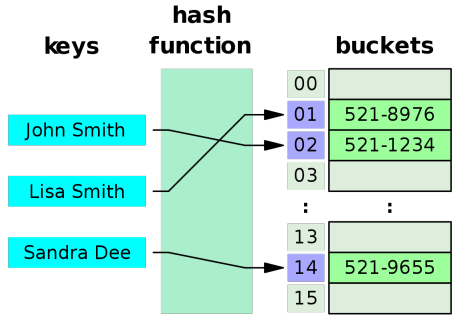


Hash Tables
Lecture 9

COSC 242 – Algorithms and Data Structures

Today’s outline

1. Overview
2. Direct addressing
3. Hash tables
4. Hash functions
5. The division method
6. Open addressing
7. Probing

Today’s outline

1. Overview
2. Direct addressing
3. Hash tables
4. Hash functions
5. The division method
6. Open addressing
7. Probing

Value-oriented data structures

Consider the data structures you have already met (arrays, stacks, queues, etc).

All have basic data management operations: insert, delete, search.

However, most are **position-oriented**. Their operations have the form:

- Insert into cell *i*, or at the top of the stack
- Delete from cell *j*, or from the front of the queue
- Retrieve what's in cell *k*, or at the top of the stack

Value-orientated data structures in COSC242

But many simple tasks involve values, not positions; e.g. find a person's number in a phone book. Here, the value is the person's name.

You've seen one value-oriented data structure in COSC241 - the heap.

Over the next 4 weeks, we'll be looking at three others: hash tables; binary search trees (BST); red-black trees (a type of BST).

A table data type

In a table every ‘object’ or ‘record’ (represented by a row) has some key that uniquely identifies it:

ALEC	8299
SHARLENE	8581
ANDY	8314
BARRY	5691
MIKE	8588
CAROL	8578
LANA	8580

Implementations

There are a few possibilities for implementing such a thing:

Array: Simple: can store keys in sorted order and use binary search to and a key. Search is O(log *n*), but insert is O(*n*).

Binary Search Tree: Implementation more difficult. Search is O(log *n*) and insert is O(log *n*).

Hash table: Use our key (first name in the previous slide) to compute an index or address. Search and insert can be O(1) if the table is big enough and the hash function is a good one.

Associative arrays

A hash table is an implementation of an **associative array**.

An **associate array**, also called a symbol table, map, or dictionary, is composed of a collection of (Key, Value) pairs.

A given Key can appear at most once in the collection.

This data structure allows for the following operations:

- Addition of a pair
- Removal of a pair
- Modification of a pair
- Value lookup associated with a Key

Associative arrays

Many applications require a dynamic set that supports only the **dictionary operations** INSERT, SEARCH, and DELETE. E.g., a symbol table in a compiler.

There are several different implementations of associate arrays.

The main two are: Hash tables, and Tree-based implementations.

For the next few lectures we will be looking at the first of these: Hash tables.

Hash table implementation

A hash table is an effective and widely-used implementation of an associative array.

The expected time to search for an element in a hash table is O(1), under some reasonable assumptions.

Worst-case search time is $\Theta(n)$, however.

Direct addressing

A hash table is a generalization of an ordinary array.

- With an ordinary array, we store the element whose Key is *k* in position *k* of the array.
- Given a key *k*, we find the element whose key is *k* by just looking in the *k*th position of the array. This is called **direct addressing**.
- Direct addressing is applicable when we can afford to allocate an array with one position for every possible key.

Hash table

We use a hash table when we do not want to (or cannot) allocate an array with one position per possible key.

- Use a hash table when the number of keys actually stored is small relative to the number of possible keys.
- A hash table is an array, but typically uses a size proportional to the number of keys to be stored (rather than number of possible keys).
- Given a key *k*, don't just use k as the index into the array. Instead, compute a function of *k*, and use that value to index into the array. We call this function a **hash function**.

Today’s outline

1. Overview
2. Direct addressing
3. Hash tables
4. Hash functions
5. The division method
6. Open addressing
7. Probing

Direct addressing

Direct addressing is a simple technique that works well when the universe *U* of keys is reasonably small.

Use case scenario

- Maintain a dynamic set.
- Each element has a key drawn from a universe U = {0, 1,…, *m* - 1} where *m* isn't too large. Here, *m* is the number of keys in our universe.
- No two elements have the same key.

