CSE 373: Data Structures and Algorithms
Lecture 6: Finishing Amortized Analysis; Dictionaries ADT; Introduction to Hash Tables

Instructor: Lilian de Greef
Quarter: Summer 2017
Today:

- Finish up Amortized Analysis
- Dictionary ADT
- Introduce Hash Tables
Reminder: No class on Monday!

Unofficial holiday – have a good 4-day weekend! 😊

- Will ask you to do a ~30 minute activity to make up for last class time
- Remember that homework 2 is also due the day after we’re back
Homework 2 update:

• There was a typo (woops!) for problem 7
• The website now has a corrected version.
Amortized Analysis

How we calculate the average time!
Amortized Cost

The amortized cost of $n$ operations is the worst-case total cost of the operations divided by $n$.

Shorthand:

If $T(n) =$ worst-case (upper bound) of total cost for $n =$ number of operations

$\Rightarrow \text{Amortized Cost} = T(n) / n$
Example: Array Stack

What’s the amortized cost of calling \texttt{push()} \( n \) times if we double the array size when it’s full?

\( n \) operations:

- \( n \) pushes at \( O(1) \) each \( \rightarrow \) total cost = \( n \)
- cost of resizing = \( n + n/2 + n/4 + n/8 + \ldots \leq 2n \) \( \rightarrow \) total cost \( \leq 2n \)

\( \rightarrow T(n) = n + 2n = 3n \)

\( \rightarrow \) Amortized cost = \( T(n)/n = 3n/n = 3 \)

\( \rightarrow \) Amortized Running time = \( O(1) \)

The \textbf{amortized cost} of \( n \) operations is the worst-case total cost of the operations divided by \( n \).
Another Perspective: Paying and Saving “Currency”

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

1 operation costs us 1$ to the computer
Another Perspective: Paying and Saving “Currency”

Use $2 for each push:
- $1 to computer,
- $1 to bank

Use $2 for each push:
- $1 to computer,
- $1 to bank

Spend our savings in the bank to resize. That way it only costs $1 to push(E)!
Example #2: Queue made of Stacks

A sneaky way to implement Queue using two Stacks

Example walk-through:

• enqueue A
• enqueue B
• enqueue C
• dequeue
• enqueue D
• enqueue E
• dequeue
• dequeue
• dequeue
Example #2: Queue made of Stacks

A sneaky way to implement Queue using two Stacks

```java
class Queue<E> {
    Stack<E> in = new Stack<E>();
    Stack<E> out = new Stack<E>();
    void enqueue(E x) { in.push(x); }
    E dequeue() {
        if (out.isEmpty()) {
            while (!in.isEmpty()) {
                out.push(in.pop());
            }
        }
        return out.pop();
    }
}
```

Wouldn’t it be nice to have a queue of t-shirts to wear instead of a stack (like in your dresser)? So have two stacks

- **in**: stack of t-shirts go after you wash them
- **out**: stack of t-shirts to wear
- if **out** is empty, reverse **in** into **out**
Example #2: Queue made of Stacks (Analysis)

```java
class Queue<E> {
    Stack<E> in = new Stack<E>();
    Stack<E> out = new Stack<E>();
    void enqueue(E x) { in.push(x); }
    E dequeue() {
        if (!out.isEmpty()) {
            while (!in.isEmpty()) {
                out.push(in.pop());
            }
        }
        return out.pop();
    }
}
```

Assume stack operations are (amortized) $O(1)$.

What’s the worst-case for `dequeue()`?
- $O(n)$ (everything is in “in”; “out” stack is empty)

What operations did we need to do to reach that condition (starting with an empty Queue)?
- $n$ queues ($n$ pushes)

Hence, what is the amortized cost?

\[
\frac{O(n)}{n} = O(1)
\]

So the average time for `dequeue()` is:

\[
O(1)
\]
Example #2: Using “Currency” Analogy

“Spend” $2 for every enqueue – $1 to the “computer”, $1 to the “bank”.

Example walk-through:
- enqueue A
- enqueue B
- enqueue C
- enqueue D
- enqueue E
- dequeue

Potential Function
Example #3: (Parody / Joke Example)

Lectures are 1 hour long, 3 times per week, so I’m supposed to lecture for 27 hours this quarter. If I end the first 26 lectures 5 minutes early, then I’d have “saved up” 130 minutes worth of extra lecture time. Then I could spend it all on the last lecture and can keep you here for 3 hours (bwahahahaha)! (After all, each lecture would still be 1 hour amortized time)
Wrapping up Amortized Analysis

• In what cases do we care more about the average / amortized run time?
  
  Few worst-cases?
  
  If we want to be fast in general but occasion worst-case is not too costly.

• In what cases do we care more about the worst-case run time?
  
  If n increases quickly?
  
  If worst-case is costly.
Taking a step back...

(Take a deep breath)
What have we covered so far?

• Abstract Data Types (ADTs)
• Two data structures
  • Stacks (both using arrays and linked-lists)
  • Queues (including circular queues)
• Asymptotic Analysis
  • Intuition for Big-O
  • Formally proving Big-O using Inductive Proofs
  • Calculating Big-O for recursive methods using Recurrence Relations
  • Big-O’s cousins: Big-Ω, Big-θ, little-o, little-ω
  • Average running time using Asymptotic Analysis
Whew!

