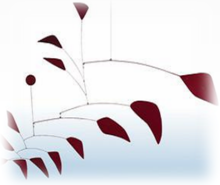
EEEC 4101-5101
Advanced Data Structures



Shahin Kamali

Topic 1b - Amortized Analysis

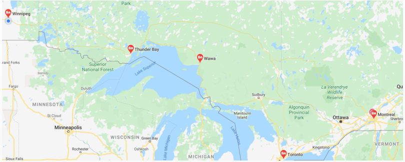
CLRS 17-1, 17-2, 17-3, 17-4

York University

Picture is from the cover of the textbook CLRS.

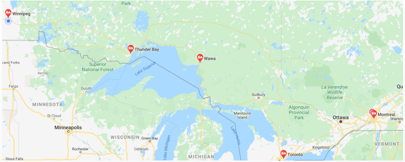
Amortized vs Average Analysis

- Assume you want to drive from Winnipeg to Montreal (distance is $\approx 2600km$).



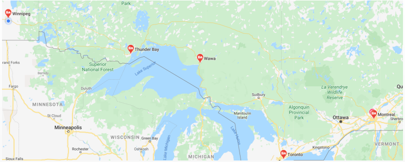
Amortized vs Average Analysis

- Assume you want to drive from Winnipeg to Montreal (distance is $\approx 2600km$).
- Solution 1: every day, choose a random number x uniformly from the range $[100,500]$ and drive x kilometre that day.



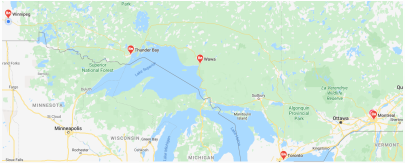
Amortized vs Average Analysis

- Assume you want to drive from Winnipeg to Montreal (distance is $\approx 2600km$).
- Solution 1: every day, choose a random number x uniformly from the range [100,500] and drive x kilometre that day.
  - You drive 300 kilometers per day **on average**



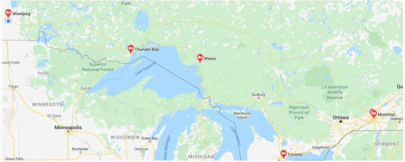
Amortized vs Average Analysis

- Assume you want to drive from Winnipeg to Montreal (distance is $\approx 2600km$).
- Solution 1: every day, choose a random number x uniformly from the range [100,500] and drive x kilometre that day.
  - You drive 300 kilometers per day **on average**
- Solution 2: drive city by city:



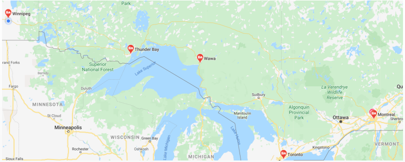
Amortized vs Average Analysis

- Assume you want to drive from Winnipeg to Montreal (distance is $\approx 2600km$).
- Solution 1: every day, choose a random number x uniformly from the range [100,500] and drive x kilometre that day.
  - You drive 300 kilometers per day **on average**
- Solution 2: drive city by city:
  - Day 1: Winnipeg to ThunderBay(700km) Day 2: Thunder Bay to Wawa (500km)
  - Day 3: Wawa to Toronto (900km) Day 4: Toronto to Montreal (500km)



Amortized vs Average Analysis

- Assume you want to drive from Winnipeg to Montreal (distance is $\approx 2600km$).
- Solution 1: every day, choose a random number x uniformly from the range [100,500] and drive x kilometre that day.
  - You drive 300 kilometers per day **on average**
- Solution 2: drive city by city:
  - Day 1: Winnipeg to ThunderBay(700km) Day 2: Thunder Bay to Wawa (500km)
  - Day 3: Wawa to Toronto (900km) Day 4: Toronto to Montreal (500km)
- On average, you drive $2600/4 = 650$km per day. We say **amortized** distance moved every day is 650km.



Amortized vs Average Analysis

- Both are concerned with the cost averaged over a sequence of **operations**.

Amortized vs Average Analysis

- Both are concerned with the cost averaged over a sequence of **operations**.
- Average case analysis relies on probabilistic assumptions about the input or the data structure
  - There is an underlying probability distribution.
  - The worst-case might be met with some small chance (you can be 'lucky' or not).

Amortized vs Average Analysis

- Both are concerned with the cost averaged over a sequence of **operations**.
- Average case analysis relies on probabilistic assumptions about the input or the data structure
  - There is an underlying probability distribution.
  - The worst-case might be met with some small chance (you can be 'lucky' or not).
- Amortized analysis consider consider a **sequence** of consecutive operations.
  - Bound the **total cost** for m operations
  - This gives the amortized cost $B(n)$ **per operation**
  - The amortized cost is only a function of n, the size of stored data
  - Unlike average case analysis, there is no probability distribution
  - Every sequence of m operations is guaranteed to have worst-case time at most $mB(n)$, regardless of the input or the sequence of operations (regardless of how lucky you are).

Amortized vs Average Analysis

- Let's compare two algorithms A and B
- A performs operations which take $\Theta(n)$ time in the worst case and $\Theta(\log n)$ **on average**.
- B performs operations which take $\Theta(n)$ time in the worst case and **amortized** $\Theta(\log n)$.

	worst-case time per operation	average/amortized time per operation	worst-case time for m operations	average time for m operations
Algorithm A	$\Theta(n)$	$\Theta(\log n)$ average		
Algorithm B	$\Theta(n)$	$\Theta(\log n)$ amortized		

Amortized vs Average Analysis

- Let's compare two algorithms A and B
- A performs operations which take $\Theta(n)$ time in the worst case and $\Theta(\log n)$ **on average**.
- B performs operations which take $\Theta(n)$ time in the worst case and **amortized** $\Theta(\log n)$.

	worst-case time per operation	average/amortized time per operation	worst-case time for m operations	average time for m operations
Algorithm A	$\Theta(n)$	$\Theta(\log n)$ average	$\Theta(m \cdot n)$	
Algorithm B	$\Theta(n)$	$\Theta(\log n)$ amortized		

Amortized vs Average Analysis

- Let's compare two algorithms A and B
- A performs operations which take $\Theta(n)$ time in the worst case and $\Theta(\log n)$ **on average**.
- B performs operations which take $\Theta(n)$ time in the worst case and **amortized** $\Theta(\log n)$.

	worst-case time per operation	average/amortized time per operation	worst-case time for m operations	average time for m operations
Algorithm A	$\Theta(n)$	$\Theta(\log n)$ average	$\Theta(m \cdot n)$	
Algorithm B	$\Theta(n)$	$\Theta(\log n)$ amortized	$\Theta(m \log n)$	

Amortized vs Average Analysis

- Let's compare two algorithms A and B
- A performs operations which take $\Theta(n)$ time in the worst case and $\Theta(\log n)$ **on average**.
- B performs operations which take $\Theta(n)$ time in the worst case and **amortized** $\Theta(\log n)$.

	worst-case time per operation	average/amortized time per operation	worst-case time for m operations	average time for m operations
Algorithm A	$\Theta(n)$	$\Theta(\log n)$ average	$\Theta(m \cdot n)$	$\Theta(m \log n)$
Algorithm B	$\Theta(n)$	$\Theta(\log n)$ amortized	$\Theta(m \log n)$	

