Hashing

Fatemeh Panahi
Department of Computer Science
University of Toronto
CSC263-Fall 2017
Lecture 5

Announcements

- $A1$ marks are released in Markus (average 79), $A2$ handout is out in Piazza.
- Office hour on Thursday: $10:00 - 11:00$ am
- Tutorial on Friday: Our new tutorial TA, Roleen Nunes will solve some hashing problems including exercises 11.1-1, 11.1-4, 11.2-2, 11.4-1, 11.4-3 from the textbook.

Today

- Direct access table
- Hash Table, Hash function
  - Division method
  - Multiplication method
- Collision handling
  - Chaining: Load Factor and simple uniform hashing
  - Open Addressing
    - Linear probing
    - Quadratic probing
    - Double Hashing

Reading Assignments

11.1-1, 11.2-1, 11.2-2

How to make search faster?

Knowing Where to Look

Hash – running time

	Unsorted list	Sorted Array	Balanced BST	Hash Table
Search	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Insert	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
Delete	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$

average-case and Under reasonable assumptions

Direct Access Table

Question 1

Q: Given $A1$ marks for $CSC263$ we want to find occurrence of each mark.

Direct-address table: store number of occurrences of each mark in array with 101 positions (one for each mark). All operations $\Theta(1)$ and memory usage small.

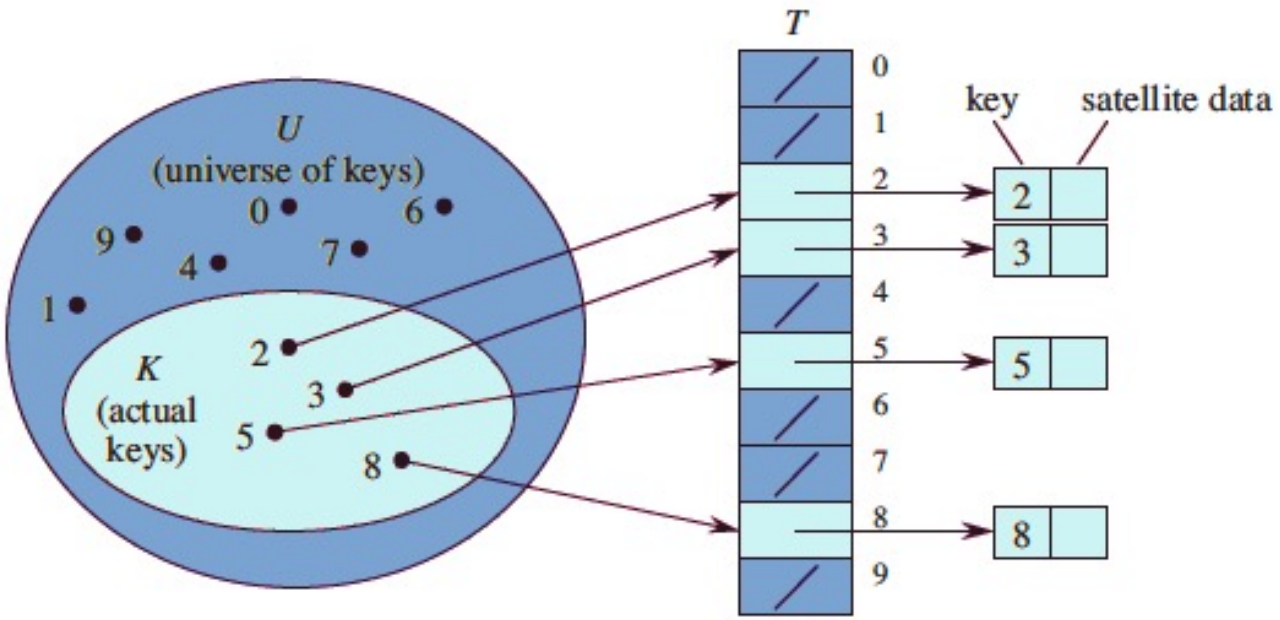
	0	1	2	3	4							98	99	100
A														

Direct-address table

- Universe of keys U: the set of all possible keys.
- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe $U = \{0, 1, \ldots, m - 1\}$
- Idea:
  - Store the items in an array, indexed by keys. an array with m positions, each position is called a "slot" or a bucket.

Direct-address table representation

- An array $T[0 \ldots m - 1]$
- Each **slot**, or **bucket**, in T corresponds to a key in U
- For an element x with key k, a pointer to x (or x itself) will be placed in location $T[k]$
- If there are no elements with key k in the set, $T[k]$ is empty, represented by NIL



Direct-address table - Operations

$DIRECT\text{-}ADDRESS\text{-}SEARCH(T, k)$

return $T[k]$

$DIRECT\text{-}ADDRESS\text{-}INSERT(T, x)$

$T[key[x]] \leftarrow x$

$DIRECT\text{-}ADDRESS\text{-}DELETE(T, x)$

$T[key[x]] \leftarrow NIL$

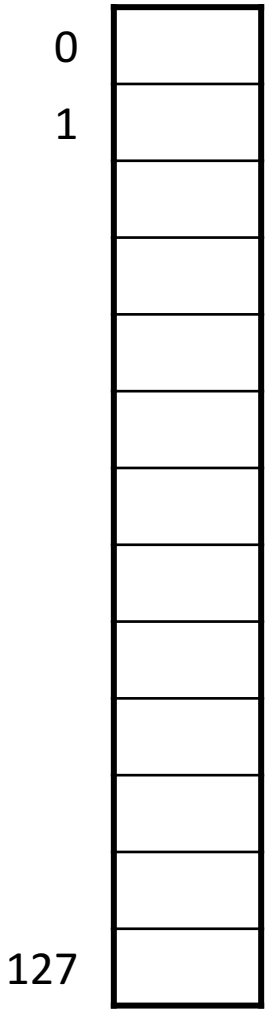
Running time for all the operations: $O(1)$

Question 2

Read a text file, keep track of number of occurrences of each character

ASCII codes 0 – 127 (7 bits)

