Hashing and Hash Tables

Hash Tables

Many applications require a dynamic set that supports only the dictionary.
Operations**,** INSERT, SEARCH and DELETE. Example: a symbol table.

A **hash table** is effective for implementing a dictionary.

– The expected time to search for an element in a hash table is O (1), under some reasonable assumptions.
– Worst-case search time is $\Theta(n)$, however.

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 -- **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 -- **Hash function.**

Direct-Address Tables

• **Scenario:**
– Maintain a dynamic set.
– Each element has a key drawn from a universe U = {0, 1, ...,m-1} where m isn't too large.
– No two elements have the same key.

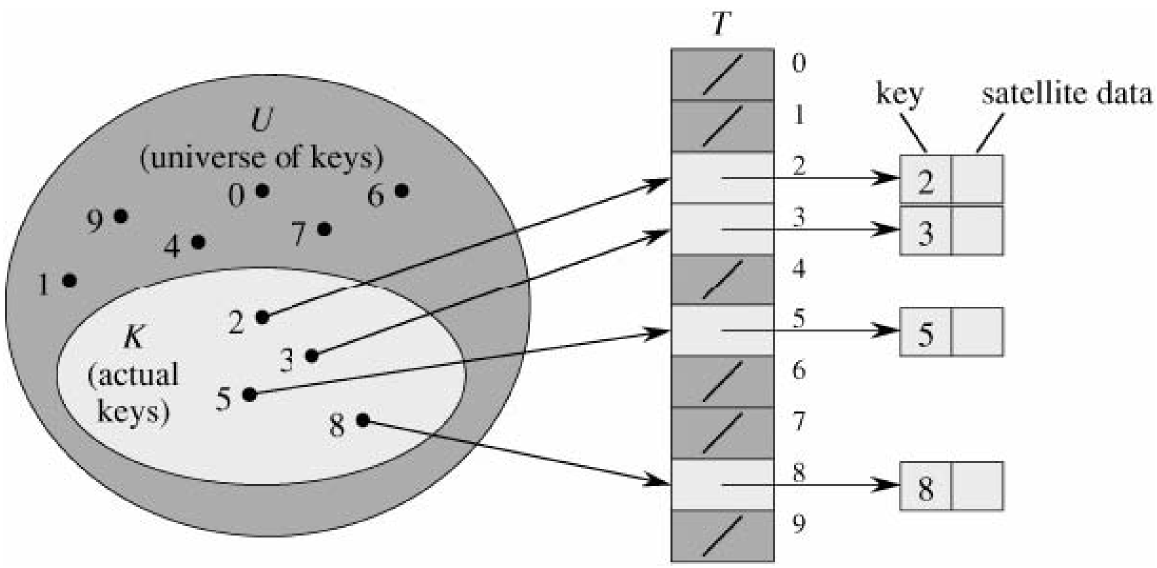
• Represent by a **direct-address table**, or array, T [0...m-1]:
– Each **slot**, or position, corresponds to a key in U.
– If there's an element x with key k, then T [k] contains a pointer to x.
– Otherwise, T [k] is empty, represented by NIL.

• Dictionary operations are trivial and take O(1) time each:

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



• The **problem** with direct addressing:

– if the universe U is large, storing a table of size |U| may be impractical or impossible.

• Often, the set K of keys actually stored is small, compared to U, so that most of the space allocated for T is wasted.

– When K << U, the space of a hash table << the space of a direct-address table.
– Can reduce storage requirements to (|K|).
– Can still get O(1) search time, but in the average case, not the worst case.

• **<u>Idea:</u>** Instead of storing an element with key k in slot k, use a function h and store the element in slot h(k).

