Amortized analysis
In amortized analysis the goal is to bound the worst case time of a sequence of operations on a data-structure.

If $n$ operations take $T(n)$ time (worst case), the amortized cost of an operation is $T(n)/n$.

In the aggregate analysis method, we directly bound $T(n)$. 
A stack with Multipop operation supports the standard stack operations (Push, Pop, IsEmpty) and the a Multipop operation:

**MULTIPOP**(S, k)

1. while not **EMPTY**(S) and k > 0
2. **POP**(S)
3. k ← k − 1

Given n operations (starting with an empty stack):
- There are at most n push operations.
- Each element is popped from the stack at most once for each time we pushed it into the stack.
- Therefore, there are at most n pop operations, including those performed during Multipop.
- Total cost is Θ(n). Amortized cost is Θ(n)/n = Θ(1).
Binary counter

- Suppose we store a counter in an array $A[0..k-1]$ of bits, where $A[0]$ is the least significant bit.
- We implement operation $\text{INCREMENT}(A)$ which increases the value of the counter by 1, as follows.

$\text{INCREMENT}(A)$

1. $i \leftarrow 0$
2. while $i < A.\text{length}$ and $A[i] = 1$
3. $A[i] \leftarrow 0$
4. $i \leftarrow i + 1$
5. if $i < A.\text{length}$
6. $A[i] \leftarrow 1$

- To bound the time complexity of $n$ increment operations (starting with $A$ containing only zeros), it suffices to bound the number of bit flips in these operations.
### Binary counter

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0</td>
<td>0 1</td>
<td>n</td>
</tr>
<tr>
<td>1 0 0 0 0 1</td>
<td>1 1/2</td>
<td>⌊n/2⌋</td>
</tr>
<tr>
<td>2 0 0 0 1 0</td>
<td>2 1/4</td>
<td>⌊n/4⌋</td>
</tr>
<tr>
<td>3 0 0 1 1 1</td>
<td>3 1/8</td>
<td>⌊n/8⌋</td>
</tr>
<tr>
<td>4 0 0 1 0 0</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5 0 0 1 0 1</td>
<td>i 1/2^i</td>
<td>⌊n/2^i⌋</td>
</tr>
<tr>
<td>6 0 0 1 1 0</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>7 0 0 1 1 1</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8 0 1 0 0 0 0</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The total number of flips is

\[
\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < \sum_{i=0}^{\infty} n/2^i = n \sum_{i=0}^{\infty} 1/2^i = 2n.
\]

Total cost of \( n \) increments is \( \Theta(n) \). Amortized cost \( \Theta(1) \).
Accounting method

- Each operation has a **cost** which is its actual time complexity.
- When an operation is performed, we **charge** a certain amount, which may be different than the cost.
- Part of a charge can be stored in objects of the data-structures.
- To pay the cost of an operation, we can use both the charge or stored credit.
- The total cost of the operation is at most the sum of the charges.
For stack with Multipop, the costs and charges of the operations are as follows:

<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
<th>charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multipop</td>
<td>$s$</td>
<td>0</td>
</tr>
</tbody>
</table>

$s$ is number of elements popped

- For a push operation, we use $1$ to pay for the cost, and the remaining $1$ is stored in the pushed element.
- The cost of Pop/Multipop is paid using the dollars stored at the popped elements.
- The total cost of $n$ operations is at most $2n$.
- The amortized time complexity of an operation is $\Theta(1)$.
For stack with Multipop, the costs and charges of the operations are as follows:

<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
<th>charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multipop</td>
<td>$s$</td>
<td>0</td>
</tr>
</tbody>
</table>

$s$ is number of elements popped

push(S,4)

- For a push operation, we use $1 to pay for the cost, and the remaining $1 is stored in the pushed element.
- The cost of Pop/Multipop is paid using the dollars stored at the popped elements.
- The total cost of $n$ operations is at most $2n$.
- The amortized time complexity of an operation is $\Theta(1)$. 

Amortized analysis
For stack with Multipop, the costs and charges of the operations are as follows:

<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
<th>charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multipop</td>
<td>s</td>
<td>0</td>
</tr>
</tbody>
</table>

$s$ is number of elements popped

- For a push operation, we use $1$ to pay for the cost, and the remaining $1$ is stored in the pushed element.
- The cost of Pop/Multipop is payed using the dollars stored at the popped elements.
- The total cost of $n$ operations is at most $2n$.
- The amortized time complexity of an operation is $\Theta(1)$.
## Stack with Multipop operation

For stack with Multipop, the costs and charges of the operations are as follows:

<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
<th>charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multipop</td>
<td>$s$</td>
<td>0</td>
</tr>
</tbody>
</table>

$s$ is number of elements popped

- For a push operation, we use $1 to pay for the cost, and the remaining $1 is stored in the pushed element.
- The cost of Pop/Multipop is payed using the dollars stored at the popped elements.
- The total cost of $n$ operations is at most $2n$.
- The amortized time complexity of an operation is $\Theta(1)$.