ASCII control characters			ASCII printable characters					
00	NULL	(Null character)	32	space	64	@	96	`
01	SOH	(Start of Header)	33	!	65	A	97	a
02	STX	(Start of Text)	34	"	66	B	98	b
03	ETX	(End of Text)	35	#	67	C	99	c
04	EOT	(End of Trans.)	36	$	68	D	100	d
05	ENQ	(Enquiry)	37	%	69	E	101	e
06	ACK	(Acknowledgement)	38	&	70	F	102	f
07	BEL	(Bell)	39	'	71	G	103	g
08	BS	(Backspace)	40	(	72	H	104	h
09	HT	(Horizontal Tab)	41	)	73	I	105	i
10	LF	(Line feed)	42	*	74	J	106	j
11	VT	(Vertical Tab)	43	+	75	K	107	k
12	FF	(Form feed)	44	,	76	L	108	l
13	CR	(Carriage return)	45	-	77	M	109	m
14	SO	(Shift Out)	46	.	78	N	110	n
15	SI	(Shift In)	47	/	79	O	111	o
16	DLE	(Data link escape)	48	0	80	P	112	p
17	DC1	(Device control 1)	49	1	81	Q	113	q
18	DC2	(Device control 2)	50	2	82	R	114	r
19	DC3	(Device control 3)	51	3	83	S	115	s
20	DC4	(Device control 4)	52	4	84	T	116	t
21	NAK	(Negative acknowl.)	53	5	85	U	117	u
22	SYN	(Synchronous idle)	54	6	86	V	118	v
23	ETB	(End of trans. block)	55	7	87	W	119	w
24	CAN	(Cancel)	56	8	88	X	120	x
25	EM	(End of medium)	57	9	89	Y	121	y
26	SUB	(Substitute)	58	:	90	Z	122	z
27	ESC	(Escape)	59	;	91	[	123	{
28	FS	(File separator)	60	<	92	\	124	|
29	GS	(Group separator)	61	=	93	]	125	}
30	RS	(Record separator)	62	>	94	^	126	~
31	US	(Unit separator)	63	?	95	_		
127	DEL	(Delete)						



Question 3

Question: There are 250 students and we want to search for students GPA. Keys are 10-digit students number.

We can use the same strategy, but it is very inefficient, an array of 10 billions is needed.

$|U|$ can be very large and the number of keys can be much smaller than U.

Direct-address table - Drawback

1. What if the keys are not integers? Cannot use keys as indices anymore!
2. When K is much smaller than U, it uses a huge space. Most space is wasted.

Good Approach:

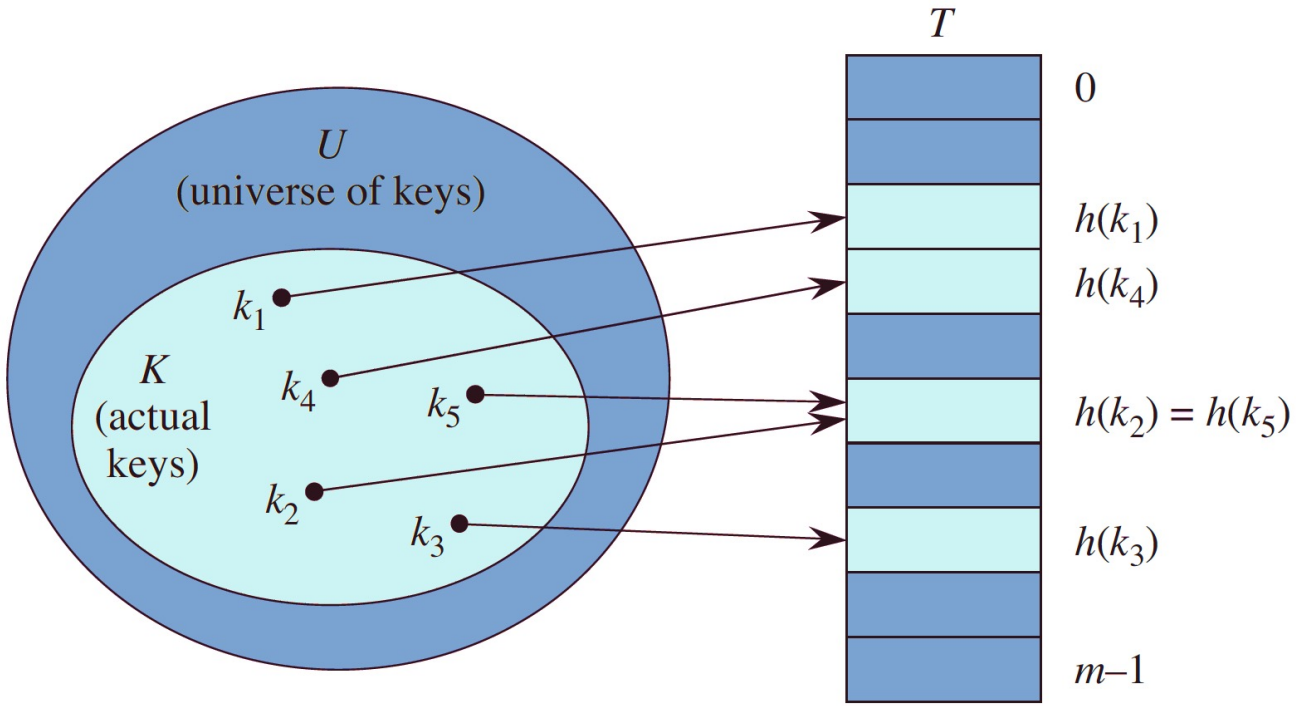
1. We need a mapping to be able to convert any type of key to an integer.
2. We need to map the universe of keys into a small number of slots.

A hash function does both.

Hash Tables

Hash function

A hash function maps U to $\{0, 1, \ldots, m-1\}$, in other words, $h(k)$ maps any key k to one of the m buckets in table T. In yet other words, in array T, $h(k)$ is the the index at which the key k is stored.



