COT 6405 Introduction to Theory of Algorithms

Topic 12. Hash tables

Data Structures

- We focus on data structures in this part
  - stack, linked list, queue, tree, pointer, object, …
- In particular, structures for dynamic sets
  - Elements have a key and satellite data
  - Dynamic sets support queries such as:
    - Search(S, k), Minimum(S), Maximum(S), Successor(S, k), Predecessor(S, k)
  - They may also support modifying operations like:
    - Insert(S, k), Delete(S, k)

Dictionary

- Dictionary
  - is a Dynamic-set data structure for storing items indexed using keys
  - Supports operations: Insert, Search, and Delete
- Hash Tables:
  - Effective way of implementing dictionaries

Types of Dictionaries

- A dictionary consists of key-element pairs in which the key is used to look up the element
- Ordered Dictionary: Elements stored in sorted order by key
- Unordered Dictionary: Elements not stored in sorted order

Example	Key	Element
English Dictionary	Word	Definition
Student Records	Student ID	Rest of record: Name, …
Symbol Table in Compiler	Variable Name	Variable’s Address in Memory
Lottery Tickets	Ticket Number	Name & Phone Number

Dictionary as a Function

- Given a key, return an element

  Key $\longrightarrow$ Element

  (domain: (range:

  type of the keys) type of the elements)

- A dictionary is a partial function. Why?
  - A function which is not defined for some of its domain. (key is not defined)
  - ‘kk’ → not defined in English dictionary

Direct-address Tables

- Direct-address Tables are ordinary arrays
- Facilitate direct addressing
  - Element whose key is k is obtained by indexing into the k-th position of the array, e.g., A[k]
- Applicable when we can afford to allocate an array with one position for every possible key
  - i.e. when the universe of keys *U* is small.
- Dictionary operations can be implemented to take O(1) time.

Direct-address Tables

Direct-Address-Search(T, k)
return T[k]

Direct-Address-Insert(T, x)
T[x.key] 🡐 x

Direct-Address-Delete(T, x)
T[x.key] 🡐 NIL

Time Analysis: O(1)

Space Analysis: ?

Direct-address Tables

Direct-Address-Search(T, k)
return T[k]

Direct-Address-Insert(T, x)
T[x.key] 🡐 x

Direct-Address-Delete(T, x)
T[x.key] 🡐 NIL

Time Analysis: O(1)

Space Analysis: O(|U|)

