上海科技大学
ShanghaiTech University

CS240 Algorithm Design and Analysis

Lecture 21

Randomized algorithms (Cont.)

Fall 2021

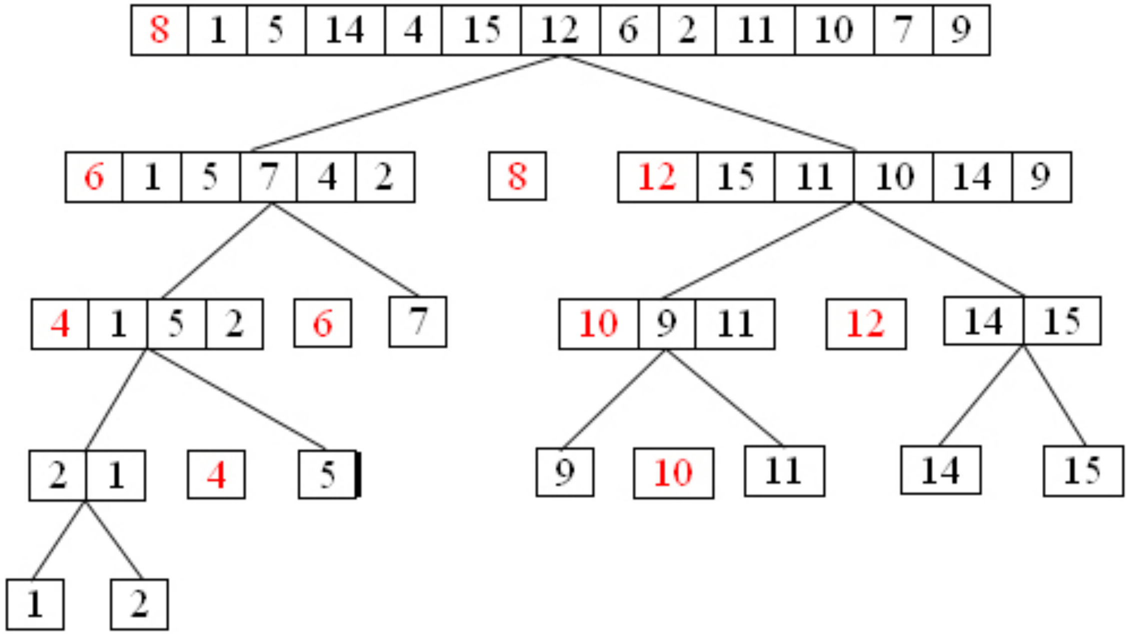
2021.12.01

Quicksort

Quicksort

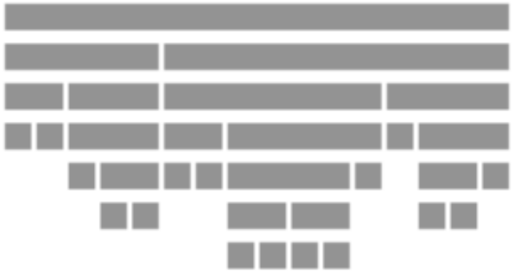
Recall the Quicksort algorithm

- 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 $O(n \log n)$.



Pivot selection is crucial

Running time.

- [Best case.] Select the median element as the pivot: quicksort runs in $\Theta(n \log n)$ time.
- [Worst case.] Select the smallest (or the largest) element as the pivot: quicksort runs in $\Theta(n^2)$ time.

Q: How to find the median element?

A: Sort?

A: Randomly choose an element as the pivot!

