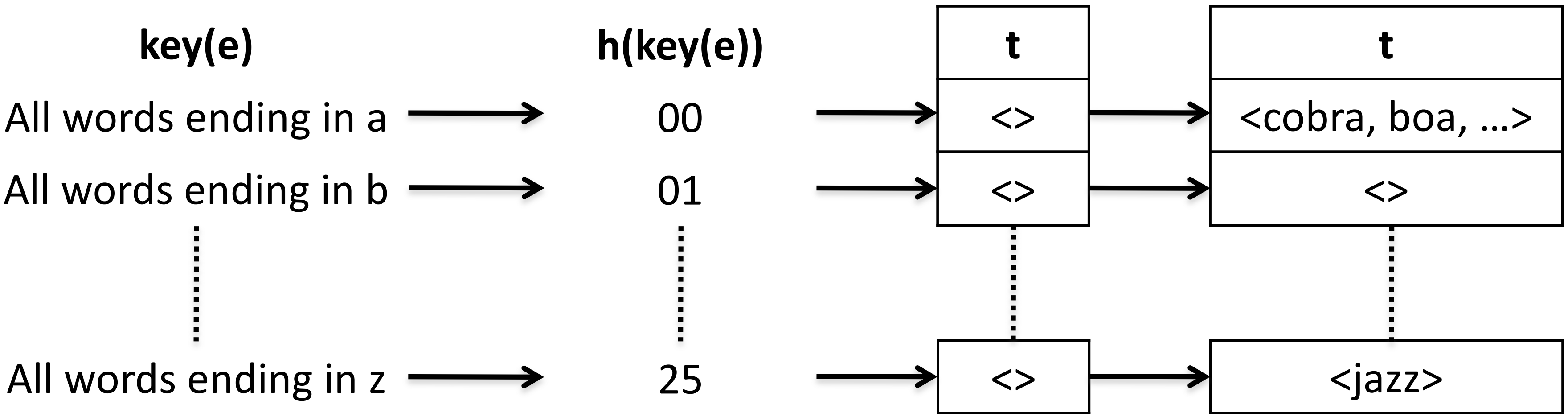
Algorithm and Data Structure Analysis (ADSA)

Hashing (2)

Previous Lecture

- Introduction to hashing
  - Use hash function h(key(e)) to obtain index of element *e* in hash table *t*
- Hashing with chaining



Previous Lecture: Symbols

- S = associative array
- t = hash table
- N = number of potential keys = $|S|$
- m = number of possible hash function values

  = $|t|$
- n = number of elements

Example Hashing

- Consider the following hash function

  $h(x) = 2x \bmod 29$ which maps a non-negative integer to a value in $\{0, \ldots, 28\}$.
- Compute $h(x)$ for $x = 2, 5, 10, 22, 34$.
- Are there two elements hashed to the same value?
- We have $h(2)=4$, $h(5)=10$, $h(22)=15$, $h(34)=10$.
- $x=5$ and $x=34$ both hashed to 10

Previous Lecture: Average Case Analysis for Hashing with Chaining

Theorem: If n elements are stored in a hash table t with m entries using hashing with chaining and a random hash function is used, the expected execution time of remove or find is $O(1+ n/m)$.

Note: a random hash function maps e to all m table entries with the same probability.

Universal Hashing

Theorem 4.1 is unsatisfactory, as the class of "all hash functions" is too big to be useful: $|H|=m^N$, thus it requires $N \log m$ bits to specify a function in H.

This drawback can be overcome with much smaller classes of hash functions, and their members can be specified in constant space.

Universal Hashing

Definition 4.2 Let *c* be a positive constant. A family *H* of functions from *Key* to *0..m-1* is called c-universal if any two distinct keys collide with a probability of at most *c/m*:

$$\forall x, y \in Key, x \neq y :$$

$$\left| \left\{ h \in H : h(x) = h(y) \right\} \right| \leq \frac{c}{m} |H|$$

Or, for a random h∈H: $prob(h(x) = h(y)) \leq \dfrac{c}{m}$

Universal Hashing

Theorem 4.3 If n elements are stored in a hash table with m entries using hashing with chaining and a random hash function from a c-universal family is used, the expected execution time of remove or find is $O(1+cn/m)$.

Proof

Follows the proof of Theorem 4.1.

