Randomized algorithms 1
Intro, hashing

CS240 Spring 2022

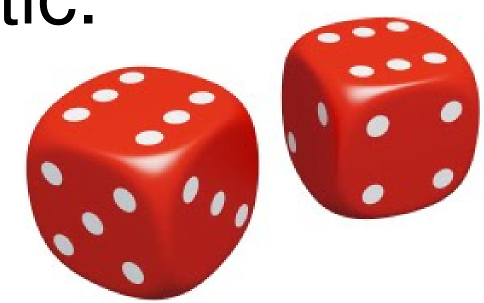
Rui Fan

Outline

- Introduction
- Probability review
- Max-cut and randomized quicksort
- Hashing
  - Closed addressing
  - Universal hashing
  - Perfect hashing

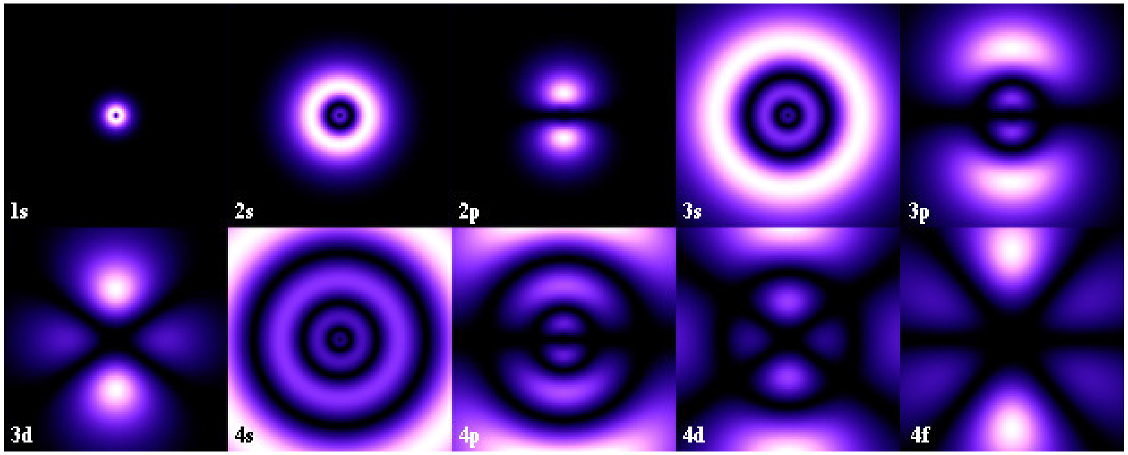
Randomized algorithms

- Till now, all of our algorithms have been deterministic.
  - Given an input, the algorithm always does the same thing.
- It turns out it's very useful to allow algorithms to be nondeterministic.
  - As the algorithm operates, it's allowed to make some random choices.
  - Running the algorithm multiple times on same input can produce different behaviors.



Why randomized algorithms?

- For many problems, randomized algorithms work better than deterministic ones.
  - Faster / uses less memory
  - Simpler, easier to understand.
  - Some problems that provably can't be solved (or solved efficiently) by deterministic algorithms can be solved by randomized ones.
  - According to quantum mechanics, the world is inherently probabilistic, so nature is randomized!

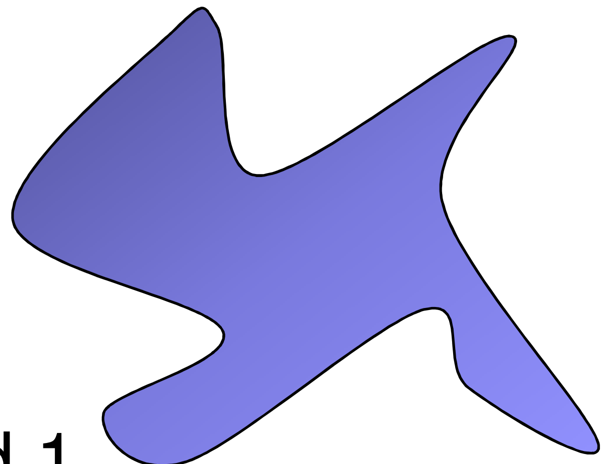


How can randomness help?

