COMP1002
DATA STRUCTURES AND ALGORITHMS

LECTURE 6: HASH TABLES

Curtin University
Discipline of Computing

Last updated: September 15, 2019

This Week

- Hashtables and hashing
- Properties of a good hashtable
- Collisions and collision handling methods
  - Open addressing (probing) approaches
  - Separate chaining
- Hashtable Big-O analysis
- Some good hash functions and poor hash functions

Hash Tables – Hashing

- A hash table stores each data element using an associated key
  - The key is later used to find the element efficiently
  - Hash tables convert the key into an index via an arithmetic function and then place the data at this index
  - This conversion is referred to as “hashing”: applying an arithmetic function to a key to map it to a location (index) in an array for storing the data associated with that key
  - The arithmetic function is called the hashing function
  - The location it maps a key to is called the hash index

A Simple Example

- Simple hash function: add up the letters in a string
  - How is this possible? Because chars are actually numbers
    - The application just reinterprets them as symbols for the user
  - Char values are encoded according to Unicode standard
    - ASCII is a subset of Unicode for the European alphabets
      - Unicode is actually an extension of ASCII in order to handle non-Latin languages such as Mandarin, Japanese, Arabic, etc
    - So let's have a look at the ASCII character set…

ASCII Character Set

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01		SOH
^B	2	02		STX
^C	3	03		ETX
^D	4	04		EOT
^E	5	05		ENQ
^F	6	06		ACK
^G	7	07		BEL
^H	8	08		BS
^I	9	09		HT
^J	10	0A		LF
^K	11	0B		VT
^L	12	0C		FF
^M	13	0D		CR
^N	14	0E		SO
^O	15	0F		SI
^P	16	10		DLE
^Q	17	11		DC1
^R	18	12		DC2
^S	19	13		DC3
^T	20	14		DC4
^U	21	15		NAK
^V	22	16		SYN
^W	23	17		ETB
^X	24	18		CAN
^Y	25	19		EM
^Z	26	1A		SUB
^[	27	1B		ESC
^\	28	1C		FS
^]	29	1D		GS
^^	30	1E	▲	RS
^-	31	1F	▼	US

Dec	Hex	Char
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	░
177	B1	▒
178	B2	▓
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Dec	Hex	Char
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	█
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	

ASCII Character Set

- So each letter can be converted into a number…
  - Range of ASCII values is 0-255
  - Each character can then be represented using a byte = 2^8
- Examples:
  - "**0**" (zero) is 48 in decimal ← numbers are all less than letters
  - "**9**" is 57
  - "**A**" is 65 "Z" is 90
  - "**a**" is 97 ← this is how "apple" is less than "bear" and "zombie"
  - "**z**" is 122
  - "**!**" is 33
- Hex values are more interesting:
  - 0011 0001 is "**1**" – clear bits 4-5 to convert from character to num (0000 0001)
  - 0100 0001 is "**A**" and 0110 0001 is "**a**" – set or clear bit 5 to convert

ASCII Character Set

- Notice the blank chars early on…
  - These are special ‘control’ chars, some of which have been defined as special ‘escape characters’ \x in Java/Python/C/C++
  - Examples:
    - 10 is LF (line feed), which is ‘\n’ in
    - 13 is CR (carriage return), which is ‘\r’
      - Under Linux, end-of-line is \n. Under Windows, eol is \r\n
    - 0 is NULL which is ‘\0’
      - Particularly important in C for null-terminated strings
    - 9 is HT (horizontal tab) == ‘\t’

  - Not all these control chars have escape chars
    - These are typically unprintable characters (don’t show up) or produce odd symbols depending on the display program
    - *e.g.* Notepad will show a □

A Simple Example

- Back to the example: hash function for string keys

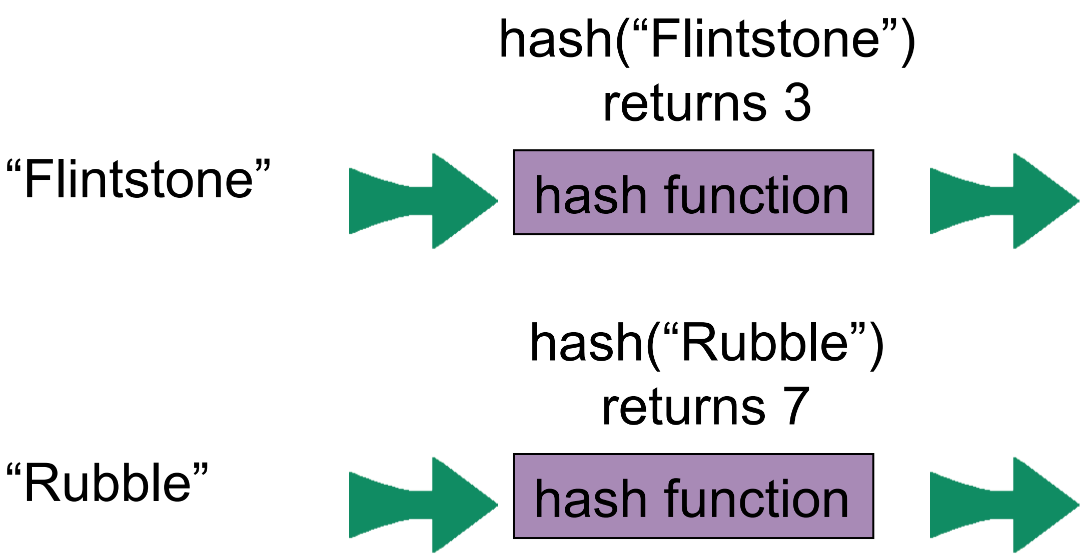
```
FUNCTION hashFunction IMPORT key (string) EXPORT hashIndex (integer)
   hashIndex ← 0
   FOR ii ← 0 TO len(key)-1 DO
      hashindex ← hashindex + key.charAt(ii)

   hashIndex ← hashindex % hashArray.length      // We'll discuss this line later
```

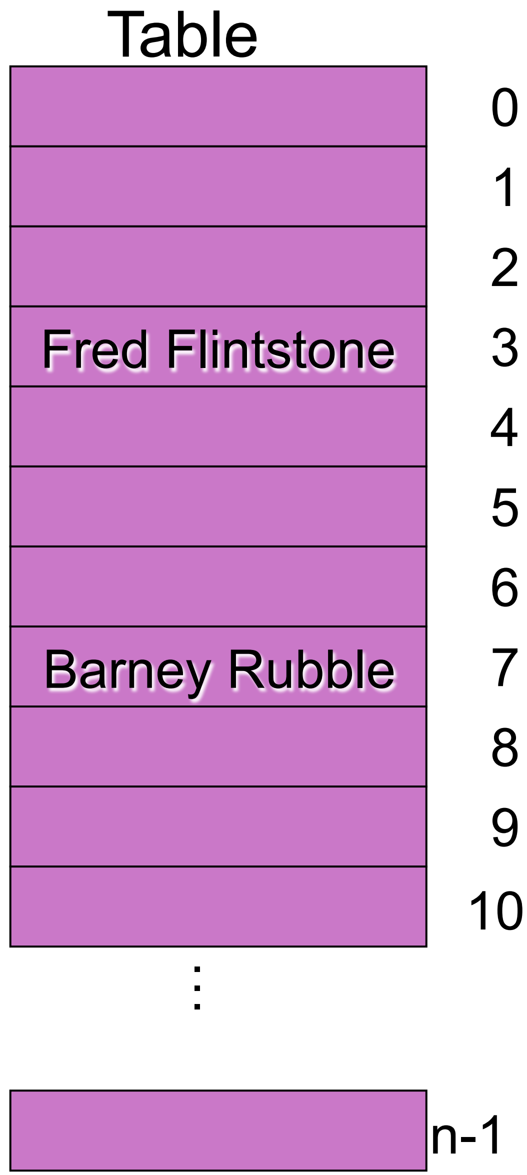
- Java
  - key.charAt(ii) returns the char at index ii (zero-based)
  - Since chars are numbers, we can add them to an int arithmetically.
  - So "abcde" would be:
    - hashIdx = 'a'+'b'+'c'+'d'+'e'; // =495
- Python
  - Use ord() function to get ASCII values
    - hashindex = hashindex + ord(key[ii])

Hashing

- So a hash table has two major components:
  - Array: (table) to store the data
  - Hash function: to map keys to integer indexes in the array
- When a new element is to be added, both a key and data must be provided to the hash table
  - The hash table hashes the key and stores both the key and data at the calculated hash index
    - Key must be unique
  - Thus time complexity is the time it takes to perform the hash calculation O(?) plus the O(1) array access time
    - Accessing data in the hash table is the same

Hash Table – Example





Keyed on last name

Hash function converts each key to a (hopefully different!) hash index

HASH FUNCTIONS

Properties of a *Good* Hash Function

- A good hash function should:
  1. Return indexes that fit within the size of the array
     - *i.e.,* [0 .. arrayLength-1]
  2. Be fast to compute
     - The hash function is a critical factor in access time
  3. Be repeatable (*i.e.,* always return same index) for a given key
  4. Distribute keys evenly over the full range of the array
     - This is to minimise collisions, a major issue in hash tables
- Properties 1–3 are easy to ensure, Property 4 is not
  - You don’t know what keys you’ll be getting in advance, so it’s not possible to ensure they will be evenly distributed

Property 1: Hash Indexes Must Fit Table

- Let's consider a simple hash function that operates on integer keys (*e.g.*, student IDs)
  - A bad implementation would be:

```
CLASS HashTable
Class field : hashArray

FUNCTION hashFunction IMPORT key (integer) EXPORT hashIndex (integer)
hashIndex ← key
```

- What's the problem with it?
  - Key can be larger than table.length or possibly negative, this will make hashIndex out of the table's index range.

Property 1: Hash Indexes Must Fit Table

- So why not make the table big enough to handle any integer key that we expect from the application?
  - That's a waste of space: there are far fewer actual data items than the number of possible key values
    - *e.g.,* Student IDs are large numbers that track *all* students ever (~15 million), but there are only ~50,000 current students
    - We thus only want to reserve space for 50,000, not all 15m
- What the hash function must do is 'compress' the range of the key down into the range of the table
  - Easy to do! Just take the modulus of the hashIndex
    - Although there are better and worse modulus values…

Property 1: Hash Indexes Must Fit Table

- The obvious modulus is the table's length, since any integer's remainder will then fit inside the table

```
FUNCTION hashFunction IMPORT key (integer) EXPORT hashIndex (integer)
    hashIndex ← key % table.length
```

- Prime numbers often make good modulo values
  - Primes will usually distribute the keys most evenly
    - … although it does depend somewhat on the hash function used
  - So when allocating the table, size it with a prime number
    - ideal prime sizes are as far as possible from powers of 2
      - 769, 1543, 3079, 6151, 12289, 24593, 49157, 98317 *etc.*
    - If the user is allowed to specify the max size of the hash table in the constructor, round the max up to the nearest prime number
      - Finding the next prime can be done by incrementing a starting candidate value and testing for primality – an $O(N^{½})$ problem

Finding the Next Prime

```
FUNCTION findNextPrime IMPORT startVal (integer) EXPORT prime (integer)

IF (startVal % 2 == 0) THEN
   primeVal ← startVal + 1            ← Even numbers are never prime, so make it odd
ELSE
  primeVal ← startVal
ENDIF
primeVal ← primeVal - 2               ← Pre-decrement since we will be looking for the NEXT prime,
                                        but want to include startVal as a candidate

isPrime ← FALSE
WHILE (NOT isPrime)                   ← There is always a prime number to be found
   primeVal ← primeVal + 2            ← Next candidate

   // Test if primeVal candidate is actually a prime number
   ii ← 3
   isPrime = TRUE
   WHILE (ii*ii <= primeVal) AND (isPrime)     ← No need to check beyond sqrt(primeVal)
      IF (primeVal % ii == 0) THEN               (if the values up to ii are not whole divisors, the final
         isPrime ← FALSE                          (ii-1)/primeVal percentile won't be divisors either)
      ELSE
         ii ← ii + 2                             ← Skip testing with even numbers
      ENDIF
   ENDWHILE
ENDWHILE
```

Finding the Next Prime (improved)

```
FUNCTION findNextPrime IMPORT startVal (integer) EXPORT prime (integer)

IF (startVal % 2 == 0) THEN
   primeVal ← startVal -1            ← Even numbers are never prime, so make it odd 1 less as primeval+2
ELSE
  primeVal ← startVal
ENDIF

isPrime ← FALSE
DO                                // Test if primeVal candidate is actually a prime number
   primeVal ← primeVal + 2                       ← Next candidate
   ii ← 3
   isPrime = TRUE
   rootVal = SQRT primeVal                       ← No need to check beyond sqrt(primeVal)
   DO
      IF (primeVal % ii == 0) THEN                (if the values up to ii are not whole divisors, the final
         isPrime ← FALSE                           (ii-1)/primeVal percentile won't be divisors either)
      ELSE
         ii ← ii + 2                             ← Skip testing with even numbers
      ENDIF
   WHILE (ii <= rootVal) AND (isPrime)
WHILE (NOT isPrime)                              ← There is always a prime number to be found
```

Property 2: Fast to Compute

- In general, access time in a hash table is:
  - (speed of hash function) + (speed of array access)
    - Array access is O(1), so the hash function is the limiting factor
- Fortunately, hash functions can be pretty fast
  - The hash function only needs to operate on the key
  - Thus it will take constant time regardless of how many elements exist in the hash table (exact time depends on hash func)
    - Constant time complexity: O(k) → ≈ O(1)
  - *e.g.,* consider string keys (such as LastName)
    - Time complexity = O(L) ≈ O(1), where L = average name length
    - Independent of table size N, hence stays O(1) regardless of N

Property 3: Repeatable

- A hash function is useless if it returns different hash indexes every time for the same key
  - It won't be able to remember where it stored keys/values!

