Hash Tables

Juan Mendivelso

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

CONTENTS

1. Dictionaries

1. Dictionaries

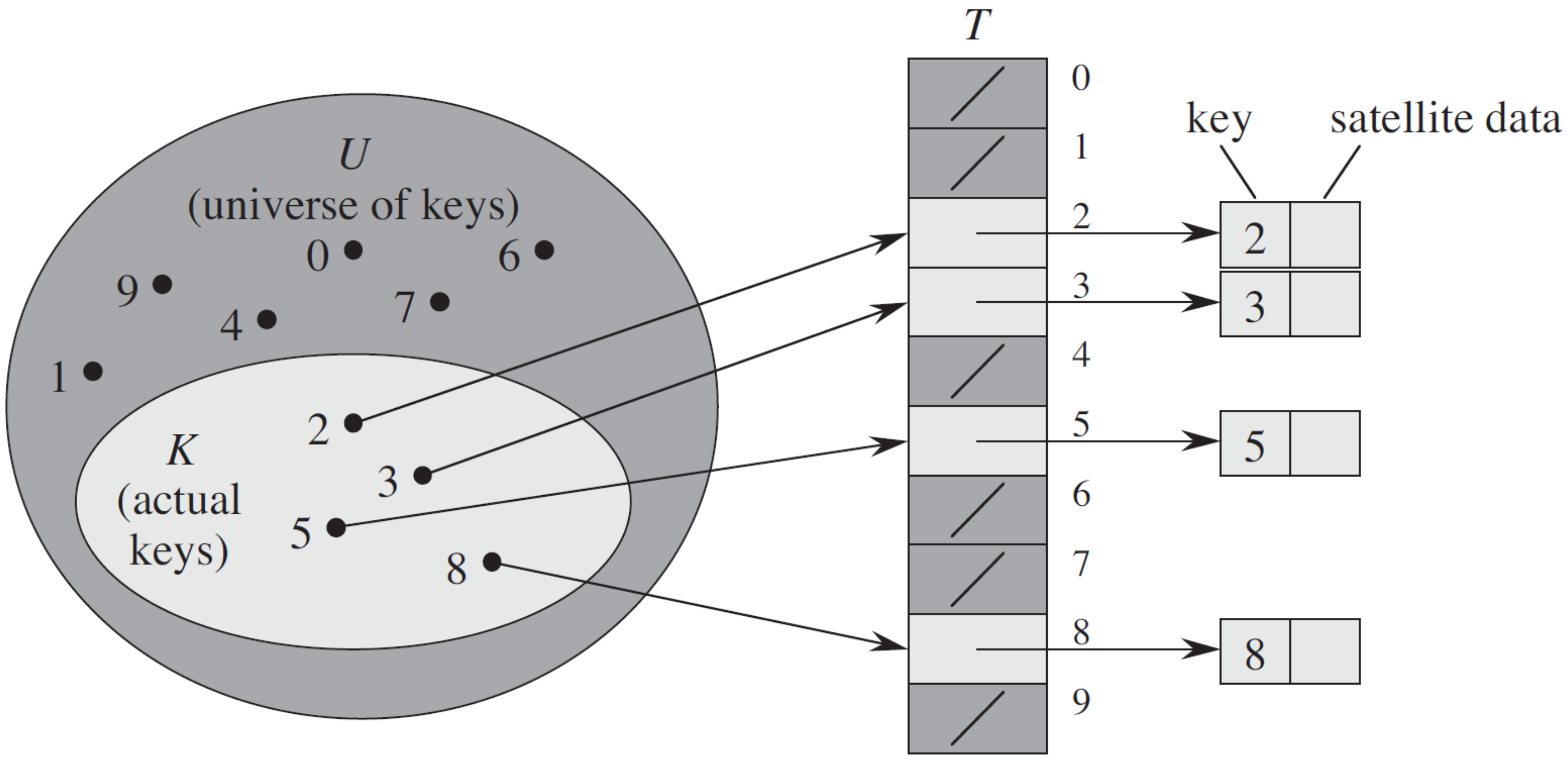
- A set is a collection of elements.
- In Computer Science, we are mainly interested in dynamic sets: there are insertions, deletions, updates and searches.
- Usually, each element is stored as a **register**. It is uniquely identified by a **key** and contains additional information called **satellite data**.
- Many applications require a dynamic set that supports only the dictionary operations (Insert, Search & Delete).
- For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language.

