Data Structures and Algorithm Analysis (CSC317)

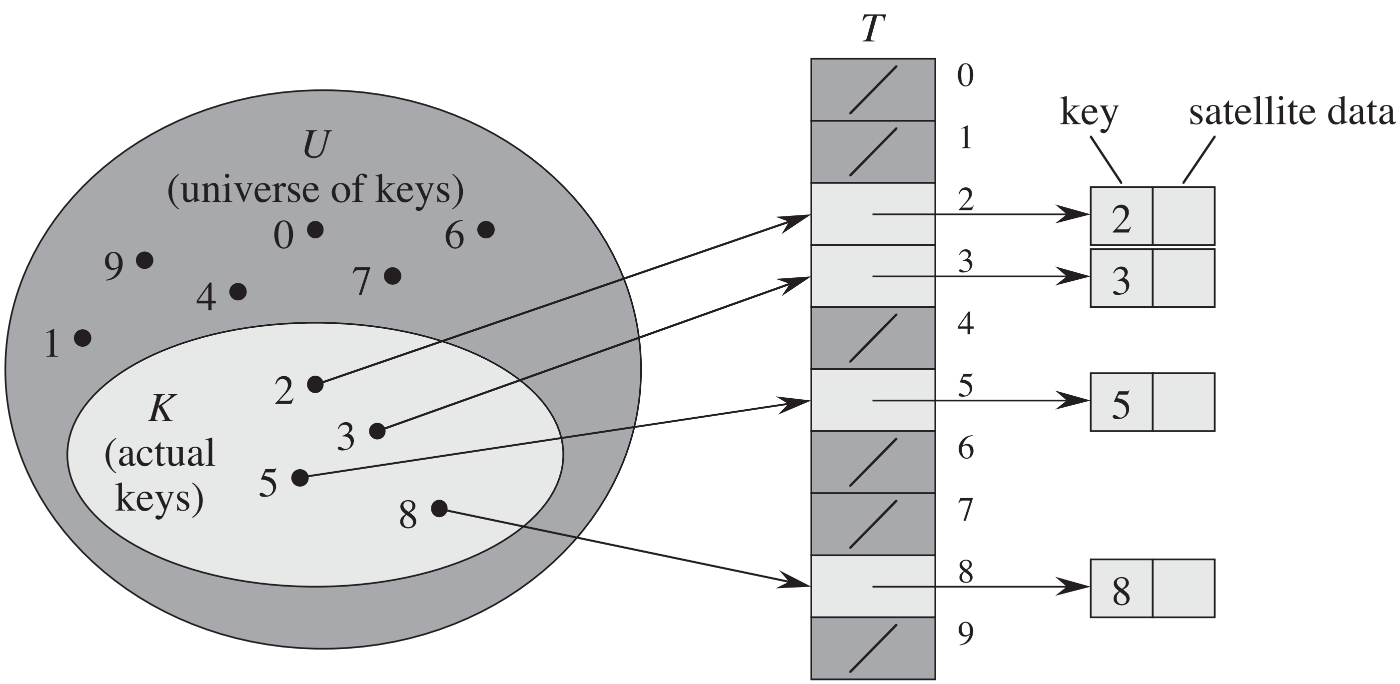
Hash tables

Hash table

- We have elements with key and satellite data
- Operations performed: Insert, Delete, Search/lookup
- We ***don't*** maintain order information
- We'll see that all operations **on average O(1)**
- **But worse case can be O(n)**

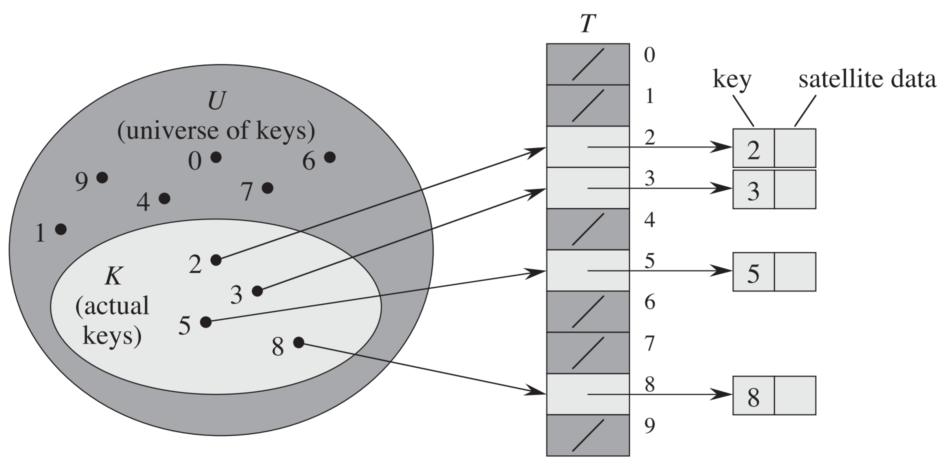
Hash table

- Simple implementation: If universe of keys comes from a small set of integers [0..9], we can **store directly in array** using the keys as indices into the slots.



