Computer Science 1001.py, Lecture 15[†]

Cuckoo Hashing
String Matching, Exact Sequences Comparison

Instructor: Benny Chor

Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Fall Semester, 2011/12
http://tau-cs1001-py.wikidot.com

Lecture 14 Highlights

- Floating point (bounded precision) numbers.
- The dictionary problem (find, insert, delete).
- Hash functions and hash tables.
- Resolving collisions using chaining.
- Measuring maximum load experimentally: Small and large cases.

Lecture 15, Plan

- Employing hashing to solve strings problems on very long strings (*e.g* proteomes and/or genomes).
- Cuckoo hashing (find operations in $O(1)$ worst case time).

- Representing characters: Ascii and Unicode.
- String matching.

Insertion, Deletion, Find for Hash Tables with Chaining

We have implemented the operations insert and find.

We deliberately did not implement deletion. But by now, this should be an easy exercise for you.

Suppose we maintain a dynamic hash table with m entries.
At every point in the execution, the number of items in the table, n, does not exceed m.
Under these conditions, insertion, deletion, and find are performed in expected time $O(1)$ per operation, and worst case time $O(\log n / \log \log n)$ per operation, with high probability (probability is over choices of inputs).

Using Hashing To Solve Large String Problems

We will discuss (on the board) the naïve solution to the following homework problem (last year), which is quadratic in the length of the input strings, thus totally infeasible if both strings are millions characters long. We will then outline how hashing with chaining can be used to yield a much more efficient algorithm.

1. באתר תמצאו שמונה קבצים, המתארים כ"א את הפרוטאום של מין שונה. הפרוטאום הוא אוסף החלבונים הידועים באותו מין. ביונקים, מספר החלבונים הוא בסביבות שלושים אלף. כל חלבון מתואר כמחרוזת מעל א"ב בן 20 אותיות, שהן החומצות האמיניות. אורך אופייני של סדרה כזו הוא כמה מאות. לפני כל סדרה מופיעה הערה המתחילה בתווים >gi, למשל

```
>gi|91680537|ref|NP_788842.2| stanniocalcin 1 [Bos taurus]
MLQNSAVLLVLVISASATHEAEQNDSVSLRKSRVAAQNSAEVIRCLNSALQVGCGAFACLENSTCDTDGM
YDICKSFLYSAAKFDTQGKAFVKESLKCIANGVTSKVFLAIRRCSTFQRMIAEVQEECYTKLNVCSVAKR
NPEAITEVVQLPNHFSNRYYNRLVRSLLDCDEDTVSTIRDSLMEKIGPNMASLFHILQTDHCAHTQQRAD
FNRRRANEPQKLKVLLRNLRGEVASPSHIKRTSQESA
>gi|58330890|ref|NP_001010991.1| transglutaminase 1 [Bos taurus]
MPSPTPSSGGPRSDVGRWGGNPWQPPTTPSPEPEPEPDRRSRRGGRSFWARCCGCCSCRNTEDDDWGPER
...
```

וכמובן שהיא אינה חלק ממחרוזת החלבון.

(א) מצאו את המחרוזת הקצרה ביותר (מעל א"ב החלבונים) אשר אינה מופיעה (ברצף) באף אחד משמונת הפרוטאומים. מה אורכה? האם זה מפתיע?

(ב) בחרו באקראי שלושה מינים מתוך השמונה. לכל זוג מינים מהשלושה, מצאו את חמש הסדרות הארוכות ביותר המשותפות לזוג הפרוטאומים.[1] כמו כן, מצאו את חמש הסדרות הארוכות ביותר המשותפות לשלושת הפרוטאומים. מה אורך הסדרות? האם זה מפתיע?
בונוס: מיהם החלבונים שמצאתם?

Large String Problems: naïve approach

Let us look at the second task. We are given two strings $\mathtt{S}$ and $\mathtt{T}$, both of length n. We want to find (contiguous) substrings of maximal length that are exactly shared by $\mathtt{S}$ and $\mathtt{T}$.

We do not know in advance what this maximal length will be. Let us say we are now looking for substrings of length ℓ.

The naïve method is to compare all substrings of length ℓ from $\mathtt{S}$ with all substrings of length ℓ from $\mathtt{T}$. A single comparison involves exactly ℓ character comparisons. The number of such substrings in each of $\mathtt{S}$ and $\mathtt{T}$ is exactly $n - \ell$. So overall, the naïve approach takes $(n - \ell)^2 \cdot \ell \approx n^2 \cdot \ell$ operations (since $\ell \ll n$).

If $n \approx 10^6$ and $\ell \approx 10^2$, such task will take about 10^{14} operations and is completely infeasible.

Using Hashing To Solve Large String Problems: Insert

Let us consider a hashing based approach. We hash all substrings of length ℓ of $\mathtt{S}$ into a hash table with chaining. The key is the substring itself (with ℓ letters). The value is the location of this substring in $\mathtt{S}$.

How much time does this take? Hashing a substring costs $O(\ell)$ operations. The number of substrings is $n - \ell = O(n)$. The expected length of chains encountered in inserting a single element is $O(1)$. So overall, inserting all substrings to the hash table takes $O(n\ell)$ expected number of operations.

Using Hashing To Solve Large String Problems: Find

We now go over all length ℓ substrings of $\mathtt{T}$, one by one, and try to find each in the hash table: We find the hash value of each substring and look it up (checking equality) in the table.

The expected length of chains encountered in looking up a single substring is $O(1)$. Each comparison is ℓ operations, and computing the hash value itself takes $O(\ell)$ operations. All by all, to process onesubstring of $\mathtt{T}$ takes $O(\ell)$ operations. There are $n - \ell = O(n)$ substrings. So this finding phase takes $O(n\ell)$ operations on average.

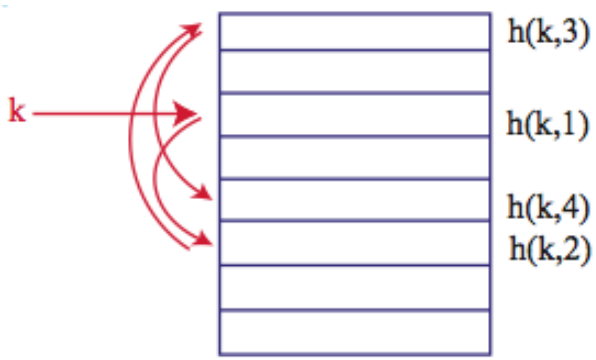
Both "global insert" and "global find" take $O(n\ell)$ operations on average (in fact the constant is 2 times the average length of chains in the table). If $n \approx 10^6$ and $\ell \approx 10^2$, such a task will take a small constant times 10^8 operations. This may cause your PC to cry, but is essentially feasible.

We did not say anything on how ℓ of maximal length substrings can be found. But something should be left to you.

Open Addressing

In open addressing, we require that each slot in the hash table contains at most one item. This obviously implies that n cannot be larger than m.

Furthermore, an item will typically not stay statically in the slot where it "tried" to enter, or where it was placed initially. Instead, it may be moved around a few times.



Open addressing is important in hardware applications where devices have many slots but each can only store one item. Fast switches and high capacity routers are examples for this.

There are many approaches to open addressing. We will describe a fairly recent one, termed cuckoo hashing (Pagh and Rodler, 2001).

Cuckoo Hashing: Motivation

We saw that for hashing with chaining, if $n \leq m$, hashing with chaining guarantees that insertion, deletion, and find are carried out in expected time $O(1)$ per operation, and worst case time $O(\log n / \log \log n)$ per operation, with high probability (probability is over choices of inputs).

In certain scenarios (*e.g.* fast routers in large internet nodes) we want find to run in $O(1)$ worst case time.

Compare this to $O(1)$ expected time and $O(\log n / \log \log n)$ worst case time of hashing with chaining.