- Solution: Avoid using time-varying values in the calculation of the hash function
  - *e.g.,* no random numbers, date/time, or hash table statistics
- Issue: Table size must be used as the modulo!
  - Resizing the table means hash function modulo changes
  - Thus you must re-hash *all existing entries* after a resize
    - *i.e.,* rebuild the hash table from scratch, re-inserting each existing element again into the new hash table

Property 4: Distributes Keys Evenly

- The toughest property to achieve
  - It is very difficult to even *evaluate* that a hash function distributes evenly
    - Need to perform statistical tests with a large set of sample keys
  - The problem is that you don't quite know what keys you will be receiving from an application
    - You can often see (in the code) if a hash function will be poor, but guaranteeing a hash function will be good is more difficult
  - Fortunately, much effort has already gone into developing hash functions that work well for a range of typical keys
    - *e.g.,* dictionary words, names, *etc.*

Collisions

- Let's take a little detour and discuss some issues affecting hash tables
  - This will help with understanding how Property 4 (Evenly Distributed) can be best achieved
    - … or at least in realising when it *isn't* achieved!
- A collision is when two (or more) keys map to the same hash index
  - Not a good thing, but must be expected to occur at least sometimes since the hash function 'compresses' the range of the key into the range of the hash table size
    - Unless you are lucky, some keys will hash to the same index

Collisions

- Since collisions are inevitable, we must handle them
- **First**: Never fill up the hashtable
  - The fuller the table, the more likely a collision
  - Try to avoid going over about 50% full (rough rule of thumb)
    - *i.e.,* allocate about twice as much space as you expect to use
    - But don’t go too sparsely-populated (< 10%) – waste of space!
- **Second**: Use a collision-handling method
  - There are four typical approaches, split into two broad categories: open addressing and separate chaining

Collision Handling Methods

- Open Addressing: Upon a collision, jump forward (‘probe’) a set amount to a new index and try again
  - If the new index is also used, repeat the probe until an empty index is found

- 1. Linear Probing – probe by step size of one every time
- 2. Quadratic Probing – probe forward by $(\text{probeNum})^2$
  - *i.e.,* probe first by 1, then 4, then 9, 16, 25, 36, 49, …
- 3. Double Hashing – use a *secondary* (different) hash function on the key to generate the probe step length

- 4. Separate Chaining: Key-value pairs are added to a linked list anchored at the colliding hash index

Property 4: Distributes Keys Evenly

- Why do we need even (*i.e.,* uniform) distribution?
  - Consider the opposite case: all keys hash to index 0
  - This means all keys end up at the same position
    - *i.e.,* every key collides – worst case!
    - Inserting the N^{th} item takes O(N) probes (for linear/quad probing)
      - Sep-chaining is O(1), double-hashing will vary but won't be good
    - Accessing an item takes (on average) N/2 probes = O(N)
      - Even for separate chaining and (probably) double-hashing
    - Terrible: we should be getting O(1) from our hash table on both!
  - Uniform = each key maps to a different index

Example Hash Functions

- Let’s take a look at a couple of hash functions proposed for use in hashtables
  - Only given in Java and here for your interest – not examinable!

  - These operate on either Strings or byte arrays so that they can apply to any type of data
    - A string can be considered a byte array that is limited to text values (alphanumerics, punctuation and symbols)
  - They are by no means ‘optimal’, since optimality will depend on your actual keys being used
    - And nobody can predict that in advance!
  - Still, they are pretty good at distributing ‘average’ keys evenly amongst the table
    - Code assumes that hashArray is the hash table array

Example Hash Functions

- Java’s hashCode() implementation for String class

```
private int hash(String key)
{
   int hashIdx = 0;

   for (int ii = 0; ii < key.length(); ii++) {
      hashIdx = (31 * hashIdx) + key.charAt(ii);
   }
   return hashIdx % hashArray.length;
}
```

  - Strange (often prime) numbers like 31 are popular
  - A variant is the ‘Bernstein’ hash function
    - Uses 33 instead of 31 – apparently this is even more effective in practice - hash functions are black magic!

Example Hash Functions

- Another fairly simple but good hash function

```
private int hash(byte[] key)
{
   int a = 63689;
   int b = 378551;
   int hashIdx = 0;

   for (int ii = 0; ii < key.length; ii++)
   {
      hashIdx = (hashIdx * a) + key[ii];
      a *= b;
   }
   return hashIdx % hashArray.length;
}
```

  - Non-commutative multiplications by weird numbers a and b are apparently important here
    - Again, only borne out in practice

Example Hash Functions

- FNV Hash

```
private int hash(byte[] key)
{
   long hashIdx = 2166136261;

   for (long ii = 0; ii < key.length; ii++) {
      hashIdx = (hashIdx * 16777619) ^ key[ii];        ← ^ is the XOR operator
   }
   return (int)(hashIdx % hashArray.length);
}
```

  - More complex and does a lot of integer overflow
    - It's not the complexity that makes it better, but in practice this is a pretty good hash
    - See http://www.isthe.com/chongo/tech/comp/fnv/

Example Hash Functions

- Shift-Add-XOR Hash

```
private int hash(byte[] key)
{
   int hashIdx = 0;
   for (int ii = 0; ii < len; ii++) {
      hashIdx = hashIdx ^ ( (hashIdx << 5) + (hashIdx << 2) + key[ii] );
   }
   return hashIdx % hashArray.length;
}
```

- Bit shifting is also pretty common in good hash functions
  - It’s a fast way to take a number to a power-of-2
  - But it’s not the bit-shifting itself that makes it a good hash!

Example Hash Functions

- ELF Hash

```
private int hash(byte[] key)
{
   int hashIdx = 0;
   int g;
   for (int ii = 0; ii < len; ii++) {
      hashIdx = (hashIdx << 4) + key[ii];
      g = hashIdx & 0xF0000000L;
      if (g != 0) {
         hashIdx = hashIdx ^ (g >> 24);            ← ^ is the XOR operator
      }
      hashIdx = hashIdx & ~g;
   }
   return hashIdx % hashArray.length;
}
```

  - Another bit-shifter: quite old, and reasonably good too

Poor Hash Functions

- Take-the-first-value:

```
private int hash(byte[] key)
{
   hashIdx = key[0];
   return hashIdx % hashArray.length;
}
```

  - Very poor:
    - Collisions with keys that start with the same value
    - Hash indexes will also only be between 0…255
      - Means slots past 255 can never be used

Poor Hash Functions

- Add-them-all-up:

```
private int hash(byte[] key)
{
   int hashIdx = 0;
   for (int ii = 0; ii < len; ii++) {
      hashIdx += key[ii];
   }
   return hashIdx % hashArray.length;
}
```

  - Poor:
    - Collisions when two keys are an anagram of one another
      - Eg: “fred”, “derf”, “rfde” all hash to the same index
    - Changing the add to a multiply doesn’t help any either – anagrams still produce the same hash index

Poor Hash Functions

- XOR-them-all-together:

```
private int hash(byte[] key)
{
   hashIdx = key[0];
   for (int ii = 0; ii < len; ii++) {
      hashIdx = hashIdx ^ key[ii];
   }
   return hashIdx % hashArray.length;
}
```

← ^ is the XOR operator

- Poor:
  - Hash indexes will also only be between 0…255
    - Means slots past 255 can never be used

COLLISION HANDLING

Linear Probing

- Upon a collision, add 1 to the index and try again
  - Repeatedly probe by 1 until a free slot (bucket) is found
  - Remember to wrap-around the array!
    - *i.e.,* if hash index == tableLen-1, then stepping by 1 will exceed the table size. In this case, we must 'wrap' and probe to index 0
- Linear probing is simple to implement, but has some unfortunate consequences if collisions build up
  - See the next slides for an example

Linear Probing – Example

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key (integer)
                      EXPORT hashIndex (integer)
  hashIndex ← key % 11
```

  - Keys: 15

Table

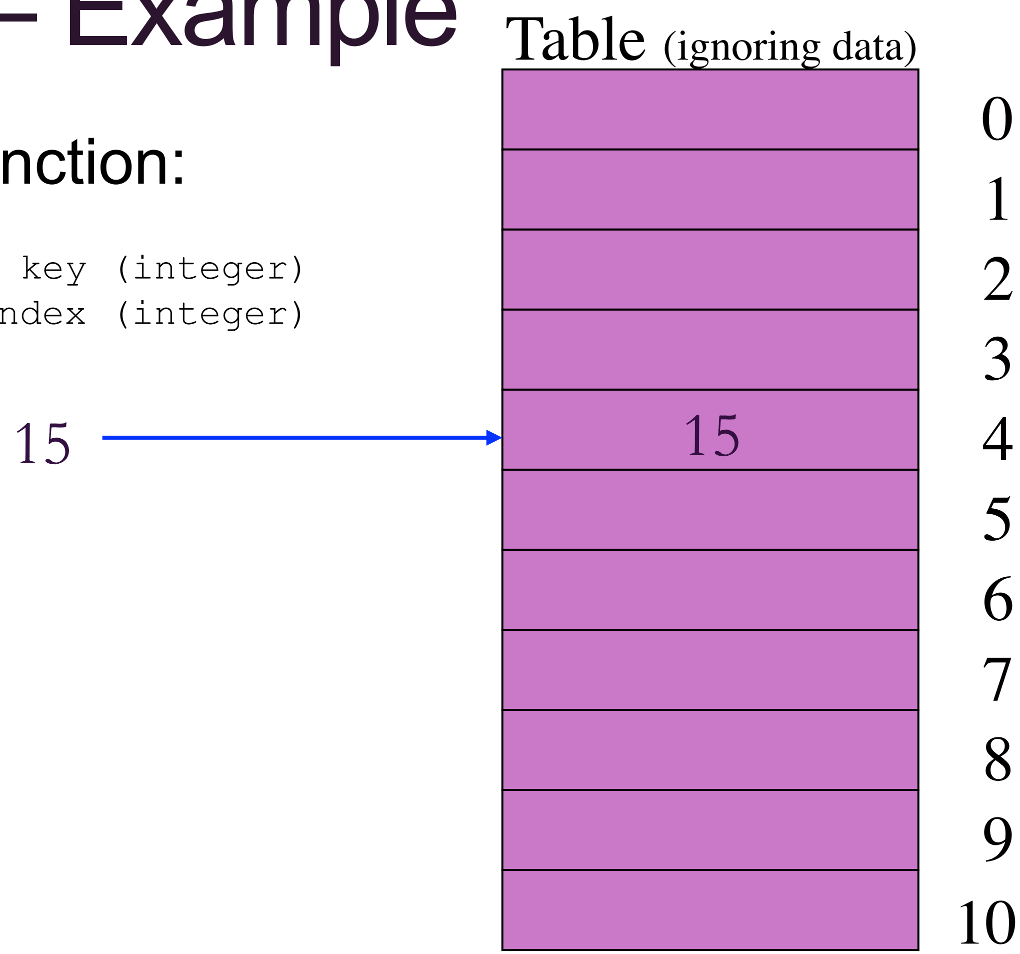
Keys	Data	
		0
		1
		2
		3
15	<data>	4
		5
		6
		7
		8
		9
		10

Linear Probing – Example

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key (integer)
                      EXPORT hashIndex (integer)
  hashIndex ← key % 11
```

