Algorithms and Data Structures

B4. Hash Tables

Gabriele Röger

University of Basel

April 11/17, 2024

Algorithms and Data Structures

April 11/17, 2024 — B4. Hash Tables

B4.1 Introduction

B4.2 Chaining

B4.3 Open Addressing

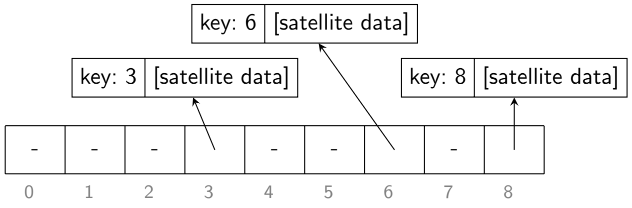
B4.4 Hash Functions

B4.5 Summary

B4.1 Introduction

Direct-address Table

- Assume you want to store elements that are associated with keys from a fixed universe $U = \{0, 1, \dots, k\}$.
- For every key, you need to store at most one element.
- Idea: Use array T (= direct access table), storing at position i a pointer to the element with key i.
- Inserting, removing and accessing the element for a key takes constant time.



Disadvantages of Direct-address Table

- If the universe is large or infinite, storing a table of size $|U|$ may be impractical or impossible.
- If the number of stored entries is small compared to the size of the universe, most space allocated for T would be wasted.

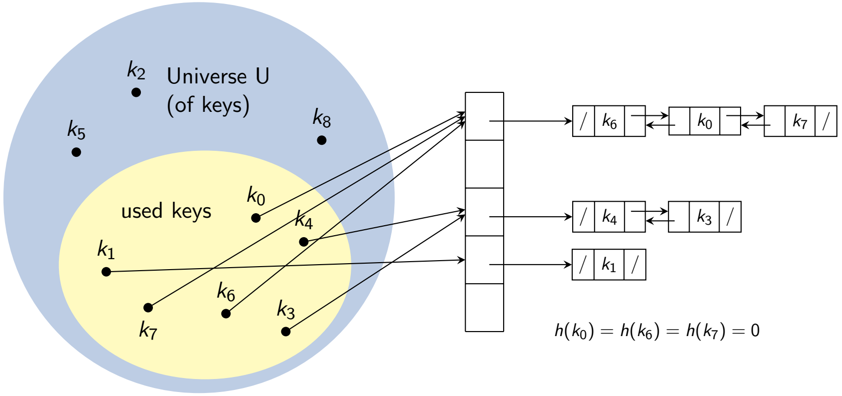
Hash Table

- Use a smaller array T (= the hash table) of size m, and
- a hash function $h : U \rightarrow \{0, \dots, m-1\}$, mapping the universe of keys into the possible positions in T.
  For example $h(k) = k \bmod m$
- We call $h(k)$ the hash value of key k.
- Problem: possible collisions
  - Different keys mapped to same hash value.
  - Unavoidable if $|U| > m$.
- Need collision resolution strategy. We will cover two methods:
  - Chaining
  - Open Addressing

B4.2 Chaining

Hashing with Chaining

Every non-empty hash-table position i points to a doubly linked list (the chain) of all the keys whose hash value is i:



Chaining: Implementation

- Search for an entry with key k
  - Search for entry with key k in list $T[h(k)]$.
- Remove entry with key k
  - Search for and remove element with key k from list $T[h(k)]$.
- Insert an entry e with key k
  - Search for entry with key k in list $T[h(k)]$.
  - If found: update linked list node to hold e.
  - If not found: prepend entry to list at $T[h(k)]$.

Chaining: Running Time I

- Assumption: Computing $h(k)$ takes constant time.
- The running time of all operations is dominated by the running time of the linked-list operations.
- All operations linear in the size of the involved linked list.
- Worst-case: All entries have the same hash value.
  $\rightsquigarrow$ worst-case running time linear in the number of entries

Independent Uniform Hashing

- "Ideal" hash function: for each key k, hash value $h(k)$ is randomly and independently chosen uniformly from the range $\{0, \dots, m-1\}$ (with m size of hash table).
- Subsequent calls of $h(k)$ for the same key k give the same output.
- Such a h is called a independent uniform hash function.
- Cannot reasonably be implemented in practise but useful for theoretical analysis.

Chaining: Running Time II

- Load factor α is defined as n/m, where
  - m is the number of positions (slots) in the hash table, and
  - n is the number of stored elements.
- α is the average number of entries in a chain.

Chaining: Running Time III

Theorem

In a hash table in which collisions are resolved by chaining, a search (successful or unsuccessful) takes $\Theta(1+\alpha)$ *time on average, under the assumption of independent uniform hashing.*

Consequence

If the number of elements n is at most proportional to the number of slots m ($n \leq cm$ for constant $c > 0$), then $\alpha \leq cm/m \in O(1)$.
→ average running time of insertion, deletion and search is $O(1)$.

