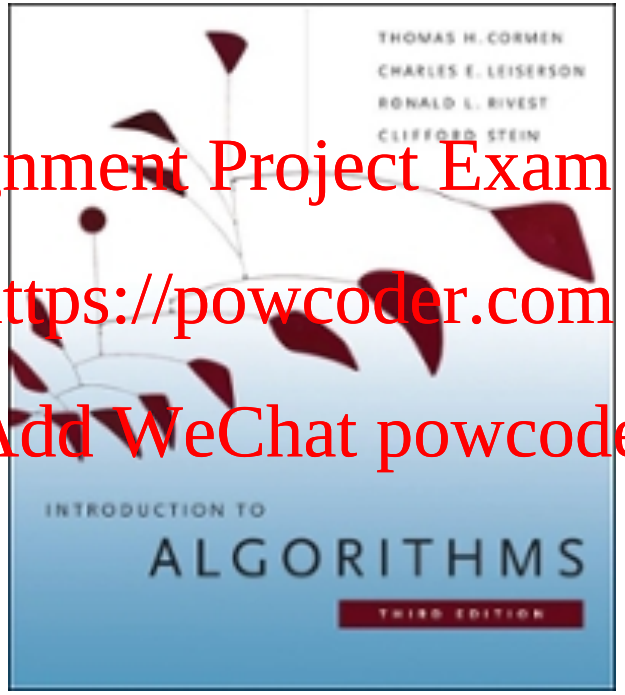
CS146 Data Structures and Algorithms



Chapter 11: Hash Tables

Why? Hashing Tables

- Motivation: symbol tables
  - A compiler uses a *symbol table* to relate symbols to associated data
    - Symbols: variable names, procedure names, etc.
    - Associated data: memory location, call graph, etc.
  - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
  - We typically don't care about sorted order

Dictionary

- **Dictionary:**
  - Dynamic-set data structure for storing items indexed using *keys*.
  - Supports Assignment Project Exam Help Delete.
  - Applications:
    - Symbol table of a compiler.
    - Memory-management tables in operating systems.
    - Predicting search keywords (Google search engine, etc.)
    - Search a person and show his/her friends, make a friend, unfriend a person in Facebook.
- **Hash Tables:**
  - Effective way of implementing dictionaries.
  - Generalization of ordinary arrays.

Hash Tables

- More formally:
  - Given a table T and a record x, with key (= symbol) and satellite data, we need to support:
    - Insert (T, x)
    - Delete (T, x)
    - Search(T, x)
  - We want these to be fast, but don't care about sorting the records
- The structure we will use is a *hash table*
  - Supports all the above in O(1) expected time!

Hashing: Keys

- In the following discussions we will consider all keys to be (possibly large) natural numbers
- *How can we convert floats to natural numbers for hashing purposes?*
- *How can we convert ASCII strings to natural numbers for hashing purposes?*

Direct Addressing

- Suppose:
  - The range of keys is 0..m-1
  - Keys are distinct
- The idea:
  - Set up an array T[0..m-1] in which
    - T[i] = x if $x \in T$ and key[x] = i
    - T[i] = NULL otherwise
  - This is called a *direct-address table*
    - Operations take O(1) time!
    - *So what's the problem?*

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.
- 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.
  - Details in Sec. 11.1.