Dynamic set by Direct-address

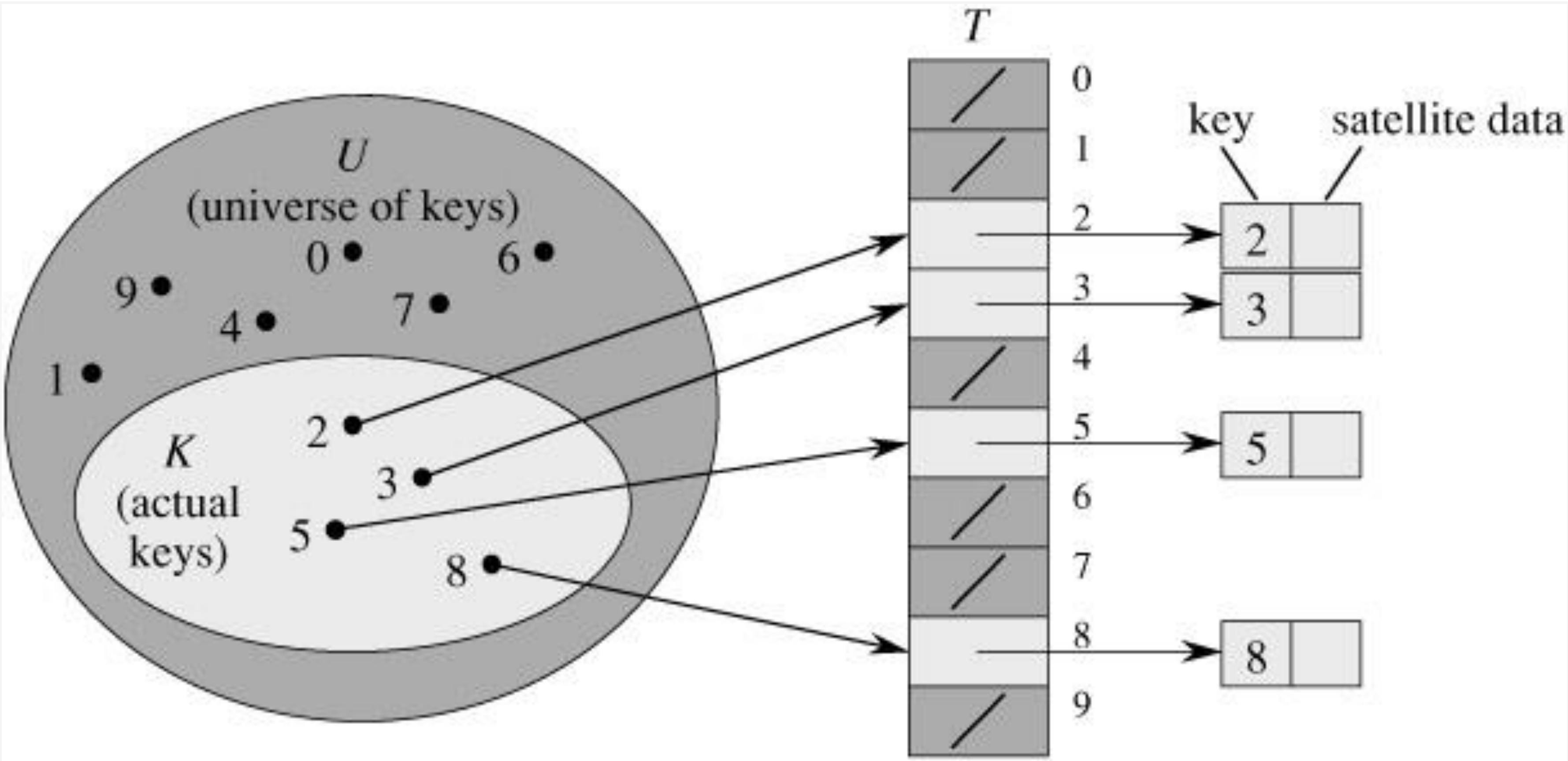


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

The Drawback of Direct-addressing

- Notation:
  - U is the Universe of all possible keys.
  - K is the set of keys actually stored in the dictionary.
  - $|K| = n$
- When U is very large, $|K| << |U|$
  - Arrays are not practical

Hash Table

- We use a table of size proportional to |K|: hash tables
  - Define hash functions that map keys to slots of the hash table.
  - However, we lose the direct-addressing ability.

