# Data Structures with “Randomness”: Hashtables

# Flashback to Data Structures…

Recall the `Dictionary` interface

- What are the `Dictionary` operations?
- What concrete `Dictionary` implementations did we study?

- What are the tradeoffs between binary search trees and hashtables?
- How often do we need to do successor/range operations?
  - Similarly: How much does locality matter?

**Let's develop a data structure with excellent (expected) point lookup/update performance but no support for range operations.**

# Hashtable Basics

- We have an underlying array of size $m$
    - We say this array has $m$ slots or buckets
- Suppose we want to store $n$ items, where $n < m$. What is ideal situation?
    - If every element has a unique, designated location, get $O(1)$ operations:
        - Insert a new item → update slot
        - Look up an item → check slot
        - Delete an item → clear slot
- Unfortunately we usually have a universe of items $U$ we may wish to store, where $|U|$ is *much much* bigger than $m$. Example universes?
    - Punchline: even with $n < m$, we can't guarantee those $n$ items their own dedicated locations because we don't know which particular $n$ items from our universe $U$ that we will be storing…

# Hash table

- But we still want $O(1)$ operations! Plus, you've been told we achieve that!
    - In reality, we settle for *expected* $O(1)$ performance…
- Idea: use a **hash function** to map each item to a slot
    - $h$ is a one-way function that maps the universe $U$ of keys to slots in our array $A$:

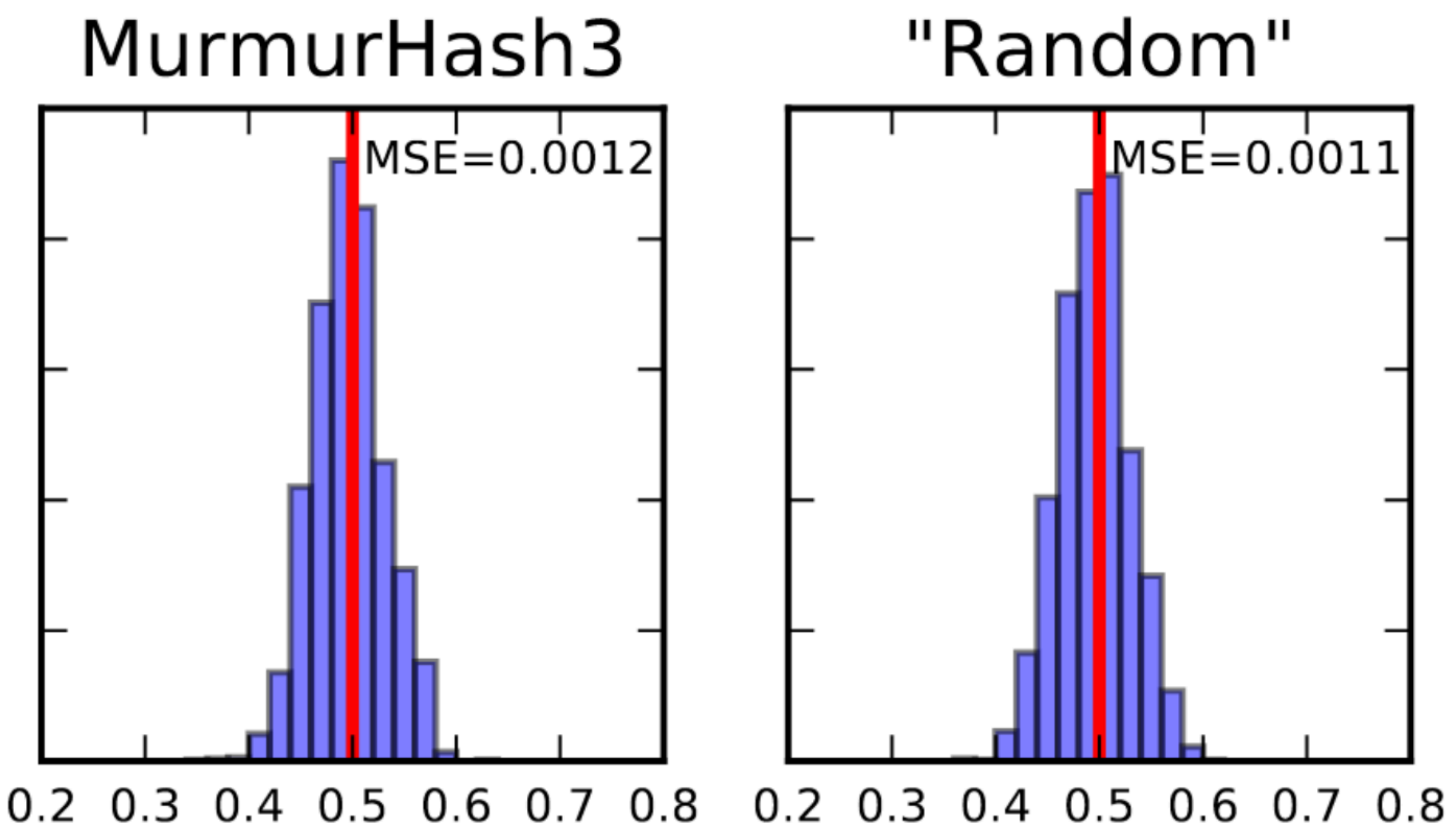
$$h : U \rightarrow \{0,1,\ldots,m-1\}$$

- So, we say an item with key $k$ hashes to slot $h(k)$, and that $h(k)$ is the item's hash value
    - Textbook gives example hash functions (and why some are bad)
    - Textbook discusses universal hashing
    - Instead, we're going to focus on analyzing the data structure under the assumption that we do in fact have a uniform hash function

# Hash function: theory versus practice

- We will *assume* hash function $h$ is *ideal* :
  - For all $i \in U, k$, assume $\Pr(h(i) = k) = 1/m$
  - Assume the hashes of all items are independent: $\Pr(h(i) = k \mid h(i_2) = k_2, h(i_3) = k_3, \ldots) = 1/m$