  - Keys:

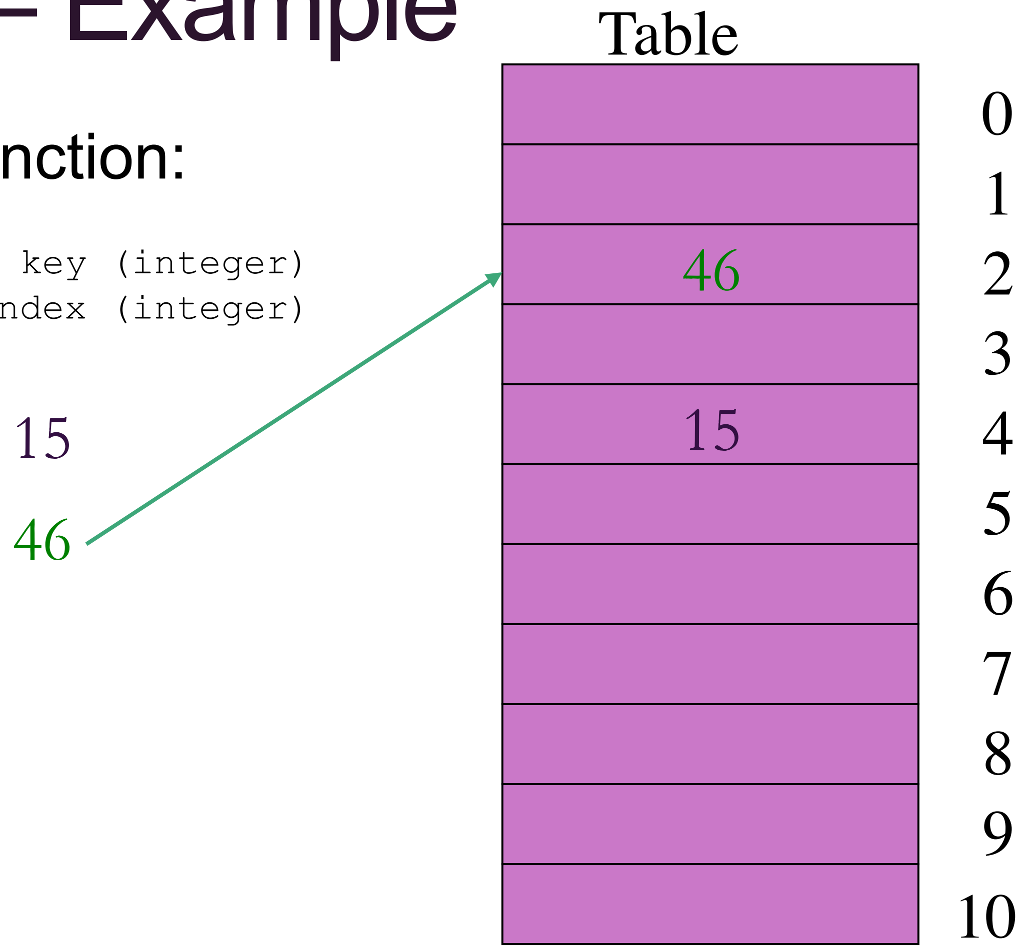


Linear Probing – Example

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key (integer)
                      EXPORT hashIndex (integer)
  hashIndex ← key % 11
```

  - Keys:

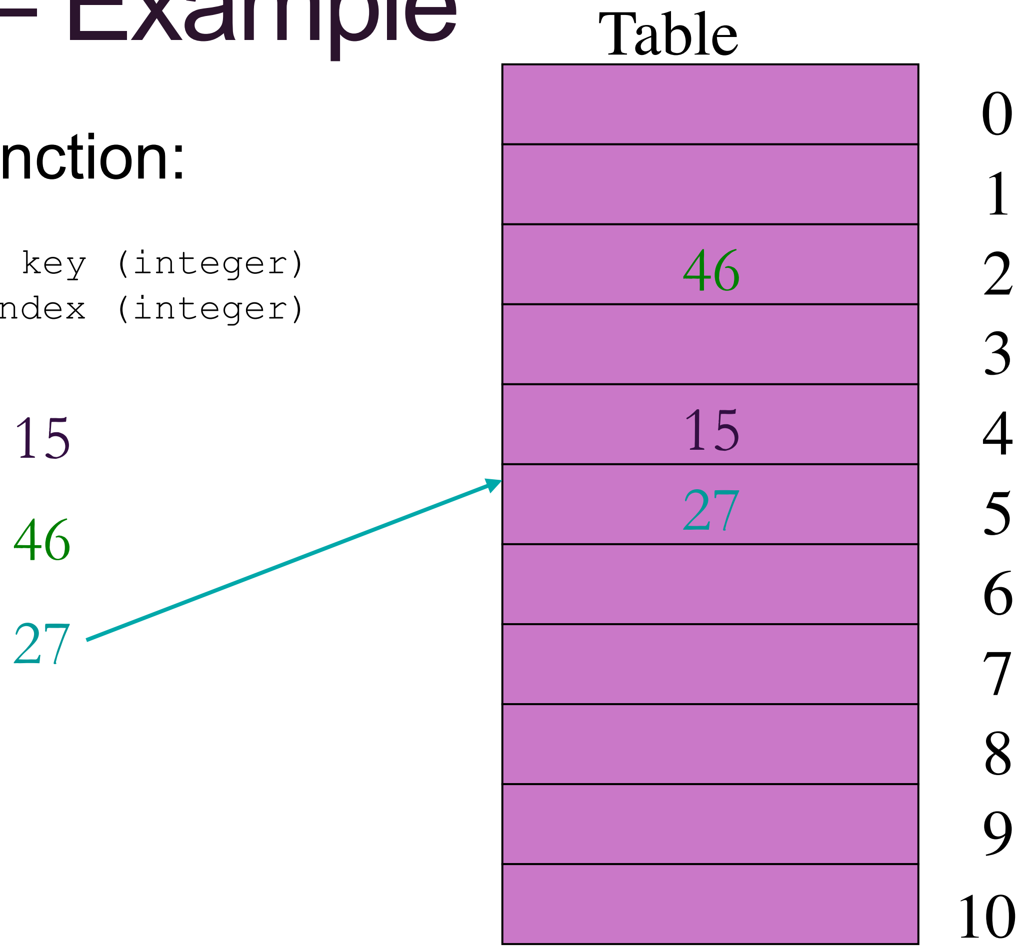


Linear Probing – Example

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key (integer)
                      EXPORT hashIndex (integer)
  hashIndex ← key % 11
```

  - Keys:



Linear Probing – Example

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key (integer)
                  EXPORT hashIndex (integer)
  hashIndex ← key % 11
```

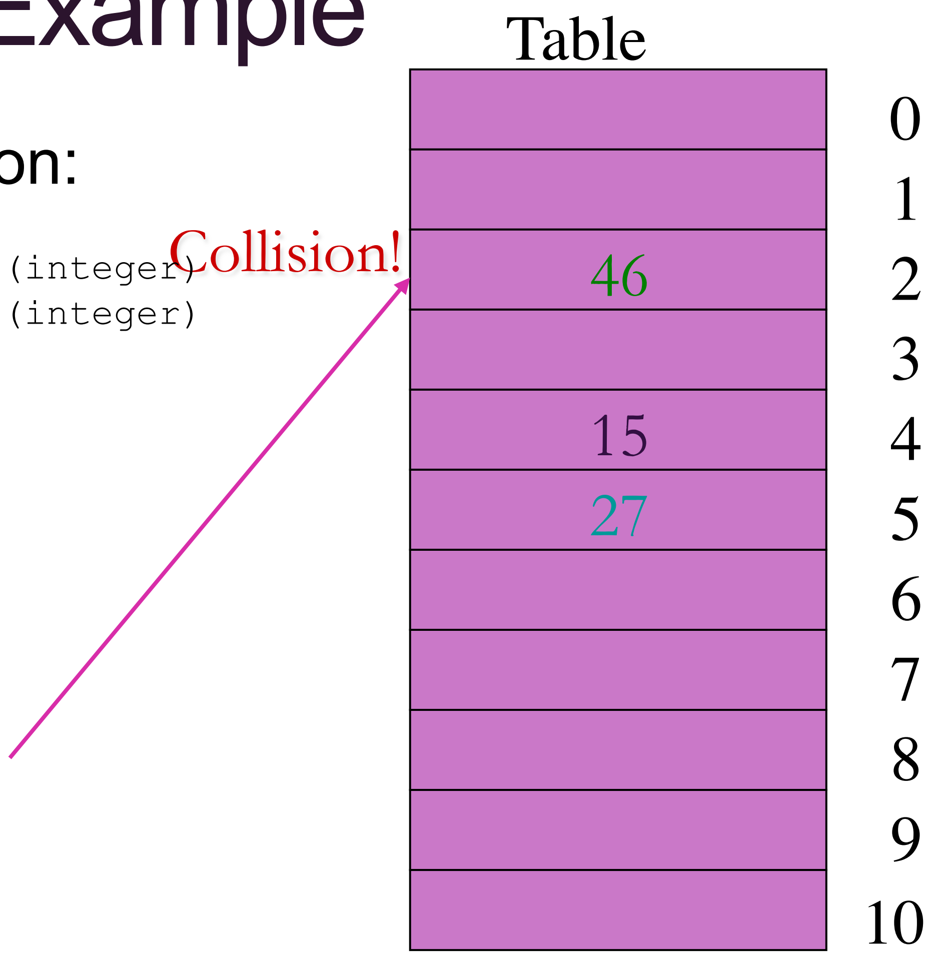
- Keys:

15

46

27

13

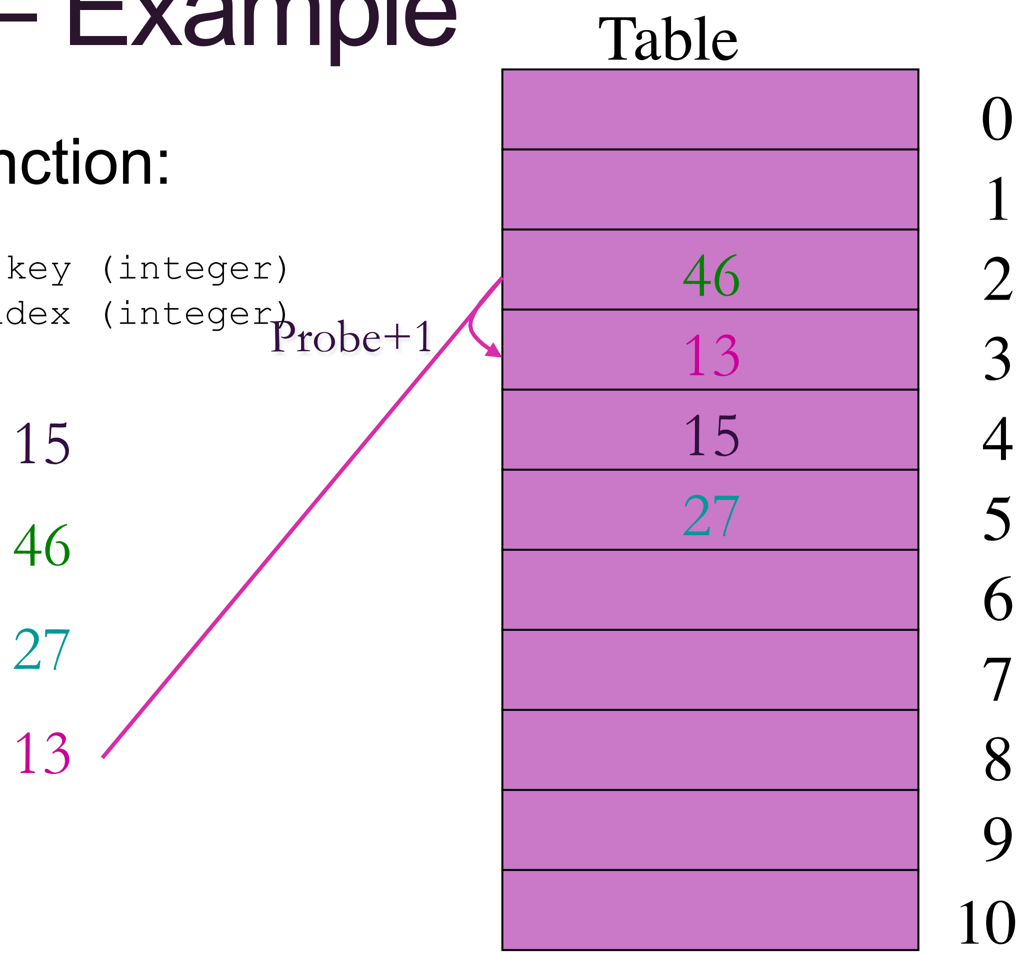


Linear Probing – Example

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key (integer)
                      EXPORT hashIndex (integer)
  hashIndex ← key % 11
```