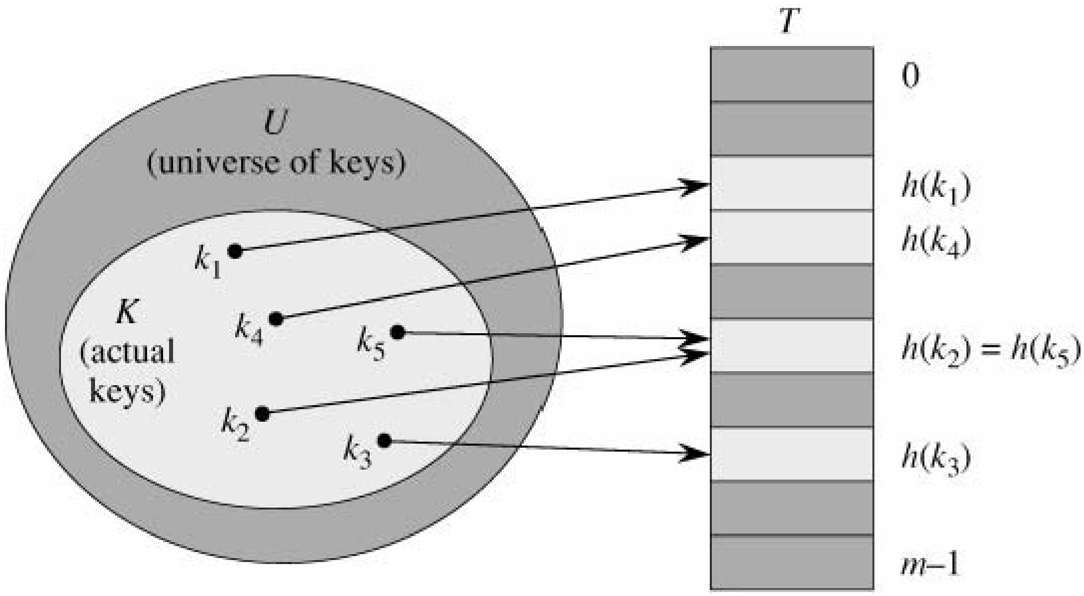
– We call h a **<u>hash function</u>**.
– h : U → {0, 1, . . . ,m-1}, so that h(k) is a legal slot number in T.
– We say that k **<u>hashes</u>** to slot h (k).

• **<u>Collisions:</u>** when two or more keys hash to the same slot.

– Can happen when there are more possible keys than slots (|U| > m).
– For a given set K of keys with |K| ≤ m, may or may not happen.

Definitely happens if |K| > m.

– Therefore, must be prepared to handle collisions in all cases.
– Use two methods: **chaining** and **open addressing.**
    • Chaining is usually better than open addressing.



Collision resolution by Chaining

Put all elements that hash to the same slot into a **linked list.**
Implementation of dictionary operations with chaining:

• **Insertion:** CHAINED-HASH-INSERT(T, x)

Insert x at the head of list T [h (key[x])]
– Worst-case running time is O(1).
– Assumes that the element being inserted isn't already in the list.
– It would take an additional search to check if it was already inserted.

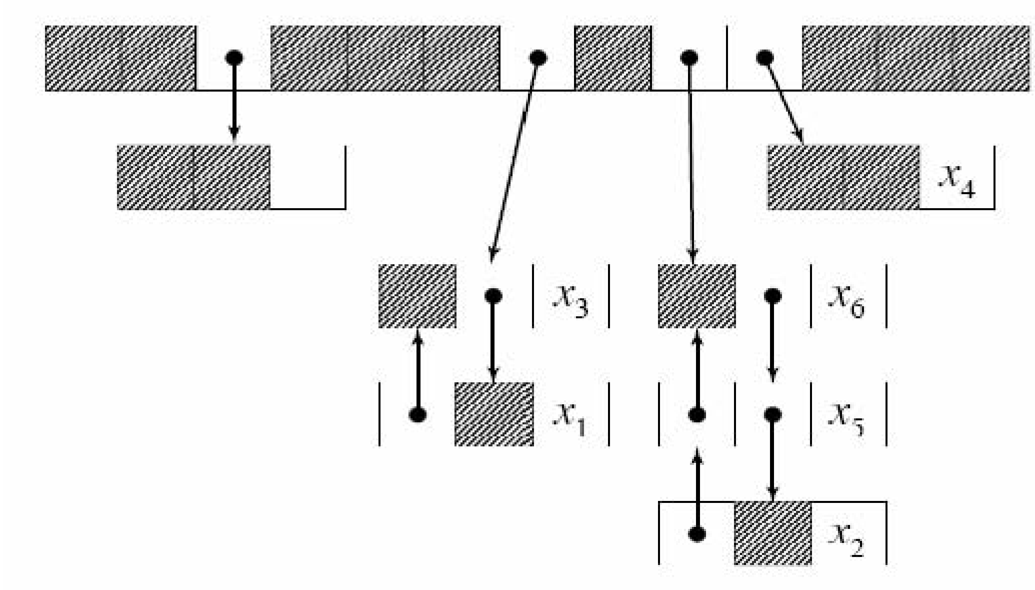
• **Search:** CHAINED-HASH-SEARCH(T, k)

