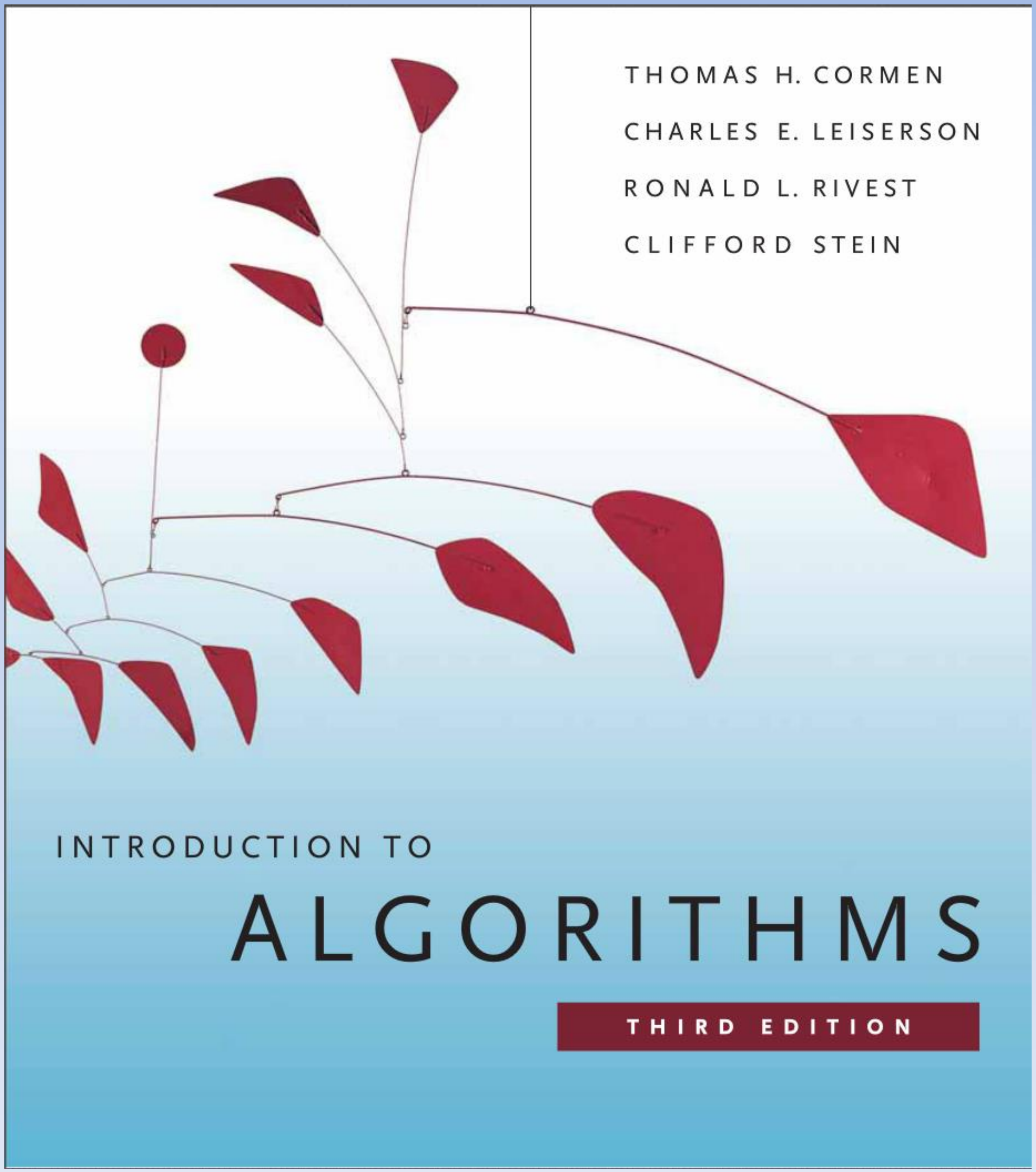
174 : Lecture 13 w/ Chapter 11



Hashing!

Contents

Chapter 11

III Data Structures

Dictionary Operations

- Insert: Insert item
- Search: Find item
- Delete: Delete an item

- Hash Tables are great w/ Dictionaries

- Look @ Analysis of Hashing!
- First : Revisit Hashing!
  - If you've seen it before (in another context), Then the Text description should flow more easily
  - Opportunity to measure current understanding

Dictionary w/ Python

```
>>> OED = {}
>>> OED["Algorithm"]="2. Math. and Computing. A procedure or set of rules used i
n calculation and problem-solving; (in later use spec.) a precisely defined set
of mathematical or logical operations for the performance of a particular task."
```

- Insert item: O(1)

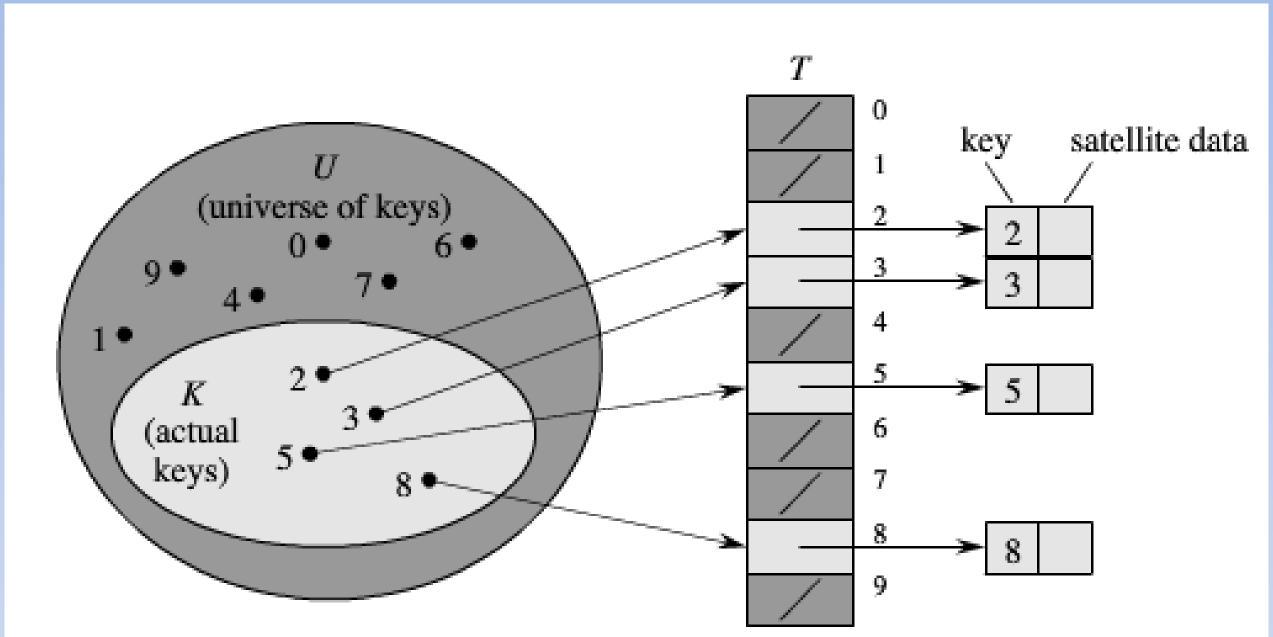
```
>>> print OED["Algorithm"]
2. Math. and Computing. A procedure or set of rules used in calculation and prob
lem-solving; (in later use spec.) a precisely defined set of mathematical or log
ical operations for the performance of a particular task.
```