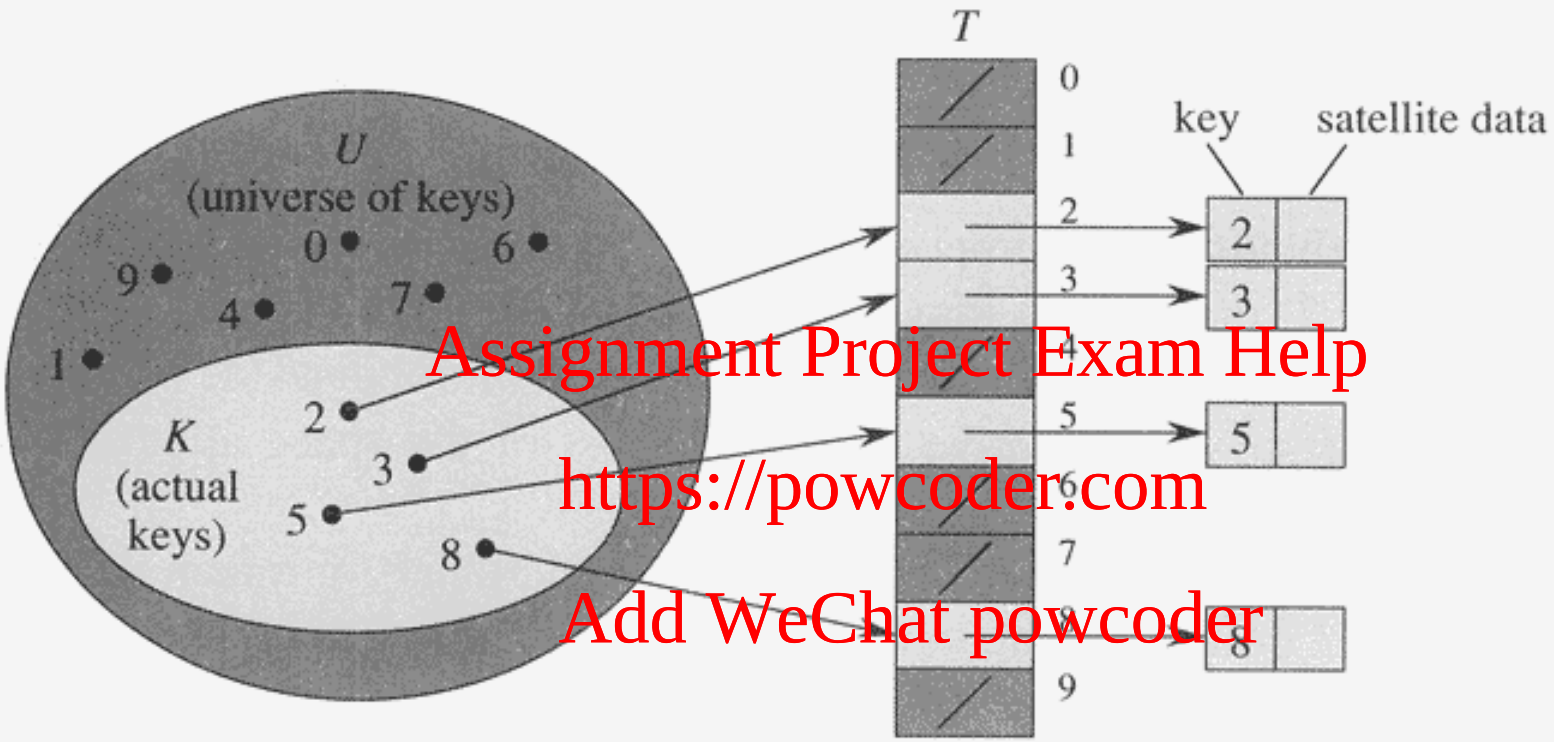


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded,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

The Problem With Direct Addressing

- Direct addressing works well when the range m of keys is relatively small
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be
- Solution: map keys to smaller range 0..m-1
- This mapping is called a *hash function*

Hash Tables

- **Notation:**
  - U – Universe of all possible keys.
  - K – Set of keys actually stored in the dictionary.
  - $|K| = n$.
- When U is very large,
  - Arrays are not practical.
  - $|K| << |U|$.
- Use a table of size proportional to $|K|$ – The hash tables.
  - However, we lose the direct-addressing ability.
  - Define functions that map keys to slots of the hash table.

Hashing

- Hash function h: Mapping from U to the slots of a hash table $T[0..m-1]$.