Amortized vs Average Analysis

- Let's compare two algorithms A and B
- A performs operations which take $\Theta(n)$ time in the worst case and $\Theta(\log n)$ **on average**.
- B performs operations which take $\Theta(n)$ time in the worst case and **amortized** $\Theta(\log n)$.

	worst-case time per operation	average/amortized time per operation	worst-case time for m operations	average time for m operations
Algorithm A	$\Theta(n)$	$\Theta(\log n)$ average	$\Theta(m \cdot n)$	$\Theta(m \log n)$
Algorithm B	$\Theta(n)$	$\Theta(\log n)$ amortized	$\Theta(m \log n)$	$\Theta(m \log n)$

Bit Counter

- Start from an initial configuration where all bits are ‘0’
- Each operation increments the encoded number
- We want to know how many bits are flipped per operation

Bit Counter

- Start from an initial configuration where all bits are '0'
- Each operation increments the encoded number
- We want to know how many bits are flipped per operation
- The i'th bit from right is flipped iff all $i-1$ bits on its right are 1 before the increment ($i \geq 0$)
    - After the flip all bits on the right will be 0.
    - In the next $2^i - 1$ operations after the flip the bit is not flipped.
    - The i'th bit is flipped once in 2^i operations

		$\lceil \log m \rceil$	...		2	1	0		
0	0	0	0	0	0	0	0	initial configuration	
0	0	0	0	0	0	0	1	after 1st increment	1 bit flipped
0	0	0	0	0	0	1	0	after 2nd increment	2 bits flipped
			⋮					⋮	⋮
0	1	1	0	1	1	1	1	after 111th increment	1 bit flipped
0	1	1	1	0	0	0	0	after 112th increment	5 bits flipped

Bit Counter

- For a sequence of m operations, the i'th bit is flipped $\frac{m}{2^i}$ times
- Total number of flips will be at most

$$\underbrace{m}_{flips\ \ of\ \ index\ \ 0} + \underbrace{\frac{m}{2}}_{flips\ \ of\ \ index\ \ 1} + \ldots + \underbrace{\frac{m}{2^{\lceil \log m \rceil}}}_{flips\ \ of\ \ index\ \ \lceil \log m \rceil} < m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m$$

		$\lceil \log m \rceil$		...	2	1	0		
0	0	0	0	0	0	0	0	initial configuration	
0	0	0	0	0	0	0	1	after 1st increment	1 bit flipped
0	0	0	0	0	0	1	0	after 2nd increment	2 bits flipped
			⋮					⋮	⋮
0	1	1	0	1	1	1	1	after 111th increment	1 bit flipped
0	1	1	1	0	0	0	0	after 112th increment	5 bits flipped

Bit Counter

- For a sequence of m operations, the i'th bit is flipped $\frac{m}{2^i}$ times
- Total number of flips will be at most

$$\underbrace{m}_{flips\ \ of\ \ index\ \ 0} + \underbrace{\frac{m}{2}}_{flips\ \ of\ \ index\ \ 1} + \ldots + \underbrace{\frac{m}{2^{\lceil \log m \rceil}}}_{flips\ \ of\ \ index\ \ \lceil \log m \rceil} < m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m$$

- **The amortized number of flips per operation is $2 = \Theta(1)$ flips**.

		$\lceil \log m \rceil$		...	2	1	0		
0	0	0	0	0	0	0	0	initial configuration	
0	0	0	0	0	0	0	1	after 1st increment	1 bit flipped
0	0	0	0	0	0	1	0	after 2nd increment	2 bits flipped
⋮								⋮	⋮
0	1	1	0	1	1	1	1	after 111th increment	1 bit flipped
0	1	1	1	0	0	0	0	after 112th increment	5 bits flipped

Bit Counter

- For a sequence of m operations, the i'th bit is flipped $\frac{m}{2^i}$ times
- Total number of flips will be at most

$$\underbrace{m}_{flips\ \ of\ \ index\ \ 0} + \underbrace{\frac{m}{2}}_{flips\ \ of\ \ index\ \ 1} + \ldots + \underbrace{\frac{m}{2^{\lceil \log m \rceil}}}_{flips\ \ of\ \ index\ \ \lceil \log m \rceil} < m \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m$$

- **The amortized number of flips per operation is $2 = \Theta(1)$ flips**.
- The worst case number of flips is $\Theta(\log m)$; but it never happens that a sequence of m operations have $m\Theta(\log m)$ flips!

		$\lceil \log m \rceil$		...	2	1	0		
0	0	0	0	0	0	0	0	initial configuration	
0	0	0	0	0	0	0	1	after 1st increment	1 bit flipped
0	0	0	0	0	0	1	0	after 2nd increment	2 bits flipped
			⋮					⋮	⋮
0	1	1	0	1	1	1	1	after 111th increment	1 bit flipped
0	1	1	1	0	0	0	0	after 112th increment	5 bits flipped

Amortized Analysis Review

- Considering a sequence of m **operations** for sufficiently large m:
  - Some operations are more 'expensive' and most are 'inexpensive'.
  - Amortized cost is the average cost over all operations
  - There is no probability distribution or randomness

Amortized Analysis Review

- Considering a sequence of m **operations** for sufficiently large m:
  - Some operations are more 'expensive' and most are 'inexpensive'.
  - Amortized cost is the average cost over all operations
  - There is no probability distribution or randomness
- We saw the amortized number of flips when incrementing a number m times is $\Theta(1)$
  - Some increment operation need $\Theta(\log m)$ flips while most operation take less flips.
  - On average, each operation needs $\Theta(1)$ flips.

Methods for Amortized Analysis

- There are three frameworks for amortized analysis.
- **Aggregate method**:
  - Sum the total cost of m operations
  - Divide by m to get the amortized cost
  - This is what we did for bit flips

Methods for Amortized Analysis

- There are three frameworks for amortized analysis.
- **Aggregate method**:
  - Sum the total cost of m operations
  - Divide by m to get the amortized cost
  - This is what we did for bit flips
- **Accounting method**
  - Analogy with a **bank account**, where there are **fixed deposits** and variable **withdrawals**

Methods for Amortized Analysis

- There are three frameworks for amortized analysis.
- **Aggregate method**:
  - Sum the total cost of m operations
  - Divide by m to get the amortized cost
  - This is what we did for bit flips
- **Accounting method**
  - Analogy with a **bank account**, where there are **fixed deposits** and variable **withdrawals**
- **Potential method**
  - Define amortized cost through **potential function** which maps the sequence of operations to an integer

Methods for Amortized Analysis

- There are three frameworks for amortized analysis.
- **Aggregate method**:
  - Sum the total cost of m operations
  - Divide by m to get the amortized cost
  - This is what we did for bit flips
- **Accounting method**
  - Analogy with a **bank account**, where there are **fixed deposits** and variable **withdrawals**
- **Potential method**
  - Define amortized cost through **potential function** which maps the sequence of operations to an integer
- Let's review these methods with an example!