**Amortized analysis**
Stack with Multipop operation

For stack with Multipop, the costs and charges of the operations are as follows:

<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
<th>charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multipop</td>
<td>$s$</td>
<td>0</td>
</tr>
</tbody>
</table>

$s$ is number of elements popped

- For a push operation, we use $1$ to pay for the cost, and the remaining $1$ is stored in the pushed element.
- The cost of Pop/Multipop is payed using the dollars stored at the popped elements.
- The total cost of $n$ operations is at most $2n$.
- The amortized time complexity of an operation is $\Theta(1)$.
For stack with Multipop, the costs and charges of the operations are as follows:

<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
<th>charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multipop</td>
<td>s</td>
<td>0</td>
</tr>
</tbody>
</table>

$s$ is number of elements popped

- For a push operation, we use $1$ to pay for the cost, and the remaining $1$ is stored in the pushed element.
- The cost of Pop/Multipop is payed using the dollars stored at the popped elements.
- The total cost of $n$ operations is at most $2n$.
- The amortized time complexity of an operation is $\Theta(1)$.
The cost of an \texttt{INCREMENT}(A) operation is the number of bit flips.

The charge of an \texttt{INCREMENT}(A) operation is $2.

We keep the following invariant: each 1 in $A$ stores $1$.

To pay the cost of \texttt{INCREMENT}(A), we use the dollars stored at the 1’s that were changed to 0’s, and $1$ from the charge (to pay for the 0 that was changed to 1).

The remaining $1$ of the charge is stored at the 0 that was changed to 1.

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
The cost of an $\text{INCREMENT}(A)$ operation is the number of bit flips.

The charge of an $\text{INCREMENT}(A)$ operation is $\$2$.

We keep the following invariant: each 1 in $A$ stores $\$1$.

To pay the cost of $\text{INCREMENT}(A)$, we use the dollars stored at the 1’s that were changed to 0’s, and $\$1$ from the charge (to pay for the 0 that was changed to 1).

The remaining $\$1$ of the charge is stored at the 0 that was changed to 1.

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 1 \\
\text{\$1}
\end{array}
\]
The cost of an \texttt{INCREMENT}(A) operation is the number of bit flips.

The charge of an \texttt{INCREMENT}(A) operation is $2.

We keep the following invariant: each 1 in $A$ stores $1$.

To pay the cost of \texttt{INCREMENT}(A), we use the dollars stored at the 1’s that were changed to 0’s, and $1$ from the charge (to pay for the 0 that was changed to 1).

The remaining $1$ of the charge is stored at the 0 that was changed to 1.

\begin{center}
\begin{tabular}{ccccccc}
0 & 0 & 0 & 0 & \textbf{1} & \textbf{0} \\
\textbf{$1$} \\
\end{tabular}
\end{center}
The cost of an \texttt{INCREMENT}(A) operation is the number of bit flips.

The charge of an \texttt{INCREMENT}(A) operation is $2.

We keep the following invariant: each 1 in \( A \) stores $1.

To pay the cost of \texttt{INCREMENT}(A), we use the dollars stored at the 1’s that were changed to 0’s, and $1 from the charge (to pay for the 0 that was changed to 1).

The remaining $1 of the charge is stored at the 0 that was changed to 1.
The cost of an \textsc{Increment}(A) operation is the number of bit flips.

The charge of an \textsc{Increment}(A) operation is $2.

We keep the following invariant: each 1 in $A$ stores $1$.

To pay the cost of \textsc{Increment}(A), we use the dollars stored at the 1's that were changed to 0's, and $1$ from the charge (to pay for the 0 that was changed to 1).

The remaining $1$ of the charge is stored at the 0 that was changed to 1.
Suppose that we use a hash table with chaining.

Let $\alpha = n/m$, where $n$ is the number of elements in the table and $m$ is the size of the table.

We want to maintain the invariant $1/4 \leq \alpha \leq 1$:

- $\alpha \leq 1$ gives $\Theta(1)$ expected search time.
- $\alpha \geq 1/4$ gives $\Theta(n)$ space.

Assume we only have insertions (no deletions).

We can maintain the invariant by performing rehashing when the $m + 1$-th element is inserted to the table: We create a new table of size $2m$ and move all elements to the new table.
The cost of Insert is 1 for inserting the element to the table, and $n$ for performing rehashing.

The charge of Insert is $3.

When we insert an element to the table, we use $1 from the charge to pay for the cost, and we store the remaining $2 in the table.

When rehashing is performed, the cost is paid using dollars stored in the table.

Immediately after rehashing, no dollars are stored in the table.

