Hashing
Observation: We can store a set very easily if we can use its keys as array indices:

\[ A: \begin{array}{c}
k_1 \\
\downarrow \\
k_2 \\
\end{array} \rightarrow \text{record with key } k_1 \]

\[ \begin{array}{c}
\uparrow \\
\end{array} \rightarrow \text{record with key } k_2 \]

E.g. \text{SEARCH}(A,k)

\text{return } A[k]
Problem: usually, the number of possible keys is *far* larger than the number of keys actually stored, or even than available memory. (E.g., strings.)

Idea of hashing: use a function $h$ to map keys into a smaller set of indices, say the integers $0..m$. This function is called a *hash function*.

E.g. $h(k) =$ position of $k$’s first letter in the alphabet.
Problem: Collisions. They are \textit{inevitable} if there are more possible key values than table slots.
Two questions:
1. How can we choose the hash function to minimize collisions?
2. What do we do about collisions when they occur?
Running times for hashing ($\Theta$ assumed):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>DELETE</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>SEARCH</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>SUCCESSOR</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>PREDECESSOR</td>
<td>$n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

So hashing is useful when worst-case guarantees and ordering are not required.
Real-World Facts (shhh!)

Hashing is *vastly* more prevalent than trees for in-memory storage.

Examples:
- UNIX shell command cache
- “arrays” in Icon, Awk, Tcl, Perl, etc.
- compiler symbol tables
- Filenames on CD-ROM
Example: Scripting Language

WORD - FREQUENCY:

count ← new array initialized to 0
for each word in the input do
    count[word] ← count[word] + 1
for each key in sort(keys[count]) do
    print key, count[key]
Resolving Collisions

Let’s assume for now that our hash function is OK, and deal with the collision resolution problem.

Two groups of solutions:
1. Store the colliding key in the hash-table array. (“Closed hashing”)
2. Store it somewhere else. (“Open hashing”)
(Note: CLRS calls #1 “open addressing.”)

Let’s look at #2 first.
Open Hashing: Collision Resolution by Chaining

Put all the keys that hash to the same index onto a linked list. Each T[i] called a bucket or slot.

- T: 1
  - Andy

- 2
  - Cindy

- 3
  - Cindy

- 20
  - Thomas
  - Tony
Code for a Chained Hash Table

HASH - INSERT(T, x)

b ← h(key[x])  \item hash to find bucket
y ← LIST - SEARCH(T[b], key[x])
if y = NIL then
    T[b] ← LIST - INSERT(T[b], x)
else  \item replace existing entry
    LIST - REPLACE(y, x)
Chained Hash Table (Continued)

HASH – SEARCH(T, k)
    return LIST - SEARCH(T[h(k)], k)

HASH - DELETE(T, x)
    b ← h(key[x])
    T[b] ← LIST - DELETE(T[b], x)
Analysis of Hashing with Chaining

- If INSERT didn't care about finding an existing record, it would take $\theta(1)$ time.
- DELETE on a doubly-linked list takes $\theta(1)$ time.
- Everything else is proportional to the length of the list.

Worst case: Everything hashes to the same slot.
Then INSERT and SEARCH take $\theta(n)$ time. Yecch.
Analysis of Hashing with Chaining (continued)

Average Case:
Assume $h(k)$ is equally likely to be any slot, regardless of other keys’ hash values. This assumption is called *simple uniform hashing*.

(By the way, we also assume throughout that $h$ takes constant time to compute.)
Average time for an unsuccessful search, assuming simple uniform hashing:
Time for hashing = $\theta(1)$.
Time to search list = $\theta(\text{avg. length of list})$
If there are $n$ items in a table with $m$ slots, then the average length of a list is $n/m$.
Call this the load factor, $\alpha$ : \[ \alpha = \frac{n}{m} \]
So avg. time to search to the end of a list is $\alpha$
So average time for an unsuccessful search = $\theta(1 + \alpha)$
Average time for successful search:

- Assume that INSERT puts new keys at the end of the list. (The result is the same regardless of where INSERT puts the key.)
- Then the work we do to find a key is the same as what we did to insert it. And that is the same as successful search.
- Let’s add up the total time to search for all the keys in the table. (Then we’ll divide by n, the number of keys, to get the average.)
- We’ll go through the keys in the order they were inserted.
Time to insert first key: $1 + 0/m$
Time to insert second key: $1 + 1/m$
Time to insert $i$th key: $1 + \frac{i-1}{m}$

