Parallel And Sequential Data Structures and Algorithms

Dictionaries, Hash Tables, Hashing

Learning Objectives

- Understand **dictionaries** and their variants
- Review **Hash Tables** from 15-122, including separate chaining
- Apply randomization to hashing:
  - Define **random hash families**
  - Analyze collision probabilities using indicator random variables
  - Define **universal hash families**
  - Analyze the cost bounds of hash tables using universal hashing

Definitions

- A dictionary is… an **abstract data type**!

Interface (Dictionary): A `dictionary<K,V>` (or `dict<K,V>` for short) stores items (key-value pairs; i.e., `(K,V)` pairs), and supports:

- **`insert(`**`k : K, v : V`**`)`**:
  add the key-value pair `(k,v)` to the dictionary
- **`find(`**`k : K`**`) -> `**`option<V>`**:
  return the value for the key `k`, or `NONE` if it doesn't exist
- **`delete(`**`k : K`**`)`**:
  delete the item with the key `k`

Naming Conventions

- Dictionaries go by many different names:
  - Maps (e.g., in Java; `map/TreeMap/HashMap`, in C++ `map/unordered_map`, the `Map` module in Ocaml
  - Associative arrays or tables
- But "Map" conflicts with our higher-order function `map`
- "Dictionary" is the common term in algorithms textbooks, and of course is the name of the datatype in Python

Persistence versus Mutation

Definition (Persistent data structure): A persistent data structure is a data structure that preserves the old versions of itself when it is updated.

- Dictionaries can be persistent or non-persistent (mutable)
- A persistent dictionary would have signatures:
  - `insert(k : K, v : V) -> dict<K,V>`
  - `delete(k : K) -> dict<K,V>`
- A mutable dictionary would have signatures:
  - `insert(k : K, v : V) -> void`
  - `delete(k : K) -> void`

Variants

Interface (Sorted Dictionary): A `SortedDict<K,V>`, where `K` is orderable, supports the `dictionary` interface plus:

- **`first() -> option<(K,V)>:`**
  return the item with the least key (or `NONE` if empty)
- **`last() -> option<(K,V)>:`**
  return the item with the greatest key (or `NONE` if empty)
- **`prev(k : K) -> option<(K,V)>:`**
  return the item with the greatest key less than `k` (`NONE` if `k` is least)
- **`next(k : K) -> option<(K,V)>:`**
  return the item with the least key greater than `k` (NONE if `k` is greatest)

Variants

Interface (Set): A `set<K>` stores a set of keys and supports:

- **`insert(k : K)`:** return the item with the least key (or `NONE` if empty)
- **`contains() -> bool`:** return True if the set contains `k`, False otherwise
- **`delete(k : K)`:** delete `k` from the set
- **`union(S : set<K>) -> set<K>`:** return the union of this set and S
- **`intersection(S : set<K>) -> set<K>`:** return the intersection of this set and S
- **`difference(S : set<K>) -> set<K>`:** return the difference of this set and S

Variants

Interface (Sorted Set): A `SortedSet<K>`, where `K` is orderable, supports the `set` interface plus:

- **`first() -> option<K>:`**
  return the least key (or `NONE` if empty)
- **`last() -> option<K>:`**
  return greatest key (or `NONE` if empty)
- **`prev(k : K) -> option<K>:`**
  return the greatest key less than `k` (`NONE` if `k` is least)
- **`next(k : K) -> option<K>:`**
  return the least key greater than `k` (NONE if `k` is greatest)

ADT Summary

`dictionary<K,V>`:

- Stores key-value pairs
- supports `insert`, `find`, `delete`

`set<K>`:

- Stores a set of keys
- supports `insert`, `contains`, `delete`
- Also `union`, `intersection`, and `difference`

`SortedDict<K,V>`:

- Stores sorted key-value pairs
- Supports `insert`, `find`, `delete`
- Also, `first`, `last`, `prev`, `next`

`SortedSet<K>`:

- Stores a set of sorted keys
- supports `insert`, `contains`, `delete`
- Also, `union`, `intersection`, and `difference`
- Also, `first`, `last`, `prev`, `next`

Hash Tables

Hash Tables

- **Hash tables** are a family of data structures that can be used to implemented (non-sorted) dictionaries and sets.
- The goal of hash tables is to achieve the best possible cost bounds: constant time!!