Search for an element with key k in list T [h(k)]
– Running time is proportional to the length of the list of elements in slot h (k).

• **Deletion:** CHAINED-HASH-DELETE(T, x)

Delete x from the list T [h (key[x])]
– Given pointer x to the element to delete, so no search is needed to find this

element.
– Worst-case running time is O (1) time if the lists are doubly linked.
– If the lists are singly linked, then deletion takes as long as searching, because we must find x's predecessor in its list in order to correctly update next pointers.



Analysis of Hashing with Chaining

Given a key, how long does it take to find an element with that key, or to Determine that there is no element with that key?

- Analysis is in terms of the **load factor α = n/m:**
  - – n = # of elements in the table.
  - – m = # of slots in the table = # of linked lists.
  - – Load factor **α** is average number of elements per linked list.
  - – Can have α < 1, α = 1, or α > 1.
- Worst case is when all n keys hash to the same slot
  - --get a single list of length n
  - --worst-case time to search is Θ(n), plus time to compute hash function.
- Average case depends on how well the hash function distributes the keys among the slots.
- **Simple uniform hashing**:
  - Any data item is equally likely hash to any entry in hash table.
  - On average, each slot in table has same # of data.

Theorem:

Under chaining and simple uniform hashing, search takes Θ(1+ α) on average**.**

Why?

Failed search: must compute h and search to end of linked list, whose average size is **α**.
Successful search: Search for k takes # of elements inserted in linked-list after k.

Hash Functions

What makes a **good hash function**?

– the assumption of simple uniform hashing

(In practice, not possible to satisfy exactly)

– Often use heuristics, based on domain of values, to create a hash function that performs well.

Example of BAD hashing function :

h(k) = floor(K/ 100)

because

1) 0 …… 99 maps to slot 0.

2) 100 …. 199 maps to slot 1.

Example of GOOD hashing function :

h(k) = k mod m

where m = any prime number

- Keys as natural numbers
  - – Hash functions assume that the keys are natural numbers.
  - – When they're not, have to interpret them as natural numbers.
  - – Example:

    Interpret a character string as an integer expressed in some radix notation.

    Suppose the string is CLRS:

    - ASCII values: C = 67, L = 76, R = 82, S = 83.
    - There are 128 basic ASCII values.
    - So interpret CLRS as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) =$ 141,764,947.

