Advance Algorithm Analysis

(COMP 502 A)

Sharoon Nasim

sharoonnasim@fccollege.edu.pk

Office: S-426 D

Randomized Algorithms

Randomized algorithms can be broadly classified into two types- Monte Carlo and Las Vegas.

Monte Carlo	Las Vegas
runs in polynomial time always	runs in expected polynomial time
output is correct with high probability	output always correct

Monte Carlo and Las Vegas are two types of probabilistic algorithms

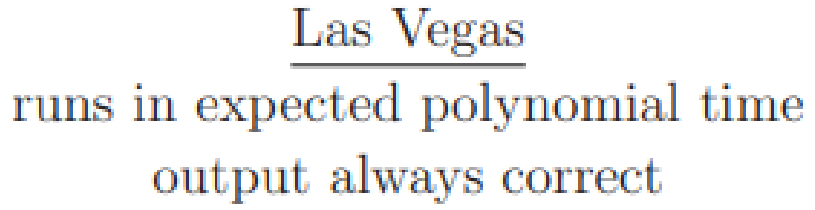
- **Definition:** Monte Carlo algorithms use random sampling to obtain numerical results. They provide an approximate solution to a problem with a certain level of confidence.
- **Definition:** Las Vegas algorithms use randomization to ensure that the algorithm always produces the correct result. The randomness is used to optimize the algorithm's running time rather than the correctness of the output.
  Quicksort algorithm.

Quicksort

Divide and conquer algorithm but have more work in the divide step rather than combine.

Different variants:

- Basic: good in average case
- Median-based pivoting: uses median finding
- Random: good for all inputs in expectation (Las Vegas algorithm)



Las Vegas

runs in expected polynomial time

output always correct

QuickSort

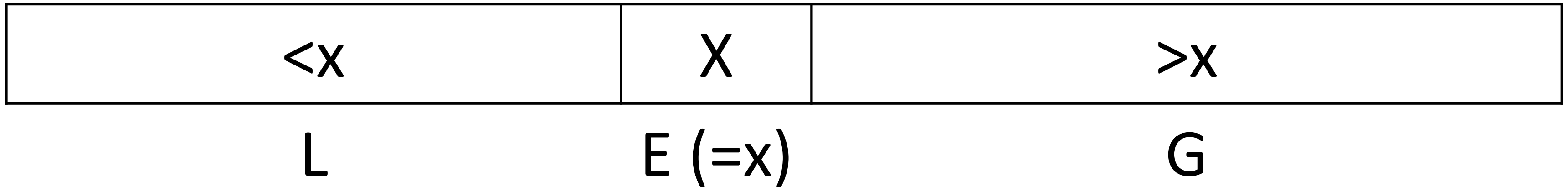
Steps of quicksort:

- Divide: pick a pivot element x in A, partition the array into sub-arrays L, consisting of all elements < x,

G consisting of all elements > x and

E consisting of all elements = x.

- Conquer: recursively sort subarrays L and G
- Combine: trivia



Basic Quicksort

1. Pivot around x = A[1] or A[n] (first or last element)
2. Remove, in turn, each element y from A
3. Insert y into L, E or G depending on the comparison with pivot x
4. Each insertion and removal takes O(1) time
5. Partition step takes O(n) time

Basic Quicksort Analysis If input is sorted or reverse sorted, we are partitioning around the min or max element each time. This means one of L or G has n−1 elements, and the other 0. This gives:

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

However, this algorithm does well on random inputs in practice

Pivot Selection Using Median Finding

Can guarantee balanced L and G using rank/median selection algorithm that runs in Θ(n) time. The first Θ(n) below is for the pivot selection and the second for the partition step.