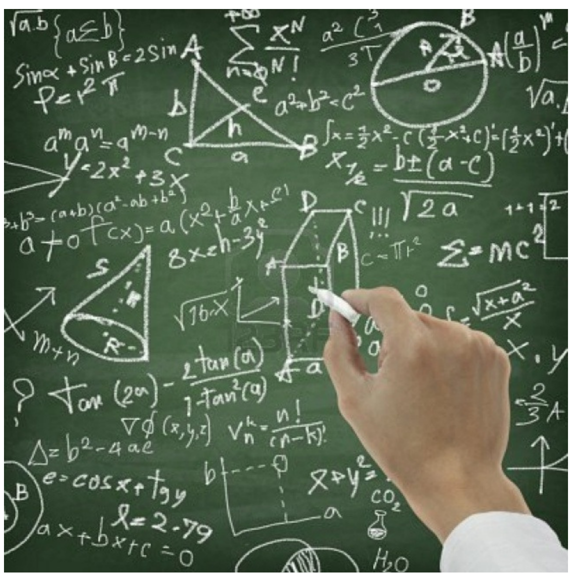
- Say you have a string of length n that's half A's and half B's.
- We want to find a location in the string with an A.
- Any deterministic algorithm takes n/2+1 steps in the worst case.
- But by checking random locations, a randomized algorithm finds an A in 2 steps in expectation.

How can randomness help?

- Measure the area of this



- Method 1



- Method 2
    - Print the shape out on a piece of paper.
    - Throw 100 darts at it.
    - See what percent land in the shape.
    - Multiply by area of your paper.

Las Vegas vs Monte Carlo

- A Las Vegas randomized algorithm always produces the right answer. But it's running time can vary depending on its random choices.
  - We want to minimize the expected running time of a Las Vegas algorithm.
- A Monte Carlo algorithm always has the same running time. But it sometimes produces the wrong answer, depending on its random choices.
  - We want to minimize the error probability of a Monte Carlo algorithm.

Probability review

- Discrete probability theory is based on events and their probabilities.
  - Events can be composed of more basic events.
  - Ex Event of rolling a 2 on a fair dice, with probability 1/6.
  - Ex Event of rolling an even number, with probability ½. Composed of basic events of rolling a 2, 4 or 6.
  - If A is event, write Pr[A]=y. E.g. Pr[roll a 2]=1/6
- Two events A, B are independent if $Pr[A \wedge B]=Pr[A]*Pr[B]$.
  - Ex Events A="2 on first roll" and B="3 on second roll" are independent, because $Pr[A \wedge B]=1/36=Pr[A]*Pr[B]=1/6*1/6$.
  - Ex Events A="2 on first roll" and B="the two rolls sum to 5" are not independent, because $Pr[A \wedge B]=1/36 \neq Pr[A]*Pr[B]=1/6*4/36$.

Probability review

- Random variables
  - A variable which takes values with certain probabilities. The probabilities sum to 1.
  - Ex X = value from roll of dice. Values are {1,2,3,4,5,6}, each with probability 1/6.
  - Ex Y = number of heads in 4 flips of fair coin. Values are {0,1,2,3,4}, with probabilities {1/16,4/16,6/16,4/16,1/16}.
  - Ex Z = number of flips of fair coin till first head. Values are {1,2,3,...}, with probabilities {1/2,1/4,1/8,...}.
  - We write Pr[X=x]=y, e.g. Pr[Z=3]=1/8.

Probability review

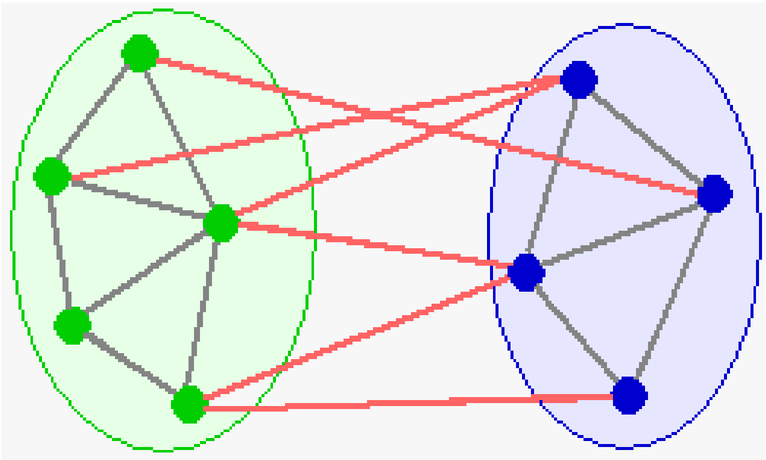
- Expectation of random variable X
  - $E[X]=\sum_x x*Pr[X=x]$.
  - The average value of X, over many trials.
  - Ex X=number of heads in 4 flips. E[X]=0*1/16+1*4/16+2*6/16+3*4/16+4*1/16=2.
    - If you flip a coin 4 times, for 1000 times, on average you see 2 heads per 4 flips.
- An indicator variable X for a event E is a random variable that's 1 of E occurs, and 0 otherwise.
- If event E has probability p of occurring, and X is E's indicator variable, then E[X]=p.
  - Because E[X]=Pr[E occurs]*1+Pr[E doesn't occur]*0=p.
  - This is a convenient fact we'll frequently use.