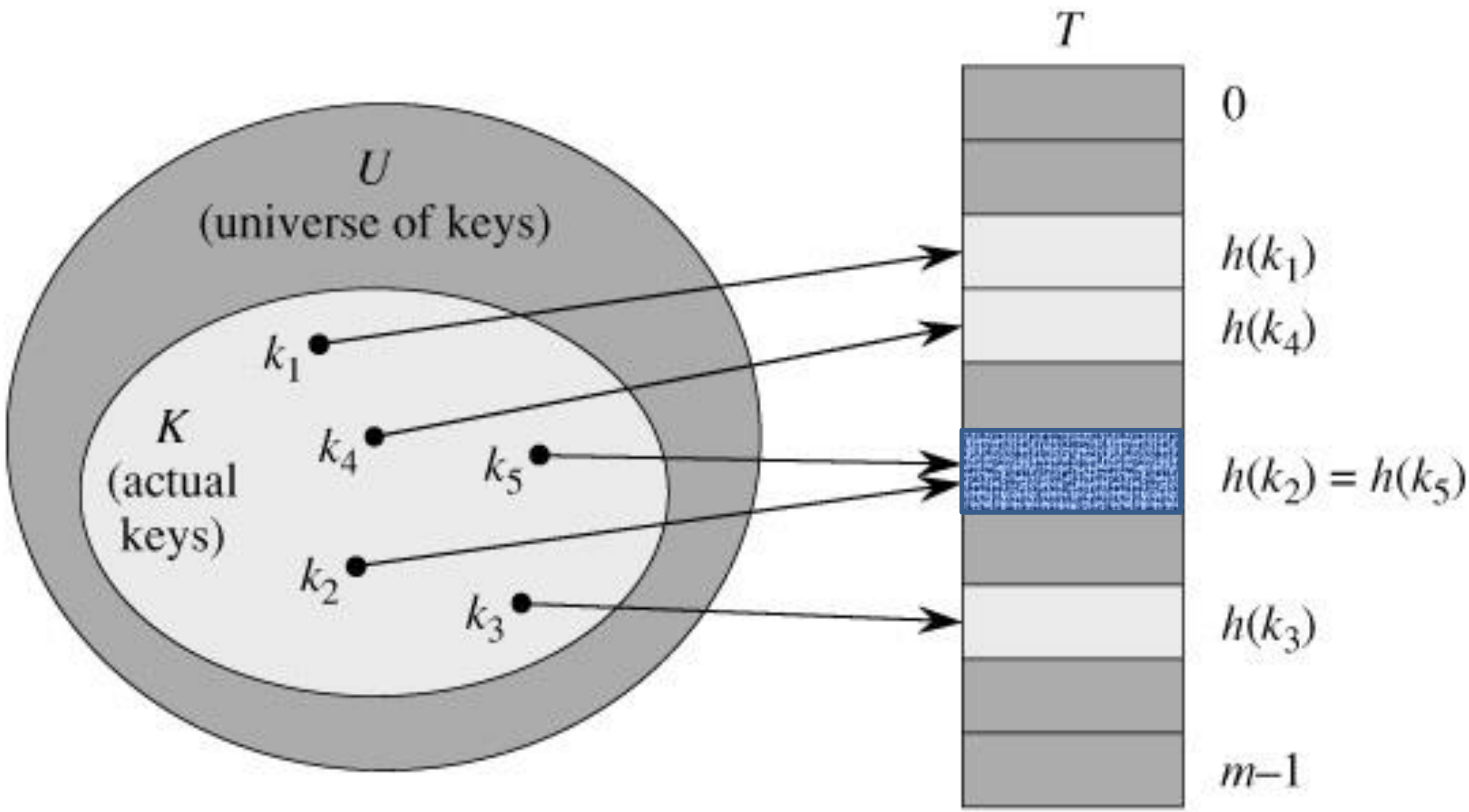
Hash function

- Hash function h: Mapping from Universe U to the slots of a hash table T[0..m–1].
- h : U → {0,1,..., m–1}
  - With arrays, key k maps to slot A[k].
  - With hash tables, key k maps or "hashes" to slot T[h(k)]
    - h(k) is the hash value of key k
- Example of Hash Function
  - h(k) = return (k mod m)
  - where k is the key, and m is the size of the table

Issues with Hashing?

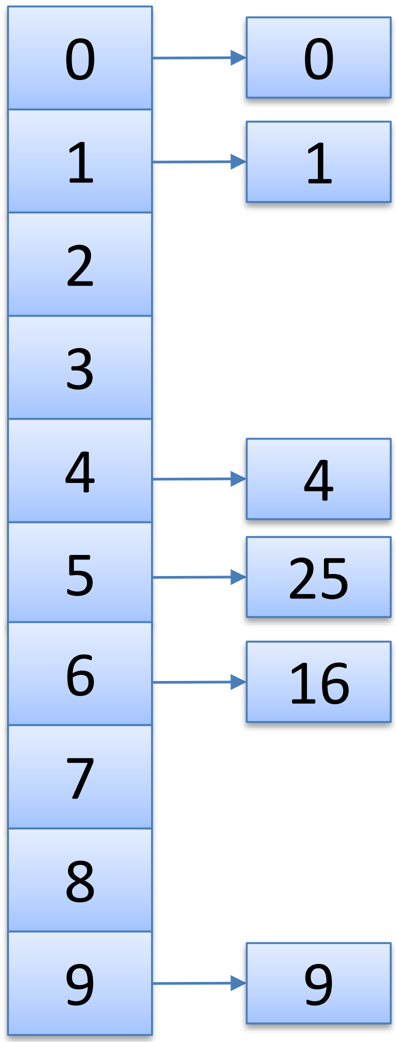
- Multiple keys can hash to the same slot: collisions
  - Design hash functions such that collisions are minimized
  - But avoiding collisions is sometimes impossible
    - Must have collision-resolution techniques

Hash Table with Collision



Collision Resolution Scheme 1: Chaining

- The hash table is an array of linked lists
- Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81



Notes:

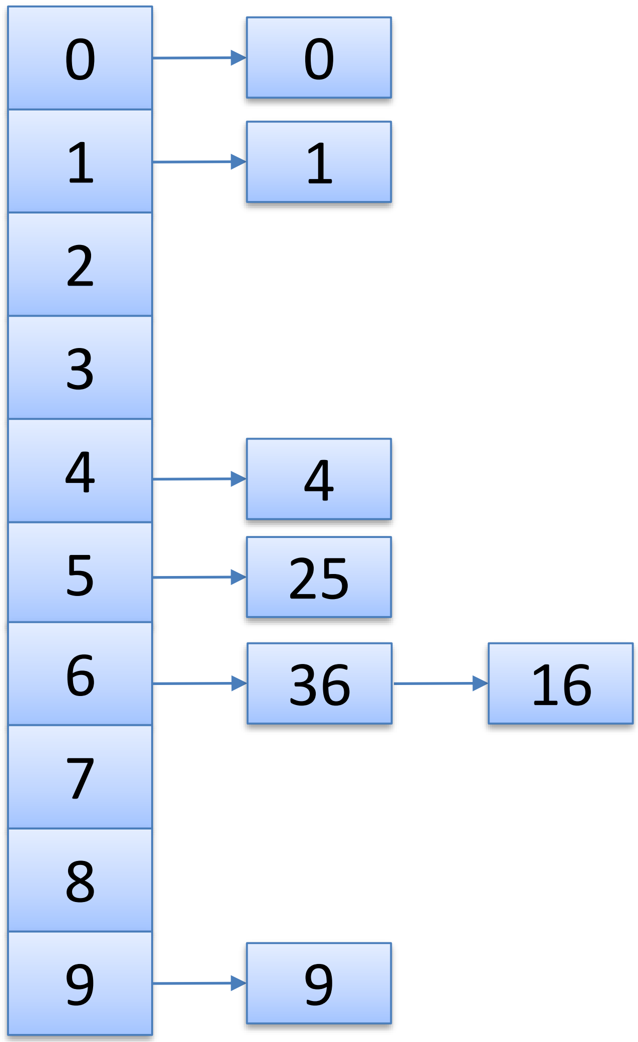
- As before, elements would be associated with the keys
- We're using the hash function h(k) = k mod m
  - m=10

Collision Resolution Scheme 1: Chaining

- The hash table is an array of linked lists
- Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Notes:

- As before, elements would be associated with the keys
- We're using the hash function h(k) = k mod m
  - m=10

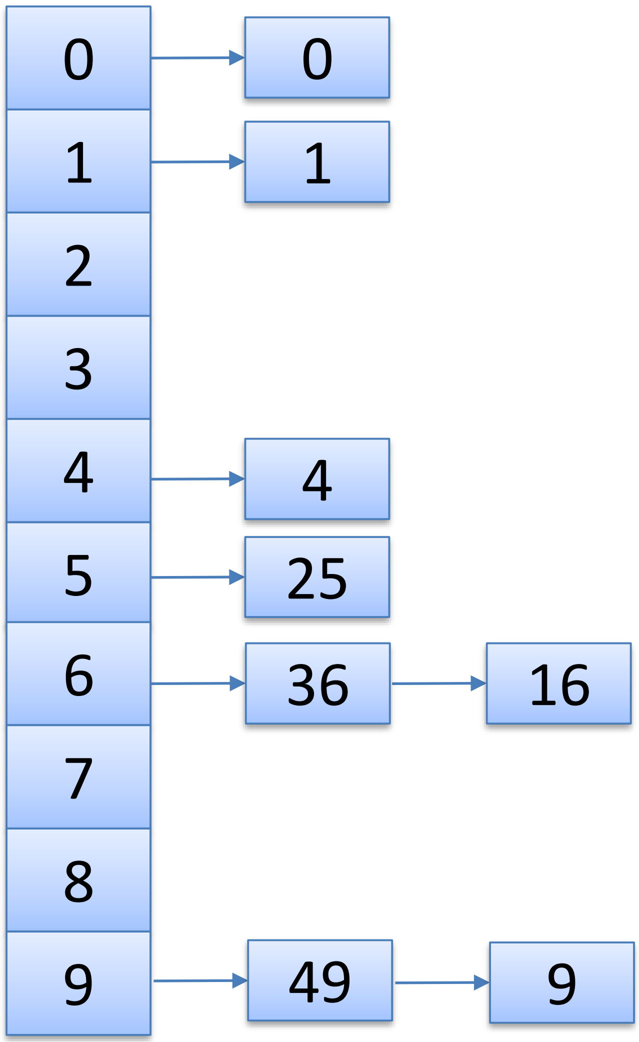


Collision Resolution Scheme 1: Chaining

- The hash table is an array of linked lists
- Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Notes:

- As before, elements would be associated with the keys
- We're using the hash function h(k) = k mod m
  - m=10