Key Idea: This sounds too good to be true. Achieving constant-time operations for hash tables is going to require **randomization**!

- You studied hash tables in 15-122.
- The goal of today is to review and pick up where you left off, and understand the theory behind why they work so well

Background: Hash Tables

Keys: The keys come from some large **universe** U. On the word RAM (or fork-join RAM), we assume the universe is $U = \{0, \dots, u-1\}$ where $u = 2^w$

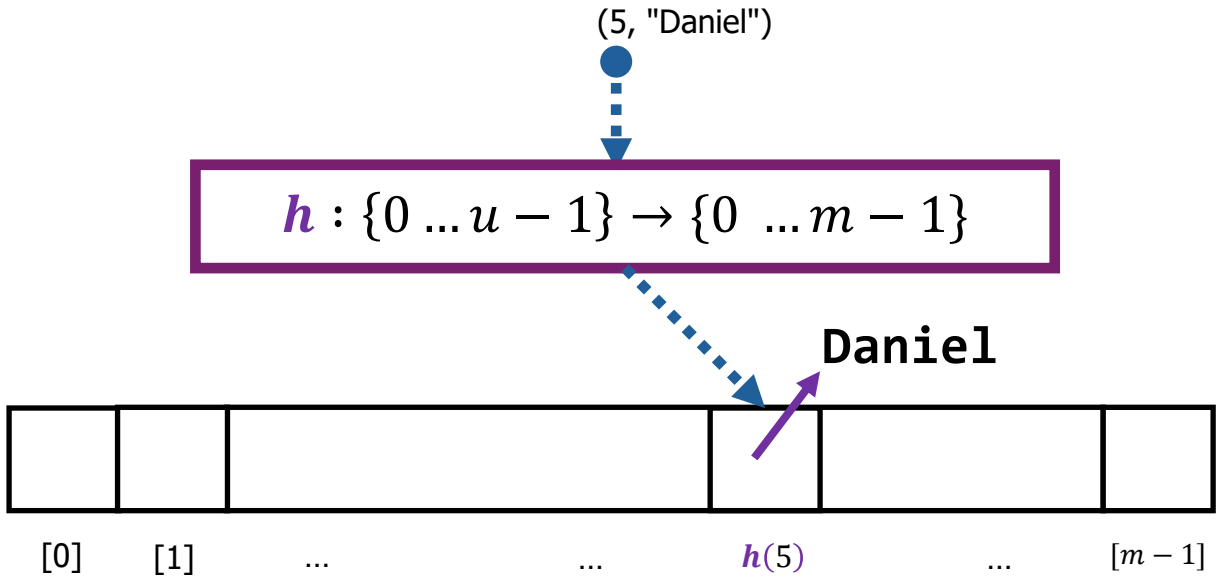
Hash Function: Given a universe U and an integer $m \leq |U|$ (typically, $m \ll |U|$), a hash function is a mapping from U to $\{0, \dots, m-1\}$, i.e.,

$$h: U \to \{0, \dots, m-1\}$$

- Intuitively, a hash function maps from a large space to a small one
- The output of a hash function is called a "hash value" or "hashcode"

Hash Table: Basics and Notation

- A hash table stores a set of items (key-value pairs)
- n items are stored in an array A of size m
- An item with key x wants to be stored at location $A[h(x)]$