Dynamic Arrays

- Problem: implement a stack stored in an array to support push (insert) operations.
- The problem is **online** in the sense that we do not know how many operations to expect

Dynamic Arrays

- Problem: implement a stack stored in an array to support push (insert) operations.
- The problem is **online** in the sense that we do not know how many operations to expect
- How large the array should be? there is a trade-off:
  - larger array: less likely to run out of space, more unused/wasted memory
  - smaller array: more likely to run out of space, less unused/wasted memory

Dynamic Arrays

- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array

i	operation

Dynamic Arrays

- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array

i	operation

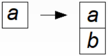
Dynamic Arrays

$\boxed{a}$

- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array

i	operation
1	insert(a)

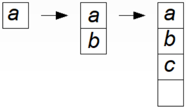
Dynamic Arrays



- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array

i	operation	
1	insert(a)	
2	insert(b)	no space: allocate array of size 2, copy 1 item

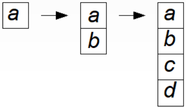
Dynamic Arrays



- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array

i	operation	
1	insert(a)	
2	insert(b)	no space: allocate array of size 2, copy 1 item
3	insert(c)	no space: allocate array of size 4, copy 2 item

Dynamic Arrays

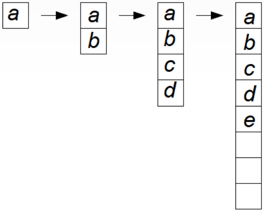


- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array

i	operation	
1	insert(a)	
2	insert(b)	no space: allocate array of size 2, copy 1 item
3	insert(c)	no space: allocate array of size 4, copy 2 item
4	insert(d)	

Dynamic Arrays

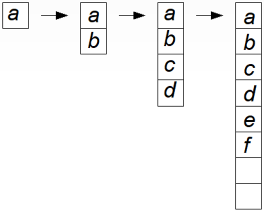
- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array



i	operation	
1	insert(a)	
2	insert(b)	no space: allocate array of size 2, copy 1 item
3	insert(c)	no space: allocate array of size 4, copy 2 item
4	insert(d)	
5	insert(e)	no space: allocate array of size 8, copy 4 item

Dynamic Arrays

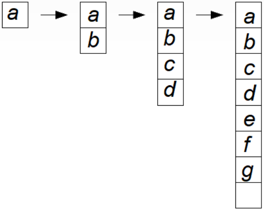
- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array



i	operation	
1	insert(a)	
2	insert(b)	no space: allocate array of size 2, copy 1 item
3	insert(c)	no space: allocate array of size 4, copy 2 item
4	insert(d)	
5	insert(e)	no space: allocate array of size 8, copy 4 item
6	insert(f)	

Dynamic Arrays

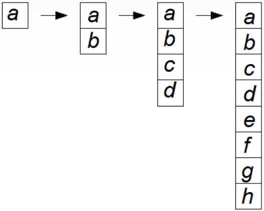
- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array



i	operation	
1	insert(a)	
2	insert(b)	no space: allocate array of size 2, copy 1 item
3	insert(c)	no space: allocate array of size 4, copy 2 item
4	insert(d)	
5	insert(e)	no space: allocate array of size 8, copy 4 item
6	insert(f)	
7	insert(g)	

Dynamic Arrays

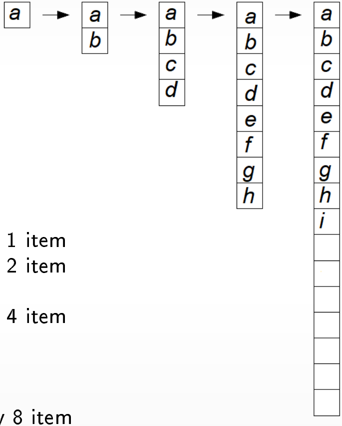
- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array



i	operation	
1	insert(a)	
2	insert(b)	no space: allocate array of size 2, copy 1 item
3	insert(c)	no space: allocate array of size 4, copy 2 item
4	insert(d)	
5	insert(e)	no space: allocate array of size 8, copy 4 item
6	insert(f)	
7	insert(g)	
8	insert(h)	

Dynamic Arrays

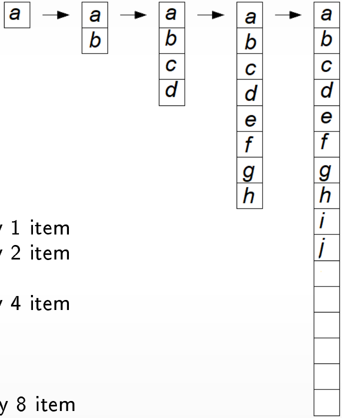
- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array



i	operation	
1	insert(a)	
2	insert(b)	no space: allocate array of size 2, copy 1 item
3	insert(c)	no space: allocate array of size 4, copy 2 item
4	insert(d)	
5	insert(e)	no space: allocate array of size 8, copy 4 item
6	insert(f)	
7	insert(g)	
8	insert(h)	
9	insert(i)	no space: allocate array of size 16, copy 8 item

Dynamic Arrays

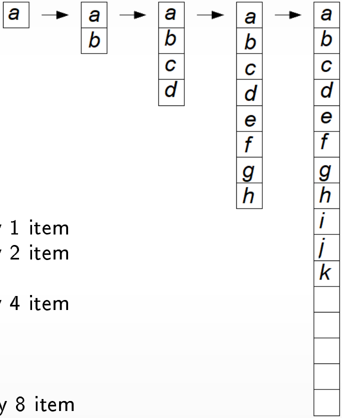
- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array



i	operation	
1	insert(a)	
2	insert(b)	no space: allocate array of size 2, copy 1 item
3	insert(c)	no space: allocate array of size 4, copy 2 item
4	insert(d)	
5	insert(e)	no space: allocate array of size 8, copy 4 item
6	insert(f)	
7	insert(g)	
8	insert(h)	
9	insert(i)	no space: allocate array of size 16, copy 8 item
10	insert(j)	

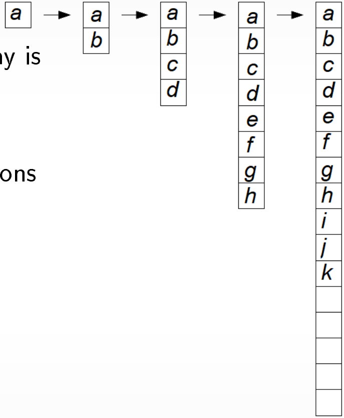
Dynamic Arrays

- Possible solution: maintain arrays with sizes that are powers of 2.
- If the array runs out of space ($n > a$):
  - allocate a new array of size a $2n$
  - copy all n items to the new array



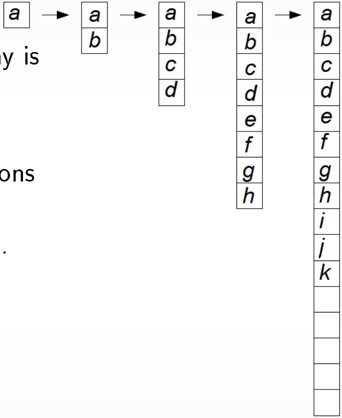
i	operation	
1	insert(a)	
2	insert(b)	no space: allocate array of size 2, copy 1 item
3	insert(c)	no space: allocate array of size 4, copy 2 item
4	insert(d)	
5	insert(e)	no space: allocate array of size 8, copy 4 item
6	insert(f)	
7	insert(g)	
8	insert(h)	
9	insert(i)	no space: allocate array of size 16, copy 8 item
10	insert(j)	
11	insert(k)	

Dynamic Arrays



- The worst-case cost occurs when the whole array is copied to a new array:
  - $\Theta(n)$ worst-case time per insert.
- Rough estimate: a sequence of m insert operations takes $O(m \cdot n)$ time.