Dahlgaard et al. 2017



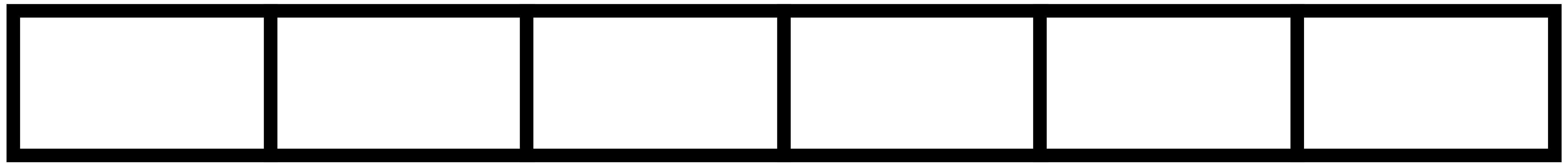

Histograms of set similarity estimates

- Such $h$s called **uniform random hash functions**
- Good hash functions do behave this way in practice
- Lots of theoretical work about weaker assumptions on the hash functions

# Hash table

- Hash function $h$, array $A$
- Item $i$ is stored in $A[h(i)]$
- $m = 6$

Amir



Beth

Chris

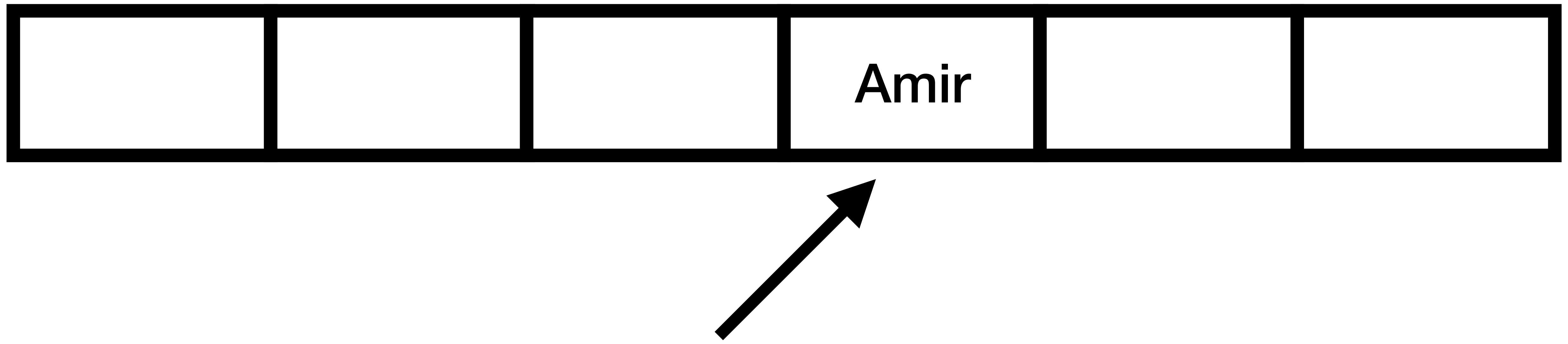
# Hash table

- Hash function $h$, array $A$
- Item $i$ is stored in $A[h(i)]$

Amir

Beth

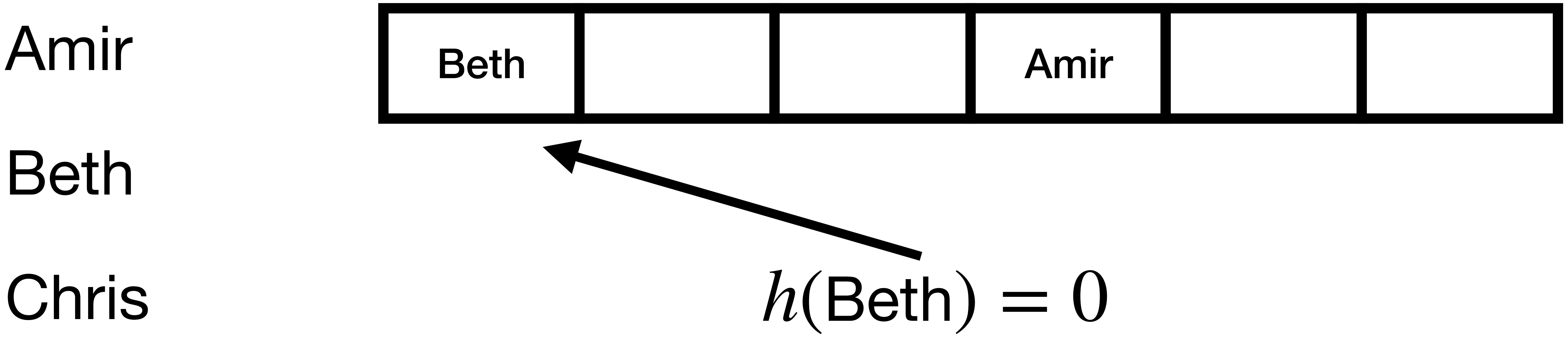
Chris




$h(\text{Amir}) = 3$

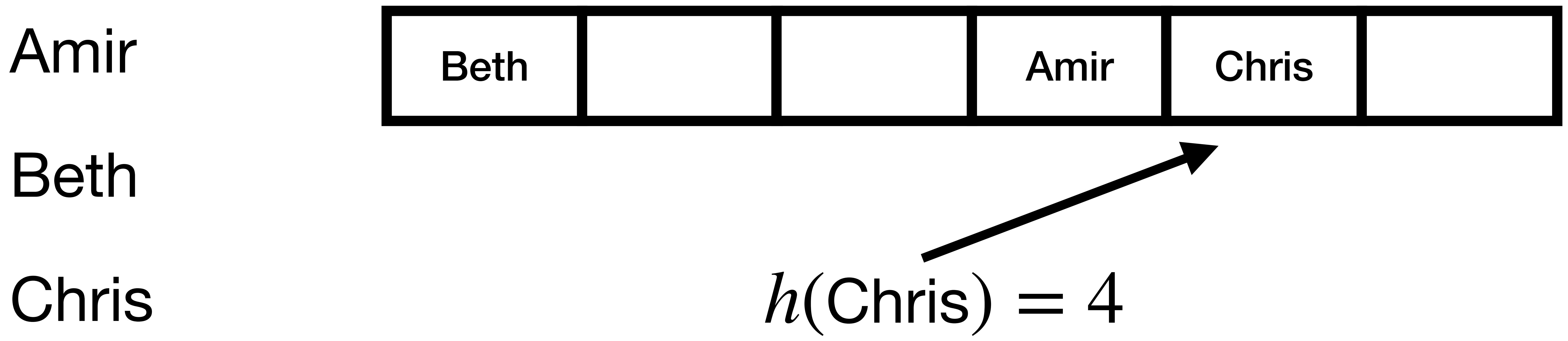
# Hash table

- Hash function $h$, array $A$
- Item $i$ is stored in $A[h(i)]$



# Hash table

- Hash function $h$, array $A$
- Item $i$ is stored in $A[h(i)]$



# Hashtable Basics

- We said that even with $n < m$, we can't guarantee those $n$ items their own dedicated locations because we don't know which particular $n$ items from our universe $U$ that we will be storing…
  - So we say a collision occurs when two unique items hash to the same slot $(h(x_1) = h(x_2), x_1 \neq x_2)$
- Practically, we need a way to manage collisions
  - Recall any strategies from data structures?
- Theoretically, we need a way to analyze the impact of collisions on our data structure performance
  - Our collision strategy needs to maintain our expected $O(1)$ performance (luckily, several do!)

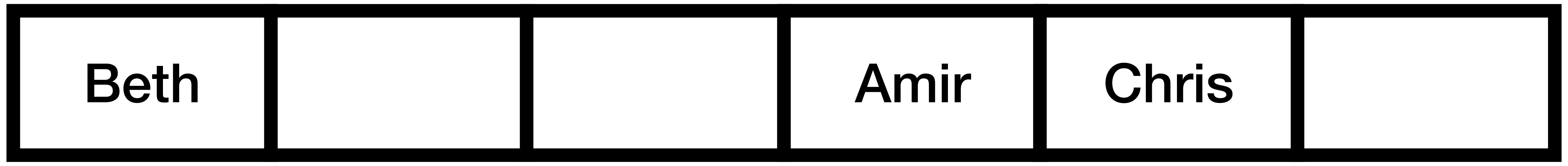
# Managing Collisions via Chaining

- Idea: store a linked list at each array entry (what kind?)
- When an item hashes to a slot, store it in the (possibly empty) linked list

