**Problem: Stack with Backup**

- New operation `Backup` occurs after every `k` operations, and writes a copy of the stack.
- Assume stack height is always $\leq k$.

- Use same accounting method analysis as for Stack with Multipop, but just add 1 more unit of amortized time to all operations other than `Backup`.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Real-time</th>
<th>Amortized time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1</td>
<td>2 + 1</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>0 + 1</td>
</tr>
<tr>
<td>Multipop</td>
<td>$\min(\delta, n)$</td>
<td>0 + 1</td>
</tr>
<tr>
<td>Backup</td>
<td>$n \leq k$</td>
<td>0</td>
</tr>
</tbody>
</table>

- Since the stack accumulates $\leq$ extra units of credit before every `Backup` operation, we can always pay for `Backup` with stored credit.
Exercise

Simulating a queue with two stacks

To simulate the operations Create, Put, and Get on a queue Q, we use two stacks, Front[Q] and Rear[Q], as follows:

```
Queue Q

Front[Q]   Rear[Q]
0 ... 0

first    last

element  element
```

Our implementation is below.

```plaintext
function Create() begin
    Q := Memory()
    Front[Q] := Stack()
    Rear[Q] := Stack()
    return Q
end

procedure Put(x, Q) begin
    Push(x, Rear[Q])
end

function Get(Q) begin
    if Empty(Front[Q]) then
        while not Empty(Rear[Q]) do
            Push(Pop(Rear[Q]), Front[Q])
        end
    return Pop(Front[Q])
end
```
Exercise cont'd

For the analysis, let us measure the actual time by the number of Pushes and Pops, and take as our potential function

$$W(Q) = 2 \cdot \text{Size}(\text{Rear}[Q]).$$

Then

$$a_{\text{Put}} := t_{\text{Put}} + \Delta W_{\text{Put}}$$

$$= 1 + 2$$

$$= O(1).$$

Suppose $x$ elements are moved from Rear $[Q]$ to Front $[Q]$ by a Get. Then

$$a_{\text{Get}} := t_{\text{Get}} + \Delta W_{\text{Get}}$$

$$= (1 + 2x) - 2x$$

$$= O(1).$$

Thus we can simulate a queue with two stacks in $O(1)$ amortized time.
Exercise

Adding an operation

\[
\text{MultiPush}(S, A, i)
\]

which is equivalent to

\[
\begin{align*}
\text{Push}(S, A(i)) \\
\text{Push}(S, A(i+1)) \\
\vdots \\
\text{Push}(S, A(C))
\end{align*}
\]

would spoil the \(O(1)\) amortized time bound on stack operations:

\[
\begin{align*}
\text{MultiPush}(S, A, m) & \quad \text{1} \\
\text{MultiPop}(S, m) & \\
\text{MultiPush}(S, A, m) & \quad \text{2} \\
\text{MultiPop}(S, m) & \\
\vdots & \\
\text{MultiPush}(S, A, m) & \quad m \\
\text{MultiPop}(S, m) &
\end{align*}
\]

\(\text{takes } \Theta(m^2) = \omega(m) \text{ time.} \quad \text{(Note that the time for } m \text{ operations must be } \omega(m) \text{ for a counterexample.)} \)
Exercise

Implicit heap with Insert and ExtractMin

In an implicit binary heap with \( n \) elements, Insert and ExtractMin both take \( O(\log n) \) time. We show that we can view Insert as taking \( O(\log n) \) amortized time and ExtractMin as taking \( O(1) \) amortized time.

To measure the actual time, we count the number of comparisons performed by these operations. Consider the following potential function for a heap \( H \):

\[
\Phi(H) := 2 \sum_{\text{nodes} v \in H} \text{depth}(v).
\]

An Insert \((k, H)\) places the new element \( k \) at \( A[n+1] \), increments \( n \), and bubbles \( A[n+1] \) up to the root as needed. Let the height of the node that \( k \) ends up at be \( h \). Then the number of comparisons performed is \( h+1 \). So the amortized time is

\[
a_{\text{Insert}} := t_{\text{Insert}} + \Delta \Phi_{\text{Insert}} \\
= h+1 + 2 \lfloor \log n \rfloor \\
\leq 3 \lfloor \log n \rfloor + 1 \\
= O(\log n).
\]

An ExtractMin \((H)\) exchanges \( A[n] \) with \( A[1] \), decrements \( n \), and bubbles \( A[1] \) down with a call to Heapify. Let the depth of the node that \( A[1] \) ends up at be \( d \). Then the number of comparisons
performed is at most $2(d+1)$. Thus the amortized time for an ExtractMin is

$$a_{\text{ExtractMin}} := t_{\text{ExtractMin}} + \Delta \bar{\tau}_{\text{ExtractMin}}$$

$$\leq 2(d+1) - 2 \lfloor \log n \rfloor$$

$$\leq 2 \lfloor \log n \rfloor + 2 - 2 \lfloor \log n \rfloor \quad (*)$$

$$= 2$$

$$= O(1).$$

Thus we can view Insert and ExtractMin as taking $O(\log n)$ and $O(1)$ amortized time, respectively.

(Notice why we chose the factor of 2 in the definition of $\bar{\tau}$: to obtain the needed cancellation in $(*)$. )
Problem (Deleting the larger half)

Implement the following operations on a set $S$ of numbers,

- $\text{Insert}(x, S)$: Add $x$ to $S$.
- $\text{Delete Larger Half}(S)$: Delete the largest $\left\lceil \frac{|S|+1/2}{2} \right\rceil$ elements from $S$.

So both operations take $O(1)$ amortized time.

Solution sketch

We implement these operations as follows:

- $\text{Insert}(x, S)$: Put $x$ onto a singly-linked, unordered list $L$.
- $\text{Delete Larger Half}(S)$: Compute the median element $x$ of $S$.

Every element $y$ in $S$, with $y \geq x$, delete from $L$.

For our amortized analysis we use the charging method:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Actual Time</th>
<th>Amortized Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$1$</td>
<td>$5$</td>
</tr>
<tr>
<td>Delete Larger Half</td>
<td>$2n$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

We take the actual time for $\text{Delete Larger Half}$, which (i) computes the median and then (ii) does a scan to delete elements, to be $2n$.

An insert takes 1 unit of actual time, but receives 5 units of amortized time; we store the 4 units of credit on the inserted element.
Solution contd:

For Delete Larger Half, let us assume every element has 4 units of credit on it before the operation.

To execute Delete Larger Half, we use 2 units from every element. (So now every element has 2 units remaining on it.)

Take the remaining 2 units from every deleted element and place those units on the elements not deleted.

Now every element left in S has 4 units of credit again (as there are at least as many elements deleted as not deleted). So the credit assumption is maintained.