

Hash Tables

Prof. Zheng Zhang

Harbin Institute of Technology, Shenzhen

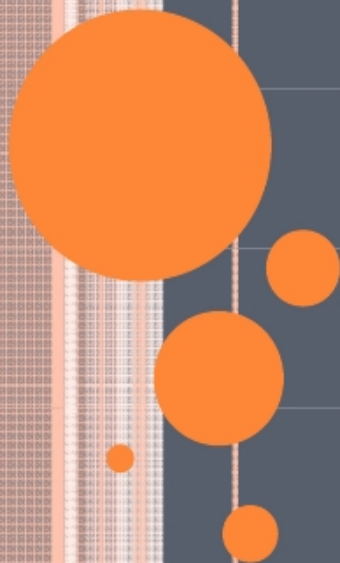


Outline

- Dynamic Sets
- Hash Tables
 - Direct-access tables
 - Resolving collisions by chaining
 - Choosing hash functions
 - Open addressing



Dynamic Sets



Dynamic Sets

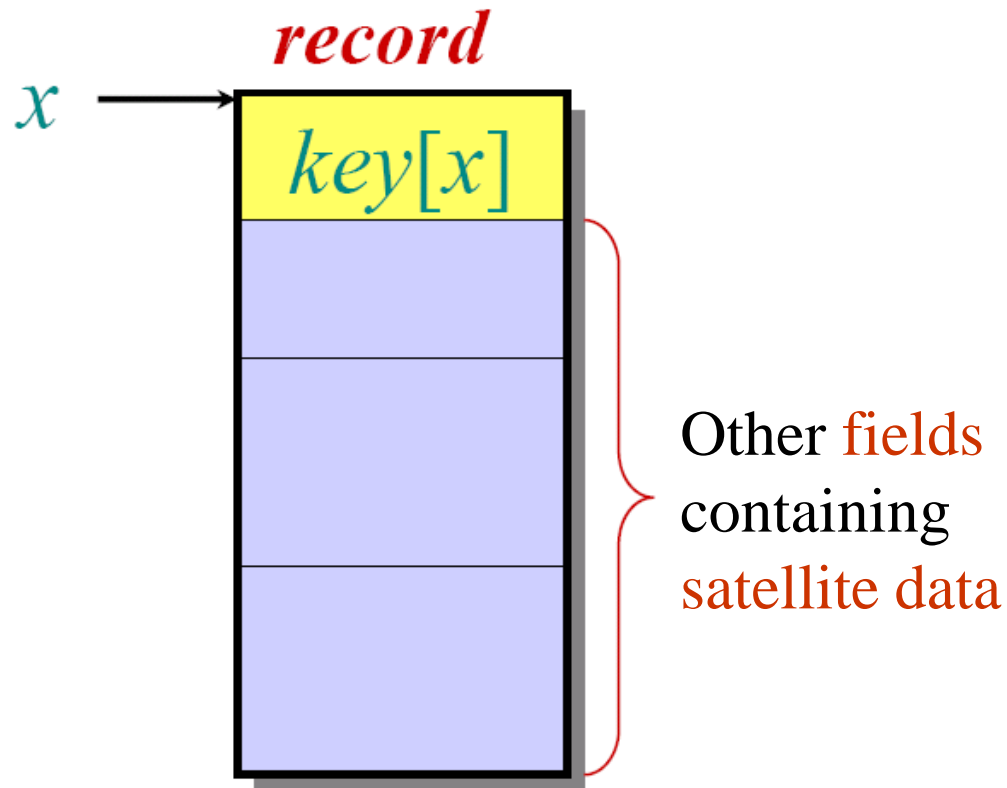
- Structures for *dynamic sets*
 - Elements have a *key* and *satellite data*
 - Dynamic sets may support *queries* such as:
 - *Search*(S, k), *Minimum*(S), *Maximum*(S), *Successor*(S, x), *Predecessor*(S, x)
 - They may also support *modifying operations* like:
 - *Insert*(S, x), *Delete*(S, x)

Hash Tables



Symbol-table problem

Symbol table S holding n *records*:



Operations on S :

- INSERT (S, x)
- DELETE (S, x)
- SEARCH (S, k)

How should the data structure S be organized?