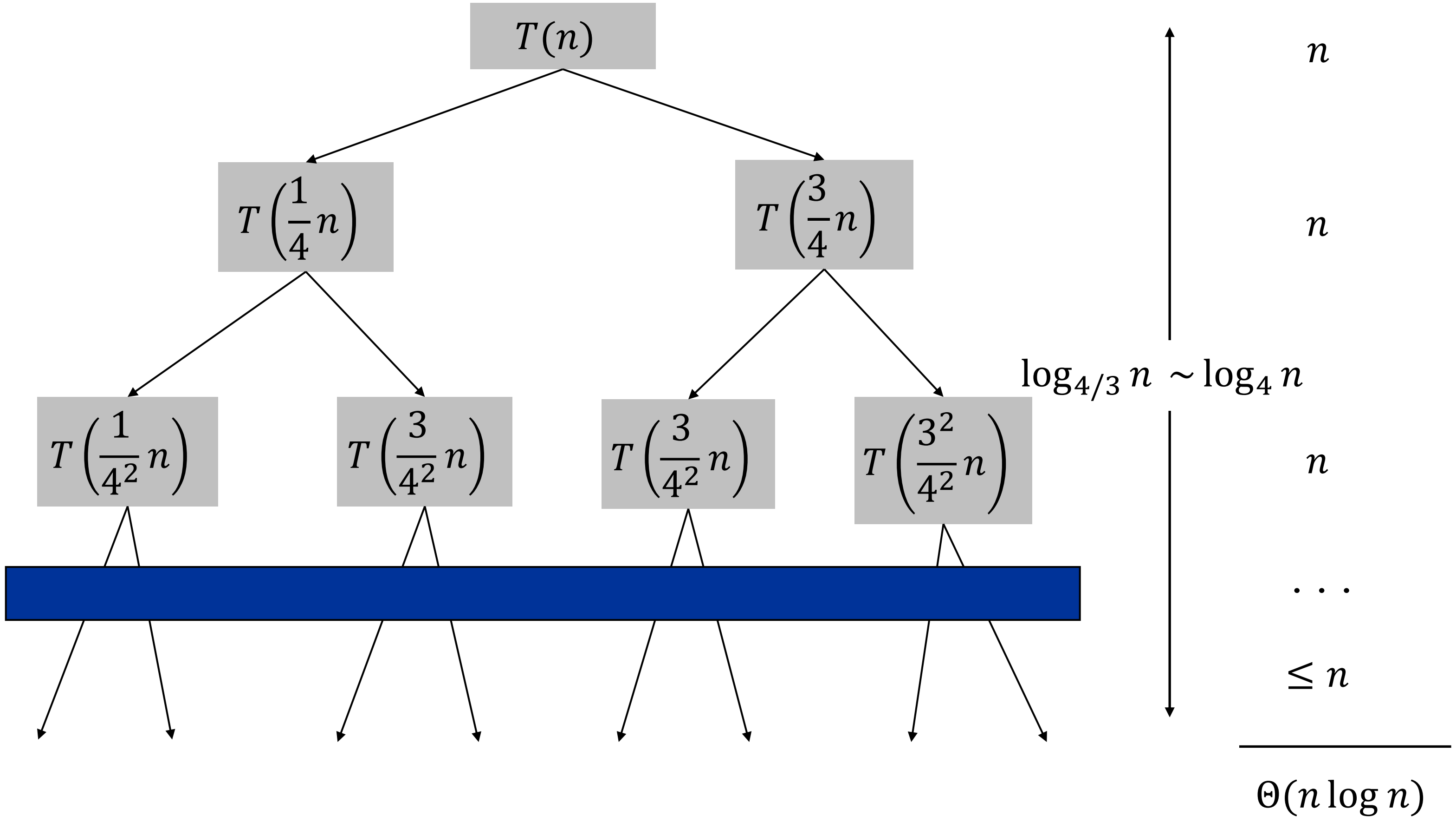
Intuition: A randomly selected pivot "typically" partitions the array as 25% vs 75%, so we have the recurrence

$$T(n) = T\left(\frac{1}{4}n\right) + T\left(\frac{3}{4}n\right) + n$$

which solves to $T(n) = \Theta(n \log n)$. (See next page.)

