COMP20007 Design of Algorithms

Hashing

Daniel Beck

Lecture 15

Semester 1, 2020

Dictionaries - Recap

- Abstract Data Structure: collection of *(key, value)* pairs.

Dictionaries - Recap

- Abstract Data Structure: collection of *(key, value)* pairs.
- Required operations: Search, Insert, Delete

Dictionaries - Recap

- Abstract Data Structure: collection of *(key, value)* pairs.
- Required operations: Search, Insert, Delete
- Last lecture: Binary Search Trees (and extensions)

$\Theta(n)$

$\Theta(\log n)$

Dictionaries - Recap

- Abstract Data Structure: collection of *(key, value)* pairs.
- Required operations: Search, Insert, Delete
- Last lecture: Binary Search Trees (and extensions)
- This lecture: Hash Tables.

Hash Tables

- A hash table is a continuous data structure with m preallocated entries.

ARRAY

Hash Tables

- A hash table is a continuous data structure with m preallocated entries.
- Average case performance for Search, Insert and Delete: $\Theta(1)$

Hash Tables

- A hash table is a continuous data structure with m preallocated entries.
- Average case performance for Search, Insert and Delete: $\Theta(1)$
- Requires a *hash function*: $h(K) \to i \in [0, m-1]$.

Hash Tables

- A hash table is a continuous data structure with m preallocated entries.
- Average case performance for Search, Insert and Delete: $\Theta(1)$
- Requires a *hash function*: $h(K) \to i \in [0, m-1]$.
- A hash function should:
  - Be efficient $\boxed{(\Theta(1)).}$
  - Distribute keys evenly (uniformly) along the table.

Identity Hash Function

STUDENT ID

734687 → POSITION

Question: if keys are integers why do I need a hash function?

I could just use the key as the index, no?

Identity Hash Function

Question: if keys are integers, why do I need a hash function? I could just use the key as the index, no?

- This is the *identity* hash function: $\boxed{h(K) = K.}$

Identity Hash Function

Question: if keys are integers, why do I need a hash function? I could just use the key as the index, no?

- This is the *identity* hash function: $h(K) = K$.
- Note that $K \in [0, m-1]$. In other words *we need to know the maximum number of keys in advance*.

Identity Hash Function

Question: if keys are integers, why do I need a hash function? I could just use the key as the index, no?

- This is the *identity* hash function: $h(K) = K$.
- Note that $K \in [0, m - 1]$. In other words *we need to know the maximum number of keys in advance*.
- Sometimes this is possible: postcodes, for example.

4 DIGITS

0 – 9999

Identity Hash Function

Question: if keys are integers, why do I need a hash function? I could just use the key as the index, no?

- This is the *identity* hash function: $h(K) = K$.
- Note that $K \in [0, m-1]$. In other words *we need to know the maximum number of keys in advance*.
- Sometimes this is possible: postcodes, for example.

  ↳ 10000 PREALLOCATE
- Many times it is not:
  - m is too large (need to preallocate)
  - Unbounded integers (student IDs)
  - Non-integer keys (games)

Hashing Integers

- For large/unbounded integers, an alternative function is $h(K) = K \mod m$

$m = 9$

$2 \mod 9 = 2$

$17 \mod 9 = 8$

$20 \mod 9 = 2$

Hashing Integers

- For large/unbounded integers, an alternative function is $h(K) = K \mod m$
- Allow us to set the size m.

Hashing Integers

- For large/unbounded integers, an alternative function is $h(K) = K \mod m$
- Allow us to set the size m.
- Small m results in lots of collisions, large m takes excessive memory. Best m will vary.

Hashing Strings

- Assume A $\mapsto 0$, B $\mapsto 1$, etc. 0 - 25
- Assume 26 characters and $m = 101$.
- Each character can be mapped to a *binary* string of length 5 ($2^5 = 32$). > 26

Hashing Strings

- Assume A $\mapsto 0$, B $\mapsto 1$, etc.
- Assume 26 characters and $m = 101$.
- Each character can be mapped to a *binary* string of length 5 ($2^5 = 32$).

We can think of a string as a long binary number:

M Y K E Y $\mapsto$ 0110011000010100010011000 (= 13379736)

4

$$13379736 \bmod 101 = 64$$

So 64 is the position of string M Y K E Y in the hash table.

Hashing Strings

We deliberately chose m to be prime.

$$13379736 = 12 \times 32^4 + 24 \times 32^3 + 10 \times 32^2 + 4 \times 32 + 24$$

With $m = 32$, the hash value of any key is the last character's value!

$m = 2^5$

Hashing Long Strings

Assume *chr* be the function that gives a character's number, so for example, $chr(\mathrm{c}) = 2$.

Hashing Long Strings

Assume *chr* be the function that gives a character's number, so for example, $chr(\mathsf{c}) = 2$.

