Algorithms

Lecture10

Hash Tables

- Direct-address tables
- Hash tables
- Hash functions
- Open addressing

Introduction

- Many applications require a dynamic set that supports only the **dictionary operations** INSERT, SEARCH, and DELETE.
- A hash table is effective for implementing a dictionary.
- The expected time to search for an element in a hash table is $O(1)$, under some reasonable assumptions.
- Worst-case search time is O(n), however.
- A hash table is a generalization of an ordinary array.
- With an ordinary array, we store the element whose key is k in position k of the array.
- Given a key k, we find the element whose key is k by just looking in the kth position of the array. This is called **direct addressing**.

Introduction

- We use a hash table when we do not want to (or can’t) allocate an array with one position per possible key.
- Use a hash table when the number of keys actually stored is small relative to the number of possible keys.
- A hash table is an array, but it typically uses a size proportional to the number of keys to be stored (rather than the number of possible keys).
- Given a key *k*, don’t just use *k* as the index into the array. Instead, compute a function of *k*, and use that value to index into the array. We call this function a **hash function**.

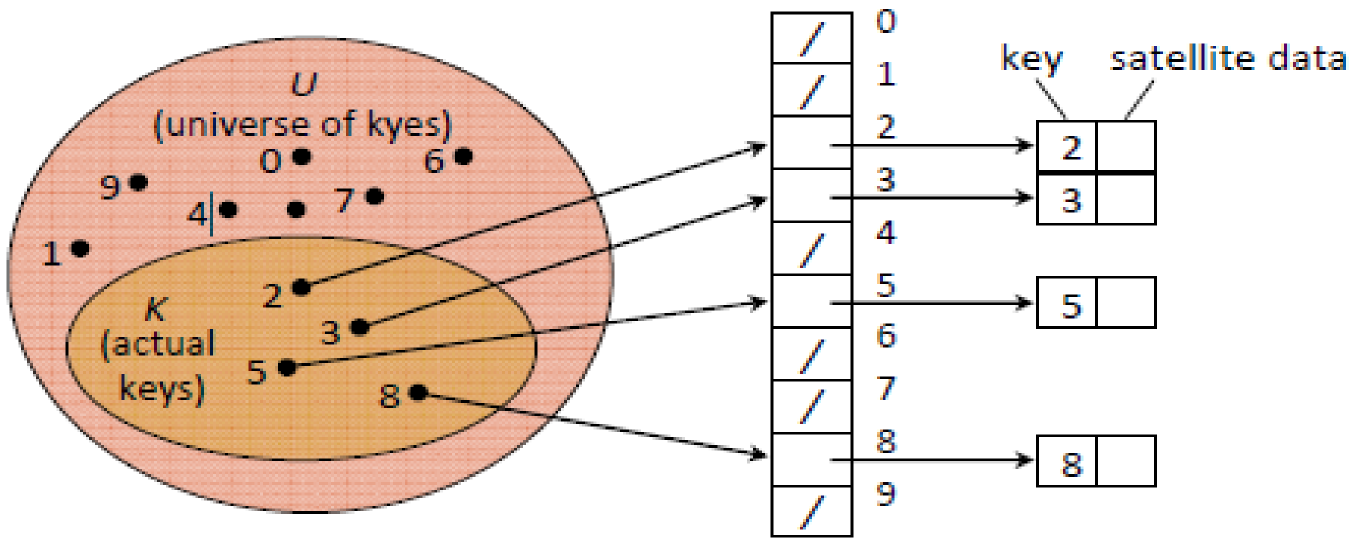
Introduction

- Issues that we’ll explore in hash tables:
- How to compute hash functions?
  - The multiplication methods.
  - The division methods.

- What to do when the hash function maps multiple keys to the same table entry? (collision)
  - Chaining.
  - Open addressing.

Direct-address tables

- Maintain a dynamic set.
- Each element has a key drawn from a universe $U = \{0, 1, \ldots, m - 1\}$ where m isn't too large.
- No two elements have the same key.
- Represent by **direct-address table**, or array, $T[0..m\text{-}1]$:
  - Each **slot**, or position, corresponds to a key in U.
  - If there is an element x with key k, then $T[k]$ contains a pointer to x.
  - Otherwise, $T[k]$ is empty, represented by NIL.



- Dictionary operations are trivial and take *O*(1) time each:

```
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
```

Problem:

- If the universe U is large, storing a table of size $|U|$ may be impractical or impossible.
- The set K of keys actually stored is small, compared to U, so that most of the space allocated for T is wasted.

