

Hash Tables & Linear Probing

Last Tuesday

Probability

Events, probabilities,
conditional probabilities,
union bound

Random variables,
expected value,
linearity of expectation
Markov's inequality

Algorithms

Approx. 3-SAT,

Big idea: we only want the average

$$\begin{array}{l} \text{Linearity of Expectation} \\ E[X+Y] = E[X] + E[Y] \quad (\text{independently useful}) \\ \text{"average sum = sum of averages"} \end{array}$$

let $Y_i = \begin{cases} 1 & \text{if } i\text{th clause satisfied} \\ 0 & \text{otherwise} \end{cases}$ ($i=1, \dots, m$)

$$E[Y_i] = 7/8 \quad (1 \text{ clause case})$$

$$E[\text{\# clauses satisfied}] = E\left[\sum_{i=1}^m Y_i\right] = \sum_{i=1}^m E[Y_i] = 7/8 m$$

let $Y_i = \begin{cases} 1 & \text{if } i\text{th clause satisfied} \\ 0 & \text{otherwise} \end{cases}$

$$Z = \sum_{i=1}^m Y_i = \text{\# clauses satisfied}$$

$$E[Z] = \frac{1}{2} E[Z | x_1 = \text{true}] + \frac{1}{2} E[Z | x_1 = \text{false}]$$

$\Rightarrow$ either $E[Z | x_1 = \text{true}]$ or $E[Z | x_1 = \text{false}]$ is $\geq E[Z]$!

QuickSort,

Random divide and conquer

① pick random 'pivot'



② Divide A into two parts



sort recursively



Expected running time of QuickSort

For each $i, j \in [n]$ w/ $i < j$,

$$X_{ij} = \begin{cases} 1 & \text{if } i\text{th smallest element is compared to } j\text{th smallest} \\ 0 & \text{otherwise} \end{cases}$$

$\sum_{i < j} X_{ij}$ = total # comparisons made by the algorithm
 $\approx$ total running time

$$E\left[\sum_{i < j} X_{ij}\right] = \text{average running time}$$

$$E\left[\sum_{i < j} X_{ij}\right] = \sum_{i < j} E[X_{ij}]$$

now analyze $E[X_{ij}]$ in isolation

$$E[X_{ij}] = \Pr[X_{ij} = 1] = \Pr[i\text{th element is compared to } j\text{th}]$$



i th elem is compared w/ j th
 $\Leftrightarrow$ i th or j th elem selected before any element in between happens w/ probability $\frac{2}{j-i+1}$

$$E[X_{ij}] = \frac{2}{j-i+1}$$

$$E\left[\sum_{i < j} X_{ij}\right] = \sum_{i < j} E[X_{ij}]$$

$$= \sum_{i < j} \frac{2}{j-i+1} = 2 \sum_{i < j} \frac{1}{j-i+1}$$

$$\leq 2n \left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \right)$$

$$\leq O(n \log n) \quad \text{with harmonic number}$$

Quickselect (HW)

Last Thursday

Probability

Hash Functions,

Universal Hash Functions

$h: [n] \rightarrow k$ s.t. for all pairs $i, j \in [n]$,
 $\Pr[h(i) = h(j)] \leq \frac{1}{k}$ (much weaker than ideal hash)

e.g. $h(x) = (ax + b) \bmod p \bmod k$

where • p prime, $> k$

• a in $\{1, \dots, p-1\}$ unif. random

• b in $\{0, \dots, p-1\}$ unif. random



Markov's Inequality

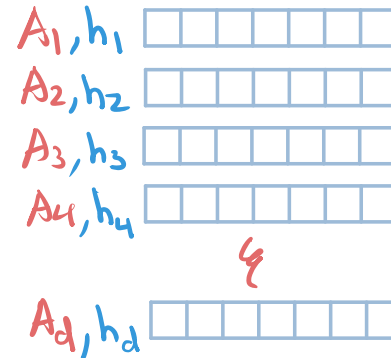
Markov's Inequality (special case)

Let $X \geq 0$ be nonnegative RV

$$\Pr[X \geq 2E[X]] \leq \frac{1}{2}$$

Algorithms

Frequency Estimation, Heavy Hitters (in streams)



estimate f_i w/
 $\min_{j=1, \dots, d} A_j[h_j(i)]$

Today

Probability

k -wise independence,
higher order moments

(generalizing universal hash
functions and Markov's inequality)

Algorithms

Hash tables w/
chaining +
linear probing

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see [Mapping Types — dict](#)). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of *key:value* pairs within the braces adds initial *key:value* pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a *key:value* pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

compact1, compact2, compact3
java.util

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

```
public interface Map<K,V>
```

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

The Map interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw `UnsupportedOperationException` if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the map. For example, invoking the `putAll(Map)` method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Many methods in Collections Framework interfaces are defined in terms of the `equals` method. For example, the specification for the `containsKey(Object key)` method says: "returns true if and only if this map contains a mapping for a key `k` such that (`key==null ? k==null : key.equals(k)`)." This specification should *not* be construed to imply that invoking `Map.containsKey` with a non-null argument `key` will cause `key.equals(k)` to be invoked for any key `k`. Implementations are free to implement optimizations whereby the `equals` invocation is avoided, for example, by first comparing the hash codes of the two keys. (The `Object.hashCode()` specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying `Object` methods wherever the implementor deems it appropriate.

Some map operations which perform recursive traversal of the map may fail with an exception for self-referential instances where the map directly or indirectly contains itself. This includes the `clone()`, `equals()`, `hashCode()` and `toString()` methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`HashMap`, `TreeMap`, `Hashtable`, `SortedMap`, `Collection`, `Set`

"Dictionaries" / "Maps"

maintain key-value pairs
(k_1, v_1), (k_2, v_2), ...

- insertion
- deletion
- lookup

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see Mapping Types — dict). Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type: strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