2. Direct-Address Tables

Direct-Address Tables

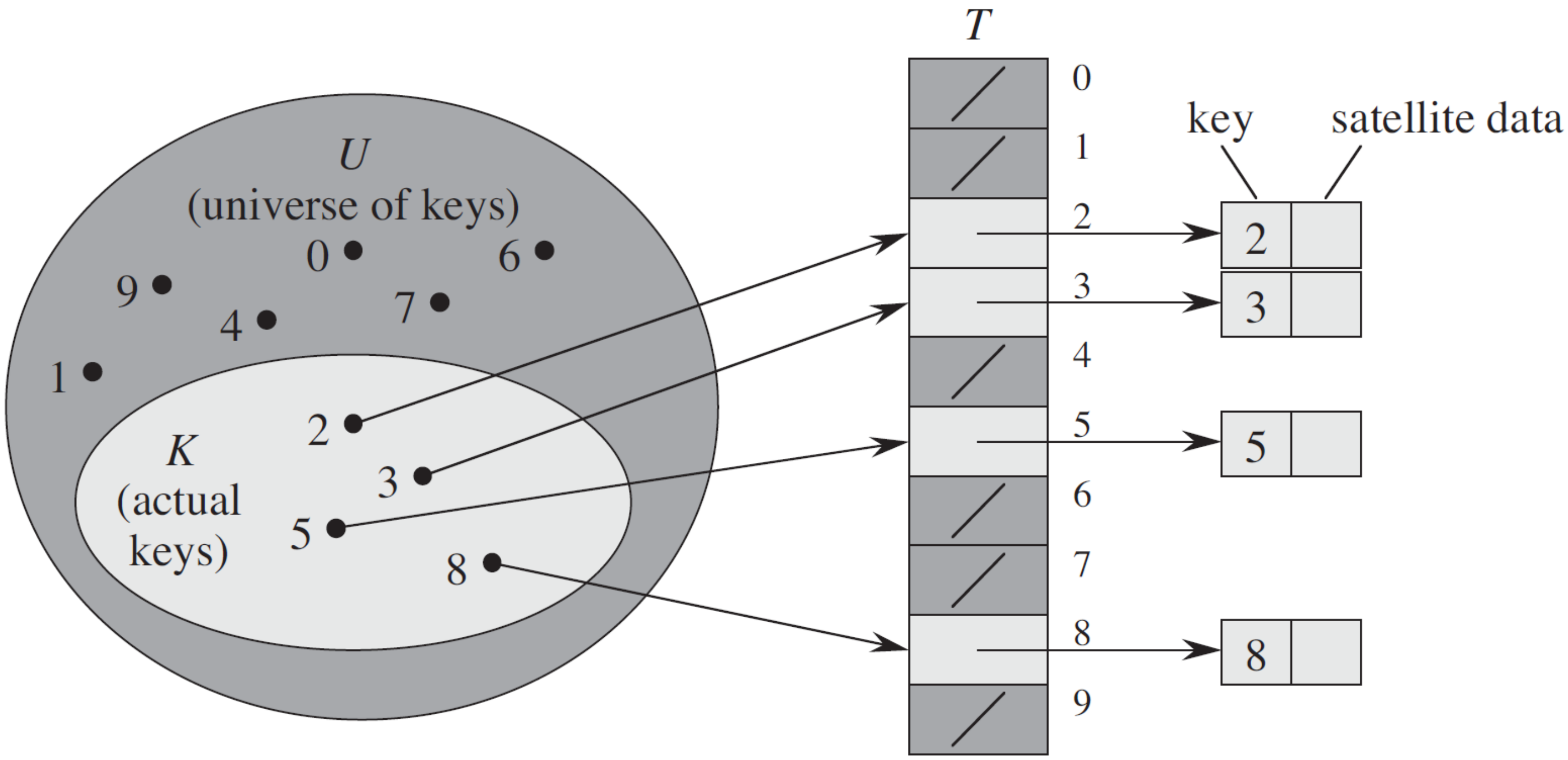
- It allows to represent dynamic sets.
- It works well when the Universe U of keys is reasonably small.
- Each element has a key drawn from U = {0,1,…,m-1}.
- The **direct-address table**, denoted by T[0..m-1], is an array in which each position, called **slot**, corresponds to a key in the universe U.
- If the table does not contain an element with key k, T[k] = NIL.