Probability review

- Linearity of expectations
  - Given random variables X, Y, E[X+Y]=E[X]+E[Y].
  - Extends to any number of random variables, e.g. E[X+Y+Z]=E[X]+E[Y]+E[Z].
  - The random variables do not have to be independent.
  - Very useful property!
  - Ex Let X=number of heads in 100 coin flips. Calculate E[X].
    - Direct method: 1*Pr[1 head]+2*Pr[2 heads]+...+100*Pr[100 heads], a very complicated sum.
    - Linearity method: $X=X_1+X_2+...+X_{100}$, where X_i=number of heads on i'th flip.
    - $E[X_i]=0*Pr[0 \text{ heads}]+1*Pr[1 \text{ head}]=1/2$.
    - $E[X]=E[X_1]+...+E[X_{100}]=100/2=50$.

Problem 1: Max-Cut

- We studied the Min-Cut problem, which is closely related to finding the max flow in a network.
- Max-Cut is the opposite of Min-Cut.
- Given a graph G, split vertices into two sides to maximize the number of edges between the sides.



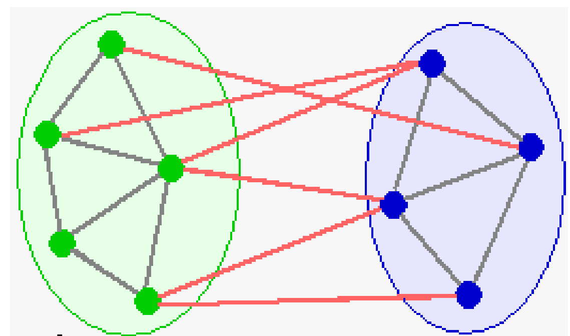
Max-Cut

- Unlike Min-Cut, Max-Cut is NP-complete.
- We'll give a very simple randomized Monte Carlo 2-approximation algorithm.
  - Monte Carlo means the algorithm sometimes returns the wrong answer, i.e. a cut that's not a 2-approximation.
  - Monte Carlo also means the algorithm always runs in a fixed amount of time.

❖ Put each node in a random side with probability ½.

Correctness

- Lemma In a graph with e edges, the algorithm produces a cut with expected size e/2.
- Proof Let X be a random variable equal to the size of the cut. We want to bound E[X].
  - For each edge e, let X_e be the indicator variable of whether e is in the cut.
    - I.e. X_e=1 if e is in the cut and 0 otherwise.
  - So $X = \sum_e X_e$.
  - Given an edge e=(i,j), e is in the cut if i and j are on different sides.
  - So Pr[e in cut]=Pr[(i in L) $\wedge$ (j in R)] + Pr[(j in L) $\wedge$ (i in R)]=1/4+1/4=1/2.
  - So $E[X_e]$=1/2.
  - So E[X]=e/2 by linearity of expectations.

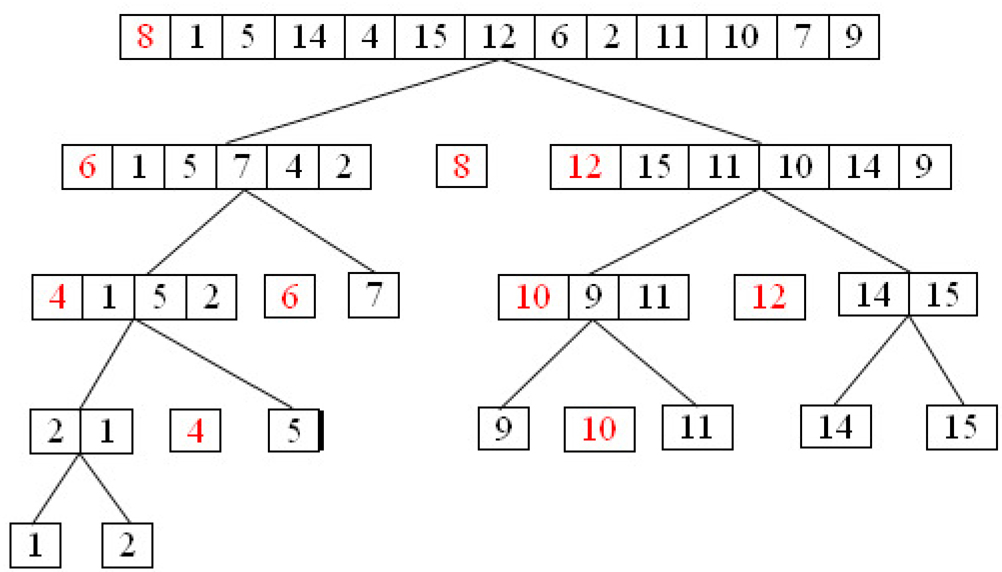


Correctness

- Since a cut can have at most e edges, the e/2 edges the algorithm outputs in expectation is a 2 approximation.
- Note that we only bounded expected size of the algorithm's cut.
  - In any particular execution, the algorithm can output a cut that's smaller or larger than e/2.
    - On average, the cut has size e/2.
  - It's possible to bound the probability the algorithm outputs a cut significantly smaller than e/2, but we won't do this.

