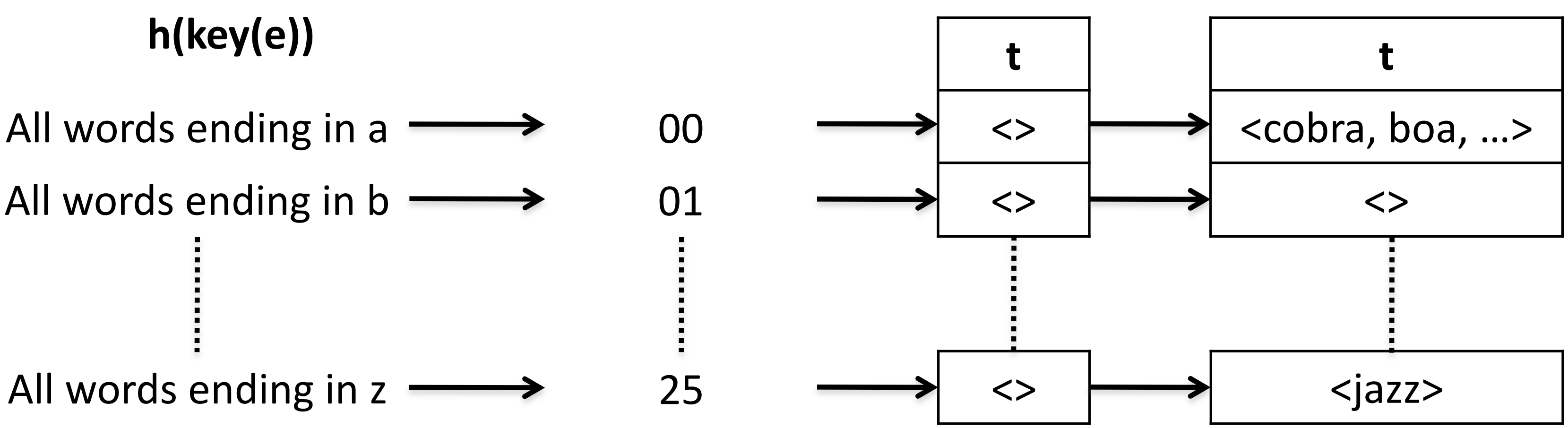
Algorithm and Data Structure Analysis (ADSA)

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

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.

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 $m = \Theta(n)$

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.

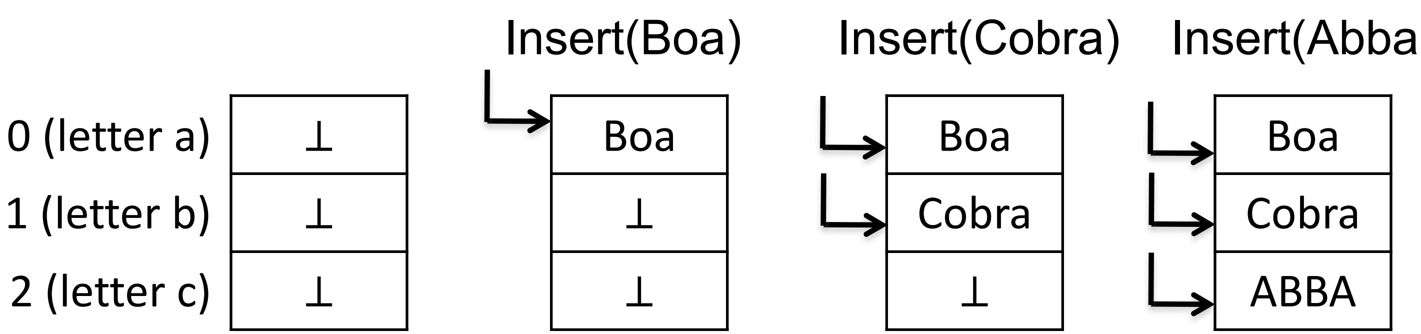
- What if the last entry is used?

