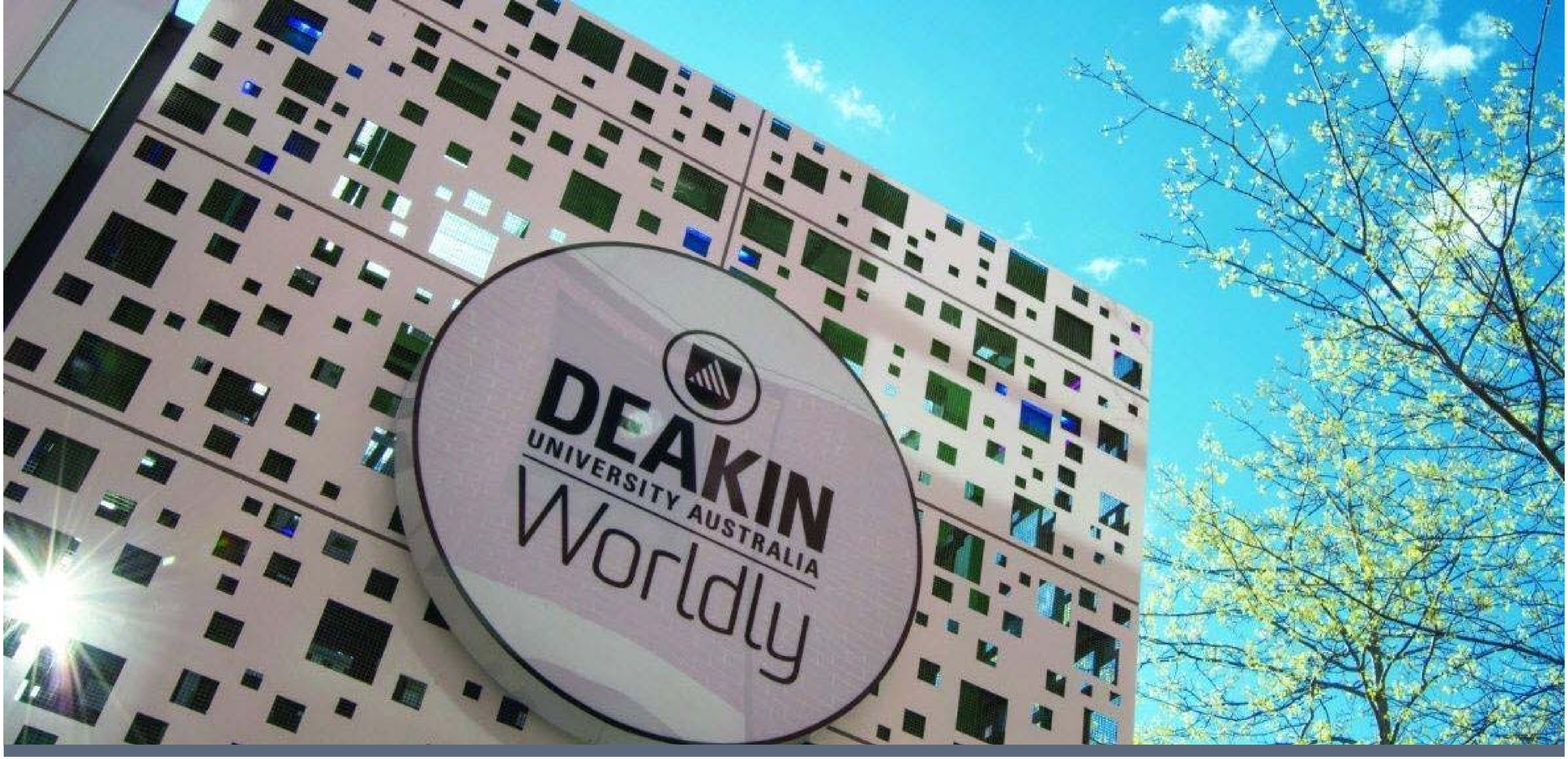


Lecture 10. Hashing and Hash Tables

SIT221 Data Structures and Algorithms

Hashing: Motivation

Worst case analysis of data structures:

Name	Insert(x)	Remove(x)	Find(x)
Linked Lists	$O(1)$	$O(1)$	$O(n)$
AVL Trees	$O(\log n)$	$O(\log n)$	$O(\log n)$

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

Hashing: Associative Arrays

Idea: Consider a different use of arrays.