Cuckoo hashing achieves this, but there are two prices to pay:

- Instead of $n \leq m$, we require $n \leq m/2$ or even $n \leq m/3$.
- The worst case time for insert may be somewhat longer.

Cuckoo Hashing

Cuckoo hashing uses two distinct hash functions, h_1 and h_2 (improved versions use three or four, but the idea is the same).

Each key, k, has two potential slots in the hash table, $h_1(k)$ and $h_2(k)$. If we search for k, all we have to do is look for it in these two locations (no chains here – at most one item per slot).

It is slightly more involved to insert a record whose key is k.

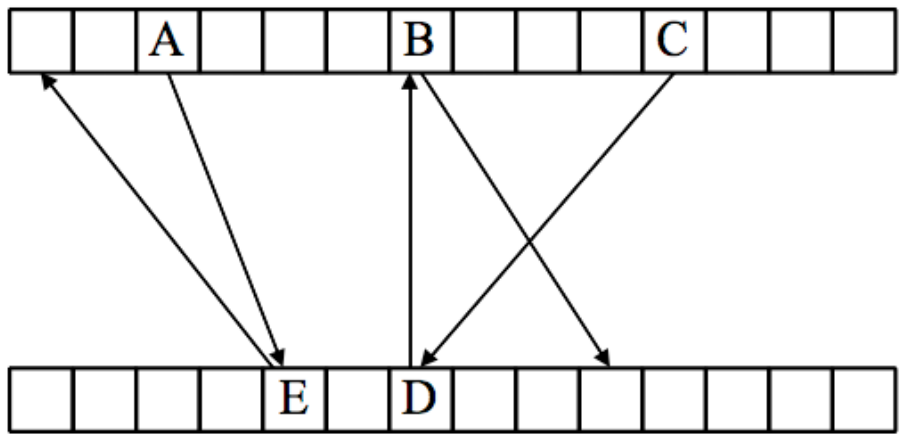
Cuckoo Hashing

It is slightly more involved to insert a record whose key is k.

- If any of the two slots, $h_1(k)$ or $h_2(k)$ is empty, k is inserted there.
- If both slots are full, pick one of the two occupants, say ℓ. Place k in ℓ's current slot.
- Assume this was location $h_1(\ell)$. Place ℓ in its other slot, $h_2(\ell)$.
- If that slot was empty, we are done.
- Otherwise, the slot is occupied by some g. Place this g in its other slot, potentially kicking out its present occupant, etc., etc., until we find an empty slot.

Cuckoo Hashing: Examples

(modified from a ppt presentation by Michael Mitzenmacher, Harvard Univ.)



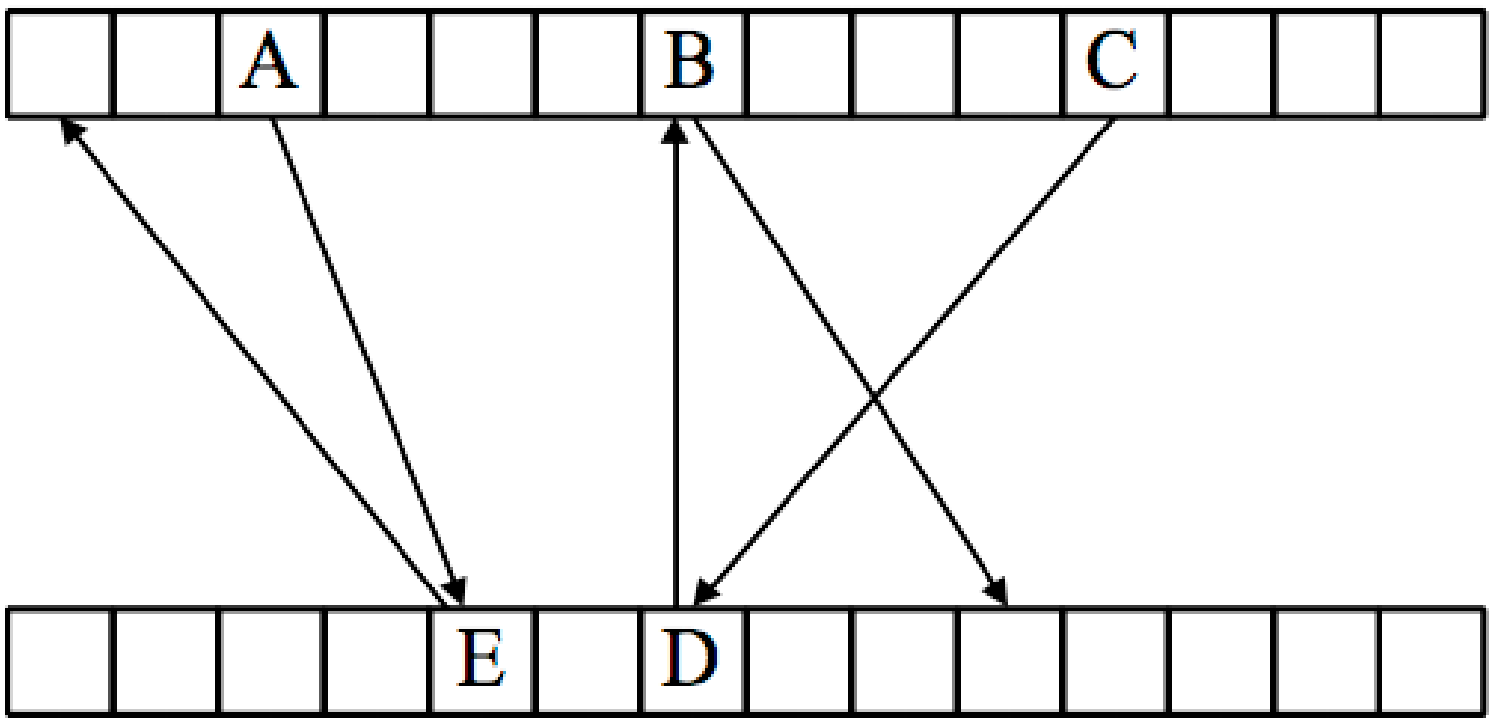
The top row represent the first half of the hash table, $m/2$ slots . The bottom row represent the second half of the hash table, $m/2$ slots. The two parts do not have any slot in common[‡].

The top row corresponds to the mapping according to $h_1(k)$, the bottom to $h_2(k)$. Arrows indicate where key k will move if bumped from its current location.

[‡]thanks to Roee Asher, whose suggestion we adopt here.