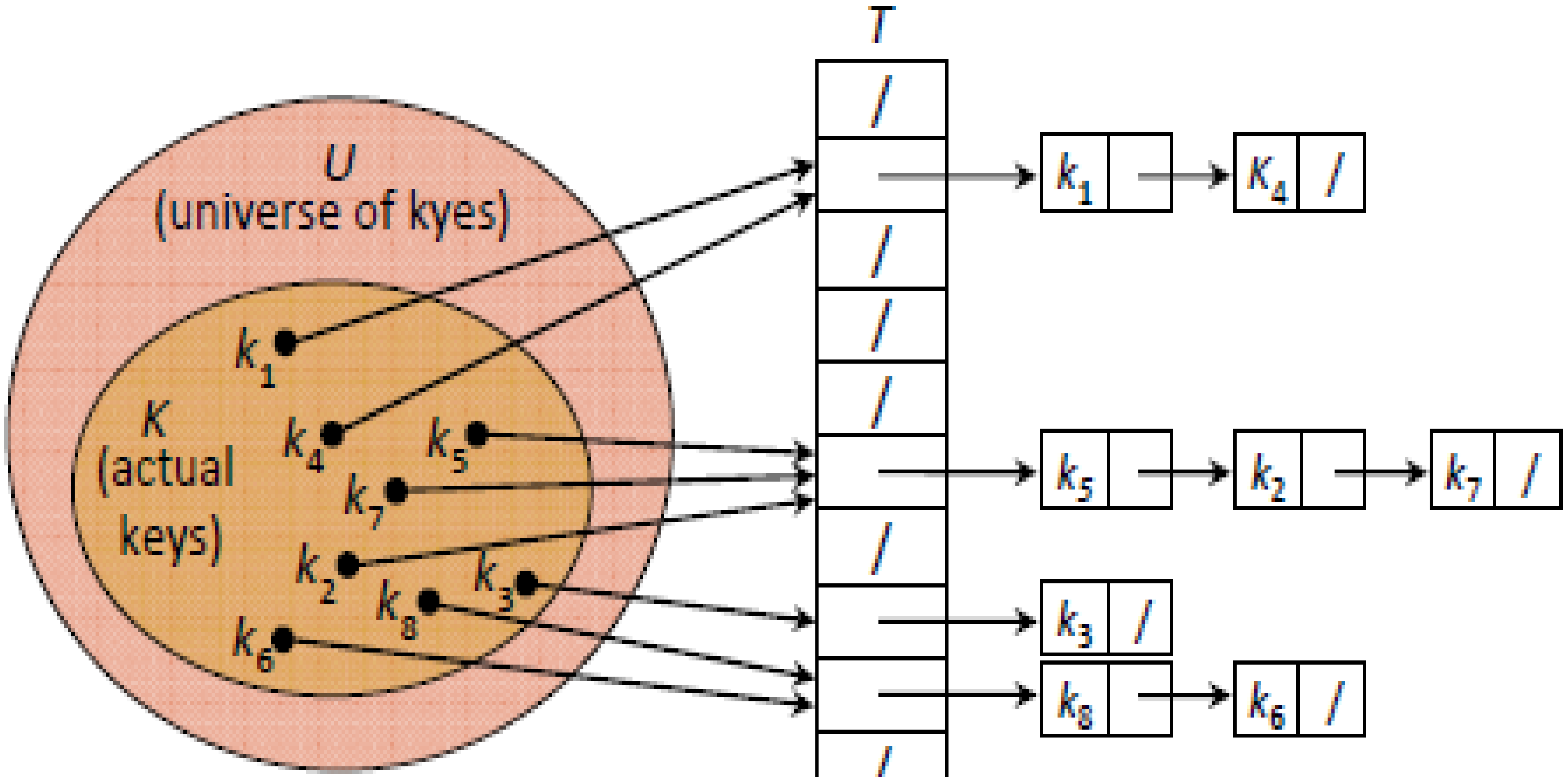
Solution: Hash tables

- When K is much smaller than U, a hash table requires much less space than a direct-address table.
- Storage requirements can be reduced to $\Theta(|K|)$.
- Searching for an element requires $O(1)$ time, but in the **average case**, not the **worst case**.