- Do not change array size on Insert or Remove.
- 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 $O(1)$

Hashing: Associative Arrays

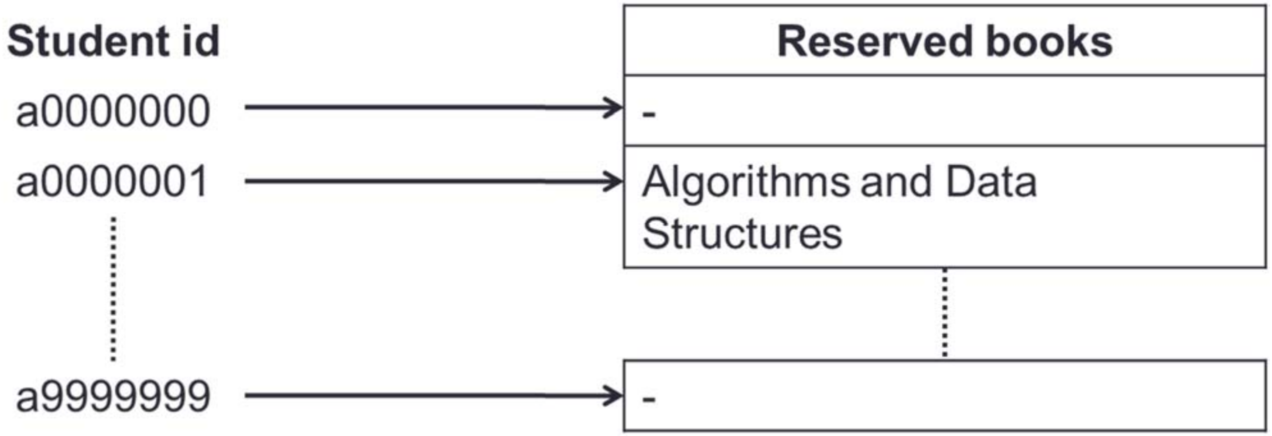
- Associative array S stores elements.
- Each element e in S has a unique key: $\mathrm{key}(e)$. Clearly, each key has a unique element.
- It needs an index in S for each possible key.

Operations:

- S.Insert(e: Element): $S \coloneqq S \cup \{e\}$
- S.Remove(k: Key): $S \coloneqq S \backslash \{e\colon k = \mathrm{key}(e)\}$
- S.Find(k: Key): if e in S, return e. Else return null.

Hashing: Associative Arrays

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



Hashing: Associative Arrays

- Let N be the number of potential keys in S
- Let n be the number of elements in S

- Having an associative array S of size N elements is too costly in terms of space.
- Want to have S of size $O(n)$.

Hash Tables: Idea

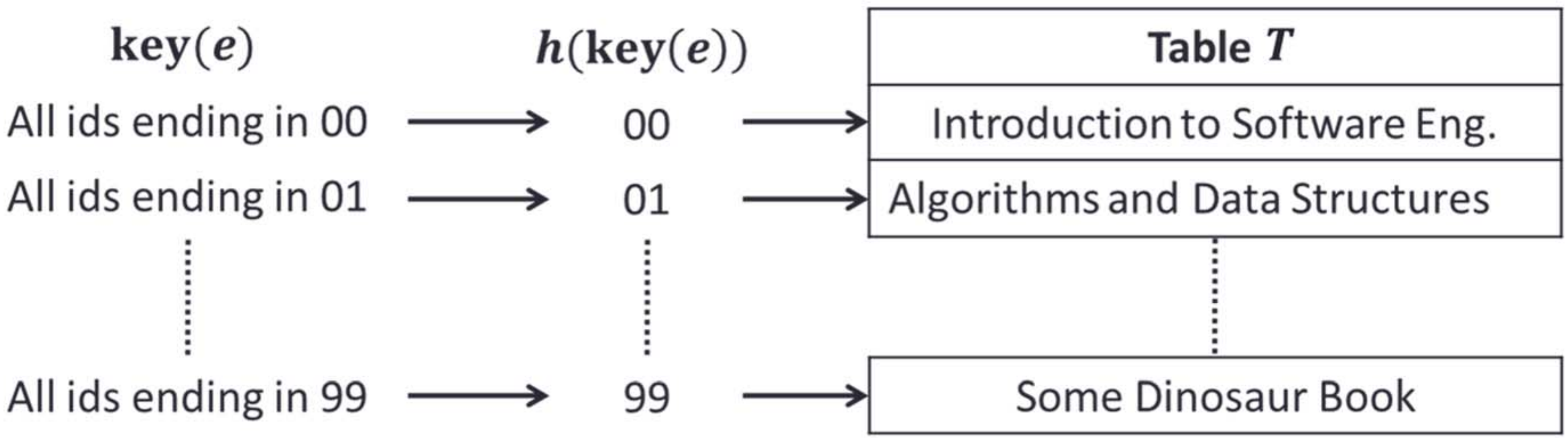
- Use *hash function* h to map potential keys to m values, where $m < N$.
- Let T be a *hash table* of size m.