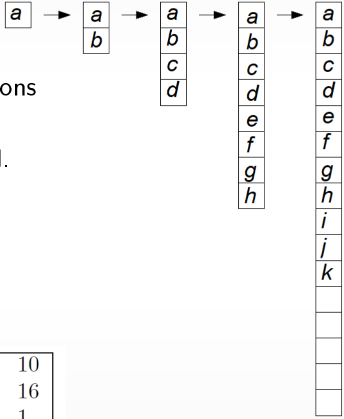
Dynamic Arrays



- The worst-case cost occurs when the whole array is copied to a new array:
  - $\Theta(n)$ worst-case time per insert.
- Rough estimate: a sequence of m insert operations takes $O(m \cdot n)$ time.
  - We can obtain a much better (smaller) bound.

Dynamic Arrays

- The worst-case cost occurs when the whole array is copied to a new array:
  - $\Theta(n)$ worst-case time per insert.
- Rough estimate: a sequence of m insert operations takes $O(m \cdot n)$ time.
  - We can obtain a much better (smaller) bound.

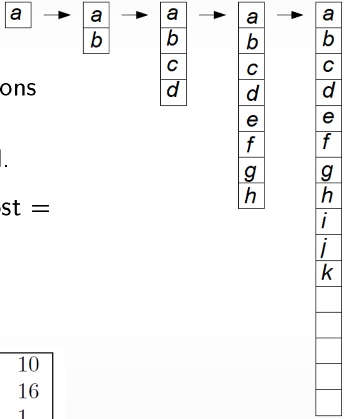


i	1	2	3	4	5	6	7	8	9	10
array size (a)	1	2	4	4	8	8	8	8	16	16
$c(i)$	**1**	**2**	**3**	1	**5**	1	1	1	**9**	1

Dynamic Arrays

- The worst-case cost occurs when the whole array is copied to a new array:
  - $\Theta(n)$ worst-case time per insert.
- Rough estimate: a sequence of m insert operations takes $O(m \cdot n)$ time.
  - We can obtain a much better (smaller) bound.
- Let $c(i)$ denote the cost of the ith insertion (cost = number of insert/copies).

$$c(i) = \begin{cases} i & \text{if } i = 2^k + 1 \text{ for some integer } k \\ 1 & \text{if otherwise} \end{cases}$$



i	1	2	3	4	5	6	7	8	9	10
array size (a)	1	2	4	4	8	8	8	8	16	16
$c(i)$	**1**	**2**	**3**	1	**5**	1	1	1	**9**	1

Aggregation for Dynamic Arrays

- Aggregate method: find total cost of m operations and divide by m

$$c(i) = \begin{cases} i & \text{if } i = 2^k + 1 \text{ for some integer } k \\ 1 & \text{if otherwise} \end{cases}$$

Aggregation for Dynamic Arrays

- Aggregate method: find total cost of m operations and divide by m

$$c(i) = \begin{cases} i & \text{if } i = 2^k + 1 \text{ for some integer } k \\ 1 & \text{if otherwise} \end{cases}$$

$$\begin{aligned}
\text{Cost of } m \text{ insertions} = \sum_{i=1}^{m} c(i) &\leq \underbrace{m}_{\text{insert new item}} + \underbrace{\sum_{j=0}^{\lfloor \log(m-1) \rfloor} 2^j}_{\text{copy old items to new array}} \\
&= m + 2^{\lfloor \log(m-1) \rfloor + 1} - 1 \\
&\leq m + 2^{\log m + 1} - 1 \\
&= m + 2m - 1 \\
&= 3m - 1 \\
&\in \Theta(m)
\end{aligned}$$

Aggregation for Dynamic Arrays

- Aggregate method: find total cost of m operations and divide by m

$$c(i) = \begin{cases} i & \text{if } i = 2^k + 1 \text{ for some integer } k \\ 1 & \text{if otherwise} \end{cases}$$

$$\begin{aligned}
\text{Cost of } m \text{ insertions} = \sum_{i=1}^{m} c(i) &\le \underbrace{m}_{\text{insert new item}} + \underbrace{\sum_{j=0}^{\lfloor \log(m-1) \rfloor} 2^j}_{\text{copy old items to new array}} \\
&= m + 2^{\lfloor \log(m-1) \rfloor + 1} - 1 \\
&\le m + 2^{\log m + 1} - 1 \\
&= m + 2m - 1 \\
&= 3m - 1 \\
&\in \Theta(m)
\end{aligned}$$

- The amortized cost is hence $\frac{\Theta(m)}{m} = \Theta(1)$

Aggregation for Dynamic Arrays

- Aggregate method: find total cost of m operations and divide by m

$$c(i) = \begin{cases} i & \text{if } i = 2^k + 1 \text{ for some integer } k \\ 1 & \text{if otherwise} \end{cases}$$

$$\begin{aligned}
\text{Cost of } m \text{ insertions} = \sum_{i=1}^{m} c(i) &\leq \underbrace{m}_{\text{insert new item}} + \underbrace{\sum_{j=0}^{\lfloor \log(m-1) \rfloor} 2^j}_{\text{copy old items to new array}} \\
&= m + 2^{\lfloor \log(m-1) \rfloor + 1} - 1 \\
&\leq m + 2^{\log m + 1} - 1 \\
&= m + 2m - 1 \\
&= 3m - 1 \\
&\in \Theta(m)
\end{aligned}$$

- The amortized cost is hence $\frac{\Theta(m)}{m} = \Theta(1)$
- The aggregate is useful for simple amortized analysis.
- Sometimes require a different technique to obtain amortized cost.

Accounting Method

- Assume you want to prove that your average (amortized) daily cost is no more than 100$.

Accounting Method

- Assume you want to prove that your average (amortized) daily cost is no more than 100$.
    - Some days you might spend much more but on average it is at most 100$

Accounting Method

- Assume you want to prove that your average (amortized) daily cost is no more than 100$.
    - Some days you might spend much more but on average it is at most 100$
- One way to do that is to assume every day 100$ is deposited into your account
- On days which you spend more than 100$, you should use accumulated credit from previous days

Accounting Method

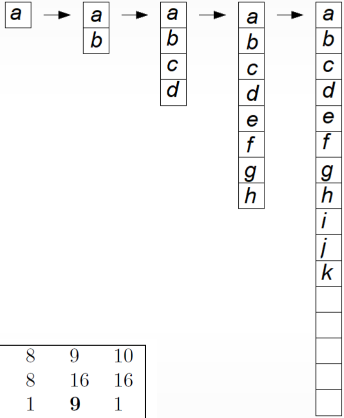
- Assume you want to prove that your average (amortized) daily cost is no more than 100$.
  - Some days you might spend much more but on average it is at most 100$
- One way to do that is to assume every day 100$ is deposited into your account
- On days which you spend more than 100$, you should use accumulated credit from previous days
- If your balance remains non-negative at the end of each day, your amortized cost is at most 100$
  - In m consecutive days your expenditure has been at most $100m \rightarrow$ amortized cost at most 100$.