Direct addressing

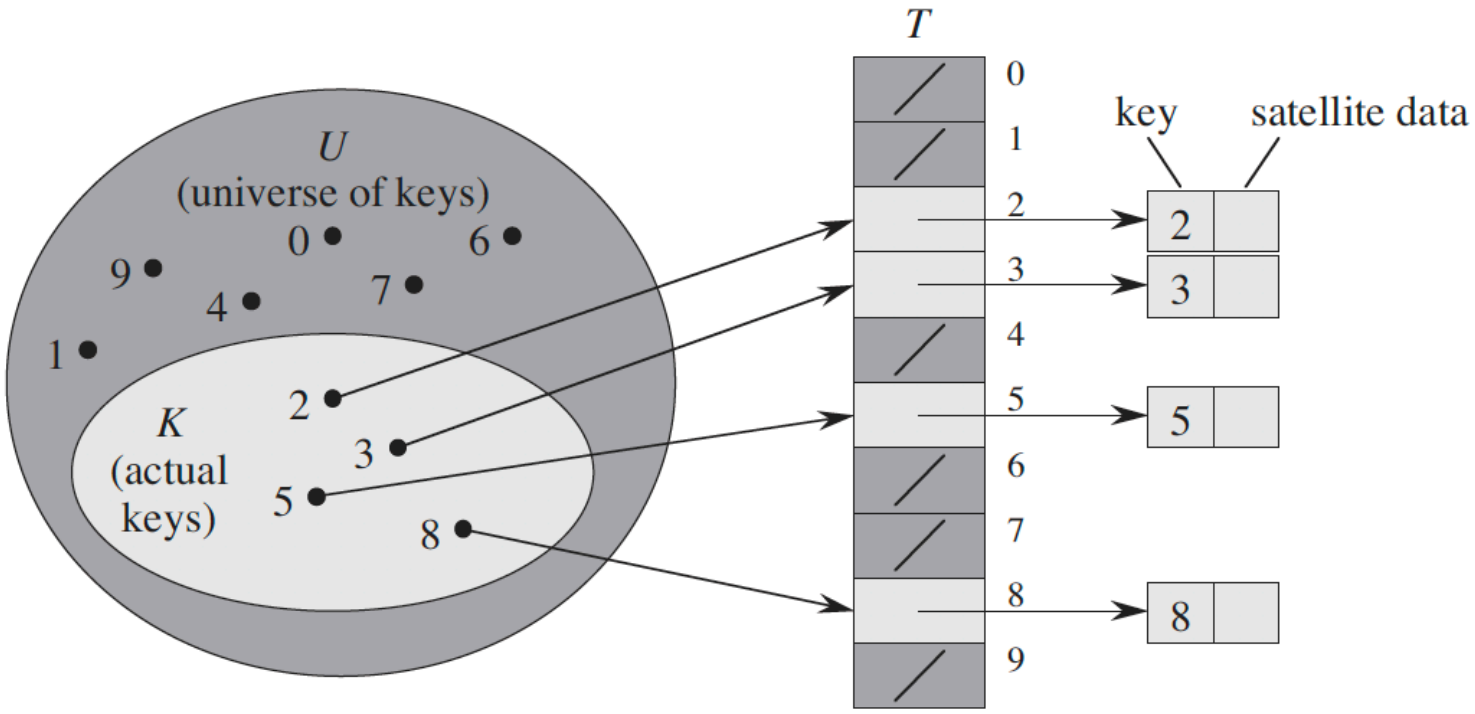
We can use a **direct-address table**, or array, T[0, …, m - 1] to represent this dynamic set.

Each slot, or position, corresponds to a key in U.

If there's an element x with key k, then $T[k]$ contains a pointer to x.

Otherwise, $T[k]$ is empty, represented by NIL.

Direct addressing



Universe of keys: $U = \{0,1,\ldots,9\}$
Actual keys: $K = \{2, 3, 5, 8\}$

Dark grey slots contain NIL

Dictionary operations

Dictionary operations are trivial and take O(1) time each:

DIRECT-ADDRESS-SEARCH(T, k)

    return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

    $T[key[x]] = x$

DIRECT-ADDRESS-DELETE(T, x)

    $T[key[x]]$ = NIL

Today’s outline

1. Overview
2. Direct addressing
3. Hash tables
4. Hash functions
5. The division method
6. Open addressing
7. Probing

Hash tables

The problem with direct addressing is if the universe U is large, storing a table of size $|U|$ may be impractical or impossible.

Often, the set K of keys actually stored is small, compared to U, so that most of the space allocated for T is *wasted*.

When K is much smaller than U, a **hash table** requires much less space than a direct-address table. Can reduce storage requirements to $\Theta(|K|)$

Can still get O(1) search time, but in the *average case*, not the *worst case*.

Example: wasted space

If you have the keys 5, 3, 8, 9, 6 then you could insert key *i* into position A[*i*] of an array A of length 10.

But if your keys are 20 digit numbers, you'd need an array of length $1x10^{20}$. Given that a Double can represent values between 10^{-308} to 10^{308}, this scenario is a real possibility in some applications.

However, you're likely to have far fewer than $1x10^{20}$ records (that is, 100 quintillion data points). So direct addressing is usually a bad idea.

Direct addressing vs Hashing

Instead of storing an element with key *k* in slot *k*, use a function *h* and store the element in slot *h*(*k*).

For hash table *T*[0,…,m - 1]

