Amortized Analysis

Dynamic Tables
Aggregate Method
Accounting method ← section
Potential Method

Hash Tables

How large should a hash table be?

As we increase the size of the hash table, the search time decreases.
Direct-access table worst-case \( O(1) \) time.

- as large as possible (time)
- as small as possible (space)

\( \Theta(n) \) size for \( n \) items (making it larger doesn’t give large payoff)

Problem: What if we don’t know \( n \) in advance?
**Solution: Dynamic Tables**

Idea: Whenever the table gets too full, i.e., overflows, "grow" it.

1. Allocate (malloc or new) a larger table.
2. Move the items from the old table to new.
3. Free old table.

To keep things simple, consider a situation where all elements are hashed to different slots.

1. Insert
2. Insert
3. Insert
4. Insert
5. Insert
6. Insert

When overflow, create table of twice the size.
Analysis

Sequence of n insert operations
Worst case cost of 1 insert = \( \Theta(n) \)
Therefore, worst case cost of n inserts = n. \( \Theta(n) = \Theta(n^2) \)
But, not all of them can be worst case! (n inserts take \( \Theta(n) \) time)

Proper Analysis

Let \( c_i \) be the cost of \( i \)th insert
\[ c_i = \begin{cases} 1 & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases} \]

<table>
<thead>
<tr>
<th>i</th>
<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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size ( c_i )</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( c_i )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Cost of $n$ inserts

$$= \sum_{i=1}^{n} C_i \leq \sum_{j=0}^{\lfloor \log(n-1) \rfloor} 2^j$$

$$= n + \sum_{j=0}^{\lfloor \log(n-1) \rfloor} 2^j \leq 3n = \Theta(n)$$

Amortized cost of 1 insert $= \Theta(n)/n = \Theta(1)$

**Amortized Analysis**

Analyze a sequence of operations to show that average cost per operation is small, even though 1 op may be expensive.

No probability - average performance in the worst case.

Types of analysis:

- Aggregate (previous example)
- Accounting (will be covered in section)
- Potential

more precise, rigorous
Revisiting B-Trees.

Assume you've done an unsuccessful search, and found the leaf into which key will be inserted. (Will take $O(\log n)$ time.) Assume B-tree insert procedure of lecture where keys are not shifted to siblings.

How much does a `Insert` cost? $O(\log n)$ where $n$ is # nodes in worst case.

What about on average?

- **NotFull**
  - `Insert`: no split necessary

- **Full**
  - `Insert`: many splits, $O(\log n)$ in worst case.

Will use potential method to analyze.
Potential Method

Data structure has potential energy, which changes when data structure is modified.

Framework

Start with data structure \( D_0 \)

\( \text{Op}_i \) transforms \( D_{i-1} \rightarrow D_i \)

Cost of \( \text{Op}_i \) is \( C_i \)

Potential function: \( \phi: \{ D_i \} \rightarrow \mathbb{R} \)

\( \phi(D_0) = 0 \) and \( \phi(D_i) > 0 \) for \( i \)

Amortized cost \( \hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1}) \)

\( \Delta \phi_i \geq 0 \) store work

\( \Delta \phi_i < 0 \) deliver work

Total amortized cost of \( n \) ops:

\[
\sum_{i=1}^{n} \hat{C}_i = \sum_{i=1}^{n} (C_i + \phi(D_i) - \phi(D_{i-1})) \\
= \sum_{i=1}^{n} C_i + \phi(D_n) - \phi(D_0) \quad \text{(telescoping)} \\
> 0 \quad = 0
\]

Amortized costs bound actual costs.
Potential Method for B-Tree Insert

\[ \phi(\text{Tree}) = \phi_1 \cdot (\# \text{ nodes which are full}) \]

i.e., 3 keys for 2-3-4 or 2-4 trees

Clearly \( \phi(D_0) \geq 0 \). Assume \( \phi(D_0) = 0 \).

Consider several cases.

No split: actual cost = \( \phi_2 \) to do insertion

\[ \phi(D_i) - \phi(D_{i-1}) = \Delta \phi_i \leq \phi_1 \cdot (1) \]

since at most one node/leaf gets full if no splits.

Exactly 1 split: actual cost = \( \phi_3 \) to do split

\[ \Delta \phi_i \leq \phi_0 \cdot 0 \]

Split causes overflow node to become not full, but might cause parent to become full

Exactly k splits: actual cost = \( k \cdot \phi_3 \)

\[ \Delta \phi_i \leq -(k-1) \cdot \phi_1 \]

Causes k nodes to become not full, and at most one node to become full.

General formula

Valid for \( k = 0, 1, 2, \ldots \)
Amortized Costs

Amortized cost \( i \) = actual cost \( i \) + \( \Delta \phi_i \)
\[ \leq k_i \cdot C_3 - (k_{i-1}) C_1 \]
(assuming \( k_i = 0, 1, 2 \ldots \) splits in \( \phi_i \))
\[ \leq C_3 \quad \text{if we choose } C_4 > C_3 \]

0o. Amortized cost of an insert is constant!

(We know that the sum of \( n \) amortized costs of inserts is \( \geq \) sum of \( n \) actual costs of inserts, thanks to defn. of potentials)

What about Deletes?

Deletes cause underflows and require merges. In the worst case, for a delete merge, may require \( O(\log n) \) merges. Similar analysis can show constant amortized cost.

Combination of inserts/deletes is harder, since splits may cause nodes to be close to underflow, and alternating inserts & deletes could cause \( O(\log n) \) splits & merges. For certain special trees, can be constant amortized cost!