Hash function - Application

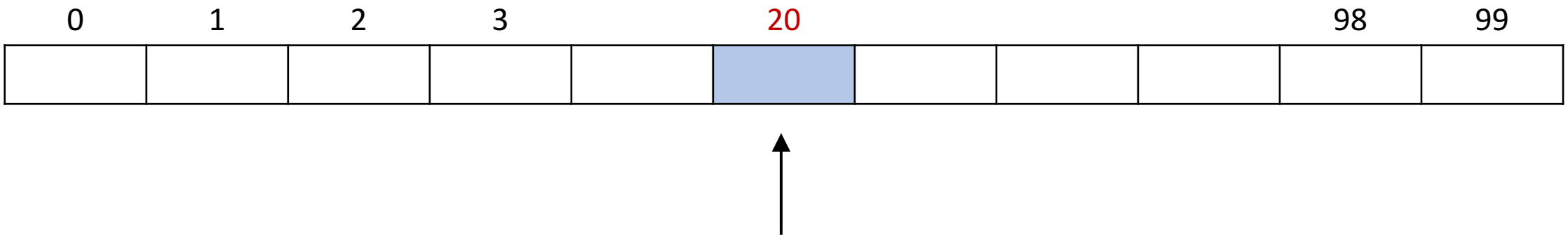
- Storing on a cloud
- Verifying downloaded applications

Revisit Question 3

- There are 250 students and we want to search for students GPA. Keys are 10-digit students number.
- Possible hash function:
- $h(student\ number) = student\ number\ \%100$ (last two digits)
- For example $h(1008489820) = 20$

Hash table- Collisions

- Two or more keys hash to the same slot!!



$$h(1008489820) = 20$$

$$h(1005678920) = 20$$

$$h(1087656920) = 20$$

$$h(1598655920) = 20$$

Hash table- Collisions

- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function

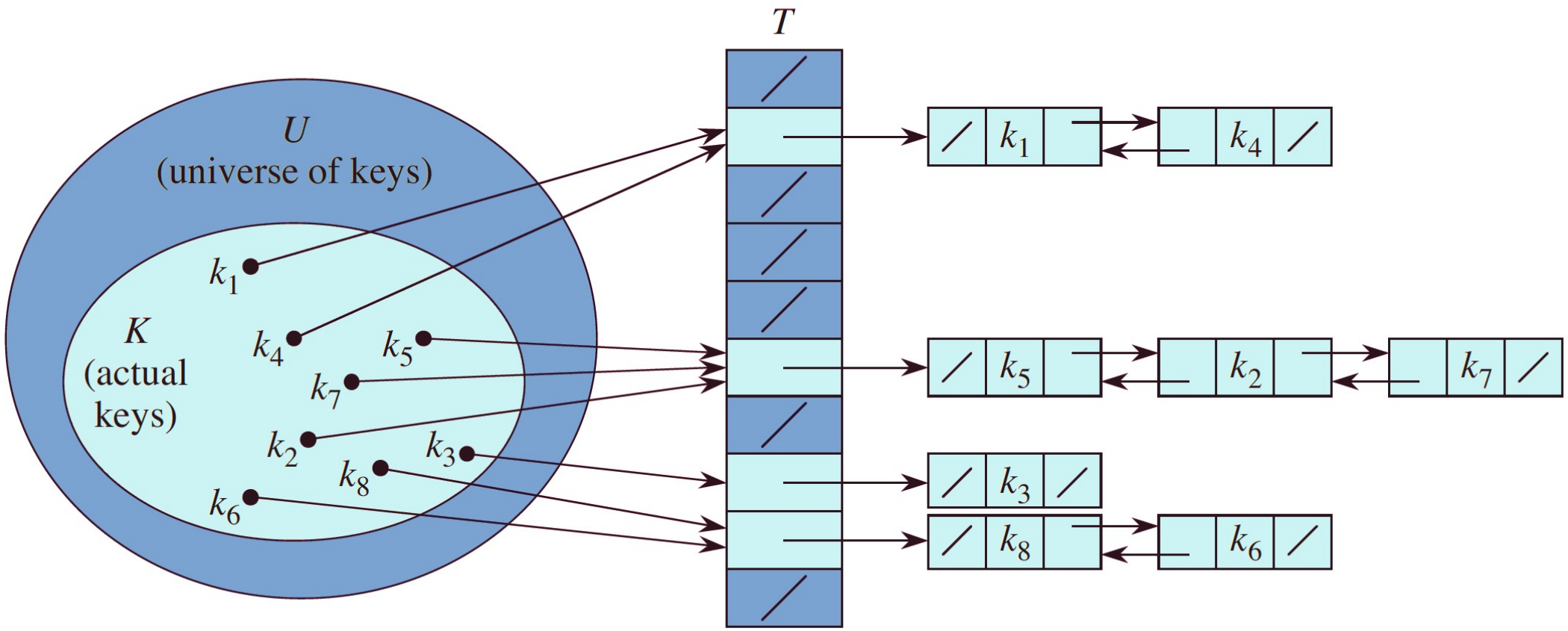
Collision handling

- We will review the following methods:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
- We will discuss chaining first, and ways to build "good" functions.

Collision handling - chaining

Idea:

- Put all elements that hash to the same slot into a linked list
- Slot *j* contains a pointer to the head of the list of all elements that hash to *j*



Collision with Chaining - Discussion

- Choosing the size of the table
  - Small enough not to waste space
  - Large enough such that lists remain short
  - Typically 1/5 or 1/10 of the total number of elements
- How should we keep the lists: ordered or not?
  - Not ordered!
    - Insert is fast
    - Can easily remove the most recently inserted elements

Insertion in a hash table

$CHAINED\text{-}HASH\text{-}INSERT(T, x)$

insert x at the head of list $T[h(key[x])]$

- Worst-case running time is $O(1)$
- Assumes that the element being inserted isn't already in the list
- It would take an additional search to check if it was already inserted

Deletion in a hash table

$CHAINED\text{-}HASH\text{-}DELETE(T, x)$

delete x from the list $T[h(key[x])]$

- Need to find the element to be deleted.
- Worst-case running time:
  - Deletion depends on searching the corresponding list

Searching in a hash table

$CHAINED\text{-}HASH\text{-}SEARCH(T, x)$

search for an element with key k in list $T[h(k)]$

- Worst case running time $O(n)$. Doesn't sound too good.

However, in practice, hash tables work really. well, that is because The worst case almost never happens.

Average case performance is really good.