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(n)$$
$$T(n) = \Theta(n \log n)$$

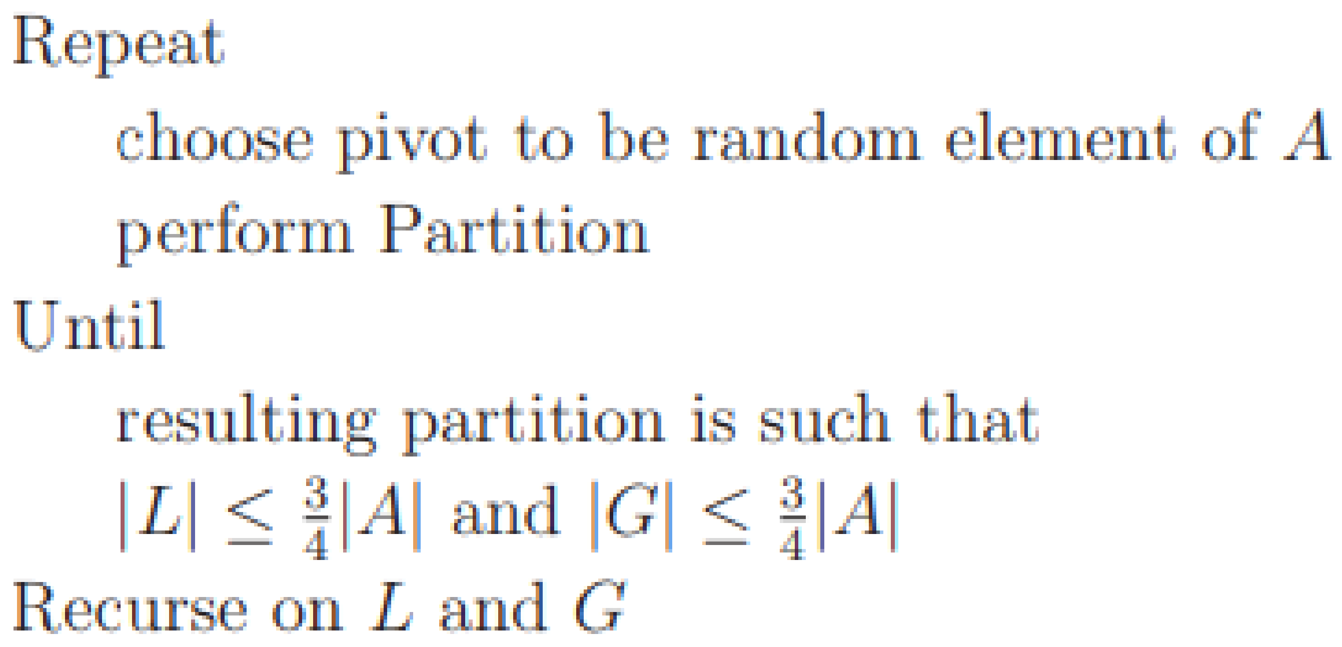
This algorithm is slow in practice and loses to mergesort.

Randomized Quicksort

x is chosen at random from array A (at each recursion, a random choice is made). Expected time is O(n log n) for all input arrays A.

“Paranoid” Quicksort

Variant of Random Quicksort



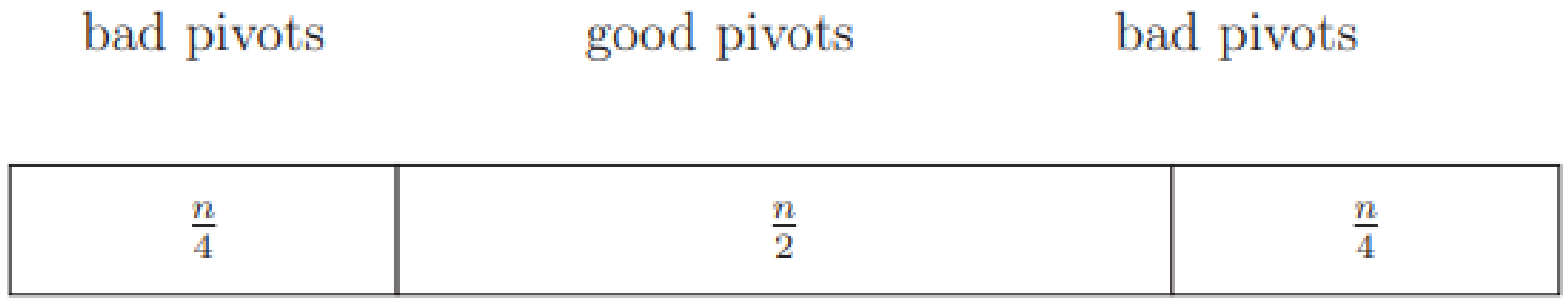
```
Repeat
    choose pivot to be random element of $A$
    perform Partition
Until
    resulting partition is such that
    $|L| \leq \frac{3}{4}|A|$ and $|G| \leq \frac{3}{4}|A|$
Recurse on $L$ and $G$
```