- Find item: O(1)

Direct Addressing versus Hashing

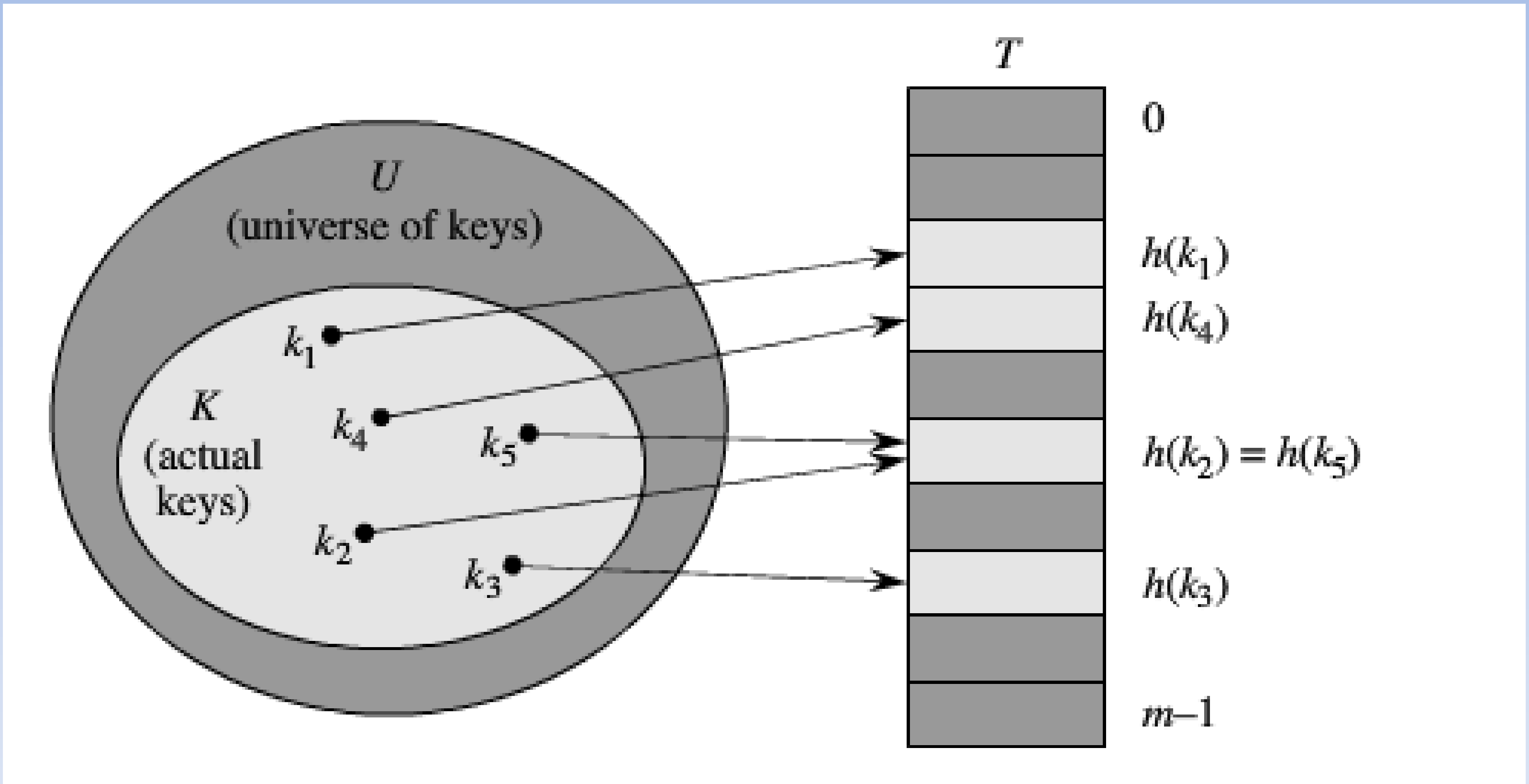
- Ordinary array uses direct addressing:
  - Offsetting by an integer index into a fixed length area of memory.
- Hash Table generalizes direct addressing:
  - Key range is large relative to the number of keys stored.
  - Hash tables use space proportional to the number of keys (not key range).
- Perfect Hashing can support searches in O(1) worst-case time
  - where the set of keys being stored is static