Solve the recurrence

$$T(n) = T\left(\frac{1}{4}n\right) + T\left(\frac{3}{4}n\right) + n$$



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.

1. 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

$$R(n) = \frac{1}{n} * \left(R(1) + R(n-1) + R(2) + R(n-2) + \cdots + R(n-1) + R(1) + (n-1) * \ (n-1)\right)$$

$$\leq \frac{2}{n} \sum_{k=1}^{n-1} R(k) + \Theta(n)$$

1. Complexity of RQuicksort

上海科技大学 ShanghaiTech University

- 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.

$$\sum_{k=1}^{n-1} klogk \leq \frac{1}{2}n^2 logn - \frac{1}{8}n^2$$

- Proof:

$$\sum_{k=1}^{n-1} klogk = \sum_{k=1}^{\left\lceil\frac{n}{2}\right\rceil-1} klogk + \sum_{k=\lceil n/2\rceil}^{n-1} klogk$$

$$\leq (logn - 1)\sum_{k=1}^{\left\lceil\frac{n}{2}\right\rceil-1} k + logn \sum_{k=\left\lceil\frac{n}{2}\right\rceil}^{n-1} k$$

$$= logn \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2\rceil-1} k$$

$$\leq \frac{1}{2}n(n-1)logn - \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{n}{2}$$

$$\leq \frac{1}{2}n^2 logn - \frac{1}{8}n^2$$

立志成才报国裕民

1. Complexity of RQuicksort

上海科技大学 ShanghaiTech University

- Now we can solve for R(n).

$$R(n) = \frac{2}{n}\sum_{k=1}^{n-1} R(k) + \Theta(n)$$

$$\leq \frac{2}{n}\sum_{k=1}^{n-1}(aklogk + b) + \Theta(n)$$

$$= \frac{2a}{n}\sum_{k=1}^{n-1} klogk + \frac{2b(n-1)}{n} + \Theta(n)$$

$$\leq \frac{2a}{n}\left(\frac{1}{2}n^2logn - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n)$$