Then we have

$h(s) = (\sum_{i=0}^{|s|-1} chr(s_i) \times 32^{|s|-i-1}) \bmod m$,

where m is a prime number. For example,

$h(\mathsf{V\ E\ R\ Y\ L\ O\ N\ G\ K\ E\ Y}) = (21 \times 32^{10} + 4 \times 32^{9} + \cdots)\ mod\ 101$

Hashing Long Strings

Assume *chr* be the function that gives a character's number, so for example, $chr(\mathrm{c}) = 2$.

Then we have

$h(s) = (\sum_{i=0}^{|s|-1} chr(s_i) \times 32^{|s|-i-1}) \bmod m$,

where m is a prime number. For example,

$h(\mathrm{V\ E\ R\ Y\ L\ O\ N\ G\ K\ E\ Y}) = (21 \times 32^{10} + 4 \times 32^{9} + \cdots)\ mod\ 101$

The term between parenthesis can become quite large and result in overflow.

Horner's Rule

Instead of

$$21 \times 32^{10} + 4 \times 32^{9} + 17 \times 32^{8} + 24 \times 32^{7} \cdots$$

factor out repeatedly:

$$(\cdots \; (\boxed{(21 \times 32 + 4)} \times 32 + 17) \times 32 + \cdots) + 24$$

$21 \times \boxed{32^9} \times 32 + 4 \times \boxed{32^9} =$

$= (21 \times 32 + 4) \times 32^9 \cdots\cdots$

Horner's Rule

Instead of

$$21 \times 32^{10} + 4 \times 32^{9} + 17 \times 32^{8} + 24 \times 32^{7} \cdots$$

factor out repeatedly:

$$(\cdots((21 \times 32 + 4) \times 32 + 17) \times 32 + \cdots) + 24 \quad \text{mod } m$$

Now utilize these properties of modular arithmetic:

$$\begin{aligned} (x + y) \bmod m &= \boxed{((x \bmod m) + (y \bmod m)) \bmod m} \\ (x \times y) \bmod m &= \boxed{((x \bmod m) \times (y \bmod m)) \bmod m} \end{aligned}$$

So for each sub-expression it suffices to take values modulo m.

Collisions

Happens when the hash function give identical results to two different keys.

Collisions

Happens when the hash function give identical results to two different keys.

We saw two solutions:

- Separate Chaining
- Linear Probing

Collisions

Happens when the hash function give identical results to two different keys.

We saw two solutions:

- Separate Chaining
- Linear Probing

Practical efficiency will depend on the table **load factor**:
$\alpha = n/m$ n = # total of records

Separate Chaining

Assign multiple records per cell (usually through a linked list)

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the n keys.

Separate Chaining

$0 < \alpha < 1$ $\alpha \geq 1$

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the n keys.
- A sucessful search requires $\boxed{1 + \alpha/2}$ operations on average.

$\hookrightarrow \sim 1$ operation

$\hookrightarrow \sim \alpha = 2$

$\hookrightarrow$ 2 operations

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the *n* keys.
- A sucessful search requires $1 + \alpha/2$ operations on average.
- An unsucessful search requires α operations on average.

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the *n* keys.
- A sucessful search requires $1 + \alpha/2$ operations on average.
- An unsucessful search requires α operations on average.
- Almost same numbers for Insert and Delete.

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the n keys.
- A sucessful search requires $1 + \alpha/2$ operations on average.
- An unsucessful search requires α operations on average.
- Almost same numbers for Insert and Delete.
- Worst case $\Theta(n)$ only with a bad hash function (load factor is more of an issue).

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the *n* keys.
- A sucessful search requires $1 + \alpha/2$ operations on average.
- An unsucessful search requires α operations on average.
- Almost same numbers for Insert and Delete.
- Worst case $\Theta(n)$ only with a bad hash function (load factor is more of an issue).
- Requires extra memory.

Linear Probing

Populate successive empty cells.

Linear Probing

Populate successive empty cells.

- Much harder analysis, simplified results show:
- A sucessful search requires $\boxed{(1/2) \times (1 + 1/(1 - \alpha))}$ operations on average.
- An unsucessful search requires $\boxed{(1/2) \times (1 + 1/(1 - \alpha)^2)}$ operations on average.

ONLY MAKES SENSE FOR

$0 < \alpha < 1$

Linear Probing

Populate successive empty cells.

- Much harder analysis, simplified results show:
- A sucessful search requires $(1/2) \times (1 + 1/(1-\alpha))$ operations on average.
- An unsucessful search requires $(1/2) \times (1 + 1/(1-\alpha)^2)$ operations on average.
- Similar numbers for Insert. Delete virtually impossible.

Linear Probing

Populate successive empty cells.

- Much harder analysis, simplified results show:
- A sucessful search requires $(1/2) \times (1 + 1/(1 - \alpha))$ operations on average.
- An unsucessful search requires $(1/2) \times (1 + 1/(1 - \alpha)^2)$ operations on average.
- Similar numbers for Insert. Delete virtually impossible.
- Does not require extra memory.

Linear Probing

Populate successive empty cells.