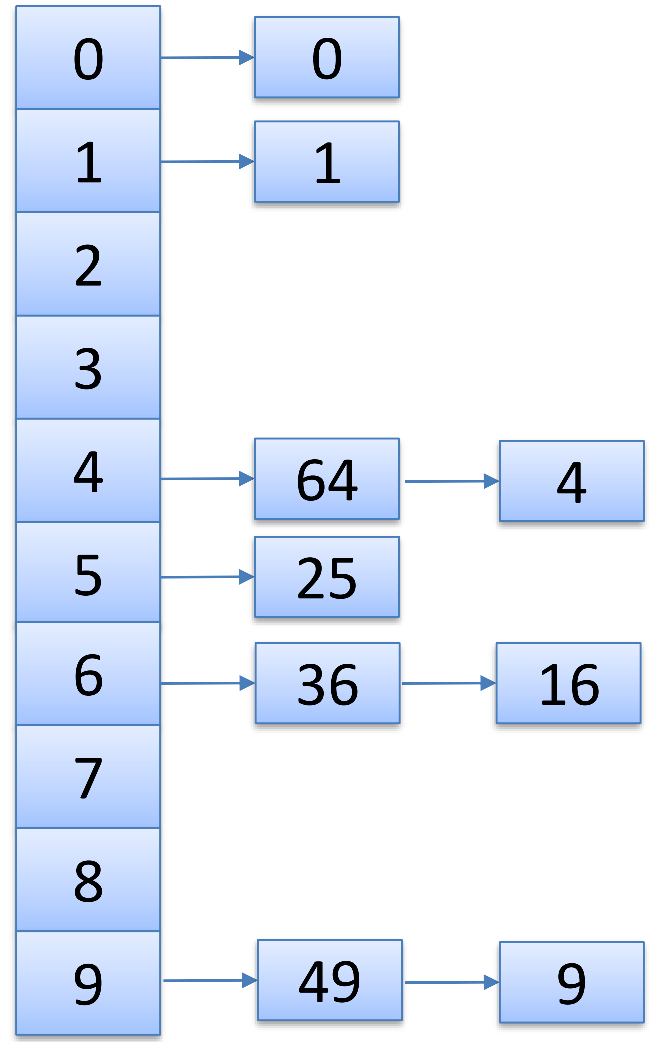


Collision Resolution Scheme 1: Chaining

- The hash table is an array of linked lists
- Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Notes:

- As before, elements would be associated with the keys
- We're using the hash function h(k) = k mod m
  - m=10

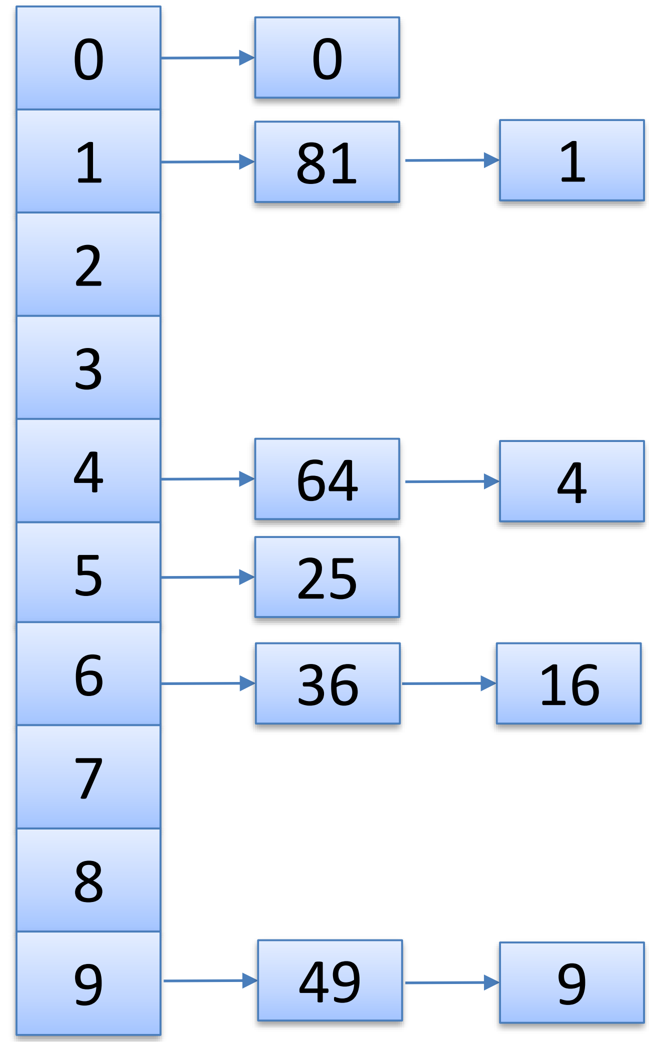


Collision Resolution Scheme 1: Chaining

- The hash table is an array of linked lists
- Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Notes:

- As before, elements would be associated with the keys
- We're using the hash function h(k) = k mod m
  - m=10



Chaining Algorithms

Chained-Hash-Insert(T, x)

insert x at the head of list T[h(x.key)]

Chained-Hash-Search(T, k)

search for an element with key k

in list T[h(k)]

Chained-Hash-Delete(T, x)

delete x from the list T[h(x.key)]

Analysis of hashing with chaining

- m = hash table size
- n = number of elements in hash table
- load factor α= n/m : average number of keys per slot
- Assume each key is equally likely to be hashed into any slot: using simple uniform hashing (SUH)
- What is the worst-case search time?
  - Unsuccessful Search → we find none
  - Successful Search → we find one

Expected time of an unsuccessful search

Theorem: In a hash table in which collisions are resolved by chaining, an unsuccessful search takes expected time $\Theta(1+\alpha)$ under SUH.

Proof:

- Under the assumption of SUH, any key 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 is exactly α.
- Consider compute the hash as O(1)
- Thus, the total time required is $\Theta(1+\alpha)$

Expected time of a successful search

Theorem: In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1+\alpha)$, on the average under SUH.

Proof: The number of elements examined during a successful search for an element x is one more than the number of elements that appear before x in x's list. (why?)

Proof (cont'd)

- To find the expected number of elements examined, we take the average, over the n elements x in the table, of 1 plus the expected number of elements added to x's list after x was added to the list.

Proof (cont'd)

- Let x_i denote the i-th element into the table, for i =1 to n, and let $k_i = x_i$.key
- Define $X_{ij} = I\{ h(k_i)=h(k_j) \}$. Under SUH, we have $\Pr\{ h(k_i)=h(k_j) \} = 1/m = E[X_{ij}]$ (why?)

Proof (cont'd)

- Hence, $\sum_{j=2}^{n} X_{1j}$ is the number of elements that are inserted before x_1, and 1+$\sum_{j=2}^{n} X_{1j}$ is the number of elements that are searched to find x_1, which is the first element into the table
- 1+$\sum_{j=3}^{n} X_{2j}$ is the number of elements that are searched to find x_2, which is the second element into the table

Proof (cont'd)

- A total of n elements are inserted into the hash table, and the average number of elements examined is equal to

$$\frac{1}{n}((1 + \sum_{j=2}^{n} X_{1j}) + (1 + \sum_{j=3}^{n} X_{2j}) + \ldots (1 + \sum_{j=n+1}^{n} X_{nj}))$$

$$= \frac{1}{n}(\sum_{i=1}^{n}(1 + \sum_{j=i+1}^{n} X_{ij}))$$

Proof (cont'd)

$$E[\frac{1}{\mathrm{n}}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})] = \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}])$$

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

$$= 1+\frac{1}{mn}(n^2-\frac{n(n+1)}{2}) = 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

$$\Theta(2+\frac{\alpha}{2}-\frac{\alpha}{2n}) = \Theta(1+\alpha).$$

W

Collision Resolution Scheme 2: Open addressing

- No list and no element stored outside the table
  - If a collision occurs, try alternate cells until empty cell is found.
  - Pro: No pointers!
- Advantage: avoid pointers, potentially yield fewer collisions and faster retrieval
  - Extra memory freed from storing pointers → more hash slots → less collisions!

Common Probing Sequence

- Assume uniform hashing
- Collision Resolution Strategies for open address
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
- We try cells $h(k,0)$, $h(k,1)$, $h(k,2)$, …, $h(k, m-1)$
  - where $h(k,i) = (h'(k) + f(i)) \bmod m$, with $f(0) = 0$
  - Function f is the collision resolution strategy
  - Function h' is the original hash function.

Probe sequence

– Given function h() $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$

– For every k, the probe sequence

$$\langle h(k,0), h(k,1), \ldots, h(k, m-1) \rangle$$

is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$

- A sequence of m slots

– How about deletion?

– Deletion from an open-address hash table is difficult

- We can NOT simply mark one cell is empty!
- Thus chaining is more common when keys must be deleted.