  - Keys:



Linear Probing – Example

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key (integer)
                    EXPORT hashIndex (integer)
  hashIndex ← key % 11
```

- Keys:

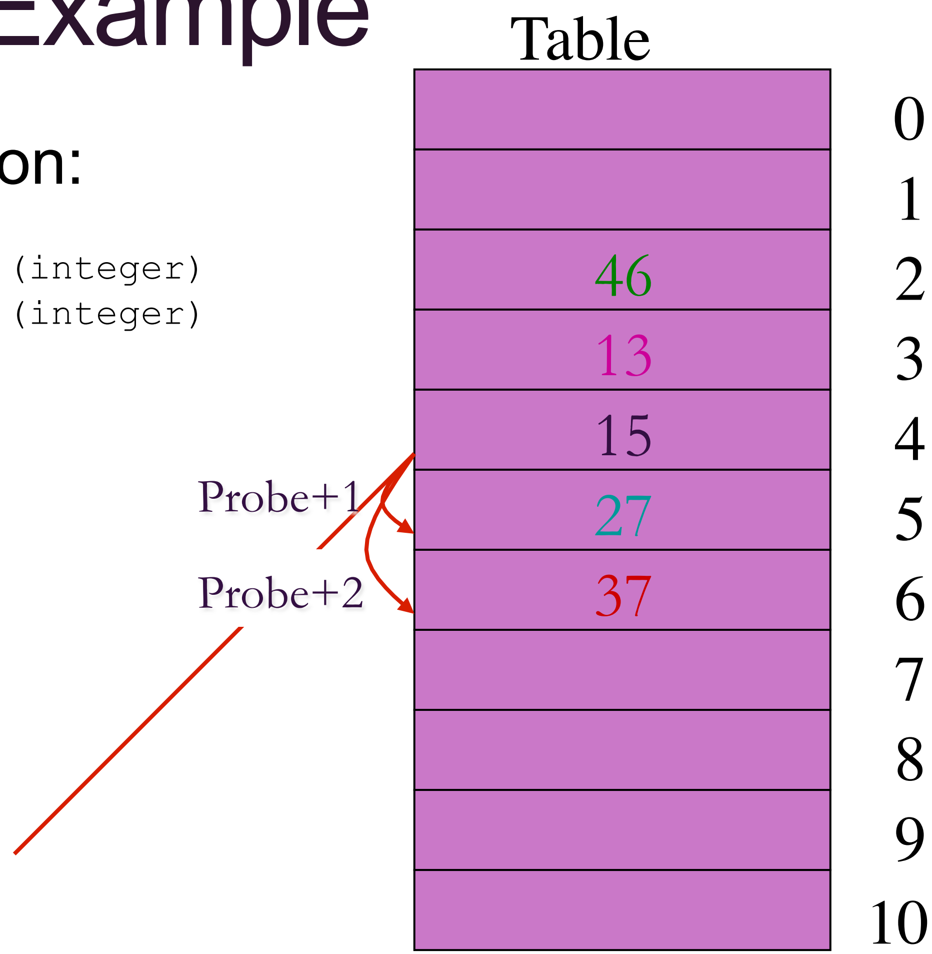
15

46

27

13

37



Linear Probing – Analysis

- When collisions occur, linear probing tends to create clusters of filled slots
  - Since it probes one step at a time, it always fills in gaps
  - … thus creating blocks of filled indexes
    - This is called primary clustering
  - Primary clusters will slow down access time
    - Need multiple probing steps to find end of the primary cluster
      - And then insertion at the end makes the cluster even larger/worse!
      - *e.g.,* try inserting 24 into the previous table
    - Can make fast O(1) insertion degrade considerably, especially if table gets pretty full

Primary Clusters

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key (integer)
                      EXPORT hashIndex (integer)
  hashIndex ← key % 11
```

  - Keys:

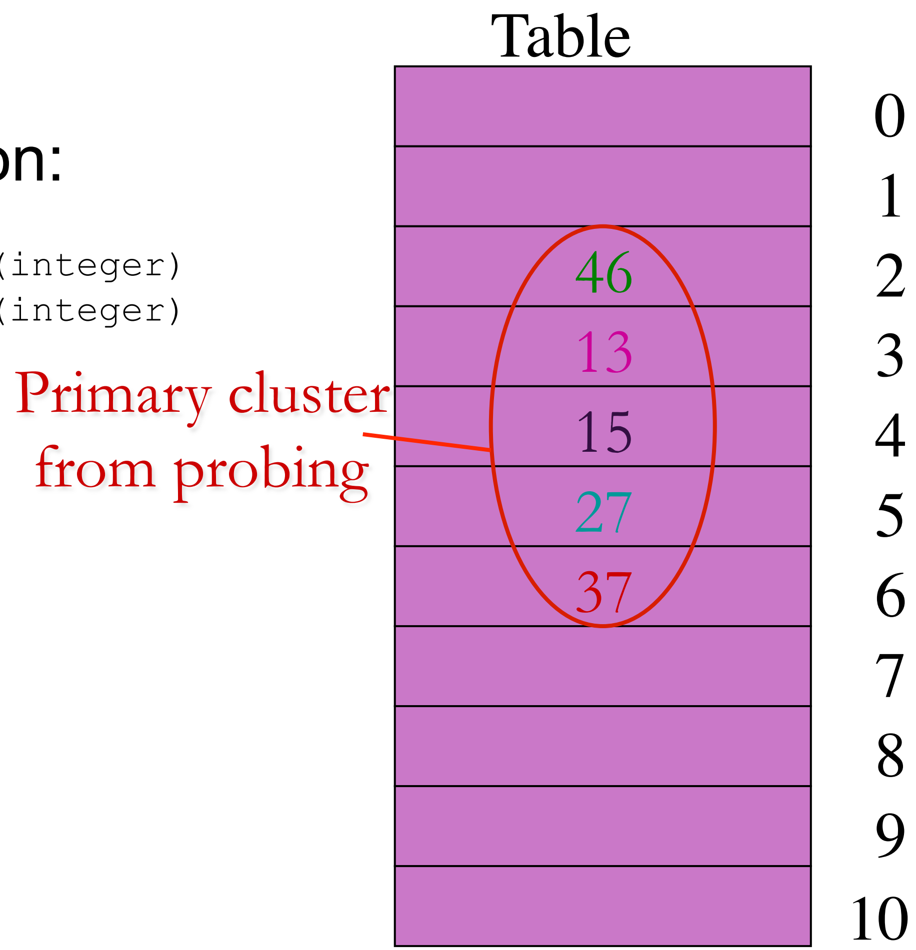
15

46

27

13

37

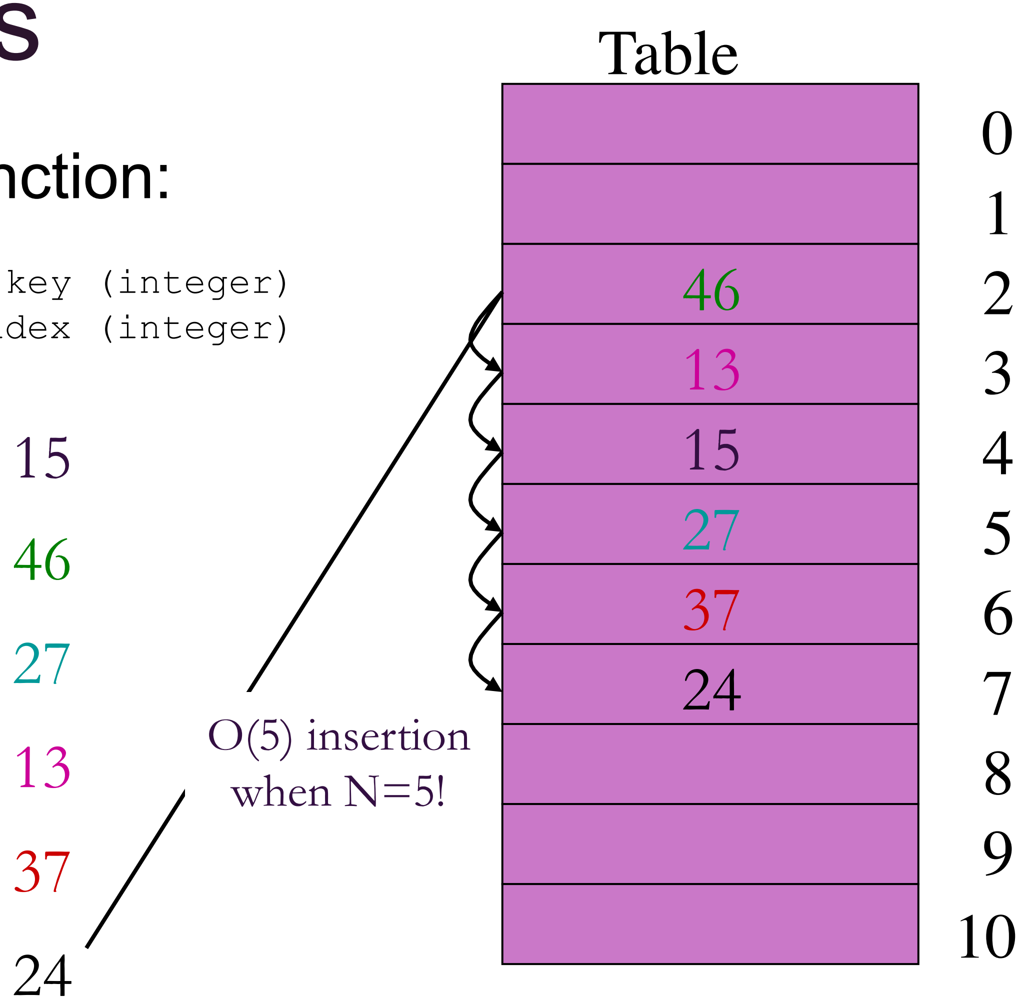


Primary Clusters

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key (integer)
                      EXPORT hashIndex (integer)
  hashIndex ← key % 11
```

  - Keys:



Quadratic Probing

- Exactly like linear probing, except that the step-size increases at each probe iteration
  - In particular, stepSize = probeNum^2
    - *i.e.,* probe1 = +1, probe2 = +4, probe3 = +9, probe4 = +16, *etc.*
  - Hence called *quadratic* probing (due to squaring of probe)

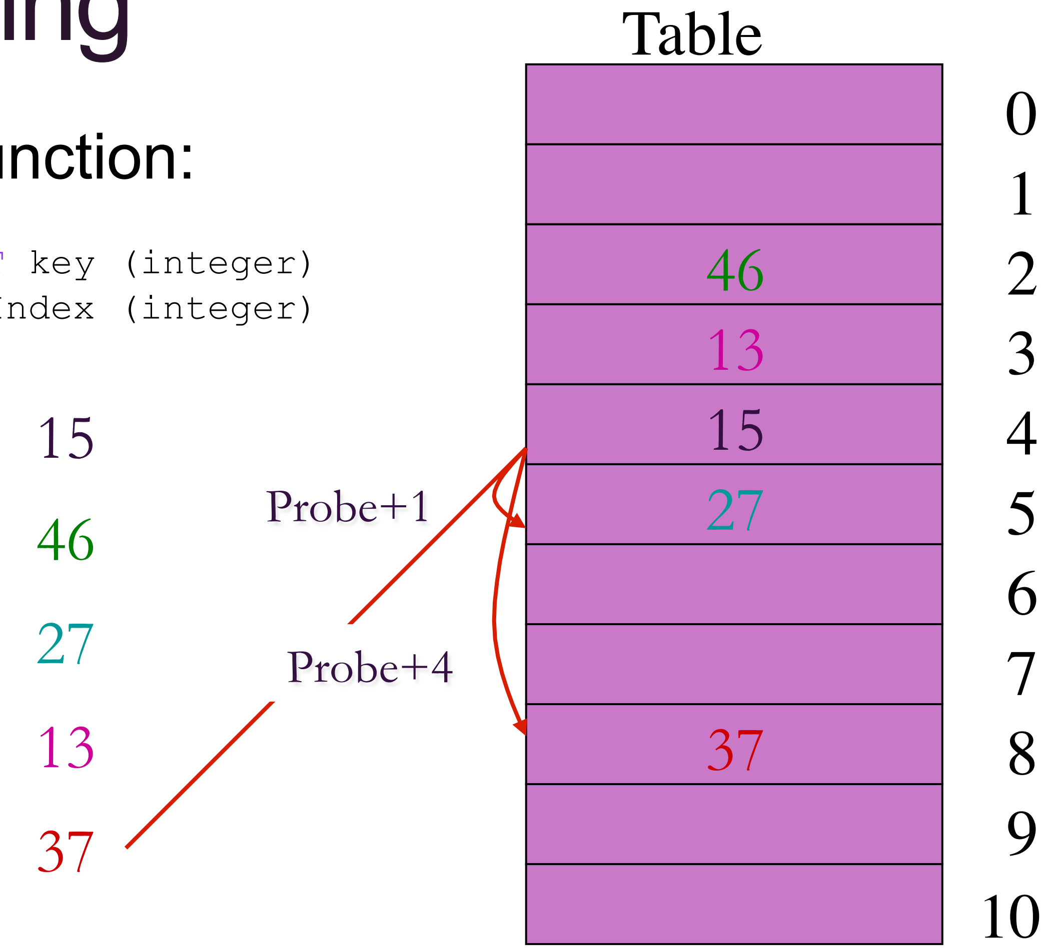
- The increasing step-size means that more gaps will be left in the table when probing
  - Avoids (or at least greatly reduces) primary clusters
  - Step size increases by > 1 as well: means that a probe can 'leap' past a cluster

Quadratic Probing

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key (integer)
                      EXPORT hashIndex (integer)
  hashIndex ← key % 11
```

  - Keys:



Quadratic Probing – Analysis

- Doesn't get primary clusters since step-size > 1
  - Ensures that gaps are left between step jumps
- <u>Not</u> inefficient: deal with (multi-)wrap-around via modulo
  - newHashIdx = (origHashIdx + probeNum^2) % tblSize

- But does get secondary clusters
  - Multiple keys mapping to the same hash index: the *initial* collision then occurs on the same place as a previous value's *initial* collision, so the new value will follow the exact same path as the previous collider
- Also, can't guarantee it will visit all slots
  - Step size always increasing: may miss empty slots

Secondary Clusters

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key
                      EXPORT hashIndex
  hashIndex ← key % 11
```

  - Keys:

15

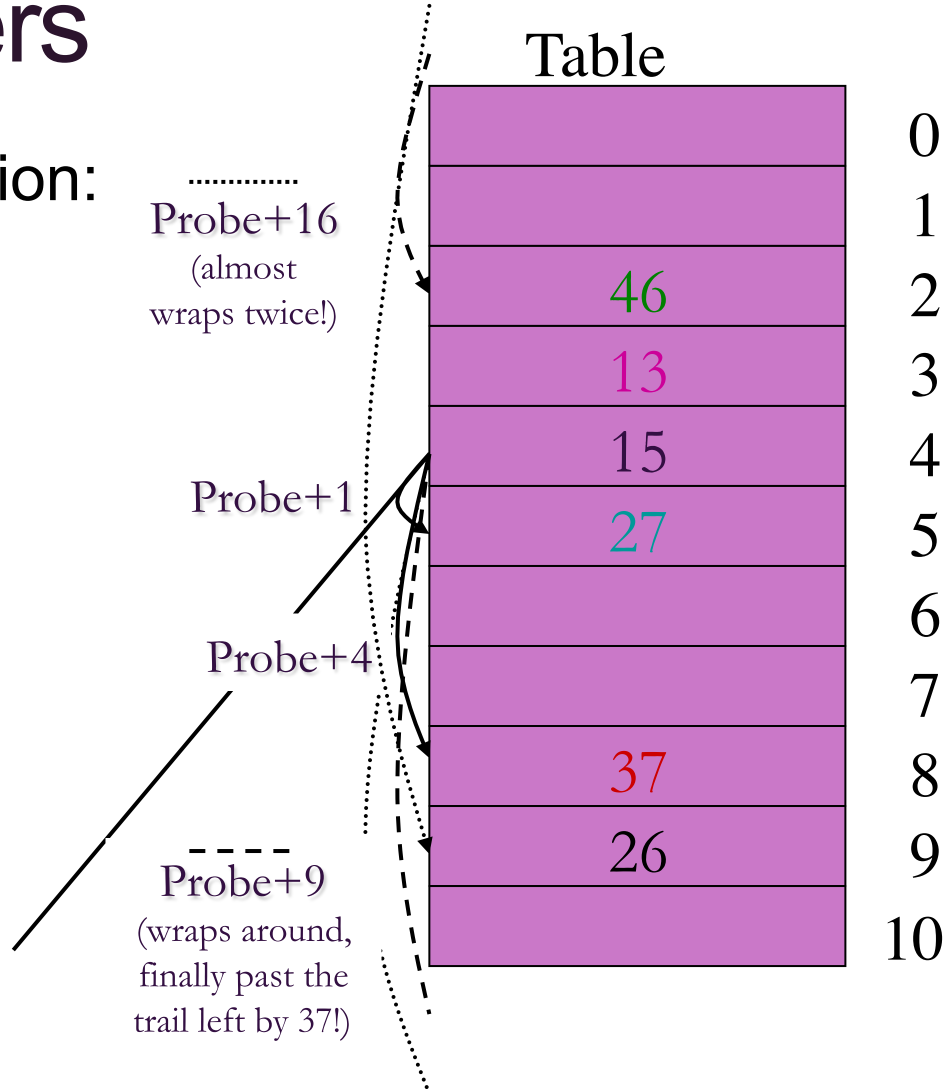
46

27

18

37

26



Double Hashing

- Quadratic probing's increasing-step-size is an issue
  - Cannot guarantee that all slots will be visited
    - So if only one free slot, the quadratic stepping might miss it
      - Although hash tables aren't usually *that* full
- The secondary clustering is also an issue: inefficient

- Double hashing seeks to solve these problems
  - Calculate a step size based on the *key*
    - Each key will have its own step-size increment
      - Greatly reduces secondary clustering
    - Step-size for a given key won't change, thus with a prime-sized table it is *guaranteed* to be able to visit all slots
      - Because no common divisor between step-size and table-size

Double Hashing

- Step-size calc is done by a *second* hash function
  - Simple hash functions are good enough: we aren't overly concerned with even distribution since it's just step-size
  - Define a maximum step-size and make that the modulo
  - Must not produce 0 step sizes though
  - A good secondary hash function is:

```
FUNCTION stepHash IMPORT key (integer)
                 EXPORT hashStep (integer)
 hashStep ← MAX_STEP – ( key % MAX_STEP ) // Step size will be between 1 and maxStep
```

    - MAX_STEP should be small-ish; certainly << table size!
    - Use a prime number for MAX_STEP

Double Hashing

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key
                      EXPORT hashIndex
  hashIndex ← key % 11
```

Secondary hash:

```
FUNCTION stepHash IMPORT key
                  EXPORT hashStep
   hashStep ← 5 - (key % 5)
```

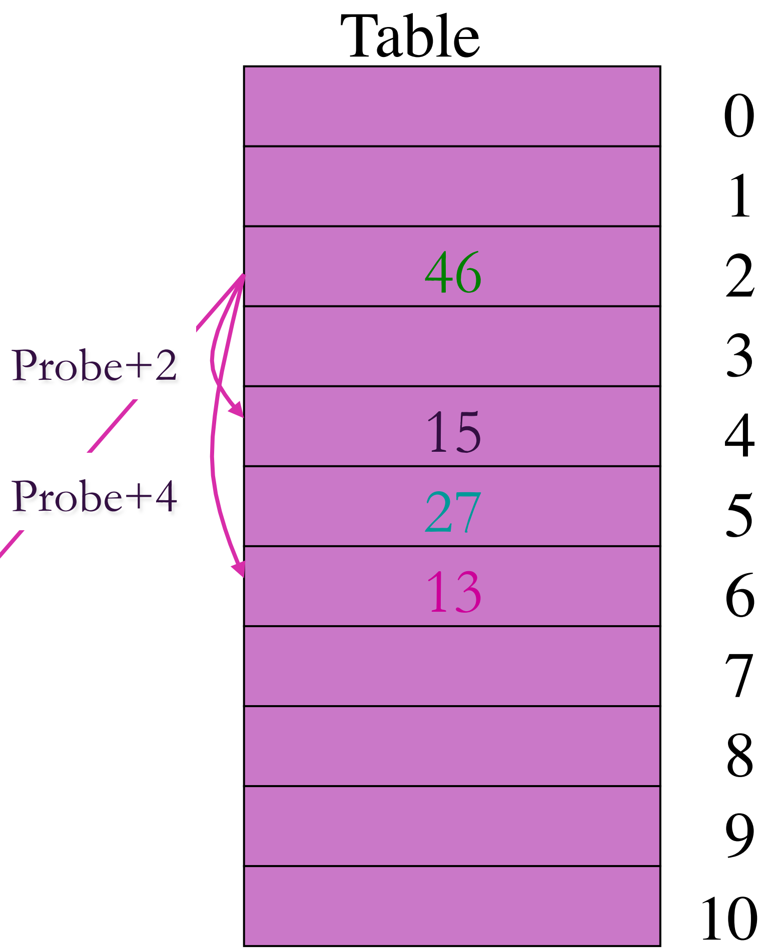
- Keys:

15

46

27

13



Double Hashing

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key
                      EXPORT hashIndex
  hashIndex ← key % 11
```

Secondary hash:

```
FUNCTION stepHash IMPORT key
                  EXPORT hashStep
   hashStep ← 5 - (key % 5)
```

- Keys:

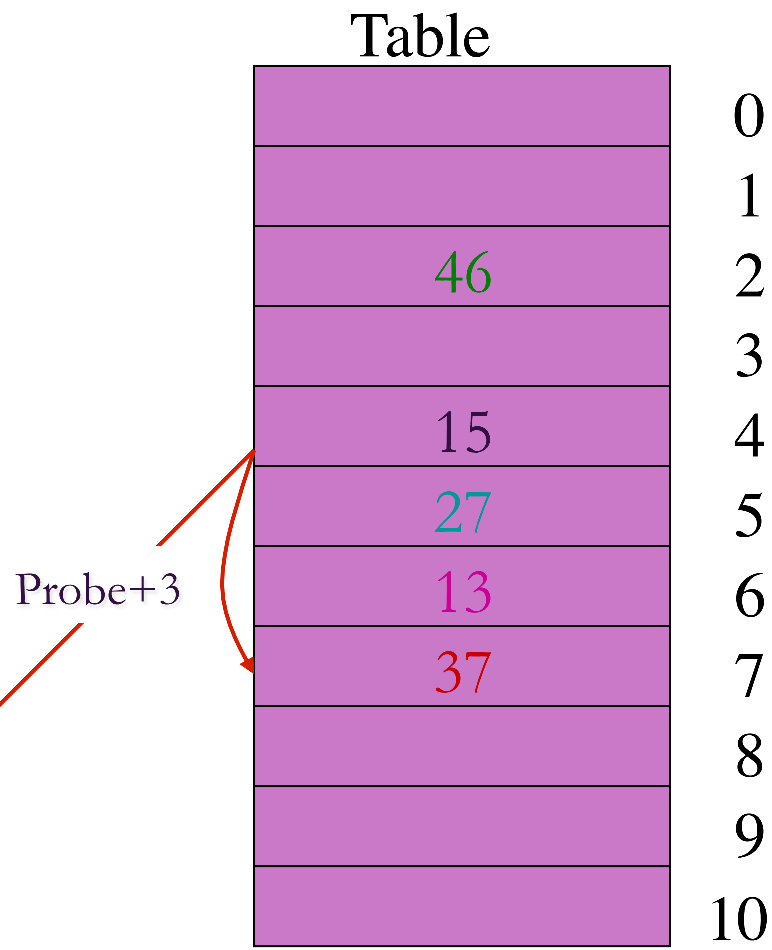
15

46

27

13

37



Double Hashing

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key
                      EXPORT hashIndex
  hashIndex ← key % 11
```

