


Estruturas de Dados / Programação 2
Hash Tables

Márcio Ribeiro
marcio@ic.ufal.br
twitter.com/marciomribeiro

Introduction

- Linear search
  - O(n)
- Binary search
  - O(log n)

That's it?

How about finding elements in constant time O(1)?!

Employee records

- How to keep 120 employee records of a company in our program?
- Which data structure would you use?!

Arrays?

- Records alphabetically; binary search on the name key
- Then, we come up with:
  - Search: O(log n)
  - Insert: O(n)
- How to access one particular word in constant time O(1)?

Andrew a@se.com	...	Jones j@se.com	...	Yuri y@se.com
0	...	23	...	119

Index: accessing in constant time O(1)

```
h(name) = capitalAsciiSum % 120
```

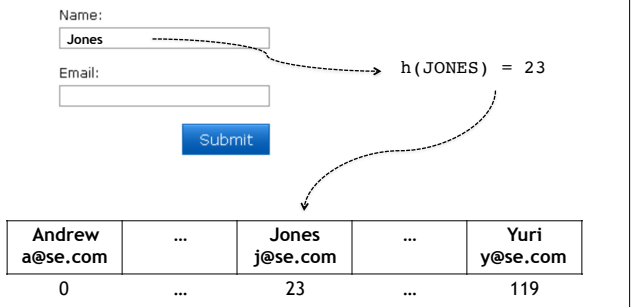
Name:
Jones
Email:
Submit

J → 74
O → 79
N → 78
E → 69
S → 83

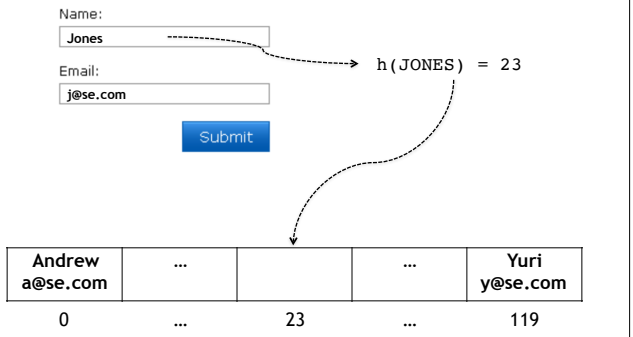
```
h(JONES) = 383 % 120 = 23
```

Now, we have a function that…

- … maps input keys (names) into array indexes!



And we can use this function not only to search but to insert!



Actually we can hash anything…

- Strings
- IDs
- Dates
- As long as we could come up with a good way to transform the key into an array index

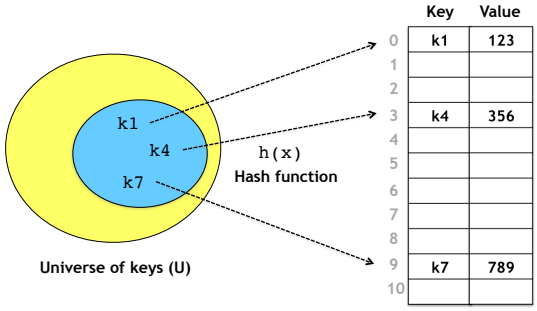
Now…

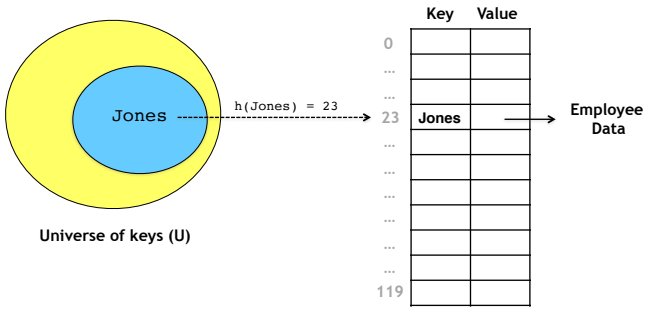
We can access elements in constant time O(1)

We can insert elements in constant time O(1)

Hash Tables

Hash tables





Hash function

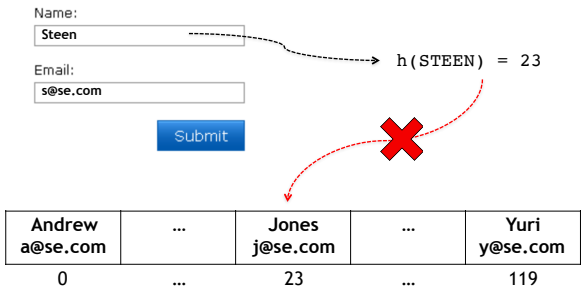
- Why should we use %?

```
h(name) = capitalAsciiSum % 120
```

- We do not cross the array boundaries
- Be careful with the function's overhead and choice!

Inserting employee "Steen"

```
h(name) = capitalAsciiSum % 120
```



Collision!