Hash table

- Simple implementation: If universe of keys comes from a small set of integers [0..9], we can store directly in array using the keys as indices into the slots.



- This is also called a direct-address table
- Search time just like in array – O(1) !

Example: Array versus Hash table

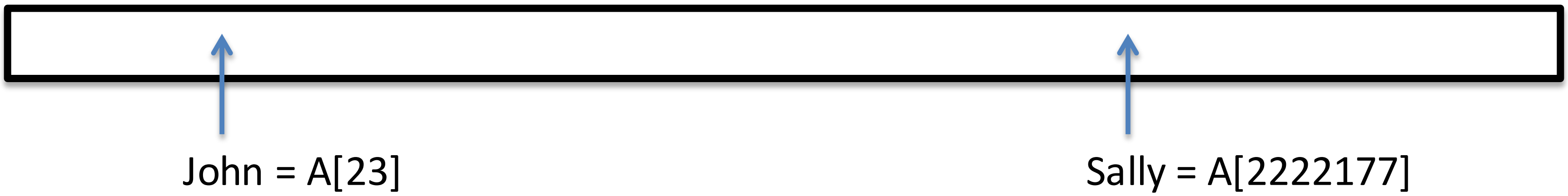
- Imagine we have keys corresponding to friends that we want to store

Example: Array versus Hash table

- Imagine we have keys corresponding to friends that we want to store

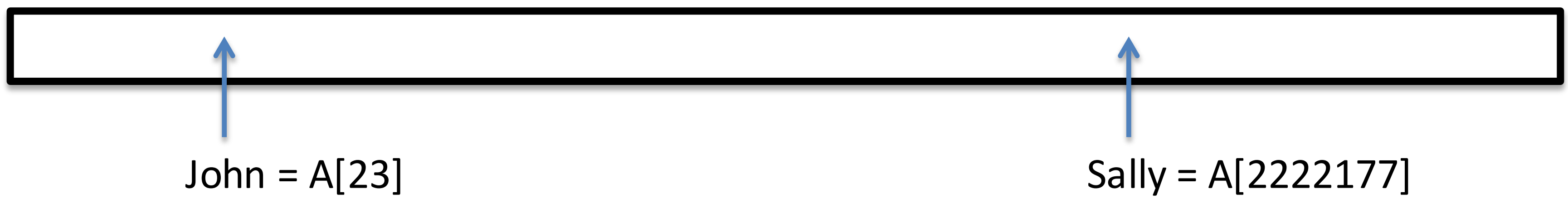
- Could use huge array, with each friend's name mapped to some slot in array (eg, one slot in array for every possible name; each letter one of 26 characters, n letters in each name..)

Example: Array versus Hash table



Pros/cons?

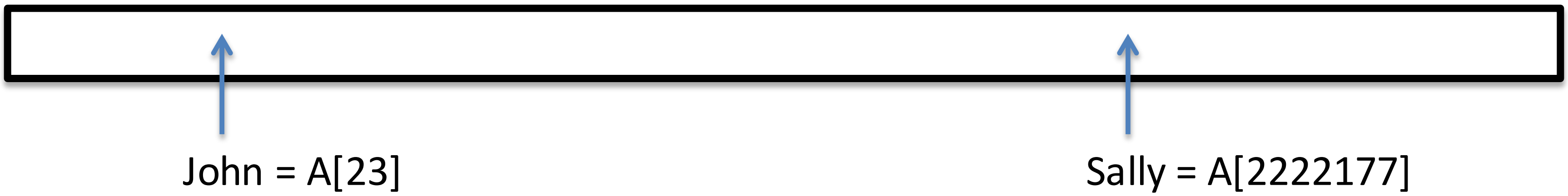
Example: Array versus Hash table



Pros/cons?

- We could insert, find key, and delete element in O(1) time – very fast!