Expected Execution Time remove/find

Proof:

Execution time for remove and find is constant time plus the time scanning the list *t[h(k)]*.

Let the random variable X be the length of the list *t[h(k)],* and let *E[X]* be the expected length of the list.

Thus the *expected* execution time = O(1 + E[X]).

Expected Execution Time remove/find

Proof (continued):

Let S be the set of n elements contained in t.

For each $e \in S$, let X_e be an indicator variable which indicates whether e hashes to the same value as k.

ie: **if** $h(key(e)) = h(k)$ **then** $X_e = 1$ **else** $X_e = 0$.

$$X = \sum_{e \in S} X_e$$

(ie how many e's are in table entry h(key(e)))

Expected Execution Time remove/find

Proof (continued):

$$E[X] = E\left[\sum_{e \in S} X_e\right]$$

$$= \sum_{e \in S} E[X_e]$$

$$= \sum_{e \in S} prob(X_e = 1)$$

Expected Execution Time remove/find

Proof (continued):

$$E[X] = \sum_{e \in S} prob(X_e = 1)$$

(From last slide)

$$= \sum_{e \in S} c / m$$

(As function h is chosen uniformly from a c-universal class: $prob(X_e = 1) \leq c / m$ *)*

$$= c \cdot n / m$$

(Because n elements in S)

Expected Execution Time remove/find

Proof (continued):

Expected execution time = $O(1 + E[X])$,

$E[X] = c \cdot n/m$

Thus the expected execution time for remove and find under hashing with chaining is $O(1 + c \cdot n/m)$. $\square$

C-universal families

For practical purposes: find c-universal families that are easy to construct and evaluate.

We will describe a simple and quite practical 1-universal family in detail…

Assumptions

- keys are bit strings of fixed length
- table size *m* is a prime number

1-universal family

Why prime? Arithmetic modulo prime is nice: the set $\mathbb{Z}_m=\{0,\ldots,m\text{-}1\}$ of numbers modulo m forms a field.

A field is a set with special elements 0 and 1, and with addition and multiplication operators, satisfying certain axioms (associative & commutative & distributive properties, existence of neutral elements, …)

1-universal family

Let $w = \lfloor \log m \rfloor$.

We subdivide the keys into pieces of *w* bits each (say in total *k* pieces). We interpret each piece as an integer in the range $0..2^w-1$ and keys as *k*-tuples of such integers.

For a key $\mathbf{x}$, we write $\mathbf{x}=(x_1,\ldots x_k)$ to denote its partition into pieces. Each x_i lies in $0..2^w-1$.

We can now define our class of hash functions.

1-universal family

For each $\boldsymbol{a}=(a_1,\ldots a_k) \in \{0..m-1\}^k$, we define a function h_a from *Key* to *0..m-1* as follows.

Let $\boldsymbol{x}=(x_1,\ldots,x_k)$ be a key and let $\boldsymbol{a}\cdot\boldsymbol{x} = \sum_{i=1}^{k} a_i x_i$ denote the scalar product of $\boldsymbol{a}$ and $\boldsymbol{x}$.

Then $h_a(\boldsymbol{x}) = \boldsymbol{a}\cdot\boldsymbol{x} \bmod m$.

1-universal family

Example

Let $m=17$, $k=4$. Then $w=4$ and we view keys as 4-tuples in the range $0..15$, for example $\mathbf{x}=(11,7,4,3)$.

A hash function is specified by a 4-tuple of integers in the range $0..16$, for example $\mathbf{a}=(2,4,7,16)$.

Then $h_a(\mathbf{x}) = (2 \cdot 11 + 4 \cdot 7 + 7 \cdot 4 + 16 \cdot 3) \bmod 17 = 7$.

1-universal family

Theorem 4.4

$$H = \{ h_{a}: \mathbf{a} \in \{0..m-1\}^{k} \}$$

is a 1-universal family of hash functions, if *m* is prime.

In other words, the scalar product between a tuple representation of a key and a random vector modulo m defines a good hash function.