That was a *lot* of algorithm analysis.

Now shifting gears completely...
on to some new data structures!
Dictionary ADT

key, value pair
word definition
Dictionary ADT

Meaning
- set of (key, value) pairs
- can compare keys

Operations
- insert (key, value)
- delete (key)
- update_value (key, new_value)
- find (key)
Uses of Dictionary ADT

Used to store information with some key and retrieve it efficiently – lots of programs do that!

Examples:

• Contacts in a phone (name: number, email)
• Orca/Husky cards (account number: balance)
• Genome maps (DNA sequence: location on genome)
• Lilian’s database of your grades (student ID: assignments, grades)
• Networks (router tables), Operating Systems (page tables), Compilers (symbol tables), Databases
• ... and so much more!

Possibly the most widely used ADT!
Motivating Hash Tables

Creative thinking time: how could you implement a dictionary using what you know so far (namely, linked-lists and arrays)?
e.g. map names (key) to phone numbers (value)
Motivating Hash Tables

Creative thinking time: how could you implement a dictionary using what you know so far (namely, linked-lists and arrays)?
e.g. map names (key) to phone numbers (value)
Motivating Hash Tables

Running times for Dictionary operations with $n$ (key, value) pairs:

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>linked-list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>sorted array</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>“Magic Array”</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
“Magic Array”

Use key to compute array index for an item in $O(1)$ time

Example: phone contacts (name, number)

$$\text{name} \rightarrow \text{index} = \text{computeIndex} (\text{name}) \rightarrow \text{array}[\text{index}] = (\text{name}, \text{number})$$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gregor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>765-4321</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123-4567</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
“Magic Array”

Use key to compute array index for an item in O(1) time

Example: phone contacts (name, number)

\[ \text{name} \rightarrow \text{index} = \text{computeIndex(name)} \rightarrow \text{array[index]} = (\text{name, number}) \]

What would be important about the indices from computeIndex?

Have different index for every key
Introducing... Hash Tables!

Closest thing to our “magic array”
Hash Tables: closest thing to our “Magic Array”

- **Average case** \(O(1)\) find, insert, and delete (when under some often-reasonable assumptions)
- **Our computeIndex function** is called a hash function
- The index from the hash function is called a hash value
  (also hash)

```
<table>
<thead>
<tr>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>tableSize - 1</td>
</tr>
</tbody>
</table>
```

All possible keys
Hash Tables: Example Illustration
Hash Functions

Hash functions need to...

- have uniformity (maps inputs as evenly as possible)
- be deterministic (always same hash for same key)
- $O(1)$

For a person’s name, would it be a good hash function to...

- Use the ASCII values of first and second letter? 
  - Joe, Joel, John...
- Use the number of letters in the name?
  
$\text{key} = \text{Cersei} \quad \text{h(Cersei)} = \text{same number every time}$
Example Hash Function

Hash function “djb2”:

```c
unsigned long
hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
}
```
Hash Functions

• Many datatypes and Objects are hashable

• When writing a class, can make it hashable!

  Do so by implementing `hashCode` method

• We’ll focus on `ints` and `Strings` in this class.
Collisions

Happens when two elements get the same index (unavoidable in practice)

\[ h(\text{Tyrion}) = 4 \quad h(\text{sansa}) = 4 \]

Homework: come up with a strategy, write it down on paper, and bring it to class on Weds
Hash Table roles

When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:

We will learn both roles, but most programmers “in the real world” spend more time as clients while understanding the library.
Hash Tables

• There are \( m \) possible keys (\( m \) typically large, even infinite)
• We expect our table to have only \( n \) items
• \( n \) is much less than \( m \) (often written \( n << m \))

Many dictionaries have this property

• Compiler: All possible identifiers allowed by the language vs. those used in some file of one program
• Database: All possible student names vs. students enrolled
• AI: All possible chess-board configurations vs. those considered by the current player
• ...
Hash Table Size

- How can we keep hash values (i.e. the indices) within the table size?

  ![Hash function diagram](image)

  \[ \text{hash function} \rightarrow \text{hash value} \rightarrow \text{index} \]

  \[ x \% y = \text{remainder of} \ \frac{x}{y} \]

  \[ 3 \% 10 \rightarrow \frac{3}{10} = 0 \text{ remainder} \]

  \[ \text{index} = \text{hash(key)} \% \text{table size} \]

  \[ h(\text{key 1}) = 3 \rightarrow \text{index} = 3 \]
  \[ h(\text{key 2}) = 17 \rightarrow \text{index} = 7 \]
  \[ h(\text{key 3}) = 27 \rightarrow \text{index} = 7 \]

  Collision!

- Table size usually prime
  - Real-life data tends to have a pattern
  - "Multiples of 61" probably less likely than "multiples of 60"
  - Helpful for a collision-handling strategy we'll see next week
Review: Hash Tables thus far...

- The hash table is one of the most important data structures – supports only **find**, **insert**, and **delete** efficiently.
  - Have to search entire table for other operations.
- Important to use a good hash function.
- Important to keep hash table at a good size.
- Side-comment: hash functions have uses beyond hash tables – examples: Cryptography, check-sums.
- Big remaining topic: Handling collision.
Homework

Come up with a collision-resolution strategy, write it down on paper, and bring it to class on Wednesday.

Goal: prime our brains for learning the most common collision-resolution strategies.