The next rehashing occurs after $m/2$ insertions, so the table stores $m$ dollars at the time of the next rehashing.
Example

Amortized analysis
Example

T $2 \quad \text{insert}(T,3)$

\[ \begin{array}{c}
\quad \\
3 \\
6 \\
\end{array} \]
Example

$T \quad 0 \quad \text{insert}(T, 4)$

\[
\begin{array}{c}
  & 6 \\
  & 3 \\
\end{array}
\]
Example

T $2 \quad \text{insert}(T,4)$

```
6
4 3
```

Amortized analysis
Example

T $4$ \text{insert}(T, 2)

Amortized analysis
Example

$0$

$\text{insert}(T, 7)$

Amortized analysis
Example

$T \quad $2 \quad \text{insert}(T,7)$

```
6
3
4
2
7
```

Amortized analysis
BB[$\alpha$] tree — definition

- Let $size(v)$ be the number of vertices in the subtree of $v$.
- A vertex $v$ is of bounded balance $\alpha$ if
  
  $$size(v.left) \geq \lfloor \alpha \cdot size(v) \rfloor \quad \text{and} \quad size(v.right) \geq \lfloor \alpha \cdot size(v) \rfloor$$

- A BB[$\alpha$] tree ($\alpha < 0.5$) is a binary search tree such that every vertex $v$ is of bounded balance $\alpha$.
- The height of a BB[$\alpha$] tree with $n$ vertices is at most $\log_{1/(1-\alpha)} n$.

\[ \alpha = \frac{1}{3} \]
To insert an element to a BB[$\alpha$] tree, insert it as a new leaf. Let $w$ be the highest vertex that is not of bounded balance $\alpha$. If $w$ exists, replace the subtree of $w$ by a balanced tree containing the same elements.

The time complexity is $O(\text{depth}(w) + \text{size}(w))$, which is $\Theta(n)$ in the worst case.

\[ \alpha = \frac{1}{3} \]
To insert an element to a BB[\(\alpha\)] tree, insert it as a new leaf. Let \(w\) be the highest vertex that is not of bounded balance \(\alpha\). If \(w\) exists, replace the subtree of \(w\) by a balanced tree containing the same elements.

The time complexity is \(O(\text{depth}(w) + \text{size}(w))\), which is \(\Theta(n)\) in the worst case.

\[ \alpha = \frac{1}{3} \]
To insert an element to a BB[α] tree, insert it as a new leaf. Let $w$ be the highest vertex that is not of bounded balance $\alpha$. If $w$ exists, replace the subtree of $w$ by a balanced tree containing the same elements.

The time complexity is $O(\text{depth}(w) + \text{size}(w))$, which is $\Theta(n)$ in the worst case.

$\alpha = 1/3$
Amortized complexity

- The cost of insert is $\text{depth}(w) + \text{size}(w)$, where $w$ is the vertex whose subtree was replaced (or the new leaf).
- The charge of insert is
  \[
  \left(1 + \frac{1}{1 - 2\alpha}\right) \log_{1/(1-\alpha)} n + \frac{2}{1 - 2\alpha}.
  \]
- We use $\leq \log_{1/(1-\alpha)} n$ dollars from the charge to pay for the $\text{depth}(w)$ cost.
- In each node on the path from the root to the new leaf, we put $1/(1 - 2\alpha)$ dollars from the charge. This requires $\leq \frac{1}{1-2\alpha} \log_{1/(1-\alpha)} n$ dollars.
- We use the dollars stored in $w$ and $2/(1 - 2\alpha)$ dollars from the charge to pay for the $\text{size}(w)$ cost.
- We need to show that $w$ contains at least $\text{size}(w) - \frac{2}{1-2\alpha}$ dollars.
After the insertion, we add $3 to the nodes on the path from the root to the new leaf.

\[ \alpha = \frac{1}{3} \]
Example

After the insertion, we add $3 to the nodes on the path from the root to the new leaf.

\[ \alpha = \frac{1}{3} \]
Example

To pay for the balancing of \( w \), we use the dollars stored in \( w \).

\[
\alpha = \frac{1}{3}
\]

Amortized analysis
Consider the tree before the balancing of $w$.

Let $M = \text{size}(w)$.

Without loss of generality, $\text{size}(w.\text{left}) \geq \text{size}(w.\text{right})$.

Since $w$ is not of bounded balance $\alpha$

\[
\text{size}(w.\text{right}) = \lfloor \alpha M \rfloor - 1
\]

\[
\text{size}(w.\text{left}) = M - \text{size}(w.\text{right}) - 1 = M - \lfloor \alpha M \rfloor
\]

Consider the insert operations of elements to the subtree of $w$ since the last balancing operation involving $w$.

The least number of such insertions is when all elements were inserted to $w.\text{left}$.

In that case, after last balancing operation involving $w$,

\[
\text{size}(w.\text{right}) = \lfloor \alpha M \rfloor - 1 \text{ and } \text{size}(w.\text{left}) \leq \lfloor \alpha M \rfloor.
\]

Therefore, the number of insert operations is at least

\[
(M - \lfloor \alpha M \rfloor) - \lfloor \alpha M \rfloor \geq M(1 - 2\alpha) - 2.
\]
Each insert operation of an element to the subtree of $w$ adds $1/(1 - 2\alpha)$ dollars to $w$.

Therefore, $w$ contains at least

$$\frac{1}{1 - 2\alpha} (M(1 - 2\alpha) - 2) = M - \frac{2}{1 - 2\alpha}$$

dollars.