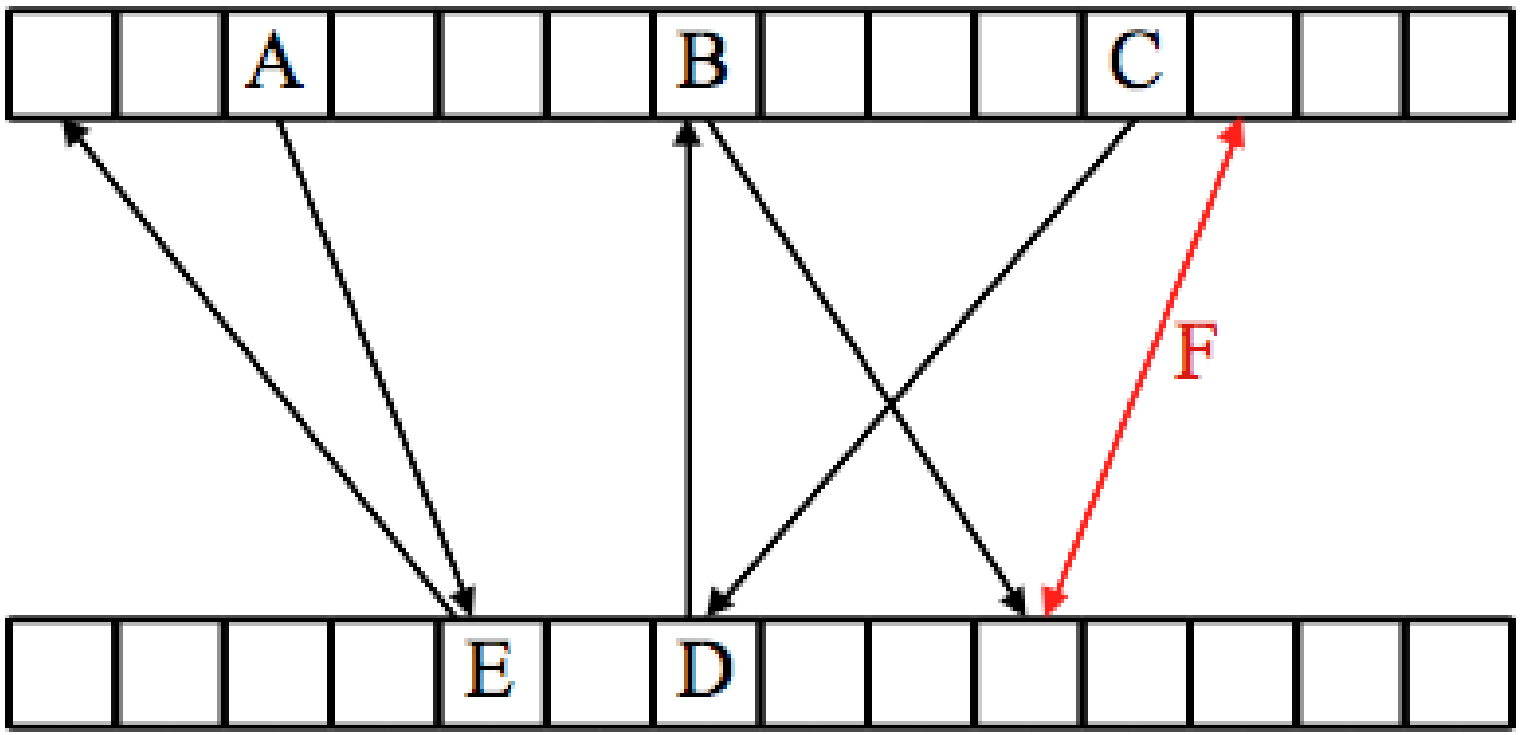
Cuckoo Hashing: Examples

(modified from a ppt presentation by Michael Mitzenmacher, Harvard Univ.)



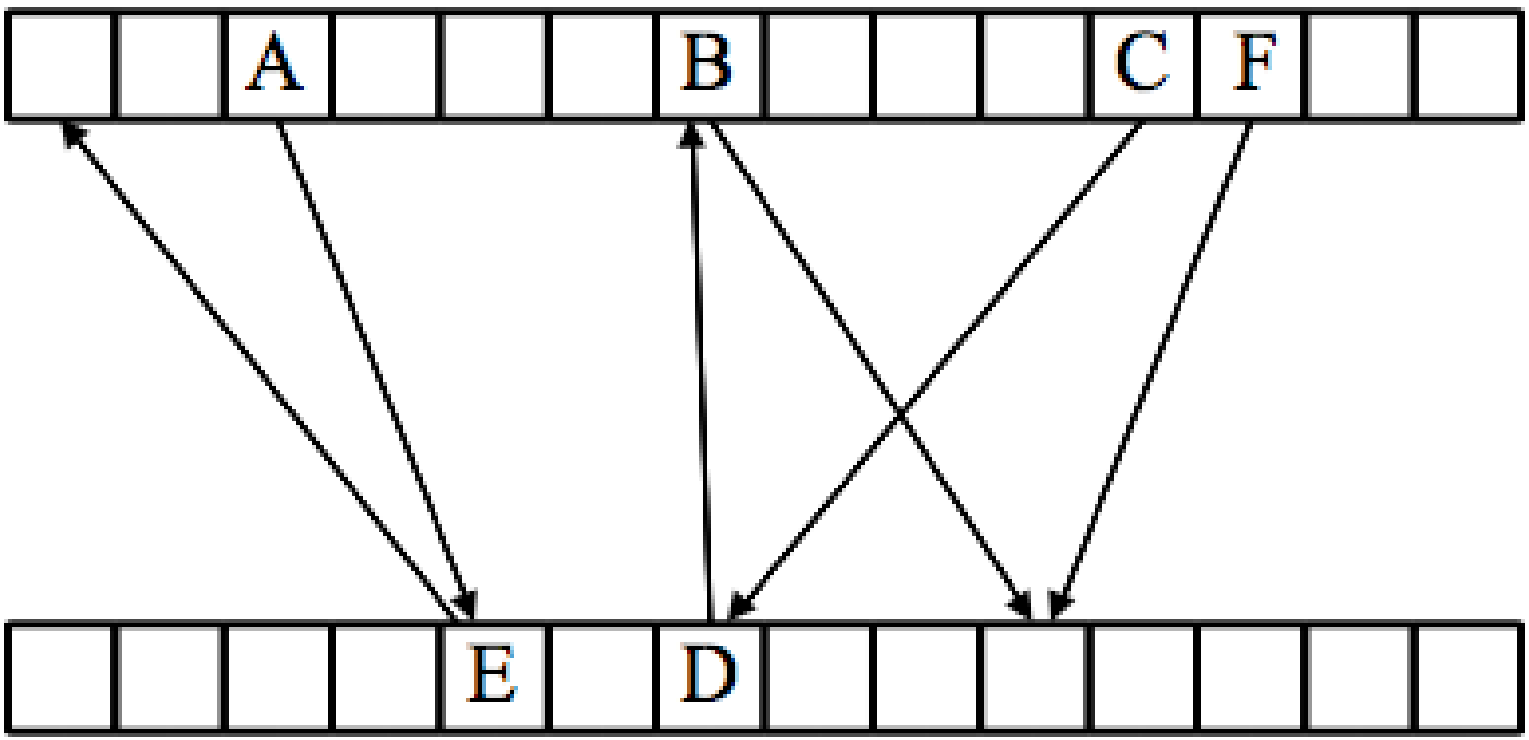
Cuckoo Hashing: Examples

(modified from a ppt presentation by Michael Mitzenmacher, Harvard Univ.)