Collision Resolution

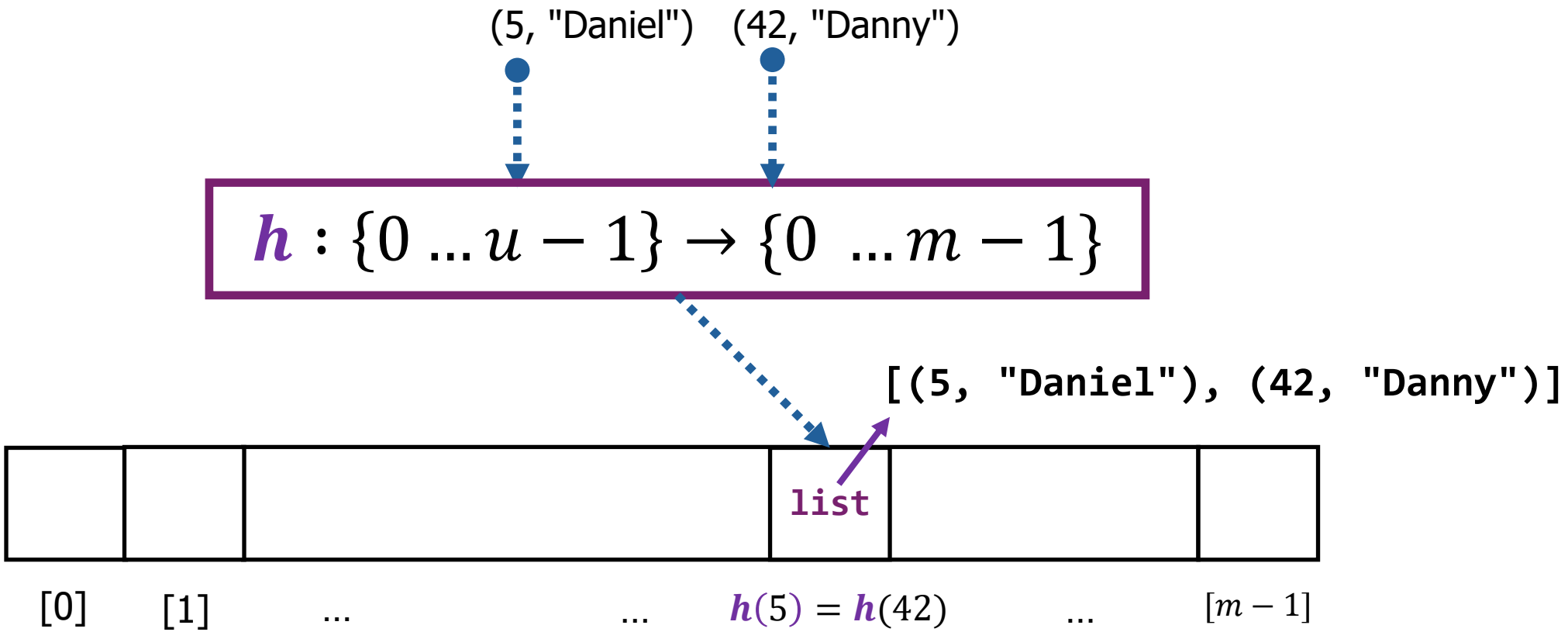
- This strategy falls apart when $h(x) = h(y)$ for some $x \neq y$
- How do we store both x and y?

Definition (Collision): A pair of distinct keys x, y collide for a given hash function h if $h(x) = h(y)$

Open Addressing: Find an alternate open slot in the array and use that

Closed Addressing: Store colliding items in a separate data structure

Open Addressing: "Chaining"



Efficiency

Theorem (Efficiency of Chaining): Assume evaluating $h(x)$ costs $O(1)$, then with separate chaining, `insert(x,_)`, `find(x)`, and `delete(x)` cost

$$\Theta(1 + |A[h(x)]|)$$

Proof:

- *Evaluating $h(x)$ costs $O(1)$ by assumptions*
- *Each operation just searches the list at $h(x)$*