Problem 2: Quicksort

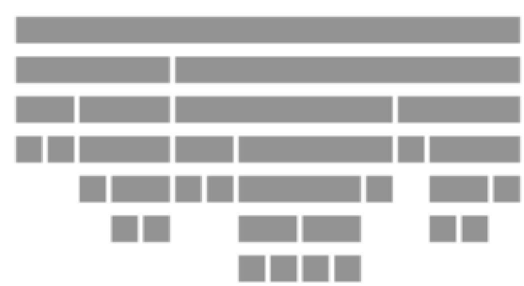
- Recall the Quicksort algorithm.
  - Pick a pivot element s.
  - Partition the elements into two sets, those less than s and those more than s.
  - Recursively Quicksort the two sets.



Complexity of Quicksort

- Let $T(n)$ be the time to Quicksort n numbers.
- $T(n)$ is small in practice.
- But in the worst case, $T(n)=O(n^2)$.
  - Occurs with very uneven splits. I.e. the rank of the pivot is very small or large.
  - Ex If pivot is smallest element, then $T(n)=T(1)+T(n-1)+n-1$. This solves to $T(n)=O(n^2)$.
    - $T(1)$ and $T(n-1)$ to recursively sort each side, n-1 to partition the elements wrt the pivot.

- As long as the pivot is near the middle, Quicksort takes $O(n \log n)$ time.
  - Ex If the pivot is always in the middle half, $[n/4, 3n/4]$, then $T(n) \leq T(n/4)+T(3n/4)+n-1$, which solves to $O(n \log n)$.



Randomized Quicksort

- Quicksort is only slow if we keep picking very small or large pivots.
- Let’s pick the pivot at random. Intuitively, we shouldn’t be unlucky and always pick small or large pivots.

- Pick a random pivot element s.
- Partition the elements into two sets, those less than s and those more than s.
- Recursively RQuicksort the two sets.

Complexity of RQuicksort

- Let $R(n)$ be the expected time to RQuicksort n numbers.
- With probability 1/n, the pivot has rank 1 (is smallest element), in which case $R(n)=R(1)+R(n-1)+n-1$.
- With probability 1/n, the pivot has rank 2, and $R(n)=R(2)+R(n-2)+n-1$.
- ...
- With probability 1/n, the pivot has rank k, and $R(n)=R(k)+R(n-k)+n-1$.
- Putting these together, we have

  $R(n) = 1/n^*(R(1)+R(n-1)+R(2)+R(n-2)+...+R(n-1)+R(1) + n^*1/n^*(n-1) =$

  $2/n^*\Sigma_k R(k) + n-1.$

Complexity of RQuicksort

- We solve the recurrence for R(n) using the substitution method. We guess $R(n) \leq an \log n + b$ for some constants a, b>0 to be determined.
- We first need the following lemma.

Lemma 1.1 $\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2$.

Proof.

$$
\begin{aligned}
\sum_{k=1}^{n-1} k \log k &= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k \\
&\leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\
&\leq \frac{1}{2} n(n-1) \log n - \frac{1}{2}(\frac{n}{2} - 1)\frac{n}{2} \\
&\leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2
\end{aligned}
$$

Complexity of RQuicksort

- Now we can solve for R(n).

$$
\begin{aligned}
R(n) &= \frac{2}{n}\sum_{k=1}^{n-1} R(k) + \Theta(n) \\
&\leq \frac{2}{n}\sum_{k=1}^{n-1}(ak\log k + b) + \Theta(n) \\
&= \frac{2a}{n}\sum_{k=1}^{n-1} k\log k + \frac{2b(n-1)}{n} + \Theta(n) \\
&\leq \frac{2a}{n}(\frac{1}{2}n^2\log n - \frac{1}{8}n^2) + \frac{2b}{n}(n-1) + \Theta(n) \\
&\leq an\log n - \frac{a}{4}n + 2b + \Theta(n) \\
&= an\log n + b + (\Theta(n) + b - \frac{a}{4}n) \\
&\leq an\log n + b
\end{aligned}
$$