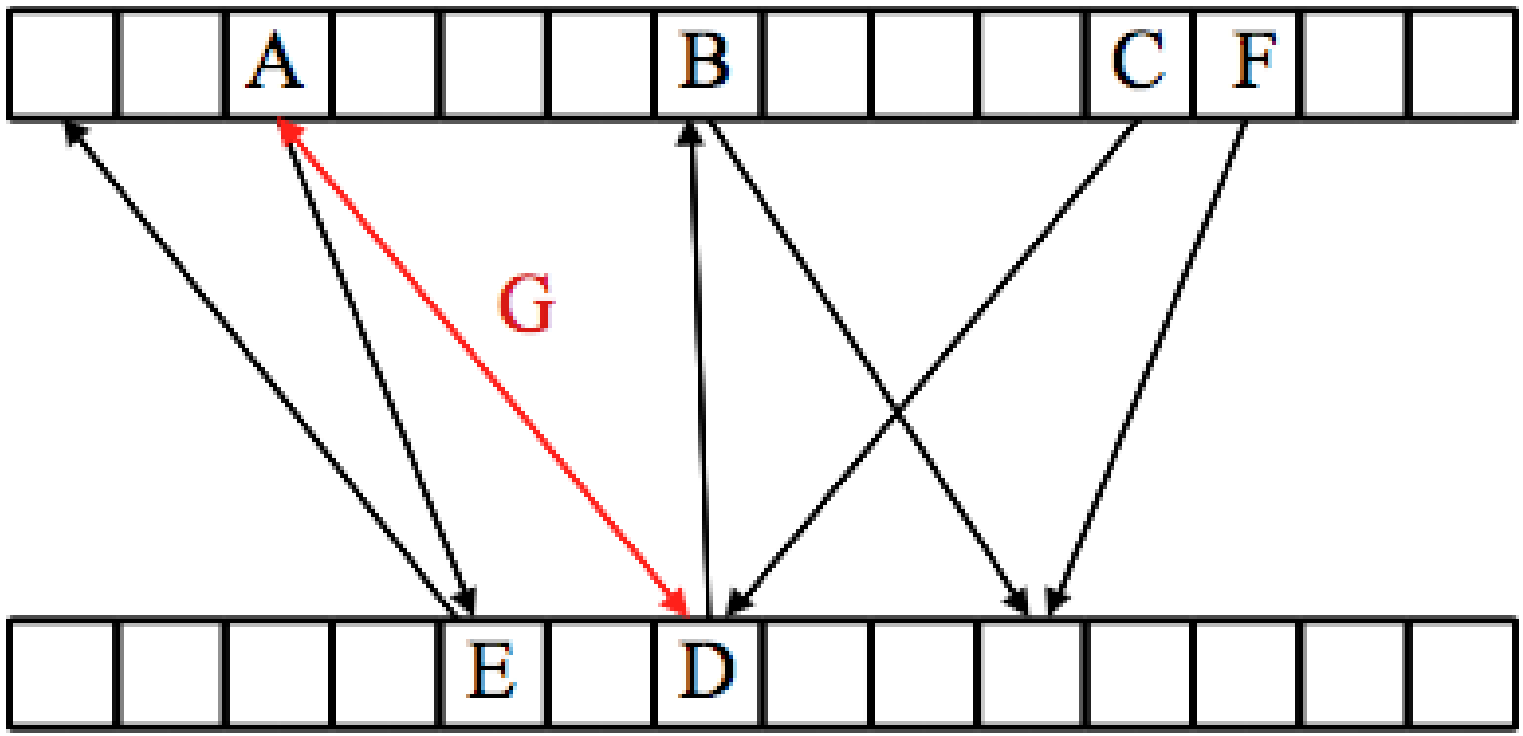
Cuckoo Hashing: Examples

(modified from a ppt presentation by Michael Mitzenmacher, Harvard Univ.)



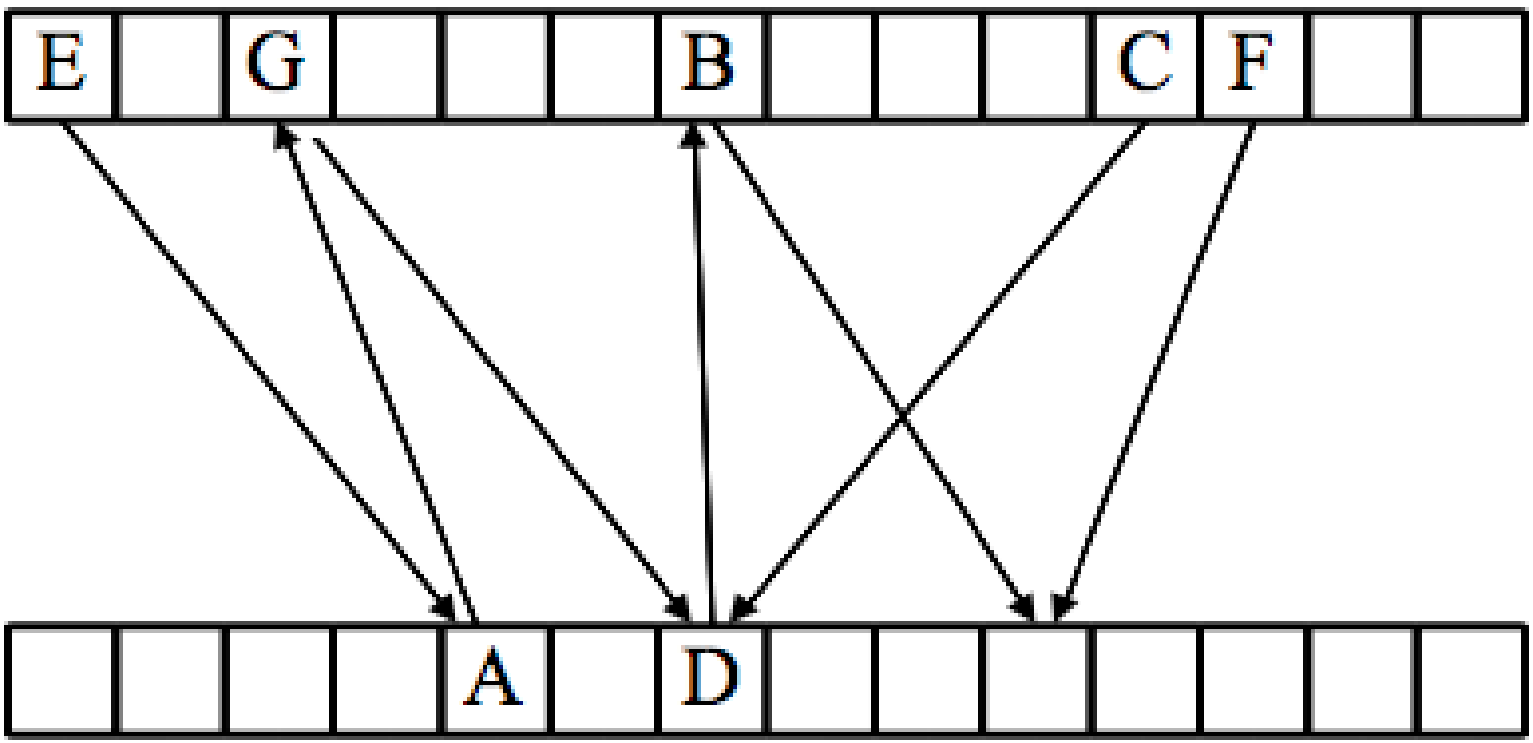
Cuckoo Hashing: Examples

(modified from a ppt presentation by Michael Mitzenmacher, Harvard Univ.)