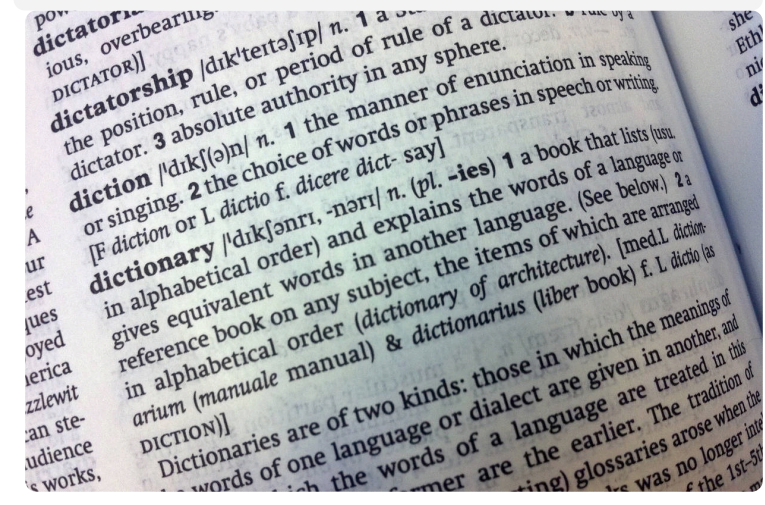
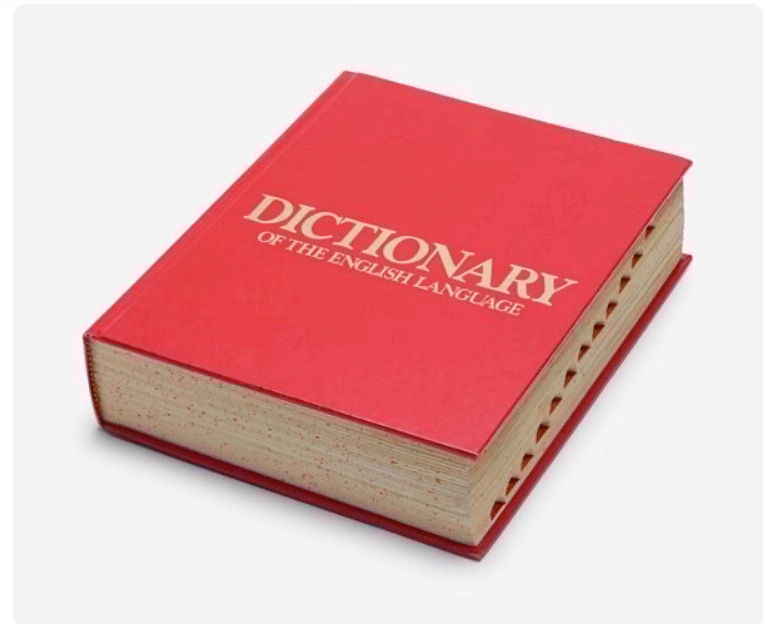
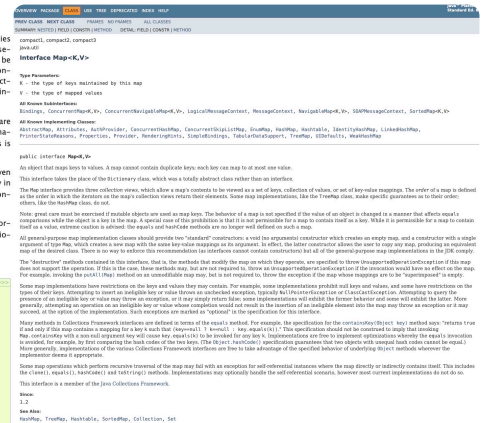
It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of *key:value* pairs within the braces adds initial *key:value* pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some *key* and extracting the value given the *key*. It is also possible to delete a *key:value* pair with `del`. If you store using a *key* that is already in use, the old value associated with that *key* is forgotten. It is an error to extract a value using a non-existent *key*.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order if you want it sorted, just use `sorted(d)` instead). To check whether a single *key* is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

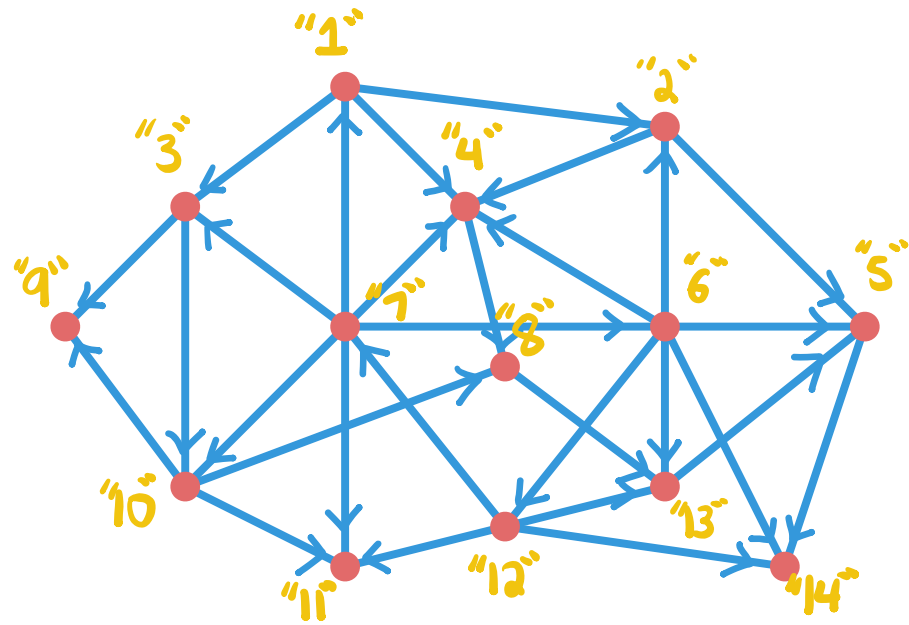


Very special case

keys = contiguous integers $\{1, \dots, n\}$

e.g. index vertices of
a graph $v_1, \dots, v_n$

(or after reindexing anything, really)



Solution: Array $A[1 \dots n]$

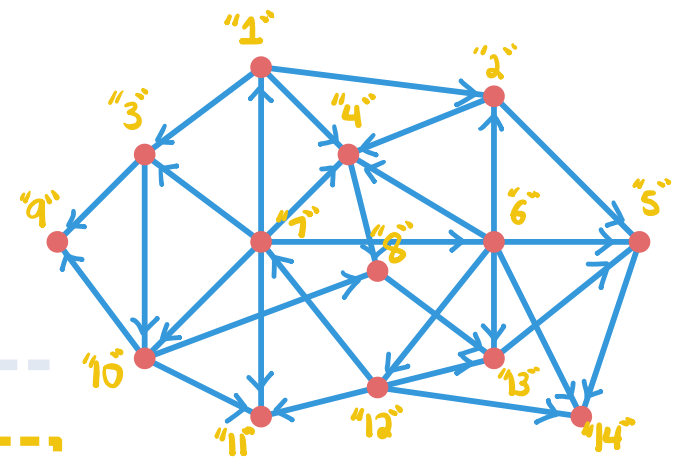


$O(1)$ access, $O(n)$ space

keys = contiguous integers $\{1, \dots, n\}$

e.g. index vertices of a graph $v_1, \dots, v_n$

(or after reindexing anything, really)



Solution: Array $A[1 \dots n]$

k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}	k_{11}	$\dots$
-------	-------	-------	-------	-------	-------	-------	-------	-------	----------	----------	---------

$O(1)$ access, $O(n)$ space

Perfect for static settings where we can reindex
the input once

Bad for dynamic settings where keys change

More general:

(U for "universe", U large)

keys = n integers out of $\{1, \dots, U\}$



e.g. strings = #'s
via bit encodings

HELLO
ASCII 72 69 76 76 79
 $\Rightarrow 72 + 69 \times 128 + 76 \times 128^2 + 76 \times 128^3 + 79 \times 128^4$

Idea 1: Array $A[1, \dots, U]$

$O(1)$ lookup / insert / delete

$O(U)$ space ($\gggg O(n)$)

keys = n integers out of $\{1, \dots, U\}$

(U for "universe", U large)



e.g. strings = #'s
via bit encodings

HELLO
ASCII 72 69 76 76 79
 $\Rightarrow 72 + 69 \times 128 + 76 \times 128^2 + 76 \times 128^3 + 79 \times 128^4$