- Store element e in index $h\big(\mathrm{key}(e)\big)$ of T.

Hash Tables: Hash Function

Example hash function:

- $\text{key}(e)$ are student ids,
- $h\big(\text{key}(e)\big)$ are last two digits of student ids.
- N is 10^7, m is 10^2.



Hash Tables: Hash Function

- A hash function $h\big(\text{key}(e)\big)$ is usually specified as the composition of two functions:

  Hash code:
  $\text{key}(e)$: keys $\rightarrow$ integers

  Compression function:
  $h(k)$: integers $\rightarrow [0, m-1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$h(\mathrm{e}) = h\big(\text{key}(e)\big)$$

- The goal of the hash function is to "disperse" the keys in a random way.

Hash Tables: Hash Codes

- Memory address:
  - We reinterpret the memory address of the key object as an integer.
  - Default hash code of all Java objects.
  - Does not work for numeric and string keys.
  - Also bad if objects can move (like in C#)!
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java/C#)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components, ignoring overflows.
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java/C#).

Hash Tables: Hash Codes

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits) as a_0, a_1 … a_{n-1}.
  - We evaluate the polynomial

    $p(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$

    at a fixed value z, ignoring overflows.
  - Especially suitable for strings (e.g., the choice z = 33 gives at most 6 collisions on a set of 50,000 English words)

Hash Tables: Compression Function

- Division:
  - $h(k) = k \bmod m$
  - The size m of the hash table is usually chosen to be a prime
  - The reason has to do with number theory…
- Multiply, Add and Divide (MAD):
  - $h(k) = (a \cdot k + b) \bmod m$
  - a and b are nonnegative integers such that $(a \bmod m) \neq 0$, otherwise, every integer would map to the same value b.

Hash Tables: Challenge

- Assume that the size of a hash table is a power of two, i.e. $m = 2, 4, 16, 32, \ldots$ etc.
- We map a key k into one of the m slots using the hash function $h(k) = k \bmod m$.
- Give one reason why this might be a bad selection for the hash function.

Hash Tables: Challenge

- Assume that the size of a hash table is a power of two, i.e. $m = 2, 4, 16, 32, \ldots$ etc.
- We map a key k into one of the m slots using the hash function $h(k) = k \bmod m$.
- Give one reason why this might be a bad selection for the hash function.

What happens if our keys are all even?

Hash Tables: Operations

Hash Tables follow the Map abstract data structure:

- get(k): if the map M has an entry with key k, return its associated value; else, return null.
- put(k, v): insert entry (k, v) with key k and value v into the map M; if key k is not already in M, then return null; else, return old value associated with k.
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- keys(): return an iterator of the keys in M.
- values(): return an iterator of the values in M.