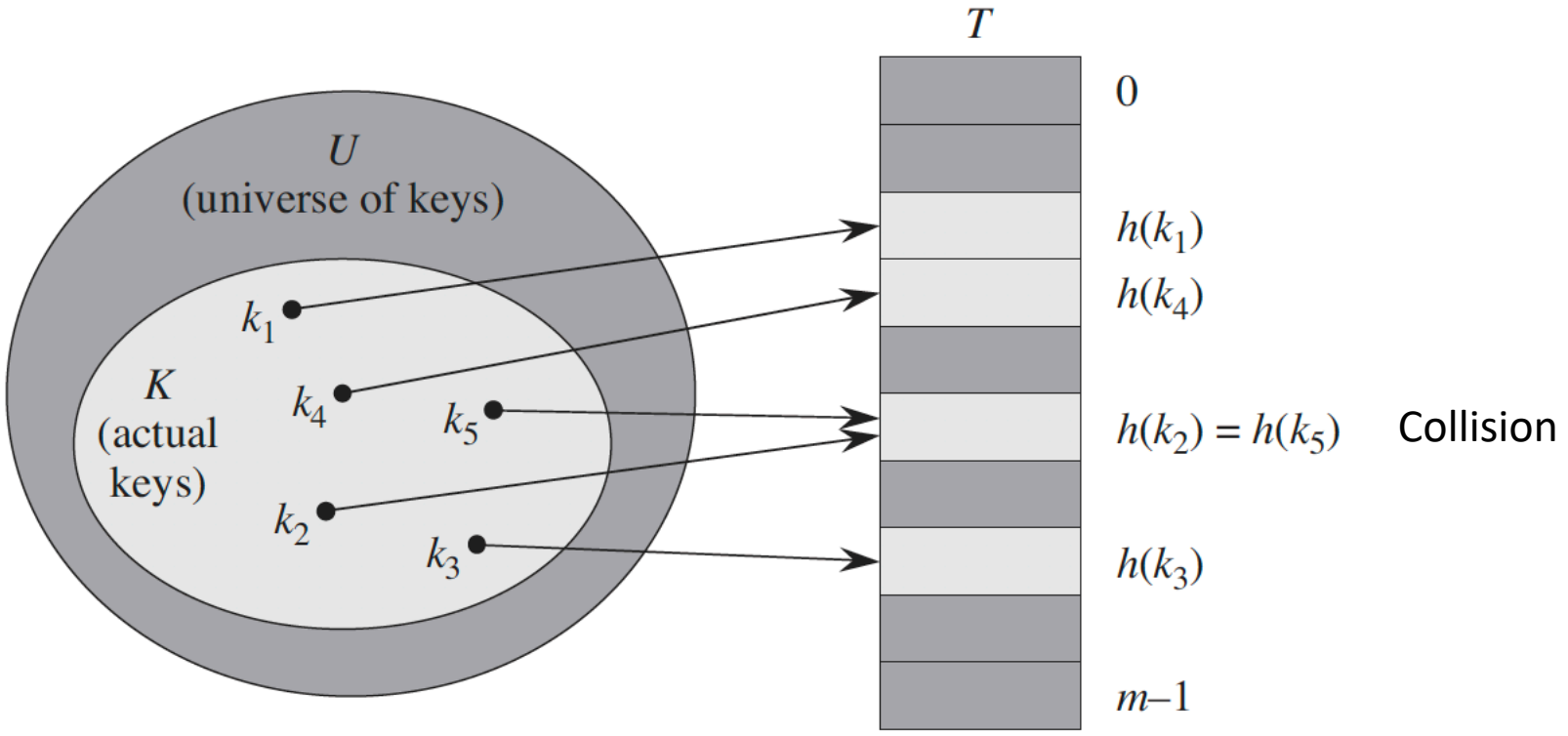
We call *h* a **hash function**.

h : *U* →{0, 1, …, *m* - 1}, so that *h*(*k*) is a legal slot number in *T*.

We say that *k* **hashes** to slot *h*(*k*)

We also say that *h*(*k*) is the **hash value** of key *k*.

Hashing



Using hash function *h* to map keys to hash-table slots.
Keys k_2 and k_5 map to the same slot, and collide.

Saving space, but now collisions

The hash function reduces the range of array indices and hence the size of the array.

Instead of a size of $|U|$, the array can have size m.

There is one hitch: two keys may hash to the same slot. We call this situation a **collision**.

Collisions

A collision can happen when there are more possible keys than slots ($|U| > m$)

For a given set K of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$.

Therefore, must be prepared to handle collisions in all cases.

Use two methods: chaining and open addressing (see Lecture 10).

Today’s outline

1. Overview
2. Direct addressing
3. Hash tables
4. Hash functions
5. The division method
6. Open addressing
7. Probing

What makes a good hash function?

A good hash function satisfies (approximately) the assumption of simple uniform hashing.

Uniform hashing - each key is equally likely to hash to any of the *m* slots, independently of where any other key has hashed to.

In practice, it's not possible to satisfy this assumption, since we don't know in advance the probability distribution that keys are drawn from, and the keys may not be drawn independently.

What makes a good hash function?

Instead, we will often use heuristics, based on the domain of the keys, to create a hash function that performs well.

A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data.

One such method is the ”division method”, which we will see shortly. This technique computes the hash value as the remainder when the key is divided by a specified prime number.

Keys as natural numbers

Hash functions assume that the keys are natural numbers.
$\mathbb{N} = \{0,1,2, \ldots\}$

When keys are not natural numbers, we have to find a way to interpret them as natural numbers.

Consider the string “CLRS”. How might we represent this as a number?

We’ll come back to this shortly…

Today’s outline

1. Overview
2. Direct addressing
3. Hash tables
4. Hash functions
5. The division method
6. Open addressing
7. Probing

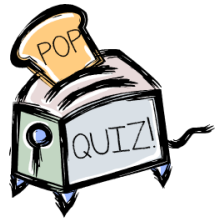
Division method

In the **division method** for creating hash functions, we map a key *k* into one of *m* slots by taking the remainder of *k* divided by *m*.

h(k) = *k* mod *m*

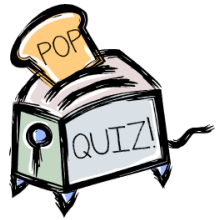
Here, *m* is the size of the array, *k* is the key, and *k*%*m* is the remainder after dividing *k* by *m*.

Pop quiz 1



If m = 12, and k = 100, what will our hash function yield?

Pop quiz 2



If m = 20, and k = 91, what will our hash function yield?

What's the problem here?

0		
1		
2		
.		
25	001364825	Whatever data goes with this key
.		
97		
98		
99		

Things to consider

Advantage: Fast, since requires just one division operation.

Disadvantage: Have to avoid certain values of m:

When selecting size of array m, avoid powers of 2. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.

What does that mean?

Why are powers of 2 a problem?

Think about how modulus works, and how the remainder relates to the original number. Such a system would likely produce many collisions.

$2341489 \% 2^{16} = 47729$

Decimal	Binary
2341489	10001110111010011100 01
47729	1011101001110001

Things to consider

Rule of thumb: m should be a prime not too close to an exact power of 2. E.g., 37 rather than 31.

