16.070
Introduction to Computers & Programming

Hashing: breaking the log n barrier

Prof. Kristina Lundqvist
Dept. of Aero/Astro, MIT

Hashing and Hash Tables

- Represent a table of names
  - Set aside an array big enough to contain one element for each possible string of letters
  - Convert from names to integers
  - Tells where person’s phone number is immediately

- Dictionary operations
  - Insert / delete /search

<table>
<thead>
<tr>
<th>Dictionary operations</th>
<th>(check, a restraint)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct-access table</td>
<td>(check, examination)</td>
</tr>
<tr>
<td>Hash table</td>
<td>(check, a bill)</td>
</tr>
<tr>
<td>Hash function</td>
<td>(check, a pattern)</td>
</tr>
<tr>
<td>Collision resolution</td>
<td>(check, a small crack)</td>
</tr>
<tr>
<td>Chaining</td>
<td>(check, move in chess)</td>
</tr>
</tbody>
</table>

“hashing”

O(n)

O(lg n)

O(1)
Direct Addressing

- **Direct addressing** is a simple technique that works well when the universe $U$ of keys is reasonably small
- Assume we have:
  - Application needs a dynamic set
  - All elements of dynamic set have keys, from Universe $U = \{0, 1, \ldots, m-1\}$ of keys, associated with them
  - $m$ is not too large
  - No two elements have the same key
- Direct-address tables
  - Implement a dynamic set as an array (direct-address table), $T[0..m-1]$
  - Each slot corresponds to a key in $U$
  - Slot $k$ points to an element in dynamic set with key $k$
  - If dynamic set contains no element with key $k$ then $T[k] = \text{NIL}$

Dictionary operations

- **Insert**
  - `direct_access_insert (T, x)`
  - $T[\text{key}[x]] := x \quad \mathcal{O}(1)$
- **Delete**
  - `direct_access_delete (T, x)`
  - $T[\text{key}[x]] := \text{NIL} \quad \mathcal{O}(1)$
- **Search**
  - `direct_access_search (T, k)`
  - \text{return} $T[k] \quad \mathcal{O}(1)$

The problem with **direct-addressing** is:

- If universe $U$ is large, storing a table of size $|U|$ is impractical
- If the set of actually stored keys $k$ is small relative to $U$, then most of the space allocated for $T$ is wasted

The advantages of **hash table** is:

- When set $k$ of keys stored in dictionary is much smaller than the universe $U$ of all keys, a hash table requires much less space than a direct-address table
- Storage requirements are reduced to $\Theta(|k|)$ instead of $\Theta(|U|)$
The differences are:

- Searching for an element using hashing requires $\Theta(1)$ on average
- Searching for an element using direct-addressing requires $\Theta(1)$ in the worst-case
- Direct-addressing stores an element with key $k$ in slot (also called a bucket) $k$
- Hashing stores an element in slot $h(k)$, where $h(k)$ is a hash function $h$ used to compute the slot from the key $k$

Some definitions

- **Hash function** $h$: is used to compute the slot in the hash table from the key $k$
- **Hash table** $T$: where hash function $h$ maps the universe $U$ of all possible keys into slots $T[0 .. m-1]$
  
  $$h: U \rightarrow \{0, 1, .., m-1\}$$

- **Hashes** means mapping key $k$ to slot $h(k)$
- **Hash value** is the $h(k)$ of key $k$
- **Collisions** are when two keys hash to the same slot
- **Chaining** is putting all elements that hash to the same slot into a linked list or double linked list for $O(1)$ time deletion

Desired properties of a Hash Function

- An ideal hash function should avoid collisions entirely
  
  - The “birthday paradox” makes this improbable
    - What is the probability that at least 2 people in a room of 23 will have the same birthday?

- A hash function must be deterministic, in that a given input $k$ should always produce the same $h(k)$ output

- Since $|U| > m$, there must be 2 keys that have the same hash value
  
  - A well designed random output hash function may minimize collisions, but we need a mechanism for handling collisions
**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 no element hashes to \( j \), then \( j \) contains NIL.

---

**Dictionary operations**

- **Insert**
  - `chained_hash_insert(T, x)`
    - insert \( x \) at head of list \( T[h(key[x])] \)
    - worst-case runtime \( \mathcal{O}(1) \)

- **Delete**
  - `chained_hash_delete(T, x)`
    - delete \( x \) from list \( T[h(key[x])] \)
    - worst-case runtime \( \mathcal{O}(1) \) if lists are doubly-linked

- **Search**
  - `chained_hash_search(T, k)`
    - search for element with key \( k \) in list \( T[h(k)] \)
    - worst-case runtime \( \mathcal{O}(1) \)
      - If the number of hash table slots \( n \) is at least proportional to the number of elements in the table \( m \) or \( n = \mathcal{O}(m) \)
      - So that \( \alpha = n/m = \mathcal{O}(m)/m = \mathcal{O}(1) \)

---

**Analysis of hashing with chaining**

- **Some definitions:**
  - **Load factor** \( \alpha \): is the ratio of the number of stored elements \( n \) divided by the number of slots \( m \) in hash table \( T \) or \( \alpha = n/m \)