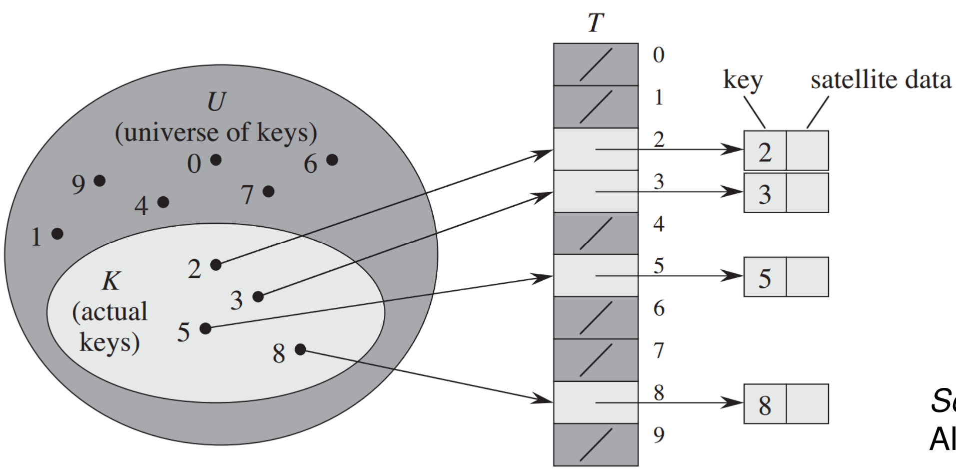
by choosing a so that $\frac{a}{4}n > \Theta(n) + b$.

Hash tables

- A hash table is a randomized data structure to efficiently implement a dictionary.
- Supports find, insert, and delete operations all in expected O(1) time.
  - But in the worst case, all operations are O(n).
  - The worst case is provably very unlikely to occur.
- A hash table does not support efficient min / max or predecessor / successor functions.
  - All these take O(n) time on average.
- A practical, efficient alternative to binary search trees if only find, insert and delete needed.

Direct addressing

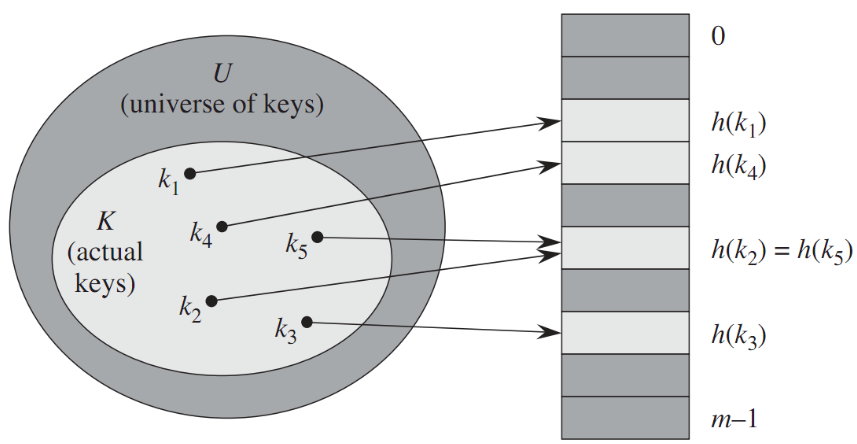
- Suppose we want to store (key, value) pairs, where keys come from a finite universe U = {0, 1, ..., m-1}.
- Use an array of size m.
  - insert(k, v) Store v in array position k.
  - find(k) Return the value in array position k.
  - delete(k) Clear the value in array position k.
- All operations take O(1) time.
- The problem is, if m is large, then we need to use a lot of memory.
  - Uses O(|U|) space.
  - Ex For 32 bit keys, need 16 GB memory. For 64 bit keys, more memory than in world.
  - Ex What about string based keys?
- Also, if only need to store few values, lots of space wasted.



Source: Introduction to Algorithms, Cormen et al

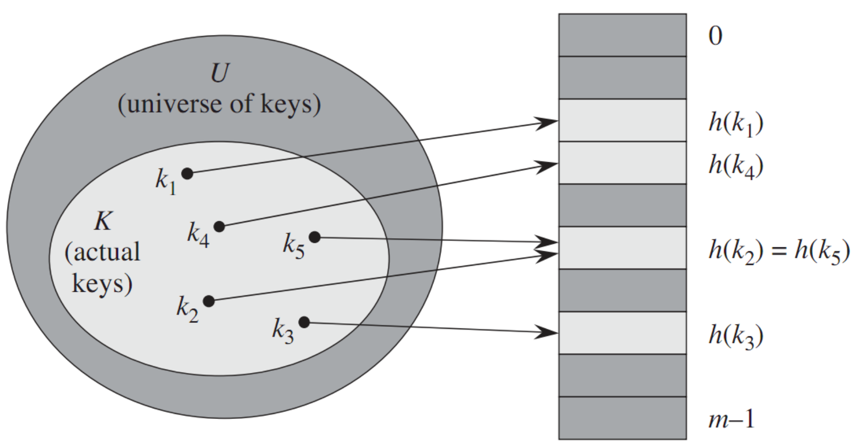
Hash table

- Similar to direct addressing, but uses much less space.
- Idea Instead of storing directly at key's location, convert key to much smaller value, and store at this location.
- A hash table consists of the following.
  - A universe U of keys.
  - An array of T of size m.
  - A hashing function $h:U \rightarrow \{0,1,...,m-1\}$.
- We'll talk later about how to pick good hash functions.
- insert(k, v) Hash key to h(k). Store v in T[h(k)].
- find(k) Return the value in T[h(k)]
- delete(k) Delete the value in T[h(k)]
- Assuming h(k) takes O(1) time to compute, all ops still take O(1) time. Uses O(m) space.
- If $m \ll |U|$, then hashing uses much less space than direct addressing.
- However, our current scheme doesn't quite work, due to collisions.