Direct Addressing

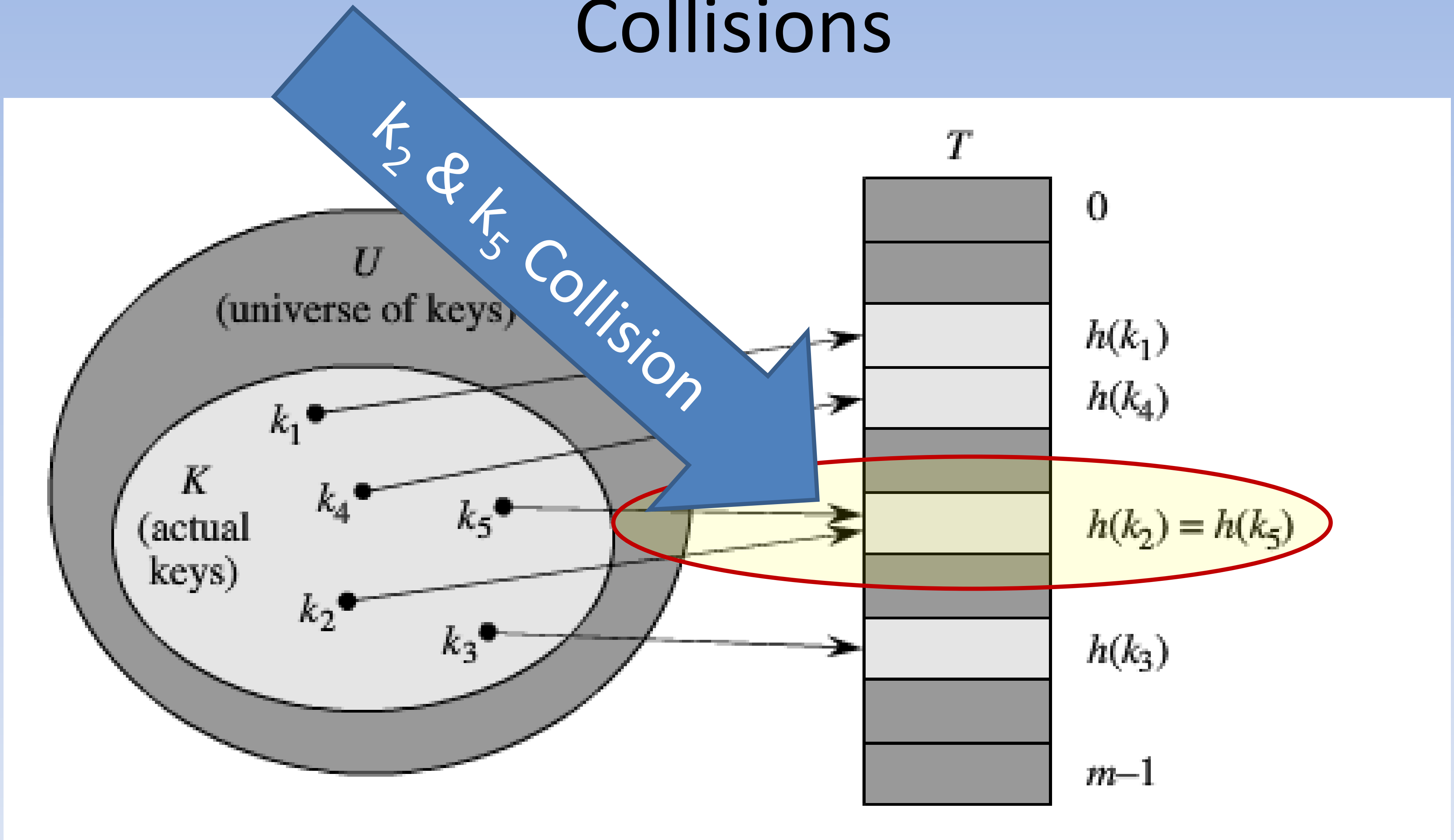


- Search, Insert, Delete: O(1)!

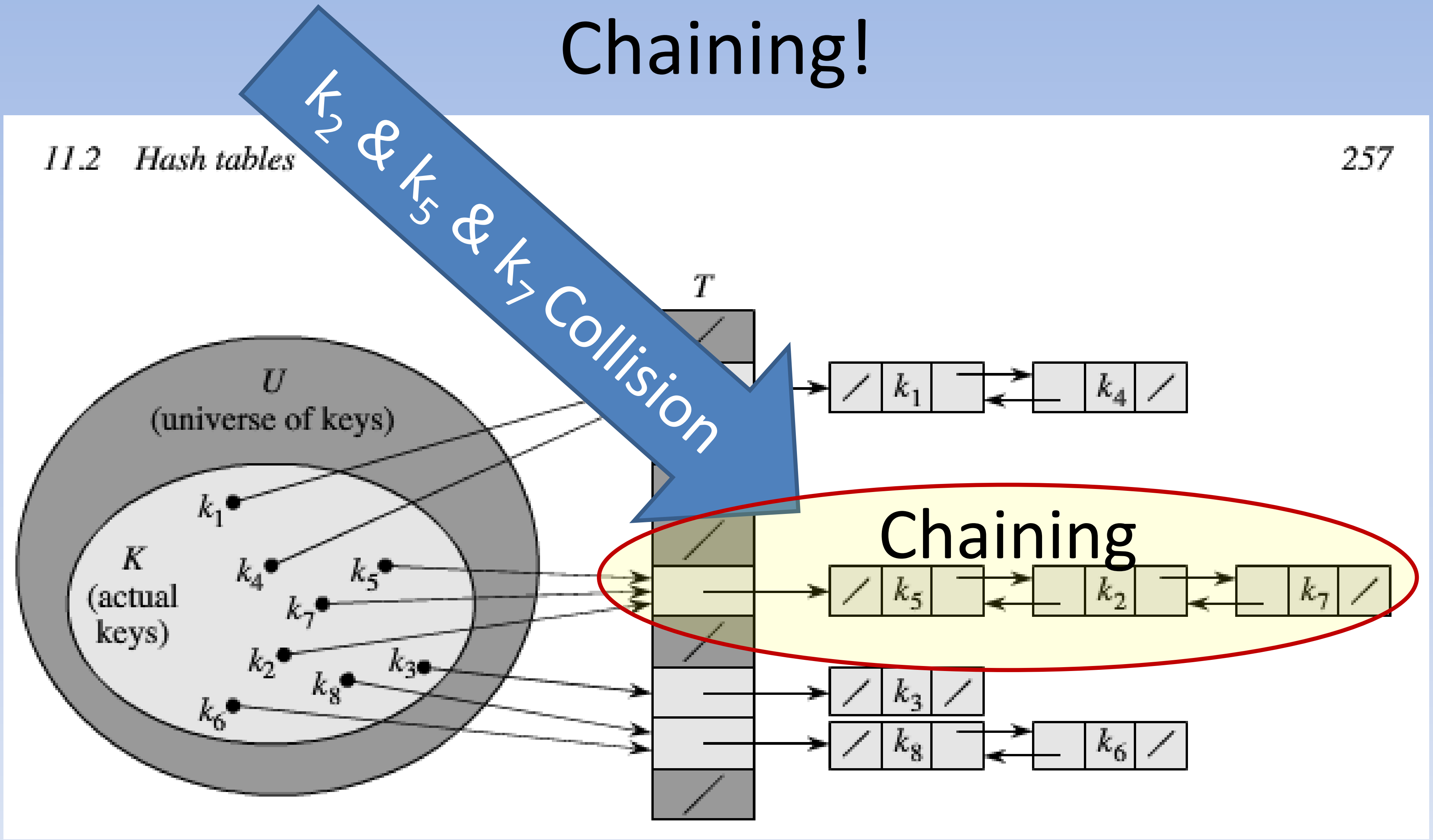
Hash Tables w/ Hash Function: h(k)



Hash Tables w/ Collisions



Hash Tables w/ Chaining!



How good is Hashing w/ Chaining ??

- How well does hashing perform w/ chaining ??

- How long to search for element w/ given key ??

Analyzing Hashing

- Hash Table T w/ m slots and n elements!
- Load factor α for T as n/m
  - average number of elements stored in a chain
  - Analysis in terms of α
    - Which can be less than, equal to, or greater than 1
- Worst-Case behavior Horrible w/ Hashing!
  - Θ(n) + time to produce Hash
- Average Performance depends on how well the hash spreads out the keys!