Another example

- Input: first name letter + first ID number
- What do you think about this hash function?

```
h(x) = (n[0] * 256 + id[0]) * 2654435761 % 128;

h(a1) = 97
h(b1) = 97
h(c1) = 97
h(d1) = 97
h(e1) = 97
…
```

From *Prime Double Hash Table*, T. Wang, 1997

Prime numbers help!

```
h(x) = (x[0] * 256 + x[1]) * 2654435761 % 113;
```

```
h(a1) = 90
h(b1) = 11
h(c1) = 45
h(d1) = 79
h(e1) = 97
…
```

```
(1 * 256 + C) % 128;
(2 * 256 + C) % 128;
```

But they are still not enough…

- Input: memory addresses

x	h(x) = x % 8	h(x) = x % 7
0	0	0
4	4	4
8	0	1
12	4	5
16	0	2
20	4	6
24	0	3
28	4	0

So, it boils down to:

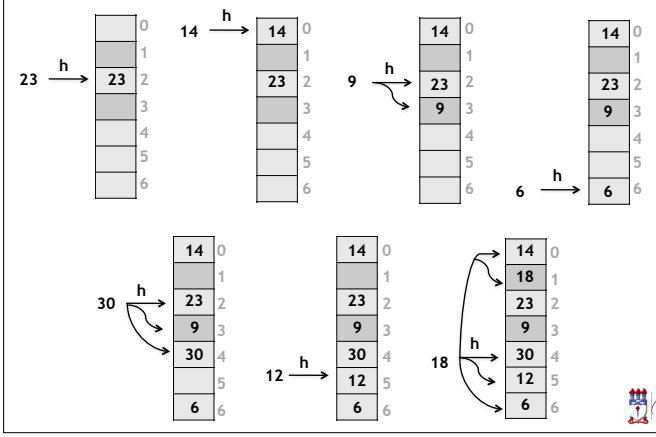
- Prime numbers help to better distribute data among the hash table
- They help to spread
- They can reduce collisions

So, we still need to deal with the collision problem!

Solution 1: Linear resolution

- h(x) points to an index already occupied?
- Try the neighborhood

Linear resolution example



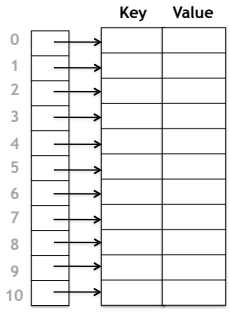
Abstract Data Type: Hash Table

Hash table ADT

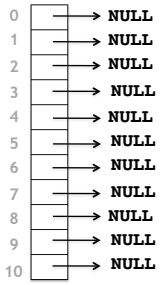
```
hash_table* create_hash_table();

void put(hash_table *ht, int key, int value);

int get(hash_table *ht, int key);

void remove(hash_table *ht, int key);

int contains_key(hash_table *ht, int key);

void print_hash_table(hash_table *ht);
```

Structs

```
struct element {
  int key;
  int value;
};


struct hash_table {
  element *table[11];
};
```



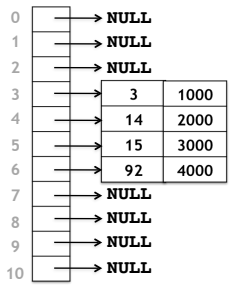
```
hash_table* create_hash_table()
{
  hash_table *new_hash_table =
        (hash_table*) malloc(sizeof(hash_table));

  int i;
  for (i = 0; i < 11; i++) {
    new_hash_table->table[i] = NULL;
  }
  return new_hash_table;
}
```



```
void put(hash_table *ht, int key, int value)
{
  int h = key % 11;
  while (ht->table[h] != NULL) {
    if (ht->table[h]->key == key) {
      ht->table[h]->value = value;
      break;
    }
    h = (h + 1) % 11;
  }

  if (ht->table[h] == NULL) {
    element *new_element =
        (element*) malloc(sizeof(element));
    new_element->key = key;
    new_element->value = value;
    ht->table[h] = new_element;
  }
}
```

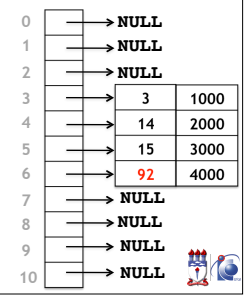
Put function in action

```
hash_table *ht = create_hash_table();

put(ht, 3, 1000);
put(ht, 14, 2000);
put(ht, 15, 3000);
put(ht, 92, 4000);
```



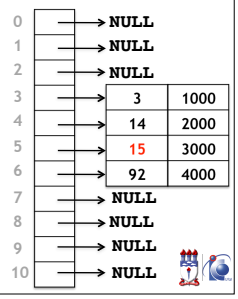
```
int get(hash_table *ht, int key)
{
  int h = key % 11;
  while (ht->table[h] != NULL) {
    if (ht->table[h]->key == key) {
      return ht->table[h]->value;
    }
    h = (h + 1) % 11;
  }
  return -100;
}
```

```
get(ht, 92);
get(ht, 16);
```




```
void remove(hash_table *ht, int key)
{
  int h = key % 11;
  while (ht->table[h] != NULL) {
    if (ht->table[h]->key == key) {
      free(ht->table[h]);
      ht->table[h] = NULL;
    }
    h = (h + 1) % 11;
  }
}
```

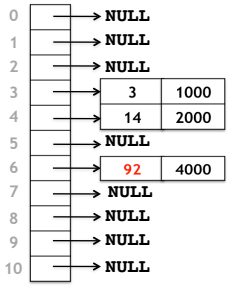
```
remove(ht, 15);
```



Getting the element whose key is 92...

```
get(ht, 92);
```

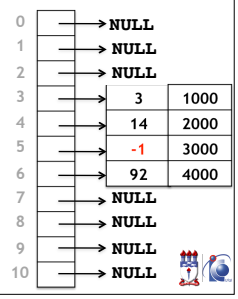
Returns -100, which means that the key was not found!