Proof

Proof of Theorem 4.4:

- Consider two keys

$$x = (x_1, \dots, x_k) \text{ and } y = (y_1, \dots, y_k)$$

- Consider the number of choices of a such that

$$h_a(x) = h_a(y)$$

- Fix index j such that $x_j \neq y_j$
- Implies $(x_j - y_j) \neq 0 (mod\ m)$

- Equation $a_j(x_j - y_j) = b(mod\ m), b \in Z_m$ has unique solution

$$a_j = (x_j - y_j)^{-1} b(mod\ m)$$

Claim: For each choice of the $a_i, i \neq j$, there is exactly one choice of a_j such that

$$h_a(x) = h_a(y)$$

$$
\begin{aligned}
h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y}) \Leftrightarrow \quad & \sum_{1\le i\le k} a_i x_i \equiv \sum_{1\le i\le k} a_i y_i && \pmod{m} \\
\Leftrightarrow \ & a_j(x_j - y_j) \equiv \sum_{i\neq j} a_i(y_i - x_i) && \pmod{m} \\
\Leftrightarrow \quad & \quad a_j \equiv (y_j - x_j)^{-1} \sum_{i\neq j} a_i(x_i - y_i) && \pmod{m} \quad .
\end{aligned}
$$

m^{k-1} ways to choose a_i with $i \neq j$ and for each such choice there is a unique choice of a_j.

In total m^k choice which implies

$$
\text{prob}(h_{\mathbf{a}}(x) = h_{\mathbf{a}}(\mathbf{y})) = \frac{m^{k-1}}{m^k} = \frac{1}{m}
$$

$\square$

Prime Table Sizes

Is it a serious restriction?

At first glance: yes.

- The user has to provide appropriate primes.
- While growing/shrinking: how to obtain new prime numbers for the new value of m?

Easy solution: consult a table of primes.

Analytical solution: not much harder.

Prime Table Sizes

From number theory:

- there is an infinite number of primes
- for any integer k there is a prime in the interval $[k^3,(k+1)^3]$

So, if we are aiming for a table size of about m, we determine k such that $k^3 \leq m \leq (k+1)^3$ and then search for a prime in this interval.

Prime Table Sizes

How does this search work?

Any nonprime in the interval must have a divisor which is at most $\sqrt{(k+1)^3} = (k+1)^{3/2}$.

We therefore iterate over the numbers from 2 to $(k+1)^{3/2}$, and for each such j remove its multiples in $[k^3,(k+1)^3]$.

For each fixed j, this takes time $((k+1)^3-k^3)/j=O(k^2/j)$.

Prime Table Sizes

The total time required is

$$\sum_{j \le (k+1)^{3/2}} O\left(\frac{k^2}{j}\right) = k^2 \sum_{j \le (k+1)^{3/2}} O\left(\frac{1}{j}\right)$$

$$= O\left(k^2 \ln\left((k+1)^{3/2}\right)\right) = O\left(k^2 \ln k\right) = o(m)$$

and hence is negligible compared with the cost of initializing a table of size *m*.

Alternative Approach to Hashing

Hashing with chaining is a closed hashing approach.

- Closed hashing: handles collision by storing all elements with the same hashed key in one table entry.
- Open hashing: handles collision by storing subsequent elements with the same hashed key in different table entries.

Hashing with Linear Probing

- Hashing with Linear Probing is an open hashing approach.
- All unused entries in t are set to $\perp$.
- When inserting, on a collision insert the element to the next free entry.

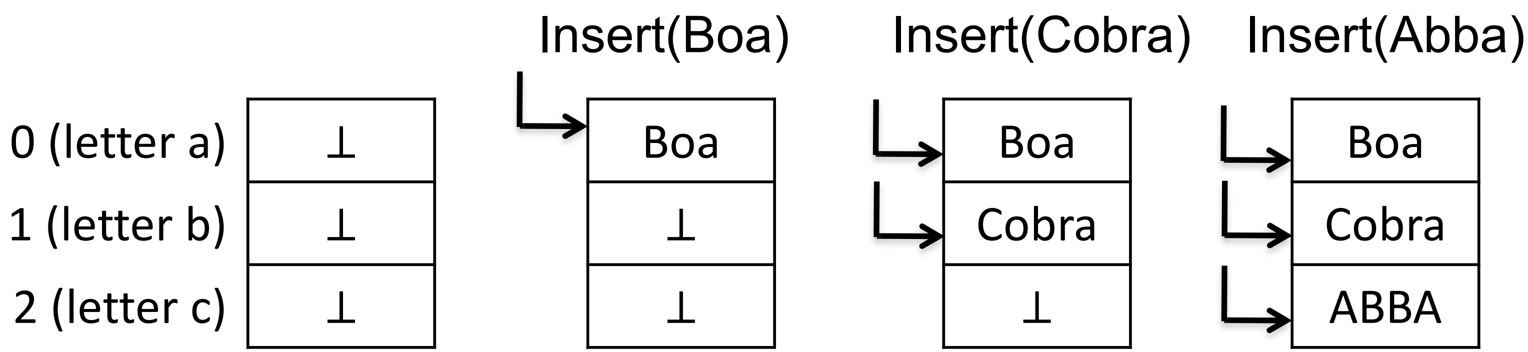
- What if the last entry is used?