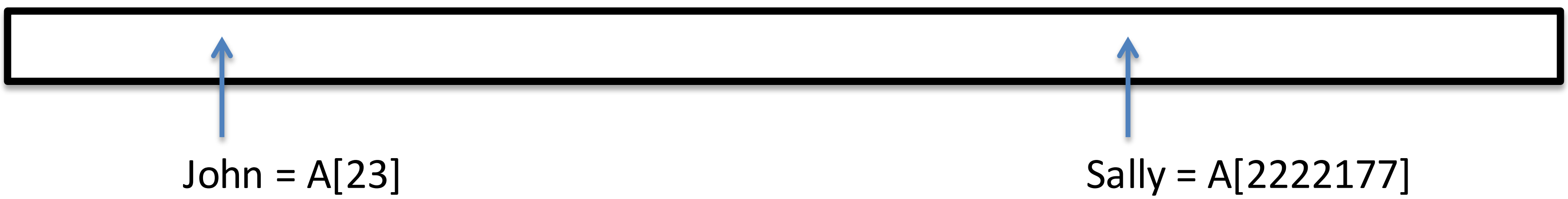
Example: Array versus Hash table



Pros/cons?

- We could insert, find key, and delete element in O(1) time – very fast!
- But huge waste of memory, with many slots empty in many applications

Example: Array versus Hash table



Pros/cons?

- We could insert, search key, and delete array element in O(1) time – very fast!

- But huge waste of memory, with many slots empty
 in many applications

Example: versus linked list

- An alternative might be to use a linked list with all the friend names linked

John -> Sally -> Bob

- Pro: This is not wasteful because we only store the names that we want

- Con: But search time is now O(n)

- We want an approach that is fast, and not so wasteful!

Example: versus linked list

- We’ll eventually want best of both worlds – advantages of array and of linked list

Hash table

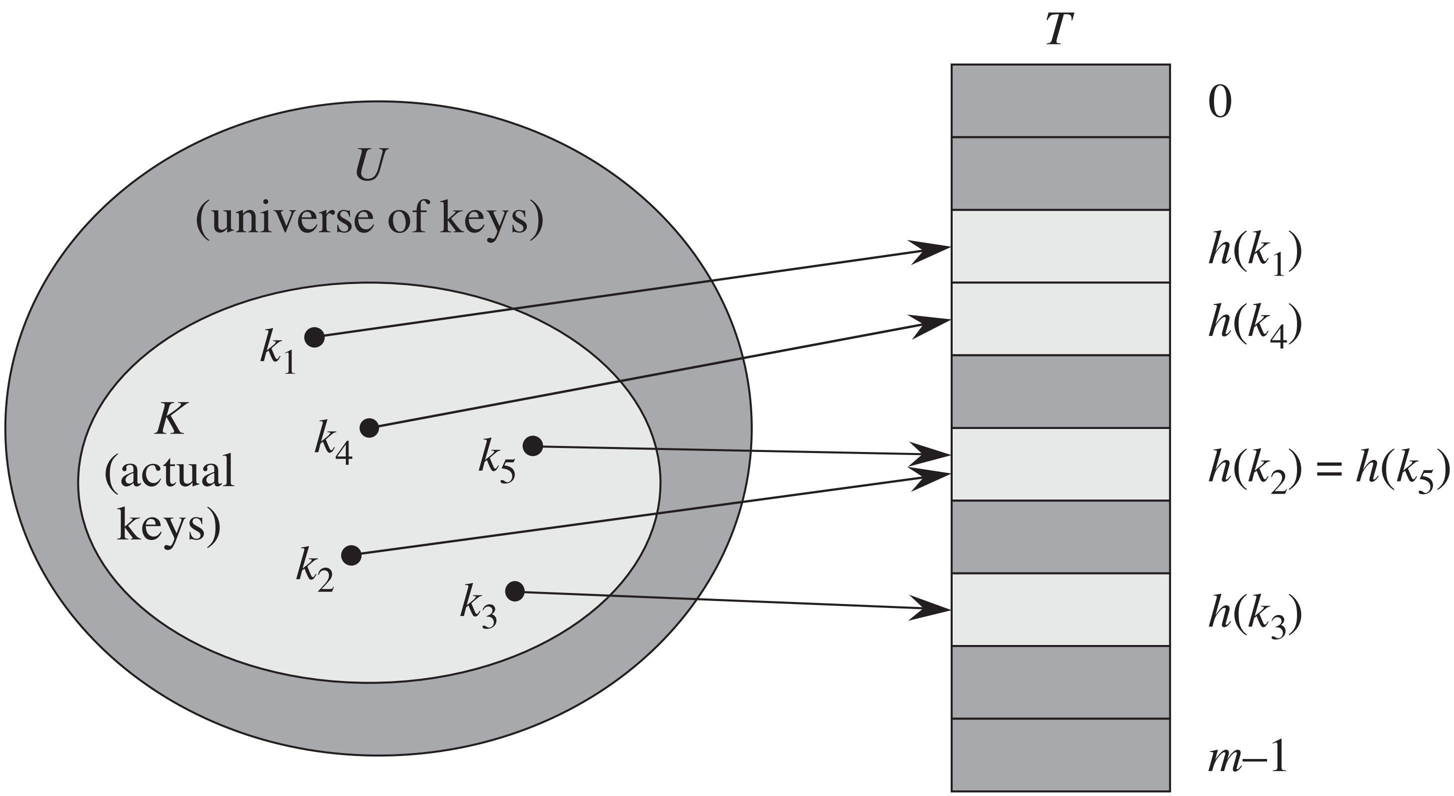
- Extremely useful alternative to static array for insert, search, and delete in O(1) time (on average) – VERY FAST