Goal: A good hashing scheme should therefore aim to keep the lists short

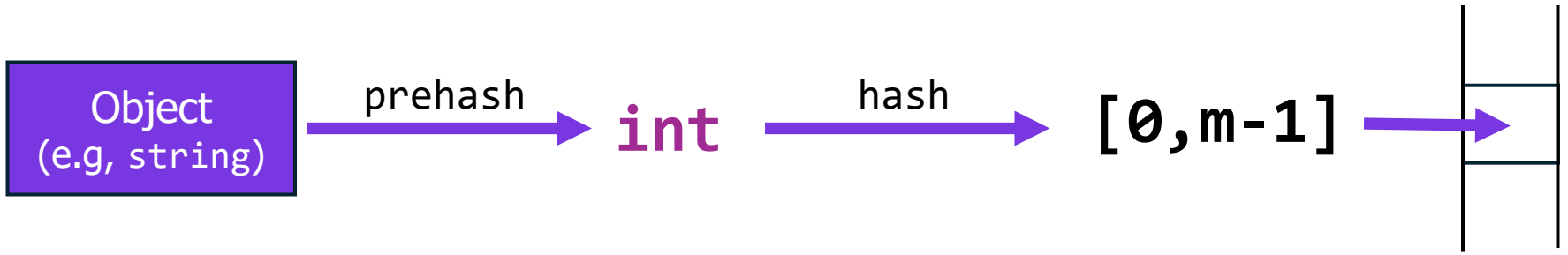
Terminology: Hashing vs Prehashing

- This setup of hashing assumes keys are integers. In real life, we often have other key types, e.g., strings are very common.

Definition (Prehash): A prehash converts an object into an integer.

Note: The term "prehashing" is only used in theory. In most programming languages, they just call prehashing "hashing". E.g., Python's `hash` or C++'s `std::hash`.

Theory Terms:



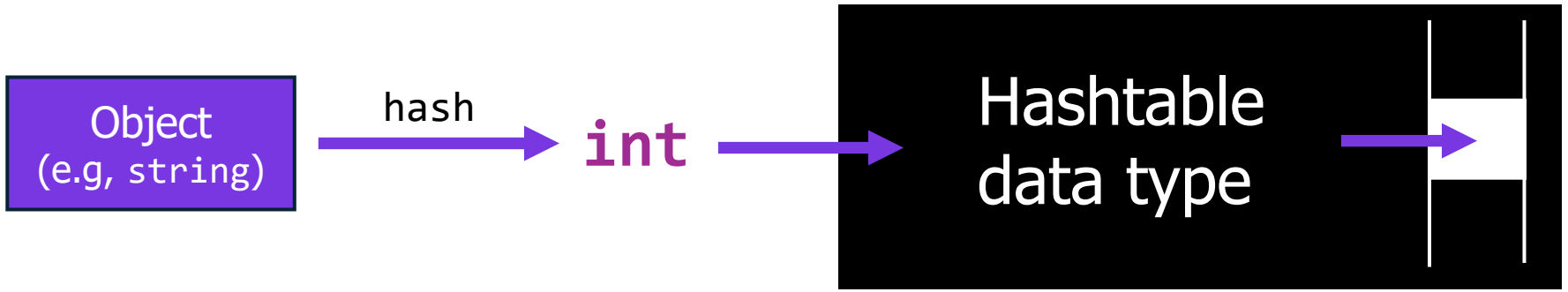
Terminology: Hashing vs Prehashing

- This setup of hashing assumes keys are integers. In real life, we often have other key types, e.g., strings are very common.

Definition (Prehash): A prehash converts an object into an integer.

Note: The term "prehashing" is only used in theory. In most programming languages, they just call prehashing "hashing". E.g., Python's `hash` or C++'s `std::hash`.

Real Life Terms:



Hash Functions

Desirable Properties

Desirable: A good hashing scheme should:

- Minimize the number of collisions to keep $|A[h(x)]|$ short
- Require $m = O(n)$, i.e., we don't want to use more than linear space
- Use a function h that can be computed in $O(1)$

Theorem (Collisions are Inevitable): For any hash function h, if $|U| > (n-1)m$, then there exists a set of n items with the same hashcode

Proof: By the pigeonhole principle, if each hashcode had at most $n-1$ keys that hash to it, then there are at most $(n-1)m$ keys in the universe

So, Hashing is Bad in Theory?

Myth: Hashing is good in practice, but "bad in theory" because for any hash function, I can find a set of keys that makes operations cost $\Theta(n)$.

- Hashing is "bad" for **any particular fixed hash function**, because you can make an input to fool it.
- Just like Quicksort was "bad" for any particular fixed pivot choice, because you could contrive an input to make it do terrible splits

Randomization to the rescue!

Randomized hash tables choose a hash function to use at random.