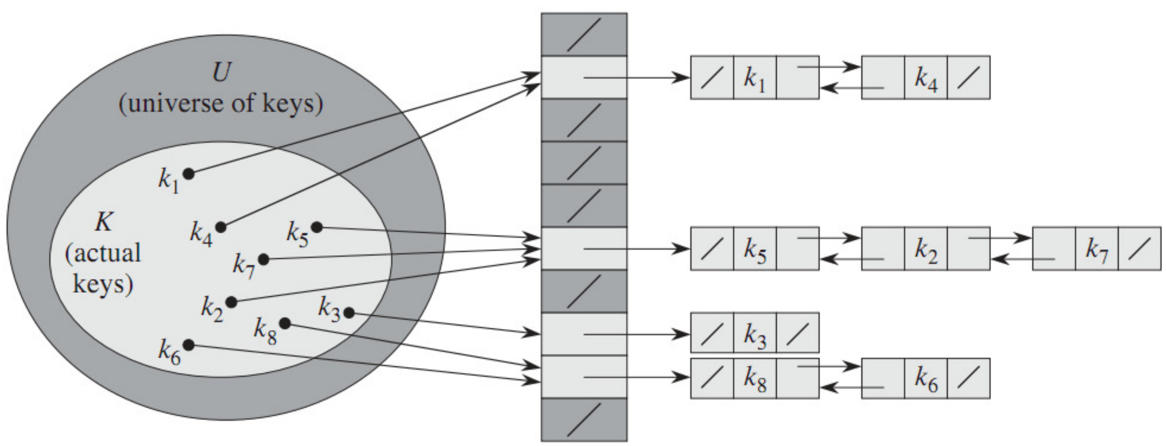
Collisions

- We store a key at array position h(k).
- But what if two keys hash to the same location, i.e. $k_1 \neq k_2$, but $h(k_1) = h(k_2)$?
  - This is called a collision.
- Collisions are unavoidable when $|U| > m$.
  - By Pigeonhole Principle, must exist at least two different keys in U that hash to same value.
- Two basic ways to deal with collisions, closed and open addressing.



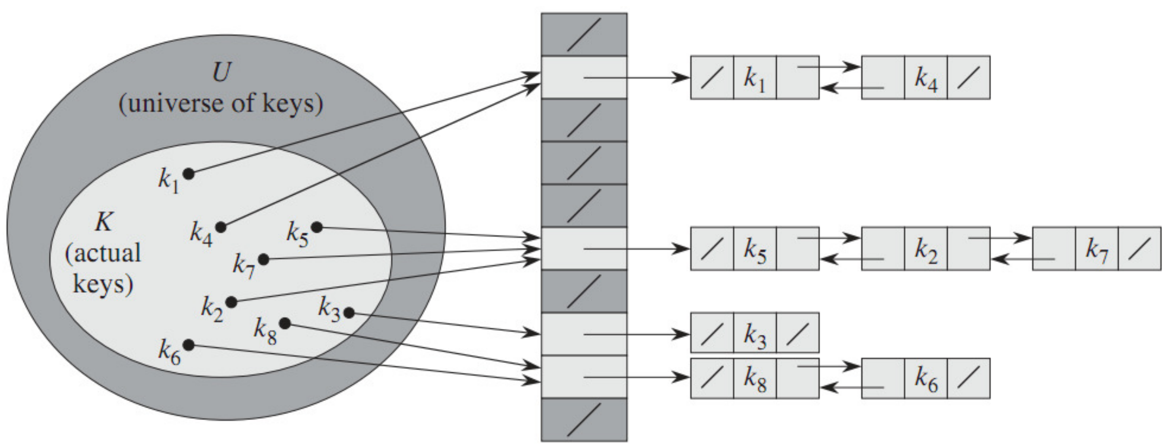
Closed addressing

- In closed addressing, every entry in hash table points to a linked list.
  - Keys that hash to the same location get added to the linked list.
  - For simplicity, we'll ignore values from now on and only focus on keys.
- insert(k) Add k to the linked list in T[h(k)].
- find(k) Search the linked list in T[h(k)] for k.
- delete(k) Delete k from the linked list in T[h(k)].
- Suppose the longest list has length $\hat{n}$, and average length list is $\bar{n}$.
  - Each operation takes worst case $O(\hat{n})$ time.
  - An operation on a random key takes $O(\bar{n})$ time.



