Introduction to
Data Structures and Algorithms

Chapter: **Hash Tables**

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl Informatik 7 (Prof. Dr.-Ing. Reinhard German)
Martensstraße 3, 91058 Erlangen

Hash Tables

- Abstract data type *Table* (ADT) with *table entries*
  - Each table entry contains a unique key *K*
  - A table entry may contain some information I (satellite data)
  - => a table entry is an ordered pair(K,I)
- An example is a compiler that needs to maintain a symbol table *T*
  - The keys of *T* are character strings which correspond to identifiers of programming language
  - The information I of each table entry (symbol table) are the attributes of the compiler parsing process

Hash Tables

- Expl 1: C program parsing

```
// Declare an external function
extern double bar (double) x);

// Define a public function
double foo (int count)
 {
    double sum = 0.0 ;
    // Sum all the values bar(1) to bar(count)
    for ( int i = 1; i <= count; i++)
         sum += bar((double) i) ;
    return sum;
 }
```

Hash table T(K,I)

Key K — Information I

Symbol name	Type	Scope
bar	function, double	extern
x	double	funct parameter
foo	function, double	global
count	Int	funct parameter
sum	double	block local
i	int	loop statement

Hash Tables

- Expl 2: Airport Codes and Names

Key *K* = Airport Code	Associated Information *I* = City
AKL	Auckland, New Zealand
DCA	Washington, D.C.
FRA	Frankfurt, Germany
GCM	Grand Cayman, Cayman Islands
GLA	Glasgow, Scotland
HKG	Hong Kong, China
LAX	Los Angeles, California
ORY	Paris, France
PHL	Philadelphia, Pennsylvania

Hash Tables

- Characteristics of an abstract data type *Table* (ADT)
  - The ADT contains character strings of variable length
  - An ADT supports typical "*Dictionary Operations*" as
    - *Searching* for a table entry (*K, I*) at given key *K*
    - *Deleting* an entry of ADT with given information *I*
    - *Inserting* an entry into ADT with special information *I*

  - The strings are considered as keys for an entry
  - The "Dictionary Operations" should be very efficient, preferably independent of the length of the table

- Such abstract data type is a generalization of an (associative) array and is called a **Hash Table**

Hash Tables

Mapping Data to a Hash Table T: Direkt adressing

- Effective access to Hash Table T
  - The set of possible key values is called the universe *U* of *keys*
  - Be $K \subseteq U$ the set of actual keys, which have to be mapped to T
  - If *K* is small relative to the number of *U* $(K < |U|)$ then we can use T simple as a **direct-address table** T[0,…,m-1]
  - Each position (slot) of the array T corresponds to a key *k* in the universe *U* : T[k] corresponds to key k