Direct-Address Tables



Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

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
```

Direct-Address Tables

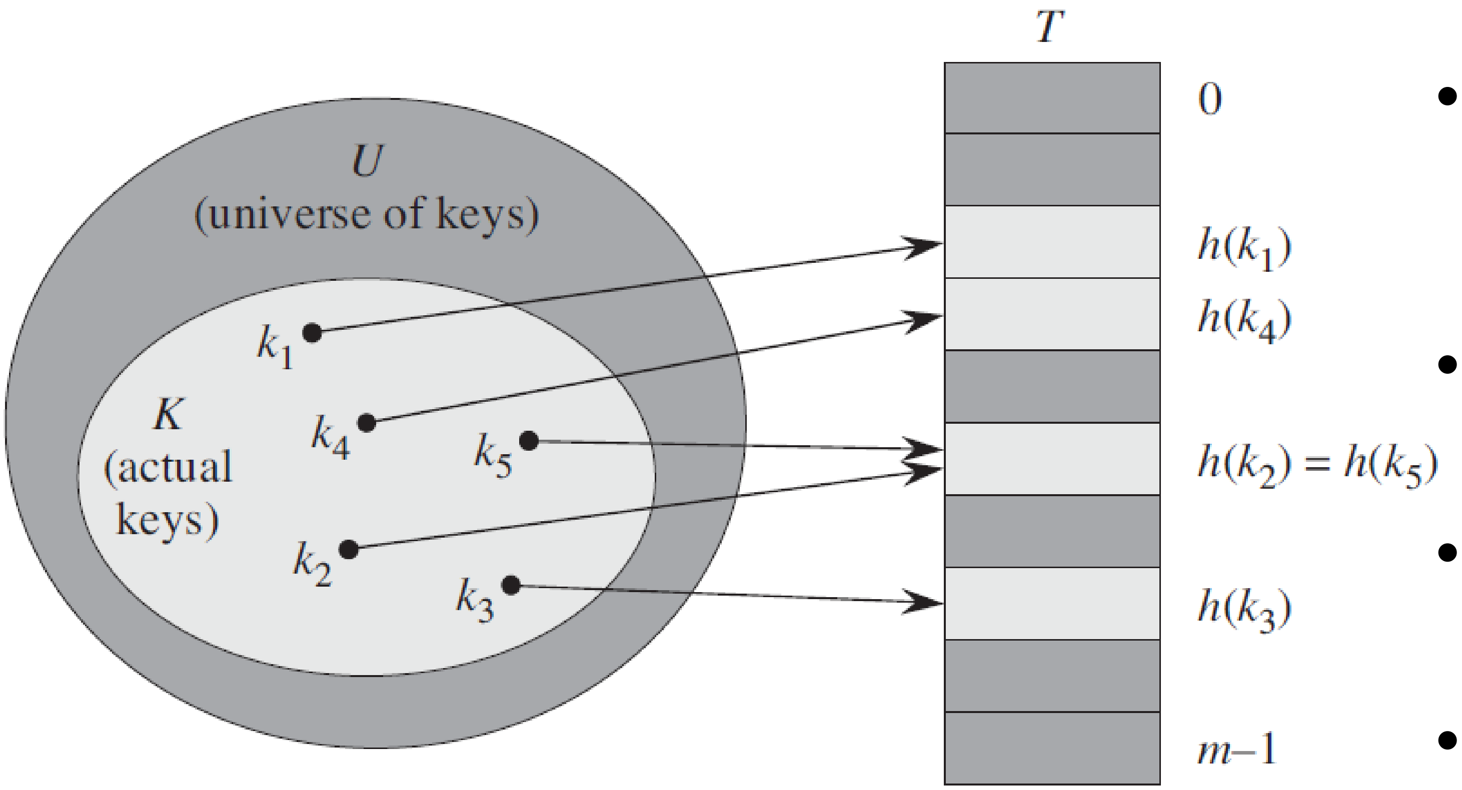
- The downside of direct addressing is obvious: If the universe U is large, storing a table T of size |U| may be impractical, or even impossible, given the memory available.
- Moreover, the set K of keys stored in a dictionary is much less smaller than the universe U of all possible keys.
- Solution: a hash table.

3. Hash Tables

Hash Tables

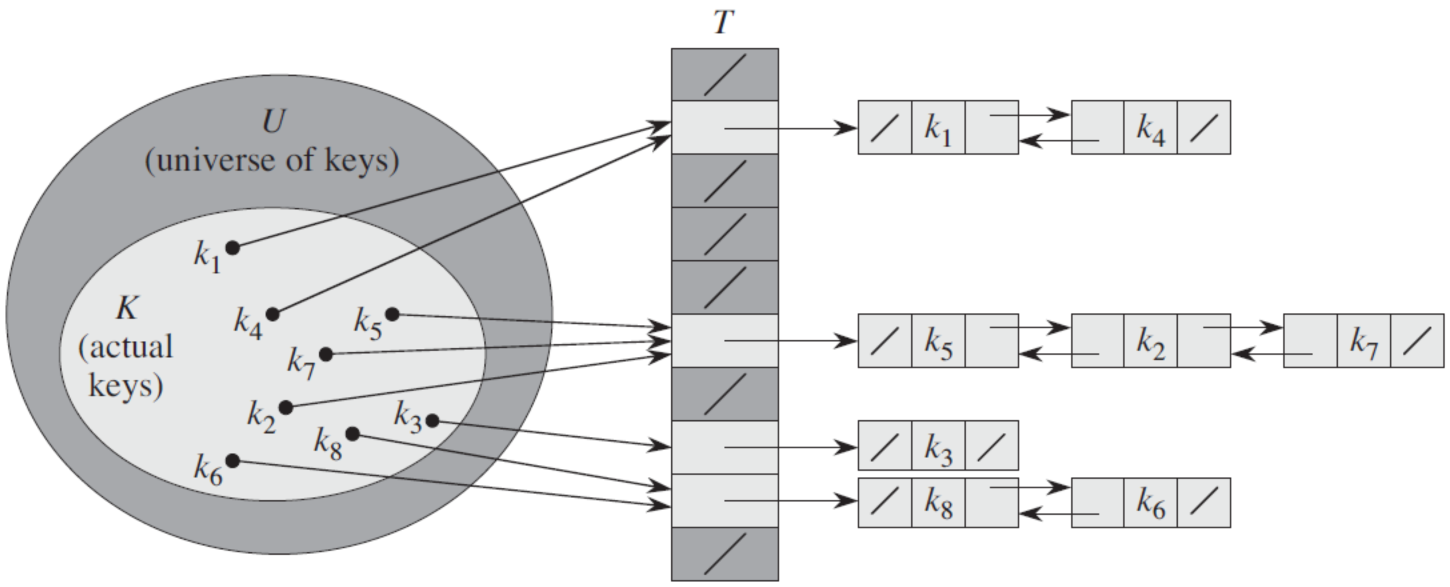
- A **hash table** requires much less storage than a direct-address table.
- We can reduce the storage to $\theta(|K|)$ while we maintain the benefit that searching for an element only takes $O(1)$ (on the average case).
- With direct addressing, an element with k is stored at slot k.
- With hashing, this element is stored at position h(k), where $h: U \to \{0,\ldots m-1\}$ is a **hash function** to compute the slot in the hash table T[0..m-1] from key k. Also, h(k) is called the **hash value** of key k.