Amir

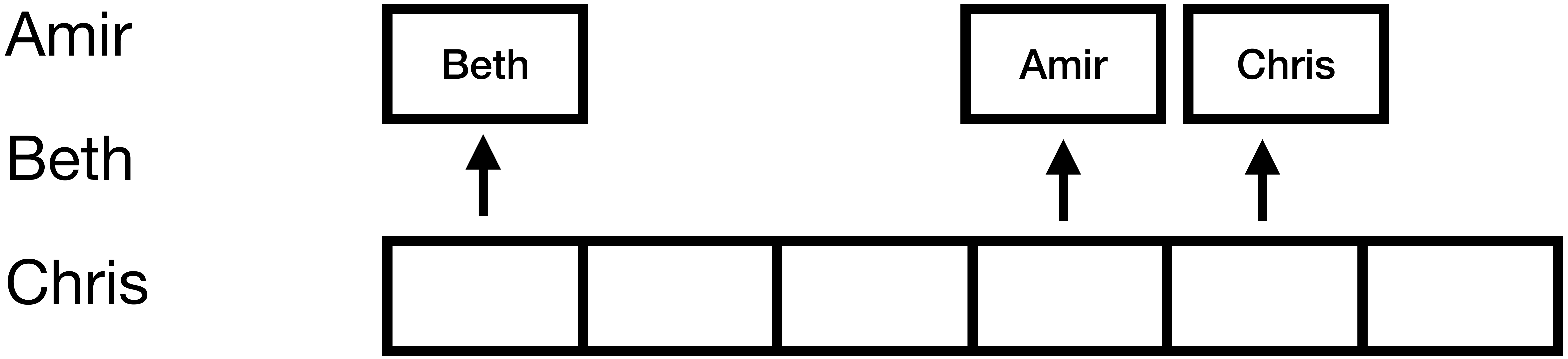
Beth

Chris



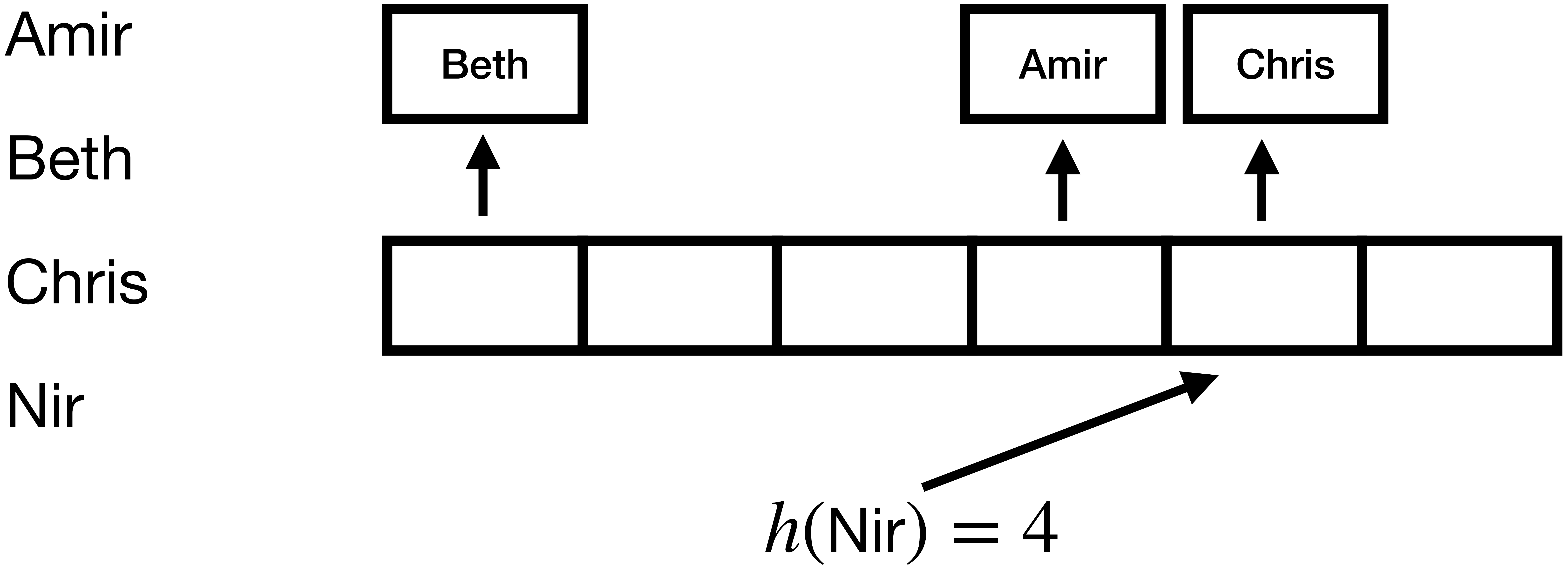
# Managing Collisions via Chaining

- Idea: store a linked list at each array entry (what kind?)
- When an item hashes to a slot, store it in the (possibly empty) linked list