- Useful when universe is large, but at any given time number of keys stored is small relative to total number of possible keys (not wasteful like a huge static array)

- We usually don't store key directly as index into array, but rather compute a hash function of the key k, h(k), as index

Hash table

- What problem can arise if we map keys to slots in a hash table? Answer: collisions; two keys map to same slot.



Collisions

- Are guaranteed to happen when number of keys in table greater than number of slots in table
- Or if “bad” hashing function – all keys were hashed to just one slot of hash table – more later
- Even by chance, collisions are likely to happen. Consider keys that are birthdays. Recall the birthday paradox – room of 28 people, then 2 people have a 50 percent chance to have same birthday.

Collisions

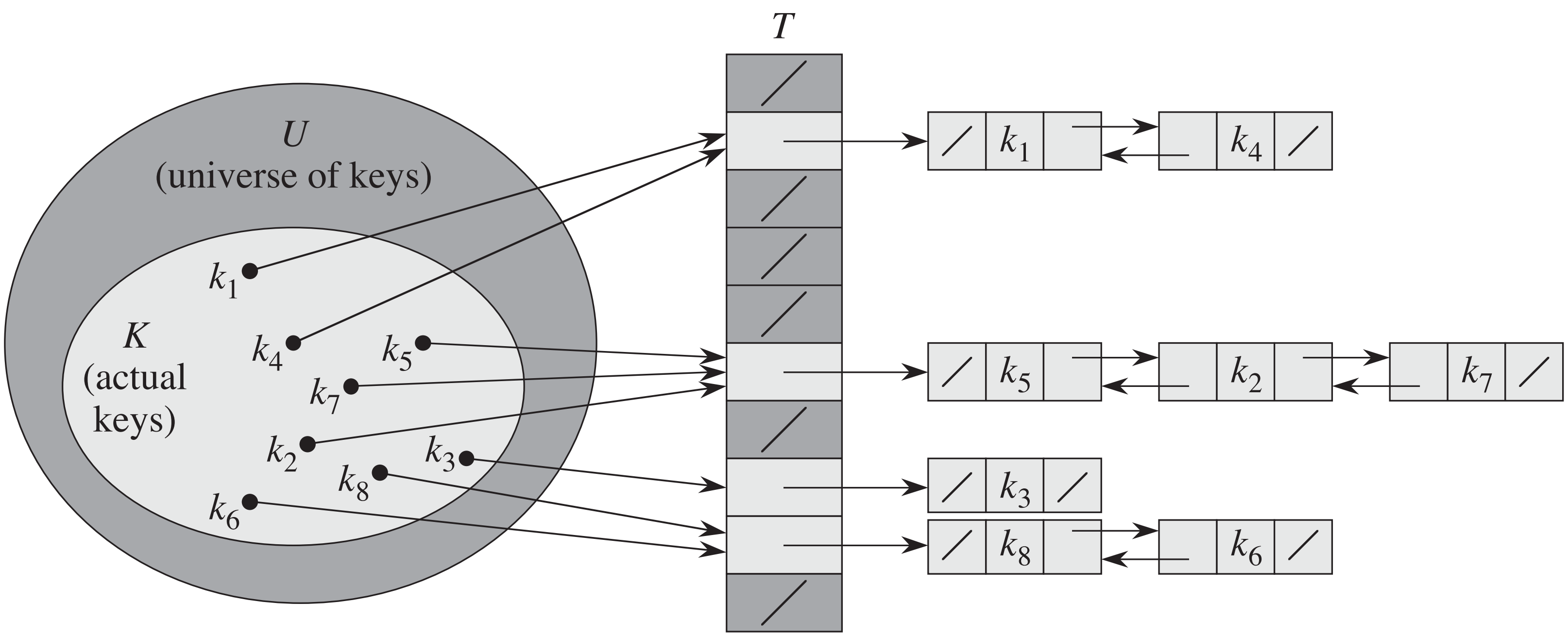
- So we have to deal with collisions!
- One solution?