“Paranoid” Quicksort Analysis

Let’s define a ”good pivot” and a ”bad pivot”-

Good pivot: sizes of L and $G \leq \frac{3}{4}n$ each

Bad pivot: one of L and G is $\leq \frac{3}{4}n$ each



We see that a pivot is good with probability > 1/2

$$T(n) \leq max_{n/4 \leq i \leq 3n/4}(T(i) + T(n-i)) + E(\#\text{iterations}) \cdot cn$$

Now, since probability of good pivot $> \frac{1}{2}$,

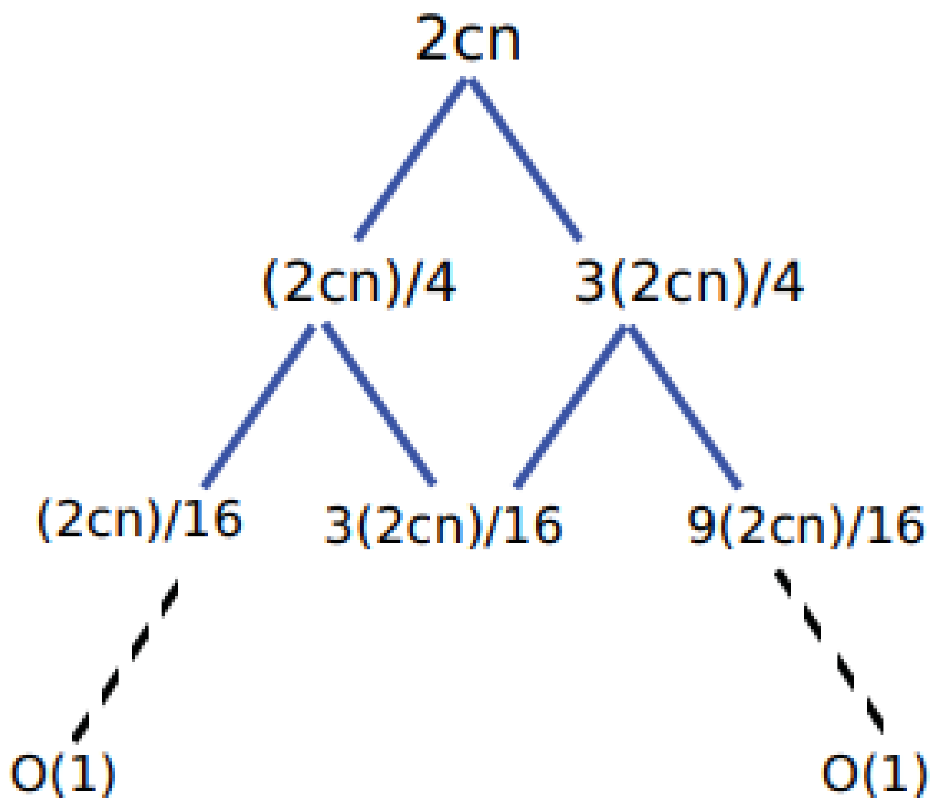
$$E(\#iterations) \leq 2$$

Let $T(n)$ be an upper bound on the expected running time on any array of n size. T(n) comprises:

- time needed to sort left subarray $T(n/4)$
- time needed to sort right subarray $T(3n/4)$
- the number of iterations to get a good call. Denote as $c \cdot n$ the cost of the partition step

$2 \cdot cn$

We see in the figure that the height of the tree can be at most $\log_{\frac{4}{3}}(2cn)$ no matter what branch we follow to the bottom. At each level, we do a total of 2cn work. Thus, expected runtime is T(n) = Θ(n log n)



$$T(n) \leq T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + 2cn$$

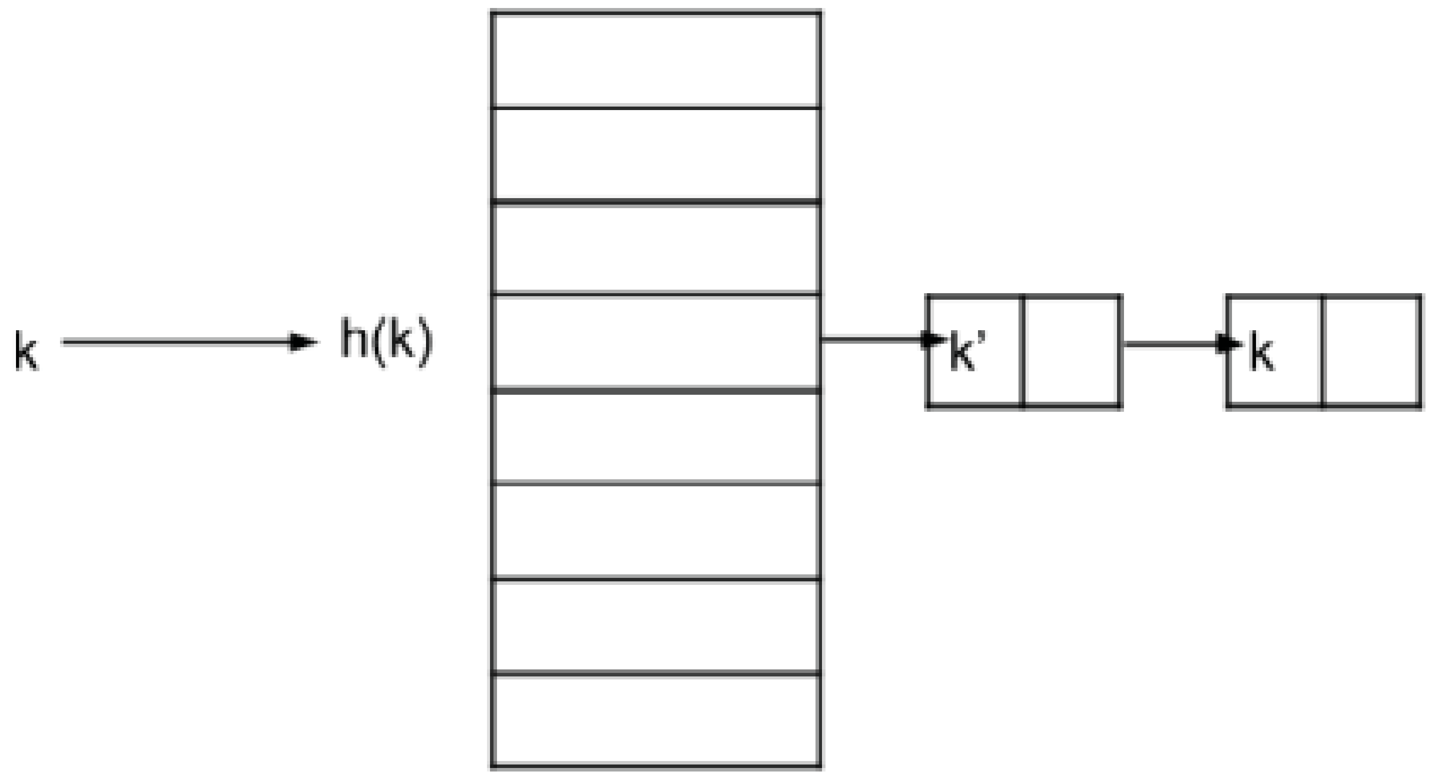
Randomized Hashing

Simple Hashing

Goal: O(1) time per operation and O(n) space complexity.