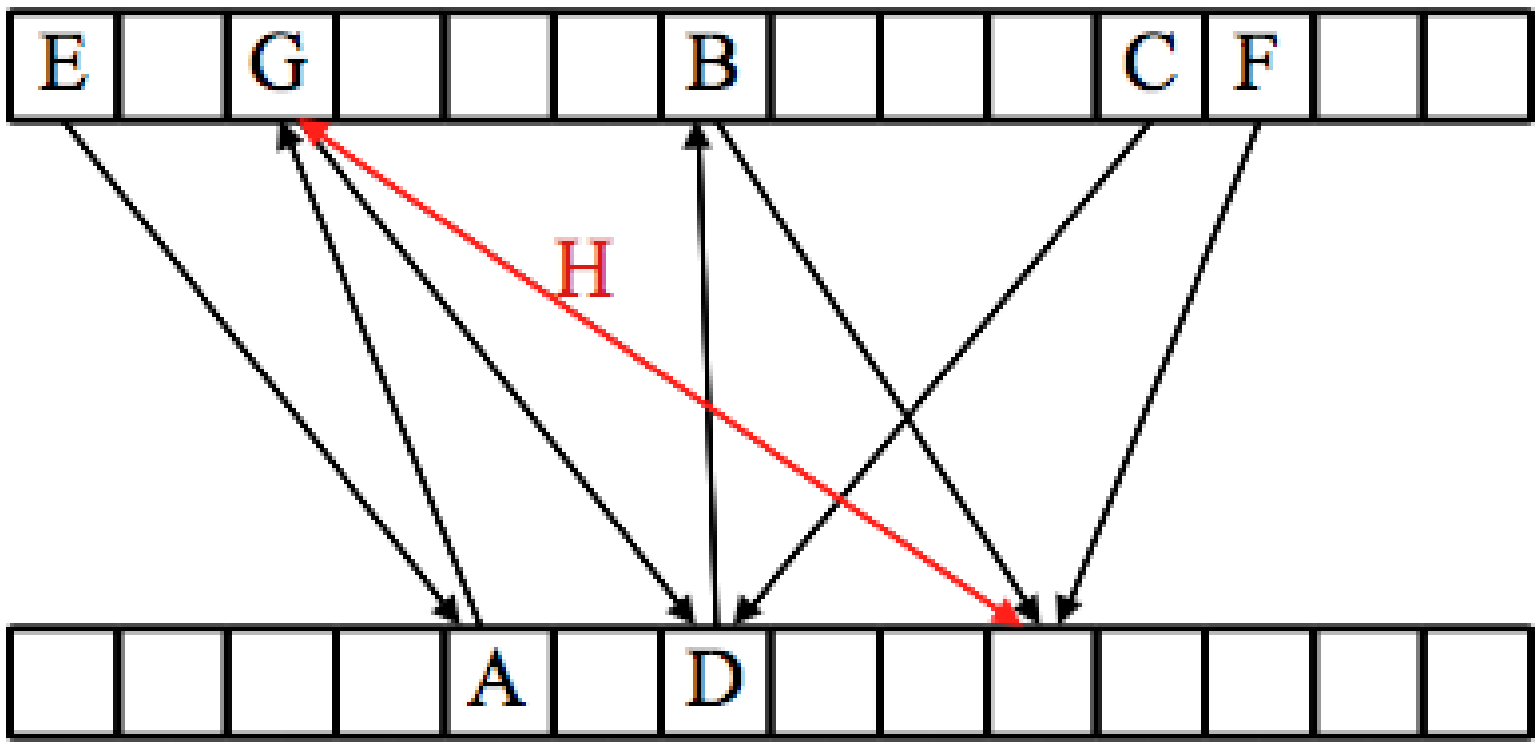
Cuckoo Hashing: Examples

(modified from a ppt presentation by Michael Mitzenmacher, Harvard Univ.)



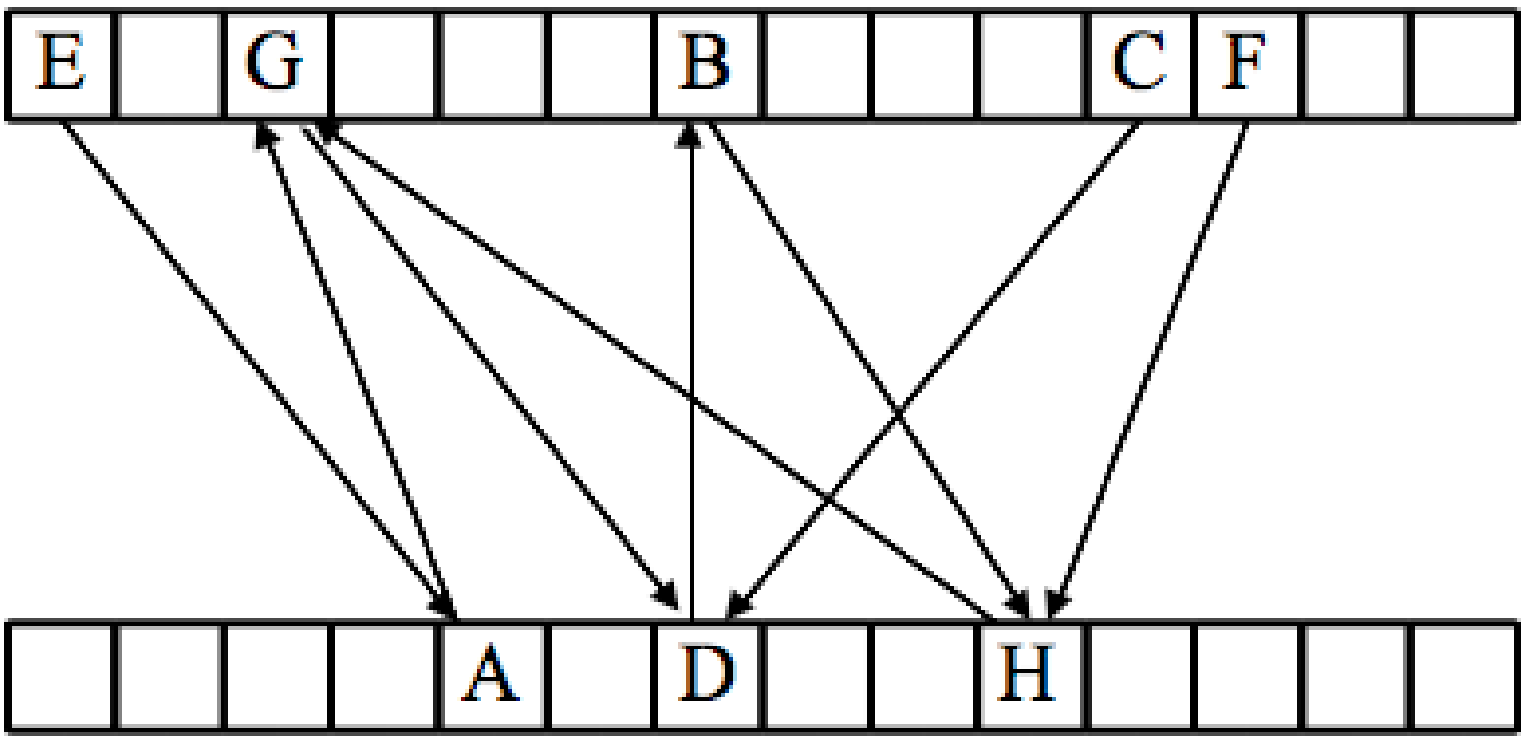
Cuckoo Hashing: Examples

(modified from a ppt presentation by Michael Mitzenmacher, Harvard Univ.)



