Design & Analysis of Algorithms
COMP 482 / ELEC 420

John Greiner

Python Dictionaries

Same idea as Java/C++ hash map, C# dictionary, Perl hash, …

How is this “magic” implemented?

Combination of Ideas

- Hash table & hash function
- Dynamic table & amortization

Hash Tables & Hash Functions

Access Time Depends on Chain Length

- What property do we want of our hash function?
- How long is each chain?
- How much time per access?

Creating a Good Hash Function is Difficult

Generally, just use those in libraries.

```python
class string:
    def __hash__(self):
        if not self:
            return 0 # empty
        value = ord(self[0]) << 7
        for char in self:
            value = c_mul(1000003, value) ^ ord(char)
        value = value ^ len(self)
        if value == -1: # reserved error code
            value = -2
        return value

key.__hash__() % table_size
```
Dynamic Table Motivation

Typically, don’t know how much data we’ll have.
- Want underlying hash table to grow, so average chain size is bounded.
- Want to retain constant-time indexing.

Focus on the latter goal first.
- We’ll have to do a little extra to combine hash tables & dynamic tables.

Adding Data when Dynamic Table is Full

Must be contiguous for constant-time indexing.

Adding Data when Dynamic Table is Full

What’s wrong with just using space at end of array?

That memory might already be in use.

Adding Data when Dynamic Table is Full

So, grab needed space elsewhere & copy everything.

Double the space.

Cost of a Series of Operations

Initially: Table size = 5, table empty

Cost of a Series of Operations

Add 5 data items, cost = 5
**Cost of a Series of Operations**

Add 5 more data items, cost = 10

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total costs:
- Copying = 15
- Adding = 20
- Total = 35

**Cost of a Series of Operations**

Add 5 more data items, cost = 15

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Cost of a Series of Operations**

Add 5 more data items, cost = 5

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Cost of Copying is Proportional to Adding**

Buy Add 1
Get Copy 2 free!

Each copy costs
- $2 \times \#\text{adds since previous copy.}$

Use expensive ops sufficiently infrequently.

**Amortized Cost**

Cost of series of $n$ operations = $m$
Each operation has amortized cost = $m/n$

Dynamic tables: $O(1)$ amortized time to add data

**Dynamic Hash Tables**

```
d = {"snow": 7, "apple": 3, "white": 20}
d["white"] = 20
d["prince"] = 16
d["apple"]
```

"snow" hash 3
"apple" hash 2
"white" hash 2
"prince" hash 1

Load factor = #items/size

Hash table "full enough."

Must rehash all data.
Hash Table & Dynamic Table Odds & Ends

Hash table size: prime or power-of-2?
- Chaining vs. open addressing
- Dynamic table expansion factor
- Dynamic table contraction
- Python’s dictionary/hash table implementation

Dynamic tables:
- Python list
- Memory heaps for GC

When want static memory size.

Incrementing Binary Counter

<table>
<thead>
<tr>
<th>Counter</th>
<th>Bits changed in increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>4</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>2</td>
</tr>
<tr>
<td>1010</td>
<td>1</td>
</tr>
<tr>
<td>1011</td>
<td>3</td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>2</td>
</tr>
<tr>
<td>1110</td>
<td>1</td>
</tr>
<tr>
<td>1111</td>
<td>5</td>
</tr>
</tbody>
</table>

Another way to sum the costs?

<table>
<thead>
<tr>
<th>Bit position</th>
<th># times bit changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

\[ \sum_{i=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^i} \rceil = O(?) \]

Amortized Analysis Approaches

- Accounting:
  1. Compute actual costs \( c_i \) of each kind of op.
  2. Define accounting costs \( \bar{c}_i \) of each kind of op, such that can assign credits \( \bar{c}_i - c_i \) consistently to data elts.
  
  - Can “overpay” on some ops, and use credits to “underpay” on other ops later.
  
  - Poor definitions lead to loose bounds.
  3. \( O(\text{max acct. cost}) \)

- Potential:
  1. Define potential function \( \Phi(D) \), such that \( \Phi(D) \geq \Phi(D_0) \).

  - Essentially assigns “credits” to data structure, rather than operations.
  
  - Poor definition leads to loose bounds.
  2. Calculate accounting costs \( \bar{c}_i = c_i + \Delta \Phi \) of each kind of op.
  3. \( O(\text{max acct. cost}) \)

Complicated approach necessary for more complicated data structures.

Multipop

<table>
<thead>
<tr>
<th>3 ops:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Push(S,x)</td>
<td>Pop(S)</td>
</tr>
<tr>
<td>Multi-pop(S,k)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Worst-case cost:</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
</tr>
<tr>
<td>O(1)</td>
</tr>
<tr>
<td>O(min(</td>
</tr>
</tbody>
</table>

Amortized cost: ?