Open addressing insertion

```
Hash-Insert ( T, k )
    i ← 0
    repeat
        j ← h( k, i )
        if T[ j ] == NIL
            then T[ j ] ← k
                    return j
            else  i ← i + 1
    until i = m
    error "hash table overflow"
```

Open addressing search

```
Hash-Search( T, k )
        i ← 0
        repeat
                j ← h( k, i )
                if T[ j ] == k
                    then return j
                i ← i + 1
        until T[ j ] = NIL or i = m
        return NIL
```

Linear Probing

- Function f is linear, e.g., f(i) = i
- h(k, i) = (h′(k) + i) mod m
  - Offsets: 0, 1, 2, …, m-1
  - Only probe m slots
- With H = h′(k), we try the following cells with wraparound:

  H, H + 1, H + 2, H + 3, …

- What does the table look like after the following insertions? (assume h’(k) = k mod m)
- Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Linear Probing

- Function f is linear, e.g., f(i) = i
- h(k, i) = (h′(k) + i) mod m
  - Offsets: 0, 1, 2, …, m-1
  - Only probe m slots
- With H = h′(k), we try the following cells with wraparound:

  H, H + 1, H + 2, H + 3, …
- What does the table look like after the following insertions? (assume h’(k) = k mod m)
- Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

0	0
1	1
2	49
3	81
4	4
5	25
6	16
7	36
8	64
9	9

Issue: Primary Clustering

- Linear Probing is easy to implement, but it suffers from the problem of primary clustering
  - i.e., the tendency to create long sequences of filled slots
- If two keys have the same initial probe position, then their probe sequences are the same.
- As more elements are inserted into the hash table, the probing sequences get longer
  - Consequently, the average search time increases
  - O(1) to O(m)

Collision Resolution Comparison

	Advantages?	Disadvantages?
Chaining	O(1+⍰) insertion, O(1+⍰) deletion	pointers
Linear Probing	no pointers	primary clustering

Quadratic Probing

- Function f is quadratic: $f(i) = i^2$
- $h(k, i) = (h'(k) + i^2) \bmod m$
  - Offsets: 0, 1, 4, 9, …
- With $H = h'(k)$, we try the following cells with wraparound:
- $H, H + 1^2, H + 2^2, H + 3^2$ …
  - A sequence of m slots
- Insert Keys: 10, 23, 14, 9, 16, 25, 36, 44, 33

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Quadratic Probing

- Function f is quadratic: $f(i) = i^2$
- $h(k, i) = (h'(k) + i^2) \bmod m$
  - Offsets: 0, 1, 4, 9, …
- With $H = h'(k)$, we try the following cells with wraparound:
- $H, H + 1^2, H + 2^2, H + 3^2 \ldots$
  - A sequence of m slots
- Insert Keys: 10, 23, 14, 9, 16, 25, 36, 44, 33

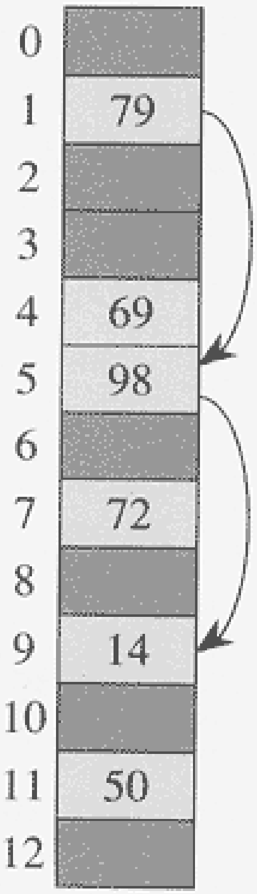
0	10
1	
2	33
3	23
4	14
5	25
6	16
7	36
8	44
9	9

Secondary Clustering

- Quadratic Probing suffers from a milder form of clustering called secondary clustering
- As with linear probing, if two keys have the same initial probe position, then their probe sequences are the same
  - since $h(k_1,0) = h(k_2,0)$ implies $h(k_1,i) = h(k_2,i)$.
- Therefore, clustering can occur around the probe sequences.