Hash Tables: Collisions

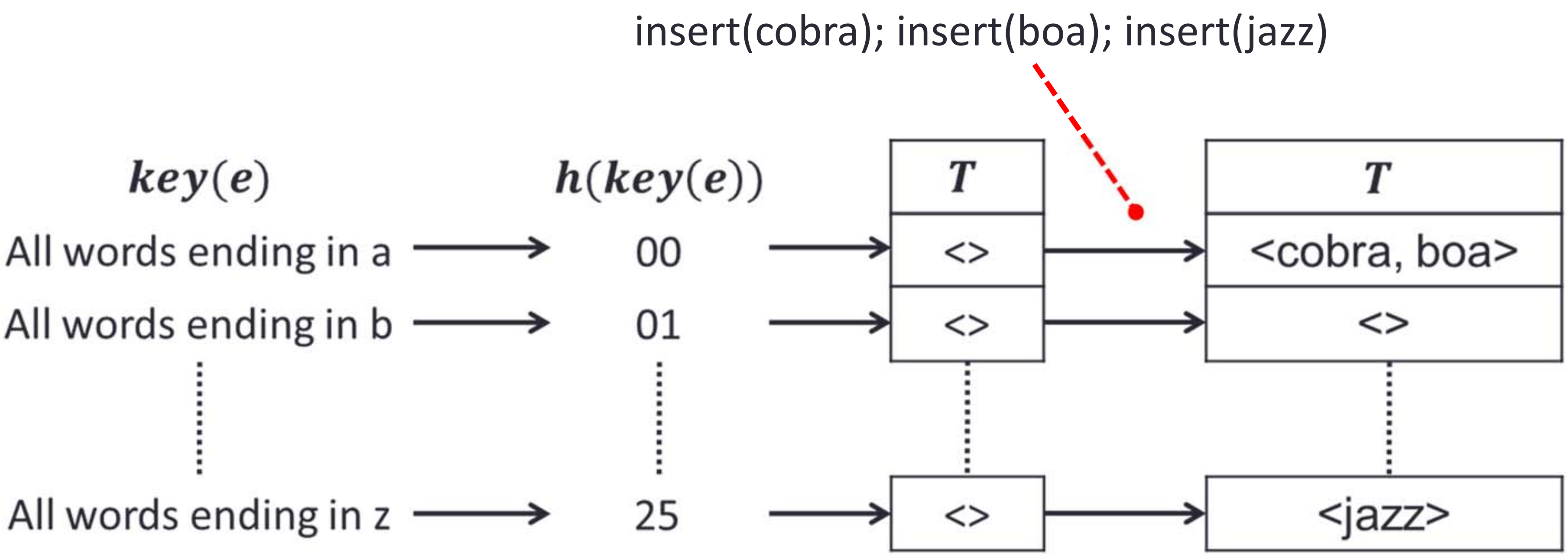
- We may use smaller tables to store elements, but this means that some elements may get stored in the same index.
- Previous example, a0000000 and a1995400.
- If only one element per table entry, only one element can be stored.

- 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: 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.

Hashing with Chaining: Insert(e)

- Insert(e: Element)
  - Get index $h\big(\text{key}(e)\big)$.
  - Add e to the end of the list in the table at $T\big[h\big(\text{key}(e)\big)\big]$.

- What is the worst case complexity?

Hashing with Chaining: Insert(e)

- Insert(e: Element)
  - Get index $h\big(\mathrm{key}(e)\big)$.
  - Add e to the end of the list in the table at $T\big[h\big(\mathrm{key}(e)\big)\big]$.

- Hash function's computation is $O(1)$.
- Worst case insert of linked list is $O(1)$.
- Thus Insert(e: Element) is $O(1)$.

* Note that we often have to perform Replace(e) instead of Insert(e) in case e is already presented in the hash table at $T\big[h\big(\mathrm{key}(e)\big)\big]$.
This operation is $O(n)$ as requires to run Find(e) first.

Hashing with Chaining: Find(k)

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

- What is the worst case complexity?

Hashing with Chaining: Find(k)

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

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

Hashing with Chaining: Remove(k)

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

- What is the worst case complexity?

Hashing with Chaining: Remove(k)

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

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

Hashing with Chaining: Average Case Analysis

Theorem: 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 $O(1 + \frac{n}{m})$.

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

Hashing with Chaining: 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 $\mathrm{E}[X]$ be the expected length of the list.

Thus the *expected* execution time is $O(1 + \mathrm{E}[X])$.