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

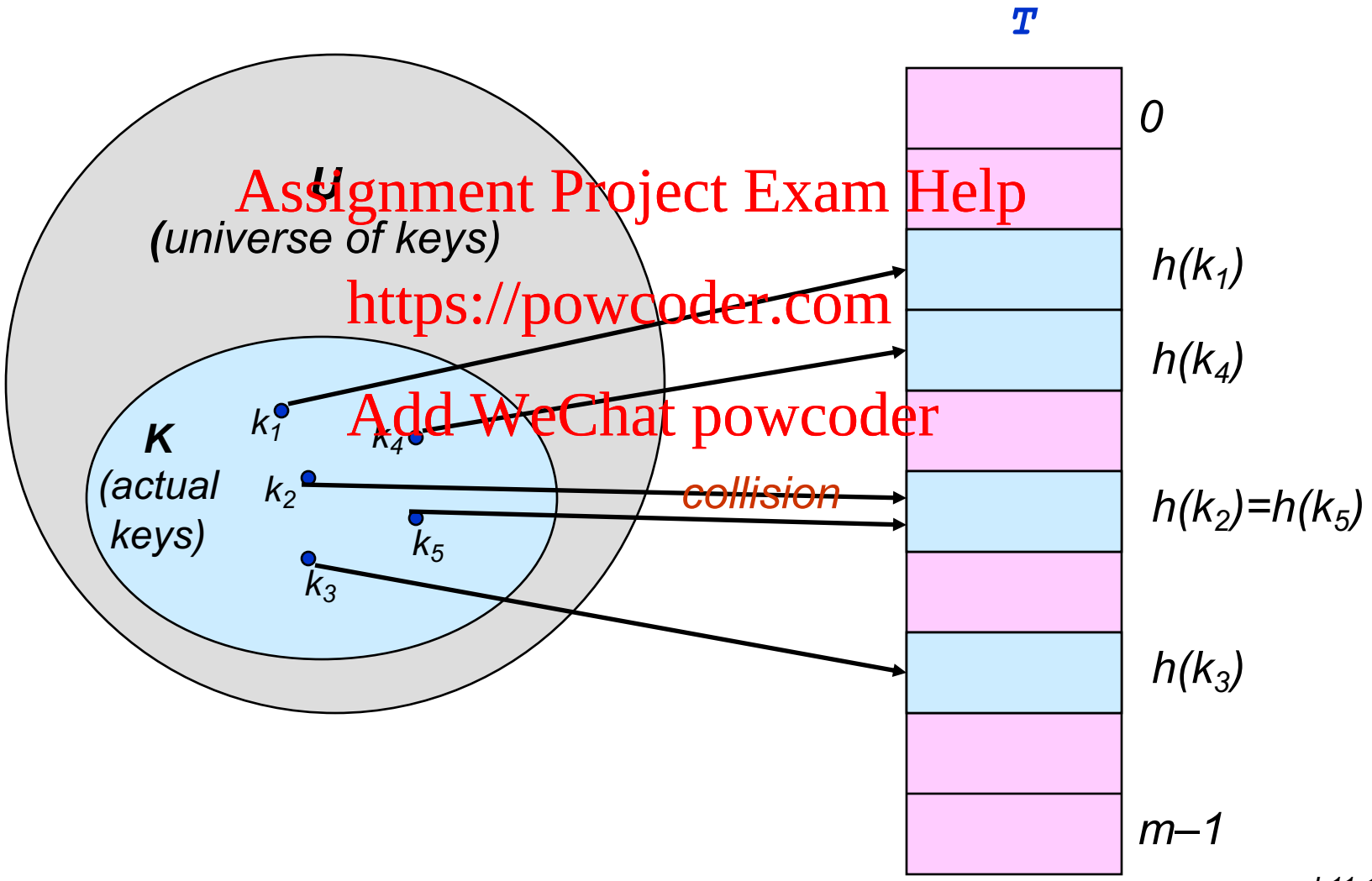
- With arrays, key k maps to slot $A[k]$.

- With hash tables, key k maps or "hashes" to slot $T[h[k]]$.

- $h[k]$ is the *hash value* of key k.

