Dictionaries
What is a dictionary?

Maintain a collection of key–value pairs

- Keys are often integers or strings but can be other types
- At most one copy of each different key
- Values may be any type of data

Operations update the collection and look up the value associated with a given key

Dictionary in everyday usage: book of definitions of words. So the keys are words and the values are their definitions
Example application

We’ve already seen one in week 1!

In the depth-first-search example, we represented the input graph as a dictionary with keys = vertices and values = collections of neighboring vertices.

The algorithm didn’t need to know what kind of object the vertices are, only that they are usable as keys in the dictionary.
Dictionary API

Create new empty dictionary

Look up key $k$ and return the associated value
  (Exception if $k$ is not included in the collection)

Add key–value pair $(k, x)$ to the collection
  (Replace old value if $k$ is already in the collection)

Check whether key $k$ is in the collection and return Boolean result

Enumerate pairs in the collection
Dictionaries in Java and Python

Python: dict type

- Create: \( D = \{ \text{key1: value1, key2: value2, ...} \} \)
- Get value: \( \text{value} = D[\text{key}] \)
- Set value: \( D[\text{key}] = \text{value} \)
- Test membership: if \( \text{key} \) in \( D \):
- List key–value pairs: for \( \text{key}, \text{value} \) in \( D\text{.items()} \):

Java: HashMap

Similar access methods with different syntax
def count_occurrences(sequence):
    D = {}  # Create dictionary
    for x in sequence:
        if x in D:  # Test membership
            D[x] += 1  # Get and then set
        else:
            D[x] = 1  # Set
    return D
Non-hashing dictionaries: Association list

Store unsorted collection of key–value pairs

- Very slow (each get/set must scan whole dictionary), $O(n)$ time per operation where $n$ is \# key–value pairs
- Can be ok when you know entire dictionary will have size $O(1)$
- We will use these as a subroutine in hash chaining
Non-hashing dictionaries: Direct addressing

Use key as index into an array

- Only works when keys are small integers
- Wastes space unless most keys are present
- Fast: $O(1)$ time to look up a key or change its value
- Important as motivation for hashing
Non-hashing dictionaries: Search trees

Binary search trees, B-trees, tries, and flat trees

- We’ll see these in weeks 6 and 7!
  (Until then, you won’t need to know much about them)
- Unnecessarily slow if you need only dictionary operations
  (searching for exact matches)
- But they can be useful for other kinds of query
  (inexact matches)
Hashing
Hash table intuition

The short version:

Use a **hash function** $h(k)$ to map keys to small integers

Use direct addressing with key $h(k)$ instead of $k$ itself
Hash table intuition

Maintain a (dynamic) array $A$ whose cells store key–value pairs

Construct and use a hash function $h(k)$ that “randomly” scrambles the keys, mapping them to positions in $A$

Store key–value pair $(k, x)$ in cell $A[h(k)]$ and do lookups by checking whether the pair stored in that cell has the correct key

When table doubles in size, update $h$ for the larger set of positions

All operations: $O(1)$ amortized time (assume computing $h$ is fast)

Complication: Collisions. What happens when two keys $k_1$ and $k_2$ have the same hash value $h(k_1) = h(k_2)$?
Hash functions: Perfect hashing

Sometimes, you can construct a function $h$ that maps the $n$ keys one-to-one to the integers 0, 1, \ldots, $n - 1$.

By definition, there are no collisions!

Works when set of keys is small, fixed, and known in advance, so can spend a lot of time searching for perfect hash function.

Example: reserved words in programming languages / compilers

Use fixed (not dynamic) array $A$ of size $n$, store key–value pair $(k, x)$ in cell $A[h(k)]$ (include value so can detect invalid keys).
Hash functions: Random

Standard assumption for analysis of hashing:

The value of $h(k)$ is a random number, independent of the values of $h$ on all the other keys

Not actually true in most applications of hashing

Results in this model are not mathematically valid for non-randomly-chosen hash functions

(nevertheless, analysis tends to match practical performance because many nonrandom functions behave like random)

This assumption is valid for Java IdentityHashMap:
Each object has hash value randomly chosen at its creation
There has been much research in cryptographic hash functions that map arbitrary information to large integers (e.g. 512 bits).

Could be used for hash functions in dictionaries by taking result modulo $n$.

Any detectable difference between the results and a random function $\Rightarrow$ the cryptographic hash is considered broken.

Too slow to be practical for most purposes.
Hash functions: Fast, practical, and provable

It’s possible to construct hash functions that are fast and practical ... and at the same time use them in valid mathematical analysis of hashing algorithms

Details depend on the choice of hashing algorithm

We’ll see more on this topic later this week!
Quick review of probability
Basic concepts

Random event: Something that might or might not happen

Probability of an event: a number, the fraction of times that an event will happen over many repetitions of an experiment

\[ \Pr[X] \]

Discrete random variable: Can take one of finitely many values, with (possibly different) probabilities summing to one

Uniformly random variable: Each value is equally likely

Independent variables: Knowing the value of any subset doesn’t help predict the rest (their probabilities stay the same)
**Expected values**

If $R$ is any function $R(X)$ of the random choices we’re making, its **expected value** is just weighted average of its values:

$$E[R] = \sum_{\text{outcome } X} \Pr[X] R(X)$$

Linearity of expectations: For all collections of functions $R_i$,

$$\sum_i E[R_i] = E\left[\sum_i R_i\right]$$

(It expands into a double summation, can do sums in either order)

If $X$ can only take the values 0 or 1, then $E[X] = \Pr[X = 1]$

So $E[\text{number of } X_i \text{ that are one}] = \sum_i \Pr[X_i = 1]$
Analysis of random algorithms

We will analyze algorithms that use randomly-chosen numbers to guide their decisions

...but on inputs that are not random (usually worst-case)

The amount of time (number of steps) that such an algorithm takes is a random variable

Most common measure of time complexity: $E[\text{time}]$

Also used: “with high probability”, meaning that the probability of seeing a given time bound is $1 - o(1)$
Chernoff bound: Intuition

Suppose you have a collection of random events
(independent, but possibly with different probabilities)

Define a random variable $X$: how many of these events happen

Main idea: $X$ is very unlikely to be far away from $E[X]$

Example: Flip a coin $n$ times
$X = \text{how many times do you flip heads?}$

Very unlikely to be far away from $E[X] = n/2$
Chernoff bound (multiplicative form)

Let $X$ be a sum of independent 0–1 random variables. Then for any $c > 1$, 

$$
\Pr[X \geq c E[X]] \leq \left( \frac{e^{c-1}}{c^c} \right)^{E[X]}
$$

(where $e \approx 2.718281828\ldots$, “Euler’s constant”)

Similar formula for the probability that $X$ is at most $E[X]/c$
Three special cases of Chernoff

\[ \Pr[X \geq c E[X]] \leq \left( \frac{e^{c-1}}{c^c} \right)^{E[X]} \]

If \( c \) is constant, but \( E[X] \) is large \( \Rightarrow \)
probability is an exponentially small function of \( E[X] \)

If \( E[X] \) is constant, but \( c \) is large \( \Rightarrow \)
probability is \( \leq 1/c^{\Theta(c)} \), even smaller than exponential in \( c \)

If \( c \) is close to one (\( c = 1 + \delta \) for \( 0 < \delta < 1 \)) \( \Rightarrow \)
probability is \( \leq e^{-\delta^2 E[X]/3} \)