Lecture 18
Hashing
• Direct-access tables
• Resolving collisions by chaining
• Choosing hash functions
Symbol-table problem

Symbol table $S$ holding $n$ records:

- **record** $x$
- **key** $[x]$
- Other **fields** containing **satellite data**

“Dictionary” ADT
Symbol-table problem

Symbol table $S$ holding $n$ records:

- **key**[$x$]
- Other **fields** containing satellite data

Operations on $S$:
- $\text{INSERT}(S, x)$
- $\text{DELETE}(S, x)$
- $\text{SEARCH}(S, k)$

“Dictionary” ADT
Symbol-table problem

Symbol table $S$ holding $n$ records:

- **key** $[x]$
- Other **fields** containing **satellite data**

Operations on $S$:
- **INSERT**($S, x$)
- **DELETE**($S, x$)
- **SEARCH**($S, k$)

“Dictionary” ADT

How should the data structure $S$ be organized?
Direct-access table

**Idea:** Suppose that the keys are drawn from the set $U \subseteq \{0, 1, \ldots, m-1\}$, and keys are distinct. Set up an array $T[0 \ldots m-1]$:

$$T[k] = \begin{cases} 
    x & \text{if } x \in K \text{ and } \text{key}[x] = k, \\
    \text{NIL} & \text{otherwise.}
\end{cases}$$

Then, operations take $\Theta(1)$ time.
**Direct-access table**

**Idea:** Suppose that the keys are drawn from the set $U \subseteq \{0, 1, \ldots, m-1\}$, and keys are distinct. Set up an array $T[0 \ldots m-1]$:  
\[
T[k] = \begin{cases} 
  x & \text{if } x \in K \text{ and } key[x] = k, \\
  \text{NIL} & \text{otherwise.}
\end{cases}
\]

Then, operations take $\Theta(1)$ time.

**Problem:** The range of keys can be large:
- 64-bit numbers (which represent $18,446,744,073,709,551,616$ different keys),
- character strings (even larger!).
Resolving collisions

- Two popular methods
  - Chaining
  - Open Addressing

- Both allow Insert, Delete and Find in expected constant time with storage $O(n)$
Hash functions

Solution: Use a hash function $h$ to map the universe $U$ of all keys into \{0, 1, \ldots, m–1\}:
Hash functions

Solution: Use a hash function $h$ to map the universe $U$ of all keys into \{0, 1, \ldots, m\text{-}1\}:

As each key is inserted, $h$ maps it to a slot of $T$. 
Hash functions

Solution: Use a hash function $h$ to map the universe $U$ of all keys into $\{0, 1, \ldots, m-1\}$:

As each key is inserted, $h$ maps it to a slot of $T$. 
Hash functions

Solution: Use a hash function $h$ to map the universe $U$ of all keys into $\{0, 1, \ldots, m-1\}$:

As each key is inserted, $h$ maps it to a slot of $T$. 
Hash functions

Solution: Use a hash function $h$ to map the universe $U$ of all keys into $\{0, 1, \ldots, m-1\}$:

As each key is inserted, $h$ maps it to a slot of $T$. 
Solution: Use a **hash function** $h$ to map the universe $U$ of all keys into $\{0, 1, \ldots, m-1\}$:

As each key is inserted, $h$ maps it to a slot of $T$. 
Solution: Use a hash function $h$ to map the universe $U$ of all keys into \( \{0, 1, \ldots, m-1\} \):

As each key is inserted, $h$ maps it to a slot of $T$. 
**Hash functions**

**Solution:** Use a *hash function* $h$ to map the universe $U$ of all keys into $\{0, 1, \ldots, m-1\}$:

When a record to be inserted maps to an already occupied slot in $T$, a *collision* occurs.
Resolving collisions by chaining

- Link records in the same slot into a list.

Worst case:
- Every key hashes to the same slot.
- Access time = $\Theta(n)$ if $|S| = n$
Resolving collisions by chaining

- Link records in the same slot into a list.

