Hash Tables


The Search Problem

- Find items with keys matching a given search key
  - Given an array A, containing n keys, and a search key x, find the index i such as x=A[i]
  - As in the case of sorting, a key could be part of a large record.

example of a record

<table>
<thead>
<tr>
<th>Key</th>
<th>other data</th>
</tr>
</thead>
</table>

Special Case: Dictionaries

- **Dictionary** = data structure that supports mainly two basic operations: **insert** a new item and **return an item with a given key**.
  - Queries: return information about the set $S$ with key $k$:
    - get ($S$, $k$)
  - Modifying operations: change the set
    - put ($S$, $k$): insert new or update the item of key $k$.
    - remove ($S$, $k$) – not very often

Direct Addressing

- **Assumptions:**
  - Key values are distinct
  - Each key is drawn from a universe $U = \{0, 1, \ldots, N - 1\}$
- **Idea:**
  - Store the items in an array, indexed by keys

- **Direct-address table** representation:
  - An array $T[0 \ldots N - 1]$
  - Each slot, or position, in $T$ corresponds to a key in $U$
  - For an element $x$ with key $k$, a pointer to $x$ (or $x$ itself) will be placed in location $T[k]$
  - If there are no elements with key $k$ in the set, $T[k]$ is empty, represented by NIL
Direct Addressing (cont’d)

(insert/delete in $O(1)$ time)

Comparing Different Implementations

- Implementing dictionaries using:
  - Direct addressing
  - Ordered/unordered arrays
  - Ordered linked lists
  - Balanced search trees

<table>
<thead>
<tr>
<th></th>
<th>put</th>
<th>get</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct addressing</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>ordered array</td>
<td>$O(N)$</td>
<td>$O(lgN)$</td>
</tr>
<tr>
<td>unordered array</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>ordered list</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>balance search tree</td>
<td>$O(lgN)$</td>
<td>$O(lgN)$</td>
</tr>
</tbody>
</table>
Hash Tables

- When \( n \) is much smaller than \( \max(U) \), where \( U \) is the set of all keys, a **hash table** requires much less space than a **direct-address table**
  - Can reduce storage requirements to \( O(n) \)
  - Can still get \( O(1) \) search time, but on the average case, not the worst case

---

Hash Tables

- Use a function \( h \) to compute the slot for each key
- Store the element in slot \( h(k) \)
- A **hash function** \( h \) transforms a key into an index in a hash table \( T[0...N-1] \):
  \[
  h : U \rightarrow \{0, 1, \ldots, N - 1\}
  \]
- We say that \( k \) hashes to \( h(k) \), hash value of \( k \).
- Advantages:
  - Reduce the range of array indices handled: \( N \) instead of \( \max(U) \)
  - Storage is also reduced
Example: HASH TABLES

Suppose that the keys are nine-digit social security numbers

Possible hash function

\[ h(ssn) = sss \mod 100 \] (last 2 digits of ssn)

e.g., if \( ssn = 10123411 \) then \( h(10123411) = 11 \)
Do you see any problems with this approach?

Collisions

- Two or more keys hash to the same slot!!
- For a given set of n keys:
  - If \( n \leq N \), collisions may or may not happen, depending on the hash function
  - If \( n > N \), collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function
Hash Functions

- A hash function transforms a key into a table address
- **What makes a good hash function?**
  1. Easy to compute
  2. Approximates a random function: for every input, every output is equally likely (**simple uniform hashing**)
- In practice, it is very hard to satisfy the simple uniform hashing property
  - i.e., we don't know in advance the probability distribution that keys are drawn from

Good Approaches for Hash Functions

- Minimize the chance that closely related keys hash to the same slot
  - Strings such as *stop*, *tops*, and *pots* should hash to different slots
- **Derive a hash value that is independent from any patterns that may exist in the distribution of the keys.**
The Division Method

- **Idea:**
  - Map a key \( k \) into one of the \( N \) slots by taking the remainder of \( k \) divided by \( N \):
    \[ h(k) = k \mod N \]