In fact, Python "dict" is implemented using hash table

Average case analysis:
Search in hashing with chaining

Analysis of Hashing with Chaining

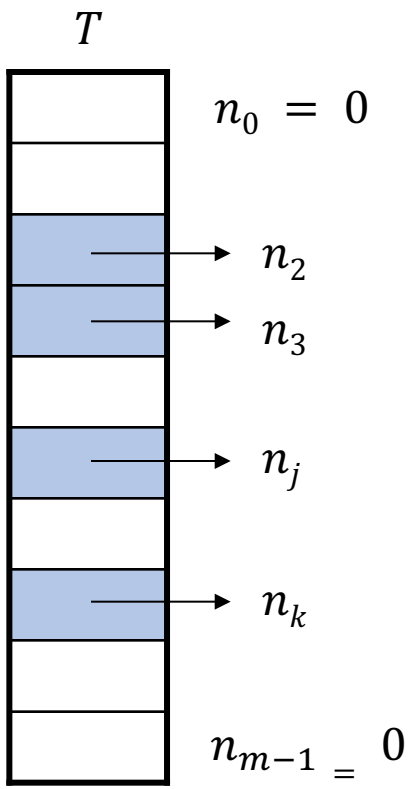
- Average case
  - depends on how well the hash function distributes the n keys among the m slots
- **Simple uniform hashing** assumption:
  - Any given element is equally likely to hash into any of the m slots (i.e., probability of collision $\Pr\big(h(x) = h(y)\big) = 1/m$)
- Length of a list:

$$T[j] = n_j, \quad j = 0, 1, \ldots, m-1$$

- Let X be the number of items hashed in a slot j. We want to compute $E(X)$. Let X_i be an indicator random variable defined as follows:

$$X_i = \begin{cases} 1 & \text{if item } i \text{ is in slot } j \\ 0 & \text{if item } i \text{ is not in slot } j \end{cases}$$

$$E(X) = \sum_{i=1}^{n} p(X_i = 1) = \sum_{i=1}^{n} \frac{1}{m} = n/m$$



Load Factor of a Hash Table

- Load factor of a hash table T:

$$\alpha = n/m$$

n = # of elements $\qquad$ m = # of slots in the table

- α encodes the average number of elements stored in a chain α can be $<, =, >$ 1

Assumption: Simple Uniform Hashing

- If $n < m$, i.e., more slots than keys stored, the running time is $\Theta(1)$
- If n/m is in the order of a constant, the running time is also $\Theta(1)$
- If n/m of higher order, e.g., $\sqrt{n}$, then it's not constant anymore.

So, in practice, choose m wisely to guarantee constant average-case running time.

Average Case for search in a hash table

An unsuccessful search (item not stored in the table) takes $\Theta(1 + \alpha)$ expected time under the assumption of simple uniform hashing

We consider two cases:

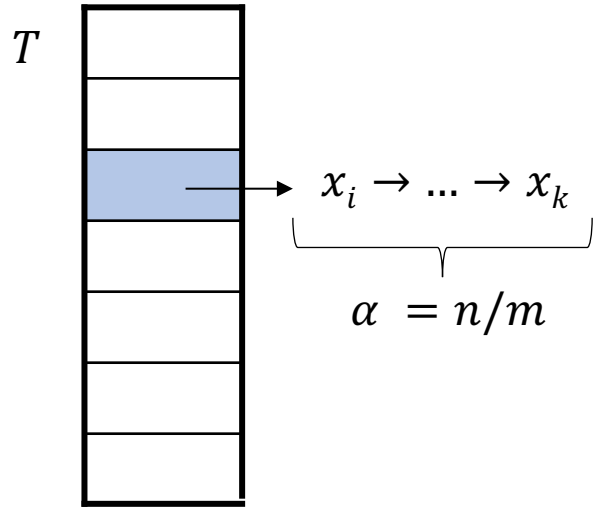
1. Unsuccessful search
2. Successful search

Case 1: Unsuccessful Search

- Searching unsuccessfully for any key k need to search to the end of the list $T[h(k)]$
- Expected length of the list:

  $$E[n_{h(k)}] = \alpha = n/m$$

- Expected number of elements examined in an unsuccessful search is α
- Add 1 step for the hashing $h(k)$,
- then the average-case running time for Search is in at most $1 + \alpha$ or $O(1 + \alpha)$



Case 2: Successful Search

- Let xi denote the ith element inserted into the table, for $i = \{1, \dots, n\}$, and let $k_i = x_i.key$.

$$x_n \rightarrow x_{n-1} \rightarrow \cdots \rightarrow x_4 \rightarrow x_3 \rightarrow x_2 \rightarrow x_1$$

- Note that we insert the elements at the head of the list.
- It means that when you search for x_i only you lookup elements x_j where $j > i$.
- We assume that the element being searched for is equally likely to be any of the n elements stored in the table.

Case 2: Successful Search (Cont'd)

For keys k_i and k_j,

we define the indicator random variable $X_{ij} = I\{h(k_i) = h(k_j)\}$.

Under the assumption of simple uniform hashing, we have $\Pr\left(h(k_i) = h(k_j)\right) = 1/m$,

$1/m$ $\quad$ $1/m$ $\quad$ $1/m$ $\quad$ 1

With $\frac{1}{n}$ probability $\quad x_1$: $\quad x_n \to x_{n-1} \to \cdots \to x_4 \to x_3 \to x_2 \to x_1$

With $\frac{1}{n}$ probability $\quad x_2$: $\quad x_n \to x_{n-1} \to \cdots \to x_4 \to x_3 \to x_2 \to x_1$

…

With $\frac{1}{n}$ probability $\quad x_{n-1}$: $\quad x_n \to x_{n-1} \to \cdots \to x_4 \to x_3 \to x_2 \to x_1$

With $\frac{1}{n}$ probability $\quad x_n$: $\quad x_n \to x_{n-1} \to \cdots \to x_4 \to x_3 \to x_2 \to x_1$

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=1+1}^{n} X_{ij}\right)\right]$$

Case 2: Successful Search (Cont'd)

- Thus, the expected number of elements examined in a successful search is

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=1+1}^{n}X_{ij}\right)\right]=\frac{1}{\mathrm{n}}\sum_{i=1}^{n}\left(1+\sum_{j=1+1}^{n}E(X_{ij})\right)=\frac{1}{\mathrm{n}}\sum_{i=1}^{n}\left(1+\sum_{j=1+1}^{n}\frac{1}{m}\right)$$

$$=1+\frac{1}{mn}\sum_{i=1}^{n}(n-i)=1+\frac{n(n-1)}{2mn}=\ 1+\frac{\alpha}{2}-\frac{\alpha}{2n}.$$

- We add 1 step for the hashing $h(k)$, so, the total would be $2+\frac{\alpha}{2}-\frac{\alpha}{2n}$
- It implies that average case running time of search is $\Theta(1+\alpha)$

Simple Uniform Hashing

We made an important assumption: Simple Uniform Hashing

- All keys are evenly distributed to the m buckets of the hash table, so that the lengths of chains at each bucket are the same.