Load factor

- The key to making closed addressing hashing fast is to make sure list lengths aren't too long.
- For this, we want the hash function to appear random.
  - Assume that any key is uniformly likely to be hashed to any table location.
- Suppose the hash table contains n items, and has size m.
- Then under the uniform hashing assumption, each table location has on average n/m keys.
  - Call $\alpha = n/m$ the load factor.
- So the average time for each operation is $O(\alpha)$.
- However, even with uniform hashing, in the worst case, all keys can hash to the same location. So the worst case performance is O(n).



Picking a hash function

- We saw that we want hash functions to hash keys to "random" locations.
  - However, note that each hash function is itself a deterministic function, i.e. h(k) always has the same value.
    - If h(k) can produce different values, we can't find key k in the hash table anymore.
- It's hard to find such random hash functions, since we don't assume anything about the distribution of input keys.
  - Ex For any hash function, there are always $\geq |U|/m$ keys from the universe hashing to the same location. So if the input is exactly this set, and $|U|/m \geq n$, then all ops take O(n) time.
- In practice, we use a number of heuristic functions.

Heuristic hash functions

- Assume the keys are natural numbers.
  - Convert other data types to numbers.
  - Ex To convert ASCII string to natural number, treat the string as a radix 128 number. E.g. “pt” → (112*128)+116 = 14452.
- Division method h(k) = k mod m
  - Often choose m a prime number not too close to a power of 2.
- Multiplication method $h(k) = \lfloor m\ (k\ A \bmod 1) \rfloor$, where A is some constant.
  - Knuth’s suggestion is $A = \frac{\sqrt{5}-1}{2} \approx 0.618034 \ldots$

Universal hashing

- As we said, regardless of the hash function, an adversary can choose a set of n inputs to make all operations O(n) time.
- Universal hashing overcomes this using randomization.
  - No matter what the n input keys are, every operation takes O(n/m) time in expectation, for a size m hash table.
  - Note O(n/m) time is optimal.
- Instead of using a fixed hash function, universal hashing uses a random hash function, chosen from some set of functions H.
- Say H is a universal hash family if for any keys $x \neq y$

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

- So if we randomly choose a hash function from H and use it to hash any keys x, y, they have 1/m probability of colliding.
- Note the hash functions in H are not random. However, we choose which function to use from H randomly.

Universal hashing

- Thm Let H be a universal hash family. Let S be a set of n keys, and let $x \in S$. If $h \in H$ is chosen at random, then the expected number of $y \in S$ s.t. $h(x) = h(y)$ is n/m.
- Proof Say $S = \{x_1, \ldots, x_n\}$.
  - Let X be a random variable equal to the number of $y \in S$ s.t. $h(x) = h(y)$.
  - Let $X_i = 1$ if $h(x_i) = h(x)$ and 0 otherwise.
  - $E[X_i] = \Pr_{h \in H}[h(x_i) = h(x)] \times 1 + \Pr_{h \in H}[h(x_i) \neq h(x)] \times 0 = 1/m$.
    - First equality follows by universal hashing property.
  - $E[X] = E[X_1] + \cdots + E[X_n] = n/m$.

Constructing universal hash family 1

- Choose a prime number p such that p > m, and p > all keys.
- Let $h_{ab}(k) = \big((ak + b) \bmod p\big) \bmod m$.
- Let $H_{pm} = \{h_{ab} \mid a \in \{1,2,\dots,p-1\}, b \in \{0,1,\dots,p-1\}\}$.
- Thm H_{pm} is a universal hash family.
- Proof Let $x, y < p$ be two different keys. For a given h_{ab} let
  $$r = (ax + b) \bmod p, \qquad s = (ay + b) \bmod p$$
- We have $r \neq s$, because $r - s \equiv a(x - y) \bmod p \neq 0$, since neither a nor $x - y$ divide p.
- Also, each pair (a, b) leads to a different pair (r, s), since
  $$a = \big((r - s)(x - y)^{-1} \bmod p\big), \qquad b = (r - ax) \bmod p$$
  - Here, $(x - y)^{-1} \bmod p$ is the unique multiplicative inverse of $x - y$ in $\mathbb{Z}_p^*$.