Accounting Method

- Accounting method overview:
  - Each operation **deposits** a fixed credit into an account (This amount is an upper bound on the amortized cost.)
  - Each operation uses 'credit' to pay its **cost**
  - Inexpensive operations save more than their cost
  - Expensive operations cost more more than they save
  - Account must remain non-negative

Accounting Method - Dynamic Arrays

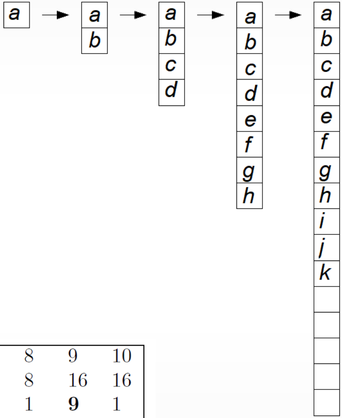
- We prove the amortized cost for insertion is 3



i	1	2	3	4	5	6	7	8	9	10
array size (a)	1	2	4	4	8	8	8	8	16	16
$c(i)$	**1**	**2**	**3**	1	**5**	1	1	1	**9**	1
total deposited	3	6	9	12	15	18	21	24	27	30
total spent	1	3	6	7	12	13	14	15	26	27
available credit	2	3	3	5	3	5	7	9	1	3

Accounting Method - Dynamic Arrays

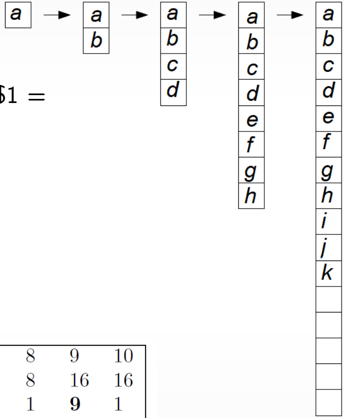
- We prove the amortized cost for insertion is 3
  - Each operation deposits \$3
  - Each write/move operation costs \$1



i	1	2	3	4	5	6	7	8	9	10
array size (a)	1	2	4	4	8	8	8	8	16	16
$c(i)$	**1**	**2**	**3**	1	**5**	1	1	1	**9**	1
total deposited	3	6	9	12	15	18	21	24	27	30
total spent	1	3	6	7	12	13	14	15	26	27
available credit	2	3	3	5	3	5	7	9	1	3

Accounting Method - Dynamic Arrays

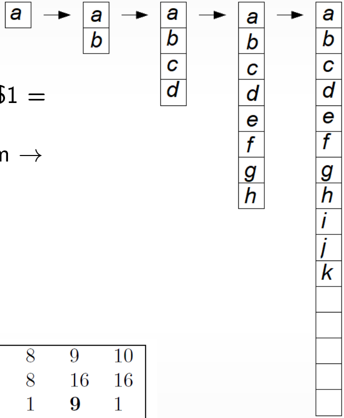
- We prove the amortized cost for insertion is 3
  - Each operation deposits \$3
  - Each write/move operation costs \$1
  - Inexpensive insertion deposits \$3 and spends \$1 = \$2 saved



i	1	2	3	4	5	6	7	8	9	10
array size (a)	1	2	4	4	8	8	8	8	16	16
$c(i)$	**1**	**2**	**3**	1	**5**	1	1	1	**9**	1
total deposited	3	6	9	12	15	18	21	24	27	30
total spent	1	3	6	7	12	13	14	15	26	27
available credit	2	3	3	5	3	5	7	9	1	3

Accounting Method - Dynamic Arrays

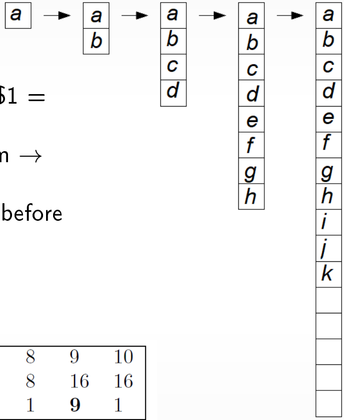
- We prove the amortized cost for insertion is 3
  - Each operation deposits \$3
  - Each write/move operation costs \$1
  - Inexpensive insertion deposits \$3 and spends \$1 = \$2 saved
  - Expensive insertion deposits \$3 and spends \$m → \$(m - 3) spent



i	1	2	3	4	5	6	7	8	9	10
array size (a)	1	2	4	4	8	8	8	8	16	16
$c(i)$	**1**	**2**	**3**	1	**5**	1	1	1	**9**	1
total deposited	3	6	9	12	15	18	21	24	27	30
total spent	1	3	6	7	12	13	14	15	26	27
available credit	2	3	3	5	3	5	7	9	1	3

Accounting Method - Dynamic Arrays

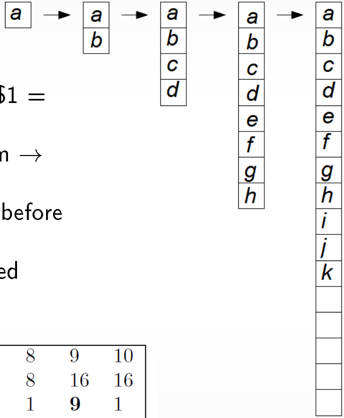
- We prove the amortized cost for insertion is 3
  - Each operation deposits \$3
  - Each write/move operation costs \$1
  - Inexpensive insertion deposits \$3 and spends \$1 = \$2 saved
  - Expensive insertion deposits \$3 and spends \$m $\rightarrow$ \$(m - 3) spent
  - Number of consecutive inexpensive insertions before expensive insertion: $(m-1)/2-1$



i	1	2	3	4	5	6	7	8	9	10
array size (a)	1	2	4	4	8	8	8	8	16	16
$c(i)$	**1**	**2**	**3**	1	**5**	1	1	1	**9**	1
total deposited	3	6	9	12	15	18	21	24	27	30
total spent	1	3	6	7	12	13	14	15	26	27
available credit	2	3	3	5	3	5	7	9	1	3