Hashing with Linear Probing

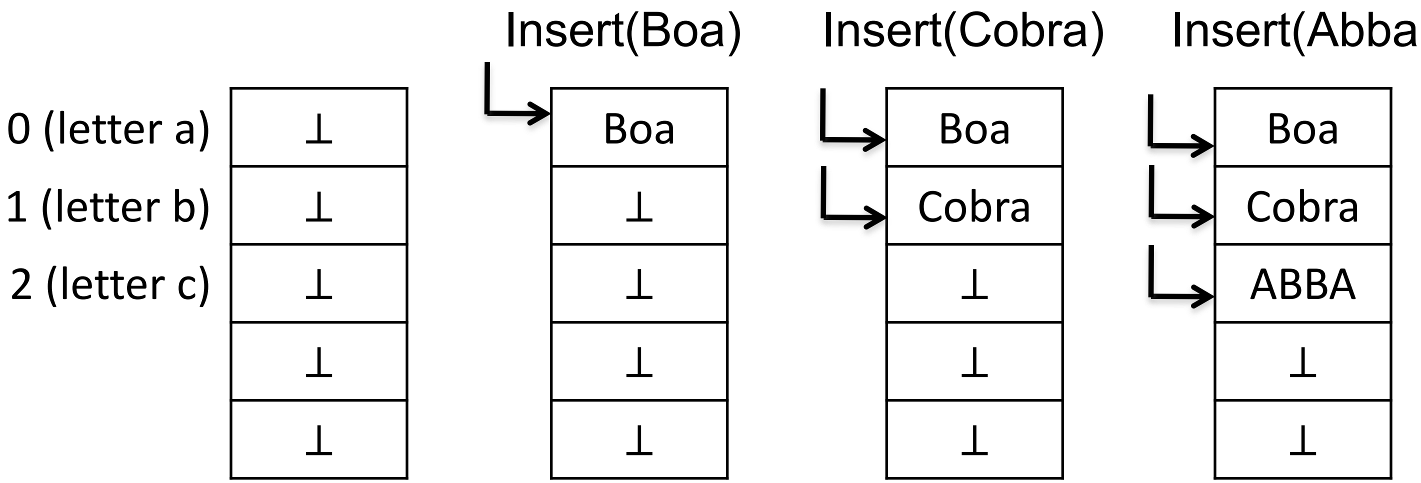
- Trivial fix: allow more entries
- Make table *t* size $m + m'$ instead of m. Choose $m' < m$.

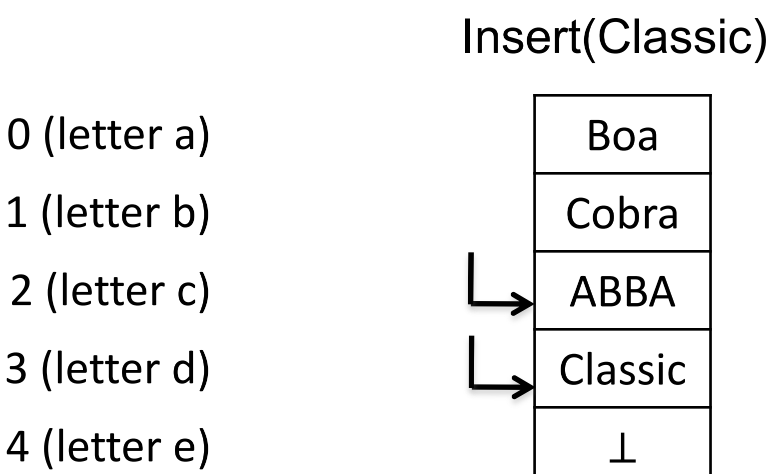
Insert(e)

- 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.



Example Inserts

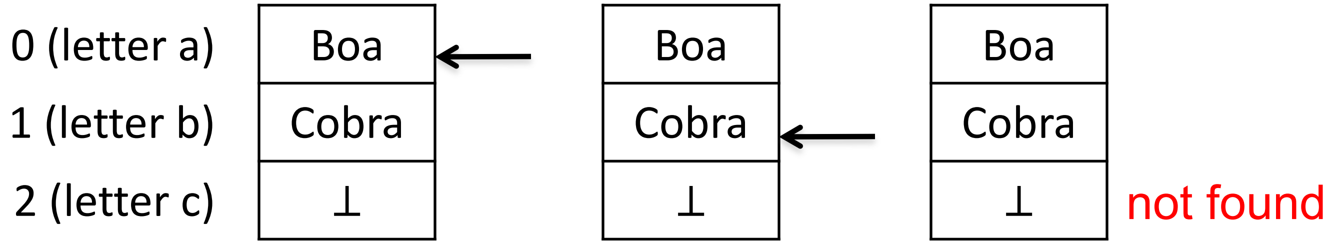




Find(k)

- 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.