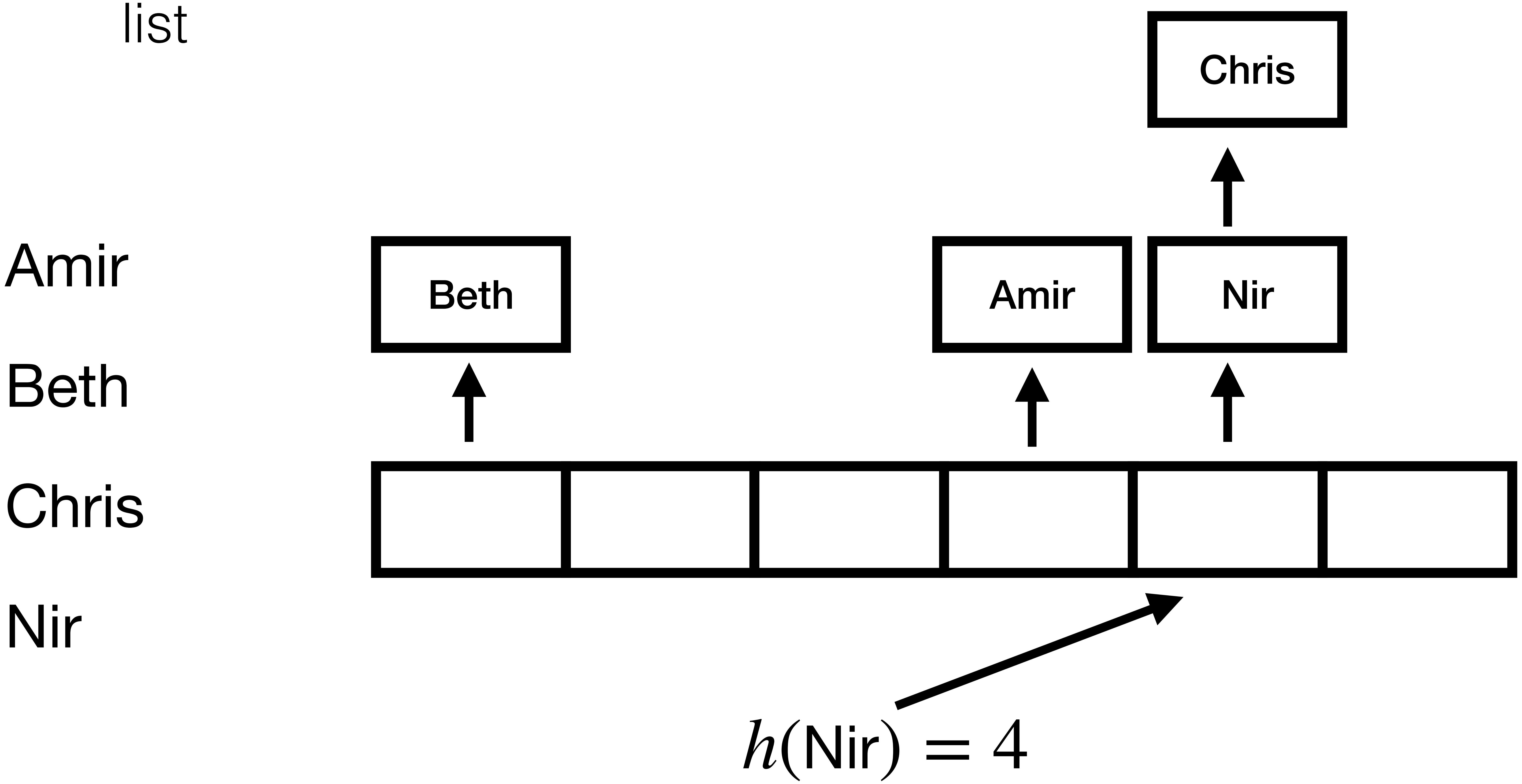
# Managing Collisions via Chaining

- Idea: store a linked list at each array entry (what kind?)
- When an item hashes to a slot, store it in the (possibly empty) linked list



