Amortized Analysis (CLRS chap. 17)

- Not just consider one operation, but a sequence of operations on a given data structure.
- Average cost over a sequence of operations.
- Probabilistic analysis:
  - Average case running time: average over all possible inputs for one algorithm (operation).
  - If using probability, called expected running time.
- Amortized analysis:
  - No involvement of probability
  - Average performance on a sequence of operations, even some operation is expensive.
  - Guarantee average performance of each operation among the sequence in worst case.

Three Methods of Amortized Analysis

- Aggregate analysis:
  - Total cost of \( n \) operations/\( n \),
- Accounting method:
  - Assign each type of operation an (different) amortized cost
  - Overcharge some operations,
  - Store the overcharge as credit on specific objects,
  - Then use the credit for compensation for some later operations.
- Potential method: (will not be discussed)
  - Same as accounting method
  - But store the credit as “potential energy” and as a whole.

Example for amortized analysis

- Stack operations:
  - \( \text{PUSH}(S,x) \), \( O(1) \)
  - \( \text{POP}(S) \), \( O(1) \)
  - \( \text{MULTIPOP}(S,k), \min(s,k) \)
    - \textbf{while} not \( \text{STACK-EMPTY}(S) \) and \( k > 0 \)
    - \textbf{do} \( \text{POP}(S) \)
    - \( k = k - 1 \)
  - Let us consider any sequence of \( n \) \text{PUSH}, \text{POP}, \text{MULTIPOP} (at any order).
  - The worst case cost for \text{MULTIPOP} in the sequence is \( O(n) \), since the stack size is at most \( n \).
  - Thus the cost of the sequence is \( O(n^2) \). Correct, but not tight.

Aggregate Analysis

- In fact, a sequence of \( n \) operations on an initially empty stack cost at most \( O(n) \). Why?

  Each object can be \text{POP} only once (including in \text{MULTIPOP}) for each time it is \text{PUSHed}. \#\text{POPs} is at most \#\text{PUSHs}, which is at most \( n \).

  Thus the average cost of an operation is \( O(n)/n = O(1) \).

  Amortized cost in aggregate analysis is defined to be average cost.
Another example: increasing a binary counter

- Binary counter of length $k$, $A[0..k-1]$ of bit array.
- **INCREMENT(A)**
  1. $i \leftarrow 0$
  2. while $i < k$ and $A[i] = 1$
  3. do $A[i] \leftarrow 0$ (flip, reset)
  4. $i \leftarrow i + 1$
  5. if $i < k$
  6. then $A[i] \leftarrow 1$ (flip, set)

Analysis of **INCREMENT(A)**

- Cursory analysis:
  - A single execution of **INCREMENT** takes $O(k)$ in the worst case (when A contains all 1s)
  - So a sequence of $n$ executions takes $O(nk)$ in worst case (suppose initial counter is 0).
  - This bound is correct, but not tight.
- The tight bound is $O(n)$ for $n$ executions.

Amortized (Aggregate) Analysis of **INCREMENT(A)**

Observation: The running time determined by #flips but not all bits flip each time **INCREMENT** is called.

<table>
<thead>
<tr>
<th>Counter value</th>
<th>Total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>31</td>
</tr>
</tbody>
</table>

- $A[0]$ flips every time, total $n$ times.
- $A[1]$ flips every other time, $\lceil n/2 \rceil$ times.
- $A[2]$ flips every forth time, $\lceil n/4 \rceil$ times.
- ....
- for $i=0,1,...,k-1$, $A[i]$ flips $\lceil n/2^i \rceil$ times.
- Thus total flips is $\sum_{i=0}^{k-1} \lceil n/2^i \rceil < n \sum_{i=0}^{\infty} 1/2^i = 2n$.

Amortized Analysis of **INCREMENT(A)**

- Thus the worst case running time is $O(n)$ for a sequence of $n$ **INCREMENTs**.
- So the amortized cost per operation is $O(1)$. 
Amortized Analysis: Accounting Method

• Idea:
  – Assign differing charges to different operations.
  – The amount of the charge is called amortized cost.
  – amortized cost is more or less than actual cost.
  – When amortized cost > actual cost, the difference is saved in specific objects as credits.
  – The credits can be used by later operations whose amortized cost < actual cost.

Accounting method: binary counter

• Let $1 represent each unit of cost (i.e., the flip of one bit).
• Charge an amortized cost of $2 to set a bit to 1.
• Whenever a bit is set, use $1 to pay the actual cost, and store another $1 on the bit as credit.
• When a bit is reset, the stored $1 pays the cost.
• At any point, a 1 in the counter stores $1, the number of 1’s is never negative, so is the total credits.
• At most one bit is set in each operation, so the amortized cost of an operation is at most $2.
• Thus, total amortized cost of $n$ operations is $O(n)$, and average is $O(1)$.

Amortized analyses: dynamic table

• A nice use of amortized analysis
• Table-insertion, table-deletion.
• Scenario:
  – A table —maybe a hash table
  – Do not know how large in advance
  – May expend with insertion
  – May contract with deletion
  – Detailed implementation is not important
• Goal:
  – $O(1)$ amortized cost.
  – Unused space always $\leq$ constant fraction of allocated space.

Dynamic table: expansion with insertion

• Table expansion
• Consider only insertion.
• When the table becomes full, double its size and reinsert all existing items.
• (comment: If this is used for hash tables, we usually perform this operation already when the table is 50% full)
• Each time we actually insert an item into the table, it’s an elementary insertion.
Aggregate analysis

- **Running time:** Charge 1 per elementary insertion. Count only elementary insertions, since all other costs together are constant per call.
  - \( c_i \) = actual cost of \( i \)th operation
    - If not full, \( c_i = 1 \)
    - If full, have \( i - 1 \) items in the table at the start of the \( i \)th operation. Have to copy all \( i - 1 \) existing items, then insert \( i \)th item, ⇒
  - **Cursory analysis:** \( n \) operations ⇒ \( n \cdot O(n) \) time for \( n \) operations.
  - Of course, we don’t always expand: If the last operation copied \( n \) keys, then the previous copying took half this time, hence total number of copy operations is
    \[
    n \left( \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots \right) = 2n
    \]
  - Therefore, aggregate analysis says amortized cost per operation = 3.