- The objective is to reduce the number of indices to be used.
- m is the size of the table. It is much less than $|U|$.

Collisions



- **Collision**: Two keys may hash to the same slot.
- The ideal solution is to avoid collisions with suitable hash functions.
- Make h appear to be “random”. But of course, it must be deterministic.
- Since |U|>m, avoiding collisions is impossible.
- Still, we should use a good hash function.
- We have effective techniques to address collisions.

Collision Resolution by Chaining



- We place all the elements that hash into the same slot into the same linked list.

CHAINED-HASH-INSERT(T, x)

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

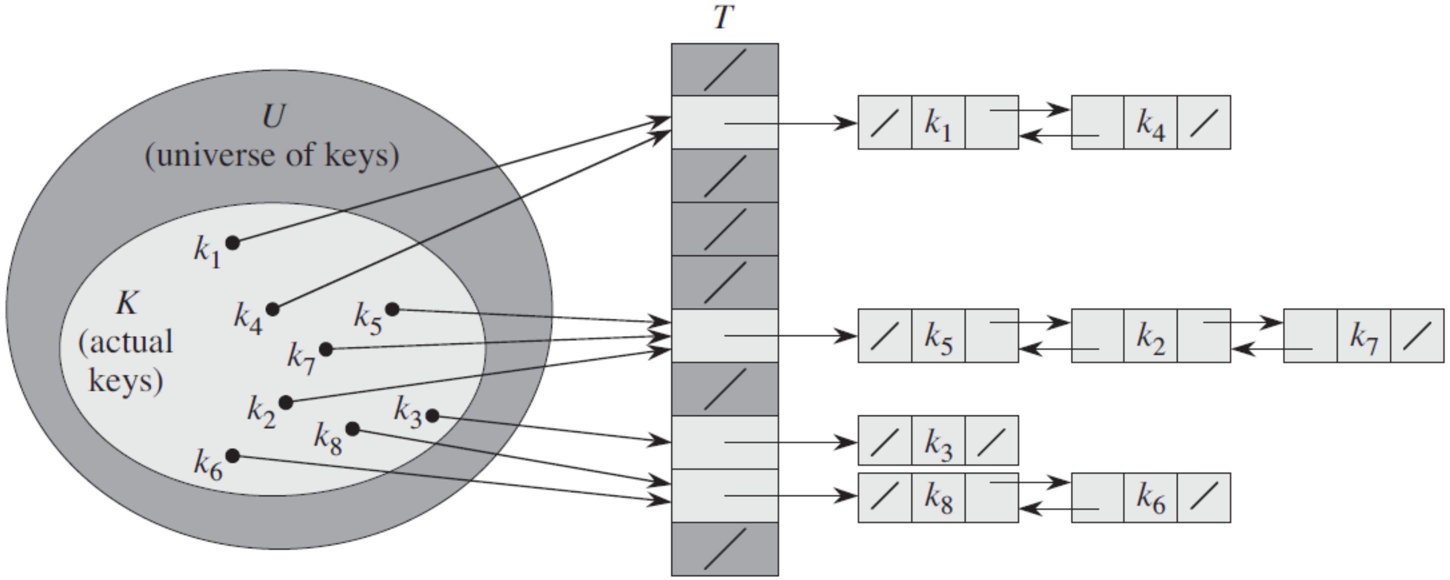
CHAINED-HASH-SEARCH(T, k)

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

CHAINED-HASH-DELETE(T, x)

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

Collision Resolution by Chaining



- Insertion: O(1) (worst case).
- Deletion: O(1) if the list is doubly linked (worst case).
- Search: size of the list.