Direct-access table

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

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

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

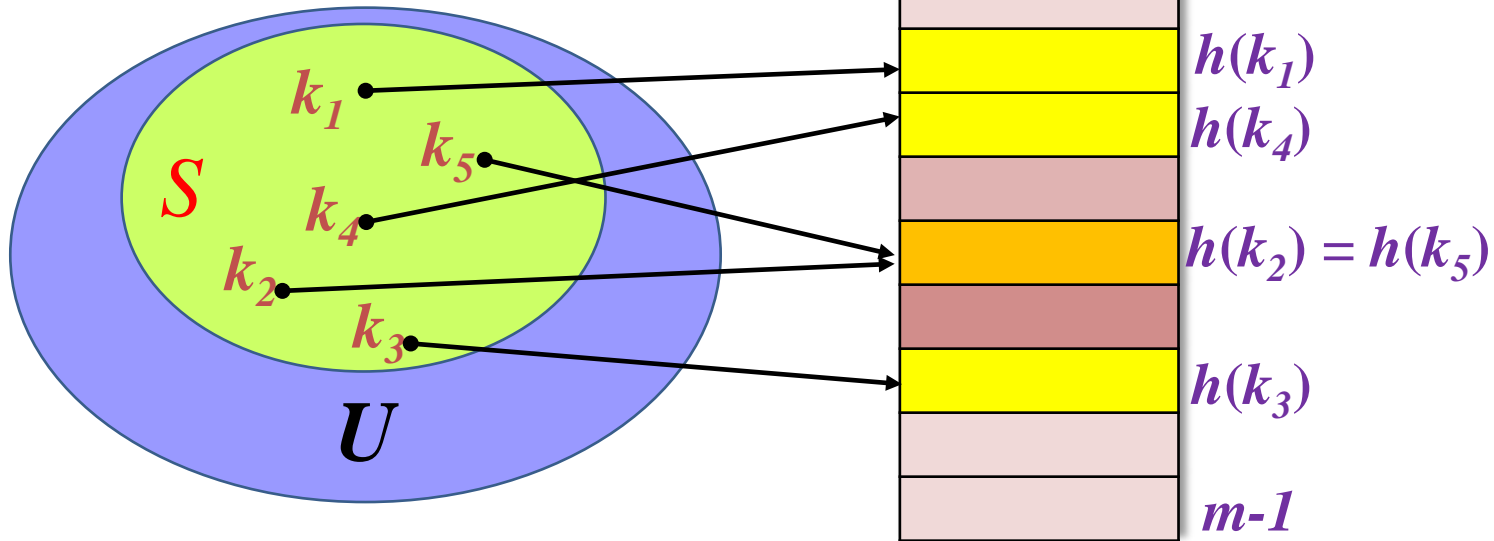
Direct-access table

Problem:

1. 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!)
2. The set K of keys actually stored maybe $\ll U$,
Most of the space allocated for T is wasted.

Hash Tables

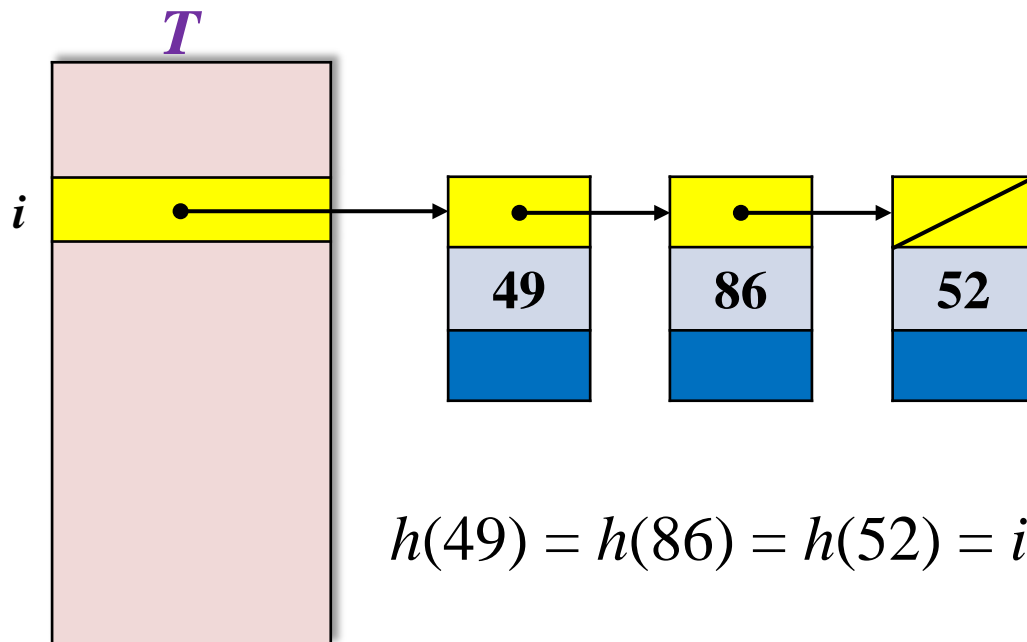
Solution: Use a *hash function* h to map the universe U of all keys into $\{0, 1, \dots, m-1\}$:



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.



$$h(49) = h(86) = h(52) = i$$

Worst case:

- Every key hashes to the same slot.
- Access time:
 $\Theta(n)$ if $|S| = n$

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

$$\alpha = n/m$$

= 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

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

*apply hash function
and access slot*

*search
the list*

Search cost

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

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

*apply hash function
and access slot*


*search
the list*

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

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



*apply hash function
and access slot*

*search
the list*

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.

Desirata:

- 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.

Division method

Assume all keys are integers, and define

$$h(k) = k \bmod m.$$

Deficiency: Don't pick an m that has a small remainder 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 $k = 1011000111\underbrace{011010}_2$ and $r = 6$ then
 $h(k) = 011010_2$.

Division method (continued)

$$h(k) = k \bmod m.$$

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.

Division method (continued)

- Example
 - If $n=2000$, with 8-bit keys. The desired number of searched keys in an unsuccessful search is about 3.

Division method (continued)

- Example
 - If $n=2000$, with 8-bit keys. The desired number of searched keys in an unsuccessful search is about 3.
 - $m=701$
 - Approximately $=2000/3$
 - A prime not close to any 2^r
 - $h(k)=k \bmod 701$

Multiplication method

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

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

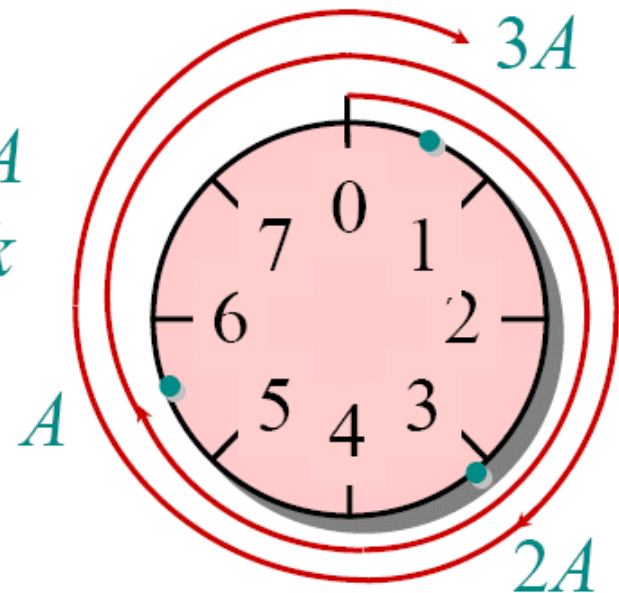
where **rsh** is the “bitwise right-shift” operator and A is an odd integer in the range $2^{w-1} < A < 2^w$.

- 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

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

$$\begin{array}{r}
 1011001 = A \\
 \times 1101011 = k \\
 \hline
 10010100 \\
 \underbrace{0110011}_{h(k)} \\
 \hline
 110011
 \end{array}$$