Accounting Method - Dynamic Arrays

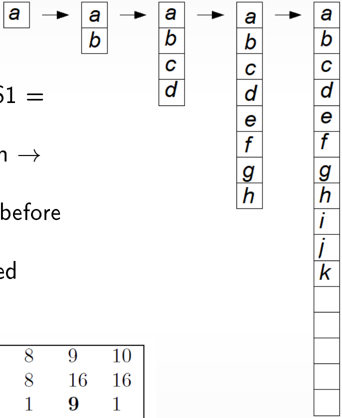
- We prove the amortized cost for insertion is 3
  - Each operation deposits \$3
  - Each write/move operation costs \$1
  - Inexpensive insertion deposits \$3 and spends \$1 = \$2 saved
  - Expensive insertion deposits \$3 and spends \$m → \$(m - 3) spent
  - Number of consecutive inexpensive insertions before expensive insertion: $(m-1)/2-1$
  - → $\$2((m-1)/2-1) = \$(m-3)$ accumulated credit since last expensive insertion



i	1	2	3	4	5	6	7	8	9	10
array size (a)	1	2	4	4	8	8	8	8	16	16
$c(i)$	**1**	**2**	**3**	1	**5**	1	1	1	**9**	1
total deposited	3	6	9	12	15	18	21	24	27	30
total spent	1	3	6	7	12	13	14	15	26	27
available credit	2	3	3	5	3	5	7	9	1	3

Accounting Method - Dynamic Arrays

- We prove the amortized cost for insertion is 3
  - Each operation deposits \$3
  - Each write/move operation costs \$1
  - Inexpensive insertion deposits \$3 and spends \$1 = \$2 saved
  - Expensive insertion deposits \$3 and spends \$m → \$(m - 3) spent
  - Number of consecutive inexpensive insertions before expensive insertion: $(m-1)/2-1$
  - → $\$2((m-1)/2-1) = \$(m-3)$ accumulated credit since last expensive insertion
  - → account remains non-negative



i	1	2	3	4	5	6	7	8	9	10
array size (a)	1	2	4	4	8	8	8	8	16	16
$c(i)$	**1**	**2**	**3**	1	**5**	1	1	1	**9**	1
total deposited	3	6	9	12	15	18	21	24	27	30
total spent	1	3	6	7	12	13	14	15	26	27
available credit	2	3	3	5	3	5	7	9	1	3

Potential method

- Define a potential function Φ that maps the state of the structure and the index of an operation to an integer
  - Potential is basically the available credit in accounting method

$$\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1)$$

  - $\hat{c}(i) \rightarrow$ amortized cost of operation i
  - $c(i) \rightarrow$ actual cost of operation i
- Total amortized cost will be total cost plus a constant independent of m.

Potential Method for Dynamic Arrays

- Define the potential to be $\Phi(i) = 2i - a_i$
- a_i denotes the size of the array after operation i

Potential Method for Dynamic Arrays

- Define the potential to be $\Phi(i) = 2i - a_i$
- a_i denotes the size of the array after operation i
- In case of an inexpensive operation, we have $c_i = 1$ and $a_i = a_{i-1}$; (the size of array does not change)
  - the amortized cost will be

$$\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1) = 1 + [2i - a_i] - [2(i-1) - a_{i-1}] = 3$$

Potential Method for Dynamic Arrays

- Define the potential to be $\Phi(i) = 2i - a_i$
- a_i denotes the size of the array after operation i
- In case of an inexpensive operation, we have $c_i = 1$ and $a_i = a_{i-1}$; (the size of array does not change)
  - the amortized cost will be

    $$\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1) = 1 + [2i - a_i] - [2(i-1) - a_{i-1}] = 3$$
- For expensive operation i, table size changes from $a_{i-1} = (i-1)$ to $a_i = 2(i-1)$ and we have $c_i = i$.
  - the amortized cost will be
    $\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1) = i + [2i - a_i] - [2(i-1) - a_{i-1}]$
    $= i + 2i - 2(i-1) - 2i + 2 + (i-1) = 3$

Potential Method for Dynamic Arrays

- Define the potential to be $\Phi(i) = 2i - a_i$
- a_i denotes the size of the array after operation i
- In case of an inexpensive operation, we have $c_i = 1$ and $a_i = a_{i-1}$; (the size of array does not change)
  - the amortized cost will be

    $$\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1) = 1 + [2i - a_i] - [2(i-1) - a_{i-1}] = 3$$
- For expensive operation i, table size changes from $a_{i-1} = (i-1)$ to $a_i = 2(i-1)$ and we have $c_i = i$.
  - the amortized cost will be
    $\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1) = i + [2i - a_i] - [2(i-1) - a_{i-1}]$
    $= i + 2i - 2(i-1) - 2i + 2 + (i-1) = 3$
- **Potential method is often the strongest method for amortized analysis**

Methods for Amortized Analysis

- There are three frameworks for amortized analysis.
- **Aggregate method**:
  - Sum the total cost of m operations
  - Divide by m to get the amortized cost
- **Accounting method**
  - Analogy with a **bank account**, where there are **fixed deposits** and variable withdrawals
- **Potential method**
  - Define amortized cost through **potential function** which maps the sequence of operations to an integer
- Let's review these methods with another example!

Special Stacks

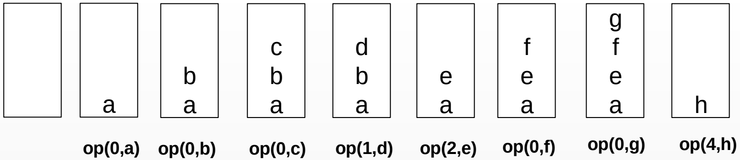
- Consider a stack with one operation $Op(n, x)$, where $n \geq 0$.

  $Op(n, x)$: pop n items from the stack and push x to it.

Special Stacks

- Consider a stack with one operation $Op(n, x)$, where $n \geq 0$.

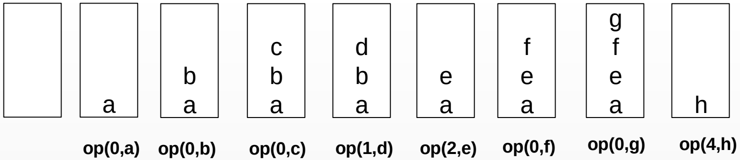
  $Op(n, x)$: pop n items from the stack and push x to it.



Special Stacks

- Consider a stack with one operation $Op(n, x)$, where $n \geq 0$.

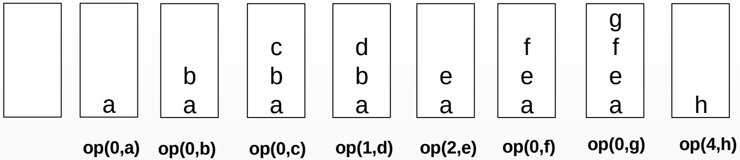
  $Op(n, x)$: pop n items from the stack and push x to it.
- What is the time complexity of each operation?
  - Assume each single push and pop has cost 1 (e.g., stack is implemented using a linked list).



Special Stacks

- Consider a stack with one operation $Op(n, x)$, where $n \geq 0$.