Hash Tables

Mapping Data to a Hash Table T: Direkt adressing

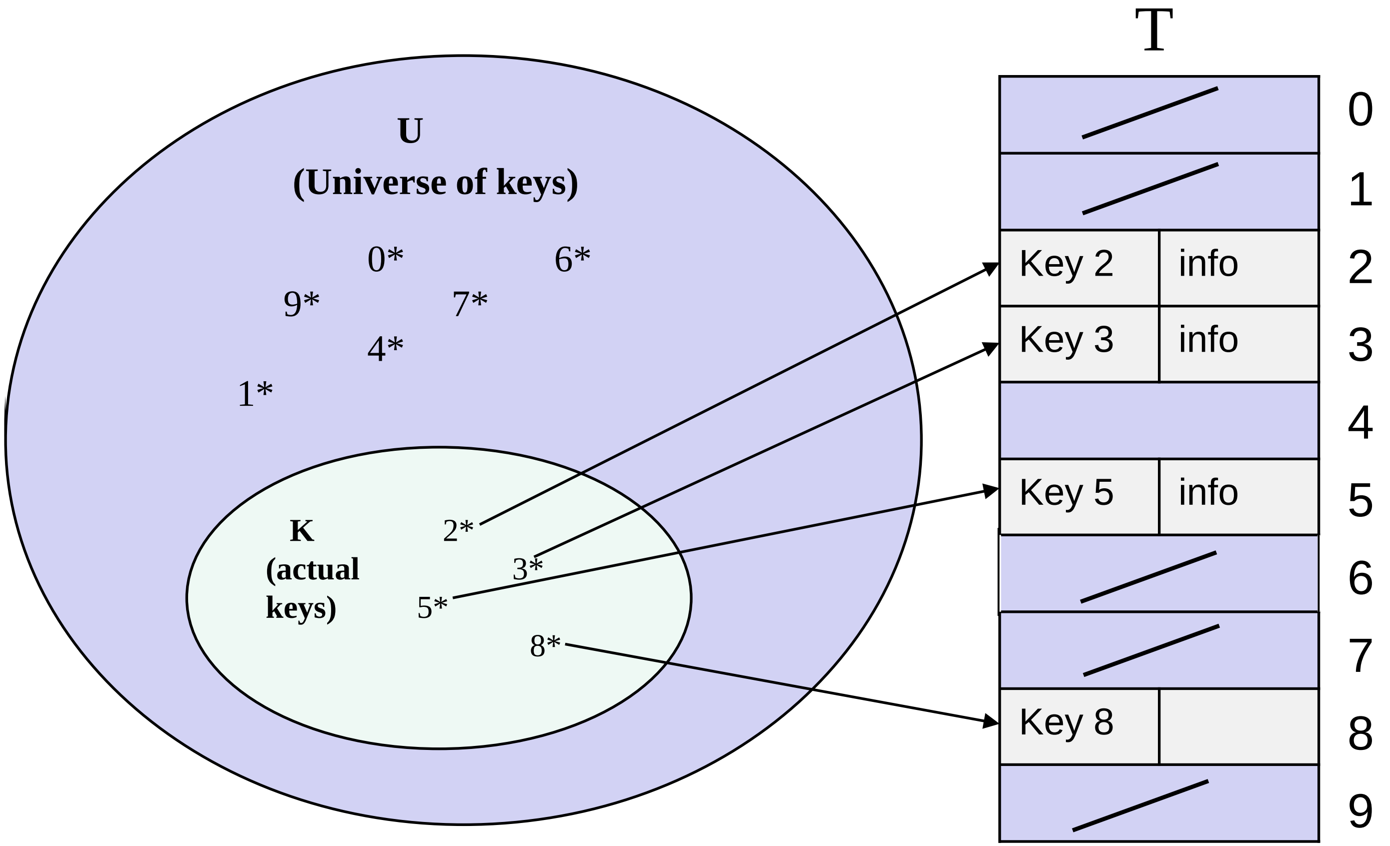
- The dictionary operations Search, Insert and Delete are trivial to implement