Hash Tables

- **Idea:** Instead of storing an element with key k in slot k, use a function h and store the element in slot $h(k)$.
- We call h a **hash function**.
- $h : U \ \{0, 1, . . . , m - 1\}$, so that $h(k)$ is a legal slot number in T.
- We say that k **hashes** to slot $h(k)$.
- We also say that $h(k)$ is the **hash value** of key k.

Hash Tables



Collisions: When two or more keys hash to the same slot.

- Can happen when there are more possible keys than slots ($|U| > m$).

Methods to resolve the collision problem.

- **Chaining**
- **Open addressing**

- Chaining is usually better than open addressing.

Collision resolution by chaining

- Put all elements that hash to the same slot into a linked list.
- Slot *j* contains a pointer to the head of the list of all stored elements that hash to *j*.
- If there are no such elements, slot *j* contains NIL.

Dictionary Operations

How to implement dictionary operations with chaining:

*CHAINED-HASH-**INSERT**(T,x):*

Insert x at the head of list T[h(key[x])]

- Worst-case running time is $O(1)$.
- Assumes that the element being inserted isn't already in the list.
- It would take an additional search to check if it was already inserted.

*CHAINED-HASH-**SEARCH**(T,k):*

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

- Running time is proportional to the length of the list of elements in slot $h(k)$.

Dictionary Operations….

*CHAINED-HASH-**DELETE**(T,x):*

Delete x from the list T[h(key[x])]

- Given pointer *x* to the element to delete, so no search is needed to find this element.
- Worst-case running time is *O*(1) time if the lists are doubly linked.
- If the lists are singly linked, then deletion takes as long as searching, because we must find *x*'s predecessor in its list.

Analysis of hashing with chaining

Given a key, how long does it take to find an element with that key?

Analysis is in terms of the **load factor** $\alpha = n / m$:

- n = # of elements in the table.
- m = # of slots in the table = # of (possibly empty) linked lists.
- Load factor is average number of elements per linked list.
- Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$.

- **Worst case** is when all *n* keys hash to the same slot
- get a single list of length *n*
- worst-case time to search is $\Theta(n)$, plus time to compute hash function.
- **Average case** depends on how well the hash function distributes the keys among the slots.`

Average-case performance

- Assume **simple uniform hashing**: any given element is equally
- likely to hash into any of the *m* slots.
- For *j* = 0, 1, …, *m*−1, denote the length of the list *T*[*j*] by *nj*, so
- that *n* = *n*0 + *n*1 + ... + *nm*−1.
- Average value of *nj* is E[*nj*] = ⍰ = *n*/*m*.
- Assume that the hash value *h*(*k*) can be computed in *O*(1) time.
- Time for the element with key *k* depends on the length *nh*(*k*)
- of the list *T*[*h*(*k*)].
- We consider two cases:
  - contains no element with key *k* $\longrightarrow$ unsuccessful.
  - contain an element with key *k* $\longrightarrow$ successful.

Average-case performance

- Assume **simple uniform hashing**: any given element is equally likely to hash into any of the *m* slots.
- For $j = 0, 1, \ldots, m-1$, denote the length of the list $T[j]$ by n_j, so
- that $n = n_0 + n_1 + \ldots + n_{m-1}$.
- Average value of n_j is $E[n_j] = \alpha = n/m$.
- Assume that the hash value $h(k)$ can be computed in $O(1)$ time.

- Time for the element with key *k* depends on the length $n_{h(k)}$ of the list $T[h(k)]$.

- We consider two cases:
  - contains no element with key *k* $\rightarrow$ unsuccessful.
  - contain an element with key *k* $\rightarrow$ successful.

Theorem 11.1

- An **unsuccessful search** takes expected timeᶿ (1+ α).

Proof:

- Under the assumption of simple uniform hashing, any key not already in the table is equally likely to hash to any of the *m* slots.
- To search unsuccessfully for any key *k*, need to search to the end of the list *T*[*h*(*k*)].
- This list has expected length E[*nh*(*k*)] = α.
- Therefore, the expected number of elements examined in an unsuccessful search is .
- Adding in the time to compute the hash function.
- The total time required isᶿ (1 +α).

Theorem 11.2

- An **successful search** takes expected time ᶱ (1+ α).

Proof:

- Assume the element being searched for is equally likely to be any of the *n* elements in the table *T*.
- During a successful search for *x*, the # of elements examined = # of elements in the list before *x* + 1.
- The expected length of that list is (*n* – *i*)/*m*.
- The expected # of elements examined in a successful search is

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

- The total time is ᶱ(2 + α /2 – α /2*n*) = ᶱ (1+ α).