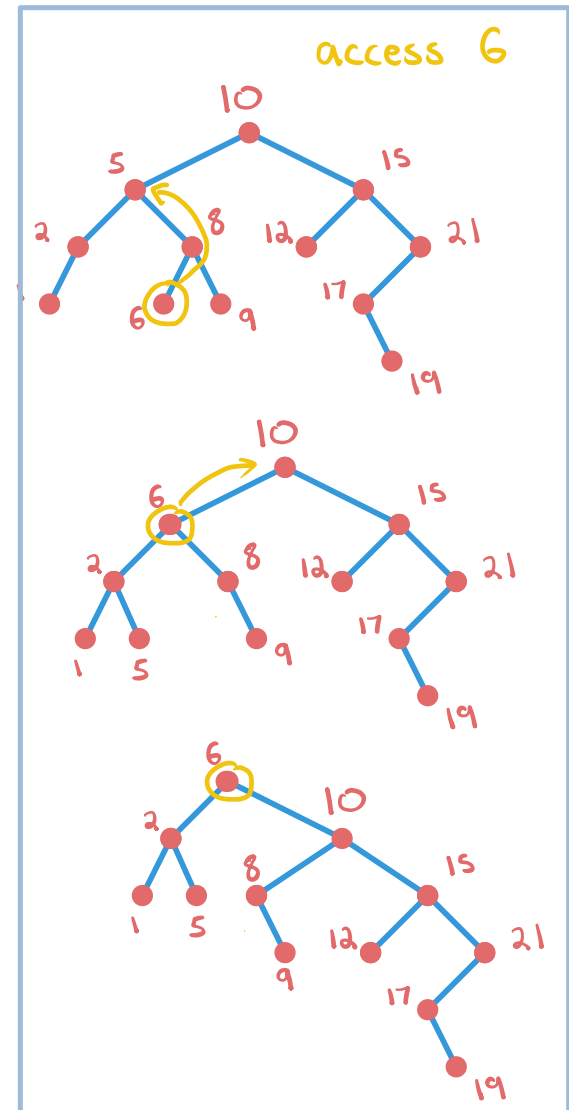
Idea 1: Array $A[1, \dots, U]$
 $O(1)$ lookup / insert / delete
 $O(U)$ space ($\gg \gg O(n)$)

Idea 2: self-balanced trees
 $O(n)$ space

$O(\log n)$ time per
lookup, insertion, deletion

(maybe amortized)

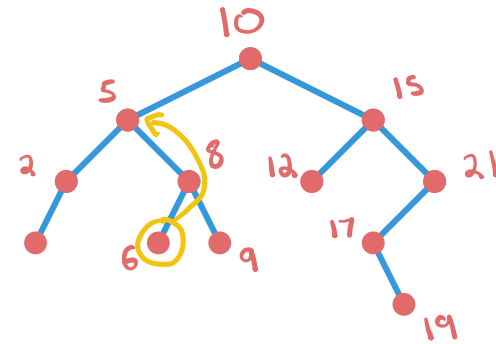
e.g. splay trees



Arrays



Search Trees



Time

$O(1)$

$O(\log n)$

Space

$O(1)$

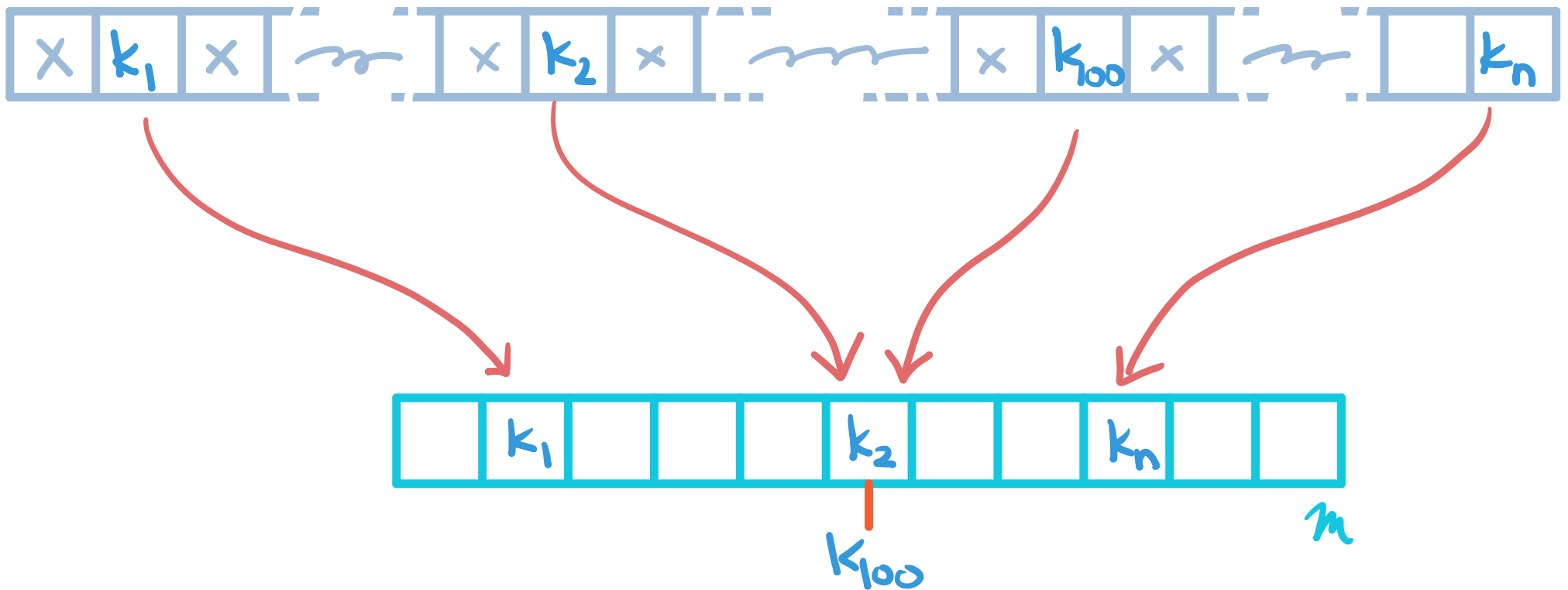
$O(n)$

Question: can we get the **best** of both worlds?

Yes, with randomization

Hashing

$$k_i \xrightarrow{\text{(is stored in)}} h(k_i) \bmod m$$



Goal: hash into a smaller array and achieve array-like performance

One problem: collisions

$$h(k_1) = h(k_2) \quad (\text{same slot})$$

Some ways to handle collisions

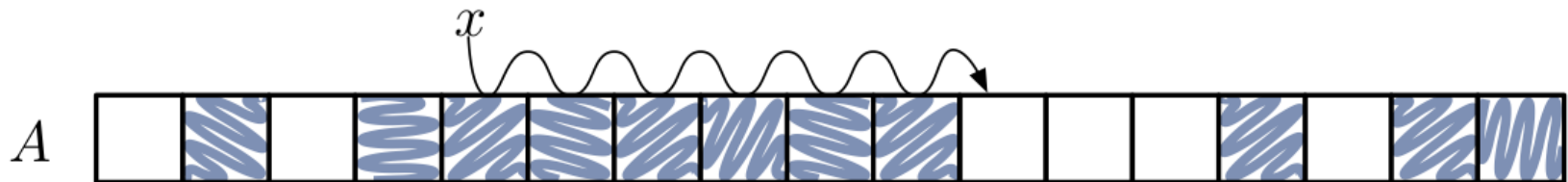
① make array big enough that there are (probably) no collisions

② "chaining": linked list at each cell

③ "cuckoo hashing" use two hash functions h_1 and h_2 .

if $h_1(k)$ and $h_2(k)$ already occupied, try to reassign one of the conflicts to their other hash, etc.

④ if $A[h(k)]$ filled, try $A[h(k)+1]$, $A[h(k)+2]$, ...