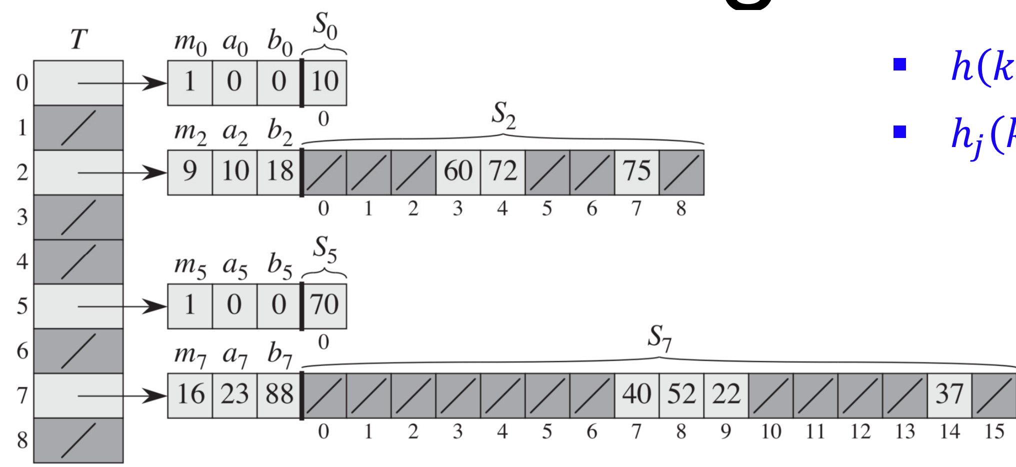
Constructing universal hash family 2

- Since there are $p(p-1)$ pairs (a,b) and $p(p-1)$ pairs (r,s) with $r \neq s$, then a random (a,b) produces a random (r,s).
- The probability x and y collide equals the probability $r \equiv s \bmod m$.
- For fixed r, number of $s \neq r$ s.t. $r \equiv s \bmod m$ is $(p-1)/m$.
- So for each r and random $s \neq r$, probability that $r \equiv s \bmod m$ is $((p-1)/m))/(p-1) = 1/m$.
- So $\Pr_{h_{ab} \in H_{pm}}[h_{ab}(x) = h_{ab}(y)] = 1/m$ and H_{pm} is universal.

Perfect hashing

- The hashing methods we've seen can ensure O(1) expected performance, but are O(n) in the worst case due to collisions.
- However, if we have a fixed set of keys, perfect hashing can ensure no collisions at all.
  - Perfect hashing maintains a static set, and allows find(k) and delete(k) in O(1) time.
  - It doesn't support insert(k).
- Ex The fixed set of keys may represent the file names on a non-writable DVD.

Perfect hashing



- $h(k) = \big((3k + 42) \bmod 101\big) \bmod 9$
- $h_j(k) = \Big(\big(a_j k + b_j\big) \bmod 101\Big) \bmod m_j$

- Suppose we want to store n items with no collisions.
- Perfect hashing uses two levels of universal hashing.
  - The first layer hash table has size m = n.
  - Use first layer hash function h to hash key to a location in T.
  - Each location j in T points to a hash table S_j with hash function h_j.
  - If n_j keys hash to location j, the size of S_j is $m_j = n_j^2$.
- We'll ensure there are no collisions in the secondary hash tables $S_1, \ldots, S_m$.
  - So all operations take worst case O(1) time.
- Overall the space use is $O(m + \sum_{j=1}^{m} n_j^2)$.
  - We'll show this is $O(n) = O(m)$.
  - So perfect hashing uses same amount of space as normal hashing.

Avoiding collisions

- Lemma Suppose we store n keys in a hash table of size m = n^2 using universal hashing. Then with probability ≥ 1/2 there are no collision.
- Proof There are $\binom{n}{2}$ pairs of keys that can collide.
  - Each collision occurs with probability 1/m = $1/n^2$, by universal hashing.
  - So the expected number of collisions is $\frac{\binom{n}{2}}{n^2} \leq \frac{1}{2}$.
  - By Markov's inequality the Pr[# collisions ≥ 1] $\leq$ E[# collisions] $\leq 1/2$.
- When building each hash table S_j, there's < 1/2 probability of having any collisions.
  - If collisions occur, pick another random hash function from the universal family and try again.
  - In expectation, we try twice before finding a hash function causing no collisions.

Space complexity

- Lemma Suppose we store n keys in a hash table of size m=n. Then the secondary hash tables use space $E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq 2n$, where n_j is the number of keys hashing to location j.
- Proof $E\left[\sum_{j=0}^{m-1} n_j^2\right] = E[\sum_{j=0}^{m-1}(n_j + 2\binom{n_j}{2})] =$ $E[\sum_{j=0}^{m-1} n_j] + 2\,E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right]$
- $\sum_{j=0}^{m-1}\binom{n_j}{2}$ is the total number of pairs of hash keys which collide in the first level hash table.
  - By universal hashing, this equals $\binom{n}{2}\frac{1}{m} = \frac{n-1}{2}$.
- $E[\sum_{j=0}^{m-1} n_j] = n$.
- So $E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq n + \frac{2(n-1)}{2} < 2n$.