Hashing with Linear Probing

- Trivial fix: allow more entries
- Make table t size $m + m'$ instead of m. Choose $m' < m$.

Insert(e)

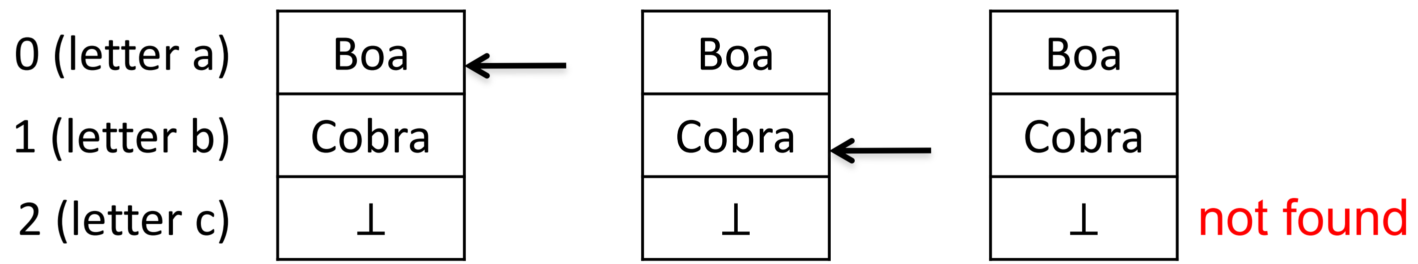
- insert(*e*: Element)
  1. Get index $i = h(key(e))$
  2. If $t[i] == \perp$, store *e* at $t[i]$
  3. If $t[i]$ is not empty, increase *i* by 1 and go to step 2.



Find(k)

- find(k: Key)
  1. Get index $i = h(k)$
  2. If $t[i] == \perp$, return not found
  3. If element e at $t[i]$ has $key(e) == k$, return found. Else increase i by 1 and go to step 2.

eg Find(ABBA)



Remove(k)

- Can't remove the element with *key(e) == k* and replace it with ⊥.
  - If we replace element *e1* at *t[i]* with ⊥, how do we find an element *e2* with the same *h(k)*?

- Instead, first remove the element with *key(e) == k* and then fix the invariant.