Hash Functions

Problem: collision

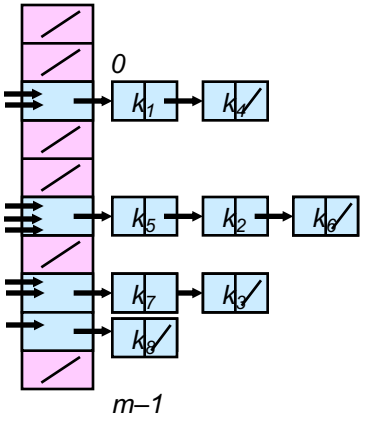


Issues with Hashing

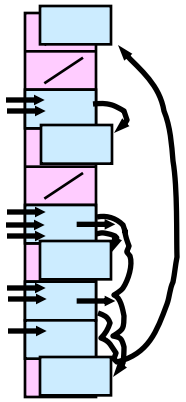
- Multiple keys can hash to the same slot – collisions are possible.
  - Design hash functions such that collisions are minimized.
  - But avoiding collisions is impossible.
    - Design collision-resolution techniques.
- Search will cost $\Theta(n)$ time in the worst case.
  - However, all operations can be made to have an expected complexity of $\Theta(1)$.

Methods of Resolution

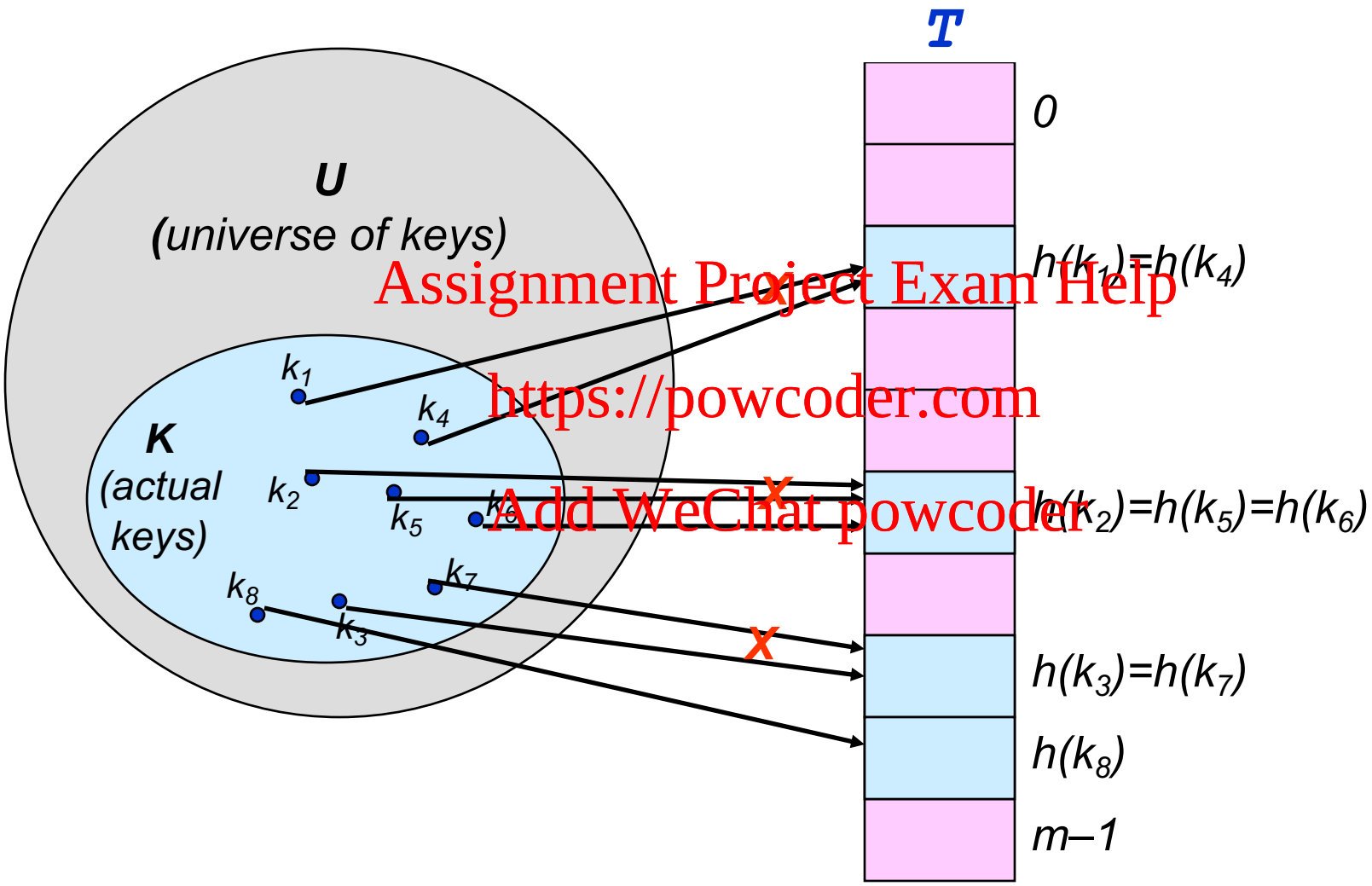
- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