- **Division method**
  - – h(k) = k mod m
  - – Advantage: Fast, since requires just one division operation.
  - – Disadvantage: Have to avoid certain values of m: ($\mathbf{2^p}$ bad)
  - – Example: m = 20 and k = 91 → h(k) = 11.
  - – Choose m as prime not too close to 2^p

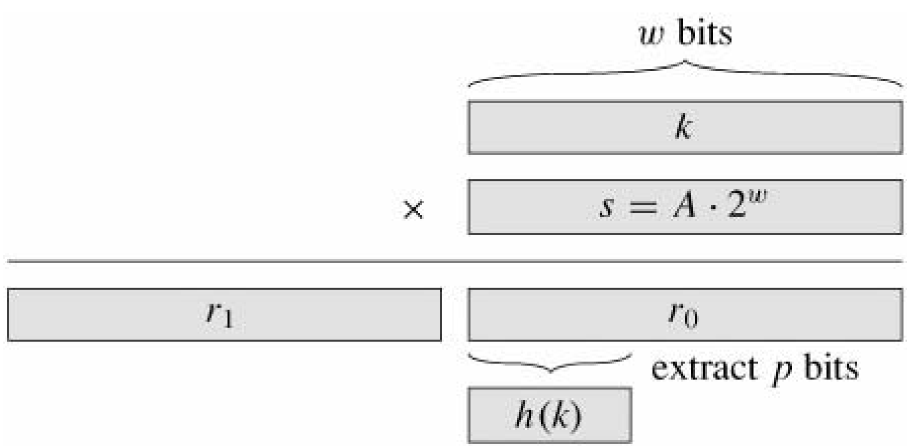
• **Multiplication Method:**

Disadvantage: Slower than division method.
Advantage: Value of m is not critical.

1. Choose constant A in the range $0 < A < 1$. (Typically, $A=s/2^w$, where w= word size and s is an integer)
2. Multiply key k by A.
3. Extract the fractional part of kA.
4. Multiply the fractional part by m (typically $m=2^p$)
5. Take the floor of the result.

Put another way, $h(k) = \lfloor m\,(kA \bmod 1)\rfloor$,
Where $kA \bmod 1 = kA - \lfloor kA \rfloor$ = fractional part of kA.

Implementation: first p bits of lowest w bits from *sk*



Example:
m = 8 (ie, p = 3), w = 5, k = 21.
Must have $0 < s < 2^5$; choose s = 13 → A = 13/32.
Compute h(k): kA = 21· 13/32 = 8+17/32
→ kA mod 1 = 17/32 □ m (kA mod 1) = 8 ·17/32 = 17/4
→ $\lfloor m\,(k A \bmod 1)\rfloor = 4$, so h(k) = 4.

Using implementation: k· s = 21 · 13 = 273 = 100010001.
Lowest w=5 bits of this: 17 → 10001
p = 3 most significant bits of 10001 → 100 (binary) → 4=h(k)

Other Methods

Folding
The key is divided into sections, and the sections are added (subtracted, multiplied) together. For example, if k=013402122, we could divide k into 3 sections: 013, 402, and 122, and then add them together to get 537.

Middle-Squaring
Take middle digits from key and square them. For example, if k=013402122, take 402 and square it resulting in 161604. If this value exceeds the table size M, one could use the middle four digits 6160.

Truncation
Simply delete part of the key and use the remaining digits. For example, if K=013402122, ignore all but the last 3 digits getting h(k)=122.

Open Addressing

Idea:

- Store all keys in the hash table T itself.
- Each slot contains either a key or NIL.
- To search for key k:
  - – Compute h(k) and examine slot h(k). Examining a slot is known as a **probe**.
  - – **T[h(k)]=k** : If slot h(k) contains key k (i.e.) , the search is successful.
  - – **T[h(k)]=nil** : If this slot contains NIL (i.e.) , the search is unsuccessful.
  - – **T[h(k)] $\neq$ k $\neq$nil :** There's a 3rd possibility: slot h(k) contains a key that is not k .
    - Probe new slot, choosing it based on k and on the number of probes so far
    - Keep probing until:
      - find key k (successful search)
      - Find NIL (unsuccessful search).

- Sequence of probes must be complete permutation of slots 0, . . , m -1
  - – can probe all slots
  - – no slot probed more than once on a given search
- Thus, the hash function is: **h(k, i)**
  - – **h** : (Key Universe) $\times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}$

  - – h(k, 0), h(k, 1), . . . , h(k,m-1) = permutation of 0, 1, . . . ,m -1.