We will still need a collision resolution strategy, because perfect hash functions are rare. We'll come back to this…

Keys that are strings

The key field may be of any ordinal type, including character strings. We need a way to convert them to numbers first.

A reasonable transformation takes into account the position of each character, say by multiplying the ASCII values by a number raised to a power that reflects the position.

E.g., we multiply the first character by the most, and the last character by the least, we could define $h(k)$:

$h(\text{“ALAN”}) = 0 \cdot 2^3 + 11 \cdot 2^2 + 0 \cdot 2^1 + 13 \cdot 2^0 = 57$

Keys that are strings

Strings up to length about 25 would fit into a 4-byte integer. What about longer strings?

Normally, we would use a BigInt or equivalent and interpret a string as a base 128 number:

h(“ALAN”) = $65 \cdot 2^3 + 76 \cdot 2^2 + 65 \cdot 2^1 + 78 \cdot 2^0 = 9811$

Class challenge 1

Interpret the string CLRS as a base 128 number:

Today’s outline

1. Overview
2. Direct addressing
3. Hash tables
4. Hash functions
5. The division method
6. Open addressing
7. Probing

Open addressing

In **open addressing**, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL.

When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.

No lists and no elements are stored outside the table, unlike in chaining (covered section 11.2, and Lecture 10).

Open addressing

In open addressing, the hash table can “fill up” so that no further insertions can be made. One consequence is that the **load factor** α can never exceed 1. The load factor is the ratio of elements to slots.

For a given hash table T: $\alpha = n/m$

Where

- n = number of elements in table T
- m = number of slots in table T

Probing

To perform an insertion using open addressing, we successively examine, or **probe**, the hash table until we find an empty slot.

Instead of being fixed in the order 0, 1, …, m – 1, which requires $\boldsymbol{\Theta}(n)$ search time (chaining worst case, with all elements in same linked list), the sequence of positions probed *depends upon the key being inserted*.

To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a *second input*.

Probing

Thus, the hash function becomes:

$h : U \times \underbrace{\{0, 1, \ldots, m - 1\}}_{\text{Probe number}} \rightarrow \underbrace{\{0, 1, \ldots, m - 1\}}_{\text{Slot number}}$

Reminder: SL#21

Probe sequence

With open addressing, we require that for every key k, the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ be a permutation of $\langle 0,1, \ldots, m-1\rangle$.

This ensures that every hash-table position is eventually filled.

Today’s outline

1. Overview
2. Direct addressing
3. Hash tables
4. Hash functions
5. The division method
6. Open addressing
7. Probing

Linear probing

Given key k, first try the home cell $h(k)$.

If already occupied, examine $h(k) + 1$, $h(k) + 2$, etc., wrapping around the array until an empty cell is found or the whole array examined.

If you like, we're defining a new hash function. The book calls this an **auxiliary hash function**, h’.

Auxiliary hash function

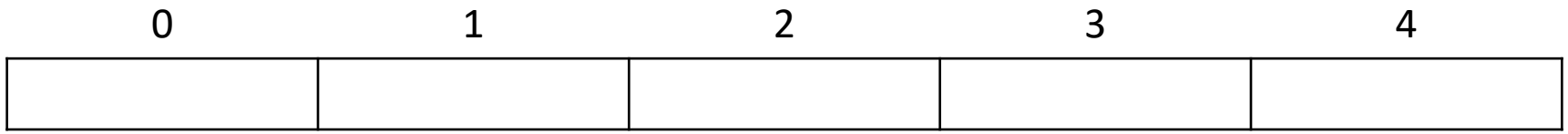
H(k, i) = (h(k) + i) % m

where *k* is the key, *i* is the number of collisions so far, and *m* is the array's capacity. The **home cell** is given by H(k, 0).

Linear probing example

Insert the following set of numbers *n* into a hash table of size 5. We will use the hash function: h(k, i) = (h(k) + i) % m. Here, we will use h(k) = k

Where *k* = key, *i* = collision count, and *m* is size of array.

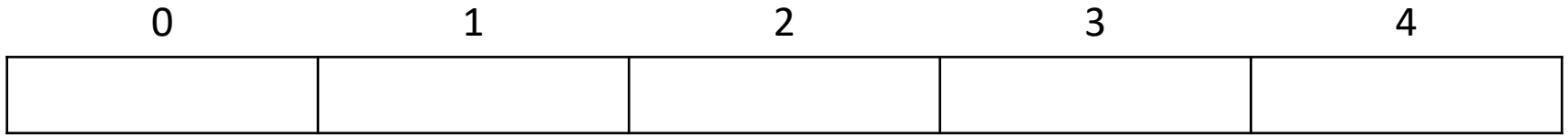


n = {17, 19, 18, 23}

m = 5

Linear probing example

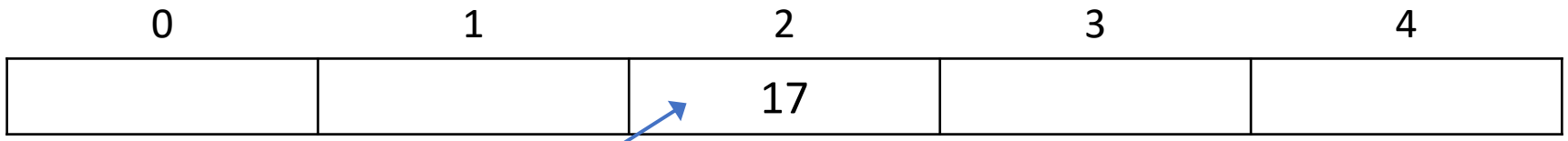
n = {17, 19, 18, 23}, m = 5, using h(k, i) = (h(k) + i) % m