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

Direct-Address-Insert(T,x)
      T[key[x]]:= x

Direct-Address-Delete(T,x)
      T[key[x]]:=NIL
```

- The runtime for each of these operations is constant O(1)

Hash Tables

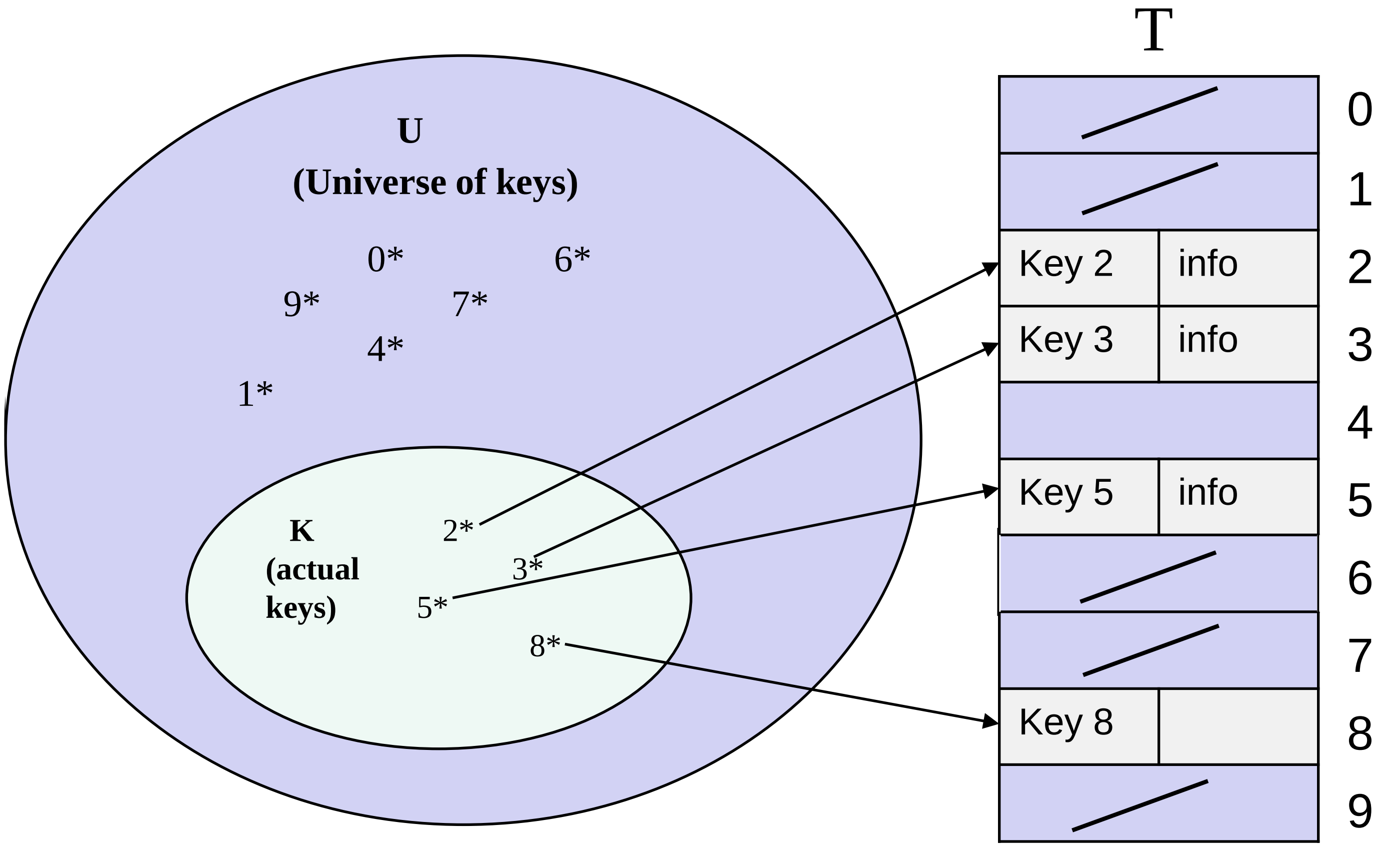
Mapping Data to a Hash Table T: Direkt addressing



- The set K = {2,3,5,8} of actual keys determines the slots in table T that one can consider as pointers to elements (K,I) of the dynamic set T

Hash Tables

Mapping Data to a Hash Table T: <u>Direkt addressing</u>



- The other slots, here blue-shaded, contain pointer NIL

Hashing

Difficulty of direct addressing

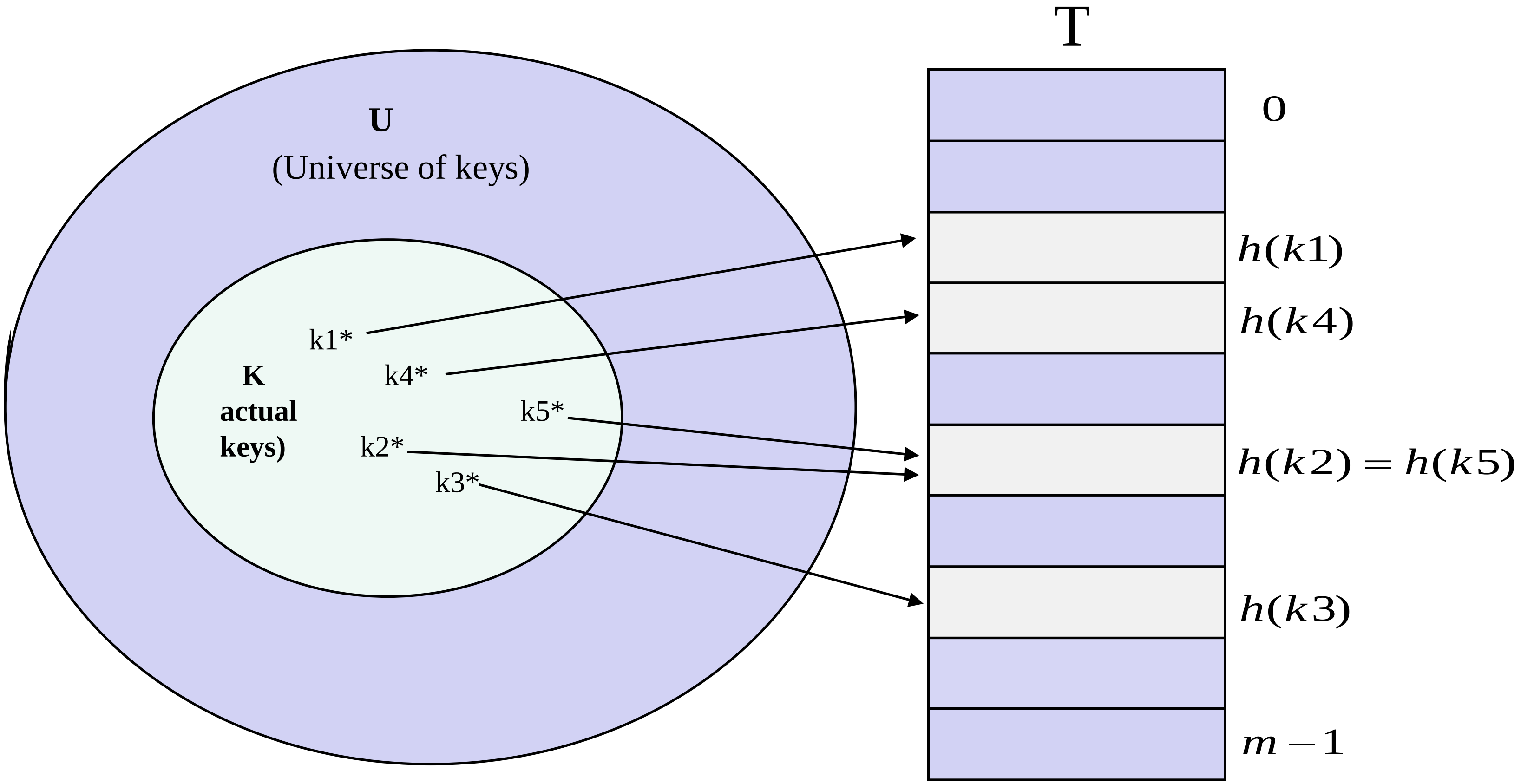
- If universe *U* is very large, storing table *T* of size I*U*I is impractical or impossible
- If set of keys *actually stored* *K*<<*U* - most of space for *T* is wasted
  - Hash table requires much less storage than direct-address table, i.e.
  - The storage requirement can be reduced to $\Theta(|K|)$ while searching for an element takes time O(1) in the average
  - Approach for Non-direct-addressing: **Hash function h**
  - h maps the universe *U* of keys into slots of hash table T[0,…,m-1]:

    $h : U \rightarrow \{0,1,\ldots,m-1\}$
  - **Hashing:** the element with key *k* is stored in slot *h*(*k)* or *h*(*k)* is the ***hash value*** of key *k*

- Basic idea: reduction of I*U*I indices of *T* to only *m* distinct values

Hashing

Using a hash function *h*



- *h* maps keys to hash-table slots
- Here: *k2* and *k5* map to the same slot - *h(k2)* = *h(k5)* → ***Collision***

Hashing & collision resolution