Hashing with Chaining: 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 X hashes to the same value as k, ie:

  if $h(\text{key}(e)) = h(k)$ then $X_e = 1$ else $X_e = 0$.

$$X = \sum_{e \in S} X_e$$ (i.e. how many elements are in table entry $h(\text{key}(e))$)

Hashing with Chaining: Average Case Analysis

Proof (continued):

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

$$= \sum_{e \in S} 1/m$$ (As function maps e to all m with equal probability)

$$= n/m$$ (Because n elements in S)

Hashing with Chaining: Average Case Analysis

Proof (continued):

Expected execution time is $O(1 + \mathrm{E}[X])$, and $\mathrm{E}[X] = \frac{n}{m}$.

Thus, the expected execution time for Remove and Find under hashing with chaining is $O(1 + \frac{n}{m})$, and constant if $m = \Theta(n)$.

Hashing: Alternative Approach to Chaining

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.
  - Each table cell inspected is referred to as a “probe”
  - Colliding items lump together, causing future collisions to cause a longer sequence of probes

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

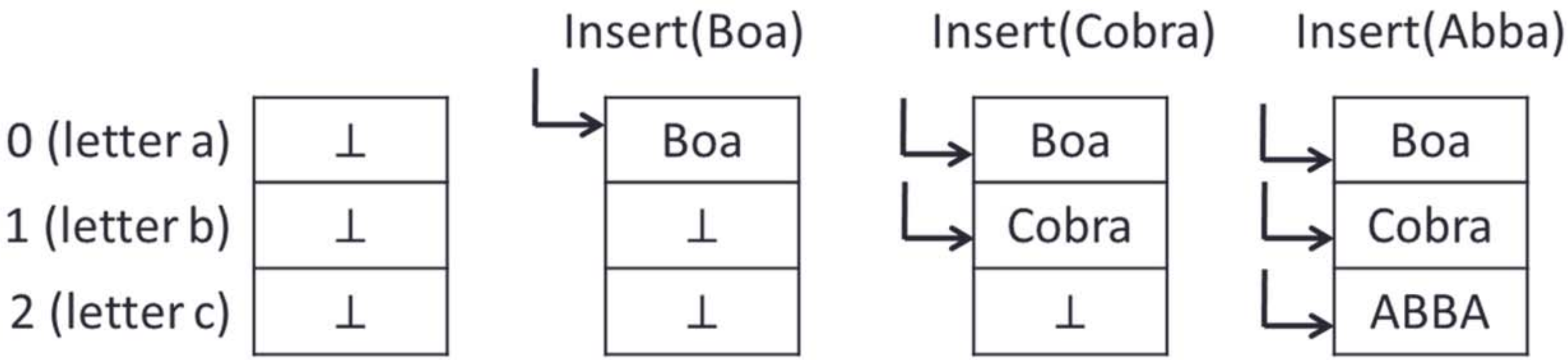
- Hashing with Linear Probing is an open hashing approach.
- All unused entries in T are set to $\perp$.
- When inserting, on a collision insert the element to the next free entry.

- Trivial fix: allow more entries (re-hash)
- Make table T size $m + m'$ instead of m.
  Choose $m' < m$.
- Is this a good fix? Is there a better way?