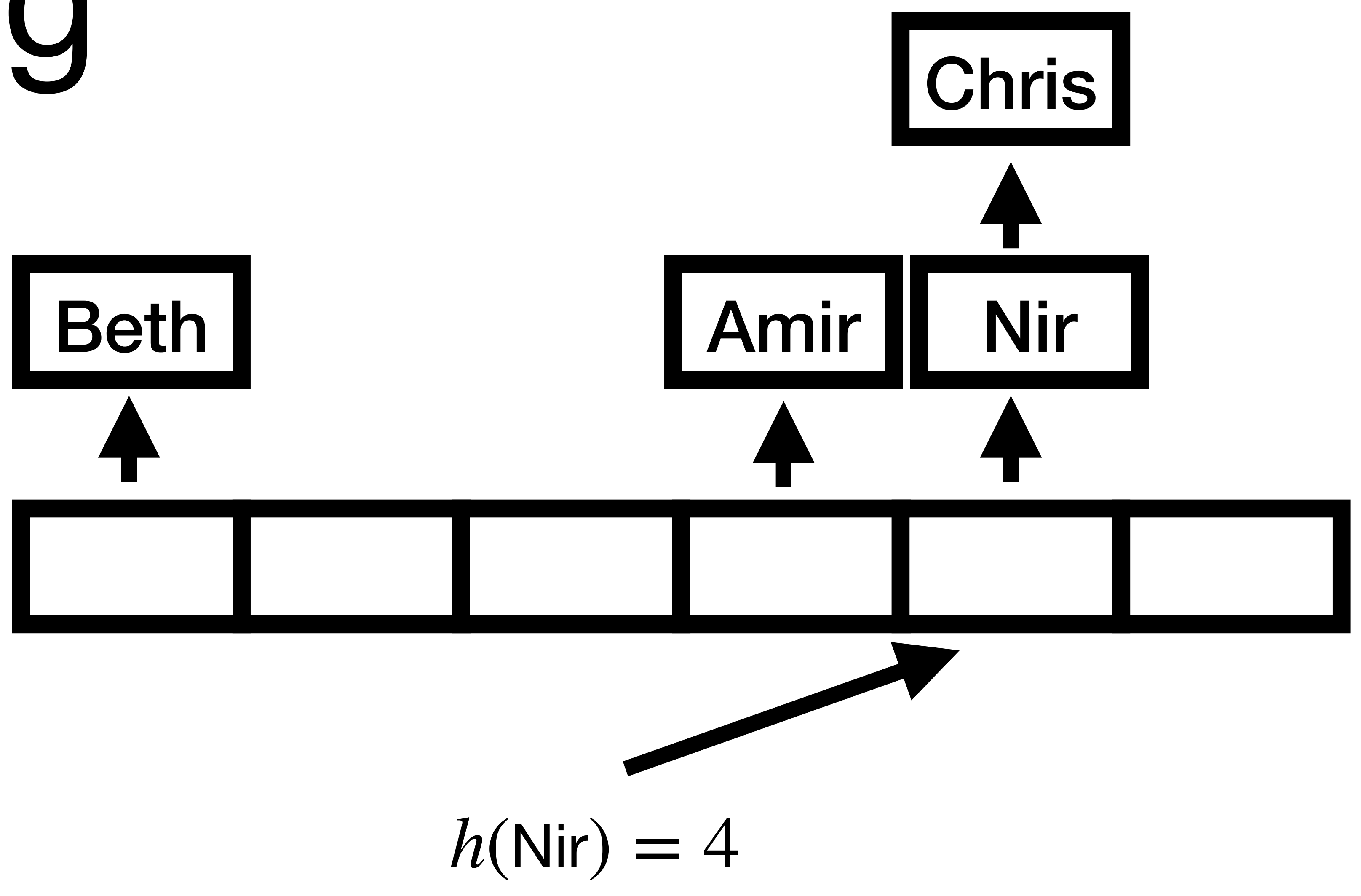
# Managing Collisions via Chaining

- Idea: store a linked list at each array entry (what kind?)
- When an item hashes to a slot, store it in the (possibly empty) linked list



# Managing Collisions via Chaining

- Store a doubly linked list at each array entry
- When an item hashes to a slot, **prepend** it to the linked list