$$\leq anlogn - \frac{a}{4}n + 2b + \Theta(n)$$

$$= anlogn + b + \left(\Theta(n) + b - \frac{a}{4}n\right)$$

$$\leq anlogn + b$$

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

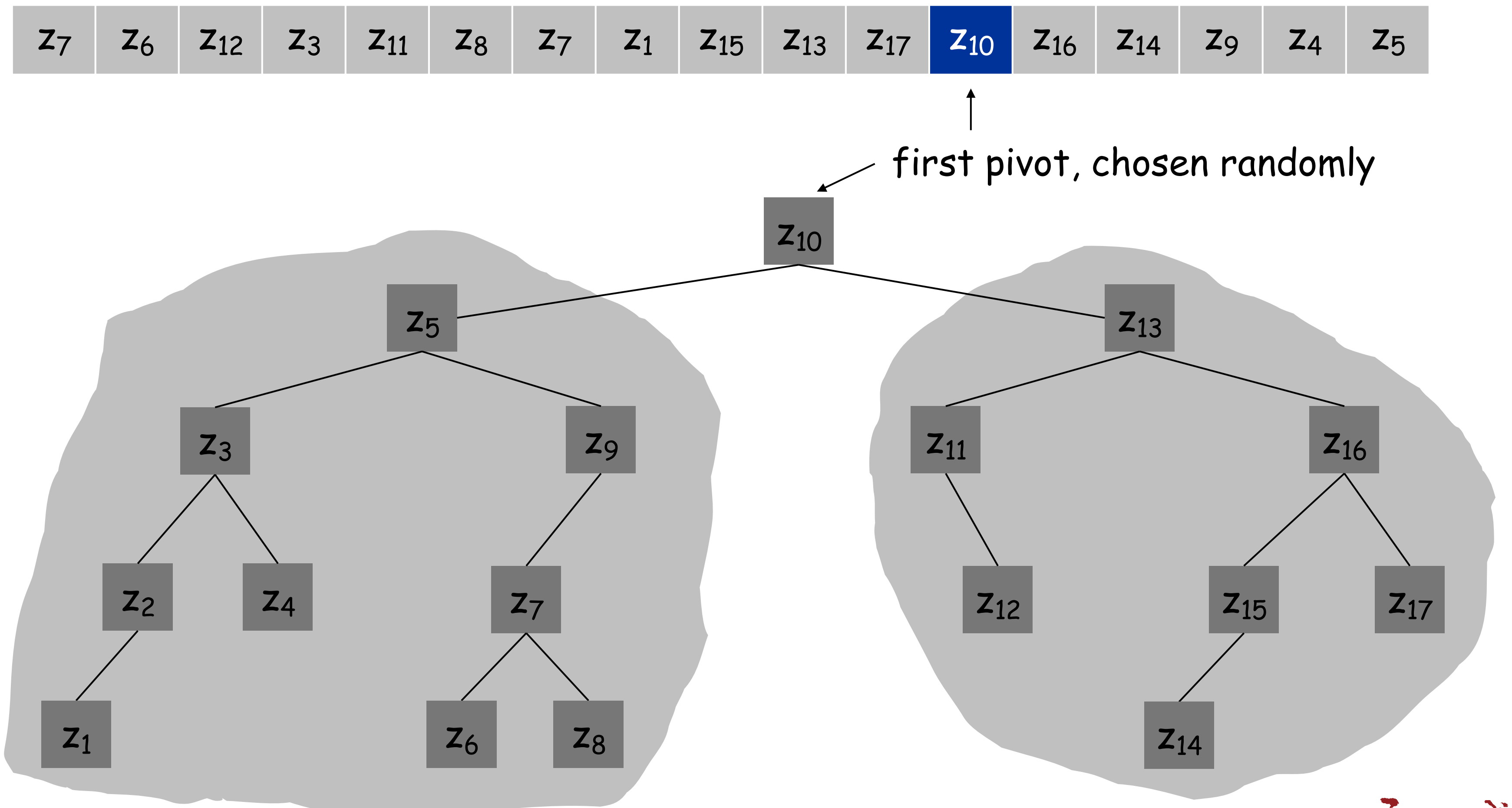
立志成才报国裕民

2. Analysis of quicksort: the binary tree representation

Assumption: All elements are distinct

Note: Running time = Θ(# comparisons)

Relabel the elements from small to large as $z_1, z_2, \ldots, z_n$



2. Analysis of quicksort (Cont.)

Theorem. Expected # of comparisons is $\Theta(n \log n)$.

Pf.

- Let $X_{ij} = 1$ if z_i is compared with z_j
- # of comparisons is $X = \sum_{i<j} X_{ij}$
- E[# of comparisons] $= \sum_{i<j} E[X_{ij}] = \sum_{i<j} \Pr[z_i \text{ and } z_j \text{ are compared}]$

	$j = 2$	3	4	…	n	
$i = 1$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	…	$\frac{1}{n}$	$O(\log n)$
2		$\frac{1}{2}$	$\frac{1}{3}$	…	$\frac{1}{n-1}$	$O(\log n)$
3			$\frac{1}{2}$	…	$\frac{1}{n-2}$	$O(\log n)$
…	…	…	…	…	…	…
$n-1$					$\frac{1}{2}$	$O(\log n)$
						$O(n \log n)$

Q: Can you show this is $\Theta(n \log n)$?