6
(and more)

"linear probing"

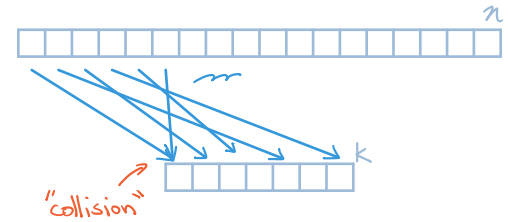
Recall:

Universal Hash Functions

$h: [n] \rightarrow k$ s.t. for all pairs $i, j \in [n]$,

$$\Pr[h(i) = h(j)] \leq \frac{1}{k}$$

(much weaker than ideal hash)



e.g. $h(x) = ((ax + b) \bmod p) \bmod k$

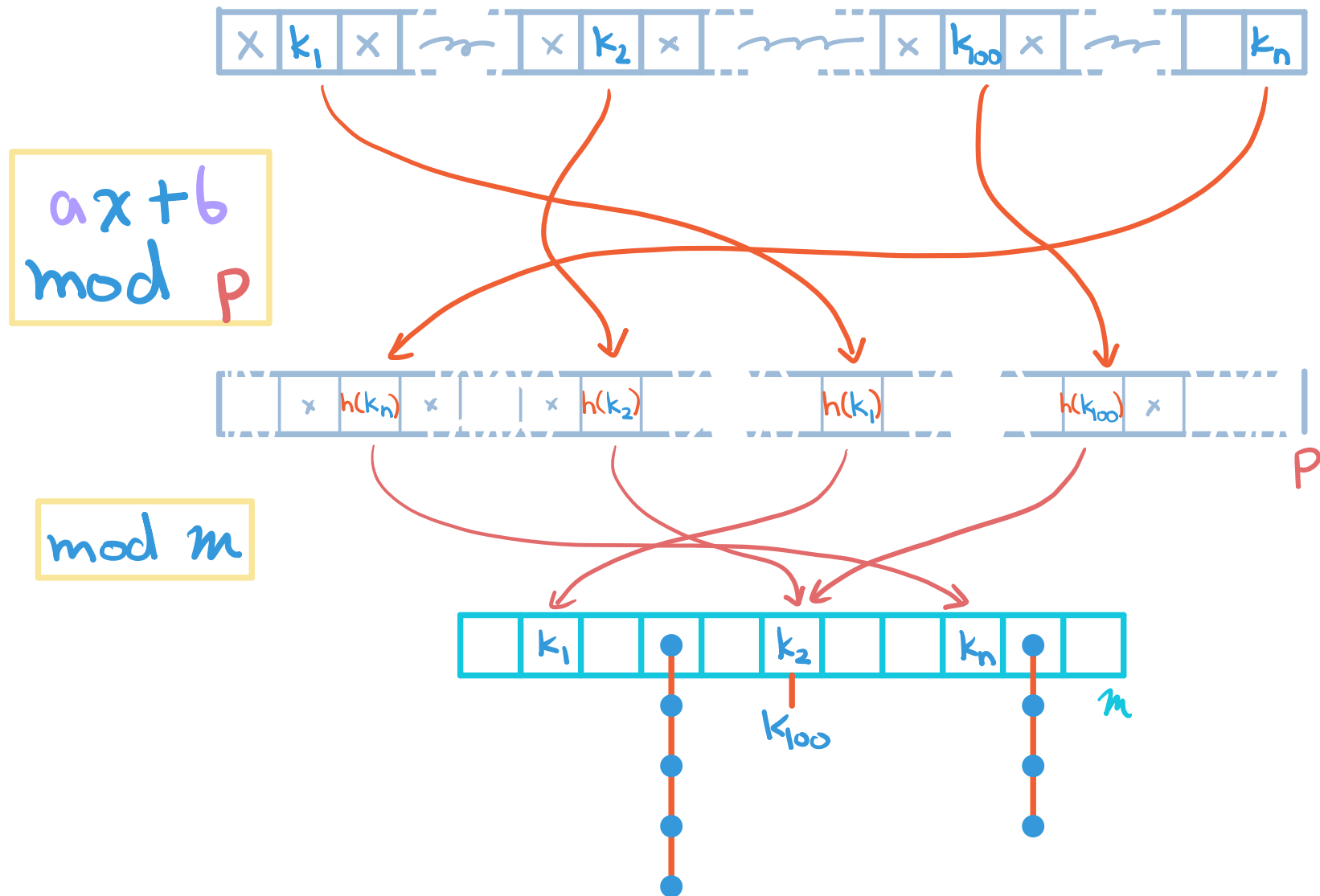
where

- p prime, $> k$
- a in $\{1, \dots, p\}$ unif. random
- b in $\{0, \dots, p-1\}$ unif. random

(proof left as exercise)

Chaining + Universal Hash Function

$$h(x) = (ax + b \bmod p) \bmod m$$

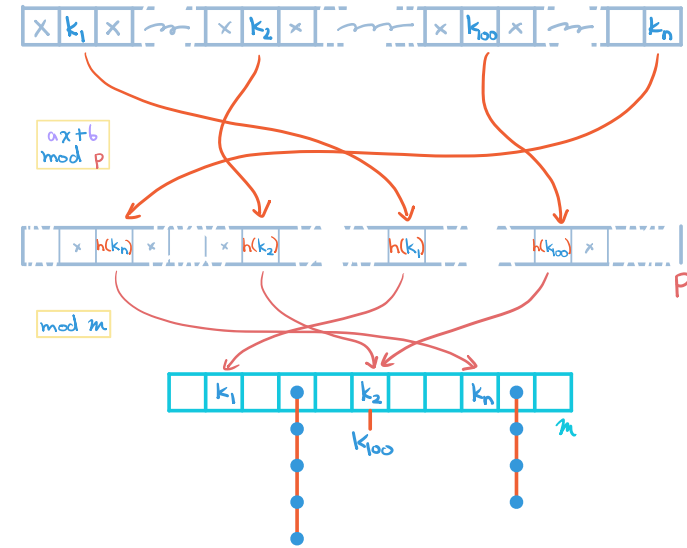


Theorem insert/delete/lookup take
 $O(1 + n/m)$ time in expectation

Proof

Chaining + universal hash

$$h(x) = (ax + b \bmod p) \bmod m$$



$n = \# \text{keys}$ $m = \text{size of array}$

Today

Probability

k-wise independence,
higher order moments

(generalizing universal hash
functions and Markov's inequality)

Algorithms

Hash tables w/
chaining +
linear probing

Today

Probability

k -wise independence,
higher order moments

(generalizing universal hash
functions and Markov's inequality)

Algorithms

Hash tables w/
chaining +
linear probing

Linear Probing

"insert x"



(insert in next empty slot)

Deletion "delete x"



① scan to find x , remove

② keep scanning to fill holes

(after deletion)



Linear Probing (insert in next empty slot)

"insert x"



Main Question: how long is "run"?
(on average)

(for $m = O(n)$, m = size of array, n = # keys)

Linear Probing (insert in next empty slot)



Main Question: how long is "run"? (on average)