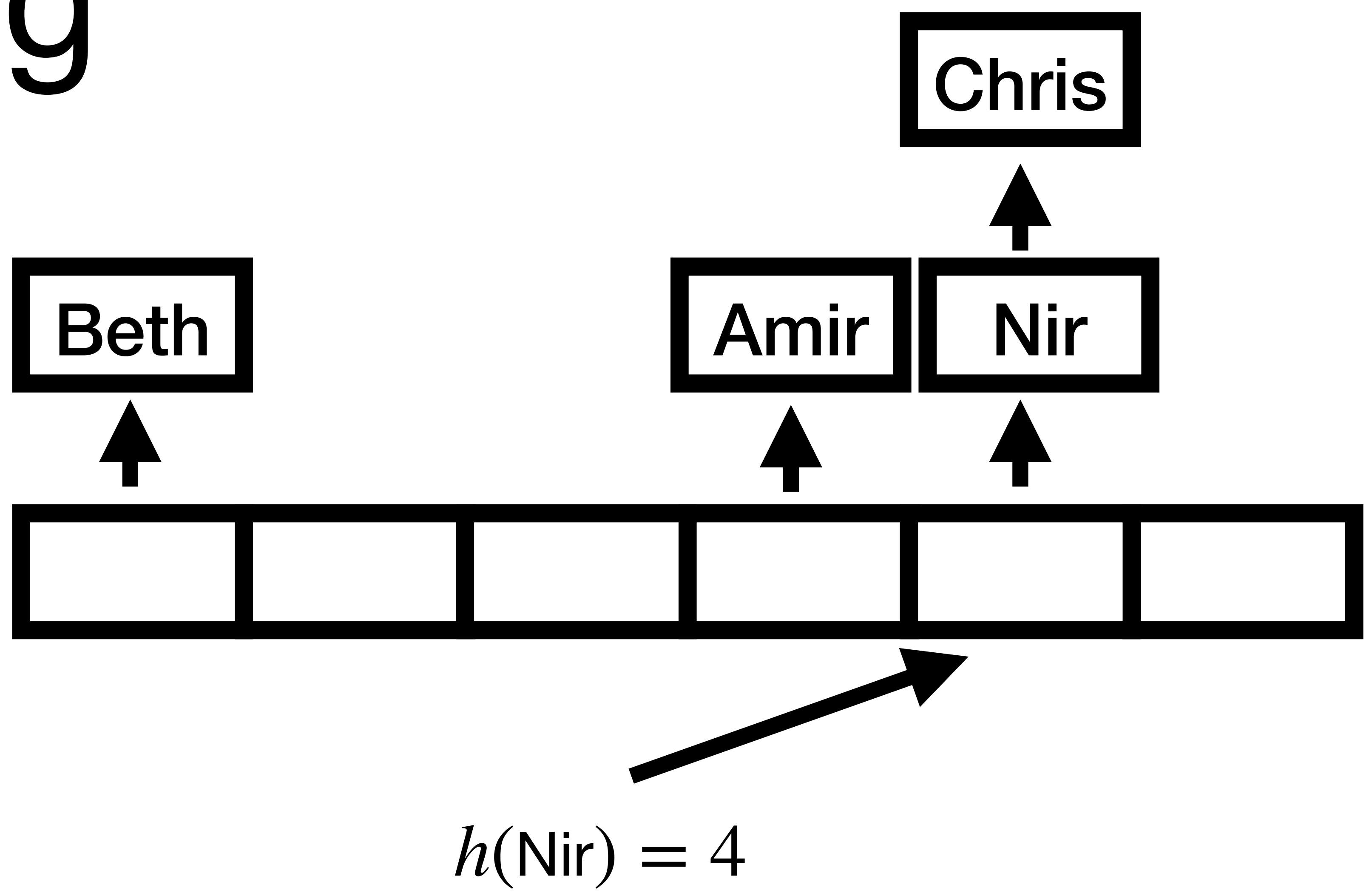



- How can we insert? (See above…)
- How can we lookup?
- How can we delete?

- (Harder) How much time do these operations take?

# Managing Collisions via Chaining

- Store a doubly linked list at each array entry
- When an item hashes to a slot, **prepend** it to the linked list



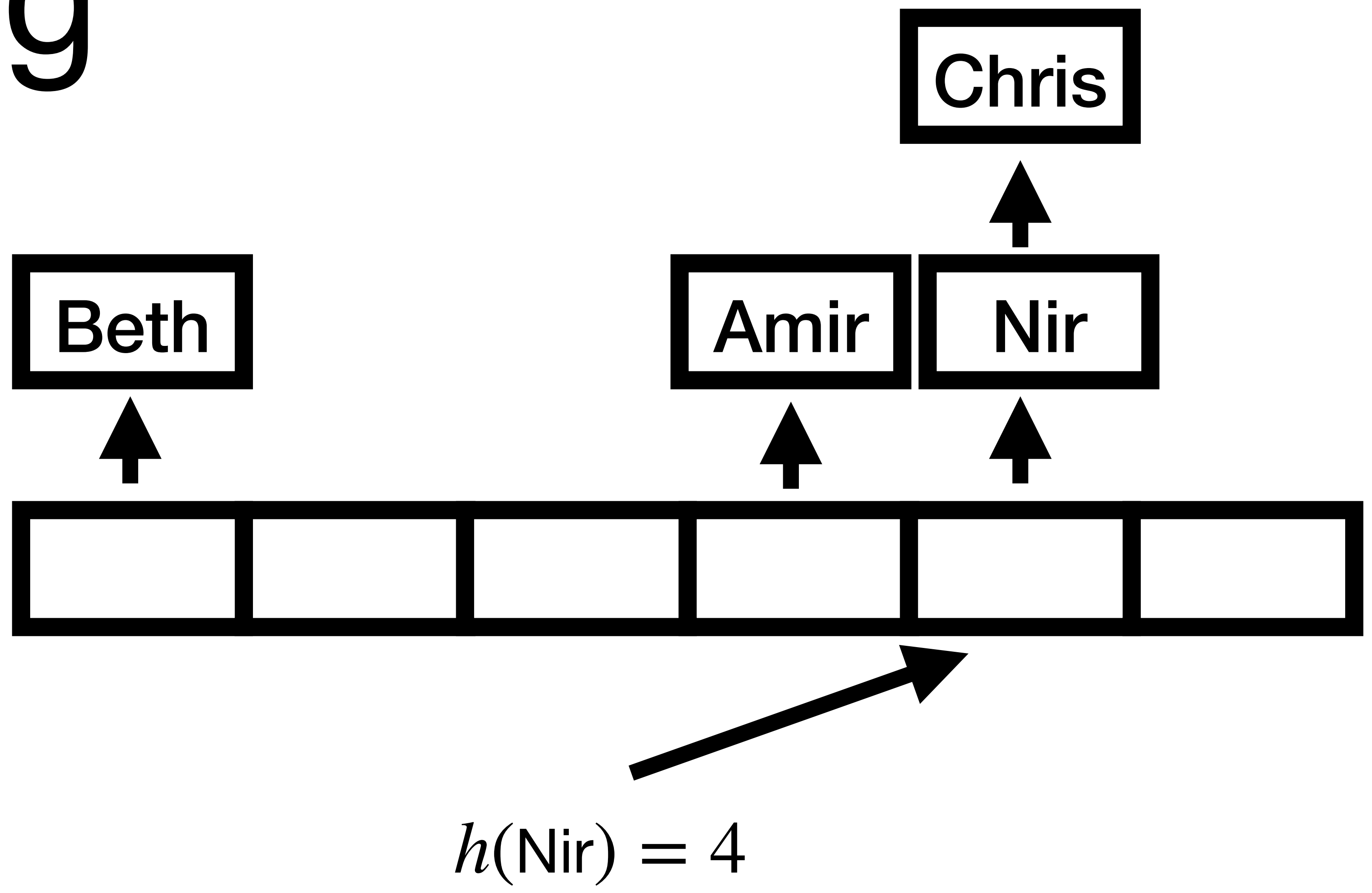

Insert($k$):

Prepend $k$ at the head of the list $A[h(k)]$

- Runtime?
  - $O(1)$ — exactly; not in expectation!
  - Note, we assume $k$ is not already in the hashtable
    - If don't want that assumption, do a lookup first!

# Managing Collisions via Chaining

- Store a doubly linked list at each array entry
- When an item hashes to a slot, **prepend** it to the linked list




Delete($k$):

Scan the list $A[h(k)]$, and delete the entry with key $k$

- Runtime?
  - $O(L)$, where $L$ is the length of the chain in slot $h(k)$
  - What do we expect $L$ to be?