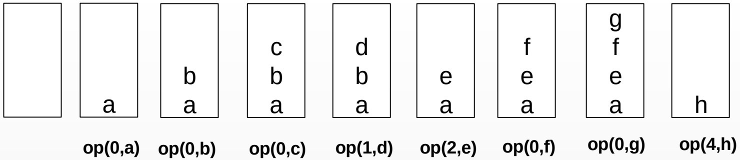
  $Op(n, x)$: pop n items from the stack and push x to it.
- What is the time complexity of each operation?
  - Assume each single push and pop has cost 1 (e.g., stack is implemented using a linked list).
- Assume $m - 1$ operations pop nothing and the m'th operation pops everything
  - A single operation can have a cost of $\Theta(m)$ in the worst case.



Special Stacks

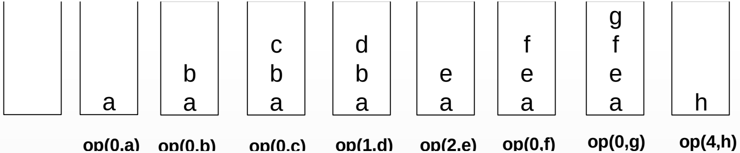
- Consider a stack with one operation $Op(n, x)$, where $n \geq 0$.

  $Op(n, x)$: pop n items from the stack and push x to it.
- What is the time complexity of each operation?
  - Assume each single push and pop has cost 1 (e.g., stack is implemented using a linked list).
- Assume $m - 1$ operations pop nothing and the m'th operation pops everything
  - A single operation can have a cost of $\Theta(m)$ in the worst case.
  - The amortized time is much better!



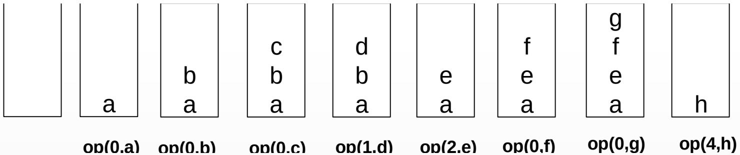
Aggregate Method for Special Stacks

- Review of aggregate method:
  - Sum the total cost of m consecutive operations
  - Divide by m to get the amortized cost



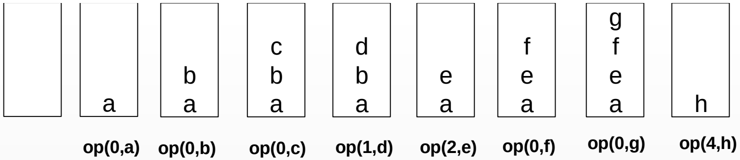
Aggregate Method for Special Stacks

- Review of aggregate method:
  - Sum the total cost of m consecutive operations
  - Divide by m to get the amortized cost
- Unlike bit flips and dynamic arrays, we cannot predict the cost of the i'th operation.
- The aggregate method is limited and cannot help for amortized analysis of special stacks!



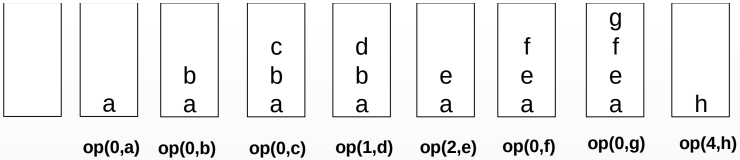
Accounting Method for Special Stacks

- Review of accounting method:
  - Each operations comes with a **fixed deposit** that is added to the **account** (defines the amortized cost).
  - For each operation, we subtract the cost of the operation from the account
    - Inexpensive operations contribute to the account
    - Expensive operations take away from the account
  - Iff the account is non-negative after each operation, the amortized cost is at most the fixed deposit.



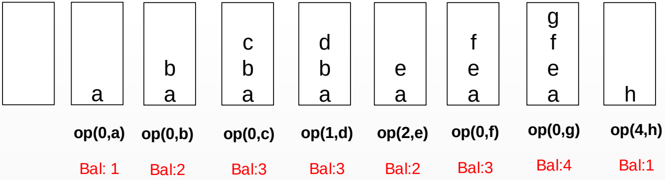
Accounting Method for Special Stacks

- Review of accounting method:
  - Each operations comes with a **fixed deposit** that is added to the **account** (defines the amortized cost).
  - For each operation, we subtract the cost of the operation from the account
    - Inexpensive operations contribute to the account
    - Expensive operations take away from the account
  - Iff the account is non-negative after each operation, the amortized cost is at most the fixed deposit.
- Often, the account can be imagined as sum of 'credits' assigned to different components of data structure



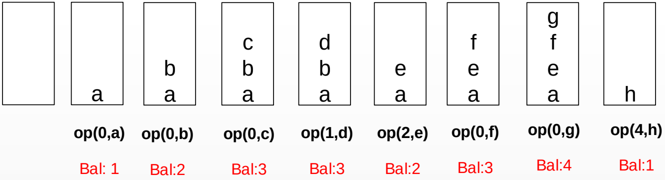
Accounting Method for Special Stacks

- We prove an amortized cost of 2 per operation $\rightarrow$ assume there is a fixed deposit of 2 per operation.



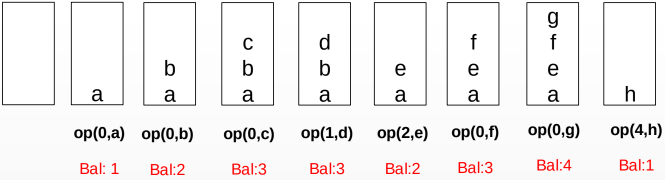
Accounting Method for Special Stacks

- We prove an amortized cost of 2 per operation → assume there is a fixed deposit of 2 per operation.
- Maintain this invariant: there is a credit of 1 for each item in the stack → account is the number of items in the stack.



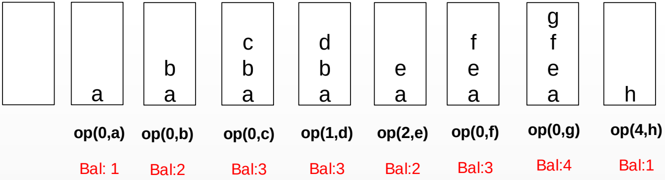
Accounting Method for Special Stacks

- We prove an amortized cost of 2 per operation $\rightarrow$ assume there is a fixed deposit of 2 per operation.
- Maintain this invariant: there is a credit of 1 for each item in the stack $\rightarrow$ account is the number of items in the stack.
- $OP(n, x)$ where $n \geq 0$:



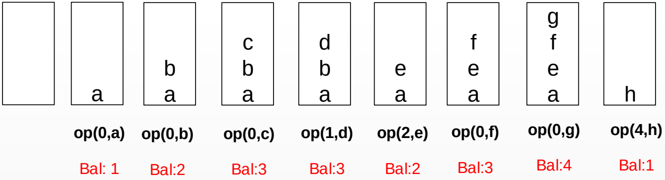
Accounting Method for Special Stacks

- We prove an amortized cost of 2 per operation $\rightarrow$ assume there is a fixed deposit of 2 per operation.
- Maintain this invariant: there is a credit of 1 for each item in the stack $\rightarrow$ account is the number of items in the stack.
- $OP(n, x)$ where $n \geq 0$:
  - Pop n items: there is a credit of 1 for each item that is popped; so the cost that the algorithm pays for pops is the same as the consumed credit $\rightarrow$ account remains positive



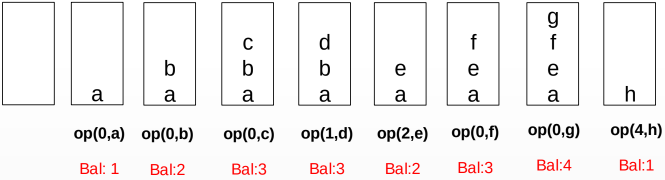
Accounting Method for Special Stacks

- We prove an amortized cost of 2 per operation $\rightarrow$ assume there is a fixed deposit of 2 per operation.
- Maintain this invariant: there is a credit of 1 for each item in the stack $\rightarrow$ account is the number of items in the stack.
- $OP(n, x)$ where $n \geq 0$:
  - Pop n items: there is a credit of 1 for each item that is popped; so the cost that the algorithm pays for pops is the same as the consumed credit $\rightarrow$ account remains positive
  - Push(x): there is a cost of 1 and fixed deposit of 2; the extra saving is stored as the credit for the item.