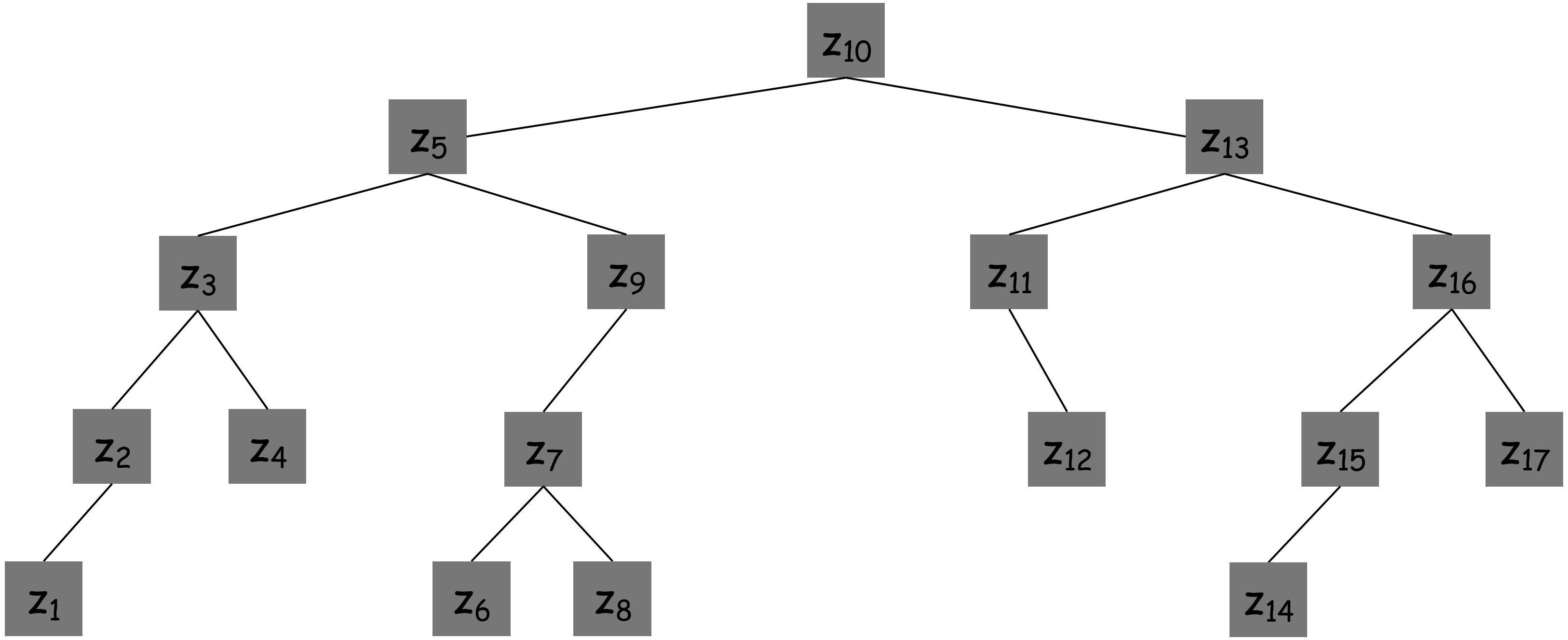
2. Analysis of quicksort

Observation 1: Element only compared with its ancestors and descendants.

- z_2 and z_7 are compared if their lowest common ancestor (lca) is z_2 or z_7.
- z_2 and z_7 are not compared if their lca is z_3, z_4, z_5, or z_6.
- Other elements cannot be the lca of z_2 and z_7

Observation 2: Every element in $\{z_i, \ldots, z_j\}$ is equally likely to be the lca of z_i and z_j

So, $\Pr[z_i \text{ and } z_j \text{ are compared}] = 2 / (j - i + 1)$.