Avg. time for successful search =

$$\frac{1}{n} \sum_{i=1}^{n} (1 + \frac{i-1}{m}) = \frac{1}{n} \sum_{i=1}^{n} 1 + \frac{1}{n} \sum_{i=1}^{n} \frac{i-1}{m} = 1 + \frac{1}{nm} \sum_{i=1}^{n} i - 1$$

$$= 1 + \frac{1}{nm} \left( \frac{(n-1)n}{2} \right) = 1 + \frac{n-1}{2m} = 1 + \frac{n}{2m} - \frac{1}{2m}$$

Recall $\alpha = \frac{n}{m}$

$$= 1 + \frac{\alpha}{2} - \frac{1}{2m} = \Theta(1 + \alpha)$$
INSERT does either a successful or an unsuccessful search, so it also takes time $\theta(1 + \alpha)$. So all operations take time $O(1 + \alpha)$. If the size of the table grows with the number of items, then $\alpha$ is a constant and hashing takes $\theta(1)$ avg. case for anything. If you don't grow the table, performance is $\Theta(n)$, even on average.
Growing

To grow: Whenever $\alpha \geq$ some threshold (e.g. 3/4), double the number of slots.

Requires rehashing everything—but by the same analysis we did for growing arrays, the amortized time for INSERT will remain $\Theta(1)$, average case.
Collision Resolution, Idea #2

Store colliders in the hash table array itself:

<table>
<thead>
<tr>
<th>T:</th>
<th>1</th>
<th>Andy</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Cindy</td>
<td></td>
</tr>
</tbody>
</table>

("Closed hashing" or "Open addressing")

20 | Tony | Insert Thomas

<table>
<thead>
<tr>
<th>20</th>
<th>Tony</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>21</th>
<th>Thomas</th>
</tr>
</thead>
</table>

hashing
Collision Resolution, Idea #2

Advantage:
– No extra storage for lists

Disadvantages:
– Harder to program
– Harder to analyze
– Table can overflow
– Performance is worse
When there is a collision, where should the new item go?
Many answers. In general, think of the hash function as having two arguments: the key and a probe number saying how many times we’ve tried to find a place for the items.

(Code for INSERT and SEARCH is in CLRS, p.238.)
Probing Methods

*Linear probing*: if a slot is occupied, just go to the next slot in the table. (Wrap around at the end.)

\[ h(k, i) = (h'(k) + i) \mod m \]

- key
- probe #
- our original hash function
- \# of slots in table
Closed Hashing Algorithms

\textbf{INSERT}(T, x) \triangleright in this version, we don't check for duplicates

\begin{align*}
p & \leftarrow \text{the first probe} \\
\text{while } T[p] \text{ is not empty do } \triangleright \text{ assumes } T \text{ is not full} \\
\quad p & \leftarrow \text{the next probe} \\
T[p] & \leftarrow x
\end{align*}
SEARCH(T, k)

p ← the first probe

while T[p] is not empty do     ⚪ again, assumes T is not full

    if T[p] is empty then
        return NIL

    else if key[T[p]] = k then
        return T[p]

else

    p ← next probe

DELETE … is best avoided with closed hashing
Example of Linear Probing

\[ h(k,i) = (h'(k)+i) \mod m \]

INSERT(d). \( h'(d) = 3 \)

\[
\begin{array}{c|c|c|c|c}
  i & 0 & 1 & 2 & 3 \\
  \hline
  h'(d,i) & 3 & 4 & 0 &
\end{array}
\]

Put d in slot 0

Problem: long runs of items tend to build up, slowing down the subsequent operations. *(primary clustering)*

hashing
Quadratic Probing

\[ h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m \]

\(\uparrow\) \(\uparrow\)

two constants, fixed at “compile-time”

Better than linear probing, but still leads to clustering, because keys with the same value for \(h'\) have the same probe sequence.
Double Hashing

Use one hash function to start, and a second to pick the probe sequence:

\[ h(k, i) = (h_1(k) + ih_2(k)) \mod m \]