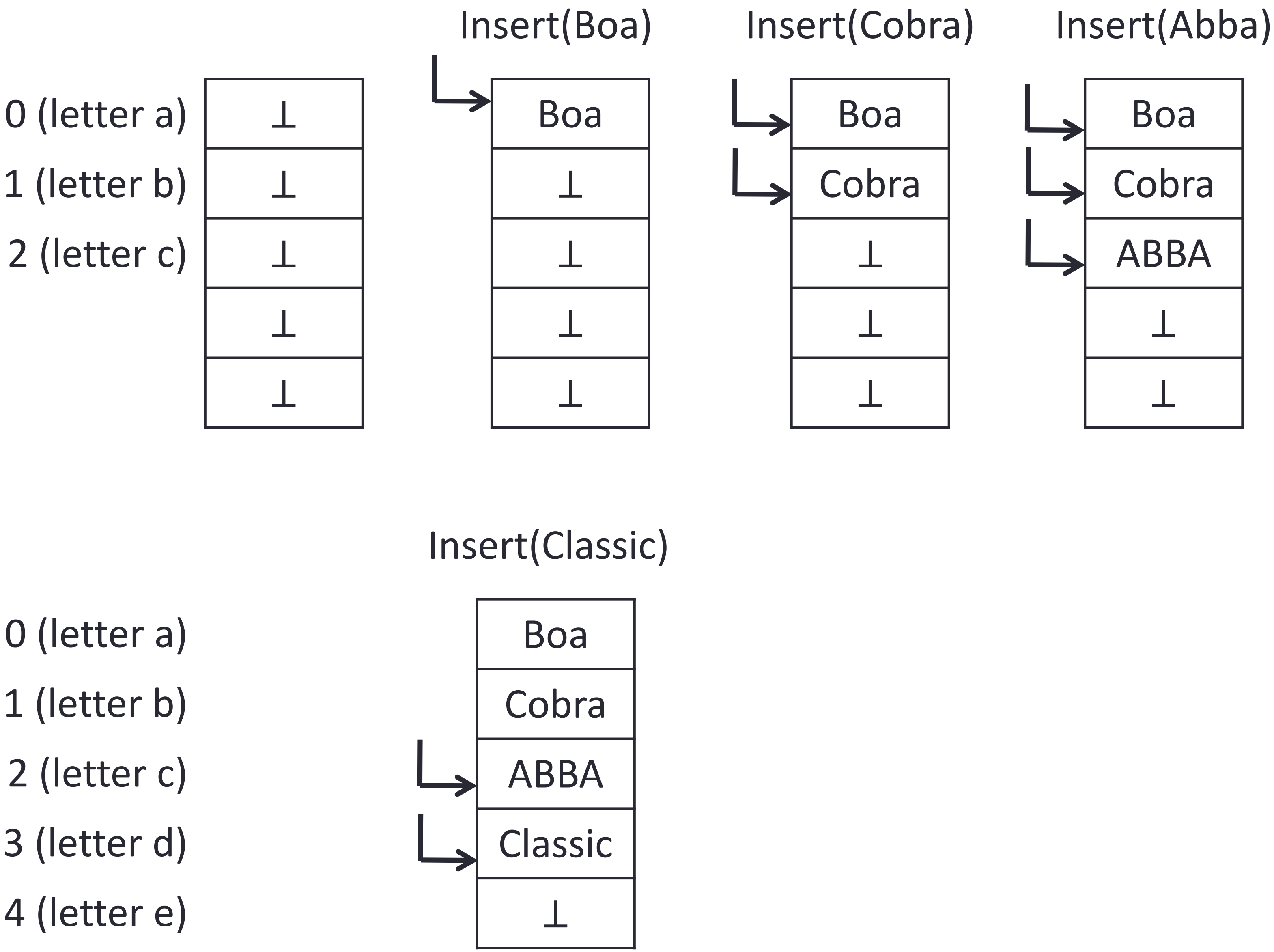
Hashing with Linear Probing: Insert(e)

Insert(e : Element)

1. Get index $i = h(\text{key}(e))$
2. If $T[i] = \perp$ (i.e. null), store e at $T[i]$
3. If $T[i]$ is not empty, increase i by 1 and go to step 2.



Hashing with Linear Probing: Insertion

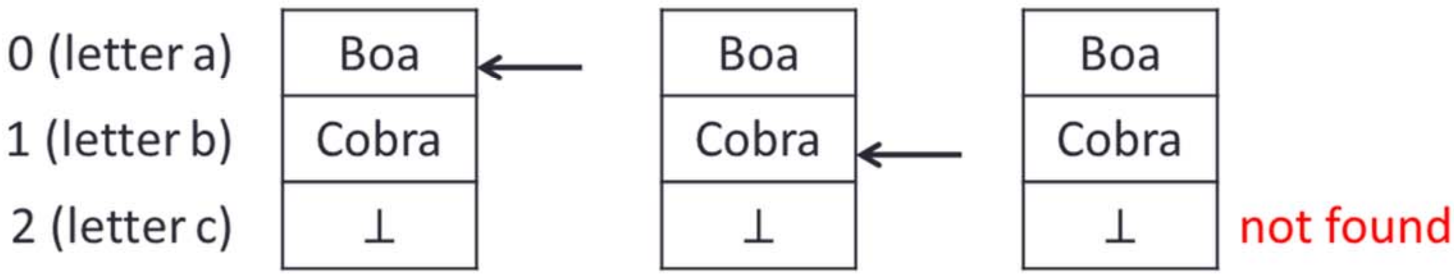


Hashing with Linear Probing: Find(k)