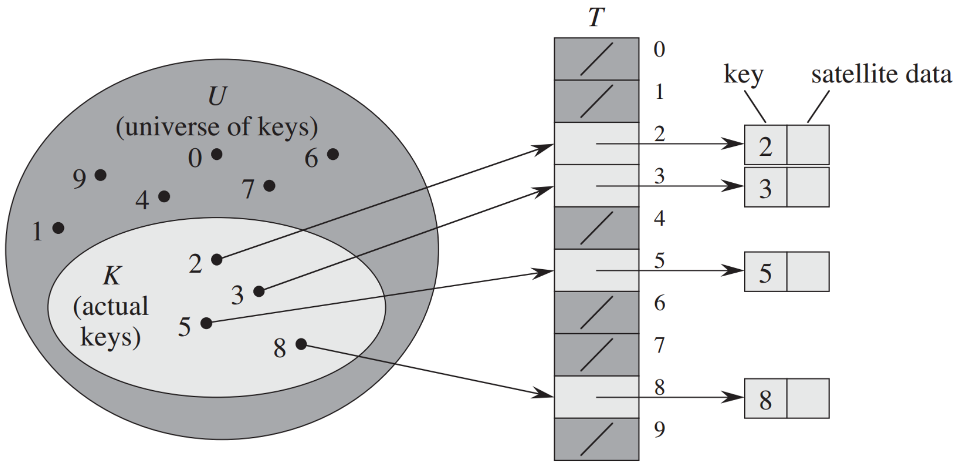
Hash Tables

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

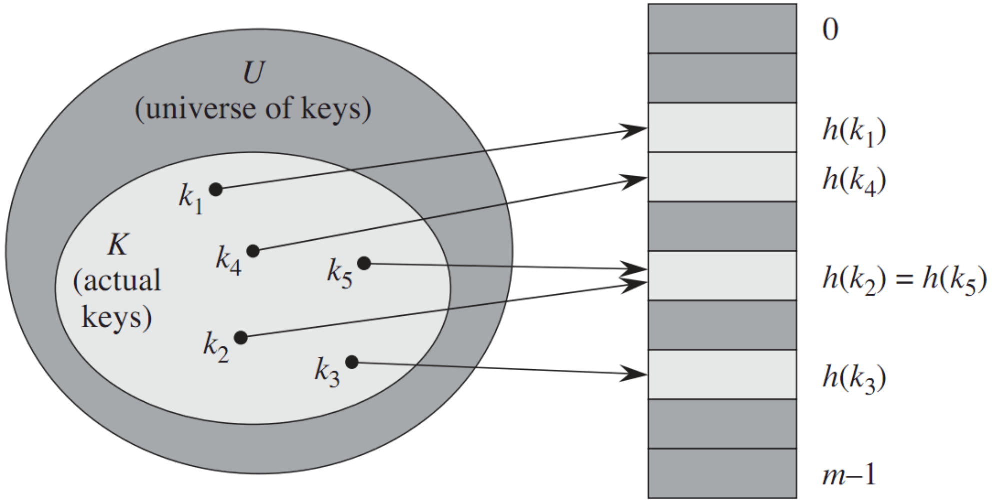
- Suppose we want to store (key, value) pairs, where keys come from a finite universe U = {0, 1, ..., m-1}.
- Use an array of size m.
  - insert(k, v) Store v in array position k.
  - find(k) Return the value in array position k.
  - delete(k) Clear the value in array position k.
- All operations take 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 4 GB memory. For 64 bit keys, more memory than in world.
- If only need to store few values, lots of space wasted.



