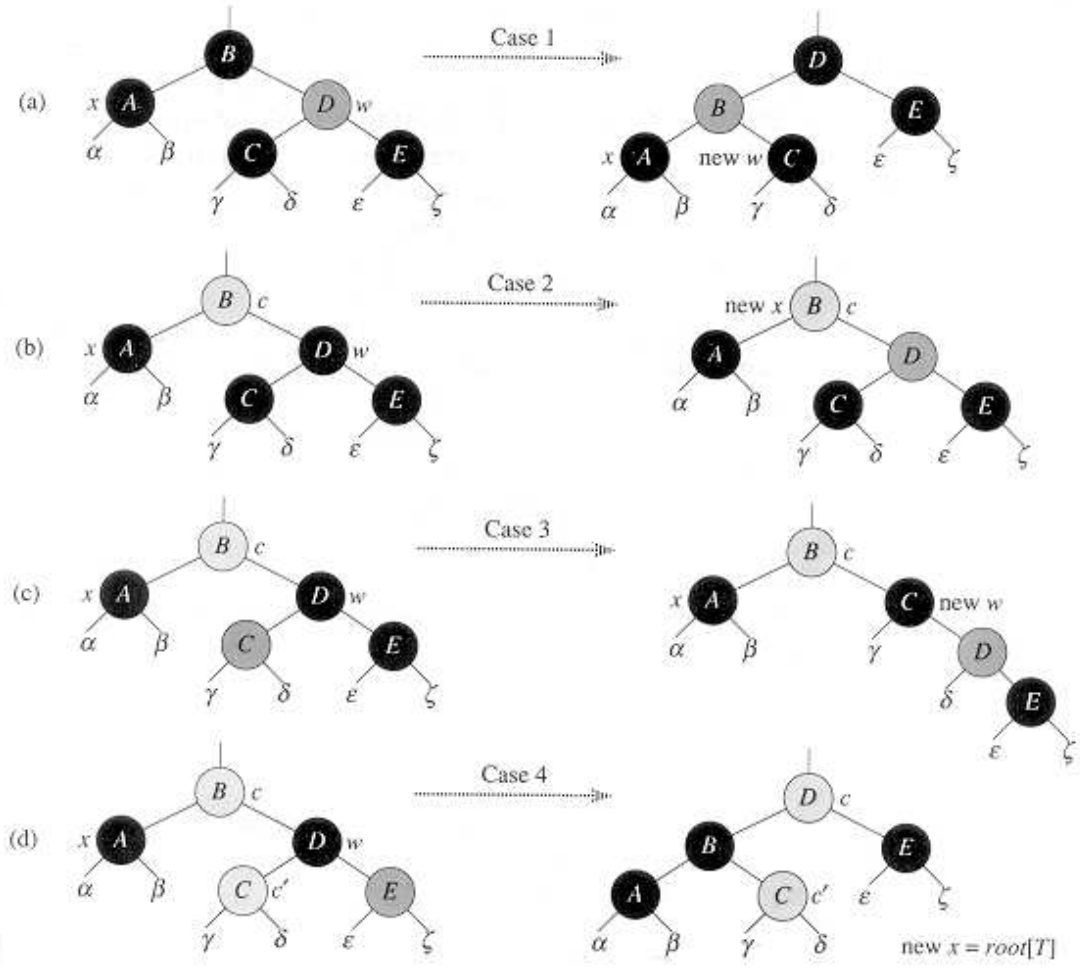
A universal class of hash functions

- Due to Carter and Wegman.
- $a \equiv b \pmod{p}$ if $p|(a-b)$.
- $\mathbf{Z}_p$: integers modulo p. p prime.
- How do we choose p ? So that all keys are in the range 0 to $p-1$.
- m: number of slots in the hash table.
- $a \in \mathbf{Z}_p^*$, $b \in \mathbf{Z}_p$.
- $h_{a,b}(k) = ((ak+b) \pmod{p}) \pmod{m}$.
- $\mathcal{H}_{p,m} = \{h_{a,b} : a \in \mathbf{Z}_p^*, b \in \mathbf{Z}_p\}$.
- Other applications of this set of hash functions: pseudo-random generators.

Perfect hashing

- Hashing can provide worst-case performance when the set of keys is static: once stored in the table, the set of keys never changes.
- Example: set of files on a CD-R (finished).
- Perfect hashing: the worst-case number of accesses to perform a search is O(1).
- Idea: two-level hashing with universal hashing at each level.
- First level: the n keys are hashed into m slots using a hashing function chosen from a family of universal hash functions.
- Instead of chaining: Use (small) secondary table S_j with an associated hash function h_j.
- Choose h_j carefully guarantees *no collisions*.

Perfect hashing with chaining



Perfect hashing: design

- $n_j =$ number of elements that hash to slot j.
- We let $m_j = |S_j| = n_j^2$.
- Idea: if $m = n^2$ and we store n keys in a table of size $m = n^2$ using a hash function randomly chosen from a set of universal hash function then the collision probability is at most 1/2.
- Find a good hash function using $O(1)$ trials.
- Expected amount of memory $O(n)$.
- Why this works: proof omitted (see Cormen if curious).
- Cryptographic hash functions: hash functions with good security properties.
- Most well-known cryptographic hash function: md5 (Rabin). You probably have encountered it if you downloaded anything large from the web.
- Hash tables in STL: not yet fully standard. Some implementations (e.g. SGI). Most functionality provided by associative container map (implemented using red-black trees).