```
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

- n: number of elements in the table.
- m: size of the table.
- $\alpha = n/m$: load factor.
- In the worst case, all elements are assigned to the same slot, i.e. $\theta(n)$.
- The average-case depends on how well the hashing function distributes the set of keys among the m slots.
- **Simple Uniform Hashing**: Any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.

Analysis of Hashing with Chaining

- n_j: length of the list T[j], j=0,1,…,m-1.
- $n = n_0 + n_1 + \dots + n_{m-1}$.
- $E[n_j] = \alpha = n/m$.
- We assume that computing h(k) takes O(1) time.
- Then, the time of the search of key k depends exclusively on $n_{h(k)}$.

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.

Proof: We need to reach the end of the corresponding list.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Analysis of Hashing with Chaining

Theorem 11.2
In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

- We assume that the element being searched is equally likely to be any of the n elements stored in the table.
- The number of elements examined during a successful search for x is one more than the elements that appear before x in the list.
- This is the number of elements that were inserted after x was inserted.
- We take the average, over the n elements in the table, of one plus the number of elements added to x's list after x was added to the list.

Analysis of Hashing with Chaining

Theorem 11.2
In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

- Let x_i denote the i-th element inserted into the table, for i=1.2,…,n, and let $k_i=x_i.key$.
- $X_{ij}=I\{h(k_i)=h(k_j)\}$.
- $E[X_{ij}]=Pr\{h(k_i)=h(k_j)\}=1/m$ under the assumption of simple uniform hashing.

Analysis of Hashing with Chaining

$$
\begin{aligned}
\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right]
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} \mathrm{E}\left[X_{ij}\right]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} \frac{1}{m}\right) \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)
\end{aligned}
$$

$$
\begin{aligned}
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n} n-\sum_{i=1}^{n} i\right) \\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}
$$

Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(2+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$. ■

Analysis of Hashing with Chaining

- Since the average-case search takes $\theta(1+\alpha)$, if $n=m$, then the search is $\theta(1)$.
- Thus, all the dictionary operations on hash tables take $O(1)$ in the average-case.

4. Hash Functions

Hash Functions

- A good hash functions satisfies approximately the assumption of simple uniform hashing.
- But we rarely know the probability distribution from which the keys are drawn.
- Moreover, the keys might not be drawn independently.
- Occasionally, we do know the distribution. For example, if the keys are drawn from real numbers k independently and uniformly distributed in the range $0 \le k < 1$, then the function $h(k) = \lfloor km \rfloor$ satisfies the simple uniform hashing assumption.

Hash Functions

- In practice, we can often employ heuristic techniques to create a hash function that performs well.
- Qualitative information about the probability distribution of keys may be useful in the design process.
- For instance, consider a compiler's symbol table.
- Close related symbols like pt and pts are likely to occur in the same program; a good hash function would minimize the chance that those symbols hash to the same slot.

Use of Radix Notation

- If the keys are not natural numbers, we find a way to interpret them as natural numbers.
- For instance pt can be interpreted as (112,116) since p=112 and t=116 in the ASCII code.
- Then, pt can be expressed as a radix-128 integer as (112*128)+116=14452.