- **Simple uniform hashing**: is when any given element is equally likely to hash into any of the \( m \) slots, independently of where any other element has hashed to.
**Analysis of hashing with chaining**

- **Worst-case behaviour**:  
  - All $n$ keys hash to the same slot, this creates a list of length $n$  
  - The worst-case time is therefore (terrible) $\Theta(n)$  
  - Which is no better than if using one linked list for all elements, plus the time it takes to compute the hash function  
  - Hash tables are **not** used for their worst-case performance

- **Theorem**: In a hash table in which collisions are resolved by chaining, an unsuccessful search takes expected time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

- **Proof**: Under the assumption of simple uniform hashing, any key $k$ not already stored in the table is equally likely to hash to any of the $m$ slots. The expected time to search unsuccessfully for a key $k$ is the expected time to search to the end of list $T[h(k)]$, which has expected length $= \alpha$. Thus, the expected number of elements examined in an unsuccessful search is $\alpha$, and the total time required (including the time for computing $h(k)$) is $\Theta(1+\alpha)$.

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

- **Proof**: If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n=O(m)$ and, consequently, $\alpha=n/m=O(m)/m=O(1)$. Thus, searching takes constant time on average. Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked, all dictionary operations can be supported in $O(1)$ time on average.
Hash functions

- The best possible hash function would hash $n$ keys into $m$ “buckets” with no more than $\lfloor n/m \rfloor$ keys per bucket. Such a function is called a perfect hash function.
- What is the big picture?
  - A hash function which maps an arbitrary key to an integer turns searching into array access, hence $O(1)$
  - To use a finite sized array means two different keys will be mapped to the same place. Thus we must have some way to handle collisions.
  - A good hash function must spread the keys uniformly, or else we have a linear search.

Example

- Suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n=2000$ character strings, where a character has 8 bits.
- We don’t mind examining an average of 3 elements in an unsuccessful search, so we allocate a hash table of size $m=701$.
- The number 701 is chosen because it is a prime near $2000/3$ but not near any power of 2.
- Treating each key $k$ as an integer, our hash function would be:
  \[ h(k) = k \mod 701 \]

Hash functions: The Division Method

- Map key $k$ into one of $m$ slots by taking the remainder of $k$ divided by $m$.
  - We use the hash function
    \[ h(k) = k \mod m \]
    - We avoid certain values of $m$, such as $m=2^p$ for binary $k$ and $m=10^p$ for decimal $k$
    - We chose $m$ as primes not close to $2^p$.

Hash functions: The Multiplication Method

- Operates in two steps:
  - Multiply the key $k$ by a constant $A$ in the range $0 < A < 1$, and extract the fractional part of $kA$.
  - Multiply this value by $m$ and take the floor of the result.
  - Resulting hash function is:
    \[ h(k) = \lfloor m(kA \mod 1) \rfloor \]
    where $kA \mod 1$ returns the fractional part of $kA$, the same as $kA - \lfloor kA \rfloor$.
  - Advantage of the multiplication method is that the value of $m$ is not critical. Typically chose it to be a power of 2.
Hash functions: The Multiplication Method

- Suppose that the word size of the machine is \( w \) bits and that \( k \) fits into a single word. We restrict \( A \) to be a fraction of the form \( s/2^w \), where \( s \) is an integer in the range \( 0 < s < 2^w \).

- First multiply \( k \) by the \( w \)-bit integer \( s = A2^w \). The result is a \( 2w \)-bit value \( r_12^w + r_0 \), where \( r_1 \) is the high-order word of the product and \( r_0 \) is the low-order word of the product. The desired \( p \)-bit hash values consist of the \( p \) most significant bits of \( r_0 \).

\[ \begin{array}{c|c}
\text{w bits} & k \\
\hline
x & s = A2^w \\
\hline
r_1 & r_0 \\
\hline
\end{array} \]

\( h(k) \) — Extract \( p \) bits

Example

- Suppose we have \( k = 123456, p = 14, m = 2^{14} = 16384 \), and \( w = 32 \).

- Choose \( A \) to be the fraction of the form \( s/2^{32} \) that is closest to \((\sqrt{5} - 1)/2\) so that \( A = 2654435769/2^{32} \).

- Then \( ks = 32770602297664 = (76300*232) + 17612864 \), and so \( r_1 = 76300 \) and \( r_0 = 17612864 \).

- The 14 most significant bits of \( r_0 \) yields the value \( h(k) = 67 \).

