Hash Tables

Subhabrata Samajder

IIIT, Delhi

winter Semester,
7^{th} June, 2023

Motivation

- Many applications only require INSERT, SEARCH, and DELETE operations.
- **Example:** A compiler maintains a symbol table, in which the keys of elements are arbitrary character strings that correspond to identifiers in the language.
- It is an effective data structure for implementing dictionaries.
  - SEARCH: Takes as long as searching for an element in a linked list, i.e., $\Theta(n)$ time in the worst case.
  - But under reasonable assumptions, the expected time is $\mathcal{O}(1)$.
- It can be seen as a generalization of arrays.

Why Hash Tables?

- **Direct Addressing:** Makes effective use of our ability to examine an arbitrary position in an array in $\mathcal{O}(1)$ time.
- **Drawback:** Is applicable when we can afford to allocate an array, i.e., one position for every possible key.
- What happens when the number of keys actually stored is *small* relative to the total number of possible keys?
  - Hash tables are an effective alternative to directly addressing.
  - Uses an array of size proportional to the $\#$ keys actually stored.
- **Index:** Computed from the key.

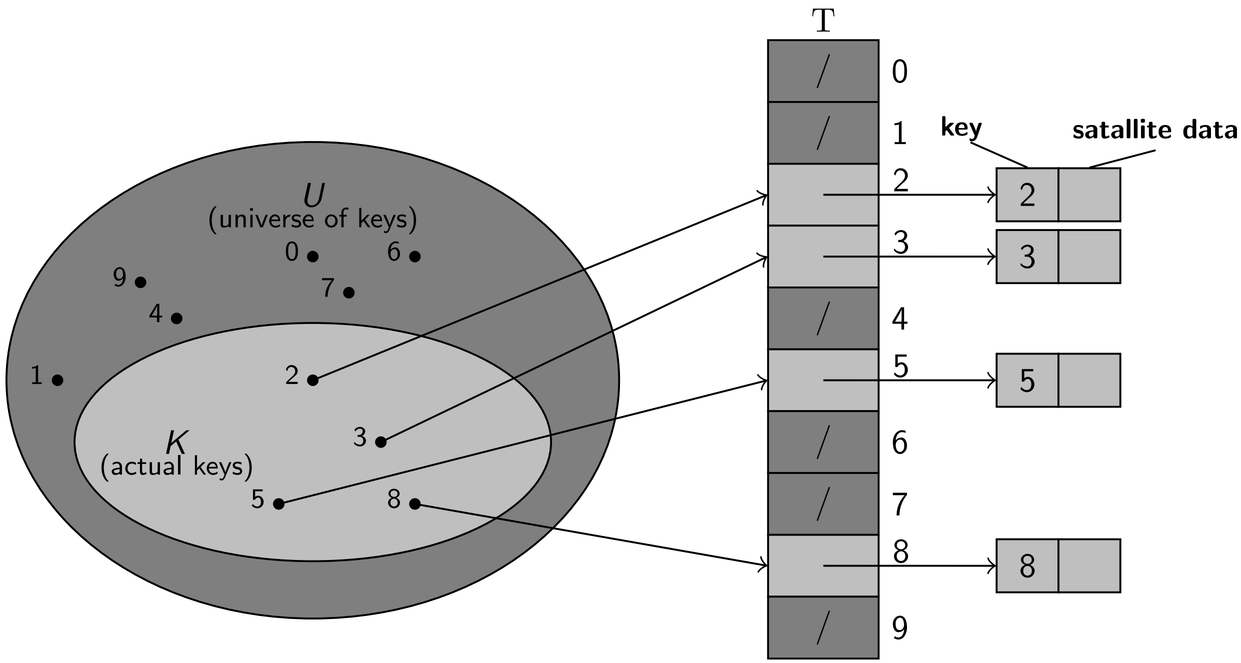
Direct Addressing

Direct Addressing

- A simple technique that works well when the universe $U = \{0, 1, ..., m-1\}$ of keys is reasonably small.

- **Assumptions:**
  - m is not too large.
  - **No Collision:** No two elements have the same key.

Direct Addressing (Cont.)



DIRECT-ADDRESS-SEARCH(T, k)

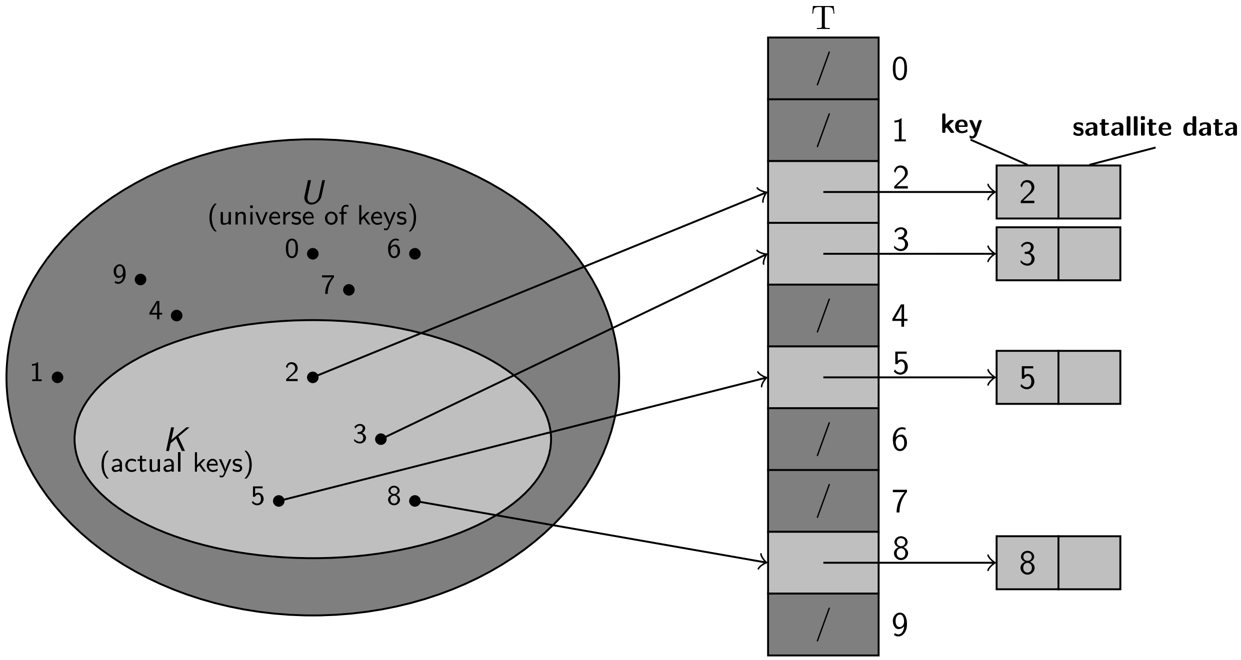
I/P: A direct-address table T and a key k.