h(k, i) = (17 + 0) % 5 = 2

Linear probing example

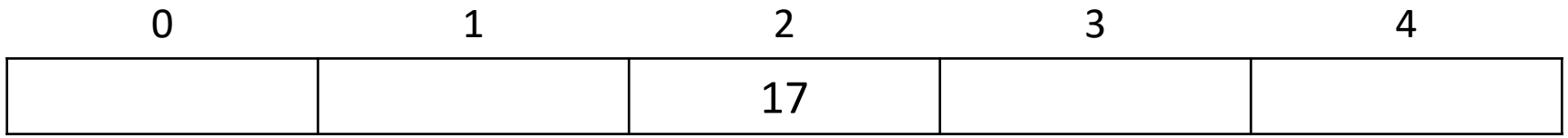
n = {17, 19, 18, 23}, m = 5, using h(k, i) = (h(k) + i) % m



h(k, i) = (17 + 0) % 5 = 2

Linear probing example

n = {~~17~~, 19, 18, 23}, m = 5, using h(k, i) = (h(k) + i) % m

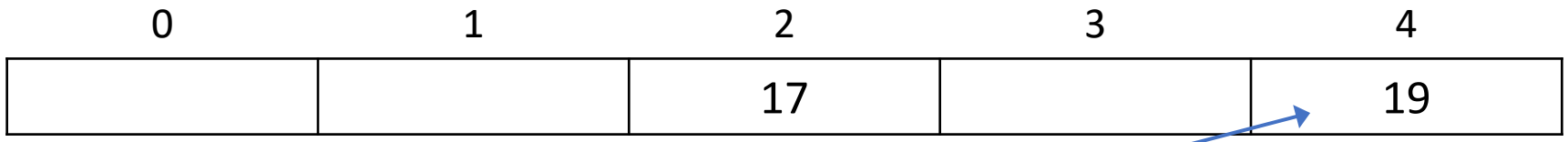


h(k, i) = (17 + 0) % 5 = 2

h(k, i) = (19 + 0) % 5 = 4

Linear probing example

n = {~~17~~, ~~19~~, 18, 23}, m = 5, using h(k, i) = (h(k) + i) % m



h(k, i) = (17 + 0) % 5 = 2

h(k, i) = (19 + 0) % 5 = 4

Linear probing example

n = {~~17~~, ~~19~~, 18, 23}, m = 5, using h(k, i) = (h(k) + i) % m

0	1	2	3	4
		17		19

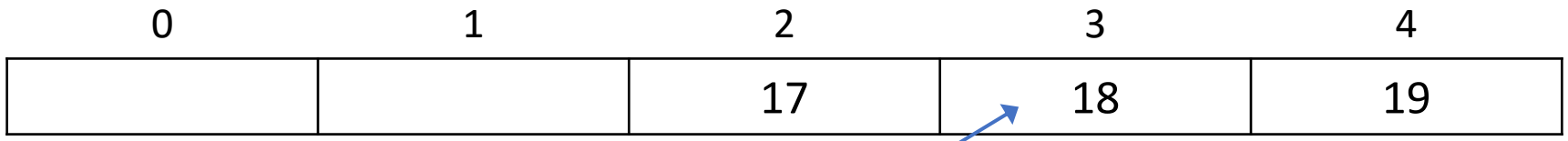
h(k, i) = (17 + 0) % 5 = 2

h(k, i) = (19 + 0) % 5 = 4

h(k, i) = (18 + 0) % 5 = 3

Linear probing example

n = {~~17~~, ~~19~~, ~~18~~, 23}, m = 5, using h(k, i) = (h(k) + i) % m



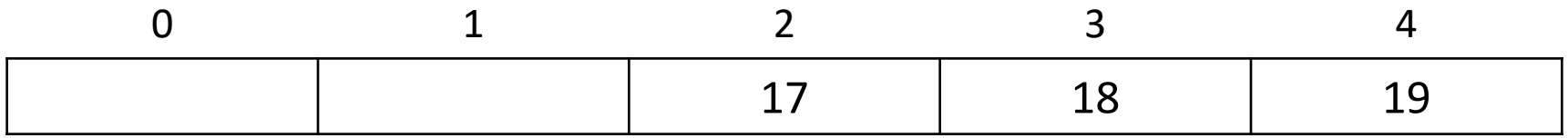
h(k, i) = (17 + 0) % 5 = 2

h(k, i) = (19 + 0) % 5 = 4

h(k, i) = (18 + 0) % 5 = 3

Linear probing example

n = {~~17~~, ~~19~~, ~~18~~, 23}, m = 5, using h(k, i) = (h(k) + i) % m



h(k, i) = (17 + 0) % 5 = 2

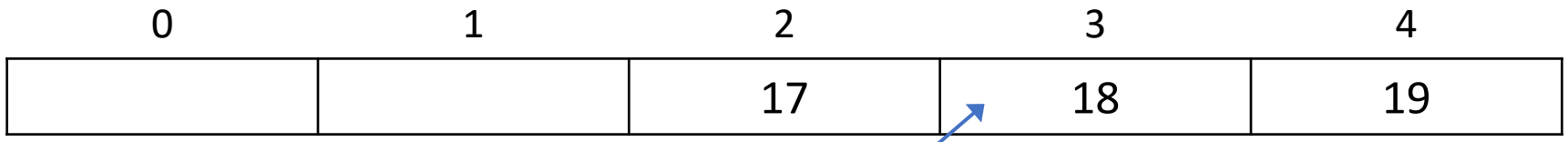
h(k, i) = (19 + 0) % 5 = 4

h(k, i) = (18 + 0) % 5 = 3

h(k, i) = (23 + 0) % 5 = 3

Linear probing example

n = {~~17~~, ~~19~~, ~~18~~, 23}, m = 5, using h(k, i) = (h(k) + i) % m



h(k, i) = (17 + 0) % 5 = 2

h(k, i) = (19 + 0) % 5 = 4

h(k, i) = (18 + 0) % 5 = 3

h(k, i) = (23 + 0) % 5 = 3

X collision 1

Linear probing example

n = {~~17~~, ~~19~~, ~~18~~, 23}, m = 5, using h(k, i) = (h(k) + i) % m