$T$

$i \quad 49 \quad 86 \quad 52$

Worst case:
- Every key hashes to the same slot.
- Access time $= \Theta(n)$ if $|S| = n$
Resolving collisions by chaining

- **INSERT(\(T, x\))**
  Insert \(x\) at head of list \(T[h(key[x])]\)

- **DELETE(\(T, x\))**
  Remove \(x\) from list \(T[h(key[x])]\)

- **SEARCH(\(T, k\))**
  Search for an element with key \(k\) in list \(T[h(key[x])]\)
Review: Analysis of Chaining

• Suppose all the lists contain exactly $t$ elements, how long do the following operations take?
  - INSERT $\Theta(1)$
  - DELETE $\Theta(t)$
  - FIND $\Theta(t)$
Average-case analysis of chaining

We make the assumption of *simple uniform hashing*: 

- Each key $k \in S$ is equally likely to be hashed to any slot of table $T$, independent of where other keys are hashed.

Let $n$ be the number of keys in the table, and let $m$ be the number of slots.

Define the **load factor** of $T$ to be

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

= average number of keys per slot.
Search cost

The expected time for an unsuccessful search for a record with a given key is

$$= \Theta(1 + \alpha).$$
Search cost

The expected time for an unsuccessful search for a record with a given key is

$$= \Theta(1 + \alpha).$$
Search cost

The expected time for an **unsuccessful** search for a record with a given key is \( = \Theta(1 + \alpha) \).

**Proof:**

- Say \( i \)-th list contains \( t_i \) elements
- Searching the \( i \)-th list takes \( \Theta(t_i + 1) \) time (we add a 1 to take care of the case where the list is empty, i.e. \( t = 0 \)).
- Because the key \( k \) is not already in the table, \( h(k) \) is equally likely to be any value
- We can compute the overall expected time (next slide)
The expected time for an *unsuccessful* search for a record with a given key is $= \Theta(1 + \alpha)$.

**Proof, continued:**

\[
E(\text{time for unsuccessful search}) = \sum_{i=1}^{m} \Pr[h(k) = i] \cdot \text{(time to search list } T[i])
\]

\[
= \frac{1}{m} \sum_{i=1}^{m} \Theta(1 + t_i) = \Theta\left(\frac{1}{m} (m + \sum_{i=1}^{m} t_i)\right)
\]

\[
= \Theta(1 + \frac{n}{m}) = \Theta(1 + \alpha)
\]
Search cost

The expected time for an unsuccessful search for a record with a given key is

\[ \Theta(1 + \alpha). \]

Expected search time \( = \Theta(1) \) if \( \alpha = O(1) \),
or equivalently, if \( n = O(m) \).
Search cost

The expected time for an *unsuccessful* search for a record with a given key is

\[ = \Theta(1 + \alpha). \]

Expected search time \( = \Theta(1) \) if \( \alpha = O(1) \), or equivalently, if \( n = O(m) \).

A *successful* search has same asymptotic bound, but a rigorous argument is a little more complicated. (See textbook.)
Choosing a hash function

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

**Desiderata:**

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.
The remaining material in these notes was not covered in the lecture.
Division method

Assume all keys are integers, and define

\[ h(k) = k \mod m. \]

**Deficiency:** Don’t pick an \( m \) that has a small divisor \( d \). A preponderance of keys that are congruent modulo \( d \) can adversely affect uniformity.
Division method

Assume all keys are integers, and define

\[ h(k) = k \mod m. \]

**Deficiency:** Don’t pick an \( m \) that has a small divisor \( d \). A preponderance of keys that are congruent modulo \( d \) can adversely affect uniformity.

**Extreme deficiency:** If \( m = 2^r \), then the hash doesn’t even depend on all the bits of \( k \):
- If \( k = 1011000111011010_2 \) and \( r = 6 \), then \( h(k) = 011010_2 \).
Division method

Assume all keys are integers, and define

\[ h(k) = k \mod m. \]

**Deficiency:** Don’t pick an \( m \) that has a small divisor \( d \). A preponderance of keys that are congruent modulo \( d \) can adversely affect uniformity.

**Extreme deficiency:** If \( m = 2^r \), then the hash doesn’t even depend on all the bits of \( k \):

- If \( k = 1011000111011010_2 \) and \( r = 6 \), then \( h(k) = 011010_2 \).
Division method (continued)

\[ h(k) = k \mod m. \]

Pick \( m \) to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

**Annoyance:**

- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we’ll see is usually superior.
Multiplication method

Assume that all keys are integers, $m = 2^r$, and our computer has $w$-bit words. Define

$$h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r),$$

where rsh is the “bitwise right-shift” operator and $A$ is an odd integer in the range $2^{w-1} < A < 2^w$.