O/P: T[k] if the key exists, else NIL.

return T[k]

Complexity: $\mathcal{O}(1)$.

Direct Addressing (Cont.)



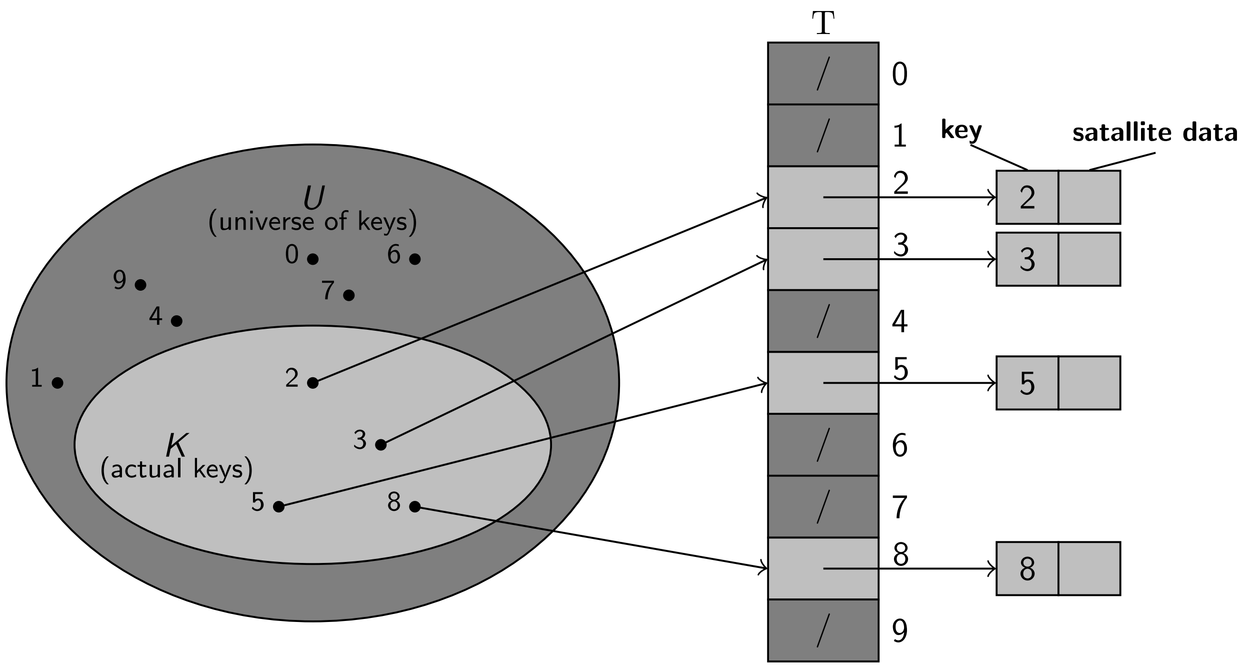
Direct-Address-Insert(T, x)

I/P: A direct-address table T and an element x.

$\mathrm{T}[key[x]] \leftarrow x$

Complexity: $\mathcal{O}(1)$.

Direct Addressing (Cont.)



Direct-Address-Delete(T, x)

I/P: A direct-address table T and an element x.

$T[key[x]] \leftarrow$ NIL

Complexity: $\mathcal{O}(1)$.

Drawbacks

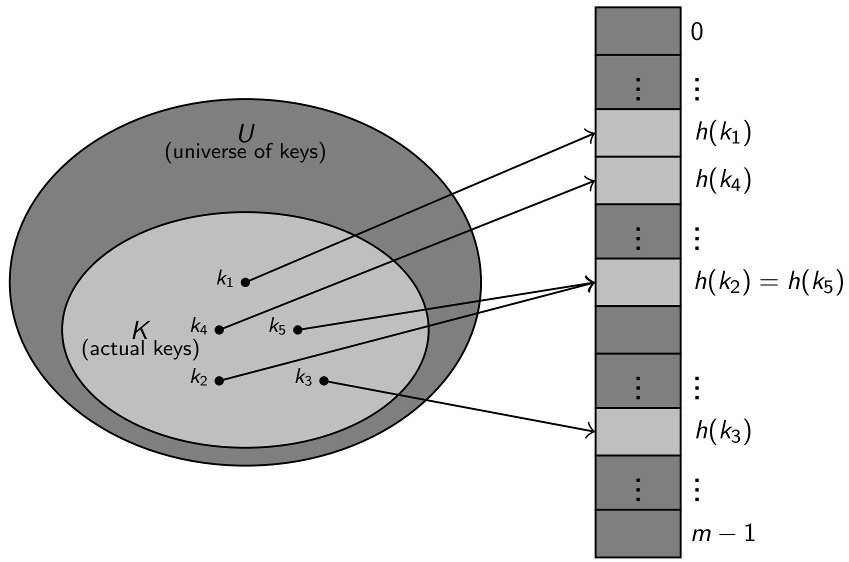
- If the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible.
- If $|U| \gg |K|$, then most of the space allocated for T would be wasted.