Universal hashing

- The worst case scenario is when \( n \) keys all hash to the same slot. This requires a \( \Theta(n) \) retrieval time. Any fixed hash function is vulnerable to the possibility of the worst case. The only effective counter measure is to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored. This method, known as universal hashing yields good performance on average.

\[ \text{Let } H \text{ be a finite collection of hash functions so that} \]
\[ \text{For every } h \in H, \text{ we have } h: U \rightarrow \{0, 1, \ldots, m-1\} \]

- This collection \( H \) is universal

- If for each pair of distinct keys \( x, y \in U \), the number of hash functions \( h \in H \) where \( h(x) = h(y) \) is \( |H|/m \)

- We interpret this to mean that:

  - Given hash function \( h \in H \) chosen randomly
  - The probability of a collision between \( x \) and \( y \) when \( x \neq y \) is \( 1/m \)
  - This is exactly the probability of a collision of \( h(x) \) and \( h(y) \) are randomly chosen from \( \{0, 1, \ldots, m-1\} \)
Collision resolution

- Two approaches
  - Separate chaining
    - m much smaller than n
    - ~n/m keys per table position
    - Put keys that collide in a list
    - Need to search lists
  - Open addressing (linear probing, double hashing)
    - m much larger than n
    - Plenty of empty table slots
    - When a new key collides, find an empty slot
    - Complex collision patterns

Open Addressing

- To perform insertion using open addressing we **probe** the hash table to find an empty slot in which to put the key. Instead of being fixed in the order 0, 1, ..., m-1 (requiring Θ(n) time), the sequence of positions is probed depending upon the key being inserted.

Open Addressing

### Advantages:
- Do not use pointers, which speed up addressing schemes, frees up space
  - Faster retrieval times
  - Reduces the number of collisions
- May store a larger table with more slots for the same memory
- Compute the sequence of slots to be examined

Open Addressing

- Extend the hash function to also include the probe number (starting from 0) as a second input.
  - h: U * {0, 1, ..., m-1} → {0, 1, ..., m-1}
- For open addressing, we require that for every key k, the **probe sequence**
  
  \[ <h(k, 0), h(k, 1), ..., h(k, m-1)> \]

  be a permutation of \(<0, 1, ..., m-1>\), so that every hash-table position is eventually considered as a slot for a new key as the table fills up.
### Pseudo code: insert

- Assume that the elements in the hash table T are keys with no satellite information; the key $k$ is identical to the element containing key $k$. Each slot contains either a key or NIL (if slot is empty).

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

### search

- The algorithm for searching for key $k$ probes the same sequence of slots that the insertion algorithm examined when key $k$ was inserted. Therefore, the search can terminate (unsuccesfully) when it finds an empty slot, since $k$ would have been inserted there and not later in its probe sequence. (this argument assumes that keys are not deleted from the hash table.) The procedure hash_search takes as input a hash table T and a key $k$, returning $j$ if slot $j$ is found to contain key $k$, or NIL of key $k$ is not present in table T.

```
hash_search(T, k)
i := 0
repeat j := h(k, i)
    if T[j] = k
        then return j
    else i := i+1
until T[j] = NIL or i=m
return NIL
```

### Pseudo code: search

### deletion

- **Deletion** from an open-address hash table is **difficult**. When we delete a key from slot $i$, we cannot simply mark that slot as empty by storing NIL in it. Doing so might make it impossible to retrieve any key $k$ during whose insertion we had probed slot $i$ and found it occupied.

- Solution: mark the slot by storing in it the special value DELETED instead of NIL. 
  → modify the procedure hash_insert to treat such a slot as empty so that a new key can be inserted. 
  No modification of hash_search is needed, since it will pass over DELETED values while searching.

- When special value is used, search times no longer dependent on the load factor $\alpha$, and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.
Open Addressing

- Assume: *uniform hashing* instead of *simple uniform hashing*.
  - The hash function in uniform hashing produces a hash sequence
  - Each key is equally likely to have any of m! permutations of \{0, 1, ..., m-1\} as its probe sequence
  - Deletion is difficult and a modification to hash_search is necessary to continue to search if a slot is marked deleted instead of NIL.
  - Chaining may be needed

Four techniques for computing probe sequences for open addressing

1. Sequential probing: \(h, h+1, h+2, h+3, \ldots\)
2. Linear probing: \(h, h+k, h+2k, h+3k, \ldots\)
3. Quadratic probing: \(h, h+1^2, h+2^2, h+3^2, \ldots\)
4. Double hashing: \(h(k,i) = (h_1(k) + ih_2(k)) \mod m\), where \(h_1\) and \(h_2\) are auxiliary hash functions.
   - All generate \(\langle h(k,0), h(k,1), \ldots, h(k, m-1)\rangle\) as a permutation of \(\{0, 1, \ldots, m-1\}\)
   - None can generate more than m2 different probe sequences as uniform hashing requires m! different probe sequences (permutations)
   - Double hashing has the greatest number and may give the best results

Analysis of Open Addressing

- **Theorem**: 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
  
  \[
  \frac{1}{1-\alpha}
  \]
  assuming uniform hashing

- **Corollary**: Inserting an element into an open-address hash table with load factor \(\alpha\) requires at most
  
  \[
  \frac{1}{1-\alpha}
  \]
  probes on average, assuming uniform hashing

- **Theorem**: Given an open-address hash table with load factor \(\alpha < 1\), the expected number of probes in a successful search is at most
  
  \[
  \frac{1}{\alpha} \ln \frac{\frac{1}{1-\alpha}}{\frac{1}{\alpha}}
  \]
  assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

- If the hash table is half full, then the expected number of probes is less than 3.38629. If it is ninety percent full, we have less than 3.66954 probes.