- Much harder analysis, simplified results show:
- A sucessful search requires $(1/2) \times (1 + 1/(1 - \alpha))$ operations on average.
- An unsucessful search requires $(1/2) \times (1 + 1/(1 - \alpha)^2)$ operations on average.
- Similar numbers for Insert. Delete virtually impossible.
- Does not require extra memory.
- Worst case $\Theta(n)$ with a bad hash function *and/or* clusters.

Double Hashing

A generalisation of Linear Probing.

Apply a second hash function in case of collision.

Double Hashing

A generalisation of Linear Probing.

Apply a second hash function in case of collision.

- First try: $h(K)$
- Second try: $(h(K) + s(K)) \mod m$
- Third try: $(h(K) + 2s(K)) \mod m$
- . . .

$\boxed{s(K) = 1}$

$(h(K) + 0 \times \cancel{s(K)}) \mod m = h(K)$

$(h(K) + 1 \times s(K)) \mod m =$

$= (h(K) + 1 \times 1) \mod m = h(K) + 1$

UNLESS $h(K) + 1 = m$

$= 0$

Double Hashing

A generalisation of Linear Probing.

Apply a second hash function in case of collision.

- First try: $h(K)$
- Second try: $(h(K) + s(K)) \mod m$
- Third try: $(h(K) + 2s(K)) \mod m$
- $\dots$

Another reason to use prime m in $h(K)$: will guarantee to find a free cell if there is one.

Double Hashing

A generalisation of Linear Probing.

Apply a second hash function in case of collision.

- First try: $h(K)$
- Second try: $(h(K) + s(K)) \mod m$
- Third try: $(h(K) + 2s(K)) \mod m$
- $\ldots$

Another reason to use prime m in $h(K)$: will guarantee to find a free cell if there is one.

Both Linear Probing and Double Hashing are sometimes referred as *Open Addressing* methods.

Rehashing

- High load factors deteriorate the performance of a hash table (for linear probing, ideally we should have $\alpha < 0.9$).

Rehashing

- High load factors deteriorate the performance of a hash table (for linear probing, ideally we should have $\alpha < 0.9$).
- *Rehashing* allocates a new table (usually around double the size) and move every item from the previous table to the new one.

Rehashing

- High load factors deteriorate the performance of a hash table (for linear probing, ideally we should have $\alpha < 0.9$).
- *Rehashing* allocates a new table (usually around double the size) and move every item from the previous table to the new one.
- *Very expensive operation*, but happens infrequently.

Summary

Hash Tables:

Summary

Hash Tables:

- Implement dictionaries.

Summary

Hash Tables:

- Implement dictionaries.
- Allow $\Theta(1)$ Search, Insert and Delete in the average case.

Summary

Hash Tables:

- Implement dictionaries.
- Allow $\Theta(1)$ Search, Insert and Delete in the average case.
- Preallocates memory (size m).

Summary

Hash Tables:

- Implement dictionaries.
- Allow $\Theta(1)$ Search, Insert and Delete in the average case.
- Preallocates memory (size m).
- Requires good *hash functions*.

Summary

Hash Tables:

- Implement dictionaries.
- Allow $\Theta(1)$ Search, Insert and Delete in the average case.
- Preallocates memory (size m).
- Requires good *hash functions*.
- Requires good collision handling.

Pros and Cons

If Hash Tables are so good, why bother with BSTs?

Pros and Cons

If Hash Tables are so good, why bother with BSTs?

- Hash Tables ignore *key ordering*, unlike BSTs.
- Queries like "give me all records with keys between 100 and 200" are easy within a BST but much less efficient in a hash table.

Pros and Cons

If Hash Tables are so good, why bother with BSTs?

- Hash Tables ignore *key ordering*, unlike BSTs.
- Queries like "give me all records with keys between 100 and 200" are easy within a BST but much less efficient in a hash table.
- Also: memory requirements of a hash table are much higher.

Pros and Cons

If Hash Tables are so good, why bother with BSTs?

- Hash Tables ignore *key ordering*, unlike BSTs.
- Queries like "give me all records with keys between 100 and 200" are easy within a BST but much less efficient in a hash table.
- Also: memory requirements of a hash table are much higher.

That being said, if hashing is applicable, a well-tuned hash table will typically outperform BSTs.

In Practice

Python dictionaries (*dict* type)

In Practice

Python dictionaries (*dict* type)

- Open addressing using *pseudo-random probing*
- Rehashing happens when $\alpha = 2/3$

In Practice

Python dictionaries (*dict* type)

- Open addressing using *pseudo-random probing*
- Rehashing happens when $\alpha = 2/3$

C++ *unordered_maps*

In Practice

Python dictionaries (*dict* type)

- Open addressing using *pseudo-random probing*
- Rehashing happens when $\alpha = 2/3$

C++ *unordered_maps*

maps → COMPLEX QUERIES

- Uses chaining.
- Rehashing happens when $\alpha = 1$

In Practice

Python dictionaries (*dict* type)

- Open addressing using *pseudo-random probing*
- Rehashing happens when $\alpha = 2/3$

C++ *unordered_maps*

- Uses chaining.
- Rehashing happens when $\alpha = 1$

Next lecture: what happens if records/data is too large?