Adapting the Size of the Hash Table

- To maintain an upper bound on the load factor (and thus constant average running times of operations), we may need to increase the size of the table.
- The change from the previous size m to size m' requires an adaptation of the hash function.
- In contrast to a size change of an array (where we just move every entry to the same index of the new memory range), we need to rehash all elements and insert them anew.

B4.3 Open Addressing

Open Addressing

- In contrast to chaining, with open addressing the entries are stored in the hash table itself.
- Hash table cannot hold more entries than size m (load factor cannot exceed 1).
- Size adaptation is analogous to chaining (need to rehash and reinsert all entries).
- To find a slot to insert an element, probe the hash table for the key until you find an empty slot:
  - If first choice for key occupied, try the second choice,
  - if second choice for key occupied, try the third choice,
  - ...
- To search for an element with key k, probe the table for the key until you find a slot that holds an element with key k.

Hash Functions for Open Addressing

- The hash function contains the probe number as a second input:

$$h : U \times \{0, \dots, m-1\} \to \{0, \dots, m-1\}$$

- Probe sequence for key k:
  $\langle h(k,0), h(k,1), h(k,2), \dots, h(k, m-1) \rangle$.
- For every key, the probe sequence must be a permutation of $\{0, \dots, m-1\}$:
  every position in the hash table included exactly once.

Open Addressing: Insertion and Search

Assumption: $key(e) = e$. Fix hash function `h`, hash table size `m`.

```
def hash_insert(T, k):
    for i in range(m): # i = 0, 1, ..., m-1
        pos = h(k, i)
        if T[pos] is None: # position empty
            T[pos] = k
            return pos
    raise Exception("hash table overflow")
```

```
def hash_search(T, k):
    for i in range(m):
        pos = h(k, i)
        if T[pos] == k:
            return pos
        if T[pos] is None:
            break
    return None # does not contain k
```

Open Addressing: Deletion?

- When deleting the element, we may not simply set the slot to `None` (Why?).
- Can mark the slot as `deleted`.
  - Insertion treats it like an empty slot.
  - Search treats it as an occupied slot.
- Disadvantage: Search times no longer depend on load factor but can take longer.
- If keys need to be deleted: consider chaining instead.
- Linear probing (a special case of open addressing) avoids need for `deleted` (later today).

Open Addressing: Running Time I

- Assumptions for running time analysis:
  - $\alpha < m$ (at least one slot empty)
  - no deletions
  - independent uniform permutation hashing:
    the probe sequence for a key is equally likely to be any permutation of $\{0, \dots, m-1\}$.
- Unsuccessful search: every probe but the last accesses an occupied slot (not containing the search key), last slot is empty.
- Successful search: some probe in the probe sequence accesses a slot with the searched key.

Open Addressing: Running Time II

Theorem

For a 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 independent uniform permutation hashing and no deletions.

Intuition:

$1/(1-\alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$

First probe always occurs, with probability α the probed slot is occupied, so a second probe occurs, ...

Corollary

Under the same assumption as in the theorem, inserting an element requires at most $1/(1-\alpha)$ probes on average.

Open Addressing: Running Time III

Theorem

For a open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most $\frac{1}{\alpha} \log_e \frac{1}{1-\alpha}$, assuming independent uniform permutation hashing with no deletions and assuming that each key in the table is equally likely to be searched for.

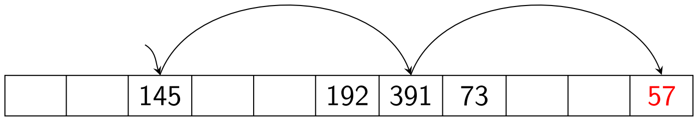
Double Hashing

- Double hashing uses two auxiliary hash functions $h_1 : U \to \{0, \dots, m-1\}$ and $h_2 : U \to \{0, \dots, m-1\}$.
- Hash function $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$
- Initial probe position $h_1(k)$ and step size $h_2(k)$ depend on k.
- $h_2(k)$ must be relatively prime to m (the only common divisor of $h_2(k)$ and m is 1).

  For example:
  - m power of 2 and $h(k)$ odd number, or
  - m prime and $h(h)$ positive integer less than m.

Double Hashing: Example

- $m = 11$, $h_1(k) = k \bmod 11$, $h_2(k) = 1 + k \bmod 9$
- Insert $k = 57$.
  - $57 \bmod 11 = 2$
  - $57 \bmod 9 = 3$



Special Case: Linear Probing

Use hash function $h_1 : U \to \{0, \dots, m-1\}$

- Probe sequence: $\langle h_1(k), h_1(k)+1, \dots, h_1(m-1), h_1(0), h_1(1), \dots, h_1(k)-1) \rangle$
- $h(k,i) = (h_1(k) + i) \bmod m$

Why is this a special case of double hashing?