The Division Method

- We map a key k into one of the m slots by h(k) = k mod m.
- Hashing by division is quite fast.
- m should not be a power of 2, since if $m=2^p$, then h(k) is just the p lowest-order bits of k.
- It’s better designing the hash function to depend on all the bits of k.
- A prime not to close to an exact power of 2 is a often a good choice for m.

The Division Method

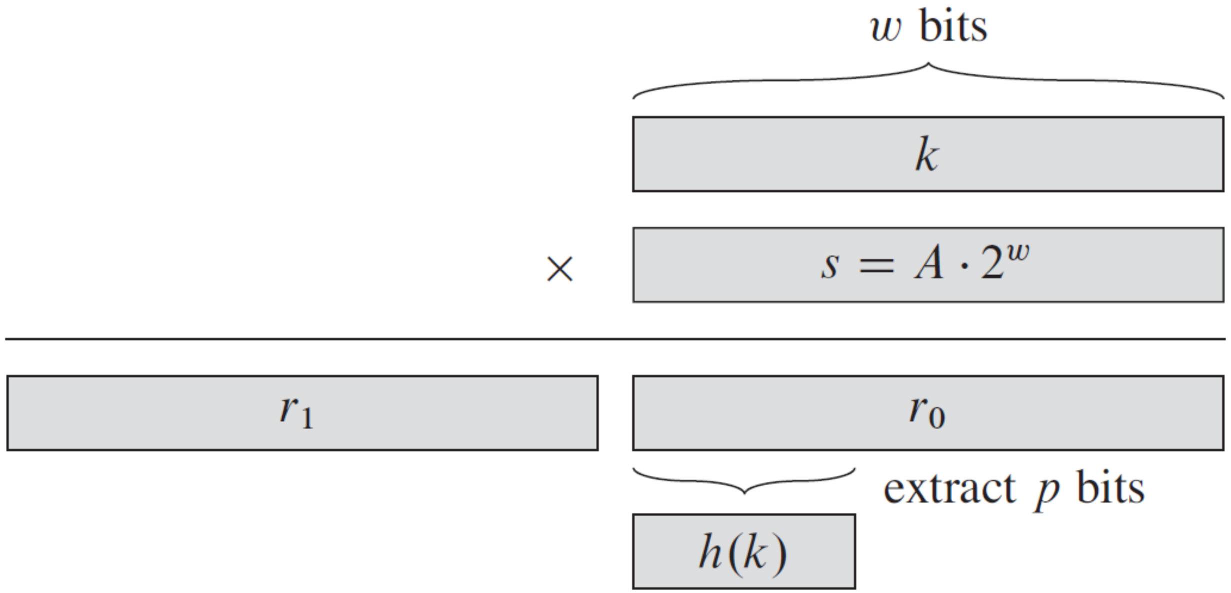
- For example, let’s say we wish to allocate n=2000 character strings, where a character has 8 bits, in a hash table that resolves collisions by chaining.
- If we don’t mind searching an average of 3 elements in an unsuccessful search, we could choose m=701 because it is a prime near 2000/3 but not near any power of 2.
- h(k) = k mod 701,

The Multiplication Method

- It has two steps:
    1. Multiply k by a constant A in the range 0<A<1 and extract its fractional part.
    2. Multiply this value by m and take the floor of the result.
- $h(k) = \lfloor m(kA \, mod \; 1) \rfloor$, where $kA \, mod \; 1 = kA - \lfloor kA \rfloor$.
- Advantage: The value of m is not critical.

The Multiplication Method

- $h(k) = \lfloor m(kA \bmod 1) \rfloor$, where $kA \bmod 1 = kA - \lfloor kA \rfloor$.
- We typically choose it to be a power of 2 ($m=2^p$) since we can easily implement the function on most computers:
  - Let w be the word size of the machine. Suppose k fits in a single word.
  - Let $A = \frac{s}{2^w}$ where s is an integer such that $0 < s < 2^w$. Then, $s = A2^w$.