Secondary hash:

```
FUNCTION stepHash IMPORT key
                  EXPORT hashStep
   hashStep ← 5 – (key % 5)
```

- Keys:

15

46

27

13

37

8

Table

	0
	1
46	2
	3
15	4
27	5
13	6
37	7
8	8
	9
	10

Double Hashing

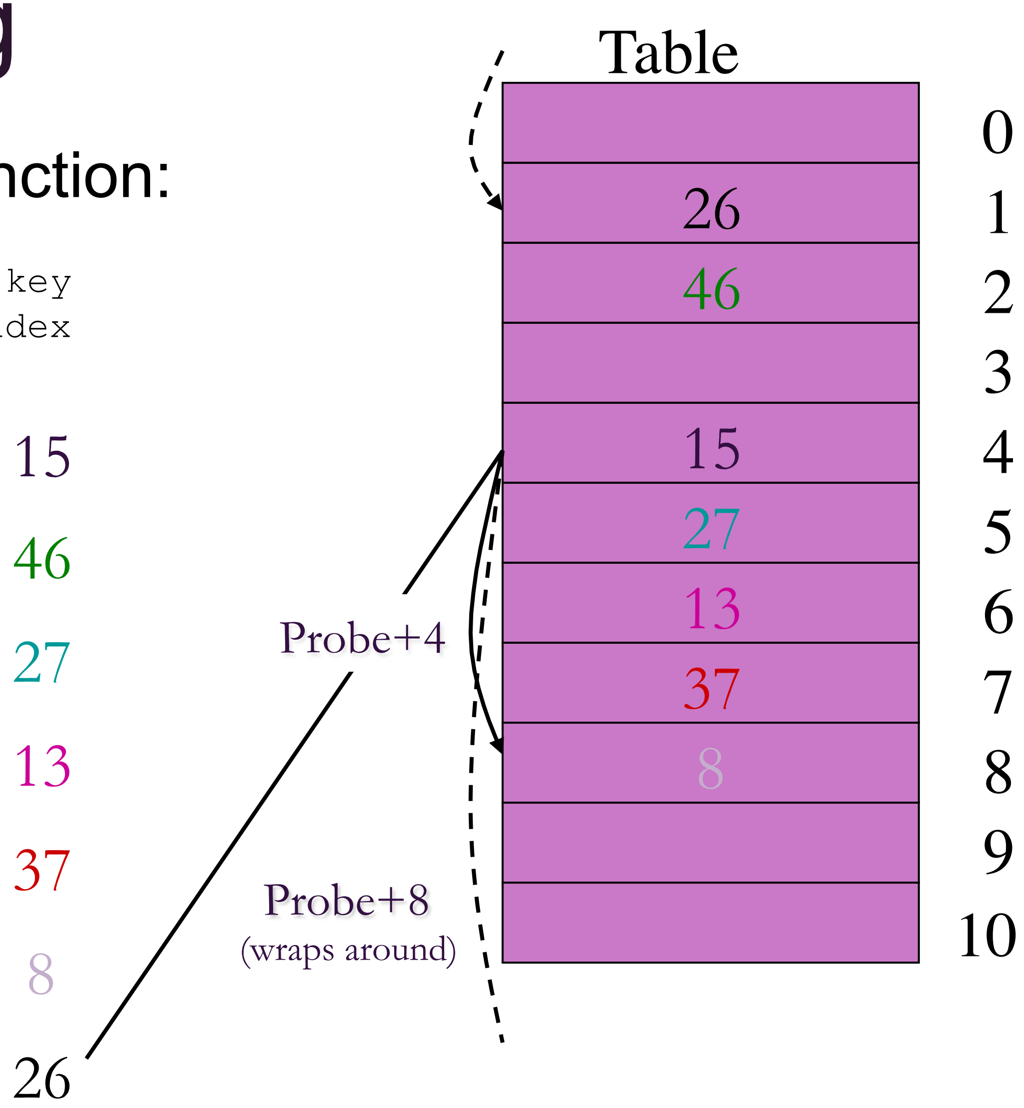
- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key
                      EXPORT hashIndex
  hashIndex ← key % 11
```

Secondary hash:

```
FUNCTION stepHash IMPORT key
                  EXPORT hashStep
   hashStep ← 5 – (key % 5)
```

- Keys:



Double Hashing – Analysis

- No primary OR secondary clustering
  - Since step sizes will often be > 1, gaps will be left when probing, thus primary clusters are unlikely
  - Each key has its own step-size, so no secondary clusters
- Having a prime table length is important here:
  - Guarantees that an empty slot will be found, since no common divisor between table length and *any* step size
    - Otherwise ‘resonances’ can occur when wrapping-around to the beginning of the table during probing – you’d keep visiting the same slots on each pass, and hence miss other slots

Separate Chaining

- The previous approaches used probing on collisions
- Separate chaining uses an entirely different method
  - Make each hashtable entry a pointer to a linked list
  - When a collision occurs, the new key-value pair is simply added to the linked list
    - That way, all items are stored at their key's hash index position
    - Collisions just mean that a chain (linked list) of items are stored at the colliding hash index position

Separate Chaining – Analysis

- This makes for quite different behaviour
  - Must ‘probe’ by traversing across the linked lists
  - No limits on the amount of items you can put in
    - Although performance still degrades as the table fills up
  - O(1) insertion … *no matter how full the table is*
    - Just insert the new item at the front of the linked list
    - But still can degrade to O(N) for access/removal
      - Just like open addressing, but now traversing the linked list
  - But makes things more complicated
    - Every entry is now a linked list : extra coding
    - Space overhead of ‘next’ pointers is also unavoidable: every entry is now data + next pointer

Separate Chaining

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key
                      EXPORT hashIndex
  hashIndex ← key % 11
```

  - Keys:

15

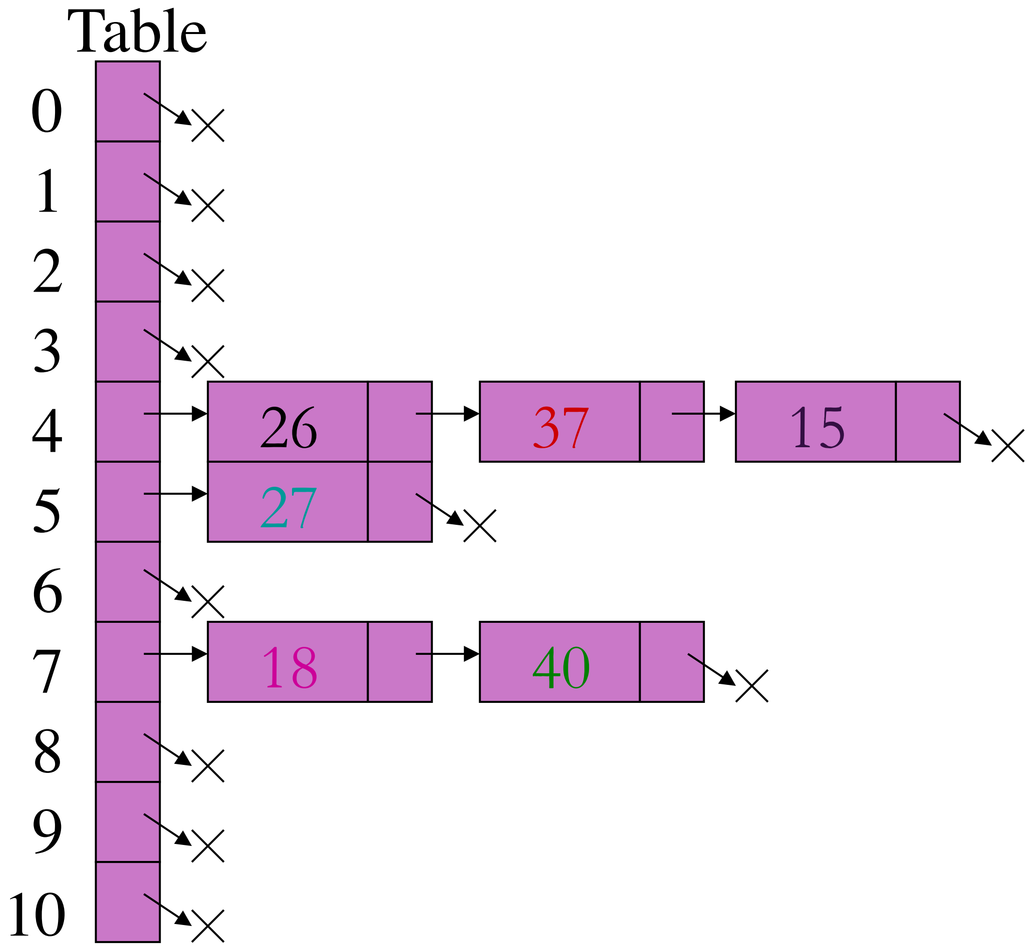
40

27

18

37

26



ACCESSING AND REMOVING

Acessing Values in a Hashtable

- So far, we've focused on inserting key-value pairs
- But accessing a value already in the hashtable is almost the same:
  - The main difference is that instead of hashing and finding a free slot, we instead hash and find the key
  - Thus we *must* store the key as well as the value in the table
    - Otherwise we won't know whether we are at the right value

Accessing Values in a Hashtable

- Separate chaining: hash the key to retrieve the hash index, then search the linked list to find that key

- Open addressing: hash the key and probe (according to the probing algorithm used in insertion) until we find that key
  - We can abort early if we hit an empty slot, since this indicates that the key isn't even in the table
- Depending on how many collisions have occurred, access time can vary from O(1) to O(N)
  - But it will generally be at the lower end: O(1)
  - …unless the load factor of hashtable is high

Removal of Items From a Hash Table

- Removal in separate chaining is easy enough:
  - Just find the item to remove (same as accessing) and delete it from the linked list it resides in

- Open addressing *seems* easy:
  - Just probe to the item (same as accessing) and make the slot empty so that it is free for subsequent inserts
  - There's a small problem with this:
    - Accessing relies on finding empty slots to conclude that the key is not in the table and so abort probing
    - But now we just removed an item that *may* have been in the middle of a chain of historical probes