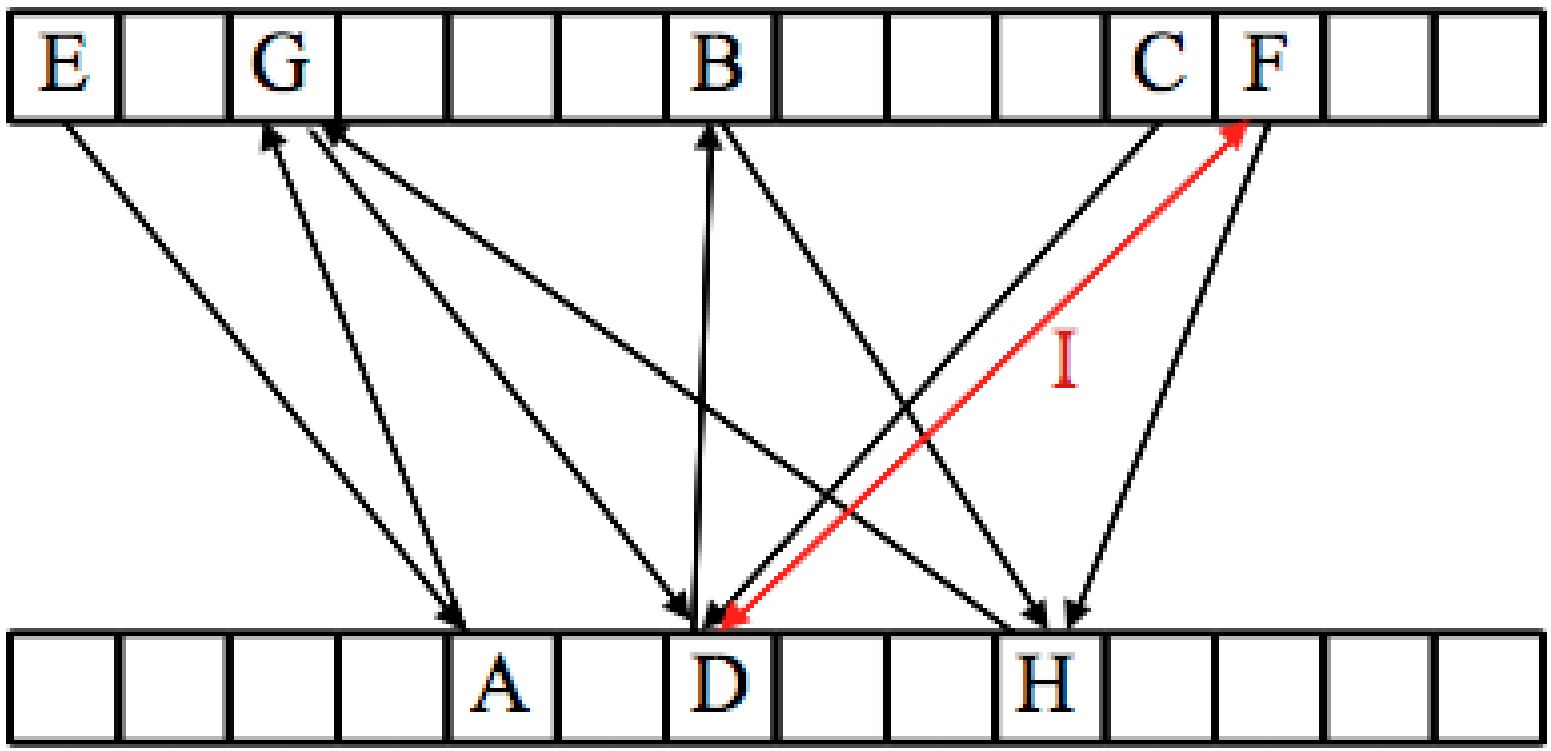
Cuckoo Hashing: Examples

(modified from a ppt presentation by Michael Mitzenmacher, Harvard Univ.)



Cuckoo Hashing: Examples

(modified from a ppt presentation by Michael Mitzenmacher, Harvard Univ.)



Cuckoo Hashing

In the last example, we have reached a cycle: Any of these nine items we try to insert when the other eight are already present kicks out another item, and we are in a never ending loop. This obviously is very bad news for our cuckoo hashing.

The underlying reason for such a deadlock is that we have nine items, `A, B, C, D, E, F,G, H, I`, such that the union of all their potential locations is just eight slots.

Notice that this is not a very likely event. With very high probability, the eighteedteen values $(18 = 9 \cdot 2)$ $h_1(A), h_1(B), ..., h_1(I)$, $h_2(A), h_2(B), ..., h_2(I)$ will atain more than , and certainly not strictly less than nine distinct values.

Cuckoo Hashing, cont.

Another possible problem is that there will be no cycle, but the path leading to the successful insertions will be very long.

Fortunately, such unfortunate cases occur with very low probability when the load factor, *i.e.* n/m, is sufficiently low. The common recommendation is to have $n/m < 1/2$.

A theoretical solution: In case of failure (or a very long path), rehash using "fresh hash functions".

A more practical solution: Maintain a very small excess zone (*e.g.* 5 excess slots for a hash table with m=10,000 slots) and place items causing trouble there. If regular search fails, search the excess zone as well.

Cuckoo Hashing Implementation and Analysis

Left as a home assignment.

Hash Functions: Wrap Up

- Hash functions map large domains $\mathcal{X}$ to smaller ranges $\mathcal{Y}$.
- Example: $h : \{0, 1, \ldots, p^2\} \mapsto \{0, 1, \ldots, p-1\}$, defined by $h(x) = a \cdot x + b \mod p$.
- Hash tables are extensively used for searching.
- If the range is smaller than the domain, there will be collisions ($x \neq y$ with $h(x) = h(y)$). In the example above, if $x_1 = x_2 (\mathrm{mod}\, p)$ then $h(x_1) = h(x_2)$.
- A good hash function should create few collisions for most subsets of the domain ("few" is relative to the size of subset).

Using Hash Functions and Tables: Wrap Up

- We explained chaining as a way to resolve collisions.
- In the data structures course, you will see additional means for collision resolution – open addressing, double hashing, etc.
- We also studied the paradigm of cuckoo hashing, using two hash functions $h_1(\cdot),\ h_2(\cdot)$ (or three, or four).

Text, Strings and Characters

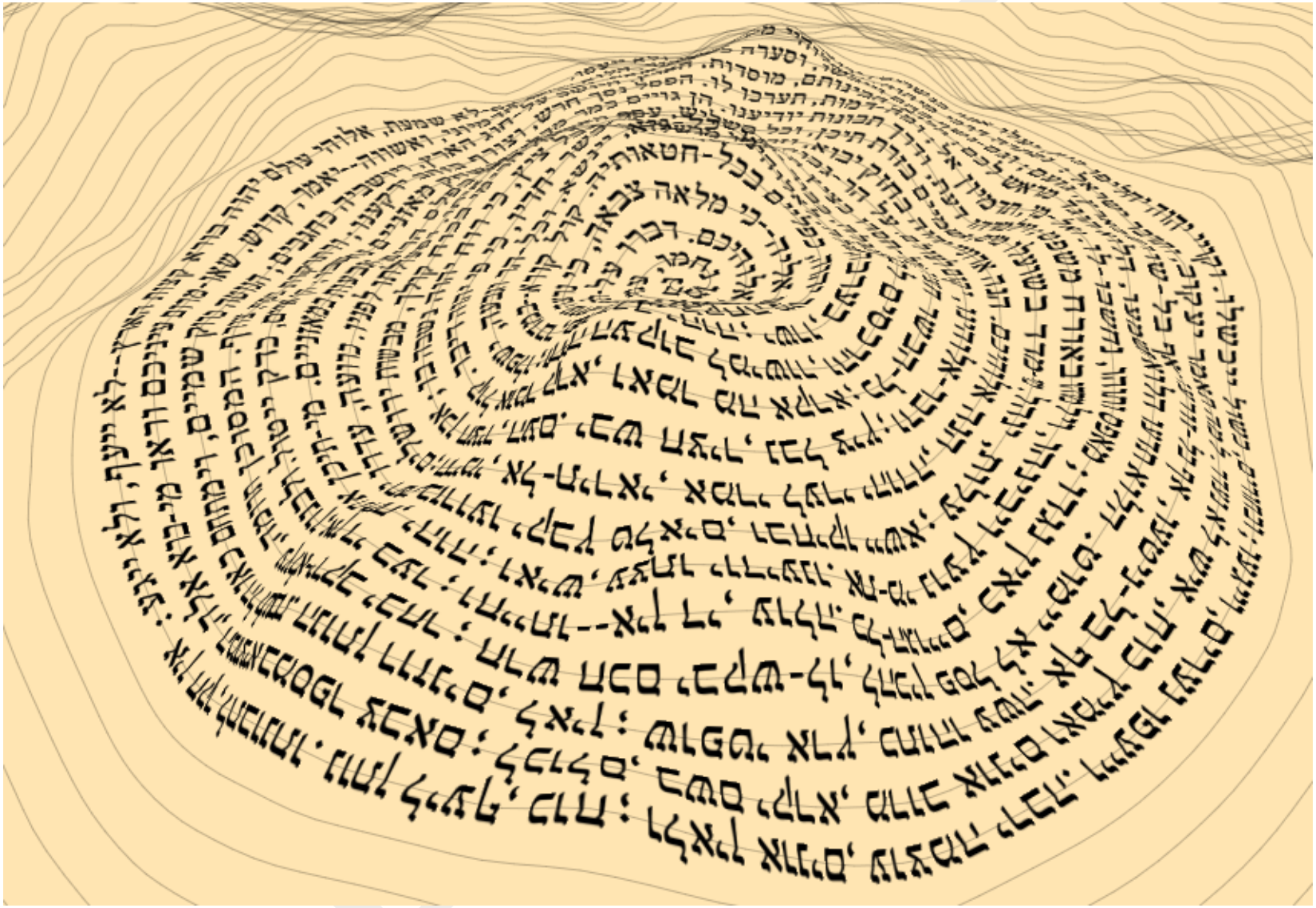


image from http://chronotext.org/Isaiah/

Strings and Characters

Much of the daily information we deal with is textual. In the written world, it is represented as text. In computer science terminology, text is a sequence, termed a string, of characters over a finite alphabet, Σ.