Can we really get this for real?

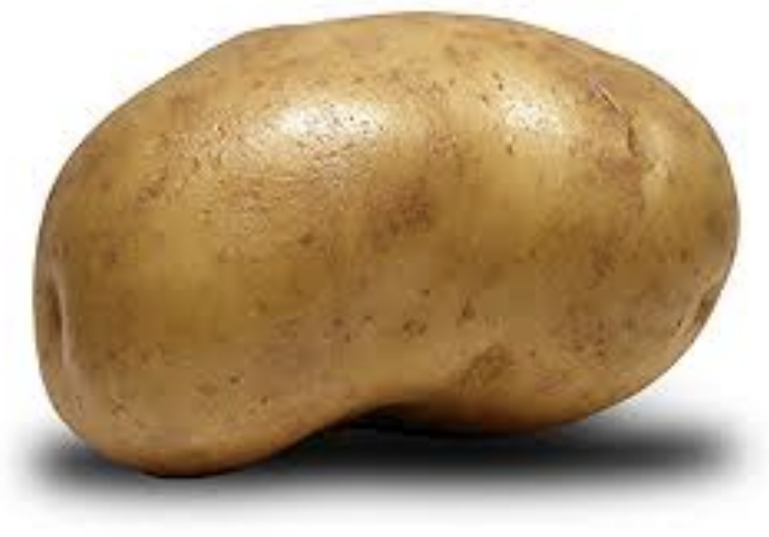
Difficult, but we try to be as close to it as possible.

Choose good hash functions

Why is it called hash?

A good hash function typically will scramble all of the bits of the input together in a way that appears totally random.

Hence the name "hash function."



Hash (noun): A mixture of meat, potatoes and vegetables into small pieces and baked and fried.

Uniform distributions for keys?

Think about inserting English words from a document into the hash table.

- We cannot really guarantee this in practice,
- We don't really know the distribution from which the keys are drawn. e.g., we cannot really tell which English words will actually be inserted into the hash table before we go through the whole document.
- So, there is no way to choose a hash function beforehand that guarantees all chains will be equally long (simple uniform hashing).

So, what can we do?

- We use some **heuristics**.

Heuristic is any approach that does not guaranteed to be optimal or perfect, but it works well in practice.

Remark

- Every object stored in a computer can be represented by a bit-string (string of 1's and 0's), which corresponds to a (large) integer, i.e., any type of key can be converted to an integer easily.
- So the only thing a hash function really needs to worry about is how to map these large integers to a small set of integers $\{0, 1, \ldots, m-1\}$, i.e., the buckets.

Good hash functions

1. h(k) depends on every bit of k

$h(k) = \text{lowest 3-}bits\ of\ k$ e.g., $h(1001010100011 01) = 5$

$h(k) = \text{sum of all bits}$ e.g., $h(100101010001101) = 7$ 👍

2. h(k) "spreads out" values

Assume there are $m = 1000$ buckets in the hash table

$h(k) = k\ mod\ 2$

$h(k) = k\ mod\ 1000$ 👍

3. h(k) is efficient to compute

$h(k) =$ solution to the PDE(partial differential equation) with parameter k

$h(k) = k\ mod\ 263$ 👍

In practice, it is difficult to get all three of them, but there are some heuristics that work well.

Two methods for hashing

1. Division method
2. Multiplication method

Division Method

- **Idea:**
  - Map a key k into one of the m slots by taking the remainder of k divided by m

    $h(k) = k \bmod m$

- **Advantage:**
  - fast, requires only one operation

- **Disadvantage:**
  - Certain values of m are bad, e.g.,
    - power of 2
    - non-prime numbers

Division Method -Example

- If $m = 2^p$, then $h(k)$ is just the least significant p bits of k
  - $p = 1 \rightsquigarrow m = 2$

    $h(k) = \{0,1\}$, least significant 1 bit of k
  - $p = 2 \rightsquigarrow m = 4$

    $h(k) = \{0,1,2,3\}$, least significant 2 bits of

- Choose m to be a prime, not close to a power of 2
  - $H_1(k) = k \bmod 97$
  - $H_2(k) = k \bmod 100$

  $H_1(25566) = 55$, $H_1(26966) = 0$

  $H_2(25566) = 66$, $H_2(26966) = 66$

	mod 97	mod 100
16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

Multiplication Method

Idea:

1. Multiply key k by a constant A, where $0 < A < 1$
2. Extract the fractional part of kA
3. Multiply the fractional part by m
4. Take the floor of the result

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m\ (k\ A\ mod\ 1) \rfloor$$

- **Disadvantage:** Slower than division method
- **Advantage:** Value of m is not critical, e.g., typically 2^p

Multiplication Method - Example

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m\ (k\ A\ mod\ 1) \rfloor$$

The value of m is not critical now (e.g. $m = 2^p$)

Assume $m = 2^3$

A = .101101

$k = 110101$

$A \times k = 1001010.0110011$

Discard 1001010

Shift 0110011 by 3 bits to the left

011.0011

Take integer part: 011 Thus $h(11010) = 011$

-

Open Addressing

another way of resolving collisions other than chaining

Open addressing

There is no chain

Then what to do when having a collision?

- Find another bucket that is free

How to find another bucket that is free?

- We probe.

How to probe?

- linear probing
- quadratic probing
- double hashing

Note: None of these methods can generate more than m^2 different probing sequences!