- **Advantage:**
  - fast, requires only one operation

- **Disadvantage:**
  - Certain values of \( N \) are bad, e.g.,
    - power of 2
    - non-prime numbers

Example - The Division Method

- If \( N = 2^p \), then \( h(k) \) is just the least significant \( p \) bits of \( k \)
  - \( p = 1 \Rightarrow N = 2 \)
    \[ h(k) = \{0, 1\}, \text{least significant 1 bit of } k \]
  - \( p = 2 \Rightarrow N = 4 \)
    \[ h(k) = \{0, 1, 2, 3\}, \text{least significant 2 bits of } k \]

- Choose \( N \) to be a prime, not close to a power of 2
  - Column 2: \( k \mod 97 \)
  - Column 3: \( k \mod 100 \)
The Multiplication Method

Idea:
- Multiply key $k$ by a constant $A$, where $0 < A < 1$
- Extract the fractional part of $kA$
- Multiply the fractional part by $N$
- Take the floor of the result

$$h(k) = \lfloor N \cdot (kA - \lfloor kA \rfloor) \rfloor$$

- Disadvantage: A little slower than division method
- Advantage: Value of $N$ is not critical, e.g., typically $2^p$

Hash Functions

- A hash function is usually specified as the composition of two functions:
  - Hash code: $h_1$: keys $\rightarrow$ integers
  - Compression function: $h_2$: integers $\rightarrow [0, N-1]$

Typically, $h_2$ is mod $N$.

- The hash code is applied first, and the compression function is applied next on the result, i.e.,
  $$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to “disperse” the keys in an apparently random way
Typical Function for $H_1$

- **Polynomial accumulation:**
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits): $a_0a_1\ldots a_{n-1}$.
  - We evaluate the polynomial $p(z) = a_0 + a_1z + a_2z^2 + \ldots + a_{n-1}z^{n-1}$ at a fixed value $z$, ignoring overflows.
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words).

- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner\’s rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time:
    - $p_0(z) = a_{n-1}$
    - $p_i(z) = a_{n-i-1} + zp_{i-1}(z)$ ($i = 1, 2, \ldots, n-1$)
  - We have $p(z) = p_{n-1}(z)$.
  - Good values for $z$: 33, 37, 39, and 41.

Compression Functions

- **Division:**
  - $h_2(y) = y \mod N$
  - The size $N$ of the hash table is usually chosen to be a prime number.
  - The reason has to do with number theory and is beyond the scope of this course.

- **Random linear hash function:**
  - $h_2(y) = (ay + b) \mod N$
  - $a$ and $b$ are random nonnegative integers such that $a \mod N \neq 0$.
  - Otherwise, every integer would map to the same value $b$.
Handling Collisions

- We will review the following methods:
  - Separate Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Handling Collisions Using Chaining

**Idea:**
- Put all elements that hash to the same slot into a linked list
- Slot $j$ contains a pointer to the head of the list of all elements that hash to $j$
Collision with Chaining

- Choosing the size of the table
  - Small enough not to waste space
  - Large enough such that lists remain short
  - Typically 1/5 or 1/10 of the total number of elements

- How should we keep the lists: ordered or not?
  - Not ordered!
    - Insert is fast
    - Can easily remove the most recently inserted elements

Algorithm `put(k, v):`  

```
// k is a new key
int h(k), put(k, v)
int n = n + 1
return t
```

- 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
Deletion in Hash Tables

**Algorithm** remove(k):
   t = A[h(k)].remove(k)
   if t ≠ null then {k was found}
      n = n - 1
   return t

- Need to find the element to be deleted.
- Worst-case running time:
  - Deletion depends on searching the corresponding list

Searching in Hash Tables

**Algorithm** get(k):
   return A[h(k)].get(k)

- Running time is proportional to the length of the list of elements in slot h(k)
Analysis of Hashing with Chaining: Worst Case

- How long does it take to search for an element with a given key?
- Worst case:
  - All \( n \) keys hash to the same slot
  - Worst-case time to search is \( \Theta(n) \), plus time to compute the hash function