Collisions

- So we have to deal with collisions!
- One solution?

Collision resolution by chaining

Collision resolution by chaining



Hash table analyses

- Worst case: all n elements map to one slot (one big linked list…).

  O(n)

Hash table analyses

- Average case: Define:

m = number of slots

n = number elements (keys) in hash table

What was n in previous diagram? (answer: 8)

alpha = load factor: $\alpha = \frac{n}{m}$

Intuitively, alpha is average number elements per linked list.

Hash table analyses

- Example: let’s take n = m; alpha = 1
- Good hash function: each element of hash table has one linked list
- Bad hash function: hash function always maps to first slot of hash table, one linked list size n, and all other slots are empty

Hash table analyses

- Good hash function should spread the keys in a balanced way across the slots of the hash table

- Each key should be equally likely to hash into each of the m slots, and each key should be independent of where other keys hashed to

- This is called simple uniform hashing: prob(key k hashes to each slot) = $\frac{1}{m}$

Hash table analyses

Define:

- Unsuccessful search: new key searched for doesn't exist in hash table (we are searching for a new friend, Sarah, who is not yet in hash table)

- Successful search: key we are searching for already exists in hash table (we are searching for Tom, who we have already stored in hash table)

Hash table analyses

Theorem:

- Assuming simple uniform hashing, unsuccessful search takes on average O(alpha + 1)
  Here the actual search time is O(alpha) and the added 1 is the constant time to compute a hash function on the key that is being searched

Hash table analyses

Theorem interpretation:

- n=m

  $\Theta(1+1) = \Theta(1)$

- n=2m

  $\Theta(2+1) = \Theta(1)$

- n=m^3

  $\Theta(m^2+1) \neq \Theta(1)$

- Summary: we say constant time on average when n and m similar order, but not generally guaranteed

Hash table analyses

Theorem:

- Intuitively: Search for key k, hash function will map onto slot h(k). We need to search through linked list in the slot mapped to, up until the end of the list (because key is not found = unsuccessful search). For n=2m, on average linked list is length 2. More generally, on average length is alpha, our load factor.

Hash table analyses

Proof with indicator random variables:

- **Consider keys j in hash table (n of them), and key k not in hash table that we are searching for. For each j:**

$$X_j = \begin{cases} 1 & \text{if key x hashes to same slot as key j} \\ 0 & \text{otherwise} \end{cases}$$

Hash table analyses

Proof with indicator random variables:

- **Consider keys j in hash table (n of them), and key k not in hash table that we are searching for. For each j:**

$X_j =$ 1 if h(x) = h(j) (same as before, just as equation)
0 otherwise

Hash table analyses

Proof with indicator random variables:

- **As with indicator (binary) random variables:**

$$E[X_j] = 1\Pr(X_j = 1) + 0\Pr(X_j = 0) = P\,r(X_j = 1)$$

- **By our definition of the random variable:**

$$= \Pr(h(x) = h(j))$$

- **Since we assume uniform hashing:**

$$= \frac{1}{m}$$

Hash table analyses

Proof with indicator random variables:

- **We want to consider key x with regards to every possible key j in hash table:**

$$E[\sum_{j=1}^{n} X_j] =$$

- **Linearity of expectations:**

$$= \sum_{j=1}^{n} E[X_j] = \sum_{j=1}^{n} \frac{1}{m} = \frac{n}{m} = \alpha$$

Hash table analyses

- We've proved that average search time is O(alpha) for unsuccessful search and simple uniform hashing. It is O(alpha+1) if we also count the constant time of mapping a hash function for a key

Hash table analyses

Theorem:

- Assuming simple uniform hashing, **successful** search takes on average O(alpha + 1)
  Here the actual search time is O(alpha) and the added 1 is the constant time to compute a hash function on the key that is being searched

- Intuitively, successful search should take less than unsuccessful. Proof in book with indicator random variables; more involved than before; we won't prove here.