Analyzing Average w/ Hashing

- Average Performance depends on how well the hash spreads out the keys!
  - But for now ignore this!
- Assume any given element is equally likely to hash into any of the m slots!
  - **Simple Uniform Hashing!**

Chains w/ Expected Length

- Our performance will depend on the length of the chains hanging off the m slots!
- $n_j = len(T[j])$ for $j = 0, 1, \ldots, m-1$ denote the length of the chain hanging off slot j
- What is the expected value of n_j under our Simple Uniform Hashing assumption ??

Chains w/ Expected Length

- What is the expected value of n_j under our Simple Uniform Hashing assumption ??
- $E[n_j] =$ α = n/m

- Now our questions is given an element :
  – What's the TIME to FIND it?
- Two cases to consider :
  1. In the first case, we fail to find it..
     - It's not IN the Hash table!
  2. In the second case, we find it!
     - In this case, the key is IN the Hash table!

Theorem 11.1

- In a Hash Table, In which Collisions are Resolved by Chaining…
- An Unsuccessful Search takes average-case Time $\Theta(1 + \alpha)$,
  - UNDER the assumption of SIMPLE UNIFORM HASHING!

Theorem 11.1 w/ THE PROOF!

- Under our Uniform Hashing Assumption, h(k) is equally likely to send k to any of the m slots!
- So, the time needed to search unsuccessfully for k, is the Time needed to search to the end of the list at T(h(k)).
- The expected length of the list at T(h(k)) is

$$\mathrm{E}\left[n_{h(k)}\right] = \alpha$$

- Thus the total time required is α plus 1, $\Theta(1 + \alpha)$
  – 1 required to compute h(k)

Theorem 11.2

- In a Hash Table, In which Collisions are Resolved by Chaining…
- Successful Searches take average-case Time $\Theta(1 + \alpha)$
  - UNDER the Uniform Hashing Assumption

Theorem 11.2

- The item being searched for is equally likely to be any of the n elements stored in the table.
- Items are inserted into Hash Table chains at the top.
  - So the number of items examined before finding out item is the number of items inserted into its slot after our item.

Theorem 11.2

- x_i denotes the ith element inserted into the table, for $i = 1, 2, \dots, n$
- $k_i = x_i.key$
- Indicator Random Variables
  - Key to simplifying calculating expected values!
- Define indicator random variable :

$$X_{ij} = \mathrm{I}\{\mathrm{h}(\mathrm{k_i}) = \mathrm{h}(\mathrm{k_j})\}$$

  - True : if two keys Hash to the same Slot!
- Under our assumption:

$$\Pr(\{h(k_i) = h(k_j)\} = 1/m$$

$$E[X_{ij}] = 1/m$$

Theorem 11.2

$$E\left[\frac{1}{n}\sum_{i=1}^{n}(1 + \#Added\ to\ slot\ with\ x_i after\ x_i)\right]$$

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

- X_{ij} is 1 when x_i and x_j has to same slot!

Theorem 11.2

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

- By Linearity of Expectation!

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

Theorem 11.2

- By Linearity of Expectation!

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

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

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

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

Theorem 11.2

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

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

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

$$= 1 + \frac{n-1}{2m}$$

$$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$

Theorem 11.2

$$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$

- $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$

- w/ 11.1 & 11.2 We can conclude our average time performance as long as the number of items is proportional to the number of slots
  - $\alpha = n/m = O(m)/m = O(1)$

Hash Functions

- You can probably guess some of what's on our Hashing wish list?
- KEY: Each key is equally likely to Hash to any of the m slots
  - Independently of where any other keys has hashed to!
- We usually do not know enough about the distribution of the keys!

Hash Functions w/ Known Distribution

- IF we know the distribution of keys:
  - LIKE: keys are random real numbers independent and uniformly distribution in the range:

$$0 \le k < 1$$

- SO, in this case the condition of simple uniform hashing is satisfied by :

$$h(k) = \lfloor km \rfloor$$

First Assumption: Keys are Natural Numbers!

- If keys are really NOT naturals, we just try to find a way to look at them as natural!
- For Example: Character String
  - Assume characters in string are represented by ASCII code.
  - Treat string of characters a big int each each characters representing a digit in a radix-128 integer.
- "David" =

$$68 * 128^4 + 97 * 128^3 + 118 * 128^2 + 105 * 128 + 100 * 128^0$$

$$= 18458981604$$

Division Method

- Basic method for mapping keys from natural numbers into some m fixed number of slots
  - h(k) = k mod m
- Put some constraints on our choice of m to improve hash:
  - Do not use Power of 2!
    - Mod'ing to the power of 2 amounts to grabbing last bits
      - Many cases low-order bits not randomly distributed
      - Like function to operate on all bits
    - Permuting characters mod (radix-1) does not change value!

Permuting characters mod (radix-1) does not change value!

- h("David") = 18458981604 % (128-1) = 107
- h("avidD") = 26287436356 % (128-1) = 107
- h("vidDa") = 31897231969 % (128-1) = 107

```
h=lambda x : sum([ord(x[i])*128**(len(x)-i-1) for i in range(len(x))])%127
```

- You can prove this!
  - Exercise: 11.3-3
- A prime not too close to an exact power of 2 is often a good choice for m.

Hashing w/ Multiplication Method

- Here we introduce A, a number between 0 and 1.
- We'll multiply our key k by A giving us a real number.
- We'll mod that with one, getting the fractional part of kA.
  - A=0.25
  - (105*A) mod 1 = 0.25
  - (106*A) mod 1 = 0.5
  - (107*A) mod 1 = 0.75
- mA gives a value between 0 and m-1