The alphabet, Σ, consists of letters. Different alphabets have different sets of letters. For example, consider the English alphabet $\{a, b, \ldots, z\}$, the Greek alphabet $\{\alpha, \beta, \ldots, \omega\}$, or the Hebrew alphabet $\{\aleph, \beth, \gimel, \ldots\}$.

In some cases there are different variants of the same letter (capital vs. lower case, printed vs. written, etc.).

In addition to letters, texts contain numerals, punctuation marks, spaces, etc.

String Matching

When we edit a text document we continuously modify a string. Other than typing, the second most common operation while editing is probably search, *i.e.* looking for a (relatively short) pattern within a (typically much longer) text. This problem is known as string matching.

In this lecture we will discuss strings in general, and their representation in the computer in particular. We will then explore three different algorithmic approaches to string matching.

An Example (courtesy of Charles Lutwidge Dodgson)

Text:

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

"Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!"

He took his vorpal sword in hand:
Long time the manxome foe he sought –
So rested he by the Tumtum tree,
And stood awhile in thought.

And, as in uffish thought he stood,
The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
And burbled as it came!

One, two! One, two! And through and through
The vorpal blade went snicker-snack!
He left it dead, and with its head
He went galumphing back.

"And, has thou slain the Jabberwock?
Come to my arms, my beamish boy!
O frabjous day! Callooh! Callay!'
He chortled in his joy.

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

Pattern: gyre and gimble

First Match

Text:

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

"Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!"

He took his vorpal sword in hand:
Long time the manxome foe he sought –
So rested he by the Tumtum tree,
And stood awhile in thought.

And, as in uffish thought he stood,
The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
And burbled as it came!

One, two! One, two! And through and through
The vorpal blade went snicker-snack!
He left it dead, and with its head
He went galumphing back.

"And, has thou slain the Jabberwock?
Come to my arms, my beamish boy!
O frabjous day! Callooh! Callay!'
He chortled in his joy.

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

Pattern: gyre and gimble

And a Second One

Text:

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

"Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!"

He took his vorpal sword in hand:
Long time the manxome foe he sought –
So rested he by the Tumtum tree,
And stood awhile in thought.

And, as in uffish thought he stood,
The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
And burbled as it came!

One, two! One, two! And through and through
The vorpal blade went snicker-snack!
He left it dead, and with its head
He went galumphing back.

"And, has thou slain the Jabberwock?
Come to my arms, my beamish boy!
O frabjous day! Callooh! Callay!'
He chortled in his joy.

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

Pattern: gyre and gimble

Jabberwocky (Through the Looking-Glass, and What Alice Found There)

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

"Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!"

He took his vorpal sword in hand:
Long time the manxome foe he sought –
So rested he by the Tumtum tree,
And stood awhile in thought.

And, as in uffish thought he stood,
The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
And burbled as it came!

One, two! One, two! And through and through
The vorpal blade went snicker-snack!
He left it dead, and with its head
He went galumphing back.

"And, has thou slain the Jabberwock?
Come to my arms, my beamish boy!
O frabjous day! Callooh! Callay!'
He chortled in his joy.

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

Lost in Translation?

Here is a partial translation, by Yuval Pinter

Twas brillig, and the slithy toves'	עֵת מַרְתֵּחַ הָיְתָה, הַלְּטוֹרוֹת
;Did gyre and gimble in the wabe	הַגְּמוּסוֹת
,All mimsy were the borogoves	סָבְּבוּ וְרָדְעוּ בְּחֵיק הַנַּרְחָק;
.And the mome raths outgrabe	כֹּה מְעוּדָּשׁוֹת הָיוּ הַבָּרְגּוֹסוֹת,
	וְהַזּוֹרְחִים גָּעֲמוּ בַּפַּרְסָתָק.
!Beware the Jabberwock, my son"	"הִשָּׁמֵר מֵהַלַּהֲגָרְן, בְּנִי יָקִיר!
The jaws that bite, the claws that	מֶלְתַּעַת נוֹגֶסֶת, צִפָּרֶן בּוֹצַעַת!
!catch	גּוֹרָה תָּגוּר מֵחוּבְחוּב, וְתָדִיר
Beware the Jubjub bird, and shun	אֶת מִלְהֶטֶת הַמַּעַף שֶׁהִיא
"!The frumious Bandersnatch	הַלַּפְתּוֹץ!"
:He took his vorpal sword in hand	נָטַל הוּא חַרְבּוֹ הַשְּׁנוֹטָה לְיָדוֹ:
Long time the manxome foe he	זְמָן רַב תָּר אָחַר הַצּוֹרֵר הַמַּצְלֵף –
—sought	אָחַר-כָּךְ הוּא נָח תַּחַת עֵץ הַגְּרִידוֹ,
,So rested he by the Tumtum tree	עָמַד בְּעוֹמְדוֹ לִזְמָן-מָה וְשָׂרְעָף.
.And stood awhile in thought	

So, gibberish remains gibberish, even when translated to Hebrew.
(the complete translation can be found here.)

String Matching in Python

Python has a built-in string class and methods. Combined with the regular expression (re) package, these methods can be used to solve all our string matching needs neatly and efficiently.

```
>>> import re
>>> occurances=re.finditer("ab", "abcabcaabbc") # pattern is ab
# creates an iterator with all occurances of pattern in text
>>> [m.start() for m in occurances] # starting points of matches
[0, 3, 7]
>>> [m.start() for m in occurances] # starting points of matches
[]      # iterator cannot be reused - must be re-created
>>> occurances=re.finditer("ab", "abcabcaabbc")
>>> [m.span() for m in occurances] # spanning intervals of matches
[(0, 2), (3, 5), (7, 9)]
```

(We'll discuss iterators later in the course.)

Of course, these built-in regular expression methods for string matching are not enough for our purposes, as we want to know what is going on "under the hood".

Representation of Characters: ASCII

Like everything else in the computer, characters are represented as binary numbers (yet, for convenience, we consider the decimal or hexadecimal representations).

The initial encoding scheme for representing characters is the so-called ASCII (American Standard Code for Information Interchange) encoding. It has 128 characters (represented by 7 bits). These include 94 printable characters (English letters, numerals, punctuation marks, math operators), space, and 33 invisible control characters (mostly obsolete).

Representation of Characters: ASCII

The initial encoding scheme for representing characters is the called ASCII (American Standard Code for Information Interchange). It has 128 characters (represented by 7 bits). These include 94 printable characters (English letters, numerals, punctuation marks, math operators), space, and 33 invisible control characters (mostly obsolete).

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

(table from Wikipedia. 8 rows/16 columns represented in hex, *e.g.* 'a' is 0x61, or 97 in decimal representation)

Representation of Characters: Unicode

With the increased popularity of computers and their usage in multiple languages (mainly those not employing Latin alphabet, even though á, â, ä, ã, å, à, ă, Å, etc. are also not expressible in ASCII), it became clear that ASCII encoding is not expressive enough and should be extended.

Demand for additional characters (*e.g.* various symbols that are not punctuation marks) and letters (*e.g.* Cyrillic, Hebrew, Arabic, Greek), possibly in the same piece of text, led to the 16 bit Unicode (and, along the way, earlier encodings).