Open Addr Removing Issue

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key
                      EXPORT hashIndex
  hashIndex ← key % 11
```

  - Assume linear probing

Table

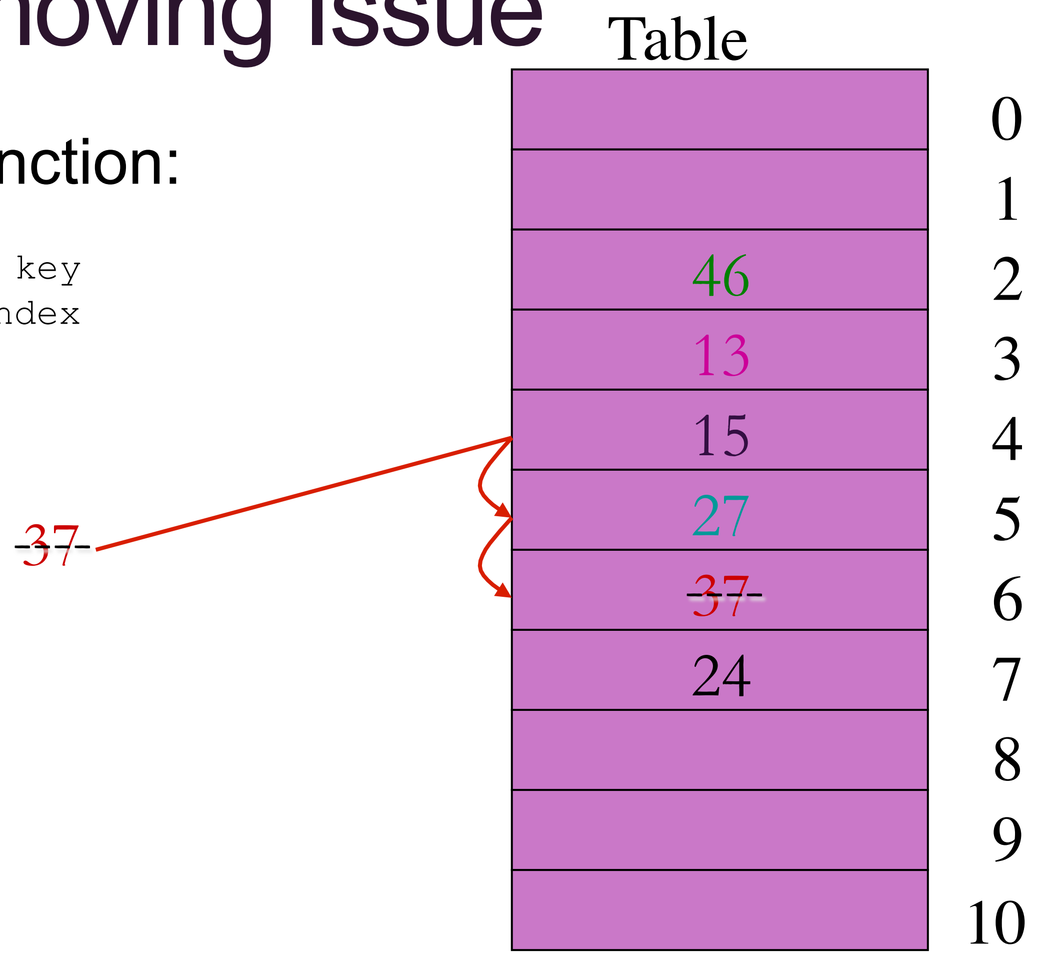
	0
	1
46	2
13	3
15	4
27	5
37	6
24	7
	8
	9
	10

Open Addr Removing Issue

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key
                      EXPORT hashIndex
  hashIndex ← key % 11
```

  - Delete 37



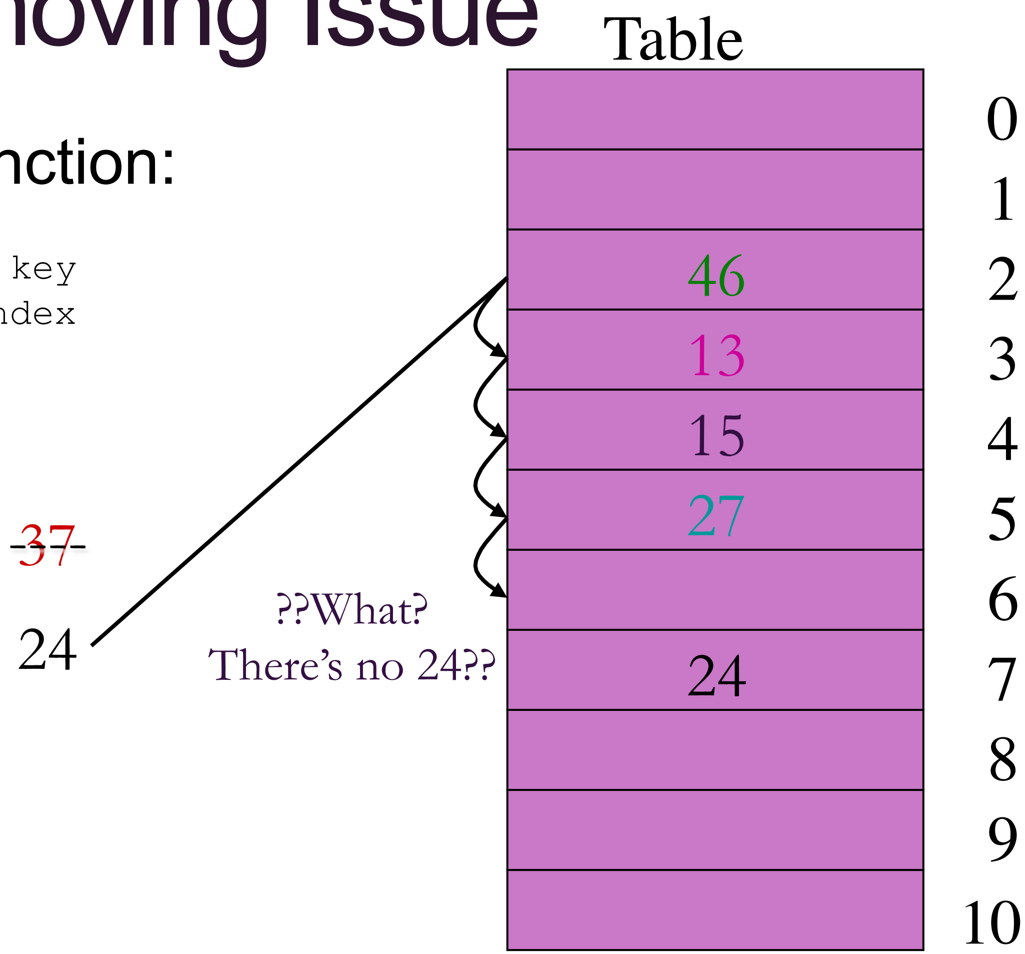
Open Addr Removing Issue

- Basic (poor!) hash function:

```
FUNCTION hashFunction IMPORT key
                     EXPORT hashIndex
  hashIndex ← key % 11
```

  - Delete 37
  - Now access 24

    ?!?!?!



Removal Solution

- The solution for removal is to mark a removed slot as "free-but-formerly-used"
  - Tells access probes that the key it is looking for might be found further down due to the removal
    - Thus access probes can only stop once it encounters a "free-and-never-been-used" slot
    - This does not affect insert: insert still adds to first free slot
  - This can be a problem for hashtables that experience a lot of adds and removes (*e.g.,* due to long lifetime)
    - Eventually, *all* slots will be marked as "free-but-formerly-used"
      - re-build the hashtable!
    - Which means access time for keys that aren't in the table degenerates to O(N):
      - only if the key exists can it stop early

Linear Probing – PseudoCode

```
CLASS DSAHashEntry
FIELD key (String)                          ← Can be any appropriate data type
FIELD value (Object)                        ← So that we can store any kind of data values!
FIELD state (int)                           ← 0 = never used, 1 = used, -1 = formerly-used

CONSTRUCTOR IMPORT nothing
key ← ""
value ← null
state ← 0
                                              There are other ways to do this!
CONSTRUCTOR IMPORT inKey, inValue
key ← inKey
value ← inValue
state ← 1

---------------------------------------------------------------------

CLASS DSAHashTable
FIELD hashArray (DSAHashEntry array)        ← The actual hashtable
FIELD count (int)                           ← Number of items inserted in the hashtable

CONSTRUCTOR IMPORT tableSize

actualSize ← NextPrime(tableSize)
allocate hashArray[actualSize]
FOR ii ← 0 TO actualSize-1 DO               ← Initialise entries
   hashArray [ii] ← allocate HashEntry
ENDFOR
```

Linear Probing – PseudoCode

```
METHOD get IMPORT inKey EXPORT retValue

hashIdx ← hash(inKey)
origIdx ← hashIdx                          ← Just in case the hashtable is 100% full!
found ← FALSE
giveUp ← FALSE

WHILE (NOT found) AND (NOT giveUp) DO
   IF (hashArray[hashIdx].state = 0) THEN        ← Stop if we hit a never-used entry
      giveUp = TRUE
   ELSEIF (hashArray[hashIdx].key = inKey)       ← Check if this is the key we want
      found = TRUE
   ELSE
      hashIdx ← (hashIdx + 1) % hashArray.length ← Probe, handling wrap-around
      IF (hashIdx = origIdx) THEN
         giveUp = TRUE                            ← Stop if we have checked all nodes
      ENDIF
   ENDIF
ENDWHILE

IF (NOT found) THEN
   ABORT                                          ← ie: throw an exception
ENDIF
retValue = hashArray [hashIdx].value
```

Linear Probing – PseudoCode

- You work out the other methods in the practical
- UML Class diagrams hint:

```
DSAHashEntry
-------------------------------------------
key (String)
value (Object)
state (int)
-------------------------------------------
+ CONSTRUCTOR/INIT(nothing)
+ CONSTRUCTOR/INIT(inKey, inValue)

+ SETTERS/GETTERS (or private
  inner class in Java - justify)
```

```
DSAHashTable
-------------------------------------------
hashArray (DSAHashEntry array)
count (int)
-------------------------------------------
+ CONSTRUCTOR/INIT(tableSize)
+ put(inKey, inValue)
+ get(inKey)
+ remove(inKey)
+ getLoadFactor()
+ export()
- resize(size)
- hash(inKey)
- stepHash(inKey)
- nextPrime(inNum)

- findKey(inKey) (optional methods)
- findEmpty(inKey)
```

- + indicates "public" methods and "–" is used for private methods.
- In Python, everything is public, but we can use an underscore to show something is "private" e.g. _hash(inKey)

Load Factor

- Notice collisions are more likely if the table has more items in it
  - More items means less unused slots

- Load factor is a measure of how ‘full’ a hash table is

$$LF = \frac{numItems}{tableCapacity}$$

  - Has important consequences for the frequency of collisions in open-addressing (probing) approaches
    - Similar for separate chaining but not as catastrophic, as we will see

Load Factor and Collisions – Open Addr

- In open addressing, load factor directly indicates the proportion of used slots
  - LF = 0.0: collision is impossible (nothing to collide with!)
  - LF = 1.0: collision is guaranteed (all slots are filled!)
    - And worse: no free slots means you can't insert the new item!
  - LF cannot exceed 1.0 in open addressing approaches
    - Can't put in more items than there are table slots!