0	1	2	3	4
		17	18	19

h(k, i) = (17 + 0) % 5 = 2

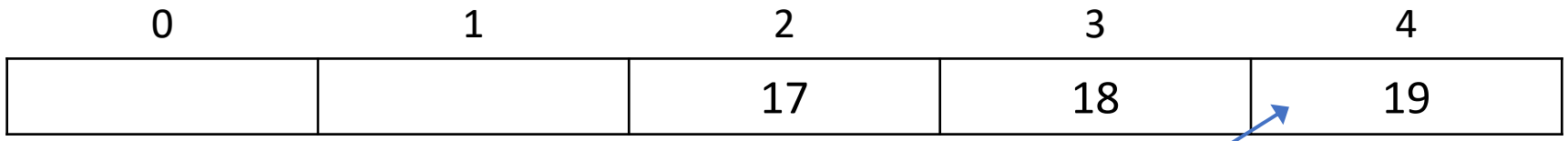
h(k, i) = (19 + 0) % 5 = 4

h(k, i) = (18 + 0) % 5 = 3

h(k, i) = (23 + 0) % 5 = 3, set i += 1
h(k, i) = (23 + 1) % 5 = 4

Linear probing example

n = {~~17~~, ~~19~~, ~~18~~, 23}, m = 5, using h(k, i) = (h(k) + i) % m



h(k, i) = (17 + 0) % 5 = 2

h(k, i) = (19 + 0) % 5 = 4

h(k, i) = (18 + 0) % 5 = 3

h(k, i) = (23 + 0) % 5 = 3, set i += 1

h(k, i) = (23 + 1) % 5 = 4

X collision 2

Linear probing example

n = {~~17~~, ~~19~~, ~~18~~, 23}, m = 5, using h(k, i) = (h(k) + i) % m

0	1	2	3	4
		17	18	19

h(k, i) = (17 + 0) % 5 = 2

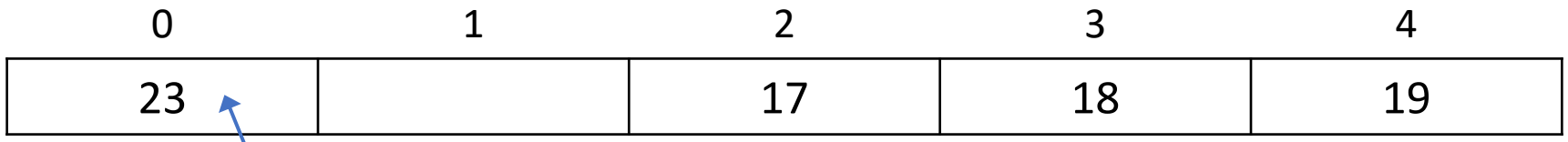
h(k, i) = (19 + 0) % 5 = 4

h(k, i) = (18 + 0) % 5 = 3

h(k, i) = (23 + 0) % 5 = 3, set i += 1
h(k, i) = (23 + 1) % 5 = 4, set i += 1
h(k, i) = (23 + 2) % 5 = 0

Linear probing example

n = {~~17~~, ~~19~~, ~~18~~, ~~23~~}, m = 5, using h(k, i) = (h(k) + i) % m



h(k, i) = (17 + 0) % 5 = 2

h(k, i) = (19 + 0) % 5 = 4

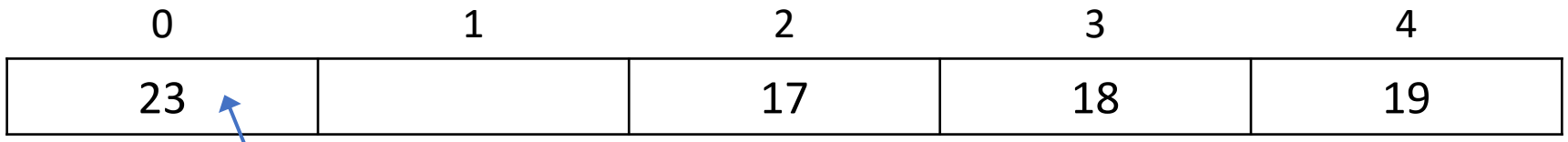
h(k, i) = (18 + 0) % 5 = 3

h(k, i) = (23 + 0) % 5 = 3, set i += 1
h(k, i) = (23 + 1) % 5 = 4, set i += 1
h(k, i) = (23 + 2) % 5 = 0

Done!

Linear probing example

n = {~~17~~, ~~19~~, ~~18~~, ~~23~~}, m = 5, using h(k, i) = (h(k) + i) % m



h(k, i) = (17 + 0) % 5 = 2

h(k, i) = (19 + 0) % 5 = 4

h(k, i) = (18 + 0) % 5 = 3

h(k, i) = (23 + 0) % 5 = 3, set i += 1
h(k, i) = (23 + 1) % 5 = 4, set i += 1
h(k, i) = (23 + 2) % 5 = 0

Primary clustering

Done!

Limitation

Linear probing suffers from **primary clustering**.