Definitions:

- u = number of keys over all possible items
- n = number of keys/items currently in the table
- m = number of slots in the table

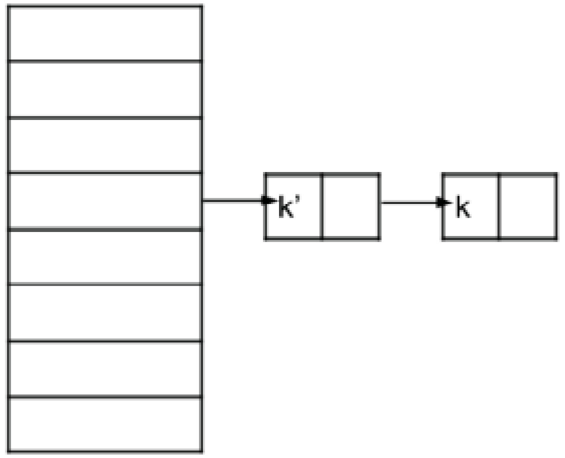


Solution: hashing with chaining

Assuming simple uniform hashing,

$$\Pr_{k_1 \neq k_2}\{h(k_1) = h(k_2)\} = \frac{1}{m}$$

we achieve Θ(1 + α) time per operation, where α = n/m is called load factor.
The m downside of the algorithm is that it requires assuming input keys are random, and it only works in average case, like basic quicksort.
Today we are going to remove the unreasonable simple uniform hashing assumption.



Randomized Hashing

- Universal Hashing
    - Random hash function
- Perfect Hashing
    - 2-level hashing

Universal Hashing

The idea of universal hashing is listed as following:

- choose a random hash function h from $\mathcal{H}$
- require $\mathcal{H}$ to be a *universal hashing family* such that

$$\Pr_{h \in \mathcal{H}} \{h(k) = h(k')\} \leq \frac{1}{m} \quad \text{for all} \quad k \neq k'$$

- now we just assume h is random, and make no assumption about input keys. (like Randomized Quicksort)

Universal hashing with a random hash function is a technique used to design hash functions that minimize the chance of collisions and provide good average-case performance.

The randomization helps prevent an adversary from predicting or deliberately choosing inputs that would cause a high number of collisions.

Homework

Theorem: For n arbitrary distinct keys and random $h \in \mathcal{H}$, where $\mathcal{H}$ is a universal hashing family,

$$E[\text{ number of keys colliding in a slot }] \leq 1 + \alpha \quad \text{where} \quad \alpha = \frac{n}{m}$$

