Algorithm and Data Structure Analysis (ADSA)

Hashing (1)

Motivation for Data Structure

- Worst case analysis of data structures:

Name	**Insert(x)**	**Remove(x)**	**Find(x)**
Linked Lists	O(1)	O(1)	Θ(n)
AVL Trees	O(log n)	O(log n)	O(log n)

- Can we have constant time insertion and removal, yet have a better find?

Associative Arrays

Idea: consider a different use of arrays.

- Don't change array size on insert or remove.
- On insert, simply insert the element at the index.
- On remove, simply clear the element at the index.
- Assume we know the index of x.
  - insert(x) is O(1)
  - remove(x) is O(1)
  - find(x) is O(1)

Associative Arrays

- Associative array *S* stores elements
- Each element *e* in *S* has a unique key: *key(e)*. Clearly, each key has a unique element.
- Need an index in *S* for each possible key.

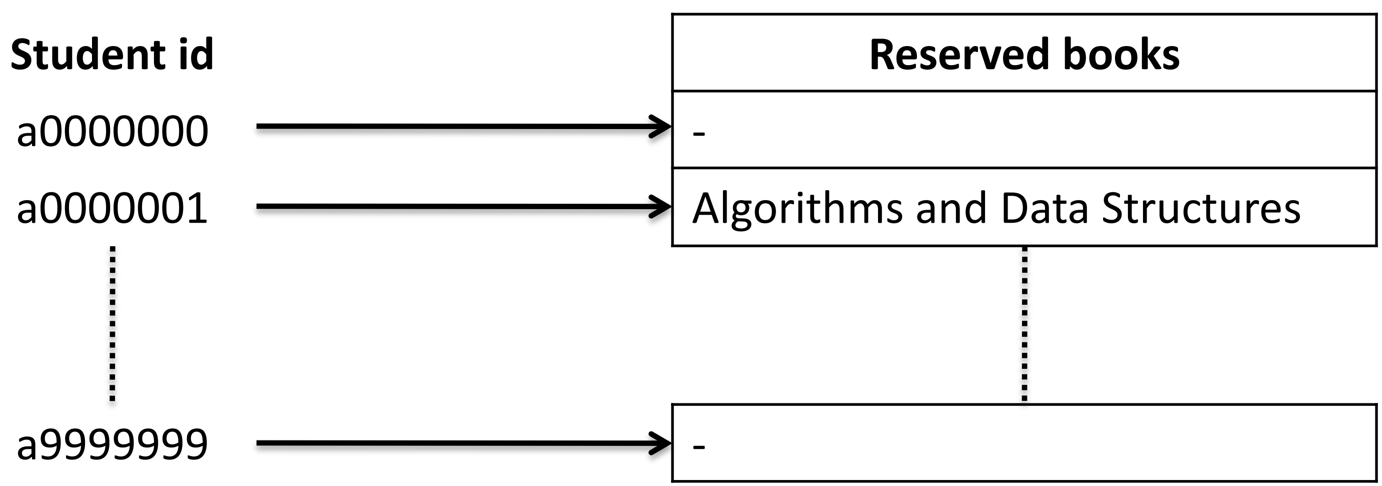
S.insert(e: Element): $S := S \cup \{e\}$

S.remove(k: Key(e)): $S := S \setminus \{e\}$

S.find(k: Key(e)): if *e* in *S*, return *e*. Else return null.

Associative Arrays

- Problem: number of possible keys is MASSIVE.
- Library example: how many students borrow books? How many student ids are there?



Hash Tables

- Idea: use hash function h to map potential N keys to m values, where $m < N$.
- Let t be a hash table of size m.
- Store element e in index $h(key(e))$ of t.