We get clusters because the steps are small and all the same size and all ignore the key.

Retrieving then requires sequential search through the cluster. Also, clusters tend to coalesce (bad news).

But if we have few collisions, then clusters stay small and linear probing is good enough.

Quadratic probing

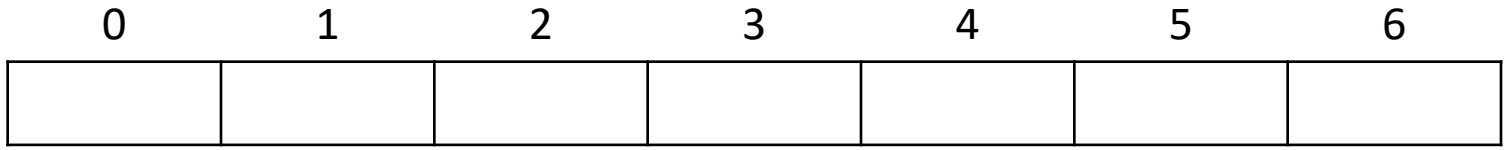
An old idea to avoid clustering is to make the step size quadratic (gets bigger with each step), e.g. use the new hash function:

$H(k, i) = (h(k) + i^2) \% m$

where *h* is the original hash function, and *i* is the number of collisions so far (drawbacks: secondary clustering and wasted space).

Quadratic probing example

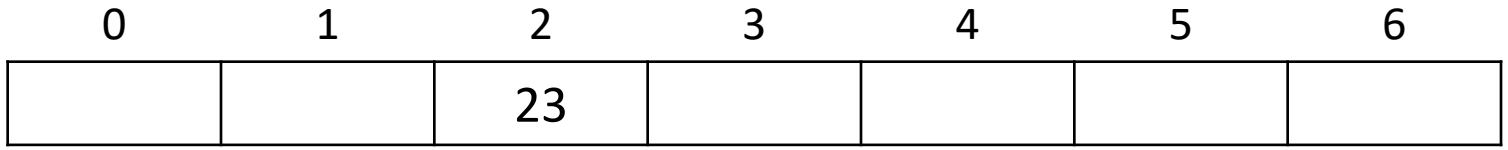
n = {23, 8, 9, 81, 83, 15}, m = 7, using $h(k, i) = (h(k) + i^2) \% m$



h(k, i) = (23 + 0) % 7 = 2

Quadratic probing example

n = {~~23~~, 8, 9, 81, 83, 15}, m = 7, using $h(k, i) = (h(k) + i^2) \% m$



h(k, i) = (23 + 0) % 7 = 2

h(k, i) = (8 + 0) % 7 = 1

Quadratic probing example

n = {~~23~~, ~~8~~, 9, 81, 83, 15}, m = 7, using $h(k, i) = (h(k) + i^2) \% m$

0	1	2	3	4	5	6
	8	23				

h(k, i) = (23 + 0) % 7 = 2

h(k, i) = (8 + 0) % 7 = 1

Quadratic probing example

n = {~~23~~, ~~8~~, 9, 81, 83, 15}, m = 7, using $h(k, i) = (h(k) + i^2) \% m$

0	1	2	3	4	5	6
	8	23				

$h(k, i) = (23 + 0) \% 7 = 2$

$h(k, i) = (8 + 0) \% 7 = 1$

$h(k, i) = (9 + 0) \% 7 = 2$

$h(k, i) = (9 + 1^2) \% 7 = 3$

Quadratic probing example

n = {~~23~~, ~~8~~, ~~9~~, 81, 83, 15}, m = 7, using $h(k, i) = (h(k) + i^2) \% m$

0	1	2	3	4	5	6
	8	23	9			

$h(k, i) = (23 + 0) \% 7 = 2$

$h(k, i) = (8 + 0) \% 7 = 1$

$h(k, i) = (9 + 0) \% 7 = 2$

$h(k, i) = (9 + 1^2) \% 7 = 3$

$h(k, i) = (81 + 0) \% 7 = 4$

Quadratic probing example

n = {~~23~~, ~~8~~, ~~9~~, ~~81~~, 83, 15}, m = 7, using $h(k, i) = (h(k) + i^2) \% m$

0	1	2	3	4	5	6
	8	23	9	81		

$h(k, i) = (23 + 0) \% 7 = 2$

$h(k, i) = (8 + 0) \% 7 = 1$

$h(k, i) = (9 + 0) \% 7 = 2$

$h(k, i) = (9 + 1^2) \% 7 = 3$

$h(k, i) = (81 + 0) \% 7 = 4$

Quadratic probing example

n = {~~23~~, ~~8~~, ~~9~~, ~~81~~, 83, 15}, m = 7, using $h(k, i) = (h(k) + i^2) \% m$

0	1	2	3	4	5	6
	8	23	9	81		83

$h(k, i) = (23 + 0) \% 7 = 2$

$h(k, i) = (8 + 0) \% 7 = 1$

$h(k, i) = (9 + 0) \% 7 = 2$

$h(k, i) = (9 + 1^2) \% 7 = 3$

$h(k, i) = (81 + 0) \% 7 = 4$

$h(k, i) = (83 + 0) \% 7 = 6$

Quadratic probing example

n = {~~23~~, ~~8~~, ~~9~~, ~~81~~, ~~83~~, ~~15~~}, m = 7, using $h(k, i) = (h(k) + i^2) \% m$

0	1	2	3	4	5	6
	8	23	9	81	15	83

$h(k, i) = (23 + 0) \% 7 = 2$

$h(k, i) = (8 + 0) \% 7 = 1$

$h(k, i) = (9 + 0) \% 7 = 2$

$h(k, i) = (9 + 1^2) \% 7 = 3$

$h(k, i) = (81 + 0) \% 7 = 4$

$h(k, i) = (83 + 0) \% 7 = 6$

$h(k, i) = (15 + 0) \% 7 = 1$

$h(k, i) = (15 + 1^2) \% 7 = 2$

$h(k, i) = (15 + 2^2) \% 7 = 5$

Done!