Hashing w/ Multiplication Method

Python 2.7.5 Shell

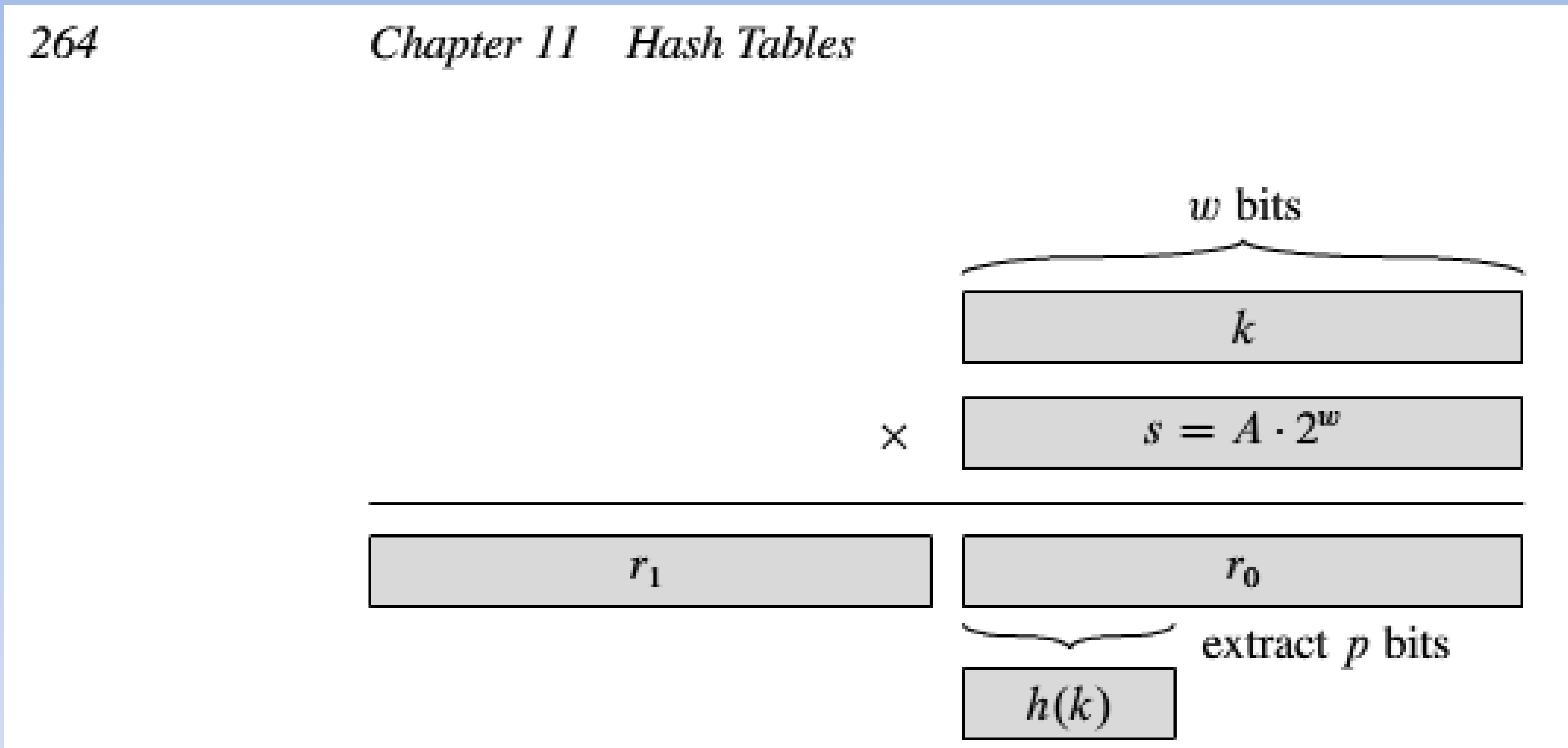
File Edit Shell Debug Options Windows Help

```
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> A=0.25
>>> 105*A
26.25
>>> (105*A) % 1
0.25
>>>
```

Ln: 8 Col: 4

1.

Hashing w/ Multiplication Method



- Now m doesn't matter:
  - Use $m=2^p$ for some p.
- Value of A does matter:
  - Knuth recommends $(\sqrt{5}-1)/2$ = 0.6280339887…

Hashing w/ Multiplication Method

- Use m=2^p for some p
  - p=7 so 2^p=128
- Use A=$(\sqrt{5}-1)/2$ = 0.6280339887…

- h("David") = 128*(18458981604*A%1) = 125
- h("avidD") = 128*(26287436356*A%1) = 13
- h("vidDa") = 128*(31897231969*A%1) = 112

Improving Worst-Case w/ Universal Hashing

- In the worst-case, bad luck (or malicious adversary) conspire to provide the worst possible sequence of keys to hash.
  - All keys hash to same slot!
- Whenever the hash function is unchanging, then this is always a possibility!
- Only effective defense is to select the hash function randomly and independent from keys!
- Universal Hashing is one approach!

Universal Hashing w/ Universal Collection of Hash Functions

- Assume we have :
  - two distinct keys k, l
  - A collection of hash function mapping to a range 0..m-1
- Our collection of hash functions is universal IF:
  - The size of the collection of all hash functions that map both k, l to the same slot is $(1/m)^{th}$ of the total number of functions in the collection.

Universal Collection Gives Good Performance !

- We know Hash Table performance depend on the length of the chains hanging off the m slots!
- Let $n_j = len(T[j])$ for $j = 0, 1, \ldots, m-1$ denote the length of the chain hanging off slot j

- What is the expected value of n_j if we choose hash function h from a Universal Collection ??

Universal Collection Gives Good Performance !

- Theorem 11.3:
  - If k is not in Hash table : $E\left[n_{h(k)}\right] \leq \alpha = (\mathrm{n}/\mathrm{m})$
  - If k is NOT in Hash table : $E\left[n_{h(k)}\right] \leq 1 + \alpha$
- Note this behavior does not depend upon any assumptions about the distribution keys!

Universal Collection Gives Good Performance ! **THE PROOF**

- Theorem 11.3:
  - If k is not in Hash table : $E\left[n_{h(k)}\right] \leq \alpha = (n/m)$
  - If k is NOT in Hash table : $E\left[n_{h(k)}\right] \leq 1 + \alpha$
- Note this behavior does not depend upon any assumptions about the distribution keys!

Key Tool: Indicator Variables

- For each pair of distinct keys k & l:

$$X_{kl} = I\{h(k) = h(l)\}$$

$$\Pr\{h(k) = h(l)\} \leq \frac{1}{m}$$

Therefore: $E[X_{kl}] \leq \frac{1}{m}$

Random Variable: Y_k

- For each key k, Y_k is the number of keys other than k that hash to the same slot:

$$Y_k = \sum_{\substack{l \in T \\ l \neq k}} X_{kl}$$

Thus:

$$E[Y_k] = E\left[\sum_{\substack{l \in T \\ l \neq k}} X_{kl}\right] = \sum_{\substack{l \in T \\ l \neq k}} E[X_{kl}]$$

E[$N_{h(k)}$]
How long is list in slot h(k)?

- If k is NOT in the hash table, then $n_{h(k)} = Y_k$
  - $E[n_{h(k)}] = E[Y_k] \leq \frac{n}{m} = \alpha$

- If k is IN the hash table, then Y_k does not include k, so
  - $n_{h(k)} = Y_k + 1$ = (n-1/m)+1
  - (n/m) – (1/m) + 1 $\leq$ α + 1

Corollary 11.4 w/ Theorem 11.3

- Corollary 11.4 uses 11.3 to show that :
  - Search operation with **n** total Inserts & Searches & Deletes
  - but O(**m**) Inserts out of total
- is O(1) for each search,
- so w/ Linearity of Expectation is O(n) for the collection of operations.