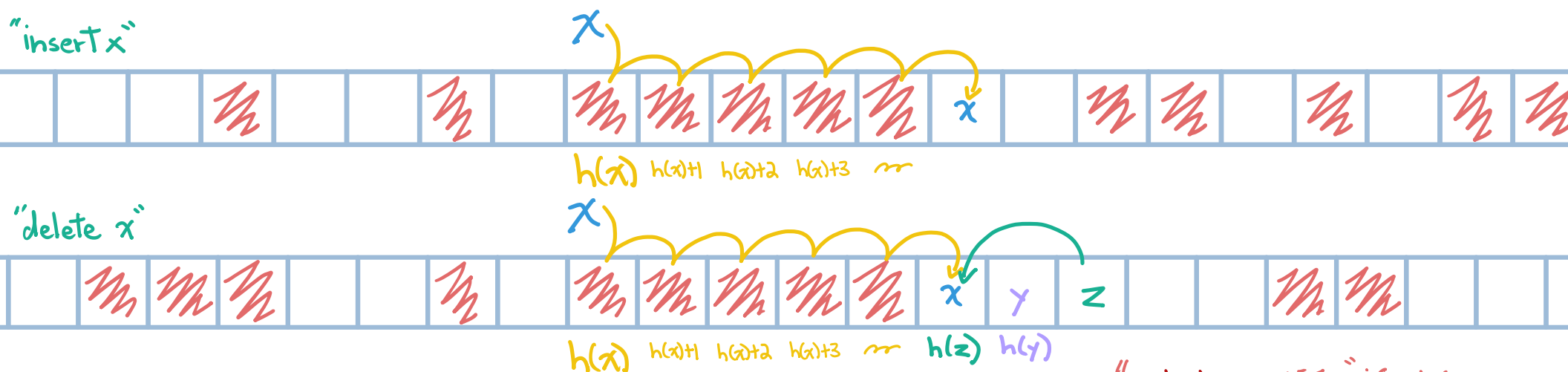
Terminology

"hashes": " x hashes to $A[i]$ " if $h(x)=i$

vs

"occupies": " x occupies $A[i]$ " if $A[i]=x$

" $[a,b]$ " = indices $\{a, a+1, \dots, b \bmod m\}$
(wraps around)



Main invariant

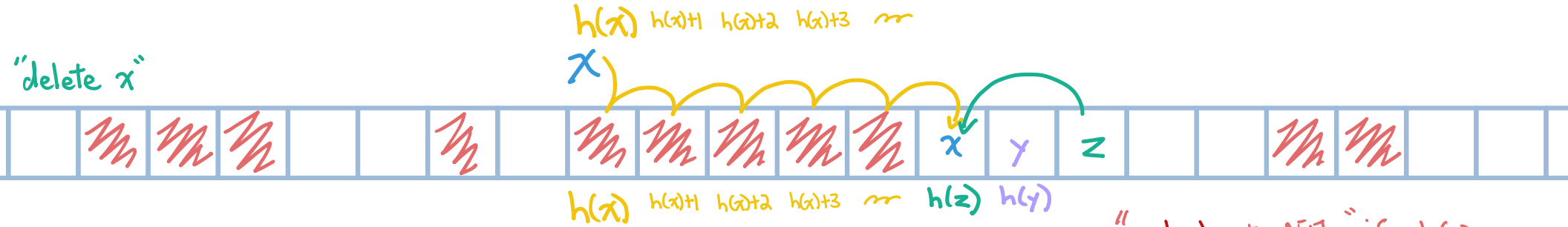
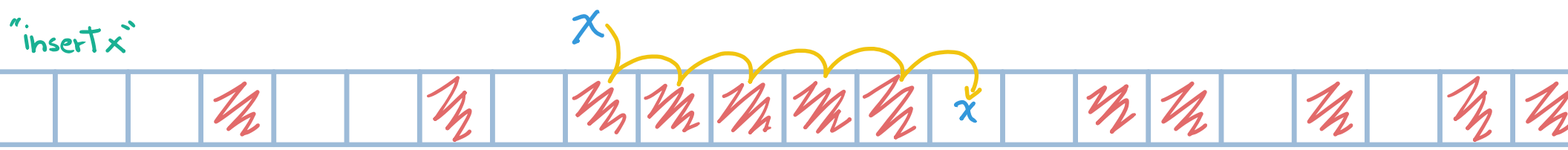
"x hashes to $A[i]$ " if $h(x) = i$

"x occupies $A[i]$ " if $A[i] = x$

" $[a, b] = \{a, a+1, \dots, b \bmod m\}$ "

if x occupies cell l , then
all of $[h(x), l]$ occupied





Main Question: how long is "run"?

(for $m = O(n)$, m = size of array, n = # keys) (on average)

"x hashes to $A[i]$ " if $h(x) = i$
 "x occupies $A[i]$ " if $A[i] = x$
 $[a, b] = \{a, a+1, \dots, b \bmod m\}$

• Knuth, 1964: $O(1)$ for ideal hash functions

what about w/ limited randomness?

k -wise independent hash functions

$$h: [U] \rightarrow [m]$$

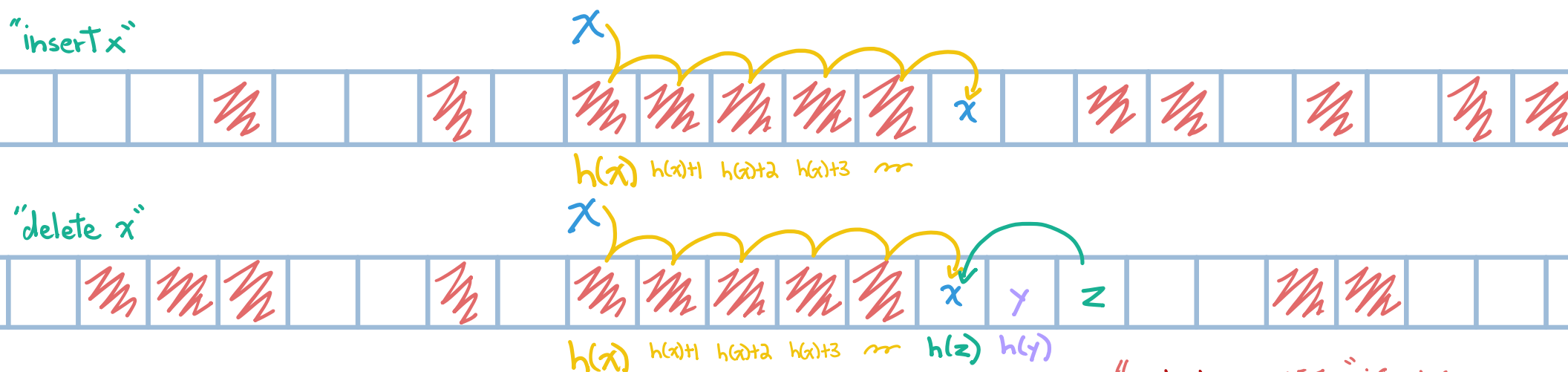
s.t. for any k keys $x_1, x_2, \dots, x_k \in [U]$,
 k values $v_1, v_2, \dots, v_k \in [m]$,