Randomized Hashing

Randomized hash tables choose a hash function to use at random. Just like randomized quicksort chose pivots at random.

- **Important question**: How do we even analyze a randomized data structure? What does this statement actually mean?

"find(x) costs O(1) in expectation"

Analyzing Random Data Structures

"find(x) costs O(1) in expectation"

1. An adversary chooses any sequence of operations (`insert`, `find`, `delete`) for our hash table
2. We (the algorithm) choose a random hash function h
3. Evaluate the cost of every operation in the sequence

Expected cost means the expected value **over the random choice of hash function** of the cost of the operation

Analyzing Random Data Structures

"find(x) costs O(1) in expectation"

Expected cost means the expected value **over the random choice of hash function** of the cost of the operation

- The quantifier order is very important:

for **any** sequence of operations,
the expected value over the random choice of hash function,
for that operation, is $O(f(n))$.

Confusion with Other Cost Measures

- **Misconception**: The expected value is the average cost over all possible inputs

No, that's called **average-case** cost (15-122)

- **Misconception**: The expected value is the average cost over all operations in the sequence

No, that's called **amortized** cost (15-122)

Expected cost means the average cost **over the random choices made by the algorithm** (e.g., random pivots, random hash function)

Random Hashing

Random Hashing

```
fun hash(x : int) -> int:
  return random_integer(0, m)   // extremely wrong!!
```

- What's wrong with this?

This isn't a "function" (i.e., pure function), it doesn't return the same output for the same input every time. Useless for hashing.

- The hash function we use must be deterministic
- The randomness comes from **choosing** a hash function from a random family of hash functions

Totally Random Hashing

Definition (Totally Random Hashing) A hash function is called **totally random** if it is selected uniformly at random from the set of all possible hash functions $h : U \to \{0,1, \dots, m-1\}$

This is equivalent to choosing, for each key $x \in U$, **an independent, uniformly random hash code** in $\{0,1, \dots, m-1\}$

- Watch out for confusing terminology: a random hash function is not a function that behaves randomly; it behaves deterministically, but it is **chosen randomly** from a set of possible hash functions

Collision Probability Analysis

Theorem (Collisions are Rare): Let $\mathcal{H}$ be the family of all possible hash functions $h : U \to \{0,1,\ldots,m-1\}$. Then, for all $x, y \in U$ such that $x \neq y$:

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{m}$$

Proof

$$\Pr_{h \in \mathcal{H}}[h(x)=h(y)] = \sum_{0 \le i < m} \Pr[h(x)=h(y) | h(x)=i] \cdot \Pr[h(x)=i]$$ Law of total probability

$$= \sum_{0 \le i < m} \Pr[h(y)=i | h(x)=i] \cdot \Pr[h(x)=i]$$

$$= \sum_{0 \le i < m} \Pr[h(y)=i] \cdot \Pr[h(x)=i]$$ Independence

$$= \sum_{0 \le i < m} \left(\frac{1}{m}\right)\left(\frac{1}{m}\right) = m\left(\frac{1}{m}\right)^2 = \frac{1}{m}$$

Space Requirement (Bad News)

Theorem (Space of Totally Random Hashing): To store any total function $h : U \to \{0,1,\ldots,m-1\}$ from the set of all possible such functions requires at least $u \cdot \log_2(m)$ bits of space

(Handwavy) Proof:

- There are u possible keys, each needs a hashcode of $\log_2 m$ bits
- So, this takes $u \cdot \log_2(m)$ bits of space

Information theory says randomness is not compressible. There are m^u possible hash functions. Storing one requires $\log_2(m^u) = u \cdot \log_2(m)$ bits

We Need a Trade-Off

- Totally random hashing gives us the smallest possible collision probability of $1/m$ in the worst case.
- But it uses too much space to be feasible

We need a family of hash functions that is efficient to compute and store but also yields low collision probability

Universal Hashing

Universal Hashing

Definition (Universal Family) A set (or family) $\mathcal{H}$ of hash functions $h : U \to \{0, \dots, m-1\}$ is called **a universal family** if for all $x \neq y$

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$$

Can compute probability by counting:

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{|h(x) = h(y)|_{h \in \mathcal{H}}}{|\mathcal{H}|}$$

Examples

We won't prove that families are universal, that requires a lot of math and a lot of patience. See **15-451** for how to prove that sets are universal!