Double Hashing

- If a collision occurs when inserting, apply a second auxiliary hash function, $h_2(k)$
  - We then probe at a distance: $h_2(k)$, $2 * h_2(k)$, $3 * h_2(k)$, etc., until find empty position.
- So, $f(i) = i * h_2(k)$, and we have two auxiliary functions:
  - $h(k, i) = (h_1(k) + i * h_2(k)) \bmod m$
- With $H = h_1(k)$, we try the following cells in sequence with wraparound:
  - H, $H + 1 * h_2(k)$, $H + 2 * h_2(k)$, $H + 3 * h_2(k)$



- $h(k, i) = (h_1(k) + i * h_2(k)) \bmod m$

- $h(14,0) = (14 \bmod 13 + 0) \bmod 13 = 1$
- $h(14,1) = (14 \bmod 13 +$
  $1 * (1+14 \bmod 11)) \bmod 13 = 5$
- $h(14,2) = (14 \bmod 13 +$
  $2 * (1+14 \bmod 11)) \bmod 13 = 9$

Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Double Hashing

- $h(k_1,0) = h(k_2,0)$, $h(k_1,i) \neq h(k_2,i)$,
  - $h(k_1,i) = (h_1(k_1) + i*h_2(k_1))$ mod m
  - $h(k_2,i) = (h_1(k_2) + i*h_2(k_2))$ mod m
  - Even if the initial probe of k_1 is equal to that of k_2, their following probes are random and not the same.
- It is one of the best methods available for open addressing, because the produced permutations are close to randomly chosen permutations. Doesn't suffer from primary or secondary clustering

Analysis of open-addressing hashing

- m = hash table size
- n = number of elements in hash table
- load factor α= n/m : average number of keys per slot
- Theorem: Given an open-address hash table with load factor α= n/m < 1, the expected number of probes in an unsuccessful search is at most 1/(1-α), assuming uniform hashing.
  – unsuccessful search →every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty.

Proof

- Define random variable X to be the number of probes made in an unsuccessful search.
- Define A_i: the event that there is an i-th probe and it is to an occupied slot.
- Then, the event $\{X \geq i\}$ is the intersection of

$$\{X \geq i\} = A_1 \cap A_2 \cap \quad \cap A_{i-1}$$

$$\begin{aligned}\Pr\{X \geq i\} &= \Pr\{A_1 \cap A_2 \cap \quad \cap A_{i-1}\} \\ &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdot \\ &\quad \cdot \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \quad \cap A_{i-2}\}\end{aligned}$$

$$\Pr\{A_1\} = \frac{n}{m}$$

Proof (Cont'd)

- Given that the first i-1 probes were to occupied slots
- n-(i-1) occupied elements in the hash table haven't been probed and there are a total of m-(i-1) slots to be explored
- The probability that there is a i-th probe to an occupied slot is (n-(i-1))/(m-(i-1))

$$\Pr[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \quad \frac{n-i+2}{m-i+2}$$

$$\leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

$n<m$, $(n-i)/(m-i) \leq n/m$ for all $0 \leq i < n$.

$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$$

Easy to estimate

- Load factor α= 0.5
- We need 1/(1-0.5) = 2 probes on average for unsuccessful search
- Load factor α= 0.9
- We need 1/(1-0.9) = 10 probes on average for unsuccessful search

Corollary

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

- Proof
  - We first find the empty slot via an unsuccessful search
  - Then insert the key
  - The expected number of probes is at most 1/(1- α)

Proof (cont'd)

- Theorem: Given an open-address hash table with load factor α<1, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

  – assuming uniform hashing
  – each key in the table is equally likely to be searched for.

Proof (Cont’d)

- Suppose we search for a key k.
  - If k is the (i+1)-st key inserted into the hash table, at the time when inserted k, i slots in the hash table had been already occupied,
  - The corresponding load factor α_i is i/m
  - According to the Corollary, Inserting k into the hash table with load factor α_i requires at most $1/(1- \alpha_i)$ probes on average,
  - A search for a key *k* follows the same probe sequence as was followed when k was inserted. Thus, the expected number of probes made in a search for k is at most $1/(1-\alpha_i) = 1/(1-i/m) = m/(m-i)$

Proof (Cont'd)

Averaging over all n keys in the hash table gives us the average number of probes in a successful search

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}$$

$$=\frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k}\leq\frac{1}{\alpha}\int_{m-n}^{m}\frac{dx}{x}=\frac{1}{\alpha}\ln\frac{m}{m-n}=\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

Changing variable: let $k = m - i$ -> $i = m-k$.
i ranges between 0 and n-1, and thus m-k ranges between 0 and n-1, and k ranges between m and m-n+1

Collision Resolution Comparison:

Expected Number of Probes in Searches

load factor $\alpha = n/m$

	Unsuccessful Search	Successful Search
Chaining	$1+\alpha$ (1 + average number of elements in chain)	$1 + \alpha/2 - \alpha/(2n)$ (1 + average number before element in chain)
Open Addressing (assuming uniform hashing)	$1 / (1 - \alpha)$	$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$