Modular wheel

Resolving collisions by open addressing

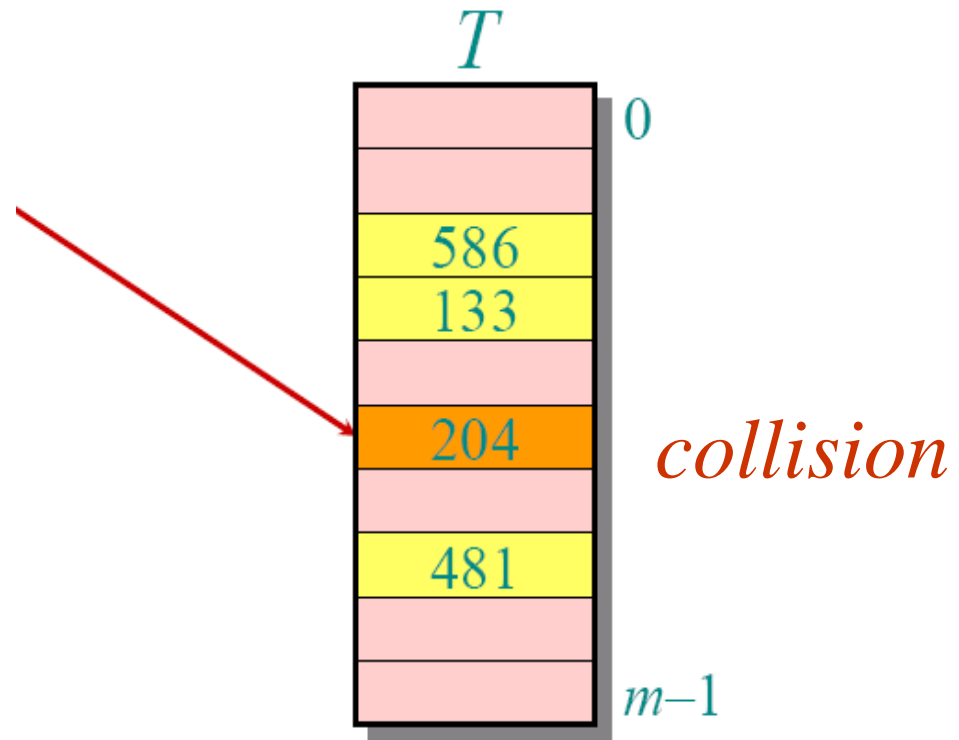
No storage is used outside of the hash table itself.

- Save pointers for a larger number of slots
- The hash function depends on both the key and probe number:
$$h: U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}.$$
- The probe sequence $\langle h(k,0), h(k,1), \dots, h(k,m-1) \rangle$ should be a permutation of $\{0, 1, \dots, m-1\}$.
- Insertion systematically probes the table until an empty slot is found.

Example of open addressing

Insert key $k=496$:

0. Probe $h(496,0)$

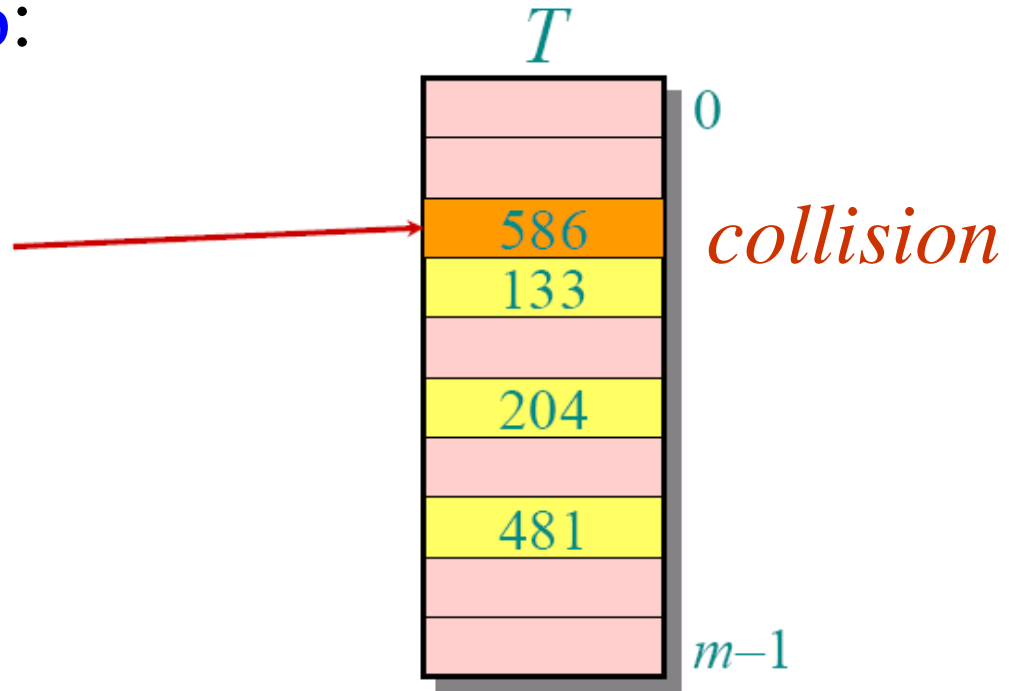


Example of open addressing

Insert key $k = 496$:

0. Probe $h(496, 0)$

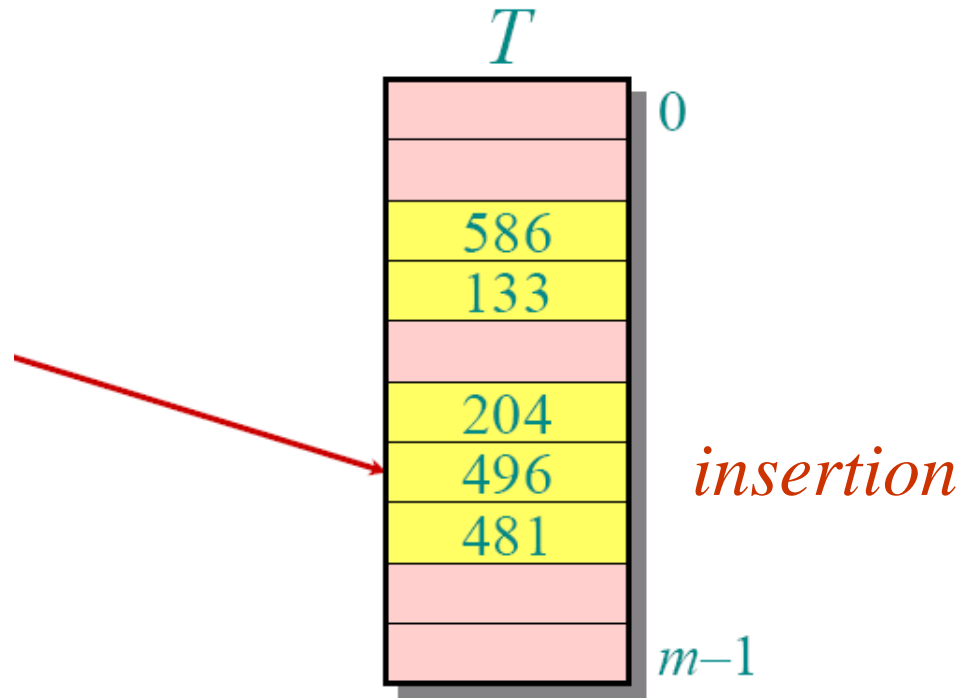
1. Probe $h(496, 1)$



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 $k=496$:

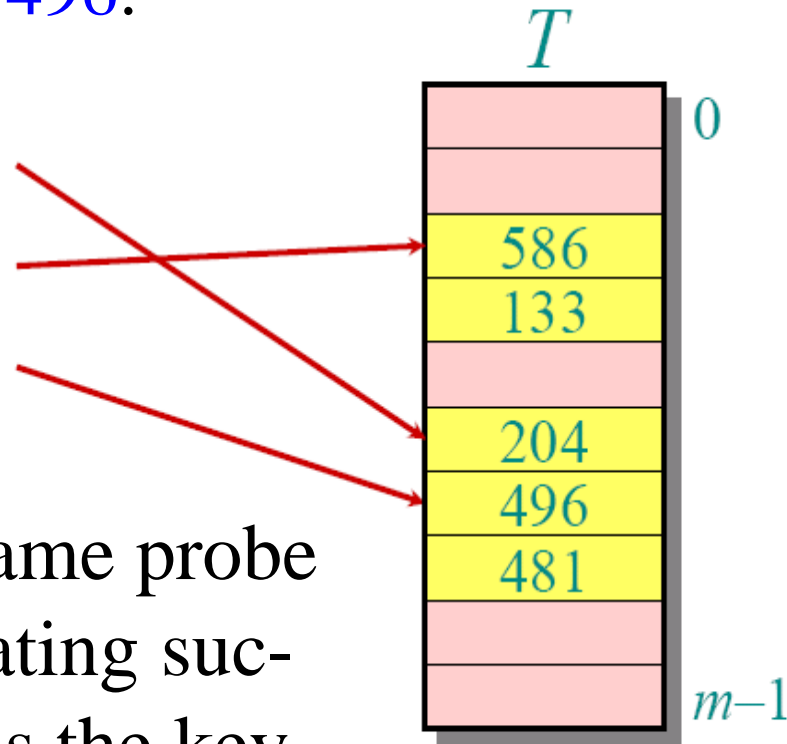
0. Probe $h(496,0)$

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

- The assumption of uniform hashing:
- Each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \dots, m-1 \rangle$ as its probe sequence.
- Probing functions:
 - Must guarantee the $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ is a permutation of $\langle 0, 1, \dots, m-1 \rangle$ for each key.
 - Try to fulfill the assumption of uniform hashing

Probing strategies

Linear probing:

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

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

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.