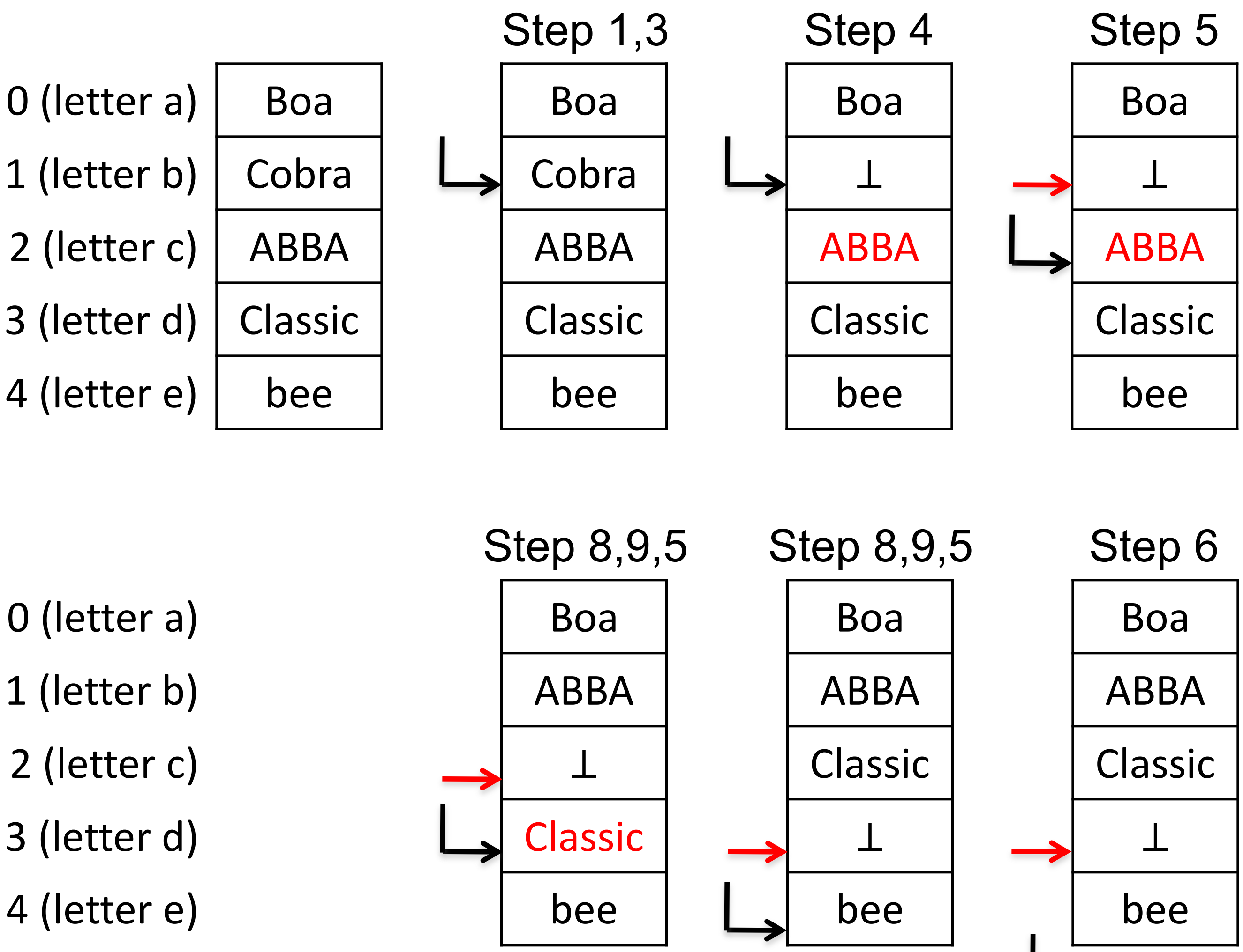
Remove(k)

- remove(k: Key)
  1. Get index $i = h(k)$
  2. If $t[i] == \perp$, return
  3. If element e at $t[i]$ has $key(e)$!= 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 and go to step 6
  8. Else set $t[i] = t[j]$ and $t[j] = \perp$
  9. set $i = j$ and go to step 5.