Hash Tables

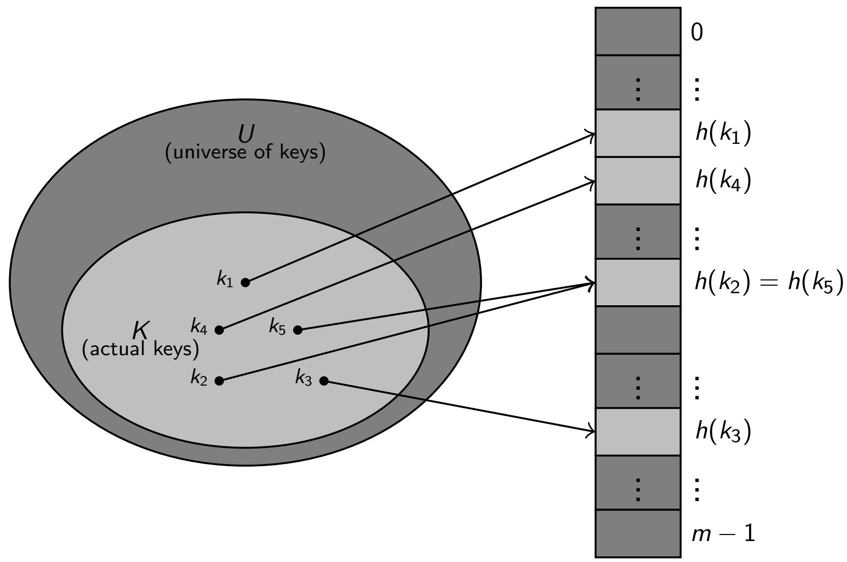
Hash Table

- Reduces storage requirements to $\Theta(|K|)$.
- But retains the advantage that searching for an element requires $\mathcal{O}(1)$ time on average.
- **Note:** The bound is for the average time, whereas for direct addressing it holds for the *worst-case time*.
- **Direct addressing:** An element with key k is stored in slot k.
- **Hash Table:**
  - Key k is stored in slot $h(k)$, where $h : U \to \{0, 1, \ldots, m-1\}$ is called a *hash function*.
  - The hash table $\mathrm{T}[0 \ldots m-1]$ has m slots.
- Only requires m values instead of $|U|$ values earlier.
- Thereby reducing the storage requirements.

Hash Table (Cont.)

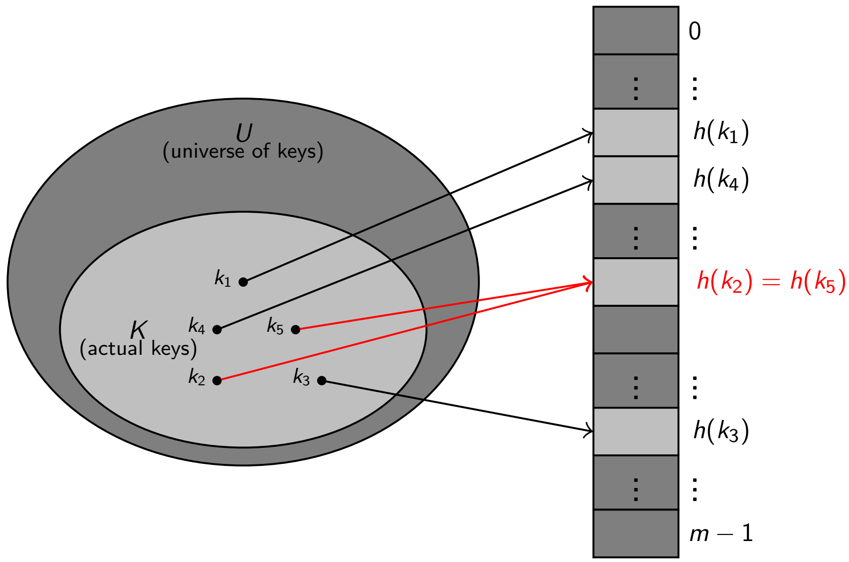


Hash Table (Cont.)



Any Problem with this approach?

Hash Table (Cont.)