Source: Introduction to Algorithms, Cormen et al

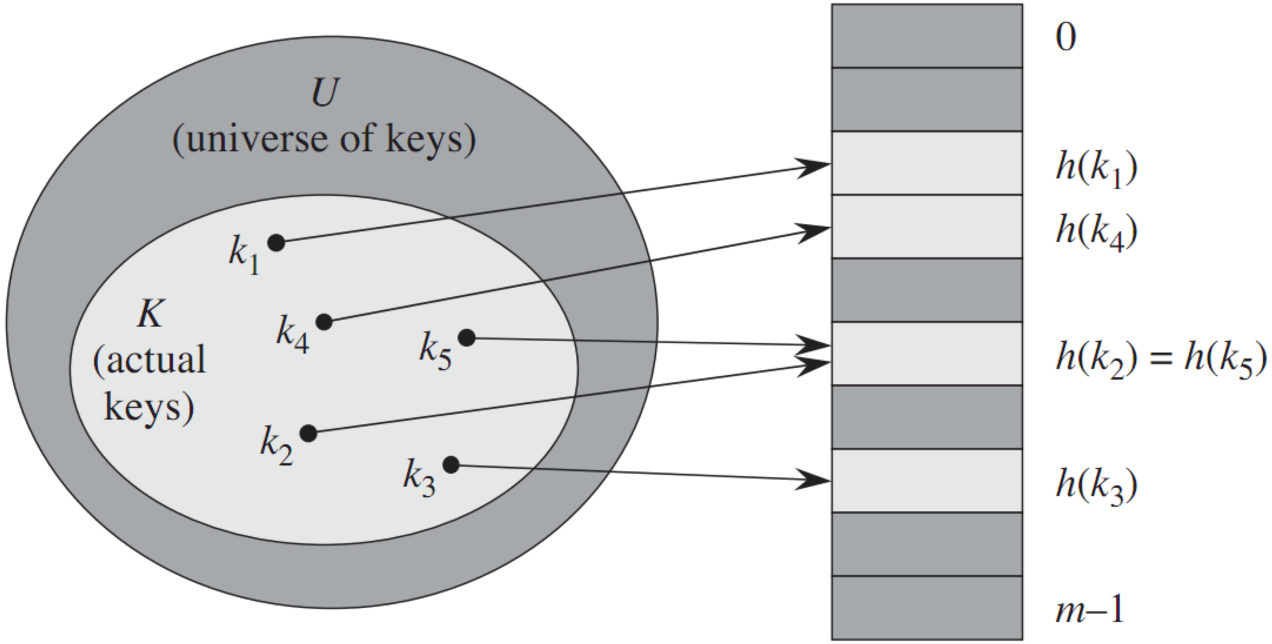
Hash Table

- 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→{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.



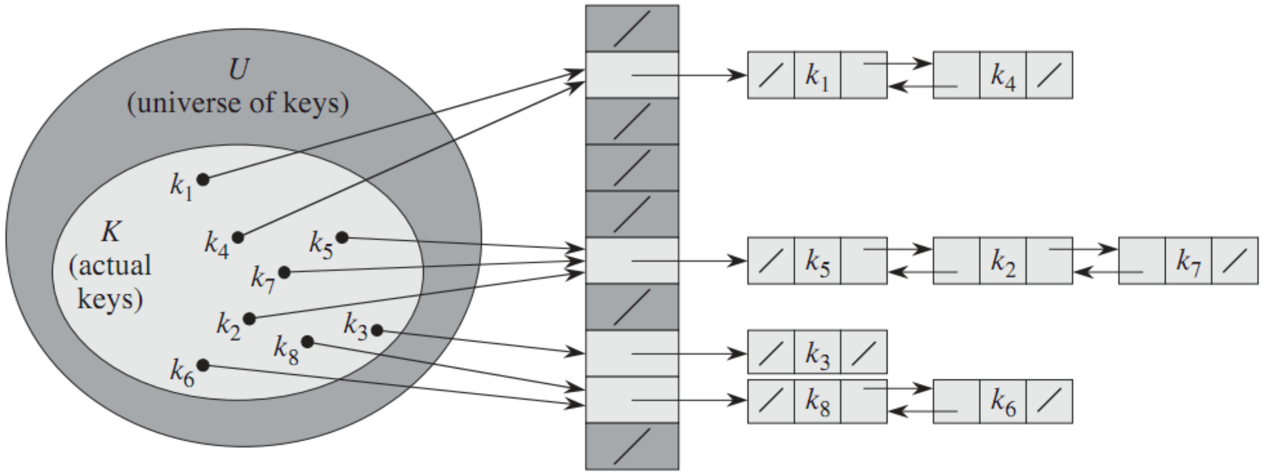
Collisions

- 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.