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 $\mathrm{key}(e) = k$, return found.
   Else increase i by 1 and go to step 2.

e.g. Find(ABBA)



Hashing with Linear Probing: Remove(k)

- Can not remove the element with $\mathrm{key}(e) = k$ and replace it with $\perp$.
- If we replace element e_1 at $T[i]$ with $\perp$, how do we find an element e_2 with the same $h(k)$?
- Instead, first remove the element with $\mathrm{key}(e) = k$ and then fix the invariant.

Hashing with Linear Probing: Remove(k)

Remove(k : Key)

1. Get index $i = h(k)$;

Find (k)

2. **If** ($T[i] == \perp$), return
3. If (element e at $T[i]$ has $\text{key}(e) \neq k$)
   increase e by 1 and go to step 2;

4. Set $T[i] = \perp$;

Repair

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 $T[j] = \perp$;
9. Set $i = j$ and go to step 5;

Hashing with Linear Probing: Deletion with Repairing

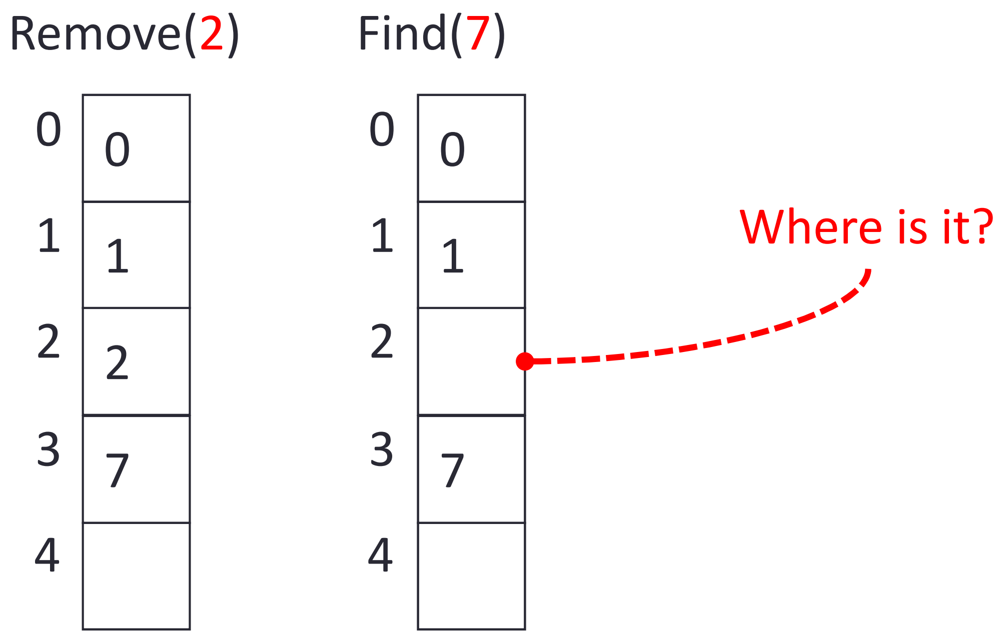
Remove(Cobra)