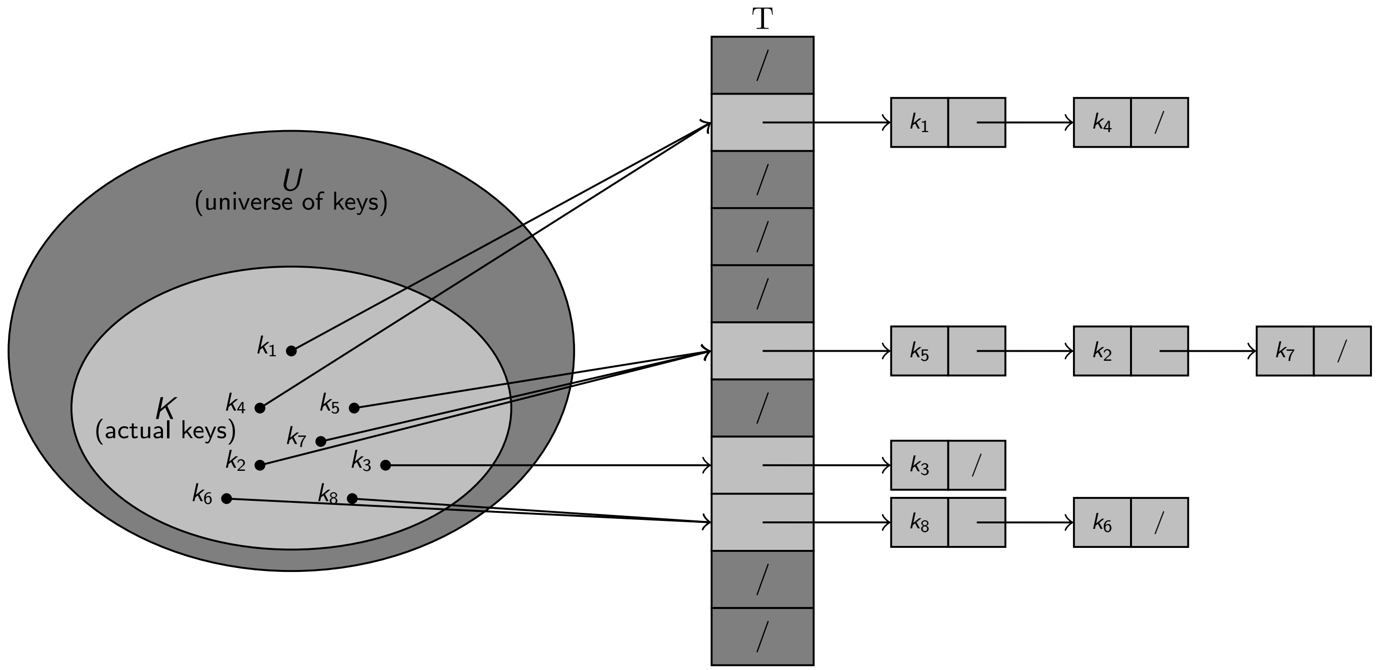
Any Problem with this approach?

- Collisions: Two keys may hash to the same slot.

Ways to Avoid Collision

- Make h appear to be "(uniformly) random".
- Thereby avoiding collisions or at least minimizing their number.
- But a hash function h must be deterministic!
- $|U| > m$ and h is an onto function $\Rightarrow$ collisions are inevitable!
- Therefore avoiding collisions altogether is impossible.
- A well-designed, "random-looking" hash function can minimize the number of collisions.
- But we still need a method for resolving the collisions that do occur.

Collision Resolution by Chaining

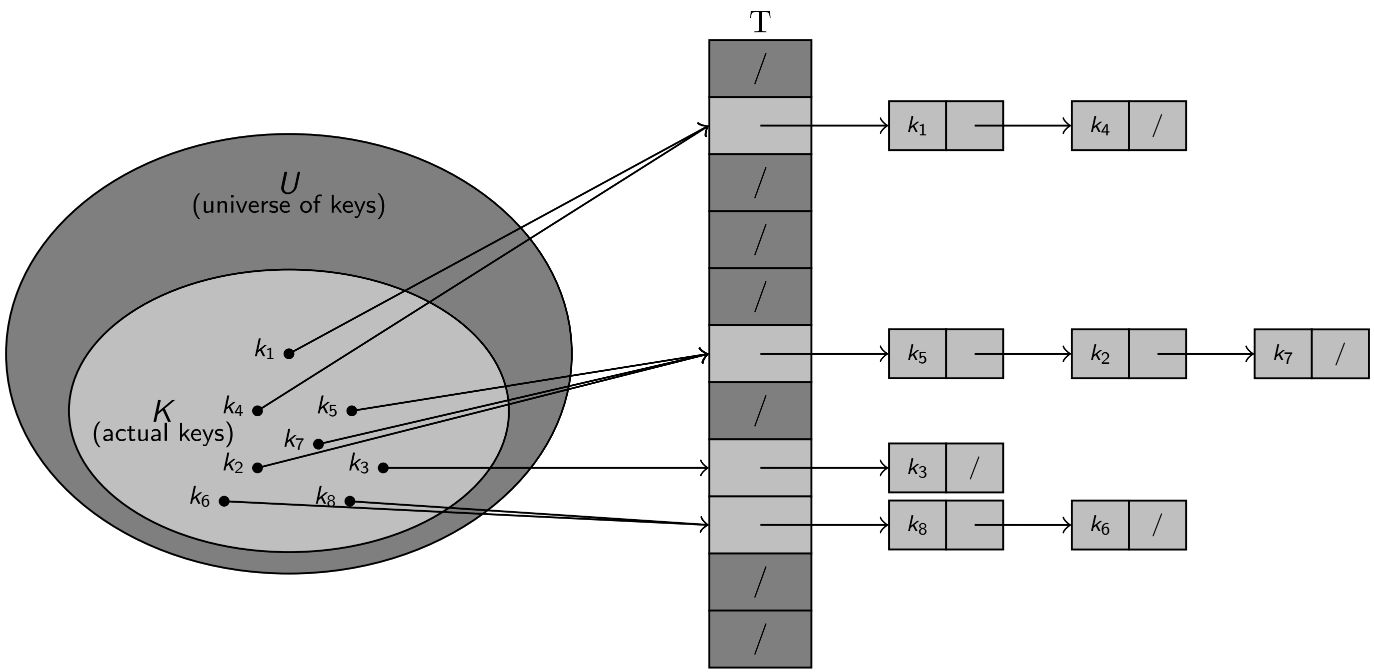


CHAINED-HASH-INSERT(T, x)