$$P[h(x_1)=v_1, h(x_2)=v_2, \dots, h(x_k)=v_k] = \frac{1}{m^k}$$

eg. $h(x) = a_0 + a_1x + a_2x^2 + \dots + a_{k-1}x^{k-1} \bmod p$

where $p \geq U$ prime,

$a_0, a_1, \dots, a_{k-1} \in [p]$ indep, uniformly random

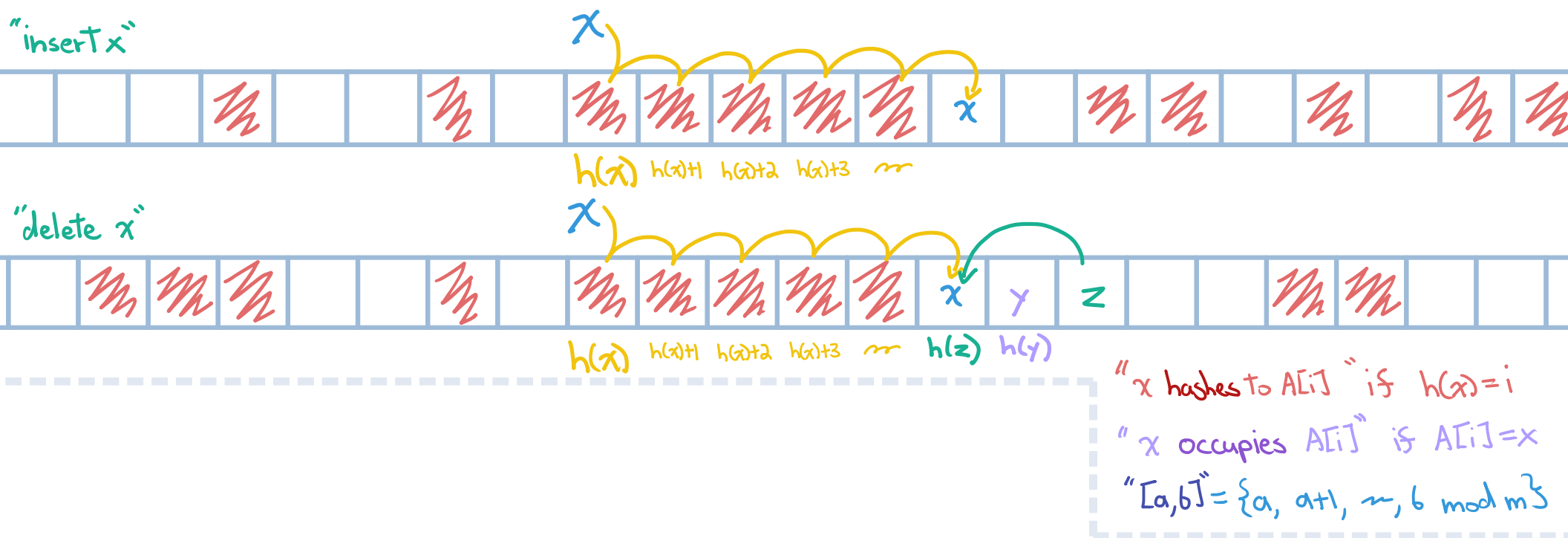


Main Question: how long is "run"?

(for $m = O(n)$, m = size of array, n = # keys) (on average)

"x hashes to $A[i]$ " if $h(x) = i$
 "x occupies $A[i]$ " if $A[i] = x$
 $[a, b] = \{a, a+1, \dots, b \bmod m\}$

- Knuth, 1964: $O(1)$ for ideal hash functions
- Schmidt, Siegel '90: $O(1)$ for $O(\log n)$ -wise independent hash function
- Pagh, Pagh, Ruzic 2007: $O(1)$ for 5-wise independent hash function



Theorem 4. Let h be 5-wise independent. For $m \geq 8n$, linear probing takes expected constant time per operation.

Today

Probability

k -wise independence,
higher order moments

(generalizing universal hash
functions and Markov's inequality)

Algorithms

Hash tables w/
chaining +
linear probing

k-wise independent random variables

$X_1, \dots, X_n$ are k-wise independent random variables if

for all distinct $X_{i_1}, \dots, X_{i_k}$

$$\Pr[X_{i_1} = y_1, X_{i_2} = y_2, \dots, X_{i_k} = y_k]$$

$$= \Pr[X_{i_1} = y_1] \Pr[X_{i_2} = y_2] \dots \Pr[X_{i_k} = y_k]$$

for all values $y_1, \dots, y_k$

Important lemma

k-wise independent random variables

$$\Pr[X_{i_1}=y_1, X_{i_2}=y_2, \dots, X_{i_k}=y_k]$$

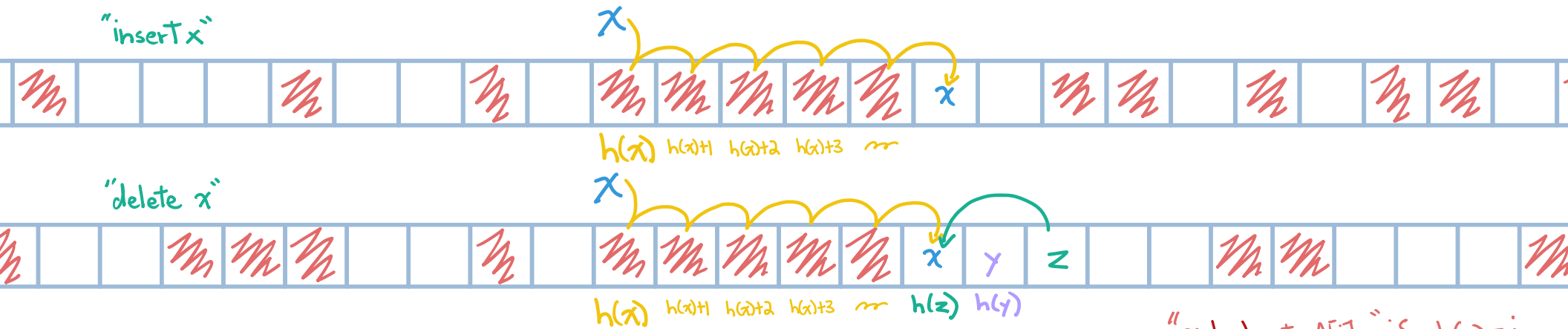
$$= \Pr[X_{i_1}=y_1] \Pr[X_{i_2}=y_2] \dots \Pr[X_{i_k}=y_k]$$

for all values $y_1, \dots, y_k$

let $X_1, \dots, X_n \in \{0,1\}$ be 4-wise independent
and $\mu = E[X_1 + \dots + X_n]$. then for $\beta > 0$,

$$\Pr[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^2\}}{\beta^4}$$

(Markov gives only $\frac{1}{1+d}$)



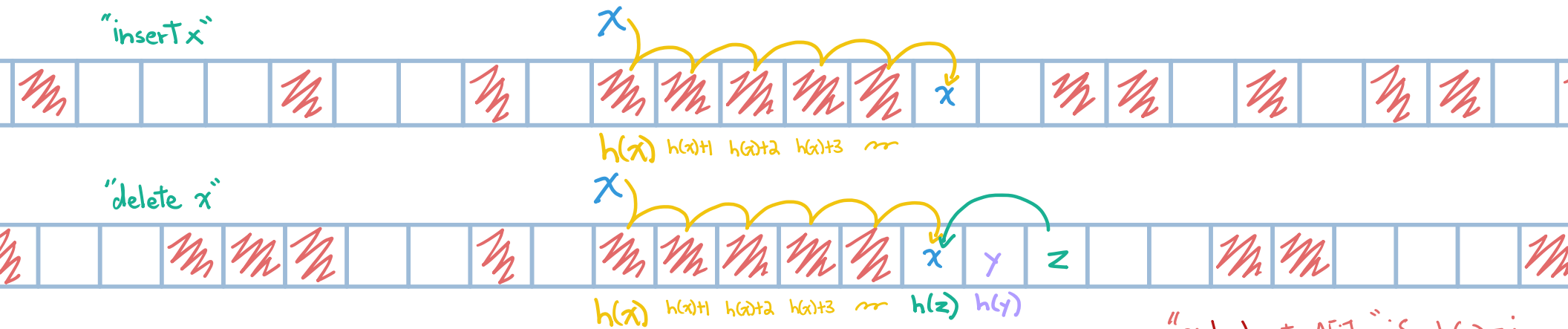
Theorem 4. Let h be 5-wise independent. For $m \geq 8n$, linear probing takes expected constant time per operation.