Perfect Hashing

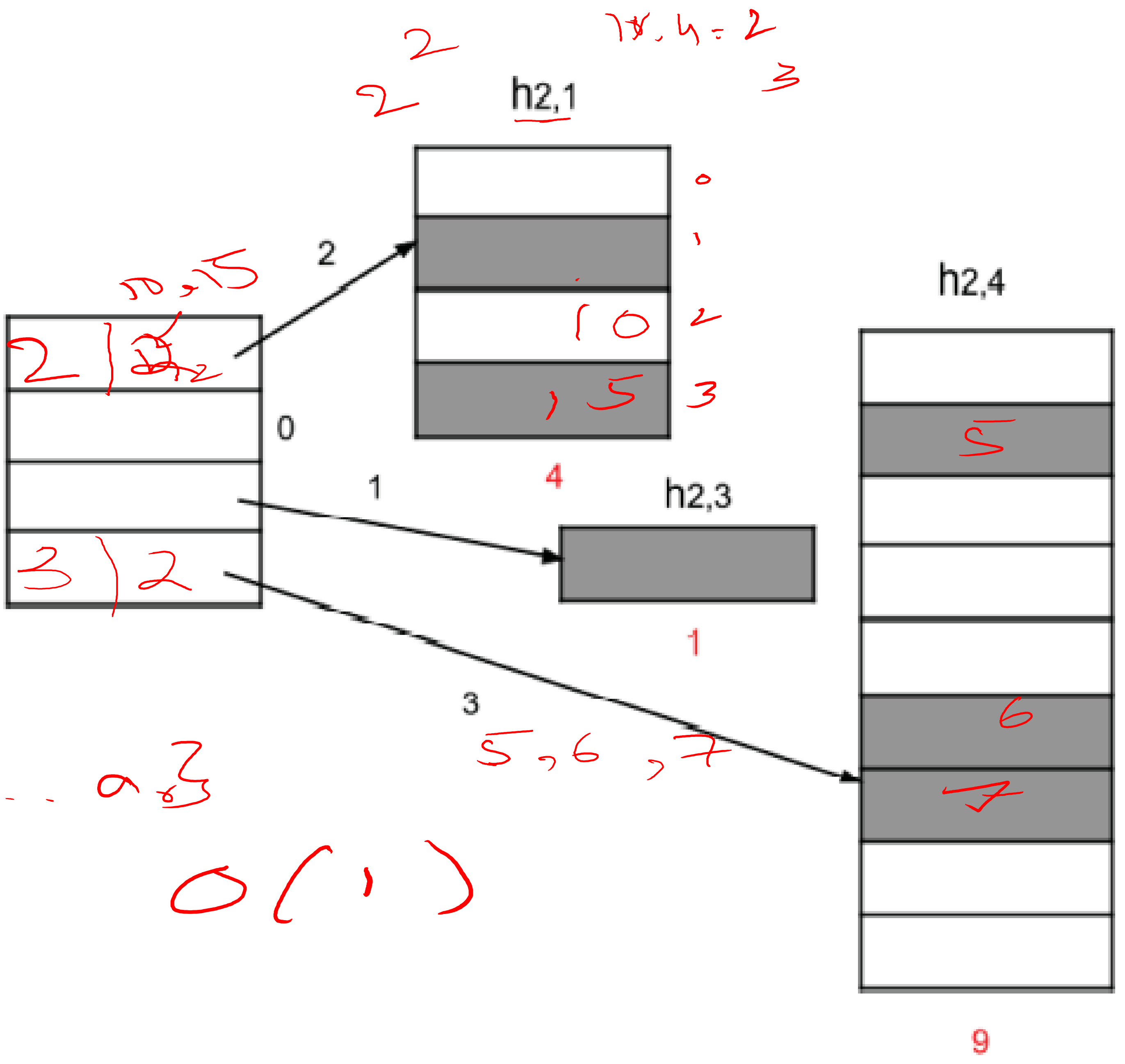
Static dictionary problem: Given n keys to store in table, only need to support search(k). No insertion or deletion will happen.

Perfect hashing: [Fredman, Komlós, Szemerédi 1984]

- polynomial build time with high probability (w.h.p.)
- $O(1)$ time for search in worst case
- $O(n)$ space in worst case

Idea: 2-level hashing

Perfect Hashing



$h_i = \{ \ldots \}$

$\{a_0, a_1, \ldots a_6\}$

$O(1)$

The algorithm contains the following two major steps:

Step 1: Pick $h_1 : \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m-1\}$ from a universal hash family for $m = \Theta(n)$ (e.g., nearby prime). Hash all items with chaining using h_1.

Step 2: For each slot $j \in \{0, 1, \ldots, m-1\}$, let l_j be the number of items in slot j. $l_j = |\{i \mid h(k_i) = j\}|$. Pick $h_{2,j} : \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m_j\}$ from a universal hash family for $l_j^2 \leq m_j \leq O(l_j^2)$ (e.g., nearby prime). Replace chain in slot j with hashing-with-chaining using $h_{2,j}$.