Hashtables

Denis Espino

Introduction

- Hashtables are a collection of key-value pairs
- They are an implementation of the *Associative Array* or *Dictionary* ADT
- They are a collection of key-value pairs
- Operations are: Insert, Lookup, Remove

Hashtables

Hashtables are implementations of the *associative array* ADT:

- stores key-value pairs inside of an *array*,
- use a *hashing function* to map keys to an index in the array
- uses a *collision resolution scheme* when two different keys result in the same index

Hash function - String

```
function hashString(string, size) {
  let result = 0;
  for (let i = 0; i < string.length; i++)
    result +=
      string.charCodeAt(i) *
      Math.pow(31, i);
  return result % size;
}
```

$$\begin{aligned} \mathrm{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m, \end{aligned}$$

Hash function - Integer

```
function hashInteger(integer, size) {
  return integer % size;
}
```

Example

```
hash('andy')    = ?
```

```
emailTable = {
 andy:
   andy794@email.com,
}
```

0	1	2	3

Example

i	0	1	2	3
Letter	a	n	d	y
ASCII	97	110	100	121
`s[i] * p`^{i}	97	3410	96100	3604711
Total				3704318

3704318 mod 4 = 2

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

Example

```
hash('andy')    = 2
```

```
emailTable = {
 andy:
    andy794@email.com,
}
```

0	1	2	3
		andy	
		andy794@email.com	

Example

```
hash('andy')    = 2
hash('daphne') = 0
```

```
emailTable = {
 andy:
   andy794@email.com,
 daphne:
   daph485@email.com,
}
```

0	1	2	3
daphne		andy	
daph485@email.com		andy794@email.com	

Example

```
emailTable = {
 andy:
   andy794@email.com,
 daphne:
   daph485@email.com,
 betty:
   bett129@email.com,
}
```

```
hash('andy')    = 2
hash('daphne') = 0
hash('betty')   = 2
```

0	1	2	3
daphne		andy	
daph485@email.com		andy794@email.com	
		betty	
		bett129@email.com	

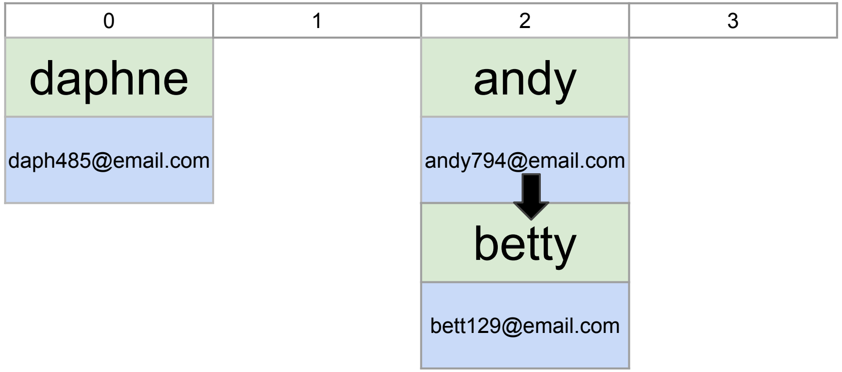
Collision Resolution Schemes

- Chaining: Each bucket in the array stores a linked list
- Open Addressing: Each bucket in the array stores the key-value pairs

The collision resolution scheme affects how operations are implemented.

Chaining

Each bucket is a linked list of key-value pairs. Collided items are chained together through a linked list.



Chaining - Insertion

```
Insert(key, value):

If load factor is too high
  then Grow

Get the index associated with the key
Get the linked list associated with the index
If the key is found in the linked list
  then update the key-value pair with the new value
else
  insert key-value pair to end of linked list
```

Chaining - Lookup

```
Lookup(key):

Get the index associated with the key
Get the linked list associated with the index
If the key is found in the linked list
  then return the associated value
else
  return undefined, null, or throw error
```

Chaining - Remove

```
Remove(key):

If load factor is too low
  then Shrink

Get the index associated with the key
Get the linked list associated with the index
If the key is found in the linked list
  then remove the key-value pair from the linked list
else
  nothing happens
```

Chaining Pros and Cons

Pros

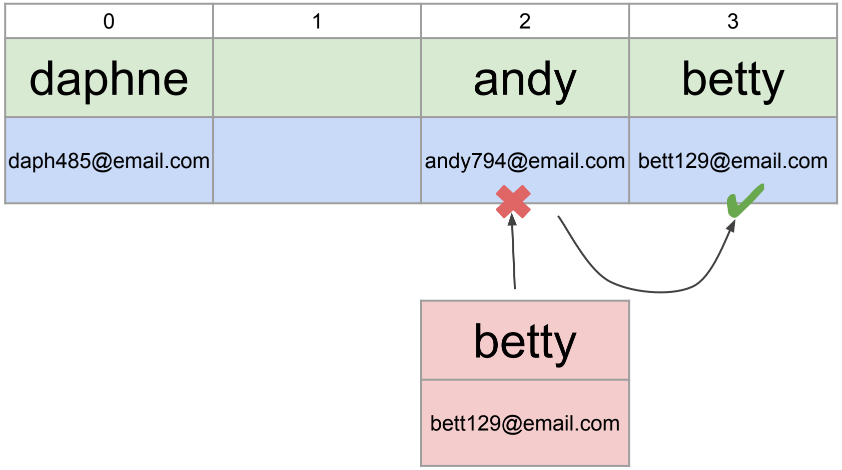
- Easy to implement compared to other collision resolution techniques such as open addressing.
- Good performance when used with an appropriate hash function with constant time insert, lookup, and remove.