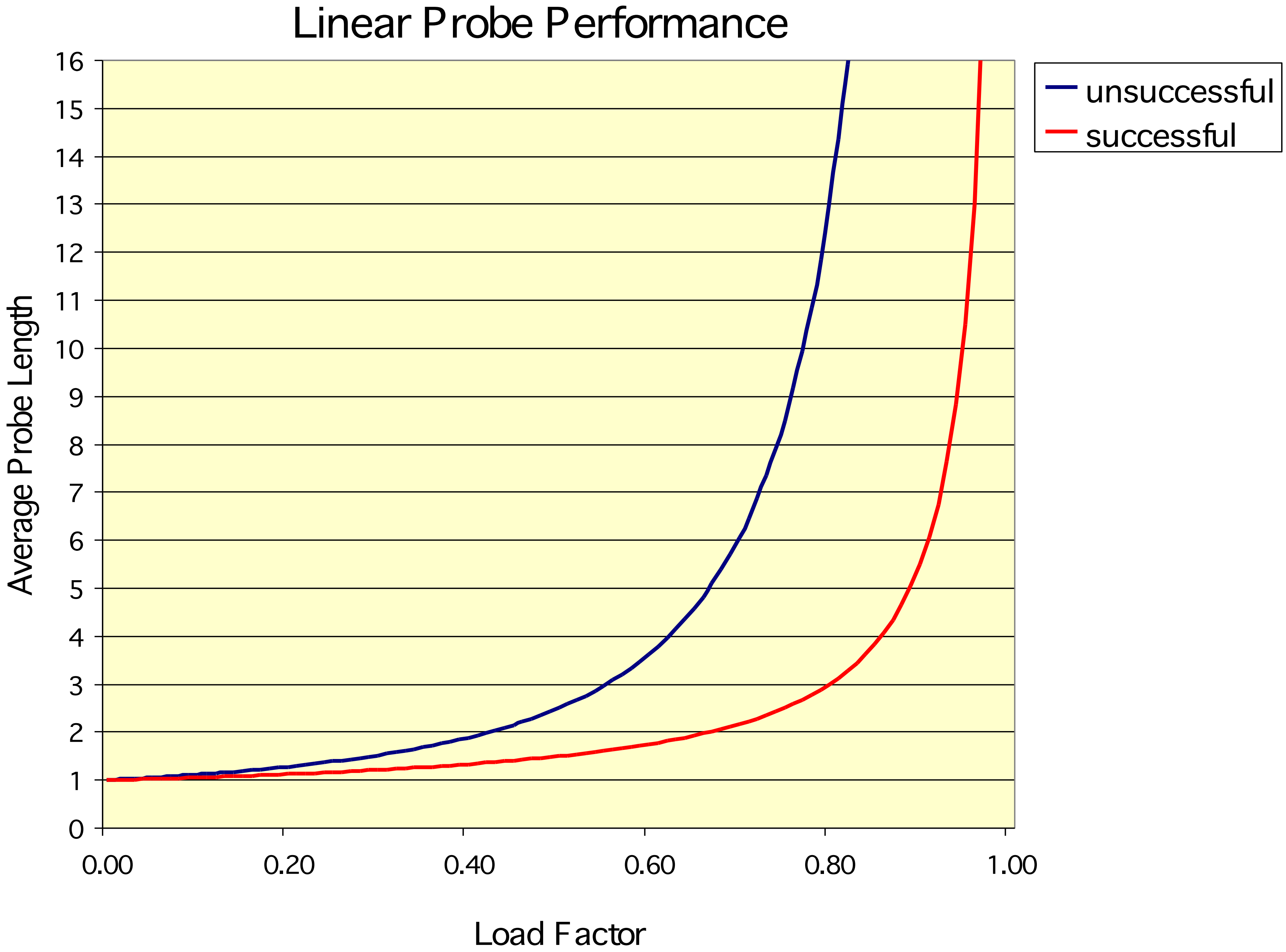
  - Collisions will increase as we move from LF = 0.0 to 1.0
    - Exactly what the curve looks like depends on the hash function; better (uniform) hash functions will only get bad at higher LFs

Load Factor and Collisions – Sep. Chain

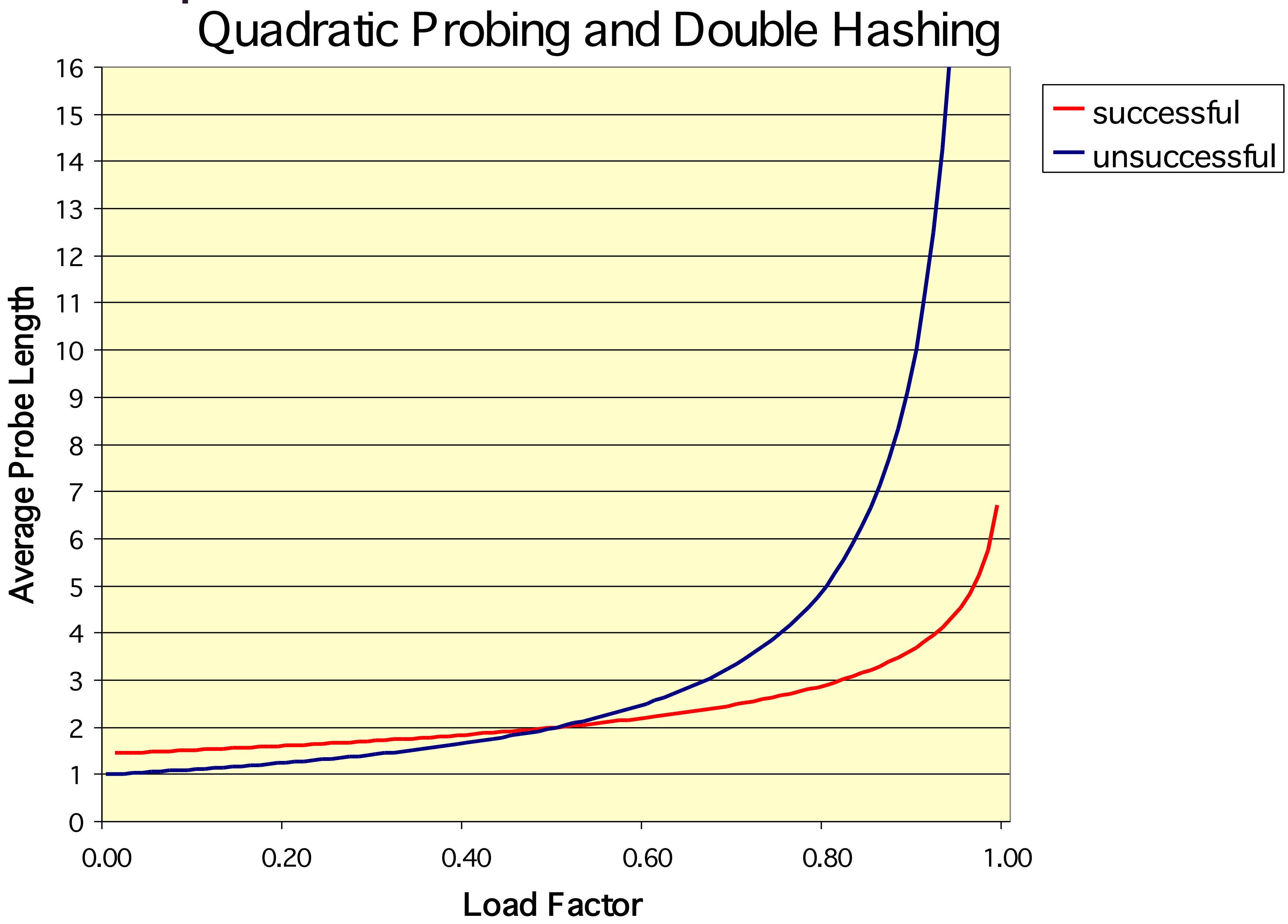
- Separate chaining is not so badly affected
  - Insertion in separate chaining is *always* O(1)
  - Only access/deletion time is affected
  - Consider: LF = 1.0 indicates that each slot has, on average, 1 entry in the linked list for that slot
    - Some entries will have more, some entries will have none
  - Thus access time will be, on average, O(2)
    - In comparison: with probing methods, access time depends on the number of probes, which is often >> 1 when LF $\rightarrow$ 1.0

  - Access time increases as LF increases: approx O(LF)
    - But only on average: some individual slots may be very full

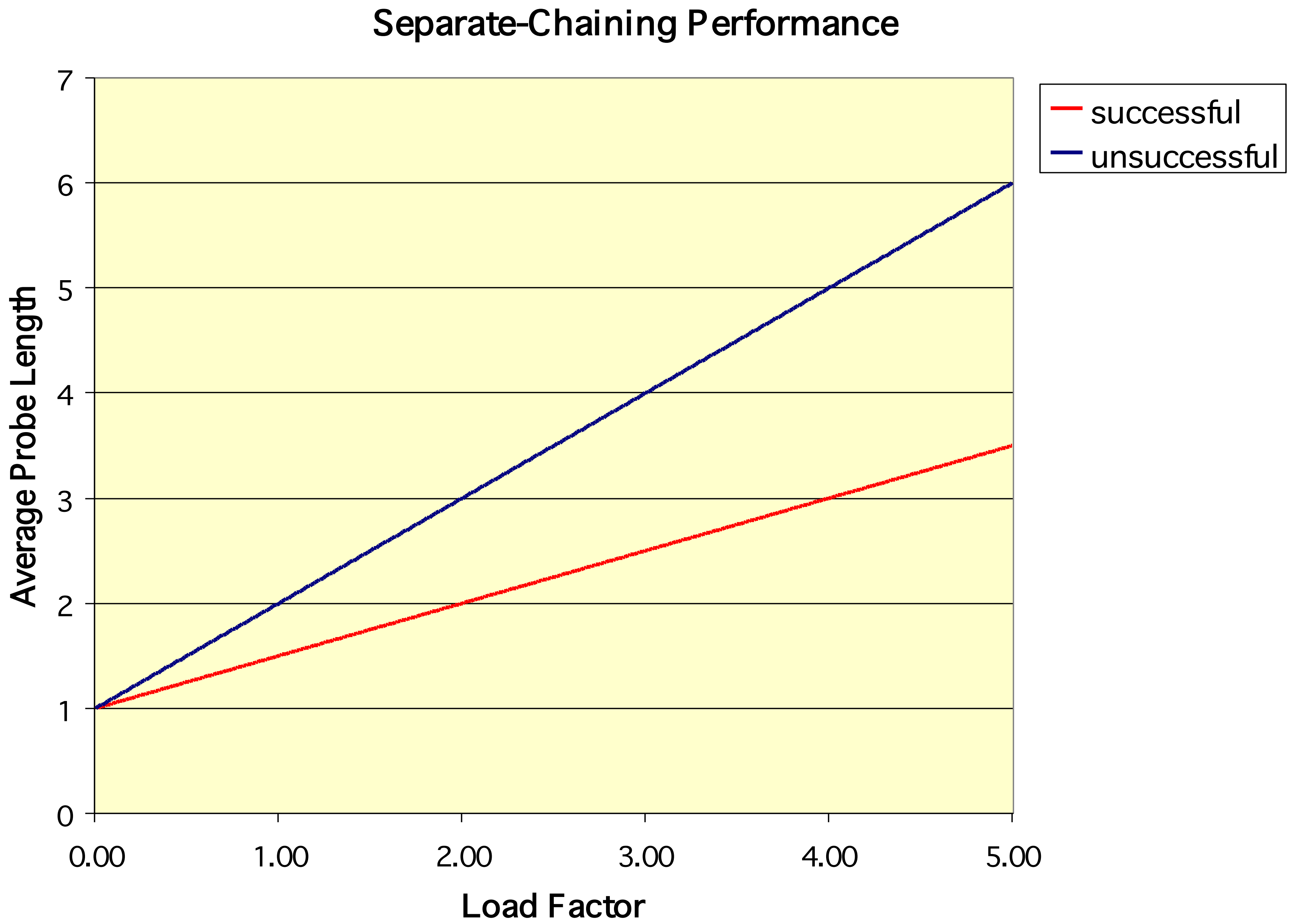
Load Factor Impact:



Load Factor Impact:



Load Factor Impact:



Performance Conclusions

- For open addressing, make sure the hash table isn't too full
  - *i.e.,* Keep the load factor below 0.6 or 0.7
- But don't make the table too big
  - LF < 0.4 means 60% of the table is unused (wasted) space
- Not so much of a problem for separate chaining
  - Overall, separate chaining tends to be less 'brittle', but also more complex to implement since you need the linked lists

Collision-Handling Comparison

- Separate chaining pros (vs open addressing methods)
  - ☑ O(1) insertion regardless of how full the table is
  - ☑ No limit on number of items that can be stored
    - *i.e.,* load factor can exceed 1.0
  - ☑ Performance is better with higher load factors
  - ☑ No need to specially mark slots with removed items
    - The linked list will automatically ‘close up’ the gap

- Separate chaining cons
  - ☒ More complicated data structure needed (array of linked lists)
  - ☒ Node pointer memory overhead for each element
  - ☒ Access time tends to be slower when load factor is low

Collision-Handling Comparison

- Linear probing (vs other open addressing approaches):
  - ☑ Simplest open addressing approach
  - ☑ Will always find a free slot
  - ☒ Suffers from primary clustering
    - This can make insertion/accessing quite slow if it gets bad
- Quadratic probing
  - ☑ Largely eliminates primary clustering
  - ☒ Suffers from secondary clustering
  - ☒ May fail to find free slots (eg: if there are only a few left)
    - … and so also can't tell when to give up looking for a key!

Collision-Handling Comparison

- Double Hashing:
  - ☑ No problems with primary or secondary clustering
  - ☑ Will always find a free slot
    - … as long as the table size is prime
  - ☒ Needs a secondary hash function (extra complexity)

Next Week

- Heaps