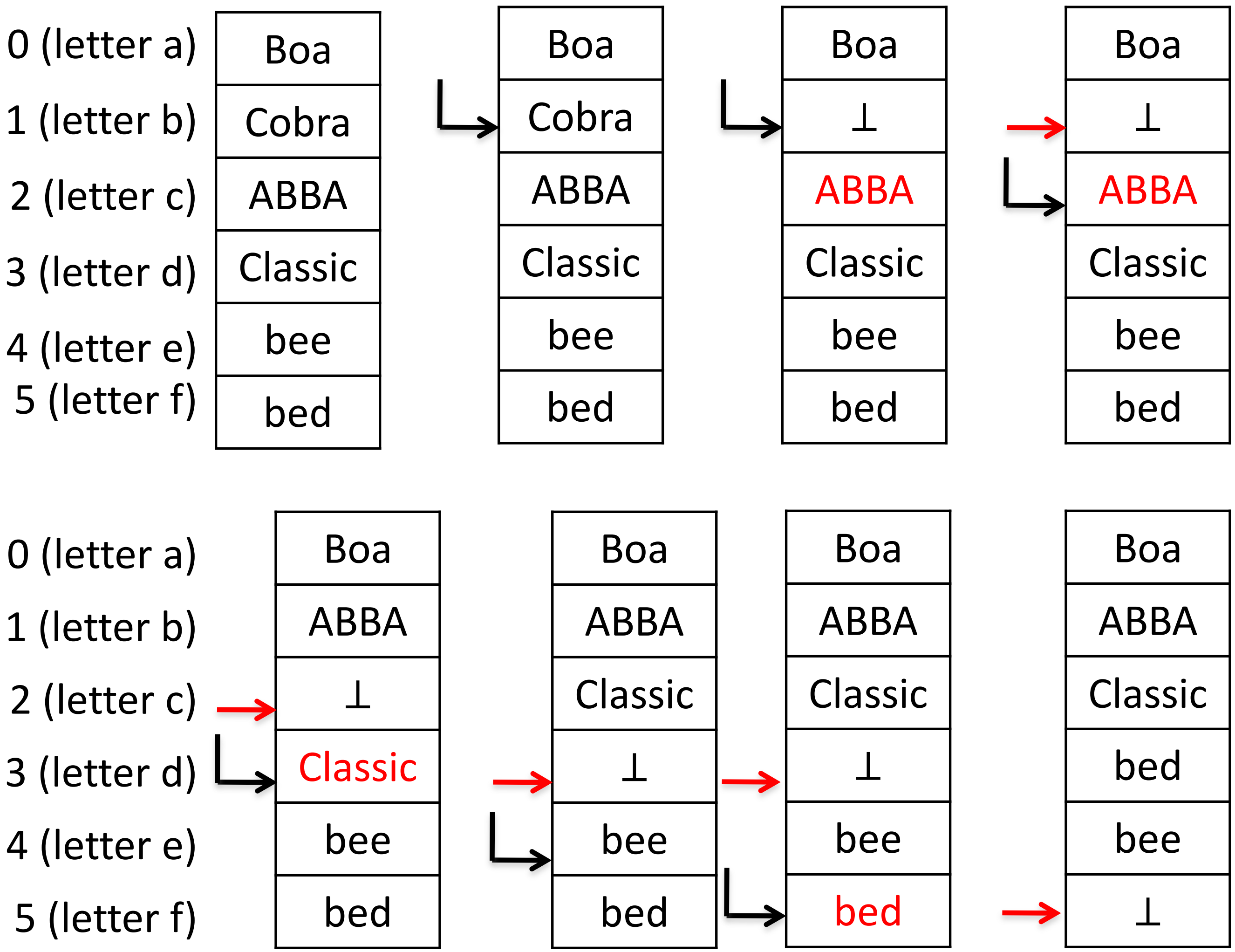
search (k)

repair

Example: Remove(Cobra)



Example: Remove(Cobra)



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.