Cons

- Increased memory usage than other techniques for large number of tables with small linked lists.
- Decreased cache efficiency, elements in a linked list can be slower to access than accessing elements from an array.

Open Addressing with Probing

Open addressing means to store the key-value pairs directly in the buckets. Probing means checking slots in a sequence.

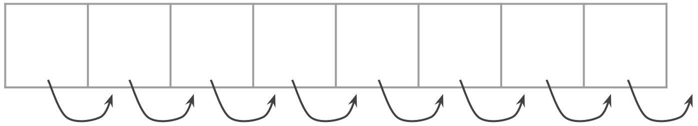


Probing Sequences

When a collision occurs, probing will check other slots in the array until it finds what it is looking for. It checks the slots according to a sequence.

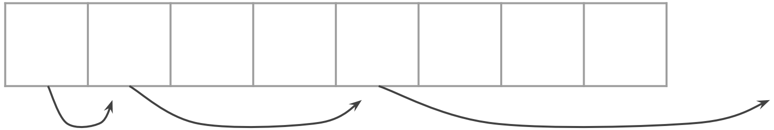
Linear Probing:

H, H+1, H+2, H+3, ...



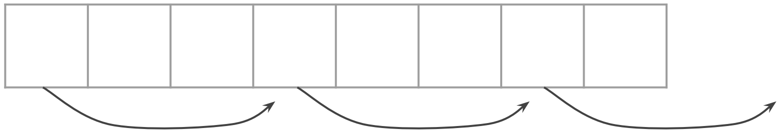
Quadratic Probing:

H, H+1, H+4, H+ 9, ...



Double Hashing

H, $H+H_2$, $H+2H_2$, $H+3H_2$, ...



Open Addressing and Deletion

0	1	2	3
daphne		andy	betty
daph485@email.com		andy794@email.com	bett129@email.com

```
hash('andy')   = 2 > 2
hash('daphne') = 0
hash('betty')  = 2 > 3
```

Finding the key ‘betty’:

- Search at index 2 because hash(‘betty’) = 2.
- Use probing until we find ‘betty’ or an empty slot.

Open Addressing and Deletion

0	1	2	3
daphne			betty
daph485@email.com			bett129@email.com

```
hash('daphne') = 0
hash('betty')  = 2
```

Finding 'betty' after deleting 'andy':

- The key 'betty' is not found at index 2.
- Index 2 is empty so we stop searching.
- The algorithm believes the key-value pair must not exist in the table.
- But this is wrong!

Tombstone Deletion 👻

To remove an element you replace it with a special value called a **tombstone**. It indicates that an element used to be here but has been removed. A tombstone does not make the search algorithm stop.

Open Addressing and Deletion

0	1	2	3
daphne		👻	betty
daph485@email.com			bett129@email.com

Finding ‘betty’ after deleting ‘andy’:

- Deleting a key-value pair leaves behind a tombstone.
- A tombstone means this space is empty but don’t stop searching.
- Search at index 2 because hash(‘betty’) = 2.
- Use probing until we find ‘betty’ or an empty slot.

Open Addressing - Insert

```
if load factor is too high
  then grow

get bucket index using the key hash
while true
  if bucket contains key
    then update key-value pair and exit
  else if bucket is tombstone
    then remember this tombstone only if no tombstone is currently being remembered
  else if bucket is empty
    if tombstone is currently being remembered
       then insert key-value pair at the tombstone and exit
    else
     then insert key-value pair at this bucket and exit
  else
    then get next bucket index using the key hash and probing algorithm
```

Open Addressing - Lookup

```
get bucket index using the key hash
while true
if bucket contains key
  then return value
else if bucket is empty
  return undefined, null, or throw error
else
  then get next bucket index using the key hash and probing algorithm
```

Open Addressing - Remove

```
if load factor is too low
  then shrink

get bucket index using the key hash
while true
if bucket contains key
  then set bucket to tombstone and break
else if bucket is empty
  do nothing
else
  then get next bucket index using the key hash and probing algorithm
```

Growing / Shrinking

Load Factor = Entries / Table size

Table size should double when load factor reaches 0.75

Table size should half when load factor reaches 0.25

Runtime

	Average	Worst Case
Space	$\Theta(n)$	$O(n)$
Search	$\Theta(1)$	$O(n)$
Insert	$\Theta(1)$	$O(n)$
Delete	$\Theta(1)$	$O(n)$

Resources

- Articles
  - https://en.wikipedia.org/wiki/Hash_table
  - https://en.wikipedia.org/wiki/Associative_array
  - https://cp-algorithms.com/string/string-hashing.html
  - https://stackoverflow.com/questions/60641061/how-does-linear-probing-handle-deletions-without-breaking-lookups
- Videos
  - https://www.youtube.com/watch?v=knV86FlSXJ8&ab_channel=MichaelSambol
  - https://www.youtube.com/watch?v=FsfRsGFHuv4&ab_channel=BroCode
  - https://www.youtube.com/watch?v=7eLDTtbzX4M&ab_channel=WilliamFiset
- Visualizer
  - https://visualgo.net/en/hashtable