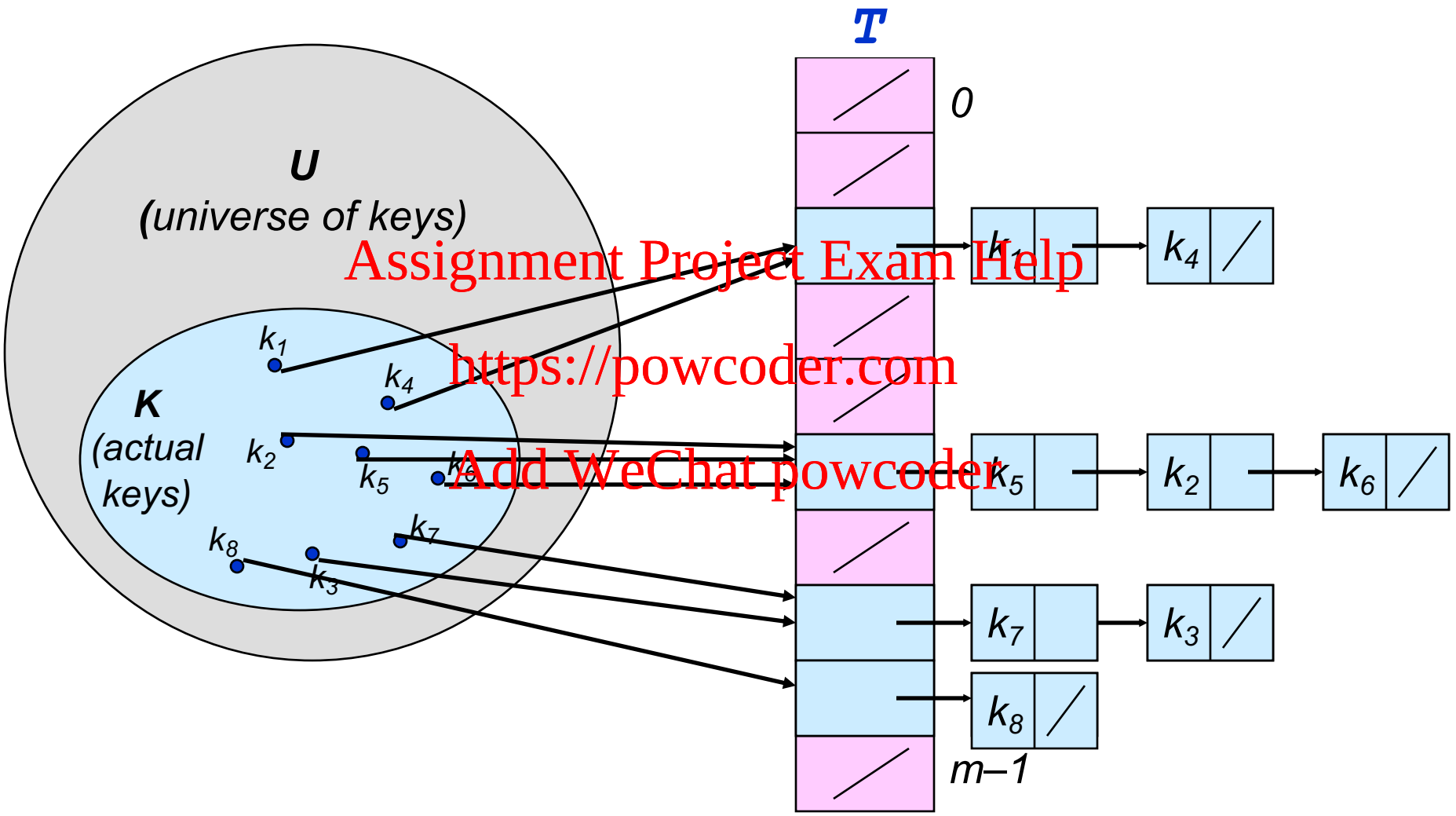
- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Collision Resolution by Chaining