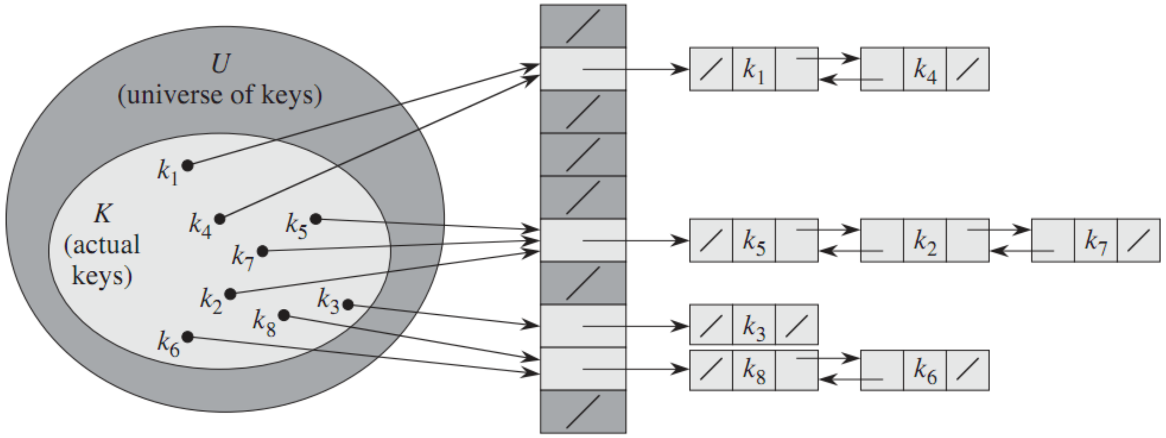
Closed Addressing

- 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.
- 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 $H_{pm} = \{h_{ab} \mid a \in \{1,2,\ldots,p-1\}, b \in \{0,1,\ldots,p-1\}\}$.
- 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.

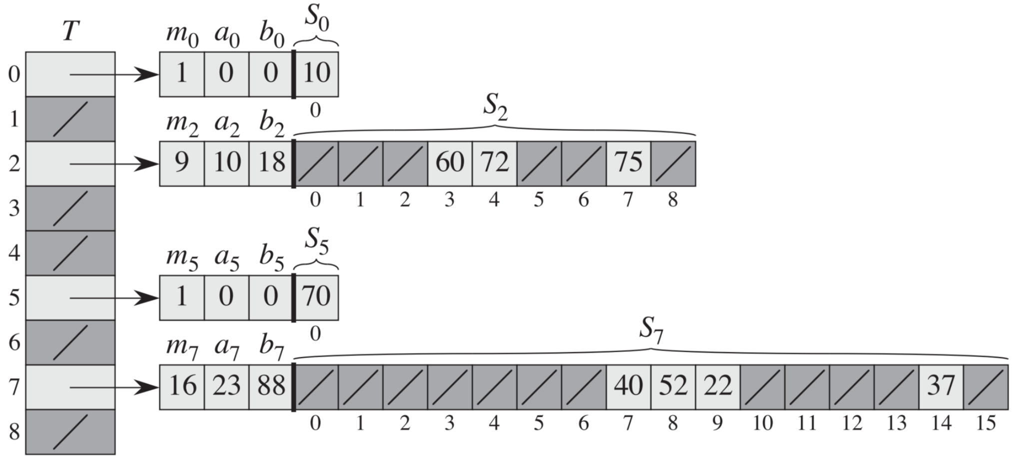
Perfect hashing

- 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

- 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.

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



Avoiding collisions

- Lemma Suppose we store n keys in a hash table of size $m = n^2$ using universal hashing. Then with probability $\geq 1/2$ 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 $E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq n + \frac{2(n-1)}{2} < 2n$.

Next Time:
Randomized algorithms (Cont.)