Accounting Method for Special Stacks

- With a fixed deposit of 2 per operation, we showed that the balance remains non-negative after each operation
- The balance was the accumulated credits stored in each item in the stack
- We conclude that the amortized cost of each operation is at most 2

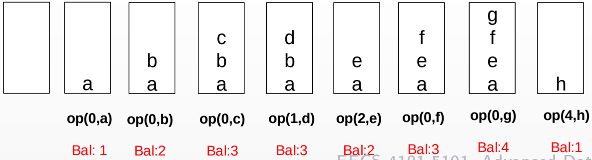


Potential Method for Special Stacks

- Review: Define a potential function $\phi(i)$ which maps the state of the structure after operation i to a positive number.
  - Potential is equivalent to the available credit after each operation in the accounting method.
- Amortized cost is the summation of actual cost and the difference in potential function:
  $\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1)$

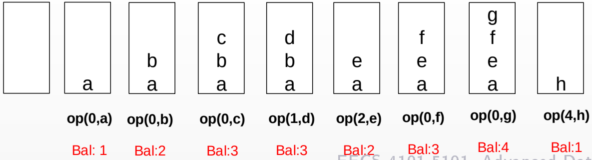
Potential Method for Special Stacks

- Review: Define a potential function $\phi(i)$ which maps the state of the structure after operation i to a positive number.
  - Potential is equivalent to the available credit after each operation in the accounting method.
- Amortized cost is the summation of actual cost and the difference in potential function:
  $\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1)$
- Define the potential to be the number of items in the stack



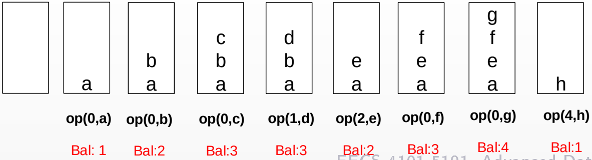
Potential Method for Special Stacks

- Review: Define a potential function $\phi(i)$ which maps the state of the structure after operation i to a positive number.
  - Potential is equivalent to the available credit after each operation in the accounting method.
- Amortized cost is the summation of actual cost and the difference in potential function:
  $\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1)$
- Define the potential to be the number of items in the stack
  - Assume operation i is $OP(n, x)$. The actual cost is $c(i) = n + 1$.



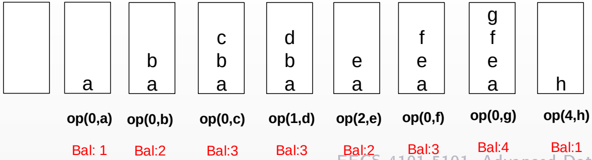
Potential Method for Special Stacks

- Review: Define a potential function $\phi(i)$ which maps the state of the structure after operation i to a positive number.
  - Potential is equivalent to the available credit after each operation in the accounting method.
- Amortized cost is the summation of actual cost and the difference in potential function:
  $\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1)$
- Define the potential to be the number of items in the stack
  - Assume operation i is $OP(n, x)$. The actual cost is $c(i) = n + 1$.
  - After the operation, the number of items is increased by $1 - n$, i.e., $\Phi(i) - \Phi(i-1) = 1 - n$.



Potential Method for Special Stacks

- Review: Define a potential function $\phi(i)$ which maps the state of the structure after operation i to a positive number.
  - Potential is equivalent to the available credit after each operation in the accounting method.
- Amortized cost is the summation of actual cost and the difference in potential function:
  $\hat{c}(i) = c(i) + \Phi(i) - \Phi(i-1)$
- Define the potential to be the number of items in the stack
  - Assume operation i is $OP(n, x)$. The actual cost is $c(i) = n + 1$.
  - After the operation, the number of items is increased by $1 - n$, i.e., $\Phi(i) - \Phi(i-1) = 1 - n$.
  - The amortized cost is $\hat{c}(i) = (n+1) + (1-n) = 2$.



More Examples of Amortized Analysis

- Fibonacci heaps: similar to binomial heaps except that they have a more 'relaxed' structure
  - Most operations can be done in constant time; for some operations, the heap should be restructured.
  - The amortized cost for Insert, ExtractMax, Merge, and IncreaseKey is $O(1)$ (champions for priority queues).

More Examples of Amortized Analysis

- Fibonacci heaps: similar to binomial heaps except that they have a more ‘relaxed’ structure
  - Most operations can be done in constant time; for some operations, the heap should be restructured.
  - The amortized cost for Insert, ExtractMax, Merge, and IncreaseKey is $O(1)$ (champions for priority queues).
- Dynamic lists and arrays
  - Update a self-adjusting linked list with Move-To-Front strategy: applications in data compression

More Examples of Amortized Analysis

- Fibonacci heaps: similar to binomial heaps except that they have a more ‘relaxed’ structure
  - Most operations can be done in constant time; for some operations, the heap should be restructured.
  - The amortized cost for Insert, ExtractMax, Merge, and IncreaseKey is $O(1)$ (champions for priority queues).
- Dynamic lists and arrays
  - Update a self-adjusting linked list with Move-To-Front strategy: applications in data compression
  - Splay trees: dynamic binary trees which move an accessed item closer to the root.
    - Ideal for real-world scenarios where there is **locality** in accesses
    - Dynamic optimality conjecture: the amortized cost of accessing an item in a splay tree is within a constant ratio of any other tree (a challenging open question).

More Examples of Amortized Analysis

- Fibonacci heaps: similar to binomial heaps except that they have a more 'relaxed' structure
  - Most operations can be done in constant time; for some operations, the heap should be restructured.
  - The amortized cost for Insert, ExtractMax, Merge, and IncreaseKey is $O(1)$ (champions for priority queues).
- Dynamic lists and arrays
  - Update a self-adjusting linked list with Move-To-Front strategy: applications in data compression
  - Splay trees: dynamic binary trees which move an accessed item closer to the root.
    - Ideal for real-world scenarios where there is **locality** in accesses
    - Dynamic optimality conjecture: the amortized cost of accessing an item in a splay tree is within a constant ratio of any other tree (a challenging open question).
- The whole field of online algorithms!