Accounting analysis

- Charge $3 per insertion of \( x \).
  - $1 pays for \( x \)’s insertion.
  - $1 pays for \( x \) to be moved in the future.
  - $1 pays for some other item to be moved. (the newest keys transfer credit to the oldest keys in the array)
- Suppose we’ve just expanded, size = \( m \) before next expansion, size = \( 2m \) after next expansion.
- Assume that the expansion used up all the credit, so that there’s no credit stored after the expansion.
- Will expand again after another \( m \) insertions.
- Each insertion will put $1 on one of the \( m \) items that were in the table just after expansion and will put $1 on the item inserted.
- Have $2m of credit by next expansion, when there are \( 2m \) items to move. Just enough to pay for the expansion, with no credit left over!

Expansion and contraction

- Define \( \alpha = \frac{n}{m} \), where \( n \) is number of keys and \( m \) is the array size. This is called the load factor.
- After expansion, \( \alpha = 0.5 \) (actually very slightly more than 0.5)
- When \( \alpha \) drops too low, contract the table.
  - Allocate a new, smaller one.
  - Copy all items.
- Bad idea: contract when \( \alpha \) becomes smaller than 0.5.
- Still want
  - \( \alpha \) bounded from below by a constant, for example 0.25.
  - (at the homework - show that any constant <0.5 will do)
  - amortized cost per either insertion or deletion = \( O(1) \). That is, any sequence of \( n \) insertions or deletions (in any order) takes \( O(n) \) time.
- Measure cost in terms of elementary insertions and deletions. (hint - a deletion could add a $1)

Splay tree

- A binary search tree (not balanced)
- Height may be larger than \( \log n \), even \( n-1 \).
- However a sequence of \( n \) operations takes \( O(n \log n) \).
- Let \( S \) be the set of keys.
- Operations:
  - Find(\( x,S \)) - check if key \( x \) is in \( S \)
  - Insert(\( x,S \))
  - Delete(\( x,S \))
  - Merge(\( S,S’ \)) Merge two trees, under the assumption that each key in the first is smaller than each key in the second.
  - Split(\( x,S \))
  - All based on
    - **Splay(\( x,S \)).** After finding \( x \), (or succ(\( x \)) if \( x \notin S \)) reorganize tree via a sequence of rotations, so that \( x \) moves to be root.
    - Disclaimer: if \( x \notin S \), then the new root is either \( \max(k \in S \mid k < x) \) or \( \min(k \in S \mid k > x) \)
Splay tree (cont.)

• For examples,
  – *insert(x,S)* - insert like in a regular search tree (so x is add as a new leaf) and perform Splay(x)
  – *merge(S,S’)*
    • Call Splay(x, S) and then make S’ the right child of the root of the tree of S.
  – *Delete(x,S)*, call Splay(x,S), remove x, then merge(left(x), right(x)).
  – Similar for others.
  – Constant number of splays called.

Splay tree (cont.)

• Splay operation is based on basic rotate(x) operation (either left or right).
• Three cases:
  – p is the parent of x and x has not grandparent
    rotate(x)
  – x is the left (or right) child of y and y is the left (or right) child of g,
    rotate(y) and then rotate(x)
  – x is the left (or right) child of y and y is the right (or left) child of z,
    rotate(x) and then rotate(x)

Splay tree (cont.)

Credit invariant

* ni - is number of nodes in the subtree rooted at x.
  Let r(x) the rank of x, is \( \lfloor \log_2 n \rfloor \)

Credit invariant: Each node x always has \( r(x) \) credits.

– recall that the actually time per splay operation could be much larger than \( \log n \).

• Lemma 1: (investment lemma). Each splay(x,S) operation requires the addition of \( \leq \log_2 n + 1 \) credits to perform the operation and maintain the credit invariant.

• Split, Merge, Insert and Delete will require adding credit of \( \leq \log_2 n \) credits.

• The cost of a single step of the splay process. In this step we rearrange \( \leq 3 \) nodes - (one or two rotation of x, parent(x) and possibly g=grand parent(x) ). Assume that the Credit Invariant holds before the rotations. Let \( r'(x) \) denotes r(x) after the rotation.

• Lemma 2: (without proof) It requires the investment of \( \leq 3(r'(x) - r(x)) \) credits, and possibly moving credits between x, p, g. So the Credit Invariant holds after this step. Here \( r'(x) \) is r(x) after the (one or two) rotation(s).

• We had do the investment after each rotation, it would be hard to bound the total investment per splay. Instead, lets see ho much we need to invest then the splay is done.

• Proof of Lemma 1. Assume the splay operation took k steps (each is a rotation or double rotation). Let \( r^{(i)}(x) \) denote r(x) after i steps. So \( r^{(i)}(x) = r(\text{root}) = \lfloor \log_2 n \rfloor \). Note also that \( r^{(i)}(x) = r(x) \)

  \[
  3(r^{(i)}(x) - r^{(i-1)}(x)) + \ldots + 3(r^{(3)}(x) - r^{(2)}(x)) + 3(r^{(2)}(x) - r^{(1)}(x)) + 3(r^{(1)}(x) - r(x)) + 1 = 3(\lfloor \log_2 n \rfloor - r(x)) + 1 \leq 3(\log_2 n) + 1
  \]