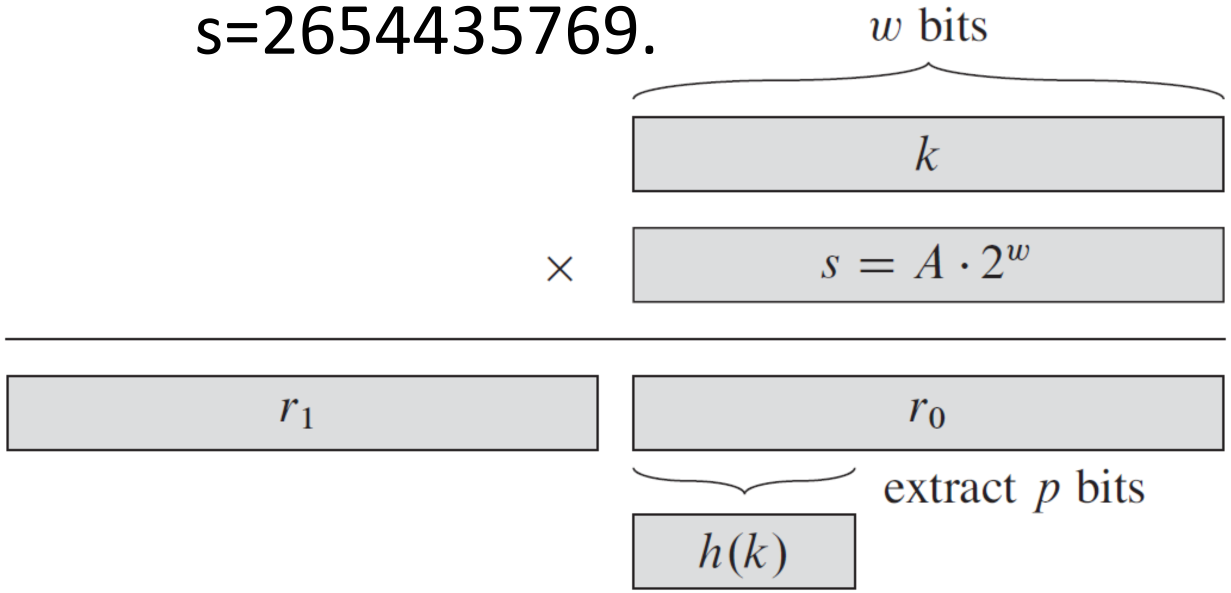


- $sk = Ak2^w = r_1 2^w + r_0$.
- We need to divide by 2^w, i.e. >>w.
- The fractional part is r_0., i.e. $kA \bmod 1$.
- We need to multiply r_0 by $m=2^p$, i.e. <<p. The integer part is the first p bits. If we don't shift, it's still the first p bits of r_0.

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

The Multiplication Method

- Although this method works with any value of A, some values are better than others. Knuth suggests $A \approx \frac{(\sqrt{5}-1)}{2} = 0.6180339887 \ldots$
- Example: k =123456, p=14, w=32, m=2^{14}=16384.
- We can choose A of the form s/2^{32}= that is closest to $\frac{(\sqrt{5}-1)}{2}$. Then, s=2654435769.



ks = 327706022297664

= 76300.2^{32}+17612864.

The 14 most significant bits or r_0 yield the value h(k)=67.

00000001000011001100000001000000

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

5. Open Addressing

Open Addressing

- All elements occupy the hash table itself.
- Each entry contains either an element or NIL.
- When searching an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.
- The load factor α can never exceed 1.
- We don't use pointers to other slots in the table to save memory.
- Instead, we use a probe sequence so that when we cannot insert k in its hashed value, we attempt to do it somewhere else.

Probe Sequence

- For a given key k, we compute the sequence of slots to be examined: **the probe sequence**. This sequence depends on k.
- Since the table has m slots (from 0 to m-1), we index such probes from 0 to m-1. This probe is included in the hash function:

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

With open addressing, we require that for every key k, the ***probe sequence***

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

- This sequence must be a permutation of 0,1,…, m-1 so that all the positions in the table are considered.

Hash-Insert

- Find the first empty slot in the probe sequence and insert the given element there.

```
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

- It uses the same probe sequence as insertion. When an empty slot is found we can safely say the element is not in the table (no deletions are allowed).

```
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
```

Deletions

- It is difficult. We cannot simply replace the slot with empty. Why?
- Solution? Set a new value for slots called *deleted*. What would need to be modified?

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
```

Uniform Hashing

- The probe sequence if each key is equally likely to be any of the m! permutations of 0,1,…, m-1.
- It’s a generalization of simple uniform hashing but applied o a whole probe sequence.
- True hashing is difficult to implement.
- Three commonly used techniques used to compute probe sequences are:
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Linear Probing

- **Auxiliary Hash Function:** h': U → {0,…,m-1}.
- **Hash Function:** h(k,i) = (h'(k) + i) mod m, for i=0,…,m-1.
- Because the initial probe determines the whole probe sequence, we can only obtain m distinct probe sequences.
- **Primary Clustering:** Long runs of occupied slots build up, increasing the average search time.
- Clusters arise because an empty slot preceded by i full slots gets filled with higher probability. What probability?
- Example: given m=5 and h'(k) = k mod 5, insert 1, 3, 11, 22, 13.
  - Does this function consider all positions for all keys?