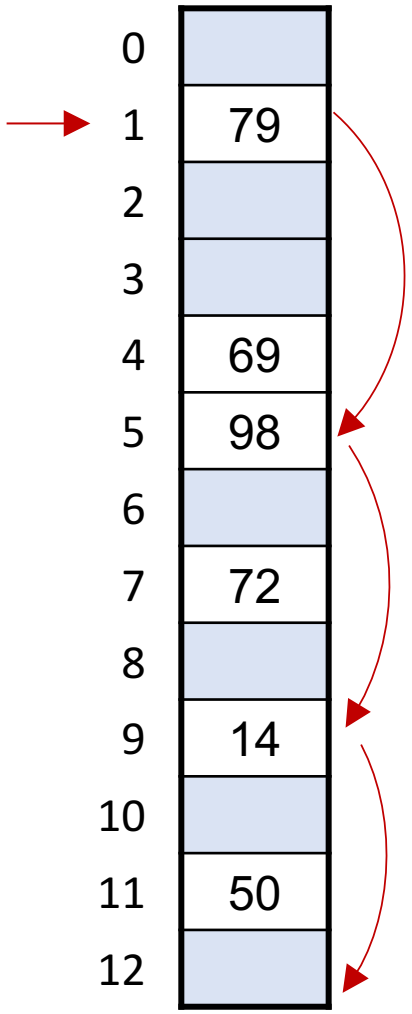
Open addressing

- If we have enough contiguous memory to store all the keys

  $(m > n) \Rightarrow$ store the keys in the table itself

No need to use linked lists anymore

- Basic idea:
  - Insertion: if a slot is full, try another one, until you find an empty one
  - Search: follow the same sequence of probes
  - Deletion: more difficult … (we'll see why)
- Search time depends on the length of the probe sequence!



Generalize hash function notation:

- A hash function contains two arguments now
  - Key value, and
  - Probe number

    $h(k, p), \quad p = 0,1, \ldots, m-1$

$< 1, 5, 9, 0, 4, 8, 12, 3, 7, 11, 2, 6, 10 >$

$< 1, 4, 7, 10, 0, 3, 6, 9, 12, 2, 5, 8, 11 >$

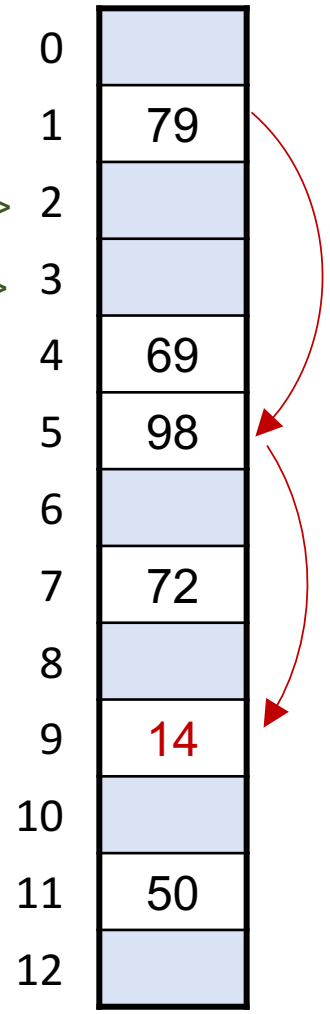
- Probe sequences

$$< h(k, 0), h(k, 1), \ldots, h(k, m-1) >$$

Must be a permutation of $< 0,1, \ldots, m-1 >$

- There are $m!$ possible permutations
- Good hash functions should be able to produce all $m!$ probe sequences

Example: insert 14



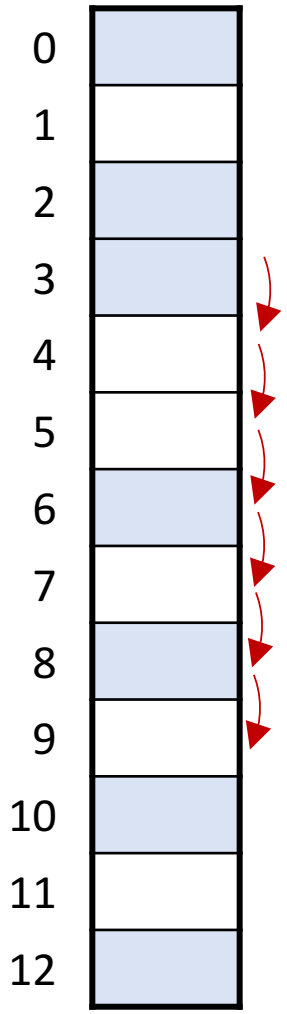
Linear Probing: Inserting a key

- **Idea**: when there is a collision, check the next available position in the table (i.e., probing)

$$h(k,i) = (h_1(k) + i) \bmod m, \qquad i = 0,1,2, \ldots$$

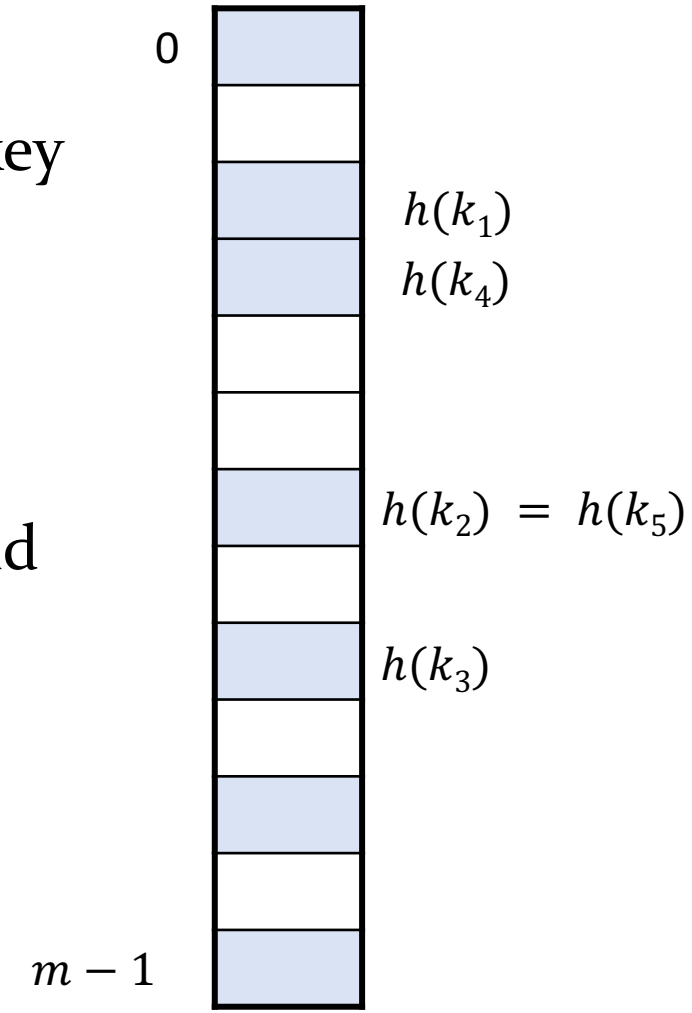
- First slot probed: $h_1(k)$
- Second slot probed: $h_1(k) + 1$
- Third slot probed: $h_1(k) + 2$, and so on

- probe sequence: $< h_1(k), h_1(k) + 1, h_1(k) + 2, \ldots . >$
- Can generate m probe sequences maximum, why?