- **Collision**: Two keys may hash to the same slot of *T*, i.e.

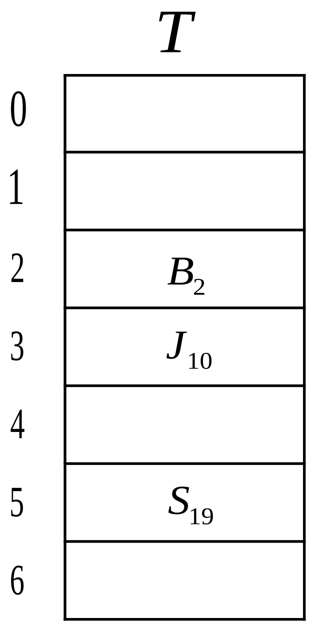
$$h(k_i) = h(k_j) \quad \text{for } k_i \neq k_j, \text{ and } \; \mathrm{i}, \mathrm{j} \in N$$

  - Reason: very often there are many more distinct keys k than table addresses: I*K*I > m of hash table T[0,…,m-1]:
  - Necessary: collision resolution policies
- Of course: best approach would be to avoid collision altogether (so-called ‘perfect hashing’) →
  - Goal to minimize the number of collisions
  - Try to find well-designed *hash functions*
  - A ‘good’ hash function will map the keys uniformly and randomly onto the full range of possible locations in table *T*

Hash functions & collision resolution

Example:

- Take letters of <u>Latin alphabet</u> as <u>keys</u> with subscripts such as A_1, B_2, C_3, R_{18} and Z_{26} where the subscript marks
  - the letter's position in alphabetical order, e.g. S_{19} for letter "S" as the 19^{th} letter in the Latin alphabet
  - tabel *T* contains space for only 7 entries, numbered from 0 to 6
  - in the table are inserted they keys $B_2, J_{10},$ and S_{19} (for simplicity we are ignoring the associated info/satellite data)
  - Which locations of *T* are used for storing $B_2, J_{10},$ and S_{19} ?
  - <u>Answer:</u> Hash function by "Division method"



	T
0	
1	
2	B_2
3	J_{10}
4	
5	S_{19}
6	

- **Hash function: Division method**

$$\boxed{h(k) = k \bmod m \quad \text{for } k \in K}$$

  - Other hash functions are e.g. *<u>Multiplication method</u>* and *<u>Universal hashing</u>*

Hash functions & collision resolution

Hash function: Division method (Example)