"Runs"



maximal occupied interval

"x hashes to $AE[i]$ " if $h(x) = i$
 "x occupies $AE[i]$ " if $AE[i] = x$
 $[a, b] = \{a, a+1, \dots, b \bmod m\}$
 k-wise independent random variables
 $\Pr[X_{i_1} = y_1, X_{i_2} = y_2, \dots, X_{i_k} = y_k]$
 $= \Pr[X_{i_1} = y_1] \Pr[X_{i_2} = y_2] \dots \Pr[X_{i_k} = y_k]$
for all values $y_1, \dots, y_k$
 let $X_1, \dots, X_n \in \{0, 1\}$ be 4-wise independent
 and $\mu = E[X_1 + \dots + X_n]$ for $\beta > 0$,
 $\Pr[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^2\}}{\beta^4}$



Theorem 4. Let h be 5-wise independent. For $m \geq 8n$, linear probing takes expected constant time per operation.

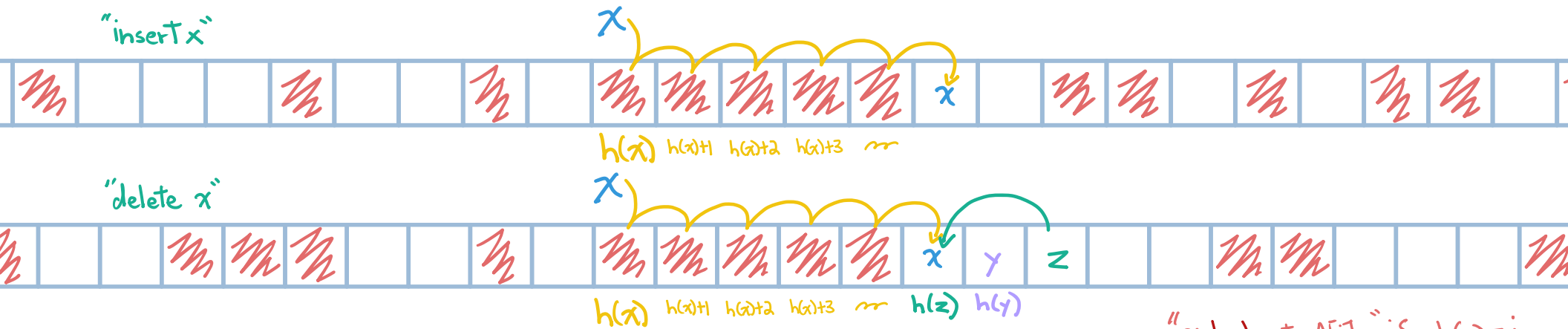
setup: fix x .

- $i = h(x)$
- $R = \text{run at } i$

goal: show $E[|R|] = O(1)$

" x hashes to $AE[i]$ " if $h(x) = i$
 " x occupies $AE[i]$ " if $AE[i] = x$
 $[a, b] = \{a, a+1, \dots, b \bmod m\}$
 k -wise independent random variables
 $\Pr[X_{i_1} = y_1, X_{i_2} = y_2, \dots, X_{i_k} = y_k]$
 $= \Pr[X_{i_1} = y_1] \Pr[X_{i_2} = y_2] \dots \Pr[X_{i_k} = y_k]$
for all values $y_1, \dots, y_k$
 let $X_1, \dots, X_n \in \{0, 1\}$ be 4-wise independent
 and $\mu = E[X_1 + \dots + X_n]$ for $\beta > 0$,
 $\Pr[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^2\}}{\beta^4}$

"run"



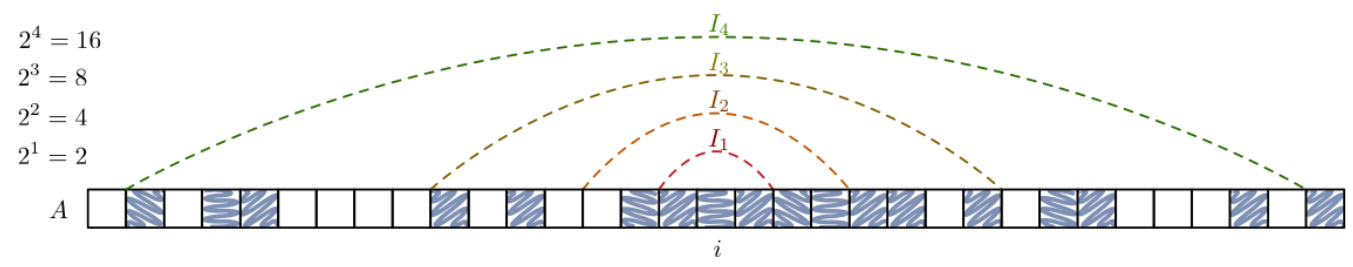
Theorem 4. Let h be 5-wise independent. For $m \geq 8n$, linear probing takes expected constant time per operation.

"Diadic intervals"

For each $k \in \mathbb{N}$, let

$$I_k = [i - (2^k - 1), i + 2^k - 1]$$

be the interval of length $2^{k+1} - 1$ centered at i .



"x hashes to $A[i]$ " if $h(x) = i$
 "x occupies $A[i]$ " if $A[i] = x$
 $[a, b] = \{a, a+1, \dots, b \bmod m\}$
 k -wise independent random variables
 $\Pr[X_{i_1} = y_1, X_{i_2} = y_2, \dots, X_{i_k} = y_k]$
 $= \Pr[X_{i_1} = y_1] \Pr[X_{i_2} = y_2] \dots \Pr[X_{i_k} = y_k]$
 for all values $y_1, \dots, y_k$

let $X_1, \dots, X_n \in \{0, 1\}$ be 4-wise independent and $\mu = E[X_1 + \dots + X_n]$ for $\beta > 0$,

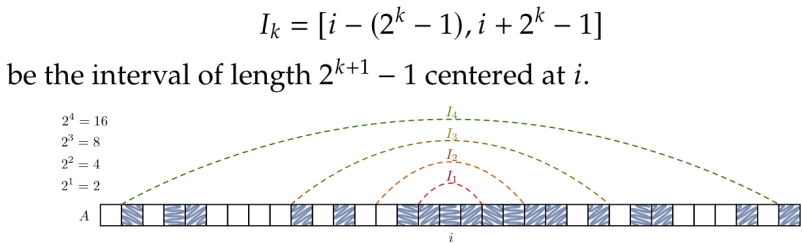
$$\Pr[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^3\}}{\beta^4}$$



setup: Six x .
 • $i = h(x)$
 • R = run at i
 goal: show $E[|R|] = O(1)$

$$|R| < 2^k \Rightarrow R \subseteq I_{k+1}$$

Theorem 4. Let h be 5-wise independent. For $m \geq 8n$, linear probing takes expected constant time per operation.