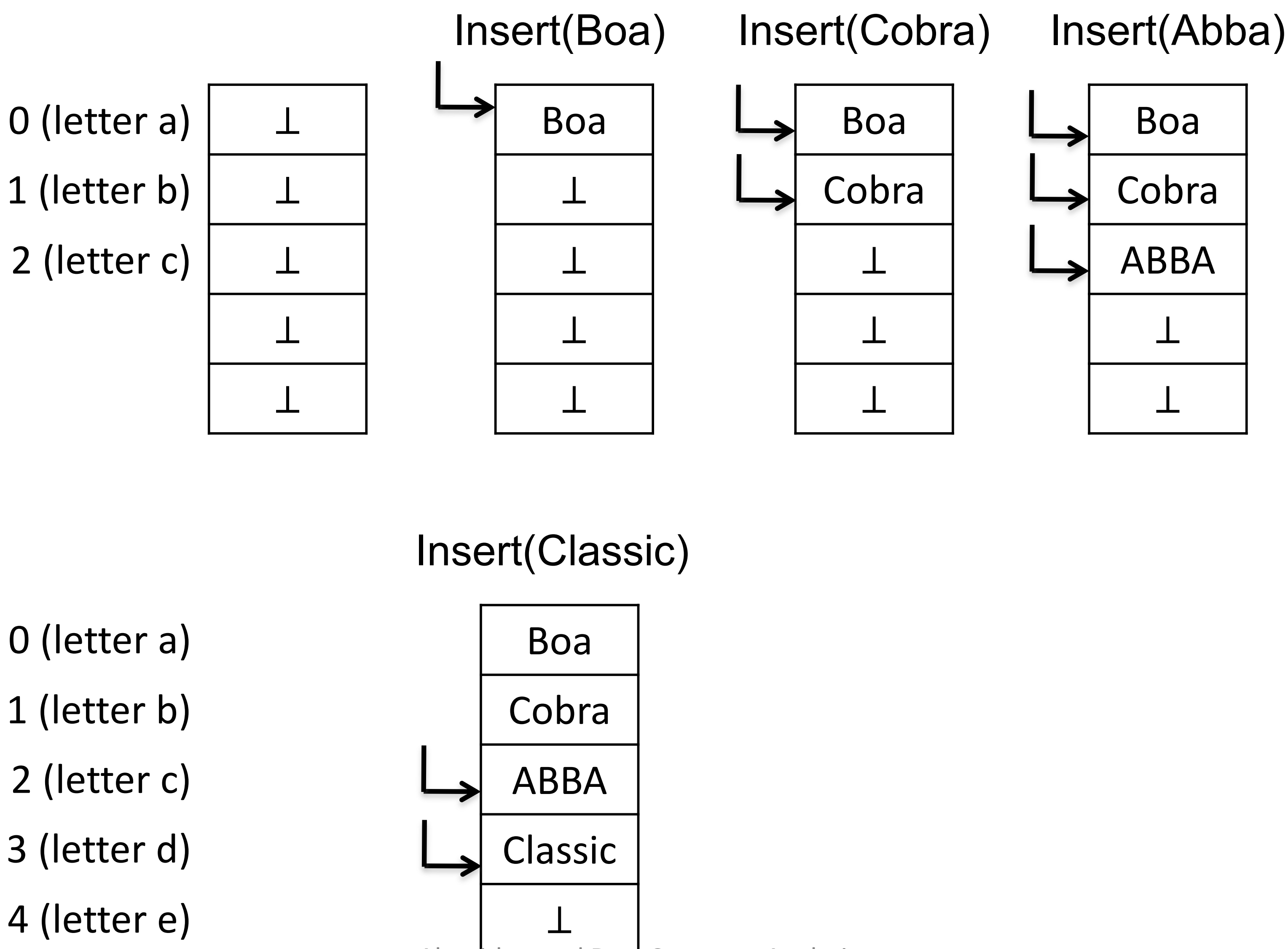
Remove(k)

- remove(*k*: Key)
  1. Get index $i = h(k)$
  2. If $t[i] == \perp$, return
  3. If element e at $t[i]$ has $key(e) \mathrel{!}= k$, increase i by 1 and go to step 2.
  4. Set $t[i] = \perp$
  5. Set index $j = i+1$
  6. If $t[j] == \perp$, return
  7. If $h(t[j]) > i$, increase j by 1
  8. Else set $t[i] = t[j]$ and $t[j] = \perp$, Set $i = j$ and go to step 5.

search (k)

repair

Example Inserts



Example: Remove(Cobra)

		Step 1,3	Step 4	Step 5
0 (letter a)	Boa	Boa	Boa	Boa
1 (letter b)	Cobra	Cobra	⊥	⊥
2 (letter c)	ABBA	ABBA	ABBA	ABBA
3 (letter d)	Classic	Classic	Classic	Classic
4 (letter e)	⊥	⊥	⊥	⊥

	Step 8,9,5	Step 8,9,5	Step 6
0 (letter a)	Boa	Boa	Boa
1 (letter b)	ABBA	ABBA	ABBA
2 (letter c)	⊥	Classic	Classic
3 (letter d)	Classic	⊥	⊥
4 (letter e)	⊥	⊥	⊥

Chaining vs. Linear Probing

Argumentation depends on the intended use and many technical parameters:

Chaining	Linear probing
+ referential integrity	+ use of contiguous memory
- waste of space	- gets slower as table fills up

A fair comparison must be based on space consumption, not only on the runtime.

Experimental results: so small differences that implementation details, used compiler, OS, … matter.