- $h(k) = k \bmod m \quad \text{for } k \in K$

T

0	
1	
2	B_2
3	J_{10}
4	
5	S_{19}
6	

  - In the example the locations in *T* for keys B_2, J_{10}, and S_{19} you compute following:

    a) identify the subscripts of the keys with *k* in the formula *h(k)*

    b) divide *k* by *m = 7* and determine the *remainder*

  - $\text{Entry 3 for } J_{10} \text{ is given by: } h(10) = 10 \bmod 7 = 3,$

    $10 : 7 = 1 \text{ remainder } 3$

  - Now lets try inserting the new keys N_{14}, X_{24}, and W_{23} into the table *T;* based on the Division method computation we get for

    $N_{14}: \; h(14) = 14 \bmod 7 = 0, \quad \text{that means slot } T[0]$

  - Because of *T*[0] is an empty slot → no problem !

Hash functions & collision resolution

Hash function: Division method (Example)

- $h(k) = k \bmod m \quad \text{for} \; k \in K$
  - key X_{24} should be placed into slot $h(X_{24}) = 3$
  - but position 3 of *T* already contains key $J_{10} \rightarrow$ Collision

    X_{24} and J_{10} collide at the same *hash address* $3 = h(J_{10}) = h(X_{24})$
  - $\Rightarrow$ Need for
- *Collision resolution policies*
  - A simple heuristic approach:
  - Here for example: Look in table *T* and find the first empty slot at lower location w.r.t. collision position and insert the colliding key
  - If all lower numbered locations are already filled, “wrap around” and start searching for empty locations at the highest numbered location, in (example) location *T*(6)

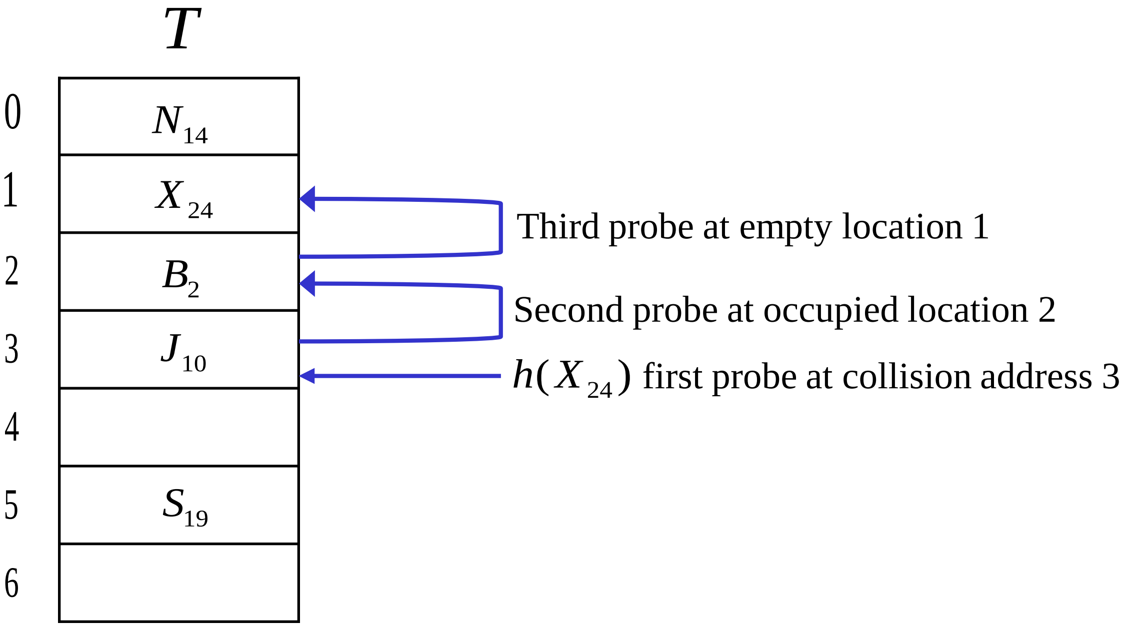
T

0	N_{14}
1	
2	B_2
3	J_{10}
4	
5	S_{19}
6	

Hash functions & collision resolution