Remove (correct version)

```
void remove(hash_table *ht, int key)
{
  int h = key % 11;
  while (ht->table[h] != NULL) {
    if (ht->table[h]->key == key) {
      free(ht->table[h]);
      ht->table[h] = NULL;
      ht->table[h]->key = -1;
    }
    h = (h + 1) % 11;
  }
}
```

```
remove(ht, 15);
get(ht, 92);
```



Infinite loop!
Homework: fix it!

Poscomp 2009

Questão 31. [FUN]

Considere uma tabela de espalhamento (tabela *hash*) de comprimento $m = 11$, que usa endereçamento aberto (*open addressing*), a técnica de tentativa linear *(linear probing)* para resolver colisões e com a função de dispersão (função *hash*) $h(k) = k \bmod m$, onde k é a chave a ser inserida. Considere as seguintes operações sobre essa tabela:

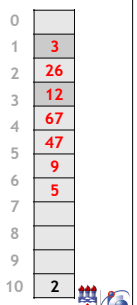
- Inserção das chaves 3, 14, 15, 92, 65, 35 (nesta ordem);
- Remoção da chave 15; e
- Inserção da chave 43.

Escolha a opção que representa esta tabela após estas operações:

A) 65 – ø – 35 – 14 – ø – 92 – 3 – ø – ø – ø – 43
B) 43 – ø – 35 – 3 – 14 – 92 – ø – ø – ø – ø – 65
C) 65 – ø – 35 – X - 14 – 92 – 3 – ø – ø – ø – 43
D) 65 – ø – 35 – 3 – 14 – 92 – ø – ø – ø – ø – 43
E) 43 – ø – 35 – 3 – 14 – X – 92 – ø – ø – ø – 65

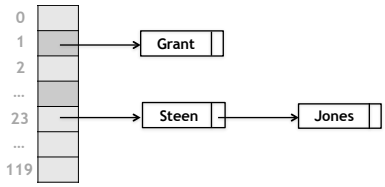
Linear resolution problems

- Clustering
  - We can mitigate: function that randomizes as much as possible
  - Besides linear
    - Quadratic: 1, 4, 9, … (almost full: bad performance)
    - Double hashing: another hash function
- In the presence of clustering, it is not efficient!
  - We cannot guarantee O(1) as we promised… ☹



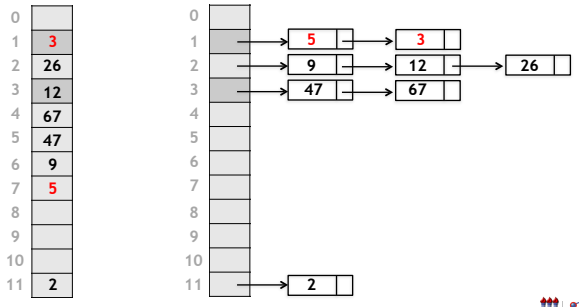
Solution 2: separate chaining

- Linear resolution: open addressing
- Separate chaining: closed addressing



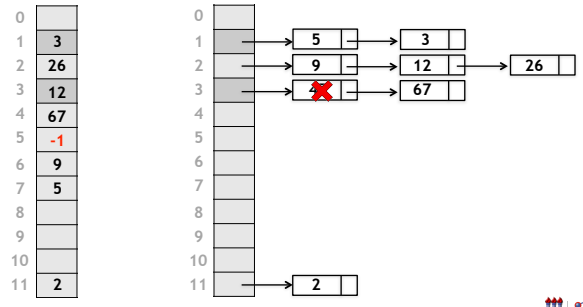
Linear *versus* Linked List: h(3) = h(5) = 1

- get(5): 7 comparisons versus 1 comparison



Removing key 47

- Linked list: no need to keep "-1" as a flag…



Application

Caching

Expensive operations

- We should avoid calling them as much as possible
- For example… if we know that
  - key = 3; then result = 220715427
  - Do we need to call this operation again?!

```
expensiveOperation(3)
```

```
public void doSomething(int key) {
  int result = expensiveOperation(key);

  // do something with result…
}
```

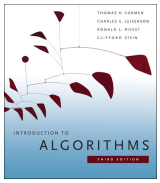
No! We do not!

- Store the result in a hash table and get it when key = 3!

```
public void doSomething(int key) {
  int result = 0;
  Map cache = new HashMap();

  if (cache.containsKey(key)) {
    result = cache.get(key);
  } else {
    result = expensiveOperation(key);
    cache.put(key, result);
  }

  // do something with result…
}
```

Hit *versus* Miss

References



Chapter 11

Chapter 10