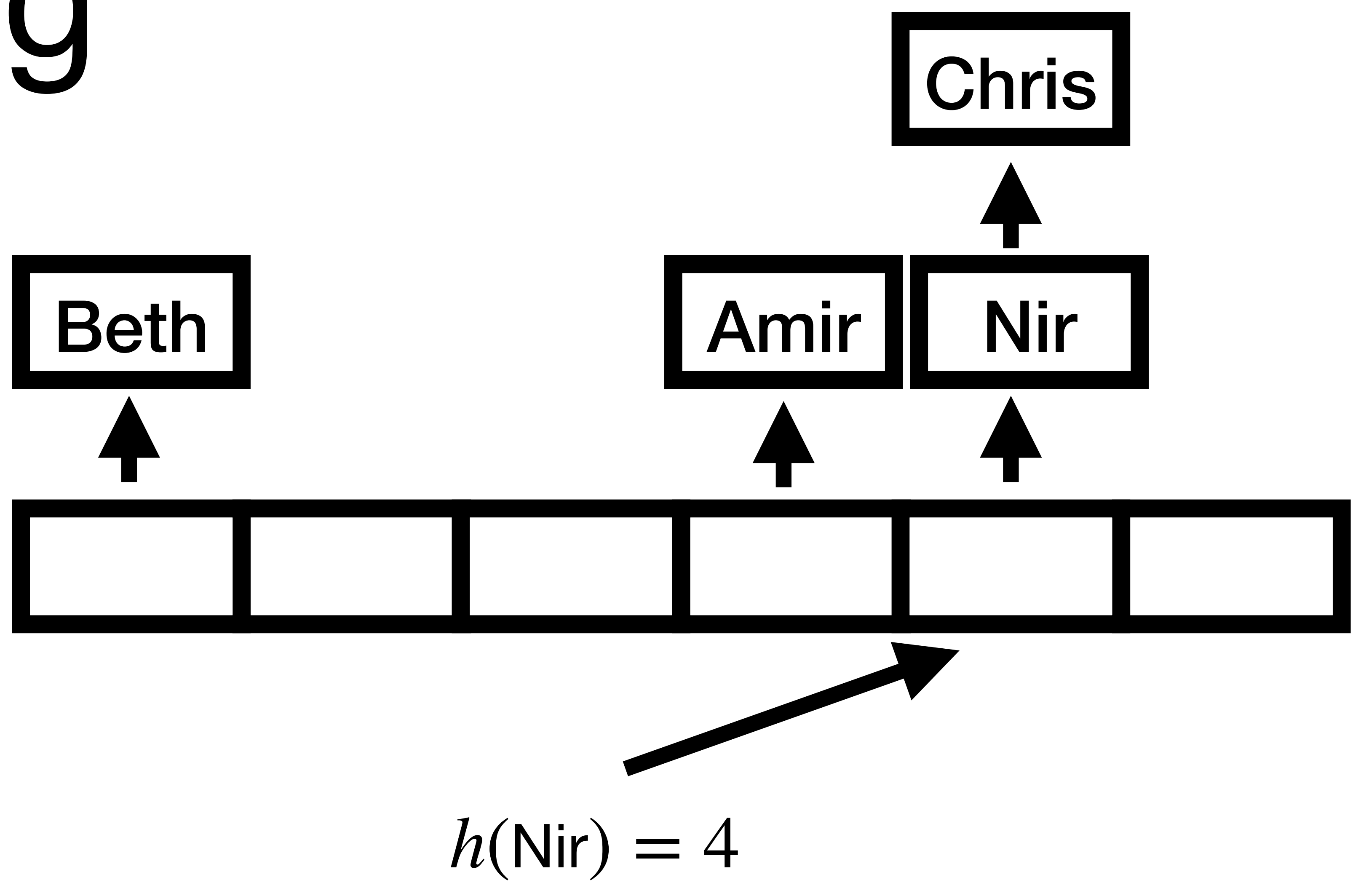
# Hashing and Chain Length

Worst-case delete time in a hash table with chaining: number of balls in a particular bin. **Question:** Expected number of balls in a particular bin $b$?

- Let $X_i$ denote indicator r.v. that item $i$ hashes to the bucket $b$
    - Assuming uniform hashing, $Pr(X_i = 1) = \frac{1}{m}$
- Let $X = \sum_{i=1}^{n} X_i$ denote the number of items that hash to bucket $b$
- By linearity of expectation, $E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m}$

# Managing Collisions via Chaining

- Store a doubly linked list at each array entry
- When an item hashes to a slot, **prepend** it to the linked list