Quadratic probing example

n = {~~23~~, ~~8~~, ~~9~~, ~~81~~, ~~83~~, ~~15~~}, m = 7, using $h(k, i) = (h(k) + i^2) \% m$

0	1	2	3	4	5	6
	8	23	9	81	15	83

$h(k, i) = (23 + 0) \% 7 = 2$

$h(k, i) = (8 + 0) \% 7 = 1$

$h(k, i) = (9 + 0) \% 7 = 2$

$h(k, i) = (9 + 1^2) \% 7 = 2$

$h(k, i) = (81 + 0) \% 7 = 4$

$h(k, i) = (83 + 0) \% 7 = 6$

$h(k, i) = (15 + 0) \% 7 = 1$

$h(k, i) = (15 + 1^2) \% 7 = 2$

$h(k, i) = (15 + 2^2) \% 7 = 5$

Secondary clustering

Done!

Something to think about

What happens with the following table, keys using quadratic probing?

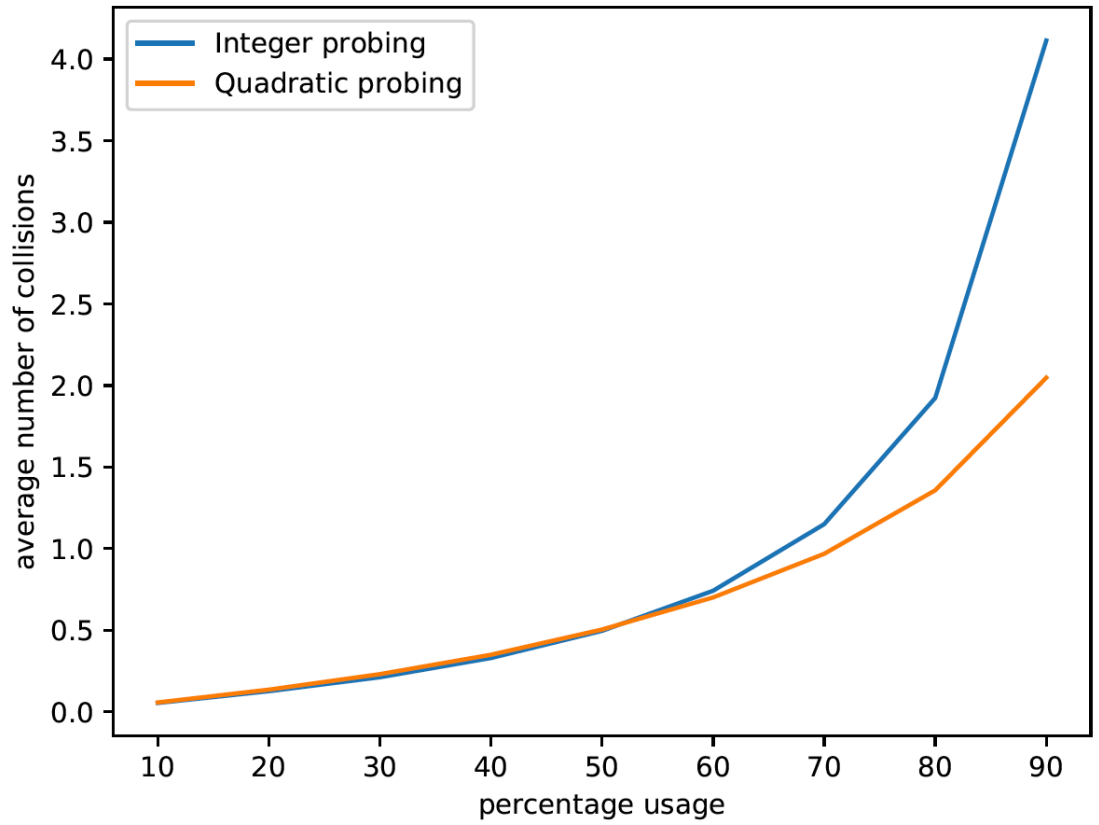
0	1	2	3	4	5	6	7

n = {23, 19, 40, 47, 27}

Try it home as a challenge.

Linear vs Quadratic probing

Proofs are good, but sometimes experiments are good enough: table size is 997; random numbers from 0 to 1000000; 0 to 90% occupancy; repeat 1000 times; how many collisions on average?



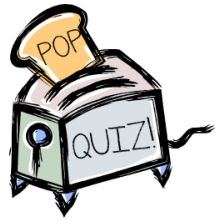
Suggested reading

Hash functions are discussed in section 11.3, with division hashing in 11.3.1.

Linear and quadratic probing are discussed in section 11.4 (called open addressing).

Solutions

Pop quiz 1

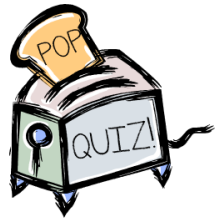


If m = 12, and k = 100, what will our hash function yield?

Answer

100 % 12 = 4

Pop quiz 2



If m = 20, and k = 91, what will our hash function yield?

Answer

91 % 20 = 11

Class challenge 1

Interpret the string CLRS as a base 128 number:

ASCII values: C = 67, L = 76, R = 82, S = 83.

There are 128 basic ASCII values.

So interpret CLRS as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0)$ = 141,766,947

Image attributions