- The set of **all possible hash functions** is universal
  - We proved this before, by showing that $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$
- The original universal hash family, by its inventors
  - Pick a prime $p \geq u$
  - Pick **random** integers $0 < a < p$, $0 \leq b < p$
  - Define:

$$h_{a,b}(x) = \big((ax + b) \bmod p\big) \bmod m$$

An Efficient Universal Hash

Theorem (Universality): The family of hash functions defined for $p \geq u$, over the random choice of $0 < a < p$ and $0 \leq b < p$ is universal.

$$h_{a,b}(x) = \big((ax + b) \bmod p\big) \bmod m$$

- Evaluating $h_{a,b}(x)$ takes $O(1)$ time
- Storing $h_{a,b}(x)$ requires storing three integers: $O(1)$ space

Applying Universal Hashing

Theorem (Universal Hashing): Let $\mathcal{H}$ be universal and let $S \subseteq U$ be a set of n keys. For any key $x \in S$, if $h \in \mathcal{H}$ is drawn randomly, the expected number of collisions between x and all other keys in S is less than n/m

Proof:

- Let C_{xy} be an indicator random variable for the event that x and y collide
- Let C_x be a random variable denoting the total number of collisions for x

$$C_x = \sum_{\substack{y \in S \\ y \neq x}} C_{xy} \xrightarrow{\text{Linearity of expectation}} \mathbb{E}[C_x] = \sum_{\substack{y \in S \\ y \neq x}} \mathbb{E}[C_{xy}] \xrightarrow[\Pr[h(x) = h(y)] \leq \frac{1}{m}]{\text{Universal}} \leq \frac{|S|-1}{m} = \frac{n-1}{m}$$

Making Hash Tables Efficient

Corollary (Chaining with Universal Hashing): With separate chaining and a hash function drawn from a universal family, on a hash table of capacity m containing n keys, insert, find, and delete cost $\Theta(1 + \alpha)$ in expectation, where $\alpha = n/m$

- *Proof: We argued that insert, find, and delete cost*

$$O(1 + |A[h(x)]|)$$

- *The expected number of collisions with x is less than $n/m = \alpha$*
- *Therefore,* $\mathbb{E}\left[|A[h(x)]|\right] \leq 1 + \alpha$

Final Cost Bounds…

- We have cost bounds in terms of $\alpha = n/m$ when the array has capacity m and we have n keys in the hash table
  - `insert`: $\Theta(1 + \alpha)$ in expectation
  - `find`: $\Theta(1 + \alpha)$ in expectation
  - `delete`: $\Theta(1 + \alpha)$ in expectation

This is constant if the table never exceeds constant load factor, e.g., if $n \leq m$ for all operations, then every operation is constant expected time

Final Cost Bounds…

- We have cost bounds in terms of $\alpha = n/m$ when the array has capacity m and we have n keys in the hash table

Table of size m

- `insert`: $\Theta(1+\alpha)$ in expectation
- `find`: $\Theta(1+\alpha)$ in expectation
- `delete`: $\Theta(1+\alpha)$ in expectation

Dynamic-size table

- `insert`: $\Theta(1)$ **amortized** in expectation
- `find`: $\Theta(1)$ in expectation
- `delete`: $\Theta(1)$ in expectation

From 15-122 If the table exceeds a constant threshold load factor, e.g., if $\alpha > 1$ after any insertion, **double the value of m** and rebuild the table with a new randomly chosen hash function

Summary

- Dictionaries, Sets, Sorted Dictionaries, and Sorted Sets are abstract data types, the former two can be implemented as hash tables.
- Hash tables with separate chaining are efficient **if the chains are kept small**
- We can keep the chains small using **randomization**; picking a hash function randomly from a family of hash functions
- **Universal hashing** allows us to implement dynamic hashtables with $\Theta(1)$ cost in expectation for find and delete, and $\Theta(1)$ amortized cost in expectation for insert.