Analysis of Hashing with Chaining: Average Case

- Average case
  - depends on how well the hash function distributes the \( n \) keys among the \( N \) slots
- Simple uniform hashing assumption:
  - Any given element is equally likely to hash into any of the \( N \) slots (i.e., probability of collision \( \Pr(h(x)=h(y)) \), is \( 1/N \))
- Length of a list:
  - \( T[j].size = n_j, \ j = 0, 1, \ldots, N - 1 \)
- Number of keys in the table:
  - \( n = n_0 + n_1 + \cdots + n_{N-1} \)
- Load factor: Average value of \( n_j \):
  - \( E[n_j] = \alpha = n/N \)
Load Factor of a Hash Table

- Load factor of a hash table $T$:
  \[ \alpha = \frac{n}{N} \]
  - $n$ = # of elements stored in the table
  - $N$ = # of slots in the table = # of linked lists
- $\alpha$ encodes the average number of elements stored in a chain
- $\alpha$ can be $<, =, > 1$

Case 1: Unsuccessful Search (i.e., item not stored in the table)

**Theorem** An unsuccessful search in a hash table takes expected time $\Theta(1 + \alpha)$ under the assumption of simple uniform hashing (i.e., probability of collision $Pr(h(x)=h(y))$, is 1/N).

**Proof**

- Searching unsuccessfully for any key $k$
  - need to search to the end of the list $T[h(k)]$
- Expected length of the list: $E[n_{h(k)}] = \alpha = n/N$
- Expected number of elements examined in this case is $\alpha$
- Total time required is:
  - $O(1)$ (for computing the hash function) + $\alpha \Rightarrow \Theta(1 + \alpha)$
Case 2: Successful Search

Successful search: $\Theta(1 + \frac{a}{2}) = \Theta(1 + a)$ time on the average

(search half of a list of length $a$ plus $O(1)$ time to compute $h(k)$)

Analysis of Search in Hash Tables

- If $N$ (# of slots) is proportional to $n$ (# of elements in the table):
  - $n = \Theta(N)$
  - $\alpha = n/N = \Theta(N)/N = O(1)$
  - $\Rightarrow$ Searching takes constant time on average
Open Addressing

- If we have enough contiguous memory to store all the keys ⇒ store the keys in the table itself
- No need to use linked lists anymore
- Basic idea:
  - put: if a slot is full, try another one, until you find an empty one
  - get: follow the same sequence of probes
  - remove: more difficult ... (we’ll see why)
- Search time depends on the length of the probe sequence!

Generalize hash function notation:

- A hash function contains two arguments now: (i) Key value, and (ii) Probe number
  \[ h(k,p), \quad p=0,1,...,N-1 \]
- Probe sequences
  \[ <h(k,0), h(k,1), ..., h(k,N-1)> \]
  - Must be a permutation of \(<0,1,...,N-1>\)
  - There are \(N!)\) possible permutations
  - Good hash functions should be able to produce all \(N!)\) probe sequences
Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

Note: None of these methods can generate more than $N^2$ different probing sequences!

Linear probing

- Idea: when there is a collision, check the next available position in the table (i.e., probing)
  \[ h(k,i) = (h_1(k) + a*i) \mod N \]
  \[ i=0,1,2,... \]
- First slot probed: $h_1(k)$
- Second slot probed: $h_1(k) + 1$ ($a=1$)
- Third slot probed: $h_1(k)+2$, and so on

- Can generate $N$ probe sequences maximum, why?
  probe sequence: $< h_1(k), h_1(k)+1, h_1(k)+2, ..., >$
Linear probing: Searching for a key

- Three cases:
  1. Position in table is occupied with an element of equal key
  2. Position in table is empty
  3. Position in table occupied with a different element

- Case 3: probe the next index until the element is found or an empty position is found
- The process wraps around to the beginning of the table

Search with Linear Probing

- Consider a hash table $A$ that uses linear probing
- $\text{get}(k)$
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    * An item with key $k$ is found, or
    * An empty cell is found, or
    * $N$ cells have been unsuccessfully probed