I/P: A hash table T and an element x.

insert x at the head of list $\mathrm{T}[h(key[x])]$

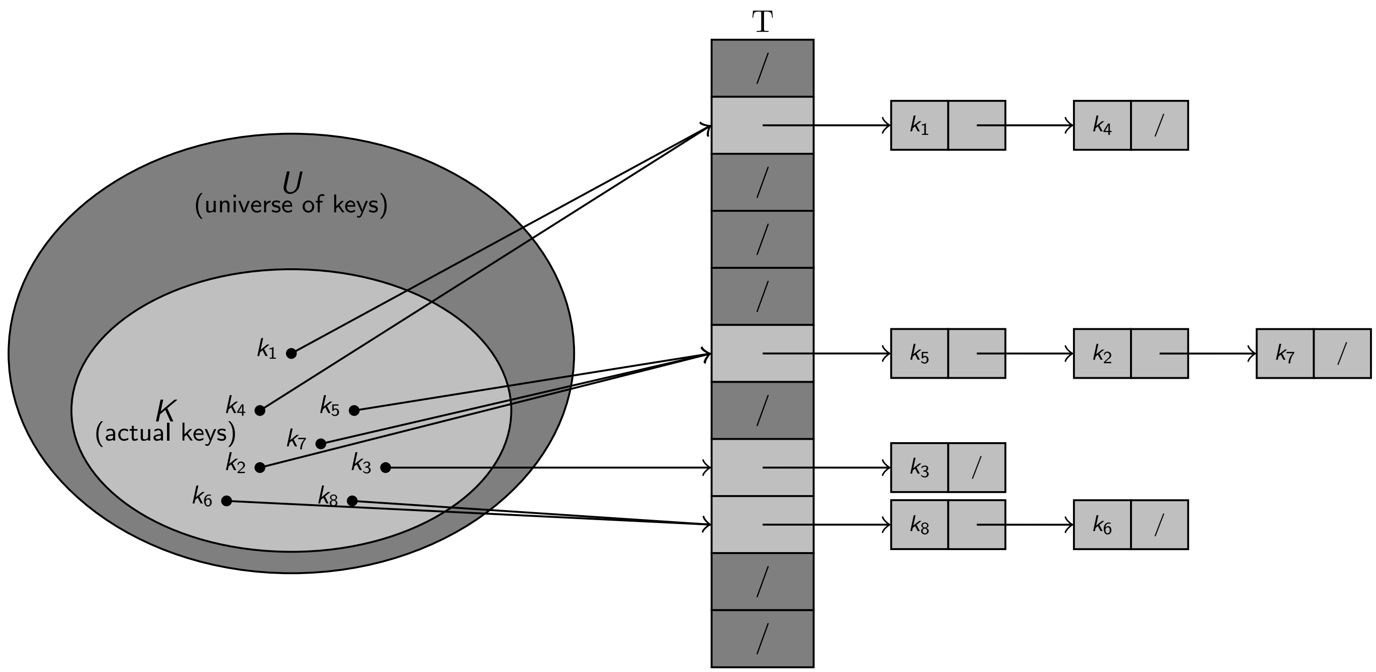
Collision Resolution by Chaining



Complexity: $\mathcal{O}(1)$

- **Note:** It is fast, as it assumes that the element x is not present in the table.
- If required, repetition can be prevented (at additional cost) by performing a search before insertion.

Collision Resolution by Chaining



CHAINED-HASH-SEARCH(T, k)

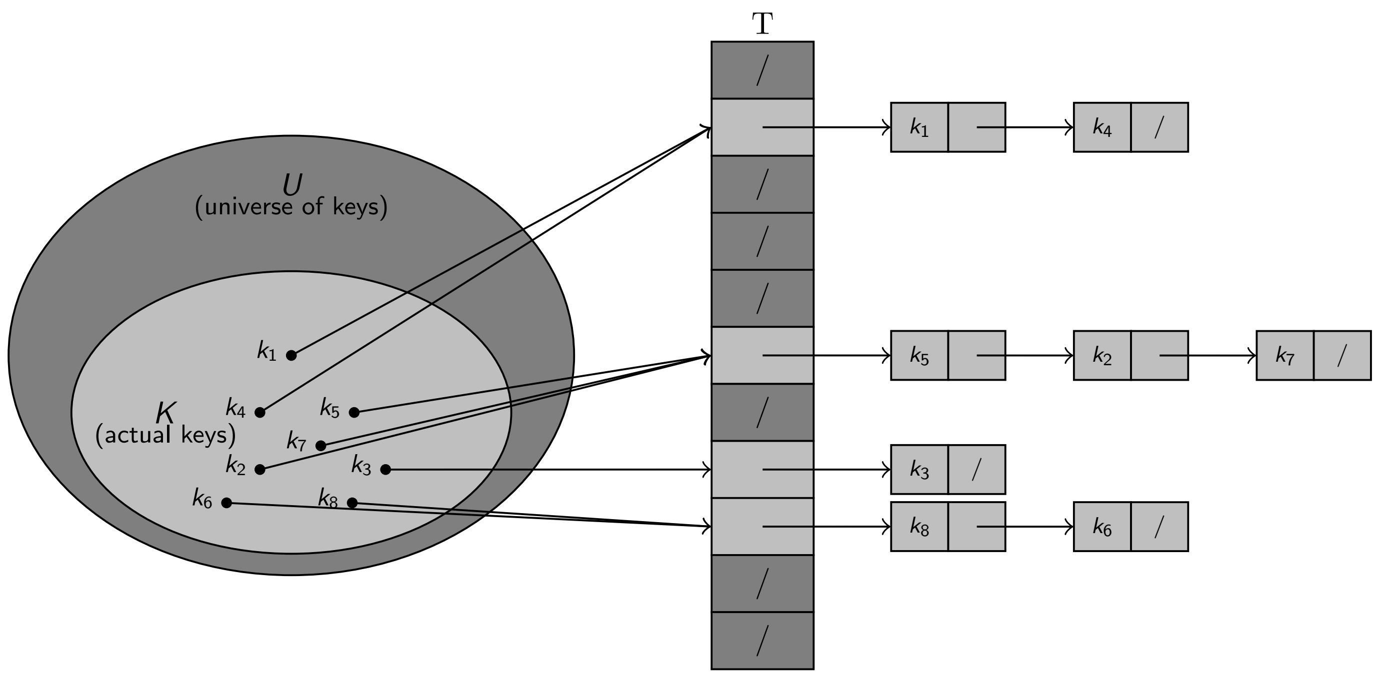
I/P: A hash table T and a key k.