Linear Probing: Searching a key

- Three cases:
  1. Position in table is occupied with an element of equal key
  2. Position in table is empty
  3. Position in table occupied with a different element
- Case 2: probe the next higher index until the element is found or an empty position is found
- The process wraps around to the beginning of the table

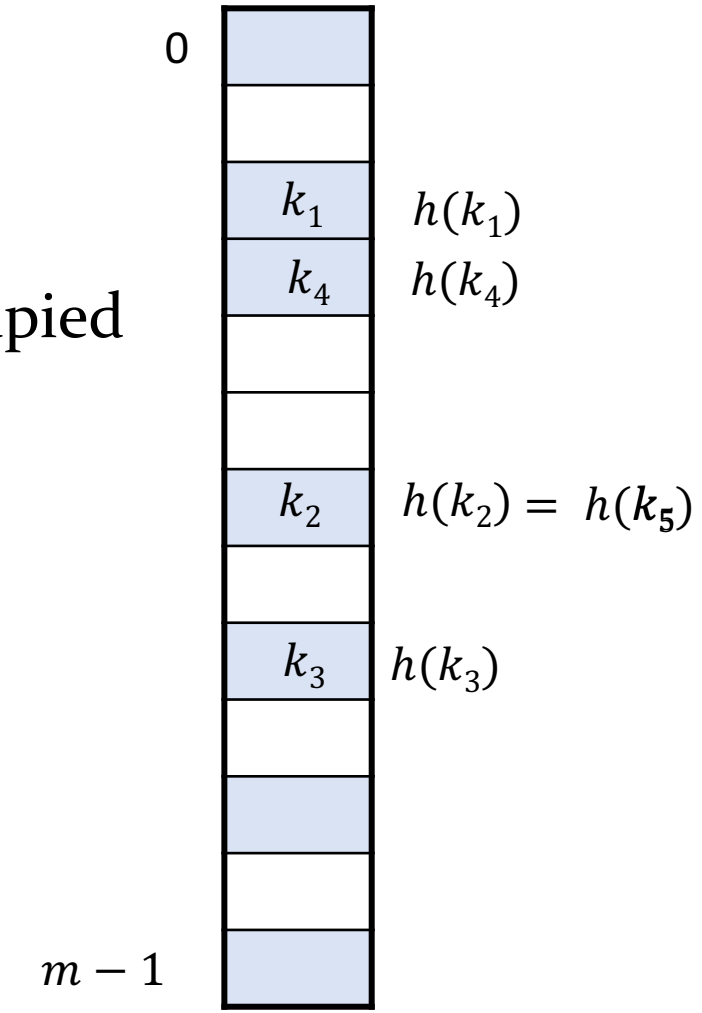


Linear Probing: Deleting a key

- Problems
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED

✓ The deleted slot can later be used for insertion

✓Searching will be able to find all the keys

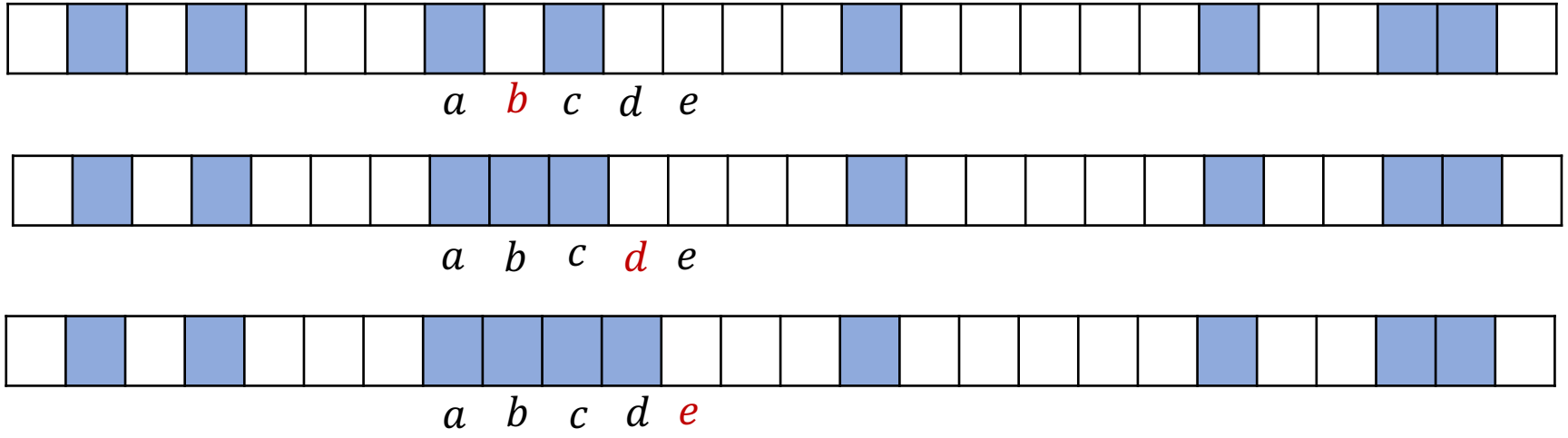


Primary Clustering Problem

- Some slots become more likely than others
- Long chunks of occupied slots are created

$\Rightarrow$ search time increases!!

initially, all slots have probability 1/m



Slot b: $2/m$

Slot d: $4/m$

Slot e: $5/m$

Quadratic Probing

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \; mod \; m$

where h': $U \rightarrow (0,1,\ldots,m-1)$, c_1 and c_2 constants

- Clustering problem is less serious but still an issue (secondary clustering)
- Need to be careful with the values of c_1 and c_2, it could jump in such a way that some of the buckets are never reachable.