Chinese characters (and additional ones, *e.g.* Byzantine musical symbols, if you really care) led to 20 bit Unicode.

Representation of Characters: Unicode (cont.)

Unicode is a variable length encoding: Different characters can be encoded using a different number of bits. For example, ASCII characters are encoded using one byte per character, and are compatible with the 7 bit ASCII encoding (a leading zero is added). Hebrew letters' encodings, for example, are in the range 1488 (hex 05D0, א) to 1514 (hex 05EA, tav, unfortunately unknown to LaTeX), reflecting the 22+5=27 letters in the Hebrew alphabet.

Python employs Unicode. The built in function `ord` returns the Unicode encoding of a (single) character (in decimal). The `chr` of an encoding returns the corresponding character.

```
>>> ord(" ")     # space
32
>>> ord("a")
97
>>> hex(ord("a"))
0x61
>>> chr(97)
a
```

Strings and Sequences: Two Less Standard Contexts

Plain text editors are one of the most popular applications (be it `troff`, `Notepad`, `Emacs`, `Word`, LaTeX, etc.).

Other than that, string operations are important in many other contexts. For example:

- Biological sequence operations. Here the text could be a chromosome, a whole genome, or even a collection of genomes. Given that the length of, say, the human genome, is approximately 3 billion letters (`A`, `C`, `G`, `T`), efficiency may be crucial (whereas if processing single `proteins` or `genes`, just hundreds or thousands letters long, we could be slightly more tolerant).
- Musical information retrieval, where, for example, you may want to whistle or hum into your smartphone, and have it retrieve the most similar piece of music out of some large collection.

Basic Flavors of String Matching/Sequence Comparison

1. Exact string matching.
   In our example, the pattern was "gyre and gimble", and we look for all exact occurrences of it in the text.

2. Inexact string matching.
   In our example, suppose the pattern was "gore and grumble", and we look for occurrences in the text that are closest to the pattern (we should define a measure of proximity in this case).

Exact String Matching

We deal with strings over a fixed alphabet, Σ. Typically Σ consists of the letters of a natural language such as English, Hebrew, Arabic, or Russian, plus space, numbers, and punctuation marks.
Other popular choices are the binary alphabet, $\{0, 1\}$, the 20 amino acids, $\{\mathtt{A}, \mathtt{R}, \mathtt{N}, \mathtt{D}, \mathtt{C}, \mathtt{Q}, \mathtt{E}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{L}, \mathtt{K}, \mathtt{M}, \mathtt{F}, \mathtt{P}, \mathtt{S}, \mathtt{T}, \mathtt{W}, \mathtt{Y}, \mathtt{V}\}$, or the 4 DNA nucleotides, $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

The input consists of two strings. A text, T, whose length we denote by n, and a shorter pattern, P, whose length we denote by m. In the jabberwocky example above, $n = 934$ and $m = 15$ (spaces are counted).

Exact String Matching (2)

Input: Two strings over a fixed alphabet, Σ.
A text of length n, $T = T[0..n-1]$, and
a shorter pattern of length m, $P = P[0..m-1]$.

Goal: Find all indices, or shifts, s, in the text ($0 \leq s \leq n-m$)
such that $T[s..s+m-1] = P[0..m-1]$.

That is, all shifts where the pattern matches the text exactly.

This is a fundamental problem in stringology. We will examine three approaches to solving it, bearing the associated complexity in mind.

1. The naïve, exhaustive, approach.
2. A randomized (coin flipping) algorithm, due to Karp and Rabin.
3. An approach based on deterministic finite automata.

The Naïve Algorithm

Goal: Find all indices, or shifts, s, in the text ($0 \leq s \leq n - m$) such that $T[s..s + m - 1] = P[0..m - 1]$.

Method: For each shift, s, $0 \leq s \leq n - m$, compare the two strings $T[s..s + m - 1]$ and $P[0..m - 1]$, character by character. If they are equal, report s (and continue).

Cost: For each shift, s, we compare two blocks of m characters. This takes m operations. The number of shifts is $n - m + 1$.
So the overall cost is $m \cdot (n - m + 1) = \theta(m \cdot n)$.

The Naïve Algorithm – Python Code

```
def naive_matches(pattern,text):
    """ naive search for exact matches of pattern inside text """
    assert(len(pattern) <= len(text))    # lengths are compatible
    n=len(text)
    m=len(pattern)
    matches=[]    # will hold all locations of match start indices
    for i in range(n-m+1):  # do not forget the +1
        if pattern==text[i:i+m]:
            matches=matches+[i]
    return(matches)
```

```
>>> naive_matches("bar","bennyXbirburbirbarYraniZbarbarossa")
[15, 24, 27]
>>>
```

Adequacy: Whether the performance of this is good enough (or not) depends on the application. For small values of m, n, like in the example above, this is surely acceptable, but not for big ones.

The usual question: Can we do better?

Step 1: From Strings to Integers

Let $P[0..m-1]$ be an m character long string (the pattern).
Let p_i be the unicode representation of the character $P[i]$.
Recall that 16 bits suffice to represent all reasonable characters.

So for all i ($0 \leq i \leq m-1$), $0 \leq p_i < B$, where $B = 2^{16}$.

We transform $P[0..m-1]$ into a base B number, denoted by N_P:
$N_P = p_0 B^{m-1} + p_1 B^{m-2} + \ldots + p_{m-3} B^2 + p_{m-2} B + p_{m-1}$.

This sum can be computed efficiently, using Hörner's rule,

$$N_P = ((((\ldots(((p_0 \cdot B) + p_1) \cdot B) + \ldots + p_{m-3}) \cdot B) + p_{m-2}) \cdot B) + p_{m-1}$$

Arithmetization

$$N_P = ((((\ldots(((p_0 \cdot B) + p_1) \cdot B) + \ldots + p_{m-3}) \cdot B) + p_{m-2}) \cdot B) + p_{m-1} \ .$$

This requires m additions and m multiplications.
The outcome is of size $\approx B^m$.

The mapping from the pattern, $P[0..m-1]$, into N_P, is one to one.
So N_P uniquely determines the pattern.

We can think of the transformation from the pattern P (a string) to N_P (a number) as an arithmetization of the string.

From Strings to Integers: Step 2

Let $T[0..n-1]$ be an n character long string (the text).

Let t_s be the unicode representation of the character $T[s]$.

Like before, for every shift, $s,\ 0 \leq s \leq n-m$, we transform the m character long "shift" $T[s..s+m-1]$ into base B number, denoted by T_s:

$$T_s = t_s B^{m-1} + t_{s+1} B^{m-2} + \ldots + t_{s+m-3} B^2 + t_{s+m-2} B + t_{s+m-1}.$$

Once we got N_P and all of $T_0, T_1, \ldots, T_{n-m}$, the string matching problem reduces to the question of finding all indices i for which $N_P = T_i$.

From Strings to Integers: Python Code

The following function computes the arithmetization of a single string:

```
def arithmetize(string,basis=2**16):
    """ convert substring to number using basis powers
        employs Horner rule """
    partial_sum=0
    for ch in string:
        partial_sum =(partial_sum*basis+ord(ch)) # Horner
    return partial_sum

>>> arithmetize("b")
98
>>> arithmetize("be")
6422629
>>> arithmetize("ben")
420913414254
>>> ((ord("b")*2**16)+ord("e"))*2**16+ord("n") # sanity check
420913414254
```

From Strings to Integers 2: Python Code

The following function computes the arithmetization of all length `m` contiguous substrings in the `text`:

```
def arithmetize_text(text,m,basis=2**16):
    """  computes arithmization of all m long subwords
         of text, using basis powers """
    t=[]
    # store list of numbers representing consecutive chunks of text
    for s in range(0,len(text)-m+1):
        t=t+[arithmetize(text[s:s+m],basis)]
        # append the next number to existing t
    return t
```