Example:

- The results are given in table *T*



T

0	N_{14}
1	X_{24}
2	B_2
3	J_{10}
4	
5	S_{19}
6	

Third probe at empty location 1

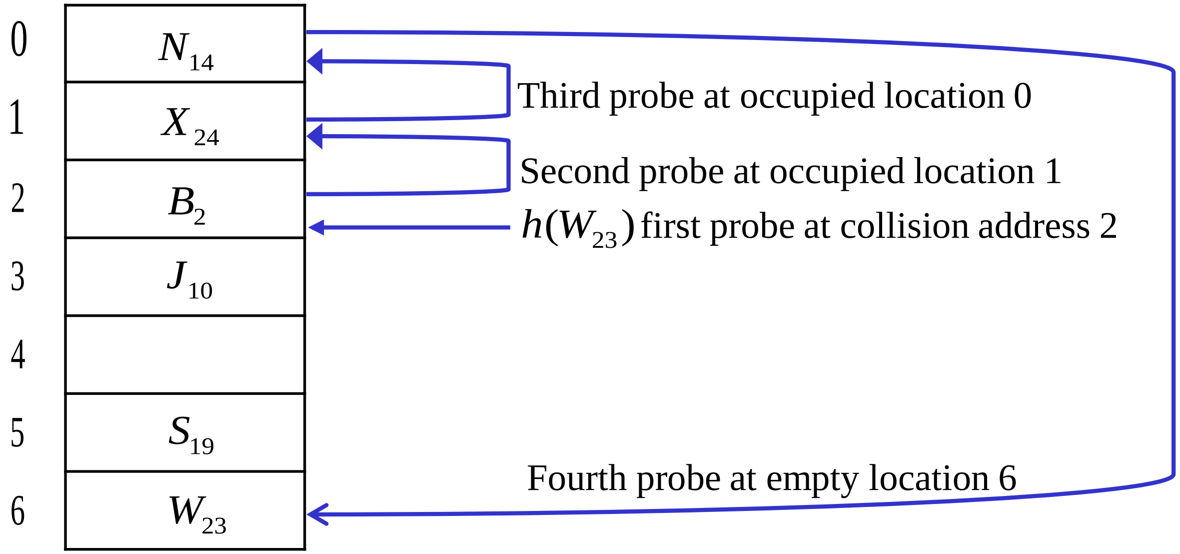
Second probe at occupied location 2

$h(X_{24})$ first probe at collision address 3

Hash functions & collision resolution

Example:

- Finally is to insert W_{23}. Location $h(W_{23}) = 2$ is already occupied by key B_2 Lower positions 1 and 0 are filled, so we wrap around an come to empty slot 6 : Here W_{23} is inserted.



- The locations examined for finding an empty slot are the ***probe sequence***
- The probe sequence for W_{23} is 2,1,0,6,5,4, and 3

Hash functions & collision resolution

Example:

- The heuristic approach for creating the probe sequence of above Example is a special application of so-called *open addressing*

- **Open addressing**

  We successively examine , or *probe*, the hash table *T* until we find an empty slot to put the key

- The probe sequence depends upon the key being inserted
- The hash function is extended to include the probe number as a second input:

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

  - For every key k, the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ must be a permutation of $\langle 0,1,\ldots,m-1 \rangle \quad \Rightarrow$
  - Every hash-table position is eventually considered as a slot for a new key as the table *T* fills up

Hash functions & collision resolution

Open addressing

- Pseudocode of inserting key *k* into hash table *T*

  (assumption: key *k* with no info/satellite data, each slot contains either a key or NIL for being empty)