Universal Class of Hash Functions w/ a little number theory

- We'll return to this w/ RSA Encryption
- Choose prime number p so all keys are in range 0..p-1 inclusive!
- Now Define:
  - $\mathbb{Z}_p$={0, 1, …, p-1} More on this set later!
  - $\mathbb{Z}^*_p$={1, …, p-1}
- Now we can define a whole collection of hash functions as:
  - $h_{ab}(k)=((ak+b) \bmod p) \bmod m$
  - $\mathcal{H}_{pm}=\{h_{ab}: a\in \mathbb{Z}^*_p,\ b\in \mathbb{Z}_p\}$ which contains p(p-1) functions

Theorem 11.5: Class $\mathcal{H}_{pm}$ is Universal!

- Take our two distinct keys k & l w/ $k \neq l$
- Given our hash function h_{ab}
  - $r = (ak + b)$ mod p
  - $s = (al + b)$ mod p
- r and s cannot be equal because
  - p is prime
  - and r-s ≡ a(k-l) (mod p)
    - r-s cannot be 0
- p is the prime number covering the range of original key!
  - Since p is prime,
  - and a and k-l are non-zero modulo p
  - THEN the product a(k-l) is non zero modulo p.
- **SO: mod p level has no collisions**
  - Remember: Hash is w/ mod m !

mod p level has no collisions

- SO: Each possible pair (a,b) yields a different resulting pair (r,s).
- There are p(p-1) possible choices for (a,b)
- So picking (a,b) uniformly at random means the resulting pair (r,s) is equally likely to be any pair of distinct values mod p.
- SO: the probability of two keys colliding is the same as the probability of $r \equiv s \bmod m$

mod p level has no collisions

- The number of choices for s from the total (p-1) :
  - once an r value is chosen
  - where r and s are not equal
  - but r and s are equal mod m
  - is at most :

$$\lceil p/m \rceil - 1 \ \leq (p-1)/m$$

- SO: The probability of choosing on of these is:

$$\frac{\left(\frac{p-1}{m}\right)}{(p-1)} = \frac{1}{m}$$

- SO!!! : $\mathcal{H}_{\mathrm{pm}}$ IS UNIVERSAL!!!

Universal Hashing

- Now a given our large prime number p
- We have a large class of hash functions to choose from
  - p*p-1

Open Addressing

- Currently we have dealt with keys colliding at same slots w/ chains!
  - Chains hang off hash table!
- Open Addressing is an alternative w/ all elements kept IN hash table!
- Slots in Hash Table are examined systematically until item is found or deemed absent!
- All slots can get “filled up” w/ Open Addressing

Open Addressing w/ NO POINTERS!!

- Could keep chain in Hash Table, but we don't
- Freeing pointers provides more storage space.
  - Just use empty space in hash table!
- Insert key by probing slots until finding an empty one to put key in!
- Slots probed are ordered based on the inserted key!

Probing Slots

- Probe sequence is determined by expanding Hash function
- Hash function includes additional parameter Probe Number (starting w/ 0)

  $h: U\ x\ \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$
- Our probe sequence must be a permutation of the set of slots (so all slots get probed!):

<h(k,0), h(k,1), …, h(k, m-1)>

= Permutation (<0, 1, …, m-1>)

Inserting & Finding

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

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

- Finding follows same probe sequence as Inserting.
- Removal can be tricky!

Open Addressing w/ Key Deletion

- Cannot just remove the item, marking it nil
  - May not know to continue probing sequence while searching
- Instead, mark item DELETED.
  - Insertion treats slots deleted as empty.
  - Search walks over deleted slots
- BUT, w/ deleted items:
  - searching no longer depends on load factor n/m!
    - Since slots with deleted items still potentially searched.
  - Frequently avoided when deletion needed!

Probing Techniques w/ Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing!

- BUT: None satisfy assumptions of Uniform Hashing
  - Limited in the number of probe sequences generated
    - m^2 versus m!
    - Double Hashing generating the most probe sequences

Linear Probing

- Auxiliary Hash Function:

$$h' : U \rightarrow \{0, 1, \dots, m-1\}$$

- Hash Function

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

- How many distinct probes??

Linear Probing

- Auxiliary Hash Function:

$$h': U \rightarrow \{0, 1, \ldots, m-1\}$$

- Hash Function

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

- How many distinct probes?? m
- Suffers from Primary Clustering
  - Long runs of occupied slots
  - Increased search time

Quadratic Probing

$$h(k, i): (h'(k) + c_1 i + c_2 i^2) \bmod m$$

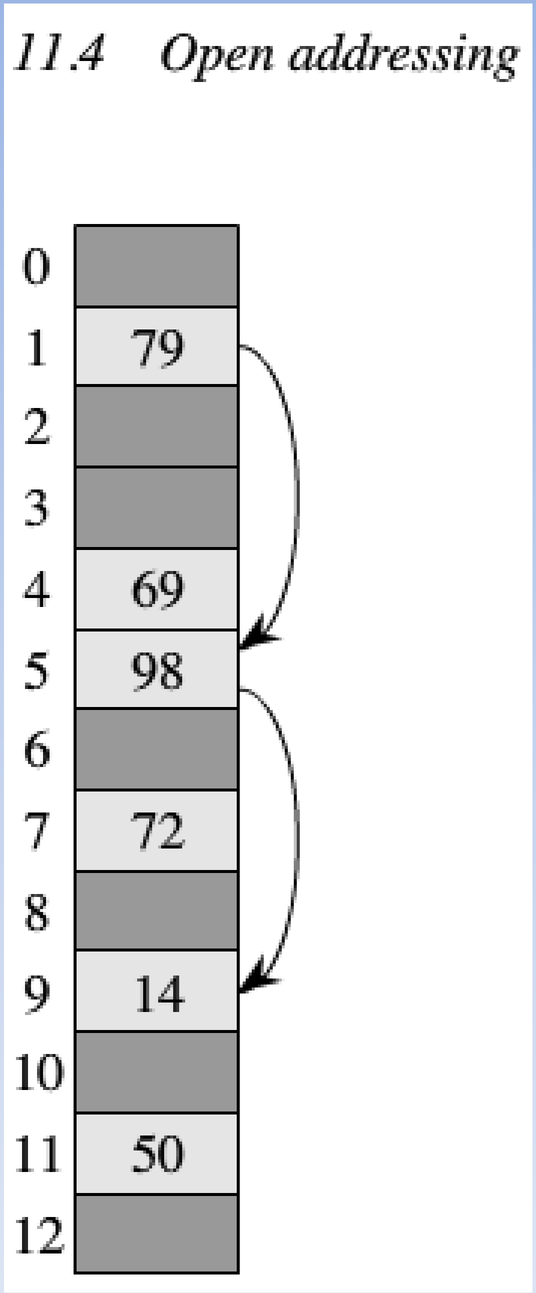
- c_1, c_2, and m are constrained
- Still suffers from clustering (secondary clustering).
- Still just m distinct probes.