		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)	⊥	⊥	⊥

Hashing with Linear Probing: Lazy Deletion

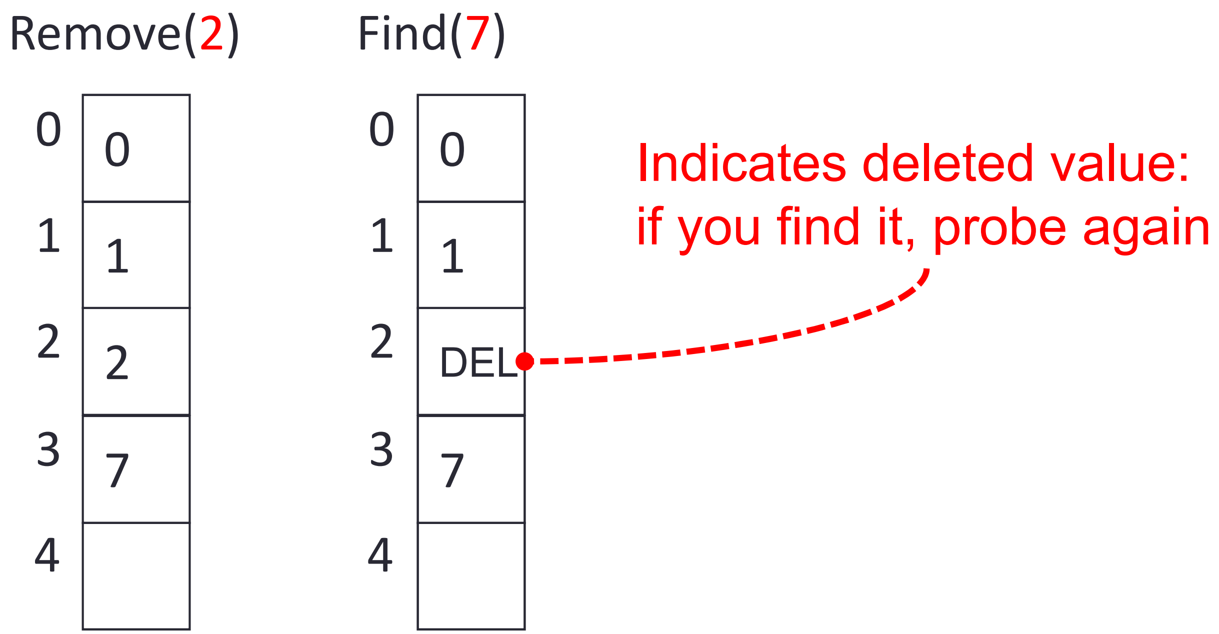
$$h(k) = k \bmod m$$



What should we do instead?

Hashing with Linear Probing: Lazy Deletion

$h(k) = k \bmod m$



But what is the problem now?

Hashing with Linear Probing: Lazy Deletion

- Use a special value **DELETED** instead of **NIL** when marking a slot as empty during deletion.
  - **Search** should treat DELETED as though the slot holds a key that does not match the one being searched for.
  - **Insert** should treat DELETED as though the slot were empty, so that it can be reused.
- **Disadvantage:** Search time is no longer dependent on the load factor $\alpha = n/m$.

  Hence, chaining is more common when keys have to be deleted.

Hashing with Linear Probing: Challenge

Suppose we have a hash table that resolves collisions using open addressing with linear probing. When removing an item from the table, we can either

- **repair the table** to get rid of possible incorrect search
- or **introduce a special marker** to skip the entry while searching. Therefore, slots with no keys contain either an EMPTY marker or a DELETED marker.

A student tries to reduce the number of DELETED markers. He\she proposes to use the following rules in the delete method:

1. If the object in the next slot is EMPTY,
   then a DELETED marker is not necessary.

2. If the object in the next slot has a different initial probe value,
   then a DELETED marker is not necessary.

Determine whether each of the above rules guarantees that searches return a correct result. Explain your answer.

Hashing: Chaining vs. Linear Probing

Argumentation depends on the intended use and many technical parameters:

Chaining

+ referential integrity
- waste of space

Linear probing

+ use of contiguous memory
- 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, etc. matter.

Hashing: Summary

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time.
- The worst case occurs when all the keys inserted into the map collide.
- The load factor $\alpha = n/m$ affects the performance of a hash table.
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is $1/(1-\alpha)$.
- In practice, hashing is very fast provided the load factor is not close to 100%
- When the load gets too high, we can re-hash....
- Applications: very numerous, e.g. computing frequencies.

Other references and things to do

- Read chapter 10.2 in Data Structures and Algorithms in Java. Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser, 2014.