```
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”
```

Hash functions & collision resolution

Open addressing

- Pseudocode of searching for key k in hash table *T*
  - The algorithm probes the same sequence of slots as in HASH-INSERT

```
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
```

- Analysis of open addressing
  - With assumption of *uniform hashing* (each key is equally likely to have any of the *m*! permutations of (0,1,…,m-1) as its probe sequence)

Average *Runtime is O(1)*

(if *load factor* $\alpha = n/m < 1$ is constant, n - number of elements stored in T, m - number of slots of T)

Hash functions & collision resolution

Open addressing

- Three techniques to compute the probe sequences
- Linear probing, quadratic probing, and double hashing
  - Always is guaranteed that $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ is a permutation of $\langle 0,1,\ldots,m-1 \rangle$
  - Double hashing produces the most probe sequences, and thus gives the best results, generally

- Linear probing
  - $\underline{h(k,i) = (h'(k) + i) \bmod m}$ for $i = 0,1,\ldots,m-1$ with auxiliary hash function $h' : U \rightarrow \{0,1,\ldots,m-1\}$
  - for key k the first slot probed is $T[h'(k)]$, then slot $T[h'(k)+1] \ldots T[\mathrm{h'(k)} + (\mathrm{m}-1)]$, then wrap around to slots $T[0], T[1], \ldots,$ until slot $T[h'(k)-1]$ $\Rightarrow$ for key k are at most m distinct probe sequences

Hash functions & collision resolution

Open addressing

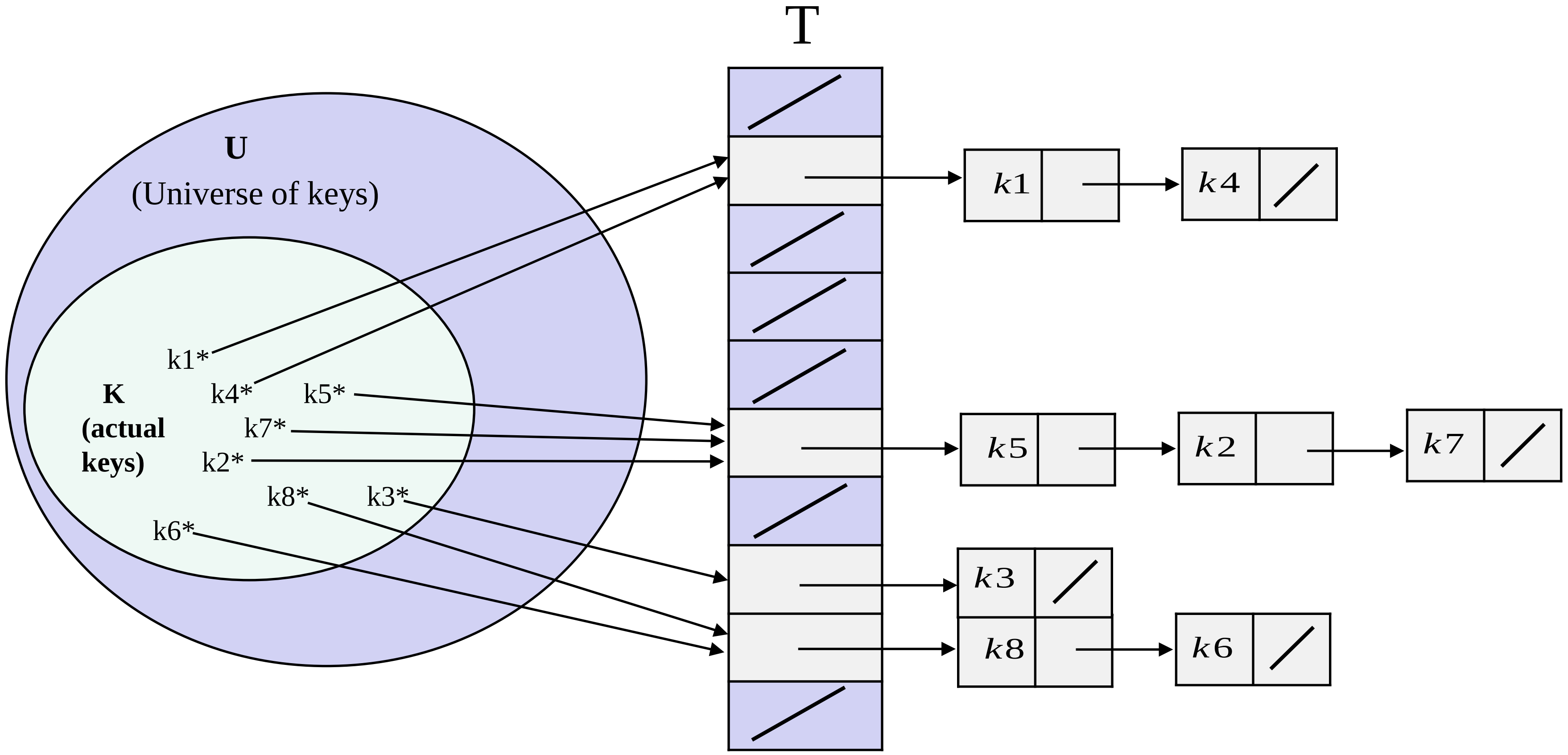
- Quadratic probing