Hash Tables

- Example hash function: *key(e)* are student ids, *h(key(e))* are last two digits of student ids.
- N is 10^7, m is 10^2.

	h(key(e))	**Table t**
All ids ending in 00	00	Introduction to Software Eng.
All ids ending in 01	01	Algorithms and Data Structures
⋮	⋮	⋮
All ids ending in 99	99	Some Dinosaur Book

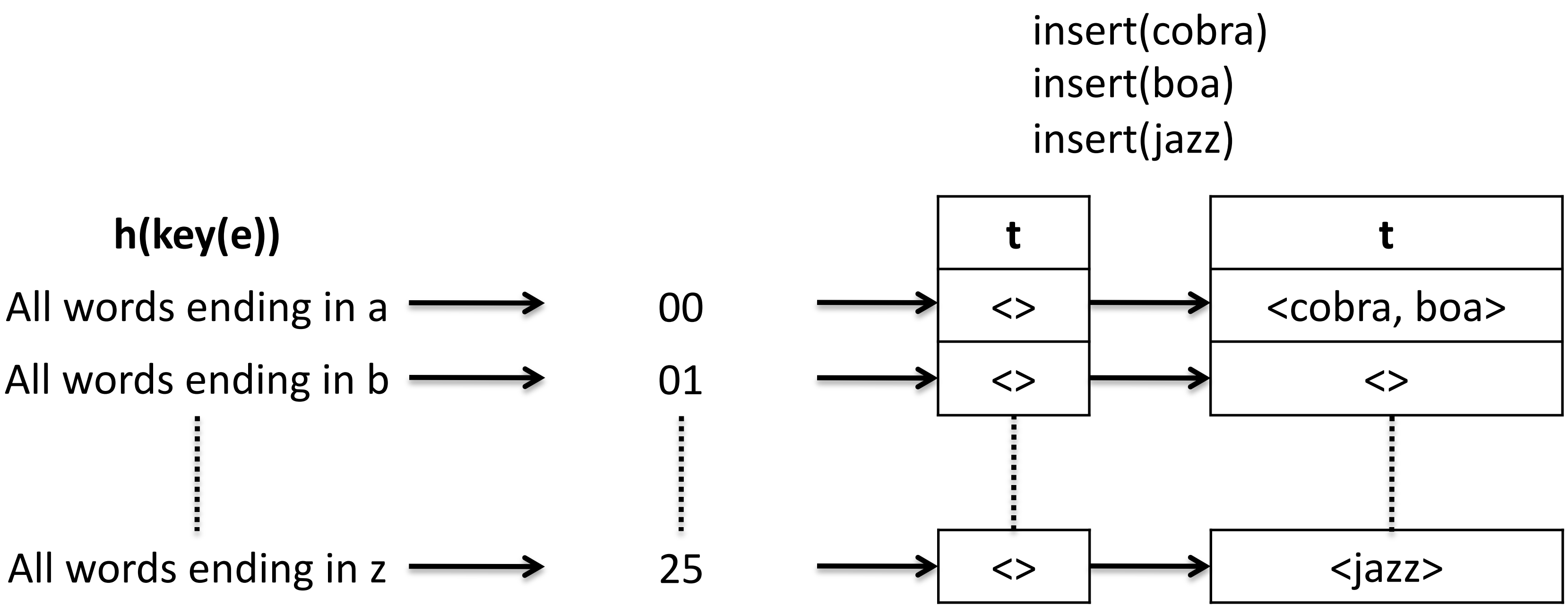
Hash Tables: Collisions

- Smaller table to store elements means some elements may get stored in the same index.
- Previous example, a0000000 and a1995400.