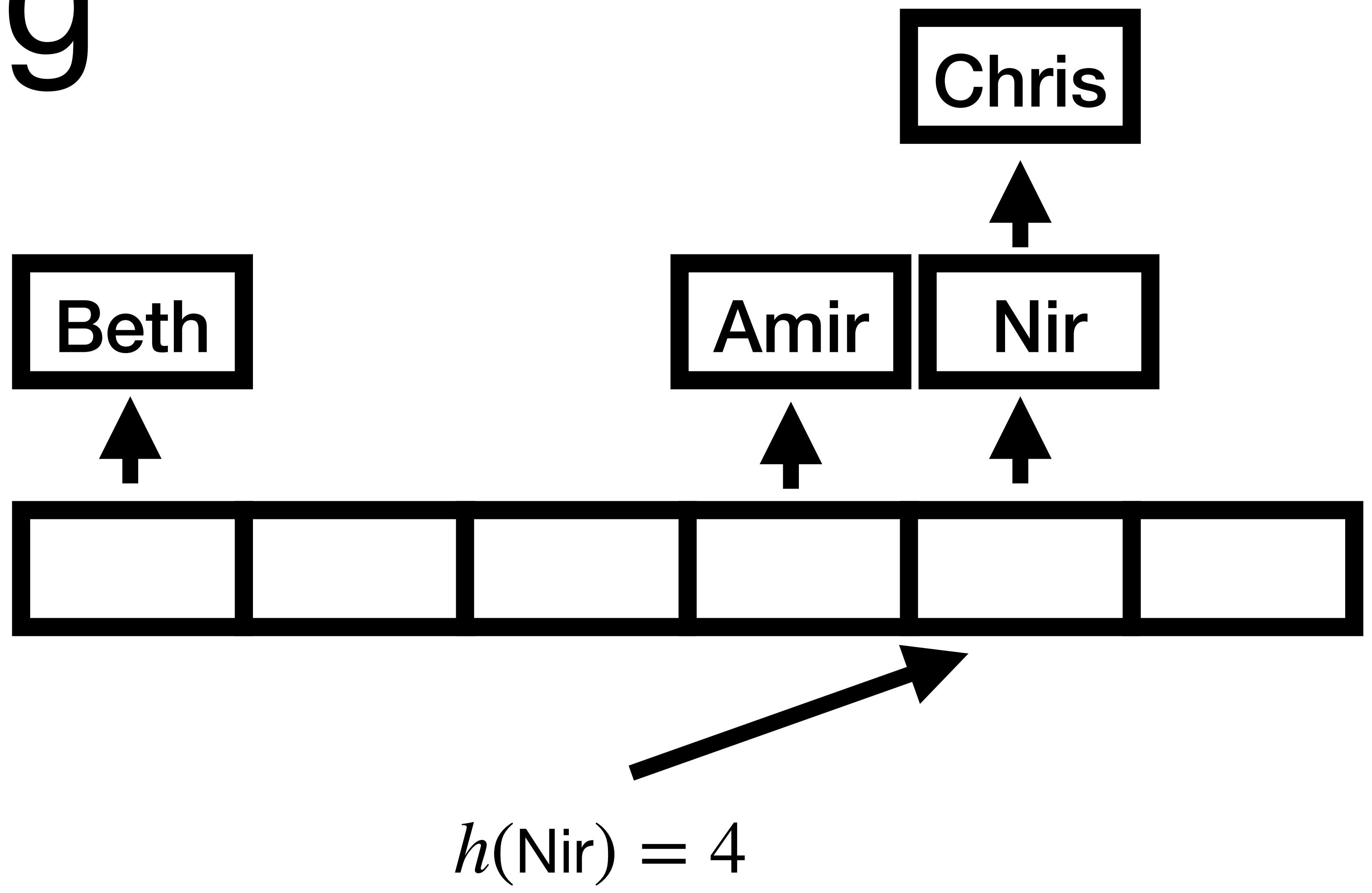



Delete($k$):

Scan the list $A[h(k)]$, and delete the entry with key $k$

- Runtime?
  - $O(L)$, where $L$ is the length of the chain in slot $h(k)$
  - What do we expect $L$ to be?
  - $E[L] = \dfrac{n}{m}$. We'll also call this the hashtable's **load factor**

# Managing Collisions via Chaining




- Store a doubly linked list at each array entry
- When an item hashes to a slot, **prepend** it to the linked list

Lookup($k$):

Scan the list $A[h(k)]$; return the entry with key $k$ if an entry exists

- Runtime?
    - (Surprisingly?) Lookup behavior is different in two cases!
        - "Successful" lookup vs. "unsuccessful"
            - Why?

# Hashing and Chain Length

Worst-case lookup time in a hash table with chaining: number of balls in a particular bin. **Question:** what's different about successful and unsuccessful cases?

- Unsuccessful lookup: must scan through entire chain
  - Cost is $O(L)$, and we showed that $E[L] = \dfrac{n}{m}$
- Successful lookup stops once we find the target element. The analysis is tricky because we always insert at the front of the list!
  - Expected cost to lookup item $x$ when $x$ is in the hashtable is the expected number of items that collided with $x$ ***after*** $x$ ***was inserted***

# Cost of Successful Lookup

- Assume that element $x$ is equally likely to be any of table's $n$ elements
  - Number of elements checked is 1 plus number of elements that appear before $x$ in list $A[h(x)]$
  - Observation: all elements are placed at the front of the list, so this is precisely the number of elements that:
    1. collided with $x$, and
    2. were inserted after $x$ was

# Cost of Successful Lookup

Expected number of collisions with $x$ that occur after $x$ is inserted?

- Let $x_i$ be the $i^{\text{th}}$ element inserted into the list
  - In other words, we insert $x_1, x_2, \ldots, x_n$ into $A$
- Let $X_{ij}$ be the indicator r.v. that equals 1 when $h(x_i) = h(x_j)$
  - Note: $X_{ij}$ is 1 when there is a collision between $x_i$ and $x_j$, 0 otherwise
- Under our uniform hashing assumption, $E[X_{ij}] = 1/m$
- With this, can we reason about the number of elements examined in a successful search?

# Cost of Successful Lookup

The expected number of elements examined in a successful search is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

Since $x$ may be any of the $n$ elements we insert, we average the contribution of each of the $n$ items

# of comparisons to find $x_i$ are 1 plus the expected number of collisions among all items inserted after $x_i$

# Cost of Successful Lookup

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

by Linearity of Expectation

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) = \frac{1}{n}\sum_{i=1}^{n}1 + \frac{1}{mn}\sum_{j=i+1}^{n}1$$

$$= 1 + \frac{1}{mn}\sum_{i=1}^{n}(n-i) = 1 + \frac{1}{mn}\left(\sum_{i=1}^{n}n - \sum_{i=1}^{n}i\right)$$

$$= 1 + \frac{1}{mn}\left(n^2 - \frac{n(n+1)}{2}\right) = 1 + \frac{1}{nm}\left(\frac{2n^2 - n^2 - n}{2}\right)$$

$$= 1 + \frac{n-1}{2m} = 1 + \frac{\frac{n}{m}}{2} - \frac{\frac{n}{m}}{2n} = O\left(1+\frac{n}{m}\right)$$

Same big-O!

# Hashtable Summary

We can get close to $O(1)$ performance for insert, lookup, and delete operations ($O(1 + n/m)$ in expectation, where $n/m$ can be controlled by resizing)

- There are other strategies for resolving collisions, but analyzing their performance is tricky
  - Linear probing: $h(k, i) = (h(k) + i) \mod m$
  - Quadratic probing: $h(k, i) = (h(k) + c_1 i + c_2 i^2) \mod m$
  - Double hashing: $h(k, i) = h(k \,||\, i)$
  - Power-of-two-choices: stored at $h_1(k)$ or $h_2(k)$, uses "cuckooing"

Hashtables are a great data structure for many applications

- As long as you don't need to iterate or sort!