  - $h(k,i) = (h'(k) + c_1 \mathrm{i} + c_2 \mathrm{i}^2) \bmod m$, where h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are constants, $i = 0, 1, \ldots, \mathrm{m} - 1$

- Double hashing
  - $h(k,i) = (h_1(\mathrm{k}) + \mathrm{i}h_2(\mathrm{k})) \bmod m$, where h_1 and h_2 are auxiliary hash functions, $i = 0, 1, \ldots, m-1$
  - The initial probe is to position $T[h_1(\mathrm{k})]$, successive probe positions are offset from previous positions by amount $h_2(\mathrm{k})$, modulo m $\Rightarrow$ Unlike at linear or quadratic probing - probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary

Hash functions & collision resolution

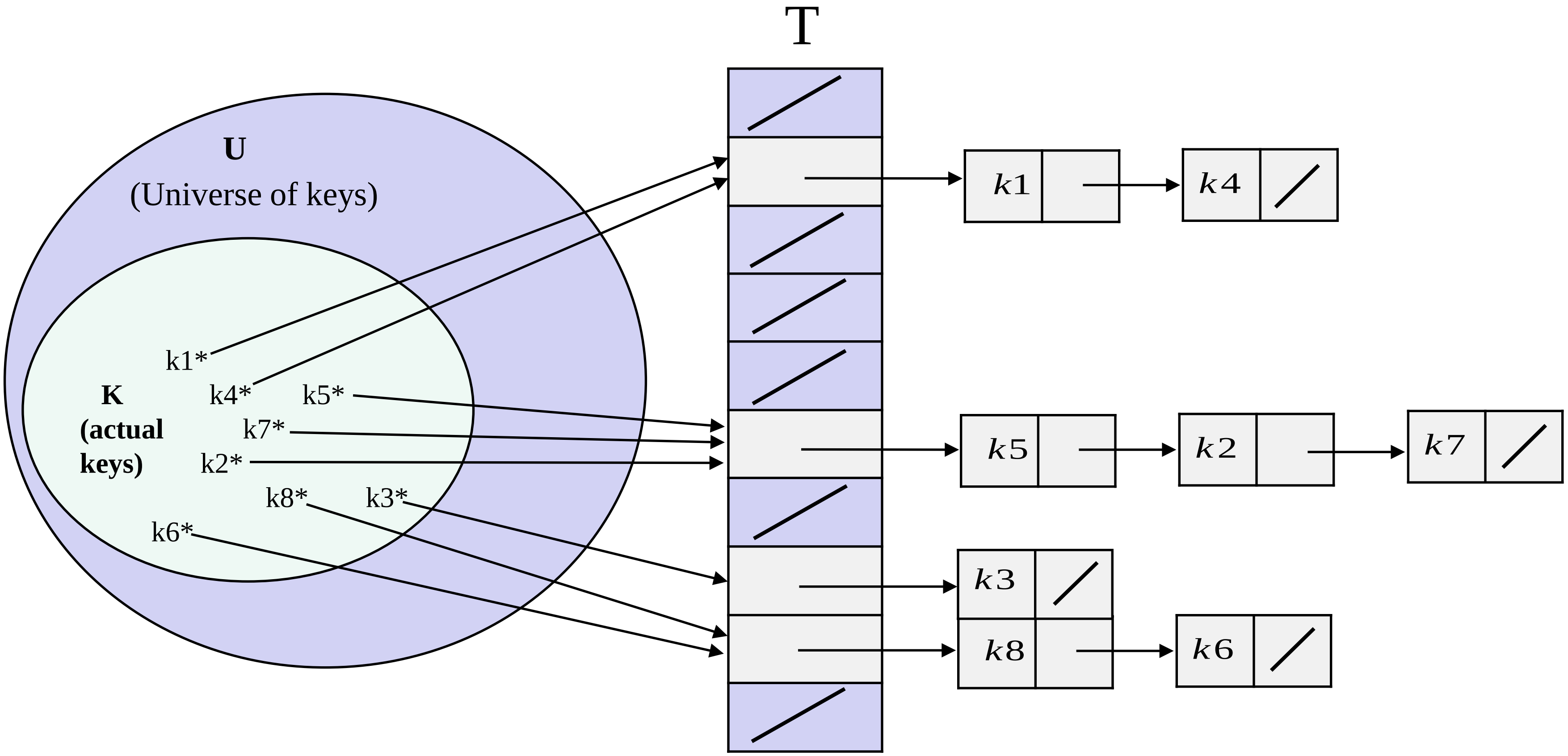
Collision resolution by chaining

- Idea: put all keys that collide at a single hash address on a linked list starting at that address



Hash functions & collision resolution

Collision resolution by chaining



- Each hash-table slot $T[j]$ contains a *linked list* of all the keys whose hash values is j, here $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$, otherwise slot j contains NIL

Hash functions & collision resolution

Collision resolution by chaining

- The dictionary operations on hash table *T* are easy to implement

```
CHAINED-HASH-INSERT(T,x)
    insert x at the head of list T[h(key[x])]

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(key[x])]
```

- The worst-case running time
  - for insertion is *O*(1)
  - for deletion is *O*(1) if the lists are doubly linked
  - for searching is $\Theta(1+\alpha)$, where α is the *load factor* - the average number of elements stored in a chain