- How do we handle collisions?
  - Think linked lists…

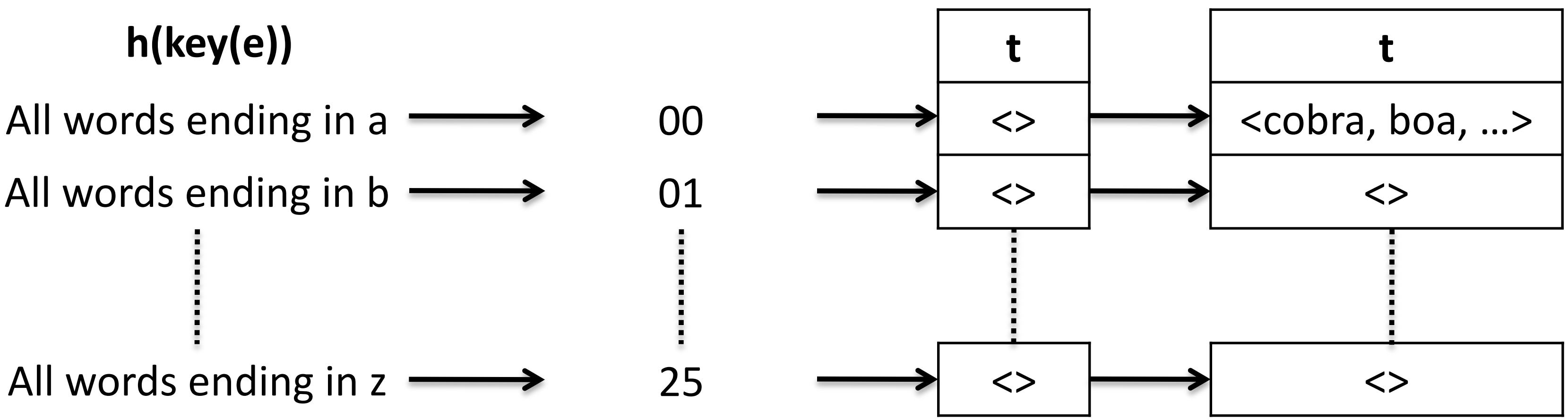
Hashing with Chaining

- Solution: let t be a table of linked lists.
- Example: Storing words.



Hashing with Chaining

- Worst case performance: hash function of elements returns the same value.
- In example, insert cobra, boa, ABBA, zebra



Chaining Limitations

- N = number of potential keys
- m = number of possible hash function values
- n = number of elements
- Thus hash functions will have sets of N/m keys mapped to the same index of t.
- As (usually) $n < N/m$, it is possible to have all n elements in one table entry.

Insert(e)

- insert(*e*: Element)
  - Get index *h(key(e))*
  - Add *e* to the end of the list at *t[h(key(e))]*

- What is the worst case complexity?

Insert(e)

- insert(*e*: Element)
  - Get index *h(key(e))*
  - Add *e* to the end of the list at *t[h(key(e))]*

- Hash function is O(1)
- Worst case insert of linked list is O(1)
- Thus insert(*e*: Element) is O(1).

Find(k)

- find(*k*: Key)
  - Get index *h(k)*
  - Search through list at *t[h(k)]*.
  - If element *e* with unique key *k* is in list, return *e*. Else return null.

- What is the worst case complexity?

Find(k)

- find(*k*: Key)
  - Get index *h(k)*
  - Search through list at *t[h(k)]*.
  - If element *e* with unique key *k* is in list, return *e*. Else return null.

- Hash function is O(1)
- Worst case find of linked list is $\Theta(n)$
- Thus find(*k*: Key) is $\Theta(n)$.

Remove(k)

- remove(*k*: Key)
  - Get index *h(k)*
  - Search through list at *t[h(k)]*.
  - If element *e* with unique key *k* is in list, remove *e*.

- What is the worst case complexity?

Remove(k)

- remove(*k*: Key)
  - Get index *h(k)*
  - Search through list at *t[h(k)]*.
  - If element *e* with unique key *k* is in list, remove *e*.

- Hash function is O(1)
- Worst case find of linked list is Θ(*n*)
- Worst case remove of linked list is O(1)
- Thus remove(*k*: Key) is Θ(*n*).

Average Case Analysis

Theorem 4.1: If n elements are stored in a hash table t with m entries 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.

Average Case Analysis

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]).

Average Case Analysis

Proof (continued):

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

For each e, 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)))

Average Case Analysis

Proof (continued):

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

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

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

Average Case Analysis

Proof (continued):

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

(From last slide)

$$= \sum_{e \in S} 1/m$$

(As function maps e to all m with equal probability)

$$= n/m$$

(Because n elements in S)

Average Case Analysis

Proof (continued):

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

$E[X] = n/m$

Thus the expected execution time for remove and find under hashing with chaining is $O(1 + n/m)$, and constant if $m = \Theta(n)$

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:

$$prob(h(x) = h(y)) \leq \frac{c}{m}$$

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

$O(1 + c \cdot n/m)$. □