Presentation made by Juan Mendivelso. Contents and figures extracted from the book: Introduction to Algorithms, Third Edition. Cormen, Leiserson, Rivests and Stein. The MIT Press. 2009.

Quadratic Probing

- **Auxiliary Hash Function:** $h': U \rightarrow \{0,\ldots,m-1\}$.
- **Hash Function:** $h(k,i) = (h'(k) + c_1 i + c_2 i^2)$ mod m, for $i=0,\ldots,m-1$.
- Because the initial probe determines the whole probe sequence, we can only obtain m distinct probe sequences.
- However, it is much better than linear probing: **Secondary Clustering**.
- In order to make full use of the table, the values of c_1 and c_2 are restricted.
- Example: given $m=5$, $h'(k) = k$ mod 5, and $h(k,i) = (h'(k) + i + i^2)$ mod m, insert 1, 3, 11, 22, 13.
  - Does this function consider all positions for all keys?

Double Hashing

- One of the best methods for open addressing because the permutations have many of the characteristics of random chosen permutations.
- **Auxiliary Hash Functions:** $h_1(k)$ and $h_2(k)$.
- **Hash Function:** $h(k,i) = (h_1(k) + i.h_2(k)) \bmod m$, for $i=0,\ldots,m-1$.
- The probe sequence depends in two ways upon the key k.
- When m is prime or a power of two, double hashing produces $\theta(m^2)$ probe sequences since each pair $(h_1(k), h_2(k))$ produces a distinct probe sequence.
- Example: given m=5, $h_1(k) = k \bmod 5$, and $h_2(k) \bmod 3$, insert 1, 3, 11, 22, 13.
  - Does this function consider all positions for all keys?

Double Hashing

- The value $h_2(k)$ must be relatively prime to the hash table size m for the entire table to be searched. This can be achieved by:
  1. Let m be a power of 2 and design h_2 so that it always produced and odd number.
  2. Let m be prime and design h_2 so that it always return a positive integer less than m. For instance,

$$\begin{aligned} h_1(k) &= k \bmod m , \\ h_2(k) &= 1 + (k \bmod m') , \end{aligned}$$

where m' is chosen to be slightly less than m (say, $m - 1$).

For example, given k = 123456, m=701 and m'=700, we have $h_1(k)=80$ and $h_2(k)=257$.

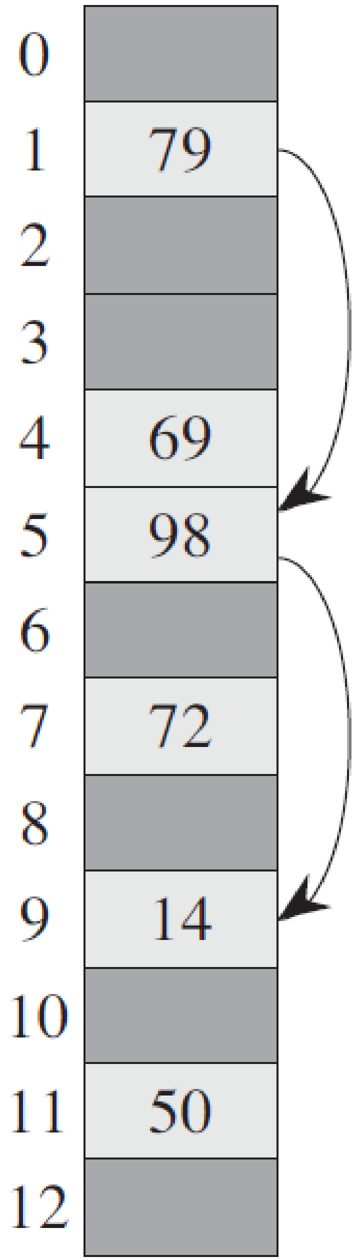
So we first probe position 80 and then we examine every 257th slot (mod m).

Double Hashing

- $m = 13$, $m' = 11$, $h_1(k) = k \bmod 13$, and $h_2(k) = 1 + (k \bmod 11)$.
- Insert keys 79, 69, 98, 72, 50, 14.

Double Hashing

- $m = 13$, $m' = 11$, $h_1(k) = k \bmod 13$, and $h_2(k) = 1 + (k \bmod 11)$.
- Insert keys 79, 69, 98, 72, 50, 14.



BIBLIOGRAPHY

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press. 2009.