O/P: T[k] if the key exists, else NIL.

search for an element with key k in list T[$h(k)$]

Worst-case Complexity: Proportional to the length of the list.

Collision Resolution by Chaining

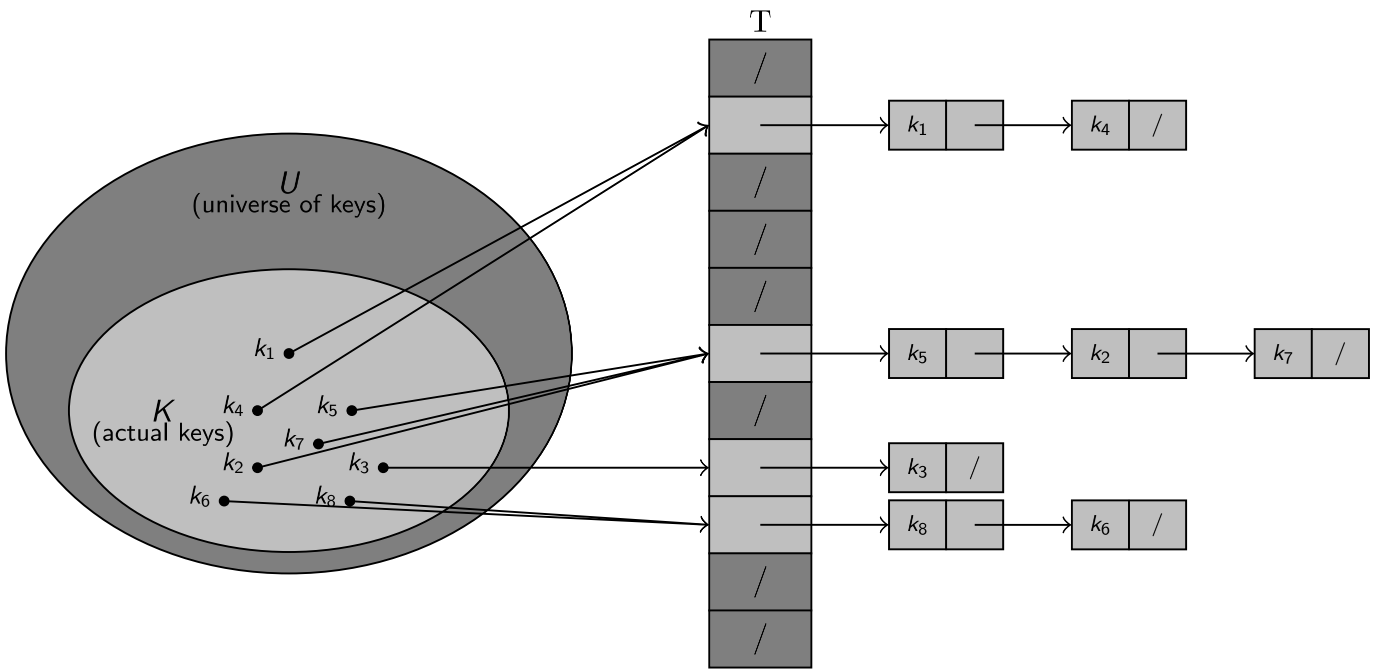


$\textsc{Chained-Hash-Delete}(\mathrm{T}, x)$

I/P: A hash table T and an element x.

delete x from the list $\mathrm{T}[h(key[x])]$

Collision Resolution by Chaining



- $\mathcal{O}(1)$: If the lists are doubly linked.
- **Note:** Takes as input an element x and not its key $k \Rightarrow$ need to search for x first.
- Singly Linked List: It would not be of great help to take as input the element x rather than the key k, since we cannot access the previous node.
- Still have to find x in the list $T[h(key[x])]$, so that the next link of x's predecessor could be properly set to splice x out.
- Therefore, deletion and searching has essentially the same running time.

How Well Does Hashing With Chaining Perform?

Question: How long does it take to search for an element with a given key?

Load Factor α: $\alpha \triangleq \frac{n}{m}$, where m denotes the # slots in T and n denotes the # elements stored in T.