Algorithm $\text{get}(k)$

```
i \leftarrow h(k)
p \leftarrow 0
\text{repeat}
c \leftarrow A[i]
\text{if } c = \emptyset
\text{return null}
\text{else if } c.getKey() = k
\text{return } c.getValue()
\text{else}
i \leftarrow (i + 1) \mod N
p \leftarrow p + 1
\text{until } p = N
\text{return null}
```
**Quadratic Probing**

\[ h(k,i) = (h_1(k) + i^2) \mod N \]

- **Probe sequence:**
  - 0th probe = \( h(k) \mod N \)
  - 1st probe = \( (h(k) + 1) \mod N \)
  - 2nd probe = \( (h(k) + 4) \mod N \)
  - 3rd probe = \( (h(k) + 9) \mod N \)
  - \ldots
  - \( i \)th probe = \( (h(k) + i^2) \mod N \)

**Quadratic Probing Example**

- **Insert(76)**: \( 76 \% 7 = 6 \)
- **Insert(40)**: \( 40 \% 7 = 5 \)
- **Insert(48)**: \( 48 \% 7 = 6 \)
- **Insert(5)**: \( 5 \% 7 = 5 \)
- **Insert(55)**: \( 55 \% 7 = 6 \)

But...

- **Insert(47)**: \( 47 \% 7 = 5 \)
Quadratic Probing:
Success guarantee for $\alpha < \frac{1}{2}$

- If $N$ is prime and $\alpha < \frac{1}{2}$, then quadratic probing will find an empty slot in $N/2$ probes or fewer, because each probe checks a different slot.
- Show for all $0 \leq i, j \leq N/2$ and $i \neq j$
  
  $$(h(x) + i^2) \mod N \neq (h(x) + j^2) \mod N$$

- By contradiction: suppose that for some $i \neq j$:
  
  $$(h(x) + i^2) \mod N = (h(x) + j^2) \mod N$$
  
  $$\Rightarrow i^2 \mod N = j^2 \mod N$$
  
  $$\Rightarrow (i^2 - j^2) \mod N = 0$$
  
  $$\Rightarrow [(i + j)(i - j)] \mod N = 0$$

Because $N$ is prime, $(i-j)$ or $(i+j)$ must be zero, and neither can be, a contradiction.

**Conclusion:** For any $\alpha < \frac{1}{2}$, quadratic probing will find an empty slot; for bigger $\alpha$, quadratic probing may find a slot.

Double Hashing

1. Use one hash function to determine the first slot
2. Use a second hash function to determine the increment for the probe sequence

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod N, \quad i=0,1,...$$

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \mod N$, so on...
- Advantage: avoids clustering
- Disadvantage: harder to delete an element
- Can generate $N^2$ probe sequences maximum
### Double Hashing: Example

- \( h_1(k) = k \mod 13 \)
- \( h_2(k) = 1 + (k \mod 11) \)
- \( h(k, i) = (h_1(k) + i h_2(k)) \mod 13 \)

Insert key 14:

- \( h_1(14, 0) = 14 \mod 13 = 1 \)
- \( h(14, 1) = (h_1(14) + h_2(14)) \mod 13 = (1 + 4) \mod 13 = 5 \)
- \( h(14, 2) = (h_1(14) + 2 h_2(14)) \mod 13 = (1 + 8) \mod 13 = 9 \)

---

### Analysis of Open Addressing

- Ignore the problem of clustering and assume that all probe sequences are equally likely.

**Unsuccessful retrieval:**

- \( \text{Prob(probe hits an occupied cell)} = a \) (load factor)
- \( \text{Prob(probe hits an empty cell)} = 1 - a \)

- Probability that a probe terminates in 2 steps: \( a(1 - a) \)
- Probability that a probe terminates in \( k \) steps: \( a^{k-1}(1 - a) \)

What is the average number of steps in a probe?

\[
E(\#\text{steps}) = \sum_{k=1}^{\infty} ka^{k-1}(1 - a) \leq \sum_{k=1}^{\infty} ka^{k-1}(1 - a) = (1 - a) \frac{1}{(1 - a)^2} = \frac{1}{1 - a}
\]
Rehashing

Idea: When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - half full ($\alpha = 0.5$)
  - when an insertion fails
  - some other threshold

- Cost of rehashing?