Double Hashing

- One of the best for Open Addressing

$$h(k,i): (h_1(k) + ih_2(k)) \text{ mod } m$$

- $h_1(k) = k \text{ mod } 13$
- $h_2(k) = 1 + (k \text{ mod } 11)$
- k=14
  - h(14,0) = 1
  - h(14,1) = 1 + (1+3) = 5
  - h(14,2) = 1 + 2*(1+3) = 9



Double Hashing

- One approach is :

$$h_1(k) = k \bmod m$$
$$h_2(k) = 1 + (k \bmod m')$$

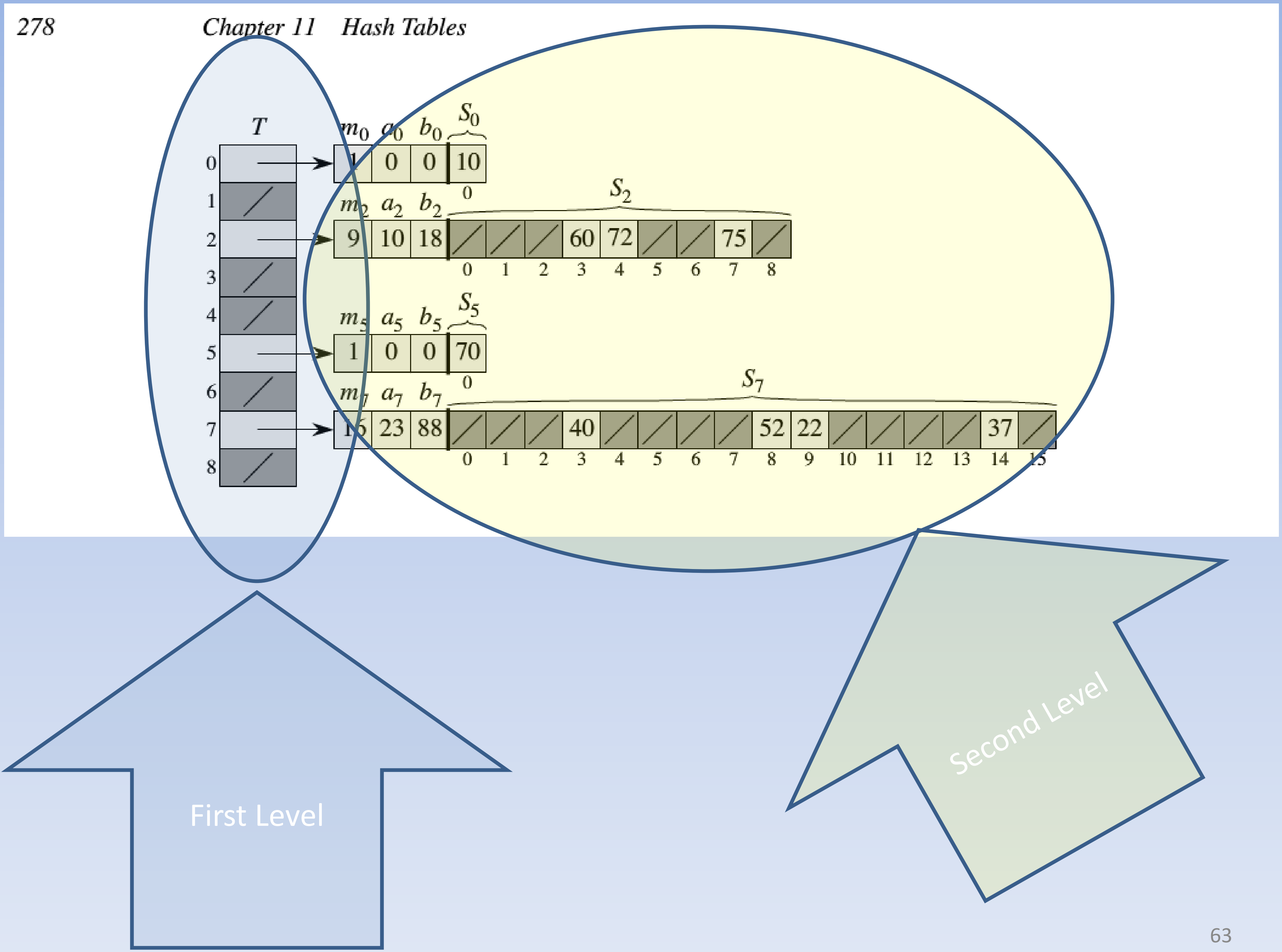
- Where
  - m = Prime Number
  - m' = is slightly less, like (m-1)
- Double hashing provides for m^2 probe sequences
  - $h_1(k)$ and $h_2(k)$ yield distinct probe sequences!