Collision Resolution by Chaining



Hashing with Chaining

Dictionary Operations:

- Chained-Hash-Insert (*T*, *x*)
  - Insert *x* at the head of list *T*[*h*(*key*[*x*])].
  - Worst-case complexity – *O*(1).
- Chained-Hash-Delete (*T*, *x*)
  - Delete *x* from the list *T*[*h*(*key*[*x*])].
  - Worst-case complexity – proportional to length of list with singly-linked lists. *O*(1) with doubly-linked lists.
- Chained-Hash-Search (*T*, *k*)
  - Search an element with key *k* in list *T*[*h*(*k*)].
  - Worst-case complexity – proportional to length of list.

Expected Cost – Interpretation

- Load factor $\alpha = n/m$ = average keys per slot.
- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$.

$\Rightarrow$ Searching takes constant time on average.

- Insertion is $O(1)$ in the worst case.

- Deletion takes $O(1)$ worst-case time when lists are doubly linked.

- Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.

Good Hash Functions

- Satisfy the assumption of *simple uniform hashing*.
  - Not possible to satisfy the assumption in practice.
- Often use heuristics, based on the domain of the keys, to create a hash function that performs well.
- Regularity in key distribution should not affect uniformity. Hash value should be independent of any patterns that might exist in the data.
  - E.g. Each key is drawn independently from U according to a probability distribution P:
    $\sum_{k:h(k)=j} P(k) = 1/m$ for $j = 0, 1, \ldots, m-1$.
  - An example is the division method.

Keys as Natural Numbers

- Hash functions assume that the keys are natural numbers.
- When they are not, have to interpret them as natural numbers.
- Example: Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
  - ASCII values: C=67, L=76, R=82, S=83.
  - There are 128 basic ASCII values.
  - So, CLRS = $67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0$
    $= 141,764,947.$

Choosing A Hash Function

- Choosing the hash function well is crucial
  - Bad hash function puts all elements in same slot
  - A good hash function:
    - Should distribute keys uniformly into slots
    - Should not depend on patterns in the data
- We discussed three methods:
  - Division method
  - Multiplication method
  - Universal hashing

Division Method

- Map a key k into one of the m slots by taking the remainder of k divided by m. That is,

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

- Example: $m = 31$ and $k = 78 \Rightarrow h(k) = 16$.
- **Advantage:** Fast, since requires just one division operation.
- **Disadvantage:** Have to avoid certain values of m.
  - Don't pick certain values, such as $m=2^p$
  - Or hash won't depend on all bits of k.
- **Good choice for m:**
  - Primes, not too close to power of 2 (or 10) are good.

Multiplication Method

- If $0 < A < 1$, $\boldsymbol{h(k) = \lfloor m\,(kA \bmod 1)\rfloor = \lfloor m\,(kA - \lfloor kA \rfloor)\rfloor}$

  where kA mod 1 means the fractional part of kA, *i.e.*, $kA - \lfloor kA \rfloor$.