\( h_2(k) \) must be relatively prime in \( m \) in order to sweep out all slots. E.g. pick \( m \) a power of 2 and make \( h_2(k) \) always odd.
Linear and quadratic probing give us \( m \) probe sequences, because each value \( h'(k) \) results in a different, fixed sequence:

\[
\begin{align*}
h'(k) &= 3 \rightarrow 3 4 5 \ldots \quad (h'(k) \text{ has values from } 0 \text{ to } m - 1) \\
h'(k) &= 8 \rightarrow 8 9 10
\end{align*}
\]

Double hashing gives about \( m^2 \) sequences, because every pair \((h_1(k), h_2(k))\) yields a different probe sequence. The analysis assumes uniform hashing, which holds that all of the \( m! \) possible probe sequences are equally likely.

Though \( m! \gg m^2 \), in practice double hashing's performance is close to uniform hashing's.
Analysis of closed hashing (assuming uniform hashing):

Recall: \( \alpha = \frac{\text{# of keys}}{\text{# of slots}} \).

Here \( 0 \leq \alpha \leq 1 \). (With open hashing, \( \alpha \) can be \( > 1 \).)

Time for unsuccessful search: let's count probes.

Worst case = \( n \) (you hit every key before you hit a blank slot)
Avg case: assume a very large table.

Probability of doing a first probe: 1

Prob of 2nd probe = prob that 1st is occupied \( \approx \alpha \)

Prob of 3rd probe = (prob of 2nd probe) \( \times \)

\[ (\text{prob. 2nd is occ.}) \approx \alpha \alpha \]
Expected # of probes  =  1 + \alpha + \alpha^2 + \ldots

< \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha}
closed hashing, unsuccessful search : \( \frac{1}{1 - \alpha} \)
open hashing unsuccessful search : \( 1 + \alpha \)
Which is better?
Note : \( \frac{1}{1 - \alpha} = 1 + \frac{\alpha}{1 - \alpha} \)

When is \( \frac{\alpha}{1 - \alpha} < a \)? When \( 0 \leq \alpha \leq 1 \), \( \frac{\alpha}{1 - \alpha} \) is always > \( \alpha \).

It's only less when \( \alpha > 1 \) - but this can't happen in closed hashing!
So open hashing always wins an unsuccessful search.
Successful search : # of probes in closed hashing is at most 
\[ \frac{1}{\alpha} \ln \frac{1}{1 - \alpha} \] (Proof omitted). This is < 4 for \( \alpha < 90\% \).
Choosing a Good Hash Function

It should run quickly, and “hash” the keys up—each key should be equally likely to fit in any slot.

General rules:
– Exploit known facts about the keys
– Try to use all bits of the key
Choosing A Good Hash Function (Continued)

Although most commonly strings are being hashed, we’ll assume $k$ is an integer.
Can always interpret strings (byte sequences) as numbers in base 256:

"cat" = 'c'×256$^2$ + 'a'×256 + 't'
The division method:

\[ h(k) = k \mod m \]  \quad (m \text{ is still the \# of slots})

Very simple— but \( m \) must be chosen carefully.

- E.g. if you’re hashing decimal integers, then \( m = \text{a power of ten} \) means you’re just taking the low-order digits.

- If you’re hashing strings, then \( m = 256 \) means the last character.

best to choose \( m \) to be a prime far from a power of 2
The multiplication method:
\[ h(k) = \lfloor m(kA \mod 1) \rfloor \]

(the fractional part of \( kA \))

Choose \( A \) in the range 0…1. Choice of \( m \) is not critical.
Hash Functions in Practice

• Almost all hashing is done on strings. Typically, one computes byte-by-byte on the string to get a non-negative integer, then takes it mod $m$.

• E.g. (sum of all the bytes) mod $m$.

• Problem: anagrams hash to the same value.

• Other ideas: xor, etc.

• Hash function in Microsoft Visual C++ class library:

\[
x = 0
\]

\[
\text{for } i \leftarrow 1 \text{ to } \text{length}[s] \text{ do }
\]

\[
x \leftarrow 33x + \text{int}_{\text{hashing}}(s[i])
\]