- Don’t pick $A$ too close to $2^{w-1}$ or $2^w$.
- Multiplication modulo $2^w$ is fast compared to division.
- The rsh operator is fast.
Multiplication method example

\[ h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r) \]

Suppose that \( m = 8 = 2^3 \) and that our computer has \( w = 7 \)-bit words:

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & \times \\
1 & 1 & 0 & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1
\end{array}
\]

\[ h(k) = 1001010011 \]

\[ A = 1011001 \]

\[ k = 1101011 \]
Multiplication method example

\[ h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r) \]

Suppose that \( m = 8 = 2^3 \) and that our computer has \( w = 7 \)-bit words:

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & \times \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1
\end{array}
\]

\( h(k) \)

\( A \)

\( k \)

Modular wheel
Multiplication method example

\[ h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r) \]

Suppose that \( m = 8 = 2^3 \) and that our computer has \( w = 7 \)-bit words:

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & \times \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & \hline \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
\end{array}
\]

\( A = k = \)

\( h(k) \)

Modular wheel
**Multiplication method example**

\[ h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r) \]

Suppose that \( m = 8 = 2^3 \) and that our computer has \( w = 7 \)-bit words:

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & \\
\times & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}
\]

Suppose that \( m = 8 = 2^3 \) and that our computer has \( w = 7 \)-bit words:
Multiplication method example

\[ h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r) \]

Suppose that \( m = 8 = 2^3 \) and that our computer has \( w = 7 \)-bit words:

\[
\begin{array}{c}
1 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}
= A
\]

\[
\begin{array}{c}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}
= k
\]

Modular wheel
Resolving collisions by open addressing

No storage is used outside of the hash table itself.

• Insertion systematically probes the table until an empty slot is found.

• The hash function depends on both the key and probe number:
  \[ h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}. \]

• The probe sequence \( \langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle \) should be a permutation of \( \{0, 1, \ldots, m-1\} \).

• The table may fill up, and deletion is difficult (but not impossible).
Example of open addressing

Insert key $k = 496$:
Example of open addressing

Insert key $k = 496$:

0. Probe $h(496,0)$

```plaintext
0  586  133  204  481
```

$T$ collision $m-1$

---

S. Raskhodnikova and A. Smith. Based on slides by E. Demaine and C.E. Leiserson

Feb. 28 2007
Example of open addressing

Insert key \( k = 496 \):

0. Probe \( h(496,0) \)

\[
\begin{align*}
T & \quad 0 \\
586 & \\
133 & \\
204 & \\
481 & \\
\end{align*}
\]

\( m - 1 \)
Example of open addressing

Insert key $k = 496$:

0. Probe $h(496,0)$

1. Probe $h(496,1)$

Collision
Example of open addressing

Insert key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
Example of open addressing

Insert key \( k = 496 \):

0. Probe \( h(496,0) \)

1. Probe \( h(496,1) \)

2. Probe \( h(496,2) \)
Example of open addressing

Search for key $k = 496$:

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.
Example of open addressing

Search for key $k = 496$:

0. Probe $h(496,0)$

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.
Example of open addressing

Search for key $k = 496$:

0. Probe $h(496,0)$

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.
Example of open addressing

Search for key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.
Example of open addressing

Search for key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.
Example of open addressing

Search for key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.
Probing strategies

Linear probing:
Given an ordinary hash function $h'(k)$, linear probing uses the hash function

$$h(k,i) = (h'(k) + i) \mod m.$$  

This method, though simple, suffers from primary clustering, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.
Double hashing

Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m.$$ 

This method generally produces excellent results, but $h_2(k)$ must be relatively prime to $m$. One way is to make $m$ a power of 2 and design $h_2(k)$ to produce only odd numbers.
Analysis of open addressing

We make the assumption of uniform hashing:

• Each key is equally likely to have any one of the $m!$ permutations as its probe sequence.
Analysis of open addressing

We make the assumption of \textit{uniform hashing:}

- Each key is equally likely to have any one of the $m!$ permutations as its probe sequence.

\textbf{Theorem.} Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$. 
Implications of the theorem

- If $\alpha$ is constant, then accessing an open-addressed hash table takes constant time.
- If the table is half full, then the expected number of probes is $1/(1-0.5) = 2$.
- If the table is 90% full, then the expected number of probes is $1/(1-0.9) = 10$. 