- How many probe sequences quadratic probing generate? m
  (the initial probe position determines the probe sequence)

Double hashing

1. Use one hash function to determine the first slot

2. Use a second hash function to determine the increment for the probe sequence

$$h(k,i) = (h_1(k) + i\, h_2(k))\ mod\ m, \qquad i = 0,1,\ldots$$

   - Initial probe: $h_1(k)$
   - Second probe is offset by $h_2(k)$ mod m, so on …

- Advantage: avoids clustering

- Disadvantage: harder to delete an element

  Can generate m^2 probe sequences maximum

Double hashing - Example

$h_1(k) = k \bmod 13$

$h_{2(k)} = 1 + (k \bmod 11)$

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 13$

- Insert key 14:

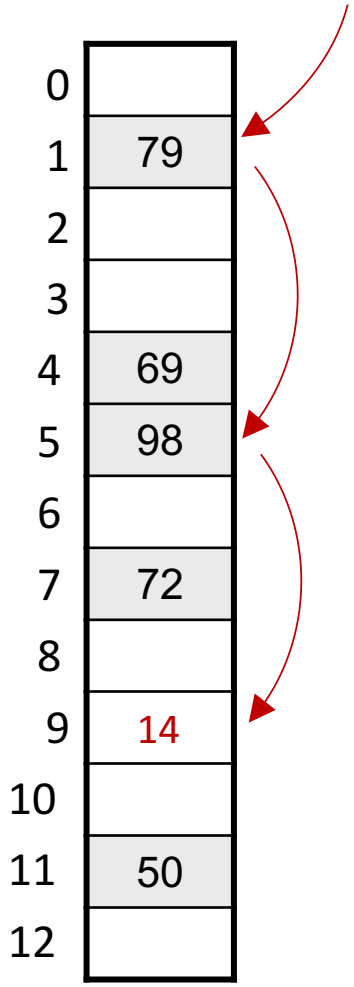
$h_1(14,0) = 14 \bmod 13 = 1$

$h(14,1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

$h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

$= (1 + 8) \bmod 13 = 9$



Analysis of Open Addressing

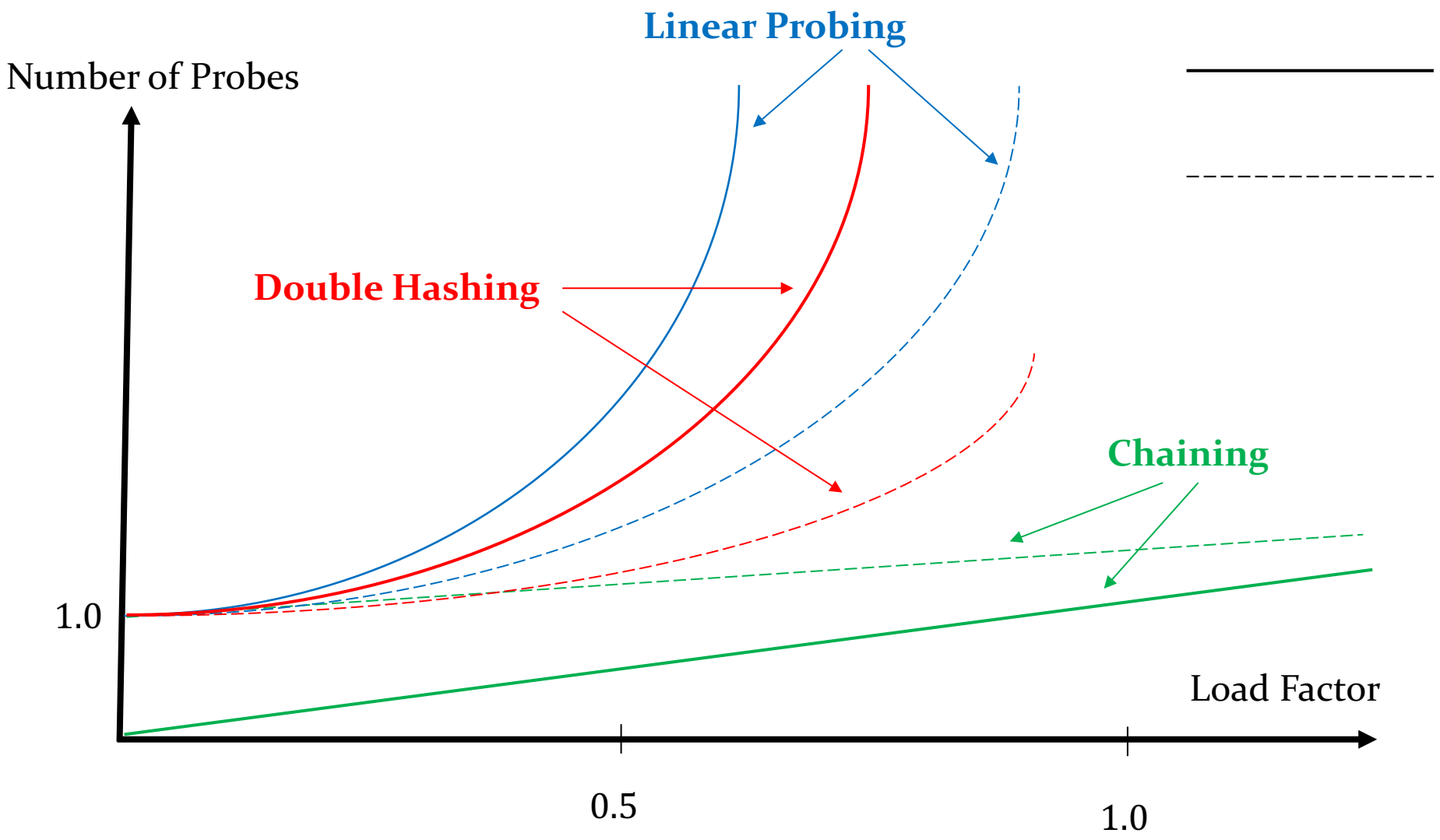
Assuming simple uniform hashing, the average-case number of probes in an unsuccessful search is $1/(1-\alpha)$.

For a successful search it is $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

In both cases, assume $\alpha < 1$

Open addressing cannot have $\alpha > 1$. Why?

Expected number of probes vs. load factor



Questions