Probing strategies

Quadratic probing:

Quadratic probing uses the hash function

$$h(k,i) = (h'(k) + c_1i + c_2i^2) \bmod m.$$

Where $h'(k)$ is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants.

To make full use of the hash table, the values of c_1 , c_2 , and m are constrained.

Much better than linear probing, still suffers from a milder form of clustering, called

Secondary clustering.

Probing strategies

Double hashing:

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

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m.$$

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.

Probing strategies

- Linear probing
 - m different probe sequence
- Quadratic probing
 - m different probe sequence
- Double hash
 - m^2 , close to the “ideal”

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.

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 $1/(1-\alpha)$.

Proof of the theorem

Proof.

- At least one probe is always necessary.
- With probability n/m , the first probe hits an occupied slot, and a second probe is necessary.
- With probability $(n-1)/(m-1)$, the second probe hits an occupied slot, and a third probe is necessary.
- With probability $(n-2)/(m-2)$, the third probe hits an occupied slot, etc.
- Observe that $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$ for $i = 1, 2, \dots, n$.

Proof (continued)

Therefore, the expected number of probes is

$$\begin{aligned} & 1 + \frac{n}{m} \left(1 + \frac{n-1}{m-1} \left(1 + \frac{n-2}{m-2} \left(\dots \left(1 + \frac{1}{m-n+1} \right) \dots \right) \right) \right) \\ & \leq 1 + \alpha (1 + \alpha (1 + \alpha (\dots (1 + \alpha) \dots))) \\ & \leq 1 + \alpha + \alpha^2 + \alpha^3 + \dots \\ & = \sum_{i=0}^{\infty} \alpha^i \\ & = \frac{1}{1-\alpha} . \quad \square \end{aligned}$$

The textbook has a more rigorous proof and an analysis of successful searches.

Analysis of open addressing

- **Corollary** Inserting an element into an open-address hash table with load factor α requires at most $1/(1-\alpha)$ probes on average, assuming uniform hashing.

Implications of the theorem

- If α 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 $1/(1-0.9) = 10$.

Consider using open addressing to insert keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m=11$, the auxiliary hash function is $h'(k)=k$. The procedure of inserting these keys into the hash table using linear probe is shown as follows.

Consider using open addressing to insert keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m=11$, the auxiliary hash function is $h'(k)=k$. The procedure of inserting these keys into the hash table using linear probe is shown as follows.

$h(10,0)=10$

										10
--	--	--	--	--	--	--	--	--	--	----

$h(22,0)=0$

22										10
----	--	--	--	--	--	--	--	--	--	----

$h(31,0)=9$

22									31	10
----	--	--	--	--	--	--	--	--	----	----

$h(4,0)=4$

22				4					31	10
----	--	--	--	---	--	--	--	--	----	----

$h(15,0)=4$
conflict!

$h(15,1)=5$

22				4	15				31	10
----	--	--	--	---	----	--	--	--	----	----

Consider using open addressing to insert keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m=11$, the auxiliary hash function is $h'(k)=k$. The procedure of inserting these keys into the hash table using linear probe is shown as follows.

$h(28,0)=6$

22				4	15	28			31	10
----	--	--	--	---	----	----	--	--	----	----

$h(17,0)=6$
conflict !

22				4	15	28			31	10
----	--	--	--	---	----	----	--	--	----	----

$h(17,1)=7$

22				4	15	28	17		31	10
----	--	--	--	---	----	----	----	--	----	----

$h(88,0)=0$
conflict !

22				4	15	28	17		31	10
----	--	--	--	---	----	----	----	--	----	----

$h(88,1)=1$

22	88			4	15	28	17		31	10
----	----	--	--	---	----	----	----	--	----	----

$h(59,0)=4$
conflict !

22	88			4	15	28	17		31	10
----	----	--	--	---	----	----	----	--	----	----

$h(59,1)=5$
conflict !

22	88			4	15	28	17		31	10
----	----	--	--	---	----	----	----	--	----	----

Consider using open addressing to insert keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m=11$, the auxiliary hash function is $h'(k)=k$. The procedure of inserting these keys into the hash table using linear probe is shown as follows.

$$h(59,2)=6$$

conflict !

22	88			4	15	28	17		31	10
----	----	--	--	---	----	----	----	--	----	----

$$h(59,3)=7$$

conflict !

22	88			4	15	28	17		31	10
----	----	--	--	---	----	----	----	--	----	----

$$h(59,4)=8$$

22	88			4	15	28	17	59	31	10
----	----	--	--	---	----	----	----	----	----	----