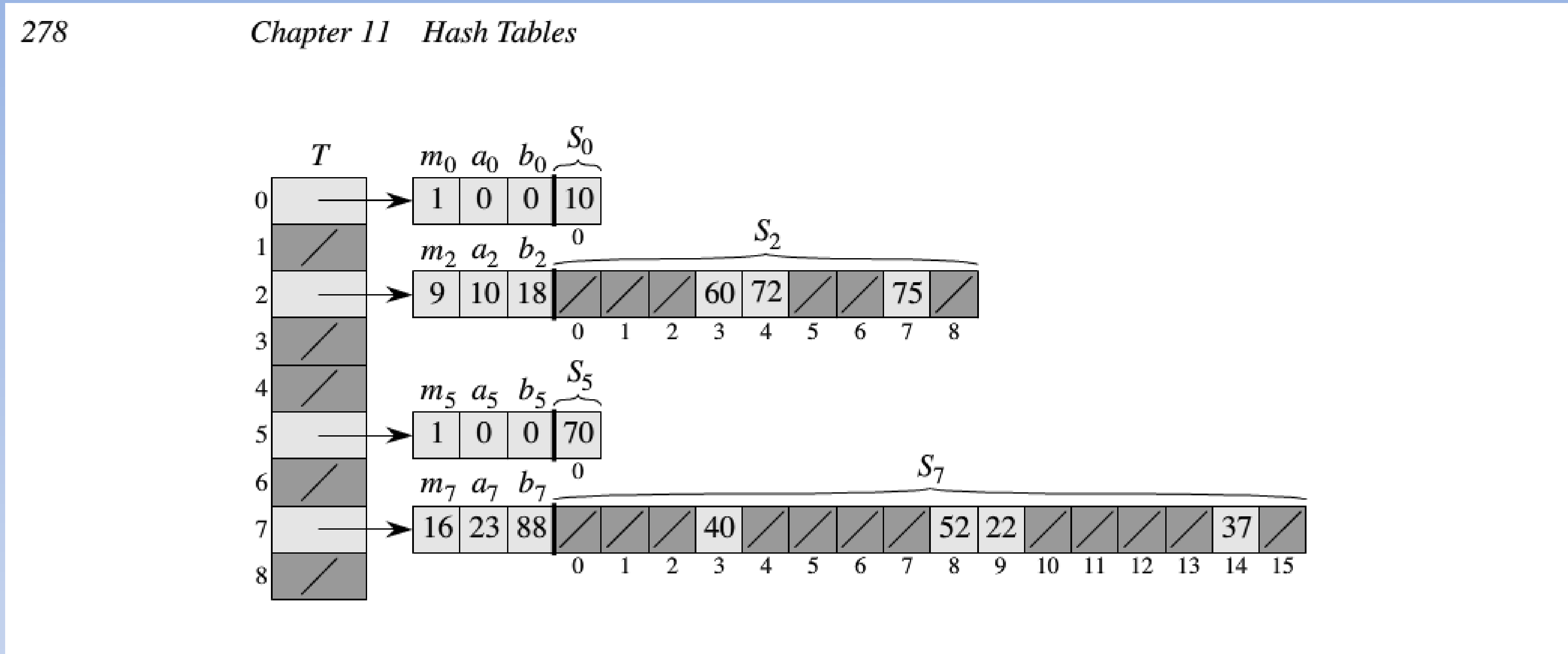
Perfect Hashing

- So far we've seen Hashing has GREAT average-case performance.
- But did you know…???
  - Hashing can also provide excellent worst-case performance…
  - IF: Keys are static!
    - Once stored the set of keys never changes!

Perfect Hashing

- Perfect Hashing uses Universal Hash Function initially
  - like hashing with chaining
- BUT : Instead of Chaining, a second hash function is used!
- TWO Levels of hashing combine to provide O(1) worst-case behavior!


278
Chapter 11 Hash Tables
T
m_0 a_0 b_0 S_0
1 0 0 10
m_2 a_2 b_2 S_2
9 10 18 60 72 75
m_5 a_5 b_5 S_5
1 0 0 70
m_7 a_7 b_7 S_7
16 23 88 40 52 22 37
First Level
Second Level



- The size of the second level hash table is :

$$S_j is\ m_j = {n_j}^2$$

- No collisions occur with secondary hashing!
- The hash function for secondary hashing for each hash table is h_j and is chosen from a set of universal hash functions!

Theorem 11.9

- If we store n keys in a hash table of size $m = n^2$ then the probability of ANY collisions is ½!

- This is a great result!
- BUT, in some cases n^2 is too large for our table size.

Theorem 11.9 w/ Proof

- If we store n keys in a hash table of size $m = n^2$ then the probability of ANY collisions is ½!

$$Potential\ Collisions = \binom{n}{2}$$

- With our universal hashing functions the probability of each collision is 1/m.
- We choose $m = n^2$
- SO w/ X as the random variable counting the number of collsions:

$$E(X) = \binom{n}{2} * \frac{1}{m} = \binom{n}{2} * \frac{1}{n^2}$$

Theorem 11.9 w/ Proof

$$\begin{aligned} \mathrm{E}[X] &= \binom{n}{2} \cdot \frac{1}{n^2} \\ &= \frac{n^2 - n}{2} \cdot \frac{1}{n^2} \\ &< 1/2\,. \end{aligned}$$

- $P\{X \geq 1\} \leq E[X]$
  - Markov Inequality
  - Similar to Birthday Paradox Analysis

Theorem 11.9

- If we store n keys in a hash table of size $m = n^2$ then the probability of ANY collisions is ½!

- SO: chances are that the probability of any collisions with $m = n^2$ is very low.
  - Trial and error should find one given a fixed set of keys with NO collisions!

Perfect Hashing w/ Too Many Keys

- Unfortunately, in some cases n^2 size table is TOO large!
- In these cases, the two level hashing is brought to bear upon the problem!

- NOW issue is to show that the quadratic space requirement is avoided!

Theorem 11.10

- If we store n keys in a hash table size m=n,
- THEN we are going to need second level hash tables of size ${n_j}^2$
  - Here n_j is the number of keys that hash to slot j.
- THEN we want the Expected sum of all of these second level hash tables to be less than 2n!!

Theorem 11.10

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions. Then, we have

$$\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n ,$$

where n_j is the number of keys hashing to slot j.

$$a^2 = a + 2\binom{a}{2} . \tag{11.6}$$

We have

$$
\begin{aligned}
&\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \\
&\quad = \mathrm{E}\left[\sum_{j=0}^{m-1}\left(n_j + 2\binom{n_j}{2}\right)\right] && \text{(by equation (11.6))} \\
&\quad = \mathrm{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(by linearity of expectation)} \\
&\quad = \mathrm{E}[n] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(by equation (11.1))}
\end{aligned}
$$

$$a^2 = a + 2\binom{a}{2} . \qquad (11.6)$$

We have

$$\begin{aligned}
\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] &= \mathrm{E}\left[\sum_{j=0}^{m-1}\left(n_j + 2\binom{n_j}{2}\right)\right] && \text{(by equation (11.6))} \\
&= \mathrm{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(by linearity of expectation)} \\
&= \mathrm{E}[n] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(by equation (11.1))}
\end{aligned}$$

Sum averages to n

$$a^2 = a + 2\binom{a}{2} . \tag{11.6}$$

We have

$$\begin{aligned} \mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] &= \mathrm{E}\left[\sum_{j=0}^{m-1}\left(n_j + 2\binom{n_j}{2}\right)\right] && \text{(by equation (11.6))} \\ &= \mathrm{E}\left[\sum_{j=0}^{m-1} n_j\right] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] && \text{(by linearity of expectation)} \\ &= \mathrm{E}\left[n\right] + 2\,\mathrm{E}\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] \end{aligned}$$

Same as #colliding key pairs

$$\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2},$$

Expected #colliding key pairs

since $m = n$. Thus,

$$\mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \le n + 2\,\frac{n-1}{2} = 2n - 1 < 2n\,.$$

■

Corollary 11.11
Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions, and we set the size of each secondary hash table to $m_j = n_j^2$ for $j = 0, 1, \ldots, m-1$. Then, the expected amount of storage required for all secondary hash tables in a perfect hashing scheme is less than $2n$.

Proof Since $m_j = n_j^2$ for $j = 0, 1, \ldots, m-1$, Theorem 11.10 gives

$$\begin{aligned} \mathrm{E}\left[\sum_{j=0}^{m-1} m_j\right] &= \mathrm{E}\left[\sum_{j=0}^{m-1} n_j^2\right] \\ &< 2n\,, \end{aligned} \tag{11.7}$$

Space required by second level is LINEAR in number of key values!

which completes the proof. ■

Corollary 11.12

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions, and we set the size of each secondary hash table to $m_j = n_j^2$ for $j = 0, 1, \ldots, m - 1$. Then, the probability is less than $1/2$ that the total storage used for secondary hash tables equals or exceeds $4n$.

- With probability less than ½ the total storage exceeds (or equals) 4n!

- Try a few random hash functions, and one is bound to give good performance!!!