- Disadvantage: Slower than the division method.
- Advantage: Value of m is not critical.
  - Typically chosen as a power of 2, i.e., $m = 2^p$, which makes implementation easy.

- Example: $m = 1000$, $k = 123$, $A \approx 0.6180339887\ldots$

$$h(k) = \lfloor 1000(123 \cdot 0.6180339887 \bmod 1)\rfloor$$
$$= \lfloor 1000 \cdot 0.018169\ldots\rfloor = 18.$$

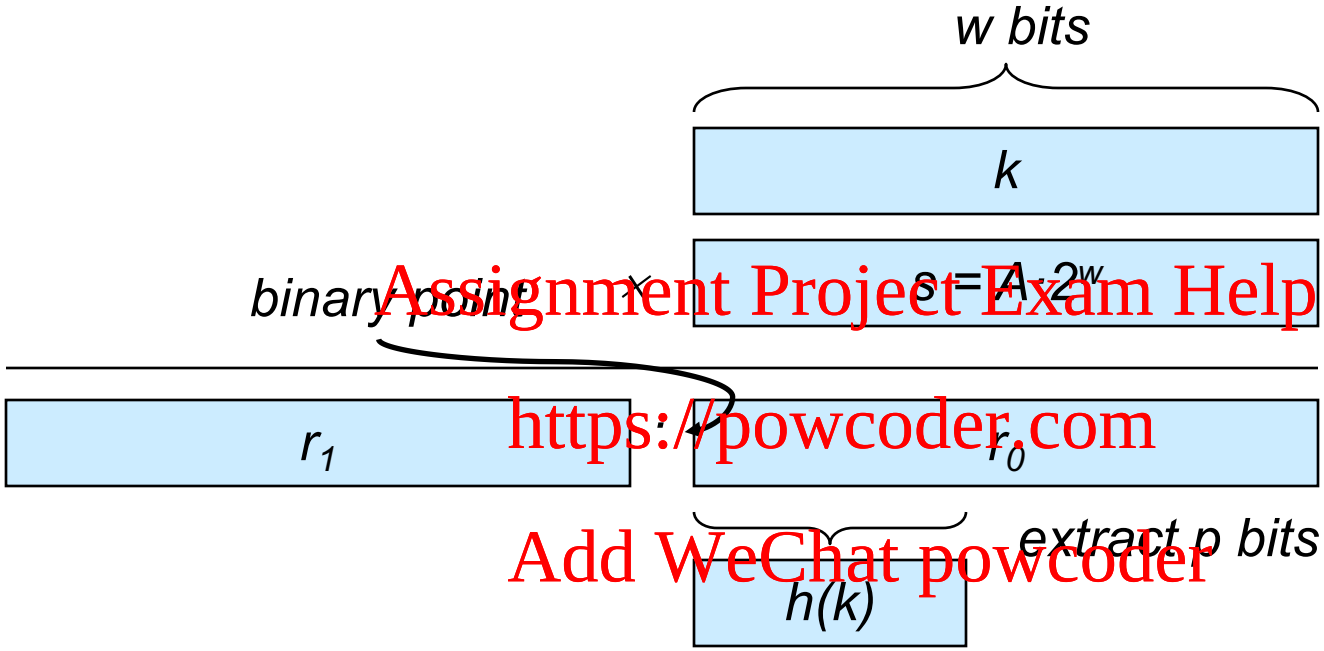
How to choose A?

- How to choose A?
  - The multiplication method works with any legal value of A.
  - But it works better with some values than with others, depending on the keys being hashed.
  - Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.