- α represents the average number of elements stored in a chain.

Worst-case Behaviour:

- All n keys hash to the same slot, creating a list of length n.
- **Complexity:** $\Theta(n)$ plus the time to compute the hash function.

Note: Hash tables are not used for their worst-case performance.

Although **Perfect hashing** does however provide good worst-case performance when the set of keys is static.

Simple Uniform Hashing

Definition (Simple Uniform Hash)

A hash function $h : U \to \{0, 1, \ldots, m-1\}$ is called a simple uniform hash, if the following holds.

- Any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.
- That is, for any two elements $x \neq y$, where y is hashed after x

  $$\Pr[h(key[x]) = i] = \frac{1}{m}, \forall\, i \in \{0, 1, \ldots, m-1\}$$

  and the events "$\{h(key[x]) = i\}$" and "$\{h(key[y]) = j\}$" are mutually independent.

For $j = 0, 1, \ldots, m-1$, let $N_j =$ "length of the list $\mathrm{T}[j]$", so that

$$N_0 + N_1 + \cdots + N_{m-1} = n.$$

Simple Uniform Hashing

Consider all the keys $k_1, \ldots, k_n$ in this order.

Then for all $j = 0, 1, \ldots, m-1$ and for all $i = 1, 2, \ldots, n$, we have

$$\Pr[h(k_i) = j] = \frac{1}{m},$$

Define, $X_{ij} \triangleq \begin{cases} 1 & \text{if } h(k_i) = j \\ 0 & \text{if } h(k_i) \neq j. \end{cases}$

Then by the assumption of simple uniform hash $\Pr[X_{ij} = 1] = \frac{1}{m}$.

Therefore, $N_j = \sum_{i=1}^{n} X_{ij}$ follows $\mathrm{Bin}\left(n, \frac{1}{m}\right)$.

Thus, $E[N_j] = n \times \frac{1}{m} = \alpha$, for all $j = 0, 1, \ldots, m-1$.

Average-case Complexity For Search and Deletion

Assumption: It takes $\mathcal{O}(1)$ time to compute the hash function and $\mathcal{O}(1)$ to access slot $h(k)$.

Theorem

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes expected time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Theorem

In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.

What does this analysis mean? If $n = \mathcal{O}(m)$, then $\alpha = \mathcal{O}(1)$, which implies that all the dictionary operations can be performed in $\mathcal{O}(1)$ time.

Exercises

1. If $h(k) = \lfloor km \rfloor$, where $k \overset{iid}{\sim} \mathcal{U}((0,1))$, then show that $h(k)$ satisfies the condition of simple uniform hashing.

Hash Functions

What Makes a Good Hash Function?

- Those which approximately satisfies the assumption of simple uniform hashing.
- Unfortunately, it is typically not possible to check this condition.

  Note:

  - One rarely knows the probability distribution according to which the keys are drawn.
  - The keys may not be drawn independently.
- Occasionally we do know the distribution (see Exercise 1).

What Makes a Good Hash Function?

- In practice, *heuristic techniques* can often be used to create a hash functions that performs well.

- Qualitative information about distribution of keys may be useful in this design process.

  A compiler's symbol table example:

  - Keys are identifiers (character strings).
  - A good hash function would minimize the chance that closely related symbols ('pt' and 'pts') hash to the same slot.

- **A good approach:** The hash value is expected to be independent of any patterns that might exist in the data.

The Division Method

$$h(k) = k \mod m,$$

Note:

- m should not be a power of 2:
  - If $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k.
  - Unless all low-order p-bit patterns are equally likely, its better to make the hash function depend on all the bits of the key.

- A good choice of m: A prime not too close to an exact power of 2.

The Division Method

$$h(k) = k \mod m,$$

Advantages:

- Quite fast since it requires only a single division operation.

Disadvantages:

- Depends on the value of m.
- Certain values of m are bad.
  - power of 2
  - non-prime numbers

The Multiplication Method

- Multiply the key k by a constant $A \in (0, 1)$.
- Extract the fractional part of kA.
- Then, multiply this value by m and take the floor of the result.
- In short, $h(k) = \lfloor m(kA \mod 1) \rfloor$.
- This method works for any value of the constant A.
- But it works better with some values than with others.
- The optimal choice depends on the characteristics of the data being hashed.
- Knuth suggests that $A \approx \frac{\sqrt{5}-1}{2} = 0.6180339887\ldots$ is likely to work reasonably well.

Advantage:

- The value of m is not critical.

Open Addressing

Open Addressing

- All elements are stored in the hash table itself.
- That is, each table entry is either an element or NIL.
- SEARCH: Systematically examine table slots until the element is found or it is clear that the element is not in the table.
- There are no lists and no elements stored outside the table.
- Thus, when the table "fills up" $\Rightarrow$ no insertions are possible $\Rightarrow$ $\alpha \leq 1$ always.

Open Addressing

- **Note:** One can store the linked lists for chaining inside the hash table, i.e., on the unused hash-table slots.

- **Advantage:** *Avoids pointers altogether*.
  - Instead *compute* the sequence of slots to be examined.

- The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

Insertion

- Successively examine, or probe, the hash table until an empty slot in which to put the key is found.
- Instead of a fixed sequence $0, 1, \ldots, m-1$ ($\Theta(n)$ time), the probing sequence *depends upon the key being inserted*.
- To do this, extend the hash function to include the probe number (starting from 0) as a second input.
- Thus, $h : U \times \{0, 1, \ldots, m-1\} \mapsto \{0, 1, \ldots, m-1\}$.
- $\therefore$ for every key k, the probe sequence

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

is a *permutation* of $\langle 0, 1, \ldots, m-1 \rangle$ $\Rightarrow$ every hash-table position is eventually considered.