- **Insertion**, act as though we're searching, and insert at the first NIL slot we find.

```
HASH-INSERT(T, k)
i = 0
repeat j = h(k, i)
    if T[j] = NIL
      then T[j] = k
           return j
      else i = i + 1
until i = m
error "hash table overflow"
```

```
HASH-SEARCH(T, k)
i =0
repeat j = h(k, i)
      if T[j] = k
            return j
      i = i + 1
 until T[j] = NIL or i = m
return NIL
```

- **Deletion:**

  – Cannot just put NIL into slot containing key we want to delete.

  Solution(?):

  – Use special value DELETED instead of NIL
  – Search should treat DELETED as though slot full
  – Insertion should treat DELETED as though slot empty
  – Disadvantage:
  search time no longer dependent just on load factor α

  $\rightarrow$**Chaining** more common when keys must be deleted.

Choosing probe sequences

- Ideally, want **uniform hashing** (generalizes simple uniform hashing)
  - each key equally likely to have any permutation of
    0, 1, . . . , m-1 as probe sequence
  - hard to implement, so we approximate it

Approx techniques produce at most m^2 probe sequences, not m! as desired.

- **Linear probing**
- **Quadratic probing**
- **Double hashing**

- **Linear probing**
  – Given key k and probe number i ($0 \le i < m$),
  h(k, i) = (h'(k) + i) mod m.
  – Iinitial probe determines the entire sequence
  $\rightarrow$ only m probing sequences.
  – **Disadavantage: primary clustering**
  – Long runs of occupied sequences build up.
  – Long runs tend to get longer
  since an empty slot preceded by i full slots gets filled
  next with probability (i + 1)/m.

Result is that the average search and insertion times increase.

- **Quadratic probing**

  $h(k, i) = (h'(k) + c1 \cdot i + c2 \cdot i^2) \bmod m$, where $c1, c2 \neq 0$ are constants.

  – Must constrain c1, c2, and m in order to ensure that we get a full permutation of 0, 1 ... m-1.

  – Disadvantage: **secondary clustering**

  – if two keys have same h', they have same probe sequence

  – Long runs get longer

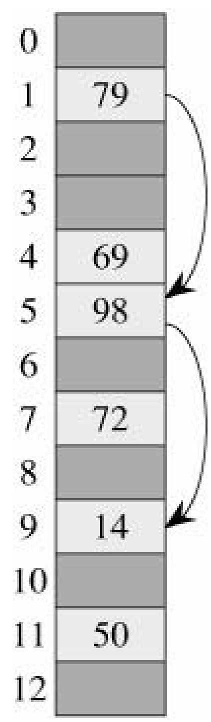
Double hashing:

- **$h(k, i) = (h1(k) + i \cdot h2(k)) \bmod m$.**
- **h2**(k) must be relatively prime to m (no common factors) to guarantee that probe sequence is full permutation
- Possibilities:

  $m = 2^p$ and $h2 > 1$ and odd

  m prime and $1 < h2(k) < m$.
- **$\Theta(m^2)$** different probe sequences,

  each h1(k), h2(k) combination gives different probe sequence.



<u>Theorem</u>

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in a **failed search** is at most $1/(1-\alpha)$, assuming uniform hashing. The same hold for the expected cost of insertion.

Theorem

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in a **successful search** is at most $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$, assuming uniform hashing.

Perfect Hashing

- Static keys
- memory access is **O(1).**

Summary

– Hash tables are the most efficient dictionaries if only operations Insert, Delete, and Find have to be supported.
– If uniform hashing is used, the expected time of each of these operations is constant.
– Universal hashing is somewhat complicated, but performs well even for adversarial input distributions.
– If the input distribution is known, heuristics perform well and are much simpler than universal hashing.
– For collision-resolution, chaining is the simplest method, but it requires more space than open addressing.
– Open addressing is either more complicated or suffers from clustering effects.