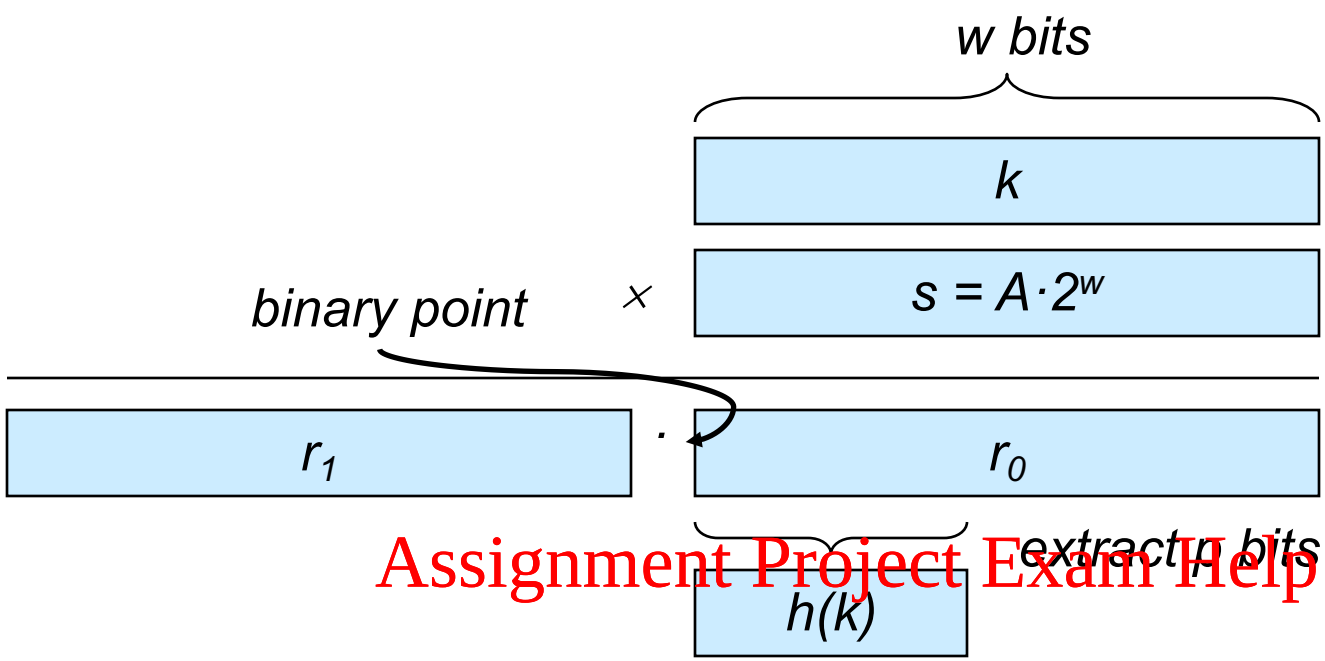
Multiplication Method. – Implementation

- Choose $m = 2^p$, for some integer p.
- Let the word size of the machine be w bits.
- Assume that k fits into a single word. (k takes w bits.)
- Let $0 < s < 2^w$. (s takes w bits.)
- Restrict A to be of the form $s/2^w$.
- Let $k \times s = r_1 \cdot 2^w + r_0$.
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$) and r_0 holds the fractional part of kA ($kA \bmod 1 = kA - \lfloor kA \rfloor$).
- We don't care about the integer part of kA.
  - So, just use r_0, and forget about r_1.

Multiplication Method – Implementation



- We want $\lfloor m\ (kA \bmod 1) \rfloor$. We could get that by shifting r_0 to the left by $p = \lg m$ bits and then taking the p bits that were shifted to the left of the binary point.
- But, we don't need to shift. Just take the p most significant bits of r_0.



Example:

- K = 123456, $p = 14$, $m = 2^{14} = 16384$, $w = 32$,
- Adapting Knuth's suggestion, A to be the fraction of the form $s/2^{32}$ that is close to $\approx (\sqrt{5} - 1)/2$, so $A = 2654435769 / 2^{32}$
- K*S = 327706022297664 = (76300 * 2^{32}) + 17612864
- $r_1 = 76300$, $r_0 = 17612864$,
- The 14 most significant bits of r_0 yield $h(k) = 67$.