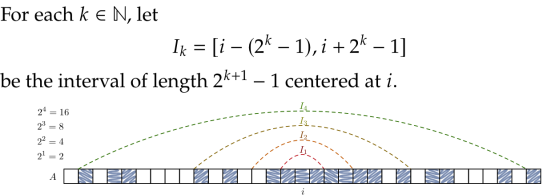
- $|R| < 2^k \Rightarrow R \subseteq I_{k+1}$
- $|R| > 2^{k-1} \Rightarrow \geq 2^{k-1}$ items (other than x) hash into R

$$\begin{aligned} &P[2^{k-1} < |R| \leq 2^k] \\ &\leq P[\geq 2^{k-1} \text{ items hash into } I_k] \end{aligned}$$

" x hashes to $A[i]$ " is $h(x)=i$
" x occupies $A[i]$ " is $A[i]=x$
" $[a,b]=\{a, a+1, \dots, b \bmod m\}$ "
 k -wise independent random variables
 $\Pr[X_1=y_1, X_2=y_2, \dots, X_k=y_k]$
 $= \Pr[X_1=y_1] \Pr[X_2=y_2] \dots \Pr[X_k=y_k]$
for all values $y_1, \dots, y_k$

let $X_1, \dots, X_n \in \{0,1\}$ be 4-wise independent
and $\mu = E[X_1 + \dots + X_n]$ for $\beta > 0$,
 $\Pr[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^2\}}{\beta^4}$

setup: six x .
• $i=h(x)$
• R = run at i
goal: show $E[|R|] = O(1)$
"Diadic intervals"



$"x \text{ hashes to } A[i]" \text{ is } h(x) = i$
 $"x \text{ occupies } A[i]" \text{ is } A[i] = x$
 $"[a, b] = \{a, a+1, \dots, b \bmod m\}$
 k -wise independent random variables
 $P_r[X_{i_1} = y_1, X_{i_2} = y_2, \dots, X_{i_k} = y_k]$
 $= P_r[X_{i_1} = y_1] P_r[X_{i_2} = y_2] \dots P_r[X_{i_k} = y_k]$
 For all values $y_1, \dots, y_k$

$$[a, b] = \{a, a+1, \dots, b \pmod m\}$$
$$Pr[X_{i_1} = y_1, X_{i_2} = y_2, \dots, X_{i_k} = y_k]$$
$$= P_r[X_{i_1}=y_{i_1}] P_r[X_{i_2}=y_{i_2}] \dots P_r[X_{i_k}=y_{i_k}]$$

for all values $y_{i_1}, \dots, y_{i_k}$

let $X_1, \dots, X_n \in \{0, 1\}$ be 4-wise independent
and $\mu = E[X_1 + \dots + X_n]$ for $\beta > 0$,

$$P_r[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^2\}}{\beta^4}$$



- $i = h(x)$

- $R = \text{run at } i$

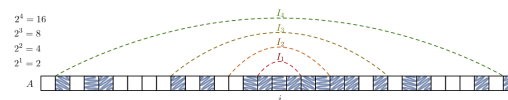
goal: show $E[|R|] = O(1)$

"Diadic intervals"

For each $k \in \mathbb{N}$, let

$$I_k = [i - (2^k - 1), i + 2^k - 1]$$

be the interval of length $2^{k+1} - 1$ centered at i .


$$X_i = \begin{cases} 1 & \text{if } i\text{th key hashes into } I_k \\ 0 & \text{otherwise} \end{cases}$$

conditional on $h(x)$,

$X_1, \dots, X_n$ are

4-wise independent

Theorem 4. Let h be 5-wise independent. For $m \geq 8n$, linear probing takes expected constant time per operation.

$$P[\geq 2^{k-1} \text{ items hash into } I_k]$$

$$X_i = \begin{cases} 1 & \text{if } i\text{th key hashes into } I_k \\ 0 & \text{otherwise} \end{cases}$$

$$Pr[X_1 + \dots + X_n \geq 2^{k-1}]$$

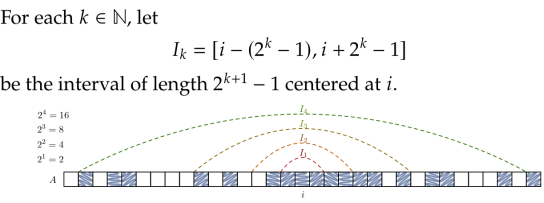
" x hashes to $AI[i]$ " is $h(x)=i$
" x occupies $AI[i]$ " is $AI[i]=x$
" $[a,b]=\{a, a+1, \dots, b \bmod m\}$ "
 k -wise independent random variables
 $Pr[X_{i_1}=y_1, X_{i_2}=y_2, \dots, X_{i_k}=y_k]$
 $= Pr[X_{i_1}=y_1] Pr[X_{i_2}=y_2] \dots Pr[X_{i_k}=y_k]$
for all values $y_1, \dots, y_k$

let $X_1, \dots, X_n \in \{0,1\}$ be 4-wise independent
and $\mu = E[X_1 + \dots + X_n]$ for $\beta > 0$,

$$Pr[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^2\}}{\beta^4}$$



setup: Six x_i .
• $i=h(x)$
• R = run at i
goal: show $E[|R|] = O(1)$
"Diadic intervals"



$"x \text{ hashes to } A[i]" \text{ is } h(x) = i$
 $"x \text{ occupies } A[i]" \text{ is } A[i] = x$
 $"[a, b] = \{a, a+1, \dots, b \bmod m\}$
 $k\text{-wise independent random variables}$

$$Pr[X_{i_1} = y_1, X_{i_2} = y_2, \dots, X_{i_k} = y_k]$$

$$= Pr[X_{i_1} = y_1] Pr[X_{i_2} = y_2] \dots Pr[X_{i_k} = y_k]$$

For all values $y_1, \dots, y_k$

k-wise independent random variables

$$\Pr[X_{i_1}=y_1, X_{i_2}=y_2, \dots, X_{i_k}=y_k]$$
$$= \Pr[X_{i_1}=y_1] \Pr[X_{i_2}=y_2] \dots \Pr[X_{i_k}=y_k]$$

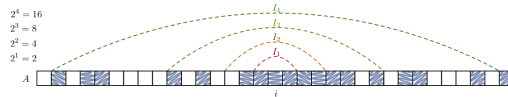
for all values $y_1, \dots, y_k$

Let $X_1, \dots, X_n \in \{0, 1\}$ be 4-wise independent and $\mu = E[X_1 + \dots + X_n]$ for $\beta > 0$,

$$Pr[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^3\}}{\beta^4}$$



- goal: show $E[|R|] = O(1)$
"Diadic intervals"

$$I_k = [i - (2^k - 1), i + 2^k - 1]$$
$$\begin{aligned} 2^4 &= 16 \\ 2^3 &= 8 \\ 2^2 &= 4 \\ 2^1 &= 2 \end{aligned}$$


Important lemma

k-wise independent random variables

$$\Pr[X_{i_1}=y_1, X_{i_2}=y_2, \dots, X_{i_k}=y_k]$$

$$= \Pr[X_{i_1}=y_1] \Pr[X_{i_2}=y_2] \dots \Pr[X_{i_k}=y_k]$$

for all values $y_1, \dots, y_k$

let $X_1, \dots, X_n \in \{0,1\}$ be 4-wise independent
and $\mu = E[X_1 + \dots + X_n]$. then for $\beta > 0$,

$$\Pr[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^2\}}{\beta^4}$$

let $X_1, \dots, X_n \in \{0, 1\}$ be 4-wise independent
and $\mu = E[X_1 + \dots + X_n]$. then for $\beta > 0$,

$$P[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^2\}}{\beta^4}$$

Proof

let $X_1, \dots, X_n \in \{0, 1\}$ be 4-wise independent
and $\mu = E[X_1 + \dots + X_n]$. then for $\beta > 0$,

$$P[X_1 + \dots + X_n \geq \mu + \beta] \leq \frac{4 \max\{\mu, \mu^2\}}{\beta^4}$$