Insertion

Assumptions:

- The elements in T are keys with no satellite information.
- The key k is identical to the element containing key k.
- Each slot contains either a key or NIL.

HASH-INSERT(T, k)

I/P: A hash table T and a key k.

```
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"
```

Searching

- For a key k the searching algorithm should probe the same sequence of slots that were examined when k was inserted.

HASH-SEARCH(T, k)

I/P: A hash table T and a key k.

O/P: j if slot j is found to contain key k, else NIL.

$i \leftarrow 0$
repeat $j \leftarrow h(k, i)$
  if $(\mathrm{T}[j] = k)$
    return j
  $i \leftarrow i + 1$
until $\mathrm{T}[j] =$ NIL or $i = m$

return NIL

Deletion

- Deletion from an open-address hash table is difficult.
- When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it.
- This might make it impossible to retrieve any key k during whose insertion we had probed slot i and found it occupied.
- A solution: Mark by storing a special value DELETED instead of NIL.
- Then modify the HASH-INSERT procedure to treat such a slot as empty so that a new key can be inserted.
- No modification of HASH-SEARCH is needed, since it will pass over DELETED values while searching.
- But now search times are no longer dependent on α.
- To avoid this *chaining is more commonly selected as a collision resolution technique when keys must be deleted*.

Uniform Hashing

Assumption:

- Uniform Hashing: For analysis purpose, it is assumed that each key is equally likely to have any of the $m!$ permutations.
- Generalizes the notion of simple uniform hashing to the situation where the hash function produces a whole probe sequence instead of a single number.

Note:

- True uniform hashing is difficult to implement.
- In practice the following 3 suitable approximations are used.
  1. Linear Probing
  2. Quadratic Probing
  3. Double Hashing.

- They all guarantee that $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ is a permutation for each key k.
- None of them are a Uniform Hash:
  - Capable of generating at most m^2 different probe sequences
  - In contrast, uniform hashing requires $m!$ different probes.
- Double hashing:
  - Has the greatest number of probe sequences.
  - Seems to give the best results.

Linear Probing

Given: An auxiliary hash function $h' : U \mapsto \{0, 1, \ldots, m-1\}$.

Linear probing: $h(k, i) = (h'(k) + i) \mod m$ for $i = 0, 1, \ldots, m-1$.

Probing Sequence: $\langle \mathrm{T}[h'(k)], \mathrm{T}[(h'(k)+1) \mod m], \ldots, \mathrm{T}[(h'(k)+m-1)) \mod m] \rangle$.

Note:

- Initial probe determines the entire probe sequence.
- Thus, there are only m distinct probe sequences.

Advantage: Easy to implement.

Disadvantage:

- Suffers from the problem of primary clustering.
  - Long runs of occupied slots build up, increasing the average search time.
  - Clusters arise since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$.
  - Long runs of occupied slots tend to get longer, and the average search time increases.

Quadratic Probing

Quadratic probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$, where c_1 and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m-1$.

Probing Sequence: $\langle \mathrm{T}[h'(k)], \mathrm{T}[(h'(k)+c_1+c_2) \mod m], \mathrm{T}[(h'(k)+2c_1+4c_2) \mod m], \ldots, \mathrm{T}[(h'(k)+(m-1)c_1+(m-1)^2 c_2) \mod m] \rangle$.

Advantage:

- Works much better than linear probing.
- But to make full use of the hash table, the values of c_1, c_2, and m are constrained.

Disadvantage:

- Secondary Clustering: $h(k_1, 0) = h(k_2, 0) \Rightarrow h(k_1, i) = h(k_2, i)$ for all i.
- Like linear probing, the initial probe determines the entire sequence, so only m distinct probe sequences are used.

Double Hashing

Double Hashing: $h(k, i) = (h_1(k) + ih_2(k)) \mod m$, where h_1 and h_2 are auxiliary hash functions.

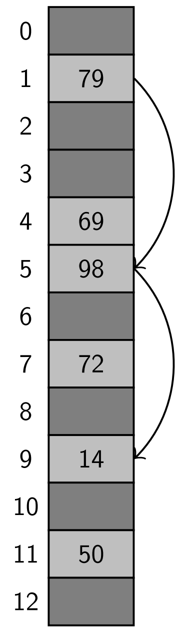
Probing Sequence:

- Initial probe is to position $\mathrm{T}[h_1(k)]$.
- Successive probe positions are offset of $h_2(k) \mod m$ from the previous position.

Advantage:

- The permutations produced have many of the characteristics of randomly chosen permutations.
- Unlike linear or quadratic probing, the probe sequence here depends in two ways upon the key k.
- So $\Theta(m^2)$ distinct probes.
- Avoids clustering.

Double Hashing



- $m = 13$, $h_1(k) = k \mod 13$, $h_2(k) = 1 + (k \mod 11)$.
- **Example:**
  - $h_1(14) = 14 \mod 13 = 1$ and $h_2(14) = 1 + (14 \mod 11) = 4$
  - $\therefore\ h(14,0) = h_1(14) = 1$, $h(14,1) = (1 + 1 \cdot 4) \mod 13 = 5$, $h(14,2) = (1 + 2 \cdot 4) \mod 13 = 9$.

Books and Other Materials Consulted

1. *Introduction to Algorithms* by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein.

Thank You for your kind attention!

Questions!!