Linear Probing: Deletion I

- Use function $g(k,q) = (q - h_1(k)) \bmod m$.
- If $h(k,i) = q$ then $g(k,q) = i$

Linear Probing: Deletion II

```python
def linear_probing_hash_delete(T, q): # delete entry at position q
    T[q] = None
    pos = q

    # search for a key that would have been inserted at position q
    # instead of its current position if q had been free.
    while True:
        pos = (pos + 1) % m # next slot in linear probing
        if T[pos] is None:
            # there is no key that would have been inserted at q.
            return
        key = T[pos] # this could be such a key
        if g(key,q) < g(key,pos):
            # indeed, this key should be moved to q.
            break
        # otherwise continue with next position

    T[q] = key # move key into slot p
    linear_probing_hash_delete(T, pos) # now pos needs to be emptied
```

Linear Probing: (Dis-)Advantage

Disadvantage: Primary clustering

- An empty slot occurring after i full slots gets filled next with probability $(i+1)/m$.
- Linear probing has a tendency to build up long runs of occupied slots (so-called clusters).
- Running time of search depends on size of clusters.

Advantage: Data locality

- Memory accessed by modern CPUs has a number of levels (registers, cache, main memory, ...).
- For example, the cache always fetches entire cache blocks from the main memory.
- Linear probing mostly "reuses" the same fetched block, avoiding frequent (slow) access to the main memory.

B4.4 Hash Functions

Static Hashing: Division and Multiplication Method

For the moment, we consider keys that are non-negative integers that fit in a machine word (32 or 64 bits).

Static hashing uses a single, fixed hash function.

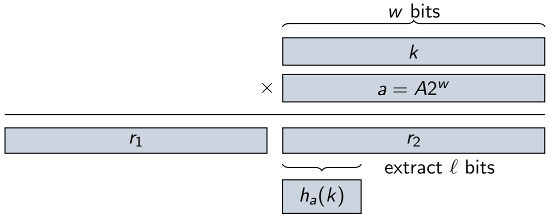
Examples (m = hash table size):

- Division method: $h(k) = k \bmod m$
  - Works well when m is a prime not too close to a power of 2.
- Multiplication method: pick some A with $0 < A < 1$. Then

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor.$$

  - $kA - \lfloor kA \rfloor$: fractional part of kA.
  - Works best if $m - 2^\ell$, where $\ell \leq w$, where w is the number of bits in a machine word.

Static Hashing: Multiply-shift Method



- $m = 2^\ell$ for integer $\ell < w$, where w is the number of bits in a machine word.
- For $0 < A < 1$, the result of $k \cdot A2^2$ is an integer with $\leq 2w$ bits (= 2 words).
- Use ℓ most significant bits of the low-order word of the product as hash value.
- Fast but no formal guarantees.

Random Hashing

- For every static hash function, an adversary can choose a sequence of keys that are all hashed to the same slot.
- Random hashing chooses the hash function randomly and independently of the keys that are going to be stored
- The special case of universal hashing guarantees good average performance, independent of the sequence of keys.

Random Hashing: Universal Hashing

- A family $\mathcal{H}$ of hash functions mapping universe U into slots $\{0, \dots, m-1\}$ is universal if for each pair of distinct keys $k, k' \in U$ there are at most $|\mathcal{H}|/m$ hash functions $h \in \mathcal{H}$ such that $h(k) = h(k')$.
- Universal hashing can be achieved in practise (e.g. using multiply-shift).
- With universal hashing and chaining, any sequence of s `insert`, `delete` and `search` operations takes $\Theta(s)$ expected time, if it starts from an empty hash table with m slots and includes at most $O(m)$ insert operations

Cryptographic Hashing

- Cryptographic hash functions are complex pseudorandom functions, designed for applications requiring properties beyond those needed here.
- Some CPUs contain specific instructions to support a fast computation of some cryptographic functions.
- A cryptographic hash function takes as input an arbitrary byte string and returns a fixed-length output.
  - E.g. SHA-256 produces a 256-bit (32-byte) output for any input.
  - We can use $h(k) = \text{SHA-256}(k) \bmod m$, or
  - create a family of such hash functions by prepending different “salt” strings a to k.

B4.5 Summary

Summary

- Hash functions map the keys of the universe to the m possible slots of the hash table.
- Since there typically are more possible keys than slots, collisions are unavoidable.
- We deal with them by chaining and open addressing (e.g. using linear probing).
- Designing good hash functions is non-trivial and often uses a random selection from a family of functions.
- With a good hash function and load factor management, insertion and (successful) search is possible in constant amortized time on average (logarithmic in the worst case).